Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Operating Systems (CSE2005)

REVIEW III

MINI SHELL IN C PROGRAMMING LANGUAGE

Faculty: PROF. MEENAKSHI S.P

Slot-F1

By-

DHRUV-19BCE2035
K. V. AYUSHNAV -19BCE2544
SAYUJYA SAMIR MALKAN- 19BCE2559

School of Computer Science and Engineering (SCOPE

INDEX

<u>ABSTRACT</u>

Shell is the command interpreter in the linux platform. Linux shell commands provides us the platform to perform operation over the terminal. Shell is equivalent to the WINDOWS cmd command prompt for the Linux platform. Our project is based on the shell . We learned the functionalities of the shell commands and implemented on our project. File management system has been the main focus over the period of time. We implemented the shell functionalites in the file management system to make a fully functioning minishell. The functionalities of our minishell includes making a directory , changing the location , renaming the file, deleting the file, copying the file. Our shell is fully based on the C programming with used some library function in order to implement the system calls and many other requirements.

1.INTRODUCTION

1.1 Definition of Shell

Shell is a UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters. In some systems, the shell is called a command interpreter. A shell usually implies an interface with a command syntax (think of the DOS operating system and its "C:>" prompts and user commands such as "dir" and "edit").

As the outer layer of an operating system, a shell can be contrasted with the kernel, the operating system's inmost layer or core of services. It gathers input from you and executes programs based on that input. When a program
finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.
On most Linux systems a program called bash (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, sh, written by Steve Bourne) acts as the shell program. Besides bash, there are other shell programs that can be installed in a Linux system. These include: ksh, tcsh and zsh.

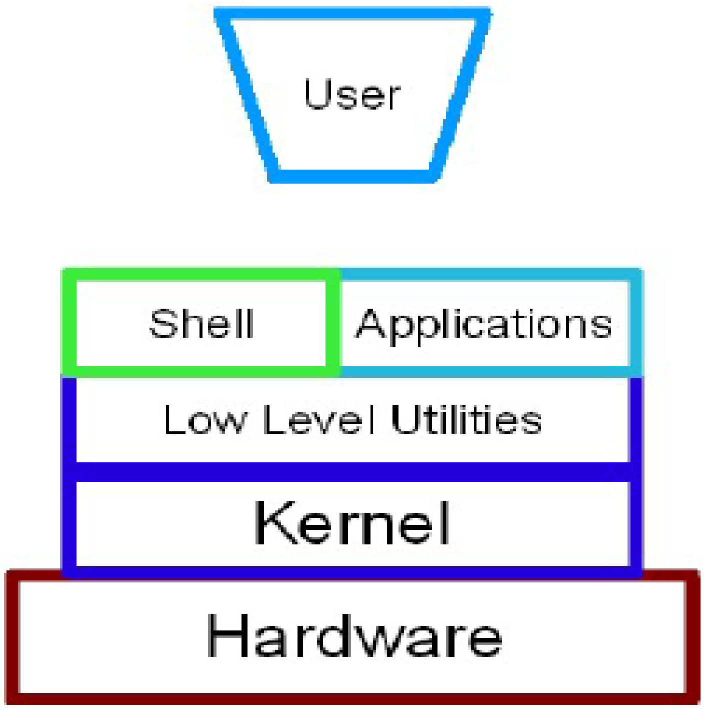


Fig:1.1 Structure of the Operating System

1.2 Basic Lifetime of Shell

1. Initialize:

In this step, a typical shell would read and execute its configuration files. These change aspects of the shell's behaviour

2. Interpret:

Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.

3. Terminate:

After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

1.3 Basic Loop in Shell

Read:

Read the command from standard input. When we provide any command to the shell. The shell reads the command from the input and provides the obtained input to the parser which is followed by the execution of the standard input.

Parse:

When the input is passed for the parsing. The command string is separated into a program and arguments. The command in our projects includes 'exit' , 'cd' , 'dir' , 'del' , 'ren' , 'copy'.

For example, if the user types the command as make Folder. Then the command is separated into two different types i.e "make" and "Folder".

Execute:

After the command is parsed then it is passed for the execution . The parsed command is executed to perform some task.

2. LITERATURE REVIEW

A literature survey in a project report is that section which shows the various analyses and research made in the field of your interest and the results already published, taking into a ccount the various parameters of the project and the extent of the project.

Since our project aims to develop shell for the file management system, we have opted to analyze the top notch projects which were developed in past. Identifying their key features, we hope to implement it in our own project: minishell for file management. To analyze any shell program , design is crucial to examine and in order to create a good shell the features are important for it to function properly.

Features:

- Easy Commands

Our project includes the use of easy commands to perform some operatio. "make folder_name " to make a new folder in the current directory, "cd folder name" to change the directory from current folder to another folder, "ren file file1" to rename the file, "copy file.c file1.c " to copy the file, "del file.c" to delete the file and "exit" command to exit from the shell.

- User friendly

It is one of the important factors for the project to be user friendly. If the project lacks the user friendly interface then the value of project won't be the same. Our project gives user the same interface as linux provides in the terminal. User which are familiar to the linux shell will be target for our project.

- Faster Execution:

User have a lot of operation to perform on the system. So making the user wait for a long time to perform a small task won't make any good. So the execution of the commands should be fast and efficient.
Our project aims for the better performance and faster execution of commands

3. OVERVIEW OF WORK

3.1 Problem description

Our projects includes the development of the mini shell in linux platform. The mini shell works similar to the linux terminal but with small features. The linux terminal has the large number of commands for the various operation. Our projects includes the commands that allows to make a new directory, change the directory, rename the file, delete the file, list all the files and folder in current directory, copy files from one folder to another, and finally exit to terminate the shell.Our project includes the C programming language to address above features. Mainly our minishell can be used as the command shell for the file management.

We used the Linux System Calls and commands for file and for process management to implement the functionalities of your shell. Our program will behave similarly to the Linux shell and wil display a prompt (i.e minishell>>) at start-up. The user should then enter one of the shell commands (with appropriate options and arguments) at the prompt followed by <Enter>, and the shell will again display the prompt after processing the command. If a command is invalid the shell should display an appropriate error message. In order to compile and link your program, you will need to implement a make file.

3.2 Software Requirements

a. Codeblocks
b. Virtual Box
c . Ubuntu

Our project was written in C programming language. The C program requires the GNU C compiler in Linux Platform. The codeblocks provides the GNU C compiler for the compling C programs. Our C program including minishell is complied with same complier.
As we do not have access to the Ubuntu in our laptop. We are using virtual environment for using the Ubuntu.

We use Orcale VirtualBox to install Ubuntu Virtually.Our project is based on the linux platform. So, we use Ubuntu as the Linux Platform. We run
our project on the terminal of Ubuntu which create as new environment for the minishell to run. We perform the execution of command in minishell.

4. SYSTEM DESIGN

4.1 The files

Our minishell uses the minishell file coded in .c format. All of the following header files are included in the programs as well as we use system call function to address our file management needs.

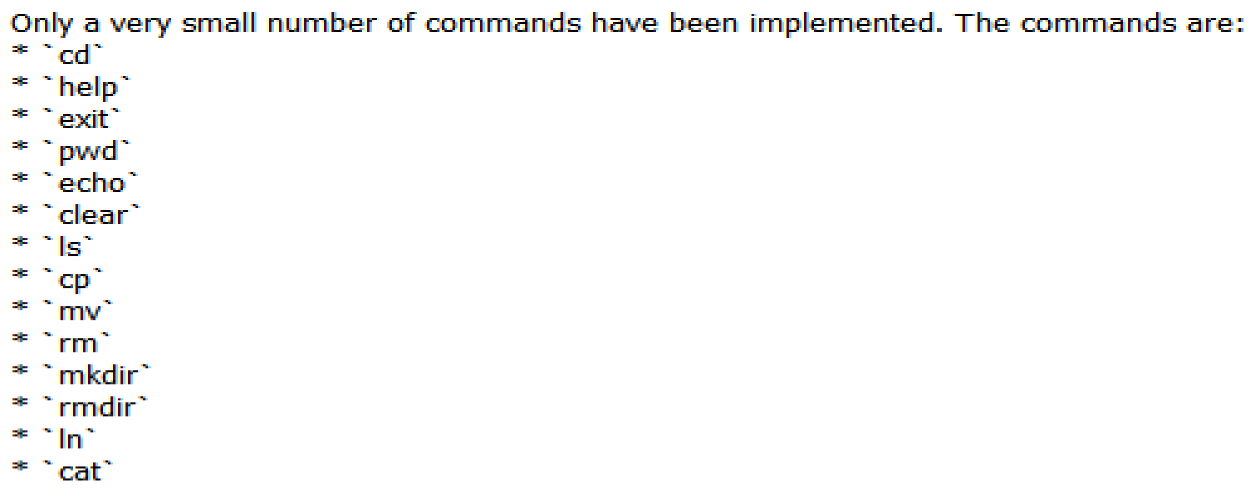
```
#include <stdio.h>          // For: printf(), fprintf(), getchar(), perror(), stderr
#include <stdlib.h>    // For: malloc(), realloc(), free(), exit(), execvp(), EXIT_SUCCESS, EXIT_FAILURE
#include <string.h>    // For: strtok(), strcmp(), strcat(), strcpy()
#include <unistd.h>    // For: chdir(), fork(), exec(), pid_t, getcwd()
#include <sys/wait.h>       // For: waitpid()
#include <fcntl.h>          // For: File creation modes
```

