FROM SPINNING PLATTERS TO SMART PARTITIONS

A deep dive into storage, filesystems, and Linux partitioning

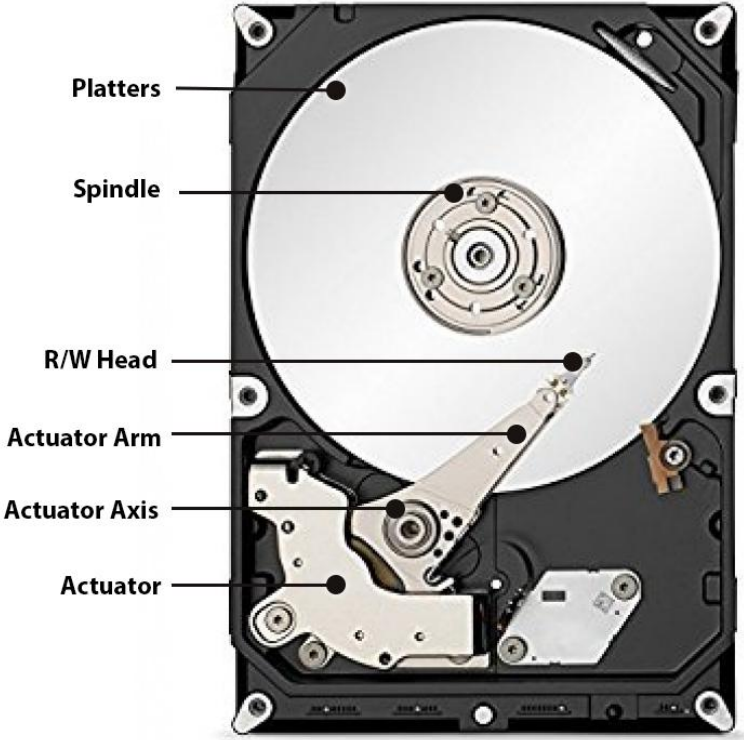
Hard drives · SSDs · Filesystems · COW vs Journaling · Partitioning Strategy

SECTION 1

What Is a Hard Drive?

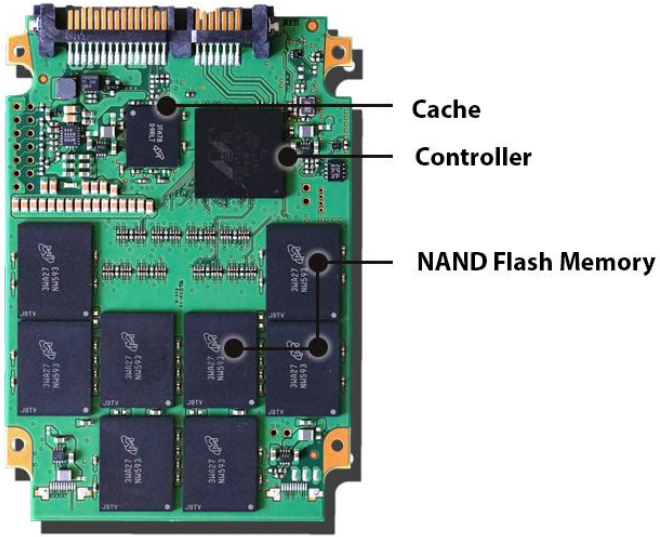
Physical storage, magnetic platters, and solid-state cells

HDD
3.5″



Shock resistant up to 55g (operating)
Shock resistant up to 350g (non-operating)

SSD
2.5″



Shock resistant up to 1500g
(operating and non-operating)

The Big Picture: Why Storage Matters

SECTION 1 · STORAGE FUNDAMENTALS

RAM (Volatile)

Fast but loses all data when power is cut. Your running programs live here - not your files.

Storage (Persistent)

Retains data without power. Everything you save stays here - OS, files, databases.

Network / Cloud

Remote persistent storage. Same fundamentals apply underneath.

Storage Evolution

Magnetic tape → **HDD** → **SSD (SATA)** → **NVMe SSD**

Each generation brought orders-of-magnitude improvements in speed and density - but the fundamental job remains: store bits reliably and retrieve them quickly.

HDD: Mechanical Architecture

SECTION 1 · HDD

- Platters - spinning magnetic disks (5400 / 7200 / 10 000 RPM)
- Read/write heads - float nanometres above the platter surface on an air cushion
- Actuator arm - positions heads radially across tracks
- Tracks - concentric rings on each platter surface
- Sectors - smallest addressable unit (traditionally 512 B, now 4 K)
- Cylinders - same track on all platter surfaces stacked vertically

- Seek time - time to move head to correct track (~3–10 ms)
- Rotational latency - waiting for sector to rotate under head (~4 ms avg at 7200 RPM)
- Transfer rate - sequential reads: ~150–200 MB/s
- Random 4K IOPS - only ~100–200 (catastrophically slow)
- Noise, vibration, shock sensitivity - physical medium = physical fragility

HDD: How Data Is Written & Why Seek Time Kills Performance

SECTION 1 · HDD

Magnetic Encoding

Each bit is stored as a tiny magnetic domain oriented North or South. Changing orientation requires passing the write head over the exact spot and pulsing an electromagnetic field.

Sequential vs Random I/O

Sequential reads are fast - the head barely moves and data streams off the platters. Random reads require a full arm seek + rotational wait for EVERY block. Databases, OS metadata, and small files are all random.

Fragmentation

When a file's blocks scatter across the disk, each fragment requires a separate seek. Over time an HDD can spend more time seeking than transferring. Defragmentation physically consolidates blocks - meaningless on SSDs.

Physical Vulnerabilities

HDDs fail from: bearing wear (gradual), head crashes (sudden), vibration interference, thermal expansion. Always assume an HDD will eventually fail - because it will.

SSD: Solid-State Architecture

SECTION 1 · SSD

Cell Types - the fundamental speed/density/endurance trade-off

SLC	MLC	TLC	QLC
1 bit/cell Fastest Highest endurance 100K+ P/E cycles Most expensive	2 bits/cell Fast Good endurance ~10K P/E cycles Prosumer	3 bits/cell Moderate speed ~3K P/E cycles Consumer standard	4 bits/cell Slowest writes ~1K P/E cycles Budget / read-heavy

Key Components

NAND Flash dies → organized into blocks → pages (smallest writable unit, typically 4–16 KB)
Controller → the CPU of the SSD, runs FTL firmware
DRAM (optional) → caches the mapping table; DRAM-less SSDs are slower and less consistent

SSD: How Data Is Written - and Why It's Complicated

SECTION 1 · SSD

The Write-Erase Asymmetry

You can write individual PAGES (4–16 KB), but to erase you must erase an entire BLOCK (256–512 pages = up to 8 MB). You cannot overwrite a used page - it must be erased first.

Write Amplification

Writing 4 KB of user data may trigger reading, modifying, erasing, and rewriting an entire 4 MB block. The ratio of physical writes to logical writes is the Write Amplification Factor (WAF). High WAF → faster cell wear.

P/E Cycles & Wear

Every Program/Erase cycle slightly degrades the cell's oxide layer. When a cell can no longer reliably store charge, it's marked bad. Consumer TLC SSDs: ~1000–3000 cycles per block.

Over-Provisioning

SSDs reserve extra NAND (7–28% on consumer drives) that the OS never sees. This spare area absorbs writes, enables background garbage collection, and replaces worn-out blocks transparently.

HDD vs SSD - reading 4Kb : Performance Comparison

SECTION 1 · COMPARISON

Metric	HDD (7200 RPM)	SATA SSD	NVMe SSD
Sequential Read	~150–200 MB/s	~550 MB/s	3–7 GB/s
Sequential Write	~120–150 MB/s	~520 MB/s	2–6 GB/s
Random 4K Read	~0.5–1 MB/s	~50 MB/s	300–700 MB/s
Latency	5–10 ms	~0.1 ms	~0.02–0.05 ms
Random IOPS	~100–200	~90,000	up to 1,000,000
Noise / Vibration	Yes	None	None
Shock resistance	Low	High	High

Caching & Data Management: HDD vs SSD

SECTION 1 · CACHE & FTL

HDD Cache (DRAM buffer on PCB)

8–256 MB buffer caches recently read/written sectors. Write-back mode: drive reports 'done' before data is on platter - faster, but power loss before flush = data loss. Write-through: safer, slower.

