

 Code *With* Harry

ULTIMATE LINUX

H A N D B O O K



YOUR COMPLETE GUIDE TO
LINUX FROM BASICS TO
ADVANCED SYSTEM MASTERY

```
harry@linux:~$ ls -la
```

```
total 36
```

```
drwxr-xr-x  5 harry harry  4096 May 12 10:15 .
drwxr-xr-x  3 root  root   4096 May 10 09:30 ..
drwxr-xr-x  2 harry harry  4096 May 11 14:22 Desktop
drwxr-xr-x  2 harry harry  4096 May 11 14:22 Documents
drwxr-xr-x  2 harry harry  4096 May 11 14:22 Downloads
-rw-r--r--  1 harry harry   220 May 10 09:30 .bash_logout
-rw-r--r--  1 harry harry  3771 May 10 09:30 .bashrc
-rw-r--r--  1 harry harry   807 May 10 09:30 .profile
```

```
harry@linux:~$ _
```



CLEAR
EXPLANATIONS



PRACTICAL
EXAMPLES



SYSTEM
ADMINISTRATION



SECURITY
BEST PRACTICES



PERFECT FOR
BEGINNERS TO
ADVANCED



EASY TO
UNDERSTAND
CONCEPTS



HANDS-ON
COMMAND
PRACTICE



MODERN
TOOLS &
WORKFLOWS



REAL-WORLD
USE CASES



BUILD PROJECTS
WITH CONFIDENCE



Linux Handbook

Introduction

SECTION 01

The handbook starts from zero: what Linux is, how it grew, and how to get a working Linux environment. After that it moves into daily skills: commands, users, permissions, processes, the filesystem, and a first look at Nginx and file transfer methods.

Why Linux Shows Up Everywhere

Linux is used in servers, databases, and supercomputers. Those are the main reasons people learn Linux, but they are not the only ones. Linux is also the default environment in cloud computing, containers, and most Android phones.

OPEN SOURCE

You can run, study, and modify Linux without paying a license fee for the operating system itself.

SERVERS

Most web, database, and cloud infrastructure runs on Linux because it is stable, scriptable, and cheap to operate.

ANDROID

The Android operating system is built on a Linux kernel. A phone is not "Ubuntu", but the kernel underneath is still Linux.

CONTROL

Linux gives you a permission model, a process model, and a filesystem you can inspect and change from the terminal.

When people say "Linux is free", they usually mean two things at once: the kernel is open source, and you do not pay a vendor for the OS license. You may still pay for a VPS, support, or commercial add-ons. The operating system itself is not the expensive part.

Place	What actually runs	Why Linux is used
Web and database servers	A Linux distribution (often Ubuntu, Debian, or RHEL)	Stable, remote-friendly, no OS license cost
Supercomputers	Linux on specialized hardware	Custom kernels, open tooling, cluster software
Android phones	Android on a Linux kernel	Hardware support and a free kernel
Your laptop or VM	Ubuntu, Fedora, or similar	Same skills as the server, easier to practice

Kernel, Operating System, Distribution

Beginners often mix up three words. Keep this table nearby. The rest of the handbook uses these words constantly.

Term	What it is	Everyday example
Kernel	The core program that talks to CPU, RAM, disk, and devices	The Linux kernel
Operating system	Kernel plus the software that lets you actually use a computer	Ubuntu, Fedora, Windows, macOS
Distribution (distro)	A packaged Linux kernel plus a chosen set of tools, desktop, and defaults	Ubuntu, Fedora, Kali Linux, Kubuntu

Strictly speaking, Linux is the kernel. In conversation, "Linux" almost always means a distribution: Ubuntu on a laptop, Debian on a VPS, Kali in a security lab. Both statements can be true at the same time.

Rule of thumb: If someone says "install Linux", they mean install a distribution. If someone says "the Linux kernel", they mean the core that schedules CPU time, manages memory, and talks to hardware.

The kernel's job, in one sentence: decide which program gets the CPU, how RAM is shared, what is on disk, and how a program is allowed to run. You will not configure the kernel in this series. You will keep seeing its fingerprints: process lists, permissions, device files, and services.

Confirm You Are on Linux

These commands are the practical check. Run them after you have any Linux environment (a VPS, WSL, or a virtual machine). Each group shows three variations of the same idea.

Print the kernel name, then more detail, then only the kernel release:

```
uname -s
uname -a
uname -r
```

`uname -s` is usually `Linux`. `uname -a` dumps architecture, hostname, and kernel version in one line.

`uname -r` is the short kernel release string you will see in docs and bug reports.

Identify the distribution, not just the kernel:

```
cat /etc/os-release
hostnamectl
cat /etc/issue
```

`/etc/os-release` is the reliable file on modern distros. `hostnamectl` (systemd machines) also shows the operating system and kernel. `/etc/issue` is the short login banner; it is less complete, but you will see it on some servers.

Check who you are, where you are, and which machine you are on:

```
whoami
pwd
hostname
```

`whoami` prints the current user. `pwd` prints the current directory. `hostname` prints the machine name. Together they answer the three questions you should ask the first time you land on a Linux box.

If `uname -s` prints `Linux` but `/etc/os-release` says `Ubuntu`, you are looking at a distribution. That is the kernel-versus-distro distinction, visible in two commands.

Distributions You Will Hear About

Ubuntu, Fedora, Kubuntu, and Kali Linux are the names you will hear first. Those are not the only options, and they are not interchangeable. Use this as a chooser, not as a shopping list you must memorize.

Distribution	Family	Typical use	Desktop notes
Ubuntu	Debian	Beginners, cloud VMs, tutorials	GNOME by default; huge community
Kubuntu	Debian (Ubuntu)	Same Ubuntu base, different desktop	KDE Plasma instead of GNOME
Fedora	Red Hat	Newer packages, closer to RHEL	Often used by people who want recent software
Kali Linux	Debian	Security testing and labs	Not a daily-driver OS for beginners
Debian	Debian	Stable servers	Ubuntu is based on Debian
Arch Linux	Independent	Rolling release, DIY setup	Powerful, not the first distro to learn on

UBUNTU

The default recommendation in this series for a first server or VM.

FEDORA

A good next distro if you want newer packages and Red Hat-style tools (`dnf` instead of `apt`).

KALI LINUX

Built for penetration testing. Use it for that job. Do not install it as your first "learn Linux" system.

KUBUNTU

Ubuntu with a different desktop. Same commands, different look.

Package commands differ by family. You do not need these yet, but you will meet them the moment you install software. Three ways to update package lists, depending on the distro:

```
sudo apt update
sudo dnf check-update
sudo pacman -Sy
```

`apt` is Debian and Ubuntu. `dnf` is Fedora and related Red Hat systems. `pacman` is Arch. Same intent, different tool.

Kali is popular for security work. That does not make it the right first install. Learn Ubuntu (or Debian) first, then open Kali when you actually need its tools.

How to Follow Along

You will get a Linux environment in the next few chapters: an online VPS, VirtualBox on Windows or Mac, or WSL on Windows. Until then, you only need two habits.

1. Read for the "why", then run the commands on a real machine.
2. Keep the wording nearby: kernel, distro, VPS, package manager.
3. When a command appears, try the extra variations, not only the first one.

Linux commands are case-sensitive. `Unames` is not `uname`. Filenames such as `/etc/os-release` must be typed as shown.

If you already have Linux open, run the three `uname` variations now and keep the output. Later chapters will keep asking the same question in richer forms: which kernel, which distro, which user, which process.



What is Linux?

Linux is the kernel at the center of many operating systems. In everyday speech, people still call the whole system "Linux". Both uses of the word are common. This chapter separates them, then shows what the kernel actually does on a running machine.

Hardware → Kernel → Shell → Programs

What an Operating System Does

An operating system sits between hardware and the programs you want to run. The hardware is the CPU, RAM, disks, and devices. Programs cannot talk to that hardware safely on their own. The operating system is the layer that makes the machine usable: it starts processes, shares memory, reads files, and decides who is allowed to do what.

Without an operating system, a computer is a box of parts. With one, you can log in, run a command, save a file, and connect to a network.

Windows, macOS, and Ubuntu are operating systems. They all do the same kind of job. They do not do it the same way.

Linux Is a Kernel

Strictly speaking, Linux is not an operating system. Linux is a kernel: the core program that talks to the hardware and keeps every other program in line.

The kernel decides:

CPU TIME

which program runs next, and for how long

MEMORY

how RAM is split across programs, and when it is taken back

STORAGE

how much disk is used, and how blocks on a drive become files

FILES

what exists, how it is listed, and who may open it

PROCESSES

how a program is started, stopped, and isolated from others

You do not configure the kernel in this handbook. You will keep meeting its work: process lists, permissions, device files, and services.

Linux vs "Linux": If someone says "Linux is a kernel", they are being precise. If someone says "this server runs Linux", they mean a full operating system built around that kernel. The second usage is the normal one.

Inspect the kernel on any Linux machine. Print the kernel name, then the full uname line, then only the release:

```
uname -s
uname -a
uname -r
```

`uname -s` should print `Linux`. `uname -a` adds hostname, kernel version, and architecture. `uname -r` is the short release string used in docs and bug reports.

See the same fact from three files and tools:

```
cat /proc/version
hostnamectl
uname -v
```

`/proc/version` is published by the kernel itself and includes the compiler used to build it. `hostnamectl` (on systemd machines) shows kernel and operating system together. `uname -v` prints the kernel build identifier.

What the Kernel Manages on Your Machine

The jobs above are not abstract. Each one has a command you can run.

CPU: how many cores the kernel can schedule, and how they are described.

```
nproc
lscpu
cat /proc/cpuinfo
```

`nproc` prints a single number: usable processors. `lscpu` summarizes architecture, cores, and model. `/proc/cpuinfo` is the raw per-core listing.

