A practical use for GetHGlobalFromStream when sharing was never your intention

devblogs.microsoft.com/oldnewthing/20211115-00

November 15, 2021

Raymond Chen

A little while ago, I noted that managing shared access to the `HGLOBAL` inside a stream can get tricky, and opined that the `GetHGlobalFromStream` should have been something like `IStreamOnHGlobal::DetachHGlobal`. But in fact there's a straightforward use case for `GetHGlobalFromStream` even when there is no sharing going on.

The `STGMEDIUM` structure is a currency for passing data into and out of an `IDataObject`. it is basically a discriminated union:

```
struct STGMEDIUM
{
    DWORD tymed;
    union
    {
        HBITMAP hBitmap;
        HMETAFILEPICT hMetaFilePict;
        HENHMETAFILE hEnhMetaFile;
        HGLOBAL hGlobal;
        LPOLESTR lpszFileName;
        IStream* pstm;
        IStorage* pstg;
    };
    IUnknown* pUnkForRelease;
};
```

The `tymed` member specifies which of the members of the union is active. Let's focus on the case where the value is `TYMED_MWBR>HGLOBAL`, in which case the structure simplifies to

```
struct STGMEDIUM
{
    DWORD tymed = TYMED_HGLOBAL;
    HGLOBAL hGlobal;
    IUnknown* pUnkForRelease;
};
```

When you are finished with the `STGMEDIUM`, you call `ReleaseStgMedium`, and in the case of a `TYMED_HGLOBAL` the rule for `ReleaseStgMedium` is

- If `pUnkForRelease` is not null, then call `pUnkForRelease->Release();`.
- If `pUnkForRelease` is null, then call `GlobalFree(hGlobal);`.

In words, if there is a `pUnkForRelease`, then it is the "owner" which controls the lifetime of the `HGLOBAL`, and you tell it that you are done by releasing the reference count on the owner. When the owner's reference count goes to zero, it destroys itself (and the `HGLOBAL`).

On the other hand, if there is no `pUnkForRelease`, then the `HGLOBAL` is "ownerless", and you just free it when you're done.

The idea here is that if the `HGLOBAL` was expensive to produce, your data object may decide to cache the result rather than having to produce it from scratch every time. In that case, when you give out the data, you can do this:

```
if (m_cachedHglobal == nullptr)
{
    RETURN_IF_FAILED(Calculate(&m_cachedHglobal));
}

pstgm->tymed = TYMED_HGLOBAL;
pstgm->hGlobal = m_cachedHglobal;
pstgm->pUnkForRelease = this;
this->AddRef(); // because we put it in pUnkForRelease
```

When the recipient of the data is finished, they will call `ReleaseStgMedium`, and `ReleaseStgMedium` will see that there is a non-null `pUnkForRelease` and instead of freeing the `HGLOBAL`, it'll release the `pUnkForRelease`.

This means that the `m_cachedHglobal` is not destroyed when the recipient of the data is finished. It lives on, so it can be returned to another client.

Finally, when the data object is destroyed, it also destroys the `m_cachedHglobal`.

This pattern means that as long as anybody still has a `STGMEDIUM` referring to your cached `HGLOBAL`, your entire data object will remain alive. But maybe your data object has a lot of stuff in it, like an entire HTML DOM, and the `HGLOBAL` is a cache of the `textContent`. Somebody who asks for the `textContent` and hangs onto it for a long time will keep your data object alive, even if they aren't using the data object any more:

```
HRESULT GetTheText(STGMEDIUM* pstgm)
{
    wil::com_ptr<IDataObject> pdto;
    RETURN_IF_FAILED(GetTheDataObject(&pdto));

    FORMATETC fe;
    fe.cfFormat = CF_UNICODETEXT;
    fe.ptd = nullptr;
    fe.dwAspect = DVASPECT_CONTENT;
    fe.lindex = -1;
    fe.tymed = TYMED_HGLOBAL;

    RETURN_IF_FAILED(pdto->GetData(&&fe, pstgm));

    return S_OK;
}
```

This function gets the data object and extracts the Unicode text. The data object is thrown away when the `com_ptr` destructs, and all that remains is the text in the `STGMEDIUM`'s `HGLOBAL`.

The catch here is that the caller of this function might decide to keep the text for a long time, and that's going to keep your big data object around for a long time. Even though all that really needs to be kept alive is the text.

This is where you can use one of the lesser powers of `GetHGlobalFromStream`.

Instead of making the data object be the cache for the `HGLOBAL`, you can make an `HGLOBAL`-backed stream with `CreateStreamOnHGlobal`, and let the stream be the one in charge of the `HGLOBAL`'s lifetime.

```
if (m_cachedStream.get() == nullptr)
{
    wil::unique_hglobal text;
    RETURN_IF_FAILED(Calculate(&text));
    RETURN_IF_FAILED(CreateStreamOnHGlobal(
        text.get(), TRUE /* fDeleteOnRelease */,
        &m_cachedStream));
    text.release(); // m_cachedStream owns it now
}

pstgm->tymed = TYMED_HGLOBAL;
RETURN_IF_FAILED(GetHGlobalFromStream(
        m_cachedStream.get(), &pstgm->hGlobal));
// The stream is the owner of the HGLOBAL
pstgm->pUnkForRelease = m_cachedStream.copy().detach();
```

This time, the owner of the `HGLOBAL` is the `m_cachedStream`, and therefore if the storage medium is retained beyond the life of the data object, the data object can destruct, and the `m_cachedStream` will deal with freeing the `HGLOBAL` on final release.

I'm guessing this might even have been the scenario for which `GetHGlobalFromStream` was originally invented.

Mind you, we're using an entire stream just to babysit an `HGLOBAL`. We could have written a custom babysitter:

```
struct IUnknownOnHGLOBAL : winrt::implements<IUnknownOnHGLOBAL, ::IUnknown>
{
    IUnknownOnHGLOBAL(HGLOBAL glob) : m_glob(glob) {}
    wil::unique_hglobal glob;
};
```

On the other hand, using `CreateStreamOnHGlobal` may end up being the easier route if the `HGLOBAL` was originally generated from a stream in the first place:

```
if (m_cachedStream.get() == nullptr)
{
    wil::com_ptr<IStream> stm;
    RETURN_IF_FAILED(CreateStreamOnHGlobal(
        nullptr, TRUE /* fDeleteOnRelease */,
        &stm));
    RETURN_IF_FAILED(SaveToStream(stm.get()));
    m_cachedStream = std::move(stm);
}

pstgm->tymed = TYMED_HGLOBAL;
RETURN_IF_FAILED(GetHGlobalFromStream(
        m_cachedStream.get(), &pstgm->hGlobal));
pstgm->pUnkForRelease = m_cachedStream.copy().detach();
```

Raymond Chen

Follow