SSD DRAM Cache

Caches the Flash Translation Layer (FTL) mapping table. DRAM-less SSDs must read the map from flash for every random access - 2–4× slower under random load. Capacitor-protected DRAM = power-loss safe.

SLC Cache (Pseudo-SLC)

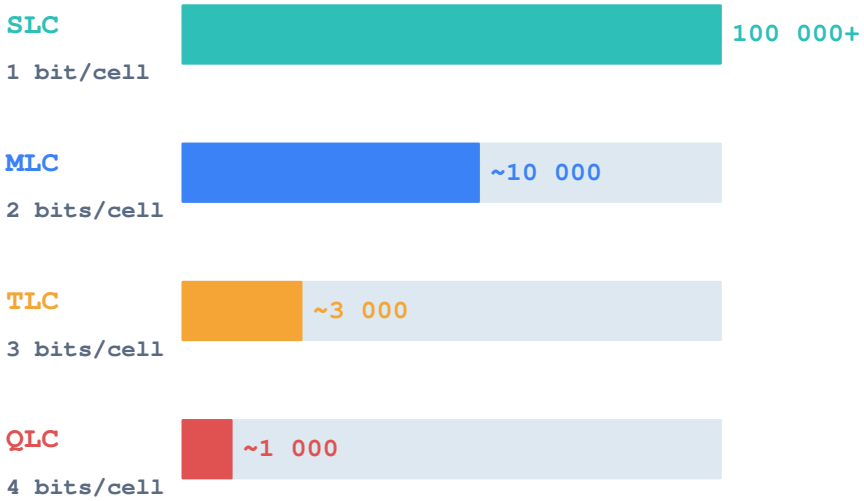
Consumer TLC/QLC drives emulate fast SLC on a portion of NAND. Writes land here first at SLC speed. Background GC later moves data to TLC cells. When the SLC cache fills, write speed plummets to raw TLC speed (~100 MB/s).

Flash Translation Layer (FTL)

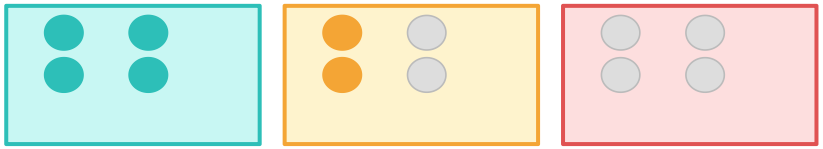
Translates LBAs (what the OS sees) to physical NAND addresses. Enables: wear leveling (spread writes evenly), garbage collection (reclaim stale blocks), bad block management. TRIM command lets the OS tell FTL which blocks are freed - enables proactive GC.

SSD Cell Wear & Flash Translation Layer (FTL)

NAND Cell P/E Cycle Limits



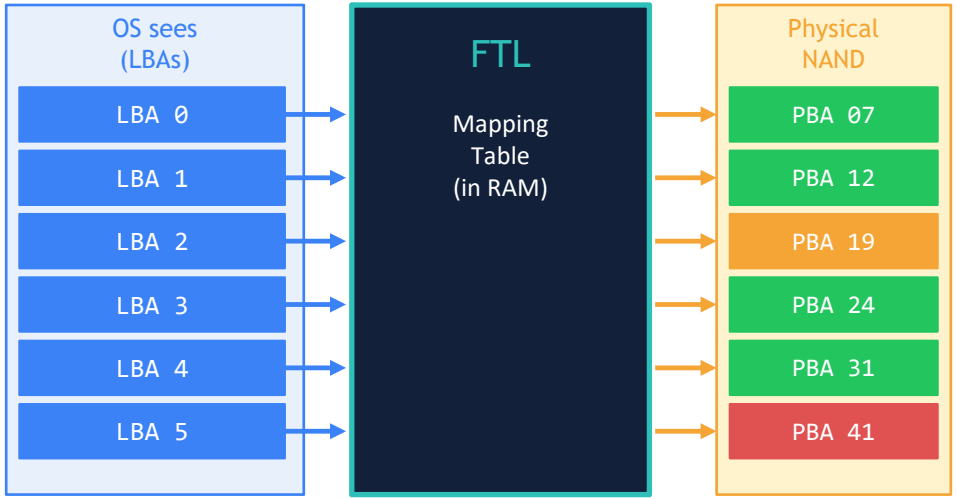
What happens inside the cell:



New cell	Worn cell	Dead cell
0 cycles	~80% P/E used	limit exceeded

Write Amplification Factor (WAF) = physical bytes written ÷ logical bytes written. WAF > 1 always. High WAF → faster cell death.

Flash Translation Layer (FTL) — what it does and why



FTL responsibilities:

Wear levelling
Rotates writes across ALL physical blocks so no single cell receives all writes. Hot LBAs (e.g. filesystem journal) are remapped to fresh cells periodically.

Garbage collection
Pages can only be written once per erase cycle. GC finds blocks with stale pages, copies live pages elsewhere, erases the block, returns it to the free pool.

TRIM / UNMAP
OS tells FTL which LBAs were freed (deleted files). FTL marks those physical pages invalid immediately, so GC can erase them proactively.

Bad block management
When a cell reaches its P/E limit, FTL permanently remaps that physical block to a spare in the over-provisioned reserve area. Transparent to the OS.

Over-provisioning: 7–28% of NAND is hidden from the OS and used exclusively as a GC working area and bad-block replacement pool.

NVMe vs SATA: The Interface Layer

SECTION 1 · INTERFACES

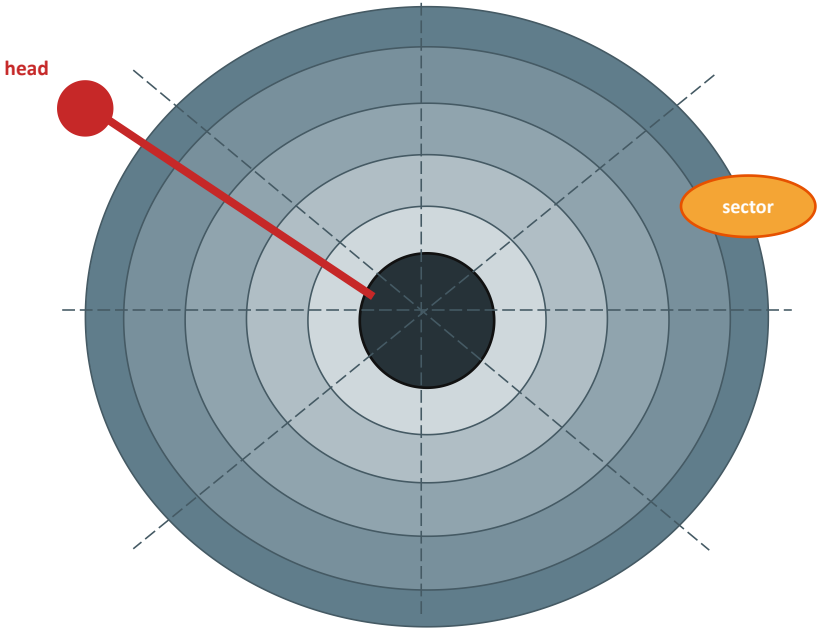
- SATA III - designed in 2003 for spinning disks
- Maximum bandwidth: 600 MB/s (shared with protocol overhead → ~550 MB/s real)
- Command queue: NCQ, max 32 commands at once
- Latency overhead: large - commands must traverse AHCI driver stack
- Connector: physical SATA cable + power - 2 connectors per drive
- Status: legacy - still fine for boot drives if NVMe slot unavailable

- NVMe over PCIe - designed specifically for solid-state storage
- PCIe 4.0 ×4: up to 7 GB/s; PCIe 5.0 ×4: up to 14 GB/s
- Queue depth: 65,535 queues × 65,535 commands each
- Latency overhead: minimal - speaks directly to CPU memory bus
- Connector: M.2 slot or U.2 - single connector, no cables
- NVMe namespaces: one drive can present multiple independent block devices

HDD vs SSD — Physical Data Organisation

SECTION 1 · VISUAL

HDD — Concentric Tracks & Sectors



Track = one concentric ring

Sector = smallest addressable unit (512 B / 4 KB)

Cylinder = same track on all platters (3D)

SSD — NAND Flash: Blocks → Pages (no rotation)

⚠ **Block 2 (red border): has a stale page → entire block must be erased before reuse**

SECTION 2

Partitions vs Filesystems

What they are, how they differ, and how they interact

What Is a Partition?

