Corey Talbert
Assignment 2 Report
10-10-2022
CS 433-01 Operating Systems
Xiaoyu Zhang

Assignment 2 Program Report: UNIX Shell

Introduction

This project is a limited implementation of an essential operating system service, the command line interface. The UNIX-style shell is a convenient way to interact with a POSIX-compliant system. The shell provides a means by which programs may be executed, the file system can be explored, and system information can be viewed. Concurrency between the parent shell and the child processes it initiates can be created by the use of a special symbol, "&". This implementation also includes a simple command history feature for recalling the last command entered by the user, redirection for reading from or writing to files, and the ability to create an ordinary pipe between two programs for inter-process communication.

POSIX API

Several functions specified by the POSIX standards are used by this shell. They are implemented in the C POSIX library, and this program uses the headers unistd.h, fcntl.h, sys/wait.h. dup() is used for the duplication of file descriptors, essential in both redirection and inter-process communication. open() and close() act on those descriptors to manage the files. To create the child process in which the command will run its program, fork() is called. exec() invokes programs indicated by the command text. In particular, execvp() is used to receive a command as a collection of tokens, and searches the locations defined in the system PATH variable for the program executable.

Design: the Command Class

There are many flags, collections, and quantities that are derived directly from the parsing of each command. They arise from the interpreted meaning of the symbolic command. For example, every valid command has a corresponding set of tokens representing the various pathnames and options that invoke a program, or flags set by special symbols that indicate concurrency, inter-process communication, or redirection. Every line of text entered on the command line results in a unique combination of these various properties.

Instead of leaving these command traits as global data of the shell program, they are encapsulated into a Command class. The Command class holds all of this data, as well as the methods by which the data members receive their state through the parsing of the command line text. A Command object, then, creates a context for the shell to carry out its purpose. By and large, the non-member functions of the shell program are simply utilities concerned with the proper execution of the command and interfacing with the operating system API. The result is that what would have been many layers of conditional flow control is broken into more manageable units.

Corey Talbert
Assignment 2 Report
10-10-2022
CS 433-01 Operating Systems
Xiaoyu Zhang

Furthermore, the parse routine is subdivided into a few distinct subprocesses. The function parse() begins the process of interpreting the command text. It calls buildPipe() where the pipe symbol “|” is detected and, if found, generates a second Command object representing the consumer process. Note that the new Command object will restart the parsing routine on its own portion of the command text. Next sanitize() is called to remove newline characters, and detect the concurrency symbol “&”, which is also removed. The splitTokens() method converts the cleaned-up command text to an array of strings. The method also checks for redirection. Before completing, parse() saves the command to the shell history.

The history feature as written does not exactly follow the pattern of separation between text parsing and program execution. Some parts of the routine occur during parsing (the member functions of the Command class), and some during execution (the main body of the shell). The problem is that the command stored in history is invoked by another command. The history command must be parsed into a Command object, but the final state of the object depends on the state of the global data of the shell where the command history is saved. Likewise, it is the Command object itself that updates the shell history. Although this set up works, it is messy and blurs the distinction between the meaning of a command and the procedure of the shell. A cleaner process would have all shell history managed outside of the Command class.

Improvements

Currently, the pipe feature does not allow redirection to work in either the consumer or producer portion of the command. Adding this code to the execution of a piped command seems feasible, and would likely require some careful use of file descriptor duplication. Some of the written procedure used for commands without a pipe could be copied and reorganized to achieve this.

A larger command history would be very useful. A structure designed specifically to manage a list of prior commands could be designed, and navigated in the same manner as the bash shell by pressing the arrow keys. That may involve using the POSIX compliant library ncurses, which provides tools for console-based programs.

Command Syntax

>[PROGRAM PATH] [PROGRAM ARGS] [Optional: PIPE ‘|’ + COMMAND] [Optional:REDIRECT ‘<’ or ‘>’ + FILENAME] [Optional:CONCURRENCY ‘&’]

Files

- prog.cpp