Smoothing over the differences (and defects) in the various implementations of IMemoryBuffer

devblogs.microsoft.com/oldnewthing/20240130-00

January 30, 2024

Raymond Chen

We saw last time that each unhappy implementation of `IMemoryBuffer` is unhappy in its own way. How can you avoid tripping over all of these differences and defects?

Fortunately, all of the implementations satisfy the following minimum requirements:

- The underlying memory is freed when the `IMemoryBuffer` and all `IMemoryBufferReference`s have been closed or destructed.
- The objects are reliable provided you call only one method at a time.

We can operate within these minimum requirements and treat it as an external memory buffer that is wrapped inside our `CustomMemoryBuffer`.

```
winrt::array_view<uint8_t> GetView(
    winrt::IMemoryBufferReference const& reference)
{
    uint8_t* buffer;
    uint32_t size;
    winrt::check_hresult(reference.as<
        ABI::Windows::Foundation::IMemoryBufferByteAccess>()->
        GetBuffer(&buffer, &size));
    return { buffer, size };
}
```

We start with a generally useful function that obtains the buffer behind an `IMemoryBufferReference` and returns it in the form of an `array_view<uint8_t>`.

```
// Takes ownership of the IMemoryBufferReference
winrt::IMemoryBuffer WrapAsMemoryBuffer(
    winrt::IMemoryBufferReference const& reference)
{
    return CreateCustomMemoryBuffer(
        GetView(reference),
        [reference]
        {
            reference.Close();
        });
}
```

The `WrapAsMemoryBuffer` method takes an `IMemoryBufferReference` and wraps it inside our `CustomMemoryBuffer`. We call `GetView` only once, and it never happens concurrently with the `Close` of the buffer, so we avoid any multithreaded race conditions.

Basically, we treat `IMemoryBufferReference` as just another source of memory with a cleanup function. That the memory source happens to be the same family as the wrapper we are producing is just a coincidence.

```
// Does not take ownership of the IMemoryBuffer
winrt::IMemoryBuffer WrapMemoryBuffer(
    winrt::IMemoryBuffer const& buffer)
{
    return WrapMemoryBuffer(buffer.CreateReference());
}
```

This overload of `WrapMemoryBuffer` uses an `IMemoryBuffer` as the source. It just creates a reference from the `IMemoryBuffer` and then wraps that reference.

Note that the `IMemoryBuffer` overload does not take ownership of the `IMemoryBuffer`, since it never closes it. This is a weird asymmetry that is bound to cause confusion. Maybe it should close the `IMemoryBuffer`?

```
// Takes ownership of the IMemoryBuffer
winrt::IMemoryBuffer WrapMemoryBuffer(
    winrt::IMemoryBuffer const& buffer)
{
    auto reference = buffer.CreateReference();
    buffer.Close();
    return WrapMemoryBuffer(reference);
}
```

Alternatively, we can ask WIL to close the buffer.

```
// Takes ownership of the IMemoryBuffer
winrt::IMemoryBuffer WrapMemoryBuffer(
    winrt::IMemoryBuffer const& buffer)
{
    auto close = wil::scope_exit([&] { buffer.Close(); });
    return WrapMemoryBuffer(buffer.CreateReference());
}
```

But maybe the function name in both cases should be something like `WrapMemoryBufferAnd-TakeOwnership`? I'm not sure. You can decide.