



Higher institute of computer science - Arianna



Documentation

Présenté par
Zareb Fares
Guesmi Mohamed Aziz

Shell command line interpreter

Sous la direction du

DR. Najar Yousra

Professor at higher institute of computer science

I. Overview

In this assignment, we will implement a command line interpreter or, as it is more commonly known, a shell. The shell should operate in two modes both interactive and batch mode. The interactive one: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished, the other mode should take a file as argument input and perform same way for each line of the file

II. Objectives

- ✓ Familiarize with the Linux programming environment.
- ✓ Learn how to create, execute, kill a process...
- ✓ Gain exposure to the necessary functionality in shells.

III. Data Structure:

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So, in our program we used different types mainly the linear data structure such as:

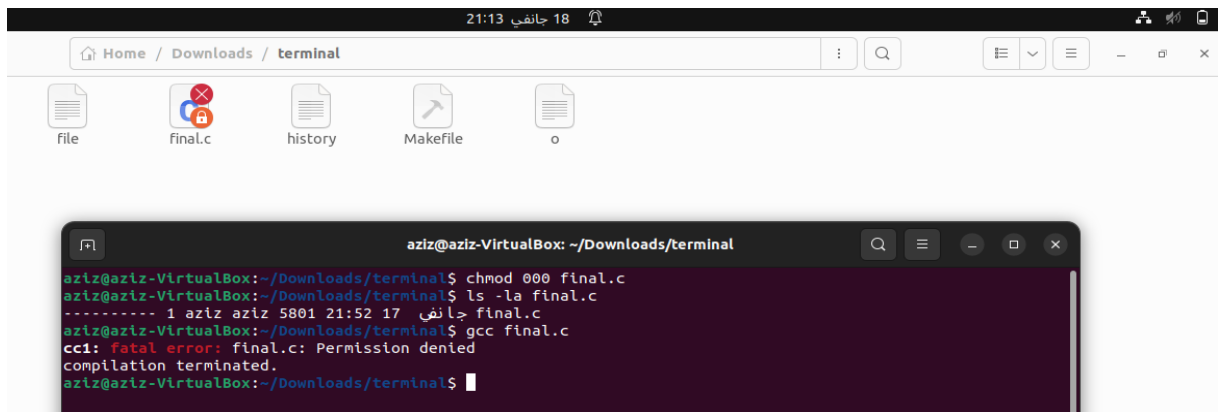
- Files: the idea behind using files was that this offers variety advantages as it keeps track of the information created after the program has been run also it offers a large amount of input that may be required for our program. Files allows to store data without having to worry about storing everything simultaneously in a program.
- Pointers: Pointers save memory space. Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location. Memory is accessed efficiently with the pointers. The pointer assigns and releases the memory as well.
it provides an alternate way to access array elements, reduces the storage space and complexity of the program, reduces the execution of the program etc...

IV. Software Installation :

1. Environment Set Up:

Every UNIX file has an owner. Initially, the owner is the user (u) who created the file but “root” can also assign it to another user. Only the owner of the file and the super user (root) can change the permissions.

To explain this, we took all the permission manually to explain certain ways to guarantee the user the possibility to manage the files.

