# Linux tutorial

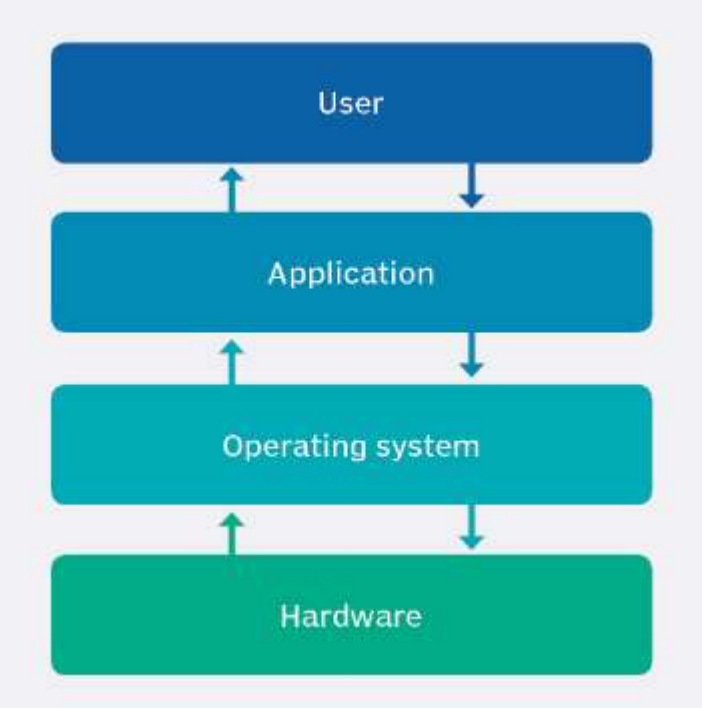
ASP 2024 Pr. M. JEDRA

## Table of contents

# Introduction

**Operating system**:

An operating system (OS) is system software  that manages computer hardware and software resources, and provides common services for Computer programs.




Operating system placement

**Types of operating systems**:

- Mobile operating system
- Real-time operating system
- Embedded operating system
- Network operating system

Examples:

Windows, Unix, Linux, Apple iOS, Mac OS, Xenomai, Android.


**Linux operating system:**

Linux is a powerful and flexible family of operating systems that are free to use and share. It was created by a person named Linus Torvalds in 1991.

**Linux system advantages:**

- A Unix-like Operating System
- Multi-user, Multitasking, Multiprocessor
- Has the X Windows GUI
- Coexists with other Operating Systems
- Runs on multiple platforms
- Open source

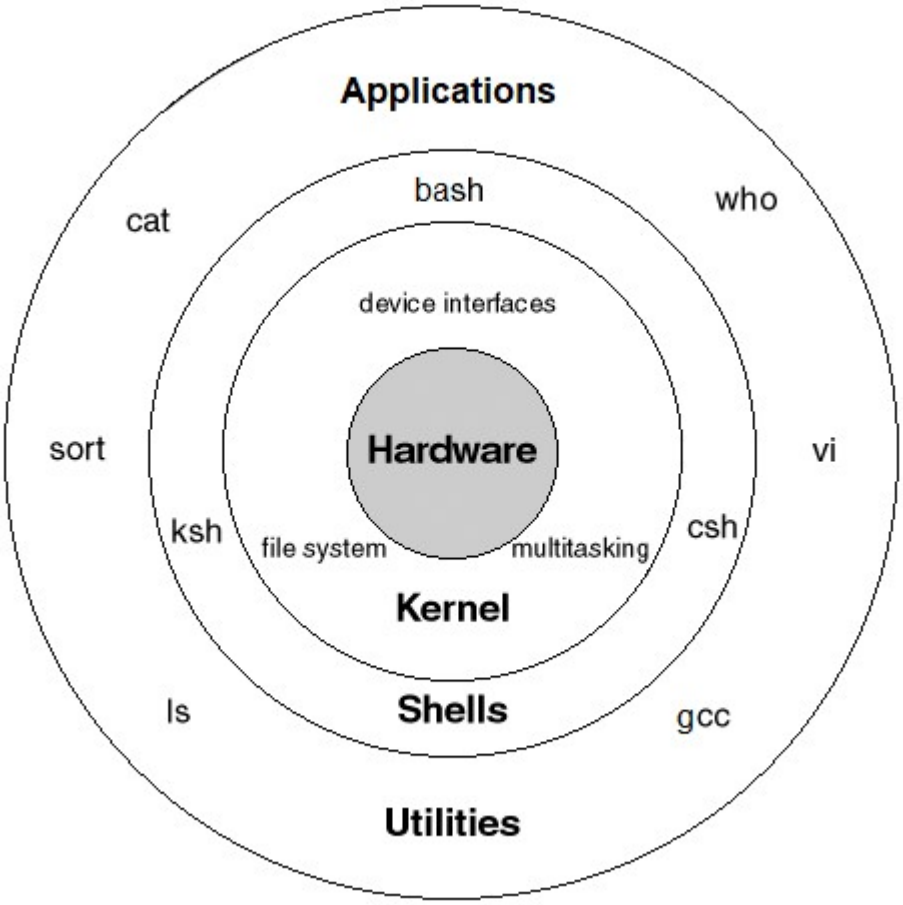
- Easy to install applications.
- Secure
- Stability
- Community
- Free

**Distributions:**

Ubuntu, Fedora, Debian, Arch Linux, CentOS, Kali Linux, Mint, OpenSUSE, Red Hat, Slackware, AlmaLinux,…etc.

