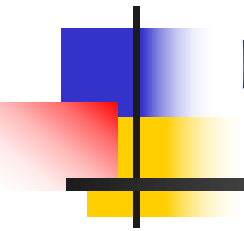
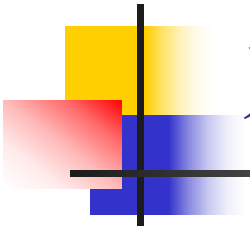


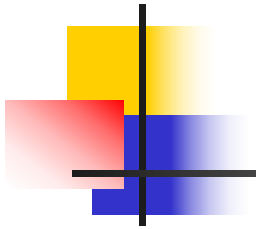
中断和异常

Linux操作系统分析



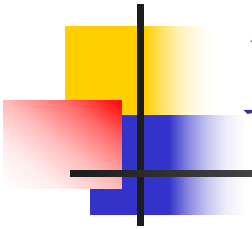
为什么会有中断

- 内核的一个主要功能就是处理硬件外设I/O
  - 处理器速度一般比外设快很多
  - 内核必须处理其他任务，只有当外设真正完成了准备好了时CPU才转过来处理外设IO
  - IO方式：
    - 轮询、中断、DMA等
  - 轮询方式效率不高
  - 中断机制就是满足上述条件的一种解决办法



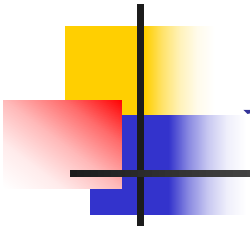
- 查看系统中断信息
  - cat /proc/interrupts

/proc/interrupts: to display every IRQ vector in use by the system



主要内容

- 中断信号的作用和中断信号处理的一般原则
- I/O设备如何引起CPU中断
- x86 CPU如何在硬件级处理中断信号
- Linux内核中软件级中断处理及其数据结构
- Linux的软中断、tasklet以及下半部分



主要内容

- 中断信号的作用和中断信号处理的一般原则
- I/O设备如何引起CPU中断
- x86 CPU如何在硬件级处理中断信号
- Linux内核中软件级中断处理及其数据结构
- Linux的软中断、tasklet以及下半部分

中断和异常

- **中断**（广义）会改变处理器执行指令的顺序，通常与CPU芯片内部或外部硬件电路产生的电信号相对应
  - 中断——异步的：
    由硬件随机产生，在程序执行的任何时候可能出现
  - 异常——同步的：
    在（特殊的或出错的）指令执行时由CPU控制单元产生

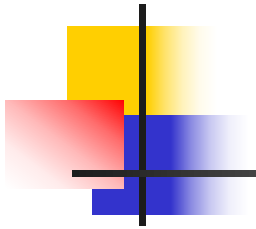
  我们用“中断信号”来通称这两种类型的中断

中断信号的作用

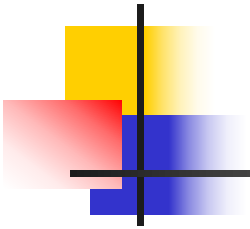
- 中断信号提供了一种特殊的方式，使得CPU转去运行正常程序之外的代码
  - 比如一个外设采集到一些数据，发出一个中断信号，CPU必须立刻响应这个信号，否则数据可能丢失
- 当一个中断信号到达时，CPU必须停止它当前正在做的事，并且切换到一个新的活动
- 为了做到这这一点，
  - 在进程的内核态堆栈保存程序计数器的当前值(即eip和cs寄存器)以便处理完中断的时候能正确返回到中断点，
  - 并把与中断信号相关的一个地址放入进程序计数器，从而进入中断的处理

中断信号的处理原则

- 快！
  - 当内核正在做一些别的事情的时候，中断会随时到来。无辜的正在运行的代码被打断
  - 中断处理程序在run的时候可能禁止了同级中断
  - 中断处理程序对硬件操作，一般硬件对时间也是非常敏感的
  - 内核的目标就是让中断尽可能快的处理完，尽其所能把更多的处理向后推迟
  - 上半部分(top half)和下半部分(bottom half)



- 允许不同类型中断的嵌套发生，这样能使更多的I/O设备处于忙状态
- 尽管内核在处理一个中断时可以接受一个新的中断，但在内核代码中还在存在一些临界区，在临界区中，中断必须被禁止



中断上下文

- 中断上下文不同于进程上下文
  - 中断或异常处理程序执行的代码不是一个进程
  - 它是一个**内核控制路径**，代表了中断发生时正在运行的进程执行
  - 作为一个进程的内核控制路径，中断处理程序比一个进程要“轻”(中断上下文只包含了很有限的几个寄存器，建立和终止这个上下文所需要的时间很少)

中断上下文举例

- 分析A,B,C,D在互相抢占上的关系

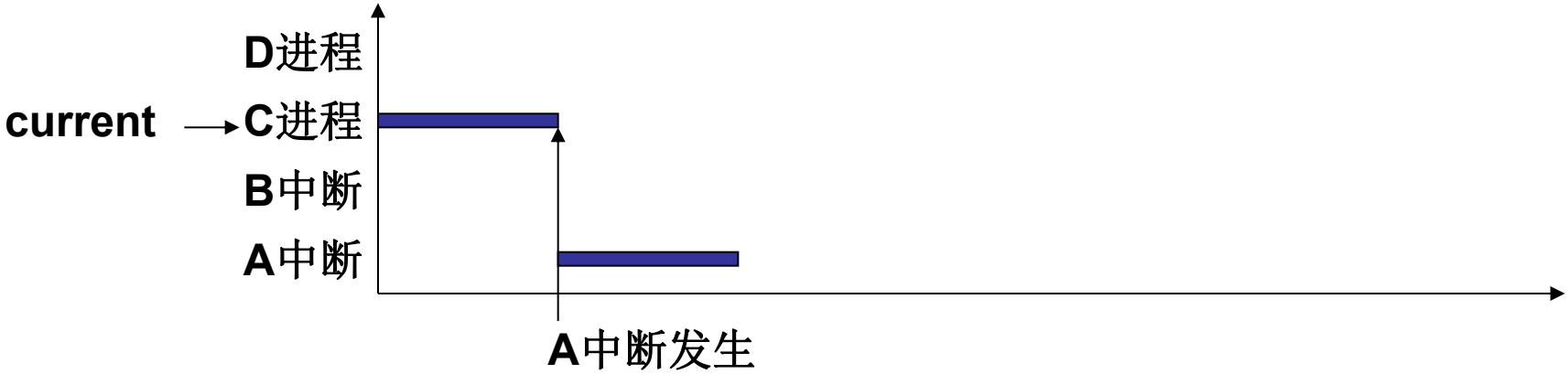
  假设:

  2个interrupt context，记为A和B

  2个process，记为C和D

1， 假设某个时刻C占用CPU运行，此时A中断发生，C被A抢占，A得以在CPU上执行。

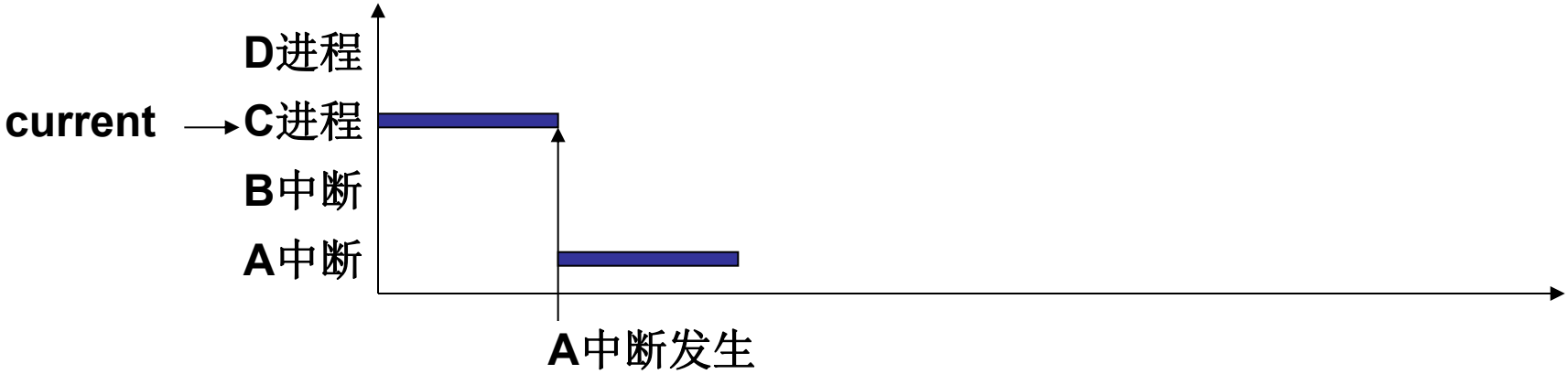
由于Linux不为中断处理程序设置process context，A只能使用 C的kernel stack作为自己的运行栈

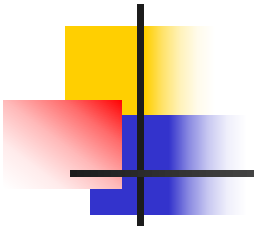


2，无论如何，Linux的interrupt context A绝对不会被某个进程C或者D抢占！！

这是由于所有已经启动的interrupt contexts，不管是interrupt contexts之间切换，还是在某个interrupt context中执行代码的过程，决不可能插入scheduler调度例程的调用。

除非interrupt context主动或者被动阻塞进入睡眠，唤起scheduler，但这是必须避免的，危险性见第3点说明。





3，关于第2点的解释：

首先，interrupt context没有process context，A中断是“借”了C的进程上下文运行的，若允许A“阻塞”或“睡眠”，则C将被迫阻塞或睡眠，仅当A被“唤醒”C才被唤醒；而“唤醒”后，A将按照C在就绪队列中的顺序被调度。这既损害了A的利益也污染了C的kernel stack。

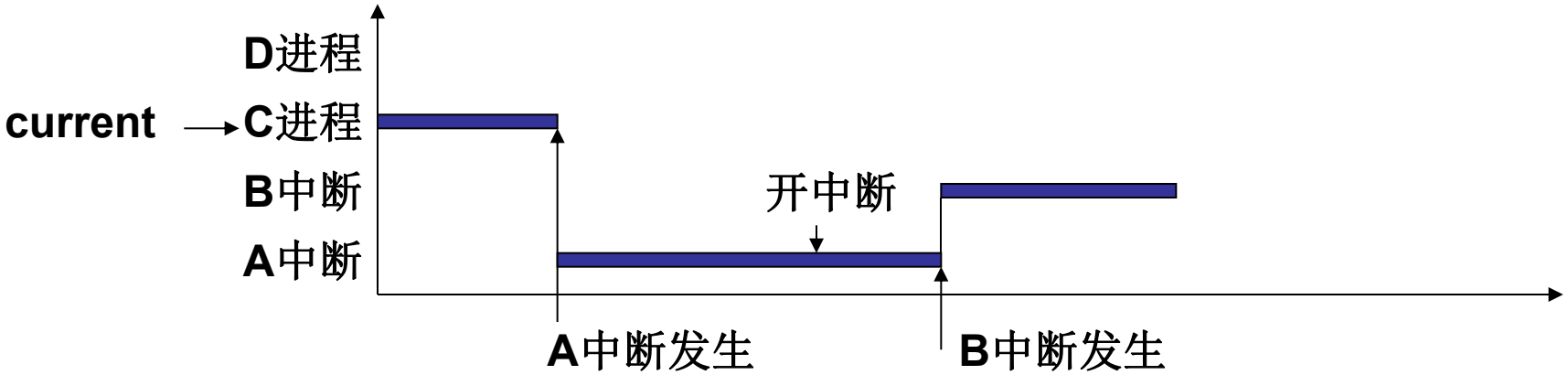
其次，如果interrupt context A由于阻塞或是其他原因睡眠，外界对系统的响应能力将变得不可忍受

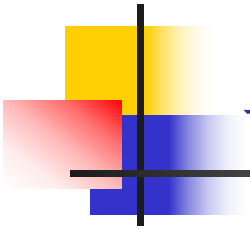
4， 那么interrupt context A和B的关系又如何呢？

由于可能在interrupt context的某个步骤打开了CPU的IF flag标志，这使得在A过程中，B的irq line已经触发了PIC，进而触发了CPU IRQ pin，使得CPU执行中断B的interrupt context，这是中断上下文的嵌套过程。

5，通常Linux不对不同的interrupt contexts设置优先级，这种任意的嵌套是允许的

当然可能某个实时Linux的patch会不允许低优先级的interrupt context抢占高优先级的interrupt context





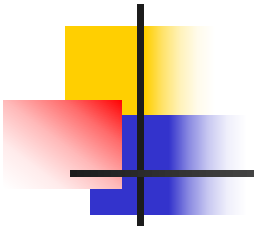
主要内容

- 中断信号的作用和中断信号处理的一般原则
- **I/O设备如何引起CPU中断**
- x86 CPU如何在硬件级处理中断信号
- Linux内核中软件级中断处理及其数据结构
- Linux的软中断、tasklet以及下半部分

中断和异常的分类

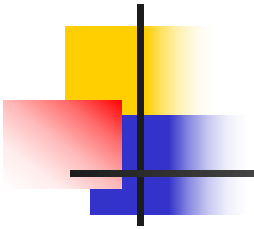
中断分为：

- 可屏蔽中断（Maskable interrupt）
  - I/O设备发出的所有中断请求(IRQ)都可以产生可屏蔽中断。
  - 可屏蔽中断可以处于两种状态:屏蔽的(masked)和非屏蔽的(unmasked)
- 非屏蔽中断（Nonmaskable interrupt）
  - 只有几个特定的危急事件才引起非屏蔽中断。如硬件故障或是掉电

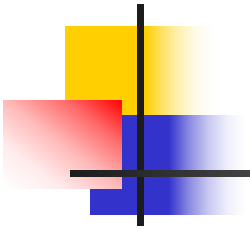


异常分为：

- 处理器探测异常
  - 由CPU执行指令时探测到一个反常条件时产生，如溢出、除0错等
- 编程异常
  - 由编程者发出的特定请求产生，通常由int类指令触发
  - 通常叫做“软中断”
    - 例如系统调用



- 对于处理器探测异常，根据异常时保存在内核堆栈中的eip的值可以进一步分为：
  - 故障(fault)：eip=引起故障的指令的地址
    - 通常可以纠正，处理完异常时，该指令被重新执行
    - 例如缺页异常
  - 陷阱(trap)：eip=随后要执行的指令的地址。
  - 异常中止(abort)：eip=？？？
    - 发生严重的错误。eip值无效，只有强制终止受影响的进程

