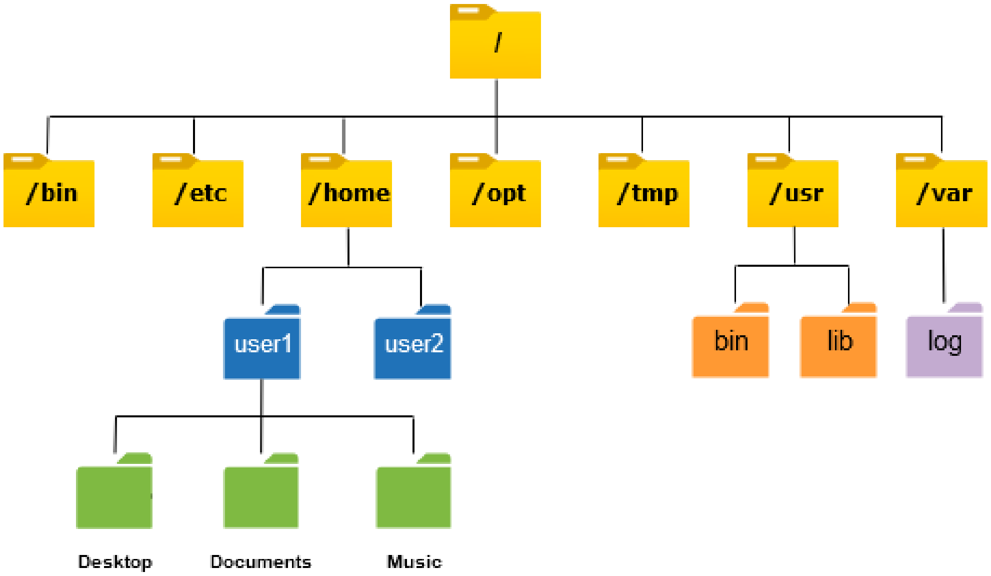
1. Directory Structure in Linux



2. Command Line Interpreter in Linux

2.1 Current Working Directory

Current working directory is the directory in which the user is currently working when the user is interacting with the terminal.

2.2 Some Terminal Commands:

a. **pwd** show current working directory

b. **ls** show the contents of the directory.
Examples:
```
ls
ls ./my_folder
ls /home/user1/Desktop
```

c. **mkdir** create new directories
Examples:
```
mkdir ~/courses    or  mkdir /home/user1/courses
mkdir ~courses/cs604
mkdir ~/courses/cs604/programs
```

You could have created all of the above directories with
mkdir –p ~/courses/cs604/programs command.

d. **cd** change current working directory
Examples:
```
cd ..   (change the working directory to the parent of the current working directory)
cd ../user2
cd /home/user2
```

e. **rm** remove directory/file

```
rm mydir
rm -r mydir
rm 1.txt
rm *.txt
```

f. cp   copy files/contents of directories

Examples

```
cp ~/file1 ~/memos/file2
```

Copy all files of a directory within the current work directory

```
cp dir/* .
```

Copy a directory within the current work directory

```
cp -a tmp/dir1 .
```

Look what these commands do

```
cp -a dir1 dir2
cp filename1 filename2
```

g. **man**   **read the online manual page for a command**

For Command-related Manual:

```
man cp  or man 1 cp
```

For System Call-related Manual:

```
man 2 fork
```

3. Compiling C/C++ program using g++ and gcc:

For C++:

Command: g++ *source_files…* -o *output_file*

For C:

Command: gcc *source_files… -o output_file*

Source files need not be cpp or c files. They can be preprocessed files, assembly files, or object files.

The whole compilation file works in the following way:
Cpp/c file(s)→Preprocessed file(s) → Assembly File(s) Generation → Object file(s) Generation → Linking

Every c/cpp file has its own preprocessed file, assembly file, and object file.

1. For running only the preprocessor, we use -E option.
2. For running the compilation process till assembly file generation, we use –S option.
3. For running the compilation process till object file creation, we use –c option.
4. If no option is specified, the whole compilation process till the generation of executable will run.

