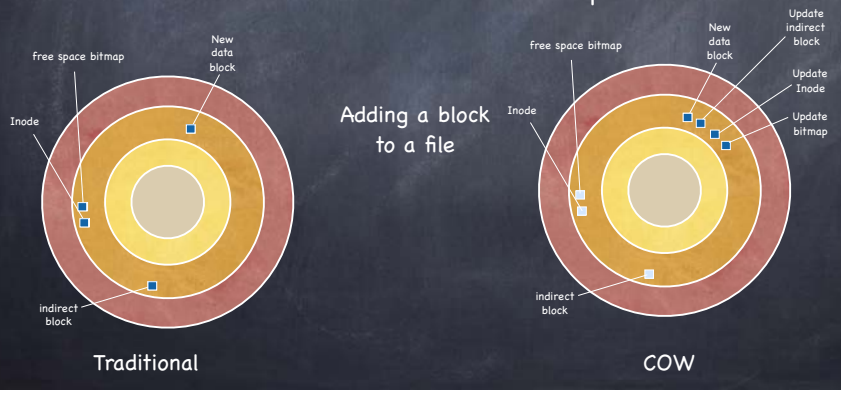
COW File Systems (copy-on-write)

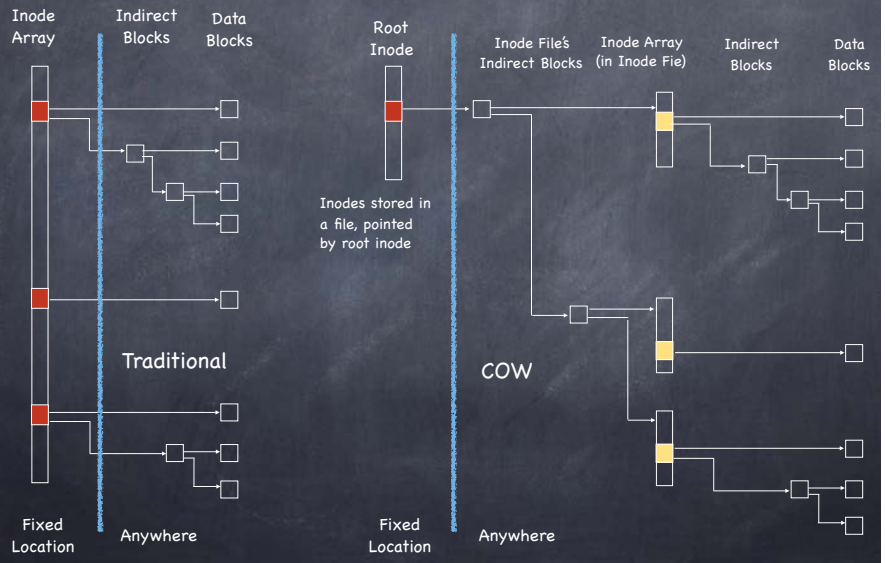
- Data and metadata not updated in place, but written to new location
  - transforms random writes into sequential writes



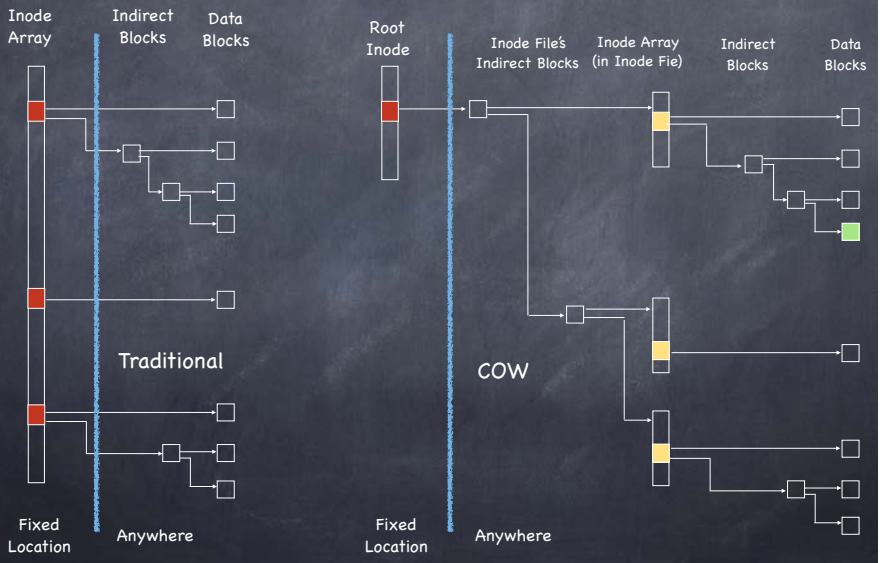
Why COW File Systems?

