Lecture 1: Linux and Network Administration

Lorenzo Bracciale

February 8, 2012

Contents

1 What is Linux?

Linux is a family of operating system based on the Linux kernel. A kernel is the piece of code of an operating system that is responsible for the core functionalities such as accessing the memory, interfacing with devices (drivers), or deciding which of the several programs running on a machine should use the CPU (scheduler) and many others tasks.

The Linux Kernel was initially written for hobby in 1991 by Linus Towalds, a 21 years old student at the University of Helsinki, and nowadays is improved and maintained by a lot of developers from all over the world. It consists in more than 6 billions lines of code witten in C but it is so flexible to been used in more than the 90% of supercomputers and in very tiny devices (it fits on a floppy). Explaining in full details Linux is out of the scope of this course; we limit our attention to what is functional to networking administration.

Aside the technical part, one of the reason of the popularity of Linux is the legal licence that regulates the distribution of the source code. This licence is called GPL [1], the GNU General Public License. Basically it states that everyone is free to copy, modify and improves the codes, allows to sell products with GPL code insides (as happens for a wide range of electronics such as consumers routers that are based on linux) however obliges companies to redistribute the GPL codes and its modification for their products, creating a so called "viral effect" (if you modify code released under GPL, your codes must be released on GPL too).

However, aside from the core functionalities, every operating system is usable if it is equipped with some programs that ultimately could exploit kernel functionalities by accessing to some API called *system calls*, for offering to the users some kind of services (graphical interface, applications, background, network services etc).

Depending on your personal taste, you can choose the right *Linux Distribution* that suits you. A *distro* is a "bundle" that comprehends Linux kernel, graphics, set of scripts and packages. There are Linux distros that fill inside a floppy disk, others optimized for security, other for usability etc.

In what follows we focus on Debian-like distribution (Ubuntu, Debian, Knoppix etc).

2 Linux shell and basic commands

A fundamental programs that we will extensively use throughout this course is the shell.
A shell is the interpreter of commands and offer to the user a command line interface (CLI). When a user login an Linux system, the system gives him a shell. On desktop Linux systems, users could login via graphical interfaces. In this case, a shell could be retrieved by executing the *Terminal* applications, or exiting from the graphical interfaces (for instance by pressing CTRL+ALT+F2 to go to another virtual terminal). Basically using a shell, users could start or stop programs.

A shell is a program itself that usually stands in the `/bin` directory of the filesystem. There exist several different shells (bash, ash, csh, sh), but we will focus on bash shell when no otherwise specified.
Being comfortable with a Linux shell is fundamental for a network administrator because several network devices could not have a graphical interfaces.
A shell usually presents this interface:

```
user@hostname:current_dir$
```

where the last characters can be `$` if we are normal users or `#` if we are the system administrator (also called superuser or *root*).

A superuser potentially could execute any code on a machine, while users are usually have restrictions for security reasons. For the sake of the network administration we will often require root privileges. If we think about capturing each packets that transit on a network interface (and that could contain password and other sensible data) we realize that this is an operation that should be done only by the superuser.

Let we start with a very basic command that is `whoami`. As the name suggest, this commands return the current user name. However if we have some doubts about the right usage of that commands we can resort to the manual page by typing `man whoami` (press `q` to exit from the manual page) or by calling the command with the help argument i.e. typing `whoami --help`.

Even if in the case of the `whoami` program, the documentation is not so interesting, soon we will see more complex commands that accept several arguments with different parameters. Reading carefully these documentations usually solves every problems related to "how can I do to do a certain thing using a program?"
In what follows we limit our attention to some basic commands because a comprehensive list of all the Linux commands could be hard to read and memorize (and every command has a lot of different parameters!), so we explain programs as we need it.

Now we can try to type `sudo whoami` to see that the system will ask us for the password and then will return `root`.
What we did? If we open the manual page of the sudo program, we can read that `sudo execute a command as another user`. We can instruct the `sudo` to execute a program as user `USER` by typing `sudo -U USER program`, or, if we do not provide any argument, it will assume that we want to execute the command as root.

3 Users

Linux associates to each user an unique number (typically a short integer number) called UID (User ID). In the same way, there exists groups and every group has a GUI (Group ID). Each user can belong to one or more group. `root` has UID 0, as the first 99 UID are reserved for special accounts.

