The Old New Thing

If NTFS is a robust journaling file system, why do you have to be careful when using it with a USB thumb drive?

1 Jan 2013 7:00 AM | 68

Some time ago, I noted that in order to format a USB drive as NTFS, you have to promise to go through the removal dialog.

But wait, NTFS is a journaling file system. The whole point of a journaling file system is that it is robust to these sorts of catastrophic failures. So how can surprise removal of an NTFS-formatted USB drive result in corruption?

Well, no it doesn't result in corruption, at least from NTFS's point of view. The file system data structures remain intact (or at least can be repaired from the change journal) regardless of when you yank the drive out of the computer. So from the file system's point of view, the answer is "Go ahead, yank the drive any time you want!"

This is a case of looking at the world through filesystem-colored glasses.

Sure, the file system data structures are intact, *but what about the user's data*? The file system's autopilot system was careful to land the plane, but yanking the drive killed the passengers.

Consider this from the user's point of view: The user copies a large file to the USB thumb drive. Chug chug chug. Eventually, the file copy dialog reports 100% success. As soon as that happens, the user yanks the USB thumb drive out of the computer.

The user goes home and plugs in the USB thumb drive, and finds that the file is corrupted.

"Wait, you told me the file was copied!"

Here's what happened:

- The file copy dialog creates the destination file and sets the size to the final size. (This allows NTFS to allocate contiguous clusters to the file.)
- The file copy dialog writes a bunch of data to the file, and then closes the handle.
- The file system writes the data into the disk cache and returns success.
- The file copy dialog says, "All done!"
- The user yanks the USB thumb drive out of the computer.
- At some point, the disk cache tries to flush the data to the USB thumb drive, but discovers that the drive is gone! Oops, all the dirty data sitting in the disk cache never made it to the drive.

Now you insert the USB drive into another computer. Since NTFS is a journaling file system, it can auto-repair the internal data structures that are used to keep track of files, so the drive itself remains logically consistent. The file is correctly set to the final size, and its directory entry is properly linked in. But the data you wrote to the file? It never made it. The journal didn't have a copy of the data you wrote in step 2. It only got as far as the metadata updates from step 1.

That's why the default for USB thumb drives is to optimize for Quick Removal. Because people expect to be able to yank USB thumb drives out of the computer as soon as the computer says that it's done.

If you want to format a USB thumb drive as NTFS, you have to specify that you are Optimizing for Performance and that you promise to warn the file system before yanking the drive, so that it can flush out all the data sitting in the disk cache.

Even though NTFS is robust and can recover from the surprise removal, that robustness does not extend to the internal consistency of the data you lost. From NTFS's point of view, that's just a passenger.

Update: It seems that people missed the first sentence of this article. Write-behind caching is disabled by default on removable drives. You get into this mess only if you override the default. And on the dialog box that lets you override the default, there is a warning message that says that when you enable write-behind caching, you must use the *Safely Remove Hardware* icon instead of just yanking the drive. In other words, this problem occurs because you explicitly changed a setting from the safe setting to the dangerous one, and you ignored the warning that came with the dangerous setting, and now you're complaining that the setting is dangerous.

Blog - Comment List MSDN TechNet

Comments

digiowl
1 Jan 2013 7:35 AM
#

in other words, the issue is not the FS but the write cache used to give the appearance of better responsiveness than there really is.

The MAZZTer
1 Jan 2013 7:50 AM
#

Responsiveness is all about appearance. If it appears responsive, it IS responsive.

metafonzie
1 Jan 2013 8:06 AM
#

@digiowl, You could simulate that behavior w/o any caching involved. - e.g. Yanking a high write latency storage device.

The issue *is* the FS and its a by design problem (i.e. not including data in the undo/redo transaction logs). If you don't like this tradeoff use some other file system :P

configurator
1 Jan 2013 8:20 AM
#

There must be a better way to design a filesystem than requiring the writer to set the

size to create contiguous blocks (Not that I know why you'd need contiguous blocks on a thumb drive).

My issue isn't with the copy not succeeding completing, it's with the intermediate state. A file copy should increase the file size as it's writing it - that way when <anything> happens, you'd be able to continue copying at the point where you were interrupted. I'm not saying that's the way the file system and shell operations should be implemented - clearly more than a few minutes thought would have to be put into the design - but I am saying these are the side effects I'd want from a power outage. A partially copied file makes sense. A file of the same size as the original file but not the same contents seems wrong.

saveddijon
1 Jan 2013 8:36 AM
#

Truth is, USB devices ought to work just fine with any filesystem. The trick?

Removable devices of any kind should have the cache sync every 2 seconds worst-case. (Typical would be 30s-1min for a non-removable device.)

Then all you need to do is wait 2 seconds after any operation before yanking the drive.

AmigaDOS worked this way. There was a cache, but the flush time was on the order of 2 seconds. Therefore you never ran into an issue when, say, writing to a floppy.

And as the previous posters have noted, if the filesystem loses coherency (i.e. when it is recovered, the recovered state does not match any state the filesystem ever had in the past) then the filesystem design is wrong. Look at 4.4BSD for the right way to do this, and later versions with delayed ordered writes, which allow caching and coherency.

ender9
1 Jan 2013 8:50 AM
#

@saveddijon: there's always a trade-off - full journaling is slower than just metadata journaling. And if you modify NTFS to support full journaling, you still haven't solved anything - as this would change the on-disk structures, older versions of Windows would not be able to read this filesystem, resulting in much lower usability (remember, when you have a portable drive, you want to be able to read it on any computer you come across).

I'm more annoyed that most manufacturers mark USB pendrives as having removable media (they don't - that's intended for things like ZIP cartridges), and Windows paying attention to that and refusing to allow partitions on such drives (why?).

Joshua
1 Jan 2013 9:57 AM
#

Repeat after me. Do not lazy flush buffers for removable drives. In fact, it's a whole lot

better if you don't enable write-behind caching for removable drives in the first place.

@saveddijon: Ext3 has an option to journal data. It's so slow it is a non-solution even if Windows supported it.

@ender9: Perhaps setting that flag is the only way to convince Windows to not lazy-flush data.

Z.T.
1 Jan 2013 10:58 AM
#

How about making the file copy dialog do (the equivalent of) fsync()?