A file generated using any option can be used to create the final executable. For example, let’s suppose that we have two source files: math.cpp and main.cpp, and we create object files:

```
g++ main.cpp –c –o main.o
g++ math.cpp –c –o math.o
```

The object files created using above two commands can be used to generate the final executable.

```
g++ main.o math.o –o my_executable
```

The file named “my_executable” is the final exe file. There is no specific extension for executable files in Linux.

4. Command Line Arguments:

Command line arguments are a way to pass data to the program. Command line arguments are passed to the main function. Suppose we want to pass two integer numbers to main function of an executable program called a.out. On the terminal write the following line:

```
./a.out 1 22
```

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers that we have passed as command line argument to the program. These arguments are passed to the main function. In order for the main function to be able to accept the arguments, we have to change the signature of main function as follows:

```
int main(int argc, char *arg[]);
```

- argc is the counter. It tells how many arguments have been passed.
- arg is the character pointer to our arguments.

argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out is also passed as command line argument. At index 0 of arg, we have ./a.out; at index 1, we have 1; and at index 2, we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi. Suppose we want to add the passed numbers and print the sum on the screen:

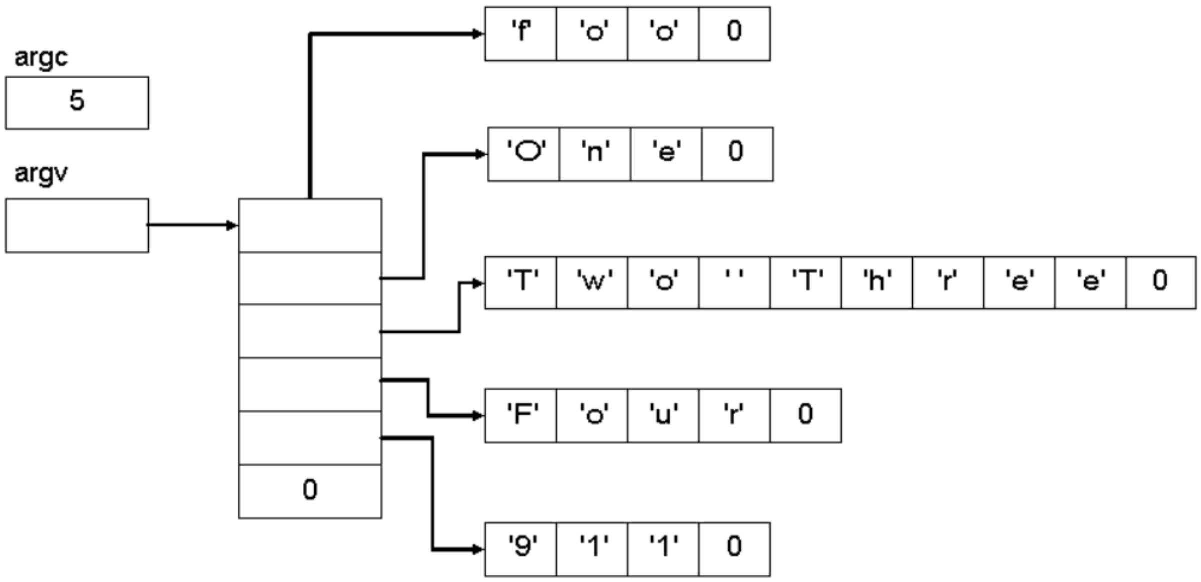
```
cout<< atoi(arg[1]) + atoi(arg[2]);
```

If our executable is in another folder (not in the current working directory), then we need to give the path of that folder before executable name:

```
../usr2/Desktop/a.out 1 22
```

5. Command Line arguments in Memory

```
foo one "two three" four 911
```


argc
5
argv
0
'f' 'o' 'o' 0
'O' 'n' 'e' 0
'T' 'w' 'o' ' ' 'T' 'h' 'r' 'e' 'e' 0
'F' 'o' 'u' 'r' 0
'9' '1' '1' 0