CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Joseph Erlinger
jerling2
951917289

Report

Introduction

In a restaurant, a customer does not lecture the chef on how to make their food, unless they want their date to leave. Instead, the customer politely asks the server for spaghetti marinara with meatballs. Similarly, a user of a computing system will find it impractical if they had to tell the operating system every step of creating a new file. The user will find it easier to order the operating system to create a new file and for it to handle all the technical details of doing so. To achieve this desired effect, the user sends a command to an interface which translates the command to the operating system. The interface that performs said operation is known as the shell.

In UNIX systems, the shell is a command-line interface (CLI) that accepts text-based commands. Each command line can contain multiple commands that are separated by a delimiter. Additionally, the shell is capable of loading shell scripts, which are files containing CLI commands. Each command can support a variable number of options and a variable number of input parameters, and generally follows the format: `command [options] [arguments]`. If the user's command is unrecognized or incorrectly formed, then the shell will notify the user of the error in their request.

For this project, the task is to create a simpler shell known as a "Pseudo-Shell". The pseudo-shell only knows eight commands (ls, pwd, mkdir, cd, cpFile, mvFile, and cat) and each command can have up to two arguments but no options. Each command line can contain multiple commands that are separated by a semi-colon as a delimiter. The pseudo-shell is capable of loading shell scripts and accepting user commands from the terminal. If the command is unrecognized or incorrectly formed, then the pseudo-shell will immediately stop processing the rest of the command line and notify the user "Error! Unrecognized command" or "Error! Invalid parameters for command".

Background

This project uses the C programming language, and its UNIX-like system calls, to manage the filesystem and file I/O. Specifically, this project does not use standard C library functions for file management in the algorithms that process shell commands. However, this project does use standard C library functions for string manipulation, tokenizing text-based commands, and file I/O for algorithms that load shell scripts and user input.

Algorithms that process shell commands use system calls to interact with the operating system and to validate inputs at the same time. System calls are robust because they accept any input, and the operating system will do the input validation. If the input to a system call is invalid, then it will return an error; otherwise, it will return a success. This property of system calls simplifies the validation and execution of shell commands.

The string tokenizing algorithm processes a text-based command using standard C library functions. The standard C library contains powerful commands for splitting strings on a delimiter. Allowing the string tokenizing algorithm to focus more on the logical aspects of string tokenizing rather than the mechanical aspects.

The algorithm that loads user input does so by reading from stdin. Likewise, the algorithm that loads shell scripts reads a file stream and redirects stdout to an output file stream. Both these processes take advantage of the standard C library functions for I/O.

Implementation

(See **Figure 3** for an overview of the system's architecture) The Pseudo-Shell begins at *main* where the user can specify to run Pseudo-Shell in either FILEMODE by including the -f option followed by a filename of a shell script file, or InteractiveMode by default. *Main* checks for the the -f option with getopt. If *main* detects the -f option, then it will use freopen to create an output.txt and redirect stdout to it. *Main* eventually transfers control to InteractiveMode or FileMode.

InteractiveMode and FileMode are similar in that they both loop forever reading text-based commands from a stream until receiving a signal from an exit command or EOF condition. The difference is that InteractiveMode prompts the user *">>>"* and reads from stdin while FileMode reads from a file stream. Both InteractiveMode and FileMode give CommandLineInterface a text-based command line string.

CommandLineInterface (CLI) accepts a text-based command string and converts it into a tokenized command line using StringFiller. Since a single command line can contain multiple commands, CLI calls StringFiller twice, first to split the string on semi-colons, and second to split each command on spaces. The CLI then passes each tokenized command to the CommandInterpreter for execution.

StringFiller tokenizes a string by using strtok and CountToken returns an integer value of the number of tokens plus one contained in a string. CountToken is more complicated than StringFiller because it must consider more cases, whereas StringFiller was like the strtok example in the Linux manual. **Figure 1** illustrates the CountToken algorithm. In the figure, *d* is the variable "if character at position *i* is a delimiter", and EOS means "end of string".

