ADMIN:DOCS

What is the Linux kernel?

The kernel is not Ubuntu and not the shell. What it does, where Linux comes from, and how you recognise the running version with uname, /proc and lsmod.

Category: Linux beginners **Published:** 12.09.2026 **Last reviewed:** 13.09.2026

Once you have used Linux for a while, the word **kernel** keeps turning up. An update installs a new one, a driver wants a particular kernel version, and after some upgrades you are suddenly told to reboot. Only: **what is this kernel actually – and why does so much hang on it?**

In short: the kernel is the part of Linux that sits between your programs and the hardware. Start a program, open a file or send data across the network, and the kernel is involved. It hands out CPU time and memory, talks to hardware through drivers, and decides which process may access what. Without it, nothing runs. Really nothing.

The kernel is not what you normally see as “Linux”. KDE or GNOME are not part of it, nor is Bash, and even `systemd` runs outside the kernel. Ubuntu, Debian or Fedora are not the kernel either. They are complete distributions that ship it together with the rest of userspace.

That split matters. The moment a driver misbehaves, an old kernel is still running after an update, or you want to know what `/proc` and `/sys` actually show, “Linux is just the operating system” does not get you very far.

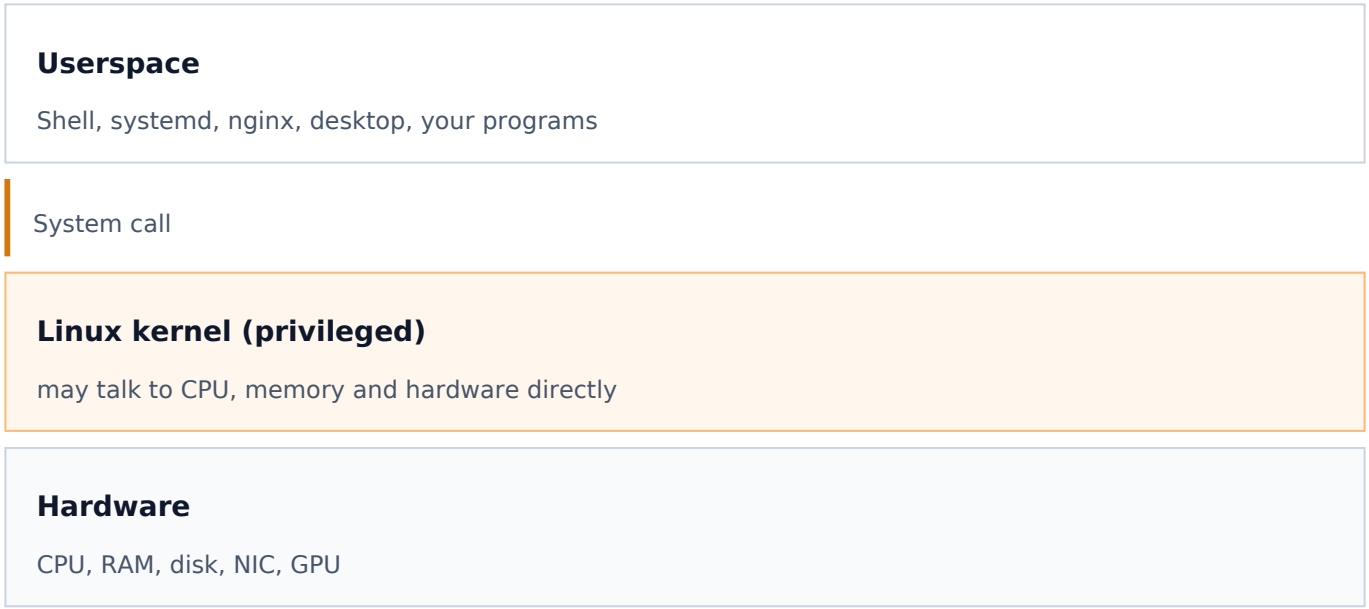
A running Linux install and a terminal are enough for the examples. No custom kernel, and nobody is unloading modules for fun. This is about what is already working on your machine all the time.

> **Typical mix-up:** “Linux” in everyday speech often means the whole installation. Technically, Linux is the kernel. Ubuntu, Fedora or Debian are distributions: kernel plus userspace, packages, init system and often a desktop.

