SPL-1 Project Report, 2023

Idiosyncratic Linux Shell

SE 305 : Software Project Lab I

Submitted by

Mahfuz Sarker Shykot

Roll # BSSE 1328

Session # 2020-21

Supervised by

Dr. Sumon Ahmed

Associate Professor

Institute of Information Technology

University of Dhaka

Institute of Information Technology

University of Dhaka

21-05-2023

Table of Contents

1. Introduction

In everyday life, we encounter different kinds of software. Most of us are concerned about how these software work, how they are built, and so forth. But the topic that is least talked about is the operating system (OS). We hardly care about the operating system, since we all are by default Windows users. But there is a lot more to explore than just going on with Windows, and, the reason why we should care about the operating system is because of the diverse usage of electronic devices and services. The operating system of each device is responsible for system services, required software libraries, run-time components, and device drivers. We are concerned about software but without this essential hardware-related software (firmware) our device is nothing but a blank box. So, the next question arose, how to explore the wide domain of operating systems? To start exploration with the operating system, Linux would be the one-stop solution for beginners. Due to being an open-source operating system, it is easy to extract the implementational details which will help us to explore it more minutely than any other operating system.

In an operating system, the kernel plays a vital role to control all system-related activities. Surprisingly it is true that whatever software we use, they communicate with the kernel and the kernel accomplishes the specific task on behalf of user requests. Again, it raises another question, how do this kernel and software are connected and communicate with each other? Well, in that case, the shell plays a key role in making the connection between the kernel and the software. Shell is an interface present between the kernel and the user. It is primarily used to access the services provided by the operating system. Now, what if we have a shell that is capable of communicating with the kernel? It would be great, as it will clarify our concept of the operating system and its working principle. And this project is all about making a shell from scratch. This project will make one confident to work with the operating systems as it has demonstrated very core and common features of the shell. As building a shell from scratch is a tedious task, we have tried to focus on the common aspects of a shell and get into the minimal details that are easy to comprehend for beginners.

In general, a shell contains numerous features including shell operations, functions, redirections, command executions, expansions, and so forth. As this project is designed for beginners, we will cover only the basic features including shell command execution, command parsing, and tokenization, shell expansions, pipelining, input-output redirection, command corrections. Therefore, this shell will be able to handle multiple argument-based commands, pipelined commands, wildcard-based statements, and aliasing frequently used commands. Shell is primarily built with C and thus so this project, so would be quite relevant to the actual shell program. Users can easily relate its functional details with that of the actual shell.

The shell comes in different flavours: Bourne shell (bsh), Bourne-again Shell (bash), Korne shell (ksl), C shell (csh), and many more variants. This project is fully focused on bash. Therefore, we will implement a basic bash in this project.

1.1 Background Study

Before we deep dive into the details of the project, some terms should be clarified for proper comprehension of the implementation. Here we will jot down some core and important concepts and terminologies related to that project.

- **Command**: Shell commands are a sequence of strings that represent some instructions that instruct the system to perform specific actions. Shell commands consist of some arguments that are delimited by space and preceded by the actual command.

  Shell commands primarily come into two types; simple and complex commands. Simple commands are nothing but some arguments delimited by space or semicolons. And when multiple simple commands are concatenated with pipeline, redirections, looping, or grouping cumulatively, it is called complex commands. In this project, we will see both of these implementations.

- **Wildcard**: This is commonly known as a pattern-matching character. When we need to perform file name or directory name substitution, we use the wildcard character. The basic wildcard character includes '?', '*', '!'.

- **Sell Session**: Session denotes the current environment or state of the terminal when we start executing the shell program. In simple terms, when a user starts a shell in the terminal and waits till exiting the program, the whole duration is noted as a shell session. One can have only one shell session at a time.

- **Pipeline**: Pipeline is one of the core features of the Unix shell. It enables the connection of the output of one program to the input of another. It is denoted as a pipe ‘|’ symbol. Using this feature, we can pass the input or output to another command.

- **Tokens**: When a group of characters forms a collective meaning, then this sequence of characters is called token. Tokens are also defined as a single logical unit. It can be found in different varieties. Some of the common tokens are identifiers, keywords, operators, and special characters.

