中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核
分析与安全

5. 中断与系统调用

授课教师：游伟 副教授

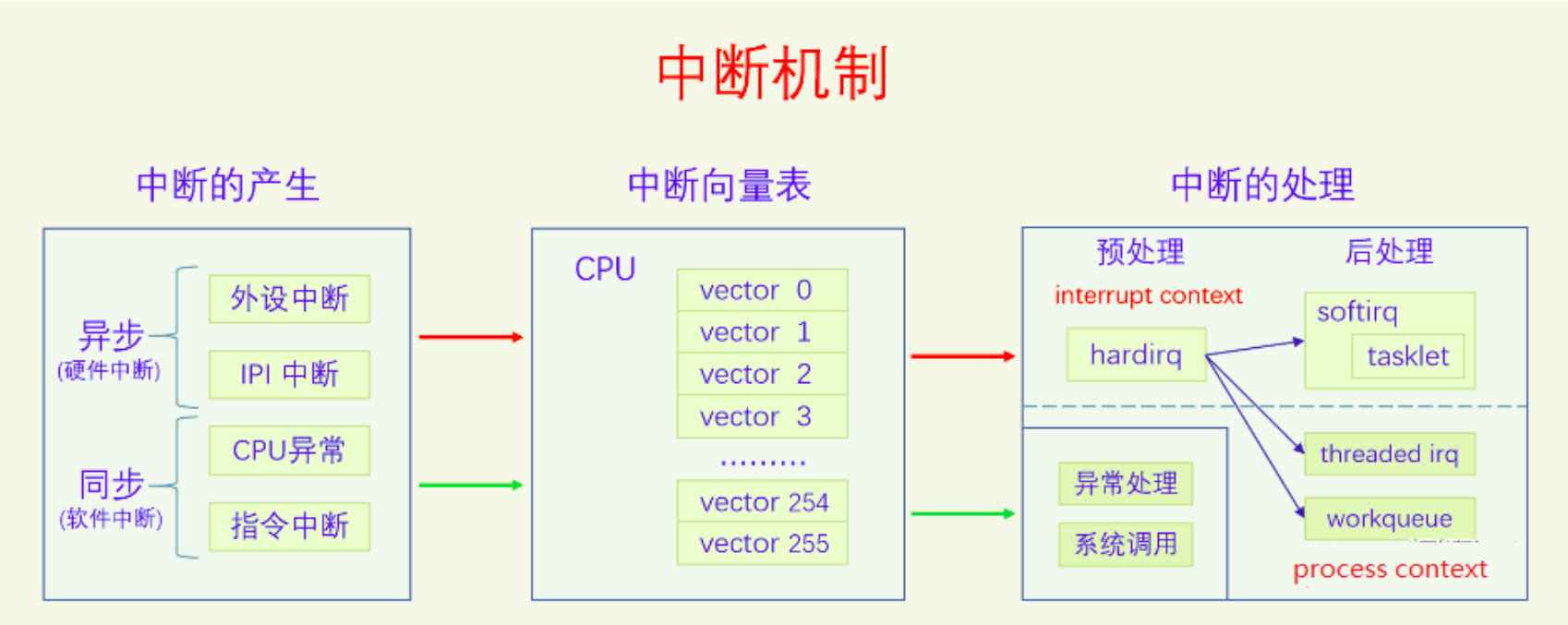
授课时间：周二14:00 – 15:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

目录

5.1 基本概念

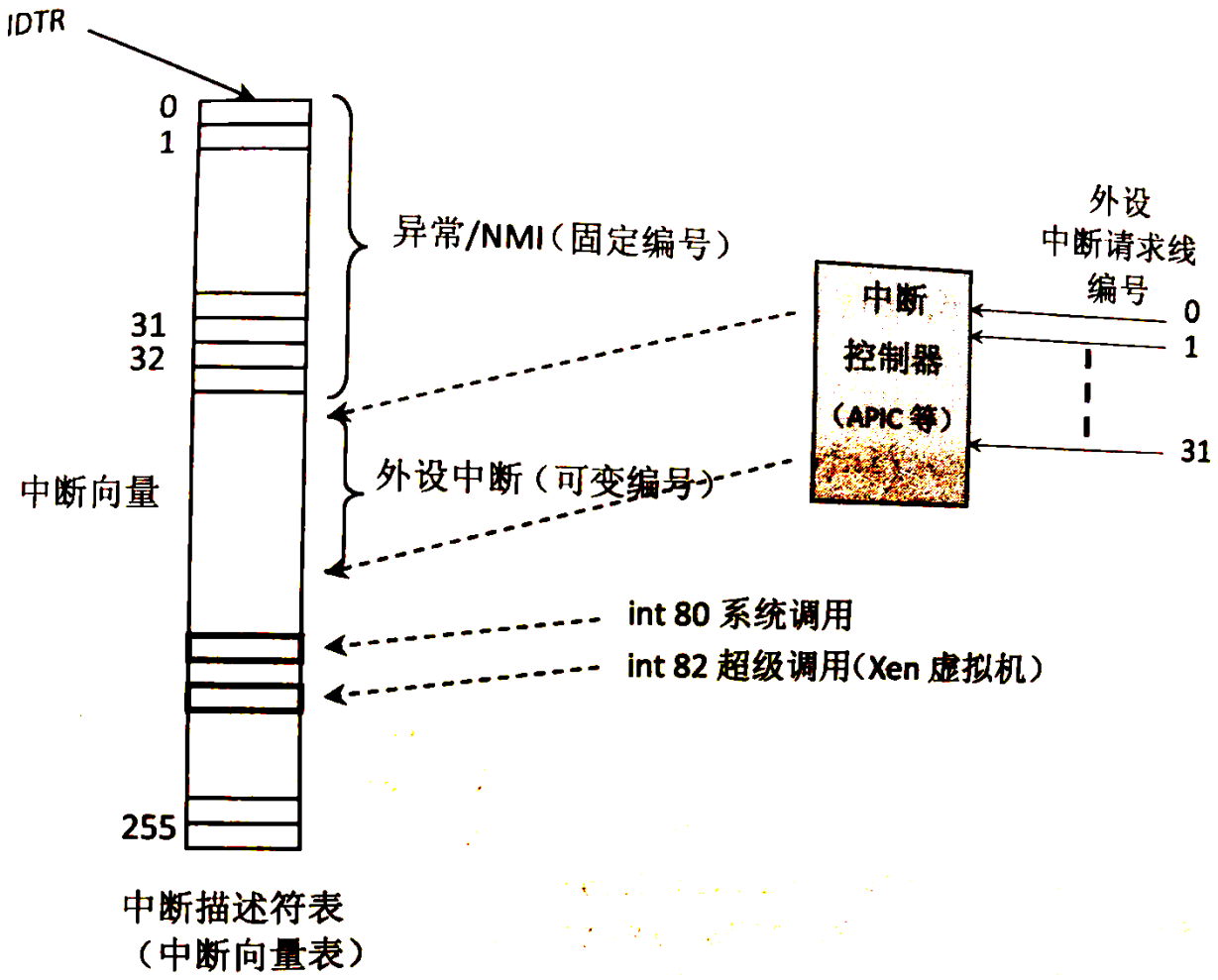
- CPU在执行指令时，收到某个中断信号转而去执行预先设定好的代码，然后再返回到原指令流中继续执行，这就是中断机制
- 操作系统是中断驱动的，中断的作用包括：外设异步通知CPU、CPU之间发送消息、处理CPU异常、实现系统调用



中断的产生

- 外部中断（硬件中断、异步中断）
  - 来源1：外设。外设产生的中断信号是异步的，一般也叫做硬件中断。硬件中断按照是否可以屏蔽分为可屏蔽中断和不可屏蔽中断。例如，网卡、磁盘、定时器都可以产生硬件中断
  - 来源2：CPU。一个CPU向另一个CPU发送中断，这种中断叫做IPI(处理器间中断)。IPI也可以看出是一种特殊的硬件中断，因为它和硬件中断的模式差不多，都是异步的
- 内部中断（软件中断、同步中断）
  - 来源1：CPU异常。 CPU在执行指令的过程中发现异常会向自己发送中断信号，这种中断是同步的，一般也叫做软件中断。
  - 来源2：中断指令，直接用CPU指令来产生中断信号，这种中断和CPU异常一样是同步的，也可以叫做软件中断。

中断向量表


IDTR
0
1
31
32
255
中断向量
异常/NMI（固定编号）
外设中断（可变编号）
中断控制器（APIC 等）
外设中断请求线编号
0
1
31
int 80 系统调用
int 82 超级调用（Xen 虚拟机）
中断描述符表
（中断向量表）

中断向量表

向量号	助记符	含义	类型	错误码	产生原因
0	#DE	Divide Error	故障	无	DIV and IDIV instructions
1	#DB	Debug Exception	故障/陷阱	无	Instruction, data, and I/O breakpoints; single-step; and others.
2	无	NMI Interrupt	中断	无	Nonmaskable external interrupt.
3	#BP	Breakpoint	陷阱	无	INT3 instruction.
4	#OF	Overflow	陷阱	无	INTO instruction.
5	#BR	BOUND Range Exceeded	故障	无	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	故障	无	UD instruction or reserved opcode.
7	#NM	No Math Coprocessor	故障	无	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	中止	有	Any instruction that can generate an exception, an NMI, or an INTR.
9	无	Coprocessor Segment Overrun(reserved)	故障	无	Floating-point instruction.
10	#TS	Invalid TSS	故障	有	Task switch or TSS access.
11	#NP	Segment Not Present	故障	有	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	故障	有	Stack operations and SS register loads.
13	#GP	General Protection	故障	有	Any memory reference and other protection checks.
14	#PF	Page Fault	故障	有	Any memory reference.
15	无	(Intel reserved. Do not use.)		无	
16	#MF	x87 FPU Floating-Point Error	故障	无	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	故障	有	Any data reference in memory
18	#MC	Machine Check	中止	无	Error codes (if any) and source are model dependent.
19	#XM	SIMD Floating-Point Exception	故障	无	SSE/SSE2/SSE3 floating-point instructions
20	#VE	Virtualization Exception	故障	无	EPT violations
21	#CP	Control Protection Exception	故障	有	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception.
22-31	无	Intel reserved. Do not use.			
32-255	无	User Defined(Non-reserved) Interrupts	中断		External interrupt or INT n instruction

中断向量表

```
root@kernel:~# cat /proc/interrupts
```

❶	❷ CPU0	❸	❹
0:	5	XT-PIC	timer
1:	10	XT-PIC	i8042
2:	0	XT-PIC	cascade
4:	1753	XT-PIC	ttyS0
8:	1	XT-PIC	rtc0
9:	0	XT-PIC	acpi
10:	0	XT-PIC	uhci_hcd:usb2, uhci_hcd:usb3, i801_smbus
11:	44	XT-PIC	ehci_hcd:usb1, uhci_hcd:usb4
12:	125	XT-PIC	i8042
24:	1157	PCI-MSI 512000-edge	ahci[0000:00:1f.2]
25:	227	PCI-MSI 32768-edge	eth0-rx-0
26:	33	PCI-MSI 32769-edge	eth0-tx-0
27:	1	PCI-MSI 32770-edge	eth0
28:	0	PCI-MSI 49152-edge	virtio0-config
29:	3	PCI-MSI 49153-edge	virtio0-requests
30:	0	PCI-MSI 65536-edge	virtio1-config
31:	12	PCI-MSI 65537-edge	virtio1-requests
NMI:	0	Non-maskable interrupts	
LOC:	6119	Local timer interrupts	
SPU:	0	Spurious interrupts	
PMI:	0	Performance monitoring interrupts	
IWI:	0	IRQ work interrupts	
RTR:	0	APIC ICR read retries	
RES:	0	Rescheduling interrupts	
CAL:	0	Function call interrupts	
TLB:	0	TLB shootdowns	
TRM:	0	Thermal event interrupts	
THR:	0	Threshold APIC interrupts	
DFR:	0	Deferred Error APIC interrupts	
MCE:	0	Machine check exceptions	
MCP:	0	Machine check polls	
ERR:	0		
MIS:	0		
PIN:	0	Posted-interrupt notification event	
NPI:	0	Nested posted-interrupt event	
PIW:	0	Posted-interrupt wakeup event	

