第13章 中断与系统调用

实验目的

- 加深对中断机制和系统调用原理的理解
- 深入了解系统调用的执行流程
- 学会增加系统调用及向内核添加内核函数

主要内容

- 背景知识
  - 中断机制
  - 系统调用
- 实验内容
  - 记录系统调用的使用次数

中断概念

- 内核的一个主要功能就是处理硬件
  - 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
  - 也可以用轮询的方式来处理，但显然效率不高
  - 中断机制就是满足上述条件的一种解决办法
- 中断
  - 直接处理由硬件发过来的中断信号
    - ✓ CPU停止正在执行的指令，转而执行中断服务例程
    - ✓ 中断服务例程一般在CPU的中断方式下运行查看相应设备的状态寄存器变化，并做相应操作
    - ✓ 当中断处理完毕以后，CPU将恢复到以前的状态，继续执行中断处理前正在执行的指令

中断与异常

- 中断（外中断）
  - 异步的，来自处理器之外的中断信号，在程序执行的任何时候可能出现
  - 会改变处理器执行指令的顺序
  - 通常与CPU芯片内部/外部硬件电路产生的电信号相对应
- 异常（内中断）
  - 同步的，在（特殊的或出错的）指令执行时由CPU控制单元产生
  - 内核为每个异常提供了一个专门的异常处理程序
  - 异常处理程序的执行一般依赖于执行程序的当前现场，不能被屏蔽掉，一旦出现应立即响应并进行处理
- 区别
  - 中断允许嵌套发生，但异常多数情况为一重
  - 异常处理过程中可能产生中断，但反之则不会发生
- “中断信号”通称这两种类型的中断

中断分类

- 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
  - 可屏蔽中断状态
    - 屏蔽的(masked)
    - 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

有选择屏蔽与全局屏蔽

- 有选择屏蔽/激活IRQ线≠全局屏蔽/激活
- 有选择屏蔽
  - 前者通过对中断控制器编程实现
  - 有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
- 全局屏蔽
  - 通过特定的指令操作CPU中的状态字，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

异常分类

- 故障(fault)
  - 程序运行中系统捕获出现的潜在可恢复错误，处理后可返回到当前指令再次执行
  - 如页面故障
- 陷阱(trap)
  - 执行特定调试指令触发的，被调试的进程遇到所设置的断点，会暂停等待
- 编程异常(programmed excption)
  - 用来实现系统调用，进程自愿进入内核态以请求系统服务
- 终止(abort)
  - 终止是发现致命的不可恢复错误，通常不会返回原程序而转向内核特殊函数处理

中断信号的作用

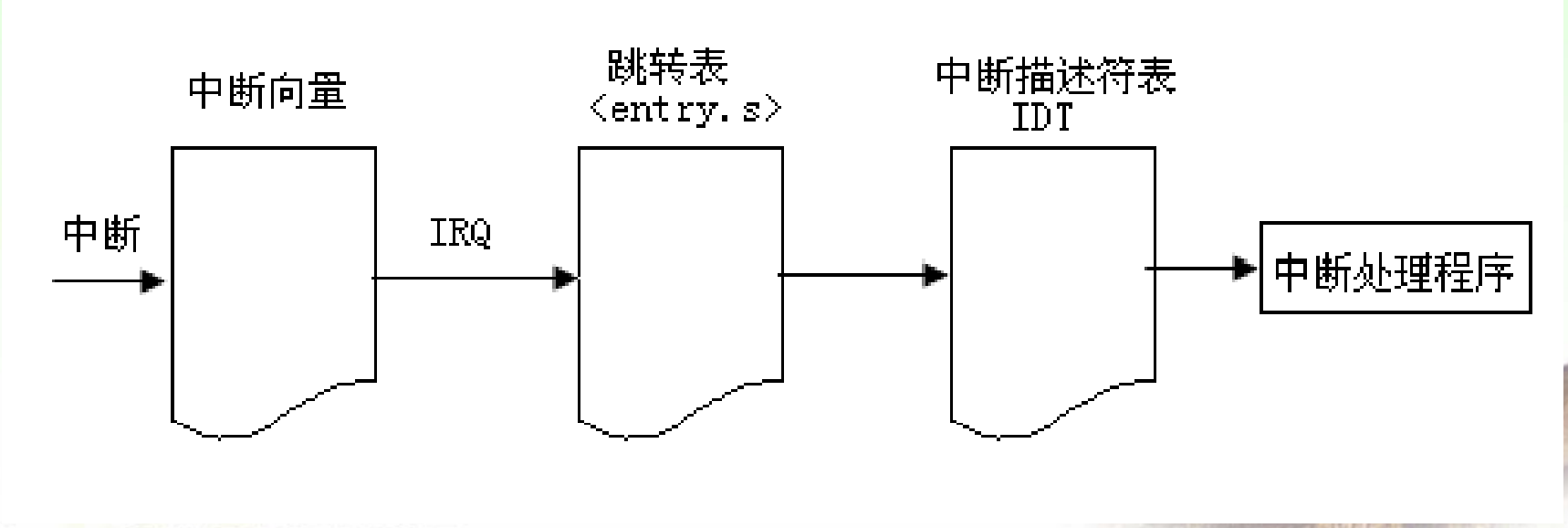
- 中断信号提供了一种特殊的方式，使得CPU转去运行正常程序之外的代码
  - 比如一个外设采集到一些数据，发出一个中断信号，CPU必须立刻响应这个信号，否则数据可能丢失
- 当一个中断信号到达时，CPU必须停止它当前正在做的事，并且切换到一个新的活动
- 为了做到这这一点，在进程的内核态堆栈保存程序计数器的当前值(即eip和cs寄存器)，并把与中断信号相关的一个地址放入进程序计数器

中断信号的处理原则

- 核心目标：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
  - 上半部分(top bottom)和下半部分(half bottom)
- 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

中断处理流程

- 中断的获取
- 中断向量的构造
- 跳转表的映射
- 中断描述符表的构造
- 中断处理程序的实现



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
  - 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
- 进程上下文与中断上下文的关系
  - 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
  - 无论如何，Linux的interrupt context A绝对不会被某个进程C或者D抢占！！
  - interrupt contexts之间切换或执行代码的过程，决不可能插入scheduler调度例程的调用
    - ✓ 首先，interrupt context 没有process context，如果被 某个进程抢占之后就没法恢复到原来的interrupt context，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

中断上下文举例

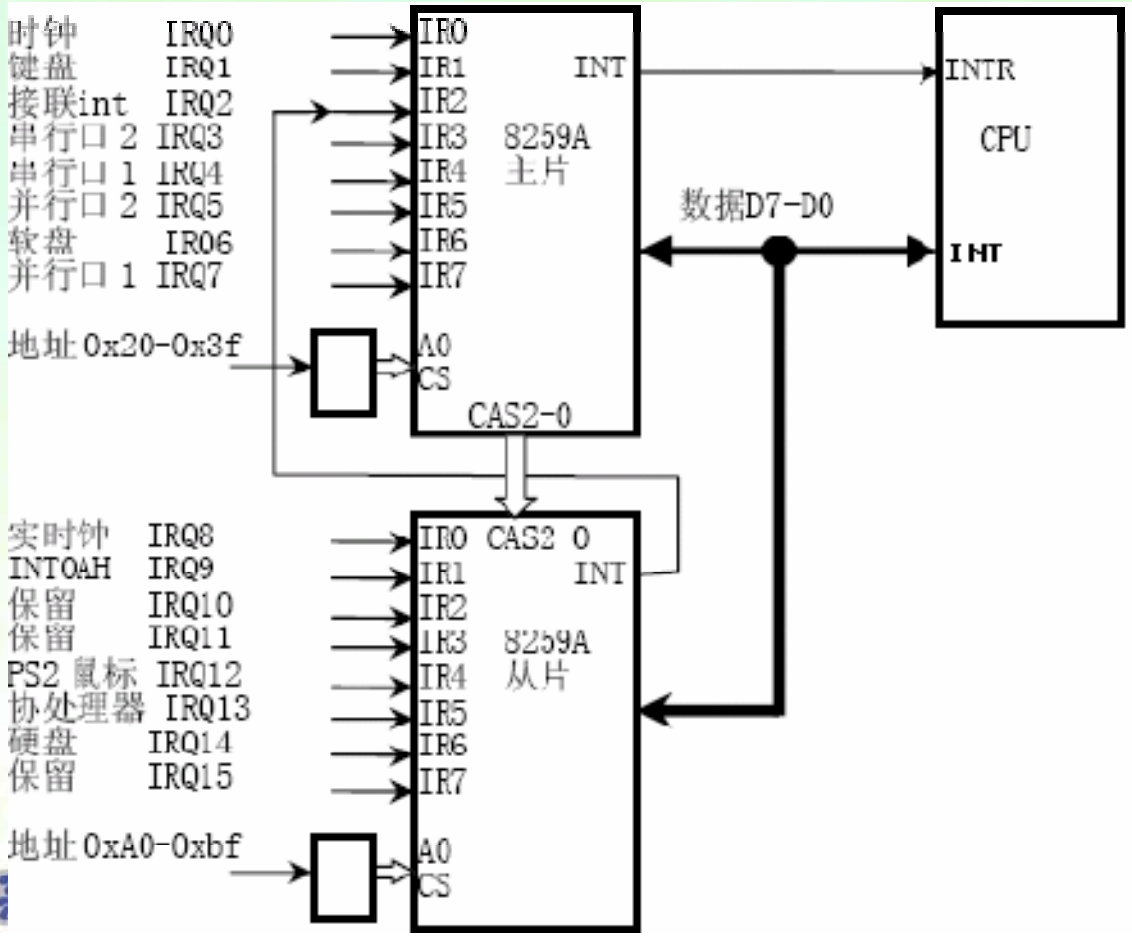
- 中断上下文之间的切换关系
  - 如果CPU的IF flag标志打开
    - ✓ 在A执行过程中，若B的irq 线已经触发了PIC，进而触发了CPU IRQ pin，使得CPU执行中断B的interrupt context，这是中断上下文的嵌套过程。
    - ✓ 通常Linux不对不同的interrupt contexts设置优先级，这种任意的嵌套是允许的
    - ✓ 也可能某个实时Linux的patch会不允许低优先级的interrupt context抢占高优先级的interrupt context

