CS 415 Operating Systems
Fall 2025

Project #1 Report Collection

Author:

Rayna Patel

raynap

952017511

Report

Introduction

The goal of this project is to build a pseudo shell: a simplified, single-threaded UNIX command-line interface. The function of a shell is to provide an interface that interprets the user's commands in order to perform operating system functions. The core functionalities that I implemented from scratch include limited, but fundamental functionality such as file system navigation (ls, pwd, cd), file manipulation (cp, mv, rm, mkdir), and file I/O (cat).

Within this project, there were two required modes of operation: interactive mode and file mode. Interactive mode works by running ./psuedo-shell and interpreting commands that the user writes into the shell itself. The file mode works by running ./psuedo-shell -f <filename> to take in an input file and run a batch of commands specified in that input file, writing the output to a file named "output.txt".

Background

This project required a deep understanding of Linux system calls which are the primary method for a user program (such as the shell) to request services from the operating system kernel. This particular project required the use of low-level system calls instead of standard C library functions. To assist with file system navigation, I used the system calls opendir, readdir, and closedir. To assist with File I/O, I used the system calls open, read, and write. To assist with File/Directory management, I used mkdir, chdir, getcwd, and unlink.

In order to parse the input from the user to use in system calls, I had to use functions provided in string_parser.h to read the input line-by-line and tokenize the input. I first split the input by the semi-colon delimiter to get individual commands. I then split each command by spaces to separate the command from its respective parameters. Using this approach alongside an understanding of system calls allowed me the background necessary to begin implementing the pseudo-shell.

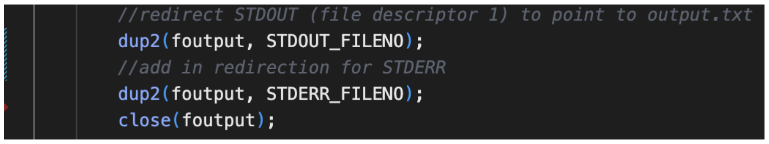
Implementation

The main.c file contains the primary logic for the shell and acts as the "brain" of the implementation. It begins by checking the argc parameter to determine which mode to enter into the shell. If no arguments are given, the shell enters into interactive mode. In interactive mode, the shell runs an infinite loop that prints the ">>>" prompt, reads a line from stdin using getline(), processes the commands by looking into command.c, and ends by freeing the allocated memory. This loop continues until the user types the "exit" command. In file mode, the shell reads commands line-by-line from the specified file and redirects all outputs to the "output.txt" file.

The strcmp() calls are used to match the command string to the corresponding function in command.c. The command.c file is the primary mapping for the shell to take the user's commands and find the appropriate system calls for the command. In main.c, error handling is used to make sure that each command has the correct number of parameters before finding the mappings in command.c. If the number of commands was not correct, an error message was printed to the user. A critical part of the implementation was memory management and ensuring no memory leaks using the free_command_line() function. I ensured this method worked by checking the shell implementation using valgrind.

