

MODEL INSTITUTE OF ENGINEERING AND TECHNOLOGY

JAMMU, JAMMU & KASHMIR



Project Title:

Implement your own shell MIET SHELL which performs the same function like other shells.

BACHELOR OF TECHNOLOGY (CSE)

2021-2025

Submitted by:	
Astha sharma	2021a1r121
Mayida	2021a1r129
Rishita Sharma	2021a1r130
Suhani Koul	2021a1r138

ACKNOWLEDGEMENT

Through this section of our report, we want to present our gratitude towards our institute MIET. Being able to work on projects , that require you to analyze and solve a problem that exists in real world, is an experience that one can achieve rarely, and we are happy and thankful to get that chance here.

We would also like to thank our mentors Prof. Ankur Gupta, Asst. Prof. Pragati Jamwal for being a guiding force throughout this period. It was a great experience to work under this project as we learned a lot of new things. We would also like to thank our HOD , Mr. Ashok Kumar and the CSE Department who encouraged and inspired us for learning new skills.

TABLE OF CONTENTS

1.ACKNOWLEDGEMENT

2.TABLE OF CONTENTS

3.INTRODUCTION

4.THEORY

5.IMPLEMENTATION

6.CODING

7.OUTPUT

8.REFERENCES

Introduction

This report has described the successful design and implementation of our own “MIET” shell. This shell program is written in C language. It performs the same functions as other shell programs like Ksh,bash.

Shell Programs: A shell program is an application that allows interacting with the computer. In a shell the user can run programs and also redirect the input to come from a file and output to come from a file. Shells also provide programming constructions such as if, for, while, functions, variables etc. Additionally, shell programs offer features such as line editing, history, file completion, wildcards, environment variable expansion, and programing constructions

In addition to command line shells, there are also Graphical Shells such as the Windows Desktop, MacOS Finder, or Linux Gnome and KDE that simplify the use of computers for most of the users. However, these graphical shells are not a substitute for command line shells for power users who want to execute complex sequences of commands repeatedly or with parameters not available in the friendly, but limited graphical dialogs and controls.

Problem Statement:

Implement your own shell “miet_shell” which performs the same function as the other shell programs like Ksh, bash.

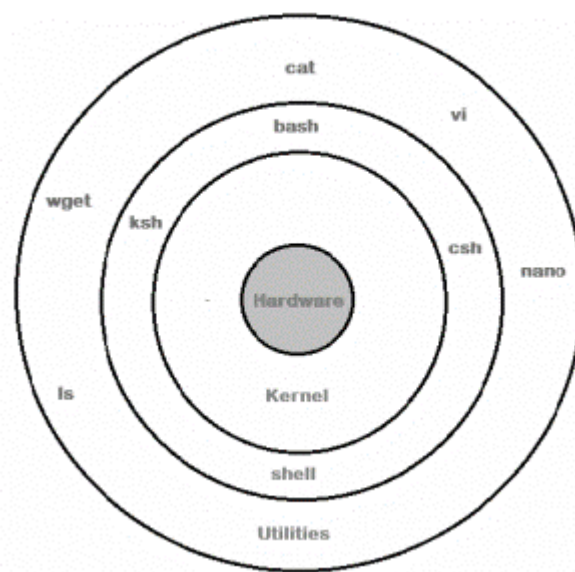
Objective:

The objective of this project is to gain experience with some advanced programming techniques like process creation and control, file descriptors, signals and possibly pipes. To do this, you will be writing your own command shell - much like csh, bsh or the DOS command shell.

Theory:

What is Shell?

A shell is special user program which provide an interface to user to use operating system services. Shell accepts 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

Shell Scripting:

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. A shell script usually has comments that describe the steps. The different operations performed by shell scripts are program execution, file manipulation and text printing. A wrapper is also a kind of shell script that creates the program environment, runs the program etc.

Types of Shells in Linux:

There are several shells are 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 (K Shell)** – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Basic lifetime of a shell

Let's look at a shell from the top down. A shell does three main things in its lifetime.

