Jason Youngers wrote:
I just came across this interesting article on why locks were made unfair in Windows Vista, when they had previously been fair. Might be a worthwhile topic to bring up in the OS class:

from www.bluebytesoftware.com/blog/PermaLink,
guid,e40c2675-43a3-410f-8f85-616ef7b031aa.aspx

Raymond just posted a brief entry about lock convoys. While he links to a few other relevant posts, none of them mention the new anti-convoy features that all Windows locks now use. So I thought that I would take the chance to do just that.

Many people claim that fair locks lead to convoys. In my experience, however, few people really know the reason why. Before Windows Vista (client OS) and Windows Server SP1 (server OS), mutexes, critical sections, and internal locks like kernel pushlocks used a lock handoff mechanism to guarantee true fairness. In other words, the lock would not actually become "available" when released so long as there were threads waiting to acquire it. Instead, the thread releasing the lock would modify the lock's state so that it appeared as if the next thread in the wait queue already owned it. The new owner thread would then simply wake up, typically via the releaser signaling an event, and the owner would find that it already owned the lock, proceeding happily. No other thread can "sneak in" between the time the new owner is woken and the time that it is actually scheduled for execution with this design.

While this sounds nice and, well, fair, it exacerbates convoys. Why? Because it effectively extends the lock hold time by the communication and context switch latency required to wake and reschedule the new owner. Context switches on Windows are anything but cheap, and tend to cost anywhere from 4,000 to 10,000+ cycles. Assume C represents the cost of a context switch. Then with a truly fair lock, the lock will be in an intermediary handed off state, with no thread actually running code under its protection, for about C cycles. It's actually worse than this. Assuming the system is busy, a thread that is woken just goes to the end of the OS's thread scheduler queue, and is required to wait until it gets allocated a timeslice in which to execute. This can make C much larger in practice. And of course on highly loaded systems the condition worsens, which can add insult to injury (as we see momentarily).

To cope with the possibility of a scheduling delay, Windows uses something called a priority boost for any thread waiting on an auto-reset event (or that owns a window); the boost temporarily increases the target thread's priority which subsequently decays after it gets scheduled. Assuming no other high priority threads are runnable, this ensures the latency is very close to C. But C's still pretty darned big...

To illustrate the problem with the fair handoff scheme, imagine we have a lock L, for which a new thread arrives every 2,000 cycles. Each such thread runs for 1,000 cycles under the protection of the lock. No problem, right? On average, the lock will be held for 1,000 cycles, unheld for 1,000 cycles, and so on. Assuming the arrival rate is
somewhat random, but statistically averages out to the values mentioned, then occasionally we might get some contention, requiring waiting. But for every contentious acquire, there should also be a big gap in time where there are no owners (or where wait queues can be drained). A system with these characteristics should balance out well. It should survive until threads begin arriving at a frequency of more than 1,000 cycles, give or take some epsilon, which is actually a doubling in throughput. It’s not even close to capacity. (Real systems depend on many more factors than this simplistic view, but you get the point.)

As soon as the lock is fair, however, this scheme quickly becomes untenable and will come to a grinding halt. Imagine that thread T0 acquires L at cycle 0; if it just so happens that T1 tries to acquire it at cycle 500, then T1 will have to wait. Remember, on average, the arrival rate is 2,000 cycles, but that’s just an average. We expect the occasional wait to occur. This wait, unfortunately, causes a domino effect from which the system will never recover. T0 then releases L at cycle 1,000, as expected, handing off ownership to T1; sadly, T1 doesn’t actually start running inside the lock until 5,000 (assuming 4,000 for C, and assuming no scheduling delay); in the 4,000 cycles it took for T1 to wake back up and start running, we expect 2 new threads will have arrived on average; these threads would see L as owned by T1 and respond by waiting. By the time those threads execute, another 10,000 cycles ($2^8(C+1,000)$) will have passed, and another 4 threads will have begun waiting. And so on. This process repeats indefinitely, the requests pile up (hopefully with a bound), and disaster strikes. The system simply won’t scale this way.

If you remove the strict fairness policy, however, the system scales. And that, my friends, is why all of the locks in Windows are now unfair.

Of course, Windows locks are still a teensy bit fair. The wait lists for mutually exclusive locks are kept in FIFO order, and the OS always wakes the thread at the front of such wait queues. (APCs regularly disturb this ordering -- a topic for another day -- which actually calls into question the merit of the original design goal of attaining fairness in the first place.) Now when a lock becomes unowned, a FIFO waking algorithm is still used, but the lock is immediately marked as unavailable. Another thread can sneak in and take the lock before the woken thread is even scheduled (although priority boosting is still, somewhat questionably, in the system). If another thread steals the lock, the woken thread may subsequently have to rewait, meaning it must go to the back of the queue, again disturbing the nice FIFO ordering.

The change to unfair locks clearly has the risk of leading to starvation. But, statistically speaking, timing in concurrent systems tends to be so volatile that each thread will eventually get its turn to run, probabilistically speaking. Many more programs would suffer from the convoy problems resulting from fair locks than would notice starvation happening in production systems as a result of unfair locks.