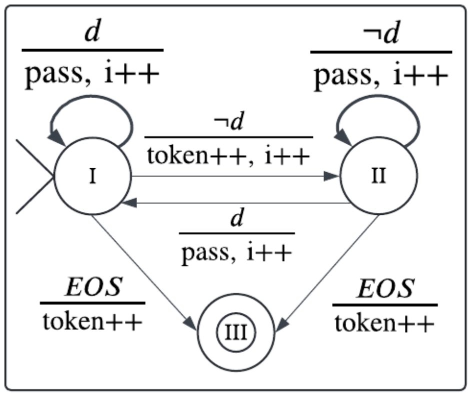

$\frac{d}{\text{pass, i++}}$
$\frac{\neg d}{\text{pass, i++}}$
$\frac{\neg d}{\text{token++, i++}}$
I
II
$\frac{d}{\text{pass, i++}}$
$\frac{EOS}{\text{token++}}$
$\frac{EOS}{\text{token++}}$
III

Figure 1: DFA for Algorithm CountToken.

CommandIntepreter (CI) uses a tokenized command to call ExecuteCommand. The CI is basically a switch-case statement. It knows ten commands, where the ninth and tenth commands are EXIT and DNE. The CI matches the first token to one of its known commands. If the command matches "exit", then the CI returns an EXIT signal to break out of the main loop. If the command does not match any of the known commands, then the CI writes "Error! Unrecognized command: [command]". Otherwise, the CI passes the rest of the tokens to ExecuteCommand along with a function pointer to the command to be executed.

ExecuteCommand is the primary effect of the Pseudo-Shell. It activates one of the eight command routines. If the command routine fails (observed by the errno variable), then ExecuteCommand writes "Error! Unsupported parameters for command: [command]".

The commands in *command.c* were straightforward to implement by calling the appropriate system calls. Except for CopyFile, because there were many cases to consider when determining where the destination file should be located. **Figure 2** shows the DFA I created to help me wrap my head around the many cases. I recommend reading my comments in the source code for more detail.

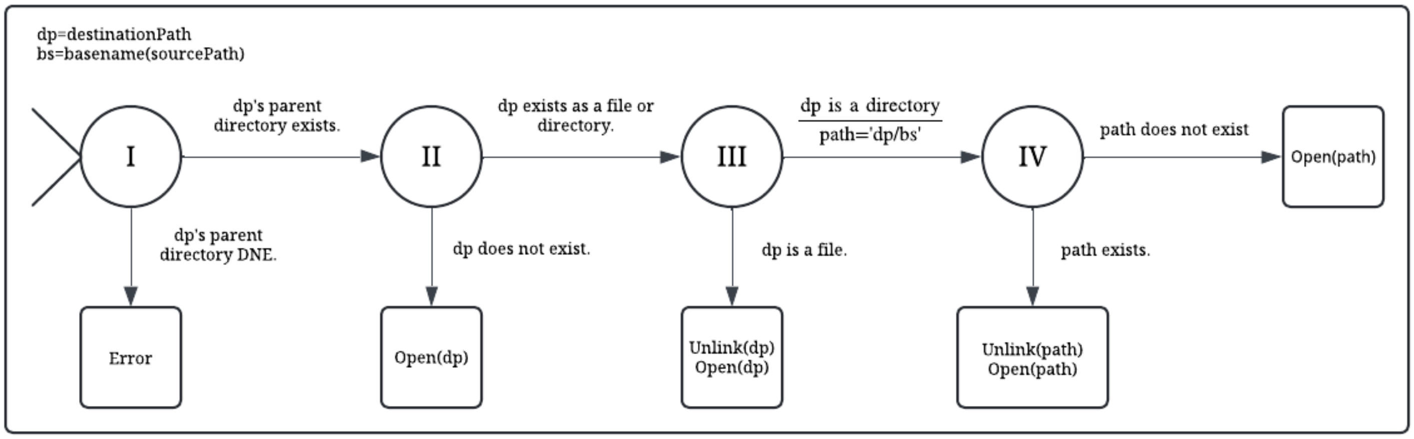


Figure 2: DFA for Algorithm OPENDESTINATIONFILE.

