# I used FILE_FLAG_SEQUENTIAL_SCAN but it didn't seem to speed up my sequential scanning

devblogs.microsoft.com/oldnewthing/20221130-00

November 30, 2022

Raymond Chen

A customer explained that they intended to open a file and read the entire contents in either a single read request or more likely a small number of large reads. Would the `FILE_FLAG_SEQUENTIAL_SCAN` flag be helpful? Or to look at it another way, if the file is going to be read in its entirety, are there situations where `FILE_FLAG_SEQUENTIAL_SCAN` would make things worse?

The customer ran an experiment with a test program that opened a file and read its entire contents, both in small chunks of 16KB and larger chunks of 100MB. They found that the `FILE_FLAG_SEQUENTIAL_SCAN` flag didn't have any significant effect upon the time it took to read the file.

So what's going on?

We saw some time ago that the `FILE_FLAG_SEQUENTIAL_SCAN` flag causes the cache manager to alter its behavior in two ways: First, the amount of prefetch increases, and second, the cache manager more aggressively evicts file data that sit behind the current file pointer.

The test program is not seeing much benefit from the `FILE_FLAG_SEQUENTIAL_SCAN` flag because the test program isn't spending any time processing the data it just read. As soon as the read completes, it just turns around and issues the next read. This means that the prefetch triggered by the `FILE_FLAG_SEQUENTIAL_SCAN` flag didn't get much of a head start. The "prefetch" was not very "pre".

Let's say that the I/O takes 20ms to complete. First, with no prefetching:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms + ϵ | App issues normal read of 64KB (ETA 20ms) |
| T = 30ms + ϵ | I/O completes and app resumes execution |

Now with prefetching:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms | Cache manager issues prefetch read of 64KB (ETA 20ms) |
| T = 10ms + ϵ | App issues normal read of 64KB<br>(fits inside cache manager, so this is a nop) |
| T = 30ms | I/O completes and app resumes execution |
| Net time savings: ϵ | |

On the other hand, if the program spends 10ms processing the data before issuing the next read, then that gives the prefetch read a 10ms head-start.

No prefetching:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms to 20ms | App processes data |
| T = 20ms | App issues normal read of 64KB (ETA 20ms) |
| T = 40ms | I/O completes and app resumes execution |

And with prefetching:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms | Cache manager issues prefetch read of 64KB (ETA 20ms) |
| T = 10ms to 20ms | App processes data |
| T = 20ms | App issues normal read of 64KB<br>(fits inside cache manager, so this is a nop) |
| T = 30ms | I/O completes and app resumes execution |
| Net time savings: 10ms | |

In general, the amount of benefit from the `FILE_FLAG_SEQUENTIAL_SCAN` flag depends on the speed of the device, how much time elapses between the completion of one read request and the initiation of the next one, and whether the next read request fits entirely inside the system-chosen prefetch.

The sequential scan flag didn't hurt, but the test program was so fast that it didn't have much opportunity to help.

Let's look at another case, where the app reads in 128KB chunks. First, with no prefetch:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms to 20ms | App processes data |
| T = 20ms | App issues normal read of 128KB (ETA 35ms) |
| T = 55ms | I/O completes and app resumes execution |

The cost of a 128KB read is slightly less than the cost of two 64KB reads due to the nature of storage media: You don't have the pay the latency of a second request. For rotational media, that latency is in the form of a seek and/or rotational penalty. For solid state media, the latency is in the form of issuing a new I/O request. For network resources, the latency is in the network.

And here's what we get with prefetch enabled:

| | |
|---|---|
| T = 10ms | Previous read completes |
| T = 10ms | Cache manager issues prefetch read of 64KB (ETA 20ms) |
| T = 10ms to 20ms | App processes data |
| T = 20ms | App issues normal read of 128KB<br>(first 64KB fits inside cache manager)<br>(I/O #2 issued for second 64KB) |
| T = 30ms | Prefetch I/O completes; I/O #2 starts (ETA 20ms) |
| T = 50ms | Second I/O completes and app resumes execution |
| Net time savings: 5ms | |

In this case, the savings was only 5ms because the app's read request did not fit entirely inside the prefetch, so a normal fetch had to occur as well.

The `FILE_FLAG_SEQUENTIAL_SCAN` flag still helped, but it didn't help as much as before.

If you know ahead of time that you are going to process the data sequentially, and you are looking to squeeze out every last drop of performance, you can have your program explicitly issue an asynchronous overlapped read of the next block as soon as the previous block completes. It can then process the previous block, and then when it would normally have

issued a read for the next block, it instead just waits for the asynchronous overlapped read to complete. In other words, the production program could manually do what the `FILE_FLAG_SEQUENTIAL_SCAN` flag tries to do automatically, but with the advantage of clairvoyance: It knows exactly how big the next read request will be, and it can issue a read of exactly that size as soon as the previous read completes, so that the I/O system can get a head start on fetching the data before the program needs it.

Raymond Chen

**Follow**