- Small writes are expensive
  - With RAID, an update requires four disk I/Os
- Caches filter reads
  - More important to make writes efficient
- Widespread adoption of flash storage
  - writing in place a 4KB page wold require erasing a 512KB erasure block
  - wear leveling, which spreads writes across all cells, important to maximize flash life
  - COW techniques used to virtualize block addresses and redirect writes to cleared erasure blocks
- Large storage capacities enable versioning
  - versioning is easy with COW!

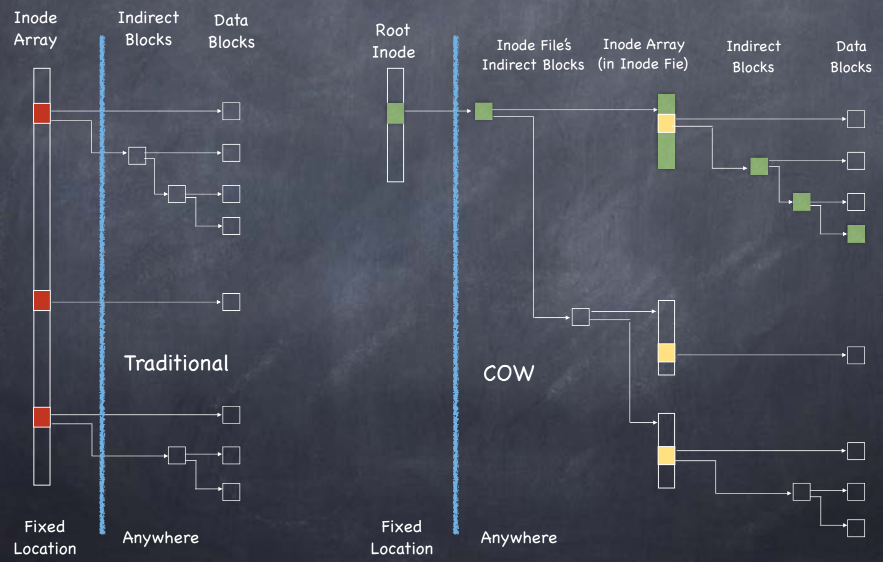
The core idea



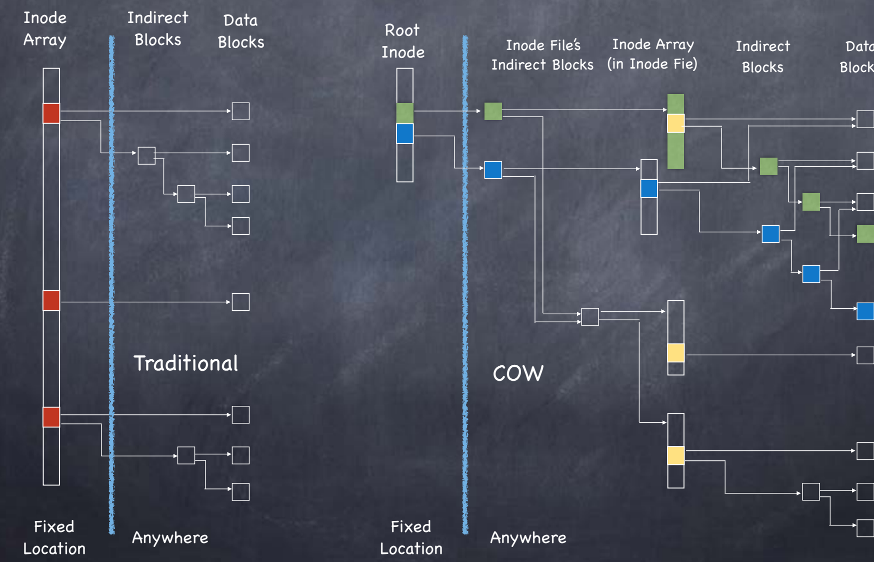
The core idea



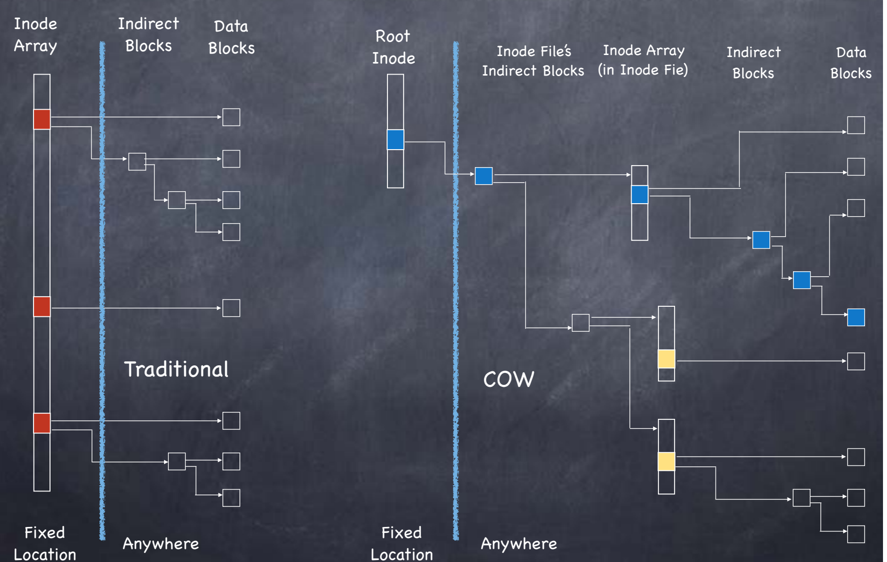
The core idea



The core idea



The core idea



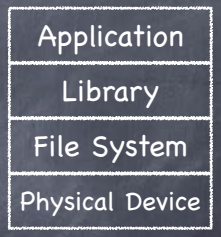
File access in FFS

- What does it take to read /Users/lorenzo/wisdom.txt?
  - Read Inode for "/" (root) from a fixed location
  - Read first data block for root
  - Read Inode for /Users
  - Read first data block of /Users
  - Read Inode for /Users/lorenzo
  - Read first data block for /Users/lorenzo
  - Read Inode for /Users/lorenzo/wisdom.txt
  - Read data blocks for /Users/lorenzo/wisdom.txt

"Cache is a man's best friend"

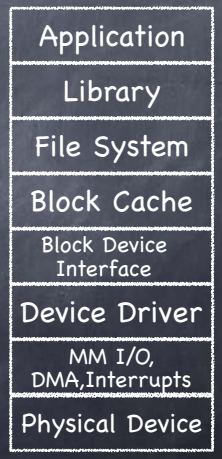
