SYSTEM PROJECT
Creating a Shell Command Interpreter

1 ING 01 :

Ines BOUKHRIS

Nour EL KAHLA

2022/2023

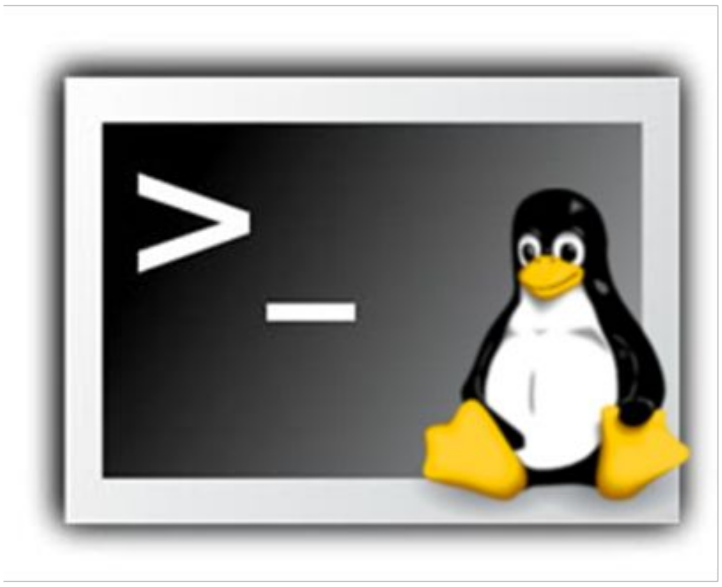
Introduction

Shell Linux :

A Linux shell is a command-line interface that allows users to interact with the Linux operating system. It provides a way for users to execute commands, navigate the file system, and perform various other tasks.Also, shell scripts are basically a series of Linux commands stored in a file that can be executed as a single command.There are several different Linux shells available, some popular examples include:

- Bash (Bourne-Again Shell) which is the default shell for most Linux distributions.
- Zsh (Z shell) which provides many advanced features such as command line completion and history.
- Fish (Friendly Interactive Shell) which is designed to be easy to use and has a built-in tutorial.

Each shell has its own syntax and features, but they all provide the same basic functionality. The choice of which shell to use is mostly a matter of personal preference.

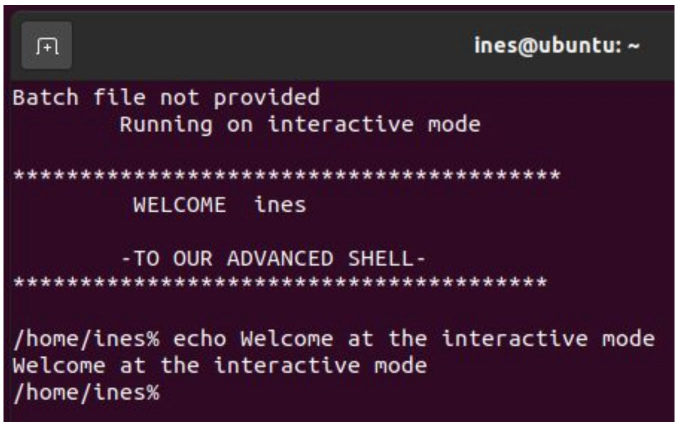


Introduction

Architecture :

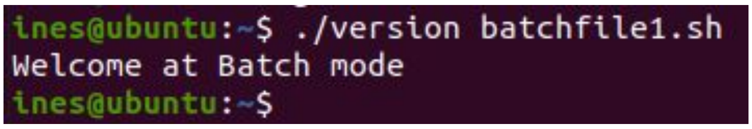
In Linux, a shell can be used in two different modes: interactive mode and batch mode.