Source: https://admindocs.de/en/linux-beginners/what-is-the-linux-kernel

Kernel, userspace and distribution

What you see on screen almost never runs *in* the kernel. Editor, browser, shell, web server: that is **userspace**. The kernel sits underneath. It runs in the CPU’s privileged mode, may talk to hardware directly, allocate memory and preempt processes. A userspace program may not. It asks through a **system call**.



The shell interprets your input and starts programs. Whatever those programs then need from files, memory, network or hardware goes to the kernel through system calls.

What the kernel is

Linux uses a **Unix-like**, monolithic kernel. That sounds denser than it is: drivers, filesystems, the network stack and memory management run in privileged kernel space. Many parts can still be loaded as modules.

What it actually does – and what it does not:

The kernel is

Layer
: privileged go-between for programs and hardware

Work
: processes, memory, drivers, VFS, network, permissions

Boundary
: reachable only via system call

Project
: the Linux kernel itself

The kernel is not

Layer
: the distribution (Ubuntu, Fedora, Debian)

Work
: `systemd`, `bash` or the package manager

Boundary
: the desktop (KDE, GNOME)

Project
: GNU as a whole

“Monolithic” means these parts share the same privileged address space. A driver is not a mini operating system sitting next to it; it is kernel code. A loaded module belongs to the kernel. It is just not baked into every installation.

Why people often say GNU/Linux

GNU in particular often ends up in the same pot as the kernel. The **GNU project** has shipped free userspace tools (`gcc`, `coreutils`, `bash`) since the 1980s. Many desktop and server distributions combine exactly that with the Linux kernel. That is why people often say GNU/Linux. There are counter-examples: Android uses the same kernel with a different userspace. BusyBox systems do too.

Source: https://admindocs.de/en/linux-beginners/what-is-the-linux-kernel

> **Tip:** When someone says “reinstall Linux”, they almost always mean the distribution. A kernel update is narrower: it swaps the privileged core and usually needs a reboot before the new version is actually running.

From Unix to Linux

Linux is not a renamed Unix source tree. The *idea* comes from Unix: a multi-user system that bundles processes, files and devices through a kernel. The *code* comes from Linus Torvalds and thousands of later maintainers. That difference still matters later for licences and certification.

1. **1969** Bell Labs: Unix begins (Thompson, Ritchie)
2. **1977** BSD at Berkeley: its own Unix line
3. **1983** System V (AT&T); GNU project, kernel still missing
4. **1987** MINIX (Tanenbaum): teaching Unix on the PC
5. **1991** Torvalds: Linux 0.01 / 0.02 on the 386
6. **1992** GPL-2.0; kernel and GNU tools used together

Unix at Bell Labs

Unix began at Bell Laboratories in the late 1960s and was considerably smaller than many mainframe systems of the time. **Ken Thompson** and **Dennis Ritchie** built a system that was soon rewritten in C. C made the kernel portable – and that is exactly what set Unix apart from many operating systems glued to one machine.

In the late 1970s **BSD** (Berkeley Software Distribution) at the University of California became an important Unix line. AT&T later bundled its commercial Unix work into **System V**, which appeared in 1983. System V later spawned variants such as Solaris and HP-UX.

Both lines remain kernel plus userspace as *one* operating system. The source was proprietary for a long time, or available only under tight licences. Anyone who wanted Unix on a PC in the

1980s bought expensive licences or made do with teaching systems.

GNU, MINIX and the gap on the PC

In 1983 **Richard Stallman** announced the GNU project: a fully free Unix-like system. Compilers, editors and base tools appeared. Its own kernel, **GNU Hurd**, became a microkernel project and did not reach everyday use for a long time.

In 1987 **Andrew S. Tanenbaum** published **MINIX**, a Unix-like teaching system for PCs. Small, readable, built as a microkernel: drivers and filesystems as far as possible outside the core. Ideal for study. Too tightly licensed and too limited as a full PC system for many users.

Exactly in that gap – free userspace available, a usable free kernel for the 386 PC missing – Linux begins.

Linus Torvalds and the first kernels

In 1991 **Linus Torvalds** was a computer-science student at the University of Helsinki. He wanted to understand how a 386 processor works in protected mode, and a system that went beyond MINIX. On 25 August 1991 he wrote to the MINIX newsgroup that he was working on a free operating system, “just a hobby, won't be big and professional like gnu”, for 386 and 486 AT clones.

