Why am I seeing two WRITE requests at the same offset from a single call to WriteFile?

devblogs.microsoft.com/oldnewthing/20221027-00

October 27, 2022

Raymond Chen

A customer was doing a little performance analysis and found an oddity: A single non-extending write request at the application layer was turning into *two* write requests at the I/O layer, both at the same offset:

Op	File	Offset	Length	Flags	Priority	Status
IRP_MJ_WRITE	test.txt	69,632	61,440	Non-cached, Write Through	Normal	SUCCESS
IRP_MJ_WRITE	test.txt	69,632	61,440	Non-cached, Write Through Paging, Synchronous Paging	Normal	SUCCESS

Friend-of-the-blog Malcolm Smith observed that the first write is non-cached. One possibility is that the first write is a flush of previously-dirty data due to a cached write or a writable memory-mapped view. The system then follows up with the second write, which is triggered by the application-level write.

However, if nobody else is writing to the file at the time the test is being run, then that scenario is ruled out.

Another possibility is that the file is compressed. In that case, the application-level write goes into the system cache, and then is flushed. This looks like two write operations from the file system's point of view, which is what the log is watching. But really, only one write is issued to the physical drive.

The customer confirmed that they are writing to a compressed file.

Malcolm explained that NTFS compression is rather expensive.

The idea behind NTFS compression is that the file is broken up into 64KB chunks, with each chunk compressed separately,[1] and each chunk is managed independently.

This means that a simple write operation that isn't a full chunk explodes into a sequence of operations:

- Read the enclosing chunk
- Decompress the enclosing chunk
- Update the uncompressed chunk to incorporate the newly-written data
- Compress the modified chunk
- Find space on the disk for the modified chunk
- Write the modified chunk to disk
- Release the space that the old chunk occupied

One consequence of this is that compressed files are *pathologically fragmented*. The location of each chunk is unlikely to be correlated with the location of any other chunk in the file, especially after a bunch of updating write operations have occurred. Every compressed chunk winds up stored in a random location on the disk.

Furthermore, all this activity entails a lot of updates to the NTFS metadata, which is not just additional work, but it creates additional synchronization bottlenecks. In particular, a write to a compressed file cannot overlap with another write or read to that file, since the write has the metadata lock. For a non-compressed file, non-extending writes can coexist with reads and other non-extending writes, since none of these operations update file location metadata. They're just writing to the sectors that hold the data.

NTFS compression can be used to reduce disk space requirements, but it is not well-suited to data that is constantly being modified. And if you're studying performance issues, compressed files are going to show up as a bottleneck.

The customer thanked Malcolm for his assistance, and noted that they were doing their performance analysis on their development system, not a production system, and that explains the unexpected presence of file compression.

Bonus reading: The Alpha AXP, epilogue: A correction about file system compression on the Alpha AXP.

[1] Or at least, you *hope* that the chunk can be compressed. If you're unlucky, the chunks won't compress, and you went to all this extra effort and got nothing for it.

Raymond Chen

Follow