Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

Journaling *entire sequence*

- Journal write
  - write all transaction entries except TxE, *wait until on-disk*
- Journal commit
  - write TxE, *wait until on-disk*
- *Checkpoint:*
  - write all pending metadata and data updates to final locations in actual bitmaps, inodes, and data blocks

Journaling *recovery*

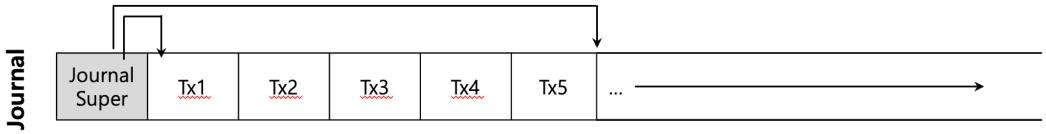
- If crash **before** transaction is written to log
  - pending update **dropped**

- During recovery
  - scan disks for all committed transactions
  - replay in order

- Issues:
  - Works, but recovery is slow…. (like `fsck`)
  - log eventually fills up, FS stops

Journaling *recovery*

- Create a journal *superblock*
  - mark first and last *uncheckpointed* Xtions



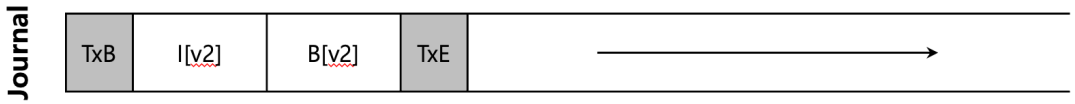
- So complete protocol is:
  - journal write
  - journal commit
  - checkpoint
  - *free* the journal space

  - periodically push free Xtions to journal superblock

Journaling *metadata*

- Still a problem : we are writing every block to disk twice
  - commit to journal
  - checkpoint to on-disk location
  - we've halved our disk bandwidth!
    - (data blocks are majority of journal)

- Metadata journaling
  - data blocks *not* written to journal
  - journal would look like:



- *Instead*: block Db written to final location

Journaling *metadata*

- *When* should we write the data blocks to disk?
  - if write data to disk after transaction:
    - file system consistent, but I[v2] might point to garbage
  - Write data to final locations *first*

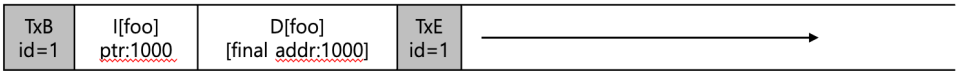
"write the pointed-to object before the pointer"

- Protocol now:
  - *data write*
  - journal metadata write
  - *wait for completion of first two steps*
  - journal commit
  - checkpoint metadata
  - free at some later time

Journaling *tricksy: block reuse*

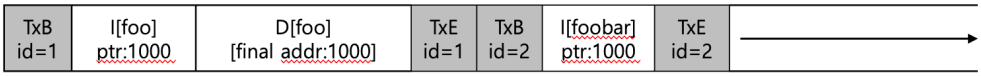
- Some metadata really should not be replayed:

1. Directory “foo” is updated by adding a file



2. Directory “foo” is entirely deleted, block 1000 freed

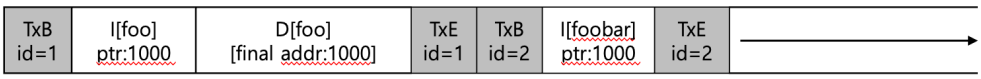
3. User creates file “foobar” using block 1000 for data…



“What’s the hideous part of the entire system? ... It’s deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated.” [Tweedie00]

Journaling *tricksy: block reuse*

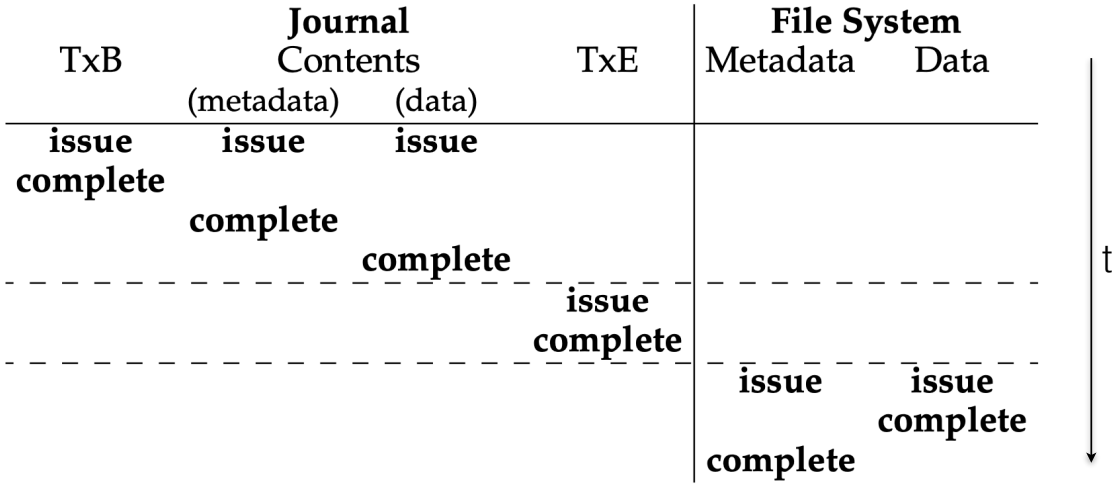
- Assume crash occurs and all this is in the log:



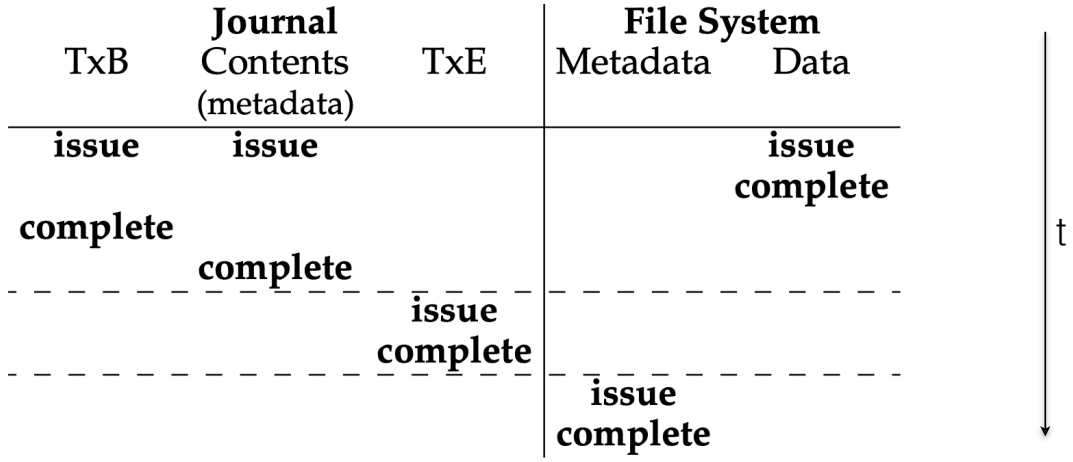
- During replay, recover process replays everything in the log
  - including the write of directory data to block 1000
  - thereby overwriting the user data from file foobar
  - but the foobar data is not journaled, so it is now lost…

- ext3 uses *revoke records* when a directory is deleted
  - Recovery first scans for revoke records
  - Revoked data not written during recovery

Data Journaling Timeline *wrapping up…*



Metadata Journaling Timeline *wrapping up…*



Other Approaches

- Soft updates
  - "*pointed-to data must always be written before pointer*"
  - for all FS data
  - difficult, depends on low-level details, hard to get right
- Copy-on-write (COW)
  - never overwrite in place
  - always allocate new blocks for data, inodes, etc.
  - change pointer to entire tree of data w/ one swap.
- Backpointer consistency
  - add "backpointer" from data to pointer that references it
    - data block has a backpointer to inode
  - when referencing the data through the inode, check that the data block has a correct backpointer
  - win is that *no ordering need be enforced between writes*

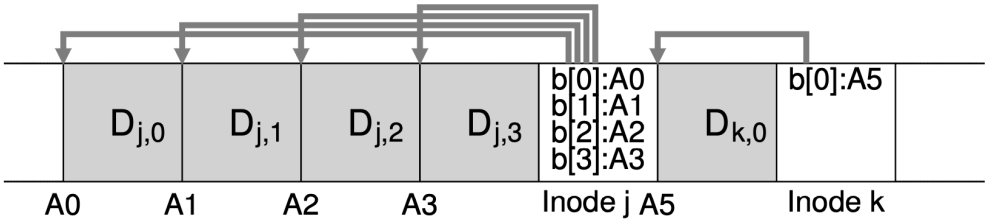
Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured (and other) File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

Log-Structure File System (LFS)