Memory: what the kernel has given out, and what is still free.

```
free -h
cat /proc/meminfo
vmstat
```

`free -h` is the readable summary (total, used, available). `/proc/meminfo` is the detailed kernel view. `vmstat` adds a live snapshot of memory, swap, and run queue.

Storage: mounted filesystems, block devices, and space in one directory.

```
df -h
lsblk
du -sh /home
```

`df -h` shows mounted filesystems and free space. `lsblk` lists disks and partitions as the kernel sees them. `du -sh /home` reports how much one tree is using. Change `/home` to any path you care about.

`/proc` is not a normal folder of files on disk. It is a window the kernel provides so user programs can read live system state. `cat /proc/meminfo` is asking the kernel a question, not opening a document.

Linux as People Actually Use the Word

When someone says "your machine has Linux, and this one has Windows", they are comparing operating systems. They are not talking about kernels in the strict sense. They mean a Linux distribution on one side and Windows on the other.

The same thing happens with "install Linux". The precise reply is "Linux is a kernel; install a distribution". In practice, the person wants Ubuntu, Fedora, or another complete system.

Both statements can be true:

Phrase	Precise meaning	Usual meaning
"Linux is a kernel"	The Linux kernel only	A correction, not a product name
"This server runs Linux"	A distribution using that kernel	Ubuntu, Debian, Fedora, and similar
"Install Linux"	Install a distribution	Pick Ubuntu unless you have a reason not to

Use the precise wording when you need it (kernel modules, kernel versions, Android's kernel). Use the everyday wording the rest of the time. This handbook does the same: "Linux" means the system you log into, unless a sentence is clearly about the kernel.

What a Distribution Is

A distribution (distro) is the Linux kernel plus a large collection of software chosen so you can actually operate a computer: a package manager, a shell, system tools, optional desktop, and default configuration.

Kernel + GNU tools + package manager + defaults = a distribution

Ubuntu is the example most beginners meet first. It ships the Linux kernel, the GNU userland (shell, core utilities), the `apt` package manager, and a set of defaults aimed at desktops and cloud VMs.

Distributions differ in four places that you will feel immediately:

PACKAGE MANAGER

`apt` on Ubuntu and Debian, `dnf` on Fedora, `pacman` on Arch

RELEASE STYLE

Ubuntu LTS stays still for years; Fedora moves faster

DESKTOP (IF ANY)

Ubuntu uses GNOME; Kubuntu is the same Ubuntu base with KDE Plasma

PURPOSE

Ubuntu for general use, Kali for security labs, Fedora closer to Red Hat

Identify the distribution, not only the kernel:

```
cat /etc/os-release
lsb_release -a
cat /etc/issue
```

`/etc/os-release` is the reliable source on modern systems. `lsb_release -a` prints the same idea in an older format (install `lsb-release` if the command is missing). `/etc/issue` is the short login banner; useful, but incomplete.

Three ways to update package metadata, depending on the family:

```
sudo apt update
sudo dnf check-update
sudo pacman -Sy
```

Same intent, different tool. You will use one of these the moment you install software.

Distribution	Base	Package tool	First use
Ubuntu	Debian	apt	Servers, VMs, this handbook
Kubuntu	Ubuntu	apt	Ubuntu with a KDE desktop
Fedora	Red Hat	dnf	Newer packages, RHEL-like tools
Debian	Debian	apt	Conservative servers
Kali Linux	Debian	apt	Security testing, not a first daily OS
Arch Linux	Independent	pacman	Rolling release, do-it-yourself

Kali Linux is built for penetration testing and is popular in that field. It is a poor first install if your goal is to learn Linux. Start with Ubuntu or Debian. Open Kali when you need its tools.

Why Linux Is Worth Learning

Linux is the default operating system for servers and infrastructure. Web servers, databases, supercomputers, containers, and most cloud images run it. That is the practical reason to learn it: the machines you will SSH into already speak this language.

Open source is the other reason. You do not pay a license fee to use the operating system. Companies also like that they can inspect the source, keep the stack in-house, and avoid being locked to a single vendor's OS. The next chapter covers how that situation came about.

You can confirm a box is a Linux server, see its distro, and see its kernel in three short commands:

```
uname -s
cat /etc/os-release
hostnamectl
```

If `uname -s` is `Linux` and `/etc/os-release` says `Ubuntu`, you are looking at a distribution. The kernel is Linux. The operating system is Ubuntu. That is the whole distinction, visible on one machine.



History of Linux

Linux did not appear from nowhere. It inherited ideas from Unix, borrowed userland tools from GNU, and became a usable operating system because people were allowed to copy it, change it, and ship it for free. This chapter is the short path from Bell Labs in 1969 to the servers and phones that run Linux today.

Unix (1969) → GNU (1983) → Linux kernel (1991) → Distributions → Android and the cloud

Unix at Bell Labs

In 1969, Ken Thompson and Dennis Ritchie began Unix at Bell Labs. Unix was not "Linux, but older". It was the operating-system idea Linux later copied: a system more than one person could use at the same time, that could run more than one program at the same time, and that treated files, devices, and processes in a consistent way.

Those ideas still describe Linux:

MULTIUSER

more than one login, each with its own files and permissions

MULTITASKING

many processes sharing one CPU, scheduled by the kernel

HIERARCHICAL FILES

a single tree of directories, starting at `/`

PORTABLE TOOLS

small programs that read text, write text, and combine with pipes

Unix became the model universities and companies taught. C was developed alongside it, which is why so much systems software still looks like C. You do not need to learn Unix itself for this handbook. You do need to recognize that Linux's command line, permissions, and "everything has a path" design come from that line of work.

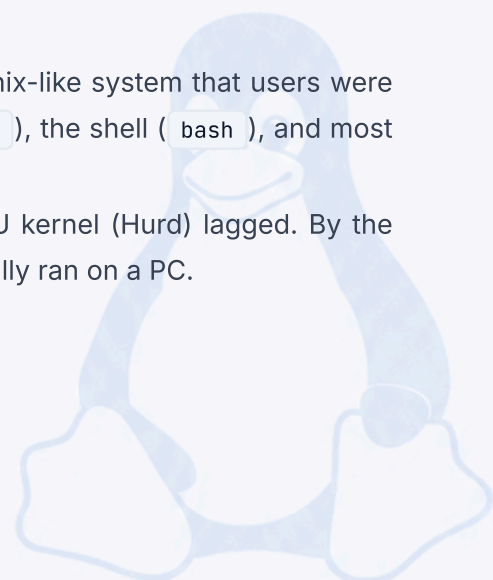
Linux is Unix-like. It is not Unix. The commands feel the same because the design is the same, not because Linux is a renamed Bell Labs product.

GNU and a Missing Kernel

In 1983, Richard Stallman started GNU: a project to build a complete Unix-like system that users were free to run, study, share, and change. GNU produced the compiler (`gcc`), the shell (`bash`), and most of the everyday utilities (`ls` , `cp` , `grep` , and the rest).

GNU did not, for a long time, have a finished kernel. The planned GNU kernel (Hurd) lagged. By the early 1990s there was a free userland waiting for a free kernel that actually ran on a PC.

That gap is why the next hobby project mattered.



A Hobby Project in Helsinki

In 1991, Linus Torvalds was a student at the University of Helsinki. He was using Minix, a Unix-like teaching system written by Andrew Tanenbaum. Minix worked. It did not let him use his PC hardware the way he wanted.

He announced a new operating system in the Minix discussion group: a personal project, not a product. The famous line was that it would not be "big and professional like GNU". It was a hobby, for 386 PCs, because he was interested in it.

Most hobby projects stay hobbies. This one did not.

In September 1991 he released the first version and made it open source. Other people could read it, change it, and ship their changes. The kernel grew because it was allowed to.

See the living result of that release on any Linux machine. Three ways to print which kernel you are running:

```
uname -r
cat /proc/version
hostnamectl
```

`uname -r` is the short version. `/proc/version` includes the build and compiler. `hostnamectl` places the kernel next to the operating system name. The numbers have changed since 0.01. The lineage has not.

GNU + Linux

The kernel needed programs around it: a compiler, a shell, file tools, a C library. GNU already had those. Combined with the Linux kernel, they made a complete operating system people could install and use.

That pairing is why some people say GNU/Linux. In this handbook, "Linux" still means the whole system, unless a sentence is clearly about the kernel. The history is the useful part: Linux became usable because a free kernel met a free userland.

Confirm you have that userland, not only a kernel:

```
bash --version
ls --version
gcc --version
```

`bash` and `ls` coming from GNU Coreutils is the usual case on Ubuntu and Debian. `gcc` may be missing until you install a compiler. The point is the split: kernel from Linux, many of the tools from GNU.

If `ls --version` mentions GNU coreutils, you are looking at that partnership on your disk. The kernel schedules `ls`. GNU wrote `ls`.

Distributions in the 1990s

Once the kernel could run a computer, people started packaging it with software of their own choosing. Those packages are distributions: a kernel, a userland, a package manager, and a set of defaults.

The early 1990s produced the first widely used distros (MCC Interim Linux, Slackware, Debian, Red Hat). Later names you already know sit on that same idea: Fedora (2003, community Red Hat), Ubuntu (2004, Debian-based), Kali Linux (2013, a Debian-based security distro). The casual list "Ubuntu, Fedora, Kali" is the present. The 1990s is when the pattern started.

Linux spread on the internet, on servers, and in universities for a simple reason: nobody had to pay to use it. Students could install it. Admins could clone it. Companies could run it without an OS license on every machine.

