Why doesn't the window manager unregister window classes when the owning DLL unloads?

devblogs.microsoft.com/oldnewthing/20070212-01

February 12, 2007

Raymond Chen

If you look at the documentation for the `UnregisterClass` function, you'll see that it calls out different behavior depending on whether you're running Windows 95 or Windows NT. Commenter Vipin asked why Windows NT doesn't follow Windows 95's lead. Back in the old days, 16-bit Windows did unregister classes automatically when a DLL unloaded. It had to do this since all of 16-bit Windows ran in a single address space. If a module's classes were not unregistered when it was unloaded, then all of these leaked classes would clog up the system until it was restarted. Since instance handles were just selectors, and by their nature, selectors could be re-used, it meant that a leaked window class could prevent some totally unrelated program from starting. With 32-bit Windows and separate address spaces, a leaked window class affects only the process into which it was leaked. The scope of the damage was contained, and indeed, the scope was limited precisely to the program that did something wrong. If a program leaked a class, it affected only itself. (I suspect there is a contigent of my readership who considers this a good thing. Make programs that screw up pay for their mistake, but don't let them impact other programs.) In addition to the philosophical reason for not unregistering classes, there is also a technological one: Sure, you can unregister the classes, but a DLL that forgets to unregister its classes may very well be so careless as to unload itself while there were still windows left undestroyed! Even if you unregistered the class, those windows are going to crash once the windows receive a message and the window procedure is called. There's also the issue of subclassing. If the module had subclassed a window, unloading the module will leave the subclassed window procedure dangling. The problem isn't solved; it's just papered over. The third reason is architectural. Unregistering a module's classes when it unloads means that there is now an "upward dependency": You've made the kernel call into the window manager. When a module unloads, the kernel needs to call into the window manager to say, "Hey, I just unloaded this module. You might want to clean up stuff." This means that non-GUI programs still have a dependency on the window manager, something you hard-core command line junkies probably would find distasteful. "Why does my non-GUI program have a dependency on the GUI?" There was no such thing as a command line 16-bit Windows program, so this sort of upward dependency wasn't really a problem. But upward dependencies violate modern software design principles. A low-level component should not depend on a higher-level

component. Since Windows NT was a “next generation” operating system, the designers chose to take it as an opportunity to clean up some of the expediencies that had built up in 16-bit Windows.

This is another example of “no matter what you do, somebody will hate it.” The Windows NT folks decide to do some architectural clean-up, the sort of thing one faction of my readership applauds, while another faction argues that, no, it should have made the architecture dirtier in order to solve the problem so applications don’t have to.

Raymond Chen

Follow