The Old New Thing

Why is the file size reported incorrectly for files that are still being written to?

26 Dec 2011 7:00 AM | 28

The shell team often gets questions like these from customers:

> Attached please find a sample program which continuously writes data to a file. If you open the folder containing the file in Explorer, you can see that the file size is reported as zero. Even manually refreshing the Explorer window does not update the file size. Even the `dir` command shows the file size as zero. On the other hand, calling `GetFileSize` reports the correct file size. If I close the file handle, then Explorer and the `dir` command both report the correct file size. We can observe this behavior on Windows Server 2008 R2, but on Windows Server 2003, the file sizes are updated in both Explorer and `dir`. Can anybody explain what is happening?

> We have observed that Windows gives the wrong file size for files being written. We have a log file that our service writes to, and we like to monitor the size of the file by watching it in Explorer, but the file size always reports as zero. Even the `dir` command reports the file size as zero. Only when we stop the service does the log file size get reported correctly. How can we get the file size reported properly?

> We have a program that generates a large number of files in the current directory. When we view the directory in Explorer, we can watch the files as they are generated, but the file size of the last file is always reported as zero. Why is that?

Note that this is not even a shell issue. It's a file system issue, as evidenced by the fact that a `dir` command exhibits the same behavior.

Back in the days of FAT, all the file metadata was stored in the directory entry.

The designers of NTFS had to decide where to store their metadata. If they chose to do things the UNIX way, the directory entry would just be a name and a reference to the file metadata (known in UNIX-land as an *inode*). The problem with this approach is that every directory listing would require seeking all over the disk to collect the metadata to report for each file. This would have made NTFS slower than FAT at listing the contents of a directory, a rather embarrassing situation.

Okay, so some nonzero amount of metadata needs to go into the directory entry. But NTFS supports hard links, which complicates matters since a file with multiple hard links has multiple directory entries. If the directory entries disagree, who's to say which one is right? One way out would be try very hard to keep all the directory entries in sync and to make the `chkdsk` program arbitrary choose one of the directory entries as the "correct" one in the case a conflict is discovered. But this also means that if a file has a thousand hard links, then changing the file size would entail updating a thousand directory entries.

That's where the NTFS folks decided to draw the line.

In NTFS, file system metadata is a property not of the directory entry but rather of the *file*, with some of the metadata replicated into the directory entry as a tweak to improve directory enumeration performance. Functions like `FindFirstFile` report the directory entry, and by putting the metadata that FAT users were accustomed to getting "for free", they could avoid being slower than FAT for directory listings. The directory-enumeration functions report the last-updated metadata, which may not correspond to the actual metadata if the directory entry is stale.

The next question is where and how often this metadata replication is done; in other words, how stale is this data allowed to get? To avoid having to update a potentially unbounded number of directory entries each time a file's metadata changed, the NTFS folks decided that the replication would be performed only from the file into *the directory entry that was used to open the file*. This means that if a file has a thousand hard links, a change to the file size would be reflected in the directory entry that was used to open the file, but the other 999 directory entries would contain stale data.

As for how often, the answer is a little more complicated. Starting in Windows Vista (and its corresponding Windows Server version which I don't know but I'm sure you can look up, and by "you" I mean "Yuhong Bao"), the NTFS file system performs this courtesy replication when the last handle to a file object is closed. Earlier versions of NTFS replicated the data while the file was open whenever the cache was flushed, which meant that it happened every so often according to an unpredictable schedule. The result of this change is that the directory entry now gets updated less frequently, and therefore the last-updated file size is more out-of-date than it already was.

Note that even with the old behavior, the file size was still out of date (albeit not as out of date as it is now), so any correctly-written program already had to accept the possibility that the actual file size differs from the size reported by `FindFirstFile`. The change to suppress the "bonus courtesy updates" was made for performance reasons. Obviously, updating the directory entries results in additional I/O (and forces a disk head seek), so it's an expensive operation for relatively little benefit.

