Why does my asynchronous I/O request return TRUE instead of failing with ERROR_IO_PENDING?

devblogs.microsoft.com/oldnewthing/20101008-00

October 8, 2010

Raymond Chen

A customer reported that their program was not respecting the `FILE_FLAG_OVERLAPPED` flag consistently:

> My program opens a file handle in `FILE_FLAG_OVERLAPPED` mode, binds it to an I/O completion callback function with `BindIoCompletionCallback`, and then issues a `WriteFile` against it. I would expect that the `WriteFile` returns `FALSE` and `GetLastError()` returns `ERROR_IO_PENDING`, indicating that the I/O operation is being performed asynchronously, and that the completion function will be called when the operation completes. However, I find that some percentage of the time, the call to `WriteFile` returns `TRUE`, indicating that the operation was performed synchronously. What am I doing wrong? I don't want my thread to block on I/O; that's why I'm issuing asynchronous I/O.

When you specify `FILE_FLAG_OVERLAPPED`, you're promising that your program knows how to handle I/O which completes asynchronously, but it does not require the I/O stack to behave asynchronously. A driver can choose to perform your I/O synchronously anyway. For example, if the write operation can be performed by writing to cache without blocking, the driver will just copy the data to the cache and indicate synchronous completion. Don't worry, be happy: Your I/O completed even faster than you expected! Even though the I/O completed synchronously, all the asynchronous completion notification machinery is still active. It's just that they all accomplished their job *before the* `WriteFile` *call returned*. This means that the event handle will still be signaled, the completion routine will still run (once you wait alertably), and if the handle is bound to an I/O completion port, the I/O completion port will receive a completion notification.

You can use the `SetFileCompletionNotificationModes` function to change some aspects of this behavior, giving some control of the behavior of the I/O subsystem when a potentially-asynchronous I/O request completes synchronously.

Raymond Chen

Follow