B.M.S College of Engineering

P.O. Box No.: 1908 Bull Temple Road,

Bangalore-560 019

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Course – Unix System Programming

Course Code – 19IS4PWUSP

AY 2020-21

<u>Report on Unix System Programming Project</u>

TERMINAL BASED FILE EXPLORER

Submitted by

Moksh Jayanth 1BM19IS094
Mohan D 1BM19IS092
N Prabhu 1BM19IS096

Submitted to

Anitha H M

B.M.S College of Engineering

P.O. Box No.: 1908 Bull Temple Road,

Bangalore-560 019

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

CERTIFICATE

Certified that the Project has been successfully presented at **B.M.S College Of Engineering** by **Moksh Jayanth, Mohan D and N Prabhu** bearing USN: **1BM19IS094, 1BM19IS092 and 1BM19IS096** in partial fulfillment of the requirements for the IV Semester degree in **Bachelor of Engineering in Information Science & Engineering** of **Visvesvaraya Technological University, Belgaum** as a part of project for the course **Unix System Programming (19IS4PWUSP)** during academic year 2020-2021.

Faculty Name – Anitha H M

Department of ISE, BMSCE

TABLE OF CONTENTS

ABSTRACT

- This project ‘Terminal based File Explorer’ can work on Linux and Unix-based operating systems.
- The File Explorer can be used in two different modes i.e
    - cursor-based user interface
    - command-line interface.
- This project involves some of the Unix system programming concepts which are implemented using C++.

INTRODUCTION

File explorer works in two modes. The application starts in normal mode, which is the default mode and used to explore the current directory and navigate around in the filesystem.
The root of the application is the directory where the application was started.
The last line of the display screen is to be used as status bar - to be used in normal and command-line modes.

1. NORMAL MODE :-

1.1 Read and display list of files and directories in the current folder

File explorer shows each file in the directory (one entry per line). The following attributes are visible for each file
File Name
File size
Ownership (User & Group) & Permissions
Last modified
The File explorer also handles scrolling (vertical overflow) in case the directory has a lot of files.
The file explorer also shows the entries "**.**" & "**..**" for current and parent directory respectively.
User is able to navigate up & down the file list using corresponding arrow keys.

1.2 Open files & directories

When enter is pressed
Directory - It will Clear the screen and Navigate into the directory and shows the files & directories inside it as specified in point 1
Files - It will open files using the corresponding default application.

2. COMMAND MODE :-

The application enters the command mode whenever the : (colon) key is pressed.
Upon entering the command mode the user should be able to enter different commands. All commands should appear in a bottom status bar

2.1 copy, move and rename

```
copy <source_file(s)> <destination_directory>
move <source_file(s)> <destination_directory>
```

Eg:

```
copy foo.txt bar.txt baz.mp4 ~/foobar
move foo.txt bar.txt baz.mp4 ~/foobar
rename foo.txt bar.txt
```

Copying / Moving of directories is also be implemented

2.2 create files and directories

```
create_file <file_name> <destination_path>
create_dir <dir_name> <destination_path>
```

Eg:

```
create_file foo.txt ~/foobar
create_file foo.txt .
create_dir folder_name ~/foobar
```

2.3 delete files and directories

```
delete_file <file_path>
delete_dir <directory_path>
```

Eg:

```
delete_file ~/foobar/foo.txt.
delete_dir ~/foobar/folder_name
```

2.4 goto

```
goto <directory_path>
```

Eg:

goto /home/darshan/

goto ~

2.5 Search a file or folder given full name.

search <filename>

Eg:

search foo.txt

Search for the given filename under the current directory recursively

2.6 On pressing 'ESC' key the application should go to Normal Mode

PROBLEM STATEMENT

Create a Terminal Based File Explorer using the concepts of UNIX System programming through C++.

The project performs various tasks on the files

In normal mode - navigate through all the directories and files.

In command mode - can be used to perform the following operations:

- copy
- move
- rename
- create file
- create directory
- delete file
- delete directory
- goto
- search

APIs USED IN THE PROJECT

- **General File APIs :-**

  a. open()
  b. read()
  c. write()
  d. close()
  e. stat()

- **Directory File APIs :-**

  a. opendir()
  b. readdir()
  c. chdir()
  d. closedir()
  e. mkdir()
  f. rmdir()

- **Symbolic link APIs :-**

  a. lstat()
  b. link()
  c. unlink()

EXPLANATION ABOUT THE APIs

What is an API?