❶：中断线编号

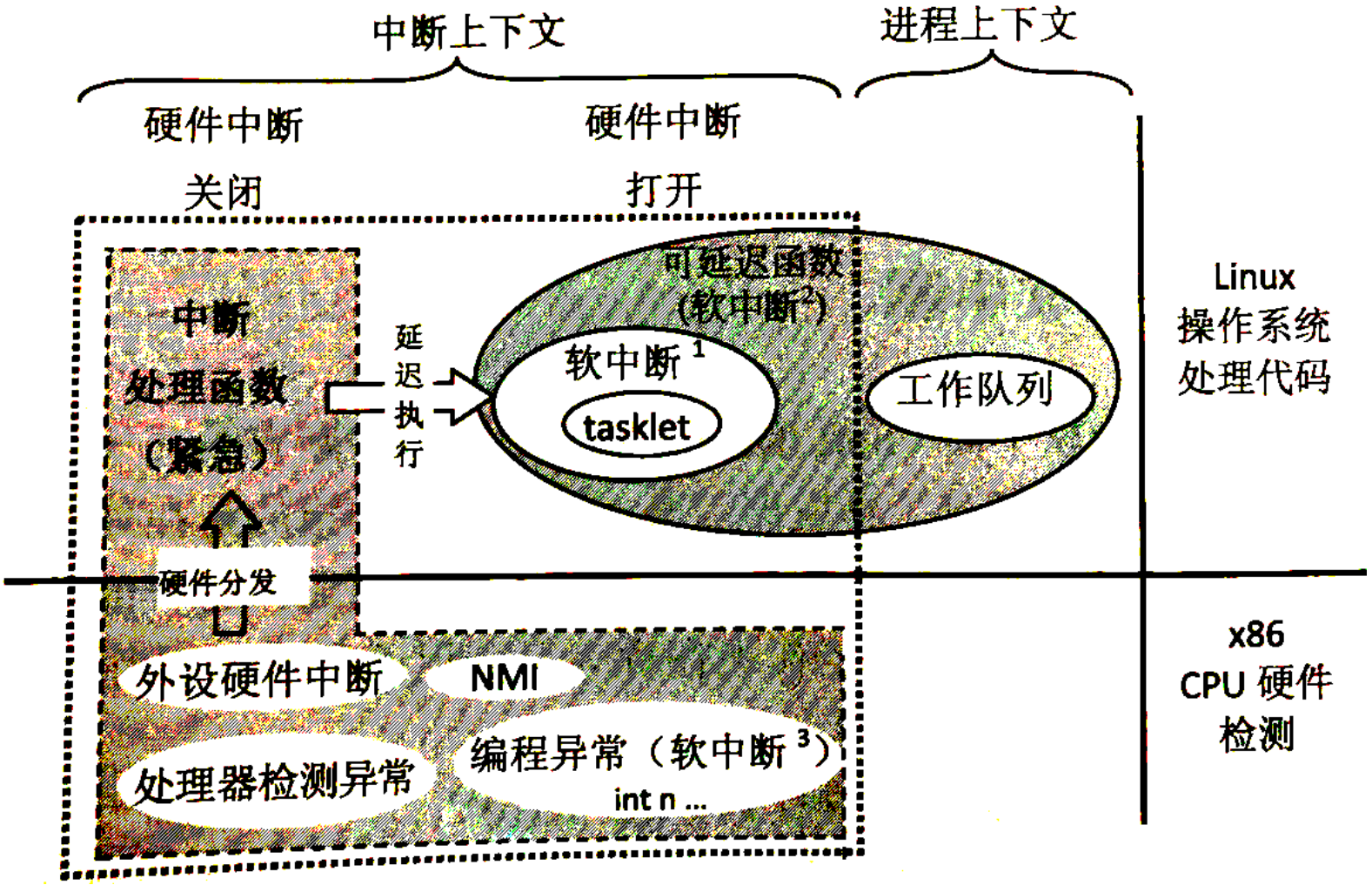
❷：已中断次数

每个CPU一列

❸：中断控制器

❹：设备名称

中断的处理


中断上下文
进程上下文
硬件中断
关闭
硬件中断
打开
中断
处理函数
（紧急）
延迟执行
可延迟函数
（软中断[2]）
软中断[1]
tasklet
工作队列
Linux
操作系统
处理代码
硬件分发
外设硬件中断
NMI
处理器检测异常
编程异常（软中断[3]）
int n ...
x86
CPU 硬件
检测

中断的处理

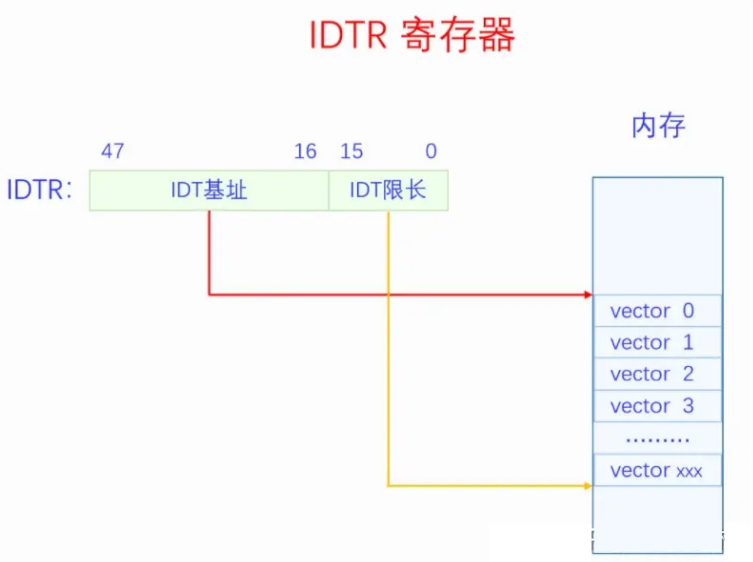
- 执行场景
  - 两种执行场景：进程执行场景、中断执行场景，同步中断的处理是进程执行场景，异步中断的处理是中断执行场景
  - 进程执行场景是可以调度、可以休眠的，而中断执行场景是不可以调度不可用休眠的
  - 进程执行场景中是可以接受中断信号的，而在中断执行场景中是屏蔽中断信号的
- 处理方式
  - 两种处理方式：1. 立即完全处理，2. 立即预处理 + 稍后完全处理；对于处理起来比较耗时的中断，可以采用后一种方式进行处理
  - “立即完全处理” 和 “立即预处理” 采用硬中断（hardirq）方式，整个过程是屏蔽中断的
  - “稍后完全处理” 在处理过程不再屏蔽中断信号，提高了系统对中断的响应性，可以分为两类：直接中断后处理有软中断（softirq）、微任务（tasklet）；线程化中断后处理有工作队列（workqueue）、中断线程（threaded_irq）

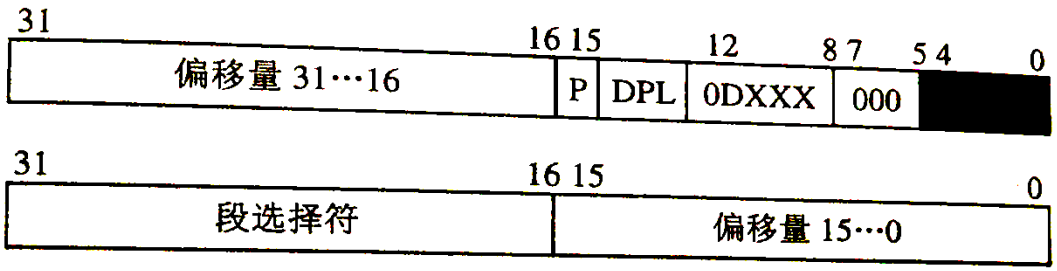
中断的处理

- 保存现场：
  - CPU收到中断信号后会首先把一些数据push到内核栈上，保存的数据是和当前执行点相关的，这样中断完成后就可以返回到原执行点。
  - 如果CPU当前处于用户态，则会先切换到内核态，把用户栈切换为内核栈再去保存数据(内核栈的位置是在当前线程的TSS中获取的)

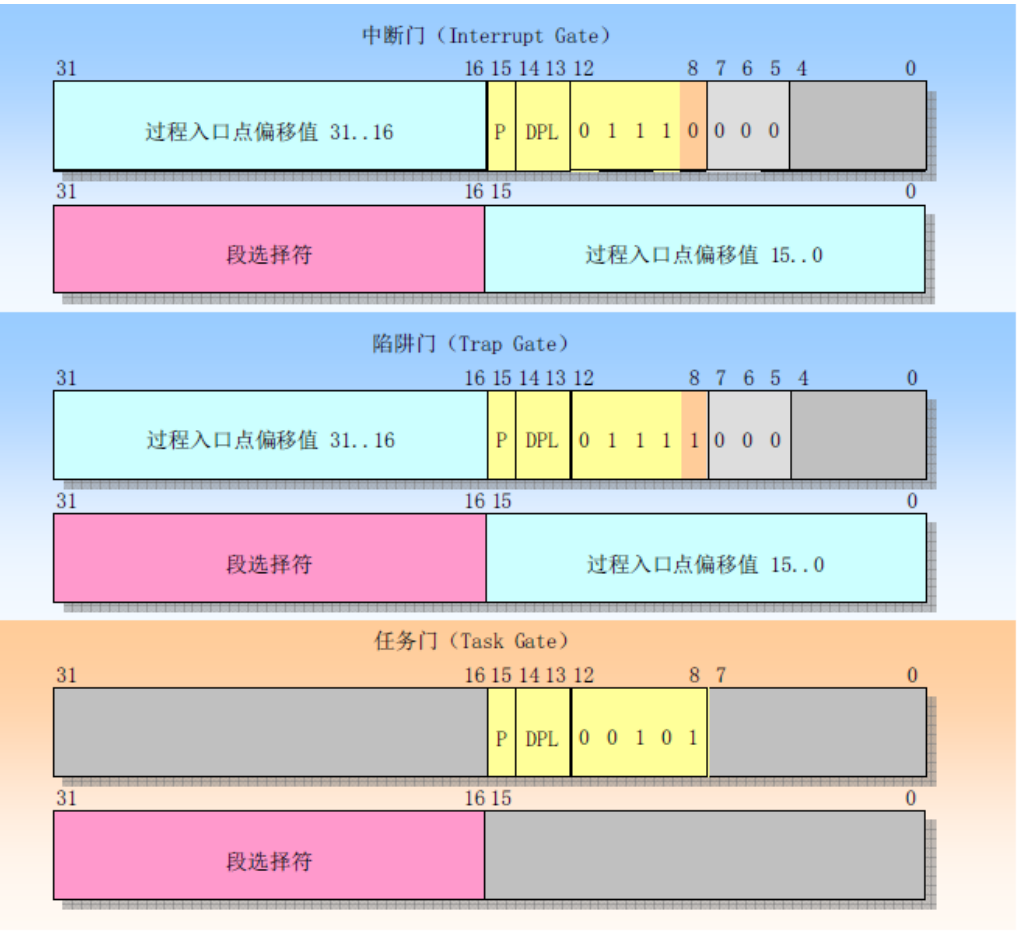
中断的处理

- 查找向量表
  - CPU会根据当前中断信号的向量号去查询中断向量表找到中断处理程序
  - 如何找到中断向量表：IDTR寄存器
  - 如何获得当前中断信号的向量号：如果是CPU异常可以在CPU内部获取，如果是指令中断，在指令中就有向量号，如果是硬件中断，则可以从中断控制器中获取中断向量号





DPL	段描述符的特权级
偏移量	入口函数地址的偏移量
P	段是否在内存中的标志
段选择符	入口函数所处代码段的选择符
D	标志位，1=32 位，0=16 位
XXX	3 位门类型码



P – 段存在标志　　DPL – 描述符特权级

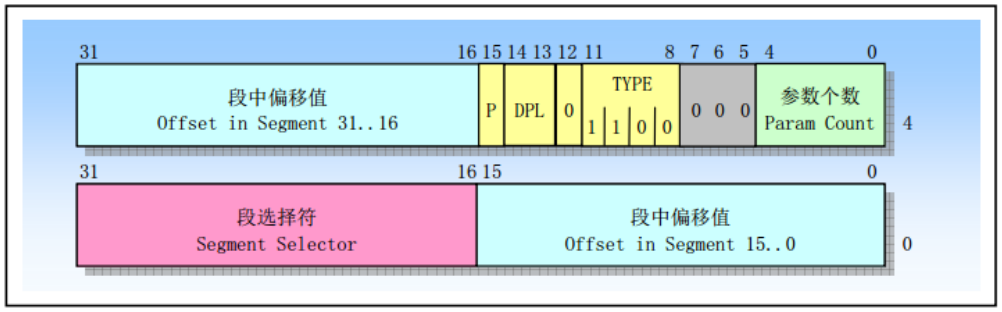
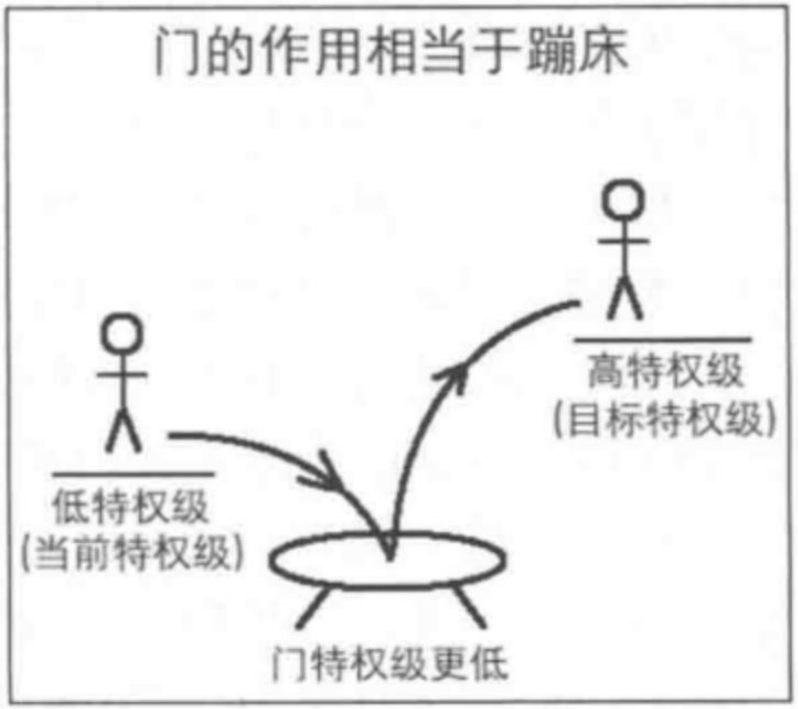


