The history of Win32 critical sections so far

devblogs.microsoft.com/oldnewthing/20140911-00

September 11, 2014

Raymond Chen

The `CRITICAL_SECTION` structure has gone through a lot of changes since its introduction back oh so many decades ago. The amazing thing is that as long as you stick to the documented API, your code is completely unaffected.

Initially, the critical section object had an owner field to keep track of which thread entered the critical section, if any. It also had a lock count to keep track of how many times the owner thread entered the critical section, so that the critical section would be released when the matching number of `LeaveCriticalSection` calls was made. And there was an auto-reset event used to manage contention. We'll look more at that event later. (It's actually more complicated than this, but the details aren't important.)

If you've ever looked at the innards of a critical section (for entertainment purposes only), you may have noticed that the lock count was off by one: The lock count was the number of active calls to `EnterCriticalSection` *minus one*. The bias was needed because the original version of the interlocked increment and decrement operations returned only the sign of the result, not the revised value. Biasing the result by 1 means that all three states could be detected: Unlocked (negative), locked exactly once (zero), reentrant lock (positive). (It's actually more complicated than this, but the details aren't important.)

If a thread tries to enter a critical section but can't because the critical section is owned by another thread, then it sits and waits on the contention event. When the owning thread releases all its claims on the critical section, it signals the event to say, "Okay, the door is unlocked. The next guy can come in."

The contention event is allocated only when contention occurs. (This is what older versions of MSDN meant when they said that the event is "allocated on demand.") Which leads to a nasty problem: What if contention occurs, but the attempt to create the contention event fails? Originally, the answer was "The kernel raises an out-of-memory exception."

Now you'd think that a clever program could catch this exception and try to recover from it, say, by unwinding everything that led up to the exception. Unfortunately, the weakest link in the chain is the critical section object itself: In the original version of the code, the out-of-

memory exception was raised while the critical section was in an unstable state. Even if you managed to catch the exception and unwind everything you could, the critical section was itself irretrievably corrupted.

Another problem with the original design became apparent on multiprocessor systems: If a critical section was typically held for a very brief time, then by the time you called into kernel to wait on the contention event, the critical section was already freed!

```
void SetGuid(REFGUID guid)
{
 EnterCriticalSection(&g_csGuidUpdate);
 g_theGuid = guid;
 LeaveCriticalSection(&g_csGuidUpdate);
}
void GetGuid(GUID *pguid)
{
 EnterCriticalSection(&g_csGuidUpdate);
 *pguid = g_theGuid;
 LeaveCriticalSection(&g_csGuidUpdate);
}
```

This imaginary code uses a critical section to protect accesses to a GUID. The actual protected region is just nine instructions long. Setting up to wait on a kernel object is way, way more than nine instructions. If the second thread immediately waited on the critical section contention event, it would find that by the time the kernel got around to entering the wait state, the event would say, "Dude, what took you so long? I was signaleded, like, a bazillion cycles ago!"

Windows 2000 added the `InitializeCriticalSectionAndSpinCount` function, which lets you avoid the problem where waiting for a critical section costs more than the code the critical section was protecting. If you initialize with a spin count, then when a thread tries to enter the critical section and can't, it goes into a loop trying to enter it over and over again, in the hopes that it will be released.

"Are we there yet? How about now? How about now? How about now? How about now? How about now? How about now? How about now? How about now? How about now? How about now?"

If the critical section is not released after the requested number of iterations, then the old slow wait code is executed.

Note that spinning on a critical section is helpful only on multiprocessor systems, and only in the case where you know that all the protected code segments are very short in duration. If the critical section is held for a long time, then spinning is wasteful since the odds that the critical section will become free during the spin cycle are very low, and you wasted a bunch of CPU.

Another feature added in Windows 2000 is the ability to preallocate the contention event. This avoids the dreaded "out of memory" exception in `EnterCriticalSection`, but at a cost of a kernel event for every critical section, whether actual contention occurs or not.

Windows XP solved the problem of the dreaded "out of memory" exception by using a fallback algorithm that could be used if the contention event could not be allocated. The fallback algorithm is not as efficient, but at least it avoids the "out of memory" situation. (Which is a good thing, because nobody really expects `EnterCriticalSection` to fail.) This also means that requests for the contention event to be preallocated are now ignored, since the reason for preallocating (avoiding the "out of memory" exception) no longer exists.

(And while they were there, the kernel folks also fixed `InitializeCriticalSection` so that a failed initialization left the critical section object in a stable state so you could safely try again later.)

In Windows Vista, the internals of the critical section object were rejiggered once again, this time to add convoy resistance. The internal bookkeeping completely changed; the lock count is no longer a 1-biased count of the number of `EnterCriticalSection` calls which are pending. As a special concession to backward compatibility with people who violated the API contract and looked directly at the internal data structures, the new algorithm goes to some extra effort to ensure that if a program breaks the rules and looks at a specific offset inside the critical section object, the value stored there is –1 if and only if the critical section is unlocked.

Often, people will remark that "your compatibility problems would go away if you just open-sourced the operating system." I think there is some confusion over what "go away" means. If you release the source code to the operating system, it makes it even *easier* for people to take undocumented dependencies on it, because they no longer have the barrier of "Well, I can't find any documentation, so maybe it's not documented." They can just read the source code and say, "Oh, I see that if the critical section is unlocked, the `LockCount` variable has the value –1." Boom, instant undocumented dependency. Compatibility is screwed. (Unless what people are saying "your compatibility problems would go away if you just open-sourced *all applications*, so that these problems can be identified and fixed as soon as they are discovered.")

Exercise: Why isn't it important that the fallback algorithm be highly efficient?

Raymond Chen

Follow