File System Implementation

Nima Honarmand

File Systems

- "FS", FFS, FAT, ext2/3/4, NTFS, …
- View disk as an array of ***blocks***
  - Each block contains one or more disk ***sectors***
    - Sectors are conventionally 512 bytes (larger in some newer "green" disks)
  - Sectors can be accessed randomly
  - Sector read/writes are atomic
- Block is FS-level concept; sector is disk-level concept
  - E.g., in Linux each disk block is 4KB (to match the page size)
- We'll just work with blocks for now
- Older FSes cared a lot about the detailed geometry of sectors on disk; new ones don't care much about the details anymore
  - Disk geometry: where sector is **x** located on the drive
- Why?
  - Because new drives do not expose much information about their geometry
  - All we know is consecutive accesses are faster than random accesses
    - Sequential vs. Random access patterns

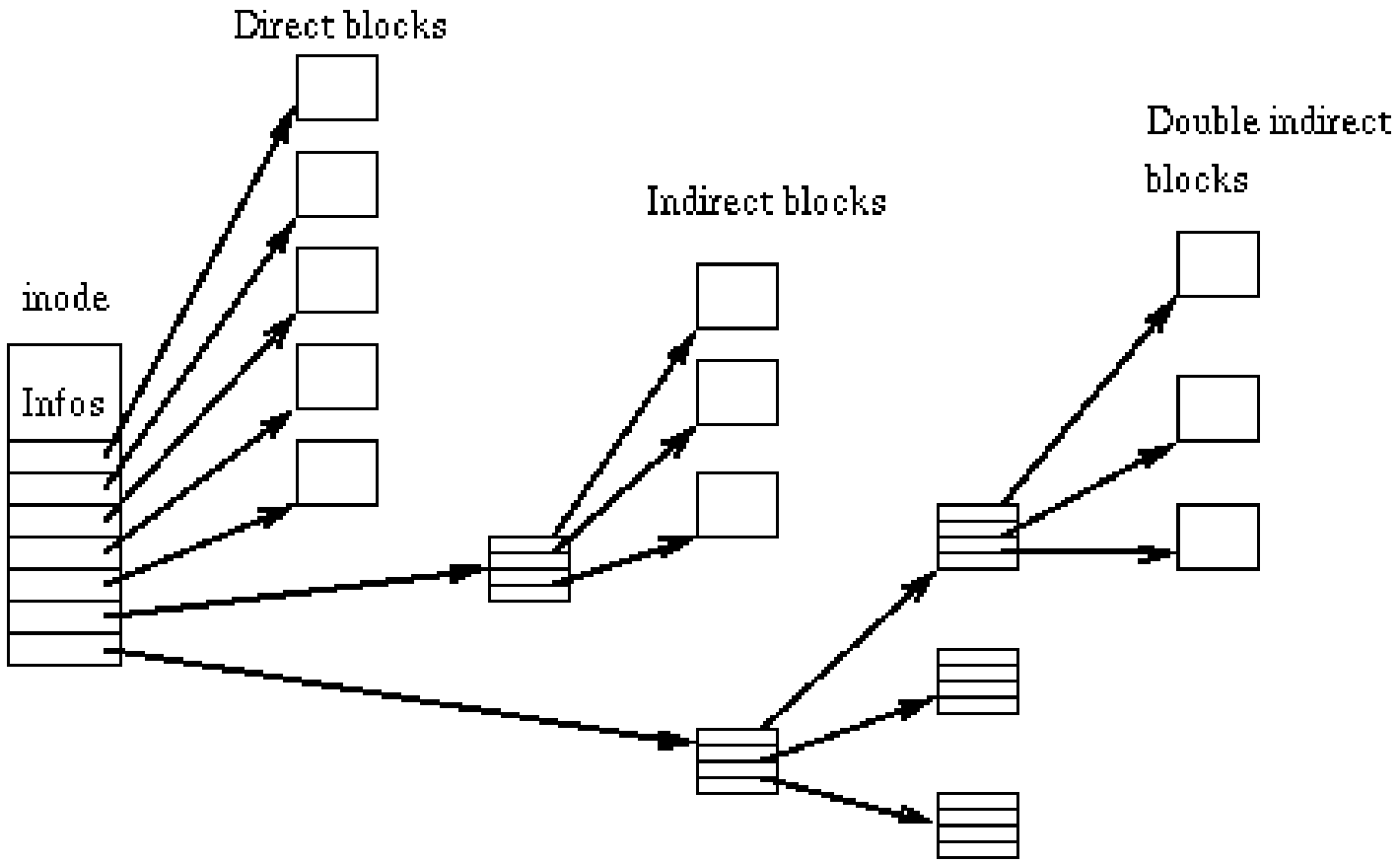
File System Data Structures

- Need to keep several data (structures) on disk
  - Details differ from one FS to the next
- ***Data blocks***
  - File contents
- ***Metadata blocks***
  - Superblock: global FS-level metadata
    - ***magic*** value to identify filesystem type
    - Places to find metadata on disk, e.g.
      - inode array
      - Free data block list/bitmap
      - Free inode list/bitmap
  - inodes: per-file metadata
    - Attributes (e.g., file or directory, size)
    - Pointers to data blocks
- Keep in mind: almost every FS operation involves accessing (reading and/or writing) both metadata and data blocks

ext2

- Very reliable, “best-of-breed” traditional file system design
- Much like the JOS file system you are building now
  - Fixed location Superblocks
  - Pre-allocate, easy to find inodes on disk using their number
  - A few direct blocks in the inode, followed by indirect blocks for large files
  - Directories are a special file type with a list of (file name, inode number) entries
  - Etc.

