Dirty deals: the story of a data corruption bug in OpenZFS

Rob Norris,
Klara Inc.

***Abstract*—In November 2023 a silent data corruption bug was discovered in the recently released OpenZFS 2.2.0. The bug was reported, and quickly correlated by the community with a new feature, "block cloning", which even before its release had gained a reputation for instability and data loss, to the point of being disabled by default in FreeBSD 14. A 2.2.1 release was quickly put together, disabling the feature.**

One of OpenZFS' key selling points is its ability to detect and correct all kinds of data corruption, so news of its apparent failure quickly spread through tech news and social media, aided by its somewhat prickly relationship with some other parts of the open-source software world. This naturally led to a lot of confusion and concern from home and business OpenZFS users alike, worried about the integrity of their data.

With many outside eyes on the bug report, a reproduction test case was developed and it was quickly shown that the bug was present on the previous 2.1 series, and it was theorised that it might have been present right back to the original open-source ZFS release. This naturally caused even more confusion and concern, as now people were concerned about the integrity of their decades-old archive storage.

Over the Thanksgiving long weekend a few OpenZFS developers and many users rallied around the problem. A workaround was identified and documented. The bug was studied and gradually understood, and a fix was developed. People shared ideas and how might identify if their data was corrupt. Others theorised about different workloads that could make the problem easier or harder to hit. Still others took it upon themselves to find ways to run older versions, to find out just how far back the problem went.

This is a story about a 15-year old bug, where it came from, how it works, how it eluded users and developers alike through that time, and how a community pulled together to understand and fix it.

I. 1. Introduction

Late in November 2023 my wife was away on a long-anticipated holiday and I was home with my teenage children. I work from home anyway, so I'm fortunate enough that this didn't change my day-to-day routine very much, but did leave me with little to do in the evenings.

I'm also in Australia, where we don't have Thanksgiving, so the week leading up to this holiday was already quiet, as most of my customers and colleagues are based in North America and were preparing for a long weekend, so I had very few urgent demands on my time. I had plenty of work to be getting on with, and was rather looking forward to the opportunity to focus.

OpenZFS 2.2.0 had not long since been released with a new feature, "block cloning", which allows a file to be copied just by referencing the existing data in it rather than actually copying it. Even before its release it had gained a reputation for instability and had been disabled by default in FreeBSD 14. I had done some work to help bring block cloning to OpenZFS and so felt some responsibility for fixing the problems with it, as well as it just being a feature I am personally interested in. For both these reasons, I'd been keeping an eye on the issue tracker and helping out where I could with fixes and improvements.

A couple of weeks earlier, a bug report[1] had been posted, describing a workload that would write files and and read back from them at high speed in parallel, that was suddenly reading back all-zeros when it should have real data. A few people could reproduce it, and noticed it had only started happening since 2.2.0, and all reports had being with a newer version of `cp` that was able to invoke the block cloning behaviour, so it was easy to assume that this was yet another block cloning problem. Since this was easy to hit on fairly common workload, a 2.2.1 release was issued that required block cloning to be explicitly enabled.

A few people had started to look into the issue. A program was written that could reproduce it fairly reliably, and it was slowly being investigated. I had been keeping an eye on the progress and offering ideas where I could. I was checking in on the morning of Thursday 23 November to find that someone had run the reproducer program on a OpenZFS 2.1.x release, and found it failed there too. We quickly understood the implications:

- block cloning was likely *not* the fault
- OpenZFS had what looked like a data corruption bug widely deployed in the field

At this point, plenty of people had turned up, quite reasonably worried about the bug and their stored data. At this point I decided to abandon my plans for the day and see what I could do to help understand and fix the bug, and also to try to calm things down.

This paper describes the technical detail of the bug and history, and then presents and discusses some of the wider issues around the OpenZFS project and community that contributed to the problem and the difficulties resolving it.

II. A brief filesystem lesson

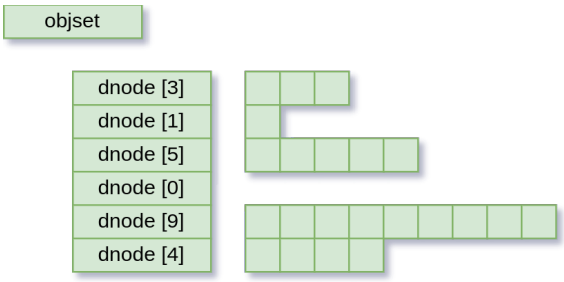
To understand the bug, you first need to know a little bit about how OpenZFS manages files internally.

