

Week 11: Reliable, High Performance Storage

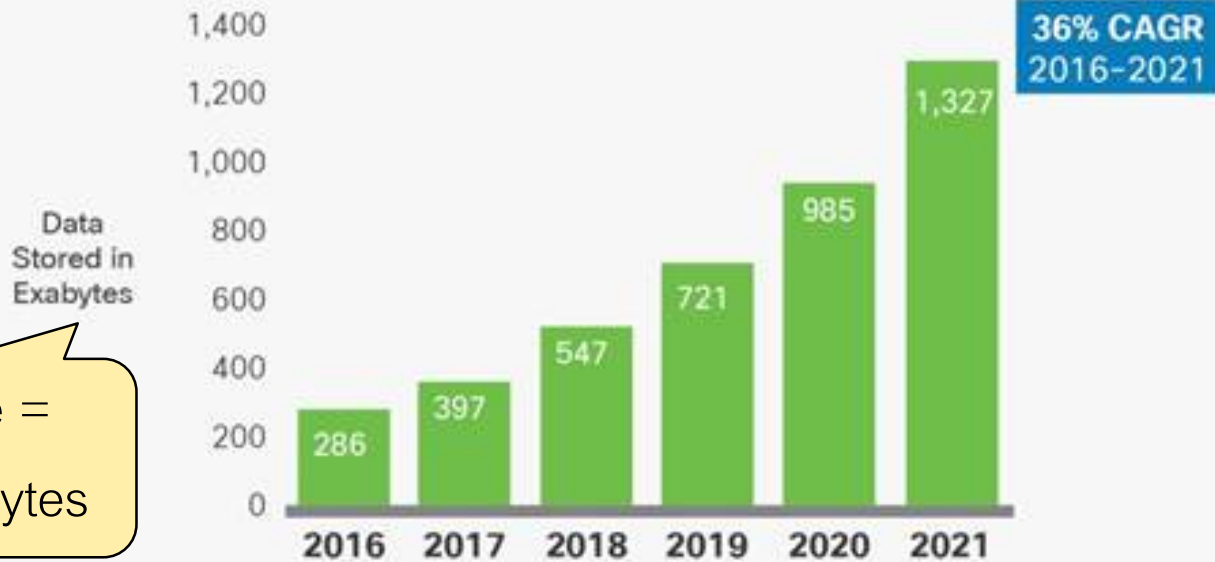
CSC469

Fall 2018



University of Toronto, Department of Computer Science

Global Data Centre Storage Utilization



1 Exabyte =
1024 Petabytes

Source: Cisco Global Cloud Index, 2016-2021;

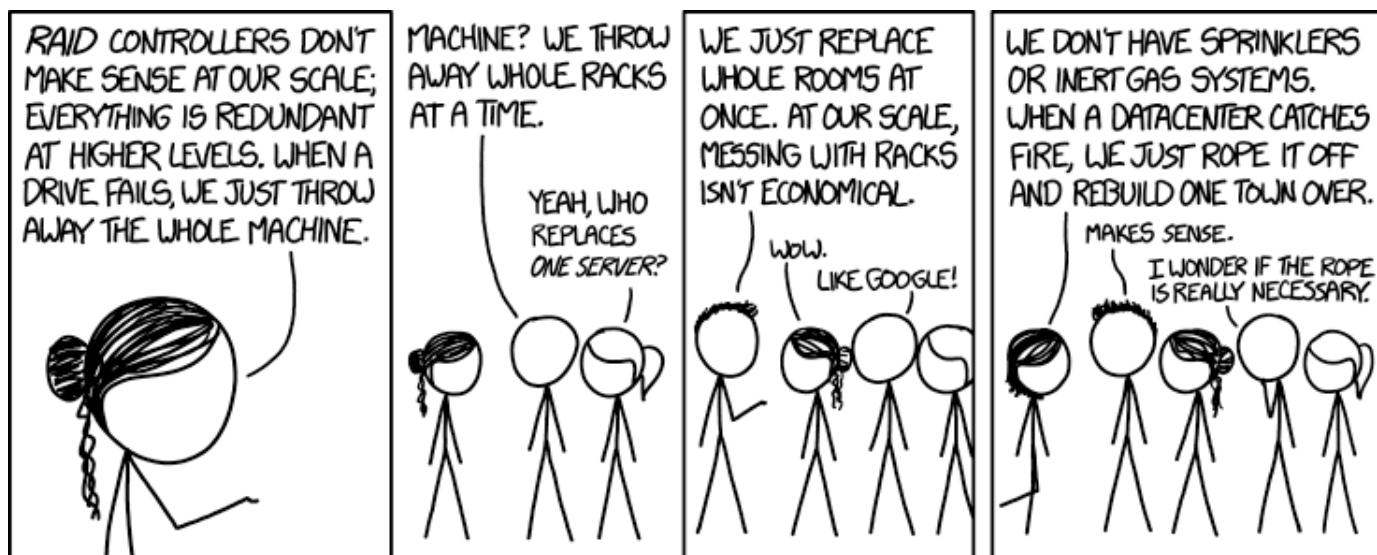
Cisco Global Cloud Index: Forecast and Methodology, 2016-2021 White Paper (updated November 19, 2018)





Data Storage at Scale

- Use distributed file systems to cope with data growth and processing demands
 - HDFS, Colossus, Ceph, Lustre, GlusterFS, ...
- Many of these are layered on top of local file systems
 - Ext4, XFS, btrfs, ZFS, ...
 - Reliability of local FS affects overall reliability



<https://xkcd.com/1737/>



Failure Propagation Examples

- Dropbox (and similar services)
 - Synchronization client on local machine can spread data corruption to the cloud storage
 - Can propagate further to copies on other devices
 - See “ViewBox: Integrating Local File Systems with Cloud Storage Services,” Zhang et al. FAST 2014.
- Lustre and Ifsck/e2fsck
 - High Performance Computing Center in Texas suffered power loss, consistency check and repair after restart, second power failure → severe data loss
 - See “Towards Robust File System Checkers,” Gatla et al. FAST 2018.



What about Performance?

- All durable storage is slower than DRAM
 - But new storage technology is closing the gap

| Technology | Read latency |
|------------------------------|---|
| DRAM | ~10 ns (64 byte line) |
| NVM (e.g., PCM) | ~20 ns (64 byte line) |
| SSD (NAND Flash) | ~25 us per block (512 or 4096 bytes) |
| HDD (magnetic rotating disk) | ~5 ms per block (512 or 4096 bytes) |

- Different characteristics need new FS designs
- Tension between speed and reliability



The Plan

- Study some example file and storage systems designed for high performance and reliability
 - Review: file systems for HDDs
 - BSD Unix Fast File System (FFS)
 - Reliability: Crash consistency mechanisms
 - Check-and-repair (e2fsck)
 - Journaling (ext3/4)
 - Soft Updates
 - Copy-on-write (btrfs)
 - Performance: File systems for SSDs (LFS, F2FS)



Basic Disk & File System properties

- Hard Disks provide large-scale non-volatile storage
- Access time determined by:
 - Seek time
 - Rotational latency
 - Transfer time
- Good performance depends on
 - Reducing seeks
 - Creating large transfers
- File systems provide logical organization and management of data on disk

