MINI LINUX SHELL

By: Aditya Shinde

Problem Statement:

Write a mini Linux Shell : having features like running the basic commands such as working pipes(from a single pipe to multiple pipes), the redirection operators (>, <, etc), introducing some control characters, remembering the history of the commands.

Project Objective :

Objective of the project is to build a mini linux shell which will replicate the features of bash shell.

Introduction:

- **What is Shell?**

A shell is an interface that allows you to interact with the kernel of an operating system.A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

- **How Does a Shell Work?**

Every shell has its own language syntax and semantics. In the standard Linux shell, bash, a command line has the form:

command [arg1] [arg2] ... [argN]

in which the first word is the command to be executed and the remaining words are arguments expected by that command. The number of arguments depends on which command is being executed. For example, the directory listing command may have no arguments-simply by the user's typing “ls” or it may have arguments prefaced by the negative “-“ character, as in “ls –al”, where “a” and “l” are arguments. The command determines the syntax for the arguments, such as which of the arguments may be grouped (as for the “a” and “l” in the “ls” command), which arguments must be preceded by a "-" character, and whether the position of the argument is important.

The command for the command line is usually the name of a file that contains an executable program, for example, “ls” and “g++” (files stored in /bin on most UNIX-style machines). In a few cases, the command is not a filename but rather a command that is implemented within the shell. For example, “cd” (change directory) is usually implemented within the shell itself rather than in a file in /bin. Because the vast majority of the commands are implemented in files, you can think of the command as actually being a filename in some directory on the machine.

Following steps that a shell must take to accomplish its job

1. *Print a prompt*
2. *Get the command line.*
3. *Parse the command.*
4. *Find the file.*
5. *Prepare the parameters.*
6. *Execute the command.*

The Bourne shell uses multiple processes to accomplish this by using the UNIX-style system calls fork(), execvp(), and wait().

Concepts Used:

A. I/O Redirection

A process, when created, has three default file identifiers: stdin, stdout, and stderr. These three file identifiers correspond to the C++ objects cin, cout, and cerr. If the process reads from stdin (using cin) then the data that it receives will be directed from the keyboard to the stdin file descriptor. Similarly, data received from stdout (using cout) and stderr (using cerr) are mapped to the terminal display. The user can redefine stdin or stdout whenever a command is entered. If the user provides a filename argument to the command and precedes the filename with a “less than” character "<” then the shell will substitute the designated file for stdin; this is called redirecting the input from the designated file.

The user can redirect the output (for the execution of a single command) by preceding a filename with the right angular brace character, ">" character.
The user can redirect the output (for the execution of a single command) by preceding a filename with the right angular brace character, ">" character.

B. Shell Pipes

The pipe is a common IPC mechanism in Linux and other versions of UNIX. By default,a pipe employs asynchronous send and blocking receive operations.Optionally, the blocking receive operation may be changed to be a non-blocking receiver. Pipes are FIFO (first-in/first out) buffers designed with an API that resembles as closely as possible a low level file I/O interface. A pipe may contain a system-defined maximum number of bytes at any given time, usually 4KB. A process can send data by writing it into one end of the pipe and another can receive the data by reading the other end of the pipe.

Methodology:

A. System calls Used

1. fork()

->

The fork() system call creates a new process that is a copy of the calling process, except that it has its own copy of the memory, its own process ID (with the correct relationships to other processes), and its own pointers to shared kernel entities such as file descriptors. After fork() has been called,

two processes will execute the next statement after the fork() in their own address spaces: the parent and the child. If the call succeeds, then in the parent process fork() returns the process ID of the newly created child process and in the child process, fork() returns a zero value.