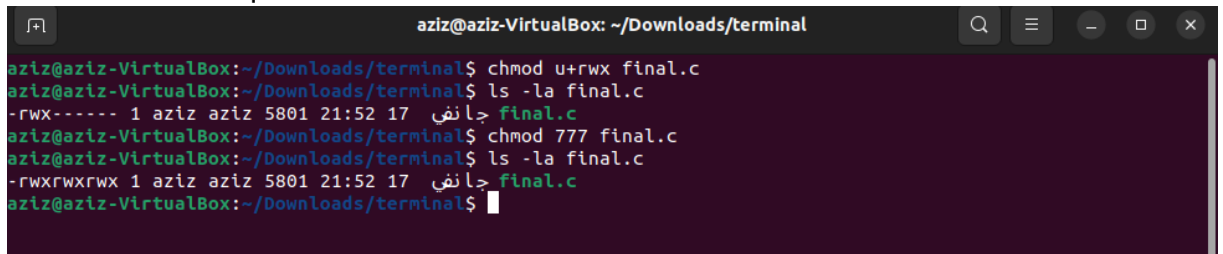


The image shows a file manager window with a sidebar containing icons for 'file', 'final.c', 'history', 'MakeFile', and 'o'. The main pane shows a terminal window titled 'aziz@aziz-VirtualBox: ~/Downloads/terminal'. The terminal output is as follows:

```
aziz@aziz-VirtualBox:~/Downloads/terminal$ chmod 000 final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$ ls -la final.c
----- 1 aziz aziz 5801 21:52 17 جانفي final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$ gcc final.c
cc1: fatal error: final.c: Permission denied
compilation terminated.
aziz@aziz-VirtualBox:~/Downloads/terminal$
```

In order to give the user, the possibility to manage the files we need to type either of the commands shown below:

Chmod u+rwX | 777 file_name



The image shows a terminal window titled 'aziz@aziz-VirtualBox: ~/Downloads/terminal'. The terminal output is as follows:

```
aziz@aziz-VirtualBox:~/Downloads/terminal$ chmod u+rwX final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$ ls -la final.c
-rwx----- 1 aziz aziz 5801 21:52 17 جانفي final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$ chmod 777 final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$ ls -la final.c
-rwxrwxrwx 1 aziz aziz 5801 21:52 17 جانفي final.c
aziz@aziz-VirtualBox:~/Downloads/terminal$
```

Or simply by using the appropriate commands in the make file:

Compile, build or execute as root or by setting the permissions:

make + command {name_file(optional)}

```
14 adcompile:
15     sudo gcc final.c -c
16
17 adbuild:
18     sudo gcc final.o -o final
19
20 adexec:
21     sudo ./final $(file)
22
23 adclean:
24     sudo rm -r final
25
26 setUserP:
27     chmod u+rwX $(file)
28
29 setAllP:
30     chmod 777 $(file) |
```

As an easy way to kind of look at the overall situation: Linux filesystem permissions cease to exist outside of the filesystem that created it, as it to say if you push you're file to git, git basically drops all the Linux filesystem permissions with the exception of the execute bit. Outside of the exception for the execute bit, files created by git clone have their permissions otherwise set by your [umask](#).

Regarding where permissions are actually stored, that information is kept in the filesystem as file metadata. As a silly analogy, files are like letters with an envelope. Editors and whatnot look at the letter whereas permissions, filename, etc are on the envelope which is stored as filesystem metadata. If you want to get into the details, some of this metadata is stored in the `inode` and some is stored in the containing directory entry. When you transfer a file, only the data (the letter) is transferred and the metadata (the envelope) is discarded unless some other mechanism is in place to save the metadata (e.g. tar file, git as previously described, etc). In order to have all the permissions to manage file cloned you need to run the command below that set a default permission to all the files created:

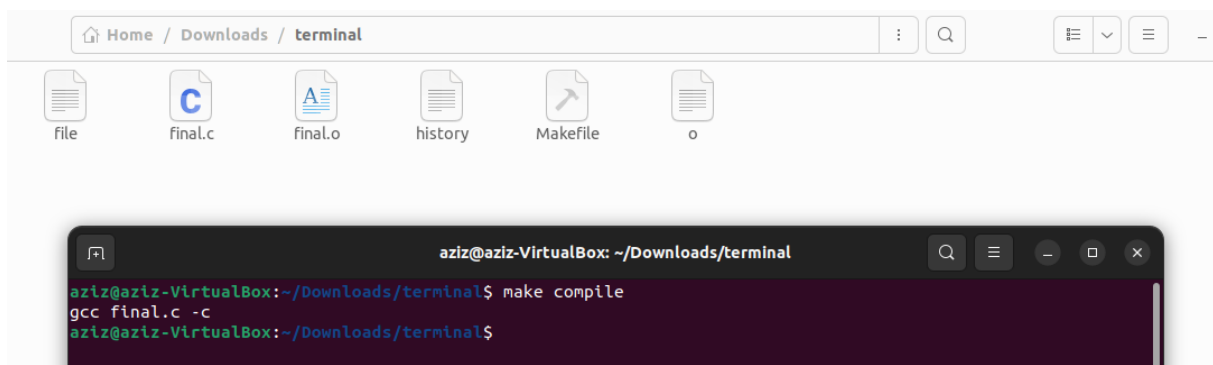
```
$ umask 000
```

2. Compilation

Before we start with compiling our `file.c` we need to install the make package in our machine:

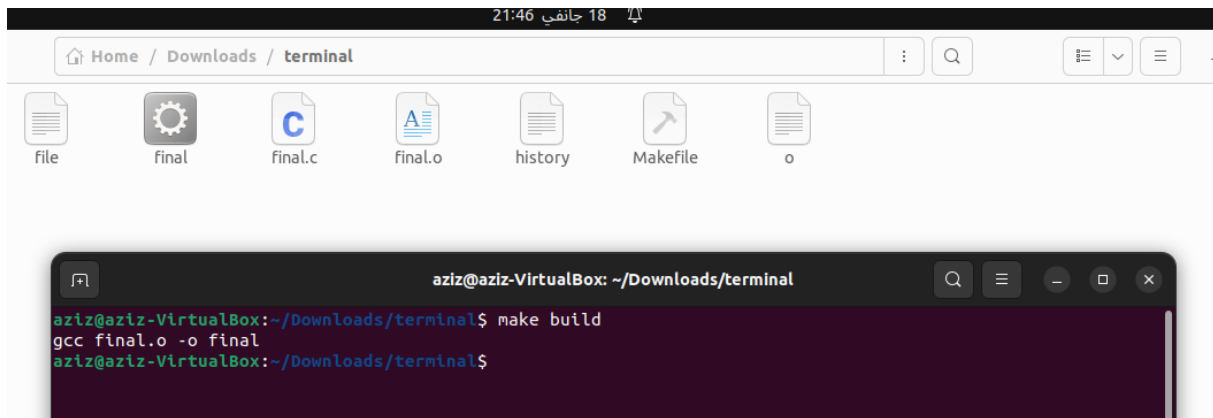
```
$sudo apt install make
```

Once it done we can start compiling our code using the make file by typing `make compile` to generate `name_file.o`



3. Generating executable

Type in the command line `make build` to generate the executable file.



4. Execution

Finally, we get to the point where we can run our code using one of the following modes:

- BATCH MODE: Read a file lines and execute its commands one by one.
- INTERACTIVE MODE: Read the user input and execute it.

V. Specifications

In this part we will handle our coding part explaining some functionalities and the purpose behind implementing them:

```
void main( int argc, char *argv[] ) {  
    switch(argc){  
        case 1 : InteractiveMode();break;  
        case 2 : BatchMode(argv);break;  
        default: printf(RED "Vous avez ajouter plusque 2 arguments dans l'invite de commande \n"  
            COLOR_RESET); break;  
    }  
}
```

We begin with our main procedure that's basically a composition of two procedures (BatchMode, InteractiveMode) under a switch case, that is to say the program will execute one of those procedures depending on how many arguments passed to the `exec` command.

1. InteractiveMode specifications :

```
void InteractiveMode(){
    char lineCommande[500],cwd[1024];

    printf(YELLOW  "Entrer la commande quit ou eof pour quitter:\n" COLOR_RESET);
    while(1){

        if (getcwd(cwd, sizeof(cwd)) != NULL)
        {
            printf(GREEN "%s " COLOR_RESET, strcat(cwd,"%"));
        }
        else
            perror("getcwd failed\n");

        fgets(lineCommande, 500, stdin);
        UpdateHistory(lineCommande);
        int Quitter = ExecuteCMD(lineCommande);
        if(Quitter) exit(0);
    }
}
```

In this procedure we declared two local variables lineCommand for the command typed by the user and cwd that takes the current directory followed % to print it for the user.

