Jose L. Muñoz, Juanjo Alins, Oscar Esparza, Jorge Mata

Linux Essentials

Fonaments Xarxes Telemàtiques (FXT)
Universitat Politècnica de Catalunya (UPC)
Dp. Enginyeria Telemàtica (ENTEL)

Contents

I Linux Essentials **5**

1 Introduction to Unix/Linux **7**
1.1 Introduction to OS . . . 7
1.2 Resources Management . . . 7
1.2.1 History . . . 7
1.2.2 OS Rings and the Kernel . . . 8
1.2.3 System Calls . . . 9
1.2.4 Modules . . . 9
1.3 User Interaction . . . 10
1.4 Implementations and Distros . . . 11
1.5 Switching Users . . . 11
1.6 Installing Software . . . 12
1.6.1 Static and Dynamic Libraries . . . 12
1.6.2 Sofware Packages . . . 14
1.6.3 Advanced Package Management Systems . . . 14
1.6.4 Installing from the Source . . . 15
1.6.5 Command Summary . . . 15

2 Processes **17**
2.1 Booting the System . . . 17
2.2 Listing Processes . . . 17
2.3 The `man` Command . . . 18
2.4 Working with the Terminal . . . 18
2.5 Other Commands for Processes . . . 19
2.6 Scripts . . . 19
2.7 Running Processes in Foreground/Background . . . 21
2.8 Signals . . . 21
2.9 Job Control . . . 22
2.10 Capturing Signals in Scripts: `trap` . . . 23
2.11 Running Multiple Commands . . . 24
2.12 Extra . . . 24
2.12.1 *Threads . . . 24
2.12.2 *Terminal Associated Signals . . . 24
2.12.3 *States of a Process . . . 25
2.12.4 *Priorities: `nice` . . . 25
2.12.5 Command Summary . . . 26
2.13 Practices . . . 26

3 Filesystem **29**
3.1 Introduction . . . 29
3.2 Basic types of files . . . 29
3.3 Hierarchical File Systems . . . 30
3.4 Storage Devices . . . 31
3.5 Disk Usage . . . 31
3.6 The path . . . 32
3.7 Directories . . . 33
3.8 Files . . . 33
3.9 PATH variable . . . 34
3.10 File content . . . 34
3.11 File expansions and quoting . . . 34
3.12 Text Files . . . 35
3.13 Commands and Applications for Text . . . 36
3.14 Links . . . 37
3.15 Unix Filesystem Permission System . . . 38
3.16 Extra . . . 40
3.16.1 *inodes . . . 40
3.17 Command summary . . . 41
3.18 Practices . . . 41

Part I

Linux Essentials

Chapter 1

Introduction to Unix/Linux

1.1 Introduction to OS

This chapter provides some basic background about the Linux Operating System. In short, an Operating System (OS) is a set of software whose purpose is to **(i)** manage the resources of a computer system while **(ii)** providing an interface for the interaction with human beings.

- **Resources management**. The computer resources that can be managed by an OS include CPU, RAM, and input/output (I/O) devices. For example, in all modern computer systems, it is possible to run more than one process simultaneously. So, the OS is responsible for allocating the CPU execution cycles and RAM memory for each process. Regarding I/O devices, these include storage devices like a Hard Disk (HDD), a Compact Disk (CD), a Digital Versatile Disc (DVD), a Universal Serial Bus (USB) device, etc. but also communication devices like wired networking (Ethernet cards) or wireless networking (WIFI cards). An introduction to the organization of Linux that allows this OS to achieve a proper way of managing and accessing system resources is provided in Section 1.2.

- **User Interaction.** Another essential issue is how the OS allows interaction between the user (human being) and the computer. This interaction includes operations over the file system (copy, move, delete files), execution of programs, network configuration, and so on. The two main system interfaces for interaction between the user and the computer - CLI and GUI - are further discussed in Section 1.3.

1.2 Resources Management

1.2.1 History

Nowadays, most deployed OS come either from Microsoft WINDOWS/DOS or from UNIX. We must remark that UNIX and DOS were initially designed for different purposes. While DOS was developed for rather simple machines called Personal Computers (PC), UNIX was designed for more powerful machines called Mainframes. DOS was originally designed to run only one user process[1] or task at a time (mono-process or mono-task) and to manage only one user in the system (mono-user).

On the other hand, UNIX was directly designed as a multi-user system. This has numerous advantages, security for example, which is necessary for protection of sensitive information, was designed at the very beginning in UNIX. UNIX was designed also as multi-task being able of managing several users running several programs at the same time. As shown in Figure 1.1, today both types of OS (WINDOWS/UNIX) are capable of managing multiple users and multiple processes and also both can run over the same type of hardware. However, we would like to point out that many of these capabilities were present in the first design of UNIX, while they have been added in DOS-like systems.

[1] A process is basically a running program.

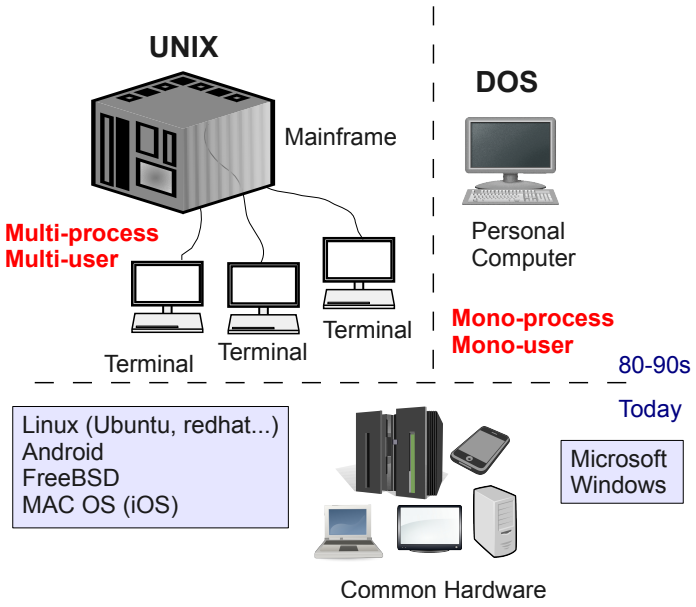


Figure 1.1: Origins of Linux.

1.2.2 OS Rings and the Kernel

Modern OS can be divided in at least three parts: **hardware**, **kernel** and **user space** (user applications or processes). See Figure 1.2.

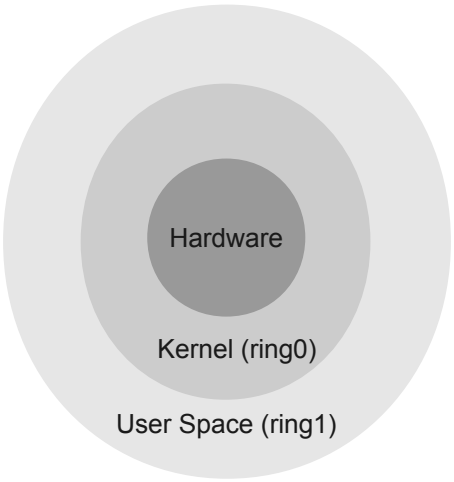


Figure 1.2: OS Rings.

These parts can be seen as “rings“. The kernel or ”ring 0” is an intermediary between user applications and the actual data processing done at the hardware level. The kernel’s primary function is to manage the computer’s resources and allow other programs to run and use these resources. These resources are at least:

- **Central Processing Unit (CPU**). The CPU is responsible executing programs. The kernel takes responsibility for deciding which of the running programs should be placed in the processor or processors.
- **Memory.** Memory is used to store both program instructions and data. For a program in execution both program

instructions and data need to be present at the memory. Often, multiple programs may want to access to memory and the kernel is responsible for deciding which memory each process can use, and determining what to do if not enough memory is available.

- **Input/Output (I/O) Devices.** These devices add some functionality to the system. Examples are keyboard, mouse, disk drives, printers, displays, etc. The kernel manages requests from user applications to perform input and output operations and provides convenient methods for using each device (typically abstracted to the point where the application does not need to know implementation details of each device).

1.2.3 System Calls

The kernel manages all the low level operations with the hardware (CPU, memory and other devices). To do so, the kernel runs in what is called "supervisor mode" which provides unrestricted access to hardware. On the other hand, the kernel provides an interface for user processes so that they can use the operating system. This interface is implemented with **system calls** (see Figure 1.3). A system call defines how a user program or user application has to request a service from the kernel. For example, a system call may allow a user program to save a file in a Hard Disk but not to write bytes at certain locations of the HDD (this operation might require supervisor mode).

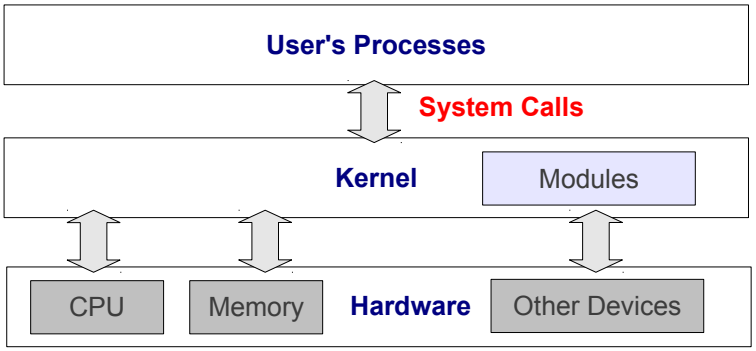


Figure 1.3: System Calls & Modular Design.

Generally, systems provide a library or API that sits between user programs and the Kernel. On Unix-like systems, that API is usually part of an implementation of the C library (libc), such as glibc, that provides wrapper functions for the system calls, often named the same as the system calls that they call. For example:

```
int kill(pid_t pid, int sig);
```

The previous function is a wrapper for a system call called kill, which is used to send a signal to a process.

1.2.4 Modules

The Linux Kernel is a **monolithic hybrid kernel**. Monolithic means that the entire operating system is working in kernel space and is alone in supervisor mode. Unlike traditional monolithic kernels, the Linux Kernel is hybrid. This means that kernel extensions, called modules, can be loaded and unloaded into the kernel upon demand while the kernel is running. In other words, modules are pieces of code that extend the functionality of the kernel without the need to reboot the system.

One type of module is the device driver, which allows the kernel to access a hardware device connected to the system. Device drivers interact with devices like hard disks, printers, network cards etc. and provide system calls.

Notice that a traditional monolithic kernel (without the possibility of having modules) has to include in its code all the possible device drivers. This has two main drawbacks: we will have larger kernels and we will need to rebuild and reboot the kernel each time we want a new functionality.

Finally, when you build a Linux kernel you can decide if you include a certain module inside the kernel (statically compiled module) or if you allow this module to be loaded at run time by your kernel (dynamic module).

The commands to list, insert and remove modules are: `lsmod`, `modprobe` and `rmmod`.

1.3 User Interaction

In the past, mainframe systems had several physical terminals connected via a serial port (often using the RS-232 serial interface). Terminals were simple monochrome displays with the minimal hardware and logic to send the text typed by the user in a keyboard and display the text received from the mainframe in the display (see Figure 1.4). On the mainframe-side, there was a command-line interpreter (CLI) or *shell*. A *shell* is a process that interprets and executes commands.

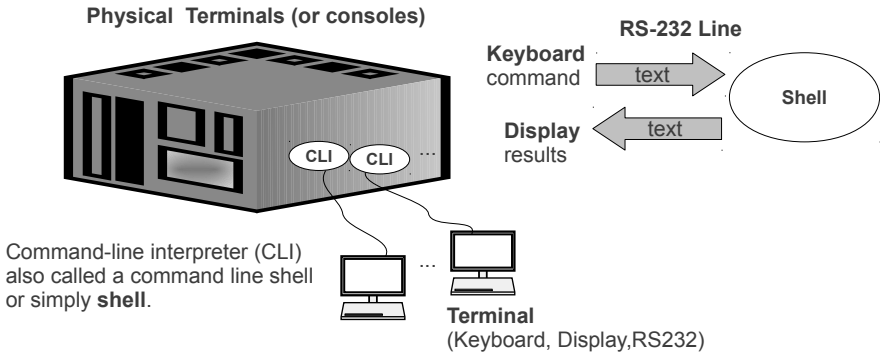


Figure 1.4: Mainframe with Old Physical Terminals.

In current systems, we have also Graphical User Interfaces or GUIs. GUI requires a graphical server (often Unix systems use the **X** server). Processes launched from the GUI are typically applications that have graphical input/output. This graphical I/O is implemented with devices such as a mouse, a keyboard, a screen, a touch screen, etc. GUIs are easier to use for novel users but in general, CLI provides you with more control and flexibility than GUI for performing system administration & configuration.

Physical terminals are not very common today, instead, **virtual consoles** are used. If you use virtual consoles to interact with our Linux system, you will not require a graphic server running on the system. Virtual consoles just manage text and they can emulate the behavior of several physical terminals. The different virtual consoles are accessed using different key combinations. By default, when Linux boots it starts six virtual consoles which can be accessed with **CRL+ALT+F1** ... **CRL+ALT+F6**. The communication between the virtual console and the *shell* is performed using a special device file in the system of the form /dev/ttyX (where X is the number of virtual console). This device file is called TTY and it emulates the "old physical communication channel". To interact with the terminal you can write to and read from the TTY. If you want to see the TTY of a terminal just type the `tty` command:

```
$ tty
/dev/tty1
```

Commands executed from a terminal are connected with a *shell* using the TTY. In the Unix jargon, it is said that the command is "attached" to a TTY.

On the other hand, we can also use a GUI if our Linux system has an graphical server running. In fact, the GUI is the default interface with the system for most desktop Linux distributions. To go from a virtual console to the graphic server, you can type **CRL+ALT+F7** in the majority of the Linux systems. Once you log into the GUI, you can also start a "terminal". In this case, the terminal is called **terminal emulator** or **pseudo-terminal**. For example, to start a pseudo-terminal you can use **ALT + F2** and then type `gnome-terminal` or `xterm`. You can also use the main menu: **MENU-> Accessories-> Terminal**. This will open a `gnome-terminal`, which is the default terminal emulator in our Linux distribution (Ubuntu). In Figure 1.5 you can observe the main features of virtual consoles and pseudo-terminals including their TTY device files. Notice that in the case of pseudo-terminals, the TTY is of the form /dev/pts/X (where X is the number of pseudo-terminal).

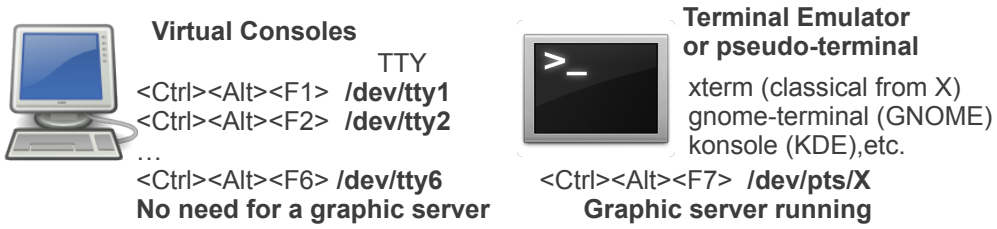


Figure 1.5: Linux Terminals.

Regarding the *shell*, this documentation refers only to Bash (Bourne Again Shell). This is because this *shell* is the most widely used one in Linux and includes a complete structured programming language and a variety of internal functions.

Note. When you open a terminal (either a virtual console or a pseudo-terminal) you will see a line of text that ends with the dollar sign "$" and a blinking cursor. When using "$" throughout this document, we will mean that we have opened a terminal with our user that is ready to receive commands.

1.4 Implementations and Distros

UNIX is now more a philosophy than a particular OS. UNIX has led to many implementations, that is to say, different UNIX-style operating systems. Some of these implementations are supported/developed by private companies like Solaris of Sun/Oracle, AIX of IBM, SCO of Unisys, IRIX of SGI or Mac OS of Apple, Android of Google, etc. Other implementations of Unix as " Linux" or "FreeBSD" are not commercial and they are supported/developed by the open source community.

In the context of Linux, we also have several distributions, which are specific forms of packaging and distributing Linux (Kernel) and its applications. These distributions include Debian, Red Hat and some derivate of these like Fedora, **Ubuntu**, Linux Mint, SuSE, etc.

1.5 Switching Users

We need some method to interact with the system as superuser (or as another user). Obviously, one possibility is to log into the system using the proper user account but it would be desirable to have commands that enable this without having to "relog". To this respect, we can use the commands `su` and `sudo`. The `su` command stands for "switch user", and allows you to become another user or execute commands as another user. For example:

```
$ su telematic
```

The previous command prompts you for the password of the user "telematic". If you don't provide a user, the `su` command defaults to the root account, which in Unix is the system administrator account. In either case, with `su` you will be prompted to enter **the password associated with the account to which you are switching**. After you execute the `su` command you will be logged as the new user until you exit. You can exit typing **`Ctrl-d`** or typing exit.

To use the `su` command on a per-command basis, you can type:

```
$ su user -c command
```

However, using `su` creates security hazards. It is potentially dangerous since it is not a good practice, for example, to have numerous people knowing and using the password of the root. Notice that when logged in as root, you can do anything in the system.

For this reason, Linux people came up with another command: `sudo`. Using the sudoers file (/etc/sudoers), system administrators can define which users or groups will be able to execute certain commands (or even any command) as root but these users will not have to know the password of the root. The `sudo` command **prompts to introduce the password of the user that is executing `sudo`** (see Figure 1.6).

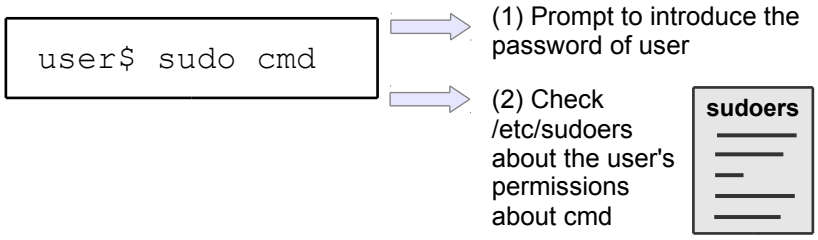


Figure 1.6: How `sudo` works.

In this way, the command `sudo` makes it easier to implement the principle of "least privilege". It also logs all commands and arguments so there is a record of who used it for what, and when. To use the `sudo` command, at the command prompt, enter:

```
$ sudo command
```

Replace command with the command for which you want to use `sudo`. If your user is configured as system administrator in the sudoers file you can get a shell as root typing:

```
user$ sudo -s
root#
```

Note. We will use $ to mean that the command is being executed as a regular user and # to mean that the command is being executed as root.

You can use the command `whoami` to know which user you are at this moment. When authenticated, a timestamp is used (stored in /var/run/sudo) and the user can use sudo without a password for 5 minutes.

1.6 Installing Software

1.6.1 Static and Dynamic Libraries

We have to understand the differences between static and dynamic libraries to fully understand the process of installing software in our Linux box (see Figure 1.7).

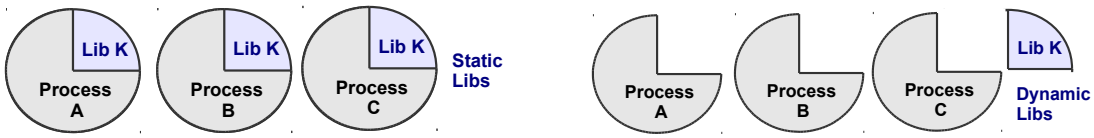


Figure 1.7: Static and dynamic libraries.

- **Static libraries**[2] or statically-linked libraries are a set of routines, external functions and variables which are resolved at compile-time and copied into the final object file by the compiler. This is called "static compiling" and program produced by this process is called a "static executable".
- With **dynamic libraries**, the kernel provides facilities for the creation and use of dynamically bound shared libraries. Dynamic binding allows external functions and variables to be referenced in user code and defined in a shared library to be resolved at run time, that is to say, when the program is loaded to become a process in the system. Therefore, the shared library code is not present in the executable image on disk. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it.

[2] In the past, libraries could only be static.

The main **advantage of static executables** is that they avoid dependency problems. Since libraries are included at compiling time, we can be sure that anything used from external libraries is available (with the correct version) before the program is executed. The main **drawback of static linking** is that the size of the executable becomes greater than in dynamic linking, as the library code is stored within the executable rather than in separate files and, which is worse, static processes in execution consume more memory than dynamically linked processes (see Figure 1.7).

On the other hand, the **main advantages of shared libraries** are that they use less disk space because the shared library code is not included in the executable programs. They also use less memory because the shared library code is only loaded once. The load time may be also reduced because the shared library code migth be already in memory. Dynamic libraries also allow the library to be updated to fix bugs and security flaws without updating the applications that use the library. The main **drawback of dynamic libraries** is that they usually establish complex relationships between the different packages of software installed in a system. For example, a configuration might enforce having several versions of the same library simultaneously installed in the system to satisfy the dependencies (required dynamic libraries) of different software packages.

For example, we are going to compile the following C program which is a typical “hello world”.

```
/* Hello World program */
#include<stdio.h>
void main()
{
  printf("Hello World\n");
}
```

We can use a text editor (like `gedit`) to save the previous text as hello.c. Then, we compile this C source file to produce an executable:

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

By default the `gcc` compiler creates dynamic executables. You can check this with the `ldd` command, which shows the dependencies of a program.

```
$ ldd hello
    linux-vdso.so.1 =>  (0x00007fff5e7ca000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fca8eca1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fca8f089000)
```

We can view the size of our executable with the `du` (disk usage) command:

```
$ du hello
12      hello
```

Compare this when we statically compile the same program:

```
$ gcc -static -o hello.static hello.c
$ ./hello.static
Hello World
$ ldd hello.static
    not a dynamic executable
$ du hello.static
860     hello.static
```

The advantages of dynamic executables are clear but the management of the different libraries on the system results in a challenge colloquially known as "dependency hell".

On Windows systems, this is called “DLL hell” (DLL comes from Dynamically Linked Library). On windows systems, it is common to distribute and install the library files that an application needs with the application itself.

On Unix-like systems, this is less common as **Package Management Systems** can be used to ensure that the correct library files are available in the system. This allows the library files to be shared between many applications leading to disk space and memory savings.

1.6.2 Sofware Packages

In Linux systems, a package of software tracks where all its files are, allowing the user to easily manage the installed software: view dependencies, uninstall, etc. Generally, in Linux, software packages copy their executables in /usr/bin, their libraries in /usr/lib and their documentation in /usr/share/doc/package/. It's worth noting here that to take full advantage of packages, one should not go installing or deleting files behind the package system's back.

There are multiple different package systems in the Linux world, the two main ones being:

- Red Hat Packages (**.rpm** files).
- Debian Packages (**.deb** files).

Packages can be managed with the commands `rpm` and `dpkg` for rpm and deb packages, respectively.

1.6.3 Advanced Package Management Systems

The package systems previously described do not manage dependencies. So, if you need to install a package that has dependencies, you have to manually install the proper versions of the packages containing the dynamic libraries on which your package depends on (at least all the libraries not currently installed on your system).

In this context, a new generation of package management systems was developed to make this tedious process easier for the user. These advanced package management systems automatically manage package dependencies. For rpm files we have `yum` and for .deb files we have `apt`. With these tools one can essentially say "install this package" and all dependent packages will be installed/upgraded as appropriate.

Of course, one has to configure where these tools must go to find out the software packages. These packages are online in **package repositories**. Thus, you have to configure the addresses of the repositories you are interested in.

In the case of Debian packages, APT uses files that lists the 'sources' from which packages can be obtained. These files are in the directory /etc/apt. APT requires to execute `apt-get update` to update the available list of packages available in online repositories. If this command is not executed, apt works with the local cache, which might be outdated.

On the other hand, in an Ubuntu system, we can type the name of an application in the console and, if it is not installed, the system will tell us how to install it:

```
$ xcowsay
The program 'xcowsay' is currently not installed.  You can install it by typing:
sudo apt-get install xcowsay
```

Then, to install the latest version of `xcowsay`, you can update first and then install:

```
$ sudo apt-get update
$ sudo apt-get install xcowsay
```

You can download (only) the required packages to install an application (in this example `gparted`):

```
$ sudo apt-get download gparted libparted-fs-resize0
$ sudo mv *.deb /var/cache/apt/archives
```

Then, when you want to install `gparted`:

```
$ sudo apt install gparted
```

The previous command does not need an online connection.

You can remove a package with:

```
$ sudo apt-get remove gparted
```

You did remove the software but typically, there are files associated with it such as configuration files and folders. To remove them:

```
$ sudo apt-get remove gparted --purge
```

1.6.4 Installing from the Source

When you need to install software that is neither in a repository or has an individual package created, you can install from the source code. This is compatible with all Linux distributions. The source code typically contains a bunch of files of the application, packed in a .tar archive and compressed using GNU Zip (.gz) or BZip2 (.bz2). Format: `<filename>.tar.gz or <filename>.tar.bz2` These types of files can be unzipped and unpacked on a directory using the `tar` command:

```
$ tar xvzf <filename>.tar.gz
$ tar xvjf <filename>.tar.bz2
```

By convention, there are files called "INSTALL" or "README" giving application-specific usage information. The typical compilation/installation steps are:

1. Unpack the tar archive (tarball):

   ```
   $ tar xzvf <package_name>.tar.gz
   $ tar xvjf <package_name>.tar.bz2
   ```

2. Change to the extracted directory

   ```
   $ cd <extracted_dir_name>
   ```

3. Run source configuration script as follows:

   ```
   $ ./configure
   ```

4. Build the source code using the GNU Make utility. In the directory of the sources there will be a Makefile file describing how to compile. You can call make (compile) as follows:

   ```
   $ make
   ```

5. Install the package as follows:

   ```
   # make install
   ```

As a final remark, we would like to mention that in general, you should avoid installing software from the source if there is a package available[3].

1.6.5 Command Summary

The table 1.1 summarizes the commands used within this section.

[3] Many times it is not too hard to make the package yourself.

Table 1.1: Summary of commands.

`lsmod`	list current system modules.
`insmod` or `modprobe`	insert a module.
`rmmod`	remove a module.
`tty`	view the associated TTY file of the terminal.
`gnome-terminal` or `xterm`	terminals.
`su`	switch to another user.
`sudo`	execute something as another user using sudoers file.
`gcc`	c compiler .
`ldd`	list dynamic libraries dependencies.
`du`	check the disk usage of files.
`dpkg`	debian packages manager.
`apt-get`	advanced package manager for debian packages.
`apt-file`	search the package that installed a certain file.

Chapter 2

Processes

2.1 Booting the System

A sequence of steps is followed to boot (start) a Linux system in which the control first goes to the BIOS, then to a boot loader and finally, to a Linux kernel (the system core). The boot loader is not absolutely necessary. Certain BIOS can load and pass control to the Linux kernel without the use of the loader but in practice this element is usually always present. Figure 2.1 illustrates the boot process.

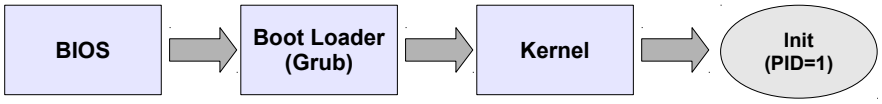


Figure 2.1: The Linux Boot Process.

Once the kernel is started, it executes the first process: *init*. A process is the abstraction used by the operating system to represent a running program. Each process in Linux consists of an address space and a set of data structures within the kernel. The address space contains the code and libraries that the process is executing, the process's variables, its stacks, and different additional information needed by the kernel while the process is running. The kernel also implements a "scheduler" to share the CPU resources available in the system among the different processes.

Linux processes have "kinship". The process that generates another process is called **parent** process. The process generated by another process is called **child** process. The processes can be parent and child both at the same time. Therefore, you can see the processes on a Linux system hierarchically organized in a tree. The root of the "tree of processes" is *init*. Finally, each process has a unique identifier called "Process ID" or PID. The PID of *init* is 1.

2.2 Listing Processes

The command `ps` provides information about the processes running on the system. If we open a terminal and type `ps`, we obtain a list of running processes. In particular, those launched from the terminal.

```
$ ps
  PID TTY          TIME CMD
21380 pts/3    00:00:00 bash
21426 pts/3    00:00:00 ps
```

Let's see what these columns mean:

- The first column is the process identifier.
- The second column is the associated terminal[1]. In the previous example is the pseudo-terminal 3. A question mark (?) in this column means that the process is not associated with a terminal.

[1]The terminal can also be viewed with the `tty` command.

- The third column shows the total amount of time that the process has been running.
- The fourth column is the name of the process.

In the example above, we see that two processes have been launched from the terminal: the `bash`, which is the shell, and the command `ps` itself. Figure 2.2 shows a scheme with the relationships of all the processes involved when we type a command in a terminal.

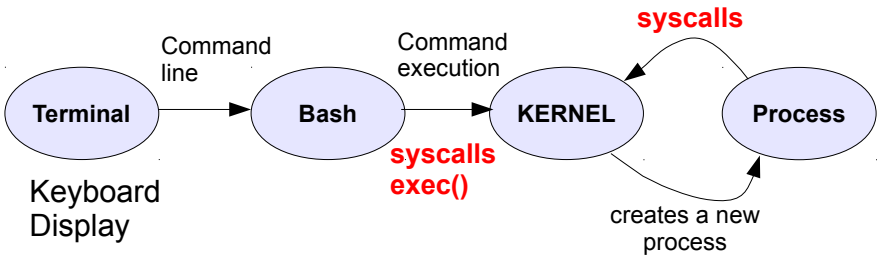


Figure 2.2: Processes related with a pseudo-terminal.

On the other hand, the command `ps` accepts parameters. For example the parameter -u reports all the processes launched by a user. Therefore, if you type:

```
$ ps -u user1
```

You will obtain a list of all the processes owned by `user1`. Some relevant parameters of `ps` are:

- ***-A*** shows all the processes from all the users.
- ***-u user*** shows processes of a particular user.
- ***-f*** shows extended information.
- ***-o format*** the format is a list separated by commas of fields to be displayed.

Examples:

```
$ ps -Ao  pid,ppid,state,tname,%cpu,%mem,time,cmd
$ ps -u user1 -o pid,ppid,cmd
```

The preceding command shows the process PID, the PID of parent process (PPID), the state of the process, the associated terminal, the % of CPU and memory consumed by the process, the accumulated CPU time consumed and the command that was used to launch the process.

2.3 The `man` Command

The `man` command shows the "manual" of other commands. Example:

```
$ man ps
```

If you type this, you will get the manual for the command "`ps`". Once in the help environment, you can use the arrow keys or `AvPag`/`RePaq` to go up and down. To search for text *xxx*, you can type `/xxx`. Then, to go to the next and previous matches you can press keys `n` and `p` respectively. Finally, you can use `q` to exit the manual.

2.4 Working with the Terminal

Bash keeps a history the commands typed. The history can be seen with the command `history`. You can retype a command of the history using `!number`. You can also press the up/down arrow to scroll back and forth through your command history.

On the other hand, the bash provides another useful feature called command line completion (also called tab completion). This is a common feature of bash and other command line interpreters. When pressing the `TAB`, bash automatically fills in partially typed commands or parameters.

Finally, X servers provide a "copy and paste" mechanism. You can easily copy and paste between pseudo-terminals and applications using your mouse. Most Linux distributions are configured in such a way that a click with the mouse's middle button (or scroll wheel) is interpreted as a paste operation of the selected text. If your mouse has only two buttons, hit both buttons simultaneously to emulate the middle button. Typically, the combinations `CRL+SHIFT+c` and `CRL+SHIFT+v` also work for copy and paste.

2.5 Other Commands for Processes

The **`pstree`** command displays all the system processes within a tree showing the relationships between processes. The root of the tree is either init or the process with the given PID.

The **`top`** command returns a list of processes in a similar way as `ps` does, except that the information displayed is updated periodically so we can see the evolution of a process' state. **`top`** also shows additional information such as memory space occupied by the processes, the memory space occupied by the exchange partition or swap, the total number of tasks or processes currently running, the number of users, the percentage processor usage, etc.

Finally, the **`time`** command gives us the duration of execution of a particular command. Example:

```
$ time ps
  PID TTY          TIME CMD
 7333 pts/0    00:00:00 bash
 8037 pts/0    00:00:00 ps

real    0m0.025s
user    0m0.016s
sys     0m0.012s
```

Real refers to actual elapsed time; User and Sys refer to CPU time used only by the process.

- Real is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- User is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- Sys is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process.

Notice that User+Sys will tell you how much actual CPU time your process used. This is across all CPUs, so if the process has multiple threads it could potentially exceed the wall clock time reported by Real.

2.6 Scripts

In our system we will have programs. The source of a program is first compiled, and the result of that compilation is executed to become a process in the system. The process directly interacts with the system kernel. Examples of languages to build programs: C, C++, etc. On the other hand, a script is interpreted. In other words, a script is written to be understood by an interpreter (see Figure 2.3). Scripting examples are shell scripts, Python scripts, PHP scripts, scripts for Javascript, etc. Typically scripts are written for small applications and they are easier to develop. However, scripts are also usually slower than programs due to the interpretation process. In this case, the interpreter is who finally interacts with the system kernel.

In particular, a shell script is a text file containing commands and special internal shell commands (if,for, while, etc.) that have to be interpreted. As its mane suggests, a shell (bash in most Linux systems) is who interprets and executes a shell script.

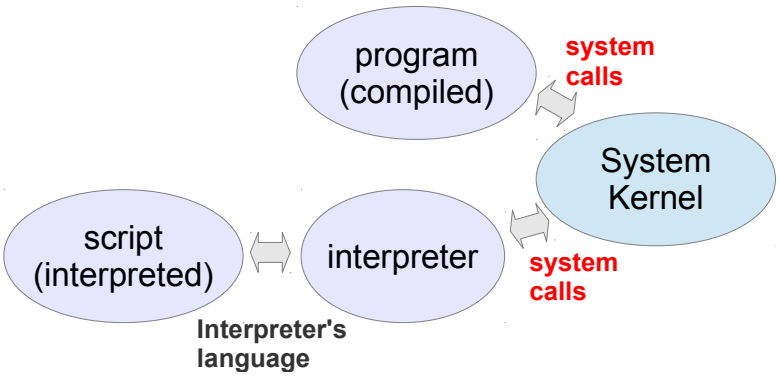


Figure 2.3: Program versus Script.

Why to write shell scripts? They are useful to create your own commands, they save your time, some tasks of your daily life can be automated and system administration can be automated too.

You can use any text editor like `vi` (CLI) or `gedit` (GUI) to write down a script. Then, you must give it execution permissions to be able to execute it. A simple example is:

```
1 ps
2 sleep 2
3 pstree
```

Then, we give execution permissions to our user and execute it:

```
$ chmod u+x firstscript.sh
$ ./firstscript.sh
  PID TTY          TIME CMD
 5936 pts/2    00:00:00 bash
 5996 pts/2    00:00:00 bash
 5997 pts/2    00:00:00 ps
init -- /usr/bin/x-term -- /usr/bin/x-term
     |                   | --bash--bash--pstree
    ...
```

The previous script command executes a `ps` and then, after approximately 2 seconds (sleep makes us wait 2 seconds) a `pstree` command. You can notice from the output of the previous script that the `bash` clones itself to execute the commands within the script. We can say that the "script PID" is the PID of the cloned (child) `bash`. This is a way of protecting the parent bash in case anything in the script goes wrong. As we will see next, we can kill processes using their PID. Another example script is the classical "Hello world" script.

```
1 #!/bin/bash
2 # Our second script, Hello world!
3 echo Hello world
```

As you can observe, the script begins with a line that starts with "#!". This is the path to the shell that will execute the script. In general, you can ignore this line (as we did in our previous example) and then the script is executed by the default shell. However, it is considered a good practice to always include it in scripts. In our case, we will build scripts always for the `bash` shell.

As you can observe, we can use the `echo` command to write to the terminal. Finally, you can also observe that text lines (except the first one) after the sign "#" are interpreted as comments.

Next, we assign execution permission and execute the script:

```
$ chmod u+x secondscript.sh
$./secondscript.sh
Hello world
```

Finally, in a script you can also read text typed by the user in the terminal. To do so, you can use the `read` command. For example, try:

```
#!/bin/bash
# Our third script, using read for fun
echo Please, type a sentence and hit ENTER
read TEXT
echo You typed: $TEXT
```

In this script we can observe some new things. The text introduced by the user in the terminal is stored in a variable called "TEXT" by the `read` command. When we want to use the content (value) of a variable, we must set the sign "$" before the name of the variable. Notice that we use the value of the variable TEXT in the `echo` command in this script.

2.7 Running Processes in Foreground/Background

By default, the `bash` executes commands interactively, i.e., the user enters a command, the `bash` marks it to be executed, waits for the output and once concluded returns the control to the user, so a new command can be executed. This type of execution is also called a **foreground command execution**. For example:

```
$ xeyes
```

This application prints eyes that follow the mouse movements. While running a command in foreground we cannot run any other commands. However, `bash` also let us run commands non-interactively or in **background**. To do so, we must use the ampersand symbol (&) at the end of the command line. Example:

```
$ xeyes &
```

Whether a process is running in foreground or background, its output goes to its attached terminal. However, a process cannot use the input of its attached terminal while running in background.

Note. We can try this with our thirdscript.sh.

2.8 Signals

A signal is a limited form of inter-process communication using the system kernel and the `kill()` syscall. Actually, a signal is just an integer. As you can observe in Figure 2.4, some signals are destined for the kernel (non-capturable signals) and other signals are destined for user-space processes (capturable signals).

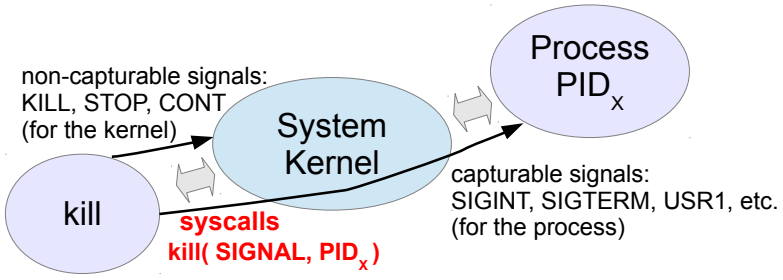


Figure 2.4: `Kill` command and `kill()` syscall.

Capturable signals are used as an asynchronous notification: when a signal is received by a user-space process, it interrupts its normal flow of execution and executes the corresponding "signal handler" (function) based on the signal number. The `kill` command can be used to send signals [2].

[2] More technically, the command `kill` is a *wrapper* around the system call `kill()`, which can send signals to processes or groups of processes in the system, referenced by their process IDs (PIDs) or process group IDs (PGIDs).

The default signal that the `kill` command sends is the termination signal (`SIGTERM`), which asks the process for releasing its resources and exit. The integer and name of signals may change between different implementations of Unix. Usually, the `SIGKILL` signal is the number 9 and `SIGTERM` is 15. You can view the signals defined in your system with `kill -l`. In Linux, the most widely used signals and their corresponding integers are:

- 9 `SIGKILL`. Non-capturable signal sent to the kernel to end a process immediately.
- 20 `SIGSTOP`. Non-capturable signal sent to the kernel to stop a process. This signal can be generated in a terminal for a process in foreground pressing **`Control-Z`**.
- 18 `SIGCONT`. Non-capturable signal sent to the kernel that resumes a previously stopped process. This signal can be generated typing `bg` in a terminal.
- 2 `SIGINT`. Capturable signal sent to a process to tell it that it must terminate its execution. It is sent in an interactive terminal for the process in foreground when the user presses **`Control-C`**.
- 15 `SIGTERM`. Capturable signal sent to a process to ask for termination. It is sent from the GUI and also this is the default signal sent by the `kill` command.
- `USR1`. Capturable signal that can be used for any desired purpose.

The `kill` command syntax is: **`kill -signal PID`**. You can use both the number and the name of the signal:

```
$ kill -9 30497
$ kill -SIGKILL 30497
```

In general, signals can be intercepted by processes, that is, processes can provide a special treatment for the received signal. However, SIGKILL, SIGSTOP and SIGCONT cannot be captured. They are destined to the kernel, which kills, stops or resumes the process. This provides a safe mechanism to control the execution of processes. SIGKILL ends the process and SIGSTOP pauses it until a SIGCONT is received.

Unix has also security mechanisms to prevent an unauthorized users from finalizing other user processes. Basically, a process cannot send a signal to another process if both processes do not belong to the same user. Obviously, the exception is the user *root* (superuser), who can send signals to any process in the system.

Finally, another interesting command is **`killall`**. This command is used to terminate execution of processes by name. This is useful when you have multiple instances of a running program.

2.9 Job Control

Job control refers to the bash feature of managing processes as jobs. For this purpose, bash provides the commands **`"jobs"`**, **`"fg"`**, **`"bg"`** and the hot keys **`Control-z`** and **`Control-c`**.

The command `jobs` displays a list of processes launched from a specific instance of bash. Each instance of bash considers that it has launched processes as *jobs*. Each job is assigned an identifier called a JID (Job Identifier).

Let's see how the job control works using some examples and a couple of graphic applications: `xeyes` and `xclock`. The application `xeyes` shows eyes that follow the mouse movement, while `xclock` displays a clock in the screen.

```
$ xeyes &
$ xeyes &
$ xclock &
$
```

The previous commands run 2 processes or instances of the application `xeyes` and an instance of `xclock`. To see their JIDs, you can type `jobs`.

```
$ jobs
[1]   Running                 xeyes &
[2]-  Running                 xeyes &
[3]+  Running                 xclock &
$
```

In this case we can observe that the JIDs are 1, 2 and 3 respectively. Using the JID, we can make a job to run in *foreground*. The command is `fg`. The following command brings to *foreground* job 2.

```
$ fg 2
xeyes
```

On the other hand, the combination of keys **Control-z** sends a stop signal (SIGSTOP) to the process that is running on *foreground*. Following the example above, we had a *job* `xeyes` in the foreground, so if you type **`Control-z`** the process will be stopped.

```
$ fg 2
xeyes
^Z
[2]+  Stopped                 xeyes
$ jobs
[1]   Running                 xeyes &
[2]-  Running                 xclock &
[3]+  Stopped                 xeyes
$
```

To resume the process that we just stopped, type the command `bg`:

```
$ bg
[2]+ xeyes &
$
```

Using the JID after the command `bg` will send the corresponding job to *background* (in the previous case we can use `bg 2` to produce the same result). The JID can also be used with the command `kill`. To do this, we must write a % sign right before the JID to differentiate it from a PID. For example, we could terminate the job "1" using the command:

```
 $ kill -s SIGTERM %1
```

Another very common shortcut is **Control-c** and it is used to send a signal (SIGINT) to terminate the process that is running on *foreground*. Example:

```
$ fg 3
xclock
^C
[1]   Terminated              xeyes
```

Notice that whenever a new process is run in *background*, the bash provides us the JID and the PID:

```
$ xeyes &
[1] 25647
```

Here, the job has JID=1 and PID=25647.

2.10 Capturing Signals in Scripts: `trap`

The `trap` command allows the user to catch signals in bash scripts. If we use this script:

```
trap "echo I do not want to finish!!!!" SIGINT
while true
do
sleep 1
done
```

Try to press **`Control-c`**.

2.11 Running Multiple Commands

If a command is successfully executed it returns a "0". If an error occurred, the command must return a positive value that provides some information about the error (many times it is just a "1"). The "return code" of the previously executed command is stored in the Bash in a special variable called "?". To view the content of this variable you can type `echo $?`.

```
$ ps -k
$ echo $?
$ ps
$ echo $?
```

Using return codes, we can also use different ways of executing commands:

- **`$ command`** the command runs in the foreground.
- **`$ command1 & command2 & ... commandN &`** commands will run in background.
- **`$ command1; command2; ... ; commandN`** sequential execution of commands.
- **`$ command1 && command2 && ... && commandN`** `commandX` is executed if the last executed command has exit successfully (return code 0).
- **`$ command1 || command2 || ... || commandN`** `commandX` is executed if the last executed command has NOT exit successfully (return code >0).

2.12 Extra

2.12.1 *Threads

Threads are a popular programming abstraction for parallel execution on modern operating systems. When threads are forked inside a program for multiple flows of execution, these threads share certain resources (e.g., memory address space, open files) among themselves to minimize forking overhead and avoid expensive inter-process communication (IPC) channel. These properties make threads an efficient mechanism for concurrent execution.

In Linux, threads, also called Lightweight Processes (LWP), created within a program, will have the same "thread group ID" as the program's PID. Each thread will then have its own thread ID (TID). To the Linux kernel's scheduler, threads are nothing more than standard processes which happen to share certain resources. Classic command-line tools such as `ps` or `top`, which display process-level information by default, can be instructed to display thread-level information. Example:

```
$ ps -T -p 2741
 PID  SPID TTY          TIME CMD
2741  2741 ?        00:00:01 nm-applet
2741  2754 ?        00:00:00 dconf worker
2741  2760 ?        00:00:00 gdbus
```

2.12.2 *Terminal Associated Signals

A shell process is a child of a terminal and when we execute a command, the command becomes a child of the shell. If the terminal is killed or terminated (without typing exit), a SIGHUP signal (hang up) sent to all the processes using the terminal (i.e. bash and currently running commands).

```
$ tty
/dev/pts/1
$ echo $PPID
11587
$ xeyes &
```

```
[1] 11646
$ ps
  PID TTY          TIME CMD
11592 pts/1    00:00:00 bash
11646 pts/1    00:00:02 xeyes
11706 pts/1    00:00:00 ps
```

If you close the terminal from the GUI, or if you type the following from another terminal:

```
$ kill -TERM 11587
```

You will observe that the `xeyes` process also dies. However, if you type exit in a shell with a process in background, you will notice that the process does not die but it becomes *orphan* and `Init` is assigned as its new parent process. There is a shell utility called `nohup`, which can be used as a wrapper to start a program and make it immune to SIGHUP. Also the output is stored in a file called nohup.out in the directory from which we invoked the command or application. Try to run the previous example with `$ nohup xeyes &`.

On the other hand, if there is an attempt to read from a process that is in background, the process will receive a SIGTTIN signal. If not captured, this signal suspends the process. Finally, if you try to exit a bash while there are stopped jobs, it will alert us. Then the command **`jobs`** can be used to inspect the state of all those jobs. If `exit` is typed again, the warning message is no longer shown and all suspended tasks are terminated.

2.12.3 *States of a Process

A process might be in several states:

- Ready (R) - A process is ready to run. Just waiting for receiving CPU cycles.
- Running (O) - Only one of the ready processes may be running at a time (for uniprocessor machine).
- Suspended (S) - A process is suspended if it is waiting for something to happen, for instance, if it is waiting for a signal from hardware. When the reason for suspension goes away, the process becomes Ready.
- Stopped (T) - A stopped process is also outside the allocation of CPU, but not because it is suspended waiting for some event.
- Zombie (Z) - A zombie process or *defunct* is a process that has completed execution but still has an entry in the process table. Entries in the process table allow a parent to end its children correctly. Usually the parent receives a SIGCHLD signal indicating that one of its children has died. Zombies running for too long may point out a problem in the parent source code, since the parent is not correctly finishing its children.

Note. A zombie process is not like an “orphan” process. An orphan process is a process that has lost its father during its execution. When processes are “ orphaned”, they are adopted by “ *Init*.”

2.12.4 *Priorities: `nice`

Each Unix process has a priority level ranging from -20 (highest priority) to 19 (lowest priority). A low priority means that the process will run more slowly or that the process will receive less CPU cycles from the kernel scheduler.

The `top` command can be used to easily change the priority of running processes. To do this, press ”r“ and enter the PID of the process that you want change it’s priority. Then, type the new level of priority. We must take into consideration that only the superuser ”root“ can assign negative priority values.

You can also use `nice` and `renice` instead of `top`. Examples.

```
$ nice -n 2 xeyes &
[1] 30497
$ nice -n -2 xeyes
nice: cannot set niceness: Permission denied
$ renice -n 8 30497
30497: old priority 2, new priority 8
```

2.12.5 Command Summary

The table 2.1 summarizes the commands used within this section.

Table 2.1: Summary of commands for process management.

man	is the system manual page.
ps	displays information about a selection of the active processes.
tty	view the associated terminal.
pstree	shows running processes as a tree.
top	provides a dynamic real-time view of running processes.
time	provides us with the duration of execution of a particular command.
sleep	do not participate in CPU scheduling for a certain amount of time.
echo	write to the terminal.
read	read from the terminal.
jobs	command to see the jobs launched from a shell.
bg and fg	command to set a process in background or foreground.
kill	command to send signals.
killall	command to send signals by name.
nice and renice	command to adjust niceness, which modifies CPU scheduling.
trap	process a signal.
nohup	command to create a process which is independent from the father process.

2.13 Practices

Exercise 2.1– In this exercise you will practice with process execution and signals.

1. Open a pseudo-terminal and execute the command to see the manual of `ps`. Once in the manual of the `ps` command, search and count the number of times that appears the pattern *ppid*.

2. Within the same pseudo-terminal, execute `ps` with the appropriate parameters in order to show the PID, the *tty* and the command of the currently active processes that have been executed from the terminal. Do the same in the virtual console number two (/dev/tty2).

3. Execute the following commands:

```
$ ps -o pid,comm
$ ps -o pid,cmd
```

   Comment the differences between the options: *cmd* and *comm*.

4. Use the `pstree` command to see the process tree of the system. Which process is the father of `pstree`? and its grandfather? and who are the rest of its relatives?

5. Open a gnome-terminal and then open a new "TAB" typing CRL+SHIFT+t. Now open another gnome-terminal in a new window. Using `pstree`, you have to comment the relationships between the processes related to the terminals that we opened.

6. Type ALT+F2 and then type `xterm`. Notice that this sequence opens another type of terminal. Repeat the same sequence to open a new `xterm`. Now, view the process tree and comment the differences with respect to the results of the previous case of gnome terminals.

7. Open three gnome-terminals. These will be noted as t1, t2 and t3. Then, type the following:

```
t1$ xeyes -geometry 200x200 -center red
t2$ xclock &
```

   Comment what you see and also which is the type of execution (foreground/background) on each terminal.

8. For each process of the previous applications (`xeyes` and `xclock`), try to find out the PID, the execution state, the *tty* and the parent PID (PPID). To do so, use the third terminal (t3).

9. Using the third terminal (t3), send a signal to terminate the process xeyes.

10. Type `exit` in the terminal t2. After that, find out who is the parent process of `xclock`.

11. Now send a signal to kill the process `xclock` using its PID.

12. Execute an `xclock` in *foreground* in the first terminal t1.

13. Send a signal from the third terminal to stop the process `xclock` and then send another signal to let this process to continue executing. Is the process executing in *foreground* or in *background*? Finally, send a signal to terminate the `xclock` process.

14. Using the job control, repeat the same steps as before, that is, executing `xclock` in *foreground* and then stopping, resuming and killing. List the commands and the key combinations you have used.

15. Execute the following commands in a pseudo-terminal:

```
$ xclock &
$ xclock &
$ xeyes &
$ xeyes &
```

    Using the job control set the first `xclock` in *foreground*. Then place it back in *background*. Kill by name the two `xclock` processes and then the `xeyes` processes. List the commands and the key combinations you have used.

16. Create a command line using execution of multiple commands that shows the processes launched from the terminal, then waits for 3 seconds and finally shows again the processes launched from the terminal.

17. Using multiple commands execution (&&, ||, etc.) create a command line that executes a `ps` command with an unsuccessful exit state and then as a result another `ps` command without parameters is executed.

18. Discuss the results of the following multiple command executions:

```
$ sleep || sleep || ls
$ sleep && sleep --help || ls && ps
$ sleep && sleep --help || ls || ps
```

Exercise 2.2– (*) This exercise deals with additional aspects about processes.

1. Create a script that asks for a number and displays the number multiplied by 7. Note. If you use the variable VAR to read, you can use `$[VAR * 7]` to display its multiplication.

2. Add signal managment to the previous script so that when the USR1 signal is received, the script prints the sentence "waiting operand".

    Tips: use `trap` **to capture USR1 and** `kill -USR1 PID` **to send this signal.**

3. Type a command to execute an `xeyes` application in background with "niceness" (priority) equal to 18. Then, type a command to view the command, the PID and the priority of the `xeyes` process that you have just executed.

Chapter 3

Filesystem

3.1 Introduction

File Systems (FS) define how information is stored in data units like HDD, DVDs, USB devices, tapes, etc. The base of a FS is the file. There's a saying in the Linux world that "everything is a file" (a comment attributed to Ken Thompson, one of the developers of UNIX). That includes directories.

Directories are just files that contain a list of other files. Files and directories are organized into a hierarchical file system, starting from the root directory / and branching out. On the other hand, files must have a name, which is tight to the following rules:

- Must be between 1 and 255 characters
- All characters can be used except for the slash "/".
- The following characters can be used in names of files but they are not recommended because they have special meanings:
  `= \ ^ ~ ' " ` * ; - ? [ ] ( ) ! & ~ < >`
- The file names are case sentive, that is to say, characters in uppercase and lowercase are distinct. For example: `letter.txt`, `Letter.txt` or `letter.Txt` do not represent the same files.

The simplest example of file is that used to store data like text, images, etc. The different FS technologies are implemented in the kernel either statically or as modules. FS define how the kernel manages files including aspects such as which meta-data is used for files, when and how a file is read or written, etc.

Examples of Disk File Systems (DFS) are reiserFS, ext2, ext3 and ext4. These are developed within the Unix environment for HDD and USB devices. For DVDs we have a file system called UDF (universal disk format). In Windows environments we have other DFS like fat16, fat32 and ntfs.

3.2 Basic types of files

Unix uses the abstraction of "file" for many purposes and thus, as mentioned, this is a fundamental concept in Unix systems. This type of abstraction allows using the API of files for devices like for example a printer. In this case, the API of files is used for writing into the file that represents the printer, which indeed means printing. Unix kernels manage three basic types of files:

- **Regular files**. These files contain data.
- **Directory files (folders)**. These files contain a list of other files. Directories are used to group other files in an structured manner.
- **Special Files**. Within this category there are several sorts of files which have some special content used by the OS.

The command `stat` can be used to view the basic type and some metadata of a file.

```
$ stat /etc/services
  File: `/etc/services'
  Size: 19281           Blocks: 40         IO Block: 4096   regular file
Device: 801h/2049d      Inode: 3932364     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2012-09-24 22:06:53.249357692 +0200
Modify: 2012-02-13 19:33:04.000000000 +0100
Change: 2012-05-11 12:03:24.782168392 +0200
 Birth: -
```

Everything is a file in Unix sytems. For example, the TTY file is a special file:

```
$ tty
/dev/pts/2
$ stat /dev/pts/2
  File: `/dev/pts/2'
  Size: 0               Blocks: 0          IO Block: 1024   character special file
Device: bh/11d  Inode: 5           Links: 1     Device type: 88,2
Access: (0620/crw--w----)  Uid: ( 1000/ jlmunoz)   Gid: (    5/     tty)
Access: 2014-03-25 18:11:04.152159417 +0100
Modify: 2014-03-25 18:11:04.152159417 +0100
Change: 2014-03-25 18:10:54.152159417 +0100
 Birth: -
```

3.3 Hierarchical File Systems

The "Linux File Tree" follows the FHS (Filesystem Hierarchy Standard). This standard defines the main directories and their content for GNU/Linux OS and other Unix-alike OS. In contrast to Windows variants, the Linux File Tree is not tied up to the hardware structure. Linux does not depend on the number of hard disks the system has (c:\, d:\ or m:\...). The whole Unix file system has a unique origin: the root (*/*). Below this directory we can find all files that the OS can access to. Some of the most significant directories in Linux (FHS) are detailed next:

/ File system root.

/dev Contains system files which represent devices physically connected to the computer.

/etc Contains system configuration files.
This directory cannot contain any binary files (such as programs or commands).

/lib Contains necessary libraries to run programs or commands.

/proc Contains special files which receive or send information to the kernel.
If necessary, it is recommended to modify these files with "special caution".

/bin Contains binaries of common system commands.

/sbin Contains binaries of administration commands which can only be executed by the system admin or superuser (*root*).

/usr This directory contains the common programs that can be used by all the system users.
The structure is the following:

- /usr/bin General purpose programs (including C/C++ compiler).
- /usr/doc System documentation.
- /usr/etc Configuration files of user programs.
- /usr/include C/C++ heading files (.h).
- /usr/info GNU information files.
- /usr/lib Libraries of user programs.
- /usr/man Manuals to be accessed by command *man*.
- /usr/sbin System administration programs.
- /usr/src Source code of those programs.

Additionally other directories may appear within */usr*, such as directories of installed programs.

/var Contains temporal data of programs (this doesn't mean that the contents of this directory can be erased).

/mnt or /media Contains mounted systems of pendrives or external disks.

/home Contains the working directories of the users of the system except for root.

3.4 Storage Devices

In UNIX systems, the kernel automatically detects and maps storage devices in the /dev directory. The name that identifies a storage device follows the following rules:

1. If there is an IDE controller:
   - `hda` to IDE bus/connector 0 master device
   - `hdb` to IDE bus/connector 0 slave device
   - `hdc` to IDE bus/connector 1 master device
   - `hdd` to IDE bus/connector 1 slave device

   For example, if a CD-ROM or DVD is plugged to IDE bus/connector 1 master device, Linux will show it as `hdc`. In addition, each hard drive can have up to 4 primary partitions (limit of PC x86 architecture) and each primary partition can also have secondary partitions. Each particular partition is identified with a number:
   - First partition: /dev/hda1
   - Second partition: /dev/hda2
   - Third partition: /dev/hda3
   - Fourth partition: /dev/hda4
2. If there is a SCSI or SATA controller these devices are listed as devices sda, sdb, sdc, sdd, sde, sdf, and sdg in the /dev directory. Similarly, partitions on these disks can range from 1 to 16 and are also in the /dev directory.

When a Linux/UNIX system boots, the Kernel requires a "mounted root filesystem". In the most simplest case, which is when the system has only one storage device, the Kernel has to identify which is the device that contains the filesystem (the root / and all its subdirectories) and then, the Kernel has to make this device "usable". In UNIX, this is called "mounting the filesystem".

For example, if we have a single SATA disk with a single partition, it will be named as /dev/sda1. In this case, we say that the device "/dev/sda1" mounts "/". The root of the filesystem "/" is called the mount point. The file /etc/fstab contains the list of devices and their corresponding mount points that are going to be used by the system.

3.5 Disk Usage

A couple of useful commands for viewing disk capacities and usage are `df` and `du`. The command `df` (abbreviation for disk free) is used to display the amount of available disk space of file systems:

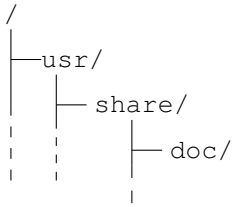
```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             108G   41G   61G  41% /
none                  1,5G  728K  1,5G   1% /dev
none                  1,5G  6,1M  1,5G   1% /dev/shm
none                  1,5G  116K  1,5G   1% /var/run
none                  1,5G     0  1,5G   0% /var/lock
```

`du` (abbreviated from disk usage) is a used to estimate file space used under a particular directory or by certain files on a file system:

```
$ du -sh /etc/apache2/
464K    /etc/apache2/
```

3.6 The path

In Unix-like systems the filesystem has a root denoted as **/** (do not get confused with the *root* user). All the files on the system are named taking the FS root as reference. In general, the *path* defines how to reach a file in the FS. For instance, */usr/share/doc* points out that the file *doc* (which is a directory) is inside the directory *share* which is inside the directory *usr*, which is under the FS root (*/*).



We have three basic commands that let us move around the FS and list its contents:

- The command `ls` (list) lists the files on the current directory.
- The command `cd` (change directory) allows us to change from one directory to another.
- The command `pwd` (print current working) prints the current directory.

The directories contain two special names:

. (a dot) which represents the current directory.

.. (two dots) which represent the parent directory.

With commands related to the filesystem you can use absolute and relative names for the files:

- **Absolute path.** An absolute path always takes the root **/** of the filesystem as starting point. Thus, we need to provide the full path from the root to the file. Example: `/usr/local/bin`.
- **Relative path.** A relative path provides the name of a file taking the current working directory as starting point. For relative paths we can use **.** (the dot) and **..** (the two dots). Examples:

  `./Desktop` or for short `Desktop` (the ./ can be omitted). This is names a file called Desktop inside the current directory.

  `./../../etc` or for short `../../etc`. This names the file (directory) etc, which is located two directories up in the FS.

Finally, the special character ~ (**`ALT GR+4`**) can used as the name of your "home directory" (typically `/home/username`). Recall that your home directory is the area of the FS in which you can store your files.

Examples:

```
$ ls /usr
bin  games  include  lib  local  sbin  share  src
$ cd /
$ pwd
/
$ cd /usr/local
$ pwd
  /usr/local
$ cd bin
$ pwd
  /usr/local/bin
$ cd /usr
$ cd ./local/bin
$ pwd
  /usr/local/bin
$ cd ../../share/doc
$ pwd
  /usr/share/doc
$ ls ~
Downloads  Videos Desktop  Music
```

3.7 Directories

In a FS, files and directories can be created, deleted, moved and copied. To create a directory, we can use `mkdir`:

```
$ cd ~
$ mkdir myfolder
```

This will create a directory called "myfolder" inside your working directory (e.g. /home/user). If we want to delete a directory, we can use `rmdir`:

```
$ rmdir ~/myfolder
```

The previous command will fail if the folder is not empty (contains some other file or directory). There are two ways to proceed: delete the content and then the directory itself or force a recursive removal with the command `rm` and the options -r (recursive) and -f (force). Example:

```
$ rm -rf myfolder
```

This is analogous to:

```
$ rm -f -r myfolder
```

The command `mv` (move) can be used to move a directory to another location:

```
$ mkdir folder1
$ mkdir folder2
$ mv folder2 folder1
```

You can also use the command `mv` to rename a directory:

```
$ mv folder1 directory1
```

Finally, to copy the contents of a directory to other place in the file system we can use the `cp` command with the option -r (recursive). Example:

```
$ cd directory1
$ mkdir folder3
$ cd ..
$ cp -r directory1 directory2
```

3.8 Files

The easiest way to create a file is using `touch`:

```
$ touch test.txt
```

This creates an empty file called `test.txt`. To remove this file, the `rm` command can be used:

```
$ rm test.txt
```

Obviously, if a file which is not in the working directory has to be removed, the complete path must be the argument of `rm`:

```
$ rm /home/user1/test.txt
```

In order to move or rename files we can use the `mv` command. For example, moving the file test.txt to the Desktop directory on your home might look like this:

```
$ mv test.txt ~/Desktop/
```

In case a name is specified in the destination, the resulting file will be renamed:

```
$ mv test.txt Desktop/test2.txt
```

Renaming a file can be done also with `mv`:

```
$ mv test.txt test2.txt
```

The copy command works similar to `mv` but the origin will not disappear after the copy operation:

```
$ cp test.txt test2.txt
```

Finally, we have hidden files. Ah hidden file is any file that begins with a ".". When a file is hidden it can not been seen with the bare `ls` command or an un-configured file manager. In most cases you won't need to see those hidden files as much of them are configuration files/directories for your desktop. There are times, however, that you will need to see them in order to edit them or even navigate through the directory structure. To do this you will need to use the option -a with `ls`:

```
$ ls -a
```

3.9 PATH variable

You may wonder how the system knows the path to commands because normally we do not type any relative or absolute path to the command but just the command name. The response is that the system utilizes the environment variable PATH. You can check the contents of PATH as:

```
$ echo $PATH
```

3.10 File content

It is usual to end the names of files with an extension, which is a dot followed by some characters. The extension is used to provide some hint about the content of a file. Examples: .txt for text files , .jpg or .jpeg for jpeg images, .htm or .html for HTML documents, etc. However, in Unix systems, the file extension is optional. If a file does not have a extension and you want to know which is its content, GNU/Linux provides you with a command that uses a guessing mechanism called *magic numbers*. The command is `file`. Example:

```
$ file /etc/services
/etc/services: ASCII English text
```

3.11 File expansions and quoting

Bash provides us with some special characters that can be used to name groups of files. This special characters have a special behavior called "filename expansion" when used as names of files. We have several filename expansions:

Character	**Meaning**
?	Expands one character.
*	Expands zero or more characters (any character).
`[ ]`	Expands one of the characters inside [ ].
!( )	Expands the file expansion **not** inside ( ).

Examples:

```
$ cp ~/* /tmp            # Copies all the files in your home to /tmp
$ cp -r * ~              # Copies everything recursively in the current directory
                         # to your home
$ rm ~/hello?            # Removes files in your home called "hello0" or
                         # "hellou" but not "hello" or "hellokitty".
$ cp ~/[Hh]ello.c /tmp   # Copies Hello.c and hello.c from your home (if they exist)
                         # to /tmp.
$ rm -r !(*.jpg)         # Deletes everything from the current directory recursively
                         # except files in the form *.jpg
```

These special characters for filename expansions can be disabled with quoting:

Character	**Action**
' (simple quote)	All characters between simple quotes are interpreted without any special meaning.
" (double quotes)	Special characters are ignored except `$`, `` ` `` and `\`
\ (backslash)	The special meaning of the character that follows is ignored.

Example:

```
$ rm "hello?"  # Removes a file called hello?
               # (but not a file called hello1).
```

3.12 Text Files

A text file typically contains human readable characters such as letters, numbers, punctuation, and also control characters such as tabs, line breaks, carrier returns, etc. The simplicity of text files allows a large amount of programs to read and modify it. Text files contain bytes representing characters that must read using a character encoding table or charset. The most well known character encoding table is the ASCII table. The ASCII table defines control and printable characters. The original specification of the ASCII table defined only 7 bits. Examples of 7-bit ASCII codification are:

```
a: 110 0001 (97d) (Ox61)
A: 100 0001 (65d) (Ox41)
```

Later, the ASCII table was expanded to 8 bits (a byte). Examples of 8-bit ASCII codification are:

```
a: 0110 0001
A: 0100 0001
```

As you may observe, to build the 8-bit codification, the 7-bit codification was maintained just setting a 0 before the 7-bit word. For those words whose codification started with 1, several specific encodings per language appeared. These codifications were defined in the ISO/IEC 8859 standard. This standard defines several 8-bit character encodings. The series of standards consists of numbered parts, such as ISO/IEC 8859-1, ISO/IEC 8859-2, etc. There are 15 parts. For instance, ISO/IEC 8859-1 is for Latin languages and includes Spanish and ISO/IEC 8859-7 is for Latin/Greek alphabet. An example of 8859-1 codification is the following:

```
ç (ASCII): 1110 0111 (231d) (Oxe7)
```

Nowadays, we have other types of encodings. The most remarkable is UTF-8 (UCS Transformation Format 8-bits), which is the default text encoding used in Linux. UFT-8 defines a variable length universal character encoding. In UTF-8 characters range from one byte to four bytes. UTF-8 matches up for the first 7 bits of the ASCII table, and then is able to encode up to 2^{31} characters unambiguously (universally). Example:

```
ç (UTF8): Oxc3a7
```

Finally, in a text file we must define how to mark a **new line**. A new line, line break or end-of-line (EOL) is a special character or sequence of characters signifying the end of a line of text. In ASCII (or compatible charsets) the characters LF (Line feed, "`\n`", 0x0A, 10 in decimal) and CR (Carriage return, "`\r`", 0x0D, 13 in decimal) are reserved to mark the end of a text line. The actual codes representing a newline vary across operating systems:

- **CR+LF**: Microsoft Windows, DEC TOPS-10, RT-11 and most other early non-Unix and non-IBM OSes, CP/M, MP/M, DOS (MS-DOS, PC-DOS, etc.), Atari TOS, OS/2, Symbian OS, Palm OS.
- LF+CR: Acorn BBC spooled text output.
- CR:Commodore 8-bit machines, Acorn BBC, TRS-80, Apple II family, Mac OS up to version 9 and OS-9.
- **LF**: Multics, Unix and Unix-like systems (GNU/Linux, AIX, Xenix, Mac OS X, FreeBSD, etc.), BeOS, Amiga, RISC OS, Android and others.

The different codifications for the newline can be a problem when exchanging data between systems with different representations. If for example you open a text file from a windows-like system inside a unix-like system you will need to either convert the newline encoding or use a text editor able of detecting the different formats (like `gedit`).

Note. For text transmission, the standard representation of newlines is CR+LF.

3.13 Commands and Applications for Text

Many applications designed to manipulate text files, called text editors, allow the user to edit and save text with several encodings. For example, a text editor that uses the GUI is "gedit". On the terminal we have also several text editors. The most remarkable one is `vi`[1] or `vim` (a enhanced version of vi). `vi` might be little cryptic but it is useful because it is present in almost any Unix system. Let's get familiarized with `vi`. For example, to start editing the file myfile.txt with `vi`, you should type:

```
$ vi myfile.txt
```

The previous command puts `vi` in *command mode* to edit myfile.txt. In this mode, you can navigate through myfile.txt and quit by typing `:q!`. Also, in this mode you can delete a line with *dd*, delete from the cursor to the end of the line with *d$* and from the cursor to the beginning of the line with *d^*. You can go to a determinate line with *Gn* (where *n* is the number of the line). If you want to edit the file, you have to press "`i`", which puts `vi` in *insertion mode*. After modifying the document, you can hit `ESC` to go back to *command mode* (default one).

To save the file you must type `:wq` and to quit without saving, you must force the exit by typing `:q!`.

On the other hand, there are also other commands to view text files. These commands are `cat`, `more` and `less`. The `less` command works in the same way as `man` does. Try:

```
$ cat /etc/passwd
$ cat /etc/passwd /etc/hostname
$ more /etc/passwd
$ less /etc/passwd
```

Another couple of useful commands are `head` and `tail`, which respectively, show us the text lines at the top of the file or at the bottom of the file.

```
$ head /etc/passwd
$ tail -3 /etc/passwd
```

A very interesting option of tail is -f, which outputs appended data as the file grows. Example:

```
$ tail -f /var/log/syslog
```

If we have a binary file, we can use `hexdump` or `od` to see its contents in hexadecimal and also in other formats. Another useful command is `strings`, which will find and show characters or groups of characters (strings) contained in a binary file. Try:

[1] There are other command-line text editors like `nano`, `joe`, etc.

```
$ hexdump /bin/ls
$ strings /bin/ls
$ cat /bin/ls
```

There are control characters in the ASCII tables that can be present in binary files but that should never appear in a text file. If we accidentally use `cat` over a binary file, the prompt may turn into a strange state. To exit this state, you must type `reset` and hit ENTER.

Other very useful commands are those that allow us to search for a pattern within a file. This is the purpose of the `grep` command. The first argument of `grep` is a pattern and the second is a file. Example:

```
$ grep bash /etc/passwd
$ grep -v bash /etc/passwd
```

Finally, another interesting command is `cut`. This command can be used to split the content of a text line using a specified delimiter. Examples:

```
$ cat /etc/passwd
$ cut -c 1-4 /etc/passwd
$ cut -d ":" -f 1,4 /etc/passwd
$ cut -d ":" -f 1-4 /etc/passwd
```

3.14 Links

A link is a special file. Links can be hard or symbolic (soft).

- **A Hard Link** is a way of giving another name to a file.
  - Each name (hard link) can use a different location in the filesystem.
  - Hard links must refer to existent data in a certain file system.
  - Two hard links offer the same functionality with different names. E.g. any of the hard links can be used to modify the data of the file.
  - A file will not exist anymore if all its hard links are removed.
- **A Symbolic Link (also called Soft Link)** is a new (different) file whose contents are a pointer to another file or directory.
  - If the original file is deleted, the soft link becomes unusable.
  - The soft link is usable again if original file is restored.
  - Soft links allow to link files and directories between different FS, which is not allowed by hard links.

The `ln` command is used to create links. If the -s option is passed as argument, the link will be symbolic. Examples:

```
 $ ln -s /etc/passwd ~/hello
 $ cat ~/hello
```

The previous `ln` command creates a symbolic link to `/etc/passwd` and the `cat` command prints the contents as text of the link. We can also use hard links to rename a file. For example:

```
 $ touch file1.txt
 $ ln file1.txt file2.txt
 $ stat file1.txt
  File: `file1.txt'
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 811h/2065d      Inode: 1056276     Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/ user)   Gid: ( 1000/ user)
Access: 2013-06-25 18:47:52.675904819 +0200
Modify: 2013-06-25 18:47:52.675904819 +0200
```

```
Change: 2013-06-25 18:48:07.667904983 +0200
 Birth: -
 $ rm file1.txt
```

The previous `ln` and `rm` commands are equivalent to the following `mv` command:

```
 $ mv file1.txt file2.txt
```

Links, especially symbolic links, are frequently used in Linux system administration. Commands are often aliased so the user does not have to know a version number for the current command, but can access other versions by longer names if necessary. Library names are also managed extensively using symlinks, for example, to allow programs to link to a general name while getting the current version.

3.15 Unix Filesystem Permission System

Introduction

Unix operating systems are organized in users and groups. Upon entering the system, the user must enter a login name and a password. The login name uniquely identifies the user. While a user is a particular individual who may enter the system, a group represents a set of users that share some characteristics. A user can belong to several groups, but at least the user must belong to one group. The system also uses groups and users to perform some of its internal management tasks. For this reason, in addition to real users, in a Unix system there will be other users. Generally, these "special" users cannot login into the system, i.e., they cannot have a GUI or a CLI.

Permissions

Linux FS provides us with the ability of having a strict control of files and directories. To this respect, we can control which users and which operations are allowed over certain files or directories. To do so, the basic mechanism (despite there are more mechanisms available) is the "Unix Filesystem Permission System".

There are three specific permissions on this permission system:

- The **read permission**, which grants the ability to read a file. When set for a directory, this permission grants the ability to read the names of files in the directory (but not to find out any further information about them such as contents, file type, size, ownership, permissions, etc.)
- The **write permission**, which grants the ability to modify a file. When set for a directory, this permission grants the ability to modify entries in the directory. This includes creating files, deleting files, and renaming files.
- The **execute permission**, which grants the ability to execute a file. This permission must be set for executable binaries (for example, a compiled C++ program) or shell scripts to allow the operating system to run them. When set for a directory, the execution permission grants the ability to traverse the directory to access to files or subdirectories, but it does not grant the permission to view the directory content (unless the read permission is set for the directory).

When a permission is not set, the rights it would grant are denied. Unlike other systems, permissions on a Unix-like system are not inherited. Files created within a directory will not necessarily have the same permissions as that directory. On the other hand, from the point of view of a file, the user is in one of the three following categories or classes:

- User Class. The user is the owner of the file.
- Group Class. The user belongs to the group of the file.
- Other Class. Neither of the two previous situations.

The most common form of showing permissions is symbolic notation. The following are some examples of symbolic notation:

`-rwxr-xr-x` a regular file whose user class has full permissions and whose group and others classes have only the read and execute permissions.

`dr-x------` a directory whose user class has read and execute permissions and whose group and others classes have no permissions.

The command to list the permissions of a file is `ls -l`. Example:

```
$ls -l /usr
total 188
drwxr-xr-x   2 root root 69632 2011-08-23 18:39 bin
drwxr-xr-x   2 root root  4096 2011-04-26 00:57 games
drwxr-xr-x  41 root root  4096 2011-06-04 02:32 include
drwxr-xr-x 251 root root 69632 2011-08-20 17:59 lib
drwxr-xr-x   3 root root  4096 2011-04-26 00:56 lib64
drwxr-xr-x  10 root root  4096 2011-04-26 00:50 local
drwxr-xr-x   9 root root  4096 2011-06-04 04:11 NX
drwxr-xr-x   2 root root 12288 2011-08-23 18:39 sbin
drwxr-xr-x 370 root root 12288 2011-08-08 08:28 share
drwxrwsr-x  11 root src   4096 2011-08-20 17:59 src
```

Change permissions (chmod)

The command `chmod` is used to change the permissions of a file or directory.

Syntax: `chmod user_type operation permissions file`

User Type	u	user
	g	group
	o	other
Operation	+	Add permission
	-	Remove permission
	=	Assign permission
Permissions	r	reading
	w	writing
	x	execution

For example, to assign the read and execute permissions to the group class of the file ”temp.txt“ we can type the following command:

```
$ chmod g+rx temp.txt
```

Changing the permissions of the file ”file1.c“ for allowing only the user to read its contents can be achieved by:

```
$ chmod u=r file1.c
```

Another example for setting the permissions of the user and the group simultaneously:

```
$ chmod u=r,g=rx file1.c
```

Another way of managing permissions is to use octal notation. With three-digit octal notation, each numeral represents a different component of the permission set: user class, group class, and other class respectively. Each of these digits is the sum of its component bits. Here is a summary of the meanings for individual octal digit values:

```
0 --- no permission
1 --x execute
2 -w- write
3 -wx write and execute
```

```
4 r-- read
5 r-x read and execute
6 rw- read and write
7 rwx read, write and execute
```

For example, to grant the read permission for all the classes, the write permission only for the user and execution for the group over the file "file1.c" you should type:

```
$  chmod 654 file1.c
```

The numeric values come from:

$$r_{user} + w_{user} + r_{group} + x_{group} + r_{other} = 400 + 200 + 40 + 10 + 4 = 654$$

Default permissions

Users can also establish the default permissions for their new created files. The `umask` command allows to define these default permissions. When used without parameters, returns the current mask value:

```
$ umask
0022
```

You can also set a mask. Example:

```
$ umask 0044
```

For security reasons, only read and write permissions can be used by the default mask but not execute. The mask tells us in fact which permission is subtracted (i.e. it is not granted).

3.16 Extra

3.16.1 *inodes

The inode (index node) is a fundamental concept in the Linux and UNIX filesystem. Each object in the filesystem is represented by an inode. Each and every file under Linux (and UNIX) has following attributes:

- Inode number.
- File type (regular file, block special, etc).
- Permissions (read, write etc).
- Owner.
- Group.
- File Size.
- Access Time (atime). This is the time that the file was last accessed, read or written to.
- Modify Time (mtime). This is the time that any inode information was last modified.
- Change Time (ctime). This is the last time the actual contents of the file were last modified.
- Number of links (soft/hard).
- Etc.

All the above information stored in an inode. In short the inode identifies the file and its attributes (as above). Each inode is identified by a unique inode number within the file system. Inode is also know as index number. An inode is a data structure on a traditional Unix-style file system such as UFS or ext4. An inode stores basic information about a regular file, directory, or other file system object. You can use `ls -i` command to see inode number of file:

```
$ ls -i /etc/passwd
32820 /etc/passwd
```

You can also use `stat` command to find out inode number and its attributes:

```
$ stat /etc/passwd

File: `/etc/passwd'
Size: 1988            Blocks: 8          IO Block: 4096   regular file
Device: 341h/833d       Inode: 32820       Links: 1
........
```

Many commands often give inode numbers to designate a file. Let us see he practical application of inode number. Let us try to delete file using inode number.

Create a hard to delete file name:

```
$ cd /tmp
$ touch "\+Xy \+\8"
$ ls
```

Try to remove this file with rm command:

```
$ rm \+Xy \+\8
```

Remove file by an inode number, but first find out the file inode number:

```
$ ls -il
```

The `rm` command cannot directly remove a file by its inode number, but we can use `find` command to delete file by inode.

```
$ find . -inum 471257 -exec rm -i {} \;
```

In this case, 471257 is the inode number that we want to delete.

Note you can also use add character before special character in filename to remove it directly so the command would be:

```
$ rm "\+Xy \+\8"
```

If you have file like name like name “2011/8/31” then no UNIX or Linux command can delete this file by name. Only method to delete such file is delete file by an inode number. Linux or UNIX never allows creating filename like this but if you are using NFS from MAC OS or Windows then it is possible to create a such file.

3.17 Command summary

The table 3.1 summarizes the commands used within this section.

3.18 Practices

Exercise 3.1– This exercise is related to the Linux filesystem and its basic permission system.

1. Open a terminal and navigate to your home directory (type `cd ~` or simply `cd`). Then, type the command that using a relative path changes your location into the directory /etc.
2. Type the command to return to home directory using an absolute path.
3. Once at your home directory, type a command to copy the file /etc/passwd in your working directory using only relative paths.
4. Create six directories named: dirA1, dirA2, dirB1, dirB2, dirC1 and dirC2 inside your home directory. You can do this with the command:

Table 3.1: Summary of commands related to the filesystem.

stat	shows file metadata.
file	guess file contents.
df	display the amount of available disk space of a filesystem.
du	file space used under a particular directory or by files of a filesystem.
cd	changes working directory.
ls	lists a directory.
pwd	prints working directory.
mkdir	makes a directory.
rmdir	removes a directory.
rm	removes a file or directory.
mv	moves a file or directory.
cp	copies a file or directory.
touch	updates temporal stamps and creates files.
gedit	graphical application to edit text.
vi	application to edit text from the terminal.
cat	shows text files.
more	shows text files with paging.
less	shows text files like `man`.
head	prints the top lines of a text file.
tail	prints the bottom lines of a text file.
hexdump and `od`	shows file data in hex and other formats.
`strings`	looks for character strings in binary files.
grep	prints text lines that match a pattern.
cut	cuts a text string.
ln	creates hard and soft links.
`chmod`	changes file permissions.
`umask`	shows/sets default access mask of new files.

```
$ mkdir dirA1 dirA2 dirB1 dirB2 dirC1 dirC2
```

Or using a functionality called "brace expansion":

```
$ mkdir dir{A,B,C}{1,2}
```

Then, write a command to delete dirA1, dirA2, dirB1 and dirB2 but not dirC1 or dirC2.

Are you able to find another command that produces the same result?

5. Delete directories dirC2 and dirC1 using the wildcard "?".

6. Create an empty file in your working directory called `temp`.

7. Type a command for viewing text to display the contents of the file, which obviously must be empty.

8. Type a command to display the file metadata and properties (creation date, modification date, last access date, inode etc.).

9. What kind of content is shown for the `temp`? and what kind basic file is?

10. Change to your working directory. From there, type a command to try to copy the file `temp` to the `/usr` directory. What happened and why?

11. Create a directory called `practices` inside your home. Inside `practices`, create two directories called `with_permission` and `without_permission`. Then, remove your own permission to write into the directory `without_permission`.

12. Try to copy the `temp` file to the directories `with_permission` and `without_permission`. Explain what has happened in each case and why.

13. Figure out which is the minimum set of permissions (read, write, execute) that the owner has to have to execute the following commands:

Commands	read	write	execute
`cd without_permission`			
`cd without_permission; ls -l`			
`cp temp ~/practices/without_permission`			

Exercise 3.2– This exercise presents practices about text files and special files.

1. Create a file called orig.txt with the `touch` command and use the command `ln` to create a symbolic link to orig.txt called link.txt. Open the `vi` text editor and modify the file orig.txt entering some text.
2. Use the command `cat` to view link.txt. What can you observe? why?.
3. Repeat previous two steps but this time modifying first the link.txt file and then viewing the orig.txt file. Discuss the results.
4. Remove all permissions from orig.txt and try to modify the link.txt file. What happened?
5. Give back the write permission to orig.txt. Then, try to remove the write permission to link.txt. Type `ls -l` and discuss the results.
6. Delete the file orig.txt and try to display the contents of link.txt with the `cat` command. Then, in a terminal (t1) edit orig.txt with the command:

```
t1$ vi orig.txt
```

   While the editor is opened, in another terminal (t2) type:

```
t2$ echo hello > link.txt
```

   Close `vi` and comment what has happened in this case.
7. Use the command `stat` to see the number of links that orig.txt and link.txt have.
8. Now create a hard link for the orig.txt file called hard.txt. Then, using the command `stat` figure out the number of "Links" of orig.txt and hard.txt.
9. Delete the file orig.txt and try to modify with `vi` hard.txt. What happened?
10. Use the `grep` command to find all the information about the HTTP protocol present in the file /etc/services (remember that Unix commands are case-sensitive).
11. Use the `cut` command over the file /etc/group to display the name of each group and its members (last field).
12. Create an empty file called text1.txt. Use text editor `vi` abn to introduce "abñ" in the file, save and exit. Type a command to figure out the type of content of the file.
13. Search in the Web the hexadecimal encoding of the letter "ñ" in ISO-8859-15 and UTF8. Use the command `hexdump` to view the content in hexadecimal of text1.txt. Which encoding have you found?
14. Find out what the character is "0x0a", which also appears in the file.
15. Open the `gedit` text editor and type "abñ". Go to the menu and use the option "Save As" to save the file with the name text2.txt and "Line Ending" type Windows. Again with the `hexdump` examine the contents of the file. Find out which is the character encoded as "0x0d".

```
$ hexdump text2.txt
0000000 6261 b1c3 0a0d
0000006
```

16. Explain the different types of line breaks for Unix (new Mac), Windows and classical Mac.

17. Open the gedit text editor and type ”abñ“. Go to the menu and use the option “Save As” to save the file with the name text3.txt and “Character Encoding” ISO-8859-15. Recheck the contents of the text file with `hexdump` and discuss the results.