The dangerous implementations of the IMemoryBuffer-Reference.Closed event

devblogs.microsoft.com/oldnewthing/20240124-00

January 24, 2024

Raymond Chen

Last time, we noted that the `IMemoryBufferReference.Closed` event is useless. Which is a good thing, because every implementation I've looked at is also wrong.

The problem comes with the need to raise the `Closed` event when the final reference is released. The implementations I've looked at all raise the event in the object's destructor as a "last gasp":

```
MemoryBufferReference::~MemoryBufferReference()
{
    m_refCount = 1; // avoid double-destruction

    Close();
}

void MemoryBufferReference::Close()
{
    if (!m_notified.exchange(true)) {
        m_closed.Invoke(this, nullptr);
        m_buffer = nullptr;
    }
}
```

We are giving out COM references to an object from its destructor, and as we noted when we discussed this trick earlier, this assumes that none of the functions who are given these COM references will call `AddRef` and retain the pointer after the call returns. (They can call `AddRef`, provided they also call `Release` the same number of times before returning.)

The problem here is that the `Closed` event does not come with any guarantee that the handler will honor this constraint. And in fact, you have a reverse guarantee: If the handler is written in C#, you will almost certainly *not* have balanced `AddRef` and `Release` calls. That's because the common language runtime (CLR) wraps inbound COM references inside a so-called runtime-callable wrapper (RCW). The runtime-callable wrapper retains a reference to

the underlying COM object, which means that the COM reference gets `AddRef`'d when it is placed inside an RCW. When the RCW gets garbage-collected, the finalizer for the RCW does a `Release` on the backing COM reference.

What will happen is that the `Closed` event handler will receive two RCWs, one for each of the event handler parameters, and the RCW will hang around after the handler returns. The `IMemoryBufferReference` object destructs despite being `AddRef`'d. Eventually, those RCWs will get cleaned up by the garbage collector, and they will `Release` the wrapped COM pointer. Unfortunately, that pointer is now pointing to freed memory because the object destructed way back when the `Closed` event was raised.[1]

Even if your `Closed` handler ignores its parameters, the CLR will still create RCWs for them, because it doesn't know that your handler ignores its parameters. Its job is to pass the two parameters to the handler, and by golly, it will pass the two parameters to the handler.

The only winning move is not to play: Never subscribe to the `Closed` event.

Fortunately, you shouldn't be tempted to subscribe to it anyway, since we also learned that the event is useless.

[1] Here's a test program I wrote to prove that it crashes. You have to run it in Release mode so that the necessary optimizations kick in.

```
using Buffer = Windows.Storage.Streams.Buffer;

void Test()
{
    bool done = false;
    Buffer.CreateMemoryBufferOverIBuffer(new Buffer(10)).
        CreateReference().Closed += (s, _) => done = true;

    // Force GC's until the MemoryBuffer's RCW runs down
    while (!done)
    {
        GC.Collect();
    }
    // The Closed event handler created a new RCW for the
    // MemoryBuffer. Force another GC to run it down, which
    // will crash.
    GC.Collect();
}
```