When you share an input queue, you have to wait your turn

devblogs.microsoft.com/oldnewthing/20130605-00

June 5, 2013

Raymond Chen

Now that we've had a quick introduction to asynchronous input, let's look at some of the details. Remember, **this is a peek under the hood at how the sausage is made**. The algorithm described here is not part of the API contract and it can change at any time, as long as it services the overall goal of serializing input.

Let's start by looking at how things worked in the 16-bit world. Even though 16-bit Windows didn't use the term *thread* (since each application was single-threaded), I will still use the term since that makes the transition to the 32-bit world more straightforward.

As a point of terminology, I say that a message *belongs to* a thread if the message targets a window which belongs to that thread. Equivalently, the thread *owns* the message.

Now, the goal is to dispatch input messages in chronological order: Only the thread which owns the next input message can retrieve it. All other input must wait their turn to come to the front of the input queue.

In 16-bit Windows, all input gets added to a system-wide input queue, and the basic algorithm used by `PeekMessage` and `GetMessage` for retrieving messages from the input queue goes like this.

- Look at the first message in the input queue:
  - If the message belongs to some other thread, then stop. Return no message to the caller.
  - Otherwise, return the message we found.
- If there are no messages in the input queue, then there is no input. Return no message.

All the rest is tweaking the boundary cases.

For example, suppose there are two input messages in the input queue, message 1 for thread A, and message 2 for thread B. Thread A calls `GetMessage`, and the above algorithm returns message 1 to thread A, at which point the new "first message" is message 2, and if thread B calls `GetMessage`, it will get message 2.

The catch is that according to the above algorithm, thread B can be told about message 2 *before thread A has finished processing message 1*. You've introduced a race condition that breaks the rule that input is processed sequentially: Thread B can race ahead of thread A and start processing message 2 before thread A can even get started processing message 1.

To fix this, we add a new state to the input queue that says "Yea, I just gave somebody an input message, and I'm waiting for that thread to finish processing it before I will hand out another input message."

- (New!) If the input queue is waiting for another thread to finish processing an input message, then stop and return no message.
- (New!) If the input queue is waiting for the current thread to finish processing an input message, then mark the input queue as no longer waiting. (We finished processing it and have come back for more!)
- Look at the first message in the input queue:
    - If the message belongs to some other thread, then stop. Return no message to the caller.
    - Otherwise, (New!) mark the input queue as waiting for the current thread to finish processing an input message, and return the message we found.
- If there are no messages in the input queue, then there is no input. Return no message.

Okay, we fixed a race condition. But now there's a new problem: Suppose thread A retrieves an input message (and therefore puts the input queue into the "waiting for thread A" state), and then thread A sends a message to thread B, and thread B wants to display a dialog box or a menu. According to the rules as we have them so far, we would have a deadlock: That `SendMessage` call will not return to the first thread until the modal UI is complete, but the modal UI cannot complete because the input queue is waiting for thread A to finish processing the input message.

The fix for this special case is that if a thread asks for an input message, and it is handling an inbound `SendMessage`, then the input queue declares that any in-progress input message has finished processing. One way of interpreting this rule is to say, "If a thread sends a message to another thread, then that implicitly completes input processing."

- (New!) If the input queue is waiting for another thread to finish processing an input message, and the current thread is processing an inbound sent message, then mark the input queue as no longer waiting.
- If the input queue is waiting for another thread to finish processing an input message, then stop and return no message.
- If the input queue is waiting for the current thread to finish processing an input message, then mark the input queue as no longer waiting.

- Look at the first message in the input queue:
    - If the message belongs to some other thread, then stop. Return no message to the caller.
    - Otherwise, mark the input queue as waiting for the current thread to finish processing an input message, and return the message we found.
- If there are no messages in the input queue, then there is no input. Return no message.

Recall that you are allowed to pass a message range filter and a window handle filter to the `PeekMessage` and `GetMessage` functions. The above algorithm was developed on the assumption that there was no message retrieval filter. First, let's add the message range filter to the algorithm:

- If the input queue is waiting for another thread to finish processing an input message, and the current thread is processing an inbound sent message, then mark the input queue as no longer waiting.
- If the input queue is waiting for another thread to finish processing an input message, then stop and return no message.
- If the input queue is waiting for the current thread to finish processing an input message, then mark the input queue as no longer waiting.
- Look at the first message in the input queue which satisfies the message range filter (New!):
    - If the message belongs to some other thread, then stop. Return no message to the caller.
    - Otherwise, mark the input queue as waiting for the current thread to finish processing an input message, and return the message we found.
- If there are no messages in the input queue which satisfy the message range filter (New!), then there is no input. Return no message.

That wasn't so hard. If you pass a message range filter, then we care only about messages that pass the filter in determining which one is "at the head of the input queue". Without this additional rule, you wouldn't be able to "peek into the future" to see if, for example, there is a mouse message in the input queue sitting behind the keyboard message that is at the front of the input queue.

Adding the window handle filter is a little trickier, because we still want to let input be processed in order (among messages which satisfy the message range filter).

- If the input queue is waiting for another thread to finish processing an input message, and the current thread is processing an inbound sent message, then mark the input queue as no longer waiting.
- If the input queue is waiting for another thread to finish processing an input message, then stop and return no message.

- If the input queue is waiting for the current thread to finish processing an input message, then mark the input queue as no longer waiting.
- Look at the first message in the input queue which satisfies the message range filter and (New!) either belongs to some other thread or belongs to the current thread and matches the window handle filter.
    - If the message belongs to some other thread, then stop. Return no message to the caller.
    - Otherwise, mark the input queue as waiting for the current thread to finish processing an input message, and return the message we found.
- If no such message exists, then there is no input. Return no message.

In other words, the window handle is used to control which message is ultimately retrieved, but it does not let you deny another thread access to input which matches the message range filter.

Whew, that's how 16-bit Windows dispatched input.

How do we port this to 32-bit Windows and asynchronous input?

First, we give each thread group its own input queue, rather than having a single system-wide input queue.

Second, whenever the above algorithm says *the input queue*, change it to say *the calling thread's input queue*.

And that's it!

In the absence of thread attachment, each thread has its own input queue, so most of the above rules have no effect. For example, You will never see a message that belongs to another thread, because messages that belong to other threads go into those other threads' input queues, not yours. You will never find that your input queue is waiting for another thread, because no other threads have access to your input queue. In the case where there is only one thread associated with an input queue, the algorithm simplifies to

> Return the first message in the input queue that satisfies both the message range filter and the window handle filter, if any.

It's only if you start attaching threads to each other that you have multiple threads associated with a single input queue, and then all these extra rules start to have an effect.

Next time, we'll explore some of the consequences of synchronous input by writing some poorly-behaving code and observing how the system responds. Along the way, we'll discover an interesting paradox introduced by the above algorithm.

Exercise: The alternative interpretation I gave above does not match the description of the rule, because the rule allows *any* thread processing an inbound `SendMessage` to clear the input queue's wait state. Why does the actual rule permit any thread clear the wait state, instead of first checking that the inbound `SendMessage` is coming from the thread that the input queue is waiting for?

Raymond Chen

Follow