

- **Reminders for this Friday:**
 - VM lab due
 - Section to go over final lab

Outline

- 1 FFS in more detail
- 2 Crash recoverability
- 3 Soft updates
- 4 Journaling

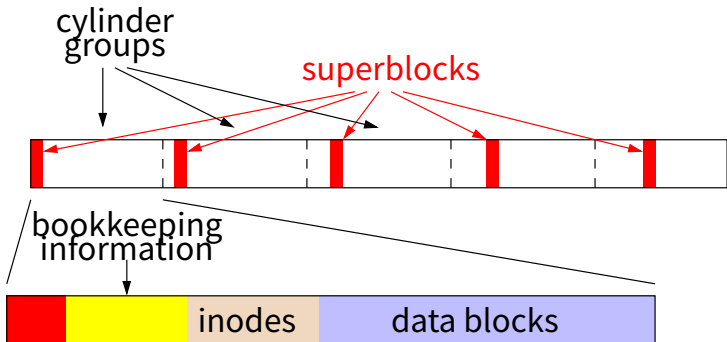
Review: FFS background

- **1980s improvement to original Unix FS, which had:**
 - 512-byte blocks
 - Free blocks in linked list
 - All inodes at beginning of disk
 - Low throughput: 512 bytes per average seek time
- **Unix FS performance problems:**
 - Transfers only 512 bytes per disk access
 - Eventually random allocation → 512 bytes / disk seek
 - Inodes far from directory and file data
 - Within directory, inodes far from each other
- **Also had some usability problems:**
 - 14-character file names a pain
 - Can't atomically update file in crash-proof way

Review: FFS [McKusic] basics

- **Change block size to at least 4K**
 - To avoid wasting space, use “fragments” for ends of files
- **Cylinder groups spread inodes around disk**
- **Bitmaps replace free list**
- **FS reserves space to improve allocation**
 - Tunable parameter, default 10%
 - Only superuser can use space when over 90% full
- **Usability improvements:**
 - File names up to 255 characters
 - Atomic *rename* system call
 - Symbolic links assign one file name to another

Review: FFS disk layout



- Each cylinder group has its own:
 - Superblock
 - Bookkeeping information
 - Set of inodes
 - Data/directory blocks

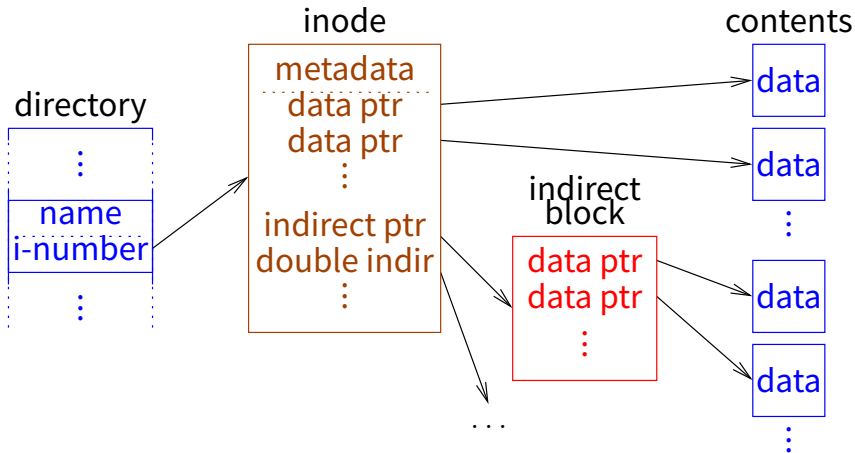
Superblock

- **Contains file system parameters**
 - Disk characteristics, block size, CG info
 - Information necessary to locate inode given i-number
- **Replicated once per cylinder group**
 - At shifting offsets, so as to span multiple platters
 - Contains magic number 0x011954 to find replicas if 1st superblock dies (Kirk McKusick's birthday?)
- **Contains non-replicated “summary information”**
 - # blocks, fragments, inodes, directories in FS
 - Flag stating if FS was cleanly unmounted

Bookkeeping information

- **Block map**
 - Bit map of available fragments
 - Used for allocating new blocks/fragments
- **Summary info within CG**
 - # free inodes, blocks/frags, files, directories
 - Used when picking cylinder group from which to allocate
- **# free blocks by rotational position (8 positions)**
 - Was reasonable in 1980s when disks weren't commonly zoned
 - Back then OS could do stuff to minimize rotational delay

Inodes and data blocks



- Each CG has fixed # of inodes (default one per 2K data)
- Each inode maps **offset** → **disk block** for one file
- An inode also contains metadata for its file
 - permissions, access/modification/change times, link count, ...

