Journaling File Systems

Operating Systems Baochun Li University of Toronto

Review: File System Implementation where the cyclem inipode the indication $t_{\rm max}$ for every file the σ event they track, but perhaps call the model them different things (such as **rew: File Sys**

The Inode Table (Closeup)

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto
<mark>Reading A File From Diskip From Disk</mark>

File Creation Timeline

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

The Crash Consistency Problem

What happens if power is lost when updating on-disk data structures?

Imagine that you need to update two on-disk data structures, A and B, to complete an operation

One of these will reach the disk first

If the system crashes after one write completes, the on-disk structure will be left in an inconsistent state

Append 4KB to the end of a file
 Append 4KB to the end of a file spread across four blocks, and data blocks (8 total, numbered 0 to 7). And data blocks (8 total, numbered 0 to $\frac{1}{2}$ allocated $\frac{1}{2}$, which is matches in the induced in the inode bitmap, and a set of $\frac{1}{2}$

Open the file, seek to the end, issue a single 4KB write single allocated data block (data block (data block 4), and include a cincle data buritmap.

Append needs three operations to the new block as well as have a bigger size due to the append), the When we would disk of the file system to the file system to the file system to the file system to the file system to

Just the data block is written to the disk

Not a problem, the file system is still consistent

Just the updated inode is written to the disk

- If we trust the inode, we will read garbage data from the data block
- We also have file-system inconsistency, since the on-disk bitmap is saying that the block is not used, but the inode disagrees

Just the updated bitmap is written to the disk

File-system inconsistency — "space leak" in the file system

Crash scenarios: two writes succeeded

The inode and bitmap are written, not the data block

Consistent, but garbage data

The inode and the data block are written, not the bitmap

inode pointing to the correct data, but the bitmap is not consistent

The bitmap and the data block are written, not the inode

Inconsistency between the bitmap and the inode No idea which file the data block belongs to

Objective of crash consistency

Move the file system from one consistent state to another, atomically

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Solution #1: The File System Checker

Idea: let inconsistencies happen, and fix them later when rebooting

In UNIX: fsck

Scans the superblock: sanity checks

Scans the inodes to build a correct version of the data bitmap, and check if it is consistent with the one in the file system — trust the inodes

Check the reference count in each inode, and see if it is consistent with the directory structure — if a file is not in any directory, add to **lost+found**

Major problem: too slow

Idea: write-ahead logging, similar to databases some small amount of space within the partition or on another device.

Famous examples: Linux ext3, ext4, ReiserFS, IBM's JFS, SGI's XFS (ported to Linux), Windows NTFS

> Before overwritting the structures (bitmap, including the structures (bitmap, including the structure overwrite down a little note somewhere else in a well-known location itmap, i **Data Journaling** Let's look at a simple example to understand how **data journaling** works.

Write what you are about to do in a log

Example: Linux ext3 file system), and the file system with a journal looks like the file system with a journal looks like this system with a journal looks like this system. The file system with a journal looks like this: $\mathcal{L}_{\mathcal{A}}$ sometimes it is placed on a separate device, or as a file within \mathcal{A} Example: Linux ox⁺²

What if a crash happens when journaling?

Writes may occur out of order due to the extensive use of **caches in the disk itself** \mathbf{E} sches in the bisk risent small pieces of the big write in any order. Thus, the big write in any order. Thus, thus,

If TxB, I[v2], B[v2], and TxE are written, but not the data block: T_{α} disk internally may disk internally may be the α (2) ikb, i[vz], b[vz], and like are written, but not the data

When replaying the journal, garbage data will be written

Bad for a data block, but if it is the superblock that is written, the file system may not be mountable! action (it has a begin and an end with matching sequence numbers). Furruns recovery, it will replay this transaction, and ignorantly copy the con-

Fixing the problem: Idea #1 To avoid this problem, this problem, this problem, the file system is used to the transaction of the transaction

First write all blocks except the TxE block to the journal: issuing these writes all at once. When these writes complete, the journal I write all blocks except the TXE block to the jou ist write all blocks except the TxE block to the journal:

two steps. First, it was all blocks except the trace of the journal, and it was all blocks except the journal,

