Mini Project Report on

CREATING A SHELL

undergone at

IT DEPARTMENT

under the guidance of

Ms. Sangeetha Suresh Harikantra

Submitted by

NADEER ALI(17IT224)
RYAN ROZARIO(17IT134)
ROHAN BABLI(17IT233)
SULAIMAN MUHAMMAD(17IT143)

III Sem B.Tech (IT)
in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

Department of Information Technology
National Institute of Technology Karnataka, Surathkal.

November2018

CERTIFICATE

This is to certify that the project entitled "CREATING A SHELL" is a Bonafide work carried out by Nadeer Ali(17IT224), Rohan Babli (17IT233), Ryan Rozario(17IT134), Sulaiman Muhammad(17IT143) students of second year Btech. IT Department National Institute of Technology,Karnataka,Surathkal, during the academic Year 2018. We certify that we have carried out the work in our own capacity and have successfully completed the work assigned to us.

PLACE: NITK, Surathkal, Mangalore
DATE:29-11-2018

_____
(Signature of Mentor)

ACKNOWLEDGEMENT

The completion of any work involves the efforts and sacrifices of many people. I have been lucky enough to have received a lot of help and support from many people during the making of this project. So, with great gratitude, I take this opportunity to acknowledge all those people whose guidance and encouragement helped me to emerge successful.

I am greatly indebted to the principal of my college, **Dr. Maheshwar Rao** and my head of the department **Dr. G. Ram Mohana Reddy** for the facilities and support extended towards me.

I consider it as my privilege and honor to express sincere gratitude to my guide, **Ms. Sangeetha Suresh Harikantra**, Assistant Lecturer, Dept. of IT for her invaluable support and encouragement.

I am also much indebted and grateful to the other teaching and non-teaching staff of my department, who extended their unlimited moral support. I would also like to thank my parents and friends for providing continuous encouragement and moral support.

CONTENTS

1. ABSTRACT

The project implements a basic shell which gives the user much access to different varieties of commands the user can use to obtain various results with regards to handling of files, interaction with the computer etc

1.1 HOW DOES THE SHELL WORK?

A shell parses commands entered by the user and executes this. To be able to do this, the workflow of the shell will look like this:

1. Start up the shell
2. Wait for user input
3. Parse user input
4. Execute the command and return the result
5. Go back to `2`.

There is one important piece to all of this though: processes. The shell is the parent process. This is the `'main'` thread of our program which is waiting for user input. However, we cannot execute the command in the `main` thread itself, because of the following reasons:

1. An erroneous command will cause the entire shell to stop working. We want to avoid this.
2. Independent commands should have their own process blocks. This is known as isolation and falls under fault tolerance.

2. INTRODUCTION

2.1 WHAT IS A SHELL?

A shell is a program which provides a user interface. With a shell, users can type in commands and run programs on a Unix system. Basically, the main function a shell performs is to read in from the terminal what one types, run the commands, and show the output of the commands.

2.2 WHY USE A SHELL?

To be truly useful a programming language must provide the following services

- comments,
- variables,
- conditional commands, and
- repeated action commands.

These extra services are provided by the shell. Different shells use different syntax for these services.

2.3 USE OF A SHELL

The main use of the shell is as an interactive shell, but one can write programs using the shell. These programs are called shell scripts.

2.4 FEATURES OF A SHELL

Some of the features of the shell are listed here:

- Customizable environment.
- Abbreviate commands. (Aliases)
- History (Remembers commands typed before.)
- Job Control. (Run programs in the background or foreground.)
- Shell Scripting. (One can write programs using the shell.)
- Keyboard shortcuts.

2.5 BASIC LIFETIME OF A SHELL

Let us 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.

2. 6 BASIC LOOP OF A SHELL

We have taken care of how the program should start up. Now, for the basic program logic: what does the shell do during its loop? 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.

3. LITERATURE SURVEY

The first Unix shell was the Thompson shell, *sh*, written by Ken Thompson at Bell Labs and distributed with Versions 1 through 6 of Unix, from 1971 to 1975. Though rudimentary by modern standards, it introduced many of the basic features common to all later Unix shells, including piping, simple control structures using if and goto, and filename wildcarding. Though not in current use, it is still available as part of some Ancient UNIX systems.