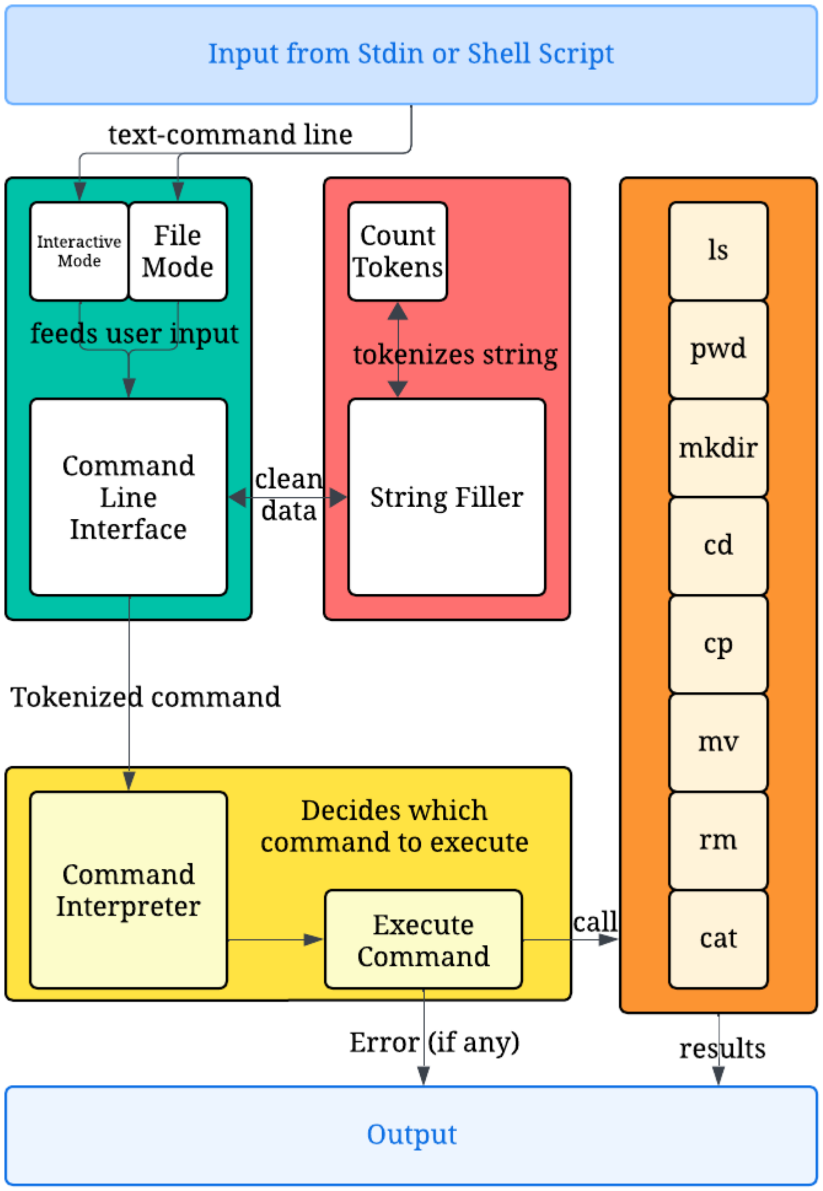
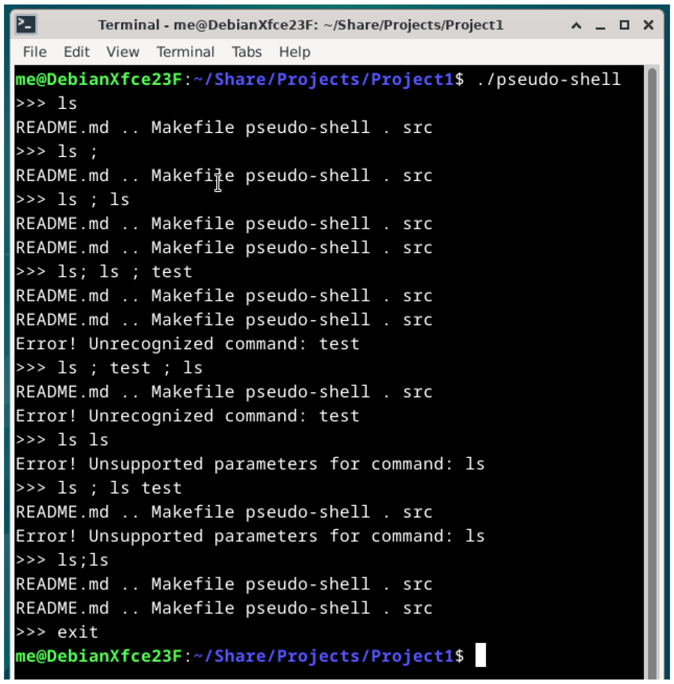


