# CIS 415 Operating Systems

## Project 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

*Melanie Wiegand*

*mwiegan6*

*951976356*

# Report

## Introduction

*This project involved constructing a pseudo-shell with various functionalities that mimic the standard Linux command inputs; specifically, the **ls, pwd, mkdir, cd, cp, mv, rm**, and **cat** commands. The shell has two modes of operation: interactive mode and file mode. In interactive mode, the user inputs commands line by line (or separated by the "**;**" delimiter) and the shell produces the appropriate output, writing directly to the terminal. The user can quit the program at any time by typing the **exit** command. In file mode, the user calls the script on a specific input file containing commands, and the shell performs each action sequentially, writing the outputs to a new file, "output.txt". The shell also displays appropriate error messages for invalid inputs, for example, an unrecognized command or an incorrect number of parameters.*

*The key aspects of this project were tokenizing and parsing the string input, implementing an interactive command loop for interactive mode, finding system-call alternatives for implementing the standard Linux commands, supporting file reading/writing in file mode, avoiding memory leaks, and ensuring all errors and invalid commands were handled gracefully.*

## Background

*While much of this project built on the skills that I learned in CS330, a lot of it was also new to me. I was relatively unfamiliar with using system calls before working on this project. I made use of the Linux manual page (https://man7.org/linux/man-pages/man2/syscalls.2.html) for finding acceptable system calls to imitate the standard Linux commands that I was implementing. The most important part of the project for me was making sure that the string parsing functions were working correctly for all of the test cases and formats. This was crucial for correctly reading the input commands and deciding which functions to execute subsequently. I had the string_parser.c file from Lab 1 which was very helpful in this regard. I also appreciated that the Lab3 Makefile was generalizable for my project; I didn't have to change very much to get it to work the way I wanted.*

*One thing that I did to streamline my process a little bit was to create a function **writeToOutput(char* msg)** that saved me some time with all the write commands in the command.c file, so I didn't have to manually type all the parameters every time. I also made use of a Github repository to easily transfer files from my computer to my ix-dev account. This allowed me to work primarily in VSCode (my preferred coding interface) while being able to easily test the program in the Ubuntu terminal. It also helped me a lot with version control, since I could go back and see exactly which changes I made that fixed or caused errors.*

## Implementation

*I found the implementation of the commands themselves to be relatively straightforward, as most of them have corresponding functions that essentially perform the task directly e.g. chdir() for **cd** and unlink() for **rm**. The commands I struggled with the most were the **cp** and **mv** commands, since they both involved a lot of opening/closing files and had separate cases for copying to a directory vs. a file. Likewise, the command loop of my main function was fairly easy to write since it was very formulaic for each of the input cases.*

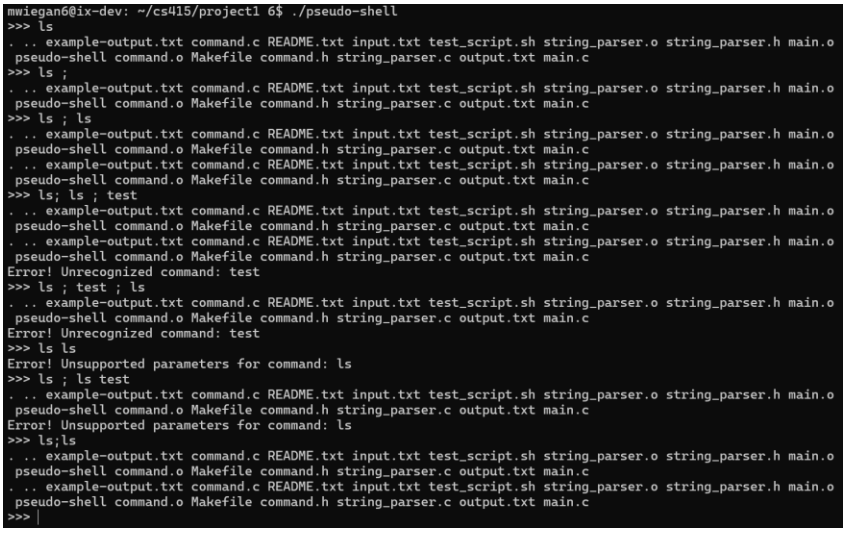
*One of the most difficult things for me conceptually was figuring out how to generalize the output to work with both the terminal (for interactive mode) and the output.txt file (for file mode). I originally tried using a global variable that represented the desired output location, but I ran into issues with declaring it in multiple places and not being able to modify the command.h header file. In office hours, Grace showed me a much easier way to do it by using the freopen command in my main function to redirect the stdout to the desired file:*

