SPL-1 Project Report

slsh: Simple Linux Shell

Submitted by

Istiaq Ahmed Fahad

BSSE Roll No: 04

BSSE Session: 2019-2020

Submitted to

Dr. Kazi Muheymin-Us-Sakib

Professor

Institute of Information Technology

Institute of Information Technology

University of Dhaka

30-05-2022

Table of Contents

Index of Figure

1. Introduction

In everyday life, we deal with different kinds of software. Most of us are concerned about how these software work, how they are built, and so on. But the topic that is least talked about is the operating system. We hardly care about the operating system, as we all are by default windows users. But there is a lot more to explore than just go on with windows. And the reason why we should care about the Operating system is because of the diverse usage of electronic devices and services. The operating system of each device is responsible for system services, required software libraries, run time components, and device drivers. We are concerned about software but without this essential hardware-related software (firmware) our device is nothing but a blank box. So, the next question arose, how to explore the wide domain of operating systems? To start exploration with the operating system, Linux would be the one-stop solution for beginners. Due to being an open-source operating system, it is easy to extract the implementational details which will help us to explore it more minutely than any other operating system.

In an operating system, the kernel plays a vital role to control all system-related activities. Surprisingly it is true that whatever software we use, they communicate with the kernel and the kernel accomplishes the specific task on behalf of user requests. Again it raises another question, how do this kernel and software are connected and communicate with each other? Well, in that case, the shell plays a key role in making the connection between the kernel and the software. Shell is an interface present between the kernel and the user. It is primarily used to access the services provided by the operating system. Now, what if we have our shell that is capable of communicating with the kernel? It would be great, as it will clarify our concept of the operating system and its working principle. And this project is all about making a shell from scratch. This project will make one confident to work with the operating systems as it has demonstrated very core and common features of the shell. As building a shell from scratch is a tedious task, we have tried to focus on the common aspects of a shell and get into the minimal details that are easy to comprehend for beginners.

In general, a shell contains numerous features including shell operations, functions, redirections, command executions, expansions, and so on. As this project is designed for beginners, we will cover only the basic features including shell command execution, command parsing, and tokenization, shell expansions, pipelining, input-output redirection, command corrections, and terminal colorization. Therefore, this shell will be able to handle multiple argument-based commands, pipelined commands, wildcard-based statements, and aliasing frequently used commands. Shell is primarily built with C and thus so this project, so would be quite relevant to the actual shell program. Users can easily relate its functional details with that of the actual shell.

The shell comes in different flavors. Bourne shell (bsh), Bourne-again Shell (bash), Korne shell (ksl), c shell (csh), and many more variants are available. This project is fully focused on bash. Therefore, we will implement a basic bash in this project.

2. Background of the Project

Before we deep dive into the details of the project, some terms should be clarified for proper comprehension of the implementation. Here we will jot down some core and important concepts and terminologies related to that project.

- **Command:** Shell commands are a sequence of strings that represent some instructions that instruct the system to perform specific actions. Shell commands consist of some arguments that are delimited by space and preceded by the actual command.

  Shell commands primarily come into two types; simple and complex commands. Simple commands are nothing but some arguments delimited by space or semicolons. And when multiple simple commands are concatenated with pipeline, redirections, looping, or grouping cumulatively, it is called complex commands. In this project, we will see both of these implementations.

- **Wildcard:** This is commonly known as a pattern-matching character. When we need to perform file name or directory name substitution, we use the wildcard character. The basic wildcard character includes '?', '*', '!'

- **Shell Session:** Session denotes the current environment/state of the terminal when we start executing the shell program. In simple terms, when a user starts a shell in the terminal and waits till exiting the program, the whole duration is noted as a shell session. One can have only one shell session at a time.

- **Pipeline:** Pipelining is one of the core features of the Unix shell. It enables the connection of the output of one program to the input of another. It is denoted as a pipe '|' symbol. Using this feature, we can pass the input or output to another command.

- **Tokens:** When a group of characters forms a collective meaning, then this sequence of characters is called tokens. Tokens are also defined as a single logical unit. It can be found in different varieties. Some of the common tokens are identifiers, keywords, operators, and special characters.

- **Parser:** Parser is a process of analyzing a sequence of strings. It is also known as syntax analysis or syntactic analysis. It takes a string, breaks them into tokens and the output is shown as a form of the parse tree.

3. Description of the Project

My project is designed to demonstrate the basic features of the Linux shell (bash). It will act as a clone of bash and it includes the following attributes:

- Executing Shell Commands
    - Read user input
    - Breaks into tokens
    - Parsing tokens
    - Perform shell expansions
    - Execute commands
- Command Name Customization (aliasing)
- Pipelined Command Execution
- Input-Output Redirection
- Terminal Color Customization
- Shell History Management
- Command Correction

These are the features that have been covered in the whole project. Now let us get into the details of each feature to learn about their working principles:

3.1 Executing Shell Commands

This is the core feature of my project where we will execute shell commands following some sequential process. Initially, the shell asks for the user input. Then this input will be broken into tokens so that necessary aliasing and command expansion can be performed. The tokens will be parsed to create relevant shell commands. Here these commands may include pipelining or command redirections. Later, it will be checked if the command is built-in or external. If it is built-in then execution will be redirected to system calls. Otherwise, for external commands, the shell will search the executable file for further execution. After the execution is complete, the user will be asked for the next command again and this Read-Evaluate-Print-Loop (REPL) will be continued till the user exit the shell. This whole process is summed up in figure []

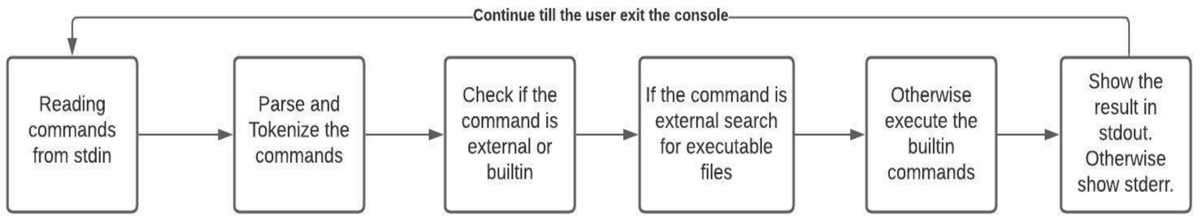

Continue till the user exit the console
Reading commands from stdin
Parse and Tokenize the commands
Check if the command is external or builtin
If the command is external search for executable files
Otherwise execute the builtin commands
Show the result in stdout. Otherwise show stderr.

Figure 1: Shell Command Execution Cycle

3.2 Command Name Customization (Aliasing)

Users can assign short names or refactor shell commands through aliasing. By using aliases, users can save a lot of time while doing tasks frequently. In that feature, we split the command arguments into two parts. The first part, followed by the keyword 'alias', is the actual command. And the second part is the aliased command. We map these two commands so that the actual command can be replaced with aliased command before execution is performed.

3.3 Pipelined Command Execution

A pipeline is a collection of one or more instructions partitioned by the '|' control operator. The output of each command in the pipeline is linked to the following command's input through a pipe. It is a very common scenario when we need to execute multiple commands at the same time. And pipelined command meets this requirement to execute multiple commands simultaneously. We have used the pipe function to link the input-output of multiple commands. And to represent the standard input-output stream we have used the file descriptors. These descriptors act as the placeholder for stdin, stdout, and stderr.

3.4 Input Output Redirection

Redirection allows us to change the standard input (stdin) and standard output (stdout) while executing shell commands in the terminal. It works with commands' file handles. This file handle can be replicated, opened, or closed and even can change the file the command read from and writes to. Before implementing this feature, we have to extract the input-output file names as well as their relevant redirection sequence. This is the most challenging part of that feature. Later on, after extracting the file names, we use the file descriptors. By replicating the descriptors, we redirect the stdin and stdout to the files mentioned in the shell command.

3.5 Terminal Color Customization

Terminal color customization denotes changing the color of the system prompt and terminal background. Sometimes working in the same environment seems monotonous. And this monotonous can be relinquished by using different variants of the terminal. However, we have integrated the ANSI codes in the eventloop of the shell to manipulate the prompt and background color.

3.6 Shell History Management

Our shell keeps track of user command history. Users can invoke specific commands using the history expansion character (!). Sometimes some important commands might need in an emergency and in that case, history plays a vital role. Initially, for a new user, our shell creates a new history file to store user-given commands. Later on, whenever the program starts, our shell search for the latest history serial number, so that the new history can be appended from that point. Also, these serial numbers are useful for invoking any particular command without typing it repeatedly.