Noel Grandin
1 Jan 2013 11:11 AM
#

Once again MS is too lazy to do the right thing, which is to auto-disable the NTFS write-behind cache when working with removable drives.

> [*Um, it's disabled by default. This problem occurs when you override the default and say "I want dangerous write-behind caching on removable drives." -Raymond*]

ErikF
1 Jan 2013 11:37 AM
#

@Noel: But how does Windows know that it's a removable drive when the USB stick claims that it's a hard drive? Whether you like it or not, the Windows definition of "removable drive" is the kind that is in a 5.25" or 3.5" form factor that holds between 120KB and 2.88MB and has no partition table. The fact that USB drives present as "block devices" with no other information doesn't really help matters any either, in my opinion.

BTW, Windows is not the only operating system that wants you to safely remove USB devices: OS X gives you a scary error dialogue when you pull a USB drive out, and Unices really don't appreciate it when you pull a mounted filesystem out.

Edward M. Grant
1 Jan 2013 12:10 PM
#

"But how does Windows know that it's a removable drive when the USB stick claims that it's a hard drive?"

And how does it know that disabling the cache is the right thing to do? Disabling the cache means I have to wait longer for the copy to 'complete' to disk cache, when I might be waiting to do other work while it's physically being written to the USB drive

before I remove it. There's simply no 'one right way' to handle USB drives, because the user can pull them out at any time and the OS can't tell when we're going to do that.

All that said, I still stick to FAT32 on my USB drives because it's the only filesystem that every machine I use can reliably read and write.

Paul Coddington
1 Jan 2013 12:49 PM
#

This about covers the point I make when techies tell me that UPS is not necessary because NTFS is robust - an intact file system that is internally consistent is not the same as retaining data that was in the middle of being changed (at any given time, more can be changing than you think). Still, not as bad as the techies who once insisted that NT machines should always use FAT on their system disk "as recommended by Microsoft" (like heck it was).

I have seen CHKDSK discard hundreds, perhaps thousands of files on a damaged NTFS system drive in the process of making it pass all future CHKDSK operations with a perfect record. This should be a lesson to all - and a case in point as to why it is a bad idea to hide the CHKDSK progress report from the end user.

Gibwar
1 Jan 2013 1:35 PM
#

@configurator: except now that didn't solve anything and added nothing but more work and more complexity. More work comes from how is the system to know that the file should be resumed? To properly implement it you'll now need to hash each block before copying it to see if its the same or not. If the hashes don't match then what? Also, what would happen if I had a file that was 2GB large and I copied another file with the same name (completely different contents) that was 2.5GB? If you implement hashing, you'll have to go through the entire file just to find out they aren't the same at all and throw the "replace this file?" dialog and have wasted tons of the users time on the operation just reading the file, not copying. If you don't implement hashing and just append the remaining 500MB on, well, I hope the program that uses the file is robust enough to determine that there is 500MB of unrelated junk tacked on the end of it for no apparent reason.

As for the "why bother setting the size ahead of time" that allows the filesystem to determine if there is enough space at the beginning of the copy and fail fast. Imagine if you had a 2GB drive and you started copying 4 1GB files on to it. Each one would fail at a different point, or maybe some of them wouldn't? By preallocating the space up front, the first two would be allowed to copy and the last two would be denied with "not enough space is available." Personally, I prefer it the way its set up now, its predictable and user friendly.

AC
1 Jan 2013 3:37 PM
#

Okay, so the maint point of the article makes sense. Just because NTFS is a journaling FS, doesn't mean you can just yank out an USB drive before the cashes are flushed. Agreed so far.

But what about the linked old article: Why do you need to "have to promise to play friendly" to format as NTFS?? I don't get that at all. Doesn't yanking out a drive formatted with the inferior FAT FS have the same problems? If 'optimize for quick removal' means limiting the disk caching, why can't this be done for NTFS?

Destroyer
1 Jan 2013 3:42 PM
#

(pressed post but didn't receive the green confirmation :-|)

What an article to start 2013! It confirms my assumptions on this exact topic have been right :-)

I think some people are speculating here and don't actually have any real world experience with USB drives. There is NO problem with NTFS here as far as I am concerned, it is totally in the hands of the user/application*.

I believe in all recent versions of Windows (certainly 7 and 8) it does disable write cache to the USB drives or reduces it, at least this is from all my experience dealing with them (which is quite a lot). This can be verified by watching the activity light flash on the USB drive after pressing "safely remove hardware", I have never, ever, seen any activity from pressing that button, confirming that there is no caching, when a USB flash drive is formatted as NTFS, even when pressing it almost instantly after a file copy in Windows.

Also, most modern USB flash drives continue to flash their activity light for several seconds after the write has completed on purpose to stop the user unplugging this, which further causes confusion as Windows will correctly report it is safe to remove the device (it is from its perspective), yet the device still flashes for another few seconds.

*The 2 second idea wouldn't help unfortunately imho, most of the time it is neither Windows or the application, it is the user. They will click log off then yank the flash drive out immediately, Windows is busy closing down all their applications that they couldn't be bothered to close down manually, and that "bong" message that they hear is a "Delayed Write Failed" error message telling them to re-save their document, oh but hang on they are logging off and probably are already walking away from the computer at that point. Windows has even TOLD them that the save was unsuccessful but they don't care. Yes, I know that it is a "Delayed" write, but it could be delayed write in the order of a few milliseconds, it makes no difference, even if the write wasn't delayed it would have probably been cut off in the middle anyway.

You can't have it both ways, users either treat the device with common sense or treat them like children so whenever they click save a big message appears and STOPS them doing whatever they were doing until the save is totally, absolutely, complete. Somehow I don't think they would be very amused with that.

The problem is most people are dealing with small document files so they very rarely see any problems until it all goes wrong. If you was working on a 4GB photoshop file etc you might think twice about pressing save and then pulling the USB drive out a second afterwards. Well, you might do it once, and probably learn your lesson.

I format all my flash drives as NTFS, it is far more reliable than FAT32.

alegr1
1 Jan 2013 8:26 PM
#

Cache behaviors are implemented at the filesystem driver. FAT may honor the removable media/device property, while NTFS may not, because historically, hard drives haven't been removable for a while.