- **Parser**: Parser is a process of analyzing a sequence of strings. It is also known as syntax analysis or syntactic analysis. It takes a string, breaks them into tokens and the output is shown as a form of the parse tree.

2. Project Overview

My project is designed to demonstrate the basic features of the Linux shell (bash). It will act as a clone of bash and it includes the following attributes:

- Executing Shell Commands
  - ✓ Read a user input
  - ✓ Breaks into tokens
  - ✓ Parsing tokens
  - ✓ Perform shell expansions
  - ✓ Execute commands
- Command Name Customization (aliasing)
- Pipelined Command Execution
- Input-Output Redirection
- Command Correction

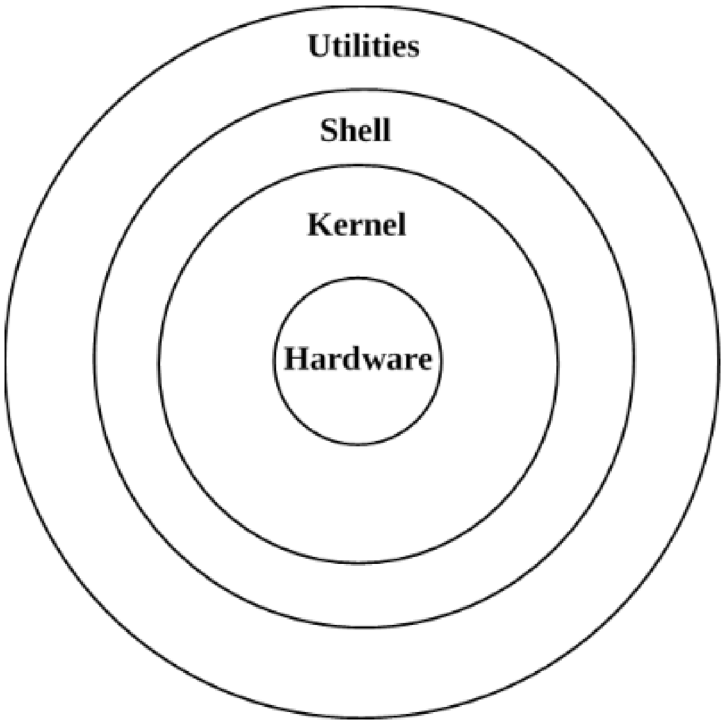
These are the features that have been covered in the whole project. Now let us get into the details of each feature to learn about their working procedures.

2.1 Basics

The project aims to handle some under-the-hood features and algorithms that actually work inside a shell. How are the commands executed properly after entering the terminal window? How are extra features like keeping the history of commands and showing help handled? All of this can be understood by creating an idiosyncratic shell.