A. What is a file, really?

If you lift the lid on an OpenZFS filesystem you'll find that OpenZFS is an object store. The top-level object is called an "object set" (or just "objset"), and can be thought of as just a giant array of objects. In turn, each object has two parts:

- the "dnode", which is a kind of metadata header that describes the object's type, size, etc, and carries a pointer to its first data block;
- the object data, which is a tree of blocks; either an "indirect" block that just points to more blocks, or a "data" block, which is the actual object data.

For the purposes of this discussion, we can consider a simplified view of the world, with the objset at the top, a list of dnodes, each with a count of attached data blocks, and then a list of data blocks:

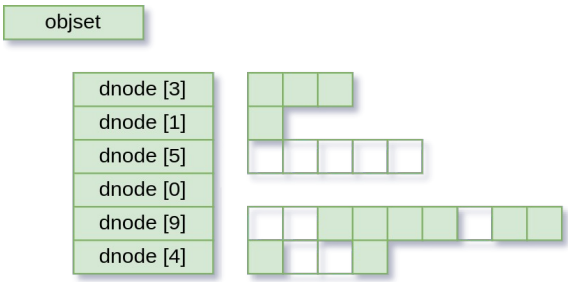


Higher-level filesystem concepts are just built out of more objects, for example, a directory is just a special kind of object that is just a dictionary of name → object index.

B. *Not all data is data*

If a data block contains all zeroes, we can use a little optimisation. Instead of allocating real space on the disk just to store a whole bunch of zeroes, we can instead keep a note to say "this block is all zeroes". We don't have to store anything then, because we can recreate it on request. This note is called a "hole", and is what gives a "sparse file" its "sparseness".

So here are our files again, with some holes:



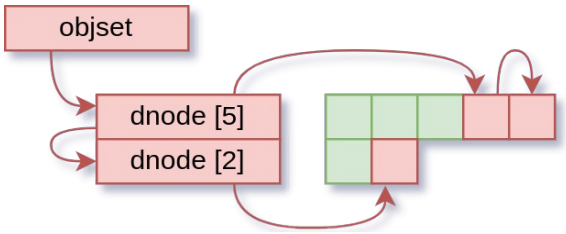
In every way, these look like they did before: the dnode still reports how many blocks they have. All that changes is that if something tries to read from one of the hole blocks, OpenZFS will create a data buffer as normal, but instead of actually going to disk and getting the data to put in the buffer, it will just zero the buffer. This requires less space on disk and fewer IO operations to "read" it.

C. *The fastest disks are memory*

Unfortunately, disks are slow, even very fast ones. Memory, however, is very fast. So, when you make a change to a file, for example, to write some new data to the end of it, OpenZFS makes that change in memory only, and later writes it (and any other changes) down to disk. This is fast, because it only touches memory, and can make the later disk part even faster, for example when your program may have written over part of a file, then written over the same part again. When it comes time to write the change to disk, it only has to write the final state, not both changes.

Of course, keeping changes in memory means we need extra housekeeping information to know that we have changes to write out when the time comes. We call the changed things "dirty", and to track this, we keep some lists.

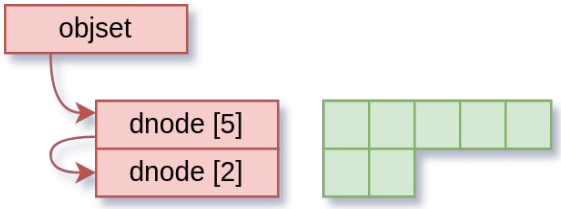
The objset has a list of dirty dnodes, and each dnode has a list of dirty blocks. So after our write, we have some nice linked lists of dirty things, ready to be written out.



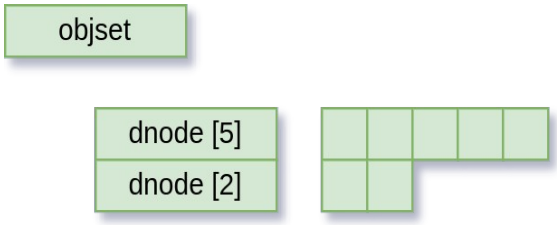
There is one crucially important detail, of course: until that data is written to disk, the version in memory is the latest, and what the application believes exists. So anything inside OpenZFS that wants to consider the current state of an object has to look at the dirty information as well, to know if it's looking at the latest state or not.