SECTION 2 · PARTITIONS

Core Definition

A partition is a contiguous range of Logical Block Addresses (LBAs) on a storage device, defined by a start address and an end address. The device itself has no awareness of what is stored inside a partition - it is simply a defined region of blocks.

Analogy: Partitions are like rooms in a building

The building (disk) doesn't care what furniture (data) is inside each room (partition). The building only knows where each room starts and ends.

What the partition table records:

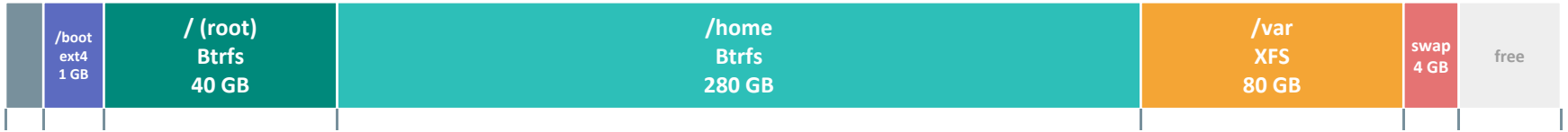
Start LBA - first addressable block belonging to this partition **End LBA** - last addressable block **Partition type GUID** - hint about intended content (Linux data, EFI System, swap, etc.) **Flags / attributes** - bootable, read-only, required, etc.

From Raw Disk to Partitions to Filesystems

SECTION 2 · VISUAL

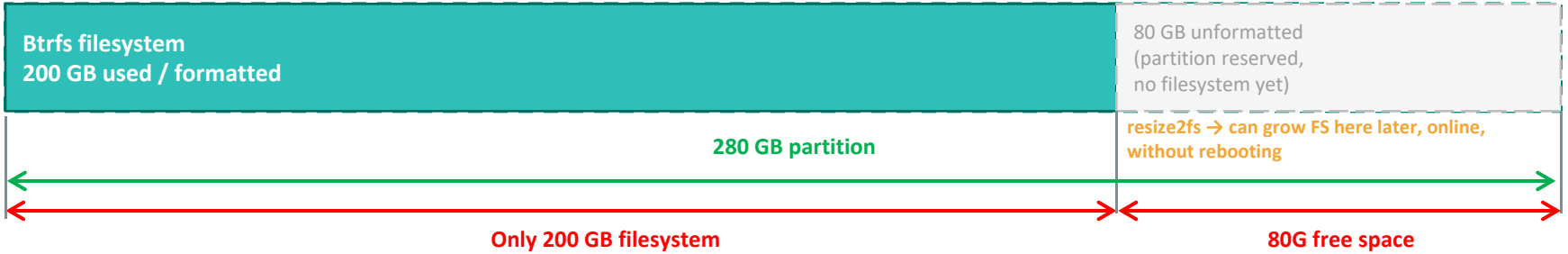
1 Raw disk — one uniform address space (LBA 0 … LBA N)

2 After partitioning — GPT records start/end LBA for each partition



GPT: each coloured region = {start_lba, end_lba, type_guid, name}

3 Filesystem inside /home partition — does NOT have to fill the whole partition



What Is a Partition Table?

A partition table is a small data structure stored at the very beginning of a disk that tells the OS how the disk is divided into regions (partitions) — where each one starts, where it ends, and what type it is.

Disk without a partition table:

```
raw, unpartitioned space — LBA 0 ..... LBA N
```

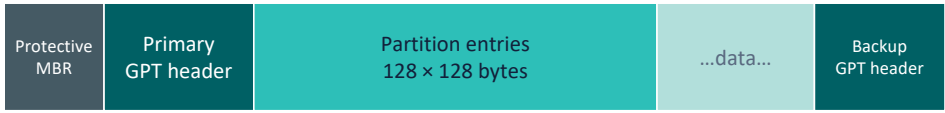
Same disk after writing a GPT partition table:

MBR (Master Boot Record) — 1983

← 512 bytes total →

- ✗ Max 4 primary partitions (or 3 + 1 extended workaround)
- ✗ Max disk size: 2 TB (32-bit LBA addressing)
- ✗ Single copy — corruption of sector 0 = unbootable disk
- ✓ Compatible with all BIOS hardware since 1983

GPT (GUID Partition Table) — 2006 / UEFI



← primary (start of disk) ·····backup at end of disk →

- ✓ Up to 128 partitions (no extended partition hack needed)
- ✓ Max disk size: 9.4 ZB (64-bit LBA addressing)
- ✓ Redundant headers + CRC32 checksums — self-healing
- ✓ Each partition has a 128-bit GUID — globally unique
- ✓ Required for UEFI boot and disks > 2 TB

⚠ The partition table does NOT store data — it only stores geometry (start LBA, end LBA, type GUID, name). The filesystem lives INSIDE the partition. Use GPT for everything new.

Partition Tables: MBR vs GPT

SECTION 2 · PARTITION TABLES

	MBR	GPT
Introduced	1983	2006 (UEFI spec)
Max partitions	4 primary (or 3+1 extended workaround)	128 (Linux: effectively unlimited)
Max disk size	2 TB	9.4 ZB (zettabytes)
Redundancy	Single copy - corruption = unbootable	Primary + backup header at end of disk
Boot standard	BIOS (legacy)	UEFI (required for >2TB disks)
Integrity check	None	CRC32 checksum on header and table
Partition IDs	1-byte type code	128-bit GUID - globally unique
Use today	Old hardware only	Always use GPT on modern systems

GPT Partition Table — Example Register for a 500 GB SSD

#	Mount / Name	Type GUID	Partition GUID (unique)	Start LBA	End LBA	Size
—	GPT Header	— *(metadata, not a partition)* —	—	0	33	34
1	/boot/efi	C12A7328-F81F-11D2-BA4B-00A0C93EC93B (EFI System)	3E6D4B27-A15C-4D2E-9F3A-82B1C4D05E71	2,048	1,050,623	512 MB
2	/boot	0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)	A1B2C3D4-E5F6-7890-ABCD-EF1234567890	1,050,624	3,147,775	1 GB
3	/ (root)	0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)	B2C3D4E5-F6A7-8901-BCDE-F12345678901	3,147,776	87,033,855	40 GB
4	/home	0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)	C3D4E5F6-A7B8-9012-CDEF-123456789012	87,033,856	673,235,967	280 GB
5	/var	0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)	D4E5F6A7-B8C9-0123-DEF0-234567890123	673,235,968	841,008,127	80 GB
6	swap	0657FD6D-A4AB-43C4-84E5-0933C84B4F4F (Linux swap)	E5F6A7B8-C9D0-1234-EF01-345678901234	841,008,128	857,785,343	8 GB
7	/tmp	0FC63DAF-8483-4772-8E79-3D69D8477DE4 (Linux filesystem)	F6A7B8C9-D0E1-2345-F012-456789012345	857,785,344	878,756,863	10 GB
—	GPT Backup	— *(metadata, not a partition)* —	—	976,773,135	976,773,167	33

Each partition entry is exactly 128 bytes. GPT reserves space for 128 entries = 32 sectors × 512 B. Start/End LBAs use 64-bit addressing (max 9.4 ZB). All LBA numbers above assume 512-byte sectors (sector size × LBA = byte offset).