The first public version **0.01** followed in September 1991, the one announced as usable, **0.02**, on 5 October 1991: a small kernel, around ten thousand lines, still depending on MINIX to develop. The working name was **Freax** for a while. On the university FTP server the archive landed as **Linux** – and the name stuck.

Important for the record: Torvalds wrote the kernel **anew**. He did not copy MINIX source. The resemblance was in the Unix idea and, for a time, the filesystem layout, not in a code lineage from AT&T Unix.

Early in 1992 he put the kernel under the **GNU General Public License Version 2**. From 1 February 1992, study, change and redistribution were allowed as long as derived kernel code stayed under the same copyleft licence. The GPL created the licence basis for others to change and pass on the kernel.

> **Context:** The 1992 debate between Tanenbaum and Torvalds was about microkernel versus monolithic kernel. Tanenbaum thought the monolithic design was obsolete. Torvalds thought it was simpler and faster on real PC hardware. Linux stayed monolithic – with modules that load drivers later, without rebuilding the whole kernel.

Linux 1.0 appeared on 14 March 1994. Torvalds has kept the mainline since; stable fixes are among other things looked after by **Greg Kroah-Hartman**. The source tree lives on kernel.org. The stable branch there is currently the **7.2** series (presently 7.2.5); longterm branches run in parallel, including **6.18**, which distributions use for years of bug and security fixes. Your install almost always uses the distro kernel, not the day-to-day mainline tarball.

GPL and the development model

A classic Unix kernel belonged to a company or a university line. Patches ran through licences and product cycles. The Linux kernel is a shared Git archive:

- Subsystem maintainers take patches for drivers, filesystems, net or architecture.
- Torvalds merges into the main archive.
- Stable and longterm branches focus on fixes and selected backports rather than new feature series.

What follows for you is sober: you do not have to build the kernel yourself to understand it. You need to know **which version is running** – and that a package update is the running kernel only after the reboot.

Linux kernel and Unix kernel compared

Linux is **Unix-like**, not “Unix” in the trademark or certification sense. In the terminal you barely notice: `ls` feels familiar. Under the hood it is a different program with a different licence and a different ABI.

POSIX describes portable interfaces. Many Linux systems are POSIX-close. Linux distributions usually do not buy the expensive **Single UNIX Specification** certification. Commercial Unix

systems historically did, because the “UNIX” mark was a selling point.

Classic Unix kernel

Origin
: AT&T Unix, later System V or BSD

Licence
: historically proprietary; BSD later permissive

Shape
: monolithic

Userspace
: tied to the OS variant

Interface
: Unix ABI of that line

Hardware
: server and workstation lines

Development
: vendor or BSD project

Linux kernel

Origin
: written anew in 1991, Unix ideas, no AT&T code

Licence
: GPL-2.0-only

Shape
: monolithic, drivers often as modules

Userspace
: independent: GNU, Android, BusyBox

Interface
: Linux syscall ABI; POSIX via the C library

Hardware
: PCs, servers, phones, supercomputers, embedded

Development
: kernel.org, companies and volunteers

Shared: processes, memory, files, net, permissions

BSD kernels (FreeBSD, OpenBSD, NetBSD) sit closer to Linux than old System V Unix: open source, monolithic, Unix heritage. Differences remain. BSD ships kernel and userspace as **one** system. Linux ships the kernel; the distro chooses init, C library and tools. Linux routinely loads drivers as modules. BSD builds more of that into the kernel, with its own module tradition.

That is why `ls` feels similar on Solaris, FreeBSD and Linux. The kernel underneath is still a different program in each case.

> ⚠ **Warning:** “Linux is a Unix” is a sloppy shortcut. Correct: Linux is a Unix-like kernel. Programs can often be ported. Binaries, kernel modules and internal interfaces are not the same.

What the kernel does

Imagine every program were allowed to drive CPU, memory and disk on its own. That ends in chaos. The kernel is the go-between that stops it.

Processes and scheduling

Every running program is a **process** (more precisely: one or more threads). The kernel creates them, separates their address spaces and decides who gets the CPU next. That is **scheduling**.

You notice it when a compiler loads the machine and the shell still responds. The kernel hands out time slices. It does not freeze the system for one program.

A shell typically starts external programs through mechanisms such as `fork` and `exec` – both kernel work. Internal built-ins such as `cd` stay in the shell, because a child process cannot change the parent shell’s directory.

Memory

Your program behaves at first as if it had its memory to itself. It does not, of course. The kernel keeps that split in place and decides which virtual pages actually land in RAM.