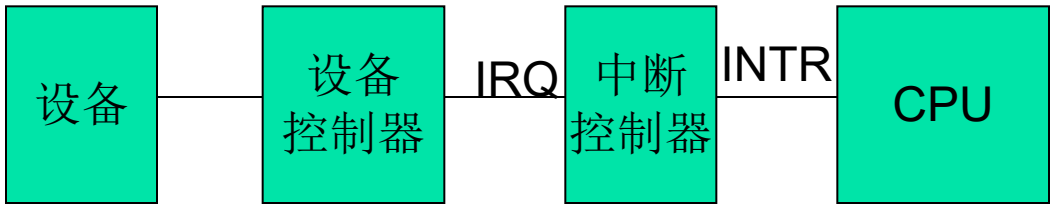


中断向量

- 每个中断和异常由0~255之间的一个数（8位）来标识，Intel称其为中断向量。
  - 非屏蔽中断的向量和异常的向量是固定的
  - 可屏蔽中断的向量可以通过对中断控制器的编程来改变

中断的产生

- 每个能够发出中断请求的硬件设备控制器都有一条称为IRQ(Interrupt ReQuest)的输出线。
- 所有的IRQ线都与一个中断控制器的输入引脚相连
- 中断控制器与CPU的INTR引脚相连



中断控制器执行下列动作：

1，监视IRQ线，对引发信号检查

2，如果一个引发信号出现在IRQ线上

a，把此信号转换成对应的中断向量

b，把这个向量存放在中断控制器的一个I/O端口，从而允许CPU通过数据总线读这个向量

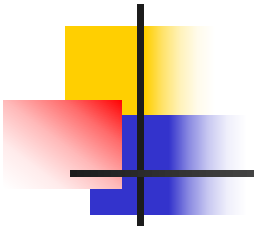
c，把引发信号发送到处理器的INTR引脚，即产生一个中断

d，等待，直到CPU应答这个信号；收到应答后，清INTR引脚

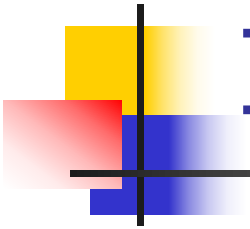
3，返回到第一步

IRQ号和中断向量号

- 中断控制器对输入的IRQ线从0开始顺序编号
  - IRQ0，IRQ1，…
- Intel给中断控制器分配的中断向量号从32开始，上述IRQ线对应的中断向量依次是
  - 32+0、32+1、…
- 可以对中断控制器编程：
  - 修改起始中断向量的值，或
  - 有选择的屏蔽/激活每条IRQ线



- 屏蔽的中断不会丢失
  - 一旦被激活，中断控制器又会将它们发送到CPU
- 有选择的屏蔽/激活IRQ线≠全局屏蔽/激活
  - 前者通过对中断控制器编程实现
  - 后者通过特定的指令操作CPU中的状态字

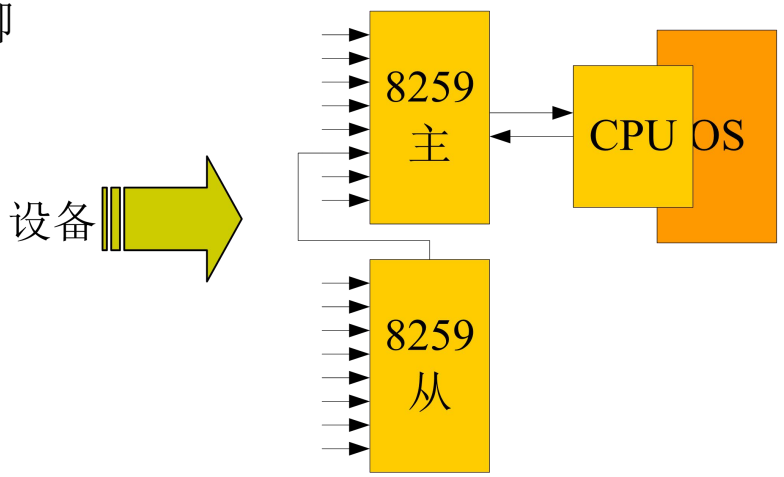


I386：开中断和关中断

- CPU可以将屏蔽所有的可屏蔽终端
  - Eflags中的IF标志：
    0=关中断；
    1=开中断。
  - 关中断时，CPU不响应中断控制器发布的任何中断请求
  - 内核中使用cli和sti指令分别清除和设置该标志

传统的中断控制器：8259A

- 传统的中断控制器使用两片8259A以“级联”的方式连接在一起
- 每个芯片可以处理最多8个不同的IRQ线
- 主从两片8259A的连接：
  - 从→主的IRQ2引脚



- 因此，一共可以处理最多15个不同的IRQ线

8259A：设置起始中断向量号

- 参见init_8259A

arch/x86/kernel/i8259_32.c

```
00289:      /*
00290:       * outb_pic - this has to work on a wide range of PC hardware.
00291:       */
00292:      outb_pic(0x11, PIC_MASTER_CMD);    /* ICW1: select 8259A-1 init */
00293:      outb_pic(0x20 + 0, PIC_MASTER_IMR);      /* ICW2: 8259A-1 IR0-7 mapped to 0x20-0x27 */
00294:      outb_pic(1U << PIC_CASCADE_IR, PIC_MASTER_IMR);    /* 8259A-1 (the master) has a slave
00295:      if (auto_eoi)  /* master does Auto EOI */
00296:          outb_pic(MASTER_ICW4_DEFAULT | PIC_ICW4_AEOI, PIC_MASTER_IMR);
00297:      else       /* master expects normal EOI */
00298:          outb_pic(MASTER_ICW4_DEFAULT, PIC_MASTER_IMR);
00299:
00300:      outb_pic(0x11, PIC_SLAVE_CMD);      /* ICW1: select 8259A-2 init */
00301:      outb_pic(0x20 + 8, PIC_SLAVE_IMR);  /* ICW2: 8259A-2 IR0-7 mapped to 0x28-0x2f */
00302:      outb_pic(PIC_CASCADE_IR, PIC_SLAVE_IMR);   /* 8259A-2 is a slave on master's IR2 */
00303:      outb_pic(SLAVE_ICW4_DEFAULT, PIC_SLAVE_IMR); /* (slave's support for AEOI in flat mode is
```

8259A：禁止/激活某个IRQ线

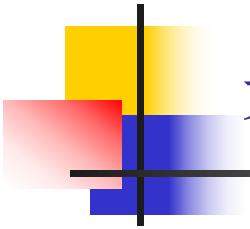
arch/x86/kernel/i8259_32.c

```
00063: void disable_8259A_irq(unsigned int irq)
00064: {
00065:     unsigned int mask = 1 << irq;
00066:     unsigned long flags;
00067:
00068:     spin_lock_irqsave(&i8259A_lock, flags);
00069:     cached_irq_mask |= mask;
00070:     if (irq & 8)
00071:         outb(cached_slave_mask, PIC_SLAVE_IMR);
00072:     else
00073:         outb(cached_master_mask, PIC_MASTER_IMR);
00074:     spin_unlock_irqrestore(&i8259A_lock, flags);
00075: }
```

```
able_8259A_irq(unsigned int irq)

gned int mask = ~(1 << irq);
gned long flags;

_lock_irqsave(&i8259A_lock, flags);
hed_irq_mask &= mask;
rq & 8)
 outb(cached_slave_mask, PIC_SLAVE_IMR);

 outb(cached_master_mask, PIC_MASTER_IMR);
_unlock_irqrestore(&i8259A_lock, flags);
```

```
00047: /*
00048:  * This contains the irq mask for both 8259A irq controllers
00049:  */
00050: unsigned int cached_irq_mask = 0xffff;
00008: #define __byte(x, y)        (((unsigned char *)&(y))[x])
00009: #define cached_master_mask  (__byte(0, cached_irq_mask))
00010: #define cached_slave_mask   (__byte(1, cached_irq_mask))
```

取变量的第x个字节



异常

- X86处理器发布了大约20种不同的异常。
- 某些异常通过硬件出错码说明跟异常相关的信息
- 内核为每个异常提供了一个专门的异常处理程序

#	Exception	Exception handler	Signal
0	Divide error	divide_error( )	SIGFPE
1	Debug	debug( )	SIGTRAP
2	NMI	)	None
3	Breakpoint		SIGTRAP
4	Overflow	overflow( )	SIGSEGV
5	Bounds check	bounds( )	SIGSEGV
6	Invalid opcode	invalid_op( )	SIGILL
7	Device not available	device_not_available( )	SIGSEGV
8	Double fault	double_fault( )	SIGSEGV
9	Coprocessor segment overrun	coprocessor_segment_overrun( )	SIGFPE
10	Invalid TSS	invalid_tss( )	SIGSEGV
11	Segment not present	segment_not_present( )	SIGBUS
12	Stack exception	stack_segment( )	SIGBUS
13	General protection	general_protection( )	SIGSEGV
14	Page Fault	e_fault( )	SIGSEGV
15	Intel reserved	None	None
16	Floating-point error	coprocessor_error( )	SIGFPE
17	Alignment check	alignment_check( )	SIGBUS
18	Machine check	ine_check( )	None
19	SIMD floating point	simd_coprocessor_error( )	SIGFPE

故障

非屏蔽中断

陷阱，断点调试

陷阱

故障，缺页

异常中止

异常处理程序

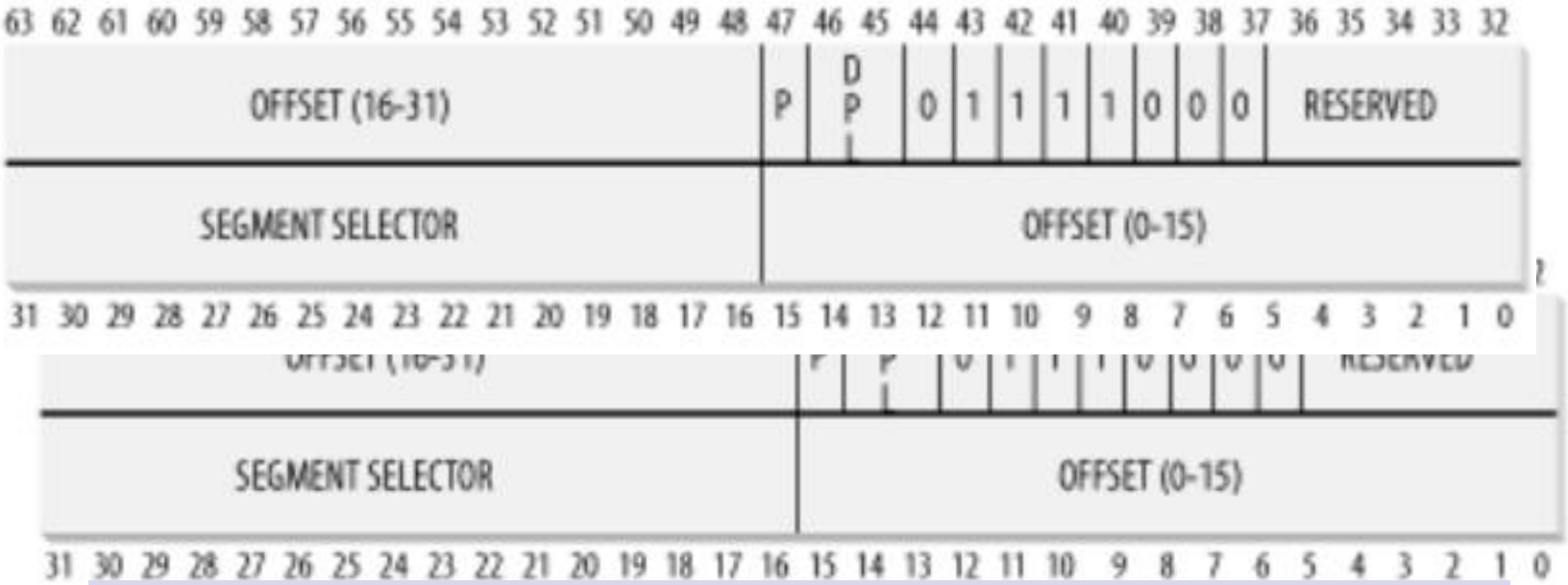
异常处理程序发出的信号

中断描述符表(Interrupt Descriptor Table，IDT)

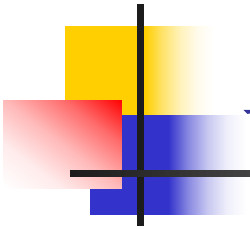
- 中断描述符表是一个系统表，它与每一个中断或者异常向量相联系
  - 每个中断或异常向量在表中有相应的中断或者异常处理程序的入口地址。
  - 每个描述符8个字节，共256项，占用空间2KB
  - 内核在允许中断发生前，必须适当的初始化IDT
- CPU的idtr寄存器指向IDT表的物理基地址
  - lidt指令

- IDT包含3种类型的描述符

Trap Gate Descriptor



陷阱门：与中断门类似，但进入陷阱门时，系统不会进入关中断状态

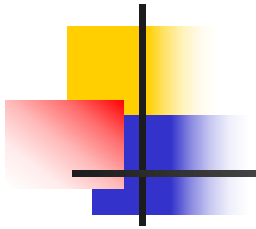


主要内容

- 中断信号的作用和中断信号处理的一般原则
- I/O设备如何引起CPU中断
- **x86 CPU如何在硬件级处理中断信号**
- Linux内核中软件级中断处理及其数据结构
- Linux的软中断、tasklet以及下半部分

中断和异常的硬件处理
进入中断/异常

- 假定：内核已经初始化，CPU在保护模式下运行
- CPU的正常运行：
  - 当执行了一条指令后，cs和eip这对寄存器包含了下一条将要执行的指令的逻辑地址。
  - 在执行这条指令之前，CPU控制单元会检查在运行前一条指令时是否发生了一个中断或者异常。
  - 如果发生了一个中断或异常，那么CPU控制单元执行下列操作：



