Caching and Consistency

- File systems maintain many data structures
  - Bitmap of free blocks and inodes
  - Directories
  - Inodes
  - Data blocks
- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?

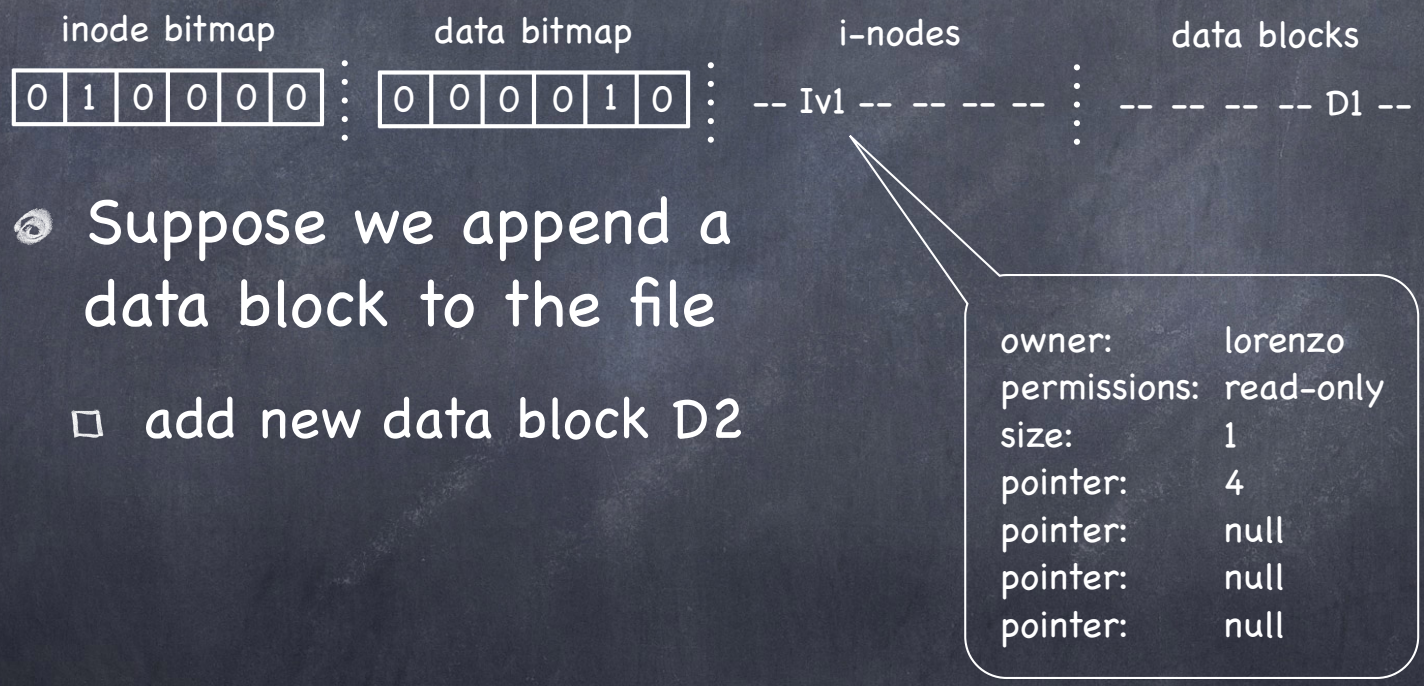
Caching and consistency

- File systems maintain many data structures
  - Bitmap of free blocks and inodes
  - Directories
  - Inodes
  - Data blocks
- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?
- Write-back caches
  - delay writes: higher performance at the cost of potential inconsistencies
- Write-through caches
  - write synchronously but poor performance (fsync)
    - do we get consistency at least?

Crash Consistency

Example: a tiny ext2

- 6 blocks, 6 inodes



- Suppose we append a data block to the file
  - add new data block D2

Example: a tiny ext2

- 6 blocks, 6 inodes

- Suppose we append a data block to the file
  - add new data block D2
  - update inode

owner:	lorenzo
permissions:	read-only
size:	1
pointer:	4
pointer:	null
pointer:	null
pointer:	null

Example: a tiny ext2

- 6 blocks, 6 inodes

- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

owner:	lorenzo
permissions:	read-only
size:	2
pointer:	4
pointer:	5
pointer:	null
pointer:	null

Example: a tiny ext2

- 6 blocks, 6 inodes

- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

owner:	lorenzo
permissions:	read-only
size:	2
pointer:	4
pointer:	5
pointer:	null
pointer:	null

What if a crash or power outage occurs between writes?

If Only a Single Write...

- Just the data block (D2) is written to disk
  - Data is written, but no way to get to it - in fact, D2 still appears as a free block
  - Write is lost, but FS (meta)data structures are consistent
- Just the updated inode (Iv2) is written to disk
  - If we follow the pointer, we read garbage
  - File system inconsistency: data bitmap says block is free, while inode says it is used. Must be fixed
- Just the updated bitmap is written to disk
  - File system inconsistency: data bitmap says data block is used, but no inode points to it. The block will never be used. Must be fixed

If Two Writes...

