Case studies

- FAT late 70s; Microsoft
  - key idea: linked list
  - Today: flash sticks
- Unix FFS mid 80's
  - key idea: tree-based multi-level index
  - Today: Linux ext2 and ext3
- NTFS early 1990s; Microsoft.
  - Key idea: variable size extents instead of fixed size blocks
  - Today: Windows 7, Linux ext4, Apple HFS
- ZFS early 2000; open source.
  - Key idea: copy on write (COW)

FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}–1 bytes

Index Structures

File Allocation Table (FAT)

- array of 32-bit entries
- file represented as a linked list of FAT entries
- file # = index of first FAT entry

Free space map

- If data block i is free, then FAT[i] = 0
- find free blocks by scanning FAT

Locality heuristics

- As simple as next fit:
  - scan sequentially from last allocated entry and return next free entry
- Can be improved through defragmentation

FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}–1 bytes

Advantages

- simple!
  - used in many USB flash keys
  - used even within MS Word!

Disadvantages

- Poor locality
  - next fit? seriously?
- Poor random access
  - needs sequential traversal
- Limited access control
  - no file owner or group ID metadata
  - any user can read/write any file
- No support for hard links
  - metadata stored in directory entry
- Volume and file size are limited
  - FAT entry is 32 bits, but top 4 are reserved
  - no more than 2^{28} blocks
  - with 4kB blocks, at most 1TB volume
  - file no bigger than 4GB
- No support for transactional updates

FFS: Fast File System

Unix, 80s

- Smart index structure
  - multilevel index allows to locate all blocks of a file
    - efficient for both large and small files [We saw that!]
- Smart locality heuristics

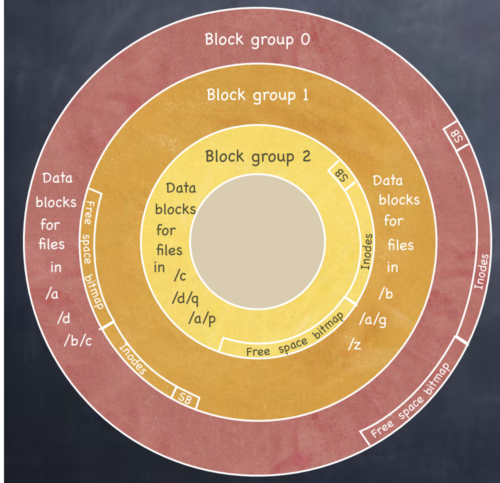
  - Standard Unix treats disks as if it were RAM
    - lots of seeks
    - fragmentation: files just grab first available data block

Making the FS Disk-Aware

- Maintain the same interface...
  - open(), close (), read(), write () etc
- ...but change implementation
  - optimize file system layout for how disks work
- Smart locality heuristics
  - block group placement
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space
    - gives up about 10% of storage to allow flexibility needed to achieve locality

