

Re: em network issues

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/net/2006-10/msg00159.html>

- *From:* Scott Long <scottl@xxxxxxxxxxx>
 - *Date:* Wed, 18 Oct 2006 21:20:57 -0600
-

Bruce Evans wrote:

On Wed, 18 Oct 2006, Kris Kennaway wrote:

I have been working with someone's system that has em shared with fxp, and a simple fetch over the em (e.g. of a 10 GB file of zeroes) is enough to produce watchdog timeouts after a few seconds.

As previously mentioned, changing the INTR_FAST to INTR_MPSAFE in the driver avoids this problem. However, others are seeing sporadic watchdog timeouts at higher system load on non-shared em systems too.

em_intr_fast() has no locking whatsoever. I would be very surprised if it even seemed to work for SMP. For UP, masking of CPU interrupts (as is automatic in fast interrupt handlers) might provide sufficient locking, but for many drivers fast with interrupt handlers, whatever locking is used by the fast interrupt handler must be used all over the driver to protect data structures that are or might be accessed by the fast interrupt handler. That means lots of intr_disable/enable()s if the UP case is micro-optimized and lots of mtx_lock/unlock_spin()s for the general case. But em has no references to spinlocks or CPU interrupt disabling.

em_intr() starts with EM_LOCK(), so it isn't obviously broken near its first statement.

Very few operations are valid in fast interrupt handlers. Locking and fastness must be considered for every operation, not only in the interrupt handler but in all data structures shared by the interrupt handler. For just the interrupt handler in em:

```
% static void
% em_intr_fast(void *arg)
% {
% struct adapter *adapter = arg;
```

Re: em network issues

This is safe because it has no side effects and doesn't take long.

```
% struct ifnet *ifp;
% uint32_t reg_icr;
% % ifp = adapter->ifp;
%
```

This is safe provided other parts of the driver ensure that the interrupt handler is not reached after `adapter->ifp` goes away. Similarly for other long-lived almost-const parts of `*adapter`.

```
% reg_icr = E1000_READ_REG(&adapter->hw, ICR);
%
```

This is safe provided reading the register doesn't change it.

```
% /* Hot eject? */
% if (reg_icr == 0xffffffff)
% return;
% /* Definitely not our interrupt. */
% if (reg_icr == 0x0)
% return;
%
```

These are safe since we don't do anything with the result.

```
% /*
% * Starting with the 82571 chip, bit 31 should be used to
% * determine whether the interrupt belongs to us.
% */
% if (adapter->hw.mac_type >= em_82571 &&
% (reg_icr & E1000_ICR_INT_ASSERTED) == 0)
% return;
%
```

This is safe, as above.

```
% /*
% * Mask interrupts until the taskqueue is finished running. This is
% * cheap, just assume that it is needed. This also works around the
% * MSI message reordering errata on certain systems.
% */
% em_disable_intr(adapter);
```

Now that we start doing things, we have various races.

The above races to disable interrupts with other entries to this interrupt handler, and may race with other parts of the driver.

After we disable driver interrupts. There should be no more races with other entries to this handler. However, `reg_icr` may be stale at this

Re: em network issues

point even if we handled the race correctly. The other entries may have partly or completely handled the interrupt when we get back here (we should have locked just before here, and then if the lock blocked waiting for the other entries (which can only happen in the SMP case), we should reread the status register to see if we still have anything to do, or more importantly to see what we have to do now (extrascheduling of the SWI handler would just wake time, but missing scheduling would break things).

```
% taskqueue_enqueue(adapter->tq, &adapter->rxtx_task);  
%
```

Safe provided the API is correctly implemented. (AFAIK, the API only has huge design errors.)

```
/* Link status change */  
% if (reg_icr & (E1000_ICR_RXSEQ | E1000_ICR_LSC))  
% taskqueue_enqueue(taskqueue_fast, &adapter->link_task);  
%
```

As above, plus might miss this call if the status changed underneath us.

```
% if (reg_icr & E1000_ICR_RXO)  
% adapter->rx_overruns++;
```

Race updating the counter.

Generally, fast interrupt handlers should avoid book-keeping like this, since correct locking for it would poison large parts of the driver with the locking required for the fast interrupt handler.

This should be an atomic add. It doesn't require any extra synchronization beyond that. It is, however, only an informational counter, and one that I forgot to fix.

Perhaps similarly for important things. It's safe to read the status register provided reading it doesn't change it. Then it is safe to schedule tasks based on the contents of the register provided we don't do anything else and schedule enough tasks. But don't disable interrupts — leave that to the task and make the task do nothing if it handled everything for a previous scheduling.

There is no other way to quiet the interrupt line from the device. Some devices will quiet their interrupt if you read the ISR or write back to the ISR. The e1000 chip doesn't seem to be like this, and instead relies on waiting for you to clear the rx and tx rings. i.e. it won't clear until you do real work, and you can't do real work in the

Re: em network issues

INTR_FAST handler. So, if you remove the `em_disable_intr`, you'll get an interrupt storm in between this handler running and the taskqueue completing. That obviously won't work.

If there is a better way to quiet the interrupt here, please let me know. My conversation with Intel about this 9 months ago lead me to believe that my method here is correct (modulo what I'll say below). Note that the other INTR_FAST drivers I've done, namely `aac` and `uhci`, have hardware that does allow you to silence the interrupt from the handler without having to do expensive work.

This would result in the task usually being scheduled when the interrupt is for us but not if it is for another device. The above doesn't try to do much more than this. However, a fast interrupt handler needs to handle the usual case to be worth having except on systems where there are lots of shared interrupts.

The performance measurements that Andre and I did early this year showed that the INTR_FAST handler provided a very large benefit. However, there are 2 things that I need to fix here:

1) The unprotected bookkeeping in the handler. Like I said above, this is informational only, and can be protected just fine with a single atomic add.

2) Protect top-half threads that want to call `em_intr_disable/enable` from the interrupt handler and taskqueue. IIRC, linux does this with a simple semaphore-like count. That means extra atomic ops and probably spinlocks, neither of which would be very good for this driver in FreeBSD. My plan here instead is to have the top-half callers set a 'paused' flag and drain the taskqueue before calling `em_disable_intr`. That way they know that there are no tasks scheduled or waiting on a lock release or in progress, and thus no chance that `em_enable_intr` will be called behind their back. I know that there is still a tricky race here, so I haven't implemented it yet. In any case, quiescing the interrupt and taskqueue is expensive, but it only needs to be done for fairly expensive and infrequent conditions (`ioctl`s, `unload/detach`).

(2) is the biggest flaw with the current INTR_FAST implementation. However, the biggest effect of it would be an extra interrupt during an `ioctl` operation, and it's unclear whether this actually poses a problem. The `unload/detach` case already drains the taskqueue, though in a different spot, so it's not a big problem there either.

The problem you pointed out with the cached copy of the interrupt status possibly going stale is a calculated risk. If a new condition happens, the chip will continue to signal an interrupt until it is handled. That will generate an interrupt on the CPU after the taskqueue calls `em_enable_intr()`, in which case it'll get handled. So, it's a lazy, but

Re: em network issues

safe, handling of an edge case.

Scott

freebsd-net@xxxxxxxxxxx mailing list

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

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