The Old New Thing

Why does my single-byte write take forever?

22 Sep 2011 7:00 AM | **15**

A customer found that a single-byte write was taking several seconds, even though the write was to a file on the local hard drive that was fully spun-up. Here's the pseudocode:

```
// Create a new file - returns quickly
hFile = CreateFile(..., CREATE_NEW, ...);

// make the file 1GB
SetFilePointer(hFile, 1024*1024*1024, NULL, FILE_BEGIN);
SetEndOfFile(hFile);

// Write 1 byte into the middle of the file
SetFilePointer(hFile, 512*1024*1024, NULL, FILE_BEGIN);
BYTE b = 42;
/ this write call takes several seconds!
WriteFile(hFile, &b, &nBytesWritten, NULL);
```

The customer experimented with using asynchronous I/O, but it didn't help. The write still took a long time. Even using FILE_FLAG_NO_BUFFERING (and writing full sectors, naturally) didn't help.

The reason is that on NTFS, extending a file reserves disk space but does not zero out the data. Instead, NTFS keeps track of the "last byte written", technically known as the *valid data length*, and only zeroes out up to that point. The data past the valid data length are logically zero but are not physically zero on disk. When you write to a point past the current valid data length, all the bytes between the valid data length and the start of your write need to be zeroed out before the new valid data length can be set to the end of your write operation. (You can manipulate the valid data length directly with the SetFileValidData function, but be very careful since it comes with serious security implications.)

Two solutions were proposed to the customer.

Option 1 is to force the file to be zeroed out immediately after setting the end of file by writing a zero byte to the end. This front-loads the cost so that it doesn't get imposed on subsequent writes at seemingly random points.

Option 2 is to make the file sparse. Mark the file as sparse with the FSCTL_SET_SPARSE control code, and immediately after setting the end of file, use the FSCTL_SET_ZERO_DATA control code to make the entire file sparse. This logically fills the file with zeroes without committing physical disk space. Anywhere you actually write gets converted from "sparse" to "real". This does open the possibility that a later write into the middle of the file will encounter a disk-full error, so it's not a "just do this and you won't have to worry about anything" solution, and depending on how randomly you convert the file from "sparse" to "real", the file may end up more fragmented than it would have been if you had "kept it real" the whole time.

Blog - Comment List MSDN TechNet

Comments

ipoverscsi
22 Sep 2011 7:23 AM
#

I didn't realize you had to mark the file as sparse ahead of time; I assumed NTFS did that automatically. After putting a little brain power to it, however, it makes sense that sparse files are optional: you don't want to confuse the developer or break compatibility.

What I find interesting is the document linked to from the fragment "make the file sparse" claims that sparse files are "an inefficient use of disk space". On the surface I find that argument preposterous -- how can NOT using disk be inefficient? -- but I suspect either 1) there are implementation details that make it so; or 2) I am reading too much into the comment. It's probably the latter.

> [*Read the sentence again: "The problem with files that contain sparse data sets is ..., and, because of this, they are an inefficient user of disk space." It's saying that if you have a sparse data set (data that is mostly zero), then normal file storage is inefficient. The concept of sparse files isn't introduced until paragraph three. (Note that automatic sparsification puts you in an overcommit situation, wherein overwriting existing data in a file can generate an "out of disk space" error.) - Raymond*]

Random832
22 Sep 2011 7:37 AM
#

"how can NOT using disk be inefficient?" Other than the fragmentation he mentioned? Well, I can see how you might not get that that makes it use more _space_...

Traditional Unix filesystems (I bring this up because on unix every file _is_ automatically a sparse file under these circumstances - which may well be the reason people assume this to be true on other systems supporting sparse files) contain an explicit address for every block in a file. This means that the metadata for fragmented files takes up no more actual disk space than for non-fragmented files. This isn't generally true of modern filesystems.

keithmo
22 Sep 2011 7:50 AM
#

"The customer experimented with using asynchronous I/O, but it didn't help."

Extending a file in NTFS is always an inherently synchronous operation.

Kyle
22 Sep 2011 8:18 AM
#

The only way to avoid the zeroing out penalty while writing to the middle of a non-sparse file is to have a data structure in the file metadata with that notes which blocks are

completely zero or not. I can't imagine that such a structure would be of overwhelming usefulness, however, as it raises new issues like what happens if the structure gets corrupted, etc.

