Maximum Wilder-Smith
013134095
Eric Vitolo
013076245

Assignment 4 - Report

Introduction:

In approaching the assignment of creating a UNIX-like shell in C++ we started with the prompt. Like a proper UNIX command prompt, ours has a prompt string that starts with the user's name, followed by the computer's name, then the current working directory or cwd. Following this there is a '$' then the space where the user will input their commands. The prompt's cwd must be dynamically updated when the user changes directories.

Design:

The basic design revolves around a loop of printing the prompt string with an accurate cwd, followed by reading the user input and entering a switch block to figure out which command is to be executed. The user's input string is compared to the possible commands to see which one is to be executed. The methods for running each command are defined in a header file, 'shell.h'. This header file defines methods that all accept the string of the user's input and output any errors or issues with running the command. The main program figures out which command is to be run, then passes the user's input minus the command characters into the respective function. If there is an output from the method, then it will be stored in 'output' and printed at the end of the cycle. For any function that requires two arguments, the user input is split based on the location of the space, and the two paths are handled separately (such as for cp). A global cwd variable was needed so that is defined in the shell.h header file and used by all subsequent functions for reference as the current path.

Techniques:

Most of the needed functions exist within the C++ filesystem class. So most of the functions simply convert the user input into a format that the filesystem class can work with. This is done by using the cwd variable along with the user input to create paths for the filesystem classes to perform manipulation on. Usually before the method call we need to verify if the specified file/directory exists, then if it does, make sure it is of the proper type before preceding.

Findings:

It is interesting to see how much work is needed to create even simple commands for a UNIX-like system. Of course, without helper classes such as filesystem this work would have been far more tedious and would have taken much longer. It is also interesting to see how many commands come in pairs, such as file deletion/creation. It is also interesting how rm and rmdir are separate commands despite having almost the same functionality and syntax. They even return errors that are very similar, although the separation of functionality is likely to increase reliability of the system.

Conclusion:

This was a very interesting project and was an interesting method of showing how even the most basic OS-human interaction is tricky. More commands would have been good to implement although that would be difficult with the time constraint. Although these commands were relatively simple, more complex ones like piping proved to be much more difficult. While this is but a simple text interface, a full human-OS interaction system such as a GUI seems quite an intense undertaking.

New Features:

The main new feature we implemented was the ability to perform basic piping. For this functionality, the first portion of the command, before the pipe, is executed in a special manner such that the output of the program can be recorded. This output is then fed into the second program as it executes. We also implemented the ‘rm’ function to delete files. The function is almost the same as the rmdir command, although we check to make sure the file is a file instead of a directory, then delete it. We added these features as piping provides support for a large amount of complex operations while the rm command allows us to delete files that have been copied during testing.