```
$ whoami
your_team  •  backend / frontend / mobile / devops
```

```
$ uname -a
Linux prod-web-03 6.8.0-generic
$ uptime
 14:02:31 up 214 days, load 0.42
$ systemctl is-system-running
running
$ _
```

Linux for Engineers Who Ship Software

A practical, no-fluff tour of the OS that runs your code

90-minute session

All engineering roles

Hands-on & command-driven

Session field notes

Internal engineering knowledge-share

OPENING

~3 min

Why Linux Is Not “Just DevOps’ Problem”

It's where your code actually lives

Staging, production, CI runners, and most cloud VMs are Linux - your code's real home is not your laptop.

It's where things break in the open

A crash, a memory leak, or a bad deploy surfaces as a Linux symptom first: a process, a log line, a permission error, an exit code.

It's underneath every tool you already use

Docker, Kubernetes, GitHub Actions runners, Android, and most “serverless” platforms are Linux wearing a costume.

```
$ docker exec -it api sh
/ # ps aux | grep node
1  node  server.js
/ # cat /proc/1/status | grep -i mem
VmRSS:    612344 kB
/ # exit

# same tools, same commands,
# every environment you deploy to.
```

~1 min

Session Roadmap

01

Foundations

History, distros, architecture

`~20 min`

02

System Internals & Control

Boot, systemd, permissions, processes

`~35 min`

03

Operating Linux Day-to-Day

Filesystem, packages, users, logs

`~30 min`

Plus: role mapping, live debugging walkthroughs, a recap, and a survival cheat sheet to take with you.

~3 min

What Linux Actually Is

Linux is a kernel, not an OS

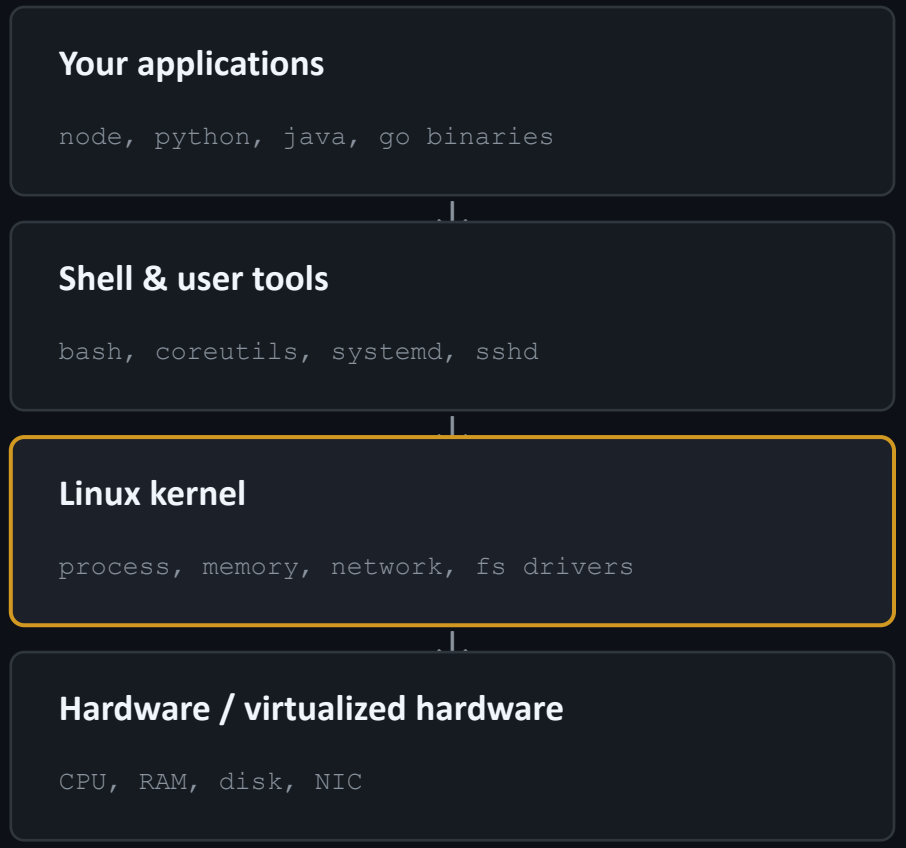
It's the core program that talks directly to hardware: CPU scheduling, memory, devices, and the filesystem. Nothing more, nothing less.

“Linux” as you know it is kernel + tooling

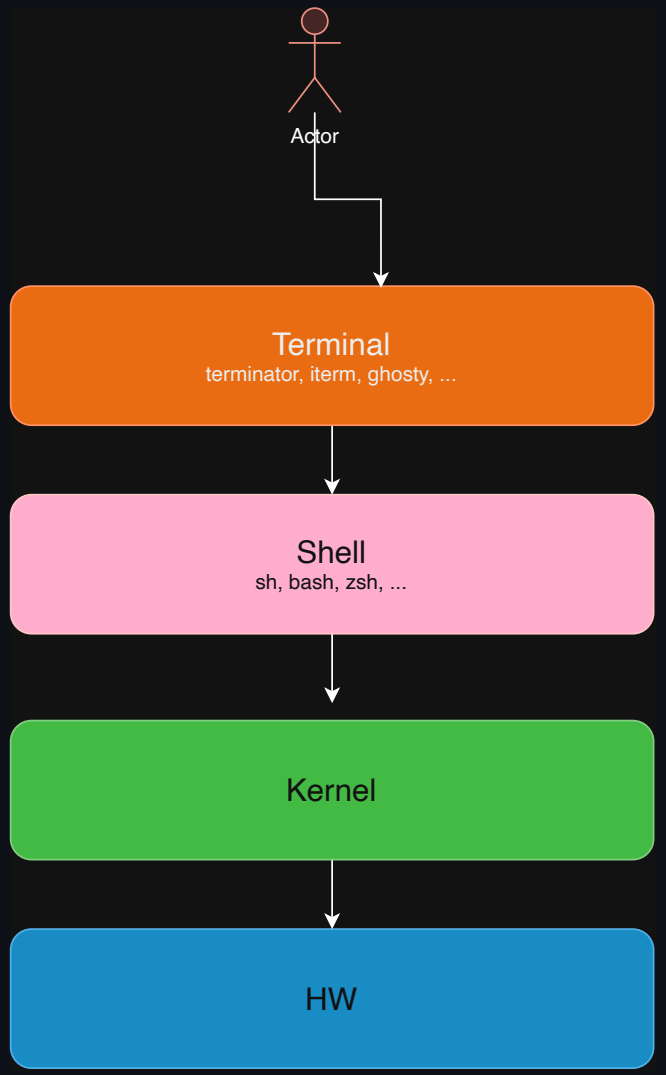
Shells, compilers, package managers, systemd - all separate projects packaged around the kernel by a distro.

You interact with the kernel indirectly

Every command you run asks the kernel to do something on your behalf through a system call.



What’s the difference between shell, terminal, and bash?


Actor
Terminal
terminator, iterm, ghosty, ...
Shell
sh, bash, zsh, ...
Kernel
HW

Why Linux Runs the World

~96%

of the top 1M web servers

100%

of the world's top 500 supercomputers

3B+

Android devices (Linux kernel inside)

Nearly all

public cloud VMs and container hosts

Same kernel, radically different jobs: a web server, a phone, a CI runner, and a supercomputer node are all running variations of the same core.

It's free, open, and endlessly customizable

No licensing cost, no vendor lock-in, source code you can actually read.

It scales from a smartwatch to a data center

The same kernel design runs embedded devices and thousand-node clusters.

Why Linux is endlessly customizable?

1- Everything is a file
2- pseudo filesystems
.
.
.
.
.