Interactive mode: In interactive mode, the shell runs in a terminal window and waits for the user to enter a command. The user can enter commands one at a time, see the results, and then enter more commands.



```
ines@ubuntu: ~
Batch file not provided
        Running on interactive mode

*****************************************
         WELCOME  ines

        -TO OUR ADVANCED SHELL-
****************************************

/home/ines% echo Welcome at the interactive mode
Welcome at the interactive mode
/home/ines%
```

Batch mode: In batch mode, the shell reads commands from a file, known as a script, and executes them in order. The script contains a series of commands that are executed automatically without any user interaction. This mode is useful for automating repetitive tasks, or for running a specific set of commands at a specific time.



```
ines@ubuntu:~$ ./version batchfile1.sh
Welcome at Batch mode
ines@ubuntu:~$
```

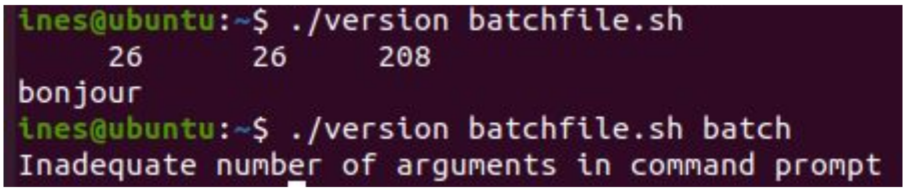
The purpose of our project :

➔ Creating a shell script written in c that support the two mode of execution the interactive and the batch mode. So, our shell will separate the basics and the advanced feature such as I/O redirection, piping, composed command, handling with errors…

➔ In this document we will describe the choices of data structures and algorithms made for our program followed by execution examples.

Main Function

- The main function is divided to many sections :
  The first one will check if the batch file is provided or not to indicate the execution mode of the shell.
  If the argument count is less than 2 : The Batch file not provided and we are Running on interactive mode.
  If the argument count is more than 2 : In this case more than 1 Batch file is provided so it will generate an error.
  If the argument count is 2 : we run the batch file : The command being executed is constructed using the snprintf function, which is used to format the string stored in "command".The shell command being executed later on as "sh" by the value stored in the v[1] array (our batch file). The system function returns the exit status of the command being executed, which is stored in the "ret" variable. If the value of "ret" is not 0, it indicates that there was an error executing the command, and an error message is printed to standard error.



```
ines@ubuntu:~$ ./version batchfile.sh
      26      26     208
bonjour
ines@ubuntu:~$ ./version batchfile.sh batch
Inadequate number of arguments in command prompt
```

(For the screenshot above we removed the system(“clear”) function to show the execution more clearly).

- In the next section, the main function reads a line of input from the user using the fgets function and stores it in the variable "lineInput". The input is then tokenized using the strtok function and stored in the argv array. The "flags" variable is used to specify the delimiter characters (space, newline, and tab) that are used to tokenize the input. If the input is "exit", the code breaks out of the loop. The while loop continues to tokenize the input until there are no more tokens to be found. The argc variable is used to keep track of the number of tokens that were found.

- After tokenization, the main function will work on the commands history. So the code will check if the first token of the input (stored in argv[0]) is "history". If it is, it calls the displayHistory() function and continues the loop, not performing any other commands. If the count variable is 0, indicating that no commands have been entered yet, it prints an error message to stderr saying "No Commands in the history". It then copies the current command lineInput to history[0] and increments count variable. Then checks if the count variable is greater than 20, if yes, it sets it to 20. To keep track of the last 20 commands entered by the user, and provide a history feature.

