File Systems 1

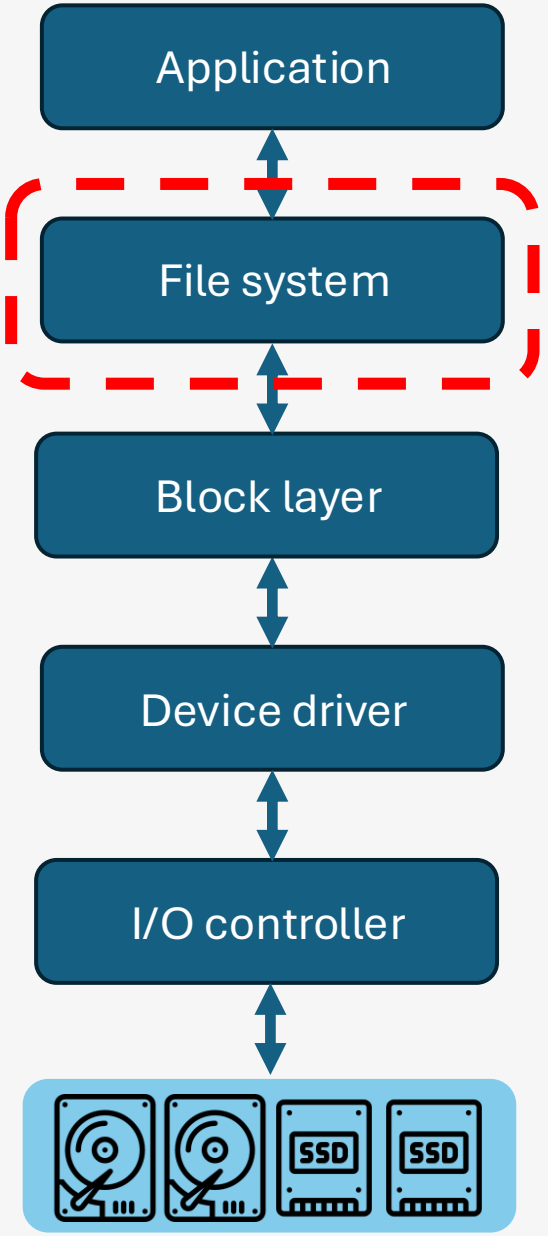
Juncheng Yang

Harvard John A. Paulson
School of Engineering
and Applied Sciences

HARVARD UNIVERSITY
MADSYS
Measurement and Design of Systems Lab

Agenda

- **Overview and user view**
- **Interface and layers**
- **Design and implementation**
- Performance, efficiency and integrity
- Case study
  - FFS
  - LFS
  - F2FS
  - Btrfs



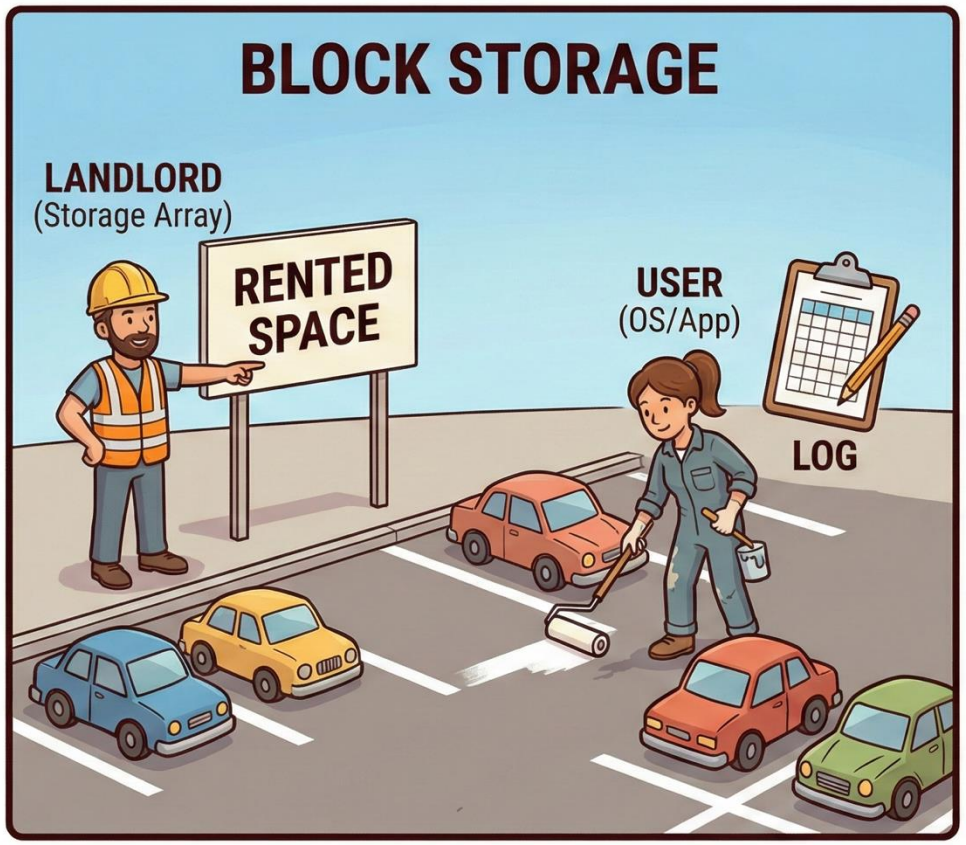
Four Questions To Ask Yourself After This Class

- What are the responsibilities of a file system? Why do we need it?
- What are the layers and core data structures in VFS?
- What are the common on-disk file system data structures? What are their purposes?
- Can you describe steps a file system performs when you call `open()`, `read()` and `write()`

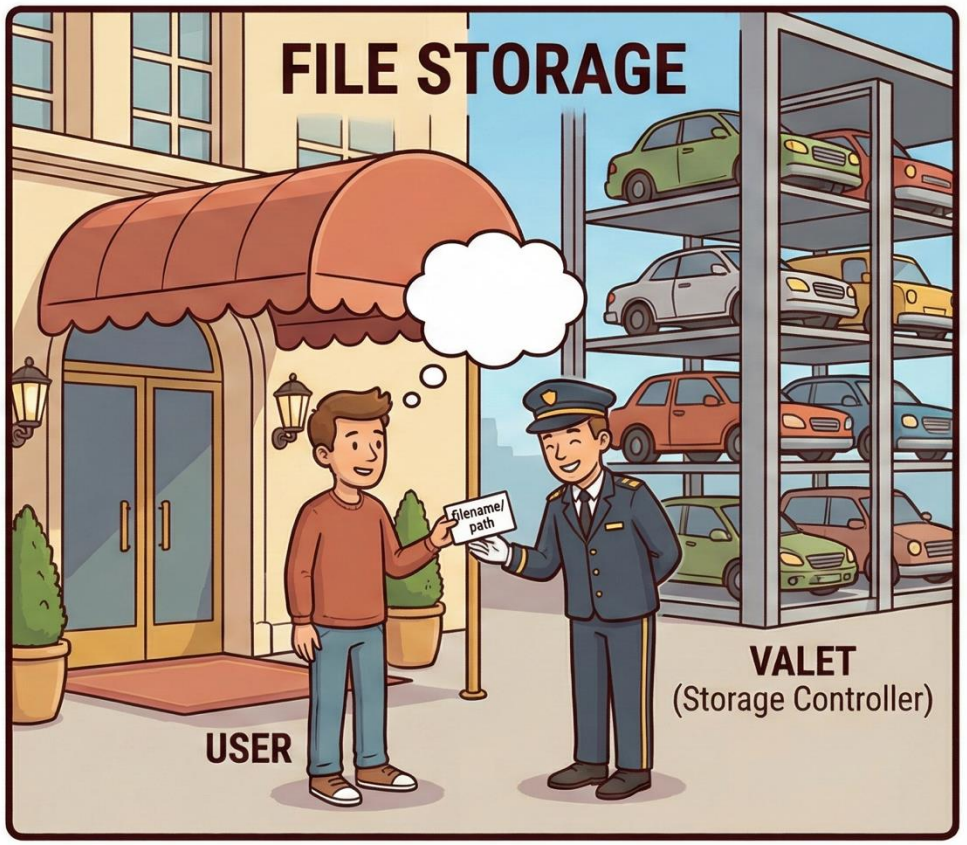
Block vs. File Storage

- **Block storage**
  - expose a device as a **linear array of fixed-size**
  - read/write by **(block address, length)**
  - user manages data organization
  - example: HDD, SSD and virtual devices
- **File storage**
  - expose data as **named files in directories** with operations like open/read/write/close, plus permissions and metadata
  - read/write by **(file path/descriptor, offset, length)**
  - example: file system

Block vs. File Storage

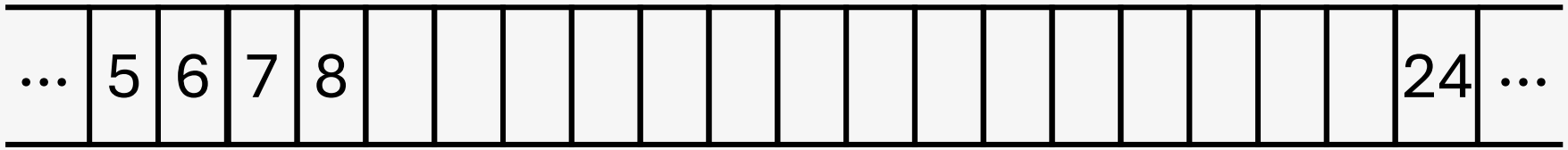


Block Storage is like renting a parking lot. The landlord just gives you a defined space. You decide how to paint the lines, where to park the cars, and you keep your own log of who is parked where.



File Storage is like a valet service. You don't know or care where the car is paris parked. You just hand over your ticket (filename/path), and the retrieves your car for you.

