Chapter 4. Interrupts and Exceptions

An *interrupt* is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside the CPU chip.

Interrupts are often divided into *synchronous* and *asynchronous* interrupts :

- *Synchronous* interrupts are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.
- *Asynchronous* interrupts are generated by other hardware devices at arbitrary times with respect to the CPU clock signals.

Intel microprocessor manuals designate synchronous and asynchronous interrupts as *exceptions* and *interrupts*, respectively. We'll adopt this classification, although we'll occasionally use the term "interrupt signal" to designate both types together (synchronous as well as asynchronous).

Interrupts are issued by interval timers and I/O devices; for instance, the arrival of a keystroke from a user sets off an interrupt.

Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a Page Fault or a requestvia an assembly language instruction such as `int` or `sysenter` for a kernel service.

We start by describing in the next section the motivation for introducing such signals. We then show how the well-known IRQs (Interrupt ReQuests) issued by I/O devices give rise to interrupts, and we detail how 80 x 86 processors handle interrupts and exceptions at the hardware level. Then we illustrate, in the section "Initializing the Interrupt Descriptor Table," how Linux initializes all the data structures required by the 80x86 interrupt architecture. The remaining three sections describe how Linux handles interrupt signals at the software level.

One word of caution before moving on: in this chapter, we cover only "classic" interrupts common to all PCs; we do not cover the nonstandard interrupts of some architectures.

4.1. The Role of Interrupt Signals

As the name suggests, interrupt signals provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it's currently doing and switch to a new activity; it does this by saving the current value of the

program counter (i.e., the content of the `eip` and `cs` registers) in the Kernel Mode stack and by placing an address related to the interrupt type into the program counter.

There are some things in this chapter that will remind you of the context switch described in the previous chapter, carried out when a kernel substitutes one process for another. But there is a key difference between interrupt handling and process switching: the code executed by an interrupt or by an exception handler is not a process. Rather, it is a kernel control path that runs at the expense of the same process that was running when the interrupt occurred (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a kernel control path, the interrupt handler is lighter than a process (it has less context and requires less time to set up or tear down).

Interrupt handling is one of the most sensitive tasks performed by the kernel, because it must satisfy the following constraints:

- Interrupts can come anytime, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. For instance, suppose a block of data has arrived on a network line. When the hardware interrupts the kernel, it could simply mark the presence of data, give the processor back to whatever was running before, and do the rest of the processing later (such as moving the data into a buffer where its recipient process can find it, and then restarting the process). The activities that the kernel needs to perform in response to an interrupt are thus divided into a critical urgent part that the kernel executes right away and a deferrable part that is left for later.
- Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible, because it keeps the I/O devices busy (see the later section "Nested Execution of Exception and Interrupt Handlers"). As a result, the interrupt handlers must be coded so that the corresponding kernel control paths can be executed in a nested manner. When the last kernel control path terminates, the kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity.
- Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible because, according to the previous requirement, the kernel, and particularly the interrupt handlers, should run most of the time with the interrupts enabled.

4.2. Interrupts and Exceptions

The Intel documentation classifies interrupts and exceptions as follows:

- Interrupts:

  Maskable interrupts

All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.

Nonmaskable interrupts

Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts . Nonmaskable interrupts are always recognized by the CPU.

- Exceptions:

Processor-detected exceptions

Generated when the CPU detects an anomalous condition while executing an instruction. These are further divided into three groups, depending on the value of the `eip` register that is saved on the Kernel Mode stack when the CPU control unit raises the exception.

Faults

Can generally be corrected; once corrected, the program is allowed to restart with no loss of continuity. The saved value of `eip` is the address of the instruction that caused the fault, and hence that instruction can be resumed when the exception handler terminates. As we'll see in the section "Page Fault Exception Handler" in Chapter 9, resuming the same instruction is necessary whenever the handler is able to correct the anomalous condition that caused the exception.

Traps

Reported immediately following the execution of the trapping instruction; after the kernel returns control to the program, it is allowed to continue its execution with no loss of continuity. The saved value of `eip` is the address of the instruction that should be executed after the one that caused the trap. A trap is triggered only when there is no need to reexecute the instruction that terminated. The main use of traps is for debugging purposes. The role of the interrupt signal in this case is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program). Once the user has examined the data provided by the debugger, she may ask that execution of the debugged program resume, starting from the next instruction.

Aborts

A serious error occurred; the control unit is in trouble, and it may be unable to store in the `eip` register the precise location of the instruction causing the exception. Aborts

are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables. The interrupt signal sent by the control unit is an emergency signal used to switch control to the corresponding abort exception handler. This handler has no choice but to force the affected process to terminate.

Programmed exceptions

Occur at the request of the programmer. They are triggered by `int` or `int3` instructions; the `into` (check for overflow) and `bound` (check on address bound) instructions also give rise to a programmed exception when the condition they are checking is not true. Programmed exceptions are handled by the control unit as traps; they are often called *software interrupts* . Such exceptions have two common uses: to implement system calls and to notify a debugger of a specific event (see Chapter 10).

Each interrupt or exception is identified by a number ranging from 0 to 255; Intel calls this 8-bit unsigned number a *vector*. The vectors of nonmaskable interrupts and exceptions are fixed, while those of maskable interrupts can be altered by programming the Interrupt Controller (see the next section).

4.2.1. IRQs and Interrupts

Each hardware device controller capable of issuing interrupt requests usually has a single output line designated as the Interrupt ReQuest (*IRQ*) line.[*] All existing IRQ lines are connected to the input pins of a hardware circuit called the *Programmable Interrupt Controller*, which performs the following actions:

[*] More sophisticated devices use several IRQ lines. For instance, a PCI card can use up to four IRQ lines.

1. Monitors the IRQ lines, checking for raised signals. If two or more IRQ lines are raised, selects the one having the lower pin number.
2. If a raised signal occurs on an IRQ line:
    a. Converts the raised signal received into a corresponding vector.
    b. Stores the vector in an Interrupt Controller I/O port, thus allowing the CPU to read it via the data bus.
    c. Sends a raised signal to the processor INTR pinthat is, issues an interrupt.
    d. Waits until the CPU acknowledges the interrupt signal by writing into one of the *Programmable Interrupt Controllers* (*PIC*) I/O ports; when this occurs, clears the INTR line.
3. Goes back to step 1.

The IRQ lines are sequentially numbered starting from 0; therefore, the first IRQ line is usually denoted as IRQ 0. Intel's default vector associated with IRQ *n* is *n*+32. As mentioned before, the mapping between IRQs and vectors can be modified by issuing suitable I/O instructions to the Interrupt Controller ports.

Each IRQ line can be selectively disabled. Thus, the PIC can be programmed to disable IRQs. That is, the PIC can be told to stop issuing interrupts that refer to a given IRQ line, or

to resume issuing them. Disabled interrupts are not lost; the PIC sends them to the CPU as soon as they are enabled again. This feature is used by most interrupt handlers, because it allows them to process IRQs of the same type serially.

Selective enabling/disabling of IRQs is not the same as global masking/unmasking of maskable interrupts. When the `IF` flag of the `eflags` register is clear, each maskable interrupt issued by the PIC is temporarily ignored by the CPU. The `cli` and `sti` assembly language instructions, respectively, clear and set that flag.

Traditional PICs are implemented by connecting "in cascade" two 8259A-style external chips. Each chip can handle up to eight different IRQ input lines. Because the INT output line of the slave PIC is connected to the IRQ 2 pin of the master PIC, the number of available IRQ lines is limited to 15.

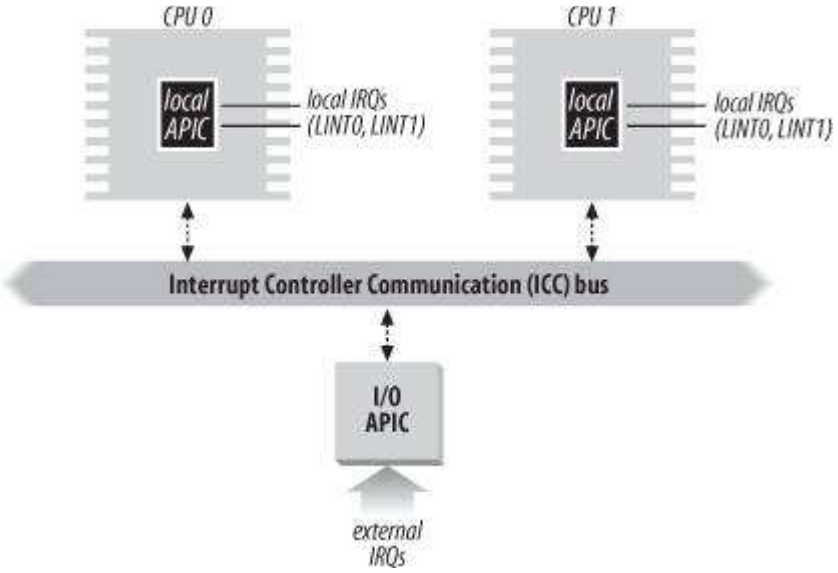
4.2.1.1. The Advanced Programmable Interrupt Controller (APIC)

The previous description refers to PICs designed for uniprocessor systems. If the system includes a single CPU, the output line of the master PIC can be connected in a straightforward way to the INTR pin the CPU. However, if the system includes two or more CPUs, this approach is no longer valid and more sophisticated PICs are needed.

Being able to deliver interrupts to each CPU in the system is crucial for fully exploiting the parallelism of the SMP architecture. For that reason, Intel introduced starting with Pentium III a new component designated as the *I/O Advanced Programmable Interrupt Controller* (*I/O APIC*). This chip is the advanced version of the old 8259A Programmable Interrupt Controller; to support old operating systems, recent motherboards include both types of chip. Moreover, all current 80 x 86 microprocessors include a *local APIC*. Each local APIC has 32-bit registers, an internal clock; a local timer device; and two additional IRQ lines, LINT 0 and LINT 1, reserved for local APIC interrupts. All local APICs are connected to an external I/O APIC, giving rise to a multi-APIC system.

Figure 4-1 illustrates in a schematic way the structure of a multi-APIC system. An *APIC bus* connects the "frontend" I/O APIC to the local APICs. The IRQ lines coming from the devices are connected to the I/O APIC, which therefore acts as a router with respect to the local APICs. In the motherboards of the Pentium III and earlier processors, the APIC bus was a serial three-line bus; starting with the Pentium 4, the APIC bus is implemented by means of the system bus. However, because the APIC bus and its messages are invisible to software, we won't give further details.

Figure 4-1. Multi-APIC system



The I/O APIC consists of a set of 24 IRQ lines, a 24-entry *Interrupt Redirection Table*, programmable registers, and a message unit for sending and receiving APIC messages over the APIC bus. Unlike IRQ pins of the 8259A, interrupt priority is not related to pin number: each entry in the Redirection Table can be individually programmed to indicate the interrupt vector and priority, the destination processor, and how the processor is selected. The information in the Redirection Table is used to translate each external IRQ signal into a message to one or more local APIC units via the APIC bus.