Once the user enters his command, we flush it in the lineCommand with a maximum length of 500, update the history by calling the updateHistory procedure provided by the lineCommand, call the ExecuteCmd and set simultaneously the value of quitter variable to 1 or 0. If lineCommand equals to quitter set the variable to 1 which true otherwise it takes 0.

Once quitter equals 1 the program is terminated.

ExecuteCmd:

This procedure is made to handle our command and the way to execute it, it also provides some error handling such as the one starting a command with one of the operators which should cause an error otherwise it passes to next step, that consists of detecting one of the operators then calls for tokenize_Buffer followed by the appropriate operator procedure.

If none of the operators exist in the input command our procedure offers various exception : change current directory if the cd command entered, quit the command interpreter if the quitter command entered or show history for the history key otherwise it calls for executeBasic to execute a simple command.

tokenize_Buffer

```
void tokenize_buffer(char** param,int *NbOfCommande,char *lineCommande,const char *c){
    char *token;
    token=strtok(lineCommande,c);
    int pc=-1;
    while(token){
        param[++pc]=malloc(sizeof(token)+1);
        strcpy(param[pc],token);
        removeWhiteSpace(param[pc]);
        token=strtok(NULL,c);
    }
    param[++pc]=NULL;
    *NbOfCommande=pc;
}
```

This procedure takes as input the lineCommand, a c character which is the operator then it loops the array and tokenize it command by command and place each one in different case of another array. Once the process is done the procedure set both param and NbOfCommand with the array obtained and its length. In this part we can notice the importance of the usage of the pointers

PV:

```
void PV(char** lineCommande,int NbOfCommande){
    int pc;
    char *argv[100];

    if(strlen(lineCommande[0])==0 )
        printf("Commande invalide\n");
    else {
        for(int i=0;i<NbOfCommande;i++){
            tokenize_buffer(argv,&pc,lineCommande[i]," ");
            if(fork()==0){
                execvp(argv[0],argv);
                printf(RED "Cette commande ( %s ) n'existe pas ou ne peut pas être exécutée\n");
                exit(1);
            }else{
                wait(NULL);
            }
        }
    }
}
```

This procedure simulates the ; in linux, firstly it raises an error if there are no commands passed with ; in the beginning of the lineCommand, otherwise it calls the tokenize_buffer with space as separator to distinct the arguments passed to the commands and execute each command independently from the other.

OR:

```
void OR(char** lineCommande,int NbOfCommande){
    int pc;
    char *argv[100];
    if(strlen((lineCommande[0]))==0 ) printf("Commande invalide\n");
    else {
        for(int i=0;i<NbOfCommande;i++){
            tokenize_buffer(argv,&pc,lineCommande[i]," ");
            if(fork()!=0){
                wait(NULL);
                break;
            }else{
                if (execvp(argv[0],argv) != -1 ){
                    break;
                }
            }
        }
    }
}
```

This procedure simulates the || in linux, firstly it raises an error if theres no commands passed with || otherwise it calls the tokenize_buffer with space as separator to distingue the arguments passed to the commands and execute the i command only if the i-1 fails.

AND:

```
void AND(char** lineCommande,int NbOfCommande){
    int i,pc,test=0;
    char *argv[100];
    if(strlen((lineCommande[0]))==0 ) printf("Commande invalide\n");
    else {
        for(i=0;i<NbOfCommande;i++){
            if (strlen(lineCommande[i])==0)
                printf (RED "Commande invalide");
            else {
                tokenize_buffer(argv,&pc,lineCommande[i]," ");
                if(fork()!=0){
                    wait(NULL);
                }
                else{
                    if ( execvp(argv[0],argv) < 0)
                        perror(RED "Commande invalide" COLOR_RESET );
                    break;
                }
                exit(1);
            }
        }
    }
}
```


This procedure simulates the && in linux, firstly it raise an error if theres no commands passed with && otherwise it calls the tokenize_buffer with space as separator to distingue the arguments passed to the commands and execute the i command only if the i-1 success.

