When you create an object with constraints, you have to make sure everybody who uses the object understands those constraints

devblogs.microsoft.com/oldnewthing/20100414-00

April 14, 2010

Raymond Chen

Here's a question that came from a customer. This particular example involves managed code, but don't let that distract you from the point of the exercise.

> I am trying to create a `FileStream` object using the constructor that takes an `IntPtr` as input. In my .cs file, I create the native file handle using `CreateFile`, as shown below.
>
> ```
> [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
> internal static extern IntPtr CreateFile(string lpFileName,
>     int dwDesiredAccess, FileShare dwShareMode,
>     IntPtr securityAttrs, FileMode dwCreationDisposition,
>     UInt32 dwFlagsAndAttributes, IntPtr hTemplateFile);
> IntPtr ptr1 = Win32Native.CreateFile(FileName, 0x40000000,
>          System.IO.FileShare.Read | System.IO.FileShare.Write,
>          Win32Native.NULL,
>          System.IO.FileMode.Create,
>          0xa0000000, // FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH
>          Win32Native.NULL);
> ```
>
> Then I create the `FileStream` object as so:
>
> ```
> FileStream fs = new FileStream(ptr1, FileAccess.Write, true, 1, false);
> ```
>
> The `fs` gets created fine. But when I try to do:
>
> ```
> fs.Write(msg, 0, msg.Length);
> fs.Flush();
> ```
>
> it fails with the error "IO operation will not work. Most likely the file will become too long or the handle was not opened to support synchronous IO operations."
>
> ```
> int hr = System.Runtime.InteropServices.Marshal.GetHRForException(e)
> ```
>
> Gives `hr` as `COR_E_IO` (`0x80131620`).
>
> The stack trace is as below.
>
> ```
> System.IO.IOException: IO operation will not work. Most likely
>     the file will become too long or the handle was not opened
>     to support synchronous IO operations.
> at System.IO.FileStream.WriteCore(Byte[] buffer, Int32 offset, Int32 count)
> at System.IO.FileStream.FlushWrite()
> at System.IO.FileStream.Flush()
> at PInvoke.Program.Main(String[] args)
> ```
>
> Can somebody point out what might be going wrong?

(For those who would prefer to cover their ears and hum when the topic of managed code arises, change `FileStream` to `fdopen`.)

The comment on the line

```
0xa0000000, // FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH
```

was provided by the customer, and that's the key to the problem. It was right there in the comment, but the customer didn't understand the consequences.

As the documentation for `CreateFile` notes, the `FILE_FLAG_NO_BUFFERING` flag requires that all I/O operations on the file handle be in multiples of the sector size, and that the I/O buffers also be aligned on addresses which are multiples of the sector size.

Since you created the file handle with very specific rules for usage, you have to make sure that everybody who uses it actually follows those rules. On the other hand, the `FileStream` object doesn't know about these rules. It just figures you gave it a handle that it can issue normal synchronous `ReadFile` and `WriteFile` calls on. It doesn't know that you gave it a handle that requires special treatment. And then the attempt to write to the handle with a plain `WriteFile` fails both because the number of bytes is not a multiple of the sector size and because the I/O buffer is not sector-aligned, and you get the I/O exception.

The solution to this problem depends on what you are trying to accomplish. Why are you passing the `FILE_FLAG_NO_BUFFERING | FILE_FLAG_WRITE_THROUGH` flags? Are you doing this just because you overheard in the hallway that it's faster? Well, yes it may be faster under the right circumstances, but in exchange for the increased performance, you also have to follow a much stricter set of rules. And in the absence of documentation to the contrary, you can't assume that a chunk of code actually adheres to your very special rules.

Like *What if two people did this?*, this is an illustration of another principle that many people forget to consider when working with objects they didn't write: *When you write your own code, do you do this?* It's sort of like the Golden Rule of programming.

Suppose you have a function which accepts a file handle and whose job is to write some data do that file handle. Do you write your function so that it performs all its I/O in multiples of the sector size from buffers which are aligned in memory in multiples of the sector size, on the off chance that somebody gave you a handle that was opened with the `FILE_FLAG_NO_BUFFERING` flag? Well, no, you don't. You just call `WriteFile` to write to it, and if you want to write 28 bytes, you write 28 bytes. Even if you perform internal buffering and your buffer size happens to be a multiple of the sector size by accident, you still don't align your I/O buffer to the sector size; and when it's time to flush the final partially-written buffer, you have a not-sector-multiple write at the very end anyway.

If you don't handle this case in your code, why would you expect others to handle it in their code?

We've seen this principle before, such as when we looked at whether the `Process.Refresh` method refreshes an arbitrary application's windows.

Raymond Chen

Follow