Why does my synchronous overlapped ReadFile return FALSE when the end of the file is reached?

devblogs.microsoft.com/oldnewthing/20150121-00

January 21, 2015

Raymond Chen

A customer reported that the behavior of `ReadFile` was not what they were expecting.

> We have a synchronous file handle (not created with `FILE_FLAG_OVERLAPPED`), but we issue reads against it with an `OVERLAPPED` structure. We find that when we read past the end of the file, the `ReadFile` returns `FALSE` even though the documentation says it should return `TRUE`.

They were kind enough to include a simple program that demonstrates the problem.

```
#include <windows.h>

int __cdecl wmain(int, wchar_t **)
{
 // Create a zero-length file. This succeeds.
 HANDLE h = CreateFileW(L"test", GENERIC_READ | GENERIC_WRITE,
              0, nullptr, CREATE_ALWAYS,
              FILE_ATTRIBUTE_NORMAL, nullptr);

 // Read past EOF.
 char buffer[10];
 DWORD cb;
 OVERLAPPED o = { 0 };
 ReadFile(h, buffer, 10, &cb, &o); // returns FALSE
 GetLastError(); // returns ERROR_HANDLE_EOF

 return 0;
}
```

The customer quoted this section from The documentation for `ReadFile`:

> Considerations for working with synchronous file handles:
>
> - If *lpOverlapped* is **NULL**, the read operation starts at the current file position and **ReadFile** does not return until the oepration is complete, and the system updates the file pointer before **ReadFile** returns.
> - If *lpOverlapped* is not **NULL**, the read operation starts at the offset that is specified in the **OVERLAPPED** structure and **ReadFile** does not return until the read operation is complete. The system updates the **OVERLAPPED** offset before **ReadFile** returns.
> - When a synchronous read operation reads the end of a file, **ReadFile** returns **TRUE** and sets `*lpNumberOfBytesRead` to zero.

and then added

> According to the third bullet point, the `ReadFile` should return `TRUE`, but in practice it returns `FALSE` and the error code is `ERROR_HANDLE_EOF`.

The problem here is that there are two concepts here, and they confusingly both use the word *synchronous*.

- A synchronous file handle is a handle opened without `FILE_FLAG_OVERLAPPED`. All I/O to a synchronous file handle is serialized and synchronous.
- A synchronous I/O operation is an I/O issued with `lpOverlapped == NULL`.

The sample program issues an asynchronous read against a synchronous handle. The third bullet point applies only to synchronous reads.

To reduce confusion, the documentation would have been clearer if it hadn't switched terminology midstream.

> - If *lpOverlapped* is **NULL**, the read operation starts at the current file position and **ReadFile** does not return until the oepration is complete, and the system updates the file pointer before **ReadFile** returns.
> - If *lpOverlapped* is not **NULL**, the read operation starts at the offset that is specified in the **OVERLAPPED** structure and **ReadFile** does not return until the read operation is complete. The system updates the **OVERLAPPED** offset before **ReadFile** returns.
> - If *lpOverlapped* is **NULL** and the read operation reads the end of a file, **ReadFile** returns **TRUE** and sets `*lpNumberOfBytesRead` to zero.

We asked what the customer was doing that caused them to trip over this confusion in the documentation.

> The customer's original code opened a file (synchronously) and read from it (synchronously). The customer is parallelizing the computation in a way that will read that single file from multiple threads. A single file pointer is therefore not suitable, because different threads will want to read from different positions.
>
> One idea would be to have each thread call `CreateFile` so that each handle has its own file position. Unfortunately, this won't work for the customer because the sharing mode on the file handle denies read sharing.
>
> The solution they came up with was to open the file synchronously (without `FILE_FLAG_OVERLAPPED`) but to read asynchronously (by using an `OVERLAPPED` structure). The `OVERLAPPED` structure lets you specify where you want to read from, so multiple threads can issue reads against the file position they want.
>
> This solution works, but the customer is concerned because this hybrid model is not well-documented in MSDN. They found a blog entry that discusses it, but even that blog entry does not discuss what happens in the multithreaded case.) In particular, they are seeing that the end-of-file behavior acts according to asynchronous rather than synchronous rules.
>
> Any advice you have on how we can pursue this model would be appreciated. Another concern is that since we do not set the `hEvent` in the `OVERLAPPED` structure, the file handle itself is used as the signal that I/O has completed, and this will cause problems if multiple I/O's are active simultaneously.

The problem is that the customer confused the two senses of synchronous, one when applied to files and one when applied to I/O operations. Since they opened a synchronous file handle, all I/O operations are serialized and execute synchronously. Passing an `OVERLAPPED` structure issues an asynchronous I/O, but since the underlying handle is synchronous, the I/O is serialized and synchronous. The customer's code therefore is not actually performing I/O asynchronously; its requests for asynchronous I/O is overridden by the fact that the underlying handle is synchronous.

The hybrid model doesn't actually realize any gains of asynchronous I/O. The use of the `OVERLAPPED` structure merely provides the convenience of combining the seek and read operations into a single call. Since the benefit is rather meager, the hybrid model is not commonly used, and consequently it is not covered in depth in the documentation. (The facts are still there, but there is relatively little discussion and elaboration.)

Based on this feedback, the customer considered switching to using an asynchronous file handle and setting the `hEvent` in the `OVERLAPPED` structure so that each thread can wait for its specific I/O to complete. In the end, however, they decided to stick with the hybrid model because switching to an asynchronous handle was too disruptive to their code base. They are satisfied with the `OVERLAPPED` technique that lets them perform the equivalent of an atomic `SetFilePointer` + `ReadFile` (even if the I/O is synchronous and serialized).

Raymond Chen

Follow