I remember Win2003 having horrible behavior with USB flash. If you forget to use safe removal, you'll get hit with some "delayed write error", even if you HAVEN'T MODIFIED A SINGLE FILE ON IT.

If you want better FS for a USB flash, use exFAT, not NTFS.

Alex Cohn
1 Jan 2013 11:50 PM
#

@Destroyer: I agree with your analysis, but there may be a very legitimate reason for the drive lights to flash for a while after Windows considers the transaction to have completed. A USB drive may use some internal cache to release the host quickly, but still need electric power to flush the data to slower flash memory.

ender
2 Jan 2013 1:04 AM
#

@Joshua: no, USB harddrives aren't seen as removable, and still have write caching disabled by default.

RichB
2 Jan 2013 1:38 AM
#

Recent versions of NTFS not only support journaling, but transactions:

msdn.microsoft.com/.../aa363764(v=vs.85).aspx

It's a shame more applications don't make use of them. It's also a shame this feature isn't publicized more in msdn blog posts.

> [*Transactions have their own cost, however. Suppose you have a USB drive with 1GB of disk space free, and there is a 2GB file on the drive that you want to overwrite with a 2GB file from your hard drive. If you tried to do this file copy via a transaction, you'd get an "out of disk space" error. -Raymond*]

Markus
2 Jan 2013 4:10 AM
#

@RichB: msdn.microsoft.com/.../hh802690(v=vs.85).aspx (Alternatives to using Transactional NTFS):

"Microsoft strongly recommends developers investigate utilizing the discussed alternatives (or in some cases, investigate other alternatives) rather than adopting an API platform which may not be available in future versions of Windows."

I agree, though, it's a shame.

Matt
2 Jan 2013 4:48 AM
#

To everyone who thinks that journaling data is a good thing, think about this.

I have a 1GB USB stick with a 500MB zip file on it. I replace the zip file with a 600MB zip file, and my OS says "I can't do that, because the disk isn't big enough" - you can't journal all of the 600MB to the disk and have a 500MB recovery space for the file incase anything goes wrong.

So what you've reduced this to is a user saying "Hang on - my disk have 1GB storage and yet I can't put a single 600MB file on it?! This is BS!".

Instead, Microsoft are optimising for the default (good) case where users click to remove their USB stick.

This is all a special case of "don't optimise for the case where users are doing it wrong".

dirk gently
2 Jan 2013 5:57 AM
#

The real WTF is that the file copy dialog says that "the job is done" before the data is flushed.

> [*The file copy dialog doesn't know about disk flushing. That occurs at a lower layer. Sometimes it happens in the hardware itself. -Raymond*]

MS Bob
2 Jan 2013 7:55 AM
#

Does Windows 8's new file copy dialog and ability to pause transfers makes any difference at all compared to how it was done earlier? I've yanked out drives while file copy was paused, used them elsewhere, then reinserted and managed to continue the transfer, but I don't know whether Microsoft has ever thought of and tested this sort of

thing.

alegr1
2 Jan 2013 7:58 AM
#

>The real WTF is that the file copy dialog says that "the job is done" before the data is flushed.

Nope. For a USB flash, the file is flushed before the file copy dialog is closed. It's either done at CloseFile, or by explicit FlushFileBuffers call.

Vista/Win7Gold used to have a big problem with that. They didn't hesitate to cache A LOT of the file and issue A LOT of pending writes. If you have a large file, you could not easily abort the copy to USB, you had to wait until all those writes completed and data flushed.

anonymous123
2 Jan 2013 8:50 AM
#

Won't you get the same corruption also with FAT?

[*Yes, but FAT isn't a journaling file system, so nobody expects it to work with FAT. - Raymond*]

David Walker
2 Jan 2013 10:47 AM
#

If you write to an NTFS-formatted flash drive, then let it site for 5 minutes on a mostly-idle system, and then do an unsafe removal ... will the data be intact? I have done this a few times (not on purpose) and I have never seen missing data as a result. Was I just lucky, or should I expect that allowing the data to "settle" for a few minutes is sufficient?

Nawak
2 Jan 2013 11:35 AM
#

"The file is correctly set to the final size"

I'm surprised that you consider that the correct behaviour... an optimized behaviour maybe, but not a correct one!

IMHO the size should be set to a value lower than what was correctly transfered.

The fact that NTFS reserves the amount of space needed for the whole file (a good thing, as Gibwar explained) should not necessarily imply that it has to directly set the file

size to its final value. On FAT (at least) the two notions (cluster chain reserved for the file vs. file's size in directory entry) were independent and I imagine NTFS would also support manipulating them independently.

The "correct" behaviour would imply updating regularly the file's size in the file's metadata. Obviously that would degrade performance a little bit, but so is keeping a journal...

From the designers POV, maybe they consider both cases to be equivalent (file is corrupted), but one is more treacherous (file's properties seem good) and I don't consider the performance gain to be worth it. (Performance could be made equivalent by setting the size also once, but at the end, like FAT did IIRC (I have memories of long downloads resulting in 0-byte files when interrupted))

digiowl
2 Jan 2013 11:58 AM
#

meh, starting to think that USB drives should have a mandatory button with a led. When you want to remove the drive, you press the button and the led starts flashing. When it goes from flashing to solid you can remove the drive.

JJJ
2 Jan 2013 12:16 PM
#

(Not that I know why you'd need contiguous blocks on a thumb drive).

When you overwrite data on a flash drive, you have to erase it first. The problem is when you erase even one single byte, the hardware requires you to erase an entire block of data (Could be any size. 16 KB, 64 KB, 512 KB...). Then you have to write your new data and re-write the rest of the data in the block.

So you want your data to be contiguous to avoid writing single bytes to a block, which really results in an entire block being erased and re-written.

Matt
2 Jan 2013 12:46 PM
#

@Nawak: How can you set the length of the file to be less than the data in the file? The length of the file is the number of sectors that have been allocated to it. Putting more data in the file than sectors are allocated to it means that you've buffer-overflowed something on disk!

In NTFS, setting the file-size is a journalable and high-cost action. You set the length up-front in order to allocate the sectors and then write data into it. This way every WriteFile only needs to update the EndOfFile pointer in the file record and write the data itself to disk, rather than doing a full journalling action to allocate more sectors (which might shuffle stuff on disk or trigger a defragment etc) several times during the copy.

