**COSC 4302 Project Shell – Written Report**

Group: Juan Lerma, Cameron Lewis, Justin Magana

Introduction:

A shell program is a computer program that allows a user to interact with an operating system’s services. Typically, operating system shells use either a command-line interface or a graphical user interface. The shell acts as an intermediary between the user and the kernel, interpreting and executing user commands. It is named a shell because it is the outermost layer around the operating system, providing a user-friendly interface to interact with the hidden system resources.

In this group project, we designed a shell program to emulate the basic functionality of Unix-like shells. Our shell provides a text-based interface where users can interact with the operating system by entering commands. These commands are broken down into separate components, like arguments and command names. Users can enter commands that the shell parses, identifies, and handles the commands entered. After the parsing phase, the shell then searches the system's PATH environment for the specified command. The shell executes the command after it has been located, and the execution process then consists of performing the specified command. Finally, the output is now shown on the console for the user to view after the execution is complete. This process captures how users interact with the operating system.

Code components:

Our code is separated into two files, minishell.h and main.c. Below is a brief explanation of their components and functions:

* minishell.h:
  + Contains macro definitions, data structure declarations, and function prototypes.
  + Ensures clear separation between implementation and interface, improving readability and organization.
* main.c:
  + Implements the shell’s core functionality.
  + Key functions are explained later.

Key functions:

* printPrompt(): Displays the shell prompt.
* readCommand(char \*buffer): Reads user input into a buffer for parsing.
* parseCommand(char \*cLine, struct command\_t \*cmd): Tokenizes user input into a structured command format.
* handleInternalCommand(struct command\_t \*cmd): Executes built-in commands like cd, pwd, and exit internally.
* parsePath(char \*dirs[]): Parses the PATH environment variable into an array of directories.
* lookupPath(char \*\*argv, char \*\*dirs): Searches the directories in the PATH for the specified command.

The program implements a basic shell program that allows the user to execute commands interactively. It starts by parsing the system's PATH environment variable to locate executable files. In an infinite loop, the shell displays a prompt with the hostname and current directory, reads user input, and finally tokenizes it into a command and arguments. Built-in commands like cd, pwd, and exit are handled internally, while external commands are searched in the PATH directories. The shell uses fork() to create a child process to execute commands using execv(), and uses the parent process to wait for the child to complete. Errors such as “invalid input” or “command not found” are handled and memory allocated for commands is freed after each execution to avoid leaks. This simple shell demonstrates the fundamental concepts of parsing, process management, and system calls in Unix-like operating systems.

How to run the project:

To run the shell project the user must use a Linux-based system with a GCC compiler, and it is important to ensure that the source files are in the same directory. To compile the program use the command “gcc -o shell main.c”. After compiling the program we can run the shell by executing “./shell”. Finally, you can interact with the shell and enter commands at the prompt such as the built-in commands cd, pwd, and exit, as well as external commands available in the system's PATH.

Conclusion:

Our shell program implements core features required for a basic interactive shell, such as command parsing, PATH searching, and process creation. It demonstrates concepts of process management, error handling, and environment variable manipulation. By creating a simple shell that can execute both built-in and external commands, we gained experience in system calls, child processes, and environment variable manipulation. While the program may be a simple shell, it serves as a solid foundation for the exploration of operating system concepts. This project has deepened our understanding of how operating systems interact with users and execute commands.