3.7 Command Correction

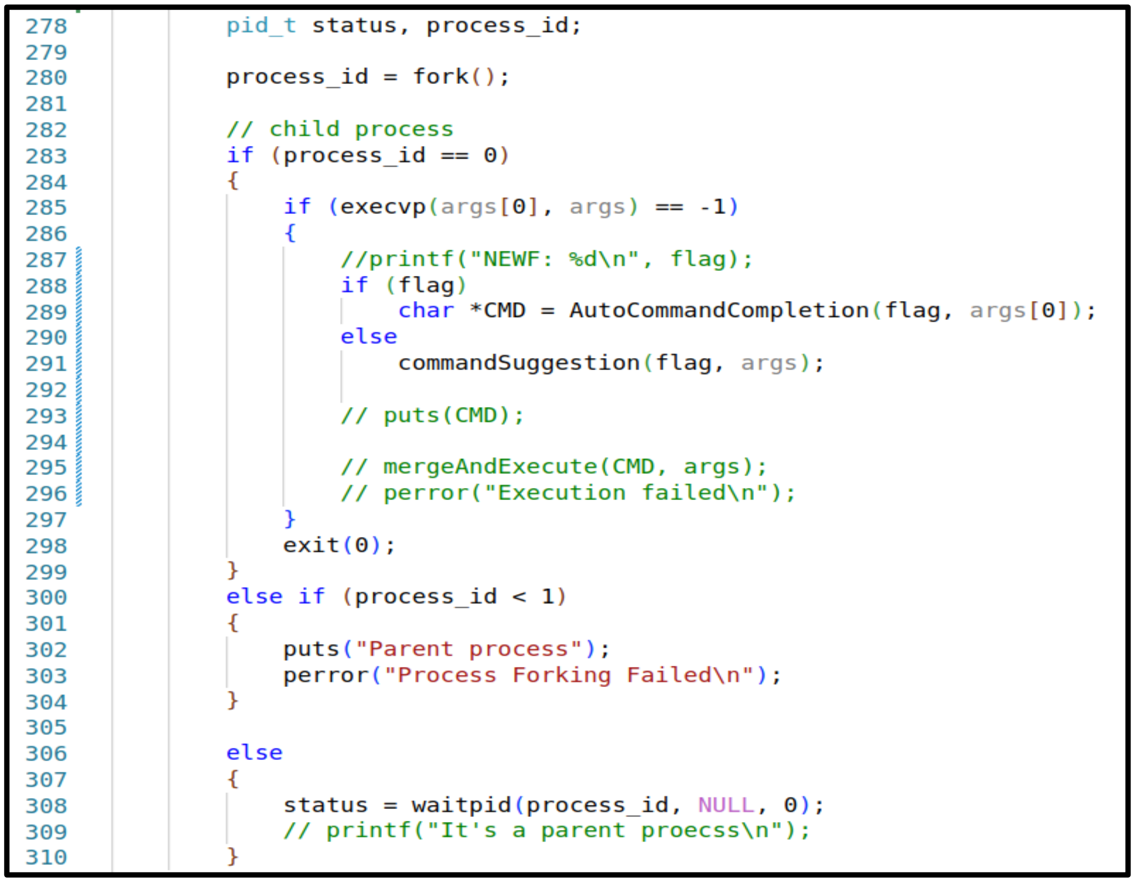
It is quite common that we type the wrong shell commands with or without the actual command. In that case, this feature will help the user to understand the relevant shell command if he/she puts any wrong commands. This feature is quite useful for those who are new to Linux and learning commands by trial and error. Initially, it will suggest to the user what can be the closest command to the given wrong command. In autocompletion mode, our shell will suggest the most accurate command by inspecting user command history.

4. Implementation and Testing

This section covers the details of how I have implemented the features mentioned above in my project. Each subsection will contain the implementation details. Also, relevant code snippets have been attached after each description for better comprehension.

4.1 Executing Shell Commands

Command execution starts with taking user input. We maintained a while loop that will continuously ask the user to provide input and will continue until the user exit the loop. However, after taking input through the **take_user_input** function the string is forwarded to the **processPipelinedCommand** function to check if it contains any pipeline or not. If so, then we go for the **executePipelinedCommands** function for further command execution. Otherwise, we pass the string to the **porcessSingleCommand** function for further modification and passed to the **execute** function for execution. Thus, the whole process of command execution goes on until the user **exit** the loop and closes our program. A code snippet of the **execute** function is shown in figure 2.

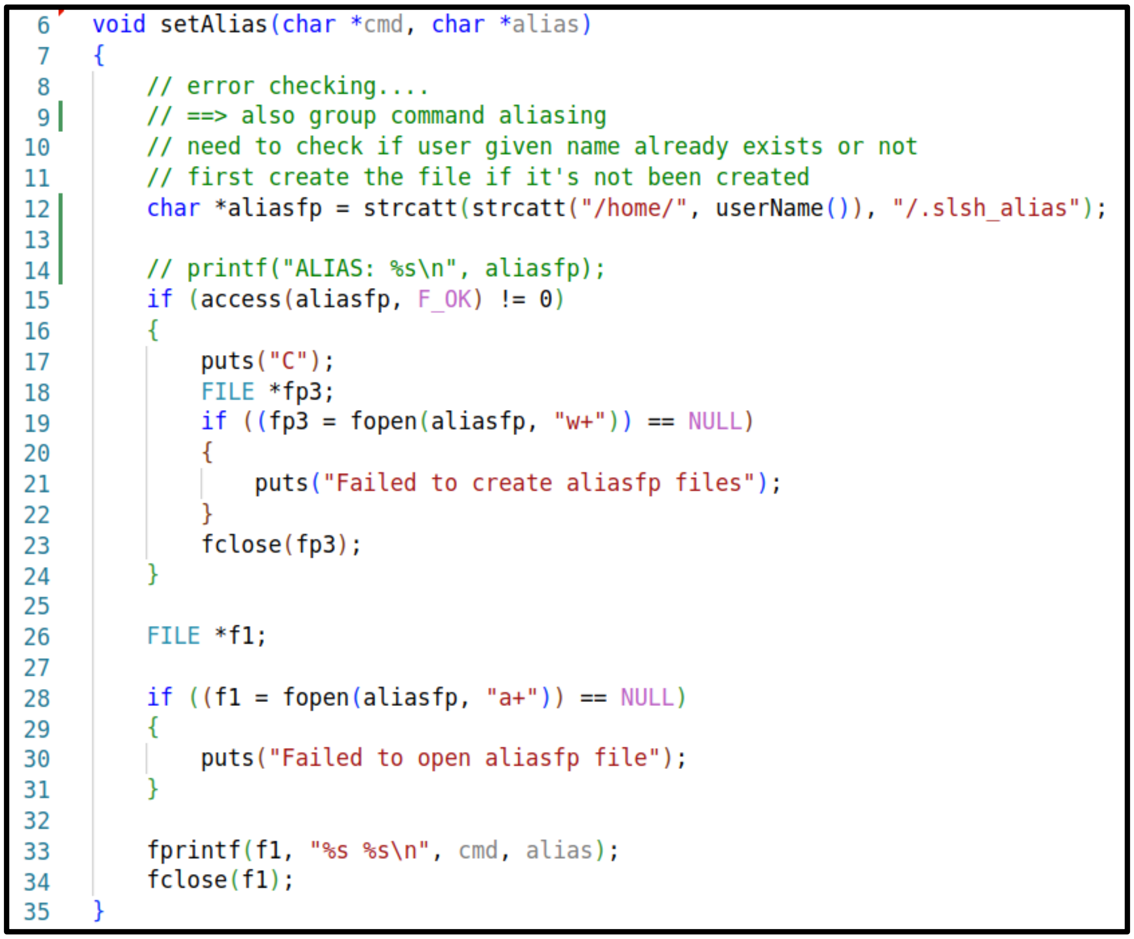


```
        pid_t status, process_id;

        process_id = fork();

        // child process
        if (process_id == 0)
        {
            if (execvp(args[0], args) == -1)
            {
                //printf("NEWF: %d\n", flag);
                if (flag)
                    char *CMD = AutoCommandCompletion(flag, args[0]);
                else
                    commandSuggestion(flag, args);

                // puts(CMD);

                // mergeAndExecute(CMD, args);
                // perror("Execution failed\n");
            }
            exit(0);
        }
        else if (process_id < 1)
        {
            puts("Parent process");
            perror("Process Forking Failed\n");
        }

        else
        {
            status = waitpid(process_id, NULL, 0);
            // printf("It's a parent proecss\n");
        }
```

Figure 2: Process Execution Code Snippet

4.2 Command Name Customization (Aliasing)

When a string is passed for parsing, this feature comes to inaction. Initially, we check each parsed token whether it contains any alias or not through the **checkForAliasing** function. Then if the token contains any alias, we replace this alias with the actual shell command. In addition, when a user alias commands a set new command, control goes to the **aliasCommands** function and it processes the set of arguments through the **setAlias** method for setting new alias commands. The execution details of the **setAlias** method are depicted in figure 3.