2. **execvp()**

   ->

   The execvp() system call changes the program that a process is currently executing. It has the form:

   execvp(char* path, char* argv[]);

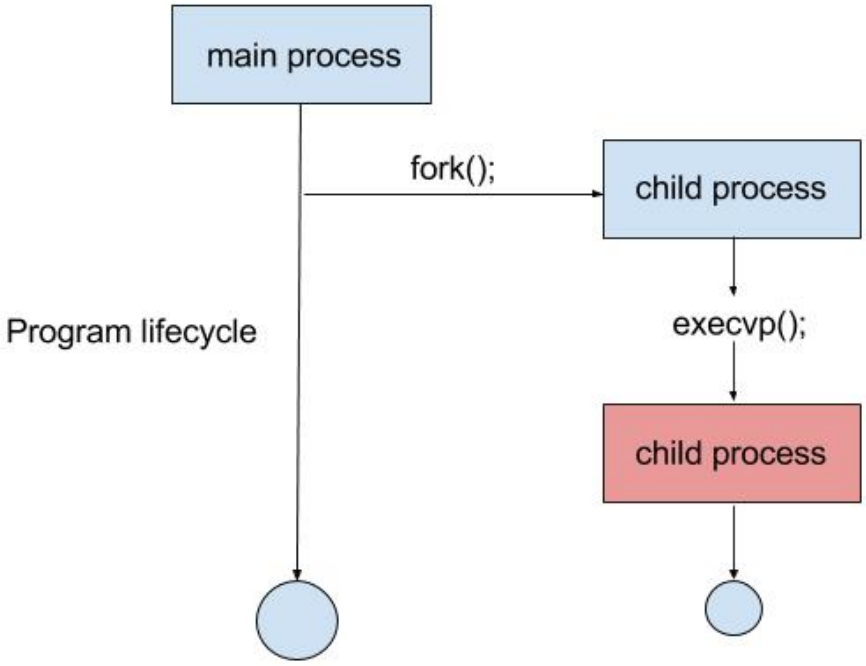
   The path argument is the pathname of a file that contains the new program to be executed. The argv[] array is a list of parameter strings. When a process encounters the execvp() system call, the next instruction it executes will be the one at the entry point of the new executable file. Thus the kernel performs a considerable amount of work in this system call. It must:

   - find the new executable file,
   - load the file into the address space currently being used by the calling process (overwriting and discarding the previous program), - set the argv array and environment variables for the new program execution, and start the process executing at the new program's entry point.

3. **wait()**

   ->

   The wait() system call is used by a process to block itself until the kernel signals the process to execute again, for example because one of its child processes has terminated. When the wait() call returns as a result of a child process's terminating, the status of the terminated child is returned as a parameter to the calling process