# File system

## Linux structure




Architecture of Linux operating system

**Kernel:**

Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. . It allocates CPU time and memory to each program and determines when each program will run. The kernel also provides an interface to programs whereby they may access files, the network, and devices.

**Shell:**

It is an interface among the kernel and user. The shell is a command line interpreter (CLI). It interprets the commands the user types in and executes them. The commands are themselves programs.

**Hardware layer**:

The hardware layer consists of several peripheral devices like CPU, HDD, and RAM.

**Types of shells:**

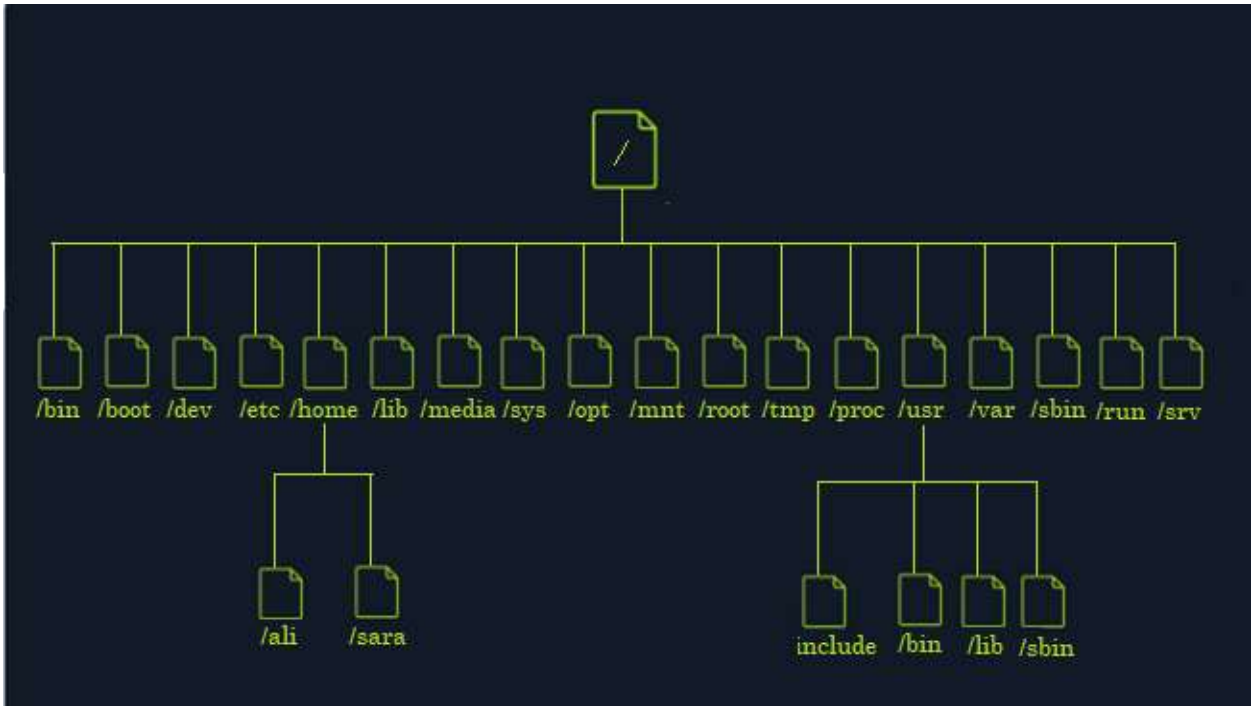
| | | |
|---|---|---|
| **sh** | Bourne shell (standard) | Steve Bourne |
| **csh** | C-shell | Berkley |
| **ksh** | Korn-shell | David Korn |
| **bash** | Bourne again shell (Linux) | |

## File system

**Linux Files**:

- Normal files: data files, executables files and text files
- Directory files: are simply containers for files and other directories.
- Special files: represent interfaces with the devices managed by the system

In the Linux operating system files are stored in a tree-like structure starting with the root directory as shown in the below diagram. The Linux file system hierarchy base begins at the root and everything starts with the root directory.




Linux file system structure

**/** - Root directory that forms the base of the file system. All files and directories are logically contained inside the root directory regardless of their physical locations.

