Understanding the Linux Kernel, 3rd Edition by Marco Cesati, Daniel P. Bovet

Interrupt Handling

As we explained earlier, most exceptions are handled simply by sending a Unix signal to the process that caused the exception. The action to be taken is thus deferred until the process receives the signal; as a result, the kernel is able to process the exception quickly.

This approach does not hold for interrupts, because they frequently arrive long after the process to which they are related (for instance, a process that requested a data transfer) has been suspended and a completely unrelated process is running. So it would make no sense to send a Unix signal to the current process.

Interrupt handling depends on the type of interrupt. For our purposes, we'll distinguish three main classes of interrupts:

I/O interrupts

An I/O device requires attention; the corresponding interrupt handler must query the device to determine the proper course of action. We cover this type of interrupt in the later section "I/O Interrupt Handling."

Timer interrupts

Some timer, either a local APIC timer or an external timer, has issued an interrupt; this kind of interrupt tells the kernel that a fixed-time interval has elapsed. These interrupts are handled mostly as I/O interrupts; we discuss the peculiar characteristics of timer interrupts in Chapter 6.

Interprocessor interrupts

A CPU issued an interrupt to another CPU of a multiprocessor system. We cover such interrupts in the later section "Interprocessor Interrupt Handling."

I/O Interrupt Handling

devices found in older PC architectures (such as ISA) do not reliably operate if their IRQ line is shared with other devices.

Interrupt handler flexibility is achieved in two distinct ways, as discussed in the following list.

IRQ sharing

The interrupt handler executes several *interrupt service routines* (*ISRs*). Each ISR is a function related to a single device sharing the IRQ line. Because it is not possible to know in advance which particular device issued the IRQ, each ISR is executed to verify whether its device needs attention; if so, the ISR performs all the operations that need to be executed when the device raises an interrupt.

IRQ dynamic allocation

An IRQ line is associated with a device driver at the last possible moment; for instance, the IRQ line of the floppy device is allocated only when a user accesses the floppy disk device. In this way, the same IRQ vector may be used by several hardware devices even if they cannot share the IRQ line; of course, the hardware devices cannot be used at the same time. (See the discussion at the end of this section.)

Not all actions to be performed when an interrupt occurs have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long noncritical operations should be deferred, because while an interrupt handler is running, the signals on the corresponding IRQ line are temporarily ignored. Most important, the process on behalf of which an interrupt handler is executed must always stay in the `TASK_RUNNING` state, or a system freeze can occur. Therefore, interrupt handlers cannot perform any blocking procedure such as an I/O disk operation. Linux divides the actions to be performed following an interrupt into three classes:

Critical

Actions such as acknowledging an interrupt to the PIC, reprogramming the PIC or the device controller, or updating data structures accessed by both the device and the processor. These can be executed quickly and are critical, because they must be performed as soon as possible. Critical actions are executed within the interrupt

instance, reading the scan code after a keyboard key has been pushed). These actions can also finish quickly, so they are executed by the interrupt handler immediately, with the interrupts enabled.

Noncritical deferrable

Actions such as copying a buffer's contents into the address space of a process (for instance, sending the keyboard line buffer to the terminal handler process). These may be delayed for a long time interval without affecting the kernel operations; the interested process will just keep waiting for the data. Noncritical deferrable actions are performed by means of separate functions that are discussed in the later section "Softirqs and Tasklets."

Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:

1. Save the IRQ value and the register's contents on the Kernel Mode stack.
2. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
3. Execute the interrupt service routines (ISRs) associated with all the devices that share the IRQ.
4. Terminate by jumping to the `ret_from_intr( )` address.

Several descriptors are needed to represent both the state of the IRQ lines and the functions to be executed when an interrupt occurs. Figure 4-4 represents in a schematic way the hardware circuits and the software functions used to handle an interrupt. These functions are discussed in the following sections.

Interrupt vectors

As illustrated in Table 4-2, physical IRQs may be assigned any vector in the range 32-238. However, Linux uses vector 128 to implement system calls.

The IBM-compatible PC architecture requires that some devices be statically connected to specific

now being used, Linux still supports 8259A-style PICs).

