If I issue a second overlapped I/O operation without waiting for the first one to complete, are they still guaranteed to complete in order?

devblogs.microsoft.com/oldnewthing/20160205-00

February 5, 2016

Raymond Chen

A customer had a question about the order in which overlapped I/O will complete.

```
WriteFile(hFile, buffer1, buffer1Length, ..., &overlapped1);
WriteFile(hFile, buffer2, buffer2Length, ..., &overlapped2);
```

Assume that the `hFile` handle is opened as `FILE_FLAG_OVERLAPPED`. Is it guaranteed that `buffer1` will be written to the file before `buffer2`?

The file system team replied that there is no such guarantee. That defeats the point of opening the file as `FILE_FLAG_OVERLAPPED`.

The point of overlapped operations is that you can have multiple operations in flight, and they will be performed in whatever order the I/O subsystem chooses. The second write may be performed before the first if the I/O subsystem thinks that would be faster. For example, maybe the second write can be coalesced with another write to the same sector. Or the disk head happens to be positioned such that seeking to the first buffer position causes it to pass the second buffer position, so the drive figures, “Well, while I’m here, I may as well write out this data, so I don’t have to seek back later.”

The customer clarified. “Our application uses overlapped I/O for performance purposes. Our actual scenario is more complicated than what we wrote, but the basic idea is that we are receiving data and writing it to a file. We require that the data be written to the file in the order issued. Do we have to wait for the first I/O to complete before we issue the second one?”

Yes. If the order in which the operations are performed is important, then you need to serialize them yourself. But assuming that the two writes are to non-overlapping ranges in the file, why do you care what order they are performed? At the end of the day, `buffer1` will be written to the location specified by `overlapped1`, and `buffer2` will be written to the location specified by `overlapped2`.

The customer explained some more: “In both calls to `WriteFile`, the offset is set to `0xFFFFFFFF`FFFFFFFF`, which means that the writes append to the file. Does this change the answer?”

Sort of, but not in a good way. The two I/O operations race into the I/O subsystem, and there’s no guarantee that the first one will reach the I/O subsystem first. That part hasn’t changed. On the other hand, since both operations are writing to the end of the file, the operations will be serialized once they reach the file system, so you are getting the worst of both worlds: Not only are the results unpredictable, you lose parallelism.

Note also that the completion callbacks may be called in an order different from the order in which the operations actually completed. In other words, it’s possible that operation 1 completes before operation 2, but your completion callback for the second operation is called before the completion callback for the first operation. There is no serialization of completion callbacks. They race out of the I/O subsystem the same way that they race in!

Curiously, the customer says that they are using overlapped operations for performance, but then they end up not wanting all the benefits that overlapped operations offer in the first place, namely letting the I/O subsystem reorder operations to improve performance. It’s possible that they read somewhere that overlapped operations offer higher performance, but didn’t understand what that meant. “We pass this flag because the flag means GO FASTER.”

Raymond Chen

Follow