

Re: rwlocks: poor performance with adaptive spinning

Re: rwlocks: poor performance with adaptive spinning

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/arch/2007-09/msg00055.html>

- *From:* John Baldwin <jhb@xxxxxxxxxxxx>
 - *Date:* Tue, 25 Sep 2007 13:14:42 -0400
-

On Monday 24 September 2007 04:57:06 pm Jeff Roberson wrote:

On Mon, 24 Sep 2007, John Baldwin wrote:

On Saturday 22 September 2007 10:32:06 pm Attilio Rao wrote:

Recently several people have reported problems of starvation with

rwlocks.

In particular, users which tried to use rwlock on big SMP environment (16+ CPUs) found them rather subjected to poor performances and to starvation of waiters.

Inspecting the code, something strange about adaptive spinning popped up: basically, for rwlocks, adaptive spinning stubs seem to be customized too down in the decisioning-loop. The desposition of the stub will let the thread that would adaptively spin, to set the respective (both read or write) waiters flag on, which means that the owner of the lock will go down in the hard path of locking functions and will performe a full wakeup even if the waiters queues can result empty. This is a big penalty for adaptive spinning which can make it completely useless. In addition to this, adaptive spinning only runs in the turnstile spinlock path which is not ideal. This patch ports the approach already used for adaptive spinning in sx locks to rwlocks:

Re: rwlocks: poor performance with adaptive spinning

http://users.gufi.org/~rookie/works/patches/kern_rwlock.diff

In sx it is unlikely to see big benefits because they are held for too long times, but for rwlocks situation is rather different. I would like to see if people can do benchmarks with this patch (maybe in private environments?) as I'm not able to do them in short times.

Adaptive spinning in rwlocks can be improved further with other tricks (like adding a backoff counter, for example, or trying to spin with the lock held in read mode too), but we first should be sure to start with a solid base.

I did this for mutexes and rwlocks over a year ago and Kris found it was slower in benchmarks. www.freebsd.org/~jhb/patches/lock_adapt.patch is

the

last thing I sent kris@ to test (it only has the mutex changes). This

might

be more optimal post-thread_lock since thread_lock seems to have heavily pessimized adaptive spinning because it now enqueues the thread and then dequeues it again before doing the adaptive spin. I liked the approach originally because it simplifies the code a lot. A separate issue is that writers don't spin at all if a reader holds the lock, and I think one

thing

to test for that would be an adaptive spin with a static timeout.

We don't enqueue the thread until the same place. We just acquire an extra spinlock. The thread is not enqueued until turnstile_wait() as before.

So I looked at this some more and now I don't understand why the trywait() and cancel() changes were done. Some background:

When the initial turnstile code was split out of the the mutex code, the main loop in mtx_lock_sleep() looked like this:

```
while (!_obtain_lock) {
```

Re: rwlocks: poor performance with adaptive spinning

Re: rwlocks: poor performance with adaptive spinning

```
ts = turnstile_lookup(); /* (A) locked ts chain
and did O(n) lookup
to find existing ts */
```

```
if (lock unlocked) { (B)
turnstile_release();
continue;
}
```

```
if (failed to set contested flag) { (C)
turnstile_release();
continue;
}
```

```
if (owner is running) { (D)
turnstile_release();
... spin ...
continue;
}
```

```
turnstile_wait(ts, ...); (E)
}
```

So one thing I noticed after a while is that every iteration of this loop was doing the O(n) lookup to find the turnstile even though we only used that in (E). The lookup was wasted overhead for the (B), (C), and (D) cases. Hence this commit a while later:

jhb 2004-10-12 18:36:20 UTC

FreeBSD src repository

Modified files:

```
sys/kern kern_condvar.c kern_mutex.c kern_synch.c
subr_sleepqueue.c subr_turnstile.c
sys/sys sleepqueue.h turnstile.h
```

Log:

Refine the turnstile and sleep queue interfaces just a bit:

- Add a new `_lock()` call to each API that locks the associated chain lock for a `lock_object` pointer or wait channel. The `_lookup()` functions now require that the chain lock be locked via `_lock()` when they are called.
- Change `sleepq_add()`, `turnstile_wait()` and `turnstile_claim()` to lookup the associated queue structure internally via `_lookup()` rather than accepting a pointer from the caller. For turnstiles, this means that the actual lookup of the turnstile in the hash table is only done when the thread actually blocks rather than being done on each loop iteration in `_mtx_lock_sleep()`. For sleep queues, this means that `sleepq_lookup()` is no longer used outside of the sleep queue code except to implement an assertion in `cv_destroy()`.
- Change `sleepq_broadcast()` and `sleepq_signal()` to require that the chain lock is already required. For condition variables, this lets the

Re: rwlocks: poor performance with adaptive spinning

Re: rwlocks: poor performance with adaptive spinning

cv_broadcast() and cv_signal() functions lock the sleep queue chain lock while testing the waiters count. This means that the waiters count internal to condition variables is no longer protected by the interlock mutex and cv_broadcast() and cv_signal() now no longer require that the interlock be held when they are called. This lets consumers of condition variables drop the lock before waking other threads which can result in fewer context switches.

MFC after: 1 month

Revision Changes Path

1.52 +16 -15 src/sys/kern/kern_condvar.c
1.151 +4 -5 src/sys/kern/kern_mutex.c
1.263 +4 -3 src/sys/kern/kern_synch.c
1.13 +31 -14 src/sys/kern/subr_sleepqueue.c
1.150 +34 -12 src/sys/kern/subr_turnstile.c
1.5 +11 -12 src/sys/sys/sleepqueue.h
1.5 +17 -16 src/sys/sys/turnstile.h

After that commit the mutex loop looked like this:

```
while (!_obtain_lock) {  
    turnstile_lock(); /* (A) locked ts chain only */  
  
    if (lock unlocked) { (B)  
        turnstile_release();  
        continue;  
    }  
  
    if (failed to set contested flag) { (C)  
        turnstile_release();  
        continue;  
    }  
  
    if (owner is running) { (D)  
        turnstile_release();  
        ... spin ...  
        continue;  
    }  
  
    turnstile_wait(ts, ...); /* (E) Does O(n) lookup */  
}
```

That is, after the above commit, we only did the O(n) lookup if we needed the actual turnstile object in (E). This decreased the overhead for the (B), (C), and (D) cases.

The changes to add trywait() and cancel() basically seem to have reverted the above commit by going back to doing the O(n) lookup in (A) thus re-adding the overhead to the (B), (C), and (D) cases. In addition trywait() / cancel() actually maintain some additional state and acquire another lock so that (B),

Re: rwlocks: poor performance with adaptive spinning

(C), and (D) have even more overhead than the first cut of the turnstile code including having to back out some of the changes in cancel() rather than just a single spinlock unlock.

This is actually why I had assumed you had enqueued the turnstile in trywait() as I thought you had dropped the chain lock in trywait() (which would require enqueuing the thread in trywait(), but would also greatly complicate cancel()). Otherwise, the current trywait() / cancel() changes seem like a step backwards to me.

Do you really think it is necessary to do the O(n) lookup even when we don't need it (B, C, D) rather than just doing it in turnstile_wait() (E)?

--

John Baldwin

freebsd-arch@xxxxxxxxxxx mailing list

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

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