Ext(2/3/4) File Systems

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)

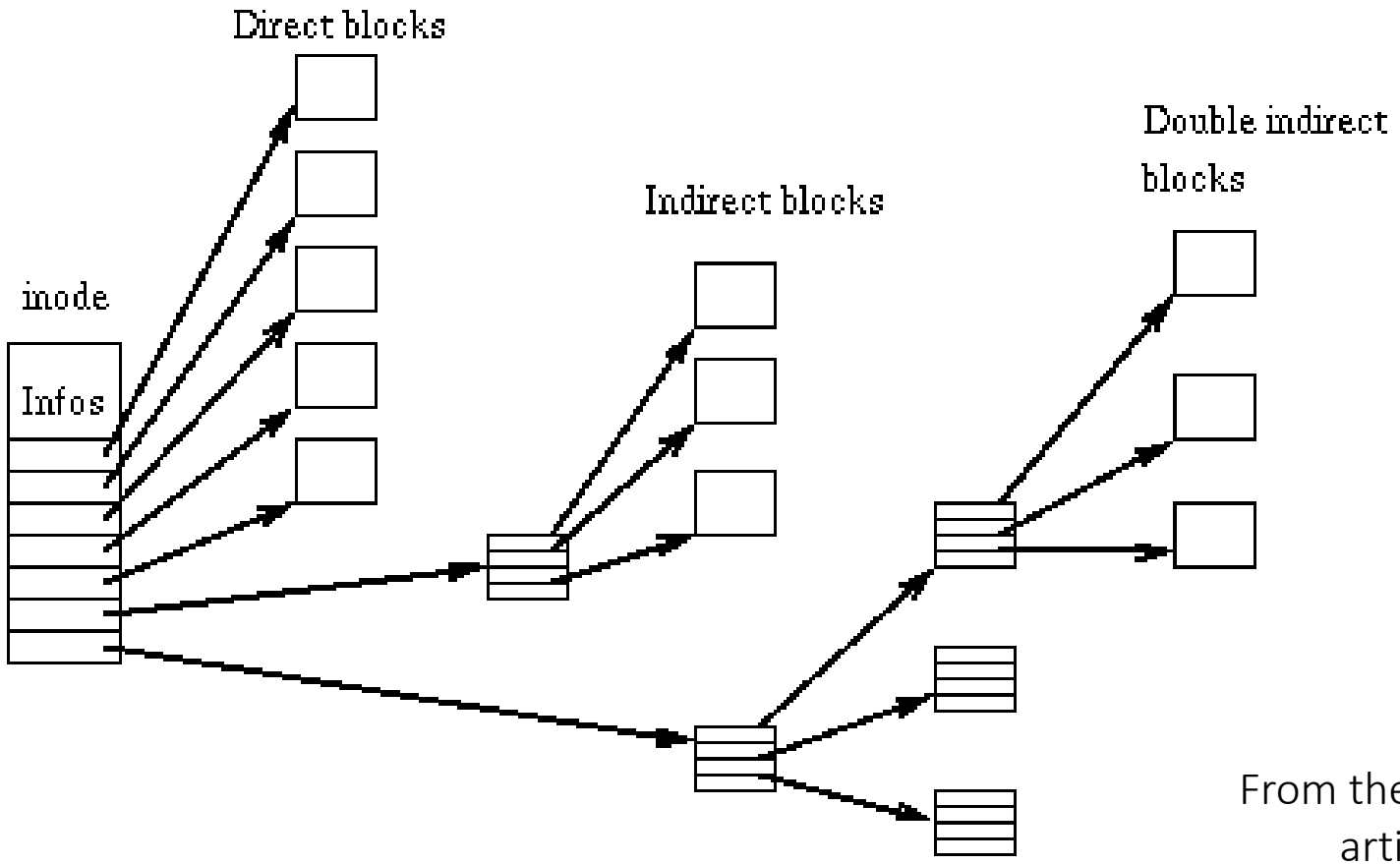
Traditional File Systems

- "FS", UFS/FFS, Ext2, …
- Several simple on disk structures
  - Superblock
    - ***magic*** value to identify filesystem type
    - Places to find metadata on disk (e.g., inode array, free block list)
  - inode array (metadata blocks)
    - Attributes (e.g., file or directory, size)
    - Pointers to data blocks
  - Data blocks
    - File contents

Ext2

- Very reliable, “best-of-breed” traditional file system design
- Much like the JOS file system you are building now
  - Fixed location super blocks
  - 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 names and inode numbers
  - Etc.

Locating/Allocating Blocks



From the 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!

Atomicity

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

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

Journaling

- Idea: Keep a log of metadata operations
  - On system crash, look at data structures that were involved
- Limits the scope of recovery
  - Faster 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 (usually metadata only)
- 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

Journaling Used in Practice

- Ext3 uses redo logging
  - Mostly, to help with corner cases caused by delete operations
    - According to Stepehn Tweedie (Ext3 reading)
- Easier to defer taking something apart … than to put it back together later
  - Hard case: delete something and reuse a block for something else before journal entry commits
  - What could go wrong with undo logging?

Batching of Journal writes

- Journaling would requires many synchronous writes
  - Synchronous writes are expensive
- 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 synchronous writes

Journaling modes

- Full data + metadata in the journal
  - Lots of data written twice, safer
- Ordered writes
  - Only metadata in the journal, but data writes only allowed after metadata is in journal
  - Faster than full data, but constrains write orderings (slower)
- Metadata only – fastest, most dangerous
  - Can write data to a block before it is properly allocated to a file

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***
- More efficient for large files
  - Ex: Disk blocks 50—300 represent blocks 0—250 of file
    - vs: Allocate and initialize 250 slots in an indirect block
    - 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

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
  - Relatively flat tree to reduce risk of corruptions
  - Big performance wins on large directories – up to 100x