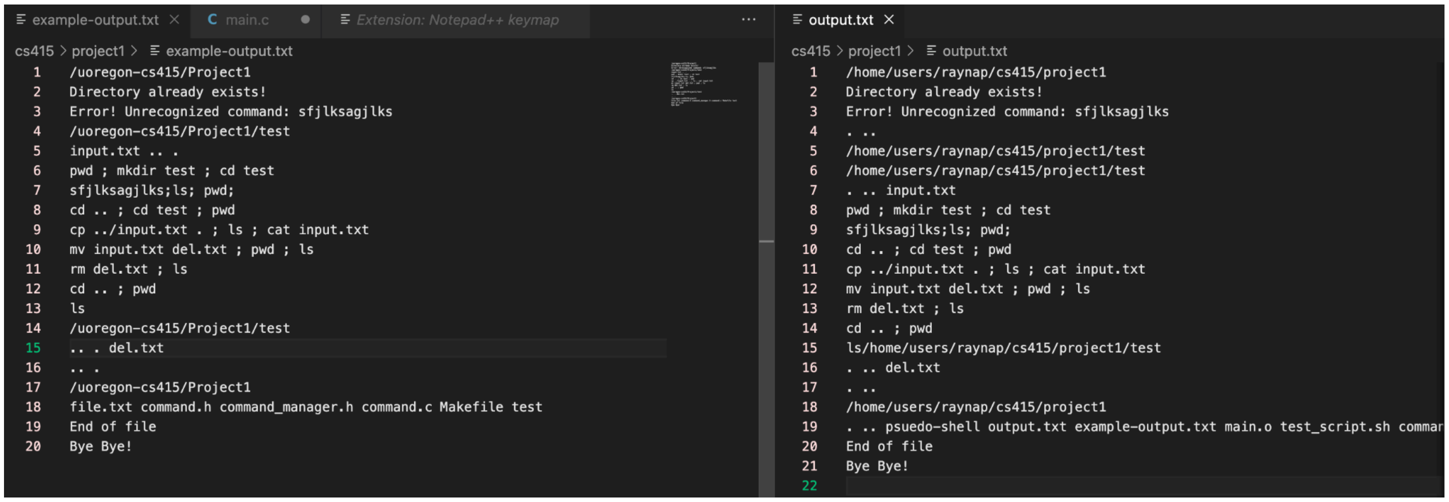
Performance Results and Discussion

The pseudo-shell was tested for correctness, compliance with project instructions, and memory safety using the provided test_scipt.sh and input.txt files. A key functionality of the shell is the ability to run in file mode and read commands from a batch file. The shell was executed with the provided test file using "./psuedo-shell -f input.txt". At first, this command printed some outputs to the output.txt file and others directly to the terminal. Upon further inspection, I realized it was an error with main.c. I was previously routing just outputs from STDOUT to the output.txt file using dup2(). I then modified this section to use dup2() to also route STDERR to the output.txt file rather than into the terminal.



```
//redirect STDOUT (file descriptor 1) to point to output.txt
dup2(foutput, STDOUT_FILENO);
//add in redirection for STDERR
dup2(foutput, STDERR_FILENO);
close(foutput);
```

This fix correctly generated the required output.txt file (upon cross-reference with example_output.txt) and demonstrated successful implementation of multi-command parsing, file I/O and manipulation, error handling, and navigation. There are slight discrepancies between the two outputs (such as new lines and some ordering, but I think this was explained to be okay in the project description). The shell was also tested in the interactive mode with every command. This included providing different numbers of arguments to see successful commands and to test the error handling. In addition, valgrind was used to detect zero memory leaks and zero errors. This showed that all dynamic memory allocation was correctly managed and deallocated using free_command_line().


example-output.txt
main.c
Extension: Notepad++ keymap
cs415 > project1 > example-output.txt
output.txt
cs415 > project1 > output.txt

```
/uoregon-cs415/Project1
Directory already exists!
Error! Unrecognized command: sfjlksagjlks
/uoregon-cs415/Project1/test
input.txt .. .
pwd ; mkdir test ; cd test
sfjlksagjlks;ls; pwd;
cd .. ; cd test ; pwd
cp ../input.txt . ; ls ; cat input.txt
mv input.txt del.txt ; pwd ; ls
rm del.txt ; ls
cd .. ; pwd
ls
/uoregon-cs415/Project1/test
.. . del.txt
.. .
/uoregon-cs415/Project1
file.txt command.h command_manager.h command.c Makefile test
End of file
Bye Bye!
```

```
/home/users/raynap/cs415/project1
Directory already exists!
Error! Unrecognized command: sfjlksagjlks
. ..
/home/users/raynap/cs415/project1/test
/home/users/raynap/cs415/project1/test
. .. input.txt
pwd ; mkdir test ; cd test
sfjlksagjlks;ls; pwd;
cd .. ; cd test ; pwd
cp ../input.txt . ; ls ; cat input.txt
mv input.txt del.txt ; pwd ; ls
rm del.txt ; ls
cd .. ; pwd
ls/home/users/raynap/cs415/project1/test
. .. del.txt
. ..
/home/users/raynap/cs415/project1
. .. psuedo-shell output.txt example-output.txt main.o test_script.sh commar
End of file
Bye Bye!

```

Conclusion

My main takeaways included a very practical understanding of the interface between a user-space program and the operating system. Being required to use read, write, open, and mkdir directly was new to me as I was brought to using low-level system calls rather than standard library functions. It was also an adjustment for me to move away from printf to write as I am used to using printf in past coding.

In addition, I got more practice in working with complex inputs and managing that through multiple delimiters. I also got more practice in ensuring that every malloc() was paired with a free() to make sure that there were no memory leaks or errors. Finally, using valgrind was another helpful refresher in debugging memory leaks and ensuring that I created a stable program. Overall, this project helped me increase my foundational understanding of how a shell operates and interacts directly with the operating system kernel.