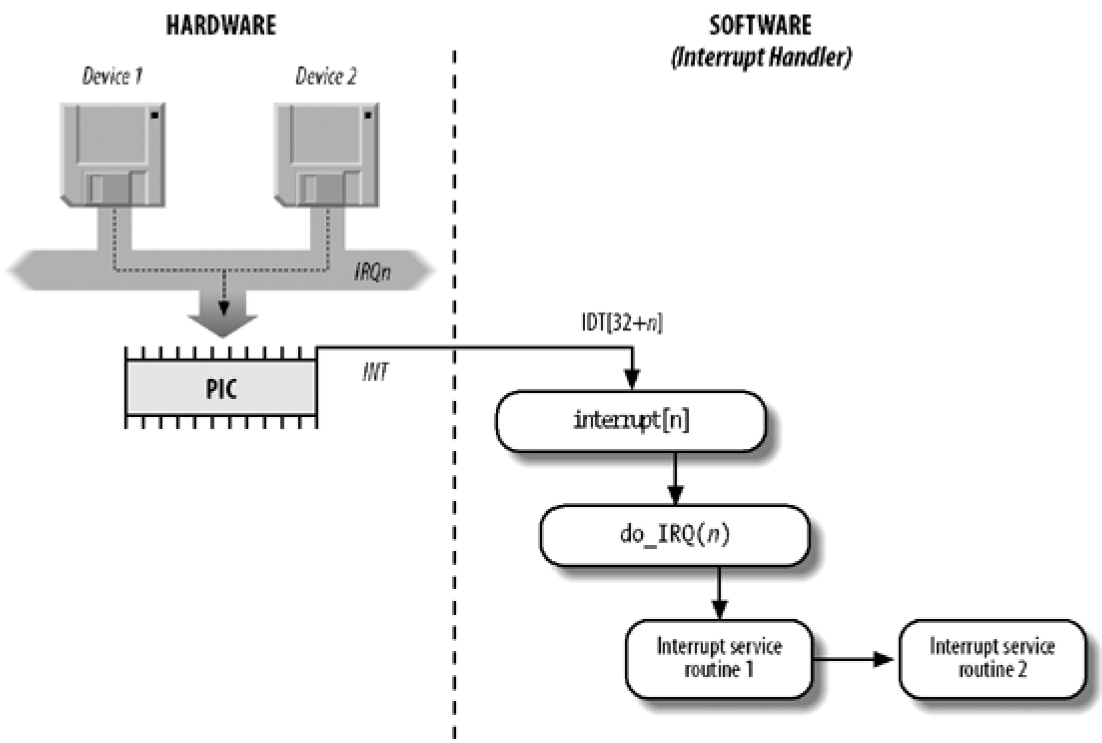

Figure 4-4. I/O interrupt handling

- The external mathematical coprocessor must be connected to the IRQ 13 line (although recent 80 × 86 processors no longer use such a device, Linux continues to support the hardy 80386 model).
- In general, an I/O device can be connected to a limited number of IRQ lines. (As a matter of fact, when playing with an old PC where IRQ sharing is not possible, you might not succeed in installing a new card because of IRQ conflicts with other already present hardware devices.)

Table 4-2. Interrupt vectors in Linux

0–19 (0x0-0x13)	Nonmaskable interrupts and exceptions
20–31 (0x14-0x1f)	Intel-reserved
32–127 (0x20-0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (see Chapter 10)
129–238 (0x81-0xee)	External interrupts (IRQs)
239 (0xef)	Local APIC timer interrupt (see Chapter 6)
240 (0xf0)	Local APIC thermal interrupt (introduced in the Pentium 4 models)
241–250 (0xf1-0xfa)	Reserved by Linux for future use
251–253 (0xfb-0xfd)	Interprocessor interrupts (see the section "Interprocessor Interrupt Handling" later in this chapter)

255 (`0xff`)	Local APIC spurious interrupt (generated if the CPU masks an interrupt while the hardware device raises it)

There are three ways to select a line for an IRQ-configurable device:

- By setting hardware jumpers (only on very old device cards).
- By a utility program shipped with the device and executed when installing it. Such a program may either ask the user to select an available IRQ number or probe the system to determine an available number by itself.
- By a hardware protocol executed at system startup. Peripheral devices declare which interrupt lines they are ready to use; the final values are then negotiated to reduce conflicts as much as possible. Once this is done, each interrupt handler can read the assigned IRQ by using a function that accesses some I/O ports of the device. For instance, drivers for devices that comply with the Peripheral Component Interconnect (PCI) standard use a group of functions such as `pci_read_config_byte( )` to access the device configuration space.

Table 4-3 shows a fairly arbitrary arrangement of devices and IRQs, such as those that might be found on one particular PC.

Table 4-3. An example of IRQ assignment to I/O devices

1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	First serial port
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor

15	47	EIDE disk controller's second chain

The kernel must discover which I/O device corresponds to the IRQ number before enabling interrupts. Otherwise, for example, how could the kernel handle a signal from a SCSI disk without knowing which vector corresponds to the device? The correspondence is established while initializing each device driver (see Chapter 13).

IRQ data structures

As always, when discussing complicated operations involving state transitions, it helps to understand first where key data is stored. Thus, this section explains the data structures that support interrupt handling and how they are laid out in various descriptors. Figure 4-5 illustrates schematically the relationships between the main descriptors that represent the state of the IRQ lines. (The figure does not illustrate the data structures needed to handle softirqs and tasklets; they are discussed later in this chapter.)

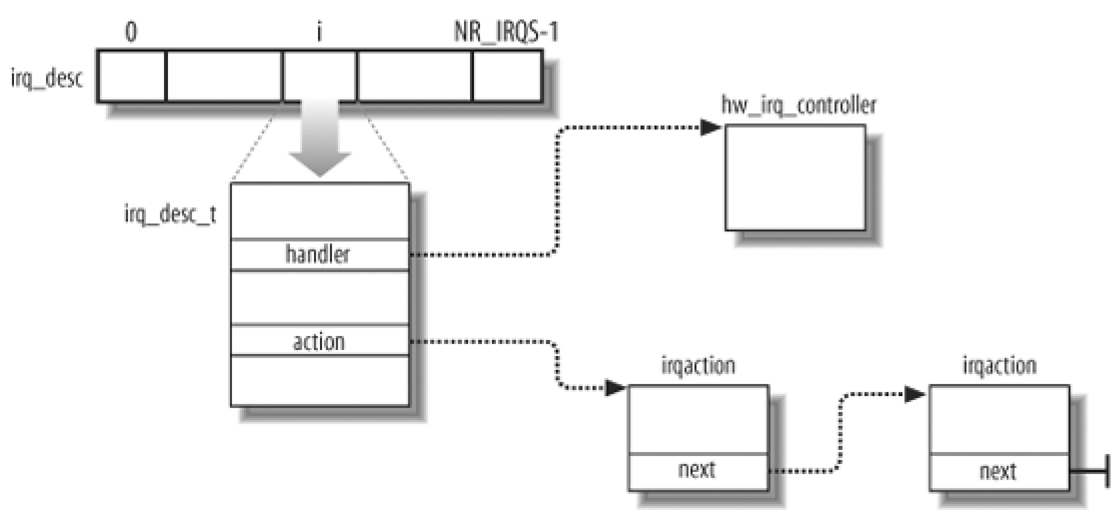

Figure 4-5. IRQ descriptors

Every interrupt vector has its own `irq_desc_t` descriptor, whose fields are listed in Table 4-4. All such descriptors are grouped together in the `irq_desc` array.