PIPE:

This function simulates the | in linux, firstly it raise an error if theres no commands passed to | otherwise the function takes in two arguments: a lineCommande which is an array of commands and NbOfCommande which is the length of the lineCommand. The PIPE creates two arrays, fd and argv, and initializes a variable called pc. It then loops through the commands in lineCommande, tokenizing each command by space and save the result in arg, the function creates a pipe using the pipe() system call and forks a child process. The child process uses the dup2() function to redirect the input and output of the pipeline as appropriate and then executes the command using execvp(). The parent process waits for the child to finish before proceeding to the next iteration. Finally, it closes the file descriptors used in the pipeline.

Redirection:

```
void executeRedirect(char** lineCommande,int NbOfCommande){
    int pc,fd;
    char *argv[100];
    removeWhiteSpace(lineCommande[1]);
    tokenize_buffer(argv,&pc,lineCommande[0]," ");
    if(strlen((lineCommande[0]))==0 )
        printf("Commande invalide\n");
    else {
        if(fork()==0){
            FILE *f = fopen(lineCommande[1],"a+");
            fd=open(lineCommande[1],O_WRONLY);
            dup2(fd,1);
            execvp(argv[0],argv);
            perror(RED "Commande invalide" COLOR_RESET);
            exit(1);
        }else{
            wait(NULL);
        }
    }
}
```

This procedure simulates the > in linux, firstly it raise an error if theres no commands passed with > otherwise it calls the tokenize_buffer with

space as separator to distinguish the arguments passed to the commands and write the return command in file if it success.

ExecuteBasic :

```
void executeBasic(char** argv){  
    if(fork()!=0){  
        wait(NULL);  
    }  
    else{  
        if (execvp(argv[0],argv) < 0 )  
        {  
            perror(RED "Cette commande n'existe pas ou ne peut pas être exécutée" COLOR_RESET );  
            exit(1);  
        }  
    }  
}
```

The function takes in one argument, argv, which is an array of arguments for the command. The function uses the fork() system call to create a child process. The parent process waits for the child process to finish using the wait() function. In the child process, the function uses the execvp() function to execute the command represented by the argv array.

History & updateHistory:

```
void History(){  
    char lineCommande[500];  
    int i = 1;  
    FILE *file = fopen("history","r");  
    if (file != NULL){  
        while (fgets(lineCommande,sizeof(lineCommande), file))  
        {  
            printf("  %d %s",i++,lineCommande);  
        }  
        fclose(file);  
    }  
}
```

This code defines a function called "History" that reads the contents of a file called "history" and prints each line to the console, preceded by a line number. The function uses the standard C library function "fopen" to open the file in read mode, and "fgets" to read each line of the file. The function also uses the "printf" function to print the line number and the

contents of the line. If the file cannot be opened, the function does nothing. Finally, the function uses the "fclose" function to close the file.

```
void UpdateHistory(char lineCommande[500]){  
  
    FILE *file = fopen("history","a+");  
    fputs(lineCommande, file);  
    fclose(file);  
  
}
```

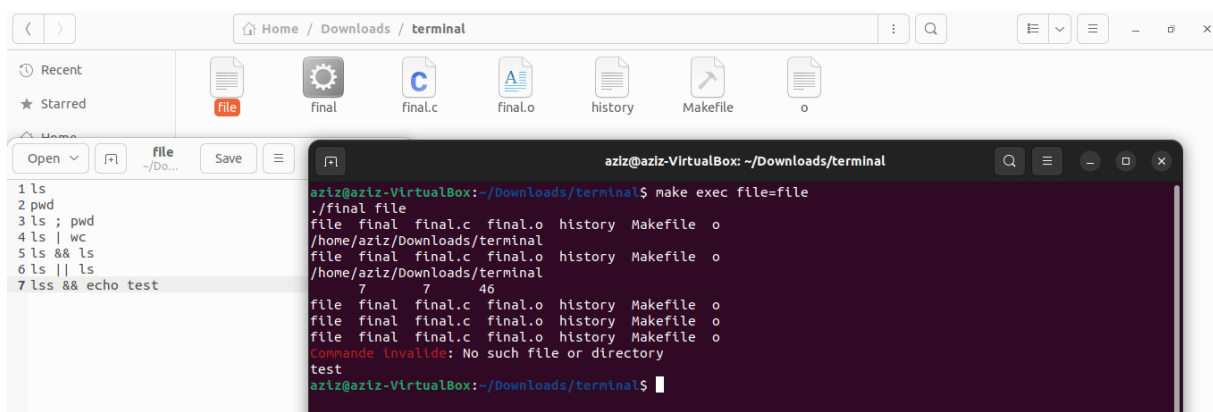
This code defines a function called "UpdateHistory" that takes a string called "lineCommande" as input. The function uses the "fopen" function to open a file called "history" in append mode, which means that new data will be written to the end of the file. The "fputs" function is then used to write the contents of "lineCommande" to the file. Finally, the function uses the "fclose" function to close the file. This function will add the new command string to the end of the history file each time it is called.