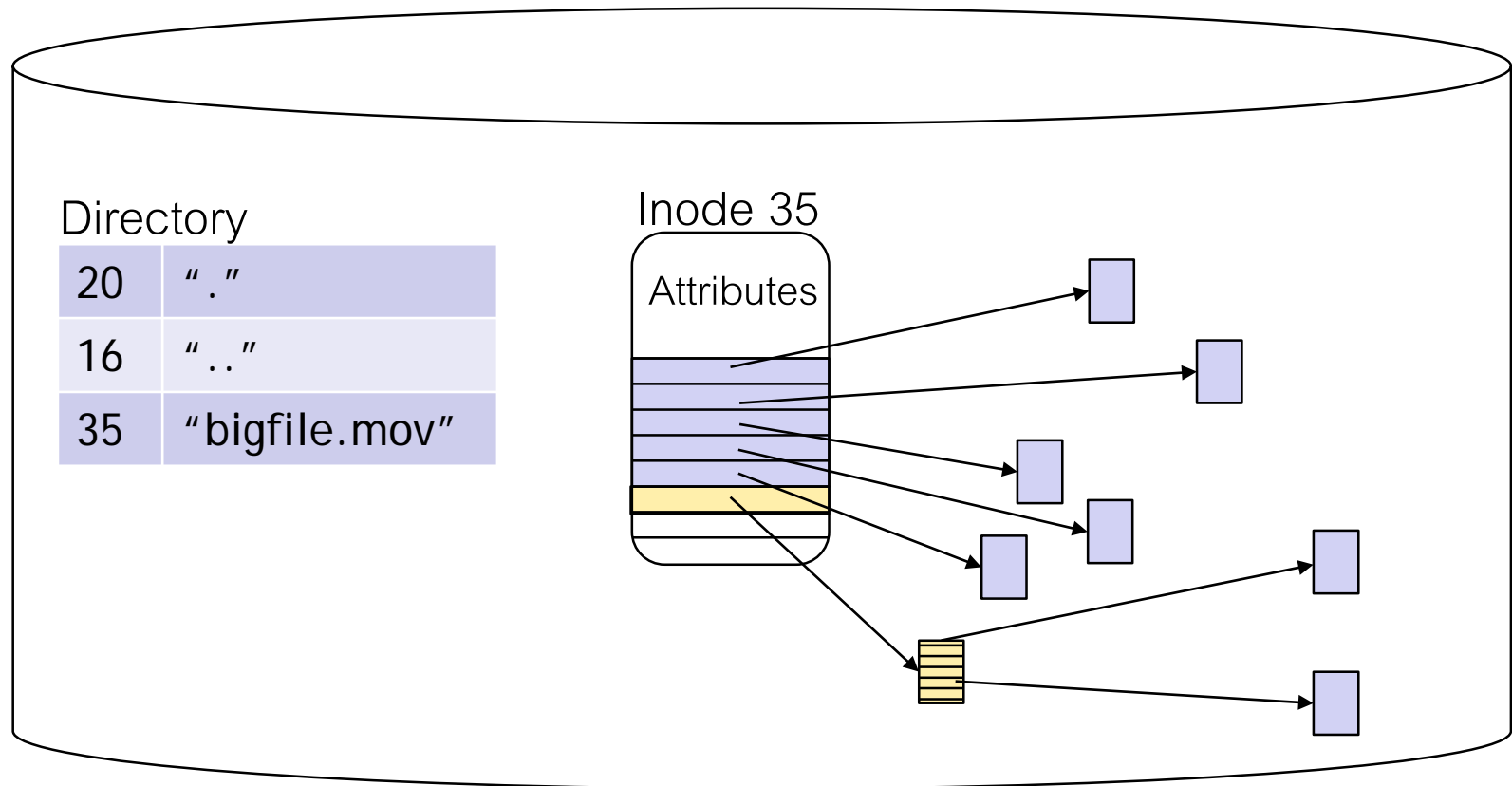




FFS: a disk-aware file system



Recall: UNIX inode

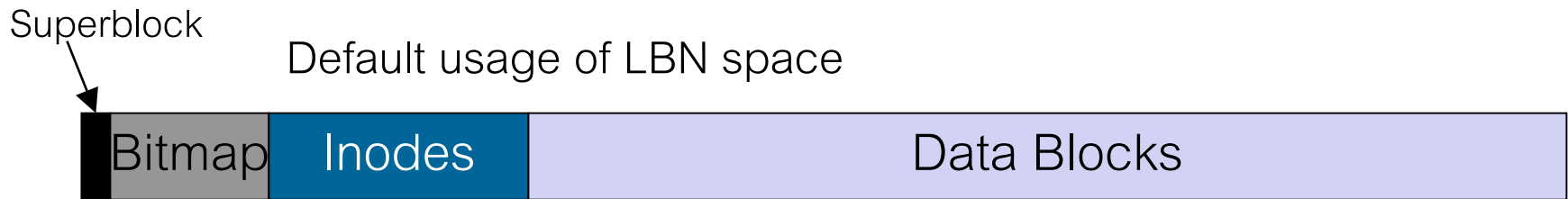


- Structure makes allocation easy – any free block
- But file data is likely to be scattered across disk



Original Unix File System

- File system uses *logical block numbers* (LBNs)
 - Device driver maps these to low-level sector addresses on disk



- Simple, straightforward implementation
 - Easy to implement and understand
- Poor bandwidth utilization (lots of seeking)
 - Inodes located at start of disk, can be far from data blocks
 - Traversing file name paths, manipulating files, directories requires **going back and forth from inodes to data blocks**
 - Does not handle *aging* well



The File System Aging Problem

- On a new FS, blocks are allocated sequentially, close to each other.



- As the FS gets older files are deleted and create random gaps.



- In aging file systems sequential data blocks can be allocated far apart.

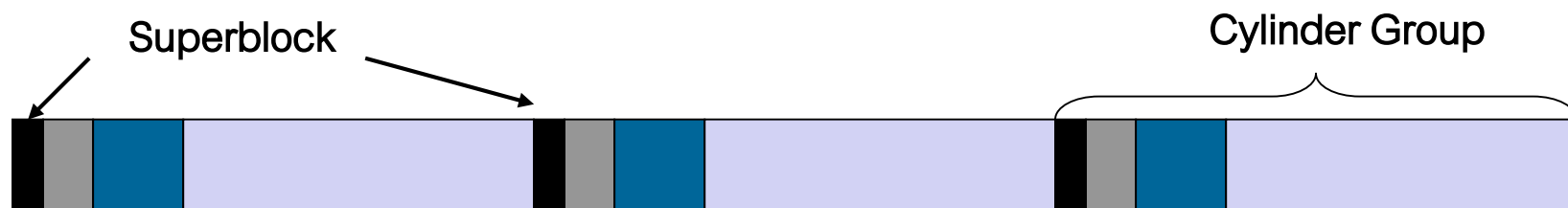


- Data blocks for new files end up scattered across the disk
- Fragmentation of an aging file system causes more seeking



Cylinder Groups

- BSD FFS addressed placement problems using **cylinder groups** (aka *allocation/block groups* in lots of modern FSs)
 - Data blocks in same file allocated in same cylinder group
 - Files in same directory allocated in same cylinder group
 - Inodes allocated in same cylinder group as file data blocks
 - Indirect blocks kept with inode and data blocks
 - Superblock *replicated* to improve reliability



