Dmitry Mazin

"cyberdemon.org is a cool domain"
home / email / mastodon / RSS feed / Telegram channel

How does Linux really handle writes?

Jun 27, 2023

Please feel free to join the lively discussion on Hacker News.

My friends – programmers and operators – I would like to talk to you about the way that file writes work in Linux.

I used to think they worked a certain way, and like John Lennon, I'm not the only one. It turns out that they work very differently. The way they really work is interesting, and important to know.

Let us begin by stating the way I used to think file writes worked.

1. you `echo "foo" > bar.txt`
2. (microseconds later) boom, done. "foo" has been written to disk.

I thought it worked this way because I thought that files lived on disk, so if you wrote to a file, you were writing to disk.

It does not work this way, and the above mental model, about files living on disk, is wrong.

Let's start by showing why that mental model is wrong.

Files don't live on disk

It's reasonable to think that files live on disk, because files are what we interact with in order to write stuff to disk.

But, that's exactly it: a file is an *interface*. The operating system uses this interface so that we can tell it what we want.

This is kind of theoretical, so let me say it again: files are an interface, much like a OOP instance with methods and attributes. The file is not the stuff on disk. It's just the abstract interface for it.

What *does* live on disk, then? Bytes.

Bytes live on disk

A disk is just a bag of bytes. These bytes have a structure, of course. If they didn't, they would be random, and we would have no way to ever make sense of them. The specific way we order bytes on disk is called an *inode*. An *inode* is what we end up representing using a *file*.

One way to think of an inode is the same way you think of a jpeg: it's a way to order bytes in a certain way. For example, an inode specifies that at a certain location in your bag of bytes, you put the file size. At another location, you put when the file was created.

Why don't we just interact with the inodes directly, then?

Why we use interfaces

You *can* directly write to disk. You need to know exactly where to write, and you need to write the bytes directly. The chance of errors is really high.

Instead, it's much nicer to just ask the operating system to do it. That way, we can focus on our own applications.

So, that's one reason we use interfaces: to abstract away stuff we don't want to do.

The other big reason is efficiency. By delegating disk access to the operating system, we give the operating system programmers permission to make a lot of efficient choices.

Disks are slow

Let's go back to this statement.

1. you `echo "foo" > bar.txt`

2. (microseconds later) boom, done. “foo” has been written to disk.

This is totally false. If it were true, computers would feel terribly slow. Disks are sloooow.

How slow? Fetching a small amount of data from SSD is 1,000 times slower than memory. Fetching that same data from a spinning hard drive is one MILLION times slower. Disk access is multiple orders of magnitude slower than memory access!

Consider that, before SSDs, disks were the last mechanical thing about computers (other than fans). In a world of speedy electron-pushing, we were *actually pushing atoms*. The disparity in speed between the electrical and mechanical world is huge.

So, operating systems do what they can to shield applications from this slowness.

How do they do it?

How do writes actually work?

Here’s how the write really works.

1. you `echo "foo" > bar.txt`
2. the operating system copies “foo” into a special place in memory called the *page cache*
3. (microseconds later) the operating system tells you the write succeeded
4. (asynchronously, up to 30 seconds later) the operating system actually writes “foo” to disk

If your mental model was “files live on disk”, the above is shocking. I mean, I used to think that the disk write happened immediately, but actually it can happen 30 seconds later!

Why the asynchronicity?

Let’s say that you made a photo sharing app with Like buttons. Photo Likes *are* stored in a database. However, if you personally were designing this app, would you…

1. Make the Like heart appear as soon as the user clicks the Like button?
2. Make the Like heart appear only once the Like has been persisted to disk?

You would go with option 1, of course, because UI responsiveness is really important.

This is an example of shielding a user from a slow operation via *asynchronicity*: you tell the user you did it, but actually, you do it later when they're not looking.

Linux does something similar. It shields the application from disk slowness by simply doing the disk write later. This is called non-blocking IO: don't make the application wait for slow disks.

This isn't the only reason the write is asynchronous, though.

Buffering also makes disk writes faster

Say this three times fast: asynchronicity enables buffering.

I want to explore this deeply some day, but I'm amazed how frequently I find buffers and queues in computer science. It's not like they're secret, but they more central to efficient computing than I ever knew.

Queues and buffers make disk writes efficient, too.