图 4-22 调用门描述符格式

Call Gate	Interrupt Gate	Trap Gate
Called by the instruction CALL and JMP	Called by the instruction INT	Called by the instruction INT
Stored in GDL and LDT	Stored in IDT	Stored in IDT
Has a special feature of transferring the parameters	A special feature is these gates all additionally prohibit future interrupt acceptance	No special feature
Flexibility is very less	Flexibility is more as compared to trap gate	Flexibility is more as compared to Interrupt gate
Very less portable options	More portability options as compared to trap gate	More portability options as compared to Interrupt gate



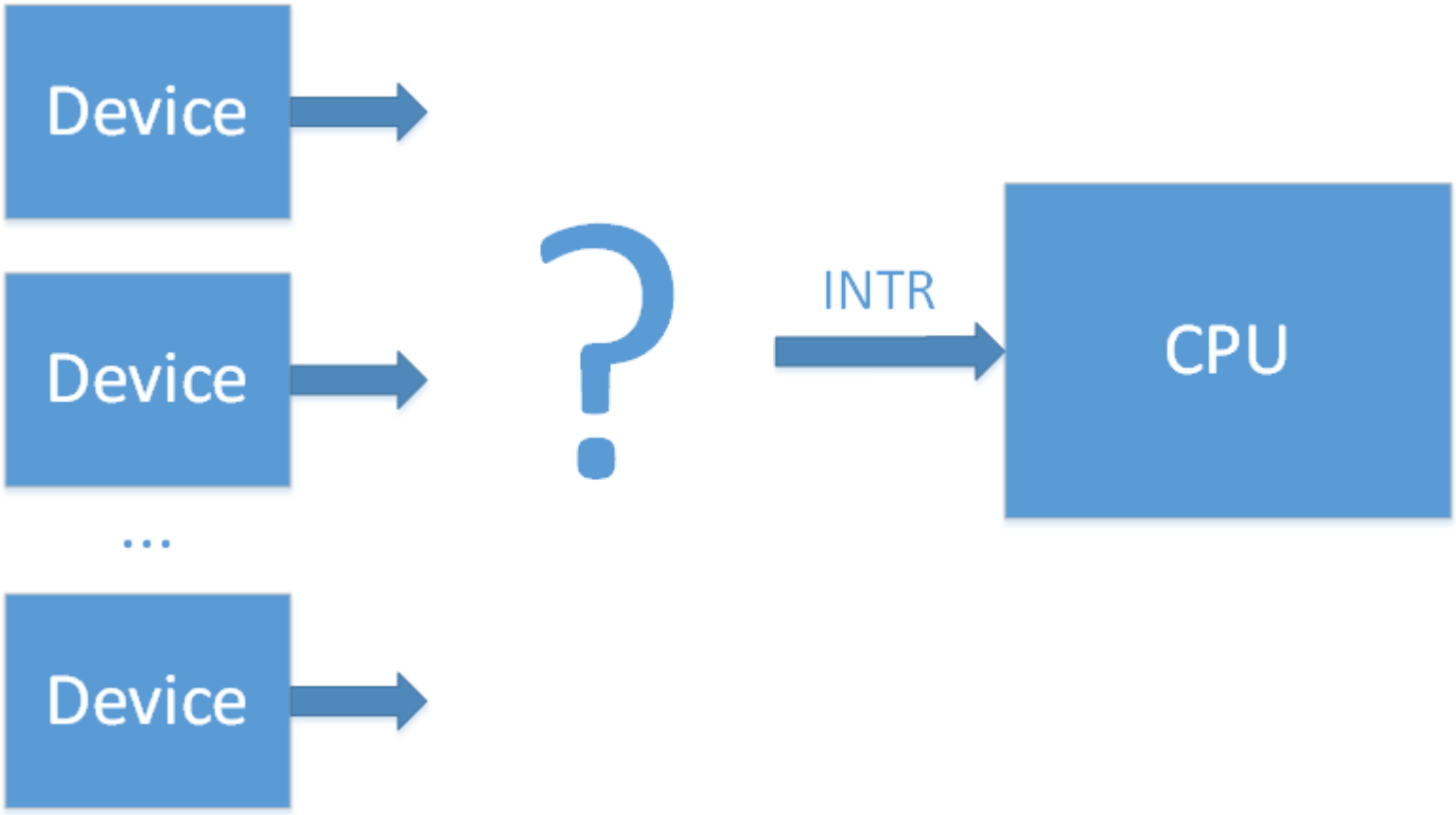
5.2 硬件中断

- 硬件中断定义与作用
  - 硬件中断是外部设备或内部硬件模块通过特定信号线（如 CPU 的 IRQ 引脚）向 CPU 发送的 异步请求，用于通知 CPU 处理紧急事件。
  - **实时性**：避免 CPU 轮询检测外设状态，及时响应突发事件（如按键输入、传感器数据就绪）。
  - **效率**：允许 CPU 在等待硬件操作（如数据传输）时执行其他任务，减少空转浪费。
- 硬件中断的分类
  - 按触发源分类
    - 外部中断：由计算机外设（如键盘、定时器、传感器）通过物理信号触发，例如 GPIO 引脚电平变化或定时器溢出。
    - 内部中断：由 CPU 内部异常触发，例如运算错误（除零、溢出）、内存访问违规或电源故障。
  - 按可屏蔽性分类
    - 可屏蔽中断（Maskable Interrupt）：可通过中断屏蔽寄存器（如 EFLAGS.IF 位）临时禁用，例如普通外设中断。
    - 不可屏蔽中断（NMI, Non-Maskable Interrupt）：优先级最高，无法通过软件屏蔽，用于处理严重硬件故障（如内存校验错、电源掉电）。

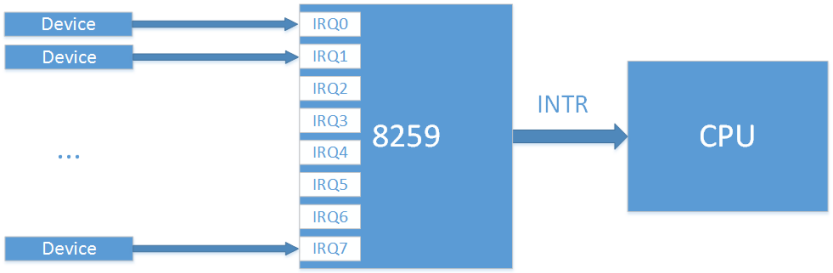
5.2 硬件中断

- 硬件中断的触发方式
  - 电平触发（Level Trigger）：持续检测中断信号线电平（如高电平有效），适用于需要长时间响应的场景（如 DMA 传输）。
  - 边沿触发（Edge Trigger）：仅在信号跳变（上升沿或下降沿）时触发，适用于瞬时事件（如按键按下）。
- 硬件中断处理流程
  1. 中断请求（IRQ）：外设通过中断控制器（如 8259A 或 NVIC）向 CPU 发送请求信号。
  2. 中断响应：CPU 完成当前指令后，检测中断优先级，若未屏蔽则进入中断响应周期。
  3. 保存上下文：CPU 自动将 程序计数器（PC）、状态寄存器（如 EFLAGS） 压入堆栈。
  4. 中断服务例程（ISR）：通过 中断向量表（IVT） 跳转到对应 ISR 入口地址。ISR 需完成外设状态处理（如读取数据）并清除中断标志。
  5. 恢复上下文：通过 IRET 指令恢复寄存器状态，返回原程序继续执行。

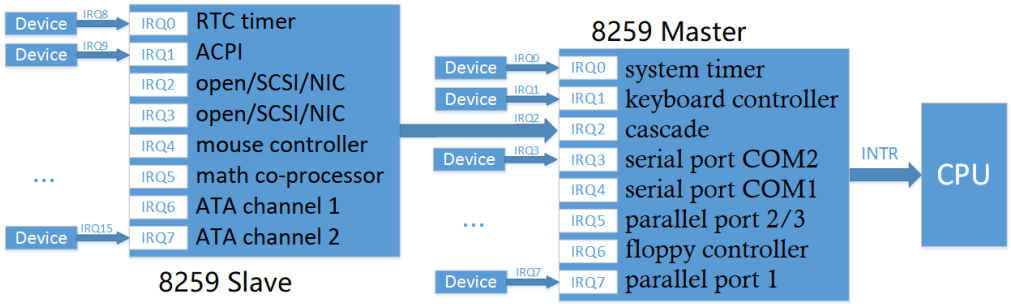
中断控制器



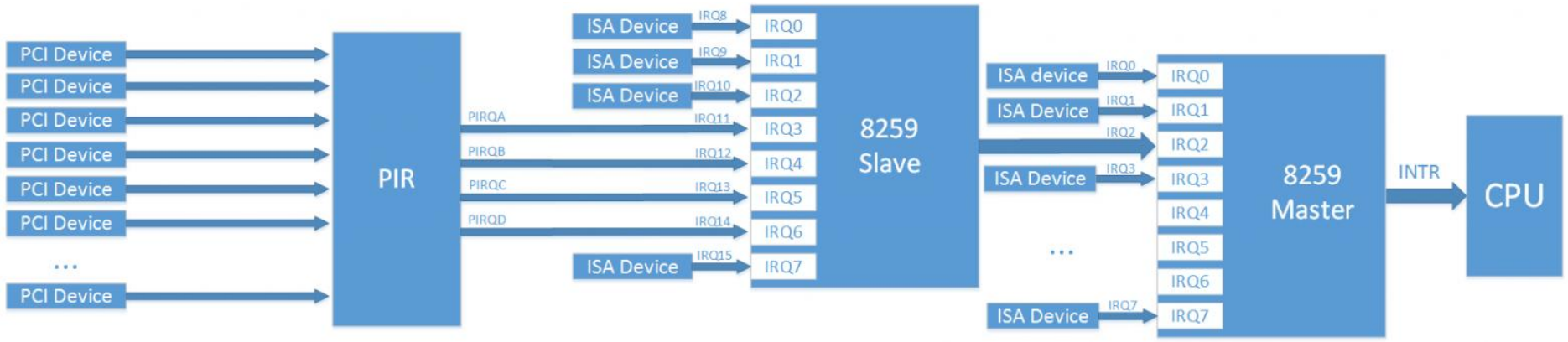
PIC (Programmable Interrupt Controller)



