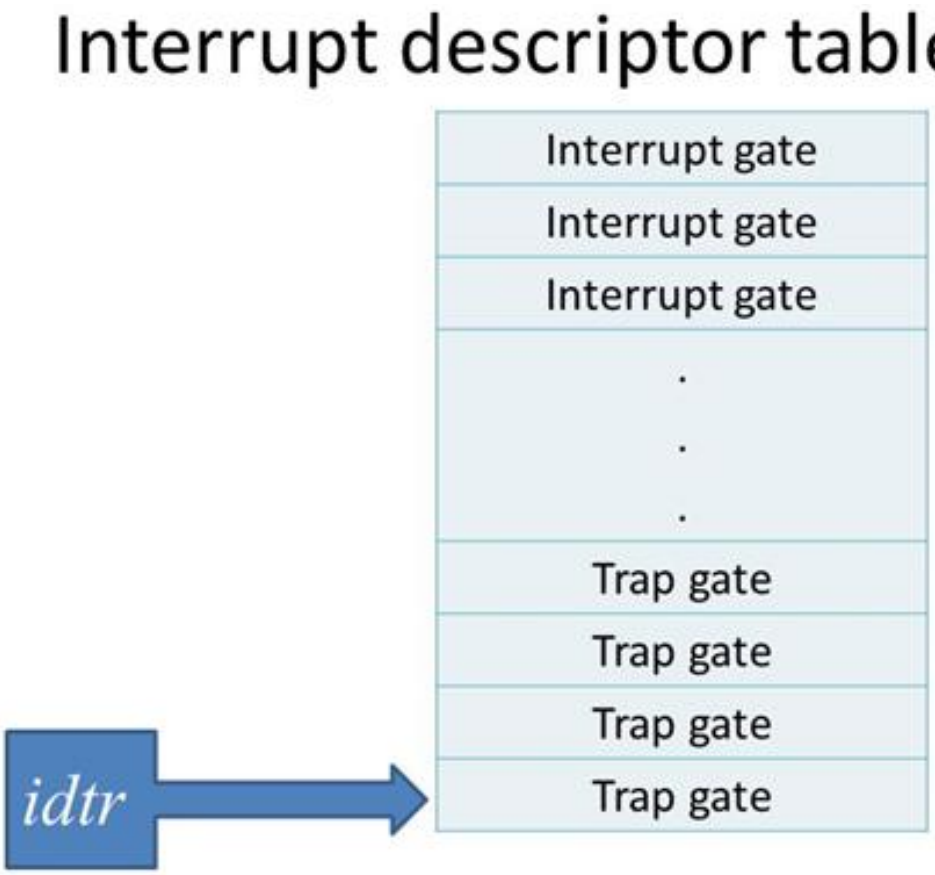
5.2 中断处理机制

西安邮电大学

中断描述表以及初始化



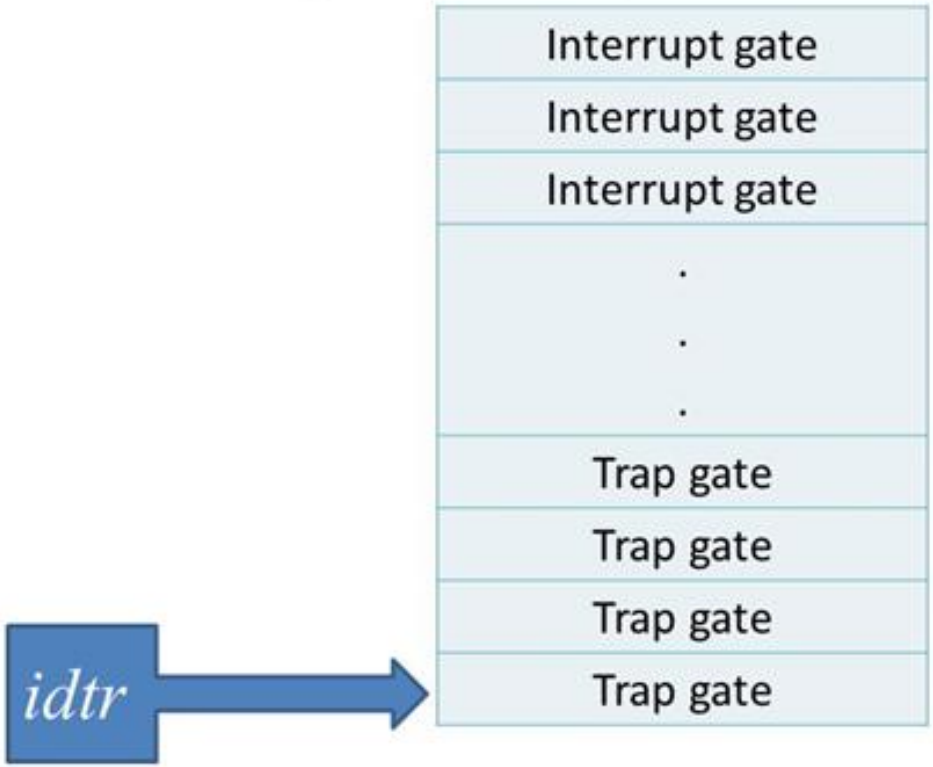
idtr: interrupt descriptor table register

上一讲我们介绍了中段描述符表，那么IDT放在什么地方？什么时候初始化？

实际上，IDT放在内核的数据段中，其起始地址放在中段描述表寄存器（IDTR）中

中断描述符表相关源代码

Interrupt descriptor table (IDT)