~4 min

Linux in Your Daily Work

Backend

- Debugging OOM kills & crashed processes
- Reading app + system logs under load
- File permissions on uploads / secrets
- Understanding container resource limits

Frontend

- Node-based build tooling runs on Linux CI
- Debugging “works locally, fails in CI”
- Reverse proxy / static hosting basics
- Reading deploy & build logs

Mobile

- Android runs a real Linux kernel
- Emulator & build-farm troubleshooting
- adb shell is a Linux shell
- CI runners for app builds are Linux

DevOps / SRE

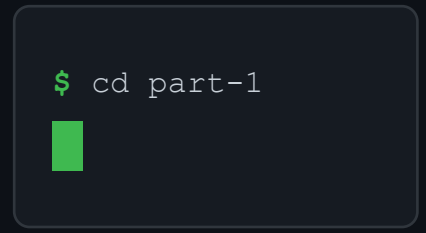
- Everything above, plus:
- Provisioning, orchestration, systemd
- Incident response & root cause
- Security, users, and access control

PART 1

Foundations

Where Linux came from, and how the pieces fit together

- From UNIX to GNU/Linux: the history that explains the naming
- The kernel, Linus Torvalds, and the open-source ecosystem
- Distros and package managers - and why servers pick different ones
- Desktop vs. server architecture, and why most servers skip the GUI



~3 min

From UNIX to GNU/Linux

1969

UNIX is born at Bell Labs

Ken Thompson & Dennis Ritchie build UNIX; its ideas - everything is a file, small composable tools - still shape Linux today.

1983

The GNU Project starts

Richard Stallman sets out to build a free, UNIX-compatible operating system: compilers, editors, shells - everything except a kernel.

1991

Linus Torvalds writes a kernel

A Finnish student posts a hobby kernel to a newsgroup. It slots perfectly into the missing piece of GNU's toolset.

Today

GNU/Linux, open source at scale

Thousands of companies and volunteers co-develop the kernel and tooling in the open - most of the internet's infrastructure runs on the result.

Why “GNU/Linux”? The kernel is Linux; most of the surrounding tools - shell, compilers, coreutils - come from GNU. Linus wrote the engine; GNU built the rest of the car.

The History and Evolution of Unix and Linux

Bell Labs Unix (1969)

Developed at Bell Labs by Ken Thompson, Dennis Ritchie, and colleagues.

1969
The original Unix implemented in C, designed for PDP-7.

University versions (BSD)

Berkeley Software Distribution derived from AT&T Unix, developed at UC Berkeley.

1970s
Freely available Unix variants for research and education.

- 4.2BSD (1983)
- 4.3BSD (1986)
- 4.4BSD (1993)
- FreeBSD (1993)
- NetBSD (1993)
- OpenBSD (1996)

Many modern systems (including macOS) have BSD heritage.

Commercial Unix

AT&T licensed Unix to companies who developed their own proprietary versions for business use.

1980s
Proprietary Unix systems for enterprise and commercial use.

- IBM AIX (1986)
- HP-UX (1986)
- Sun Solaris (1992)
- digital Tru64 UNIX (1994)
- ORACLE Solaris ((later) Oracle Solaris)

Linux (1991)

Created by Linus Torvalds. Unix-like kernel written from scratch, inspired by Unix but not derived from it.

1991
Unix-like, but written from scratch and open source.

Linux powers thousands of systems today:

Servers · Cloud · Mobile · Desktops · Embedded · ...

Key

- Origin
- University / Open Source Lineage
- Commercial / Proprietary Lineage
- Modern Open Source

Richard Stallman Resume

- Richard Stalman print machine in MIT
- GNU Project
- Free Software Foundation
- Major Software Contributions
    - Emacs
    - Gcc
    - …
- GNU General Public License (GPL)

~3 min

The Kernel & the Open Source Engine Behind It

Developed in the open, by thousands

Linux kernel development happens on public mailing lists and Git repos - anyone can read every patch that ever went in.

Corporate-funded, community-governed

Companies like Intel, Google, Red Hat, and Meta pay engineers to contribute - but no single company owns it.

A new kernel release roughly every 9-10 weeks

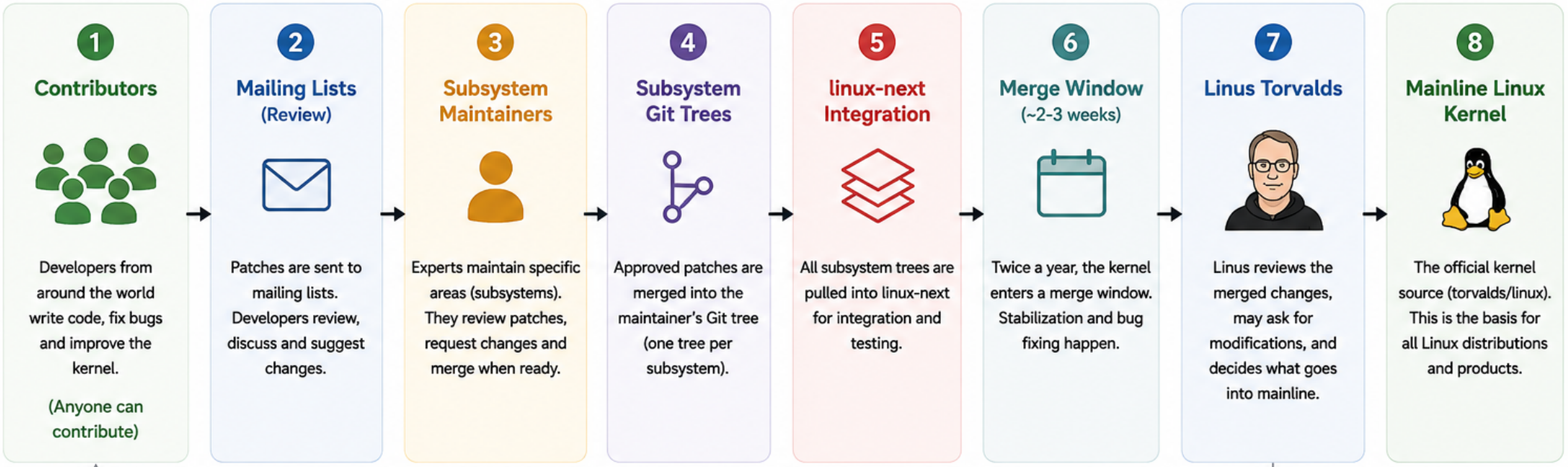
Continuous, predictable releases - distros then pick a version and stabilize it for their users.

```
$ uname -r
6.8.0-45-generic
$ cat /proc/version
Linux version 6.8.0 (gcc 13.2)
$ git log --oneline -1  # kernel.org
a1b2c3d Merge branch 'net-next'
```

Why you should care: this is the same open development model behind Kubernetes, Docker, and most of your CI/CD toolchain. It's not a Linux quirk - it's how modern infrastructure gets built.

How the Linux Kernel is Developed

From contributors to Linus Torvalds and mainline



Who Contributes?

- Individual Developers
- Red Hat Engineers
- Google Engineers
- Intel Engineers
- AMD Engineers
- Arm Engineers
- Microsoft Engineers
- Many Others

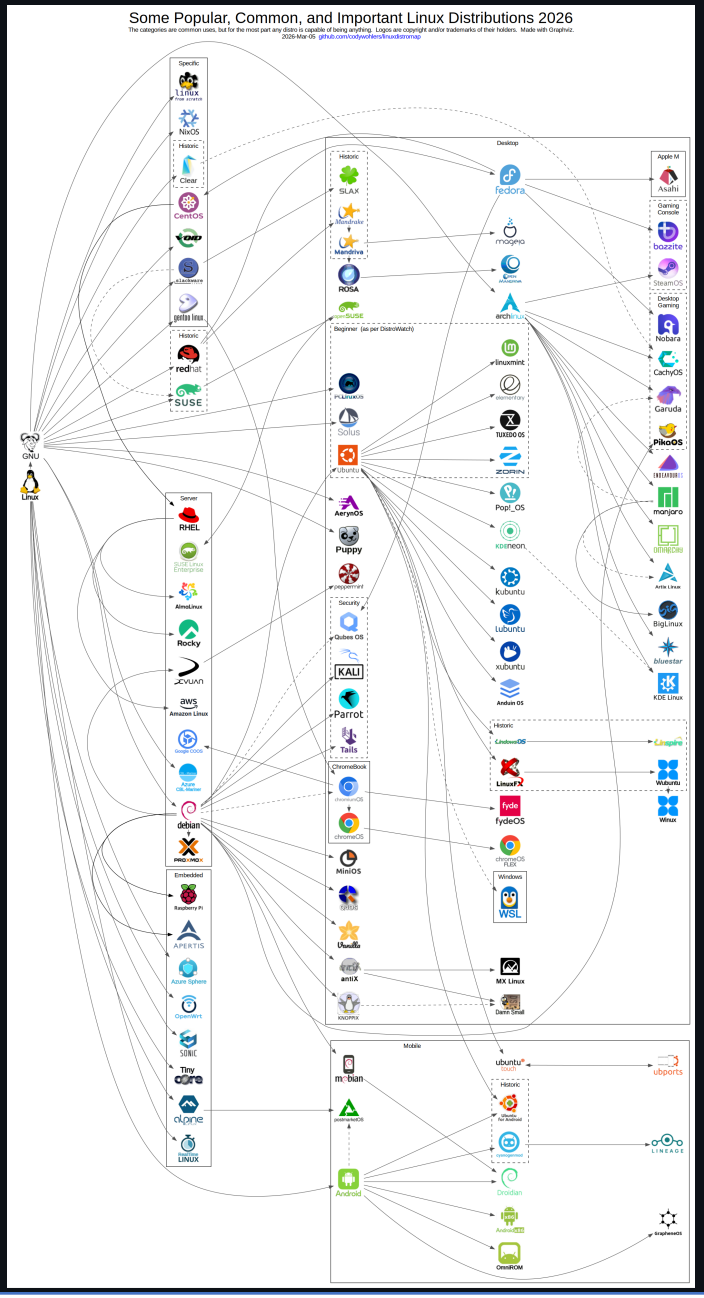
The Linux kernel is 100% open source. Anyone can read the code, participate, and contribute.
Merit, review and collaboration drive every change.