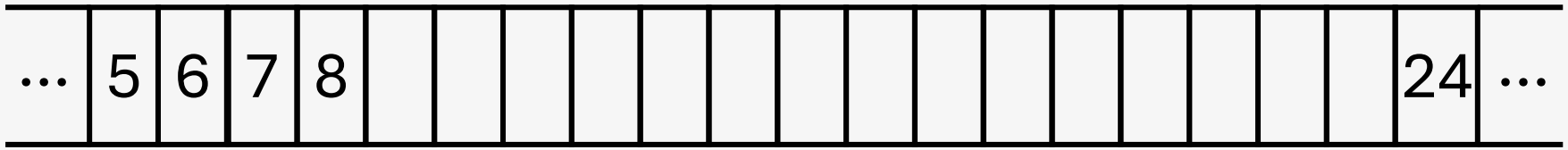
Data organization and naming



OS view of disk

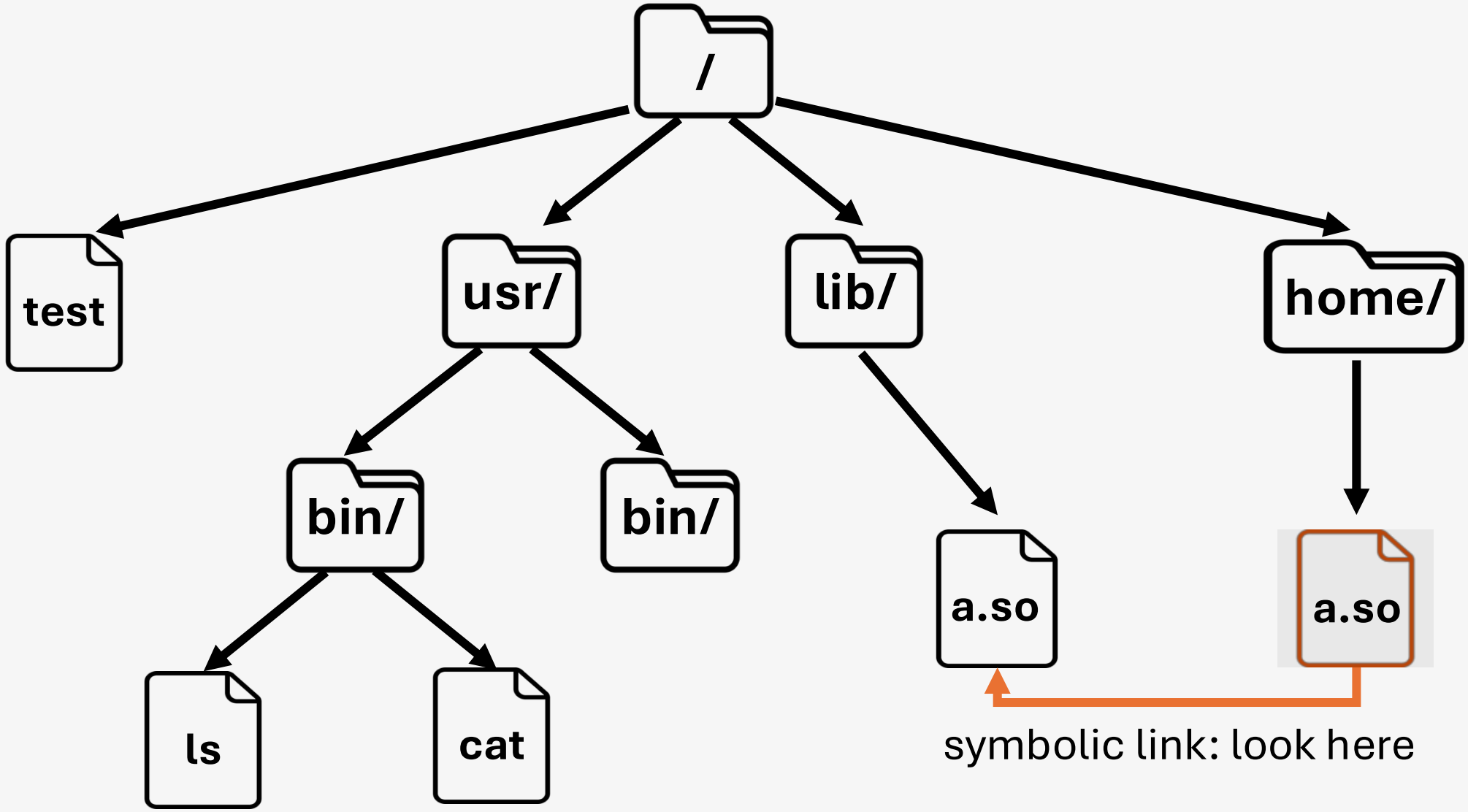
- How do we organize data?
  - how can you make sure that you can find what you want later?
  - not nearly as easy as it seems, e.g., where is your calculus 1 exam paper?
  - How do you find free space?
- This is changing with LLM
  - increasingly natural language is used to find information
  - research: how should we organize data in the LLM era?

Data organization and naming: possible solutions

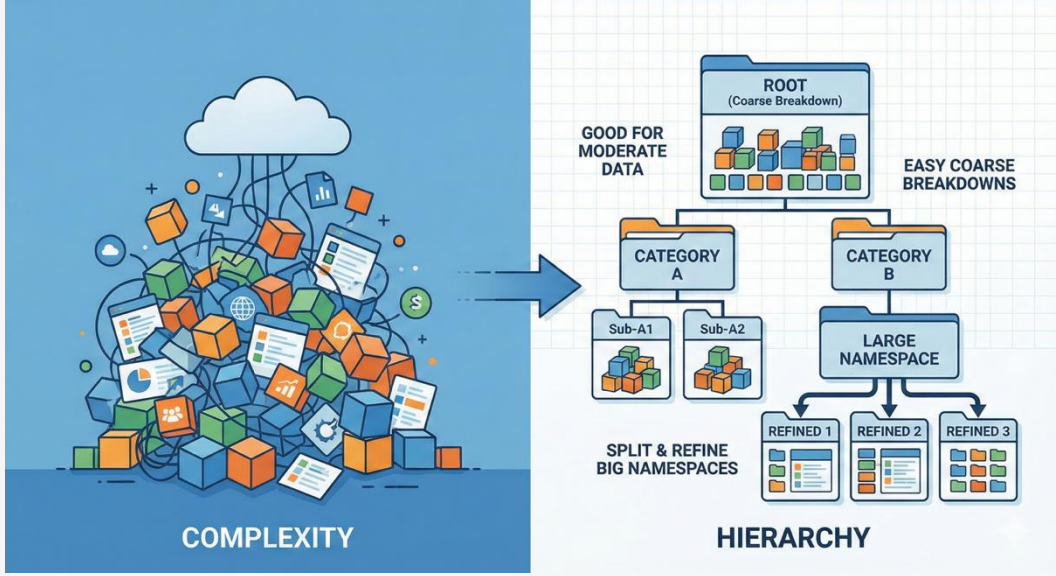


OS view of disk

- Flat namespace
  - early systems
  - problem: naming collisions
- Hierarchical namespace (tree)
  - modern standard
  - files are leaves and directories are internal nodes

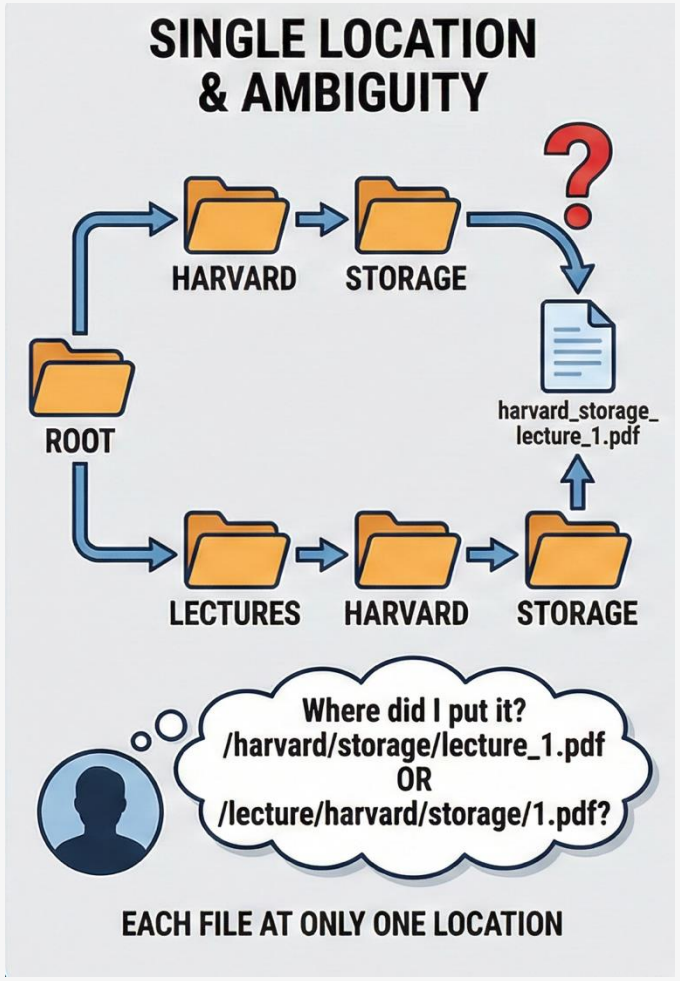

/
test
usr/
lib/
home/
bin/
bin/
a.so
a.so
ls
cat
symbolic link: look here

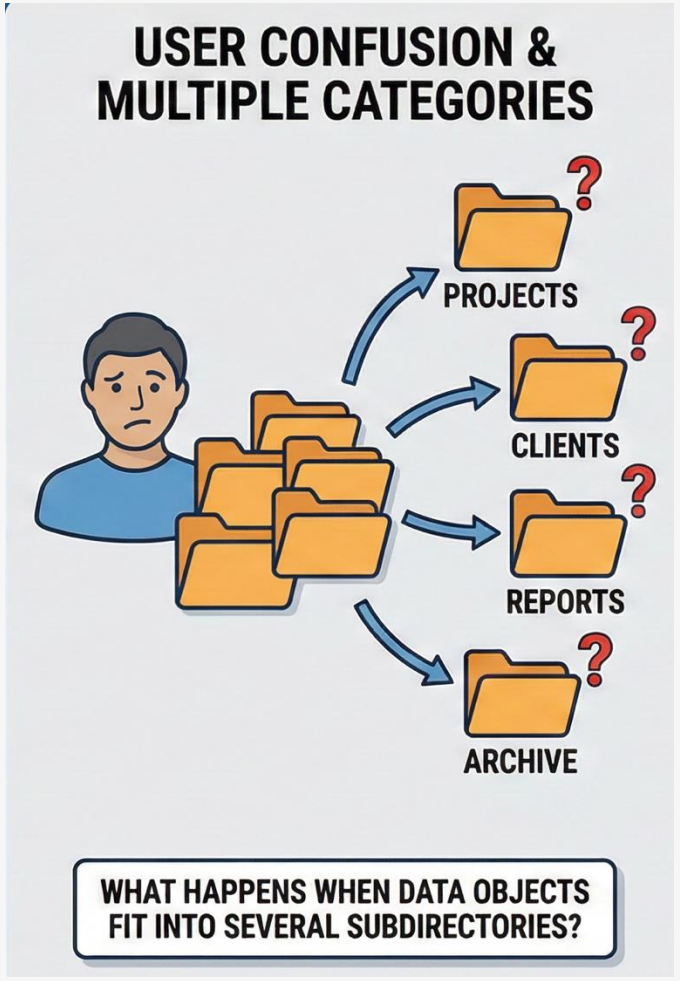
Directory and File Structure

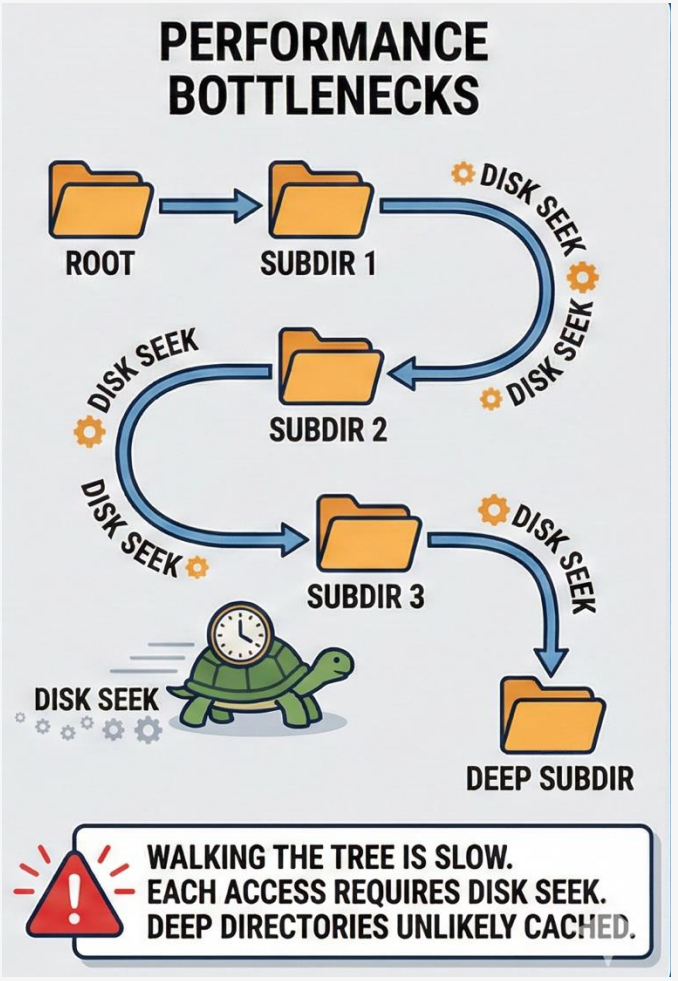


- Hierarchies help manage complexity
  - mimics human brain categorization
- Benefits
  - **cognitive load management**: more human-readable, easy for search (early pruning)
  - **structure:** embed information in the path
  - **scalability via splitting**: split a directory when it gets too big

