CS162
Operating Systems and Systems Programming
Lecture 21

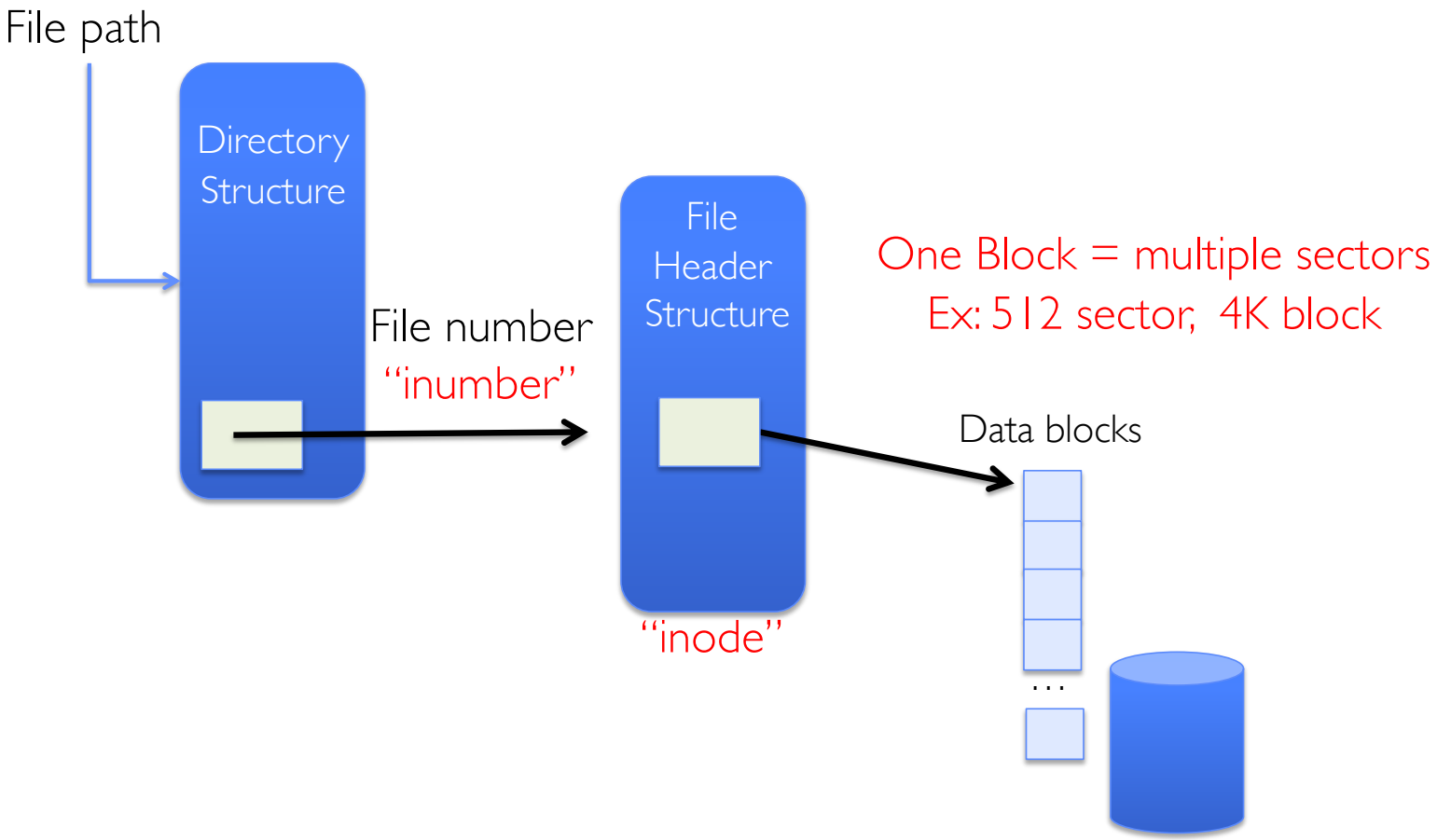
Filesystems 3: Reliability and Transactions

April 13th, 2021

Profs. Natacha Crooks and Anthony D. Joseph

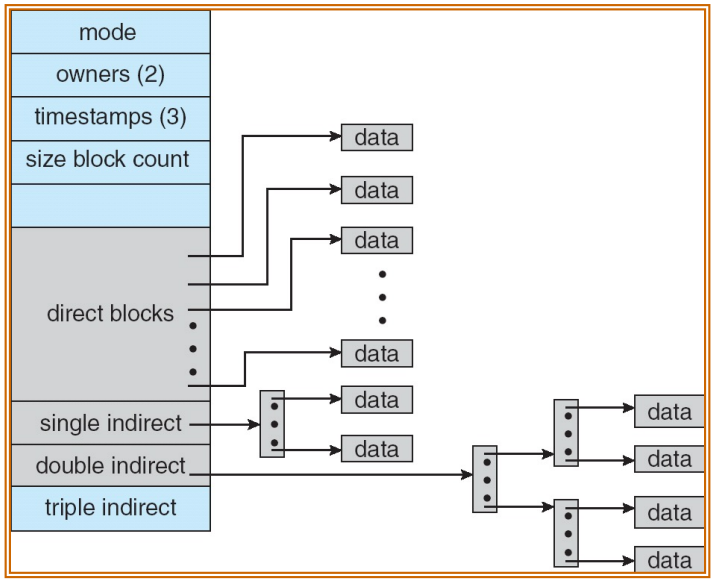
http://cs162.eecs.Berkeley.edu

Recall: Components of a File System



Recall: Multilevel Indexed Files (Original 4.1 BSD)

- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect block
    (Four I/Os per block!)

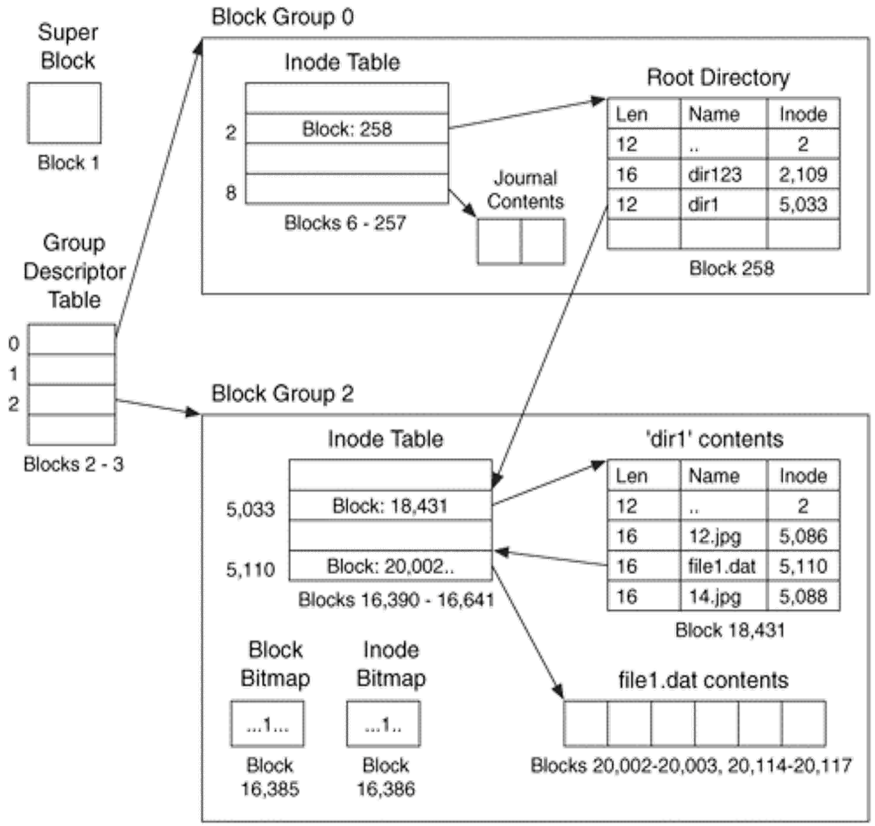


Recall: Fast File System (BSD 4.2, 1984)

- Same inode structure as in BSD 4.1
  - same file header and triply indirect blocks like we just studied
  - Some changes to block sizes from 1024 ⇒ 4096 bytes for performance
- Paper on FFS: "A Fast File System for UNIX"
  - Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry
  - Off the "resources" page of course website – Take a look!

- Optimization for Performance and Reliability:
  - Distribute inodes among different tracks to be closer to data
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned later)

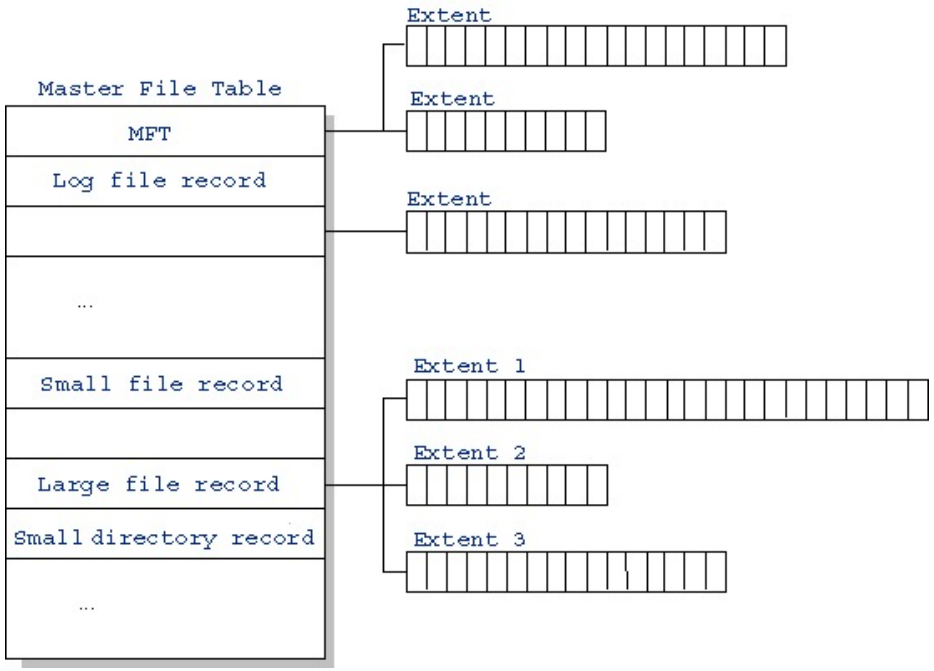
Recall: Linux Ext2/3 Disk Layout



- Disk divided into block groups
  - Provides locality
  - Each group has two block-sized bitmaps (free blocks/inodes)
  - Block sizes settable at format time: 1K, 2K, 4K, 8K…
- Actual inode structure similar to 4.2 BSD
  - With 12 direct pointers instead of 10
- Ext3: Ext2 with Journaling
  - Several degrees of protection with comparable overhead
  - We will talk about Journaling Today!

