Linux Filesystems

A Complete 11-Chapter Course on ext4, Btrfs, ZFS, LVM & RAID

Topics covered:

ext4, Btrfs & ZFS compared · LVM & live resizing

LVM snapshots for live backups · RAID levels & mdadm

Traditional layered stacks vs. integrated filesystems

Capstone: a real RAID 10 + LVM + ext4 stack, built and broken on purpose

Exercises: 33 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · resolves grub1-8's own deferred LVM promise

Table of Contents

01 What a Filesystem Actually Does

02 ext4 — The Traditional Default

03 Btrfs — Copy-on-Write & Snapshots

04 ZFS — A Different Philosophy Entirely

05 LVM — Logical Volume Manager

06 LVM Snapshots & Practical Use Cases

07 RAID Concepts & Levels

08 Software RAID with mdadm

09 Combining Layers — Traditional Stack vs. Integrated Filesystems

10 Choosing the Right Filesystem/Stack for a Real Scenario

11 Capstone: Building a Real Storage Stack

CHAPTER 1 OF 11

What a Filesystem Actually Does

Linux Filesystems

Chapter 1 · What a Filesystem Actually Does

`grub1-5` covered everything that happens *before* a filesystem exists — firmware, bootloaders, partition tables. This course picks up exactly where that left off, and resolves something `linux1`'s own installer did silently: choosing partition sizes and confirming formatting, running the real operations this chapter finally explains.

The Layer Above the Partition Table

A partition table (MBR or GPT, per `grub1-5`) divides a disk into partitions — but a partition table entry says nothing about what *kind* of data actually lives inside a given partition. That's a filesystem's own job: the real structure organizing raw bytes into files and directories, imposed within a partition. This course starts exactly where `grub1`'s own scope ended.

The VFS — One Interface, Many Filesystems

The **Virtual Filesystem Switch** (VFS) is a kernel abstraction layer that lets every program use the same system calls — `open`, `read`, `write` — regardless of which actual filesystem a given file happens to live on. This is exactly why plugging in a USB stick formatted with a completely different filesystem just works with the same `ls`/`cp` commands — the VFS translates underneath, invisibly.

Block Devices vs. Filesystems — A Real, Important Distinction

A **block device** (`/dev/sda1`) is a raw, undifferentiated sequence of blocks, addressable only by block number — genuinely no structure of its own. A **filesystem** is the structure imposed *on top of* that raw block device — inodes, directories, file data, metadata — turning "a big blob of bytes" into files and folders you can actually navigate.

```
mkfs.ext4 /dev/sdb1
```

`mkfs` (make filesystem) is the literal act of imposing that structure for the very first time — exactly the operation `linux1`'s own installer ran silently, on your behalf, without ever explaining it.

Inodes, Superblocks, and a Preview of Journaling

- **Superblock** — metadata about the filesystem itself (size, block size, type, last-mounted time), stored at a known location, with backup copies elsewhere on disk for resilience
- **Inode** — metadata about one file (permissions, owner, size, timestamps, and pointers to its actual data blocks). Critically, the **filename itself is not stored in the inode** — a directory is just a mapping of names to inode numbers, a genuinely surprising fact worth sitting with; it's also exactly why a hard link works
- **Journaling** — previewed here, covered in full in `fs1-2`'s own ext4 chapter — keeps the filesystem consistent even if power is lost mid-write

Mounting — Attaching a Filesystem to the Directory Tree

`mount` takes a filesystem living on a specific block device and attaches it at a specific point in the overall directory tree — a **mount point**. Before mounting, that point is just an ordinary, empty directory; after, everything under it transparently belongs to the mounted filesystem.

```
UUID=1234-5678  /home  ext4  defaults  0  2
```

A real `/etc/fstab` line — the persistent configuration for what mounts automatically at boot. Fields, left to right: the device (identified by UUID, more reliable than a device name that can shift between boots), the mount point, the filesystem type, mount options, the dump backup flag, and the `fsck` check order.

Resolving linux1's Own Glossed-Over Formatting Step

During Debian installation, `linux1`'s own installer asked you to choose partition sizes and confirm formatting — silently running `mkfs` and writing a real `fstab` entry on your behalf, a genuine black box at the time. The actual mechanism is now on the table.

	WHAT IT ACTUALLY IS	OPERATIONS THAT MAKE SENSE
Block device	A raw sequence of addressable blocks, no structure	Reading/writing raw blocks, mkfs
Filesystem	The structure (inodes, directories, metadata) imposed on top	Creating/reading/writing files, mounting

SEE BOTH LAYERS ON A REAL SYSTEM AT ONCE

`lsblk -f` shows every block device alongside the filesystem type, label, and UUID actually living on it — a fast, direct way to see this chapter's own distinction reflected on a real, running system.

MKFS IS IMMEDIATELY, IRREVERSIBLY DESTRUCTIVE

Running `mkfs` on a partition that already holds data is destructive by default on most systems — it does not check for or warn about existing data first. Always double- and triple-check the target device before running it; a real, common, costly mistake, in the same spirit as `grub1`'s own repeated warnings about targeting the wrong device.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a partition table entry alone can't tell you what filesystem (if any) actually lives inside that partition.

View solution

EXERCISE 2

Explain why a file's name is not stored inside its own inode, and what this reveals about what a directory actually is.

View solution

EXERCISE 3

Write a real fstab line that mounts a device with UUID 9a8b-7c6d as ext4 at /data, and explain what each field means.

View solution

CHAPTER 1 QUICK REFERENCE

- A partition table (grub1-5) divides a disk; a **filesystem** structures what's inside one partition
- The **VFS** gives every program the same open/read/write calls, regardless of the actual filesystem underneath
- **Block device** — raw, unstructured blocks; **filesystem** — the structure (inodes, directories, metadata) on top
- **Superblock** — metadata about the filesystem; **inode** — metadata about one file, with no filename stored in it
- **mount** attaches a filesystem to a directory tree location; **fstab** makes that persistent across reboots
- `lsblk -f` — see block devices and their filesystems together, directly

- mkfs is destructive by default — always confirm the target device first

CHAPTER 2 OF 11

ext4 — The Traditional Default

Linux Filesystems

Chapter 2 · ext4 — The Traditional Default

`fs1-1` previewed journaling without explaining the mechanism. This chapter delivers that in full, plus everything else that makes ext4 the long-standing default Linux filesystem — and an honest answer to when it's still the right choice.

Journaling — Delivering on Chapter 1's Own Preview

Before making a change to the actual filesystem structure, ext4 first writes a small record of the *intended* change to a dedicated journal area. If the system crashes mid-operation, on the next mount ext4 replays the journal to either complete or cleanly roll back the incomplete change — avoiding the pre-journaling era's own slow, full-filesystem scan after every unclean shutdown.

- **journal** — full data and metadata journaling; the safest mode, the slowest
- **ordered** (the default) — metadata is journaled; actual data is written before its metadata commits — a genuinely good balance
- **writeback** — metadata only; the fastest, but the weakest guarantee — data itself could still be corrupted after a crash even though the filesystem's own *structure* stays consistent

Extents — A Real Improvement Over Block Mapping

Older filesystems (ext2/ext3) tracked a file's data location as a long list of individual block pointers — genuinely inefficient for a large, mostly contiguous file, which might need thousands of individual pointers. ext4's own **extents** represent a large contiguous run of blocks as one compact record — "blocks N through N+1000 belong to this file" — dramatically more efficient for large files, both in metadata size and in reducing fragmentation-related overhead.

Creating and Tuning an ext4 Filesystem

```
mkfs.ext4 /dev/sdb1
mkfs.ext4 -L mydata /dev/sdb1     # with a volume label

tune2fs -l /dev/sdb1              # list current filesystem parameters
tune2fs -L newlabel /dev/sdb1     # change the label after creation
```

`tune2fs` is the real, primary tool for adjusting ext4 parameters after creation — labels, reserved-block percentage, forced-check intervals — without needing to reformat and lose everything.

fsck — Checking and Repairing

```
fsck.ext4 /dev/sdb1
```

Checks, and can repair, filesystem consistency. Genuinely important: the check assumes nothing else is actively modifying the filesystem while it runs. For a root filesystem that can't simply be unmounted while running, `grub1-9`'s own chroot-rescue material is directly relevant — boot a live environment, mount the target filesystem there instead, and run `fsck` from outside it.