VI. Software usage:

In this part we are going to test our executable file by trying some commands in both modes such as simple commands, piped, redirection, composition of commands etc...

1. Batch mode :

As it mentioned the batch mode made to read a file of commands as argument and execute it line by line.

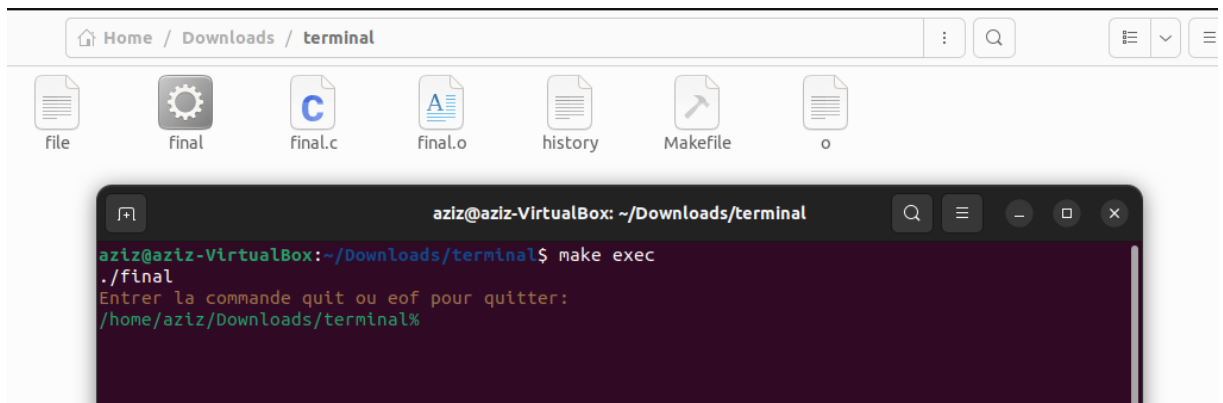


```
aziz@aziz-VirtualBox: ~/Downloads/terminal  
aziz@aziz-VirtualBox:~/Downloads/terminal$ make exec file=file  
./final file  
file final final.c final.o history Makefile o  
/home/aziz/Downloads/terminal  
file final final.c final.o history Makefile o  
/home/aziz/Downloads/terminal  
7 7 46  
file final final.c final.o history Makefile o  
file final final.c final.o history Makefile o  
file final final.c final.o history Makefile o  
Commande invalide: No such file or directory  
test  
aziz@aziz-VirtualBox:~/Downloads/terminal$
```

2. Interactive Mode :

The interactive mode is a shell that provides the user a command interpreter which contains our directory followed by an input where

the user can perform a different type of command execution, that is to say that our interpreter is a simulation of linux command line.