D. *Data first*

When it comes time to write everything out, first the blocks are written:



And then the dnodes are written, and everything is on disk.



As noted, this is a very simplified explanation, but is a useful conceptual model.

III. Beware of holes

The crux of this bug is around hole detection.

Sometimes an application wants to know where the holes in a file are. A file copying program is a good example: its whole job is to read all the data from one file, and write it to another. If it can find out that there's a hole, then it saves itself the trouble of reading that block, and it can also tell the filesystem to make a hole in the copy too. This allows even less space to be used, and even fewer IO operations to be issued.

For a program to find out about holes, it can call `lseek()`[2] with the `SEEK_HOLE` or `SEEK_DATA` options, which request the position of the next hole or the next data region, respectively

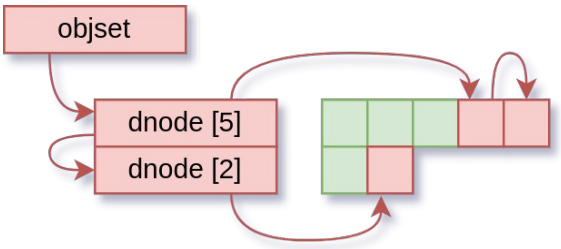
Holes are actually stored on disk in OpenZFS as a zero-length block pointer. There's also a clever optimisation, which is that if a program writes a data block of all-zeroes, the compression system will notice and "compress" it down to a hole. However, block pointers and compression belong to the IO layer, which is quite a way away from the object layer (the DMU).

As a result, there's no way currently to discover if a dirty block will eventually be stored as a hole. For this reason,

`lseek()` checks if the dnode is dirty, and if so, waits for it to be written out before it goes looking for holes.

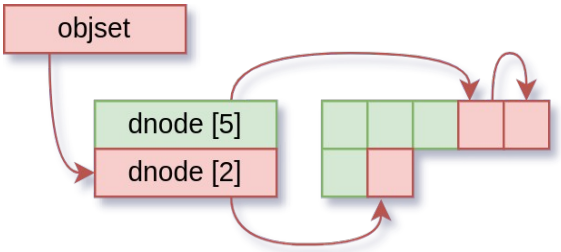
A. *Hidden dirt*

So now we come to the actual bug: the dirty check. Recall how we track dirty things:



Before we fixed the bugbug-fix, the dirty check for `lseek()` used to be "is this dnode on the dirty list?". Note that this is a subtly different question from what we actually want to know, which is "is this dnode dirty?".

There is a moment in the dnode write process where the dnode is taken off the "dirty" list, and put on the "synced" list. This happens before the data blocks are written out, so there is a moment where the dnode appears clean while its data blocks are dirty:



This gap is *tiny*. It's hard to estimate, but it's in the tens of CPU instructions. If `lseek()` comes along in that moment, the dirty check fails, so it goes into the "search for data/holes" codepath. Because that process only sees fully allocated and written blocks, it actually sees something more like this:

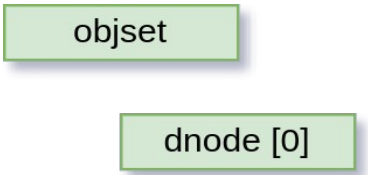
And so, if a program asks about holes, it may well end up wrongly finding them in the last couple of blocks of the file.

B. *Copying a hole*

Of course, the data is there, and if the program had tried to read it it would have found it without issue. Unfortunately, OpenZFS gave it reason to believe that it wasn't there, so it didn't bother.

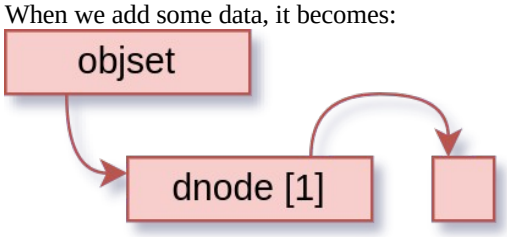
The key part of the test case that reproduced this bug was that it used the `/bin/cp` program from GNU Coreutils[3], which is almost ubiquitous on Linux systems. In v9.2 it started using `lseek()` to discover holes in source files.

When a new file is created, it starts life as:

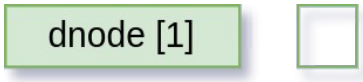


That is, just the metadata, no data.