**/bin** - Contains the executable programs that are part of the Linux operating system. Many Linux commands, such as cat, cp, ls, more, and tar, are locate in /bin

**/boot** - Contains the Linux kernel and other files needed by LILO and GRUB boot managers.

**/dev** - Contains all device files. Linux treats each device as a special file.

**/etc** - Contains most system configuration files.

**/home** - Home directory is the parent to the home directories for users.

**/lib** - Contains library files, including loadable driver modules needed to boot the system.

**/media** - Directory for mounting files systems on removable media like DVD-ROM drives, flash drives, …..

**/mnt** - A directory for temporarily mounted file systems.

**/opt** - Optional software packages copy/install files here.

**/proc** - A special directory in a virtual memory file system. It contains the information about various aspects of a Linux system.

**/root** - Home directory of the root user.
**/run -** Gives applications a standard place to store transient files they require like sockets and process IDs.

**/sbin** - Contains administrative binary files. (mount, shutdown, umount, ….).
**/srv -** Contains data for services (HTTP, FTP, etc.) offered by the system.

**/sys** - A special directory that contains information about the devices, as viewed by the Linux kernel.

**/tmp** - Temporary directory which can be used as a scratch directory (storage for temporary files). The contents of this directory are cleared each time the system boots.

**/usr** - Contains subdirectories for many programs such as the X or GUI Window System.

**/usr/bin** - Contains executable files for many Linux commands. It is not part of the core Linux operating system.

**/usr/include** - Contains header files for C programming languages

**/usr/lib** - Contains libraries for C programming languages.

**/usr/sbin** - Contains administrative commands.

**/var** - Contains various system files such as log, mail directories, print spool, etc. which tend to change in numbers and size over time.

Directories can be accessed by their name. Linux uses also the symbols to represent directories.

**Symbols** :

. This directory.

~ Home directory.

.. The parent directory.

/ The root directory.

**Absolute and relative paths:**

**Absolute path-name:** An absolute path is defined as the specifying the location of a file or directory from the root directory (/). To write an absolute path-name:

- Start at the root directory ( / ) and work down.
- Write a slash ( / ) after every directory name (last one is optional)

Example:

```
/user/lib

/etc/network/interfaces
```

**Relative path:** relative path is defined as the path related to the present working directly. It starts at your current directory and never starts with a /.

## User session

```
login: usernamre<rc>
Password: xxxxxxxxxxxxxxxxx<rc>
```

If the login or password is not correct the system give you the response:

```
Login incorrect
Login:username <rc>
Password:xxxxxxxxxxxxxxxx <rc>
username@sysname:~$
```

If the login is correct the prompt of the user appears in the terminal in a common format of `username@sysname:~$`. In this example, the prompt is displaying the username, the sysname, and if that user is using the system as a normal user (`$`) or a super user (`#`). The user can now type a program. Once programs terminate, control is returned to the shell and the user receives another prompt (`$`), indicating that another command may be entered.

The super user on a Linux system is called **`root`**. Anything that can be done on system can be done by **`root`**.

**`exit, logout`**, or **`Ctrl-d`** Exits the shell or your current session.

**`$shutdown –h time “message”`**

The **`shutdown`** command is used for shutting down the system (poweroff) if you are the super user.

Example:

```
$shutdown –h now
```

**Login shell startup files:**

When you run a login shell it reads and executes a number of commands from the files on start-up, in the following order:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bash_login`
- `~/.profile`

When an interactive shell that is not a login shell is started. The shell in this case reads and executes commands from:

- `/etc/bash.bashrc`
- `~/.bashrc`

When an interactive login shell exits, or a non-interactive login shell executes the exit built-in command, the shell reads and executes commands from the file `~/.bash_logout`, if it exists.

# Information commands

They are commands used by the user to obtain information about the system.

### $date [option] [+format]

The **date** command displays the current date and time. It can also be used to set the system date and time. To do this, you need to log in as the **root** user. **option** refers to additional flags that modify the behavior of the date command, **format** indicate in which format the date must be displayed.

Examples:

```
$ date
Tue Jan 25 14:20:34 EST 2022
$ date +%F
2022-01-25
$ date -u
Tue Jan 25 14:20:34 UTC 2022
```

### $who [option] [filename]

The **who** command is a simple and effective way to display information about currently logged-in users.

### $cal [option] [[month]year]

The **cal** command is a calendar command in Linux which is used to see the calendar of a specific month or a whole year.

Example:

```
$ cal 4 2010
    April 2010
Su Mo Tu We Th Fr Sa
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
$
```

## $man [option][command]

The **man** command is used to display the manual pages for other commands and utilities. It provides detailed documentation about the usage, options, and functionality of commands. To quit the man page you must press **q**.

Manual pages are organized into different sections, each serving a specific purpose. The primary sections include:

- **NAME:** Provides the name and a brief description of the command.
- **SYNOPSIS:** Describes the syntax of the command.
- **DESCRIPTION:** Offers a detailed explanation of the command's functionality.
- **OPTIONS:** Lists the available command-line options and their descriptions.
- **EXAMPLES:** Provides practical examples demonstrating command usage.
- **SEE ALSO:** Suggests related commands or resources.

