Introduction

As we've seen so far, the file system handles a collection of data structures to implement the required abstractions: files, directories, and all associated metadata. Unlike most data structures (such those in a running program's memory), file system data structures must persist, i.e., be stored on media that keep data even when power is lost (such as hard disks or flash-based SSDs).

A file system must update persistent data structures even if the power goes off or the system crashes. What happens if someone trips over the power cord and the machine loses power in the middle of updating on-disk structures? Or the OS fails due to a bug? Updates to persistent data structures can be hard due to power loss and crashes, causing a novel and intriguing problem in file system implementation known as crash-consistency.

This issue is easy to grasp. Assume you need to update two on-disk structures, A and B, to accomplish a task. Because the disk can only service one request at a time, one of these will arrive first (either A or B). The on-disk structure will be inconsistent if the system crashes or loses power after one write. So we have a challenge that all file systems face:

HOW TO UPDATE THE DISK DESPITE CRASHES

As a result, the on-disk state may only be partially updated. After a crash, the system wants to remount the file system (in order to access files and such). Given that crashes can occur at any time, how can we ensure the file system maintains a reasonable on-disk image?

This chapter will describe the topic in greater detail and discuss solutions utilized by file systems. We'll start by looking at the file system checker (fsck) used by older file systems. This will be followed by journaling (also known as write-ahead logging), which adds overhead to each write but recovers faster from crashes or power outages. We'll cover the basics of journaling, as well as some of the more recent journaling file systems like Linux ext3.

A Detailed Example

Let’s start with an example of journaling. We’ll require a workload that modifies on-disk structures. So let’s say the job is simple: appending a single data block to a file. To append a file, first open it, then use lseek() to shift the file offset to the end of the file, and then close it.
Assume we are utilizing simple file system structures on the disk, as we have seen before. This modest example has an inode bitmap (8 bits per inode), a data bitmap (8 bits per data block), inodes (0 to 7) and data blocks (8 total, numbered 0 to 7). This is the file system diagram:

The graphic shows a single inode (inode 2), which is marked in the inode bitmap, and a single data block (data block 4), which is likewise marked in the data bitmap. Because this is the initial version, it is labeled I[v1] (due to the workload described above).
Let’s look into this inode too. We see in I[v1]:

owner : remzi
 permissions : read-write
size
pointer
pointer
pointer
pointer
: 1 :4
: null : null : null

The file size is 1 (it has one block allocated), the first direct pointer points to block 4 (the file’s initial data block, Da), and the other three direct pointers are set to null (indicating that they are not used). Real inodes include many more fields; see earlier chapters.
The inode (which must link to the new block and record the new larger size owing to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to signal that the new data block has been allocated.
So we have three blocks in the system memory to write to disk. The new inode (I[v2]) looks like this:

owner : remzi
 permissions : read-write
 size
pointer
pointer
pointer
pointer
: 2

: 4
: 5
: null : null

The new data bitmap (B[v2]) is 00001100. Finally, the data block (Db) contains anything users put into files. Stolen music?
The final on-disk image of the file system should look like this:

The file system must make three sepa-rate writes to the disk, one for each inode (I[v2]), bitmap (B[v2]), and data block (D[v2]) (Db). The dirty inode, bitmap, and new data will first sit in main memory (in the page cache or buffer cache) for a while before the file system decides to write them to disk (after 5 or 30 seconds). Sadly, a crash can prevent these upgrades from taking place. If a crash occurs after one or two of these writes but not all three, the file system may be left in an odd condition.

Crash Scenarios

Let's look at various crash scenarios to better grasp the issue. Assume only one write succeeds; there are three possible outcomes listed below:

Disk writes only the data block (Db). But there's no inode pointing to it, and no bitmap saying the block is allocated. It's as if the write never happened. From the perspective of file-system crash consistency, this instance is fine.

Only the modified inode (I[v2]) is saved. In this situation, the inode points to disk address (5), but Db has not yet been written there. With that pointer, we'll read junk from the disk (the old contents of disk address 5).

