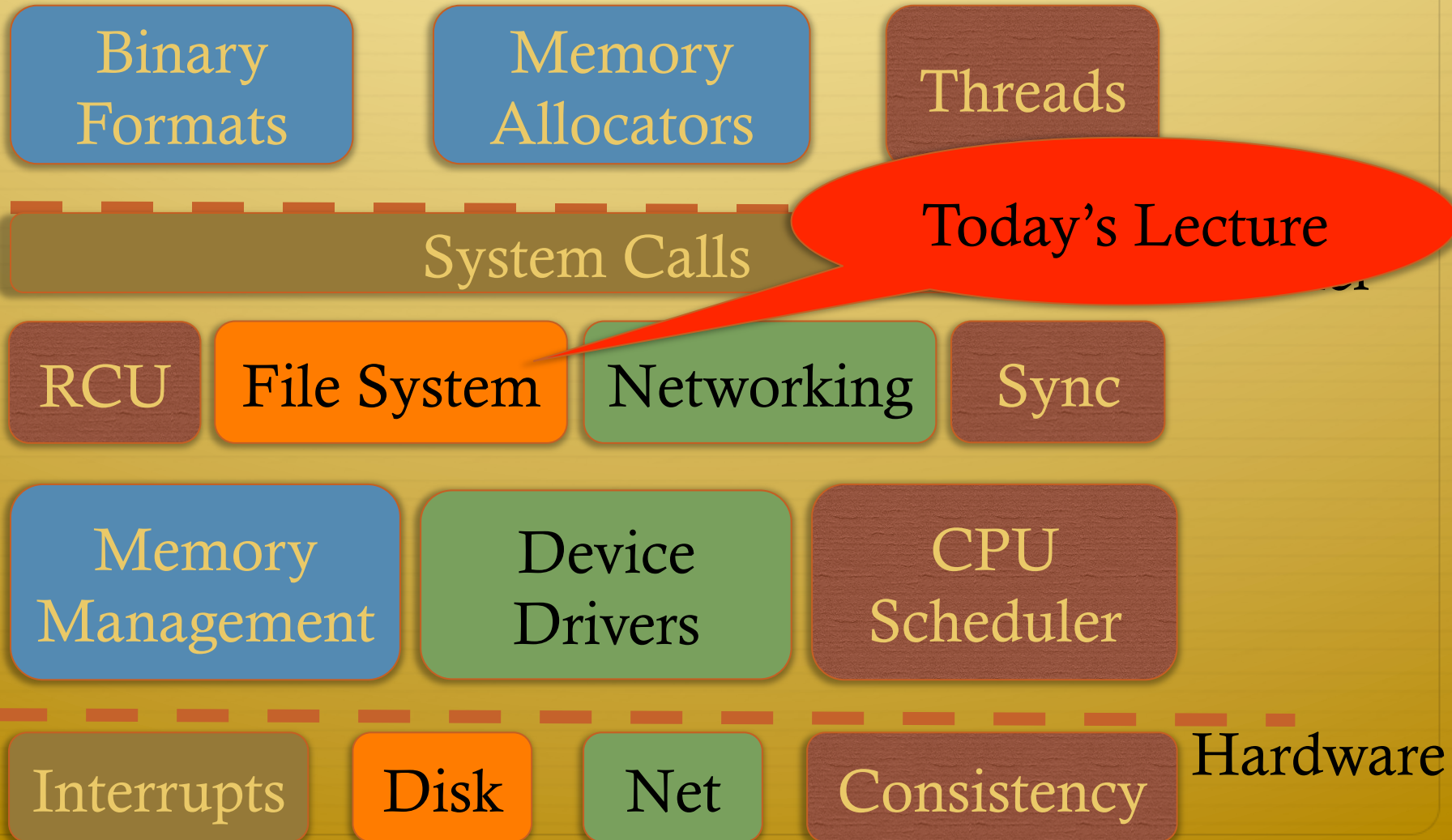




Ext3/4 file systems

Don Porter
CSE 506

Logical Diagram



Ext2 review



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

File systems and crashes



- ✦ What can go wrong?
 - ✦ Write a block pointer in an inode before marking block as allocated in allocation bitmap
 - ✦ Write a second block allocation before clearing the first – block in 2 files after reboot
 - ✦ Allocate an inode without putting it in a directory – “orphaned” after reboot
 - ✦ Etc.

