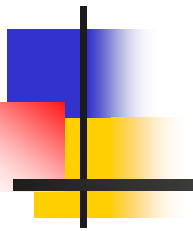


中断、异常和系统调用

王金伟

1、中断的种类

- 由CPU外部产生的—“中断”（interrupt）
- 由专设指令（如INT）产生的—“陷阱”（trap）
- 由CPU本身在执行指令时产生的—“异常”（exception）

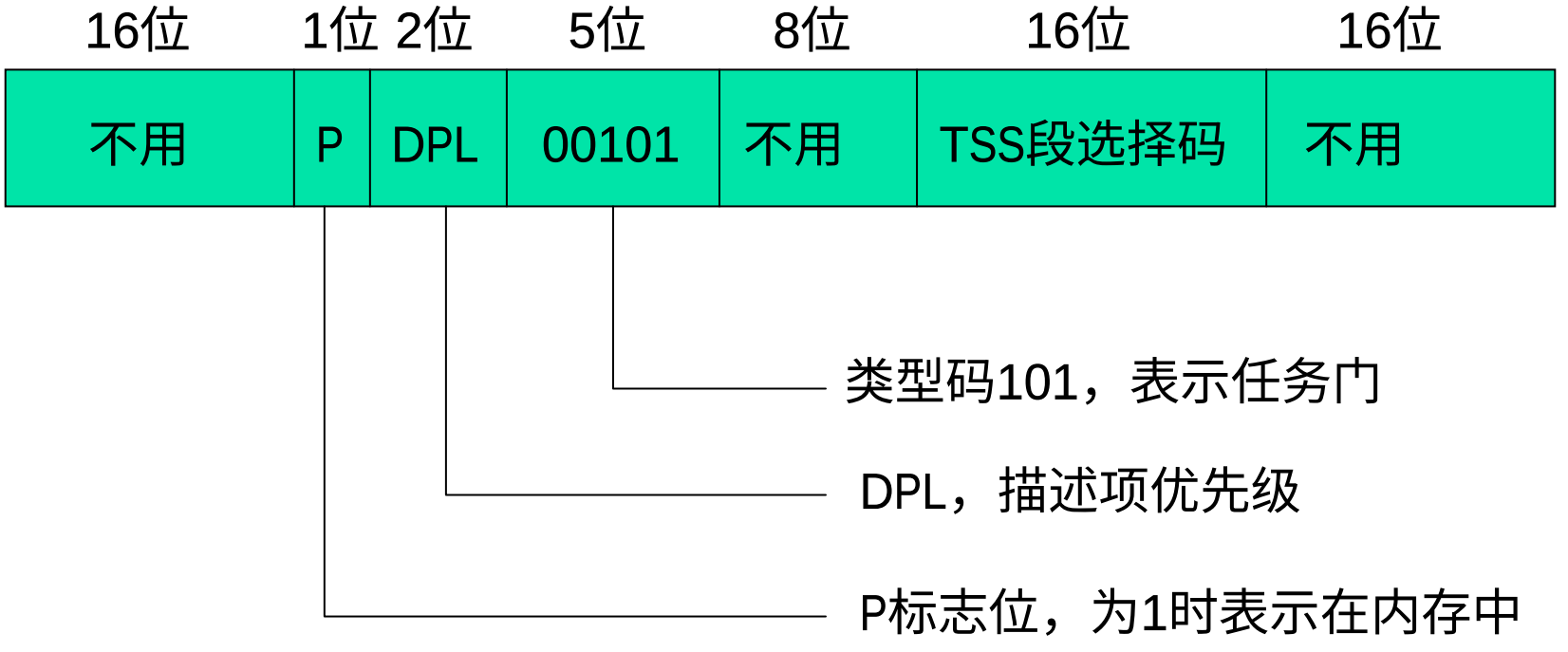
