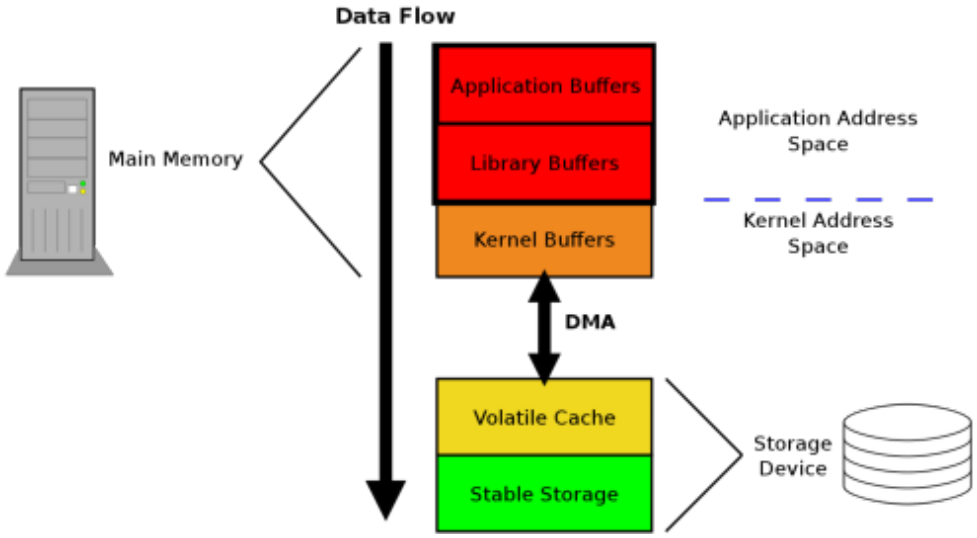
Ensuring data reaches disk

September 7, 2011
This article was contributed by Jeff Moyer

In a perfect world, there would be no operating system crashes, power outages or disk failures, and programmers wouldn't have to worry about coding for these corner cases. Unfortunately, these failures are more common than one would expect. The purpose of this document is to describe the path data takes from the application down to the storage, concentrating on places where data is buffered, and to then provide best practices for ensuring data is committed to stable storage so it is not lost along the way in the case of an adverse event. The main focus is on the C programming language, though the system calls mentioned should translate fairly easily to most other languages.

I/O buffering

In order to program for data integrity, it is crucial to have an understanding of the overall system architecture. Data can travel through several layers before it finally reaches stable storage, as seen below:



At the top is the running application which has data that it needs to save to stable storage. That data starts out as one or more blocks of memory, or buffers, in the application itself. Those buffers can also be handed to a library, which may perform its own buffering. Regardless of whether data is buffered in application buffers or by a library, the data lives in the application's address space. The next layer that the data goes through is the kernel, which keeps its own version of a write-back cache called the page cache. Dirty pages can live in the page cache for an indeterminate amount of time, depending on overall system load and I/O patterns. When dirty data is finally evicted from the kernel's page cache, it is written to a storage device (such as a hard disk). The storage device may further buffer the data in a volatile write-back cache. If power is lost while data is in this cache, the data will be lost. Finally, at the very bottom of the stack is the non-volatile storage. When the data hits this layer, it is considered to be "safe."

To further illustrate the layers of buffering, consider an application that listens on a network socket for connections and writes data received from each client to a file. Before closing the connection, the server ensures the received data was written to stable storage, and sends an acknowledgment of such to the client.

After accepting a connection from a client, the application will need to read data from the network socket into a buffer. The following function reads the specified amount of data from the network socket and writes it out to a file. The caller already determined from the client how much data is expected, and opened a file stream to write the data to. The (somewhat simplified) function below is expected to save the data read from the network socket to disk before returning.

```
 0 int
 1 sock_read(int sockfd, FILE *outfp, size_t nrbytes)
 2 {
 3      int ret;
 4      size_t written = 0;
 5      char *buf = malloc(MY_BUF_SIZE);
 6
 7      if (!buf)
 8              return -1;
 9
10      while (written < nrbytes) {
11              ret = read(sockfd, buf, MY_BUF_SIZE);
12              if (ret =< 0) {
13                      if (errno == EINTR)
14                              continue;
15                      return ret;
16              }
17              written += ret;
18              ret = fwrite((void *)buf, ret, 1, outfp);
19              if (ret != 1)
20                      return ferror(outfp);
21      }
22
23      ret = fflush(outfp);
24      if (ret != 0)
25              return -1;
26
27      ret = fsync(fileno(outfp));
28      if (ret < 0)
29              return -1;
30      return 0;
31 }
```

Line 5 is an example of an application buffer; the data read from the socket is put into this buffer. Now, since the amount of data transferred is already known, and given the nature of network communications (they can be bursty and/or slow), we've decided to use libc's stream functions (fwrite() and fflush(), represented by "Library Buffers" in the figure above) in order to further buffer the data. Lines 10-21 take care of reading the data from the socket and writing it to the file stream. At line 22, all data has been written to the file stream. On line 23, the file stream is flushed, causing the data to move into the "Kernel Buffers" layer. Then, on line 27, the data is saved to the "Stable Storage" layer shown above.