When ext4 Is Still the Right Choice

ext4 is mature, extremely well-tested, has the broadest compatibility and tooling support of any Linux filesystem, and has genuinely lower resource overhead than `fs1-3`'s Btrfs or `fs1-4`'s ZFS. The right default when you don't specifically need snapshots, checksumming, or integrated volume management, and simplicity, stability, and raw performance matter most. Btrfs and ZFS aren't free upgrades over ext4 — each trades some of this simplicity for genuinely new capabilities, a real tradeoff this course takes seriously rather than treating ext4 as simply "the old one."

MODE	WHAT'S PROTECTED	RELATIVE SPEED
journal	Data and metadata	Slowest
ordered (default)	Metadata, with data ordered before it commits	Balanced
writeback	Metadata only	Fastest

CHECK THE REAL JOURNALING MODE DIRECTLY

`dumpe2fs -h /dev/sdb1` reads the actual superblock and shows the current journaling mode and other real filesystem parameters — `fs1-1`'s own superblock material, made directly checkable.

NEVER RUN FSCK ON A MOUNTED, IN-USE FILESYSTEM

A genuinely serious, easy-to-make mistake: running `fsck` against a filesystem that's currently mounted and actively being written to can itself **cause** corruption — the exact opposite of the intended effect. Always unmount first, or check from a live/rescue environment for a root filesystem that can't be unmounted while running.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own three journaling modes, which mode you'd choose for a database server where data integrity matters more than raw write speed, and why.

View solution

EXERCISE 2

Explain why extents are a genuine improvement over block-by-block mapping specifically for a large, mostly-contiguous file like a video, referencing what each approach actually stores.

View solution

EXERCISE 3

A system administrator wants to run fsck on the currently-mounted root filesystem while the system is up and running. Explain why this is dangerous, and the correct alternative approach.

View solution

CHAPTER 2 QUICK REFERENCE

- **journal** (safest/slowest), **ordered** (default, balanced), **writeback** (fastest/weakest) — the three real journaling modes
- **Extents** represent contiguous block runs compactly, a real improvement over per-block mapping for large files
- `mkfs.ext4` creates; `tune2fs` adjusts parameters afterward without reformatting
- `fsck.ext4` checks/repairs — never on a mounted, actively-used filesystem
- `dumpe2fs -h` — read real filesystem parameters directly from the superblock
- ext4 remains the right default for maturity, compatibility, and low overhead when snapshots/checksumming/integrated volumes aren't specifically needed

- Btrfs (fs1-3) and ZFS (fs1-4) trade ext4's simplicity for genuinely new capabilities — real tradeoffs, not free upgrades

CHAPTER 3 OF 11

Btrfs — Copy-on-Write & Snapshots

Linux Filesystems

Chapter 3 · Btrfs — Copy-on-Write & Snapshots

`fs1-2` closed with a promise: Btrfs isn't a free upgrade over ext4, it's a genuinely different philosophy. This chapter is that difference, in full — starting with the one structural change that makes everything else in this chapter possible.

Copy-on-Write — The Real Structural Difference

ext4's own model: modifying a file generally updates data **in place**, overwriting existing blocks, with journaling protecting against a crash mid-write. Btrfs takes a genuinely different approach — **Copy-on-Write (COW)**. Modifying a file never overwrites existing blocks in place; Btrfs writes the new version to a fresh location, then atomically updates the metadata pointer to reference the new blocks instead of the old ones. The old blocks remain completely untouched until nothing references them anymore.

This single mechanism is what makes snapshots, below, essentially free and instant — a snapshot is just "a pointer to the metadata exactly as it existed at this moment," which COW naturally preserves, since old blocks are never overwritten while still referenced by anything.

Subvolumes — Btrfs's Own Internal Partitioning

A genuinely new concept with no ext4 equivalent: a **subvolume** is an independently mountable, independently snapshottable "filesystem within a filesystem," all sharing the same underlying storage pool.

```
btrfs subvolume create /mnt/data/mysubvol
btrfs subvolume list /mnt/data
```

A real use: separate subvolumes for `/`, `/home`, and `/var/log`, each independently snapshottable, without needing separate *partitions* the way ext4 would require for the same isolation.

Snapshots — The Real Payoff of COW

```
btrfs subvolume snapshot /mnt/data/mysubvol /mnt/data/mysubvol-snapshot
```

This completes almost instantly and initially consumes almost no extra space — the snapshot and the original share every single block, per COW. Only as the original (or the snapshot) is subsequently modified do the two start to diverge, with new blocks written only for whichever side changed, while unmodified blocks stay shared. A snapshot of a 500GB subvolume with no changes yet takes essentially zero additional disk space — not another 500GB. A real, practical use: taking a snapshot immediately before a risky system update, so a bad update can be rolled back simply by switching back to the pre-update snapshot — this is genuinely how tools like openSUSE's own Snapper rollback system work underneath.

Checksumming and Self-Healing

A genuinely new capability ext4 doesn't have at all: Btrfs checksums both data and metadata by default. On every read, the checksum is verified — if a block has silently corrupted (*bit rot*, often from a failing disk), Btrfs can **detect** this, something ext4 simply cannot do on its own; ext4 would just return the corrupted data, unaware anything was wrong. Running with redundancy (RAID1, previewed here, fuller coverage in `fs1-7` / `fs1-8`), Btrfs can automatically **self-heal** — reading the good copy from the mirror and repairing the corrupted one, transparently.

```
btrfs scrub start /mnt/data
```

Proactively reads and verifies every checksum on the filesystem, catching (and, where redundancy exists, repairing) corruption before it's ever encountered by a real read — a genuinely valuable maintenance operation with no ext4 equivalent.

A Concrete Worked Example — Snapshot Before a Risky Change

```
btrfs subvolume snapshot / /.snapshots/pre-update-$(date +%Y%m%d)
apt full-upgrade
# if something breaks: boot into the snapshot instead of the current (broken) root subvolume
```

Take the snapshot, run the upgrade, and if it breaks something, the pre-update state is right there, ready to boot into instead — no restore process, no waiting.

	WRITE MODEL	SNAPSHOT COST	CORRUPTION DETECTION
ext4	In-place, journaled	No native snapshots	None — silently returns corrupted data
Btrfs	Copy-on-write	Near-instant, near-free	Checksummed on every read, self-healing with redundancy

REAL SPACE ACCOUNTING NEEDS A BTRFS-AWARE TOOL

`btrfs filesystem df /mnt/data` shows accurate space usage that accounts for COW and shared blocks between a subvolume and its snapshots — a traditional `df` can be genuinely misleading here, since it doesn't understand block sharing.

SNAPSHOTS ARE NOT BACKUPS

A snapshot typically lives on the **same** physical disk or pool as the data it's snapshotting — it protects against "I made a mistake" or "this update broke something," but not against the disk itself failing entirely. A real backup strategy still needs data copied to genuinely separate storage; a snapshot alone is not that.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own COW model, why a Btrfs snapshot of an unmodified 500GB subvolume takes almost no additional disk space, while a full copy of the same data would take another 500GB.

View solution

EXERCISE 2

Explain why ext4 would silently return corrupted data after a bit-rot event on a failing disk, while Btrfs would detect the problem, referencing what each filesystem actually does (or doesn't do) on a read.

View solution

EXERCISE 3

A team relies entirely on nightly Btrfs snapshots of their production data, with no separate backup system, reasoning "we have snapshots, we're covered." Explain the real gap in this reasoning, using this chapter's own warn-box.

View solution

CHAPTER 3 QUICK REFERENCE

- **Copy-on-write** — modifications write new blocks instead of overwriting in place; old blocks live on until unreferenced
- **Subvolumes** — independently mountable/snapshottable filesystems-within-a-filesystem, no separate partitions needed
- Snapshots are near-instant and near-free — they share every block with the original until something changes

- Btrfs checksums data and metadata by default — real corruption detection ext4 doesn't have, plus self-healing with redundancy
- `btrfs scrub` proactively verifies and repairs before a real read ever hits corrupted data
- `btrfs filesystem df` — real, COW-aware space accounting, unlike a traditional df
- Snapshots are not backups — same disk/pool, no protection against actual disk failure

CHAPTER 4 OF 11

ZFS — A Different Philosophy Entirely

Linux Filesystems

Chapter 4 · ZFS — A Different Philosophy Entirely

Btrfs (`fs1-3`) shares two real ideas with what's coming next — copy-on-write and built-in checksums. ZFS takes both further, and adds one more: it doesn't just sit *on top of* a partition, it manages the disks, the redundancy, and the filesystems all as one integrated system. That's the genuinely different philosophy this chapter is about.