~4 min

Distros & Package Managers

A distro = kernel + tooling + package manager + defaults, bundled and supported as one coherent system.

Distro	Family	Package mgr	Typical use
Ubuntu	Debian-based	`apt`	General purpose, cloud, dev machines
Debian	Debian	`apt`	Stability-first servers
RHEL	Red Hat	`dnf / yum`	Enterprise servers, long support
Fedora	Red Hat	`dnf`	Cutting-edge desktop / upstream to RHEL
Arch	Independent	`pacman`	Rolling release, full control, DIY
Alpine	Independent	`apk`	Minimal size - the default for containers



https://github.com/codywohlers/linuxdistromap

~3 min

Server Distros vs. Desktop Distros

Server distros optimize for stability & size

Minimal packages, longer support windows, headless by default, predictable upgrade cycles.

Desktop distros optimize for usability

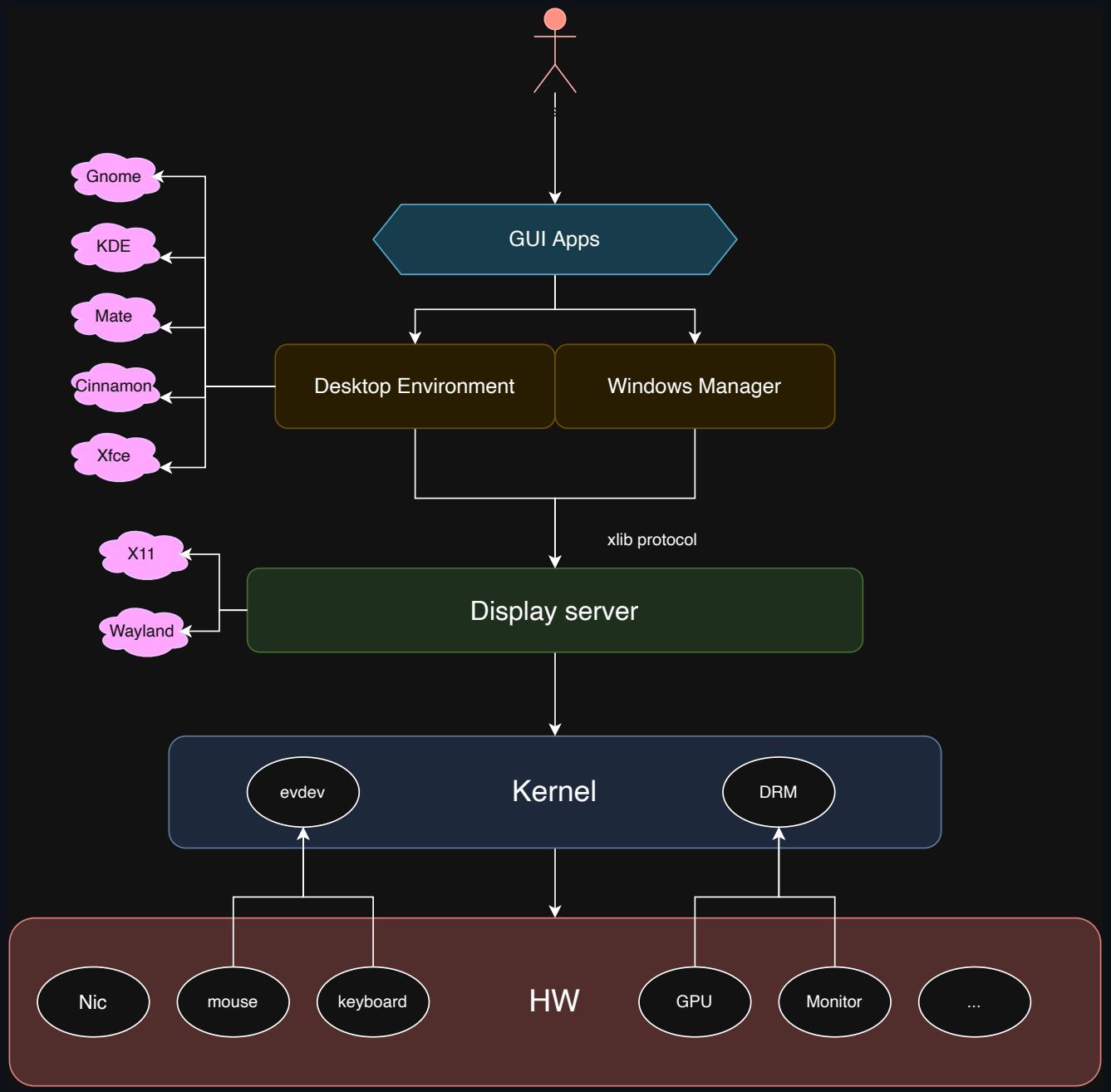
GUI, drivers, media codecs, fast-moving package versions - great for laptops, wrong for prod.

In containers, size and surface area matter most

Smaller base images pull faster, patch faster, and give attackers less to work with.

```
$ docker images
REPOSITORY    TAG       SIZE
ubuntu        22.04     77.8MB
debian        bookworm  124MB
alpine        3.19      7.38MB

# same "FROM" line, wildly different
# footprint - alpine is ~10-15x
# smaller than ubuntu/debian

FROM alpine:3.19
RUN apk add --no-cache nodejs npm
```

Linux GUI Design



~4 min

Desktop vs. Server Architecture

X11 vs. Wayland, at a glance

Two protocols for drawing windows on screen. X11 is the old, flexible standard; Wayland is the modern, simpler, more secure replacement most distros now default to.

Desktop environments sit on top

GNOME and KDE bundle a display protocol, window manager, and apps into one cohesive desktop experience.

Servers skip all of it

No display server, no window manager, no desktop environment - just the kernel, systemd, and the services you actually run.

