Disk and File I/O performance with ETW traces: Why is System doing so much stuff?

devblogs.microsoft.com/oldnewthing/20201126-00

November 26, 2020

Raymond Chen

Last time, I shared some preliminary notes on analyzing Disk and File I/O performance with ETW traces. Here are some notes on the mysterious *System* file. (These notes also apply to *Process Monitor*.)

What is this *System* process, and why is it doing a ton of I/O?

The *System* process represents work done in the kernel by drivers, not associated with any particular process. There are a few common reasons for activity in *System*.

One is I/O issued by drivers, such as anti-malware. You can identify at least some of those anti-malware-initiated I/O by looking for I/O issued to the anti-malware's databases or executables.

Another is delayed writes from the disk cache. When an application writes to a file, you'll see a File I/O logged for the application's write operation, but that one usually completes into the disk cache. Dirty data in the disk cache is lazily written back to the drive, and that work gets charged to the *System*.

There's also a counterpart to lazy-writing, and that's prefetching. If the system detects sequential I/O, it will issue a speculative read-ahead from the file into the disk cache. These speculative reads are charged to *System* as well.

This work done by *System* means that a raw tally of file I/O activity may double-count some of the I/O.

Consider an application that reads from a 64KB file as a series of sixteen 4KB reads. If you look at the File I/O operations for that file, you may see the following:

Line #	Process	I/O Type	Offset	Size
1	contoso.exe	Read	0	4,096

2	contoso.exe	Read	4,096	4,096
3	System	Read	4,096	61,440
4	contoso.exe	Read	8,192	4,096
5	contoso.exe	Read	12,288	4,096
6	contoso.exe	Read	16,384	4,096
⋮	⋮	⋮	⋮	⋮
17	contoso.exe	Read	61,440	4,096
Total				126,976

Let's walk through what happened.

At line 1, the application issued a 4KB read to read the start of the file. This read went through normally.

At line 2, the application issued a 4KB read to read the next part of the file. The system realized that the application appears to be doing a sequential read, so it initiated its own 60KB read from the file in anticipation of further reads coming soon. That system-initiated read was logged as line 3.

The reads from lines 2 and 3 were coalesced at the disk layer, and a single 60KB read was issued to the disk (not shown here). The data went into the disk cache, and the first 4KB of the data was also returned to the application.

For rotational media, once you pay for the cost of seeking the disk head to the right spot, the additional cost of a 60KB read over a 4KB read is negligible. You may as well get the extra 56KB while you're already there, since getting there was the hard part.

At line 4, the application issued another 4KB read. The system's speculation paid off, and the read was satisfied from the disk cache.

The same thing happens for lines 5 through 17. These reads were successfully speculated by the system, and they were all satisfied from the disk cache.

This is great: Read-ahead speculation and the disk cache made the application run much faster. But if all you look at is the *Totals*, it looks like we read 124KB of data from the disk: 64KB issued by the application, and another 60KB mysteriously issued by *System*. You might wonder "Why is *System* coming in and issuing all this I/O? Can't the system just leave me alone?"

But now you know: *System* was issuing all this I/O in order to make *your* I/O run faster.

Bonus chatter: There's another category of prefetch which occurs at application launch. The system traces the I/O operations performed by an application when it starts up, and it uses this historical information the next time the application starts up to decide which data to request from the hard drive before allowing the application to start. This serves two purposes: First of all, it gets the data ready before the application requests it. What's more, since it's a bulk request, the disk system can reorder the I/O operations to be more efficient. For example, if an application typically reads a file at offset 0, then offset 327,680, and then offset 4096, the prefetch will issue all the requests at once, and the disk I/O system will probably combine the reads at offset 0 and 4096 together.

Bonus bonus chatter: Yet another category of prefetch comes from *Superfetch*. One of the things that Superfetch does is predict that certain pieces of data are going to be used and use low-priority I/O to get that data into memory ahead of time.

Bonus bonus bonus chatter: If you have a very fast SSD, the system realizes this and turns off Superfetch, ReadyBoot, and the defragmenter.

Raymond Chen

Follow