Interrupt requests coming from external hardware devices can be distributed among the available CPUs in two ways:

Static distribution

> The IRQ signal is delivered to the local APICs listed in the corresponding Redirection Table entry. The interrupt is delivered to one specific CPU, to a subset of CPUs, or to all CPUs at once (broadcast mode).

Dynamic distribution

> The IRQ signal is delivered to the local APIC of the processor that is executing the process with the lowest priority.
>
> Every local APIC has a programmable *task priority register* (TPR), which is used to compute the priority of the currently running process. Intel expects this register to be modified in an operating system kernel by each process switch.
>
> If two or more CPUs share the lowest priority, the load is distributed between them using a technique called *arbitration* . Each CPU is assigned a different arbitration

priority ranging from 0 (lowest) to 15 (highest) in the arbitration priority register of the local APIC.

Every time an interrupt is delivered to a CPU, its corresponding arbitration priority is automatically set to 0, while the arbitration priority of any other CPU is increased. When the arbitration priority register becomes greater than 15, it is set to the previous arbitration priority of the winning CPU increased by 1. Therefore, interrupts are distributed in a round-robin fashion among CPUs with the same task priority.[*]

[*] The Pentium 4 local APIC doesn't have an arbitration priority register; the arbitration mechanism is hidden in the bus arbitration circuitry. The Intel manuals state that if the operating system kernel does not regularly update the task priority registers , performance may be suboptimal because interrupts might always be serviced by the same CPU.

Besides distributing interrupts among processors, the multi-APIC system allows CPUs to generate *interprocessor interrupts* . When a CPU wishes to send an interrupt to another CPU, it stores the interrupt vector and the identifier of the target's local APIC in the Interrupt Command Register (ICR) of its own local APIC. A message is then sent via the APIC bus to the target's local APIC, which therefore issues a corresponding interrupt to its own CPU.

Interprocessor interrupts (in short, IPIs) are a crucial component of the SMP architecture. They are actively used by Linux to exchange messages among CPUs (see later in this chapter).

Many of the current uniprocessor systems include an I/O APIC chip, which may be configured in two distinct ways:

- As a standard 8259A-style external PIC connected to the CPU. The local APIC is disabled and the two LINT 0 and LINT 1 local IRQ lines are configured, respectively, as the INTR and NMI pins.
- As a standard external I/O APIC. The local APIC is enabled, and all external interrupts are received through the I/O APIC.

4.2.2. Exceptions

The 80x86 microprocessors issue roughly 20 different exceptions .[*] The kernel must provide a dedicated exception handler for each exception type. For some exceptions, the CPU control unit also generates a *hardware error code* and pushes it on the Kernel Mode stack before starting the exception handler.

[*] The exact number depends on the processor model.

The following list gives the vector, the name, the type, and a brief description of the exceptions found in 80x86 processors. Additional information may be found in the Intel technical documentation.

0 - "*Divide error*" (*fault*)

Raised when a program issues an integer division by 0.

1- "*Debug*" (*trap or fault*)

Raised when the `TF` flag of `eflags` is set (quite useful to implement *single-step execution* of a debugged program) or when the address of an instruction or operand falls within the range of an active debug register (see the section "Hardware Context" in Chapter 3).

2 - Not used

Reserved for nonmaskable interrupts (those that use the NMI pin).

3 - "*Breakpoint*" (*trap*)

Caused by an `int3` (breakpoint) instruction (usually inserted by a debugger).

4 - "*Overflow*" (*trap*)

An `into` (check for overflow) instruction has been executed while the `OF` (overflow) flag of `eflags` is set.

5 - "*Bounds check*" (*fault*)

A `bound` (check on address bound) instruction is executed with the operand outside of the valid address bounds.

6 - "*Invalid opcode*" (*fault*)

The CPU execution unit has detected an invalid opcode (the part of the machine instruction that determines the operation performed).

7 - "*Device not available*" (*fault*)

An ESCAPE, MMX, or SSE/SSE2 instruction has been executed with the `TS` flag of `cr0` set (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3).

8 - "*Double fault*" (*abort*)

Normally, when the CPU detects an exception while trying to call the handler for a prior exception, the two exceptions can be handled serially. In a few cases, however, the processor cannot handle them serially, so it raises this exception.

9 - "*Coprocessor segment overrun*" (*abort*)

Problems with the external mathematical coprocessor (applies only to old 80386 microprocessors).

10 - "*Invalid TSS*" (*fault*)

The CPU has attempted a context switch to a process having an invalid Task State Segment.

11 - "*Segment not present*" (*fault*)

A reference was made to a segment not present in memory (one in which the `Segment-Present` flag of the Segment Descriptor was cleared).

12 - "*Stack segment fault*" (*fault*)

The instruction attempted to exceed the stack segment limit, or the segment identified by `ss` is not present in memory.

13 - "*General protection*" (*fault*)

One of the protection rules in the protected mode of the 80x86 has been violated.

14 - "*Page Fault*" (*fault*)

The addressed page is not present in memory, the corresponding Page Table entry is null, or a violation of the paging protection mechanism has occurred.

15 - Reserved by Intel

16 - "*Floating-point error*" (*fault*)

The floating-point unit integrated into the CPU chip has signaled an error condition, such as numeric overflow or division by 0.[*]

[*] The 80 x 86 microprocessors also generate this exception when performing a signed division whose result cannot be stored as a signed integer (for instance, a division between -2,147,483,648 and -1).

17 - "*Alignment check*" (*fault*)

The address of an operand is not correctly aligned (for instance, the address of a long integer is not a multiple of 4).

18 - "*Machine check*" (abort)

A machine-check mechanism has detected a CPU or bus error.

19 - "*SIMD floating point exception*" (fault)

The SSE or SSE2 unit integrated in the CPU chip has signaled an error condition on a floating-point operation.

The values from 20 to 31 are reserved by Intel for future development. As illustrated in Table 4-1, each exception is handled by a specific exception handler (see the section "Exception Handling" later in this chapter), which usually sends a Unix signal to the process that caused the exception.

Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	`divide_error( )`	`SIGFPE`
1	Debug	`debug( )`	`SIGTRAP`
2	NMI	`nmi( )`	None
3	Breakpoint	`int3( )`	`SIGTRAP`
4	Overflow	`overflow( )`	`SIGSEGV`
5	Bounds check	`bounds( )`	`SIGSEGV`
6	Invalid opcode	`invalid_op( )`	`SIGILL`
7	Device not available	`device_not_available( )`	None
8	Double fault	`doublefault_fn( )`	None
9	Coprocessor segment overrun	`coprocessor_segment_overrun( )`	`SIGFPE`
10	Invalid TSS	`invalid_TSS( )`	`SIGSEGV`
11	Segment not present	`segment_not_present( )`	`SIGBUS`
12	Stack segment fault	`stack_segment( )`	`SIGBUS`
13	General protection	`general_protection( )`	`SIGSEGV`

Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
14	Page Fault	`page_fault( )`	`SIGSEGV`
15	Intel-reserved	None	None
16	Floating-point error	`coprocessor_error( )`	`SIGFPE`
17	Alignment check	`alignment_check( )`	`SIGBUS`
18	Machine check	`machine_check( )`	None
19	SIMD floating point	`simd_coprocessor_error( )`	`SIGFPE`

4.2.3. Interrupt Descriptor Table

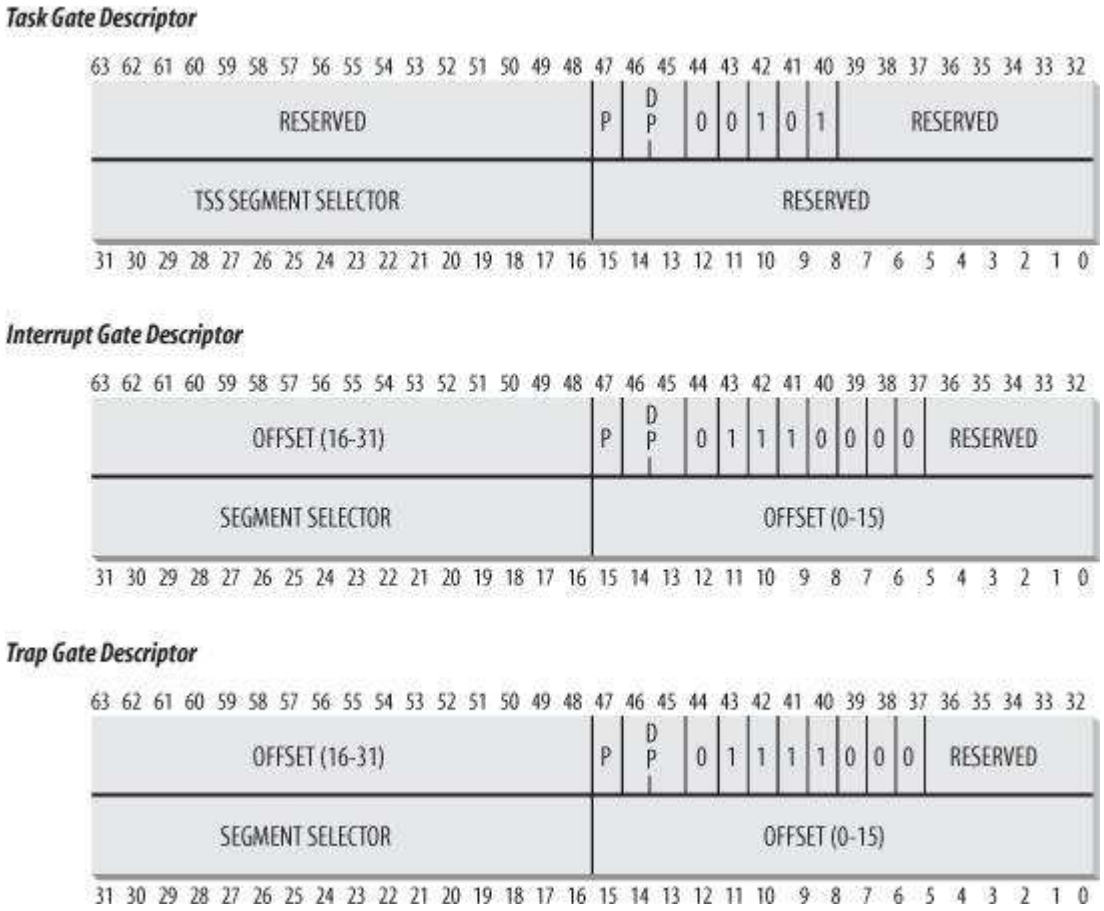
A system table called *Interrupt Descriptor Table* (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts.