1， 确定与中断或者异常关联的向量i（0~255）

2，读idtr寄存器指向的IDT表中的第i项

3，从gdtr寄存器获得GDT的基地址，并在GDT中查找，以读取IDT表项中的段选择符所标识的段描述符

4，确定中断是由授权的发生源发出的。

- 中断：中断处理程序的特权不能低于引起中断的程序的特权（数值小于等于CPL）
- 编程异常：还需比较CPL与对应IDT表项中的DPL

这个描述符指定中断或异常处理程序所在段

禁止低特权级用户访问特殊的门

只允许从低特权级“陷入”到高特权级，反之不可以

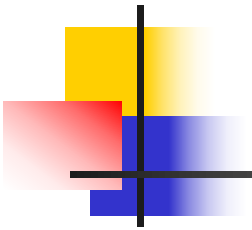
5，检查是否发生了特权级的变化，一般指是否由用户态陷入了内核态。

如果是由用户态陷入了内核态，控制单元必须开始使用与新的特权级相关的堆栈

a，读tr寄存器，访问运行进程的tss段

b，用与新特权级相关的栈段和栈指针装载ss和esp寄存器。这些值可以在进程的tss段中找到

c，在新的栈中保存ss和esp以前的值，这些值指明了与旧特权级相关的栈的逻辑地址



6，若发生的是故障，用引起异常的指令地址修改cs和eip寄存器的值，以使得这条指令在异常处理结束后能被再次执行

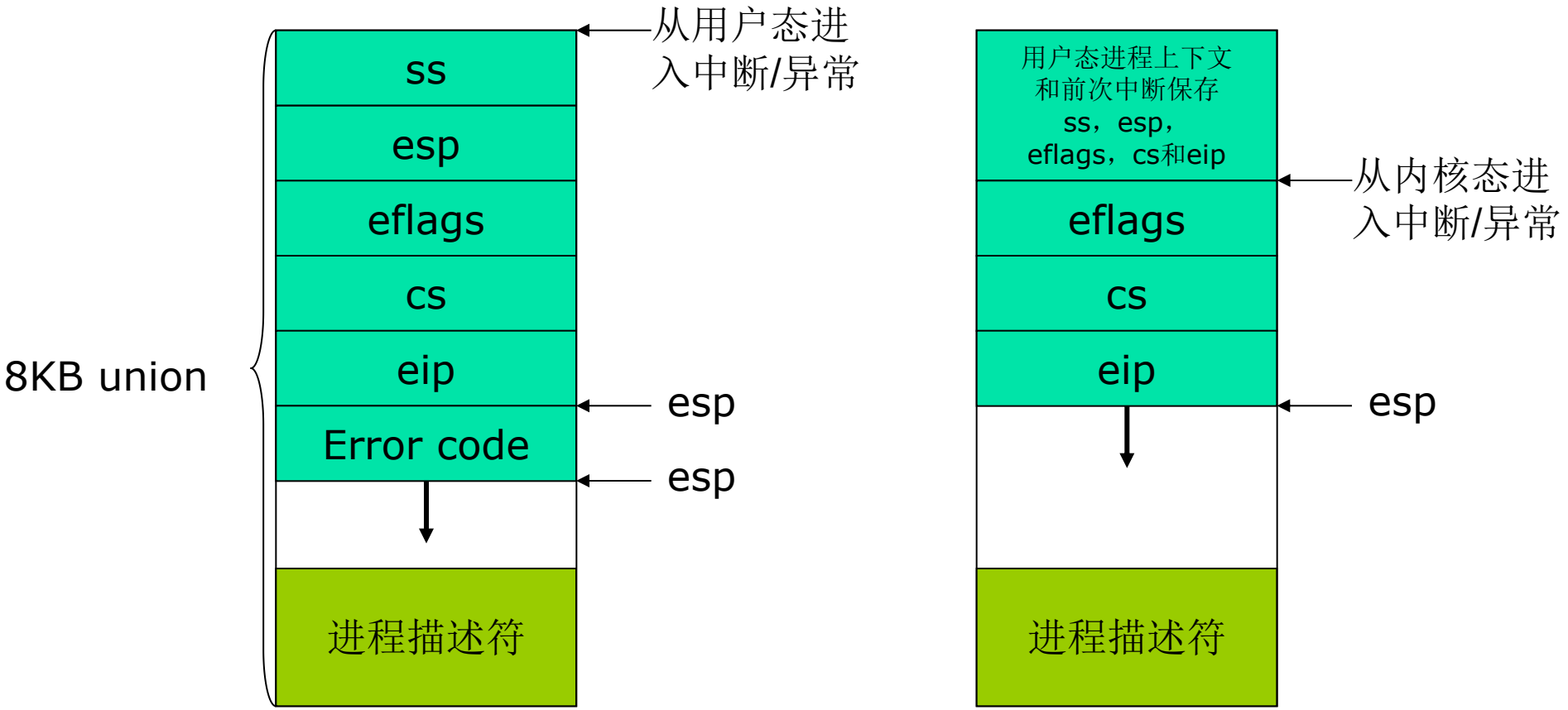
7，在栈中保存eflags、cs和eip的内容

8，如果异常产生一个硬件出错码，则将它保存在栈中

9，装载cs和eip寄存器，其值分别是IDT表中第i项门描述符的段选择符和偏移量字段。这对寄存器值给出中断或者异常处理程序的第一条指定的逻辑地址

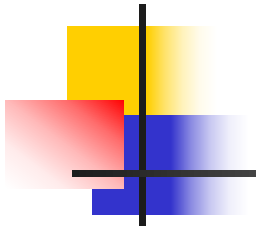
此时的进程内核态堆栈

(注意此进程可以是任意一个进程，中断处理程序不关心这个)



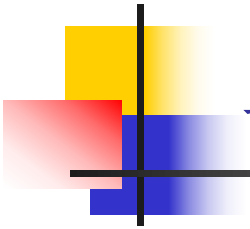
从中断/异常返回

- 中断/异常处理完后，相应的处理程序会执行一条iret汇编指令，这条汇编指令让CPU控制单元做如下事情:
  1，用保存在栈中的值装载cs、eip和eflags寄存器。如果一个硬件出错码曾被压入栈中，那么弹出这个硬件出错码
  2，检查处理程序的特权级是否等于cs中最低两位的值(这意味着进程在被中断的时候是运行在内核态还是用户态)。若是，iret终止执行；否则，转入3



3，从栈中装载ss和esp寄存器。这步意味着返回到与旧特权级相关的栈

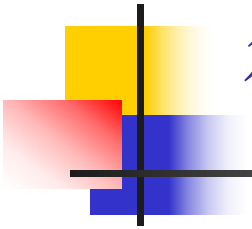
4，检查ds、es、fs和gs段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且特权级比当前特权级高，则清除相应的寄存器。这么做是防止怀有恶意的用户程序利用这些寄存器访问内核空间



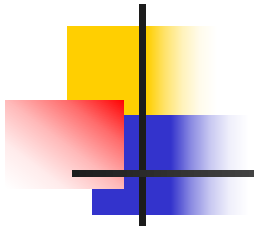
主要内容

- 中断信号的作用和中断信号处理的一般原则
- I/O设备如何引起CPU中断
- x86 CPU如何在硬件级处理中断信号
- **Linux内核中软件级中断处理及其数据结构**
- Linux的软中断、tasklet以及下半部分

中断和异常处理程序的嵌套执行



- 当内核处理一个中断或异常时，就开始了一个新的内核控制路径
- 当CPU正在执行一个与中断相关的内核控制路径时，linux不允许进程切换。不过，一个中断处理程序可以被另外一个中断处理程序中断，这就是中断的嵌套执行



- 抢占原则
  - 普通进程可以被中断或异常处理程序打断
  - 异常处理程序可以被中断程序打断
  - 中断程序只可能被其他的中断程序打断
- Linux允许中断嵌套的原因
  - 提高可编程中断控制器和设备控制器的吞吐量
  - 实现了一种没有优先级的中断模型

初始化中断描述符表

- 内核启动中断前，必须初始化IDT，然后把IDT的基地址装载到idtr寄存器中
- int指令允许用户进程发出一个中断信号，其值可以是0－255的任意一个向量。

  所以，为了防止用户用int指令非法模拟中断和异常，IDT的初始化时要很小心的设置特权级，可以通过把相应描述符的DPL字段设置成0来实现。
- 然而用户进程有时必须要能发出一个编程异常。为了做到这一点，只要把相应的中断或陷阱门描述符的特权级设置成3

初始化中断描述符表

- Linux中的中断门、陷阱门和系统门定义
  - 中断门
    - 用户态的进程不能访问的一个Intel中断门(特权级为0)，所有的中断都通过中断门激活，并全部在内核态
  - 系统门
    - 用户态的进程可以访问的一个Intel陷阱门(特权级为3)，通过系统门来激活4个linux异常处理程序，它们的向量是3，4，5和128。因此，在用户态下可以发布int3，into，bound和int $0x80四条汇编指令
  - 陷阱门
    - 用户态的进程不能访问的一个Intel陷阱门(特权级为0)，大部分linux异常处理程序通过陷阱门激活

在系统引导的过程中，内核调用/arch/i386/kernel/traps.c文件中的trap_init函数来对中断进行初始化。该函数通过调用同一文件下的set_trap_gate等多个函数来对中断描述符进行初始化。

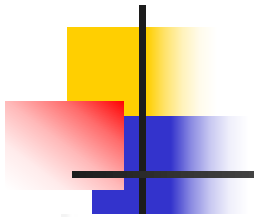
初始化中断描述符表

这几个函数都把相应的门中的段描述符设置成内核代码段的选择符，偏移字段设置成addr。

```
00308: static inline void set_intr_gate(unsigned int n, void *addr)
00309: {
00310:         BUG_ON((unsigned)n > 0xFF);
00311:         _set_gate(n, GATE_INTERRUPT, addr, 0, 0, __KERNEL_CS);
00312: }

00314: /*
00315:  * This routine sets up an interrupt gate at directory privilege level 3.
00316:  */
00317: static inline void set_system_intr_gate(unsigned int n, void *addr)
00318: {
00319:         BUG_ON((unsigned)n > 0xFF);
00320:         _set_gate(n, GATE_INTERRUPT, addr, 0x3, 0, __KERNEL_CS);
00321: }
```

include/asm-x86/desc.h

不同的是系统门中特权级对应的位DPL被置成3。



```
00323: static inline void set_trap_gate(unsigned int n, void *addr)
00324: {
00325:         BUG_ON((unsigned)n > 0xFF);
00326:         _set_gate(n, GATE_TRAP, addr, 0, 0, __KERNEL_CS);
00327: }
00328:
00329: static inline void set_system_gate(unsigned int n, void *addr
00330: {
00331:         BUG_ON((unsigned)n > 0xFF);
00332: #ifdef CONFIG_X86_32
00333:         _set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
00334: #else
00335:         _set_gate(n, GATE_INTERRUPT, addr, 0x3, 0, __KERNEL_CS
00336: #endif
00337: }
```

```
00290: static inline void _set_gate(int gate, unsigned type, void *addr,
00291:                              unsigned dpl, unsigned ist, unsigned seg)
00292: {
00293:     gate_desc s;
00294:     pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
00295:     /*
00296:      * does not need to be atomic because it is only done once at
00297:      * setup time
00298:      */
00299:     write_idt_entry(idt_table, gate, &s);
00300: }
```

- 其中，pack_gate函数的主要工作是设置相应的中断描述符，a为低32位，b为高32位。write_idt_entry的工作则是将在pack_gate函数中设置好的中断描述符填入到中断描述符表的相应位置。

```
00064: static inline void pack_gate(gate_desc *gate, unsigned char type,
00065:                              unsigned long base, unsigned dpl, unsigned flags,
00066:                              unsigned short seg)
00067: {
00068:     gate->a = (seg << 16) | (base & 0xffff);
00069:     gate->b = (base & 0xffff0000) |
00070:         (((0x80 | type | (dpl << 5)) & 0xff) << 8);
00071: }
```

write_idt_entry宏展开后代码如下：

QUOTE:

```
#define write_idt_entry(dt, entry, a, b) write_dt_entry(dt, entry, a, b)


static inline void write_dt_entry(void *dt, int entry, __u32 entry_a, __u32 entry_b)
{
__u32 *lp = (__u32 *)((char *)dt + entry*8);
*lp = entry_a;
*(lp+1) = entry_b;
}
```

进入保护模式前IDT表的初始化

arch/x86/boot/pm.c

```
00139: /*
00140:  * Set up the IDT
00141:  */
00142: static void setup_idt(void)
00143: {
00144:         static const struct gdt_ptr null_idt = {0, 0};
00145:         asm volatile("lidtl %0" : : "m" (null_idt));
00146: }
```

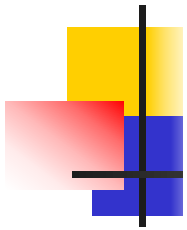
IDT的初步初始化（head_32.S）

arch/x86/kernel/head_32.S

- 用ignore_int()函数填充256个idt_table表项

```
setup_idt:
        lea ignore_int,%edx
        movl $(__KERNEL_CS << 16),%eax
        movw %dx,%ax                    / * selector = 0x0010 = cs */
        movw $0x8E00,%dx                / * interrupt gate - dpl=0, present *

        lea idt_table,%edi
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi
        dec %ecx
        jne rp_sidt
```

注意：此后还有关于异常相关入口的调整，使用了宏



```
ignore_int:
        cld
#ifdef CONFIG_PRINTK
        pushl %eax
        pushl %ecx
        pushl %edx
        pushl %es
        pushl %ds
        movl $(__KERNEL_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        cmpl $2,early_recursion_flag
        je hlt_loop
        incl early_recursion_flag
        pushl 16(%esp)
        pushl 24(%esp)
        pushl 32(%esp)
        pushl 40(%esp)
        pushl $int_msg
#ifdef CONFIG_EARLY_PRINTK
        call early_printk
#else
        call printk
#endif
        addl $(5*4),%esp
        popl %ds
        popl %es
        popl %edx
        popl %ecx
        popl %eax
#endif
        iret
```

arch/x86/kernel/head_32.S

```
int_msg:
        .asciz "Unknown interrupt or fault at EIP %p %p %p\n"

fault_msg:
/* fault info: */
        .ascii "BUG: Int %d: CR2 %p\n"
/* pusha regs: */
        .ascii "    EDI %p  ESI %p  EBP %p  ESP %p\n"
        .ascii "    EBX %p  EDX %p  ECX %p  EAX %p\n"
/* fault frame: */
        .ascii "    err %p  EIP %p   CS %p  flg %p\n"
        .ascii "Stack: %p %p %p %p %p %p %p %p\n"
        .ascii "       %p %p %p %p %p %p %p %p\n"
        .asciz "       %p %p %p %p %p %p %p %p\n"
```

Start_kernel中的IDT表初始化

- trap_init()
- init_IRQ()
  - 阅读native_init_IRQ

在init/main.c中:

```
asmlinkage void __init start_kernel(void)
{
……
//设定系统规定的异常/中断
trap_init();
//设置外部IRQ中断
init_IRQ();
……
}
```

trap_init()函数片段

```
void __init trap_init(void)
{
.....
set_trap_gate(0,&divide_error);
set_intr_gate(1,&debug);
set_intr_gate(2,&nmi);
set_system_intr_gate(3, &int3); /* int3-5 can be called from all */
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,GDT_ENTRY_DOUBLEFAULT_TSS);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
```

