**Bo Sun**

**COSC 4302-01**

**Operating Systems**

**4/28/24**

**Simple Shell Project Report**

**A red and black logo

Description automatically generated**

**Submitted by:**

**M. K. Dhillon - L########**

**Will Lang – L20545362**

**Tom Le – L20531694**

**Table of Contents**

|  |  |
| --- | --- |
| **Topic** | **Page No.** |
| **Introduction** | **3** |
| **How Shell Programs Work** | **3** |
| **Designing Our Shell Projects** | **5** |
| **Reviewing Our Programs** | **5** |
| **Analyzing Inputs and Outputs** | **11** |
| **Conclusion** | **13** |
| **Cited Works** | **14** |

**Introduction**

The shell command line interpreter is a program that allows users to interact with the computer by typing commands through a text-based, or character-oriented human-computer interface instead of using a mouse or keyboard. So, when a user enters a command, the shell interprets the command and calls the OS to carry out the requested process. The shell maintains the kernel interface between the user and the OS’ lowest-level core component. It allows for tasks to be automated, making work much easier to manage and less time-consuming.

UNIX users often have access to several command-line interpreters for functional sessions. When a user logs in to the system actively, a shell program continues to operate for the duration of the session. Shell preferences are often kept in the user's profile, such as /etc/passwd and /etc/shadow files, or key-based authentication systems like SSH keys. Regardless, the user is free to employ any other shell functions accessible to them.

The Unix shell is a command and scripting language used by the operating system to manage system execution through shell scripting. Shells designed for different operating frameworks often provide comparable functionality.

**How Shell Programs Work**

A shell uses different and unique methods to execute a user’s command. The basic shell approach is to correlate the users’ inputs to pre-defined trusted commands built into the device. A device should not treat every user input as a trusted command, because it could break the device easily. That is why Unix-based Operating Systems have a user mode and a supervisor mode.

Different stages of Shell that allow easy-to-use user interaction, command execution, scripting, and system control:

* B shell /bin/sh

Created by Stephen R. Bourne, the purpose of the Borne shell is used as a scripting language instead of an interactive shell. The B shell is the standard shell for many Unix operating systems. It usually uses the $ character as the prompt, so you can tell if your OS is using the Bourne shell.

* C shell /bin/csh

Created by Bill Joy, the C shell was an alternative to the B shell and introduced various features and syntax inspired by the C programming language that were not included in the B shell.

* K shell /bin/ksh

Created by David Korn, the K shell includes features from both the B shell and C shell, in addition to other new shells.

* Bash

Created by Brian Fox, Bash was created for the GNU project as a free software intended to replace the B shell. Bash is based on the B shell and contains features from both the C shell and K shell.

* tcsh

Developed by Ken Greer, the Tenex C shell, or tcsh, is the default shell for FreeBSD OS. Tcsh is a C shell with improved overall interactive features.

* zsh

Developed by Paul Falstad, the Z shell, or zsh, is the default shell for macOS and has capabilities like those found in bash, ksh, tcsh, and others.

To effectively process user inputs in a simple UNIX-style shell, there are different ways that could be used. One way includes creating a new thread for each input request and in our project, we created a simple Shell in C that runs on UNIX computers. A shell's main function is to interpret instructions and scripts that are delivered either directly through a file or the command line. A command interpreter processes instructions with several parameters, whereas a shell script file interpreter receives and executes commands from a file.

**Designing Our Projects**

We plan to design the project by following the program skeleton given to us by the textbook. The best way to summarize the design we were given was that the program finds directories that commands might be in, and then we use the gets() commands to grab the user’s input. Once we grab the input, we parse it and save the user command and parameters into the command struct. Then, we iterate through every folder that might have the command with the user’s input command concatenated beside the predefined directories. If the concatenated string matches an actual directory of a command, it saves the command to execute later. Finally, we use execv() with the directory to the command and the parameters in argv[] to run the command in a forked process. For our threaded program, we would do the same, but with a thread rather than a whole forked process.

**Reviewing Our Programs**

This paragraph will go more in-depth over our program that we made based on the design above. First, let us analyze anything that we need outside of the program, which is minishell.h.

A screen shot of a computer code

Description automatically generated

In short, this is a file specifically made to create max amounts of arguments, argument lengths, what we have defined as a whitespace, and the data structure command\_t. If we wanted to, we could have removed this file and defined them there, but this helps to save space.

A black background with white text

Description automatically generated

This is the header for the program that imports libraries that we will need or headers. This would have been much bigger if we had removed minishell.h. Moving forward, the most important library we have included here is the one for string methods, so we can copy, concatenate, and manipulate strings in C. We have also included stdlib.h, and unistd.h, and stdio.h because we needed to allocate memory, create child processes, run execv, and print statements. Without these, we would not be able to do anything.

