Linux kernel interrupt handler mutex protection?

Do I need to protect my interrupt handler being called many times for the same interrupt?

Given the following code, I am not sure on the system calls I should make. I am getting rare, random dead-locks with this current implementation :-

```
void interrupt_handler(void)
{
    down_interruptible(&sem);  // or use a lock here ?

    clear_intr(); // clear interrupt source on H/W

    wake_up_interruptible(...);

    up(&sem); // unlock?

    return IRQ_HANDLED;
}

void set/clear_intr()
{
    spin_lock_irq(&lock);
    RMW(x); // set/clear a bit by read/modify/write the H/W interrupt routing register
    spin_unlock_irq(&lock);
}

void read()
{
    set_intr();  // same as clear_intr, but sets a bit
    wait_event_interruptible(...);
}
```

1. Should `interrupt_handler` : `down_interruptible` be `spin_lock_irq` / `spin_lock_irqsave` / `local_irq_disable` ?
2. Should `set/clear_intr` : `spin_lock_irq` be `spin_lock_irqsave` / `local_irq_disable` ?
3. Can it (H/W -> kernel -> driver handler) keep generating/getting interrupts until its cleared? Can the `interrupt_handler` keep getting called while within it?
4. If as currently implemented the interrupt handler is reentrant then will it block on the `down_interruptible` ?

From LDD3 :-

> must be *reentrant*—it must be capable of running in more than one context at the same time.

Edit 1) after some nice help, suggestions are :-

1. remove `down_interruptible` from within `interrupt_handler`
2. Move `spin_lock_irq` outside set/clear methods (no need for `spin_lock_irqsave` you say?) *I really don't see the benefit to this?!*

Code :-

```
void interrupt_handler(void)
{
    read_reg(y); // eg of other stuff in the handler

    spin_lock_irq(&lock);

    clear_intr(); // clear interrupt source on H/W

    spin_unlock_irq(&lock);

    wake_up_interruptible(...);

    return IRQ_HANDLED;
```

```
}

void set/clear_intr()
{
    RMW(x);
}

void read()
{
    error_checks(); // eg of some other stuff in the read method

    spin_lock_irq(&lock);

    set_intr();  // same as clear_intr, but sets a bit

    spin_unlock_irq(&lock);

    wait_event_interruptible(...);

    // more code here...
}
```

Edit2) After reading some more SO posts : reading Why kernel code/thread executing in interrupt context cannot sleep? which links to Robert Loves article, I read this :

> some interrupt handlers (known in Linux as fast interrupt handlers) run with all interrupts on the local processor disabled. This is done to ensure that the interrupt handler runs without interruption, as quickly as possible. More so, all interrupt handlers run with their current interrupt line disabled on all processors. This ensures that two interrupt handlers for the same interrupt line do not run concurrently. It also prevents device driver writers from having to handle recursive interrupts, which complicate programming.

And I have fast interrupts enabled (SA_INTERRUPT)! So no need for mutex/locks/semaphores/spins/waits/sleeps/etc/etc!

linux linux-kernel linux-device-driver interrupt

edited Jul 5 '11 at 15:21

asked Jul 4 '11 at 10:46

Ian Vaughan
4,074 5 25 49

You answered (LDD3 quote) your own first question (and several other rephrasings of that). In general, prevent locking on time-critical things (like interrupt handling). I'm sorry I can't provide more answer, +1 from me – sehe Jul 4 '11 at 10:59

@sehe yes, but should that call to `down / up` be there? – Ian Vaughan Jul 4 '11 at 11:42

@sehe and with it there, the method cannot be said to be reentrant!? – Ian Vaughan Jul 4 '11 at 11:51

2 @IanVaughan: You should only use `spin_lock_irq` when you know for certain that no interrupts are disabled when your code is called. I'd suggest you change at least the calls in the interrupt handler to use `spin_lock_irqsave` instead. – Hasturkun Jul 4 '11 at 16:13

1 @Ian Vaughan - You are not allowed to use down/up in an interrupt handler or atomic context!!! – Dipstick Jul 5 '11 at 19:41

3 Answers

Don't use semaphores in interrupt context, use `spin_lock_irqsave` instead. quoting LDD3:

> If you have a spinlock that can be taken by code that runs in (hardware or software) interrupt context, you must use one of the forms of spin_lock that disables interrupts. Doing otherwise can deadlock the system, sooner or later. If you do not access your lock in a hardware interrupt handler, but you do via software interrupts (in code that runs out of a tasklet, for example, a topic covered in Chapter 7), you can use spin_lock_bh to safely avoid deadlocks while still allowing hardware interrupts to be serviced.

As for point 2, make your `set_intr` and `clear_intr` require the caller to lock the spinlock, otherwise you'll find your code deadlocking. Again from LDD3:

> To make your locking work properly, you have to write some functions with the assumption that their caller has already acquired the relevant lock(s). Usually, only your internal, static functions can be written in this way; functions called from outside must handle locking explicitly. When you write internal functions that make assumptions about locking, do yourself (and anybody else who works with your code) a favor and document those assumptions explicitly. It can be very hard to come back months later and figure out whether you need to hold a lock to call a particular function or not.

edited May 23 '14 at 12:33
m-ric
1,486 1 14 30

answered Jul 4 '11 at 11:30
Hasturkun
19.9k 1 38 61

Do you mean don't use `down_interruptible` within `interrupt_handler` ? – Ian Vaughan Jul 4 '11 at 11:33

1 @IanVaughan: Yes, that's asking for trouble. I've expanded my answer. – Hasturkun Jul 4 '11 at 11:36

And do you mean either replace `down_interruptible` with `spin_lock_irqsave` (which would definitely make it non-reentrant)? Or ensure I use `spin_lock_irqsave` within my set/clear (instead of the current `spin_lock_irq`) – Ian Vaughan Jul 4 '11 at 11:37

re edit: I dont see how moving the lock/unlock outside the method helps, either the caller does it or its done for me, either way its the same? Unless I am locking bigger/more code. – Ian Vaughan Jul 4 '11 at 11:43

@IanVaughan: I mostly suggested moving the lock/unlock outside the method to avoid a deadlock, assuming your interrupt handles does more under the lock than just calling `clear_intr` . Anyway, you should only need to use `spin_lock_irqsave` for resources shared with the interrupt handler. using `spin_lock_irqsave` does not make your code non-reentrant (quite the opposite really, since it's usually used to ensure that state does not become inconsistent), you should keep your locked sections as small as possible though.. – Hasturkun Jul 4 '11 at 11:52

Use spinlock in interrupt context because you don't want to sleep in interrupt context if you didn't acquired a lock.

answered Mar 20 '13 at 5:52
a.saurabh
190 1 7

The code you posted does not look like a device driver irq handler. The irq handlers in kernel drivers return irqreturn_t and take in int irq_no, void * data as arguements.

You have also not specified if you are registering a threaded handler or a non threaded handler. A non threaded irq handler cannot have any sleeping calls whether or not you hold any spinlocks. wait_event, mutex, semaphore, etc, are all sleeping calls and must not be used in a non threaded irq handler. You can however, hold a spinlock to prevent interruption to your interrupt handler. This will ensure that maskable irqs and scheduler do not interrupt your irq handler in the middle.

In a threaded irq handler, such things as sleeping calls (wait queues, mutex, etc) can be used but are still not recommended.

answered Sep 11 '13 at 7:55

community wiki
preetam