If you really need the actual file size right now, you can do what the first customer did and call `GetFileSize`. That function operates on the actual file and not on the directory entry, so it gets the real information and not the shadow copy. Mind you, if the file is being continuously written-to, then the value you get is already wrong the moment you receive it.

Why doesn't Explorer do the `GetFileSize` thing when it enumerates the contents of a directory so it always reports the accurate file size? Well, for one thing, it would be kind of presumptuous of Explorer to second-guess the file system. "Oh, gosh, maybe the file system is lying to me. Let me go and verify this information via a slower alternate mechanism." Now you've created this environment of distrust. Why stop there? Why not also verify file contents? "Okay, I read the first byte of the file and it returned 0x42, but I'm not so sure the file system isn't trying to trick me, so after reading that byte, I will open the volume in raw mode, traverse the file system data structures, and find the first byte of the file myself, and if it isn't 0x42, then somebody's gonna have some explaining to do!" If the file system wants to lie to us, then *let the file system lie to us*.

All this verification takes an operation that could be done in 2 + *N*/500 I/O operations and slows it down to 2 + *N*/500 + 3*N* operations. And you're reintroduced all the disk seeking that all the work was intended to avoid! (And if this is being done over the network, you can definitely feel a 1500× slowdown.) Congratulations, you made NTFS slower than FAT. I hope you're satisfied now.

If you were paying close attention, you'd have noticed that I wrote that the information is propagated into the directory when the last handle to the *file object* is closed. If you call `CreateFile` twice on the same file, that creates two file objects which refer to the same underlying file. You can therefore trigger the update of the directory entry from another

program by simply opening the file and then closing it.

```
void UpdateFileDirectoryEntry(__in PCWSTR pszFileName)
{
    HANDLE h = CreateFileW(
        pszFileName,
        0,                  // don't require any access at all
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,               // lpSecurityAttributes
        OPEN_EXISTING,
        0,                  // dwFlagsAndAttributes
        NULL);              // hTemplateFile
    if (h != INVALID_HANDLE_VALUE) {
        CloseHandle(h);
    }
}
```

You can even trigger the update from the program itself. You might call a function like this every so often from the program generating the output file:

```
void UpdateFileDirectoryEntry(__in HANDLE hFile)
{
    HANDLE h = ReOpenFile(
        hFile,
        0,                  // don't require any access at all
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        0);                 // dwFlags
    if (h != INVALID_HANDLE_VALUE) {
        CloseHandle(h);
    }
}
```

If you want to update all file directory entries (rather than a specific one), you can build the loop yourself:

```
// functions ProcessOneName and EnumerateAllNames
// incorporated by reference.

void UpdateAllFileDirectoryEntries(__in PCWSTR pszFileName)
{
    EnumerateAllNames(pszFileName, UpdateFileDirectoryEntry);
}
```

Armed with this information, you can now give a fuller explanation of why ReadDirectoryChangesW does not report changes to a file until the handle is closed. (And why it's not a bug in ReadDirectoryChangesW.)

Bonus chatter: Mind you, the file system could expose a flag to a FindFirstFile-like function that means "Accuracy is more important than performance; return data that is as up-to-date as possible." The NTFS folks tell me that implementing such a flag wouldn't be all that hard. The real question is whether anybody would bother to use it. (If not, then it's a bunch of work for no benefit.)

Bonus puzzle: A customer observed that whether the file size in the directory entry was being updated while the file was being written depended on what directory the file was created in. Come up with a possible explanation for this observation.

Bonus reading:

- NTFS misreports free space?
- The four stages of NTFS file growth.

Blog - Comment List MSDN TechNet

Comments

Henke37
26 Dec 2011 7:52 AM
#

When in doubt regarding strange file locking behavior, blame the antivirus application and/or the search indexing service.

Cesar
26 Dec 2011 8:17 AM
#

What I find interesting is: why does Unix not need this optimization? Looking at a popular Unix filesystem (the ext2 family of filesystems), I see that the only things in a directory entry are the inode number, the file name, and optionally the file type (regular file, directory, symbolic link, device, ...). And yet Unix does not seem to have any slowdown because of it.