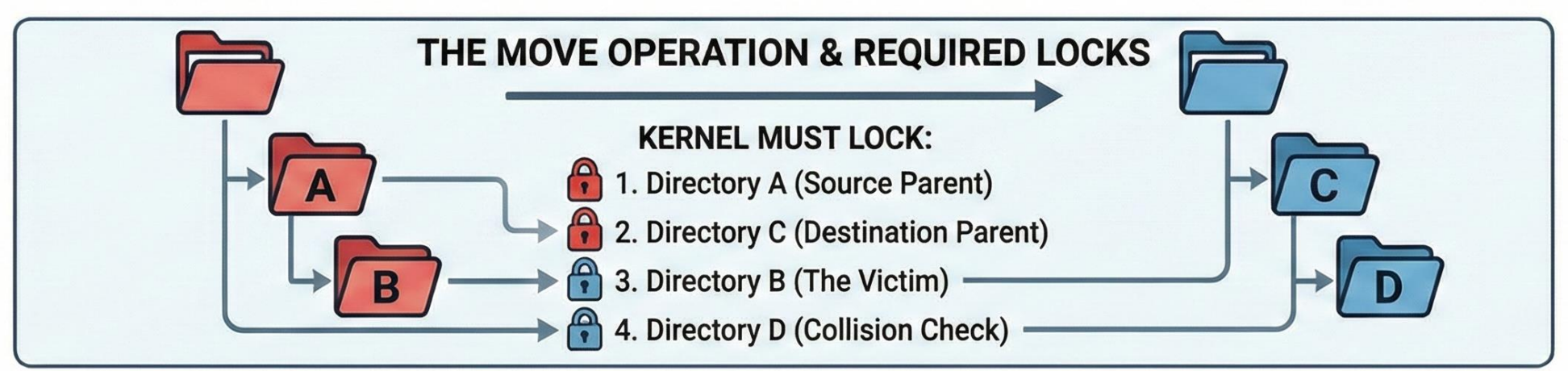
Directory and File Structure: Challenges

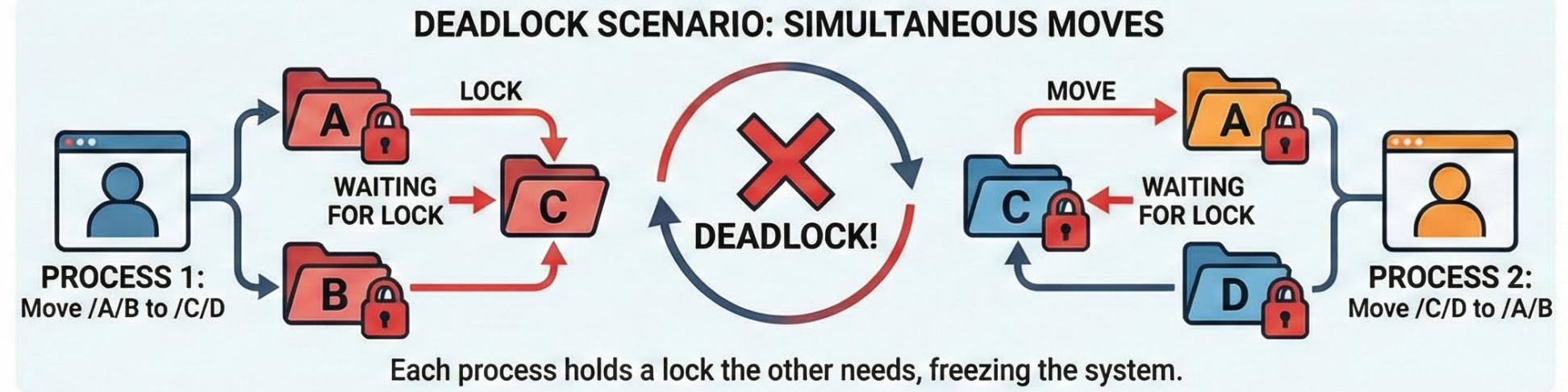






Directory and File Structure: Challenges

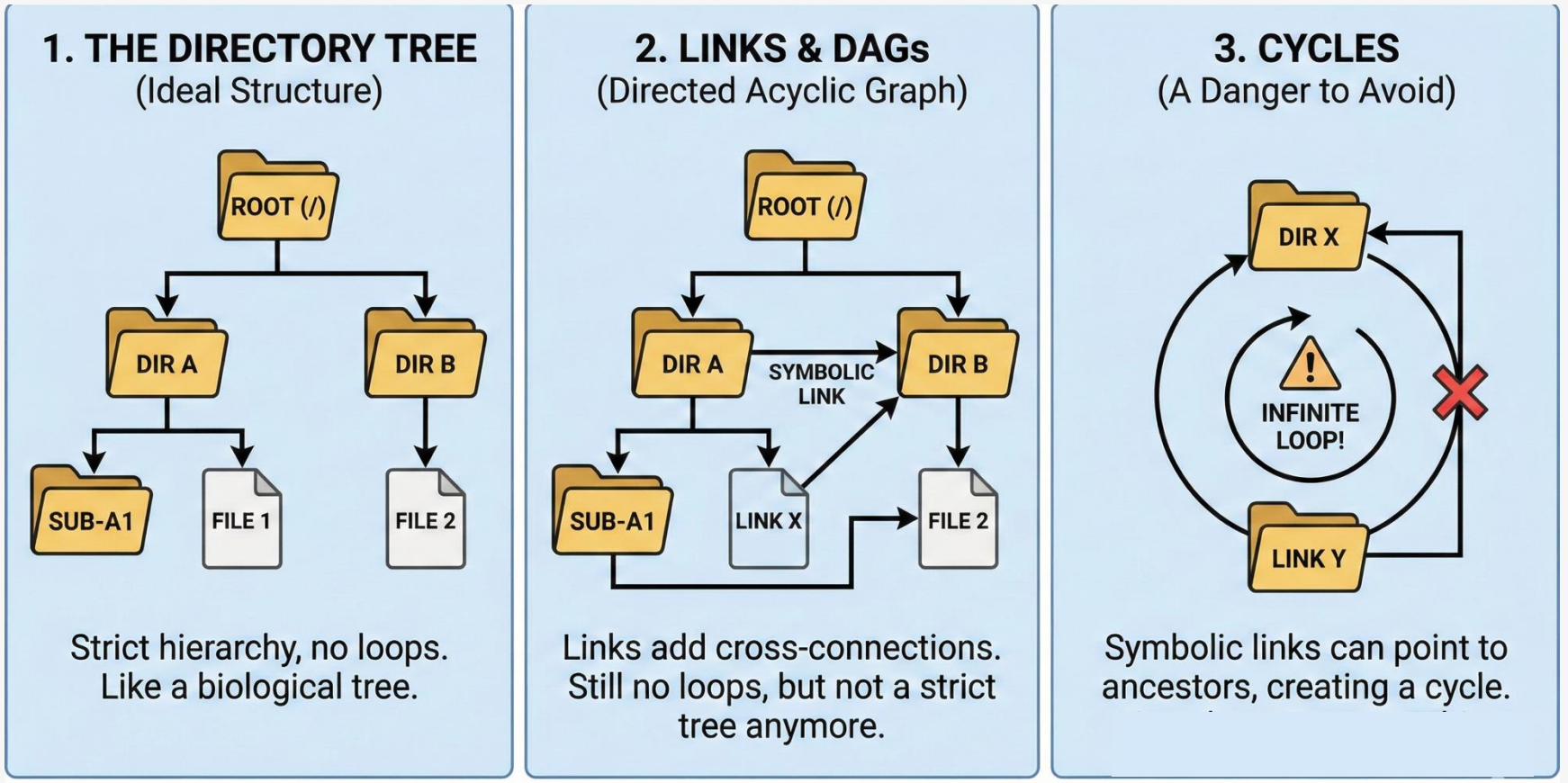




Move requires multiple locks that can easily lead to deadlock

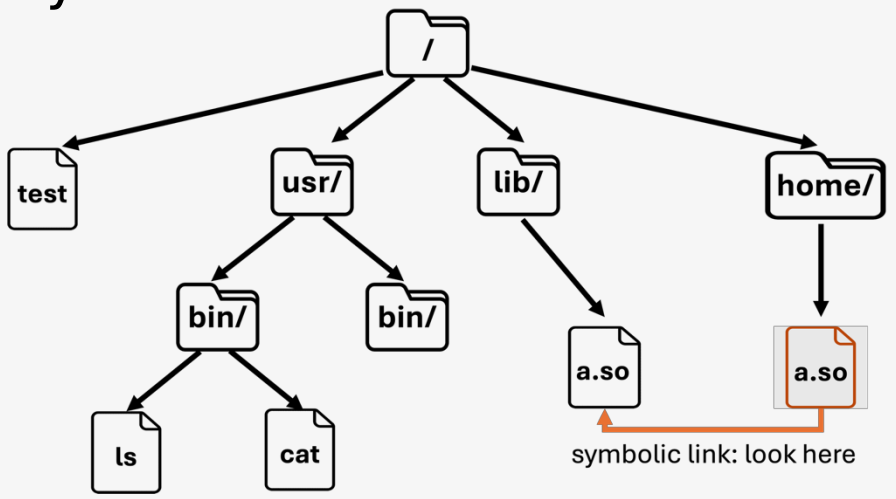
Directory and File Structure: Tree and DAG

- Links turn the hierarchy into a Directed Acyclic Graph (DAG)
- Cycles over hard links are usually prevented by the kernel to avoid infinite loops, symbolic links can still create cycles