X86中断的硬件支持

- 8259a中断控制器
  - 硬件设备通过8259a向CPU发出中断请求
- 中断向量表寄存器IDTR
  - 存放中断向量表IDT的基址
- CPU的四种门
  - 任务门、中断门，陷阱门和调用门
- 任务状态段TSS
  - 复杂的数据结构，用于保存当前任务的运行状态
  - Linux中断机制利用了其中的堆栈指针
- 硬件的中断机制

8259A中断控制器

- 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，一共可以处理最多15个不同的IRQ线



中断控制器工作机制

- 监视IRQ线，对引发信号检查
  - 如果一个引发信号出现在IRQ线上
    - ✓把此信号转换成对应的中断向量
    - ✓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
    - ✓把引发信号发送到处理器的INTR引脚，即产生一个中断
    - ✓等待，直到CPU应答这个信号；收到应答后，清INTR引脚
- 重复上述过程

中断向量

- 中断处理在内核的入口
- 在保护模式下的实现机制是中断描述符表IDT
  - IDT的位置由CPU的中断描述符表寄存器idtr确定
    - ✓ idtr是个48位的寄存器
    - ✓ 高32位是IDT的基址，低16位为IDT的界限(通常为2k=256*8)。

	47 16	15 0
IDTR寄存器	基地址	界限

  - 包含256个中断描述符，对应硬件提供的256个中断服务例程的入口，每个中断描述符8个字节
    - ✓ 非屏蔽中断向量和异常向量是固定的
    - ✓ 可屏蔽中断的向量可以通过对中断控制器的编程来改变
  - 内核在允许中断发生前，必须适当的初始化IDT

Linux的中断向量分配情况

- 每个中断/异常都有一个向量号，该号的值在0~255之间，该值是中断/异常在IDT中的索引

向量范围	用途
0~19（0x0~0x13）	非屏蔽中断和异常
21~31（0x14~0x1f）	Intel 保留
32~127（0x20~0x7f）	外部中断（IRQ）
128（0x80）	系统调用
129~238（0x81~0xee）	外部中断（IRQ）
239（0xef）	本地 APIC 时钟中断
240~250（0xf0~0xfa）	由 Linux 留做将来使用
251~255（0xfb~0xff）	处理器间中断

- 常见异常中断号及处理程序表

#	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

IRQ线和中断向量号

- 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、…

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

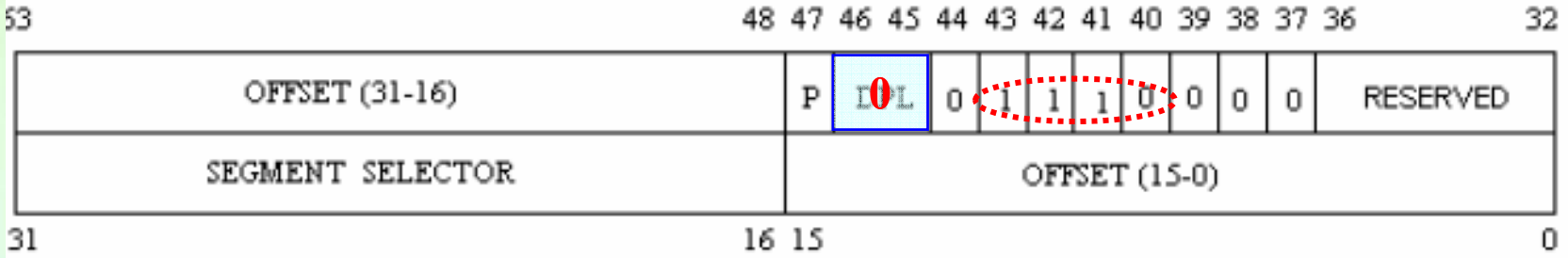
中断描述符表分类

- 中断门
  - 用户态进程不能访问的一个Intel中断门(特权级为0)，
  - 所有中断都通过中断门激活，并全部在内核态
- 陷阱门
  - 用户态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
- 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，即
    - ✓向量是3，4，5和128，相应地，在用户态下可以发布int3，overflow，bound和int $0x80四条汇编指令

