

Case Study: Linux ext3 and Journaling

In this note, we will study the Linux **ext3** file system. The ext3 FS adds **journaling** (also known as **write-ahead logging**) to the file system in order to recover more quickly from operating system crashes or power losses. We will discuss why this is important and understand the basic machinery of journaling, including a few different flavors that ext3 implements.

32.1 An Example

To start things off, though, let's look at an example. Let's say we are trying to append a block to an existing file. For simplicity, let's assume we are using a simplified version Linux **ext2** [T98], which is an intellectual descendent of the FFS file system [MJLF84].

Before we do this write, the file is on disk in the form of an inode, one (or more) existing data blocks, and some bitmaps that mark the inode and data blocks as in-use. This might look something like this (on a tiny file system):

```
i-node | data | inodes | data blocks
bitmap | bitmap | |
010000 | 000010 | -- Iv1 -- -- -- | -- -- -- -- D1 --
```

Inside the first version of the inode (Iv1), it looks like this:

```

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

```

In this simplified inode, the 'size' of the file is '1' (it has one block allocated), the first direct pointer points to block 4 (the first data block of the file, D1), and all three other direct pointers are set to null (indicating that they are not used). Of course, real inodes have many more fields and pointers and so forth..

Inside the data bitmap (B1), we have a bit indicating the data block 4 is in use. And finally, of course, we see that disk block 4 holds the contents of the first block of the file (D1).

When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which now must contain a pointer to the new block as well as an updated size count to reflect the new size of the file), the new data block D2, and a new version of the data bitmap to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (Iv2) now looks like this:

```

owner      : remzi
permissions : read-only
size       : 2
pointer    : 4
pointer    : 5
pointer    : null
pointer    : null

```

The updated data bitmap (B2) now looks like this:

```
0 0 0 0 1 1
```

and then there is data block itself (D2) which is just filled with whatever it is users put into files.

What we would like is for the final on-disk image of the file system to look like this:

```
i-node | data | inodes | data blocks
bitmap | bitmap | |
010000 | 000011 | -- Iv2 -- -- -- | -- -- -- -- D1 D2
```

and thus we must perform three separate writes to the disk, one each for the inode (Iv2), bitmap (B2), and data block (D2).

THE CRUX OF THE PROBLEM

The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do ensure the file system stay ensure the on-disk image is in a reasonable state?

32.2 Writing To Disk: Examples

To understand this better, let's look at some example crash scenarios. Imagine only a single write succeeds. There are three possible cases:

- **Just the data block (D2) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred.
- **Just the updated inode (Iv2) is written to disk.** In this case, the inode points to the disk address (5) where D2 was about to be written, but D2 has not yet been written there. Thus, if we trust that pointer, we will read **garbage data** from the disk (the old contents of disk address 5).

Further, we have a new problem, which we call a **file system inconsistency**. The on-disk bitmap is telling us that

data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency, and to use the file system, we must somehow fix this (more on that below).

- **Just the updated bitmap (B2) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again, but there is little to do as we have no idea which file this block should have been a part of.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (Iv2) and bitmap (B2) are written to disk, but not data (D2).** In this case, the file system is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
- **The inode (Iv2) and the data block (D2) are written, but not the bitmap (B2).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we should fix that, either by removing that block from the inode (and thus losing data) or by simply trusting the inode and updating the bitmap to indicate that the data block 5 is in use.
- **The bitmap (B2) and data block (D2) are written, but not the inode (Iv2).** In this case, we again have an inconsistency between the inode and the data bitmap, but there is only one way to fix it: by clearing the bitmap (because no inode points to that data block, and thus we have no idea which file D2 is a part of). Thus, even though the data block was written to disk, it is lost.

32.3 The Consistent Update Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another (e.g., after the inode, bitmap, and new data block have been written to disk), but we can't do this easily because the disk only commits one write at a time, and crashes may occur between these updates. We call this general problem that arises for all file systems the **consistent update problem**.

32.4 Some Solutions

Early file systems took a simple approach to the file system update problem. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of a tool that does this is **fsck**, a Unix tool for find such inconsistencies and repairing them [M86]. Note, such an approach can't fix all problems (e.g., the example above where the file system looks consistent but the inode points to garbage data); the only goal is to make sure the file system metadata is internally consistent.

(could have more fsck details here)

However, fsck (and similar approaches) have a fundamental problem: they are *too slow*. With a very large disk volume, just reading the entire disk may take many minutes or even hours. Worse, it seems kind of crazy. Consider our example above with just a few blocks being written to the disk; what a waste to scan the entire disk just to see if one of those three writes didn't finish! It's kind of like dropping your keys on the floor in your bedroom, and then commencing "a search the entire house for the keys!" recovery algorithm, starting in the basement and working your way through every room. It may work, but it sure seems odd. Thus, as disks (and multi-disk RAID systems) grew in size, people started to look for other solutions.

32.5 Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we sometimes call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87].

The basic idea is as follows. When updating the disk, before over-writing the structures in place, first write down a little note (somewhere else on the disk) describing what you are about to do. Writing this little note is the “write ahead” part, and we write it to a structure we call the “log”; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk.

