CIS 415 Operating Systems

Project <1> Report Collection

Submitted to:

Prof. Allen Malony

Author:

Sophia Zhang

<951870312 - sophiaz>

<sophiaz>

Report

Introduction

The goal of this project was to implement a pseudo shell capable of handling basic UNIX system commands. I had to implement two modes: file mode and interactive mode. In file mode, users could input a `-f` *flag accompanied by an input file. This file contained commands that would be executed, with the results written to an* `output.txt` *file. In interactive mode, users experienced an environment similar to a shell, albeit with limited functionalities. The available commands available in this pseudo shell were* `ls`, `pwd`, `mkdir` , `rm` , `cat`, `cd`, `cp`, *and* `mv`. *Another feature included putting multiple commands in a single input separated by the deliminator* `;` , *this will parse through the entire input and read each individual command.*

Background

The main methods I used to implement the pseudo shell involved handling system calls, string parsing, and command execution. A key method I implemented was `getline()`, a function from the standard C library. It offered flexibility in handling input, allowing for easy manipulation and error handling. This function reads input from the user or from a file when in file mode (triggered by the `-f` flag).

Another crucial helper function I developed was `string_filler()`. This string tokenization function splits the input string based on a chosen delimiter. It proved particularly useful in processing multiple commands. By tokenizing the general input once and then again, I could isolate individual commands for execution. This approach made building the flow and error handling much easier. This function `string_filler()` was implemented together during class lab. I found this extremely helpful in getting started on the pseudo shell implementation process.

One resource that I found particularly helpful was the project 1 PDF and Linux man page. The project provided a few hints and explanation on how to get started. And the man page, mentioned several times during lab, was very helpful in understanding each system call and its use cases.

Implementation

I began the project by setting up a continuous while loop that would continue to run until the user inputted exit. Otherwise, it would read commands inputted by the user and write the expected result. I used the tokenization function to help tokenize the commands to a conditional that determined which command to execute. Beyond that was each command calling a function accordingly in `command.c` , where all the system calls were implemented.

One step I particularly struggled with was implementing file mode. I was confused about how to run all these commands and output them to a file without actually writing them to the command line. This step took me a while to fully understand and implement. I first opened the two files, `input.txt` and `output.txt`, to prepare for reading and writing. Initially, I tried to write to the command line and then write the result back to the file. This led to some segmentation faults and dead ends. Then, I discovered that `freopen()` was an option that allowed all `stdout` to be redirected to a specific file. This was particularly helpful in the file mode implementation.

```
if(argc >= 3 && strcmp(argv[1], "-f") == 0){
            lineSize = getline(&input, &storedInput, fd);
            if(lineSize == -1) {
                free(input);
                break;
            }
        } else {
            write(STDOUT_FILENO, ">>> ", strlen(">>> "));
            lineSize = getline(&input, &storedInput, stdin);
            if(strcmp(input, "exit\n") == 0){
                free(input);
                break;
            }
        }
```

Performance Results and Discussion

I believe my code meets all the criteria as it passes all the test cases with no memory leaks. Although, I think I can include some more error handling inside the `command.c` *file. This would include looking more into the error handling the specific system call would provide rather than manually using a conditional to check if it has all the correct parameters or* `NULL` *information.*

Conclusion

This project developed my understanding of system calls and the command line. I found this project interesting to implement and seeing how the actual shell gets handled in the background is very knowledge able. Overall, I believe my project meets all the requirement.