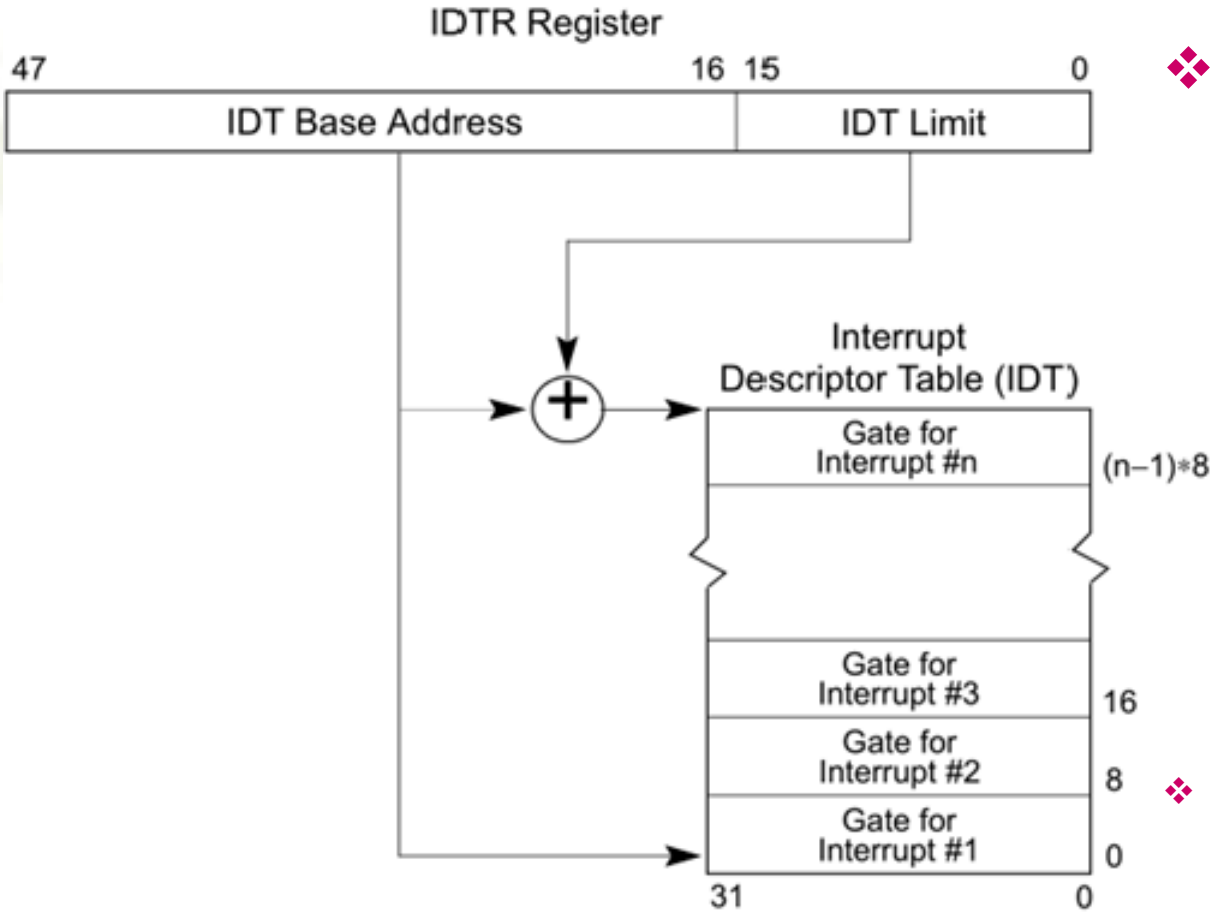
idtr: interrupt descriptor table register

idt_descr变量定义于arch/x86/kernel/head_32.S

```
idt_descr:
    .word IDT_ENTRIES*8-1
# idt contains 256 entries
    .long idt_table
```

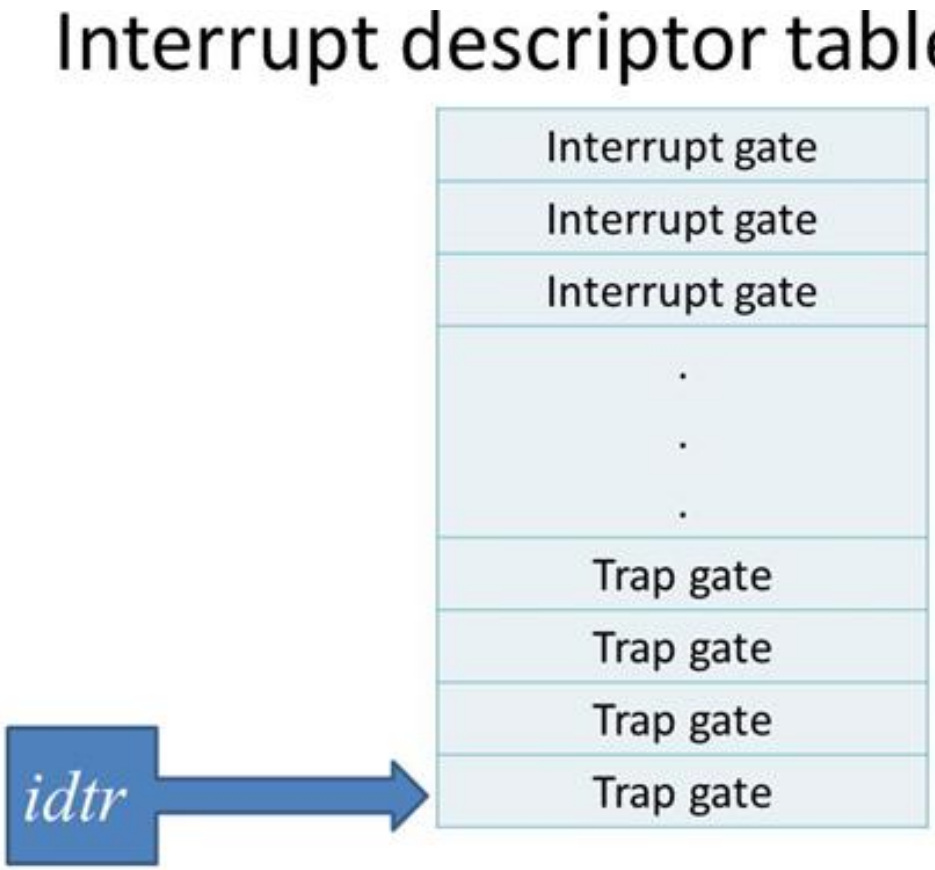
（第一句表示中断描述表包含256项中断描述符，第二句表示中断描述表的入口地址）

初始化中断描述符表



- Linux内核在系统的初始化阶段要初始化可编程控制器；将中断描述符表的起始地址装入中段描述符表（IDTR）寄存器，并初始化表中的每一项，当计算机运行在实模式时，中断描述符表被初始化，并由BIOS使用。
- 真正进入了Linux内核，中断描述符表就被移到内存的另一个区域，并为进入保护模式进行预初始化