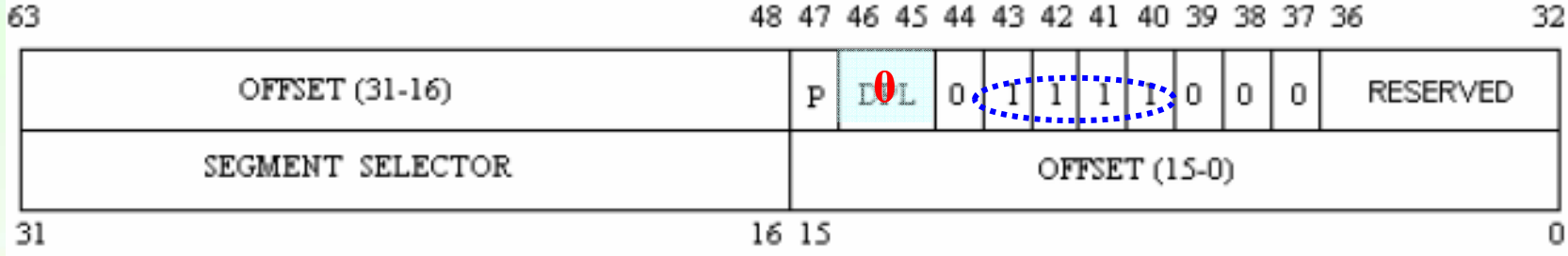
3种门的类型描述符

- 每个描述占8个字节

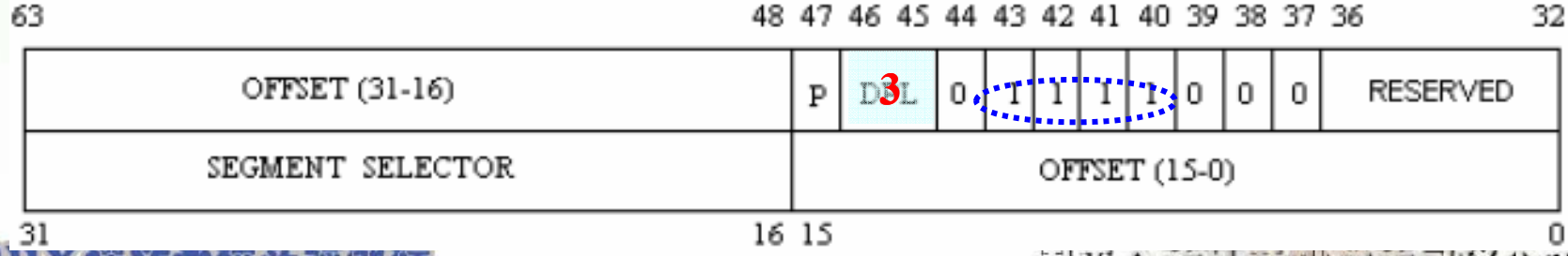
中断门描述符



陷阱门描述符

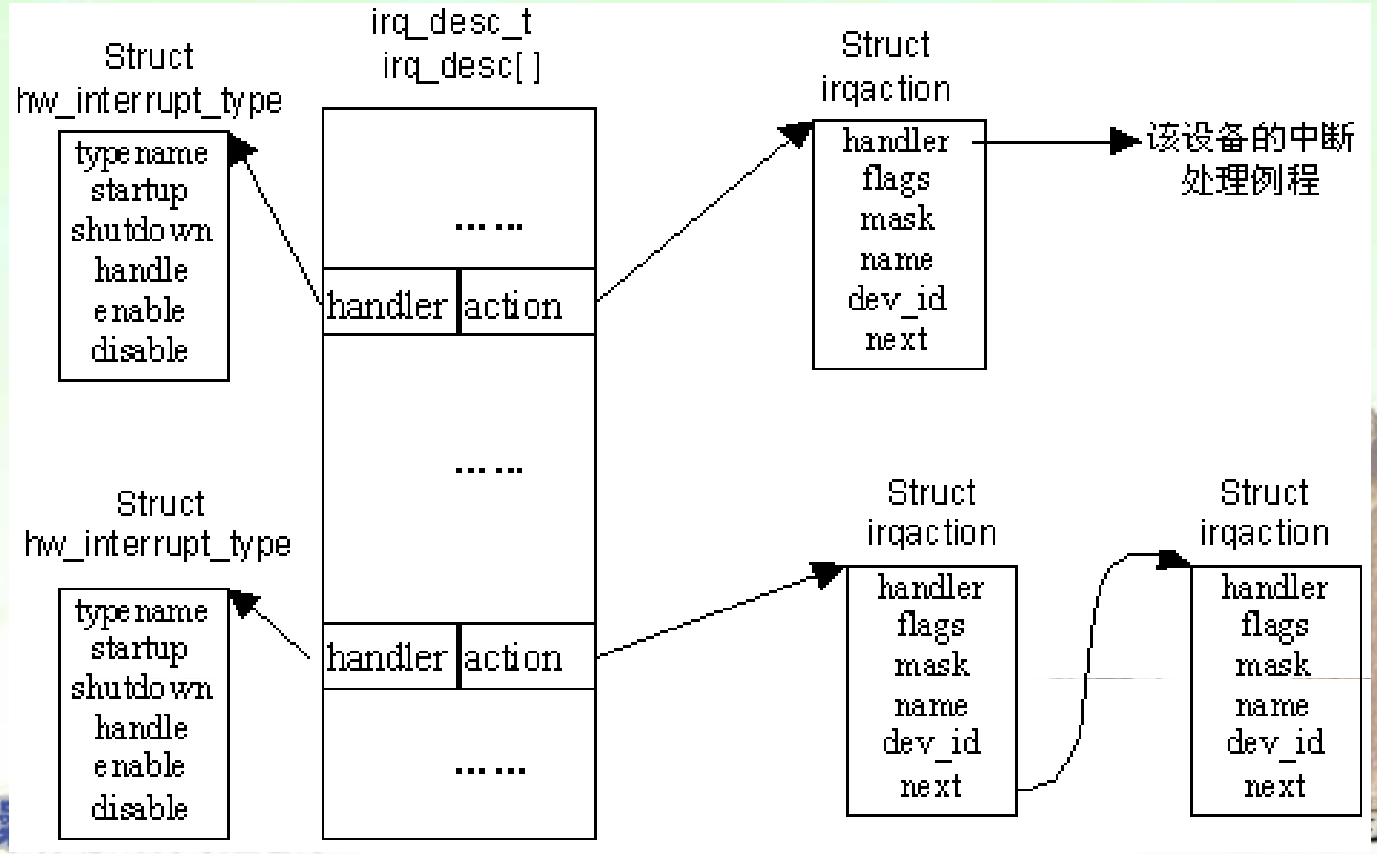


系统门描述符



中断处理相关数据结构

- hw_interrupt_type结构
- irq_desc_t结构
- irqaction结构



hw_interrupt_type

- 抽象的中断控制器，包含一系列的指向函数的指针
  - typename: 控制器的名字
  - startup/ shutdown: 允许/禁止从给定控制器的IRQ所产生的事件
  - enable和disable: 功能与startup和shutdown相同

```
struct hw_interrupt_type {
    const char * typename;
    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 (*end)(unsigned int irq);
    void (*set_affinity)(unsigned int irq, unsigned long mask);
};
```

8259A的hw_interrupt_type

- 8259A的中断控制操作

```
static struct hw_interrupt_type i8259A_irq_type = {
    "XT-PIC",
    startup_8259A_irq,
    shutdown_8259A_irq,
    enable_8259A_irq,
    disable_8259A_irq,
    mask_and_ack_8259A,
    end_8259A_irq,
    NULL
};
```

- 所有连接到8259A的外设中断都要设置

```
void make_8259A_irq(unsigned int irq)
{
    disable_irq_nosync(irq);
    io_apic_irqs &= ~(1<<irq);
    irq_desc[irq].handler = &i8259A_irq_type;
    enable_irq(irq);
}
```

irq_desc_t和irq_desc结构

- irq_desc_t描述中断源
- irq_desc[ ]描述所有的中断源

```
typedef struct {
        unsigned int status;            /* IRQ status */
        hw_irq_controller *handler;
        struct irqaction *action;       /* IRQ action list */
        unsigned int depth;             /* nested irq disables */
        spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];
```

中断控制器处理例程

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

```
#define NR_IRQS 224
```

```
irq_desc_t irq_desc[NR_IRQS] __cacheline_aligned =
    { [0 ... NR_IRQS-1] = { 0, &no_irq_type, NULL, 0, SPIN_LOCK_UNLOCKED}};
```

irq_desc_t结构

- 参数说明
  - status：代表IRQ的状态
    - IRQ是否被禁止
    - 有关IRQ的设备当前是否正被自动检测
  - handler：指向hw_interrupt_type
  - action：指向irqaction结构组成的队列的头
    - 正常情况下每个IRQ只有一个操作，因此链接列表的正常长度是1（或者0）
    - 如果IRQ被两个或者多个设备所共享，队列中就有多个操作
  - depth：irq_desc_t的当前用户的个数
    - 主要是用来保证在中断处理过程中IRQ不会被禁止

irqaction结构

- 包含内核接收到特定IRQ之后应采取的操作
- 用来实现IRQ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

```
struct irqaction {        中断处理程序
        void (*handler)(int, void *, struct pt_regs *);
        unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;   链表
};
```

irqaction结构

- 参数说明
  - flags: 指示中断类型
    - ✓ SA_INTERRUPT：快速中断处理程序
    - ✓ SA_SAMPLE_RANDOM：这个中断是源于物理随机性的
    - ✓ SA_SHIRQ：此IRQ与其它struct irqaction共享
  - mask: 中断屏蔽字
    - ✓ 在x86或者体系结构无关的代码中不会使用
    - ✓ 在SPARC64的移植版本中要跟踪有关软盘的信息时才会使用它
  - name: 产生中断的硬件设备的名字
  - dev_id: 标识硬件类型的一个唯一的ID
    - ✓ Linux支持的所有硬件设备的每一种类型，都有一个由制造厂商定义的在此成员中记录的设备ID
  - next: 指向下一个irqaction
    - ✓ 如果IRQ是共享的，则指向队列中下一个struct irqaction

中断描述符表的初始化

- 实模式下的初始化
  - 在系统引导时进行
  - 执行call setup_idt
    - ✓将中断向量表中的每项初始化为默认的中断服务例程ignore_int
  - 执行lidt idt_descr
    - ✓装载中断描述表的偏移地址到IDTR寄存器
- 保护模式下的初始化
  - 在start_kernel中完成，重新设置中断向量号
    - ✓trap_init()：对一些系统保留的中断向量的初始化
    - ✓init_IRQ()：用于外设中断

setup_idt

- 定义位置
  - arch/i386/kernel/head.S
- 用ignore_int填充256个idt_table表项

```
setup_idt:
        lea ignore_int,%edx     将默认的中断服务例程偏移地址装载到edx寄存器
        movl $(__KERNEL_CS << 16),%eax
        movw %dx,%ax                    /* selector = 0x0010 = cs */
        movw $0x8E00,%dx                /* interrupt gate - dpl=0, present */

        lea SYMBOL_NAME(idt_table),%edi  //将中断向量表的首地址装载到edi寄存器中
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi 将edi移到下一个描述符处
        dec %ecx
        jne rp_sidt
        ret
```

EDX

addr_high	attributes

EAX

__KERNEL_CS	addr_low

ignore_int

```
/* This is the default interrupt "handler" :-) */
int_msg:
        .asciz "Unknown interrupt\n"
        ALIGN
ignore_int:
        cld
        pushl %eax
        pushl %ecx
        pushl %edx
        pushl %es
        pushl %ds
        movl $(__KERNEL_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        pushl $int_msg
        call SYMBOL_NAME(printk)
        popl %eax
        popl %ds
        popl %es
        popl %edx
        popl %ecx
        popl %eax
        iret
```

start_kernel()函数

```
asmlinkage void __init start_kernel(void)
{
        char * command_line;
        unsigned long mempages;
        extern char saved_command_line[];
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
        lock_kernel();
        printk(linux_banner);
        setup_arch(&command_line);
        printk("Kernel command line: %s\n", saved_command_line);
        parse_options(command_line);
        trap_init();
        init_IRQ();
        sched_init();
        softirq_init();
        time_init();
```

trap_init()函数

```
void __init trap_init(void)
{
#ifdef CONFIG_EISA
        if (isa_readl(0x0FFFD9) == 'E'+('I'<<8)+('S'<<16)+('A'<<24))
                EISA_bus = 1;
#endif

        set_trap_gate(0,&divide_error);
        set_trap_gate(1,&debug);
        set_intr_gate(2,&nmi);
        set_system_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_trap_gate(8,&double_fault);
        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(15,&spurious_interrupt_bug);
        set_trap_gate(16,&coprocessor_error);
        set_trap_gate(17,&alignment_check);
        set_trap_gate(18,&machine_check);
        set_trap_gate(19,&simd_coprocessor_error);

    set_system_gate(SYSCALL_VECTOR,&system_call);
```

20个异常

系统调用

init_IRQ()函数

❖往中断向量表中填外部中断的中断门，共填224个

```
void __init init_IRQ(void)
{
        int i;

#ifndef CONFIG_X86_VISWS_APIC
        init_ISA_irqs();
#else
        init_VISWS_APIC_irqs();
#endif
        /*
         * Cover the whole vector space, no vector can escape
         * us. (some of these will be overridden and become
         * 'special' SMP interrupts)
         */
        for (i = 0; i < NR_IRQS; i++) {
                int vector = FIRST_EXTERNAL_VECTOR + i;
                if (vector != SYSCALL_VECTOR)
                        set_intr_gate(vector, interrupt[i]);
        }
```

```
#define FIRST_EXTERNAL_VECTOR  0x20

#define SYSCALL_VECTOR         0x80
```