What Is a Filesystem?

SECTION 2 · FILESYSTEMS

Core Definition

A filesystem is a data structure layered ON TOP of a block device (partition or logical volume) that defines how files and directories are named, organised, stored, and retrieved. The partition just gives raw blocks - the filesystem gives them meaning.

What a filesystem defines:

File naming

Allowed characters, max length, case sensitivity

Directory structure

How parent/child relationships are stored (tree, B-tree, hash table)

Inodes / metadata

Permissions, timestamps, owner, size, pointer to data blocks

Free space tracking

Bitmap, extent tree, or B-tree of free extents

Data block mapping

How an inode maps to the actual data blocks on disk

Crash consistency

Journaling, COW, or nothing - what survives a power cut

The Key Distinction: Partition ≠ Filesystem

SECTION 2 · KEY CONCEPT

Key facts:

- A partition is a block device: /dev/sda2 - raw, unformatted blocks. The OS can read individual blocks, but there are no files yet.
- A filesystem is created on top: mkfs.ext4 /dev/sda2 writes filesystem structures into those raw blocks.
- The filesystem does NOT have to use all of the partition's space. resize2fs can shrink an ext4 filesystem independently of the partition boundary.
- This separation is deliberate: you can shrink the filesystem, then shrink the partition, then grow it again - each step independently controlled.
- Practical use: leave unallocated space at partition end → grow filesystem online later without downtime (especially powerful with LVM).

Two Filesystems on One Partition - and Why

SECTION 2 · ADVANCED CONCEPT

Loop Devices

A regular file within a filesystem can be treated as a block device: losetup /dev/loop0 disk.img → mkfs.ext4 /dev/loop0 → mount /dev/loop0 /mnt. Result: a complete second filesystem living inside a file on the first filesystem.

LVM Thin Provisioning

LVM Logical Volumes decouple filesystems from partitions entirely. Multiple LVs can live within one PV (physical volume on a partition), each with its own filesystem. This is the production-grade way to have multiple FSes on one physical partition.

Container Images

Docker/Podman layers are filesystem images stacked via overlay mounts. Each layer is essentially a filesystem image inside a file.

LUKS Encrypted Vault

LUKS container is a file → opened as loop device → filesystem inside. Entire encrypted FS lives as a single file on the outer FS.

VM Disk Images

.qcow2 / .vmdk files contain complete virtual disks with their own partition tables and filesystems, living inside the host filesystem.

SECTION 3

Filesystem Types

FAT · Journaling · Copy-on-Write · and beyond

The Problem All Filesystems Must Solve

SECTION 3 · CORE PROBLEM

Crash Consistency

A file operation (write, rename, delete) involves multiple disk writes: data blocks, inode update, directory entry, free space bitmap. If power dies between any two of these, the filesystem is in an inconsistent state. How you handle this defines your FS architecture.

Example: appending 1 byte to a file requires updating:

1

Data block

Write the actual new byte to a disk block

2

Inode

Update file size and block pointer in the inode

3

Block bitmap

Mark the new block as allocated in the free space map

Power dies between steps 1 and 2: data is on disk but inode doesn't point to it → lost data. Dies between 2 and 3: inode is updated but bitmap says block is free → block allocated twice (corruption). Different FS architectures handle this differently.

Table-Based Filesystems: FAT32 & exFAT

SECTION 3 · FAT

How FAT Works

The File Allocation Table is a flat linked list of cluster entries. Each cluster entry points to the next cluster of the file, or marks EOF. The directory entry gives the first cluster. Navigation = follow the chain.

The FAT32 Limits

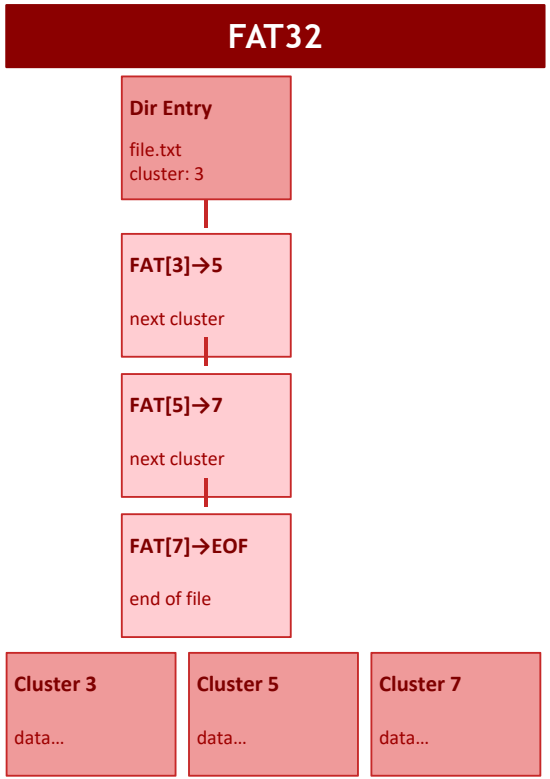
Max file size: 4 GB (32-bit cluster size field). Max volume: 2 TB. No permissions, no symlinks, no hard links. No journaling - power loss mid-write can corrupt the FAT itself, making files unreadable.

exFAT - Microsoft's 'FAT for Flash':

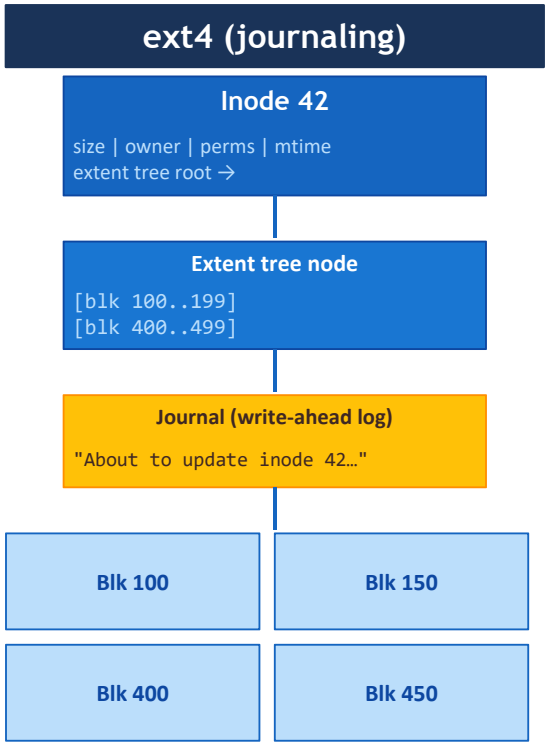
- Removes 4 GB file size limit (64-bit cluster count)
- Improved allocation bitmap - faster free space lookup than FAT32
- Still no journaling, no Unix permissions, no hard links
- Supported natively by Windows, macOS, and Linux (kernel 5.4+)
- Use case: USB drives and SD cards shared between OS - NOT for system partitions

Visual: How FAT, ext4, and Btrfs Organise Data

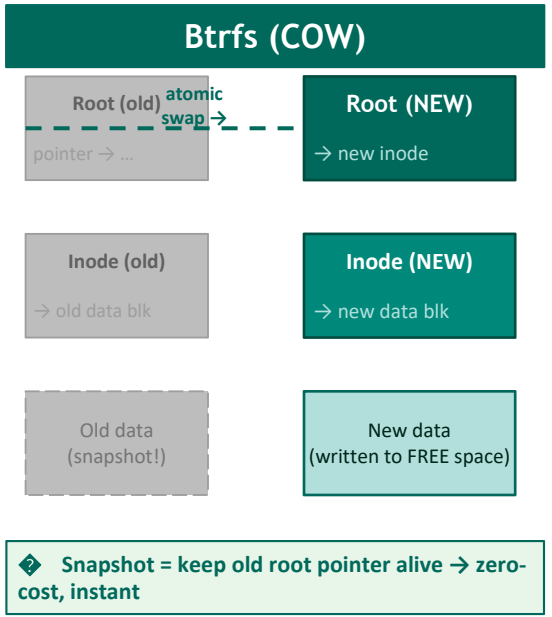
SECTION 3 · VISUAL



Problem: must follow whole chain to find block N → O(N) random seeks



Overwrite = modify in-place + journal commit → safe but double-write overhead



Snapshot = keep old root pointer alive → zero-cost, instant

Never overwrites! New version → free space Old = snapshot / freed later

Journaling Filesystems: ext4, XFS, NTFS

SECTION 3 · JOURNALING

The Journal: Write-Ahead Log

Before making any change, the filesystem writes its intention to a dedicated journal area: 'I am about to update inode 42 and block 1337.' Only after the journal commit does it apply the changes. On crash: replay the journal to reach consistency.

