Linux Packages Management:

Install and manage packages and tools is a very important skill, fundamental for every day tasks working with any kind of OS. Everything that we install in a Linux based OS is contained in a package, from a tool to a browser or even Minecraft.

We install packages with a program called a package manager. Of course, there is not only one package manager. There are couple of them and each one has its own strengths and weak points. In most of the Linux OSs we find two of them all the time. One is called **dpkg** (short for de-package). The other one, easiest I must say, is called **apt** (short for advanced package tool).

dpkg:

This is a low-level package manager. This package manager has two main flaws. First of all, in order to install a package, we have to go look for It on the internet, we have to open the browser and go to the website of for example “Discord” and download the package for Linux. What we do here is just download a file with a “.deb” extension. This extension means that this is a Linux package, more specifically, a Debian-based package.

The second flaw of this package manager is that most pf the time a package depends on another package to function. Usually, we’d expect that all comes together, but it doesn’t. So, in the middle of the installation, we’ll find that we have something called **“dependency problems”**. So, if we want to install those dependencies, we’d have to go over the internet, find them, download them, install them, and then if they have dependencies we’d have to repeat the process till there are no more dependencies.

apt:

As its name indicates it, is an advanced, package manager. Apt solves both flaws of the dpkg manager. For most of the cases we will only use the command and then we reference the name of the package we want to install, and that’s it. What this package manager does is that it goes to a site where the most of packages are, searches for the package we requested and also see what the dependencies are and installs all of them too.

New Commands:

APT

**apt update**: This command we use it to get a **list** of any **new** **updates** from the **repositories** that we use to download the packages.

**apt list**: This command **returns a list** with all the **packages** we have **installed**. If we use **–-upgradable** we’ll be upgrading all the **packages** that can be **upgraded**

apt list

apt list --upgradable

**apt install**: This is the main command we’ll use to **install** any **new packages** in the system. The **--fix-broken** fixes a **previous** **installation** that **returned** this **error**.

apt install [package name]

apt –-fix-broken install

**apt reinstall**: **Reinstalls** a package

**apt upgrade**: **upgrade** the **system** by **installing**/**upgrading** packages

**apt full-upgrade**: **upgrade** the **system** by **removing** old apps and **installing**/**upgrading** packages

{…}APT

**apt edit-sources**: **Edits** the **sources** that we use as repositories to install or upgrade our packages or the system

**apt satisfy**: We give to it a **string** of **dependencies** and apt **installs** those **dependencies**

**apt remove**: This command **removes** **a** **package** from the installed packages. This way of removing only removes the application, not your user data so if you install it again you can have access to your old data.

**apt purge**: This command **removes** **a** **package** from the installed packages. This way of removing removes the application and your user data.

snapd:

Snapd is a “store” or a repository where we can upload our apps and packages so other people can have access to them. The same way we can gain access to other people’s work by using it. Of course, we’d have to install it and we can do that very easily with **“apt install snapd”**. We can get very useful apps from this store like Visual Studio Code for example (consider that we need the

“ --classic ” switch and the name which is “code”)

Git and Pip:

We already looked into these commands. Those are commands specifics to programing languages. Git is used basically to clone other people’s repository while Pip or Pip3 is used specifically for Python dependences

Daemons (Services):

Every time we launch any program, one or more (often more) processes are launched. This is because the programs depend broadly on its processes to work. What we call Daemons, are the processes that the Kernel starts by its own in order to maintain the OS working. We also call the daemons, services.

How can we identify if a process is a daemon?

These processes usually have a letter **“d”** at the end of the process’s name. For example, if we look for the ssh process by grepping it from the **“ps -aux”** this is what we’d see:

We can control all these daemons by using a process that’d be something like the master daemon. Its name is **systemd**.

┌──(alice㉿Alice)-[~]

└─$ pstree

systemd─┬─ModemManager───2\*[{ModemManager}]

├─NetworkManager───2\*[{NetworkManager}]

├─3\*[VBoxClient───VBoxClient───2\*[{VBoxClient}]]

├─VBoxClient───VBoxClient───3\*[{VBoxClient}]

├─VBoxService───8\*[{VBoxService}]

├─agetty

├─colord───2\*[{colord}]

├─cron

├─dbus-daemon

├─haveged

├─lightdm─┬─Xorg───{Xorg}

├─systemd-journal

├─systemd-logind

├─systemd-udevd

├─udisksd───4\*[{udisksd}]

├─upowerd───2\*[{upowerd}]

└─xcape───{xcape}

Systemd:

This is the service manager, the first process, the daemon master, the file system mounter, the one who starts all services. When we boot our Terminal, this talks to the Linux kernel, then the kernel starts the first process, systemd. Once this process is started, it uses one of its processes called forking that’s used to start all the other processes.

When systemd refers to a daemon or a service, it calls them **units**.

New Commands:

systemctl:

**systemctl start**: This command **starts a determined process**

[systemctl] [start] [process name]

**systemctl stop**: This command **stops a determined process**

[systemctl] [stop] [process name]

**systemctl restart**: This command **restarts a determined process**. Some processes also support the reload functionality so we can use the **reload** or **restart** command.

