Robert Love

Third Edition

Linux Kernel Development

A thorough guide to the design and implementation of the Linux kernel

Developer's Library

Linux Kernel Development

Third Edition

they are ordinary functions. What differentiates interrupt handlers from other kernel functions is that the kernel invokes them in response to interrupts and that they run in a special context (discussed later in this chapter) called *interrupt context*. This special context is occasionally called *atomic context* because, as we shall see, code executing in this context is unable to block. In this book, we will use the term interrupt context.

Because an interrupt can occur at any time, an interrupt handler can, in turn, be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. Therefore, while it is important to the hardware that the operating system services the interrupt without delay, it is also important to the rest of the system that the interrupt handler executes in as short a period as possible.

At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware: *Hey, hardware, I hear ya; now get back to work!* Often, however, interrupt handlers have a large amount of work to perform. For example, consider the interrupt handler for a network device. On top of responding to the hardware, the interrupt handler needs to copy networking packets from the hardware into memory, process them, and push the packets down to the appropriate protocol stack or application. Obviously, this can be a lot of work, especially with today's gigabit and 10-gigabit Ethernet cards.

Top Halves Versus Bottom Halves

These two goals—that an interrupt handler execute quickly *and* perform a large amount of work—clearly conflict with one another. Because of these competing goals, the processing of interrupts is split into two parts, or halves. The interrupt handler is the *top half*. The top half is run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware. Work that can be performed later is deferred until the *bottom half*. The bottom half runs in the future, at a more convenient time, with all interrupts enabled. Linux provides various mechanisms for implementing bottom halves, and they are all discussed in Chapter 8, "Bottom Halves and Deferring Work."

Let's look at an example of the top-half/bottom-half dichotomy, using our old friend, the network card. When network cards receive packets from the network, they need to alert the kernel of their availability. They want and need to do this immediately, to optimize network throughput and latency and avoid timeouts. Thus, they immediately issue an interrupt: *Hey, kernel, I have some fresh packets here!* The kernel responds by executing the network card's registered interrupt.

The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are the important, time-critical, and hardware-specific work. The kernel generally needs to quickly copy the networking packet into main memory because the network data buffer on the networking card is fixed and miniscule in size, particularly compared to main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped. After the networking data is safely in the main memory, the interrupt's job is done, and it can

return control of the system to whatever code was interrupted when the interrupt was generated. The rest of the processing and handling of the packets occurs later, in the bottom half. In this chapter, we look at the top half; in the next chapter, we study the bottom.

Registering an Interrupt Handler

Interrupt handlers are the responsibility of the driver managing the hardware. Each device has one associated driver and, if that device uses interrupts (and most do), then that driver must register one interrupt handler.

Drivers can register an interrupt handler and enable a given interrupt line for handling with the function `request_irq()`, which is declared in `<linux/interrupt.h>`:

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

The first parameter, `irq`, specifies the interrupt number to allocate. For some devices, for example legacy PC devices such as the system timer or keyboard, this value is typically hard-coded. For most other devices, it is probed or otherwise determined programmatically and dynamically.

The second parameter, `handler`, is a function pointer to the actual interrupt handler that services this interrupt. This function is invoked whenever the operating system receives the interrupt.

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

Note the specific prototype of the handler function: It takes two parameters and has a return value of `irqreturn_t`. This function is discussed later in this chapter.

Interrupt Handler Flags

The third parameter, `flags`, can be either zero or a bit mask of one or more of the flags defined in `<linux/interrupt.h>`. Among these flags, the most important are

- `IRQF_DISABLED`—When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled. Most interrupt handlers do not set this flag, as disabling all interrupts is bad form. Its use is reserved for performance-sensitive interrupts that execute quickly. This flag is the current manifestation of the `SA_INTERRUPT` flag, which in the past distinguished between "fast" and "slow" interrupts.
- `IRQF_SAMPLE_RANDOM`—This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do *not* set

this if your device issues interrupts at a predictable rate (for example, the system timer) or can be influenced by external attackers (for example, a networking device). On the other hand, most other hardware generates interrupts at nondeterministic times and is, therefore, a good source of entropy.