- Motivation

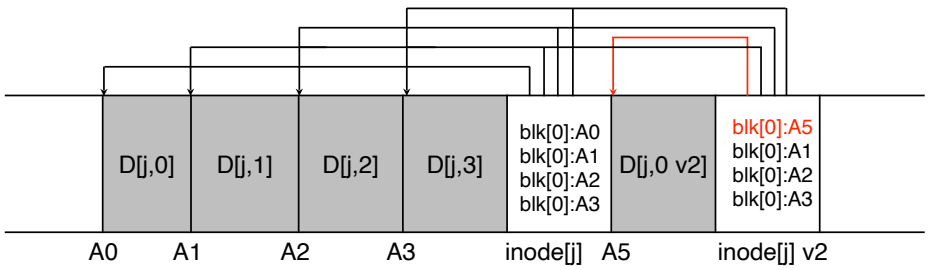
  Reads already fast because large buffer caches
  - but writes are slow, need to be ordered, and *synchronous*
  - common operations, such as creating a small file, require many random writes
- Idea:
  - many synchronous small writes $\Longrightarrow$ single large log write
  - writes ordered s.t. any pointed-to data is in log *prior to* ptr
  - periodically flush large chunks of log to disk



files *j*, *k* ***in-memory log***

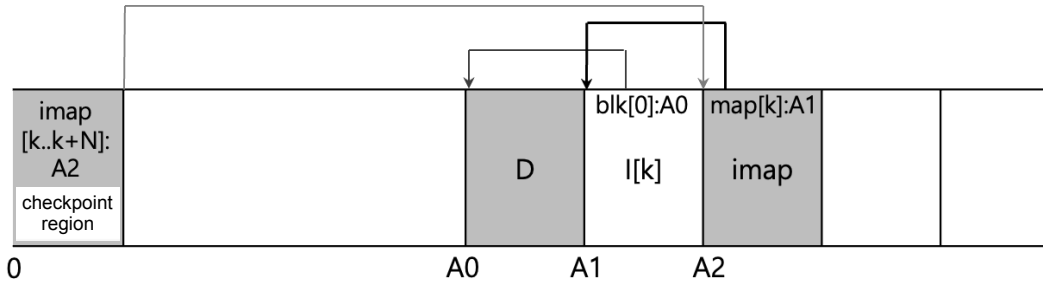
Modifying a file

- Let's say we overwrite the first block of file *j*
  - write the new block at end of log
  - write the altered inode at end of log
  - update the *imap* and append new copy at end of log



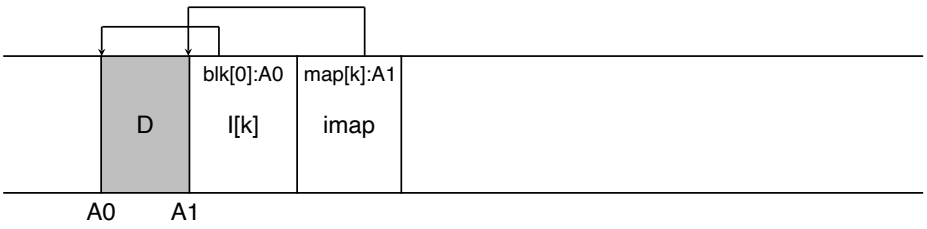
LFS Issues *inode location*

- Most recent version of inodes scattered throughout the disk!
  - have an *inode map* (or *imap*) that maps inumbers to most recent version of an inode
    - inode map cached in memory
    - periodically written to disk (e.g. every 30 seconds)
    - new chunks are written into log along w/ everything else:



Reading a file *recap*

- read the checkpoint region to find *imap:*
  - *imap* inumbers to most recent version of an inode
  - inode points to data



LFS

- Periodically write log to disk
  - dependencies between writes are respected by order in log
  - therefore any *prefix of the log is self-consistent*
- At recovery from a crash:
  - the on-disk log will have no holes, i.e. it's a prefix and will be self-consistent
  - any incomplete transactions (file create, etc.) are marked as garbage
  - most recent inode map is read and disk is *ready to be used*
- In particular:
  - no re-executions
  - no rollbacks (other than marking a few Xtions as garbage)

LFS *how large should written chunks be?*

- Each write incurs a fixed positioning overhead T_{pos}, so the time to write out *D* MB is:

$$T_{\text{write}} = T_{\text{position}} + \frac{D}{R_{\text{peak}}}$$

(R_{peak} is peak rate)

- Effective rate is therefore:

$$R_{\text{eff}} = \frac{D}{T_{\text{pos}} + \frac{D}{R_{\text{peak}}}} = F \times R_{\text{peak}}$$

(*F* is percent of R_{peak} of peak rate)

- Solving for D:
  - $D = \frac{F}{1-F} \times R_{\text{peak}} \times T_{\text{pos}}$
- With F=0.9, peak transfer of 1 GB, positioning time of 10 msec:
  - $F = 9 \times 1000\text{MB/sec} \times 0.01\text{sec} = 90\text{MB}$