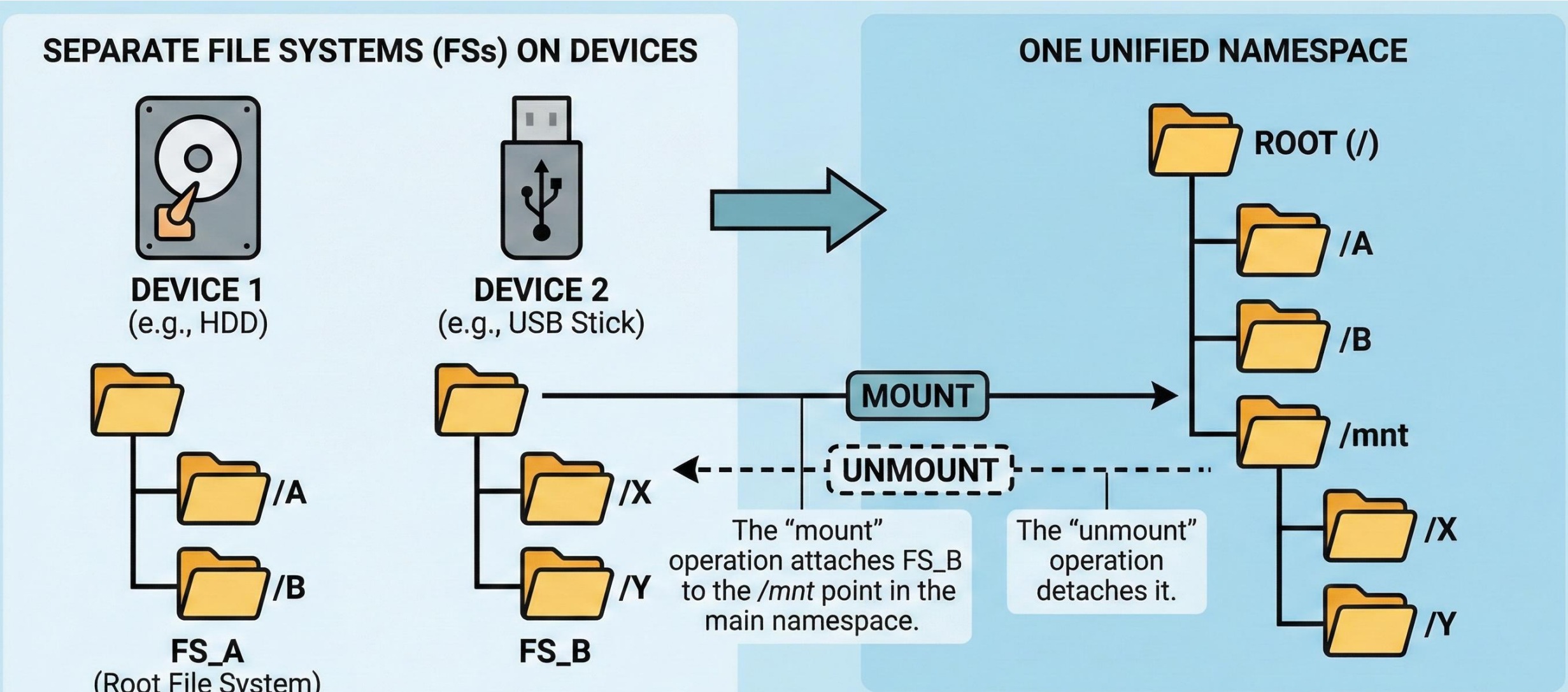


Managing namespace: mount/unmount

- We can have multiple FS on one or more devices, but only **one namespace**
  - must combine the FSs into one namespace
  - starts with a “root file system”
  - “mount” operation attaches one FS into the namespace
  - “unmount” detaches a previously-attached file system



Managing namespace: mount/unmount

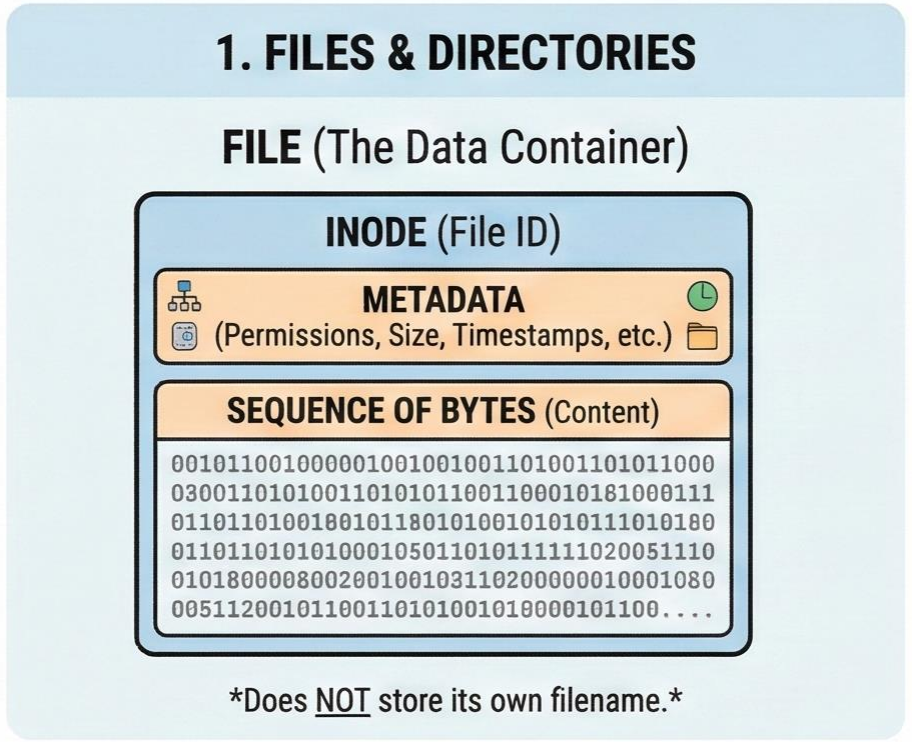


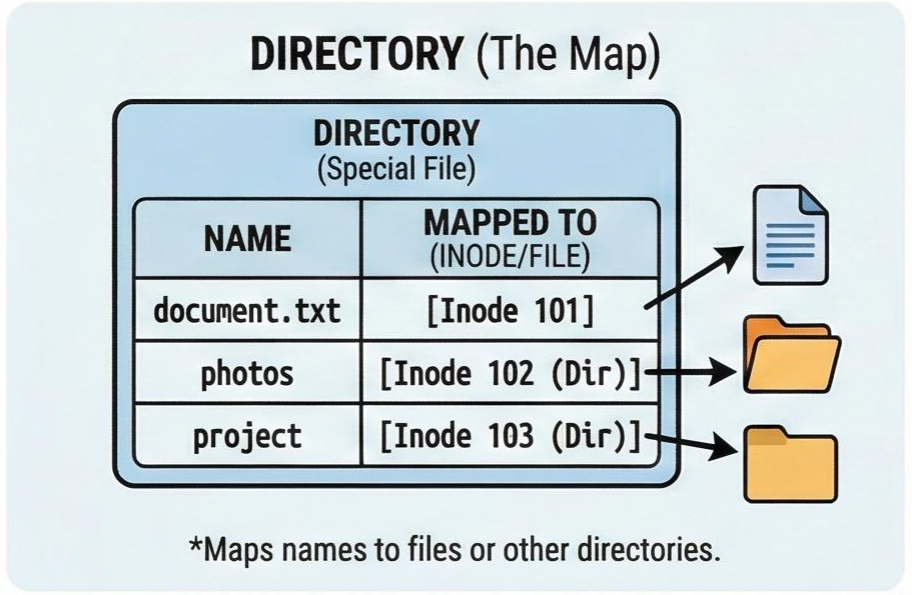
Abstractions

- **File**: metadata (inode) and a sequence of bytes, *it does not store filename*
- **Directory**: a special file that maps from names to files or directories
- **Hard Link:** a second name for the *same* file (inode).
  - delete the original file, the data remains
  - constraint: within partitions (inode numbers are only unique *within* a partition)
- **Soft Link (Symbolic Link):** a tiny file containing a *path string*
  - e.g., ”/harvard/cs2640/slides/2_hdd.pdf"
  - point to a name, not a file (inode), delete the original file leads to dangling link
  - benefit: can cross partitions and even point to network locations

Abstractions

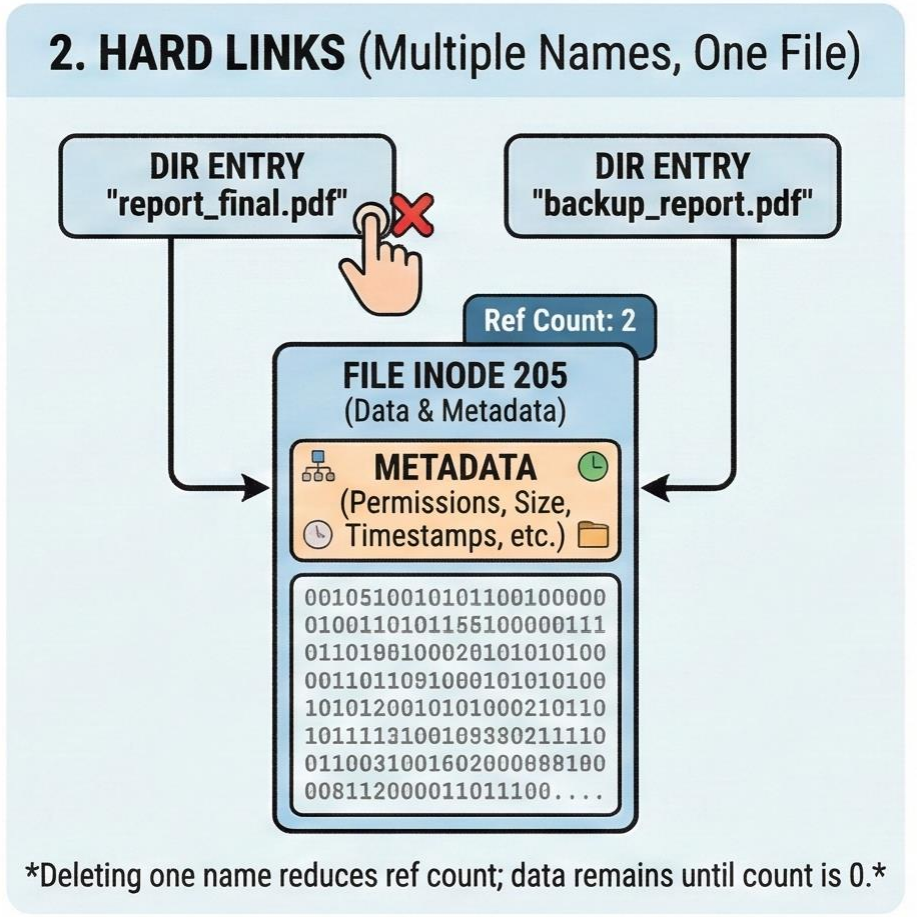
- **File**
  - metadata (inode) and a sequence of bytes, *it does not store filename*
- **Directory**
  - a special file that maps from names to files or directories

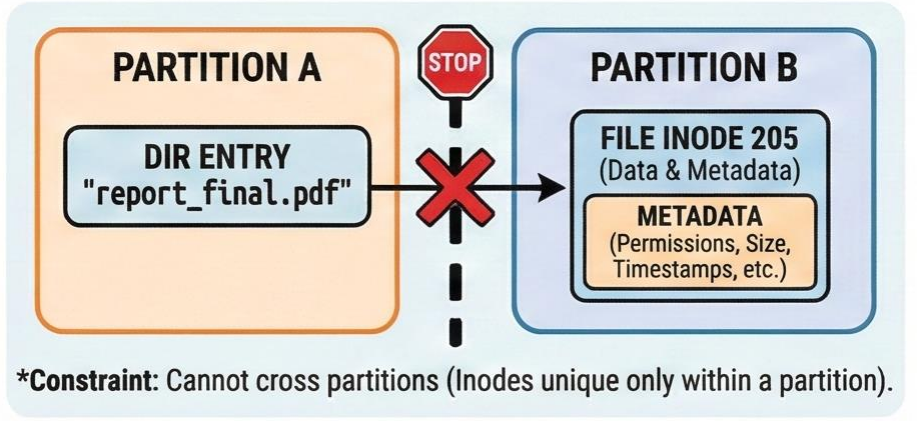




Abstractions: Hard Link

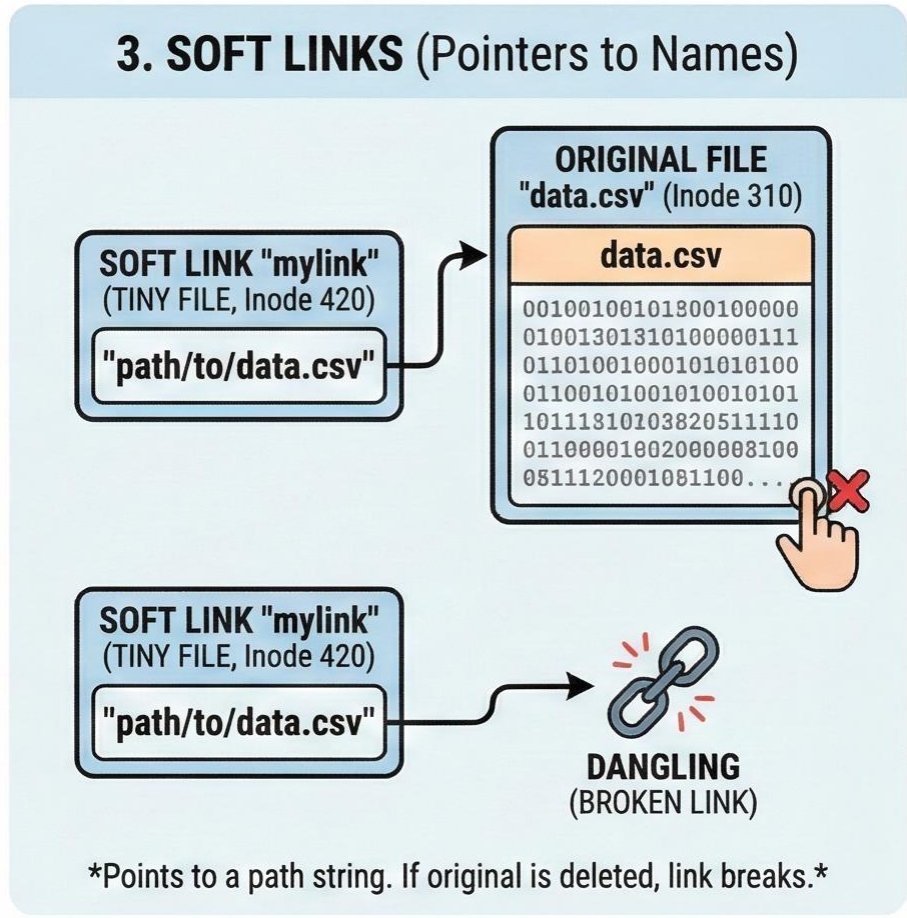
- A second name for the *same* file (inode)
  - delete the original file, the data remains
  - constraint: within partitions (inode numbers are only unique *within* a partition)

