Andrew Kim, Kobe Lee, Celine Tran

CS 4310

Professor Talari

Assignment 4 Report

Introduction:

The primary function of a shell is to allow the user to have access to a computer through the execution of various commands by serving as an interface and command interpreter. The most popular shells are Bash, which stands for Bourne Again Shell, Tcsh, Ksh, and Zsh. Shell scripting is beneficial in that it allows for quicker access to various computer functions that would otherwise take longer with the use of a standard GUI-based operating system. For example, the process of navigating to a deeply hidden file through the use of a Windows file service would involve clicking through numerous folders in order to arrive at a certain destination -- this can be achieved in a single line through the shell. This project seeks to implement the most basic and common shell commands -- cd, pwd, mkdir, rmdir, ls, cp, and exit. Through the medium of the relatively low-level language C, we focused on the implementations of each of these functions one by one, determining the logic that fuels the execution of each. In the process, we were forced to consider the pieces that go into a function like changing directory from the bottom up.

Design/Techniques:

Within the implementation of the actual shell, we learned how to go forward through trial and error. In the beginning, we tried to write out each of the functions in one main file; however, we quickly realized that this was inefficient and made for code that was extremely difficult to read. For that reason, within the main file, we designed the basic skeleton of what a shell is -- an interface that loops through customer input. Within our shell.c file, which effectively serves as our main file, we set a double pointer “command” to whatever the user’s input was at any given time. From there, we went systematically down the list with functions going from simple to complex. We decided to complete the ls function first, as it serves as an effective baseline function that we could base the others on. This allowed for us to be able to also verify the other functions and whether they produced correct results. In terms of buffer size, we decided to consistently use 1024 bytes as a cap, as this was more than sufficient for our needs. With the pwd function established, we were able to utilize cd in order to manipulate the “present working directory” -- the two worked hand in hand. Cp wasn’t able to be worked on until pwd and cd were established, in the case that the file was in another directory -- we were then able to complete it at this point. Finally, mkdir, and rmdir worked hand in hand in simply creating or removing folders within the current directory. Finally, in order to maintain a collaborative coding environment, we used Repl.it’s collaboration feature.

Findings:

The most difficult functions to implement were mkdir and rmdir. Upon researching critical elements in the processes that go behind each, we found that most implementations relied upon the built-in mkdir command that comes default with the C language. For example, the following resource speaks solely to the mkdir() function that takes in two arguments (the name and the mode). The logic that goes into the function itself is not referenced: https://www.geeksforgeeks.org/create-directoryfolder-cc-program/#:~:text=The%20mkdir()%20function%20creates,empty%20directory%20with%20name%20filename.&text=Note%3A%20A%20return%20value%20of,%2C%20and%20%2D1%20indicates%20failure.

Furthermore, the cd command is very closely tied to the chdir function and therefore relies upon it heavily. Another important element that we did not consider until we got to implementing is that adding additional parameters requires much further logic to be written. The use of flags like -i and -r would require hundreds of lines of code more -- the complexity of the lexical analysis increases proportionally to the number of input factors and arguments to be parsed.

Conclusion:

All in all, the development of a shell is a complex undertaking that has a large number of moving parts. The need for a language like C or a C++ with pointers that overall lies closer to the machine than higher level languages like Java and Python is clear. Even with the implementation of the most bare-bones commands, there are hundreds of lines that are necessary. There is a reason that the majority of these commands are built into C and can be referenced directly with simple import statements and function declarations. The basic shell architecture involving a pipeline with the use of input analysis and subsequent parsing is the precedent that is to be utilized across all shells.