CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

*Arturo Diaz*

*arturod*

*951759326*

**Report**

**Introduction**

*This project involved the creation of a pseudo-shell that supports both interactive and file modes. The pseudo-shell allows potential users to run Unix-like commands such as ls, pwd, mkdir, cd, mv, cp, rm, and cat. The shell takes commands either interactively or from an input file and processes them one at a time, supporting command chaining using semicolons (;). Not all functionality was present with these commands, for example ls takes no arguments, but with ls in a real terminal environment you can pass in a directory to check contents of a specific directory. The project’s goal was to implement system call-based functionality while respecting a restricted set of allowed system calls, which included calls like readdir, opendir, getcwd, write, and others.*

*The shell was designed to execute commands in sequence and provide feedback, such as errors for unrecognized commands or missing parameters. This project helped improve understanding of how basic operating system functionalities, like file management and system calls, are implemented at a low level.*

**Background**

*The primary focus of this project was implementing Unix-like shell commands using system calls. Each command required the use of system calls like opendir() and readdir() for directory listing, mkdir() for directory creation, and remove() for file removal. To avoid using restricted system calls such as execvp() and syscall(), commands like ls and mv had to be manually implemented using system calls for reading and writing files.*

*In the development of this pseudo-shell, one key decision was to handle commands like mv without the use of rename(), which required reading the source file, writing it to the destination, and then deleting the original. Similarly, commands like cat and cp used a combination of open(), read(), and write() to handle file input/output operations.*

**Implementation**

*The pseudo-shell operates in two modes: interactive and file mode. In interactive mode, the shell prompts the user with >>> and waits for command input. In file mode, the shell reads commands from a batch file and executes them in sequence. The commands are parsed using strtok() to split the input by spaces and semicolons. Each command is processed by functions that handle the appropriate system calls.*

*For example, the mkdir command uses the mkdir() system call to create directories, while the readdir() system call is used to implement the ls command. The cp command reads data from the source file using read() and writes it to the destination using write(). Error handling is implemented using perror() to display system error* *messages for issues such as permission errors or file not found errors.*

***A computer screen shot of white text

Description automatically generated*** ***A black background with white text

Description automatically generated***

*Figure SEQ Figure \\* ARABIC 1: Some Algorithm implementation*

*One challenge was ensuring that both* ***file\_mode*** *and* ***interactive\_mode*** *called the same command functions, but with different output destinations: one needed to print results to the terminal, and the other needed to print to a file. This required careful handling of output redirection, as the same functions had to be reusable for both modes. It was particularly difficult to manage the redirection efficiently while avoiding code duplication.*

*The solution I used involved calling system functions within the* ***file\_mode*** *function. Since all the command functions already used the write() system call, I could redirect the output from the terminal to a file using one simple line of code:*

**

*This allowed the commands to write their output to a file (output.txt) when running in* ***file\_mode****, while the same functions continued to print to the terminal when in* ***interactive\_mode****. This approach minimized the need for separate implementations for each mode and made the code more maintainable.*

**Performance Results and Discussion**

*The pseudo-shell was tested with various Unix-like commands, both in interactive mode and by reading from a batch file. Commands like ls, pwd, mkdir, and rm performed as expected, displaying directory contents, showing the current directory, creating directories, and deleting files, respectively. The mv and cp commands were also functional, handling file movements and copies without relying on restricted system calls like rename().*

*While most commands ran smoothly, there were some edge cases, such as incorrect file paths or missing arguments, where appropriate error messages were displayed. For example, attempting to rm a directory resulted in an error, as intended, since rm was designed to handle only files, not directories. Similarly, the shell correctly reported errors for unrecognized commands or missing parameters, such as when running mkdir without a directory name..*

**Conclusion**

*This project deepened my understanding of how basic Unix commands are implemented using system calls. By building a pseudo-shell without relying on higher-level command execution functions like execvp(), I gained insight into the underlying mechanisms of file and directory manipulation. Additionally, the project provided valuable experience with string manipulation, memory management, and error handling in C.*

*Overall, the pseudo-shell successfully replicates the behavior of several common Unix commands, and the project was a rewarding exercise in low-level programming with system calls.*