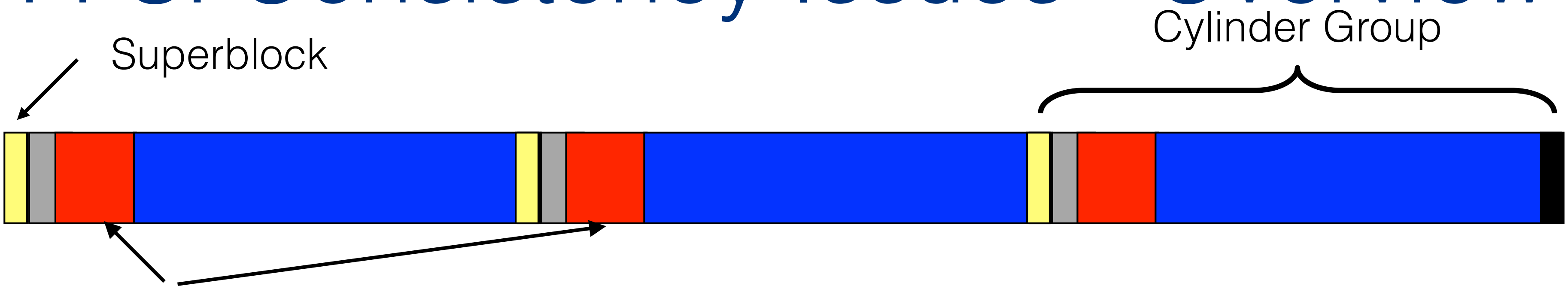
Reliability and Write Optimizations

Reliability and Write Optimizations

- How do we guarantee consistency of on-disk storage?
- How do we handle OS crashes and disk errors?
- How do we optimize writes?

FFS: Consistency Issues - Overview



- Inodes: fixed size structure stored in cylinder groups

- Metadata updates must be synchronous operations. Why?

- File system operations affect multiple metadata blocks
  - Write newly allocated inode to disk before its name is entered in a directory.
  - Remove a directory name before the inode is deallocated
  - Deallocate an inode (mark as free in bitmap) before that file's data blocks are placed into the cylinder group free list.

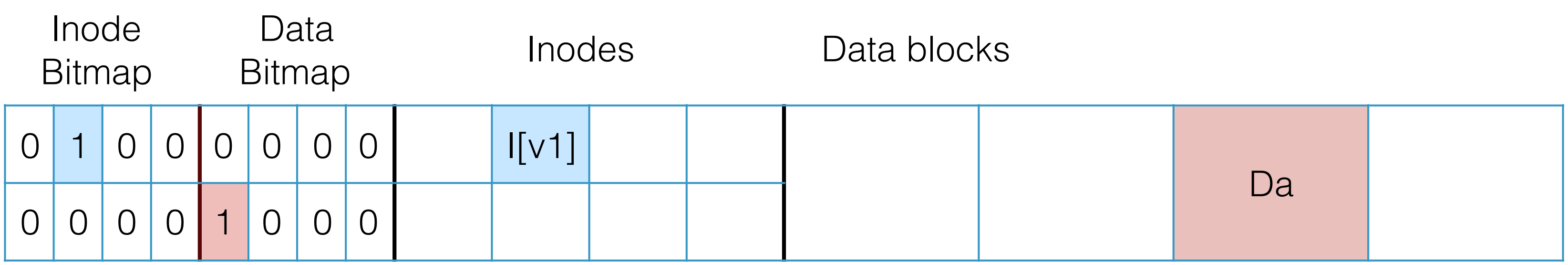
FFS Observation 1: Crash recovery

- If the OS crashes in between any of these synchronous operations, then the file system is in an inconsistent state.

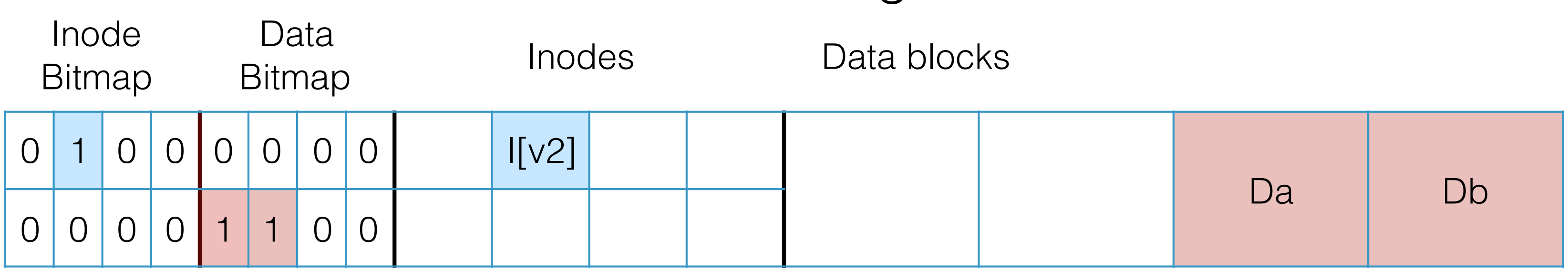
- Solutions (overview):
  - fsck – post-crash recovery process to scan file system structure and restore consistency
    - All data blocks pointed to by inodes (and indirect blocks) must be marked allocated in the data bitmap
    - All allocated inodes must be in some dir entries
    - Inode link count must match
  - Log updates to enable roll-back or roll-forward