The Integrated Pool/Dataset Model

The traditional stack — and Btrfs, mostly — still separates layers: a partition or RAID array, then (optionally) LVM, then a filesystem on top. Each layer is configured independently, with its own tools and its own failure modes.

ZFS collapses all of that into one system. A **zpool** is built directly from raw disks or partitions, with redundancy (mirroring, RAID-Z) configured at pool-creation time — no separate mdadm or LVM layer needed underneath at all. **Datasets** (ZFS's own term for filesystems) are then carved out of that pool on demand, each one a lightweight, independently-configurable filesystem sharing the pool's underlying storage and free space.

```
zpool create tank mirror /dev/sdb /dev/sdc
zfs create tank/data
zfs list
```

One command creates a mirrored, redundant storage pool from two raw disks. Another carves a dataset out of it. No partitioning step, no separate RAID tool, no separate volume manager — ZFS *is* all three layers.

Real Strengths

- **Checksums** — data and metadata, verified on every read, same principle as Btrfs (`fs1-3`) — ZFS pioneered this approach.
- **Snapshots and clones** — the same COW-based, near-instant, near-free mechanism as Btrfs: `zfs snapshot tank/data@backup1` .
- **send/receive** — a genuinely unique, powerful capability: replicate exactly the changed blocks between a snapshot and its predecessor to another pool or host, efficiently and incrementally.

- **RAID-Z** — ZFS's own answer to RAID5/6, designed specifically to avoid the "RAID write hole" (a power-loss-during-write scenario that can silently corrupt a traditional RAID array's parity) by tying redundancy directly into the same transactional, copy-on-write model as everything else in ZFS.

A Worked Example — send/receive Replication

```
zfs snapshot tank/data@daily-2026-07-14
zfs send tank/data@daily-2026-07-14 | ssh backup-host zfs receive backup/data

# the next day, send only what changed since yesterday's snapshot
zfs send -i tank/data@daily-2026-07-13 tank/data@daily-2026-07-14 | ssh backup-host zfs receive backu
```

The `-i` incremental form sends only the blocks that changed between the two snapshots, not the entire dataset again — a genuinely efficient replication mechanism for keeping a remote backup pool current, night after night, without re-transferring unchanged data.

An Honest Note on Linux Licensing & Kernel Integration

ZFS originated at Sun Microsystems and is licensed under the CDDL — a license the Linux kernel's own maintainers and the FSF consider legally incompatible with the GPL for the purpose of shipping ZFS built directly into the mainline Linux kernel. This is a real, ongoing legal position, not a technical limitation of ZFS itself.

The practical consequence: ZFS on Linux (distributed as **OpenZFS**) ships as an out-of-tree kernel module built via DKMS, not bundled with the kernel the way Btrfs has been since 2009. Some distributions make this easy (Ubuntu ships it as a straightforward installable option); others require more manual setup. On FreeBSD, by contrast, ZFS ships natively with no such licensing tension at all — this friction is genuinely Linux-specific, not a limitation of ZFS itself.

	VOLUME/RAID MANAGEMENT	LINUX KERNEL INTEGRATION	UNIQUE CAPABILITY
Btrfs	Built-in, native	Mainline since 2009	Subvolumes as first-class Linux-native citizens
ZFS	Built-in, native (zpool)	Out-of-tree module (DKMS) — CDDL/GPL friction	send/receive incremental replication

CHECK POOL HEALTH AT A GLANCE

`zpool status` shows real-time pool health, any checksum errors found and corrected, and whether a resilver (rebuilding redundancy after a disk replacement) is currently in progress.

A KERNEL UPDATE CAN OUTRUN THE DKMS MODULE

Because the ZFS module builds against the currently-installed kernel via DKMS, a kernel update can leave the new kernel without a working ZFS module until DKMS finishes rebuilding it — `zpool import` can fail entirely on the new kernel in the meantime. If this happens, `grub1-4`'s own live-edit skill applies directly: boot into the previous kernel from the GRUB menu while the module catches up, rather than assuming ZFS itself is broken.

Hands-On Exercises

EXERCISE 1

Explain what a traditional stack (ext4 or Btrfs on top of LVM on top of mdadm RAID) requires that a single `zpool create tank mirror /dev/sdb /dev/sdc` command does not, and why.

View solution

EXERCISE 2

A team runs a nightly backup by re-copying an entire 2TB dataset to a remote host every night, even though only a small fraction of the data changes day to day. Explain how ZFS's own send/receive mechanism would improve this, and why.

View solution

EXERCISE 3

After a routine kernel update and reboot, a server's ZFS pool fails to import with no obvious disk-level error. Using this chapter's own warn-box, explain the likely cause and the correct next step.

View solution

CHAPTER 4 QUICK REFERENCE

- **zpool** — ZFS's integrated storage pool, built directly from raw disks, redundancy configured at creation time
- **dataset** — a lightweight filesystem carved out of a pool, ZFS's own term for what Btrfs calls a subvolume
- No separate LVM or mdadm layer needed — ZFS manages volumes and redundancy itself
- Checksums + snapshots/clones — the same COW-based guarantees as Btrfs

- `zfs send` / `zfs receive` — efficient, incremental, snapshot-to-snapshot replication
- **RAID-Z** — ZFS's own RAID5/6 answer, designed to avoid the traditional RAID write hole
- CDDL vs. GPL — real, ongoing licensing friction that keeps ZFS out of the mainline Linux kernel, unlike Btrfs
- `zpool status` — real-time pool health, checksum errors, resilver progress

CHAPTER 5 OF 11

LVM — Logical Volume Manager

Linux Filesystems

Chapter 5 · LVM — Logical Volume Manager

Btrfs and ZFS (`fs1-3` , `fs1-4`) manage volumes themselves, internally. This chapter is about the traditional alternative for filesystems that don't — ext4 and XFS most commonly — a separate layer purpose-built to sit between raw storage and the filesystem: **LVM**, the Logical Volume Manager. It's also the chapter that finally delivers on a promise `grub1-8` deliberately left open.

The PV/VG/LV Model

- **Physical Volume (PV)** — a raw disk or partition initialized for LVM's use.
- **Volume Group (VG)** — a pool combining one or more PVs into a single pool of storage, potentially spanning multiple physical disks.
- **Logical Volume (LV)** — a "virtual partition" carved out of a VG — this is what actually gets formatted with a filesystem and mounted.

```
pvcreate /dev/sdb1
vgcreate myvg /dev/sdb1
lvcreate -L 50G -n mylv myvg
mkfs.ext4 /dev/myvg/mylv
mount /dev/myvg/mylv /mnt/data
```

Why This Extra Layer Exists — Flexibility

The value LVM adds beyond a plain partition: a logical volume isn't tied to the physical geometry of a single disk. A volume group can span multiple physical disks, so a logical volume can be larger than any single disk in the machine. And, most importantly, logical volumes can be resized **live** — without unmounting, without downtime, and without the rigid fixed-size commitment a plain partition forces at creation time.

Live Resizing — LVM's Real Payoff

Growing a logical volume can be done while it's mounted and actively in use:

```
lvextend -L +20G /dev/myvg/mylv
resize2fs /dev/myvg/mylv
```

Shrinking is genuinely riskier, and the order matters: the *filesystem* must be shrunk first (`resize2fs` to the smaller target size), and only then can the logical volume itself be reduced (`lvreduce`). Doing it in the wrong order truncates the logical volume out from under data the filesystem still thinks it owns — see the warn-box below.