I believe the real cause is a difference in the API. On Unix, the API to list the contents of a directory (opendir/readdir/closedir) returns the exact same things the filesystem has in the directory entry: inode number, file type, and file name. On the other hand, the Windows API (FindFirstFile/FindNextFile/FindClose) returns much more data (see the WIN32_FIND_DATA struct for the details). That data includes the file size.

The NTFS designers had no choice. The level of detail returned by the Unix directory enumeration API would be a better match to what they wanted to do, but they had to maintain compatibility with a DOS-era API which also returned the file size. They could create a better API, but nobody would use it. Unix systems never had this problem, because the inode was always a separate thing, and software written for Unix already knew it had to stat() every file (for those who do not know Unix, stat() returns much of the inode metadata, including the file size and times), and that stat()ing every file could be expensive.

alegr1
26 Dec 2011 8:30 AM
#

@Henke37:

And this (antivirus) is what caused VB6 (?) disapppearing files debacle, years ago.

And for what it's worth, the search indexing service occasionally interfers with the file deletion and write access. You may have seen once in a while that you cannot delete a file because it's opened by SYSTEM.

Why CreateFile always gets the most recent file size - because all FILE_OBJECT's for an open file (no matter which hardlink was used) refer to the same metadata copy in memory, maintained by the FS driver. This also allows to maintain file sharing and locking semantics, and also APPEND access semantics.

If required, FindFile could be changed to see if there is such a cached metadata record. Though I'm not sure if it's much requested feature.

Mark
26 Dec 2011 10:06 AM
#

Cesar: calling stat on every file is also what causes *nix file managers to be so slow, and why I hate preview/metadata logic in Explorer with a passion.

Joshua
26 Dec 2011 10:20 AM
#

I would have used it, but I ended up calling GetFileInformation myself on every file.

@Cesar: Modern UNIX systems keep the inode physically near the directory on which the file was first created, which ends up being good enough.

Joe
26 Dec 2011 11:03 AM
#

There is another explanation; namely that just because a program is writing data to a file doesn't mean that data is being committed to a disk. Point being that even if the file size was reporting something other than zero on a regular basis, it still wouldn't be accurate for a file that is still being written!

Joshua
26 Dec 2011 11:10 AM
#

@Joe: I use it as something like a progress bar for a program that doesn't normally have it.

Incidentally, this still works on Vista if you use Cygwin's ls. The explanation is left to the reader.

Jonathan
26 Dec 2011 1:05 PM

#

I've always wondered about that behavior. It did bother me somewhat, being that I use a lot of progs whose only progress is in log files. I usually use an implementation of tail, which shows a streaming log.

I'd expect the API to show a coherent picture of the FS, regardless of any lazy-writing or caching effect. When processing FindFirst, I imagine NTFS could lookup whether any dir entry points to a file that's currently open and take the size from there, no disk access needed. Don't know how complex that would be though.

Jan Ringoš
26 Dec 2011 2:05 PM
#

I work in environment where my simplest option to track behavior, communications and data flow, is to view many log files at once (I even wrote myself notepad-like tail, called dynamic log viewer ;-)). The change in NTFS behavior in Server 2008 was slightly inconveniencing at first, but because of it I eventually stopped looking at file sizes and started using more reliable displays. So... positive for me in the end.

640k
26 Dec 2011 5:02 PM
#

Why doesn't the APIs (FFF) use the fresh metadata already in memory instead of physical reading invalid data from a slow device? That should be the first optimization these guys are doing.

Jack Mathews
26 Dec 2011 7:53 PM
#

"Why doesn't the APIs (FFF) use the fresh metadata already in memory instead of physical reading invalid data from a slow device? That should be the first optimization these guys are doing."

Back when the system was designed, memory was at a premium. An ironic statement to a person whose handle is 640k.

Evan
26 Dec 2011 9:56 PM
#

First: Raymond, thank you for an awesome blog entry. This actually may help to explain quite a bit about the FindFirstFileEx API, which I've previously found strange. (In particular, why is there a flag you can pass to suppress returning short file names which is documented to perhaps substantially increase speed, but giving the size is fast

anyway.)