- **Initialize**: In this step, a typical shell would read and execute its configuration files. These change aspects of the shell's behavior.
- **Interpret**: Next, the shell reads commands from stdin(which could be interactive, or a file) and executes them.
- **Terminate**: After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

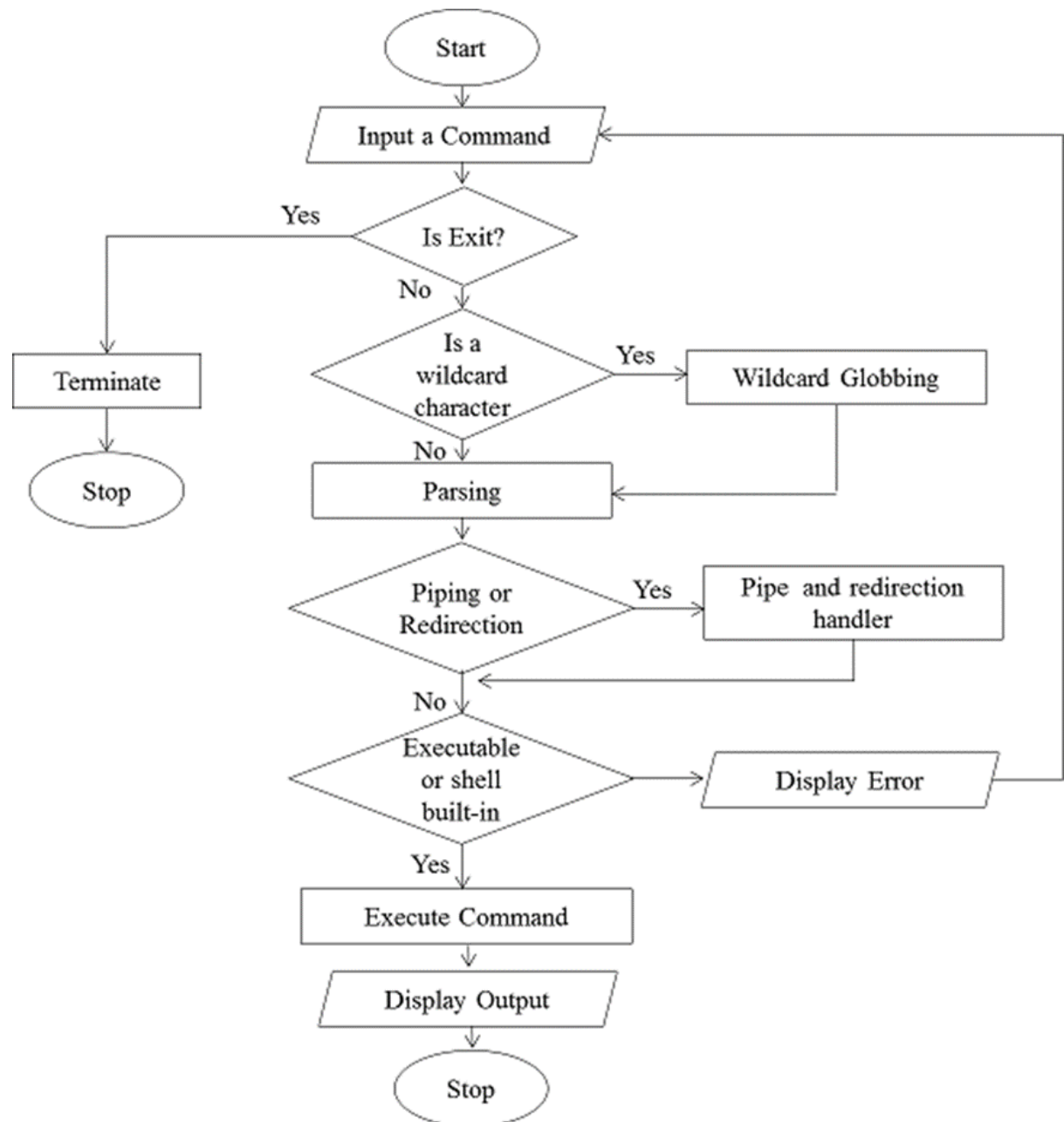
The Basics

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 **except** Printing current directory name and asking for next input.