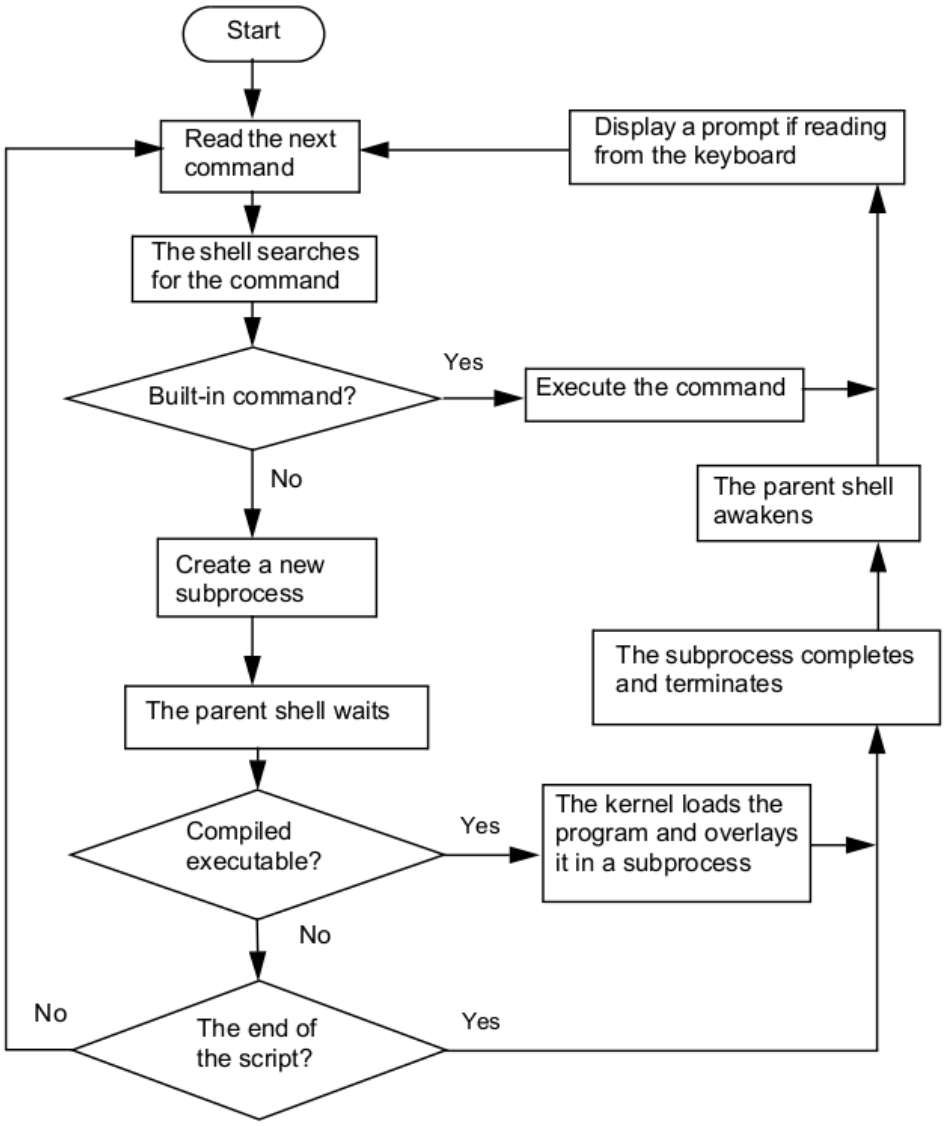


B. Implementation

1. Software Requirements:

- Language: C
- Platform : Linux

2. Flowchart



3. Brief description of the working

1. An infinite while loop is created, printing current directory and waiting for single line user input.
2. If the user calls for cd, the directory is changed to a given path if the path exists.
3. If the user calls setenv with proper format, then a new environment variable is created with specified value. If an environment variable already exists, then it's value is changed.
4. If user calls printenv, then:

- If no environment variable is given, all environment variables are printed with their values.
- Else the values of specified variables are printed.

5. If the user calls history, then the history file is printed.
6. If the user types exit/quit/x, the history file is deleted, all memory is freed, loop is broken and the program terminates.
7. If there is piping in input, after all validity checks, two child processes are created and then both internal and external commands are executed.
8. If there is redirection, after all validity checks, a child process is created and both internal and external commands are executed with proper redirections to the files.
9. In all other cases, user input is executed using a child process and if anything invalid, an error is displayed.
10. In all these cases, the input is saved in the history file in sequence.
11. Then all memory is freed and the loop continues from step 1.

4. Code:

I. Libraries Included:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include <stdarg.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat
```

II. Define Statements:

Following are various ***defined*** statements since these are used in our program regularly.

```
#define print_error_message fprintf(stderr,"\033[1;91mERROR: Sorry! No such command exists in my mini shell.\n\033[0m");
#define print_to_file fprintf(history_ptr, "\t%d. %s\n", serial, user_input);
#define RESET printf("\033[0m");
#define RED printf("\033[1;91m");
#define CLEAR printf("\e[1;1H\e[2J");
```

III. Created Functions:

```
int count_argument_numbers(char*);
```

-> This function counts the number of arguments in user input keeping in mind all whitespaces, double quotes, redirection and piping symbols.

