How can I investigate the possibility of a lot of leaked window classes (RegisterClass)?

devblogs.microsoft.com/oldnewthing/20171006-00

October 6, 2017

Raymond Chen

A customer reported that they are seeing intermittent failures from the `RegisterClass` function with the error `ERROR_NOT_ENOUGH_MEMORY` on some systems, typically after their program and other programs related to their program have been running for some time. They suspect that there is a leak somewhere caused by failure to unregister window classes, but they need help tracking this down.

> We thought we could inspect the system to see how many classes have been registered indirectly via the following pseudocode:
>
> ```
> Foreach running processId
>   Foreach atom in (0xC000 .. 0xFFFF)
>     If (GetClassInfo(HInstanceFromProcessId(processId), atom))
>       Inspect the returned class name
> ```
>
> But when we run this, we get a constant set of used atoms for all processes, and we can't see the class name. Is there something else that we can use to help track down this leak?

Okay, there are a few things wrong with the pseudocode above.

First of all, instance handles are meaningful only in the context of a process, and unless a specific process is specified (which is very rare), they are interpreted as in the context of the current process. So the call to `HInstanceFromProcessId` is not all that useful because it gives you an instance handle you can't use. What you end up doing is taking the instance handle from that other process and then checking if anybody in the *current* process with the same instance handle has registered a class with the specified atom.

Second, the `GetClassInfo` function gets information about the class which is registered against the provided instance/name pair. In order to get through all the classes that belong to a process, you have to enumerate through not only all the atoms, but also all the instance handles. And you have to include all the instances of modules that have ever been loaded (even if they were subsequently unloaded). This is not the sort of thing you can easily brute-force your way through, seeing as a module can be loaded at almost any 64KB boundary.

Fortunately, all is not lost.

It so happens that in current versions of Windows, registered class names, registered clipboard format names, and registered window message names all come from the same atom table. I wish to reiterate that this is an implementation detail which can change at any time, so don't take any dependencies on this. I provide this information for diagnostic purposes, which is what we have here.

The customer can do this once they encounter the problem:

```
Foreach atom in (0xC000 .. 0xFFFF)
  If (GetClipboardFormatName(atom, buffer, bufferSize))
      Print buffer
```

This will print out the names of all the classes, clipboard formats, and registered window messages. There is room in the atom table for 16,384 atoms, and in practice there aren't more than a hundred or so, so if you see more than 15,000 entries, that's a really good sign that you're leaking classes.

Leaking classes is generally hard to do because the class name is typically something you hard-code into your program, and it takes a lot of work to type 15,000 different strings into your progam (much less keep track of them all). Since you are unlikely to have 15,000 different window procedures in your program, you probably don't need 15,000 different classes.

There are some frameworks which generate class names programmatically, and it might be one of those frameworks that is the source of the large amount of class names (and the consequent leaking thereof).

The customer wrote back: "Thanks! That showed us exactly what was leaking: Tons of `HwndWrapper[OurApp;;guid]` classes, with all different GUIDs. It turns out that we were creating a dispatcher object on a background thread. Creating a dispatcher registers the class, which leaks when the background thread terminates."

The customer continued that they were able to fix the leak by using a synchronization context so that the object is created on the UI thread. One of my colleagues provided an alternative solution of calling `Dispatcher.BeginInvokeShutdown` or `Dispatcher.InvokeShutdown` before terminating the background thread. That will shut down the dispatcher cleanly (instead of ripping the thread out from under it), which will destroy the window and unregister the window class.

Raymond Chen

Follow