4.2 How it works

When we provide any command to the shell. The shell reads the command from the input and provides the obtained input to the parser which is followed by the execution of the standard input.When the input is passed for the parsing. The command string is separated into a program and arguments. The command in our projects includes ‘exit’ , ‘cd’ , ‘dir’ , ‘del’ , ‘copy’. For example, if the user types the command as make Folder. Then the command is separated into two different types i.e “make” and “Folder”. After the command is parsed then it is passed for the execution . The parsed command is executed to perform some task

4.3 Commands

Our minishell can accept and execute the following commands which is discussed in details below under implementation title.



```
Only a very small number of commands have been implemented. The commands are:
* `cd`
* `help`
* `exit`
* `pwd`
* `echo`
* `clear`
* `ls`
* `cp`
* `mv`
* `rm`
* `mkdir`
* `rmdir`
* `ln`
* `cat`
```

5. IMPLEMENTATION

5.1 Description of Modules/Programs

Shell termination

- Syntax : exit.
- Example : minishell>>exit.

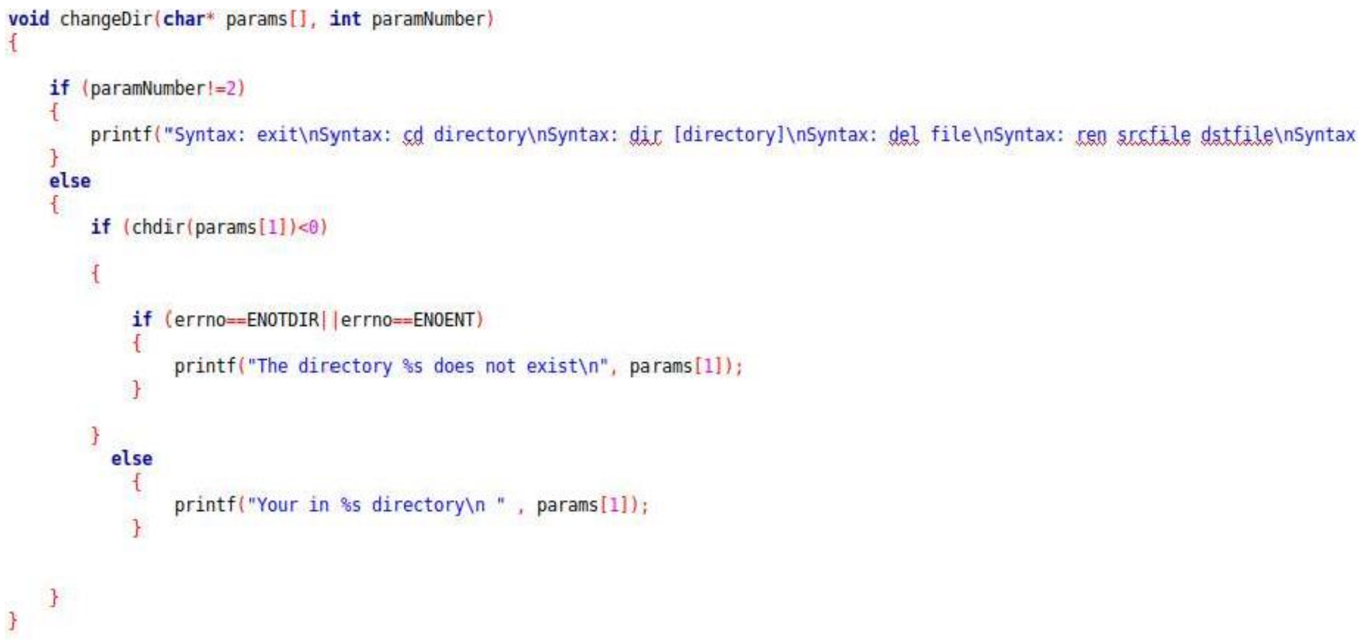
The exit command should terminate the execution of the shell and control should be sreturned to the Linux shell using the void exit(int status) system call.

Change Directory

Syntax : cd directory.
- Example : minishell>cd ~/lab4.

The cd command changes the current working directory of the user to the specified directory (if exists) using the int chdir(const char *path) system call.

- If the specified directory does not exist then the message “The directory directory does not exist” should be displayed on the screen.



```
void changeDir(char* params[], int paramNumber)
{

    if (paramNumber!=2)
    {
        printf("Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax
    }
    else
    {
        if (chdir(params[1])<0)

        {

            if (errno==ENOTDIR||errno==ENOENT)
            {
                printf("The directory %s does not exist\n", params[1]);
            }

        }
          else
            {
                printf("Your in %s directory\n " , params[1]);
            }


    }
}
```

Listing the content of the current directory

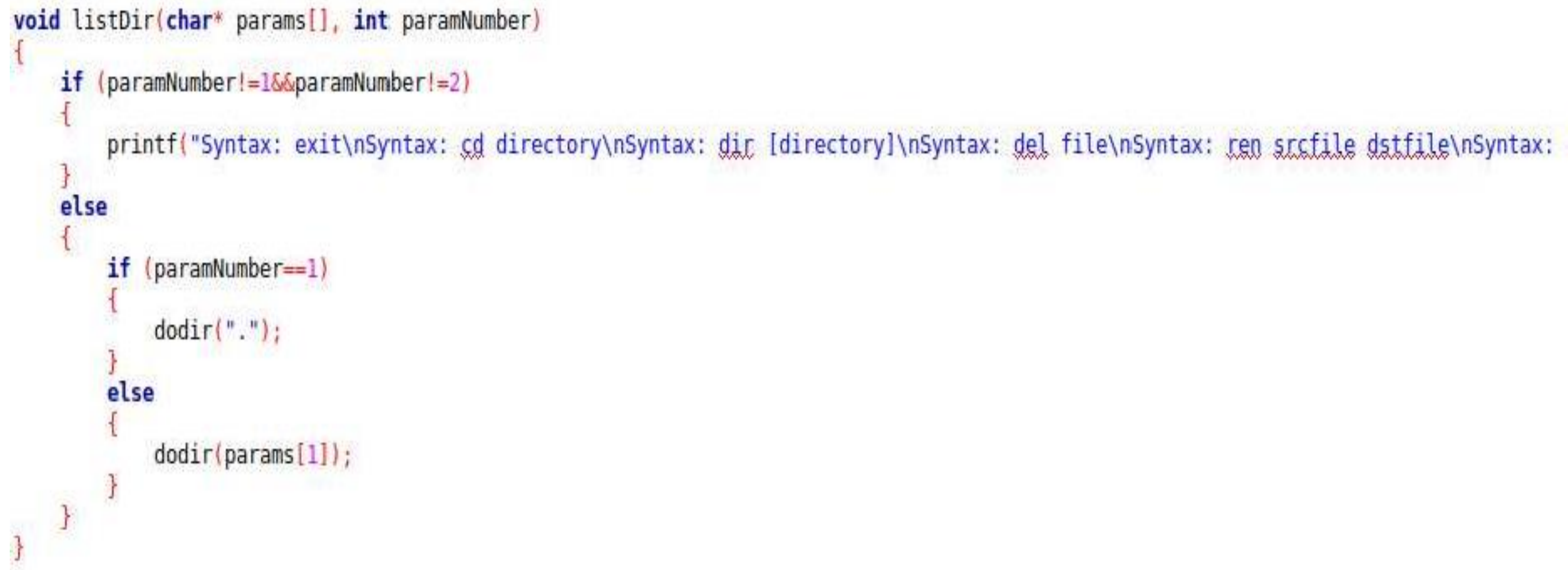
-Syntax : dir [directory].