Cylinder group organization

- **Good example of being device-aware for performance!**



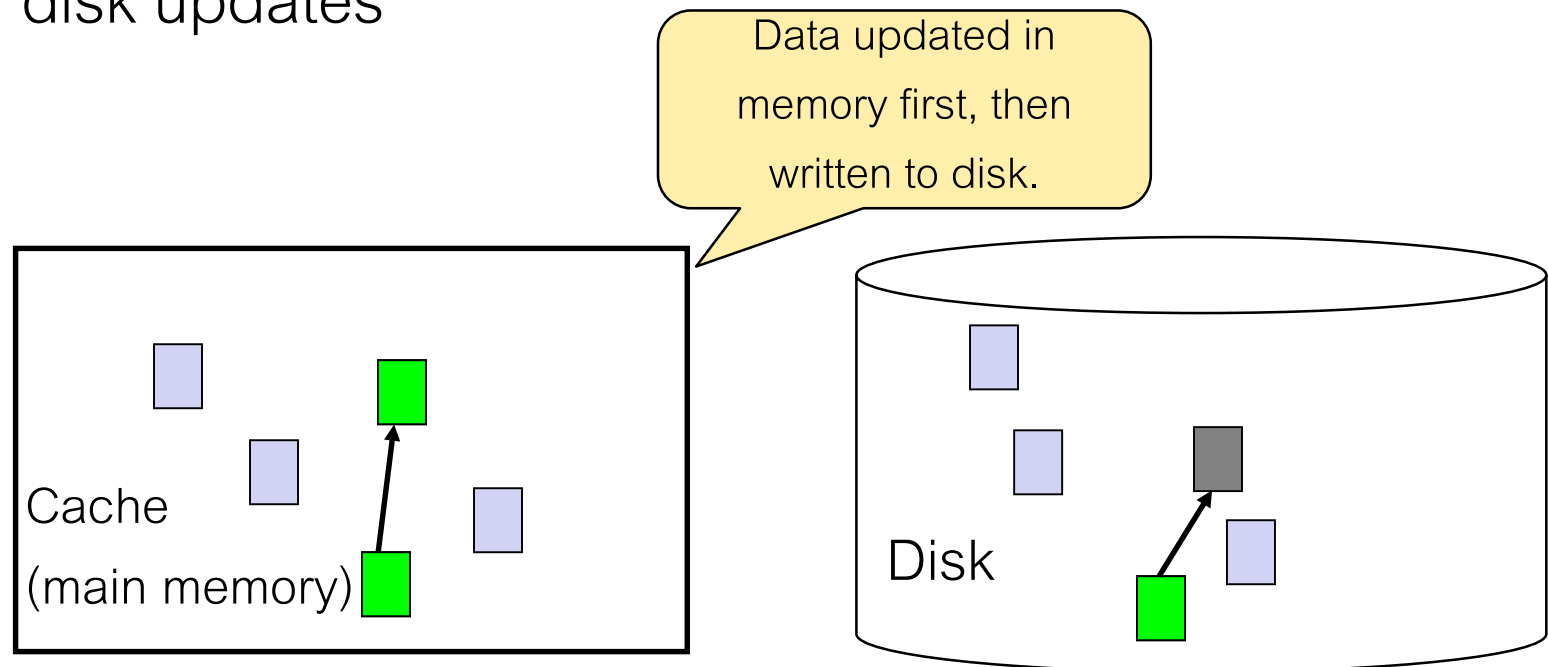
Reliability

- How do we guarantee consistency of on-disk storage?
 - Especially in the face of crash failures
 - This is called *crash consistency*: a guarantee that the file system will recover to some consistent state after a failure
- Generally concerned with consistency of file system metadata, not consistency of user data stored in files.



Metadata update problem

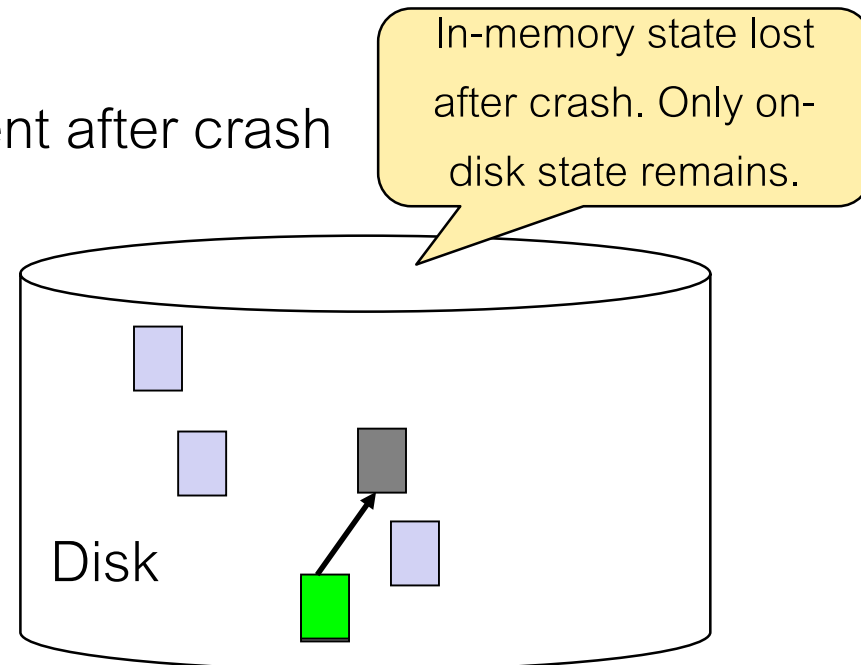
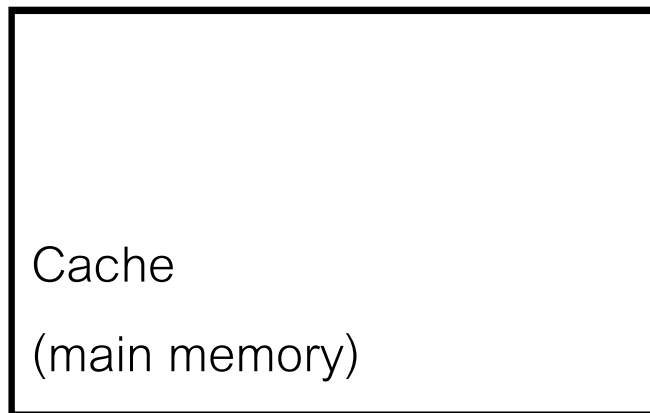
- File system metadata
 - inodes, directory blocks, allocation bitmaps
- Interdependencies must be handled carefully during disk updates