异常处理

- CPU产生的大部分异常都由linux解释为出错条件。当一个异常发生时，内核就向引起异常的进程发送一个信号通知它发生了一个反常条件。例如，如果进程执行了一个除0操作，CPU就产生一个“Divide error”异常，并由相应的处理程序向当前进程发送相应信号，让进程采取相应措施或直接终止进程执行。
- 异常处理有一个标准的结构，由三部分组成
  1. 在内核态堆栈中保存大多数寄存器的内容
  2. 调用C语言的函数
  3. 通过ret_from_exception()从异常处理程序退出
- 观察entry_32.S，并找到C语言函数的定义之处

arch/x86/kernel/entry_32.S

```
KPROBE_ENTRY(page_fault)
        RING0_EC_FRAME
        pushl $do_page_fault
        CFI_ADJUST_CFA_OFFSET 4
        ALIGN
error_code:
```

arch/x86/mm/fault.c

```
00573: /*
00574:  * This routine handles page faults.  It determines the address,
00575:  * and the problem, and then passes it off to one of the appropriate
00576:  * routines.
00577:  */
00578: #ifdef CONFIG_X86_64
00579: asmlinkage
00580: #endif
00581: void __kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code)
00582: {
```

arch/x86/kernel/entry_32.S

```
ENTRY(divide_error)
        RING0_INT_FRAME
        pushl $0                        # no error code
        CFI_ADJUST_CFA_OFFSET 4
        pushl $do_divide_error
        CFI_ADJUST_CFA_OFFSET 4
        jmp error_code
        CFI_ENDPROC
END(divide_error)
```

arch/x86/kernel/trap_32.S

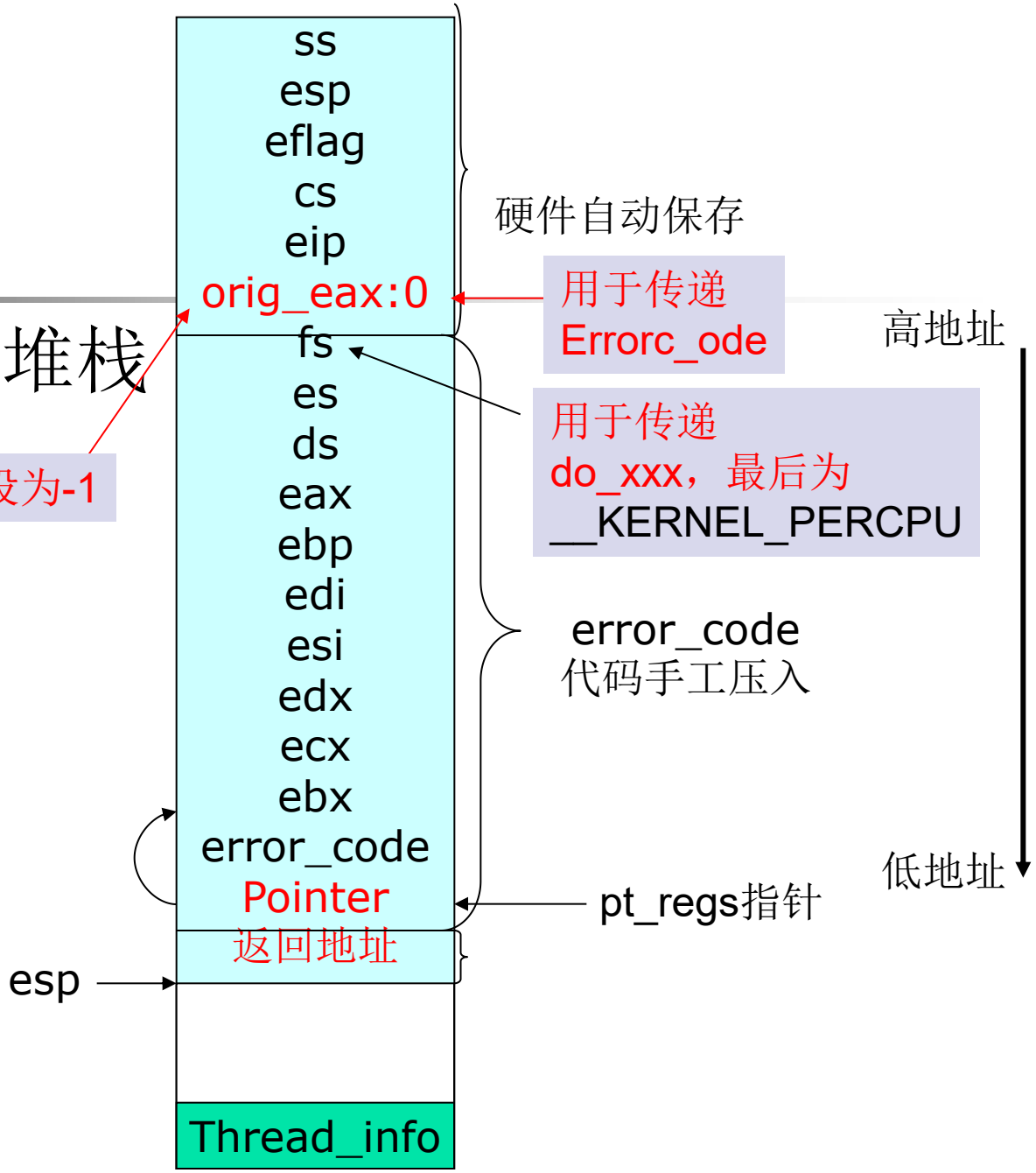
```
00579: #define DO_VM86_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr)     \
00580: void do_##name(struct pt_regs *regs, long error_code)                     \
00581: {                                                                       \
00582:     siginfo_t info;                                                     \
00583:     info.si_signo = signr;                                              \
00584:     info.si_errno = 0;                                                  \
00585:     info.si_code = sicode;                                              \
00586:     info.si_addr = (void __user *)siaddr;                               \
00587:     trace_hardirqs_fixup();                                             \
00588:     if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr)\
00589:                 == NOTIFY_STOP)                                         \
00590:         return;                                                         \
00591:     do_trap(trapnr, signr, str, 1, regs, error_code, &info);   \
00592: }
00593:
00594: DO_VM86_ERROR_INFO(0, SIGFPE,  "divide error", divide_error, FPE_INTDIV, regs->ip)
00595: #ifndef CONFIG_KPROBES
00596: DO_VM86_ERROR(3, SIGTRAP, "int3", int3)
00597: #endif
00598: DO_VM86_ERROR(4, SIGSEGV, "overflow", overflow)
00599: DO_VM86_ERROR(5, SIGSEGV, "bounds", bounds)
```

arch/x86/kernel/entry_32.S

- 阅读error_code
  - 该函数的主要功能：
    - 按照pt_regs结构定义的堆栈数据格式完成相应的入栈操作，进一步完成现场的保存
    - 把堆栈地址中的do_handler_name()函数的地址装入edi寄存器中，并在这个位置写入fs值，使栈结构进一步与pt_regs结构完全一致。
    - 最后执行call *%edi指令

```
KPROBE_ENTRY(page_fault)
        RINGO_EC_FRAME
        pushl $do_page_fault
        CFI_ADJUST_CFA_OFFSET 4
        ALIGN
error_code:
        /* the function address is in %fs's slot on the
stack */
        pushl %es
        CFI_ADJUST_CFA_OFFSET 4
        /*CFI_REL_OFFSET es, 0*/
        pushl %ds
        CFI_ADJUST_CFA_OFFSET 4
        /*CFI_REL_OFFSET ds, 0*/
        pushl %eax
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET eax, 0
        pushl %ebp
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET ebp, 0
        pushl %edi
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET edi, 0
        pushl %esi
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET esi, 0
        pushl %edx
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET edx, 0
        pushl %ecx
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET ecx, 0
        pushl %ebx
        CFI_ADJUST_CFA_OFFSET 4
        CFI_REL_OFFSET ebx, 0
        cld
        pushl %fs
        CFI_ADJUST_CFA_OFFSET 4
        /*CFI_REL_OFFSET fs, 0*/
        movl $(__KERNEL_PERCPU), %ecx
        movl %ecx, %fs
        UNWIND_ESPFIX_STACK
        popl %ecx
        CFI_ADJUST_CFA_OFFSET -4
        /*CFI_REGISTER es, ecx*/
        movl PT_FS(%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
        mov  %ecx, PT_FS(%esp)
        /*CFI_REL_OFFSET fs, ES*/
        movl $(__USER_DS), %ecx
        movl %ecx, %ds
        movl %ecx, %es
        movl %esp,%eax                  # pt_regs pointer
        call *%edi
        jmp ret_from_exception
        CFI_ENDPROC
KPROBE_END(page_fault)
```

此时的内核态堆栈



当异常发生时，如果控制单元没有自动地把一个硬件错误代码插入到栈中，相应的汇编语言片段会包含一条pushl $0指令，在栈中垫上一个空值,如果错误码已经被压入堆栈,则没有这条指令。然后，把异常处理函数的地址压进栈中；函数的名字由异常处理程序名与do_前缀组成。

pt_regs结构(恢复现场所需的上下文)

include/asm-x86/ptrace.h

栈顶（低地址）

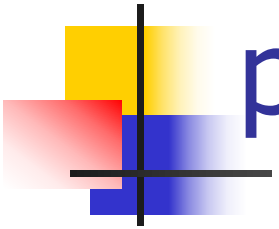
```
00038: struct pt_regs {
00039:         unsigned long bx;
00040:         unsigned long cx;
00041:         unsigned long dx;
00042:         unsigned long si;
00043:         unsigned long di;
00044:         unsigned long bp;
00045:         unsigned long ax;
00046:         unsigned long ds;
00047:         unsigned long es;
00048:         unsigned long fs;
00049:         /* int  gs; */
00050:         unsigned long orig_ax;
00051:         unsigned long ip;
00052:         unsigned long cs;
00053:         unsigned long flags;
00054:         unsigned long sp;
00055:         unsigned long ss;
00056: };
```

1. SAVE_ALL 和 RESTORE_ALL 保存和恢复的寄存器
2. 异常处理函数中的 Error_code 为保持一致而保存的数

1. 中断（狭）和系统调用保存的中断号和系统调用号
2. 或者，CPU 为产生硬件错误码的异常保存的硬件错误码
3. 或者，为保持一致，在异常处理函数中，随便保存的一个无效的数

CPU 在进入中断（广）前自动保存的寄存器

栈底（高地址）



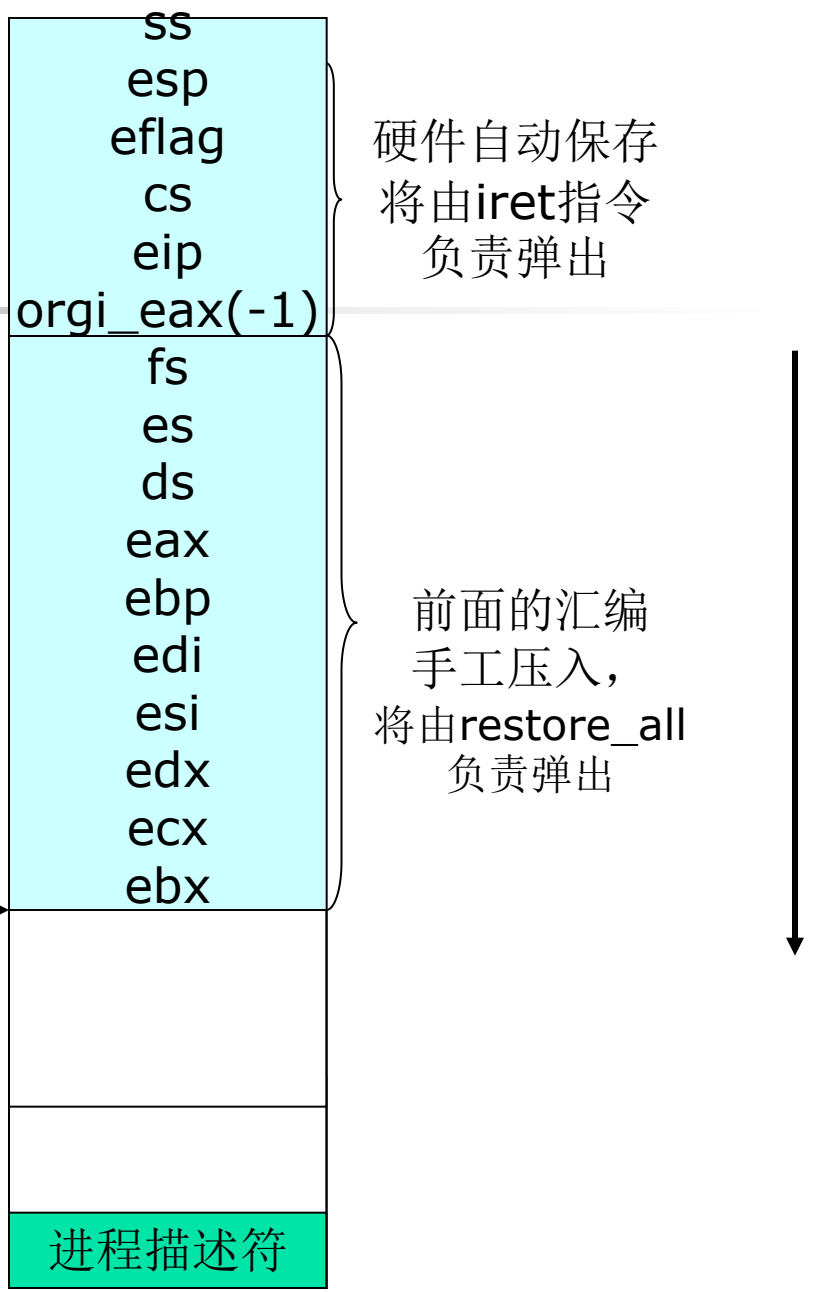
pt_regs结构

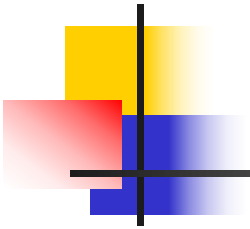
```
*       0(%esp) - %ebx
*       4(%esp) - %ecx
*       8(%esp) - %edx
*       C(%esp) - %esi
*      10(%esp) - %edi
*      14(%esp) - %ebp
*      18(%esp) - %eax
*      1C(%esp) - %ds
*      20(%esp) - %es
*      24(%esp) - %fs
*      28(%esp) - orig_eax
*      2C(%esp) - %eip
*      30(%esp) - %cs
*      34(%esp) - %eflags
*      38(%esp) - %oldesp
*      3C(%esp) - %oldss
```

异常处理

- 当C函数终止时，根据堆栈中的返回地址，CPU从call *%edi这条指令的下一条指令开始继续执行，即：

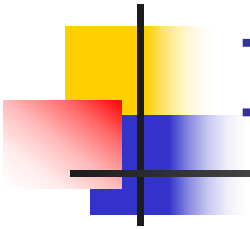
jmp ret_from_exception





中断处理

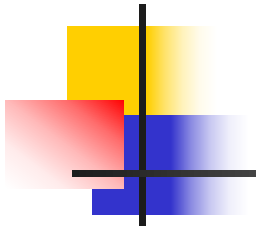
- 中断跟异常不同，它并不是表示程序出错，而是硬件设备有所动作，所以不是简单地往当前进程发送一个信号就OK的
- 主要有三种类型的中断：
  - I/O设备发出中断请求
  - 时钟中断
  - 处理器间中断(在SMP, Symmetric Multiprocessor上才会有这种中断)



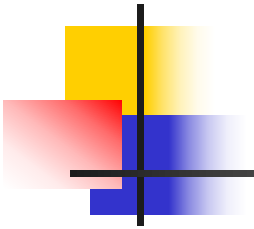
I/O中断处理

- I/O中断处理程序必须足够灵活以给多个设备同时提供服务
  - 比如几个设备可以共享同一个IRQ线(2个8359级联也只能提供15根IRQ线，所以外设共享IRQ线是很正常的)