@JJJ
2 Jan 2013 12:47 PM
#

The filesystem only writes in units of one or more blocks to the block device. Smaller writes are cached (that's what the filesystem cache is mainly for).

dnm
2 Jan 2013 12:51 PM
#

Oh wow! I wish i had read this last weekend when I screwed up my USB tyring to make a U-boot device using NTFS....It had worked the first time when I did not yank it right away. And then I did not why this was not working. Thanks a ton Raymond for the insight

Killer{R}
2 Jan 2013 2:12 PM
#

@digiowl "meh, starting to think that USB drives should have a mandatory button with a led"

I think the most reliable solution will be to add few teeth around USB socket that will hold device in mouth til it will be safe to remove it. Hope USB standard-developers will read this message. Name the new standard USSB - Universal Serial Shark Bus. And paint them in red, to avoid mess with bluetooth.

Nawak
2 Jan 2013 3:13 PM
#

@Matt:

I don't know about NTFS but on FAT FS at least, the space allocated is described in the FAT (by chaining clusters of disk sectors together) and the file's size is in the directory's entry for the file. The space allocated for the file is nearly always different from the file size, because the allocation is at least the file's size rounded up to the next cluster (often several kB). So the differentiation between space allocated and file size is built-in. It's even a founding principle. The logic would be that the file's allocation is always at most 1 cluster (minus 1 byte) larger than the file, but nothing prevents you from having a 1 byte file using a 1 GB allocation.

You can for instance use that technique to reduce the number of FAT writes so that a larger proportion of the writes serve for data and not for the FS bookkeeping.

(1 sector of 512 bytes describes the state of 128 clusters, so in the best case you can allocate 128 clusters at once, which is better than 128 successive writes)

Unused space is reclaimed at close() time or by a scandisk.

Anyway, as I said, I imagined that NTFS had a similar differentiation and that a file's size didn't have to change alongside with its allocation. If that's not the case then, well, the inconsistency that me react came for free with the FS design, and not from a particular decision about what would the "correct" behaviour be in the case exposed by Raymond.

Destroyer
2 Jan 2013 4:54 PM
#

@David - yes, in fact, more like 5 seconds should be enough from what I have seen/do, at least when doing any copying functions in explorer or robocopy or Office suite. Other programs - who knows.

The problem with transactions is that we are dealing with loss of saved work most of the time, the only thing using an atomic write would do is mean the file is not there rather than there but corrupt. Yes it may stop the old copy getting corrupted but people shouldn't work from important files from a memory stick without a backup regardless!

It actually may cause the number of instances of corruption/loss of saved work to increase due to the increase in write time, and also, when users have pulled the disk out at least I can open up the .docx in notepad sometimes and get their typing back, yes the formatting is lost but that's tough. If it was using transactional NTFS there would be no corrupt .docx file there to try and recover!!

Matt
3 Jan 2013 1:51 AM
#

@Nawak: If you allocate the data upfront then you don't have to do so many resizes of the file when you hit inevitable boundaries.

It's like the difference in C# between

var q = new List<int>()

for(int i = 0; i < 10000000; i++) q.Add(i);

and

var q = new List<int>(10000000)

for(int i = 0; i < 10000000; i++) q.Add(i);

The second one is much faster because you allocated the right amount of space up front and never need to resize.

In NTFS you can of course open a file and start spitting data into it, but if you know how big it's gonna be when you close the handle after the write, you might as well give NTFS the opportunity to preallocate space on the disk for it (which is why CopyFile does that).

Tom A. Mason
3 Jan 2013 3:54 AM
#

The solution here is clearly that the external case of the USB pendrive should be electrified to give the user a shock preventing them from removing it until the write has been completed.

xpclient
3 Jan 2013 10:32 PM
#

More guidance needed from MS on whether NTFS or exFAT is better for flash memory. I mean NTFS has a number of feature advantages despite exFAT being optimized for wear leveling.

> [*The guidance is already built-in by the fact that you have no choice. By default, USB thumb drives cannot be formatted NTFS, and the system volume cannot be formatted exFAT. -Raymond*]

Nawak
4 Jan 2013 12:42 AM
#

@Matt:

I guess you are proving my point. In your examples, what does the "Count()" method return if you stop the Add loop at 5000000? I guess it returns 5000000 in both cases, so it proves that allocation and size are considered different by this class, and that "optimizing" doesn't necessarily imply changing the observed behaviour of the Count() method.

My original point was that stopping the 2nd example in the middle should not make Count() return 10000000, because that's kind of what NTFS is doing in Raymond's example. IMHO, it shouldn't return the full size if the full write has been interrupted!

(And if Count() *does* return 10000000 instead of 5000000 when the loop is shorter than (initially) expected, well that would mean that NTFS's design is meeting everyone's "natural" expectations except mine, and that's really ok)

dirk gently
4 Jan 2013 2:10 AM
#

[The file copy dialog doesn't know about disk flushing. That occurs at a lower layer. Sometimes it happens in the hardware itself. -Raymond]

ok, so the problem is that the "lower layer" is not able to signal that the data has actually been flushed. The dialog waves happily to the user that the job is done, but from the user perspective the job is not done. What the user requested, is that data is

written to the device, not to the device's volatile cache (which the user is probably even unaware of).

> [*The solution is to use the default setting of "Optimize for quick removal." Remember, you got into this pickle because you changed a default, and the dialog box where you change the default says, "To disconnect this device from the computer, click the Safely Remove Hardware icon in the taskbar notification area." You chose a dangerous setting and ignored the warning that came with it, and now you're blaming Windows. This is why we can't have nice things. -Raymond*]

Matt
4 Jan 2013 2:20 AM
#

@Nawak: I think you are looking at the problem from the wrong level. Setting the length operation is an optimisation FOR ntfs made BY the CopyFile API in Windows.