We also have a new issue called file-system inconsistency. The bitmap on disk says data block 5 hasn't been allocated, but the inode says it has. To use the file system, we must address the conflict between the bitmap and the inode (more on that below).

Only the updated bitmap (B[v2]) is saved. The bitmap says block 5 is allocated, but no inode points to it. As a result, the file system is inconsistent again, and block 5 is never used by the file system.

This attempt to write three blocks to disk has three more crash situations. For example, two writes succeed while one fails:

Not all data (I[v2]) is written to disk (Db). The inode points to block 5, the bitmap indicates that 5 is in use, and therefore everything seems good from the file system metadata standpoint. But there's a catch: 5 is full of junk.

• Only the inode (I[v2]) and data block (Db) are written (B[v2]). In this scenario, the inode points to the correct data on disk, but there is a discrepancy between the inode and the old bitmap (B1). So, before accessing the file system, we must again solve the issue.

• Only the bitmap (B[v2]) and data block (Db) are written (I[v2]). In this scenario, the inode and the data bitmap are inconsistent. We don't know which file it belongs to because no inode points to it.

Crash Consistency Issues

This should show the multiple issues that crashes can cause our on-disk file system image: inconsistency in file system data structures, space leaks, and returning garbage data to a user. We'd like to atomically change the file system from one state (before the file was appended to) to another (e.g., after the inode, bitmap, and new data block have been written to disk). It's difficult since the disk only commits one writing at a time, and crashes or power outages may occur between updates. This is the crash-consistency problem (we could also call it the consistent-update problem).

Solution #1

Early file systems used a primitive crash consistency method. They opted to let inconsistencies occur and then correct them later (when rebooting). fsck2 is a typical example of this lazy method. fsck is a UNIX tool for discovering and fixing such inconsistencies. Consider the instance where the file system appears to be consistent but the inode points to junk data. The only purpose is to ensure internal consistency of file system metadata.

fsck has several phases. It is run before the file system is mounted and made available to users (fsck assumes no other file-system activity is occurring).

fsck does the following:

- Superblock: fsck initially verifies the superblock, mostly by comparing the file system size to the number of blocks allocated. In this instance, the system (or administrator) may choose to use an alternate copy of the superblock.
- Free blocks: Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of what blocks are currently allocated within the file system. It uses this knowledge to create correct allocation bitmaps, so any mismatch between bitmaps and inodes is addressed by trusting the inode information. Also, all inodes are checked to see if they are indicated as used in the inode bitmaps.
- Inode state: Every inode is checked for corruption or other problems. For example, fsck verifies each allocated inode’s type field (e.g., regular file, directory, symbolic link, etc.). If the inode field problems are not easily fixed, the inode is emptied by fsck and the inode bitmap is updated.
- Inode links: fsck verifies the number of links in each allocated inode. In case you’ve forgotten, the link count is the number of folders that have a reference (or link) to this specific file. fsck produces its own link counts for every file and directory in the file system to validate the link count. If the newly calculated count does not match the inode count, the inode count must be fixed. As soon as an allocated inode is discovered, it is transferred to the lost+found directory.
- Duplicates: fsck checks for duplicate pointers, if two inodes refer to the same block. An obvious faulty inode may be deleted. The pointed-to block could also be cloned, providing each inode its own copy.

- Bad blocks: While scanning the list of all pointers, faulty block pointers are checked for. In this case, the pointer is regarded “bad” if the address it points to is larger than the parti- tion size. In this scenario, fsck just removes (clears) the pointer from the inode or indirect block.
- Directory checks: fsck doesn’t understand what’s inside user files, although directories include information created by the file system. So, fsck verifies that the first entries are “.” and “..”, that each inode referenced to in a directory entry is allocated, and that no directory is linked to more than once in the hierarchy.

As you can see, writing a working fsck involves extensive understanding of the file system; testing it in all circumstances can be difficult. But fsck (and similar methods) have a deeper, more fundamental issue: they are excessively slow. Scanning a huge disk volume to find all allocated blocks and read the directory tree may take minutes or hours. Fsck performance became prohibitive as disk size and RAID adoption expanded (despite recent enhancements).