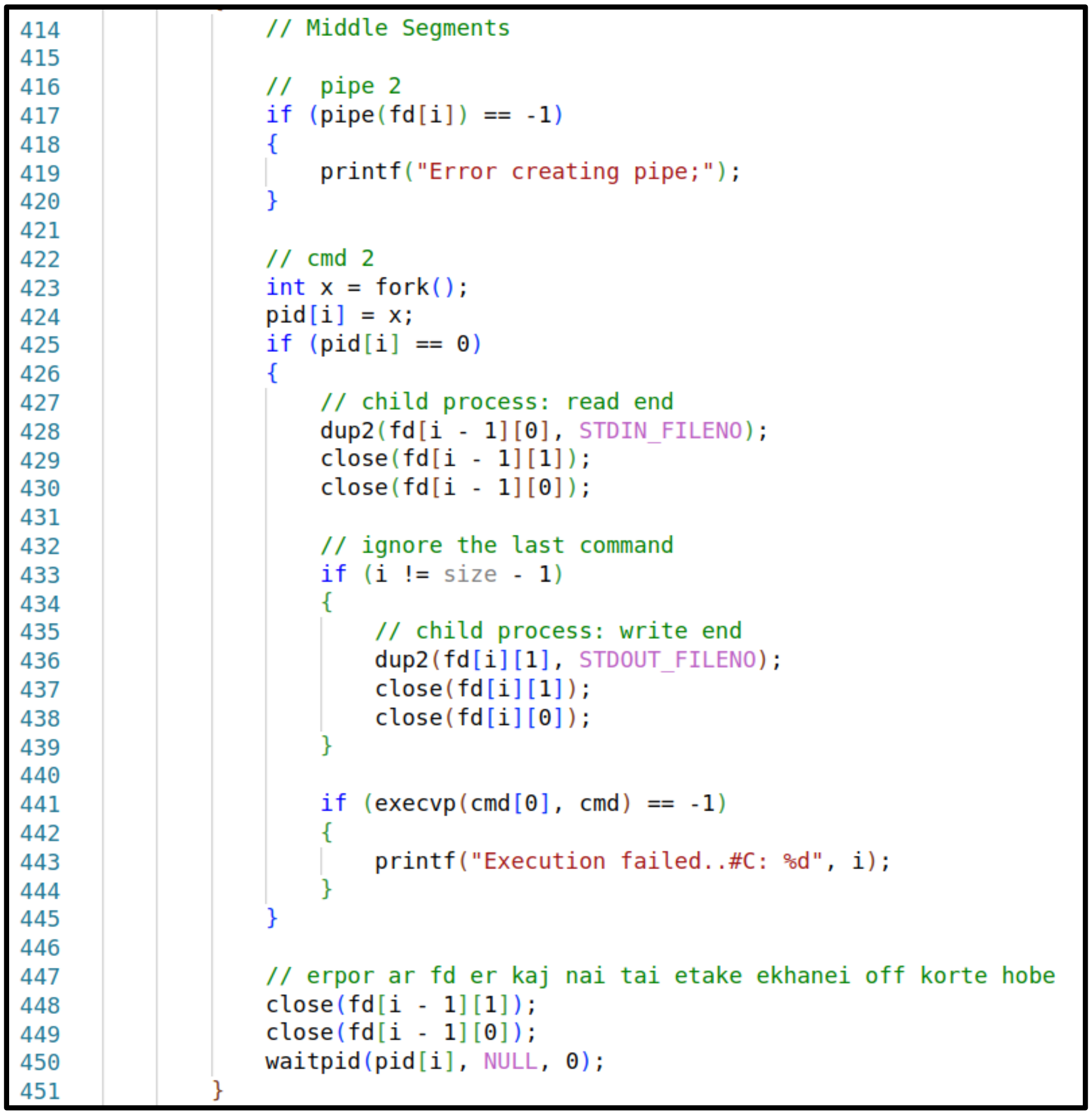


```
void setAlias(char *cmd, char *alias)
{
    // error checking....
    // ==> also group command aliasing
    // need to check if user given name already exists or not
    // first create the file if it's not been created
    char *aliasfp = strcatt(strcatt("/home/", userName()), "/.slsh_alias");

    // printf("ALIAS: %s\n", aliasfp);
    if (access(aliasfp, F_OK) != 0)
    {
        puts("C");
        FILE *fp3;
        if ((fp3 = fopen(aliasfp, "w+")) == NULL)
        {
            puts("Failed to create aliasfp files");
        }
        fclose(fp3);
    }

    FILE *f1;

    if ((f1 = fopen(aliasfp, "a+")) == NULL)
    {
        puts("Failed to open aliasfp file");
    }

    fprintf(f1, "%s %s\n", cmd, alias);
    fclose(f1);
}
```

Figure 3: Set Aliased Command Code Snippet

4.3 Pipelined Command Execution

Previously we have seen how a command executes form user given instructions. When the given string is passed to the **processPipelinedCommand** method, pipeline command execution begins. In further steps, we look for the pipe '|' symbol and redirection symbol '>', '<' for proper command modification. Based on this, we extract the set of arguments and passed the parsed string to the **executePipelinedCommands** method for final execution. This method deals with both pipelines and redirecting multiple commands. *Command redirection* will be discussed in detail in the following sub-section. However, in the executePipelinedCommands method, we used the concept of *file descriptor* to denote proper standard input-output. Also, we have used the **pipe** system method to communicate between newly designated stdin and stdout. The way it is accomplished is shown in figure 4.