(a) Single PIC



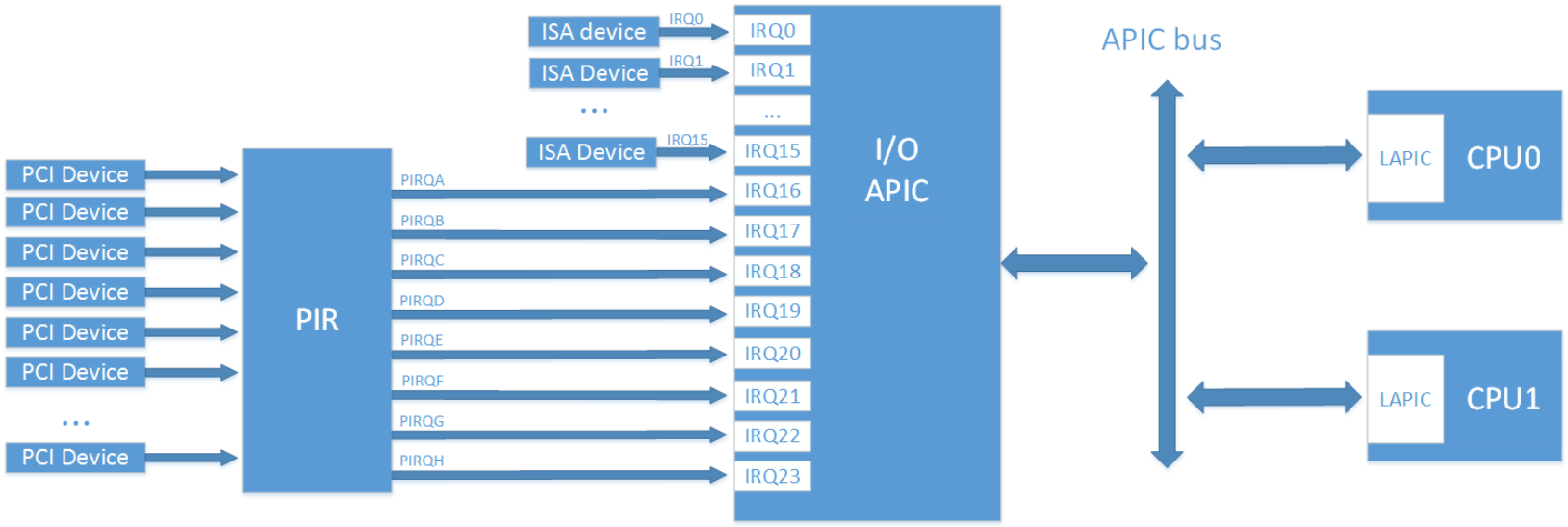
(b) Dual PIC



(c) Dual PIC with PIRQ Lines (PIRQ: Programmable Interrupt Request)

APIC（Advanced PIC）

- APIC 分成两部分：LAPIC 位于每个 CPU 内部，IOAPIC 与外设相连
- 外设发出的中断信号经过 IOAPIC 处理之后发送某个或多个 LAPIC，再由 LAPIC 决定是否交由 CPU 进行实际的中断处理



MSI（Message Signaled Interrupts）

- MSI(Message Signaled Interrupts)是一种中断方式, 依靠设备将一小段中断描述数据写入特定地址来通知CPU中断的产生。

注册中断处理程序

- 中断处理程序是驱动程序的组成部分。
- 驱动程序通过以下函数注册并激活一个中断处理程序。

```
int request_irq (unsigned int irq,
        irqreturn_t (*handler) (int, void *, struct pt_regs *),
        unsigned long irqflags,
        const char *devname,
        void *dev_id)
```

request_irq( )的参数

- 第1个参数irq：要分配的中断号。
- 第2个参数handler：一个指针，指向处理这个中断的实际中断处理程序。
- 第3个参数irqflags：可以为0，也可以是SA_INTERRUPT, SA_SAMPLE_ RANDOM, SA_SHIRQ其中的一个或多个标志的位掩码。
- 第4个参数devname：与中断相关的设备的ASCII文本表示法。
- 第5个参数dev_id：主要用于共享中断线。

request_irq( )的返回值

- 若成功执行，返回0；
  如果返回非0值，表示有错误发生，在这种情况下，指定的中断处理程序不会被注册。

- **注意：** request_irq( )函数可能会睡眠，因此，不能在中断上下文或其它不允许阻塞的代码中调用该函数。

使用request_irq( )函数

```
if (request_irq (irqn, my_interrupt,
                 SA_SHIRQ, “my_device”, dev) )
{
        printk(KERN_ERR “my_device:
                cannot register IRQ %d, \n”, irqn);
        return –EIO;
}
```

释放中断处理程序

- 卸载驱动程序时，需要注销相应的中断处理程序，并释放中断线。
- 可以调用void free_irq (unsigned int irq, void *dev_id)来释放中断线。
- 如果指定的中断线不是共享的，那么，该函数删除处理程序的同时将禁用这条中断线。
  如果中断线是共享的，则仅删除dev_id所对应的处理程序，而这条中断线本身只有在删除了最后一个处理程序时才会被禁用。

编写中断处理程序

- 典型的中断处理程序声明：

```
static irqreturn_t intr_handler (int irq, void *dev_id,
        struct pt_regs *regs)
```

- 其返回值为irqreturn_t型，实际上是int型。
- 中断处理程序可能返回两个特殊的值：
  - IRQ_NONE：当中断处理程序检测到一个中断，但该中断对应的设备并不是在注册处理函数期间指定的产生源时，返回该值；
  - IRQ_HANDLED：当中断处理程序被正确调用，且确实是它所对应的设备产生了中断时，返回该值。

- **在include/linux/interrupt.h中有如下几行，定义了中断处理程序的返回值：**

```
typedef int irqreturn_t;

#define IRQ_NONE        (0)
#define IRQ_HANDLED     (1)
#define IRQ_RETVAL(x)   ((x) != 0)
```

典型中断处理程序的参数说明

- **irq**：是这个处理程序要响应的中断的中断线号。现在，该参数已没有太大用处。
- **dev_id**：一个通用指针，它与传递给request_irq( )的参数dev_id必须一致。如果该值有唯一确定性，那么它就相当于一个cookie，可以用来区分共享同一中断处理程序的多个设备。
- **regs**：一个指向结构的指针，该结构包含处理中断之前处理器的寄存器和状态。除了调试，很少用到它。

中断处理程序和重入

- 当一个给定的中断处理程序正在执行时，相应的中断线在所有处理器上都会被屏蔽掉，以防止在同一中断线上接收到另一个新的中断。因此，同一个中断处理程序绝对不会被同时调用以处理嵌套的中断。
- 所以，Linux的中断处理程序是无需重入的。
  这极大地简化了中断处理程序的编写。

共享的中断处理程序

- 共享的处理程序与非共享的处理程序在注册和运行方式上比较相似，但差异主要有三点：
  - request_irq( )的参数irqflags必须设置SA_SHIRQ标志。
  - 对每个注册的中断处理程序来说，dev_id参数必须唯一。指向任一设备结构的指针就可以满足这一要求。
  - 中断处理程序必须能够区分它的设备是否真的产生了中断。

中断处理程序实例

- 例：来自RTC (real_time clock)驱动程序的一个实际的中断处理程序，它在drivers/char/rtc.c中定义。
- 在RTC驱动程序装载时，会调用rtc_init( )函数，进行初始化。
- 在rtc_init( )函数中，将注册中断处理程序，代码见下页：

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

- 其中的第2个参数是中断处理程序rtc_interrupt，其代码见下页。

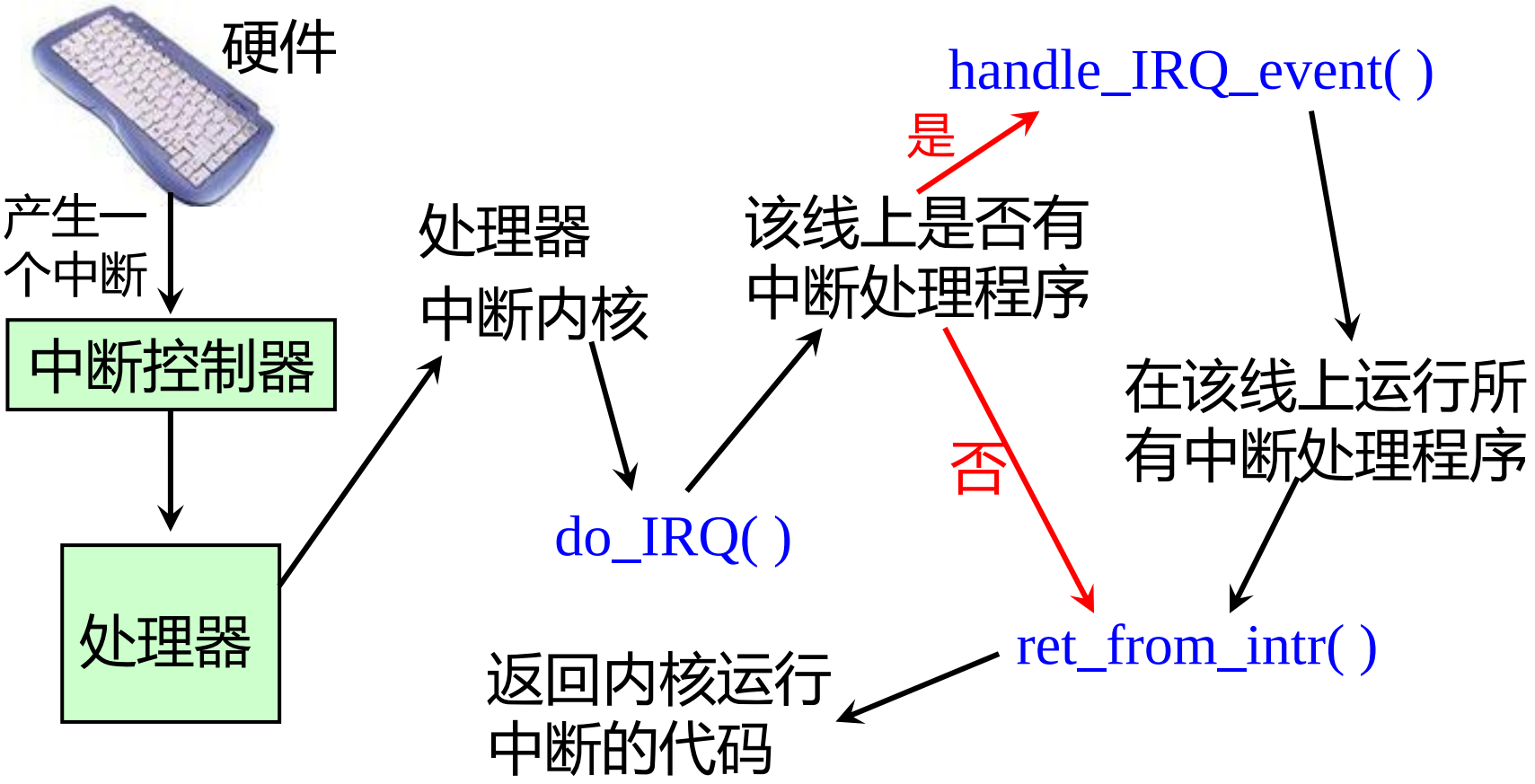
中断处理程序rtc_interrupt( )

```
static unsigned long rtc_irq_data = 0;

irqreturn_t rtc_interrupt (int irq, void *dev_id,
                        struct pt_regs *regs)
{
    spin_lock (&rtc_lock);
    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);
    if (rtc_status & RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer,
                jiffies + HZ/rtc_freq + 2*HZ/100);
```

rtc_interrupt( ) (续)

```
    spin_unlock (&rtc_lock);
    /* Now do the rest of the actions */
    spin_lock(&rtc_task_lock);
    if (rtc_callback)
        rtc_callback->func(rtc_callback->private_data);
    spin_unlock(&rtc_task_lock);
    wake_up_interruptible(&rtc_wait);

    kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

    return IRQ_HANDLED;
}
```

中断从硬件到内核的路由



- 在内核中，中断的旅程开始于预定义入口点，对于每条中断线，处理器都会跳到对应的一个唯一位置。这样，内核就可以知道所接收中断的IRQ号了。
- 然后，内核调用do_IRQ( )函数。

do_IRQ( )函数

```
unsigned int do_IRQ(struct pt_regs *regs)
{
        int irq = regs->orig_eax & 0xff;
        ……

        return 1;
}
```

- **通过orig_eax读出堆栈中的值，并将高位屏蔽掉，得到IRQ号。**

补充：pt_regs结构

```
struct pt_regs {
        long ebx;       long ecx;
        long edx;       long esi;
        long edi;       long ebp;
        long eax;       int  xds;
        int  xes;       long orig_eax;
        long eip;       int  xcs;
        long eflags;                long esp;
        int  xss;
};
```

do_IRQ( ) ▸> handle_IRQ_event( )

```
int handle_IRQ_event (unsigned int irq, struct pt_regs * regs,
                struct irqaction * action) {
        int status= 1;
        int retval = 0;
        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable( );
        do {
                status |= action->flags;
                retval |= action->handler(irq, action->dev_id, regs);
                action = action->next;
        } while (action);
        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);
        local_irq_disable( );
        return status;
}
```

