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).

Intro and class layout

- **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 request, via an assembly language instruction such as **int or sysenter** for a kernel service.
- In this Class,
  - 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.
  - The remaining 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

- 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 (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.
- 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**.
  - 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. 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.
- 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.

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.
  - **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).
  - **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.

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:
  - Monitors the IRQ lines, checking for raised signals. If two or more IRQ lines are raised, selects the one having the lower pin number.
  - If a raised signal occurs on an IRQ line:
    - **Converts** the raised signal received into a corresponding **vector**.
    - **Stores the vector** in an Interrupt Controller I/O **port**, thus allowing the CPU to read it via the data bus.
    - Sends a raised signal to the processor **INTR pin**, that is, issues an interrupt.
    - 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**.
  - Goes back to step 1.

Mapping and disabling IRQs

- 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.**
- 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.

PIC 8259 configuration

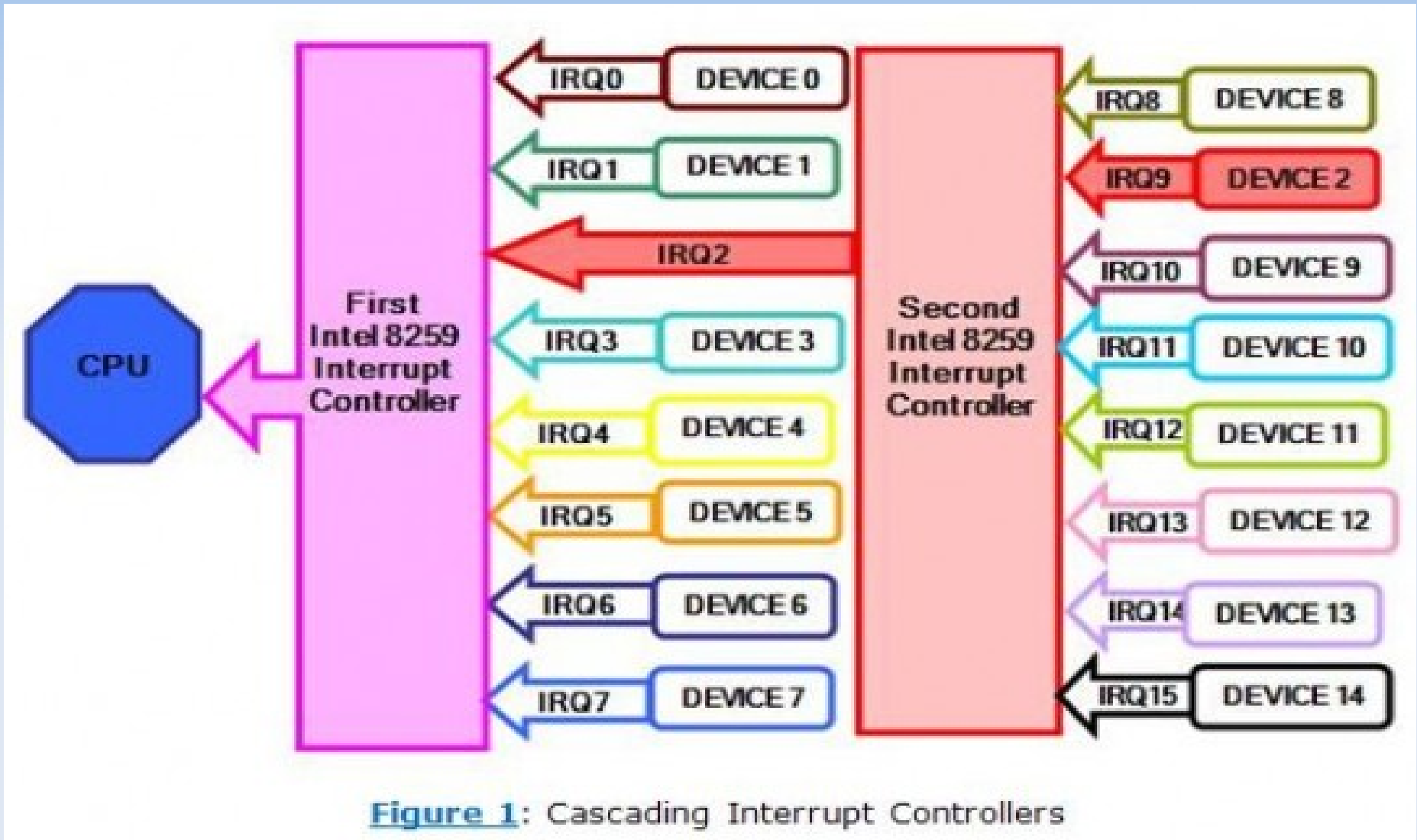


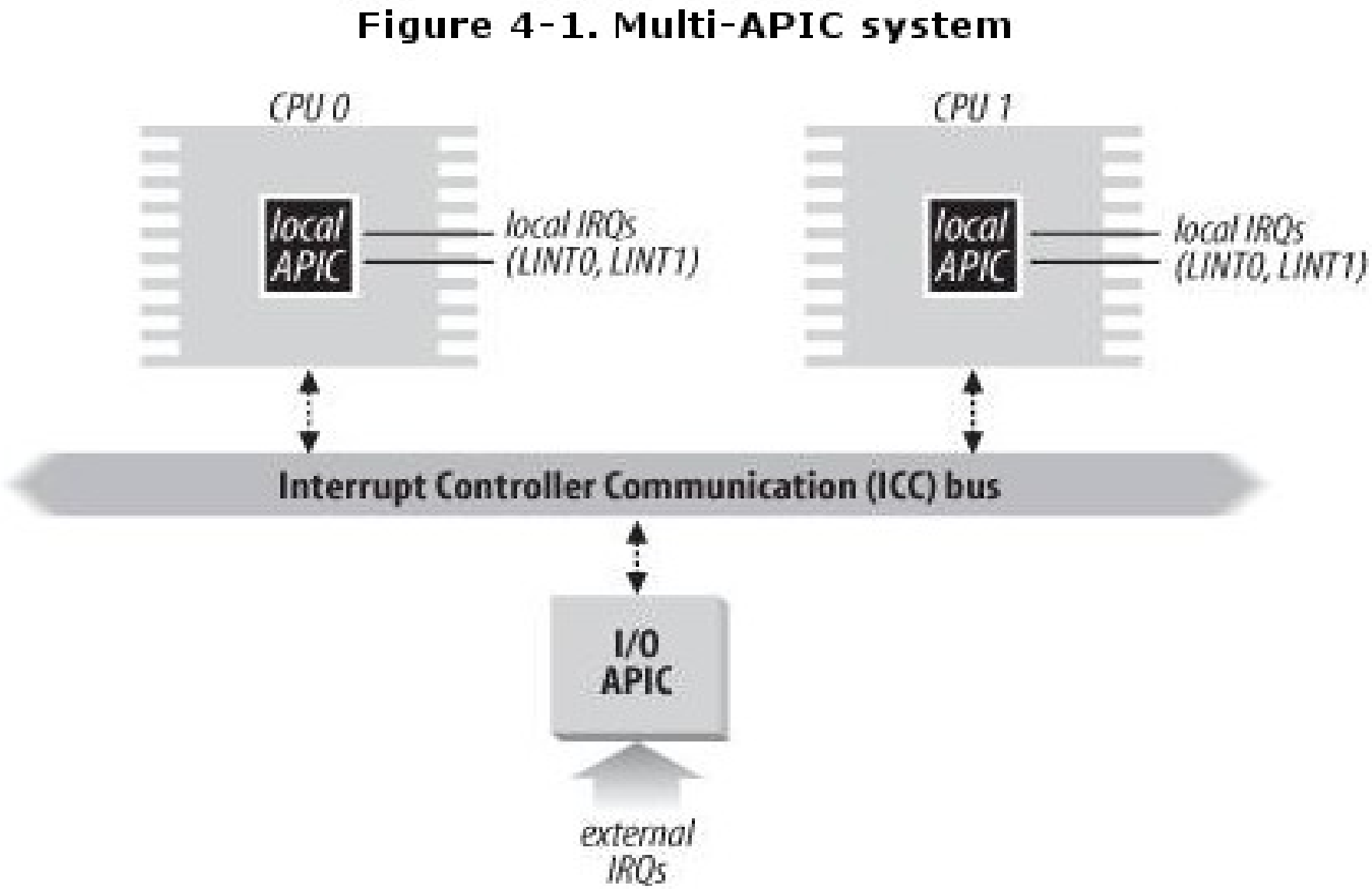
Figure 1: Cascading Interrupt Controllers

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**.

Multi-APIC system



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.

IPIs

- 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.

current uniprocessor systems

- 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 exact number depends on the processor model.) The values from 20 to 31 are reserved by Intel for future development.
- Each exception is handled by a **specific exception handler**, which usually sends a **Unix signal to the process** that caused the exception.
- 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 following list gives the vector, the name, the type, and a brief description of the exceptions found in 80x86 processors.
- **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 .