I/O APIs

Now that we've hopefully solidified the relationship between APIs and the layering model, let's explore the intricacies of the interfaces in a little more detail. For the sake of this discussion, we'll break I/O down into three different categories: system I/O, stream I/O, and memory mapped (mmap) I/O.

System I/O can be defined as any operation that writes data into the storage layers accessible only to the kernel's address space via the kernel's system call interface. The following routines (not comprehensive; the focus is on write operations here) are part of the system (call) interface:

Operation	Function(s)
Open	`open()`, `creat()`
Write	`write()`, `aio_write()`, `pwrite()`, `pwritev()`
Sync	`fsync()`, `sync()`
Close	`close()`

Stream I/O is I/O initiated using the C library's stream interface. Writes using these functions may not result in system calls, meaning that the data still lives in buffers in the application's address space after making such a function call. The following library routines (not comprehensive) are part of the stream interface:

Operation	Function(s)
Open	`fopen()`, `fdopen()`, `freopen()`
Write	`fwrite()`, `fputc()`, `fputs()`, `putc()`, `putchar()`, `puts()`
Sync	`fflush()`, followed by `fsync()` or `sync()`
Close	`fclose()`

Memory mapped files are similar to the system I/O case above. Files are still opened and closed using the same interfaces, but access to the file data is performed by mapping that data into the process' address space, and then performing memory read and write operations as you would with any other application buffer.

Operation	Function(s)
Open	`open()`, `creat()`
Map	`mmap()`
Write	`memcpy()`, `memmove()`, `read()`, or any other routine that writes to application memory
Sync	`msync()`
Unmap	`munmap()`

Close	`close()`

There are two flags that can be specified when opening a file to change its caching behavior: `O_SYNC` (and related `O_DSYNC`), and `O_DIRECT`. I/O operations performed against files opened with `O_DIRECT` bypass the kernel's page cache, writing directly to the storage. Recall that the storage may itself store the data in a write-back cache, so `fsync()` is still required for files opened with `O_DIRECT` in order to save the data to stable storage. The `O_DIRECT` flag is only relevant for the system I/O API.

Raw devices (`/dev/raw/raw`*N*) are a special case of `O_DIRECT` I/O. These devices can be opened without specifying `O_DIRECT`, but still provide direct I/O semantics. As such, all of the same rules apply to raw devices that apply to files (or devices) opened with `O_DIRECT`.

Synchronous I/O is any I/O (system I/O with or without `O_DIRECT`, or stream I/O) performed to a file descriptor that was opened using the `O_SYNC` or `O_DSYNC` flags. These are the synchronous modes, as defined by POSIX:

- `O_SYNC`: File data and all file metadata are written synchronously to disk.
- `O_DSYNC`: Only file data and metadata needed to access the file data are written synchronously to disk.
- `O_RSYNC`: Not implemented

The data and associated metadata for write calls to such file descriptors end up immediately on stable storage. Note the careful wording, there. Metadata that is not required for retrieving the data of the file may not be written immediately. That metadata may include the file's access time, creation time, and/or modification time.

It is also worth pointing out the subtleties of opening a file descriptor with `O_SYNC` or `O_DSYNC`, and then associating that file descriptor with a libc file stream. Remember that `fwrite()`s to the file pointer are buffered by the C library. It is not until an `fflush()` call is issued that the data is known to be written to disk. In essence, associating a file stream with a synchronous file descriptor means that an `fsync()` call is not needed on the file descriptor after the `fflush()`. The `fflush()` call, however, is still necessary.

When Should You Fsync?

There are some simple rules to follow to determine whether or not an `fsync()` call is necessary. First and foremost, you must answer the question: is it important that this data is saved now to stable storage? If it's scratch data, then you probably don't need to `fsync()`. If it's data that can be regenerated, it might not be that important to `fsync()` it. If, on the other hand, you're saving the result of a transaction, or updating a user's configuration file, you very likely want to get it right. In these cases, use `fsync()`.

The more subtle usages deal with newly created files, or overwriting existing files. A newly created file may require an `fsync()` of not just the file itself, but also of the directory in which it was created (since this is where the file system looks to find your file). This behavior is actually file system (and mount option) dependent. You can either code specifically for each file system and mount option combination, or just perform `fsync()` calls on the directories to ensure that your code is portable.