2、X86 CPU对中断的硬件支持

- 引入了“门”（**gate**）

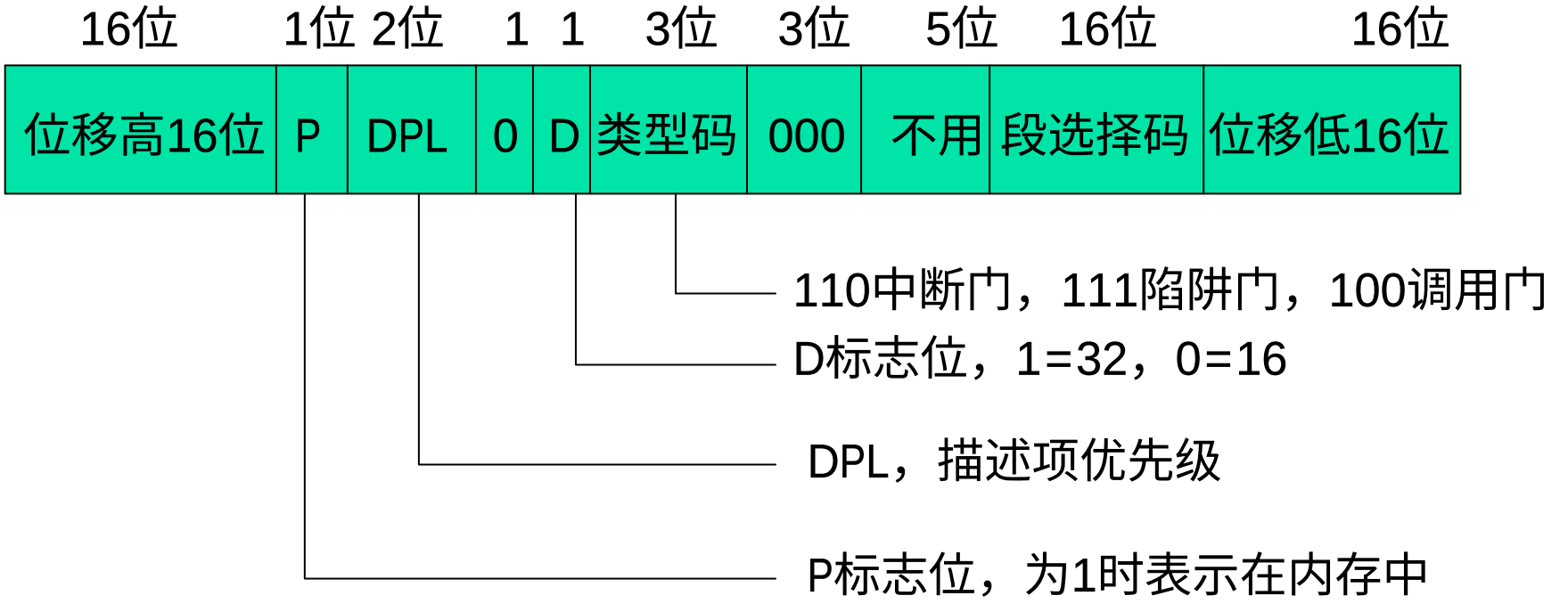
- “门”的类型和结构：

  任务门、中断门、陷阱门、调用门

任务门的结构（64位）


16位
1位
2位
5位
8位
16位
16位
不用
P
DPL
00101
不用
TSS段选择码
不用
类型码101，表示任务门
DPL，描述项优先级
P标志位，为1时表示在内存中