Example: create a `file1.dat` under `/dir1/` in Ext3

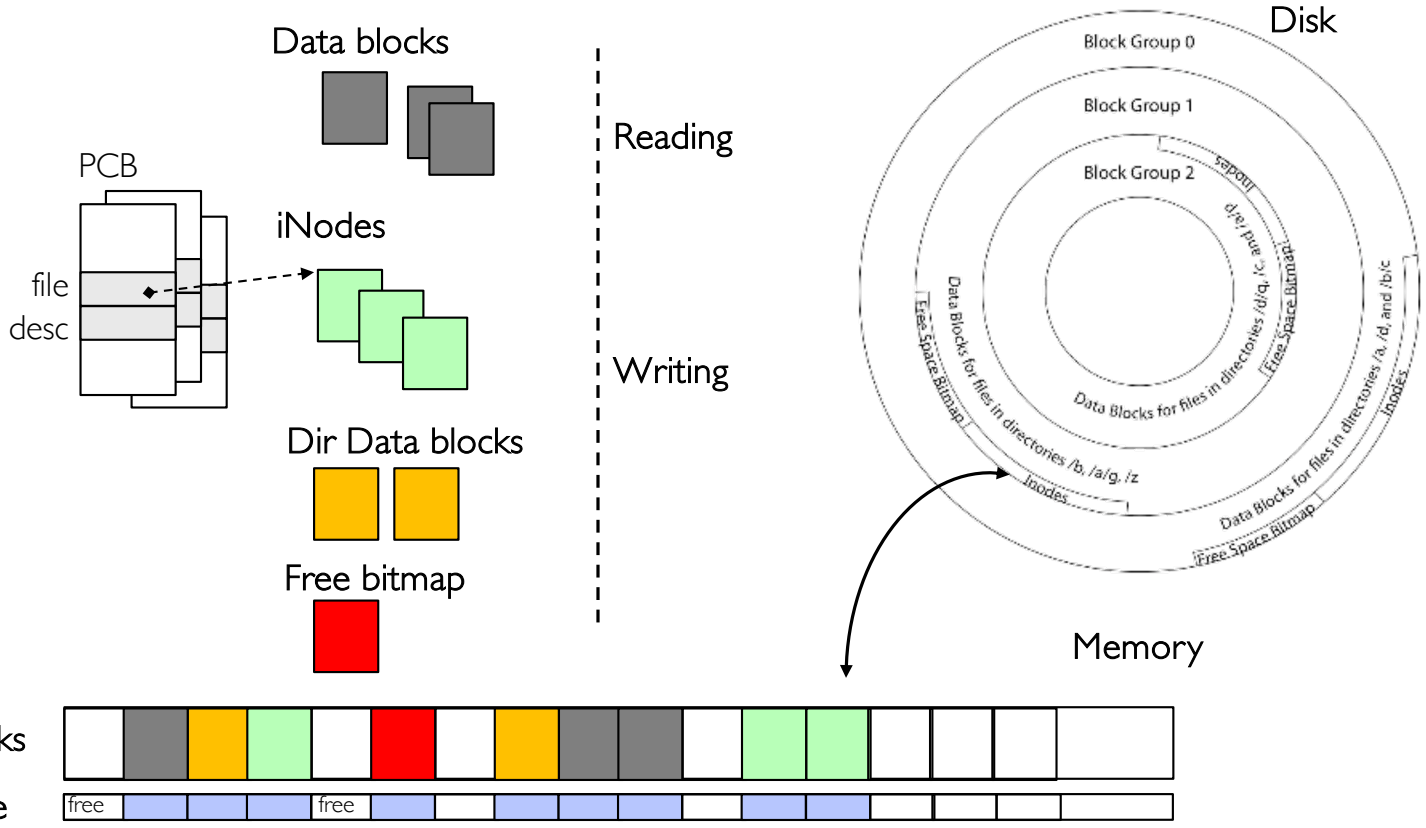
Recall: NTFS



http://ntfs.com/ntfs-mft.htm

- Master File Table
  - Database with flexible 1KB entries for metadata/data
  - Variable-sized attribute records (data or metadata)
  - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
  - Block pointers cover runs of blocks
  - Similar approach in Linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability
  - Discussed Today!

Recall: File System Buffer Cache



- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap

Recall: Delayed Writes

- Buffer cache is a writeback cache (writes are termed "Delayed Writes")
- `write()` copies data from user space to kernel buffer cache
  - Quick return to user space
- `read()` is fulfilled by the cache, so `read`s see the results of `write`s
  - Even if the data has not reached disk
- When does data from a `write` syscall finally reach disk?
  - When the buffer cache is full (e.g., we need to evict something)
  - When the buffer cache is flushed periodically (in case we crash)

Recall: Delayed Writes (Advantages)

- Performance advantage: return to user quickly without writing to disk!

- Disk scheduler can efficiently order lots of requests
  - Elevator Algorithm can rearrange writes to avoid random seeks
- Delay block allocation:
  - May be able to allocate multiple blocks at same time for file, keep them contiguous
- Some files never actually make it all the way to disk
  - Many short-lived files!

Recall: Dealing with Persistent State

- Buffer Cache: write back dirty blocks periodically, even if used recently
  - Why? To minimize data loss in case of a crash
  - Linux does periodic flush every 30 seconds
- Not foolproof! Can still crash with dirty blocks in the cache
  - What if the dirty block was for a directory?
    - » Lose pointer to file's inode (leak space)
    - » File system now in inconsistent state ☹

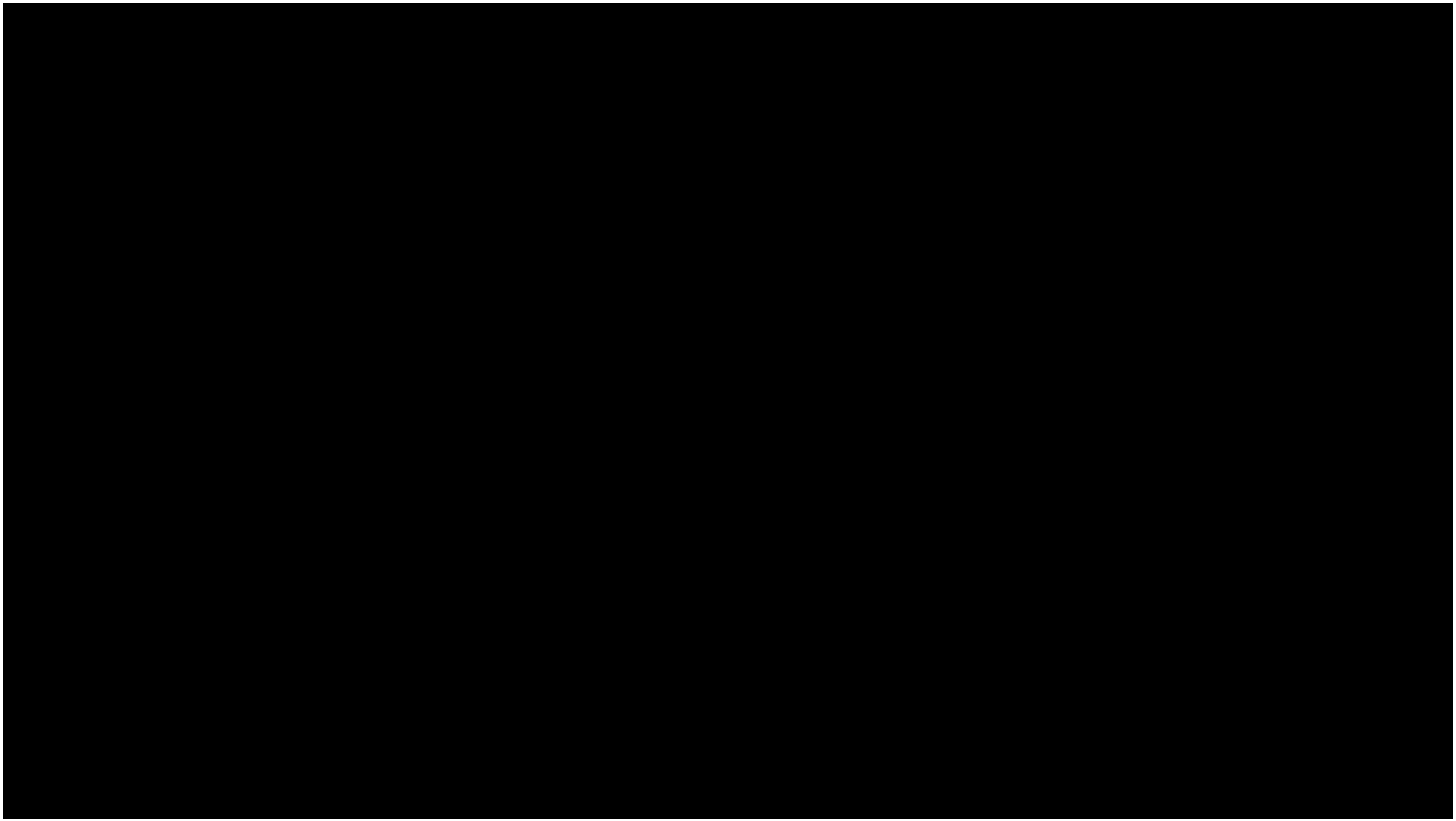
Takeaway: File systems need recovery mechanisms

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Measured in "nines" of probability: e.g., 99.9% probability is "3-nines of availability"
  - Key idea here is independence of failures

- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone

- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply **availability**: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems



HOW TO MAKE FILE SYSTEMS MORE *DURABLE*?

