CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

*Jack Parker*

*jparke10*

*951871227*

**Pseudo-Shell Report**

**Introduction**

This project’s primary goal was to build a shell-like command line interface, which can execute commands in a way like the built-in implementations supplied with standard Unix shells like *bash* and *zsh*. This pseudo-shell allows an end user to perform a limited set of critical system functions, such as navigating a file system, manipulating said file system, and performing input/output operations. The shell can also interpret and handle inconsistencies in users’ commands, from typos to incorrect parameters, to allow a safer user experience.

**Background**

I used standard methods throughout my pseudo-shell project – the shell program uses a simple endless loop, reads from *stdin* (or a given file pointer if executed with the ***-f*** flag) and parses commands using a series of tokenizing functions. Once a command is found in the user’s input (by comparing the string to an array of character pointers), the associated function is executed. Built-in command functions utilize system call POSIX wrappers (in the case of *open()*, *read()*, *unlink(),* and *write()* ) as well as the standard C library’s indirect system call function, *syscall()*, which is unfortunately limited in portability but enables more useful syscalls like *getdents*. If a built-in command has an output of some kind, such as is the case with *cat*, output will be printed to the terminal (using the *stdout* stream) for the end user’s convenience.

**Implementation**

I approached my implementation of the pseudo-shell program by breaking it into four “stages”. I wanted to implement the endless loop, handle the process of parsing user input (through tokenization), identify the given command from the tokenized input, and execute the routine of system calls from the command’s associated function. I implemented the endless loop using a *while* with the condition of the function getline(3) reaching EOF on either *stdin* or a FILE\* pointer when executed with the *-f* flag. Tokenization of the user’s input string allowed some simple recycling of code from the previous CS 415 Lab 1 assignment, where repeated calls are made to *strtok\_r* to create an array of character pointers containing each delimited token.

**A group of black and blue text

Description automatically generated**My toughest problem to solve by far was connecting the dots from the tokenized string input to the actual command functions, whose prototypes are defined in *command.h*. Within the confines of the assignment, I couldn’t modify the function parameters of said prototypes, and because C contains no polymorphism of any kind, it made implementing the function calls using pointers quite tough. I ended up using what feels a bit like a duct-tape solution, but is much cleaner than using an exhaustive *switch* statement referring to each command:

A computer screen shot of a code

Description automatically generatedA computer code with many lines

Description automatically generated with medium confidenceSome system calls were executed using the syscall(2) library function, which means there’s a possibility of code needing to be reconfigured for different Linux distributions.

**Performance Results and Discussion**

Overall, program (on my machine) performs as expected within the confines of the assignment. The only strict “speed” performance metric I had to make an executive decision on was the *cat* command’s routine to print to the console. I ended up opting to have the *read()* system call load the entire text file into memory, as reading and writing files to FILENO\_STDOUT one byte at a time was noticeably slowed, to the point you could see the individual characters scrolling in the terminal. I believe there’s also a slight possibility that this program may not compile on a system architecture other than arm64 Debian (on my M2 MacBook Air), due to the system calls used.

**Conclusion**

I’m overall happy with what I learned in this project, I feel it was a good exercise in writing a user-facing program from start to finish, while many projects in previous CS classes have come with a large amount of starter code which made the troubleshooting and debugging process trivial.

A black screen with white text

Description automatically generated