Desktop stack

```
GNOME / KDE (desktop environment)
Wayland / X11 (display server)
systemd (init & services)
Linux kernel
```

Server stack

```
Your services (nginx, node, postgres...)
systemd (init & services)
Linux kernel
```

PART 2

System Internals & Control

The technical core: boot, services, permissions, and processes

- What happens between power-on and a running system
- systemd: units, targets, and services
- Users, permissions, and the principle of least privilege
- Processes, signals, and finding what's misbehaving

```
$ cd part-2
```

~4 min

What Happens When Linux Boots

BIOS / UEFI

Hardware self-test, finds a bootable disk

Bootloader

GRUB loads the kernel into memory

Kernel

Kernel initializes, mounts root filesystem

initramfs

Temporary root fs with drivers to reach real root fs

systemd (PID 1)

First real process; starts every other service

Target reached

e.g. multi-user.target - system is “up”

```
$ systemd-analyze
Startup finished in 2.1s (kernel) + 4.8s (initrd) + 9.3s (userspace) = 16.2s
$ systemd-analyze blame | head -3
3.210s  cloud-init.service
1.845s  docker.service
```

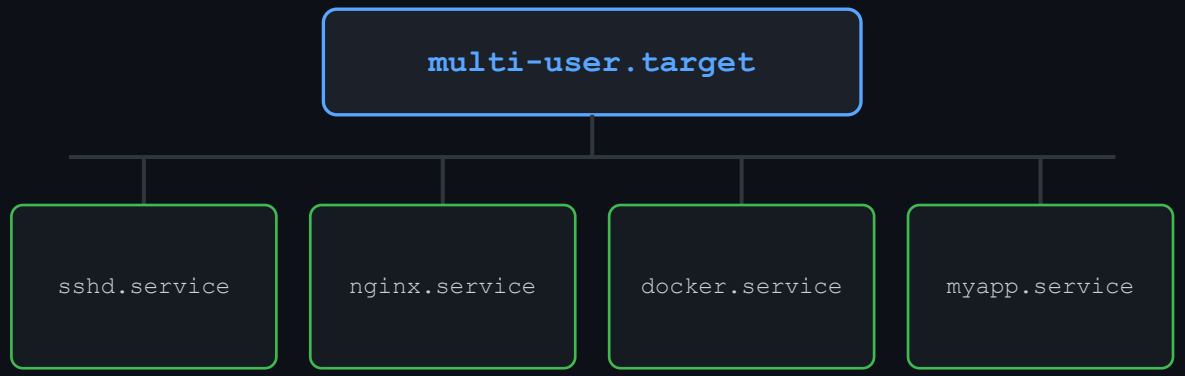
systemd: Units, Targets & Services

Unit
The basic building block systemd manages: a service, a mount, a timer, a socket.

Target
A named group of units to reach - like multi-user.target (“normal running system, no GUI”).

Service
A unit type that runs and supervises a program, restarting it if it dies, if configured to.

Every long-running program you deploy - nginx, your API, docker, sshd - is a systemd service unit.




```
$ cat /etc/systemd/system/myapp.service
[Unit]
Description=My API server
After=network.target

[Service]
ExecStart=/usr/bin/node /srv/app/index.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

1. Before systemd: the Unix way
   Init

2. System V init
   Works with some scripts

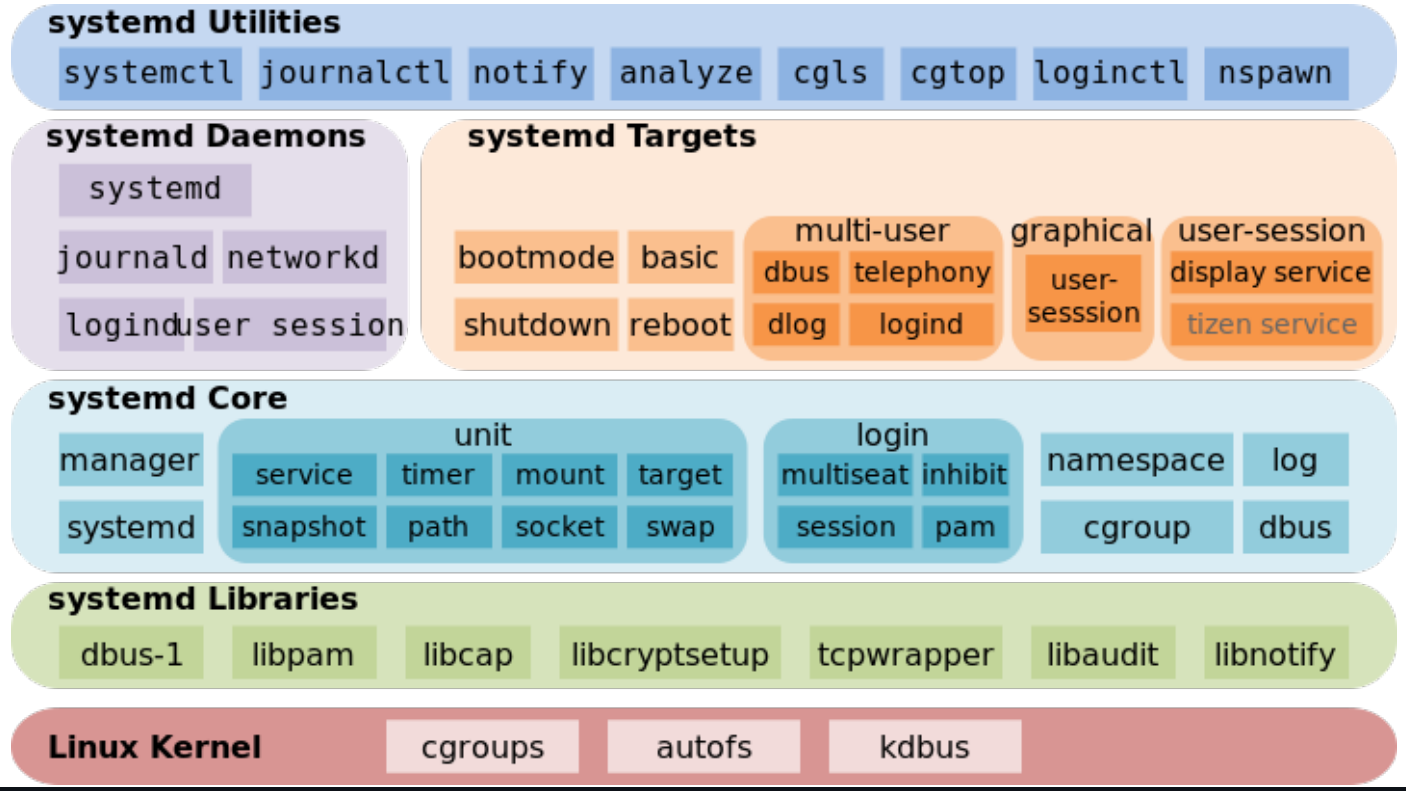
3. Systemd
   Works with units, target, and services

Alternatives: unit, s6, OpenRC, Upstart, ….

Table 1

0	→	**halt**
1	→	single-user/rescue
2	→	multi-user
3	→	multi-user + networking
5	→	graphical
6	→	reboot

systemd
│
├── service (.service) → Runs a program
├── target (.target) → Groups other units
├── timer (.timer) → Starts a unit on a schedule
├── socket (.socket) → Starts a service when a socket is used
├── mount (.mount) → Mounts a filesystem
├── device (.device) → Represents a hardware device
└── path (.path) → Watches a file or directory




```
debian@backup:/etc/systemd/system$ cat sshd.service
[Unit]
Description=OpenBSD Secure Shell server
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target auditd.service
ConditionPathExists=!/etc/ssh/sshd_not_to_be_run

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStartPre=/usr/sbin/sshd -t
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/usr/sbin/sshd -t
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartPreventExitStatus=255
Type=notify
RuntimeDirectory=sshd
RuntimeDirectoryMode=0755

[Install]
WantedBy=multi-user.target
Alias=sshd.service
```

A timer schedules another unit.

A **unit** is any resource that `systemd` manages.

A service runs a process.

A target **doesn't run programs**. Instead, it groups other units to define a desired system state.

Target	Purpose	GUI	Networking	Typical Use
`graphical.target`	Full desktop	✅	✅	Desktop PCs, laptops
`multi-user.target`	Multi-user text mode	❌	✅	Servers
`rescue.target`	Single-user recovery	❌	Limited	System repair
`emergency.target`	Minimal emergency shell	❌	❌	Serious boot failures
`poweroff.target`	Shut down	❌	❌	Power off
`reboot.target`	Restart	❌	❌	Reboot
`halt.target`	Stop CPU without powering off	❌	❌	Rarely used
`suspend.target`	Suspend to RAM	❌	Restored on wake	Sleep mode
`hibernate.target`	Suspend to disk	❌	Restored on wake	Hibernate

~5 min

Debugging a Service That Won't Start

`systemctl status myapp`

Is it running? What was the last error?

`systemctl start / stop / restart myapp`

Control the service directly

`systemctl enable / disable myapp`

Start automatically on boot, or not

`journalctl -u myapp`

Full log history for this one unit

```
$ systemctl status myapp
● myapp.service - My API server
   Active: failed (Result: exit-code)
   Process: 8842 ExecStart=/usr/bin/node
            (code=exited, status=1)

$ journalctl -u myapp -n 20 --no-pager
Error: Cannot find module 'dotenv'
myapp.service: Main process exited,
  code=exited, status=1/FAILURE
myapp.service: Failed with result 'exit-code'

# root cause found: a missing dependency
# from the deploy, not a mystery crash
$ npm install --production && systemctl restart myapp
```

~4 min

Users, Groups, Root & sudo

Every process runs as a user

That user (and its groups) determines exactly what the process is allowed to touch - files, ports, devices.

root can do (almost) anything

UID 0 bypasses most permission checks - which is exactly why you don't run app servers as root.

sudo grants temporary, logged root power

Run one command as another user (usually root) without ever sharing the root password.

Principle of least privilege

Give every user and service only the access it needs - nothing “just in case.”