When we add some data, it becomes:



If we manage to catch it during the moment when it's not on a list, it looks like:



`cp` begins with `lseek(..., 0, SEEK_DATA)`, which is to say, "find the first data item from position 0". When it finds data, it returns `0`, because that's where the data is. If it fails, it sets error `ENXIO`, to indicate "no data here". So `cp` thinks there is no data, doesn't bother to read, and writes zeroes (or a hole) to the output.

Incidentally, that's why this isn't "corruption" in the traditional sense (and why a scrub doesn't find it): no data was lost. `cp` didn't read data that *was* there, and it wrote some zeroes which OpenZFS safely stored.

C. *Measure twice*

While the conceptual problem runs deep and needs more work to fix properly, a good-enough workaround was implemented and released in OpenZFS 2.2.2[4]:

```
diff --git module/zfs/dnode.c module/zfs/dnode.c
index 029d9df8a..fdc3ea3a4 100644
--- module/zfs/dnode.c
+++ module/zfs/dnode.c
@@ -1786,7 +1786,8 @@ dnode_is_dirty(dnode_t *dn)
    mutex_enter(&dn->dn_mtx);

    for (int i = 0; i < TXG_SIZE; i++) {
-       if (multilist_link_active(&dn-
>dn_dirty_link[i])) {
+       if (multilist_link_active(&dn-
>dn_dirty_link[i]) ||
+           list_head(&dn->dn_dirty_records[i])) {
            mutex_exit(&dn->dn_mtx);
            return (B_TRUE);
        }
```

This is taking the existing "is this dnode on a list?" check, and extending it with a "is there any dirty data?" check. It's very simple, and with this change in place we have been unable to reproduce the problem.

This change does have a small conceptual problem in that it doesn't actually close the gap where the dnode is not on the dirty list, but rather, changes the test such that if it does happen to fall in that gap, it will have a something else it can check to ensure that the overall test still produces the correct result.

This makes the assumption that all dnode changes will be accompanied by a dirty record. This is not technically the case, for example, certain kinds of file attribute changes will only modify the dnode. There don't appear to be any possible dnode-only changes that could be interpreted as a hole → data transition, so the fix achieves its purpose for now. However anything that looks at only dirty list membership as an indicator of overall dirtiness may need to take the same

precautions in the future, so work continues to find a more correct fix[5].

IV. Why workarounds worked

In the course of exploring the problem, we found that setting the config parameter `zfs_dmu_offset_next_sync=0` stopped the problem occuring. This quickly became the recommended workaround. It also brought some confusion of its own, as it appeared to improve performance, leading some to wonder if they should just leave it off permanently. It also turns out that it doesn't actually stop the bug from occurring, just makes it vastly more difficult to hit.

A. *Transaction pipeline*

To understand why this is we need to know a little about the transaction pipeline.

At any point in time, OpenZFS has three transactions in flight:

- The `OPEN` transaction, which receives new changes as they arrive.
- The `QUIESCING` transaction, where the changes are combined and prepared to be written to disk.
- The `SYNCING` transaction, which is being written out to disk right now.

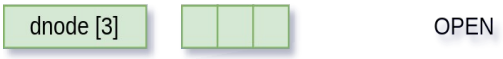
Once the `SYNCING` transaction is written, everything is moved along. `QUIESCING` becomes the new `SYNCING`, `OPEN` moves to `QUIESCING`, and a new `OPEN` transaction is created. As before, this is a simplified but useful explanation.

Changes are only added to the `OPEN` transaction. Once it moves to the next state, it can't be modified again. So a busy object can end up with changes on multiple transactions. This means that when we ask "is this object dirty?", what we're actually asking is "is this object dirty on transaction N?" or "is this object dirty on *any* transaction?"

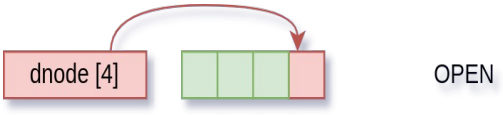
B. *How a change works*

Here's a visual representation of how changes flow through the transaction pipeline. This is conceptual, of course - "clean" objects are not on any transaction, but then again, "transactions" are not a real thing either!

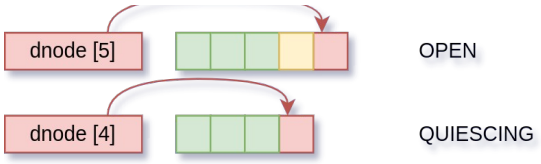
So lets imagine a nice normal clean object, on the current transaction.



