Using modular arithmetic to avoid timing overflow problems

devblogs.microsoft.com/oldnewthing/20050531-22

May 31, 2005

Raymond Chen

In an earlier article, I presented a simple way of avoiding timing overflows which seemed to create a bit of confusion. The short version: Given a starting time **start**, an ending time **end** and an interval **interval**, the way to check whether the interval has elapsed is to use the expression `end - start >= interval`. The naive expression `end >= start + interval` suffers from integer overflow problems. To simplify the discussion, let's operate in base-100 instead of base-2^{32}. The same logic works, but I think operating in base-100 will be easier to follow. Base-100 means that we remember only the last two digits of any number. Consider a starting time of `start = 90` and an interval of `interval = 10`. Using the wrong expression yields `end >= start + interval = 90+10 = 100 = 0`. In other words, `end >= 0` which is always true since `end` has the range `0...99`. As a result, the wrong expression will think that the interval has expired prematurely. Using the correct expression, we have `end - 90 >= 10`. Of the numbers `0..99`, the ones that give a difference less than 10 are `90` through `99`. Once `end = 0`, the result is `0 - 90 = 10`, which correctly indicates that 10 ticks have elapsed since 90 once the timer reaches 0. You can work through a similar mistake using `start = 89` instead of `start = 90`; in this case, the wrong expression becomes `end >= start + interval = 89 + 10 = 99`, or in other words, `end >= 99`. This has the opposite problem from the previous case, namely that the expression will fail to detect that the interval has expired once the timer rolls over. But why does the `end - start` expression work? It's very simple: You just have to remember your rules of arithmetic from elementary school. `(x - c) - (y - c) = x - c - y + c = x - y` In other words, subtracting the same value from both terms of a difference does not affect the final value. This rule applies even to modular arithmetic (because, as the mathematicians like to say, the set of integers modulo *n* form an additive group). This rule is useful because it lets you delay the overflow as long as possible by subtracting the starting point from all your time markers; it has no effect upon time intervals. Wouldn't it be great if `start = 0`? Then the overflow won't happen for 100 ticks. Well, you can act "as if" the starting point were `start = 0` by simply subtracting `start` from all your time markers. Those who prefer a graphical view can think of time passing as the hands around a clock (which wraps around at 60 minutes, say). When you decide to record your start point, rotate the clock so that the "12" precisely lines up with wherever the hand happens to be. You can now read off the elapsed

time directly from your rotated clock. Rotating your clock is the same as subtracting (or adding) a constant to all time markers. Of course, this trick falls apart once you have to measure time intervals that come close to the wraparound time of your timer. In our 100-tick timer, for example, trying to measure the passage of 90 ticks is very difficult because there is only a 10-tick window where the inequality is satisfied. If we fail to catch the timer during that window, we miss it and have to wait another 90 ticks.

So don't do that. In practical terms, this means that you shouldn't use `GetTickCount` to measure time intervals longer than 15 days.

Raymond Chen

Follow