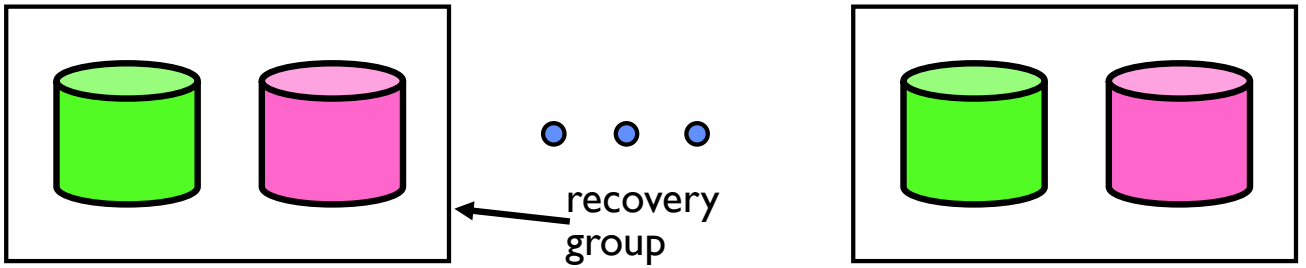
How to Make File Systems more Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects

- Make sure writes survive in short term
  - Either abandon delayed writes or
  - Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache

- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…

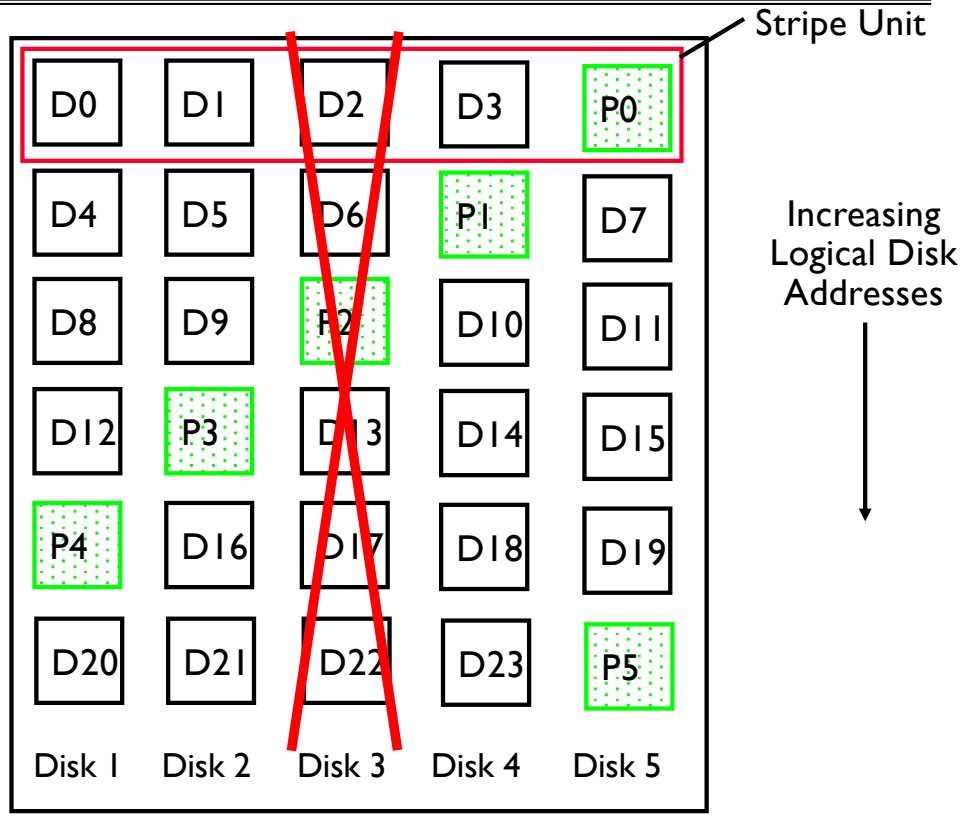
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation synchronized (challenging)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk attached to system for immediate replacement

RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
- Suppose Disk 3 fails, then can reconstruct: D2=D0⊕D1⊕D3⊕P0



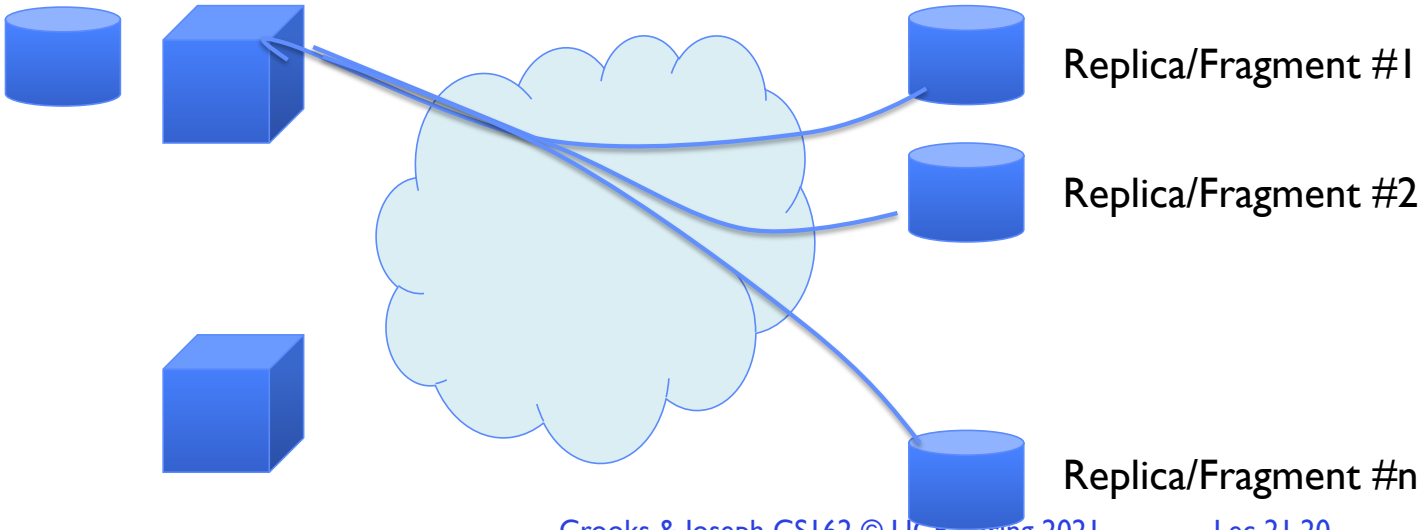
- Can spread information widely across internet for durability
  - RAID algorithms work over geographic scale

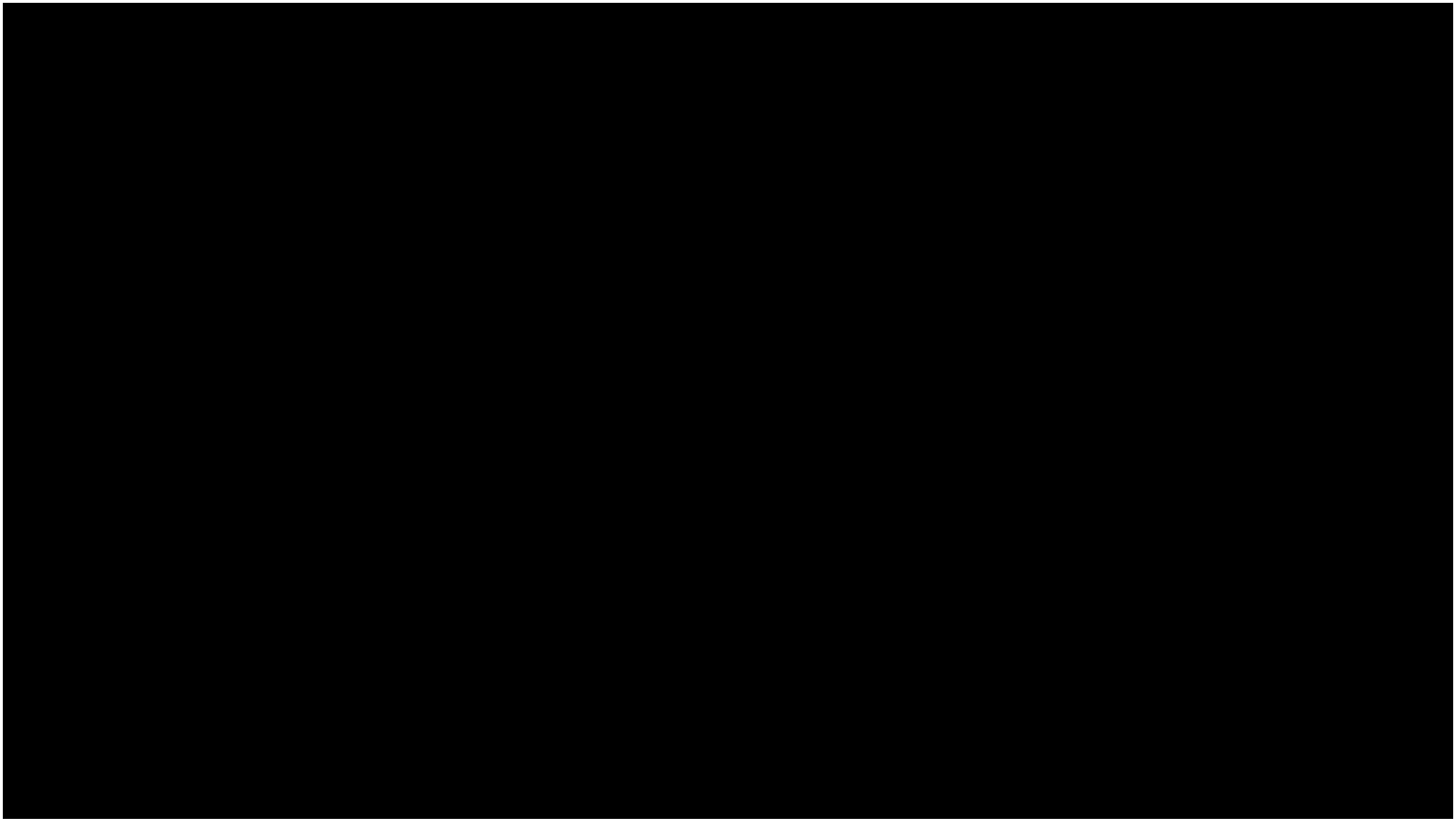
RAID 6 and other Erasure Codes

- In general: RAID X is an "erasure code"
  - Must have ability to know which disks are bad: Treat missing disk as an "Erasure"
- Today, disks so big that: RAID 5 not sufficient!
  - Time to repair disk is sooooo long, another disk might fail in process!
  - "RAID 6" – allow 2 disks in replication stripe to fail
  - Requires more complex erasure code, such as EVENODD code (see readings)
- More general option for general erasure code: Reed-Solomon codes
  - Based on polynomials in GF(2^k) (i.e., k-bit symbols)
  - m data points define a degree m polynomial; encoding is n points on the polynomial
  - Any m points can be used to recover the polynomial; $n - m$ failures tolerated
- Erasure codes not just for disk arrays. For example, geographic replication
  - E.g., split data into $m = 4$ chunks, generate $n = 16$ fragments and distribute across the Internet
  - Any 4 fragments can be used to recover the original data – very durable!

Higher Durability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads
  - Simple replication: read any copy
  - Erasure coded: read m of n
- Low availability for writes
  - Can't write if any one replica is not up
  - Or – need relaxed consistency model
- Reliability? – availability, security, durability, fault-tolerance





HOW TO MAKE FILE SYSTEMS MORE *RELIABLE*?