## $echo [argument]

**echo** displays argument to the screen

Example:

```
$echo Hello World
Hello World
$
$echo "Hello World"
Hello World
$
$echo Hello ; echo World
Hello
World
$
```

## $clear

The **clear** command clears the screen.

# Directory manipulation commands

**$mkdir *directory***

The **mkdir** command creates a new ***directory*** in the working directory.

**$rmdir *directory***

The command **rmdir** command removes a ***directory*** in the working directory if it is empty.

**$cd *directory***

The **cd** command changes the current directory to ***directory.*** If you execute this command without specifying a directory, it changes the current directory to your home directory.

**$pwd**

This command displays the path of present working directory.

**$ls [option] directory**

**ls** lists ***directory*** contents.

**[options]**

- **-a** lists all files, including hidden files and directories (their name begin with .)
- **-c** lists files in columns
- **-l** lists files with their permissions
- **-d** lists only directories
- **-l** lists files with their i-nodes

The shell has special characters (wildcards) to define search criteria for file names:

***** represents all characters.

**?** represents a single character.

**[...]** represents a range of characters.
**[!...]** Matches any single character that is not in the range of characters.

To treat ***,?,[ ]** as literals in the text and not as wildcards, you can escape the wildcard by adding '****' before the wildcard.

Examples:

```
$ls *.cpp
FFT_prog.cpp      Spline2.cpp    Prog1.cpp
$
$ls *[0-9]
Calendar2024      classe1       classe2       Texte30
$
$ls ??z
Abz
SOz
77z
$
```

## $tree

The **tree** command-line program is used to recursively list or display the content of a directory in a tree-like format.

# File manipulation commands

**$touch *file***

This command creates a new empty file inside the working directory or update modification time of the file if it exists.

**$cat *file1 file2 …***

The **cat** command concatenates and displays files. This is the command you run to view the contents of a file.

**$rm *file***

**rm** removes a file.

**$rm –rf directory**

**rm –rf** recursively removes the directory and all files and sub directories in the directory structure.

**$mv *sourcefile1 destinationfile***

**mv** moves files or directories. If the ***destinationfile*** is a directory ***sourcefile*** will be moved into ***destinationfile***. Otherwise ***sourcefile*** will be renamed to ***destinationfile***.

**$rename *oldname newname***

The **rename** command is used to rename files.

**$cp *oldfile newfile***

The **cp** command copies ***oldfile*** in ***newfile***

**$cp *file1 file2… directory***

In this case **cp** copies files in the ***directory.***

## $ln oldname newname

The **ln** command is used to create hard or symbolic links to files or directories. A hard link creates a new name for a file or directory (the same i-node). A symbolic link (option –s) creates a new file that contains the path to the original file or directory.

Example:

```
$ls *.cpp
FFT_prog.cpp      Spline2.cpp   Prog1.cpp

$ln Spline2.cpp Splinesource.cpp

$ls –i Spline*.cpp
4540031   Spline2.cpp     3540031 Splinesource.cpp
$
```

## $find [directory] [criteria] [command]

The **find** command is powerful tool used to recursively find files in directory that match criteria. If no arguments are supplied it find all files in the current directory.

**[criteria]:**

**-name <filename>** searches files with specific name
**-user <username>** searches files by owner
**-group <groupname>** searches files by owner
**-type <character>** searches files by type
**-size <n>** searches files by size. n represents the number of blocks (512 bytes)
**-inum <n>** searches files by inode. n represents the inode number
**-mtime <n>** Finds files based on modification time. n represents the number of days ago.
**-perm <n>** searches files by permissions. n represents permissions in octal.

**[command]:**
**-print** print the pathname of each file found
**-exec command {}\;** executes a command on each file found.
**-ok <command> {}\;** execute command on each file with conversational mode

Examples:

```
$find  . -name *.cpp -print
```
Displays found files with suffix .cpp

```
$find  . -size 10 -print
```
Displays found files with the size of 10 block

```
$find  . -size 0 -exec rm {}\;
```
Removes all empty found files

```
$find  . -type d -ok ls -l {}\;
```
Displays the contents of all found subdirectories with their permissions in conversational mode.

## $locate pattern

The **locate** command lists files that match pattern. The **locate** command is much faster than **find** command.

## $a2ps -Pprinter  textfile

This command prints ***textfile*** by named ***printer.***

# Working with file contents

**$file *pathfile***

The command **file** determine the type file of ***pathfile*** .

**$head *file***

The **head** command writes the first ten lines of a file to the screen.

**$head –n N *file***

The **head** command can also display the first N lines of a file

**$tail *file***

Similar to **head**, the **tail** command writes the last ten lines of a file to the screen

**$less *file***

The command **less** writes the contents of a file onto the screen a page at a time.