Example: update

- Consider a simple update: append 1 data block to a file
- Assume a similar FS structure as seen before:



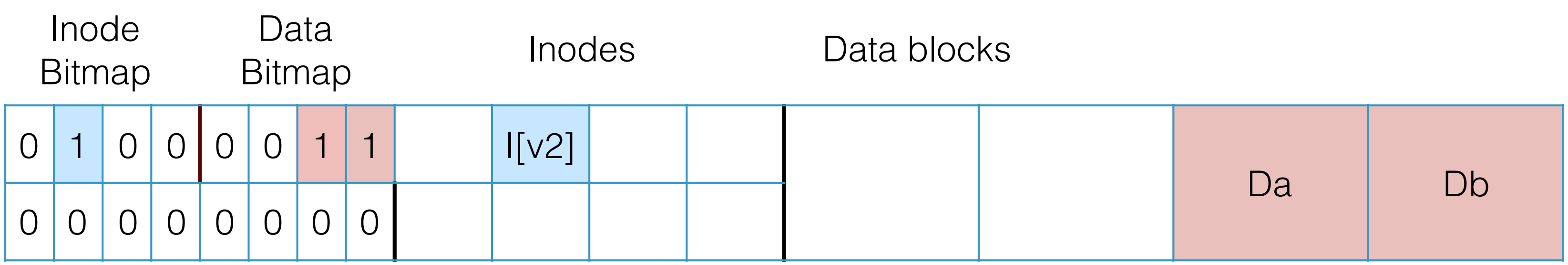
- Add a data block: Db .. What changes?

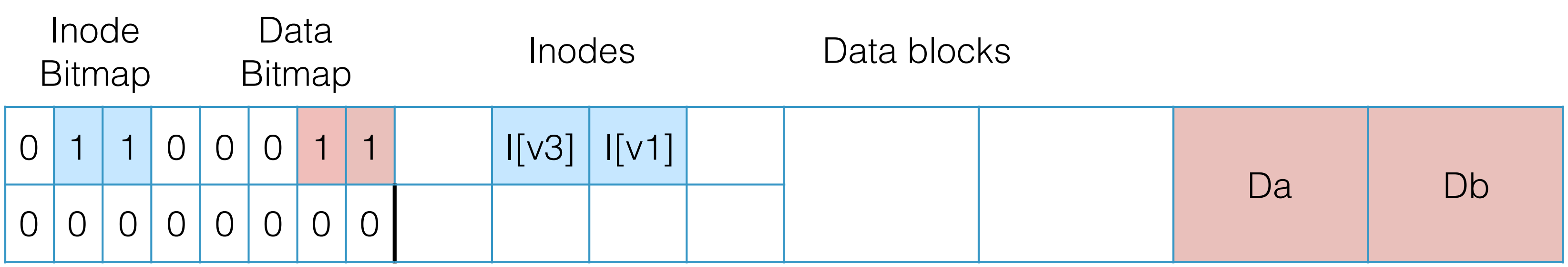


- Three writes: I[v2], Data Bitmap, Db

Example2: create file

- Create a new **empty** file in the first block of the directory that Inode 1 represents





- Writes: I[v3], I[v1], Inode Bitmap, Data block

Crash Consistency

- What if only one write succeeds before a crash?

1. Just Db write succeeds

2. Just I[v2] write succeeds

3. Just B[v2] write succeeds

Other Crash Scenarios

- What if only two writes succeed before a crash?

1. Only I[v2] and Data Bitmap writes succeed.

2. Only I[v2] and Db writes succeed.

3. Only Data Bitmap and Db writes succeed.

Solution #1 fsck

- fsck: UNIX tool for finding inconsistencies and repairing them
- Cannot fix all problems!
  - When Db is garbage – cannot know that's the case
  - Only cares that FS metadata is consistent!
- Similar tools exist on other systems

fsck

- What does it check?
  - 1. Superblock: sanity checks
    - Use another superblock copy if suspected corruption
  - 2. Free blocks: scan inodes (incl. all indirect blocks), build bitmap
    - inodes / data bitmaps inconsistency => resolve by trusting inodes
    - Ensure inodes in use are marked in inode bitmaps
  - 3. Inode state: check inode fields for possible corruption
    - e.g., must have a valid “mode” field (file, dir, link, etc.)
    - If cannot fix => remove inode and update inode bitmap
  - 4. Inode links: verify links# for each inode
    - Traverse directory tree, compute expected links#, fix if needed
    - If inode discovered, but no dir refers to it => move to “lost+found”

5. Duplicates: check if two different inodes refer to same block
   - Clear one if obviously bad, or, give each inode its own copy of block
6. Bad blocks: bad pointers (outside of valid range)
   - Just remove the pointer from the inode or indirect block
7. Directory checks: integrity of directory structure
   - E.g., make sure that "." and ".." are the first entries, each inode in a directory entry is allocated, no directory is linked more than once

fsck limitations

- So, fsck helps ensure integrity
- Only FS integrity, cannot do anything about lost data!
- Bigger problem: too slow!
  - Disks are very large nowadays – scanning all this could take hours!
  - Even for small inconsistency, must scan whole disk!

