When is pthread_spin_lock the right thing to use (over e.g. a pthread mutex)?

Given that pthread_spin_lock is available, when would I use it, and when should one not use them ?

i.e. how would I decide to protect some shared data structure with either a pthread mutex or a pthread spinlock ?

c pthreads

asked Jul 6 '11 at 21:43
Lyke
1,464 3 11 21

4 Answers

The short answer is that a spinlock can be better when you plan to hold the lock for an extremely short interval (for example to do nothing but increment a counter), and contention is expected to be rare, but the operation is occurring often enough to be a potential performance bottleneck. The advantages of a spinlock over a mutex are:

1. On unlock, there is no need to check if other threads may be waiting for the lock and waking them up. Unlocking is simply a single atomic write instruction.
2. Failure to immediately obtain the lock does not put your thread to sleep, so it may be able to obtain the lock with much lower latency as soon a it does become available.
3. There is no risk of cache pollution from entering kernelspace to sleep or wake other threads.

Point 1 will always stand, but point 2 and 3 are of somewhat diminished usefulness if you consider that good mutex implementations will probably spin a decent number of times before asking the kernel for help waiting.

Now, the long answer:

What you need to ask yourself before using spinlocks is whether these potential advantages outweigh one rare but very real disadvantage: what happens when the thread that holds the lock gets interrupted by the scheduler before it can release the lock. This is of course rare, but it can happen even if the lock is just held for a single variable-increment operation or something else equally trivial. In this case, any other threads attempting to obtain the lock will keep spinning until the thread the holds the lock gets scheduled and has a chance to release the lock. **This may never happen** if the threads trying to obtain the lock have higher priorities than the thread that holds the lock. That may be an extreme case, but even without different priorities in play, there can be very long delays before the lock owner gets scheduled again, and worst of all, once this situation begins, it can quickly escalate as many threads, all hoping to get the lock, begin spinning on it, tying up more processor time, and further delaying the scheduling of the thread that could release the lock.

As such, I would be careful with spinlocks... :-)

answered Jul 7 '11 at 3:22
R..
103k 10 137 325

The spinlock is a "busy waiting" lock. It's main advantage is that it keeps the thread active and won't cause a context switch, so if you know that you will only be waiting for a very short time (because your critical operation is very quick), then this may give better performance than a mutex. Conversely a mutex will cause less demand on the system if the critical section takes a long time and a context switch is desirable.

TL;DR: It depends.

answered Jul 6 '11 at 21:47
Kerrek SB
213k 26 347 568

The safest method with a performance boost is a hybrid of the two: an adaptive mutex.

When your system has multiple cores you spin for a few thousand cycles to capture the best case of low or no contention, then defer to a full mutex to yield to other threads for long contended locks.

Both POSIX (`PTHREAD_MUTEX_ADAPTIVE_NP`) and Win32 (`SetCriticalSectionSpinCount`) have adaptive mutexes, many platforms don't have a POSIX spinlock API.

answered Oct 4 '12 at 20:25
Steve-o
9,542 2 17 38

Spinlock has only interest in MP context. It is used to execute pseudo-atomical tasks. In monoprocessor system the principle is the following :

1. Lock the scheduler (if the task deals with interrupts, lock interrupts instead)
2. Do my atomic tack
3. Unlock the scheduler

But in MP systems we have no guaranties that an other core will not execute an other thread that could enter our code section. To prevent this the spin lock has been created, its purpose is to postpone the other cores execution preventing concurrency issue. The critical section becomes :

1. Lock the scheduler
2. SpinLock (prevent entering of other cores)
3. My task
4. SpinUnlock
5. Task Unlock

If the task lock is omitted, during a scheduling, an other thread could try to enter the section an will loop at 100% CPU waiting next scheduling. If this task is an high-priority one, it will produce a deadlock.

edited Oct 4 '12 at 20:03
Rob Kielty
3,377 3 16 33

answered Oct 4 '12 at 16:22
Sebastien Kurz
1