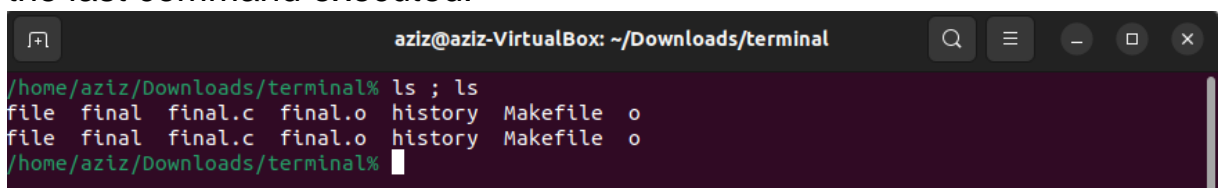


Functionalities under the interactive mode

Both interactive and batch mode perform the same sort of work with a different input, so in this we are going to execute the different functionalities maintained in our code.

a. Composition;

It is used to execute multiple commands in one go. Several commands can be chained together using this operator. The execution of the command succeeding this operator will always execute after the command preceding it has executed irrespective of the exit status of the preceding command. The commands always execute sequentially. Commands separated by a semicolon operator are executed sequentially, the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.



b. Composition &&

The second command will only execute if the first command has executed successfully i.e, its exit status is zero. This operator can be used to check if the first command has been successfully executed. This is one of the most used commands in the command line.

```
aziz@aziz-VirtualBox: ~/Downloads/terminal
/home/aziz/Downloads/terminal% ls && pwd
file final final.c final.o history Makefile o
/home/aziz/Downloads/terminal
/home/aziz/Downloads/terminal% wrong && ls
Commande invalide: No such file or directory
/home/aziz/Downloads/terminal%
```

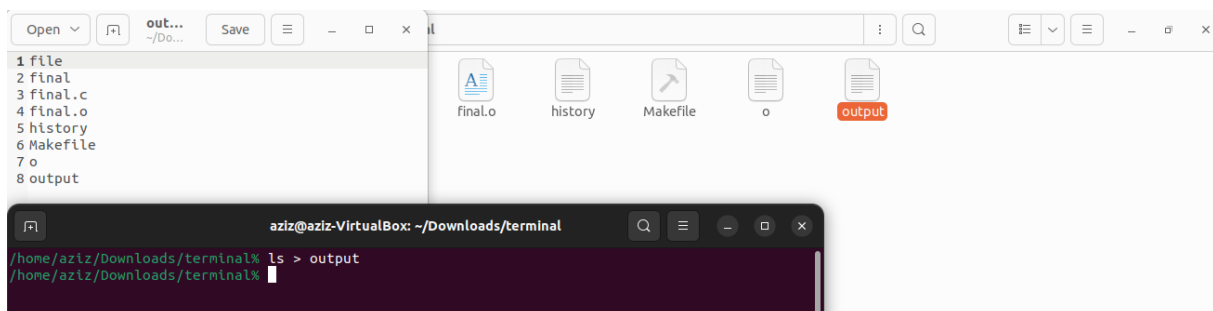
c. Composition ||

The second command is executed only when the first command fails (returns a non-zero exit status).

```
aziz@aziz-VirtualBox: ~/Downloads/terminal
/home/aziz/Downloads/terminal% ls || pwd
file final final.c final.o history Makefile o
/home/aziz/Downloads/terminal% wrong || ls
file final final.c final.o history Makefile o
/home/aziz/Downloads/terminal%
```

d. Output redirection >

We can overwrite the standard output using the '>' symbol.



```
aziz@aziz-VirtualBox: ~/Downloads/terminal
/home/aziz/Downloads/terminal% ls > output
/home/aziz/Downloads/terminal%
```