Main Function

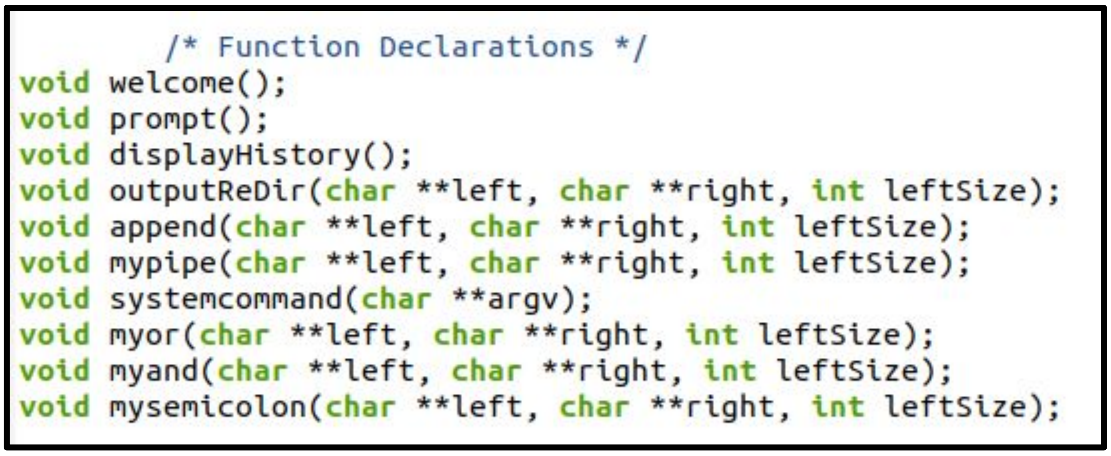
- The next step is to verify if the input stored in argv is one of the special characters : (">", ">>", "|", "||", "&&", ";"). For each character it set the special_c variable to true and it will assigns a number to the variable special from 2 to 8 and also it will indicate for each character's position in the specialpos variable.

- After that, if any special character was found in the input by checking if the special_c variable is set to TRUE. If it is, it checks the position of the special character. If it was found at was found at the first position or the last position, it sets special to 5. This indicate that the input is not in a proper format and cannot be executed . Else If the special character is found in a valid position, it separates the input into two parts: left and right. The left part consists of all the tokens before the special character and the right part consists of all the tokens after the special character.

- The switch statement in the main function will use the value of the special variable to determine which function to take.These functions perform the appropriate action for the special character found in the input. For example, the outputReDir function redirects the output of the command in the left variable to a file specified in the right variable. The mypipe function pipes the output of the command in the left variable as input to the command in the right variable.

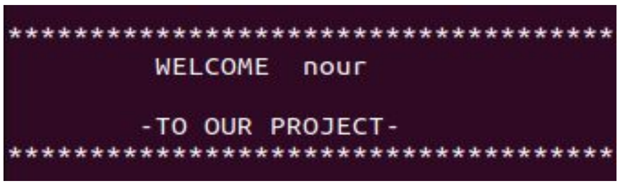
- Finally, the code resets the argc variable to 0 and the special_c variable to FALSE, indicating that no input has been processed yet. It also sets every element of the argv array to NULL, effectively clearing the array. This is done so that the next input entered by the user can be processed and stored in the argv array. This way, the code can be ready to process the next input.

Functions



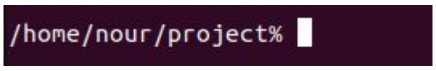
```
        /* Function Declarations */
void welcome();
void prompt();
void displayHistory();
void outputReDir(char **left, char **right, int leftSize);
void append(char **left, char **right, int leftSize);
void mypipe(char **left, char **right, int leftSize);
void systemcommand(char **argv);
void myor(char **left, char **right, int leftSize);
void myand(char **left, char **right, int leftSize);
void mysemicolon(char **left, char **right, int leftSize);
```

- The welcome function is used to display a welcome message to the user : It uses the printf function to print a series of strings and characters to the console. The getenv("USER") function is used to retrieve the current user's username, which is then included in the welcome message.



