File Systems 2

Juncheng Yang

Feb 11

Harvard John A. Paulson
School of Engineering
and Applied Sciences

HARVARD UNIVERSITY
MADSYS
Measurement and Design of Systems Lab

Recap

- File system interface
  - file, directory, soft link, hard link
  - POSIX APIs
  - virtual file system (VFS)
    - four pillars: superblock, inode, dentry, file
- File system implementation
  - file allocation strategy

Five Questions To Ask Yourself After This Class

- 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()`
- How do file systems guarantee data integrity?
- What are the innovations in FFS?
- What are the challenges of designing a log-structured file system and how does LFS overcome them?

File System Implementations

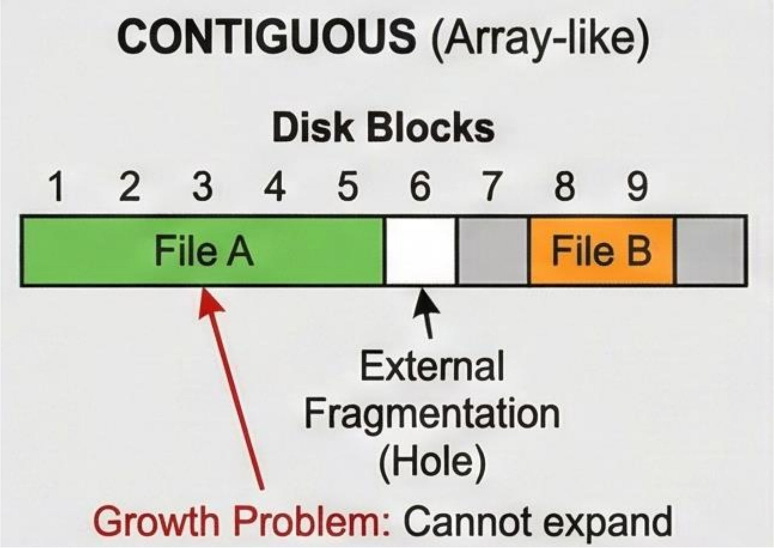
- Responsibilities of file systems
- On-disk data structures

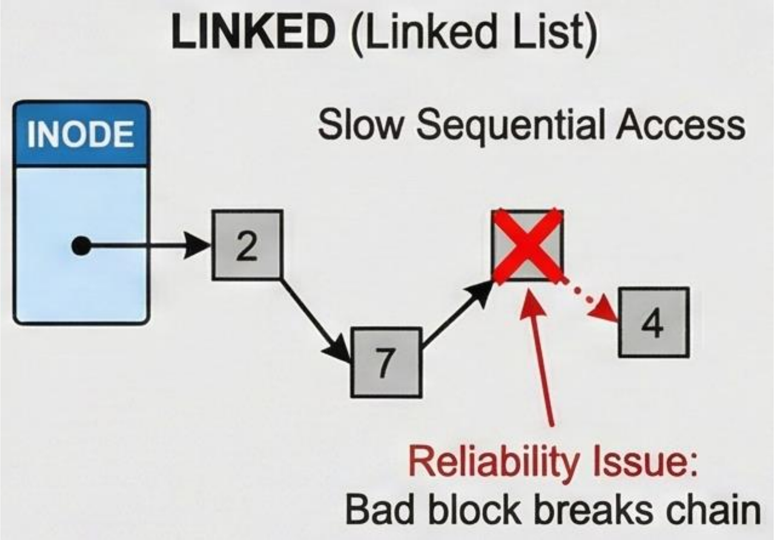
File System Responsibilities

- 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

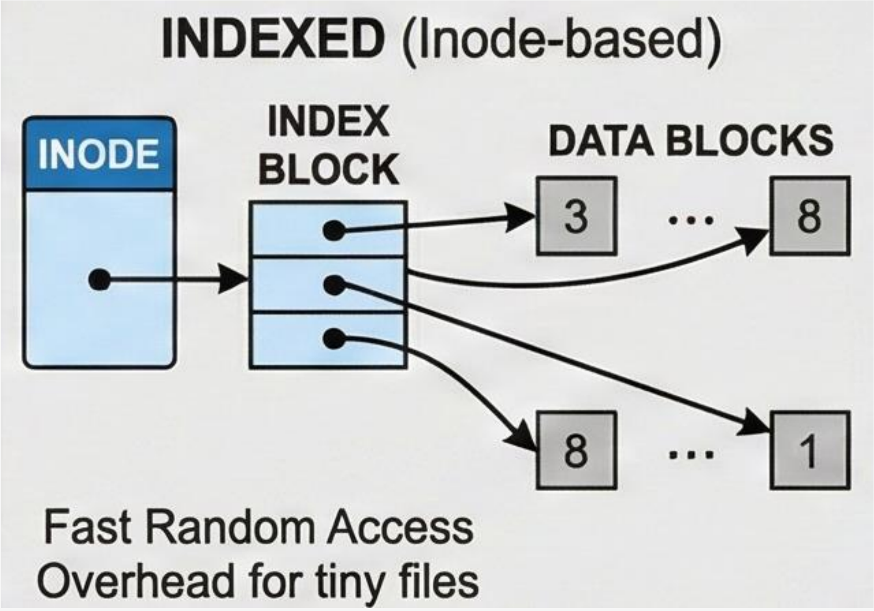
- How to assign blocks to a file
  - goal: maximize utilization (space) and minimize access time (speed)
- **Contiguous** (like array)
  - simple and high read performance
  - external fragmentation: deletions lead to holes
  - growth problem: move file each time
- **Linked allocation** (linked list)
  - 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)





