Linux操作系统分析
Chapter 5 中断和异常

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

October 21, 2014

Outline

Outline

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
  - 轮询方式效率不高
- 内核应当处理其它任务，只有当外设真正完成/准备好了时才转过来处理外设IO
- 中断机制就是满足上述条件的一种解决办法。
  （回忆IO方式：轮询、中断、DMA等）

查看系统中断信息

- cat /proc/interrupts

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

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：（狭义）
    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，并切换到一个新的活动以响应这个中断信号
- 为了做到这这一点，
  在进程的内核态堆栈中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

1. 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)
2. 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
3. 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

中断上下文与进程上下文

- 程序的运行必然有上下文，中断处理程序亦然。
- 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，执行内核代码，在中断发生时正在运行的进程上下文中执行
  - 作为一个内核控制路径，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，建立和终止中断上下文所需要的时间很少)
- 假设：

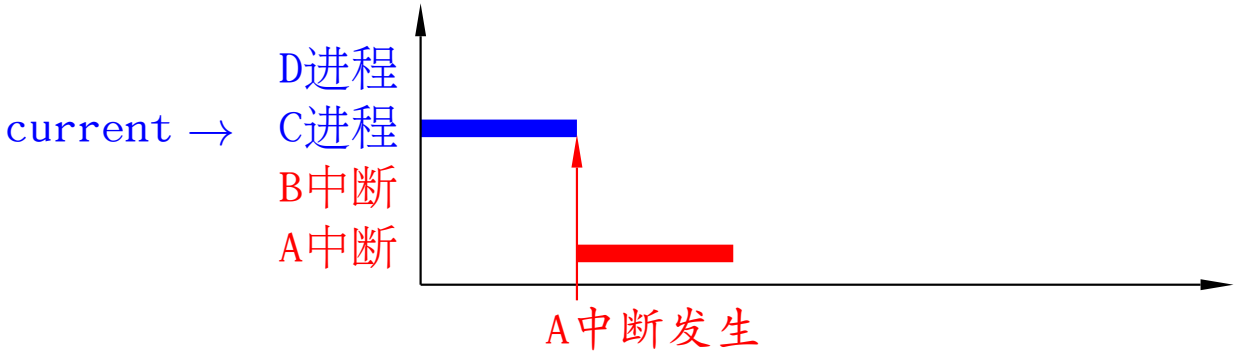
  2个interrupt context，记为A和B

  2个process，记为C和D

  分析A,B,C,D在互相抢占上的关系

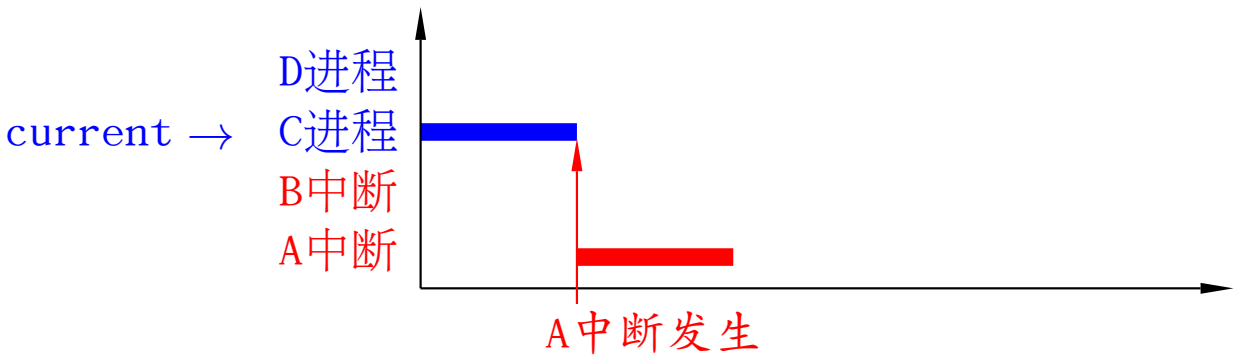
中断上下文与进程上下文

1. 假设某个时刻C占用CPU运行，此时A中断发生，
   则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



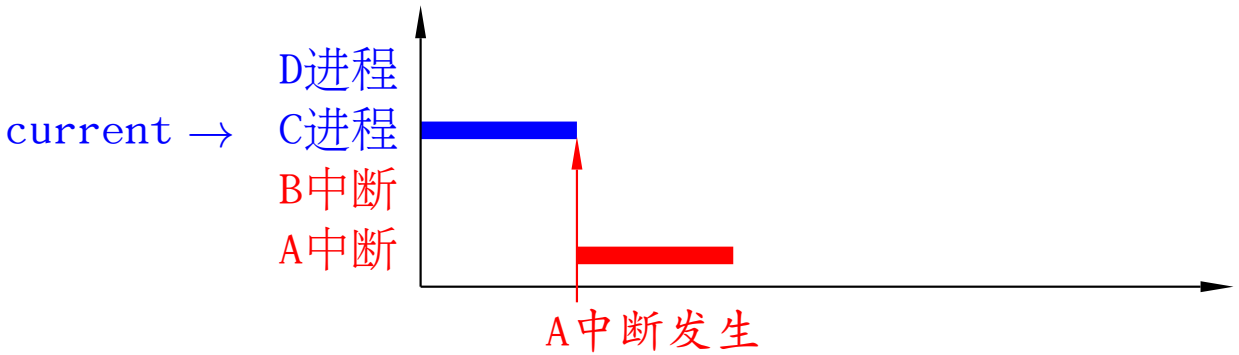
中断上下文与进程上下文

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。



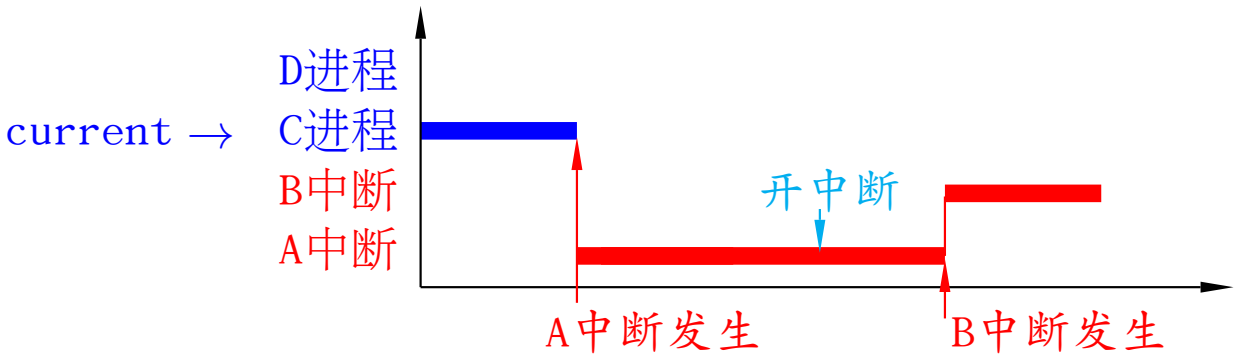
中断上下文与进程上下文

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



中断上下文与进程上下文

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



Outline

中断和异常（Intel文档）

- 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(masked)和非屏蔽(unmasked)
  - 非屏蔽中断（Nonmaskable interrupt, NMI）
    - 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

中断和异常（Intel文档）

- 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Linux使用int 0x80来实现系统调用

中断和异常（Intel文档）

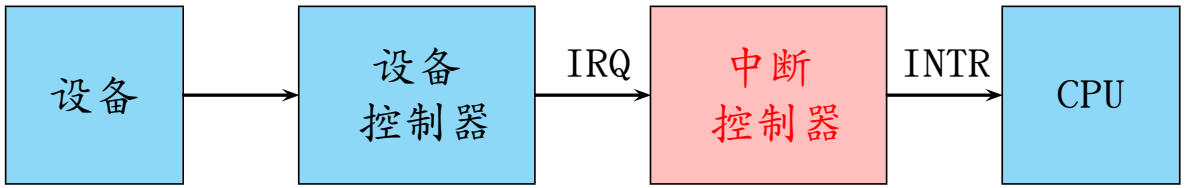
- 对于处理器探测异常，根据返回地址（即异常时保存在内核堆栈中的eip的值）的不同，可以进一步分为：
  1. 故障(fault)：eip=引起故障的指令的地址
     - 通常可以纠正，处理完异常时，该指令被重新执行
     - 例如缺页异常
  2. 陷阱(trap)：eip=随后要执行的指令的地址。
  3. 异常中止(abort)：eip=？？？
     - 发生严重的错误。eip值无效，只有强制终止受影响的进程

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
     1. 把此信号转换成对应的中断向量
     2. 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
     3. 把引发信号发送到处理器的INTR引脚，即在CPU上产生一个中断
     4. 等待，直到CPU应答此信号；收到应答后，清INTR引脚
  3. 返回第一步

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
  - IR0，IR1，… ，IR7
- 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
- 注意：
  1. 屏蔽 $\neq$ 丢失：
     - 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
  2. 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

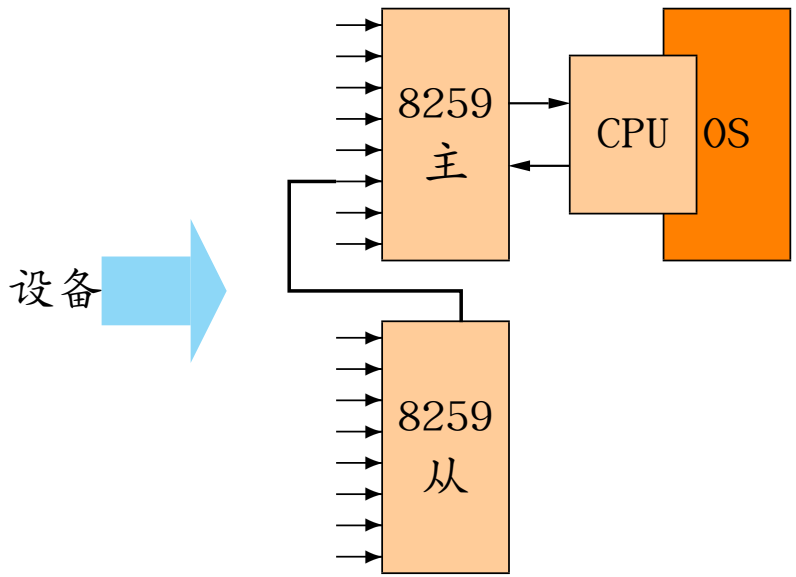
- CPU可以屏蔽所有的可屏蔽中断
  - Eflags中的IF标志：
    0=关中断；
    1=开中断。
  - 关中断时，CPU不响应中断控制器发布的任何中断请求
  - 内核中使用cli和sti指令分别清除和设置该标志

