CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Colby H. Roberts
colbyr
951979360

Report

Introduction

Project 1 represents an engaging venture into the world of UNIX systems, where we endeavor to craft a pseudo-shell that adheres to UNIX standards. Within this shell, we're granted the capability to execute fundamental UNIX commands, thus enabling us to traverse directories seamlessly. This newfound functionality encompasses essential tasks such as directory navigation - granting us the power to change directors, to make directories for maintaining organized file structures, and printing of the working directory. Equally significant is the command set that facilitates file management, comprising operations like file copying, facilitating the safe transfer of data between locations. Additionally, we're equipped to perform file movement, which permits us to orchestrate the organized relocation of files within the system, contributing to better data management practices. The project also introduces the file removal operation, a fundamental aspect of system housekeeping. These diverse functionalities bestowed upon our pseudo shell open the gateway to a myriad of applications, offering users the flexibility to interact with their UNIX-like systems efficiently and effectively while gaining a deeper understanding of the intricacies underpinning these vital computing processes.

Background

Many of the algorithms I needed to implement proved relatively straightforward, particularly as I pondered how to recreate the functions within $command.c$. For instance, when working on $copy\_file$, I drew upon a Python implementation I'd encountered in the past. To gain insights into C file operations, I conducted research on topics like "how to open a file in C," "how to write to files in C," and "how to read files in C." This research was instrumental in understanding the intricacies of working with files in C. I implemented it character by character instead of line by line due to the latter approach's tendency to eliminate newline characters "$\backslash n$". I visualized the process of copying a file, akin to transcribing text from a computer to a piece of paper, where maintaining newline characters was essential for preserving the content's format and structure. This approach helped me tailor the implementation to meet the specific requirements of the task, ensuring a faithful replication of the source file.

