I haven't used a desktop email client in years. None of them could handle the volume of email I get without at least occasionally corrupting my mailbox. Pine, Eudora, and outlook have all corrupted my inbox, forcing me to restore from backup. How is it that desktop mail clients are less reliable than gmail, even though my gmail account not only handles more email than I ever had on desktop clients, but also allows simultaneous access from multiple locations across the globe? Distributed systems have an unfair advantage, in that they can be robust against total disk failure in a way that desktop clients can't, but none of the file corruption issues I've had have been from total disk failure. Why has my experience with desktop applications been so bad?

Well, what sort of failures can occur? Crash consistency (maintaining consistent state even if there's a crash) is probably the easiest property to consider, since we can assume that everything, from the filesystem to the disk, works correctly; let's consider that first.

Crash Consistency

Pillai et al. had a paper and presentation at OSDI '14 on exactly how hard it is to save data without corruption or data loss.

Let's look at a simple example of what it takes to save data in a way that's robust against a crash. Say we have a file that contains the text `a foo` and we want to update the file to contain `a bar`. The pwrite function looks like it's designed for this exact thing. It takes a file descriptor, what we want to write, a length, and an offset. So we might try

```
pwrite([file], "bar", 3, 2)  // write 3 bytes at offset 2
```

What happens? If nothing goes wrong, the file will contain `a bar`, but if there's a crash during the write, we could get `a boo`, `a far`, or any other combination. Note that you may want to consider this an example over sectors or blocks and not chars/bytes.

If we want atomicity (so we either end up with `a foo` or `a bar` but nothing in between) one standard technique is to

make a copy of the data we're about to change in an undo log file, modify the "real" file, and then delete the log file. If a crash happens, we can recover from the log. We might write something like

```
creat(/dir/log);
write(/dir/log, "2,3,foo", 7);
pwrite(/dir/orig, "bar", 3, 2);
unlink(/dir/log);
```

This should allow recovery from a crash without data corruption via the undo log, at least if we're using `ext3` and we made sure to mount our drive with `data=journal`. But we're out of luck if, like most people, we're using the default[1] – with the default `data=ordered`, the `write` and `pwrite` syscalls can be reordered, causing the write to `orig` to happen before the write to the log, which defeats the purpose of having a log. We can fix that.

```
creat(/dir/log);
write(/dir/log, "2, 3, foo");
fsync(/dir/log);  // don't allow write to be reordered past pwrite
pwrite(/dir/orig, 2, "bar");
fsync(/dir/orig);
unlink(/dir/log);
```

That should force things to occur in the correct order, at least if we're using ext3 with `data=journal` or `data=ordered`. If we're using `data=writeback`, a crash during the the `write` or `fsync` to log can leave `log` in a state where the filesize has been adjusted for the write of "bar", but the data hasn't been written, which means that the log will contain random garbage. This is because with `data=writeback`, metadata is journaled, but data operations aren't, which means that data operations (like writing data to a file) aren't ordered with respect to metadata operations (like adjusting the size of a file for a write).

We can fix that by adding a checksum to the log file when creating it. If the contents of `log` don't contain a valid checksum, then we'll know that we ran into the situation described above.

```
creat(/dir/log);
write(/dir/log, "2, 3, [checksum], foo");  // add checksum to log file
fsync(/dir/log);
```

```
pwrite(/dir/orig, 2, “bar”);
fsync(/dir/orig);
unlink(/dir/log);
```

That’s safe, at least on current configurations of ext3. But it’s legal for a filesystem to end up in a state where the log is never created unless we issue an fsync to the parent directory.

```
creat(/dir/log);
write(/dir/log, “2, 3, [checksum], foo”);
fsync(/dir/log);
fsync(/dir);  // fsync parent directory of log file
pwrite(/dir/orig, 2, “bar”);
fsync(/dir/orig);
unlink(/dir/log);
```

That should prevent corruption on any Linux filesystem, but if we want to make sure that the file actually contains “bar”, we need another fsync at the end.

```
creat(/dir/log);
write(/dir/log, “2, 3, [checksum], foo”);
fsync(/dir/log);
fsync(/dir);
pwrite(/dir/orig, 2, “bar”);
fsync(/dir/orig);
unlink(/dir/log);
fsync(/dir);
```

That results in consistent behavior and guarantees that our operation actually modifies the file after it’s completed, as long as we assume that `fsync` actually flushes to disk. OS X and some versions of ext3 have an fsync that doesn’t really flush to disk. OS X requires `fcntl(F_FULLFSYNC)` to flush to disk, and some versions of ext3 only flush to disk if the the inode changed (which would only happen at most once a second on writes to the same file, since the inode mtime has one second granularity), as an optimization.

