On priority inversion in the use of a spinlock to ensure atomic access to a shared_ptr

devblogs.microsoft.com/oldnewthing/20250407-00/?p=111054

April 7, 2025

In my discussion of the internal implementation of `std::atomic<std::shared_ptr<T>>`, I noted that the use of a spinlock without a blocking fallback could result in a deadlock due to priority inversion. Commenter Anton Siluanov noted, "While priority inversion is a thing, from atomic I'd expect as quick as possible operation. And mutex impl can be done manually."

It's true that the spinlock is not held for long, but even the tiniest window will get hit, and probably sooner than you would like. My colleague Larry Osterman phrases this as "One in a million is next Tuesday." James Hamilton (formerly of Microsoft, now at Amazon) described it more mundanely as "At scale, rare event's aren't rare."

In this case, the race condition occurs if a higher priority thread tries to enter the spinlock while a lower priority thread holds it. And even though it's a small race window by instruction count, it can actually be quite a long time if there is a poorly-timed context switch, and an even longer time if the control block has been paged out.

Thread 1 (low priority)	Thread 2 (high priority)
set lock bit	
increment refcount in control block	danger zone
clear the lock bit	
set lock bit	

If the high priority thread runs during the danger zone, and the process has no other idle processors (say, because it's a uniprocessor system, or the other processors are busy running medium-priority threads), then you have a priority inversion deadlock, because the high priority thread is consuming the resource that the low priority thread needs in order to release the lock.

Yes, it is a narrow race window, but narrow windows will get hit, and if they hit in just the wrong way, it will ruin somebody’s day.