File System Reliability: (Difference from Block-level reliability)

- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete

- Having RAID doesn't necessarily protect against all such failures
  - No protection against writing bad state
  - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

- But durability is not quite enough…!

Storage Reliability Problem

- A single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors

- At a physical level, operations complete one at a time
  - Want concurrent operations for performance

- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
  - Example: transfer funds from one bank account to another
  - What if transfer is interrupted after withdrawal and before deposit?

- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Two Reliability Approaches

Careful Ordering and Recovery

- FAT & FFS + (fsck)
- Each step builds structure,
- Data block ⇐ inode ⇐ free ⇐ directory
- Last step links it into rest of FS
- Recover scans structure looking for incomplete actions

Versioning and Copy-on-Write

- ZFS, …
- Version files at some granularity
- Create new structure linking back to unchanged parts of old
- Last step is to declare that the new version is ready

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed
- Approach taken by
  - FAT and FFS (fsck) to protect filesystem structure/metadata
  - Many app-level recovery schemes (e.g., Word, emacs autosaves)

Berkeley FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

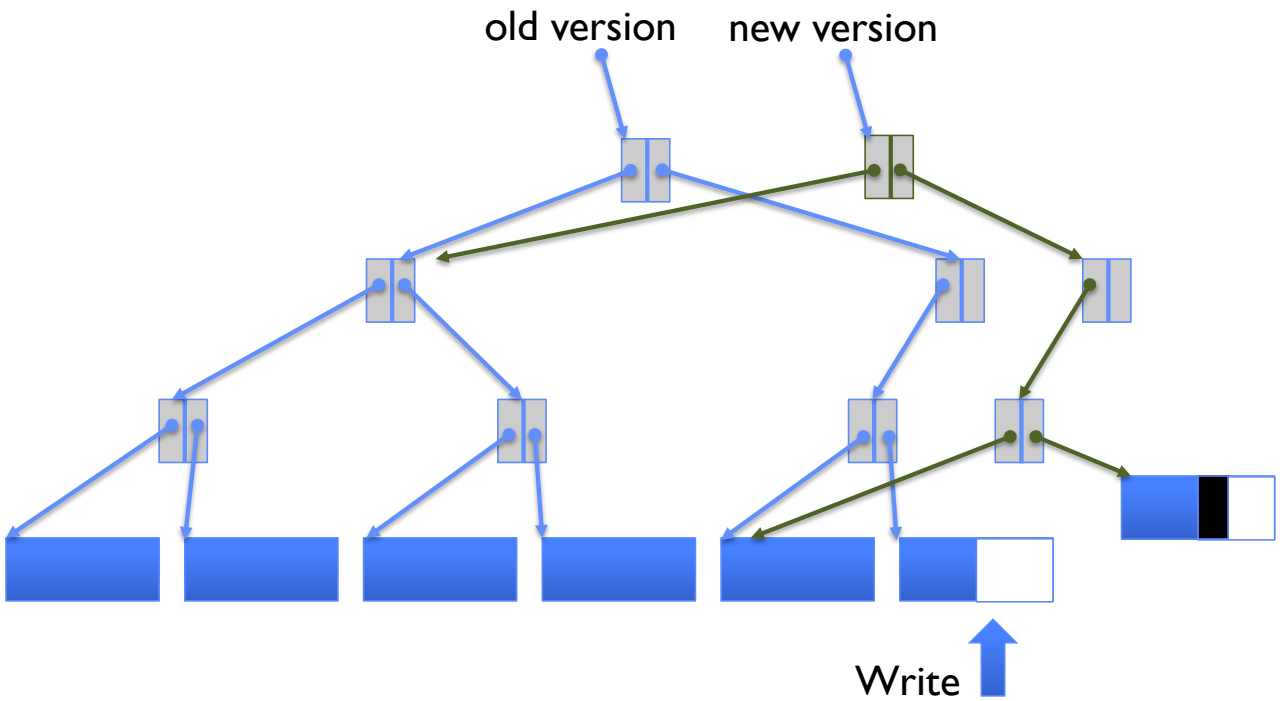
Reliability Approach #2: Copy on Write File Layout

- Recall: multi-level index structure lets us find the data blocks of a file
- Instead of over-writing existing data blocks and updating the index structure:
  - Create a new version of the file with the updated data
  - Reuse blocks that don't change much of what is already in place
  - This is called: Copy On Write (COW)

- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel

- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

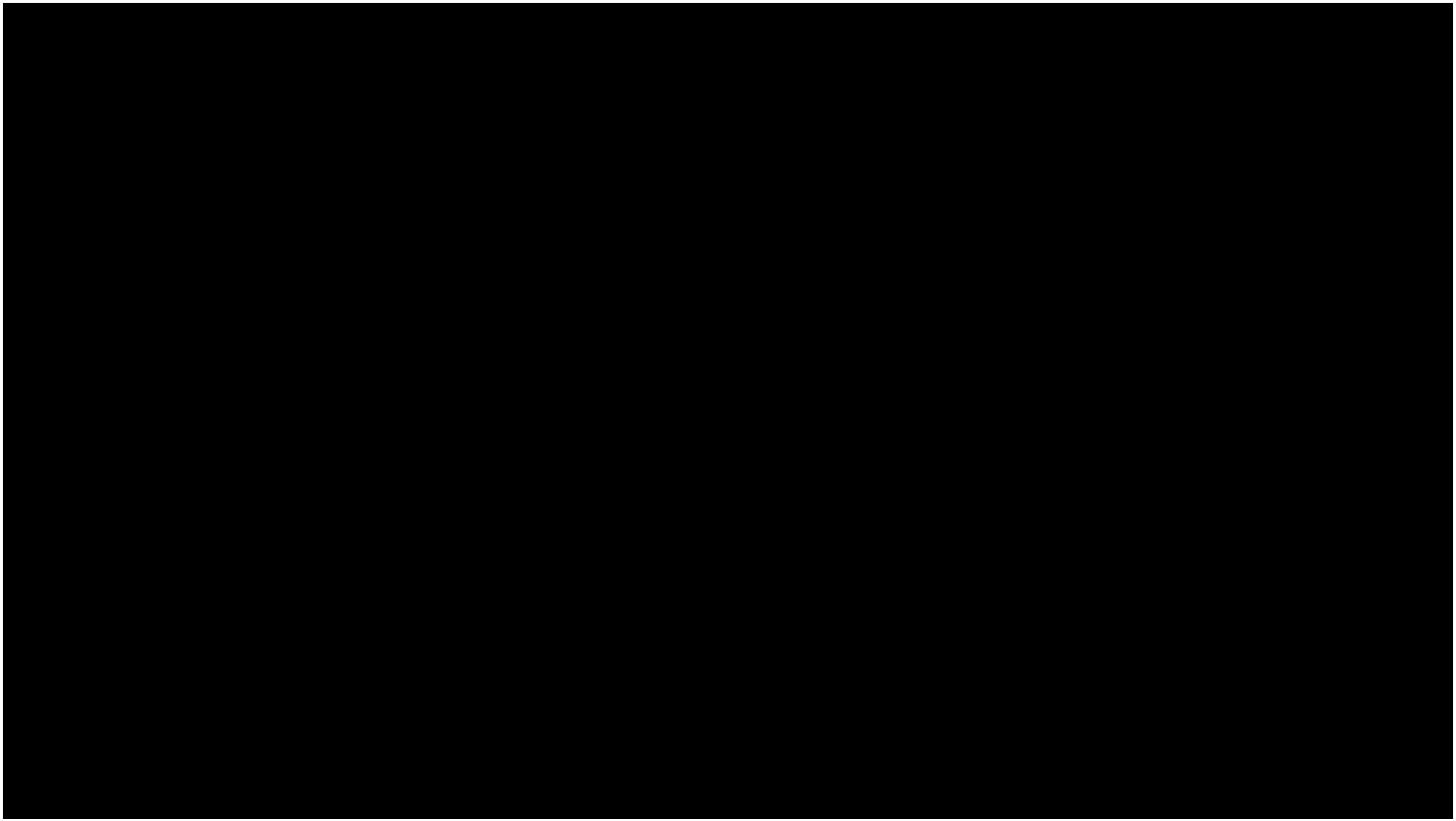
COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

Example: ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated



TRANSACTIONS

More General Reliability Solutions

- Use Transactions for atomic updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
  - Most modern file systems use transactions internally to update filesystem structures and metadata
  - Many applications implement their own transactions

- Provide Redundancy for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)

Transactions

- Closely related to critical sections for manipulating shared data structures
- They extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Many ad-hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
  - Applications use temporary files and rename

Key Concept: Transaction

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.

- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from *memory* to *persistent storage*

