xavier roche's homework

random thoughts and sometimes pieces of code

- RSS

Search

- Blog
- Archives

Everything You Always Wanted to Know About Fsync()

Nov 15th, 2013

And then the developer wondered:

> is my file properly sync'ed on disk ?

You probably know *more or less* how databases (or things that look like one) store their data on disk in a *permanent* and *safe* way. Or at least, you know the basic principles. Or not ?

Being on AC(I)D

There are a bunch of concepts that first must be understood: what is **atomicity**, **consistency**, and **durability** ? These concepts apply on databases (see ACID), but also on the underlying filesystem.

- **Atomicity**: a write operation is fully executed at once, and is not *interleaved* with another one (if, for example, someone else is writing to the same location)

  Atomicity is typically guaranteed in operations involving filename handling ; for example, for rename, "*specification requires that the action of the function be atomic*" – that is, when renaming a file from the old name to the new one, at no circumstances should you ever see the two files at the same time.

- **Consistency**: integrity of data must be maintained when executing an operation, even in a crash event – for example, a power outage in the middle of a `rename()` operation shall not leave the filesystem in a "weird" state, with the filename being unreachable because its metadata has been corrupted. (ie. either the operation is lost, or the operation is committed.)

  Consistency is guaranteed on the filesystem level ; but you also need to have the same guarantee if you build a database on disk, for example by serializing/locking certain operations on a working area, and committing the transaction by changing some kind of generation number.

- **Durability**: the write operation is durable, that is, unplugging the power cord (or a kernel panic, a crash…) shall not lose any data (hitting the hard disk with a hammer is however not covered!)

> This is an important one – at a given point, you must ensure that the data is actually written on disk *physically*, preventing any loss of data in case of a sudden power outage, for example. This is absolutely critical when dealing with a client/server architecture: the client may have its connection or transaction aborted at any time without troubles (ie. the transaction will be retried later), but once the server acknowledges it, no event should ever cause it to be lost (think of responsibility in a commercial transaction, or a digital signature, for example). For this reason, having the data committed in the internal system or hard disk cache is NOT durable for obvious reasons (unless there is a guarantee that no such power outage could happen – if a battery is used on a RAID array, for example).
>
> On POSIX systems, durability is achieved through sync operations (`fsync()`, `fdatasync()`, `aio_fsync()`): "*The fsync() function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the fsync() call is recorded on the disk.*". [Note: The difference between fsync() and fdatasync() is that the later does not necessarily update the meta-data associated with a file – such as the "last modified" date – but only the file data.]

Now that these concepts are a bit clearer, let's go back to our filesystem!

Hey, What is a File, By The Way ?

If we want to simplify the concept, let's consider the filesystem on POSIX platforms as a very simple *flat* storage manager, allowing to read/write data blobs and basic properties (such as the modified time) indexed by an *integer* number (hint: they sometimes call that the *inode number*).

For example, you may want to read the file #4242's data. And later, write some data on file #1234.

To have a more convenient way to handle files (because "I need to send you the presentation number 155324" would not be really convenient in the real world), we use the **filename/directory** concepts. A file has a name, and it is *contained* within a *directory structure*. You may put files and directories in a directory, building a hierarchical structure. But everything rely on our previous basic data blobs to store both filename and the associated index.

As an example, reading the file `foo/bar.txt` (ie. the file `bar.txt` within the `foo` directory) will require to access the data blob associated with the directory `foo`. After parsing this opaque data blob, the system will fetch the entry for `bar.txt`, and open the associated data blob. (And yes, there is obviously a root entry, storing references to first-level entries, allowing to access any file top-down)

If I now want to create a new file named `foo/baz.txt`, it will require the system to access the data blob associated with the directory `foo`, add an entry named `baz.txt` with a new allocated index for the upcoming file, and write the updated directory blob back, and from this point, write to the newly allocated blob. The operation therefore involves **two** data structures: the **directory entry**, and the **file** itself.

Keeping My File(name) Safe

Let's go back to our database problem: what is the impact of having *two* data structures for our files ?

Atomicity and **consistency** of filenames are handled for us by the filesystem, so this is not really a bother.

What about **durability** ?

We know that `fsync()` provides guarantees related to data and meta-data sync'ing. But if you look closer to the specification, the only data involved are the one related to the file itself – not its directory entry. The "metadata" concept involves modified time, access time etc. – **not** the *directory entry* itself.

It would be cumbersome for a filesystem to provide this guarantee, by the way: on POSIX systems, you can have an arbitrary number of directory links to a filename (or to another directory entry). The most common case is *one*, of course. But you may delete a file being used (the file entry will be removed by the system when the file is closed) – the very reason why erasing a log file which is flooding a filesystem is a futile and deadly action – in such case, the number of links will be *zero*. And you may also create as many hard-links as you want for a given file/directory entry.