In other words, CopyFile acts like this:

HANDLE hFrom := CreateFile(pFrom, GENERIC_READ);

SIZE_T hFromLength := GetFileSizeEx(hFrom);

HANDLE hTo := CreateFile(pTo, GENERIC_WRITE);

: NtSetInformationFile(hTo, hFromLength);

CopyLoop(hFrom, hTo);

CloseHandles(hFrom, hTo);

From NTFS's point of view it's been asked to create an exactly X MB file of zeros and then every zero is then overwritten with data. If you stop halfway through, the rest of the file is zeroes because that's what CopyFile said it wanted, and because CopyFile's error path is to delete the destination file, assuming no power failures, memory corruptions or invasive monitoring you won't be able to tell the difference.

This optimisation is over not first setting the length, in which case NTFS would say that the length is however much data you've written to it so far.

This isn't NTFS being clever, it's CopyFile being clever. And since nobody else should be looking (or caring) about the length of the destination file until you've actually written it, and you get an important performance boost for the boring task of copying lots of big files to your thumb-drive, since power off kills your data anyway, and since the error path in CopyFile deletes the file, this isn't a normally observable optimisation, and hence is a good optimisation to take.

Nawak
4 Jan 2013 7:38 AM
#

@Matt: I understand what you are saying, but where NTFS's "fault" lies is that NtSetInformationFile only offer to change a file's allocation by changing its size.

I agree that it is CopyFile being clever, but CopyFile could have been more clever if NTFS had proposed a service to only change allocation. As it is now, NTFS can consider itself right because what CopyFile instructs it to do is to actually do two "data write passes":

A) One (invisible) write pass to set the file size: The EOF information is changed *and* all the data belonging to the allocated space is considered part of the file (that's why I call it a write pass). (An interesting note is that the doc doesn't say that the file is filled with zero, so you may end up with somebody else's data?)

B) One write pass to fill the file with the correct data. This pass doesn't change the file's size.

When NTFS is repairing the filesystem by looking at the journal, it doesn't revert the file's size since it was changed on step A and not step B (step A was correctly completed)

If I were to assign the blame (that's always the fun part of being a user), I would still blame NTFS. Yes even if I just said that it did what it was instructed to do! Because by not offering a service to change a file's allocation without changing its size, it forced CopyFile to worsen a case to get acceptable performance:

- either CopyFile didn't preallocate: the corruption caused by the unexpected drive removal would be more visible but the global performance would have sucked (too many file-info updates),

- or it did preallocate (by pre-sizing): the corruption caused by the unexpected drive removal is much less visible but the global performance is correct.

I think CopyFile ended up doing the right choice because the nominal case (no unexpected drive removal) is faster and the slower alternative wouldn't have made the corruption go away anyway.

(BTW, when I said in an earlier post that a wrong decision had been made regarding the performance vs the "correct behaviour", I thought that it was NTFS being clever with what it had been asked to do, but now, knowing that one cannot allocate without resizing, I think that NTFS's wrong decision is actually to not have the allocation service!)

alegr1
4 Jan 2013 8:57 AM
#

@Nawak:

Besides from file length, NTFS maintains valid data length for a file. Any read beyound valid data length brings zeros; this is why you can't read old data this way. This makes it unnecessary to do an explicit zero-write pass. If you have enough privileges, you can modify valid data length and read the old data.

FAT doesn't have a valid data length. If you set file length, you may be able to read old data, or not, depending on whether you've got pre-erased flash blocks. But this is OK, because FAT is not considered secure.

Matt
4 Jan 2013 9:34 AM

#

@Nawak: CopyFile doesn't ask NTFS to do two data passes. Setting the information length allocates more blocks from the bitmap into the NTFS file structure. It doesn't write any data (other than the change to the file inode and journalling the change) to disk

I think perhaps you are confused about the NTFS file inode - each file has three key "lengths" with regards to data:

* A "reserve" capacity, which is the number of allocated blocks that the file has exclusive rights to use. Writing data to these blocks is not journalled, but changing the capacity up or down is journalled. Reserve capacity is measured in whole data chunks.

* A "commit" length, which is the number of bytes in the allocated blocks that have valid data. This pointer allows NTFS to avoid writing 10MB of zeroes when you allocate a new file and tell it via NtSetInformationFile that the file is 10MB big. Whenever someone in usermode asks for data after this length, NTFS returns zeroes instead of data from the disk.

* A "filesize" length, which is the reported length of the file.

Consequently we can see that the actual actions going on are:

1. CreateFile, which allocates the inode in the first place, is a journalled action and which sets the filesize to 0, commit to 0 and potentially allocates some reserve.

2. NtSetInformationFile, which sets the filesize to the size requested, sets the "reserve" capacity to equal than or bigger than the size, but does not modify the "commit" length (unless you shrink the file). This is journalled since allocating blocks from the bitmap is a journalable action.

3. WriteFile, which spits data into committed data chunks and updates the "commit" pointer.

If the NtSetInformationFile wasn't there, then whenever the "commit" pointer reached the "reserve" pointer, the filesystem would have to allocate more "reserve" for the file, suffering a massive delay for the journal and the filesystem lock. Since NTFS wants files to be small on disk, this happens depressingly often in a 10MB copy - hence why CopyFile chooses not to do it this way.

Note that because WriteFile isn't a journalled action, any power cut will either leave the file uncreated (before 1), created by zero size (before 2), or created with a correct destination size but with corrupted data (after 2).

If your suggestion of rolling back the length was implemented, you'd suffer all of the same problems, except now you'd lock and journal the filesystem much more often, but still wouldn't protect against the file being corrupted - since midway through the WriteFile after you've committed, say 5MB of the 10MB file the power could go and you'll be left with a 5MB commit with 4.5MB of data in it, instead of a 10MB file with 4.5MB of data in it.

Note that from the user's perspective both of these are equivalent: The user asked you to copy a file, he took the stick out too early, and he ended up with a file on his USB stick that he can't open in MS Word when he gets to work.

Evan

4 Jan 2013 11:06 AM
#

@digiowl: "meh, starting to think that USB drives should have a mandatory button with a led. When you want to remove the drive, you press the button and the led starts flashing. When it goes from flashing to solid you can remove the drive."

Mine has that. Of course, the button is in software, and instead of a flashing LED it pops up a little balloon that says it's now safe to remove. :-)

Nawak
4 Jan 2013 11:23 AM
#

@Matt and alegr1: thanks, yes, the "commit" field presence answers my question about reading old data. I was quite sure that there was something to prevent it, but it wasn't spelt out in the doc of NtSetInformationFile, hence my question.