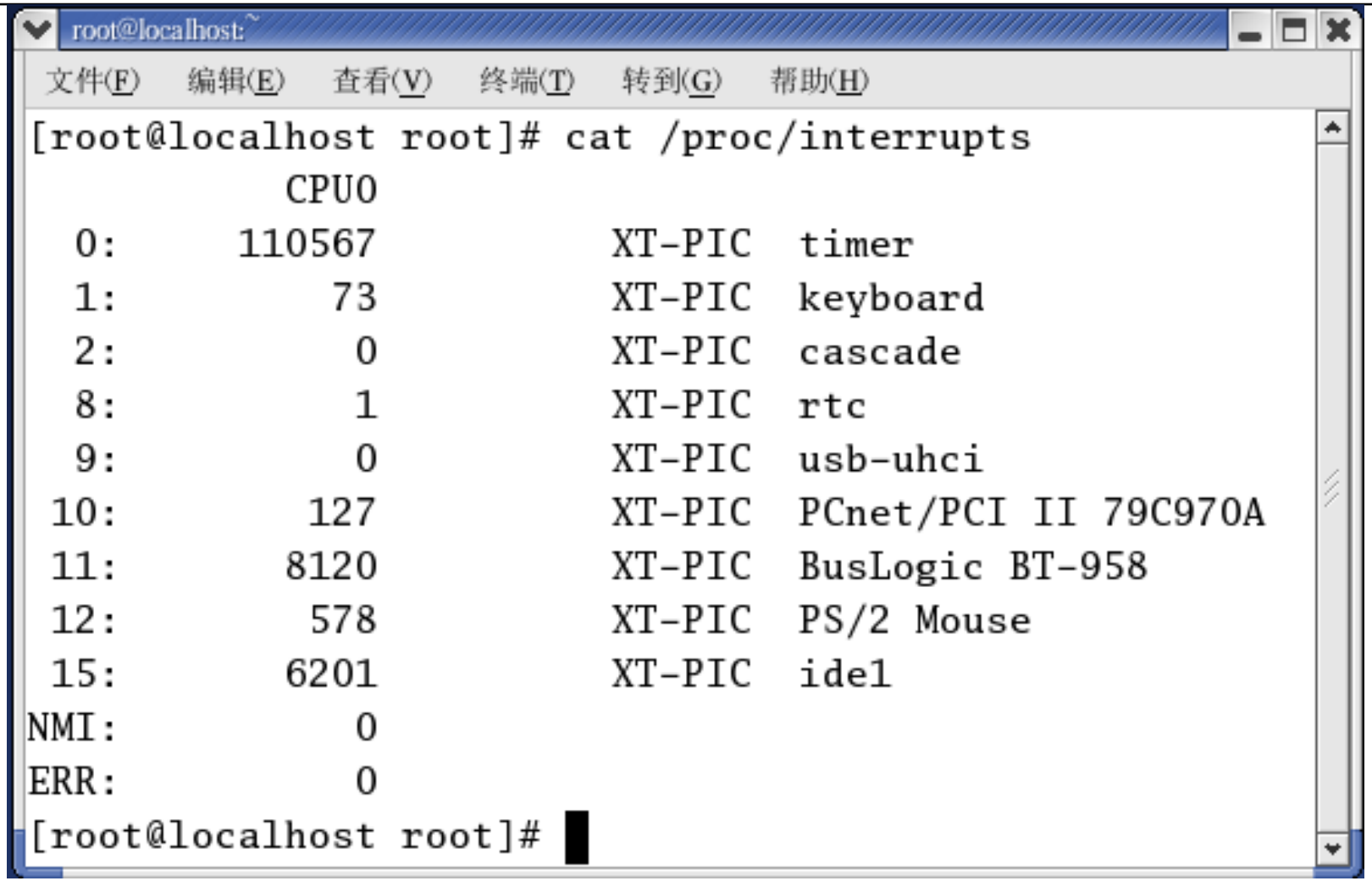
补充：irqaction数据结构

```
struct irqaction {
    irqreturn_t (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    cpumask_t mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
    int irq;
    struct proc_dir_entry *dir;
};
```

/proc/interrupts

- procfs是一个虚拟文件系统，它只存在于内核内存，一般安装于/proc目录下。
- /proc/interrupts文件：存放系统中与中断相关的统计信息。见下页。

/proc/interrupts文件的内容



中断控制

- Linux内核提供了一组接口用于操作机器上的中断状态。这些接口为我们提供了能够禁止当前处理器的中断系统、或屏蔽掉整个机器的一条中断线的能力。

禁止和激活中断

- **用于禁止当前处理器上的本地中断，随后又激活它们的语句为：**

```
local_irq_disable( )；
/*禁止中断……*/
local_irq_enable( )；
```

- **这两个函数通常以单个汇编指令实现，其代码在include/asm-i386/system.h中，见下页：**

```
#define local_irq_disable( )                    \
        __asm__ __volatile__("cli": : :"memory")
#define local_irq_enable( )                     \
        __asm__ __volatile__("sti": : :"memory")
```

- **如果在调用local_irq_disable( )之前已经禁止了中断，那么该例程往往会带来潜在的危险。同样local_irq_enable( )也存在潜在危险。**
- **因此需要一种机制把中断恢复到以前的状态而不是简单地禁止或激活。**

- **在禁止中断之前应该保存中断系统的状态；而在准备激活中断时，只需要把中断恢复到它们原来的状态。如下所示：**

```
unsigned long flags;
local_irq_save(flags);
/*……*/
local_irq_restore(flags);
```

- **对local_irq_save( )的调用和对local_irq_restore( )的调用必须在同一个函数中进行。**

禁止指定中断线

- 在某些情况下，只禁止整个系统中一条特定的中断线就够了。这就是所谓的屏蔽掉一条中断线。
- Linux提供了四个接口：
  - void disable_irq(unsigned int irq);
  - void disable_irq_nosync(unsigned int irq);
  - void enable_irq(unsigned int irq);
  - void synchronize_irq(unsigned int irq);

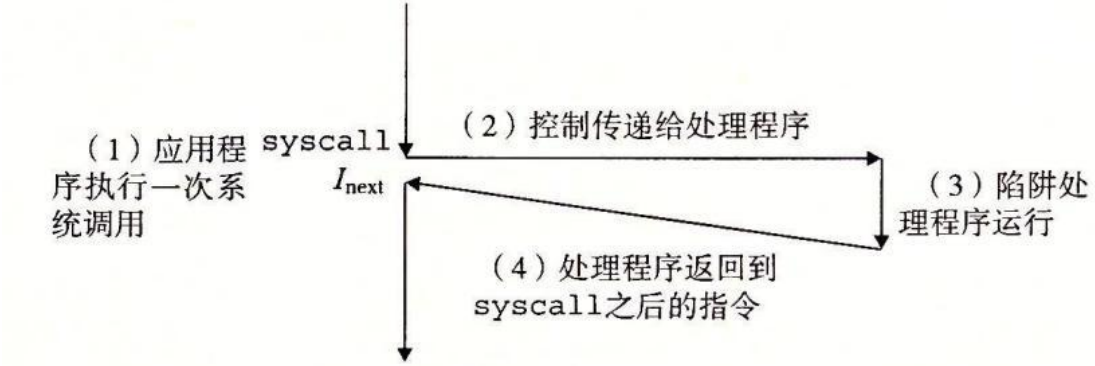
中断系统的状态

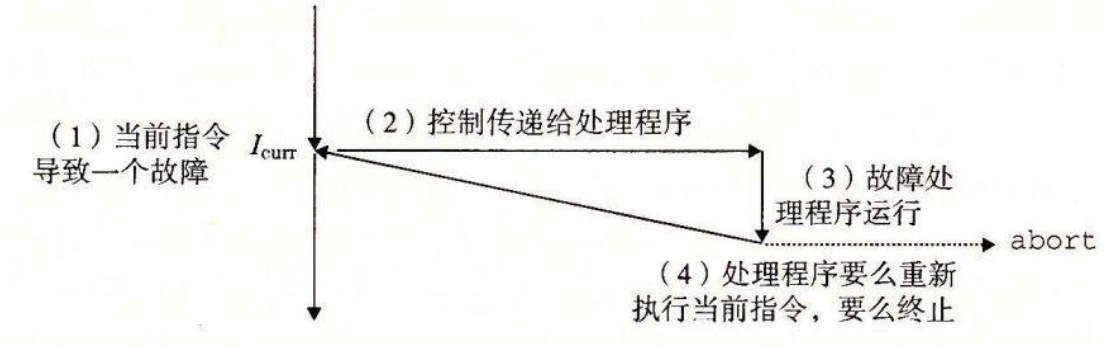
- 宏irqs_disabled( )：如果本地处理器上的中断系统被禁止，则它返回非0，否则返回0。
- 宏in_interrupt( )：如果内核处于中断上下文中，返回非0。说明内核此时正在执行中断处理程序，或者正在执行下半部处理程序。
- 宏in_irq( )：只有在内核确实正在执行中断处理程序时才返回非0。

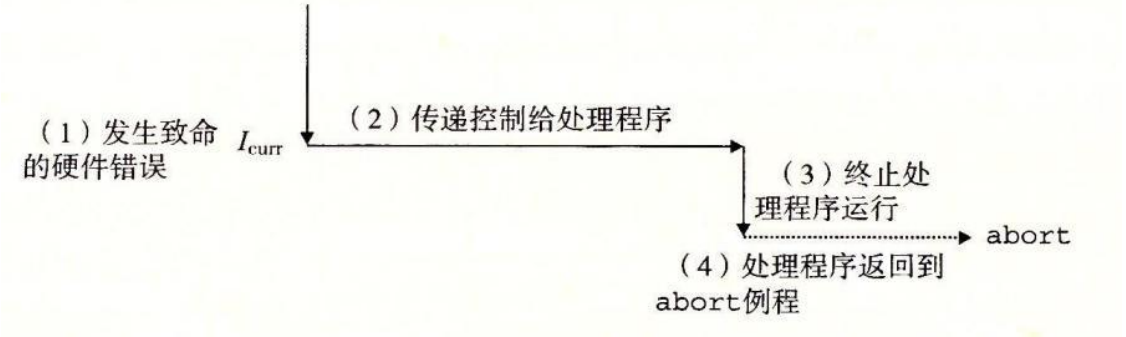
5.3 软件中断

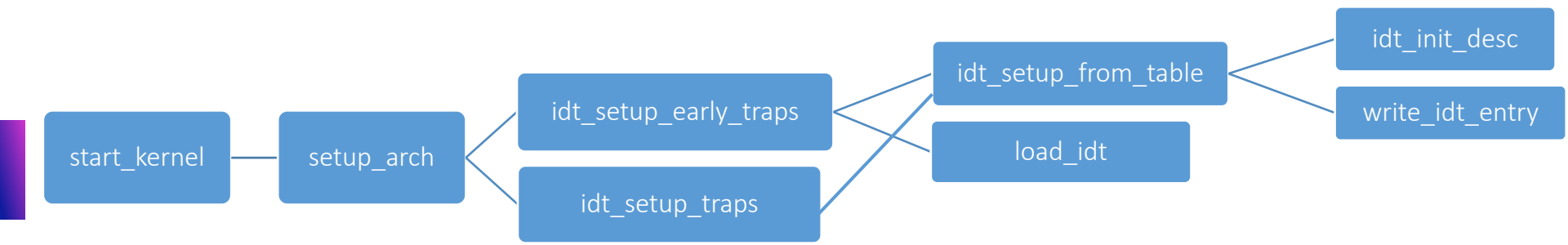
- CPU异常：CPU在执行指令的过程中遇到了异常就会给自己发送中断信号，CPU异常分为以下3类：
  - 陷阱(trap)：陷阱并不是错误，而是想要陷入内核来执行一些操作，中断处理完成后继续执行之前的下一条指令
  - 故障(fault)：程序遇到了问题需要修复，若能修复，则处理完成后会重新执行之前的指令，若无法修复那就是错误，当前进程将会被杀死
  - 中止(abort)：系统遇到了很严重的错误，无法修改，一般系统会崩溃
- 指令中断：执行特定的指令而发生的中断
  - CPU异常是在执行某些指令产生例外时发生的，而指令中断则是在执行特定指令时必然发生的
  - 指令INT n可以产生任意中断，Linux用int 0x80作为系统功能调用的指令
  - CPU把指令中断当作"陷阱"来处理

