

Re: [PATCH 0/3] 64-bit futexes: Intro

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-06/msg00762.html>

- *From:* Linus Torvalds <torvalds@xxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Mon, 2 Jun 2008 13:22:57 -0700 (PDT)
-

[I added Nick and DavidH to the Cc, since they have at least historically shown interest in locking algorithms]

On Mon, 2 Jun 2008, Ingo Molnar wrote:

i suspect `_any_` abstract locking functionality around a data structure can be implemented via atomic control over just a single user-space bit.

Well, theory is theory, and practice is different.

That's **especially** true when it comes to locking, which is just so tightly coupled to the exact details of which atomics are fast on a particular architecture.

Also, different people will want to see different performance profiles.

For example, it makes a **huge** difference whether you are strictly fair or not. I can almost guarantee that a 100% fair implementation may be really "nice" from a theoretical standpoint, but suck really badly in practice, because if you want best performance, then you want the lock to have a very strong CPU affinity – and the faster you can do your lock ops, the more of a unfairness and CPU affinity they get!

And rwlocks in particular are actually much more interesting in this respect, because they not only have that CPU affinity fairness, but also the reader-vs-writer fairness. You optimize for one load, and it may give you almost perfect performance, but at the cost of sucking at another load.

For example, some loads are almost entirely read-read locks, with only very occasional write locks for some uncommon config change thing. Do you want to optimize for that? Maybe. And yes, you can make those uncontended read-read locks go really quickly, but then (especially if you continue to let reads through even when writers want to contend), that can slow down writers a **lot**, to the point of starvation.

Different CPU's will also show different patterns.

Anyway, I was busy most of the weekend, but I've now coded up a partial actual example implementation. Its' probably buggy. Uli is very correct in saying that it's easy to screw up futex'es, but even in the absense of futexes, it's just easy to screw up any threaded logic.

But if anybody wants to look at a rough draft that at least limps along `_partially_`, there's

<http://git.kernel.org/?p=linux/kernel/git/torvalds/rwlock.git;a=summary>

for you.

And I'll freely admit that

- (a) the above thing is pretty hacked up. No guarantees as to correctness or anything else.
- (b) I looked `_mainly_` at the "all readers" case, and considered that the primary goal. Which explains why it does really well on that particular load.
- (c) However, I refuse to starve writers. In fact, my thing will always prefer a writer to any new readers, on the assumption that the sanest `rwlock` situation is the "tons of readers, occasional writer".
- (d) it does ok for me on the write-write contention case too, but the random mixed "all threads do 5% writelocks" test-case seems to suck.

As an exmample: the current `glibc` code I have seems to be uniformly and boringly middle-of-the-road. It really doesn't seem to be horrible at all. That said, the above thing `_is_ 2-3x` faster for me on that read-read case (from single thread up to 20 threads – but just tested on one particular machine), so if that is what you want to aim for, it's certainly easy to beat.

But for the write case, while I can easily beat it for a low-thread count with little contention, my example thing above benchmarks about equal for bigger thread counts (caveat: again – I've only tested on one machine, which is a single-socket dual-core thing. Caveat emptor).

And the `pthread`s implementation actually beats my hacked-up one at least for the 5% random writelocks case. It's not beating it by a huge amount, but it's not in the noise level either (maybe 15%). And I bet the `pthread`s implementation is a hell of a lot better tested ;)

That bit can be used as a lock and if all access to the state of that atomic variable uses it, arbitrary higher-order atomic state transitions can be derived from it. The cost would be a bit more instructions in the fastpath, but there would still only be a single atomic op (the `acquire op`), as the `unlock` would be a natural barrier (on x86 at least).

No, "unlocks as a natural barrier" only works for exclusive kernel locks (spin_unlock and write_unlock). There we can just do a write to unlock. But for anything that wants to handle contention differently than just spinning, the unlock path needs to be able to do an atomic "unlock and test if I need to do something else", because it may need to wake things up.

Feel free to try out my code, and laugh at it. Getting locking right is definitely not simple, and I would be really surprised if I got everything right. It's meant as a "hey, here's real code people can at least play with" for anybody who is interested in this kind of thing.

Linus

--

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>