Even if we assume fsync issues a flush command to the disk, some disks ignore flush directives for the same reason

fsync is gimped on OS X and some versions of ext3 – to look better in benchmarks. Handling that is beyond the scope of this post, but the Rajimwale et al. DSN '11 paper and related work cover that issue.

Filesystem semantics

When the authors examined ext2, ext3, ext4, btrfs, and xfs, they found that there are substantial differences in how code has to be written to preserve consistency. They wrote a tool that collects block-level filesystem traces, and used that to determine which properties don't hold for specific filesystems. The authors are careful to note that they can only determine when properties don't hold – if they don't find a violation of a property, that's not a guarantee that the property holds.

File system configuration		Atomicity				Ordering			
		One sector overwrite	*Append content*	*Many sector overwrite*	*Directory operation*	*Overwrite → Any op*	*Append → Any op*	*Dir-op → Any op*	*Append → Rename*
ext2	*async*		✘	✘	✘	✘	✘	✘	✘
	sync		✘	✘	✘				
ext3	*writeback*		✘	✘		✘	✘		✘
	ordered			✘		✘			
	data-journal			✘					
ext4	*writeback*		✘	✘		✘	✘		✘
	ordered			✘		✘	✘		
	no-delalloc			✘		✘			
	data-journal			✘					
btrfs				✘			✘	✘	
xfs	*default*			✘		✘	✘		
	wsync			✘		✘			

Xs indicate that a property is violated. The atomicity properties are basically what you'd expect, e.g., no X for single sector overwrite means that writing a single sector is atomic. The authors note that the atomicity of single sector overwrite sometimes comes from a property of the disks they're using, and that running these filesystems on some disks won't give you single sector atomicity. The ordering properties are also pretty much what you'd expect from their names, e.g., an X in the "Overwrite -> Any op" row means that an overwrite can be reordered with some operation.

After they created a tool to test filesystem properties, they then created a tool to check if any applications rely on any potentially incorrect filesystem properties. Because invariants are application specific, the authors wrote checkers for each application tested.

Application	Types									Unique static vulnerabilities
	Across-syscalls atomicity	Atomicity			Ordering			Durability		
		Appends and truncates	Single-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	Safe file flush	Other	
Leveldb1.10	1‡	1		1	2	1	3	1		**10**
Leveldb1.15	1	1		1	1		2			**6**
LMDB			1							**1**
GDBM	1			1			1		2	**5**
HSQLDB		1		2	1		3	2	1	**10**
Sqlite-Roll									1	**1**
Sqlite-WAL										**0**
PostgreSQL			1							**1**
Git	1			1	2	1	3		1	**9**
Mercurial	2	1		1		1	4		2	**10**
VMWare				1						**1**
HDFS				1			1			**2**
ZooKeeper			1				1	2		**4**
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3#*
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

Application	ext3-w	ext3-o	ext3-j	ext4-o	btrfs
Leveldb1.10	3	1	1	2	4
Leveldb1.15	2	1	1	2	3
LMDB					
GDBM	3	3	2	3	4
HSQLDB					4
Sqlite-Roll	1	1	1	1	1
Sqlite-WAL					
PostgreSQL					
Git	2	2	2	2	5
Mercurial	4	3	3	6	8
VMWare					
HDFS					1
ZooKeeper	1	1		1	1
Total	16	12	10	17	31

(c) Under Current File Systems.

	Ordering	DO	AG	CA
ext3-w	Dir ops and file-sizes ordered among themselves, before sync operations.	✓	4K	×
ext3-o	Dir ops, appends, truncates ordered among themselves. Overwrites before non-overwrites, all before sync.	✓	4K	✓
ext3-j	All operations are ordered.	✓	4K	✓
ext4-o	Safe rename, safe file flush, dir ops ordered among themselves	✓	4K	✓
btrfs	Safe rename, safe file flush	✓	4K	✓

The authors find issues with most of the applications tested, including things you'd really hope would work, like LevelDB, HDFS, Zookeeper, and git. In a talk, one of the authors noted that the developers of sqlite have a very deep understanding of these issues, but even that wasn't enough to prevent all bugs. That speaker also noted that version control systems were particularly bad about this, and that the developers had a pretty lax attitude that made it very easy for the authors to find a lot of issues in their tools. The most common class of error was incorrectly assuming ordering between syscalls. The next most common class of error was assuming that syscalls were atomic[2]. These are fundamentally the same issues people run into when doing multithreaded programming.