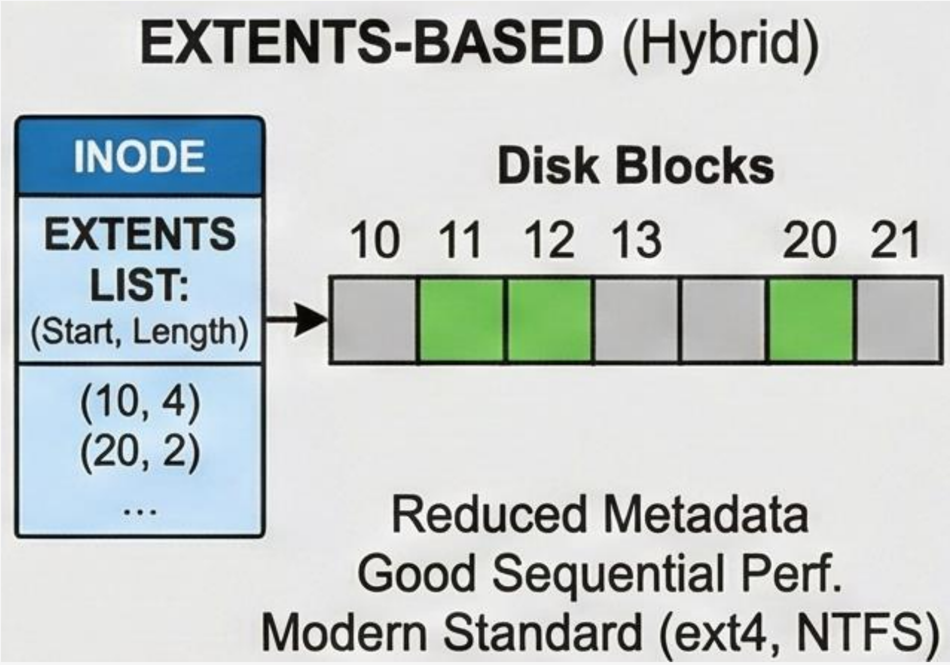
File Allocation Strategies

- 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



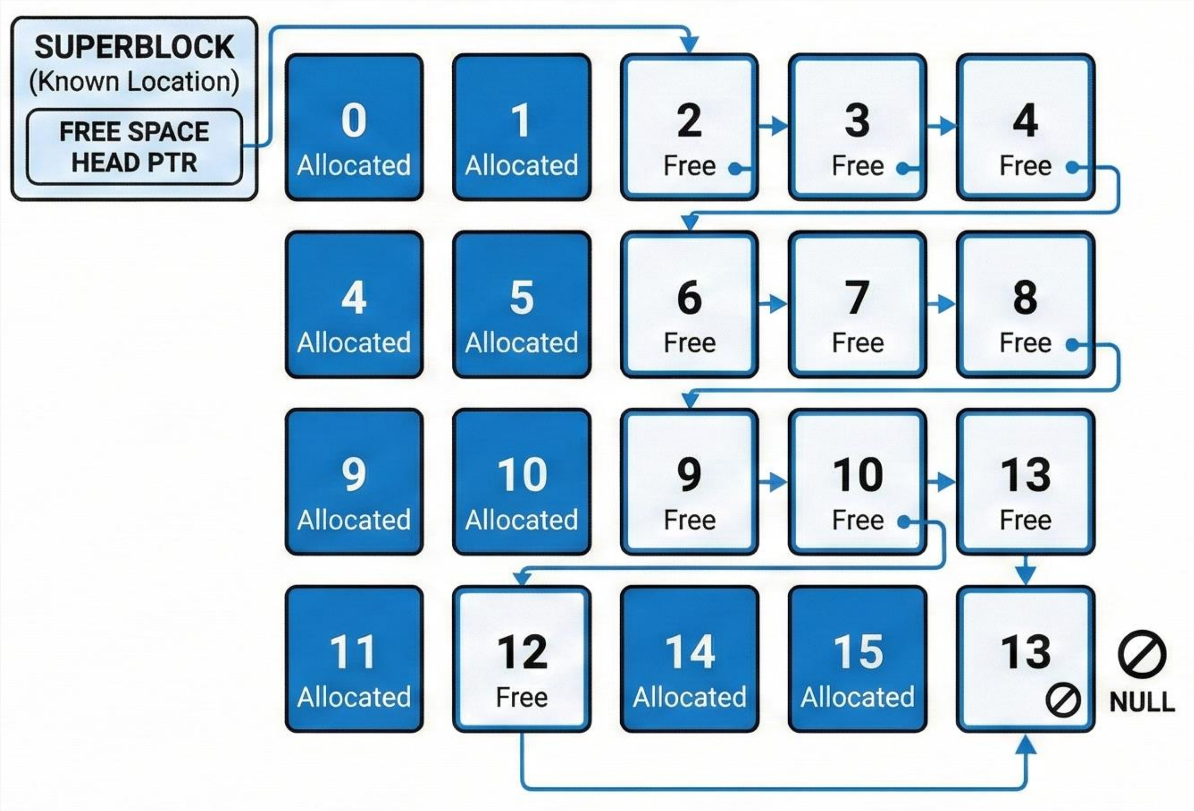
File Allocation Strategies

- 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



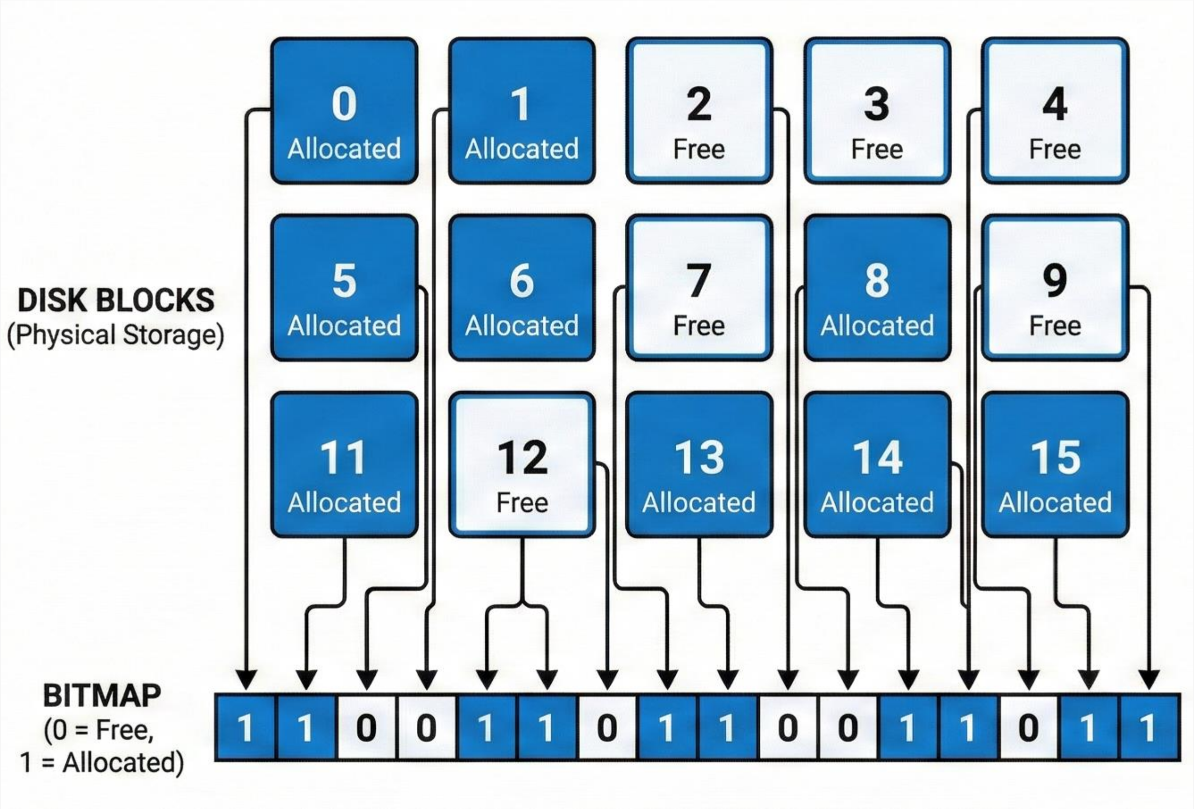
Free Space Management

- 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



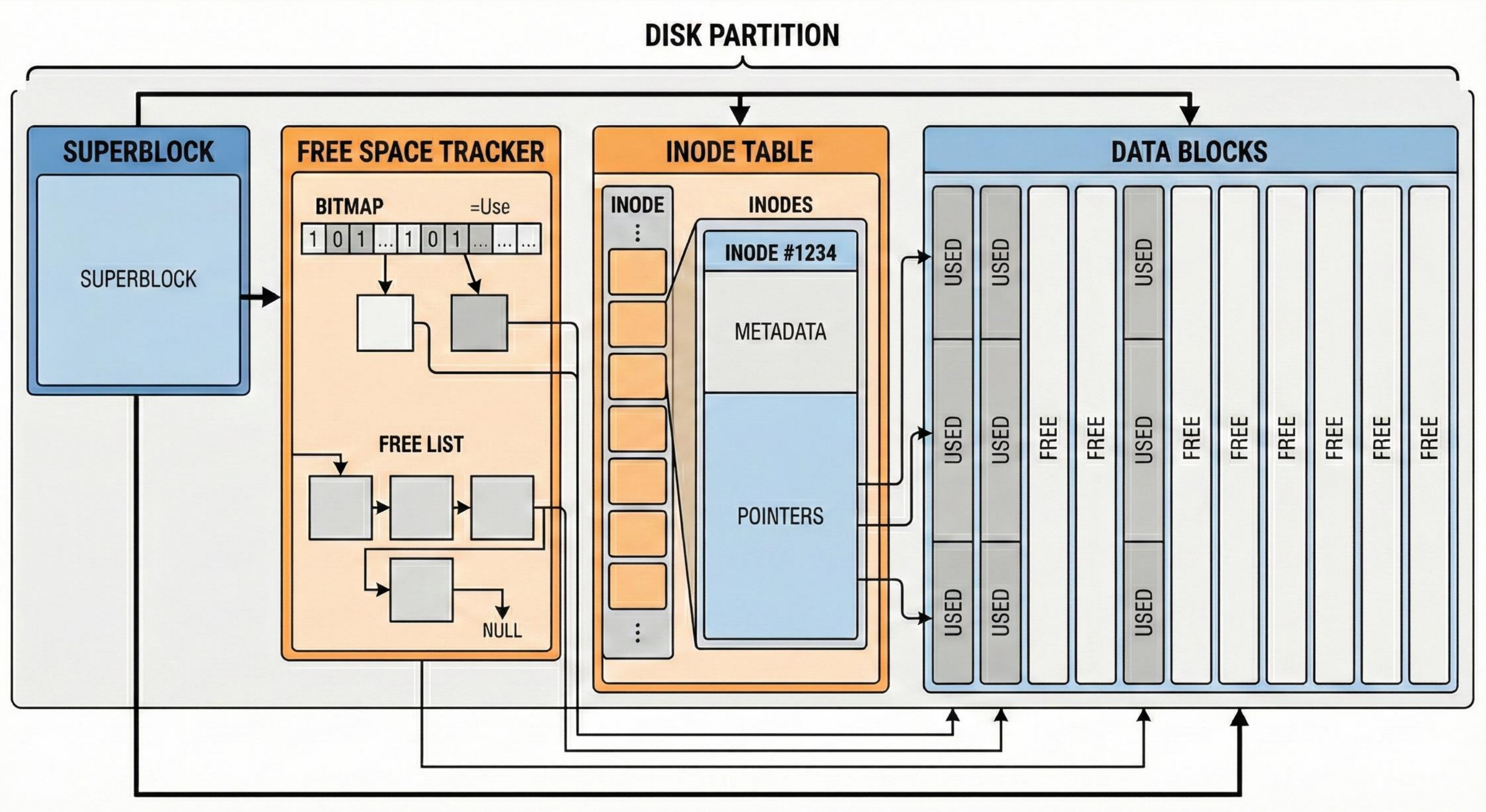
Free Space Management

- **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



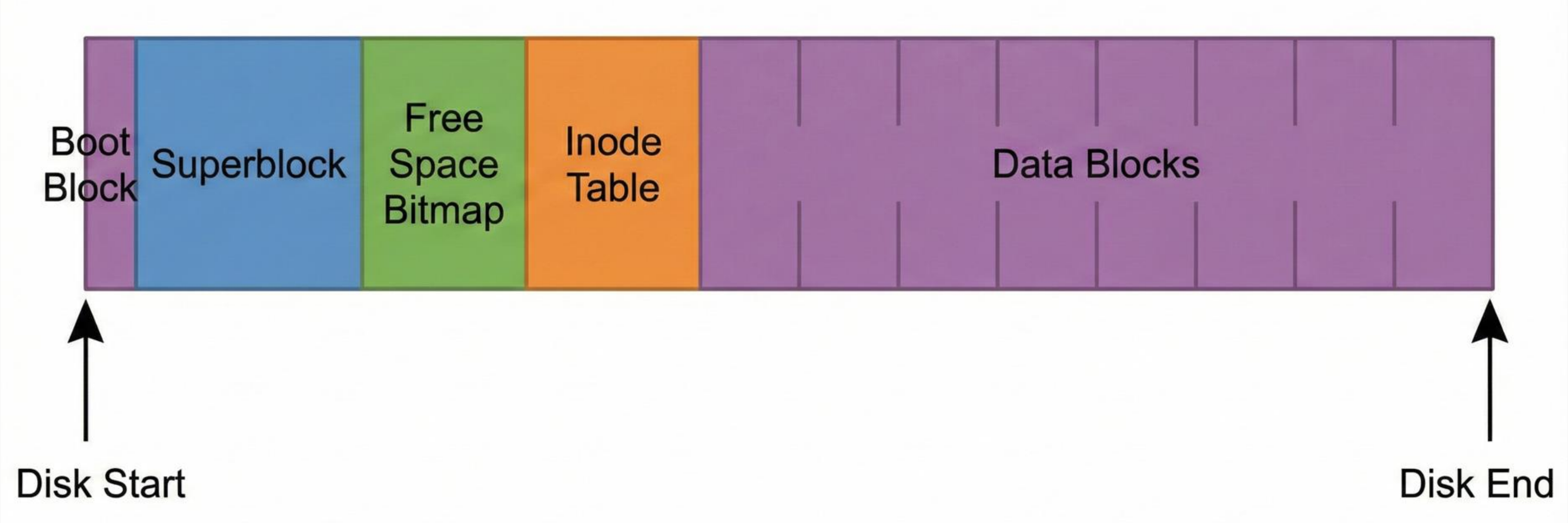
On-disk Data Structures: Inode

- 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

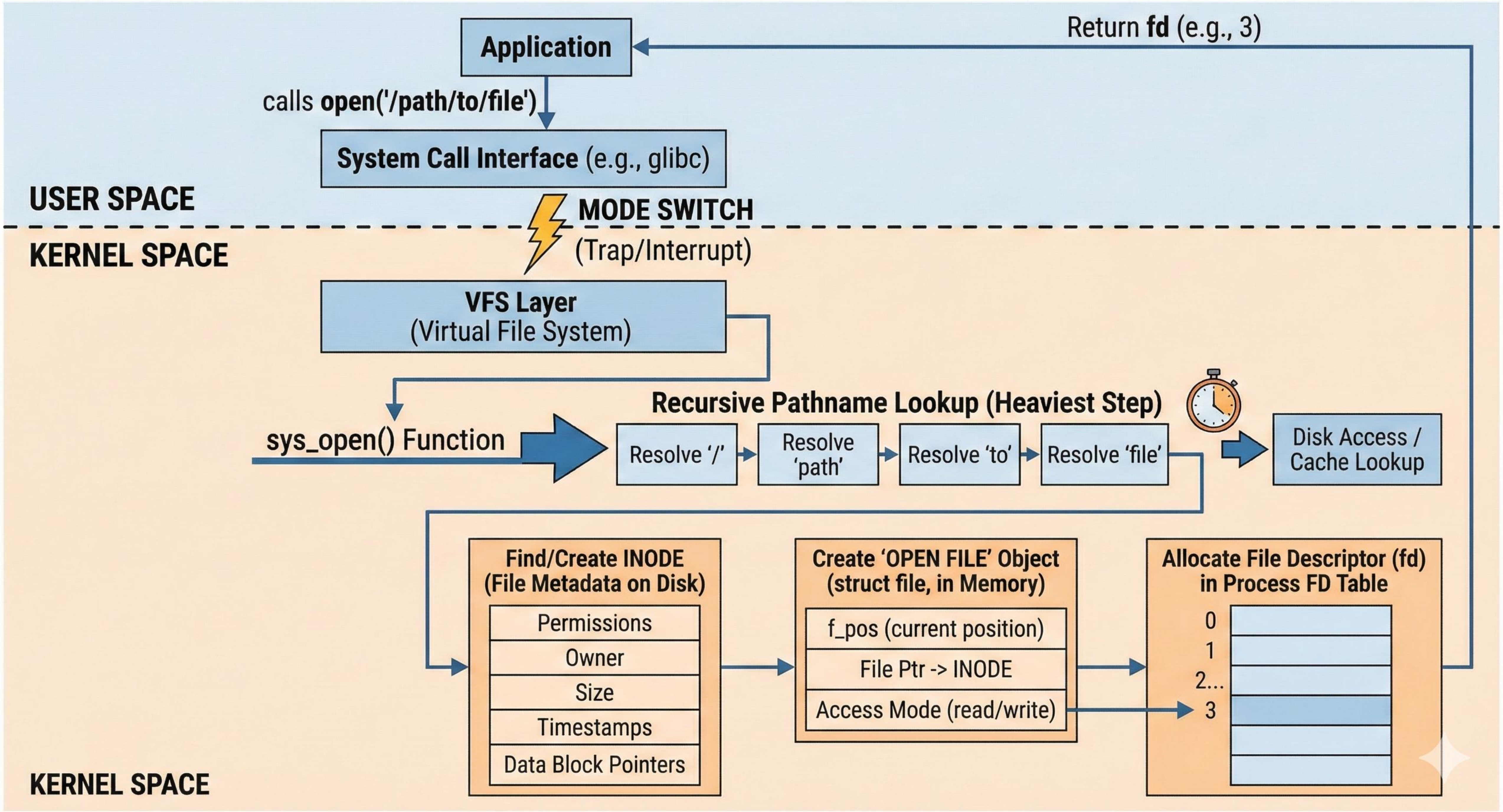
Feature	VFS Superblock (struct super_block)	On-Disk Superblock (e.g., struct ext4_super_block)
Persistence	**Volatile** (destroyed on umount)	**Persistent**
Structure	**Generic** (same for all filesystems)	**Specific** (layout is unique to the filesystem type)
Contents	Contains **runtime** state: mount 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

Request Flow

What happens when you call open()

- The mode switch (user to kernel)
- The Virtual File System Layer
  - the kernel lands in a function like `sys_open()`
  - recursive pathname lookup (the heaviest step) to find/create inode
  - create the "Open File" Object
- Allocate a File Descriptor


Return **fd** (e.g., 3)
Application
calls **open('/path/to/file')**
System Call Interface (e.g., glibc)
USER SPACE
MODE SWITCH
(Trap/Interrupt)
KERNEL SPACE
VFS Layer
(Virtual File System)
Recursive Pathname Lookup (Heaviest Step)
sys_open() Function
Resolve '/'
Resolve 'path'
Resolve 'to'
Resolve 'file'
Disk Access / Cache Lookup
Find/Create INODE (File Metadata on Disk)
Permissions
Owner
Size
Timestamps
Data Block Pointers
Create 'OPEN FILE' Object (struct file, in Memory)
f_pos (current position)
File Ptr -> INODE
Access Mode (read/write)