Alternative solution: Journaling

- Aka Write-Ahead-Logging

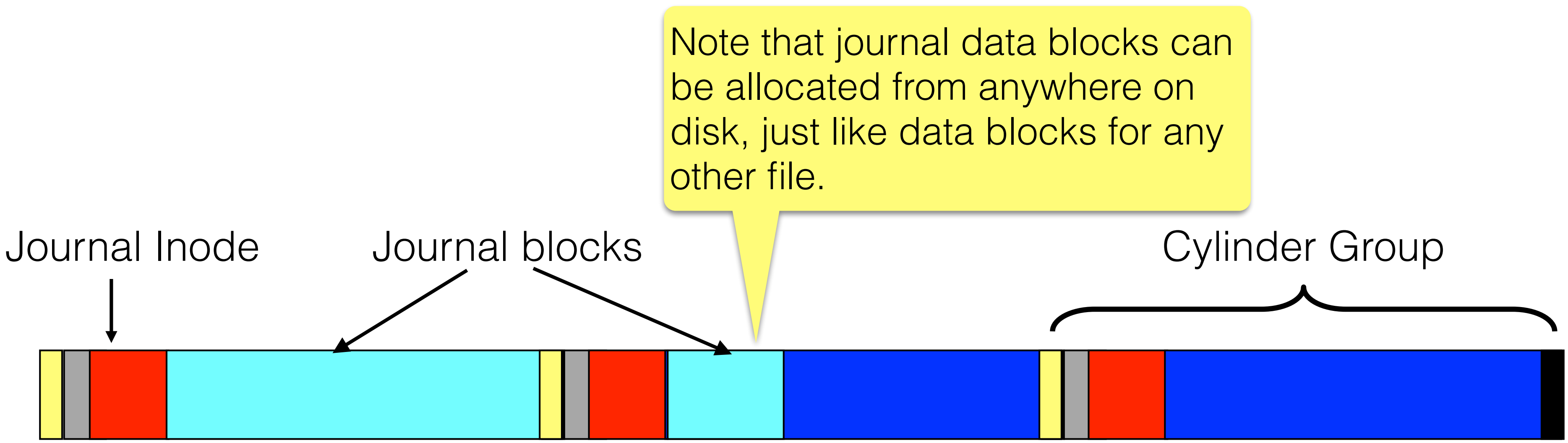
- Basic idea:
  - When doing an update, before overwriting structures, first write down a little note (elsewhere on disk) saying what you plan to do.
  - i.e., “Log” the operations you are about to do.

- If a crash takes place during the actual write => go back to journal and retry the actual writes.
  - Don’t need to scan the entire disk, we know what to do!
  - Can recover data as well

- If a crash happens before journal write finishes, then it doesn’t matter since the actual write has NOT happened at all, so nothing is inconsistent.

Linux Ext3 File System

- Extends ext2 with journaling capabilities
  - Backwards and forwards compatible
    - Identical on-disk format
  - Journal can be just another large file (inode, indirect blocks, data blocks)



What goes in the “log”

- Transaction structure:
  - Starts with a “transaction begin” (TxBegin) block, containing a transaction ID
  - Followed by blocks with the content to be written
    - Physical logging: log exact physical content
    - Logical logging: log more compact logical representation
  - Ends with a “transaction end” (TxEnd) block, containing the corresponding TID

Journal Entry	TxBegin (TID=1)	Updated inode	Updated Bitmap	Updated Data block	TxEnd (TID=1)

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

- Sequence of operations

  1. Write the transaction (containing Iv2, Bv2, Db) to the log

  2. Write the blocks (Iv2, Bv2, Db) to the file system

  3. Mark the transaction free in the journal

- Crash may happen at any point!

  - If between 1 and 2 => on reboot, replay non-free transactions (called redo logging)

  - If during writes to the journal (step 1) => tricky!

Data Journaling Example

- One solution: write each block at a time
  - Slow!
  - Ideally issue multiple blocks at once.
  - Unsafe though! What could happen?
  - Normal operation: Blocks get written in order, power cuts off before TxEnd gets written => We know transaction is not valid, no problem.

TxBegin	I[v2]	B[v2]	Db	???

  - However, Internal disk scheduling: TxBegin, Iv2, Bv2, TxEnd, Db
  - Disk may lose power before Db written

TxBegin	I[v2]	B[v2]	???	TxEnd

- Problem: Looks like a valid transaction!

Data Journaling Example

To avoid this, split into 2 steps

1. Write all except TxEnd to journal (Journal Write step)

TxBegin	I[v2]	B[v2]	Db

2. Write TxEnd (only once 1. completes) (Journal Commit step)
   => final state is safe!

TxBegin	I[v2]	B[v2]	Db	TxEnd

3. Finally, now that journal entry is safe, write the actual data and metadata to their right locations on the FS (Checkpoint step)

4. Mark transaction as free in journal (Free step)

Journaling: Recovery Summary

- If crash happens before the transaction is committed to the journal
  - Just skip the pending update
- If crash happens during the checkpoint step
  - After reboot, scan the journal and look for committed transactions
  - Replay these transactions
  - After replay, the FS is guaranteed to be consistent
  - Called redo logging

Journal Space Requirements

- How much space do we need for the journal?
  - For every update, we log to the journal => sounds like it's huge!