Delivering on grub1-8's Deferred Promise

`grub1-8` deliberately deferred LVM considerations in the context of multi-boot partitioning. Here's the real answer: GRUB **can** boot a root filesystem that lives on an LVM logical volume — `grub-mkconfig` detects LVM automatically and writes the correct boot parameters. But `/boot` itself is commonly kept **outside** LVM, as its own small, plain partition, because GRUB's ability to parse LVM metadata directly to locate a kernel and initrd is more limited and more fragile than reading a plain partition — especially once LVM is layered on RAID, or the volume is encrypted. This is exactly why Debian's own default installer layout (seen back in `linux1`) carves out a small separate `/boot` partition even when the rest of the disk uses LVM.

In a real multi-boot setup, this makes LVM genuinely useful: each Linux distribution's own root filesystem can live in its own logical volume within a shared volume group, letting disk space be divided among multiple distros flexibly rather than requiring fixed-size partitions decided up front — while GRUB itself, and ideally `/boot`, stays on plain partitions for maximum compatibility across every OS involved.

A Worked Example — Growing a Live Filesystem

```
# the /mnt/data volume is running low on space; the VG still has free room
vgs                          # confirm free space exists in the volume group
lvextend -L +20G /dev/myvg/mylv
resize2fs /dev/myvg/mylv     # grows the filesystem to fill the new space, live, no unmount
df -h /mnt/data
```

	RESIZABLE LIVE?	SPANS MULTIPLE DISKS?	SNAPSHOT SUPPORT?
Plain partition	No — fixed at creation	No	No
LVM logical volume	Grow: yes. Shrink: filesystem-dependent, order matters	Yes, via the volume group	Yes — LVM snapshots (COW-based, distinct from Btrfs/ZFS's own)

A QUICK SUMMARY VIEW OF ALL THREE LAYERS

`pvs`, `vgs`, and `lvs` each print a one-line-per-item summary of physical volumes, volume groups, and logical volumes respectively. `pvdisplay` / `vgdisplay` / `lvdisplay` give the full detail view for each when a summary isn't enough.

SHRINK ORDER MATTERS — GET IT BACKWARDS AND YOU LOSE DATA

Shrinking must always go filesystem first, then logical volume — `resize2fs` to the smaller size, *then* `lvreduce`. Reducing the logical volume before the filesystem has been shrunk to fit truncates the space out from under data the filesystem still believes it owns, corrupting it. Also worth knowing: **XFS cannot be shrunk at all** — an XFS filesystem can only grow, never shrink, regardless of order.

Hands-On Exercises

EXERCISE 1

A server has two physical disks, /dev/sdb and /dev/sdc, and needs one filesystem larger than either disk alone. Write the sequence of LVM commands needed to build this, from raw disks to a mounted filesystem.

View solution

EXERCISE 2

An administrator runs `lvreduce` on a logical volume to shrink it before running `resize2fs` on the filesystem it holds. Explain what goes wrong and what the correct order should have been.

View solution

EXERCISE 3

A user setting up a triple-boot machine (echoing grub1-8's own scenario) asks whether they can put GRUB's own /boot partition on an LVM logical volume alongside their distros' root filesystems. Using this chapter's own explanation, answer their question.

View solution

CHAPTER 5 QUICK REFERENCE

- **PV → VG → LV** — physical volume, volume group, logical volume, in that order
- `pvcreate` / `vgcreate` / `lvcreate` — the creation commands for each layer
- A volume group can span multiple physical disks — a logical volume can exceed any single disk's size

- Growing is live and safe: `lvextend` then `resize2fs` / `xfs_growfs`
- Shrinking order matters: filesystem first (`resize2fs`), then `lvreduce` — reversed order corrupts data
- XFS can only grow, never shrink, regardless of order
- `/boot` is commonly kept outside LVM as a plain partition for GRUB compatibility, even when the rest of the disk uses LVM
- `pvs` / `vgs` / `lvs` — quick one-line-per-item summaries of each layer

CHAPTER 6 OF 11

LVM Snapshots & Practical Use Cases

Linux Filesystems

Chapter 6 · LVM Snapshots & Practical Use Cases

`fs1-5` 's own compare-table flagged LVM snapshots as "COW-based, distinct from Btrfs/ZFS's own." This chapter is that distinction, in full — plus the one real-world use case that makes LVM snapshots worth knowing well: safely backing up data that's actively being written to.

How LVM Snapshots Work — COW at the Block Layer, Not the Filesystem Layer

Btrfs and ZFS snapshots (`fs1-3` , `fs1-4`) are implemented inside the filesystem itself, which natively understands blocks being shared between a subvolume/dataset and its snapshots. LVM snapshots are a genuinely different mechanism, working one layer **below** the filesystem — the filesystem sitting on top has no idea a snapshot even exists.

Creating an LVM snapshot doesn't copy the volume. It creates a new, small logical volume that initially holds nothing, reserved purely to record *changes* made to the original after the snapshot point. When the original volume is written to, LVM first copies the block's **old** content into the snapshot volume before letting the write proceed — copy-on-write, but implemented as "move the old data out of the way," not Btrfs's more sophisticated multi-way block sharing. This means an LVM snapshot has to be sized in advance to hold however many changed blocks are expected during its lifetime — a real, practical difference from Btrfs/ZFS's own dynamically-growing snapshots.

Real Commands

```
lvcreate -L 5G -s -n mylv-snap /dev/myvg/mylv
mount /dev/myvg/mylv-snap /mnt/snap

# roll the origin back to the snapshot's point-in-time state
lvconvert --merge /dev/myvg/mylv-snap

# done with the snapshot — release its reserved space
lvremove /dev/myvg/mylv-snap
```

`-s` marks the new logical volume as a snapshot of `mylv` ; `-L 5G` reserves 5GB purely for the snapshot's own changed-block store — not a full copy of the original volume's size.

Why the Reserved Size Matters — Snapshot Overflow

Because LVM snapshots pre-allocate a fixed amount of space for tracking changed blocks, rather than growing dynamically the way Btrfs/ZFS snapshots effectively do, a snapshot that runs out of reserved space while the original keeps changing simply **overflows** — becoming invalid and unusable. This is a real, practical gotcha that needs active monitoring, not a one-time sizing decision that can be forgotten.

A Real Use Case — Safe Backups of a Live Database

`dbsec1-8` covered backup security — encryption, tested restores, backups as a ransomware target. Here's the mechanical problem LVM snapshots solve underneath all of that: backing up a live, actively-written database file directly risks capturing an inconsistent, mid-write state — a backup that looks complete but is actually corrupted, because different parts of the file were captured at different moments while writes kept happening in between.

An LVM snapshot solves this cleanly. The snapshot itself is taken near-instantaneously, freezing the origin volume's exact state at that moment from the snapshot's own point of view — while the *original* keeps being written to completely normally, with essentially no downtime. The backup then reads from the frozen, unchanging snapshot at leisure, with a fully consistent view throughout, regardless of how long the backup itself takes or how much the live data changes in the meantime.

```
lvcreate -L 10G -s -n db-snap /dev/dbvg/dblv
mount -o ro /dev/dbvg/db-snap /mnt/db-backup
tar czf /backup/db-$(date +%Y%m%d).tar.gz /mnt/db-backup
umount /mnt/db-backup
lvremove /dev/dbvg/db-snap
```

The archive this workflow produces is exactly the kind of artifact `dbsec1-8` 's own encryption-at-rest guidance should be applied to before it's stored anywhere long-term — this chapter solves the "how do I capture a consistent snapshot of live data" problem; `dbsec1-8` already covers what happens to that snapshot once it's a backup file sitting in storage.

	SPACE ALLOCATION	LAYER	OVERFLOW RISK
Btrfs / ZFS snapshot	Dynamic, grows with actual changes	Inside the filesystem itself	None — bounded only by overall pool space
LVM snapshot	Fixed, reserved at creation time	Block layer, below the filesystem	Real — exceeding reserved space invalidates the snapshot

WATCH USAGE BEFORE IT OVERFLOWS

`lvs` shows each snapshot's current usage as a percentage of its own reserved space — checking this during a long-running backup or before extending a snapshot's lifetime is the practical way to avoid a silent overflow.

SIZE FOR THE CHANGE VOLUME, NOT THE DATA VOLUME

The `-L` size on an LVM snapshot needs to cover the total amount of data expected to **change** on the origin during the snapshot's entire lifetime — not the origin's total size. A heavily-written database left snapshotted for hours can overflow a snapshot that seemed generously sized, silently invalidating it and losing the point-in-time state it existed to preserve.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own mechanism description, why an LVM snapshot needs a size reserved in advance, while a Btrfs or ZFS snapshot does not.