ext4 Journaling Modes:

writeback	ordered (default)	full journal
Only metadata journaled. Data can be written before or after metadata. Fastest, least safe - data may be wrong on crash but FS structure is valid.	Data blocks are written to their final location BEFORE the metadata journal commit. Balance of safety and performance. This is the Linux default.	Both data AND metadata go through the journal - every write is journaled twice. Safest, but ~2× write overhead. Used in high-availability scenarios.

ext4 key features: extents (replace block maps → better large-file performance), delayed allocation, 1 EB max volume, 64-bit block numbers, online defrag.

XFS: high parallelism, excellent for large files and many concurrent writes. B+tree directory indexing - stays fast with millions of files. Online grow (not shrink). Default on RHEL/Rocky.

Copy-on-Write Filesystems: Btrfs, ZFS, APFS

SECTION 3 · COW

Core Principle: Never Overwrite In Place

When a block must be updated, COW filesystems write the NEW version to a free location, update the parent metadata pointer to reference the new block, then release the old block. The old version remains valid until the pointer is atomically swapped.

What COW enables - for free:

Atomic commits

Either the entire transaction lands (new root pointer visible) or nothing does. No journal needed - the pointer swap IS the commit.

Instant snapshots

Just freeze the old root pointer. The snapshot is an immutable view of the filesystem tree at a point in time. Cost: ~0 ms, ~0 bytes initially.

Block-level checksums

Every data and metadata block has a hash (CRC32c or SHA256). Silent corruption - a flipped bit - is detected and (with redundancy) self-healed.

Transparent compression

Compress each block before writing to its COW destination. Works per-file or globally. Algorithms: lz4 (fast), zstd (balanced), zlib (dense).

Deduplication

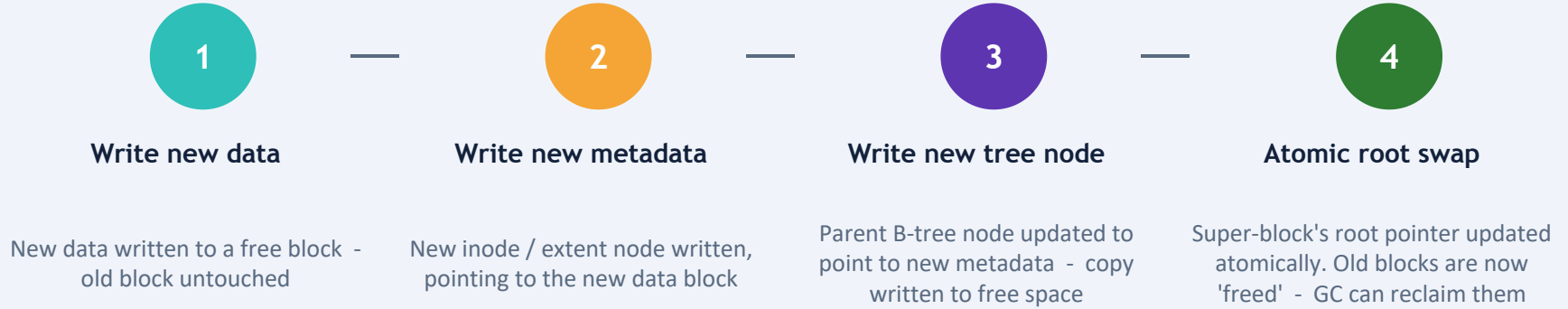
Identical block content → single shared physical block. Multiple files reference the same data without extra storage. Copy-on-write ensures isolation.

Send / Receive

Stream the diff between two snapshots over a pipe - efficient incremental backup without rsync scanning the entire tree.

COW Deep Dive: How a Write Actually Works

SECTION 3 · COW INTERNALS



Why this means snapshots are free:

- Before step 4 (root swap), the old root pointer still points to a valid, consistent tree. A snapshot just keeps this old pointer alive - preventing GC from reclaiming those blocks.
- The snapshot is an immutable read-only view. Initial disk cost = zero bytes. Space is consumed only as the live filesystem diverges from the snapshot.
- On Btrfs: 'btrfs subvolume snapshot / /.snapshots/snap1' - instant. On ZFS: 'zfs snapshot pool/dataset@snap1' - instant.
- Snapshots enable: system rollback before package updates, point-in-time backups, testing destructive operations safely.

COW Trade-offs: The Cost of All That Power

SECTION 3 · COW TRADE-OFFS

Write Amplification

Even small writes trigger a cascade of metadata tree updates up to the root. Writing 4 KB of data may cause 5–6 additional metadata block writes. On SSDs this is usually acceptable; on HDDs under random workloads it's punishing.

Fragmentation on HDDs

Because data is never written in-place, files fragment continuously over time. On SSDs fragmentation is irrelevant (no seek penalty). On spinning disks COW + heavy write workloads eventually degrades read performance significantly.

✓ Ideal for SSDs

No seek penalty for fragmentation. Checksums catch silent corruption (bit rot) - SSDs and modern disks can silently flip bits. Snapshots are invaluable for SSD-based systems.

⚠ Caution for Databases

Databases do fine-grained, high-frequency updates to B-tree pages. COW + database = massive write amplification. Use nodatacow attribute or a separate non-COW partition for database data directories.

Btrfs: Linux's Native COW Filesystem

SECTION 3 · BTRFS

- Subvolumes - independent filesystem trees within one Btrfs volume. Each has its own snapshot history, can be mounted separately, and can be sent/received independently.
- Snapshots - instant, space-efficient. snapper integrates with package managers: automatic snapshot before/after every dnf/apt operation → instant rollback.
- Online resize - grow or shrink while mounted (shrink limited by actual data usage).
- RAID 0/1/10 - built into the filesystem; mdadm not needed. RAID 5/6: still has known reliability issues - avoid in production.
- Send / Receive - 'btrfs send | btrfs receive' streams the diff between two snapshots. Efficient incremental backup over SSH.
- Transparent compression - per-file or global; algorithms: lz4 (fastest), zstd (best ratio/speed balance), zlib (highest compression). Typical savings: 30–60% on code, docs, logs.
- Checksums - CRC32c on all data and metadata blocks by default. SHA256 available. Silent corruption detected immediately on read.
- Deduplication - inline (slow, high RAM) or offline via duperemove/bees. Best for VM images, backups, containers.
- Used by: Fedora (default / since F33), openSUSE (default), Steam Deck OS, Synology NAS DSM.
- Current status (2026): stable for single-disk and RAID 1/10. Avoid RAID 5/6 for critical data.

ZFS: The Gold Standard of Storage Integrity

SECTION 3 · ZFS

Architecture: No Partitions Needed

ZFS combines volume manager + filesystem. Physical disks → zpool → datasets (filesystems) + zvols (block devices). ZFS bypasses the partition layer entirely. One zpool can span multiple disks with built-in RAID.

End-to-End Integrity

Every block carries a 256-bit checksum (SHA256/SHA512). On read, checksum is verified. On mismatch with RAID redundancy: ZFS reads the redundant copy, fixes the corrupt block, and logs the event - fully automatic self-healing.

Caching Architecture

ARC (Adaptive Replacement Cache): RAM-based read cache, far smarter than page cache. L2ARC: SSD-based extension of ARC. ZIL (ZFS Intent Log): write-ahead log for sync writes, can be accelerated with a dedicated NVMe device (SLOG).