**$more *file***

Similar to **less**, the **more** command is useful for displaying the contents of the file page by page. To see the next page the user must use the space bar, or **q** to quit.

# Redirections

## Standard input/output

When a user logs into the system three streams are opened and one number called descriptor is assigned to each of those streams:

standard input : **stdin** 0 (keyboard where commands are typed)

standard output: **stdout** 1 (screen were the results of the commands are displayed)

standard error : **stderr** 2 (screen where errors are displayed)


## Redirections

In Linux we can redirect the input and the output of commands.

**>** redirect the standard output to a file, overwriting any existing contents of the file. If no file exists, it creates one.

**>>** redirect the standard output to a file and appends to any existing contents. If no file exists, it creates one.

**<** redirect the standard input from a file to the command preceding the less-than sign.

**<<** redirect the standard input to here-is-document is a way to append input until a certain sequence (usually EOF) is encountered.

Examples:

```
$ls
Student
Professeur
Spline.cpp
$ls > F
$ls
Student
Professeur
Spline.cpp
F
$cat F
Student
Professeur
Spline.cpp
$date >> F
$cat F
Student
Professeur
Spline.cpp
Tue Jan 25 14:20:34 EST 2022
$
```

`$wc -l< F > G  wc -l` count lines number of F and prints it in file G

```
$cat G
4
$
$wc << end
a b c d
e f g h
end
2 8 16
$
```

# Pipes

Piping is when you take the output of one command and use it as an input to another command. The pipe (|) metacharacter is placed between two commands to achieve this.

***$command1 | command2***

Pipes are unidirectional and usually used to avoid using temporary files.

Example:

`$who|wc -l` count the number of users connected to the system

# Filters

## **$sort *file***

The **sort** filter sorts the file content in an alphabetical order.

Example:

```
$cat  Fruits
Banana
Apple
Orange
Kiwi
Lemon
Cheery
Avocados
Pear
Peach
$sort Fruits
Apple
Avocados
Banana
Cheery
Kiwi
Lemon
Orange
Peach
Pear
$
```

## **$grep 'string' *textfile***

The most common use of **grep** is to filter lines of text containing (or not containing) a certain string.

Example:

```
$cat  Fruits
Banana
```

```
Apple
Orange
Kiwi
Lemon
Cheery
Avocados
Pear
Peach
$
$grep  Pea Fruits
Pear
Peach
$
```

### $wc *textfile*

**wc** counts words, lines and characters in the text.

### $cut *file*

The **cut** filter can select columns from files, depending on a delimiter or a count of bytes.

Example:

`$ls -l |cut -d " " -f 1` displays the column of permissions

### $*tr* [option] *set1* [*set2*] < *file*

**tr** translate or delete characters in a file. If we don't pass any options to **tr**, it will replace each character in ***set1*** with each character in the same position in ***set2***.

Example:

```
$cat  Fruits | tr 'o' 'O'
Banana
Apple
Orange
Kiwi
LemOn
Cheery
AvOcadOs
Pear
Peach
$
```

**$uniq *file***

**uniq** removes duplicates lines from a file.

# User management

In Linux each user is registered in two system files: /etc/passwd and /etc/group.

## The /etc/passwd file

The /etc/passwd file contains basic user attributes. This is an ASCII file that contains an entry on a single line for each user.

An entry in the **/etc/passwd** file has the following form:

*Name*:*Password*: *UserID*:*GroupID*:*Gecos*: *HomeDirectory*:*Shell*

Example:

```
sara:x:3450:3450: Cadi Ayyad:sara:/bin/bash
```

**$useradd *username***

The **useradd** command creates a new user account.

**$userdel *username***

The **userdel** command deletes a user account.

**$usermod *username***

The **usermod** command modifies user account attributes such as username.

**$whoami**

The **whoami** command is used to display the username of the current user.

**$id**

The **id** command will give you your user id, primary group id, and a list of the groups that you belong to.

**$su *username***

The **su** command lets you switch to another user's account or execute commands as a different user.

**$sudo *command***

**sudo** is a command in Linux that allows users to run commands with privileges that only root user have.

## The /etc/group file

The **/etc/group** file contains basic group attributes. This is an ASCII file that contains records for system groups. Each record appears on a single line and is the following format:

*Name*:*Password*:*ID*:*User1*,*User2*,...,*Usern*

Example:

```
student:x:3450:sara
```

**$groupadd *groupname***

The **groupadd** command creates a new group

**$groupdel *groupname***

The **groupdel** command removes a group

**$groups**

The **groups** command to see a list of groups where the user belongs to.

## The /etc/shadow file

User passwords are encrypted and stored in **/etc/shadow**. The /etc/shadow file is read only and can only be read by root.