-Example : minishell>dir ~/lab4.

The dir command lists all the files that are in the specified directory (if exists) using the DIR *opendir(const char *dirname), int closedir(DIR *dirptr), and struct dirent *readdir(DIR *dirptr) system calls. If no directory argument is specified (enclosed brackets indicate an optional argument) then the content of the current directory should be displayed.

- If the specified directory does not exist then the message “The directory directory does not exist” should be displayed on the screen.

- You should implement a function void dir(char *directory) that will call the appropriate system calls.



```
void listDir(char* params[], int paramNumber)
{
    if (paramNumber!=1&&paramNumber!=2)
    {
        printf("Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax:
    }
    else
    {
        if (paramNumber==1)
        {
            dodir(".");
        }
        else
        {
            dodir(params[1]);
        }
    }
}
```

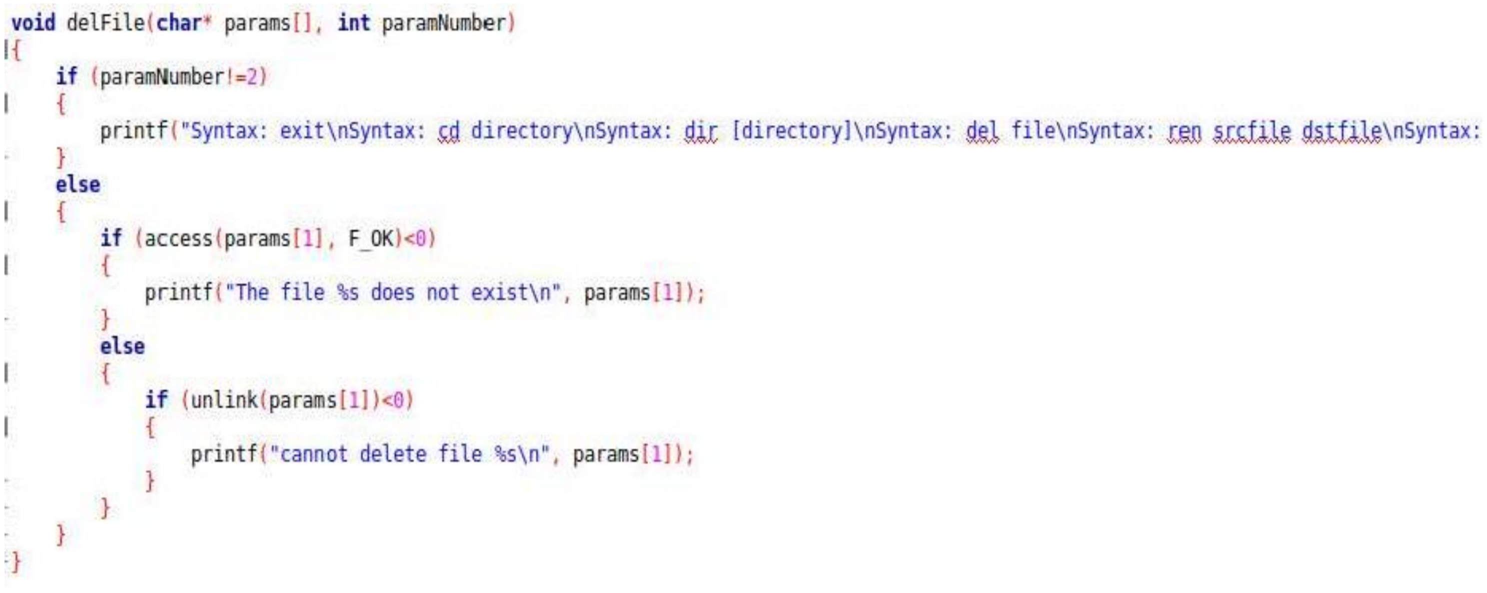
Deleting a file

Syntax : del file.

Example : minishell>del ~/lab4/minishell.o

The del command deletes the specified file using the int unlink(char *pathname) system call.

If the specified file does not exist then the message “The file file does not exist” should be displayed on the screen. File existence should be checked with the int access(char *pathname, int amode) system call.



```
void delFile(char* params[], int paramNumber)
{
    if (paramNumber!=2)
    {
        printf("Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax:
    }
    else
    {
        if (access(params[1], F_OK)<0)
        {
            printf("The file %s does not exist\n", params[1]);
        }
        else
        {
            if (unlink(params[1])<0)
            {
                printf("cannot delete file %s\n", params[1]);
            }
        }
    }
}
```

Renaming a file.

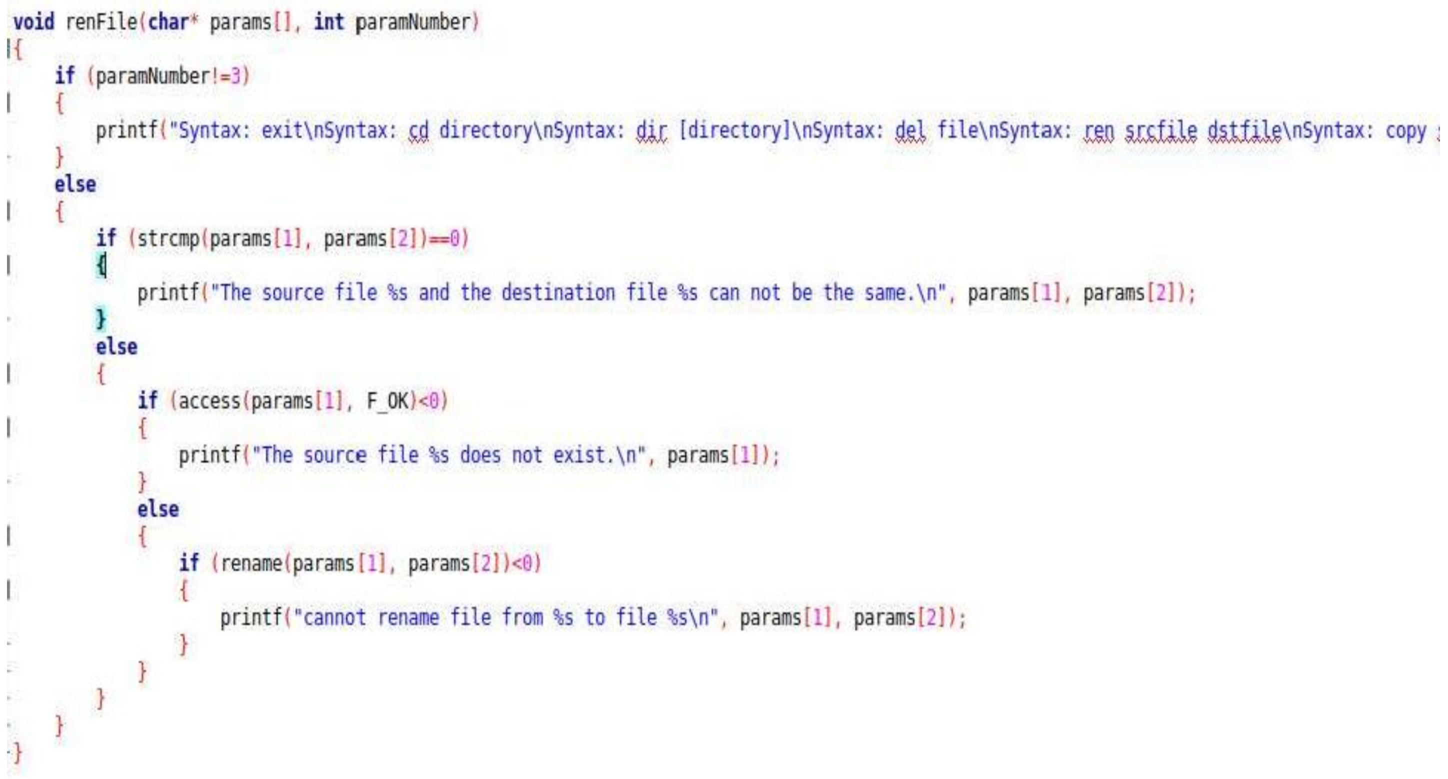
Syntax : ren srcfile dstfile.

