CHAPTER 10

Interrupt Handling

Although some devices can be controlled using nothing but their I/O regions, most real devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different from, and far slower than, that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened.

That way, of course, is interrupts. An *interrupt* is simply a signal that the hardware can send when it wants the processor's attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device's interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.

It is difficult to demonstrate the use of interrupts without a real hardware device to generate them. Thus, the sample code used in this chapter works with the parallel port. Such ports are starting to become scarce on modern hardware, but, with luck, most people are still able to get their hands on a system with an available port. We'll be working with the *short* module from the previous chapter; with some small additions it can generate and handle interrupts from the parallel port. The module's name, *short*, actually means *short int* (it is C, isn't it?), to remind us that it handles *int*errupts.

Before we get into the topic, however, it is time for one cautionary note. Interrupt handlers, by their nature, run concurrently with other code. Thus, they inevitably raise issues of concurrency and contention for data structures and hardware. If you succumbed to the temptation to pass over the discussion in Chapter 5, we understand. But we also recommend that you turn back and have another look now. A solid understanding of concurrency control techniques is vital when working with interrupts.

Preparing the Parallel Port

Although the parallel interface is simple, it can trigger interrupts. This capability is used by the printer to notify the *lp* driver that it is ready to accept the next character in the buffer.

Like most devices, the parallel port doesn't actually generate interrupts before it's instructed to do so; the parallel standard states that setting bit 4 of port 2 (`0x37a`, `0x27a`, or whatever) enables interrupt reporting. A simple *outb* call to set the bit is performed by *short* at module initialization.

Once interrupts are enabled, the parallel interface generates an interrupt whenever the electrical signal at pin 10 (the so-called ACK bit) changes from low to high. The simplest way to force the interface to generate interrupts (short of hooking up a printer to the port) is to connect pins 9 and 10 of the parallel connector. A short length of wire inserted into the appropriate holes in the parallel port connector on the back of your system creates this connection. The pinout of the parallel port is shown in Figure 9-1.

Pin 9 is the most significant bit of the parallel data byte. If you write binary data to */dev/short0*, you generate several interrupts. Writing ASCII text to the port won't generate any interrupts, though, because the ASCII character set has no entries with the top bit set.

If you'd rather avoid wiring pins together, but you do have a printer at hand, you can run the sample interrupt handler using a real printer, as shown later. However, note that the probing functions we introduce depend on the jumper between pin 9 and 10 being in place, and you need it to experiment with probing using our code.

Installing an Interrupt Handler

If you want to actually "see" interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect your interrupt, it simply acknowledges and ignores it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in *<linux/interrupt.h>*, implement the interrupt registration interface:

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
```

```
                const char *dev_name,
                void *dev_id);

    void free_irq(unsigned int irq, void *dev_id);
```

The value returned from *request_irq* to the requesting function is either `0` to indicate success or a negative error code, as usual. It's not uncommon for the function to return `-EBUSY` to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

`unsigned int irq`
: The interrupt number being requested.

`irqreturn_t (*handler)(int, void *, struct pt_regs *)`
: The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

`unsigned long flags`
: As you might expect, a bit mask of options (described later) related to interrupt management.

`const char *dev_name`
: The string passed to *request_irq* is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

`void *dev_id`
: Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, `dev_id` can be set to `NULL`, but it a good idea anyway to use this item to point to the device structure. We'll see a practical use for `dev_id` in the section "Implementing a Handler."

The bits that can be set in `flags` are as follows:

`SA_INTERRUPT`
: When set, this indicates a "fast" interrupt handler. Fast handlers are executed with interrupts disabled on the current processor (the topic is covered in the section "Fast and Slow Handlers").

`SA_SHIRQ`
: This bit signals that the interrupt can be shared between devices. The concept of sharing is outlined in the section "Interrupt Sharing."

`SA_SAMPLE_RANDOM`
: This bit indicates that the generated interrupts can contribute to the entropy pool used by */dev/random* and */dev/urandom*. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption. Such random numbers are extracted from an entropy pool that is contributed by various random events. If your device generates interrupts at truly random times, you should set this flag. If, on the other hand, your interrupts are

predictable (for example, vertical blanking of a frame grabber), the flag is not worth setting—it wouldn't contribute to system entropy anyway. Devices that could be influenced by attackers should not set this flag; for example, network drivers can be subjected to predictable packet timing from outside and should not contribute to the entropy pool. See the comments in *drivers/char/random.c* for more information.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea, it often isn't, especially if your device does not share interrupts. Because the number of interrupt lines is limited, you don't want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used. Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

It is possible, for example, to run a frame grabber on the same interrupt as a modem, as long as you don't use the two devices at the same time. It is quite common for users to load the module for a special device at system boot, even if the device is rarely used. A data acquisition gadget might use the same interrupt as the second serial port. While it's not too hard to avoid connecting to your Internet service provider (ISP) during data acquisition, being forced to unload a module in order to use the modem is really unpleasant.

The correct place to call *request_irq* is when the device is first opened, *before* the hardware is instructed to generate interrupts. The place to call *free_irq* is the last time the device is closed, *after* the hardware is told not to interrupt the processor any more. The disadvantage of this technique is that you need to keep a per-device open count so that you know when interrupts can be disabled.

This discussion notwithstanding, *short* requests its interrupt line at load time. This was done so that you can run the test programs without having to run an extra process to keep the device open. *short*, therefore, requests the interrupt from within its initialization function (*short_init*) instead of doing it in *short_open*, as a real device driver would.

The interrupt requested by the following code is `short_irq`. The actual assignment of the variable (i.e., determining which IRQ to use) is shown later, since it is not relevant to the current discussion. `short_base` is the base I/O address of the parallel interface being used; register 2 of the interface is written to enable interrupt reporting.

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
            SA_INTERRUPT, "short", NULL);
   if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
                short_irq);
```

```
        short_irq = -1;
    }
    else { /* actually enable it -- assume this *is* a parallel port */
        outb(0x10,short_base+2);
    }
}
```

The code shows that the handler being installed is a fast handler (`SA_INTERRUPT`), doesn't support interrupt sharing (`SA_SHIRQ` is missing), and doesn't contribute to system entropy (`SA_SAMPLE_RANDOM` is missing, too). The *outb* call then enables interrupt reporting for the parallel port.

For what it's worth, the i386 and x86_64 architectures define a function for querying the availability of an interrupt line:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

This function returns a nonzero value if an attempt to allocate the given interrupt succeeds. Note, however, that things can always change between calls to *can_request_irq* and *request_irq*.

The /proc Interface

Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected. Reported interrupts are shown in */proc/interrupts*. The following snapshot was taken on a two-processor Pentium system:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
           CPU0       CPU1
  0:    4848108         34    IO-APIC-edge  timer
  2:          0          0          XT-PIC  cascade
  8:          3          1    IO-APIC-edge  rtc
 10:       4335          1   IO-APIC-level  aic7xxx
 11:       8903          0   IO-APIC-level  uhci_hcd
 12:         49          1    IO-APIC-edge  i8042
NMI:          0          0
LOC:    4848187    4848186
ERR:          0
MIS:          0
```

The first column is the IRQ number. You can see from the IRQs that are missing that the file shows only interrupts corresponding to installed handlers. For example, the first serial port (which uses interrupt number 4) is not shown, indicating that the modem isn't being used. In fact, even if the modem had been used earlier but wasn't in use at the time of the snapshot, it would not show up in the file; the serial ports are well behaved and release their interrupt handlers when the device is closed.

The */proc/interrupts* display shows how many interrupts have been delivered to each CPU on the system. As you can see from the output, the Linux kernel generally handles

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.