interrupt[i]为函数指针数组，这些函数指针实际上是宏，在形式上它们是一样的。

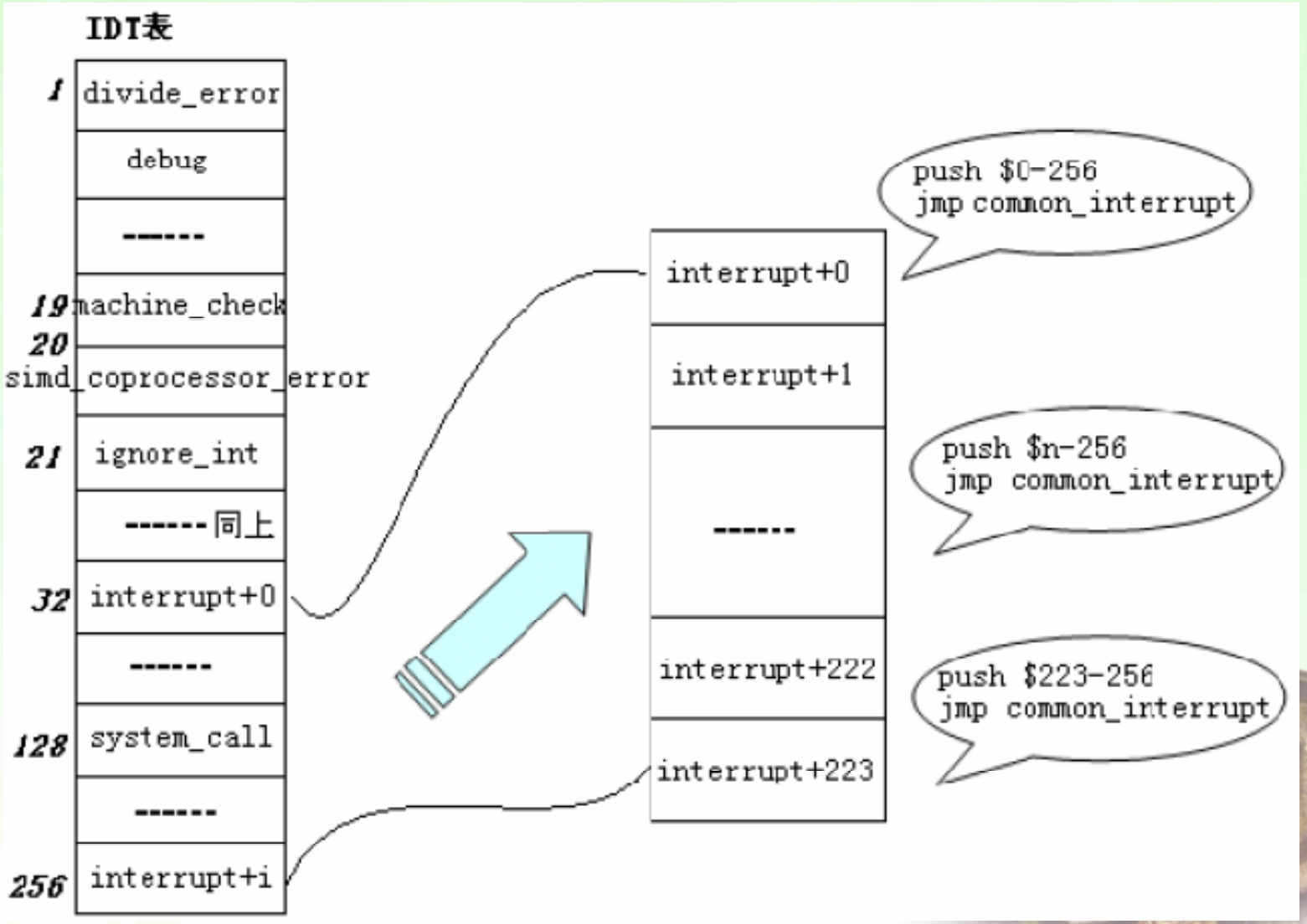
每个中断程序入口的定义

```
BUILD_COMMON_IRQ()

#define BI(x,y) \
        BUILD_IRQ(x##y)

#define BUILD_16_IRQS(x) \
        BI(x,0) BI(x,1) BI(x,2) BI(x,3) \
        BI(x,4) BI(x,5) BI(x,6) BI(x,7) \
        BI(x,8) BI(x,9) BI(x,a) BI(x,b) \
        BI(x,c) BI(x,d) BI(x,e) BI(x,f)

/*
 * ISA PIC or low IO-APIC tri
 * (these are usually mapped
 */
BUILD_16_IRQS(0x0)
```

```
#define BUILD_IRQ(nr) \
asmlinkage void IRQ_NAME(nr); \
__asm__( \
"\n"__ALIGN_STR"\n" \
SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \
        "pushl $"#nr"-256\n\t" \
        "jmp common_interrupt");
```

```
#define BUILD_COMMON_IRQ() \
asmlinkage void call_do_IRQ(void); \
__asm__( \
        "\n" __ALIGN_STR"\n" \
        "common_interrupt:\n\t" \
        SAVE_ALL \
        SYMBOL_NAME_STR(call_do_IRQ)":\n\t" \
        "call " SYMBOL_NAME_STR(do_IRQ) "\n\t" \
        "jmp ret_from_intr\n");
```

初始化后IDT结构


IDT表
1 divide_error
debug
------
19 machine_check
20
simd_coprocessor_error
21 ignore_int
------同上
32 interrupt+0
------
128 system_call
------
256 interrupt+i
interrupt+0
interrupt+1
------
interrupt+222
interrupt+223
push $0-256
jmp common_interrupt
push $n-256
jmp common_interrupt
push $223-256
jmp common_interrupt

中断处理的一般过程

- 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
  - 在内核态堆栈保存IRQ的值和寄存器的内容
  - 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
  - 执行共享这个IRQ的所有设备的中断服务例程
  - 跳到ret_from_intr()的地址

中断请求队列的初始化

- X86只支持256个中断向量，这对众多的外设显然是不够的
- Linux中，每个外部中断向量都有一个中断请求队列，允许外设的中断为多个中断源共享
- 在填中断向量表之前，要对中断请求队列进行初始化

中断请求队列相关数据结构

```
typedef struct{
        unsigned intstatus; //IRQ状态
        hw_irq_controller*handler;//中断服务队列共用“中断通道”的控制器
        structirqaction*action;//指向中断服务队列
        unsigned intdepth;
        spinlock_tlock;
} irq_desc_t;
extern irq_desc_tirq_desc[NR_IRQS];中断请求队列数组
struct irqaction{
        void (*handler)(int, void *, structpt_regs*);//指向具体的设备中断处理程序
        unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
        struct irqaction*next;
};
```

中断队列初始状态

- 在IDT表的初始化完成之初中断队列都为空
  - 即使打开中断且发生外设中断，也得不到实际服务
  - 具体中断服务程序由设备驱动程序通过**request_irq()**向系统登记以挂到中断请求队列上

```
void __init init_ISA_irqs (void){
    int i;
#ifdef CONFIG_X86_LOCAL_APIC
    init_bsp_APIC();
#endif
    init_8259A(0);
    for (i = 0; i < NR_IRQS; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = 0;
        irq_desc[i].depth = 1;

        if (i < 16) {
            /*16 old-style INTA-cycle interrupts:*/
            irq_desc[i].handler = &i8259A_irq_type;
        } else {
            /** 'high' PCI IRQs filled in on demand*/
            irq_desc[i].handler = &no_irq_type;
        }
    }
} ? end init_ISA_irqs ?
```

setup_irq()

- 注册中断处理程序
  - 成功时返回0，失败时返回-1
  - 调用时，内核需要在/proc/irq中创建一个与中断对应的项
  - 不可以在中断上下文或不允许阻塞的代码中使用，该函数调用可能会睡眠（与内存申请相关）

```
int setup_irq(unsigned int irq, struct irqaction * new)
{
        ...
        int shared = 0;
        unsigned long flags;
        struct irqaction *old, **p;
        irq_desc_t *desc = irq_desc + irq;
```

- 调用方法

```
if (request_irq(irqn,my_interrupt,SA_SHIRQ,"my_device",dev)){
        printk(KERN_ERR "my_device:cannot register IRQ %d\n",irqn);
        return -EIO;
}
```

free_irq()

- 释放中断处理程序
  - 如果指定的中断线不是共享的，则该函数删除处理程序的同时将禁用该中断线
  - 如果是共享的，则仅删除dev_id所对应的处理程序
  - 中断线本身只有在删除最后一个处理程序时才会被禁用

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

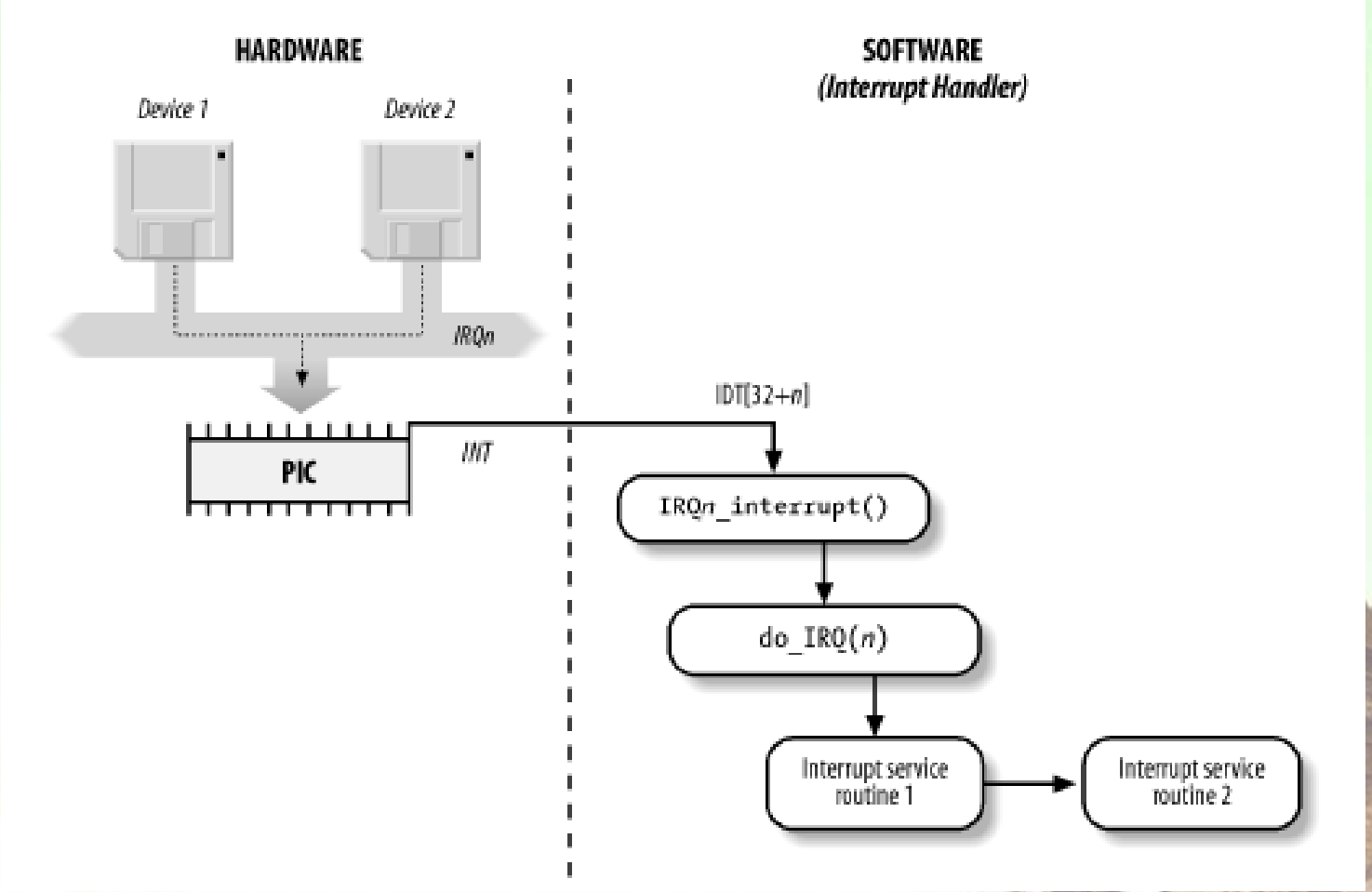
中断处理

- 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)