Exception list

- **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.
- **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.

Exception list

- **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.
- **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.

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
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

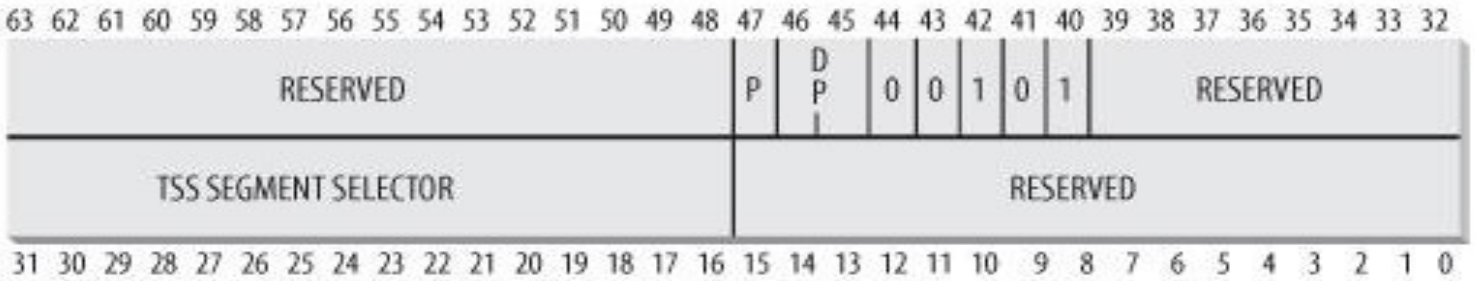
4.2.3. Interrupt Descriptor Table

- 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.
- 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 40-43 identifies the descriptor type.