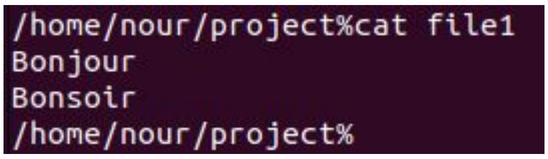
```
*************************************
         WELCOME  nour

        -TO OUR PROJECT-
*************************************
```

- The prompt function is used to display the current working directory as a command prompt for the user : The getcwd function is used to get the current working directory and stores it in the c character array. The first argument of this function is the buffer where the current directory path will be stored and the second argument is the size of the buffer. Then the printf function is used to display the current working directory stored in the c array, followed by the percent sign symbol "% " as the standard command prompt symbol.



```
/home/nour/project% 
```

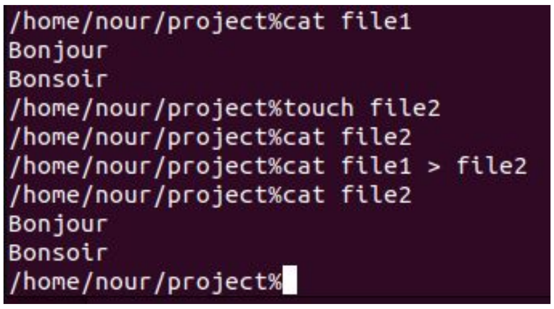
- The systemcommand function runs a command specified by the input argument argv : The fork() function creates a new child process by duplicating the current process. The execvp() function is then called in the child process to execute the command specified by argv, which is an array of strings that contains the name of the command and its arguments. If the execvp() function returns -1, it means that the command was not found, and an error message is printed. The waitpid() function is then called in the parent process to wait for the child process to finish before continuing.



```
/home/nour/project%cat file1
Bonjour
Bonsoir
/home/nour/project%
```

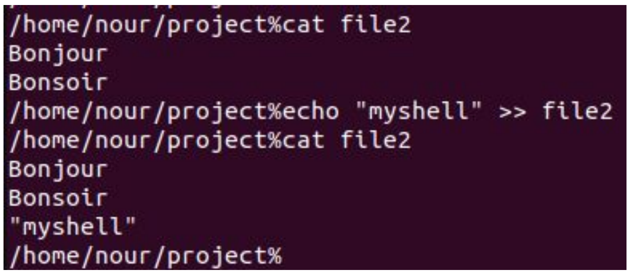
Functions

- The outputReDir function is used to redirect the output of a command to a file : It takes three parameters:
  1- left is a double pointer to an array of strings, which represents the command and its arguments that will be executed.
  2- right is a double pointer to an array of strings, which represents the file to which the output will be redirected.
  3- leftSize is an integer that represents the size of the left command array.
  Then it uses the open function to open the file specified in str in write only mode and truncate or create the file with permissions 0777, and assigns the file descriptor to the variable file.
  Then it uses the dup and dup2 functions to redirect the output of the command from the standard output (stdout) to the file descriptor. The dup function is used to save the current state of the stdout, and dup2 is used to redirect the stdout to the file descriptor, which causes the output of the command to be written to the file.
  After that, it calls the systemcommand function to execute the left command.
  Finally, it uses dup2 function to redirect the stdout back to the saved state, and the close function to close the saved state.



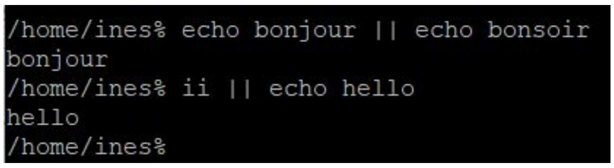
```
/home/nour/project%cat file1
Bonjour
Bonsoir
/home/nour/project%touch file2
/home/nour/project%cat file2
/home/nour/project%cat file1 > file2
/home/nour/project%cat file2
Bonjour
Bonsoir
/home/nour/project%
```

- The append function has the same role as the outputReDir function but the only difference is that it append the output to the file and doesn't truncate it.



```
/home/nour/project%cat file2
Bonjour
Bonsoir
/home/nour/project%echo "myshell" >> file2
/home/nour/project%cat file2
Bonjour
Bonsoir
"myshell"
/home/nour/project%
```