Example : minishell>ren ~/lab4/minishell.c ~/lab4/minishell1.c.

The ren command renames a source file srcfile (if exists) into a destination file dstfile using the int rename(const char *oldpath, const char *newpath) system call.

-If the source file srcfile does not exist then the message “The source file srcfile does not exist.” should be displayed on the screen.

-If the source file srcfile and the destination file dstfile are the same then the message “The source file srcfile and the destination file dstfile can not be the same.” should be displayed on the screen.



```
void renFile(char* params[], int paramNumber)
{
    if (paramNumber!=3)
    {
        printf("Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax: copy
    }
    else
    {
        if (strcmp(params[1], params[2])==0)
        {
            printf("The source file %s and the destination file %s can not be the same.\n", params[1], params[2]);
        }
        else
        {
            if (access(params[1], F_OK)<0)
            {
                printf("The source file %s does not exist.\n", params[1]);
            }
            else
            {
                if (rename(params[1], params[2])<0)
                {
                    printf("cannot rename file from %s to file %s\n", params[1], params[2]);
                }
            }
        }
    }
}
```

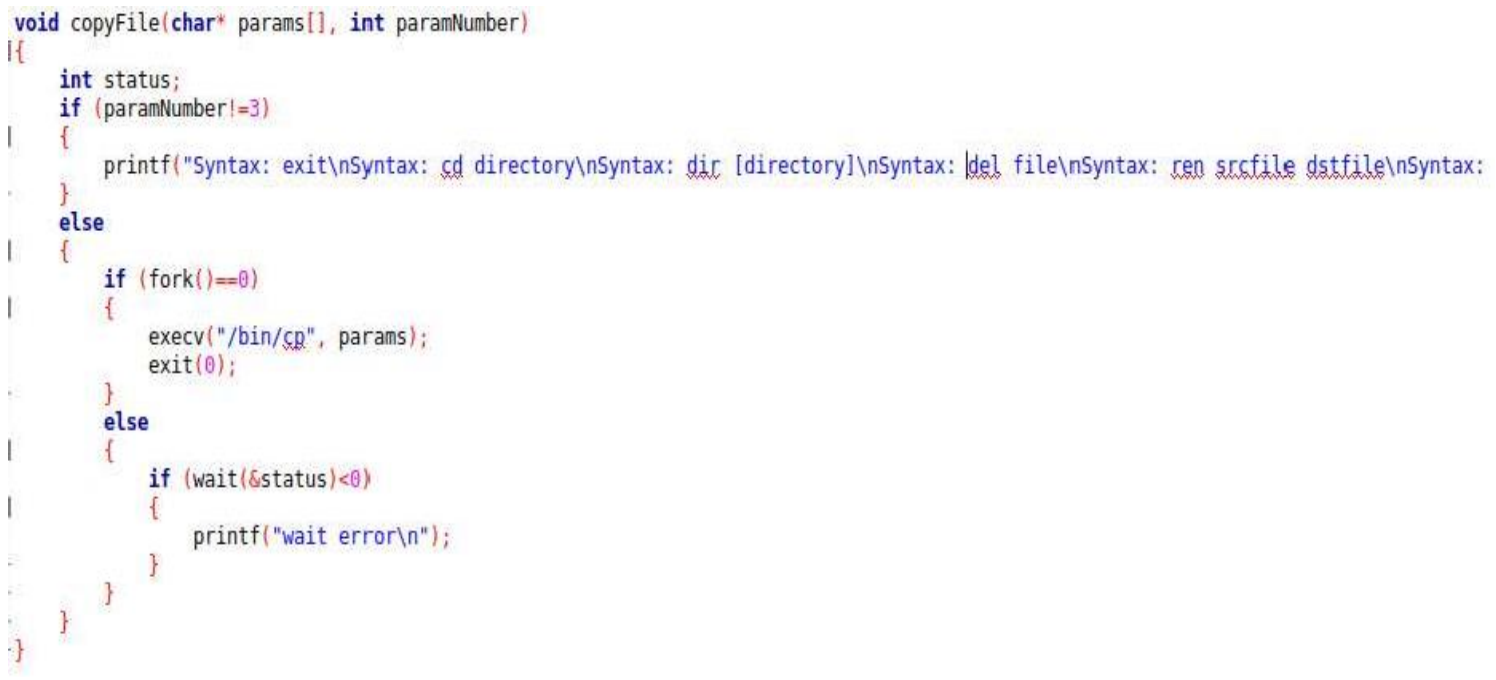
Copying a file.

Syntax : copy srcfile dstfile.

Example : minishell>copy ~/lab4/minishell.o ~/lab4/minishell.o.

The copy command copies a source file srcfile (if exists) to a destination file dstfile by using the Linux cp command. Consequently, the copy command must be implemented as an external minishell command.

Thus, the minishell must create a new child process to run the cp command using the pid_t fork( ), int execv(const char*path, char *const argv[]) and pid_t wait(int *status) system calls



```
void copyFile(char* params[], int paramNumber)
{
    int status;
    if (paramNumber!=3)
    {
        printf("Syntax: exit\nSyntax: cd directory\nSyntax: dir [directory]\nSyntax: del file\nSyntax: ren srcfile dstfile\nSyntax:
    }
    else
    {
        if (fork()==0)
        {
            execv("/bin/cp", params);
            exit(0);
        }
        else
        {
            if (wait(&status)<0)
            {
                printf("wait error\n");
            }
        }
    }
}
```

.

.