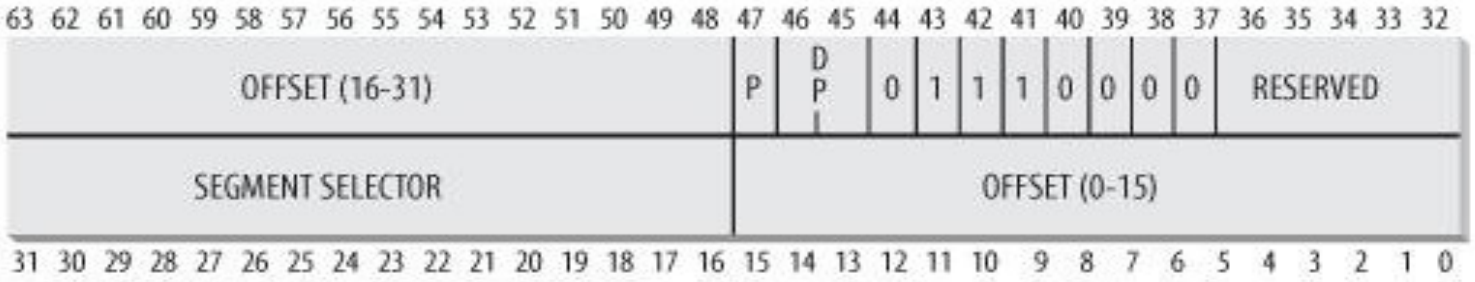
Figure 4-2. Gate descriptors' format

Figure 4-2. Gate descriptors' format

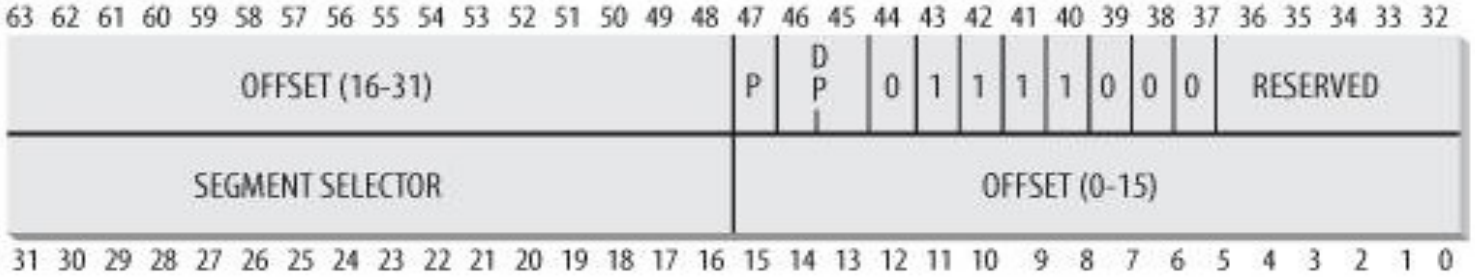
Task Gate Descriptor



Interrupt Gate Descriptor

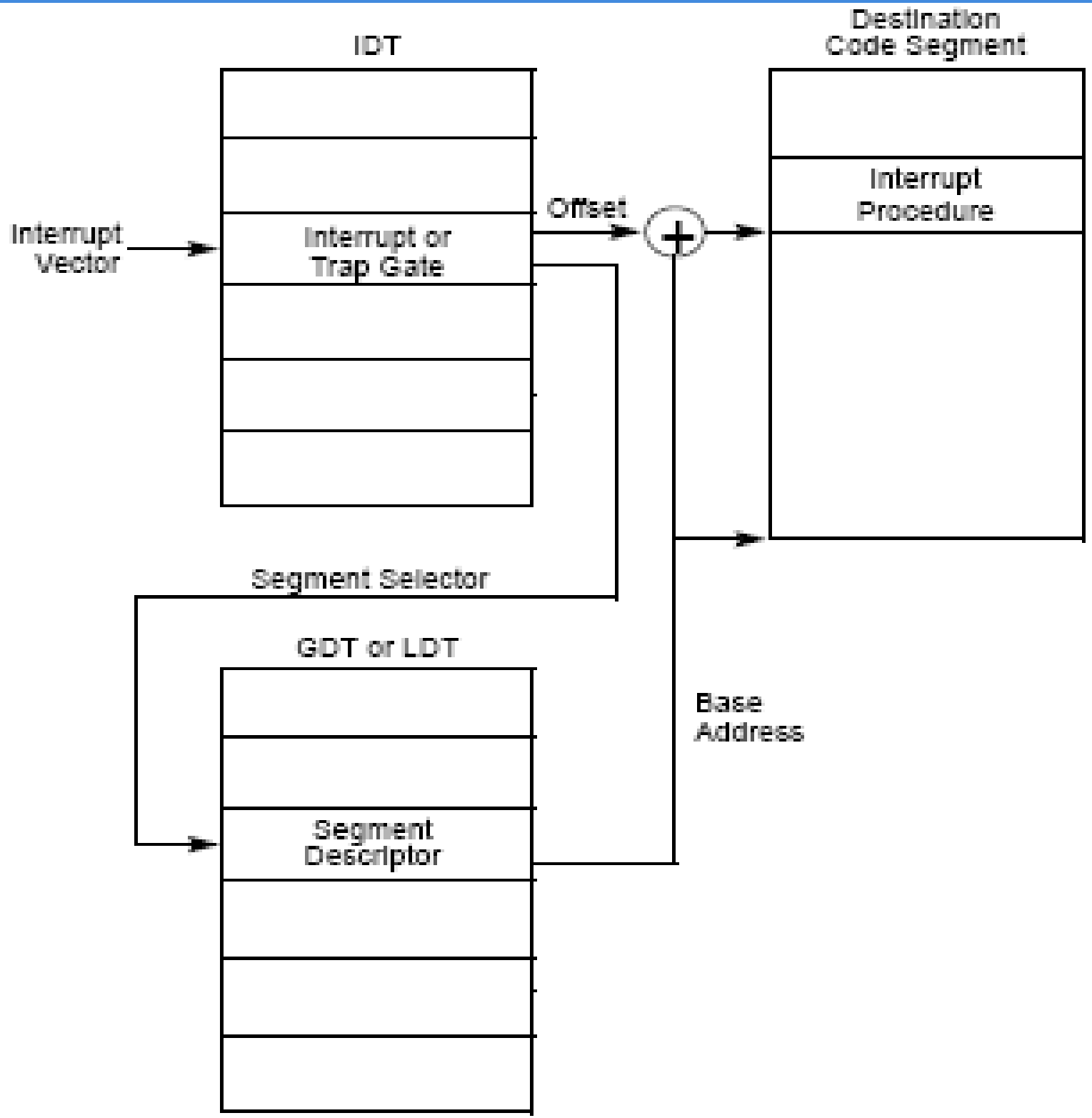


Trap Gate Descriptor



descriptors

- 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.).


