Linux device driver handling multiple interrupt sources/vectors

I am writing a device driver to handle interrupts for a PCIe card, which currently works for any interrupt vector raised on the IRQ line.

But it has a few types that can be raised, flagged by the Vector register. So now I need to read the vector information and be a bit cleverer...
So, do I :-

1/ Have separate dev nodes `/dev/int1` , `/dev/int2` , etc for each interrupt type, and just doc that `int1` is for vector type A etc?
1.1/ As each file/char-devices will have its own `minor` number, when opened I'll know which is which. *i think.*
1.2/ ldd3 seems to demo this method.

2/ Have one node `/dev/int` *(as I do now)* and have multiple processes hanging off the same `read` method? *sounds better?!*
2.1/ Then only wake the correct process up...?
2.2/ Do I use separate `wait_queue_head_t wait_queue` s? Or different `flag` /test conditions?

In the `read` method:-

```
wait_event_interruptible(wait_queue, flag);
```

In the handler *not real code!* :-

```
int vector = read_vector();
if vector = A then
    wake_up_interruptible(wait_queue, flag)
    return IRQ_HANDLED;
else
    return IRQ_NONE/IRQ_RETVAL?
```

EDIT: notes from peoples comments :-
1) my user-space code `mmap` 's all of the PCIe firmware registers
2) User-space code has a few threads, each perform a blocking `read` on the device driver device nodes, which then returns data from the firmware when an interrupt occurs. I need the correct thread woken up depending on the interrupt type.

linux device-driver linux-device-driver interrupt

edited Mar 22 '11 at 11:15

asked Mar 21 '11 at 15:18
Ian Vaughan
4,074 5 25 49

3 Answers

I am not sure I understand correctly what you mean with the Vector register (a pointer to some documentation would help me precise for your case).

Anyway, any PCI device gets a unique interrupt number (given by the BIOS or some firmware on other architectures than x86). You just need to register this interrupt in your driver.

```
priv->name = DRV_NAME;
err = request_irq(pdev->irq, your_irqhandler, IRQF_SHARED, priv->name,
            pdev);
if (err) {
    dev_err(&pdev->dev, "cannot request IRQ\n");
    goto err_out_unmap;
}
```

One other thing that I do not really understand is why you would export your interrupts as a dev node: interrupts are certainly something that need to remain in your driver/kernel code. But I guess here you want to export a device that is then accessed in userspace. I just find

/dev/int no to be a good naming.

For your question about multiple dev nodes: if your different interrupt sources then provide access to different hardware resources (even if on the same PCI board) I would go for option 1), with a wait_queue for each device. Otherwise, I would go for option 2)

Since your interrupts are coming from the same physical device, if you chose option 1) or option 2), the interrupt line will have to be shared and you will have to read the *vector* in your interrupt handler to define which *hardware resource* raised the interrupt.

For option 1), it would be something like this:

```
static irqreturn_t pex_irqhandler(int irq, void *dev) {
    struct pci_dev *pdev = dev;
    int result;

    result = pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &myirq);

    if (result) {
         int vector = read_vector();
         if (vector == A) {
                set_flagA(flag);
         } else if (vector == B) {
                set_flagB(flag);
         }
         wake_up_interruptible(wait_queue, flag);
         return IRQ_HANDLED;
    } else {
         return IRQ_NONE;
}
```

For option 2, it would be similar, but you would have only one if clause (for the respective vector value) in every different interrupt handler that you would request for every node.

edited Mar 22 '11 at 12:39 answered Mar 22 '11 at 8:10
Longfield
767 5 16

sorry docs are proprietary/custom firmware. but the memory map contains a source vector so I know which type has been raised. Thanks, I have the `request_irq` code bit working, my driver currently hangs off this single IRQ, but the PCIe card raises different interrupts, which as we all know, all raise the same IRQ line. – Ian Vaughan Mar 22 '11 at 10:57