Figure 3: Overview of Pseudo-Shell's software architecture.

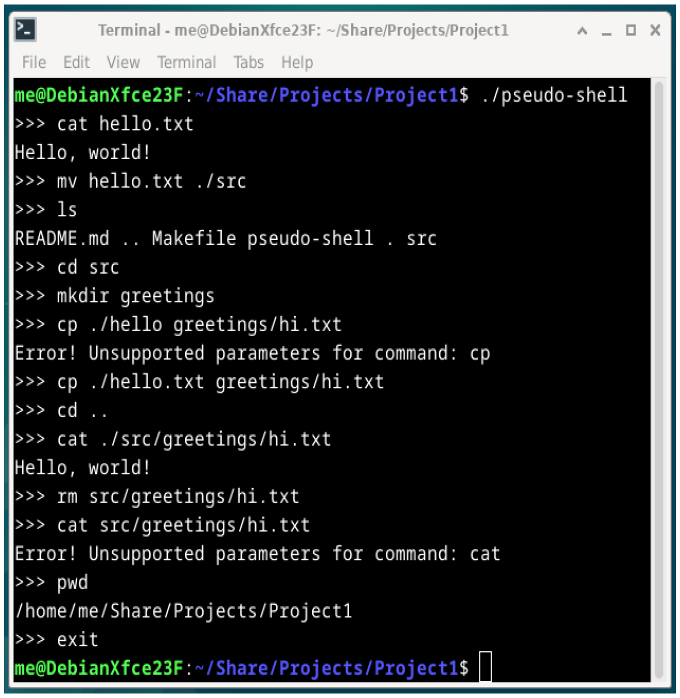
Performance Results and Discussion

The Pseudo-Shell matches the expected behavior in the project description **(Figure 4, Figure 5)**. Furthermore, the Pseudo-Shell passes all tests in Alex's test script **(Figure 7)**. Therefore, all Pseudo-Shell commands should be working as expected, and the Pseudo-Shell should not contain any memory leaks or memory errors. Additionally, the Pseudo-Shell handles edge cases **(Figure 6)** that are not included in the test script such as "`cp file …/`" and "`mv file /`" which should return a "directory does not exist" error and "permission denied" error respectively. Also, one of my personal favorites, the edge case "`mv file ./`" should not behave like "`rm file`" and delete the newly created file.


Terminal - me@DebianXfce23F: ~/Share/Projects/Project1
File Edit View Terminal Tabs Help