Each program therefore sees a **virtual address space**. The kernel maps virtual pages to physical RAM. Under memory pressure it can drop unused pages or – if swap exists – page them out. A process must not simply read another process’s memory. That is the boundary between userspace programs. The kernel itself sits outside that unprivileged space.

System calls

Userspace does not just walk into the kernel. The path is a system call. Typical examples, conceptually, not a C reference:

- `open` / `read` / `write` / `close` for files and many devices,
- `fork` / `execve` / `exit` for processes,
- `socket` / `connect` / `bind` for the network,
- `mmap` for memory mappings.

Source: https://admindocs.de/en/linux-beginners/what-is-the-linux-kernel

1. **Userspace** `cat file.txt` calls `open()` / `read()`
2. **Linux kernel** VFS + driver; check permissions, fill buffers
3. **Storage** filesystem supplies the bytes
4. **Back to the process** output on stdout

`cat` does not open a circuit board. `cat` calls the kernel. The kernel checks permissions, finds the filesystem driver and hands bytes back.

Drivers and hardware

Without a matching driver the kernel can do precious little with your network card. Sitting in a PCIe slot is not enough. Only the driver knows how that particular piece of hardware must be talked to – down to registers and DMA.

The same goes for graphics, memory controllers, USB and filesystem drivers. A missing driver shows up as “device not recognised”, not as a magical userspace error.

Filesystems

Applications see paths such as `/etc/hosts`. Underneath sit ext4, XFS, btrfs, NFS or something else entirely. The kernel puts the **Virtual File System (VFS)** in between. The same `open` family serves local disks, network shares and special files under `/proc` and `/sys`.

`/proc` and `/sys` are virtual filesystems. Their contents do not live on the SSD like ordinary files; they expose running kernel and system state. That is why you can query the running kernel there without guessing at a config file.

Networking

Sockets, routing, firewall hooks, TCP/IP processing: that lives in the kernel. Userspace programs speak `connect` and `send`. Whether packets leave the NIC is up to the network stack plus driver.

Source: https://admindocs.de/en/linux-beginners/what-is-the-linux-kernel

Tools such as `ss` or `ip` read that state. They *are* not the stack.

Permissions and isolation

Every process runs with an identity (UID, GID, plus capabilities). File access is checked by the kernel against inode permissions. Details are in the article on <code class="badge-code">chmod. Here one sentence is enough: the shell cannot “override” permissions. If the kernel refuses access, you still get `Permission denied`.

That is also one of the big differences between a container and a VM: a container does not bring its own kernel. It simply uses the host’s. A VM boots its own. So a container is not a tiny VM, even if it looks that way at first glance. Light weight has a price here: you are tied to the host kernel version.

Modules: loadable kernel code

A desktop kernel does not hard-wire every Wi-Fi chip on earth. Many drivers arrive as a **module** (`.ko`). The kernel loads them when needed, often automatically via `udev`.

□□ Practical example:

Look at the loaded modules:

```
lsmod | head
```

Details on a particular module, if that name exists on your system:

```
modinfo xhci_hcd
```

`lsmod` lists what is *in* the kernel now. `modinfo` describes a module file even if it is not loaded.

> ⚠ **Warning:** `rmmod` or `modprobe -r` unloads kernel code. On a machine whose network or disk access needs exactly that module, that is a fast way to a hung system. Reading is enough as an exercise. Unloading does not belong in beginner day-to-day work.

Looking at the running kernel

Theory without your own machine stays folklore. The following commands are read-only. You usually do not need root; `dmesg` can be restricted on some systems.

Version and build

□□ Practical example:

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

`-r` is the running release, for example a distro string such as `6.8.0-60-generic` or a 7.2 variant on a rolling distribution. `-a` adds machine, processor and the build text.

The same information lives in the kernel itself:

```
cat /proc/version
```

`/proc/version` is not an archive on disk. It is a statement from the running kernel.

> **Tip:** Compare `uname -r` with what the package manager shows as installed. If they differ, a new kernel is installed but not yet booted.

Messages at boot

The kernel writes to a ring buffer. After boot it records which hardware it found and which drivers it bound.

```
journalctl -k -b | less
```

`-k` limits to kernel messages, `-b` to the current boot. Under systemd that is usually the convenient path. You read the ring buffer itself like this:

```
dmesg | less
```

On many distributions unprivileged users may not read `dmesg`. The kernel parameter `kernel.dmesg_restrict` controls that. If `dmesg` returns nothing or `Operation not permitted`, stay with `journalctl -k` or an invocation with the right privileges.

