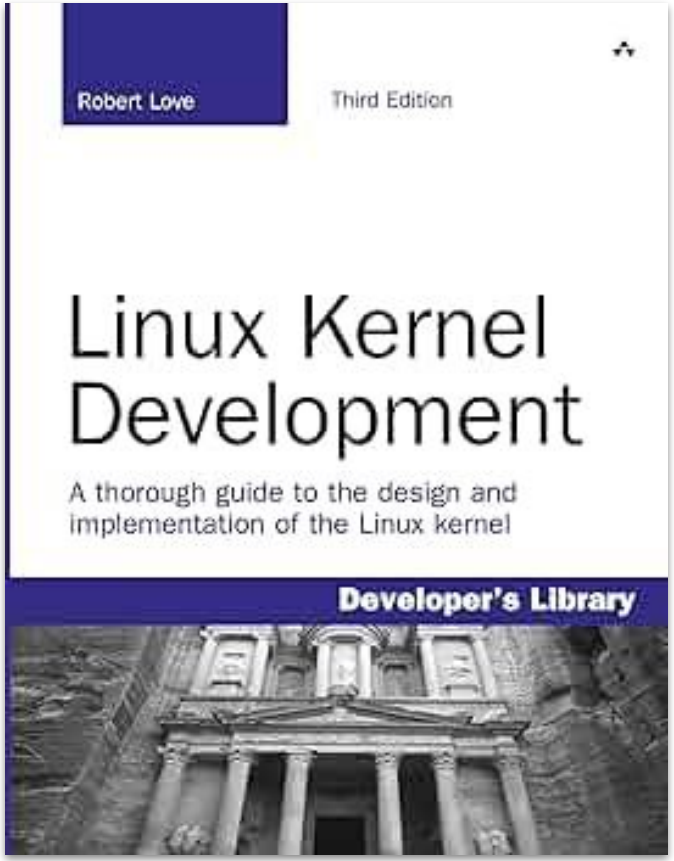
Interrupts, Exceptions, and Signals in Linux

Understanding the differences, resolving misconceptions, and observability.

Zach Croft

Credits

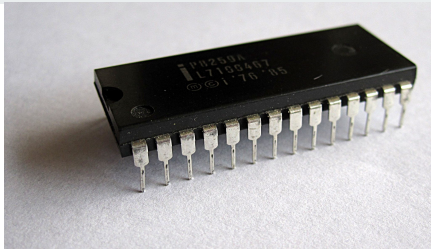

Robert Love
Third Edition
Linux Kernel Development
A thorough guide to the design and implementation of the Linux kernel
Developer's Library

Overview

- Why interrupts?
- Asynchronous vs. synchronous interrupts vs. signals
- Execution mode vs. execution context
- Interrupt handlers, request_irq() and free_irq(), shared and unshared IRQ lines
- Top half vs. bottom half
- Kernel code race condition handling
- Can I remove the default RTC handler with free_irq()?
- Demo: add a shared IRQ line to the timer to print “hello world”
- Puzzle: Nested Python signal handlers
- Conclusion

Not planning on directly discussing:

- Nitty-gritty details of kernel source code (please don’t ask me, I don’t understand it…)!
- Details of use cases like syscalls, page faults, SIGINTs, etc.
- Most detail is on top-halves, not bottom-halves (the book has 30 pages on bottom-halves though)!



Why interrupts?

- A core OS responsibility is to manage hardware. Therefore, the kernel needs to communicate with the hardware. Processors can be orders of magnitude faster than the external hardware, it is not ideal for the kernel to block until receiving a response (e.g. reading from a disk).
- The OS could occasionally poll the hardware to check “are you done yet,” but this is inefficient. Flip the script and let the hardware notify the kernel.
- An interrupt can be physically produced by a signal to input pins on an interrupt controller.
- The controller sends a signal to the processor, which notifies the OS and **interrupt handlers**.

Asynchronous vs. synchronous interrupts, signals

- Each physical device has a unique **interrupt request (IRQ) line**.
  - E.g. system timer is 0, keyboard interrupt is 1, etc.
  - Some are hardcoded, others are dynamically assigned (e.g. plug in a cable to connect your iPhone).
- These interrupts are **asynchronous**, meaning they can happen at any time, agnostic to the system timer (different than the much, much faster CPU clock).
- **Synchronous** interrupts, otherwise known as *exceptions*, are produced by the processor, synchronously with respect to the system timer.
  - E.g. divide by zero, page fault, syscall, etc.
- **Signals** are how the kernel notifies a process that something occurred.
  - E.g. CTRL+C sends an async keyboard interrupt, kernel interprets this, and kernel sends SIGINT to process.
    - If process has no signal handler registered, SIGINT is defaulted to terminate.
- Since asynchronous interrupts may come at random times, the kernel offers a flag IRQF_SAMPLE_RANDOM to use the interrupts to contribute to the entropy pool (random #s).