- Inode and data bitmap updates succeed
  - Good news: file system is consistent!
  - Bad news: reading new block returns garbage
- Inode and data block updates succeed
  - File system inconsistency. Must be fixed
- Data bitmap and data block succeed
  - File system inconsistency
  - No idea which file data block belongs to!

The Consistent Update Problem

- Several file systems operations update multiple data structures
  - Create new file
    - update inode bitmap and data bitmap
    - write new inode
    - add new file to directory file
- Would like to atomically move FS from one consistent state to another
- Even with write through we have a problem
  - Disk only commits one write at a time!

Solution 1: File System Checker

- Ethos: If it happens, I'll do something about it
  - Let inconsistencies happen and fix them post facto
    - during reboot
- Classic example: fsck
  - Unix, 1986

FSCK Summary

- Sanity check the  uperblock for corruption
  - Is FS size larger than total blocks that have been allocated?
  - Is FS size "reasonable"?
  - On inconsistencies,
    - use another copy of the  uperblock
    - overwrite values in SB with those found in the file system

FSCK Summary

- Sanity check the superblock
- Check validity of free block and i-node bitmaps
  - Scan i-nodes, indirect blocks, etc to understand which blocks are allocated
  - On inconsistency, i-nodes win: override free block bitmap
  - Perform similar check on i-nodes to update inode bitmap

FSCK Summary

- Sanity check the superblock
- Check validity of free block and i-node bitmaps
- Check that i-nodes are not corrupted
  - e.g., check type (dir, regular file, symlink, etc) field
  - if issues can't be fixed, clear i-node and update i-node bitmap

FSCK Summary

- Sanity check the superblock
- Check validity of free block and inode bitmaps
- Check that i-nodes are not corrupted
- Check i-node hard links
  - Scan through the entire directory tree, recomputing the number of hard links for each file
  - If inconsistency, fix link count in inode
  - If no directory refers to allocated inode, move to lost+found directory

FSCK Summary

- Sanity check the superblock
- Check validity of free block and inode bitmaps
- Check that i-nodes are not corrupted
- Check i-node hard links
- Check for duplicate blocks, bad pointers
  - two inodes pointing to the same block
    - clear one inode (if bad), or copy block (to each, its own!)
  - pointer pointing to a node outside partition

FSCK Summary

- Sanity check the superblock
- Check validity of free block and i-node bitmaps
- Check that i-nodes are not corrupted
- Check i-node hard links
- Check for duplicate blocks, bad pointers
- Check directories
  - Check that . and .. are the first entries
  - Check that each i-node referred to is allocated
  - Check that directory tree is a tree
    - directory files must have a single link

FSCK Summary

- Sanity check the superblock
- Check validity of free block and i-node bitmaps
- Check that i-nodes are not corrupted
- Check i-node hard links
- Check for duplicate blocks, bad pointers
- Check directories

S-L-O-W

Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work
- Rely on metadata consistency
  - updating a file in vi
    - delete old file
    - write new file

Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work
- Rely on metadata consistency
  - updating a file in vi
    - write new version to temp
    - move old version to other temp
    - move new version to real file
    - unlink old version
      - if crash, look in temp area and send "there may be a problem" email to user

Solution 2: Ordered Updates

- Three rules towards a (quickly) recoverable FS:
  - Never reuse a resource before nullifying all pointers to it (e.g., nullify an i-node pointer to a data block before reallocating that block to another i-node)
  - Never point to a structure before it has been initialized (e.g., must initialize i-node before a directory entry references it)
  - Never clear last pointer to live resource before setting a new one (e.g., when renaming a file, do not remove old name for an i-node until after new name has been written)
- How?
  - Keep a partial order on buffered blocks

Solution 2: Ordered Updates

- Example: Create file A:
  - Create file A in i-node block X and directory block Y
- "Never point to a structure before it has been initialized"
  - Y cannot be written before X is
  - Y depends on X $Y \rightarrow X$
- Can delay both writes, as long as order is preserved
  - Suppose you create a second file B in blocks X and Y
  - Can write each block only once to cover both creates!

Problem: Cyclic Dependencies

- Suppose you create file A, unlink file B
  - Both files in same directory block & i-node block

(Never point to a structure (A's i-node) before it has been initialized)

- Can't write directory block until i-node A initialized
  - Or, after crash, directory will point to bogus i-node
  - Worse, same i-node no. might be reallocated
    - could end up with file name A being an unrelated file

(Never reuse a resource (B's i-node) without nullifying all pointers to it)

- Can't write i-node block until dir entry B cleared
  - Or B's link count could become smaller than directory entries
  - File could be deleted while link to it still exists in directory

Soft Updates

(Ganger et al.)

- "Soft Updates: A Solution to the Metadata Update Problem in File Systems" ACM TOCS, May 2000
  - tracks dependencies at a finer granularity
  - clever and complex

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
- 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 | Bv2 | D2 | TxEnd (includes TxID and blocks' final addresses)
    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, Bv2, D2 to the FS proper [a.k.a checkpoint]
  - 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 sequentially is slow TxBegin | Iv2 | B2 | D2 | TxEnd
  - 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!
- TxEnd must block until prior blocks are on disk
  - Transaction committed when TxEnd on disk

Back to

Where is this from?