```
$ whoami
deploy
$ id
uid=1001(deploy) gid=1001(deploy)
groups=1001(deploy),999(docker)

$ sudo systemctl restart nginx
[sudo] password for deploy:

$ sudo -l
User deploy may run:
  (root) /usr/bin/systemctl restart nginx

# scoped sudo - deploy can restart nginx,
# nothing more
```

~5 min

File Permissions: Why “Permission Denied” Happens

rwxr-xr--

Owner	Group	Others
read · write · execute	read · write · execute	read · write · execute

chmod
Change what owner / group / others can do: chmod 750 deploy.sh

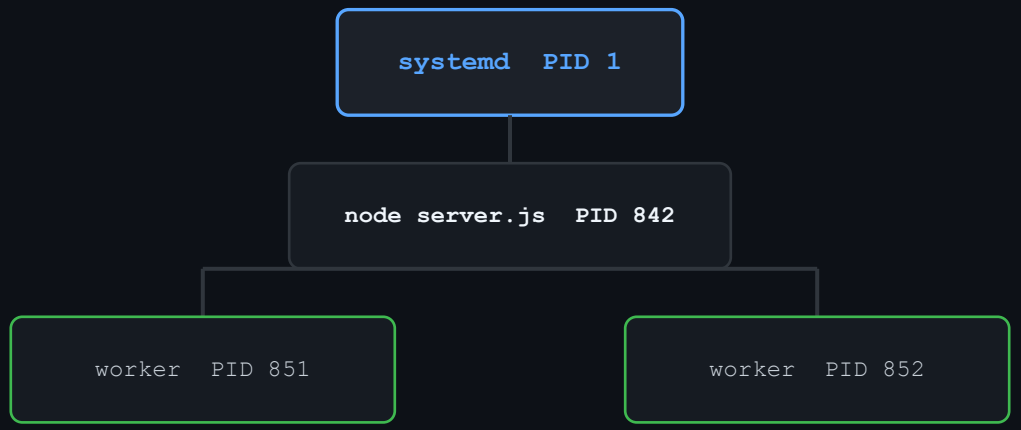
chown
Change which user owns a file: chown appuser app.log

chgrp
Change which group owns a file: chgrp deploy config.yml

```
$ ./deploy.sh
bash: ./deploy.sh: Permission denied

$ ls -l deploy.sh
-rw-r--r-- 1 root root 812 deploy.sh
# owner is root, and there's no 'x' bit
# for anyone - nobody can execute it

$ chmod +x deploy.sh
$ chown deploy:deploy deploy.sh
$ ls -l deploy.sh
-rwxr-xr-- 1 deploy deploy 812 deploy.sh

$ ./deploy.sh
Deploying v1.4.2...  ✓
```

~4 min

Processes: PID, Parent/Child & Signals



PID 842 is the parent of 851 and 852. If 842 dies, its children are usually reparented or killed depending on how they were started.

Foreground vs. background
& backgrounds a process; fg/bg and job control move it between them.

Common signals

SIGTERM (15)
“Please shut down” - the polite, default request

SIGKILL (9)
“Die now” - kernel force-kills, no cleanup possible

SIGHUP (1)
“Reload config” - many daemons re-read config on this

SIGINT (2)
What Ctrl+C sends to a foreground process

ZOMBIE PROCESS

Not alive, not dead...
just waiting to be cleaned up!

1 BIRTH OF A PROCESS

2 LIFE OF THE CHILD

3 THE CHILD FINISHES

4 WAITING FOR DAD

5 ZOMBIE MODE

6 DAD FINALLY CHECKS

THE BIG IDEA

A **zombie process** is a finished process that is waiting for its parent to **CLEAN IT UP.**

Small but important:
Too many zombies?
Your system will feel the pain ☹

~5 min

Finding a Stuck Process or Port Conflict

`ps aux`

List every process on the system

`top / htop`

Live view: CPU, memory, per-process

`kill <pid> / kill -9 <pid>`

Ask, then force, a process to stop

`nice / renice`

Set or change CPU scheduling priority

`lsof -i :PORT`

Find what's bound to a given port

```
$ npm start
Error: listen EADDRINUSE :3000

$ lsof -i :3000
COMMAND  PID   USER  NAME
node    18422  deploy  *:3000 (LISTEN)

# a previous run never shut down cleanly
$ kill 18422
$ lsof -i :3000
# (nothing - port is free)

$ npm start
Server listening on :3000  ✓
```

PART 3

Operating Linux Day-to-Day

Filesystem conventions, software installation, users, and logs

- → Where config, binaries, logs, and data actually live
- → Installing and managing software without guesswork
- → Why “works on my machine” happens - and how to prevent it
- → Creating service users and reading logs like an incident responder

```
$ cd part-3
```

~4 min

The Linux Filesystem Hierarchy

What's in /etc

Config files

```
debian@knowledge-sharing:/etc$ ls
NetworkManager          ca-certificates       dbus-1          fstab      hosts.allow      ld.so.conf.d   magic.mime      nsswitch.conf  profile.d   rc6.d       shadow
  sudo_logsrvd.conf  ucf.conf
X11                     ca-certificates.conf  debconf.conf    gai.conf   hosts.deny       ldap           mime.types      opt            protocols   rcS.d       shadow-
  sudoers            udev
acpi                    cloud                 debian_version  group      init.d           libaudit.conf  mke2fs.conf     os-release     python3     resolv.conf shells
  sudoers.d          ufw
adduser.conf            cracklib              default         group-     initramfs-tools  locale.alias   modprobe.d      pam.conf       python3.11  resolvconf  skel
  sv                 update-motd.d
alternatives            cron.d                deluser.conf    grub.d     inputrc          locale.gen     modules         pam.d          qemu        rmt         ssh
  sysctl.conf        vim
apparmor.d              cron.daily            dhcp            gshadow    iproute2         localtime      modules-load.d  passwd         rc0.d       rpc         ssl
  sysctl.d           wgetrc
apt                     cron.hourly           dib-manifests   gshadow-   issue            logcheck       motd            passwd-        rc1.d       runit       subgid
  systemd            xattr.conf
bash.bashrc             cron.monthly          dpkg            gss        issue.net        login.defs     mtab            perl           rc2.d       security    subgid-
  terminfo           xdg
bash_completion         cron.weekly           e2scrub.conf    host.conf  kernel           logrotate.d    netconfig       polkit-1       rc3.d       selinux     subuid
  timezone           xml
bindresvport.blacklist  cron.yearly           environment     hostname   ld.so.cache      machine-id     network         ppp            rc4.d       services    subuid-
  tmpfiles.d
binfmt.d                crontab               ethertypes      hosts      ld.so.conf       magic          networks        profile        rc5.d       sgml        sudo.conf
  tuned
debian@knowledge-sharing:/etc$
```

/etc/sudoers

```
root    ALL    =    (ALL:ALL)    ALL
|       |           |          |
|       |           |          └─ Any command
|       |           └─ As any user : any group
|       └─ From any host
└─ Who gets the permission
```

/etc/sysctl.d

This directory contains Linux kernel parameter configuration files.
A later setting can override an earlier setting for the same parameter.

10-foo.conf
50-bar.conf
99-sysctl.conf

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
```

```
sudo sysctl --system
```

