When should one use a spinlock instead of mutex?

I think both are doing the same job,how do you decide which one to use for synchronization?

synchronization mutex spinlock

asked May 3 '11 at 13:01

compile-fan
2,723 8 30 45

1 possible duplicate of Spinlock Versus Semaphore! – Paul R May 3 '11 at 13:21

1 Mutex and Semaphore are not the same thing, so I don't think this is a duplicate. The Answer of the referenced article states this correctly. For more details see barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore – nanoquack Apr 28 '14 at 5:05

5 Answers

The Theory

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

The Problem

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time. If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept by far and it might even exceed the time the thread would have wasted by constantly polling on a spinlock. On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

The Solution

Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. IOW, a spinlock wastes only CPU time on those systems for no real benefit. If the thread was put to sleep instead, another thread could have ran at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.

On a multi-core/multi-CPU systems, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput.

The Practice

Since very often programmers cannot know in advance if mutexes or spinlocks will be better (e.g. because the number of CPU cores of the target architecture is unknown), nor can operating systems know if a certain piece of code has been optimized for single-core or multi-core environments, most systems don't strictly distinguish between mutexes and spinlocks. In fact, most modern operating systems have hybrid mutexes and hybrid spinlocks. What does that actually mean?

A hybrid mutex behaves like a spinlock at first on a multi-core system. If a thread cannot lock the mutex, it won't be put to sleep immediately, since the mutex might get unlocked pretty soon, so instead the mutex will first behave exactly like a spinlock. Only if the lock has still not been obtained after a certain amount of time (or retries or any other measuring factor), the thread is really put to sleep. If the same system runs on a system with only a single core, the mutex will not spinlock, though, as, see above, that would not be beneficial.

A hybrid spinlock behaves like a normal spinlock at first, but to avoid wasting too much CPU time, it may have a back-off strategy. It will usually not put the thread to sleep (since you don't want that to happen when using a spinlock), but it may decide to stop the thread (either immediately or after a certain amount of time) and allow another thread to run, thus increasing chances that the spinlock is unlocked (a pure thread switch is usually less expensive than one that involves putting a thread to sleep and waking it up again later on, though not by far).

Summary

If in doubt, use mutexes, they are usually the better choice and most modern systems will allow them to spinlock for a very short amount of time, if this seems beneficial. Using spinlocks can sometimes improve performance, but only under certain conditions and the fact that you are in doubt rather tells me, that you are not working on any project currently where a spinlock might be beneficial. You might consider using your own "lock object", that can either use a spinlock or a mutex internally (e.g. this behavior could be configurable when creating such an object), initially use mutexes everywhere and if you think that using a spinlock somewhere might really help, give it a try and compare the results (e.g. using a profiler), but be sure to test both cases, a single-core and a multi-core system before you jump to conclusions (and possibly different operating systems, if your code will be cross-platform).

edited Mar 7 '14 at 0:00

Kevin Ballard
96.8k 11 209 240

answered May 3 '11 at 13:44

Mecki
40.5k 18 92 133

4 Thanks for such a great explanation. **Bows** – Myth17 Oct 2 '13 at 19:23

superb explaination... I have a doubt regarding spinlock i,e can i use a spinlock in ISR ? if no why not – haris Mar 31 '14 at 17:45

1 @Mecki If I'm not mistaken, I believe you suggested in your answer that time slicing only occurs on single-processor systems. This is not correct! You can use a spin lock on a single-processor system and it will spin until its time quantum expires. Then another thread of the same priority can take over (just like what you described for multi-processor systems). – fumoboy007 Apr 8 '14 at 21:44

@fumoboy007 "and it will spin until its time quantum expires" // Which means you waste CPU time/battery power for absolutely nothing w/o any single benefit, which is utterly moronic. And no, I nowhere said that time slicing only happens on single core systems, I said on single core systems there is *ONLY* time slicing, while there is *REAL* parallelism om multicore systems (and also time slicing, yet irrelevant for what I wrote in my reply); also you entirely missed the point on what a hybrid spinlock is and why it works well on single and multicore systems. – Mecki Apr 9 '14 at 13:42

2 @fumoboy007 Thread A holds the lock and is interrupted. Thread B runs and wants the lock, but cannot get it, so it spins. On a multicore system, Thread A can continue to run on another core while Thread B is still spinning, release the lock, and Thread B can continue within its current time quantum. On a single core system, there is only one core Thread A could run to release the lock and this core is held busy by Thread B spinning. So there is no way the spinlock could ever be released before Thread B exceeded its time quantum and thus all spinning is just a waste of time. – Mecki Apr 10 '14 at 20:32

Continuing with Mecki's suggestion, this article pthread mutex vs pthread spinlock on Alexander Sandler's blog, Alex on Linux shows how the `spinlock` & `mutexes` can be implemented to test the behavior using #ifdef.

However, be sure to take the final call based on your observation, understanding as the example given is an isolated case, your project requirement, environment may be entirely different.

edited Nov 5 '14 at 10:25
reevesy
2,576 1 14 17

answered May 16 '11 at 10:36
TheCottonSilk
3,236 1 16 34

Spinlock and Mutex synchronization mechanisms are very common today to be seen.

Let's think about Spinlock first.

Basically it is a busy waiting action, which means that we have to wait for a specified lock is released before we can proceed with the next action. Conceptually very simple, while implementing it is not on the case. For example: If the lock has not been released then the thread was swap-out and get into the sleep state, should do we deal with it? How to deal with synchronization locks when two threads simultaneously request access ?

Generally, the most intuitive idea is dealing with synchronization via a variable to protect the critical section. The concept of Mutex is similar, but they are still different. Focus on: CPU utilization. Spinlock consumes CPU time to wait for do the action, and therefore, we can sum up the difference between the two:

In homogeneous multi-core environments, if the time spend on critical section is small than use Spinlock, because we can reduce the context switch time. (Single-core comparison is not important, because some systems implementation Spinlock in the middle of the switch)

In Windows, using Spinlock will upgrade the thread to DISPATCH_LEVEL, which in some cases may be not allowed, so this time we had to use a Mutex (APC_LEVEL).

answered Nov 8 '13 at 2:28
Marcus Thornton
923 2 8 22

Please also note that on certain environments and conditions (such as running on windows on dispatch level >= DISPATCH LEVEL), you cannot use mutex but rather spinlock. On unix - same thing.

Here is equivalent question on competitor stackexchange unix site:
http://unix.stackexchange.com/questions/5107/why-are-spin-locks-good-choices-in-linux-kernel-design-instead-of-something-more

Info on dispatching on windows systems:
http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/IRQL_thread.doc

edited Jul 22 '13 at 11:40

answered Jul 22 '13 at 11:34
Dan Jobs
16 4

I currently have a situation where there are hundreds of objects, where each one occasionally needs to be locked. Because there are so many items, the probability of two or more threads trying to lock any one object simultaneously is very small. In this case I was thinking of using something like boost::detail::spinlock, in the knowledge that the locks will almost never need to actually 'spin', while avoiding the potential overhead of a mutex. But because I learned that mutexes can actually behave like spinlocks for a short while I think I'll just stick to std::mutex. Thanks.

edited Nov 22 '14 at 14:12

answered Nov 20 '14 at 13:29
Peter Ritter
1 1