Fsck performance became prohibitive as disk size and RAID adoption expanded (despite recent enhancements).

The core concept of fsck looks a bit irrational. In our previous example, only three blocks are written to the drive; scanning the entire disk to address problems that happened during that update is prohibitively expensive. This is like putting your keys on the bedroom floor and then searching the entire house for them, starting in the basement and working your way up. It may work, but it’s inefficient. Other solutions became necessary as disks (and RAIDs) increased.

Solution #2

The most prevalent answer to the problem of continuous updates comes from the area of database management systems. Write-ahead logging was created to handle exactly this type of issue. For historical reasons, we term write-ahead logging journaling. Many current file systems, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS, adopt this principle.

The main premise is this. Before updating the disk, make a note (somewhere else on the disk, in a known area) stating what you're about to do. Also known as write-ahead logging, the act of writing this note is the "write ahead" aspect.
This ensures that if a crash occurs during an update (overwrite) of a structure, you may go back and look at the note you made and try again, rather than having to scan the entire disk. Journaling adds work during updates to lessen effort required during recovery.

Now we'll look at how ext3, a prominent journaling file system, incorporates journaling. The disk is separated into block groups, and each block group has an inode bitmap, data bitmap, inodes, and data blocks. The new key structure is the journal, which takes up little space in the partition or elsewhere. Ext2 file system (without journaling):

A journaled ext3 file system looks like this:

The journal's presence and use are the main differences.

Data Journaling

Let’s look at an example of data journaling. Data journaling is a feature of the Linux ext3 file system, on which this article is based.

Let’s say we want to rewrite the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk. We will now write them to the log before writing them to the final disk locations (a.k.a. journal). This is how it will appear in the log:

We used five blocks here. TxB contains information on the pending file system update (e.g., final addresses of blocks I[v2], B[v2], and Db) and a transaction identifier (TID). Putting the exact physical contents of the update in the journal is known as physical logging (an alternative idea, logical logging, puts a more compact logical representation of the update in the journal, e.g., “this update wishes to append data block Db to file X”, which is a little more complex but may save space in the log and possibly improve performance). The last block (TxE) contains the TID and marks the end of the transaction.

Once this transaction is safely stored on disk, we can checkpoint the file system and overwrite the previous structures. To checkpoint the file system (i.e., bring it up to date with the journal update), we write I[v2], B[v2], and Db to their respective disk locations as shown above. So, our starting operation sequence:

1. Journal write: Write the transaction to the log, including a transaction-begin block, all pending data and metadata modifications, and a transaction-end block.
2. Checkpoint: Update the pending information and data in the file system.

In our case, we’d start with TxB, I[v2], B[v2], Db, and TxE. After these writes, we would checkpoint I[v2], B[v2], and Db to their ultimate destinations on disk.

Problems arise when a crash happens while writing to the journal. We’re trying to write TxB, I[v2], B[v2], Db, TxE to disk. One simple method is to issue one at a time, waiting for each to finish before issuing the next. But this is sluggish. In an ideal world, we’d issue all five block writes at once, turning five writes into a single sequential write. However, this is risky since the disk may do internal scheduling and complete little chunks of the main write in any order. So the disk can first write TxB, I[v2], B[v2], and TxE, then write Db. If the disk loses power between (1) and (2), this is what happens:

What’s the issue? The transaction appears to be genuine (it has a begin and an end with matching sequence numbers). Because it is arbitrary user data, the file system cannot see that fourth block as being incorrect. By rebooting and running recovery, the system will replay this transaction and copy the garbage block ‘??’ to where Db should be. Unmountable file systems can be caused by corrupted or missing superblocks, which can corrupt or delete arbitrary user data in a file.