- After "checkpoint" step, the transaction is not needed anymore because metadata and data made it safely to disk
  - So the space can be freed (free step).

- In practice: circular log

Metadata Journaling

- Recovery is much faster with journaling
  - Replay only a few transactions instead of checking the whole disk

- However, normal operations are slower
  - Every update must write to the journal first, then do the update
    - Writing time is at least doubled
  - Journal writing may break sequential writing. Why?
    - Jump back-and-forth between writes to journal and writes to main region
  - Metadata journaling is similar, except we only write FS metadata (no actual data) to the journal:

Journal Entry	TxBegin	I[v2]	B[v2]	TxEnd

Metadata Journaling

- What can happen now?
  - Say we write data after checkpointing metadata
  - If crash occurs before all data is written, inodes will point to garbage data!
  - How do we take care of this?

- Write data BEFORE writing metadata to journal!
  1. Write data, wait until it completes
  2. Metadata journal write
  3. Metadata journal commit
  4. 4. Checkpoint metadata
  5. Free

- If write data fails => as if nothing happened, sort of (from the FS's point of view)!

- If write metadata fails => same!

Summary: Journaling

- Journaling ensures file system consistency
- Complexity is in the size of the journal, not the size of the disk!
- Is fsck useless then?
- Metadata journaling is the most commonly used
  - Reduces the amount of traffic to the journal, and provides reasonable consistency guarantees at the same time.
- Widely adopted in most modern file systems (ext3, ext4, ReiserFS, JFS, XFS, NTFS, etc.)

Ext3 final notes

- Lacks modern FS features (e.g., extents)
  - For recoverability, this may actually be an advantage
  - FS metadata is in fixed, well-known locations, and data structures have redundancy
  - When faced with significant data corruption, ext2/3 may be recoverable when a tree-based file-system may not

Next up…

- Log-structured file systems

FFS

- Disk block index stored in inodes
- Metadata stored in inodes
- Directory entry stores file name and inode number
- Free blocks: bitmap
- Read performance?
  - Locate related blocks in same cylinder group
  - Locate inodes close to data blocks
- Write performance?
  - Block reallocation – reduces fragmentation, controls aging
  - Soft updates – alternative to journaling; ensures consistency without limiting performance
  - For rest of failure issues – background fsck

FFS Observation 2: Performance

- Performance of FFS: optimized for disk block clustering, using properties of the disk to inform file system layout

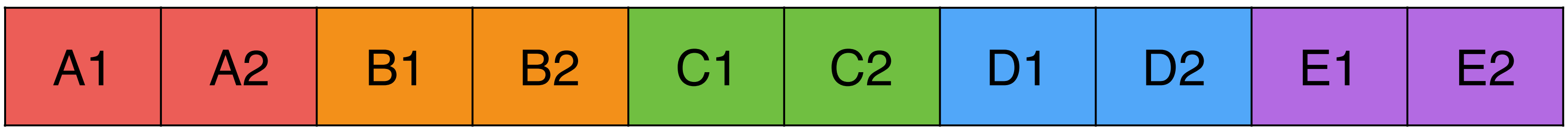
- Observation: Memory is now large enough
  - Most reads that go to the disk are the first read of a file. Subsequent reads are satisfied in memory by file buffer cache.

- => there is no performance problem with reads. But write calls could be made faster.

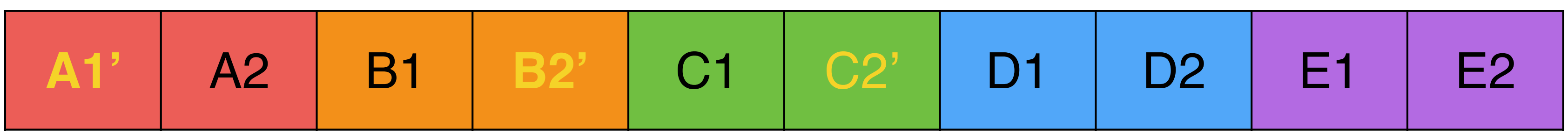
- Writes are not well-clustered. Why?

Log-Structured File System (LFS)

- Traditional FSs:
  - Files laid out with spatial locality in mind



  - Changes in place to mitigate seeks => e.g. A1’, B2’, C2’



  - Avoids fragmenting files, keeps locality => Reads perform well

Log-Structured File System (LFS)

- Another approach:
  - Memory is increasing => don't care about reads, most will hit in mem.
  - Assume writes will pose the bigger I/O penalty