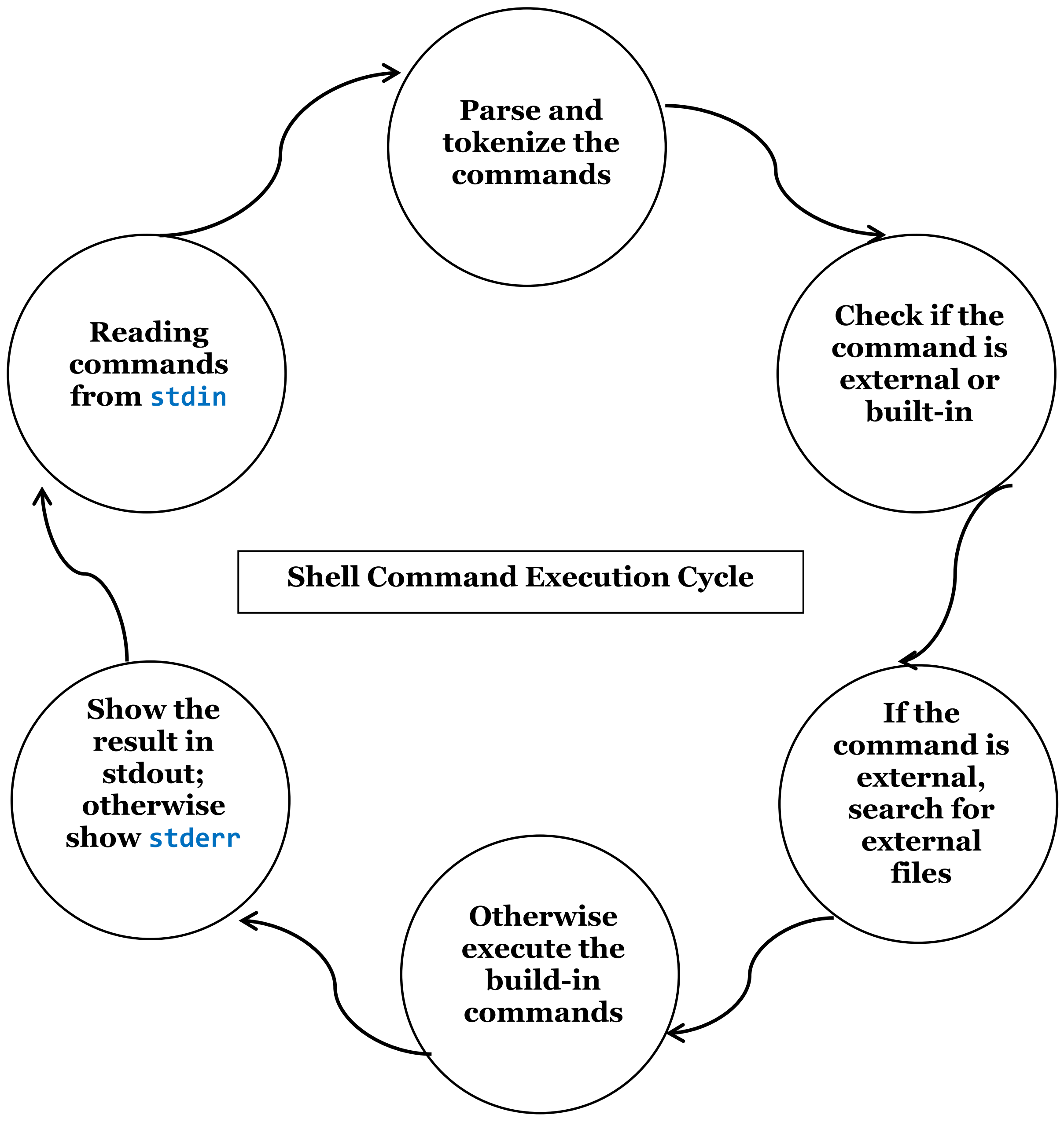
A shell is a special user program that provides an interface for the user to use OS services and accept human-readable commands from a user and convert them into something which can be understood by kernel – a computer program that is the core of an OS, with complete control over everything in the system managing file, process, I/O, memory, etc.

Linux OS = Kernel + GNU system utilities and libraries + Other management scripts + Installation scripts



2.2 Executing Shell Commands

This is the core feature of my project where we will execute shell commands following some sequential process. Initially, the shell asks for the user input. Then this input will be broken into tokens so that necessary aliasing and command expansion can be performed. The tokens will be parsed to create relevant shell commands. Here these commands may include pipelining or command redirections. Later, it will be checked if the command is built-in or external. If it is built-in then execution will be redirected to system calls. Otherwise, for external commands, the shell will search the executable file for further execution. After the execution is complete, the user will be asked for the next command again and this Read-Evaluate-Print-Loop (**REPL**) will be continued till the user exit the shell. This whole process is summed up in the following figure.


Parse and tokenize the commands
Check if the command is external or built-in
If the command is external, search for external files
Otherwise execute the build-in commands
Show the result in stdout; otherwise show stderr
Reading commands from stdin
Shell Command Execution Cycle

2.3 Command Name Customization (Aliasing)

Users can assign short names or refactor shell commands through aliasing. By using aliases, users can save a lot of time while doing tasks frequently. In that feature, we split the command arguments into two parts. The first part, followed by the keyword 'alias', is the actual command. And the second part is the aliased command. We map these two commands so that the actual command can be replaced with aliased command before execution is performed.

