Contents

About these reading notes

These are my own personal reading notes that I took (me, Franklin) as I read the textbook. I'm providing these to you as an additional resource for you to use while you're reading chapters from the textbook. These notes do not stand alone on their own — you might be able to get the *idea* of a chapter while reading these, but you're definitely not going to **get** the chapter by reading these alone.

These notes are inconsistently all of the following:

- Me summarizing parts of the text.
- Me commenting on parts of the text.
- Me asking questions to myself about the text.
    - … and *sometimes* answering those questions.

The way that I would expect you to read or use these notes is to effectively permit me to be your inner monologue while you're reading the textbook. As you're reading chapters and sections within chapters, you can take a look at what I've written here to get an idea of how I'm thinking about this content.

Chapter 42: Crash Consistency: FSCK and Journaling

The big question here is: How does a file system handle a crash? More importantly, how does it deal with data that is corrupted because it didn't get fully written, or some other issue (random gamma rays) corrupted the data?

Remember that the disk only guarantees that the write of a single sector is atomic, and we've learned that the sector size (what the disk considers to be an atomic unit) and block size (what the file system considers to be an atomic unit) are different from one another.

It's also important to note that changes to the file system itself aren't atomic. Think about that for a second (and also maybe look back at chapter 40). What does a file system have to do when you write a new file to the disk? How many different data structures need to be updated? Can you write those changes to disk in one single atomic operation? (Uh, no). If your OS crashes in the middle of updating the data structures for writing a file, the file system itself is "inconsistent". For example, you might have updated inodes, but not updated the inode bitmap. Or alternatively, you updated the free space bitmap, but never assigned that block to an inode. Or alternatively, … you get the idea.

There's two approaches that this chapter will explore:

1. Deal with the problem *after* it's happened (fsck) at the expense of possibly losing data but keeping writes to the FS fast.

2. Deal with the problem *before* it's happened (journal) at the expensive of slowing down writes, but making sure you *can't* have half-written data.

42.1 A detailed example

The basic setup here is that we've got three blocks to write to a simple file system when appending to an existing file:

1. A write to the inode to update a direct pointer (1 block)
2. A write to replace the entire data block bitmap (1 block)
3. A write to write the data block itself (1 block)

Again, remember that block size != sector size, and only sector writes are atomic. If the system crashes at any point in between writing these blocks, the data structures may be left in an inconsistent state (corrupted).

The diagram here (pg 2) is an **extremely** simplified version of the FS diagrams that we saw in chapter 40, and the text is pretty much just stepping through the operations that are happening to add a new block to this file (update the data bit map, update the inode itself, write the data block).

> which is just filled with whatever it is users put into files. Stolen music perhaps?

This dates the book. Nobody steals music anymore. We steal TV shows. Also nobody saves those files to their disk, do they?

The last little bit about this (just before "Crash Scenarios") is talking a bit about virtual memory (paging). You don't need to be too concerned about this right now, other than to know that calling `write` on a file doesn't actually change anything on a disk immediately, the memory manager will handle this.

Crash scenarios

Three block writes (data, inode, data bitmap), three crash scenarios where only one of those three blocks is written:

1. We crash after only the data block has been written. No big deal, the data block might be written, but as far as the inode and data block bitmap are concerned, that data doesn't exist and the block is free to be used.
2. We crash after just the inode is written. This means that the direct pointer has been written, but the data block bitmap hasn't been updated, so the data in this block may be wildly overwritten by *other* file operations and inodes. This implicitly means that two inodes may be pointing at the same data block. Further, the inode currently points to uninitialized data (0s, random bits),

so the file contents itself are corrupted. Our file system is now "inconsistent", one of its data structures says one thing and a different data structure says another thing (an inode says "I point at block B" and the data bitmaps says "block B is free").
3. We crash after the bitmap has been written. The only issue that comes up here is that there will be a block that is effectively unusable because the bitmap says it's used, even though no inodes point to it. It's a leak!

There are three more crash scenarios where $\frac{2}{3}$ blocks are written:

1. We crash after the inode and bitmap are written. This is similar to 2 above (the inode points to uninitialized data), but this time we don't have to worry about some other inode being assigned to the same block.
2. We crash after the inode and data are written. This is similar to 2 above (multiple inodes may be pointing at the same data), but this time we don't have to worry about uninitialized data.
3. The data and bitmap are written. This is exactly the same as 3 above.

The crash consistency problem

In short, we *want* those three things to happen atomically, but we can't guarantee that. Bam, crash consistency problem.

Solution #1: the file system checker

Haha: fsck. Fsck == File System ChecKer.

Aside: I say this "fisk" (rhymes with disk), despite what the footnote says. Yay people trying to pronounce acronyms.

The main goal in file system checking is to make the data structures in the file system *consistent*. A file system checker cannot do anything about corrupted *data* that the data structures refer to.