5.3 软件中断


（1）应用程序执行一次系统调用
syscall
I_{next}
（2）控制传递给处理程序
（3）陷阱处理程序运行
（4）处理程序返回到syscall之后的指令


（1）当前指令导致一个故障
I_{curr}
（2）控制传递给处理程序
（3）故障处理程序运行
abort
（4）处理程序要么重新执行当前指令，要么终止


（1）发生致命的硬件错误
I_{curr}
（2）传递控制给处理程序
（3）终止处理程序运行
abort
（4）处理程序返回到abort例程




```c
#define IDT_ENTRIES		256

typedef struct gate_struct {
	u16	offset_low;
	u16	segment;
	struct idt_bits bits;
	u16	offset_middle;
} gate_desc;

gate_desc idt_table[IDT_ENTRIES] __page_aligned_bss;

void __init idt_setup_early_traps(void)
{
	idt_setup_from_table(idt_table, early_idts, ARRAY_SIZE(early_idts), true);
	load_idt(&idt_descr);
}

void __init idt_setup_traps(void)
{
	idt_setup_from_table(idt_table, def_idts, ARRAY_SIZE(def_idts), true);
}

static void
idt_setup_from_table(gate_desc *idt, const struct idt_data *t, int size, bool sys)
{
	gate_desc desc;

	for (; size > 0; t++, size--) {
		idt_init_desc(&desc, t);
		write_idt_entry(idt, t->vector, &desc);
		if (sys)
			set_bit(t->vector, system_vectors);
	}
}

static inline void idt_init_desc(gate_desc *gate, const struct idt_data *d)
{
	unsigned long addr = (unsigned long) d->addr;

	gate->offset_low	= (u16) addr;
	gate->segment		= (u16) d->segment;
	gate->bits		= d->bits;
	gate->offset_middle	= (u16) (addr >> 16);
}

static inline void write_idt_entry(gate_desc *idt, int entry, const gate_desc *gate)
{
	memcpy(&idt[entry], gate, sizeof(*gate));
}
```

```c
#define G(_vector, _addr, _ist, _type, _dpl, _segment)	\
	{						\
		.vector		= _vector,		\
		.bits.ist	= _ist,			\
		.bits.type	= _type,		\
		.bits.dpl	= _dpl,			\
		.bits.p		= 1,			\
		.addr		= _addr,		\
		.segment	= _segment,		\
	}

/* Interrupt gate */
#define INTG(_vector, _addr)				\
	G(_vector, _addr, DEFAULT_STACK, GATE_INTERRUPT, DPL0, __KERNEL_CS)

/* System interrupt gate */
#define SYSG(_vector, _addr)				\
	G(_vector, _addr, DEFAULT_STACK, GATE_INTERRUPT, DPL3, __KERNEL_CS)

/* Task gate */
#define TSKG(_vector, _gdt)				\
	G(_vector, NULL, DEFAULT_STACK, GATE_TASK, DPL0, _gdt << 3)


static const __initconst struct idt_data early_idts[] = {
	INTG(X86_TRAP_DB,		debug),
	SYSG(X86_TRAP_BP,		int3),
	INTG(X86_TRAP_PF,		page_fault),
};

static const __initconst struct idt_data def_idts[] = {
	INTG(X86_TRAP_DE,		divide_error),
	INTG(X86_TRAP_NMI,		nmi),
	INTG(X86_TRAP_BR,		bounds),
	INTG(X86_TRAP_UD,		invalid_op),
	INTG(X86_TRAP_NM,		device_not_available),
	INTG(X86_TRAP_OLD_MF,		coprocessor_segment_overrun),
	INTG(X86_TRAP_TS,		invalid_TSS),
	INTG(X86_TRAP_NP,		segment_not_present),
	INTG(X86_TRAP_SS,		stack_segment),
	INTG(X86_TRAP_GP,		general_protection),
	INTG(X86_TRAP_SPURIOUS,		spurious_interrupt_bug),
	INTG(X86_TRAP_MF,		coprocessor_error),
	INTG(X86_TRAP_AC,		alignment_check),
	INTG(X86_TRAP_XF,		simd_coprocessor_error),
	TSKG(X86_TRAP_DF,		GDT_ENTRY_DOUBLEFAULT_TSS),
	INTG(X86_TRAP_DB,		debug),
	INTG(X86_TRAP_MC,		&machine_check),
	SYSG(X86_TRAP_OF,		overflow),
	SYSG(IA32_SYSCALL_VECTOR,	entry_INT80_32),
};
```

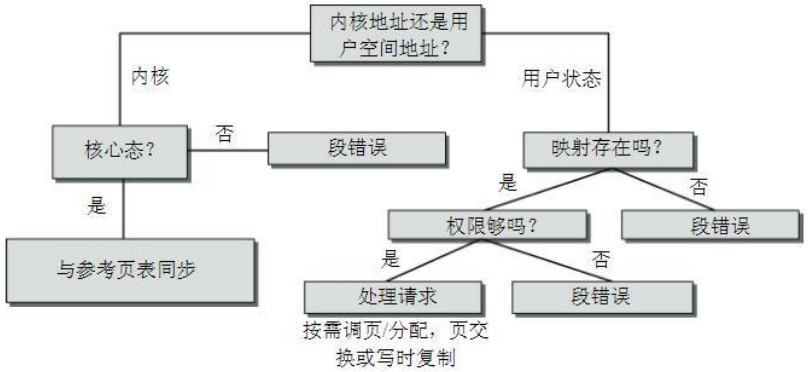
除零异常

```
static void do_error_trap(struct pt_regs *regs, long error_code, char *str,
    unsigned long trapnr, int signr, int sicode, void __user *addr)
{

/* Modification by UV - Begin */
    if (trapnr == X86_TRAP_DE)
    {
        printk("(%d) divide error\n", current->pid);
    }
/* Modification by UV - End */

    RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");

    /*
     * WARN*()s end up here; fix them up before we call the
     * notifier chain.
     */
    if (!user_mode(regs) && fixup_bug(regs, trapnr))
        return;

    if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
            NOTIFY_STOP) {
        cond_local_irq_enable(regs);
        do_trap(trapnr, signr, str, regs, error_code, sicode, addr);
    }
}

#define IP ((void __user *)uprobe_get_trap_addr(regs))
#define DO_ERROR(trapnr, signr, sicode, addr, str, name)           \
dotraplinkage void do_##name(struct pt_regs *regs, long error_code)    \
{                                                  \
    do_error_trap(regs, error_code, str, trapnr, signr, sicode, addr); \
}

DO_ERROR(X86_TRAP_DE, SIGFPE, FPE_INTDIV, IP, "divide error", divide_error)
```

```
ENTRY(divide_error)
    ASM_CLAC
    pushl   $0              # no error code
    pushl   $do_divide_error
    jmp common_exception
END(divide_error)

common_exception:
    /* the function address is in %gs's slot on the stack */
    pushl   %fs
    pushl   %es
    pushl   %ds
    pushl   %eax
    movl    $(__USER_DS), %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    $(__KERNEL_PERCPU), %eax
    movl    %eax, %fs
    pushl   %ebp
    pushl   %edi
    pushl   %esi
    pushl   %edx
    pushl   %ecx
    pushl   %ebx
    SWITCH_TO_KERNEL_STACK
    ENCODE_FRAME_POINTER
    cld
    UNWIND_ESPFIX_STACK
    GS_TO_REG %ecx
    movl    PT_GS(%esp), %edi       # get the function address
    movl    PT_ORIG_EAX(%esp), %edx     # get the error code
    movl    $-1, PT_ORIG_EAX(%esp)      # no syscall to restart
    REG_TO_PTGS %ecx
    SET_KERNEL_GS %ecx
    TRACE_IRQS_OFF
    movl    %esp, %eax          # pt_regs pointer
    CALL_NOSPEC %edi
    jmp ret_from_exception
END(common_exception)
```

缺页异常

```
ENTRY(page_fault)
    ASM_CLAC
    pushl   $do_page_fault
    ALIGN
    jmp common_exception
END(page_fault)

dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2(); /* Get the faulting address */
    enum ctx_state prev_state;

/* Modification by UV - Begin */
    if (strcmp(current->comm, "page_fault") == 0)
    {
        printk("(%d: %lx) page falt\n", current->pid, address);
    }
/* Modification by UV - End */

    prev_state = exception_enter();
    if (trace_pagefault_enabled())
        trace_page_fault_entries(address, regs, error_code);

    __do_page_fault(regs, error_code, address);
    exception_exit(prev_state);
}

static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
        unsigned long address)
{
    prefetchw(&current->mm->mmap_sem);

    if (unlikely(kmmio_fault(regs, address)))
        return;

    /* Was the fault on kernel-controlled part of the address space? */
    if (unlikely(fault_in_kernel_space(address)))
        do_kern_addr_fault(regs, hw_error_code, address);
    else
        do_user_addr_fault(regs, hw_error_code, address);
}
```



内核页面由于使用频繁，通常不会被换出，所以这里是"unlikely"。

```
do_kern_addr_fault()
    --> vmalloc_fault()
```

对于用户空间，需要区分多种情况，page fault的处理显得更为复杂。

```
do_user_addr_fault()
    --> handle_mm_fault()
        --> handle_pte_fault()
```

5.4 系统调用

- 为了和用户空间的进程进行交互，内核提供了一组接口。通过该接口，应用程序可以访问硬件设备和其它操作系统资源。这组接口称为系统调用（System Call）。除异常和中断外，系统调用是应用惟一的内核入口

- 系统调用层的作用：
  - 为用户空间提供了硬件的抽象接口
  - 保证了系统的稳定和安全（统一接口为完全的权限判断机制提供了基础）
  - 提供系统虚拟化

系统调用概述

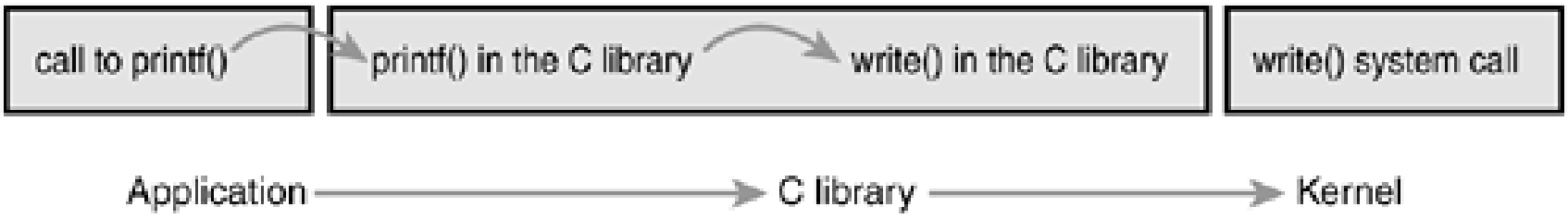
- 为了和用户空间的进程进行交互，内核提供了一组接口。通过该接口，应用程序可以访问硬件设备和其它操作系统资源。这组接口称为**系统调用（System Call）。**除异常和中断外，系统调用是应用惟一的内核入口
- 系统调用层的作用：
  - 为用户空间提供了硬件的抽象接口
  - 保证了系统的稳定和安全（统一接口为完全的权限判断机制提供了基础）
  - 提供系统虚拟化

API、POSIX和C库

- 一般情况下，应用程序通过API间接调用系统调用。API和系统调用之间并不是一一对应。API可以在不同操作系统上实现（具体实现迥异），而为应用提供完全相同的接口。例如printf
- 在Unix世界中，API符合POSIX标准
- POSIX（Portable Operating System Interface of Unix）是一组与可移植性相关的标准，涉及多个方面，API的一致性是其中重要的内容

API、POSIX和C库

- Linux和大多数Unix一样，在C库中实现了Unix的主要API（提供了绝大多数POSIX API），Linux上的C库为libc



系统调用

- 系统调用以函数的形式实现，返回值都为long型数据。
- 系统调用命名规则为：sys_系统调用名

File: v4.10/source/kernel/sys.c

```
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current);
}
```

```
asmlinkage long sys_getpid(void) {
    ……
}
```

```
SYSCALL_DEFINE1(umask, int, mask)
{
        mask = xchg(&current->fs->umask, mask & S_IRWXUGO);
        return mask;
}
```

```
asmlinkage long sys_umask(int mask) {
    ……
}
```

系统调用

- 函数定义前加宏asmlinkage ,表示这些函数通过堆栈而不是通过寄存器传递参数
- 系统调用在用户空间和内核空间有不同的返回值类型，在用户空间为int，在内核空间为long
- 通常，用负值表示错误、0表示成功（不全是）系统调用在出现错误的时候C库会把错误码写入errno全局变量，通过调用perror()库函数，把该变量翻译成用户可以理解的错误字符串。

系统调用号

- 每个系统调用被赋予一个系统调用号（ID），一旦分配就不能更改。
- 系统调用号其实是系统调用表的索引，系统调用表中存放系统调用实现函数的入口

- 内核记录了系统调用表中的所有已注册过的系统调用的列表，存储在sys_call_table中，在x86-64中，它定义在arch/x86/entry/syscall_64.c中

系统调用号

- arch/x86/entry/syscall_64.c

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
        /*
         * Smells like a compiler bug -- it doesn't work
         * when the & below is removed.
         */
        [0 ... __NR_syscall_max] = &sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

- arch/x86/include/generated/asm/syscalls_64.h

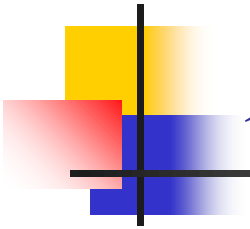
__SYSCALL_64(0, sys_read,)
__SYSCALL_64(1, sys_write,)
__SYSCALL_64(2, sys_open,)
……

系统调用号

- 当用户空间的进程执行一个系统调用时，就用系统调用号指明到底执行哪个系统调用。进程不会提及系统调用的名称
- Linux有一个“未实现”系统调用sys_ni_syscall()，它除了返回-ENOSYS外不做任何其他工作，这个错误号就是专门针对无效的系统调用而设的。

系统调用的性能

- Linux系统调用比其他许多操作系统执行得要快。
- 原因：
  - 上下文切换时间短
  - 系统调用处理程序和每个系统调用本身也都非常简洁



系统调用的硬件原理

- 1 基于中断的系统调用机制
  - 1.1 使用int 指令进入系统调用
  - 1.2 使用iret指令退出系统调用
- 2 快速系统调用机制
  - 2.1 快速系统调用概述
  - 2.2 使用sysenter指令进入系统调用
  - 2.3 使用sysexit指令退出系统调用

1.1 使用int 指令进入系统调用

- Int指令硬件流程-无运行级别切换

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 6-5):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure.

