The subtleties of CreateStreamOnHGlobal, part 4: Non-movable memory

devblogs.microsoft.com/oldnewthing/20211001-00

October 1, 2021

Raymond Chen

Last time, I noted that some people ignore the requirement that the memory block passed to `CreateStreamOnHGlobal` be movable. What happens next?

The documentation is a bit of two minds on this. On the one hand, it says that the memory must be movable. On the other hand, it also talks about how you need to be careful if the memory is not movable (and how using movable memory is strongly recommended). I suspect this ambivalence comes from a compatibility constraint: Although the function is documented as requiring movable memory, in practice, people have provided fixed memory and have relied upon particular implementation details that let them get away with it, so the system has to let you continue to get away with it, at least to the extent that you got away with it before.

So what happens if you ignore the admonitions and pass fixed memory anyway?

The underlying stream doesn't realize that the memory block is fixed and continues operating under the assumption that it is movable. It still dutifully locks and unlocks the memory block around each access, even though locking and unlocking fixed memory has no effect.

The scariest change is when the underlying memory block needs to be reallocated, because reallocation of a fixed block behaves very differently from reallocation of a movable block. Reallocating a movable block will not change the memory handle, not even if the underlying memory moves to a new address. If the underlying memory moves, that the next attempt to lock the memory block returns the *new* address.

On the other hand, reallocating a fixed memory block does change the handle if the underlying memory moves, because fixed memory blocks use the memory address as the handle. This means that when a reallocation happens, the internal `HGLOBAL` changes!

If you have multiple streams sharing the same `HGLOBAL`, but which aren't clones of each other, then the consequences of the `HGLOBAL` changing address are quite dire. It means that once one stream reallocates the `HGLOBAL`, all the other streams are left with a dangling

pointer, which means a use-after-free bug and a security hotfix soon thereafter.

As I noted, the solution here is not to create multiple streams from the same `HGLOBAL`, but rather to create a single `HGLOBAL`-based stream from the `HGLOBAL`, and then clone that stream. Clones of an `HGLOBAL`-based stream are aware of each other and can coordinate their actions.

But wait, we're not done yet.

Since reallocation changes the `HGLOBAL`, it means that if a reallocation occurs, then the `HGLOBAL` you passed when you created the stream got freed out from under you!

```
HGLOBAL hglob = GetInitialHGlobal();
IStream* stream;
CreateStreamOnHGlobal(hglob, FALSE, &stream);
WriteToStream(stream); // ← hglob may no longer be valid!
```

If you're sticking with the whole `fDeleteOnRelease = FALSE` thing, then you need to check with the stream just before it is destructed, "Hey, so what's the `HGLOBAL` you're managing right now?" In other words, "If you were to be destructed *right now*, what `HGLOBAL` would you free?"

You need to ask this question immediately before the final `Release` of the stream. Asking any sooner risks the possibility that the stream resizes again after you asked, and the answer you got is no longer valid.

The hard part is knowing exactly when to ask this question. It's not like the stream lets you know that it's about to be destroyed. You have to keep your eye on the stream and make sure it never lands in the hands of somebody who is going to extend its lifetime.

Personally, I would limit my use of `fDeleteOnRelease = FALSE` to the case of creating a brand new empty stream, writing data into the stream (being careful not to give to anybody who will extend the lifetime), and then extracting the `HGLOBAL` from it.

And even that is pretty risky. I would probably use an alternate approach of leaving `fDeleteOnRelease = TRUE` and letting the stream itself control the lifetime of the `HGLOBAL`:

```
// Non-RAII version to make everything explicit.
IStream* stream = nullptr;

if (SUCCEEDED(
    CreateStreamOnHGlobal(nullptr, TRUE, &stream))) {

    WriteStuffToStream(stream);

    // Lifetime of the HGLOBAL is controlled by the stream.
    HGLOBAL hglob = nullptr;
    if (SUCCEEDED(GetHGlobalFromStream(stream, &hglob))) {
        DoStuffWith(hglob);
    }

    // This Release call frees the HGLOBAL
    stream->Release();
}
```

Bonus chatter: If I were sent back in time to make small design changes to `CreateStreamOnHGlobal` with the benefit of hindsight, I think I would have done something like this:

```
HRESULT CreateStreamOnHGlobal(HGLOBAL hGlobal, REFIID riid, void** ppv);
```

Creating a stream always takes ownership of the `hGlobal`. This avoids the problem of an object that intentionally leaks memory.

```
interface IStreamOnHGlobal : IUnknown
{
    HRESULT DetachHGlobal(HGLOBAL* result);
};
```

Instead of the `GetHGlobalFromStream` function, we use this interface that can be queried from the stream. Using an interface avoids undefined behavior if somebody calls `GetHGlobalFromStream` with a stream that didn't come from `CreateStreamOnHGlobal`. The introduction of this interface explains why I changed the signature of `CreateStreamOnHGlobal` to let you specify the output interface.

The interface method `DetachHGlobal` behaves differently from `GetHGlobalFromStream`. The method performs an ownership transfer, rather than sharing ownership. Once you detach the `HGLOBAL` from a stream, it cannot be put back, and any future stream operations return an error (perhaps `RO_E_CLOSED` from the future).

This avoids the problem of a stream that has had its `HGLOBAL` freed out from under it: At no point is ownership of the `HGLOBAL` ever shared. It belongs clearly to somebody at all times. If you want to access the `HGLOBAL`, you have to take ownership of it.

Raymond Chen

Follow