Inode allocation

- **Each file or directory created requires a new inode**
- **New file? Put inode in same CG as directory if possible**
- **New directory? Use different CG from parent**
 - Consider CGs with greater than average # free inodes
 - Chose CG with smallest # directories
- **Within CG, inodes allocated randomly (next free)**
 - Would like related inodes as close as possible
 - OK, because one CG doesn't have that many inodes
 - All inodes in CG can be read and cached with small # of reads

Fragment allocation

- **Allocate space when user writes beyond end of file**
- **Want last block to be a fragment if not full-size**
 - If already a fragment, may contain space for write – done
 - Else, must deallocate any existing fragment, allocate new
- **If no appropriate free fragments, break full block**
- **Problem: Slow for many small writes**
 - May have to keep moving end of file around
- **(Partial) solution: new `stat` struct field `st_blksize`**
 - Tells applications file system block size
 - `stdio` library can buffer this much data

Block allocation

- **Try to optimize for sequential access**
 - If available, use rotationally close block in same cylinder (obsolete)
 - Otherwise, use block in same CG
 - If CG totally full, find other CG with quadratic hashing
i.e., if CG # n is full, try $n + 1^2, n + 2^2, n + 3^2, \dots \pmod{\#CGs}$
 - Otherwise, search all CGs for some free space
- **Problem: Don't want one file filling up whole CG**
 - Otherwise other inodes will have data far away
- **Solution: Break big files over many CGs**
 - But large extents in each CGs, so sequential access doesn't require many seeks
 - How big should extents be?

Block allocation

- **Try to optimize for sequential access**
 - If available, use rotationally close block in same cylinder (obsolete)
 - Otherwise, use block in same CG
 - If CG totally full, find other CG with quadratic hashing
i.e., if CG # n is full, try $n + 1^2, n + 2^2, n + 3^2, \dots$ (mod #CGs)
 - Otherwise, search all CGs for some free space
- **Problem: Don't want one file filling up whole CG**
 - Otherwise other inodes will have data far away
- **Solution: Break big files over many CGs**
 - But large extents in each CGs, so sequential access doesn't require many seeks
 - How big should extents be?
 - Extent transfer time should be much greater than seek time

Directories

- Directories have normal inodes with different type bits
- Contents considered as 512-byte *chunks*
- Each chunk has `direct` structure(s) with:
 - 32-bit inumber
 - 16-bit size of directory entry
 - 8-bit file type (added later)
 - 8-bit length of file name
- Coalesce when deleting
 - If first `direct` in chunk deleted, set inumber = 0
- Periodically compact directory chunks
 - But can never move directory entries across chunks
 - Recall only 512-byte sector writes atomic w. power failure

Updating FFS for the 90s

- **No longer wanted to assume rotational delay**
 - With disk caches, want data contiguously allocated
- **Solution: Cluster writes**
 - FS delays writing a block back to get more blocks
 - Accumulates blocks into 64KiB *clusters*, written at once
- **Allocation of clusters similar to fragments/blocks**
 - Summary info
 - Cluster map has one bit for each 64K if all free
- **Also read in 64K chunks when doing read ahead**

Outline

- 1 FFS in more detail
- 2 **Crash recoverability**
- 3 Soft updates
- 4 Journaling

Fixing corruption – fsck

- **Must run FS check (fsck) program after crash**
- **Summary info usually bad after crash**
 - Scan to check free block map, block/inode counts
- **System may have corrupt inodes (not simple crash)**
 - Bad block numbers, cross-allocation, etc.
 - Do sanity check, clear inodes containing garbage
- **Fields in inodes may be wrong**
 - Count number of directory entries to verify link count, if no entries but count $\neq 0$, move to `lost+found`
 - Make sure size and used data counts match blocks
- **Directories may be bad**
 - Holes illegal, `.` and `..` must be valid, file names must be unique
 - All directories must be reachable

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Example: Suppose all data written asynchronously**
 - Any subset of data structures may be updated before a crash
- **Delete/truncate a file, append to other file, crash**
 - New file may reuse block from old
 - Old inode may not be updated
 - Cross-allocation!
 - Often inode with older mtime wrong, but can't be sure
- **Append to file, allocate indirect block, crash**
 - Inode points to indirect block
 - But indirect block may contain garbage!

