Recursive Lock (Mutex) vs Non-Recursive Lock (Mutex)

POSIX allows mutexes to be recursive. That means the same thread can lock the same mutex twice and won't deadlock. Of course it also needs to unlock it twice, otherwise no other thread can obtain the mutex. Not all systems supporting pthreads also support recursive mutexes, but if they want to be POSIX conform, they have to.

Other APIs (more high level APIs) also usually offer mutexes, often called Locks. Some systems/languages (e.g. Cocoa Objective-C) offer both, recursive and non recursive mutexes. Some languages also only offer one or the other one. E.g. in Java mutexes are always recursive (the same thread may twice "synchronize" on the same object). Depending on what other thread functionality they offer, not having recursive mutexes might be no problem, as they can easily be written yourself (I already implemented recursive mutexes myself on the basis of more simple mutex/condition operations).

What I don't really understand: What are non-recursive mutexes good for? Why would I want to have a thread deadlock if it locks the same mutex twice? Even high level languages that could avoid that (e.g. testing if this will deadlock and throwing an exception if it does) usually don't do that. They will let the thread deadlock instead.

Is this only for cases, where I accidentally lock it twice and only unlock it once and in case of a recursive mutex, it would be harder to find the problem, so instead I have it deadlock immediately to see where the incorrect lock appears? But couldn't I do the same with having a lock counter returned when unlocking and in a situation, where I'm sure I released the last lock and the counter is not zero, I can throw an exception or log the problem? Or is there any other, more useful use-case of non recursive mutexes that I fail to see? Or is it maybe just performance, as a non-recursive mutex can be slightly faster than a recursive one? However, I tested this and the difference is really not that big.

multithreading locking deadlock mutex

edited Oct 9 '08 at 16:28

asked Oct 9 '08 at 15:19
Mecki
40.5k 18 92 133

4 Answers

The difference between a recursive and non-recursive mutex has to do with ownership. In the case of a recursive mutex, the kernel has to keep track of the thread who actually obtained the mutex the first time around so that it can detect the difference between recursion vs. a different thread that should block instead. As another answer pointed out, there is a question of the additional overhead of this both in terms of memory to store this context and also the cycles required for maintaining it.

However, there are other considerations at play here too.

Because the recursive mutex has a sense of ownership, the thread that grabs the mutex must be the same thread that release the mutex. In the case of non-recursive mutexes, there is no sense of ownership and any thread can usually release the mutex no matter which thread originally took the mutex. In many cases, this type of "mutex" is really more of a semaphore action, where you are not necessarily using the mutex as an exclusion device but use it as synchronization or signaling device between two or more threads.

Another property that comes with a sense of ownership in a mutex is the ability to support priority inheritance. Because the kernel can track the thread owning the mutex and also the identity of all the blocker(s), in a priority threaded system it becomes possible to escalate the priority of the thread that currently owns the mutex to the priority of the highest priority thread that is currently blocking on the mutex. This inheritance prevents the problem of priority inversion that can occur in such cases. (Note that not all systems support priority inheritance on such mutexes, but it is another feature that becomes possible via the notion of ownership).

If you refer to classic VxWorks RTOS kernel, they define three mechanisms:

- **mutex** - supports recursion, and optionally priority inheritance

- **binary semaphore** - no recursion, no inheritance, simple exclusion, taker and giver does not have to be same thread, broadcast release available
- **counting semaphore** - no recursion or inheritance, acts as a coherent resource counter from any desired initial count, threads only block where net count against the resource is zero.

Again, this varies somewhat by platform - especially what they call these things, but this should be representative of the concepts and various mechanisms at play.

edited Jun 20 '09 at 11:54 answered Oct 10 '08 at 1:09
Tall Jeff ♦
6,304 4 30 52

your explanation about non-recursive mutex sounded more like a semaphore. A mutex (whether recursive or non-recursive) has a notion of ownership. – Jay D Jan 7 '11 at 22:14

@JayD It's very confusing when people argue about things like these.. so who's the entity that defines these things? – Pacerier Dec 8 '11 at 16:09

2 @Pacerier The relevant standard. This answer is e.g. wrong for posix (pthreads) , where unlocking a normal mutex in a thread other than the thread that locked it is undefined behavior, while doing the same with an error checking or recursive mutex results in a predicable error code. Other systems and standards might behave very different. – nos Aug 6 '12 at 21:26

2 That is awe-Really-some..Thanks – jparthj Feb 14 '13 at 5:12

Perhaps this is naive, but I was under the impression that the central idea of a mutex is that the locking thread unlocks the mutex and then other threads may do the same. From computing.llnl.gov/tutorials/pthreads: – user657862 Oct 22 '13 at 17:31

The answer is *not* efficiency. Non-reentrant mutexes lead to better code.

Example: A::foo() acquires the lock. It then calls B::bar(). This worked fine when you wrote it. But sometime later someone changes B::bar() to call A::baz(), which also acquires the lock.

Well, if you don't have recursive mutexes, this deadlocks. If you do have them, it runs, but it may break. A::foo() may have left the object in an inconsistent state before calling bar(), on the assumption that baz() couldn't get run because it also acquires the mutex. But it probably shouldn't run! The person who wrote A::foo() assumed that nobody could call A::baz() at the same time - that's the entire reason that both of those methods acquired the lock.

The right mental model for using mutexes: The mutex protects an invariant. When the mutex is held, the invariant may change, but before releasing the mutex, the invariant is re-established. Reentrant locks are dangerous because the second time you acquire the lock you can't be sure the invariant is true any more.

If you are happy with reentrant locks, it is only because you have not had to debug a problem like this before. Java has non-reentrant locks these days in java.util.concurrent.locks, by the way.

edited Nov 2 '12 at 14:16 answered Nov 16 '08 at 7:44
David Harkness Jonathan
18.2k 2 38 75 **1,100** 6 6

2 It took me a while to get what you were saying about the invariant not being valid when you grab the lock a second time. Good point! What if it were a read-write lock (like Java's ReadWriteLock) and you acquired the read lock and then re-acquired the read lock a second time in the same thread. You wouldn't invalidate an invariant after acquiring a read lock right? So when you acquire the second read lock, the invariant is still true. – dgrant Jul 9 '11 at 0:29

@Jonathan Does *Java has non-reentrant locks these days in java.util.concurrent.locks*?? – user454322 Oct 13 '12 at 19:49

+1 I guess, that the most common use for reentrant lock is inside of a single class, where some methods can be called from both guarded and non-guarded code pieces. This can be actually always factored out. @user454322 Sure, `Semaphore` . – maaartinus Sep 27 '14 at 20:45

As written by Dave Butenhof himself:

"The biggest of all the big problems with recursive mutexes is that they encourage you to completely lose track of your locking scheme and scope. This is deadly. Evil. It's the "thread eater". You hold locks for the absolutely shortest possible time. Period. Always. If you're calling something with a lock held simply because you don't know it's held, or because you don't know whether the callee needs the mutex, then you're holding it too long. You're aiming a shotgun at your application and pulling the trigger. You presumably started using threads to get concurrency; but you've just PREVENTED concurrency."

answered Aug 7 '09 at 14:20

Chris Cleeland
2,359 1 9 18

5 Also note the final part in Butenhof's response: `...you're not DONE until they're [recursive mutex] all gone.. Or sit back and let someone else do the design.` – user454322 Oct 13 '12 at 19:04

> The right mental model for using mutexes: The mutex protects an invariant.

Why are you sure that this is really right mental model for using mutexes? I think right model is protecting data but not invariants.

The problem of protecting invariants presents even in single-threaded applications and has nothing common with multi-threading and mutexes.

Furthermore, if you need to protect invariants, you still may use binary semaphore wich is never recursive.

answered Dec 16 '08 at 11:41

Corpse

True. There are better mechanisms to protect an invariant. – ActiveTrayPrntrTagDataStrDrvr Feb 24 '14 at 17:51