Typical Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

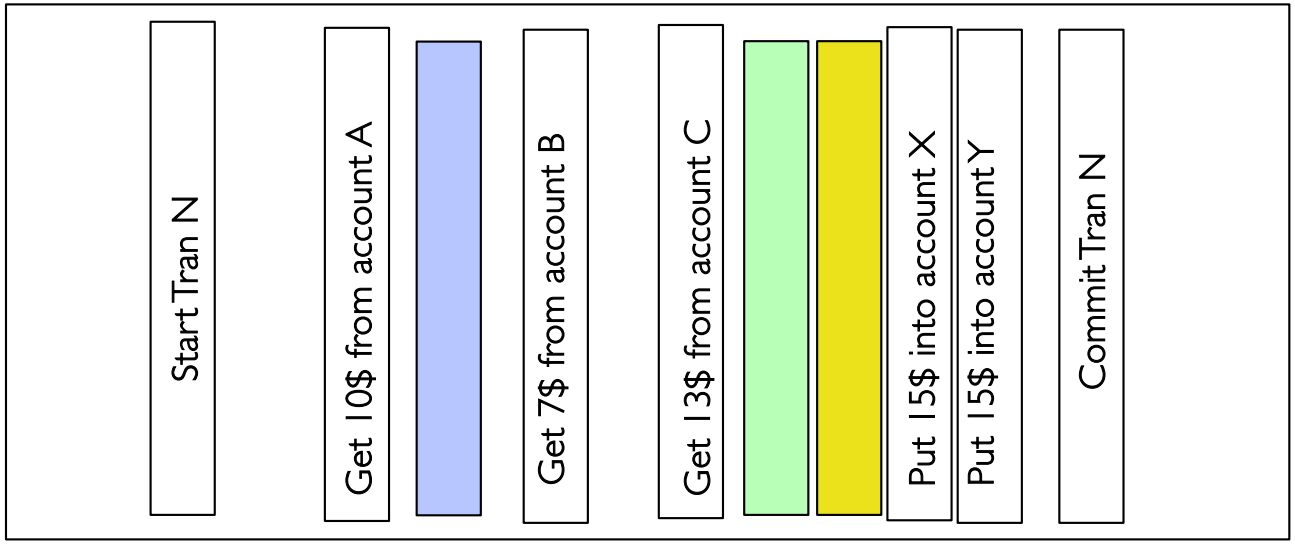
"Classic" Example: Transaction

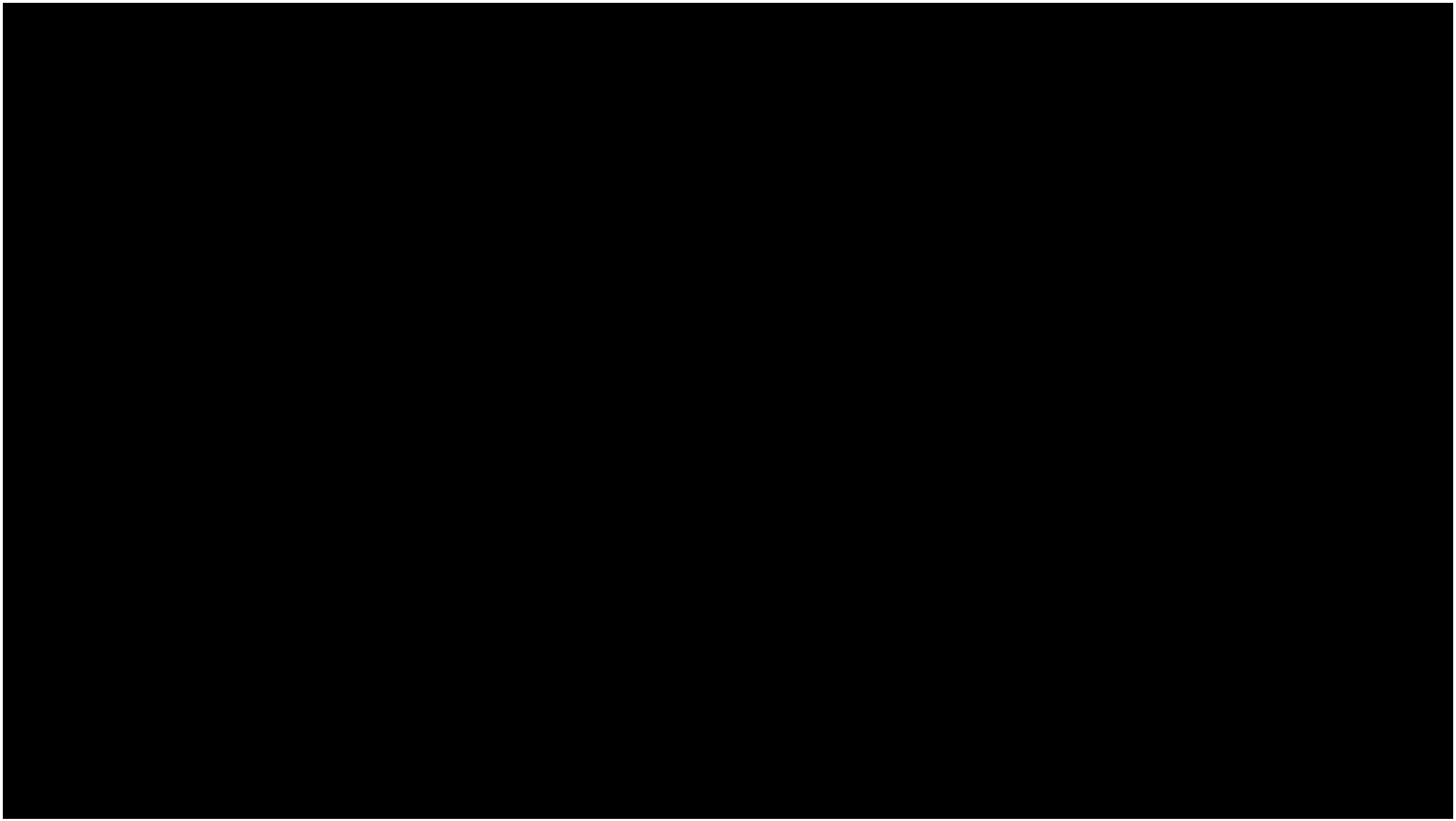
```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00 WHERE
   name = 'Alice';

 UPDATE branches SET balance = balance - 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Alice');

 UPDATE accounts SET balance = balance + 100.00 WHERE
   name = 'Bob';

 UPDATE branches SET balance = balance + 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Bob');
COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions





TRANSACTIONAL FILESYSTEMS

Transactional Filesystems

- Better reliability through use of log
  - Changes are treated as transactions
  - A transaction is committed once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log

- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery

Journaling File Systems

- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a journal or intention list
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied to file system
  - e.g., modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

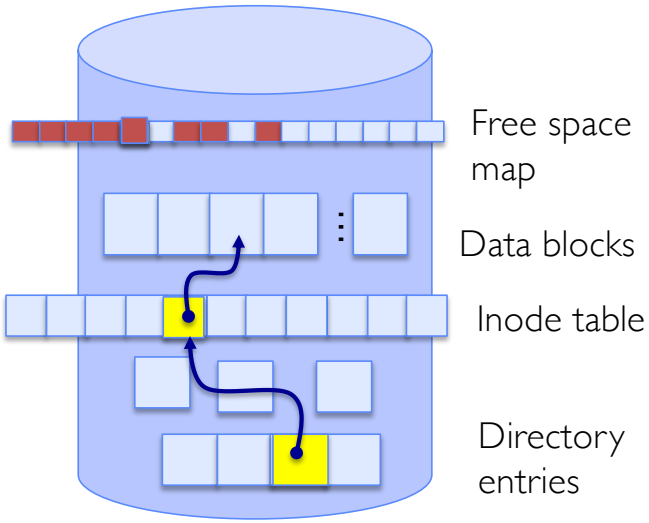
- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
  - Some options: whether or not to write all data to journal or just metadata

- Other examples: NTFS, Apple HFS+, Linux XFS, JFS, ext4

Creating a File (No Journaling Yet)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

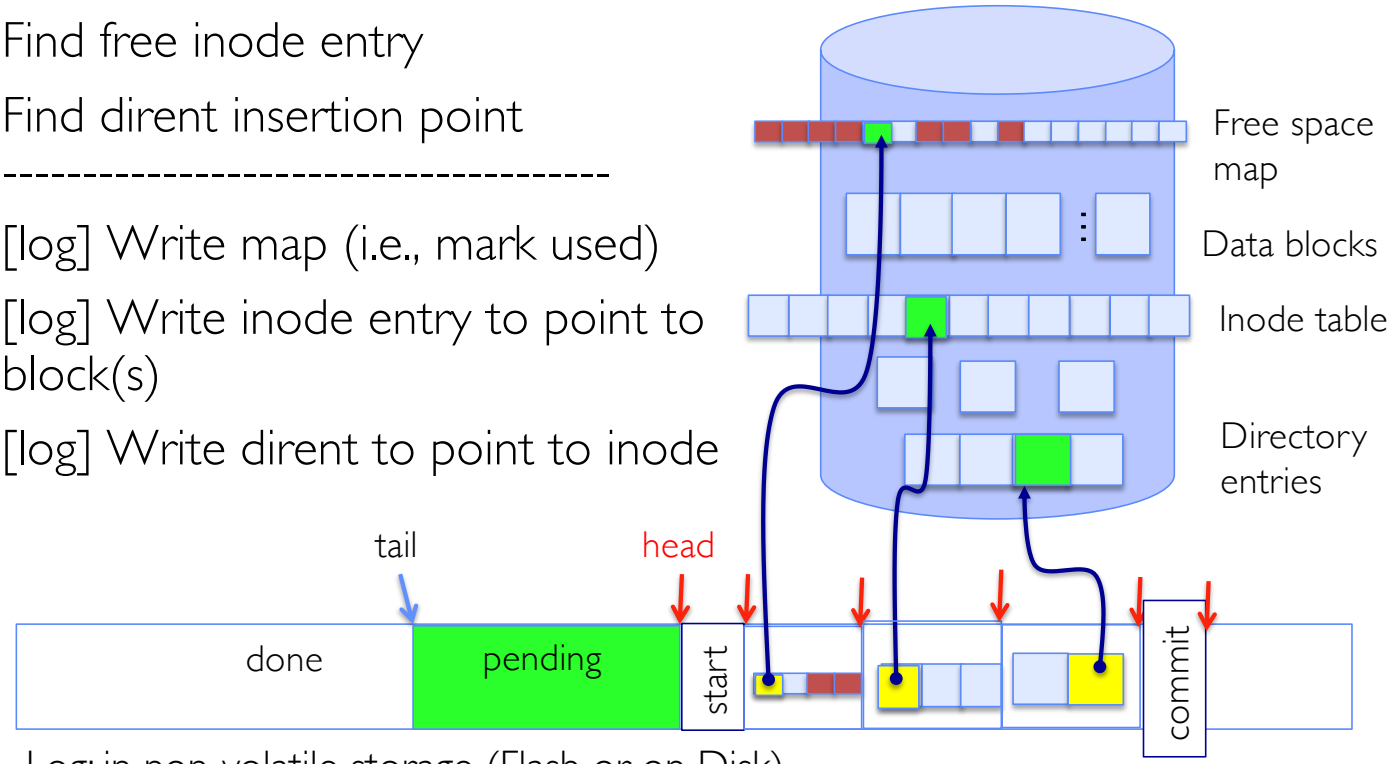
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Creating a File (With Journaling)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

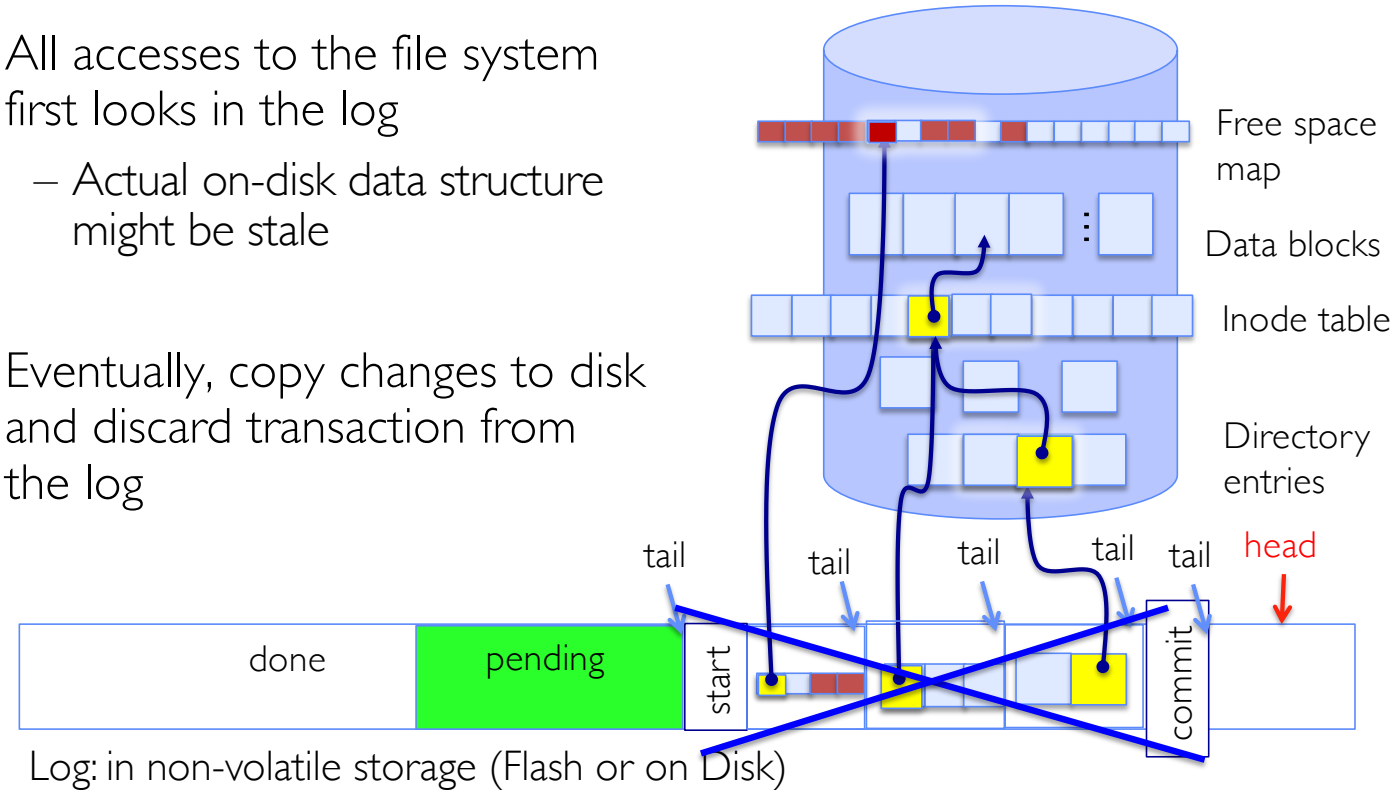
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



Log: in non-volatile storage (Flash or on Disk)

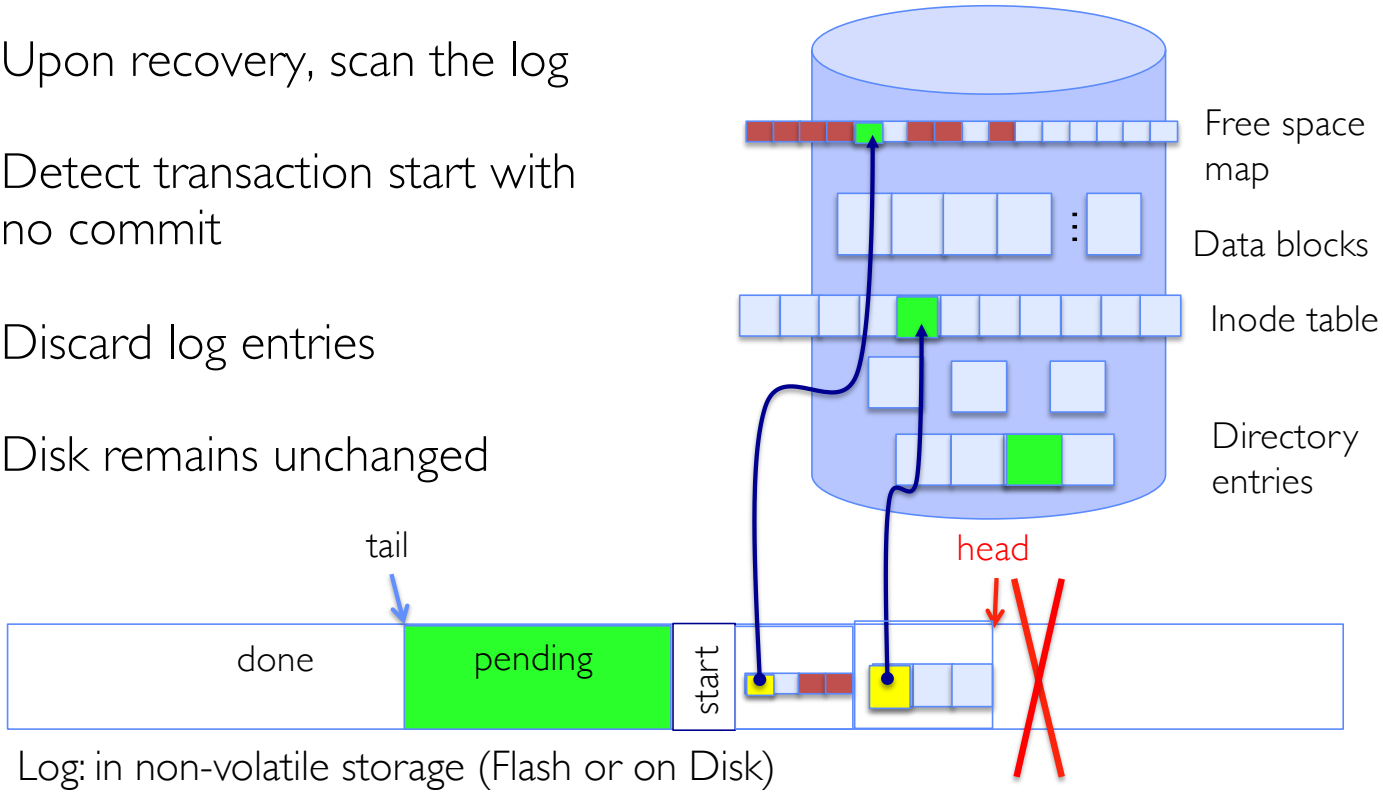
After Commit, Eventually Replay Transaction

- All accesses to the file system first looks in the log
  - Actual on-disk data structure might be stale
- Eventually, copy changes to disk and discard transaction from the log



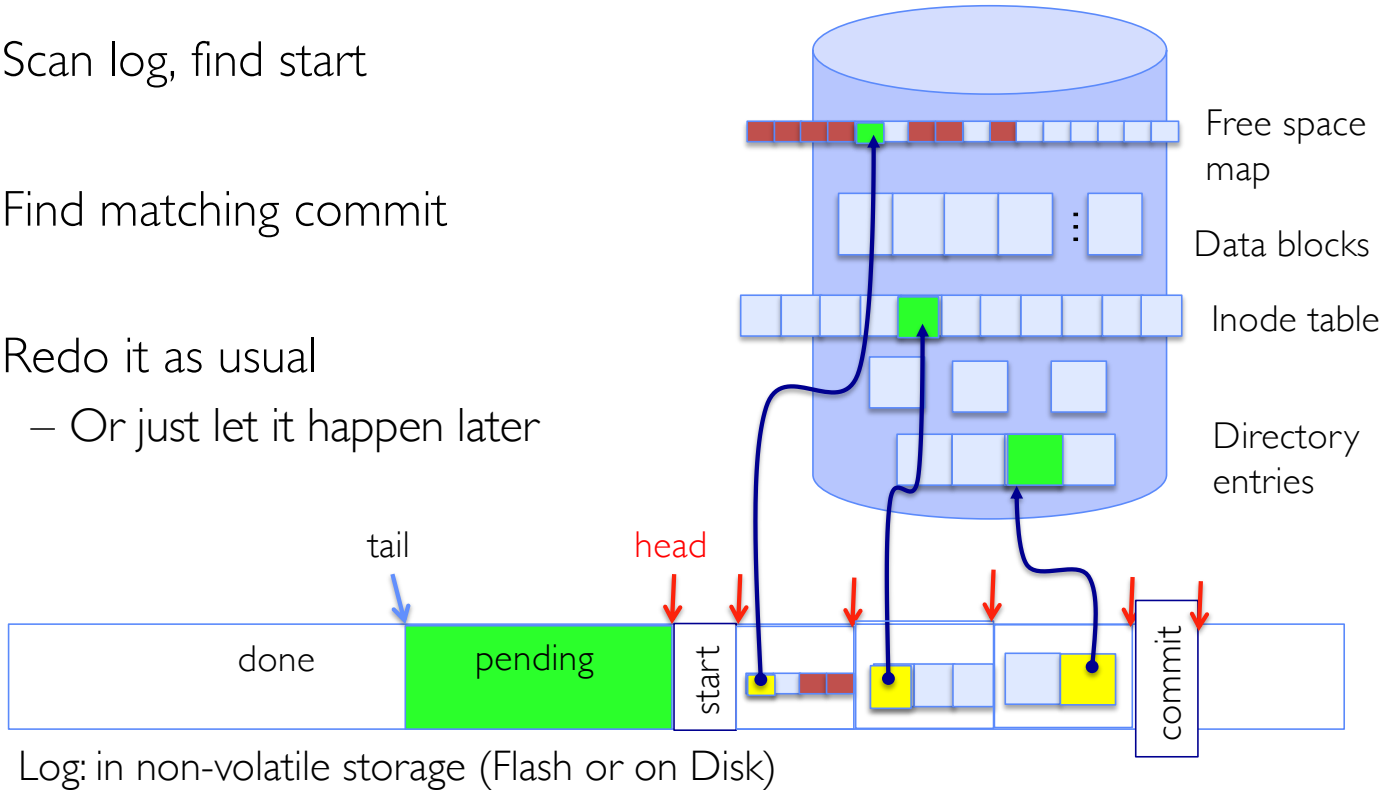