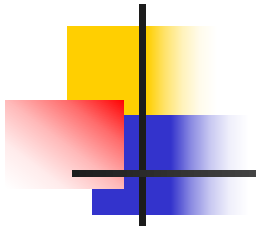
    这就意味着仅仅中断向量解决不了全部问题



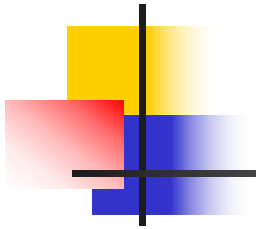
- 灵活性以两种不同的方式达到
  - IRQ共享：
    中断处理程序执行多个中断服务例程(interrupt service routines, ISRs)。每个ISR是一个与单独设备(共享IRQ线)相关的函数
  - IRQ动态分配：一条IRQ线在可能的最后时刻才与一个设备相关联



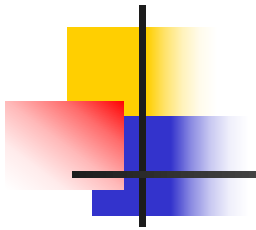
- 为了保证系统对外部的响应，一个中断处理程序必须被尽快的完成。因此，把所有的操作都放在中断处理程序中并不合适
- Linux中把紧随中断要执行的操作分为三类
  - 紧急的(critical)
    一般关中断运行。诸如对PIC应答中断，对PIC或是硬件控制器重新编程，或者修改由设备和处理器同时访问的数据



- 非紧急的(noncritical)
  如修改那些只有处理器才会访问的数据结构(例如按下一个键后读扫描码)，这些也要很快完成，因此由中断处理程序立即执行，不过一般在开中断的情况下



- 非紧急可延迟的(noncritical deferrable)
  如把缓冲区内容拷贝到某个进程的地址空间(例如把键盘缓冲区内容发送到终端处理程序进程)。这些操作可以被延迟较长的时间间隔而不影响内核操作，有兴趣的进程将会等待数据。内核用下半部分这样一个机制来在一个更为合适的时机用独立的函数来执行这些操作



不管引起中断的设备是什么，所有的I/O中断处理程序都执行四个相同的基本操作

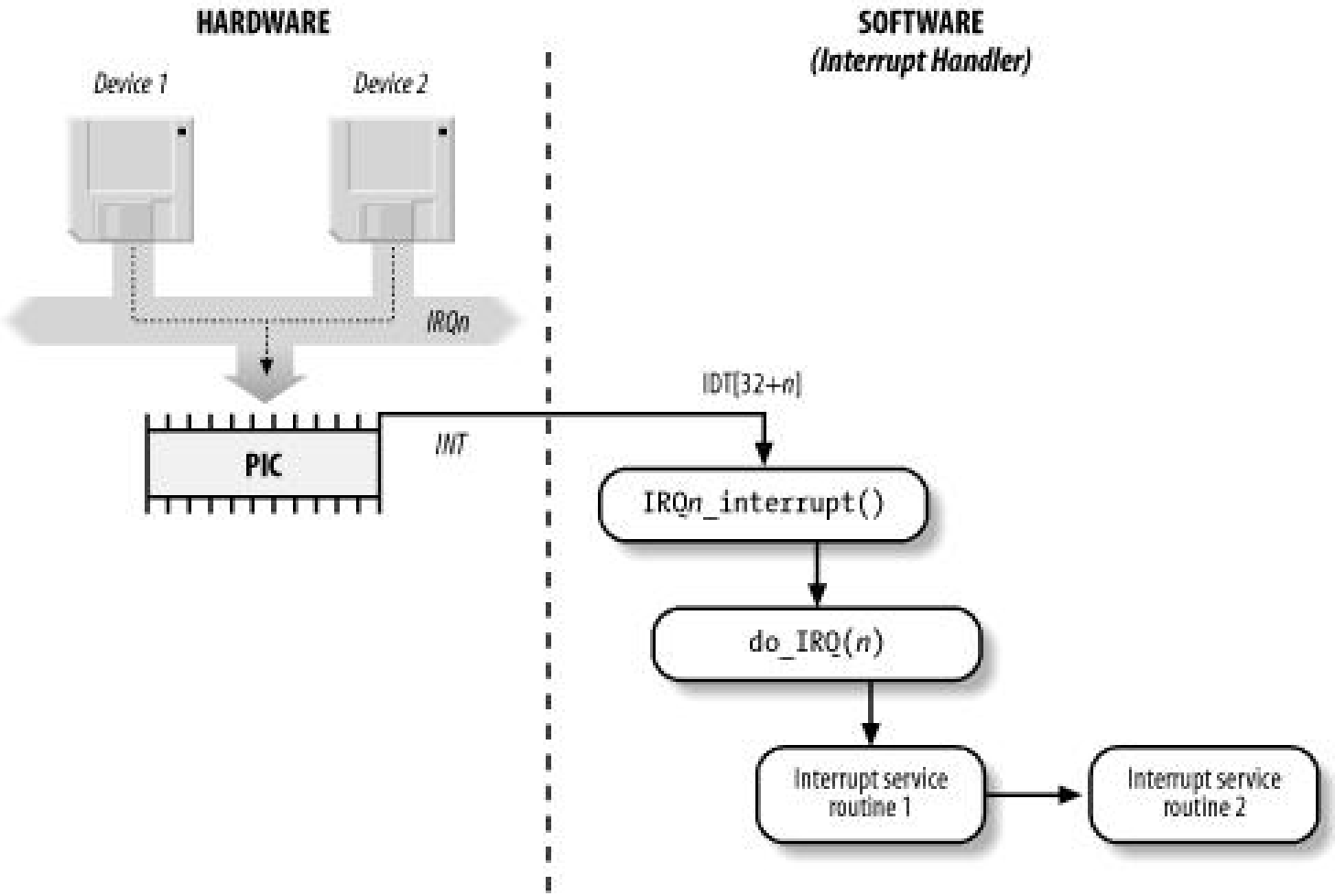
1，在内核态堆栈保存IRQ的值和寄存器的内容

2，为正在给IRQ线服务的PIC发送一个应答，这将允许PIC进一步发出中断

3，执行共享这个IRQ的所有设备的中断服务例程

4，跳到ret_from_intr()的地址后中断跳出

中断处理示意图


HARDWARE
SOFTWARE
(Interrupt Handler)
Device 1
Device 2
IRQn
IDT[32+n]
INT
PIC
IRQn_interrupt()
do_IRQ(n)
Interrupt service routine 1
Interrupt service routine 2

Linux中的中断向量分配表

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
240-250 (0xf0-0xfa)	Reserved by Linux for future use
251-255 (0xfb-0xff)	Interprocessor interrupts

Linux中的设备中断

IRQ号与I/O设备之间的对应关系是在初始化每个设备驱动程序时建立的

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

中断处理

- 系统初始化时，调用init_IRQ()函数用新的中断门替换临时中断门来更新IDT

arch/x86/kernel/i8259_32.c

```
00387:      for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
00388:           int vector = FIRST_EXTERNAL_VECTOR + i;
00389:           if (i >= NR_IRQS)
00390:                break;
00391:           /* SYSCALL_VECTOR was reserved in trap_init. */
00392:           if (!test_bit(vector, used_vectors))
00393:                set_intr_gate(vector, interrupt[i]);
00394:      }
```

这段代码在interrupt数组中找到用于建立中断门的中断处理程序地址。

各函数的地址被复制到IDT相应表项的中断门中

Interrupt数组的定义(比较隐晦)

arch/x86/kernel/entry_32.S

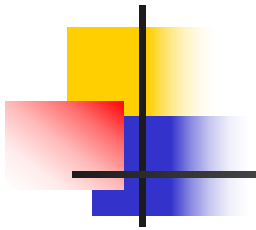
```
/*
 * Build the entry stubs and pointer table wi
 * some assembler magic.
 */
.section .rodata,"a"
ENTRY(interrupt)
.text

ENTRY(irq_entries_start)
        RING0_INT_FRAME
vector=0
.rept NR_IRQS
        ALIGN
 .if vector
        CFI_ADJUST_CFA_OFFSET -4
 .endif
1:      pushl $~(vector)
        CFI_ADJUST_CFA_OFFSET 4
        jmp common_interrupt
 .previous
        .long 1b
 .text
vector=vector+1
.endr
END(irq_entries_start)

.previous
END(interrupt)
```

```
/*
 * the CPU automatically disables interrupts when exec
 * so IRQ-flags tracing has to follow that:
 */
        ALIGN
common_interrupt:
        SAVE_ALL
        TRACE_IRQS_OFF
        movl %esp,%eax
        call do_IRQ
        jmp ret_from_intr
ENDPROC(common_interrupt)
        CFI_ENDPROC
```

Save_all代码片段

```
00038: struct pt_regs {
00039:         unsigned long bx;
00040:         unsigned long cx;
00041:         unsigned long dx;
00042:         unsigned long si;
00043:         unsigned long di;
00044:         unsigned long bp;
00045:         unsigned long ax;
00046:         unsigned long ds;
00047:         unsigned long es;
00048:         unsigned long fs;
00049:         /* int  gs; */
00050:         unsigned long orig_ax;
00051:         unsigned long ip;
00052:         unsigned long cs;
00053:         unsigned long flags;
00054:         unsigned long sp;
00055:         unsigned long ss;
00056: };
```

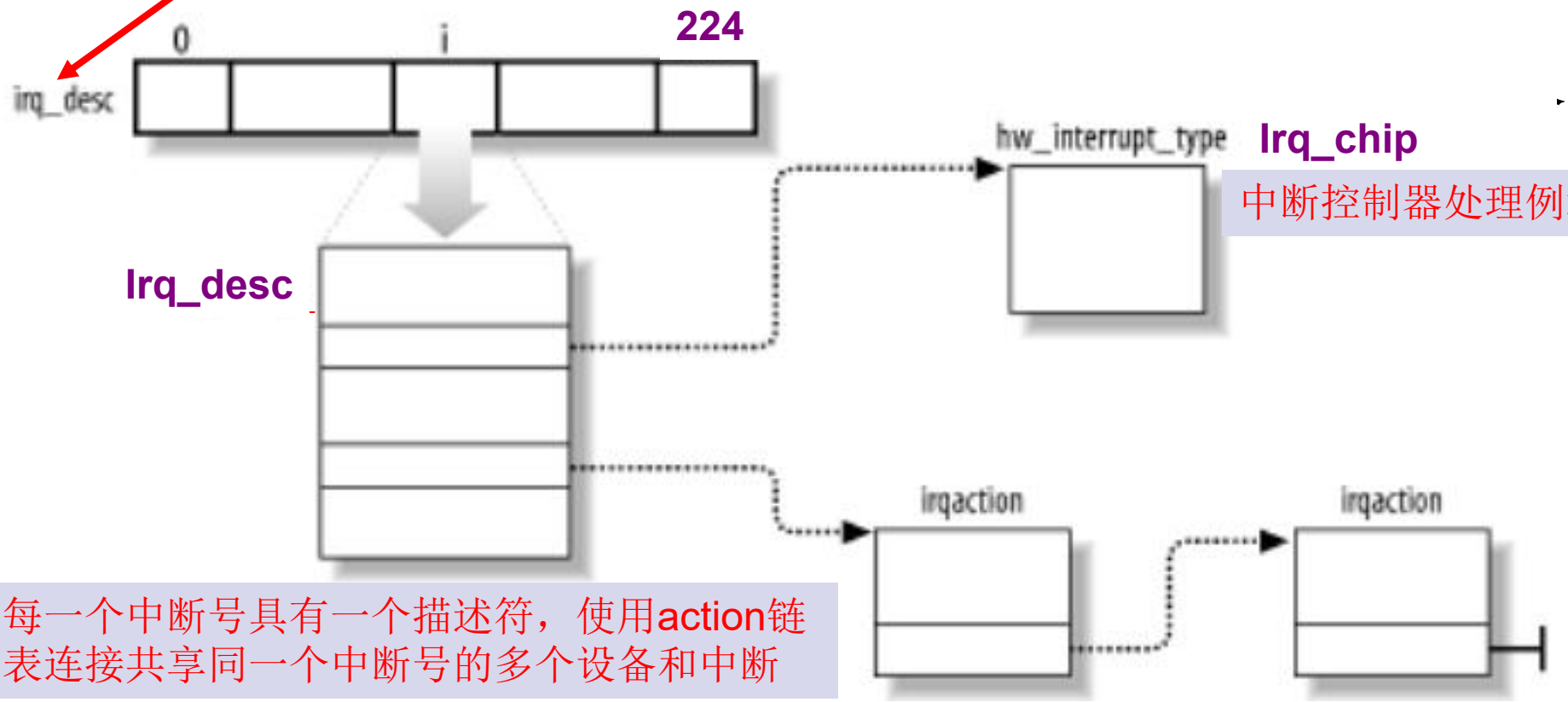
```
#define SAVE_ALL \
        cld; \
        pushl %fs; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET fs, 0;*/\
        pushl %es; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET es, 0;*/\
        pushl %ds; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET ds, 0;*/\
        pushl %eax; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET eax, 0;\
        pushl %ebp; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ebp, 0;\
        pushl %edi; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET edi, 0;\
        pushl %esi; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET esi, 0;\
        pushl %edx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET edx, 0;\
        pushl %ecx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ecx, 0;\
        pushl %ebx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ebx, 0;\
```



- 因此，每个中断程序入口操作为：
  - 将中断向量入栈
  - 保存所有其他寄存器
  - 调用do_IRQ
  - 跳转到ret_from_intr

do_IRQ（查看do_IRQ源码）

- do_IRQ使用的数据结构（体系结构无



每一个中断号具有一个描述符，使用action链表连接共享同一个中断号的多个设备和中断

include/linux/irq.h

kernel/irq/handle.c

- 查看相关数据结构
- 查看irq_desc数组的定义和最初的初始化

```
00050: struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
00051:     [0 ... NR_IRQS-1] = {
00052:         .status = IRQ_DISABLED,
00053:         .chip = &no_irq_chip,
00054:         .handle_irq = handle_bad_irq,
00055:         .depth = 1,
00056:         .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
00057: #ifdef CONFIG_SMP
00058:         .affinity = CPU_MASK_ALL
00059: #endif
00060:     }
00061: };
```

irqaction数据结构

- 用来实现IRQ的共享，维护共享irq的特定设备和特定中断，所有共享一个irq的链接在一个action表中，由中断描述符中的action指针指向

```
00060: struct irqaction {
00061:         irq_handler_t handler;
00062:         unsigned long flags;
00063:         cpumask_t mask;
00064:         const char *name;
00065:         void *dev_id;
00066:         struct irqaction *next;
00067:         int irq;
00068:         struct proc_dir_entry *dir;
00069: };
```