```
char** find_all_paths();
```

-> This function will find all paths that are available in shell to execute the external commands.

```
char** separate_user_input(char*, int);
```

-> This function parse the user input into different arrays keeping in mind all whitespaces, double quotes, redirection and piping symbols.

```
int is_present(char**, int, char*);
```

-> This function checks whether a particular string is present in the parsed user input

```
int find_positions(char**,int,char*,int**);
```

-> This function counts and finds all accurance of a particular string in the parsed user input.

```
char*** split_commands(char**, int, int);
```

-> This function will split the parsed user input into two different commands at a given position.

```
char** find_command(char**,int,int);
```

-> This function finds the main command after removing symbols and redirected files.

```
char* get_program_path(char*, char**);
```

-> This function finds the path of the external command.

```
void print_message(char*,char);
```

-> This function prints a center-aligned message on screen.

```
void print_env_var_error(char**);
```

-> If the user types an environment variable without echo or printenv, then show error and correct form.

```
void execute_cd_command(char**,int);
```

-> This function executes the change directory operation.

```
void execute_history(char **);
```

-> This function executes the history command using cat, i.e., print the history file.

```
void execute_commands(char**,char**,int);
```

-> This function executes all internal and external commands in the shell.

```
char** execute_env_var(char**,int,char**);