中断处理执行操作分类

- 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)
  - 一般在开中断的情况下，如修改那些只有处理器才会访问的数据结构
  - 也要很快完成，因此由中断处理程序立即执行
- 非紧急可延迟的(noncritical deferrable)
  - 内核用下半部分机制来在一个更为合适的时机用独立的函数来执行
  - 这些操作可以被延迟较长的时间间隔而不影响内核操作

中断响应与服务处理过程示意图



Linux的中断处理机制

- **Linux中断服务程序被分成上半部分(top half)和下半部分(bottom half)两个处理函数**
  - 上半部分函数每当接收到中断，立即开始工作，在关中断状态下只做严格限时且与硬件相关的任务，然后“标记中断”，通知下半部分做剩余工作
  - 这种部分工作在关中断状态下处理，另外的工作由可中断代码来处理称为中断下半部分处理，可见这是一种任务延迟处理机制

Linux的下半部分处理实现机制

- 软中断(softirq)
  - Linux沿用最早的BH思想,是一种软中断机制
  - 最多可注册32个软中断，目前版本已预定义6个
  - softirq保留给对时间要求严格的下半部分使用
- 任务队列(task queue)
  - 内核定义一组队列，每个队列包含一个由等待调用的函数组成的链表
- 小任务(tasklet)
  - 基于软中断来实现，能更好支持SMP，锁保护要求低
  - 在软中断上下文中运行，所有tasklet代码必须是原子的
- 工作队列(work queue)
  - 把一个任务延迟，并交给内核线程去完成，且该任务总是在进程上下文中执行

中断响应与服务过程

❖准备阶段

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

中断响应与服务过程

❖堆栈切换

- CPU从中断控制器中取得中断向量，然后根据这个中断向量和IDTR从中断向量表IDT中找到对应的中断门(外部中断不进行中断门优先级别的检查)
- 由于CPL与DPL不同，因此需要进行堆栈切换
  - ✓CPU从当前的TSS中用于内核0级的堆栈指针（ESP和SS）装入ESP和SS，切换到内核堆栈
  - ✓然后CPU将原来用户堆栈的指针、EFLAGS、返回地址（CS和EIP）压入到新的堆栈中，即内核堆栈中

中断响应与服务过程

- 进入中断服务程序入口
  - CPU根据中断门以及目标代码段的设置到达中断总服务程序的入口：IRQ0xXX_interrupt
  - 假设IRQ0x03_interrupt;
    - ✓IRQ0x03_interrupt先将中断请求号-256（为了与系统调用号区别开）得到的数值压入堆栈
    - ✓然后跳转到common_interrupt执行
- 对应汇编指令

```
IRQn_interrupt:
        pushl $n-256
        jmp common_interrupt
```

中断响应与服务过程

- 堆栈保存
  - 在common_interrupt中，先通过宏操作SAVE_ALL，保存现场
  - 再将ret_from_intr地址压入堆栈
  - 然后调用do_IRQ()
- 对应汇编指令

```
common_interrupt:
        SAVE_ALL
        call do_IRQ
        jmp $ret_from_intr
```

中断响应与服务过程

- 进入do_IRQ()
  - 根据中断调用号0x03从irq_desc[]数组中找到相应中断请求队列头
  - 调用handle_IRQ_event()
- do_IRQ()核心代码

```
int irq=regs.orig_eax & 0xff;        //1
irq_desc[irq].handler->ack(irq);  //2
handle_IRQ_event(irq,&regs,irq_desc[irq].action);//3
irq_desc[irq].handler->end(irq);  //4
处理下半部分                                    //5
```

  - 1句将$n-255转换成n，取得对应的中断向量
  - 2句应答PIC的中断，并禁用这条IRQ线
  - 3句调用handle_IRQ_event()执行中断服务例程
  - 4句通知PIC重新激活这条IRQ线，允许处理同类型中断

中断响应与服务过程

- 进入handle_IRQ_event()
  - 通过一个do-while循环，依次执行在前步中找到的中断请求队列中的各个服务程序，让每个服务程序去辨认本次中断请求是否来自各个服务对象
    - ✓ 队列中的每个中断服务程序执行时，先检查各自的中断源（一般是读相应设备上的中断状态寄存器）看是否有来自该设备的中断请求，如果没有，就马上返回
    - ✓ 有就继续执行

handle_IRQ_evnet()

```
int handle_IRQ_event(unsigned int irq,
  struct pt_regs * regs, struct irqaction * action){
     int status;
     int cpu = smp_processor_id();
     irq_enter(cpu, irq);
     status = 1; /* Force the "do bottom halves" bit */
     if (!(action->flags & SA_INTERRUPT))
          __sti();
     do {
          status |= action->flags;
          action->handler(irq, action->dev_id, regs);
          action = action->next;
     } while (action);
     if (status & SA_SAMPLE_RANDOM)
          add_interrupt_randomness(irq);
     __cli();
     irq_exit(cpu, irq);
     return status;
}
```

中断响应与服务过程

- 从handle_IRQ_event()返回到do_IRQ()
  - 在do_IRQ()最后还要处理软中断请求（以前称为bottom half)
  - 然后从do_IRQ()返回到entry.s标号为ret_from_intr处继续执行
    - ✓在前边已经把ret_from_intr的地址压入堆栈中，因此do_IRQ()返回时CPU将从那里开始执行

中断响应与服务过程

- 进入ret_from_intr
  - 检验中断前CPU是否运行在VM86模式，返回用户空间
  - 检验中断前CPU运行于用户空间还是系统空间
    - ✓ 如果中断发生在系统空间就转移到RESTORE_ALL
    - ✓ 否则转移到ret_with_reschedule（发生在用户空间）
      - 先检查一下是否要进行进程调度，如果需要调用schedule()调度进程
- 执行宏RESTORE_ALL
  - 与开始的SAVE_ALL相对应，就是“恢复现场”把调用前寄存器的值复原
- 执行ret从中断返回
  - CPU会切换堆栈（从系统到用户）就此完成一次中断过程。

编写中断处理程序

- 中断处理程序声明格式
  - static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs)
- 参数说明
  - irq：中断线号
  - dev_id：必须与传递给request_irq()的参数dev_id一致
  - regs：包含处理中断之前处理器的寄存器及状态，除非调试，很少使用
- 返回值
  - IRQ_NONE：检测到中断，但该中断对应的设备不是注册处理函数期间指定的产生源
  - IRQ_HANDLED：处理程序被正确调用，且是所对应的设备产生的中断
- 说明
  - 无须重入，给定中断处理程序运行期间，相应中断线在所有处理器上都会被屏蔽掉

pt_regs结构(恢复现场所需的上下文)

栈顶（低地址）

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int    xds;
        int    xes;
        long orig_eax;
        long eip;
        int    xcs;
        long eflags;
        long esp;
        int    xss;
```

栈底（高地址）

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

共享的中断处理程序说明

- 基本前提
  - request_irq()的参数flags必须设置SA_SHIRQ标志
  - 对每个注册的中断处理程序来说，dev_id参数必须唯一
  - 中断处理程序必须能区分它的设备是否真的产生了中断
- 内核对共享中断的处理
  - 依次调用该中断线上注册的每个处理程序
  - 中断处理程序必须知道是否对此中断负责（dev_id）

中断控制

- 操纵机器上的中断状态，实现同步
  - 禁止当前处理器上的中断系统
  - 屏蔽整个机器的一条中断线的能力

函　数	说　明
local_irq_disable()	禁止本地中断传递
local_irq_enable()	激活本地中断传递
local_irq_save()	保存本地中断传递的当前状态，然后禁止本地中断传递
local_irq_restore()	恢复本地中断传递到给定的状态
disable_irq()	禁止给定中断线，并确保该函数返回之前在该中断线上没有处理程序在运行
disable_irq_nosync()	禁止给定中断线
enable_irq()	激活给定中断线
irqs_disabled()	如果本地中断传递被禁止，则返回非0；否则返回0
in_interrupt()	如果在中断上下文中，则返回非0，如果在进程上下文中，则返回0
in_irq()	如果当前正在执行中断处理程序，则返回非0，否则，返回0

禁止和激活中断

- 函数原型
  - local_irq_disable()
  - local_irq_enable()
- 说明
  - 在x86中，前者等价与cli指令，后者等价于sti指令
  - 无条件地禁止/激活当前处理器上的本地中断，调用前需要做状态保存

