CS 415 Operating Systems

Project 1 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Luke Marshall
lmarshal
951612676

Report

Introduction

This project is to make an imitation shell which provides users with the basic functionality of a real shell. The running project imitates a command-line interface on the computer's actual shell by allowing the user to type a subset of the UNIX commands and have each of the commands result in similar behavior to their UNIX counterparts. The shell can either be run in interactive mode or file mode. In interactive mode the user sees a prompt and can type in a command, this command will then execute if it is supported by the shell. The user then will see another prompt at which time they can either input another command or terminate the program. In file mode the user gives a file to the program execution, which then parses the file, completes the commands within the file line by line, outputs the result if any to an output text file, and then terminates.

The shell provides error reporting to the user when they input commands not supported by the imitation shell, and when they provide incorrect or too many arguments to the commands. The commands themselves are taken in and executed using only basic C functionality, string functions, and system calls.

Background

Since the operating systems we are working with are mostly written in C, the language has a great built-in infrastructure around communicating with the OS, including being able to use system calls provided by the OS directly in its programs. This allows us to create a working shell by communicating directly with the OS instead of abstracting that direct contact away in library functions. The use of system calls also requires the handling of output differently as nothing is done for the caller. File descriptors are used instead of file streams, errors are reported differently, and if done wrong the results can be strange; all these things require their own way of being managed and everything must be done explicitly.

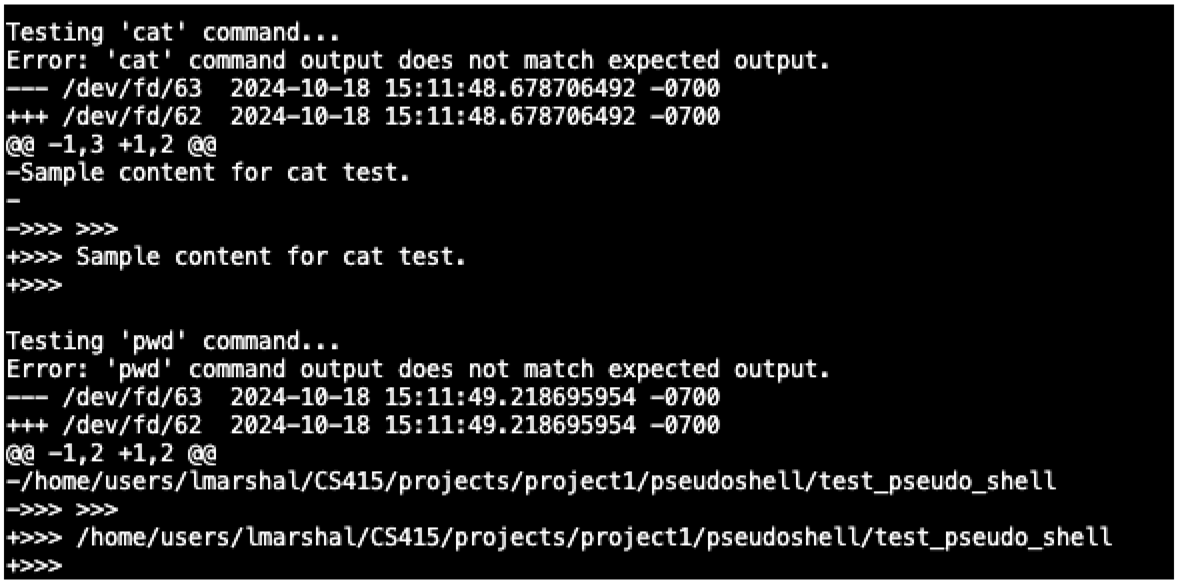
Implementation

I made the choice to separate out the two modes, interactive and file, of the project. The main function of the program deduces from the arguments of the program execution which mode the user wants to run the program in. Once this is settled, main calls the function corresponding to the mode. The difference in these functions comes down to either having to output the CLI and then accept input line-by-line, or if all the lines are available at once from the input file. Once the functions have a hold of the line they want to execute, though, the functionality is the same again so they call the same functions for tokenization and calling of functions that actually execute the commands with system calls.

Each of the commands we implemented has their own corresponding function. These are called when the corresponding command is entered in the command line, with the appropriate arguments. Each of the functions uses at least one system call to accomplish the command functionality, and most have more. I performed error-checking whenever possible, reporting any errors to the user, and then continuing on with execution if possible.

Performance Results and Discussion

The performance of the project is almost perfect. I handle the taking of input from either a fake command line or an input file if given, corresponding to user and file modes. I handle error reporting as prescribed in the instructions, reporting when an unsupported command is given, and when wrong, too many, or too few arguments are given to a command. But my output for the cat and pwd commands is slightly off according to the bash test script…



```
Testing 'cat' command...
Error: 'cat' command output does not match expected output.
--- /dev/fd/63  2024-10-18 15:11:48.678706492 -0700
+++ /dev/fd/62  2024-10-18 15:11:48.678706492 -0700
@@ -1,3 +1,2 @@
-Sample content for cat test.
-
->>> >>>
+>>> Sample content for cat test.
+>>>

Testing 'pwd' command...
Error: 'pwd' command output does not match expected output.
--- /dev/fd/63  2024-10-18 15:11:49.218695954 -0700
+++ /dev/fd/62  2024-10-18 15:11:49.218695954 -0700
@@ -1,2 +1,2 @@
-/home/users/lmarshal/CS415/projects/project1/pseudoshell/test_pseudo_shell
->>> >>>
+>>> /home/users/lmarshal/CS415/projects/project1/pseudoshell/test_pseudo_shell
+>>>
```

The diff between the output from my commands is almost the same i.e.

Expected: /dev/fd/63 2024-10-18 15:11:49.218695954 -0700
versus:
Actual: /dev/fd/62 2024-10-18 15:11:49.218695954 -0700

which only differs by a 63 vs a 62 in the lines and

Expected: /home/users/lmarshal/CS415/projects/project1/pseudoshell/test_pseudo_shell
versus:
Actual: /home/users/lmarshal/CS415/projects/project1/pseudoshell/test_pseudo_shell

Which collectively have the same things in them. I was told this difference has something to do with there being a root vs a home but I executed the test script on my Debian VM, Ubuntu ix-dev, and MacOS desktop all with the same results. I'm not sure the actual difference it is finding or how to remedy this.

Conclusion

I really enjoyed this project, especially implementing the command functions. I did learn a lot as well. There were a few big things like how the system calls are actually made with C and how the library functions are implemented to use them and manage the results, but it was mostly small things within C like getline automatically allocating for its buffer, redirecting stream outputs, and tokenization.