The PWB shell or Mashey shell, *sh*, was an upward-compatible version of the Thompson shell, augmented by John Mashey and others and distributed with the Programmer's Workbench UNIX, circa 1975-1977. It focused on making shell programming practical, especially in large shared computing centers. It added shell variables (precursors of environment variables, including the search path mechanism that evolved into $PATH), user-executable shell scripts, and interrupt-handling. Control structures were extended from if/goto to if/then/else/endif, switch/breaksw/endsw, and while/end/break/continue. As shell programming became widespread, these external commands were incorporated into the shell itself for performance.

But the most widely distributed and influential of the early Unix shells were the Bourne shell and the C shell. Both shells have been used as the coding base and model for many derivative and work-alike shells with extended feature sets.

3.1 BOURNE SHELL

The Bourne shell, sh, was a complete rewrite by Stephen Bourne at Bell Labs. Distributed as the shell for UNIX Version 7 in 1979, it introduced the rest of the basic features considered common to all the Unix shells, including here documents, command substitution, more generic variables and more extensive builtin control structures. The language, including the use of a reversed keyword to mark the end of a block, was influenced by ALGOL 68. Traditionally, the Bourne shell program name is sh and its path in the Unix file system hierarchy is /bin/sh. But a number of compatible work-alikes are also available with various improvements and additional features.

3.2 C SHELL

The C shell, csh, was written by Bill Joy while a graduate student at University of California, Berkeley and widely distributed with BSD Unix. The language, including the control structures and the expression grammar, was modeled on C. The C shell also introduced a large number of features for interactive work, including the history and editing mechanisms, aliases, directory stacks, tilde notation, cdpath, job control and path hashing. On many systems, csh may be a symbolic link or hard link to TENEX C shell (tcsh), an improved version of Joy's original csh. Though the C shell's interactive features have been copied in most other current shells, the language itself has not been widely copied. The only work-alike is Hamilton C shell, written by Nicole Hamilton, first distributed on OS/2 in 1988 and on Windows since 1992.

4. PROBLEM DEFINITION

Shell scripts are used to automate administrative tasks, encapsulate complex configuration details and get at the full power of the operating system. The ability to combine commands allows you to create new commands, thereby adding value to your operating system. Furthermore, combining a shell with graphical desktop environment allows you to get the best of both worlds

- Automate your daily tasks
- Create your own commands with optionally accepting input from the user
- Portability, executing the same script in your mac and your Linux based systems.

5. IMPLEMENTATION

5.1 THE CODE

5.1.1 THE MAIN

```c
int main(int argc,char** agrv){
    //load the configuration
    shell_init();

    //run an REPL loop
    repl_loop();
    //Perform any shutdown cleanup
    return 0;
}
```

Calls the init function which is the interface for the program as well as the loop function. It takes in command line arguments in the form of argc and argv.

5.1.2 INTERFACE FUNCTION

```c
void shell_init(void){
    clear();
    printf("This is a shell created for a project of IT202\n");
    printf("UNIX PROJECT IT202 NITK\n");
    printf("Team Members: Ryan, Nadeer, Sulaiman, Rohan\n");
    sleep(2);
    clear();
}
```

Just an interface for the program before it proceeds

5.1.3 LOOPING FUNCTION

```
void repl_loop(void){
    char *line;
    char **args;
    int status;
    char* user=getenv("USER");
    //read commands from user
    //parse the command
    //execute
    do{
        printf("\n%s@it202@",user);
        printDir();
        line=cmd_read();


        args=cmd_split(line);
        status=cmd_execute(args);

        //free memory
        free(line);
        free(args);
    }while(!status);


}
```

This function is used repeatedly in the program. It displays the USER name in the terminal(getenv()) as well as the current working directory(printDir()).

It then calls the read, split, and execute functions sequentially in a continuous do while loop until the USER wants to exit.

At the end of the loop it frees up space using the free function for line and args

5.1.4 READ FUNCTION

```c
#define T_CMD_SIZE 1024

char* cmd_read(void){
    int cmd_size = T_CMD_SIZE;
    char *cmd_str = malloc(cmd_size * sizeof(char));
    int pos = 0;
    int c;

    if(cmd_str){
        while(1){
            c = getchar();

            if (c == EOF || c == '\n'){
                    cmd_str[pos] = '\0';
                    return cmd_str;
            }
            else {
                    cmd_str[pos] = c;
            }

            pos++;
            if (pos>cmd_size){
                cmd_size += T_CMD_SIZE;
                cmd_str = realloc(cmd_str,cmd_size * sizeof(char*));

                if(!cmd_str){
                    printf("Space not allocated\n");
                    exit(EXIT_FAILURE);
                }
            }
        }
    }
    else{
        printf("Space not allocated\n");
        exit(EXIT_FAILURE);
    }
}
```

This function basically interprets the line entered by the user.