@Matt: Why couldn't NtSetInformationFile also allow just modifying the "reserve" field? (With another flag, also keeping the current behaviour)

Then the scenario would be:

1. CreateFile => (reserve=X, commit=0, filesize=0) (X: unknown, implementation detail)

2. NtSetInformationFile => (reserve>=FinalSize, commit=0, filesize=0)

3a. WriteFile => (reserve>=FinalSize, commit=WriteSize, filesize=WriteSize)

3b. WriteFile => (reserve>=FinalSize, commit=WriteSize*2, filesize=WriteSize*2)

etc.

This would still leave the strangeness of incomplete data and complete size, but only for the last write.

This is still solvable, even with WriteFile being unjournalled.

This is how I envision WriteFile's (simplified) operations:

A) Allocate space if necessary (update of "reserve" possible)

B) Write data into committed space

C) Update "commit" and "filesize" if necessary (if the write enlarged the file)

Now, without being a full journaled FS, NTFS could do these operations in this order (just to maintain the coherency that Nawak needs so much) but I understand that a good optimization is to do A+C in one disk write, and since you wouldn't want to lose track of sectors, you cannot order this way "(A1)find space->(B)write data->(A2+C)update reserve&commit&filesize". So that optimization forces the order to A+C->B (ie, the order which can lead to what I called an "incorrect" behaviour)

But when A isn't necessary (space has already been allocated), the order could still be B->C instead of C->B with the same performance. Is it worth the added complexity? My opinion would have been yes but the NTFS team already has answered no!

PS:

I hope my answers don't show up as stubbornness. It is a subject I find very interesting and I take the opportunity of this blog post (and its comments) to clear my misconceptions! ;-)

You always learn a lot by reading rebuttals of your solutions!

Matt
4 Jan 2013 12:22 PM
#

@Nawak: There's four reasons:

1. "Reserve" isn't part of the quota system, length is. Consequently allowing people to have very large reserves (i.e. consuming lots of disk) with trivial quantities of quota being committed could be used maliciously.

2. "Reserve" is an implementation detail. In the current system NTFS (and other filesystems - NtSetInformationFile isn't NTFS specific) is allowed to ignore hints via NtSetInformationFile. If you make it contractual you start making life harder for future implementors who then have to be compatible with the concept of a "reserve" when it might not make sense to (e.g. network drives have no such concept). Although NTFS doesn't currently do this, it could in principle automatically reclaim "reserve" from files that don't need it. If the user specified that they wanted X amount of reserve and NTFS then did reclaim it, are you in breach of the API contract now?

3. You can't switch B and C in any model. The inode writeback always happens after the data has completed, otherwise there's a possibility that you'll update the commit length past where data has been written. If a power cut happens then and you remount you'll see garbage data at the end of the file, which might be part of another user's super-secret document. Note that this means that when there's a powercut, data you've written might "become" garbage, but garbage will never "become" data in your file. Also filesize is always less than reserve (by the law of NTFS) and commit only changes in a journaled way, so the filesystem remains consistent in the even of a powercut - but the data in your file might not.

4. Most importantly, your solution doesn't actually present itself to the user as more intuitive or less annoying: Consider the two systems (the status quo and your suggestion) in the case where a 100MB transfer is interrupted at 4.5MB:

In the status quo, the file is reported as 100MB. 100MB worth of actual space on the device is allocated to the file, and when you read it you get 4.5MB of good data and 95.5MB of zeroes. The user is angry because his presentation will not open in Powerpoint (i.e. "stupid windows corrupted my file").

In your suggestion, the file is reported as 4.5MB. 100MB worth of actual space on the device is allocated to the file, and when you read it you get 4.5MB of good data and then the file ends. The user is still angry because his presentation still won't open in Powerpoint (i.e. "stupid windows corrupted my file"). Even more annoyingly, when he tries to copy a 450MB file next to it, he'll get a "disk full" error because 100MB is committed to the other file; but the user only thinks 4.5MB is on the USB stick. "Stupid Windows. I have a 512 MB disk - why can't I put 450+4.5 = 454.5MB worth of files on it!".

Matt
4 Jan 2013 12:31 PM
#

To clarify in (2): NtSetInformationFile isn't a hint when setting file length (it is an instruction to change the file-length that must not be ignored). What's a hint is that the filesystem can use this opportunity to update the commit for the file. Currently it doesn't have to if it doesn't want to.

Also (4) is particularly insidious if a program allocates a 2GB reserve for a file and then crashes before it can put data in (e.g. your browser doing a big download and then crashes). When you start wondering why your disk is running low, no program - including the quota helper - will be able to help you - since all of these huge files are being reported by the filesystem as being really small! At least if you force the commit to be near the filesize you are able to detect these big corrupted files pretty quickly and can delete them if you no longer want them.

jas88
4 Jan 2013 2:17 PM
#

@configurator: "Not that I know why you'd need contiguous blocks on a thumb drive"

Two reasons: first, it's still more efficient to read a single large block than multiple small ones (on the computer level, at least; within the drive itself, thanks to wear-levelling etc, it probably ends up as a series of non-consecutive blocks anyway); secondly, NTFS happens to encode data locations on disk as (start,length) tuples, so "10Mb starting from 0x100" is much more compact than "1 block at 0x100, 1 block at 0x102, 3 blocks at 0x105, 1 block at 0x101". (There is also a free space bitmap, one bit per cluster,

@Paul Coddington: "Still, not as bad as the techies who once insisted that NT machines should always use FAT on their system disk "as recommended by Microsoft" (like heck it was)."

They'd probably misinterpreted some of the early documentation about ARC (non-x86) systems requiring a FAT partition to boot from (x86 BIOS just executes the first 8k of the NTFS partition, which is $Boot, the boot loader; ARC boot loaders are a bit more complex, hence need a whole file system they can understand). The journalling point, I actually recall early IBM demos showing off their shiny new journelled AIX system with "see, you don't need to do a proper shutdown, just pull the plug, because it's journalled" - of course, in reality that takes a lot more than just a journalled file system to be a sane move to make.