初始化陷阱门和系统门



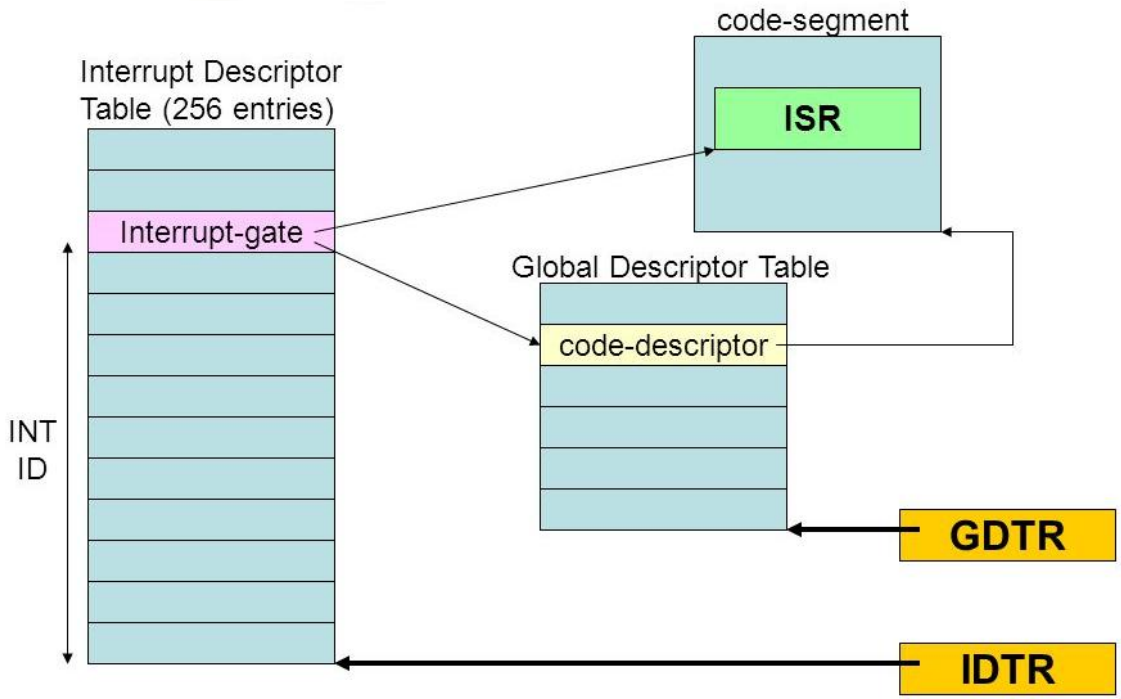
idtr: interrupt descriptor table register

- trap_init()函数用于设置中断描述符表开头的19个陷阱门和系统门
- 这些中断向量都是CPU保留用于异常处理的 ， 例:

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
set_trap_gate(19,&simd_coproce
ssor_error);
set_system_gate(SYSCALL_VECTOR
,&system_call);  (初始化系统门)
```

中断门的设置