The IDT format is similar to that of the GDT and the LDTs examined in Chapter 2. Each entry corresponds to an interrupt or an exception vector and consists of an 8-byte descriptor. Thus, a maximum of 256 x 8 = 2048 bytes are required to store the IDT.

The `idtr` CPU register allows the IDT to be located anywhere in memory: it specifies both the IDT base physical address and its limit (maximum length). It must be initialized before enabling interrupts by using the `lidt` assembly language instruction.

The IDT may include three types of descriptors; Figure 4-2 illustrates the meaning of the 64 bits included in each of them. In particular, the value of the `Type` field encoded in the bits 4043 identifies the descriptor type.

Figure 4-2. Gate descriptors' format



The descriptors are:

Task gate

Includes the TSS selector of the process that must replace the current one when an interrupt signal occurs.

Interrupt gate

Includes the Segment Selector and the offset inside the segment of an interrupt or exception handler. While transferring control to the proper segment, the processor clears the `IF` flag, thus disabling further maskable interrupts.

Trap gate

Similar to an interrupt gate, except that while transferring control to the proper segment, the processor does not modify the `IF` flag.

As we'll see in the later section "Interrupt, Trap, and System Gates," Linux uses interrupt gates to handle interrupts and trap gates to handle exceptions.[*]

[*] The "Double fault " exception, which denotes a type of kernel misbehavior, is the only exception handled by means of a task gate (see the section "Exception Handling" later in this chapter.).

4.2.4. Hardware Handling of Interrupts and Exceptions

We now describe how the CPU control unit handles interrupts and exceptions. We assume that the kernel has been initialized, and thus the CPU is operating in Protected Mode.

After executing an instruction, the `cs` and `eip` pair of registers contain the logical address of the next instruction to be executed. Before dealing with that instruction, the control unit checks whether an interrupt or an exception occurred while the control unit executed the previous instruction. If one occurred, the control unit does the following:

1. Determines the vector *i* ($0 \leq i \leq 255$) associated with the interrupt or the exception.
2. Reads the *i* th entry of the IDT referred by the `idtr` register (we assume in the following description that the entry contains an interrupt or a trap gate).
3. Gets the base address of the GDT from the `gdtr` register and looks in the GDT to read the Segment Descriptor identified by the selector in the IDT entry. This descriptor specifies the base address of the segment that includes the interrupt or exception handler.
4. Makes sure the interrupt was issued by an authorized source. First, it compares the Current Privilege Level (CPL), which is stored in the two least significant bits of the `cs` register, with the Descriptor Privilege Level (DPL) of the Segment Descriptor included in the GDT. Raises a "General protection " exception if the CPL is lower than the DPL, because the interrupt handler cannot have a lower privilege than the program that caused the interrupt. For programmed exceptions, makes a further security check: compares the CPL with the DPL of the gate descriptor included in the IDT and raises a "General protection" exception if the DPL is lower than the CPL. This last check makes it possible to prevent access by user applications to specific trap or interrupt gates.
5. Checks whether a change of privilege level is taking place that is, if CPL is different from the selected Segment Descriptor's DPL. If so, the control unit must start using the stack that is associated with the new privilege level. It does this by performing the following steps:
    a. Reads the `tr` register to access the TSS segment of the running process.
    b. Loads the `ss` and `esp` registers with the proper values for the stack segment and stack pointer associated with the new privilege level. These values are found in the TSS (see the section "Task State Segment" in Chapter 3).
    c. In the new stack, it saves the previous values of `ss` and `esp`, which define the logical address of the stack associated with the old privilege level.
6. If a fault has occurred, it loads `cs` and `eip` with the logical address of the instruction that caused the exception so that it can be executed again.
7. Saves the contents of `eflags` , `cs`, and `eip` in the stack.
8. If the exception carries a hardware error code, it saves it on the stack.

9. Loads `cs` and `eip`, respectively, with the Segment Selector and the Offset fields of the Gate Descriptor stored in the *i* th entry of the IDT. These values define the logical address of the first instruction of the interrupt or exception handler.

The last step performed by the control unit is equivalent to a jump to the interrupt or exception handler. In other words, the instruction processed by the control unit after dealing with the interrupt signal is the first instruction of the selected handler.

After the interrupt or exception is processed, the corresponding handler must relinquish control to the interrupted process by issuing the `iret` instruction, which forces the control unit to:

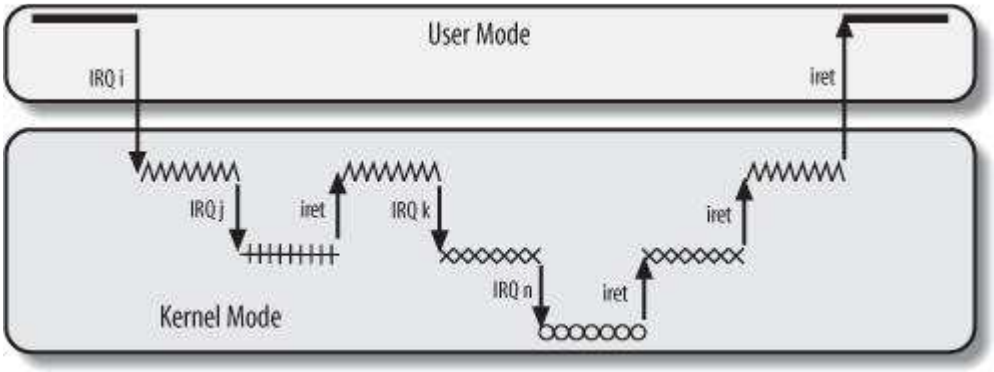
1. Load the `cs`, `eip`, and `eflags` registers with the values saved on the stack. If a hardware error code has been pushed in the stack on top of the `eip` contents, it must be popped before executing `iret`.
2. Check whether the CPL of the handler is equal to the value contained in the two least significant bits of `cs` (this means the interrupted process was running at the same privilege level as the handler). If so, `iret` concludes execution; otherwise, go to the next step.
3. Load the `ss` and `esp` registers from the stack and return to the stack associated with the old privilege level.
4. Examine the contents of the `ds`, `es`, `fs`, and `gs` segment registers; if any of them contains a selector that refers to a Segment Descriptor whose DPL value is lower than CPL, clear the corresponding segment register. The control unit does this to forbid User Mode programs that run with a CPL equal to 3 from using segment registers previously used by kernel routines (with a DPL equal to 0). If these registers were not cleared, malicious User Mode programs could exploit them in order to access the kernel address space.

4.3. Nested Execution of Exception and Interrupt Handlers

Every interrupt or exception gives rise to a *kernel control path* or separate sequence of instructions that execute in Kernel Mode on behalf of the current process. For instance, when an I/O device raises an interrupt, the first instructions of the corresponding kernel control path are those that save the contents of the CPU registers in the Kernel Mode stack, while the last are those that restore the contents of the registers.

Kernel control paths may be arbitrarily nested; an interrupt handler may be interrupted by another interrupt handler, thus giving rise to a nested execution of kernel control paths , as shown in Figure 4-3. As a result, the last instructions of a kernel control path that is taking care of an interrupt do not always put the current process back into User Mode: if the level of nesting is greater than 1, these instructions will put into execution the kernel control path that was interrupted last, and the CPU will continue to run in Kernel Mode.

Figure 4-3. An example of nested execution of kernel control paths



The price to pay for allowing nested kernel control paths is that an interrupt handler must never block, that is, no process switch can take place until an interrupt handler is running. In fact, all the data needed to resume a nested kernel control path is stored in the Kernel Mode stack, which is tightly bound to the current process.

Assuming that the kernel is bug free, most exceptions can occur only while the CPU is in User Mode. Indeed, they are either caused by programming errors or triggered by debuggers. However, the "Page Fault " exception may occur in Kernel Mode. This happens when the process attempts to address a page that belongs to its address space but is not currently in RAM. While handling such an exception, the kernel may suspend the current process and replace it with another one until the requested page is available. The kernel control path that handles the "Page Fault" exception resumes execution as soon as the process gets the processor again.

Because the "Page Fault" exception handler never gives rise to further exceptions, at most two kernel control paths associated with exceptions (the first one caused by a system call invocation, the second one caused by a Page Fault) may be stacked, one on top of the other.

In contrast to exceptions, interrupts issued by I/O devices do not refer to data structures specific to the current process, although the kernel control paths that handle them run on behalf of that process. As a matter of fact, it is impossible to predict which process will be running when a given interrupt occurs.

An interrupt handler may preempt both other interrupt handlers and exception handlers. Conversely, an exception handler never preempts an interrupt handler. The only exception that can be triggered in Kernel Mode is "Page Fault," which we just described. But interrupt handlers never perform operations that can induce page faults, and thus, potentially, a process switch.

Linux interleaves kernel control paths for two major reasons:

- To improve the throughput of programmable interrupt controllers and device controllers. Assume that a device controller issues a signal on an IRQ line: the PIC transforms it into an external interrupt, and then both the PIC and the device

controller remain blocked until the PIC receives an acknowledgment from the CPU. Thanks to kernel control path interleaving, the kernel is able to send the acknowledgment even when it is handling a previous interrupt.

- To implement an interrupt model without priority levels. Because each interrupt handler may be deferred by another one, there is no need to establish predefined priorities among hardware devices. This simplifies the kernel code and improves its portability.

On multiprocessor systems, several kernel control paths may execute concurrently. Moreover, a kernel control path associated with an exception may start executing on a CPU and, due to a process switch, migrate to another CPU.

4.4. Initializing the Interrupt Descriptor Table

Now that we understand what the 80x86 microprocessors do with interrupts and exceptions at the hardware level, we can move on to describe how the Interrupt Descriptor Table is initialized.

Remember that before the kernel enables the interrupts, it must load the initial address of the IDT table into the `idtr` register and initialize all the entries of that table. This activity is done while initializing the system (see Appendix A).

The `int` instruction allows a User Mode process to issue an interrupt signal that has an arbitrary vector ranging from 0 to 255. Therefore, initialization of the IDT must be done carefully, to block illegal interrupts and exceptions simulated by User Mode processes via `int` instructions. This can be achieved by setting the DPL field of the particular Interrupt or Trap Gate Descriptor to 0. If the process attempts to issue one of these interrupt signals, the control unit checks the CPL value against the DPL field and issues a "General protection " exception.

In a few cases, however, a User Mode process must be able to issue a programmed exception. To allow this, it is sufficient to set the DPL field of the corresponding Interrupt or Trap Gate Descriptors to 3 that is, as high as possible.

Let's now see how Linux implements this strategy.

4.4.1. Interrupt, Trap, and System Gates

As mentioned in the earlier section "Interrupt Descriptor Table," Intel provides three types of interrupt descriptors : Task, Interrupt, and Trap Gate Descriptors. Linux uses a slightly different breakdown and terminology from Intel when classifying the interrupt descriptors included in the Interrupt Descriptor Table:

Interrupt gate

An Intel interrupt gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). All Linux interrupt handlers are activated by means of interrupt gates , and all are restricted to Kernel Mode.