@Cesar: "What I find interesting is: why does Unix not need this optimization? ... And yet Unix does not seem to have any slowdown because of it."

I think it goes beyond what you say to the UI.

A typical mode of looking at the directory contents in Linux is by running 'ls', which doesn't show you the size. In theory, 'ls -l' can be substantially more expensive than a plain 'ls'. (The docs even warn about this potential problem in the context of the flags which tell 'ls' to color the results or mark directories and executable files with / and *. The former can be done on dt_type-supporting systems with just the information from readdir(), but the latter needs a stat().) In contrast, Explorer will often (always?) need the size.

I actually already want, for independent reasons, to run some statistics about how much time the extra lookup costs, so if you're interested in how much difference it makes on Unix to do just a sequence of readdir calls compared to readdir+stat, or how much difference the FindExInfoBasic flag to FindFirstFileEx makes, I'll post back here if I do that investigation and this discussion is still open or you can email me at evaned@gmail.com.

@Joshua: "Modern UNIX systems keep the inode physically near the directory on which the file was first created, which ends up being good enough."

It's not good enough if you're on a networked file system though. I sometimes have 'ls's take several seconds; I have recently started to wonder if the fact that I use the --color flag (which causes those stat()s) is the main cause of that.

Tony
27 Dec 2011 3:45 AM
#

"Now you've created this environment of distrust. Why stop there? Why not also verify file contents?"

It's not an environment of distrust, it's acknowledgement of a technical limitation and working around it to provide 100% accurate results. What's the big deal?

[*That last 1% is pretty expensive, since it increases the cost 1500×. -Raymond*]

Weland
27 Dec 2011 4:50 AM
#

@cesar: Actually, a fairly valid reason why things aren't necessarily *that* slow for most common usage scenarios is that the inode table can be cached. So actually, most of the time, you don't go for a walk around the whole filesystem looking for that annoying inode.

heterodox
27 Dec 2011 7:32 AM
#

No one's taken a swing at the puzzle yet? Okay, I'll step up to the plate (and miss, most likely). Could that happen if a FAT32 partition were mounted in one of the NTFS folders?

Gabe
27 Dec 2011 8:10 AM
#

Raymond, I think the argument is that the last 1% shouldn't cost 1500x as much. Since the incorrect sizes are only for files that are currently open, the correct information should be somewhere in the computer's memory. For every enumerated file, NTFS just has to look in its dictionary of cached files to see if there's more recent metadata. Since that information is probably in RAM, it should be substantially cheaper than 1500x more. I'd expect that this lookup would cost no more than 5% additional.

[*Like I said in the article, "The NTFS folks tell me that implementing such a flag wouldn't be all that hard." -Raymond*]

Gabe
27 Dec 2011 9:02 AM
#

Raymond: If getting the latest metadata costs 150000% more, then a flag for it makes sense (assuming anybody would bother to use it). But if the latest metadata can come from cache instead of hitting the disk (adding, say, only 5%), why bother with a flag -- why not just do the lookup for every enumerated file?

If people really object to the slight perf hit, they could create a registry entry for disabling the lookup, like they did for creating 8.3 filenames and updating last access times.

[*The 1500x cost is if Explorer decides to ignore the file size that comes out of FindFirstFile and gets the current size directly. If it happens inside NTFS, then it would naturally be cheaper. But it would also introduce other weirdness. "When I type* `dir` *I get the correct size of the file as it is being generated, but when the program finishes, the size reverts to the incorrect value again!" (Because the file closure updates the directory entry only for the hard link that was used to open it, and you were observing it through a different hard link.) -Raymond*]

Right & wrong
27 Dec 2011 4:59 PM
#

Right+wrong is better than wrong+wrong.

Two wrongs doesn't make it right, Raymond.