Linux Trade-offs

OpenZFS is stable on Linux (6.x kernel). CDDL license is incompatible with GPL - ZFS cannot be merged into the kernel; installed as a DKMS module. RAM hungry: 1 GB RAM per TB of storage is a rough guideline for ARC. Not default on any major distro.

Special Filesystems: tmpfs, proc, sysfs, and the VFS

SECTION 3 · VIRTUAL FS

tmpfs - RAM-Backed Filesystem

Looks like a regular directory, behaves like a regular filesystem, but lives entirely in RAM (and swap). Lost on reboot. Used for /tmp, /run, /dev/shm. Mount: 'mount -t tmpfs -o size=4G tmpfs /tmp'. Lightning fast - no disk I/O.

proc & sysfs - Kernel as Files

/proc exposes running processes and kernel parameters as files. /proc/cpuinfo, /proc/meminfo, /proc/net/ - all virtual files generated on-demand by the kernel. /sys exposes hardware and driver configuration. Reading /proc/cpuinfo doesn't read a disk - the kernel generates it on the fly.

devtmpfs & udev

/dev contains device nodes - special files that represent hardware. devtmpfs populates /dev automatically as devices appear. 'ls -la /dev/sda' shows a block device node; reading/writing it reads/writes the raw disk.

VFS - Virtual Filesystem Switch

The kernel's VFS layer is the abstraction that makes all of this work. To the OS, everything is a file. The VFS routes read/write calls to the appropriate driver: ext4, Btrfs, tmpfs, proc - the application doesn't know or care which.

Visual: Filesystem Behaviour on HDD vs SSD — and Why It Matters

SECTION 3 · VISUAL

Filesystem	HDD (spinning)	SATA/NVMe SSD	NVMe SSD + heavy random write
FAT32 exFAT	✓ OK Small files/USB fine. Fragmentation over time. No crash safety.	✓ OK Fine for removable media. TRIM not supported.	⚠ Poor No TRIM. Wastes SSD capability. Use only for /boot/efi.
ext4 (journaling)	✓ Good Excellent. Sequential I/O, minimal fragmentation.	✓ Good Works well. Use noatime + discard=async.	✓ Good Solid and predictable. Best for /boot and databases.
XFS (journaling)	✓ Good Excellent for large files & concurrent writes. Cannot shrink.	✓ Great Ideal for /var. High IOPS, parallel metadata ops.	✓ Great Best filesystem for /var on NVMe. High write parallelism.
Btrfs (COW)	⚠ Caution COW → fragmentation grows. Slow random reads over time. Avoid for /var.	✓ Good Checksums catch bit rot. Snapshots + compression. Ideal /home.	✓ Great Full benefit: fast GC, low fragmentation penalty, snapshots.
ZFS (COW + pool)	✓ Great Designed for large HDD arrays. Self-healing with redundancy.	✓ Great ARC cache shines. L2ARC on SSD accelerates read cache.	⚠ RAM! Needs lots of RAM for ARC (1 GB/TB guideline). Otherwise excellent.

⚠ Key rule: Btrfs/ZFS COW on /var or DB data dirs = write amplification. Use ext4 or XFS there, and set nodatacow on any Btrfs mount used by databases.

See the filesystems:

https://oos.wimic.agh.edu.pl/oos/3/

SECTION 4

Partitioning Linux Disks

Strategy, tools, mount options, and LVM

Linux Directory Structure: I/O Profiles

SECTION 4 · DIRECTORY ANALYSIS

Directory	Content	Write Frequency	File Size / Count	Notes
/	OS core, init, config	Very Low	Mixed	Stable after install; updates only
/home	User files, docs, media	Low–Med	Large files	Rare writes, large sequential reads
/var	Logs, databases, caches, mail	Very High	Many small	THE hot-write directory
/var/log	System logs	Continuous	Small, appended	Append-only, high volume
/var/lib	Database files (postgres, etc.)	Very High	Medium	Random reads/writes - DB I/O
/tmp	Temporary build files	High	Varies	tmpfs ideal - evaporates on reboot
/boot	Kernel, initrd, GRUB	Very Low	Small	Writes only on kernel updates
/boot/efi	EFI system partition	Rarely	Small	Must be FAT32; UEFI reads it
/usr	Programs, libraries, shared	Very Low	Many medium	Read-only in practice; updates only
/opt	Third-party software	Low	Mixed	Similar to /usr

Why Separate Partitions?

SECTION 4 · RATIONALE

Isolation

A full /var (logs, caches) does NOT crash the OS. / still has free space. Without isolation, a rogue log file can fill the root filesystem and make the system unbootable.

Different Mount Options

/home → noexec, nosuid (users can't execute binaries or suid tricks). /tmp → noexec, nosuid, nodev. These security options can't be mixed on one partition.

Different Filesystems

/var benefits from XFS (high IOPS, many small files). /home benefits from Btrfs (snapshots, compression). You can't have two FSes on one partition normally.

Different Backup Strategies

/home: backed up nightly. /var/cache, /tmp: never backed up. /boot: backed up before kernel updates. Separate partitions → separate backup policies.

Quotas & Accounting

Per-partition disk quotas for multi-user systems. Easier capacity planning - you know exactly how much space each subsystem uses.

Independent Resize & Replace

Move /home to a new, larger disk without touching /. Add a faster NVMe just for /var. Replace / with a fresh install keeping /home intact.

Choosing the Right Filesystem for Each Mount

SECTION 4 · FS SELECTION

/boot/efi	**FAT32**	UEFI specification requirement. Must be FAT32 - no exceptions.	100–512 MB
/boot	**ext4**	GRUB must read it at boot - universal support. No COW complexity needed.	512 MB–1 GB
/ (root)	**Btrfs or ext4**	Btrfs: system snapshot rollback via snapper. ext4: maximum stability and simplicity.	20–50 GB
/home	**Btrfs**	Large files → compression saves 30–50%. Per-user subvolumes → per-user snapshots. noatime eliminates read-triggered writes.	Remainder
/var	**XFS or ext4**	Thousands of small files, high random write frequency. XFS excels here. AVOID Btrfs COW - write amplification for tiny random writes is painful.	20–100 GB
/tmp	**tmpfs**	RAM-backed - zero disk I/O. Auto-cleaned on reboot. Size-limited to prevent RAM exhaustion.	size=4G (RAM)

Practical Partitioning Example: 500 GB SSD

SECTION 4 · PRACTICAL EXAMPLE

Option A: Direct Partitions

/dev/sda1 512 MB	**FAT32**	/boot/efi
/dev/sda2 1 GB	**ext4**	/boot
/dev/sda3 40 GB	**Btrfs**	/
/dev/sda4 280 GB	**Btrfs**	/home
/dev/sda5 80 GB	**XFS**	/var
/dev/sda6 rest	**ext4**	/data

Option B: With LVM (recommended)

/dev/sda1	512 MB	**FAT32**	/boot/efi
/dev/sda2	1 GB	**ext4**	/boot
/dev/sda3	rest	**LVM PV**	Volume Group 'vg0'
lv_root	40 GB	**Btrfs**	/
lv_home	280 GB	**Btrfs**	/home
lv_var	80 GB	**XFS**	/var
lv_tmp	10 GB	**ext4**	/tmp (fallback)
(free)	~89 GB	-	Unallocated - grow any LV online later