Sidenote: kernel-internal disk write routines

- **BSD has three ways of writing a block to disk**
 1. **bdwrite – delayed write**
 - Marks cached copy of block as dirty, does not write it
 - Will get written back in background within 30 seconds
 - Used if block likely to be modified again soon
 2. **bawrite – asynchronous write**
 - Start write but return immediately before it completes
 - E.g., use when appending to file and block is full
 3. **bwrite – synchronous write**
 - Start write, sleep and do not return until safely on disk

Ordering of updates

- **Must be careful about order of updates**
 - Write new inode to disk before directory entry
 - Remove directory name before deallocating inode
 - Write cleared inode to disk before updating CG free map
- **Solution: Many metadata updates synchronous** (`bwrite`)
 - Doing one write at a time ensures ordering
 - Of course, this hurts performance
 - E.g., `untar` much slower than disk bandwidth
- **Note: Cannot update buffers on the disk queue**
 - E.g., say you make two updates to same directory block
 - But crash recovery requires first to be synchronous
 - Must wait for first write to complete before doing second
 - Makes `bawrite` as slow as `bwrite` for many updates to same block

Performance vs. consistency

- **FFS crash recoverability comes at *huge* cost**
 - Makes tasks such as untar easily 10-20 times slower
 - All because you *might* lose power or reboot at any time
- **Even slowing normal case does not make recovery fast**
 - If fsck takes one minute, then disks get 10× bigger, then 100× ...
- **One solution: battery-backed RAM**
 - Expensive (requires specialized hardware)
 - Often don't learn battery has died until too late
 - A pain if computer dies (can't just move disk)
 - If OS bug causes crash, RAM might be garbage
- **Better solution: Advanced file system techniques**
 - Topic of rest of lecture

Outline

- 1 FFS in more detail
- 2 Crash recoverability
- 3 **Soft updates**
- 4 Journaling

First attempt: Ordered updates

- **Want to avoid crashing after “bad” subset of writes**
- **Must follow 3 rules in ordering updates [Ganger]:**
 1. Never write pointer before initializing the structure it points to
 2. Never reuse a resource before nullifying all pointers to it
 3. Never clear last pointer to live resource before setting new one
- **If you do this, file system will be recoverable**
- **Moreover, can recover quickly**
 - Might leak free disk space, but otherwise correct
 - So start running after reboot, scavenge for space in background
- **How to achieve?**
 - Keep a partial order on buffered blocks

Ordered updates (continued)

- **Example: Create file A**
 - Block X contains an inode
 - Block Y contains a directory block
 - Create file A in inode block X , dir block Y
 - By rule #1, must write X before writing Y
- **We say $Y \rightarrow X$, pronounced “ Y depends on X ”**
 - Means Y cannot be written before X is written
 - X is called the **dependee**, Y the **depender**
- **Can delay both writes, so long as order preserved**
 - Say you create a second file B in blocks X and Y
 - Only have to write each out once for both creates

Problem: Cyclic dependencies

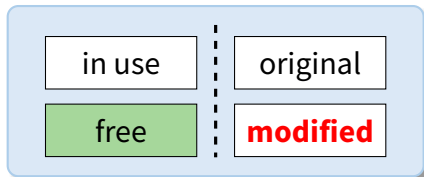
- **Suppose you create file *A*, unlink file *B*, but delay writes**
 - Both files in same directory block & inode block
- **Can't write directory until *A*'s inode initialized (rule #1)**
 - Otherwise, after crash directory will point to bogus inode
 - Worse yet, same inode # might be re-allocated
 - So could end up with file name *A* being an unrelated file
- **Can't write inode block until *B*'s dir. entry cleared (rule #2)**
 - Otherwise, *B* could end up with too small a link count
 - File could be deleted while links to it still exist
- **Otherwise, fsck has to be slow**
 - Check every directory entry and every inode link count

Cyclic dependencies illustrated

inode block directory block

inode #4	$\langle -, \#0 \rangle$
inode #5	$\langle B, \#5 \rangle$
inode #6	$\langle C, \#7 \rangle$
inode #7	