API is an abbreviation for Application Programming Interface which is a collection of communication protocols and subroutines used by various programs to communicate between them. A programmer can make use of various API tools to make its program easier and simpler. Also, an API facilitates the programmers with an efficient way to develop their software programs

API used in the project

1) **General file APIs**

open() : The open() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if O_CREAT is specified in *flags*) be created by open().
On success, open() return the new file descriptor (a nonnegative integer).
On error, -1 is returned and *errno* is set to indicate the error.

read() : read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. See also NOTES.
On error, -1 is returned, and errno is set to indicate the error. In this case, it is left unspecified whether the file position (if any) changes.

write() : write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and *errno* is set appropriately.

If *count* is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

close() : close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see fcntl(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

close() returns zero on success.
On error, -1 is returned, and *errno* is set to indicate the error.

stat() : stat() and fstatat() retrieve information about the file pointed to by *pathname*; the differences for fstatat() are described below.
On success, zero is returned.
On error, -1 is returned, and *errno* is set to indicate the error.

2) **Directory file APIs**

opendir() : The opendir() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.
The opendir() functions return a pointer to the directory stream.
On error, NULL is returned, and *errno* is set to indicate the error.

readdir() : The readdir() function returns a pointer to a *dirent* structure representing the next

directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.
On success, readdir() returns a pointer to a *dirent* structure.(This structure may be statically allocated; do not attempt to free(3) it.)
If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set to indicate the error. To distinguish end of stream from an error, set *errno* to zero before calling readdir() and then check the value of *errno* if NULL is returned.

chdir() : chdir() changes the current working directory of the calling process to the directory specified in the path.
On success, zero is returned.
On error, -1 is returned, and *errno* is set to indicate the error.

closedir() : The closedir() function closes the directory stream associated with *dirp*. A successful call to closedir() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.
The closedir() function returns 0 on success.
On error, -1 is returned, and *errno* is set to indicate the error.

mkdir() : The *mkdir*() function shall create a new directory with name *path*. The file permission bits of the new directory shall be initialized from *mode*. These file permission bits of the *mode* argument shall be modified by the process' file creation mask.
Upon successful completion, *mkdir*() shall return 0. Otherwise, -1 shall be returned, no directory shall be created, and *errno* shall be set to indicate the error.

rmdir()
The *rmdir*() function shall remove a directory whose name is given by *path*. The directory

shall be removed only if it is an empty directory.

Upon successful completion, the function *rmdir*() shall return 0. Otherwise, -1 shall be returned, and *errno* set to indicate the error. If -1 is returned, the named directory shall not be changed.

3) **Symbolic link APIs**

lstat()

lstat() is identical to stat(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

link() : link() creates a new link (also known as a hard link) to an existing file.

If a newpath exists, it will *not* be overwritten. This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

unlink() : unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name refers to a symbolic link, the link is removed.

If the name refers to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

On success, zero is returned.

On error, -1 is returned, and *errno* is set to indicate the error.

ADVANTAGE OF APIs -

Efficiency: API produces efficient, quicker and more reliable results than the outputs produced by human beings in an organization.

Flexible delivery of services: API provides fast and flexible delivery of services according to developers requirements.

Integration: The best feature of API is that it allows movement of data between various sites and thus enhances integrated user experience.

Automation: As API makes use of robotic computers rather than humans, it produces better and automated results.

New functionality: While using API the developers find new tools and functionality for API exchanges.

IMPLEMENTATION / CODE

Some of the important code Snippets of the project are given below:

Display files :-

```
/*********************************************** DISPLAY ALL FILES AND
FOLDERS ********************************************/

void displayFiles(){

    clr();
    struct stat fileInfo;

    for(auto itr = top; itr < min(bottom, noOfFiles()); itr++){
        lstat(fileNames[itr]->d_name,&fileInfo);
        cout<<"$"<<itr+1<<" : \t\t";
        (S_ISDIR(fileInfo.st_mode)) ? cout<<"d" :
S_ISSOCK(fileInfo.st_mode) ? cout<<"s": cout<<"-";

        (S_IRUSR & fileInfo.st_mode) ? cout<<"r" : cout<<"-"; (S_IWUSR
& fileInfo.st_mode) ? cout<<"w" : cout<<"-"; (S_IXUSR &
fileInfo.st_mode) ? cout<<"x" : cout<<"-";

        (S_IRGRP & fileInfo.st_mode) ? cout<<"r" : cout<<"-"; (S_IWGRP
& fileInfo.st_mode) ? cout<<"w" : cout<<"-"; (S_IXGRP &
fileInfo.st_mode) ? cout<<"x" : cout<<"-";

        (S_IROTH & fileInfo.st_mode) ? cout<<"r" : cout<<"-"; (S_IWOTH
& fileInfo.st_mode) ? cout<<"w" : cout<<"-"; (S_IXOTH &
fileInfo.st_mode) ? cout<<"x" : cout<<"-";

        int SIZE_OF_FILE = fileInfo.st_size;
        if(SIZE_OF_FILE/1048576 > 1)
cout<<"\t\t"<<SIZE_OF_FILE/1048576<<" MB";
        else if(SIZE_OF_FILE/1024 > 1)
cout<<"\t\t"<<SIZE_OF_FILE/1024<<" KB";
        else cout<<"\t\t"<<SIZE_OF_FILE<<" B";
        if((S_ISDIR(fileInfo.st_mode))){
            cout<<"\t\t"<<"\033[1;32m"<<fileNames[itr]-
>d_name<<"\033[0m";
        }
        else{              cout<<"\t\t"<<"\033[1;36m"<<fileNames[itr]-
>d_name<<"\033[0m";
```

```
        }
        cout<<"\n";
    }
    printCWD();
    printNormalMode();
    return;
}
```

Create File/Directory :-

```
/***************************************** CREATE FILE/DIRECTORY
*******************************************/

void create_file(){
    int len = commandTokens.size();
    string destination = commandTokens[len-1];

    for(int i=1; i<len-1; i++){
        string fname = commandTokens[i];
        open((destination + '/' +
fname).c_str(),O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    }
}

void create_dir(){
    int len = commandTokens.size();
    string destination = commandTokens[len-1];

    for(int i=1; i<len-1; i++){
        string fname = commandTokens[i];
        mkdir((destination + '/' +
fname).c_str(),S_IRUSR|S_IWUSR|S_IXUSR);
    }
}
```

Search File/Directory :-

```
/************************************************SEARCH
****************************************************/

bool search_helper(string dirName, string tobeSearch){
    DIR *di;
    struct dirent *diren;
    struct stat fileInfo;

    if(!(di = opendir(dirName.c_str()))){
        printStatusLine("Can't         open          the          directory
");
        return false;
    }

    chdir(dirName.c_str());
    while((diren = readdir(di))){
        lstat(diren->d_name,&fileInfo);
        string dname =  string(diren->d_name);
        if(tobeSearch == dname){
            processCurrentDIR(dirName.c_str());
            return true;
        }
        if(S_ISDIR(fileInfo.st_mode)){
            if( (dname == ".") || (dname == "..") ){
                continue;
            }
            bool t =  search_helper(dirName + '/' + dname, tobeSearch);
            if(t) return true;
        }
    }
    chdir("..");
    closedir(di);
    return false;
}

bool search(){
    string tbs = commandTokens[1];
    return search_helper(cwd, tbs);
}
```

Delete File/Directory :-

```
/************************************ DELETE FILE/DIRECTORY
*******************************************/

int delete_file(){
    string destination = commandTokens[1];
    int status = unlink(destination.c_str());
    return status;
}

void delete_dir_helper(string destination){
    DIR *di;
    struct dirent *diren;
    struct stat fileInfo;

    if(!(di = opendir(destination.c_str()))){
        printStatusLine("Can't open the directory
");
        return;
    }
    chdir(destination.c_str());
    while((diren = readdir(di))){
        lstat(diren->d_name,&fileInfo);
        if(S_ISDIR(fileInfo.st_mode)){
            if(strcmp(".",diren->d_name)==0 || strcmp("..",diren-
>d_name)==0){
                continue;
            }
            delete_dir_helper(diren->d_name);
            rmdir(diren->d_name);
        }
        else{
            unlink(diren->d_name);
        }
    }
    chdir("..");
    closedir(di);
}

int delete_dir(){
    string destination = commandTokens[1];
    if(destination==cwd){
        printStatusLine("You are present inside the directory which you
want to delete!");
        return 0;
    }
    delete_dir_helper(destination);
    rmdir(destination.c_str());
    return 1;
}
```