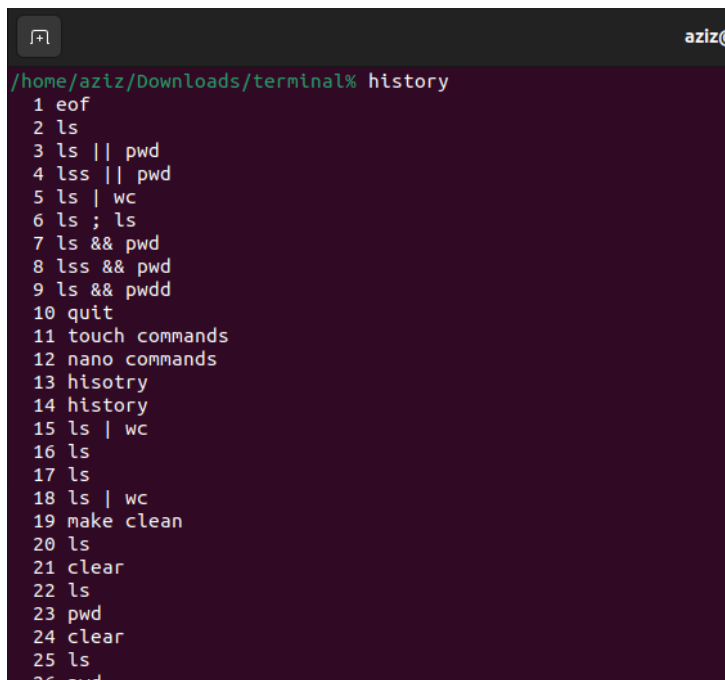
e. Pipe |

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '|'.

```
aziz@aziz-VirtualBox: ~/Downloads/terminal
/home/aziz/Downloads/terminal% ls | wc
      8      8     53
/home/aziz/Downloads/terminal%
```

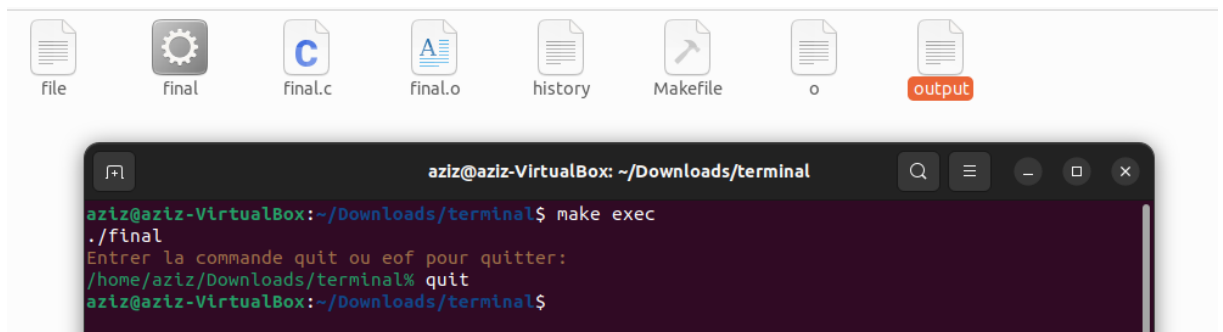
f. History & Quit

Save all the commands executed

A terminal window with a dark purple background. The prompt is `aziz@`. The user has entered the command `history`. The terminal displays a list of 26 commands, each preceded by a line number. The commands are: 1 eof, 2 ls, 3 ls || pwd, 4 lss || pwd, 5 ls | wc, 6 ls ; ls, 7 ls && pwd, 8 lss && pwd, 9 ls && pwdd, 10 quit, 11 touch commands, 12 nano commands, 13 hisotry, 14 history, 15 ls | wc, 16 ls, 17 ls, 18 ls | wc, 19 make clean, 20 ls, 21 clear, 22 ls, 23 pwd, 24 clear, 25 ls, 26 pwd.

```
/home/aziz/Downloads/terminal% history
1 eof
2 ls
3 ls || pwd
4 lss || pwd
5 ls | wc
6 ls ; ls
7 ls && pwd
8 lss && pwd
9 ls && pwdd
10 quit
11 touch commands
12 nano commands
13 hisotry
14 history
15 ls | wc
16 ls
17 ls
18 ls | wc
19 make clean
20 ls
21 clear
22 ls
23 pwd
24 clear
25 ls
26 pwd
```

Type the command Quit to exit from the interactive mode



VII. Conclusion

This is a UNIX command line interpreter based on the simple shell. It reads user input from the command line interactive mode or from file which is batch mode, interprets it, and executes commands.