本小节参考?Intel 64 and IA-32 Architectures Software Developer Manua?

1.1 使用int 指令进入系统调用

- 进入系统调用-无运行级别切换
- 如果不发生堆栈切换,处理器在调用一个中断或者异常handler时将做以下工作:
- 1.把EFLAGS,CS,和EIP存放器〔以这种顺序〕的当前目录压栈。
- 2.把错误代码压栈.
- 3.把段选择子分别赋给CS和EIP存放器作为新的代码段和新的指令指针〔位于中断门或者陷阱门〕.
- 4.如果调用通过中断门，那么去除在EFLAGS存放器中的IF标志.
- 5.开场执行handler程序。

本小节参考?Intel 64 and IA-32 Architectures Software Developer Manua?

1.1 使用int 指令进入系统调用

- Int指令硬件流程-有运行级别切换

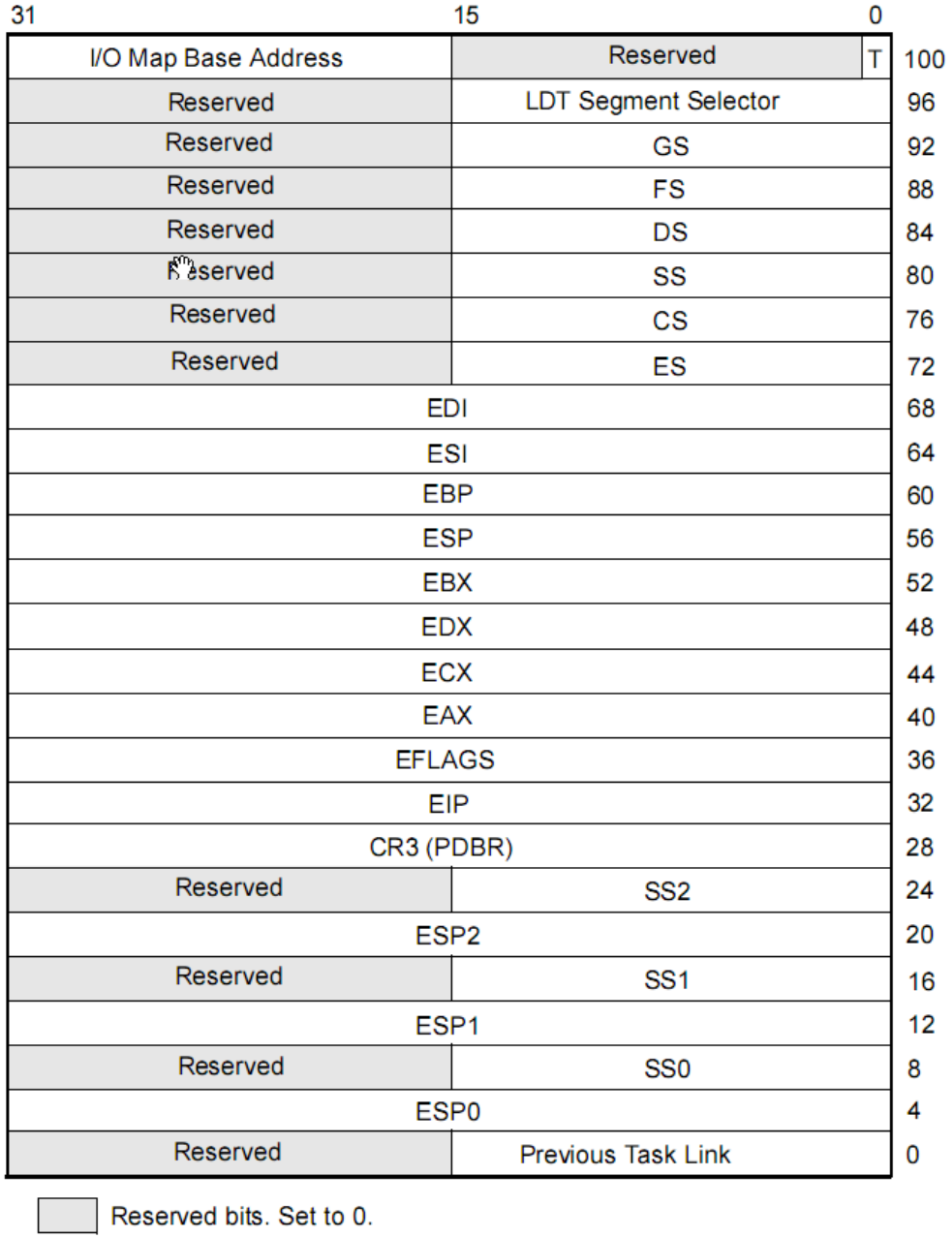
If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

1.1 使用int 指令进入系统调用

- 进入系统调用-有运行级别切换
- 如果不发生堆栈切换,处理器将做以下工作：
- 1.临时保存当前SS,ESP,EFLAGS,CS和EIP存放器的当前目录
- 2.从TSS取到段选择子赋给SS和ESP存放器作为新的堆栈(就是被调用的特权级别的堆栈),并转到新的堆栈.
- 3.临时把用于中断程序堆栈的被保存的SS,ESP,EFLAGS,CS和EIP值压到一个新的堆栈.
- 4.把错误代码放到一个新的堆栈中(如果适当的话).
- 5.把段选择子分别赋给CS和EIP存放器作为新的代码段和新的指令指针〔位于中断门或者陷阱门〕.
- 6.如果调用通过中断门，那么去除在EFLAGS存放器中的IF标志.

- TSS
  - Task State Segment
  - Intel设计用来存放硬件上下文，实现任务切换



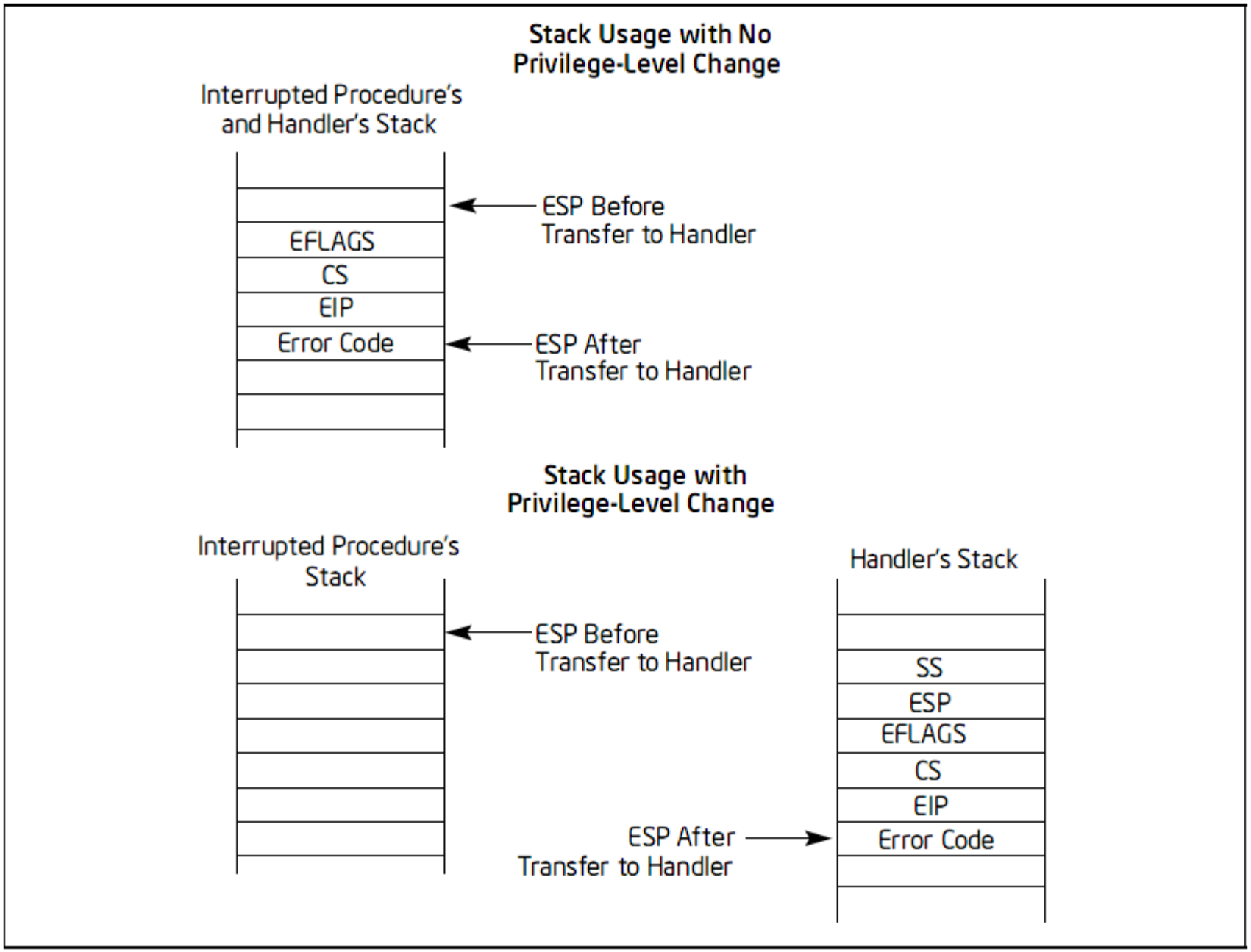


Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

1.2 使用iret指令退出系统调用

- 没有运行级别切换时的硬件操作如下：：

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately.
4. Resumes execution of the interrupted procedure.

- 有运行级别切换时的硬件操作如下：：

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

1.2 使用iret指令退出系统调用

- 没有运行级别切换时的硬件操作如下：
- 1.恢复CS和EIP存放器在中断或异常之前的值.
- 2.恢复EFLAGS存放器.
- 3.适当增加堆栈指针.
- 4.重新执行中断程序.
- 有运行级别切换时的硬件操作如下：
- 1.执行特权检测.
- 2.恢复CS和EIP存放器在中断或异常之前的值.
- 3.恢复EFLAGS存放器.
- 4.恢复SS和ESP存放器在中断或异常之前的值,以引起堆栈切换,返回到中断处理之前的堆栈.
- 5.重新执行中断程序.