Correctly reasoning about re-ordering behavior and inserting barriers correctly is hard. But even though shared memory concurrency is considered a hard problem that requires great care, writing to files isn't treated the same way, even though it's actually harder in a number of ways.

Something to note here is that while btrfs's semantics aren't inherently less reliable than ext3/ext4, many more applications corrupt data on top of btrfs because developers aren't used to coding against filesystems that allow directory operations to be reordered (ext2 is perhaps the most recent widely used filesystem that allowed that reordering). We'll probably see a similar level of bug exposure when people start using NVRAM drives that have byte-level atomicity. People almost always just run some tests to see if things work, rather than making sure they're coding against what's legal in a POSIX filesystem.

Hardware memory ordering semantics are usually well documented in a way that makes it simple to determine precisely which operations can be reordered with which other operations, and which operations are atomic. By contrast, here's the ext manpage on its three data modes:

> journal: All data is committed into the journal prior to being written into the main filesystem. ordered: This is the default mode. All data is forced directly out to the main file system prior to its metadata being committed to the journal. writeback: Data ordering is not preserved – data may be written into the main filesystem after its metadata has been committed to the journal. **This is rumoured to be** the highest-throughput option. It guarantees internal filesystem integrity, however it can allow old data to appear in files after a crash and journal recovery.

The manpage literally refers to rumor. This is the level of documentation we have. If we look back at our example where we had to add an `fsync` between the `write(/dir/log, "2, 3, foo")` and `pwrite(/dir/orig, 2, "bar")` to prevent reordering, I don't think the necessity of the `fsync` is obvious from the description in the manpage. If you look at the hardware memory ordering "manpage" above, it specifically defines the ordering semantics, and it certainly doesn't rely on rumor.

This isn't to say that filesystem semantics aren't documented anywhere. Between lwn and LKML, it's possible to get a good picture of how things work. But digging through all of that is hard enough that it's still quite common for there

to be long, uncertain discussions on how things work. A lot of the information out there is wrong, and even when information was right at the time it was posted, it often goes out of date.

When digging through archives, I've often seen a post from 2005 cited to back up the claim that OS X `fsync` is the same as Linux `fsync`, and that OS X `fcntl(F_FULLFSYNC)` is even safer than anything available on Linux. Even at the time, I don't think that was true for the 2.4 kernel, although it was true for the 2.6 kernel. But since 2008 or so Linux 2.6 with ext3 will do a full flush to disk for each fsync (if the disk supports it, and the filesystem hasn't been specially configured with barriers off).

Another issue is that you often also see exchanges like this one:

Dev 1: Personally, I care about metadata consistency, and ext3 documentation suggests that journal protects its integrity. Except that it does not on broken storage devices, and you still need to run fsck there.
Dev 2: as the ext3 authors have stated many times over the years, you still need to run fsck periodically anyway.
Dev 1: Where is that documented?
Dev 2: linux-kernel mailing list archives.
Dev 3: Probably from some 6-8 years ago, in e-mail postings that I made.

Where's this documented? Oh, in some mailing list post 6-8 years ago (which makes it 12-14 years from today). I don't mean to pick on filesystem devs. The fs devs whose posts I've read are quite polite compared to LKML's reputation; they generously spend a lot of their time responding to basic questions and I'm impressed by how patient the expert fs devs are with askers, but it's hard for outsiders to troll through a decade and a half of mailing list postings to figure out which ones are still valid and which ones have been obsoleted!

In their OSDI 2014 talk, the authors of the paper we're discussing noted that when they reported bugs they'd found, developers would often respond "POSIX doesn't let filesystems do that", without being able to point to any specific POSIX documentation to support their statement. If you've followed Kyle Kingsbury's Jepsen work, this may sound familiar, except devs respond with "filesystems don't do that" instead of "networks don't do that". I think this is understandable, given how much misinformation is out there. Not being a filesystem dev myself, I'd be a bit surprised if I don't have at least one bug in this post.

Filesystem correctness

We've already encountered a lot of complexity in saving data correctly, and this only scratches the surface of what's involved. So far, we've assumed that the disk works properly, or at least that the filesystem is able to detect when the disk has an error via SMART or some other kind of monitoring. I'd always figured that was the case until I started looking into it, but that assumption turns out to be completely wrong.