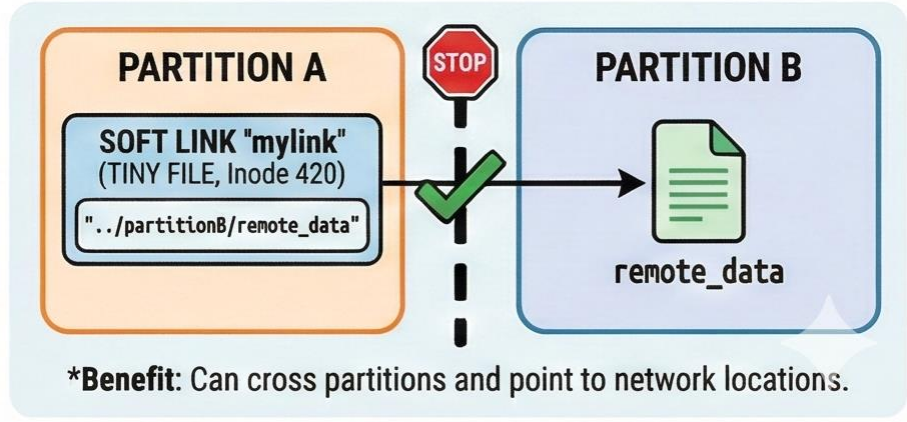




Abstractions: Soft Link

- A tiny file containing a *path string*
  - point to a name, not a file (inode), delete the original file leads to dangling link
  - benefit: can cross partitions and even point to network locations

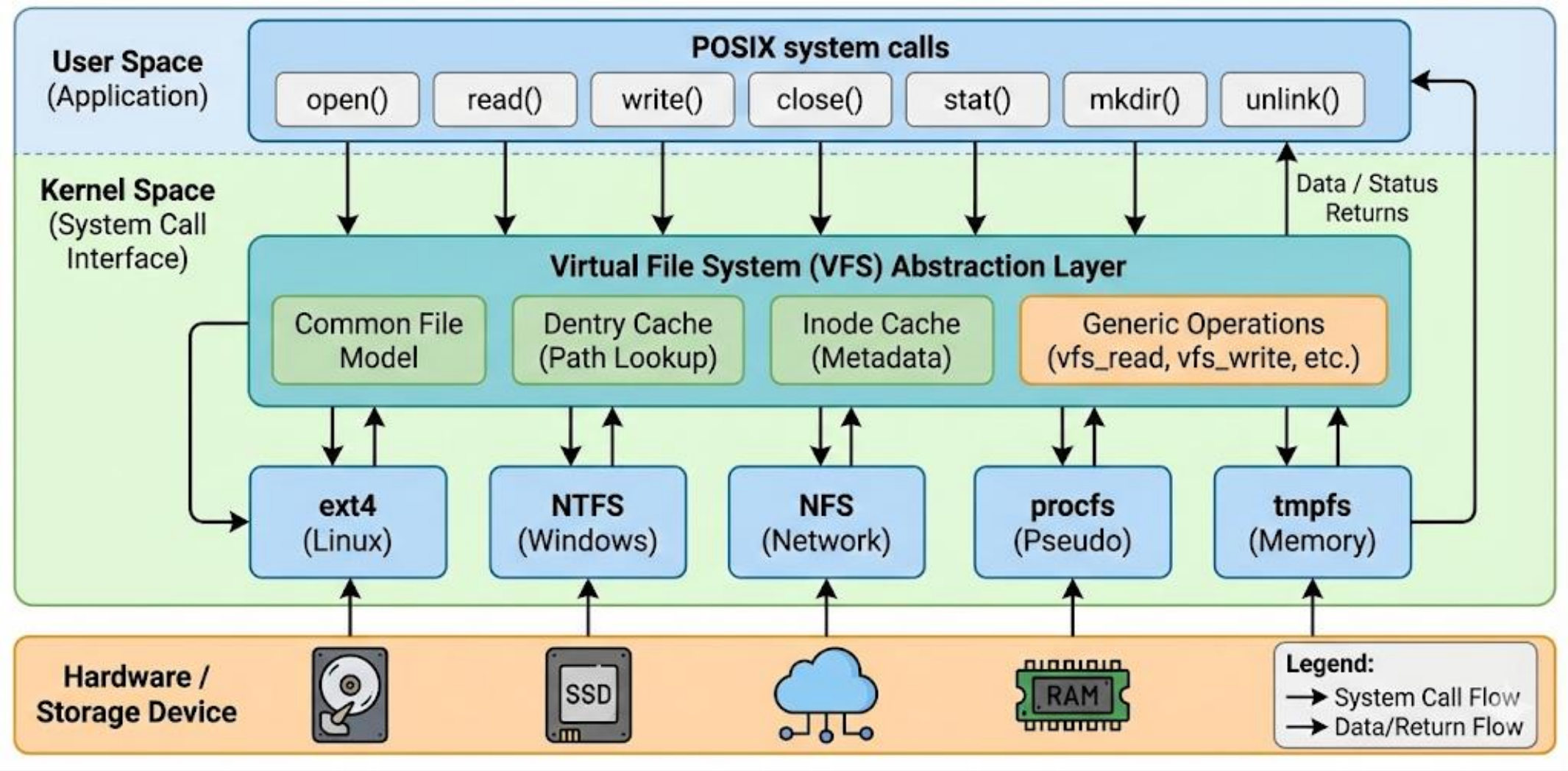




Interface

- POSIX API
- Virtual File System (VFS)

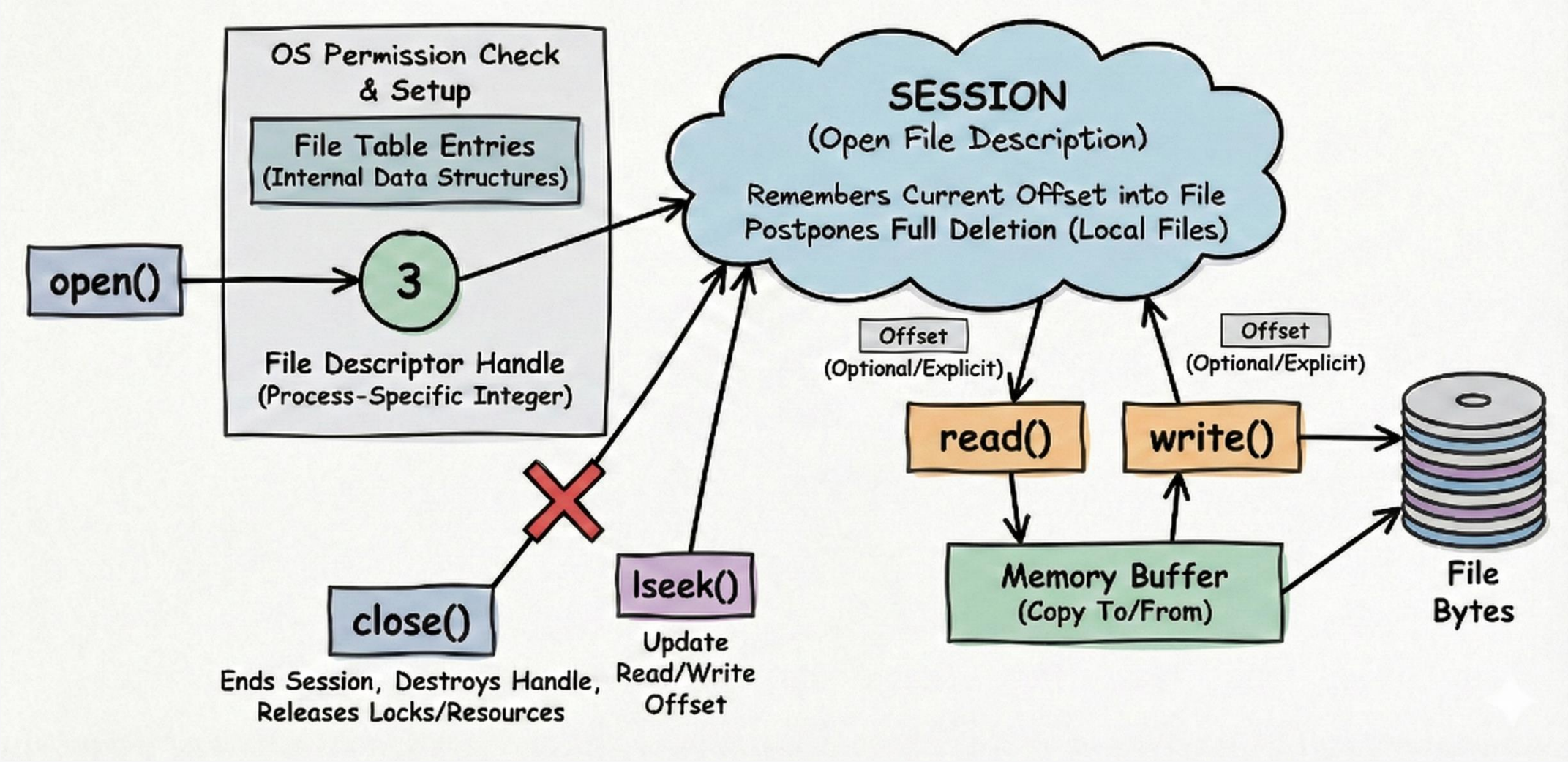
Overview


User Space
(Application)
POSIX system calls
open()
read()
write()
close()
stat()
mkdir()
unlink()
Kernel Space
(System Call
Interface)
Data / Status
Returns
Virtual File System (VFS) Abstraction Layer
Common File
Model
Dentry Cache
(Path Lookup)
Inode Cache
(Metadata)
Generic Operations
(vfs_read, vfs_write, etc.)
ext4
(Linux)
NTFS
(Windows)
NFS
(Network)
procfs
(Pseudo)
tmpfs
(Memory)
Hardware /
Storage Device
SSD
RAM
Legend:
System Call Flow
Data/Return Flow

POSIX APIs (file systems)

- File I/Os
  - `open()`, `read()`, `write()`, `close()`, `lseek()`, `fsync()`, `unlink()`, `truncate()`, `rename()`, `link()`, `readlink()`
  - many others, `_at` family, vector variants, soft/hard link operations

