CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Vikram Thirumaran

UO ID - vikram

952046110

Report

Introduction

This project, developed for CIS 415 Operating Systems, focused on building a pseudo-shell that simulates the core functionalities of a Unix-like shell, and allows the user to execute Linux commands. Users can input commands that the shell interprets and executes, with support for common features like piping, I/O redirection, and built-in command parsing. The project served to deepen our understanding of systems programming and operating system principles such as process creation, inter-process communication, and file descriptor manipulation.

The shell has two modes, a file mode that reads all commands from an input file and an interactive mode that lets the user dynamically execute commands one after another until they exit the shell. This program has 3 components that work together: a main function that loops through input and directs the flow of the program(main.c), a parser function called by main to tokenize the input(string_parser.c), and lastly, the implementations of each Linux command using only system calls called by main after input has been parsed(command.c). This project also implemented Error checking for invalid files, incorrect number of inputs as well as invalid commands.

Background

Many of the simpler Linux commands, such as `mkdir` and `rm`, had straightforward implementations due to the direct availability of corresponding system calls. The `mkdir` system call, for instance, provides the core functionality required for the `mkdir` command, with only minimal additions needed for error checking. As a result, commands like `mkdir`, `rm`, and `cd` essentially act as wrappers around the `mkdir`, `remove`, and `chdir` system calls, respectively.

Commands involving reading from or writing to files and the console followed a similar pattern. The `listDir` function, for example, uses `opendir` and `readdir` to iterate through directory contents and then writes each entry to the console. Similarly, `displayFile` reads a file's contents using the open and read system calls and displays them.

The `copyFile` function proved to be the most challenging, as it needed to handle both file-to-file and file-to-directory copying. This required using the `stat` function to determine if the destination path was a directory. If so, the path had to be modified to include the original file name. Once `copyFile` was implemented, the `moveFile` function was straightforward to build—it simply calls `copyFile` and then deletes the original file using remove.

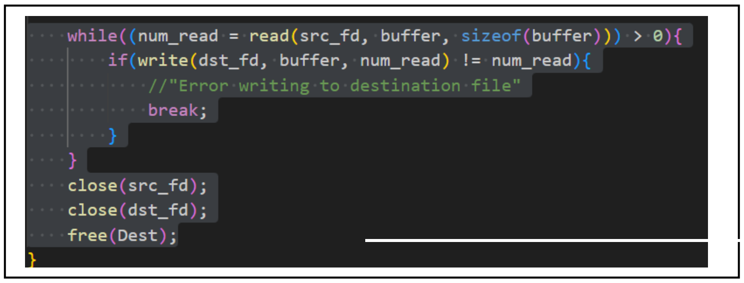
Implementation

The shell was implemented in C and organized into several modular components for clarity and maintainability. The `main.c` file handles the core user interface and input loop, while `command.c` and `command.h` are responsible for interpreting and executing both built-in and external commands. Tokenization of user input is handled in `string_parser.c` and `string_parser.h`, which break down raw command strings into manageable arguments. Compilation is streamlined through a `Makefile`, which ensures all source files are compiled correctly and linked into a single executable.

The shell supports a subset of common built-in commands such as `cd, pwd, and ls`, as well as arbitrary external commands available in the system's environment. One of the more consistent challenges was managing memory allocation and deallocation—especially with dynamically allocated token arrays—as well as writing robust error handling for edge cases and failed system calls.

A specific issue I encountered arose during the implementation of the cp command, housed in the `copyFile` function within `command.c`. During testing, I noticed memory leaks occurring when executing the test script, which caused significant concern as I was initially hesitant to add extensive error checking. Upon closer inspection, the leaks were isolated to tests involving the `cp` and `mv` commands. Since my `mv` implementation was essentially a wrapper around `cp` that removed the original file afterward, I was able to narrow down the issue quickly. The root of the problem was a lack of proper memory cleanup inside a loop where I used `malloc()`—specifically, I had neglected to free allocated memory after use.

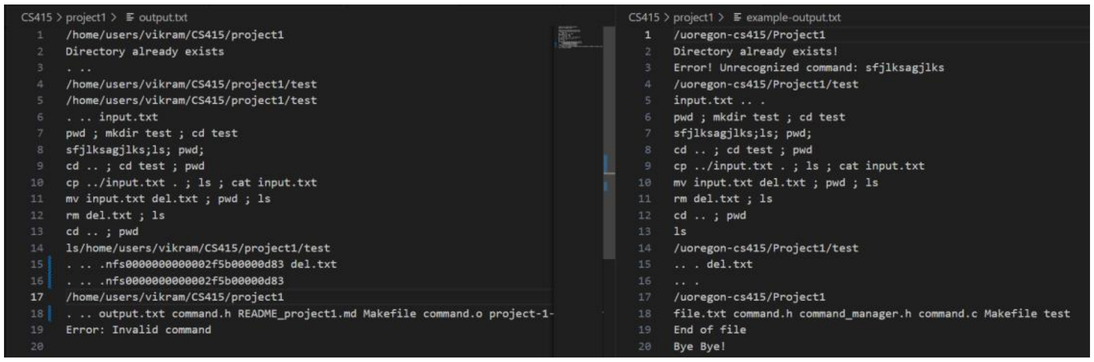
This debugging process taught me a valuable lesson: rather than rushing to patch bugs or apply temporary fixes, taking the time to step back and assess the broader context of a problem can often reveal the most effective and elegant solution. This experience reinforced the importance of deliberate, thoughtful debugging and solidified my understanding of memory management in C.



```c
    while((num_read = read(src_fd, buffer, sizeof(buffer))) > 0){
        if(write(dst_fd, buffer, num_read) != num_read){
            //"Error writing to destination file"
            break;
        }
    }
    close(src_fd);
    close(dst_fd);
    free(Dest);
}
```

Performance Results and Discussion

The output is not an exact match, (maybe due to my use of write instead of print? But the test script runs without any problems, and the invalid commands listed here still give corresponding error messages. Additionally, the make script runs smoothly, and make clean is a neat command as well, squaring away any binaries/executables in an orderly manner.


CS415 > project1 > output.txt
CS415 > project1 > example-output.txt

```
/home/users/vikram/CS415/project1
Directory already exists
. ..
/home/users/vikram/CS415/project1/test
/home/users/vikram/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/vikram/CS415/project1/test
. .. .nfs000000000002f5b00000d83 del.txt
. .. .nfs000000000002f5b00000d83
/home/users/vikram/CS415/project1
. .. output.txt command.h README_project1.md Makefile command.o project-1-
Error: Invalid command

```

```
/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!
```

Conclusion

This project provided a valuable opportunity to engage directly with operating system APIs and gain a deeper understanding of how shells operate at a low level. It offered hands-on experience with process management and inter-process communication (IPC) in C, and revealed the many intricacies involved in shell development. Looking ahead, potential enhancements could include adding support for background processes, maintaining command history, and implementing more robust parsing for quotes and escape characters.

Prior to this project, I often underestimated the importance of thorough error handling. Even in a relatively simple program like a shell, users can introduce a wide variety of unexpected inputs, and the system must be resilient enough to handle each edge case gracefully. One of the most interesting takeaways was seeing how low-level system calls interact with standard C library functions. For example, I initially encountered incorrect output when using `printf` alongside `write`. Because `write` executes immediately and `printf` buffers its output, `write` would often print first, even if `printf` was called beforehand. To resolve this, I replaced most `printf` statements with `write` to ensure consistent output behavior.