It allocates an initial fixed size for the user to enter(cmd_size). Then it is allocated to cmd_str where cmd_str will take in the inputted line.

It reads character by character and updates it to cmd_str until we reach the end of line

If the user has inputted more data than initially expected by cmd_size, we allocate more memory to cmd_size and subsequently to cmd_str and then proceed.

cmd_str is then returned.

5.1.5 SPLIT FUNCTION

```
#define CMD_SIZE 64
#define DELIM " "

char** cmd_split(char* line){
    int cmd_size = CMD_SIZE;
    char **blocks = malloc(cmd_size * sizeof(char*));
    char *cmd_arg;
    int pos = 0;

    if(blocks){
        cmd_arg = strtok(line, DELIM);
        while (cmd_arg != NULL) {
                blocks[pos] = cmd_arg;
                pos++;

                if (pos>cmd_size){
                    cmd_size += CMD_SIZE;
                    blocks = realloc(blocks,cmd_size * sizeof(char*));

                    if(!blocks){
                        printf("Space not allocated\n");
                        exit(EXIT_FAILURE);
                    }

                }


                cmd_arg = strtok(NULL, DELIM);
        }
    }
    else{
        printf("Space not allocated\n");
        exit(EXIT_FAILURE);
    }
    free(cmd_arg);
    blocks[pos] = NULL;
    return blocks;

}
```

This function like read function, allocates memory to ‘blocks’. cmd_arg is separated into words by a delimiter “ “ using the strtok function.

Each word is then given to blocks and returned back to repl_loop

5.1.6 EXECUTE FUNCTION

```
int cmd_execute(char **args){
    int i;
    if (args[0] == NULL) {
    //empty command
        return 1;
    }
    int n_cmd=sizeof(builtin_str) / sizeof(char *);

    for (i = 0; i<n_cmd; i++){
        if (strcmp(args[0], builtin_str[i]) == 0) {

            return (*builtin_func[i])(args);
        }
    }
    return run_external(args);
}
```

It reads blocks from the split function and checks if any of the words are part of the built functions or an external command. It then calls the required functions if it is true.

5.1.7 BUILT-IN ARRAY

```
char *builtin_str[] = {
    "cd",
    "help",
    "exit",
    "find",
    "wc"
};
```

It gives the list of built functions

5.1.8 REFERENCE TO BUILT-IN FUNCTION

```
int (*builtin_func[]) (char **) = {
    &built_cd,
    &built_help,
    &built_exit,
    &built_find,
    &built_wc
};
```

Gives a reference to the built functions

5.1.9 FOR-EXTERNAL COMMANDS FUNCTION

```c
int run_external(char **args){
    pid_t pid = fork();
    if (pid == 0){
        if (execvp(args[0],args) < 0 ){
            perror("The command could not be found");
            exit(EXIT_FAILURE);
        }
    }
    else if (pid < 0) {
        perror("Failed to fork the child");
        exit(EXIT_FAILURE);
    }
    else{
        //waiting for child to terminate
        //better implementation would be to check all chid processes
        wait(NULL); // DOOUUBBTTT
        return 0;
    }
}
```

This function would run if in the execute function, this function was called.

The fork function is called to make a child process and the execvp function is applied to the required word given to it by the execute function.

It then checks if the command given is part of the bin folder by going through all of them. If it exists, the command is executed. Otherwise if not found, an error message would result.

5.1.10 EXIT, HELP AND CD FUNCTIONS

```c
int built_exit(char **args){
    return 1;
}

int built_help(char **args){
    int i;
    printf("UNIX PROJECT IT202 NITK\n");
    printf("Team Members: Ryan, Nadeer, Sulaiman, Rohan\n");
    printf("The following are built in:\n");
    int n_cmd=sizeof(builtin_str) / sizeof(char*);
    for (i = 0; i < n_cmd; i++){
        printf("  %s\n", builtin_str[i]);
    }
    return 0;
}


int built_cd(char **args){
    if (args[1] == NULL) {
        fprintf(stderr, "Arguments expected to cd");
    }
    else {
        if (chdir(args[1]) != 0) {
            perror("Not a valid directory");
        }
    }
    return 0;
}
```

Exit, help and cd are built functions.