```

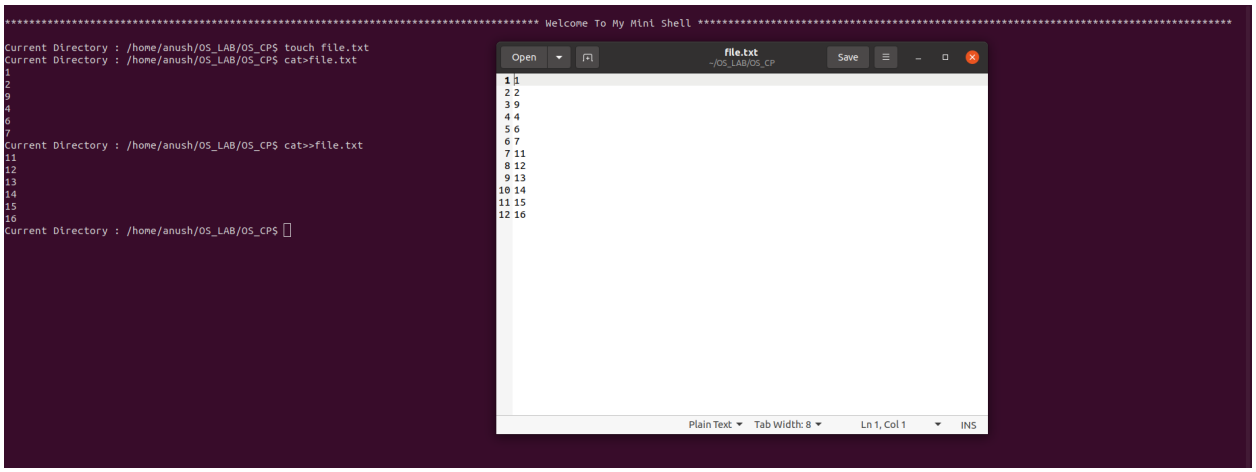
-> This function replaces the value of the environment variable into our input array.

Main Function

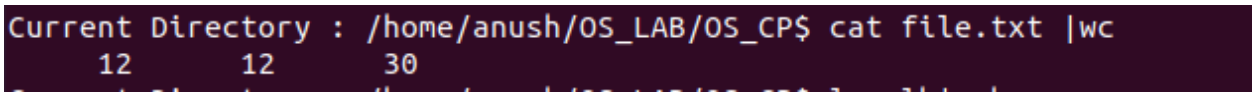
```
int main()
{
    system("clear");
    char **all_paths = find_all_paths();
    Fchar *temp_path = (char *)malloc(PATH_MAX * sizeof(char)); // Set value of PATH to all paths and store the value.
    for (int j = 0; j < total_paths; j++)
    {
        strcat(temp_path, all_paths[j]);
        if (j != (total_paths - 1))
            strcat(temp_path, ":");
    }
    char *t = "PATH";
    setenv(t, temp_path, 1);
   FILE *history_ptr = fopen("/tmp/history.txr", "w");  // Create a file to store the user commands in "w+" mode.
    print_message("Welcome To My Mini Shell", '*'); // Print Welcome Message
    printf("\n");
    int serial = 0; // Variable to keep track of number of inputs to maintain in history file.
    while (1)
    {
        char current_directory[PATH_MAX]; // Variable to store current direcory.
        if (getcwd(current_directory, sizeof(current_directory)) == NULL)
        {
            fprintf(stderr, "\033[1;91mERROR: Not able to access current working directory.\nProcess Terminated!\n\033[0m");
            break;
        }
```

Output Screenshot:

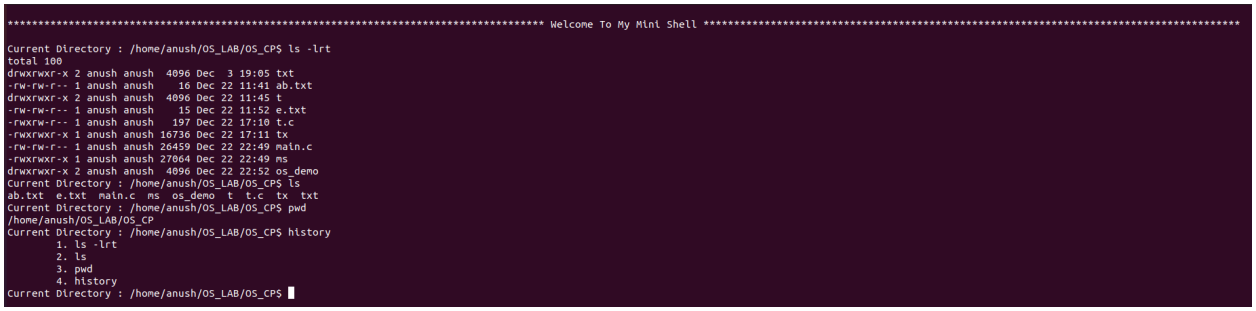
1. **Redirection**



2. Pipe



3. History Commands



4. Basic Command

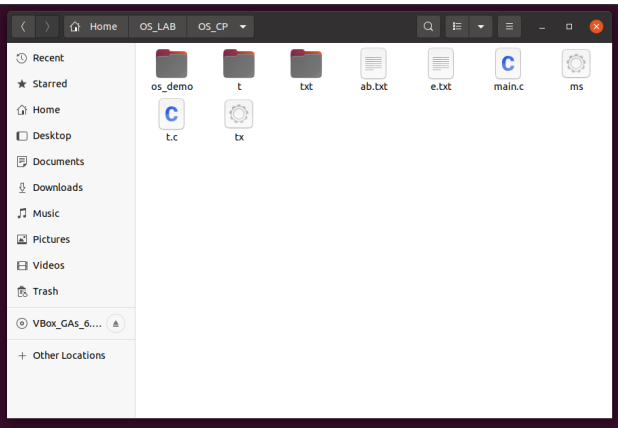
```
********************************************************************************** Welcome To My Mini Shell **********************************************************************************