Original organization



inode block directory block

inode #4	$\langle A, \#4 \rangle$
inode #5	$\langle B, \#5 \rangle$
inode #6	$\langle C, \#7 \rangle$
inode #7	

Create file A

inode block directory block

inode #4	$\langle A, \#4 \rangle$
inode #5	$\langle -, \#5 \rangle$
inode #6	$\langle C, \#7 \rangle$
inode #7	

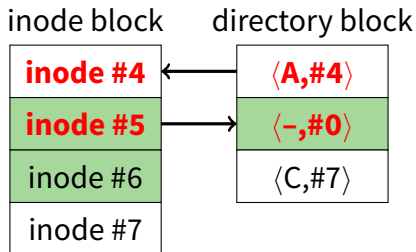
Remove file B

More problems

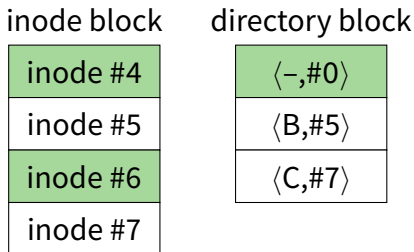
- **Crash might occur between ordered but related writes**
 - E.g., summary information wrong after block freed
- **Block aging**
 - Block that always has dependency will never get written back
- **Solution: *Soft updates* [Ganger]**
 - Write blocks in any order
 - But keep track of dependencies
 - When writing a block, temporarily roll back any changes you can't yet commit to disk
 - I.e., can't write block with any arrows pointing to dependees
...but can temporarily undo whatever change requires the arrow

Breaking dependencies with rollback

Buffer cache



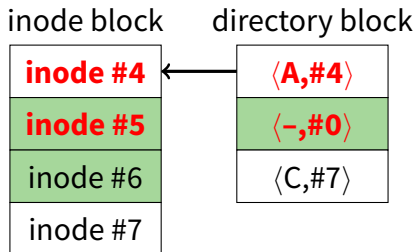
Disk



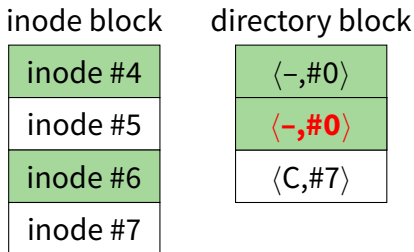
- Deleted Created file A and deleted file B
- Now say we decide to write directory block...
- Can't write file name A to disk—has dependee

Breaking dependencies with rollback

Buffer cache



Disk



- **Undo file A before writing dir block to disk**
 - Even though we just wrote it, directory block still dirty
- **But now inode block has no dependees**
 - Can safely write inode block to disk as-is...

Breaking dependencies with rollback

Buffer cache

inode block

inode #4
inode #5
inode #6
inode #7

directory block

<A,#4>
<-,#0>
<C,#7>

Disk

inode block

inode #4
inode #5
inode #6
inode #7

directory block

<-,#0>
<-,#0>
<C,#7>

- Now inode block clean (same in memory as on disk)
- But have to write directory block a second time...

Breaking dependencies with rollback

Buffer cache

inode block

inode #4
inode #5
inode #6
inode #7

directory block

<A,#4>
<-,#0>
<C,#7>

Disk

inode block

inode #4
inode #5
inode #6
inode #7

directory block

<A,#4>
<-,#0>
<C,#7>

- All data stably on disk
- Crash at any point would have been safe

Soft updates

- **Structure for each updated field or pointer, contains:**
 - old value
 - new value
 - list of updates on which this update depends (*dependees*)
- **Can write blocks in any order**
 - But must temporarily undo updates with pending dependencies
 - Must lock rolled-back version so applications don't see it
 - Choose ordering based on disk arm scheduling
- **Some dependencies better handled by postponing in-memory updates**
 - E.g., when freeing block (e.g., because file truncated), just mark block free in bitmap after block pointer cleared on disk

Simple example

- **Say you create a zero-length file *A***
- **Depender: Directory entry for *A***
 - Can't be written until dependees on disk
- **Dependees:**
 - Inode – must be initialized before dir entry written
 - Bitmap – must mark inode allocated before dir entry written
- **Old value: empty directory entry**
- **New value:** $\langle \text{filename } A, \text{inode } \# \rangle$
- **Can write directory block to disk any time**
 - Must substitute old value until inode & bitmap updated on disk
 - Once dir block on disk contains *A*, file fully created
 - Crash before *A* on disk, worst case might leak the inode

Operations requiring soft updates (1)

1. Block allocation

- Must write the disk block, the free map, & a pointer
- Disk block & free map must be written before pointer
- Use Undo/redo on pointer (& possibly file size)

2. Block deallocation

- Must write the cleared pointer & free map
 - Just update free map after pointer written to disk
 - Or just immediately update free map if pointer not on disk
- **Say you quickly append block to file then truncate**
 - You will know pointer to block not written because of the allocated dependency structure
 - So both operations together require no disk I/O!