For example, every disk write has overhead. Given that overhead, we would rather do one big write of 1 megabyte than 100 little writes of 10 kilobytes each. Buffering disk writes allows the operating system to merge those little writes into bigger writes.

Because the operating system decouples the file write from the actual disk write, if you do a bunch of file writes quickly, they will bunch up into a little queue. The operating system can then merge them.

There are actually way, way more tricks that the operating system does to writes to make them performant, all of them enabled by this asynchronicity. Perhaps I'll cover them in another article, but if you are dying to know, I highly recommend the File System and Disk chapters of Gregg's Systems Performance.

Anyway, we understand now that asynchronous disk writes give us a huge speed boost, but there's an elephant in the room: at what cost?

The tradeoff between efficiency and durability

What happens if you unplug your computer before the operating system writes the data to disk? Well, you will lose the data. It's as simple as that.

I'd love to find a discussion where the decision was deliberately made, but it's also kind of a no-brainer: until recently, disk writes were a million times slower than memory access. Of course the default would be to make writes asynchronous.

What if we do want to make our writes durable, though?

`O_SYNC` and `sync`: making writes durable

While I found the fact that writes are asynchronous surprising, many programmers, certainly database programmers, know this well.

There are many ways to make writes durable. I'll cover two of them.

`sync` syscall

There's a system call called `sync`, which you can call at any time. It says: operating system, dump everything from page cache to disk NOW! And the operating system will do it. You can even put that command into your shell right now just to try it.

In programs, you will often see a series of writes interspersed with `sync` calls. Like, every `n` writes, there will be a `sync` call. This allows applications to finely tune the tradeoff between durability and write throughput.

`O_SYNC` file mode

Programmers can also open a file in `O_SYNC` mode. This works much closer to the mental model at the beginning of this article: the write only completes once the data has been persisted to disk.

Demonstrating async writes

Time for a little science demonstration. I'm going to write to a file. I'm going to read from that file. I will show you that both operations will complete seconds before the disk is accessed.

This is what I'm going to paste into my shell, all at once.

```
echo "foo" > example.txt
dd if=example.txt
```

And here is the output of `bpftrace`, which allows us to trace kernel events.

Specifically, we are going to trace when `vfs_read` and `vfs_write` finish (all you need to know is that those are the functions that are called when we read and write a file). We will also trace `block_rq_issue`, which is called when the disk driver actually writes to disk.

```
# write starts
11:32:05 kfunc:vmlinux:vfs_write

# write finishes
11:32:05 kretfunc:vmlinux:vfs_write

# read starts
11:32:05 kfunc:vmlinux:vfs_read

# read finishes
11:32:05 kretfunc:vmlinux:vfs_read

# **5 seconds later** we actually write "foo" to disk!
11:32:10 tracepoint:block:block_rq_issue
```

Note that we never even read disk, even though we read the file. That's because the read came from the page cache!

Caveat: this behavior isn't universal

Whether or not the page cache is used for reads and writes is actually filesystem-specific. However, all the most common Linux filesystems (ext4, btrfs, XFS) use the page cache. The story with ZFS is more complicated. While it uses the page cache, it also has its own relatively complex caching mechanism.

Conclusion

I hope I helped make your mental model of file writes more subtle. Files don't live on disk, bytes do. Files merely represent those bytes. This misdirection makes programming easier, and allows the operating system to shield our fast applications from slow disks.

There is a lot more to this subject – the difference between file IO and disk IO is even more profoundly different than I let on. For example, if you write 1 byte to a file, how much will get written to disk? Would you be surprised if I told you that that 1 byte write causes 65 THOUSAND bytes to get written to disk? If that's interesting to you, I urge you to subscribe to this blog. More stuff like this is coming.

Thanks for reading.

PS: I'm looking for a Linux job

I've had a happy career as a cloud-based SRE. Most recently, I ran the infrastructure team at Rollbar. But, now, I want to work more directly with Linux. I'm London-based and am interviewing for Linux engineering and/or HPC jobs. If you have any leads, reach out! I'm happy to share my CV. Thanks!

Here some of my most interesting posts.

- How does Linux really handle writes?
- Reddit is OpenAI's Moat
- What's Writing to My Brand-new Disk?
- On OpenAI's Terrible Arguments Against Transparency
- In the Age of AI, Don't Let Your Skills Atrophy
- Using ChatGPT to Generate Git Commit Messages
- I Hereby Declare a Sabbatical
- Pink Lexical Slime: The Dark Side of Autocorrect