We’ll now describe how Linux ext3 incorporates journaling into the file system. Most of the on-disk structures are identical to ext2, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:

```
Superblock | Group0 | Group1 | ... | GroupN
```

A Linux ext3 file system with a journal thus looks like this:

```
Superblock | Journal | Group0 | Group1 | ... | GroupN
```

The real difference is just the presence of the journal, and of course, how it is used.

32.6 Data Journaling: An Example

Let's do a simple example to understand how Linux ext3 **data journaling** mode works. Say we have our canonical update again, where we wish to write the inode (Iv2), bitmap (B2), and data block (D2) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:

```
TxBegin | Iv2 | B2 | D2 | TxEnd
```

You can see we have written five blocks here. The transaction begin tells us about this update, including information about the pending update to the file system and some kind of transaction identifier (TID). The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block D2 to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system. Thus, we issue the writes Iv2, B2, and D2 to their disk locations as seen above. If these writes complete successfully, we are basically done (though at some point we should free those log entries so that we can use them again later). Thus, the complete sequence is as follows:

1. Write the transaction (containing Iv2, B2, D2) to the log
2. Write the blocks (Iv2, B2, D2) to the file system proper
3. Mark the transaction free in the journal

Of course, a crash may happen at any time during this sequence. If it happens after the transaction is committed to disk but before (or during) the writes of Iv2, B2, and D2, we can now **recover**. When the system boots, it will scan the log and look

for transactions that have committed to the disk but have not yet been freed; these transactions are thus **replayed**, with the file system again attempting to write Iv2, B2, and D2 to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. Again, if these writes complete, the recovery code will make this transaction as free and thus the file system will be in a consistent state.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (TxBegin | Iv2 | B2 | D2 | TxEnd) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxBegin, Iv2, B2, and TxEnd and only later (2) write D2. Unfortunately, if the disk loses power between (1) and (2), this is what we will see on disk:

```
TxBegin | Iv2 | B2 | ??? | TxEnd
```

Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end, after all, and some stuff in between). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '???' to the location where D2 is supposed to live.

To avoid this, some journaling systems issue a transactional write in two steps. First, they write all blocks except the TxEnd block to the journal:

```
TxBegin | Iv2 | B2 | D2 |
```

Then, only when those writes complete, do they issue the write of the TxEnd block, thus leaving the journal in this final, safe state:

```
TxBegin | Iv2 | B2 | D2 | TxEnd
```

What we really need to understand here is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte write will either happen or not (and never be half-written); thus, to make sure the write of TxEnd is atomic, one should make it a single 512-byte block.

32.7 Simple Data Journaling: Costs

Unfortunately, there are costs to journaling. Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation is slower. In particular, for each write to disk, we are not also writing to the journal first, thus doubling write traffic. Further, between writes to the journal and writes to the main file system, there is a costly seek.

Because of these costs, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata journaling**), and it is nearly the same, except that user data is not written to the journal. Thus, when performing the same update as above, the following would be written to the journal:

```
TxBegin | Iv2 | B2 | TxEnd
```

and D2 would simply be written to the file system. This modification does raise an interesting question; when should we write D2 to disk? Here are two possible options.

Option 1:

1. Write **D2** to disk
2. Write the transaction (containing Iv2, B2) to the journal
3. Write the blocks (Iv2, B2) to the file system proper
4. Mark the transaction free in the journal

Option 2:

1. Write the transaction (containing Iv2, B2) to the journal
2. Write the blocks (Iv2, B2, and **D2**) to the file system proper
3. Mark the transaction free in the journal

Thus, in ordered journaling, we can either write the data block D2 before the transaction, or after (with the other blocks). Option 2 is simpler and may perform better, but has a problem: it may end up with a consistent file system but one that has Iv2 pointing to garbage data. Specifically, if the file system is writing Iv2, B2, and D2 to disk and only manages to complete the first two writes before crashing, D2 will not be on the disk. The file system will then try to recover, but notice that D2 is *not* in the log. Thus, it will replay the writes to Iv2 and B2, and produce a consistent file system. However, Iv2 will be pointing to garbage data. Thus, option 1 is attractive, as it guarantees that Iv2 will never point to garbage by forcing it to disk first.

In real systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use non-ordered metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes.

32.8 Tricky Cases

There are some interesting corner cases that make journaling more tricky. One day, I will write about them.

32.9 Summary

We have introduced the concept of journaling. Journaling reduces recovery time from $O(\text{size of the disk volume})$ to $O(\text{size of the log})$, thus speeding recovery substantially after a crash and restart. Because of this, most file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data. However, journaling is not the only way to guarantee consistent updates to disk, as we will see in subsequent notes.

References

- [H87] "Reimplementing the Cedar File System Using Logging and Group Commit", Robert Hagmann, SOSP 1987.
- [M86] "Fsync - The UNIX File System Check Program"
Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry
April 1986
- [MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August, 1984. Volume 2, Number 3.
pages 181-197.
- [T98] "Journaling the Linux ext2fs File System"
Stephen C. Tweedie, The Fourth Annual Linux Expo, May 1998.
- [T01] "The Linux ext2 File System"
Theodore Ts'o, June, 2001.
Available: <http://e2fsprogs.sourceforge.net/ext2.html>