Locating/Allocating Blocks



Source: Wikipedia article on ext2

File Systems and Crashes

- What can go wrong?
  - Write a block pointer in an inode
    … before marking block as used in bitmap
  - Write a reclaimed block into an inode
    … before removing old inode that points to it
  - Allocate an inode
    … without putting it in a directory
    - Inode is “orphaned”
  - etc.

Deeper Issue

- Operations span multiple on-disk data structures
  - Requires more than one disk write
    - Multiple disk writes not performed together
      - Single sector writes aren't guaranteed either (e.g., power loss)
- Disk writes are always a series of updates
  - System crash can happen between any two updates
    - Crash between dependent updates leaves structures inconsistent!
- Writes are not sent to disk immediately
  - Almost everything is cached in memory: superblocks, inodes, free-list bitmaps, data blocks (page cache)

Atomicity

- Property that something either happens or it doesn't
  - No partial results
- Desired for disk updates
  - Either inode bitmap, inode, *and* directory are all updated
    - … or none of them are
- Preventing corruption is fundamentally hard
  - If the system is allowed to crash

Solutions 1: fsck

- When file system mounted, mark on-disk superblock
  - If system is cleanly shut down, last disk write clears this bit
  - If the file system isn't cleanly unmounted, run ***fsck***
- Does linear scan of all bookkeeping
  - Checks for (and fixes) inconsistencies
  - Puts orphaned pieces into /lost+found

fsck Examples

- Walk directory tree
  - Make sure each reachable inode is marked as allocated
- For each inode, check the data blocks
  - Make sure all referenced blocks are marked as allocated
- Double-check that allocated blocks and inodes are reachable
  - Otherwise should not be allocated (should be in free list)
- Summary: very expensive, slow scan of file system
  - Can take many hours on a large partition

Solution 2: Journaling

- Idea: Keep a log of metadata operations
  - On system crash, look at data structures that were involved
- Limits the scope of recovery
  - Recovery faster than fsck
    - Cheap enough to be done while mounting

Two Ways to Journal (Log)

- Two main choices for a journaling scheme
  - (Borrowed/developed along with databases)
  - Often referred to as ***logging***
    - Called ***journaling*** for filesystems
- Undo Logging: write how to go back to sane state
- Redo Logging: write how to go forward to sane state
- In all cases, a ***Transaction*** is the set of changes we are going to make to service a high-level operation
  - E.g., a write() or a rename() system call

Undo Logging

1. Write what you are about to do (and how to undo)
   - "How to undo" is basically the content of disk block before the write
2. Make changes on rest of disk
3. Write ***commit record*** of the transaction to log
   - Marks logged operations as complete

- If system crashes before (3)
  - Execute undo steps when recovering

- Undo steps must be on disk before other changes

Redo Logging

1. Write planned operations (disk changes) to the log
   - At the end, write a commit record
2. Make changes on rest of disk
3. When updates are done, mark transaction entry obsolete

- If system crashes during (2) or (3)
  - Re-execute all steps when recovering
- ext3 uses redo logging

Batching of Journal Writes

- Journaling would require many ordered writes
  - Ordered writes are expensive
    - Have to wait until first one completes before sending a second one
  - Significantly reduce disk bandwidth utilization
- Can batch multiple transactions into big one
  - Use a heuristic to decide on transaction size
    - Wait up to 5 seconds
    - Wait until disk block in the journal is full
- Batching reduces number of entries and thus ordered writes

Journaling Modes

- ***Full journaling***
  - Both data + metadata in the journal
  - Lots of data written twice, safer
- ***Metadata journaling + ordered data writes***
  - Only metadata in the journal
  - Data writes only allowed before metadata is in journal
    - Why not after?
    - Because inode can point to garbage data if crash
  - Faster than full data, but constrains write orderings
- ***Metadata journaling + unordered data writes***
  - Fastest, most dangerous
  - Data write can happen anytime w.r.t. metadata journal

ext4

- ext3 has some limitations
  - Ex: Can't work on large data sets
    - Can't fix without breaking backwards compatibility
- ext4 removes limitations
  - Plus adds a few features

Example

- ext3 limited to 16 TB max size
  - 32-bit block numbers (2^{32} * 4k block size)
  - Can't make bigger block sizes on disk
  - Can't fix without breaking backwards compatibility
- ext4 – 48 bit block numbers

Indirect Blocks vs. Extents

- Instead of representing each block, represent contiguous chunks of blocks with an ***extent***

- ext4 supports extents
  - 4 extents stored in the inode; if more needed store them in a tree

+ More efficient for large files
  - Ex: Disk blocks 50—300 represent blocks 0—250 of file
    - v.s. allocating and initializing 250 slots in direct/indirect blocks
    - Deletion requires marking 250 slots as free

- Worse for highly fragmented or sparse files
  - If no contiguous blocks, need one extent for each block
  - Basically a more expensive indirect block scheme

Static inode Allocations

- When ext3 or ext4 file system created
  - Create all possible inodes
    - Can't change count after creation
- If need many files, format for many inodes
  - Simplicity
    - Fixed inode locations allows easy lookup
    - Dynamic tracking requires another data structure
      - What if that structure gets corrupted?
    - Bookkeeping more complicated when blocks change type
  - Downsides
    - Wasted space if inode count is too high
    - Available capacity, but out of space if inode count is too low
- Some FSes allow dynamic inode allocation (e.g., XFS)

Directory Scalability

- ext3 directory can have 32,000 sub-directories/files
  - Painfully slow to search
    - Just a simple array on disk (linear scan to lookup a file)
- ext4 replaces structure with an HTree
  - Hash-based custom Btree
  - Allows unlimited directories
  - Relatively flat tree to reduce risk of corruptions
  - Big performance wins on large directories – up to 100x