

Midterm evaluations

- **Thank you for doing midterm evaluations!**
 - Clearly lots of room for improvement, and your feedback helps
- **Many said midterm was too long, wanted solutions**
 - I sent out solutions by email this afternoon
 - We will ensure much less time pressure for final
 - Maybe point out ideas of what to study in lecture
- **Lots of feedback about lecture**
 - Too fast, not coordinated enough with Pintos, many topics not in textbook. (E.g., "Lecture only made sense after Pintos" when supposed to be other way)
 - I need to avoid using terms I haven't defined yet
 - Will place optional reference materials on syllabus
- **Complaints about project grading**

1/37

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

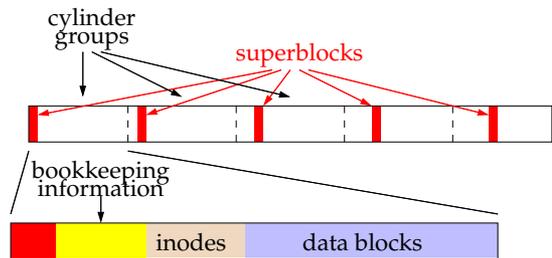
2/37

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

3/37

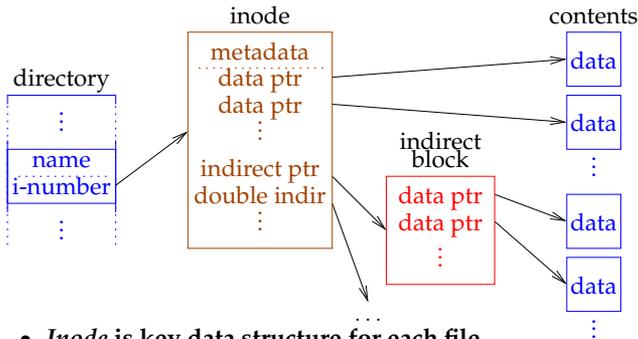
FFS disk layout



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

4/37

Basic FFS data structures



- **Inode is key data structure for each file**
 - Has permissions and access/modification/inode-change times
 - Has *link count* (# directories containing file); file deleted when 0
 - Points to data blocks of file (and indirect blocks)
- **By convention, inode #2 always root directory**

5/37

FFS superblock

- **Superblock contains file system parameters**
 - Disk characteristics, block size, CG info
 - Information necessary to get inode given i-number
- **Replicated once per cylinder group**
 - At shifting offsets, so as to span multiple platters
 - Contains magic number to find replicas if 1st superblock dies
- **Contains non-replicated "summary info"**
 - # blocks, fragments, inodes, directories in FS
 - Flag stating if FS was cleanly unmounted

6/37

Cylinder groups

- Groups related inodes and their data
- Contains a number of inodes (set when FS created)
 - Default one inode per 2K data
- Contains file and directory blocks
- Contains bookkeeping information
 - Block map – bit map of available fragments
 - Summary info within CG – # free inodes, blocks/frags, files, directories
 - # free blocks by rotational position (8 positions)
[Was reasonable in 1980s when disks weren't commonly zoned]

7/37

Inode allocation

- Allocate inodes in same CG as directory if possible
- New directories put in new cylinder groups
 - 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

8/37

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
 - (Partial) solution: new `stat` struct field `st_blksize`
 - Tells applications file system block size
 - `stdio` library can buffer this much data

9/37

Block allocation

- Try to optimize for sequential access
 - If available, use rotationally close block in same cylinder
 - 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

10/37

Directories

- Inodes like files, but 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 (NEW)
 - 8-bit length of file name
- Coalesce when deleting
 - If first `direct` in chunk deleted, set inumber = 0
- Periodically compact directory chunks

11/37

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 64K 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

12/37

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 with 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, ...
 - All directories must be reachable

13/37

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**
 - Of course, this hurts performance
 - E.g., untar much slower than disk b/w
- **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

15/37

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Example: Suppose all data written asynchronously**
- **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

14/37

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 while slowing ordinary usage, recovery slow**
 - If fsck takes one minute, then disks get 10 \times bigger ...
- **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

16/37

First attempt: Ordered updates

- **Must follow three rules in ordering updates:**
 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

17/37

Ordered updates (continued)

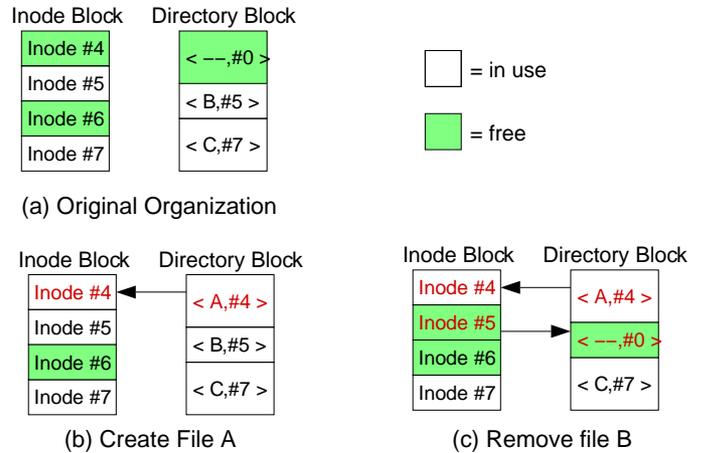
- **Example: Create file A**
 - Block X contains an inode
 - Block Y contains a directory block
 - Create file A in inode block X, dir block Y
- **We say $Y \rightarrow X$, pronounced "Y depends on X"**
 - Means Y cannot be written before X is written
 - X is called the **dependee**, Y the **depender**
- **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