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.

Flowchart



Implementation

To install the readline library, open the terminal window and write
sudo apt-get install libreadline-dev

It will ask for your password. Enter it. Press y in the next step.

- Printing the directory can be done using **getcwd**.
- Getting user name can be done by **getenv("USER")**
- Parsing can be done by using **strsep("")**. It will separate words based on spaces. Always skip words with zero length to avoid storing of extra spaces.
- After parsing, check the list of built-in commands, and if present, execute it. If not, execute it as a system command. To check for built-in commands, store the commands in an array of character pointers, and compare all with **strcmp()**.
- Note: "cd" does not work natively using **execvp**, so it is a built-in command, executed with **chdir()**.
- For executing a system command, a new child will be created and then by using the **execvp**, execute the command, and wait until it is finished.
- Detecting pipes can also be done by using **strsep("|")**. To handle pipes, first separate the first part of the command from the second part. Then after parsing each part, call both parts in two separate new children, using **execvp**. Piping means passing the output of first command as the input of second command.
- Declare an integer array of size 2 for storing file descriptors. File descriptor 0 is for reading and 1 is for writing.
- Open a pipe using the **pipe()** function.
- Create two children.
- Declare an integer array of size 2 for storing file descriptors. File descriptor 0 is for reading and 1 is for writing.
- Open a pipe using the **pipe()** function.
- Create two children.

In child 1->

Here the output has to be taken into the pipe.

Copy file descriptor 1 to stdout.

Close file descriptor 0.

Execute the first command using `execvp()`

In child 2->

Here the input has to be taken from the pipe.

Copy file descriptor 0 to stdin.

Close file descriptor 1.

Execute the second command using `execvp()`

- Wait for the two children to finish in the parent.

Coding

// Implement your own shell MIET shell which performs the same function like other shell

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<readline/readline.h>
#include<readline/history.h>
```

```
#define MAXCOM 1000 // max number of letters to be supported
#define MAXLIST 100 // max number of commands to be supported
```

// Clearing the shell using escape sequences

```
#define clear() printf("\033[H\033[J")
```

// Greeting shell during startup

```
void init_shell()
{
    clear();
    printf("\n\n\n*****"
           "*****");
    printf("\n\n\n\t***MIET SHELL***");
    printf("\n\n\t-WELCOME-");
    printf("\n\n\n*****"
           "*****");
    char* username = getenv("USER");
    printf("\n\n\nUSER is: @%s", username);
    printf("\n");
}
```

```

        sleep(1);
        clear();
    }

```

// Function to take input

```

int takeInput(char* str)
{
    char* buf;

    buf = readline("\n>>> ");
    if (strlen(buf) != 0) {
        add_history(buf);
        strcpy(str, buf);
        return 0;
    } else {
        return 1;
    }
}

```

// Function to print Current Directory.

```

void printDir()
{
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    printf("\nDir: %s", cwd);
}

```

// Function where the system command is executed

```

void execArgs(char** parsed)
{
    // Forking a child
    pid_t pid = fork();

    if (pid == -1) {
        printf("\nFailed forking child..");
    }
}

```

```

        return;
    } else if (pid == 0) {
        if (execvp(parsed[0], parsed) < 0) {
            printf("\nCould not execute command..");
        }
        exit(0);
    } else {
        // waiting for child to terminate
        wait(NULL);
        return;
    }
}

```

// Function where the piped system commands is executed

```
void execArgsPiped(char** parsed, char** parsedpipe)
```

```

{
    // 0 is read end, 1 is write end
    int pipefd[2];
    pid_t p1, p2;

    if (pipe(pipefd) < 0) {
        printf("\nPipe could not be initialized");
        return;
    }
    p1 = fork();
    if (p1 < 0) {
        printf("\nCould not fork");
        return;
    }

    if (p1 == 0) {
        // Child 1 executing..
        // It only needs to write at the write end
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
    }
}

```

```

        close(pipefd[1]);

        if (execvp(parsed[0], parsed) < 0) {
            printf("\nCould not execute command 1..");
            exit(0);
        }
    } else {
        // Parent executing
        p2 = fork();

        if (p2 < 0) {
            printf("\nCould not fork");
            return;
        }