Current Directory : /home/anush/OS_LAB/OS_CP$ ls
ab.txt  e.txt  main.c  ms  os_demo  t  t.c  tx  txt
Current Directory : /home/anush/OS_LAB/OS_CP$ ls -lrt
total 100
drwxrwxr-x 2 anush anush  4096 Dec  3 19:05 txt
drwxrwxr-x 2 anush anush  4096 Dec  8 17:56 os_demo
-rw-rw-r-- 1 anush anush    16 Dec 22 11:41 ab.txt
drwxrwxr-x 2 anush anush  4096 Dec 22 11:45 t
-rw-rw-r-- 1 anush anush    15 Dec 22 11:52 e.txt
-rwxrw-r-- 1 anush anush   197 Dec 22 17:10 t.c
-rwxrwxr-x 1 anush anush 16736 Dec 22 17:11 tx
-rw-rw-r-- 1 anush anush 26459 Dec 22 22:49 main.c
-rwxrwxr-x 1 anush anush 27064 Dec 22 22:49 ms
Current Directory : /home/anush/OS_LAB/OS_CP$ cal
   December 2021
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

Current Directory : /home/anush/OS_LAB/OS_CP$ pwd
/home/anush/OS_LAB/OS_CP
Current Directory : /home/anush/OS_LAB/OS_CP$
```

5. CD Command

```
Current Directory : /home/anush/OS_LAB/OS_CP$ mkdir os_demo
Current Directory : /home/anush/OS_LAB/OS_CP$ cd os_demo
Current Directory : /home/anush/OS_LAB/OS_CP/os_demo$
```



6. Quit Command

```
********************************************************************************** Welcome To My Mini Shell **********************************************************************************

Current Directory : /home/anush/OS_LAB/OS_CP$ pwd
/home/anush/OS_LAB/OS_CP
Current Directory : /home/anush/OS_LAB/OS_CP$ quit
```

```
********************************************************************************** Good Bye **********************************************************************************

                                                This Mini Shell is created by Aditya Shinde and Anush Shinde
```

7. Reasonable list of external linux commands

```
********************************************************************************* Welcome To My Mini Shell ********************************************************************************

Current Directory : /home/anush/OS_LAB/OS_CP$ cal
   December 2021
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

Current Directory : /home/anush/OS_LAB/OS_CP$ date
Wednesday 22 December 2021 11:20:20 PM IST
Current Directory : /home/anush/OS_LAB/OS_CP$
```

```
MAN(1)                                                                  Manual pager utils                                                                  MAN(1)

NAME
       man - an interface to the system reference manuals

SYNOPSIS
       man [man options] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [man options] [section] term ...
       man -f [whatis options] page ...
       man -l [man options] file ...
       man -w|-W [man options] page ...

DESCRIPTION
       man  is  the  system's manual pager.  Each page argument given to man is normally the name of a program, utility or function.  The manual page associated with each of these arguments is then
       found and displayed.  A section, if provided, will direct man to look only in that section of the manual.  The default action is to search in all of the available sections following  a  pre-
       defined order (see DEFAULTS), and to show only the first page found, even if page exists in several sections.

       The table below shows the section numbers of the manual followed by the types of pages they contain.

       1   Executable programs or shell commands
       2   System calls (functions provided by the kernel)
       3   Library calls (functions within program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions, e.g. /etc/passwd
       6   Games
       7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
       8   System administration commands (usually only for root)
       9   Kernel routines [Non standard]

       A manual page consists of several sections.

       Conventional  section names include NAME, SYNOPSIS, CONFIGURATION, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUE, ERRORS, ENVIRONMENT, FILES, VERSIONS, CONFORMING TO, NOTES, BUGS, EXAMPLE,
       AUTHORS, and SEE ALSO.

       The following conventions apply to the SYNOPSIS section and can be used as a guide in other sections.

       bold text          type exactly as shown.
       italic text        replace with appropriate argument.
       [-abc]             any or all arguments within [ ] are optional.
       -a|-b              options delimited by | cannot be used together.
       argument ...       argument is repeatable.
       [expression] ...   entire expression within [ ] is repeatable.

       Exact rendering may vary depending on the output device.  For instance, man will usually not be able to render italics when running in a  terminal,  and  will  typically  use  underlined  or
       coloured text instead.

 Manual page man(1) line 1 (press h for help or q to quit)
```