Similarly, if you encounter a system failure (such as power loss, `ENOSPC` or an I/O error) while overwriting a file, it can result in the loss of existing data. To avoid this problem, it is common practice (and advisable) to write the updated data to a temporary file, ensure that it is safe on stable storage, then rename the temporary file to the original file name (thus replacing the contents). This ensures an atomic update of the file, so that other readers get one copy of the data or another. The following steps are required to perform this type of update:

1. create a new temp file (on the same file system!)
2. write data to the temp file
3. `fsync()` the temp file
4. rename the temp file to the appropriate name
5. `fsync()` the containing directory

Checking For Errors

When performing write I/O that is buffered by the library or the kernel, errors may not be reported at the time of the `write()` or the `fflush()` call, since the data may only be written to the page cache. Errors from writes are instead often reported during calls to `fsync()`, `msync()` or `close()`. Therefore, it is very important to check the return values of these calls.

Write-Back Caches

This section provides some general information on disk caches, and the control of such caches by the operating system. The options discussed in this section should not affect how a program is constructed at all, and so this discussion is intended for informational purposes only.

The write-back cache on a storage device can come in many different flavors. There is the volatile write-back cache, which we've been assuming throughout this document. Such a cache is lost upon power failure. However, most storage devices can be configured to run in either a cache-less mode, or in a write-through caching mode. Each of these modes will not return success for a write request until the request is on stable storage. External storage arrays often have a non-volatile, or battery-backed write-cache. This configuration also will persist data in the event of power loss. From an application programmer's point of view, there is no visibility into these parameters, however. It is best to assume a volatile cache, and program defensively. In cases where the data is saved, the operating system will perform whatever optimizations it can to maintain the highest performance possible.

Some file systems provide mount options to control cache flushing behavior. For ext3, ext4, xfs and btrfs as of kernel version 2.6.35, the mount option is "`-o barrier`" to turn barriers (write-back cache flushes) on (the default), or "`-o nobarrier`" to turn barriers off. Previous versions of the kernel may require different options ("`-o barrier=0,1`"), depending on the file system. Again, the application writer should not need to take these options into account. When barriers are disabled for a file system, it means that fsync calls will not result in the flushing of disk caches. It is expected that the administrator knows that the cache flushes are not required before she specifies this mount option.

Appendix: some examples

This section provides example code for common tasks that application programmers often need to perform.

1. Synchronizing I/O to a file stream
2. Synchronizing I/O using file descriptors (system I/O) This is actually a subset of the first example and is independent of the `O_DIRECT` open flag (so will work whether or not that flag was specified).
3. Replacing an existing file (overwrite).
4. sync-samples.h (needed by the above examples).

Index entries for this article

Kernel	Data integrity
GuestArticles	Moyer, Jeff

(Log in to post comments)

Ensuring data reaches disk

Posted Sep 9, 2011 5:44 UTC (Fri) by **pr1268** (subscriber, #24648) [Link]

Fascinating and enlightening article. I didn't previously know much difference between what `fflush()` and `fsync()` do. Thanks, Jeff!

I have a general question about data syncing: In your sample function code below, assuming it were a `main()`, and lines 23 and 27 (and their sanity checks) were removed, would normal termination of the program cause the data to reach the disk? I'm certain from my man pages travels that an implied library-buffer flush would occur, but would the kernel buffer(s) be sync'ed? Thanks.

P.S. You forgot to `free(buf);` (and shame on you for leaking memory!). :-)

Reply to this comment

s/below/above/

Posted Sep 9, 2011 9:24 UTC (Fri) by **pr1268** (subscriber, #24648) [Link]

s/below/above/

In my defense, your article (and sample code) were *below* the comment editor on my Web browser screen.

Reply to this comment

Ensuring data reaches disk

Posted Sep 9, 2011 14:31 UTC (Fri) by **phro** (subscriber, #29295) [Link]

If the fflush() and fsync() calls were removed, normal termination of the program would typically flush stdio streams (see the exit(3) man page for caveats). The data would still be buffered by the kernel's page cache.

Thanks for pointing out the memory leak. ;-)

Reply to this comment

Ensuring data reaches disk

Posted Sep 9, 2011 8:23 UTC (Fri) by **k8to** (subscriber, #15413) [Link]

The problem with fsync() is that while it may be necessary to, for example, ensure that your newly created file actually gets contents on disk before you overwrite the origina (the update pattern), there is no guarantee that this has acceptable performance implications.

Say you are updating the state of a system frequently, eg every 30 seconds, and you wish to create a new file representing this new state, and replace the previous one. You can call open(tempname), write(tmpfile), fsync(tmpfile), close(tmpfile), rename(), fsync(dir) and be sure that this data has landed as it shouldat the end of this string of actions, and that your window of data incoherency is at least limited to the time between the rename and the second fsync, if there is any incoherency at all.

Unfortunately, for many filesystems, the first fsync will cause an I/O storm, pushing out a very larger amount of writeback cached data, because the filesystem is not designed for partial synchronization to disk. Similarly the second fsync may cause an I/O storm.