Table 4-4. The irq_desc_t descriptor

`handler`	Points to the PIC object (`hw_irq_controller` descriptor) that services the IRQ line.
`handler_data`	Pointer to data used by the PIC methods.
`action`	Identifies the interrupt service routines to be invoked when the IRQ occurs. The field points to the first element of the list of `irqaction` descriptors associated with the IRQ. The `irqaction` descriptor is described later in the chapter.
`status`	A set of flags describing the IRQ line status (see Table 4-5).
`depth`	Shows 0 if the IRQ line is enabled and a positive value if it has been disabled at least once.
`irq_count`	Counter of interrupt occurrences on the IRQ line (for diagnostic use only).
`irqs_unhandled`	Counter of unhandled interrupt occurrences on the IRQ line (for diagnostic use only).
`lock`	A spin lock used to serialize the accesses to the IRQ descriptor and to the PIC (see Chapter 5).

An interrupt is *unexpected* if it is not handled by the kernel, that is, either if there is no ISR associated with the IRQ line, or if no ISR associated with the line recognizes the interrupt as raised by its own hardware device. Usually the kernel checks the number of unexpected interrupts received

raised, the kernel disables the line if the number of unhandled interrupts is above 99,900 (that is, if less than 101 interrupts over the last 100,000 received are expected interrupts from hardware devices sharing the line).

The status of an IRQ line is described by the flags listed in Table 4-5.

Table 4-5. Flags describing the IRQ line status

IRQ_INPROGRESS	A handler for the IRQ is being executed.
IRQ_DISABLED	The IRQ line has been deliberately disabled by a device driver.
IRQ_PENDING	An IRQ has occurred on the line; its occurrence has been acknowledged to the PIC, but it has not yet been serviced by the kernel.
IRQ_REPLAY	The IRQ line has been disabled but the previous IRQ occurrence has not yet been acknowledged to the PIC.
IRQ_AUTODETECT	The kernel is using the IRQ line while performing a hardware device probe.
IRQ_WAITING	The kernel is using the IRQ line while performing a hardware device probe; moreover, the corresponding interrupt has not been raised.
IRQ_LEVEL	Not used on the 80 × 86 architecture.
IRQ_MASKED	Not used.
IRQ_PER_CPU	Not used on the 80 × 86 architecture.

The `depth` field and the `IRQ_DISABLED` flag of the `irq_desc_t` descriptor specify whether

function enables the IRQ line and clears its `IRQ_DISABLED` flag.

During system initialization, the `init_IRQ( )` function sets the `status` field of each IRQ main descriptor to `IRQ _DISABLED`. Moreover, `init_IRQ( )` updates the IDT by replacing the interrupt gates set up by `setup_idt( )` (see the section "Preliminary Initialization of the IDT," earlier in this chapter) with new ones. This is accomplished through the following statements:

```
for (i = 0; i < NR_IRQS; i++)
   if (i+32 != 128)
       set_intr_gate(i+32,interrupt[i]);
```

This code looks in the `interrupt` array to find the interrupt handler addresses that it uses to set up the interrupt gates . Each entry *n* of the `interrupt` array stores the address of the interrupt handler for IRQ *n* (see the later section "Saving the registers for the interrupt handler"). Notice that the interrupt gate corresponding to vector 128 is left untouched, because it is used for the system call's programmed exception.

In addition to the 8259A chip that was mentioned near the beginning of this chapter, Linux supports several other PIC circuits such as the SMP IO-APIC, Intel PIIX4's internal 8259 PIC, and SGI's Visual Workstation Cobalt (IO-)APIC. To handle all such devices in a uniform way, Linux uses a *PIC object*, consisting of the PIC name and seven PIC standard methods. The advantage of this object-oriented approach is that drivers need not to be aware of the kind of PIC installed in the system. Each driver-visible interrupt source is transparently wired to the appropriate controller. The data structure that defines a PIC object is called `hw_interrupt_type` (also called `hw_irq_controller`).

For the sake of concreteness, let's assume that our computer is a uniprocessor with two 8259A PICs, which provide 16 standard IRQs. In this case, the `handler` field in each of the 16 `irq_desc_t` descriptors points to the `i8259A_irq_type` variable, which describes the 8259A PIC. This variable is initialized as follows:

```
struct hw_interrupt_type i8259A_irq_type = {
    .typename      = "XT-PIC",
    .startup       = startup_8259A_irq,
    .shutdown      = shutdown_8259A_irq,
    .enable        = enable_8259A_irq,
```

```
        .disable      = disable_8259A_irq,
```


The first field in this structure, "XT-PIC", is the PIC name. Next come the pointers to six different functions used to program the PIC. The first two functions start up and shut down an IRQ line of the chip, respectively. But in the case of the 8259A chip, these functions coincide with the third and fourth functions, which enable and disable the line. The `mask_and_ack_8259A( )` function acknowledges the IRQ received by sending the proper bytes to the 8259A I/O ports. The `end_8259A_irq( )` function is invoked when the interrupt handler for the IRQ line terminates. The last `set_affinity` method is set to `NULL`: it is used in multiprocessor systems to declare the "affinity" of CPUs for specified IRQs — that is, which CPUs are enabled to handle specific IRQs.