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions



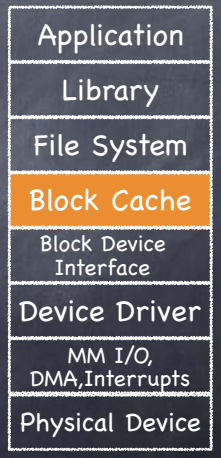
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions



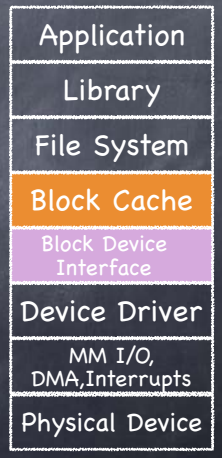
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches recently read blocks
  - Buffers recently written blocks



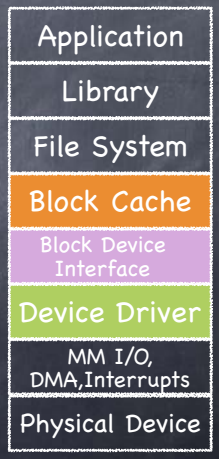
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches recently read blocks
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized block



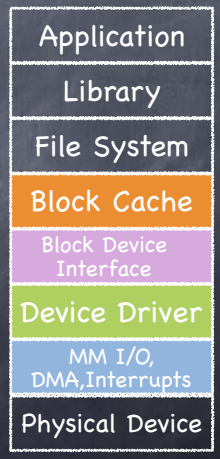
The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches recently read blocks
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized block
  - Translates OS abstractions and hw specific details of I/O devices



The Abstraction Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches recently read blocks
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized block
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications



Caching and consistency

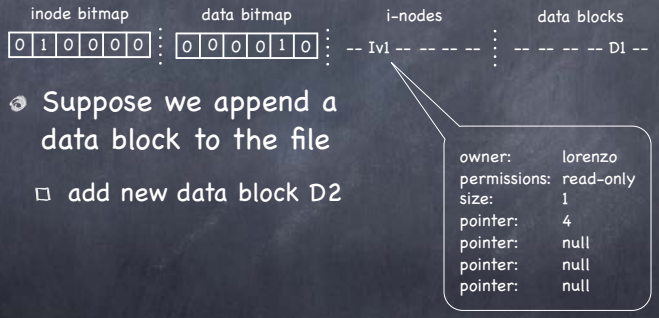
- File systems maintain many data structures
  - Bitmap of free blocks and inodes
  - Directories
  - Inodes
  - Data blocks
- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?