File I/O


OS Permission Check
& Setup
File Table Entries
(Internal Data Structures)
open()
3
File Descriptor Handle
(Process-Specific Integer)
SESSION
(Open File Description)
Remembers Current Offset into File
Postpones Full Deletion (Local Files)
Offset
(Optional/Explicit)
Offset
(Optional/Explicit)
read()
write()
close()
Ends Session, Destroys Handle,
Releases Locks/Resources
lseek()
Update
Read/Write
Offset
Memory Buffer
(Copy To/From)
File
Bytes

POSIX APIs (file systems)

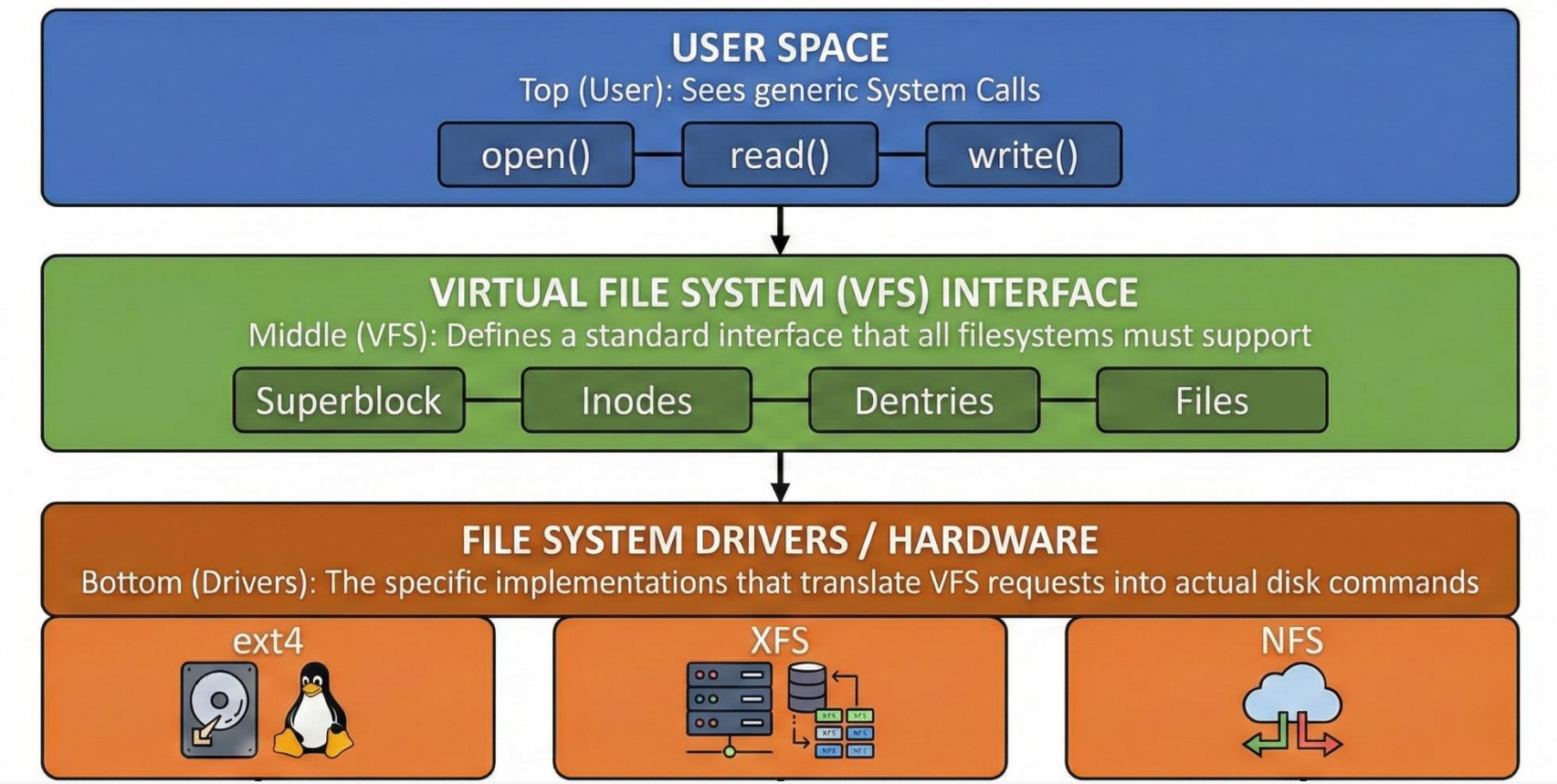
- File I/Os
  - `open(), read(), write(), close(), lseek(), fsync(), unlink(), truncate(), rename(), link(), readlink()`
  - many others, `_at` family, vector variants, soft/hard link operations
- File management
  - `creat(), unlink(), truncate(), rename()`
- Directory operations
  - `opendir(), readdir(), closedir()`
  - `chdir(), getcwd()`

POSIX APIs (file systems)

- Metadata operations
  - `stat(), chown(), utime(), umask()`
- Others
  - locking: `flock(), fcntl()`
  - memory-mapped I/O: `mmap(), munmap(), msync(), madvise()`
    - VM APIs, but uses page cache (read and write)
  - special file creation: `mkfifo()`

Virtual File System (VFS): Three Layers

- Translation layer between user interface and different file systems



Why VFS

- **Unified namespace and abstraction**
  - mount a network drive (NFS) at `/mnt/data`, applications read/write to it just like a local disk
- **Pseudo-Filesystems**
  - VFS enables for filesystems that don't exist on disk at all
  - **/proc:** a virtual filesystem
    - reading `/proc/cpuinfo` doesn't read a disk
    - the VFS generates that text on the fly from kernel variables
  - **/sys:** exposes hardware and driver info

Virtual File System (VFS): Four Pillars

- **Superblock**: global metadata of one mounted file system
- **Inode** (also called vnode):
  - holds metadata (permissions, size, timestamps, block locations)
- **Dentry**: a directory entry
  - the link between a name and an inode
- **File**: an open file instance
  - tracks the state of the open file instance (offset and the mode)
- These are all in-memory, superblock and inode have on-disk components with the same names, but they are different

Virtual File System (VFS): Superblock

- Represents: a *mounted filesystem instance*
- One superblock per filesystem created during mount
- Contains filesystem-wide metadata and state:
  - FS type/ops (ext4, xfs, nfs) and pointers to FS-specific structures
  - block size, limits, feature flags
  - the root dentry of that mount
  - lists/caches of inodes
  - …

Virtual File System (VFS): Inode

- Represents**:** a *file object on disk*
- Describes everything about a file ***except*** its name and its actual data content
  - file type (regular, dir, symlink…), file size
  - permissions, owner, timestamps, link count
  - pointers to page cache
- Every file has exactly one inode
  - inode is identified by a unique **inode number** within that filesystem

Virtual File System (VFS): Dentry

- Represents a specific *component* of a path
  - the "glue" that links a human-readable string to inode
- VFS uses dentries heavily for pathname resolution (/a/b/c) and caches them in the dentry cache (dcache)
- Can exist in multiple states:
  - positive dentry: name resolved →
  - negative dentry: name lookup fail
- Multiple dentries can re
  - hard links: different nar

Why don’t we use the full path?

- **Mount point**: /mnt/disk can points to different disks at different time
- **Permission**: you may have access to /a/b but not /a

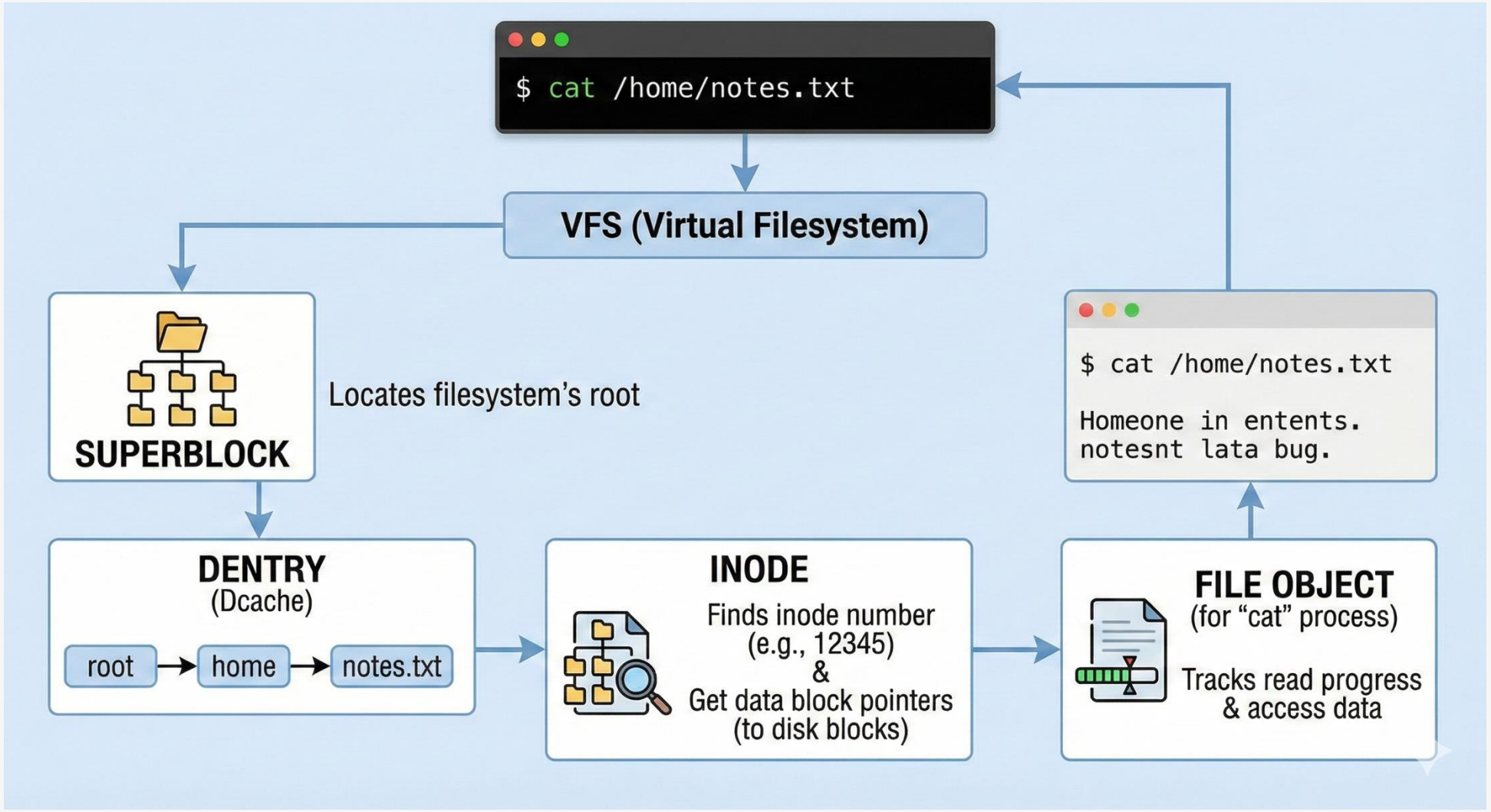
Virtual File System (VFS): File

- Represents: an *open file description*
  - created when you open()
  - destroyed when the last reference is closed
- Holds:
  - current file position (offset)
  - open flags (O_RDONLY, O_APPEND, etc.)
  - pointer to the dentry and inode
- Multiple file objects can refer to the same inode:
  - each `open()` creates one
  - `dup()` shares the same struct

Virtual File System (VFS): Four Pillars

Object	Represent	Key Data	Primary Purpose
Superblock	Entire Filesystem	FS Type, block size	Overall FS metadata and management
Inode	Single File	Permissions, size, page list	Metadata (size, owner) and data location
Dentry	Path Component	Filename -> Inode Pointer	Mapping a name to an inode
File	Open Instance	Cursor position, access mode	Tracking process-specific state (offset)

