

Re: Locking etc. (Long, boring, redundant, newbie questions)

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/hackers/2007-03/msg00230.html>

- *From:* Robert Watson <rwatson@xxxxxxxxxxxxx>
 - *Date:* Wed, 28 Mar 2007 10:40:58 +0100 (BST)
-

On Wed, 28 Mar 2007, Duane Whitty wrote:

I know this is trivial material but I believe I have finally come to an understanding about some things that I have been struggling with. Maybe by putting this in words I can help some other newbies like myself who are hoping to eventually understand more advanced topics and contribute to FreeBSD. But please, correct me if you would on topics I still do not have correct. And thanks in advance for your help and patience!

We are working to improve our documentation in this area, but there are a number of issues relating to consistent use of terminology, some of which you are running into.

I have been reading non-stop as much as I can on synchronizing code execution. In the FreeBSD docs, and in other docs, there is talk about mutexes that block and mutexes that sleep. But this is not true is it. What is really meant is that depending on the type of mutex a thread is trying to acquire, the thread will either spin or it will sleep waiting for the lock to become available. Am I correct so far?

We basically have two kinds of mutexes: mutexes that only spin, and mutexes that may also sleep. The former category is intended for use in synchronizing with fast interrupts and in the scheduler, and are called "spin mutexes". The latter are intended for use in pretty much any other case, and are called "default mutexes". In terms of implementation, the main behaviors are:

- Spin mutexes will never sleep, and hence can be used in a borrowed execution context during interrupt delivery, and likewise in the scheduler (such as in implementing sleep). They disable interrupt delivery on the current CPU, and as such, are quite expensive to acquire on some architectures.
- Default mutexes may sleep, but by default are "adaptive", meaning that they will try spinning where it makes sense (i.e., when the holder of the lock is executing on another CPU).

Unless you are working in the scheduler or synchronizing in/with a fast interrupt handler, do not use spin mutexes.

Re: Locking etc. (Long, boring, redundant, newbie questions)

Maybe this method of talking about mutexes happens because we don't manipulate lock structures directly, but rather use routines which acquire these locks for us in a consistent way. So for instance when we call `mtx_lock(&some_lock)` and the lock is contested, our thread sleeps. It gets put on a sleep queue waiting for the lock to become available so that we can safely access the kernel data structure which this mutex protects. Is this accurate so far?

Yes, although for reasons of optimization, when contending a lock we may spin instead of sleeping if the thread holding the mutex is in the run state. This avoids the overhead of putting the current thread to sleep and then waking it up later. The benefits of this optimization are significant and easily measurable.

Along the same line as above, if we call `mtx_lock_spin(&some_lock)`, and the lock is contested, our thread trying to acquire the lock spins. This means we go into a tight loop monopolizing whichever CPU we are running on until the mutex becomes available. But, if we spin for so long that we use up our quantum of time scheduled to us, a panic happens, because when we try to acquire a spin mutex, interrupts are turned off and so we can't do a context switch. If a thread slept with interrupts disabled, then interrupts would stay disabled, which must not happen.

Pretty much. We disable interrupts for the following reason: as spin mutexes may be acquired in fast interrupt handlers, they may be running on the stack of an existing thread, which may also hold locks. As such, we can't allow the fast handler to acquire any locks that are either already held by the thread, or that might violate the lock order. By restricting fast interrupt handlers to holding only spin locks, and by making spin locks disable interrupts, we prevent that deadlock scenario. "Slow" interrupt handlers run in complete thread contexts, called `itthreads`, and are, as such, able to acquire default mutexes and sleep in the scheduler. In FreeBSD 7.x, we have moved to a model in which device drivers can register both fast and threaded handlers, whereas in 6.x they had to pick one (and hence if they needed both, they had to pick the fast handler and use a task queue for threaded work).

I'm not very sure on this point, but is the above the reason why interrupt service routines, also known as Fast ISRs (?), use `mtx_lock_spin()` mutexes? They are supposed to be as fast as possible, and they don't context switch. As well, isn't it basically agreed upon that Fast ISRs are really the only place to use spin mutexes? Maybe I'm way off here but it sure would be nice finally putting this one away.

Spin locks are, FYI, slower than default mutexes. The reason is that they have to do more work: they not only perform an atomic operation/memory barrier to set the cross-CPU lock state, but they also have to disable interrupts to synchronize with fast interrupt handlers. In general, you are right: you should only use a spin mutex if you are running in a fast handler, or synchronizing with a fast handler. The one general exception is the scheduler itself, which must protect its data structures with spin locks in order to implement sleeping primitives. As such, the scheduler lock and various other low level locks (such as in turnstiles) are implemented with spin locks and not default mutexes. Since default mutexes spin adaptively, the reduced overhead of contention experienced with spin locks (i.e., no scheduler overhead for simple contention cases) is also experienced with default mutexes.