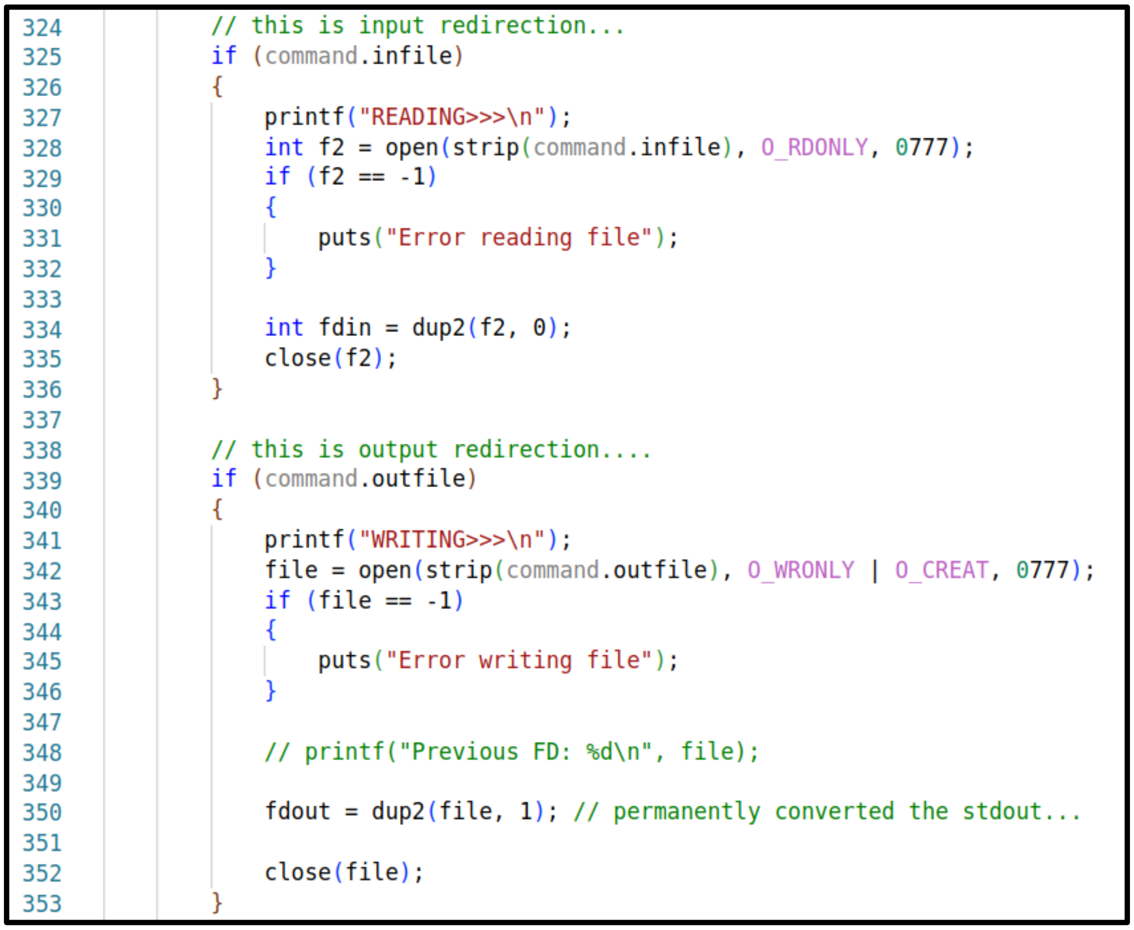


```
            // Middle Segments

            //  pipe 2
            if (pipe(fd[i]) == -1)
            {
                printf("Error creating pipe;");
            }

            // cmd 2
            int x = fork();
            pid[i] = x;
            if (pid[i] == 0)
            {
                // child process: read end
                dup2(fd[i - 1][0], STDIN_FILENO);
                close(fd[i - 1][1]);
                close(fd[i - 1][0]);

                // ignore the last command
                if (i != size - 1)
                {
                    // child process: write end
                    dup2(fd[i][1], STDOUT_FILENO);
                    close(fd[i][1]);
                    close(fd[i][0]);
                }

                if (execvp(cmd[0], cmd) == -1)
                {
                    printf("Execution failed..#C: %d", i);
                }
            }

            // erpor ar fd er kaj nai tai etake ekhanei off korte hobe
            close(fd[i - 1][1]);
            close(fd[i - 1][0]);
            waitpid(pid[i], NULL, 0);
        }
```

Figure 4: Pipelined Command Execution Code Snippet

4.4 Input Output Redirection

Following the previous sub-section, we have learned how the pipeline command has been implemented. Therefore, we may have noticed that while parsing the pipelined command we also check for redirection symbols whether they appear or not. The most crucial part of that feature is parsing accurate commands and I/O files from the given input. And this task is done by the **parse** method where we distinguish between pipelined command, redirected command, and I/O file. Then these arguments are passed through the **executePipelinedCommands** method. The reason we have used that particular method twice in two different features is due to their interrelation. While we execute pipeline commands, we deal with multiple commands and in that case, we also need to consider command redirection. Because this feature also deals with multiple commands where instead of multiple commands, here we use the I/O file where stdin-stdout is redirected. The execution process is partially mentioned in the following figure 5.

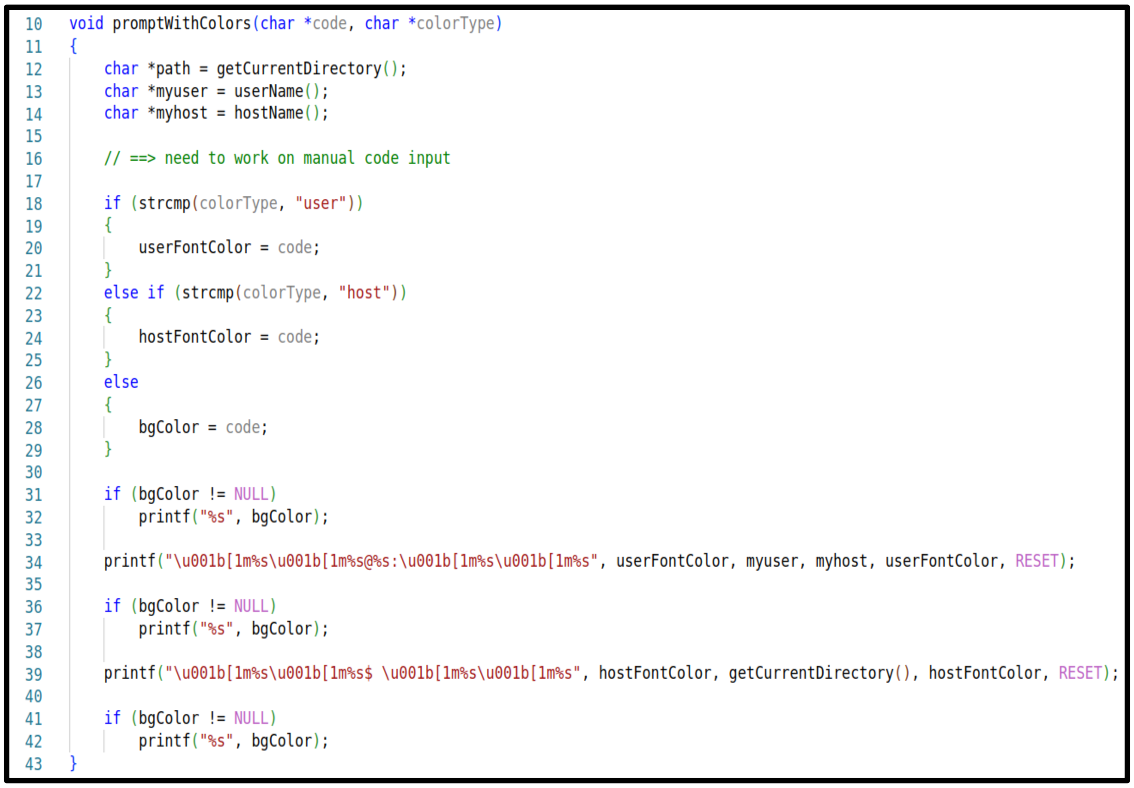


```c
// this is input redirection...
if (command.infile)
{
    printf("READING>>>\n");
    int f2 = open(strip(command.infile), O_RDONLY, 0777);
    if (f2 == -1)
    {
        puts("Error reading file");
    }

    int fdin = dup2(f2, 0);
    close(f2);
}

// this is output redirection....
if (command.outfile)
{
    printf("WRITING>>>\n");
    file = open(strip(command.outfile), O_WRONLY | O_CREAT, 0777);
    if (file == -1)
    {
        puts("Error writing file");
    }

    // printf("Previous FD: %d\n", file);

    fdout = dup2(file, 1); // permanently converted the stdout...

    close(file);
}
```

Figure 5: Stdin-Stdout Redirection Code Snippet

3.5 Terminal Color Customization

This feature's execution starts from the beginning of each session. We have some predefined macro for identifying the font and background color. Whenever we need to use those macros, we just need to pass them through the printing statements. A small demonstration of how this macro has been used is shown in figure 6. Whenever we invoke the color command, its handle is forwarded to the **cmd_Execute** method. Then we check the user's given arguments that whether he wants to change font color (foreground) or background color. Then to change the foreground color we invoke the **selectFGColor** method and alternatively for changing the background color the **selectBGColor** method is invoked. And to apply this change throughout the whole session (eventloop) the control is passed to the **eventLoopWithColors** method. This method applies changes to all other segments of our shell program.