2.4 Pipelined Command Execution

A pipeline is a collection of one or more instructions partitioned by the '|' control operator. The output of each command in the pipeline is linked to the following command's input through a pipe. It is a very common scenario when we need to execute multiple commands at the same time. And pipelined command meets this requirement to execute multiple commands simultaneously. We have used the pipe function to link the input-output of multiple commands. And to represent the standard input-output stream we have used the file descriptors. These descriptors act as the placeholder for `stdin`, `stdout`, and `stderr`.

2.5 Input Output Redirection

Redirection allows us to change the standard input (`stdin`) and standard output (`stdout`) while executing shell commands in the terminal. It works with commands' file handles. This file handle can be replicated, opened, or closed and even can change the file the command reads from and writes to. Before implementing this feature, we have to extract the input-output file names as well as their relevant redirection sequence. This is the most challenging part of that feature. Later on, after extracting the file names, we use the file descriptors. By replicating the descriptors, we redirect the stdin and `stdout` to the files mentioned in the shell command.

2.6 Command Correction

It is quite common that we type the wrong shell commands with or without the actual command. In that case, this feature will help the user to understand the relevant shell command if s/he puts any wrong commands. This feature is quite useful for those who are new to Linux and learning commands by trial and error. Initially, it will suggest to the user what can be the closest command to the given wrong command. In autocompletion mode, our shell will suggest the most accurate command by inspecting the user command history.

3. User Manual

This section covers the details of how I have implemented the features mentioned above in my project. Each subsection will contain the implementation details.

3.1 Executing Shell Commands

Command execution starts with taking user input. We maintained a while loop that will continuously ask the user to provide input and will continue until the user exits the loop. However, after taking input through the `takeUserInputFunction` function the string is forwarded to the `processPipelinedCommandsFunction` function to check if it contains any pipeline or not. If so, then we go for the `executePipelinedCommandsFunction` function for further command execution. Otherwise, we pass the string to the `porcessSingleCommandsFunction` function for further modification and passed to the execute function for execution. Thus, the whole process of command execution goes on until the user exits the loop and closes our program.

3.2 Command Name Customization (Aliasing)

When a string is passed for parsing, this feature comes to inaction. Initially, we check each parsed token whether contains any alias or not through the `checkForAliasingFunction` function. Then if the token contains any alias, we replace this alias with the actual shell command. In addition, when a user alias commands a set new command, control goes to the `aliasCommandsFunction` function and it processes the set of arguments through the `settingAliasfunction` method for setting new alias commands.

3.3 Pipelined Command Execution

Previously we have seen how a command executes form user given instructions. When the given string is passed to the `processPipelinedCommandsFunction` method, pipeline command execution begins. In further steps, we look for the pipe '|' symbol and redirection symbol '>', '<' for proper command modification. Based on this, we extract the set of arguments and passed the parsed string to the `executePipelinedCommandsFunction` method for final execution. This method deals with both pipelines and redirecting multiple commands. Command redirection will be discussed in detail in the following sub-section. However, in the `executePipelinedCommands` method, we used the concept of file descriptor to denote proper standard input-output. Also, we have used the pipe system method to communicate between newly designated `stdin` and `stdout`.

3.4 Input Output Redirection

Following the previous sub-section, we have learned how the pipeline command has been implemented. Therefore, we may have noticed that while parsing the pipelined command we also check for redirection symbols whether they appear or not. The most crucial part of that feature is parsing accurate commands and I/O files from the given input. And this task is done by the parse method where we distinguish between pipelined command, redirected command, and I/O file. Then these arguments are passed through the `executePipelinedCommandsFunction` method. The reason we have used that particular method twice in two different features is due to their interrelation. While we execute pipeline commands, we deal with multiple commands and in that case, we also need to consider command redirection. Because this feature also deals with multiple commands where instead of multiple commands, here we use the I/O file where `stdin` and `stdout` is redirected.