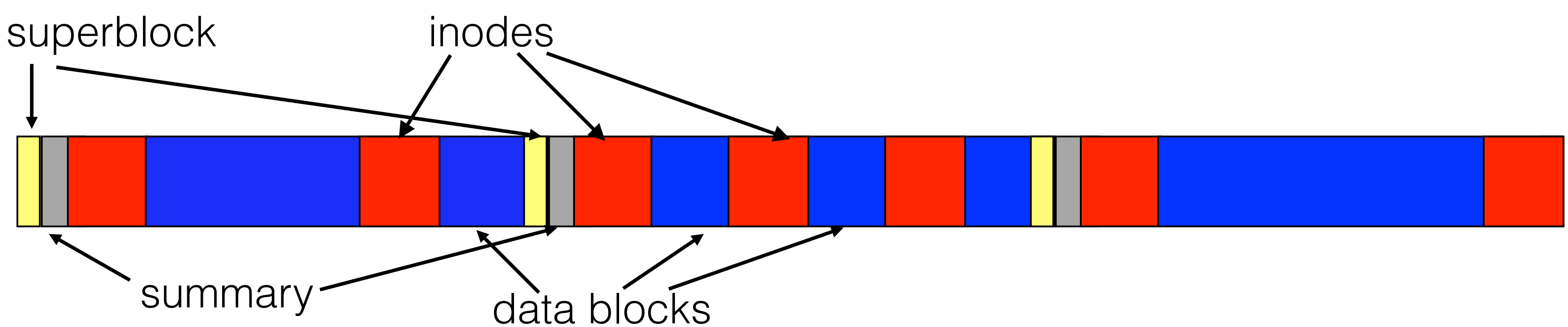
    => Treat storage as a circular log (Log Structured File System)

- Positive side-effects?
  - Write throughput improved (batched into large sequential chunks)
  - Crash recovery - simpler

- Disadvantages?
  - Initial assumption may not hold => reads much slower on HDDs. Why?

Log-Structured File System (LFS)

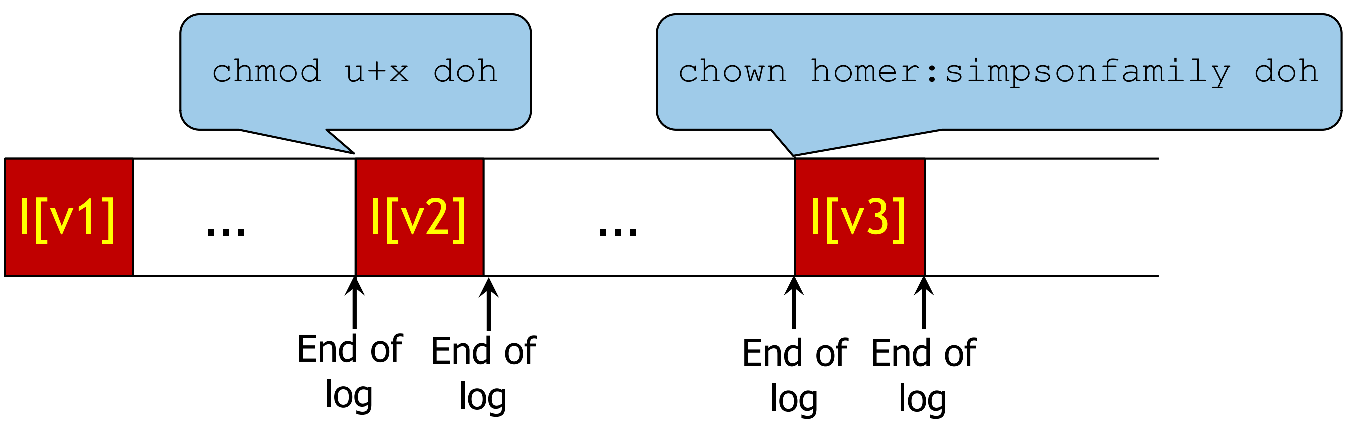
- Ousterhout 1989
- Write all file system data in a continuous log.
- Uses inodes and directories from FFS



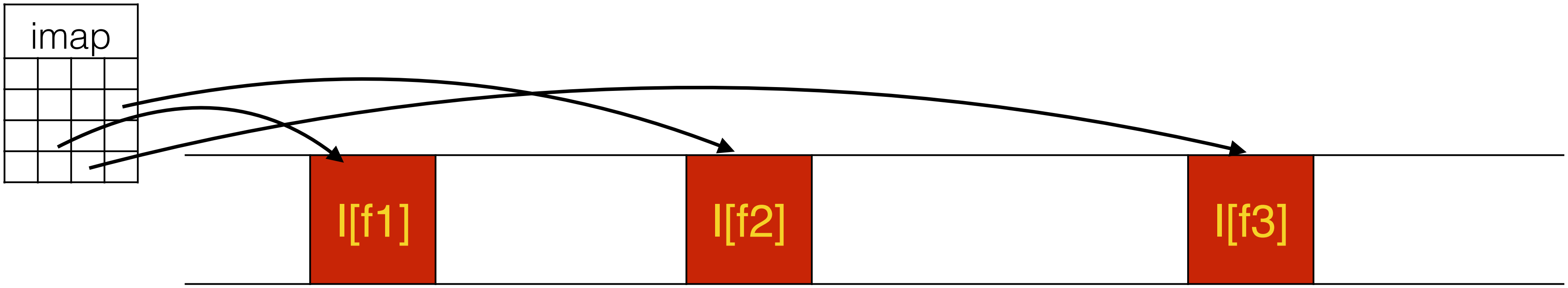
- Segments: each has a summary block
- Summary block contains pointer to next one
- Need a fresh segment => first clean an existing partially-used segment (garbage collection)

LFS: Locating inodes

- So, how do we find the inodes though?
  - In typical UNIX FSs, it's easy – array on disk at fixed locations
  - Superblock => Inode Table addr; Then add Inode# * InodeSize
- LFS: not so easy. Why?
  - Updates are sequential => inodes scattered all over the disk
  - Also, inodes not static, keep moving



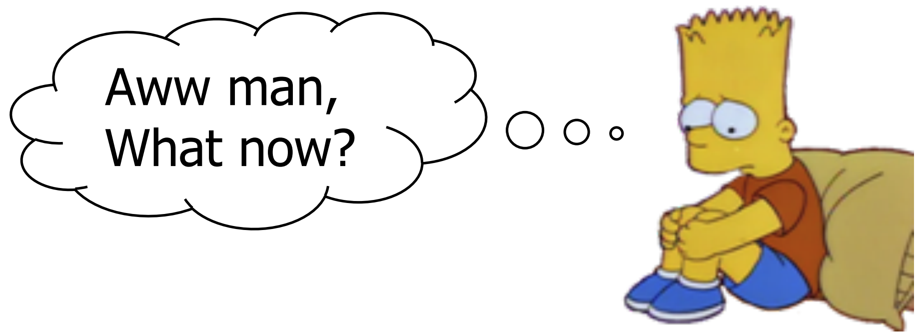
LFS: Locating inodes



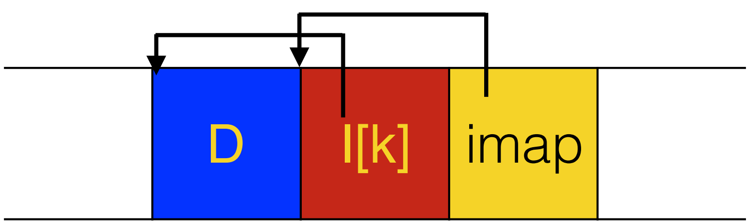
- LFS: Needs an inode map (imap) to find the inodes
  - An inode number is no longer a simple index from the start of an inode table as before
- Inode map must keep persistent, to know where inodes are
  - So it has to be on disk as well..
  - So, .. where exactly is the inode map stored on disk?

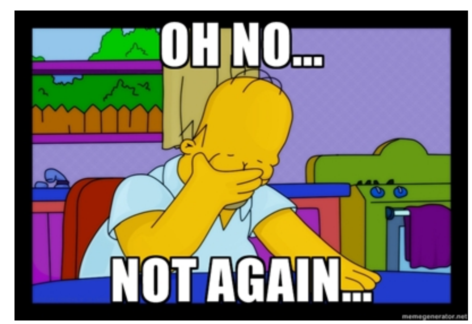
LFS: Locating inodes

- Put it on a fixed part of the disk
  - Inode map gets updated frequently though
    => Seeks↑ Performance↓



  - Instead, place chunks of inode map next to new information



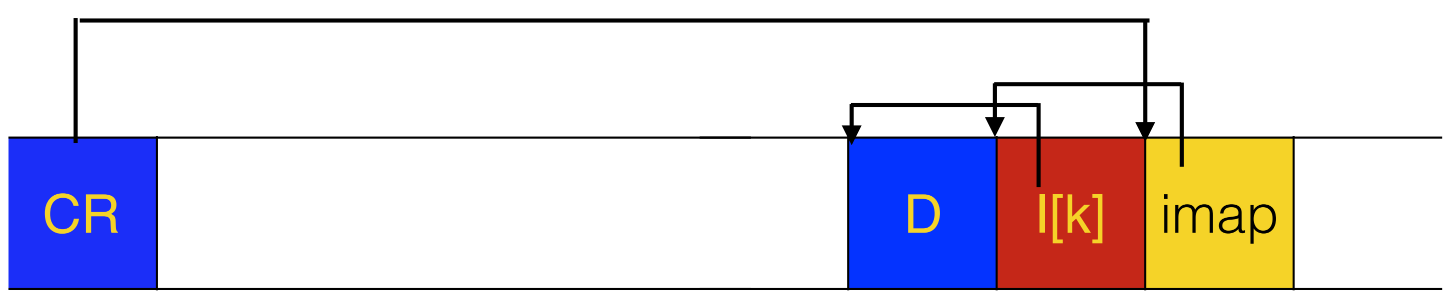


  - Hold on, but then how do we now find the imap?

LFS: Locating inodes

- Ok fine! The file system must have some fixed and known location on disk to begin a file lookup

- Checkpoint region (CR)
  - Pointers to the latest pieces of the inode map
  - So, find imap pieces by reading the CR!



Crash Recovery

- What if the system crashes while LFS is writing to disk?
- LFS normally buffers writes in segment
  - When full (or at periodic time intervals), writes segment to disk
- LFS writes segments and periodically also updates the CR
- Crashes can happen at any point
  - How do we handle crashes during these writes?

- Solution:
  1. Uncommitted segments – easy: reconstruct from log after reboot
  2. CR: Keep two CRs, at either end of the disk; alternate writes
     - Update protocol (header, body, last block)

Garbage Collection - Complicated!

- LFS repeatedly writes latest version of a file to new locations on disk
- Older versions of files (garbage) are scattered throughout the disk
- Must periodically find these obsolete versions of file data and clean up => free blocks for subsequent writes
- Cleaning done on a segment-by-segment basis
    - Since the segments are large chunks, it avoids the situation of having small “holes” of free space
- Garbage collection in LFS – interesting research problem!
    - Series of papers were published looking at its performance
    - Depending on when cleaning happens there may be significant performance loss.

LFS Summary

- A new approach that prioritizes update performance

- Gist: Instead of overwriting in place, append to log and reclaim (garbage collect) obsolete data later.

- Advantage:
  - Very efficient writes

- Disadvantages:
  - Less efficient reads (more indirection does not solve everything)
    - But assumes most reads hit in memory anyway.
  - Garbage collection is a tricky problem

- LFS inspired the logging features of journaling file systems in use today. E.g., Ext3/Ext4

Redundancy

- We saw how the journal helps recover from a crash during a write, by replaying the log entries of the write operations

- This assumes that the disk is still usable after rebooting

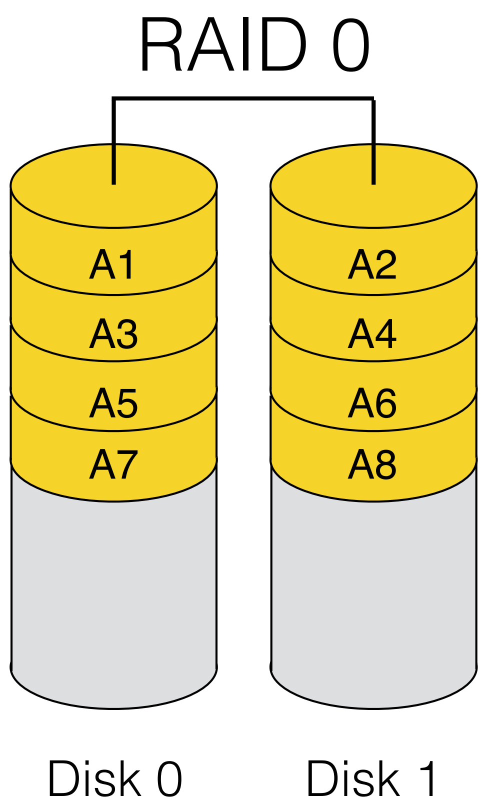
- What if we have disk failures?

- What can we do to prevent data loss?
    - Have more than one copies of the data: Redundancy!

RAID

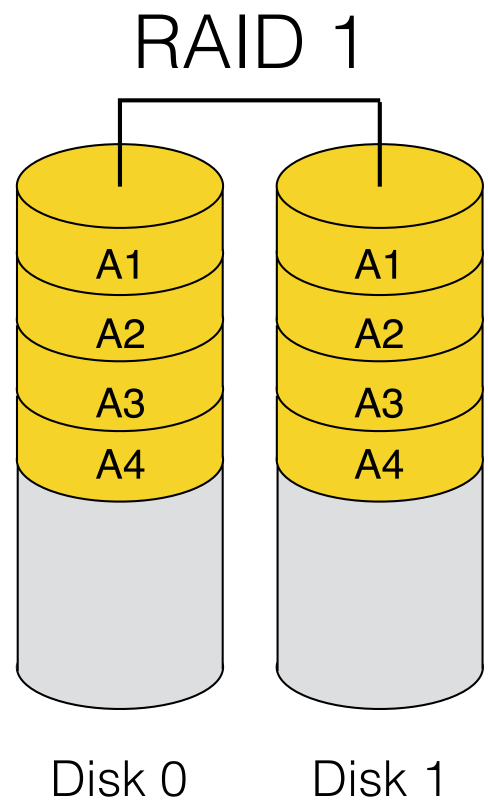
- Redundant Array of ~~Inexpensive~~ Independent Disks (RAID)
- Reliability strategies:
  - Data duplicated – mirror images, redundant full copy => one disk fails, we have the mirror
  - Data spread out across multiple disks with redundancy => Can recover from a disk failure, by reconstructing the data
- Concepts:
  - Redundancy/Mirroring: keep multiple copies of the same block on different drives, just in case a drive fails
  - Parity information: XOR each bit from 2 drives, store checksum on 3rd drive
- Multiple RAID levels – we'll only look at a few

Some Standard RAID levels



Striping

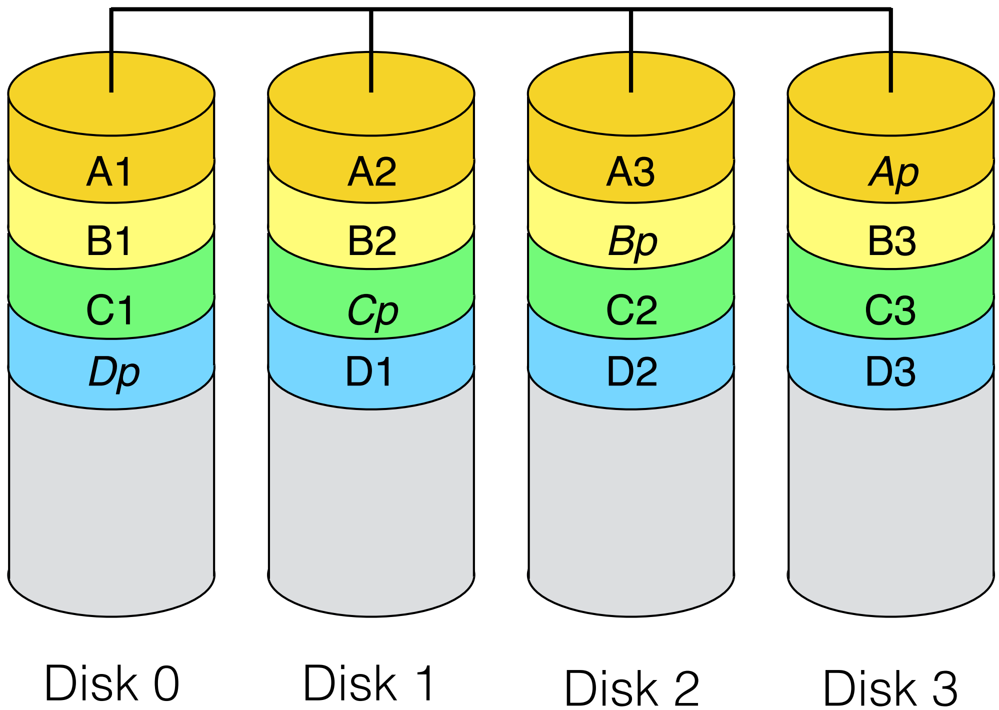
- Files are divided across disks
- Improves throughput
- If one drive fails, the whole volume is lost



