How can I expose a pre-existing block of memory as a Windows Runtime object without copying the data?

devblogs.microsoft.com/oldnewthing/20240126-00

January 26, 2024

Raymond Chen

Let's implement an `IMemoryBuffer`. The basic idea is that the `IMemoryBuffer` object controls the memory, and it grants access to the memory by handing out objects which implement `IMemoryBufferReference`.

Here's an implementation using C++/WinRT. We start with the `MemoryLifetime`.

```
namespace winrt
{
    using namespace winrt::Windows::Foundation;
}

struct MemoryLifetime
{
    using MemoryCleanupHandler = winrt::DeferralCompletedHandler;

    MemoryLifetime(
        winrt::array_view<uint8_t> view,
        MemoryCleanupHandler const& cleanup)
    : m_view(view)
    {
        m_cleanup.add(cleanup);
    }

    ~MemoryLifetime()
    {
        m_cleanup();
    }

    // Not copyable, not assignable.
    MemoryLifetime& operator=(MemoryLifetime const&) = delete;
    MemoryLifetime(MemoryLifetime const&) = delete;

    winrt::array_view<uint8_t> m_view;
    winrt::event<MemoryCleanupHandler> m_cleanup;
};
```

The `MemoryLifetime` object represents a block of memory that is cleaned up at destruction by the provided `MemoryCleanupHandler` delegate. All not-yet-closed `IMemoryBuffer` and `IMemory-BufferReference` objects retain a strong reference to the `MemoryLifetime`.

I pull a couple of sneaky tricks in dealing with the delegate. First, I reuse the `Deferral-CompletedHandler`, since it is a delegate for `void()`, which is what we want too.

Second, I store the delegate in a `winrt::event` rather than as a delegate directly. I'm taking advantage of a few features of `winrt::event`:

- It detects delegates which are not agile and puts them in an agile wrapper so that we can raise the event from any thread.
- It catches exceptions that are thrown from the delegate, which is good because any uncaught exception in a destructor terminates the process because destructors default to `noexcept`.

The `MemoryLifetime` is kept in a `shared_ptr`. This next class helps us manage that pointer.

```
struct MemoryLifetimeTracker
{
    MemoryLifetimeTracker(std::shared_ptr<MemoryLifetime> lifetime)
        : m_lifetime(std::move(lifetime)) {}

    std::shared_ptr<MemoryLifetime> Lifetime()
    {
        auto lock = winrt::slim_shared_lock_guard(m_srwlock);
        return m_lifetime;
    }

    winrt::array_view<uint8_t> GetView()
    {
        auto lock = winrt::slim_shared_lock_guard(m_srwlock);
        return m_lifetime ? m_lifetime->m_view
                          : winrt::array_view<uint8_t>{};
    }

    // For IMemoryBufferByteAccess
    HRESULT GetBuffer(uint8_t** buffer, uint32_t* size) noexcept
    {
        auto view = GetView();
        *buffer = view.data();
        *size = view.size();
        return S_OK;
    }

    std::shared_ptr<MemoryLifetime> Reset()
    {
        auto lock = winrt::slim_lock_guard(m_srwlock);
        return std::exchange(m_lifetime, {});
    }

private:
    winrt::slim_mutex m_srwlock;
    std::shared_ptr<MemoryLifetime> m_lifetime;
};
```

The code in `Reset()` to clean up the `m_lifetime` is tricky because we must hold the lock in order to access `m_lifetime`, but we don't want `MemoryLifetime`'s destructor to run from inside the lock, because we don't know what sorts of shenanigans the cleanup delegate will get up to, and we don't want to hold the lock across what could be a very long and dangerous function. So we exchange the shared pointer while under the lock, and then return it. The caller will then allow the shared pointer to destruct, outside the lock. (It's okay for the `MemoryLifetimeTracker` to destroy the shared pointer without a lock. There are no conflicting threads at that point.)