```
for (i = 0; i< NR_IRQS; i++) {
 int vector = FIRST_EXTERNAL_VECTOR + i;
 if (vector != SYSCALL_VECTOR)
        set_intr_gate(vector, interrupt[i]); }
```

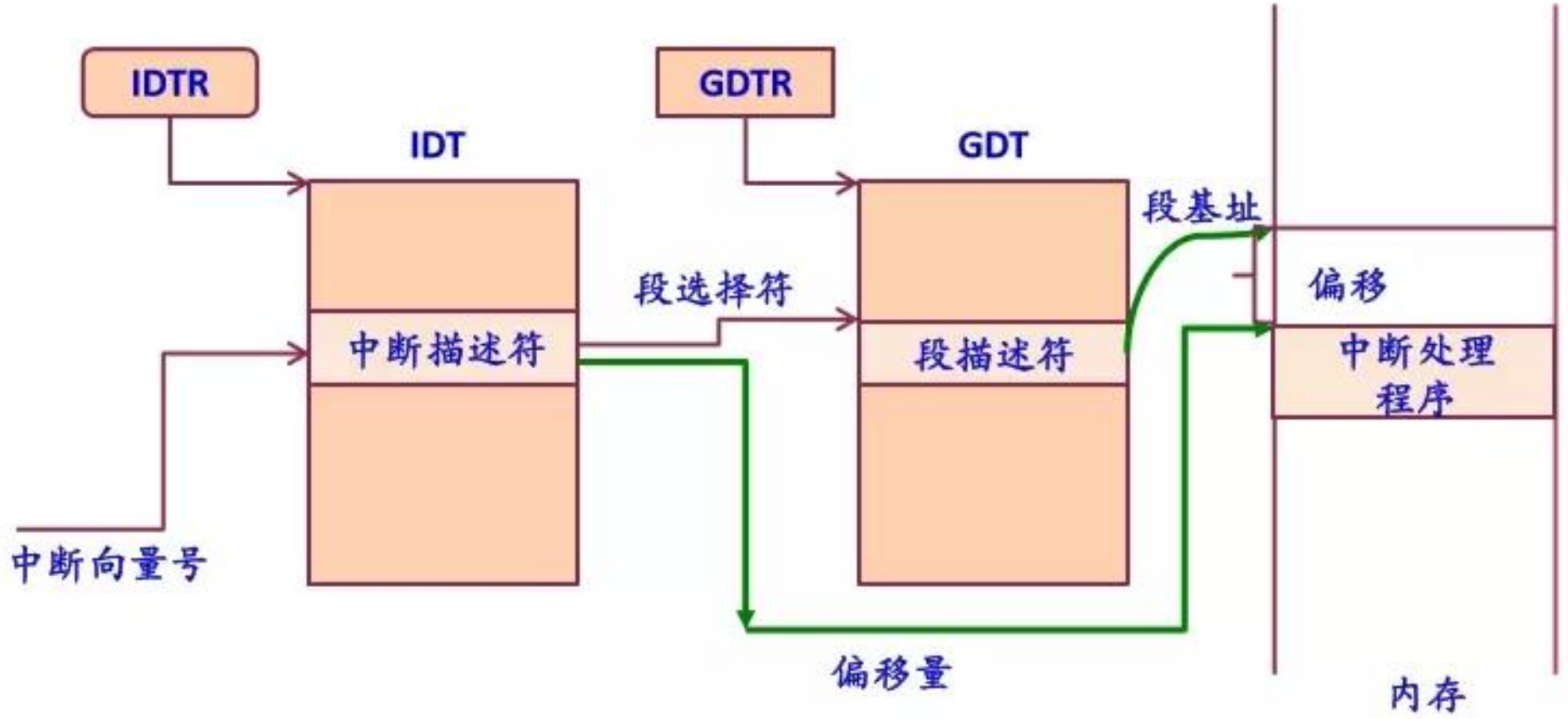


- 中断门的设置是由init_IRQ()函数中的一段代码完成的（讲完后出现代码框）：
- 设置时必须跳过用于系统调用的向量0x80
- 中断处理程序的入口地址是一个数组interrupt[]，数组中的每个元素是指向中断处理例程（ISR）的指针。
- 每个中断处理例程属于内核中的代码段，其段基地址存放于全局描述表（GDT）中

中断处理过程

中断和异常 的硬件处理	• 从硬件的角度看CPU如何处理中断和异常
中断请求队列的建立	• 方便外设共享中断线
中断处理程序的执行	• 中断处理程序 • 中断服务例程（ISR）
从中断返回	

X86中的中断处理



X86中的中断处理

当CPU执行了当前指令之后，CS和EIP这对寄存器中所包含的内容就是下一条将要执行指令的虚地址。

在对下一条指令执行前，CPU先要判断在执行当前指令的过程中是否发生了中断或异常。

如果发生了一个中断或异常，那么CPU将做以下事情：

1. 确定所发生中断或异常的向量i（在0～255之间）
2. 通过IDTR寄存器找到IDT表，读取IDT表第i项（或叫第i个门），从GDRT寄存器获得GDT的地址;结合中断描述符中的段选择符,在GDT表中获得中断处理程序对应的段描述符,从该段描述符中获得中断/异常处理程序所在的段基址，与其偏移量相加，得到中断处理程序的入口地址。

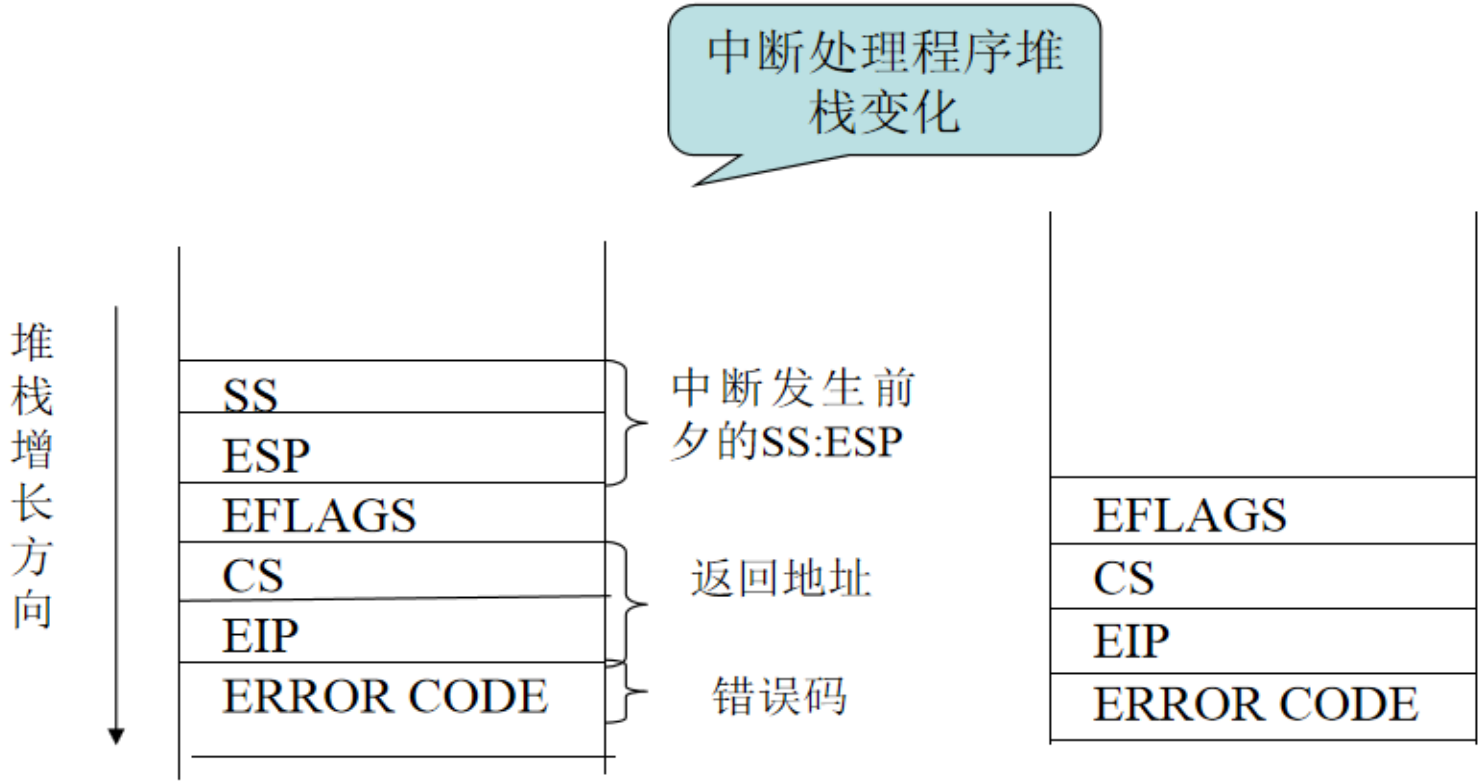
其中要进行“段”级、“门”级两步有效性检查，还要检查是否发生了特权级的变化 。