Mount Options That Matter

SECTION 4 · MOUNT OPTIONS

Option	Effect	Recommended for
noatime	Disables access time (atime) updates on reads. Each 'cat file' no longer triggers a metadata write.	/home, /var, all SSDs
relatime	Updates atime only if it's older than mtime. Safer default - some apps rely on atime ordering.	Default safe option on /
noexec	Prevents execution of binaries from this filesystem. Blocks direct exploit payloads.	/tmp, /var, /home (security hardening)
nosuid	Ignores setuid/setgid bits. Prevents privilege escalation via suid binaries on user-writable mounts.	/tmp, /home
compress=zstd	Btrfs: transparent zstd compression. Excellent ratio+speed balance. Saves 30–60% on text/code.	Btrfs /home, /
nodatacow	Btrfs: disable COW for specific files/directories. Falls back to journaling. Essential for databases.	Btrfs + PostgreSQL/MySQL data dirs
discard=async	SSD TRIM: asynchronously informs SSD of freed blocks. Enables proactive GC. ext4/XFS/Btrfs all support.	All SSDs - use async not sync
barrier=0	Disables write barriers (cache flush ordering). DANGEROUS - only safe with UPS or battery-backed cache.	Never without battery-backed cache

Partitioning & Formatting Tools in Linux

SECTION 4 · TOOLS

- **fdisk** /dev/sda - interactive text UI, now supports GPT. Classic tool, still widely used.
- **gdisk** /dev/sda - GPT-focused successor. Better GPT handling, same workflow as fdisk.
- **parted** /dev/sda - supports MBR and GPT, scripting-friendly (no interactive prompt needed), used by installers.
- **gparted** - GUI frontend to parted. Excellent for visual partition resizing. Run from a live USB.
- **lsblk** - list block devices, their sizes, mount points, and relationships in a tree. First thing to run on a new system.
- **blkid** - show UUID, filesystem type, and label for each partition. Essential for writing /etc/fstab.
- **mkfs.ext4 /dev/sda2** - format partition as ext4. Options: -L label, -m 1 (reduce reserved blocks).
- **mkfs.xfs /dev/sda5** - format as XFS. Options: -L label, -f (force).
- **mkfs.btrfs /dev/sda3** - format as Btrfs. Options: -L label, -d single (metadata duplication).
- **tune2fs -m 1 /dev/sda2** - reduce ext4 reserved block percentage (default 5% → 1% on large volumes).
- **resize2fs /dev/sda2 80G** - resize ext4 filesystem (partition must be resized first with parted).
- **xfs_growfs /var** - grow XFS filesystem online to fill available space. xfs_admin for tuning.

LVM: Logical Volume Manager - Why It Changes Everything

SECTION 4 · LVM

Physical Volumes (PV)

/dev/sda3, /dev/sdb1, /dev/nvme0n1p3 - actual partitions

Volume Group (VG)

vg0 - pool of all PV space, addressed as one

Logical Volumes (LV)

lv_root, lv_home, lv_var - virtual block devices, arbitrarily sized

Filesystems

ext4, XFS, Btrfs - formatted on top of each LV

Key LVM Capabilities:

- Online resize - grow an LV and filesystem while mounted and in use. No downtime: lvextend -L +50G vg0/lv_var && xfs_growfs /var
- Span multiple disks - add a new disk as a PV, extend the VG, then extend any LV across it. Transparent to the filesystem.
- Snapshots - block-level COW snapshots of any LV, regardless of filesystem type. Used for consistent backups.
- Thin provisioning - allocate more logical space than physical. Overcommit storage; allocate physical only as needed.
- Cache volumes - attach a fast NVMe as an lvmcache to accelerate a slow HDD logical volume automatically.
- Migration - pvmove migrates data between PVs online. Replace a failing disk without downtime.

Alignment, 4K Sectors, and Why They Matter

SECTION 4 · ALIGNMENT

The 512 → 4K Transition

Old disks: 512-byte physical sectors. Modern disks: 4096-byte (4K/4Kn) physical sectors. Many modern drives emulate 512-byte sectors (512e) for compatibility while having 4K physical sectors underneath.

Misalignment Penalty

If a partition starts at a non-4K-aligned offset, every 4K filesystem write spans two physical 4K sectors: Read-Modify-Write old sector 1 + Read-Modify-Write old sector 2. That's 4 physical I/Os instead of 1. Performance halved, wear doubled on SSDs.

Modern Tool Defaults (Safe)

fdisk, gdisk, and parted all default to 1 MiB alignment for new partitions. 1 MiB covers: all 4K sector sizes, all SSD erase block sizes (128 KB–4 MB). You rarely need to think about this - but understand why.

Checking Alignment

```
parted /dev/sda align-check optimal 1
```

Checks partition 1 alignment. Also check cat /sys/block/sda/queue/physical_block_size and /optimal_io_size for the device's actual physical sector size.

SECTION 5

Putting It All Together

Decision frameworks, summary tables, and next steps

Decision Framework: Choosing Your Storage Stack

SSD or HDD?

SSD → Enable TRIM (discard=async), prefer COW FS or XFS/ext4 with noatime. HDD → Prefer journaling FS, minimise fragmentation, separate /home.

What is the workload?

Large sequential files → XFS or Btrfs+compress. Many small random files → XFS or ext4. System snapshots → Btrfs+snapper. Databases → ext4/XFS + nodatacow on Btrfs.

How many disks?

Single → Careful partition sizing + LVM for flexibility. Multiple → Btrfs RAID 1/10, or ZFS for integrity. mdadm + LVM for maximum control.

Need snapshots for rollback?

Yes → Btrfs with snapper (desktop/workstation). ZFS with zfs-auto-snapshot (server). Both integrate with package managers for automatic pre-update snapshots.

Need maximum flexibility?

Always use LVM between partitions and filesystems. Leave 10–20% of VG unallocated. Use thin provisioning for development/test VMs.

Need data integrity above all?

ZFS: end-to-end checksums, self-healing, ARC cache, ZIL for sync writes. Requires more RAM (minimum 8 GB, ideally 32+ GB for large pools).

Quick Reference: Recommended Configuration

Mount Point	Filesystem	Key Mount Options	Reason
/boot or /boot/efi	FAT32	`defaults`	UEFI requirement
/boot (for BIOS)	ext4	`defaults`	GRUB compatibility
/	Btrfs or ext4	`compress=zstd, discard=async`	Snapshots (Btrfs) or stability (ext4)
/home	Btrfs	`noatime, compress=zstd`	Large files, compression, per-user snapshots
/var	XFS or ext4	`noatime, discard=async`	High-freq small writes; avoid COW
/var/lib/postgresql	ext4 or XFS	`noatime, nodatacow (Btrfs)`	DB random I/O - COW is harmful
/tmp	tmpfs	`size=4G, noexec, nosuid`	RAM speed, auto-clean on reboot
/	LVM over all	-	Online resize, flexibility, snapshots

SECTION 6

Extra info

Case Sensitivity — It's the Filesystem, Not the Kernel

SECTION · FILESYSTEM

How a filename lookup works

The kernel passes the filename string to the filesystem driver UNCHANGED. The driver alone decides whether 'File.txt' and 'file.txt' are the same path or two different ones. The kernel has no opinion on the matter — it is completely encoding- and case-agnostic.

Filesystem	Case-sensitive?	Notes
ext4	✓ **Yes (default)**	Can be made case-insensitive per-directory (kernel 5.2+, casefold option)
XFS	✓ **Yes**	Always case-sensitive, no toggle
Btrfs	✓ **Yes (default)**	Per-directory case-insensitive flag (+F) available
ZFS	✓ **Yes (default)**	casesensitivity=insensitive per dataset
FAT32 / exFAT	✗ **No**	Case-preserving but case-insensitive — standard USB behaviour
NTFS	✗ **No (default)**	Case-insensitive by default; case-sensitive mode available in Windows 10+
tmpfs	✓ **Yes**	RAM filesystem — always case-sensitive

How Many Different Files Can You Have Named 'file.txt'?

