But then we ran into problems when we started posting 10,000 messages per second

devblogs.microsoft.com/oldnewthing/20090126-00

January 26, 2009

Raymond Chen

Once upon a time, a long, long time ago, there was a research team inside Microsoft who was working on alternate models for handling input. I don't know what eventually came of that project, and I don't even remember the details of the meeting, but I do remember the punch line, so I'm just going to make up the rest.

The research project broke up the duties of their system into a few components. The two that are important to the story are a driver component which received information from various hardware devices and transmitted that information via the `PostMessage` function to another component whose job it was to study those input messages and route them to the appropriate application. (In 16-bit Windows, the `PostMessage` function was specifically written so it could be called from device drivers during hardware interrupts.) Each time the driver received information from a hardware device, it posted a message to its helper program.

Everything seemed to go reasonably smoothly. The device driver received a hardware event, it posted a message to the helper program, and the helper program retrieved the message and processed it. But once they cranked up the hardware devices to produce information at a higher rate (and therefore produced input with much finer resolution), the events started coming in faster and faster, and their design started to collapse under the pressure.

The research team asked to meet with the user interface team to help work out their problems under load. They outlined their design and explained that it worked well at low data rates, "but then we ran onto problems when we started posting 10,000 messages per second."

At that point, the heads of all the user interface people just sat there and boggled for a few seconds.

"That's like saying your Toyota Camry has stability problems once you get over 500 miles per hour."

If you're going to be pumping huge quantities of data through the message queue, creating a separate message for each one is crazy. Think about it: Suppose you're posting 10,000 messages per second. The thread whose job it is to process the messages gets pre-empted and doesn't run for 50 milliseconds. That's 500 messages behind schedule already. Now suppose it takes a dozen page faults (which take 8ms each, say). Now you're another 1000 messages behind. Windows NT sets an arbitrary limit of 10,000 unprocessed messages in a message queue so that a runaway program won't drain the desktop heap and roach everything. A few hiccups in your process will quickly send you over that limit.

For this usage pattern, you want to switch from *one event per message* to a *signal on the transition* (or *edge triggering*).

When the first event occurs, post a single message to the helper program saying *there is work to do* and set a flag saying *helper window has been notified*. Meanwhile, stash the information you would have included in the message into a privately-managed queue. If an event arrives when the *helper window has been notified* flag is set, then don't post a message; just append the work item to the queue. When the helper window receives the *there is work to do* message, it calls back into the driver to say *Okay, give me some work to do*. After it does the work, it calls into the driver to say *Okay, what else do you want me to do?* (Alternatively, you can have the helper window grab the entire work list at once.) When the helper window asks for work to do and comes back empty-handed, then clear the *helper window has been notified* flag so the next time an event occurs, a new message will be posted to kick-start the helper window.

Commenter Hayden proposed a number of other mechanisms. The *send a list of work items rather than just one* technique works well if you know when the list of work items is complete and therefore is ready to send. The second technique is the one I described here; it works well if the producer doesn't really know when each chunk of incoming work is finished, or if the work that comes in is continuous. The third mechanism merely avoids the message queue altogether and uses a semaphore instead.

The point is not to try to drive your Camry at 500 miles per hour. Find a way to get your work done while keeping the Camry well within its design parameters, or look for a different vehicle.

Raymond Chen

Follow