3.5 Command Correction

The last feature to focus on is command correction. We are already acknowledging from the description that it has two modes one is for basic suggestion mode and the other is autocompletion mode. Both of these modes use a common method, the `BKTreeGenerationFunction` method that implements the **BK-Tree** data structure. This is the initial method where we built a tree data structure that contains the closest command related to the user given wrong commands. In between this method, we invoke the `nodeCreationFunction` method where each node represents a new closest word. Also, to find those closest words, we search through the directories passed to the `readCommandOutputFunction` function. Then we use the `LevenshteinEditDistanceFunction` method (**Levenshtein edit distance algorithm**) to calculate the level of closeness to the command given by the user and this value is put over the tree edges. For autocompletion mode, we add another method called `getAutoCompletedCommands`. This method matches the wrong user command to the closest and most used command of that user collected from the user history. The `frequencyCalculatorFunction` method handles the frequency of the closest and most used command. Thus, we perform command correction in our shell program.

4. Challenges Encountered

While accomplishing the project, I had to face some challenges that were not too easy to resolve. Some of those challenges are listed below:

Understanding system process: As my shell executes different shell commands, there I need to handle system processes. The system process is split into parent and child processes. Whenever we run any process, we execute the child process. Multiple instances of a single process can execute concurrently by different applications, each time a child process is invoked for execution.

- **Dynamic Memory Allocation**

This was a very challenging part of my project because handling memory with `malloc` and `calloc` is quite tricky. Any miscalculation may ruin the whole project's execution. However, one observation that I have found is about the segmentation fault. Segmentation fault was quite frequent and the reason behind is faulty array declaration. Sometimes our array goes out of bounds or we allocate already allocated spaces. In such cases, segmentation faults arose. Therefore, we should be careful while working with character pointers, array declaration, and string manipulation.

- **Parsing**

One of the core parts of my project was parsing users given to extract commands, arguments, and I/O files. I have to handle different cases of input to extract the I/O files. In Input-output redirection, I have handled different cases:

command < infile > outfile

command > outfile < file

command | command > outfile

Therefore, I have to manipulate the parsing process for different combinations of user-given commands.

- **Managing Large Codebase**

Working with a large codebase is a tedious task. It becomes very difficult to track changes on different files. That's why making the required header files and using makefile might lessen your hassle. Makefile makes it easy to compile multiple files even if you have headers or an extra library in your project.

- **Debugging Codes**

Frequently I got segmentation faults or dumped core errors but I wasn’t sure where it happening. Then this issue was instantly solved by debugging the .cpp files.

5. Conclusion

We have discussed how to implement a shell throughout the whole writing. Now, it should be clear how the shell works and how all the commands are being processed. This project helped us to understand the concept of pipelining in command execution, redirection as input-output alteration, the wildcard in the file, or directory name expansion. Besides this, we have also learned how the system process is managed by using fork system calls, how to alias or refactor command statements, and how to display proper suggestions for wrong commands. Overall, from a birds-eye, we covered the most common aspect of shell specifically for a bash.

But that is not the end of our journey. We can still implement shell scripting, shell functions, and conditional in further extension. It will make our shell more relatable to the actual shell environment.

Reference

[1] GNU Operating System; May 20, 2023
https://www.gnu.org/software/bash/manual/html_node/Basic-Shell-Features.html

[2] Computer Hope; May 20, 2023
https://www.computerhope.com/jargon/f/file-descriptor.htm

[3] Linux Hint; May 20, 2023
https://linuxhint.com/fork_linux_system_call_c/

[4] Develop Paper; May 20, 2023
https://developpaper.com/linux-pipeline-communication-c-language-programming-example/

[5] GitHub; May 20, 2023
https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797

[6] Medium; May 20, 2023
https://medium.com/future-vision/bk-trees-unexplored-data-structure-ec234f39052d

[7] Medium; May 20, 2023
https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

[8] Scaler; May 20, 2023
https://www.scaler.com/topics/data-structures/kmp-algorithm/

[9] Programiz; May 20, 2023
https://www.programiz.com/c-programming/c-dynamic-memory-allocation

[10] Tech Mint; May 20, 2023
https://www.tecmint.com/linux-process-management/

THE END