中断门、陷阱门和调用门结构（64位）



中断优先级别检测

- 如果中断源是INT指令：

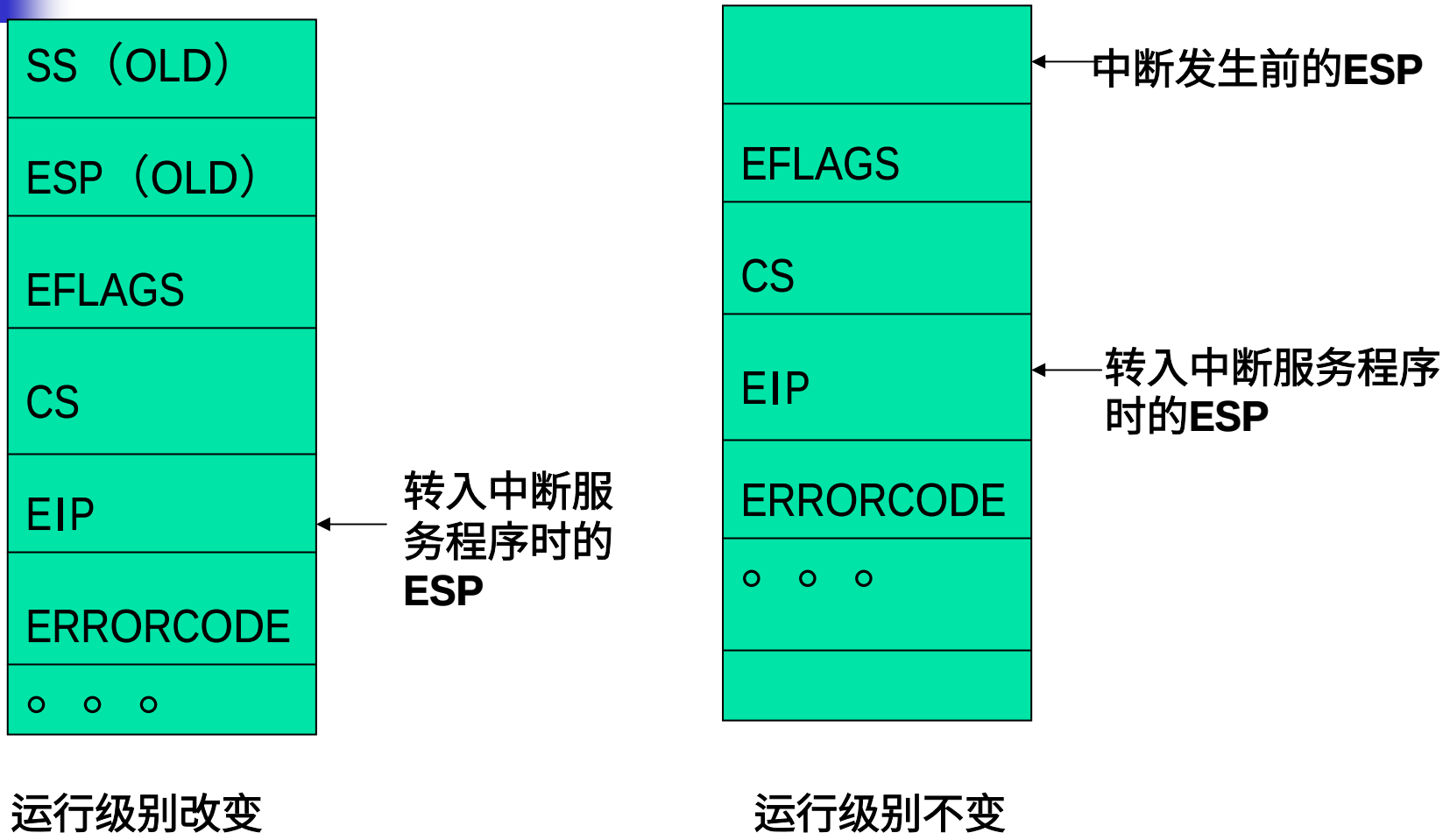
  DPL1>=CPL>=DPL

- 如果中断塬是由CPU外部或异常产生的 ：

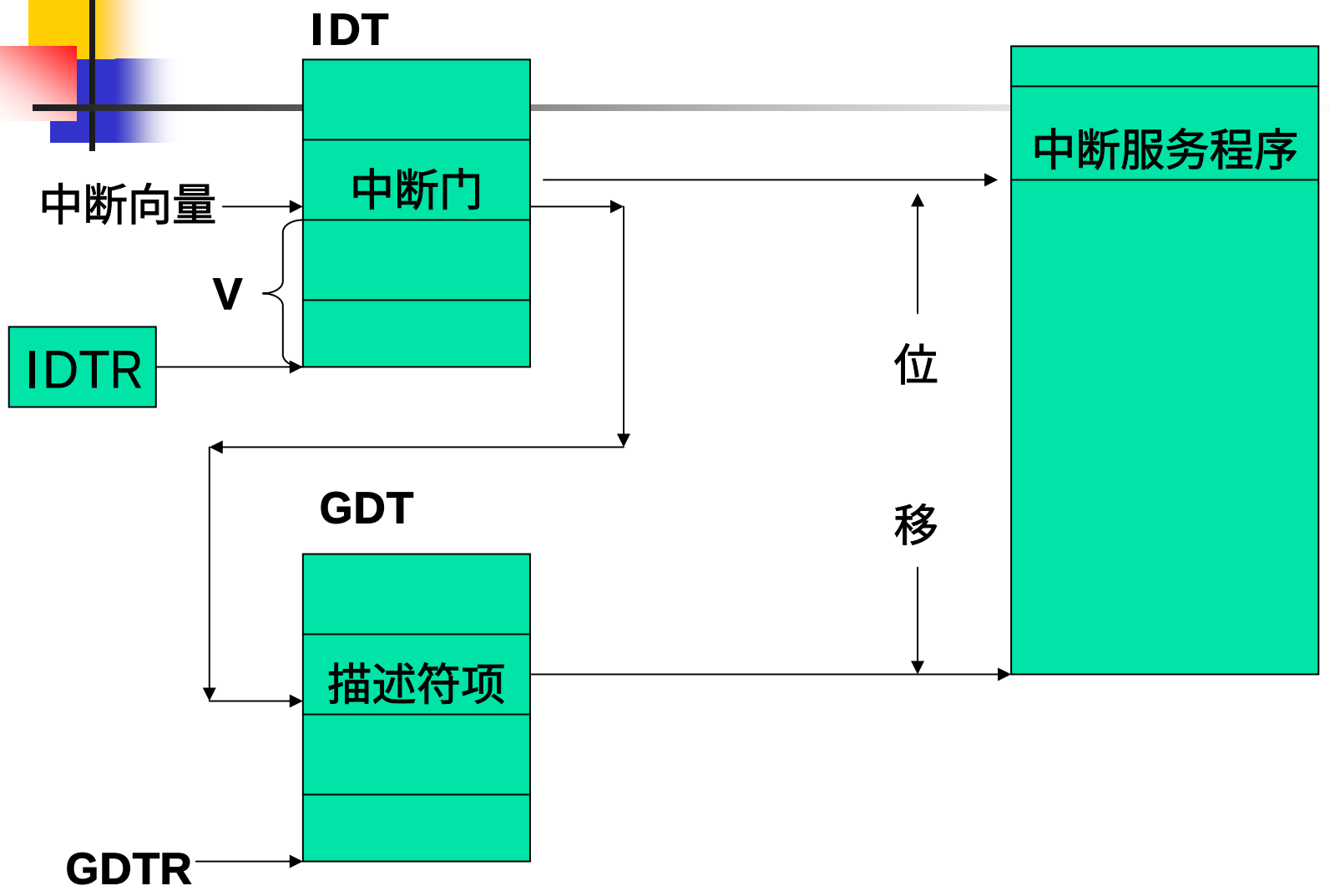
  CPL>=DPL1

其中DPL1是目标代码段的优先级别，CPL是CPU当前的运行级别，DPL是中断门的优先级别。

中断的堆栈操作


SS（OLD）
ESP（OLD）
EFLAGS
CS
EIP
ERRORCODE
o o o
转入中断服务程序时的ESP
运行级别改变
EFLAGS
CS
EIP
ERRORCODE
o o o
中断发生前的ESP
转入中断服务程序时的ESP
运行级别不变

中断机制示意图



3、中断向量表IDT的初始化

- Linux内核在初始化阶段完成对页式虚存管理的初始化之后，调用trap_init()和init_IRQ()两个函数进行中断机制的初始化

- trap_init()是对一些系统保留的中断向量的初始化。

- Init_IRQ()主要是用于对外设中断的初始化。

3.1 trap_init()

- 设置了中断向量表开头的**20**个陷阱门（**0-19**）

  set_trap_gate(向量号，服务程序入口地址）

  包括：被**0**除、调试异常、缺页异常等等。

- 然后是对系统调用向量的初始化

set_system_gate(SYSCALL_VECTOR,&syste
m_call) **0x80**

宏：_set_gate (gate-addr,type,dpl,addr)

- 用来设置中断描述符表中第n项中断描述符。
- **gate-addr** ：idt表项地址
- **Type**： 门的类型
- **Dpl**： 门的优先级别
- **Addr**：服务程序的地址

Set-intr-gate（）、set_trap_gate（）、set_system_gate（）就是通过 _set_gate 宏设置相应的“门”的。

例如：**set_set_trap_gate(unsigned int n, void *addr)**

```
{
  _set_gate(idt_table+n,15,0,addr);
}
```

3.2 Init_IRQ()

(1)调用init_ISA_irqs（）对PC的中断控制器8259A进行初始化，以及其他工作。

(2)通过一个循环调用set_intr_gate(vector,interrupt[i])

