

Linux introduction

Network Infrastructures labs

Marco Spaziani Brunella



SAPIENZA
UNIVERSITÀ DI ROMA

- **Readings:**

- Unix Power Tools, Powers et al., O'Reilly
- Linux in a Nutshell, Siever et al., O'Reilly

- **Lecture outline:**

- Linux File System
- Users and Groups
- Shell
- Text Editors
- Misc commands

What is Linux?

Linux is not an OS, is a kernel. The kernel provides all the interfaces to interact with hardware devices to user applications.

What is GNU/Linux?

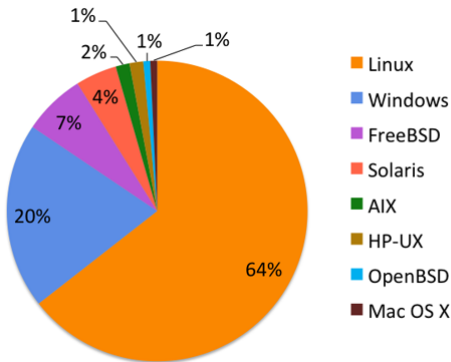
GNU/Linux is an OS based on the Linux kernel. It wraps the Linux Kernel with a set of GNU tools (gcc, gdb, ...).

GNU/Linux comes in different flavours, called distributions (or *distros*) such as:

- Debian GNU/Linux
- Ubuntu, Xubuntu, Lubuntu, Kubuntu
- Fedora
- Red-Hat Enterprise Linux
- Kali Linux

Why we use GNU/Linux?

Is one of the most spread OS in datacenters and enterprise networks.



Apache HTTP host OS adoption

Interacting with a Linux machine

The main way of interacting with a Linux machine is via a "shell" that runs in a Terminal Emulator.

There are many terminal emulators, such as:

- Xterm *
- LX Terminal (the one on the VM)
- Konsole
- gnome-terminal

From now on, everything with a * is something that I suggest you to use

The Shell & commands

The shell is a command interpreter and enables you to interact with the Operating System. As always, there are many available shells, although the most common is Bash* (Bourne Again SHell).

Inside the shell, you can execute "commands", which are programs that may take or not parameters in order to execute correctly.

We will indicate the shell from now on via:

```
user@localhost:$ command
```

Every command takes tons of parameters in order to configure his behaviour. To know more about a command, use the man command, which will open the Linux manual entry for that command:

```
user@localhost:$ man command
```

How files and directories are managed inside Linux?

In Linux, files and directories are organized in a reverse tree. The "root" of the tree is the "/" directory.

Files are leaves of the tree, while directories are branches.

Absolute and relative paths

Supposing that we are in a particular directory (working directory) of the previous image, lets say "spool", and we want to go to "var".

We can specify the path to "var" in 2 different ways:

- Absolute → `/var` (*from "root" to "var"*)
- Relative → `../` (*from current directory to upper one*)

With absolute paths, we specify the paths to the folder (or file) starting from the root directory.

With relative paths, we specify the paths to the folder (or file) starting from the working directory.

In particular, `./` means "current directory" and `../` means "upper directory".

Moving around the FS

We can move inside the File System with the shell via dedicated commands, such as:

```
pwd #prints the current (working) directory\\  
  
cd "path" #changes the current directory to  
          #the one specified in "path"  
          #the path can be absolute or relative
```

For example:

```
user@localhost:$ pwd  
/home/user  
user@localhost:$ cd ../  
user@localhost:$ pwd  
/home  
user@loclahost:$ cd /etc  
user@localhost:$ pwd  
/etc
```

List files in a directory

We can list all the files and folder in a directory via the `ls` command.

```
ls      #list directory contents
```

```
ls -a   #list directory contents  
        #including hidden files and folders
```

```
ls -l   #list directory contents  
        #with all the details  
        #such as permissions, UID, GID, size etc...
```

Of course, options can be combined to form more complex patterns, such as:

```
ls -alh #discover it yourself!
```

Create, remove and copy files/directories

```
touch "filename"          #creates a file called "filename"

mkdir "folder_name"       #creates a folder called "folder_name"

rm "path_to_file"         #removes a file in "path_to_file"

rm -rf "path_to_dir"      #removes a directory in "path_to_dir"

cp "source_file" "dest_folder" #copies "source_file"
                                #to "dest_folder"
```

What does individual folders contain?

According to the Filesystem Hierarchy Standard:

Everything is a file

In Linux, barely everything is a file. Your keyboard, your NIC, ecc... is associated with a type of file, inside the `/dev` folder, called "special" or "device" file. It acts as an interface for the device driver, and enables the OS to actually "speak" with the device.

Device files does not need to correspond to physical devices, but they can represent a "pseudo" device, such as `/dev/zero`, `/dev/null` and `/dev/random`. Other kind of files are available, such as "link", which are files that links to another file on the filesystem.

Files that begins with a dot are treated by the OS as hidden files.

Users and Groups

Users and Groups are a nice way to keep things in order and to provide security among files and directories.

Every user is associated with a home directory, which will normally be placed inside the /home folder, and a User ID (UID).

Every user belong to a group of users, wich is identified by a Group ID.

These concepts allow us to define permissions on file and directories.

There is a special user, called "superuser" or "root", which is the system administrator and can access every file of every user.

Permissions

Every file and directory has some permissions associated, telling if the User who owns the file, the Group of users with the file belongs and all the other users can Read, Write or eXecute the file.

For the file type, we have a "d" for a directory and a "-" for a normal file.

Changing file permissions and owner 1/2

Permissions are organized as 3 (user, group, other) chunks of 3 (read, write, execute) one-hot bits.

In order to change permissions to file or folders we use the *chmod* command.

Changing file permissions and owner 2/2

```
chmod "permission_seq" "file_folder_name"  
#changes "file_folder_name"  
#permissions to "permission_seq"  
  
chmod +x "file_folder_name" #adds execute permissions  
                             #to "file_folder_name"  
  
chmod -r "file_folder_name" #removes read permissions  
                             #to "file_folder_name"
```

For example, suppose we have a file called "foo.txt" and we want to change his permissions to r - x - w x r - - :

```
user@localhost:$ touch foo.txt  
user@localhost:$ chmod 534 foo.txt  
user@localhost:$ ls -al foo.txt  
-r-x-wxr-- 1 user user 0 ott 12 19:32 foo.txt
```

Edit files

In order to edit files, you will need a text editor, such as:

- vi(m)*, (vim stands for vi-iMproved)
- nano
- gedit
- emacs

For a tutorial on vim:

```
user@localhost:$ vimtutor
```

STDIN, STDOUT, STDERR

Every time a command executes inside a shell, it opens 3 streams:

- STDIN
- STDOUT
- STDERR

Excluding case of redirection (see later on), the STDIN of a program is the keyboard and is where the program takes his inputs, the STDOUT is the monitor and is where the program sends his output and the STDERR where the program sends its errors (typically the monitor).

Interacting with STDOUT

We can print strings and files content to STDOUT with:

```
echo "string" #prints "string" to stdout
```

```
cat "filename" #prints the content of "file_name" to stdout
```

For example, supposing that foo.txt contains "alice":

```
user@localhost:$ echo Hello World!
```

```
Hello World!
```

```
user@localhost:$ cat foo.txt
```

```
alice
```

I/O Redirection 1/2

We can redirect the stdout of a program to a file, for example

```
user@localhost:$ echo Hello World! > helloworld.txt
```

```
user@localhost:$ cat helloworld.txt
```

```
Hello World!
```

```
user@localhost:$ echo Hello World! >> helloworld.txt
```

```
user@localhost:$ cat helloworld.txt
```

```
Hello World!
```

```
Hello World!
```

Similarly, we can redirect stdin of a command from a file:

```
user@localhost:$ printf "5\n4\n3\n2\n1" > sort.txt
```

```
user@localhost:$ sort < sort.txt
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

I/O Redirection 2/2

We can have more expressive I/O redirections, summarized in the table below:

Pipes 1/2

We can create more complex commands starting from simpler ones via pipes.

Basically, we redirect the stout of the first command to the stdin of the second one.

This process can be reiterated.

Pipes 2/2

A pipe is created in a shell via "|" between two commands:

```
user@localhost:$ command_1 | command_2
```

For example:

```
user@localhost:$ printf "5\n4\n3\n2\n1" > sort.txt
```

```
user@localhost:$ cat sort.txt | sort
```

```
1
2
3
4
5
```

Processes

Every command executed in the shell spawns a process. Also the processes are organized in an hierarchical fashion, with the child processes having only one father process. The father of all processes is `init`.

Normally, killing a father will also kill its children. Processes communicate one another using signals. Background processes are called daemons.

Managing processes

```
ps axu      #list all active processes
```

```
top          #same as before, but better
```

```
htop        #same as before, but better
```

```
kill -signal_num PID #sends signal_num to the process  
                      #with PID
```

Combining pipelines and a new command called `grep`, we can search for a particular process

```
ps axu | grep "process_name" #gives information about  
                              #"process_name" only
```

```
kill -9 "process_name" #sends SIGKILL to "process_name"
```