Crash Recovery: Discard Partial Transactions

- Upon recovery, scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



Crash Recovery: Keep Complete Transactions

- Scan log, find start
- Find matching commit
- Redo it as usual
  - Or just let it happen later



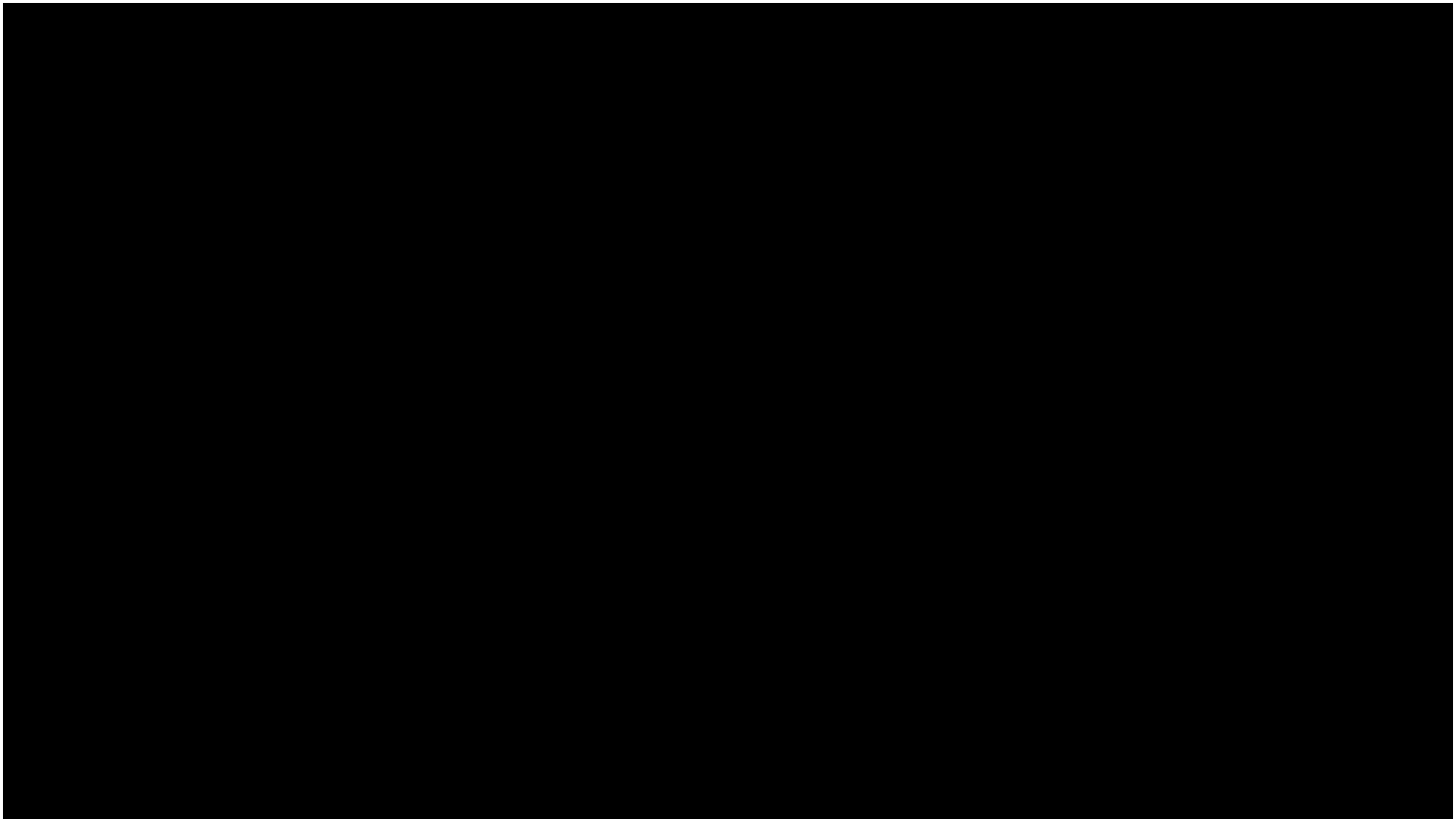
Journaling Summary

Why go through all this trouble?

- Updates atomic, even if we crash:
  - Update either gets fully applied or discarded
  - All physical operations *treated as a logical unit*

Isn't this expensive?

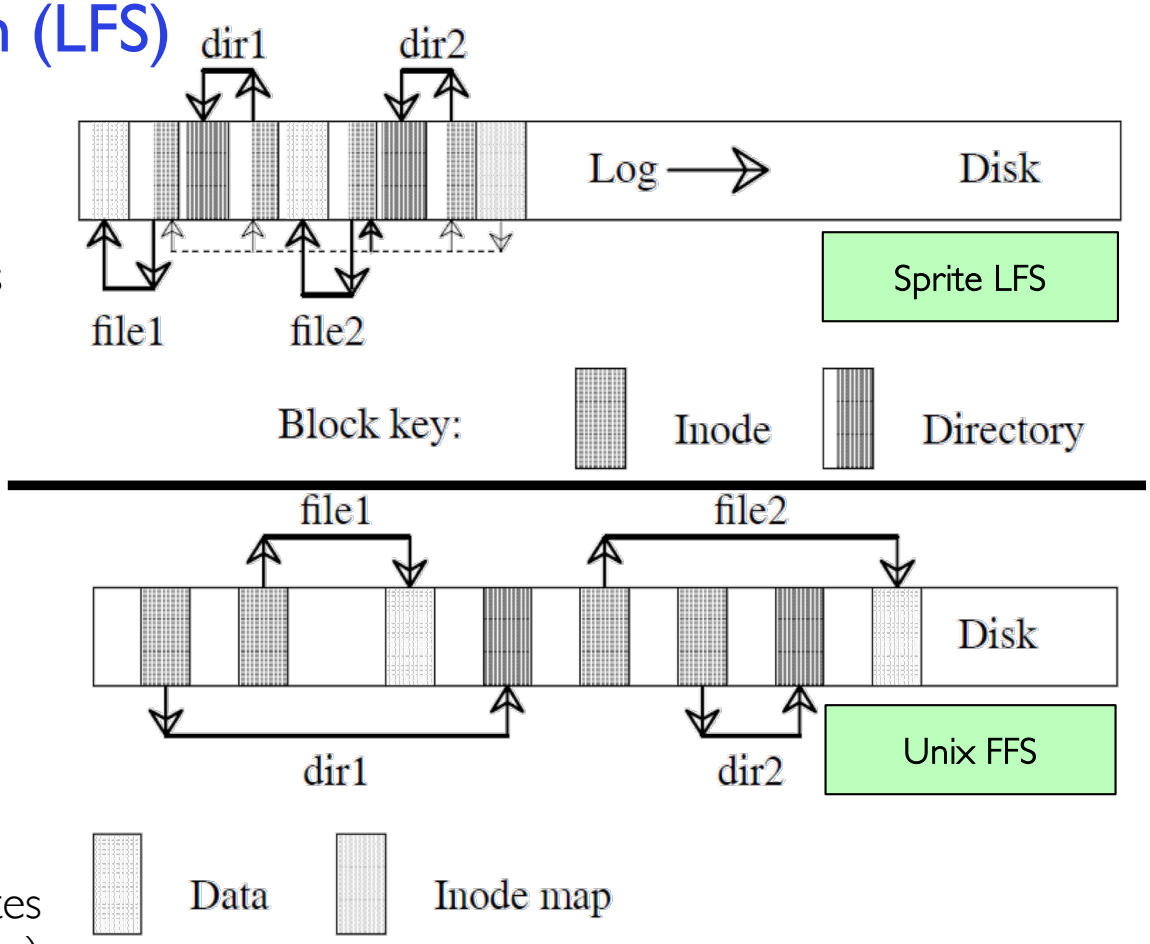
- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems journal metadata updates only
  - Record modifications to file system data structures
  - But apply updates to a file's contents directly



LOG STRUCTURED FILESYSTEMS

The Log Structured File System (LFS)

- Log Structured File System:
  - The LOG IS the storage
- Log: One continuous sequence of blocks that wrap around whole disk
  - Inodes put into log when changed and point to new data in the log
- Simple example:
  - Create two new files:
    - » dir1/file1 and dir2/file2
    - » Must write new data blocks for files and for new information in directories
  - LFS writes everything sequentially
  - Unix FFS requires 10 non-sequential writes (inodes written twice for ease of recovery)
- Paper on resources page!

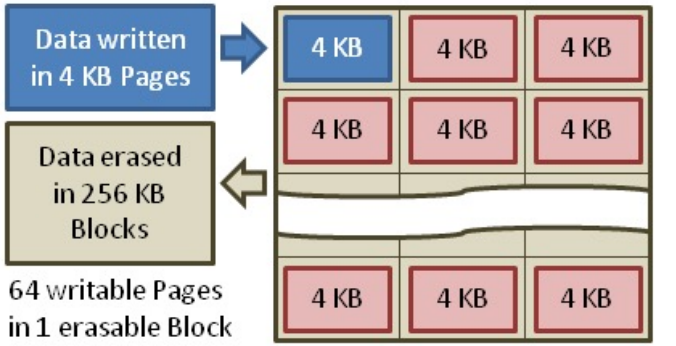


Going Further – Log Structured File Systems

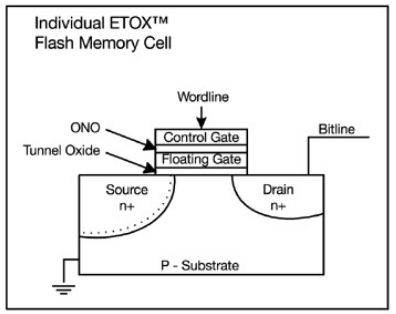
- The log IS what is recorded on disk
  - File system operations logically replay log to get result
  - Create data structures to make this fast
  - On recovery, replay the log
- Index (inodes) and directories are written into the log too
- Large, important portion of the log is cached in memory
  - Relies on Buffer Cache to make reading fast
- Do everything in bulk: log is collection of large segments
- Each segment contains a summary of all the operations within the segment
  - Fast to determine if segment is relevant or not
- Free space is approached as continual cleaning process of segments