```
unsigned long flags;

local_irq_save(flags);                  /* 禁止中断..*/
/*    ...              */
local_irq_restore(flags) ;              /*中断被恢复到它们原来的状态..*/
```

禁止/激活指定中断线

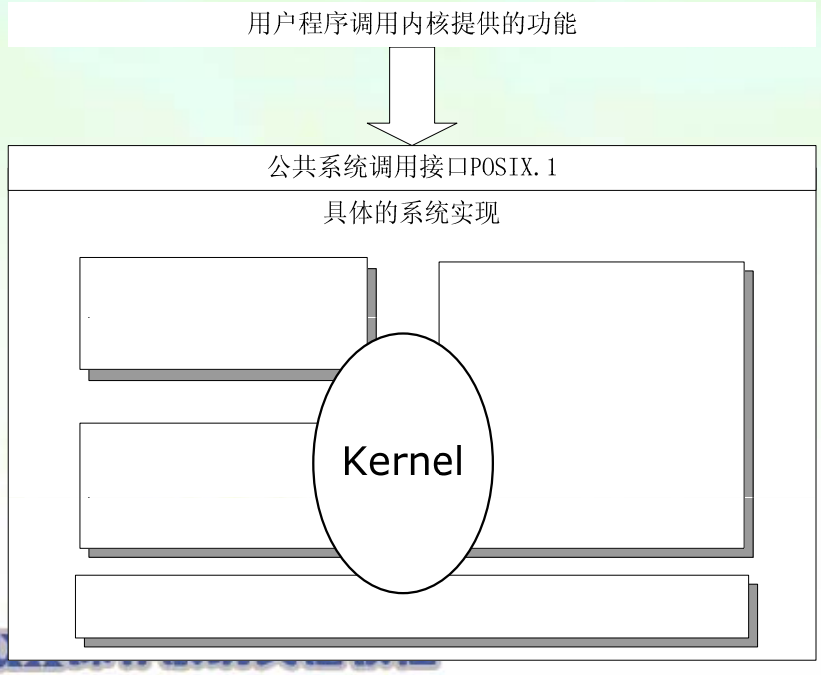
- 函数原型
  - void disable_irq(unsigned int irq)
  - void disable_irq_nosync(unsigned int irq)
  - void enable_irq(unsigned int irq)
  - void synchronize_irq(unsigned int irq)
- 说明
  - disable_irq()在当前正执行的所有处理程序完成后才能返回
    - ✓ 调用者需确保不在指定线上传递新的中断，还要确保所有已经开始执行的处理程序都全部退出
  - disable_irq_nosync不会等待当前中断处理程序执行完毕
  - synchronize_irq等待一个特定的中断处理程序的退出
  - 上述函数可以嵌套调用，但在指定中断线上， disable_irq_nosync或disable_irq()需有相匹配的enable_irq()

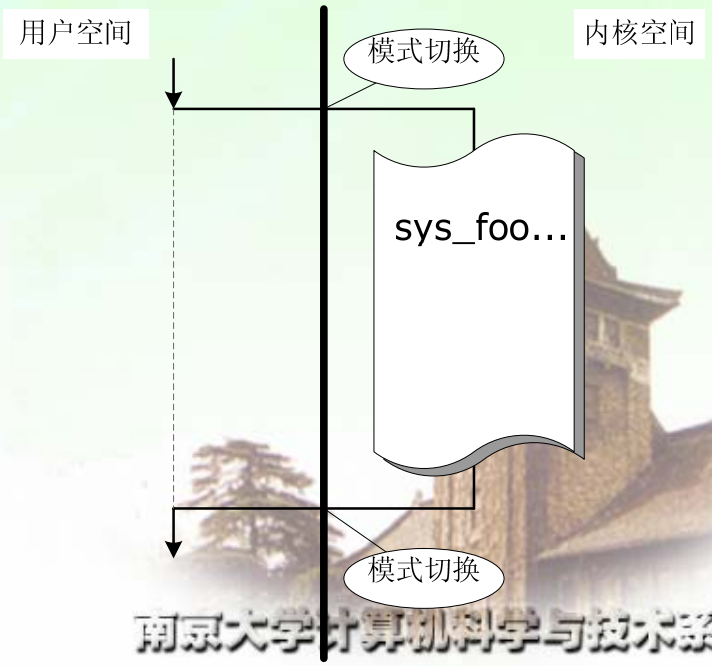
主要内容

- 背景知识
  - 中断机制
  - 系统调用
- 实验内容
  - 记录系统调用的使用次数

系统调用基本概念

- 位于用户空间进程和硬件设备之间的一个中间层
- 为用户态进程与硬件设备交互提供一组接口
  - 为用户空间提供一种硬件的抽象接口
  - 是除异常和陷入外，用户空间访问内核的唯一入口
  - 保证系统的稳定与安全





系统调用与操作系统安全机制的关系

- 保护模式的操作系统
  - 操作系统内核运行在特权级别（内核态），用户应用程序运行在非特权级别（用户态）
  - 应用程序无法直接调用任何操作系统代码，对入口地址的调用将引起异常，并被OS捕获