/etc/systemd

```
debian@knowledge-sharing:/etc/systemd$ ls
journald.conf  logind.conf  network  networkd.conf  pstore.conf  sleep.conf  system  system.conf  timesyncd.conf  user  user.conf
debian@knowledge-sharing:/etc/systemd$ cd system/
debian@knowledge-sharing:/etc/systemd/system$ ls
cloud-init.target.wants                multi-user.target.wants      sockets.target.wants    sysinit.target.wants
dbus-org.freedesktop.timesync1.service network-online.target.wants sshd-keygen@.service.d systemd-resolved.service.wants
getty.target.wants                     paths.target.wants           sshd.service            timers.target.wants
debian@knowledge-sharing:/etc/systemd/system$
```

/etc/crontab

```
debian@knowledge-sharing:/etc$ cat crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || { cd / && run-parts --report /etc/cron.daily; }
47 6    * * 7   root    test -x /usr/sbin/anacron || { cd / && run-parts --report /etc/cron.weekly; }
52 6    1 * *   root    test -x /usr/sbin/anacron || { cd / && run-parts --report /etc/cron.monthly; }
#
debian@knowledge-sharing:/etc$
```

/etc/fstab

/etc/fstab

```
# <file system> <mount point>      <type>    <options>                    <dump>  <pass>
# /etc/fstab: static file system information.

UUID                                    MOUNT POINT    TYPE     OPTIONS              DUMP   PASS
UUID=6d3b2f1a-5e6c-4c1e-9d3b-8a7b1c2d3e4f    /            ext4     defaults             0      1
UUID=9f8e7d6c-3b2a-4d1f-a6e1-7b2c3d4e5f6a    /home        ext4     defaults             0      2
UUID=3a2b1c0d-9e8f-4a1b-b2c3-d4e5f6a7b8c9    /mnt/data    xfs      defaults,nofail,_netdev  0      2
```

/etc/hosts.deny

`/etc/hosts.deny` is part of the old **TCP Wrappers** mechanism on Linux. It was used to control which clients were allowed or denied access to certain network services.

sshd: 192.168.1.100

ALL: ALL

/etc/hostname

hostname: Contains the machine's own hostname.
hosts: Maps hostnames to IP addresses locally.

/etc/logrotate.d

```
/var/log/nginx/*.log {
    daily —-> Rotate the logs every day
    rotate 14 —-> Keep 14 old rotations
    compress —-> Compress old logs
    delaycompress —-> Don't compress the most recently rotated log immediately
    missingok —-> If the log doesn't exist
    notifempty —-> If the log is empty
}
```

AppArmor

Application security for Linux

AppArmor is a Linux Security Module (LSM) that confines programs by restricting the files, capabilities and resources they can access.

Why AppArmor?

Reduces attack surface
Limits what an application can do if it is compromised.

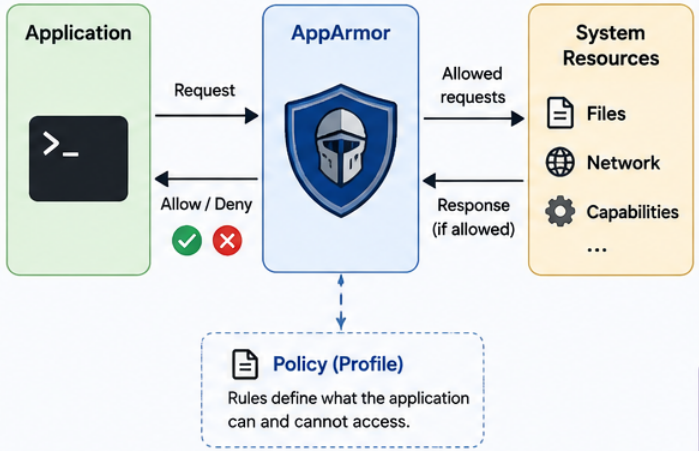
Least privilege
Grants only the minimum access required for the application to work.

System-wide protection
Built into the Linux kernel (LSM), always on and hard to bypass.

Easy to manage
Ships with many default profiles and simple tools.

How AppArmor Works

AppArmor sits between an application and system resources.
It checks every request and allows or denies it based on a profile.



Profile Example

Profiles are written in a simple language that whitelists/blacklists actions.

```
# Sample AppArmor profile
/usr/bin/nginx {
  # Allow reading configuration
  /etc/nginx/          r,
  /etc/nginx/**        r,

  # Allow network access
  network inet stream,

  # Allow reading certificates
  /etc/ssl/**          r,

  # Deny everything else by default
  deny /**             wklx,
}
```

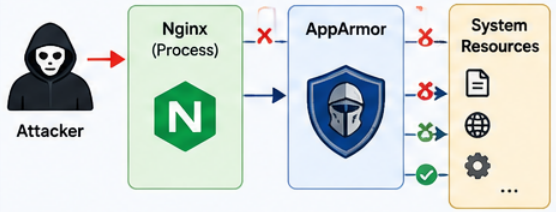
Enforcement Modes

- **Enforce (default)** – Violations are blocked and logged.
- **Complain** – Violations are logged but allowed.
- **Disable** – Profile is loaded but not enforced.
- **Unconfined** – No profile applied.

Example: Protecting Nginx

If an attacker exploits Nginx, AppArmor prevents it from:

- Writing to sensitive files
- Executing arbitrary programs
- Accessing the network it doesn't need

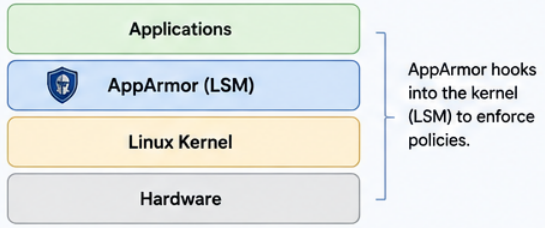


= Blocked by AppArmor = Allowed by AppArmor

Common Management Commands

```
# Show AppArmor status
sudo aa-status                          # Overall status

# List loaded profiles
sudo aa-status --profiles               # Show all profiles

# Put a profile in complain mode
sudo aa-complain /usr/bin/nginx         # Log only, don't block

# Enforce a profile again
sudo aa-enforce /usr/bin/nginx          # Enforce mode

# Disable a profile
sudo aa-disable /usr/bin/nginx          # Profile not applied

# Create a new profile interactively
sudo aa-genprof /usr/bin/myapp          # Walk through permissions

# Check logs (violations are logged to syslog/journal)
sudo journalctl -k | grep DENIED
                                        # View denied events
```