Execution mode vs. execution context

Execution modes: privilege levels to determine access, managed by a register in the CPU.

- **Kernel space:** unrestricted access to memory and hardware. Exclusive to kernel, with a protected memory space.
- **User space:** subset of machine's available resources, cannot do all system operations, directly access hardware or physical memory. Make system calls (syscalls) to the kernel for access to privileged resources.

Execution context: what triggered the code to run and what resources are available.

- **Process context:** code running on behalf of a specific process/thread. Can sleep, schedule, and access process state. Has an associated task (current).
- **Interrupt context:** code running because hardware triggered an interrupt. **Not tied to any process.** Cannot sleep or schedule. So, must be fast and use atomic operations (more on this later).
    - If slept, what process would the scheduler manage / wake up?

Interrupt handlers, shared vs. unshared IRQ lines

- The function the kernel runs in response to a specific interrupt is called an **interrupt handler**.
  - See below kernel functions (not syscalls) for registering and unregistering interrupt handlers.
    - Declared in <linux/interrupt.h> (*include/linux/interrupt.h* in the Linux repository).
- "IRQF_SHARED—This flag specifies that the interrupt line can be shared among multiple interrupt handlers."
  - Each shared handler must specify the ***dev** argument as a 'cookie' or unique ID for the *handler (not IRQ line)*.
  - If the handler is not shared, ***dev** can be NULL. Otherwise, it usually points to a device struct with metadata, which is guaranteed to be unique per 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.

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

Top half vs. bottom half

- Reminder: interrupt handlers operate in *interrupt context* and therefore cannot sleep or block.
  - Keep interrupt handler code short and sweet to not hog the CPU for too long.
  - But what if an interrupt needs to result in doing some blocking operation?
  - E.g. plugging in a USB results in I/O to read device info, mounting file systems, etc.
- These are conflicting goals–return quickly, but do a lot of work. So, the kernel splits up interrupt response into two phases: a top half and a bottom half.
  - **Top half:** *the interrupt handler;* the critical info work, like acknowledging receipts, resetting hardware, etc.
    - At a minimum, disables this IRQ line on this processor. At a maximum, disables all IRQ lines on this processor.
  - **Bottom half:** deferred, less immediate work done asynchronously.
- Interrupt handlers do not need to be reentrant:
  - Since top halves are supposed to be very quick, the kernel masks out the corresponding interrupt line on all processors, preventing another interrupt on the same line from being received.

Top half vs. bottom half (continued)

- Bottom halves *defer work later*. When is later? **Not now.**
  - Waits for all IRQ lines on this processor to be enabled before running.
- Bottom halves are made of up of:
  - **Softirq:** statically defined bottom halves that need to be compiled into the kernel ahead of time.
    - Can run simultaneously on any processor; even two of the same type can run concurrently.
    - At the time of the book, only networking and block devices use softirqs directly without tasklets.
  - **Tasklet:** dynamically defined bottom halves built on top of softirqs.
    - Think of softirqs as mailboxes and tasklets as mail. Softirqs define priorities and the running mechanism. Tasklets are what gets run.
    - Two of the same type of tasklet cannot run simultaneously. Simpler than softirqs.
  - **Kernel timers:** defer work until a specific time has passed.
  - **Work queue:** The only bottom half that isn't in interrupt context, but process context (can sleep).
    - Easiest to use but slowest. Runs in a kernel thread unlike the other three.

Kernel code race condition handling

- In standard, user-space code, to protect critical data from being accessed concurrently, you could do one of two things:
  - Run it in a single-threaded environment.
  - Run it in a multi-threaded environment and use locks appropriately.
- In kernel-space code, neither of these on their own is enough.
  - Running program ‘foo’ in a single-processor kernel still allows for interrupt handler code (in the kernel) to use the same kernel code as ‘foo’ and potentially access critical data structures.
    - If you lock this section in the program ‘foo,’ then the interrupt handler will block forever!
  - To prevent data races in kernel code, you must disable *local* interrupts temporarily.
    - In a multiprocessor kernel, disable local interrupts AND use locks.
- Interrupt handlers do not need to be reentrant:
  - Since top halves are supposed to be very quick, the kernel masks out the corresponding interrupt line on this local processor, preventing another interrupt on the same line from being received. Other CPUs can use that IRQ line, so still lock if necessary!

```
spin_lock_irqsave(&lock, flags);

/* critical section */

spin_unlock_irqrestore(&lock, flags);
```