- 操作系统提供服务的实现
  - 受保护的中断机制
    - ✓ 授权给应用程序保护等级的中断号才可以被应用程序调用
    - ✓ 对未授权的中断号的调用将引起保护异常
    - ✓ Linux仅给应用程序授了4个中断号:3,4,5及80h
  - 系统调用
    - ✓ 被称为应用程序的陷阱，实现用户态与内核态的切换

系统调用涉及到的空间/模式切换

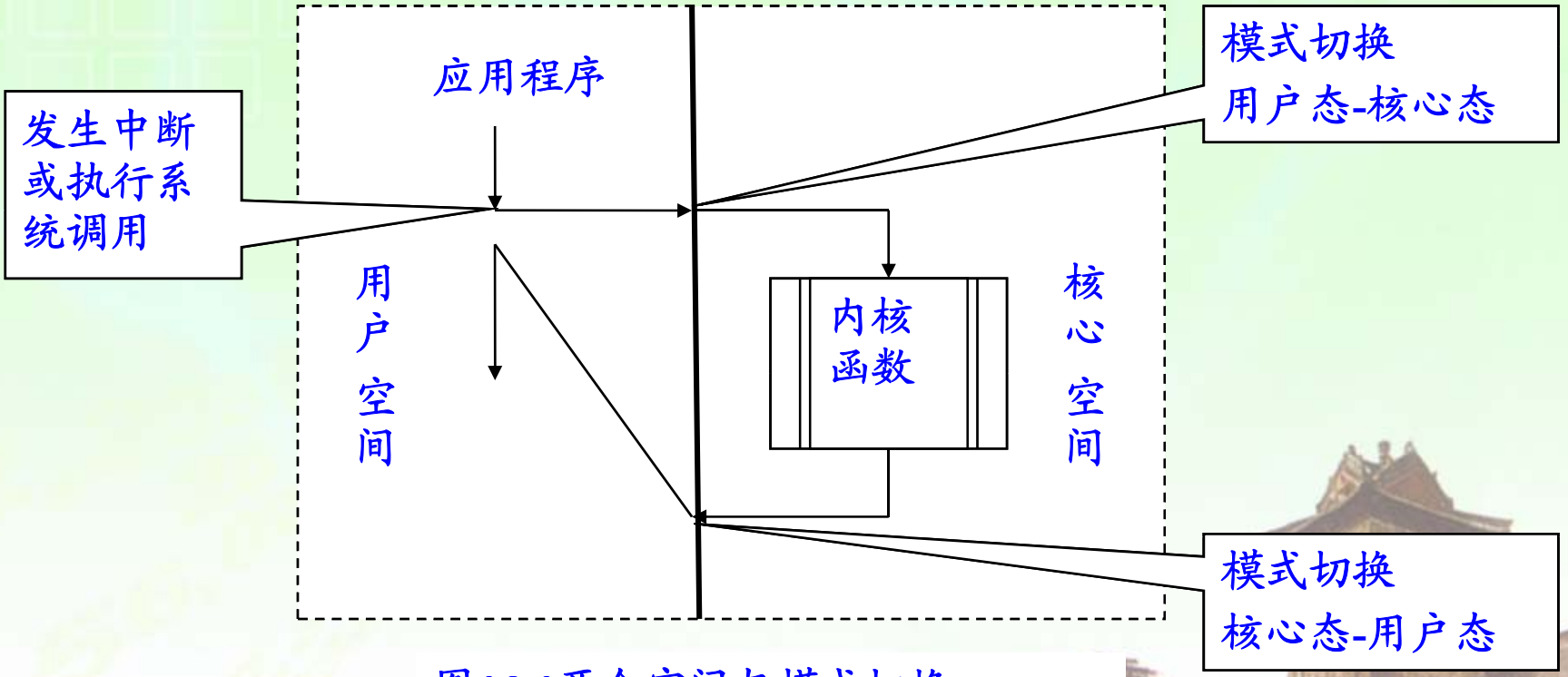
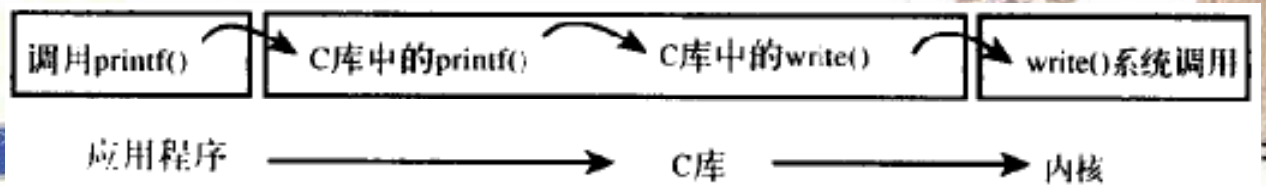


图14-1两个空间与模式切换

系统调用、应用程序接口及库函数

- 应用编程接口API
  - 只是一个函数定义
  - API与系统调用之间关系
    - ✓ 首先，API可直接提供用户态的服务(比如一些数学函数)
    - ✓ 其次，一个单独的API可调用几个系统调用
    - ✓ 不同的API可调用同一个系统调用
- 库函数
  - 实现POSIX的绝大部分API，包括标准C库函数和系统调用
  - 对系统调用的封装
    - ✓ 引用封装例程，发布系统调用
    - ✓ 一般每个系统调用对应一个封装例程
    - ✓ 通过封装例程定义出给用户的API

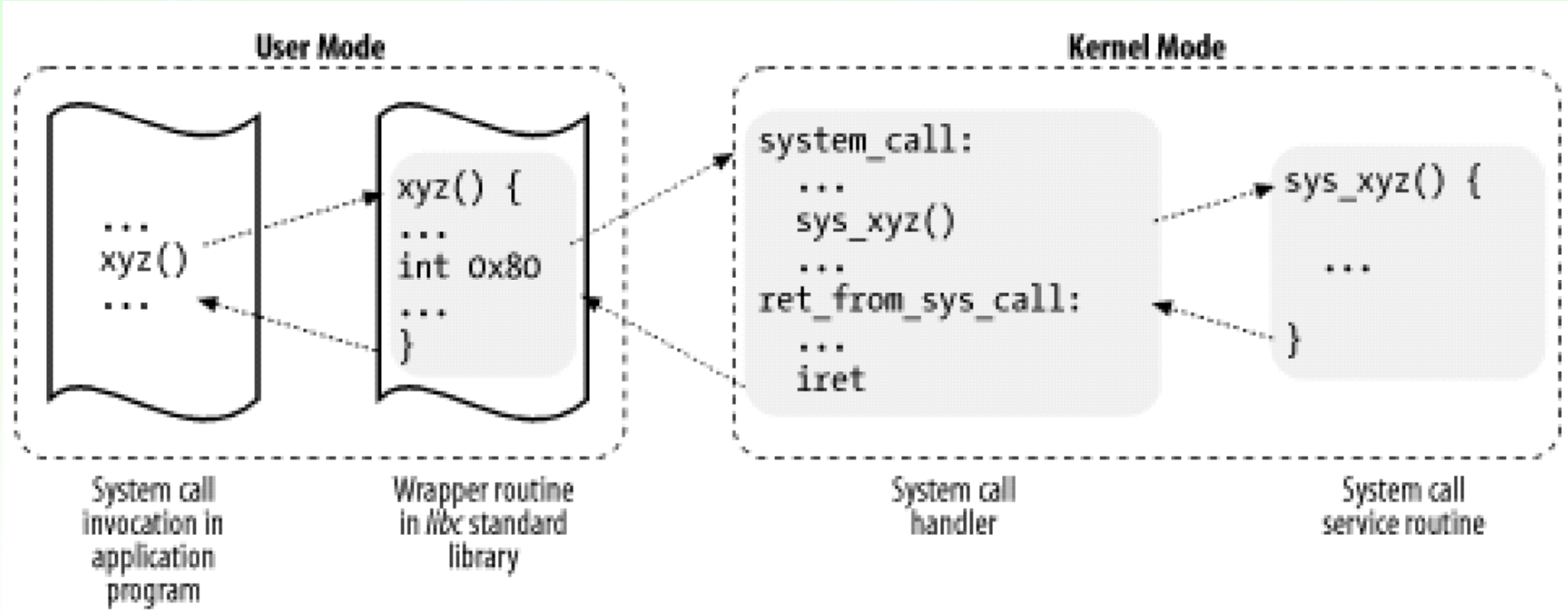


设计系统调用的基本原则

- 从数据获取途径
  - 系统调用可应用于必须获得内核数据的场景，如获得中断或系统时间等
- 从安全角度
  - 在内核中提供的服务比用户空间提供的安全性会更好，因为它不能被非法访问
- 从效率角度
  - 在核心空间实现系统服务能避免和用户空间来回传递数据，效率往往比在用户空间实现高许多

应用程序与系统调用之间的关系

- 封装例程
  - 屏蔽下层的复杂性
  - 将系统调用封装成应用程序能够直接调用的函数(库函数)

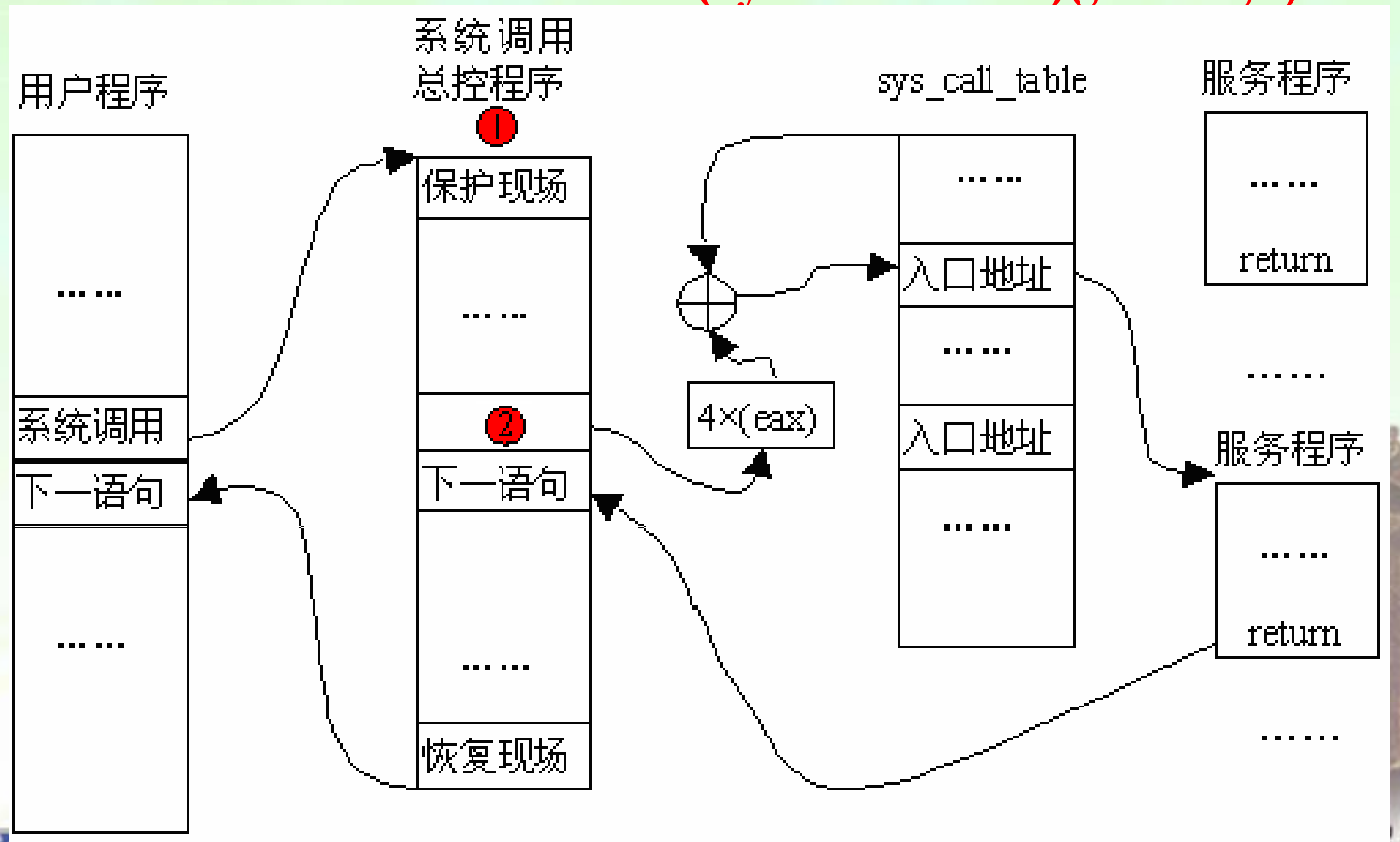


系统调用执行流程

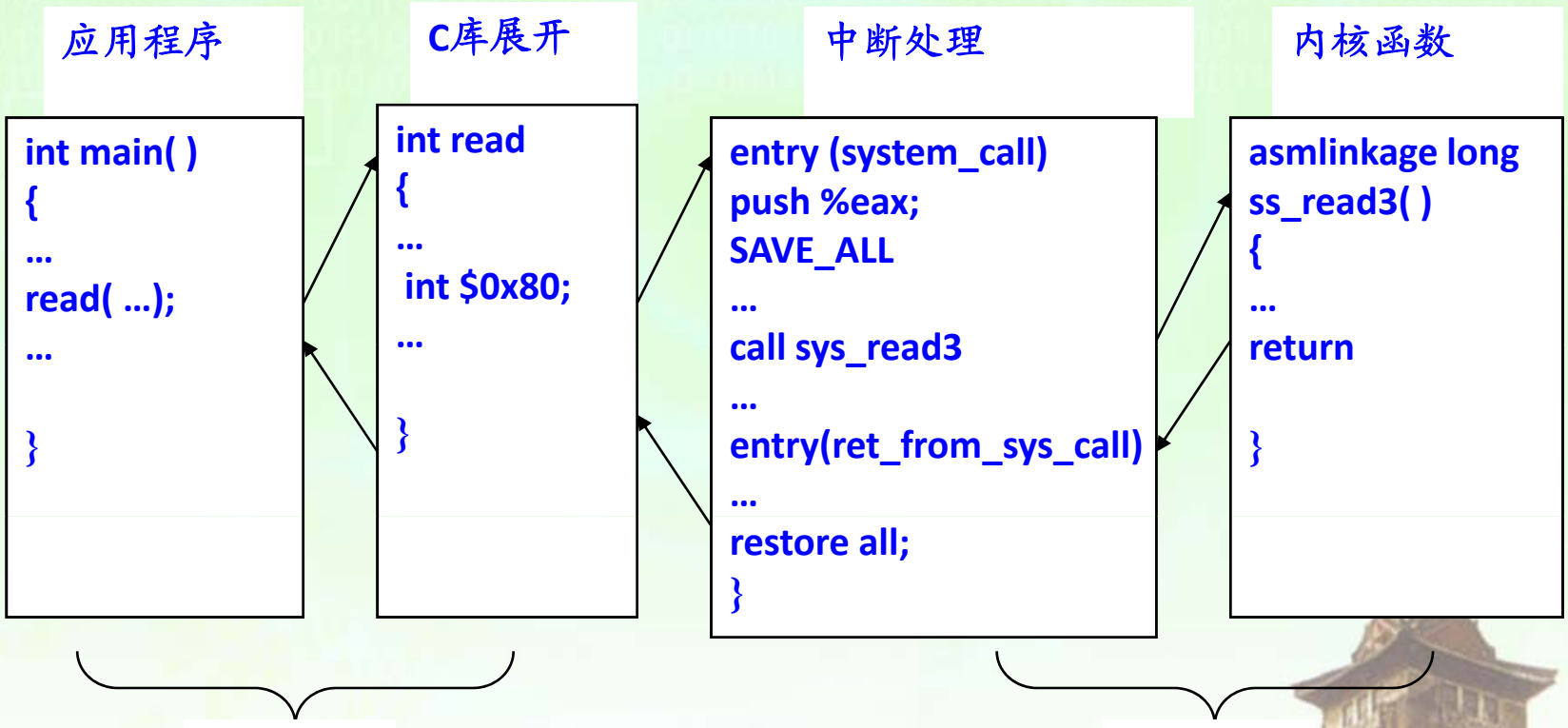
❖ 说明