Mirroring

- Capacity is half
- Any drive can serve a read
- Improved read throughput
- Write throughput is slower
- If one drive fails, no data lost

RAID 5



- Block level striping
- Distributed parity
- A failed disk can be reconstructed from the rest

Solid State Disks (SSD)

- Replace rotating mechanical disks with non-volatile memory
  - Battery-backed RAM
  - NAND flash

- Advantages: faster

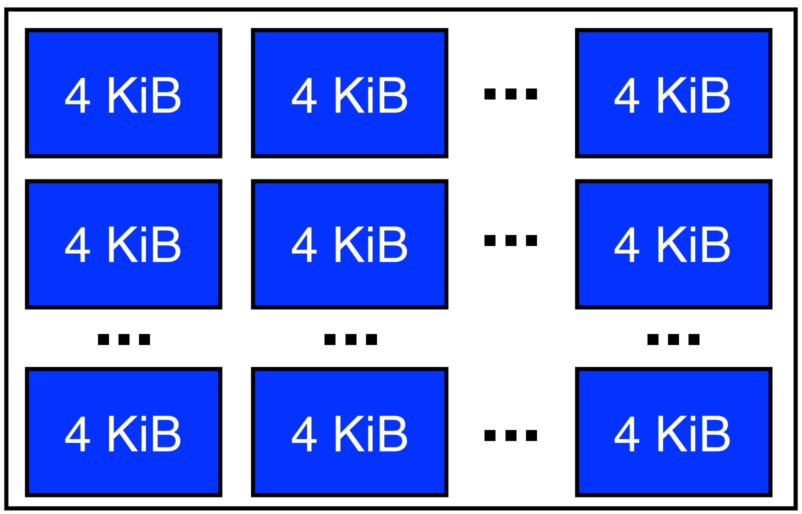
- Disadvantages:
  - Expensive
  - Wear-out (flash-based)

- NAND flash storage technology
  - Read / write / erase! operations

SSD Characteristics

- Data cannot be modified “in place”
  - No overwrite without erase

- Terminology:
  - Page (unit of read/write), block (unit of erase operation)



Data written in 4KB pages

Data erased in blocks of typically >= 128 pages

- Uniform random access performance!
  - Disks typically have multiple channels so data can be split (striped) across blocks, speeding access time

Writing

- Consider updating a file system block (e.g. a bitmap allocation block in ext2 file system)
    - Find the block containing the target page
    - Read all active pages in the block into controller memory
    - Update target page with new data in controller memory
    - Erase the block (high voltage to set all bits to 1)
    - Write entire block to drive
- Some FS blocks are frequently updated
    - And SSD blocks wear out (limited erase cycles)

SSD Algorithms

- Wear levelling
  - Always write to new location
  - Keep a map from logical FS block number to current SSD block and page location
  - Old versions of logically overwritten pages are “stale”

- Garbage collection
  - Reclaiming stale pages and creating empty erased blocks

- RAID 5 (with parity checking) striping across I/O channels to multiple NAND chips

File Systems and SSDs

- Typically, same FSs as for hard disk drives
  - ext4, Btrfs, XFS, JFS and F2FS support SSDs
- No need for the FS to take care of wear-leveling
  - Done internally by the SSD
  - But the TRIM operation is used to tell the SSD which blocks are no longer in use. (Otherwise a delete operation doesn't go to disk)
- Some flash file systems (F2FS, JFFS2) help reduce write amplification (esp. for small updates – e.g., FS metadata)
- Other typical HDD features – do we want these?
  - Defragmentation
  - Disk scheduling algorithms

SSD Reliability

- FAST2016 paper "Flash reliability in production" Google study on:
  - Millions of drive days over 6 years
  - 10 different drive models
  - 3 different flash types: MLC, eMLC and SLC
  - Enterprise and consumer drives
- Full paper: https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder
- Key points: http://www.zdnet.com/article/ssd-reliability-in-the-real-world-googles-experience/

Summary: File System Goals

- Efficiently translate file name into file number using a directory
- Sequential file access performance
- Efficient random access to any file block
- Efficient support for small files (overhead in terms of space and access time)
- Support large files
- Efficient metadata storage and lookup
- Crash recovery

Summary: File System Components

- Index structure to locate each block of a file
- Free space management
- Locality heuristics
- Crash recovery