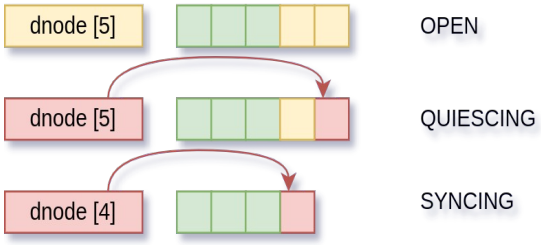
We append a block to it, which makes it dirty in memory on the current transaction.



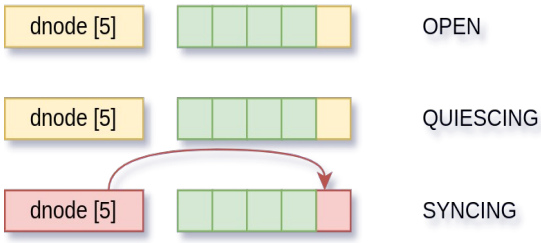
On the next transaction, we append another block to it. Yellow here shows the changes that aren't on disk yet but are considered "unchanged" on this transaction.



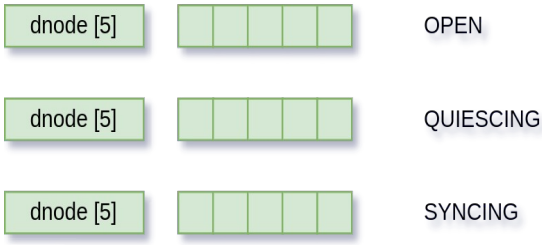
This time we don't make any modifications, so this object is "clean" on the OPEN transaction. The first change we made is now on the SYNCING transaction, and starts getting written out.



The transactions move along again and the second change starts getting written out.



All changes are are now written.

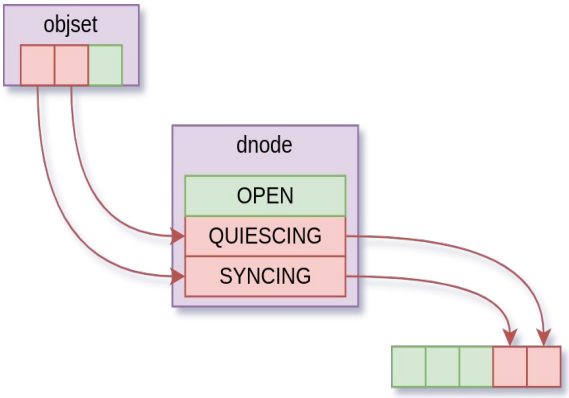


C. *All things are dirty*

Earlier I wrote:

> The objset has a list of dirty dnodes, and each dnode has a list of dirty blocks. So after our write, we have some nice linked lists of dirty things, ready to be written out.

But with multiple transactions in flight, we actually need a dirty dnode list and a dirty block list for each transaction, which means things actually look more like this:



This of course means the dirty check is more complex too. I wrote:

> As a result, there's no way currently to discover if a dirty block will eventually be stored as a hole. For this reason, `lseek()` checks if the dnode is dirty, and if so, it waits for it to be written out before it goes looking for holes.

The user's view of the object (via system calls like `lseek()` is whatever is on the `OPEN` transaction. Because looking for holes means looking at the on-disk state, this means we have to wait until *all* pending changes are on disk. So our dirty check here has to be "is this object dirty on any transaction?", which means checking if the dnode is on *any* dirty list.

D. *Wait for service*

Recall that

> `lseek()` checks if the dnode is dirty, and if so, it waits for it to be written out before it goes looking for holes.

Even if the object is dirty only on a single transaction, that wait can be long if the pool is very busy. So OpenZFS has a config option, `dmu_offset_next_sync` which controls what the `dmu_offset_next()` function (one of the work functions implementing `lseek()`) does when the dnode is dirty.

If set to `0`, and the object is dirty, then instead of waiting, the function simply returns an error, which causes `lseek()` to return "data here". This is allowed; it's always safe to say there's data where there's a hole, because reading a hole area will always find "zeroes", which is valid data. It can mean a loss of efficiency, because the application can't do special things for holes, but at least the data is always intact. And, because it's a simple dirty check, it's fast.

If it's set to `1` (the default), it will wait instead. Specifically, it will do a dirty check, and if it's dirty, it will wait until the currently-syncing transaction is written out. Then it will do another dirty check; if the object is still dirty, it will return an error (and so `lseek()` returns "data here"), and if clean, it will go into hole detection as normal.