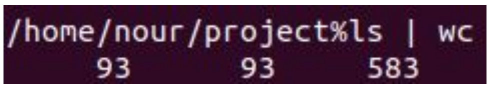
- The myor function only runs the second command if the first one fails :
  the child process execute the left command. Then the child process exits using the exit(1) function. The parent process then checks the exit status of the child process using the WEXITSTATUS(status). If the exit status is not 0, indicating that the left command failed, the parent process forks another child process and runs the execvp() function to execute the right command.



```
/home/ines% echo bonjour || echo bonsoir
bonjour
/home/ines% ii || echo hello
hello
/home/ines%
```

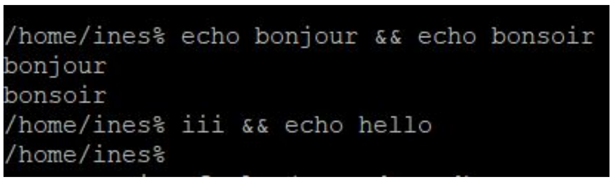
Functions

- The mypie function is used to combine two, the output of one command acts as input to another command: In this specific implementation, the parent process creates a pipe using the pipe() function, before forking the child processes. This creates a pipe that can be used to pass data between the two child processes. Once the pipe is created, the parent process forks the first child process, which in turn forks another child process. These child processes run the execvp() function to execute the left and right commands respectively. The left command writes its output to the pipe, while the right command reads its input from the pipe, allowing the two commands to communicate with each other through the pipe. The parent process waits for the child processes to complete using waitpid(pid,NULL, 0).



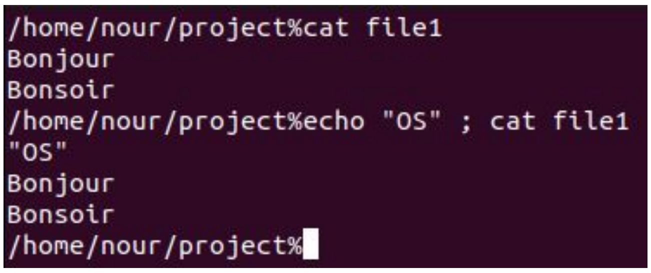
```
/home/nour/project%ls | wc
     93      93     583
```

- The myand function only runs the second command if the first one succeed : the child process execute the left command. Then the child process exits using the exit(1) function. The parent process then checks the exit status of the child process using the WEXITSTATUS(status). If the exit status is equal to 0, indicating that the left command succeed, the parent process forks another child process and runs the execvp() function to execute the right command.



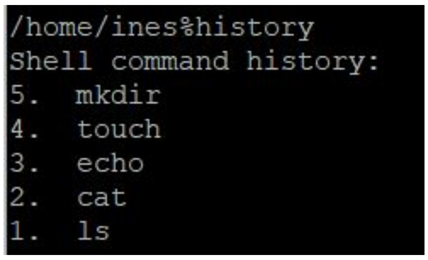
```
/home/ines% echo bonjour && echo bonsoir
bonjour
bonsoir
/home/ines% iii && echo hello
/home/ines%
```

- The mysemicolon function execute the two commands. First, the parent process fork the first child process to execute the left command. Then the parent process waits for the first child to terminate his execution to fork another child process to execute the right command.



```
/home/nour/project%cat file1
Bonjour
Bonsoir
/home/nour/project%echo "OS" ; cat file1
"OS"
Bonjour
Bonsoir
/home/nour/project%
```

- The displayHistory function displays the command history stored in the history array. It enters a for loop that iterates from 0 to 20, or until the histCount becomes 0. Inside the for loop, the function prints the index of the history command using histCount, and then enters a nested while loop to print the characters of the command stored in the history array at the current index. The nested while loop continues until the end of the command is reached, which is indicated by a newline or null character. After the command is printed, histCount is decremented by 1 and the loop continues to the next index.



```
/home/ines%history
Shell command history:
5.  mkdir
4.  touch
3.  echo
2.  cat
1.  ls
```