You can continue working, in an awkward fashion, by leaving this blocking call in a thread, but you cannot avoid the fact that the fsync() calls may cause you to be no longer able to meet reasonable workload targets.

If you have this combination of performance sensitivity and correctness, you have to go down the ugly path of specialcasing on a filesystem and platform basis.

Reply to this comment

Ensuring data reaches disk

Posted Sep 9, 2011 9:10 UTC (Fri) by **trasz** (guest, #45786) [Link]

It might be worth mentioning that ZFS has an interesting solution for that: ZIL. Basically, it's a small log intended just for operations that need to be synced to disk; since it's a log, the writes are fast, and it's possible to move it to a separate log device.

Reply to this comment

Ensuring data reaches disk
Posted Sep 9, 2011 15:59 UTC (Fri) by **k8to** (subscriber, #15413) [Link]

Yep, on some systems it's as cheap as you could hope!

Reply to this comment

Ensuring data reaches disk
Posted Sep 23, 2011 20:56 UTC (Fri) by **oak** (guest, #2786) [Link]

How much using fdatasync() instead of fsync() would help?

Reply to this comment

Ensuring data reaches disk
Posted Sep 9, 2011 8:56 UTC (Fri) by **jvoss2** (subscriber, #7065) [Link]

The "if (ret =< 0) {" line probably should read "if (ret < 0) {"?

Reply to this comment

Ensuring data reaches disk
Posted Sep 9, 2011 19:00 UTC (Fri) by **valyala** (guest, #41196) [Link]

No. read(2) returns zero on the end of file. In our case this means that the incoming connection has been closed. The code just returns success (0) in this case, while not all the requested data has been read. The drawback of this design decision is that the caller of sock_read() cannot determine this case. It would be better returning the number of actual bytes written to the output file instead of just 'success/failure' code.

Reply to this comment

Ensuring data reaches disk
Posted Sep 9, 2011 14:23 UTC (Fri) by **baruchb** (subscriber, #6054) [Link]

Where is the "sync-samples.h" file #included in the sample code? Is it actually needed?

Reply to this comment

How disappointing
Posted Sep 9, 2011 16:26 UTC (Fri) by **sionescu** (subscriber, #59410) [Link]

Articles like this make me think again that the Unix-Haters handbook is still as valid as ever, maybe it should be updated.

Better OSes allow one to write code like (the equivalent of) this:

fd = open(path, O_WRONLY | O_REPLACE);
...

close(fd, CLOSE_COMMIT);

which atomically replaces the file's data - keeping intact its metadata(perms, xattrs, selinux context, etc...)

Again, how disappointing...

Reply to this comment

How disappointing

Posted Sep 9, 2011 21:15 UTC (Fri) by **butlerm** (subscriber, #13312) [Link]

Some of those better OS's have to be rebooted on a regular basis because an in-use binary file can't be replaced, either. So unless your filesystem implements multiversion read concurrency, locking up everything until the file is closed is more trouble than it is worth.

Reply to this comment

How disappointing

Posted Sep 9, 2011 22:45 UTC (Fri) by **sionescu** (subscriber, #59410) [Link]

Who said anything about locking ?

Reply to this comment

How disappointing

Posted Sep 11, 2011 0:12 UTC (Sun) by **butlerm** (subscriber, #13312) [Link]

>Who said anything about locking ?

I mention locking because it is the most common way to implement atomic commit semantics, from the perspective of all other processes. Your idea makes great sense as long as you have multiversion read concurrency, so that existing openers can see an old, read only version of the file indefinitely.

POSIX simply has a different solution for that, as I am sure you know - the name / inode distinction, which allows you to delete a file, or rename replace it with a new version without locking other processes out, waiting, or disturbing existing openers.

It is unfortunate of course that there is no standard call to clone an existing file's extended attributes and security context for use in a rename replace transaction - perhaps one should be added, it would be a worthwhile enhancement. Hating UNIX when it is vastly superior to the most widely distributed alternative in this respect seems a bit pointless to me.

Reply to this comment

How disappointing

Posted Sep 11, 2011 16:00 UTC (Sun) by **sionescu** (subscriber, #59410) [Link]

No, it's the common way of implementing atomic commit when *modifying* the data, but it's not what I have in mind, which is this:

open(path, O_REPLACE) only allocates a new inode

close(fd, CLOSE_COMMIT) atomically replaces the reference to the old inode with the new inode(just like rename) copying all metadata except for the (a|c|m)time, then calls fsync()

easy, isn't it ?

Reply to this comment

How disappointing

Posted Sep 11, 2011 20:45 UTC (Sun) by **nix** (subscriber, #2304) [Link]

Sure. Practicalities: you could do it to open() (though you'd have to get the change into POSIX before Ulrich would let it past), but you could never do that to close() without breaking every C program ever written. You could call it close_replace(), perhaps?

