

freebsd-arch: Re: mtx\_lock\_recurse/mtx\_unlock\_recurse functions (proof-of-concept).

## Re: mtx\_lock\_recurse/mtx\_unlock\_recurse functions (proof-of-concept).

**Source:** <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/arch/2004-04/0020.html>

---

**From:** Robert Watson ([rwatson\\_at\\_FreeBSD.org](mailto:rwatson_at_FreeBSD.org))

**Date:** 04/09/04

Date: Thu, 8 Apr 2004 18:33:00 -0400 (EDT)

To: Pawel Jakub Dawidek <[pjd@FreeBSD.org](mailto:pjd@FreeBSD.org)>

On Thu, 8 Apr 2004, Pawel Jakub Dawidek wrote:

- > *As was discussed, it will be helpful to have functions, that are able to*
- > *acquire lock recursively, even if lock itself isn't recursable.*
- >
- > *Here is a patch, that should implement this functionality:*
- >
- > [http://people.freebsd.org/~pjd/patches/mtx\\_lock\\_recurse.patch](http://people.freebsd.org/~pjd/patches/mtx_lock_recurse.patch)
- >
- > *I also added a KASSERT() to protect against mixed locking modes, e.g.:*

As you know, this is of interest to me because several situations have come up in the network stack locking code where we've worked around locking issues by conditionally acquiring a lock based on whether it's not already held. For example, in the `rwatson_netperf` branch, we conditionally acquire the socket buffer lock in the `KQueue` filter because we don't know if we're being called from a context that has already acquired the mutex:

```
@@ -1875,8 +1915,11 @@
filt_sowrite(struct knote *kn, long hint)
{
    struct socket *so = kn->kn_fp->f_data;
    - int result;
    + int needlock, result;

    + needlock = !SOCKBUF_OWNED(&so->so_snd);
    + if (needlock)
    + SOCKBUF_LOCK(&so->so_snd);
        kn->kn_data = sbspace(&so->so_snd);
        if (so->so_state & SS_CANTSENDMORE) {
            kn->kn_flags |= EV_EOF;
@@ -1891,6 +1934,8 @@
        result = (kn->kn_data >= kn->kn_sdata);
    else
```

Re: mtx\_lock\_recurse/mtx\_unlock\_recurse functions (proof-of-concept).

freebsd-arch: Re: mtx\_lock\_recurse/mtx\_unlock\_recurse functions (proof-of-concept).

```
    result = (kn->kn_data >= so->so_snd.sb_lowat);
+ if (needlock)
+ SOCKBUF_UNLOCK(&so->so_snd);
    return (result);
}
```

This stems directly from the fact that in general we wanted to avoid recursion on socket buffer mutexes, but that in one case due to existing implementation constraints, we don't know what state the lock will be in. In this case, that happens because invocation of filt\_sowrite() sometimes occurs directly from the KQueue code to poll a filter, and sometimes from the socket code during a wakeup that invokes KNOTE(). There are arguably at least two ways in which the current code might be modified to try and address this:

- (0) Use recursive mutexes so that this recursion is permitted.
- (1) Use separate APIs to enter the per-object filters so that the origins of the invocation (and hence locking state) are more clear.
- (2) Avoid holding existing locks over calls into KNOTE(), which goes off and does a lot of stuff.

Neither of these is 100% desirable, and I guess I'd prefer (1) over (2) since (2) conflicts with the use of locks as a form of reference counting. However, the easy answer (0) is also undesirable because the socket code generally doesn't need recursion of the socket buffer mutex. Given the ability to selectively say "Ok, recursion here is fine" we would change the above conditional locking into such a case.

Robert N M Watson FreeBSD Core Team, TrustedBSD Projects  
robert@fledge.watson.org Senior Research Scientist, McAfee Research

---

freebsd-arch@freebsd.org mailing list

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

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