18/37

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**
 - 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 dir entry B cleared**
 - 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 very slow**
 - Check every directory entry and inode link count

Cyclic dependencies illustrated



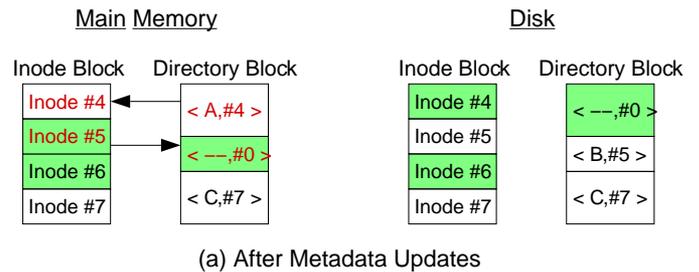
19/37

20/37

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

Breaking dependencies w. rollback

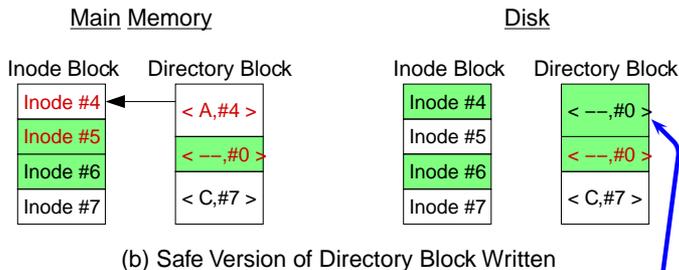


- **Now say we decide to write directory block...**

21/37

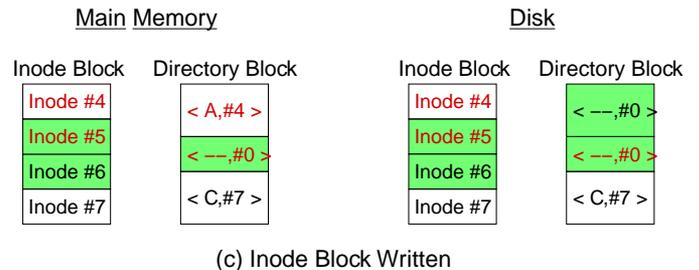
22/37

Breaking dependencies w. rollback



- **Note: Directory block still dirty**
- **But now inode block has no dependencies**
- **Say we write inode block out...**

Breaking dependencies w. rollback

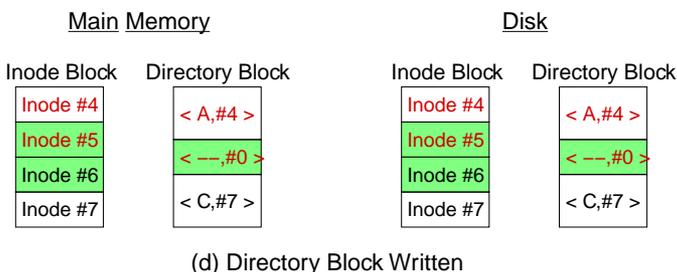


- **Now inode block clean (same in memory as on disk)**
- **But have to write directory block a second time...**

22/37

22/37

Breaking dependencies w. rollback



- All data stably on disk

22/37

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., Just mark block as free in bitmap after pointer cleared

23/37

Simple example

- **Create 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: (filename A, inode #)**

24/37

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 do disk I/O

25/37

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**

26/37

Soft update issues

- **fsync – syscall to flush file changes to disk**
 - Must also flush directory entries, parent directories, etc.
- **umount – 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
 - Fix LRU buffer eviction to know about dependencies

27/37

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
 - Can do in background after mounting to recuperate free space
 - 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)

28/37

An alternative: Journaling

- Reserve a portion of disk for *write-ahead log*
 - Write any metadata operation first to log, then to disk
 - After crash/reboot, re-play the log (efficient)
 - My re-do already committed change, but won't miss anything
- Performance advantage:
 - Log is consecutive portion of disk
 - Multiple log writes very fast (at disk b/w)
 - 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)

29/37

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

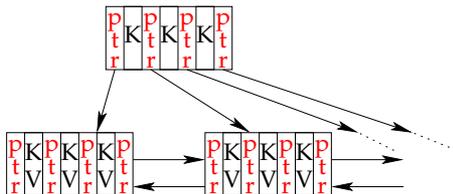
30/37

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)

31/37

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 w. n items, all operations $O(\log n)$:
 - Retrieve closest (key, value) to target key k
 - Insert a new (key, value) pair
 - Delete (key, value) pair

32/37

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

33/37

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 → (start block, # blocks)
 - 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# → (block #, free-map)
 - So free inodes tracked right in leaf of B+-tree

34/37

More B+-trees in XFS

- **Free extents tracked by *two* B+-trees**
 1. start block # → # free blocks
 2. # free blocks → 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, but next block near previous one
 - When first writing to file, but data near inode
- **#2 allows you to do best fit allocation**
 - Leave large free extents for large files

35/37

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

36/37

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 w. soft updates)
 - Possible contention for end of log
 - *fsync* must sync other operations' metadata to log, too

37/37