> It is run *before* the file system is mounted and made available (`fsck` assumes that no other file-system activity is on-going while it runs);

Why is that? Why couldn't we run `fsck` on a file system while the file system is being used? "Used" here means reading from and writing to.

Order of operations:

1. Check the superblock in the file system. This step is basic sanity checking, lots of checking that this impossible situation isn't true (i.e., size of the file system is greater than blocks allocated).

Aside: If the file system checker finds that the superblock is inconsistent, what *could* it do? Look back at the simple FS in chapter 40: can you fix what's stored in the superblock?

2. Go through the inodes and verify that nobody is pointing at what is considered to be a free block in the data bitmap. Basically: make sure that we didn't crash after having written an inode but before having written the data bitmap. Likewise, let's make sure that all of the bits marked in the inode bitmap actually point at inodes.
3. Similar to the superblock check, fscking will also check all inodes for basic consistency. The things that it can check are things that have well-known values, like "type" on an inode (a regular file, directory, link, etc).
4. Checking *links*. An inode that doesn't have a name (it's not linked in any directory) gets put into the mysterious `lost+found` directory. Wait, what does this mean? What does it mean for an inode to not be linked to? Think back to the file system implementation, how are inodes related to each other and structured?

   Aside: Who should have permission to read the `lost+found` directory? Who should have permission to read the "file" that's dropped into the `lost+found` directory?
5. Check all inodes to make sure no two inodes point to the same data blocks (look back up: this is one of those failure modes).
6. Check that all inode pointers actually point to something on the file system's address space (e.g., if you have a 2GB disk, but the inode pointer is pointing at 493GB, then something is wrong). Importantly, this is **not** checking the contents of the data blocks.
7. Directory checks. Directories are inodes, but not files. The checker can check everything it knows about with regard to directories. **Aside**: is the *name* of a directory something that a file system check can do anything about?

In summary, file system checking is slow, especially when we're possibly trying to recover from just a single missing block (4KB out of possibly TB of data).

It's really important to note that `fsck` **cannot** repair your data. The only thing that checking the file system can do is check the **file system** and its data structures.

43.2 Solution #2: Journaling (or Write-Ahead Logging)

Dear Diary…

Basically all modern file systems use journaling to maintain consistency of the file system metadata and data *while* in operation.

The short of the idea is that we're going to try and avoid doing in-place structure updates and instead write some order of operations in a separate, well-known area (the journal) before committing the writes. If we write down what we planned to do before a crash happens, when a crash *does* happen, we can pretty quickly figure out how far we got then continue along. If a crash happens while we're writing down the steps, no big deal, the changes to the file system weren't being done in-place, so no corruption happens. Neat!

The tradeoff here is that we're spending more time for every write to save time when trying to recover from a crash.

Aside: How do *you* make this tradeoff? How do you (as a human person) make a decision between "I want writes to be fast and I don't care about disk consistency" vs "My disk should always be in a consistent state, even if writes take longer."

The ext3 file system is *basically* ext2 + journaling.

Data journaling

Just write *everything* to the journal area before modifying data structures. We write each of these block sets to the journal as a **transaction** (Database ideas?!). When we write the blocks to the journal, we write transaction start and end blocks flanking the blocks to write to the disk.

Aside: We just spent a bunch of time talking about what happens when a crash happens in between writing 3 blocks, now we're writing *5* (the original 3 plus 2 more for transaction start and transaction end). What is the outcome of crashing in between writing those blocks to the journal area? How can the file system recover from a botched write? (The answer to this is later)

Only after the journal is complete are we ready to start modifying the data structures on the disk. The part where we start writing the journal to the disk is called a "checkpoint".

Aside: Before we read any more about this, the overhead here seems absurd – for every write we're going to be writing the same data *twice*: once to the journal, then again to the actual file system.

Now we consider writing the *transaction* to disk. Again, we're dealing with 5 blocks here instead of 3. Oh no! No matter what order we send those 5 blocks in (batch or otherwise), the *disk* can schedule those writes because they are not collectively atomic! That means that the transaction start and end blocks could be written before the *rest* of the blocks and the *journal* would be in an inconsistent state.

So instead of doing all 5 writes at once, we do *four*: the transaction begin, the inode, the data bitmap, and the data. Once we've confirmed that the first four are written, we can send the transaction end write.

It's really important to note atomicity again: just like with threads, atomicity is a problem with disks (is this the *same* atomicity problem, or a different atomicity problem?). We (the OS) make sure that

the transaction end (the commit) is an atomic write by making sure that it fits into a single sector (512 bytes).

We're using more database words here: Journal write, journal **commit**, and checkpoint.

Aside: What came first, the relational database transaction, or the journaling file system transaction? It's definitely the relational database transaction, relational databases are way older than journaling file systems.

Recovery

Ok, so a crash happens (this means that the system gets turned off, or, uh, you spilled coffee on it or something). On the next boot, the journal gets "replayed" *before* the file system is mounted and other read/writes start to happen. This *might* mean that the recovery operation writes over data that was *already* written, but… that's no big deal because it's just writing the same data over itself. The main goal here is consistency and we're definitely maintaining consistency.