The Prabhakaran et al. SOSP 05 paper examined how filesystems respond to disk errors in some detail. They created a fault injection layer that allowed them to inject disk faults and then ran things like `chdir`, `chroot`, `stat`, `open`, `write`, etc. to see what would happen.

Between ext3, reiserfs, and NTFS, reiserfs is the best at handling errors and it seems to be the only filesystem where errors were treated as first class citizens during design. It's mostly consistent about propagating errors to the user on reads, and calling `panic` on write failures, which triggers a restart and recovery. This general policy allows the filesystem to gracefully handle read failure and avoid data corruption on write failures. However, the authors found a number of inconsistencies and bugs. For example, reiserfs doesn't correctly handle read errors on indirect blocks and leaks space, and a specific type of write failure doesn't prevent reiserfs from updating the journal and committing the transaction, which can result in data corruption.

Reiserfs is the good case. The authors found that ext3 ignored write failures in most cases, and rendered the filesystem read-only in most cases for read failures. This seems like pretty much the opposite of the policy you'd want. Ignoring write failures can easily result in data corruption, and remounting the filesystem as read-only is a drastic overreaction if the read error was a transient error (transient errors are common). Additionally, ext3 did the least consistency checking of the three filesystems and was the most likely to not detect an error. In one presentation, one of the authors remarked that the ext3 code had lots of comments like "I really hope a write error doesn't happen here" in places where errors weren't handled.

NTFS is somewhere in between. The authors found that it has many consistency checks built in, and is pretty good about propagating errors to the user. However, like ext3, it ignores write failures.

The paper has much more detail on the exact failure modes, but the details are mostly of historical interest as many of the bugs have been fixed.

It would be really great to see an updated version of the paper, and in one presentation someone in the audience asked if there was more up to date information. The presenter replied that they'd be interested in knowing what things look like now, but that it's hard to do that kind of work in academia because grad students don't want to repeat work that's been done before, which is pretty reasonable given the incentives they face. Doing replications is a lot of work, often nearly as much work as the original paper, and replications usually give little to no academic credit. This is one of the many cases where the incentives align very poorly with producing real world impact.

The Gunawi et al. FAST 08 is another paper it would be great to see replicated today. That paper follows up the paper we just looked at, and examines the error handling code in different file systems, using a simple static analysis tool to find cases where errors are being thrown away. Being thrown away is defined very loosely in the paper — code like the following

```
if (error) {
    printk("I have no idea how to handle this error\n");
}
```

is considered *not* throwing away the error. Errors are considered to be ignored if the execution flow of the program doesn't depend on the error code returned from a function that returns an error code.

With that tool, they find that most filesystems drop a lot of error codes:

	By % Broken		By Viol/Kloc	
Rank	FS	Frac.	FS	Viol/Kloc

1	IBM JFS	24.4	ext3	7.2
2	ext3	22.1	IBM JFS	5.6
3	JFFS v2	15.7	NFS Client	3.6
4	NFS Client	12.9	VFS	2.9
5	CIFS	12.7	JFFS v2	2.2
6	MemMgmt	11.4	CIFS	2.1
7	ReiserFS	10.5	MemMgmt	2.0
8	VFS	8.4	ReiserFS	1.8
9	NTFS	8.1	XFS	1.4

10	XFS	6.9	NFS Server	1.2

Comments they found next to ignored errors include: “Should we pass any errors back?”, “Error, skip block and hope for the best.”, “There’s no way of reporting error returned from ext3_mark_inode_dirty() to user space. So ignore it.“, “Note: todo: log error handler.“, “We can’t do anything about an error here.”, “Just ignore errors at this point. There is nothing we can do except to try to keep going.”, “Retval ignored?”, and “Todo: handle failure.”

One thing to note is that in a lot of cases, ignoring an error is more of a symptom of an architectural issue than a bug per se (e.g., ext3 ignored write errors during checkpointing because it didn’t have any kind of recovery mechanism). But even so, the authors of the papers found many real bugs.

Error recovery

Every widely used filesystem has bugs that will cause problems on error conditions, which brings up two questions. Can recovery tools robustly fix errors, and how often do errors occur? How do they handle recovery from those problems? The Gunawi et al. OSDI 08 paper looks at that and finds that fsck, a standard utility for checking and repairing file systems, “checks and repairs certain pointers in an incorrect order … the file system can even be unmountable after”.