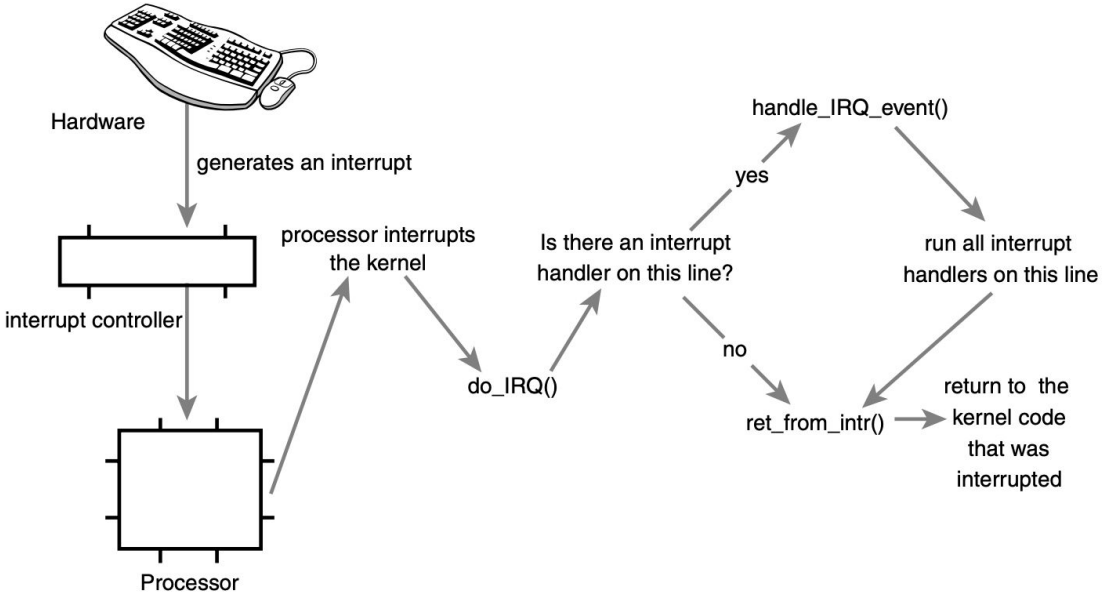


Figure 7.1 The path that an interrupt takes from hardware and on through the kernel.

Can I remove the default RTC handler with free_irq()?

- RTC is the real-time clock, with a driver in <driver/char/rtc.c>, handles setting the system clock, provides an alarm, etc. *This is different from the system timer, which is used for scheduling.*
  - When the driver loads, it registers the interrupt handler.
  - Note: the ***dev** argument is non-null. Since an external source would not know the address of that argument, one cannot free this specific handler.
    - This uses the IRQF_SHARED flag, so you can add handlers. However, if RTC did not allow for shared handlers, it would effectively block others from swapping out one handler for their own.

*dev

```
/* 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;
}
```

Demo: print “hello world” every second

- Fail:
  - Create a kernel module that tries to register a shared handler to IRQ 10 (system timer arch-timer in cat /proc/interrupts).
  - Kernel gives a -22 EINVAL. Kernel doesn’t want you to hook into something like this so it is unshareable–it is critical and additional latency will mess things up!
- Success:
  - Instead, use the kernel-approved approach:
    - Register a bottom-half function.
    - Top-half (hardirq): Timer hardware interrupt fires, acknowledges hardware.
    - Bottom-half (softirq/tasklet): Process expired timers, call callbacks like ours.

Puzzle: Nested Python signal handlers

.. To VSCode

Conclusion

- While we may never write kernel code at Smartleaf, these present interesting optimizations:
  - Interrupts can be asynchronous and make your system less deterministic, so consider disabling some.
- You can look at /proc/interrupts to see the count of interrupts received from devices, to easily tell if a device is communicating with the OS or not.
- Better understanding of what is going on when you send CTRL+C or kill -9.
  - Ctrl+C sends a SIGINT that asks the process to gracefully terminate itself. The process could have a signal handler that prints “I don’t care!” and not terminate itself.
  - kill -9 sends a SIGKILL and the kernel simply marks the process as dead, ungracefully, risking bypassing cleanup that the process may want to do.
- Interrupts are cool.