System gate

An Intel trap gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The three Linux exception handlers associated with the vectors 4, 5, and 128 are activated by means of system gates , so the three assembly language instructions `into` , `bound` , and `int $0x80` can be issued in User Mode.

System interrupt gate

An Intel interrupt gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The exception handler associated with the vector 3 is activated by means of a system interrupt gate, so the assembly language instruction `int3` can be issued in User Mode.

Trap gate

An Intel trap gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). Most Linux exception handlers are activated by means of trap gates .

Task gate

An Intel task gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). The Linux handler for the "Double fault " exception is activated by means of a task gate.

The following architecture-dependent functions are used to insert gates in the IDT:

`set_intr_gate(n,addr)`

Inserts an interrupt gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the interrupt handler. The DPL field is set to 0.

`set_system_gate(n,addr)`

Inserts a trap gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the exception handler. The DPL field is set to 3.

`set_system_intr_gate(n,addr)`

Inserts an interrupt gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to `addr`, which is the address of the exception handler. The DPL field is set to 3.

`set_trap_gate(n,addr)`

Similar to the previous function, except the DPL field is set to 0.

`set_task_gate(n,gdt)`

Inserts a task gate in the *n* th IDT entry. The Segment Selector inside the gate stores the index in the GDT of the TSS containing the function to be activated. The Offset field is set to 0, while the DPL field is set to 3.

4.4.2. Preliminary Initialization of the IDT

The IDT is initialized and used by the BIOS routines while the computer still operates in Real Mode. Once Linux takes over, however, the IDT is moved to another area of RAM and initialized a second time, because Linux does not use any BIOS routine (see Appendix A).

The IDT is stored in the `idt_table` table, which includes 256 entries. The 6-byte `idt_descr` variable stores both the size of the IDT and its address and is used in the system initialization phase when the kernel sets up the `idtr` register with the `lidt` assembly language instruction.[*]

[*] Some old Pentium models have the notorious "f00f" bug, which allows User Mode programs to freeze the system. When executing on such CPUs, Linux uses a workaround based on initializing the `idtr` register with a fix-mapped read-only linear address pointing to the actual IDT (see the section "Fix-Mapped Linear Addresses" in Chapter 2).

During kernel initialization, the `setup_idt( )` assembly language function starts by filling all 256 entries of `idt_table` with the same interrupt gate, which refers to the `ignore_int( )` interrupt handler:

```
setup_idt:
lea ignore_int, %edx
movl $(_ _KERNEL_CS << 16), %eax
movw %dx, %ax /* selector = 0x0010 = cs */
movw $0x8e00, %dx /* interrupt gate, dpl=0, present */
lea idt_table, %edi
mov $256, %ecx
```

```
rp_sidt:
movl %eax, (%edi)
movl %edx, 4(%edi)
addl $8, %edi
dec %ecx
jne rp_sidt
ret
```

The `ignore_int( )` interrupt handler, which is in assembly language, may be viewed as a null handler that executes the following actions:

1. Saves the content of some registers in the stack.
2. Invokes the `printk( )` function to print an "Unknown interrupt" system message.
3. Restores the register contents from the stack.
4. Executes an `iret` instruction to restart the interrupted program.

The `ignore_int( )` handler should never be executed. The occurrence of "Unknown interrupt" messages on the console or in the log files denotes either a hardware problem (an I/O device is issuing unforeseen interrupts) or a kernel problem (an interrupt or exception is not being handled properly).

Following this preliminary initialization, the kernel makes a second pass in the IDT to replace some of the null handlers with meaningful trap and interrupt handlers. Once this is done, the IDT includes a specialized interrupt, trap, or system gate for each different exception issued by the control unit and for each IRQ recognized by the interrupt controller.

The next two sections illustrate in detail how this is done for exceptions and interrupts.

4.5. Exception Handling

Most exceptions issued by the CPU are interpreted by Linux as error conditions. When one of them occurs, the kernel sends a signal to the process that caused the exception to notify it of an anomalous condition. If, for instance, a process performs a division by zero, the CPU raises a "Divide error " exception, and the corresponding exception handler sends a `SIGFPE` signal to the current process, which then takes the necessary steps to recover or (if no signal handler is set for that signal) abort.

There are a couple of cases, however, where Linux exploits CPU exceptions to manage hardware resources more efficiently. A first case is already described in the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3. The "Device not available " exception is used together with the `TS` flag of the `cr0` register to force the kernel to load the floating point registers of the CPU with new values. A second case involves the "Page Fault " exception, which is used to defer allocating new page frames to the process until the last possible moment. The corresponding handler is complex because the exception may, or may not, denote an error condition (see the section "Page Fault Exception Handler" in Chapter 9).

Exception handlers have a standard structure consisting of three steps:

1. Save the contents of most registers in the Kernel Mode stack (this part is coded in assembly language).
2. Handle the exception by means of a high-level C function.
3. Exit from the handler by means of the `ret_from_exception( )` function.

To take advantage of exceptions, the IDT must be properly initialized with an exception handler function for each recognized exception. It is the job of the `trap_init( )` function to insert the final valuesthe functions that handle the exceptionsinto all IDT entries that refer to nonmaskable interrupts and exceptions. This is accomplished through the `set_trap_gate( )`, `set_intr_gate( )`, `set_system_gate( )`, `set_system_intr_gate( )`, and `set_task_gate( )` functions:

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
set_intr_gate(2,&nmi);
set_system_intr_gate(3,&int3);
set_system_gate(4,&overflow);
set_system_gate(5,&bounds);
set_trap_gate(6,&invalid_op);
set_trap_gate(7,&device_not_available);
set_task_gate(8,31);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
set_trap_gate(12,&stack_segment);
set_trap_gate(13,&general_protection);
set_intr_gate(14,&page_fault);
set_trap_gate(16,&coprocessor_error);
set_trap_gate(17,&alignment_check);
set_trap_gate(18,&machine_check);
set_trap_gate(19,&simd_coprocessor_error);
set_system_gate(128,&system_call);
```

The "Double fault" exception is handled by means of a task gate instead of a trap or system gate, because it denotes a serious kernel misbehavior. Thus, the exception handler that tries to print out the register values does not trust the current value of the `esp` register. When such an exception occurs, the CPU fetches the Task Gate Descriptor stored in the entry at index 8 of the IDT. This descriptor points to the special TSS segment descriptor stored in the 32nd entry of the GDT. Next, the CPU loads the `eip` and `esp` registers with the values stored in the corresponding TSS segment. As a result, the processor executes the `doublefault_fn()` exception handler on its own private stack.

Now we will look at what a typical exception handler does once it is invoked. Our description of exception handling will be a bit sketchy for lack of space. In particular we won't be able to cover:

1. The signal codes (see Table 11-8 in Chapter 11) sent by some handlers to the User Mode processes.
2. Exceptions that occur when the kernel is operating in MS-DOS emulation mode (vm86 mode), which must be dealt with differently.
3. "Debug " exceptions.

4.5.1. Saving the Registers for the Exception Handler

Let's use `handler_name` to denote the name of a generic exception handler. (The actual names of all the exception handlers appear on the list of macros in the previous section.) Each exception handler starts with the following assembly language instructions:

```
handler_name:
pushl $0 /* only for some exceptions */
pushl $do_handler_name
jmp error_code
```

If the control unit is not supposed to automatically insert a hardware error code on the stack when the exception occurs, the corresponding assembly language fragment includes a `pushl $0` instruction to pad the stack with a null value. Then the address of the high-level C function is pushed on the stack; its name consists of the exception handler name prefixed by `do_`.

The assembly language fragment labeled as `error_code` is the same for all exception handlers except the one for the "Device not available " exception (see the section "Saving and Loading the FPU, MMX, and XMM Registers" in Chapter 3). The code performs the following steps:

1. Saves the registers that might be used by the high-level C function on the stack.
2. Issues a `cld` instruction to clear the direction flag `DF` of `eflags` , thus making sure that autoincreases on the `edi` and `esi` registers will be used with string instructions .[*]

   [*] A single assembly language "string instruction," such as `rep;movsb` , is able to act on a whole block of data (string).

3. Copies the hardware error code saved in the stack at location `esp+36` in `edx`. Stores the value -1 in the same stack location. As we'll see in the section "Reexecution of System Calls" in Chapter 11, this value is used to separate `0x80` exceptions from other exceptions.
4. Loads `edi` with the address of the high-level `do_handler_name( )` C function saved in the stack at location `esp+32`; writes the contents of `es` in that stack location.
5. Loads in the `eax` register the current top location of the Kernel Mode stack. This address identifies the memory cell containing the last register value saved in step 1.
6. Loads the user data Segment Selector into the `ds` and `es` registers.
7. Invokes the high-level C function whose address is now stored in `edi`.

The invoked function receives its arguments from the `eax` and `edx` registers rather than from the stack. We have already run into a function that gets its arguments from the CPU registers: the `_ _switch_to( )` function, discussed in the section "Performing the Process Switch" in Chapter 3.

4.5.2. Entering and Leaving the Exception Handler

As already explained, the names of the C functions that implement exception handlers always consist of the prefix `do_` followed by the handler name. Most of these functions invoke the *do_trap()* function to store the hardware error code and the exception vector in the process descriptor of `current`, and then send a suitable signal to that process:

```
current->thread.error_code = error_code;
current->thread.trap_no = vector;
force_sig(sig_number, current);
```

The current process takes care of the signal right after the termination of the exception handler. The signal will be handled either in User Mode by the process's own signal handler (if it exists) or in Kernel Mode. In the latter case, the kernel usually kills the process (see Chapter 11). The signals sent by the exception handlers are listed in Table 4-1.

The exception handler always checks whether the exception occurred in User Mode or in Kernel Mode and, in the latter case, whether it was due to an invalid argument passed to a system call. We'll describe in the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10 how the kernel defends itself against invalid arguments passed to system calls. Any other exception raised in Kernel Mode is due to a kernel bug. In this case, the exception handler knows the kernel is misbehaving. In order to avoid data corruption on the hard disks, the handler invokes the `die( )` function, which prints the contents of all CPU registers on the console (this dump is called *kernel oops*) and terminates the `current` process by calling `do_exit( )` (see "Process Termination" in Chapter 3).

When the C function that implements the exception handling terminates, the code performs a `jmp` instruction to the `ret_from_exception( )` function. This function is described in the later section "Returning from Interrupts and Exceptions."

4.6. 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."

4.6.1. I/O Interrupt Handling

In general, an I/O interrupt handler must be flexible enough to service several devices at the same time. In the PCI bus architecture, for instance, several devices may share the same IRQ line. This means that the interrupt vector alone does not tell the whole story. In the example shown in Table 4-3, the same vector 43 is assigned to the USB port and to the sound card. However, some hardware 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 handler immediately, with maskable interrupts disabled.

Noncritical

> Actions such as updating data structures that are accessed only by the processor (for 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.

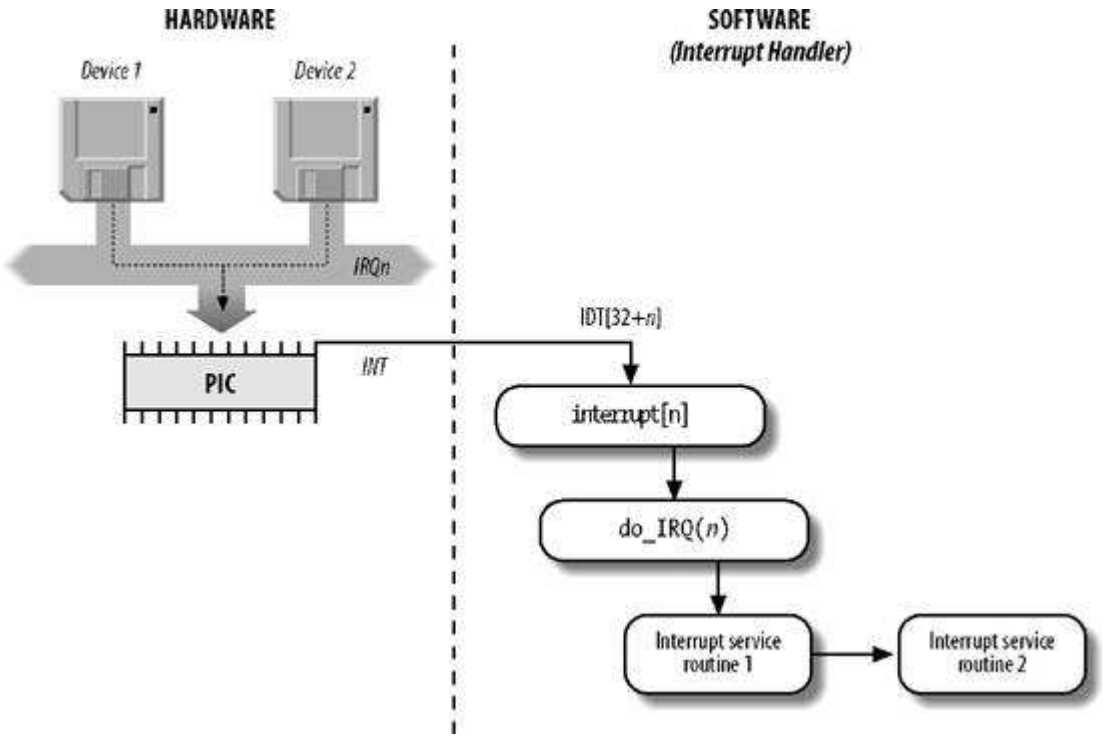
4.6.1.1. Interrupt vectors

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 IRQ lines. In particular:

- The interval timer device must be connected to the IRQ 0 line (see Chapter 6).
- The slave 8259A PIC must be connected to the IRQ 2 line (although more advanced PICs are 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 x 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

Vector range	Use

Table 4-2. Interrupt vectors in Linux

Vector range	Use
019 (0x0-0x13)	Nonmaskable interrupts and exceptions
2031 (0x14-0x1f)	Intel-reserved
32127 (0x20-0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (see Chapter 10)
129238 (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)
241250 (0xf1-0xfa)	Reserved by Linux for future use
251253 (0xfb-0xfd)	Interprocessor interrupts (see the section "Interprocessor Interrupt Handling" later in this chapter)
254 (0xfe)	Local APIC error interrupt (generated when the local APIC detects an erroneous condition)
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

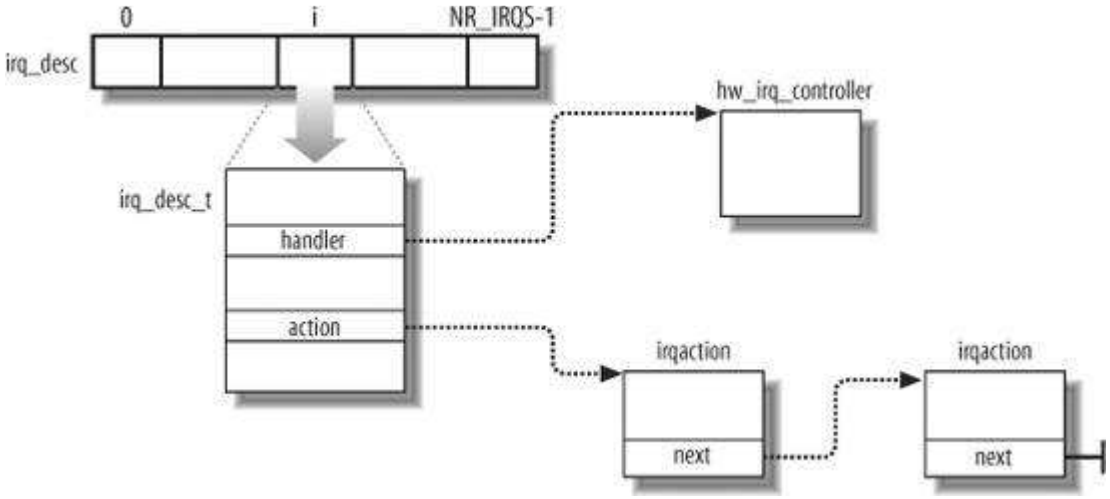
IRQ	INT	Hardware device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	First serial port
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor
14	46	EIDE disk controller's first chain
15	47	EIDE 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).

4.6.1.2. 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

Field	Description
`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 on an IRQ line, so as to disable the line in case a faulty hardware device keeps raising an interrupt over and over. Because the IRQ line can be shared among several devices, the kernel does not disable the line as soon as it detects a single unhandled interrupt. Rather, the kernel stores in the `irq_count` and `irqs_unhandled` fields of the `irq_desc_t` descriptor the total number of interrupts and the number of unexpected interrupts, respectively; when the 100,000th interrupt is 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

Flag name	Description
`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 x 86 architecture.
`IRQ_MASKED`	Not used.
`IRQ_PER_CPU`	Not used on the 80 x 86 architecture.

The `depth` field and the `IRQ_DISABLED` flag of the `irq_desc_t` descriptor specify whether the IRQ line is enabled or disabled. Every time the `disable_irq( )` or `disable_irq_nosync( )` function is invoked, the `depth` field is increased; if `depth` is equal to 0, the function disables the IRQ line and sets its `IRQ_DISABLED` flag.[*] Conversely, each invocation of the `enable_irq( )` function decreases the field; if `depth` becomes 0, the function enables the IRQ line and clears its `IRQ_DISABLED` flag.

[*] 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.

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,
.ack = mask_and_ack_8259A,
.end = end_8259A_irq,
.set_affinity = NULL
};
```

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

Field name	Description
`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

Flag name	Description
`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

Field name	Description
`__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)

4.6.1.3. 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 break a tie, the multi-APIC system uses the values in the arbitration priority registers of local APICs, as explained earlier. Because such values are automatically changed after every interrupt, the IRQ signals are, in most cases, fairly distributed among all CPUs.[*]

[*] 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.

In short, when a hardware device raises an IRQ signal, the multi-APIC system selects one of the 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.

4.6.1.4. 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.
- The *hard IRQ stack* is used when handling interrupts. There is one hard IRQ stack for each CPU in the system, and each stack is contained in a single page frame.
- The *soft IRQ stack* is used when handling deferrable functions (softirqs or tasklets; see the later 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.

4.6.1.5. 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 following two assembly language instructions:

[*] 256 vectors is an architectural limit for the 80x86 architecture. 32 of them are used or reserved for the CPU, so the usable vector space consists of 224 vectors.

```
pushl $n-256
jmp common_interrupt
```

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 ret_from_intr( ) (see the later section "Returning from Interrupts and Exceptions").

4.6.1.6. The do_IRQ( ) function

The do_IRQ( ) function is invoked to execute all interrupt service routines associated with an 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:

a. 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).
b. 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.
c. 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).
d. Stores the current value of the `esp` stack pointer register in the `previous_esp` field of the `thread_info` descriptor in the `irq_ctx` union of the local CPU (this field is used only when preparing the function call trace for a kernel oops).
e. Loads in the `esp` stack register the top location of the hard IRQ stack of the local CPU (the value in `hardirq_ctx[smp_processor_id( )]` plus 4096); the 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").

4.6.1.7. 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;
}
irq_desc[irq].handler->end(irq);
spin_unlock(&(irq_desc[irq].lock));
```

Before accessing the main IRQ descriptor, the kernel acquires the corresponding spin lock. We'll see 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 interrupts even when the IRQ line is disabled in the PIC.

`IRQ_INPROGRESS` *is set*

In a multiprocessor system, another CPU might be handling a previous occurrence of 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.[*]

[*] 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.

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!

4.6.1.8. Reviving a lost interrupt

The `_ _do_IRQ( )` function is small and simple, yet it works properly in most cases. Indeed, the `IRQ_PENDING`, `IRQ_INPROGRESS`, and `IRQ_DISABLED` flags ensure that interrupts are correctly handled even when the hardware is misbehaving. However, things may not work so smoothly in a multiprocessor system.

Suppose that a CPU has an IRQ line enabled. A hardware device raises the IRQ line, and the multi-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.

4.6.1.9. 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 `SA_INTERRUPT` flag is clear.
2. Executes each interrupt service routine of the interrupt through the following code:

```
3.  retval = 0;
4.  do {
5.  retval |= action->handler(irq, action->dev_id, regs);
6.  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).

7. Disables local interrupts with the `cli` assembly language instruction.
8. 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 number
- The remaining fields correspond to the register values pushed on automatically by the control unit

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.

4.6.1.10. 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 */dev/fd0* device file, which corresponds to the first floppy disk on the system.[*] The program can do this either by directly accessing */dev/fd0* or by mounting a filesystem on it. Floppy disk controllers are usually assigned IRQ 6; given this, a floppy driver may issue the following request:

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

```
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);
```

First, the `irq0` variable of type `irqaction` is initialized: the `handler` field is set to the address of the `timer_interrupt( )` function, the `flags` field is set to `SA_INTERRUPT`, the `name` field is set to *"timer"*, and the fifth field is set to `NULL` to show that no `dev_id` value is used. Next, the kernel invokes `setup_irq( )` to insert `irq0` in the list of `irqaction` descriptors associated with IRQ 0.

4.6.2. 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.

[*] 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: the paging unit combines write transfers into larger chunks before copying them into the frame buffer.

`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 Cache and the TLB" in Chapter 2.