To get a list of UID/GID of our system it suffices to view the content of `/etc/passwd` using the command `cat` that "concatenate files and print to standard output" :

```
ninux@ale:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
...
```

This file contains one line for each user of the system. On each line, separated by comma, there are:

- The user name
- An ”x”. In the old Linux system here there was the encrypted (hashed) password of the user. Given that this file is visible by any user, malicious users could execute bruteforce or dictionary attacks to try to decode all the password of other user (and root!). Now usually password are stored in`/etc/shadow` (that is readable just for root) and `/etc/passwd` contains only this placeholder.
- User ID (UID)
- The primary Group ID (GID)
- Extra information (such as the real name of the user)
- The home directory
- The absolute path of a command or shell (e.g. `/bin/bash` or `/bin/false` to does not allow this user to login via shell)

When an user log in, the system gives him the the corresponding shell and set him in his home directory.

If you are interested into seeing the groups you can take a look at the file `/etc/group` whose goal is to bind a group name with its GID and with the list of users that belongs to that group.

As we told, with commands `su` you can ”switch user” , with the command `sudo` you can execute one command as another user, and with the command `whoami` you can see what is your current user name.

If you are interested in seeing the users currently logged inside your system you can launch the `who` command, while with the `last` command you will obtain the dump of the last logged users in the system.

4 Files

Files play a fundamental role in every Linux distribution, not only because programs and configuration data are stored into files, but also because it represents a more general interface to simply access several things. For example, to instruct the Linux kernel to forward or deny forwarding of IP packet, it suffices to write an ”1” in a ”special” file that is `/proc/sys/net/ipv4/forwarding`.

Moreover, also the interface to write data to network (socket) is based on a file like abstraction so that writing "hello" to a file is quite similar to write it on a TCP connection.

When logged in, a user can move inside directory and view files, according to his permissions.
The system set an *environment variable* (what is it will be clean later on) to keep track of where the user is: that is called *working directory*.
Given that the user log in inside his home directory, when he bumps to another directory, we say that he is changing his working directory.

4.1 Commands

Let we start with a bunch of commands that allow us to handle file:

ls lists files in the current working directory. Popular options are `-l` to view the output as a long list, `-a` (all) to view also the files that start with a dot, `-h` to print the file size in an human readable format (e.g. 25G). So `ls -alh` comprends all these options in a once.

cd changes working directory, for example `cd MYDIR`

pwd prints working directory

cp copy a file. With -r option copies recursively the content of a directory. For example `cp FILEORIG FILEDEST`

mv moves a file. Useful also for renaming a file. For example `mv FILEORIG FILEDEST`

rm removes (deletes) a file. With -r option remove a recursevely the content of a directory, with -f option do not ask for confirmation. Beware, dangerous!!! For example `rm -rf MYDIR` will wipe out MYDIR

cat concatenates files and print them to standard output. For instance `cat MYFILE` will dump the content of the file to the screen (standard output).

tail views the last lines of a file (by default 10 lines). With the -f options will show continuously the last line of a file, very useful to see a log while it is populated!

head views the first lines of a file.

mkdir creates a new directory. For example `mkdir MYDIR` rmdir removes and empties a directory. If we want to remove a directory with files within, we should use the `rm` command as said before.

To search files and inside a file it would be useful to get practice with the following commands:

find to find a file "example.txt" inside the "/etc" directory we can type `find /etc -name example.txt`. It will search for that filename inside all the subdirectory of `/etc` and print on screen the results.

locate is used to find a file as well `locate example.txt`, however it works on a different way; instead of scanning a dir for finding a file, it lookup in a local database. This database must be created (and keeping updated) by root using the command `updatedb`

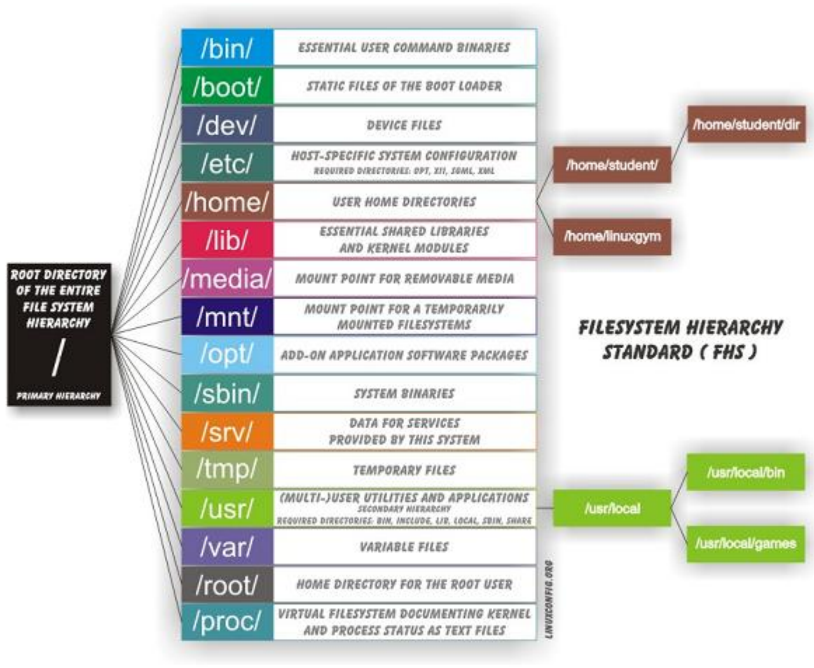
grep without any parameters it is useful to obtain only the lines of a file that contains a given word. For instance `grep yourusername /etc/passwd` will return the line of passwd in which is written your username.

Finally, we present the `link` command. With this command is it possible to create links between files or directories. For instance by giving `ln -s DIR1 MYLINK` we are creating a symbolic link between the new created MYLINK and DIR1, so that we can do `cd MYLINK` to access to DIR1. This kind of links (symbolic, because of the `-s` option) are usually more used in system/network administration as they create a file with a symbolic path indicating the abstract location of another file/directory. Conversely, hard links refer to the specific location of physical data.

4.2 Directories

Linux have several main directory that stands inside the `root` directory (that is `/`). This structure is depicted in figure 1. For instance, a lot of programs could be in the `/usr/bin` directory while their configuration file could stay inside the `/etc/` directory and their source code could stay inside `/usr/src`.

Some particular directory are the `/dev` and the `/proc` directories.
The `/dev` directory contains the special device files for all the devices. When you plug an USB stick on your linux machine, for example, a device file will come out (e.g. `/dev/sda1`).
Examples of devices can be hard disks (e.g. `/dev/sda`), partitions of hard disk (e.g. `/dev/sda1`), sound card (e.g. `/dev/dsp`), serial port (e.g. `/dev/tty1`), usb devices and so on.



ROOT DIRECTORY OF THE ENTIRE FILE SYSTEM HIERARCHY
/
PRIMARY HIERARCHY
/bin/
ESSENTIAL USER COMMAND BINARIES
/boot/
STATIC FILES OF THE BOOT LOADER
/dev/
DEVICE FILES
/etc/
HOST-SPECIFIC SYSTEM CONFIGURATION
REQUIRED DIRECTORIES: OPT, X11, SGML, XML
/home/
USER HOME DIRECTORIES
/lib/
ESSENTIAL SHARED LIBRARIES AND KERNEL MODULES
/media/
MOUNT POINT FOR REMOVABLE MEDIA
/mnt/
MOUNT POINT FOR A TEMPORARILY MOUNTED FILESYSTEMS
/opt/
ADD-ON APPLICATION SOFTWARE PACKAGES
/sbin/
SYSTEM BINARIES
/srv/
DATA FOR SERVICES PROVIDED BY THIS SYSTEM
/tmp/
TEMPORARY FILES
/usr/
(MULTI-)USER UTILITIES AND APPLICATIONS
SECONDARY HIERARCHY
REQUIRED DIRECTORIES: BIN, INCLUDE, LIB, LOCAL, SBIN, SHARE
/var/
VARIABLE FILES
/root/
HOME DIRECTORY FOR THE ROOT USER
/proc/
VIRTUAL FILESYSTEM DOCUMENTING KERNEL AND PROCESS STATUS AS TEXT FILES
/home/student/
/home/student/dir
/home/linuxgym
/usr/local
/usr/local/bin
/usr/local/games
FILESYSTEM HIERARCHY STANDARD (FHS)
LINUXCONFIG.ORG

Figure 1: Linux directory structure

Want to copy the entire content of an usb stick to a file? `dd if=/dev/sda1 of=/home/myhome/myusbstickfile`.
`dd` is the linux utility to convert and copy a file (you must only specify the input file `if` and the output file `of`), but how we said there are many things on Linux that looks like a file.
Want to fill a file with 1G of random data? `dd` and `/dev/random` (a special device file that "produces" random data) comes in handy: `dd if=/dev/random of=/home/myhome/randomdata bs=1M count=1024`
Want to wipe out all your hard disk drive? `dd if=/dev/zero of=/dev/sda bs=1M count=1024`, ok please don't do it!
Other devices file are `/dev/zero` (that contains zero, indeed), `/dev/null` that is a "blackhole". `/dev/null` is usefull when you want to discard the output of a command; in this case all you need is just to redirect that output to `/dev/null`.

The `/proc` filesystem acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime.

The goal of the `/proc` filesystem is to provide an easy way to view kernel and information about currently running processes. As a result, some commands (ps for example) read `/proc` directly to get information about the state of the system. The premise behind /proc is to provide such information in a readable manner instead of having to invoke difficult to understand system calls.

So, to modify some parameters of the Linux kernel we can find the right file inside the /proc filesystem or invoke the sysctl command (that do that for you). Let we make an example. Suppose we want to configure our machine to ignore ICMP echo message so that it will no more answer to PING.

We can view the current status of the variable icmp_echo_ignore_all.

```
root@ale:~# sysctl -a | grep net.ipv4.icmp_echo_ignore_all
net.ipv4.icmp_echo_ignore_all = 0
```

Then we can set that variable to 1, to ignore such request.

```
root@ale:~# sysctl -w net.ipv4.icmp_echo_ignore_all=1
net.ipv4.icmp_echo_ignore_all = 1
```

To make these change permanent we should edit the file /etc/sysctl.conf.

Alternatively we can access to the right /proc file directly that is /proc/sys/net/ipv4/icmp_echo_ignore_all. We can view the value by reading the file:

```
root@ale:~# cat /proc/sys/net/ipv4/icmp_echo_ignore_all
0
```

and set the value by writing a 1 inside the file:

```
root@ale:~# echo "1" > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

Now do a `ls -alh /proc/sys/net/ipv4/*` to see how many variables you can change for the behaviour of the ip protocol stack of your machine.

4.3 File Permissions

On Linux each file has some permission that could be read, write or execute and that could refer to the owner, the the group that the owner belongs to, and others. Let we make an example. Let we create an empty file with the command `touch example.txt` and let we view its permission with the command `ls -l example.txt`. We should see something like this:

```
ninux@ale:~$ ls -l example.txt
-rw-rw-r-- 1 me mygroup 0 2012-01-19 12:25 example.txt
```

This line says (from right to left) that we have a file called `example.txt`, modified last time on `2012-01-19 12:25`, created by user `me` that belongs to `mygroup`, that has permission represented by the string `-rw-rw-r--` . The string `-rw-rw-r--` tell us the permissions of that file. Given that this is a file, the first char is just an `-`, if we were listing a directory, we would see a `d`. After that three group of three char that means that the owner has permission `rw-` (read and write), members of the owner group has rights to read and write the file as well (`rw-`) while others have only the right to view the file `r--`. If we want to grant permission of writing this file also to others, we can give the command:

```
ninux@ale:~$ chmod a+w example.txt
ninux@ale:~$ ls -l example.txt
-rw-rw-rw- 1 me mygroup 0 2012-01-19 12:25 example.txt
```

On the same way, to revoke the write permission to owner or his group we can give the commands `chmod o-w` and `chmod g-w` or just `chmod og-w`. Often permission are given using a binary mask. For instance the triplet `110` means read and write, while `101` means read and execute. If we map these binary string to decimal we obtain a number from 0 to 7 in which 0 means "no permission", while 7 means "all permission". We can give all permission to read, write and execute to owner and group (so "7"), and only the permission of read a file to others (so "4") by executing the following command :

```
ninux@ale:~$ chmod 774 example.txt
ninux@ale:~$ ls -l example.txt
-rwxrwxr-- 1 me mygroup 0 2012-01-19 12:25 example.txt
```

There are other permission like sticky bit, setguid and setguid, but we do not address this topic now.

4.4 Mounting

We conclude by presenting the `mount` and the `unmount` commands. Basically all files accessible in a Unix system are arranged in one big tree, the file hierarchy, rooted at /. These files can be spread out over several devices. The mount command serves to attach the filesystem found on some device to the big file tree. Conversely, the `umount` command will detach it again.

`mount -t type device dir` tells the kernel to attach the filesystem found on device (which is of type type) at the directory dir. The previous contents (if any) and owner and mode of dir become invisible, and as long as this filesystem remains mounted, the pathname dir refers to the root of the filesystem on device.

As network administrators we could need to mount remote directory (for instance a shared directory) in the local filesystem.

4.5 Archiving, compression and decompression

It is not rare to download from the Internet a compressed file such as `filename.tar.gz`.
The `tar` utility allow us to create archives (tar) that put inside just one file, one or several directories an the files that are within.
The same utility allows also to compress this archive using different compression algorithms, where probably the most commons one are *gzip* and *bzip*.

To create a compressed archive of a directory we can just type `tar cfvz nameofarchive.tar.gz target_dir`.
The options `cvfz` instruct `tar` to: `c` creates a new archive containing the target_dir , `v` produces a verbose output, `f` read the archive from or write the archive to the specified file and finally `z` compresses the resulting archive with gzip.

This is the same of creating an uncompressed archive with `tar cfv nameofarchive.tar target_dir`. and then compress it by calling the `gzip` program on it `gzip nameofarchive.tar`.

If we need a better compression we can resort to bzip so we need only to give the `j` options instead of the `z` options, to produce a `nameofarchive.tar.bz2` file. This is equal to call `bzip2 nameofarchive.tar`.

Usually gzip compressed files have extension `.gz`, bzip2 compressed files have extension `.bz2`, tar files have extension `.tar`. However to being sure if a file is compressed or not, we can just call the `file` command against that file that will give us a lot of information about that file:

```
aquilante:~ orazio$ file test.tar.gz
```

```
test.tar.gz: gzip compressed data, was "test.tar", from Unix, last modified: Tue Jan 24 13:2
```

This is very useful for doing backup or moving data easily across the network. Compression has an important role in archiving *logs*. Several programs (e.g. the popular web server *apache*, but also the linux kernel by itself) print information (log) on a files tha are typically stored in the `/var/log` directory. These files are text files and are very useful for troubleshooting, however these logs can grow in size a lot. Compressing old logs files result in a space saving. The tools `zcat`, `zless` help us to read compressed (with gzip) text files, decompressing them on the fly while we are reading those.

5 Shell

When we type a given string to a terminal, it will be interpreted by our shell program.
To view which is our shell it is enough to write:

```
root@ale:~# echo $SHELL
/bin/bash
```

Indeed, usually the default shell is `bash`.
When we want to launch a program we need to provide to the shell the path where it can find the program (i.e. where it can find the executable file).
Suppose that our working directory is `/home/me` we want to call the `ls` command.
We can use the absolute path and invoking `/bin/ls`, or a relative path by typing `../../bin/ls`, or alternatively we can change our working directory `cd /bin` and than launch the program `./ls`. However usually we just write `ls`, so how our shell know where is the that binary? The shell variable `PATH` does the magic and specify an ordered list of directory in which the shell should try to find an executable if no path is provided.

```
root@ale:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

So when we type `ls` the shell search on `/usr/local/sbin` then on `/usr/local/bin` , `/usr/sbin` , `/usr/bin` , `/sbin` and finally it find it inside the `/bin` directory.
To view which is the pathname of the files (or links) which would be executed in the current environment we can type:

```
root@ale:~# which ls
/bin/ls
```

So far we see two different shell variables, PATH and SHELL but how you can guess there are plenty of them. You can see them by typing `set`. Moreover they are not the only variables on a system, if you type `env` you would see the list of the environmental variables. If you are interested refer to the related man pages.

Bash is a powerful shell that allows us to write simple and complex script such as "execute that program, if it fails parse the results and do other actions". For a guide to bash scripting plese refer to [2].

5.1 IO Channels

Under UNIX, all programs that run are given three open files when they are started by a shell: `stdin`, `stdout` and `stderr`. Standard input (or stdin) is where the input came from and normally points to our terminal device (that guess what? it is a file and we can discover it typing `tty`). We can redirect this input using the < character. For instance

```
grep yourusername < /etc/passwd
```

will pass to the command `grep username` the input contained in `/etc/passwd`. On the same way, standard output (or stdout) is where the output of a command will be driven to. It will It normally points at your terminal as well, but you can redirect it. If we type:

```
echo "hello world" > myfile
```

will write the output of the command `echo "hello world"` to `myfile`. Obviously the output of the echo program is to repeat the argument.

Finally, standard error (or stderr) is where error output from your program goes. This normally points at your terminal as well, but you can redirect it. All these "IO Channels" are mapped into small integer number (file descriptor or FD) by the linux kernel when we execute a program: by default STDIN is FD 0, while STDOUT is FD 1, and STDERR is FD 2 (if not redirected).

Now, there are lots of redirection symbols that you can use, and here are some of them:

$<$ file means open a file for reading and associate with STDIN.

$<<$ token Means use the current input stream as STDIN for the program until token is seen. We will ignore this one until we get to scripting.

$>$ file means open a file for writing and truncate it and associate it with STDOUT.

$>>$ file means open a file for writing and seek to the end and associate it with STDOUT. This is how you append to a file using a redirect.

$n > \&m$ means redirect FD n to the same places as FD m. Eg, $2 > \&1$ means send STDERR to the same place that STDOUT is going to.

5.2 Pipelines

Pipes allow separate processes to communicate without having been designed explicitly to work together. This allows tools quite narrow in their function to be combined in complex ways. For instance:

```
ls | grep x
```

redirect the output of `ls` to the the input of the `grep` command.

5.3 Tricks

Several tricks on the shell could make your life easier. The easiest one is using arrows UP and DOWN to scroll between last given commands, but there are many more. For instance by pressing `CTRL+a` and `CTRL+e` you can go to th beginning or to the end of your command, while `CTRL+k` will delete all th charachers from the cursor to the end of the command.
If you press `ESC + .` the last argument of your last command will be returned and this is very useful when you provide for instance a long filepath as an argument of a program and this happens not so rarely. More "tricks" are available in the `readline` man page.
It is worth to present also the utility `history`. As the name suggests, its goal is to show the last commands. If you want to repeat one command, it suffices to write `!NUMBER` where the `number` is provided by `history` aside each command.

6 Process

Each program that run on a Linux machine is a process. To obtain the list of running processes it suffices to do:

```
ninux@ale:~$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  24004  1964 ?        Ss    2011   0:02 /sbin/init
root         2  0.0  0.0      0     0 ?        S     2011   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S     2011   0:39 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S     2011   0:00 [kworker/u:0]
root         6  0.0  0.0      0     0 ?        S     2011   0:00 [migration/0]
root         7  0.0  0.0      0     0 ?        S<    2011   0:00 [cpuset]
root         8  0.0  0.0      0     0 ?        S<    2011   0:00 [khelper]
root         9  0.0  0.0      0     0 ?        S<    2011   0:00 [netns]
[....]
```

The `ps` command report a snapshot of current process running, displaying some information like the user that run that program, the percentage of memory occupation and the cpu usage, the command that create that process and the PID. The PID is an unique number, called process id, associated to every process. Note that if we execute several time the same binary file, Linux assign to every running instance a different pid. Pid are again an usefull way to instruct linux to do something to a given process, such as terminating or whatever.

To brutally kill a process we can type `kill -9 PID` where pid is the pid of the process we want to kill. This command will send a signal (SIGKILL) to that piece of code and force it to disconnect so it is not so nice to do that (imagine the case in which we kill a process while it is writing to a configuration file ...).

A more polite signal to send to a process to terminate it (and that could be handled) it is the so called SIGTERM to a process `kill -15 PID` . This will cause a graceful shutdown. As usual `man kill` give us useful information on that command. SIGKILL is the only one signal that can not be handled. This means that when you write a program, you can perform special tasks in response to a given signal; these functions are called *handlers*.

Signals are particularly important for speaking with process that run in background (so called "daemons"). Almost all the network services or routing are implemented inside daemons.
For instance when you write a routing daemon, you may want to intercept SIGTERM so to clean the routing table before quitting your daemon.

To continuosly view the running processes, the `top` comes in handy. Type ”q” to quit the top command, or open another shell and kill -9 it! :-)

We finish saying that every process have a ”niceness” that is a scheduling priority that is a number between -20 (most favorable scheduling) to 19 (least favorable). We can set this parameter using the `nice` command. It could be useful when we want to tune some process with hard time requirements on a low power machine.

6.1 Foreground, Background and Screen

When you launch a program, you can append a `&` char to the command line to send it to the background. Let we make an example and launch a program called `tcpdump`. This program will be described later on, however it prints on screen information about packets it see on the network, with the option `ip6` it will display only ipv6 packets, so:

```
root@ale:~# tcpdump ip6
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

As we see, if we launch this program it will inhibits any other commands. We have to interrupt the program (`ctrl+c`) to go back to our shell. If we launch the program with the string`tcpdump ip6 &` we launch the program in the background and we can execute more commands while the program is running (we can check its status using the `ps` command). To quit that program we can *kil*l it or we can type `fg` (foreground). The meaning of this last program should be auto-explicative.
Now suppose we need to run a command in a shell and keep running the command even after we close the session. The trick we learned so far (background and foreground) are not useful, so we have to resort to other commands. There are several way to do that such as `nohup` and `disown` , but probably the most *comfortable* is to use `screen`.

With screen we can execute a command on a *virtual terminal* (screen), than we can detach from that shell and disconnect ourself from the machine. That screen will remain alive and when we connect again to the machine we can re attach to that screen and resume interacting.
Let we try to create a new *named* screen by executing *screen -S myname*. A new shell will show up. Now we can execute our testing program *tcpdump ip6* and then press *CTRL+a* (together) and after *d* (detach). *CTRL+a* precedes any screen commands. Now we can log-off from the machine and then login

again. To resume our screen we need to write *screen -r myname*.

Inside screen we can create more virtual terminal by digit *CTRL+a* and then *c* (create), navigate between the consoles with *CTRL+a* and *n* (next) or *CTRL+a* and *p* (preview). To quit a screen just type *exit*. Quit is different from detach!

7 Packaging tool and program installation

One of the popular action that we have to do on a Linux machine is to install a new program. Usually we can do that in two different way: by compiling the source code and compile and install it, or by using the packaging tool shipped with our distribution (if any).

In the first case the usual procedure relies on the so called *autotools* [3] that are a suite of programming tools designed to assist in making source-code packages portable to many Unix-like systems.
For instance suppose we want to compile `tinc`, a popular VPN software.
We grab from the web the link of the source and then we go in the `/usr/src` directory and download it:

```
root@ale:/usr/src# wget http://tinc-vpn.org/packages/tinc-1.0.16.tar.gz
--2012-01-20 15:11:23--  http://tinc-vpn.org/packages/tinc-1.0.16.tar.gz
Resolving tinc-vpn.org... 137.56.127.68
Connecting to tinc-vpn.org|137.56.127.68|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 519460 (507K) [application/x-gzip]
Saving to: ‘tinc-1.0.16.tar.gz’

100%[=========================>] 519,460     1.52M/s   in 0.3s

2012-01-20 15:11:24 (1.52 MB/s) - ‘tinc-1.0.16.tar.gz’ saved [519460/519460]
```

Then we decompress the archive :