关中断和开中断，参见include/asm-x86/irqflags.h

```
static inline void native_irq_disable(void) {
    asm volatile(" cli" : : :" memory" );
}
static inline void native_irq_enable(void) {
    asm volatile(" sti" : : :" memory" );
}
```

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：
  - 从片→主片的IRQ2引脚



- 因此，一共可以处理最多15个不同的IRQ线

传统的中断控制器：8259A

- 8259A：设置起始中断向量号
  - 参见arch/x86/kernel/i8259_32.c::init_8259A()

```
void init_8259A(int auto_eoi) {
    ...
    /*
     * outb_pic - this has to work on a wide range of PC hardware.
     */
    outb_pic(0x11, PIC_MASTER_CMD); /* ICW1: select 8259A-1 init */
    outb_pic(0x20 + 0, PIC_MASTER_IMR); /* ICW2: 8259A-1 IR0-7 mapped to 0x20-0x27 */
    outb_pic(1U << PIC_CASCADE_IR, PIC_MASTER_IMR); /* 8259A-1 (the master) has a
slave on IR2 */
    if (auto_eoi) /* master does Auto EOI */
        outb_pic(MASTER_ICW4_DEFAULT | PIC_ICW4_AEOI, PIC_MASTER_IMR);
    else /* master expects normal EOI */
        outb_pic(MASTER_ICW4_DEFAULT, PIC_MASTER_IMR);

    outb_pic(0x11, PIC_SLAVE_CMD); /* ICW1: select 8259A-2 init */
    outb_pic(0x20 + 8, PIC_SLAVE_IMR); /* ICW2: 8259A-2 IR0-7 mapped to 0x28-0x2f */
    outb_pic(PIC_CASCADE_IR, PIC_SLAVE_IMR); /* 8259A-2 is a slave on master's IR2 */
    outb_pic(SLAVE_ICW4_DEFAULT, PIC_SLAVE_IMR); /*
    ...
}
```

传统的中断控制器：8259A

- 8259A：禁止/激活某个IRQ线，参见arch/x86/kernel/i8259_32.c

```
void disable_8259A_irq(unsigned int irq) {
    unsigned int mask = 1 << irq;
    unsigned long flags;
    spin_lock_irqsave(&i8259A_lock, flags);
    cached_irq_mask |= mask;
    if (irq & 8)
        outb(cached_slave_mask, PIC_SLAVE_IMR);
    else
        outb(cached_master_mask, PIC_MASTER_IMR);
    spin_unlock_irqrestore(&i8259A_lock, flags);
}
```

```
...
unsigned int cached_irq_mask = 0xffff;
```

include/asm-x86/i8259.h中：

```
#define __byte(x, y) (((unsigned char *)&(y))[x])
#define cached_master_mask (__byte(0, cached_irq_mask))
#define cached_slave_mask (__byte(1, cached_irq_mask))
```

传统的中断控制器：8259A

- 8259A：禁止/激活某个IRQ线，参见arch/x86/kernel/i8259_32.c

```
void enable_8259A_irq(unsigned int irq) {
    unsigned int mask = ~(1 << irq);
    unsigned long flags;
    spin_lock_irqsave(&i8259A_lock, flags);
    cached_irq_mask &= mask;
    if (irq & 8)
        outb(cached_slave_mask, PIC_SLAVE_IMR);
    else
        outb(cached_master_mask, PIC_MASTER_IMR);
    spin_unlock_irqrestore(&i8259A_lock, flags);
}
```

```
...
unsigned int cached_irq_mask = 0xffff;
```

```
include/asm-x86/i8259.h中：
#define __byte(x, y) (((unsigned char *)&(y))[x])
#define cached_master_mask (__byte(0, cached_irq_mask))
#define cached_slave_mask (__byte(1, cached_irq_mask))
```

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	type	Exception handler	Signal
0	Divide error	fault	divide_error()	SIGFPE
1	Debug	fault/trap	debug()	SIGTRAP
2	NMI	interrupt	nmi()	None
3	Breakpoint	trap	int3()	SIGTRAP
4	Overflow	trap	overflow()	SIGSEGV
5	Bounds check	fault	bounds()	SIGSEGV
6	Invalid opcode	fault	invalid_op()	SIGILL
7	Device not available	fault	device_not_available()	SIGSEGV
8	Double fault	Abort	double_fault()	SIGSEGV
9	Coprocessor segment overrun	fault	coprocessor_segment_overrun()	SIGFPE

异常

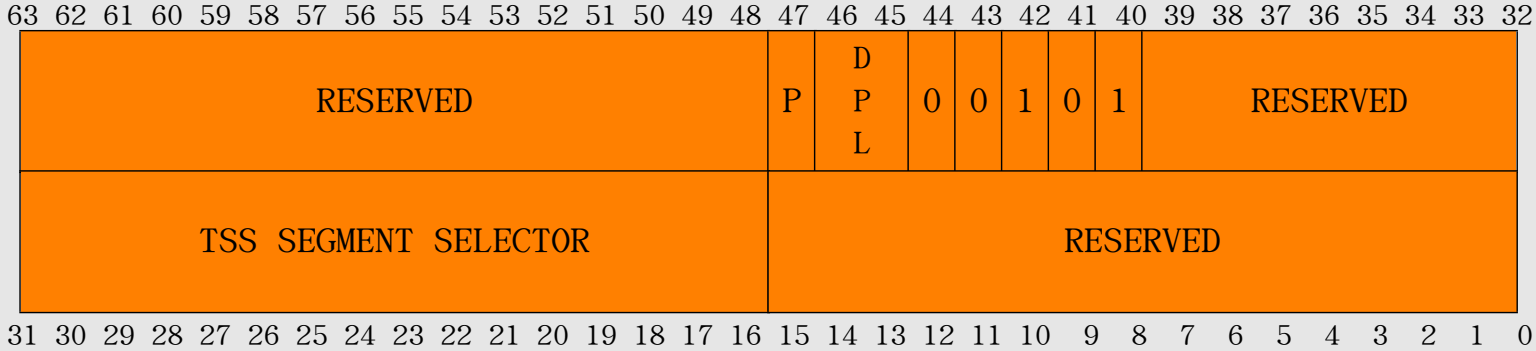
#	Exception	type	Exception handler	Signal
10	Invalid TSS	fault	invalid_tss()	SIGSEGV
11	Segment not present	fault	segment_not_present()	SIGBUS
12	Stack exception	fault	stack_segment()	SIGBUS
13	General protection	fault	general_protection()	SIGSEGV
14	Page Fault	fault	page_fault()	SIGSEGV
15	Intel reserved	-	None	None
16	Floating-pointer error	fault	coprocessor_error	SIGFPE
17	Alignment check	fault	alignment_check()	SIGBUS
18	Machine check	Abort	machine_check()	None
19	SIMD floating point	fault	simd_coprocessor_error()	SIGFPE

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

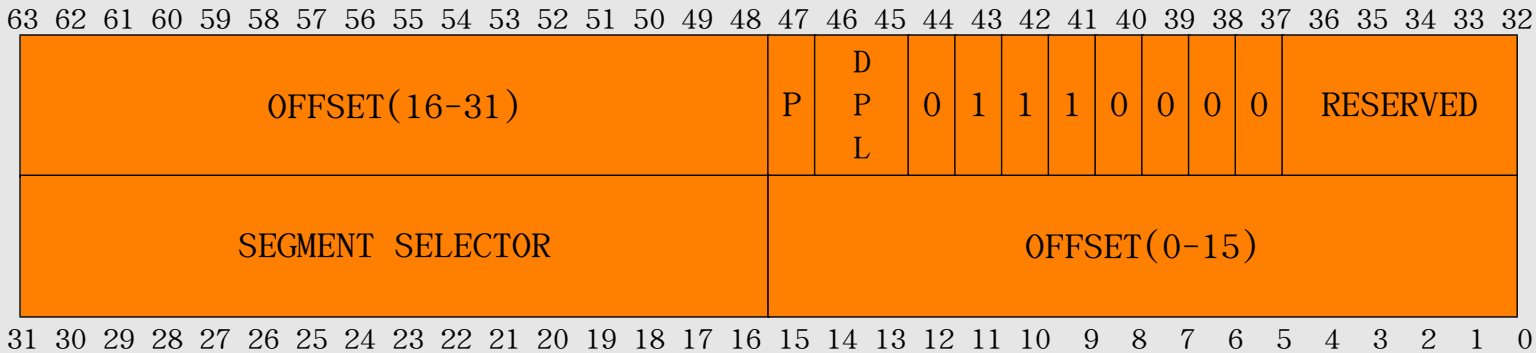
- 中断描述符表是一个系统表，它与每一个中断或者异常向量相联系
  - 每个向量在表中有相应的中断或者异常处理程序的入口地址。
  - 每个描述符8个字节，共256项，占用空间2KB
  - 内核在允许中断发生前，必须先对IDT表进行适当的初始化
- CPU的idtr寄存器指向IDT表的物理基地址
  - lidt指令
- IDT包含3种类型的描述符：任务门、中断门和陷阱门
  - 陷阱门与中断门类似，但进入陷阱门时，系统不会进入关中断状态

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

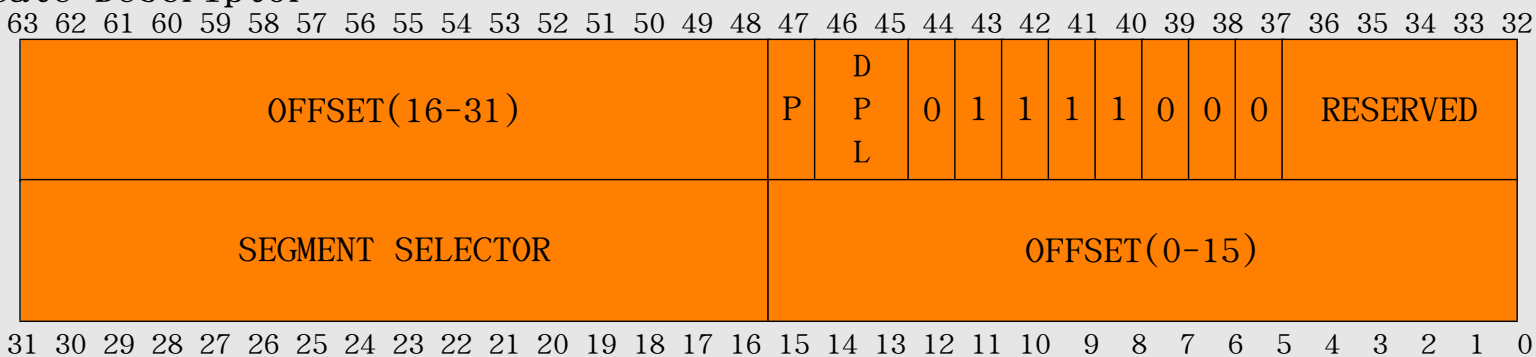
Task Gate Descriptor



Interrupt Gate Descriptor



Trap Gate Descriptor



Outline

Outline

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控制单元中断当前任务的运行转而进入中断处理
- 硬件进入中断处理的过程如下：

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

中断和异常的硬件处理：进入中断/异常

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
     （对应GDT表项中的DPL vs CS寄存器中的CPL）
     （注：中断例程只能由内核提供，用户进程运行时可以发生中断）
   - 编程异常：还需比较CPL与对应IDT表项中的DPL

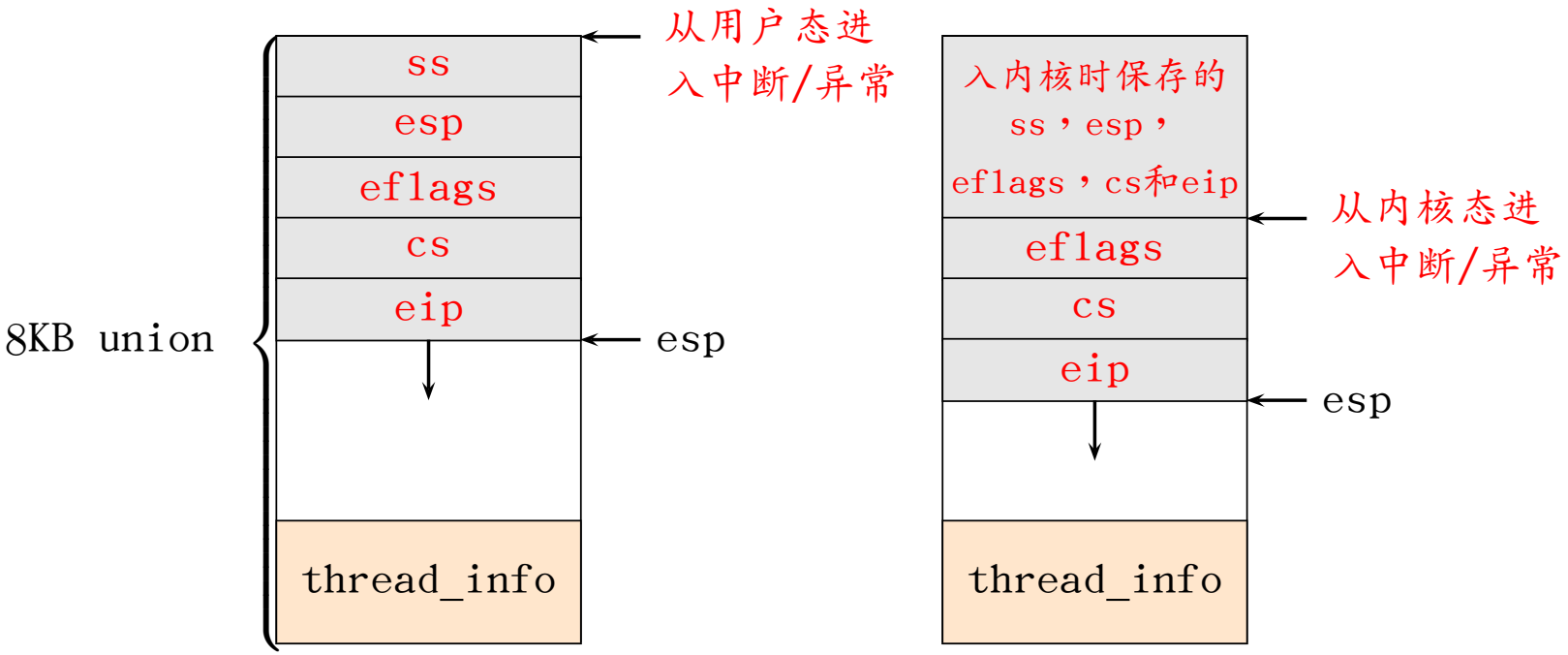
5. 检查是否发生了特权级的变化，一般指是否由用户态陷入了内核态。
   若是，控制单元必须开始使用与新的特权级相关的堆栈a，即内核态栈
   1. 读tr寄存器，访问运行进程的tss段
   2. 用与新特权级相关的栈段和栈指针装载ss和esp寄存器。
      这些值可以在进程的tss段中找到
   3. 在新的栈中保存ss和esp以前的值，这些值指明了与旧特权级相关的栈的逻辑地址

中断和异常的硬件处理：进入中断/异常

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

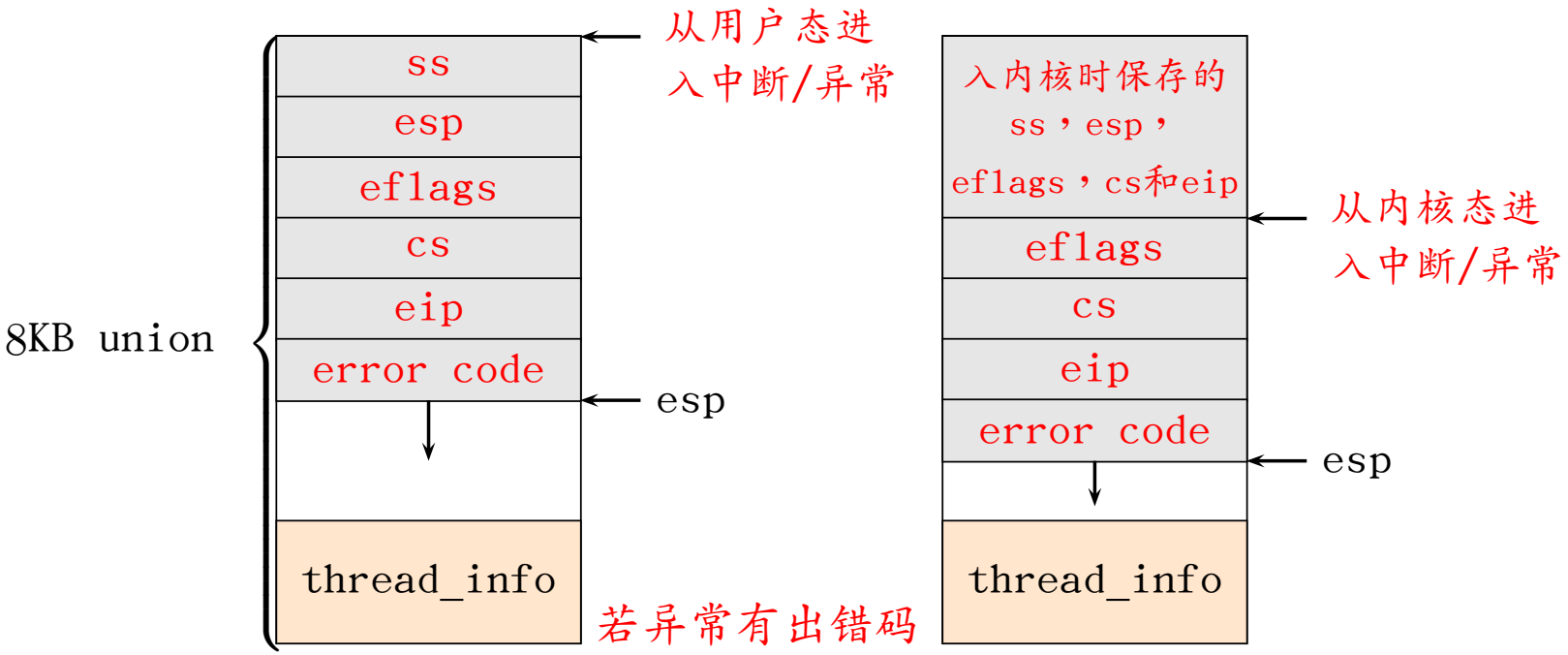
中断和异常的硬件处理：进入中断/异常

- 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)



中断和异常的硬件处理：进入中断/异常

- 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)



Outline

中断和异常的硬件处理：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

Outline

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

Outline

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
- Linux在启动和初始化的不同阶段，均有对IDT表的初始化和加载。
  1. 进入保护模式前IDT表的初始化
  2. idt_table的初步初始化
  3. Start_kernel中idt_table的初始化

1、进入保护模式前IDT表的初始化

- arch/x86/boot/pm.c::go_to_protected_mode中调用setup_idt来初始化一个临时的IDT表

```
struct gdt_ptr {
    u16 len;
    u32 ptr;
} __attribute__((packed));
...
static void setup_idt(void) {
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

- 此时，IDT表的起始地址填充为0，其长度填充为0

2、idt_table的初步初始化

- idt_table的定义，参见arch/x86/kernel/traps_32.c

```
gate_desc idt_table[256]
        __attribute__((__section__(" .data.idt" ))) = { { { { 0, 0 } } }, };
```

其中，门描述符gate_desc在include/asm-x86/desc_defs.h中定义：

```
struct desc_struct {
    union {
        struct {
            unsigned int a;
            unsigned int b;
        };
        struct {
            u16 limit0;
            u16 base0;
            unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
            unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
        };
    };
} __attribute__((packed));
...
typedef struct desc_struct gate_desc;
```

2、idt_table的初步初始化

- idt_table的初步初始化为：
  1. 用ignore_int()函数填充256个idt_table表项，参见arch/x86/kernel/head_32.S::setup_idt
  2. 使用early_divide_err、early_illegal_opcode、early_protection_fault和early_page_fault初始化对应的异常向量

setup_idt

```
/*
 * setup_idt
 *
 * sets up a idt with 256 entries pointing to
 * ignore_int, interrupt gates. It doesn't actually load
 * idt - that can be done only after paging has been enabled
 * and the kernel moved to PAGE_OFFSET. Interrupts
 * are enabled elsewhere, when we can be relatively
 * sure everything is ok.
 *
 * Warning: %esi is live across this function.
 */
```

2、idt_table的初步初始化

```
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
```

2、idt_table的初步初始化

ignore_int

```
/* This is the default interrupt "handler" :-) */
    ALIGN
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
```

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

2、idt_table的初步初始化

```
early_page_fault:
    movl $14,%edx
    jmp early_fault

early_fault:
    cld
#ifdef CONFIG_PRINTK
    pusha
    movl $(__KERNEL_DS),%eax
    movl %eax,%ds
    movl %eax,%es
    cmpl $2,early_recursion_flag
    je hlt_loop
    incl early_recursion_flag
    movl %cr2,%eax
    pushl %eax
    pushl %edx /* trapno */
    pushl $fault_msg
#ifdef CONFIG_EARLY_PRINTK
    call early_printk
#else
    call printk
```

- 以early_page_fault为例
  - 该例程输出异常发生时的关键硬件上下文

```
#endif
#endif
    call dump_stack
hlt_loop:
    hlt jmp hlt_loop
```

```
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"
```

3、Start_kernel中idt_table的初始化

- 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。例如int3
  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
- 然而用户进程有时必须要能发出一个编程异常，例如int 0x80。
  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
- 由此，Linux在x86的中断门/陷阱门的基础上实现了Linux的中断门、陷阱门和系统门

3、Start_kernel中idt_table的初始化

1. Linux的中断门
   - 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
2. Linux的系统门
   1. 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
3. Linux的陷阱门
   1. 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

3、Start_kernel中idt_table的初始化

- 下列体系结构相关的函数用来在IDT中设置门，参见include/asm-x86/desc.h

```
static inline void set_intr_gate(unsigned int n, void *addr) { \
    BUG_ON((unsigned)n > 0xFF);
    _set_gate(n, GATE_INTERRUPT, addr, 0, 0, __KERNEL_CS);
}
...
static inline void set_system_intr_gate(unsigned int n, void *addr) {
    BUG_ON((unsigned)n > 0xFF);
    _set_gate(n, GATE_INTERRUPT, addr, 0x3, 0, __KERNEL_CS);
}
static inline void set_trap_gate(unsigned int n, void *addr) {
    BUG_ON((unsigned)n > 0xFF);
    _set_gate(n, GATE_TRAP, addr, 0, 0, __KERNEL_CS);
}
static inline void set_system_gate(unsigned int n, void *addr) {
    " ,
    _set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
    ...
}
```

3、Start_kernel中idt_table的初始化

- 下列体系结构相关的函数用来在IDT中设置门，参见include/asm-x86/desc.h

```
static inline void pack_gate(gate_desc *gate, unsigned char type,
            unsigned long base, unsigned dpl, unsigned flags,
            unsigned short seg) {
    gate->a = (seg << 16) | (base & 0xffff);
    gate->b = (base & 0xffff0000) | (((0x80 | type | (dpl << 5)) & 0xff) << 8);
}
...
static inline void _set_gate(int gate, unsigned type, void *addr,
            unsigned dpl, unsigned ist, unsigned seg) {
    gate_desc s;
    pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
    ...
    write_idt_entry(idt_table, gate, &s);
}
```

3、Start_kernel中idt_table的初始化

- Start_kernel中的IDT表初始化：trap_init，init_IRQ

```
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); /* int3/4 can be called from all */
    set_system_gate(4, &overflow);
    set_trap_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);
    set_trap_gate(12, &stack_segment);
    set_trap_gate(13, &general_protection);
    set_intr_gate(14, &page_fault);
    ...
    set_trap_gate(19, &simd_coprocessor_error);
    set_system_gate(SYSCALL_VECTOR, &system_call);
    ...
}
```

3、Start_kernel中idt_table的初始化

- Start_kernel中的IDT表初始化：trap_init，init_IRQ

init_IRQ() 即native_init_IRQ()

```
void __init native_init_IRQ(void) {
    ...
    /* all the set up before the call gates are initialised */
    pre_intr_init_hook();
    ...
    for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (i >= NR_IRQS)
            break;
        /* SYSCALL_VECTOR was reserved in trap_init. */
        if (!test_bit(vector, used_vectors))
            set_intr_gate(vector, interrupt[i]);
    }
    /* setup after call gates are initialised (usually add in
     * the architecture specific gates)
     */
    intr_init_hook();
    ...
}
```

Outline

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
- 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

异常处理

- 以14号缺页异常的处理为例：
  - 在arch/x86/kernel/traps_32.c::trap_init中，14号异常的处理入口函数被设置为page_fault

```
set_intr_gate(14, &page_fault);
```

  - page_fault的原型在arch/x86/kernel/traps_32.c中被申明为：

```
asmlinkage void page_fault(void);
```

  - page_fault实际上是在arch/x86/kernel/entry_32.S中以汇编码的形式定义的

```
KPROBE_ENTRY(page_fault)
    ...
    pushl $do_page_fault
    ...
error_code:
```

异常处理

- 以14号缺页异常的处理为例：
  - 在arch/x86/kernel/traps_32.c::trap_init中，14号异常的处理入口函数被设置为page_fault

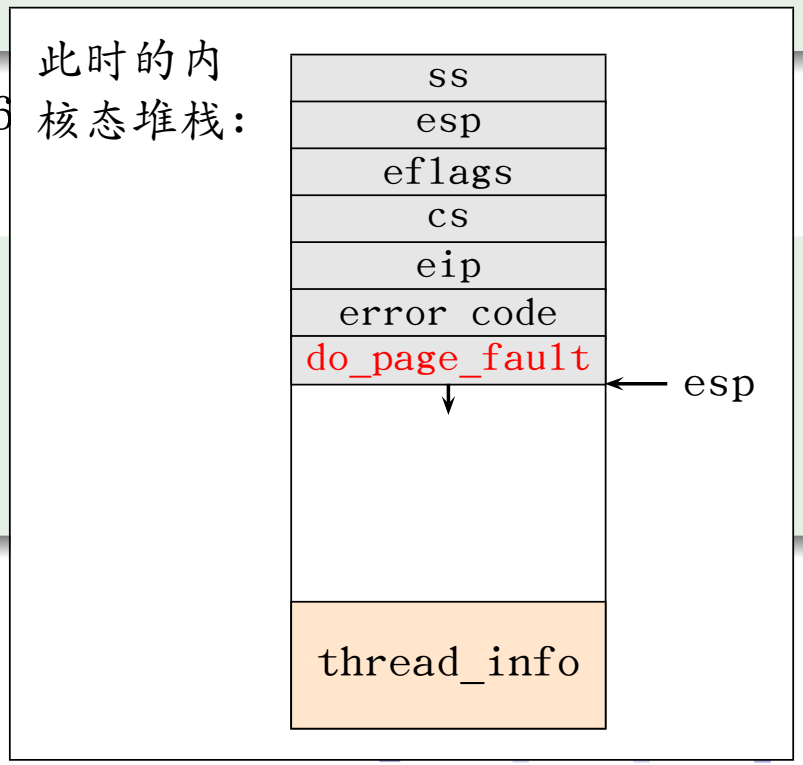
```
set_intr_gate(14, &page_fault);
```

  - page_fault的原型在arch/x86/kernel/traps_32.c中被申明为：

```
asmlinkage void page_fault(void);
```

  - page_fault实际上是在arch/x86以汇编码的形式定义的

```
KPROBE_ENTRY(page_fault)
    ...
    pushl $do_page_fault
    ...
error_code:
```



异常处理

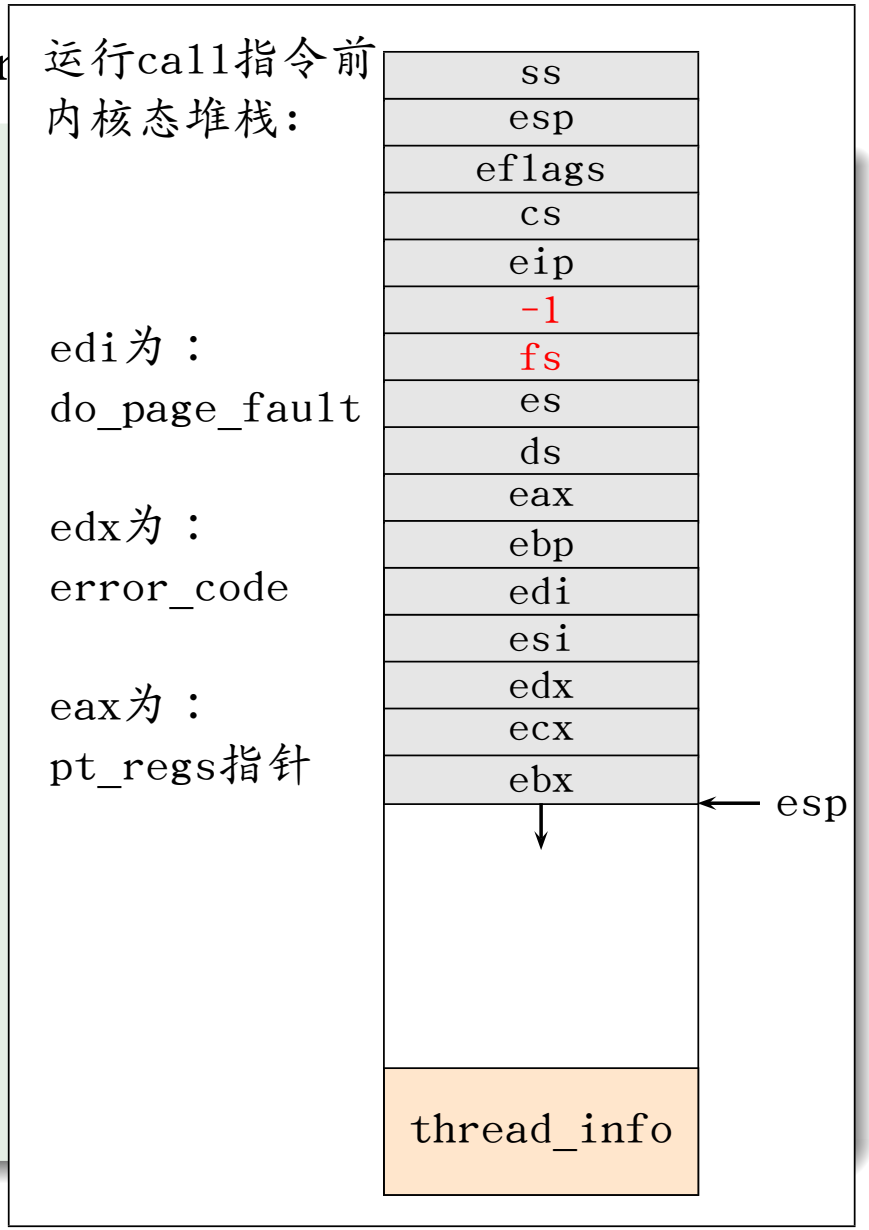
- 阅读error_code，参见arch/x86/kernel/entry_32.S

```
error_code:
/* the function address is in %fs's slot on the stack */
    pushl %es
    pushl %ds
    pushl %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    cld
    pushl %fs
    movl $(__KERNEL_PERCPU), %ecx
    movl %ecx, %fs
    popl %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)
    movl $(__USER_DS), %ecx
    movl %ecx, %ds
    movl %ecx, %es
    movl %esp,%eax # pt_regs pointer
    call *%edi
    jmp ret_from_exception
```

异常处理

- 阅读error_code，参见arch/x86/ker

```
error_code:
/* the function address is in %fs's slot on the stack */
    pushl %es
    pushl %ds
    pushl %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    cld
    pushl %fs
    movl $(__KERNEL_PERCPU), %ecx
    movl %ecx, %fs
    popl %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)
    movl $(__USER_DS), %ecx
    movl %ecx, %ds
    movl %ecx, %es
    movl %esp,%eax # pt_regs pointer
    call *%edi
    jmp ret_from_exception
```

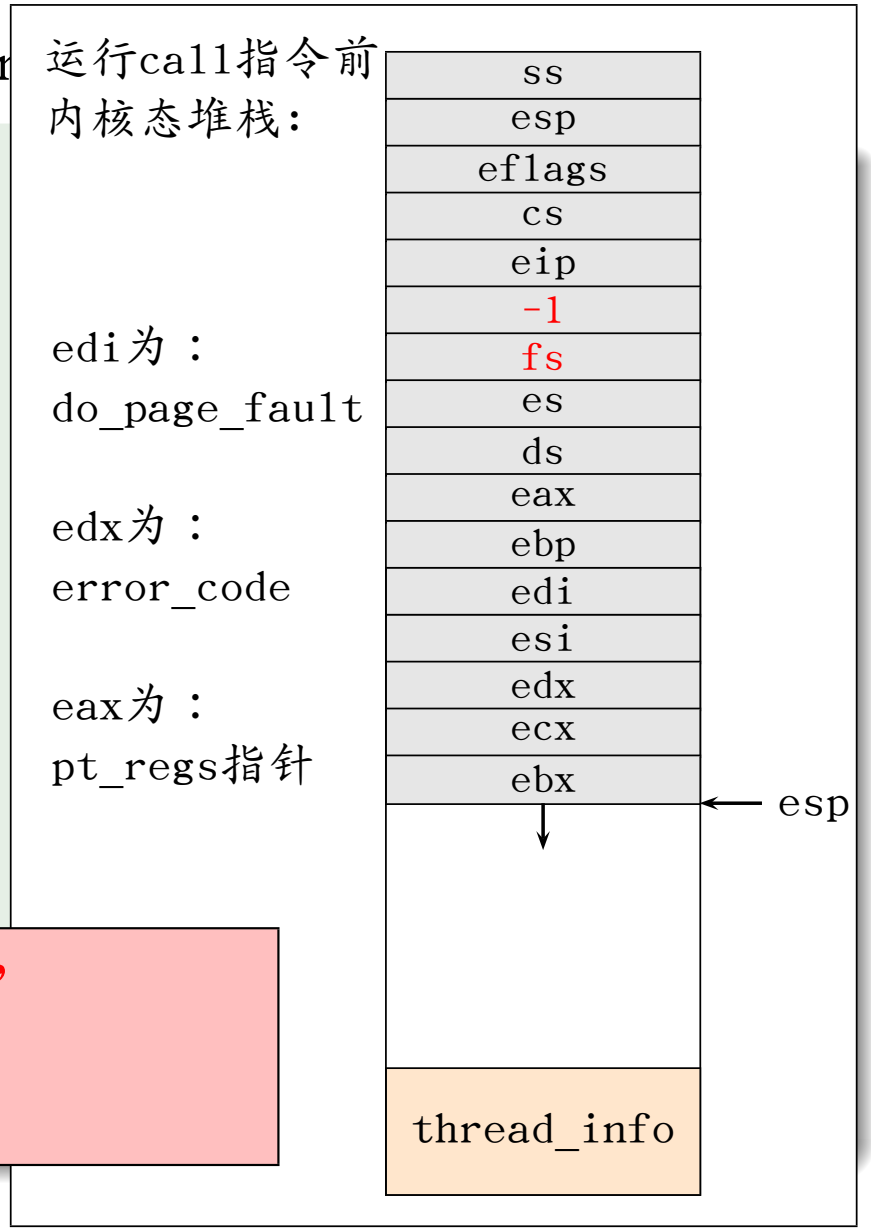


异常处理

- 阅读error_code，参见arch/x86/ker

```
error_code:
/* the function address is in %fs's slot on the stack */
    pushl %es
    pushl %ds
    pushl %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    cld
    pushl %fs
    movl $(__KERNEL_PERCPU), %ecx
    movl %ecx, %fs
    popl %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)
    movl $
    movl %
    movl %
    movl %
    call *
    jmp re
```

do_page_fault为寄存器传参，
它的第一个参数取自eax，
第二个参数取自edx



异常处理

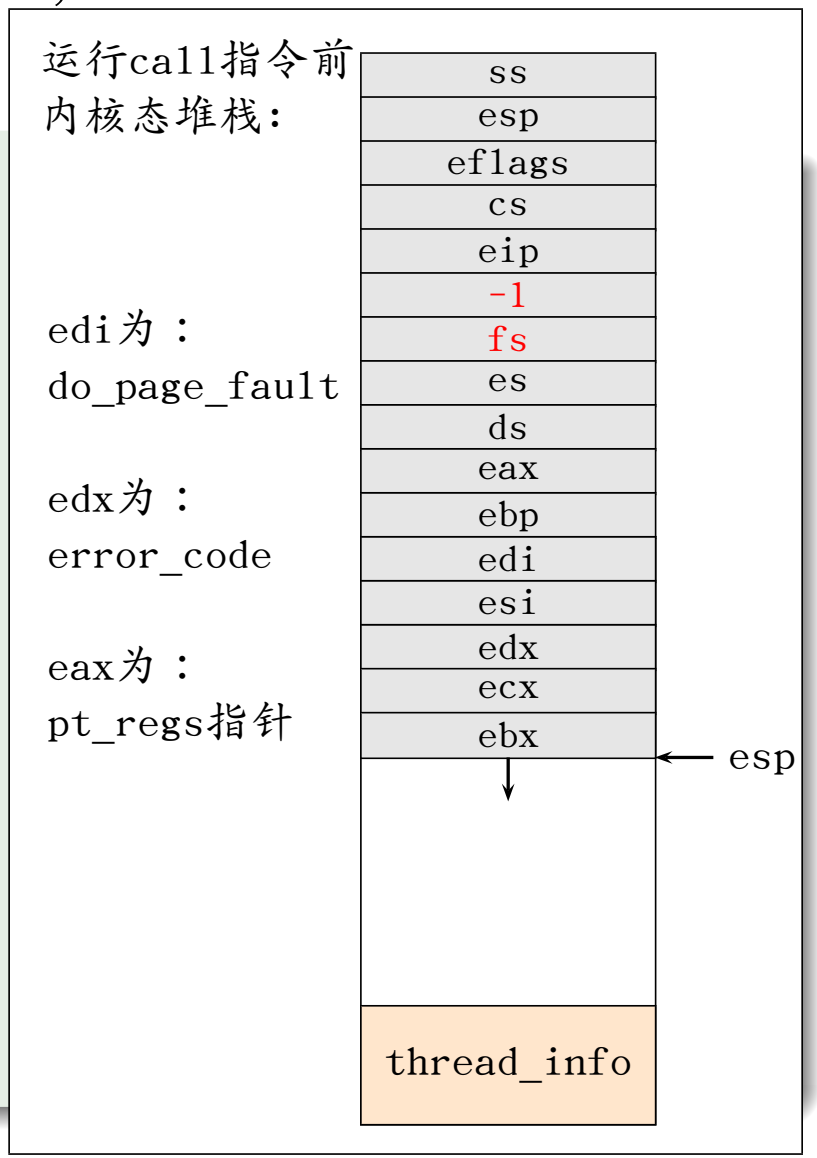
- do_page_fault在arch/x86/mm/fault.c中定义

```
void __kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code) {
...
}
```

异常处理

- pt_regs结构(恢复现场所需的上下文)，
  参见include/asm-x86/ptrace.h

```
struct pt_regs {
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long bp;
    unsigned long ax;
    unsigned long ds;
    unsigned long es;
    unsigned long fs; /* int gs; */
    unsigned long orig_ax;
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};
```



异常处理

- 当C函数终止时，根据内核栈中的返回地址， 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

低级异常处理小结

- 低级中断处理过程可以归纳为：
  1. 保存上下文
  2. 调用异常相关的C语言函数，如do_page_fault
  3. 恢复上下文，从异常返回

Outline

中断的类型

- 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上才有)

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8259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
  1. 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
  2. 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

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

设备中断

IRQ	INT	Hardware Device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	Frist 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

Linux的中断处理原则

- 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

1. 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

Linux的中断处理原则

2. 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

3. 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

Linux的中断处理原则

- 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

低级中断处理

- 系统初始化时，调用arch/x86/kernel/i8259_32.c::init_IRQ()用新的中断门替换临时中断门来更新IDT

```
/* Overridden in paravirt.c */
void init_IRQ(void) __attribute__((weak, alias(" native_init_IRQ" )));

void __init native_init_IRQ(void) {
    ...
    for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (i >= NR_IRQS)
            break;
        /* SYSCALL_VECTOR was reserved in trap_init. */
        if (!test_bit(vector, used_vectors))
            set_intr_gate(vector, interrupt[i]);
    }
    ...
}
```

  - 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。

低级中断处理

- Interrupt数组，在include/asm-x86/hw_irq_32.h被声明为

  extern void (*const interrupt[NR_IRQS]) (void);

  但其定义比较隐晦，参见arch/x86/kernel/entry_32.S

```
.section .rodata," a"
ENTRY(interrupt)
.text
ENTRY(irq_entries_start)
vector=0
.rept NR_IRQS
1:      pushl $~(vector)
        jmp common_interrupt
.previous
        .long 1b
.text
vector=vector+1
.endr
END(irq_entries_start)
.previous
END(interrupt)
.previous
```

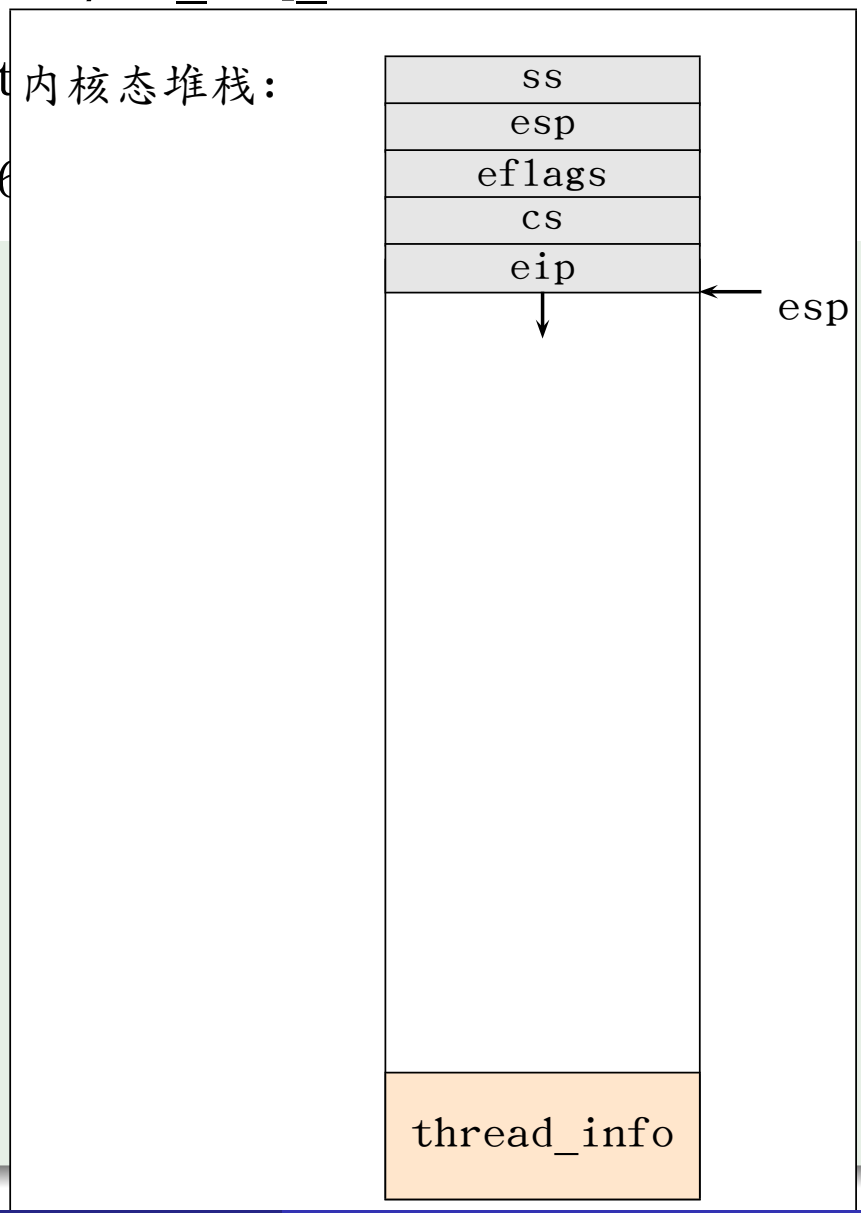
低级中断处理

- Interrupt数组，在include/asm-x86/hw_irq_32.h被声明为

  extern void (*const int

  但其定义比较隐晦，参见arch/x86

```
.section .rodata," a"
ENTRY(interrupt)
.text
ENTRY(irq_entries_start)
vector=0
.rept NR_IRQS
1:      pushl $~(vector)
        jmp common_interrupt
.previous
        .long 1b
.text
vector=vector+1
.endr
END(irq_entries_start)
.previous
END(interrupt)
.previous
```



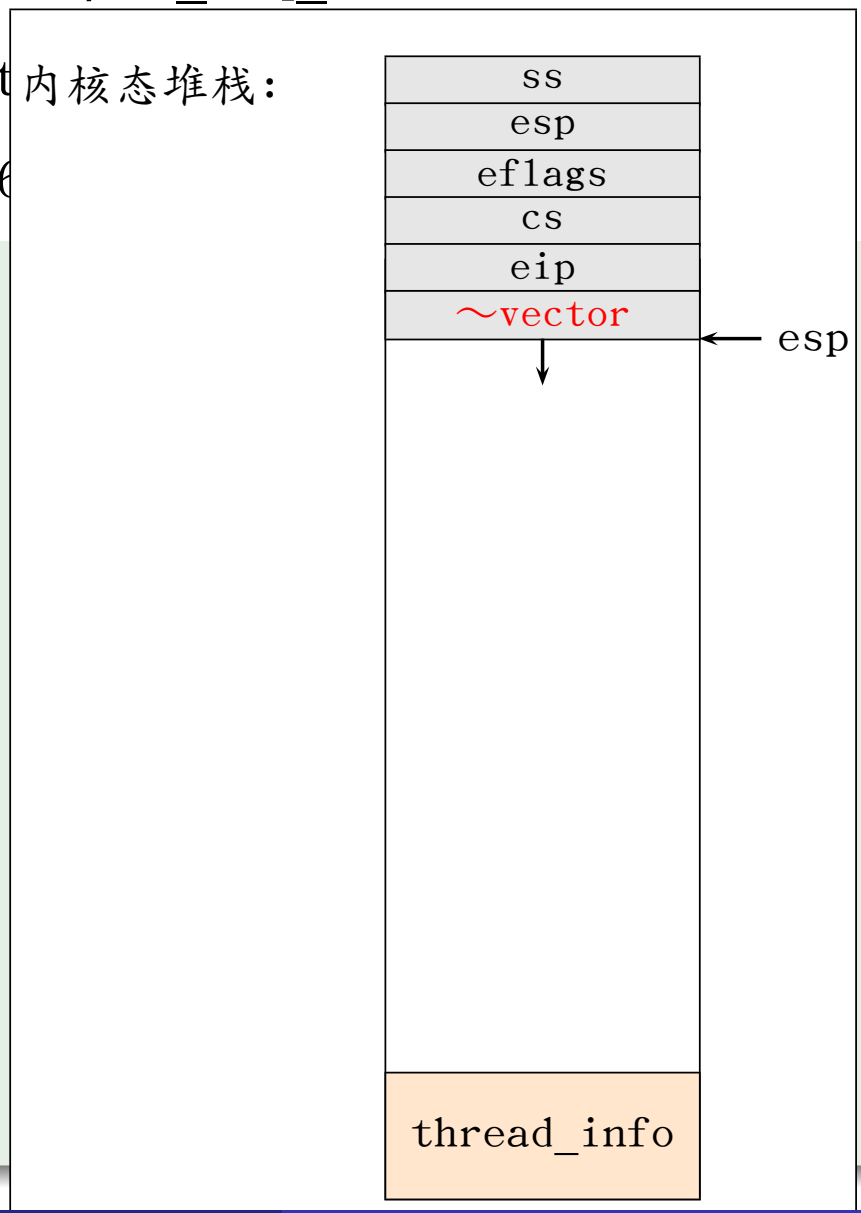
低级中断处理

- Interrupt数组，在include/asm-x86/hw_irq_32.h被声明为

  extern void (*const int

  但其定义比较隐晦，参见arch/x86

```
.section .rodata," a"
ENTRY(interrupt)
.text
ENTRY(irq_entries_start)
vector=0
.rept NR_IRQS
1:      pushl $~(vector)
        jmp common_interrupt
.previous
        .long 1b
.text
vector=vector+1
.endr
END(irq_entries_start)
.previous
END(interrupt)
.previous
```



低级中断处理

- arch/x86/kernel/entry_32.S::common_interrupt为

```
/*
 * the CPU automatically disables interrupts when executing an IRQ vector,
 * 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
```

低级中断处理

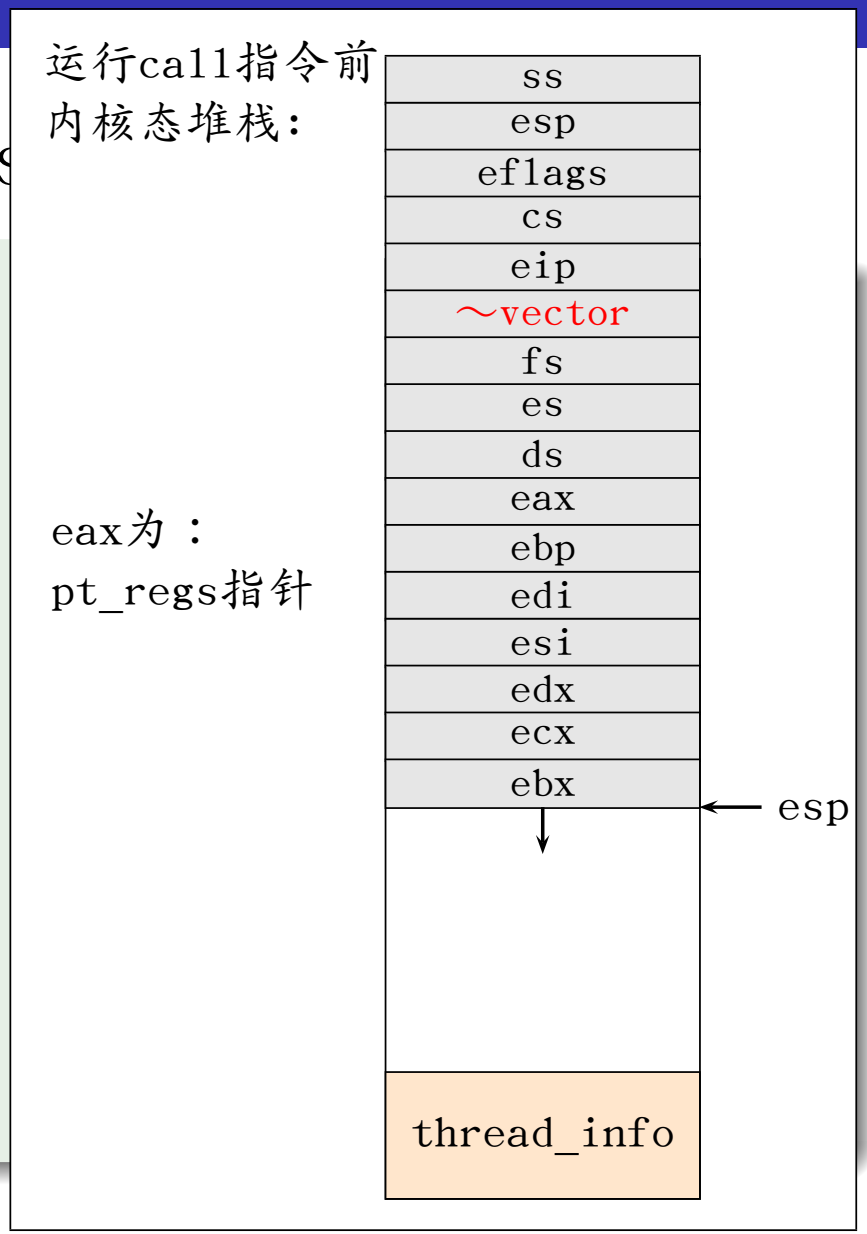
- 其中，arch/x86/kernel/entry_32.S::SAVE_ALL，精简后如下：

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

低级中断处理

- 其中，arch/x86/kernel/entry_32.S

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



低级中断处理

- do_IRQ在arch/x86/kernel/irq_32.c中定义，
  它使用寄存器eax传参
  （eax在common_interrupt中指向栈顶的pt_regs结构）

```
unsigned int do_IRQ(struct pt_regs *regs) {...}
```

- 小结，每个中断程序入口操作（即低级中断处理）为：
  1. 将中断向量信息入栈
  2. 保存所有其他寄存器
  3. 调用do_IRQ
  4. 跳转到ret_from_intr

Outline

中断描述符表irq_desc[]

- do_IRQ在arch/x86/kernel/irq_32.c中提供，它使用与体系结构无关的中断描述符表irq_desc[]：

```
unsigned int do_IRQ(struct pt_regs *regs) {
    struct pt_regs *old_regs;
    /* high bit used in ret_from_ code */
    int irq = ~regs->orig_ax;
    struct irq_desc *desc = irq_desc + irq;
    ...
```

1. 从寄存器上下文中获得irq号
2. 从Linux中断描述符中找到irq号对应的中断描述符desc

- Linux中断描述符表：irq_desc数组
  包含了NR_IRQS(通常为224=256-32)个irq_desc_t描述符。
  在include/linux/irq.h中声明：

```
extern struct irq_desc irq_desc[NR_IRQS];
```

中断描述符表irq_desc[]

- 中断描述符数据结构irq_desc在include/linux/irq.h中定义

```
/**
 * struct irq_desc - interrupt descriptor
 *
 * @handle_irq: highlevel irq-events handler [if NULL, __do_IRQ()]
 * @chip: low level interrupt hardware access
 * @msi_desc: MSI descriptor
 * @handler_data: per-IRQ data for the irq_chip methods
 * @chip_data: platform-specific per-chip private data for the chip
 * methods, to allow shared chip implementations
 * @action: the irq action chain
 * @status: status information
 * @depth: disable-depth, for nested irq_disable() calls
 * @wake_depth: enable depth, for multiple set_irq_wake() callers
 * @irq_count: stats field to detect stalled irqs
 * @irqs_unhandled: stats field for spurious unhandled interrupts
 * @last_unhandled: aging timer for unhandled count
 * @lock: locking for SMP
 * @affinity: IRQ affinity on SMP
 * @cpu: cpu index useful for balancing
 * @pending_mask: pending rebalanced interrupts
 * @dir: /proc/irq/ procfs entry
 * @affinity_entry: /proc/irq/smp_affinity procfs entry on SMP
 * @name: flow handler name for /proc/interrupts output
 */
```

中断描述符表irq_desc[]

- 中断描述符数据结构irq_desc在include/linux/irq.h中定义

```
struct irq_desc {
    irq_flow_handler_t handle_irq;
    struct irq_chip *chip;
    struct msi_desc *msi_desc;
    void *handler_data;
    void *chip_data;
    struct irqaction *action; /* IRQ action list */
    unsigned int status; /* IRQ status */
    unsigned int depth; /* nested irq disables */
    unsigned int wake_depth; /* nested wake enables */
    unsigned int irq_count; /* For detecting broken IRQs */
    unsigned int irqs_unhandled;
    unsigned long last_unhandled; /* Aging timer for unhandled count */
    spinlock_t lock;
    ...
    const char *name;
} ____cacheline_internodealigned_in_smp;
```

中断描述符表irq_desc[]

- 中断描述符表irq_desc[]的定义和最初的初始化，参见kernel/irq/handle.c

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .status = IRQ_DISABLED,
        .chip = &no_irq_chip,
        .handle_irq = handle_bad_irq,
        .depth = 1,
        .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
#ifdef CONFIG_SMP
        .affinity = CPU_MASK_ALL
#endif
    }
};
```

- Linux中断描述符表irq_desc[]的几个关键数据结构和类型
  1. irqaction数据结构
  2. irq_chip
  3. irq_flow_handler_t

irq_action数据结构

- irqaction数据结构，参见include/linux/interrupt.h
  - 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

```
struct irqaction {
    irq_handler_t handler;
    unsigned long flags;
    cpumask_t mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
    int irq;
    struct proc_dir_entry *dir;
};
```

  - 设置irqaction的函数：kernel/irq/manage.c::setup_irq

irq_action数据结构

- 以时钟中断为例，在时钟中断初始化时注册时钟的irqaction（参见arch/x86/mach-default/setup.c）

```
static struct irqaction irq0 = {
    .handler = timer_interrupt,
    .flags = IRQF_DISABLED | IRQF_NOBALANCING | IRQF_IRQPOLL,
    .mask = CPU_MASK_NONE,
    .name = " timer"
};
/**
 * time_init_hook - do any specific initialisations for the system timer.
 *
 * Description:
 *     Must plug the system timer interrupt source at HZ into the IRQ listed
 *     in irq_vectors.h:TIMER_IRQ
 **/
void __init time_init_hook(void) {
    irq0.mask = cpumask_of_cpu(0);
    setup_irq(0, &irq0);
}
```

irq_chip数据结构

- 为特定PIC编写的低级I/O例程，参见include/linux/irq.h

```
/**
 * struct irq_chip - hardware interrupt chip descriptor
 *
 * @name: name for /proc/interrupts
 * @startup: start up the interrupt (defaults to ->enable if NULL)
 * @shutdown: shut down the interrupt (defaults to ->disable if NULL)
 * @enable: enable the interrupt (defaults to chip->unmask if NULL)
 * @disable: disable the interrupt (defaults to chip->mask if NULL)
 * @ack: start of a new interrupt
 * @mask: mask an interrupt source
 * @mask_ack: ack and mask an interrupt source
 * @unmask: unmask an interrupt source
 * @eoi: end of interrupt - chip level
 * @end: end of interrupt - flow level
 * @set_affinity: set the CPU affinity on SMP machines
 * @retrigger: resend an IRQ to the CPU
 * @set_type: set the flow type (IRQ_TYPE_LEVEL/etc.) of an IRQ
 * @set_wake: enable/disable power-management wake-on of an IRQ
 *
 * @release: release function solely used by UML
 * @typename: obsoleted by name, kept as migration helper
 */
```

irq_chip数据结构

- 为特定PIC编写的低级I/O例程，参见include/linux/irq.h

```
struct irq_chip {
      const char *name;
      unsigned int (*startup)(unsigned int irq);
      void (*shutdown)(unsigned int irq);
      void (*enable)(unsigned int irq);
      void (*disable)(unsigned int irq);
      void (*ack)(unsigned int irq);
      void (*mask)(unsigned int irq);
      void (*mask_ack)(unsigned int irq);
      void (*unmask)(unsigned int irq);
      void (*eoi)(unsigned int irq);
      void (*end)(unsigned int irq);
      void (*set_affinity)(unsigned int irq, cpumask_t dest);
      int (*retrigger)(unsigned int irq);
      int (*set_type)(unsigned int irq, unsigned int flow_type);
      int (*set_wake)(unsigned int irq, unsigned int on);
      ...
};
```

irq_chip数据结构

- 例如8259的irq_chip，参见arch/x86/kernel/i8259_32.c

```
static struct irq_chip i8259A_chip = {
    .name = "XT-PIC",
    .mask = disable_8259A_irq,
    .disable = disable_8259A_irq,
    .unmask = enable_8259A_irq,
    .mask_ack = mask_and_ack_8259A,
};
```

irq_chip数据结构

- 为一个中断设置irq_chip的函数有
  - set_irq_chip_and_handler_name
    - set_irq_chip
    - 等
- 例如：在init_IRQ（即native_init_IRQ）中，调用的pre_intr_init_hook可能如下定义（arch/x86/mach-default/setup.c）

```
void __init pre_intr_init_hook(void) {
    init_ISA_irqs();
}
```