设置中断向量表的从0x20项以后的224项，但是跳过0x80项（系统调用项）；

Interrput[I]是函数指针数组中的第I个函数指针。

Interrput[]函数指针数组

- 经过一系列的宏定义，最终就是这样一组函数指针：

IRQ0x01_interrupt()一直到

IRQ0x0f_interrupt()

中断门（描述符）示意图

其他	**IRQ0x01_interrupt()**

IRQ0xXX_interrupt()

- 每个**IRQ0xXX_interrupt()**并不是最终的中断服务程序，他们具有相似的功能和代码：

（1）先把中断向量的相关数值压入堆栈

（2）再跳转到 **common_interrupt**这个公共的中断服务程序中进行具体细节的处理。

4、中断请求队列的初始化

- 外设的中断门是可以让多个中断源所共用（中断向量0x20以上的，除了0x80系统调用）。
- 所以为每个通用的中断门都设置了一个中断请求队列，每个队列有一个队头。
- 队列中就是共用一个中断门的服务程序的相关描述结构。

几个相关的数据结构 1

数组Irq_desc[]

请求队列的队头数组。

每个元素是一个请求队列的队头。

共有224项，对应着224个中断门。

handler项指向hw_interrupt_type数据结构。

其中主要是对公用中断通道进行控制的函数指针。

Action项 用来维护中断服务程序描述项构成的单链队列。

几个相关的数据结构 2

Hw_interrupt_type 数据结构

有一些函数指针

用来控制“中断通道”，具体的函数取决于所用的中断控制器。

在系统初始化时通过**init_IRQ**中调用**ini_ISA_irqs()**设置好了。

几个相关的数据结构 3

Irqaction数据结构

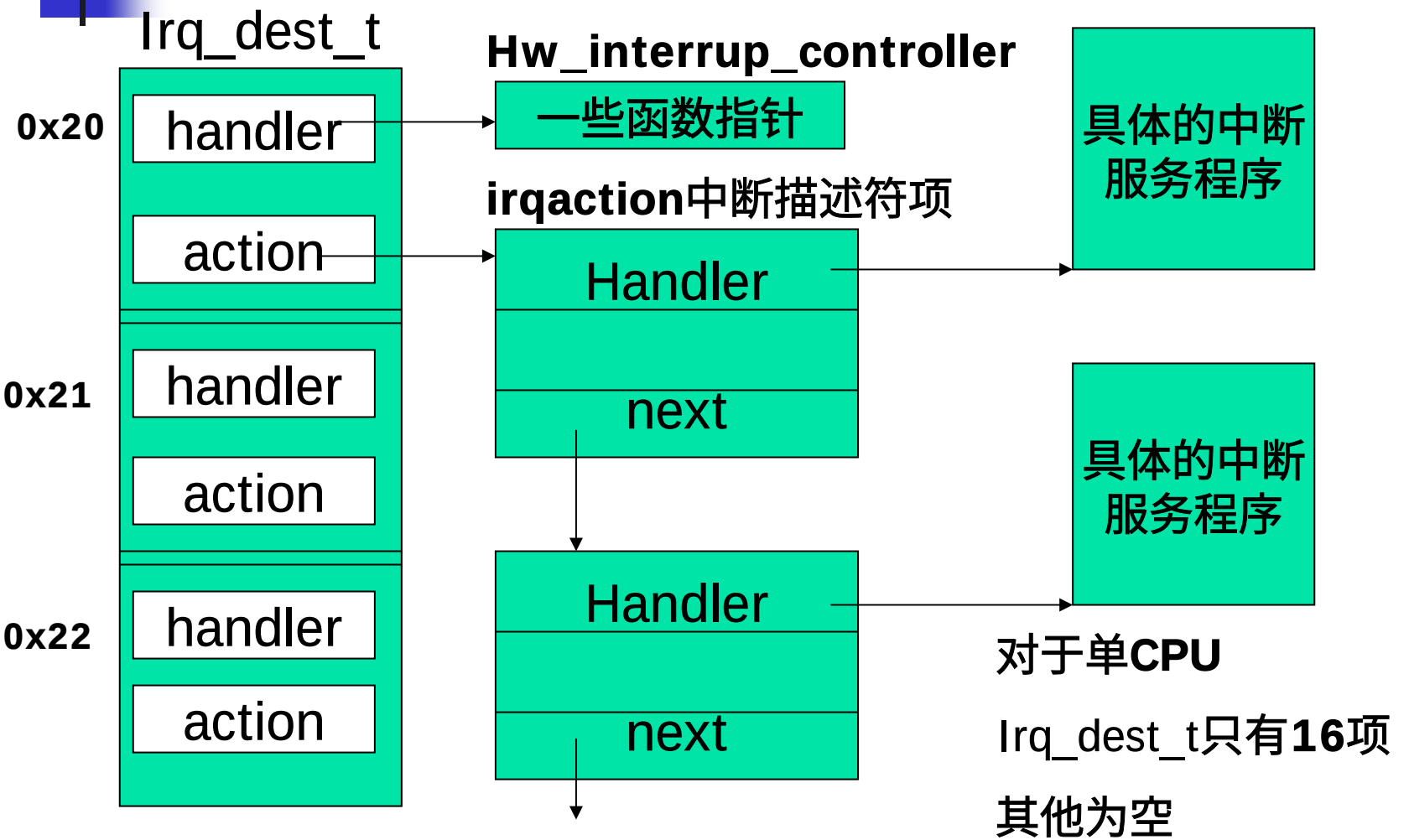
是中断服务程序描述符项

Handler指向具体的中断服务程序的入口

Dev_id 是设备号，由设备自己辨认

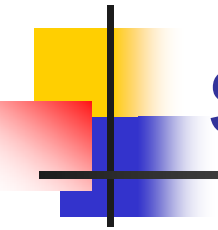
Next 指向下一个中断描述符项

三个结构之间的关系图


Irq_dest_t
0x20
handler
action
0x21
handler
action
0x22
handler
action
Hw_interrup_controller
一些函数指针
irqaction中断描述符项
Handler
next
Handler
next
具体的中断服务程序
具体的中断服务程序
对于单CPU
Irq_dest_t只有16项
其他为空

Request_irq()

- 设备初始化程序通过他将中断服务程序向系统登记。（指明中断请求号**IRQ**）
- 就是分配了一个中断服务程序描述符项
- 也就是分配了一个**irqaction**数据结构并进行相应的设置
- 调用**setup_irq()**



setup_irq()

- 把分配并设置好的**iqraction**链入到相应中断请求号所对应的中断队列中

5、中断响应和服务的过程

- 假设外设的驱动程序都已经初始化并已经将自己的中断服务程序挂入到相应的中断请求队列中
- 系统正在用户空间运行，中断是开着的。
- 某个外设已经产生了一次中断请求，通过**8259A**到达**CPU**的“中断请求”引线**INTR**
- **CPU**在执行完当前指令后就来响应中断。

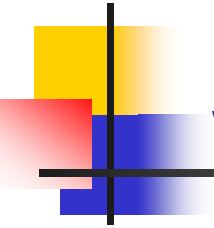
Step 1

CPU从中断控制器（8259A）中取得中断向量，然后根据这个中断向量和IDTR从中断向量表IDT中找到对应的中断门(因为是外部中断，故不进行中断门优先级别的检查)；

所以CPU根据中断门的设置找到中断服务程序段的段描述符将CPL与其DPL相比较为3＞0，满足条件；

因为CPL与DPL不同所以要进行堆栈切换，CPU从当前的TSS中用于内核0级的堆栈指针（ESP和SS）装入ESP和SS，切换到内核堆栈；

然后CPU将原来用户堆栈的指针、EFLAGS、返回地址（CS和EIP）压入到新的堆栈中，即内核堆栈中。



Step 2

CPU根据中断门以及目标代码段的设置到达中断总服务程序的入口：IRQ0xXX_interrupt，假设IRQ0x03_interrupt；

IRQ0x03_interrupt先将中断请求号-256（为了与系统调用号区别开）得到的数值压入堆栈；

然后跳转到common_interrupt执行

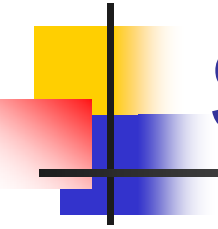
Step 3

在common_interrupt中，
先通过宏操作
SAVE_ALL，保存现场，
就是将一系列寄存器的植
压入堆栈。

再将ret_from_intr地址压入
堆栈。（后面会用到）

然后调用do_IRQ()

堆栈的示意图：



Step 4

进入**do_IRQ()**

1、从堆栈中得到中断调用号也就是**0x03**

2、根据中断调用号从**irq_desc[]**数组中找到响应的中断请求队列头。

3、一套通过循环防止中断铅套的机制（可以具体看源码，这里略）

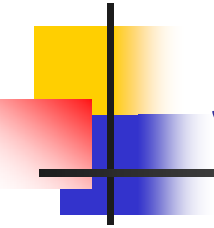
4、最终要调用**handle_IRQ_event()**。

Step 5

进入**handle_IRQ_event()**。

它通过一个**do-while**循环，依次执行在第**4**步中找到的中断请求队列中的各个服务程序，让每个服务程序去辨认本次中断请求是否来自各个服务对象。

队列中的每个中断服务程序执行时，先检查各自的中断源（一般是读相应设备上的中断状态寄存器）看是否有来自该设备的中断请求，如果没有，就马上返回。有就继续执行。



Step 6

从handle_IRQ_event()返回到do_IRQ()
在do_IRQ()最后还要
处理软中断请求，（以前称为bottom half)

然后从do_IRQ()返回到
entry.s标号为ret_from_intr处继续执行。
（在前边已经把ret_from_intr的地址压入堆栈中，因此do_IRQ()返回时CPU将从那里开始执行）

Step 7

进入**ret_from_intr**开始执行：

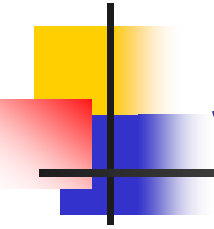
检验中断前**CPU**是否运行在**VM86**模式。

检验中断前**CPU**运行于用户空间还是系统空间。

如果中断发生在系统空间就 转移到**restore_all**

否则转移到**ret_with_reschedule**

假定是发生在用户空间，转移到**ret_with_reschedule**



Step 8

ret_with_reschedule

先检查一下是否要进行进程调度，如果需要调用**schedule()**调度进程（进程一章讨论）。

执行宏**RESTORE_ALL**：

与开始的**SAVE_ALL**相对应，就是“恢复现场”把调用前寄存器的值复原

执行**ret**从中断返回，**CPU**会切换堆栈（从系统到用户）

就此完成一次中断过程。

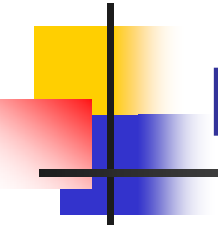
6、软中断与Bottom Half(BH)

问题：

* 关中断的时间太长可能使**CPU**不能及时响应其他的中断请求，会使中断丢失。

* 如果开中断的时间太长会造成中断不断进行这样一种“不安定因素”

所以采用一种折中的办法**Bottom Half**



Bh 机制

将一次中断分成两个部分：

一部分要求在中断请求发生后“立即”执行，必须在关中的条件下执行，不允许有干扰，可以叫做**top half**。

另一部分可以稍微延迟一些执行，可以在开中的条件下执行，称为**bottom half（bh）**。

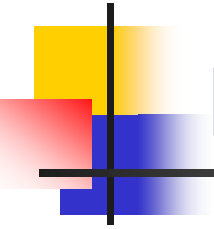
Bh 机制（续1）

- 设置了一个bh_base[]函数指针数组，大小32，其中每个元素代表一个bh服务函数。
- 两个相应的32位无符号整数相当于中断请求寄存器和中断屏蔽寄存器。
- 通过mark_bh(),将“中断请求寄存器”的相应位设成1，进行中断申请。
- 在每次执行完do_IRQ()最后,检测“中断请求寄存器”中相应的位。

Bh 机制（续2）

- 如果为“1”，则说明有中断请求，就转入执行do_softirq()。
- 在do_softirq()中执行bh_action()

- Bh_action()根据中断号，从bh_base[]中找出相应的服务程序函数进行执行。
- 例如执行TIMER_BH



Bh机制是在全局串行化了。

- 即只每个时刻允许其中一个**cpu**执行一个软中断服务程序。（过于严格了）。
- 于是，引入了其他的机制：如**tasklet**机制，等。
- **Tasklet**机制允许几个**cpu**同时执行不同的软中断服务程序。

7、页面异常的进入和返回

*为页面异常设置的中断门指向程序入口**page_fault**

*当发生页面异常时，**CPU**穿过中断门直接到达**page_fault()**，其他的与外设发生中断的过程基本一致（这里略）

*有一点不同：要把异常的出错代码也压入堆栈

页面异常的进入过程 1

到达页面异常处理入口**page_fault**：

*先将**do_page_fault**（）的地址压入堆栈，为进入具体的服务程序做好了准备。

*跳转到**error_code**处执行：

（同中断类似，**error_code**是各种异常处理共用的程序入口，好象是**common_interrupt)**

页面异常的进入过程 2

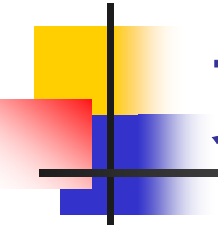
到达error_code ：

*进行“保护现场”的工作，将寄存器内容压入堆栈

*然后将do_page_fault函数指针转到EDI中，将出错代码转到ESI中，将堆栈指针转到EDX中（为调用do_page_fault做准备）

*通过call * %edi 调用do_page_fault

堆栈示意图：



页面异常的返回过程

从do_page_fault（内存已讲过）返回后：

*检查是否有软中断

*跳到ret_from_intr执行（与中断相同，就是恢复现场）

*从异常返回（用户空间）

8、时钟中断

*主要用于定时的对进程进行调度，防止一个进程陷入死循环而抓住CPU不放。

（与进程调度联系紧密，这里只做简单介绍）

*在初始化时调用time_init()进行时钟中断初始化。

在此之前已经对进程调度机制进行了初始化。

设置时钟中断的服务程序为time_interrupt（）

time_interrupt（）

*进行时间精度的一些计算

*调用do_timer_interrupt()

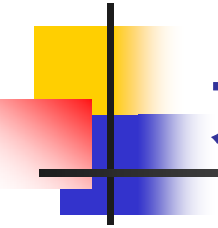
*在do_timer_interrupt()中调用do_timer

*在do_timer()中做一些与进程调度有关的操作，再通过mk_bh()将bh_task_vec[TIMEER_BH]挂到tasklet_hi_vec队列中，使CPU在中断返回之前执行与TIMER_BH对应的函数timer_bh()

timer_bh()

*先调用**update_times()**处理实时时钟的进位记数、精度的矫正以及计算和积累**CPU**负荷的统计信息等工作。

*然后调用**run_timer_list()**检查系统中已经设置的各个定时器（**timer**），如果某个定时器已经“到点”就执行该定时器预定的函数。（与进程调度联系紧密）



大部分在BH中完成

时钟中断服务大部分操作在“后半”，也就是在**bh**函数中完成；

在关中状态下完成的只是关键的少量的工作，大部分还是在开中状态下完成的。

9、系统调用

- 通过中断指令**INT 0x80** 实现。
- 通过一个系统调用**sethostname**分析系统调用的过程。
- **Sethostname（name，len）**：设置计算机的“主机名”，**name**为要设置的主机名，**len**为**name**字符串的长度。

系统调用过程 1

假设进入到系统调用库函数**sethostname**：

*****将参数**name**和**len**分别存入相应的寄存器（用于传递参数）

*****将代表**sethostname**（）的系统调用号**0x4a**存入寄存器**eax**

*****调用中断指令 **int $0x80**，**CPU**穿过陷阱门**0x80**陷阱门的中断服务程序指向**system_call**

系统调用过程 2

进入system_call不关中，系统调用可以被中断

*先保存现场，寄存器压栈（用SAVE_ALL）

*检查系统调用号（eax中的内容）是否超出范围。

*调用sys_call_table数组中的第eax项，系统调用号对应的那一项（这里是第0x4a项，74项） sys_sethostname

sys_call_table数组是一个函数指针数组（大小为256，linux中定义了221个系统调用），系统调用时以系统调用号为下标可以找到相应的系统调用的函数指针，

系统调用过程 3

进入sys_sethostname:

*进行设置主机名的相关操作（具体过程略）

*修复地址：

系统调用过程 4

从具体的系统调用函数(sys_sethostname

返回到system_call():

*具体的系统调用函数将返回值写入eax中，进入ret_from_sys_call

*与中断相似，先检查有无软中断需要执行，再进行现场恢复，寄存器内容出栈。

*返回sethostname

系统调用过程 5

返回**sethostname**：

*检查**eax**中的返回值如果是负值，说明出错，跳转到**__syscall_error()**：

* **__syscall_error()**中先将**eax**中的值还原成出错代码，再调用**__error_location**将出错代码写入全局量**errono**。返回

*如果**eax**中的返回值为**0**，表示成功，正常返回。