View solution

EXERCISE 2

An administrator wants to back up a 200GB live database file that receives roughly 2GB of writes per hour, and plans to run the backup over a 6-hour window using an LVM snapshot. Recommend a reserved snapshot size and explain your reasoning.

View solution

EXERCISE 3

Explain why backing up a live, actively-written database file directly (without a snapshot) risks producing a corrupted backup, and how the LVM snapshot workflow in this chapter avoids that risk.

View solution

CHAPTER 6 QUICK REFERENCE

- LVM snapshots implement COW at the block layer, below the filesystem — the filesystem has no idea a snapshot exists
- A snapshot volume stores only changed blocks (the origin's *old* content, copied out before being overwritten), not a full copy
- `lvcreate -L <size> -s -n <name> <origin>` — create a snapshot with a fixed reserved size

- `lvconvert --merge` — roll the origin back to the snapshot's point-in-time state
- Reserved space is fixed, not dynamic — exceeding it overflows and invalidates the snapshot
- `lvs` — monitor snapshot usage percentage before it overflows
- Real use case: freeze a consistent, point-in-time view of live data for a backup, with no downtime on the original
- The resulting backup file is exactly what `dbsec1-8`'s own encryption-at-rest guidance applies to

CHAPTER 7 OF 11

RAID Concepts & Levels

Linux Filesystems

Chapter 7 · RAID Concepts & Levels

Every prior chapter has assumed working disks. This chapter is about combining several — RAID (Redundant Array of Independent Disks). It's a conceptual chapter on purpose: the five levels worth actually knowing, and the real tradeoffs each one makes. `fs1-8` covers the real `mdadm` commands to build one.

What RAID Actually Trades Off

RAID combines multiple physical disks into one logical unit, but every level makes a genuinely different tradeoff between three things: **performance** (via striping data across disks), **redundancy** (via mirroring or parity), and **usable capacity** (how much of the raw disk space ends up available for real data versus spent on redundancy). No RAID level maximizes all three at once — that tension runs through every level below.

RAID 0 — Striping (Performance, Zero Redundancy)

Data is split ("striped") across all disks with no redundancy at all. Usable capacity is the full sum of every disk. Performance is the best of any level here, since reads and writes spread across multiple disks in parallel. But losing **any single disk** in the array loses **all** the data — reliability actually gets *worse* than a single disk, not better, since more disks means more independent chances of one failing.

Honest tradeoff: only worth it when redundancy already exists elsewhere (a database replica, a separate backup) and pure speed matters more than this specific array's own survival.

RAID 1 — Mirroring (Redundancy, Halved Capacity)

Every disk is an exact duplicate of every other. Usable capacity is the size of the smallest single disk — with two disks, that's 50% of the raw total. Survives any single disk failure completely (with two disks). Read performance can genuinely improve (reads spread across mirrors); write performance is roughly that of one disk, since every write has to go to every mirror.

Honest tradeoff: expensive in capacity — paying for N−1 disks' worth of pure redundancy — but the simplest, most predictable RAID level, with the least complicated failure story of any level here.

RAID 5 — Striping with Distributed Parity

Data is striped like RAID 0, but one disk's worth of capacity is used for parity data, distributed across all disks rather than living on one dedicated disk. Survives exactly **one** disk failure. Usable capacity is (N−1) disks' worth — better capacity efficiency than RAID 1 as disk count grows.

The real, well-known weakness is **rebuild time**. Replacing a failed disk requires reading every remaining disk to reconstruct the failed one's data — on large modern drives this can take many hours, and a **second** disk failure during that rebuild window loses the entire array. This risk grows directly with disk size, which is the genuine, practical reason RAID 5 is increasingly discouraged on today's largest drives. It's also where the "RAID write hole" lives — a power loss mid-write to a stripe can leave data and its parity inconsistent, exactly the problem ZFS's RAID-Z (`fs1-4`) was designed to avoid by tying parity into the same transactional model as everything else in ZFS.

RAID 6 — Striping with Double Distributed Parity

Like RAID 5, but with **two** parity blocks distributed across the array — survives **two** simultaneous disk failures. Usable capacity is (N−2) disks' worth. This directly addresses RAID 5's own rebuild-window vulnerability: even if a second disk fails during the rebuild after the first, RAID 6 can still survive it.

Honest tradeoff: write performance is worse than RAID 5 (two parity calculations per write instead of one), and it needs at least four disks to be worthwhile at all.

RAID 10 — Mirroring + Striping Combined

A nested/hybrid level: disks are paired and mirrored (RAID 1), then those mirrored pairs are striped together (RAID 0). Gets RAID 0's performance *and* real redundancy — survives multiple disk failures, as long as no single mirrored pair loses both of its own disks. Rebuilding a failed disk just means re-mirroring from its own surviving partner — much faster, and much lower-risk, than reconstructing from parity across the whole array the way RAID 5/6 do.

Honest tradeoff: usable capacity is only 50% of raw, same as RAID 1, and it needs at least four disks (two mirrored pairs minimum). Often the preferred choice for performance-critical database workloads specifically because of its fast, low-risk rebuilds.

Tying to perf1's Disk I/O & Storage Bottlenecks

`perf1`'s own Disk I/O & Storage Bottlenecks chapter covered diagnosing a disk-bound workload — `iostat`, `%util`, IOPS — without addressing the underlying storage architecture itself. Striping (RAID 0/5/6/10) is a direct architectural answer to a genuinely disk-bound bottleneck diagnosed there: spreading

read/write load across multiple physical disks in parallel raises the effective IOPS and throughput ceiling those same diagnostic tools would measure. But striping doesn't fix a bottleneck that was never really disk-bound in the first place — a workload misdiagnosed as disk-bound when it's actually CPU- or memory-bound gains nothing from RAID at all. The diagnostic step from `perf1` still has to come first.

LEVEL	MIN DISKS	USABLE CAPACITY	SURVIVES	REBUILD RISK
RAID 0	2	100% of raw	Nothing	N/A — no redundancy to rebuild
RAID 1	2	50% of raw	1 disk	Low — simple re-mirror
RAID 5	3	(N−1)/N of raw	1 disk	High — long parity rebuild, 2nd failure loses array
RAID 6	4	(N−2)/N of raw	2 disks	Moderate — survives a 2nd failure mid-rebuild
RAID 10	4	50% of raw	Multiple, if not both disks in one pair	Low — fast re-mirror, no parity math

THERE IS NO "BEST" LEVEL, ONLY THE RIGHT TRADEOFF

Choosing a RAID level is a genuine decision informed by workload (read-heavy vs. write-heavy), disk count, disk size (rebuild-time risk), and how much capacity is worth spending on redundancy — not a search for a single objectively-best option.

RAID IS NOT A BACKUP

Echoing `fs1-3`'s own "snapshots are not backups" warning: RAID protects against a physical disk failing, nothing more. It does nothing against accidental deletion, ransomware, or a filesystem-level bug — any of those get written identically and consistently to every disk in the array, redundancy and all. A real backup on genuinely separate storage is still required regardless of RAID level.

Hands-On Exercises

EXERCISE 1

A team wants to maximize raw read/write speed for a scratch/cache workload where the data can be regenerated at any time if lost. Recommend a RAID level and explain why, using this chapter's own tradeoffs.

View solution

EXERCISE 2

A team is running RAID 5 on very large (18TB) drives and is considering migrating to RAID 6. Explain the specific risk this chapter names that motivates that migration.

View solution

EXERCISE 3

A team running RAID 10 for their production database was hit by ransomware that encrypted every file in place. They assumed their RAID array meant they were protected. Explain why they weren't, using this chapter's own warn-box.