Reply to this comment

How disappointing

Posted Sep 11, 2011 21:10 UTC (Sun) by **sionescu** (subscriber, #59410) [Link]

Why POSIX ? There are other Linux-specific open flags, did Ulrich object to every one of them ?

The new syscall could be called close2, adding a "flags" parameter - in the spirit of accept4() et al.

Reply to this comment

How disappointing

Posted Sep 11, 2011 21:52 UTC (Sun) by **nix** (subscriber, #2304) [Link]

True, though close2() is a horrible name (as is wait$num() and accept$num()): give it a name that reflects its purpose.

Reply to this comment

How disappointing

Posted Sep 11, 2011 22:14 UTC (Sun) by **sionescu** (subscriber,

#59410) [Link]

How about "close_with_flags" ?

Reply to this comment

How disappointing

Posted Sep 11, 2011 23:51 UTC (Sun) by **nix** (subscriber, #2304) [Link]

Again, ugh ('with'?). I'd simply say close_replace(), no need for a flag or indeed any parameters at all. This means it has the same prototype as close(), so if anyone wants to choose between calling close() or close_replace() at runtime, they can just use a function pointer.

Reply to this comment

How disappointing

Posted Sep 23, 2011 0:59 UTC (Fri) by **spitzak** (guest, #4593) [Link]

If it is opened with the atomic-replace semantic, I would just have plain close() do the replacement.

There may be a need to somehow "abort" the file so that it is as though you never started writing it. But it may be sufficient to do this if the process owning the fd exits without calling close().

I very much disagree with others that say POSIX should be followed. The suggested method of writing a file is what is wanted in probably 95% of the time that files are written. It should be the basic operation, while "dynamic other processes can see the blocks change as I write them" is an extremely rare operation that should be the one requiring complex hacks.

Reply to this comment

How disappointing

Posted Nov 8, 2020 23:05 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

And then what happens if the file is hardlinked, and you want to modify the original file, not make a modified copy...

The trouble with POSIX is it is based on Unix and, sorry guys, Unix is crap as a commercial OS. It won because it was cheap and good enough.

And I curse it regularly because, unlike a lot of people today, I've

actually had experience of real commercial OSs. Trouble is, they've died because they cost too much to maintain :-(

(Mind you, I've used real commercial OSs that had those flags to do fancy file-system stuff, and when they have bugs they really do have bugs ...)

Cheers,
Wol

Reply to this comment

How disappointing

Posted Nov 9, 2020 7:52 UTC (Mon) by **jem** (subscriber, #24231) [Link]

“The trouble with POSIX is it is based on Unix and, sorry guys, Unix is crap as a commercial OS. It won because it was cheap and good enough.”

No, commercial Unix lost to Windows because *Windows* was cheap and good enough. Only 99 dollars!. (This was not a real ad, though.)

I also don't buy the argument that Unix cost more to maintain *per user*. Back then, Unix was a multi-user operating system that was centrally administered. Then came DOS and Windows and every user had their individual problems.

Reply to this comment

How disappointing

Posted Nov 9, 2020 10:11 UTC (Mon) by **Wol** (subscriber, #4433) [Link]

Except I was comparing Unix against proprietary OS's from the likes of DEC, Honeywell, Pr1me, etc.

While Unix was eating the mini-computers' lunch, yes, Windows came along and started eating its lunch ...

Cheers,
Wol

Reply to this comment

License of example files

Posted Sep 9, 2011 20:05 UTC (Fri) by **chrish** (subscriber, #351) [Link]

This is a really useful article. However, the code examples in the appendix would be a lot more useful if they were available under a liberal license (BSD 3-clause, or even public domain) rather than GPL v3 or greater. I guess the latter is the default license for Red Hat (and that's great), but it's not the most useful license for short code examples.

Reply to this comment

License of example files

Posted Sep 9, 2011 20:15 UTC (Fri) by **phro** (subscriber, #29295) [Link]

Yeah, I thought of that. I'll see about posting the examples elsewhere with a less restrictive license.

Reply to this comment

License of example files

Posted Nov 8, 2020 23:09 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

How about "these examples are too simple to be worthy of copyright" ... ?

The law itself sets out a vague line, so just declare that this stuff falls the wrong side of the line.

Cheers,
Wol

Reply to this comment

code feedback ...

Posted Sep 10, 2011 21:02 UTC (Sat) by **vapier** (subscriber, #15768) [Link]

```
 5      char *buf = malloc(MY_BUF_SIZE);
```

you forgot to free(buf) after the while loop and in the early returns inside of the loop. might as well just use the stack: char buf[MY_BUF_SIZE];

```
11              ret = read(sockfd, buf, MY_BUF_SIZE);
```

common mistake. the len should be min(MY_BUF_SIZE, nrbytes - written). otherwise, if (nrbytes % MY_BUF_SIZE) is non-zero, you read too many bytes from the sockfd and they get lost.

```
12              if (ret =< 0) {
```

typo ... should be "<=" as "=<" doesn't compile.

```
18              ret = fwrite((void *)buf, ret, 1, outfp);
19              if (ret != 1)
```

unless you build this with a C++ compiler, that cast is not needed. and another common mistake: the size/nmemb args are swapped ... the size is "1" (since sizeof(*buf) is 1 (a char)), and the number of elements is "ret". once you fix the arg order, the

method of clobbering the value of ret won't work in the "if" check ...

```
27      ret = fsync(fileno(outfp));
28      if (ret < 0)
29              return -1;
30      return 0;
```

at this point, you could just as easily write:

```
    return fsync(fileno(outfp));
```

Reply to this comment

code feedback ...