中断处理过程堆栈变化

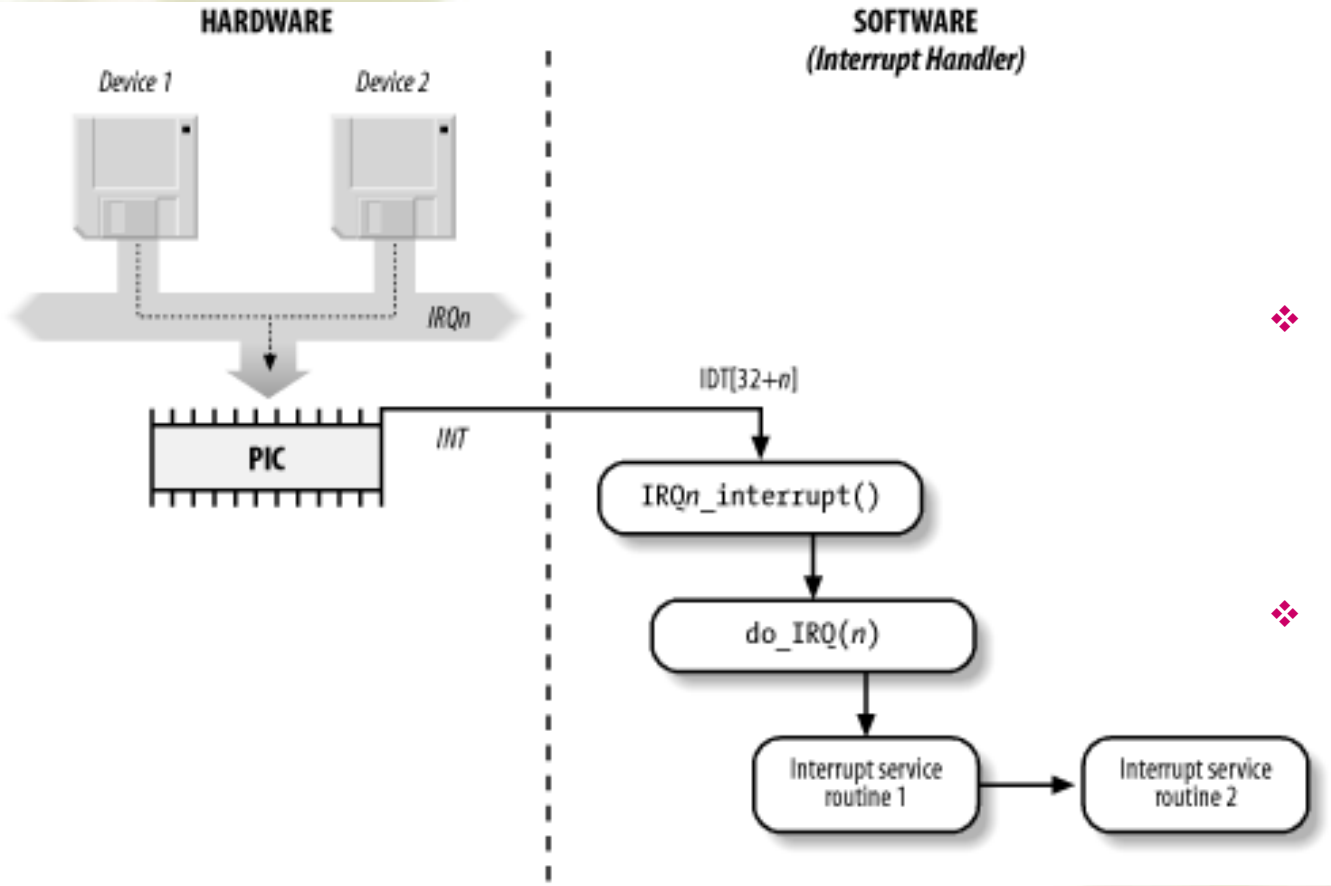
当中断发生在用户态（特权级为3），而中断处理程序运行在内核态（特权级为0），特权级发生了变化，所以会引起堆栈的更换。也就是说，从用户堆栈切换到内核堆栈。而当中断发生在内核态时，即CPU在内核中运行时，则不会更换堆栈，

从图可以看出，当从用户态堆栈切换到内核态堆栈时，先把用户态堆栈的值压入中断程序的内核态堆栈中，同时把EFLAGS寄存器自动压栈，然后把被中断进程的返回地址压入堆栈。如果异常产生了一个硬错误码，则将它也保存在堆栈中。如果特权级没有发生变化，则压入栈中的内容如图右边。前面已经得到中断处理程序的入口地址，于是，CPU就跳转到了中断或异常处理程序。

中断处理过程堆栈变化


中断处理程序堆栈变化
堆栈增长方向
SS
ESP
EFLAGS
CS
EIP
ERROR CODE
中断发生前夕的SS:ESP
返回地址
错误码
EFLAGS
CS
EIP
ERROR CODE

中断处理程序与中断服务例程

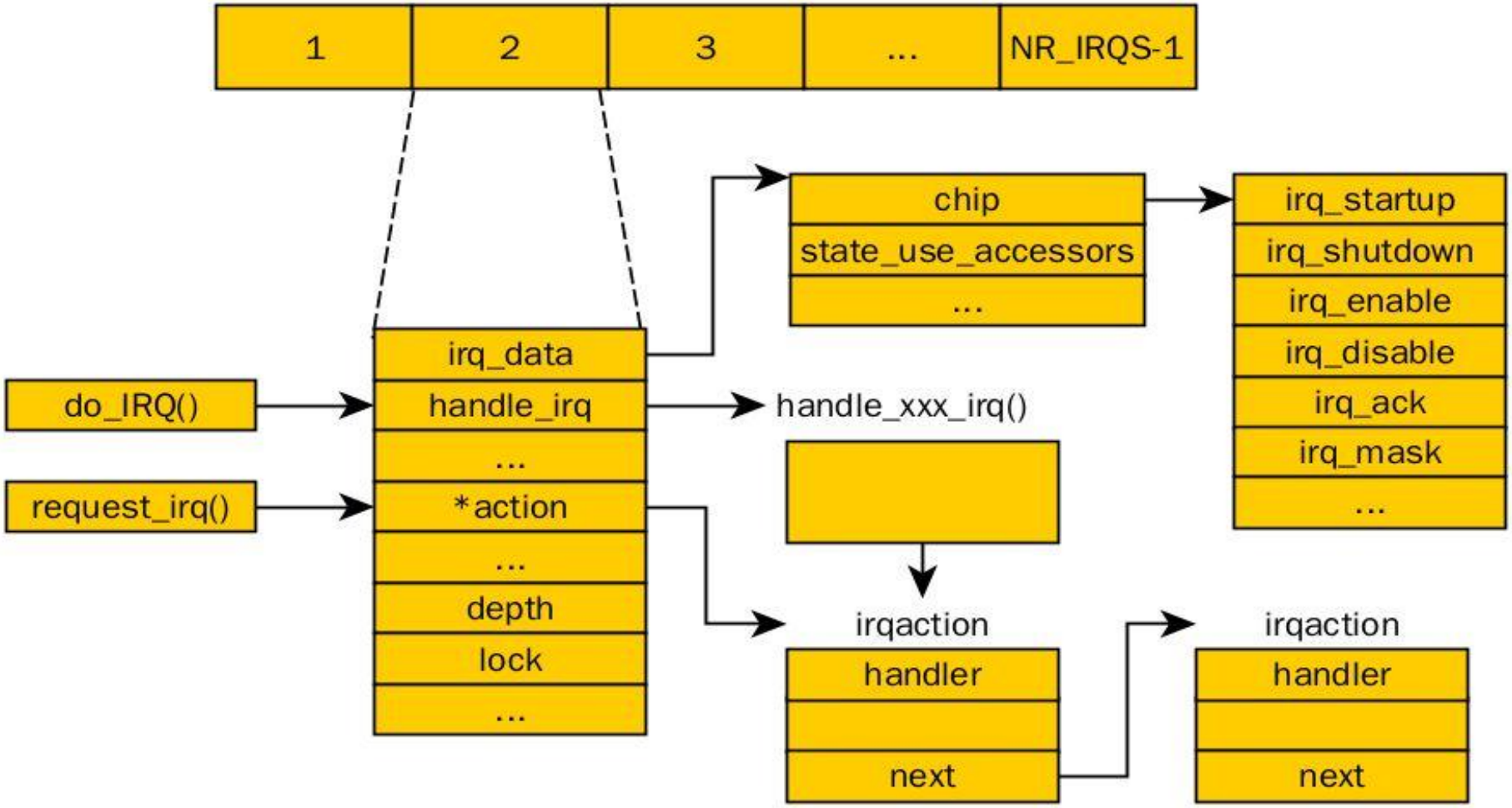


- 当外设发出中断请求时IRQ，通过中断控制器PIC的int引脚向CPU请求中断处理，则中断处理机制启动。其中
- 中断处理程序：共享同一条中断线的所有中断请求，有一个总的中断处理程序
- 中断服务例程（Interrupt Service Routine）：每个中断请求都有自己单独的中断服务例程

中断描述符-IRQ数据结构

在内核中，对于每一个外设的IRQ都用struct irq_desc来描述，我们称之中断描述符。内核中会有一个数据结构保存了关于所有IRQ的中断描述符信息，放在一个数组中，这个irq_desc[NR_IRQS]数组是Linux内核中维护IRQ资源的管理单元，它记录了某IRQ号对应的流控处理函数，中断控制器、中断服务程序、IRQ自身的属性、资源等信息，是内核中断子系统的一个核心数组。