中断处理程序

链表

include/linux/interrupt.h

- 设置irqaction的函数：setup_irq

irq_chip数据结构

- 为特定PIC编写的低级I/O例程
- 例如8259的

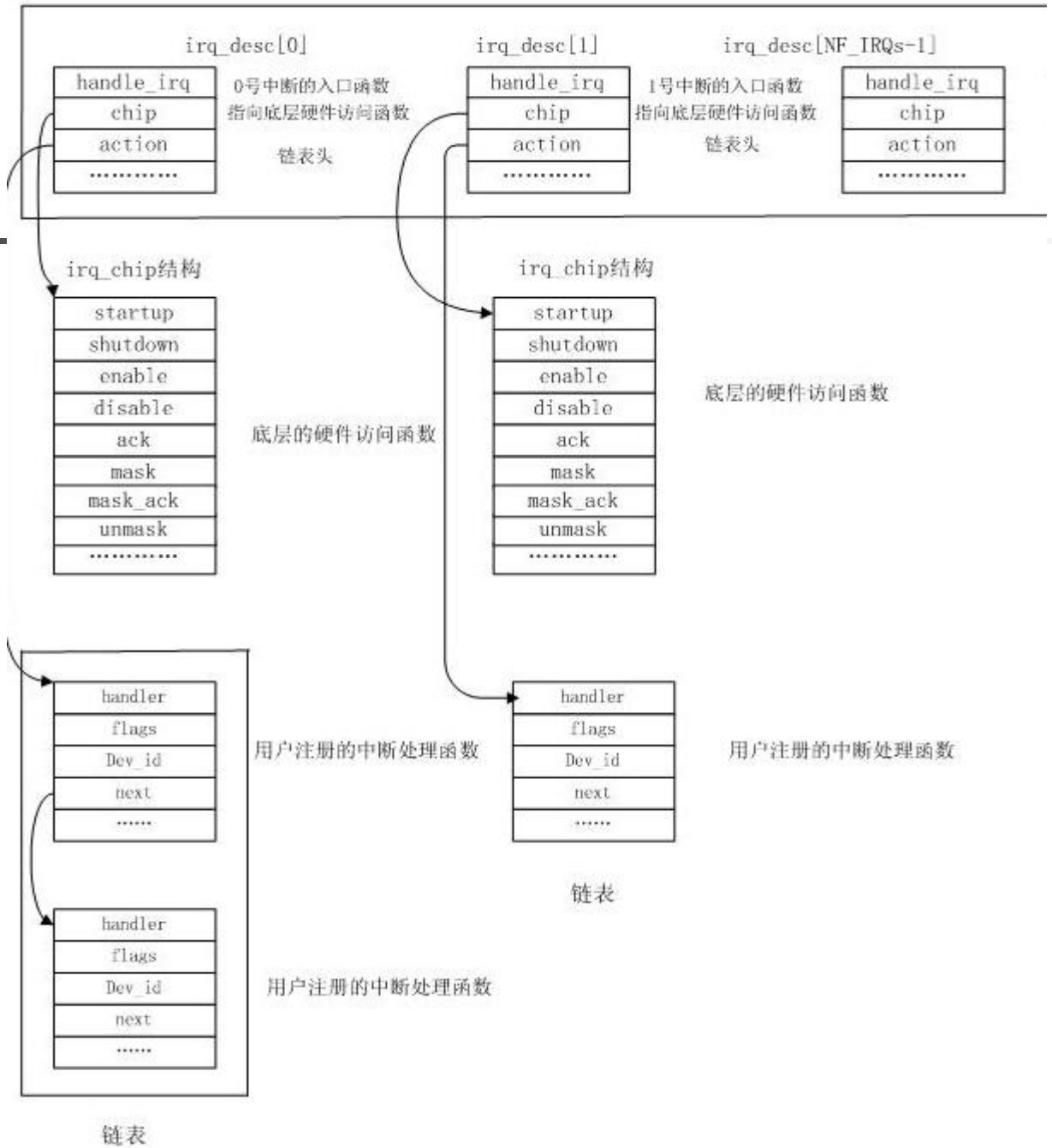
arch/x86/kernel/i8259_32.c

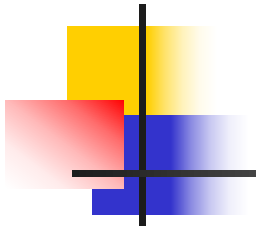
```
00035: static struct irq_chip i8259A_chip = {
00036:         .name           = "XT-PIC",
00037:         .mask           = disable_8259A_irq,
00038:         .disable        = disable_8259A_irq,
00039:         .unmask         = enable_8259A_irq,
00040:         .mask_ack       = mask_and_ack_8259A,
00041: };
```

set_irq_chip

- 为一个中断设置irq_chip
  - set_irq_chip_and_handler_name等

irq_desc结构数组：
每个数组项用来描述一个中断；
中断总入口函数asm_do_IRQ根据中断号调用里面的handle_irq成员

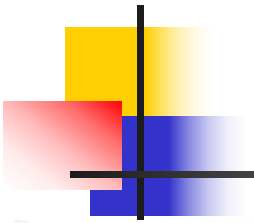




- 例如：在init_IRQ（即native_init_IRQ）`能如下

```
00030: void __init pre_intr_init_hook(void)
00031: {
00032:         init_ISA_irqs();
00033: }
```

arch/x86/mach-default/setup.c



arch/x86/kernel/i8259_32.c

```
00354: void __init init_ISA_irqs (void)
00355: {
00356:         int i;
00357:
00358: #ifdef CONFIG_X86_LOCAL_APIC
00359:         init_bsp_APIC();
00360: #endif
00361:         init_8259A(0);
00362:
00363:         /*
00364:          * 16 old-style INTA-cycle interrupts:
00365:          */
00366:         for (i = 0; i < 16; i++) {
00367:                 set_irq_chip_and_handler_name(i, &i8259A_chip,
00368:                                               handle_level_irq, "XT");
00369:         }
00370: }
```

arch/x86/kernel/i8259_32.c

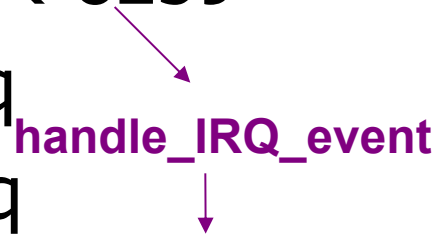
- 又如：make_8259A_irq

```
00107: void make_8259A_irq(unsigned int irq)
00108: {
00109:     disable_irq_nosync(irq);
00110:     io_apic_irqs &= ~(1<<irq);
00111:     set_irq_chip_and_handler_name(irq, &i8259A_chip, handle_level_irq,
00112:                                   "XT");
00113:     enable_irq(irq);
00114: }
```

irq_flow_handler_t

- __set_irq_handler设置handle_irq数据项

- handle_level_irq ←8259
- handle_simple_irq
- handle_fasteoi_irq
- handle_edge_irq
- handle_percpu_irq



Action→handle

- 在setup_irq时，给定

arch/x86/mach-default/setup.c

```
00100: void __init time_init_hook(void)
00101: {
00102:         irq0.mask = cpumask_of_cpu(0);
00103:         setup_irq(0, &irq0);
00104: }
```

```
00086: static struct irqaction irq0  = {
00087:         .handler = timer_interrupt,
00088:         .flags = IRQF_DISABLED | IRQF_NOBALANCING | IRQF_IRQPOLL
00089:         .mask = CPU_MASK_NONE,
00090:         .name = "timer"
00091: };
```

小结：中断处理过程

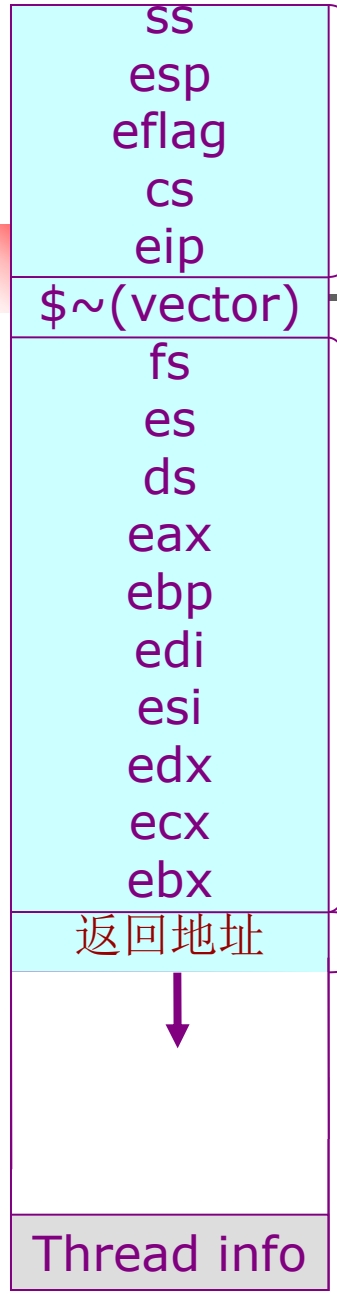
- 在调用do_IRQ之前，要为保存寄存器
  - 在interrupt数组中定义的中断处理程序
  - 每个入口地址转换成汇编码是如下的一

```
interrupt[irq]:
        pushl $~(vector)
        jmp common_interrupt
```

这里对所有的中断处理程序都执行相同的

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

```
#define SAVE_ALL \
        cld; \
        pushl %fs; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET fs, 0;*/\
        pushl %es; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET es, 0;*/\
        pushl %ds; \
        CFI_ADJUST_CFA_OFFSET 4;\
        /*CFI_REL_OFFSET ds, 0;*/\
        pushl %eax; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET eax, 0;\
        pushl %ebp; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ebp, 0;\
        pushl %edi; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET edi, 0;\
        pushl %esi; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET esi, 0;\
        pushl %edx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET edx, 0;\
        pushl %ecx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ecx, 0;\
        pushl %ebx; \
        CFI_ADJUST_CFA_OFFSET 4;\
        CFI_REL_OFFSET ebx, 0;\
        movl $(__USER_DS), %edx; \
        movl %edx, %ds; \
        movl %edx, %es; \
        movl $(__KERNEL_PERCPU), %edx; \
        movl %edx, %fs
```



硬件自动保存

SAVE_ALL

从do_IRQ返回后要执行的指令地址ret_from_intr

esp

do_IRQ()的函数声明

arch/x86/kernel/irq_32.c

```
00064: /*
00065:  * do_IRQ handles all normal device IRQ's (the special
00066:  * SMP cross-CPU interrupts have their own specific
00067:  * handlers).
00068:  */
00069: unsigned int do_IRQ(struct pt_regs *regs)
00070: {
```

do_IRQ执行时内核态的堆栈

中断处理

- do_IRQ()函数的等价代码：

```
int irq = ~regs->orig_ax;                    //1
irq_desc[irq]->handle_irq(irq, desc);        //2
    mask_ack_irq(desc, irq);                 //3
    handle_IRQ_event(irq,&regs,irq_desc[irq].action);//4
    irq_desc[irq].handler->end(irq);         //5
处理下半部分                                  //6
```

1句取得对应的中断向量

2句调用中断处理句柄，对8259，就是handle_level_irq

3句应答PIC的中断，并禁用这条IRQ线。(为串行处理同类型中断)

4调用handle_IRQ_event()执行中断服务例程，例如timer_interrupt

5句通知PIC重新激活这条IRQ线，允许处理同类型中断

中断服务例程

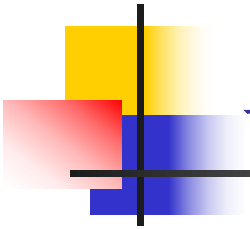
- 一个中断服务例程实现一种特定设备的操作， handle_IRQ_evnet()函数依次调用这些设备例程
  - 这个函数本质上执行了如下核心代码:

```
do{
action->handler(irq,action->dev_id,regs);
action = action->next;
}while (action)
```

注册外部中断

- 中断程序的注册
  - 一个模块被希望来请求一个中断通道(或者 IRQ, 对于中断请求), 在使用它之前要注册它, 并且当结束时释放它. 函数声明在 <linux/interrupt.h>, 实现中断注册接口:

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,

                const char *dev_name,
                void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

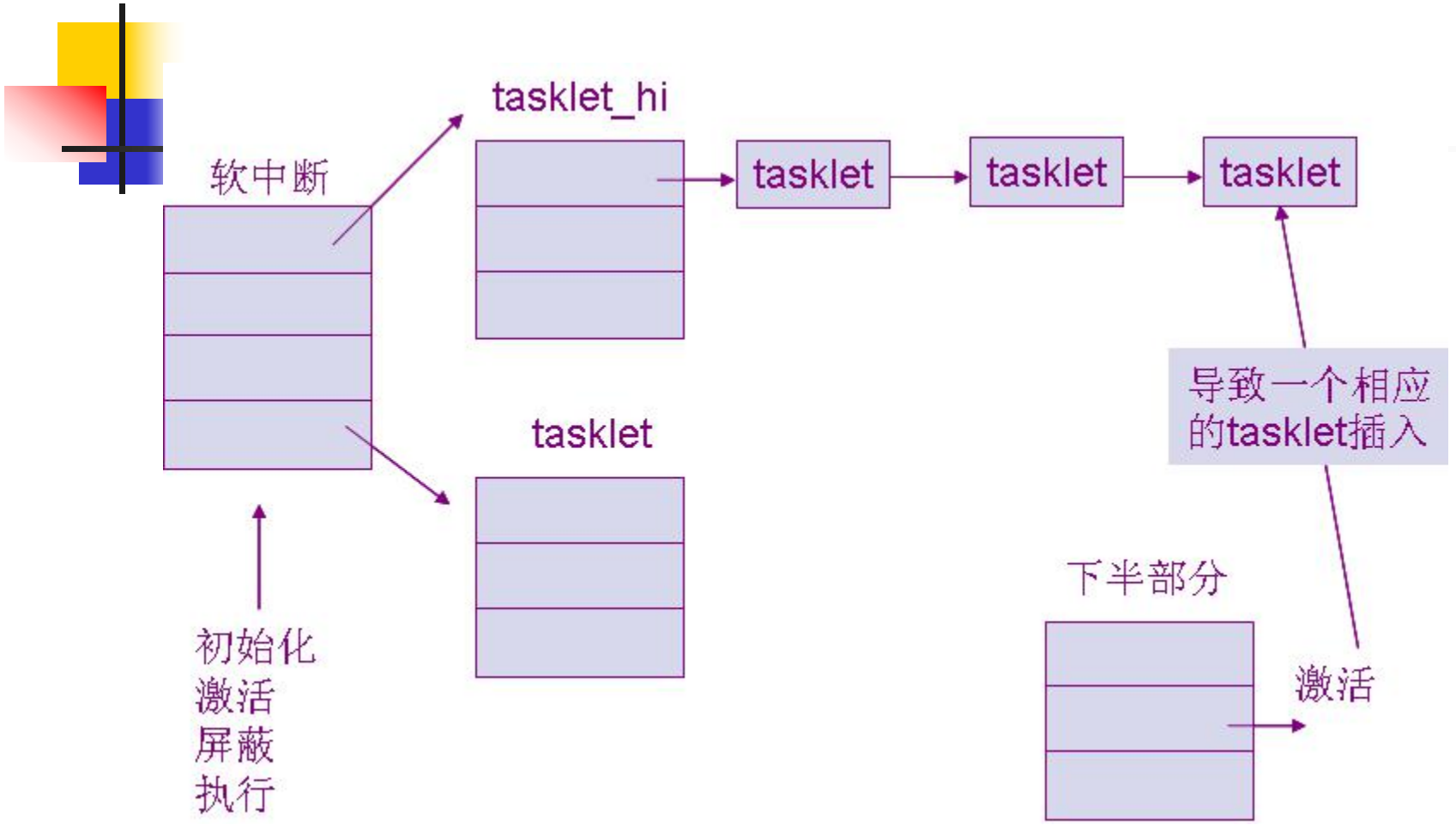


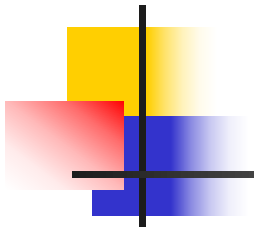
主要内容

- 中断信号的作用和中断信号处理的一般原则
- I/O设备如何引起CPU中断
- x86 CPU如何在硬件级处理中断信号
- Linux内核中软件级中断处理及其数据结构
- **Linux的软中断、tasklet以及下半部分**