At this point, we know that it’s quite hard to write files in a way that ensures their robustness even when the underlying filesystem is correct, the underlying filesystem will have bugs, and that attempting to repair corruption to the filesystem may damage it further or destroy it. How often do errors happen?

Error frequency

The Bairavasundaram et al. SIGMETRICS ‘07 paper found that, depending on the exact model, between 5% and 20% of disks would have at least one error over a two year period. Interestingly, many of these were isolated errors –

38% of disks with errors had only a single error, and 80% had fewer than 50 errors. A follow-up study looked at corruption and found that silent data corruption that was only detected by checksumming happened on .5% of disks per year, with one extremely bad model showing corruption on 4% of disks in a year.

It's also worth noting that they found very high locality in error rates between disks on some models of disk. For example, there was one model of disk that had a very high error rate in one specific sector, making many forms of RAID nearly useless for redundancy.

That's another study it would be nice to see replicated. Most studies on disk focus on the failure rate of the entire disk, but if what you're worried about is data corruption, errors in non-failed disks are more worrying than disk failure, which is easy to detect and mitigate.

Conclusion

Files are hard. Butler Lampson has remarked that when they came up with threads, locks, and condition variables at PARC, they thought that they were creating a programming model that anyone could use, but that there's now decades of evidence that they were wrong. We've accumulated a lot of evidence that humans are very bad at reasoning about these kinds of problems, which are very similar to the problems you have when writing correct code to interact with current filesystems. Lampson suggests that the best known general purpose solution is to package up all of your parallelism into as small a box as possible and then have a wizard write the code in the box. Translated to filesystems, that's equivalent to saying that as an application developer, writing to files safely is hard enough that it should be done via some kind of library and/or database, not by directly making syscalls.

Sqlite is quite good in terms of reliability if you want a good default. However, some people find it to be too heavyweight if all they want is a file-based abstraction. What they really want is a sort of polyfill for the file abstraction that works on top of all filesystems without having to understand the differences between different configurations (and even different versions) of each filesystem. Since that doesn't exist yet, when no existing library is sufficient, you need to checksum your data since you will get silent errors and corruption. The only questions are whether or not you detect the errors and whether or not your record format only destroys a single record when corruption happens, or if it destroys the entire database. As far as I can tell, most desktop email client developers have chosen to go the route

of destroying all of your email if corruption happens.

These studies also hammer home the point that conventional testing isn't sufficient. There were multiple cases where the authors of a paper wrote a relatively simple tool and found a huge number of bugs. You don't need any deep computer science magic to write the tools. The error propagation checker from the paper that found a ton of bugs in filesystem error handling was 4k LOC. If you read the paper, you'll see that the authors observed that the tool had a very large number of shortcomings because of its simplicity, but despite those shortcomings, it was able to find a lot of real bugs. I wrote a vaguely similar tool at my last job to enforce some invariants, and it was literally two pages of code. It didn't even have a real parser (it just went line-by-line through files and did some regexp matching to detect the simple errors that it's possible to detect with just a state machine and regexes), but it found enough bugs that it paid for itself in development time the first time I ran it.

Almost every software project I've seen has a lot of low hanging testing fruit. Really basic random testing, static analysis, and fault injection can pay for themselves in terms of dev time pretty much the first time you use them.

Appendix

I've probably covered less than 20% of the material in the papers I've referred to here. Here's a bit of info about some other neat info you can find in those papers, and others.

Pillai et al., OSDI '14: this papers goes into much more detail about what's required for crash consistency than this post does. It also gives a fair amount of detail about how exactly applications fail, including diagrams of traces that indicate what false assumptions are embedded in each trace.

Chidambara et al., FAST '12: the same filesystem primitives are responsible for both consistency and ordering. The authors propose alternative primitives that separate these concerns, allow better performance while maintaining safety.

Rajimwale et al. DSN '01: you probably shouldn't use disks that ignore flush directives, but in case you do, here's a protocol that forces those disks to flush using normal filesystem operations. As you might expect, the performance

for this is quite bad.

Prabhakaran et al. SOSP '05: This has a lot more detail on filesystem responses to error than was covered in this post. The authors also discuss JFS, an IBM filesystem for AIX. Although it was designed for high reliability systems, it isn't particularly more reliable than the alternatives. Related material is covered further in DSN '08, StorageSS '06, DSN '06, FAST '08, and USENIX '09, among others.

