Assignment One: Mini Shell

Due: March 12th

Introduction

The shell is integral component of any operating system: it is the interactive component that allows the user to control the operating system and issue commands. Your objective for this assignment is to implement a shell with basic IO redirection and a few other features. *You may work in groups of two for this assignment. Note that the difficulty will include roughly another person's worth of work!*

Basics

At its most basic level, the shell is responsible for executing commands. A shell is connected to the keyboard, something that is determined by calling fstat(). The keyboard input is interpreted by the program in several ways:

- Regular typing is ignored so commands may be entered
- A carriage return (enter) triggers the execution of the current terminal contents
- Tab will autocomplete the current command (groups only)

Interpreting Commands

When a command is entered, the Parser class I have included automatically parses the input into a list, if said command is valid. Since our shell supports IO redirection, the linked list elements will be each command, in the order they were separated in by pipe (|) operators. The file redirection operators (<, >); input and output respectively, are allowed once per command, any extras are discarded. It is assumed that we will not have any complicated operators that redirect specific streams (i.e. >2, that redirects stderr). Here is a summary of the functionality you need to support:

- Input Redirection (cmd < file): redirects the contents of file to the command's stdin.
- Input Redirection (cmd > file): redirects the stdout of command to file.
- Pipe operator (cmd1 | cmd2): redirects the stdout of cmd1 to the stdin of cmd2

Executing Commands

The main objective of Part One is to execute and redirect the input/output of these commands properly. You must iterate over the linked list structure, fork()'ing a new process for each command and then invoking the new process with execve(). At this point you wait for the command to finish and store its result for the next section of the pipeline. The commands are in the correct execution order after being parsed Note: you must use a pipe and the dup2() function to appropriately redirect the output. Refer to the in-class example to get started. Keep in mind that some commands may pipe to the next command **and output to a file**. This is a more complicated scenario you need to look out for.

Groups of Two

If you choose to work in a group, you will need to implement an autocomplete feature for the terminal. The autocomplete function should first search the directory you are in, and then the $PATH environment variable, if nothing is found. It should behave the same as the bash autocomplete function, except for allowing the user to use tab multiple times to select an option. Instead, your program should print the available options and ask the user which option they want (i.e., enter option 1-10). If there are more than 10, prompt the user to further complete the command. This should keep the program logic a lot simpler than if we allowed navigation through a list of suggestions as bash does.

To implement this, I suggest you get started with getting the correct behaviour when the user presses tab. cin can be used to detect the tab character being entered, but you'll need to modify the while loop in the main function to use something that breaks on tab delimiters. Getline() will stop on new lines, and you want tab characters as well!

Afterwards, implement the search function to find commands that match. I personally would grab the path environment variable from the shell that launched it, then assemble that into a list you can search when the user presses tab. Use std::getenv() for this. You may choose to store the suggestion options in whatever structure you deem appropriate. To search the current directory, simply read it with std::filesystem::directory_iterator. There is an example of this in the IO Programming lecture slides!

Starter Code

What I have given you for starter code is a main() procedure that manages a shell, and a Parser class that takes a line of input and converts it into Command classes for you to use. This will allow you to focus on the system calls and what not, rather than messing around creating a parser which is a topic better suited for another course.

Deliverables

The first deliverable the first part of the report, write a page or so on the data structures and methods you used to implement part one (and two). For the last bit, look at the code in Parser.cpp. This code makes heavy use of standard library functions, as I wanted to complete it quickly and show off some modern C++ idioms and patterns. How does this differ from the programming styles we've used in class? Discuss (in abstract) the implications writing code like this might have. Do not analyze the code unless you want to, most of it is likely things you haven't seen before. I am only looking for comments on this style of programming, and how it does or doesn't follow the principals we discussed in class. You only need to write a paragraph or so. *Hint: would writing this in an old fashioned way work better? Perhaps it would allow me to only traverse the command string a single time for each pipeline of commands! Also, how risky is it to liberally use standard libraries? Could we run into problems finding slowdown?*

Submit a zip file containing your source code and report via canvas.

Bonus

As a group of one, you may choose to extend the parser and shell to support redirection of stderr for a 10% bonus. This is represented by adding the number two (2<, 2>) to either of the file redirection operators, and will output stderr to a file.

Alternatively, you may complete part two for the same 10% bonus. ***You may not do both to receive 20%, this option is only for those that would rather implement the autocomplete section.***

Groups of two may receive a 10% bonus by extending the autocomplete to display a guess at what the command is (i.e. the top command) and allow the user to press right arrow to fill it in.