- `IRQF_TIMER`—This flag specifies that this handler processes interrupts for the system timer.
- `IRQF_SHARED`—This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line. More information on shared handlers is provided in a following section.

The fourth parameter, `name`, is an ASCII text representation of the device associated with the interrupt. For example, this value for the keyboard interrupt on a PC is keyboard. These text names are used by `/proc/irq` and `/proc/interrupts` for communication with the user, which is discussed shortly.

The fifth parameter, `dev`, is used for shared interrupt lines. When an interrupt handler is freed (discussed later), `dev` provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know *which* handler to remove on a given interrupt line. You can pass `NULL` here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. (And unless your device is old and crusty and lives on the ISA bus, there is a good chance it must support sharing.) This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure: This pointer is unique and might be useful to have within the handlers.

On success, `request_irq()` returns zero. A nonzero value indicates an error, in which case the specified interrupt handler was not registered. A common error is `-EBUSY`, which denotes that the given interrupt line is already in use (and either the current user or you did not specify `IRQF_SHARED`).

Note that `request_irq()` can sleep and therefore cannot be called from interrupt context or other situations where code cannot block. It is a common mistake to call `request_irq()` when it is unsafe to sleep. This is partly because of *why* `request_irq()` can block: It is indeed unclear. On registration, an entry corresponding to the interrupt is created in `/proc/irq`. The function `proc_mkdir()` creates new procfs entries. This function calls `proc_create()` to set up the new procfs entries, which in turn calls `kmalloc()` to allocate memory. As you will see in Chapter 12, "Memory Management," `kmalloc()` can sleep. So there you go!

An Interrupt Example

In a driver, requesting an interrupt line and installing a handler is done via `request_irq()`:

```
if (request_irq(irqn, my_interrupt, IRQF_SHARED, "my_device", my_dev)) {
        printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
        return -EIO;
}
```

In this example, `irqn` is the requested interrupt line; `my_interrupt` is the handler; we specify via flags that the line can be shared; the device is named `my_device`; and we passed `my_dev` for `dev`. On failure, the code prints an error and returns. If the call returns zero, the handler has been successfully installed. From that point forward, the handler is invoked in response to an interrupt. It is important to initialize hardware and register an interrupt handler in the proper order to prevent the interrupt handler from running before the device is fully initialized.

Freeing an Interrupt Handler

When your driver unloads, you need to unregister your interrupt handler and potentially disable the interrupt line. To do this, call

```
void free_irq(unsigned int irq, void *dev)
```

If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via `dev` is removed, but the interrupt line is disabled only when the last handler is removed. Now you can see why a unique `dev` is important. With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and enable `free_irq()` to remove only the correct handler. In either case (shared or unshared), if `dev` is non-`NULL`, it must match the desired handler. A call to `free_irq()` must be made from process context.

Table 7.1 reviews the functions for registering and deregistering an interrupt handler.

Table 7.1 **Interrupt Registration Methods**

Function	Description
`request_irq()`	Register a given interrupt handler on a given interrupt line.
`free_irq()`	Unregister a given interrupt handler; if no handlers remain on the line, the given interrupt line is disabled.

Writing an Interrupt Handler

The following is a declaration of an interrupt handler:

```
static irqreturn_t intr_handler(int irq, void *dev)
```

Note that this declaration matches the prototype of the `handler` argument given to `request_irq()`. The first parameter, `irq`, is the numeric value of the interrupt line the handler is servicing. This value is passed into the handler, but it is not used very often, except in printing log messages. Before version 2.0 of the Linux kernel, there was not a `dev` parameter and thus `irq` was used to differentiate between multiple devices using the

same driver and therefore the same interrupt handler. As an example of this, consider a computer with multiple hard drive controllers of the same type.

The second parameter, `dev`, is a generic pointer to the same `dev` that was given to `request_irq()` when the interrupt handler was registered. If this value is unique (which is required to support sharing), it can act as a cookie to differentiate between multiple devices potentially using the same interrupt handler. `dev` might also point to a structure of use to the interrupt handler. Because the `device` structure is both unique to each device and potentially useful to have within the handler, it is typically passed for `dev`.

