Asynchronous input vs synchronous input, a quick introduction

devblogs.microsoft.com/oldnewthing/20130604-00

June 4, 2013

Raymond Chen

One of the topics I covered at my PDC talk was the asynchronous input model. I don't think I ever discussed it on this Web site, so I guess I'll do it now, so that I can point people at it in the future.

In the old days of 16-bit Windows, input was synchronous. All input went into a system-wide input queue, and the intuitive rule for input processing is that input messages are dispatched in chronological order. If the user clicked on one window, and then clicked on some other window, the first window must receive and process its click message before the second window will receive its own click message.

This model ensures that the user sees input being processed in the order it was generated, which is good. It also means that if an application stops processing input, it gums up the entire input system, which is bad. But since 16-bit Windows was coöperatively multi-tasked anyway, getting upset that a bad application could clog up the whole system by refusing to process input is missing the big picture: A bad application could clog up the whole system by simply refusing to release control of the CPU back to the operating system.

It's like getting upset that somebody with the keys to your house could mess up the order of books on your shelf. I mean, they have *the keys to your house*. If they wanted to make your life miserable, they would do much more than just screw with your book collection.

In Win32, the input queues were made asynchronous. As well as a system input queue, there is also an input queue for each thread. Hardware devices post events into the system input queue, but the messages doesn't stay in the system input queue for very long: A dedicated thread known as the *raw input thread* takes the events out of the system input queue and distributes them to the appropriate application input queues. (This distribution can't be done at the time the hardware devices post the event, because that happens asynchronously. In unix speak, the hardware devices post the events into the system input queue from the *bottom half*, and the raw input thread processes them from the *top half*.)

The advantage of this model is that one thread that stops processing input does not prevent other threads from processing input, because the input for the two threads has been separated and are not dependent on each other. (This property of asynchronous input has been called the Holy Grail.)

More generally, input-related activities that were global in 16-bit Windows were made local in Win32. Each input queue has its own focus window, its own active window, its own caret, its own mouse cursor, and so on.

Things get weird when you have cross-thread input attachment, which can happen either when a parent/child or owner/owned relationship crosses a thread boundary, or by an explicit attachment created by the `AttachThreadInput` function. If cross-thread attachment is active, then all the threads which are attached to each other (which I call a *thread group*) share an input queue, and then the coöperative rules for input apply to that group of threads. It's like putting on your skinny tie and shoulder-padded jacket and inviting all your friends over to pretend that it's the 1980's all over again.

Anyway, that's just an introduction. We'll dig in deeper as the week progresses.

Raymond Chen

Follow