An entry in the **/etc/shadow** file has the following form:

*Name*:*Password*:*Last change*:*Min age*:*Max age*: *warn*:*inactive*:*expire*:::

Example:

```
sara:$1$NAnoMEmP$GgRfy2.YxwJ6Mnb/cDyM3.O/:14564:0:90:7:::
```

- Username: `sara`
- Encrypted password: `$1$NAnoMEmP$GgRfy2.YxwJ6Mnb/cDyM3.O/`
- Last password change: `14564 days` (since January 1, 1970).
- Minimum password age: `0 days`
- Maximum password age: `90 days`
- Password warning period: `7 days`
- Account Expiration Date: (empty, indicating no expiration)

**$passwd *username***

The **passwd** command sets and changes passwords for users. For changing password users will have to provide their old password before twice entering the new one.

Example:

```
$passwd
Changing password for sara
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
$
```

# File permissions

## File permissions definition

Every file or directory within Linux has a set of permissions that control who may read, write and execute the contents. Each of these permissions is represented by an abbreviation and has an octal value.

| | **Abbreviation** | **Octal value** | **File** | **Directory** |
|---|---|---|---|---|
| read | r | 4 | The file can be viewed or copied. | The contents of the directory can be listed |
| write | w | 2 | Allows the content of the file to be modified. | Files can be created or deleted within the directory. |
| execute | x | 1 | The file can be executed (shell scripts or executables only). | Access to the directory is controlled. |

The **`ls -l`** command lists files and directories of a directory. The displayed list contains a single detailed information line for each file. It is organized in nine columns.

Example:

```
$ls –l
-rwxrwxr-x 1 sara sara 5224 Dec 30 03:22 hello
-rw-rw-r-- 1 sara sara  221 Dec 30 03:59 hello.c
drwxrwxr-x 5 sara sara 1024 Dec 31 14:52 data
$
```

The first column contains ten characters for each file. The first character indicates the file type followed by three groups of three characters.

```
<type><user><group><others>
   -    rwx     rwx     r-x
```

| First character | File type |
|---|---|
| **-** | normal file |
| **d** | directory |
| **l** | symbolic link |
| **p** | named pipe |
| **b** | block device |
| **c** | character device |
| **s** | socket |

All access restrictions apply to users, only one user is exempt from access controls: the super user with login: **root** and UID=0

## Changing file permissions

To change the file permissions we use the **chmod** command. Only the owner of a file can change the permissions for user (**u**), group(**g**), or others (**o**), by adding (**+**)or subtracting(**-**) the read, write, and execute permissions.

**$chmod [option] *permission file***

Typically, the **chmod** command is used in two ways: the symbolic method and the absolute form.

The first way is the symbolic method, which lets you specify permissions with combination of abbreviations.

| User class | Operator | Access type |
|---|---|---|
| u(user) | +(add access) | r |
| g(group) | -(remove access) | w |
| o(others) | =(set exact access) | x |
| a(all) | | |

Examples:

`$chmod u+x F` Add the read permission to the user

`$chmod u+x F` Add the write permission to the group

`$chmod o=x F` the permissions of others are only set to x

`$chmod u+x F` Add the x permission to all

The other way to use the **chmod** command is the absolute form, in which you specify a set of three numbers that together determine all the access classes and types. Add the numbers of the permissions you want to give for each type of users.

Example:

```
$chmod 664 F
```

664 = -rw-rw-r--

## Default file permissions:

When creating a file or directory, a set of default permissions are applied. These default permissions are determined by the file creation mask. The **umask** command displays or sets the file creation mask mode.

**$umask [-S] [mode]**

Sets the file creation mask to mode if specified in octal or symbolic form. If mode is omitted, the current mode will be displayed. Using the **-S** argument allows **umask** to display or set the mode with symbolic notation.

An easy way to understand change defaults permissions is to specify the mask mode in the octal form. The value we pass as an argument is subtracted from the max/full permission set. There are two full permission sets:

- **File ->** The full permission set for a file is 666 (rw-rw-rw-)
- **Directory ->** The full permission set for a directory is 777(rwxrwxrwx)