中断描述符-IRQ数据结构



中断线共享的数据结构-irqaction

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

- 每个设备能共享一个单独的IRQ，因此内核要维护多个irqaction描述符，其中每个描述符涉及一个特定的硬件设备和一个特定的中断。
- Handler：指向一个具体I/O设备的中断服务例程
- Flags：用一组标志描述中断线与I/O设备之间的关系。
- Name：I/O设备名
- dev_id：指定I/O设备的主设备号和次设备号
- Next：指向irqaction描述符链表的下一个元素

注册和注销中断服务例程

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

- 中断描述符表IDT初始化后，必须通过 request_irq() 函数将相应的中断服务例程挂入中断请求队列，即对其进行注册，其中，参数中的中断服务例程handler被挂入中断请求列表中。
- 在关闭设备时，必须通过调用free_irq()函数释放所申请的中断请求号

中断处理程序的执行

CPU从中断控制器的一个端口取得中断向量I

根据I从中断描述符表IDT中找到相应的中断门

从中断门获得中断处理程序的入口地址

判断是否要进行堆栈切换

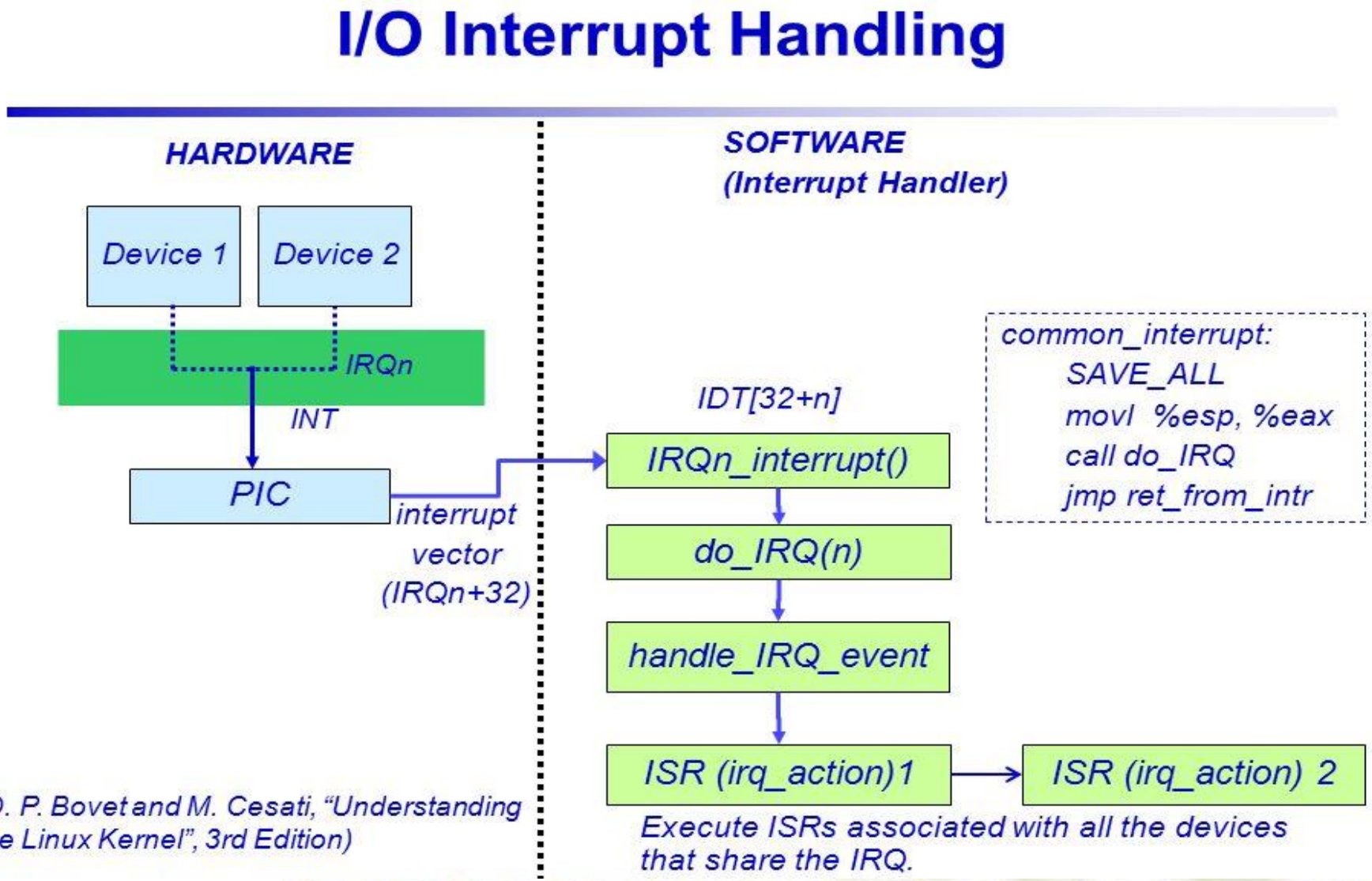
调用do_IRQ()

对所接收的中断进行应答 ，并禁止这条中断线

调用handle_IRQ_event（）来运行对应的中断服务例程

关于这些源代码的分析，将在动手实践一讲中具体查看和分析。

中断处理程序的执行


I/O Interrupt Handling
HARDWARE
SOFTWARE
(Interrupt Handler)
