# CSE 506: Operating Systems

File Systems

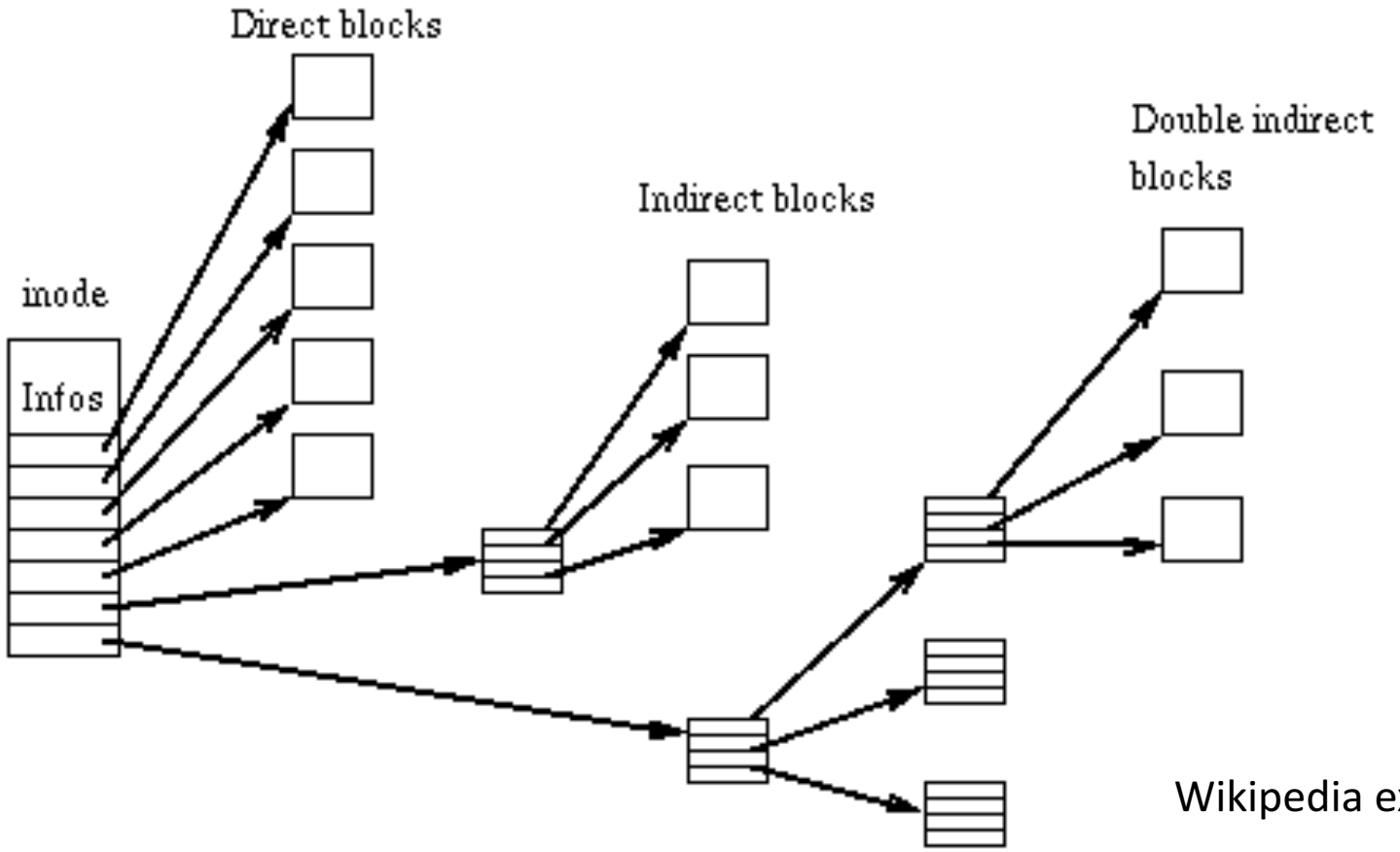
# Traditional File Systems

- "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
    - Attributes (e.g., file or directory, size)
    - Pointers to data blocks
      - Several ***direct*** blocks for small files
      - ***{Singly, Doubly, Triply}-Indirect*** blocks for large files
  - Blocks
    - File contents

# Working with a File System

- Need to ***format*** disk prior to use
  - Write a superblock
    - With correct magic number
  - Write details about disk size/number of blocks
    - Need a free list or bitmap
  - Write first several inodes
    - Usually “root” directory inode has designated index (e.g., “2”)
- Done with ***newfs***
  - Works on raw device (via /dev/diskdriver)
  - For course project, format filesystem on first mount
    - Avoid the hassle of user access to raw devices in your OS

# Locating/Allocating Blocks




Wikipedia ext2

# Tracking Free Objects on Disk

- Use blocks pointed to from inode
  - On erase, must replace freed blocks onto free "list"
- Disk size traditionally known in advance
  - Disk maintains list of free blocks
    - Easy to keep track of in a bitmap
  - Virtual machine disks can be resized
    - Requires resizing filesystem to accept new blocks
      - Add elements to free list or mark bits in free map
- Need to maintain list of free inodes too
  - Otherwise must probe inode map for free slot
  - Superblock should remember head of list

# 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 reference count
  - Make sure all referenced blocks are marked as allocated
- Double-check that blocks and inodes are reachable
  - Or in free list
- Summary: very expensive, slow scan of file system

# 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: write how to go back to sane state
- Redo: write how to go forward to sane state

# Undo Logging

1. Write what you are about to do (and how to undo)
2. Make changes to rest of disk
3. Write ***commit record*** to log
   - Marks logged operations as complete

- If system crashes before log commit record
  - Execute undo steps when recovering
- Undo steps *must* be on disk before other changes

# Redo Logging

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

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

# Journaling Used in Practice

- Ext3 uses redo logging
- Easier to defer taking something apart
  … than to put it back together later
  - Delete something
  - Reuse a block for something else
    - Before journal entry commits
- Only works if data comfortably fits into memory
  - Databases often use undo logging
    - Avoid loading and writing large data sets twice

# Atomicity Strategies

- Write journal log entry to disk
  - Include transaction number (sequence counter)
  - Write global counter to indicate log entry was written
    - This write is point at which journal is "committed"
      - Sometimes called a linearization point
    - Either the sequence number is written or not
      - Sequence number not written until log entry is on disk
- Can also overwrite same spot at the end of log entry
  - First write entry with "incomplete" flag
  - Second entry with identical contents and "complete" flag

# Batching of Journal writes

- Journaling would requires many synchronous writes
  - Synchronous writes are expensive
- Can batch multiple transactions into big one
  - Assuming no fsync()
  - 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

# 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 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 look up 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