As described earlier, multiple devices can share a single IRQ. Therefore, the kernel maintains `irqaction` descriptors (see Figure 4-5 earlier in this chapter), each of which refers to a specific hardware device and a specific interrupt. The fields included in such descriptor are shown in Table 4-6, and the flags are shown in Table 4-7.

Table 4-6. Fields of the irqaction descriptor

`handler`	Points to the interrupt service routine for an I/O device. This is the key field that allows many devices to share the same IRQ.
`flags`	This field includes a few fields that describe the relationships between the IRQ line and the I/O device (see Table 4-7).
`mask`	Not used.
`name`	The name of the I/O device (shown when listing the serviced IRQs by reading the */proc/interrupts* file).
`dev_id`	A private field for the I/O device. Typically, it identifies the I/O device itself (for instance, it could be equal to its major and minor numbers; see the section "Device Files" in Chapter 13), or it points to the device driver's data.
`next`	Points to the next element of a list of `irqaction` descriptors. The elements in the list refer to hardware devices that share the same IRQ.
irq	IRQ line.
dir	Points to the descriptor of the */proc/irq/n* directory associated with the IRQ*n*.

Table 4-7. Flags of the irqaction descriptor

SA_INTERRUPT	The handler must execute with interrupts disabled.
SA_SHIRQ	The device permits its IRQ line to be shared with other devices.
SA_SAMPLE_RANDOM	The device may be considered a source of events that occurs randomly; it can thus be used by the kernel random number generator. (Users can access this feature by taking random numbers from the */dev/random* and */dev/urandom* device files.)

Finally, the `irq_stat` array includes `NR_CPUS` entries, one for every possible CPU in the system. Each entry of type `irq_cpustat_t` includes a few counters and flags used by the kernel to keep track of what each CPU is currently doing (see Table 4-8).

Table 4-8. Fields of the irq_cpustat_t structure

`_ _softirq_pending`	Set of flags denoting the pending softirqs (see the section "Softirqs" later in this chapter)
`idle_timestamp`	Time when the CPU became idle (significant only if the CPU is currently idle)
`_ _nmi_count`	Number of occurrences of NMI interrupts
`apic_timer_irqs`	Number of occurrences of local APIC timer interrupts (see Chapter 6)

IRQ distribution in multiprocessor systems

Linux sticks to the Symmetric Multiprocessing model (SMP); this means, essentially, that the kernel should not have any bias toward one CPU with respect to the others. As a consequence, the kernel tries to distribute the IRQ signals coming from the hardware devices in a round-robin fashion among all the CPUs. Therefore, all the CPUs should spend approximately the same fraction of their execution time servicing I/O interrupts.

In the earlier section "The Advanced Programmable Interrupt Controller (APIC)," we said that the multi-APIC system has sophisticated mechanisms to dynamically distribute the IRQ signals among the CPUs.

During system bootstrap, the booting CPU executes the `setup_IO_APIC_irqs( )` function to initialize the I/O APIC chip. The 24 entries of the Interrupt Redirection Table of the chip are filled, so that all IRQ signals from the I/O hardware devices can be routed to each CPU in the system according to the "lowest priority" scheme (see the earlier section "IRQs and Interrupts"). During system bootstrap, moreover, all CPUs execute the `setup_local_APIC( )` function, which takes care of initializing the local APICs. In particular, the task priority register (TPR) of each chip is initialized to a fixed value, meaning that the CPU is willing to handle every kind of IRQ signal, regardless of its priority. The Linux kernel never modifies this value after its initialization.

All task priority registers contain the same value, thus all CPUs always have the same priority. To

CPUs and delivers the signal to the corresponding local APIC, which in turn interrupts its CPU. No other CPUs are notified of the event.

All this is magically done by the hardware, so it should be of no concern for the kernel after multi-APIC system initialization. Unfortunately, in some cases the hardware fails to distribute the interrupts among the microprocessors in a fair way (for instance, some Pentium 4-based SMP motherboards have this problem). Therefore, Linux 2.6 makes use of a special kernel thread called *kirqd* to correct, if necessary, the automatic assignment of IRQs to CPUs.

The kernel thread exploits a nice feature of multi-APIC systems, called the IRQ affinity of a CPU: by modifying the Interrupt Redirection Table entries of the I/O APIC, it is possible to route an interrupt signal to a specific CPU. This can be done by invoking the `set_ioapic_affinity_irq( )` function, which acts on two parameters: the IRQ vector to be rerouted and a 32-bit mask denoting the CPUs that can receive the IRQ. The IRQ affinity of a given interrupt also can be changed by the system administrator by writing a new CPU bitmap mask into the */proc/irq/n/smp_affinity* file (*n* being the interrupt vector).

The *kirqd* kernel thread periodically executes the `do_irq_balance( )` function, which keeps track of the number of interrupt occurrences received by every CPU in the most recent time interval. If the function discovers that the IRQ load imbalance between the heaviest loaded CPU and the least loaded CPU is significantly high, then it either selects an IRQ to be "moved" from a CPU to another, or rotates all IRQs among all existing CPUs.

