



B.P. PODDAR INSTITUTE OF MANAGEMENT AND TECHNOLOGY

137, V.I.P. ROAD, PODDAR VIHAR, KOLKATA – 700052

DEPARTMENT OF INFORMATION TECHNOLOGY

Term Paper on

“ A Mini Linux Shell”

Operating System (PCC-CS502)

Third Year, Semester V

Academic Year 2020-21

SUBMITTED

BY

Debangsh Bose (11500219055)

Subhradip Barik (11500219056)

Soham Nandi (11500219057)

Randrita Sarkar (11500219058)

Index

<u>Topic Name</u>	<u>Page No</u>
▪ Acknowledgements -----	1
▪ Abstract -----	2
▪ Introduction -----	3
○ About Kernel	
○ CLI & GUI	
○ Basic Lifetime of a Shell	
▪ Description -----	6
○ Lexical Analyser	
○ Parser	
○ Executor	
○ Linux Shell: The Basics	
○ Mini Shell	
○ How Shells Start Process	
▪ Conclusion -----	11
▪ References -----	12

Acknowledgements

We are humbled with gratitude and gratitude to all those who have assisted us in putting this notion, much above the level of simplicity, into something concrete.

We would like to express our heartfelt gratitude to our Operating System teacher, Mr. Balaram Ghosal Sir, as well as our principal, for providing us with the wonderful opportunity to do this wonderful project on the topic "Write a mini Linux Shell," which also assisted us in conducting extensive research and learning about many new things. We are really thankful to you.

Finally, without the help and supervision of our team, no endeavour at any level can be performed satisfactorily. Thank you to everyone else.

Abstract

The project was about writing your own **Linux Shell**. We wanted to work on it as it was the first time we would be developing something of this kind. A mini Linux shell is a program that behaves exactly like the Linux shell, albeit with limited functionality. It supports built-in shell commands like "cd" and "exit". It supports input and output redirection. It supports background process. Since it's a Linux shell, the only feasible language to use was **C**; as it is native to Linux. Also, the *POSIX* library was required for the implementation of the shell.

In this project we will be creating our mini shell adding the feature of "cd", "ls", "help" and "exit"

Introduction

You interact with shell indirectly if you use any major operating system. When you use the terminal in Ubuntu, Linux Mint, or any other Linux distribution, you are dealing with shell. However, before we can comprehend shell scripting, we must first become acquainted with the following terms:

- **Kernel**
- **Shell**
- **Terminal**

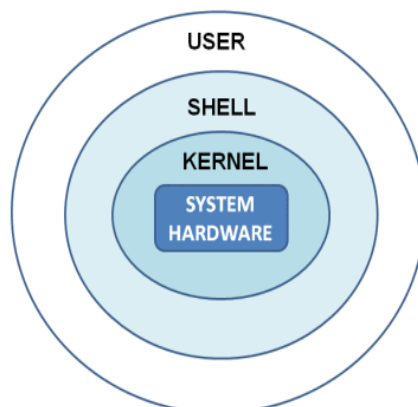
What is kernel?

The kernel is a computer program that serves as the heart of a computer's operating system, controlling everything in the system. It is in control of the following Linux system resources:

- **File management**
- **Process management**
- **I/O management**
- **Memory management**
- **Device management**

What is Shell?

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

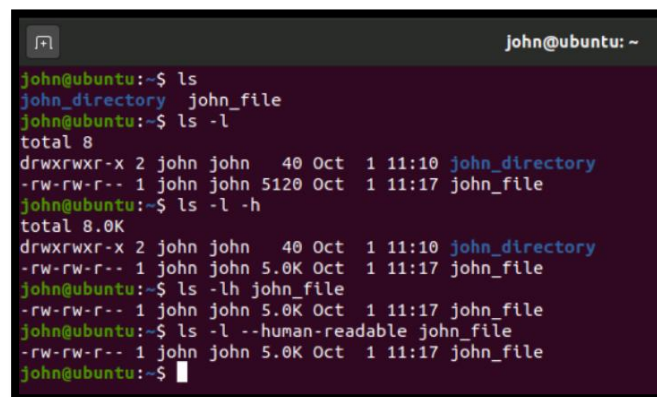


Shell is broadly classified into two categories –

- **Command Line Shell**
- **Graphical shell**

Command Line Shell:

Shell can be accessed by user using a command line interface. A special program called **Terminal** in linux/macOS or **Command Prompt** in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc.



```
john@ubuntu: ~  
john@ubuntu:~$ ls  
john_directory  john_file  
john@ubuntu:~$ ls -l  
total 8  
drwxrwxr-x 2 john john 40 Oct 1 11:10 john_directory  
-rw-rw-r-- 1 john john 5120 Oct 1 11:17 john_file  
john@ubuntu:~$ ls -l -h  
total 8.0K  
drwxrwxr-x 2 john john 40 Oct 1 11:10 john_directory  
-rw-rw-r-- 1 john john 5.0K Oct 1 11:17 john_file  
john@ubuntu:~$ ls -lh john_file  
-rw-rw-r-- 1 john john 5.0K Oct 1 11:17 john_file  
john@ubuntu:~$ ls -l --human-readable john_file  
-rw-rw-r-- 1 john john 5.0K Oct 1 11:17 john_file  
john@ubuntu:~$
```

Graphical Shell:

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.