[*It may not be right, but it's what it does. I find it interesting that people who normally freak out when the file system has some weirdo feature to accommodate an old DOS API are asking for the file system to add a weirdo feature to an old DOS API. (As noted in the article, the file system folks say it wouldn't be too hard to add the*

feature, but nobody has asked for it.) -Raymond]

Mark VY
27 Dec 2011 7:08 PM
#

I have a somewhat offtopic question. FAT did not have anything like inodes, and mostly didn't need them, because the directory entry contains everything the inode would have had: where to find the file on disk, file size, etc. This means no support for hard links, which makes me wonder: how are . and .. implemented? In particular, how do you solve the problem you mentioned, of keeping the metadata consistent? A directory can have many directory entries, one for each child, plus one from itself, and one from its parent. I can only think of two solutions. One is that the relevant data is stored in the direcotry itself, instead of as file system metadata. That seems ugly. Another way might be that whenever somebody wants metadata for directory stuff, we look it up in stuff\. and whenever somebody asks for stuff\.., we use stuff\..\. That's not so ugly, but is this what was actually done?

xpclient
27 Dec 2011 7:25 PM
#

The shell does have its own bugs for reporting file size like the infamous Vista status bar bug which misreports size (reports the double size and it keeps multiplying) as many times as you press Refresh. Explorer status bar also is going progressively worse in each version of Windows: social.msdn.microsoft.com/.../69de0cc1-b91b-4fd8-96c3-8299f8ed0488 Hiding useful info under the pretense of some insignificant performance gains has been in-fashion starting with the Vista shell, so users take double the number of steps for simple tasks like viewing size.

alegr1
28 Dec 2011 8:46 AM
#

@Mark:

'.' is implemented by the file path "canonicalization". '..' is an actual entry in the FAT directory.

Mark VY
28 Dec 2011 2:22 PM
#

Ok, cool, but if you have a directory called c:\stuff, and also C:\stuff\things, and someone is in C:\ and asks for the metadata on "stuff", and someone else is in C:\stuff\things, and they ask for the metadata on "..", how will they get the same answer?

alegr1
28 Dec 2011 5:04 PM
#

@Mark:

Win32 tracks the current directory (SetCurrentDirectory). A partial path may go through conversion to a canonical path, by appending a current directory to it. But the filesystem driver might be able to do relative open by a partial path, because OBJECT_ATTRIBUTES includes a handle to the base directory. If you ask for "stuff" while in C:, the base directory in C:\, and the relative path is 'stuff'. If you're in C:\Stuff\things, '..' entry in this directory points to c:\stuff.

Acq
28 Dec 2011 6:05 PM
#

Strangely enough, I observe on XP (I'll let others try other incarnations) that first findfile after boot certainly accesses MFT. It can be very slow, proportional to the number of files in the directory. Once the MFT entries are in the RAM is findfile fast.

Mark VY
28 Dec 2011 6:41 PM
#

@alegr1

So you're saying that it keeps track of the whole path, and to resolve .. it basically does simple string manipulation? And DOS worked this way too? Then why bother having .. appear as a directory entry in the first place, if you don't actually use it? This sounds like it would make resolving .. pretty slow, if you're in C:\dir1\dir2\dir3\dir4\dir5\dir6\dir7\dir8\dir9, and you want .. then it would have to go all the way from c:\ to dir8. I suppose that this is not too slow, and perhaps it is fast enough to seem instantaneous, but it seems a bit wasteful. Or am I misunderstanding?

Gabe
28 Dec 2011 11:42 PM
#

For those curious, here's Part 2 of "NTFS misreports free space":
blogs.msdn.com/.../10.aspx

Neil
30 Dec 2011 3:00 AM
#

So, is FindFirstFile the best way to access this "fast" metadata, for when you don't need all metadata?

arghhhhhhhhhh
30 Dec 2011 1:23 PM
#

I like the original behavior better, imo half of the changes post XP64/Server 2003 are steps backwards. You keep saying "..the file system folks say it wouldn't be too hard to add the feature, but nobody has asked for it...", but who asked for a feature of not showing the file size at all? I seriously doubt someone asked to make something that previously worked "ok" to not work at all.