View solution

CHAPTER 7 QUICK REFERENCE

- Every RAID level trades off performance, redundancy, and usable capacity — none maximizes all three
- **RAID 0** — striping, full capacity, zero redundancy, best performance, worst reliability
- **RAID 1** — mirroring, 50% capacity, survives 1 failure, simplest failure story
- **RAID 5** — striping + 1 distributed parity, (N−1)/N capacity, real rebuild-window risk on large disks
- **RAID 6** — striping + 2 distributed parity, (N−2)/N capacity, survives 2 failures, slower writes
- **RAID 10** — mirrored pairs, striped together, 50% capacity, fast low-risk rebuilds, good for databases
- RAID striping raises the IOPS/throughput ceiling `perf1`'s own tools measure — but only for a genuinely disk-bound workload
- RAID is not a backup — it protects against disk failure only, not deletion, ransomware, or corruption written identically everywhere

CHAPTER 8 OF 11

Software RAID with mdadm

Linux Filesystems

Chapter 8 · Software RAID with mdadm

`fs1-7` covered the levels and their tradeoffs conceptually. This chapter is the real tool Linux uses to build and manage them: **mdadm** ("multiple disk admin"), and a full, real walkthrough of the moment a RAID array actually earns its keep — a disk failing and getting replaced.

Creating a RAID Array

```
# RAID 1 (mirroring) across two partitions
mdadm --create /dev/md0 --level=1 --raid-devices=2 /dev/sdb1 /dev/sdc1

# RAID 5 (striping + distributed parity) across three partitions
mdadm --create /dev/md0 --level=5 --raid-devices=3 /dev/sdb1 /dev/sdc1 /dev/sdd1

mkfs.ext4 /dev/md0
mount /dev/md0 /mnt/raid
```

`--level` and `--raid-devices` map directly onto `fs1-7`'s own concepts — `--level=1` is mirroring, `--level=5` is striping with one distributed parity block, and so on. Once created, `/dev/md0` behaves like any other block device — format it and mount it directly, or, per `fs1-5`, put LVM on top of it instead of formatting it directly (the full layered stack is covered in `fs1-9`).

Persisting the Array Configuration

```
mdadm --detail --scan >> /etc/mdadm/mdadm.conf
update-initramfs -u
```

Without writing the array's own configuration to `mdadm.conf`, the array may not reassemble correctly — or with the wrong device ordering — on the next reboot. This is an easy step to forget right after `--create` succeeds and everything appears to be working. `update-initramfs -u` ensures the initramfs itself knows how to assemble the array early in the boot process, which matters especially when the root filesystem itself lives on RAID.

Monitoring Array Health

```
cat /proc/mdstat
```

The primary, always-available way to check array status. Shows every active array, its member disks, and a health indicator like `[UU]` — each letter represents one member disk, `U` meaning up/healthy, an underscore meaning missing or failed. A RAID 5 array showing `[UU_]` instead of `[UUU]` is degraded, missing exactly the redundancy `fs1-7` covered.

```
mdadm --detail /dev/md0
```

Fuller detail than `/proc/mdstat` — array state, rebuild progress, and event count. For genuine production use, `mdadm --monitor --scan` can run as a daemon that emails an alert the moment an array degrades or a disk fails, rather than relying on someone remembering to check manually.

A Real Disk Failure Simulation

```
# mark a member as failed
mdadm /dev/md0 --fail /dev/sdc1
cat /proc/mdstat                    # now shows a degraded array, e.g. [U_]

# remove the failed member before physically pulling the disk
mdadm /dev/md0 --remove /dev/sdc1

# -- physically replace the disk, partition the new one to match --

# add the replacement — triggers an automatic rebuild
mdadm /dev/md0 --add /dev/sdc1
cat /proc/mdstat                    # shows "recovering", a percentage, speed, and ETA
```

The Rebuild Window in Practice

Watching `/proc/mdstat` climb through a RAID 5 rebuild is `fs1-7`'s own rebuild-risk warning made concrete: while the state shows `recovering` and a percentage ticking upward, the array is running with zero further redundancy for a RAID 5 array — exactly the exposed window `fs1-7` warned about, now visible directly in the monitoring output rather than as an abstract risk.

COMMAND	PURPOSE
mdadm --create	Build a new array from raw disks/partitions

COMMAND	PURPOSE
mdadm --detail --scan	Print config for persisting to mdadm.conf
cat /proc/mdstat	Quick array status and health at a glance
mdadm --detail	Full status: state, rebuild progress, events
mdadm --fail / --remove / --add	The failed-disk replacement sequence

PERSIST THE CONFIG RIGHT AFTER CREATION, NOT "LATER"

Run `mdadm --detail --scan >> /etc/mdadm/mdadm.conf` immediately after `mdadm --create` succeeds — it's easy to forget once the array is already working, and the gap only becomes visible at the worst possible time: the next reboot.

NEVER PULL A DISK BEFORE --FAIL AND --REMOVE

Physically removing a disk from a live array before both `--fail` and `--remove` have been run against it leaves Linux potentially still attempting I/O against a device that's already physically gone — a real path to an unclean array state and additional corruption risk, on top of whatever originally prompted the disk's removal.

Hands-On Exercises

EXERCISE 1

Write the full mdadm command to create a RAID 6 array named /dev/md0 from four disks: /dev/sdb1, /dev/sdc1, /dev/sdd1, /dev/sde1. Cross-check the resulting usable capacity against fs1-7's own compare-table.

View solution

EXERCISE 2

An administrator creates a new mdadm array, formats and mounts it successfully, but skips writing its configuration to mdadm.conf. Explain what's likely to go wrong, and when.

View solution

EXERCISE 3

An administrator notices a disk showing signs of failure and, wanting to act fast, immediately pulls it out of the running server without running any mdadm commands first. Explain what this chapter warns is risky about that, and what the correct sequence would have been.

View solution

CHAPTER 8 QUICK REFERENCE

- `mdadm --create /dev/mdX --level=N --raid-devices=N <devices>` — build an array
- `mdadm --detail --scan >> /etc/mdadm/mdadm.conf` — persist the config immediately, not later
- `update-initramfs -u` — needed if root itself lives on the array
- `cat /proc/mdstat` — quick health check, `[UU]` vs. `[U_]`
- `mdadm --detail /dev/mdX` — full state, rebuild progress, event count
- Failed-disk sequence: `--fail`, then `--remove`, then physically replace, then `--add`
- Never physically pull a disk before `--fail`/`--remove` — risks an unclean array state
- Watching a RAID 5 rebuild in `/proc/mdstat` makes fs1-7's own rebuild-window risk directly visible

CHAPTER 9 OF 11

Combining Layers — Traditional Stack vs. Integrated Filesystems

Linux Filesystems

Chapter 9 · Combining Layers — Traditional Stack vs. Integrated Filesystems

Every piece is now on the table: ext4 (`fs1-2`), Btrfs (`fs1-3`), ZFS (`fs1-4`), LVM (`fs1-5` , `fs1-6`), RAID concepts (`fs1-7`) and `mdadm` (`fs1-8`). This chapter is the course's own synthesis: two genuinely different philosophies for assembling them into a real storage stack, laid out honestly, tradeoffs and all.

The Traditional Layered Stack

Raw disks → `mdadm` RAID → LVM (PV/VG/LV) → ext4 (or XFS) → mount. Four separate subsystems, each configured, monitored, and troubleshot independently — `mdadm --detail` for RAID health, `lvs` for LVM, `tune2fs` / `dumpe2fs` for the filesystem itself.

The real benefit is composability: any single layer can be swapped without touching the others — hardware RAID in place of `mdadm` , XFS in place of ext4 — the classic Unix "do one thing well" philosophy applied to storage, genuinely mature and well understood.

The real cost is that each layer has zero awareness of the others. RAID doesn't know about the filesystem's own data; the filesystem doesn't know the RAID layer even exists. And, per `fs1-2` , ext4 itself does no content checksumming at all — nothing in this entire four-layer stack is actually verifying that the bytes coming back out are the bytes that were written.

The Integrated Approach — Btrfs and ZFS

Btrfs's own built-in RAID profiles and ZFS's own RAID-Z (`fs1-4`) eliminate the separate RAID layer entirely — the filesystem *is* the volume manager *is* the redundancy layer, all one system.

This is the concrete mechanism behind the "self-healing" claim from `fs1-3` / `fs1-4` : checksumming can only repair corruption automatically if the same system that detects the bad checksum also knows exactly which redundant copy holds the good data — and that's only possible once there's no longer a layer boundary between "the data" and "the redundancy" for that information to get lost across. A traditional `mdadm` array has no such awareness — it faithfully mirrors or reconstructs whatever bytes it's given, healthy or corrupted, because it operates below the filesystem with no concept of "correct" content at all.