Example:

```
$umask 022
```

File-> 666-022 = 644 = -rw-r- -r- -

Directory->777-022=755= drwxr-xr-x

# Process control

## Process

A process = program in execution + environment

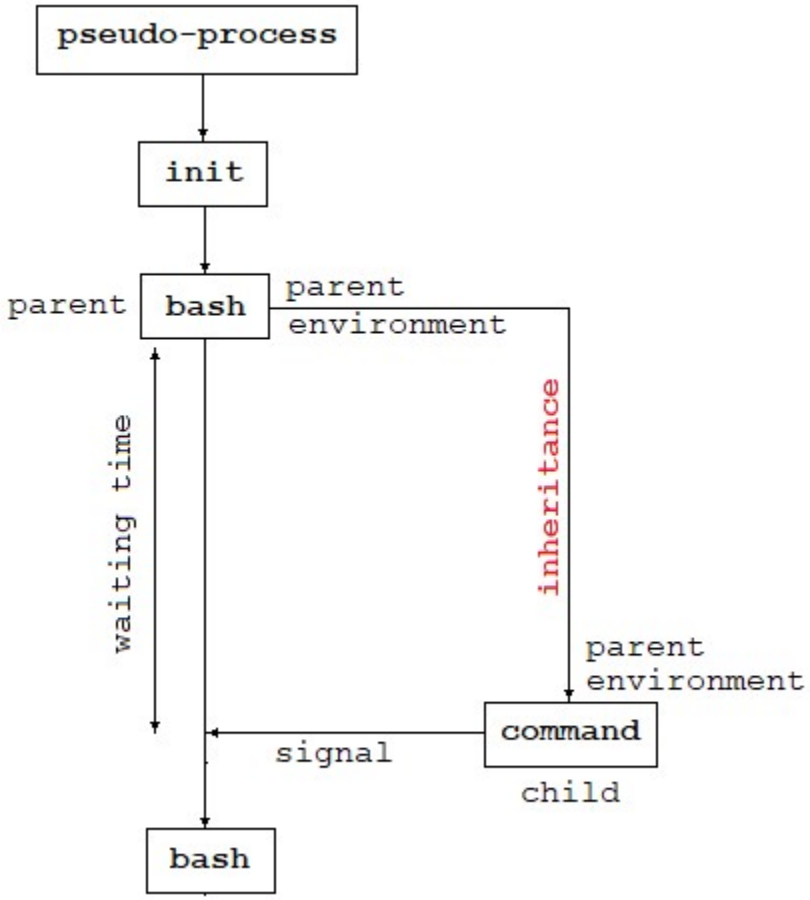
The environment is defined by a set of information provided by the operating system to the program to be executed correctly. Those information are:

- the process identifier (PID)
- the parent process identifier (PPID)
- user identifier (UID)
- group identifier (GID)
- working directory
- Priority (NI)
- CPU time taken by the process (TIME)
- process start time (STIME)
- terminal type associated with the process (TTY)

- ……

## Process Creation

A new process can be created by the fork mechanism. The new process consists of a copy of the address space of the original process. Fork mechanism creates new process from existing process. Existing process is called the parent process and the process created newly is called child process. Each process has its own environment, which is copied from the parent process's environment.




Linux process creation

The system boot process (pseudo-process) has no parent. Its PID is equal to 0.

The **`init`** process is the child of the boot system process. Its PID is equal to 1. It is responsible directly or indirectly for all process in the system. It displays login in the screen.

A user connected signifies a process shell is running.

A process can be run in two ways:

**Foreground process:** Every process when started runs in foreground by default, receives input from the keyboard, and sends output to the screen.

**Background process:** It runs in the background without keyboard input and waits till keyboard input is required. It is disconnected from the terminal and cannot communicate with the user.

Adding **&** along with the command starts it as a background process.

A process in Linux can go through different states after it's created and before it's terminated. These states are:

- Running
- Sleeping
- Stopped
- Zombie

A process in **running** state means that it is running or it's ready to run.

The process is in a **sleeping** state when it is waiting for a resource to be available.

A process enters a **stopped** state when it receives a stop signal.

**Zombie** state is when a process is dead but the entry for the process is still present in the process table.

**Daemon process:** daemon process is a process which runs continuously in background until the system shutdown (for example **`kswapd`**), daemon process usually starts with the system boot up and unlike other process it does not respond to signals from the keyboard.

# Process management commands

## $ps [option]

The **ps** command can be used to list all the running processes.

```
$ps
  PID TTY          TIME CMD
16524 pts/0    00:00:00 bash
17319 pts/0    00:00:00 ps
$
```

## $top

The **top** command is used to show all the running processes within the working environment of Linux.

## $jobs [option]

The **jobs** command lists all currently running background jobs.

## $kill [-signal number] PID

The **kill** command terminates a process. The process receives a signal from the **kill** command. There are numerous signal types that you can use.

## $sleep <nombre>[suffix]

The **sleep** command is used to delay the execution of scripts or commands for a specified amount of time. The default delay time si in seconds and you can set it in minutes (m), hours (h), and days (d).

```
$sleep 5;date
Sat Apr 17 13:08:27 CEST 2024
$
```

### $nice –n <-nice value> <command>

The **nice** command starts a new process and assigns it a priority nice value at the same time. Nice value ranges from -20 to 19, where -20 is of the highest priority. 0 is the default value.

### $nohup command [option] &

The **nohup** command is used to run a command in such a way that it continues to run even after you log out or close the terminal.

Once a job is started or executed using the nohup command, **stdin** will not be available to the user and **nohup.out** file is used as the default file for **stdout** and **stderr**. If the output of the nohup command is redirected to some other file, **nohup.out** file is not generated.

### $time [option] command

The **time** command can display how long it takes to execute a command. This command displays real time (the time from start to finish of the call), user time (amount of CPU time spent in user mode) and system time (amount of CPU time spent in kernel mode).