Metadata update problem

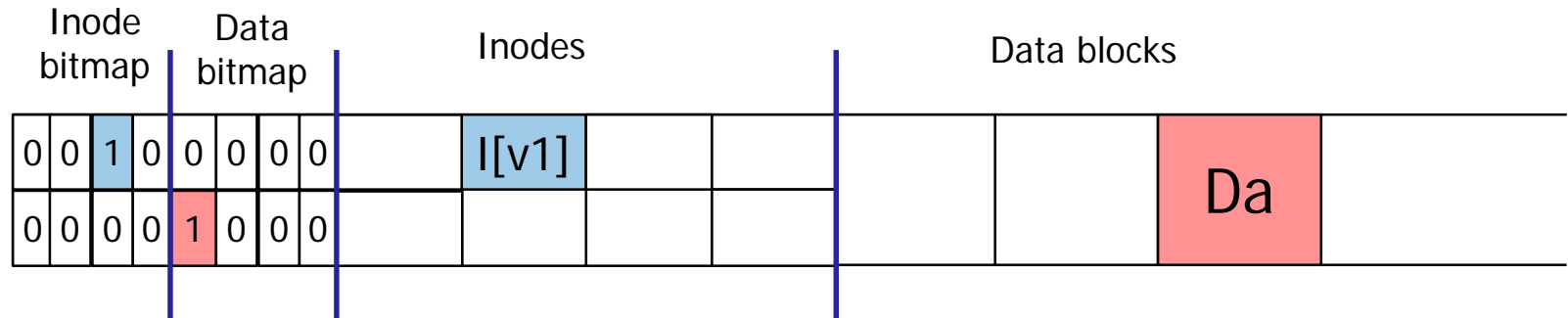
- File system metadata
 - inodes, directory blocks, allocation bitmaps
- Interdependencies must be handled carefully during disk updates
 - So that file system is consistent after crash



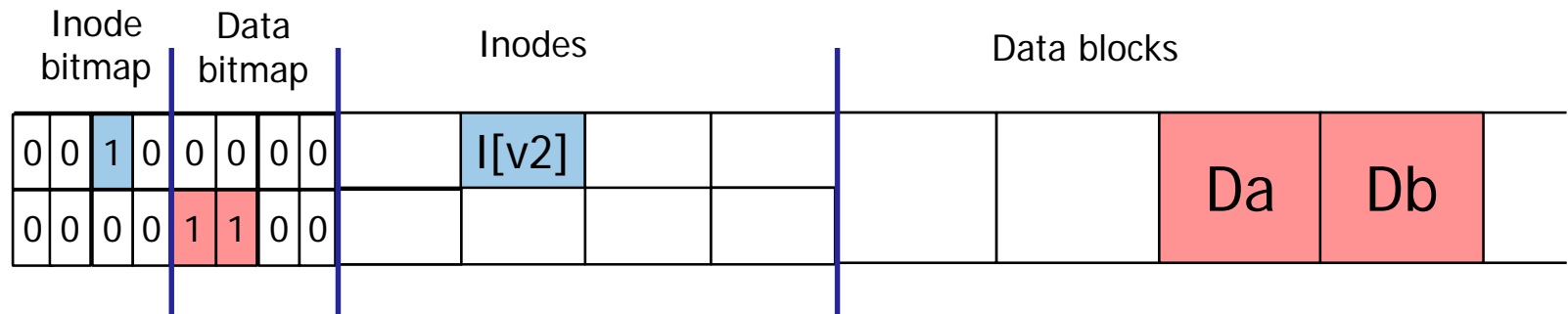


Example: update

- Consider a simple update: append 1 data block to a file
- Assume original Unix FS / FFS / ext2, etc structure, for simplicity:



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



- Three writes: Inode I[v2], Data bitmap B[v2], Data block Db



Crash consistency

- What if only one write succeeds before a crash?
 - 1. **Just Db write succeeds.**
 - No inode change, no bitmap change → as if the write did not occur
 - FS not inconsistent, but data is lost
 - 2. **Just I[v2] write succeeds.**
 - No data block → will read garbage data from disk.
 - No bitmap entry, but inode has a pointer to Db → FS inconsistency (dangling pointer)
 - 3. **Just B[v2] write succeeds.**
 - Bitmap says Db is allocated, inode has no pointer to it → FS inconsistent + Db can never be used again (space leaked)



Other crash scenarios

- What if two of the three writes succeed before a crash?
 - 1. Only $I[v2]$ and $B[v2]$ writes succeed.
 - Inode and bitmap agree \rightarrow FS metadata is consistent
 - However, Db contains garbage (or worse, old data)
 - 2. Only $I[v2]$ and Db writes succeed.
 - Inode points to correct data, but disagrees with $B[v1]$ \rightarrow FS inconsistency, must fix this or Db could be allocated again!
 - 3. Only $B[v2]$ and Db writes succeed.
 - Again, inode and bitmap info does not match
 - Db was written, but no inode points to it (which file is it part of? No idea!)

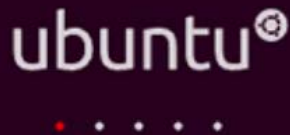


Crash consistency problem

Solution #1: Check-and-repair



You have seen this in action



ubuntu

Your disk drives are being checked for errors, this may take some time
Checking disk 1 of 1 (70 % complete)
Press C to cancel all checks currently in progress



Scanning and repairing drive (C:): 27% complete

www.windows-help-central.com

Similar tools exist for most file systems



Example check-and-repair: fsck

- e2fsck: Linux tool for finding inconsistencies in ext2/ext3/ext4 file systems and repairing them.
- Limitations:
 - Only FS integrity, cannot do anything about lost data
 - Correct repair is difficult
 - Allocated block with no inode pointing to it? Lost+found.
 - Block with multiple inodes pointing to it? Duplicate.
- Bigger problem: too slow!
 - Disks are very large – scanning all metadata could take hours (or days for multi-disk volume)
 - Even for small inconsistency, must scan whole file system



Crash consistency problem

Solution #1: Journaling



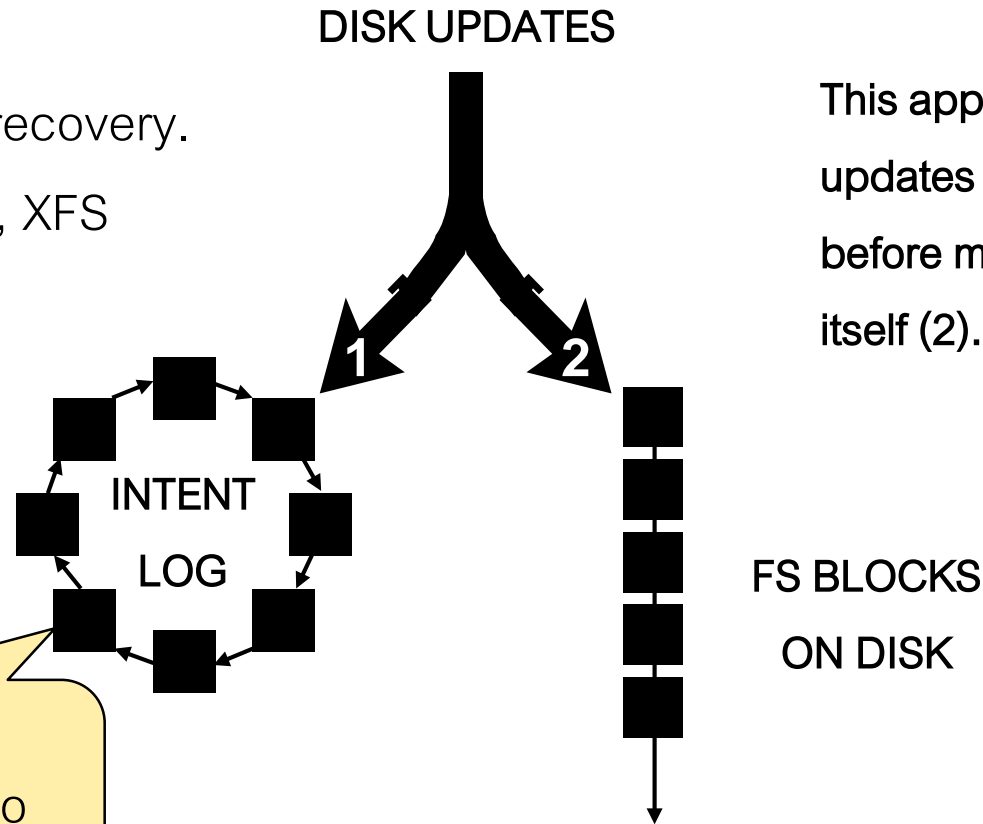
Journaling (aka Write-Ahead Logging)

- 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.