Therefore, in theory, you may create a file, write some data, synchronize it, close the file, and see your precious file lost forever because of a power outage. Oh, the filesystem must guarantee consistency, of course, but not durability unless *explicitly* asked by the client – which means that a filesystem check *may* find your directory entry partially written, and decide to achieve consistency by taking the previous directory blob entry, **wiping** the unreferenced file entry (note: if you are "lucky" enough, the file will be expelled in `lost+found`)

The filesystem can, of course, decide to be gentle, and commit all filename operations when fsync'ing. It may also, such as for ext3, commit **everything** when fsync'ing a file – causing the infamous and horrendous lags in firefox or thunderbird.

But if you need to have **guarantees**, and not just *hope* the filesystem "*will be gentle*", and do not want to "*trust the filesystem*" (yes, someone actually told me that: you *need* to "trust the filesystem" – I swear it), you have to actually make sure that your **filename** entry is properly sync'ed on disk following the POSIX specification.

Oh, and by the way: according to POSIX, *The fsync() function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the fsync() call is recorded on the disk.*

But things are sometimes a bit obscure on the implementation side :

- **Linux/ext3**: *If the underlying hard disk has write caching enabled, then the data may not really be on permanent storage when fsync() / fdatasync() return*. (do'h!)
- **Linux/ext4**: *The fsync() implementations in older kernels and lesser used filesystems does not know how to flush disk caches*. (do'h!) – issue adressed quite recently
- **OSX**: *For applications that require tighter guarantees about the integrity of their data, Mac OS X provides the F_FULLFSYNC fcntl. The `F_FULLFSYNC` fcntl asks the drive to flush all buffered data to permanent storage* (hey, fsync was supposed to do that, no ? guys ?) *(Edit: no, fsync is actually not required to do that – thanks for the clarification Florent!)*

But we may assume that on Linux with ext4 (and OSX with proper flags ?) the system is properly propagating write barriers.

On Windows, using `FlushFileBuffers()` is probably the way to go.

Syncing Filenames

I told you that a filesystem was actually a bunch of *flat* data blobs with associated metadata, and that a file had actually two parts: its directory entry (let's assume there is only one directory entry for the sake of simplicity), and its actual data. We already know how to sync the later one ; do we have a way to do the same for the directory container itself ?

On POSIX, you may actually open a directory as if you were opening a file (hint: a directory is *a file that contains directory entries*). It means that `open()` *may* successfully open a directory entry. But on the other hand, you generally can not open a directory entry for *writing* (see POSIX remark regarding `EISDIR`: *The named file is a directory and oflag includes O_WRONLY or O_RDWR*), and this is perfectly logical: by directly writing to the internal directory entry, you may be able to mess up with the directory structure, ruining the filesystem **consistency**.

But can we fsync() written data using a file descriptor opened *only* for reading ? The question is… yes, or at least "*yes it should*" – even POSIX group had *editorial* inconsistencies regarding fdatasync and aio_fsync(), leading to incorrect behavior on various implementations. And the reason it should execute the operation is because requesting the completion of a write operation does not have to require actual write access – which have already been checked and enforced.

On Windows… err, there is no clear answer. You can not call `FlushFileBuffers()` on a directory handle as far as I can see.

Oh, a last funny note: how do you sync the **content** of a *symbolic link* (and its related meta-data), that is, the filename pointed by this link ? The answer is… you can't. Nope. This is not possible with the current standard (hint: you can not `open()` a symbolic link). Which means that if you handle some kind of database generation update based on symbolic links (ie. changing a "last-version" symlink to the latest built generation file), you have zero guarantee over durability.

Conclusion

Does it means that we need to call `fsync()` *twice*, one on the file data, and one on its parent directory ? When you need to achieve *durability*, the answer is obviously **yes**. (Remember that file file/filename will be sync'ed on disk anyway by the operating system, so you do not actually need to do that for every single file – only for those you want to have a durability guarantee at a given time)

However, the question is causing some headache on the POSIX standard, and as a follow-up to the austin-group (ie. POSIX mailing-list) discussion, an editorial clarification request is still pending and is waiting for feedback from various implementors. (you may also have a look at the comp.unix.programmer discussion)

TL;DR: syncing a file is not as simple as it seems!

Posted by Xavier Roche Nov 15th, 2013 c, posix, programming

Tweet

« Coding is like Cooking I Do Not Want Your Search Bar »

Comments

Recent Posts

- Redirecting Stdio Is Hard
- I Found a Bug in Strncat()
- Coucal, Cuckoo-hashing-based Hashtable With Stash Area C Library
- C Corner Cases (and Funny Things)
- A Basic Glance at the Virtual Table
- A Story of Realloc (and Laziness)
- Security Is Painful
- What Books Really Helped You ?
- Embrace Unicode (and Do Not Worry)
- What Are Your GCC Flags ?
- Fancy Standalone Visual C++ Compiler
- I Do Not Want Your Search Bar
- Everything You Always Wanted to Know About Fsync()
- Coding Is Like Cooking
- Creating Deletable and Movable Files on Windows
- An Unusual Case of Case (/switch)
- Replacing JNI Crashes by Exceptions on Android
- HTTrack on Android
- MD5 Is Your Friend
- So, I Made a Hashtable

GitHub Repos

- Status updating...

@xroche on GitHub