2.1 快速系统调用概述

- sysenter/sysexit
  - sysenter 指令用于由 Ring3 进入 Ring0
  - sysexit 指令用于由 Ring0 返回 Ring3
- 与中断机制的区别
  - sysenter/sysexit 指令并不成对
    - sysenter 指令并不会把 SYSEXIT 所需的返回地址压栈，
    - sysexit 返回的地址并不一定是 sysenter 指令的下一个指令地址
- 与中断机制比的优势
  - 没有特权级别检查的处理
  - 没有压栈的操作
  - 所以执行速度比 INT n/IRET 快

2.1 快速系统调用概述

- 系统调用性能测试测试硬件：
  - Intel® Pentium® III CPU, 450 MHzProcessor Family: 6 Model: 7 Stepping: 2

	用户模式花费的时间	核心模式花费的时间
基于 sysenter/sysexit 指令的系统调用	9.833 microseconds	6.833 microseconds
基于中断 INT n 指令的系统调用	17.500 microseconds	7.000 microseconds

  - 各种

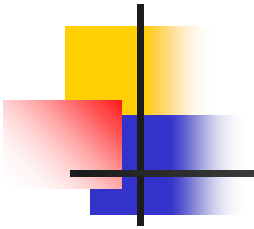
CPU	Int0x80	sysenter
Athlon XP 1600+	277	169
800MHz mode 1 athlon	279	170
2.8GHz p4 northwood ht	1152	442

2.1 快速系统调用概述

- sysenter/sysexit 引入的存放器包括：
  - SYSENTER_CS_MSR —
    - 用于指定要执行的 Ring 0 代码的代码段选择符
    - 由它还能得出目标 Ring 0 所用堆栈段的段选择符；
  - SYSENTER_EIP_MSR —
    - 用于指定要执行的 Ring 0 代码的起始地址；
  - SYSENTER_ESP_MSR—
    - 用于指定要执行的Ring 0代码所使用的栈指针

2.2 使用sysenter指令进入系统调用

- 在 Ring3 的代码调用了 sysenter 指令之后，CPU 会做出如下的操作：
  - 1将 SYSENTER_CS_MSR 的值装载到 cs 存放器
  - 2将 SYSENTER_EIP_MSR 的值装载到 eip 存放器
  - 3将 SYSENTER_CS_MSR 的值加 8〔Ring0 的堆栈段描述符〕装载到 ss 存放器。
  - 4将 SYSENTER_ESP_MSR 的值装载到 esp 存放器
  - 5将特权级切换到 Ring0
  - 6开场执行指定的 Ring0 代码



Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

2.3 使用sysexit指令退出系统调用

- 在 Ring0 代码执行完毕，调用 SYSEXIT 指令退回 Ring3 时，CPU 会做出如下操作：
  - 1将 SYSENTER_CS_MSR 的值加 16〔Ring3 的代码段描述符〕装载到 cs 存放器
  - 2将存放器 edx 的值装载到 eip 存放器
  - 3将 SYSENTER_CS_MSR 的值加 24〔Ring3 的堆栈段描述符〕装载到 ss 存放器
  - 4将存放器 ecx 的值装载到 esp 存放器
  - 5将特权级切换到 Ring3
  - 6继续执行 Ring3 的代码

系统调用处理程序

- 用户空间的程序无法直接访问内核代码，调用其中的函数（用户空间、内核空间隔离）
- 调用系统调用通过软中断实现（引发一个异常，让内核去执行异常处理函数，即系统调用处理程序）
- x86中通过int 0x80调用系统调用，相应的处理函数为sytem_call()，实现在arch/x86/entry/entry_64.S中

通过INT 0x80中断方式进入系统调用

在 2.6以前的 Linux 2.4 内核中，用户态 Ring3 代码请求内核态 Ring0 代码完成某些功能是通过系统调用完成的，而系统调用的是通过软中断指令(int 0x80) 实现的。在 x86 保护模式中，处理 INT 中断指令时

1. CPU 首先从中断描述表 IDT 取出对应的门描述符
2. 判断门描述符的种类
3. 检查门描述符的级别 DPL 和 INT 指令调用者的级别 CPL，当 CPL<=DPL 也就是说 INT 调用者级别高于描述符指定级别时，才能成功调用
4. 根据描述符的内容，进行压栈、跳转、权限级别提升
5. 内核代码执行完毕之后，调用 IRET 指令返回，IRET 指令恢复用户栈，并跳转会低级别的代码

在发生系统调用，由 Ring3 进入 Ring0 的这个过程浪费了不少的 CPU 周期，例如，系统调用必然需要由 Ring3 进入 Ring0，权限提升之前和之后的级别是固定的，CPL 肯定是 3，而 INT 80 的 DPL 肯定也是 3，这样 CPU 检查门描述符的 DPL 和调用者的 CPL 就是完全没必要。正是由于如此，Intel x86 CPU 从 PII 300(Family 6，Model 3，Stepping 3)之后，开始支持新的系统调用指令 sysenter/sysexit

通过sysenter指令方式进入系统调用

sysenter 指令用于由 Ring3 进入 Ring0，SYSEXIT 指令用于由 Ring0 返回 Ring3。由于**没有特权级别检查**的处理，也**没有压栈**的操作，所以执行速度比 INT n/IRET 快了不少。

sysenter和sysexit都是CPU原生支持的指令集

不同系统调用方法的性能比较:

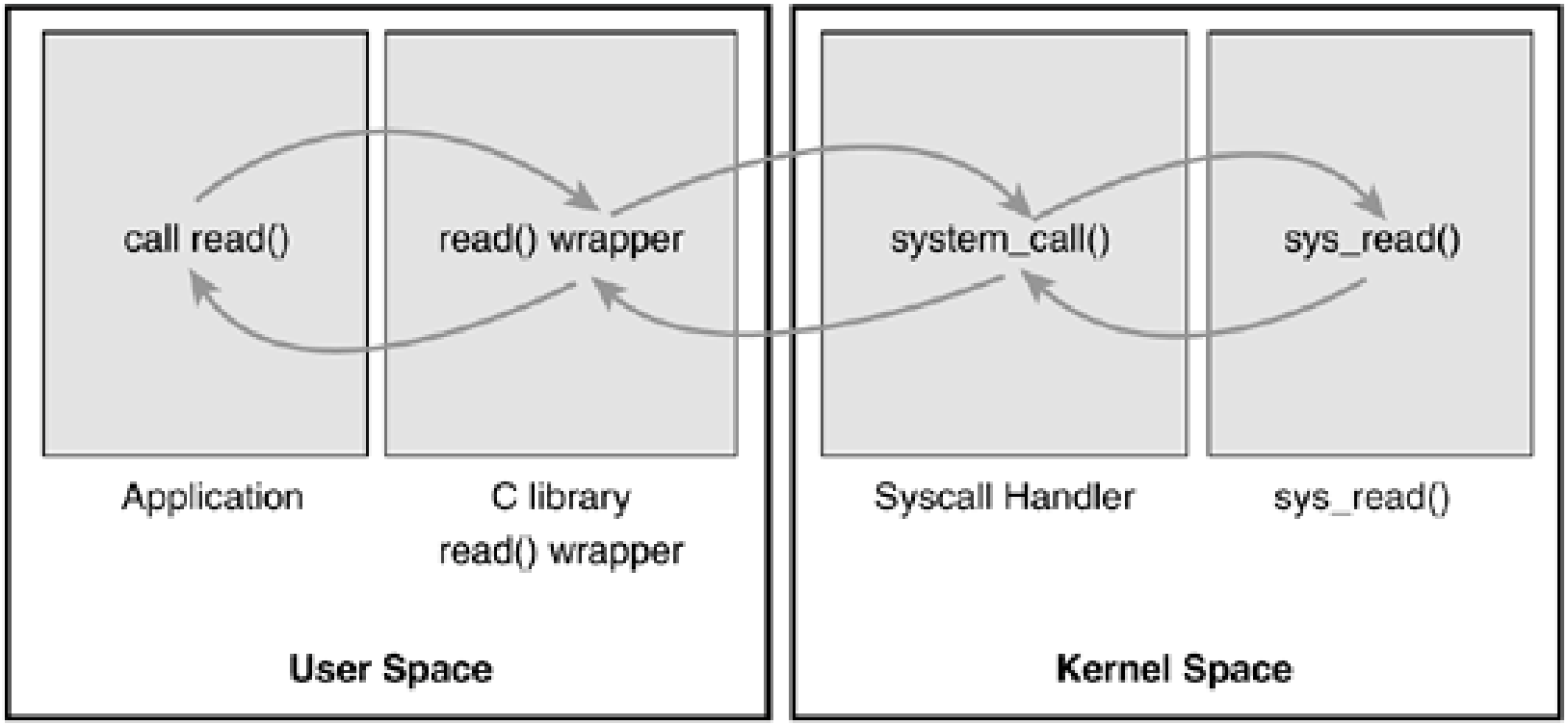
	用户模式花费的时间	核心模式花费的时间
基于 sysenter/sysexit 指令的系统调用	9.833 microseconds	6.833 microseconds
基于中断 INT n 指令的系统调用	17.500 microseconds	7.000 microseconds

CPU	Int0x80	sysenter
Athlon XP 1600+	277	169
800MHz mode 1 athlon	279	170
2.8GHz p4 northwood ht	1152	442

系统调用处理程序

- 通过软中断陷入内核后，还需要通知内核调用的是哪个系统调用（系统调用号）和调用的参数
- 在x86上，系统调用号和参数通过寄存器传递给内核，其中系统调用号使用eax寄存器
- System_call函数使用系统调用号作为索引访问系统调用表，调用相应的系统调用实现函数

```
call *sys_call_table(,%eax,4)
```


call read()
read() wrapper
system_call()
sys_read()
Application
C library
read() wrapper
Syscall Handler
sys_read()
User Space
Kernel Space

系统调用处理程序

- 在x86中，系统调用参数传递也通过寄存器，参数依次使用ebx，ecx、edx、esi和edi，所有参数大小均为32位
- 系统调用的返回值也存放在eax寄存器中
- 在64位系统中，使用RAX寄存器传递系统调用号，RDI、RSI、RDX、RCX、R8、R9这6个寄存器则用来传递参数，系统调用的返回值也存放在RAX寄存器中

系统调用的实现

- Linux中不提倡多用途的系统调用，系统调用的实现应该“提供机制而不是策略”

- 调用系统调用时，必须考虑用户态/内核态数据交互的问题，交互方式有2种：
  - 通过寄存器
  - 通过指针

系统调用的实现

- 用户空间和内核空间通过指针拷贝数据由copy_from_user和copy_to_user函数实现

- 在引用一个用户空间的指针时，内核必须：
  - 指针指向的内存属于用户空间
  - 指针指向的内存在进程的地址空间中（而不是别的进程）
  - 对此内存的操作符合内存设置（读、写）

对内核而言，处理用户空间传入的数据是极大的考验！

```
asmlinkage long sys_silly_copy(unsigned long *src,
                               unsigned long *dst,
                               unsigned long len)
{
        unsigned long buf;

        /* fail if the kernel wordsize and user wordsize do not match */
        if (len != sizeof(buf))
                return -EINVAL;

        /* copy src, which is in the user's address space, into buf */
        if (copy_from_user(&buf, src, len))
                return -EFAULT;

        /* copy buf into dst, which is in the user's address space */
        if (copy_to_user(dst, &buf, len))
                return -EFAULT;

        /* return amount of data copied */
        return len;
}
```

系统调用的实现

- 在实现一个系统调用时，还必须考虑当前进程是否有执行相应操作的权限。其中capable()函数用来检查当前进程是否具有超级用户（root）权限，如：

```
asmlinkage long sys_am_i_popular (void)
{
        /* check whether the user possesses the CAP_SYS_NICE capability */
        if (!capable(CAP_SYS_NICE))
                return EPERM;

        /* return zero for success */
        return 0;
}
```

系统调用上下文

- 内核在执行系统调用时处于进程上下文，current指向当前进程
- 在进程上下文中，内核可以休眠，并且可以被抢占，应该注意同步问题，保证系统调用是可重入的
- 当系统调用返回时，首先返回到system_call()中，由其负责切换回用户空间

增加系统调用

- 步骤：
  1. 在系统调用表中追加一个表项（覆盖所有要支持的体系结构）
  2. 在asm/unistd.h中定义系统调用号
  3. 将系统调用实现函数放置在一个必须编译进内核的源码文件中，如kernel/目录下的.c文件中

```
ENTRY(sys_call_table)
        .long sys_restart_syscall    /* 0 */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open               /* 5 */

    ...

        .long sys_mq_unlink
        .long sys_mq_timedsend
        .long sys_mq_timedreceive       /* 280 */
        .long sys_mq_notify
        .long sys_mq_getsetattr
```

The new system call is then appended to the tail of this list:

```
        .long sys_foo
```

```
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall  0
#define __NR_exit             1
#define __NR_fork             2
#define __NR_read             3
#define __NR_write            4
#define __NR_open             5

...
#define __NR_mq_unlink        278
#define __NR_mq_timedsend     279
#define __NR_mq_timedreceive  280
#define __NR_mq_notify        281
#define __NR_mq_getsetattr    282
```

The following is then added to the end of the list:

```
#define __NR_foo              283
```

```
#include <asm/thread_info.h>

/*
 * sys_foo  everyone's favorite system call.
 *
 * Returns the size of the per-process kernel stack.
 */
asmlinkage long sys_foo(void)
{
        return THREAD_SIZE;
}
```

从用户空间访问系统调用

- 对于新增加的系统调用，glibc库中无现成的封装API，可通过**_syscalln()**宏（n为0～6，表示系统调用参数个数）调用系统调用
- 每个宏有2＋2×n个参数：
  - 系统调用返回值类型
  - 系统调用名
  - 参数类型和值

从用户空间访问系统调用

```
long open(const char *filename, int flags, int mode)




#define __NR_open 5
_syscall3(long, open, const char *, filename, int, flags, int, mode)
```

```
#define __NR_foo 283
__syscall0(long, foo)

int main ()
{
        long stack_size;

        stack_size = foo ();
        printf ("The kernel stack size is %ld\n", stack_size);

        return 0;
}
```

实践任务

1. 添加系统调用（源代码层面修改系统调用表、使用内核模块在运行时修改系统调用表），用户调用后打印当前进程pid。
2. 添加一个内核模块，增加键盘按键中断的监听，打印当前输入。
3. RootKit：将按键中断的数据发送到恶意进程中。