@Nawak: Because of NTFS's extent-based structure, you could theoretically write the metadata for your file as "this 10Gb file consists of 10Gb starting at sector 0x100", then go ahead and write the whole 10 Gb without touching that metadata again. (Or rather, to avoid exposing junk data, first you write out the 10 Gb, then you store that location in the Master File Table, then mark those data clusters and the MFT record as being occupied; with write-barriers to enforce ordering, removal or power failure before step 2 means there is no file; after step 2, the recovery process can set the usage bitmaps and index files appropriately when mounting, if needed.)

alegr1
4 Jan 2013 3:26 PM
#

Ultimately, on graceful close, the reserved allocation will be truncated to the current file length, even if there was a reservation.

> [*Um, the discussion is about what happens when there is not a graceful close. - Raymond*]

Gabe
4 Jan 2013 5:11 PM
#

Matt: The problem with having a corrupt file with the correct metadata is that it's not obvious that the file bits are corrupt. It's all too easy to see the correct size and timestamp and think it's the same file, only to find out the hard way that it's mostly zeros.

Perhaps a better thing to do is create the file under a different name, and only rename it to the proper name once the copy is complete. Of course this has the problem of what happens if somebody creates a file (or worse, directory) of that name while the copy is in progress.

xpclient
4 Jan 2013 10:19 PM
#

Now here's the thing. On Windows 8, my new 32 GB Transcend flash drive is set to 'Optimize for quick removal' and yet the Format dialog lists NTFS, exFAT and FAT32 as available choices.

xpclient
4 Jan 2013 10:38 PM
#

I checked that same drive on XP/Vista/7/8 and XP blocks NTFS if it's set for Quick Removal and allows NTFS if set for Performance but Vista and later OSes allow formatting it as NTFS irrespective of whether it's set for Quick Removal or Performance. Bug or by design?

ErikF
5 Jan 2013 1:34 AM
#

@Gabe: Short of enforcing some sort of checksum on all the data and metadata, I don't see how you can tell if a file is corrupt simply by looking at its metadata anyways. Sure,

if the file is significantly shorter or longer than you think it should be then there's probably a problem, but often the problem is that the file is missing the last 4KB of a 10MB file.

It is too bad that there is no locking mechanism in the USB physical spec. That would remove a lot of the problems that we are talking about (and magnify other problems, like the "file is in use, therefore preventing removal" error!)

alegr1
5 Jan 2013 6:29 AM
#

Does Transcend present itself as a USB hard drive?

Matt
5 Jan 2013 6:42 AM
#

@xpclient: FAT32 can't be used for drives with more than 4GB. Windows8 will either choose NTFS or FAT32 for thumb drives, and since FAT32 is out of the question, you're stuck with NTFS regardless of how you choose to "optimise" the drive. If Windows8 blocked your drive from being formatted as NTFS and hence at all because you select "optimise for quick removal" or truncated your device to a 4GB FAT device then you'd just complain about that instead.

@gabe: This suggestion doesn't get fixed by using Nawak's suggestion. Suppose CopyFile doesn't set the length up front and does 500 WriteFiles to transfer the data. In this case, NTFS will up the commit, say, 250 times on the file, journalling each time. At each write, NTFS is allowed to up the filesize before the data has reached disk, it just isn't allowed to update the commit before the data reaches the disk. Consequently it is possible that on the 500th WriteFile NTFS journalling ups the reserve, ups the file-length at the same time (i.e. coaleses the inode writeback) and then the power goes (or the disk gets removed) before the data from the final WriteFile reaches the disk.

In this case we're back to the status quo: The user can't open the file in Powerpoint (i.e. it is corrupt), the filesize is the same as the filesize from the file being copied and the attributes are correct. If you open the file on the other computer you'll see zeroes at the end because data writes aren't journalled.

A better way of solving this problem is to ask a different question: "how can a user be sure the CopyFile succeeded?". If we're willing to ignore the fact that devices can also implement caches, the answer is simple: The only way to be sure that the data is still there is to ask NTFS to unmount the drive (to flush everything to the disk), remount it, open the file, read it all out and compare the bytes (or a hash of those bytes).

Now explorer could do this - but doing so would completely tank the performance for the user. Which leads us to the next question: Which would a user prefer: Being able to copy 100MB onto their disk in 2 minutes but require them to unmount the drive, or require the user to wait 5 minutes but not require them to? Bear in mind when answering this question that Microsoft's competitors might go for the 2 minutes solution and if Microsoft goes for the 5 minute one they might be berated in the press for "having slow file copy times" and for destroying your thumb-drives twice as quickly as their competitors.

Klimax
5 Jan 2013 9:57 AM
#

@Matt:

FAT32 has much higher limit for partition size and 4GB limit is for files only.

xpclient
5 Jan 2013 10:05 AM
#

@Matt Do you notice a tone of complaint in my comment? I asked a question. There is still exFAT so from what I read on blogs.msdn.com/.../108205.aspx it can be inferred that NTFS is blocked unless 'Optimize for performance' is set.

alegr1
5 Jan 2013 11:57 AM
#

FAT32 (artificial) volume size limit is 32 GB. Over than that, exFAT should be recommended. Actually, it makes sense to use exFAT for smaller drives, unless you plan to use the drive at systems where exFAT is not inbox, or not installed as update.

>how can a user be sure the CopyFile succeeded?

1. Make sure CopyFile issues FlushFileBuffers for removeable volume.

2. For a removable volume, limit amount of write-back to 1-2 seconds worth of data, and have a short write-back idle delay. This is for applications that don't use CopyFile.

@xpclient
5 Jan 2013 12:34 PM
#

>>> "Do you notice a tone of complaint in my comment?".

Yes. Almost everything you say on Rayomond's blog is phrased in a critical way. Rather than "Bug or feature?" you could have phrased it less agressively, e.g. "why is that?" or "am I missing something?", rather than just assuming that things not working as you expect is because the feature is broken.

f0dder
6 Jan 2013 5:52 PM
#

I was thinking that having an in-memory "eventual filesize hint" for NTFS could have been a nice thing - something that wouldn't be used in the on-disk structures, simply a best-effort thing for the driver to take into consideration while the operation is still running. If the process crashes, file size would be trimmed to the (journalled as normally) FS metadata updated by the last write.