  - Detect what is live or not within a segment
  - Copy live portion to new segment being formed (replay)
  - Garbage collection entire segment
  - No bit map

What about Flash Filesystems?

- Cannot overwrite pages!
  - Must move contents to an erased page
  - Small changes ⇒ lots of rewriting of data/wear out!
- Program/Erase (PE) Wear
  - Permanent damage to gate oxide at each flash cell
  - Caused by high program/erase voltages
  - Issues: trapped charges, premature leakage of charge
  - *Need to balance how frequently cells written: "Wear Leveling"*
- Flash Translation Layer (FTL)
  - Translates between Logical Block Addresses (at OS level) and Physical Flash Page Addresses
  - Manages the wear and erasure state of blocks and pages
  - Tracks which blocks are garbage but not erased
- Management Process (Firmware)
  - Keep freelist full, Manage mapping,
  - Track wear state of pages
  - Copy good pages out of mostly empty blocks before erasure



Typical NAND Flash Pages and Blocks

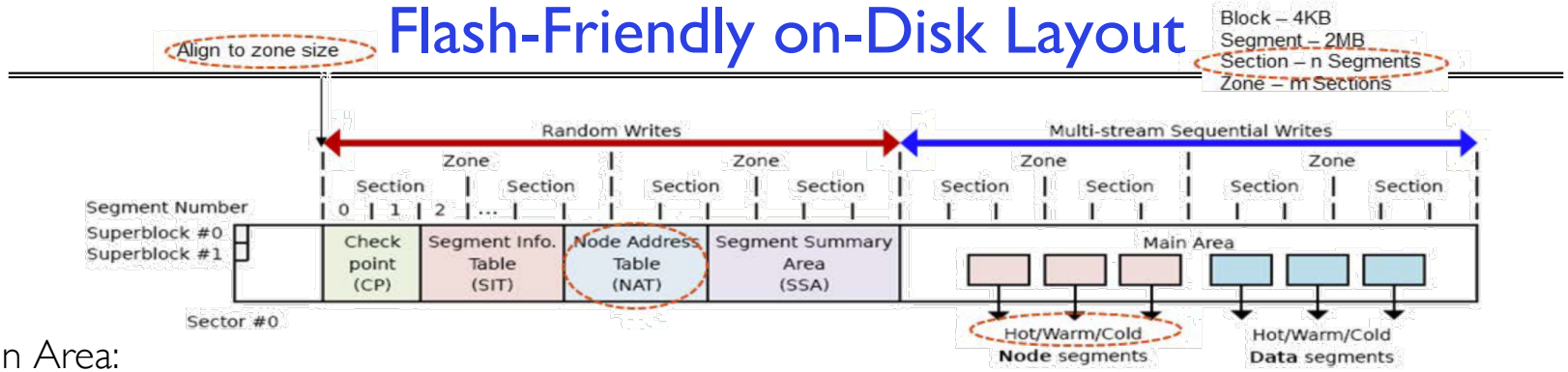


Single FLASH storage bit

Example Use of LFS: F2FS: A Flash File System

- File system used on many mobile devices
  - Including the Pixel 3 from Google
  - Latest version supports block-encryption for security
  - Has been "mainstream" in Linux for several years now
- Assumes standard SSD interface
  - With built-in Flash Translation Layer (FTL)
  - Random reads are as fast as sequential reads
  - Random writes are bad for flash storage
    - » Forces FTL to keep moving/coalescing pages and erasing blocks
    - » Sustained write performance degrades/lifetime reduced
- Minimize Writes/updates and otherwise keep writes "sequential"
  - Start with Log-structured file systems/copy-on-write file systems
  - Keep writes as sequential as possible
  - Node Translation Table (NAT) for "logical" to "physical" translation
    - » Independent of FTL
- For more details, check out paper in Readings section of website
  - "F2FS: A New File System for Flash Storage" (from 2015)
  - Design of file system to leverage and optimize NAND flash solutions