I am not exporting the *interrupts* per-sa, I am exposing a device node for user-space to use, as I might not of been clear, I have updated my Q, see also my other comment reply. And the name `int` is just a pseudo name for here. Yes, your right with having to read the *vector*. *(and without meaning to insult, but your option 1 code looks very much like my pseudo code in my Q!? (but is actual code!))* – Ian Vaughan Mar 22 '11 at 11:06

Don't worry, I don't feel insulted, there are not 10 ways of doing irq handling. What I wanted to mean with this example is that you can have one single irq handler that takes all the possible sources in the vector, or many irq handlers that share this interrupt and all address one irq source in the vector. For version 1) you would have one device with many queues and for version 2) you would have many devices with each one queue. From your edited question, I would say option 2 maybe looks simpler and cleaner. – Longfield Mar 22 '11 at 12:43

If you have different chanel you can `read()` from, then you should definitely use different minor number. Imagine you have a card whith four serial port, you would definitely want four `/dev/ttySx` .
But does your device fit whith this model ?

answered Mar 22 '11 at 7:56
shodanex
7,639 5 28 64

I like the analogy of the serial ports, I might go down this route, ensuring that the card firmware handles concurrency! *can of worms* – Ian Vaughan Mar 22 '11 at 11:13

First, I assume you're not trying to get your code into the mainline kernel. If you are, expect a vigorous discussion about the best way to do this. If you're writing a simple interrupt handling driver for a card which is mostly driven by `mmap` from user-space, there are a lot of ways to solve this problem.

If you use multiple device nodes (option 1), you can also implement `poll` so that a single application can open multiple device nodes and wait for a selection of interrupts. The minor number will be sufficient to tell them apart. If you have a wake queue for each vector, you can wake only the relevant listeners. You'll need to latch the vector after a successful `poll` to be sure that the `read` succeeds.

If you use a single device node (option 2), you'll need to add some extra magic so that the threads can register their interest in particular interrupt vectors. You could do this with an ioctl, or have the threads `write` the interrupt vectors to the device. Each thread should open the device node to get its own file descriptor. You can then associate the list of requested vectors with each open file descriptor. As a bonus, you can let the application `read` the interrupt vector from the device, so it knows which one happened.

You'll need to think about how the interrupt gets cleared. The interrupt handler will need to remove the interrupt, then store the result so it can be passed to user-space. You might find a `kfifo` useful for this rather than a wait queue. If you have a fifo for each open file descriptor, you can distribute the interrupt notifications to each listening application.

edited Mar 22 '11 at 13:36

answered Mar 22 '11 at 7:57
Adrian Cox
3,344 14 39

nope, not submitting anything, just writing a bespoke PCIe driver for in house card, and yes, all of the registers on the PCIe card are `mmap` ed in the userspace application. I like the single node with a `write` , I was thinking I'd have to add an `ioctl` as you state, very nifty. I dont need to pass the result of the interrupt per-se, just release the correct blocking `read` (I do a `copy_to_user` and return some data). *(I have updated my Q with some clarifications)* – Ian Vaughan Mar 22 '11 at 10:50

I dont think I need `poll` as the user-space application has a thread for each node it `open` s and pends/blocks on a `read` with. – Ian Vaughan Mar 22 '11 at 11:17

I've worked on a few devices like that. If you want separate threads to monitor different interrupts, you either need multiple device nodes, or for each thread to open the single device node and acquire its own file descriptor. Do you need to queue up and count interrupts, or should each `read` block until the next one comes along? – Adrian Cox Mar 22 '11 at 12:45

I believe your two options are as per my 1 & 2? ("either multiple devices...or...") Is that a correct assumption? I do not need to queue them up or count (each thread will have 1 blocking `read` , it will be activated by my driver and then loop round and block again.) – Ian Vaughan Mar 22 '11 at 13:24

I've edited my answer to refer back to your points 1&2, and mentioned threading. I've still left in a few points that aren't really relevant given your comments. – Adrian Cox Mar 22 '11 at 13:36