[systemctl] [restart] [process name]

[systemctl] [reload-or-restart] [process name]

**systemctl status**: This command **shows the status a determined process**

[systemctl] [status] [process name]

{…}systemctl:

**systemctl disable**: This command **stops a process from loading at boot**. This won’t stop the daemon, but it will prevent it from starting by itself next time we boot the system.

[systemctl] [disable] [process name]

**systemctl enable**: This command **starts a process at boot**. This won’t start the daemon, but it will start it next time we boot the system.

systemctl enable [process name]

How to know if they’re active and enabled:

[systemctl] [is-enable] [process name]

[systemctl] [is-active] [process name]

**systemctl list-units**: This command will deliver a **list of all the available units**. On the other hand, we can see the active services by using the switch t (type).

[Systemctl] [list-units]

[Systemctl] [list units] [-t service]

Managing Processes:

Usually when we have a problem in windows, we just open the process manager, or task manager and if something is wrong, we just kill the process that’s working off. In Linux, we do it a bit different, through the command line, with commands, of course, is Linux duh. We’ll continue to using the ps commands but we’ll go deeper so first let’s explain how do we stop or “kill” a process and all the ways that we have to do it.

As we said before, every process has its own ID, is the number that they receive when they are born (remember that systemd has ID=1 because he’s the firstborn child). We can use this ID to reference a process and manage it or stop it from working.

Workflow:

So, the workflow here is to first get the process’s ID with the **ps** command in any of its variations, and then use the **kill** command in any of its variations to kill it.

Types of Processes:

**Foreground Processes:**

These are the ones that we can see, like for example the ones that we start manually in the command prompt (ps, sleep, ping etc…). Usually, we know is a foreground process because we can kill it or stop it using **[Ctrl]+C**, which is the **stop** version of **kill**, the default one.

We can put the process to sleep using **[Ctrl]+Z** and then keep using the same terminal.

We can see our processes started in the terminal, or “jobs” using the command **jobs**

**Foreground to Background:**

We can easily send a process from the foreground to the Background by usingits job ID, which we can see when we use the **job** command and passing it to the **bg** command.

**IMPORTANT:** When you turn it to a background process you won’t be able to stop it from here. You have to turn it back to a foreground process by using the **fg** command.

“Task Manager” on Linux’s Terminal:

The same way as we have it on windows, we can also have it on Linux by using two commands. One gives us a simple table in the command line while the other it is also in the command line, but is more graphical:

top:

┌──(alice㉿Alice)-[~]

└─$ top

top - 13:29:35 up 4:09, 1 user, load average: 0.06, 0.02, 0.00

Tasks: 132 total, 2 running, 130 sleeping, 0 stopped, 0 zombie

%Cpu(s): 5.1 us, 4.1 sy, 0.0 ni, 90.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

MiB Mem : 3929.8 total, 2835.7 free, 586.0 used, 508.2 buff/cache

MiB Swap: 975.0 total, 975.0 free, 0.0 used. 3104.2 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

510 root 20 0 545368 181780 70112 S 4.7 4.5 0:40.36 Xorg

1024 alice 20 0 433792 103424 84456 S 2.0 2.6 0:04.81 qterminal

960 alice 20 0 203872 28600 18796 S 0.7 0.7 1:10.02 panel-13-cpugra

912 alice 20 0 629652 90372 69520 S 0.3 2.2 0:23.41 xfwm4

953 alice 20 0 344112 27392 17956 S 0.3 0.7 0:00.09 Thunar

1043 alice 20 0 476920 44724 32904 S 0.3 1.1 0:00.22 polkit-gnome-au

1 root 20 0 101572 11528 8416 S 0.0 0.3 0:03.34 systemd

2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd

3 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu\_gp

htop:

Graphical user interface, application

Description automatically generated

New Commands:

Ps:

**ps**: This command by its own will only show you the processes that are open in the foreground, the ones that you can see.

[ps]

Simplified Switches for [ps]

┌──(alice㉿Alice)-[~]

└─$ ps --help simple

Usage:

ps [options]

Basic options:

-A, -e all processes

-a all with tty, except session leaders

a all with tty, including other users

-d all except session leaders

-N, --deselect negate selection

r only running processes

T all processes on this terminal

x processes without controlling ttys

**kill**: This command sends a signal to the process. This signal varies depending on the switch that we use when we reference the process. Meaning that we can not only stop it but put it to sleep and other bunch of options.

[kill] [process ID]

Simplified Signals for [kill]

┌──(alice㉿Alice)-[~]

└─$ kill -l

HUP INT QUIT ILL TRAP IOT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR SYS

Main kill signals:

There are a lot of kill signals but right now we will focus on the ones that we use the most:

**SIGTERM**: This is the default one, not mandatory. **kill**

**SIGSTOP**: This is one just stops the process. It doesn’t kill it. **[Ctrl]+[Z] (once)**

**SIGCONT**: This is one continues a stopped process. **[Ctrl]+[Z] (second time)**

**SIGINT**: This is one interrupts it. not mandatory. **[Ctrl]+[C]**

**SIGKILL**: This is one kills the process. Mandatory **kill -9**

**pkill**: This is like **pgrep** that returns you the name by grepping it. What it does is that instead of the processes ID, you give it a name. This way if you have 900 pings, you can kill them all at the same time. The same switches from kill are to apply.