There are several shells available for Linux systems like –

BASH (Bourne Again SHell) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.

CSH (C SHell) – The C shell's syntax and usage are very similar to the C programming language.

KSH (Korn SHell) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built-in functions.

Basic lifetime of a Shell:

Let's take a look at a shell from top to bottom. During its existence, a shell does three things.

- **Initialize:** A normal shell would read and run its configuration files at this step. These alter certain features of the shell's behaviour.
- **Interpret:** The shell then reads instructions from stdin (which could be interactive or from a file) and executes them.
- **Terminate:** After executing its commands, the shell executes any shutdown commands, frees up any memory, and exits.

These steps are so general that they could apply to many programs. Our shell will be so simple that there won't be any configuration files, and there won't be any shutdown command. So, we'll just call the looping function and then terminate. But in terms of architecture, it's important to keep in mind that the lifetime of the program is more than just looping.



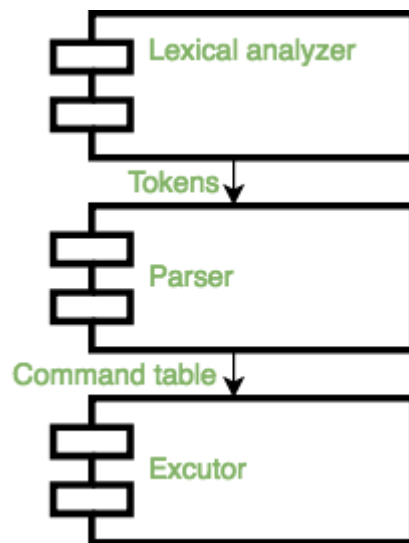
```
GNU nano 2.5.3 File: miniShell.c
int main(int argc, char **argv){
    //load config files if any
    //run command loop
    miniShell();

    //performing shutdown
    return EXIT;
}
```

Description

So we've taken care of how the program should start up. Now, for the basic program logic: what does the shell do? Well, a simple way to handle commands is with three steps:

- **Read:** Read the command from standard input.
- **Parse:** Separate the command string into a program and arguments.
- **Execute:** Run the parsed command.



Lexical Analyzer:

The first part in input parsing is the lexical analysis stage where the input is read character by character to form tokens, we will be using a command called lex to build our file, in this file we will define our pattern followed by the: token name the lexical analyzer will read the input character by character and when the pattern matches the string on the left it will be converted to the string on the right.

ex:

Command input: ls -al

Parser:

After the tokens have been formed from the input, tokens pass as a stream to the parser which parses the input to detect syntax error and execute the assigned semantic actions.

Executor:

After the command table has been built, the executor is responsible for creating a process for every command in the table and handle any redirection if needed.

All Linux operating systems have a terminal window to write in commands. But how are they executed properly after they are entered? Also, how are extra features like keeping the history of commands and showing help handled?

Linux Shell: The Basics

After a command is entered, the following things are done:

1. Command is entered and if length is non-null, keep it in history.
2. Parsing: Parsing is the breaking up of commands into individual words and strings
3. Checking for special characters like pipes, etc is done
4. Checking if built-in commands are asked for.
5. If pipes are present, handling pipes.
6. Executing system commands and libraries by forking a child and calling execvp.
7. Printing current directory name and asking for next input.
- 8.

For keeping history of commands, recovering history using arrow keys and handling autocomplete using the tab key, we will be using the readline library provided by GNU.

Basic Pseudo Code For a Shell

```
int
main (int argc, char **argv)
{
    while (1){
        int childPid;
        char * cmdLine;

        printPrompt();

        cmdLine= readCommandLine();

        if ( isBuiltInCommand(cmdLine)){
            executeBuiltInCommand(cmdLine);
        } else {
            childPid = fork();
            if (childPid == 0){
                exec ( getCommand(cmdLine));
            } else {
                if (runInForeground(cmdLine)){
                    wait (childPid);
                } else {
                    record in list of background jobs
                }
            }
        }
    }
}
```

Based on it we have built our own “Mini Shell”

```
int main()
{
    char inputString[MAXCOM], *parsedArgs[MAXLIST];
    char* parsedArgsPiped[MAXLIST];
    int execFlag = 0;
    init_shell();

    while (1) {
        // print shell line
        printDir();
        // take input
        if (takeInput(inputString))
            continue;
        // process
        execFlag = processString(inputString,
            parsedArgs, parsedArgsPiped);
        // execflag returns zero if there is no command
        // or it is a builtin command,
        // 1 if it is a simple command
        // 2 if it is including a pipe.

        // execute
        if (execFlag == 1)
            execArgs(parsedArgs);

        if (execFlag == 2)
            execArgsPiped(parsedArgs, parsedArgsPiped);
    }
    return 0;
}
```