Device 1
Device 2
IRQn
INT
PIC
interrupt
vector
(IRQn+32)
IDT[32+n]
IRQn_interrupt()
do_IRQ(n)
handle_IRQ_event
ISR (irq_action)1
ISR (irq_action) 2
common_interrupt:
SAVE_ALL
movl %esp, %eax
call do_IRQ
jmp ret_from_intr
(D. P. Bovet and M. Cesati, "Understanding the Linux Kernel", 3rd Edition)
Execute ISRs associated with all the devices that share the IRQ.

中断返回（ret_from_intr）

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

- 所有的中断处理程序在处理完之后都要走到ret_from_intr这里；
- 判断进入中断前是用户空间还是内核空间
- 如果进入中断前是内核空间，则直接调用RESTORE_ALL
- 如果进入中断前是用户空间，则可能需要进行一次调度；如果不调度，则可能有信号需要处理；最后，还是走到 RESTORE_ALL
- RESOTRE_ALL 和 SAVE_ALL 是相反的操作，将堆栈中的寄存器恢复
- 最后，调用 iret 指令 ，将处理权交给 CPU从中断返回时，CPU要调用恢复中断现场的宏。

从中断/异常/系统调用返回

中断返回除了返回现场外，OS还做了什么？

带着四个问题来讨论

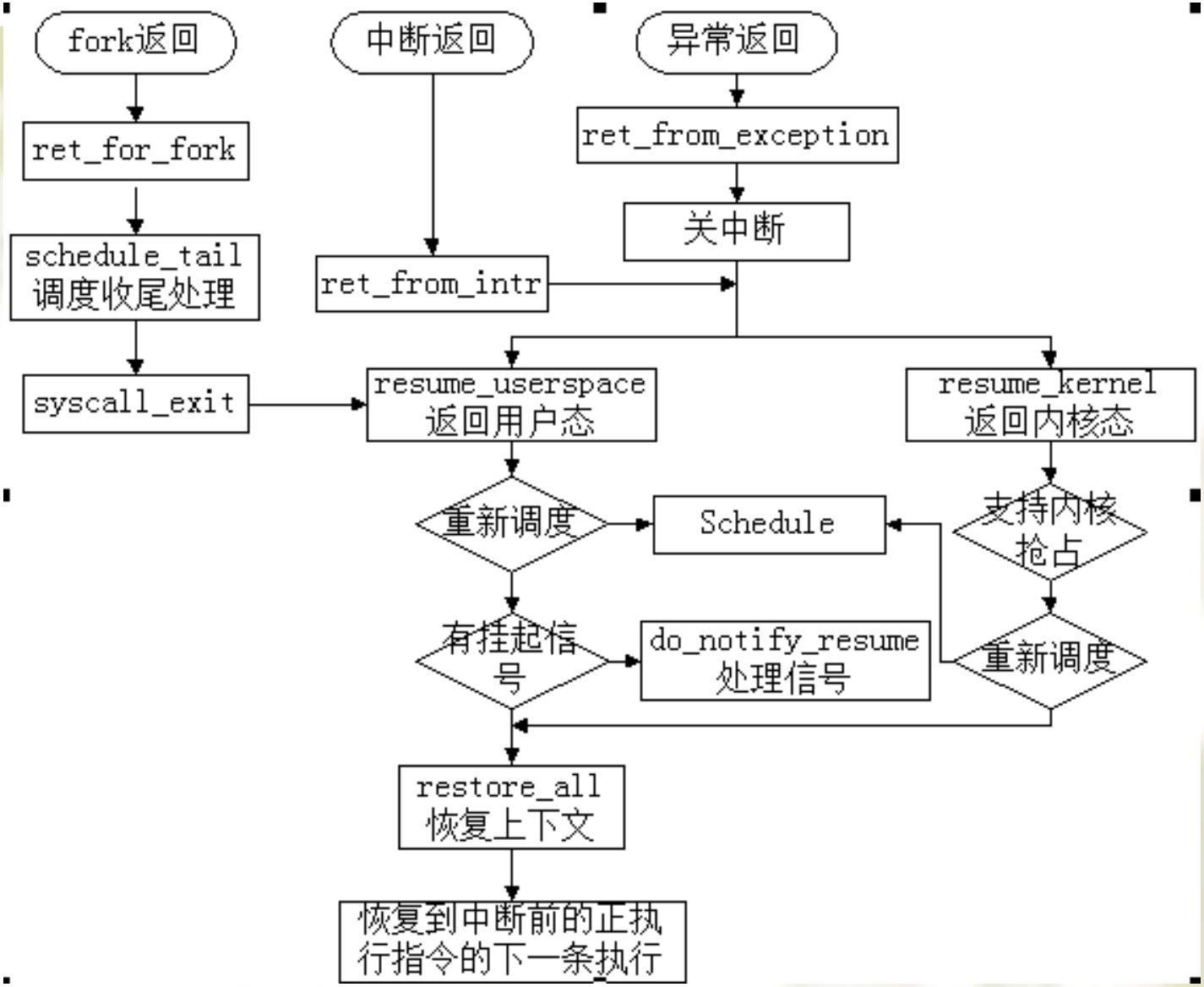
1、内核调度与中断/异常/系统（统称中断）调用的关系如何？

2、信号处理与中断/异常/系统调用的关系如何？