The assembly language code of the interprocessor interrupt handlers is generated by the `BUILD_INTERRUPT` macro: it saves the registers, pushes the vector number minus 256 on the 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

4.7. Softirqs and Tasklets

We mentioned earlier in the section "Interrupt Handling" that several tasks among those executed by the kernel are not critical: they can be deferred for a long period of time, if necessary. Remember that the interrupt service routines of an interrupt handler are serialized, and often there should be no occurrence of an interrupt until the corresponding interrupt handler has terminated. Conversely, the deferrable tasks can execute with all interrupts enabled. Taking them out of the interrupt handler helps keep kernel response time small. This is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds.

Linux 2.6 answers such a challenge by using two kinds of non-urgent interruptible kernel functions: the so-called *deferrable functions*[*] (*softirqs* and *tasklets*), and those executed by means of some work queues (we will describe them in the section "Work Queues" later in this chapter).

[*] These are also called *software interrupt*s, but we denote them as "deferrable functions" to avoid confusion with programmed exceptions, which are referred to as "software interrupts " in Intel manuals.

Softirqs and tasklets are strictly correlated, because tasklets are implemented on top of softirqs. As a matter of fact, the term "softirq," which appears in the kernel source code, often denotes both kinds of deferrable functions. Another widely used term is *interrupt context* : it specifies that the kernel is currently executing either an interrupt handler or a deferrable function.

Softirqs are statically allocated (i.e., defined at compile time), while tasklets can also be allocated and initialized at runtime (for instance, when loading a kernel module). Softirqs can run concurrently on several CPUs, even if they are of the same type. Thus, softirqs are reentrant functions and must explicitly protect their data structures with spin locks. Tasklets do not have to worry about this, because their execution is controlled more strictly by the kernel. Tasklets of the same type are always serialized: in other words, the same type of tasklet cannot be executed by two CPUs at the same time. However, tasklets of different types can be executed concurrently on several CPUs. Serializing the tasklet simplifies the life of device driver developers, because the tasklet function needs not be reentrant.

Generally speaking, four kinds of operations can be performed on deferrable functions:

Initialization

Defines a new deferrable function; this operation is usually done when the kernel initializes itself or a module is loaded.

Activation

Marks a deferrable function as "pending" to be run the next time the kernel schedules a round of executions of deferrable functions. Activation can be done at any time (even while handling interrupts).

Masking

Selectively disables a deferrable function so that it will not be executed by the kernel even if activated. We'll see in the section "Disabling and Enabling Deferrable Functions" in Chapter 5 that disabling deferrable functions is sometimes essential.

Execution

Executes a pending deferrable function together with all other pending deferrable functions of the same type; execution is performed at well-specified times, explained later in the section "Softirqs."

Activation and execution are bound together: a deferrable function that has been activated by a given CPU must be executed on the same CPU. There is no self-evident reason suggesting that this rule is beneficial for system performance. Binding the deferrable function to the activating CPU could in theory make better use of the CPU hardware cache. After all, it is conceivable that the activating kernel thread accesses some data structures that will also be used by the deferrable function. However, the relevant lines could easily be no longer in the cache when the deferrable function is run because its execution can be delayed a long time. Moreover, binding a function to a CPU is always a potentially "dangerous" operation, because one CPU might end up very busy while the others are mostly idle.

4.7.1. Softirqs

Linux 2.6 uses a limited number of softirqs . For most purposes, tasklets are good enough and are much easier to write because they do not need to be reentrant.

As a matter of fact, only the six kinds of softirqs listed in Table 4-9 are currently defined.

Table 4-9. Softirqs used in Linux 2.6

Softirq	Index (priority)	Description
`HI_SOFTIRQ`	0	Handles high priority tasklets
`TIMER_SOFTIRQ`	1	Tasklets related to timer interrupts

Table 4-9. Softirqs used in Linux 2.6

Softirq	Index (priority)	Description
NET_TX_SOFTIRQ	2	Transmits packets to network cards
NET_RX_SOFTIRQ	3	Receives packets from network cards
SCSI_SOFTIRQ	4	Post-interrupt processing of SCSI commands
TASKLET_SOFTIRQ	5	Handles regular tasklets

The index of a sofirq determines its priority: a lower index means higher priority because softirq functions will be executed starting from index 0.

4.7.1.1. Data structures used for softirqs

The main data structure used to represent softirqs is the `softirq_vec` array, which includes 32 elements of type `softirq_action`. The priority of a softirq is the index of the corresponding `softirq_action` element inside the array. As shown in Table 4-9, only the first six entries of the array are effectively used. The `softirq_action` data structure consists of two fields: an `action` pointer to the softirq function and a `data` pointer to a generic data structure that may be needed by the softirq function.

Another critical field used to keep track both of kernel preemption and of nesting of kernel control paths is the 32-bit `preempt_count` field stored in the `tHRead_info` field of each process descriptor (see the section "Identifying a Process" in Chapter 3). This field encodes three distinct counters plus a flag, as shown in Table 4-10.

Table 4-10. Subfields of the preempt_count field (continues)

Bits	Description
07	Preemption counter (max value = 255)
815	Softirq counter (max value = 255).
1627	Hardirq counter (max value = 4096)
28	`PREEMPT_ACTIVE` flag

The first counter keeps track of how many times kernel preemption has been explicitly disabled on the local CPU; the value zero means that kernel preemption has not been explicitly disabled at all. The second counter specifies how many levels deep the disabling of deferrable functions is (level 0 means that deferrable functions are enabled). The third counter specifies the number of nested interrupt handlers on the local CPU (the value is increased by `irq_enter( )` and decreased by `irq_exit( )`; see the section "I/O Interrupt Handling" earlier in this chapter).

There is a good reason for the name of the `preempt_count` field: kernel preemptability has to be disabled either when it has been explicitly disabled by the kernel code (preemption counter not zero) or when the kernel is running in interrupt context. Thus, to determine whether the current process can be preempted, the kernel quickly checks for a zero value in the `preempt_count` field. Kernel preemption will be discussed in depth in the section "Kernel Preemption" in Chapter 5.

The `in_interrupt( )` macro checks the hardirq and softirq counters in the `current_thread_info( )->preempt_count` field. If either one of these two counters is positive, the macro yields a nonzero value, otherwise it yields the value zero. If the kernel does not make use of multiple Kernel Mode stacks, the macro always looks at the `preempt_count` field of the `thread_info` descriptor of the current process. If, however, the kernel makes use of multiple Kernel Mode stacks, the macro might look at the `preempt_count` field in the `tHRead_info` descriptor contained in a `irq_ctx` union associated with the local CPU. In this case, the macro returns a nonzero value because the field is always set to a positive value.

The last crucial data structure for implementing the softirqs is a per-CPU 32-bit mask describing the pending softirqs; it is stored in the `_ _softirq_pending` field of the `irq_cpustat_t` data structure (recall that there is one such structure per each CPU in the system; see Table 4-8). To get and set the value of the bit mask, the kernel makes use of the `local_softirq_pending( )` macro that selects the softirq bit mask of the local CPU.

4.7.1.2. Handling softirqs

The `open_softirq( )` function takes care of softirq initialization. It uses three parameters: the softirq index, a pointer to the softirq function to be executed, and a second pointer to a data structure that may be required by the softirq function. `open_softirq( )` limits itself to initializing the proper entry of the `softirq_vec` array.

Softirqs are activated by means of the `raise_softirq( )` function. This function, which receives as its parameter the softirq index `nr`, performs the following actions:

1. Executes the `local_irq_save` macro to save the state of the `IF` flag of the `eflags` register and to disable interrupts on the local CPU.
2. Marks the softirq as pending by setting the bit corresponding to the index `nr` in the softirq bit mask of the local CPU.
3. If `in_interrupt()` yields the value 1, it jumps to step 5. This situation indicates either that `raise_softirq( )` has been invoked in interrupt context, or that the softirqs are currently disabled.
4. Otherwise, invokes `wakeup_softirqd()` to wake up, if necessary, the *ksoftirqd kernel thread of the local CPU (see later)*.
5. Executes the `local_irq_restore` macro to restore the state of the `IF` flag saved in step 1.

Checks for active (pending) softirqs should be perfomed periodically, but without inducing too much overhead. They are performed in a few points of the kernel code. Here is a list of the most significant points (be warned that number and position of the softirq checkpoints change both with the kernel version and with the supported hardware architecture):

- When the kernel invokes the `local_bh_enable( )` function[*] to enable softirqs on the local CPU

  [*] The name `local_bh_enable( )` refers to a special type of deferrable function called "bottom half" that no longer exists in Linux 2.6.

- When the `do_IRQ( )` function finishes handling an I/O interrupt and invokes the `irq_exit( )` macro
- If the system uses an I/O APIC, when the `smp_apic_timer_interrupt( )` function finishes handling a local timer interrupt (see the section "Timekeeping Architecture in Multiprocessor Systems" in Chapter 6)
- In multiprocessor systems, when a CPU finishes handling a function triggered by a `CALL_FUNCTION_VECTOR` interprocessor interrupt
- When one of the special *ksoftirqd/n* kernel threads is awakened (see later)

4.7.1.3. The do_softirq( ) function

If pending softirqs are detected at one such checkpoint (`local_softirq_pending()` is not zero), the kernel invokes `do_softirq( )` to take care of them. This function performs the following actions:

1. If `in_interrupt( )` yields the value one, this function returns. This situation indicates either that `do_softirq( )` has been invoked in interrupt context or that the softirqs are currently disabled.
2. Executes `local_irq_save` to save the state of the `IF` flag and to disable the interrupts on the local CPU.
3. If the size of the `thread_union` structure is 4 KB, it switches to the soft IRQ stack, if necessary. This step is very similar to step 2 of `do_IRQ( )` in the earlier section "I/O Interrupt Handling;" of course, the `softirq_ctx` array is used instead of `hardirq_ctx`.
4. Invokes the `_ _do_softirq( )` function (see the following section).
5. If the soft IRQ stack has been effectively switched in step 3 above, it restores the original stack pointer into the `esp` register, thus switching back to the exception stack that was in use before.
6. Executes `local_irq_restore` to restore the state of the `IF` flag (local interrupts enabled or disabled) saved in step 2 and returns.