IDT
Destination
Code Segment
Interrupt
Vector
Interrupt or
Trap Gate
Offset
Interrupt
Procedure
Segment Selector
GDT or LDT
Base
Address
Segment
Descriptor

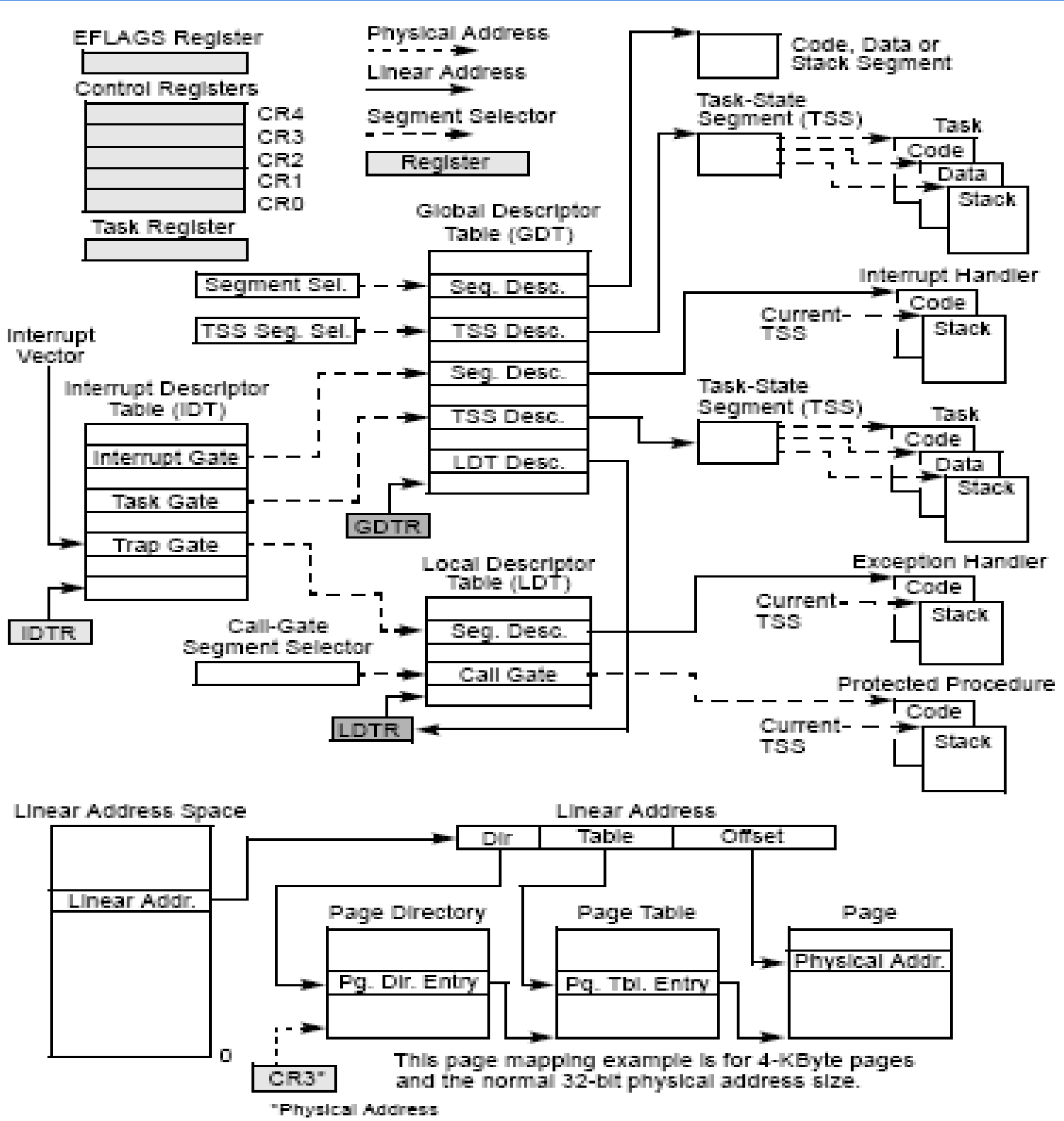
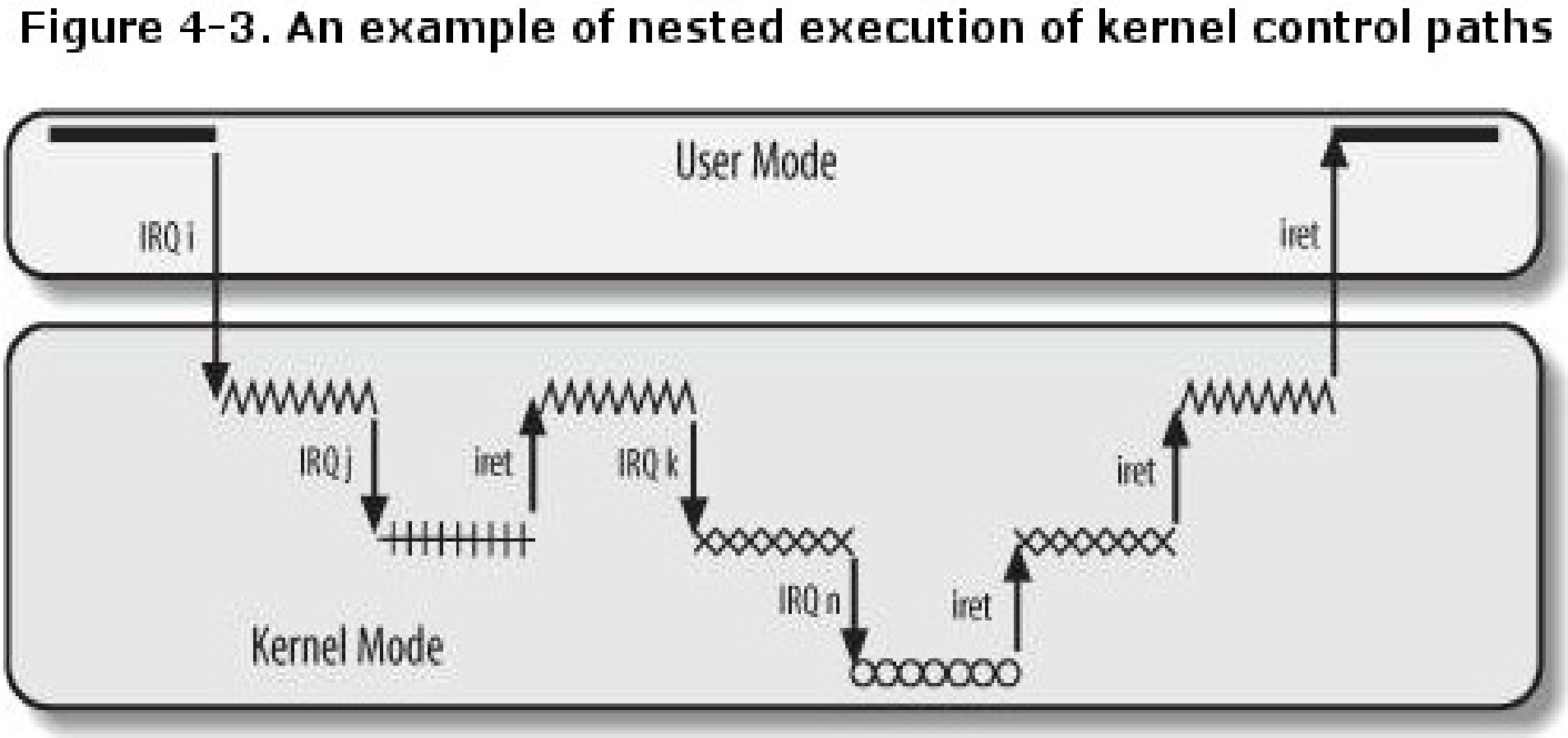

EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Physical Address
Linear Address
Segment Selector
Register
Code, Data or Stack Segment
Task-State Segment (TSS)
Task
Code
Data
Stack
Global Descriptor Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LDT Desc.
Interrupt Handler
Code
Stack
Current-TSS
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
GDTR
IDTR
Task-State Segment (TSS)
Task
Code
Data
Stack
Local Descriptor Table (LDT)
Call-Gate Segment Selector
Seg. Desc.
Call Gate
LDTR
Exception Handler
Code
Stack
Current-TSS
Protected Procedure
Code
Stack
Current-TSS
Linear Address Space
Linear Addr.
0
Linear Address
Dir
Table
Offset
Page Directory
Pg. Dir. Entry
Page Table
Pg. Tbl. Entry
Page
Physical Addr.
CR3*
This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.
*Physical Address

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



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**.

- **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 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**.

- 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.**

  - **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.**

Nested exceptions

- 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**.

KCP interleaving

- An **interrupt handler may preempt both other interrupt handlers and exception handlers**. Conversely, an exception handler never preempts an interrupt handler.
- 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.
- 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.

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**.
  - 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.

Exception handlers

- Exception handlers have a **standard structure consisting of three steps**:
  - **Save the contents of most registers** in the Kernel Mode stack (this part is coded in assembly language).
  - **Handle the exception** by means of a high-level C function.
  - **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 values, the functions that handle the exceptions, into all IDT entries that refer to nonmaskable interrupts and exceptions.

Filling the idt with exception handlers

```
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);
```

4.6. Interrupt Handling

- Interrupts 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.

  - **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.

Urgency

- 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."

Table 4-2. Interrupt vectors in Linux

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

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).