  - 其中，init_ISA_irqs在arch/x86/kernel/i8259_32.c中定义

```
void __init init_ISA_irqs (void) {
    ...
    for (i = 0; i < 16; i++) {
        set_irq_chip_and_handler_name(i, &i8259A_chip, handle_level_irq, "XT");
    }
}
```

irq_chip数据结构

- 又如：arch/x86/kernel/i8259_32.c::make_8259A_irq

```
void make_8259A_irq(unsigned int irq)
{
    disable_irq_nosync(irq);
    io_apic_irqs &= ~(1<<irq);
    set_irq_chip_and_handler_name(irq, &i8259A_chip, handle_level_irq, "XT");
    enable_irq(irq);
}
```

关于irq_flow_handler_t

- 缺省为handle_bad_irq
- __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

- 以kernel/irq/chip.c::handle_level_irq为例：
  它调用kernel/irq/handle.c::handle_IRQ_event，
  进一步调用atction->handler，从而使得设备相关的
  中断处理函数得以运行
- 关于action->handler（设备相关的中断处理函数）：
  在kernel/irq/manage.c::setup_irq时，由irqaction参数给定。

Outline

do_IRQ的调用和返回

- 在调用do_IRQ之前，要保存上下文
  返回之后，要恢复上下文
- 在interrupt[]数组中定义的中断处理程序中
  - 每个入口地址转换成汇编码是如下的一些指令

```
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代码

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

arch/x86/kernel/irq_32.c::do_IRQ()

- 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

kernel/irq/handle.c::handle_IRQ_evnet()和中断服务例程

- 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)
```

Outline

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提供了三种方法来执行可延迟操作（函数）：
  1. 软中断
     - 软中断由内核静态分配（编译时确定）
  2. 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
     - Tasklet可以在运行时分配和初始化（例如装入一个内核模块时）
  3. 工作队列（ work queues ）

软中断、tasklet以及下半部分

- 一般而言，可延迟函数上可以执行4种操作
  1. 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
  2. 激活：
     设置可延迟函数在下一轮处理中执行
  3. 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
  4. 执行：
     在特定的时间执行可延迟函数

Outline

软中断softirq管理的关键数据结构

- Linux2.6.26使用有限个软中断，
  具体参见include/linux/interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
#ifdef CONFIG_HIGH_RES_TIMERS
    HRTIMER_SOFTIRQ,
#endif
    RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
};
```

  - 软中断的编号越小，优先级越高

软中断softirq管理的关键数据结构

- 软中断的向量表softirq_vec定义，参见kernel/softirq.c

```
static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;
```

- 每一个软中断向量由数据结构softirq_action定义，参见include/linux/interrupt.h

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

软中断的主要操作

- 软中断的关键操作包括
  1. 软中断向量的初始化
  2. 软中断的触发
  3. 软中断的屏蔽和激活？？local_bh_enable/disable
  4. 软中断的检查和处理

1、软中断向量的初始化

- 初始化函数为kernel/softirq.c::open_softirq

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

例如

在kernel/softirq.c::softirq_init中：

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

在net/core/dev.c::net_dev_init中：

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

3、软中断的触发

- 软中断触发函数为raise_softirq，它对所属CPU的irq_stat::__softirq_pending中的对应位置1

irq_stat定义，参见include/asm – x86/hardirq_32.h

```
typedef struct {
    unsigned int __softirq_pending;
...
} ____cacheline_aligned irq_cpustat_t;

DECLARE_PER_CPU(irq_cpustat_t, irq_stat);
```

- raise_softirq
  →raise_softirq_irqoff
  →include/linux/interrupt.h::__raise_softirq_irqoff

```
#define __raise_softirq_irqoff(nr) do { or_softirq_pending(1UL << (nr)); } while (0)
```

4、软中断的检查和处理 I

- 宏local_softirq_pending()用于访问所属CPU的irq_stat::__softirq_pending域
- 在某些特定的时机，会读取该域的值以检查是否有软中断被挂起，若有软中断被挂起，就调用do_softirq 。
  这种时机，称为检查点，例如
  - 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??
  - 当那个特定的进程ksoftirqd被唤醒时
  - …
- 在每个检查点，若有软中断被挂起，就调用do_softirq
  - 软中断处理的核心代码是__do_softirq
- 在处理完一轮软中断之后，若发现又有新的软中断被激活，则再次执行软中断处理主流程；
  重复执行超过一定次数，就唤醒ksoftirqd进程继续处理

4、软中断的检查和处理 II

```
asmlinkage void __do_softirq(void) {
    ...
restart: /* Reset the pending bitmask before enabling irqs */
    set_softirq_pending(0);
    local_irq_enable();
    h = softirq_vec;
    do {
        if (pending & 1) {
            h->action(h);
            rcu_bh_qsctr_inc(cpu);
        }
        h++;
        pending >>= 1;
    } while (pending);
    local_irq_disable();
    pending = local_softirq_pending();
    if (pending && --max_restart)
        goto restart;
    if (pending) wakeup_softirqd();
    ...
}
```

Ksoftirqd内核线程

- Linux在初始化时调用kernel/softirq.c::spawn_ksoftirqd()为每一个逻辑CPU创建一个ksoftirqd内核线程

  ```
  static DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
  ```

- 创建ksoftirqd的关键代码为：

```
p = kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu);
...
kthread_bind(p, hotcpu);
per_cpu(ksoftirqd, hotcpu) = p;
```

Ksoftirqd内核线程

- 必要时，ksoftirqd内核线程会被唤醒

```
/*
 * we cannot loop indefinitely here to avoid userspace starvation,
 * but we also don't want to introduce a worst case 1/HZ latency
 * to the pending events, so lets the scheduler to balance
 * the softirq load for us.
 */
static inline void wakeup_softirqd(void) {
    /* Interrupts are disabled: no need to stop preemption */
    struct task_struct *tsk = __get_cpu_var(ksoftirqd);
    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

- 例如，在do_softifq()开中断处理完若干次pending的软中断之后，若发现又有新的软中断pending，则唤醒ksoftirqd内核线程

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

Ksoftirqd内核线程

- ksoftirqd内核线程的核心工作是：

```
while (local_softirq_pending()) {
    ...
    do_softirq();
    ...
}
```

- 使用ps -lef查看ksoftirqd的信息，例如进程号、优先级

Outline

Tasklet

- Tasklet是I/O驱动程序中实现可延迟函数的首选方法。
  在软中断之上实现。

1. tasklet的数据结构及其组织
2. tasklet的使用及相关接口
3. tasklet的处理

1、tasklet的数据结构及其组织

1. tasklet的数据结构：tasklet_struct
   参见include/linux/interrupt.h

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

- state值：tasklet的状态

```
enum {
    TASKLET_STATE_SCHED, /* Tasklet is scheduled for execution */
    TASKLET_STATE_RUN /* Tasklet is running (SMP only) */
};
```

- count值：tasklet的屏蔽情况
  - >0：被屏蔽
  - =0：没有被屏蔽

1、tasklet的数据结构及其组织

2. tasklet的组织：Tasklet和高优先级的tasklet
   - 分别存放在每个CPU的tasklet_vec和tasklet_hi_vec链表中
   - 链表头定义如下，参见kernel/softirq.c

```
/* Tasklets */
struct tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};
/* Some compilers disobey section attribute on statics when not initialized — RR */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec) = { NULL };
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec) = { NULL };
```

   - 这两个percpu链表在kernel/softirq.c::softirq_init中进行初始化。

2、Tasklet的使用及相关接口

- 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tasklet_vec或者tasklet_hi_vec的相应CPU的链表上，将在合适的时机得到处理

1. tasklet的初始化：kernel/softirq::tasklet_init

2、Tasklet的使用及相关接口

2. tasklet的禁止和允许：屏蔽情况
   - tasklet_enable
     - tasklet_struct::count减1
   - tasklet_disable和tasklet_disable_nosync
     - tasklet_struct::count加1
   - 参见include/linux/interrupt.h，对于一个已经被激活的tasklet
     - count值大于0，表示tasklet被屏蔽，不能处理它
     - count值等于0，处理

2、Tasklet的使用及相关接口

3. tasklet的激活
   - 调用tasklet_schedule或者tasklet_hi_schedule将tasket插入到对应的链表中
   - 原则1：已经在链表中的tasklet，即状态为TASKLET_STATE_SCHED，不再重复插入
   - 原则2：激活一个tasklet时，还需要触发相关软中断，使得tasklet有机会得到处理

3、tasklet的处理

- tasklet分别在软中断TASKLET_SOFTIRQ和HI_SOFTIRQ中得到处理
- tasklet_action和tasklet_hi_action分别是tasklet_vec和tasklet_hi_vec链表中的tasklet的处理函数
  - 找到CPU对应的那个项，遍历执行
- 它们分别被注册为软中断的软中断处理函数，参见kernel/softirq.c::softirq_init

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

Outline

工作队列和工作线程

- 相关数据结构
  - 参见kernel/workqueue.c:
    - workqueue_struct；cpu_workqueue_struct
  - 参见include/linux/workqueue.h
    - work_struct；delayed_work
- 入列，参见kernel/workqueue.c
  - queue_work；queue_delayed_work
- 工作队列的处理
  - run_workqueue
  - worker_thread
- create_workqueue用来创建一个工作者队列
  - 查看Linux-2.6.26中，工作者队列的创建情况
  - 进一步查看工作者队列的实例“events”的使用情况

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：
  1. 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
  2. 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
  3. 挂起的信号
     - 如果一个信号发送到进程，那必须处理它
  4. 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

Outline

作业

- 名词解释：故障和陷阱

小结

Thanks !

The end.