Kernel handles are not reference-counted

devblogs.microsoft.com/oldnewthing/20070829-00

August 29, 2007

Raymond Chen

Here's a question that floated past some time ago:

> In my code, I have multiple objects that want to talk to the same handle (via `DeviceIoControl`). Each time I create an object, I use `DuplicateHandle` to increment the reference count on the handle. That way, when each object calls `CloseHandle`, only the last one actually closes the handle. However, when I run the code, I find as soon as the first object calls `CloseHandle`, the handle is no longer valid and nobody else can use it. What flags do I need to pass to `CreateFile` to get this to work?

In other words, the code went something like this:

```
// h is the handle that we want to share with a new CFred object
CFred *MakeFred(HANDLE h)
{
 // "Duplicate the handle to bump the reference count"
 // This code is wrong - see discussion
 // All error checking removed for expository purposes
 HANDLE hDup;
 DuplicateHandle(GetCurrentProcess(), h,
                 GetCurrentProcess(), &hDup,
                 0, FALSE, DUPLICATE_SAME_ACCESS);
 return new CFred(h);
}
```

Kernel handles aren't reference-counted. When you call `CloseHandle`, that closes the handle, end of story.

From the original problem statement, we know that the `CFred` object closes the handle when it is destroyed. Just for argument's sake, let's say that the caller goes something like this:

```
CFred *pfred1 = MakeFred(h);
CFred *pfred2 = MakeFred(h);
delete pfred1;
delete pfred2;
```

What actually happens when you run this fragment?

The first time we call `MakeFred` we take the original handle `h` and duplicate it, but we give the original handle to the `CFred` constructor and leak the `hDup`! The original poster assumed that duplicating a handle merely incremented the handle's imaginary reference count, so that `h == hDup`. (Which would also have made the original poster wonder why we even bother having a `lpTargetHandle` parameter in the first place.)

When `pfred1` is deleted, it closes its handle, which is `h`. This closes the `h` handle and renders it invalid and available to be recycled for another `CreateFile` or other operation that creates a handle.

When `pfred2` is deleted, it also closes its handle, which is still `h`. This is now closing an already-close handle, which is an error. If we had bothered calling a method on `pfred2` that used the handle, it would have gotten failures from those operations as well, since the handle is no longer valid. (Well, if we're lucky, we would have gotten a failure. If we were unlucky, the handle would have been recycled and we ended up performing a `DeviceIoControl` on somebody else's handle!)

Meanwhile, the calling code's copy of `h` is also bad, since `pfred1` closed it when it was deleted.

What we really want to do here is duplicate the handle and pass the **duplicate** to each object. The `DuplicateHandle` function creates a new handle that refers to the same object as the original handle. That new handle can be closed without affecting the original handle.

```
// h is the handle that we want to share with a new CFred object
CFred *MakeFred(HANDLE h)
{
 // Create another handle that refers to the same object as "h"
 // All error checking removed for expository purposes
 HANDLE hDup;
 DuplicateHandle(GetCurrentProcess(), h,
                 GetCurrentProcess(), &hDup,
                 0, FALSE, DUPLICATE_SAME_ACCESS);
 return new CFred(hDup);
}
```

The fix is one word, highlighted in blue. We give the duplicated handle to the `CFred` object. That way, it gets its own handle which it is free to close any time it wants, and it won't affect anybody else's handle.

You can think of `DuplicateHandle` as a sort of `AddRef` for kernel objects. Each time you duplicate a handle, the reference count on the kernel object goes up by one, and you gain a new reference (the new handle). Each time you close a handle, the reference count on the kernel object drops by one.

In summary, a handle is not a reference-counted object. When you close a handle, it's gone. When you duplicate a handle, you gain a new obligation to close the duplicate, in addition to the existing obligation to close the original handle. The duplicate handle refers to the same object as the original handle, and it is the underlying object that is reference-counted. (Note that kernel objects can have reference from things that aren't handles. For example, an executing thread maintains a reference to the underlying thread object. Closing the last handle to a thread will not destroy the thread object because the thread keeps a reference to itself as long as it's running.)

Raymond Chen

Follow