To avoid this, the file system does a two-step transactional write. First, it writes all blocks except TxE to the journal at once. After this, the journal will look like this (assuming our append workload continues):
After that, the file system writes the TxE block, leaving the journal in this safe state:
The disk’s atomicity guarantee is a key part of this procedure. The disk ensures that any 512-byte

To ensure that the writing of TxE is atomic, it should be a single 512-byte block. So, our present file system update mechanism comprises three phases:

1. Journal write: Write the transaction’s contents (TxB, metadata, and data) to the log.
2. Journal commit: Write the transaction commit block (including TxE) to the log.
3. Checkpoint: Write the update’s contents (metadata and data) to disk.

Recovery

Let’s see how a file system can use the journal to recover from a crash. During this update sequence, a crash is possible. If the crash occurs before the transaction is successfully logged (i.e., before Step 2), the pending update is simply skipped. If the file system crashes after the transaction commits but before the checkpoint, the update can be recovered as follows. In order to recover the file system, the system must first scan the log for committed transactions. These transactions are then replayed (in order), with the file system attempting to write out the blocks to their final on-disk locations. This type of logging is termed redo logging and is one of the simplest. In this way, the file system validates that the on-disk structures are consistent and can proceed to mount the file system and prepare for future requests.

It is acceptable for a crash to occur during check-pointing, even after certain modifications to block final positions have been performed. Or, in the worst situation, they are re-formed after healing. Because recovery is an uncommon activity (only performed after a system crash), a few redundant writes are harmless.

Batching Log Updates

Notably, the basic protocol can add a lot of unnecessary disk traffic. Assume we create two files, file1 and file2, in the same directory. An updated inode bitmap (for allocating new inodes) is required to create a file, as well as the file’s newly generated inode.

the new directory en- attempt data block and the parent directory inode (which now has a new modification time). Because the files are in the same directory, and assuming they have inodes within the same inode block, we may end up writing the same blocks over and over again if we weren’t cautious.

Some file systems (e.g., Linux ext3) do not commit updates one by one to disk, but rather buffer all updates into a global transaction. When the two files are created, the file system simply labels the in-memory inode bitmap, file inodes, directory data, and directory inode as dirty and adds them to the current transaction’s list of blocks. This single global transaction comprising all of the above updates is committed when the timeout of 5 seconds has passed. By buffering updates, a file system can avoid excessive disk write traffic.

The Finite Log

So we have a mechanism for changing file-system structures on disk. When it's time to write to disk, the file system first carefully writes the transaction information to the journal (a.k.a. write-ahead log), then checkpoints those blocks to their final places on disk.
But the log has a limit. It will soon be full if we keep adding transactions (as shown). So, what happens next?

INSERT IMAGE

When the log is filled, two issues arise. The first is less critical: the larger the log, the longer it takes to recover because the process must replay all transactions within it. This causes the file system to be "less than useful" since when the log is full (or almost full), no additional transactions may be committed to disk (i.e., useless).
To solve these issues, journaling file systems reuse the log as a circular data structure, hence the name "circular log". The file system must operate after a checkpoint to do so. That is the file system should release the space occupied by a checkpointed transaction, allowing the log space to be reused. For example, you could simply store the oldest and newest non-checkpointed transactions in a journal superblock, leaving the rest free.

INSERT IMAGE

The journal superblock (not to be confused with the main file system superblock) stores enough information to identify which transactions have not yet been checkpointed, allowing for faster recovery and cyclic log usage. To our simple protocol we add another step:

1. Journalwrite: Write the contents of the transaction (TxB and the update) to the log.
2. Journal commit: Write the transaction commit block (TxE) to the log; wait for it to finish.
3. Checkpoint: Update the file system content to their final destinations.
4. Free: Update the journal superblock to label the transaction free.

Our data journaling protocol is complete. In any case, we are duplicating each data block on the disk, which is a costly endeavor, especially when a system crash occurs. Can you keep consistency without writing data twice?

Metadata Journaling

While recovery is now faster (reading the journal and replaying a few transactions rather than the entire disk), typical file system operation is still slow. Writing to the journal first now doubles write traffic, especially under sequential write workloads, which now proceed at half the drive's max write capacity. Also, between journal and main file system writes, there is a costly seek, which adds overhead for particular workloads.