Posted Sep 13, 2011 6:00 UTC (Tue) by **jzbiciak** (guest, #5246) [Link]

> *another common mistake: the size/nmemb args are swapped ... the size is "1" (since sizeof(*buf) is 1 (a char)), and the number of elements is "ret". once you fix the arg order, the method of clobbering the value of ret won't work in the "if" check ...*

I don't think it's an error. In the example, if `fwrite` returns anything other than '1', then it reports an error. This is an "all-or-nothing" `fwrite`. If it fails, 'ret' will be 0, otherwise it will be 1. The semantic is "write 1 buffer of size 'ret' bytes."

I see nothing wrong with this, and it matches the `if (ret != 1)` statement that follows. Sure, you don't get to find out how many bytes did get written, but the code wasn't interested in that anyway. And, it's one less variable that's "live across call," so the resulting compiler output may be fractionally smaller/faster. (While I can think of smaller microoptimizations, this type of microoptimization *is* pretty far down the list, I must admit.)

Personally, I think the code might be clearer breaking 'ret' up into multiple variables. For example, if you did switch size/nmemb, you might rewrite the loop like so:

```
      while (tot_written < nrbytes) {
              int remaining = nrbytes - tot_written;
              int to_read   = remaining > MY_BUF_SIZE ? MY_BUF_SIZE : remaining;

              read_ret = read(sockfd, buf, to_read);
              if (read_ret <= 0) {
                      if (errno == EINTR)
                              continue;
                      return read_ret;
              }
              write_ret = fwrite((void *)buf, 1, read_ret, outfp);
              tot_written += write_ret;
              if (write_ret != read_ret)
                      return ferror(outfp);
      }
```

Written that way, you could easily add a way to return how many bytes *did* get written.

Also, the return value is inconsistent. I think "`return ferror(outfp)`" is wrong. `ferror` returns non-zero on an error, but it isn't guaranteed to be negative. The other paths

through this function return positive values on success, so shouldn't it be simply "`return -1;`" to match the `read` error path (which also simply returns -1, and maybe should be written as such)? ie:

```
while (tot_written < nrbytes) {
        int remaining = nrbytes - tot_written;
        int to_read   = remaining > MY_BUF_SIZE ? MY_BUF_SIZE : remaining;

        read_ret = read(sockfd, buf, to_read);
        if (read_ret <= 0) {
                if (errno == EINTR)
                        continue;
                return -1;
        }
        write_ret = fwrite((void *)buf, 1, read_ret, outfp);
        tot_written += write_ret;
        if (write_ret != read_ret)
                return -1;
}
```

Reply to this comment

code feedback ...

Posted Sep 13, 2011 6:08 UTC (Tue) by **jzbiciak** (guest, #5246) [Link]

Err... I guess the `read` error path returns -1 *or* 0, which again I think may be an error, unless you wanted to return 0 when the connection drops before "`nrbytes`" gets read. Oops.

That raises a different question: If you exit early due to the socket dropping, you won't `fflush/fsync`. Seems like you want a '`break`' if `read` returned 0 and `errno != EINTR`, don't you?

Reply to this comment

code feedback ...

Posted Sep 14, 2011 5:33 UTC (Wed) by **vapier** (subscriber, #15768) [Link]

in the past i've been bitten where fwrite was given a char* and size==num bytes to write and nmemb==1 (like in the example here). but perhaps that was a bug in the lower layers (it was a ppc/glibc setup). i do know that size==sizeof(*buf) has always worked for me ;).

Reply to this comment

code feedback ...

Posted Sep 14, 2011 11:59 UTC (Wed) by **jwakely** (subscriber, #60262) [Link]

the cast isn't needed with a C++ compiler either

Reply to this comment

code feedback ...

Posted Sep 16, 2011 13:43 UTC (Fri) by **renox** (subscriber, #23785) [Link]

> you forgot to free(buf) after the while loop and in the early returns inside of the loop. might as well just use the stack: char buf[MY_BUF_SIZE];

Is-it such a good idea?
I though that it was better to keep the stack small.

Reply to this comment

code feedback ...

Posted Sep 16, 2011 19:17 UTC (Fri) by **bronson** (subscriber, #4806) [Link]

That was painfully true 20 years ago. With modern memory management it doesn't matter much. Within reason of course -- a 20 MB buffer should probably still go on the heap.