Exit function would exit the program

Help function shows the available built functions

cd function helps us to go to a directory using the chdir function

5.1.11 FIND FUNCTION

```c
#define LINE_LENGTH 300;
int built_find(char **args){
    if (args[1] == NULL || args[2] == NULL) {
        fprintf(stderr, "Arguments expected to find");
        return 0;
    }
    FILE *fp;
    int l=LINE_LENGTH;
    char *line= malloc(l * sizeof(char));
    fp=fopen(args[1],"r");
    if(!fp){
        perror("The file is not present");
        return 0;
    }
    int ln=0;
    while(fgets(line,l,fp)!=NULL){
        ln++;
        if(strstr(line,args[2])){
            printf("%d:%s\n",ln,line);
        }
    }
    fclose(fp);
    free(line);
    return 0;
}
```

This function is a built function.

This function is implemented by using file operations which open a file and check if the argument matches with any of the words in the file using the strstr function.

If found, the line containing the word is displayed.

5.1.12 WC FUNCTION

```
int built_wc(char **args){
    if (args[1] == NULL) {
        fprintf(stderr, "Arguments expected to wc");
        return 0;
    }
    FILE *fp;
    int l=LINE_LENGTH;
    char *line= malloc(l * sizeof(char));
    fp=fopen(args[1],"r");
    if(!fp)
    {
        int i=1;
        int n=0;
        while(args[i]!=NULL){
            n++;
            i++;

        }
        printf("%d",n);
        return 0;

    }
    else
    {
        int c, n=0;
        //code = malloc(1000);
        while ((c = fgetc(fp)) != EOF)
        {
            if (c==' ' || c=='\n')
                n++;
        }
        printf("%d",n);
    fclose(fp);
    free(line);
    return 0;


}
}
```

This function is a built function.

This function can do either of 2 things. a) counts the number of words in a command
b) counts the number of words in a file

If a), It counts the number of words using a counter associated with ‘args’ which contains all the words in the command

If b), It reads the words by using a counter and “ “ and “\n” as delimiters.

5.2 OUTPUTS

5.2.1 INTERFACE

```
This is a shell created for a project of IT202
UNIX PROJECT IT202 NITK
Team Members: Ryan, Nadeer, Sulaiman, Rohan
```

5.2.2 OPENING LOOK (SHOWS CURRENT WORKING DIRECTORY)

```
nadeer@it202@Dir:/home/nadeer/Documents>>
```

5.2.3 MAKE DIRECTORY

```
nadeer@it202@Dir:/home/nadeer/Documents>>mkdir testf
nadeer@it202@Dir:/home/nadeer/Documents>>cd testf
nadeer@it202@Dir:/home/nadeer/Documents/testf>>
```

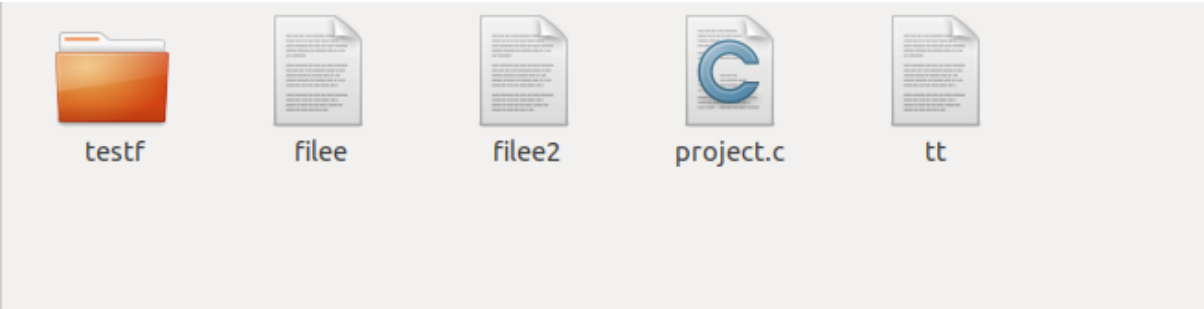
5.2.4 CD COMMAND

```
nadeer@it202@Dir:/home/nadeer/Documents>>cd testf
nadeer@it202@Dir:/home/nadeer/Documents/testf>>cd ..
nadeer@it202@Dir:/home/nadeer/Documents>>
```