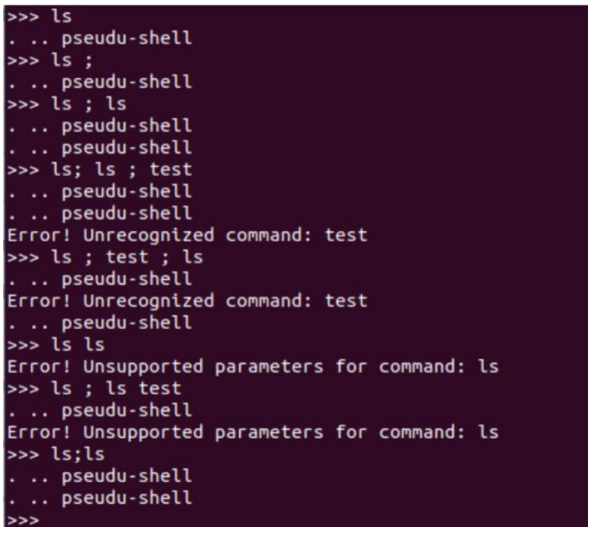
```
// set output to output.txt, or create
new file if it doesn't yet exist

FILE output = freopen("output.txt", "w+",
stdout);
if  (!output)
{
    perror("Error opening output.txt");
    fclose(input);
    return 1;
}
```

*I also had some issues with memory leaks that were tricky to figure out, but Valgrind was very helpful. Most of the leaks were from not freeing my allocated memory if the program ended early (like if a file was unreachable or the user exited the program). I was able to prevent all major leaks, though I had some memory blocks that were classified as "still reachable" by Valgrind. I was told that I would not be marked down for this.*

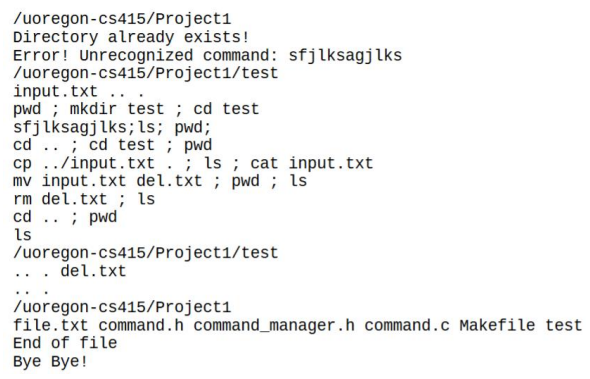
```
All tests completed.
==3857572==
==3857572== HEAP SUMMARY:
==3857572==     in use at exit: 319,550 bytes in 7,479 blocks
==3857572==   total heap usage: 31,696 allocs, 24,217 frees, 1,706,595 bytes allocated
==3857572==
==3857572== LEAK SUMMARY:
==3857572==    definitely lost: 0 bytes in 0 blocks
==3857572==    indirectly lost: 0 bytes in 0 blocks
==3857572==      possibly lost: 0 bytes in 0 blocks
==3857572==    still reachable: 319,550 bytes in 7,479 blocks
==3857572==         suppressed: 0 bytes in 0 blocks
==3857572== Rerun with --leak-check=full to see details of leaked memory
==3857572==
==3857572== For lists of detected and suppressed errors, rerun with: -s
==3857572== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## Performance Results and Discussion




*My program's behavior in interactive mode compared with the expected behavior from the project description. My script is able to correctly handle multiple commands separated by semicolons and produce appropriate error messages for invalid commands. There is one difference: for the* ***ls ; test ; ls*** *command, my program flags the "test" as invalid and skips the rest of the line. Contrarily, the example screenshot shows the program running the final* ***ls*** *command after giving the error message for "test." I debated whether to implement this behavior, but I believe the way I have it now is more accurate to the verbal description of the requirements—the project description says, "If an error is encountered you can skip processing the rest of the line." The screenshot below of the "example_output.txt" file supports this idea as well.*


*My output from calling* ***./pseudo-shell -f input.txt*** *compared with the example output. Aside from some minor distinctions (differently named paths, extra files in my base directory, different error messages, etc.), the components are all the same, indicating that my program runs as expected.*

## Conclusion

*This project taught me a lot about using system calls to program as opposed to the more standard ways of implementing functionality. It was definitely challenging to build the program without using standard functions like fopen and printf. There were a few times when I had to go back and change some lines where I used functions*

*that weren't system calls out of habit. It was nice to learn about these more rudimentary calls that can be used to mimic the behavior of the functions that I am familiar with.*

*I also feel that I got a lot more experience with using the terminal to run and test my programs. It has been a full year since I took CS330 so I felt a little rusty with the commands at first, but I found that the knowledge came back pretty quickly. I feel much more confident with using ssh and Github to easily communicate between my ix-dev and my computer, and I got more practice with checking for memory leaks and errors using Valgrind. I also got much more comfortable with the syntax for creating a Makefile, as well as using separate .c and .h files for cleaner implementation.*