Process Current Directory :-

```
/*****************************************   PROCESS   CURRENT   DIRECTORY
*****************************************/

void processCurrentDIR(char const* dir){
    DIR* di;
    struct dirent* direntStructure;

    if(!(di=opendir(dir))){
        printAlertLine("Directory                is                empty
");
        return;
    }
    chdir(dir);
    getcwd(cwd,cwdSize);
    fileNames.clear();

    while((direntStructure=readdir(di))){
        fileNames.push_back(direntStructure);
    }

    closedir(di);
    resetPointers();
    displayFiles();
    return;
}
```

Rename :-

```
/************************************************RENAME****************
************************************/

void rename(){
    string oldName = commandTokens[1];
    string newName = commandTokens[2];
    rename(oldName.c_str(),newName.c_str());
}
```

Goto :-

```
/************************************************GOTO******************
************************************/

void goTo(){
    string destination = commandTokens[1];
    if(destination == "/") home();
    else processCurrentDIR(destination.c_str());
```

Copy file/directory :-

```
/*****************************************COPY-
FILE/DIRECTORY***************************************************/

void copy_helper(string fname, string path){
    char b[1024];
    int fin,fout, nread;
    fin = open(fname.c_str(),O_RDONLY);
    fout = open((path).c_str(),O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    while((nread = read(fin,b,sizeof(b)))>0){
        write(fout,b,nread);
    }
}

void copy(int i){
    int len = commandTokens.size();
    string destination = commandTokens[len-1];

    string fname = commandTokens[i];
    string path = destination+'/'+fname;
    copy_helper(fname, path);

}

void copy_dir_helper(string dirName, string destination){
    DIR *di;
    struct dirent *diren;
    struct stat fileInfo;

    if(!(di = opendir(dirName.c_str()))){
        printStatusLine("Can't          open          the          directory
");
        return;
    }

    chdir(dirName.c_str());
    while((diren = readdir(di))){
        lstat(diren->d_name,&fileInfo);
        string dname =  string(diren->d_name);
        if(S_ISDIR(fileInfo.st_mode)){

            if( (dname == ".") || (dname == "..") ){
                continue;
            }
            mkdir((destination            +            '/'            +
dname).c_str(),S_IRUSR|S_IWUSR|S_IXUSR);
            copy_dir_helper(dname , destination + '/' + dname);
        }
        else{
            copy_helper(dname,destination + '/' + dname);
        }
    }
    chdir("..");
    closedir(di);
    return;
}

void copy_dir(int i){
    int len = commandTokens.size();
    string destination = commandTokens[len-1];

    string dname = commandTokens[i];
    mkdir((destination+'/'+dname).c_str(),S_IRUSR|S_IWUSR|S_IXUSR);
    copy_dir_helper(cwd + '/' + dname, destination + "/" + dname);
}

void copyWrapper(){
    int len = commandTokens.size();
    string destination = commandTokens[len-1];
    struct stat fileInfo;
    for(int i=1; i<len-1; i++){
        string loc = cwd + '/' + commandTokens[i];
        lstat(loc.c_str(), &fileInfo);
        if(S_ISDIR(fileInfo.st_mode)){
            copy_dir(i);
        }
        else{
            copy(i);
        }
    }
}
```

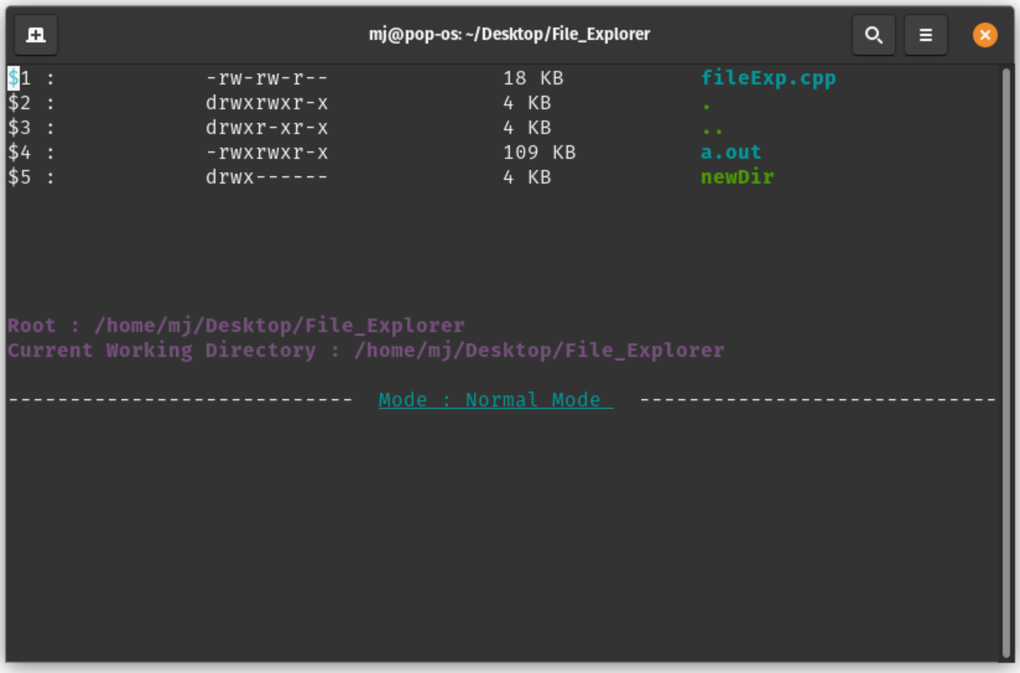
Normal Mode :-

```
/********************************** SCROLLING AND VERTICAL OVERFLOW FIX
*************************************/

void scrollUp(){
    if(cursor>1){
        cursor--;
        moveCursor(cursor,0);
        return;
    }
    if(top==0){      printAlertLine("You      hit      the      top
");
        return;
    }
    top--;
    bottom--;
    displayFiles();
    moveCursor(cursor,0);
    return;
}

void scrollUpK(){
    top = max(top-MAX, 0);
    bottom = top+MAX;
    displayFiles();
    moveCursor(cursor,0);
    return;
}

void scrollDownL(){
    bottom = min(bottom+MAX, noOfFiles());
    top = bottom - MAX;
    displayFiles();
    moveCursor(cursor,0);
    return;
}

void scrollDown(){
    if(cursor<noOfFiles() && cursor<MAX){
        cursor++;
        moveCursor(cursor,0);
        return;
    }
    if(bottom==noOfFiles()){
        printAlertLine("You          hit          the          bottom
");
        return;
    }
    top++;
    bottom++;
    displayFiles();
    moveCursor(cursor,0);
    return;
```

```
}
```