Year	What happened	Why it matters
1969	Unix begun at Bell Labs	Multiuser, multitasking model
1983	GNU project starts	Free compiler, shell, and core tools
1991	Linux 0.01 released	Free kernel that ran on a PC
1993 - 1994	Debian and Red Hat appear	Distros become the way people install Linux
2003	Fedora; Android Inc. founded	Community Red Hat; phones will use the kernel
2004	Ubuntu 4.10	The beginner-friendly Debian derivative
2005	Google acquires Android Inc.	Linux on nearly every Android phone

Debian and Fedora are two surviving families. Ubuntu is Debian with a different default experience. Kubuntu is Ubuntu with KDE instead of GNOME. Kali is Debian aimed at security work. The names change. The packaging idea does not.

Android and the Kernel in Your Pocket

In 2003, Android Inc. began building a mobile operating system on a Linux kernel. Google acquired the company in 2005. The product people call Android is not Ubuntu. It does not use `apt` or a normal GNU desktop. The kernel underneath is still Linux.

That is why "Linux is only for servers" is false. Servers are the reason most people in this handbook learn it. Phones are why the kernel ships in billions of devices. Supercomputers, routers, TVs, and cars sit in the same category: a Linux kernel plus a custom userland.

On a phone you usually cannot open a normal Ubuntu shell. On a server you can. The inspect commands from the previous chapter are for the server and VM path this handbook follows.

Why Companies Chose Linux

Large organizations moved servers to Linux for reasons that are not only "it is cheaper" or "it is faster". Open source meant they could run as many machines as they wanted without an operating-system license on each one. Source access meant they could audit what ran in their infrastructure instead of taking a vendor's binary as a black box. Control meant they could keep the stack in-house: patch on their schedule, keep old versions alive, and avoid being stuck on someone else's upgrade calendar.

That combination is privacy in the infrastructure sense: the company decides what the OS is allowed to do, and who is allowed to see it. Performance and features matter. Independence from a proprietary OS vendor is often the reason the migration starts.

The rest of this handbook assumes that world: you will log into a Linux machine, not because it is fashionable, but because that is what the servers already run.



What is a VPS?

A VPS (Virtual Private Server) is a Linux machine you rent in the cloud. You reach it over SSH. Commands you type run on that machine, not on your laptop.

It is cheap: billed for 1, 12, or 24 months. Pick Linux, not Windows. The OS has no license fee.

A VPS keeps your own computer clean. No local VM, no Linux install on the host. Use it for experiments, web apps, and cron jobs.

Pick Ubuntu LTS

This handbook uses Hostinger as the worked example. Other VPS hosts work the same way: a plan, Ubuntu, a root password, then SSH.

Plan	CPU	RAM	Disk
KVM 1	1	4 GB	smaller NVMe
KVM 2	2	8 GB	100 GB NVMe
KVM 4 / 8	more	more	more

KVM 2 for 24 months is a solid default. Choose the newest **Ubuntu LTS** (Long Term Support). Version numbers change (22.04, 24.04, 26.04, ...). LTS matters more than the number.

1. Select the plan and Ubuntu LTS. Apply a coupon if you have one.
2. Pay, then set a **root** password. Write it down. Resetting it later is possible; forgetting it is painful.
3. Skip extra add-ons if you do not need them. Finish setup and wait a few minutes.

SSH from your terminal

The panel gives two doors: **VPS Dashboard** and **SSH Access**. Copy the SSH command (or type it). The same steps work on Windows, macOS, and Linux.

```
ssh root@203.0.113.10
ssh -p 22 root@203.0.113.10
ssh -i ~/.ssh/id_rsa root@203.0.113.10
```

Use your real IP. The first form is what the panel usually copies. The second sets port 22 explicitly. The third uses a key instead of a password.

The first connection asks whether to trust the host. Type `yes`. Then paste the root password. Nothing is echoed as you type.

You should see `Welcome to Ubuntu` and a prompt like `root@srv...`. That prompt is the VPS.

```
hostname
uname -a
cat /etc/os-release
```

After login, every command runs on the rented machine. `ls` lists the VPS disk. Closing the laptop lid does not stop the server.

If something breaks

On the VPS page you can reset the root password, copy the SSH command again, or open the host's browser terminal. That browser terminal is the same machine as `ssh`.

[VPS Dashboard](#) → [SSH Access](#) → [browser terminal](#)



Installing Linux Through VirtualBox on Windows

What is virtualization?

Virtualization is a software technology that creates a second computer inside your laptop. That guest has its own RAM, disk, and OS. The laptop in front of you is the **host**. The Ubuntu box inside VirtualBox is the **guest**. They stay isolated.

Use this path only if the host is powerful. Otherwise rent a VPS (previous chapter). Give the VM as much RAM and CPU as you can spare.

Install VirtualBox

1. Search for **download VirtualBox for Windows**.
2. Open the official site and download **Windows hosts**.
3. Run the installer: Next → accept the license → Install → Finish.

It installs like any other Windows program. VirtualBox opens when it is done.

Create an Ubuntu VM

Install **Ubuntu LTS** (Long Term Support). Pick the ISO that matches the host CPU: Intel or AMD (almost always x86-64). Do not download the ISO twice.

1. New machine → Linux → Ubuntu.
2. Set a simple username and password you will remember.
3. Give the disk enough room (100 GB if the host can take it). Cut this down if the PC is already slow.
4. Finish, then start the VM. The first boot is often slow. Wait.

Log in with the user you created. You now have a desktop that is independent from Windows.

Open a terminal

Activities → type terminal → Enter. Shortcuts if you want them:

```
Ctrl + Alt + T
Activities → Terminal
right-click desktop → Open in Terminal
```

Confirm the guest, then list the current directory (next chapter covers `ls` properly):

```
uname -a
cat /etc/os-release
ls
ls -l
ls -la
```

`ls` is names only. `ls -l` is the long listing. `ls -la` adds hidden files.

The Windows terminal (right-click Start → Terminal) talks to Windows. The Ubuntu terminal inside the VM talks to Linux. Use the one in the guest for this handbook.

Installing Linux Through WSL on Windows

What is WSL?

WSL (Windows Subsystem for Linux) runs Ubuntu inside Windows without VirtualBox. Use it when you only need a Linux terminal. Use VirtualBox when you want a full Ubuntu desktop.

Open PowerShell as Administrator.

```
wsl --install
wsl --install -d Ubuntu
wsl --list --online
```

The first command installs WSL with the default distro. The second pins Ubuntu. The third lists names you can pass to `-d`. Reboot if Windows asks.

Open **Ubuntu** from the Start menu. Create a UNIX username and password (this is not your Windows password).

```
uname -a
cat /etc/os-release
ls -la
```

Later, from PowerShell:

```
wsl --list --verbose
wsl -d Ubuntu
wsl --shutdown
```

WSL files live in Linux. Windows files live under `/mnt/c`. Do not treat WSL as a VPS: it is not reachable from the public internet unless you set that up yourself.



Installing Linux Through VirtualBox on Mac

How to use VirtualBox on Mac

Same idea as VirtualBox on Windows: the Mac is the host, Ubuntu is an isolated guest. Use this only if the Mac has spare RAM and CPU. Otherwise use a VPS.

Intel Macs need an AMD64 Ubuntu ISO. Apple Silicon Macs need ARM64.

Install

1. Search for **download VirtualBox for macOS** and install from the official site.
2. Download **Ubuntu LTS** for the matching architecture.
3. New → Linux → Ubuntu. Set a username and password. Give it as much RAM as you can spare and a large disk (100 GB if the Mac can take it).
4. Start the VM. The first boot is often slow. Log in, then open Terminal.

```
uname -m
uname -a
cat /etc/os-release
ls -la
```

`uname -m` should be `x86_64` (Intel) or `aarch64` (Apple Silicon). If the VM will not boot, the ISO architecture is usually wrong.

Cmd + Space → Terminal	(macOS <code>host</code>)
Ctrl + Alt + T	(Ubuntu guest, <code>if</code> the shortcut works)
Activities → Terminal	(Ubuntu guest)

The Mac Terminal talks to macOS. The Terminal app inside the VM talks to Ubuntu. Use the guest for Linux commands.



Basic Linux Commands

cat command

`cat` stands for **concatenate**. Most beginners first use it to print the contents of a file directly in the terminal.

```
cat file.txt
```

Example:

```
cat hello.txt
```

Output:

```
Hello Harry  
Welcome to Linux!  
This is my first file.
```

Useful `cat` examples

Read multiple files:

```
cat file1.txt file2.txt
```

It prints both files one after another.

Show line numbers:

```
cat -n file.txt
```

Output:

```
1 Hello Harry  
2 Welcome to Linux!  
3 This is my first file.
```

Create a file using `cat` :

```
cat > hello.txt
```

Now type:

```
Hello Linux  
This is my first file.
```

Press **Ctrl + D** to save and exit.

Append to a file:

```
cat >> hello.txt
```

Anything you type will be added to the end of the file. Again, **Ctrl + D** finishes it.

Important point

`cat` dumps the **entire file** into your terminal.

That's great for small files, but imagine:

```
cat /var/log/syslog
```

If the file has 50,000 lines, your terminal will get flooded.

That's where `less` becomes useful.

less command

`less` lets you **open a file and scroll through it interactively**.

```
less file.txt
```

For example:

```
less /var/log/syslog
```

Instead of printing everything at once, you get a viewer.

Navigation inside `less`

Key	Action
↑ / ↓	Move one line
Space	Move one page down
b	Move one page up
Enter	Move one line down
g	Go to beginning
G	Go to end
/word	Search for a word
n	Next search result
q	Quit

This makes `less` extremely useful for **logs and large files**.

Searching

Suppose you open:

```
less server.log
```

Press:

```
/error
```

and hit **Enter**.

`less` searches for `error`.

Then press:

```
n
```

to jump to the next occurrence.



cat vs less

`cat` → Quickly print a `file`
`less` → Open a `file` and explore it

```
cat config.txt
```

Good when `config.txt` is small.

```
less config.txt
```

Better when it's large and you want to scroll/search.

. and ..