Reply to this comment

code feedback ...

Posted Jan 25, 2012 20:34 UTC (Wed) by **droundy** (subscriber, #4559) [Link]

I prefer to avoid putting buffers on the stack simply to reduce the difficulties associated with buffer overflows. Not for security reasons (most of my programming is scientific), but simply so overwriting the buffer won't trash the stack, making debugging harder. And also to allow valgrind to immediately recognize a buffer overwrite...

Reply to this comment

Ensuring data reaches disk

Posted Sep 16, 2011 9:16 UTC (Fri) by **scheck** (guest, #4447) [Link]

"Recall that the storage may itself store the data in a write-back cache, so fsync() is still required for files opened with O_DIRECT in order to save the data to stable storage."

Why should I use fsync() for files opened with O_DIRECT and why has the storage device's cache anything to do with it?

Apart from that a very nice and comprehensible article. Thank you.

Reply to this comment

Ensuring data reaches disk

Posted Sep 16, 2011 10:35 UTC (Fri) by **andresfreund** (subscriber, #69562) [Link]

If the storage device has a write cache but no independent power supply you have the problem that you will loose data on power loss because O_DIRECT will only

guarantee that the write reaches the device, not that it reaches persistent storage inside that device.
For that you need to issue some special commands - which e.g. fsync() knows how to do.
Besides an O_DIRECT write doesn't guarantee that metadata updates have reached stable storage.

Reply to this comment

Ensuring data reaches disk

Posted Nov 8, 2020 21:55 UTC (Sun) by **yzou93** (guest, #142976) [Link]

An awesome article.
My question about fsync() is how the OS could control/know the device-internal caching behavior.
When designing a block device hardware, for example if Samsung wants to design a new SSD, is a cache control support for fsync() command issued from OS required?

Thank you.

Reply to this comment

Ensuring data reaches disk

Posted Nov 8, 2020 23:15 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

Not that this is necessarily the way it's done, but linux does have a disk database. I see that often enough watching the raid stuff. It's quite possible (though probably not) that linux looks up the drive characteristics.

Cheers,
Wol

Reply to this comment

Ensuring data reaches disk

Posted Nov 9, 2020 9:56 UTC (Mon) by **farnz** (subscriber, #17727) [Link]

Yes, such a command is needed, and the various interface specs (ATA, SCSI, NVMe) all have standardised commands for flushing the cache.

At a minimum, you get a FLUSH CACHE or SYNCHRONIZE CACHE type command, which is specified as not completing until all data in the cache is in persistent storage; this is enough to implement fsync() behaviour; beyond that, you can also have forced unit access (FUA) commands, which do not complete until the data written is on the persistent media, and even partial flush commands that only affect some sections of the drive.

There's an added layer of complexity in that some standards have queued flushes which act as straight barriers (all commands before the flush complete, then the

flush happens, then the rest of the queue); others have queued flushes that only affect commands issued before the flush in this queue (and can over-flush by flushing data from later commands in the queue), and yet others only have unqueued flushes which require you to idle the interface, wait for the flush to complete, and then resume issuing commands.

Reply to this comment

Ensuring data reaches disk

Posted Nov 9, 2020 10:17 UTC (Mon) by **Wol** (subscriber, #4433) [Link]

Ouch. As a database guy I'm desperate for "queued flush straight barrier", because if you want data integrity that at least makes reasoning possible - "if the transaction log is incomplete, revert; if the data write is complete, continue; if the log is complete and the data write isn't, re-play the log".

If you can't be sure what has or hasn't hit the disk - the nightmare scenario is "part of the log, and part of the data" - then you get the hoops that I believe SQLite and PostgreSQL go through :-(

Cheers,
Wol

Reply to this comment

Ensuring data reaches disk

Posted Nov 9, 2020 17:11 UTC (Mon) by **zlynx** (subscriber, #2285) [Link]

More fun is consumer grade SSDs that protect their metadata during power-loss but not necessarily the data.

I had to rebuild a btrfs volume because my laptop battery ran down in the bag and on reboot the drive contained blocks saying writes had completed, but those data blocks had old data in them. In other words, data that had been committed to physical storage (or that was CLAIMED by the drive) was no longer present after power-loss. It probably had to fsck or equivalent on the Flash FTL and lost some bits.

btrfs gets very upset about that.

I guess this behavior is still better than some older SSDs which had to be secure-erased and reformatted after losing their entire FTL? I guess.