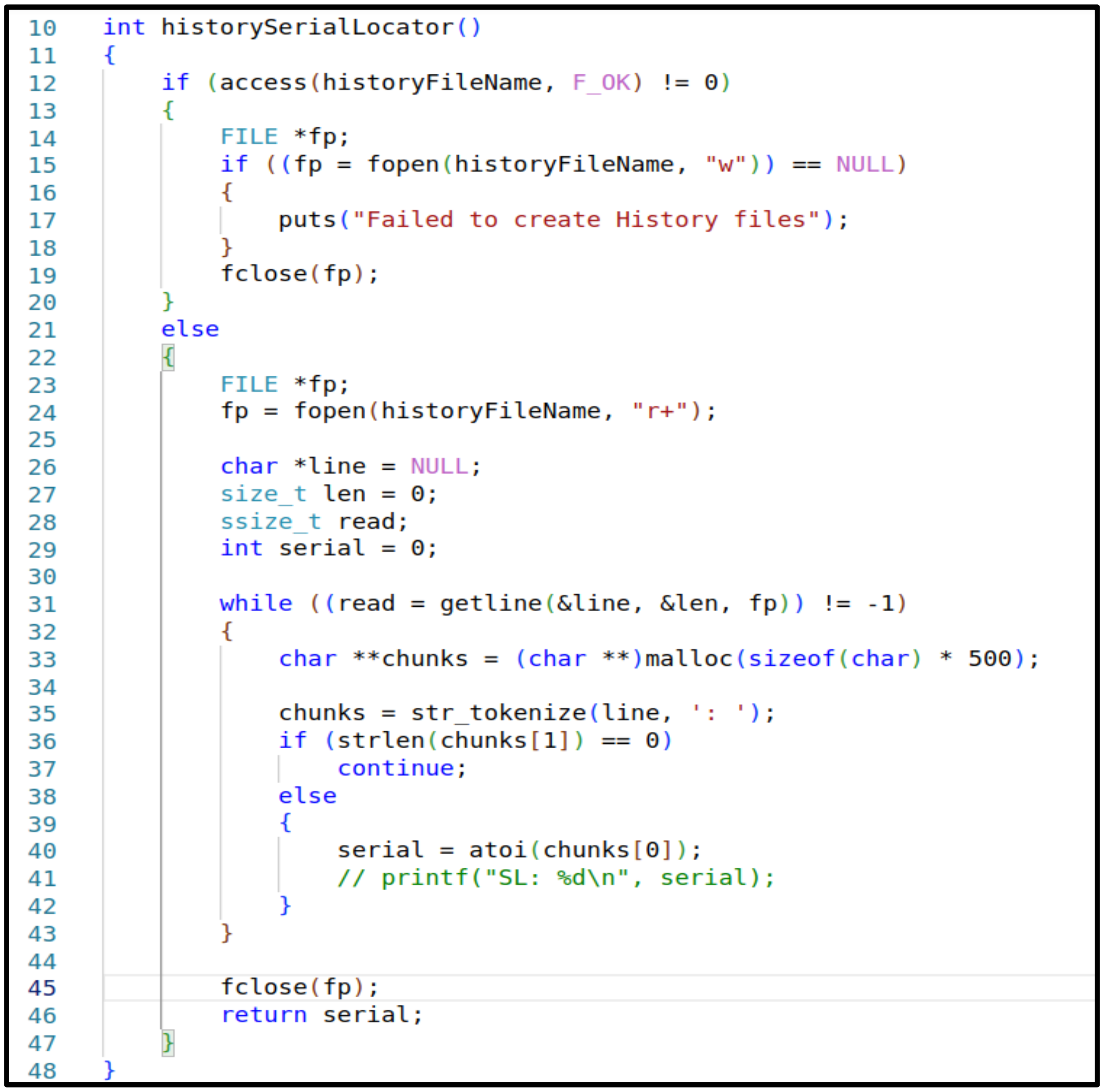


```c
void promptWithColors(char *code, char *colorType)
{
    char *path = getCurrentDirectory();
    char *myuser = userName();
    char *myhost = hostName();

    // ==> need to work on manual code input

    if (strcmp(colorType, "user"))
    {
        userFontColor = code;
    }
    else if (strcmp(colorType, "host"))
    {
        hostFontColor = code;
    }
    else
    {
        bgColor = code;
    }

    if (bgColor != NULL)
        printf("%s", bgColor);

    printf("\u001b[1m%s\u001b[1m%s@%s:\u001b[1m%s\u001b[1m%s", userFontColor, myuser, myhost, userFontColor, RESET);

    if (bgColor != NULL)
        printf("%s", bgColor);

    printf("\u001b[1m%s\u001b[1m%s$ \u001b[1m%s\u001b[1m%s", hostFontColor, getCurrentDirectory(), hostFontColor, RESET);

    if (bgColor != NULL)
        printf("%s", bgColor);
}
```

Figure 6: Colored Prompt Code Snippet

4.6 Shell History Management

Usually, history management is related to each session of the shell program. Here, we have implemented the same strategy where we start tracking each user's commands from the very beginning of our program. It starts with the method **historySerialLocator** that detects the serial of current history in the. slsh_history file so that the next command can be appended in the right order. You can find the code snippet related to that method in figure 7. Then after each command, we store it in a **history** structure where we temporarily save the history. After that, when the session ends, we invoke the **writeHistory** method to write the updated history fetched from the history structure. However, to show any particular command using history expansion character (!), we use **the showParticularHistory** method. It immediately extracts the command from history and put it into execution.

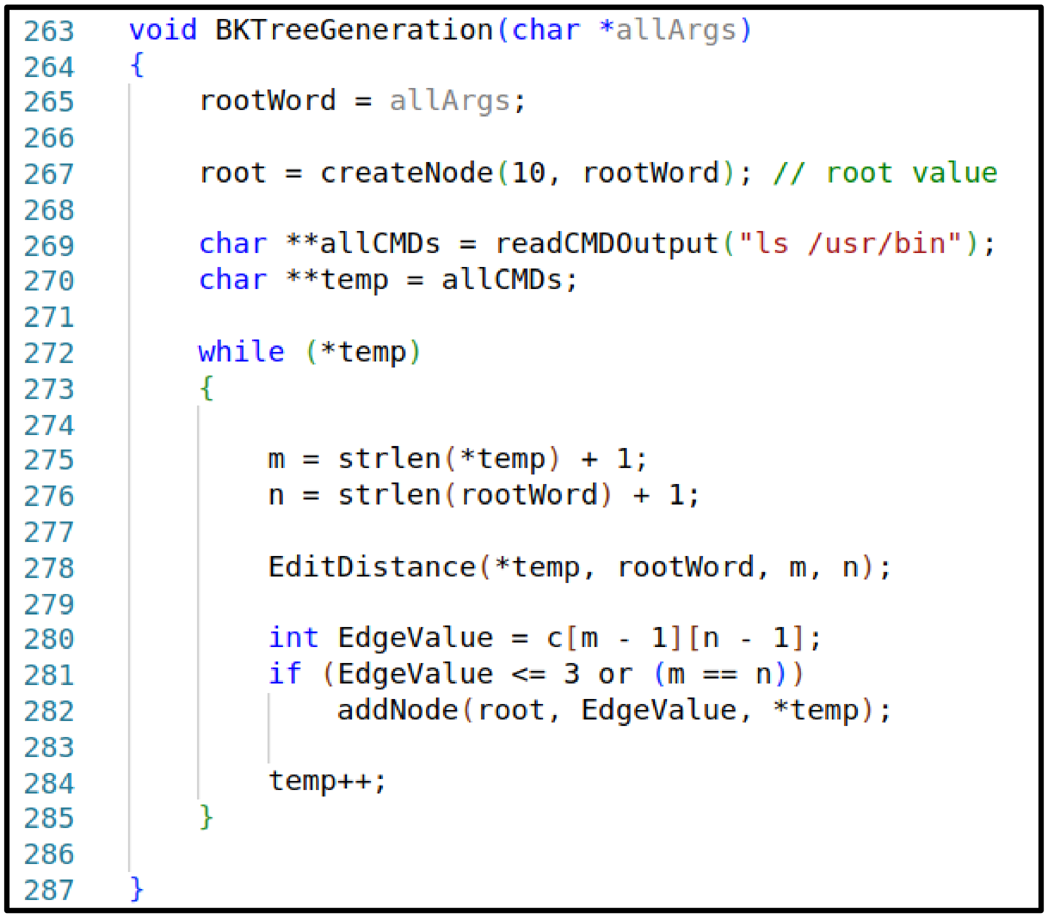


```c
int historySerialLocator()
{
    if (access(historyFileName, F_OK) != 0)
    {
        FILE *fp;
        if ((fp = fopen(historyFileName, "w")) == NULL)
        {
            puts("Failed to create History files");
        }
        fclose(fp);
    }
    else
    {
        FILE *fp;
        fp = fopen(historyFileName, "r+");

        char *line = NULL;
        size_t len = 0;
        ssize_t read;
        int serial = 0;

        while ((read = getline(&line, &len, fp)) != -1)
        {
            char **chunks = (char **)malloc(sizeof(char) * 500);

            chunks = str_tokenize(line, ': ');
            if (strlen(chunks[1]) == 0)
                continue;
            else
            {
                serial = atoi(chunks[0]);
                // printf("SL: %d\n", serial);
            }
        }

        fclose(fp);
        return serial;
    }
}
```

Figure 7: History Serial Collection Code Snippet

4.7 Command Correction

The last feature to focus on is command correction. We are already acknowledging from the description that it has two modes one is for basic suggestion mode and the other is autocompletion mode. Both of these modes use a common method, the **BKTreeGeneration** method that implements the *BK-Tree* data structure. This is the initial method where we built a tree data structure that contains the closest command related to the user given wrong commands. The implementation is demonstrated in the following figure no 8. In between this method, we invoke the **createNode** method where each node represents a new closest word. Also, to find those closest words, we search through the directories passed to the **readCMDOutput** function. Then we use the **EditDistance** method (*Levenshtein* distance algorithm) to calculate the level of closeness to the user given command and this value is put over the tree edges. For autocompletion mode, we add another method called **getAutoCompletedCommand.** This method matches the wrong user command to the closest and most used command of that user collected from the user history. The **frequencyCalculator** method handles the frequency of the closest and most used command. Thus, we perform command correction in our shell program.