In Linux, `.` and `..` are special directory shortcuts.

. = Current directory

A single dot means "the directory I'm currently in."

If you're here:

```
/home/harry/projects
```

then:

```
.
```

means:

```
/home/harry/projects
```

Example:

```
ls .
```

means:

List the files in the current directory.

You can also run a program in the current directory:

```
./script.sh
```

Here `./` means:

Look for `script.sh` in the current directory.

.. = Parent directory

Two dots mean "the directory one level above my current directory."

If you're here:

```
/home/harry/projects
```

then:

```
..
```

means:

```
/home/harry
```

```
cd ..
```

moves you **one directory up**.

For example:

```
/home
├─ harry
│   └─ projects ← you're here
```

Running:

```
cd ..
```

takes you to:

```
/home/harry
```

Running it again:

```
cd ..
```

takes you to:

```
/home
```

You can combine them:

```
cd ../../
```

means:

Go up two directories.

~ = Home directory

A tilde means **your home directory**.

If your user is `harry`, then `~` is:

```
/home/harry
```

```
cd ~
```

```
ls ~
```

`cd ~` takes you home. `ls ~` lists that directory.

Easy way to remember

```
. → current directory
.. → parent directory
~ → home directory
```

Creating Users

Linux is a multi-user operating system, which means multiple users can have their own accounts, files, permissions, and access levels.

Check the current user

```
whoami
```

This tells you **which user you are currently logged in as**.

For example:

```
harry
```

You can also use:

```
id
```

This shows more information about the current user, including their user ID (UID) and groups.

Create a new user

On Ubuntu, the easiest way is:

```
sudo adduser john
```

Linux will ask you to:

- Set a password
- Enter the user's full name
- Enter some optional information

A new user account called `john` is then created.

Linux also creates a home directory for the user:

```
/home/john
```

So each user gets their own place to store personal files and configuration.

Switch to the new user

You can switch users with:

```
su - john
```

The `-` is important because it starts a **login shell**, loading John's environment and taking you to his home directory.

Now:

```
whoami
```

will show:

```
john
```

And:

```
pwd
```

will show:


```
/home/john
```

To return to your previous user:

```
exit
```

See information about a user

```
id john
```

You might see something like:

```
uid=1001(john) gid=1001(john) groups=1001(john)
```

Here:

- `uid` = User ID
- `gid` = Group ID
- `groups` = Groups the user belongs to

Give a user sudo access

On Ubuntu, the `sudo` group gives users administrative privileges.

```
sudo usermod -aG sudo john
```

Now `john` can run administrative commands using:

```
sudo command
```

For example:

```
sudo apt update
```

Important: `sudo` does not mean the user has permanently become root. It allows the user to execute specific commands with elevated privileges.

After adding a user to `sudo`, the user may need to **log out and log back in** for the new group membership to appear in their session.

Change a user's password

```
sudo passwd john
```

This lets you set or change John's password.

See which groups a user belongs to

```
groups john
```

For example:

```
john : john sudo
```

This tells us that John belongs to both the `john` group and the `sudo` group.

Delete a user

```
sudo userdel john
```

This removes the user account.

If you also want to remove their home directory and files:

```
sudo userdel -r john
```

Be careful with `-r` because it permanently removes the user's home directory.

What `usermod -aG` means

```
sudo usermod -aG sudo john
```

breaks down as:

sudo	usermod	-aG	sudo	john	
					└ user to modify
					└ group to add
					options
					modify user
					run as administrator

usermod

`usermod` means **modify user**.

It changes properties of an existing Linux user.

-G

`-G` means **set the user's supplementary groups**.

For example:

```
sudo usermod -G sudo john
```

means:

Put John in the `sudo` group.

But there's an important problem.

If John is already a member of other supplementary groups, using `-G` without `-a` can replace his existing supplementary group memberships.

That's why we usually use `-aG`.

-a

`-a` means **append**.

So:

```
-aG
```

essentially means:

Add this group to the user's existing supplementary groups.

Therefore:

```
sudo usermod -aG sudo john
```

means:

Add John to the `sudo` group without removing him from his existing groups.

Why the `a` matters

Suppose John currently belongs to:

```
john
developers
docker
```

If you run:

```
sudo usermod -G sudo john
```

you could end up with:

```
sudo
```

and lose the previous supplementary group memberships.

But:

```
sudo usermod -aG sudo john
```

results in:

```
john
developers
docker
sudo
```

`-G` specifies groups, while `-aG` adds a group without removing existing supplementary groups.



Package Management

On Ubuntu and Debian, the main tool for installing, updating, and removing software is `apt`.

```
sudo apt install nginx
```

installs Nginx.

`apt update`

```
sudo apt update
```

This **does not update your installed software**.

Instead, it contacts the configured software repositories and downloads the latest information about available packages and their versions.

Think of it like:

"Check the app store and find out what's available."

You'll see output similar to:

```
Hit:1 http://archive.ubuntu.com/ubuntu ...
Get:2 http://security.ubuntu.com/ubuntu ...
Reading package lists... Done
```

After this, your system knows which packages have newer versions available.

`apt upgrade`

```
sudo apt upgrade
```

This actually **installs the available updates** for packages already installed on your system.

```
apt update
↓
Check what updates are available

apt upgrade
↓
Actually install those updates
```

Usual workflow:

```
sudo apt update
sudo apt upgrade
```

You can also write:

```
sudo apt update && sudo apt upgrade
```

The `&&` means:

Run the second command only if the first command succeeds.

A common beginner mistake is thinking `sudo apt update` means "update Linux." It doesn't.

Command	What it does
<code>apt update</code>	Updates the package information/index
<code>apt upgrade</code>	Updates installed packages
<code>apt install nginx</code>	Installs a package
<code>apt remove nginx</code>	Removes a package

Phone app-store version of the same idea:

`apt update` → check which apps have updates.

`apt upgrade` → download and install those updates.

What can go wrong

`apt upgrade` can break things, although on a normal Ubuntu server it's generally safe and routine. It is designed to upgrade packages without removing installed packages or making major dependency changes whenever possible. An updated package can still introduce a bug, incompatibility, or configuration change.

- A new version of **Nginx/Apache** could have a configuration incompatibility.
- A **kernel update** could cause hardware or driver issues.
- A **database update** could introduce compatibility problems.
- A library update could affect an application that depends on it.
- A service might fail to restart correctly after an update.

Don't blindly assume every upgrade is risk-free. On a production server, have **backups or snapshots and a rollback plan** before major upgrades.

upgrade vs full-upgrade

```
sudo apt upgrade
```

Conservative. It generally won't remove packages to satisfy dependency changes.

```
sudo apt full-upgrade
```

More aggressive. It can **install or remove packages** when necessary to complete the upgrade.

`apt update` refreshes the information. `apt upgrade` installs available updates. `apt full-upgrade` can make larger dependency changes, including removing packages.

Search for a package

```
apt search nginx
```

Useful when you don't know the exact package name.

See package information

```
apt show nginx
```

Shows version, dependencies, description, and package size.

Install a package

```
sudo apt install nginx
```

You can install multiple packages at once:

```
sudo apt install git curl wget
```

Remove a package

```
sudo apt remove nginx
```

This removes the package but generally leaves its configuration files.

```
sudo apt purge nginx
```

`purge` removes the package **and its configuration files**.

remove → remove software

purge → remove software + configuration

See installed packages

```
apt list --installed
```

You can combine this with `grep`:

```
apt list --installed | grep nginx
```

Which version is installed

```
apt policy nginx
```

Useful for troubleshooting package versions.

Clean unused packages

```
sudo apt autoremove
```

Removes packages that were installed as dependencies but are no longer needed.

```
sudo apt clean
```

clears downloaded package files from the local cache.



Repositories

When you run `sudo apt install nginx`, APT doesn't magically find Nginx on the internet. It gets packages from configured **repositories**.

```
Repository
↓
APT package lists
↓
apt install
↓
Package downloaded
↓
Package installed
```

Ubuntu's repository configuration lives here:

```
/etc/apt/sources.list
/etc/apt/sources.list.d/
```

apt vs apt-get

You'll encounter both:

```
apt
apt-get
```

`apt` is the modern, user-friendly interface for package management. `apt-get` is an older command that is still widely used, especially in scripts.

This handbook uses `apt`.



Groups and Permissions

The easiest way to understand Linux permissions is to start with the problem Linux is trying to solve. Imagine a server with three people:

- Harry - administrator
- Alice - developer
- Bob - another developer

There might be files like:

```
/home/harry/private.txt
/home/project/app.py
/etc/nginx/nginx.conf
```

Linux needs to answer:

Who is allowed to read, modify, or execute each file?

That's what **users, groups, and permissions** are for.

Users

Every person or process interacting with Linux operates as a **user**.

```
whoami
```

Example:

```
harry
```

Linux internally identifies users using a **UID (User ID)**.

```
id harry
```

might show:

```
uid=1000(harry) gid=1000(harry) groups=1000(harry),27(sudo)
```

Don't worry about every number yet.

A username is basically a human-friendly name for a UID.

Groups

Groups exist so that you don't have to give permissions to users one by one.

Imagine you have 20 developers. Instead of listing Alice, Bob, Charlie, and everyone else, you create a group called `developers` and put everyone into it. Then give the **group** access to the project.

```
developers
├─ Alice
├─ Bob
└─ Charlie
```

Now if you give the `developers` group access to a file, all three developers get that access.

A user can belong to multiple groups:

```
Harry
├─ harry
├─ sudo
├─ developers
└─ docker
```

```
groups harry
id harry
```

Create a group and add a user:

```
sudo groupadd developers
sudo usermod -aG developers harry
```

```
-a → append
-G → supplementary groups
```

Add Harry to the developers group without removing his existing supplementary groups.

After changing group membership, the user generally needs to **log out and back in** before the new membership appears in their existing session.