So now we understand the apparent performance gain with `dmu_offset_next_sync=0` – it just never waits. But why does that also appear to mitigate the bug?

E. *Doubly dirty*

Recall that the bug occurs in the moment where the dnode is taken off the dirty list, but not yet put on the syncing list, and in that moment it's not a list, so it's incorrectly considered to be clean.

Since there's actually multiple lists, but this gap only occurs on the `SYNCING` list, it gives some idea of where the window you have to hit is. It's not a change "just made", but actually made two transactions ago (from the perspective of the user). On a quiet system that can be whole seconds ago. On a busy object, with changes on multiple in-flight transactions, a single dirty check (ie `dmu_offset_next_sync=0`) is always going to result in "dirty", and so `lseek()` returning "data here", because even if there's a gap on the `SYNCING` list, the dnode will also be on the `OPEN` or `QUIESCING` dirty lists, and so still be considered dirty. So, the only time we can hit a gap is for the "some time later" case. Not out of the question, but incredibly difficult for any non-contrived workload (or even contrived – to my knowledge it has only been seen on a quite large and unusual ARM64 system[6], and even then it's hard to hit).

In the "wait for sync" mode (`dmu_offset_next_sync=1`), we have two dirty checks, and either of them can go wrong. For the first check to fail it's the same as above; the "some time later" case. It's the second that is more interesting, with a wait in between.

To get the wrong answer, the dnode must only be on the `SYNCING` list, so that when it's removed, it's not on any other lists. The two dirty checks are one transaction apart, so for the second to *incorrectly* return "clean", the dnode must be dirty on two consecutive transactions. That's not hard to arrange of course, but that alone isn't really going to make much difference.

The difference is just the timing. The first (or only) dirty check can happen any time; whenever `lseek()` is called. The second however is always called just the after the transaction has synced and the next one is beginning sync. The object in question is dirty there, and is removed from the dirty list and added to the syncing list before it starts syncing, which means there's a better-than-usual chance that it's happening just as `dmu_offset_next()` loops and does the dirty check. Overall, it raises the chances of hitting the gap and getting the wrong answer.

V. History of a bug

This bug has a long history, as can be seen by reading multiple earlier patches that made changes around it. All had seen *something* wrong, but hadn't fully understood it. Looking back through the commit history, there's multiple commits that have modified the logic, but if we look very closely, all are effectively swapping between checking the dnode to checking the data and back again:

•Nov 2021: `de198f2` Fix lseek(SEEK_DATA/SEEK_HOLE) mmap consistency[7]

```
-    for (i = 0; i < TXG_SIZE; i++) {
```

```
-       if (list_link_active(&dn-
>dn_dirty_link[i]))
-           break;
+   if (dn->dn_dirtyctx != DN_UNDIRTIED) {
+       for (i = 0; i < TXG_SIZE; i++) {
+           if (!list_is_empty(&dn-
>dn_dirty_records[i])) {
+               clean = B_FALSE;
+               break;
+           }
+       }
```

•May 2019: `2531ce3` Revert "Report holes when there are only metadata changes"[8]

```
    for (i = 0; i < TXG_SIZE; i++) {
        if (multilist_link_active(&dn-
>dn_dirty_link[i])) {
-
-           list_t *list = &dn-
>dn_dirty_records[i];
-           [checks against dn_dirty_records]
```

•Mar 2019: `ec4f9b8` Report holes when there are only metadata changes[9]

```
    for (i = 0; i < TXG_SIZE; i++) {
        if (multilist_link_active(&dn-
>dn_dirty_link[i])) {
+
+           list_t *list = &dn-
>dn_dirty_records[i];
+           [checks against dn_dirty_records]
```

•Nov 2017: `454365b` Fix dirty check in dmu_offset_next()[10]

```
-   if (dn->dn_dirtyctx != DN_UNDIRTIED) {
-       for (i = 0; i < TXG_SIZE; i++) {
-           if (!list_is_empty(&dn-
>dn_dirty_records[i])) {
-               clean = B_FALSE;
-               break;
-           }
+   for (i = 0; i < TXG_SIZE; i++) {
+       if (list_link_active(&dn-
>dn_dirty_link[i])) {
+           clean = B_FALSE;
+           break;
```