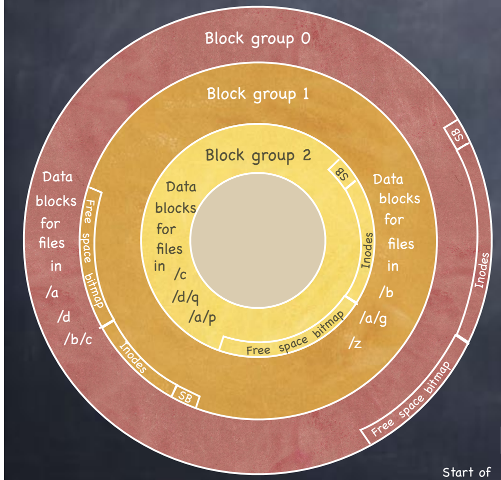
---

Locality heuristics: block group placement



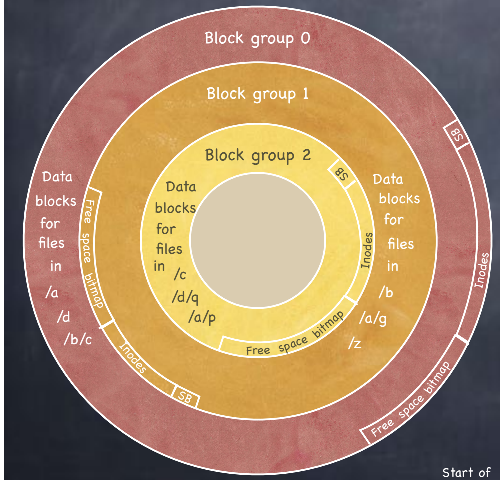
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File placement
  - when a new regular file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality

Locality heuristics: block group placement



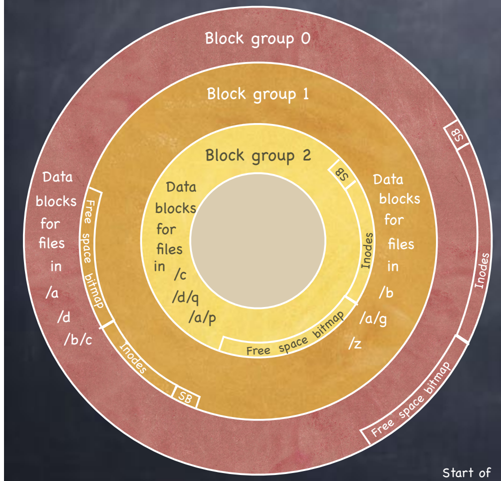
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality

Locality heuristics: block group placement

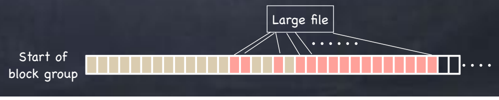


- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality

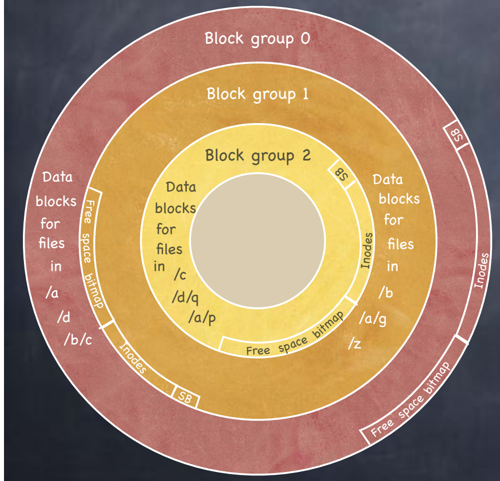
Locality heuristics: block group placement



- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups. Keep a superblock copy in each block group
- File Placement
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS places it in a different block from the parent's directory
- Data Placement
  - first free heuristics
  - trade short term for long term locality



Locality heuristics: reserved space



- When a disk is full, hard to optimize locality
  - file may end up scattered through disk
- FFS presents applications with a smaller disk
  - about 10%-20% smaller
  - user write that encroaches on reserved space fails
  - super user still able to allocate inodes to clean things up

Long File Exception

- Blocks of a huge file not all in the same block group
  - 12 blocks in a group (direct index)
  - others divided in "chunks"
- Locality lost when moving between chunks
  - choose chunk size to amortize cost of seeks

Say we want to achieve 90% of peak transfer
- transfer rate is 40 MB/s
- positioning time (seek+rotation) is 10ms

$$\text{chuck size} = \frac{40\text{MB}}{\text{s}} \times \frac{1\text{s}}{1000\text{ms}} \times 90\text{ms} = 3.6\text{ MB}$$

- In practice, FFS uses 4 MB chunks

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?

