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

devblogs.microsoft.com/oldnewthing/20111226-00

December 26, 2011

Raymond Chen

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 $2 + N/500 + 3N$ 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:

Raymond Chen

Follow