Because writing every data block to disk twice is expensive, people have tried many methods to improve performance. For example, the above journaling mode is often referred to as data journaling (as in Linux ext3) (in addition to the metadata of the file system). Ordered journaling (or just metadata journaling) is a similar concept, except that user data is not written to the journal. So, upon executing the same update, the journal would have the following information:

Because most disk I/O traffic is data, not writing data twice minimizes the I/O load of journaling. When should we write data blocks to disk?
Consider our file add example again to better grasp the issue. The update has three blocks: I, B, and Db. The first two are metadata and will be logged and checked; the third will be written to the file system just once. When should we save Db? So what?
In fact, for metadata-only journaling, the order of the data writes matters. What if we write Db to disk after the trans-action (I[v2] and B[v2])? The file system is consistent, although I[v2] may point to junk data. Consider the instance where I[v2] and B[v2] are written but Db is not. A recovery attempt is made. Because Db isn't in the log, the file system will replay writes to I[v2] and B[v2] (from the perspective of file-system metadata). However, I[v2] will point to junk data, i.e., whatever was in Db's slot.
To avoid this, certain file systems (like Linux ext3) write data blocks (of ordinary files) first, before metadata. The protocol is as follows:

1. Data write: At final location, write data; wait for completion (waiting is optional; details below).
2. Journal metadata write: Write the begin block and metadata to the log; wait for writes to complete.
3. Journal commit: Write the transaction commit block (including TxE) to the log; wait for the write to finish.
4. Checkpoint metadata: Write the metadata update to its final file system location.
5. Free: Later, mark the transaction free in journal superblock.

A file system can prevent garbage pointers by requiring data write first. Crash consistency relies on the rule of “write the pointed-to object before the object that points to it” (GP94) (see below for details).
Metadata journaling (ext3) is more common than full data journaling. For example, both NTFS and SGI’s XFS use metadata journaling. ext3 supports data, ordered, and unordered modes (in unordered mode, data can be written at any time). These techniques all maintain metadata consistent, but their data semantics vary.
Finally, forcing the data write (Step 1) before writing to the journal (Step 2) is not essential for correctness. Writes to data, the transaction-begin block, and journaled metadata can be issued concurrently, as long as Steps 1 and 2 are completed before issuing the journal commit block (Step 3).

Solution #3

ing more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled backpointer-based consistency (or BBC), no ordering is enforced between writes. To achieve consistency, an additional back pointer is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained.

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled optimistic crash consistency, this new approach issues as many writes to disk as possible by using a generalized form of the transaction checksum, and includes a few other techniques to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required.

Summary

We introduced the issue of crash consistency and discussed possible solutions. On newer computers, the earlier way of generating a file system checker may be too slow. So journaling is now widely used. In short, journaling reduces recovery time from O(disk volume size) to O(log volume size), greatly speeding up post-crash recovery. As a result, many current file systems journal. Journaling can take several forms; the most common is ordered metadata journaling, which minimizes traffic to the journal while maintaining sufficient consistency guarantees for both file system information and user data (see Figure 1). Finally, solid user data guarantees are undoubtedly one of the most critical things to provide; however, recent research shows that this field is still evolving.

For data and metadata journaling, see Figure 42.1; for only metadata journaling, see Figure 42.2.
Each row in the picture represents the logical time that a write can be issued or completed. This means that if you write to the transaction begin block (TxB) first, you can write to the transaction end block (TxE) thereafter. A similar constraint holds for checkpointing writes to data and metadata blocks. Horizontal dashed lines indicate write-ordering constraints.

INSERT IMAGE

The metadata journaling system has a similar timeline. The data write can logically be issued at the same time as the transaction begin and journal write, but it must be issued and completed before the transaction finish. Finally, the timestamps for each write in the timelines are arbitrary. In reality, the I/O subsystem determines completion time and may reorganize writes to improve performance. The only guarantees we have regarding ordering are those for protocol consistency (and are shown via the horizontal dashed lines in the figures).