SECTION · CASE SENSITIVITY DEMO

On a case-sensitive filesystem (ext4 / XFS / Btrfs / ZFS) the kernel treats ALL of these as distinct files:

file.txt	File.txt	fIle.txt	FIle.txt
fiLe.txt	FiLe.txt	fILe.txt	FILe.txt
filE.txt	FilE.txt	fIlE.txt	FIlE.txt
fiLE.txt	FiLE.txt	fILE.txt	FILE.txt

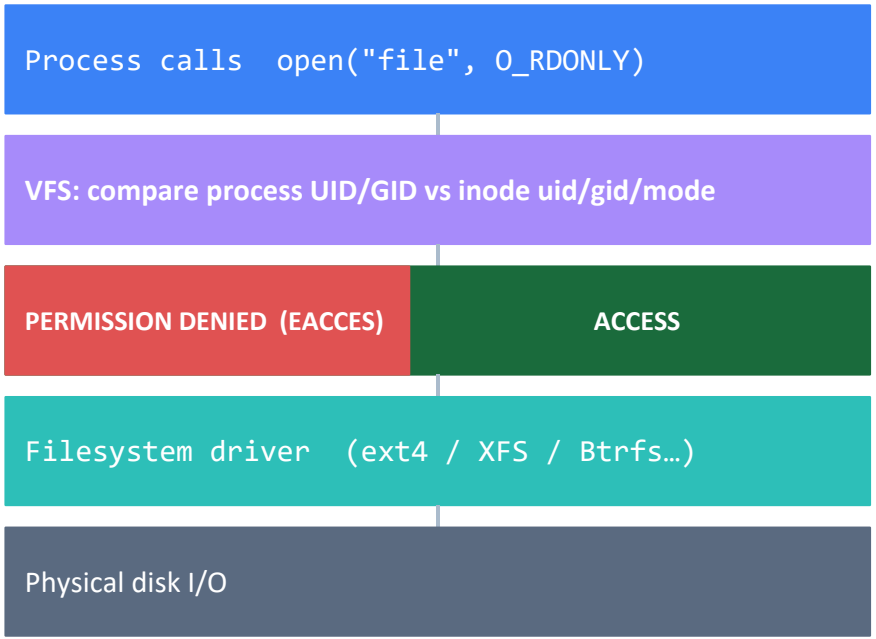
= 16 completely different files in the same directory

On FAT32 or NTFS (case-insensitive) all 16 names point to the SAME file

```
# Try this on Linux (ext4) — all 16 commands create DIFFERENT files:
touch file.txt File.txt fIle.txt fiLe.txt filE.txt FIle.txt FiLe.txt FILe.txt
touch FILE.txt fILE.txt fiLE.txt filE.txt FIlE.txt FiLE.txt fIlE.txt FILE.txt
ls -1 | wc -l   # → 16
```

Users, Groups & Permissions — The Kernel's Job (VFS Layer)

SECTION · KERNEL / VFS



Where metadata is STORED vs ENFORCED

Stored: in the filesystem inode (uid, gid, mode fields on disk).
Enforced: by the kernel VFS — before the request ever reaches the driver.
The filesystem driver never sees a request it isn't allowed to serve.

root always bypasses permission checks

UID 0 (root) skips the rwx check entirely inside the kernel. This is hardcoded in the VFS layer — no filesystem can override it.

FAT32 / exFAT — no inode permissions

FAT has no uid/gid/mode fields on disk. The kernel's vfat driver synthesises fake permissions from mount options:
uid=1000, gid=1000, fmask=133
These exist in RAM only — never written to disk. The enforcement mechanism is identical; only the source of the metadata differs.

File Encoding — What Is Actually Stored in a Filename?

SECTION · ENCODING

Linux kernel rule: filenames are just bytes

The kernel stores filenames as raw byte sequences. It has no idea what encoding they use. The only forbidden bytes are 0x00 (NUL — string terminator) and 0x2F (slash — path separator). Everything else is legal.

What encoding is actually used?

By convention: UTF-8 everywhere on modern Linux (set via locale, e.g. LANG=en_US.UTF-8). But this is a userspace convention — the kernel never validates it. You can create a file whose name is valid ISO-8859-2 and the kernel won't complain.

Encoding	Bytes for 'ą'	Bytes for '€'	Used where
ASCII	N/A (not representable)	N/A	Legacy English-only systems
ISO-8859-2	0xB1 (1 byte)	N/A	Old Polish/Central European
UTF-8	0xC4 0x85 (2 bytes)	0xE2 0x82 0xAC (3 bytes)	All modern Linux/macOS/Web
UTF-16	0x0105 (2 bytes)	0x20AC (2 bytes)	Windows NTFS filenames internally

Why this causes real bugs

A file created on a Polish Windows (UTF-16 NTFS) and copied to Linux may display as garbled characters if the terminal locale doesn't match. The bytes are the same — the interpretation differs.

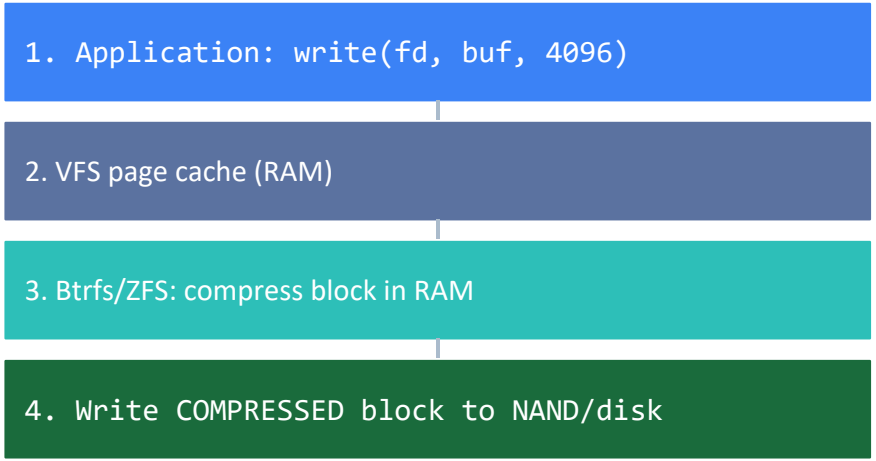
NTFS is special

NTFS stores filenames in UTF-16LE internally. The Linux ntfs3 driver automatically translates to/from UTF-8 when you mount an NTFS partition — translation is transparent to userspace.

Btrfs & ZFS — Transparent Compression and Why They Need Lots of RAM

SECTION · COW FILESYSTEMS

How transparent compression works:



Compression algorithms available:

lz4	zstd	zlib
Fastest · Low ratio Best for /var, databases	Fast · Good ratio Default choice for /home	Slow · Best ratio Archives, static data

Why do Btrfs & ZFS need lots of RAM?

B-tree metadata in RAM (both)

The entire filesystem B-tree (inode table, extent map, checksum tree) must be cached in RAM for fast lookups. The bigger the filesystem, the larger this tree. Evicting it causes every metadata operation to hit the disk.

ARC — Adaptive Replacement Cache (ZFS)

ZFS replaces the OS page cache with its own smarter cache called ARC. By default it can use up to 50% of available RAM. Guideline: 1 GB RAM per 1 TB of storage for comfortable ARC operation. On a 32 TB NAS you want at least 32 GB RAM.

COW write buffers (both)

Every write first lands in RAM (page cache), gets compressed, checksummed, and only then flushed to a new free location. Under heavy write load, dirty pages accumulate in RAM before each sync.

Deduplication tables (ZFS)

Optional block-level dedup requires a hash table in RAM — one entry per block. A 1 TB pool with 4 KB blocks = 256 million entries. At ~320 bytes each = ~80 GB RAM. This is why ZFS dedup is disabled by default.

Questions?

Further Reading & Tools

Arch Wiki
Partitioning, Btrfs, LVM, XFS - the most comprehensive Linux storage documentation available. wiki.archlinux.org

OpenZFS Docs
openzfs.github.io - authoritative ZFS documentation, tuning guides, and hardware recommendations

Btrfs Wiki
btrfs.readthedocs.io - official Btrfs documentation including gotchas and RAID status

fio
Flexible I/O Tester - benchmark any filesystem/partition configuration: fio --name=randread --ioengine=libaio --rw=randread --bs=4k

iostat / iotop
iostat -x 1 - per-device I/O stats. iotop - per-process I/O (like top, for disk)

blktrace
Low-level block I/O tracing - see exactly what operations reach the disk. Combined with blkparse and btt for analysis.