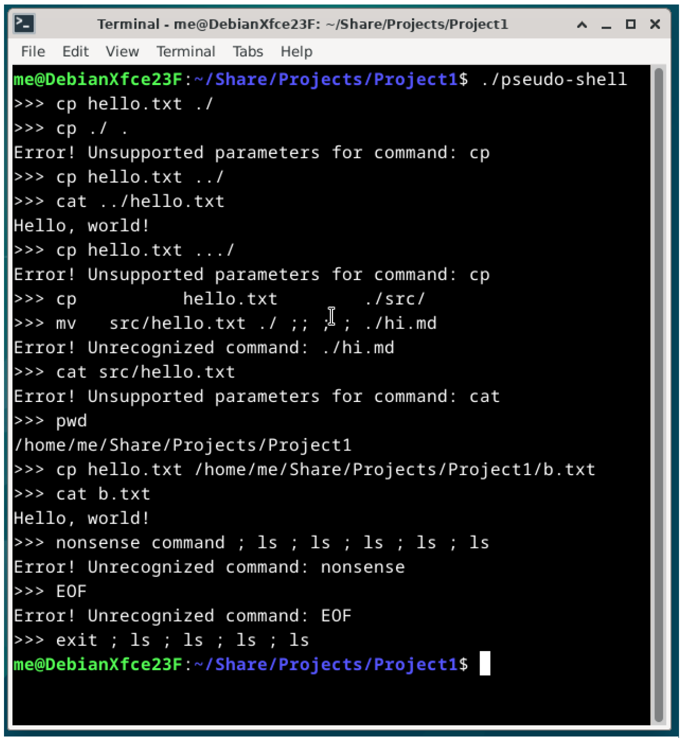
```
me@DebianXfce23F:~/Share/Projects/Project1$ ./pseudo-shell
>>> ls
README.md .. Makefile pseudo-shell . src
>>> ls ;
README.md .. Makefile pseudo-shell . src
>>> ls ; ls
README.md .. Makefile pseudo-shell . src
README.md .. Makefile pseudo-shell . src
>>> ls; ls ; test
README.md .. Makefile pseudo-shell . src
README.md .. Makefile pseudo-shell . src
Error! Unrecognized command: test
>>> ls ; test ; ls
README.md .. Makefile pseudo-shell . src
Error! Unrecognized command: test
>>> ls ls
Error! Unsupported parameters for command: ls
>>> ls ; ls test
README.md .. Makefile pseudo-shell . src
Error! Unsupported parameters for command: ls
>>> ls;ls
README.md .. Makefile pseudo-shell . src
README.md .. Makefile pseudo-shell . src
>>> exit
me@DebianXfce23F:~/Share/Projects/Project1$
```

Figure 4: Matches the expected output in the project description.


Terminal - me@DebianXfce23F: ~/Share/Projects/Project1
File Edit View Terminal Tabs Help

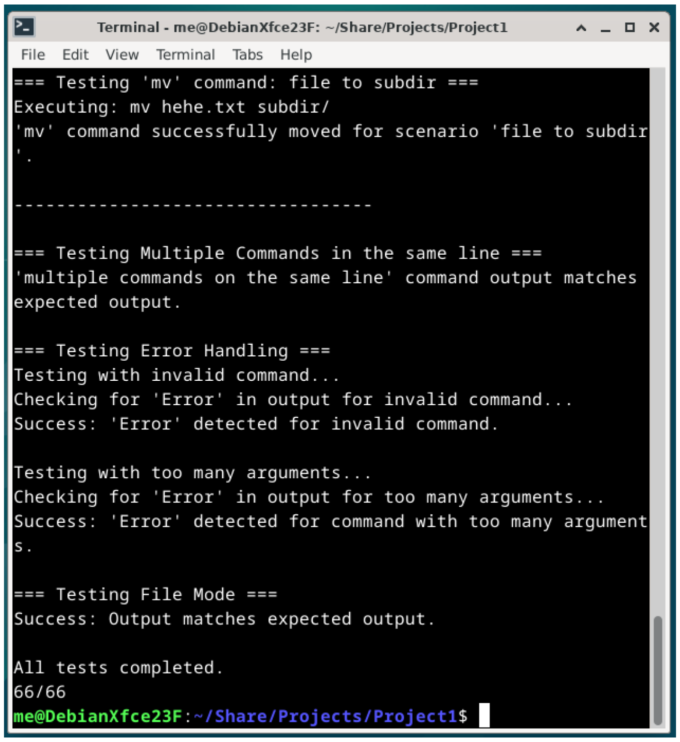
```
me@DebianXfce23F:~/Share/Projects/Project1$ ./pseudo-shell
>>> cat hello.txt
Hello, world!
>>> mv hello.txt ./src
>>> ls
README.md .. Makefile pseudo-shell . src
>>> cd src
>>> mkdir greetings
>>> cp ./hello greetings/hi.txt
Error! Unsupported parameters for command: cp
>>> cp ./hello.txt greetings/hi.txt
>>> cd ..
>>> cat ./src/greetings/hi.txt
Hello, world!
>>> rm src/greetings/hi.txt
>>> cat src/greetings/hi.txt
Error! Unsupported parameters for command: cat
>>> pwd
/home/me/Share/Projects/Project1
>>> exit
me@DebianXfce23F:~/Share/Projects/Project1$
```