  - 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.)

Selecting a IRQ line for a device

- 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.

example

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

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

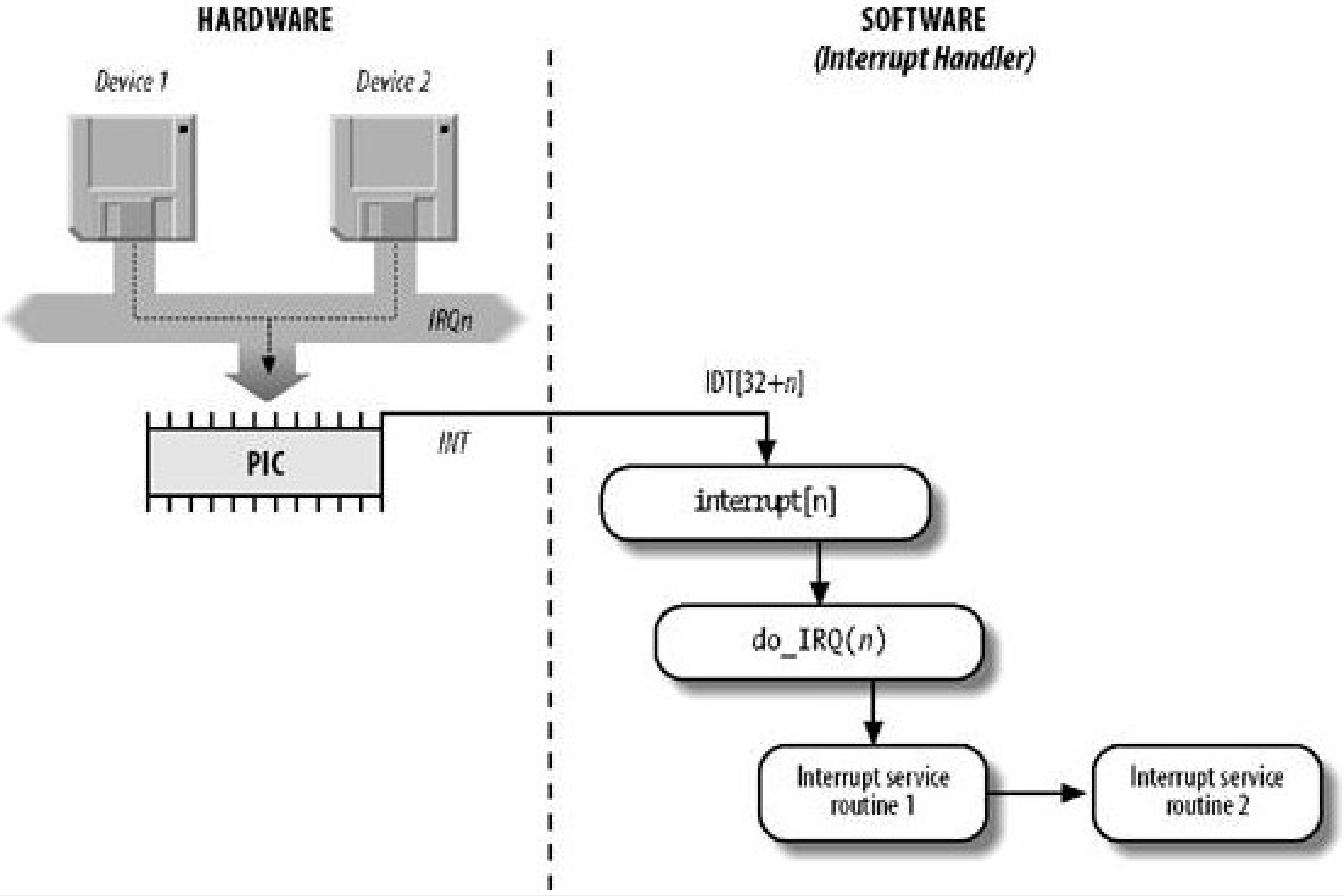
The kernel must discover which I/O device corresponds to the IRQ number before enabling interrupts. Otherwise, for example, how could the kernel handle a signal from a SCSI disk without knowing which vector corresponds to the device? The correspondence is established while initializing each device driver.

Basic actions performed by I/O interrupt handlers

- Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:
  - **Save** the IRQ value and the register's contents on the Kernel Mode stack.
  - Send an **acknowledgment** to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
  - **Execute** the interrupt service routines (ISRs) associated with all the devices that share the IRQ.
  - **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.