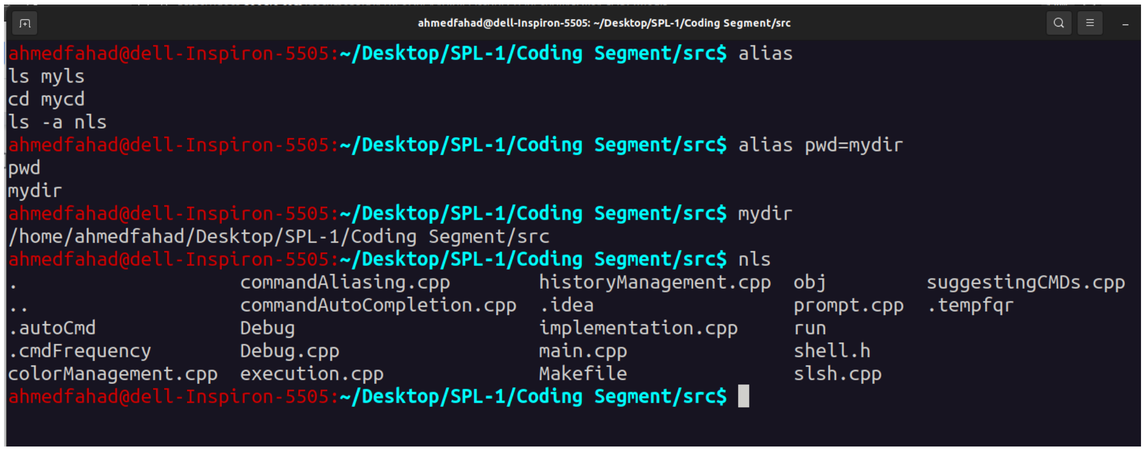
```c
void BKTreeGeneration(char *allArgs)
{
    rootWord = allArgs;

    root = createNode(10, rootWord); // root value

    char **allCMDs = readCMDOutput("ls /usr/bin");
    char **temp = allCMDs;

    while (*temp)
    {

        m = strlen(*temp) + 1;
        n = strlen(rootWord) + 1;

        EditDistance(*temp, rootWord, m, n);

        int EdgeValue = c[m - 1][n - 1];
        if (EdgeValue <= 3 or (m == n))
            addNode(root, EdgeValue, *temp);

        temp++;
    }

}
```

Figure 8: BK-Tree Generation Code Snippet

5. User Interface

In this section, we will analyze our project outcome through some input-output of our project. A suitable description will be added for better comprehension of what we have delineated below.

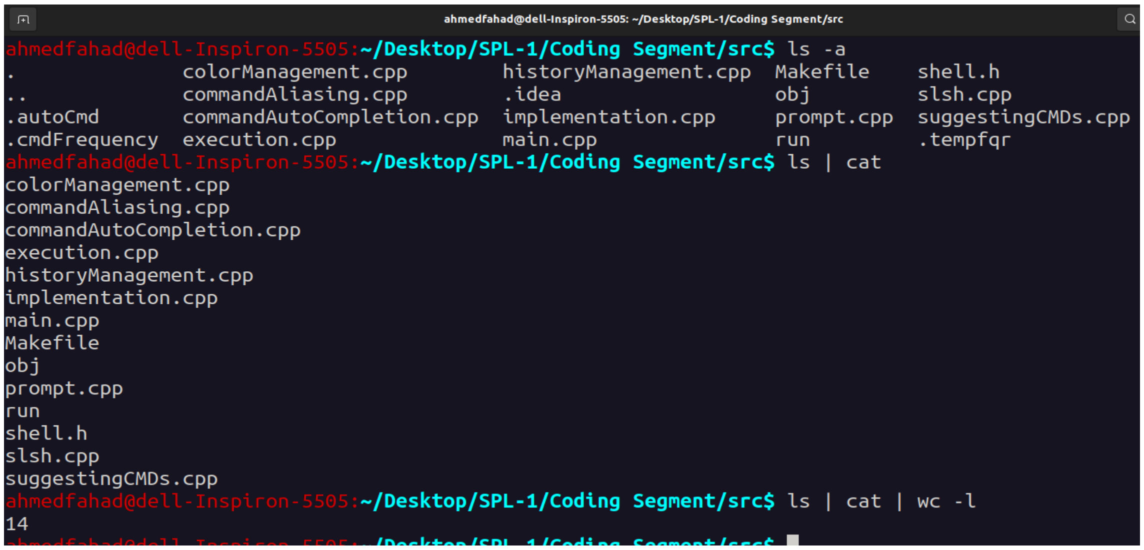
In the following figure, we can observe how alias commands are saved and stored through our shell. A noticeable point is that we can set multiple arguments-based commands as an alias in our shell. It is quite an important feature because sometimes we need to use long or complex commands that are hard to remember. In that case, we can replace those commands with some alias and make our work easier and more efficient.


ahmedfahad@dell-Inspiron-5505: ~/Desktop/SPL-1/Coding Segment/src

```
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ alias
ls myls
cd mycd
ls -a nls
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ alias pwd=mydir
pwd
mydir
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ mydir
/home/ahmedfahad/Desktop/SPL-1/Coding Segment/src
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ nls
.                    commandAliasing.cpp        historyManagement.cpp  obj         suggestingCMDs.cpp
..                   commandAutoCompletion.cpp  .idea                  prompt.cpp  .tempfqr
.autoCmd             Debug                      implementation.cpp     run
.cmdFrequency        Debug.cpp                  main.cpp               shell.h
colorManagement.cpp  execution.cpp              Makefile               slsh.cpp
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ 
```

Figure 9: Command Aliasing

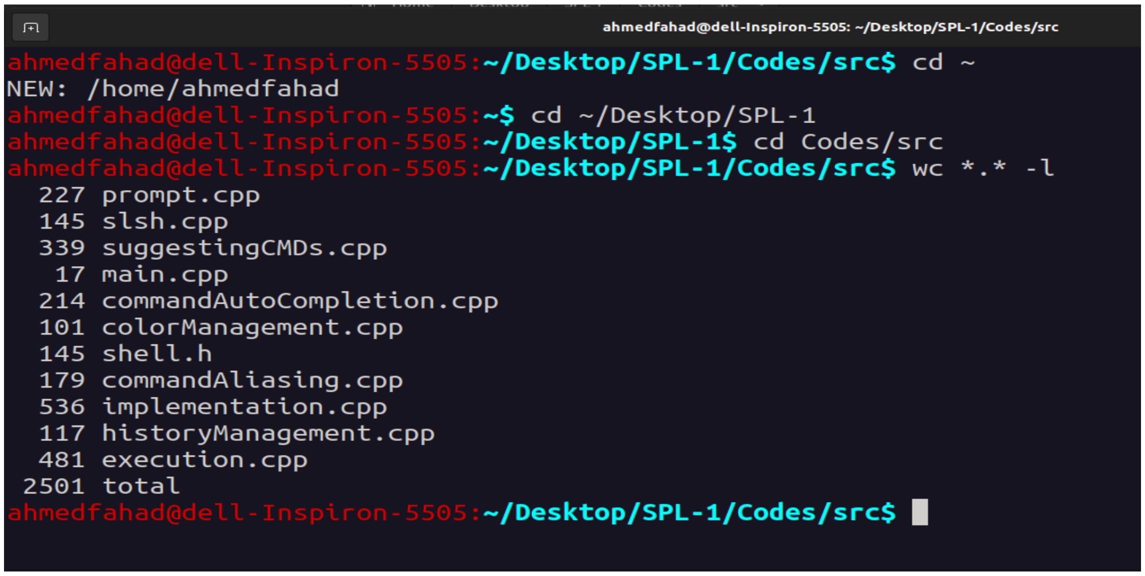
The execution way and the demonstration of pipelined commands are shown in the figure []. We need to concatenate multiple commands along with their arguments using the pipe '|' symbol. And the outcome will be as follows.