Virtual File System (VFS): Four Pillars



File System Implementations

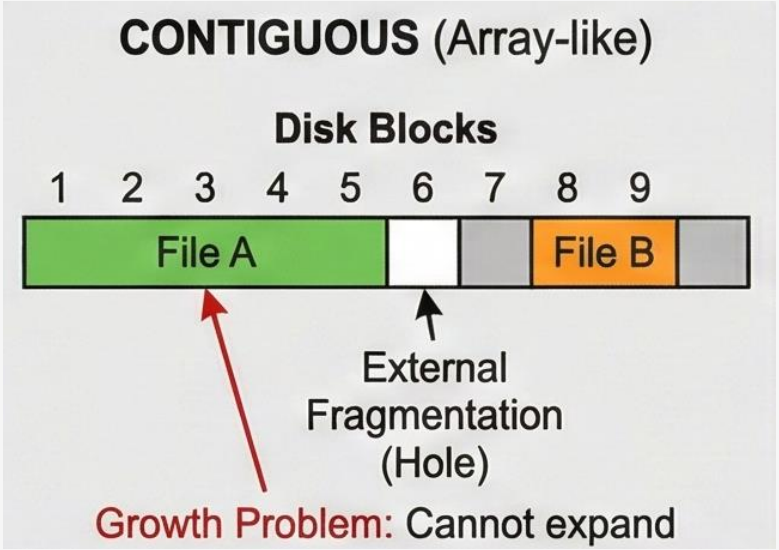
- Responsibilities of file systems
- On-disk data structures

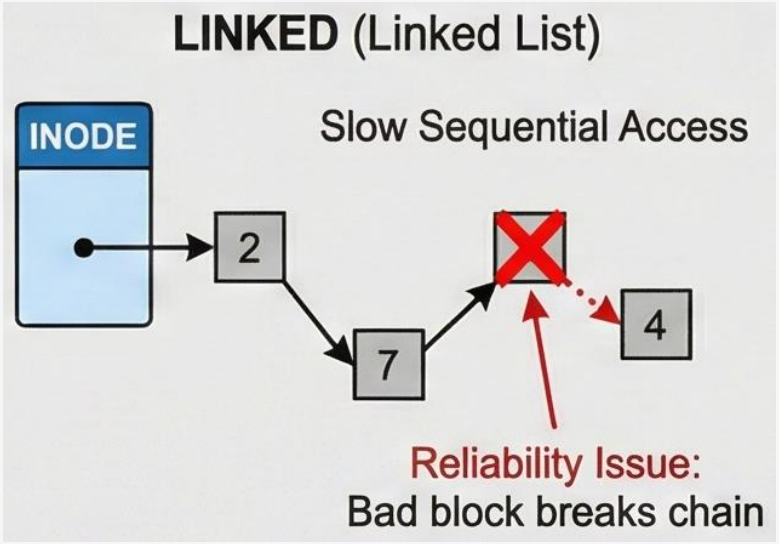
File System Responsibilities

- Map file data to blocks
  - how to organize data on disk?
  - how to find the blocks of a file?
  - how to store this information on disk?
- Track allocated and free space
  - which blocks are free?
  - how to find free blocks?

File Allocation Strategies

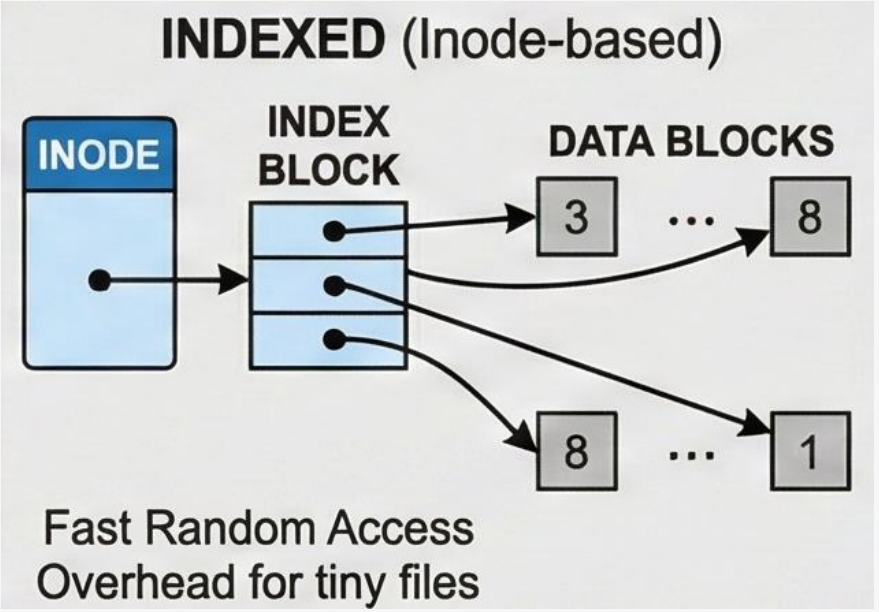
- How to assign blocks to a file
  - goal: maximize utilization (space) and minimize access time (speed)
- **Contiguous** (like array)
  - simple and high read performance
  - external fragmentation: deletions lead to holes
  - growth problem: move file each time
- **Linked allocation** (linked list)
  - poor read performance
  - reliability: one bad block -> no pointer to later blocks
  - overhead: pointer is 4 or 8 bytes
  - example: FAT (store all pointers in a table)





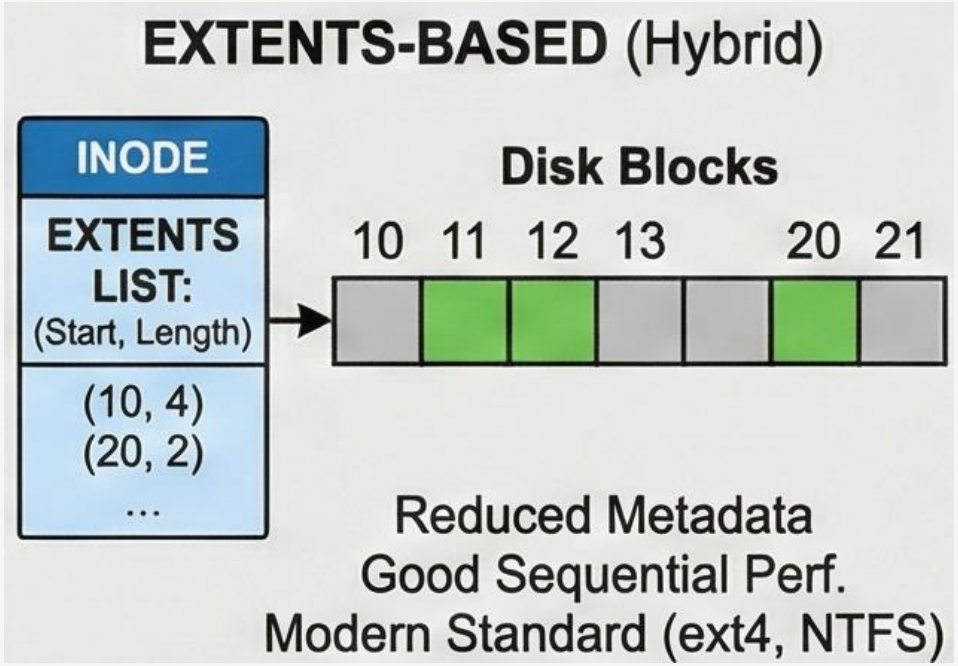
File Allocation Strategies

- Indexed allocation (inode)
  - a special block (Index Block): a list of pointers to data blocks
  - fast random access and no fragmentation
  - overhead: tiny file requires an index block
  - size limit: one index block has limited pointers, use multi-level indexing



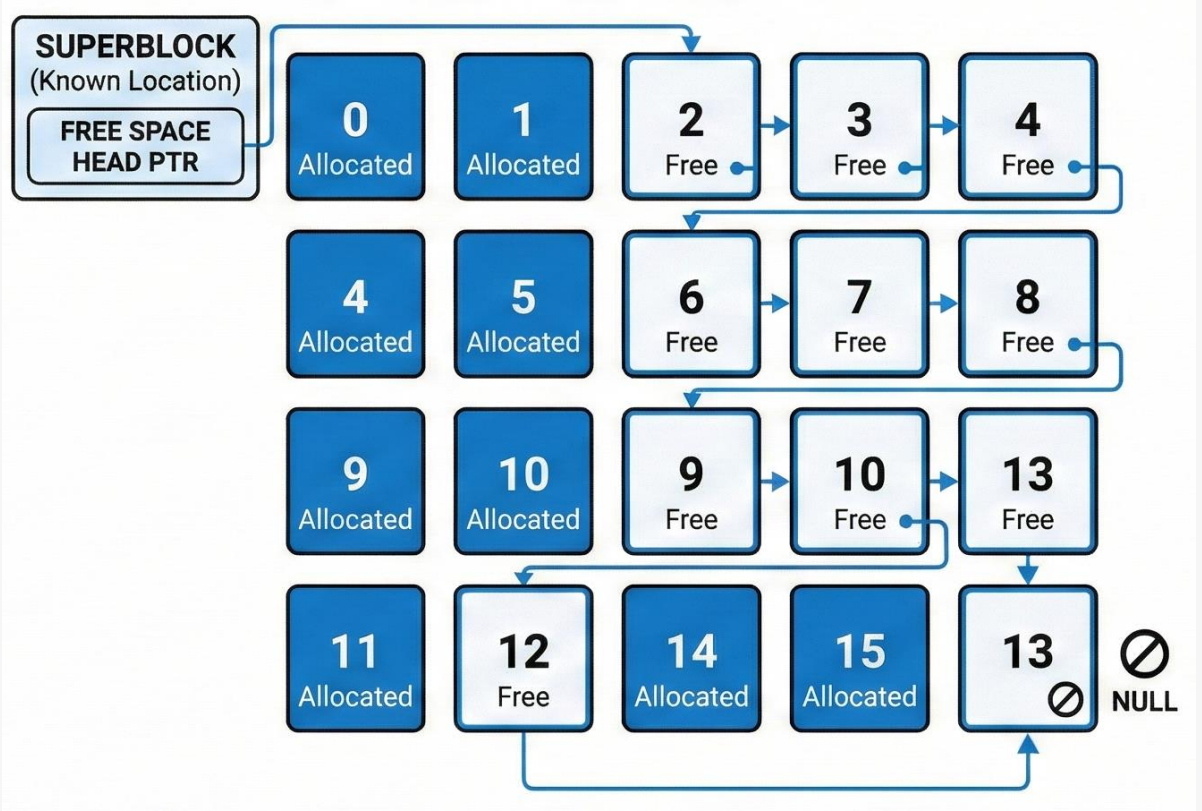
File Allocation Strategies

- Extents-based allocation
  - hybrid combining contiguous and indexed allocation
  - allocate chunks instead of blocks: store (start, length)
  - reduced metadata, good sequential performance, low fragmentation
  - modern standard: used in ext4, XFS, Btrfs and NTFS



