Persistence

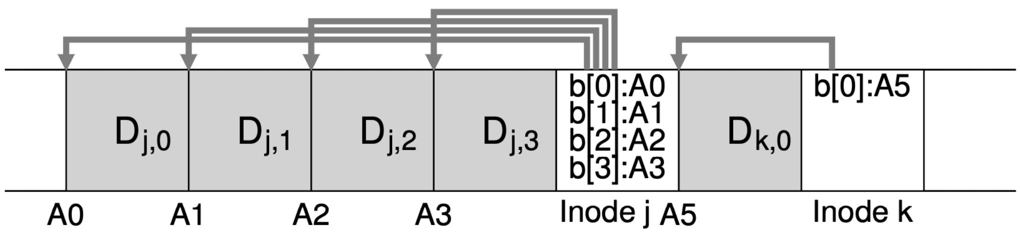
- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured (and other) File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

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
  - never overwrite in place
  - always allocate new blocks for data, inodes, etc.
  - change pointer to a 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 is enforced between writes*

Log-Structure File System (LFS)

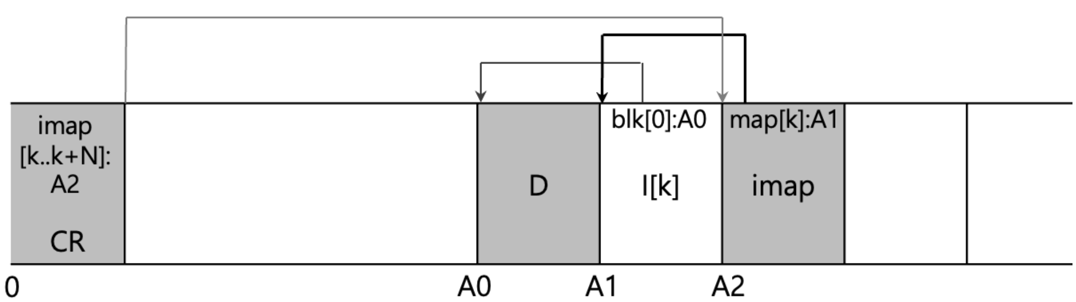
- Motivation
  - reads sped up by large buffer caches
  - writes are slow, need to be ordered, and *synchronous*
  - common operations, such as creating a small file, require many random writes
- Idea:
  - many synchronous small writes $\Longrightarrow$ single large log write
  - writes ordered s.t. any data pointed to defined in log prior to ptr
  - periodically flush log to disk



files *j*, *k*

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
  - written to disk as periodic checkpoint (e.g. every 30 seconds)
  - new chunks are written into log along w/ everything else:



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:
  - $F = 9 \times 1000\text{MB/sec} \times 0.01\text{sec} = 90\text{MB}$

LFS Issues *need for a cleaner*

- no overwrite means
  - files, dirs, etc. become fragmented
  - parts of the log no long active
    - all but most-recent versions of inodes
    - data that has been modified
    - imap chunks
- *cleaner process* asynchronously copies live data
  - from *full segments* to clean new segments
  - cleaned segments are empty, can be used again
  - might use this opportunity to segregate by age, activity, etc.
    - segment full of rarely-changing data rarely needs cleaning

LFS *cleaning costs*

- Cleaner
  - read some number of live segments
  - copy live data out into fewer new segments
  - old segments are now free.
- But....write amplification! Let:
  - N : num segments to be cleaned
  - u : percent of these segments that is live
  - write cost (wc): *write amplification* of each new byte

```
write cost  = (#readSegs +#writeLive + #writeNew) / (#writeNew)
            = (N + N*u + (N*(1-u)) / (N * (1-u))
            = 2/(1-u)
```

  - if utilization low, say 10%: *wc = 2.22*
  - if utilization high, say 90%: *wc = 20.00*

LFS *cleaner notes*

- advantages
  - asynchronous
  - can be done in bulk
- opportunities
  - *older data less likely to be modified than new data*
  - can segregate data based on age for cleaner writes
- implemented on bare disk
  - log chunk to be written to disk is many pages long
  - LFS can report consistency check of all blocks back to OS
  - LFS guarantees that pg *i* written correctly before pg *i+1*