Allocate File Descriptor (fd) in Process FD Table
0
1
2...
3
KERNEL SPACE

What happens when you call read()

- Assume buffered I/O
- The mode switch (user to kernel)
- The Virtual File System Layer
    - find offset and inode
    - check cache: return on hit
    - move to file-system-specific code: offset -> LBA
- Block layer-> driver -> controller -> disk
- Copy from kernel (page cache) to user buffer

What happens when you read data

operation	inode			data			
	/	cs2640	s1.pdf	/	cs2640	s1.pdf block1	s1.pdf block2
open(/cs2640/s1.pdf)	**read**	**read**	**read**	**read**	**read**		
read()			**r+w**			**read**	
read()			**r+w**				**read**

What happens when you call write()

- Assume buffered I/O
- The mode switch (user to kernel)
- The Virtual File System Layer
  - find offset and inode
  - check cache
    - hit: modify page
    - miss: full page write vs. partial page write (read-modify-write)
  - return
- Asynchronous flush to disk

What happens when you write data

operation	bitmap		inode			data			
	data	inode	/	cs2640	s1.pdf	/	cs2640	s1.pdf block1	s1.pdf block2
create(/cs2640/s1.pdf)		**r+w**	**read**	**r+w**	**r+w**	**read**	**r+w**		
write()	**r+w**	**r+w**			**r+w**			**write**	
write()	**r+w**	**r+w**			**r+w**				**write**

FS Integrity

"What I read back is exactly what I wrote."

Main Challenges for Data Integrity

- Hardware failure
- File system and application issue
  - bugs
  - unclean shutdown, e.g., power loss
  - incorrect usage: concurrent updates without protection
- Silent data corruption
  - bit flip in memory

Two Types of Integrity

- Metadata integrity (the file system structure)
  - more challenging and complex
  - most problems from atomicity gap
    - file system operations requires multiple updates on disk
- Data integrity (the content)
  - less discussed
  - protection via checksum
    - standard file systems (ext4, XFS) rely on disk ECC