SOURCE CODE:

```
/*****************************************************************
 *         File:  miniShell.c
 *        Author:  Ayushnav,Dhruv,Sayujya
 17
 *    Description:  A simple implementation of the Unix Shell in the
 *                    C Programming language.
 *****************************************************************/

#include <stdio.h>       // For: printf(), fprintf(), getchar(), perror(), stderr
#include <stdlib.h>     // For: malloc(), realloc(), free(), exit(), execvp(), EXIT_SUCCESS, EXIT_FAILURE
#include <string.h>     // For: strtok(), strcmp(), strcat(), strcpy()
#include <unistd.h>     // For: chdir(), fork(), exec(), pid_t, getcwd()
#include <sys/wait.h>   // For: waitpid()
#include <fcntl.h>       // For: File creation modes

#define BUILTIN_COMMANDS 5     // Number of builtin commands defined

/*
 * Environment variables
 */
char PWD[1024];                    // Present Working Directory
char PATH[1024];       // Path to find the commands

/*
 * Built-in command names
 */
char * builtin[] = {"cd", "exit", "help", "pwd", "echo"};

/*
 * Built-in command functions
 */

/*
 * Function:  shell_cd
 * -------------------
 *  changes current working directory
 *
 * args: arguments to the cd command, will consider only the first argument after the command name
 */
int shell_cd(char ** args){
        if (args[1] == NULL){
                fprintf(stderr, "minsh: one argument required\n");
        }
        else if (chdir(args[1]) < 0){
                perror("minsh");
        }
        getcwd(PWD, sizeof(PWD));   // Update present working directory
        return 1;
}
```

```
/*
 * Function:  shell_exit
 * --------------------
 *  exits from the shell
 *
 * return: status 0 to indicate termination
 */
int shell_exit(char ** args){
	return 0;
}

/*
 * Function:  shell_help
 * --------------------
 *  prints a small description
 *
 * return: status 1 to indicate successful termination
 */
int shell_help(char ** args){
	printf("\nA mini implementation of the Unix Shell by Ayushnav,Dhruv,Sayujya\n");
	printf("\nCommands implemented: ");
	printf("\n\t- help");
	printf("\n\t- exit");
	printf("\n\t- cd dir");
	printf("\n\t- pwd");
	printf("\n\t- echo [string to echo]");
	printf("\n\t- clear");
	printf("\n\t- ls [-ail] [dir1 dir2 ...]");
	printf("\n\t- cp source target (or) cp file1 [file2 ...] dir");
	printf("\n\t- mv source target (or) mv file1 [file2 ...] dir");
	printf("\n\t- rm file1 [file2 ...]");
	printf("\n\t- mkdir dir1 [dir2 ...]");
	printf("\n\t- rmdir dir1 [dir2 ...]");
	printf("\n\t- ln [-s] source target");
	printf("\n\t- cat [file1 file2 ...]");
	printf("\n\n");
	printf("Other features : ");
	printf("\n\t* Input, Output and Error Redirection (<, <<, >, >>, 2>, 2>> respectively)  : ");
	printf("\n\t* Example: ls -i >> outfile 2> errfile [Space mandatory around redirection operators!]");
	printf("\n\n");
	return 1;
}

/*
 * Function:  shell_pwd
 * -------------------
 *  prints the present working directory
 *
 * return: status 1 to indicate successful termination
 */
int shell_pwd(char ** args){
		printf("%s\n", PWD);
		return 1;
}

/*
 * Function:  shell_echo
 * --------------------
 *  displays the string provided
 *
 * return: status 1 to indicate successful termination
 */
int shell_echo(char ** args){
		int i = 1;
		while (1){
				// End of arguments
				if (args[i] == NULL){
						break;
				}
				printf("%s ", args[i]);
				i++;
		}
		printf("\n");
}

/*
 * Array of function pointers to built-in command functions
 */
int (* builtin_function[]) (char **) = {
		&shell_cd,
		&shell_exit,
		&shell_help,
		&shell_pwd,
		&shell_echo
};


/*
 * Function:  split_command_line
 * ----------------------------
 *  splits a commandline into tokens using strtok()
 *
 * command: a line of command read from terminal
 *
 * returns: an array of pointers to individual tokens
 */
char ** split_command_line(char * command){
	int position = 0;
	int no_of_tokens = 64;
	char ** tokens = malloc(sizeof(char *) * no_of_tokens);
	char delim[2] = " ";

	// Split the command line into tokens with space as delimiter
	char * token = strtok(command, delim);
	while (token != NULL){
		tokens[position] = token;
		position++;
		token = strtok(NULL, delim);
	}
	tokens[position] = NULL;
	return tokens;
}

/*
 * Function:  read_command_line
 * ----------------------------
 *  reads a commandline from terminal
 *
 * returns: a line of command read from terminal
 */
char * read_command_line(void){
	int position = 0;
	int buf_size = 1024;
	char * command = (char *)malloc(sizeof(char) * 1024);
	char c;

	// Read the command line character by character
	c = getchar();
	while (c != EOF && c != '\n'){
		command[position] = c;

		// Reallocate buffer as and when needed
		if (position >= buf_size){
			buf_size += 64;
			command = realloc(command, buf_size);
		}

		position++;
		c = getchar();
	}
	return command;
}

/*
 * Function:  start_process
 * ------------------------
 *  starts and executes a process for a command
 *
 * args: arguments tokenized from the command line
 *
 * return: status 1
 */
int start_process(char ** args){

	int status;
```

```
        pid_t pid, wpid;

        pid = fork();

        if (pid == 0){  // It's the child process

                        // Find the path of the command
                        char cmd_dir[1024];
                        strcpy(cmd_dir, PATH);
                        strcat(cmd_dir, args[0]);

                        // Execute the required process
                        if ( execv( cmd_dir, args ) == -1){ // Error
                                perror("minsh");
                        }

                        exit(EXIT_FAILURE);             // To exit from child process
        }
        else if (pid < 0){      // Error in forking
                        perror("minsh");
        }
        else{           // It's the parent process
                do{
                        wpid = waitpid(pid, &status, WUNTRACED);
                } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        }

        return 1;
}

/*
 * Function:  shell_execute
 * ------------------------
 *  determines and executes a command as a built-in command or an external command
 *
 * args: arguments tokenized from the command line
 *
 * return: return status of the command
 */
int shell_execute(char ** args){

            if (args[0] == NULL){  // Empty command
                        return 1;
            }

            // Copy the current Standard Input and Output file descriptors
            // so they can be restored after executing the current command
            int std_in, std_out, std_err;
            std_in = dup(0);
            std_out = dup(1);
            std_err = dup(2);
```

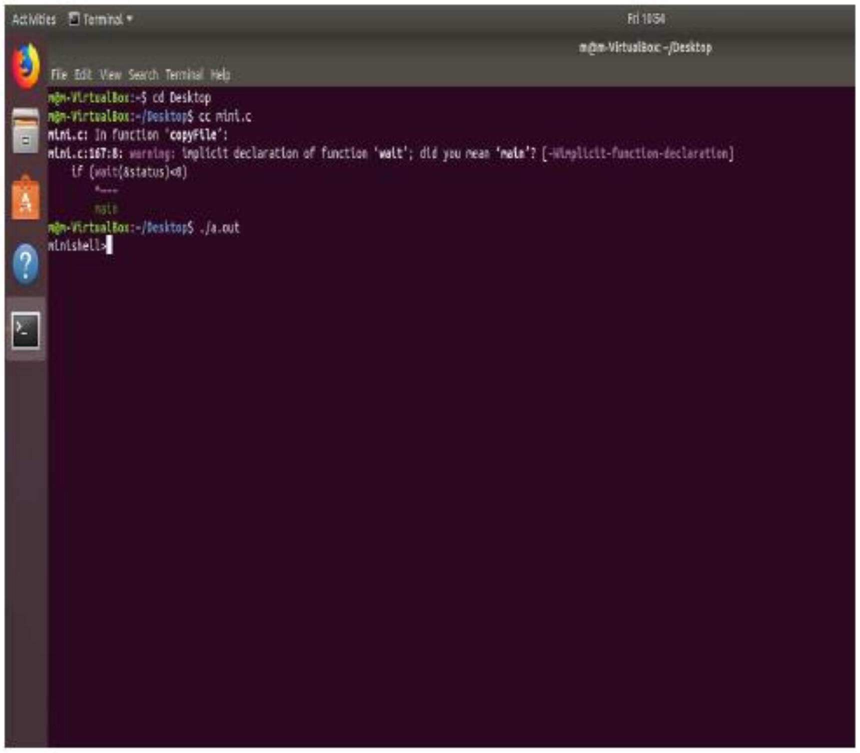
```
// Check if redirection operators are present
int i = 1;

while ( args[i] != NULL ){
        if ( strcmp( args[i], "<" ) == 0 ){          // Input redirection
                int inp = open( args[i+1], O_RDONLY );
                if ( inp < 0 ){
                        perror("minsh");
                        return 1;
                }

                if ( dup2(inp, 0) < 0 ){
                        perror("minsh");
                        return 1;
                }
                close(inp);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
        }
        else if ( strcmp( args[i], "<<" ) == 0 ){ // Input redirection
                int inp = open( args[i+1], O_RDONLY );
                if ( inp < 0 ){

                        perror("minsh");
                        return 1;
                }

                if ( dup2(inp, 0) < 0 ){
                        perror("minsh");
                        return 1;
                }
                close(inp);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
        }
        else if( strcmp( args[i], ">") == 0 ){     // Output redirection

                int out = open( args[i+1], O_WRONLY | O_TRUNC | O_CREAT, 0755 );
                if ( out < 0 ){
                        perror("minsh");
                        return 1;
                }

                if ( dup2(out, 1) < 0 ){
                        perror("minsh");
                        return 1;
                }
                close(out);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
        }
        else if( strcmp( args[i], ">>") == 0 ){   // Output redirection (append)
                int out = open( args[i+1], O_WRONLY | O_APPEND | O_CREAT, 0755 );
                if ( out < 0 ){
                        perror("minsh");
                        return 1;
                }

                if ( dup2(out, 1) < 0 ){
                        perror("minsh");
                        return 1;

                }
                close(out);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
        }
        else if( strcmp( args[i], "2>") == 0 ){   // Error redirection
                int err = open( args[i+1], O_WRONLY | O_CREAT, 0755 );
                if ( err < 0 ){
                        perror("minsh");
                        return 1;
                }

                if ( dup2(err, 2) < 0 ){
                        perror("minsh");
                        return 1;
                }
                close(err);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
        }
        else if( strcmp( args[i], "2>>") == 0 ){ // Error redirection
                int err = open( args[i+1], O_WRONLY | O_CREAT | O_APPEND, 0755 );

                if ( err < 0 ){
                        perror("minsh");
                        return 1;
                }

                if ( dup2(err, 2) < 0 ){
                        perror("minsh");
                        return 1;
                }
                close(err);
                args[i] = NULL;
                args[i+1] = NULL;
                i += 2;
```

```
		}
		else{
			i++;
		}
	}

	// If the command is a built-in command, execute that function
	for(i = 0 ; i < BUILTIN_COMMANDS ; i++){
		if ( strcmp(args[0], builtin[i]) == 0 ){
			int ret_status = (* builtin_function[i])(args);

			// Restore the Standard Input and Output file descriptors
			dup2(std_in, 0);
			dup2(std_out, 1);
			dup2(std_err, 2);

			return ret_status;
		}
	}

	// For other commands, execute a child process
	int ret_status = start_process(args);

	// Restore the Standard Input and Output file descriptors
	dup2(std_in, 0);
	dup2(std_out, 1);
	dup2(std_err, 2);

	return ret_status;
}

/*
 * Function:  shell_loop
 * --------------------
 *  main loop of the Mini-Shell
 */
void shell_loop(void){

	// Display help at startup
	int status = shell_help(NULL);

      char * command_line;
      char ** arguments;
	status = 1;

      while (status){
            printf("minsh> ");
            command_line = read_command_line();
		if ( strcmp(command_line, "") == 0 ){
			continue;
		}
            arguments = split_command_line(command_line);
```

```
                status = shell_execute(arguments);
        }
}

/*
 * Function:  main
 */
int main(int argc, char ** argv){
        // Shell initialization
             getcwd(PWD, sizeof(PWD));   // Initialize PWD Environment Variable
             strcpy(PATH, PWD);           // Initialize the command PATH
             strcat(PATH, "/cmds/");                  // ^^

        // Main loop of the shell
        shell_loop();

        return 0;
}
```