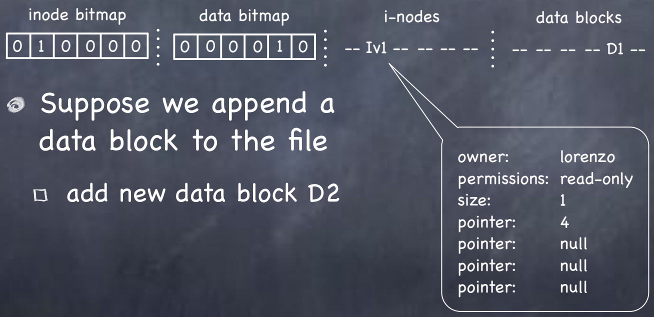
Crash Consistency

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



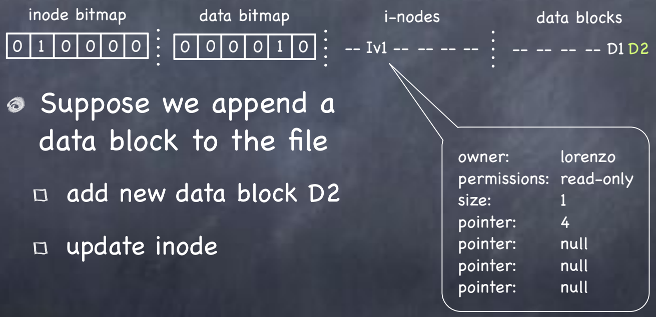
inode bitmap	data bitmap	i-nodes	data blocks
0 1 0 0 0 0	0 0 0 0 1 0	-- Iv1 -- -- -- --	-- -- -- -- D1 --

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

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

Example: a tiny ext2

- 6 blocks, 6 inodes



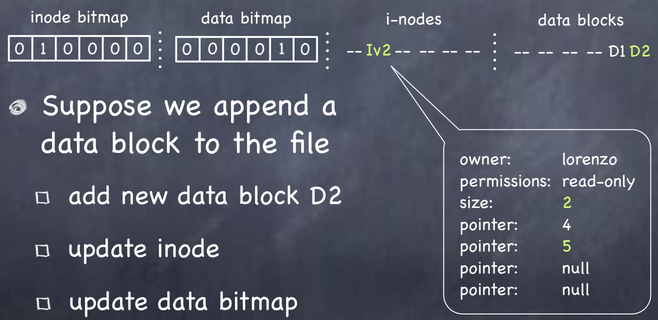
inode bitmap	data bitmap	i-nodes	data blocks
0 1 0 0 0 0	0 0 0 0 1 0	-- Iv1 -- -- -- --	-- -- -- -- D1 D2

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

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

Example: a tiny ext2

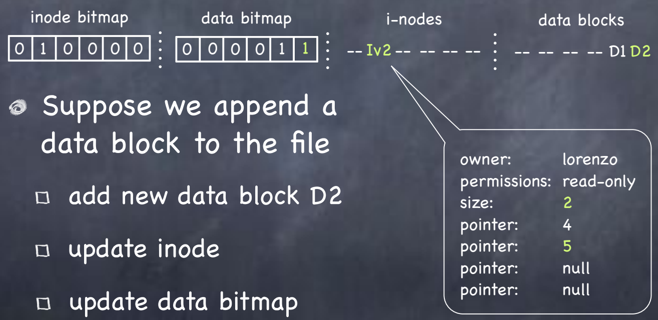
- 6 blocks, 6 inodes



- 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?

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 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 Superblock
  - Is FS size larger than total blocks used by superblock + inodes?
  - Is FS size "reasonable"?
  - Is the number of free blocks and inodes in the Superblock equal to the number of free blocks and inodes in the file system?
  - On inconsistencies,
    - use another copy of the Superblock
    - overwrite values in SB with those found in the file system

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, symlink, etc) 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
        - clear one inode (if bad), or copy 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 $Y \to X$
- 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 & 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
    - 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 in directory

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

inode bitmap	data bitmap	i-nodes	data blocks
0 | 1 | 0 | 0 | 0 | 0	0 | 0 | 0 | 0 | 1 | 0	-- Iv1 -- -- -- --	-- -- -- -- D1 --

- 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 [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 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!
- TxEnd must block until prior blocks are on disk
    - Transaction committed when TxEnd on disk