I know that an overlapped file handle requires an lpOverlapped, but why does it (sometimes) work if I omit it?

devblogs.microsoft.com/oldnewthing/20120411-00

April 11, 2012

Raymond Chen

A customer observed that the formal requirements for the `ReadFile` function specify that if the handle was opened with `FILE_FLAG_OVERLAPPED`, then the `lpOverlapped` parameter is mandatory. But the customer observed that in practice, passing `NULL` results in strange behavior. Sometimes the call succeeds, and sometimes it even returns (horrors!) valid data. (Actually the more horrifying case is where the call succeeds and returns bogus data!) Now sure, you violated one of the requirements for the function, so the behavior is undefined. But why doesn't `ReadFile` just flat-out fail if you call it incorrectly? The answer is that the `ReadFile` function doesn't know whether you're calling it correctly. The `ReadFile` function doesn't know whether the handle you passed was opened for overlapped or synchronous access. It just trusts that you're calling it correctly and builds an asynchronous call to pass into the kernel. If you passed a synchronous handle, well, it just issues the I/O request into the kernel anyway, and you get what you get. This quirk traces its history all the way back to the *Microsoft Windows NT OS/2 Design Workbook*. As originally designed, Windows NT had a fully asynchronous kernel. There was no such thing as a blocking read. If you wanted a blocking read, you had to issue an asynchronous read (the only kind available), and then block on it. As it turns out, developers vastly prefer synchronous reads. Writing asynchronous code is hard. So the kernel folks relented and said, "Okay, we'll have a way for you to specify at creation time whether you want a handle to be synchronous or asynchronous. And since lazy people prefer synchronous I/O, we'll make synchronous I/O the default, so that lazy people can keep being lazy." The `ReadFile` function is a wrapper around the underlying `NtReadFile` function. If you pass an `lpOverlapped`, then it takes the `OVERLAPPED` structure apart so it can pass the pieces as an `IoStatusBlock` and a `ByteOffset`. (And if you don't pass an `lpOverlapped`, then it needs to create temporary buffers on the stack.) All this translation takes place without the `ReadFile` function actually knowing whether the handle you passed is asynchronous or synchronous; that information isn't available to the `ReadFile` function. It's relying on you, the caller, to pass the parameters correctly. As it happens, the `NtReadFile` function does detect that you are trying to perform synchronous I/O on an asynchronous handle and fails with

`STATUS_INVALID_PARAMETER` (which the `ReadFile` function turns into `ERROR_INVALID_PARAMETER`), so you know that something went wrong. Unless you are a pipe or mailslot. For some reason, if you attempt to issue synchronous I/O on an asynchronous handle to a pipe or mailslot, the I/O subsystem says, "Sure, whatever." I suspect this is somehow related to the confusing no-wait model for pipes.

Long before this point, the basic ground rules for programming kicked in. "Pointers are not `NULL` unless explicitly permitted otherwise," and the documentation clearly forbids passing `NULL` for asynchronous handles. The behavior that results from passing invalid parameters is undefined, so you shouldn't be surprised that the results are erratic.

Raymond Chen

Follow