CIS 415 Operating Systems

Project <1> Report Collection

Submitted to:

Prof. Allen Malony

Author:

<Zachary Weisenbloom >

<UO ID - zweisenb>

<951911117 >

Report

Introduction

This project implements a basic shell that allows the user to execute Linux commands. All Linux commands were implemented using system calls such as write, open, close etc. 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 different components, a main function that loops through input and directs the flow of the program, a parser function called by main to tokenize the input, and lastly, the implementations of each Linux command using only system calls called by main after input has been parsed. This project also implemented Error checking for invalid files, incorrect number of inputs as well as invalid commands.

Background

Many of the more simple Linux commands such as mkdir and rm there where only one way to implement these methods. The "mkdir" system call has the exact functionality needed for the Linux command except for error checking so these commands were essentially rappers on these system calls. The "remove" and the "chdir" system calls provide for similarly simple solutions for the "rm" and "cd" commands. Commands that required reading and writing to and from a console or file also had similar implementations. The listDir function for example uses opendir and readdir to open and read the contents then looping through and write the contents. displayFile has similar logic but uses "open" and "read" system calls instead. copy File was the most difficult function because it needed to account for both copying to a directory and copying to a file. This requires the use of the "stat" function to check if the path is a directory and if it is the destination path then needs to be modified to name the file the same as the origin file. After copyFile was implemented moveFile is simple since it just needs to call copy file and then remove the origin file.

Implementation

As described above the implementation of the project had 3 components. The first component was the main function that read user input. The number of arguments are checked to determine whether the user should use file

```
if (stat(destinationPath, &statbuf) == 0 && S_ISDIR(statbuf.st_mode)) {
        Dest = (char*)malloc(sizeof(char)*(strlen(sourcePath) + strlen(destinationPath) + 2));
        strcpy(Dest, destinationPath);
        strcat(Dest, "/");
        strcat(Dest, basename(sourcePath));
        Dest[strlen(sourcePath) + strlen(destinationPath)+1] = '\0';
    }else{
        Dest = (char*)malloc(sizeof(char)*(strlen(destinationPath))+1);
        strcpy(Dest, destinationPath);
        Dest[strlen(destinationPath)] = '\0';
    }
```

mode (more than 1 argument) or interactive mode (just one argument). the arguments for the commands are then parsed from the chosen input stream first by the delimiter ";" that separates commands and then by a space that separates arguments. In addition to removing the delimiter, the parser also removes all newline characters. The number of tokens is stored in a struct along with the parsed tokens. After the input is parsed, each command is checked and if it has the correct number of arguments is executed. One of the most challenging parts of this project was making sure that didn't have any memory leaks or errors in the parser or command implementations.

In my copyFile command I originally used strdump to directly copy my new destination path back into the passed in destinationPath variable. While this worked, it caused a major memory leak because I was writing over the end of the destinationPath string. I changed my implementation to dynamically allocate a new Dest string that was the length of the source path plus the destination path to make sure that my new path had enough space.

Performance Results and Discussion

The only issue I was having regarding expected output was that my cat didn't match up with the expected output unless I specifically used a printf to print the newline character in main.c instead of using a write. Using either method works fine in regular use. The first screenshot is the time each command to run in interactive mode. The second screenshot is the time it took to run the contents of lab1_input.txt in file mode. The cp and mv commands took the longest to execute at .000304 and .00512 seconds respectively. This makes since because they both have to read and write to a file. Ls also took a large amount of time since it had to read the names of many files in a directory. In comparison pwd cd were both very fast.

```
zweisenb@ix-dev:~/415/Project 1$ ./pseudo-shell
>>> ls
. .. Makefile here abcdefgh bob test.txt george example-output.txt string_parser.c command.o test test_cat_file.txt test_script.sh abc abcde abc.txt pasidjfapsoidfjpoas
idjfpoaisjdfpoaisjdpfoiajspdofijaposdifjaposdijf d bobing output.txt aspodfijaposidfjpoasidjfpoaisjdfpoasijdf main.c pseudo-shell command.h ge you odsafj a aposidfjpoas
idjfpoapapafpa3jaosjqh apsoidfjpoasidjfpoasijdfpoasijdfpoiasjdfpoiajsdfpoiasjpdofiaspdofji command.c string_parser.o lab1_input.txt paosidfj string_parser.h main.o inpu
t.txt
The function took 0.000259 seconds to execute.
>>> pwd
/home/users/zweisenb/415/Project 1
The function took 0.000025 seconds to execute.
>>> mkdir random
The function took 0.000105 seconds to execute.
>>> cp input.txt ./you/
The function took 0.000304 seconds to execute.
>>> mv input.txt ./you/
The function took 0.000512 seconds to execute.
>>> rm ./you/input.txt
The function took 0.000167 seconds to execute.
>>> cat test_cat_file.txt
Sample content for cat test.

The function took 0.000050 seconds to execute.
>>> cd ..
The function took 0.000015 seconds to execute.
>>>
```

```
The function took 0.000865 seconds to execute.
```

Conclusion

Before completing this project, I felt like I underestimated the importance of error handling in many of my programs. Even in a program as simple as a shell, there are so many ways for the user to enter unexpected input and a shell needs to be able to handle every edge case. One of the most interesting things that I learned was how lower-level system calls interact with standard c library functions. For example, I was getting a lot of wrong outputs when I was using printf in conjunction with write. Since write executes right away, even if you execute printf first write will often print first. To fix this issue I changed most of my printf to write statements.