Shell Output:

```
*****

****Welcome To Mini LinuxShell****

-Created By BPPIMTIANS-

*****
```

```
***Welcome To Mini LinuxShell***
Created by @BPPIMTIANS....
List of Commands supported:
>cdir
>ls
>exit
>all other general commands available in UNIX shell
>pipe handling
>improper space handling
```

```
Dir: /home/randritas
>>> hi

Hi randritas.
Thank You for using Mini LinuxShell
Use help to know more..

Dir: /home/randritas
>>> cdir x.txt

Dir: /home/randritas/x.txt
>>> █
```

How Shells Start Processes:

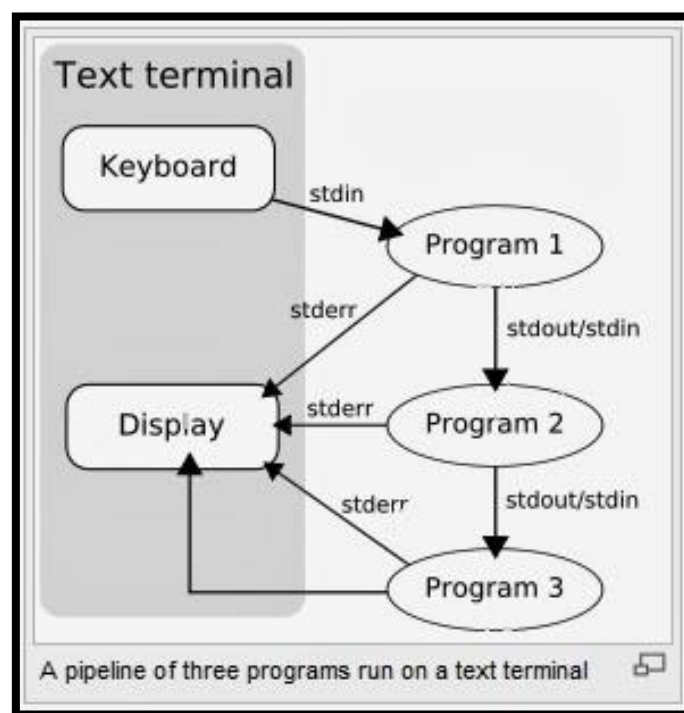
Starting processes is the main function of shells. So writing a shell means that you need to know exactly what's going on with processes and how they start. That's why I'm going to take us on a short diversion to discuss processes in Unix-like operating systems.

There are only two ways of starting processes on Unix. The first one (which almost doesn't count) is by being Init. You see, when a Unix computer boots, its kernel is loaded. Once it is loaded and initialized, the kernel starts only one process, which is called Init. This process runs for the entire length of time that the computer is on, and it manages loading up the rest of the processes that you need for your computer to be useful. Since most programs aren't Init, that leaves only one practical way for processes to get started: the `fork()` system call.

When this function is called, the operating system makes a duplicate of the process and starts them both running. The original process is called the "parent", and the new one is called the "child". `fork()`

returns 0 to the child process, and it returns to the parent the process ID number (PID) of its child. In essence, this means that the only way for new processes to start is by an existing one duplicating itself. This might sound like a problem. Typically, when you want to run a new process, you don't just want another copy of the same program – you want to run a different program. That's what the `exec()` system call is all about.

It replaces the current running program with an entirely new one. This means that when you call `exec`, the operating system stops your process, loads up the new program, and starts that one in its place. A process never returns from an `exec()` call (unless there's an error). With these two system calls, we have the building blocks for how most programs are run on Unix. First, an existing process forks itself into two separate ones. Then, the child uses `exec()` to replace itself with a new program. The parent process can continue doing other things, and it can even keep tabs on its children, using the system call `wait()`.



Conclusion

Shell is very important to create your own command or to run multiple commands with one script file to perform various functions. In this project we grasp the basic idea of shell and its functionality. We learned how to built your own function in shell and how to execute it. Moreover, we understood the usage of variables in shell scripting. Each shell does the same job but understand different commands and provide different built in functions. Shell is very handy to accomplish different useful tasks, and there is a lot to uncover.

References

- <https://www.geeksforgeeks.org/introduction-linux-shell-shell-scripting/>
- [https://www.geeksforgeeks.org/fork-system-call/#:~:text=Fork%20system%20call%20is%20used,the%20fork\(\)%20system%20call.](https://www.geeksforgeeks.org/fork-system-call/#:~:text=Fork%20system%20call%20is%20used,the%20fork()%20system%20call.)
- <https://github.com/linuxartisan/basicshell>
- <https://linuxhint.com/create-simple-shell-scripts-linux/>
- <https://neilkakkar.com/unix.html>

Thank You!