Write-ahead logging (briefly)

Benefit of logging is
strictly for reliability/recovery.
Used by ext3/4, JFS, XFS



This approach journals
updates to an intent log (1)
before modifying the file
itself (2).

Intent log, log, and
journal are all used to
refer to the same thing.



Design Alternatives

- Where should the journal be kept?
 - Ordinary file maintained by file system
 - Special set of blocks in FS (same as dedicating blocks for inodes)
 - Separate device
- What should be written to the journal?
 - Metadata only, or also data?
 - Logical (record changes) or physical (whole block)?
 - Undo (log old value) or redo (log new value)?
- At what granularity are updates flushed to disk?
 - Single update? File system operation? Group operations?
- How is journal space reclaimed? (Checkpointing)



What goes in that “note”

- 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) → Recovery must identify complete transactions



Data Journaling Example

- One solution: write each block synchronously
 - 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 reorders writes: 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**). Wait.

| | | | |
|---------|-------|-------|----|
| TxBegin | I[v2] | B[v2] | Db |
|---------|-------|-------|----|

- 2. Write TxEnd (only after 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 final 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 data and metadata 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
 - Doubles the volume of writes to storage
 - Journal writing may break sequential writing. Why?
 - Jump back-and-forth between writes to journal and writes to main region (depends on how updates are batched)
 - Metadata journaling is similar, except we only write FS metadata (no actual data) to the journal:

| | | | | |
|---------------|--------------------|------------------|-------------------|------------------|
| Journal entry | TxBegin (TID=1) | Updated inode | Updated Bitmap | TxEnd (TID=1) |
|---------------|--------------------|------------------|-------------------|------------------|



Metadata Journaling

- What new issues do we have to handle?
 - Suppose we write data after journaling the metadata
 - If crash occurs before all data is written, the inodes will point to garbage data!
- How do we prevent this?
 - Soln: Write data to final location *BEFORE* writing metadata to journal!
 1. Write data, wait until it completes
 2. Metadata journal write
 3. Metadata journal commit
 4. Checkpoint metadata
 5. Free journal entries
- If failure while writing data → as if none of the updates occurred
- If failure while journaling metadata → same outcome, as if no updates occurred



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 many modern file systems (ext3, ext4, JFS, XFS, NTFS, etc.)



Case Study: ext3

- Designed for backwards/forwards compatibility with ext2
 - Identical on-disk metadata structures
- Journal is just an ordinary (large) file
 - allocated when ext3 file system is created
- Separates journaling from file system concerns
 - Journaling is handled by JBD layer
 - ext3 file system uses JBD API to identify file system operations and modified blocks
 - JBD groups multiple fs operations into a single transaction
- Uses physical redo journaling
- Mount option to choose journaling mode
 - Default – metadata only



Writeback vs. Ordered Data Mode

- Mount option for how to handle non-journaled data blocks
 - **data=writeback** – data blocks are not journaled and are not ordered with respect to (journaled) metadata blocks
 - File can contain garbage (or worse!) after crash

"it makes things much smoother, since now the actual data is no longer in the critical path for any journal writes, but anybody who thinks that's a solution is just incompetent. ... If your data gets written out long after the metadata hit the disk, you are going to hit all kinds of bad issues if the machine ever goes down."

Linus Torvalds

- **data=ordered** – data blocks must be flushed to disk before metadata blocks are committed to journal
 - Still challenging since blocks can change type
 - E.g. rmdir, reuse dirent entry block for data, crash



One Problem: Stale metadata in journal

- T1: Create new file in dir foo -> adds entry to directory block
 - Direntry block gets added to current open transaction
 - T1 commits (direntry block written to journal on disk, then commit block)
- T2: “rm -rf foo”
 - Direntry block gets freed, bitmap block, parent direntry block added to transaction
 - T2 commits (blocks written to journal on disk)
- T3: File append -> freed direntry block allocated to file
 - Bitmap block, inode block added to transaction
 - T3 commits
 - Data block written out before metadata blocks
 - Bitmap and inode block written to journal on disk, then commit block
- Crash!

| | | | | | | | | | | |
|--------------------|-------------------|------------------|--------------------|--------------------|--------|------------------|--------------------|--------|-------|------------------|
| TxBegin (TID=1) | Direntry block | TxEnd (TID=1) | TxBegin (TID=2) | Parent Direntry | Bitmap | TxEnd (TID=2) | TxBegin (TID=3) | Bitmap | Inode | TxEnd (TID=3) |
|--------------------|-------------------|------------------|--------------------|--------------------|--------|------------------|--------------------|--------|-------|------------------|



Problem: Stale metadata replayed

- Recovery replays committed transactions

| | | | | | | | | | | |
|--------------------|-------------------|------------------|--------------------|--------------------|--------|------------------|--------------------|--------|-------|------------------|
| TxBegin (TID=1) | Direntry block | TxEnd (TID=1) | TxBegin (TID=2) | Parent Direntry | Bitmap | TxEnd (TID=2) | TxBegin (TID=3) | Bitmap | Inode | TxEnd (TID=3) |
|--------------------|-------------------|------------------|--------------------|--------------------|--------|------------------|--------------------|--------|-------|------------------|

- Replay of T1 overwrites data block appended to file (written before T3 committed) with old directory entry content
- Solutions?
 - Make certain delete operations synchronous
 - Don't reuse metadata blocks until checkpointed out of journal
 - Ext3: Delete in T2 adds "revoke record" for freed direntry block
 - Recovery scans for revokes first, does not replay revoked blocks



Solutions to Metadata Update

- Synchronous writes
 - Each update is completed at disk before next one is allowed
 - FFS → SLOW!
- Logged writes
 - Journaling file systems
 - Soft Updates



Soft Updates

- Delayed metadata writes
 - No waiting for disk
- Protect consistency of disks by controlling order of writes
 - use write-back caching for all (non-fsync) updates
 - make sure updates propagate to disk in the correct order
 - works great, but update ordering can't solve every problem



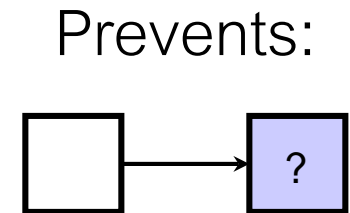
Basic Update Ordering Rules

- Purpose: integrity of metadata pointers
 - in face of unpredictable system failures
 - similar to rules of programming with pointers

- Rules:

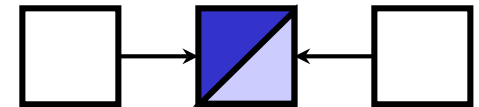
1. Resource Allocation

- initialize resource before setting pointer



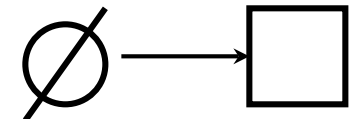
2. Resource De-allocation

- nullify previous pointer before reuse



3. Resource “Movement”

- set new pointer before nullifying old one

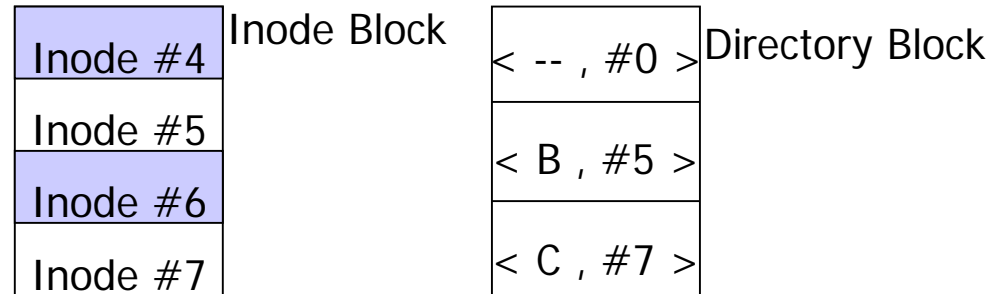


- Notice that something always left dangling
 - ... assuming a badly-timed crash

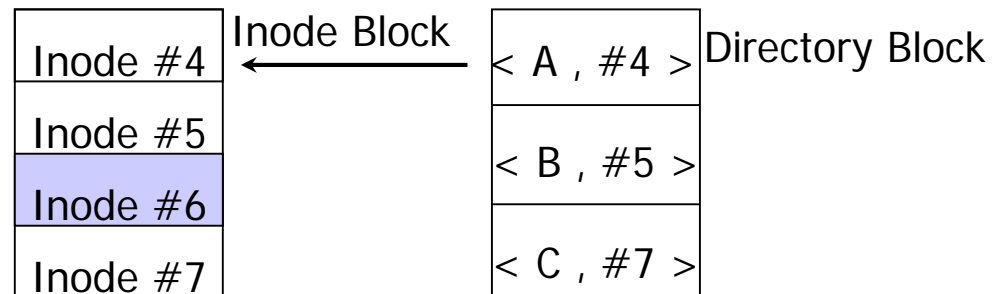


Example: Cyclic Dependencies

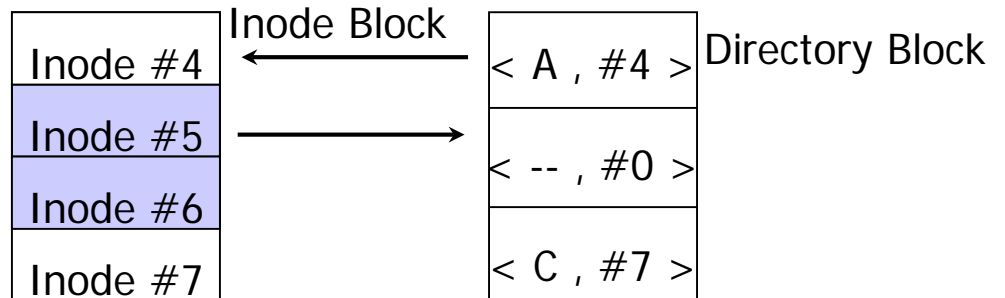
- Original Organization



- Create File A



- Remove File B



Free
Inode

In-use
Inode



Soft Updates Crash Recovery

- Need to deal with all of the dangling stuff
 - worse: need to find it all first
- Traditional recovery examines entire contents (recall fsck)
 - Post-crash time to mount grows with FS size and # of files (hours or days now)
- Journal recovery requires journal replay
 - Time proportional to amount of un-checkpointed metadata
 - Limited by random write throughput
- Soft Updates requires no post-crash recovery
 - Only inconsistencies are “leaked” blocks
 - Can find free blocks in background while using file system



Issues with Soft Updates

- Complexity!
 - 12 different data structures used for dependency tracking
 - McKusick & Ganger, 1999 USENIX FREENIX track paper
 - Each FS metadata structure uses subset of these with different relationships
 - Any change to file system structure requires rethinking it all
 - Anomalies with 'rm' of large directory tree required changes to file system inode (new 'effnlink' field)
- Space overhead
 - Dependence info + old and new versions of each change
- Computation & disk traffic overhead
 - May write single block multiple times to commit all updates



FFS Observation: Performance

- The performance of FFS is optimized for disk block clustering => fast reads (less seeking)
 - 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.
- perhaps there is no performance problem with reads.
- But, write calls could be made faster.
- Writes are not well-clustered (poor spatial locality). Why?



Update in place: good for reads

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



- Update-in-place (to mitigate seeks) → e.g., A1', B2', C2'



- Avoids fragmenting files, keeps locality → Reads perform well.
 - Writes .. not so much!



Log-Structured File System (LFS)

- Some took 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

- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
 1. **Disk bandwidth** scaling significantly (40% a year)
 - Latency did not improve much
 2. **Large main memories in machines**
 - Large buffer caches
 - Absorb large fraction of read requests
 - Can use for writes as well
 - Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
 - Rosenblum and Ousterhout (Berkeley, '91)



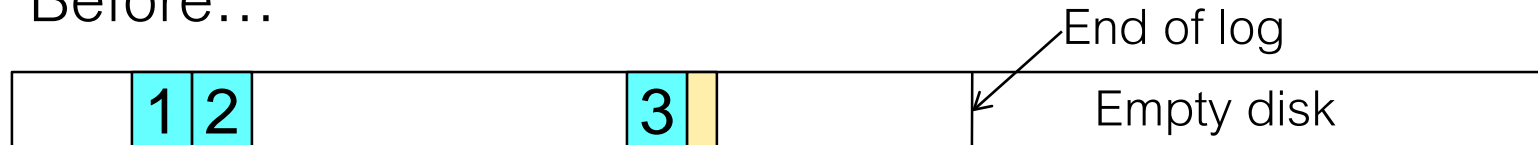
LFS Approach

- Treat the disk as a single log, keep appending data and metadata updates
 - Collect writes in disk cache, write out entire collection in one large disk request
 - Leverages disk bandwidth
 - No seeks (assuming read/write head is at end of log)
 - All info written to disk is appended to log
 - Data blocks, attributes, inodes, directories, etc.
- Simple, eh?
 - Alas, only in abstract

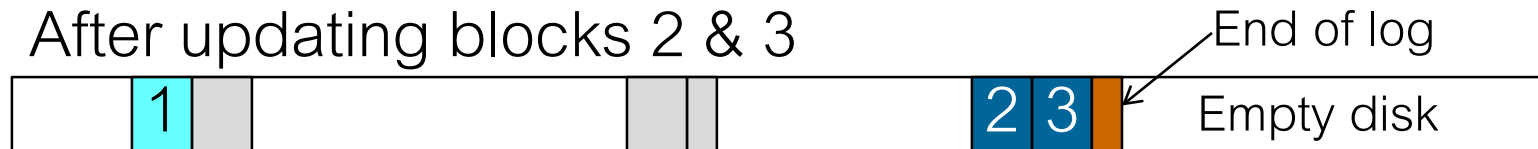







Example: Update file in LFS

Before...