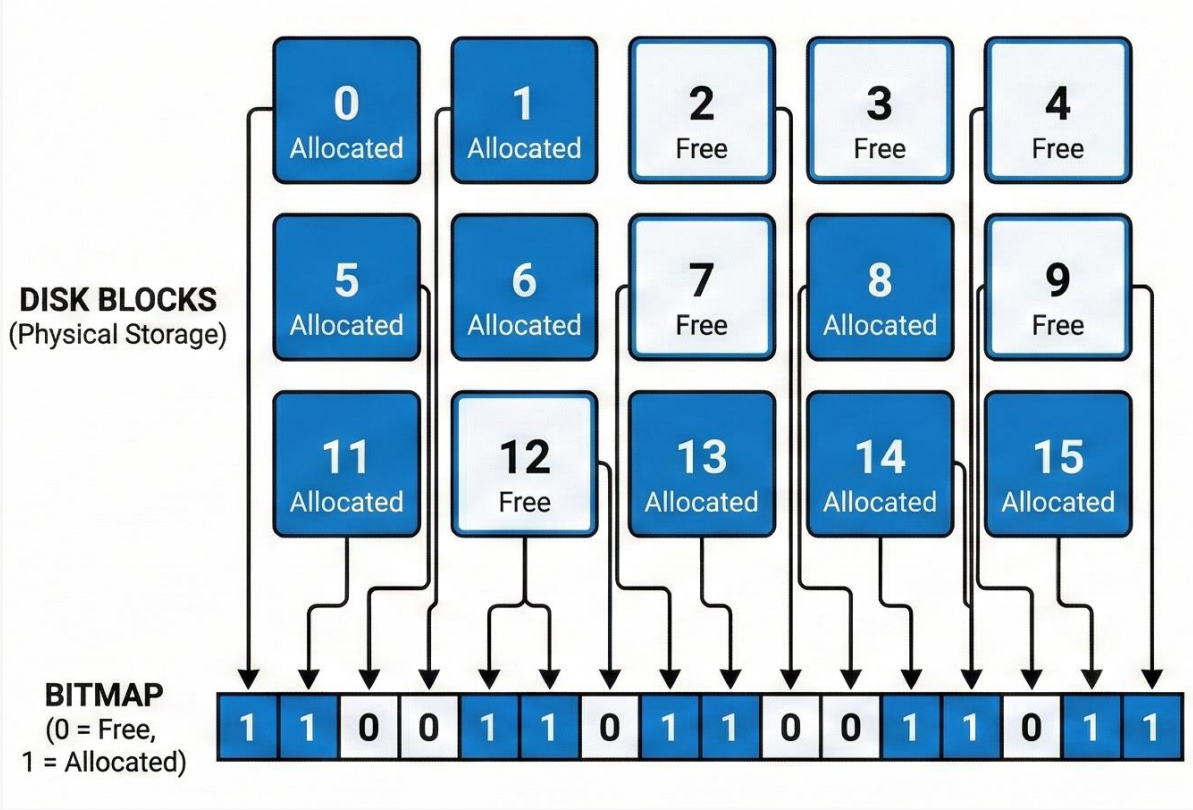
Free Space Management

- Problems: how to track and allocate free blocks
- **Free list: simple**
  - maintain a linked list of free blocks and store the list in free blocks
  - allocation: pop up one
  - problem: no spatial locality



Free Space Management

- **Bitmap: common**
  - tracking: array of bits for each block
  - allocation: scan array and take first free block
  - allocation (better): find free regions
    - use multi-level summaries to search for contiguous blocks efficiently
    - e.g., level 0: block bitmap, level 1: 64-block bitmap, level 2: 256 block bitmap
  - easy update, good scalability
  - ext4 uses bitmap



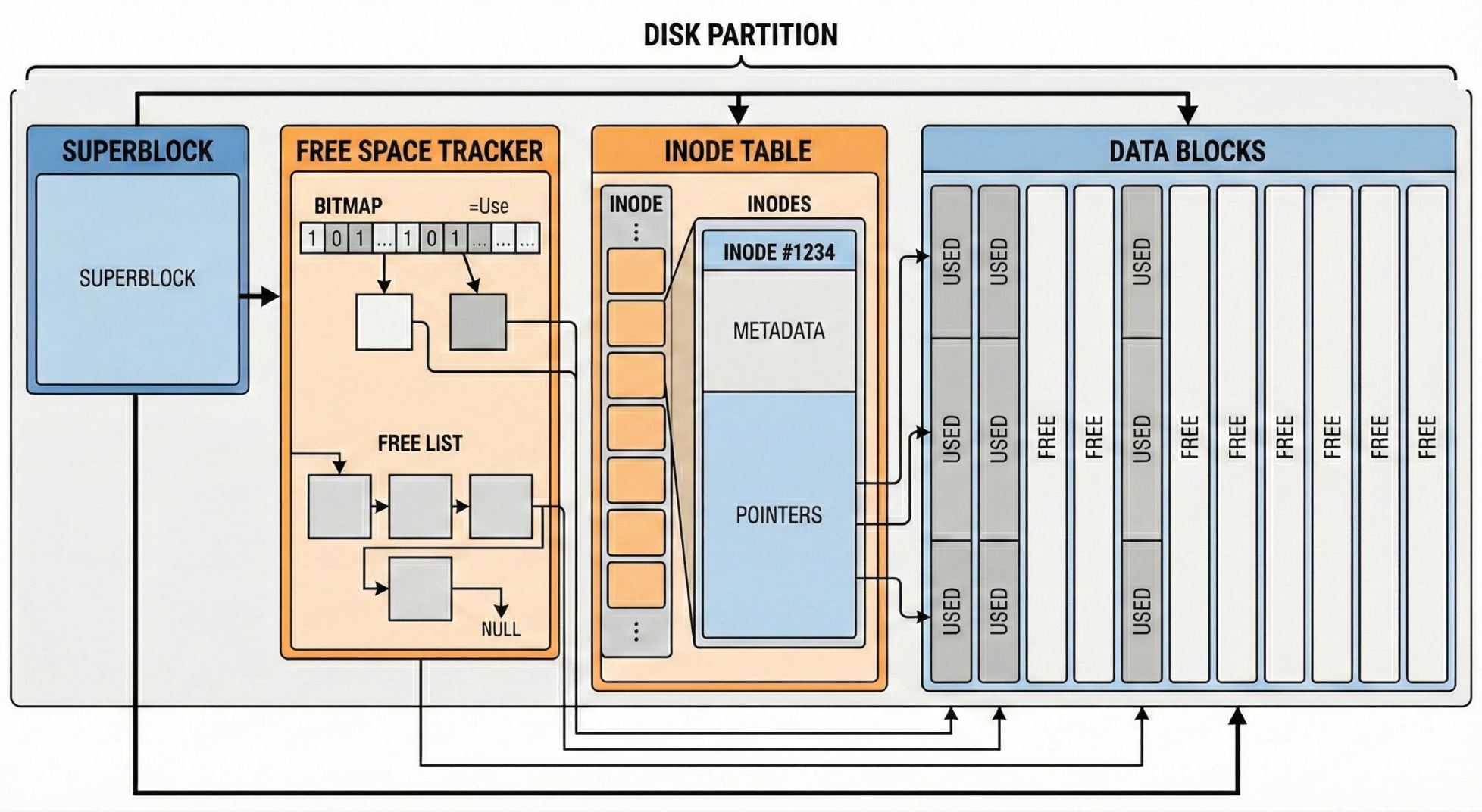
Free Space Management

- **Free extent list: sometimes better**
  - extent: a contiguous range of blocks, represent as (start, size)
  - tracking: maintain lists of free extents in two trees
    - address-ordered: for coalescing on free
    - size-ordered: for finding a certain size
  - downsides
    - small unfilled extents: huge unbounded metadata, slow search, many merges
    - hard to maintain: concurrency, consistency
  - used by XFS

On-disk Data Structures

- **Superblock**: FS metadata
  - block size, capacity, inode count, block count, state, volume name, magic
  - stored at multiple locations for redundancy
- **Bitmap or free list**: tracking free space
- **Inode**: metadata, permissions, and the pointer map
- **Data blocks**

On-disk Data Structures



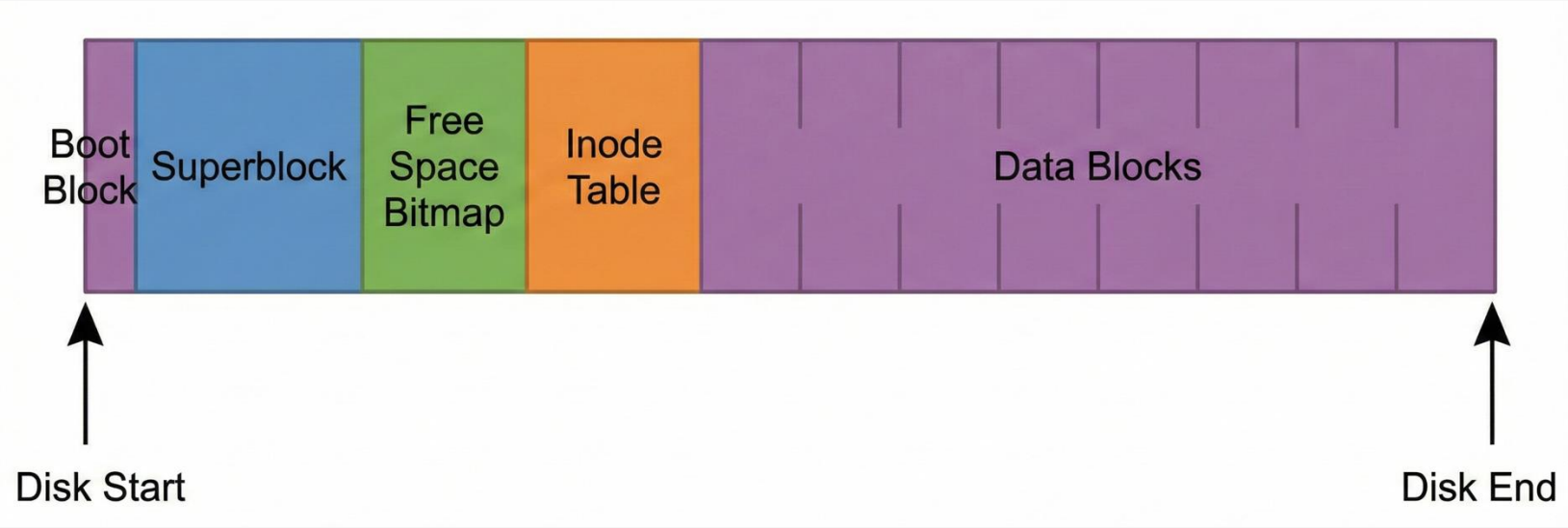
On-disk Data Structures: Inode

- Metadata
  - file type: regular file, directory, symlink…
  - permission, owner identifiers: UID and GID
  - timestamps: mtime, ctime, atime
  - file size, flags, attributes
  - link count: #directory entries (hard links) pointing to it
- Data locations (one of the following)
  - block pointers (classic inode design: ext2/3, many others)
    - direct pointers and indirect pointers (pointing to index blocks with more block pointers)
  - extents (common modern approach: ext4, XFS)
- Two types of inodes: file and directory

inode does not store filename!

File system storage layout (physical view)

- Dictate file system performance
- Need to match the characteristics of the underlying storage devices
  - HDD: slow random read and write, prefer sequential operations, so locality and large transfers are important
  - SSD: small random writes are bad (GC, WA, tail latency)



Comparing on-disk and in-memory (VFS) structures

Feature	VFS Superblock (struct super_block)	On-Disk Superblock (e.g., struct ext4_super_block)
Persistence	**Volatile** (destroyed on umount)	**Persistent**
Structure	**Generic** (same for all filesystems)	**Specific** (layout is unique to the filesystem type)
Contents	Contains **runtime** state: mount flags, reference counts, and pointers to operation functions	Contains **static** config: total inode count, block count, UUID, pointers to inode tables

Feature	VFS Inode (struct inode)	On-Disk Inode (e.g., struct ext4_inode)
Location	**RAM** (Kernel Memory)	**Disk** (Inode Table)
Identity	**Generic** (standardized interface for kernel)	**Specific** (optimized for specific storage format)
Lifecycle	**Volatile** (created when a file is accessed)	**Persistent** (exists until the file is deleted)
Content	**Runtime** state: locks, wait queues, dirty flags, reference counts	**Persistent** data: permissions, owner, timestamps, and pointers to data blocks