What could go wrong?

Expectation: stored data is persistent and correct.

1. **Device failure**:  
   - disk crash (permanent failure)  
   - bit flips on storage medium (*What about host memory?*)  
   - transient or permanent sector read errors

2. **OS crash or power failure** during filesystem manipulation

3. **Accidental data deletion/corruption** by users.

4. **Malicious tampering** by attacker.
Last Line of Defense: Backups!

Good *regular* backups can help with all of these issues.

- Once a day or more frequently to limit data loss.
- Need a history of backups, not just latest snapshot (→ bit errors, human error, attacks).
- Backups should not be reachable from host, even if fully compromised (→ attacks).
- Downside: restoring from backup can be very slow.
Dealing with Human Error

*Accidental data deletion or corruption due to configuration errors or software bugs.*

- **Snapshotting** filesystem: the filesystem takes a (readonly) "snapshot" at regular intervals (e.g., every 24h).
  - copy-on-write makes this relatively cheap
  - Examples: ZFS, btrfs (Linux), HAMMER (DragonflyBSD)

- **Versioning** filesystem: *every* file version is retained for some time (e.g., last 30 days)
  - Similar to Dropbox, but part of the low-level FS (→ efficiency)
  - Example: HAMMER retains a version every 30–60 seconds on sync
Dealing with Device Failures (1/3)

Partial failures: **bit rot** (= bit flips), **bad sectors**, and **transient read errors**.

» Bit rot: aging effects and electro-magnetic interference (EMI) can corrupt data on disk
   → **silent** read errors

» Individual sectors of a disk can fail
   → **explicit** read errors

» **Detection**: associate **checksum** with each block

» **Mitigation**: error-correcting codes, redundant blocks
Dealing with Device Failures (2/3)

Total device failures: disk crashes, controller failures,…

- **Mirroring**: store every block on multiple disks

  - **Advantages**:
    - very effective: works as long as at least one disk survives
    - reads can be faster than on single disk because parallel reads can be dispatched to different (or multiple) mirror disks

  - **Disadvantages**:
    - capacity exposed to FS limited to smallest drive
    - expensive
    - *synchronous* writes can be slower than on single disk because *all* disks must finish write
Dealing with Device Failures (3/3)

Can we do better than mirroring?

» **RAID**: Redundant Array of Independent Disks
  → originally: *Inexpensive* disks (Patterson et al., 1988)

» Goal: combine many *not so fast, not so reliable* disks into one logical volume that is *faster* and/or more *reliable*.

» Many different **RAID levels** exist can be nested and combined

» Standard levels: 0–6

» many vendor-specific variants exist
RAID 0 — Striping

**Idea**: distribute writes across all disks simultaneously

» with \( d \) disks, write block \( n \) to disk \( n \mod d \)

» This makes the disk array **less reliable**: data loss if any disk fails

» But the array is (up to) \( d \) times **faster than a single disk**
  » logically sequential write or read of \( d+ \) blocks = parallel write/read
  » random reads/writes likely go to different disks

» **Full capacity** of all disks available
RAID 5 — Block-level Parity

**Idea:** use *parity bits* to recover lost blocks

- With *d* disks, for every *d - 1* blocks, write one *parity block*.
- Distribute parity blocks across all disks
  - *Why?*
- Can tolerate loss of any one disk
  - *Replace and rebuild array* before next one fails
- Reads: almost as fast as RAID 0 (parallelized)
- Writes: faster than a single drive, but not as fast as RAID 0
- Capacity: \((d-1)/d\) of total disk space available
Other RAID Levels

» RAID 1: just another name for mirroring
  » can be combined to form RAID 1+0
    ➞ striped across mirrored disks

» RAID 2: stripe at byte level with error-correcting code

» RAID 3: stripe at byte level with dedicated parity disk

» RAID 4: stripe at block level with dedicated parity disk

» RAID 6: like RAID 5, but with two (different) parity blocks for every $d - 2$ blocks
  ➞ can tolerate two disk crashes
  ➞ Why is this becoming more important?
Dealing with Crashes

What if the OS crashes / the system loses power in the middle of a filesystem update?

How do we achieve crash consistency?

1. Run a tool to check for and repair inconsistencies on next boot (→ fsck)
2. Keep a log of ongoing operations (→ journaling)
3. Order all disk writes such that version on disk is always consistent (→ soft updates)
fsck — filesystem check

After a crash, run a tool to repair the filesystem.

- **Approach**: read entire filesystem, find all inconsistencies, guess correct state and fixup

- **Limitations**: cannot detect and/or fix all inconsistencies

- **Inefficient**: very, very slow on large disks

- With large RAIDs, fsck run can easily take more than 24h...
Write-Ahead Logging / Journaling

**Idea:** keep a log of ongoing operations.

- Special area on disk (or second disk/SSD) that holds records describing in-flight operations.
- Write-ahead logging:
  1. **journal**: write record in log (blocks to write)
  2. **journal**: write completion record
     - *How to combine this step with the first write?*
  3. **checkpoint**: perform updates in place
- After a crash, **replay** completed operations.
- Data journaling vs. meta-data journaling