After updating blocks 2 & 3



-  File inode before write
-  File data before write
-  File inode after write
-  File data after write
-  Obsolete blocks

- Modified blocks (data & metadata) are written to new location at end of log



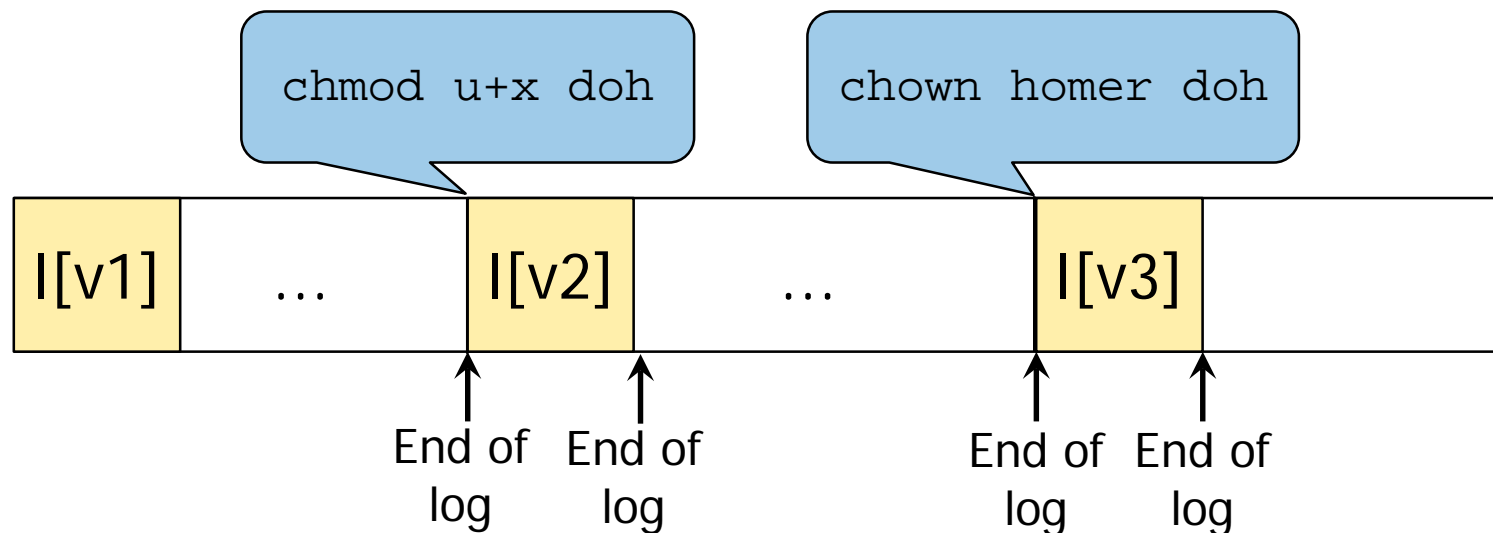
LFS Challenges

- LFS has **two challenges** it must address for it to be practical
 1. **Locating data written to the log**
 - FFS places files in a location, LFS writes data “at the end”
 2. **Managing free space on the disk**
 - Disk is finite, so log is finite, cannot always append
 - Need free space and which must be contiguous
 - What happens to the stale blocks in old parts of log?



LFS problem 1: Locating inodes

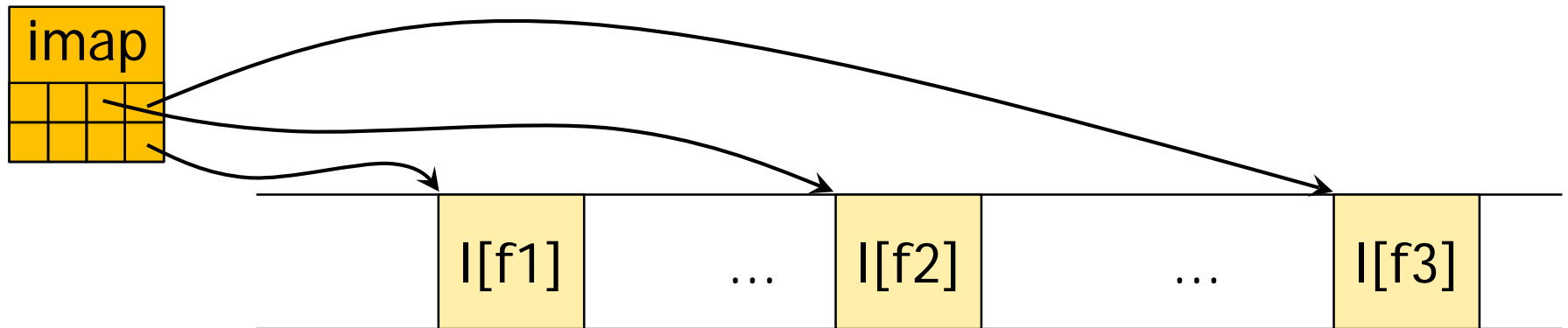
- So, how do we find the inodes?
 - In typical UNIX FSs, it's easy – array on disk at fixed locations
 - Superblock \rightarrow Inode Table addr; Then add $\text{Inode\#} * \text{InodeSize}$
- LFS: not so easy. Why?
 - Updates are sequential \rightarrow inodes scattered all over the disk
 - Also, inodes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

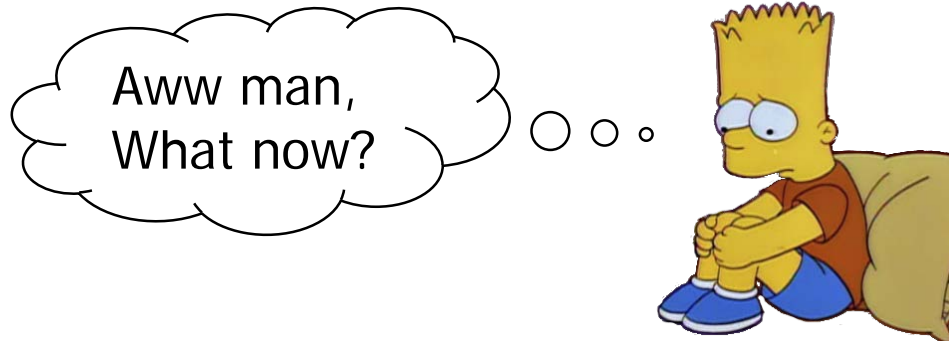


- Must keep inode map persistent, to know where inodes are
 - So it has to be on disk as well..
 - So, .. **where exactly is the inode map stored on disk?**

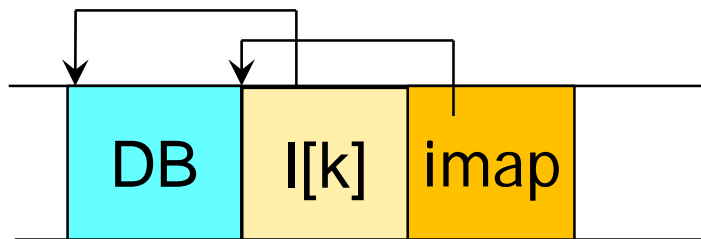


LFS: Locating inodes

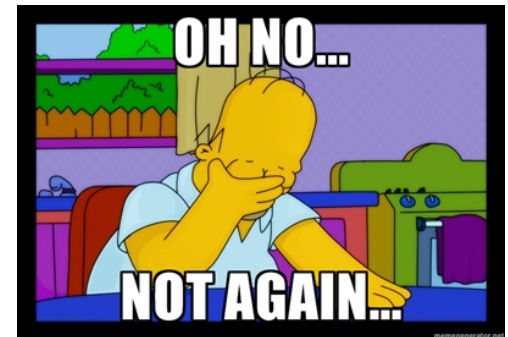
- Put it on a fixed part of the disk
 - Inode map gets updated frequently though \rightarrow Seeks \uparrow Performance \downarrow