The return value of an interrupt handler is the special type `irqreturn_t`. An interrupt handler can return two special values, `IRQ_NONE` or `IRQ_HANDLED`. The former is returned when the interrupt handler detects an interrupt for which its device was not the originator. The latter is returned if the interrupt handler was correctly invoked, and its device did indeed cause the interrupt. Alternatively, `IRQ_RETVAL(val)` may be used. If `val` is nonzero, this macro returns `IRQ_HANDLED`. Otherwise, the macro returns `IRQ_NONE`. These special values are used to let the kernel know whether devices are issuing spurious (that is, unrequested) interrupts. If all the interrupt handlers on a given interrupt line return `IRQ_NONE`, then the kernel can detect the problem. Note the curious return type, `irqreturn_t`, which is simply an `int`. This value provides backward compatibility with earlier kernels, which did not have this feature; before 2.6, interrupt handlers returned `void`. Drivers may simply typedef `irqreturn_t` to `void` and define the different return values to no-ops and then work in 2.4 without further modification. The interrupt handler is normally marked `static` because it is never called directly from another file.

The role of the interrupt handler depends entirely on the device and its reasons for issuing the interrupt. At a minimum, most interrupt handlers need to provide acknowledgment to the device that they received the interrupt. Devices that are more complex need to additionally send and receive data and perform extended work in the interrupt handler. As mentioned, the extended work is pushed as much as possible into the bottom half handler, which is discussed in the next chapter.

> **Reentrancy and Interrupt Handlers**
>
> Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled. Consequently, the same interrupt handler is never invoked concurrently to service a nested interrupt. This greatly simplifies writing your interrupt handler.

Shared Handlers

A shared handler is registered and executed much like a nonshared handler. Following are three main differences:

- The `IRQF_SHARED` flag must be set in the `flags` argument to `request_irq()`.
- The `dev` argument must be unique to each registered handler. A pointer to any per-device structure is sufficient; a common choice is the `device` structure as it is

both unique and potentially useful to the handler. You *cannot* pass NULL for a shared handler!

- The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt. This requires both hardware support and associated logic in the interrupt handler. If the hardware did not offer this capability, there would be no way for the interrupt handler to know whether its associated device or some other device sharing the line caused the interrupt.

All drivers sharing the interrupt line must meet the previous requirements. If any one device does not share fairly, none can share the line. When `request_irq()` is called with `IRQF_SHARED` specified, the call succeeds only if the interrupt line is currently not registered, or if all registered handlers on the line also specified `IRQF_SHARED`. Shared handlers, however, can mix usage of `IRQF_DISABLED`.

When the kernel receives an interrupt, it invokes sequentially each registered handler on the line. Therefore, it is important that the handler be capable of distinguishing whether it generated a given interrupt. The handler must quickly exit if its associated device did not generate the interrupt. This requires the hardware device to have a status register (or similar mechanism) that the handler can check. Most hardware does indeed have such a feature.

A Real-Life Interrupt Handler

Let's look at a real interrupt handler, from the real-time clock (RTC) driver, found in `drivers/char/rtc.c`. An RTC is found in many machines, including PCs. It is a device, separate from the system timer, which sets the system clock, provides an alarm, or supplies a periodic timer. On most architectures, the system clock is set by writing the desired time into a specific register or I/O range. Any alarm or periodic timer functionality is normally implemented via interrupt. The interrupt is equivalent to a real-world clock alarm: The receipt of the interrupt is analogous to a buzzing alarm.

When the RTC driver loads, the function `rtc_init()` is invoked to initialize the driver. One of its duties is to register the interrupt handler:

```
/* register rtc_interrupt on rtc_irq */
if (request_irq(rtc_irq, rtc_interrupt, IRQF_SHARED, "rtc", (void *)&rtc_port)) {
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
        return -EIO;
}
```

In this example, the interrupt line is stored in `rtc_irq`. This variable is set to the RTC interrupt for a given architecture. On a PC, the RTC is located at IRQ 8. The second parameter is the interrupt handler, `rtc_interrupt`, which is willing to share the interrupt line with other handlers, thanks to the `IRQF_SHARED` flag. From the fourth parameter, you can see that the driver name is rtc. Because this device shares the interrupt line, it passes a unique per-device value for `dev`.

Finally, the handler itself: