

操作系统内核

- 基于Linux

第6讲

中断处理

主讲：杨文川

内容

- 1 中断机制概述
- 2 中断处理机制
- 3 中断下半部处理机制
- 4 时钟中断机制
- 5 实践-中断上半部的代码分析及应用

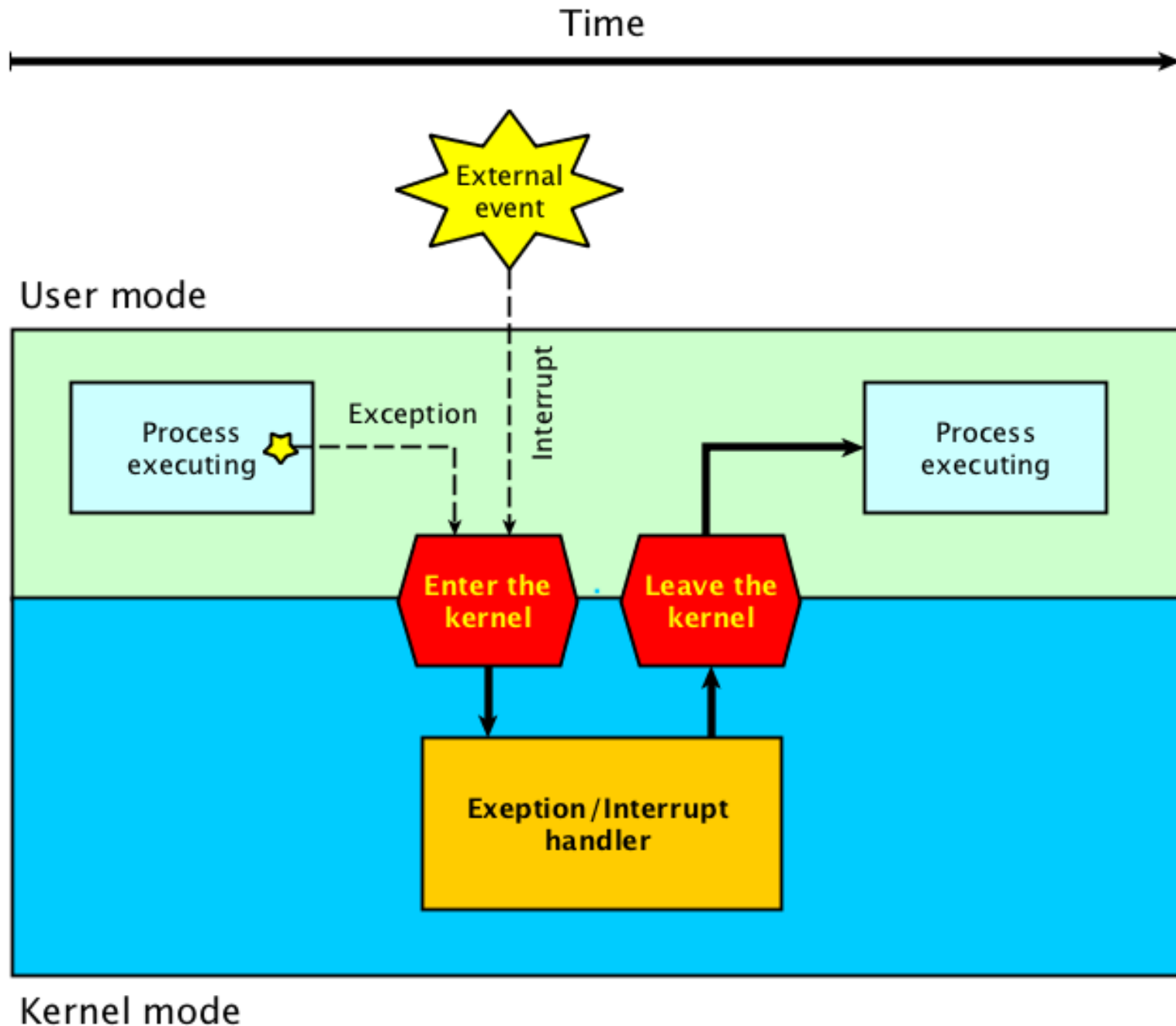


内容导航：

1 中断机制

中断是什么？

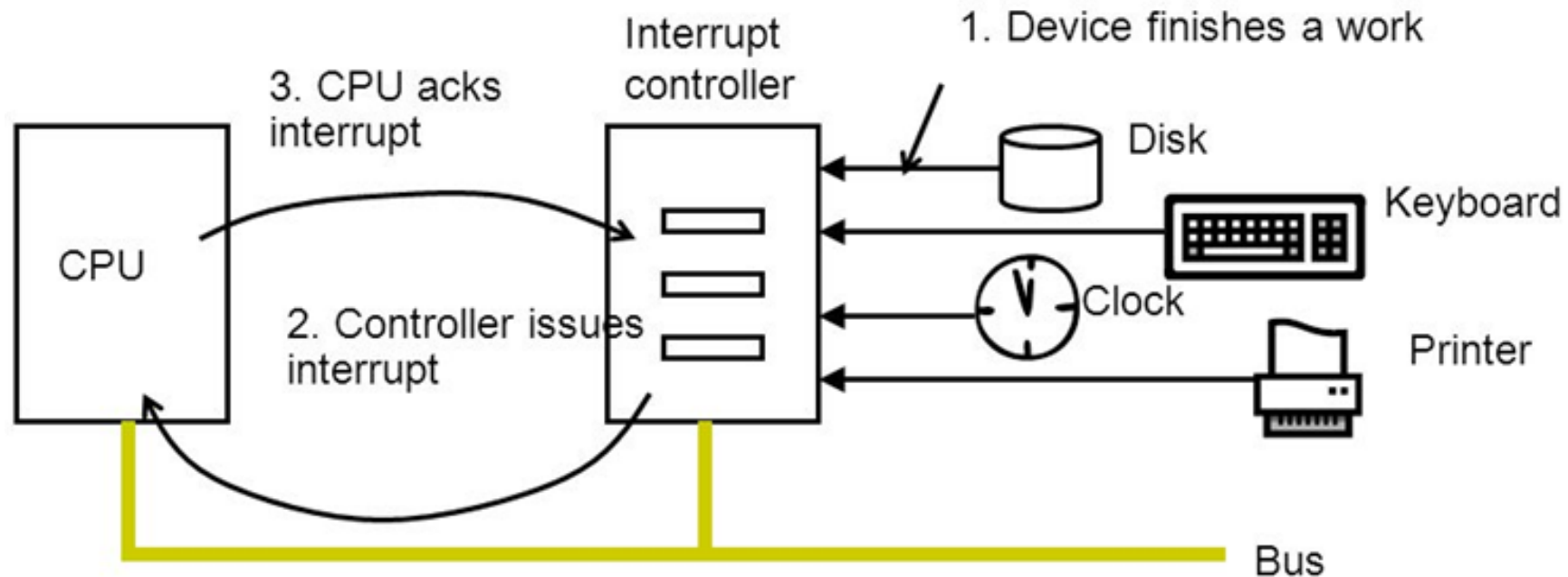
- 中断是CPU对系统发生的某个事件作出的一种反应
- 当中断发生时，CPU暂停正在执行的程序，保留现场后，自动转去执行相应事件的处理程序，
- 处理完成后返回断点，继续执行被打断的程序。



为什么引入中断

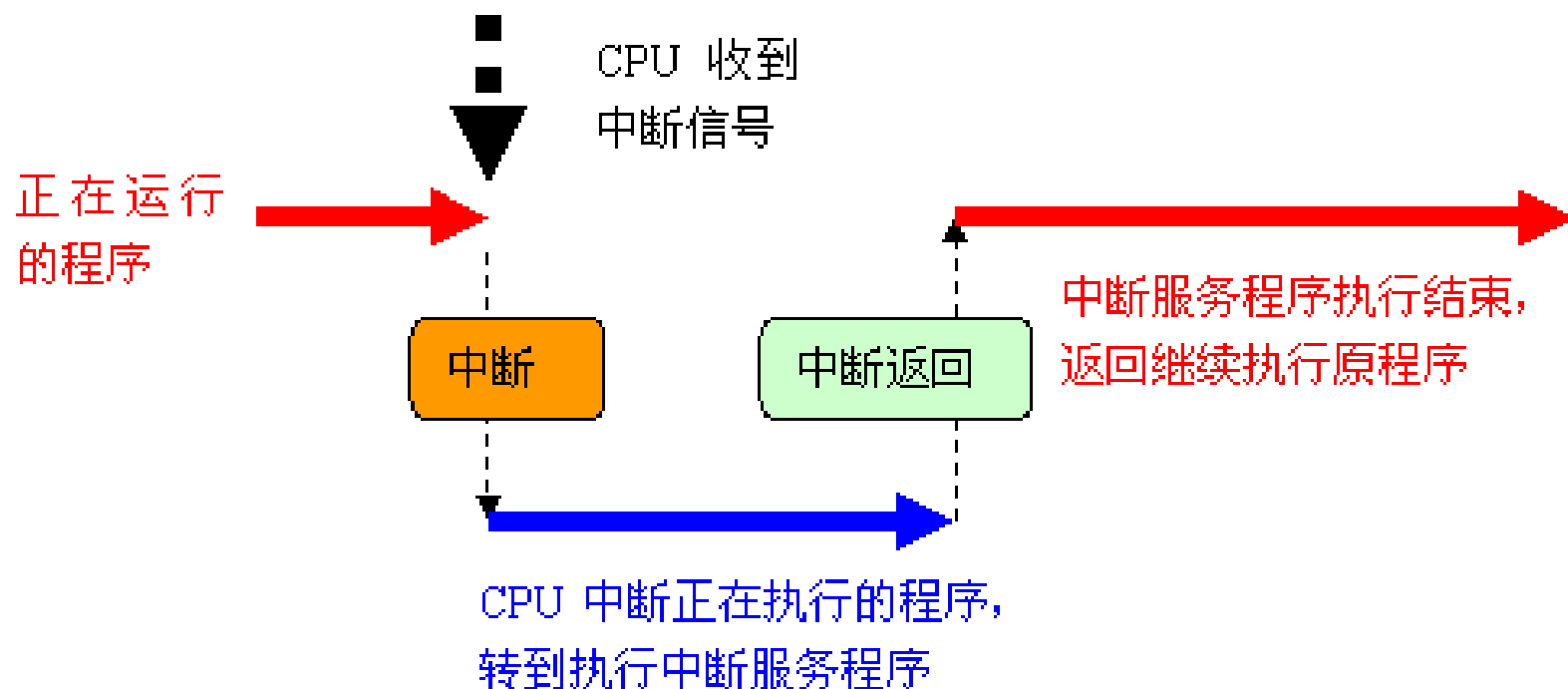
- 中断的引入，是为了支持CPU和设备之间的并行操作。为什么可以并行？
- 当CPU启动设备进行输入/输出后，设备便可以独立工作，CPU转去处理自己的事情；当设备完成输入/输出后，通过向CPU发中断报告此次输入/输出的结果，让CPU决定如何处理以后的事情，如图所示。

Interrupts



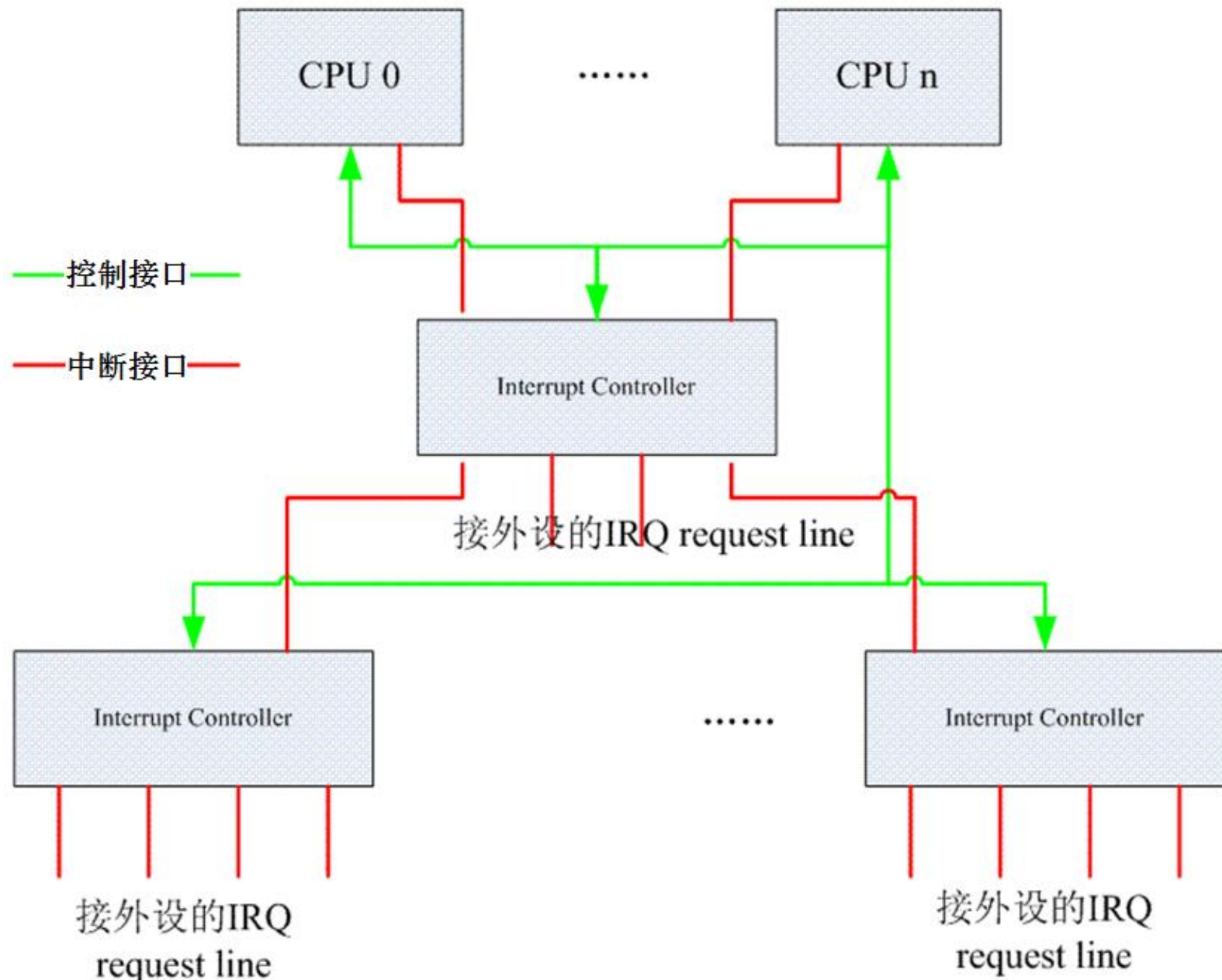
CPU什么时候响应中断

- CPU收到中断信号，并不立即响应，
- 而是在执行每条指令周期的最后一个时钟周期，一旦检测到中断信号有效，并且中断允许标志置1时，cpu才在当前指令执行完后，转入中断响应周期。



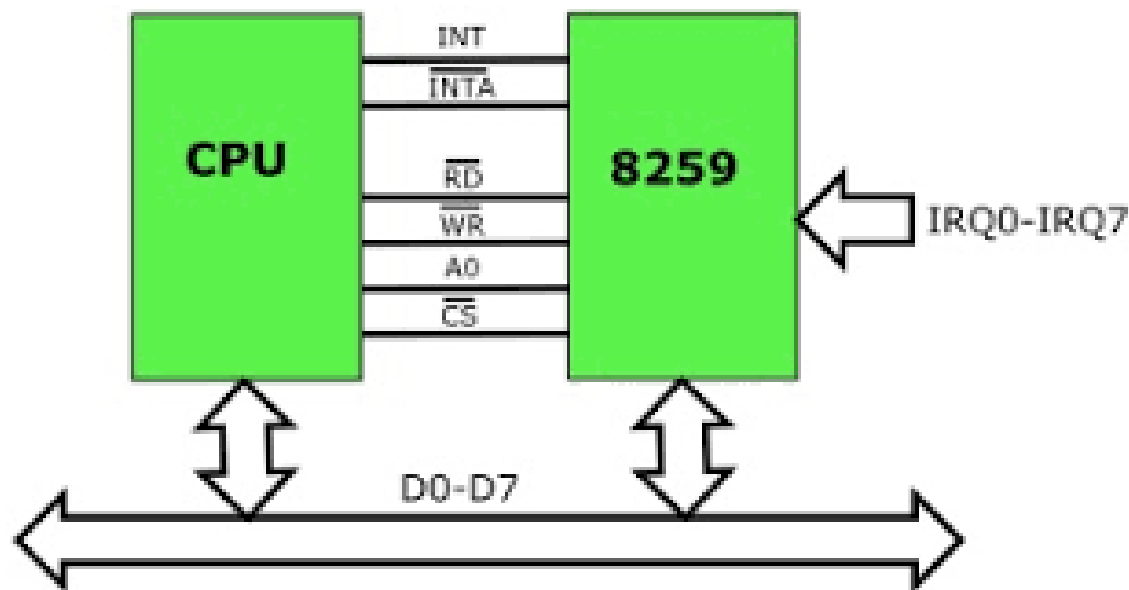
中断模型解释

- 系统中有若干个CPU用来接收中断事件并进行处理，
- 若干个中断控制器形成树状的结构，汇集系统中所有外设的中断请求线，并将中断事件分发给某一个CPU进行处理。
- 此模型实际上是一种C/S模型，外设发出请求，这个请求并不是马上传给CPU，而是由中断控制器进行收集，
- 中断控制器相当于中介，在外设与CPU之间架起桥梁。
- 当CPU接受到请求后，给予应答。



作为中介的中断控制器

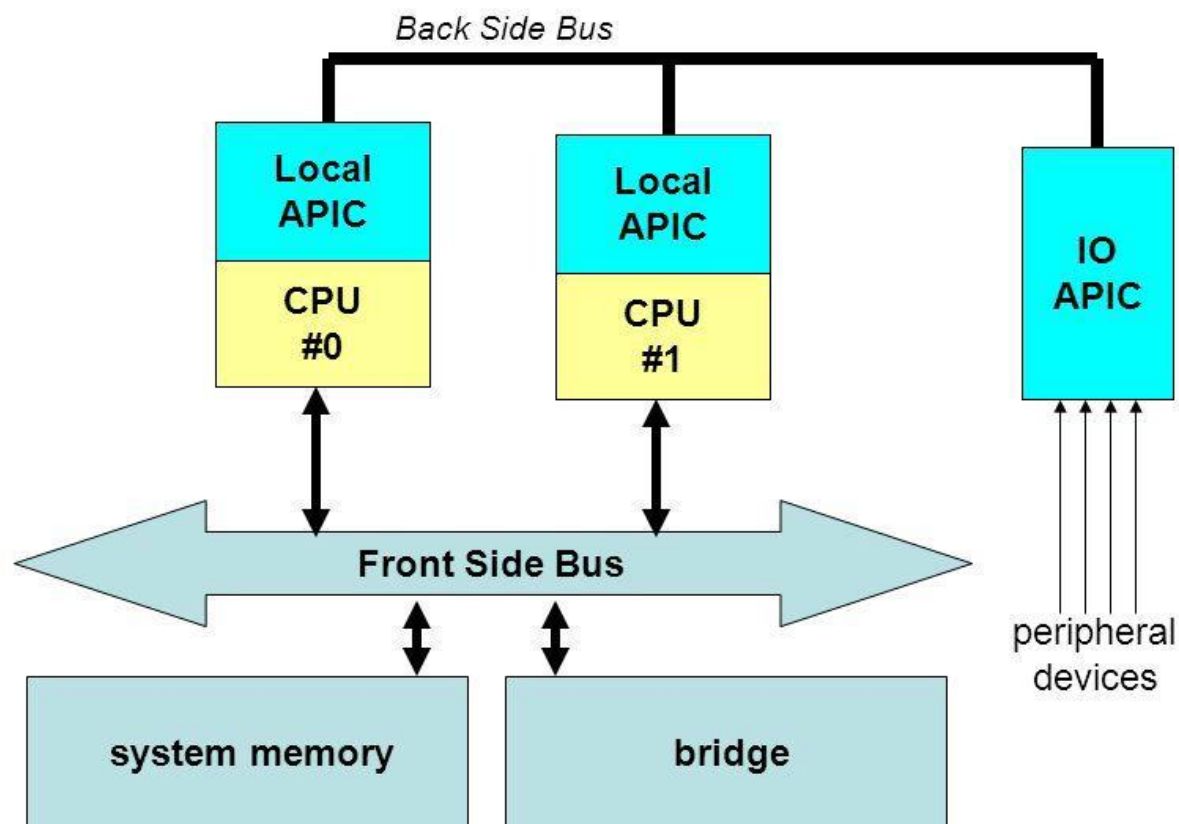
- 中断控制器此处不详述，下面以8259中断控制器为例，说明中断请求IR0~IR7是从外设发来的，然后通过中断控制器的INT引脚向CPU发出中断请求，然后CPU通过中断应答引脚INTA应答请求。
- 目前x86采用的是APIC(高级可编程控制器)。



以X86的8259中断控制器为例

高级可编程中断控制器(APIC)

Multiprocessor topology



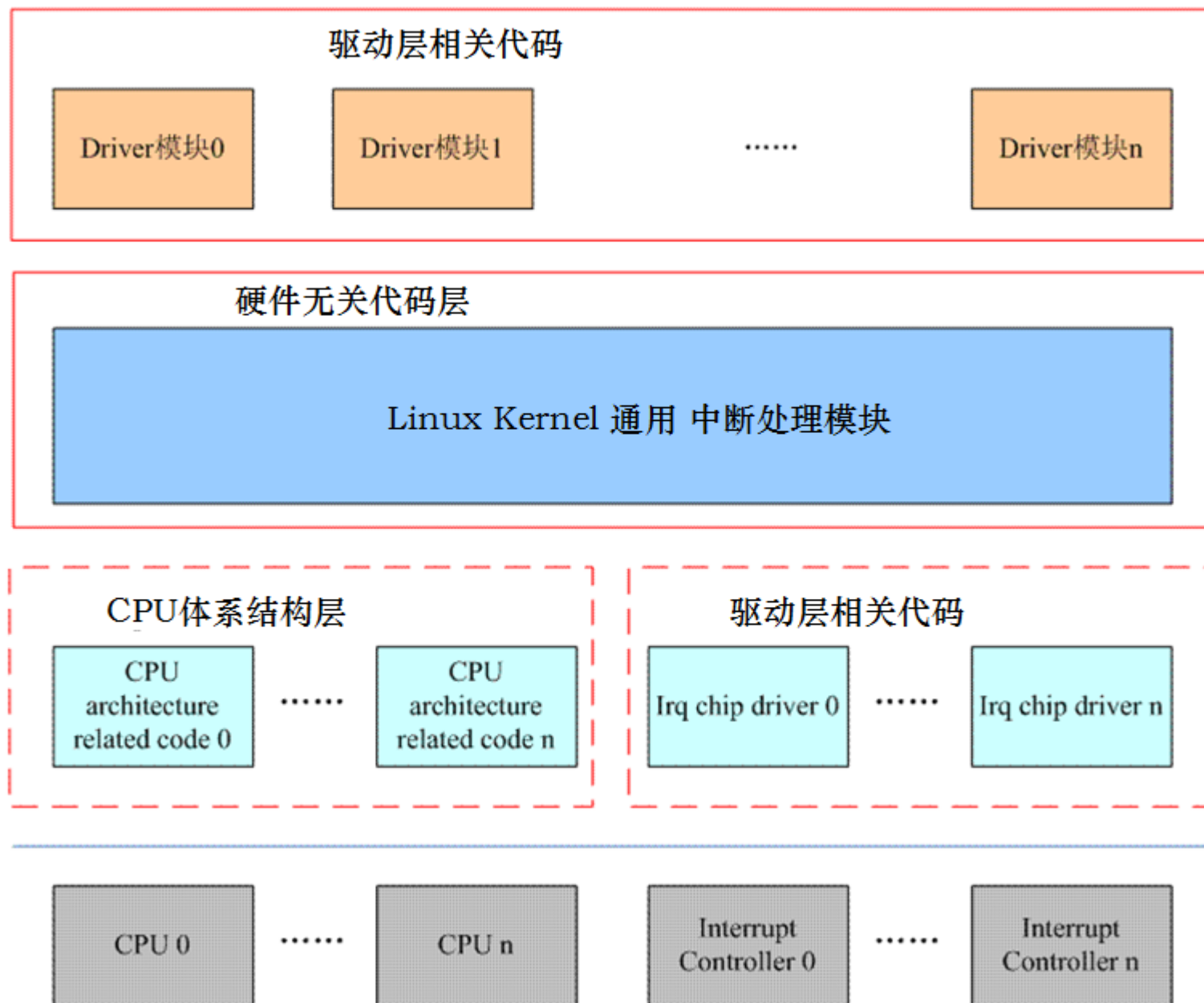
- 每个x86的核有一个本地APIC，这些本地APIC通过中断控制器通信(Interrupt Controller Communication)总线，连接到IO APIC上。
- IO APIC收集各个外设的中断，并翻译成总线上的信息，传递给某个CPU上的本地APIC。

机制与策略分离的中断机制

- 1. 尽管中断与CPU密切相关，但是CPU的设计，独立于中断控制器的设计。
- 2. 尽管中断是操作系统非常重要的组成部分，但是操作系统的设计者只负责提供接口，通过该接口，可以调用针对具体设备的中断服务程序。
- 3. 中断和对中断的处理，被解除了耦合。无论是你在需要加入新的中断时，还是在你需要改变现有中断的服务程序时、又或是取消对某个中断支持的时候，CPU架构和操作系统都无需作改变，这其中的功臣就是中断控制器。

Linux内核中断子系统分成4个部分

- (1)硬件无关的代码，也称为Linux 内核 通用中断处理模块。
- (2)CPU 体系结构相关的中断处理。
- (3)中断控制器的驱动代码。
- (4)普通外设的驱动。
- 下面将对这几部分做一个介绍。



Linux内核中断子系统各部分的详述

- (1)硬件无关的代码，称之为 Linux 内核 **通用中断处理模块**。
- 无论是哪种CPU，哪种中断控制器，其中断处理的过程都有一些相同的内容，这些相同的内容被抽象出来。此外，各个外设的驱动代码中，也希望能用一个统一的接口，实现中断相关的管理。
- (2)CPU 体系结构相关的中断处理。
- 和系统使用的具体的CPU 体系结构相关。
- (3)中断控制器的驱动代码。
- 和系统使用的中断控制器相关。
- (4)普通外设的驱动。
- 这些驱动将使用Linux 内核 通用中断处理模块 的 API，来实现自己的驱动逻辑。

中断向量 - 中断源的类型

向量范围	用途
0~19	不可屏蔽中断和异常
20~31	Intel保留
32~127	外部中断 (IRQ)
128 (0x80)	用于系统调用的可编程异常
129~238	外部中断
239	本地APIC时钟中断
240	本地APIC高温中断
241~250	Linux保留
251~253	处理器间中断
254	本地APIC错误中断
255	本地APIC伪中断



· 中断向量 - 每个中断源，都被分配一个8位无符号整数作为类型码，即中断向量

· 中断向量和中断请求号之间的关系如下：

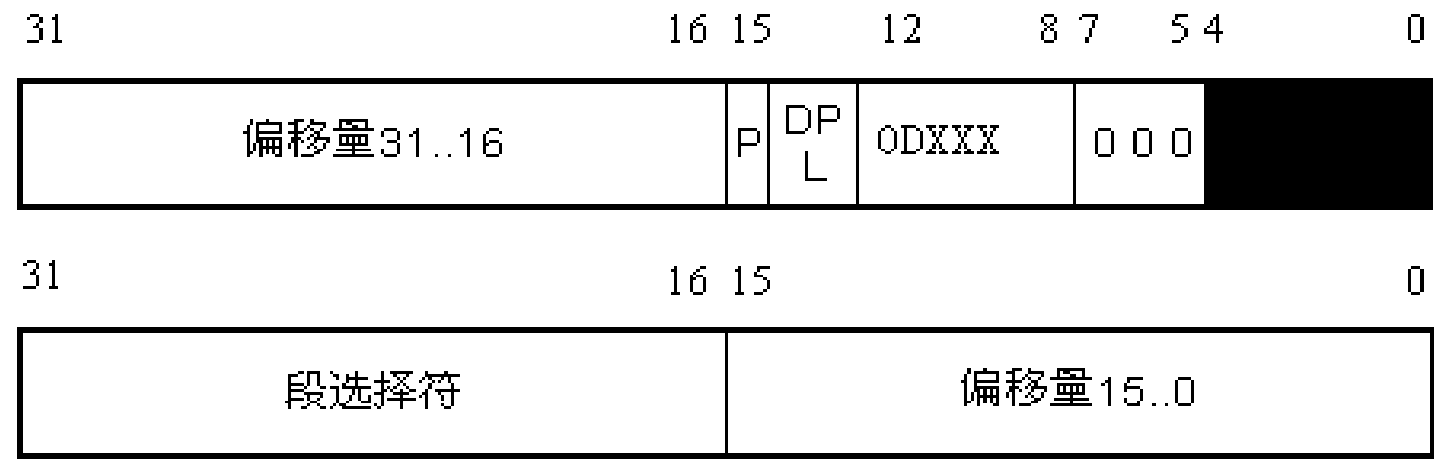
· 中断向量 $I=f(irq)$
• (x86中: $I=32+irq$)

中断描述符表

- **中断描述符表**(IDT)-中断向量表，每个中断占据一个表项
- 在实地址模式中，CPU把内存中从0开始的1K字节，作为一个中断向量表。
- 表中的每个表项占四个字节，由两个字节的段地址和两个字节的偏移量组成，这样构成的地址，便是相应中断处理程序的入口地址。
- 在保护模式下，由四字节的表项构成的中断向量表，显然满足不了要求。这是因为：
 - 除了两个字节的段描述符，偏移量必用四字节来表示；
 - 要有反映模式切换的信息。
- 因此在保护模式下，中断向量表中的表项由8个字节组成

中断描述符表

- 如图所示，中断向量表 也称为 中断描述符表IDT(Interrupt Descriptor Table)。
- 其中的每个表项叫做一个 门描述符(gate descriptor)。

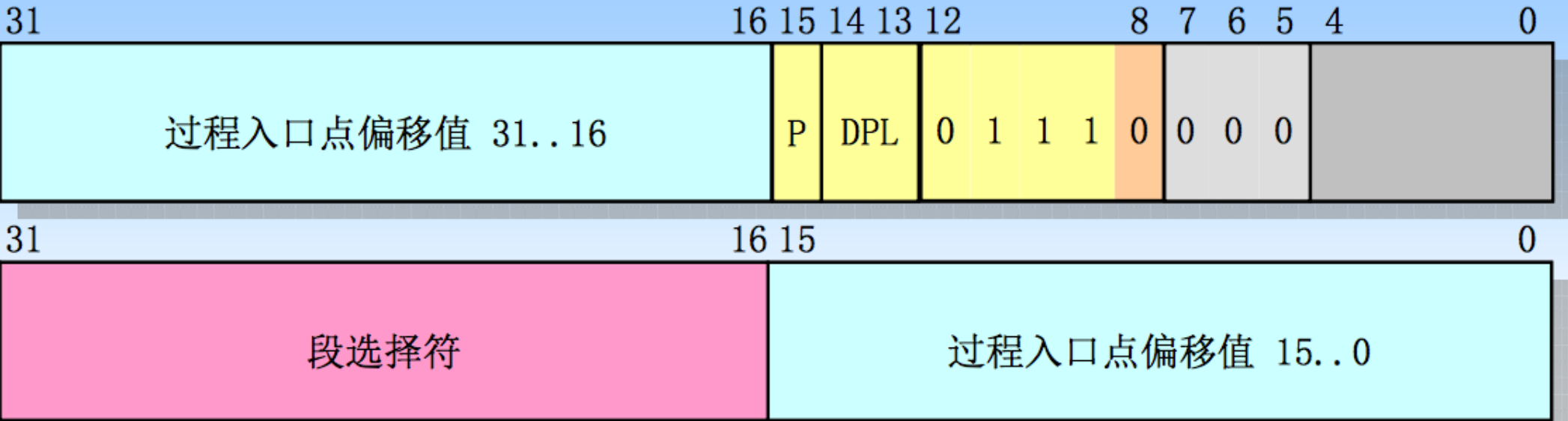


- DPL 段描述符的特权级
- 偏移量 入口函数地址的偏移量
- P 段是否在内存中的标志
- 段选择符 入口函数所处代码段的选择符
- D 标志位, 1=32位, 0=16位
- XXX 3位门类型码

中断描述符表中的门

- “门” 的含义是当中断发生时必须先通过这些门，然后才能进入相应的处理程序。其中类型占3位，表示门描述符的类型，主要门描述符为：
- (1)中断门(Interrupt gate)
- 其类型码为110，中断门中的请求特权级(DPL)为0，因此，用户态的进程不能访问Intel的中断门。所有的中断处理程序，都由中断门激活，并全部限制在内核态。

中断门（Interrupt Gate）



中断描述符表中的门

- (2)陷阱门(Trap gate)
- 其类型码为111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时，维持中断标志位(IF)不变，也就是说，不关中断。
- (3)系统门(System gate)
- 这是Linux内核特别设置的，用来让用户态的进程访问Intel的陷阱门，因此，门描述符的DPL为3。系统调用就是通过系统门进入内核的。

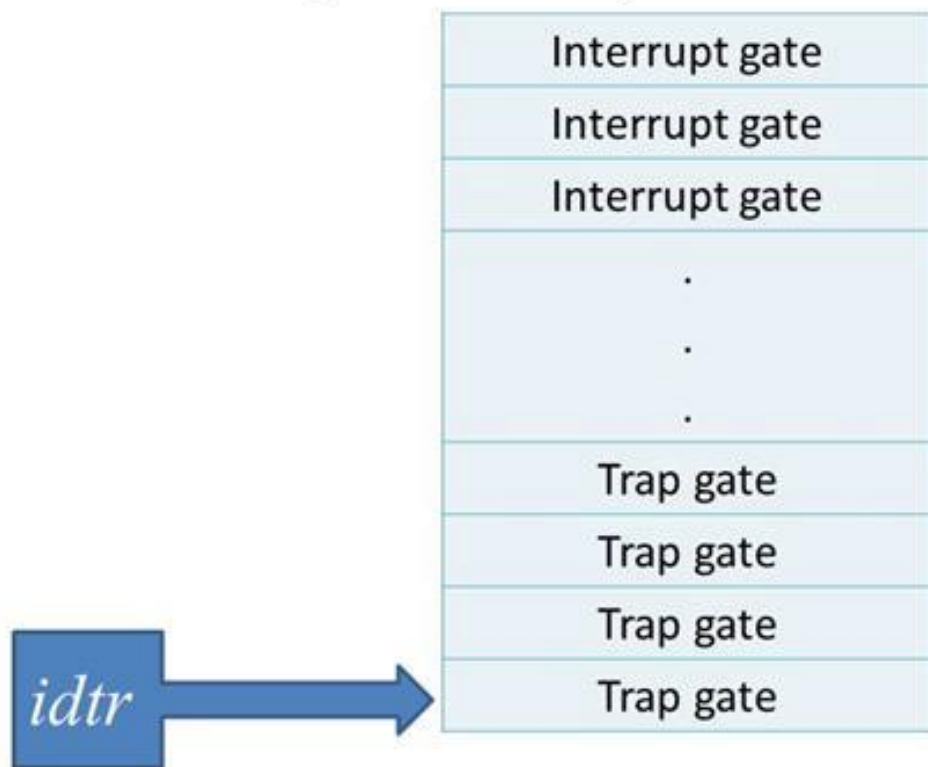


内容导航：

2 中断处理机制

中断描述表以及初始化

Interrupt descriptor table (IDT)

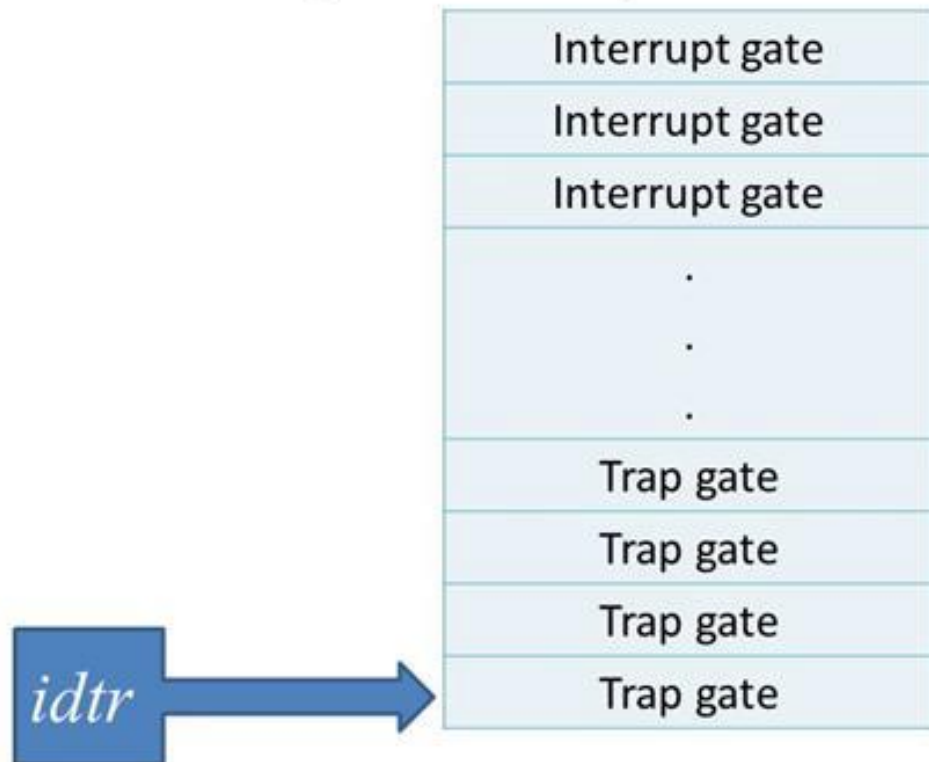


idtr: interrupt descriptor table register

- 前面介绍了中断描述符表，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，
- 其起始地址放在中断描述符表寄存器 (IDTR) 中

中断描述符表相关源代码

Interrupt descriptor table (IDT)

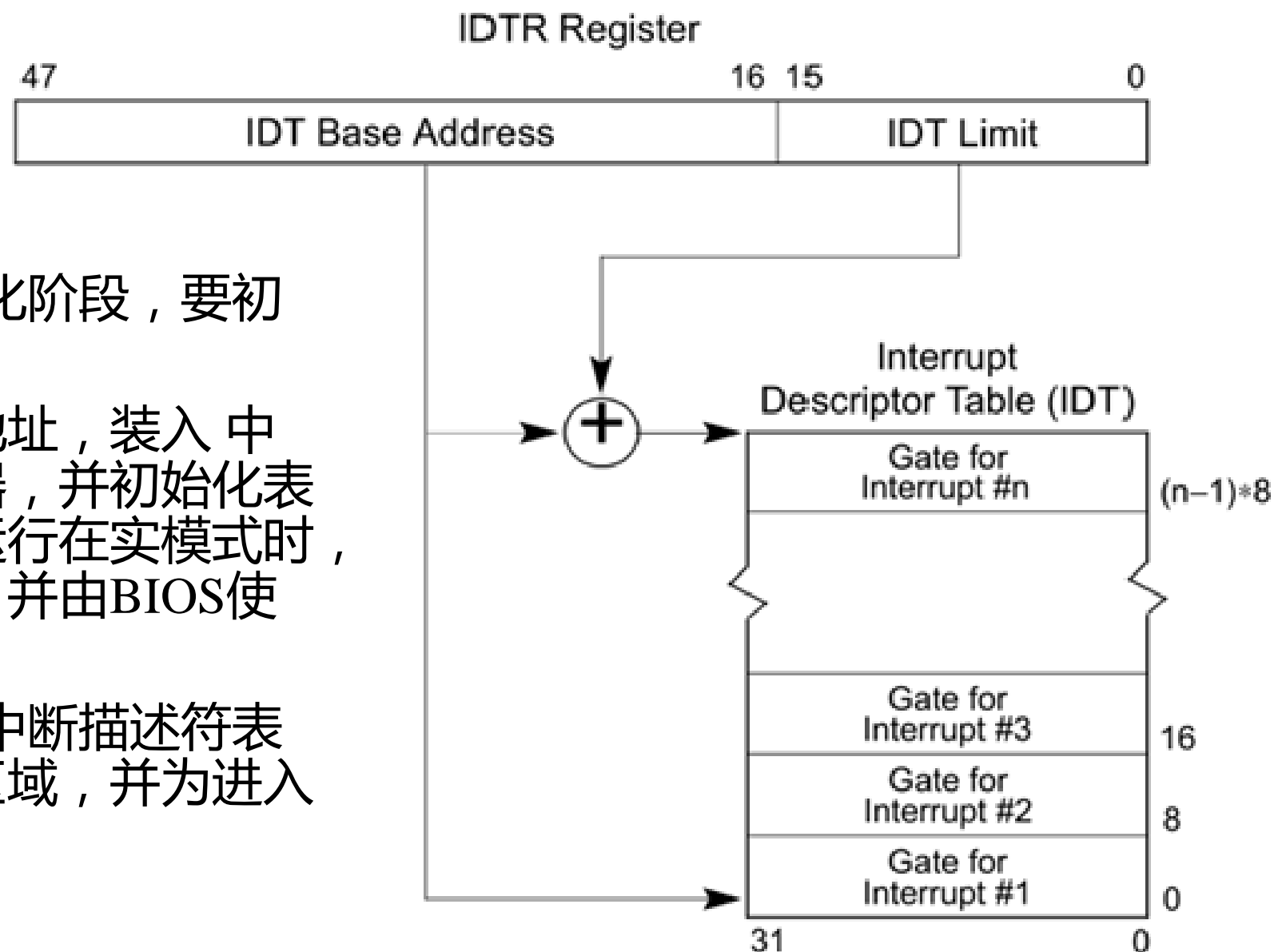


idtr: interrupt descriptor table register

- `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

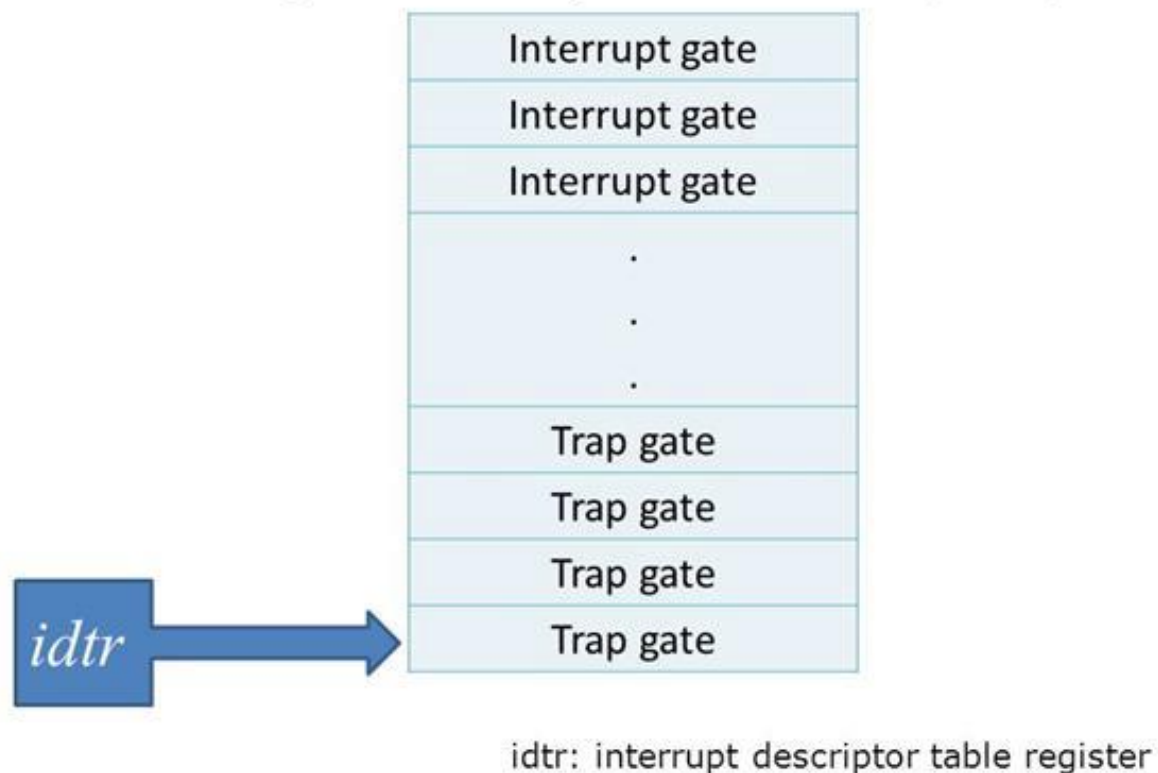
初始化中断描述符表

- 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 中断描述符表(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

Interrupt descriptor table (IDT)



- `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,÷_error);`
 - `set_trap_gate(1,&debug);`
 - `set_trap_gate(19,&simd_coprocessor_error);`
 - `set_system_gate(SYSCALL_VECTOR,&system_call);` (初始化系统门)

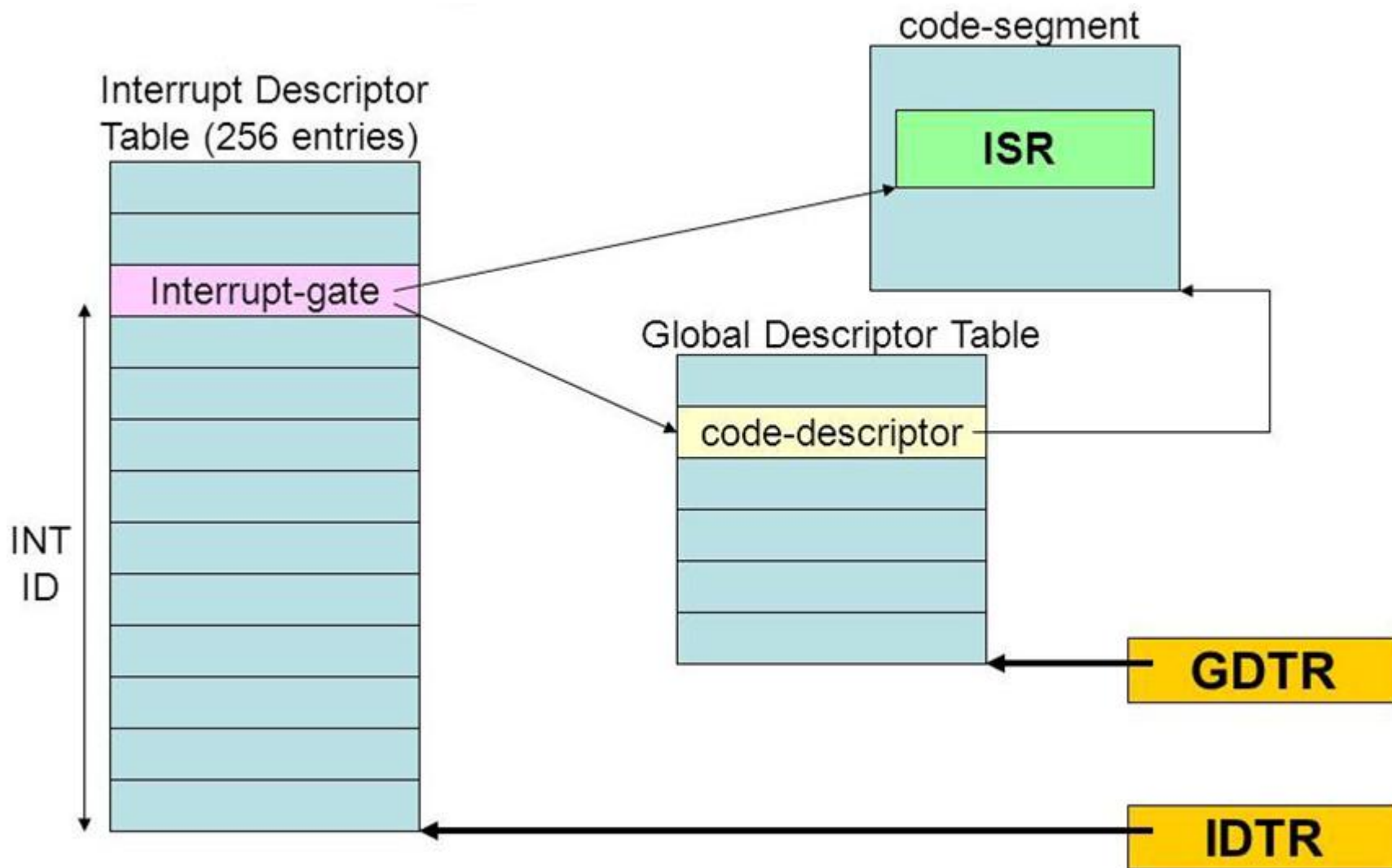
中断门的设置

- 中断门的设置是由init_IRQ()函数中的一段代码完成的
- 设置时必须跳过用于系统调用的向量0x80


```
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

- 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**)

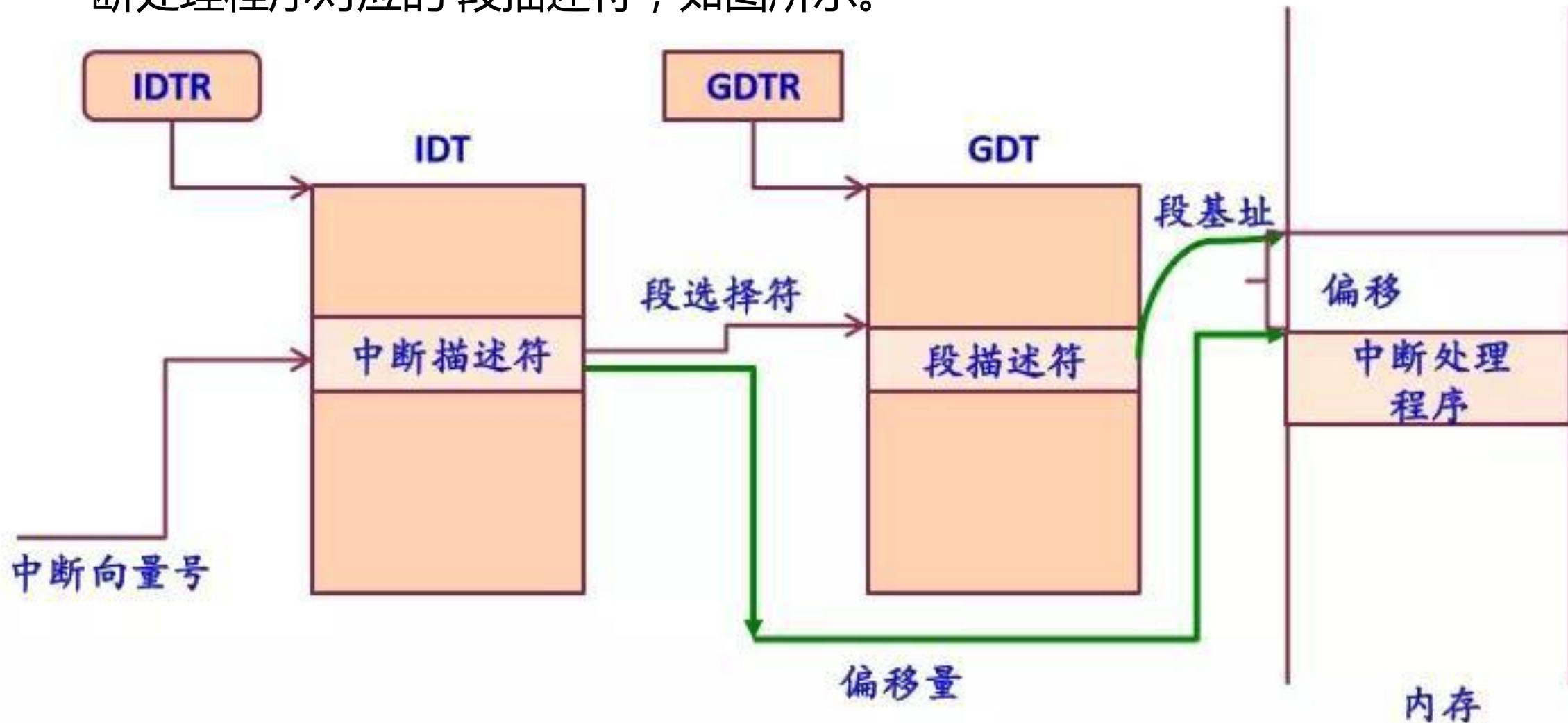
从中断返回

X86中的中断处理

- 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)

X86中的中断处理

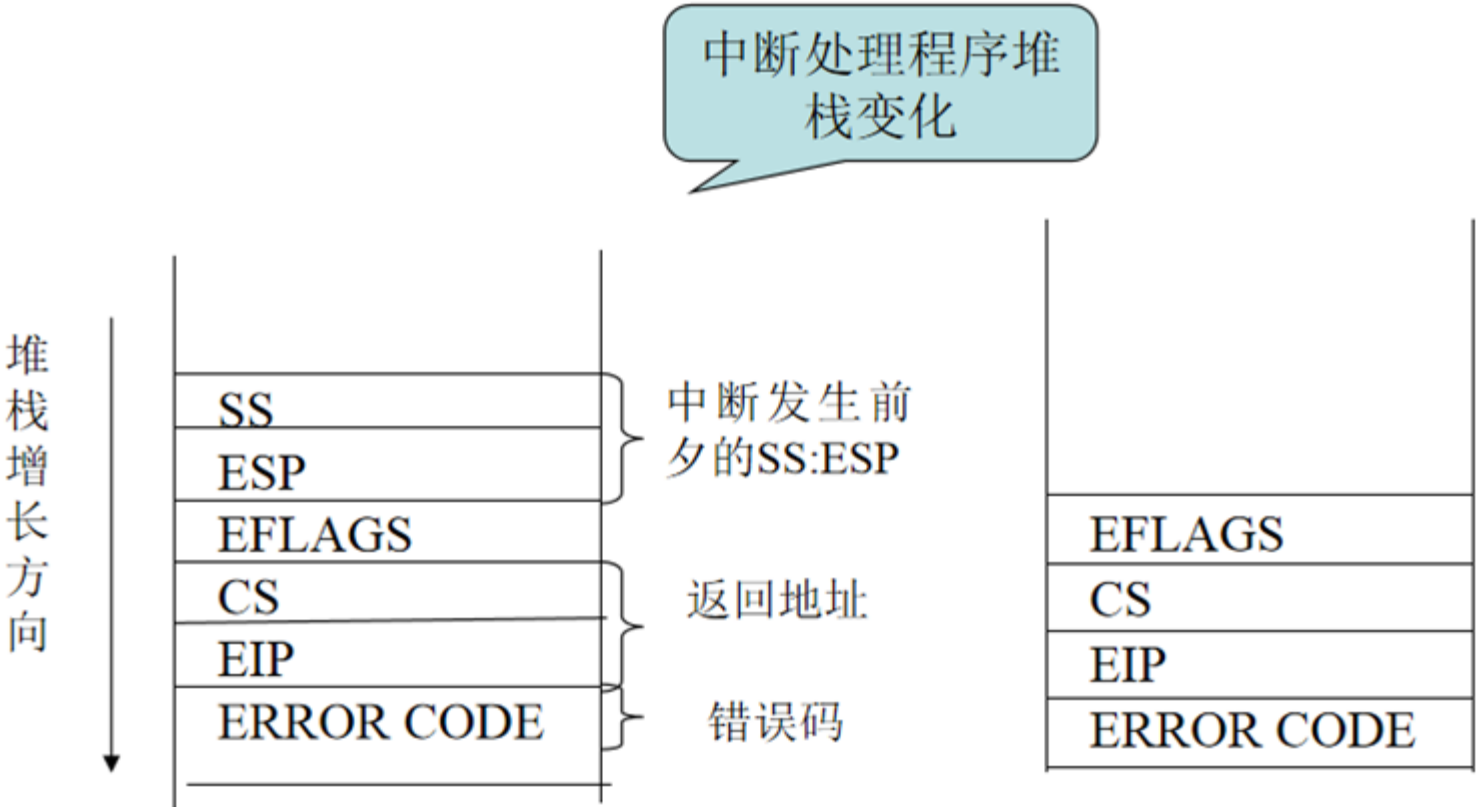
- 2.通过IDTR寄存器找到IDT表，读取IDT表第i项(或叫第i个门)，从GDTR寄存器获得GDT的地址；结合中断描述符中的段选择符，在GDT表中获得中断处理程序对应的 段描述符，如图所示。



中断处理过程堆栈变化

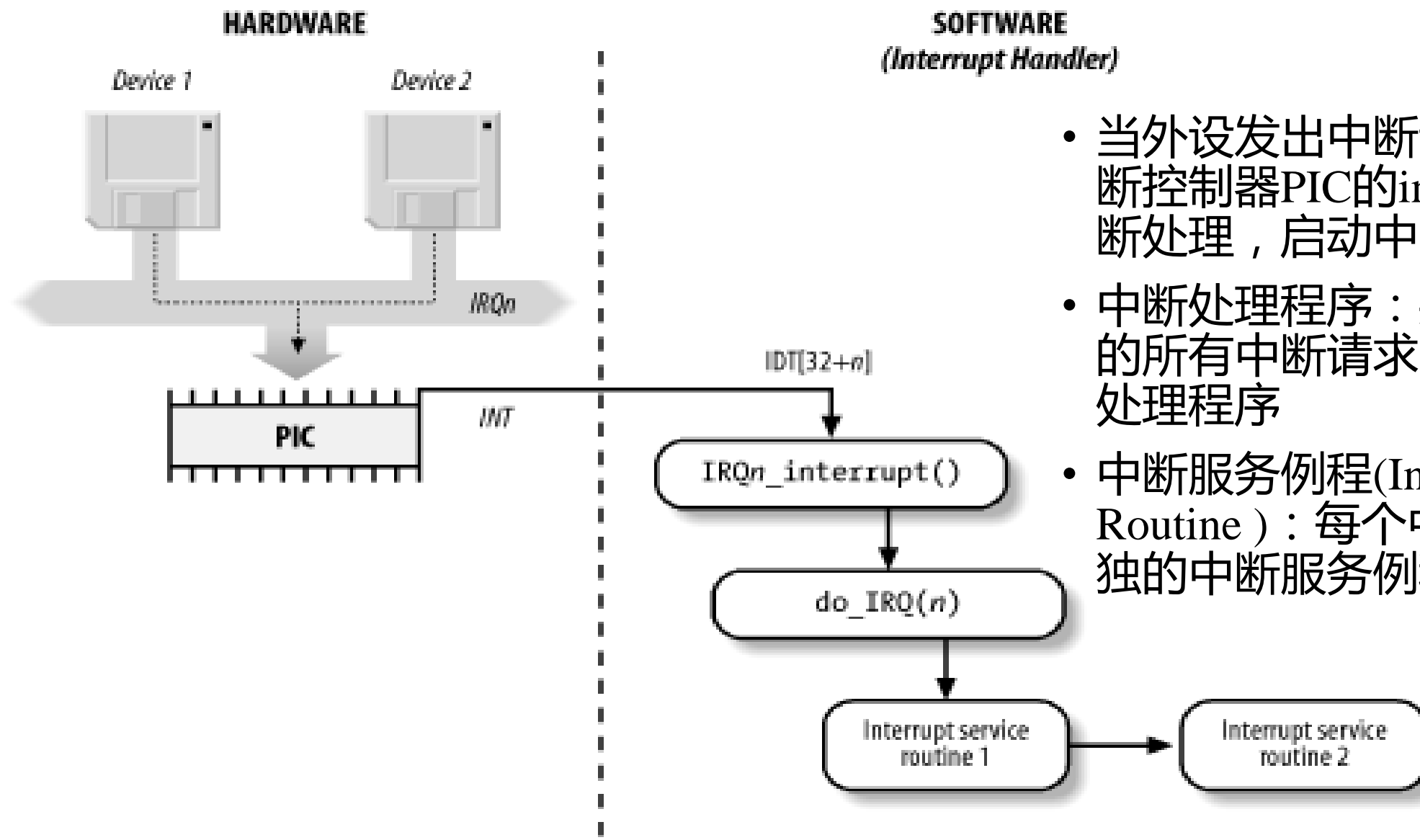
- 从该 段描述符 中获得 中断/异常处理程序 所在的段基址，与其偏移量相加，得到 中断处理程序的入口地址。
- 其中要进行“段”级、“门”级两步有效性检查，还要检查是否发生了特权级的变化。
- 当中断发生在用户态(特权级为3)，而 中断处理程序 运行在内核态(特权级为0)，特权级发生了变化，所以会引起堆栈的更换。也就是说，从 用户堆栈 切换到 内核堆栈。
- 而当中断发生在内核态时，即CPU在内核中运行时，不会更换堆栈。

中断处理过程堆栈变化



- 由图可看出，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，再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。

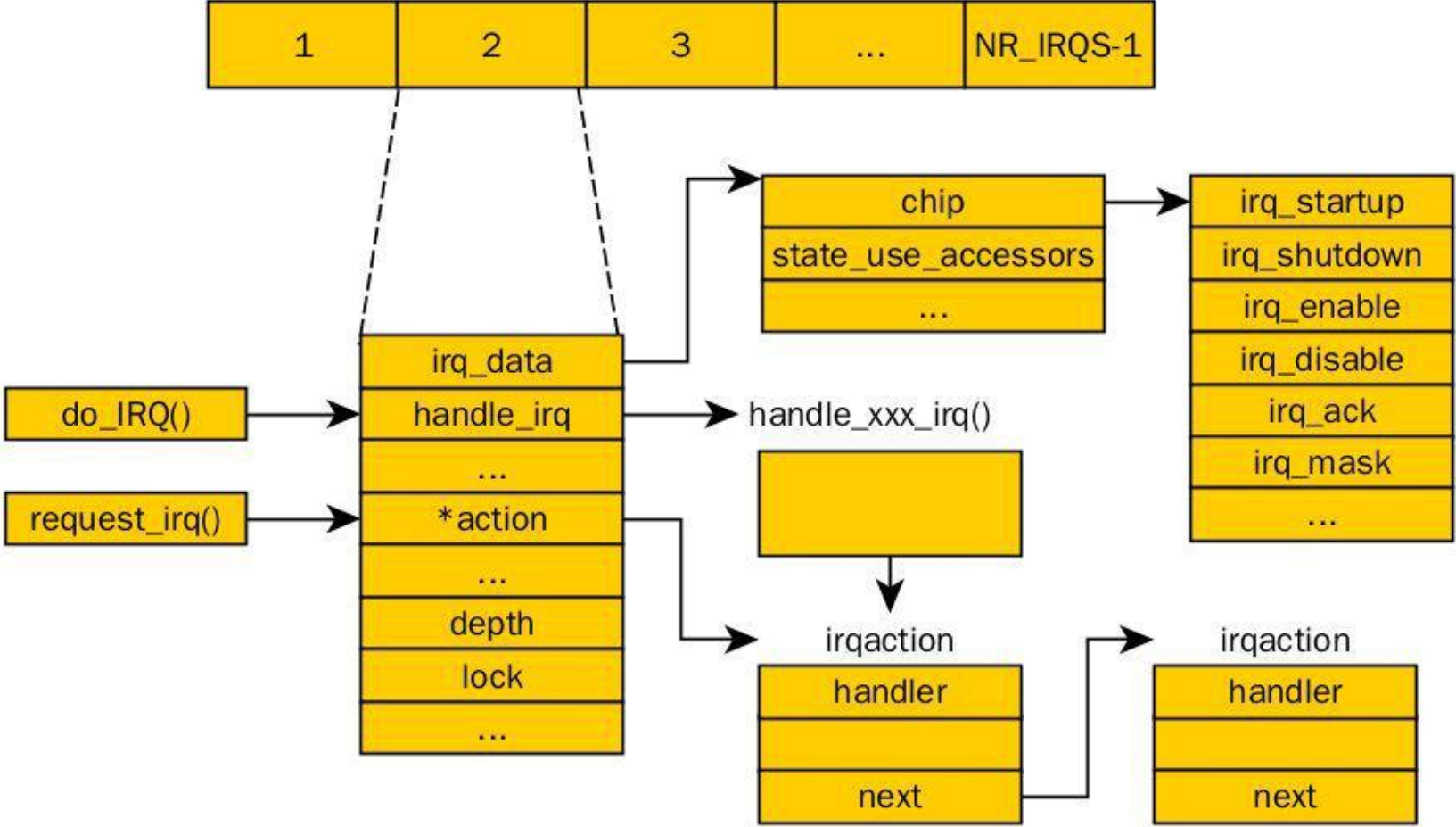
中断处理程序与中断服务例程



- 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，启动中断处理机制，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，如图所示。

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，称为 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;  
};
```

在include/linux/interrupt.h

- 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  
)
```

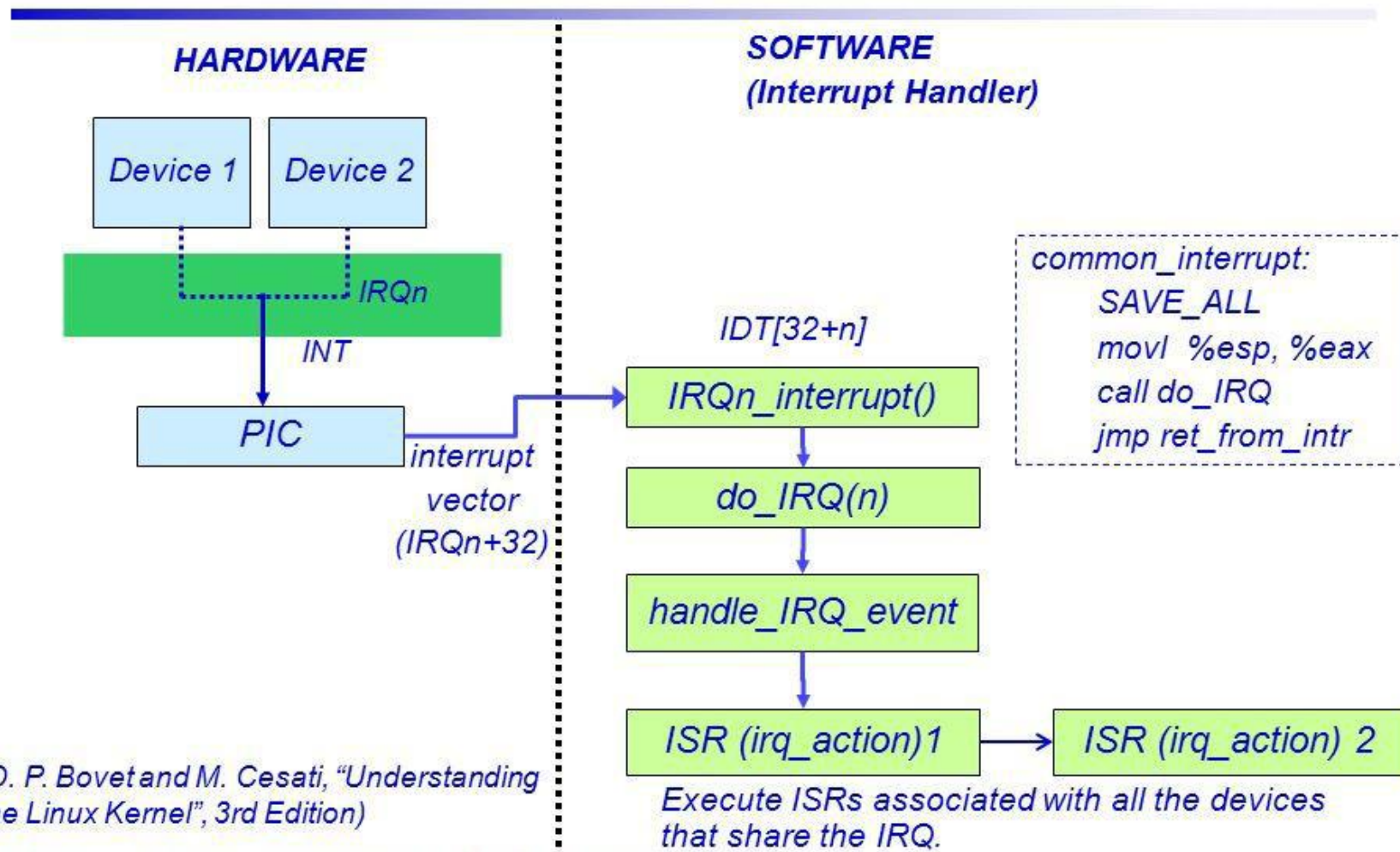
在include/linux/interrupt.h

- 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，如图所示。

I/O Interrupt Handling

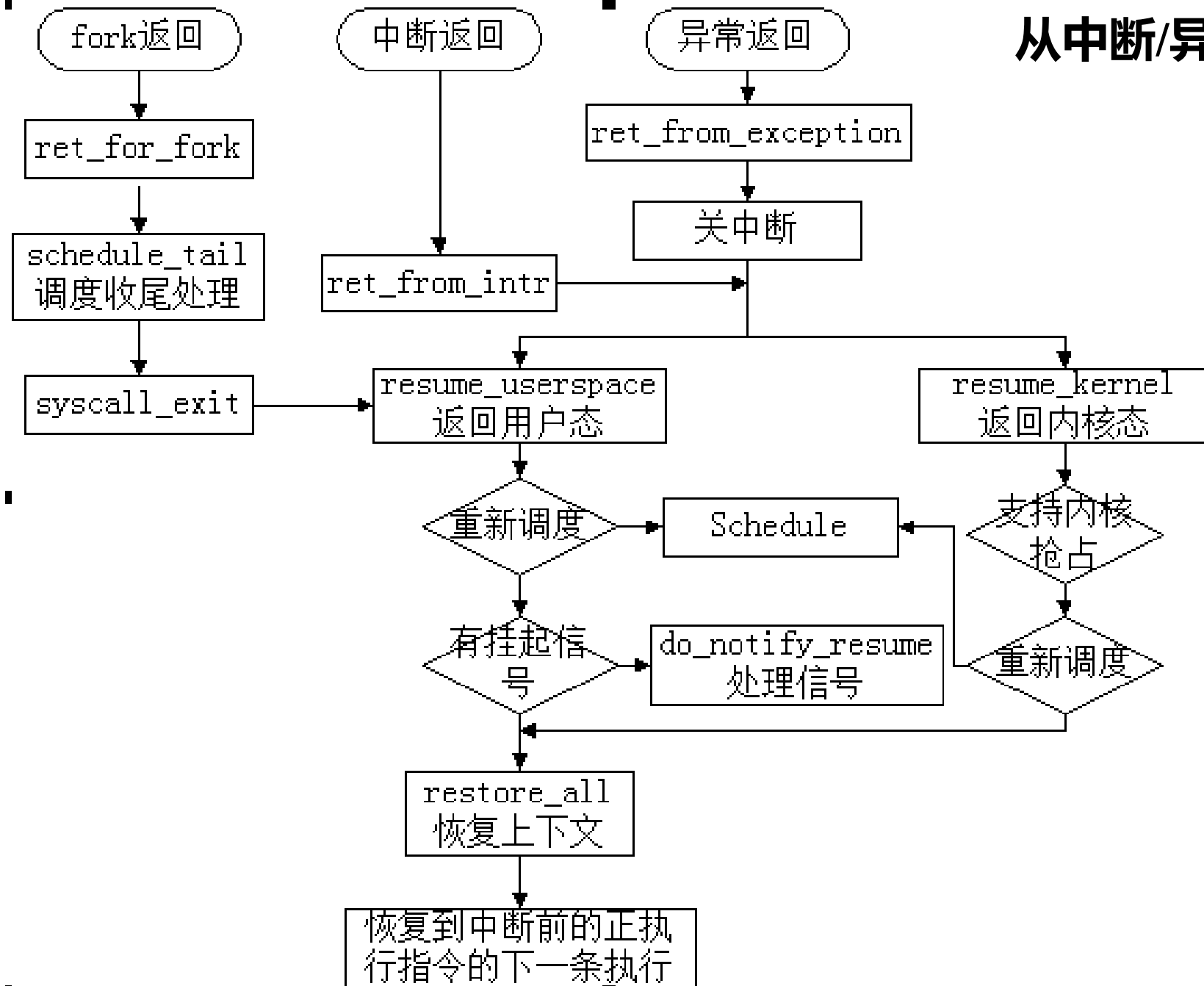


中断返回(ret_from_intr)

```
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
- RESTORE_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



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。

从中断/异常/系统调用返回

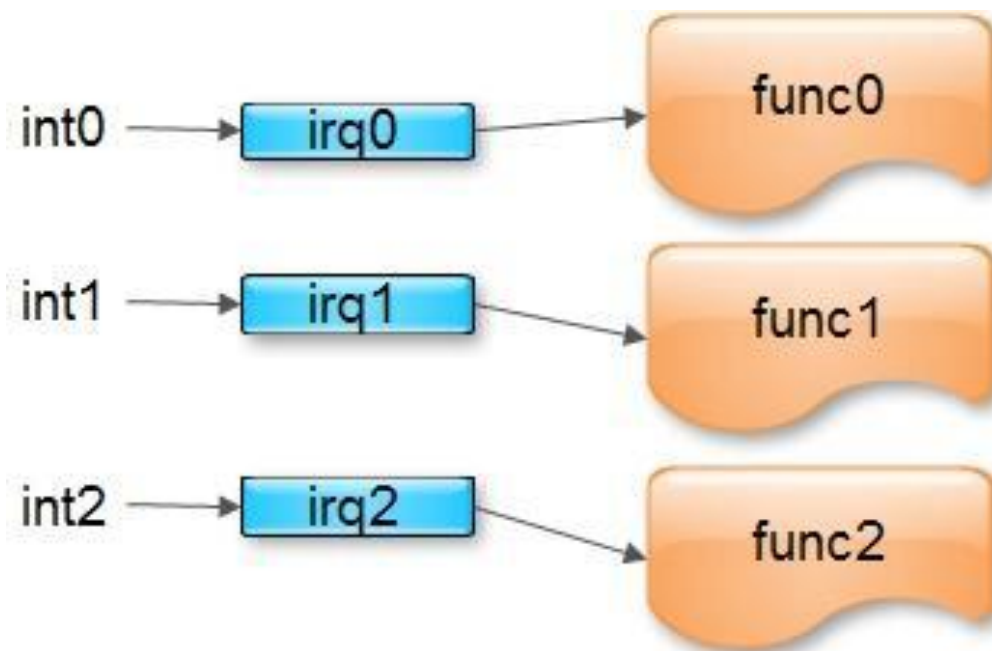
- 3)信号处理是在当前进程从内核态返回用户态时进行的，在发生中断、异常(包括系统调用)、或fork时，都有可能从内核态返回用户态，都是处理信号的时机。
- 注意：只有当前进程的信号才能在此时得到处理。其它非正在运行的进程的信号无法处理。



3 中断下半部处理机制

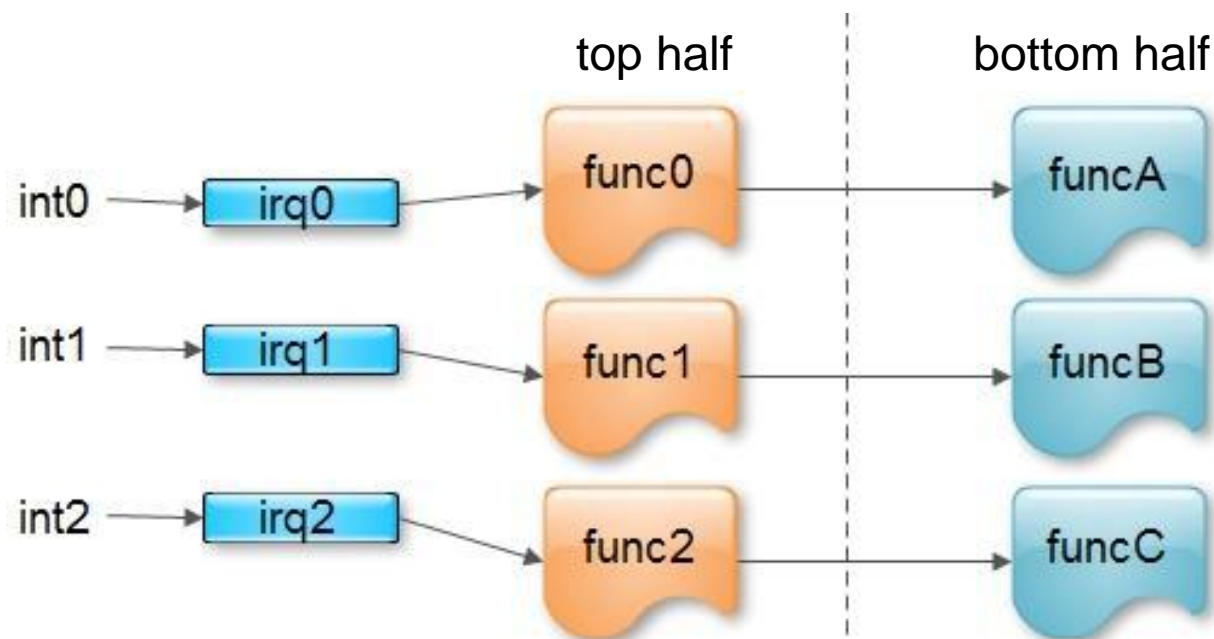
中断的基本机制

- 中断服务程序一般都是在中断请求关闭的条件下执行的，以避免嵌套而使中断控制复杂化。
- 但是，中断是一个随机事件，它随时会到来，如果关中断的时间太长，CPU就不能及时响应其他的中断请求，从而造成中断的丢失。
- 因此，内核的目标就是尽可能快的处理完中断请求，尽其所能把更多的处理向后推迟。
- 右图是中断的基本模型，在中断向量表中填入中断处理程序的入口地址，然后跳到该程序执行。



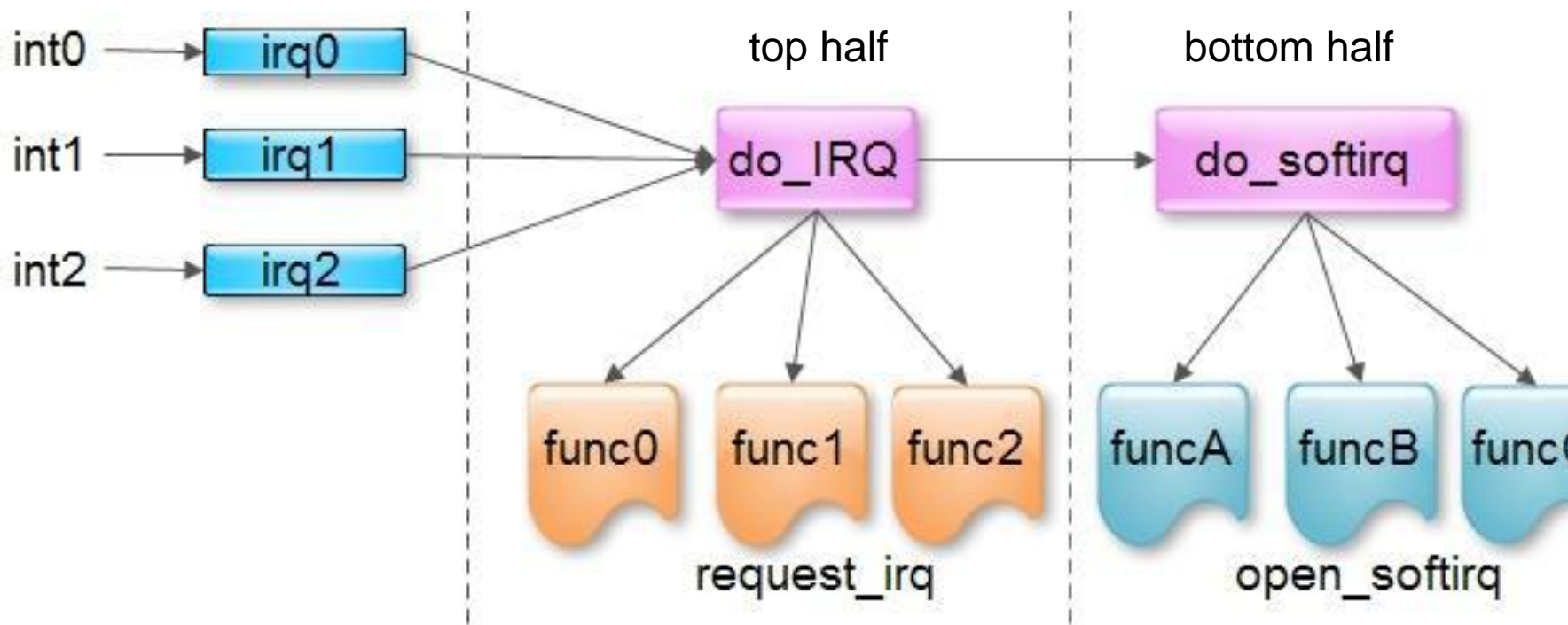
中断的下半部

- 随着系统的不断复杂，中断处理函数要做的事情也越来越多，多到都来不及接收新的中断了，于是发生了中断丢失，这显然不行！
- 为此内核把中断处理分为两部分：
- 上半部(top half)不可中断 和 下半部(bottom half)可中断，
- 上半部(也就是中断服务程序)内核立即执行，而下半部(就是一些内核函数)留着稍后处理



软中断机制

- 不管是中断的上半部，还是下半部，都是一种概念，实际上它们都是内核中的一个函数，这些函数写好后，什么时候执行，如何执行，这是内核必须统一管理的。
- 在中断机制中，介绍了中断注册函数`request_irq()`把中断服务例程添加到中断请求队列中。其执行是由`do_IRQ`完成的。与之对应，通过`open_softirq`添加下半部对应的处理函数。而对其执行则是通过`do_softirq`，也就是软中断机制完成的。



软中断类型

- 下半部的处理方式主要有soft_irq , tasklet , workqueue三种 ,
- 他们在使用方式和适用情况上各有不同。
- soft_irq用在对下半部执行时间要求比较紧急的场合 , 在中断上下文执行。
- tasklet和work queue在普通的驱动程序中用的相对较多 ,
- 主要区别是tasklet是在中断上下文执行 , 而workqueue是在进程上下文 , 因此可以执行可能睡眠的操作。

软中断类型

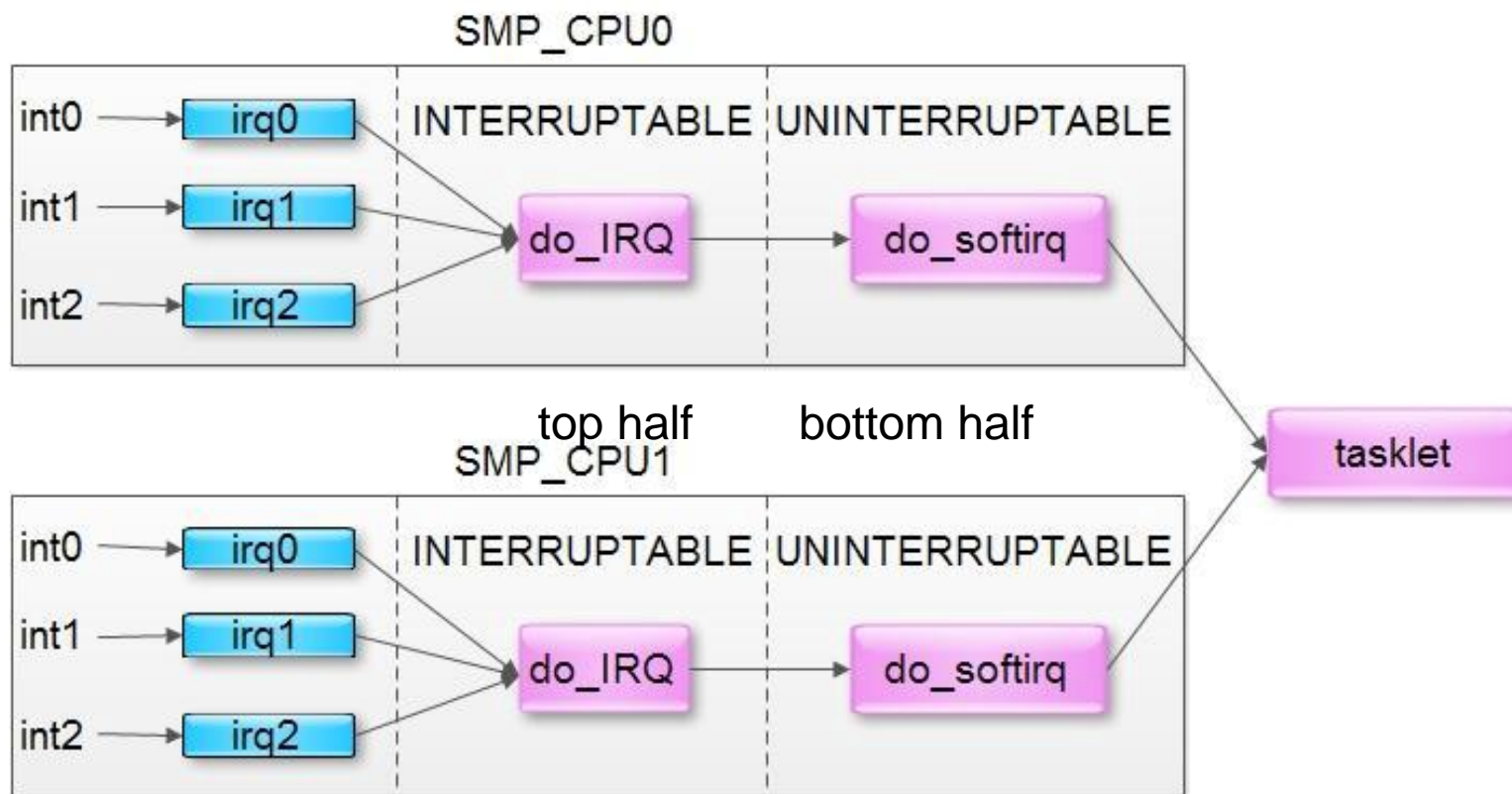
- 每个软中断在内核中以softirq_action表示，内核目前实现了10中软中断，定义在linux/interrupt.h中，如图所示。

```
1  enum
2  {
3      HI_SOFTIRQ=0,           /* 高优先级tasklet */ /* 优先级最高 */
4      TIMER_SOFTIRQ,         /* 时钟相关的软中断 */
5      NET_TX_SOFTIRQ,        /* 将数据包传送到网卡 */
6      NET_RX_SOFTIRQ,        /* 从网卡接收数据包 */
7      BLOCK_SOFTIRQ,         /* 块设备的软中断 */
8      BLOCK_IOPOLL_SOFTIRQ,  /* 支持IO轮询的块设备软中断 */
9      TASKLET_SOFTIRQ,       /* 常规tasklet */
10     SCHED_SOFTIRQ,         /* 调度程序软中断 */
11     HRTIMER_SOFTIRQ,       /* 高精度计时器软中断 */
12     RCU_SOFTIRQ,           /* RCU锁软中断，该软中断总是最后一个软中断 */
13     NR_SOFTIRQS            /* 软中断数，为10 */
14 };
```

在include/linux/interrupt.h

小任务(tasklet)机制

- 小任务(tasklet)机制是I/O驱动程序中，实现可延迟函数的首选方法；
- 小任务和工作队列是延期执行工作的机制，其实现基于软中断，但他们更易于使用，因而更适合于设备驱动程。
- 所谓小任务，就是执行一些迷你任务。



小任务数据结构

```
struct tasklet_struct {  
    struct tasklet_struct *next; /*指向链表  
中的下一个结构*/  
    unsigned long state; /* 小任务的状态 */  
    atomic_t count; /* 引用计数器 */  
    void (*func) (unsigned long); /* 要调用  
的函数 */  
    unsigned long data; /* 传递给函数的参数  
*/  
};
```

在include/linux/interrupt.h

- State域的取值为有以下两个：
 - TASKLET_STATE_SCHED，表示小任务已被调度，正准备投入运行，
 - TASKLET_STATE_RUN表示小任务正在运行，它只有在多处理器系统上才使用，任何时候单处理器系统都清楚，一个小任务是不是正在运行。
- count域是小任务的引用计数器。
 - 如果它不为0，则小任务被禁止，不允许执行；
 - 只有当它为零，小任务才被激活，并且在被设置为挂起时，小任务才能够执行。
- 结构中的func域就是下半部中要推迟执行的函数，data是它唯一的参数。

编写自己的小任务并调度

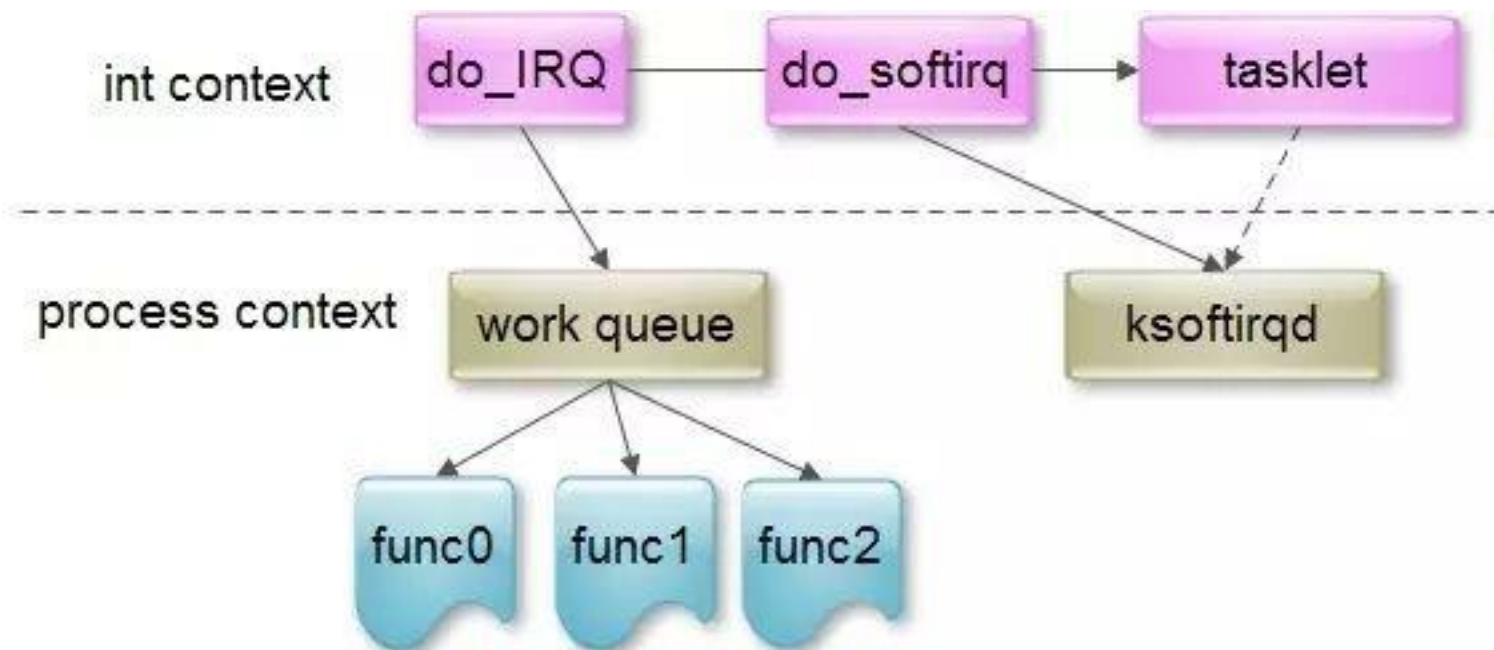
- 声明和使用小任务
 - DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
- 编写自己的小任务处理程序
 - void tasklet_handler(unsigned long data)
- 小任务不能睡眠，不能在小任务中使用信号量或者其它产生阻塞的函数。但它运行时可以响应中断
- 调度或杀死小任务
 - tasklet_schedule(&my_tasklet); /*调度*/
 - tasklet_kill() /*杀死*/

工作队列(workqueue)机制

- 前面的机制都是在中断上下文中。
- 因为它们不可挂起，而且由于是串行执行，只要有一个处理时间较长，则会导致其他中断响应的延迟。
- 为了完成这些不可能完成的任务，于是出现了工作队列
- 工作队列就是一组内核线程，作为中断守护线程来使用。

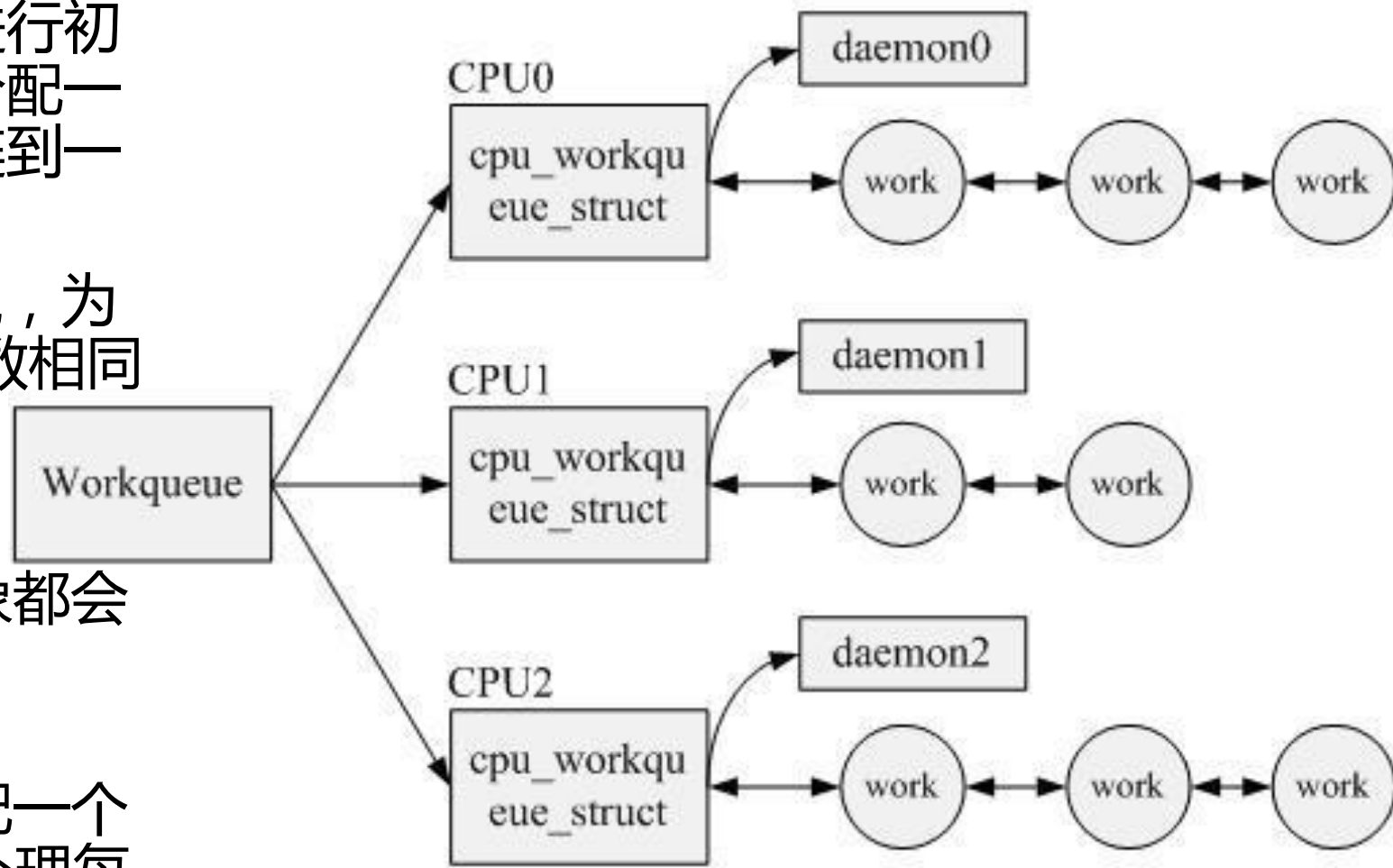
工作队列(workqueue)机制

- 多个中断可以放在一个线程中，也可以每个中断分配一个线程。
- 工作队列对线程作了封装，使用起来更方便。如图所示。
- 因为工作队列是线程，故可使用所有能够在线程中使用的方法。



工作队列运行机制

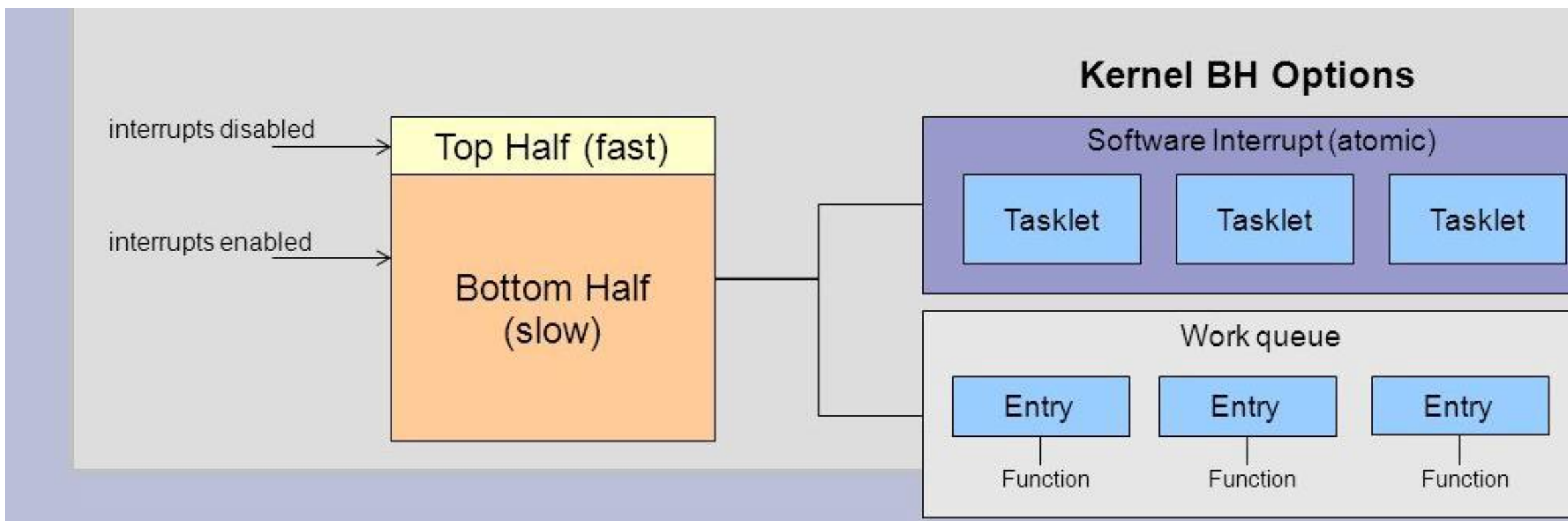
- 当用户调用工作队列(workqueue)初始化接口函数，对工作队列进行初始化时，内核就开始为用户分配一个工作队列对象，并且将其链到一个全局的workqueue队列中。
- 然后内核根据当前CPU的情况，为workqueue对象分配与CPU个数相同的cpu_workqueue_struct对象，
- 每个cpu_workqueue_struct对象都会有一条任务队列。
- 紧接着，内核为每个cpu_workqueue_struct对象分配一个内核线程，即内核daemon去处理每个队列中的任务。



何时使用哪种中断处理机制

- Request_irq挂的中断函数要尽量简单，只做必须在屏蔽中断情况下要做的事情。
- 中断的其他部分都在下半部中完成。
- 软中断的使用原则很简单，最好不用。它甚至都不算是一种真正的中断处理机制，只是tasklet的实现基础。
- 工作队列也要少用，如果不是必须要用到线程才能用的某些机制，就不要使用工作队列，
- 为什么对工作队列机制详细介绍，是希望大家从这种设计机制中得到启发。

- 其实对于中断来说，只是对中断进行简单的处理，大部分工作是在驱动程序中完成的。除了上述情况，一般都使用小任务tasklet。
- 即使是下半部，也只是作必须在中断中要做的事情，如保存数据等，其他都交给驱动程序去做。





内容导航：

4 Linux时间系统

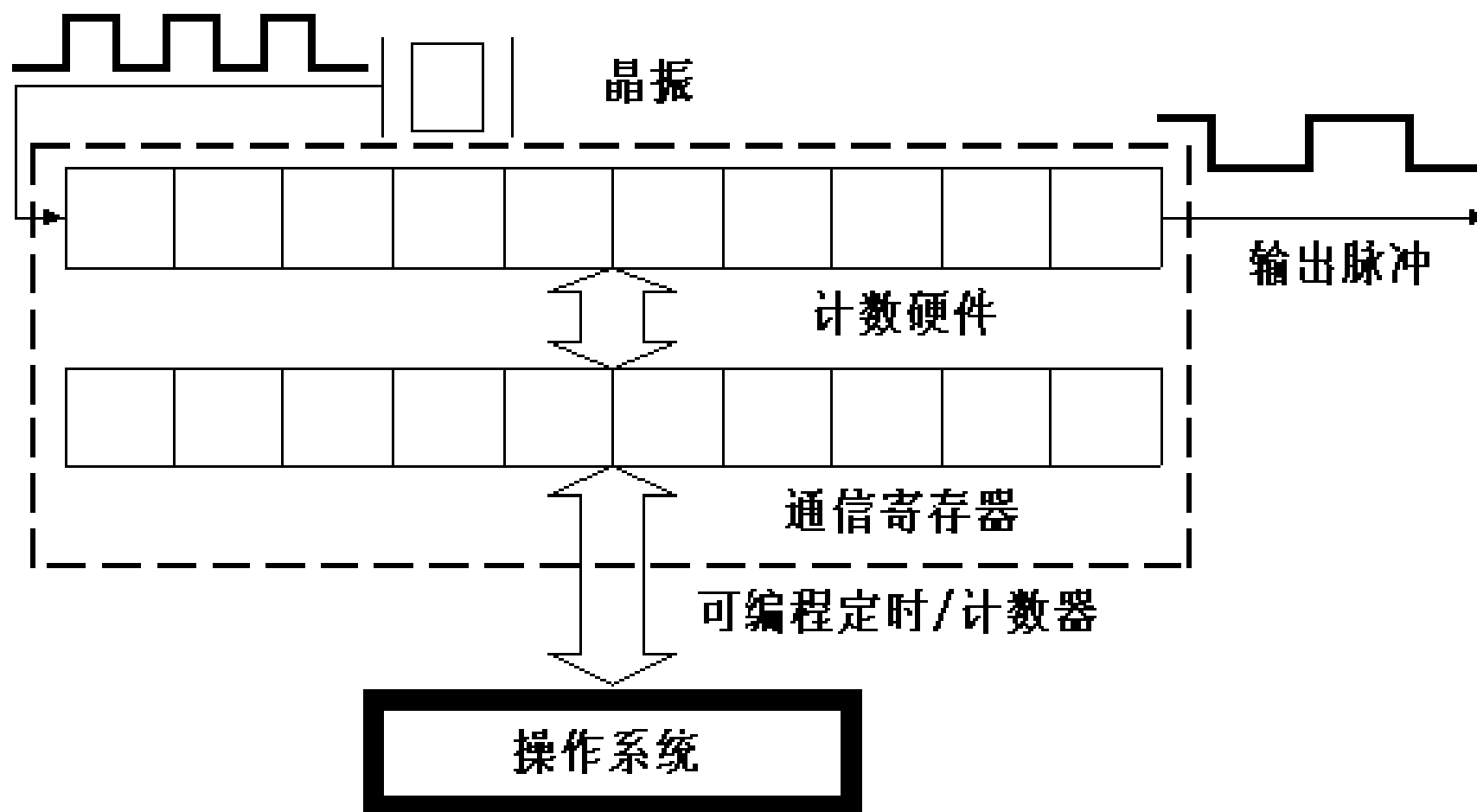
时钟中断

- “时钟中断”是特别重要的一个中断，因为整个操作系统的活动都受到它的激励。
- 系统利用时钟中断维持系统时间、促使进程的切换，以保证所有进程共享CPU；利用时钟中断进行记帐、监督系统工作以及确定未来的调度优先级等工作。
- 可以说，“时钟中断”是整个操作系统的脉搏。

基本时钟硬件

- Linux的OS时钟的物理产生原因，是可编程定时/计数器产生的输出脉冲，这个脉冲送入CPU，就可以引发一个中断请求信号，我们就把它叫做时钟中断。
- 时钟中断的周期，也就是脉冲信号的周期，我们叫做“滴答”或“节拍”(tick)。
- 从本质上说，时钟中断只是一个周期性的信号，完全是硬件行为，该信号触发CPU去执行一个中断服务程序。
- 这是最简单的时钟硬件，在目前的系统中，还有更多相关的时钟硬件。

基本时钟硬件



与时钟有关的硬件

- 实时时钟RTC (Real Time Clock)
 - 用于长时间存放系统时间的设备，即使关机后也可依靠主板CMOS电池继续保持系统的计时.
- 可编程间隔器PIT(Programmable Interval Timer)
 - 该设备可以周期性的发送一个时间中断信号。在Linux系统中,该中断时间间隔由HZ表示，这个时间间隔也被称为一个节拍(tick).
- 时间戳计数器TSC(Time Stamp Clock)
 - CPU附带了一个64位的时间戳寄存器，当时钟信号到来的时候，该寄存器内容自动加1

与时钟有关的硬件

- 高精度计时器(HPET)
 - 这是一种由intel开发的新型定时芯片。该设备有一组寄时器,每个寄时器对应有自己的时钟信号,时钟信号到来的时候就会自动加1。
- CPU本地定时器
 - 在处理器的本地APIC提供的一个定时设备。•可以单次或者周期性的产生中断信号。
- 高精度定时器(hrtimer)
 - 提供纳秒级的定时精度，以满足对精确时间有迫切需求的应用程序或内核驱动，例如多媒体应用，音频设备的驱动程序等等

- 在这6个相关硬件中，其中有2个计时器(counter)，有2个定时器(timer)，每个都有其特定的功能。
- 时钟中断其他相关内容，请查阅附录的材料。



内容导航：

课程思政

课程思政

增强对中华民族的认同感，坚定科技报国的信心和决心

自2019年谷歌公司停止与华为的部分合作以来，华为鸿蒙系统便走上了历史发展的车道，并一直备受操作系统领域相关人士的高度关注。那么，华为鸿蒙系统究竟是怎样一个系统呢？它和你心目中的国产操作系统究竟相差几何？

鸿蒙系统OS 1.0：2019年8月9日，华为在广东东莞举行了华为开发者大会，正式发布了鸿蒙系统OS 1.0，并宣布实行开源。鸿蒙系统OS 1.0是一款“面向未来”的全场景分布式国产操作系统，可以按需扩展，实现更广泛的系统安全，主要用于智能手机、智慧屏、智能手表等物联网设备，特点是低时延。鸿蒙系统OS 1.0实现了模块化耦合，对于不同设备可进行弹性部署，设有三层架构：第一层是内核，第二层是基础服务，第三层是程序框架。2019年8月10日发布的荣耀智慧屏等设备，均搭载有鸿蒙系统OS 1.0。

增强对中华民族的认同感，坚定科技报国的信心和决心

鸿蒙系统OS 2.0：2020年9月10日，鸿蒙系统OS 1.0升级至鸿蒙系统OS 2.0，在关键的分布式软总线、分布式数据管理、分布式安全等分布式能力上进行了全面升级，为开发者提供了完整的分布式设备与应用开发生态。目前，华为已与美的集团、九阳股份有限公司等家电厂商建立了合作关系，这些厂商将发布搭载鸿蒙系统OS 2.0的全新家电产品。此外，在2021年初，华为面向旗下部分型号的手机提供了升级渠道；目前，90%以上的华为手机均可升级至鸿蒙系统OS 2.0。这些都会极大程度地促进华为快速构建鸿蒙系统的应用生态。

综上所述可知，华为鸿蒙系统的诞生，已然拉开了永久性改变全球操作系统格局的序幕，我们应该增强对中华民族的认同感，坚定科技报国的信心和决心，坚信在此背景下，自主可控的国产操作系统一定可以早日在世界操作系统领域发挥核心引领作用。



- 1、中断是CPU对系统发生的某个事件作出的一种反应，请简述为什么引入中断。
- 2、请对机制与策略分离的中断机制做一个简述。
- 3、中断子系统分成哪4个部分？
- 4、请简述中断处理程序的执行流程。
- 5、请简述中断下半部处理机制是什么？与上半部处理机制有何区别？
- 6、下半部的处理方式主要有哪三种？其中哪个是在进程上下文执行的？
- 7、请简述Linux系统的时钟中断。

谢谢！

THANKS