Figure 5: Showing all the commands.


Terminal - me@DebianXfce23F: ~/Share/Projects/Project1
File Edit View Terminal Tabs Help

```
me@DebianXfce23F:~/Share/Projects/Project1$ ./pseudo-shell
>>> cp hello.txt ./
>>> cp ./ .
Error! Unsupported parameters for command: cp
>>> cp hello.txt ../
>>> cat ../hello.txt
Hello, world!
>>> cp hello.txt .../
Error! Unsupported parameters for command: cp
>>> cp          hello.txt        ./src/
>>> mv   src/hello.txt ./ ;; ; ; ./hi.md
Error! Unrecognized command: ./hi.md
>>> cat src/hello.txt
Error! Unsupported parameters for command: cat
>>> pwd
/home/me/Share/Projects/Project1
>>> cp hello.txt /home/me/Share/Projects/Project1/b.txt
>>> cat b.txt
Hello, world!
>>> nonsense command ; ls ; ls ; ls ; ls ; ls
Error! Unrecognized command: nonsense
>>> EOF
Error! Unrecognized command: EOF
>>> exit ; ls ; ls ; ls ; ls
me@DebianXfce23F:~/Share/Projects/Project1$
```

Figure 6: Stress testing on edge cases.


Terminal - me@DebianXfce23F: ~/Share/Projects/Project1
File Edit View Terminal Tabs Help

```
=== Testing 'mv' command: file to subdir ===
Executing: mv hehe.txt subdir/
'mv' command successfully moved for scenario 'file to subdir
'.

---------------------------------

=== Testing Multiple Commands in the same line ===
'multiple commands on the same line' command output matches
expected output.

=== Testing Error Handling ===
Testing with invalid command...
Checking for 'Error' in output for invalid command...
Success: 'Error' detected for invalid command.

Testing with too many arguments...
Checking for 'Error' in output for too many arguments...
Success: 'Error' detected for command with too many argument
s.

=== Testing File Mode ===
Success: Output matches expected output.

All tests completed.
66/66
me@DebianXfce23F:~/Share/Projects/Project1$
```

Figure 7: Passes Alex's test script.

Conclusion

The Pseudo-Shell is a story about a translator. On the surface, the Pseudo-Shell translates human commands into machine commands, and machine output into readable text. At the lower level, the Pseudo-Shell performs many steps of data manipulation and error handling to return the desired result. The interface paradigm such as the pseudo-shell is necessary to keep modular systems flexible. Without interfaces, every user will have to know the technical details of every system they interact with (or vice versa).

This project has taught me how to use system calls, Makefiles, and some bash programming. Learning Makefiles and bash programming was a side effect of testing and development. For instance, if I wanted to add another file to the system, then I would need to update the Makefile. Additionally, if I wanted to streamline tests, then I would need to program a shell script. On the other hand, system calls were something I learned actively by mostly reading the Linux manual pages. I was pleasantly surprised to learn that system calls were not too complicated. Since the operating system does most of the work, the programmer's job is to format the inputs and catch errors the operating system spits out. One idea I am curious about is how well my system can translate to multi-threading. My thought is probably not well, because most of the error handling is done by checking the global errno variable. This makes me suspect that handling errors will be a large task for multi-threaded systems.