        // Child 2 executing..
        // It only needs to read at the rear end
        if (p2 == 0) {
            close(pipefd[1]);
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);
            if (execvp(parsedpipe[0], parsedpipe) < 0) {
                printf("\nCould not execute command 2..");
                exit(0);
            }
        } else {
            // parent executing, waiting for two children
            wait(NULL);
            wait(NULL);
        }
    }
}

```

// Help commands built-in

```
void openHelp()
```

```

{
    puts("\n**WELCOME TO MIET SHELL HELP**")
        "\n-Use the shell at your own risk..."
        "\nList of Commands supported:"
        "\n>cd"
        "\n>ls"
        "\n>exit"
        "\n>all other general commands available in UNIX shell"
        "\n>pipe handling"
        "\n>improper space handling");

    return;
}

```

// Function to execute builtin commands

```

int ownCmdHandler(char** parsed)
{
    int NoOfOwnCmds = 4, i, switchOwnArg = 0;
    char* ListOfOwnCmds[NoOfOwnCmds];
    char* username;

    ListOfOwnCmds[0] = "exit";
    ListOfOwnCmds[1] = "cd";
    ListOfOwnCmds[2] = "help";
    ListOfOwnCmds[3] = "hello";

    for (i = 0; i < NoOfOwnCmds; i++) {
        if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {
            switchOwnArg = i + 1;
            break;
        }
    }

    switch (switchOwnArg) {
    case 1:

```

```

        printf("\nGoodbye\n");
        exit(0);
case 2:
    chdir(parsed[1]);
    return 1;
case 3:
    openHelp();
    return 1;
case 4:
    username = getenv("USER");
    printf("\nHello %s.\nMind that this is "
           "not a place to play around."
           "\nUse help to know more..\n",
           username);
    return 1;
default:
    break;
}

return 0;
}

// function for finding pipe
int parsePipe(char* str, char** strpiped)
{
    int i;
    for (i = 0; i < 2; i++) {
        strpiped[i] = strsep(&str, "|");
        if (strpiped[i] == NULL)
            break;
    }

    if (strpiped[1] == NULL)
        return 0; // returns zero if no pipe is found.
    else {

```

```

        return 1;
    }
}

// function for parsing command words
void parseSpace(char* str, char** parsed)
{
    int i;

    for (i = 0; i < MAXLIST; i++) {
        parsed[i] = strsep(&str, " ");

        if (parsed[i] == NULL)
            break;
        if (strlen(parsed[i]) == 0)
            i--;
    }
}

int processString(char* str, char** parsed, char** parsedpipe)
{
    char* strpiped[2];
    int piped = 0;

    piped = parsePipe(str, strpiped);

    if (piped) {
        parseSpace(strpiped[0], parsed);
        parseSpace(strpiped[1], parsedpipe);
    } else {
        parseSpace(str, parsed);
    }
}

```



```

        if (ownCmdHandler(parsed))
            return 0;
        else
            return 1 + piped;
    }

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;
}

```

Output

```
*****  
  
****MIET SHELL****  
  
-WELCOME-  
  
*****
```

```
supde@ubuntu: /media/supde/FILES/Documents/  
Dir: /media/supde/FILES/Documents/interhips/2nd  
>>> ls  
a.out  shell.c  
  
Dir: /media/supde/FILES/Documents/interhips/2nd  
>>> mkdir gfg  
  
Dir: /media/supde/FILES/Documents/interhips/2nd  
>>> ls  
a.out  gfg  shell.c  
  
Dir: /media/supde/FILES/Documents/interhips/2nd  
>>> hello  
  
Hello supde.  
Mind that this is not a place to play around.  
Use help to know more..  
  
Dir: /media/supde/FILES/Documents/interhips/2nd  
>>> exit  
  
Goodbye
```

REFERENCES:

- ALL LECTURES TAUGHT BY, Asst. PROFESSOR PRAGTI JAMWAL
- ALL LABS HELD BY Asst. PROFESSOR PRAGATI JAMWAL
- GOOGLE SEARCH
- www.geekforgeeks.org