The performance cost of reading a registry key

devblogs.microsoft.com/oldnewthing/20060222-11

February 22, 2006

Raymond Chen

The registry is a convenient place to record persistent cross-process data in a uniform and multi-thread-safe manner. It roams with the user if you store it in `HKEY_CURRENT_USER`, and individual keys can be secured (even on systems that use FAT, which doesn't otherwise support security). But that doesn't mean that it's free. The cost of opening a key, reading a value, and closing it is around 60,000 to 100,000 cycles (I'm told). And that's assuming the key you're looking for is in the cache. If you open the key and hold it open, then the act of reading a value costs around 15,000 to 20,000 cycles. (These numbers are estimates for Windows XP; actual mileage may vary.) Consequently, you shouldn't be reading a registry key in your inner loop. Not only does it cost you CPU time at query time, but the constant hammering of the registry means that the data structures used by the registry to locate and store your key (including the entry in the registry cache) are kept in the system working set. Don't read a registry key on every mouse move; read the value once and cache the result. If you need to worry about somebody changing the value while your program is running, you can establish a protocol for people to follow when they want to change a setting. Windows, for example, uses functions such as `SystemParametersInfo` to manipulate settings that are normally cached rather than read from the registry each time they are needed. Calling the update function both updates the registry and the in-memory cache. If you can't establish a mechanism for coordinating changes to the setting, you can set a change notification via the `RegNotifyChangeKeyValue` function so that you are notified when the value changes. Whenever possible, optimize for the common case, not the rare case. The common case is that the registry value hasn't changed. By using a notification mechanism, you move the cost of "But what if the value changed?" out of your inner loop and into code that doesn't execute most of the time. (Remember, the fastest code is code that never runs.) Of course, you don't want to burn a thread waiting on the notification event. I use the thread pool. The `RegisterWaitForSingleObject` function lets you tell the thread pool, "Hey, could you call me when this object is signalled? Thanks." The thread pool then does the work of combining this with all the other handles it has been asked to wait for into a giant `WaitForMultipleObjects` call. That way, one thread can handle multiple waits.

One caveat to heed with the `RegNotifyChangeKeyValue` function is that the notification has thread affinity! If the thread that calls the `RegNotifyChangeKeyValue` function exits, the notification is raised. This means that you shouldn't call the function from a thread pool thread, since the system will destroy threads in the thread pool when the work list goes idle and their presence is no longer needed. If you mess up and call it from a thread pool thread, you'll find that the event keeps firing spuriously as the thread pool cleanup code runs, making the cure as bad as the disease! Instead, you should create the wait from a persistent thread (say, the thread that actually cares about the value!) and register the wait there. When the event fires on the thread pool, handle the change, then ask your persistent thread to start a new cycle of `RegNotifyChangeKeyValue`. That way, the event is always associated with your persistent thread instead of with a transient thread pool thread.

Raymond Chen

Follow