What does the CS_CLASSDC class style do?

devblogs.microsoft.com/oldnewthing/20060602-00

June 2, 2006

Raymond Chen

Last time, I talked about the historical background for the `CS_OWNDC` class style and why it starts out sounding like a good idea but when you think about it some more turns out to be an awful idea.

The `CS_CLASSDC` class style is the same thing, but worse, for it takes all the problems of `CS_OWNDC` and magnifies them. Recall that the `CS_OWNDC` class style instructs the window manager to create a DC for the window and use that single DC in response to calls to `BeginPaint` and `GetDC`. The `CS_CLASSDC` takes this one step further and creates a DC for **all the windows of that class**. So that problem I showed last time with a function that thought it had two different DCs for a window can now happen even across windows. You think you have one DC for one window and another DC for another window, but in fact they are the same!

What makes this even worse is that two threads can both be using the same DC at the same time. There is nothing in GDI to forbid it; it's simply a race to see which thread's changes prevail: "Last writer wins." Imagine two threads that happen each to have a `CS_CLASSDC` window from the same window class, and suppose both windows need to be repainted. Each window gets a `WM_PAINT` message, and the two threads both go into their painting code. But what these threads don't know is that they are operating on the **same DC**.

Thread A	Thread B
HDC hdc = BeginPaint(hwnd, &ps);	
	HDC hdc = BeginPaint(hwnd, &ps);
SetTextColor(hdc, red);	
	SetTextColor(hdc, blue);
DrawText(hdc, …);	
	DrawText(hdc, …);

The code running in Thread A fully expected the text to be in red since it set the text color to red and then drew text. How was it to know that just at that moment, Thread B went and changed it to blue?

This is the sort of race condition bug that you'll probably never be able to study under controlled conditions. You'll just get bug reports from customers saying that maybe once a month, an item comes out the wrong color, and maybe you'll see it yourself once in a while, but it will never happen when you have debugger breakpoints set. Even if you add additional diagnostic code, all you'll see is this:

```
...
SetTextColor(hdc, red);
ASSERT(GetTextColor(hdc) == red); // assertion fires!
DrawText(hdc, ...);
```

Great, the assertion fired. The color you just set isn't there. Now what are you going to do? Maybe you'll just say "Stupid buggy Windows" and change your code to

```
// Stupid buggy Windows. For some reason,
// about once a month, the SetTextColor doesn't
// work and we have to call it twice.
do {
 SetTextColor(hdc, red);
} while (GetTextColor(hdc) != red);
DrawText(hdc, ...);
```

And even that doesn't fix the problem, because Thread B might have changed the color to blue after the `GetTextColor` and the call to `DrawText`. Now, it's only once every six months that the item comes out the wrong color.

You swear at Microsoft and vow to develop Mac software from now on.

Okay, so now I hope I've convinced you that `CS_CLASSDC` is a horrifically bad idea. But if it's so fundamentally flawed, why does it exist in the first place?

Because 16-bit Windows is co-operatively multi-tasked. In the 16-bit world, you don't have to worry about another thread sneaking in and messing with your DC because, as I already noted, the fact that you were running meant that nobody else was running. This whole multi-threaded disaster scenario could not occur, so `CS_CLASSDC` is only slightly wackier than `CS_OWNDC`. The introduction of pre-emptive multi-tasking with multiple threads in a single process is what took us into the world of "this has no chance of ever working properly". The class style exists so people who used it in 16-bit code can port to Win32 (as long as they promise to remain a single-threaded application), but no modern software should use it.

Raymond Chen

Follow