4.7.1.4. The _ _do_softirq( ) function

The `_ _do_softirq( )` function reads the softirq bit mask of the local CPU and executes the deferrable functions corresponding to every set bit. While executing a softirq function, new pending softirqs might pop up; in order to ensure a low latency time for the deferrable funtions, `_ _do_softirq( )` keeps running until all pending softirqs have been executed. This mechanism, however, could force `_ _do_softirq( )` to run for long periods of time, thus considerably delaying User Mode processes. For that reason, `_ _do_softirq( )` performs a fixed number of iterations and then returns. The remaining pending softirqs, if any, will be handled in due time by the *ksoftirqd* kernel thread described in the next section. Here is a short description of the actions performed by the function:

1. Initializes the iteration counter to 10.
2. Copies the softirq bit mask of the local CPU (selected by `local_softirq_pending( )`) in the `pending` local variable.
3. Invokes `local_bh_disable( )` to increase the softirq counter. It is somewhat counterintuitive that deferrable functions should be disabled before starting to execute them, but it really makes a lot of sense. Because the deferrable functions mostly run with interrupts enabled, an interrupt can be raised in the middle of the `_ _do_softirq( )` function. When `do_IRQ( )` executes the `irq_exit( )` macro, another instance of the `_ _do_softirq( )` function could be started. This has to be avoided, because deferrable functions must execute serially on the CPU. Thus, the first instance of `_ _do_softirq( )` disables deferrable functions, so that every new instance of the function will exit at step 1 of `do_softirq( )`.
4. Clears the softirq bitmap of the local CPU, so that new softirqs can be activated (the value of the bit mask has already been saved in the `pending` local variable in step 2).
5. Executes `local_irq_enable( )` to enable local interrupts.
6. For each bit set in the `pending` local variable, it executes the corresponding softirq function; recall that the function address for the softirq with index `n` is stored in `softirq_vec[n]->action`.
7. Executes `local_irq_disable()` to disable local interrupts.
8. Copies the softirq bit mask of the local CPU into the `pending` local variable and decreases the iteration counter one more time.
9. If `pending` is not zeroat least one softirq has been activated since the start of the last iterationand the iteration counter is still positive, it jumps back to step 4.
10. If there are more pending softirqs, it invokes `wakeup_softirqd( )` to wake up the kernel thread that takes care of the softirqs for the local CPU (see next section).
11. Subtracts 1 from the softirq counter, thus reenabling the deferrable functions.

4.7.1.5. The ksoftirqd kernel threads

In recent kernel versions, each CPU has its own *ksoftirqd/n* kernel thread (where *n* is the logical number of the CPU). Each *ksoftirqd/n* kernel thread runs the `ksoftirqd( )` function, which essentially executes the following loop:

```
for(;;) {
set_current_state(TASK_INTERRUPTIBLE );
schedule( );
/* now in TASK_RUNNING state */
while (local_softirq_pending( )) {
preempt_disable();
do_softirq( );
preempt_enable();
cond_resched( );
}
}
```

When awakened, the kernel thread checks the `local_softirq_pending()` softirq bit mask and invokes, if necessary, `do_softirq( )`. If there are no softirqs pending, the function puts the current process in the `TASK_INTERRUPTIBLE` state and invokes then the `cond_resched()`

function to perform a process switch if required by the current process (flag `TIF_NEED_RESCHED` of the current `thread_info` set).

The *ksoftirqd/n* kernel threads represent a solution for a critical trade-off problem.

Softirq functions may reactivate themselves; in fact, both the networking softirqs and the tasklet softirqs do this. Moreover, external events, such as packet flooding on a network card, may activate softirqs at very high frequency.

The potential for a continuous high-volume flow of softirqs creates a problem that is solved by introducing kernel threads. Without them, developers are essentially faced with two alternative strategies.

The first strategy consists of ignoring new softirqs that occur while `do_softirq( )` is running. In other words, the `do_softirq( )` function could determine what softirqs are pending when the function is started and then execute their functions. Next, it would terminate without rechecking the pending softirqs. This solution is not good enough. Suppose that a softirq function is reactivated during the execution of `do_softirq( )`. In the worst case, the softirq is not executed again until the next timer interrupt, even if the machine is idle. As a result, softirq latency time is unacceptable for networking developers.

The second strategy consists of continuously rechecking for pending softirqs. The `do_softirq( )` function could keep checking the pending softirqs and would terminate only when none of them is pending. While this solution might satisfy networking developers, it can certainly upset normal users of the system: if a high-frequency flow of packets is received by a network card or a softirq function keeps activating itself, the `do_softirq( )` function never returns, and the User Mode programs are virtually stopped.

The *ksoftirqd/n* kernel threads try to solve this difficult trade-off problem. The `do_softirq( )` function determines what softirqs are pending and executes their functions. After a few iterations, if the flow of softirqs does not stop, the function wakes up the kernel thread and terminates (step 10 of `_ _do_softirq( )`). The kernel thread has low priority, so user programs have a chance to run; but if the machine is idle, the pending softirqs are executed quickly.

4.7.2. Tasklets

Tasklets are the preferred way to implement deferrable functions in I/O drivers. As already explained, tasklets are built on top of two softirqs named `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`. Several tasklets may be associated with the same softirq, each tasklet carrying its own function. There is no real difference between the two softirqs, except that `do_softirq( )` executes `HI_SOFTIRQ`'s tasklets before `TASKLET_SOFTIRQ`'s tasklets.

Tasklets and high-priority tasklets are stored in the `tasklet_vec` and `tasklet_hi_vec` arrays, respectively. Both of them include `NR_CPUS` elements of type `tasklet_head`, and each element consists of a pointer to a list of *tasklet descriptors*. The tasklet descriptor is a data structure of type `tasklet_struct`, whose fields are shown in Table 4-11.

Table 4-11. The fields of the tasklet descriptor

Field name	Description
`next`	Pointer to next descriptor in the list
`state`	Status of the tasklet
`count`	Lock counter
`func`	Pointer to the tasklet function
`data`	An unsigned long integer that may be used by the tasklet function

The `state` field of the tasklet descriptor includes two flags:

`TASKLET_STATE_SCHED`

When set, this indicates that the tasklet is pending (has been scheduled for execution); it also means that the tasklet descriptor is inserted in one of the lists of the `tasklet_vec` and `tasklet_hi_vec` arrays.

`TASKLET_STATE_RUN`

When set, this indicates that the tasklet is being executed; on a uniprocessor system this flag is not used because there is no need to check whether a specific tasklet is running.

Let's suppose you're writing a device driver and you want to use a tasklet: what has to be done? First of all, you should allocate a new `tasklet_struct` data structure and initialize it by invoking `tasklet_init( )`; this function receives as its parameters the address of the tasklet descriptor, the address of your tasklet function, and its optional integer argument.

The tasklet may be selectively disabled by invoking either `tasklet_disable_nosync( )` or `tasklet_disable( )`. Both functions increase the `count` field of the tasklet descriptor, but the latter function does not return until an already running instance of the tasklet function has terminated. To reenable the tasklet, use `tasklet_enable( )`.

To activate the tasklet, you should invoke either the `tasklet_schedule( )` function or the `tasklet_hi_schedule( )` function, according to the priority that you require for the tasklet. The two functions are very similar; each of them performs the following actions:

1. Checks the `TASKLET_STATE_SCHED` flag; if it is set, returns (the tasklet has already been scheduled).
2. Invokes `local_irq_save` to save the state of the `IF` flag and to disable local interrupts.
3. Adds the tasklet descriptor at the beginning of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`, where `n` denotes the logical number of the local CPU.

4. Invokes `raise_softirq_irqoff( )` to activate either the `TASKLET_SOFTIRQ` or the `HI_SOFTIRQ` softirq (this function is similar to `raise_softirq( )`, except that it assumes that local interrupts are already disabled).
5. Invokes `local_irq_restore` to restore the state of the `IF` flag.

Finally, let's see how the tasklet is executed. We know from the previous section that, once activated, softirq functions are executed by the `do_softirq( )` function. The softirq function associated with the `HI_SOFTIRQ` softirq is named `tasklet_hi_action( )`, while the function associated with `TASKLET_SOFTIRQ` is named `tasklet_action( )`. Once again, the two functions are very similar; each of them:

1. Disables local interrupts.
2. Gets the logical number `n` of the local CPU.
3. Stores the address of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` in the `list` local variable.
4. Puts a `NULL` address in `tasklet_vec[n]` or `tasklet_hi_vec[n]`, thus emptying the list of scheduled tasklet descriptors.
5. Enables local interrupts.
6. For each tasklet descriptor in the list pointed to by `list`:
    a. In multiprocessor systems, checks the `TASKLET_STATE_RUN` flag of the tasklet.
        - If it is set, a tasklet of the same type is already running on another CPU, so the function reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` and activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again. In this way, execution of the tasklet is deferred until no other tasklets of the same type are running on other CPUs.
        - Otherwise, the tasklet is not running on another CPU: sets the flag so that the tasklet function cannot be executed on other CPUs.
    b. Checks whether the tasklet is disabled by looking at the `count` field of the tasklet descriptor. If the tasklet is disabled, it clears its `TASKLET_STATE_RUN` flag and reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`; then the function activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again.
    c. If the tasklet is enabled, it clears the `TASKLET_STATE_SCHED` flag and executes the tasklet function.

Notice that, unless the tasklet function reactivates itself, every tasklet activation triggers at most one execution of the tasklet function.

4.8. Work Queues

The *work queues* have been introduced in Linux 2.6 and replace a similar construct called "task queue" used in Linux 2.4. They allow kernel functions to be activated (much like deferrable functions) and later executed by special kernel threads called *worker threads* .

Despite their similarities, deferrable functions and work queues are quite different. The main difference is that deferrable functions run in interrupt context while functions in work queues run in process context. Running in process context is the only way to execute functions that

can block (for instance, functions that need to access some block of data on disk) because, as already observed in the section "Nested Execution of Exception and Interrupt Handlers" earlier in this chapter, no process switch can take place in interrupt context. Neither deferrable functions nor functions in a work queue can access the User Mode address space of a process. In fact, a deferrable function cannot make any assumption about the process that is currently running when it is executed. On the other hand, a function in a work queue is executed by a kernel thread, so there is no User Mode address space to access.

4.8.1.

4.8.1.1. Work queue data structures

The main data structure associated with a work queue is a descriptor called `workqueue_struct`, which contains, among other things, an array of `NR_CPUS` elements, the maximum number of CPUs in the system.[*] Each element is a descriptor of type `cpu_workqueue_struct`, whose fields are shown in Table 4-12.

[*] The reason for duplicating the work queue data structures in multiprocessor systems is that per-CPU local data structures yield a much more efficient code (see the section "Per-CPU Variables" in Chapter 5).

Table 4-12. The fields of the cpu_workqueue_struct structure

Field name	Description
`lock`	Spin lock used to protect the structure
`remove_sequence`	Sequence number used by `flush_workqueue( )`
`insert_sequence`	Sequence number used by `flush_workqueue( )`
`worklist`	Head of the list of pending functions
`more_work`	Wait queue where the worker thread waiting for more work to be done sleeps
`work_done`	Wait queue where the processes waiting for the work queue to be flushed sleep
`wq`	Pointer to the `workqueue_struct` structure containing this descriptor
`tHRead`	Process descriptor pointer of the worker thread of the structure
`run_depth`	Current execution depth of `run_workqueue( )` (this field may become greater than one when a function in the work queue list blocks)

The `worklist` field of the `cpu_workqueue_struct` structure is the head of a doubly linked list collecting the pending functions of the work queue. Every pending function is represented by a `work_struct` data structure, whose fields are shown in Table 4-13.