Joshua
22 Sep 2011 8:44 AM
#

@Kyle: you mean like implementing sparse files by hand.

I suspect this particular customer hadn't realized that you have to explicitly ask for sparse files.

Kyle
22 Sep 2011 9:15 AM
#

@Joshua

Not quite what I mean. It seems like preallocation is what's desired (to bypass the possible disk over commitment), without the zeroing penalty for writing to the middle of the file. As we know, disk space allocation is cheap compared to zeroing out that disk space. My solution was more of one that only Microsoft could implement by changing the NTFS on-disk data structures. Obviously not something to be taken lightly in any way, shape or form. And anyway, as I point out, adding a data structure that explicitly declares some allocated blocks to be zero just asks for massive file corruption since corrupting a localized data structure is much more likely than corruption on a massive scale, say on a multi-gigabyte file.

NT
22 Sep 2011 9:31 AM
#

I thought the answer was going to be that he'd managed to accidentally pass an address as the nNumberOfBytesToWrite parameter, but I bet that's just a typo. ;)

Anonymous Coward
22 Sep 2011 4:42 PM
#

Note that using SetFileValidData is generally not an option, since you need the SE_MANAGE_VOLUME_NAME privilege. Even if you have it, you must make sure that no one else can read the file while you're busy with it, and that when you're done all valid bits of the file have been written. And your program could crash or be killed or there might be a power cut, so the file must be in a location inaccessible without SE_MANAGE_VOLUME_NAME for the duration of the operation. Not for general use indeed.

ErikF
22 Sep 2011 6:25 PM
#

At the point where the code has the comment "This takes several seconds!", if I would start trying to put the pieces together. I've asked Windows to create a 1GB file and write something halfway through. Knowing the average transfer speed of a hard drive, this should be expected (and I should be grateful for caching)! If I don't want to wait for the write to commit, then I should probably be using async calls/completion ports instead, and then take my chances if something happens later on.

It's a good thing that the customer wasn't trying this on a FAT volume or across the network!

[*Indeed, now that you mention it, the customer never noticed that the SetEndOfFile was wicked-fast! That cost has to go somewhere. -Raymond*]

Simon Buchan
22 Sep 2011 6:34 PM
#

I'm assuming SetFileValidData() was invented for the registry, and exposed to user-mode for SQL Server?

@Kyle: I'm assuming NTFS doesn't lazily zero dense files for simplicity - any situation it is useful for is probably one that a sparse file is needed.

@Raymond: Do you think SetEndOfFile() doesn't zero-extend so it can optimize the likely case of sequentially filling the file data?

Simon Buchan
22 Sep 2011 6:34 PM
#

I'm assuming SetFileValidData() was invented for the registry, and exposed to user-mode for SQL Server?

@Kyle: I'm assuming NTFS doesn't lazily zero dense files for simplicity - any situation it is useful for is probably one that a sparse file is needed.

@Raymond: Do you think SetEndOfFile() doesn't zero-extend so it can optimize the likely case of sequentially filling the file data?

Gabe
22 Sep 2011 11:32 PM
#

@Simon: I'd guess that it's actually fairly rare that somebody would call SetEndOfFile()

because they want to add a whole lot of zero-bytes to the end of a file. The most common use cases are probably truncating a file or reserving space for either memory mapping or sequentially filling it with data (like the COPY command).

Neil
23 Sep 2011 3:28 AM
#

I take it there's no option 3, where you set the end of the file (thus committing physical disk space) but then mark it "not quite sparse" so that the zero pages don't have to be written to disk (as if the file was sparse).

Simon Farnsworth
23 Sep 2011 7:22 AM
#

@Neil:

You'd need to redesign NTFS to do that; right now, it stores a list of extents against a file. A portion of the file not covered by any extents reads back as zeroes, as there's no data to provide. You commit disk space when you add an extent to that list; the filesystem then knows that if it reads that bit of physical disk, it will get the right data back.

To fix this, you'd need to add a flag to every extent, to indicate that this is a preallocated extent, and should read as zeros. Every read would have to check that flag, and ensure that it returns zeros instead of reading the disk. The complexity implications of this are high, the benefits are low, and thus it will struggle to get out of the "minus 100" state.

David Walker
23 Sep 2011 8:20 AM
#

@keithmo: That is exactly what the linked Microsoft article says. Did you bother to read it? :-)