•Mar 2017: `66aca24` SEEK_HOLE should not block on txg_wait_synced()[11]

```
-   for (i = 0; i < TXG_SIZE; i++) {
-       if (list_link_active(&dn-
>dn_dirty_link[i]))
-           break;
+   if (dn->dn_dirtyctx != DN_UNDIRTIED) {
+       for (i = 0; i < TXG_SIZE; i++) {
+           if (!list_is_empty(&dn-
>dn_dirty_records[i])) {
+               clean = B_FALSE;
+               break;
+           }
+       }
```

We've independently confirm the bug existed back as far as OpenZFS 0.6.5 (September 2015), however the proto-bug seems to go back even further, to a 2006 commit[12] in early Sun ZFS. Among other changes, we see:

```
    for (i = 0; i < TXG_SIZE; i++) {
-       if (dn->dn_dirtyblksz[i])
+       if (list_link_active(&dn-
>dn_dirty_link[i]))
            break;
    }
```

This is where it changed the dnode dirty check from checking a property of the dnode itself to checking if the dnode was present on an external list. That alone is not a bug; it might be that there's something else preventing the off-list gap being hit (eg another lock), so maybe the bug was never in this version of ZFS. What has changed is that it's no longer possible to understand the dirty state directly; it's now reliant on what's happening elsewhere in the system.

Due to difficulty getting access to old hardware and the answer actually being of little more than curiosity value, it's not known if this bug can actually be reproduced on early ZFS. This would be an interesting experiment for a sufficiently motivated reader.

A. Clone car

The original bug appeared to point to block cloning as being the cause of the problem, and it was treated as such until the problem was reproduced on an earlier version of OpenZFS without block cloning. It turned out not to be directly involved, but did distract from the real issue for a while.

However some people did report that disabling block cloning entirely appeared to helped. So far we haven't really found any reason why it would make a difference, other than at a distance by changing the timing – it's faster to apply a clone than a write, because there's no actual data to copy or write, so that allows more dirty dnodes and/or more transactions to be processed in a shorter amount of time. This alone means nothing, as the gap is still the same amount of time, but with other variables like hardware and workload, it could change it in the right way.

Another reason cloning was implicated was that the bug had been seen on FreeBSD. Its `/bin/cp` is not the Coreutils version, and doesn't look for holes, but does call `copy_file_range()`, which hands off copying to the kernel to do the copy however it would like. This function has become known as "the cloning function" because the kernel might choose to service the copy via clones. However on FreeBSD it will also use `lseek()` under the hood to try to replicate holes in the file, and so can run into the same problem. `copy_file_range()` on Linux doesn't do this, and that's most peoples experience of cloning, leading to confusion.

VI. Instant Community

Once word got out that there was a data corruption bug, many people arrived on the scene, most of them concerned about their own exposure. This added an element of urgency, as even if the problem turned out to be nothing, not taking it seriously could damage OpenZFS' reputation. However, it also brought with it an opportunity, in that there were many interested people gathered in one place, who could be part of the solution.

Once the reproducer script had been posted, people started to try it and reported their results. Meanwhile, others were studying those reports to try to find the common elements. This helped us to identify the version of Coreutils that appeared to be causing problems, and once we had that, we were able to put out a more specific request: that people try the test case and report results including specific configurations and versionstest-request.

This request was wildly successful, in ways I had not expected. It served its primary purpose of gathering enough

data to confirm a theory, which helped to focus the search onto a specific code path, and from there identify the issue and develop a fix.

Less deliberately, it also had a galvanising effect on everyone that had arrived. People took this request and tried it and far more combinations than we expected, or were even useful. They discussed the variations directly, in the ticket, and tried to come up with other ways to reproduce it. They theorised on timing issues, other pieces of software that might trigger the bug and possible differences across machine architectures and operating systems. Those conversations brought new ideas and helped with understanding the problem more deeply, and then people carried what they learned back to their other communities – tech forums, storage communities, specific operating system or distro communities, and so on. With it, they generally took a positive message rather than a negative one: that yes, this is a real problem, but it's also complicated, and it's being actively worked on by some interested and passionate people – including them!

That's why I rate this process as a qualified success. As I wrote in the ticket after the whole thing started to wind downrobn-conclusion:

> On a personal note, I've really enjoyed the way everyone here (and in other forums and chats) really got stuck into this problem, trying to understand it, testing so many combinations of software versions, OS versions and hardware, sharing ideas for detecting and recovering, reassuring each other, and etc. The bug obviously sucks and I wish it hadn't existed, but I'm very happy to have been a part of this little community coming together to solve an ancient mystery!