Table 4-13. The fields of the work_struct structure

Field name	Description
pending	Set to 1 if the function is already in a work queue list, 0 otherwise
entry	Pointers to next and previous elements in the list of pending functions
func	Address of the pending function
data	Pointer passed as a parameter to the pending function
wq_data	Usually points to the parent cpu_workqueue_struct descriptor
timer	Software timer used to delay the execution of the pending function

4.8.1.2. Work queue functions

The `create_workqueue("foo" )` function receives as its parameter a string of characters and returns the address of a `workqueue_struct` descriptor for the newly created work queue. The function also creates *n* worker threads (where *n* is the number of CPUs effectively present in the system), named after the string passed to the function: *foo/0*, *foo/1*, and so on. The `create_singlethread_workqueue( )` function is similar, but it creates just one worker thread, no matter what the number of CPUs in the system is. To destroy a work queue the kernel invokes the `destroy_workqueue( )` function, which receives as its parameter a pointer to a `workqueue_struct` array.

`queue_work( )` inserts a function (already packaged inside a `work_struct` descriptor) in a work queue; it receives a pointer `wq` to the `workqueue_struct` descriptor and a pointer `work` to the `work_struct` descriptor. `queue_work( )` essentially performs the following steps:

1. Checks whether the function to be inserted is already present in the work queue (`work->pending` field equal to 1); if so, terminates.
2. Adds the `work_struct` descriptor to the work queue list, and sets `work->pending` to 1.
3. If a worker thread is sleeping in the `more_work` wait queue of the local CPU's `cpu_workqueue_struct` descriptor, the function wakes it up.

The `queue_delayed_work( )` function is nearly identical to `queue_work( )`, except that it receives a third parameter representing a time delay in system ticks (see Chapter 6). It is used to ensure a minimum delay before the execution of the pending function. In practice, `queue_delayed_work( )` relies on the software timer in the `timer` field of the `work_struct` descriptor to defer the actual insertion of the `work_struct` descriptor in the work queue list. `cancel_delayed_work( )` cancels a previously scheduled work queue function, provided that the corresponding `work_struct` descriptor has not already been inserted in the work queue list.

Every worker thread continuously executes a loop inside the `worker_thread( )` function; most of the time the thread is sleeping and waiting for some work to be queued. Once awakened, the worker thread invokes the `run_workqueue( )` function, which essentially

removes every work_struct descriptor from the work queue list of the worker thread and executes the corresponding pending function. Because work queue functions can block, the worker thread can be put to sleep and even migrated to another CPU when resumed.[*]

[*] Strangely enough, a worker thread can be executed by every CPU, not just the CPU corresponding to the cpu_workqueue_struct descriptor to which the worker thread belongs. Therefore, queue_work( ) inserts a function in the queue of the local CPU, but that function may be executed by any CPU in the systems.

Sometimes the kernel has to wait until all pending functions in a work queue have been executed. The flush_workqueue( ) function receives a workqueue_struct descriptor address and blocks the calling process until all functions that are pending in the work queue terminate. The function, however, does not wait for any pending function that was added to the work queue following flush_workqueue( ) invocation; the remove_sequence and insert_sequence fields of every cpu_workqueue_struct descriptor are used to recognize the newly added pending functions.

4.8.1.3. The predefined work queue

In most cases, creating a whole set of worker threads in order to run a function is overkill. Therefore, the kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer. The predefined work queue is nothing more than a standard work queue that may include functions of different kernel layers and I/O drivers; its workqueue_struct descriptor is stored in the keventd_wq array. To make use of the predefined work queue, the kernel offers the functions listed in Table 4-14.

Table 4-14. Helper functions for the predefined work queue

Predefined work queue function	Equivalent standard work queue function
schedule_work(w)	queue_work(keventd_wq,w)
schedule_delayed_work(w,d)	queue_delayed_work(keventd_wq,w,d) (on any CPU)
schedule_delayed_work_on(cpu,w,d)	queue_delayed_work(keventd_wq,w,d) (on a given CPU)
flush_scheduled_work( )	flush_workqueue(keventd_wq)

The predefined work queue saves significant system resources when the function is seldom invoked. On the other hand, functions executed in the predefined work queue should not block for a long time: because the execution of the pending functions in the work queue list is serialized on each CPU, a long delay negatively affects the other users of the predefined work queue.

In addition to the general *events* queue, you'll find a few specialized work queues in Linux 2.6. The most significant is the *kblockd* work queue used by the block device layer (see Chapter 14).

4.8. Work Queues

The *work queues* have been introduced in Linux 2.6 and replace a similar construct called "task queue" used in Linux 2.4. They allow kernel functions to be activated (much like deferrable functions) and later executed by special kernel threads called *worker threads* .

Despite their similarities, deferrable functions and work queues are quite different. The main difference is that deferrable functions run in interrupt context while functions in work queues run in process context. Running in process context is the only way to execute functions that can block (for instance, functions that need to access some block of data on disk) because, as already observed in the section "Nested Execution of Exception and Interrupt Handlers" earlier in this chapter, no process switch can take place in interrupt context. Neither deferrable functions nor functions in a work queue can access the User Mode address space of a process. In fact, a deferrable function cannot make any assumption about the process that is currently running when it is executed. On the other hand, a function in a work queue is executed by a kernel thread, so there is no User Mode address space to access.

4.8.1.

4.8.1.1. Work queue data structures

The main data structure associated with a work queue is a descriptor called `workqueue_struct`, which contains, among other things, an array of `NR_CPUS` elements, the maximum number of CPUs in the system.[*] Each element is a descriptor of type `cpu_workqueue_struct`, whose fields are shown in Table 4-12.

[*] The reason for duplicating the work queue data structures in multiprocessor systems is that per-CPU local data structures yield a much more efficient code (see the section "Per-CPU Variables" in Chapter 5).

Table 4-12. The fields of the cpu_workqueue_struct structure

Field name	Description
`lock`	Spin lock used to protect the structure
`remove_sequence`	Sequence number used by `flush_workqueue( )`
`insert_sequence`	Sequence number used by `flush_workqueue( )`
`worklist`	Head of the list of pending functions
`more_work`	Wait queue where the worker thread waiting for more work to be done sleeps
`work_done`	Wait queue where the processes waiting for the work queue to be flushed sleep
`wq`	Pointer to the `workqueue_struct` structure containing this descriptor
`tHRead`	Process descriptor pointer of the worker thread of the structure

Table 4-12. The fields of the cpu_workqueue_struct structure

Field name	Description
run_depth	Current execution depth of run_workqueue( ) (this field may become greater than one when a function in the work queue list blocks)

The worklist field of the cpu_workqueue_struct structure is the head of a doubly linked list collecting the pending functions of the work queue. Every pending function is represented by a work_struct data structure, whose fields are shown in Table 4-13.

Table 4-13. The fields of the work_struct structure

Field name	Description
pending	Set to 1 if the function is already in a work queue list, 0 otherwise
entry	Pointers to next and previous elements in the list of pending functions
func	Address of the pending function
data	Pointer passed as a parameter to the pending function
wq_data	Usually points to the parent cpu_workqueue_struct descriptor
timer	Software timer used to delay the execution of the pending function

4.8.1.2. Work queue functions

The create_workqueue("foo") function receives as its parameter a string of characters and returns the address of a workqueue_struct descriptor for the newly created work queue. The function also creates *n* worker threads (where *n* is the number of CPUs effectively present in the system), named after the string passed to the function: *foo/0*, *foo/1*, and so on. The create_singlethread_workqueue( ) function is similar, but it creates just one worker thread, no matter what the number of CPUs in the system is. To destroy a work queue the kernel invokes the destroy_workqueue( ) function, which receives as its parameter a pointer to a workqueue_struct array.

queue_work( ) inserts a function (already packaged inside a work_struct descriptor) in a work queue; it receives a pointer wq to the workqueue_struct descriptor and a pointer work to the work_struct descriptor. queue_work( ) essentially performs the following steps:

1. Checks whether the function to be inserted is already present in the work queue (work->pending field equal to 1); if so, terminates.
2. Adds the work_struct descriptor to the work queue list, and sets work->pending to 1.
3. If a worker thread is sleeping in the more_work wait queue of the local CPU's cpu_workqueue_struct descriptor, the function wakes it up.

The queue_delayed_work( ) function is nearly identical to queue_work( ), except that it receives a third parameter representing a time delay in system ticks (see Chapter 6). It is used to ensure a minimum delay before the execution of the pending function. In practice, queue_delayed_work( ) relies on the software timer in the timer field of the work_struct descriptor to defer the actual insertion of the work_struct descriptor in the work queue list. cancel_delayed_work( ) cancels a previously scheduled work queue function, provided that the corresponding work_struct descriptor has not already been inserted in the work queue list.

Every worker thread continuously executes a loop inside the worker_thread( ) function; most of the time the thread is sleeping and waiting for some work to be queued. Once awakened, the worker thread invokes the run_workqueue( ) function, which essentially removes every work_struct descriptor from the work queue list of the worker thread and executes the corresponding pending function. Because work queue functions can block, the worker thread can be put to sleep and even migrated to another CPU when resumed.[*]

[*] Strangely enough, a worker thread can be executed by every CPU, not just the CPU corresponding to the cpu_workqueue_struct descriptor to which the worker thread belongs. Therefore, queue_work( ) inserts a function in the queue of the local CPU, but that function may be executed by any CPU in the systems.

Sometimes the kernel has to wait until all pending functions in a work queue have been executed. The flush_workqueue( ) function receives a workqueue_struct descriptor address and blocks the calling process until all functions that are pending in the work queue terminate. The function, however, does not wait for any pending function that was added to the work queue following flush_workqueue( ) invocation; the remove_sequence and insert_sequence fields of every cpu_workqueue_struct descriptor are used to recognize the newly added pending functions.

4.8.1.3. The predefined work queue

In most cases, creating a whole set of worker threads in order to run a function is overkill. Therefore, the kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer. The predefined work queue is nothing more than a standard work queue that may include functions of different kernel layers and I/O drivers; its workqueue_struct descriptor is stored in the keventd_wq array. To make use of the predefined work queue, the kernel offers the functions listed in Table 4-14.

Table 4-14. Helper functions for the predefined work queue

Predefined work queue function	Equivalent standard work queue function
schedule_work(w)	queue_work(keventd_wq,w)
schedule_delayed_work(w,d)	queue_delayed_work(keventd_wq,w,d) (on any CPU)
schedule_delayed_work_on(cpu,w,d)	queue_delayed_work(keventd_wq,w,d) (on a given CPU)
flush_scheduled_work( )	flush_workqueue(keventd_wq)

The predefined work queue saves significant system resources when the function is seldom invoked. On the other hand, functions executed in the predefined work queue should not block for a long time: because the execution of the pending functions in the work queue list is serialized on each CPU, a long delay negatively affects the other users of the predefined work queue.

In addition to the general *events* queue, you'll find a few specialized work queues in Linux 2.6. The most significant is the *kblockd* work queue used by the block device layer (see Chapter 14).