But after pondering a bit:

1) starts at -1000, is probably hard to do right, and doesn't bring much to the table.

2) doesn't solve the "file /contents/ might be corrupt" problem.

2) while (likely) reducing fragmentation, you still get "a zillion" metadata updates, so the copy will be slower than really pre-allocating the output file size.

Bonus question: is there a way to pre-allocating a file size and then writing to an arbitrary location in it, without (NTFS) having to zerofill from the former last-valid location to the current random offset? Last time I played around with that was back on XP, and SetFileValidData() seemed a bit dangerous and privilege-hungry for general purpose. Sparse Files were obviously counterproductive to the goal of avoiding fragmentation :)

Matt
7 Jan 2013 12:52 AM
#

@fodder: "is there a way to pre-allocating a file size and then writing to an arbitrary location in it, without (NTFS) having to zerofill from the former last-valid location to the current random offset?"

Yes. Sparse files.

JamesJohnston
7 Jan 2013 7:56 AM
#

Raymond: "Write-behind caching is disabled by default on removable drives. You get into this mess only if you override the default."

xpclient: "I checked that same drive on XP/Vista/7/8 and XP blocks NTFS if it's set for Quick Removal and allows NTFS if set for Performance but Vista and later OSes allow formatting it as NTFS irrespective of whether it's set for Quick Removal or Performance."

There seems to be an underlying story here which I've somehow missed. Did Windows Vista introduce support for NTFS and Quick Removal? Did prior versions of Windows not support that configuration?

Roman
7 Jan 2013 9:18 AM
#

Raymond, I think the confusion arose because your first sentence says you need to do X to enable formatting as NTFS, while later on you talk about the write-behind cache. Frankly, I'm still confused, even after your clarification :)

On my Win7, I was able to format a USB hard drive as NTFS without making any promises or agreeing to any warning dialogs. I assume the write caching is disabled, based on what you say about that being the default. Therefore, I should be safe to yank it and expect the just-copied *data* to survive too (not just filesystem metadata), right?

> [*If the rules changed in Windows 7, nobody told me about it. (Not that I expected them to.) But the point of discussion is "If I yank a thumb drive before delayed writes are complete, why doesn't journaling fix the problem?" The rules around delayed writes may have changed, but that doesn't affect the discussion. -Raymond*]

f0dder
7 Jan 2013 9:59 AM
#

@Matt: "Yes. Sparse files."

You didn't read my entire post? Let me quote the relevant part: "Sparse Files were obviously counterproductive to the goal of avoiding fragmentation :)"

Nawak
7 Jan 2013 1:42 PM
#

>> Ultimately, on graceful close, the reserved allocation will be truncated to the current file length, even if there was a reservation.

> [Um, the discussion is about what happens when there is not a graceful close

@Raymond: Indeed, but alegr1's remark can apply to the fsck done during the drive remount. Since the last operation didn't finish, the allocated space could be reclaimed. Also, reading back I think that alegr1's remark applied to Matt's second comment about some (apparently small) files' allocation staying very large after a browser crash. In that case, the OS would do the graceful close (from the FS POV) on process exit, wouldn't it?

@Matt:

1) Reserve being part of quota: well, ok, but could as well be! I think including would still be fair (since reserved space isn't available to other users), so another decision I'll have to contest! ;-). Note that I don't advocate having a "reserve" marginally different from the "size" after a close (or remount). This would only be a hint for opened files.

2) Implementation detail? Let's call it optional functionality! If the reservation promise isn't kept then it's only a performance degradation for the offending FS. So I think every FS could return OK without doing anything. Reserve space qualify as a hint and isn't therefore contractual so no API would be broken in your case. If you want to talk about implementation details, being able to fill up a file with zeroes at no cost when resizing it, *that* is an implementation detail! And implementing this could be way harder for another FS than answering OK to a hint it didn't take.

3) You are right. So that's always A->B->C with A being optional. So if you implement a service that allocates and set the reserve, there's no extra cost at all, since you still have to at least update the commit at each write (commit+filesize in my solution, but maybe the informations aren't in the same physical places?)

4) I already acknowledged that. The benefit was that the corruption was more visible.

@Matt and others: Yes, none of what I propose would magically uncorrupt a file from a drive that has been removed before the end of the operation. The initial point was to make it more visible that something went wrong, even if I admit that not everyone knows what size a particular file should be... Wait! No, the start of all this was my surprise of seeing Raymond say that having the final size at the end of the auto-repair was the correct behaviour. I now know that it is indeed correct considering what CopyFile asked. Hilarity ensued when I asked for another service! ;-)

@f0dder:

- I agree that when an FS/API is already designed, coded and tested, that kind of service would have to gain a lot of points to justify the work. Still doesn't make the requested behaviour "incorrect" or wrong to examine.

- for your second "2)": there wouldn't be more metadata updates than there already is. As Matt noted, you still have to change the commit size at each write. If my uninformed opinion is correct, you could update the size in the same write.

Roman
7 Jan 2013 7:04 PM
#

> But the point of discussion is "If I yank a thumb drive before delayed writes are complete, why doesn't journaling fix the problem?"

Oh, I see. This makes sense. May I humbly suggest that the title of this entry is somewhat misleading then? It _sounds_ like you intend to discuss why you can't just yank an NTFS formatted USB drive: note the part where the title seems to assert that "you have to be careful when using [NTFS] with a USB thumb drive" - but you go on to clarify that this only applies if delayed writes are permitted.

Matt
8 Jan 2013 1:06 AM
#

@f0dder: "You didn't read my entire post? Let me quote the relevant part: "Sparse Files were obviously counterproductive to the goal of avoiding fragmentation :)"

Well sparse files are the only way to achieve what you want - and the fragmentation of sparse files across disk is a fairly small problem - ranges are automatically collected and nearly full sparse files are automatically converted to non-sparse files by background tasks in Windows. Sparse files are ordered collections of "ranges" of blocks where blocks outside of those ranges are zero. All other files have a "valid data" pointer where the data before that value holds non-garbage data on disk.

Consequently if you want to write the last 4KB of a 1GB file without writing the other

blocks to disk you have to either use sparse files, or suffer the large write of zeroes back to disk.