Why does setting the horizontal scroll bar range for the first time also set the vertical range, and vice versa?

devblogs.microsoft.com/oldnewthing/20160727-00

July 27, 2016

Raymond Chen

A customer observed that if a window has never set any scroll bar parameters, then it reports its scroll bar range as *nMin* = *nMax* = 0 in both horizonal and vertical directions. But once you set the (say) horizontal scroll bar range to anything (say, *nMin* = 0, *nMax* = 999), then the vertical scroll bar range reports itself as *nMin* = 0, *nMax* = 100. Why does changing one scroll bar affect the other?

For convenience, let's use the notation [*nMin*, *nMax*] to represent a scroll bar range.

```
SCROLLINFO si = { sizeof(si), SIF_RANGE };
GetScrollInfo(hwnd, SB_HORZ, &si); // produces [0, 0]
GetScrollInfo(hwnd, SB_VERT, &si); // produces [0, 0]
SetScrollRange(hwnd, SB_HORZ, 0, 999);
GetScrollInfo(hwnd, SB_HORZ, &si); // produces [0, 999]
GetScrollInfo(hwnd, SB_VERT, &si); // produces [0, 100]
```

This is a case of incomplete virtualization. Every standard scroll bar defaults to a range of [0, 100]. In practice, few windows create scroll bars, so the window manager doesn't allocate scroll bar information until a window activates its scroll bars. But once a window activates any scroll bar, the window manager allocates scroll bar information for both directions.

This behavior came from 16-bit Windows, so let's calculate how much memory 16-bit Windows is saving by using this one weird trick. The scroll bar information for each direction is 8 bytes (four 16-bit values: minimum, maximum, position, and page size), and let's say that heap overhead is two pointers per allocation. Delay-allocating the scroll bar information in one direction on a 16-bit system means that instead of putting 8 bytes of memory in the main heap allocation for a window, you instead put just 2 bytes of memory in the main heap allocation, but an additional cost of 8 + 2 × 2 = 12 bytes of heap memory if the window actually uses the scroll bar.

Let's say that ten percent of the windows in the system use scroll bars, a rather high estimate, I think, especially when you consider dialog boxes which have tons of windows without any scroll bars.[1] With that assumption, the average cost per window drops from 8 bytes to 2 +

10% × 12 = 3.2 bytes per window for a single scroll bar direction, or from 16 bytes to 6.4 bytes for the pair.

We can save even more memory by putting the horizontal and vertical scroll bar info together in the same allocation, since that reduces the heap overhead, and it means that you need to leave only one forwarding pointer behind to cover two blocks of data. With this additional assumption, the average cost per window drops from 16 bytes to ~~2 + 10% × 16 = 3.6 bytes~~ 2 + 10% × 20 = 4 bytes.

This savings by using combined storage for both directions does mean that the cost for a window that uses only one of the two directions is ~~2 + 16 = 18 bytes~~ 2 + 20 = 22 bytes, when it would have been 2 + (2 + 12) = 16 bytes if the two allocations had been separate. Most of the time, a window that has a horizontal scroll bar will also have a vertical scroll bar. (Edit boxes are a notable exception.) If we say that half of the time will a window have only one of the scroll bars, then the tradeoff is a 50% chance of 16 bytes against an 50% chance of 4 + 24 = 28 bytes, for an average cost of 22 bytes per window, which is no better than the 22 bytes per window from combining the two allocations.

Given that 16-bit Windows had only 64KB of memory for all window-related objects, reducing the base memory cost of a window from 102 bytes to 88 bytes is a huge savings.

Okay, let's return to the present. If standard scroll bars default to [0, 100], why does reading the scroll bar range of an uninitialized scroll bar return [0, 0] instead of [0, 100]?

Actually, reading the scroll bar range of an uninitialized scroll bar doesn't return a range of [0, 0]. What's actually happening is that the call to `GetScrollInfo` is failing with the error code `ERROR_NO_SCROLLBARS`, and you are reading back the zero values that were already in the `SCROLLINFO` structure that you passed in.

When you call `SetScrollRange` (or `SetScrollInfo` or `SetScrollPos`) the window manager initializes the scroll bar information on demand, and that's where the default values of [0, 100] are established. Those values then get read out by the subsequent calls to `GetScrollInfo`.

Basically, Windows pretends that all windows have a scroll bar with a range of [0, 100], but it doesn't allocate any memory to record that information until you use it.

Exercise: Theoretically, the window manager could also have avoided allocating the memory if you set the range to [0, 100], since that's the default range. Why doesn't it bother with this optimization?

What you're seeing is that the virtualization is incomplete. When you try to read the scroll range from an uninitialized scroll bar, the `GetScrollInfo` function could have reported a range of [0, 100] instead of simply failing the call. Or possibly report a range of [0, 100] *and*

fail the call, reporting the default range to cover for the programs (like the one above) that ignore the return value.

My guess is that the original designers of the window manager chose to expose this "uninitialized" state explicitly on the off chance that some program might[2] want to check whether scroll bars are initialized so that they can perform some super-precise optimization.

Though in practice I bet nobody does.

[1] I just ran a quick test on my system. My guesses were waaaaaaay too generous.

Window type	With scroll bars	Total	Percent
Top-level	1	465	0.21%
Child	7	1034	0.68%
Overall	8	1499	0.53%

[2] This design principle dates back to the days when Windows assumed that programmers were super-experts who wanted fine control of everything by default. "Here's your fine control. Good luck."

Raymond Chen

Follow