Reply to this comment

Ensuring data reaches disk

Posted Nov 9, 2020 18:27 UTC (Mon) by **farnz** (subscriber, #17727) [Link]

> To be fair to btrfs, that's it's USP compared to ext4 - when hardware fails, it lets you know that your data has been eaten at the time of the issue, and not months down the line.
>
> And knowing consumer hardware, chances are very high that it did commit everything properly, and then had a catastrophic failure when there was a surprise power-down. Unfortunately, unless you have an acceptance lab verifying that kit complies with the intent of the spec, it often complies with the letter of the spec (if you're lucky) and no more :-(

Reply to this comment

Opening device nodes

Posted Sep 16, 2011 22:50 UTC (Fri) by **bjencks** (subscriber, #80303) [Link]

What are the semantics when you open a device node (either block, e.g. disk, or char, e.g. tape)? Does the kernel ever use page cache for device files? Does O_DIRECT do anything? What about O_SYNC? Does fsync always generate a barrier?

Also, what about the different disk abstraction layers (LVM, dm-crypt, MD RAID, DRBD, etc) -- what's involved in passing an fsync() all the way down the stack?

Reply to this comment

avoiding orphan files

Posted Sep 19, 2011 12:06 UTC (Mon) by **aeriksson** (guest, #73840) [Link]

The upshot of using the write/sync/rename workflow is of course that the original file is left untouched until there is a fully comitted replacment ready on disk. The downside is that you need to create a temporary file with a temporary filename while doing it. This is bad for crash recovery, where you'd leave orphan files on the filesystem.

Is there a way to sole that which I have overlooked?

```
fd=open("",O_UNNAMED);
....
rename_unnamed(fd,"/some/file");
```

Reply to this comment

avoiding orphan files

Posted Sep 19, 2011 17:15 UTC (Mon) by **mathstuf** (subscriber, #69389) [Link]

As someone who uses symlinks to manage dotfiles in another repository, the write/sync/rename workflow is annoying as all hell. Tin (since moved to slrn), gpg, pidgin, weechat (which is why I'm still using irssi), and more all force me to manually copy the file to the real location and remake the symlink. If there was a way to do the workflow and then replay the writes to an fd returned by open() on the original path,

it would be much better. So far, it looks as if all of the above solutions still fail for me.

Also in the write/sync/rename workflow, what happens if the temp file is on a separate filesystem? There's a copy involved there, so there is a time when the file is not atomically replaced (unless I'm missing some guarantee by POSIX in this case).

Reply to this comment

avoiding orphan files

Posted Sep 19, 2011 18:17 UTC (Mon) by **nybble41** (subscriber, #55106) [Link]

> Also in the write/sync/rename workflow, what happens if the temp file is on a separate filesystem?

In the write/sync/rename workflow, this is never supposed to occur. The temp file must always be on the same filesystem as the real file for the atomic-rename guarantee to apply.

Naturally, this can be extremely difficult to achieve in some cases. The file may be a symlink, which must be fully resolved to a symlink-free path to determine the real filesystem. The file may be the target of a bind mount, in which case I doubt there is any portable way to determine which filesystem it came from. And there there's the possibility that you can write to the file, but not the directory _containing_ the file...

The write/sync/rename process is hardly an ideal way to implement atomic replacement semantics. There are simply too many potential points of failure.

Reply to this comment

avoiding orphan files

Posted Jan 25, 2012 20:38 UTC (Wed) by **droundy** (subscriber, #4559) [Link]

> The write/sync/rename process is hardly an ideal way to implement atomic replacement semantics. There are simply too many potential points of failure.

True, but it's also the only one we've got, right?

Reply to this comment

avoiding orphan files

Posted Nov 8, 2020 23:17 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

Except it doesn't work - see my comment about hard-linked files ...

Cheers,
Wol

Reply to this comment

Direct Reads

Posted Apr 21, 2013 3:16 UTC (Sun) by **nikm** (guest, #90499) [Link]

I am interesting if there is a way to ensure that when you are reading data, these data is directly from the disk. As I wonder in case of reading even if the DIRECT_IO flag is used the data propably is read from kerner buffer. Is that correct?

thanx

Reply to this comment

Direct Reads

Posted Apr 22, 2013 11:37 UTC (Mon) by **etienne** (guest, #25256) [Link]

> ... data read from disk and not kernel buffers ...

Maybe echo 1, 2 or 3 to /proc/sys/vm/drop_caches ?

Reply to this comment

Ensuring data reaches disk

Posted Dec 1, 2014 13:47 UTC (Mon) by **ppai** (guest, #100047) [Link]

I'm wondering about this case:
Directory "a/b/c" already exists.

create("/tmp/whatever")
write("/tmp/whatever")
fsync("/tmp/whatever")
os.makedirs("a/b/c/d/e")
rename("/tmp/whatever", "a/b/c/d/e/obj.data")
fsync("a/b/c/d/e/")

Is it really required to fsync dirs all the way from e to a ? Fsync is totally not necessary for "a/b/c" as it already existed. But after doing a makedirs(), there's no way to know which subtree of "a/b/c/d/e" needs fsync().
Is it reasonable to fsync only the containing directory and expect the filesystem to take care of the rest (to make sure the entire tree makes it to disk) ?

Reply to this comment