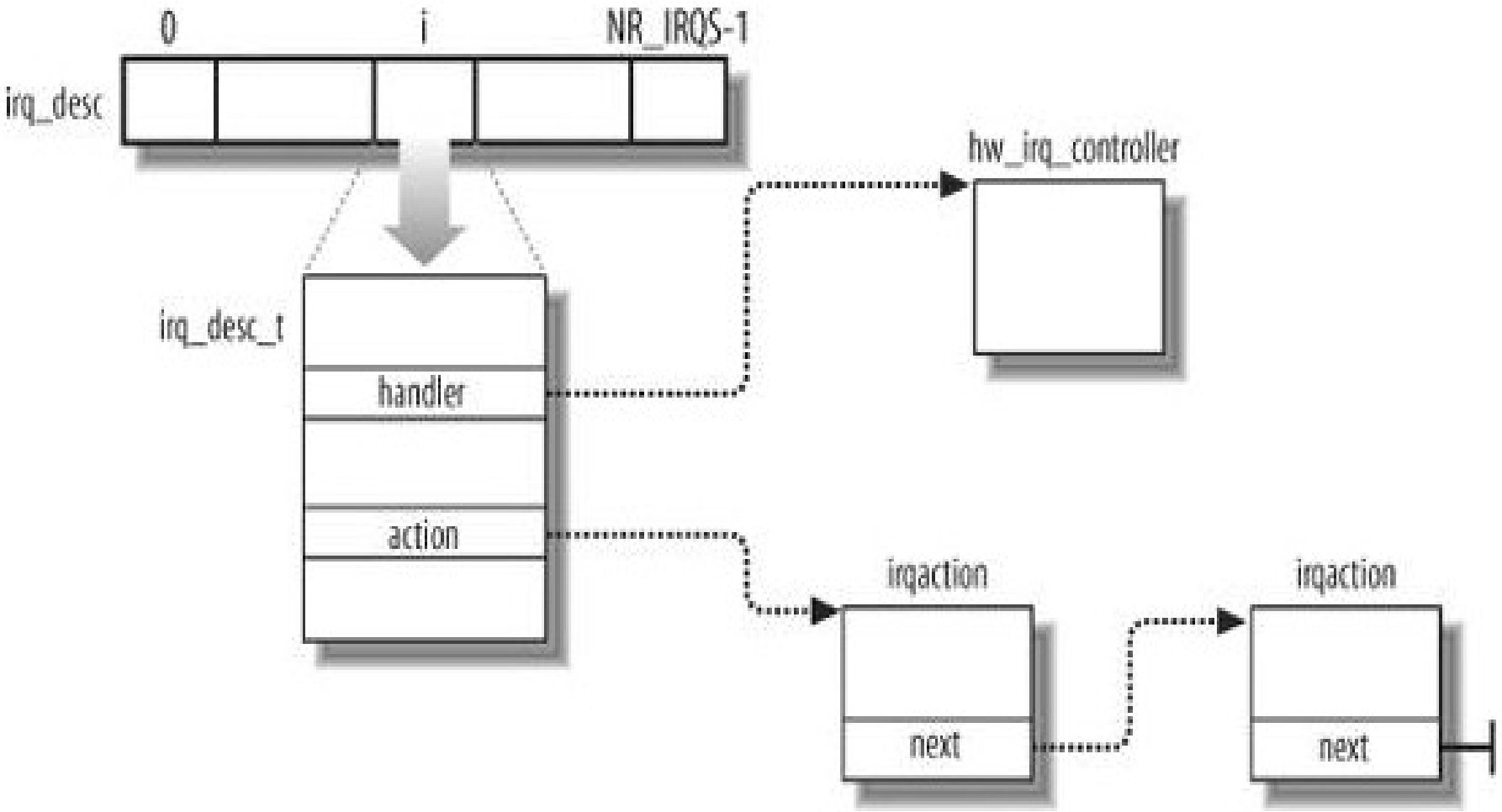
hardware circuits and the software functions used to

Figure 4-4. I/O interrupt handling



4.6.1.2. IRQ data structures

Figure 4-5. IRQ descriptors



PIC object

- 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).

Example: i8259A_irq_type

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
    };
```

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.

Configuring dynamic distribution of interrupts among CPUs

- 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.

- 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.**

4.7. 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.

Softirqs and tasklets

- 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.
- 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.
- Another widely used term is **interrupt context** : it specifies that the kernel is currently **executing either an interrupt handler or a deferrable function**.
- Some differences:
  - 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.

4.7.1. Softirqs

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
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.

Checking for pending softirqs

- 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
  - 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)

4.7.1.5. 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( );
    }
}
```