Where AppArmor Fits



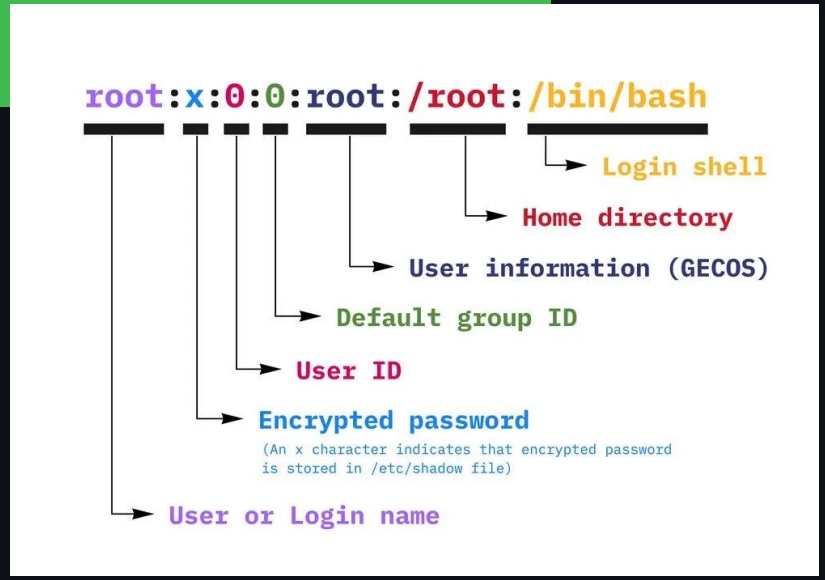
Learn More

https://apparmor.net/

man 5 apparmor.d

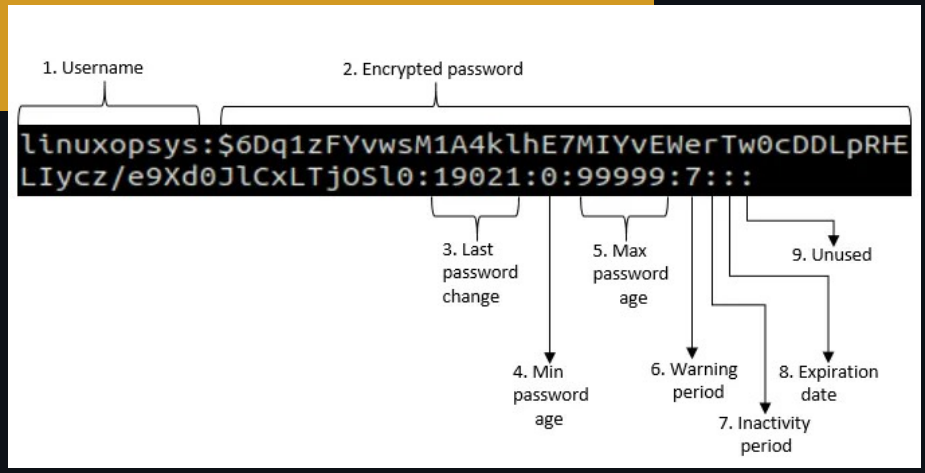
/etc/passwd File:

The /etc/passwd file is a **plain-text database** housing fundamental user information. Each line in the file represents a user account and is divided into fields **separated by colons (:)**



/etc/shadow File:

For **heightened security**, critical user authentication information, **particularly hashed passwords,** resides in the /etc/shadow file, accessible exclusively to privileged users.



/etc/ssh and /etc/sshd

- [ ] **Authentication**
  Keys only, no root/passwords
- [ ] **Access control**
  Allow-list users, firewall
- [ ] **Cryptography**
  Modern ciphers, ed25519 keys
- [ ] **Session limits**
  Timeouts, connection throttling
- [ ] **Forwarding**
  Disable X11, TCP, agent
- [ ] **Logging**
  Verbose logs, fail2ban
- **Key hygiene**
  Strict file permissions
- [ ] **Misc**
  Banner, patching, ssh-audit
- **SFTP-only**
  Chroot, forced command

What's in /var

Logs, variable data

/var/lib/docker

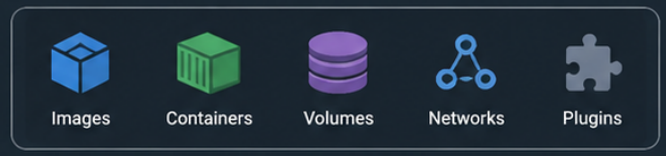
Docker's data root directory – stores all Docker resources and state on the host.

You can change this location with:
`{ "data-root": "/path/to/docker" }`
in `/etc/docker/daemon.json`

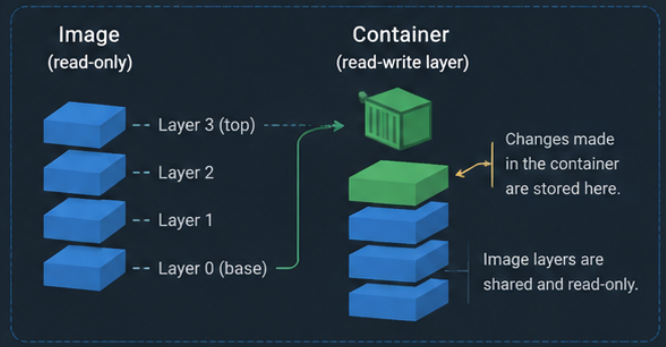
`/var/lib/docker`

- `buildkit/` — BuildKit data (build cache, records, etc.)
- `containers/` — Writable layer of containers (changes made inside running containers)
- `image/` — Images and layers (read-only layer data)
  - `overlay2/`
  - `overlay/`
  - `imagedb/`
  - `content/`
- `volumes/` — Named volumes data
  - `<volume_name>/_data/`
- `network/` — Networking data (networks, endpoints, etc.)
- `plugins/` — Docker plugins
- `runtimes/` — OCI runtimes
- `swarm/` — Swarm state (if Docker Swarm is used)
- `tmp/` — Temporary files
- `trust/` — Docker trust data
- `builder/` — Legacy builder cache (older Docker versions)
- `.tmp/` — Internal temporary directory
- `metadata.db` — Internal metadata database (BoltDB)
- `config/` — Daemon and client configuration
- `key.json` — TLS key for communication (e.g., with registry)

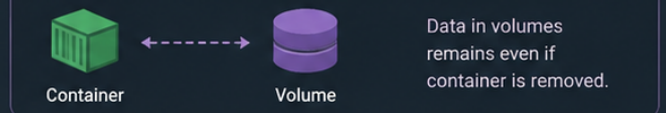
What lives here?



How images & containers use storage



Persistent data with volumes



Good to know

- Deleting containers does not delete images or volumes.
- Use `docker system prune` to clean unused data.
- Large size here usually means images, containers or volumes are consuming space.

Useful commands

```
$ docker system df          # See space usage
$ docker system prune -a    # Remove unused data (careful!)
$ docker volume ls          # List volumes
$ docker image ls           # List images
```

All Docker data lives under `/var/lib/docker` by default (on Linux). Back it up if needed, and monitor its size.

What's in /home

User directories

Where is root home dir?

~3 min

Mounts, Paths & Symlinks: Where Things Live

Mounts attach a device to a path

A separate disk, a USB stick, or a network share is “mounted” onto a directory - there's no C:\ drive letter, everything hangs off one root tree.

Absolute vs. relative paths

/etc/nginx/nginx.conf is absolute (always the same file); config/app.yml is relative to wherever you currently are.

Symlinks point to another path

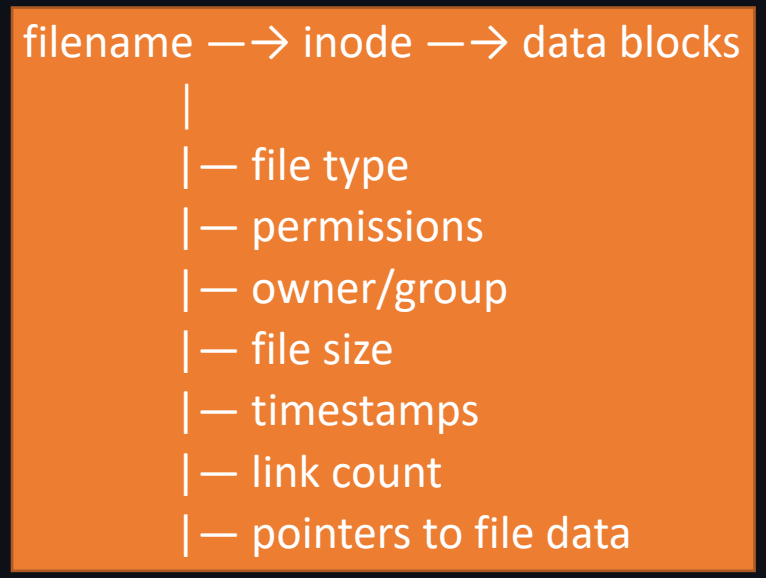
A lightweight pointer - deleting the link doesn't delete the target, and updating the target updates everywhere it's linked.