Loaded modules and a hardware glance

```
lsmod
```

```
ls /sys/module | wc -l
```

`/sys/module` is the sysfs view of modules. Again kernel state, not a config file you maintain by hand.

Package versus running version

On Debian and Ubuntu the kernel image package shows what is *installed*:

```
uname -r
dpkg -l 'linux-image-*' | awk '/^ii/ { print $2, $3 }'
```

On Fedora or RHEL analogously `rpm -q kernel` or `dnf list installed kernel`. The package name is distro business. The rule is the same everywhere: **installed ≠ running**, until you have rebooted.

> **Typical failure:** After a kernel update “nothing looks different” until the machine has loaded the new kernel. Userspace services can restart without a reboot. The kernel cannot.

From power-on to userspace

At power-on, Linux is not running yet. Firmware (UEFI or older BIOS) starts the bootloader. The bootloader loads the kernel and usually an **initramfs**: a small, temporary root filesystem in RAM so the kernel has disk and crypto drivers before the real root filesystem is there.

1. **Firmware**UEFI / BIOS

2. **Bootloader** e.g. GRUB
3. **Linux kernel + initramfs**
4. **First process** `systemd` / init
5. **Userspace** services, login, shell, desktop

From then on the kernel is “only” the go-between – the go-between without which no service exists. GRUB menu entries, UKI, Secure Boot: separate topics.

What you deliberately skip as a beginner

Understanding the kernel is not the same as *building* it.

Building your own kernel as a beginner buys you surprisingly little at first. As long as your distribution ships a matching kernel, let it do its job.

- Delete old kernel packages only after you have booted the new one successfully. The previous bootloader entry is the rescue if an update does not fit.
- No `rmmod` on memory, disk or network drivers “to try it”.
- Mainline from kernel.org is for developers and distro packagers. On the working machine, stay with the distro kernel.

Anyone who later builds kernels needs a test box, not the only laptop.

Command Reference (Cheatsheet)

Command / path	Function	Typical example
`uname -r`	running kernel release	`uname -r`

Command / path	Function	Typical example
`uname -a`	release, machine, build text	`uname -a`
`/proc/version`	version text of the running kernel	`cat /proc/version`
`journalctl -k -b`	kernel messages of this boot	`journalctl -k -b	less`
`dmesg`	kernel ring buffer (watch permissions)	`dmesg	less`
`lsmod`	loaded modules	`lsmod	head`
`modinfo`	metadata of a module file	`modinfo xhci_hcd`
`/sys/module`	sysfs view of modules	`ls /sys/module`
`ls /boot`	installed kernel images (distro)	`ls /boot/vmlinuz*`

Further Resources

Resource	Description	Type / link
The Linux Kernel Archives	Current stable and longterm branches	kernel.org
Kernel administration docs	Kernel README and admin guide	Kernel admin-guide
`kernel.dmesg_restrict`	Who may read the kernel ring buffer	sysctl: dmesg_restrict
Linux 0.01 release notes	Historical first release on kernel.org	RELNOTES-0.01
Torvalds, 25 August 1991	Original announcement on `comp.os.minix`	Newsgroup posting
GNU GPL 2.0	Licence text the kernel is under	GNU GPL-2.0
GNU and Linux	Relationship of GNU userspace and the Linux kernel	Linux and GNU
POSIX	Portable operating-system interfaces	POSIX / IEEE 1003
AdminDocs: command line	Shell starts programs; the rest goes through system calls	Command-line processor on Linux

Source: https://admindocs.de/en/linux-beginners/what-is-the-linux-kernel

Resource	Description	Type / link
AdminDocs: permissions	What the kernel checks on file access	chmod on Linux

Conclusion

Anyone who administers Linux works with the kernel constantly, even without compiling it. `uname`, `/proc`, `/sys`, kernel logs and modules are therefore not specialist topics for kernel developers. They are ordinary tools for looking at a system.

After an update it is not the package that decides which kernel is really running, but the next boot. That distinction is what separates a working diagnosis from guesswork.

> **Tip:** After the next distro update, note `uname -r` before and after the reboot. The difference is the kernel, not the feeling that “it somehow got updated”.