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?

Example: a tiny ext2

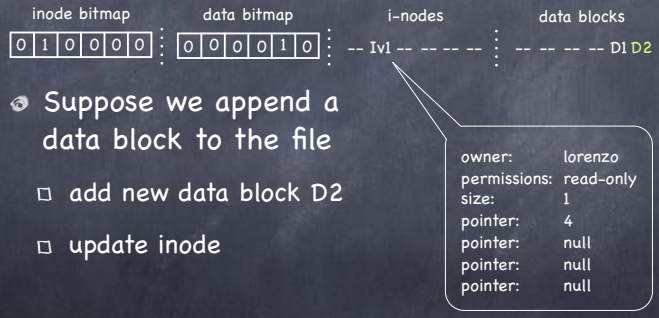
- 6 blocks, 6 inodes



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

Example: a tiny ext2

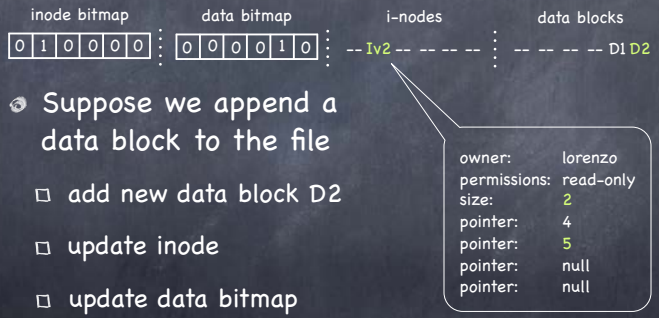
- 6 blocks, 6 inodes



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

Example: a tiny ext2

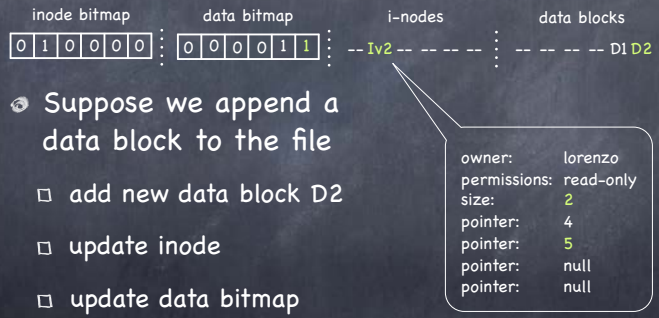
- 6 blocks, 6 inodes



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

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

What if a crash or power outage occurs between writes?

What if only a single write succeeds?

- 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 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

What if two writes succeed?

- 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 superblock

The Superblock

- One logical superblock per file system, at a well-known location(s). It contains
  - size of FS
  - list of free blocks (today, a bitmap)
  - number of free blocks and index of next free block
  - size of inode list
  - number of free nodes and index of next free node
  - locks for free block and free node lists
  - flag to indicate superblock has been modified

FSCK Summary

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

FSCK Summary

- Sanity check the superblock
- Check validity of free block and inode bitmaps
- Check that inodes are not corrupted
  - e.g., check type (dir, regular file, symbolic link) field
  - if it can't be fixed, clear inode and update inode bitmap

FSCK Summary

- Sanity check the superblock
- Check validity of free block and inode bitmaps
- Check that inodes are not corrupted
- Check inode links
  - Scan through the entire directory tree, recomputing the number of 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 inodes are not corrupted
- Check inode links
- Check for duplicates
  - two inodes pointing to the same block

FSCK Summary

- Sanity check the superblock
- Check validity of free block and inode bitmaps
- Check that inodes are not corrupted
- Check inode links
- Check for duplicates
- Check directories
  - Check that . and .. are the first entries
  - Check that each inode 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 inode bitmaps
- Check that inodes are not corrupted
- Check inode links
- Check for duplicates
- 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
  - Never write a pointer before initializing the structure it points to
  - Never clear last pointer to live resource before setting a new one
- How?
  - Keep a partial order on buffered blocks

Solution 2: Ordered Updates

- Example: Create file A:
  - Create file A in inode block X and directory block Y
- "Never write a pointer before initializing the structure it points to"
  - 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
  - Must 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 & inode block
- Can't write directory until inode A initialized
  - Or after crash directory will point to bogus inode
  - Worse, same inode no. might be reallocated
  - So, could end up with file name A being an unrelated file
- Can't write inode 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 exist