Permissions

Every file and directory in Linux has permissions.

```
ls -l
```

You might see:

```
-rwxr-xr-- 1 harry developers 1234 Aug 24 app.sh
```

```
-rwxr-xr--
|  |  |
|  |  └─ Others
|  └─── Group
└────── Owner
```

There are **three categories of people** Linux cares about:

Owner

The specific user who owns the file. Here: `harry`.

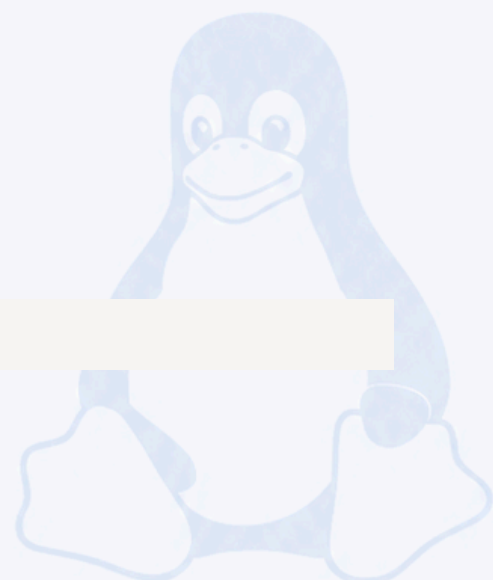
Group

The group associated with the file. Here: `developers`.

Others

Everyone else.

```
Owner | Group | Others
```



Read, write, execute

Each category can have three basic permissions:

```
r = read
w = write
x = execute
```

```
rwX → read, write, execute
r-X → read, NO write, execute
--- → nothing
```

Understanding `-rwxr-xr--`

Ignore the first `-` for a moment.

```
rwX | r-X | r--
Owner | Group | Others
```

Owner

```
rwX
```

Can read, write, and execute.

Group

```
r-X
```

Can read and execute, **not** write.

Others

```
r--
```

Can only read.

The owner can do everything, the group can read and execute, and everyone else can only read.

The first character

```
-rwxr-xr--
^
```

that first character tells you the **file type**.

```
- regular file
d directory
l symbolic link
```

So `-rwxr-xr--` is a regular file. `drwxr-xr-x` is a directory.

Permissions on directories

For a **directory**, the letters mean something slightly different.

r - read

See the directory's contents / list filenames.

w - write

Create, delete, or rename entries in the directory, subject to the other permission rules.

x - execute

Enter / traverse the directory.

```
Directory:
r → list contents
w → create/delete/rename entries
x → enter/traverse
```

```
mkdir test
ls -ld test
```

Then experiment with permissions.

Changing permissions with `chmod`

Suppose `ls -l script.sh` shows:

```
-rw-r--r-- script.sh
```

It's not executable.

```
chmod +x script.sh
```

Now it becomes executable (for example `-rwxr-xr-x`), and you can run:

```
./script.sh
```

Letters

```
chmod u+x script.sh
```

`u` means **user/owner**. `u+x` means: add execute permission for the owner.

```
u = user/owner
g = group
o = others
a = all
```

```
chmod u+x script.sh
chmod g+w file.txt
chmod o-r file.txt
chmod g=rx file.txt
```

`g+w` gives the group write. `o-r` takes read away from others. `g=rx` sets the group's permissions to **read + execute** exactly, and removes any other group permissions.

Numbers

```
r = 4
w = 2
x = 1
```

Add them together.

```
rwX = 4 + 2 + 1 = 7
rw- = 4 + 2 = 6
r-X = 4 + 1 = 5
r-- = 4 = 4
```

```

-rwx = 2 + 1 = 3
-rw- = 2 = 2
--x = 1 = 1
--- = 0

```

The three digits are owner, group, others:

```

chmod 755 file
  |||
  ||  Others
  |   Group
  |   Owner

```

```
chmod 755 script.sh
```

```

7 = rwx → Owner: read + write + execute
5 = r-x → Group:  read + execute
5 = r-x → Others: read + execute

```

```
chmod 644 file.txt
```

```

6 | 4 | 4 → rw- | r-- | r--
Owner: read + write
Group:  read
Others: read

```

Common for ordinary files.

```
chmod 700 private.txt
```

```
700 = rwx --- ---
```

Only the owner has access.

Letters are useful for a small change. Numbers set the complete permission set at once.

Why 777 is dangerous

```
777 → rwx | rwx | rwx
```

Everyone can read, modify, and execute the file. Don't casually run `chmod 777 file` just because you get "permission denied."

Don't solve permission problems by blindly using `chmod 777`. Understand who actually needs access and give the minimum required permissions.

Changing ownership

```
-rw-r--r-- 1 harry developers app.py
```

Owner is `harry`. Group is `developers`.

```
sudo chown alice app.py
```

Now Alice owns it.


```
sudo chown alice:developers app.py
```

```
chown OWNER:GROUP FILE
```

Owner → alice. Group → developers.

```
sudo chgrp developers app.py
```

changes only the group.

chown → change ownership

chgrp → change group

chmod → what they can **do**

Recursive **-R**

```
project/
├─ app.py
├─ config.txt
├─ logs/
│  └─ app.log
```

`sudo chown lovish:developers project` changes **only project** itself.

```
sudo chown -R lovish:developers project
```

-R means **recursive**: the directory and everything underneath it. Be careful. A mistake in the path can change ownership of a huge number of files and potentially break the system.

Reading one `ls -l` line

```
drwxr-x--- 3 harry harry 4096 Aug 23 10:21 harry
```

Diagram illustrating the components of the `ls -l` line:

- Permissions: `drwxr-x---`
- Hard links: `3`
- Owner: `harry`
- Group: `harry`
- Size: `4096`
- Date/time: `Aug 23 10:21`
- Name: `harry`

d means directory. Permissions:

```
rwX | r-x | ---
Owner Group Others
```

Harry has full access. The `harry` group can read and enter, but cannot write. Others have no access.

The `3` is the **hard-link count**. For directories, it is related to the directory itself and its subdirectories.

The two `harry` s are not redundant:

```
harry (first) = owner/user
harry (second) = group
```

When you create a user, Linux commonly creates a group with the same name.

4096 is the directory's size in bytes as reported by `ls` - the metadata, not the total size of everything inside. For that:

```
du -sh harry
```

Aug 23 10:21 is the last modification time. The final `harry` is the name.

Directory	Owner	Group	Owner	Group	Others
harry	harry	harry	rwX	r-X	---
lovish	lovish	lovish	rwX	r-X	---
rohan	rohan	rohan	rwX	r-X	---

How Linux chooses which permissions apply

Suppose:

```
-rwxr----- 1 harry developers app.py
```

```
rwX | r-- | ---
Owner Group Others
```

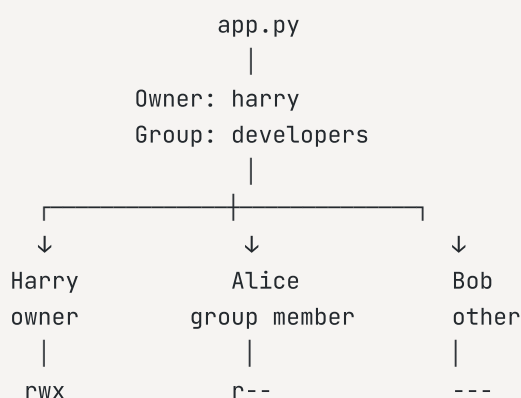
Harry (owner) → rwX . Members of developers → r-- . Everyone else → --- .

When Alice, a member of `developers`, tries to access `app.py`, Linux asks:

Is Alice the owner? **No**. Is Alice in the file's group (`developers`)? **Yes**.

It uses the **group** set: `r--`. Alice can read, not write or execute.

If Bob is neither the owner nor in `developers`, Linux uses **others**: `---`. Bob gets no access.



Linux doesn't combine permissions. If Alice is both the **owner** and in `developers`, Linux uses the **owner** permissions only.

```

Is user the owner?
↓
YES → use OWNER permissions
↓ NO
Is user in the file's group?
↓

```

YES → use GROUP permissions
↓ NO
Use OTHERS permissions

Worked example

```
sudo adduser alice
sudo adduser bob
sudo groupadd developers
sudo usermod -aG developers alice
sudo usermod -aG developers bob
sudo touch /opt/project.txt
sudo chown alice:developers /opt/project.txt
ls -l /opt/project.txt
sudo chmod 640 /opt/project.txt
```

```
640
|||
||└─ Others: no permissions
|└─ Group: read
└─ Owner: read + write
```

Owner → Alice. Group → developers. Others → everyone else.



Once you understand this, commands like `ps`, `top`, `kill`, and `systemctl` stop feeling like random commands.

What is a process?

A process is a running program.

When you run `python app.py`, Linux starts a process for that Python program. When you run Nginx, Linux starts one or more Nginx processes. Even `ls` is a process: it starts, does its work, and exits.

Program = code stored on disk

Process = that program currently running

Every process has a PID

Linux gives every running process a **PID - Process ID**.

```
ps
```

PID	TTY	TIME	CMD
1234	pts/0	00:00:00	bash
5678	pts/0	00:00:00	ps

5678 is the PID of the `ps` command itself. A PID is the process's unique ID.

```
ps
```

`ps` means **process status**. Plain `ps` shows processes associated with your current terminal. On a server you usually want more:

```
ps aux
```

Processes from **all users**, with detail:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	16900	11000	?	Ss	Aug24	0:03	/sbin/init
harry	1234	0.0	0.2	12000	5000	pts/0	Ss	15:30	0:00	bash
root	987	0.1	1.2	100000	50000	?	S	15:20	0:10	nginx

USER	→ who owns the process
PID	→ process ID
%CPU	→ CPU usage
%MEM	→ memory usage
STAT	→ process state
START	→ when it started
TIME	→ CPU time used
COMMAND	→ command/program

Finding a particular process

```
ps aux | grep nginx
```

`|` is a **pipe**. It takes the output of one command and sends it to another.