Deeper issue



- ✦ Operations like creation and deletion span multiple on-disk data structures
 - ✦ Requires more than one disk write
- ✦ Think of disk writes as a series of updates
 - ✦ System crash can happen between any two updates
 - ✦ Crash between wrong two updates leaves on-disk data structures inconsistent!

Atomicity



- ✦ The property that something either happens or it doesn't
 - ✦ No partial results
- ✦ This is what you want for disk updates
 - ✦ Either the inode bitmap, inode, and directory are updated when a file is created, or none of them are
- ✦ But disks only give you atomic writes for a sector ☹
- ✦ Fundamentally hard problem to prevent disk corruptions if the system crashes

fsck



- ✦ Idea: When a file system is mounted, mark the on-disk super block as mounted
 - ✦ If the system is cleanly shut down, last disk write clears this bit
- ✦ Reboot: If the file system isn't cleanly unmounted, run fsck
- ✦ Basically, does a linear scan of all bookkeeping and checks for (and fixes) inconsistencies

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 all allocated blocks and inodes are reachable
- ✦ Summary: very expensive, slow scan of the entire file system

Journaling



- ✦ Idea: Keep a log of what you were doing
 - ✦ If the system crashes, just look at data structures that might have been involved
- ✦ Limits the scope of recovery; faster fsck!

Undo vs. redo logging



- ✦ Two main choices for a journaling scheme (same in databases, etc)
- ✦ Undo logging:
 - 1) Write what you are about to do (and how to undo it)
 - ✦ Synchronously
 - 2) Then make changes on disk
 - 3) Then mark the operations as complete
- ✦ If system crashes before commit record, execute undo steps
 - ✦ Undo steps **MUST** be on disk before any other changes! Why?

Redo logging



- ✦ Before an operation (like create)
 - 1) Write everything that is going to be done to the log + a commit record
 - ✦ Sync
 - 2) Do the updates on disk
 - 3) When updates are complete, mark the log entry as obsolete
- ✦ If the system crashes during (2), re-execute all steps in the log during fsck

Which one?



- ✦ Ext3 uses redo logging
 - ✦ Tweedie says for delete
- ✦ Intuition: It is easier to defer taking something apart than to put it back together later
 - ✦ Hard case: I delete something and reuse a block for something else before journal entry commits
- ✦ Performance: This only makes sense if data comfortably fits into memory
 - ✦ Databases use undo logging to avoid loading and writing large data sets twice

Atomicity revisited



- ✦ The disk can only atomically write one sector
- ✦ Disk and I/O scheduler can reorder requests
- ✦ Need atomic journal “commit”

Atomicity strategy



- ✦ Write a journal log entry to disk, with a transaction number (sequence counter)
- ✦ Once that is on disk, write to a global counter that indicates log entry was completely written
 - ✦ This single write is the point at which a journal entry is atomically “committed” or not
 - ✦ Sometimes called a **linearization point**
- ✦ Atomic: either the sequence number is written or not; sequence number will not be written until log entry on disk

Batching



- ✦ This strategy requires a lot of synchronous writes
 - ✦ Synchronous writes are expensive
- ✦ Idea: let's batch multiple little transactions into one bigger one
 - ✦ Assuming no `fsync()`
 - ✦ For up to 5 seconds, or until we fill up a disk block in the journal
 - ✦ Then we only have to wait for one synchronous disk write!

Complications



- ✦ We can't write data to disk until the journal entry is committed to disk
 - ✦ Ok, since we buffer data in memory anyway
 - ✦ But we want to bound how long we have to keep dirty data (5s by default)
 - ✦ JBD adds some flags to buffer heads that transparently handles a lot of the complicated bookkeeping
 - ✦ Pins writes in memory until journal is written
 - ✦ Allows them to go to disk afterward

More complications



- ✦ We also can't write to the in-memory version until we've written a version to disk that is consistent with the journal
- ✦ Example:
 - ✦ I modify an inode and write to the journal
 - ✦ Journal commits, ready to write inode back
 - ✦ I want to make another inode change
 - ✦ Cannot safely change in-memory inode until I have either written it to the file system or created another journal entry

Another example



- ✦ Suppose journal transaction1 modifies a block, then transaction 2 modifies the same block.
- ✦ How to ensure consistency?
 - ✦ Option 1: stall transaction 2 until transaction 1 writes to fs
 - ✦ Option 2 (ext3): COW in the page cache + ordering of writes

Yet more complications



- ✦ Interaction with page reclaiming:
 - ✦ Page cache can pick a dirty page and tell fs to write it back
 - ✦ Fs can't write it until a transaction commits
 - ✦ PFRA chose this page assuming only one write-back; must potentially wait for several
- ✦ Advanced file systems need the ability to free another page, rather than wait until all prerequisites are met