软中断、tasklet以及下半部分

- 对内核来讲，可延迟中断不是很紧急，可以将它们从中断处理例程中抽取出来，保证较短的中断响应时间
- Linux2.6提供了三种方法
  - 可延迟的函数
    - 软中断、tasklet
      - Tasklet在软中断之上实现
      - 一般原则：在同一个CPU上软中断/tasklet不嵌套
      - 软中断由内核静态分配（编译时确定）
      - Tasklet可以在运行时分配和初始化（例如装入一个内核模块时）
  - 工作队列（ work queues ）


tasklet_hi
软中断
tasklet
tasklet
tasklet
导致一个相应
的tasklet插入
tasklet
下半部分
初始化
激活
屏蔽
执行
激活



- 一般而言，可延迟函数上可以执行4种操作
  - 初始化：定义一个新的可延迟函数，通常在内核初始化时进行
  - 激活：设置可延迟函数在下一轮处理中执行
  - 屏蔽：有选择的屏蔽一个可延迟函数，这样即使被激活也不会被运行
  - 执行：在特定的时间执行可延迟函数

软中断

include/linux/interrupt.h

- Linux2.6.26使用有限

```
00266: enum
00267: {
00268:      HI_SOFTIRQ=0,
00269:      TIMER_SOFTIRQ,
00270:      NET_TX_SOFTIRQ,
00271:      NET_RX_SOFTIRQ,
00272:      BLOCK_SOFTIRQ,
00273:      TASKLET_SOFTIRQ,
00274:      SCHED_SOFTIRQ,
00275: #ifdef CONFIG_HIGH_RES_TIMERS
00276:      HRTIMER_SOFTIRQ,
00277: #endif
00278:      RCU_SOFTIRQ,    /* Preferable RCU
00279: };
```

优先级0：处理高优先级的tasklet和下半部分

优先级1：与时钟中断相关的tasklet

优先级2：把数据包传送到网卡

优先级3：从网卡接受数据包

优先级4：块设备相关

优先级5：处理tasklet

优先级6：调度SMP相关

- 在softirq_vec中定义

kernel/softirq.c

```
00049: static struct softirq_action softirq_vec[32] __cacheline_aligned_i
```

优先级对应于softirq_vec的下标

include/linux/interrupt.h

```
struct softirq_action
{
        void (*action)(struct softirq_action *);
        void *data;
};
```

软中断函数及其参数

软中断的初始化

- 初始化软中断函数

kernel/softirq.c

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;
}
```

- 分别在softirq_init和net_dev_init、blk_dev_init等中初始化

例如 **kernel/softirq.c**

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
```

net/core/dev.c

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

软中断的触发

- raise_softirq include/asm-x86/hardirq_32.h

```
00007: typedef struct {
00008:         unsigned int __softirq_pending;
00009:         unsigned long idle_timestamp;
00010:         unsigned int __nmi_count;  /* arch dependent */
00011:         unsigned int apic_timer_irqs;     /* arch dependent */
00012:         unsigned int irq0_irqs;
00013:         unsigned int irq_resched_count;
00014:         unsigned int irq_call_count;
00015:         unsigned int irq_tlb_count;
00016:         unsigned int irq_thermal_count;
00017:         unsigned int irq_spurious_count;
00018: } ____cacheline_aligned irq_cpustat_t;

00020: DECLARE_PER_CPU(irq_cpustat_t, irq_stat);
```

软中断的检查

include/linux/irq_cpustat.h

- local_softirq_pending

- 在某些特定的时机，会检查是否有软中断被挂起
  - 调用local_bh_enable重新激活软中断时
  - 当do_IRQ完成了I/O中断的处理时

```
if (softirq_pending(cpu))
    do_softirq();
```

  - 当那个特定的进程**ksoftirqd**被唤醒时
  - …
- 这种时机，称为检查点

在每个检查点

- 若有软中断被挂起，就调用do_softirq
  - 判断是否可以执行软中断
  - 若可以，就执行软中断
    - 执行后，若发现又有新的软中断被激活，就唤醒ksoftirqd进程，来触发do_softirq的另一次执行

arch/x86/kernel/irq_32.c

```
00230:      h = softirq_vec;
00231:
00232:      do {
00233:          if (pending & 1) {
00234:              h->action(h);
00235:              rcu_bh_qsctr_inc(cpu);
00236:          }
00237:          h++;
00238:          pending >>= 1;
00239:      } while (pending);
```

```
if (pending)
    wakeup_softirqd();
```

Ksoftirqd内核线程

kernel/softirq.c

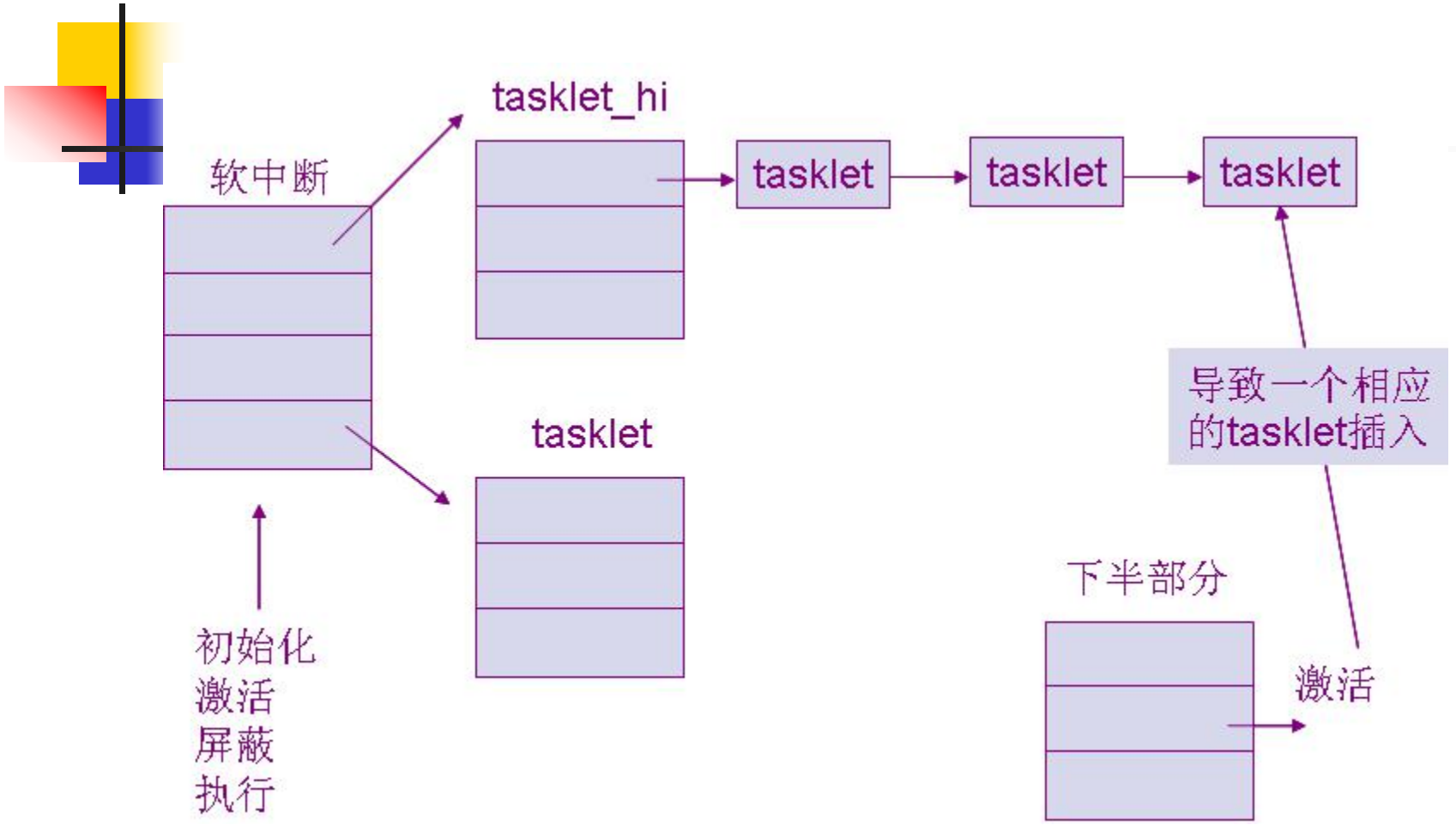
.....

```
00524:          while (local_softirq_pending()) {
00525:                  /* Preempt disable stops cpu going offline.
00526:                     If already offline, we'll be on wrong CPU:
00527:                     don't process */
00528:                  if (cpu_is_offline((long)__bind_cpu))
00529:                          goto ↓wait_to_die;
00530:                  do_softirq();
00531:                  preempt_enable_no_resched();
00532:                  cond_resched();
00533:                  preempt_disable();
00534:          }
```

.....

Tasklet

- Tasklet是I/O驱动程序中实现可延迟函数的首选方法
- 建立在HI_SOFTIRQ和TASKLET_SOFTIRQ等软中断之上
- Tasklet和高优先级的tasklet
  - 分别存放在tasklet_vec和tasklet_hi_vec数组中
    - 数组的每一项针对一个CPU，代表这个CPU上的tasklet列表
  - 分别由tasklet_action和tasklet_hi_action处理
    - 找到CPU对应的那个项，遍历执行


tasklet_hi
软中断
tasklet
tasklet
tasklet
导致一个相应
的tasklet插入
tasklet
下半部分
初始化
激活
屏蔽
执行
激活

include/linux/interrupt.h

```
00320: struct tasklet_struct
00321: {
00322:         struct tasklet_struct *next;
00323:         unsigned long state;
00324:         atomic_t count;
00325:         void (*func)(unsigned long);
00326:         unsigned long data;
00327: };
```

0：enable
>0：disable

kernel/softirq.c

```
00356: /* Tasklets */
00357: struct tasklet_head
00358: {
00359:         struct tasklet_struct *head;
00360:         struct tasklet_struct **tail;
00361: };
00363: /* Some compilers disobey section attribute on statics when not
00364:    initialized -- RR */
00365: static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec) = { NULL };
00366: static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec) = { NULL };
```

Tasklet的使用

- 当需要使用tasklet时，可以按照如下方法进行
  - 1、分配一个tasklet的数据结构，并初始化 ====相当于声明（定义）一个tasklet
  - 2、可以禁止/允许这个tasklet ====相当于定义了一个是否允许使用tasklet的窗口
  - 3、可以激活这个tasklet ====这个tasklet被插入task_vec或者task_hi_vec的相应CPU的链表上，将在合适的时机得到处理

激活tasklet的方法

include/linux/interrupt.h

- 即将tasklet插入到合适的链表中
  - Tasklet_schedule
  - Tasklet_hi_schedule

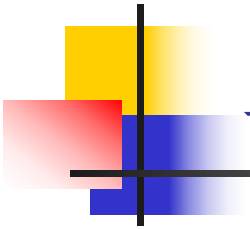
tasklet使用

- 如下：
- //定义一个处理函数
  - void my_tasklet_function(unsigned long);
- //定义了一个名叫my_tasklet的tasklet并将其与处理函数绑定，而传入参数为data
  - DECLARE_TASKLET(my_tasklet, my_tasklet_function, data);
- 在需要调度tasklet的时候引用一个tasklet_schedule()函数就能使系统在适当的时候进行调度运行：
  tasklet_schedule(&my_tasklet);

Tasklet使用模板

```
/*定义 tasklet 和底半部函数并关联*/
void xxx_do_tasklet(unsigned long);
DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);

/*中断处理底半部*/
void xxx_do_tasklet(unsigned long)
{
  ...
}
```

```
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {
14   ...
15   tasklet_schedule(&xxx_tasklet);
16   ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int __init xxx_init(void)
21 {
22   ...
23   /*申请中断*/
24   result = request_irq(xxx_irq, xxx_interrupt,
25         IRQF_DISABLED, "xxx", NULL);
26   ...
27   return IRQ_HANDLED;
28 }
29
30 /*设备驱动模块卸载函数*/
31 void __exit xxx_exit(void)
32 {
33   ...
34   /*释放中断*/
35   free_irq(xxx_irq, xxx_interrupt);
36   ...
37 }
```



工作队列workqueue

- 工作队列和tasklet这两种下半部机制的主要区别在于：
  - Tasklet在软中断的上下文中运行，所有的代码必须是原子的，不能睡眠、不能使用信号量或其它产生阻塞的函数；
  - 工作队列在一个内核线程上下文运行，并且可以在延迟一段确定的时间后才执行；有更多的灵活性，它可以使用信号量等能够睡眠的函数。

工作队列和工作线程

- 相关数据结构 **kernel/workqueue.c**
  - workqueue_struct；cpu_workqueue_struct

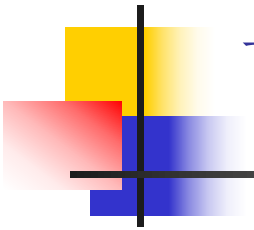
include/linux/workqueue.h

  - work_struct；delayed_work

kernel/workqueue.c

- 入列 queue_work；queue_delayed_work
- 工作队列的处理
  - run_workqueue
  - worker_thread

工作队列的使用（使用内核默认队列）

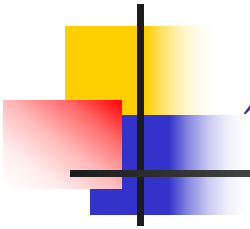


- 工作队列：使用方法和tasklet相似，如下：
  - struct work _struct my_wq; //创建一个工作队列my_wq
  - void my_wq_func(unsigned long); //定义一个处理函数
  - 通过INIT_WORK()可以初始化这个工作队列并将工作队列与处理函数绑定，如下：
    - INIT_WORK(&my_wq, (void (*)(void *))my_wq_func, NULL); //初始化工作队列并将其与处理函数绑定
  - 同样，使用schedule_work(&my_irq)；来在系统在适当的时候需要调度时使用运行。