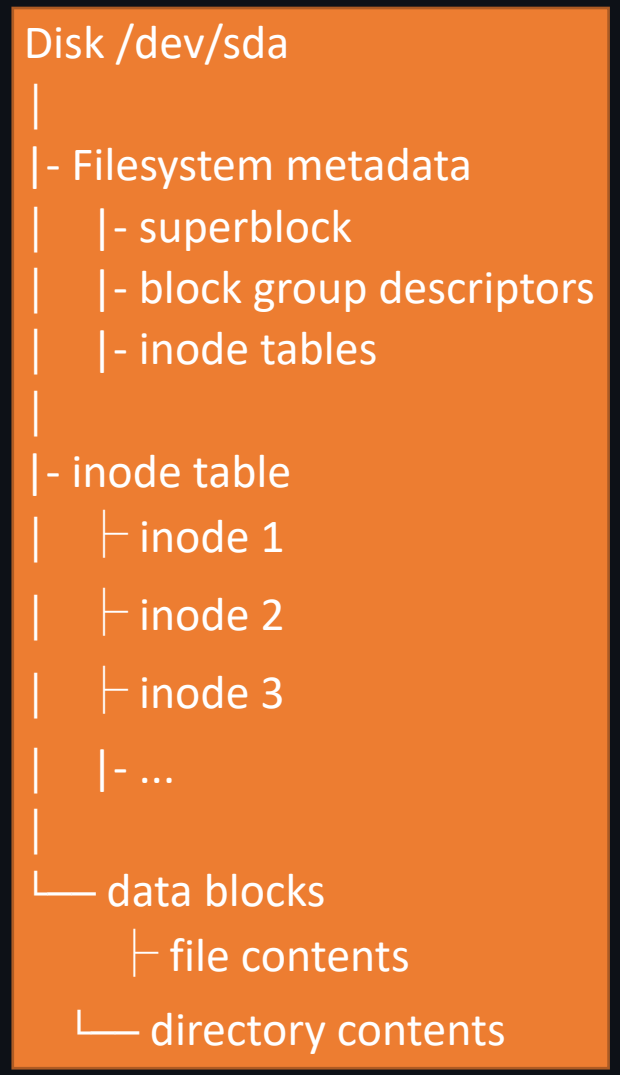
```
$ df -h
Filesystem  Size  Used  Mounted on
/dev/sda1    40G   18G  /
/dev/sdb1   100G   62G  /data

$ ls -l /etc/nginx/sites-enabled/
app.conf -> /etc/nginx/sites-available/app.conf

$ find / -name '*.log' 2>/dev/null | head -3
/var/log/nginx/access.log
/var/log/app/myapp.log
/var/log/syslog
```

Symlinks VS Hardlink

Feature	Soft link (Symlink)	Hard link
Points to	A **path/name**	The same **inode**
Can cross filesystems	✅ Yes	❌ No
Can point to directories	✅ Yes	Usually ❌
If original is deleted	❌ Link becomes broken	✅ Still works
Inode	Different inode	**Same inode**
Command	`ln -s file link`	`ln file link`

An **inode** is a data structure used by Linux filesystems to store **metadata about a file**, but not usually its filename or actual file content.





~4 min

Installing & Managing Software

Package managers resolve dependencies for you

apt, dnf, pacman, apk all do the same job: fetch a package plus everything it needs from a trusted repository.

Update your index before you install

apt update refreshes the list of what's available; apt upgrade actually installs newer versions.

Three different layers of “installing”

OS packages (apt), language packages (npm/pip), and containers (docker pull) all solve different problems - know which one you're touching.

cd /etc/apt

```
$ sudo apt update
Fetched 2,847 kB in 1s

$ sudo apt install nginx
The following NEW packages will be
installed: nginx nginx-common ...
Setting up nginx ...

$ systemctl status nginx
● active (running)

$ sudo apt remove nginx
$ sudo apt list --installed | grep nginx
```

Why “Works on My Machine” Happens

Different package versions

Your laptop has Node 20 installed globally; the CI image pins Node 18.

Different OS / libc

glibc (Debian/Ubuntu) vs. musl (Alpine) handle some native modules differently.

Different permissions or users

Your local user owns everything; the container runs as a restricted, non-root user.

Missing environment configuration

An env var, secret, or config file quietly exists on your machine and nowhere else.

The fix is always the same idea: make the environment reproducible - pin versions, build from the same base image, and never depend on something that only exists on one machine.

Why a Docker Container Behaves Differently

A container shares your kernel, not your OS

It's an isolated process tree + filesystem, running on the host's Linux kernel - not a separate virtual machine.

Different default user

Many images run as root by default inside the container; some run as a locked-down non-root user - permissions differ accordingly.

A separate, ephemeral filesystem

Files written inside the container vanish when it's removed, unless you mount a volume - a common source of “where did my data go.”

```
$ docker run --rm -it myapp:latest sh
/ # whoami
node
/ # id
uid=1000(node) gid=1000(node)
/ # cat /etc/os-release | head -1
PRETTY_NAME="Alpine Linux v3.19"
/ # ls /app
dist  node_modules  package.json

# same code, different distro, different
# user, different filesystem - three good
# reasons behavior can shift
```

Namespace

1. PID namespace
2. Mount namespace
3. Network namespace
4. UTS namespace (domain name, hostname, …)
5. IPC namespace (Inter-Process Communication)
6. User namespace
7. Cgroup namespace
8. Time namespace

Cgroup

group processes together and control how many resources they can use.

~4 min

User Management for Services

`useradd -r -s /usr/sbin/nologin appuser`

Create a service account - no login shell

`usermod -aG docker deploy`

Add an existing user to a group

`passwd deploy`

Set or change a password

`groups deploy`

Show which groups a user belongs to

`id appuser`

Show UID, GID, and group membership

```
$ sudo useradd -r -m -d /srv/app \
    -s /usr/sbin/nologin appuser
$ sudo chown -R appuser:appuser /srv/app

$ sudo -u appuser node /srv/app/index.js

# the app now runs as a dedicated,
# unprivileged user - not root, and
# not your personal login either

$ id appuser
uid=999(appuser) gid=999(appuser)
groups=999(appuser)
```

Logging: syslog, journald & App vs. System Logs

System logs vs. application logs

systemd/kernel/auth events live in journald or /var/log; your app's own log output is a separate stream you configure.

journald is the modern default

Structured, indexed, queryable by service, time, or priority - systemd's built-in logging system.

/var/log still matters

Many services (nginx, some databases) still write plain-text logs directly to files under /var/log.

Where to look first

`journalctl`

Every systemd-managed service's logs

`/var/log/syslog`

General system events (Debian/Ubuntu)

`/var/log/nginx/`

Web server access & error logs

`/var/log/auth.log`

Login & sudo activity

Syslog / journald

What's in /var/log/nginx

What's in /var/log/syslog

What's in /var/log/auth.log

What's in /var/log/kern.log

What's in /var/log/cron

What's in /var/log/postgresql

cat VS tail

~5 min

Troubleshooting a Failed App Using Logs

```
journalctl -u myapp -f
```

Follow logs live, as they happen

```
journalctl -u myapp --since "10 min ago"
```

Only recent entries

```
journalctl -p err -u myapp
```

Only error-level and above

```
tail -f /var/log/nginx/error.log
```

Follow a plain-text log file live

```
grep -i "timeout" app.log
```

Search log content for a keyword

```
$ journalctl -u myapp --since "10 min ago"
myapp: connecting to db at db-primary:5432
myapp: connection timeout after 5000ms
myapp: retrying (1/3)...
myapp: connection timeout after 5000ms
myapp: retrying (2/3)...
myapp: FATAL: could not reach database

$ ping -c 2 db-primary
ping: db-primary: Name or service
  not known

# root cause: DNS resolution, not the
# app or the database itself
```

Recap: Key Takeaways

Linux is a kernel

Everything else - shell, tools, systemd - is assembled around it by a distro.

systemd runs your services

Units, targets, and journalctl explain almost every “why did it stop” question.

Permissions follow every process

Users, groups, and least privilege decide what can touch what.

Processes and signals are how you intervene

ps, kill, and lsof find and stop what's misbehaving.

The filesystem hierarchy is a map

/etc, /var/log, and /home aren't arbitrary - they're a convention you can rely on.

Logs are your first move in any incident

journalctl and /var/log tell you what happened before you start guessing.

~2 min

Linux Survival Commands

Services

```
systemctl status <svc>
systemctl restart <svc>
journalctl -u <svc> -f
```

Processes

```
ps aux | grep <name>
top / htop
kill <pid>  |  kill -9 <pid>
lsof -i :<port>
```

Permissions

```
ls -l
chmod +x file
chown user:group file
sudo -l
```

Files & disk

```
df -h
du -sh *
find / -name "pattern"
tail -f file.log
```

Packages

```
apt update && apt install pkg
apt remove pkg
dpkg -l | grep pkg
```

System info

```
uname -a
uptime
whoami  |  id
```

Questions?

Bring your weirdest “Linux did something confusing” story - there's a good chance it maps to something from today.

```
$ echo "thanks for coming"
thanks for coming
$ man <anything>   # start here next time
$ _
```