**COSC 4302 Operating Systems**

**Instructor: Dr. Bo Sun**

**Grader: Sridhar Balaji Gurusubburajan**

**Shell Project Report**

**Group Members:**

**Teresa Iles**

**Eli Peveto**

**Sai Terapalli**

Date Submitted: November 30, 2021

Introduction

A shell program is an interface that allows a user to interact with the operating system. The user can enter commands to the operating system to execute applications and manage files. When the operating system executes an application, it finds the file for that application and uses the execv() command to execute it.

When a user types a command in the shell, they are typing a filename for the operating system to execute. Many commands also require arguments. These are parameters that get passed to the executing program that the program needs to use.

For example, to compile a C program, we type:

gcc filename1.c -o filename2.

In this example, “gcc” is the command, and “filename1.c”, “-o”, “filename2” are arguments for the command. The “filename1” represents the name of the file to be compiled. The “-o” is an option to create a named output file, given by “filename2”. Without the option, the default is to create an object file called “a.out” for the C program.

The shell program needs to be able to interpret the command and arguments provided by the user. Then, it needs to create a child process to execute the command entered by the user. After executing, it will show the command line prompt again, waiting for a new command. The user can exit the shell by entering ‘q’ or ‘Q’ for quit.

Implementation

The textbook outlines a strategy for creating the shell program, along with some helpful code to get started. The steps to implementing this shell are:

1. Initiate the shell.
2. Use the environment variable to get all the directory paths available to the user.
3. Print a prompt and wait for user to enter a command.
4. Read and store the command line entered by the user.
5. Parse the command line string and store the command information in the struct variable.
6. Find the file that matches the command name. Search through each directory that was discovered in Step 2. Look for the file that matches the command name.
7. If the command file exists, create a child process to run the command.
8. After executing, the prompt will appear again, waiting for the next command.
9. Create a way for user to exit the shell program.

Running our Program

The program is provided in the file “ShellProgram2.c”. To run the program, first we need to compile it.

Type: gcc ShellProgram2 -o OutputFilename

After it compiles, type: ./OutputFilename

To test the program, you will need to have an idea of what command you want to use and any files you may need to work with the command. Be sure to put the absolute file path if you reference a file as an argument if it’s not in the current directory.

Bugs/Problems

We had a recurring problem in the program: a segmentation fault error. After doing some research online, it seems that this type of error occurs when you try to write or copy information to a place in memory that is not allocated or specified. We found a helpful tutorial on how to use the gdb debugger to debug this issue. This debugger proved to be very helpful in resolving the issues as it allowed us to see exactly what was going on. Some of the useful commands that we used in the debugger were “backtrace”, “frame number”, and “print variableName”.

Once we got past the segmentation fault errors, we were able to get the program to run and went into a testing phase. We found that when we typed a fake command such as “fakecommand x y z 1 2 3”, then later typed in another command, such as “ls”, there was an error because the “command” variable still had all the arguments from the fake command. Since these arguments were not valid for the “ls” command, they caused a problem and “ls” would not run. To resolve this issue, we created a struct command\_t “empty” variable, that has everything in it set to NULL or 0. Then, each time the loop starts up again for receiving a new command, we assign this “empty” command\_t variable to the “command” variable we use in the program. This resets the values in the “command” variable for each new iteration of the loop.

We also noticed in testing that if the user hits <enter> without entering anything, there was a segmentation fault. We were able to fix this with an “if” statement so that now the user can type <enter> with no data, and it will display another prompt.

Another problem we noticed is that we did not figure out how to make the shell program execute the “cd” command successfully. We do not yet know the solution to this issue.

Output

Here is a screen shot of sample output from the program. Here the program takes the command ls and lists the files and directories.

Text

Description automatically generated

Next, we will make a new directory called “testing”. Then type the command ls again, and we see that “testing” has been added.

Graphical user interface

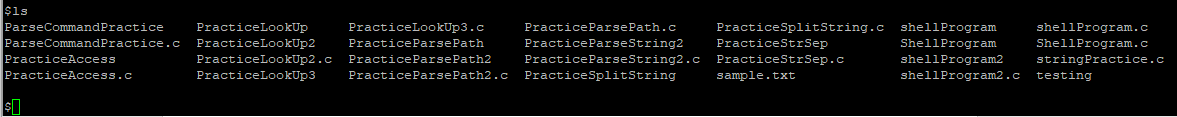
Description automatically generated

Next, we will enter a fake command. We get an error message and prompt appears again.

A screenshot of a computer

Description automatically generated with medium confidence

Now, we enter “ls” again, and it works.



Then, we try the command “cat”. It works.

Text

Description automatically generated

Here, the user presses <enter> multiple times.

Shape

Description automatically generated with medium confidence

Finally, when we are ready to quit, we type ‘q’ or ‘Q’.

Text

Description automatically generated with medium confidence

Conclusion

The shell program was an interesting challenge. We were able to create the shell program that uses the library files available in the Linux to run shell commands. It was challenging to understand how to get all the different functions to work together, and to learn how to manipulate strings and use pointers with strings.