Accessing a block of memory represented by a Windows Runtime IMemoryBuffer

devblogs.microsoft.com/oldnewthing/20240122-00

January 22, 2024

Raymond Chen

A customer had a large block of memory that they wanted to expose as a Windows Runtime object. They could have used a `Windows.Storage.Streams.IBuffer`, but the problem with buffers is that they make a copy of the data, and the customer didn't want the waste of copying large blocks of memory around.

An alternative to `IBuffer` is `IMemoryBuffer`. This interface is used to represent a block of memory with a custom lifetime policy.

There are two pieces to the `IMemoryBuffer` pattern. First, you have the `IMemoryBuffer` itself, which is the currency that is used to represent these blocks of memory. And then you have the `IMemoryBufferReference`, which is a ticket that grants *access to* the data.

From the consumer's point of view, the usage pattern goes like this.

1. Somebody gives you an `IMemoryBuffer` that represents a block of memory.
2. Anybody who wants to access the memory block calls call `IMemoryBuffer.CreateReference` to obtain a reference to the data in the form of an `IMemoryBufferReference`, and then from the reference, uses `IMemoryBufferByteAccess.GetBuffer` to obtain the raw pointer to the data.
3. When each consumer is finished using the pointer, allow the `IMemoryBufferByteAccess` to destruct or use `IClosable` to close the reference prior to destruction.
4. Repeat the above two steps as many times as you like.
5. The memory is freed when all `IMemoryBuffer` and `IMemoryBufferReference` objects are closed or allowed to destruct.
6. Note: If you close the `IMemoryBuffer`, then future calls to `CreateReference` return a reference to an empty buffer.

The mental model is that you have some main object that controls the memory lifetime. You can ask that main object for permission to access the memory, and it gives you a `IMemory-BufferReference` as the token that represents that permission. The token's `IMemoryBuffer-`

`ByteAccess.GetBuffer` method gives you a pointer to that data. The token is valid until it is closed or destructed. And the memory block itself is valid until the `IMemoryBuffer` and all outstanding references are closed or destructed.

```
// C++/WinRT

void ProcessDataFromMemoryBuffer()
{
    IMemoryBuffer buffer = ObtainMemoryBufferFromSomewhere();

    IMemoryBufferReference reference = buffer.CreateReference();

    // Obtain a pointer to the data the hard way.
    uint8_t* data;
    uint32_t capacity;
    winrt::check_hresult(reference.as<IMemoryBufferByteAccess>()
        ->GetBuffer(&data, &capacity));

    ⟦ access 'capacity' bytes starting at 'data' ⟧

    // Destruction of "reference" variable will release the
    // IMemoryBufferReference and close the reference.
    // But we can do it explicitly if we like.
    reference.Close();

    // Destruction of "buffer" variable will release the
    // IMemoryBufferReference and close the reference.
    // But we can do it explicitly if we like.
    buffer.Close();
}
```

C++/WinRT provides the `data()` extension method on `IMemoryBufferReference` that does the `GetBuffer` for you and returns a raw pointer to the underlying memory. We can therefore simplify the above example to this:

```
void ProcessDataFromMemoryBuffer()
{
    IMemoryBuffer buffer = ObtainMemoryBufferFromSomewhere();

    IMemoryBufferReference reference = buffer.CreateReference();

    uint8_t* data = reference.data();
    uint32_t capacity = reference.Capacity();

    ⟦ access 'capacity' bytes starting at 'data' ⟧
}
```

The story in C++/CX is a bit complicated due to the need to interoperate between C++/CX's hat pointers and traditional COM.

```
// C++/CX

void ThrowIfFailed(HRESULT hr)
{
    if (FAILED(hr)) {
        throw Exception::CreateException(hr);
    }
}

void ProcessDataFromMemoryBuffer()
{
    IMemoryBuffer^ buffer = ObtainMemoryBufferFromSomewhere();

    IMemoryBufferReference^ reference = buffer->CreateReference();

    // Obtain a pointer to the data.
    ComPtr<IMemoryBufferByteAccess> access;
    ThrowIfFailed(reinterpret_cast<IInspectable*>(reference)
                    ->QueryInterface(IID_PPV_ARGS(&access)));

    uint8_t* data;
    uint32_t capacity;
    ThrowIfFailed(access->GetBuffer(&data, &capacity));

    ⟦ access 'capacity' bytes starting at 'data' ⟧

    // Destruction of "access" and "reference" variables will
    // release the IMemoryBufferReference, respectively.
    // But we can do it explicitly if we like.
    access = nullptr;
    delete reference; // this calls IClosable::Close()

    // Destruction of "buffer" variable will release the
    // IMemoryBuffer, but we can close it explicitly if we like.
    delete buffer; // this calls IClosable::Close()
}
```

In C#, you have to import the COM interface and drop to unsafe code to access the raw pointers.

```
[ComImport]
[Guid("fbc4dd2d-245b-11e4-af98-689423260cf8")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public unsafe interface IMemoryBufferByteAccess {
  void GetBuffer([Out] byte** buffer, [Out] uint* capacity);
}

unsafe void ProcessDataFromMemoryBuffer()
{
  // "using" statements ensure that the objects are disposed.
  using (IMemoryBuffer buffer = ObtainMemoryBufferFromSomewhere())
  using (IMemoryBufferReference reference = buffer.CreateReference()) {
    // Obtain a pointer to the data.
    byte* data;
    uint capacity;
    ((IMemoryBufferByteAccess)reference).GetBuffer(&data, &capacity);

    ⟦ access 'capacity' bytes starting at 'data' ⟧
  }
}
```

C# 8.0's *using declaration* lets you avoid the nesting.

```
unsafe void ProcessDataFromMemoryBuffer()
{
  using IMemoryBuffer buffer = ObtainMemoryBufferFromSomewhere();
  using IMemoryBufferReference reference = buffer.CreateReference();

  // Obtain a pointer to the data.
  byte* data;
  uint capacity;
  ((IMemoryBufferByteAccess)reference).GetBuffer(&data, &capacity);

  ⟦ access 'capacity' bytes starting at 'data' ⟧
}
```

Next time, we'll look at part of the `IMemoryBufferReference` that is broken and should be avoided.