The next piece is the `IMemoryBufferReference`. This is the most complicated part.

```
struct CustomMemoryBufferReference :
    winrt::implements<
        CustomMemoryBufferReference,
        winrt::IMemoryBufferReference,
        winrt::IClosable,
        ::Windows::Foundation::IMemoryBufferByteAccess>
{
    using ClosedEventHandler = winrt::TypedEventHandler<
        winrt::IMemoryBufferReference, winrt::IInspectable>;

    static_assert(!outer(), "Must not be composable.");

    CustomMemoryBufferReference(
        std::shared_ptr<MemoryLifetime> const& lifetime)
    : m_tracker(lifetime)
    {
        NonDelegatingAddRef();
    }

    uint32_t Capacity()
    {
        return m_tracker.GetView().size();
    }

    STDMETHOD(GetBuffer)(uint8_t** buffer, uint32_t* size)
        noexcept override
    {
        return m_tracker.GetBuffer(buffer, size);
    }

    decltype(std::declval<implements>().Release())
        __stdcall Release() noexcept override
    {
        auto count = NonDelegatingRelease();
        if (count == 1)
        {
            count = Close(count);
        }
        return count;
    }

    winrt::event_token Closed(ClosedEventHandler const& handler)
    {
        return m_closed.add(handler);
    }

    void Closed(winrt::event_token token)
    {
        m_closed.remove(token);
    }
```

```
    uint32_t Close(uint32_t count = 0)
    {
        if (!m_notified.exchange(true, std::memory_order_relaxed))
        {
            m_closed(*this, nullptr);
            m_tracker.Reset();
            count = NonDelegatingRelease();
        }
        return count;
    }

    MemoryLifetimeTracker m_tracker;
    std::atomic<bool> m_notified;
    winrt::event<ClosedEventHandler> m_closed;
};
```

The `CustomMemoryBufferReference` is constructed with a shared pointer to a `MemoryLifetime` that gives us access to the underlying memory.

We follow the general pattern of giving away a COM reference just before the object destructs, but since the cleanup can also be explicitly triggered via `Close()`, we put the “notified” flag in the `Close()` method.

If we call `Close()` as part of the final application-visible `Release()`, we want to return the revised reference count so that it’s easier to debug the application by observing the return value of `Release()` to figure out whether that was the final `Release()`. We pass the original reference count as a parameter, and if the `Close()` method raises the `Closed` event, then it returns the revised reference count.

if the `Close()` method is called via the projection, it is done with no parameters, so the count parameter defaults to zero. Furthermore, the projected `Close()` is void, so our `uint32_t` return value is ignored. (We are taking advantage of C++/WinRT’s use of CRTP.)

The last piece is the `CustomMemoryBuffer`.

```
struct CustomMemoryBuffer :
    winrt::implements<
        CustomMemoryBuffer,
        winrt::IMemoryBuffer,
        winrt::cloaked<winrt::IMemoryBufferByteAccess>,
        winrt::IClosable>
{
    using MemoryCleanupHandler = winrt::DeferralCompletedHandler;

    CustomMemoryBuffer(
        winrt::array_view<uint8_t> view,
        MemoryCleanupHandler const& cleanup)
    : m_lifetime(std::make_shared<MemoryLifetime>(view, cleanup))
    {
    }

    // IMemoryBuffer
    winrt::IMemoryBufferReference CreateReference()
    {
        return winrt::make<CustomMemoryBufferReference>(
            m_tracker.Lifetime());
    }

    // IMemoryBufferByteAccess
    STDMETHOD(GetBuffer)(uint8_t** buffer, uint32_t* size)
        noexcept override
    {
        return m_tracker.GetBuffer(buffer, size);
    }

    // IClosable
    void Close() { m_tracker.Reset(); }

    MemoryLifetimeTracker m_tracker;
};

template<typename T>
winrt::IMemoryBuffer CreateCustomMemoryBuffer(
    winrt::array_view<T> view,
    winrt::DeferralCompletedHandler const& cleanup)
{
    auto byte_view = winrt::array_view(
        reinterpret_cast<uint8_t*>(view.data()),
        view.size() / sizeof(T));
    return winrt::make<CustomMemoryBuffer>(byte_view, cleanup);
}

inline winrt::IMemoryBuffer CreateCustomMemoryBuffer(
    void* buffer, uint32_t size,
    winrt::DeferralCompletedHandler const& cleanup)
{
```

```
    return CreateCustomMemoryBuffer(
        { reinterpret_cast<uint8_t*>(buffer), size },
        cleanup);
}
```

The `CustomMemoryBuffer` is our implementation of `IMemoryBuffer`. You create it from an `array_view` and a handler that is called when all outstanding references have been released or closed. We also provide a convenience overload for `void*` buffers.

Here’s an example usage of our implementation:

```
winrt::IMemoryBuffer
    CreateSharedMemoryBuffer(uint32_t size)
{
    winrt::handle mapping =
        winrt::check_pointer(
            CreateFileMappingW(INVALID_HANDLE_VALUE,
                nullptr, PAGE_READWRITE, 0, size, nullptr)) };
    auto view = winrt::check_pointer(
        MapViewOfFile(mapping.get(), FILE_MAP_WRITE, 0, 0, size));
    return CreateCustomMemoryBuffer(view, size, [view]
        {
            winrt::check_bool(UnmapViewOfFile(view));
        });
}
```

We create and map an unnamed file mapping and create a `CustomMemoryBuffer` around that block of memory, with a cleanup delegate that unmaps the view.

We’ll come back to this helper class later after we look at some other implementations.