```
root@ale:/usr/src# tar xvfz tinc-1.0.16.tar.gz
```

This will create a new dir (in our case `/usr/src/tinc-1.0.16`). We enter inside this directory and we find an `INSTALL` file. Usually programs have `INSTALL` or `README` file with instructions and it is always a good a idea to read them. In this case the procedure to install `tinc` is quite standard. We should run the

`./configure` script that will search in our system for the presence of the right libraries needed by the program. If the configure went fine, we can type `make`. This will invoce the `Makefile` that is responsible for compiling the binary. If also this step is accomplished, we can install the program by giving a `make install`. This last command will invoke the same Makefile but with a different *target* and it will call the piece of the script that copy the binaries, configuration files etc, in our system.

Using this procedure we can compile and install every program under Linux, holding the maximum control on the compiling options, optimizing it for our CPU architecture, and being sure we downloaded the right version (e.g. the latest). However it is not rare that programs have *dependencies* that is need other programs to being installed.
In some cases these dependencies are a lot (20, 30 libraries or other programs), so a lot of distributions provide a way to simplify this operation. On Debian-like sistem the `apt` utility come in handy. It connect to *repositories* where there are already compiled program and handle dependencies. With `apt-cache update` we download the list of available programs from repositories. Then we can search for our program with `apt-cache search tinc` and find the right package. After that we can simply install it (and all the needed dependencies!) with`apt-get install tinc`. That's it!

8 Editors

There are plenty of different text editors under Linux. Choosing the right text editor it is a matter of personal taste and sometimes there are religion wars against different editors fans, so that a profane could ask why so much argue for a text editor.
One straight response is that because if you code, configure, write documentation and administrate a Linux machine, you have to spend a lot of time using a text editor, so the better is your confidence with that, the lesser is the time you will spend.
In what follows we focus on `vim` that was one of the first one (it was designed for Qwerty keyboards that contains no arrow keys!) and it is shipped with many systems included the several embedded (e.g. linux routers) . It could results a little bit hard at the very first look, however after a little bit of practice we will be comfortable and `vim` allow us to go super fast.
`vim` stands for `Vi iMproved`, while `vi` is original editor. In many system, you can launch vim with `vi` or `vim` as well (sometimes there is an alias by default in bash that substitute `vi` with `vim`). Conversely, other system are shipped with `vi` and we need to install the `vim` package to exploit the advanced functionalities of this editor (e.g. `apt-get install vim`). In what follows we call `vi` to refer to the improved version.

Let we start getting practice with an example.
On a shell type `vi test`. The editor will open.
Now type press `i` and the you can write "Hello World". Then press the `<Esc>` button and then the fancy combination `:wq`.
As you can see, `vi` has several *modes* (six basic modes and other advanced modes): when you started the editor and after that you pressed the `<Esc>`button you were in the Normal (or Command) mode.
When in *Normal Mode* you pressed `i` you entered into the *Insert Mode*.
If for any reason you do not know which mode you are in, you can always get back to Normal Mode by typing `<Esc>` twice.
In Normal Mode, with `:w` we can save the file, while with `:q` we can quit. In the last example we combine these two instructions in `:wq` (or `:x` if you are in hurry). In you want to quit without saving, just type `:q!`

We will not mention all the possible commands and modes. We limit to the few one that we need to our purpose, however if you want to go fast just check the manual or the plenty of documentation available online.

Now re-open the same file by writing `vi test` and you will be in Normal Mode. If you press `o` you will start inserting a new line under the current line (Insert Mode). So now you can type "I said Hello World", than go in press `<ESC>` to come back in Normal Mode and than press `:w` to save our masterpiece. Than with arrows [1] go on the first line and press `dd`. Now if we have some regrets we can just press `u` to Undo the last action.

Ok, now fill the file with some lines. Try, in normal mode, to write `100 i` , then write a sentence (for instance "Test" and then `<Enter>` to get a new line), and then press `<ESC>` . You should see your line repeated one hundred of times.
Now you can go to line 42 by typing in Normal Mode `:42`. You don't believe that this is the right line? Just type `:set number` to see line numbers (`:set nonumber` to hide them).
Now copy that line by pressing `yy` (yank) go wherever you want inside the file and then press `p` (paste) to paste your copied line.

If you want to delete a single char it suffices to press `x`. If you want to delete 200 chars, just do `200x`.

To search for a word inside the document type `/WORDTOSEARCH` and then browse occurrence of the found strings (if any) by using the keys `n` (next) and `p` (previous).

[1] A lot of people claim that Vi users should not use arrows, just `h j k l` for matter of speed and portability however, for our purpose, arrows probably are easier

Actually the expression that you can search can be more complex, for instance you can tell `vi` to search the word "World" just if it happens at the end of one line. In this case you can just search for the "expression" `/World$` (`$` means end of line, while `^` means begin of a line). The search command in `vi` accept the so called *Regular Expressions* that are way a powerful way to match string in a text (e.g. find a word of 5 char that start with a *w*). Writing regular expressions are not so much intuitive, however after a little of practice they will completely repay your efforts. If you are interested, you can search for that in the web and many web sites will pop up such this one [4].

Finally it could be also useful to know how to substitute all the occurrences of *wordA* with *wordB*, we just need to type `:%s/wordA/wordB/g`

References

[1] `http://www.gnu.org`

[2] `http://tldp.org/LDP/abs/html/`

[3] `http://en.wikipedia.org/wiki/GNU_{}build_{}system`

[4] `http://vimregex.com/`