ahmedfahad@dell-Inspiron-5505: ~/Desktop/SPL-1/Coding Segment/src

```
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ ls -a
.               colorManagement.cpp        historyManagement.cpp  Makefile    shell.h
..              commandAliasing.cpp        .idea                  obj         slsh.cpp
.autoCmd        commandAutoCompletion.cpp  implementation.cpp     prompt.cpp  suggestingCMDs.cpp
.cmdFrequency   execution.cpp              main.cpp               run         .tempfqr
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ ls | cat
colorManagement.cpp
commandAliasing.cpp
commandAutoCompletion.cpp
execution.cpp
historyManagement.cpp
implementation.cpp
main.cpp
Makefile
obj
prompt.cpp
run
shell.h
slsh.cpp
suggestingCMDs.cpp
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ ls | cat | wc -l
14
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Coding Segment/src$ 
```

Figure 10: Pipeline Command Execution

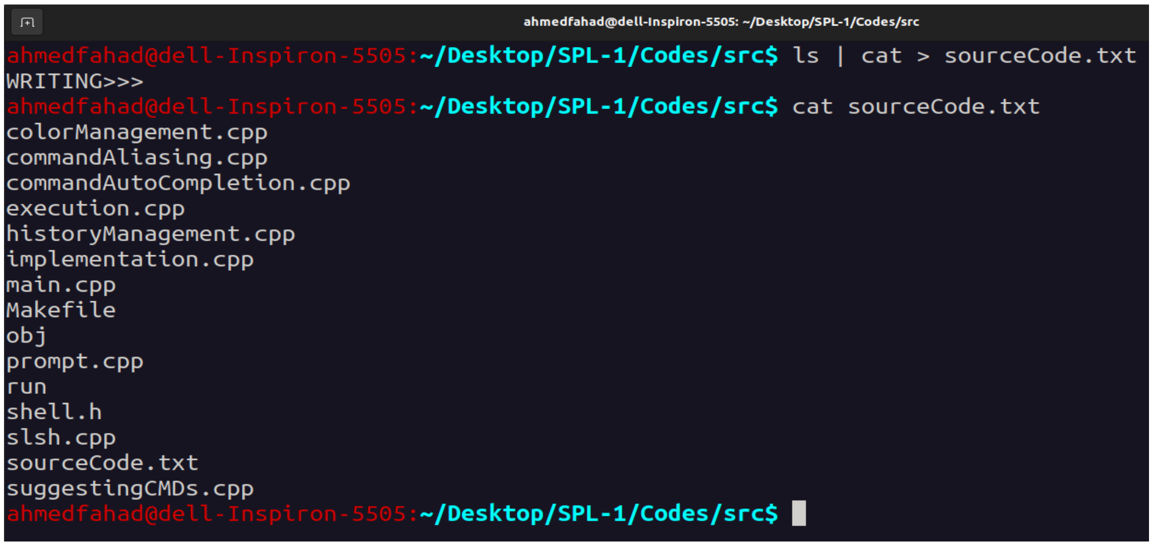
Tilde (~) expansion is a very time-saving and efficient way to switch any file or directories in the shell. Through the tilde sign, we denote the home directory (/home/username). We can observe how we shift to a specific directory from the home direction using that tilde expansion in the 2nd command. Another notable point here is File expansion (Wildcard). This wildcard expansion is widely used for expanding file names or directories as we can see in the following figure.



```
ahmedfahad@dell-Inspiron-5505: ~/Desktop/SPL-1/Codes/src
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ cd ~
NEW: /home/ahmedfahad
ahmedfahad@dell-Inspiron-5505:~$ cd ~/Desktop/SPL-1
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1$ cd Codes/src
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ wc *.* -l
  227 prompt.cpp
  145 slsh.cpp
  339 suggestingCMDs.cpp
   17 main.cpp
  214 commandAutoCompletion.cpp
  101 colorManagement.cpp
  145 shell.h
  179 commandAliasing.cpp
  536 implementation.cpp
  117 historyManagement.cpp
  481 execution.cpp
 2501 total
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$
```

Figure 11: Tilde and Wildcard Expansion

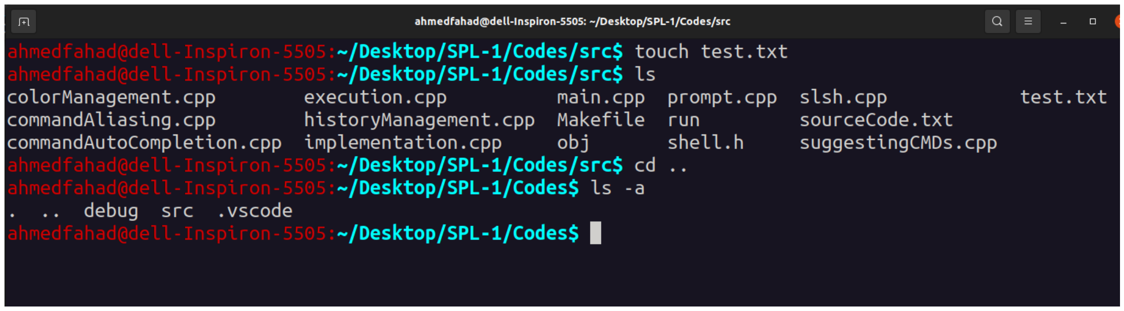
In the following snapshot, it is clear that we have redirected the output from the standard output (terminal) to another file. In this way, we can also replace our stdin with other input sources.



```
ahmedfahad@dell-Inspiron-5505: ~/Desktop/SPL-1/Codes/src
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ ls | cat > sourceCode.txt
WRITING>>>
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ cat sourceCode.txt
colorManagement.cpp
commandAliasing.cpp
commandAutoCompletion.cpp
execution.cpp
historyManagement.cpp
implementation.cpp
main.cpp
Makefile
obj
prompt.cpp
run
shell.h
slsh.cpp
sourceCode.txt
suggestingCMDs.cpp
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$
```

Figure 12: Input-Output Redirection

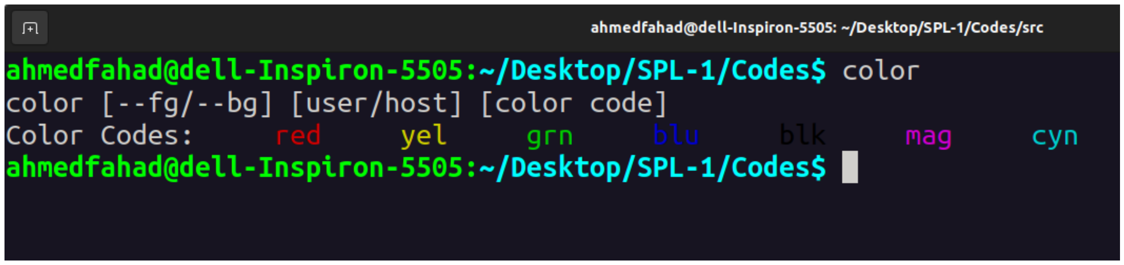
As we have seen in previous snapshots, we might have understood how the commands execute. Moreover, the details have been covered in the implementation section.



```
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ touch test.txt
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ ls
colorManagement.cpp        execution.cpp          main.cpp   prompt.cpp  slsh.cpp           test.txt
commandAliasing.cpp        historyManagement.cpp  Makefile   run         sourceCode.txt
commandAutoCompletion.cpp  implementation.cpp     obj        shell.h     suggestingCMDs.cpp
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ cd ..
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes$ ls -a
.  ..  debug  src  .vscode
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes$
```