    - newer file systems (ZFS, BtrFS) actively track data integrity

File System Inconsistency: Orphaned Indoe

- `creat("/home/file.txt")`
  - find a free Inode (#99) and mark it as used in the **Inode Bitmap**
  - initialize **Inode** #99 on disk (set owner, permissions)
  - add the entry {"file.txt", 99} to the **parent directory's data block**
- Power fails after Step 2 but before Step 3
  - Inode #99 is marked "Used"
  - the file system has allocated resources for it
  - however, no directory contains a link to Inode #99
- Fix: this is what fsck finds and moves to /lost+found.

File System Inconsistency: Double Allocation

- Delete file_A, then create file_B
  - **File A:** remove dir entry, decrement link count, mark Inode #500 and data blocks as free
  - **File B:** allocates Inode #500, initialize Inode, insert dir entry
- Due to write reordering or a crash, both files own Inode #500
- Fix: no unless we have more information
- Same can happen to data blocks
- Similarly, crash during a delayed deletion causes zombie files

File System Inconsistency: Garbage Tail

- Append 4KB to a file
  - write the new data to Block #800
  - update file size in Inode and add Block #800 to list
- OS update Inode on disk, power loss before data is flushed
  - file system is structurally valid
  - file tail is garbage
- Fix: ext4 handles this with ordered mode, forcing data to be flushed *before* the metadata

File System Inconsistency: Directory Loop

- `mv /a/b /c/d`
  - add link to b in d
  - change b's parent pointer (..) to /c/d
  - remove link to b from a

note that this is in b’s data

- power loss between step 2 and 3
  - directory b is now reachable from *two* parents
  - if the move was crafted poorly (e.g., moving a parent into its own child), you can create a cycle
  - recursive traversal programs (like find) will loop infinitely until crash

Protect File System Integrity

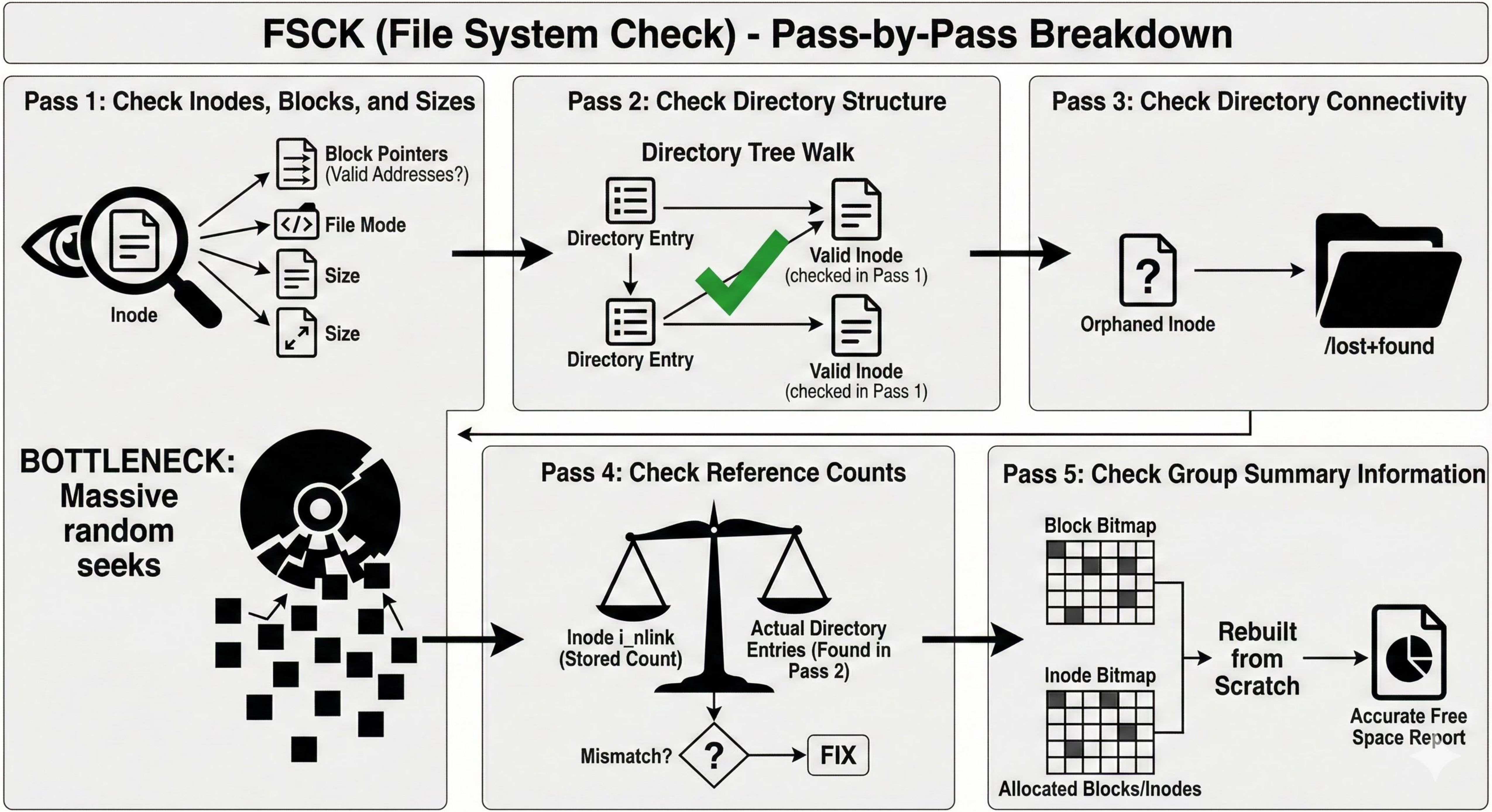
- The ordering rule
- File system consistency check (FSCK)
- Journaling (write-ahead logging)
- CoW (copy-on-write)

The Ordering Rule

- If file system object A depends on object B, then B must reach stable storage before A
  - Most all system-level inconsistencies stem from violating this rule.
- **Pointer Rule**
  - never write a pointer (directory entry/inode) to disk until the object it points to (inode/data block) is initialized on disk
- **Reuse Rule**
  - never reuse a resource (block/inode) until the previous owner's pointer to it has been cleared from disk

File system consistency check (FSCK)

- Five passes over the disk (ext4 as an example)
  - Pass 1 – inodes, blocks, sizes
    - ensure that the basic "building blocks" are correct
  - Pass 2 – directory structure
    - ensure the "folders" correctly point to the "files"
  - Pass 3 – directory connectivity
    - ensure there are no "floating" directories
  - Pass 4 – reference (hard link) counts
  - Pass 5 – group summary / bitmap
    - synchronize the file system’s "map"
- May run extra sub-passes 1B/1C/1D when needs to re-scan to resolve duplicate/bad blocks
- Details can be found in optional materials (OSTEP)


FSCK (File System Check) - Pass-by-Pass Breakdown
Pass 1: Check Inodes, Blocks, and Sizes
Inode
Block Pointers (Valid Addresses?)
File Mode
Size
Size
Pass 2: Check Directory Structure
Directory Tree Walk
Directory Entry
Directory Entry
Valid Inode (checked in Pass 1)
Valid Inode (checked in Pass 1)
Pass 3: Check Directory Connectivity
Orphaned Inode
/lost+found
BOTTLENECK: Massive random seeks
Pass 4: Check Reference Counts
Inode i_nlink (Stored Count)
Actual Directory Entries (Found in Pass 2)
Mismatch?
FIX
Pass 5: Check Group Summary Information
Block Bitmap
Inode Bitmap
Allocated Blocks/Inodes
Rebuilt from Scratch
Accurate Free Space Report

File system consistency check (FSCK)

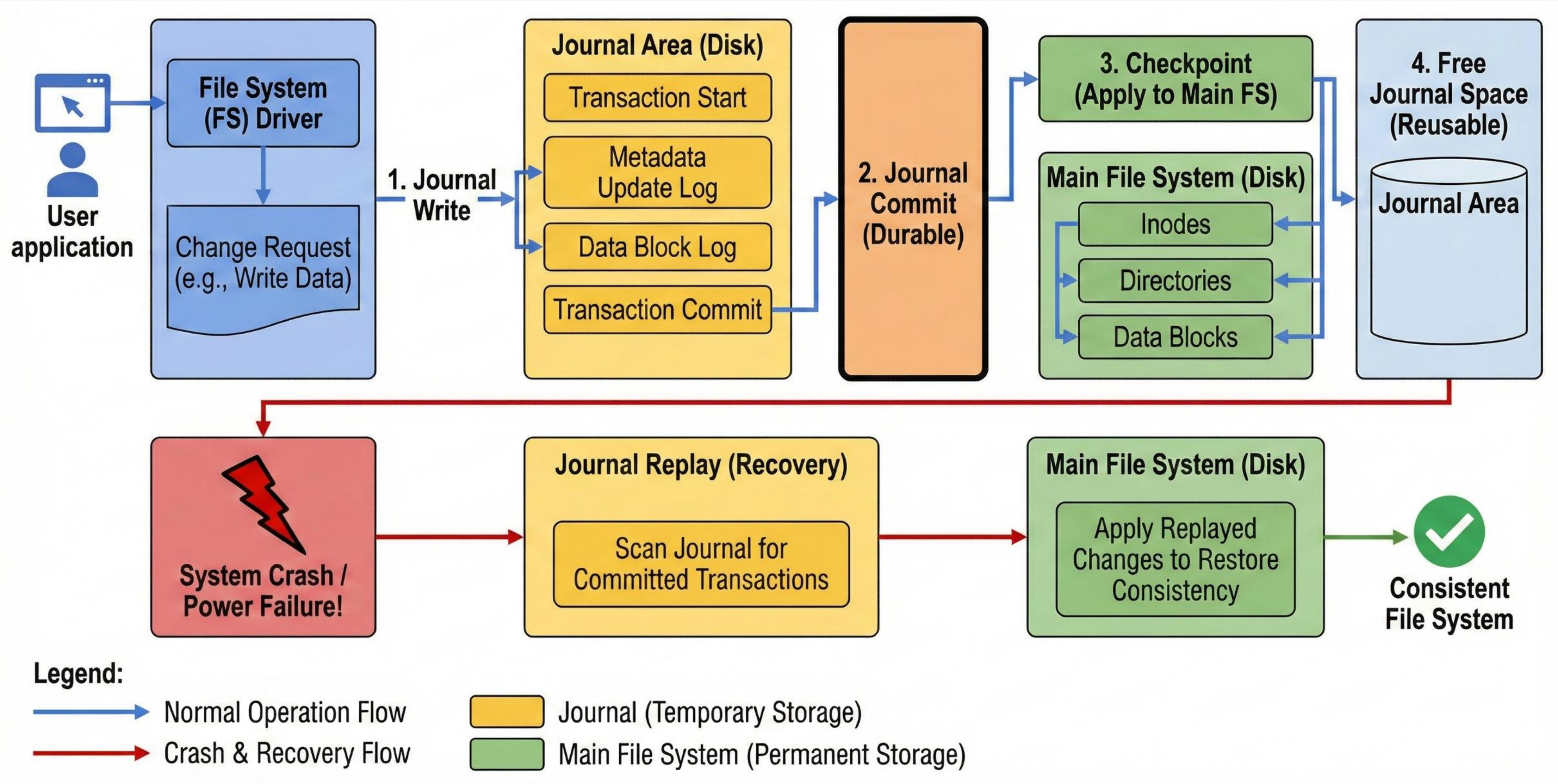
- Scan the disk, time complexity O(N)
- A lot of random I/Os (mostly pass 2,3,4)
  - due to separation of metadata and data
- Time to scan a 20 TB disk
  - 20,000,000 MB / 20 MB/s = 1,000,000 seconds = 12 days
  - you cannot use disk during the time

- Better to prevent inconsistency before it happens!

Journaling

- Never modify the filesystem structures until a description of the change is written to a separate "Journal”
  - write intended changes to a log first, so that a safe version always present before modification
  - log space reclaimed periodically
  - during crash recovery, can roll forward intended changes, recovery time proportional to un-checkpointed log size

Journaling



User application
File System (FS) Driver
Change Request (e.g., Write Data)
1. Journal Write
Journal Area (Disk)
Transaction Start
Metadata Update Log
Data Block Log
Transaction Commit
2. Journal Commit (Durable)
3. Checkpoint (Apply to Main FS)
Main File System (Disk)
Inodes
Directories
Data Blocks
4. Free Journal Space (Reusable)
Journal Area
System Crash / Power Failure!
Journal Replay (Recovery)
Scan Journal for Committed Transactions
Main File System (Disk)
Apply Replayed Changes to Restore Consistency
Consistent File System
Legend:
Normal Operation Flow
Crash & Recovery Flow
Journal (Temporary Storage)
Main File System (Permanent Storage)

Journaling—Problems

- Double write
  - consumes bandwidth and reduce lifetime
- Seek penalty (ping-pong effect)
  - to write a file, the disk head must
    - Seek to the Journal (Write metadata)
    - Seek to the Data Block (Write content)
    - Seek *back* to the Journal (Write Commit Record)
    - Seek *back* to the Metadata Table (Checkpoint/Flush)
- Serialization overhead
  - journal write cannot be paralleled even if you have multiple cores

CoW (Copy on Write)

- Never overwrite live data
- **Shadow Paging**
  - when you modify data
  - **allocate:** OS finds a **new, empty block**
  - **write:** writes the modified data to this new location
  - **pointer swap:** *after* the write is successful, update the metadata
- Benefits
  - Instant snapshot
  - Stronger integrity by forming a merkle tree (storing a hash for every block)

CoW (Copy on Write): Bubble Up Effect

- Update one byte in a file
  - create a new data block
  - create a new Inode for the file (update pointer)
  - create a new Inode for the directory (update filename to Inode mapping)
  - create a new Inode for the parent directory
  - ...
  - the final step is an atomic update of the root

- If power fails at any point *before* the root update, the filesystem treats it is as if the write never happened

CoW (Copy on Write): Problems

- Write amplification
  - bubble up effect
- Fragmentation
  - each update is in a random place

File System Integrity Summary

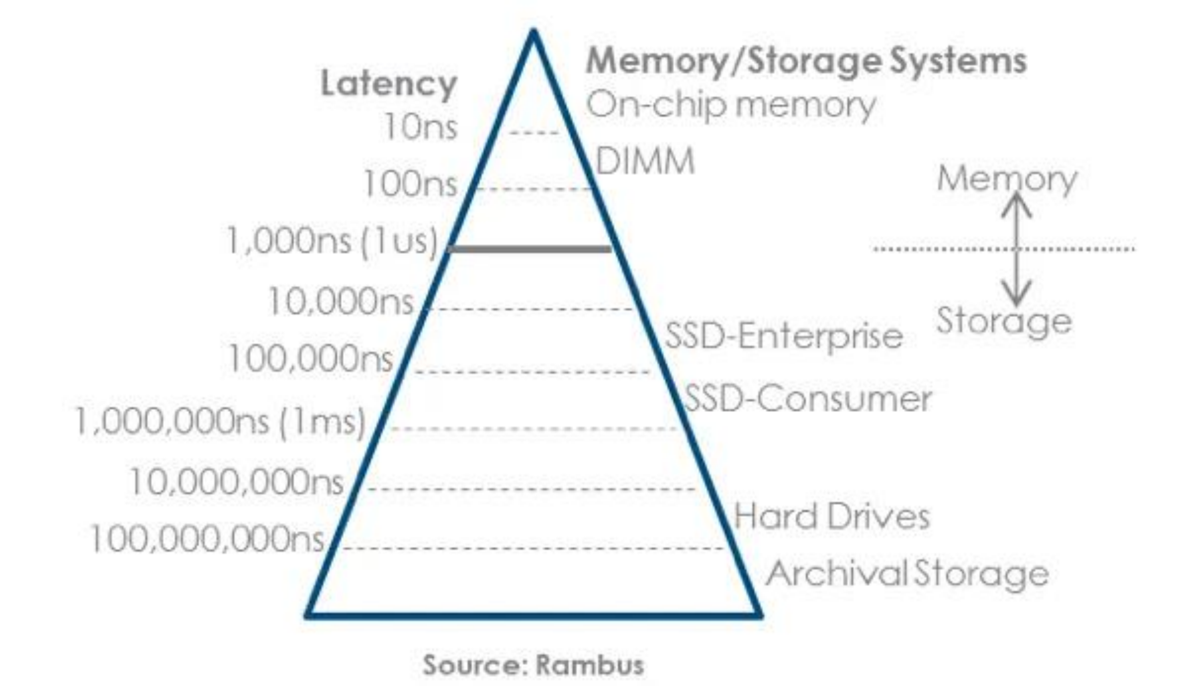
Feature	Journaling (ext4, XFS)	Copy-on-Write (ZFS, Btrfs)
Update Strategy	Overwrite in place + log changes	Write to new space + pointer swap
Crash Safety	Relies on Replaying the Log	Relies on atomic root update
Snapshots	Slow	Instant
Data Integrity	Metadata consistency only	Full Data + Metadata
Fragmentation	Low (updates stay in place)	High (updates move around)

FS Performance and Efficiency

How to Improve Performance: User View

- Performance is largely dictated by underlying device

- Page cache
- Memory-mapped I/O
- Asynchronous I/O and io_uring



File System Performance

- File system can be slower than device
  - Metadata operations
  - Consistency overhead: flush, journaling
  - kernel I/O stack: many layers, file system lock contention

Slower in terms of IOPS, bandwidth and bandwidth

- Tuning file system performance
  - trade off feature: `noatime, relatime, lazytime`
  - less aggressive flush and journaling: `commit=60, data=writeback`
- File system design
  - HDD: centered around sequential write and locality (place related data closer)
  - SSD: centered around reducing GC overhead (reduce random writes, use TRIM)

File System Efficiency

- Space efficiency: consumed space / data size
- Source of inefficiency
  - allocation unit (internal fragmentation), 512B, 4KB, 64KB
  - metadata: inode, pointers, journaling, reserved space

File System Efficiency

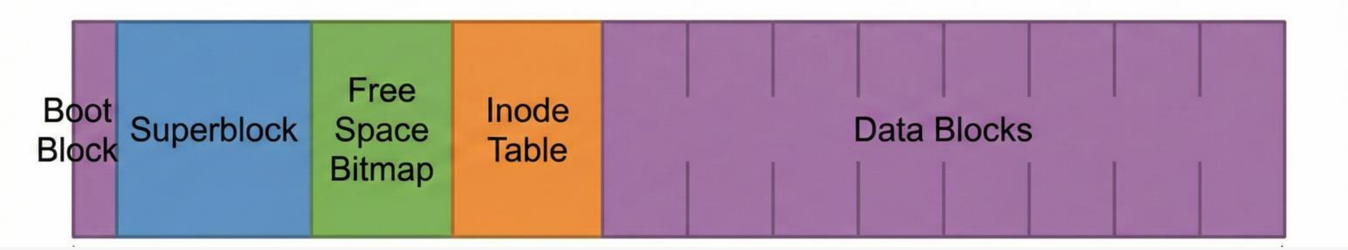
- Tail Packing / Block Suballocation
  - store multiple tiny files in one block or split a block to smaller fragments
- Sparse Files
  - if a file contains a lot of empty data (zeros), do not write these zeros
- Deduplication
- Compression

Fast File System (FFS)

A fast file system for UNIX, TOCS'84 (released as part of BSD 4.2)
https://dl.acm.org/doi/10.1145/989.990

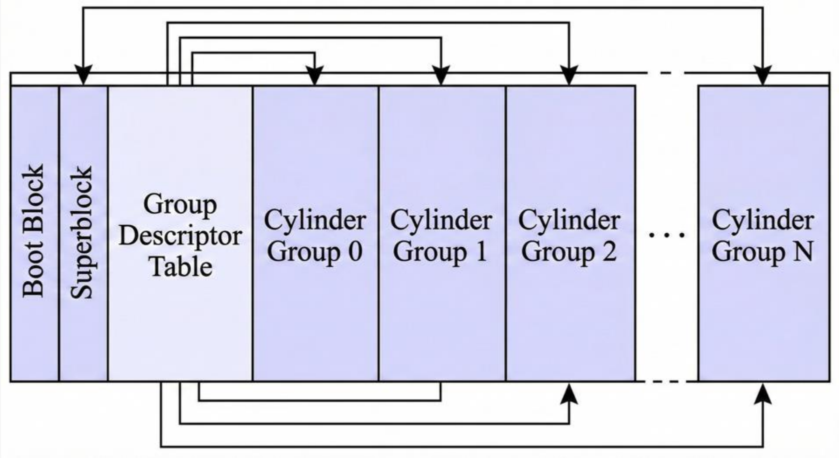
Fast File System (FFS)

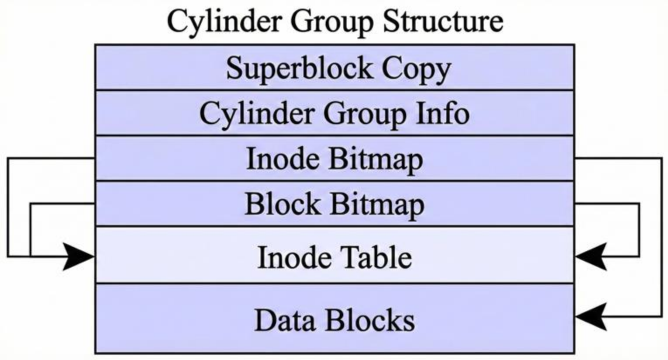
- The first disk-aware file system
  - demonstrates that disk layout matters for performance
- Original Unix file system: treat the disk like a flat list of blocks
  - poor performance, often delivering 2% of the disk's potential bandwidth
  - long seek distance
    - Inodes (metadata) are stored at the start of the disk, reading a file requires long seeks back and forth between inode and the data
  - tiny blocks: 512B, reading large files requires massive metadata processing overhead
  - fragmentation

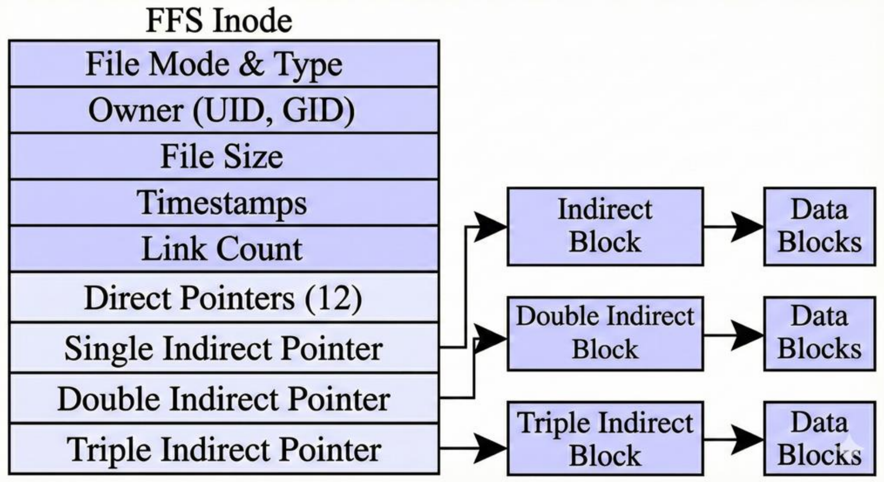


FFS: Cylinder Groups

- Idea: keep related data closer
- Innovation 1:
  - break the disk into Cylinder Groups (CG)
  - each CG can be viewed as a mini file system, contains
    - a superblock copy (for redundancy)
    - bitmap for tracking free blocks
    - Inodes
    - data blocks







FFS: Smart Allocation to Create Locality

- Place related data (Inode and data blocks) close
- Problem: filling up a cylinder group too quickly
- Load balance
  - spread directories evenly across disks
  - force a jump if the file is too large
    - the first twelve direct blocks are placed in the same group
    - each subsequent indirect block and all blocks it points to are placed in a different group

FFS: Large Block Size and Fragments

- Small block size
  - less space waste, low throughput due to massive seeks
- Large block size
  - higher throughput but higher internal fragmentation
- FFS uses a large block size (4KB)
  - modifies libc to buffer writes and issue 4KB chunks
  - except for the tail block that uses 512B
- Smart choice: disk density quickly improved, but seek time did not

FFS: Optimizations are Still Relevant Today

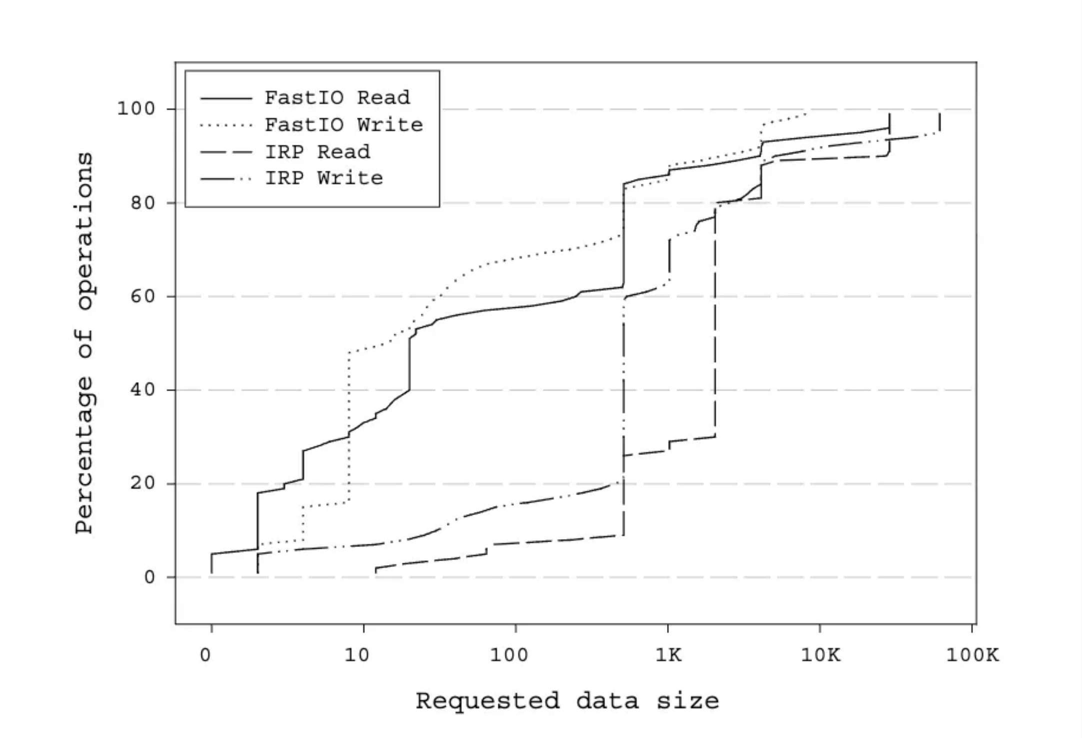
- Cylinder group: block groups in ext4
- Bitmap: a new and popular free-space management solution that replaces linked lists
- Make FS more usable
  - long file names
  - symbolic links to enable spanning across FS
  - introduce an atomic rename operation

Log-Structured File System (LFS)

The design and implementation of a log-structured file system, TOCS'92
https://dl.acm.org/doi/10.1145/146941.146943

Motivation

- Small writes are slow for HDDs
- User writes are often small
- Write amplification
  - e.g., create a new file of size one block
    - update the file inode bitmap
    - update directory data block (create name to file inode mapping)
    - update the directory inode (timestamps, size)
    - write the new file data block
    - update the data bitmap (mark the data block as allocated)
    - update file inode



https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/File System Usage.pdf

LFS: Log-Structured File Systems

- Never write in-place
- Buffer changes in memory
- When the buffer is full (forming a **Segment**), it writes the entire batch to the disk in one long, continuous burst
- Indication
  - the newest version of any block is "somewhere later in the log"
  - the disk layout is chronological, not spatially organized by file

LFS: Challenges and Solutions

- how large should the DRAM buffer be?
  - amortize the seek cost
- how to find updated Inode?
  - indirection: inode map
- how to reconstruct from scattered metadata
  - checkpoint of full metadata
- how to remove invalid data
  - garbage collection
- how to choose which segment to clean
  - cost–benefit analysis

LFS: Log-Structured File Systems

```
[ Superblock ] [ Checkpoint 0 ] [ Checkpoint 1 ] [ Segment 1 ] [ Segment 2 ] ... [ Segment N ]
     |              |               |           ^            ^
     |              +---------------+           |            |
     |          (Points to latest Imap)         |            |
     |                                          |            |
     +--- Static Information                    +--- The "Log" (Data written here)
```

- Checkpoint
  - pointers to the latest inode-map blocks
  - log head, file system geometry, checksum, and segment usage table...
  - two checkpoints for crash consistency
- Segment

Log-Structured File Systems

```
_______________________________________________________________________
|            |           |           |           |          |        |
| Seg Summary | Data Block | Data Block |   Inode    |   Inode   |  Imap  |
|   (Block)   |  (File A)  |  (File B)  |  (File A)  |  (File B) | (Chunk)|
|____________|___________|___________|___________|__________|_______|
```

imap[i] = <segment#, offset-within-segment>

- Segment summary
  - reverse mapping from block to Inode
  - used by the cleaner to interpret what's in the segment
- Inodes
  - written *next* to the data they describe, improving read locality
- Inode map (imap) chunk
  - Inodes keep moving, Imap tracks current location
  - only the updated part of the inode map is written, the checkpoint has the full map

Log-Structured File Systems

- On reboot:
  - Read checkpoint region
  - Reconstruct the latest imap
  - Use it to find inodes and files
- On crash:
  - the same as reboot
  - read the new segments since checkpoint to roll forward

Log-Structured File Systems: Read

- Similar to traditional FS: first find Inode then read data blocks
- A useful way to think about LFS reads:
  - **checkpoint** → tells you where the latest **imap** is
  - **Imap** → tells you where the latest **inode** is
  - **Inode** → tells you where the latest **data blocks** are
  - fetch data blocks

Log-Structured File Systems: Write

- In memory
  - dirty file data blocks and metadata changes accumulate
- Flush to disk
  - pack data blocks and Inodes into a segment
  - write the segment sequentially
  - update the Imap entries for updated inodes
  - occasionally write a checkpoint
- Many scattered updates become one large sequential write

Log-Structured File Systems: Garbage Collection

- Why: LFS never overwrites data
  - segments contain a mix of
    - **live blocks** (the newest version)
    - **dead blocks** (superseded by newer writes)
- Garbage collection: compact segments to reclaim space
  - pick a segment
  - check if a block is live: find identity in seg summary, consult Inode and Imap
  - copy live blocks into a new segment
  - mark the old segment free

Log-Structured File Systems: Garbage Collection

- Q: which segment to clean?
- Options
  - old segments: reclaimed data more likely to remain live
  - segments with less live data: reclaim more space, but reclaimed data may become invalid soon
- LFS solution: cost-benefit analysis that combines the two
  - score= $\frac{(1-u)\times age}{1+u}$
  - u = segment utilization (fraction still live), 1−u = benefit
  - age = an estimate of data “stability”
  - 1+u approximates the cost: **1** to read the segment plus **u** to rewrite its live data
- LFS maintains a segment usage table that records, per segment, live bytes and age of youngest block

Summary

- File system implementation
  - how to allocate file
  - how to track free space
  - the four main on-disk data structures
- Integrity
  - journaling and CoW
- FFS
  - cylinder group and smart allocation
- LFS
  - on-disk data structures
  - read, write and garbage collection