WaitForInputIdle waits for any thread, which might not be the thread you care about

devblogs.microsoft.com/oldnewthing/20100326-00

March 26, 2010

Raymond Chen

We saw last time that the `WaitForInputIdle` function waits only once for a process to go input idle. Even if the process later stops processing messages, `WaitForInputIdle` will return immediately and say, "Yeah, he's idle." The way a process is determined to be input idle is that it is waiting for user input when there is none. This translates into the process sitting in a function like `GetMessage` or `WaitMessage` when there are no messages. But what if a process has more than one thread? And what if one of the threads is waiting for input, while the other is busy and unresponsive? The `WaitForInputIdle` function will treat the process as having gone input idle, even if the ready thread is just displaying your splash screen and the busy thread is the one preparing the main window. The `WaitForInputIdle` function doesn't know that the main window is more important than the splash screen; as far as the window manager is concerned, you've got two threads, each with a window. What this means for your application is that you need to know that, if you create multiple threads, then the moment any of them goes input idle, the entire process is treated as input idle, and you need to be ready for people who were waiting for you via `WaitForInputIdle` to start trying to talk to your application. As I noted last time, the `WaitForInputIdle` function is really just a stopgap to help with the transition from 16-bit Windows to 32-bit Windows. Whereas 16-bit programs could just charge ahead knowing that the program it launched is ready (because if it weren't ready, then it would still have control of the CPU), 32-bit programs need to wait for this faked-up version of `Yield` . And since it was created merely to aid in porting 16-bit programs, the `WaitForInputIdle` function didn't really worry about multiple threads. After all, 16-bit Windows didn't support multiple threads per process, so all 16-bit programs were necessarily single-threaded. If you're porting one of these programs, your initial 32-bit version is also going to be single-threaded. (At least I hope you're not going to try to add multiple threads right off the bat. The first step in porting is just to get the program to run without adding any new features!) In fact, back in the old days, the `WaitForInputIdle` function tried a bit too hard to emulate the `Yield` behavior from 16-bit Windows. When the target application received a message, it was taken *out of* the input idle state, and went back into the state when it once again reached a state where it was waiting for input. In other words, the one-line summary of the `WaitForInputIdle` function was actually correct at the time it was written. The old mechanism for `WaitForInputIdle` ,

taking the process in and out of the input idle state, mirrored the 16-bit behavior of Windows, but *only if you had a single-threaded application*. If you had multiple threads, then the input-idle state starts getting all wonky as each thread updates the global idle state:

	Thread 1	Thread 2
1	Busy	Busy
2	Idle	
	(application marked as idle)	
3		Idle
	(application marked as idle)	
4		Busy
	(application marked as busy)	
5	Busy	
	(application marked as busy)	

There is only one state that gets updated by each thread as they go idle or busy. All you really know is that if all threads are busy, then the input idle state will be *busy* and that if all threads are idle, then the input idle state will be *idle*. But if one thread is idle and the other is busy, then the process idle state is unpredictable; it depends on what the most recent transition was. For example, notice that at step 2, we have one idle thread (thread 1) and one busy thread (thread 2), and the process is marked *idle*. On the other hand, at step 4, we also have one idle thread (thread 2) and one busy thread (thread 1), but this time, the process is marked *busy*. Of course, since this behavior was intended to mimic the 16-bit programming model, the fact that it got all confused when applications created multiple threads was "out of scope". Creating multiple threads meant that you have gone past the simple "Just trying to get it to work" stage and have moved on to adding Win32-specific enhancements. You were creating a situation that the `WaitForInputIdle` function was not designed to handle. My guess is that the unpredictable nature of the original design led the window manager folks to take a deep breath and go back to the spirit of the `WaitForInputIdle` function: To tell when a program has finished its initialization.

So now it decides that when you've finished initialization, you've finished initialization. It's a one-way door.

Raymond Chen

Follow