Write ordering



- ✦ Issue, if I make file 1 then file 2, can I have a situation where file 2 is on disk but not file 1?
 - ✦ Yes, theoretically
- ✦ API doesn't guarantee this won't happen (journal transactions are independent)
 - ✦ Implementation happens to give this property by grouping transactions into a large, compound transactions (buffering)

Checkpointing



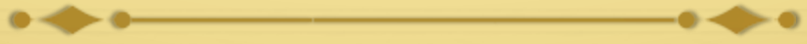
- ✦ We should “garbage collect” our log once in a while
 - ✦ Specifically, once operations are safely on disk, journal transaction is obviated
 - ✦ A very long journal wastes time in fsck
- ✦ Journal hooks associated buffer heads to track when they get written to disk
 - ✦ Advances logical start of the journal, allows reuse of those blocks

Journaling modes



- ✦ Full data + metadata in the journal
 - ✦ Lots of data written twice, batching less effective, 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

Revoke records



- ✦ When replaying the journal, don't redo these operations
 - ✦ Mostly important for metadata-only modes
- ✦ Example: Once a file is deleted and the inode is reused, revoke the creation record in the log
 - ✦ Recreating and re-deleting could lose some data written to the file

ext3 summary



- ✦ A modest change: just tack on a journal
- ✦ Make crash recovery faster, less likely to lose data
- ✦ Surprising number of subtle issues
 - ✦ You should be able to describe them
 - ✦ And key design choices (like redo logging)

ext4



- ✦ ext3 has some limitations that prevent it from handling very large, modern data sets
 - ✦ Can't fix without breaking backwards compatibility
 - ✦ So fork the code
- ✦ General theme: several changes to better handle larger data
 - ✦ Plus a few other goodies

Example



- ✦ Ext3 fs limited to 16 TB max size
 - ✦ 32-bit block numbers ($2^{32} * 4k$ block size), or “address” of blocks on disk
 - ✦ Can't make bigger block numbers on disk without changing on-disk format
 - ✦ Can't fix without breaking backwards compatibility
- ✦ Ext4 – 48 bit block numbers

Indirect blocks vs. extents



- ✦ Instead of represent each block, represent large contiguous chunks of blocks with an extent
- ✦ More efficient for large files (both in space and disk scheduling)
- ✦ Ex: Disk sectors 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

Extents, cont.



- ✦ Worse for highly fragmented or sparse files
 - ✦ If no 2 blocks are contiguous, will have an extent for each block
 - ✦ Basically a more expensive indirect block scheme
 - ✦ Propose a block-mapped extent, which essentially reverts to a more streamlined indirect block

Static inode allocations



- ✦ When you create an ext3 or ext4 file system, you create all possible inodes
- ✦ Disk blocks can either be used for data or inodes, but can't change after creation
- ✦ If you need to create a lot of files, better make lots of inodes
- ✦ Why?

Why?



✦ Simplicity

- ✦ Fixed location inodes means you can take inode number, total number of inodes, and find the right block using math
 - ✦ Dynamic inodes introduces another data structure to track this mapping, which can get corrupted on disk (losing all contained files!)
- ✦ Bookkeeping gets a lot more complicated when blocks change type
- ✦ Downside: potentially wasted space if you guess wrong number of files

Directory scalability



- ✦ An ext3 directory can have a max of 32,000 sub-directories/files
 - ✦ Painfully slow to search – remember, this is just a simple array on disk (linear scan to lookup a file)
- ✦ Replace this in ext4 with an HTree
 - ✦ Hash-based custom BTree
 - ✦ Relatively flat tree to reduce risk of corruptions
 - ✦ Big performance wins on large directories – up to 100x

Other goodies



- ✦ Improvements to help with locality
 - ✦ Preallocation and hints keep blocks that are often accessed together close on the disk
- ✦ Checksumming of disk blocks is a good idea
 - ✦ Especially for journal blocks
- ✦ Fsync on a large fs gets expensive
 - ✦ Put used inodes at front if possible, skip large swaths of unused inodes if possible

Summary



- ✦ ext2 – Great implementation of a “classic” file system
- ✦ ext3 – Add a journal for faster crash recovery and less risk of data loss
- ✦ ext4 – Scale to bigger data sets, plus other features
 - ✦ Total FS size (48-bit block numbers)
 - ✦ File size/overheads (extents)
 - ✦ Directory size (HTree vs. a list)