For me, this was a hugely significant reminder of what open source at its best can be - individuals coming together and bringing their own unique experiences and abilities to bear on an interesting technical problem. It's something I'm thinking about a lot and am looking for more ways to channel into future OpenZFS work – hopefully next time without the emergency!

VII. People need to eat

I work as an OpenZFS developer, however, I am paid to work on specific customer projects, not on general OpenZFS maintenance or support. This presented something of a quandary for me on the first day of this event. I am committed to my customers, but I'm also committed to OpenZFS in the long term – customers will come and go, but OpenZFS as a whole will continue. At the time it wasn't clear just how big the problem was, but I understood the reputational damage that could result if it went badly, and that reputation is a big part of why my customers choose OpenZFS. So I decided that it was better to take the short-term loss of income and try to contribute to a positive outcome.

As it was, I didn't end up taking a hit – my employer, Klara, Inc. is heavily invested in OpenZFS' future, and enough of our customers had heard about the bug and and were worried about it, and were happy to contribute to its solution. I am very grateful to Klara and our wonderful customers, as through them I got the best possible outcome.

However, this highlights a sustainability problem within OpenZFS. Business customers can pay for new features, and for some kinds of maintenance work. However, there is no specific funding that I'm aware of for general maintenance work, project management work, code review, or any feature work that is not useful or interesting to businesses. So when a problem like this arises, it can often be on individuals to make difficult choices.

All other things being equal, I should not have had to decide between my responsibilities to my family, to my employer & customers, and to an open source project used by thousands of individuals and companies. I am personally in a very fortunate position that I have flexibility in my schedule, a supportive family, an invested employer and customers, and can survive on not much sleep for a few days at a time. Not everyone is in this position, and that should not be our expectation.

This is not a suggestion that you give me money. This is a suggestion that as users, you think hard about the open source software you use and depend on and make sure you are doing what you can to give back some of the value that you derive from it, whether that's in money, time or expertise. And for people involved in these kinds of open source projects, we need to make sure we are making it as easy as possible for people to give back to us, and ensure that we're prioritising the right things and spreading our resources appropriately so that no individual has to make disproportionate personal sacrifices when something unexpected comes along.

VIII. Conclusion

All software has bugs, and OpenZFS is no exception. Far more importantly however, all significant software has individuals invested in its future. I assert that the key to long-term sustainability of open source software is creating places and pathways for those individuals to bring their own unique talents to the software they love without it being an undue burden on them or anyone else. I consider that we can we can pull together and do what we need to in an emergency as proof that this is possible. The challenge for OpenZFS from here is in establishing the systems and processes to ensure this way of working is the norm, rather than the expectation.

Acknowledgements

Personal thanks to the management team at Klara Inc. for giving me the opportunity to work on OpenZFS and continuing to support that work wherever it may lead.

References

[1] Stock et al., (2023), *some copied files are corrupted (chunks replaced by zeros)*, openzfs/zfs#15526

[2] Kerrisk, M. (2011), `lseek(2)`, https://www.man7.org/linux/man-pages/man2/lseek.2.html

[3] Free Software Foundation, *GNU Coreutils*, https://www.gnu.org/software/coreutils/coreutils.html

[4] Norris, R., (2023), *dnode_is_dirty: check dnode and its data for dirtiness* openzfs/zfs#15571

[5] Norris, R., (2023), *dnode_is_dirty: use dn_dirty_txg to check dirtiness*, openzfs/zfs#15615

[6] Kozicki, B., (2023), (comment), openzfs/zfs#15526 comment

[7] Behlendorf, B., (2021), *Fix lseek(SEEK_DATA/SEEK_HOLE) mmap consistency*, openzfs/zfs@de198f2

[8] Behlendorf, B., (2019), *Revert "Report holes when there are only metadata changes"*, openzfs/zfs@2531ce3

[9] Behlendorf, B., (2019), *Report holes when there are only metadata changes*, openzfs/zfs@ec4f9b8

[10] Behlendorf, B., (2017), *Fix dirty check in dmu_offset_next()*, openzfs/zfs@454365b

[11] Banerjee, D., (2017), *SEEK_HOLE should not block on txg_wait_synced()*, openzfs/zfs@66aca24

[12] Ahrens et al., (2006), *6395371 ASSERT in dmu_tx_count_free: blkid + i < dn->dn_phys->dn_nblkptr*, illumos/illumos-gate@c543ec0

[13] Norris, R., (2023), (comment), openzfs/zfs#15526 comment

[14] Norris, R., (2023), (comment), openzfs/zfs#15526 comment