代码清单 10.3　工作队列使用模板

```
1  /*定义工作队列和关联函数*/
2  struct work_struct xxx_wq;
3  void xxx_do_work(unsigned long);
4
5  /*中断处理底半部*/
6  void xxx_do_work(unsigned long)
```

```
/*中断处理顶半部*/
irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
 ...
 schedule_work(&xxx_wq);
 ...
 return IRQ_HANDLED;
}

/*设备驱动模块加载函数*/
int xxx_init(void)
{
 ...
 /*申请中断*/
 result = request_irq(xxx_irq, xxx_interrupt,
      IRQF_DISABLED, "xxx", NULL);
 ...
 /*初始化工作队列*/
 INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
 ...
}

/*设备驱动模块卸载函数*/
void xxx_exit(void)
{
 ...
 /*释放中断*/
 free_irq(xxx_irq, xxx_interrupt);
 ...
}
```



从中断和异常返回

- 中断和异常的终止目的很清楚，即恢复某个程序的执行，但是还有几个问题要考虑
  - 内核控制路径是否嵌套
    - 如果仅仅只有一条内核控制路径，那CPU必须切换到用户态
  - 挂起进程的切换请求
    - 如果有任何请求，必须调度；否则，当前进程得以运行
  - 挂起的信号
    - 如果一个信号发送到进程，那必须处理它
  - 等等

阅读Entry.S中从中断和异常返回的代码

- 阅读ULK3（中文，188页）中的图
  - 深入理解linux内核第三版
  - 出口函数根据中断发生时为内核态还是用户态分别调resume_kernel和resume_userspace处理

```
ret_from_exception:
    cli ; missing if kernel preemption is not supported
ret_from_intr:
    movl $-8192, %ebp ; -4096 if multiple Kernel Mode stacks are used
    andl %esp, %ebp
    movl 0x30(%esp), %eax
    movb 0x2c(%esp), %al
    testl $0x00020003. %eax
    jnz resume_userspace
    jpm resume_kernel
```

恢复内核控制路径 resume_kernel

- 允许内核抢占
  - 调用 need_resched
- 不允许直接回复现场

```
resume_kernel:
    cli
    cmpl $0, 0x14(%ebp)
    jz need_resched
restore_all:
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    popl %ebp
    popl %eax
    popl %ds
    popl %es
    addl $4, %esp
    iret
```

恢复用户态程序 resume_userspace:

- 该程序如不需要重新调度或处理挂起信号等直接回复现场，否则运行 work_pending

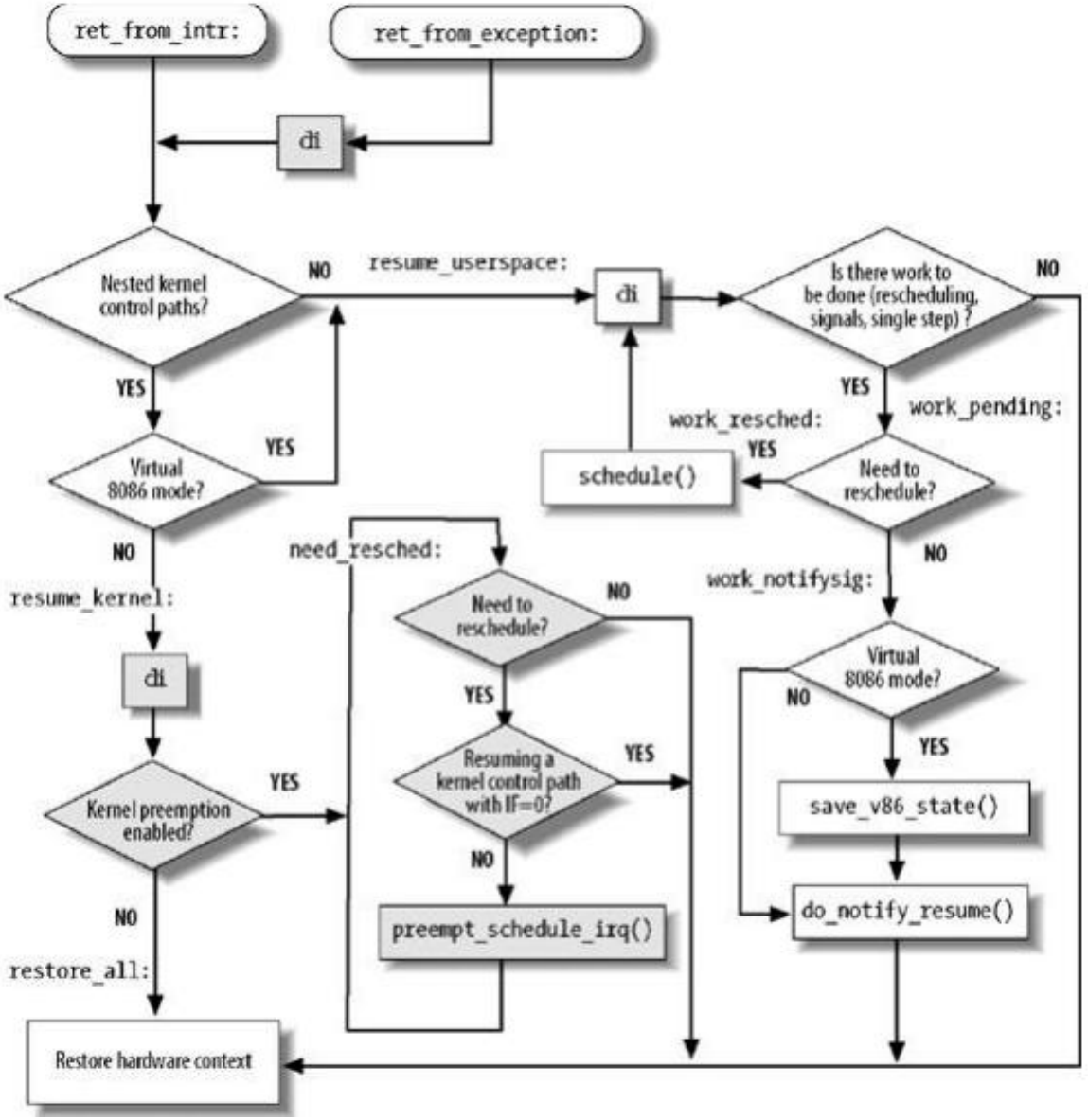
```
resume_userspace:
    cli
    movl 0x8(%ebp), %ecx
    andl $0x0000ff6e, %ecx
    je restore_all
    jmp work_pending
```

检测重调度标志并处理

- 如果进程切换标志被挂起，调用schedule()进行进程调度。
- 对挂起信号等的处理调用 work_notifysig

```
work_pending:
     testb $(1<<TIF_NEED_RESCHED), %cl
     jz work_notifysig
work_resched:
     call schedule
     cli
     jmp resume_userspace
```

中断和异常返回处理流程



关于thread_info描述符的解释

- 对于每一个进程而言，内核为其单独分配了一个内存区域，这个区域存储的是内核栈和该进程所对应的thread_info结构。

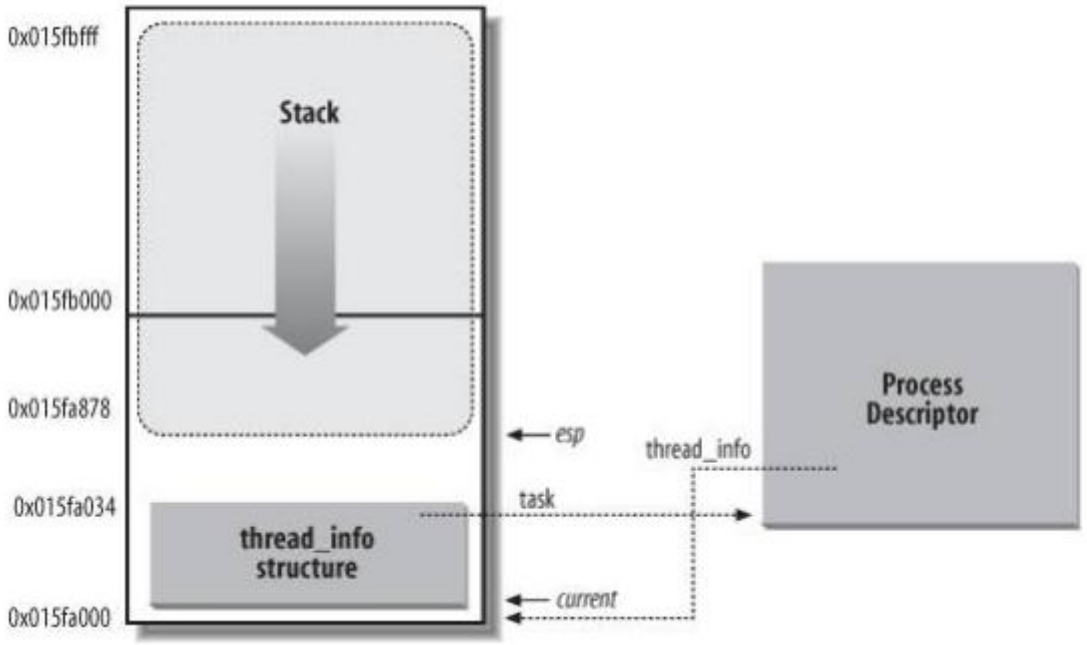
```
struct thread_info {
        struct task_struct   *task;        /* main task structure */
        struct exec_domain   *exec_domain; /* execution domain */
        unsigned long        flags;        /* low level flags */
        unsigned long        status;             /* thread-synchronous flags */
        __u32                cpu;          /* current CPU */
        int                  preempt_count;      /* 0 => preemptable,
                                                    <0 => BUG */
        mm_segment_t         addr_limit;   /* thread address space:
                                                    0-0xBFFFFFFF user-thread
                                                    0-0xFFFFFFFF kernel-thread
                                                 */
        void                 *sysenter_return;
        struct restart_block    restart_block;
        unsigned long           previous_esp;   /* ESP of the previous stack in
                                                   case of nested (IRQ) stacks
                                                 */
        __u8                 supervisor_stack[0];
};
```

关于thread_info中flags相关说明

include/asm/thread_info.32.h

```
/*
 * thread information flags
 * - these are process state flags that various
 *   assembly files may need to access
 * - pending work-to-be-done flags are in LSW
 * - other flags in MSW
 */
#define TIF_SYSCALL_TRACE   0      /* syscall trace active */
#define TIF_SIGPENDING             1      /* signal pending */
#define TIF_NEED_RESCHED    2      /* rescheduling necessary */
#define TIF_SINGLESTEP             3      /* restore singlestep on return to
                                     user mode */
#define TIF_IRET            4      /* return with iret */
#define TIF_SYSCALL_EMU            5      /* syscall emulation active */
#define TIF_SYSCALL_AUDIT   6      /* syscall auditing active */
#define TIF_SECCOMP         7      /* secure computing */
#define TIF_HRTICK_RESCHED  9      /* reprogram hrtick timer */
#define TIF_MEMDIE          16
#define TIF_DEBUG           17     /* uses debug registers */
#define TIF_IO_BITMAP              18     /* uses I/O bitmap */
#define TIF_FREEZE          19     /* is freezing for suspend */
#define TIF_NOTSC           20     /* TSC is not accessible in userland */
#define TIF_FORCED_TF              21     /* true if TF in eflags artificially */
#define TIF_DEBUGCTLMSR            22     /* uses thread_struct.debugctlmsr */
#define TIF_DS_AREA_MSR     23      /* uses thread_struct.ds_area_msr */
#define TIF_BTS_TRACE_TS        24      /* record scheduling event timestamps */
```

- 问题：为何ebp寄存器就可得到当前进程的thread_info?



系统调用

- 系统门的初始化

```
void __init trap_init(void)
{
.....

   #ifdef CONFIG_X86_32
set_system_trap_gate(SYSCALL_VECTOR, &system_call);
      set_bit(SYSCALL_VECTOR, used_vectors);
  #endif
.....
}
```

1 #ifdef CONFIG_X86_32 2 # define SYSCALL_VECTOR 0x80 3 #endif

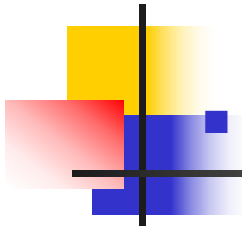
系统调用

- SYSCALL_VECTOR变量的定义
  - SYSCALL_VECTOR是系统调用的中断号，在/arch/x86/include/asm/irq_vectors.h中定义：

```
#ifdef CONFIG_X86_32
 # define SYSCALL_VECTOR          0x80
 #endif
```

它在/arch/x86/kernel/entry_32.S中定义

```
ENTRY(system_call)
    RING0_INT_FRAME            # can't unwind into user space anyway
    pushl_cfi %eax             # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
                 # system call tracing in operation / emulation
    testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp)        # store the return value
.....
```



- sys_call_table在/arch/x86/kernel/syscall_table_32.S中定义：

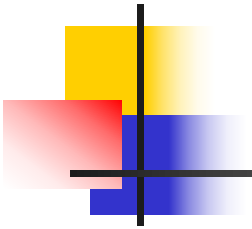
```
ENTRY(sys_call_table)
    .long sys_restart_syscall    /* 0 - old "setup()" system call, used for restarting */
    .long sys_exit
    .long ptregs_fork
    .long sys_read
    .long sys_write
    .long sys_open        /* 5 */
    .long sys_close
.....................
```

系统调用表与调用号

- 核心中为每个系统调用定义了一个唯一的编号，这个编号的定义在linux/include/asm/unistd.h中（最大为NR_syscall）
- 同时在内核中保存了一张系统调用表，该表中保存了系统调用编号和其对应的服务例程地址。第n个表项包含系统调用号为n的服务例程的地址。
- 系统调用陷入内核前，需要把系统调用号一起传入内核。而该标号实际上是系统调用表(sys_call_table)的下标
- 在i386上，这个传递动作是通过在执行int $0x80前把调用号装入eax寄存器实现。
- 这样系统调用处理程序一旦运行，就可以从eax中得到系统调用号，然后再去系统调用表中寻找相应服务例程。

系统调用代码解析

```
ENTRY(system_call)
    RING0_INT_FRAME
    ASM_CLAC
    pushl_cfi %eax          //保存系统调用号；
    SAVE_ALL                //可以用到的所有CPU寄存器保存到栈中
    GET_THREAD_INFO(%ebp)   //ebp用于存放当前进程thread_info结构的地址
    testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax  //检查系统调用号（系统调用号应小于NR_syscalls），
    jae syscall_badsys      //不合法，跳入到异常处理
syscall_call:
    call *sys_call_table(,%eax,4) //合法，对照系统调用号在系统调用表中寻找相应服务例程
    movl %eax,PT_EAX(%esp)      //保存返回值到栈中
syscall_exit:
    testl $_TIF_ALLWORK_MASK, %ecx   //检查是否需要处理信号
    jne syscall_exit_work       //需要,进入 syscall_exit_work
restore_all:
    TRACE_IRQS_IRET             //不需要,执行restore_all恢复，返回用户态
irq_return:
    INTERRUPT_RETURN            //相当于iret
```



- 谢谢大家认真听讲