The real cost: less mix-and-match flexibility — swapping ZFS's own RAID-Z for a different vendor's RAID implementation isn't really an option the way swapping `mdadm` for hardware RAID is in the traditional stack — and, per `fs1-4`, ZFS specifically still carries real Linux kernel-integration friction.

A Concrete Illustration — Where Silent Corruption Gets Caught

Revisit `fs1-3`'s own bit-rot scenario through the lens of layering. In the traditional stack: a bit flips on one disk's physical media. `mdadm` has no checksum, so it can't distinguish the bad block from a good one — it does its normal job with whatever bytes it reads. LVM, above that, has no checksum either. ext4, above that, has none either (`fs1-2`). The corrupted data sails all the way up to the application, undetected at every single layer along the way.

In the integrated approach, the filesystem stores the checksum right alongside the data it protects and verifies it on every read — regardless of which physical disk or redundancy scheme sits underneath, because there's no longer a boundary for the corruption to slip past unnoticed.

Which Approach the Site's Own Courses Actually Use

Worth being honest about: `ws1`'s own web server course, and most of this site's Linux material generally, assumes a fairly traditional setup — plain partitions, sometimes LVM — rather than Btrfs or ZFS. That's not an oversight; it reflects that ext4 is still the overwhelmingly common real-world default across most Linux distributions' own installers. This directly echoes `fs1-2`'s own honest framing: "traditional" doesn't mean "obsolete," it means "still the default almost everywhere."

	FLEXIBILITY	CROSS-LAYER CORRUPTION DETECTION	COMPLEXITY / TOOLING
Traditional (mdadm + LVM + ext4)	High — any layer independently swappable	None — no layer checksums content	Four separate subsystems and toolsets
Integrated (Btrfs / ZFS)	Lower — RAID/volume layer isn't separately swappable	Real — checksum + redundancy managed together	One system, one toolset

TELLING THE TWO MODELS APART AT A GLANCE

`lsblk` shows separate `md`/LVM devices layered on top of each other in a traditional stack; `btrfs filesystem show` or `zpool status` shows one integrated pool directly, with no separate RAID device visible underneath it at all.

"INTEGRATED" IS A TRADEOFF, NOT A STRICT UPGRADE

A shop already deeply invested in `mdadm`/LVM tooling, monitoring, and staff familiarity has a real, legitimate migration cost to weigh against Btrfs/ZFS's real corruption-detection advantage — this chapter's own comparison is not an argument that the integrated approach is simply better in every case.

Hands-On Exercises

EXERCISE 1

Explain why a traditional mdadm + LVM + ext4 stack cannot detect a single bit-flip on one disk, walking through what each layer does and doesn't check.

View solution

EXERCISE 2

A team wants to switch from hardware RAID controllers to a different vendor's hardware RAID controllers in the future without disrupting their filesystem or volume layout. Which architecture (traditional or integrated) makes this easier, and why?

View solution

EXERCISE 3

A colleague argues that every Linux server should immediately migrate from mdadm+LVM+ext4 to ZFS, since ZFS "objectively fixes" the corruption-detection gap. Using this chapter's own warn-box, push back on the word "objectively."

View solution

CHAPTER 9 QUICK REFERENCE

- Traditional stack: raw disks → mdadm RAID → LVM → filesystem, four independent, swappable layers
- Integrated stack: Btrfs/ZFS collapse volume management, RAID, and the filesystem into one system
- Cross-layer corruption detection is only possible once the layer boundary itself disappears
- No layer in the traditional stack checksums content — corruption below the RAID layer sails through untouched
- Integrated approaches trade some mix-and-match flexibility for real corruption detection and self-healing
- `lsblk` reveals a traditional stack's separate layers; `zpool status` / `btrfs filesystem show` reveal an integrated pool
- ext4/traditional setups remain the real-world default across most distros — not obsolete, still the norm
- This is a genuine tradeoff decision, not a strict "integrated is always better" upgrade

CHAPTER 10 OF 11

Choosing the Right Filesystem/Stack for a Real Scenario

Linux Filesystems

Chapter 10 · Choosing the Right Filesystem/Stack for a Real Scenario

`fs1-9` laid the traditional and integrated approaches out honestly, side by side. This chapter turns that comparison into a practical framework, then applies it to three real, different scenarios — including one already on this site.

A Practical Decision Framework

1. Does the workload need live resizing or flexible space allocation over time? (points toward LVM, `fs1-5`)
2. Does the data need protection against silent corruption over its lifetime? (points toward Btrfs/ZFS, `fs1-3` / `fs1-4`)
3. Is there already deep staff familiarity and working tooling around a specific stack? (a real, legitimate factor favoring the traditional stack, per `fs1-9`'s own warn-box)
4. Does the system need a simple, dependable GRUB boot path? (favors a plain or LVM-based `/boot`, per `fs1-5` / `grub1-8`)
5. Is kernel-version stability across routine updates critical? (a real point against ZFS specifically, per `fs1-4`'s own DKMS gotcha)
6. Does the workload benefit from instant, cheap snapshots for rollback? (Btrfs/ZFS native, or LVM snapshots per `fs1-6`)
7. How many simultaneous disk failures must redundancy survive, and how large are the disks? (the RAID-level and rebuild-window question from `fs1-7`)

Applying the Framework to ws1's Own Web Server

`ws1` (Setting Up a Web Server on Debian) covers HTTPS, UFW, fail2ban, SSH hardening, security headers — a single production Debian instance serving web content and logs, with modest, predictable storage needs. Running it through the framework: no exotic corruption-detection requirement at scale, no advanced replication need, a strong preference for kernel-update stability on a production server (question 5 counts directly against ZFS here), and deep site-wide familiarity with ext4/LVM as the default (question 3).

Conclusion: ext4, optionally on top of LVM for future resize flexibility, is the right, appropriately boring choice for `ws1` — not because Btrfs or ZFS are worse filesystems, but because none of their specific strengths are

actually load-bearing for this particular server. This directly validates `fs1-2`'s own "When ext4 Is Still the Right Choice" framing and `fs1-9`'s own honest note about the site's real-world defaults, applied concretely rather than left abstract.

Applying the Framework to a Backup/Archive Server

A different scenario: a server whose entire job is storing large volumes of long-term backup data. Here the framework answers differently — question 2 (corruption over time) matters a great deal, since backup data can sit untouched for months, and undetected bit rot could go unnoticed until a restore is actually attempted and fails. Question 6 (cheap snapshots) matters directly for point-in-time backup retention. And ZFS's own `send`/`receive` (`fs1-4`) is a direct, purpose-built fit for replicating backups to an offsite host efficiently.

Conclusion: ZFS is a genuinely strong, deliberate fit here. The DKMS/kernel-update friction named in `fs1-4` is a real cost, but an acceptable one on a specialized backup server where checksumming and `send`/`receive` are precisely the strengths the whole system exists to use.

Applying the Framework to a High-Performance Database Server

A third scenario: a database server under heavy, latency-sensitive read/write load. Question 7 points toward RAID 10 specifically, per `fs1-7`'s own recommendation, for its fast, low-risk rebuilds. Worth naming directly here: copy-on-write filesystems (Btrfs/ZFS) can introduce real write amplification for database-style workloads doing lots of small, in-place-feeling updates, since COW never truly writes in place — an overhead ext4/XFS's own direct in-place writes don't carry.

Conclusion: RAID 10 (`fs1-7`/`fs1-8`) with LVM on top for live resizing (`fs1-5`), formatted with ext4 or XFS, is often the more appropriate stack for a heavy database workload — the traditional stack, chosen deliberately rather than by default.

SCENARIO	CHOSEN STACK	DECIDING FACTOR
ws1 web server	ext4 (+ optional LVM)	Stability, simplicity, no unused advanced features
Backup/archive server	ZFS	Corruption detection over time, send/receive replication
Database server	RAID 10 + LVM + ext4/XFS	Fast low-risk rebuilds, no COW write amplification

START FROM REQUIREMENTS, NOT TRENDS

There is no universal answer here — always start from the numbered questions above, applied to the actual workload, rather than choosing a filesystem because it's the newest or most talked about.

DON'T PAY FOR FEATURES YOU'LL NEVER USE

Choosing ZFS or Btrfs specifically for their advanced capabilities on a system that will never actually exercise them adds real operational complexity — DKMS for ZFS, a comparatively newer filesystem for Btrfs on some workloads — for no corresponding benefit. This echoes `fs1-9`'s own warn-box directly: match the stack to the actual requirement, not to which system sounds more advanced.

Hands-On Exercises

EXERCISE 1

Using this chapter's own seven-question framework, evaluate a small home NAS used purely for storing family photos and videos long-term, rarely accessed but never wanted to silently degrade. Recommend a stack and justify it against the framework.

View solution

EXERCISE 2

A team wants to run ZFS on their appliance-style embedded Linux devices specifically because "it has the best features," despite the devices needing extremely predictable, hands-off kernel updates in the field. Using this chapter's own warn-box and framework, explain the risk in that reasoning.

View solution

EXERCISE 3

Explain, in the chapter's own terms, why ws1's web server and the backup/archive server scenario reach opposite conclusions on question 2 (corruption over time) even though both are running on the same underlying disk hardware.

View solution

CHAPTER 10 QUICK REFERENCE

- Seven-question framework: resizing, corruption protection, staff familiarity, GRUB simplicity, kernel stability, snapshots, failure/rebuild tolerance
- ws1's web server → ext4 (+ optional LVM) — no advanced feature is actually load-bearing here
- Backup/archive server → ZFS — corruption detection and send/receive are exactly the point
- Database server → RAID 10 + LVM + ext4/XFS — fast rebuilds, no COW write amplification
- COW filesystems can add real write-amplification overhead for small-write, database-style workloads
- Always start from actual requirements, never from which filesystem is newest or most discussed

- Advanced features chosen but never used add complexity with no corresponding benefit

CHAPTER 11 OF 11

Capstone: Building a Real Storage Stack

Linux Filesystems

Chapter 11 · Capstone — Building a Real Storage Stack

Ten chapters of concepts and commands, now combined into one real build: a redundant RAID 10 array, LVM on top of it, ext4 formatted and mounted, a live snapshot taken, and then a deliberate, simulated disk failure and replacement — proving the theory from every prior chapter for real, in order.

The Plan

Four disks (`/dev/sdb`, `/dev/sdc`, `/dev/sdd`, `/dev/sde`), assembled into RAID 10 — chosen per `fs1-7` / `fs1-10`'s own recommendation for fast, low-risk rebuilds and predictable performance. LVM sits on top for live resize flexibility (`fs1-5` / `fs1-9`'s own traditional-stack pattern), formatted with ext4 (`fs1-2`, validated by `fs1-10`'s decision framework for this kind of general-purpose workload). `/boot` itself would live on its own small, plain partition outside this array entirely, per `fs1-5`'s own GRUB-compatibility guidance — not built here in detail, since partitioning and GRUB itself were already covered in full in `grub1`.

Step 1 — Building the RAID 10 Array

```
mdadm --create /dev/md0 --level=10 --raid-devices=4 /dev/sdb1 /dev/sdc1 /dev/sdd1 /dev/sde1
mdadm --detail --scan >> /etc/mdadm/mdadm.conf
update-initramfs -u
cat /proc/mdstat                    # confirm [UUUU] — all four members healthy
```

Step 2 — LVM on Top of the Array

```
pvcreate /dev/md0
vgcreate datavg /dev/md0
lvcreate -L 50G -n datalv datavg
mkfs.ext4 /dev/datavg/datalv
mount /dev/datavg/datalv /mnt/data
```

Step 3 — Taking a Live Snapshot

```
lvcreate -L 5G -s -n datalv-snap /dev/datavg/datalv
lvs                                  # confirm the snapshot exists, check its usage %
```

Directly reusing `fs1-6`'s own real-world pattern — a live snapshot taken immediately before doing something deliberately risky, protecting the current, known-good state before the next step intentionally breaks something.

Step 4 — Simulating a Disk Failure

```
mdadm /dev/md0 --fail /dev/sdc1
cat /proc/mdstat                     # shows a degraded array
```

Because this is RAID 10, data stays fully available right through the failure — the mirrored partner of the failed disk is still up. This is `fs1-7`'s own RAID 10 claim ("survives multiple disk failures, as long as no single mirrored pair loses both of its own disks") demonstrated concretely rather than left theoretical.

```
mdadm /dev/md0 --remove /dev/sdc1
# -- physically replace the disk --
mdadm /dev/md0 --add /dev/sdc1
cat /proc/mdstat                     # shows recovering, a percentage climbing quickly
```

The rebuild here is fast and comparatively low-risk specifically because RAID 10 recovery means re-mirroring from the surviving partner disk — not RAID 5/6's own much slower whole-array parity reconstruction. This is `fs1-7`'s own rebuild-risk comparison, made concrete rather than abstract.

Step 5 — Verifying Recovery

```
mdadm --detail /dev/md0             # confirm [UUUU] again, array fully healthy
lvs                                  # confirm the snapshot is untouched, still holds the pre-failure
```

Worth being explicit about a real limitation of this exact stack: nothing in it — not ext4, not LVM, not `mdadm` — ever actually checksums the data content itself, per `fs1-9`'s own honest layering discussion. This build trusts that the RAID rebuild reconstructed the data correctly; it has no independent way to verify that at the content level, the way a Btrfs- or ZFS-based capstone (per `fs1-3` / `fs1-4`) would have gotten for free. That's

a deliberate, informed tradeoff here, following `fs1-10`'s own framework for this kind of workload — not an oversight.

Chapter Attribution Table

PIECE OF THE BUILD	CHAPTER
RAID 10 concepts (why this level)	fs1-7
mdadm array creation, monitoring, failure/rebuild	fs1-8
LVM layer (PV/VG/LV)	fs1-5
ext4 formatting choice	fs1-2, fs1-10
Live snapshot before a risky step	fs1-6
Traditional vs. integrated layering decision	fs1-9
Overall stack choice justification	fs1-10
/boot placement note	fs1-5, grub1-8

STILL OUT OF SCOPE

No checksumming or self-healing exists anywhere in this stack — a deliberate choice per `fs1-10`'s framework for this workload, but a real limitation if bit-rot protection ever became an actual requirement, at which point Btrfs or ZFS (`fs1-3` / `fs1-4`) would be the correct answer instead. The snapshot taken in Step 3 is **not** a backup, per `fs1-3`'s and `fs1-6`'s own explicit warnings — it lives on the same array it's protecting. No `/boot` partition or GRUB configuration was actually built here (that's `grub1`'s own dedicated territory). And this was a single-node build — network storage, iSCSI, and clustering are not covered anywhere in this course.

CHECK STATUS AFTER EVERY STEP, NOT AT THE END

Run each command's own status check (`cat /proc/mdstat`, `lvs`) immediately after the step that could have gone wrong, not after batching several steps together. Catching a problem right after the step that caused it is dramatically easier than untangling it after several more steps have piled on top.

Hands-On Exercises

EXERCISE 1

Rebuild this capstone's own plan, but for a scenario where bit-rot protection genuinely is a hard requirement. Identify exactly which two chapters' worth of changes would be needed, and what would be dropped from this chapter's own stack as a result.

View solution

EXERCISE 2

Explain why the RAID 10 rebuild in Step 4 was low-risk enough to not require the LVM snapshot from Step 3 for protection, while a hypothetical RAID 5 rebuild in the same scenario would make that snapshot's timing genuinely important.

View solution

EXERCISE 3

A reader finishes this capstone and concludes "great, my data is now fully protected." Using this chapter's own warn-box, identify what's still missing from that conclusion.

View solution

CHAPTER 11 QUICK REFERENCE — THE FULL BUILD

- 1. `mdadm --create` — RAID 10 across four disks (fs1-7, fs1-8)
- 2. `pvcreate` / `vgcreate` / `lvcreate` / `mkfs.ext4` — LVM + ext4 on top (fs1-5, fs1-2)
- 3. `lvcreate -s` — a live snapshot before the risky step (fs1-6)
- 4. `mdadm --fail` / `--remove` / `--add` — simulate and recover from a disk failure (fs1-8)
- 5. `mdadm --detail` / `lvs` — verify full recovery
- This stack has no content checksumming anywhere — a deliberate tradeoff, not an oversight
- The snapshot is not a backup — same array, same physical risk