Re: Locking etc. (Long, boring, redundant, newbie questions)

A somewhat newer (?) type of locking procedure is where a thread will spin when trying to acquire a lock which is contested, for about the same amount of time as it would take to do a context switch. If this time is roughly (approached|exceeded)? then the thread sleeps. The reasoning is that in the time it would take to do a context switch if the thread were to sleep, the lock the thread is trying to acquire may become available. If we get lucky and the lock becomes available while we are spinning then we have been more efficient. I hope I am still on the right track here?

This is essentially correct. This technique, called an "adaptive mutex" is used in Solaris and FreeBSD (and probably others). Right now, we use a simple strategy to decide whether to spin or not: we check to see if the thread holding the lock is currently running. If it's not, we assume it will be held for a long time and put the contending thread to sleep, falling back to historic sleep mutex behavior. There has been some investigating of more mature strategies; for example, Solaris uses a back-off scheme, I believe.

Another item in the documentation I had confused as being part of the above issue(s) is that it is permissible for a thread to sleep while holding certain types of locks and that for other types of locks a thread must not sleep. Am I correct in my understanding now that a thread must not sleep while holding a mutex because doing so could lead to a deadlock?

There are really two notions of sleep: bounded, and unbounded. The difference is a bit sketchy if you try to nail it down too precisely, but intuitively (and in practice) it works quite well:

- While holding a spin mutex or in a critical section, it is not permitted to perform any activity that might cause the current thread to sleep in any form.

- While holding a default mutex or rwlock, it is not permitted to perform any activity that might cause the current thread to sleep in an unbounded way.

I.e., you can hold one default mutex and sleep waiting for a second sleep mutex, since the "unbounded" property is considered to propagate back through callers. However, because default mutexes may be acquired in interrupt thread contexts, if a series of locks become dependent on one another due to multiple acquisitions, you could end up with nasty deadlocks not detectable by the lock order checker: an ithread could end up waiting, via mutexes, on a thread that is effectively waiting on the ithread. For example, a storage device driver ithread could be waiting on locks held by a thread sleeping in the file system code waiting on a block read, leading to deadlock.

The restriction of not performing unbounded sleeps with a default mutex is conservative: after all, there are certainly mutexes that in principle could be safe to hold across an unbounded sleep as they are never acquired or depended on in an interrupt thread context. Solaris takes the approach of allowing this sort of sleep. One proposal has been to change our mutex definition so that mutexes can be declared as "top half", meaning that they are only ever acquired outside of ithread context, and an unbounded sleep might occur while the mutex is held. The lock order checker can then check that a mutex that is not "top half" is never acquired before a "top half" mutex, which could lead to a deadlock. The FreeBSD approach, however, has generally been to use different lock types for "sleepable" vs "non-sleepable" locks, which can make things a lot easier to inspect and verify.

Re: Locking etc. (Long, boring, redundant, newbie questions)

Another scenario I have just learned is that a semaphore should not be wrapped inside a mutex, for the same reason; a deadlock. I guess this is just a specific case of not allowing the possibility of a thread sleeping while holding a mutex, which is what could happen if a thread is waiting on any kind of condition variable?

Waiting on a semaphore, a condition variable, or a tsleep/msleep counts as a potentially unbounded wait, and therefore is not allowed while holding a mutex or rwlock. It is allowed using an sx lock, which is "sleepable". Note, btw, that the kernel semaphore implementation (sema(9)) is not the same as the System V IPC semaphore implementation (semctl(2)) or the POSIX semaphore implementation (sem_init(3)).

If I am mostly correct (I hope), my enlightenment, if I have any came about when I started working on the System V IPC code. When I started trying to learn about semaphores I found out I needed to learn about a lot of other stuff. And still do. Perhaps the most difficult thing is translating from textbook type examples to real kernel code. Anyhow, I hope this message wasn't too long or unbearable. Please send me any comments that you would like to, on or off list.

In general, we discourage the use of sema(9) synchronization in the kernel; it's in use for a few specific subsystems, but we would prefer that you use default mutexes, rwlocks, sx locks, and condition variables/msleep. These in-kernel facilities can be used to implement user-visible semaphores.

Robert N M Watson
Computer Laboratory
University of Cambridge

Thanks again!

Duane Whitty

P.S.

I think I'm just about
ready to tackle fixing
sysv sems to wakeup
just the proper number
of processes, now that I
think I know what an
up'ed semaphore is, what
a semaphore set is for, how
semops work, and adjust on
exit values, and not least,
undos. My head hurts :-)

freebsd-hackers@xxxxxxxxxxx mailing list
<http://lists.freebsd.org/mailman/listinfo/freebsd-hackers>
To unsubscribe, send any mail to "freebsd-hackers-unsubscribe@xxxxxxxxxxx"

Re: Locking etc. (Long, boring, redundant, newbie questions)

Re: Locking etc. (Long, boring, redundant, newbie questions)

freebsd-hackers@xxxxxxxxxxx mailing list

<http://lists.freebsd.org/mailman/listinfo/freebsd-hackers>

To unsubscribe, send any mail to "freebsd-hackers-unsubscribe@xxxxxxxxxxx"