Figure 13: Command Execution

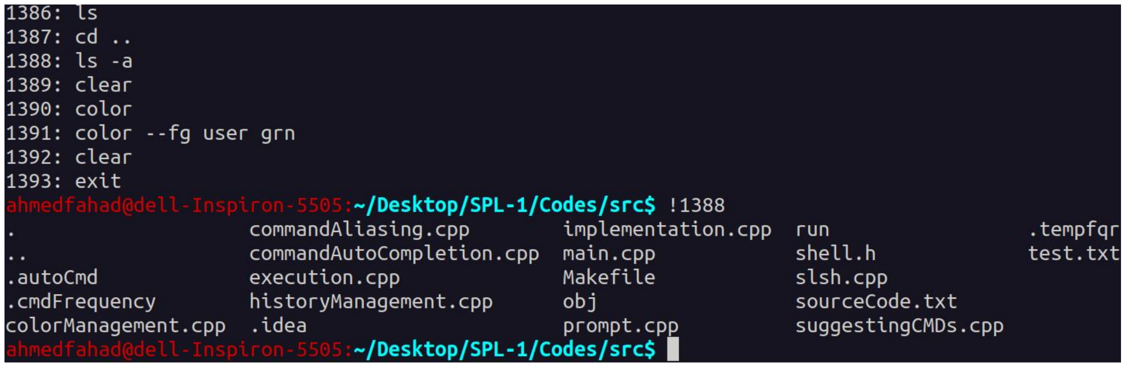
This figure represents how it looks after changing its color from red to green through our program. You can also learn about the command on how to change color shown in the following figure.



```
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes$ color
color [--fg/--bg] [user/host] [color code]
Color Codes:      red     yel     grn     blu     blk     mag     cyn
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes$
```

Figure 14: Colorized Terminal

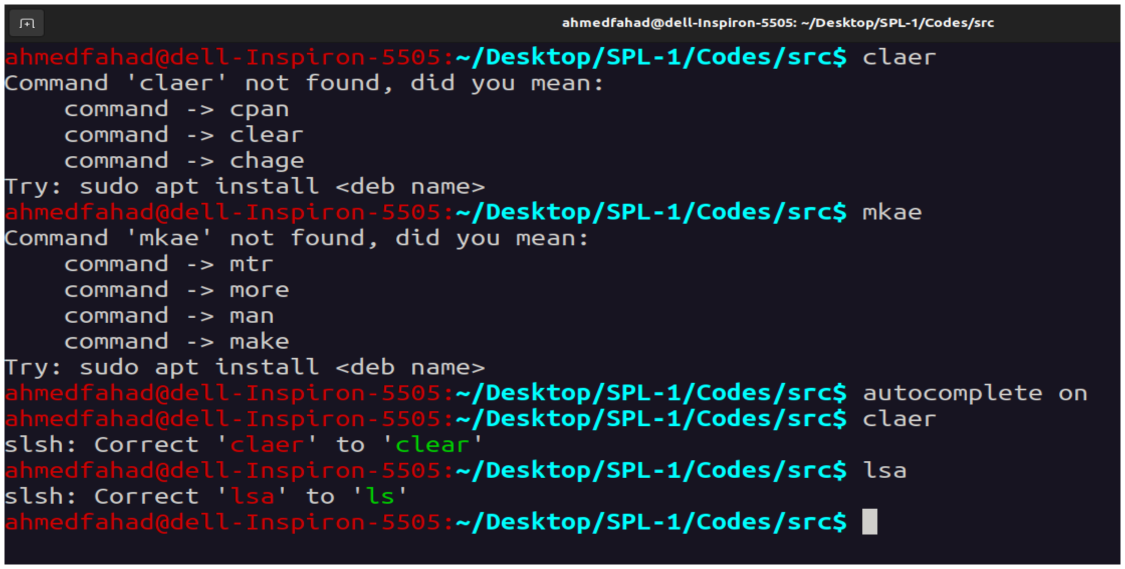
Here we can see, that we already wrote about 1393 commands. Also, we can access any specific command using the history expansion symbol (!).



```
1386: ls
1387: cd ..
1388: ls -a
1389: clear
1390: color
1391: color --fg user grn
1392: clear
1393: exit
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ !1388
.                    commandAliasing.cpp        implementation.cpp  run                 .tempfqr
..                   commandAutoCompletion.cpp  main.cpp            shell.h             test.txt
.autoCmd             execution.cpp              Makefile            slsh.cpp
.cmdFrequency        historyManagement.cpp      obj                 sourceCode.txt
colorManagement.cpp  .idea                      prompt.cpp          suggestingCMDs.cpp
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$
```

Figure 15: History Management

Initially, the command shows multiple suggestions against the wrong command. But when we turn the autocompletion mode on, it shows the closest and most used correct command for users.



```
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ claer
Command 'claer' not found, did you mean:
    command -> cpan
    command -> clear
    command -> chage
Try: sudo apt install <deb name>
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ mkae
Command 'mkae' not found, did you mean:
    command -> mtr
    command -> more
    command -> man
    command -> make
Try: sudo apt install <deb name>
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ autocomplete on
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ claer
slsh: Correct 'claer' to 'clear'
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$ lsa
slsh: Correct 'lsa' to 'ls'
ahmedfahad@dell-Inspiron-5505:~/Desktop/SPL-1/Codes/src$
```

Figure 16: Command Suggestion

6. Challenges Faced

While accomplishing the project, I had to face some challenges that were not too easy to resolve. Some of those challenges are listed below:

- **Understanding system process:** As my shell executes different shell commands, there I need to handle system processes. The system process is split into parent and child processes. Whenever we run any process, we execute the child process. Multiple instances of a single process can execute concurrently by different applications, each time a child process is invoked for execution.

- **Dynamic Memory Allocation:** This was a very challenging part of my project because handling memory with malloc and calloc is quite tricky. Any miscalculation may ruin the whole project execution. However, one observation that I have found is about segmentation fault. Segmentation fault was quite frequent and the reason behind is faulty array declaration. Sometimes our array goes out of bounds or we allocate already allocated spaces. In such cases, segmentation fault arose. Therefore, we should be careful while working with character pointer, array declaration, and string manipulation.

- **Parsing:** One of the core parts of my project was parsing users given to extract commands, arguments, and I/O files. I have to handle different cases of input to extract the I/O files. In Input-output redirection, I have handled different cases like:
  - command < infile > outfile
  - command > outfile < file
  - command | command > outfile

  Therefore, I have to manipulate the parsing process for different combinations of user-given commands.

- **Managing Large Codebase:** Working with a large codebase is a tedious task. It becomes very difficult to track changes on different files. That's why making the required header files and using makefile might lessen your hassle. Makefile makes it easy to compile multiple files even if you have headers or an extra library in your project.

- **Debugging Codes:** Frequently I got segmentation faults or dumped core errors but I wasn’t sure where it happening. Then this issue was instantly solved by debugging the cpp files. IDE like VScodes performs debugging quite efficiently.

7. Conclusion

We have discussed how to implement a shell throughout the whole writing. Now, it should be clear how the shell works and how all the commands are being processed. This project helped us to understand the concept of pipelining in command execution, redirection as input-output alteration, the wildcard in the file, or directory name expansion. Besides this, we have also learned how the system process is managed by using fork system calls, how to alias or refactor command statements, and how to display proper suggestions to wrong commands. Overall, from a birds-eye, we covered the most common aspect of shell specifically for a bash.

But that is not the end of our journey. We can still implement shell scripting, shell functions, and conditional in further extension. It will make our shell more relatable to the actual shell environment.

Reference

[1]https://www.gnu.org/software/bash/manual/html_node/Basic-Shell-Features.html, GNU Operating System, 29/05/2022

[2]https://www.computerhope.com/jargon/f/file-descriptor.htm, Computer Hope, 29/05/2022

[3]https://linuxhint.com/fork_linux_system_call_c/, Linux Hint, 29/05/2022

[4]https://developpaper.com/linux-pipeline-communication-c-language-programming-example/, Develop Paper, 29/05/2022

[5]https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797, GitHub, 29/05/2022

[6]https://medium.com/future-vision/bk-trees-unexplored-data-structure-ec234f39052d, Medium, 29/05/202

[7]https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0, Medium, 29/05/2022

[8]https://www.scaler.com/topics/data-structures/kmp-algorithm/, Scaler, 29/05/2022

[9]https://www.programiz.com/c-programming/c-dynamic-memory-allocation, Programiz, 29/05/2022

[10]https://www.tecmint.com/linux-process-management/, Tech Mint, 29/05/2022