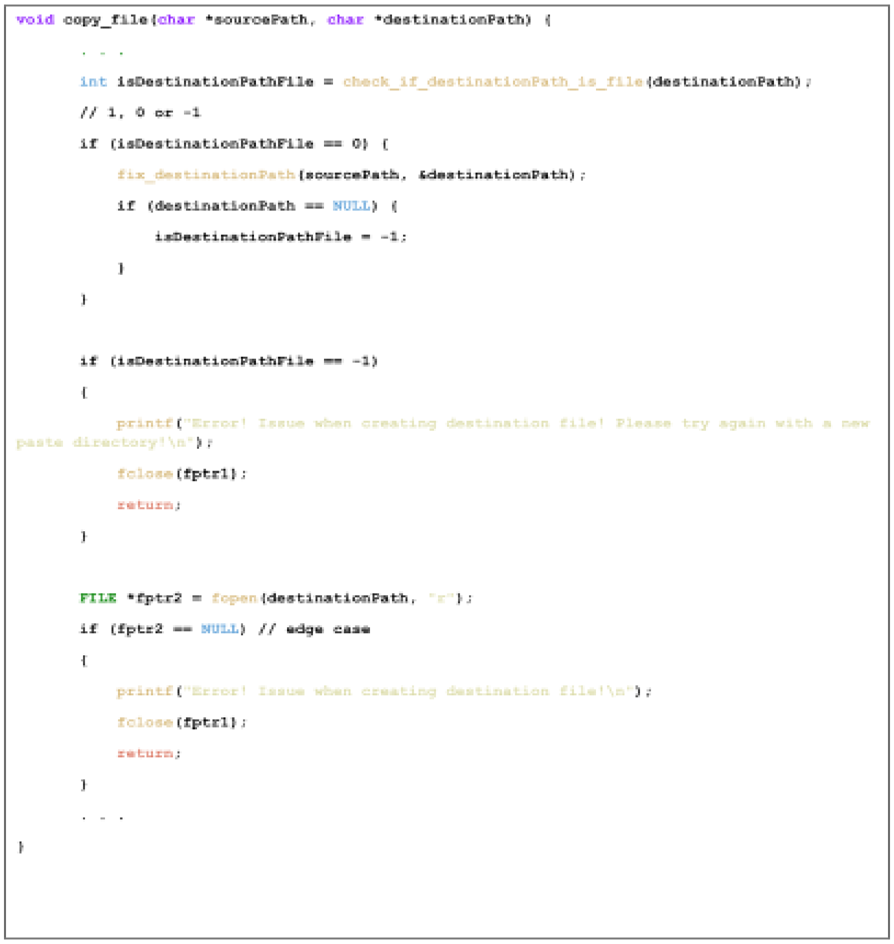
Implementation

$main.c$ provides a command-line utility for executing Unix commands in both interactive and file-driven modes. The $execute\_command$ function uses the $string\_parser$ and $command$ header files to parse and handle a variety of Unix commands before executing them. $execute\_command$ processes commands separated by semicolons and tokenizes them to determine the command and its arguments, then executes the commands with the appropriate logic and error handling. $main.c$ has two operating modes: a file-driven mode that receives commands from an input file and outputs the output to "output.txt," and an interactive mode that allows human input. The program handles complex commands with multiple inputs, allowing users to enter and execute a series of Unix commands ending with "$exit$" in interactive mode.

$command.c$ provides a collection of functions to interact with the file system and perform various system-related tasks. These functions include listing the contents of a directory $list\_dir$, displaying the current working directory $show\_current\_dir$, creating a new directory $make\_dir$, changing the current directory $change\_dir$, copying a file from a source to a destination path $copy\_file$, moving a file from a source to a destination path $moveFile$), deleting a file $delete\_file$, and displaying the content of a file $display\_file$. These functions are intended for use in command-line utilities that let users manage files and directories and run

Unix-like commands. When problems arise with these file system operations, the functions handle the error cases and provide informative error messages.

$string\_parser.c$ provides a robust and versatile utility for tokenizing input strings based on user-specified delimiters. It has three crucial functions: $count\_token$, which counts the number of tokens in an input string while taking into account edge cases like delimiters at the beginning or end of the string, str_filler, which builds a command-line structure from tokenized data while removing any newlines and controlling memory allocation, and $free\_command\_line$, which safely deallocates the memory used by the command_line structure With efficient and memory-safe operations, this string parser is built to handle a variety of use cases where string processing and token extraction are crucial.



```
void copy_file(char *sourcePath, char *destinationPath) {

        . . .

        int isDestinationPathFile = check_if_destinationPath_is_file(destinationPath);

        // 1, 0 or -1

        if (isDestinationPathFile == 0) {

            fix_destinationPath(sourcePath, &destinationPath);

            if (destinationPath == NULL) {

                isDestinationPathFile = -1;

            }

        }


        if (isDestinationPathFile == -1)

        {

            printf("Error! Issue when creating destination file! Please try again with a new
paste directory!\n");

            fclose(fptr1);

            return;

        }


        FILE *fptr2 = fopen(destinationPath, "r");

        if (fptr2 == NULL) // edge case

        {

            printf("Error! Issue when creating destination file!\n");

            fclose(fptr1);

            return;

        }

        . . .

}
```

To the left here, we have the middle third of $copy\_file$. Here, in some cases, the destination may not be valid, and we need to parse it, for example, a directory, $fopen$ requires a file path and $files/even_more_files$is not a file, which I check in $isDestination - PathFile$. After that, if the destination path is a file, I open/create the destination file. If the destination path is not a file but a directory, we append the file name (and however many "$\_copy$"s I need for it to not overwrite a file, in $fix\_destinationPath$. After that, I print transcribe into the second file character by character.

Performance Results and Discussion

The performance of this project, is $O(n)$ where n is the length of the files, that every function that deals with files iterates at most over the given source files once to either copy, move, display, or remove or the list directory, which is $O(m)$ where m is the number of files in the current directory. However, the print working directory, change directory, and make directory functions are all $O(1)$ because they all use system calls, (to my understanding of how they work behind the scenes).

Conclusion

Overall I learned much about how the UNIX system works, how It's implemented, and the hours of coding that went into the behind-the-scenes work of the terminal editor. Finally, I just hope that $fopen$ has write permissions on the grader's machine.