Get all processes → search that output for "nginx".

A cleaner way when you already know the name:

```
pgrep nginx
pgrep -a nginx
```

`pgrep` returns PIDs. `pgrep -a` also prints the command lines.

top and htop

`ps` gives you a **snapshot**. `top` gives you a **live view**.

```
top
```

The display continuously updates. Use it for: *What's consuming my CPU or RAM right now?*

```
htop
```

A more interactive viewer: scroll, sort by CPU or memory, search, kill. On Ubuntu you may need `sudo apt install htop`.

```
ps      → snapshot
top     → live monitoring
htop    → interactive live monitoring
```

If you see `%CPU` at 95.0 on a Python process, that process is consuming a lot of CPU. `%MEM` is roughly how much of the system's memory it is using. When the server is slow, run `top` and look for those columns.

kill

```
kill 1234
```

`kill` **doesn't necessarily mean "immediately destroy the process."** By default it sends **SIGTERM**: "Please shut down." A well-behaved application can catch this and clean itself up.

```
kill -9 1234
```

`-9` sends **SIGKILL**: "Stop immediately." The process cannot catch SIGKILL.

```
kill PID
↓
Ask process to terminate gracefully

kill -9 PID
↓
Forcefully terminate process
```

Try normal `kill` first. Don't use `kill -9` as the default.

```
pkill nginx
```

sends a termination signal to matching processes, without looking up a PID first. Matching the wrong name can terminate more than you intended.

What is a service?

A **service** is software that typically runs in the background and provides some function to the system or other programs.

```
nginx    → web server
ssh      → remote access
mysql    → database
cron     → scheduled tasks
```

You don't want to run `nginx` by hand every time the server boots. **systemd** manages that. You talk to it with `systemctl`.

```
systemd
  ↓
manages services
  ↓
systemctl
  ↓
you control those services
```

systemctl status

```
systemctl status nginx
```

- nginx.service - A high performance web server
 - Loaded: loaded
 - Active: active (running)
 - Main PID: 987

`active (running)` means it is up. `inactive (dead)` means it is not. `failed` means it failed to start or stopped unexpectedly.

Start, stop, restart, reload

```
sudo systemctl start nginx
sudo systemctl stop nginx
sudo systemctl restart nginx
sudo systemctl reload nginx
```

`start` starts it **now**. That does not mean it will start after reboot. `restart` is stop then start - common after editing `/etc/nginx/nginx.conf`. `reload` asks the service to reload configuration **without** fully stopping. Not every service supports reload.

```
restart → stop + start
reload  → reload configuration while staying running
```

For Nginx, reload is often better when you only changed configuration and want to avoid dropping connections.

Enable and disable

```
sudo systemctl enable nginx
sudo systemctl disable nginx
```

```
start    → start it NOW
enable   → start it AUTOMATICALLY on boot
stop     → stop it now
disable  → don't start automatically at boot
```

Disable does not necessarily stop a service that is already running.

```
sudo systemctl enable --now nginx
sudo systemctl disable --now nginx
```

`--now` means do it immediately **and** change the boot behavior.

Listing and checking

```
systemctl list-units --type=service
systemctl list-unit-files --type=service
systemctl is-active nginx
systemctl is-enabled nginx
```

```
list-units      → what's currently loaded/active
list-unit-files → what service definitions exist and their enablement state
```

`is-active` / `is-enabled` print a short word (`active` , `enabled`) - useful in scripts.

Service logs

```
journalctl -u nginx
journalctl -u nginx -n 50
journalctl -u nginx -f
```

`-u` means **unit**. `-n 50` is the last 50 entries. `-f` means **follow**, like `tail -f`.

When the website isn't working

1. `sudo systemctl status nginx`
2. If it's stopped: `sudo systemctl start nginx`
3. If it failed: `sudo journalctl -u nginx -n 50`
4. `ps aux | grep nginx`
5. `top`

That's a troubleshooting path, not a list of commands to memorize in isolation.

Process vs service

A **process** is a specific running instance of a program (`nginx` , PID 1234). A **service** is a managed background unit (`nginx.service`). One service can have **multiple processes**.

They are related. They are not the same thing.

```
nginx.service
|
├─ nginx process PID 1234
```



```
|— nginx process PID 1235  
└─ nginx process PID 1236
```

```
PROCESS    → running program, identified by PID  
SERVICE   → background application managed by systemd  
systemctl  → controls services  
journalctl → reads systemd/service logs
```



Environment Variables, PATH and .bashrc

This chapter is about how Linux knows where commands are, and how programs get configuration from the shell.

What is an environment variable?

An environment variable is a **named value stored in the shell's environment**.

```
echo $HOME
```

might output `/home/harry` . So `HOME = /home/harry` .

```
echo $USER
```

might give `harry` .

```
VARIABLE_NAME = VALUE
```

See your environment variables

```
printenv
env
```

You'll see things such as:

```
USER=harry
HOME=/home/harry
SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Both can inspect the environment. They have somewhat different purposes beyond the basics.

Access a variable

Use `$` before the name:

```
echo $HOME
echo $USER
echo $SHELL
```

The `$` means: give me the value stored in this variable.

Without `$` , `echo HOME` simply prints `HOME` .

Create your own variable

```
name="Harry"
echo $name
```

There must be **no spaces** around `=` .

Correct: `name="Harry"`

Wrong: `name = "Harry"` - the shell interprets that differently and you'll get an error.

Shell variable vs environment variable

`name="Harry"` creates a **shell variable**. It isn't automatically passed to programs you launch.

```
export name="Harry"
export APP_ENV="production"
```

Now programs launched from this shell can receive the variable. `APP_ENV` is part of the environment inherited by child processes.

Why they exist

They're useful for configuring applications. Instead of hardcoding `production` in the code, export `APP_ENV=production` and let the application read it.

Common names: `DATABASE_URL`, `API_KEY`, `PORT`, `APP_ENV`, `DEBUG`.

Environment variables are not automatically a secure secret store. Sensitive values can be exposed through process inspection, logs, debugging, or misconfiguration. Putting a password in an environment variable does not make it magically secure.

PATH

```
echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Directories are separated by `:`.

```
PATH
|
├─ /usr/local/sbin
├─ /usr/local/bin
├─ /usr/sbin
├─ /usr/bin
├─ /sbin
└─ /bin
```

When you type `ls` you didn't type `/usr/bin/ls`. The shell searches `PATH` until it finds an executable named `ls`.

Where a command comes from

```
which ls
command -v ls
type ls
```

`which` might print `/usr/bin/ls`. `command -v` tells you how the shell resolves the command. `type` can reveal an alias, function, builtin, or external command.

```
type cd
```

might say `cd is a shell builtin`. Not every command is a standalone executable.

Why `./` is sometimes necessary

Create `hello.sh`, `chmod +x hello.sh`, then `hello.sh` might say **command not found**. `./hello.sh` works.

The current directory `.` usually **isn't in your `PATH`**. `./hello.sh` means: run `hello.sh` from the current directory.

That's also a security reason: automatically searching `.` for commands can run an unintended local executable.

Adding a directory to `PATH`

If your scripts live in `/home/harry/scripts` and you want to type `backup` instead of `/home/harry/scripts/backup`:

```
export PATH="$PATH:/home/harry/scripts"
```

The existing `PATH` is expanded first: take my existing `PATH` and append this directory.

If you instead run `export PATH="/home/harry/scripts"`, you **replace** `PATH`. Then `ls`, `cat`, and `sudo` may no longer be found by their normal names.

When modifying `PATH`, usually preserve the existing value rather than replacing it.

```
echo $PATH
command -v backup
```

Temporary vs permanent

`export APP_ENV=production` lasts for this shell (and programs it launches). Close the session and it normally disappears.

For Bash, put lasting settings in `~/.bashrc`:

```
export APP_ENV=production
export PATH="$PATH:$HOME/scripts"
```

Then start a new shell, or:

```
source ~/.bashrc
```

`source` tells the current shell to **read and execute** that file. Shorthand: `source ~/.bashrc`. That `.` means `source`. It is **not** the same `.` as "current directory" in a path.

Worked example

```
echo $PATH
mkdir -p ~/scripts
```

Create `~/scripts/hello`:

```
#!/bin/bash
echo "Hello from my script!"

chmod +x ~/scripts/hello
hello
```

`hello` won't normally work yet because `~/scripts` isn't in `PATH`. `~/scripts/hello` will.

```
export PATH="$PATH:$HOME/scripts"
hello
command -v hello
```

You should get something like `/home/harry/scripts/hello` .

Environment Variables

- |
- ├─ HOME
- ├─ USER
- ├─ SHELL
- ├─ PATH
- └─ APP_ENV

↓

Programs inherit them

PATH

- |
- ├─ /usr/local/bin
- ├─ /usr/bin
- ├─ /bin
- └─ ~/scripts

↓

Shell searches these
directories for commands



Archiving and compression are not the same thing.

`tar` primarily **archives** files (combines many files/directories into one archive). `gzip` **compresses** data. `zip` generally does both in one format.

Why archives?

```
project/
├─ app.py
├─ config.py
├─ index.html
├─ styles.css
└─ images/
   └─ logo.png
      └─ banner.jpg
```

Sending every file individually is inconvenient. You want one file containing everything. That's **archiving**: `project/` → `project.tar`.

Then compress it: `project.tar` → `gzip` → `project.tar.gz`. Now it's both an archive and compressed.

