A principled approach: Transactions

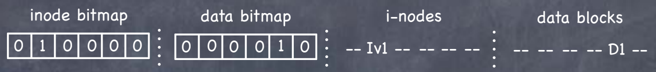
- Group together actions so that they are
  - Atomic: either all happen or none
  - Consistent: maintain invariants
  - Isolated: serializable (schedule in which transactions occur is equivalent to transactions executing sequentially
  - Durable: once completed, effects are persistent
- Critical sections are ACI, but not Durable
- Transaction can have two outcomes:
  - Commit: transaction becomes durable
  - Abort: transaction never happened
    - may require appropriate rollback

Solution 3: Journaling (write ahead logging)

- Turns multiple disk updates into a single disk write
  - "write ahead" a short note to a "log", specifying changes about to be made to the FS data structures
  - if a crash occurs while updating FS data structures, consult log to determine what to do
    - no need to scan entire disk!

Data Jounaling: an example

- We start with



- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | B2 | D2 | TxEnd
    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What if we crash before the log is updated?
  - if no commit, nothing made it into FS - ignore changes!
- What if we crash after the log is updated?
  - replay changes in log back to disk!

Journaling and Write Order

- Issuing the 5 writes to the log TxBegin | Iv2 | B2 | D2 | TxEnd sequentially is slow
  - Issue at once, and transform in a single sequential write!?
- Problem: disk can schedule writes out of order
  - first write TxBegin, Iv2, B2, TxEnd

  Disk loses power →
  - then write D2
- Log contains: TxBegin | Iv2 | B2 | ?? | TxEnd
  - syntactically, transaction log looks fine, even with nonsense in place of D2!
- Set a Barrier before TxEnd
  - TxEnd must block until data on disk

Back to

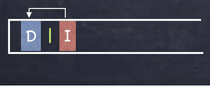
Where is this from?

The early 90s

- Growing memory sizes
    - file systems can afford large block caches
    - most reads can be satisfied from block cache
    - performance dominated by write performance
- Growing gap in random vs sequential I/O performance
    - transfer bandwidth increases 50%-100% per year
    - seek and rotational delay decrease by 5%-10% per year
    - using disks sequentially is a big win
- Existing file system perform poorly on many workloads
    - 6 writes to create a new file of 1 block
        - new inode | inode bitmap | directory data block that includes file | directory inode (if necessary) | new data block storing content of new file | data bitmap
    - lots of short seeks

Log structured file systems

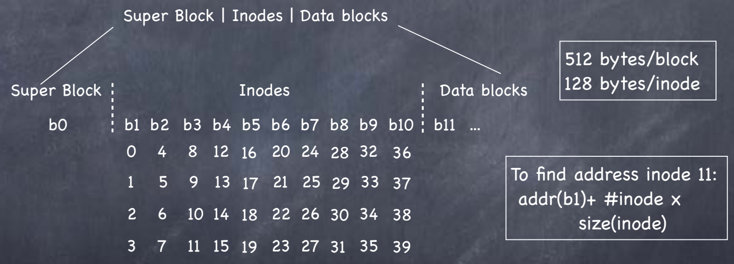
- Use disk as a log
    - buffer all updates (including metadata!) into an in-memory segment
    - when segment is full, write to disk in a long sequential transfer to unused part of disk
- Virtually no seeks
    - much improved disk throughput
- But how does it work?
    - suppose we want to add a new block to a 0-sized file
    - LFS paces both data block and inode in its in-memory segment



Fine.
But how do we find the inode?

Finding inodes

- in UFS, just index into inode array



Super Block | Inodes | Data blocks

Super Block	Inodes										Data blocks	
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	...
	0	4	8	12	16	20	24	28	32	36		
	1	5	9	13	17	21	25	29	33	37		
	2	6	10	14	18	22	26	30	34	38		
	3	7	11	15	19	23	27	31	35	39		

512 bytes/block
128 bytes/inode

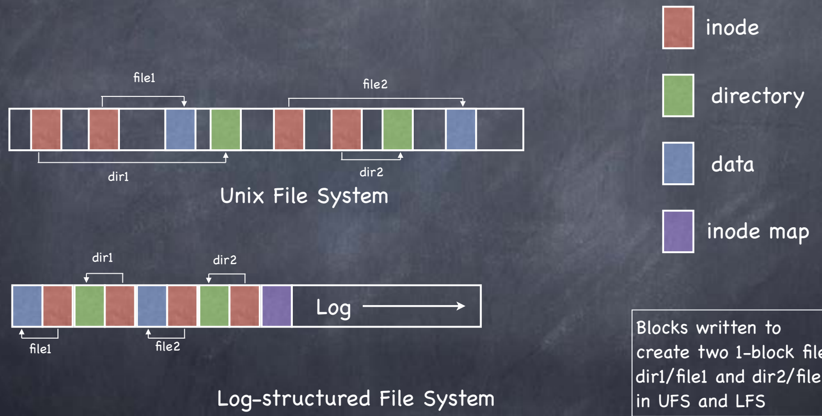
To find address inode 11:
addr(b1)+ #inode x size(inode)

- Same in FFS (but Inodes are at divided (at known locations) between block groups

Finding inodes in LFS

- Inode map: a table indicating where each inode is on disk
  - Inode map blocks are written as part of the segment
  - ... so need not seek to write to imap
- but how do we find the blocks of the Inode map?
  - Normally, Inode map cached in memory
  - On disk, found in a fixed checkpoint region
    - updated periodically (every 30 seconds)
- The disk then looks like

LFS vs UFS



Blocks written to create two 1-block files: dir1/file1 and dir2/file2 in UFS and LFS

Reading from disk in LFS

- Suppose nothing in memory...
  - read checkpoint region
  - from it, read and cache entire inode map
  - from now on, everything as usual
    - read inode
    - use inode's pointers to get to data blocks
- When the imap is cached, LFS reads involve virtually the same work as reads in traditional file systems

modulo an imap lookup

Garbage collection

- As old blocks of files are replaced by new, segment in log become fragmented
- Cleaning used to produce contiguous space on which to write
  - compact M fragmented segments into N new segments, newly written to the log
  - free old M segments
- Cleaning mechanism:
  - How can LFS tell which segment blocks are live and which dead?
    - Segment Summary Block
- Cleaning policy
  - How often should the cleaner run?
  - How should the cleaner pick segments?

Segment Summary Block

- Kept at the beginning of each segment
- For each data block in segment, SSB holds
  - The file the data block belongs to (inode#)
  - The offset (block#) of the data block within the file
- During cleaning, to determine whether data block D is live:
  - use inode# to find in imap where inode is currently on disk
  - read inode (if not already in memory)
  - check whether a pointer for block block# refers to D's address
- Update file's inode with correct pointer if D is live and compacted to new segment

Which segments to clean, and when?

- When?
  - when disk is full
  - periodically
  - when you have nothing better to do
- Which segments?
  - utilization: how much it is gained by cleaning
    - segment usage table tracks how much live data in segment
  - age: how likely is the segment to change soon
    - better to wait on cleaning a hot block, since free blocks are going to quickly reaccumulate

Crash recovery

- The journal is the file system!
- On recovery
  - read checkpoint region
    - may be out of date (written periodically)
    - may be corrupted
      - 1) two CR blocks at opposite ends of disk / 2) timestamp blocks before and after CR
      - use CR with latest consistent timestamp blocks
  - roll forward
    - start from where checkpoint says log ends
    - read through next segments to find valid updates not recorded in checkpoint
      - when a new inode is found, update imap
      - when a data block is found that belongs to no inode, ignore