Carmelo Estrada

Eric Platt

Shell Report

Introduction:

A shell program essentially takes in commands and gives them to the operating system to perform. In linux, the original shell program sh was at one point improved and now the standard shell is bash, or the born-again shell. We wrote a very simple, stripped down shell that has a few commands found in existing shell programs in order to better understand operating systems. The shell we wrote, CPPShell, is written in C and is a command line interface that uses the terminal to interact with the input and output of text. The commands we implemented in the CPPShell are: cd to change the directory, pwd to print the working directory, mkdir to make a new directory, rmdir to remove an empty directory, ls to list contents of a directory, cp to copy a file to another, exit to terminate the shell, and a fork to run executables.

Design:

The program is written in C and uses a header file to link each command which are each in their own c file. CPPShell.c contains the main and includes all the headers to each command. The main is contains a loop that allows the program to keep accepting command requests until it receives an exit command. The loop checks the command string against all programmed cases in an if else structure and runs the appropriate function, passing in arguments if needed. To run an executable command, the shell runs them on their own forked process and waits for it to finish. The shell will print a color-coded prompt with a string that has “CPPShell” and the current directory, a command with argument can then be typed in.

Techniques:

We are leveraging several linux libraries that provide most of the functionality for our commands. By including libraries such as sys/wait.h and dirent.h, we can ask the OS to wait and access files respectively, saving us from writing granular low-level code to do such things. The shell handles arguments and handles error cases for the C functions we use. For example, when calling cd to change directory, we can pass the path argument to he chdir C function if an argument is provided, or if we see that no argument was provided, we can pass a period to the chdir function as an assumption so the user does not have to do that. In the cp copy file function, we use the stdlib.h to call fopen to open a source file and destination file, and fputc to copy a buffer of data continuously from source to target until we reach the end of file, then we close both files once the copy is made. The ls list directory contents command uses dirent.h to access directory contents and lists them in a tab delimited format. The real ls in bash has better formatting but ours is functional enough. We even detect of each item is a file or a directory and highlight directories with color to differentiate them using the printf color codes for terminals (ie the \033 escape char with [#m codes). The pwd command simply uses the uninstd.h library to call the getcwd function, which fills a string up to a certain length with the current working directory. Mkdir is also simple as it uses the mkdir.h library to call the mkdir c function. We check if the function returns a zero or not to check if an error occurred, in which case we then call perror to print the dynamic error message. On other errors we use printf instead of perror when we want a less verbose message. In

the rmdir remove directory command, we started with a simple method that would only remove empty directories using the remove function of diren.h and later added functionality to remove files in that directory to make it more powerful using the ftw.h library. In order to run other executables, we had to use a fork to run them on another process and execute using the execvp C function and make the shell wait for that process to finish using the waitpid function.

Findings:

Shells typically do not have commands written from scratch, but rather call on the OS to perform many of the functions needed and provides a user-friendly interface with some error handling to avoid basic errors or provide feedback on errors. Shells also can provide some decision-making structures to combine many tools and functions in one place. To run other executables, we had to concurrently run another process using a fork, making the shell wait until the new process finishes.

Conclusion:

The shell is a layer that makes it easier for users to use facilities and functions of the OS while providing a command line interface through a terminal. Without the OS libraries provided to C by the linux, the project would be a difficult endeavor, since we rely on functions that the OS provides such as fgets to get user input from the terminal. By building on these building block functions, we were able to make a functional, although simple, shell relatively quickly.

Resources:

1) Basic Shell Template

https://sites.cs.ucsb.edu/~tyang/class/170s15/projects/HW0.html

2) G4G Linux Shell

https://www.geeksforgeeks.org/making-linux-shell-c/

3) C Tutorial with fork, exec and other commands we will use

https://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

4) Very simple shell video tutorials

https://www.youtube.com/watch?v=QUCSyDFPbOI

https://www.youtube.com/watch?v=z4LEuxMGGs8

5) Video on fork() command

https://www.youtube.com/watch?v=9seb8hddeK4

6) Shell video with fork and exec (long)

https://www.youtube.com/watch?v=5Qim9wufNP0