Cory Agami, Javier Lores
9/18/2014

Linux Shell Project Report

1 Overview

The goal of this project was to create a working Linux command shell, supportive of full I/O redirection, background processing of commands, up to two pipes, and a novel *'ioacct'* command to see total bytes read/written by a command.

Our implementation utilizes a dynamically resizable array class to handle the arguments, allowing us to have unlimited length of command strings, and makes the program extremely scalable and modular. We've sectioned off all major and most minor processing to separate functions, resulting in extremely modular, scalable, and transportable code.

A comprehensive pseudocode diagram is provided at the end of this report.

2 Design

Designing the program took many iterations and a few overhauls, but resulted in a remarkably clean and compact codebase.

2.1 Dynamic Array Class

Writing our own dynamic array class, complete with create/delete/resize/print functions made reading and parsing arguments and commands effortless and quick, and made cleaning up memory trivial.

2.2 Memory Management

Managing our memory inside the main function was made easy by using as little arguments as possible, and consolidating. Memory cleanup via free() was handled at the end of each execution of our central while loop, while we made sure to set each dynamically allocated variable to NULL upon reset. This coupled with using gdb and valgrind allowed us to keep a close watch on any possible memory leaks, and eliminate them quickly.

2.3 Assumptions

Our program contains user-input validation to check for malformed commands (ie. a redirect with no file, or a pipe with no command) and also ambiguous commands (ie. a pipe and input redirect on the same line).

The only assumptions we rely on are that the user not try to execute commands with more than two pipes, or more than two redirects. Other than that our input validation should be able to catch and prevent and other input errors.

3 DEVELOPMENT PROCESS

We met every day for a week for anywhere from 3 – 8 hours to come up with a good strategy and implement it efficiently and correctly. Of course, there were times when we had to rethink our initial strategy and modify/create/delete/move certain parts of the code.

This was both of our first major projects working in strictly C, and we had some issues managing the memory in the beginning. This evolved into us deciding to create our own argslist class to hold all the commands and arguments after they'd been parsed, and included functions to easily print the contents of a given argslist, to create and delete argslist with correct memory management technique, and to add or remove specific elements.

Also when it came time to implement piping, we had to move a lot of the source code around to have it work properly. We took the main command execution code out of main and put a central execute function that determines whether the given command is a built-in or not, and if there are pipes it determines how many there are (up to 2), and then splits the arguments up into their respective argument lists (3 if you have 2 pipes) and executes. We separated the forking code into its own function to allow for easily creating multiple children, as you need to do with piping.

IOacct and background processing were easy to implement from here; we just added the necessary parsing before passing the arglists to the execution functions. For an ioacct command we simply read rchar and wchar from the processes proc/[pid]/io file and output to the screen, and for a background '&' command, we simply call waitpid() function with WNOHANG specified to return to parent immediately.

3.1 ADDITIONAL ISSUES FACED

Initially we had some issues with background execution, where executing a function in background mode would cause all subsequent functions to be processed in BG mode. To fix this we created a new function reapChildren() to clean up any existing zombie processes with waitpid(WNOHANG) and placed a few calls to it at the beginning, and 'exit' portions of the main while loop.

Another problem faced was with IOacct implementation, where we were trying to get the bytes read/written data from the proc/[pid]/io file of a child… though the code was executing in the parent after the child process had ended, so we couldn't access the file. To circumvent this we simply execute the process in the background and added a while loop to continually read the processes io file until it exited, then we use the last read data as the output. This seemed to work very well to fix our issue.

3.2 DIVISION OF WORK

Cory Agami – Terminal display, parsing input (for commands, arguments, piping), IOacct & background processing, I/O redirection, managing zombie processes, built-in command processing, searching PATH environment variable, Project Report

Javier Lores – Created argslist.c dynamic array data structure for storing our arguments, handled memory leaks with gdb & valgrind, single & dual-piping execution logic, user input validation (for ambiguous/malformed input), code consolidation, README

4 FINAL THOUGHTS

We would have liked to implement piping in a way to allow for infinitely many pipes, and we experimented with ways of doing so and it helped us to structure the code in a much more modular fashion, but for time's sake decided to leave it at supporting two.

Overall we are both very happy with the results of this project. This was each of our first major project to be completely done in C, and we ran up against many unprecedented problems (mainly dealing with memory errors/leaks) but through working together we absolved the issues and learned quite a bit about both memory management and how exactly the back-end of a Linux terminal works. I can say this was a very rewarding project indeed.

Cory Agami, Javier Lores
9/18/2014

5 PSEUDOCODE

```
#includes
function declarations

while true :
        reap children processes (catch possible zombie processes)

        get { hostname, username, current directory, $OLDPWD(NULL to start) }
        print username@hostname prompt

        create 3 arg lists (allocate mem properly, initialize to NULL)
        read sommand line input
                parse out whitespace,
                do input error handling (malformed/ambiguous commands)
                fill first argslist with full command line
        check for IOacct/background process commands, adjust first argslist

        if cmd = built-in (cd, ioacct, exit) :
                if cd :
                        chdir(), handle special cases (-, ~, blank, &)
                if exit[n] :
                        reap child processes, return n;

        else :
                parse out possible pipes in first argslist
                        fill second and third argslists with piped commands

                if no pipes:
                        parse first arglist for redirects, adjust array, store redirect stream
                        search PATH for cmd

                        child = fork()
                        if child == 0 (inside child) :
                                check redirects:
                                        if in redirect, open file RDONLY, dup STDIN, close
                                        if out redirect, creat() file 0644 perms, dup STDOUT, close
                                        if append redirect, open file WRONLY|APPEND, dup STDOUT, close
                                execv(cmd path, args)
                        else if background:
                                waitpid(-1, (int *)NULL, WNOHANG);
                        else if ioacct:
                                read from proc/[pid]/io file while waitpid(WNOHANG) == 0
                                print bytes read/written
                        else (inside parent):
                                waitpid(-1, (int *)NULL, 0)

                if 1 pipe:
```

Cory Agami, Javier Lores
9/18/2014

```
        create pipe int array pipefd[2], call pipe()
        parse second argslist for redirects

        child = fork()
        if child == 0 (inside second argument):
                set input fd to pipefd[0], close fd's
                check redirects:
                        if in redirect, open file RDONLY, dup STDIN, close
                        if out redirect, creat() file 0644 perms, dup STDOUT, close
                        if append redirect, open file WRONLY|APPEND, dup STDOUT, close
                execv(2nd cmd path, args)
        else (in parent):
                child2 = fork()
                if child == 0:
                        set output fd to pipefd[1], close fd's
                        execv(1st cmd path, args)
        close pipes
        call waitpid(-1, (int*)NULL, 0) on both processes

if 2 pipes:
        create pipe in array pipefd[4], call pipe twice
        parse third arglist for redirects

        child = fork()
        if child == 0 (1st cmd):
                set output to pipefd[1], close fd's
                execv(1st cmd path, args)
        else:
                child2 = fork()
                if child2 == 0 (2nd cmd):
                        set input fd to pipefd[0], output to pipefd[3], close fd's
                        execv(cmd path 2, args)
                else:
                        child3 = fork()
                        if child3 == 0 (3rd cmd):
                                set input to pipefd[2], close fd's
                                do redirect fd dup2's
                                execv(cmd path 3, args)
        close pipes
        call waitpid(-1, (int*)NULL, 0) on all 3 processes
```