So try to imagine this for a second in terms of the transaction above. Your file system starts checkpointing the transaction to disk and crashes in the middle of it. Let's say that you've succeeded in writing the inode changes just before the disk crashes (so `I[v2]` is physically written to disk, but `B[v2]` and `Db` are not). When the system starts, the OS will see that it hasn't finished checkpointing the transaction because `TxE` is still there. So it will start replaying the transaction from the beginning, and will write `I[v2]` *again*. That's totally OK!

Batching log updates

Some workloads will start writing the same blocks over and over again, so we can buffer these log updates and start merging them together.

Aside: Gee… this sounds *awfully* familiar. (look back at disk scheduling)

This is a more involved operation than what the disk might have done with write coalescing, because we're talking about merging data structures rather than ordering writes that might just overwrite another block, but the *basic* idea is still the same.

Making the log finite

Resources are finite.

Aside: Gee… this sounds *awfully* familiar. (and if it doesn't, it will when you start looking at virtual memory)

The journal can't go on forever. The file system has an additional responsibility with logs to track the free space *within* a log. The authors also mention using a circular log (a circular *queue*???) to represent the log.

To maintain free space about the journal, the journal itself might have a superblock with that free space metadata (is this a file system within a file system?). It may share similar properties to the main file system superblock, in that it has a bitmap indicating which parts of the journaling area are free.

Aside: Ok, the authors restate my concern from above: we're doing this work *twice*.

Metadata journaling

Writing twice is *bad*, what do?

There are different kinds of journaling. Data journaling (what we've been looking at) will write *everything* to the journal, including the data blocks.

Ordered journaling or **metadata journaling** is the same as data journaling, except it doesn't include the user data. The idea here is that we just write the data block to its target location *before* writing to the journal. What's the worst that could happen? We have a data block written to the file system with no inode that points to it, and the data bitmap still thinks it's free space.

Linux lets you choose what kind of journaling is going to be used (data, ordered journaling or unordered journaling). Specifically, these are mount options for `ext` (i.e., `mount -o data=(journal|ordered|writeback)`, see `man ext4`).

Aside: Neat, I've seen these options before but only in the context of "these flags will make things go faster". Now I know why!

Tricky case: block reuse

I think that the issue they're describing here is limited to unordered metadata journaling. Unordered metadata journaling here means that we don't know when the data block is written, it might be written before the metadata or it might be written after, we don't care (this is defined at the top of pg 15). The block that's getting reused is from a previously created and deleted directory, and only the file inode is written to the journal before the crash.

Actually, I *think* it doesn't matter? The key here is that we're deleting something, but the journal doesn't know about deletes?

Aside: Why wouldn't the journal know about deletes?

The authors then say that ext3 *does* know about deletes by adding a new kind of journal entry called a "revoke" record.

Note that Figure 42.1 is poorly placed above this discussion, when it actually belongs in the next section…

Wrapping up journaling: a timeline

This section describes figures 42.1 and 42.2, pretty much showing the lifecycle of a transaction as it gets committed to disk.

42.4 Solution #3: other approaches

Journaling seems like a neat idea, but everyone's got an opinion. Fsck is pretty lazy, and the defining attribute of any programmer is laziness…

"Soft updates" is effectively doing I/O scheduling operations at the file system level – make sure that block writes happen in such a way that they *can't* be corrupted.

I first thought: Copy-on-write has *nothing* to do with copying, the authors say that we just write new data structures to otherwise free space. But what we're really doing is *copying existing structures* to free space, then modifying them, then updating the file system pointers to point to the new data structures. I'd assume that the pointer changes can be done atomically by ordering the data structures such that the region where pointers are kept can be guaranteed to be written atomically by the disk.

If you're deeply interested in file system stuff, you should proceed as the authors suggest and look at LFS in chapter 43.

Backpointer-based consistency seems to be included as if to say "Look, we published a paper!" They describe this as a way to amend fscking rather than an inconsistency prevention scheme.

"Transaction checksum" – used, but never defined. If I were a guessing person (and I'm not going to read their paper), there's some kind of checksum used here to identify similar or duplicate writes. Again, this last little chunk very much feels like a self-congratulatory (and genuinely well-deserved) pat on the back.

42.5 Summary

> Journaling reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log).

OK yeah, that makes sense, right? Trading write time for recovery speed.

Aside: But if journalling means we don't need to do fsck, why do journaling file systems like ext3 and ext4 have corresponding fsck commands?

A major takeaway from this chapter is that just like with threads, file systems have "operations that turn into multiple steps that aren't atomic" causes problems. The solution with threads was locks. Is a journal in a file system kind of like a lock? Is it? How does fsck fit into this? Could you add fsck-ing to data structures being concurrently modified by threads? Does that even make any sense?