Multiple Kernel Mode stacks

As mentioned in the section "Identifying a Process" in Chapter 3, the `thread_info` descriptor of each process is coupled with a Kernel Mode stack in a `thread_union` data structure composed by one or two page frames, according to an option selected when the kernel has been compiled. If the size of the `thread_union` structure is 8 KB, the Kernel Mode stack of the current process is used for every type of kernel control path: exceptions, interrupts, and deferrable functions (see the later section "Softirqs and Tasklets"). Conversely, if the size of the `thread_union` structure is 4 KB, the kernel makes use of three types of Kernel Mode stacks:

- The *exception stack* is used when handling exceptions (including system calls). This is the stack contained in the per-process `thread_union` data structure, thus the kernel makes use of a different exception stack for each process in the system.

section "Softirqs and Tasklets"). There is one soft IRQ stack for each CPU in the system, and each stack is contained in a single page frame.

All hard IRQ stacks are contained in the `hardirq_stack` array, while all soft IRQ stacks are contained in the `softirq_stack` array. Each array element is a union of type `irq_ctx` that span a single page. At the bottom of this page is stored a `thread_info` structure, while the spare memory locations are used for the stack; remember that each stack grows towards lower addresses. Thus, hard IRQ stacks and soft IRQ stacks are very similar to the exception stacks described in the section "Identifying a Process" in Chapter 3; the only difference is that the `thread_info` structure coupled with each stack is associated with a CPU rather than a process.

The `hardirq_ctx` and `softirq_ctx` arrays allow the kernel to quickly determine the hard IRQ stack and soft IRQ stack of a given CPU, respectively: they contain pointers to the corresponding `irq_ctx` elements.

Saving the registers for the interrupt handler

When a CPU receives an interrupt, it starts executing the code at the address found in the corresponding gate of the IDT (see the earlier section "Hardware Handling of Interrupts and Exceptions").

As with other context switches, the need to save registers leaves the kernel developer with a somewhat messy coding job, because the registers have to be saved and restored using assembly language code. However, within those operations, the processor is expected to call and return from a C function. In this section, we describe the assembly language task of handling registers; in the next, we show some of the acrobatics required in the C function that is subsequently invoked.

Saving registers is the first task of the interrupt handler. As already mentioned, the address of the interrupt handler for IRQ *n* is initially stored in the `interrupt[n]` entry and then copied into the interrupt gate included in the proper IDT entry.

The `interrupt` array is built through a few assembly language instructions in the *arch/i386/kernel/entry.S* file. The array includes `NR_IRQS` elements, where the `NR_IRQS` macro yields either the number 224 if the kernel supports a recent I/O APIC chip,[*] or the number 16 if the kernel uses the older 8259A PIC chips. The element at index *n* in the array stores the address of

The result is to save on the stack the IRQ number associated with the interrupt minus 256. The kernel represents all IRQs through negative numbers, because it reserves positive interrupt numbers to identify system calls (see Chapter 10). The same code for all interrupt handlers can then be executed while referring to this number. The common code starts at label `common_interrupt` and consists of the following assembly language macros and instructions:

```
common_interrupt:
    SAVE_ALL
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
```

The `SAVE_ALL` macro expands to the following fragment:

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $ _ _USER_DS,%edx
movl %edx,%ds
movl %edx,%es
```

`SAVE_ALL` saves all the CPU registers that may be used by the interrupt handler on the stack, except for `eflags` , `cs`, `eip`, `ss`, and `esp`, which are already saved automatically by the control unit (see the earlier section "Hardware Handling of Interrupts and Exceptions"). The macro then loads the selector of the user data segment into `ds` and `es`.

After saving the registers, the address of the current top stack location is saved in the `eax` register; then, the interrupt handler invokes the `do_IRQ( )` function. When the `ret` instruction of `do_IRQ( )` is executed (when that function terminates) control is transferred to

interrupt. It is declared as follows:

```
_ _attribute_ _((regparm(3))) unsigned int do_IRQ(struct pt_regs *regs)
```

The `regparm` keyword instructs the function to go to the `eax` register to find the value of the `regs` argument; as seen above, `eax` points to the stack location containing the last register value pushed on by `SAVE_ALL`.

The `do_IRQ( )` function executes the following actions:

1. Executes the `irq_enter( )` macro, which increases a counter representing the number of nested interrupt handlers. The counter is stored in the `preempt_count` field of the `thread_info` structure of the current process (see Table 4-10 later in this chapter).

2. If the size of the `thread_union` structure is 4 KB, it switches to the hard IRQ stack.In particular, the function performs the following substeps:

    1. Executes the `current_thread_info( )` function to get the address of the `thread_info` descriptor associated with the Kernel Mode stack addressed by the `esp` register (see the section "Identifying a Process" in Chapter 3).

    2. Compares the address of the `thread_info` descriptor obtained in the previous step with the address stored in `hardirq_ctx[smp_processor_id( )]`, that is, the address of the `thread_info` descriptor associated with the local CPU. If the two addresses are equal, the kernel is already using the hard IRQ stack, thus jumps to step 3. This happens when an IRQ is raised while the kernel is still handling another interrupt.

    3. Here the Kernel Mode stack has to be switched. Stores the pointer to the current process descriptor in the `task` field of the `thread_info` descriptor in `irq_ctx` union of the local CPU. This is done so that the `current` macro works as expected while the kernel is using the hard IRQ stack (see the section "Identifying a Process" in Chapter 3).

4. Stores the current value of the `esp` stack pointer register in the `previous_esp`

previous value of the `esp` register is saved in the `ebx` register.

3. Invokes the `_ _do_IRQ( )` function passing to it the pointer `regs` and the IRQ number obtained from the `regs->orig_eax` field (see the following section).

4. If the hard IRQ stack has been effectively switched in step 2e above, the function copies the original stack pointer from the `ebx` register into the `esp` register, thus switching back to the exception stack or soft IRQ stack that were in use before.

5. Executes the `irq_exit( )` macro, which decreases the interrupt counter and checks whether deferrable kernel functions are waiting to be executed (see the section "Softirqs and Tasklets" later in this chapter).

6. Terminates: the control is transferred to the `ret_from_intr( )` function (see the later section "Returning from Interrupts and Exceptions").

The _ _do_IRQ( ) function

The `_ _do_IRQ( )` function receives as its parameters an IRQ number (through the `eax` register) and a pointer to the `pt_regs` structure where the User Mode register values have been saved (through the `edx` register).

The function is equivalent to the following code fragment:

```
spin_lock(&(irq_desc[irq].lock));
irq_desc[irq].handler->ack(irq);
irq_desc[irq].status &= ~(IRQ_REPLAY | IRQ_WAITING);
irq_desc[irq].status |= IRQ_PENDING;
if (!(irq_desc[irq].status & (IRQ_DISABLED | IRQ_INPROGRESS))
        && irq_desc[irq].action) {
    irq_desc[irq].status |= IRQ_INPROGRESS;
    do {
        irq_desc[irq].status &= ~IRQ_PENDING;
        spin_unlock(&(irq_desc[irq].lock));
        handle_IRQ_event(irq, regs, irq_desc[irq].action);
        spin_lock(&(irq_desc[irq].lock));
    } while (irq_desc[irq].status & IRQ_PENDING);
```

```
        irq_desc[irq].status &= ~IRQ_INPROGRESS;
    }
```


in Chapter 5 that the spin lock protects against concurrent accesses by different CPUs. This spin lock is necessary in a multiprocessor system, because other interrupts of the same kind may be raised, and other CPUs might take care of the new interrupt occurrences. Without the spin lock, the main IRQ descriptor would be accessed concurrently by several CPUs. As we'll see, this situation must be absolutely avoided.

After acquiring the spin lock, the function invokes the `ack` method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding `mask_and_ack_8259A( )` function acknowledges the interrupt on the PIC and also disables the IRQ line. Masking the IRQ line ensures that the CPU does not accept further occurrences of this type of interrupt until the handler terminates. Remember that the `_ _do_IRQ( )` function runs with local interrupts disabled; in fact, the CPU control unit automatically clears the `IF` flag of the `eflags` register because the interrupt handler is invoked through an IDT's interrupt gate. However, we'll see shortly that the kernel might re-enable local interrupts before executing the interrupt service routines of this interrupt.

When using the I/O APIC, however, things are much more complicated. Depending on the type of interrupt, acknowledging the interrupt could either be done by the `ack` method or delayed until the interrupt handler terminates (that is, acknowledgement could be done by the `end` method). In either case, we can take for granted that the local APIC doesn't accept further interrupts of this type until the handler terminates, although further occurrences of this type of interrupt may be accepted by other CPUs.

The `_ _do_IRQ( )` function then initializes a few flags of the main IRQ descriptor. It sets the `IRQ_PENDING` flag because the interrupt has been acknowledged (well, sort of), but not yet really serviced; it also clears the `IRQ_WAITING` and `IRQ_REPLAY` flags (but we don't have to care about them now).

Now `_ _do_IRQ( )` checks whether it must really handle the interrupt. There are three cases in which nothing has to be done. These are discussed in the following list.

`IRQ_DISABLED` *is set*

A CPU might execute the `_ _do_IRQ( )` function even if the corresponding IRQ line is disabled; you'll find an explanation for this nonintuitive case in the later section

"Reviving a lost interrupt." Moreover, buggy motherboards may generate spurious

the same interrupt. Why not defer the handling of *this* occurrence to *that* CPU? This is exactly what is done by Linux. This leads to a simpler kernel architecture because device drivers' interrupt service routines need not to be reentrant (their execution is serialized). Moreover, the freed CPU can quickly return to what it was doing, without dirtying its hardware cache; this is beneficial to system performance. The `IRQ_INPROGRESS` flag is set whenever a CPU is committed to execute the interrupt service routines of the interrupt; therefore, the `_ _do_IRQ( )` function checks it before starting the real work.

`irq_desc[irq].action` *is* `NULL`

This case occurs when there is no interrupt service routine associated with the interrupt. Normally, this happens only when the kernel is probing a hardware device.

Let's suppose that none of the three cases holds, so the interrupt has to be serviced. The `_ _do_IRQ( )` function sets the `IRQ_INPROGRESS` flag and starts a loop. In each iteration, the function clears the `IRQ_PENDING` flag, releases the interrupt spin lock, and executes the interrupt service routines by invoking `handle_IRQ_event( )` (described later in the chapter). When the latter function terminates, `_ _do_IRQ( )` acquires the spin lock again and checks the value of the `IRQ_PENDING` flag. If it is clear, no further occurrence of the interrupt has been delivered to another CPU, so the loop ends. Conversely, if `IRQ_PENDING` is set, another CPU has executed the `do_IRQ( )` function for this type of interrupt while this CPU was executing `handle_IRQ_event( )`. Therefore, `do_IRQ( )` performs another iteration of the loop, servicing the new occurrence of the interrupt.[*]

Our `_ _do_IRQ( )` function is now going to terminate, either because it has already executed the interrupt service routines or because it had nothing to do. The function invokes the `end` method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding `end_8259A_irq( )` function reenables the IRQ line (unless the interrupt occurrence was spurious). When using the I/O APIC, the `end` method acknowledges the interrupt (if not already done by the `ack` method).

Finally, `_ _do_IRQ( )` releases the spin lock: the hard work is finished!

Reviving a lost interrupt

The _ _do_IRQ( ) function is small and simple, yet it works properly in most cases. Indeed, the

APIC system selects our CPU for handling the interrupt. Before the CPU acknowledges the interrupt, the IRQ line is masked out by another CPU; as a consequence, the `IRQ_DISABLED` flag is set. Right afterwards, our CPU starts handling the pending interrupt; therefore, the `do_IRQ( )` function acknowledges the interrupt and then returns without executing the interrupt service routines because it finds the `IRQ_DISABLED` flag set. Therefore, even though the interrupt occurred before the IRQ line was disabled, it gets lost.

To cope with this scenario, the `enable_irq( )` function, which is used by the kernel to enable an IRQ line, checks first whether an interrupt has been lost. If so, the function forces the hardware to generate a new occurrence of the lost interrupt:

```
spin_lock_irqsave(&(irq_desc[irq].lock), flags);
if (--irq_desc[irq].depth == 0) {
    irq_desc[irq].status &= ~IRQ_DISABLED;
    if (irq_desc[irq].status & (IRQ_PENDING | IRQ_REPLAY))
           == IRQ_PENDING) {
        irq_desc[irq].status |= IRQ_REPLAY;
        hw_resend_irq(irq_desc[irq].handler,irq);
    }
    irq_desc[irq].handler->enable(irq);
}
spin_lock_irqrestore(&(irq_desc[irq].lock), flags);
```

The function detects that an interrupt was lost by checking the value of the `IRQ_PENDING` flag. The flag is always cleared when leaving the interrupt handler; therefore, if the IRQ line is disabled and the flag is set, then an interrupt occurrence has been acknowledged but not yet serviced. In this case the `hw_resend_irq( )` function raises a new interrupt. This is obtained by forcing the local APIC to generate a self-interrupt (see the later section "Interprocessor Interrupt Handling"). The role of the `IRQ_REPLAY` flag is to ensure that exactly one self-interrupt is generated. Remember that the `_ _do_IRQ( )` function clears that flag when it starts handling the interrupt.

Interrupt service routines

As mentioned previously, an interrupt service routine handles an interrupt by executing an operation specific to one type of device. When an interrupt handler must execute the ISRs, it invokes the `handle_IRQ_event( )` function. This function essentially performs the following steps:

1. Enables the local interrupts with the `sti` assembly language instruction if the


```
retval = 0;
do {
    retval |= action->handler(irq, action->dev_id, regs
    action = action->next;
} while (action);
```

At the start of the loop, `action` points to the start of a list of `irqaction` data structures that indicate the actions to be taken upon receiving the interrupt (see Figure 4-5 earlier in this chapter).

3. Disables local interrupts with the `cli` assembly language instruction.

4. Terminates by returning the value of the `retval` local variable, that is, 0 if no interrupt service routine has recognized interrupt, 1 otherwise (see next).

All interrupt service routines act on the same parameters (once again they are passed through the `eax`, `edx`, and `ecx` registers, respectively):

`irq`

The IRQ number

`dev_id`

The device identifier

`regs`

A pointer to a `pt_regs` structure on the Kernel Mode (exception) stack containing the registers saved right after the interrupt occurred. The `pt_regs` structure consists of 15 fields:

- The first nine fields are the register values pushed by `SAVE_ALL`

- The tenth field, referenced through a field called `orig_eax`, encodes the IRQ

The first parameter allows a single ISR to handle several IRQ lines, the second one allows a single ISR to take care of several devices of the same type, and the last one allows the ISR to access the execution context of the interrupted kernel control path. In practice, most ISRs do not use these parameters.

Every interrupt service routine returns the value 1 if the interrupt has been effectively handled, that is, if the signal was raised by the hardware device handled by the interrupt service routine (and not by another device sharing the same IRQ); it returns the value 0 otherwise. This return code allows the kernel to update the counter of unexpected interrupts mentioned in the section "IRQ data structures" earlier in this chapter.

The `SA_INTERRUPT` flag of the main IRQ descriptor determines whether interrupts must be enabled or disabled when the `do_IRQ( )` function invokes an ISR. An ISR that has been invoked with the interrupts in one state is allowed to put them in the opposite state. In a uniprocessor system, this can be achieved by means of the `cli` (disable interrupts) and `sti` (enable interrupts) assembly language instructions.

The structure of an ISR depends on the characteristics of the device handled. We'll give a couple of examples of ISRs in Chapter 6 and Chapter 13.

Dynamic allocation of IRQ lines

As noted in section "Interrupt vectors," a few vectors are reserved for specific devices, while the remaining ones are dynamically handled. There is, therefore, a way in which the same IRQ line can be used by several hardware devices even if they do not allow IRQ sharing. The trick is to serialize the activation of the hardware devices so that just one owns the IRQ line at a time.

Before activating a device that is going to use an IRQ line, the corresponding driver invokes `request_irq( )`. This function creates a new `irqaction` descriptor and initializes it with the parameter values; it then invokes the `setup_irq( )` function to insert the descriptor in the proper IRQ list. The device driver aborts the operation if `setup_irq( )` returns an error code, which usually means that the IRQ line is already in use by another device that does not allow interrupt sharing. When the device operation is concluded, the driver invokes the `free_irq( )` function to remove the descriptor from the IRQ list and release the memory area.

Let's see how this scheme works on a simple example. Assume a program wants to address the