5.2 TEST CASES

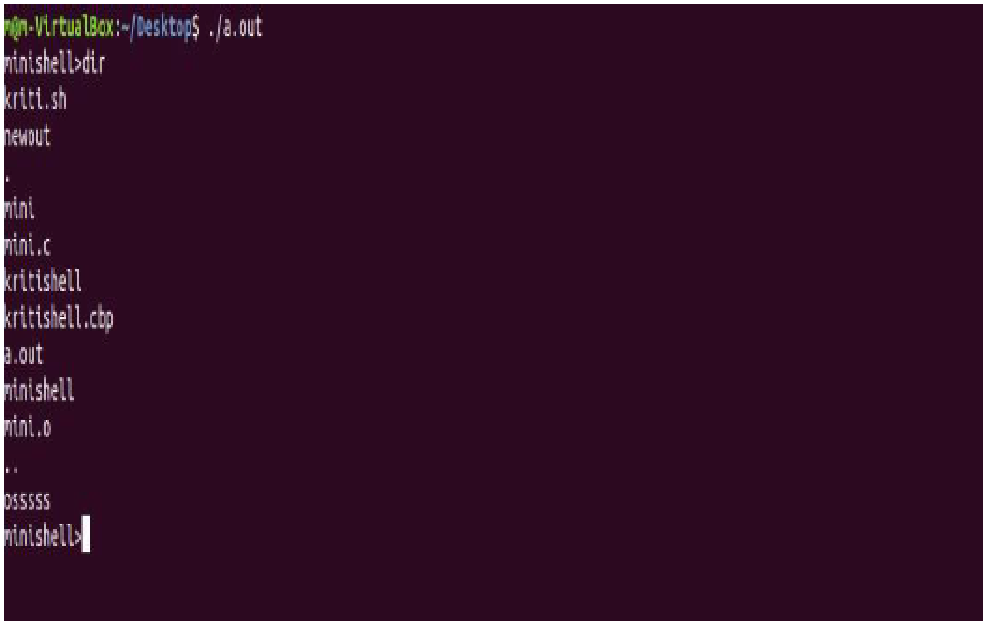
5.3 Execution screenshots



Activities
Terminal
Fri 10:54
m@m-VirtualBox: ~/Desktop
File Edit View Search Terminal Help

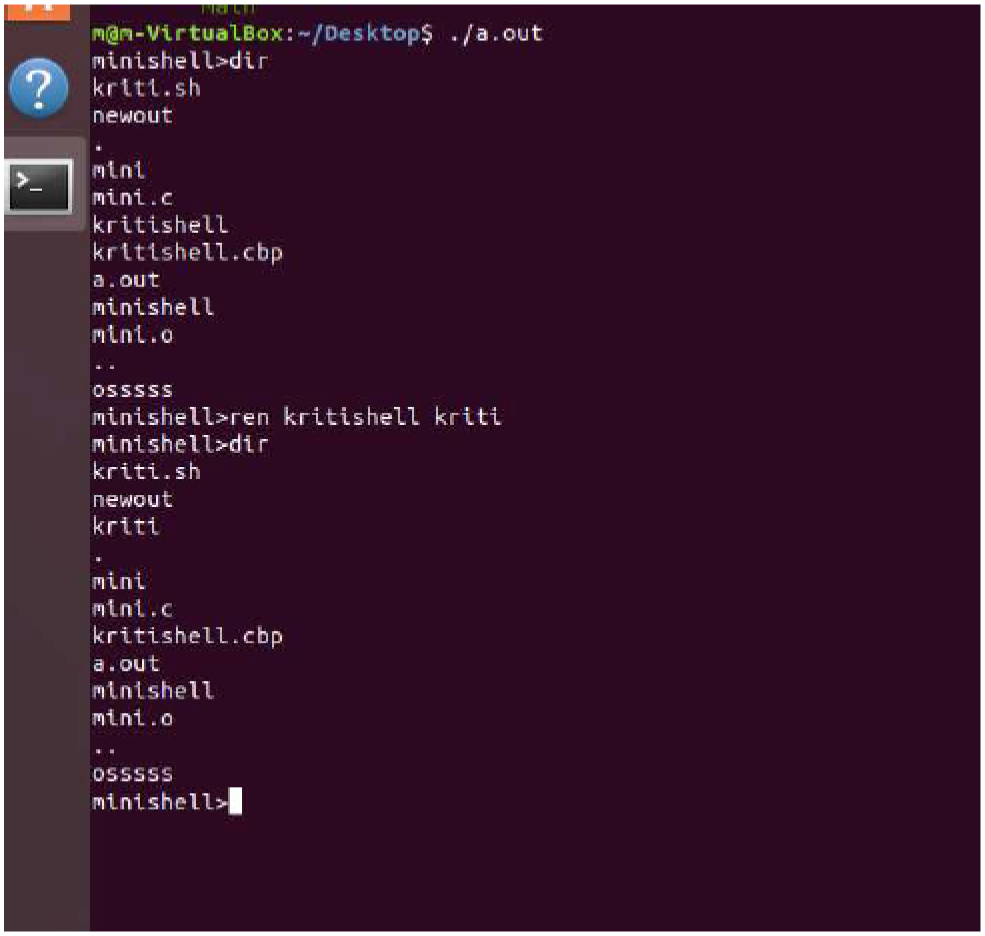
```
m@m-VirtualBox:~$ cd Desktop
m@m-VirtualBox:~/Desktop$ cc mini.c
mini.c: In function 'copyFile':
mini.c:167:8: warning: implicit declaration of function 'wait'; did you mean 'main'? [-Wimplicit-function-declaration]
    if (wait(&status)<0)
        ^~~~
        main
m@m-VirtualBox:~/Desktop$ ./a.out
minishell>
```

Fig 2.1 screenshot of starting the minishell



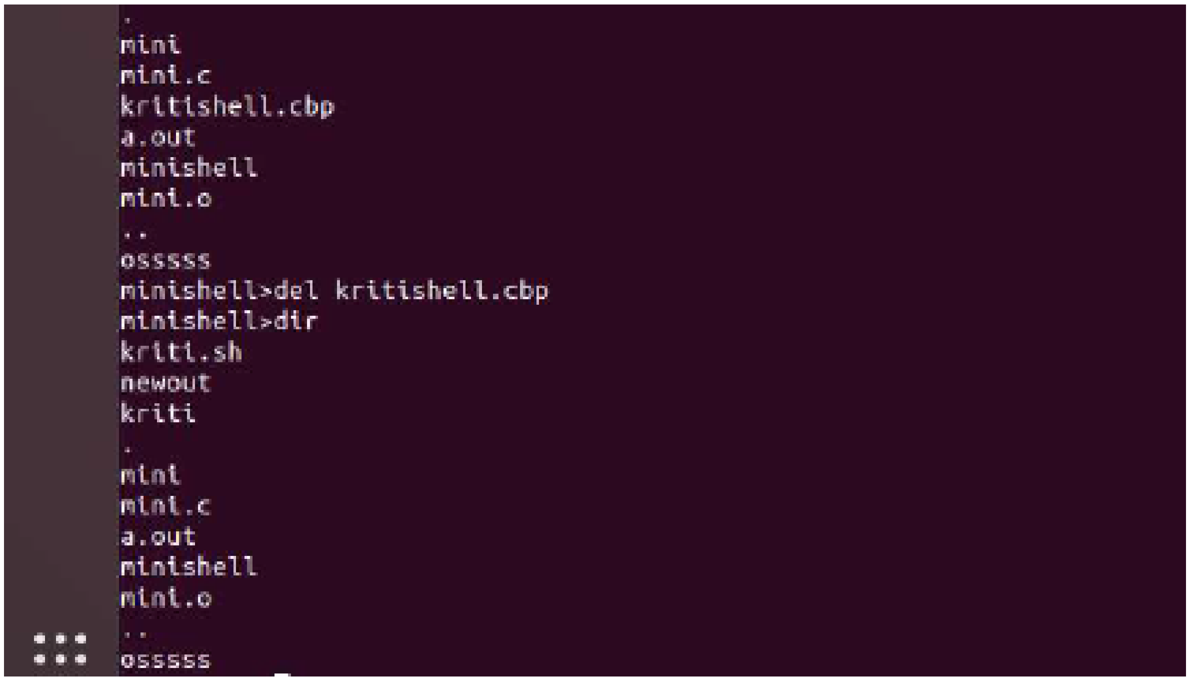
```
m@m-VirtualBox:~/Desktop$ ./a.out
minishell>dir
kriti.sh
newout
.
mini
mini.c
kritishell
kritishell.cbp
a.out
minishell
mini.o
..
osssss
minishell>
```

Fig 2.2 screenshot of the execution of dir command



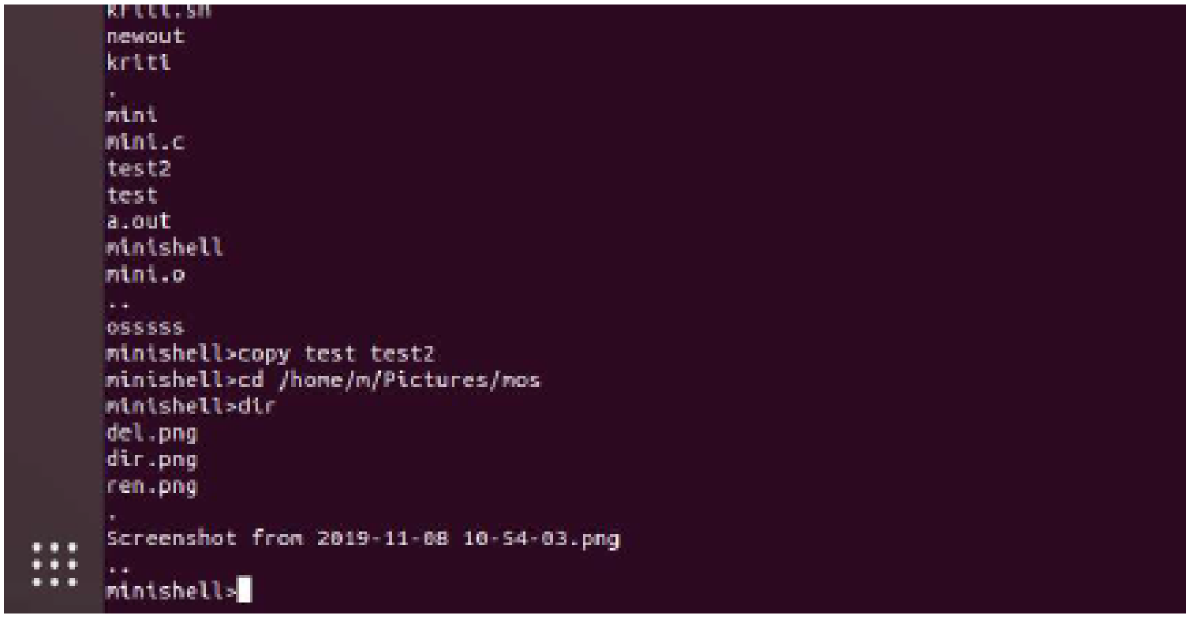
```
m@m-VirtualBox:~/Desktop$ ./a.out
minishell>dir
kriti.sh
newout
.
mini
mini.c
kritishell
kritishell.cbp
a.out
minishell
mini.o
..
osssss
minishell>ren kritishell kriti
minishell>dir
kriti.sh
newout
kriti
.
mini
mini.c
kritishell.cbp
a.out
minishell
mini.o
..
osssss
minishell>
```

Fig 2.3 screenshot of the execution of ren command



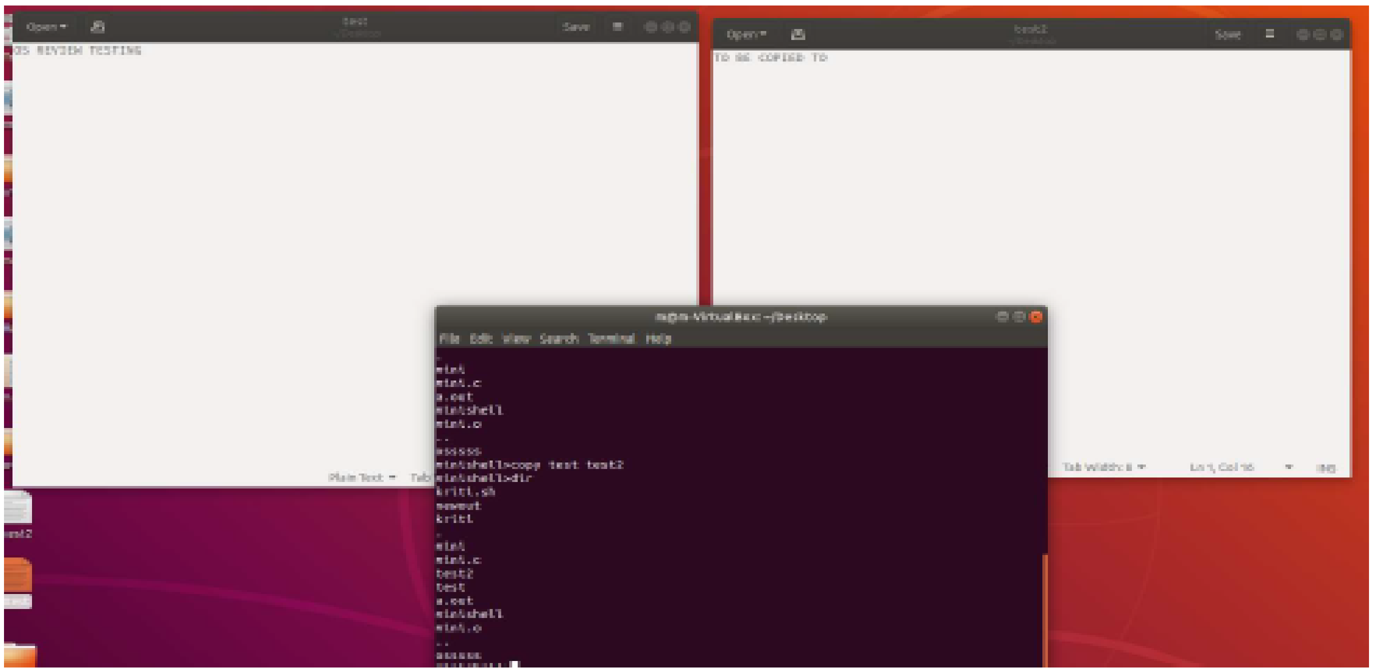
```
.
mini
mini.c
kritishell.cbp
a.out
minishell
mini.o
..
osssss
minishell>del kritishell.cbp
minishell>dir
kriti.sh
newout
kriti
.
mini
mini.c
a.out
minishell
mini.o
..
osssss
```

Fig 2.4 screenshot of the execution of del command



```
kriti.sh
newout
kriti
.
mini
mini.c
test2
test
a.out
minishell
mini.o
..
osssss
minishell>copy test test2
minishell>cd /home/m/Pictures/mos
minishell>dir
del.png
dir.png
ren.png
.
Screenshot from 2019-11-08 10-54-03.png
..
minishell>
```