When all these writes complete, write the TxE block (using **the "write barrier" mechanism supported by the disk):** block, while the independent in the final interest of σ_i while the σ_i when those with the files complete; there the write the write $\frac{1}{2}$ blow the barrier three manishi supported b

To make sure TxE is written atomically, make it 512 bytes. vided by the disk. It turns out that the disk guarantees that any 512-byte ake sure TXE is written atomically, make it 512 bytes. $\,$

Fixing the problem: Idea #2

When writing a transaction to a journal, include a checksum in the TxB and TxE blocks

- If there is a mismatch between the stored checksum and the computed one, a crash has occurred
- ACM SOSP 2005 paper, eventually used in Linux ext4

Recovery

If a crash happens before the transaction is logged, do nothing and skip the pending update

For those transactions that committed (TxE block written) successfully

Redo the log by replaying all committed transactions

To improve performance, buffer all the updates in the memory cache as a global transaction, and avoid excessive writes to the disk

Some time after the on-disk structures are updated (called "checkpoint"), mark the transaction free in the journal by updating a journaling superblock CRASH CONSISTENCY: FSCK AND JOURNALING 13 oldest and newest non-checkpointed transactions in the log in a **journal**

But we are still writing each data block to the **disk twice!**

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Writing each data block to the disk twice is a heavy cost to pay for rare crashes!
 pay for rare crashes! ten to the journal. Thus, when performing the same update as above, the

Idea: the user data is not written to the journal at all

14 CRASH CONSISTENCY: FSCK AND JOURNALING

If we wish to make sure that the inode will not point to garbage data blocks, simply write data blocks first before **writing the metadata to the journal** I trajfing the metodete to the issue substantially reduces I writhig the inclaudia to the journal

Both Windows NTFS and SGI's XFS (ported to Linux) use metadata journaling — my two favourites! Both Windows NTFS and SGI's XFS (ported to Linux) us Let's a metal consider the constant of a file the constant of the constant of the constant of the standard standard standard the standard standa pourinain between blocks. The update consists of the update consists of the update consists of the update consis

Metadata journaling: a timeline CRASH CONSISTENCY: FOR AN ORDER IN A GENERAL EXPORT OF THE CONSTRUCTION OF THE CONSTRUCTION OF THE CONSTRUCTIO

Tricky case: block reuse with deletion then really controlled the property of the state of t The particular example Tweedie gives is as follows. Suppose you are

The user adds an entry to a directory, foo, by creating a file

The content of this directory (say, data block 1000) will be written to the log (metadata journaling) adds an entry to food (say by creating a file), and thus the contents of \mathcal{A} f the content of this directory (say, data block TOOO) will be

Journal TxBank and TxB

Tricky case: block reuse with deletion for directories are considered metadata directories are considered metadata (because metadata) are written to assume that the food of the food of the food directory data is block in the log thus the log thus the log thus

The user then deletes everything in the directory as well as he data block 1000 for reuse Then the user creates a new file, foobar, which reuses block 1000 th ptr:1000 [final addr:1000] id=1 At this point, the user deletes everything in the directory as well as the

foobar's inode and data are committed to the disk, but only its inode is committed to the journal $u \in \mathbb{R}^n$ used to belong to food. The inode of food of food. The inode of food \mathbb{R}^n

During recovery, the replay overwrites foobar's data with the old directory! in block 1000 in the file foobar is *not* journaled.

Idea #1: Never reuse blocks until the deletion of these blocks is checkpointed out of the journal

Idea #2: (Linux ext3) add a new type of record to the journal, known as a revoke record

Deleting a directory will cause a revoke record to be written to the journal

When replaying the journal, the system first scans for such revoke records — any such revoked data is never replayed

Used by Sun ZFS (Jeff Bonwick, who also designed slab allocation) and Apple APFS (used since MacOS 10.11 High Sierra, iOS 11)

Idea: never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk

After a number of updates are completed, copy-on-write file systems flip the root structure of the file system to include pointers to the newly updated structures

What we've covered so far

Three Easy Pieces: Chapter 42 (Crash Consistency)