3、内核抢占与中断/异常/系统调用的关系如何？

4、内核线程的调度有何特别之处？中断/异常/系统调用返回时，内核线程会发生调度吗？

从中断/异常/系统调用返回



从中断/异常/系统调用返回

1）中断返回和异常返回的流程基本一致，差别主要在于异常返回时，需要先关一次中断。因为Linux实现中，异常使用的是陷阱门，通过时不会自动关中断；而中断使用的是中断门，通过时会自动关中断。

2）中断/异常(包括系统调用)返回时，是进行调度(schedule)的重要时机点，其中，时钟中断返回时是调度依赖的最主要的时机点，时钟中断处理函数中不会直接进行调度，只是根据相应的调度算法，决定是否需要调度，以及调度的下一个任务，如果需要调度，则设置调度标志NEED_RESCHED标记。调度(schedule)的实际执行是在中断返回的时候，检查NEED_RESCHED标记，如果设置则进行调度。

3）信号处理是在当前进程从内核态返回用户态时进行的，在发生中断、异常(包括系统调用)、或fork时，都有可能从内核态返回用户态，都是处理信号的时机。注意：只有当前进程的信号才能在此时得到处理。其它非正在运行的进程的信号无法处理。

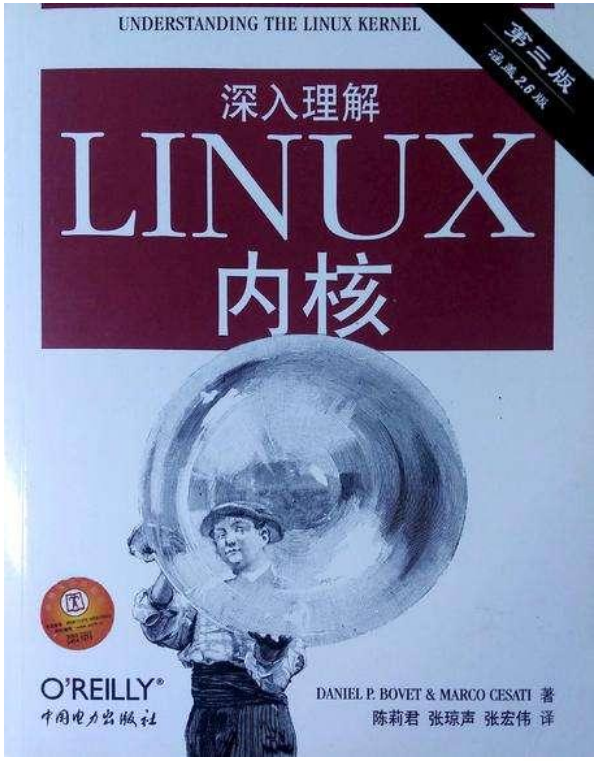
关于内核枪占和内核线程的调度，非常有趣，留给读者来思考。

动手实践

Linux内核之旅网站 http://www.kerneltravel.net/

电子杂志栏目第八期“中断”，向读者依次解释中断概念，解析Linux中的中断实现机理以及Linux下中断如何被使用。

参考资料



深入理解Linux内核 第三版第四章

Linux内核设计与实现 第三版第七章

http://www.wowotech.net/ ，蜗窝科技网站关于中断有一系列的文章，非常详细的介绍了中断的方方面面。

带着思考离开

中断返回除了返回现场外，从源代码角度分析内核还做了什么？

谢谢大家！

THANK YOU