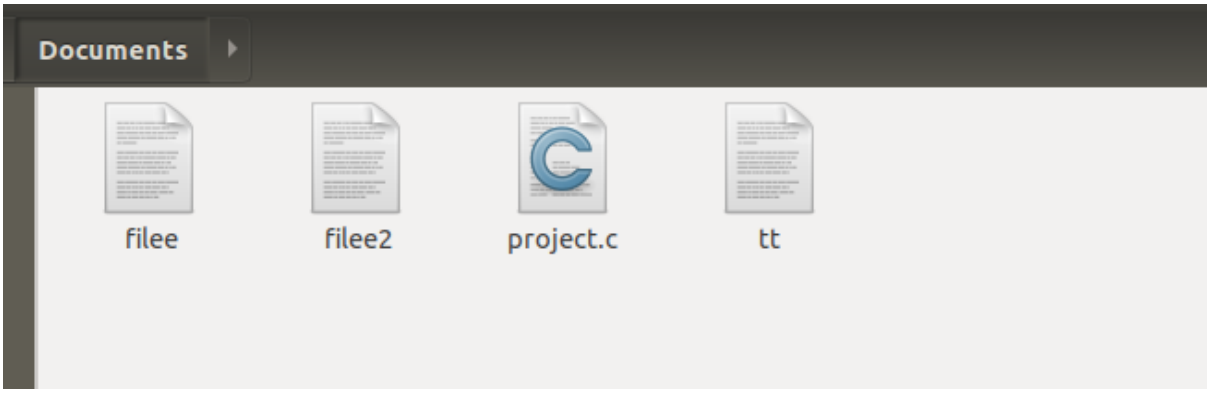
5.2.5 DELETING DIRECTORY

```
nadeer@it202@Dir:/home/nadeer/Documents>>rmdir testf
nadeer@it202@Dir:/home/nadeer/Documents>>
```