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



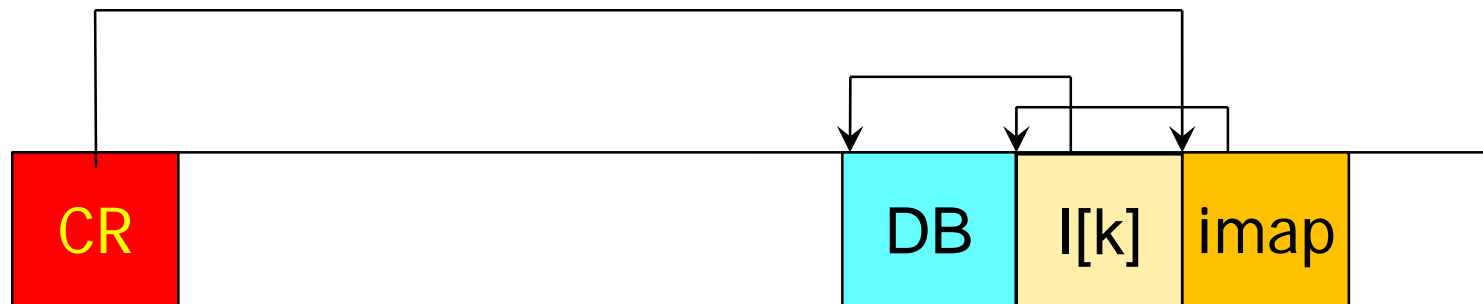
- Hold on, but then how do we now find the imap?





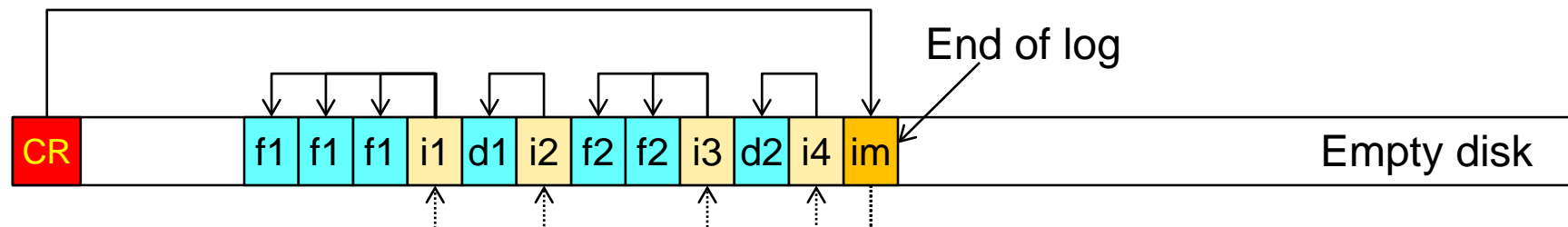
LFS: Locating inodes

- 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
 - Find imap pieces by reading the CR!





LFS Layout

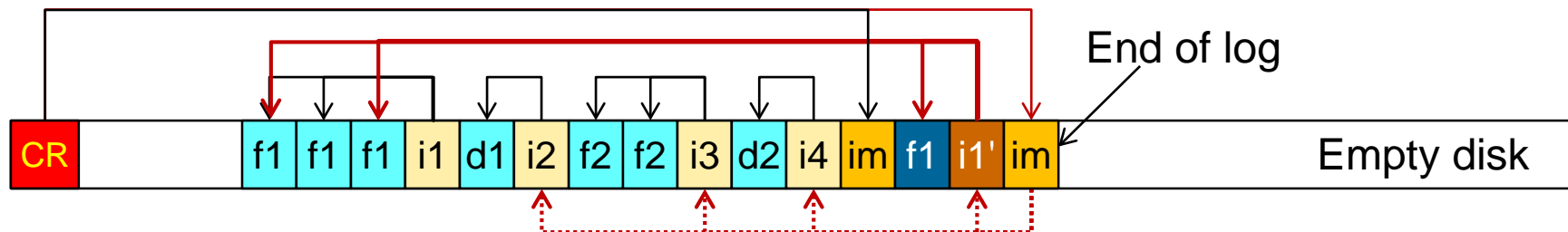


- ix** File inode before write
- fx** File data before write
- dx** Directory data before write
- im** Inode map block
- CR** Checkpoint region (fixed location)

- Suppose we just created directory d1 with file f1, and directory d2 with file f2.



LFS Layout – updating a file



ix File inode before write

ix File inode after write

fx File data before write

fx File data after write

dx Directory data before write

im Inode map block

CR Checkpoint region (fixed location)

- Update file f1
- New inode map generated with each write to disk



LFS Problem 2: Free Space Management

- LFS append-only quickly runs out of disk space
 - Need to recover deleted blocks
- Approach:
 - Fragment log into segments, write segment-at-once



- Chain active segments on disk
 - Segments can be anywhere



- Reclaim space by **cleaning** segments
 - Read segment
 - Copy live data to end of log
 - Now have free segment you can reuse
- High cleaning overhead is a big problem



LFS: Crash Recovery

- Traditional Unix FS: changes can be made anywhere
 - Must scan entire disk to restore metadata consistency after crash (fsck)
- LFS: most recent changes are at end of log
 - Much faster/easier to achieve a consistent state
- LFS borrows 2 database techniques for recovery after a crash:
 - *Checkpoints* define a consistent file system state
 - *Roll-forward* recovers information written since the last checkpoint



LFS Checkpoints

- Recall CR: Special, fixed location on disk containing:
 - Addresses of all inode map blocks
 - Segment usage table (for cleaning)
 - Address of last segment written
 - Timestamp
- Crash can occur while writing the checkpoint
 - Keep two checkpoint regions
 - Write header (with timestamp), body of CR, then one last block (also with a timestamp)
 - On recovery, use region with most recent (and consistent!) timestamp



LFS Roll-Forward

- Can recover data written since last checkpoint
- Examine segments written since segment identified by checkpoint region
 - Segments contain *special summary blocks* and *directory operation logs* that record changes to file system metadata
 - Operation log entry is guaranteed to appear *before* modified directory or inode blocks
 - If directory or inode is inconsistent, operation log entry can be used to correct it



LFS Today

- Implementations exist for Linux, BSD, others
 - NILFS/NILFS2, F2FS
 - NILFS supports snapshots in a fairly natural manner
- Used in specialized applications
 - Checkpointing file system for parallel applications
- Ideal match for SSD storage
 - Why?
 - Writes are still expensive due to block erase
 - Write amplification problem
 - **Wear-leveling** to balance wear-out of cells



Next Time...

- Overview of how NAND Flash SSDs operate
- F2FS – a file system for NAND Flash SSD
- Announcements
 - Tutorial this week – more help with A3!
 - Also, use Piazza
 - Use the A3 testing system integrated on MarkUs once your code works locally! (Should be operational today)
 - Course evals:
 - <http://www.uoft.me/course-evals>