Example:

```
$time date
Sat Apr 17 13:08:27 CEST 2010
real 0m0.014s
user 0m0.008s
sys  0m0.006s
$
```

### $at [option] <runtime> command

The **at** command is capable of executing a command at a specified time and date, or at a given time interval. We can use minutes, hours, days, or weeks.

Examples:

```
$at 16:00 command
$at 17:15 Fri command
$at noon command
$at noon
>command1
>command2
>command3
…
Ctrl+D
$
```

**$df [option]**

The **df** command shows the amount of disk space available being used by the file systems.

**$free [option]**

The **free** command shows the free space and used space of the memory RAM in the system.

**$du [option] [directory/file]**

The **du** command is a powerful utility that allows users to analyze and report on disk usage within directories and files.

# Environment variables

In Linux environment variables are a set of dynamic named values, stored within the system that are used by applications launched in shell. These variables have a name and their respected value. They allow the user to customize how the system works and the behavior of the applications on.

**Common environment variables:**

| | |
|---|---|
| EDITOR | The program to run to perform edits. |
| HOME | The Home directory of the user. |
| LOGNAME | The login name of the user. |
| MAIL | The location of the user's local inbox. |
| OLDPWD | The previous working directory. |
| PATH | A colon separated list of directories to search for commands. |
| SHELL | The path to the current user shell |
| PS1 | The primary prompt string. |
| PWD | The present working directory. |
| USER | The username of the user. |

**$echo $VARIABLE_NAME**

**echo** displays the specified environment variable.

Example:

```
$echo $HOME
/home/sara
$
```

**$printenv [VARIABLE_NAME]**

**printenv** displays all or the specified environment variables.

Example:

```
$printenv HOME
/home/sara
$echo $HOME
/home/sara
$printenv
TERM=xterm-256color
SHELL=/bin/bash
USER=sara
PATH=/usr/local/bin:/usr/bin:/bin
MAIL=/var/mail/sara
PWD=/home/sara
LANG=en_US.UTF-8
HOME=/home/sar
…
$
```

**$env [option] [Variable= value]….[Command]**

The **env** command displays the current environment or sets the environment for the execution of a command.

**$export Variable=[value]**

The **export** command is used to set or export variables to child processes. The variables that are not exported are called local variables. The **export** command allows variables to be used by subsequently executed commands.

Example:

```
$ TUTORIAL=Linux
$echo $TUTORIAL
Linux
$bash
$echo $TUTORIAL
$exit
exit
```

```
$export TUTORIAL
$bash
$echo $TUTORIAL
Linux
$exit
exit
$
```

In the above example `TUTORIAL` was defined in the current environment. When you start a child process it inherits all the environment variables that were exported in your current environment. Since `TUTORIAL` was not exported it was not set in the spawned bash shell. When you have exported `TUTORIAL` you saw that it was indeed available in the child process.

**$export -p**

This command lists all names that are exported in the current shell.

**$set [option] [arguments]**

The **set** command displays local and environment variables.

**$unset VARIABLE_NAME**

The **unset** command deletes shell and environment variables.

Example:

```
$echo $TUTORIAL
Linux
$unset TUTORIAL
$echo $TUTORIAL
$
```

**$history**

**histoy** displays a list of commands in the shell history.

**$alias [name=[value]]**

**`alias`** lists or creates aliases. If no arguments are provided the current list of aliases is displayed.

Example:

```
$ls -l
total 4
-rw-r--r-- 1 sara sara 221 Nov 13 11:30 file.txt
….

$alias ll='ls -l'
$ll
total 4
-rw-r--r-- 1 sara sara 221 Nov 13 11:30 file.txt
…..
$
```

# Basic Linux tools

**$compress [option] *Textfilename***

This command compress the ***Textfilename*** and place it in a file called ***Textfilename.Z***

**$uncompress [option] *Textfilename.Z***

This command uncompress the file ***Textfilename.Z*** .

**$gzip [option] *Textfilename***

This command **gzip** zip the file ***Textfilename*** and place it in a file called ***Textfilename*.gz.**

**$gunzip [option] *Textfilename.gz Directory***

**guzip** is used to unzip the file ***Textfilename*.gz**

**$tar -cvf *Archivename.tar Directory***

The **tar** program compress a ***Directory*** in the file ***Archivename.tar***

**-c** create archive
**-v** verbose i.e display progress while creating archive
**-f** archive file name
**-x** extract archive
**-z** compress archive using **gzip** program.

**$tar -cxf *Archivename.tar***

The **tar** program extracts an archive in the current directory.

**$gcc [option] source.c**

The **gcc** command is used to compile a source files written mainly in C or C++ language. The output file obtained after compiling the source file is named **a.out** if no name is specified before. The output file can be executed using **./a.out.**

Example:

```
$gcc -o hello hello.c
$./hello
Hello World
$
```

**$nano hello.c**

**nano** is a command line text editor like **vi**, **vim,** et **emacs**