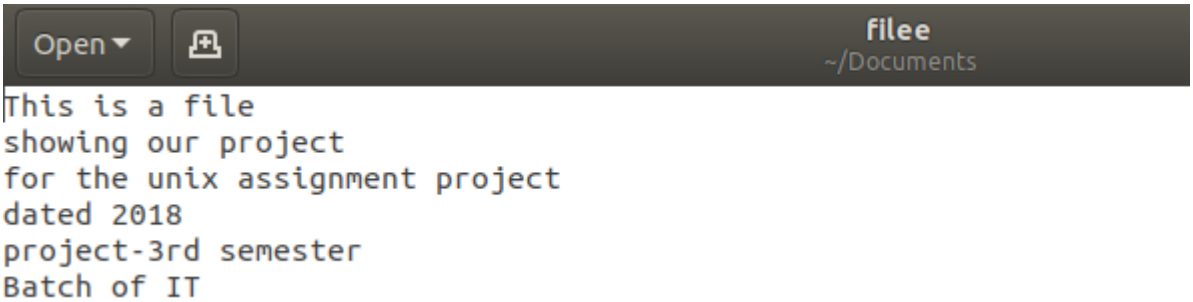
A) BEFORE REMOVING DIRECTORY



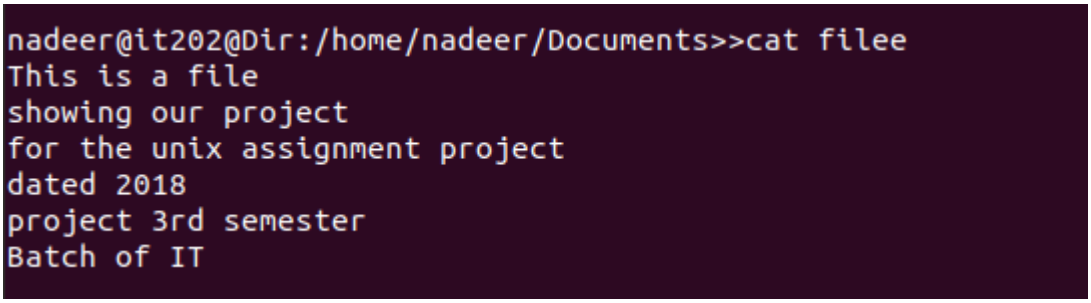
B) AFTER REMOVING



5.2.6 A TEXT FILE THAT IS CREATED



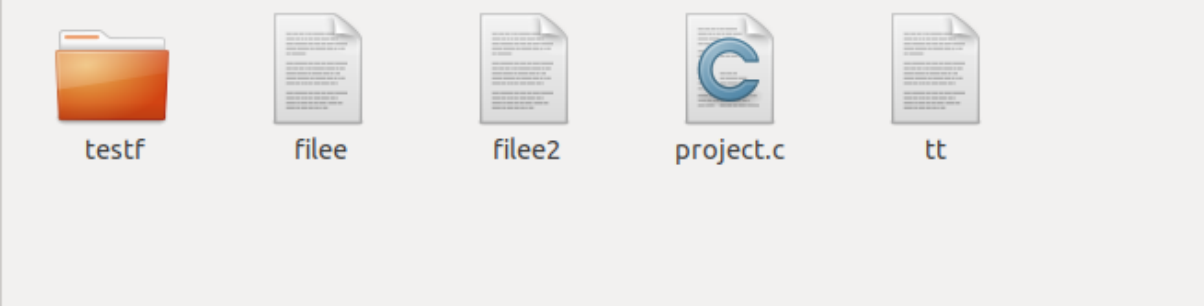
5.2.7 CAT COMMAND



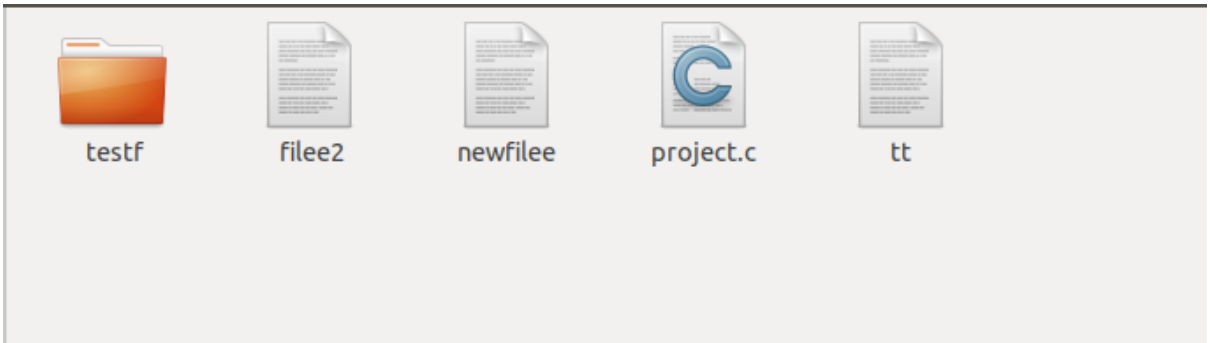
5.2.8 MV COMMAND

```
nadeer@it202@Dir:/home/nadeer/Documents>>mv filee newfilee
```

A) BEFORE MV



B) AFTER MV



5.2.9 FIND COMMAND

```
nadeer@it202@Dir:/home/nadeer/Documents>>find filee project
2:showing our project

3:for the unix assignment project

5:project-3rd semester


nadeer@it202@Dir:/home/nadeer/Documents>>
```

5.2.10 WC FUNCTION

A)FOR A COMMAND

```
nadeer@it202@Dir:/home/nadeer/Documents>>wc this is our project
4
nadeer@it202@Dir:/home/nadeer/Documents>>
```

B) FOR A FILE

```
nadeer@it202@Dir:/home/nadeer/Documents>>wc filee
20
nadeer@it202@Dir:/home/nadeer/Documents>>
```

5.2.11 HELP FUNCTION

```
nadeer@it202@Dir:/home/nadeer/Documents>>help
UNIX PROJECT IT202 NITK
Team Members: Ryan, Nadeer, Sulaiman, Rohan
The following are built in:
  cd
  help
  exit
  find
  wc

nadeer@it202@Dir:/home/nadeer/Documents>>
```

5.2.12 EXIT FUNCTION

```
sulaiman@it202@Dir:/home/sulaiman/Desktop/unix project>>help
UNIX PROJECT IT202 NITK
Team Members: Ryan, Nadeer, Sulaiman, Rohan
The following are built in:
  cd
  help
  exit
  find
  wc

sulaiman@it202@Dir:/home/sulaiman/Desktop/unix project>>exit
sulaiman@sulaiman-Lenovo-Y520-15IKBN:~/Desktop/unix project$
```

6. CONCLUSION

The project implemented a basic shell giving the user an array of useful commands.

These commands are able to handle and modify files and various other tasks.

7. FUTURE WORK

1. Implementation of pipelining

2. Job scheduling

8. REFERENCES

1. https://indradhanush.github.io/blog/writing-a-unix-shell-part-1/

2. https://indradhanush.github.io/blog/writing-a-unix-shell-part-2/

3. https://indradhanush.github.io/blog/writing-a-unix-shell-part-3/

4. https://www.geeksforgeeks.org/making-linux-shell-c/

5. https://brennan.io/2015/01/16/write-a-shell-in-c/