Fig 2.5 screenshot of the execution of cd command

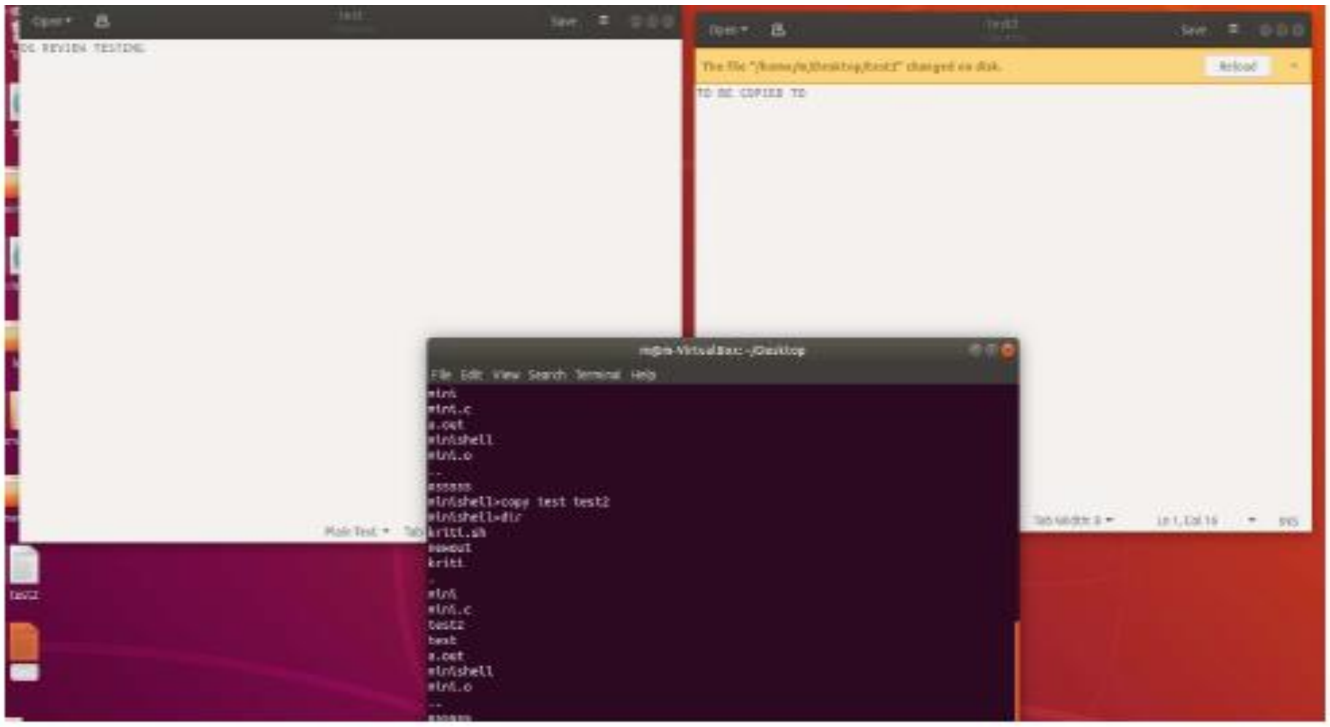


```
OS REVIEW TESTING
```

```
TO BE COPIED TO
```

```
.
mini
mini.c
a.out
minishell
mini.o
..
ssssss
minishell>copy test test2
minishell>dir
kriti.sh
mmmout
kriti
.
mini
mini.c
test2
test
a.out
minishell
mini.o
..
ssssss
```

Fig 2.6.1 screenshot of the execution of copy command (copying the contents of test1 to test2)

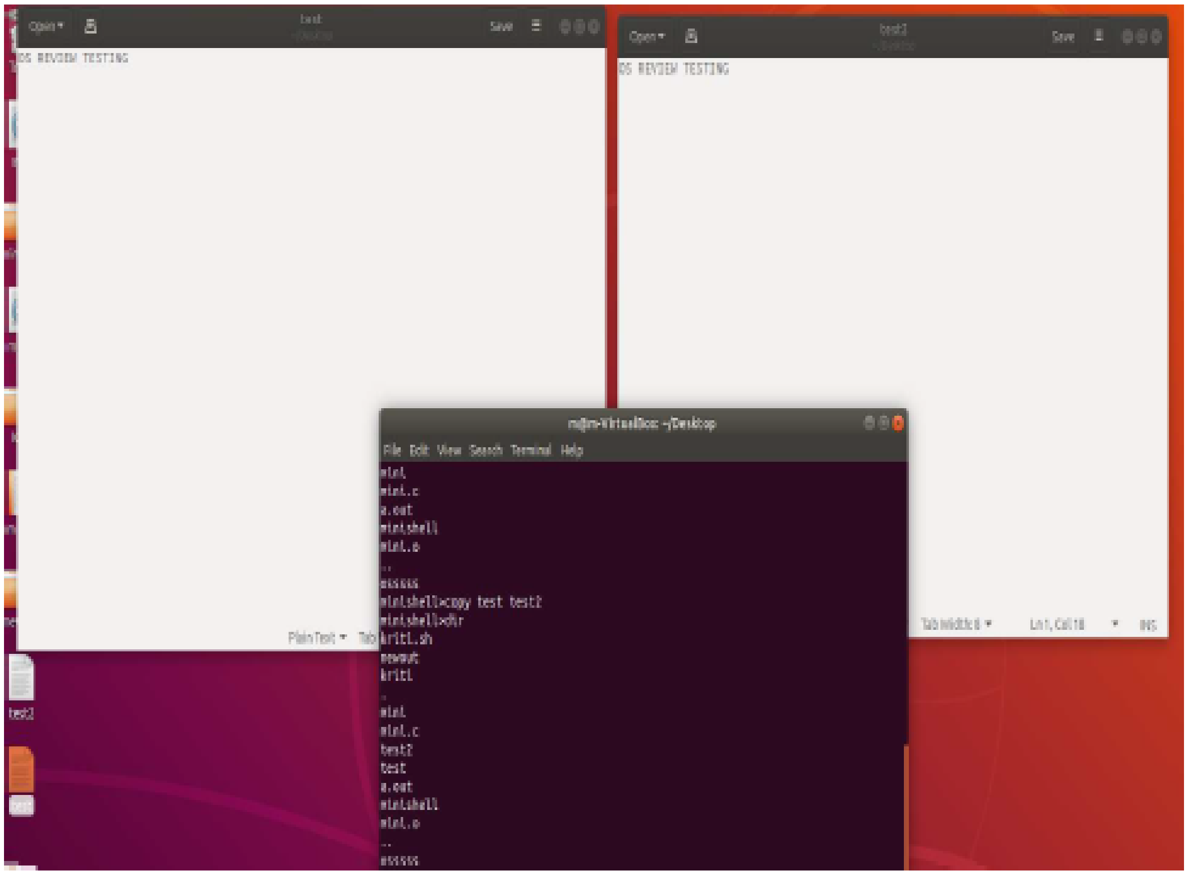


```
OS REVIEW TESTING
```

```
The file "/home/m/Desktop/test2" changed on disk.
TO BE COPIED TO
```

```
mini
mini.c
a.out
minishell
mini.o
..
ssssss
minishell>copy test test2
minishell>dir
kriti.sh
mmmout
kriti
.
mini
mini.c
test2
test
a.out
minishell
mini.o
..
ssssss
```

Fig 2.6.2 screenshot of the execution of copy command (copying the contents of test1 to test2)



```
mini
mini.c
a.out
minishell
mini.o
..
ssssss
minishell>copy test test2
minishell>dir
kriti.sh
newout
kriti
.
mini
mini.c
test2
test
a.out
minishell
mini.o
..
ssssss
```

Fig 2.6.3 screenshot of the execution of copy command(the contents of file 2 has been updated with the contents of file test1)

6. CONCLUSION AND FUTURE REFERENCES

This project is the pre-model for the development of the shell. Our project aims for the file management using minishell. Implementation of our project in more fascinating and innovative ways will lead to development of the new shell which could function similar to the linux shell. Focusing on the file management with new commands will lead to better development of shell with better functionalities. Since our project is based on C programming, the further implementation and change in program's module is not the big Deal.

7. REFERENCES

1. THE DESIGN OF UNIX OPERATING SYSTEM BY MAURICE J. BACHN PAGE NO – 232. SECTION – THE SHELL

2. STEPHEN BRENNAN PAPER ON LINUX SHELL 16 JANUARY 2015