Operations requiring soft updates (2)

3. Link addition (see [simple example](#))

- Must write the directory entry, inode, & free map (if new inode)
- Inode and free map must be written before dir entry
- Use undo/redo on i# in dir entry (ignore entries w. i# 0)

4. Link removal

- Must write directory entry, inode & free map (if nlinks==0)
 - Must decrement nlinks only after pointer cleared
 - Clear directory entry immediately
 - Decrement in-memory nlinks once pointer written
 - If directory entry was never written, decrement immediately (again will know by presence of dependency structure)
- **Note: Quick create/delete requires no disk I/O**

Soft update issues

- ***fsync* – syscall to flush file changes to disk**
 - Must also flush directory entries, parent directories, etc.
- ***unmount* – flush all changes to disk on shutdown**
 - Some buffers must be flushed multiple times to get clean
- **Deleting large directory trees frighteningly fast**
 - *unlink* syscall returns even if inode/indir block not cached!
 - Dependencies allocated faster than blocks written
 - Cap # dependencies allocated to avoid exhausting memory
- **Useless write-backs**
 - Syncer flushes dirty buffers to disk every 30 seconds
 - Writing all at once means many dependencies unsatisfied
 - Fix syncer to write blocks one at a time
 - Tweak LRU buffer eviction to know about dependencies

Soft updates fsck

- **Split into foreground and background parts**
- **Foreground must be done before remounting FS**
 - Need to make sure per-cylinder summary info makes sense
 - Recompute free block/inode counts from bitmaps – very fast
 - Will leave FS consistent, but might leak disk space
- **Background does traditional fsck operations**
 - Do after mounting to recuperate free space
 - Can be using the file system while this is happening
 - Must be done in foreground after a media failure
- **Difference from traditional FFS fsck:**
 - May have many, many inodes with non-zero link counts
 - Don't stick them all in lost+found (unless media failure)

Outline

- 1 FFS in more detail
- 2 Crash recoverability
- 3 Soft updates
- 4 **Journaling**

An alternative: Journaling

- **Biggest crash-recovery challenge is inconsistency**
 - Have one logical operation (e.g., create or delete file)
 - Requires multiple separate disk writes
 - If only some of them happen, end up with big problems
- **Most of these problematic writes are to metadata**
- **Idea: Use a *write-ahead* log to *journal* metadata**
 - Reserve a portion of disk for a log
 - Write any metadata operation first to log, then to disk
 - After crash/reboot, re-play the log (efficient)
 - May re-do already committed change, but won't miss anything

Journaling (continued)

- **Group multiple operations into one log entry**
 - E.g., clear directory entry, clear inode, update free map—either all three will happen after recovery, or none
- **Performance advantage:**
 - Log is consecutive portion of disk
 - Multiple operations can be logged at disk b/w
 - Safe to consider updates committed when written to log
- **Example: delete directory tree**
 - Record all freed blocks, changed directory entries in log
 - Return control to user
 - Write out changed directories, bitmaps, etc. in background (sort for good disk arm scheduling)

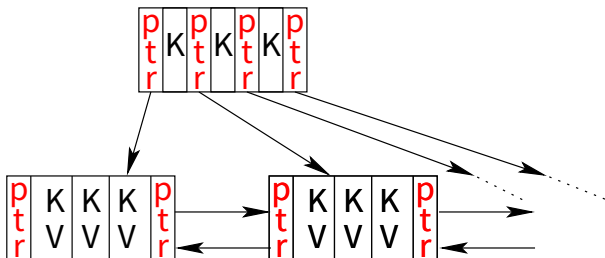
Journaling details

- **Must find oldest relevant log entry**
 - Otherwise, redundant and slow to replay whole log
- **Use checkpoints**
 - Once all records up to log entry N have been processed and affected blocks stably committed to disk...
 - Record N to disk either in reserved checkpoint location, or in checkpoint log record
 - Never need to go back before most recent checkpointed N
- **Must also find end of log**
 - Typically circular buffer; don't play old records out of order
 - Can include begin transaction/end transaction records
 - Also typically have checksum in case some sectors bad