```
request_irq(6, floppy_interrupt,
                SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL);
```

As can be observed, the `floppy_interrupt( )` interrupt service routine must execute with the interrupts disabled (`SA_INTERRUPT` flag set) and no sharing of the IRQ (`SA_SHIRQ` flag missing). The `SA_SAMPLE_RANDOM` flag set means that accesses to the floppy disk are a good source of random events to be used for the kernel random number generator. When the operation on the floppy disk is concluded (either the I/O operation on */dev/fd0* terminates or the filesystem is unmounted), the driver releases IRQ 6:

```
free_irq(6, NULL);
```

To insert an `irqaction` descriptor in the proper list, the kernel invokes the `setup_irq( )` function, passing to it the parameters `irq _nr`, the IRQ number, and `new` (the address of a previously allocated `irqaction` descriptor). This function:

1. Checks whether another device is already using the `irq _nr` IRQ and, if so, whether the `SA_SHIRQ` flags in the `irqaction` descriptors of both devices specify that the IRQ line can be shared. Returns an error code if the IRQ line cannot be used.

2. Adds `*new` (the new `irqaction` descriptor pointed to by `new`) at the end of the list to which `irq _desc[irq _nr]->action` points.

3. If no other device is sharing the same IRQ, the function clears the `IRQ _DISABLED`, `IRQ_AUTODETECT`, `IRQ_WAITING`, and `IRQ _INPROGRESS` flags in the `flags` field of `*new` and invokes the `startup` method of the `irq_desc[irq_nr]->handler` PIC object to make sure that IRQ signals are enabled.

Here is an example of how `setup_irq( )` is used, drawn from system initialization. The kernel initializes the `irq0` descriptor of the interval timer device by executing the following instructions in the `time_init( )` function (see Chapter 6):

```
struct irqaction irq0  =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
```

```
setup_irq(0, &irq0);
```


the kernel invokes `setup_irq( )` to insert `irq0` in the list of `irqaction` descriptors associated with IRQ 0.

Interprocessor Interrupt Handling

Interprocessor interrupts allow a CPU to send interrupt signals to any other CPU in the system. As explained in the section "The Advanced Programmable Interrupt Controller (APIC)" earlier in this chapter, an interprocessor interrupt (IPI) is delivered not through an IRQ line, but directly as a message on the bus that connects the local APIC of all CPUs (either a dedicated bus in older motherboards, or the system bus in the Pentium 4-based motherboards).

On multiprocessor systems, Linux makes use of three kinds of interprocessor interrupts (see also Table 4-2):

`CALL_FUNCTION_VECTOR` (*vector* `0xfb`)

> Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding interrupt handler is named `call_function_interrupt( )`. The function (whose address is passed in the `call_data` global variable) may, for instance, force all other CPUs to stop, or may force them to set the contents of the Memory Type Range Registers (MTRRs).[*] Usually this interrupt is sent to all CPUs except the CPU executing the calling function by means of the `smp_call_function( )` facility function.

`RESCHEDULE_VECTOR` (*vector* `0xfc`)

> When a CPU receives this type of interrupt, the corresponding handler — named `reschedule_interrupt( )` — limits itself to acknowledging the interrupt. Rescheduling is done automatically when returning from the interrupt (see the section "Returning from Interrupts and Exceptions" later in this chapter).

`INVALIDATE_TLB_VECTOR` (*vector* `0xfd`)

> Sent to all CPUs but the sender, forcing them to invalidate their Translation Lookaside Buffers. The corresponding handler, named `invalidate_interrupt( )`, flushes

some TLB entries of the processor as described in the section "Handling the Hardware

stack, and then invokes a high-level C function having the same name as the low-level handler preceded by `smp_`. For instance, the high-level handler of the `CALL_FUNCTION_VECTOR` interprocessor interrupt that is invoked by the low-level `call_function_interrupt( )` handler is named `smp_call_function_interrupt( )`. Each high-level handler acknowledges the interprocessor interrupt on the local APIC and then performs the specific action triggered by the interrupt.

Thanks to the following group of functions, issuing interprocessor interrupts (IPIs) becomes an easy task:

`send_IPI_all( )`

Sends an IPI to all CPUs (including the sender)

`send_IPI_allbutself( )`

Sends an IPI to all CPUs except the sender

`send_IPI_self( )`

Sends an IPI to the sender CPU

`send_IPI_mask( )`

Sends an IPI to a group of CPUs specified by a bit mask

[*] In contrast to `disable_irq_nosync( )`, `disable_irq(n)` waits until all interrupt handlers for IRQ *n* that are running on other CPUs have completed before returning.

[*] There is an exception, though. Linux usually sets up the local APICs in such a way to honor the *focus processor*, when it exists. A focus process will catch all IRQs of the same type as long as it has received an IRQ of that type, and it has not finished executing the interrupt handler. However, Intel has dropped support for focus processors in the Pentium 4 model.

[*] 256 vectors is an architectural limit for the 80×86 architecture. 32 of them are used or reserved for the CPU, so the usable vector space consists of 224 vectors.

[*] Because `IRQ_PENDING` is a flag and not a counter, only the second occurrence of the interrupt can be recognized. Further occurrences in each iteration of the `do_IRQ( )`'s loop are simply lost.

[*] Floppy disks are "old" devices that do not usually allow IRQ sharing.

[*] Starting with the Pentium Pro model, Intel microprocessors include these additional registers to easily customize cache operations. For instance, Linux may use these registers to disable the hardware cache for the addresses mapping the frame buffer of a PCI/AGP graphic card while maintaining the "write combining" mode of operation:

Understanding the Linux Kernel, 3rd Edition by Marco Cesati, Daniel P. Bovet