  - Comparison with Ext4, Btrfs, Nilfs2, etc

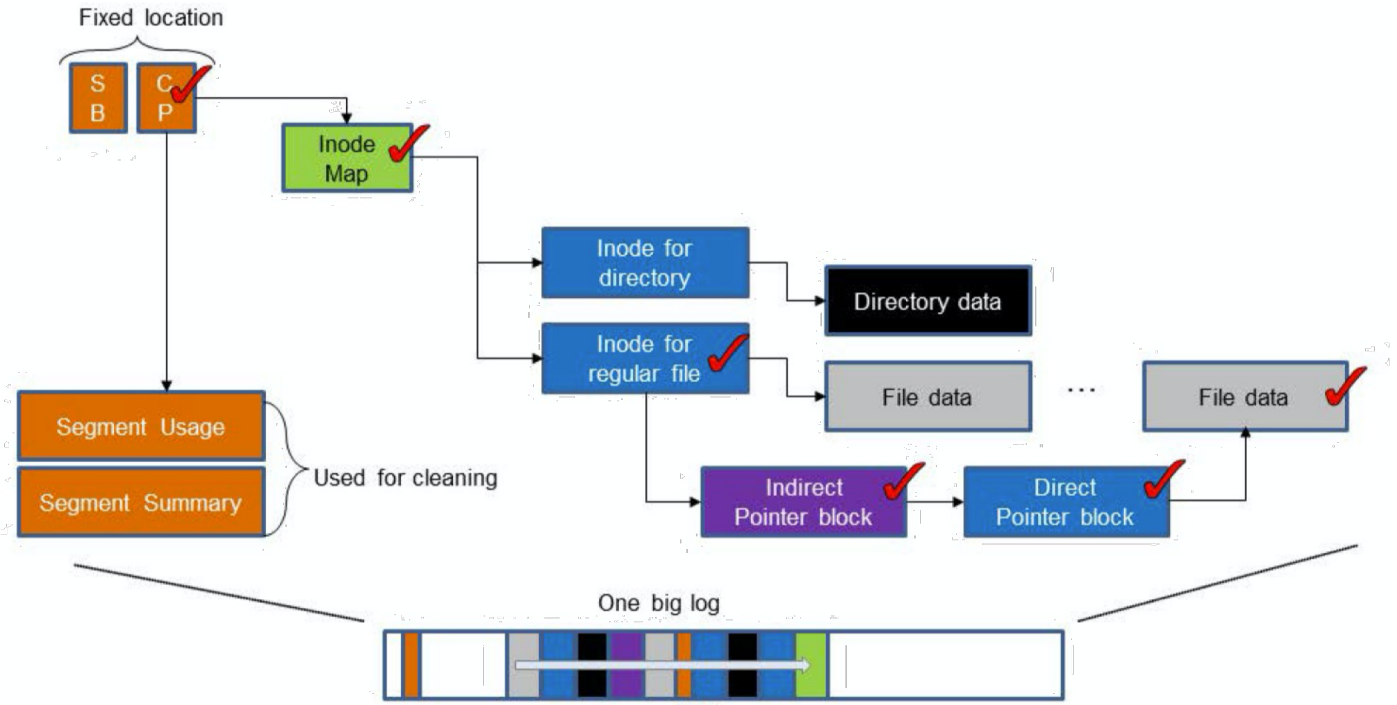
Flash-Friendly on-Disk Layout



- Main Area:
  - Divided into segments (basic unit of management in F2FS)
  - 4KB Blocks. Each block typed to be *node* or *data.*
- Node Address Table (NAT): *Independent of FTL!*
  - Block address table to locate all "node blocks" in Main Area
- Updates to data sorted by predicted write frequency (Hot/Warm/Cold) to optimize FLASH management
- Checkpoint (CP): Keeps the file system status
  - Bitmaps for valid NAT/SIT sets and Lists of orphan inodes
  - Stores a consistent F2FS status at a given point in time
- Segment Information Table (SIT):
  - Per segment information such as number of valid blocks and the bitmap for the validity of all blocks in the "Main" area
  - Segments used for "garbage collection"
- Segment Summary Area (SSA):
  - Summary representing the owner information of all blocks in the Main area

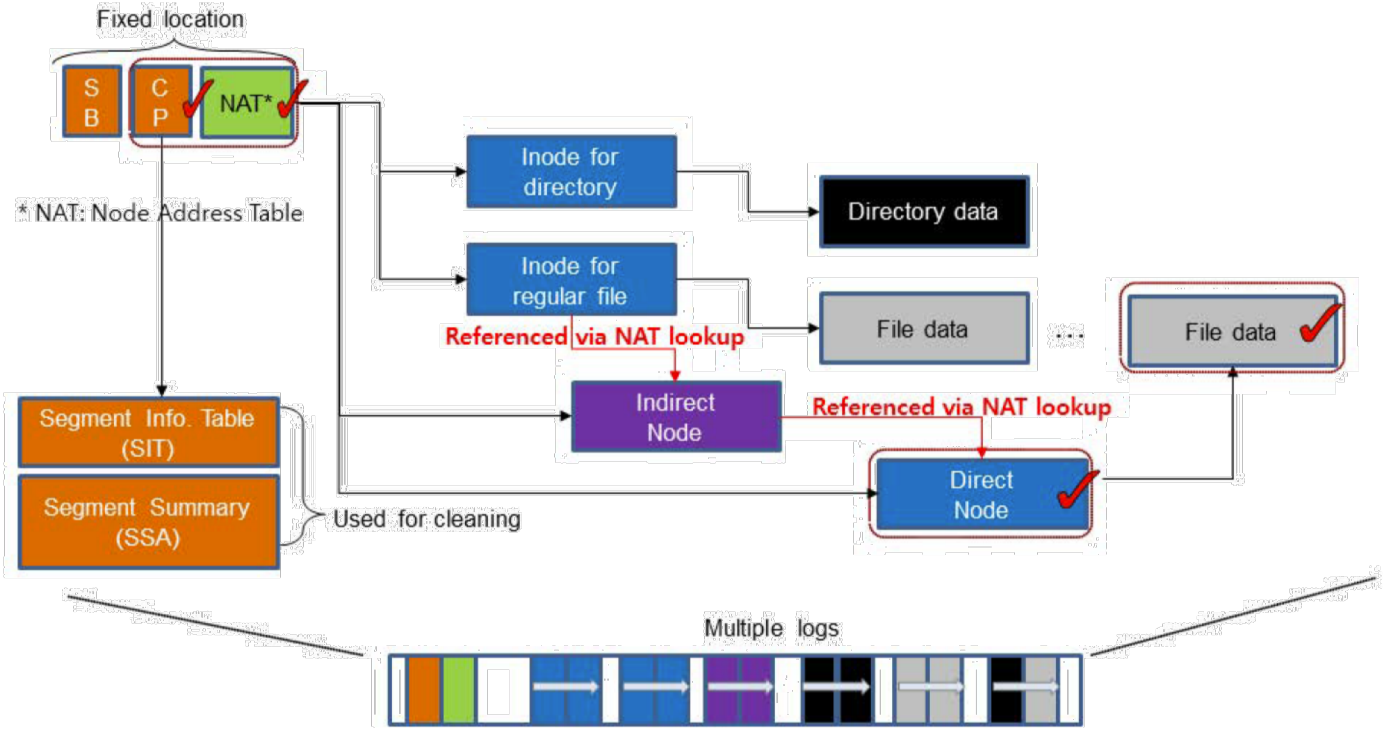
Normal LFS Index Structure: Forces Cascading Updates when Updating Data

- **Update propagation issue: wandering tree**
- **One big log**



F2FS Index Structure: Indirection and Multi-Head Logs Optimize Updates

- **Restrained update propagation: node address translation method**
- **Multi-head log**



Summary (1/2)

- File system operations involve multiple distinct updates to blocks on disk
  - Need to have all or nothing semantics
  - Crash may occur in the midst of the sequence
- Traditional file system perform check and recovery on boot
  - Along with careful ordering so partial operations result in loose fragments, rather than loss
- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?

Summary (2/2)

- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free
- Use of Log to improve Reliability
  - Journaled file systems such as ext3, NTFS
- Transactions over a log provide a general solution
  - Commit sequence to durable log, then update the disk
  - Log takes precedence over disk
  - Replay committed transactions, discard partials