Gunawi et al. FAST '08 : Again, much more detail than is covered in this post on when errors get dropped, and how they wrote their tools. They also have some call graphs that give you one rough measure of the complexity involved in a filesystem. The XFS call graph is particularly messy, and one of the authors noted in a presentation that an XFS developer said that XFS was fun to work on since they took advantage of every possible optimization opportunity regardless of how messy it made things.

Bairavasundaram et al. SIGMETRICS '07: There's a lot of information on disk error locality and disk error probability over time that isn't covered in this post. A followup paper in FAST08 has more details.

Gunawi et al. OSDI '08: This paper has a lot more detail about when fsck doesn't work. In a presentation, one of the authors mentioned that fsck is the only program that's ever insulted him. Apparently, if you have a corrupt pointer that points to a superblock, fsck destroys the superblock (possibly rendering the disk unmountable), tells you something like "you dummy, you must have run fsck on a mounted disk", and then gives up. In the paper, the authors reimplement basically all of fsck using a declarative model, and find that the declarative version is shorter, easier to understand, and much easier to extend, at the cost of being somewhat slower.

Memory errors are beyond the scope of this post, but memory corruption can cause disk corruption. This is especially annoying because memory corruption can cause you to take a checksum of bad data and write a bad checksum. It's also possible to corrupt in memory pointers, which often results in something very bad happening. See the Zhang et al. FAST '10 paper for more on how ZFS is affected by that. There's a meme going around that ZFS is safe against memory corruption because it checksums, but that paper found that critical things held in memory aren't checksummed, and that memory errors can cause data corruption in real scenarios.

The sqlite devs are serious about both documentation and testing. If I wanted to write a reliable desktop application, I'd start by reading the sqlite docs and then talking to some of the core devs. If I wanted to write a reliable distributed application I'd start by getting a job at Google and then reading the design docs and postmortems for GFS, Colossus, Spanner, etc. J/k, but not really.

We haven't looked at formal methods at all, but there have been a variety of attempts to formally verify properties of filesystems, such as SibylFS.

This list isn't intended to be exhaustive. It's just a list of things I've read that I think are interesting.

Update: many people have read this post and suggested that, in the first file example, you should use the much simpler protocol of copying the file to modified to a temp file, modifying the temp file, and then renaming the temp file to overwrite the original file. In fact, that's probably the most common comment I've gotten on this post. If you think this solves the problem, I'm going to ask you to pause for five seconds and consider the problems this might have. First, you still need to fsync in multiple places. Second, you will get very poor performance with large files. People have also suggested using many small files to work around that problem, but that will also give you very poor performance unless you do something fairly exotic. Third, if there's a hardlink, you've now made the problem of crash consistency much more complicated than in the original example. Fourth, you'll lose file metadata, sometimes in ways that can't be fixed up after the fact. That problem can, on some filesystems, be worked around with ioctls, but that only sometimes fixes the issue and now you've got fs specific code to preserve correctness even in the non-crash case. And that's just the beginning. The fact that so many people thought that this was a simple solution to the problem demonstrates that this problem is one that people are prone to underestimating, even they're explicitly warned that people tend to underestimate this problem!

If you liked this, you'll probably enjoy this post on CPU bugs.

Thanks to Leah Hanson, Katerina Barone-Adesi, Jamie Brandon, Kamal Marhubi, Joe Wilder, David Turner, Benjamin Gilbert, Tom Murphy, Chris Ball, Joe Doliner, Alexy Romanov, Mindy Preston, Paul McJones, and Evan Jones for comments/discussion.

---

1. Turns out some commercially supported distros only support `data=ordered`. Oh, and when I said `data=ordered`

was the default, that's only the case if pre-2.6.30. After 2.6.30, there's a config option, `CONFIG_EXT3_DEFAULTS_TO_ORDERED`. If that's not set, the default becomes `data=writeback`. [return]

2. Cases where overwrite atomicity is required were documented as known issues, and all such cases assumed single-block atomicity and not multi-block atomicity. By contrast, multiple applications (LevelDB, Mercurial, and HSQLDB) had bad data corruption bugs that came from assuming appends are atomic.

   That seems to be an indirect result of a commonly used update protocol, where modifications are logged via appends, and then logged data is written via overwrites. Application developers are careful to check for and handle errors in the actual data, but the errors in the log file are often overlooked.

   There are a number of other classes of errors discussed, and I recommend reading the paper for the details if you work on an application that writes files.

   [return]

← Big company vs. startup work and pay *Should I buy ECC memory? →*

Archive *Popular* *About (hire me!)* *Twitter* *RSS*