`tar`

`tar` originally means **Tape Archive**.

```
tar [options] archive-name files
```

```
tar -cf project.tar project/
```

`-c` = create
`-f` = file

Create an archive called `project.tar` containing the `project` directory.

```
ls -lh project.tar
```

List without extracting:

```
tar -tf project.tar
```

`-t` = list contents
`-f` = archive file

Extract:

```
tar -xf project.tar
```

`-x` = extract
`-f` = file

You'll often see `-v` as well (**verbose** - print files as it works):

```
tar -xvf project.tar
tar -cvf project.tar project/
```

```
c → create
x → extract
t → list
v → verbose
f → archive file
```

Gzip

```
gzip project.tar
```

normally produces `project.tar.gz` and replaces the original `project.tar`.

```
gunzip project.tar.gz
```

returns `project.tar`.

```
gzip → compress
gunzip → decompress
```

tar.gz

Instead of `tar` then `gzip`, do both:

```
tar -czf project.tar.gz project/
tar -xzf project.tar.gz
```

`z` means use gzip compression.

```
c = create    z = gzip    f = file
x = extract   z = gzip    f = file
```

You may also see `.tar.bz2` and `.tar.xz` - same idea, different compression. `.tar.gz` is not the only compressed tar format.

Why the name: `.tar` is the archive, `.gz` is gzip. So `software.tar.gz` is a tar archive that has been gzip-compressed.

zip

Unlike plain `tar`, a ZIP archive generally **combines and compresses files**.

```
zip -r project.zip project/
unzip project.zip
unzip project.zip -d extracted/
unzip -l project.zip
```

`-r` is **recursive** (needed for a directory). `-d` is the destination. `-l` lists without extracting.

tar vs zip

Command/format	Archive?	Compression?
<code>tar</code>	yes	no, by itself
<code>gzip</code>	no	yes
<code>tar.gz</code>	yes	yes
<code>zip</code>	yes	yes

`tar` → combines files into one archive
`gzip` → compresses data
`tar.gz` → `tar` + `gzip`
`zip` → archive + compression in one format

Worked example

```

mkdir project
touch project/app.py project/config.py project/index.html

tar -cf project.tar project/
tar -tf project.tar
gzip project.tar
gunzip project.tar.gz
tar -xf project.tar

tar -czf project.tar.gz project/
tar -xzf project.tar.gz

zip -r project.zip project/
unzip project.zip
  
```

Don't memorize letter piles like `tar -czvf` as random noise. Build them from the flags.

```

tar -cf archive.tar folder/
tar -tf archive.tar
tar -xf archive.tar
tar -czf archive.tar.gz folder/
tar -xzf archive.tar.gz
gzip file
gunzip file.gz
zip -r archive.zip folder/
unzip -l archive.zip
unzip archive.zip
  
```

Cronjobs

Cron jobs **automatically run commands or scripts at scheduled times**.

You might want Linux to create a backup every night, run a script every hour, clean temporary files every Sunday, check something every 5 minutes, or generate a report every morning. Instead of running the command yourself, **cron runs it for you**.

What is cron?

`cron` is a service that runs scheduled tasks in the background. A scheduled task is called a **cron job**.

```
Cron
|
├─ Every 5 minutes → run backup
├─ Every day → clean logs
└─ Every Sunday → run maintenance
```

On Ubuntu:

```
systemctl status cron
```

You want `Active: active (running)`. If it isn't:

```
sudo systemctl start cron
sudo systemctl enable cron
```

`enable` makes it start automatically when the server boots.

crontab

A **crontab** is a file containing scheduled tasks for a user.

```
crontab -l
crontab -e
```

`-l` means **list**. `-e` means **edit**. If there are no jobs you might see `no crontab for harry`.

Example job:

```
* /5 * * * * echo "Hello" >> /home/harry/cron.log
```

Every 5 minutes, append `Hello` to `cron.log`.

The five fields

```
* * * * * command
| | | | |
| | | | └─ Day of week   (0-7)
| | | └─── Month        (1-12)
| | └───── Day of month (1-31)
| └────── Hour          (0-23)
└──────── Minute         (0-59)
```

`*` means **every possible value**. So `* * * * *` is every minute of every hour, every day.

Common schedules

```
* /5 * * * * /home/harry/backup.sh
```

`* /5` means every 5 minutes: 12:00, 12:05, 12:10, ...

```
0 * * * * /home/harry/backup.sh
```

Minute `0` of every hour: 01:00, 02:00, 03:00, ... That is **not** "every 0 hours." It is: at minute 0 of every hour.

```
30 2 * * * /home/harry/backup.sh
```

Every day at 02:30.

```
0 3 * * 0 /home/harry/backup.sh
```

Every Sunday at 3:00 AM.

`0` or `7` = Sunday

`1` = Monday

`2` = Tuesday

`3` = Wednesday

`4` = Thursday

`5` = Friday

`6` = Saturday

```
0 0 1 * * /home/harry/report.sh
```

Midnight on the first day of every month.

Commas for several values, a hyphen for a range:

```
0 9 * * 1,3,5 /home/harry/report.sh
```

```
0 9 * * 1-5 /home/harry/report.sh
```

9:00 AM on Monday, Wednesday, and Friday. Then 9:00 AM Monday through Friday.

`*` every value

`*/5` every `5`

`1,5,10` specific values

`1-5` range

Scripts and absolute paths

Usually you don't put a huge command in the crontab. Create a script:

```
nano /home/harry/backup.sh
```

```
#!/bin/bash
```

```
tar -czf /home/harry/backup.tar.gz /home/harry/project
```

```
chmod +x /home/harry/backup.sh
```

```
0 2 * * * /home/harry/backup.sh
```

Every day at 2 AM.

Cron's environment is not your interactive terminal. Don't rely on `backup.sh` or even `tar` being found the way they are in your shell. Prefer `/home/harry/backup.sh`, and inside scripts use explicit paths when it matters (`/usr/bin/tar`). Find a command with `which tar` or `command -v tar` .

Where output goes

You won't see cron output in your terminal. Redirect it:

```
* * * * * /home/harry/test.sh >> /home/harry/cron.log 2>&1
```

`>>` appends normal output. `2>&1` sends errors to the same place. Then `cat` or `tail` the log.

Cron jobs belong to users

Harry's `crontab -e` belongs to Harry. Alice's belongs to Alice. `crontab -l` shows **your** jobs.

Root has its own crontab: `sudo crontab -e` . A root job runs with **root privileges**, so a mistake is much more serious.

Removing jobs

Edit with `crontab -e` and delete the line. `crontab -r` removes the user's **entire** crontab. Don't use `-r` casually on a machine you care about.

If it didn't run

Don't assume a job ran just because you added it.

```
crontab -l
tail /home/harry/cron.log
systemctl status cron
```

Cron uses the **server's timezone**, so timezone configuration matters when you schedule jobs.

A first check that cron works:

```
* /5 * * * * echo "Cron works!" >> /home/harry/cron.log
```

Wait a few minutes, then `cat /home/harry/cron.log` . If you see the line repeating, cron is doing its job.



Understanding Linux Filesystem

Once you understand the filesystem hierarchy, Linux stops feeling like a random collection of folders. Unlike Windows, Linux doesn't have drives like `C:\` , `D:\` . Everything starts from **one root directory**:

`/` .

```

/
├─ bin
├─ boot
├─ dev
├─ etc
├─ home
├─ lib
├─ media
├─ mnt
├─ opt
├─ proc
├─ root
├─ run
├─ sbin
├─ srv
├─ sys
├─ tmp
├─ usr
└─ var

```

You don't need to memorize every directory. These are the ones that matter first.

`/` - Root

`/` is the **root of the filesystem**. Everything is somewhere underneath it.

```

cd /
ls

```

`/` vs `/root`

These are **not the same thing**.

- `/` → root of the entire filesystem
- `/root` → home directory of the root user

```

/
├─ home
│   └─ harry
└─ root

```

`/home` - Users' files

This is where normal users generally keep their personal files.

```

/home
├─ harry
├─ alice
└─ bob

```

If your username is `harry` , `/home/harry` is usually your **home directory**. These also get you there:

```

cd ~
cd

```

`/home/harry` and `~` refer to the same place when you're logged in as `harry`.

/etc - Configuration

`/etc` contains **system-wide configuration files**.

```
/etc/ssh/
/etc/apt/
/etc/systemd/
/etc/passwd
/etc/hosts
```

```
cat /etc/hosts
cat /etc/passwd
```

`/etc/hosts` has hostname/IP mappings. `/etc/passwd` has information about users.

`/etc` = **configuration**

/var - Variable data

`/var` contains data that **changes frequently while the system is running**: `/var/log`, `/var/cache`, `/var/lib`.

```
ls /var/log
```

On many distributions you'll find `/var/log/syslog` and `/var/log/auth.log`. If something goes wrong with a server, `/var/log` is often one of the first places you look.

`/var` = data that **varies/changes**

/usr - Userland programs and resources

Historically `/usr` meant "**Unix System Resources**", not "your user's files." It holds a huge amount of software, commands, libraries, and shared resources: `/usr/bin`, `/usr/sbin`, `/usr/lib`, `/usr/share`.

```
ls /usr/bin
which python3
```

You might see `/usr/bin/python3`, `/usr/bin/grep`, `/usr/bin/curl`.

`/usr` = most installed **user-space programs and resources**

/bin - Essential commands

`/bin` traditionally contains essential executables such as `ls`, `cp`, `mv`, `cat`, `rm`.

On **modern Linux**, `/bin` is often a symbolic link to `/usr/bin`:

```
ls -ld /bin
```

You may see `/bin` → `usr/bin`. So `/bin` is not the only place commands live.

`/bin` traditionally contains essential commands; on modern systems it is commonly merged with `/usr/bin`.

/tmp - Temporary files

```
cd /tmp
touch test.txt
```

Programs use `/tmp` for temporary storage. **Don't store important files there.** They may be deleted on reboot or by automatic cleanup.

