A comparison of various implementations of the Windows Runtime IMemoryBuffer

devblogs.microsoft.com/oldnewthing/20240129-00

January 29, 2024

Raymond Chen

In my studies of the `IMemoryBuffer` interface, I found three implementations of that interface in the Windows Runtime.

- `Windows.Foundation.MemoryBuffer`, obtained from `Buffer.CreateMemoryBufferOverIBuffer()`.
- `Windows.Graphics.Imaging.BitmapBuffer`, obtained from `SoftwareBitmap.LockBuffer()`.
- Unnamed class obtained from `PerceptionFrame.FrameData`.

We also wrote our own fourth implementation, which we called `CustomMemoryBuffer`, that lets you turn any block of memory into a `MemoryBuffer`.

All four of them behave differently. Let's compare.

	Memory-Buffer	**Bitmap-Buffer**	**Frame-Data**	**Custom-Memory-Buffer**
Thread-safe?	No	Yes	Yes	Yes
IMemoryBuffer supports IMemoryBufferByteAccess?	No	No	Yes	Yes
CreateReference after Close	Empty			
Empty references raise Closed event?	Yes	No	No	Yes
Raises Closed event automatically when released?	Yes	No	Yes	Yes

Can extend lifetime during Closed event handler	No	Yes	No	Yes
Buffer valid during Closed event?	Yes	No	No	Yes
Can call methods during Closed event	Yes	Yes	No	Yes
Buffer of empty or closed reference	pointer = `nullptr` and size = 0			
Memory freed when…	IMemoryBuffer and all IMemoryBufferReferences have been closed or destructed			

All happy memory buffers look alike. Each unhappy memory buffer is unhappy in its own way.

The standard `MemoryBuffer` has the problem of not being thread-safe. If you call `Close` at the same time as `CreateReference`, you may experience use-after-free bugs. And if you call `Close` twice simultaneously, you can add to your woes null pointer crashes, over-release of the underlying `IBuffer`, and double-raising of the the `Closed` event, depending on exactly how the race plays out.

All four implementations agree that if you call `CreateReference` on a closed `IMemoryBuffer`, you get an “empty reference”. An empty reference is one that protects no memory. If you ask for the buffer of an empty reference, you get a null pointer and a size of zero.

In all of the implementations except `FrameData`, empty references raise the `Closed` event.

The `BitmapBuffer`‘s memory buffer reference raises the `Closed` event only on an explicit call to `Closed`. The others raise the `Closed` event either on explicit closure or when the last reference is released. This means that `BitmapBuffer` reference’s `Closed` event is even more unreliable than the `Closed` event already is by its nature.

The `MemoryBuffer` and `FrameData` ignore attempts by the `Closed` event handler to extend the reference’s lifetime. The biggest consequence of this is that the `Closed` event in those implementations will corrupt memory if consumed from a GC language. The `BitmapBuffer` sneakily passes this test because it is masked by the other defect of simply not raising the `Closed` event in the dangerous scenario in the first place.

The `BitmapBuffer` and `FrameData` raise the `Closed` event after freeing the memory, which means that the event is useless for triggering cleanup: Since you are told that the memory has been freed only after it happened, all you're really learning is that "Oops, you already corrupted memory."

The `FrameData` has the bonus insult of passing you an `IMemoryBufferReference` in the `Closed` handler that cannot be used! Any attempt to obtain the buffer's capacity or pointer will hang. (That's because it raises the `Closed` event while still holding its internal lock. Calling to outside code while holding a lock is a bad idea for reasons like this.)

Our `CustomMemoryBuffer` tries to avoid all of these little defects.

But what if you are forced to use one of the other three implementations of `IMemoryBuffer`, or some other fifth implementation from an external source that isn't even on the list. Seeing as the first three attempts at implementing `IMemoryBuffer` all failed in different ways, what confidence do you have that an unknown implementation will be well-behaved?

We'll solve this problem next time. The answer is right under our nose.