A computer screen shot of white text

Description automatically generated

Next, we will be analyzing the first method and one of the more important ones. This method takes in the arguments of the input commands in argv and the directories that the command could be in in dir. It first checks for some test cases where it cannot do its job and ends the method. If the input can be worked with, it creates a temporary string called tpath, which is used to compare later. While in the for loop, tpath becomes the first directory in dir[], and we concatenate a ’/’ and the name of the input command from the user, so that it looks like [directory]/[command]. Then, we attempt to access the file, and if we get something back, we have found the file. We iterate through every directory attempting to access that directory with the name of the command until dir[] is null. Once we do find the command directory, we copy the directory onto argv[0]. If not, we return null.

A computer screen with white text

Description automatically generated

This method’s goal is to parse the string by removing the whitespaces and placing the separate substrings into argv[]. Before we place them in, we must allocate memory to place the arguments substrings into argv[]. Finally, it copies the first argv[] element into name, since the first substring of the input in the shell should be the command name.

A computer screen shot of white text

Description automatically generated

The point of this method is to grab all the directories the input command might be in, so what it does is that empties dir[] in case it was used before, then it separates the directories into substrings by separating the getEnvVar by ‘:’. Finally, it stores those substrings into dir[] for a lookup path to use later. Anything after that is just print statements for debugging purposes.

A black screen with white text

Description automatically generated

In summary, the first command is a glorified print command we use to show the user can input a command into the shell program, and readCommand just grabs the command input by the user and stores it.

A screenshot of a computer program

Description automatically generated

Finally, we get to the main method, which puts all of these to use. We plan to go more in-depth about what happens to inputs in the next section. For now, the main method parses paths where the command could be found and it brings in the data structure command\_t for storing the inputs. Then, when it is a while loop that only closes if you input quit, it reads the input command from the user, parses the input, attempts to find the command directory, and runs the command by forking a child process that runs execv with the command directory and parameters in argv. This is how it works for the first part of the project. As for the second part, it is a little different.

A computer screen with white text

Description automatically generated

A computer screen shot of white text

Description automatically generated

The main difference between the first part and the second part of this project is how we run execv. The first part is in a process, and the second part is in a thread. As shown above, these code snippets are only in the second part. We create a method that the thread in main can run, which creates a thread that can run execv. We must fork the thread because execv closes the process with it. Without it, the thread would run execv and close the whole process, including the shell program that the thread came from. Finally, we altered the main method, so that it creates, joins, and accommodates for the threads created and joined. We needed to include sleep in the process to allow for the thread to be fully complete before moving on in the while loop.

**Analyzing Inputs and OutputsA screenshot of a computer program

Description automatically generated**

Firstly, we plan to go over the program running, step by step and what happens in the background. First, we run the command “gcc project1.c -o p” or “gcc project2.c -lpthread -o p" and follow up with “./p”. This compiles the program and runs it. What happens in the background before we can even input is that we run the command parse paths in order to grab the directories that the next input command might come from. Then, the while loop begins, which first allocates room for our input command and runs printprompt to add the fancy “>>” before our input command. Once we input the command, “ls -l”, readcommand gets it and separates the string into substrings by splitting it at whitespaces such as “ ”, and the substring elements are placed into argv[] using parsecommand. If the input command is “quit”, it will break the while loop, which ends the program. What follows is the command directory is looked for using the lookuppath method. It searches everywhere for the command directory by just concatenating “/ls” to every path available and attempting to access it. If it finds the command directory, lookuppath returns the directory “yadda/yadda/ls” to argv[0], which goes to command.name, and every element in argv is moved down, so argv is now solely parameters. Next, the main method checks if the directory is null and prints if it is null. Finally, it forks a child process or a thread and runs execv(command.name, command.argv), where command.name is the location of the command and every element is argv is a parameter that was parsed into argv earlier. Therefore, the elaborate input of execv would look like execv(“yadda/yadda/ls”,

[“-l”,NULL,…]). Once the command runs, either in the thread or child process, it closes and continues the while loop until “quit” is inputted.

**Conclusion**

Our Simple Shell Project reports on how the shell command line interpreters could reduce user effort and automate UNIX system tasks for easier use. We examined shell capabilities like command interpretation and parsing and used processes like fork and thread to interpret UNIX commands. Working on this project taught us about shell scripting and its foundation in operating system development.

**Cited Sources**

Jones, M. “Evolution of Shells in Linux.” IBM Developer, 6 Dec. 2011, developer.ibm.com/tutorials/l-linux-shells/.

Behery, Anthony. “Linux Shells for Beginners – Bash, Zsh, and Fish Explained.” freeCodeCamp.Org, freeCodeCamp.org, 15 Dec. 2022, www.freecodecamp.org/news/linux-shells-explained/.

“Unix / Linux - What Is Shells?” Tutorialspoint, www.tutorialspoint.com/unix/unix-what-is-shell.htm. Accessed 15 Apr. 2024.