`/tmp` = "I need this temporarily."

The tree

```
/
├── home/      → Users' personal files
├── etc/       → System configuration
├── var/       → Changing data, logs, caches
├── usr/       → Programs and shared resources
├── bin/       → Essential commands
├── tmp/       → Temporary files
└── root/     → Root user's home directory
```

Directory	Purpose
<code>/boot</code>	Files needed to boot Linux
<code>/dev</code>	Devices represented as files
<code>/proc</code>	Information about processes/kernel
<code>/sys</code>	Kernel/device information
<code>/run</code>	Runtime data since boot
<code>/mnt</code>	Temporary/manual mount points
<code>/media</code>	Usually removable media
<code>/opt</code>	Optional/third-party software
<code>/sbin</code>	Traditionally system administration commands

Walk it once: `cd /` then `ls`, then `/home`, `/etc`, `/var`, `/usr`, `/tmp`. `pwd` will keep reminding you: Linux is one tree starting at `/`.

Seven to remember: `/` everything · `/home` users · `/etc` configuration · `/var` changing data/logs · `/usr` programs · `/bin` essential commands · `/tmp` temporary files.

Understanding Nginx

Nginx is a **web server**. It listens on a port (usually 80 or 443), receives HTTP requests, and sends back files or responses. On a Linux VPS, this is how a browser on your laptop reaches a site running on the machine you rented.

You already met Nginx as a **service** (`systemctl status nginx`). This chapter is the other half: what it serves, where the files live, and how to put a page on the network.

Browser → Port 80 → Nginx → files in `/var/www/html`

Install and run

On Ubuntu:

```
sudo apt update
sudo apt install nginx
sudo systemctl enable --now nginx
```

`enable --now` starts it immediately **and** on boot.

```
systemctl status nginx
systemctl is-active nginx
ps aux | grep nginx
```

You want `active (running)`. Several Nginx **processes** under one **service** is normal.

See the default page

From the server:

```
curl http://127.0.0.1
curl http://localhost
curl -I http://127.0.0.1
```

The first two print the HTML. `-I` prints headers only (look for `HTTP/1.1 200` and `Server: nginx`).

From your laptop, open `http://YOUR_SERVER_IP` in a browser. If nothing loads, the service may be down, or a firewall may be blocking port 80.

```
sudo ss -tlnp | grep ':80'
sudo nginx -t
sudo journalctl -u nginx -n 50
```

`ss` shows whether something is listening on 80. `nginx -t` tests the configuration **without** applying it. Logs explain a failed start.



Where Nginx keeps things

```

/etc/nginx/nginx.conf      → main config
/etc/nginx/sites-available/ → site definitions
/etc/nginx/sites-enabled/  → sites that are actually on
/var/www/html/             → default files Nginx serves

```

`sites-enabled` entries are usually **symbolic links** to files in `sites-available`. Enabling a site is linking it. Disabling it is removing the link - the file in `sites-available` stays.

```

ls /etc/nginx/sites-available
ls /etc/nginx/sites-enabled
ls -l /etc/nginx/sites-enabled

```

The default site often points at `/var/www/html`. That's why replacing `index.html` there changes what visitors see.

Serve your own page

```

ls -l /var/www/html
sudo nano /var/www/html/index.html

```

Put something simple:

```

<!DOCTYPE html>
<html>
  <head><title>Hello</title></head>
  <body><h1>Hello from Nginx</h1></body>
</html>

```

Save, then:

```
curl http://127.0.0.1
```

You should see your heading. No restart is required for a static HTML change - Nginx reads the file on each request.

If you edit **configuration** (not HTML), test then reload:

```

sudo nginx -t
sudo systemctl reload nginx
sudo systemctl restart nginx

```

`nginx -t` first. `reload` keeps the process up and rereads config. `restart` is stop then start - use it when reload isn't enough.

`reload` for config. Edit HTML in `/var/www/html` and just refresh the browser. Don't `chmod 777` the web root to "make it work" - fix ownership or group access instead (`www-data` is Nginx's usual user on Ubuntu).

A minimal config picture

A server block (virtual host) answers: which port, which files.

```
listen 80
↓
server_name _
↓
root /var/www/html
↓
index index.html
```

You don't need to write one from scratch yet. The default Ubuntu site already does this. When you add a second site later, you copy a file in `sites-available`, link it into `sites-enabled`, run `nginx -t`, then `reload`.

If the page doesn't load

1. `systemctl status nginx` - is it running?
2. `sudo nginx -t` - is the config valid?
3. `curl http://127.0.0.1` - does it work **on the server**?
4. Browser to the public IP - if curl works but the browser doesn't, look at firewall / security group / Hostinger firewall for port 80.

```
sudo ufw status
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

Only if you are using UFW. On some VPS panels the firewall is in the dashboard, not `ufw`.



SSH gives you a shell. FileZilla gives you **folders**: drag a file from your laptop onto the server, or pull one back. Under the hood it should use **SFTP** (SSH File Transfer Protocol) on port **22** - the same door as `ssh`, not old unencrypted FTP on port 21.

Laptop → SFTP (port 22) → files on the VPS

Connect with FileZilla

1. Install FileZilla from filezilla-project.org (the **client**, not the server).
2. Open **File** → **Site Manager** (or the Quickconnect bar).
3. Protocol: **SFTP**. Host: your server IP. Port: **22**. User: `root` or `ubuntu` (whatever you SSH with). Password or key file: the same credentials as SSH.
4. Connect. Accept the host key the first time, as you did with `ssh`.

You'll see **local** files on one side and **remote** files on the other. The remote side is the VPS disk. Dragging a file **uploads**. Dragging the other way **downloads**.

Field	Typical value
Protocol	SFTP
Host	<code>203.0.113.10</code> (your IP)
Port	<code>22</code>
User	<code>root</code> or <code>ubuntu</code>
Password / key	Same as SSH

If FileZilla asks for FTP, switch to SFTP. FTP sends passwords in the clear and is the wrong tool for a VPS. A connection refused on 21 is a hint you pointed at FTP by mistake.

Remote path examples: `/home/harry`, `/var/www/html`, `/etc/nginx`. Dropping HTML into `/var/www/html` is how you update the Nginx page from your laptop without `nano` on the server.

After an upload, permissions still matter. A file owned by `root` and mode `600` may not be readable by Nginx (`www-data`). Fix with `chown` / `chmod` on the server, not by making everything `777`.

The same job in the terminal

FileZilla is optional. These three copy a local file to the server:

```
scp page.html ubuntu@203.0.113.10:/var/www/html/
sftp ubuntu@203.0.113.10
rsync -av page.html ubuntu@203.0.113.10:/var/www/html/
```

`scp` is a one-shot copy. `sftp` opens an interactive session (`put`, `get`, `ls`, `cd`, `bye`). `rsync` copies and is better when you sync a whole folder repeatedly.

Download the other way (server) → laptop:

```
scp ubuntu@203.0.113.10:/var/www/html/index.html .
sftp ubuntu@203.0.113.10
rsync -av ubuntu@203.0.113.10:/var/www/html/index.html .
```

In `sftp`, `get index.html` pulls a file; `put index.html` pushes one.

Copy a whole directory:

```
scp -r site/ ubuntu@203.0.113.10:/var/www/html/
rsync -av site/ ubuntu@203.0.113.10:/var/www/html/
```

`rsync` trailing slashes matter: `site/` copies the **contents**; `site` can nest an extra `site/` folder on the destination. `-a` preserves permissions and timestamps. `-v` is verbose.

Use a key the same way as SSH:

```
scp -i ~/.ssh/id_rsa page.html ubuntu@203.0.113.10:/var/www/html/
sftp -i ~/.ssh/id_rsa ubuntu@203.0.113.10
rsync -av -e "ssh -i ~/.ssh/id_rsa" page.html ubuntu@203.0.113.10:/var/www/html/
```

What to use when

FILEZILLA

browsing folders visually, occasional uploads, not living in the terminal.

SCP

one file or a small tree, quickly.

RSYNC

a project folder you keep updating; it only transfers what changed.

All three talk to the same Linux machine. Nginx doesn't care whether `index.html` arrived from FileZilla, `scp`, or `nano`. It serves whatever is in the web root.



Conclusion

You started with a word people argue about - kernel, operating system, distribution - and ended with a Linux machine you can log into, install software on, and put a web page on.

That machine might be a VPS, a VirtualBox guest, or WSL. The commands did not change. `ls` lists a directory. `systemctl` talks to services. Nginx serves files. Cron runs overnight without you watching.

What you can do now

- Tell a **kernel** from a **distribution**, and pick Ubuntu LTS on purpose.
- Reach a box with `ssh`, and know that the prompt is **that** computer, not your laptop.
- Create users, put them in groups, and read `-rwxr-xr--` without guessing.
- Install and update packages with `apt`, and know that `update` is not `upgrade`.
- Find a process, stop a service, read its logs.
- Put a value in `PATH` without wiping `PATH`.
- Pack a folder with `tar` / `gzip` / `zip`.
- Schedule a job, with absolute paths, and a log file.
- Walk `/`, `/home`, `/etc`, `/var`, `/usr` and know why they exist.
- Install Nginx, drop an `index.html` in `/var/www/html`, and move files with FileZilla or `scp`.

Habits worth keeping

Test config before you reload (`nginx -t`). Prefer `kill` before `kill -9`. Prefer `chmod 644` / `755` before `777`. Prefer `usermod -aG` over `-G` alone. Write down the root password. Use LTS.

The next time something breaks, you already have a path: **status** → **logs** → **process list** → `top`. The next time you need a file on the server, you already have three ways to put it there.

Linux will keep feeling large. That's normal. The tree still starts at `/`. The shell still searches `PATH`. The kernel still schedules the CPU. Everything in this handbook is a way of looking at that same machine - clearly enough to change it on purpose.

