Home
Resources
- Procurement Dept.
- Department of HR
- Library
- Training
- Restroom

Recursive mutexes by David Butenhof

Original could be found here: Google Groups

```
Newsgroups: comp.programming.threads
From: David Butenhof
Date: Tue, 17 May 2005 17:19:22 GMT
Local: Tues,May 17 2005 1:19 pm
Subject: Re: recursive mutexes

Uenal Mutlu wrote:
> "David Schwartz" wrote
>>    We really mean what we're saying. Really, really. Recursive mutexes are
>>really bad and they really do hide serious bugs.

> This is simply not true. Recursive locking is a superset of non-recursive locking.
> Everthing possible in non-recursive locking is possible in recursive-locking too,
> except deadlocking himself. So then how can recursive-locking be more dangerous
> than non-recursive locking? This is simple basic logic.
```

Simple, basic logic.

First, implementation of efficient and reliable threaded code revolves around one simple and basic principle: follow your design. That implies, of course, that you have a design, and that you understand it.

A correct and well understood design does not require recursive mutexes. While recursive mutexes may seem a convenience for debugging, in fact it's the opposite -- you're better off with a "strict" debugging mutex (like the POSIX error-check attribute) that checks, tracks, and enforces ownership restrictions a "normal" mutex may ignore in favor of runtime efficiency.

Many implementations may have arrived at the concept of recursive mutexes for any number of reasons -- some perhaps even because someone really thought they were a good idea. But allow me to explain, for the sake of context, why POSIX has recursive mutexes. Bear with me, because I'll follow into some more objective commentary.

In the early days of POSIX, we were also working with the Open Software Foundation to provide a thread library for DCE (known commonly as "DCE threads" or sometimes "D4 threads" because the interface vaguely resembles that of the draft 4 version of the POSIX threads amendment). We came to the realization that the biggest contraint was compatibility with existing NON-threaded operating systems.

The biggest problem with threading existing code is that locking requires analysis and understanding of the data and code relationships. That can be a stupendous and impractical job for something the size and complexity of, for example, the typical C runtime of a non-threaded operating system. Especially when you consider that we were supplying reference code for upwards of a dozen operating systems. Most (though not all) were from the "UNIX" family -- but of vastly differing branches. Analyzing and repairing even one was infeasible, and we couldn't ignore any of them.

There's one common idiom to deal with this sort of task, external locking: (lock_mutex(&a); x = malloc(size); unlock_mutex(&a);). But of course every facility in the process that calls malloc() needs to agree on a single mutex "a". And because something you call while holding the lock might also call malloc(), the lock must have the property that the owning thread can re-lock without deadlocking.

But unless you can do enough analysis to identify all possible execution paths, you can only use a single mutex within the process: a "global lock". There need be only one; there CAN be only one. Because if you know that it's OK to have more than one, you don't need any at all; you can simply lock properly in the first place, where needed.

And so DCE threads has pthread_lock_global() and pthread_unlock_global(). But if that's all that's necessary, why does POSIX have recursive mutexes?

Because of a dare.

We were pushing in the POSIX working group for our concept of attributes objects. And in particular the idea that one could support a range of potentiallyf useful and non-standard fundamental mutex behaviors without substantially complicating a simple and fast "inline lock" code path or invalidating basic POSIX semantics; that is, all the complication would be kept out of the main and common lock code. Some people just didn't believe us.

So I proved it by generalizing "the global lock" into a recursive mutex attribute. Of course it worked, though we never actually bothered to DO anything with the proof. However, having implemented it, I didn't bother to delete it, and it went into our code base. And that made it part of DCE threads on the next code drop to OSF. And it seemed silly to have it and not document it. Besides, I also built a strict "error-check" mutex type that would rigidly enforce mutex ownership, and that was useful for debugging.

But nobody was supposed to use recursive mutexes. For the original intended purpose, only the global mutex would work anyway. And if you could analyze the code paths enough to know that a separate mutex was safe, why the heck would anyone want the overhead and complication of a recursive mutex instead of just doing it right? I still didn't delete it, but I more or less stopped thinking about it for many years. POSIX finally became threaded with the approval of POSIX 1003.1c-1995, and POSIX 1003.1, 1996 edition, integrated it all into a single document.

And then along came The Open Group, which had already successfully tied together the "1170" common interfaces of disparate UNIX environments into a single portable specification, UNIX 93. And then followed with UNIX 95, adding more common features. All very nice. And now they were working on UNIX 98, and it would include threads.

But they didn't want just "POSIX threads". They wanted common and reasonably widely accepted EXTENSIONS to threads. Many of these extensions were really useful. Some were utterly stupid (like pthread_setconcurrency(), meaningful only to pitifully busted 2-level scheduler hacks, though I won't say any more at the risk of beginning to sound a little biased ;-)). In particular, though, almost everyone thought that recursive mutexes should be in there. And they are.

OK, I said I'd actually comment on the objective facts. So here are a couple...

1) 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.

I've often joked that instead of picking up Djikstra's cute acronym we should have called the basic synchronization object "the bottleneck". Bottlenecks are useful at times, sometimes indispensible -- but they're never GOOD. At best they're a necessary evil. Anything. ANYTHING that encourages anyone to overuse them, to hold them too long, is bad. It's NOT just the straightline performance impact of the extra recursion logic in the mutex lock and unlock that's the issue here -- it's the far greater, wider, and much more difficult to characterize impact on the overall application concurrency.

People often say "I added threads to my application and it got slower. Stupid threads". And the answer is almost always, no (but we'll be more tactful here), "uninformed programmer". They forget to unlock when they need to, because they forget that where you unlock is just as important as where you lock. Threading is NOT just about about "a different model for application structure", it's about concurrency. Locks kill concurrency. Locks held longer than necessary for "safety" kill concurrency even worse.

2) Sometimes you need to unlock. Even if you're using recursive mutexes. But how do you know how to unlock? Threaded programming is built around predicates and shared data. When you hand a recursive mutex down from one routine to another, the callee cannot know the state of predicates in the caller. It has to assume there are some, because it cannot verify there aren't; and if the caller had known that there were no broken predicates, it should have allowed concurrency by unlocking.

So how can you wait? You need to release (not just unlock) the mutex in order to allow some other thread to change a predicate. But if you release, you've left your predicates dangling in the wind... unchecked, unknown, unprotected. That's idiotic design, and the most fundamental error in the Java language. "Don't call while holding broken predicates", is all they can say by way of excuse. But if there are no broken predicates, you UNLOCK so the application can have a chance to act concurrent. If you're ever going to design a language that tries to do this, make sure it has real first-class support for predicates, so that it understands who they are and what they mean, and can make decisions like this for you, reliably. At the very least it has to be able to diagnose when you blow it... and Java can't even do that.

POSIX, luckily, doesn't provide the mechanism to perform this sort of data demolition. You can only unlock, and you cannot detect when an unlock will release. That is, when you call pthread_cond_wait() on a recursive mutex, you may NOT release... and in that case you've deadlocked. You'll never continue from your predicate loop until some other thread changes the predicate, which it can't do because you hold the lock. The rest of the application may or may not eventually come to a halt, but you sure haven't done it any good. You're squeezing the entire application through your bottleneck.

Recursive mutexes are a hack. There's nothing wrong with using them, but they're a crutch. Got a broken leg or library? Fine, use the crutch. But at least be aware that you're using a crutch, and why; and once in a while check out the leg (or library) to be sure you still need the crutch. And if it's not healing up, go see a doctor, because that's just not OK. When you have no choice, there's no shame in using a crutch... but you can't run very well on a crutch, and you'll also be slowing down anyone who depends on you.

Recursive mutexes can be a great tool for prototyping thread support in an existing library, exactly because it lets you defer the hard part: the call path and data dependency analysis of the library. But for that same reason, always remember that you're not DONE until they're all gone, so you can produce a library you're proud of, that won't unnecessarily contrain the concurrency of the entire application.

Or sit back and let someone else do the design.

--
Dave Butenhof, David.Buten...@hp.com
HP Utility Pricing software, POSIX thread consultant
Manageability Solutions Lab (MSL), Hewlett-Packard Company
110 Spit Brook Road, ZK2/3-Q18, Nashua, NH 03062