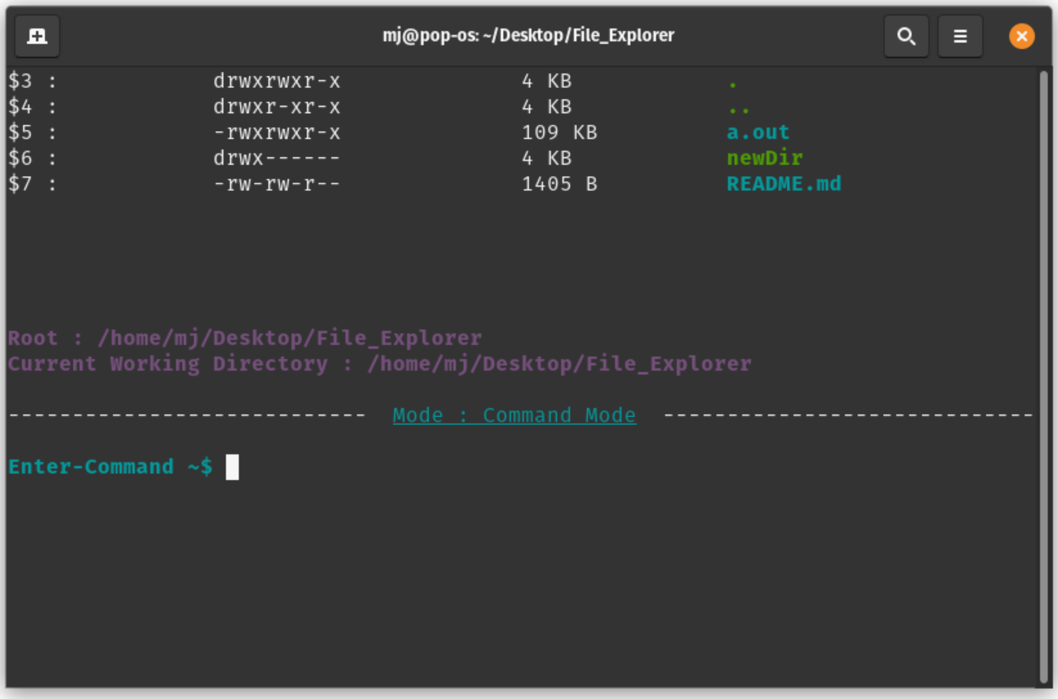
```
/*************************************    GO    TO    PARENT    DIRECTORY
***************************************/

void levelUp(){
    if(cwd==rootPath){
        printAlertLine("You're already present in the home directory");
        return;
    }
    backS.push(string(cwd));
    processCurrentDIR("../");
    return;
}

/****************************************    GO    TO    HOME    DIRECTORY
*************************************/

void home(){
    if(cwd==rootPath){
        printAlertLine("You're already present in the home directory");
        return;
    }
    backS.push(string(cwd));
    processCurrentDIR(rootPath.c_str());
    return;
}

/**************************************    MOVE    BACK    AND    FORWARD
***************************************/

void moveBack(){
    if(!backS.size())return;

    string prevDirectory = backS.top();
    backS.pop();
    forwardS.push(string(cwd));
    processCurrentDIR(prevDirectory.c_str());
    return;
}

void moveForward(){
    if(!forwardS.size())return;

    string nextDirectory = forwardS.top();
    forwardS.pop();
    backS.push(string(cwd));
    processCurrentDIR(nextDirectory.c_str());
    return;
}

/**************************************** ENTER INTO THE FOLDER
*******************************************/

void enter(){
    struct stat fileInfo;
    char *fileName = fileNames[cursor+top-1]->d_name;
    lstat(fileName,&fileInfo);

    if(S_ISDIR(fileInfo.st_mode)){
        if(strcmp(fileName,"..")==0 ){
            levelUp();
            return;
        }
        if(strcmp(fileName,".")==0) return;
        backS.push(string(cwd));
        processCurrentDIR((string(cwd)+'/'+string(fileName)).c_str());
    }
    else{
        pid_t pid=fork();
        if(pid==0){
            printAlertLine("File opened in default editor
");
            execl("/usr/bin/xdg-open","xdg-open",fileName,NULL);
            exit(1);
        }
    }
    return;
}
```

RESULTS/ SNAPSHOTS OF THE PROJECT DESIGN



```
mj@pop-os: ~/Desktop/File_Explorer
$1 :          -rw-rw-r--          18 KB          fileExp.cpp
$2 :          drwxrwxr-x          4 KB           .
$3 :          drwxr-xr-x          4 KB           ..
$4 :          -rwxrwxr-x          109 KB         a.out
$5 :          drwx------          4 KB           newDir




Root : /home/mj/Desktop/File_Explorer
Current Working Directory : /home/mj/Desktop/File_Explorer

----------------------------  Mode : Normal Mode   ----------------------------
```

NORMAL MODE



```
mj@pop-os: ~/Desktop/File_Explorer
$3 :          drwxrwxr-x          4 KB           .
$4 :          drwxr-xr-x          4 KB           ..
$5 :          -rwxrwxr-x          109 KB         a.out
$6 :          drwx------          4 KB           newDir
$7 :          -rw-rw-r--          1405 B         README.md




Root : /home/mj/Desktop/File_Explorer
Current Working Directory : /home/mj/Desktop/File_Explorer

----------------------------  Mode : Command Mode  ----------------------------

Enter-Command ~$ 
```

COMMAND MODE

REFERENCES

- Stack Overflow - https://stackoverflow.com
- SlideShare(APIsReference)- https://www.slideshare.net/1987suni/unix-file-apis-16271500
- GeeksforGeeks - https://www.geeksforgeeks.org/
- Working of a shell reference - https://brennan.io/2015/01/16/write-a-shell-in-c/