Using DuplicateHandle to help manage the ownership of kernel handles

devblogs.microsoft.com/oldnewthing/20161020-00

October 20, 2016

Raymond Chen

A customer was using a third party I/O library that also gave you access to the underlying `HANDLE`, in case you needed it. The customer needed that underlying `HANDLE` so it could pass it to `_open_osfhandle()` and get a C runtime file descriptor, which could then be used by other C runtime functions to operate on the I/O object.

Everything was great until it came time to close the I/O object, because both the I/O library and the C runtime tried to close the handle. and that resulted in assertion failures due to invalid handles.

The problem here is that both the I/O library and the C runtime think that they are responsible for closing the handle. The I/O library wants to close the handle because it created the handle in the first place, and the special method to obtain that underlying `HANDLE` wasn't transferring ownership of the handle to you; it merely gave you a handle that you could borrow. On the other hand, the `_open_osfhandle()` function will close the handle when the file descriptor is closed, because the function assumes that you're giving it not only the handle, but also the responsibility to close the handle.

Neither library has a way to change the handle semantics. There isn't a way to tell the I/O library or the C runtime, "Hey, don't close that handle."

The solution here is to use the `DuplicateHandle` function to create a brand new handle that refers to the same underlying kernel object. You can then pass the duplicate to `_open_osfhandle()`. Both the I/O library and the C runtime library will close their respective handles. Since each handle is closed exactly once, balance is restored to the universe.

Exercise (easy): Suppose you have a C runtime file descriptor, and you want to take the underlying kernel handle and give it to another library, which will close the handle you give it. How do you manage this without running into a double-close bug?

Exercise (slightly harder): Suppose your program needs more than 2048 C runtime file descriptors, which is more than `_setmaxstdio` accepts. Fortunately, your program doesn't actively use all of the descriptors at the same time, so you're thinking that you can virtualize the file descriptors by "paging them in" and "paging them out" from underlying kernel handles. How would you do this?

Raymond Chen

Follow