Case study: XFS [Sweeney]

- **Main idea: Think big**
 - Big disks, files, large # of files, 64-bit everything
 - Yet maintain very good performance
- **Break disk up into *Allocation Groups (AGs)***
 - 0.5 – 4 GB regions of disk
 - New directories go in new AGs
 - Within directory, inodes of files go in same AG
 - Unlike cylinder groups, AGs too large to minimize seek times
 - Unlike cylinder groups, no fixed # of inodes per AG
- **Advantages of AGs:**
 - Parallelize allocation of blocks/inodes on multiprocessor (independent locking of different free space structures)
 - Can use 32-bit block pointers within AGs (keeps data structures smaller)

B+-trees



- **XFS makes extensive use of B+-trees**
 - Indexed data structure stores ordered Keys & Values
 - Keys must have an ordering defined on them
 - Stored data in blocks for efficient disk access
- **For B+-tree with n items, all operations $O(\log n)$:**
 - Retrieve closest $\langle \text{key}, \text{value} \rangle$ to target key k
 - Insert a new $\langle \text{key}, \text{value} \rangle$ pair
 - Delete $\langle \text{key}, \text{value} \rangle$ pair

B+-trees continued

- See any algorithms book for details (e.g., [Cormen])
- **Some operations on B-tree are complex:**
 - E.g., insert item into completely full B+-tree
 - May require “splitting” nodes, adding new level to tree
 - Would be bad to crash & leave B+tree in inconsistent state
- **Journal enables atomic complex operations**
 - First write all changes to the log
 - If crash while writing log, incomplete log record will be discarded, and no change made
 - Otherwise, if crash while updating B+-tree, will replay entire log record and write everything

B+-trees in XFS

- **B+-trees are complex to implement**
 - But once you've done it, might as well use everywhere
- **Use B+-trees for directories (keyed on filename hash)**
 - Makes large directories efficient
- **Use B+-trees for inodes**
 - No more FFS-style fixed block pointers
 - Instead, B+-tree maps: file offset \rightarrow \langle start block, # blocks \rangle
 - Ideally file is one or small number of contiguous extents
 - Allows small inodes & no indirect blocks even for huge files
- **Use to find inode based on inumber**
 - High bits of inumber specify AG
 - B+-tree in AG maps: starting $i\# \rightarrow$ \langle block #, free-map \rangle
 - So free inodes tracked right in leaf of B+-tree

More B+-trees in XFS

- **Free extents tracked by *two* B+-trees**
 1. start block # \rightarrow # free blocks
 2. # free blocks \rightarrow start block #
- **Use journal to update both atomically & consistently**
- **#1 allows you to coalesce adjacent free regions**
- **#1 allows you to allocate near some target**
 - E.g., when extending file, put next block near previous one
 - When first writing to file, put data near inode
- **#2 allows you to do best fit allocation**
 - Leave large free extents for large files

Contiguous allocation

- **Ideally want each file contiguous on disk**
 - Sequential file I/O should be as fast as sequential disk I/O
- **But how do you know how large a file will be?**
- **Idea: *delayed allocation***
 - *write* syscall only affects the buffer cache
 - Allow write into buffers before deciding where to place on disk
 - Assign disk space only when buffers are flushed
- **Other advantages:**
 - Short-lived files never need disk space allocated
 - *mmaped* files often written in random order in memory, but will be written to disk mostly contiguously
 - Write clustering: find other nearby stuff to write to disk

Journaling vs. soft updates

- **Both much better than FFS alone**
- **Some limitations of soft updates**
 - Very specific to FFS data structures (E.g., couldn't easily add B-trees like XFS—even directory rename not quite right)
 - Metadata updates may proceed out of order (E.g., create *A*, create *B*, crash—maybe only *B* exists after reboot)
 - Still need slow background fsck to reclaim space
- **Some limitations of journaling**
 - Disk write required for every metadata operation (whereas create-then-delete might require no I/O with soft updates)
 - Possible contention for end of log on multi-processor
 - *fsync* must sync other operations' metadata to log, too