- ①： 系统调用总控程序system_call
- ②： 系统服务例程

call * SYMBOL NAME(sys call table)(,%eax,4)



系统调用执行流程示例


应用程序
C库展开
中断处理
内核函数
int main( )
{
…
read(…);
…
}
int read
{
…
int $0x80;
…
}
entry (system_call)
push %eax;
SAVE_ALL
…
call sys_read3
…
entry(ret_from_sys_call)
…
restore all;
}
asmlinkage long
ss_read3( )
{
…
return
}
用户态
核心态

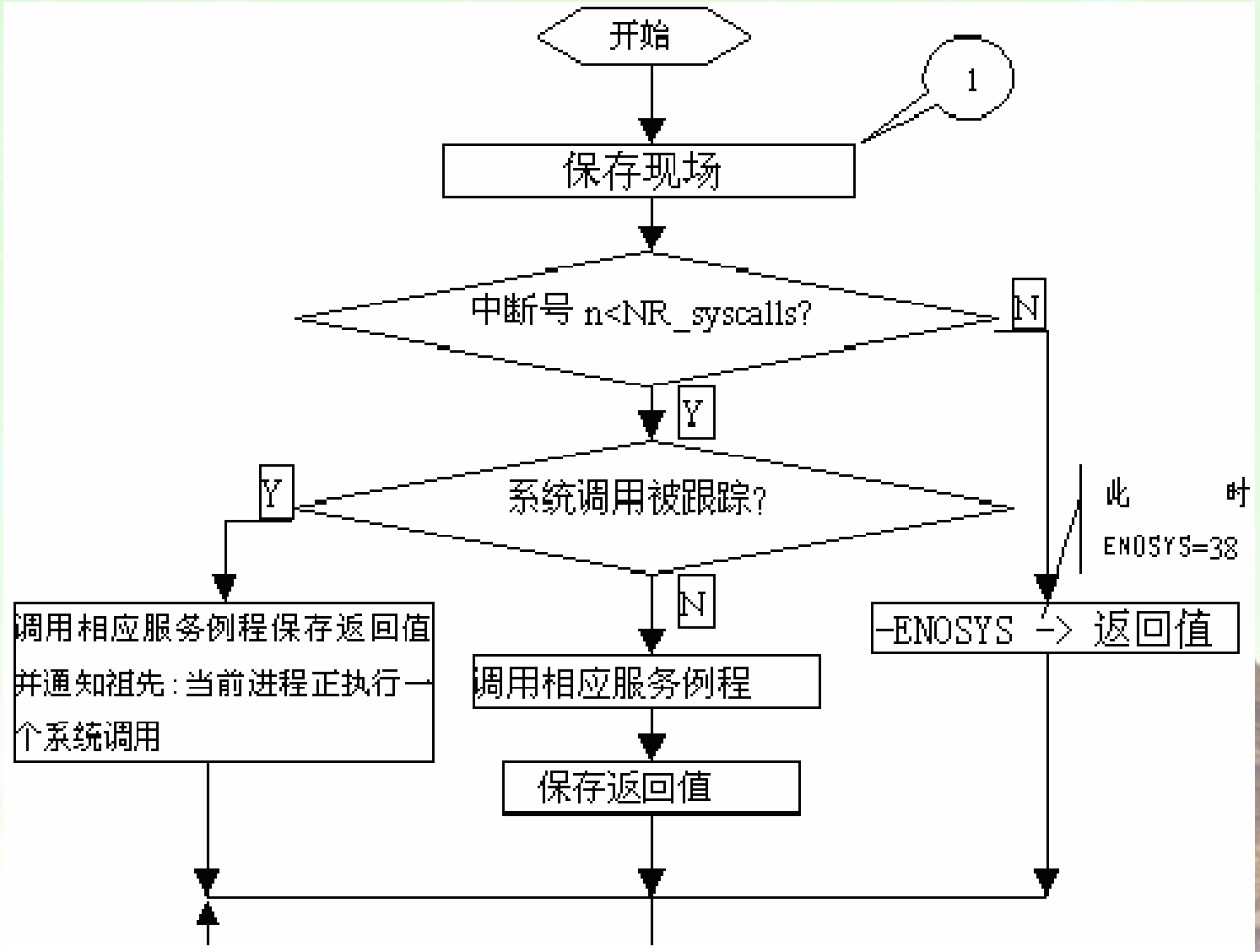
系统调用执行过程

- 设置相应的通用寄存器内容
- 调用中断指令“int $0x80”，陷入操作系统内核，调用相应的ISR（系统调用服务例程）
  - ISR 将各个寄存器按照一定顺序压栈
  - ISR 以%eax 寄存器的值为索引，到系统调用函数入口表中查找并调用
- 从Stack 中恢复系统调用前寄存器的值
- 通过iret 指令返回用户态
- 通过%eax 寄存器取得系统调用返回值

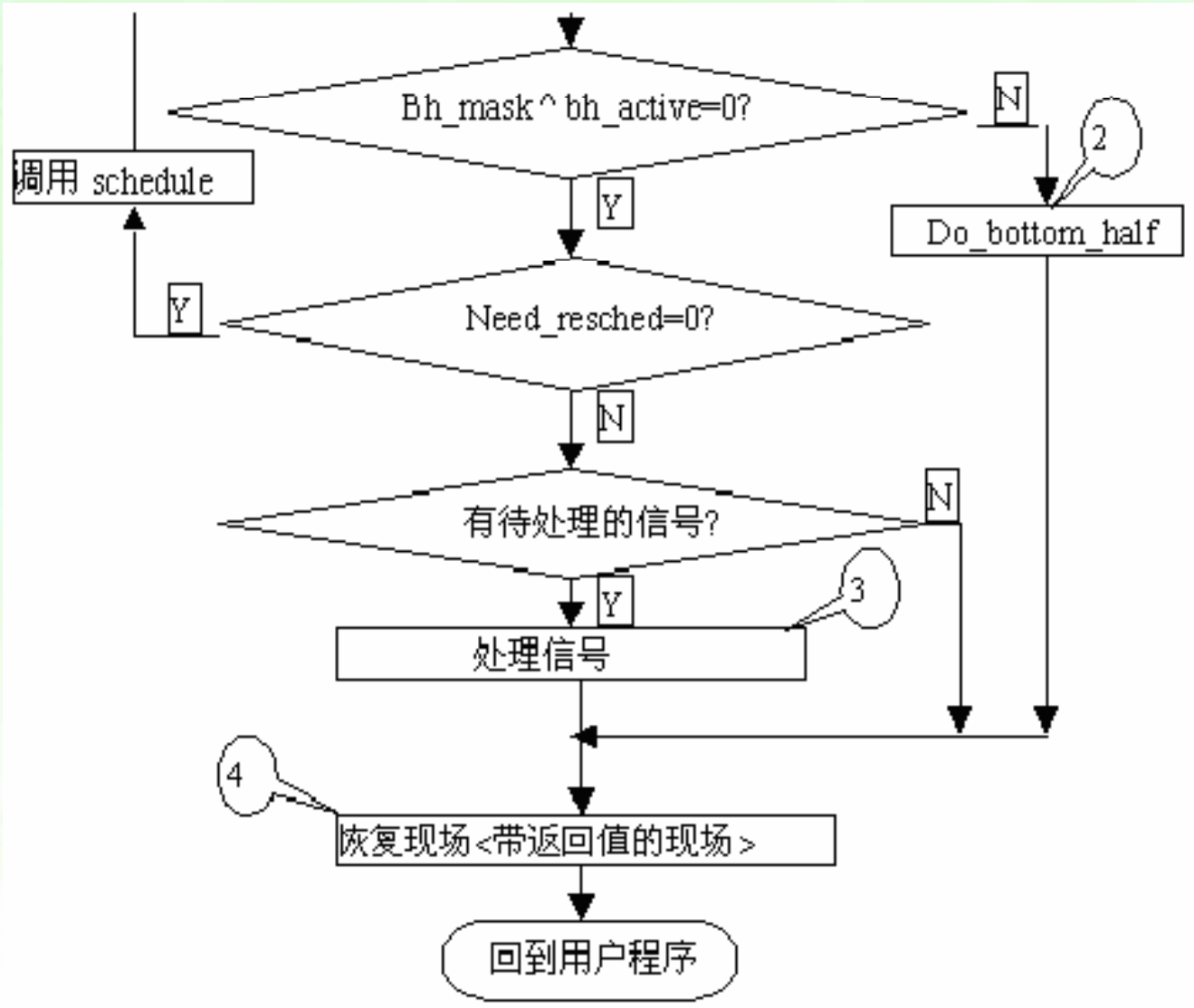
系统调用执行机理

- 用户程序调用系统调用总控程序system_call
  - 系统调用总控程序的入口地址system_call挂在中断0x80上
  - 用户程序执行 int $0x80 实现“从用户程序到系统调用总控程序”的进入
    - ✓int $0x80被封装在标准C库中，用户程序只需用标准系统调用函数实现调用
  - 处理过程
    - ✓在进程的内核态堆栈中保存大多数寄存器的内容
    - ✓调用对应系统调用服务例程处理系统调用
    - ✓通过ret_from_sys_call()从系统调用返回服务程序
  - system_call通过 “call * SYMBOL_NAME(sys_call_table)(,%eax,4)调用服务程序

system_call()执行流程



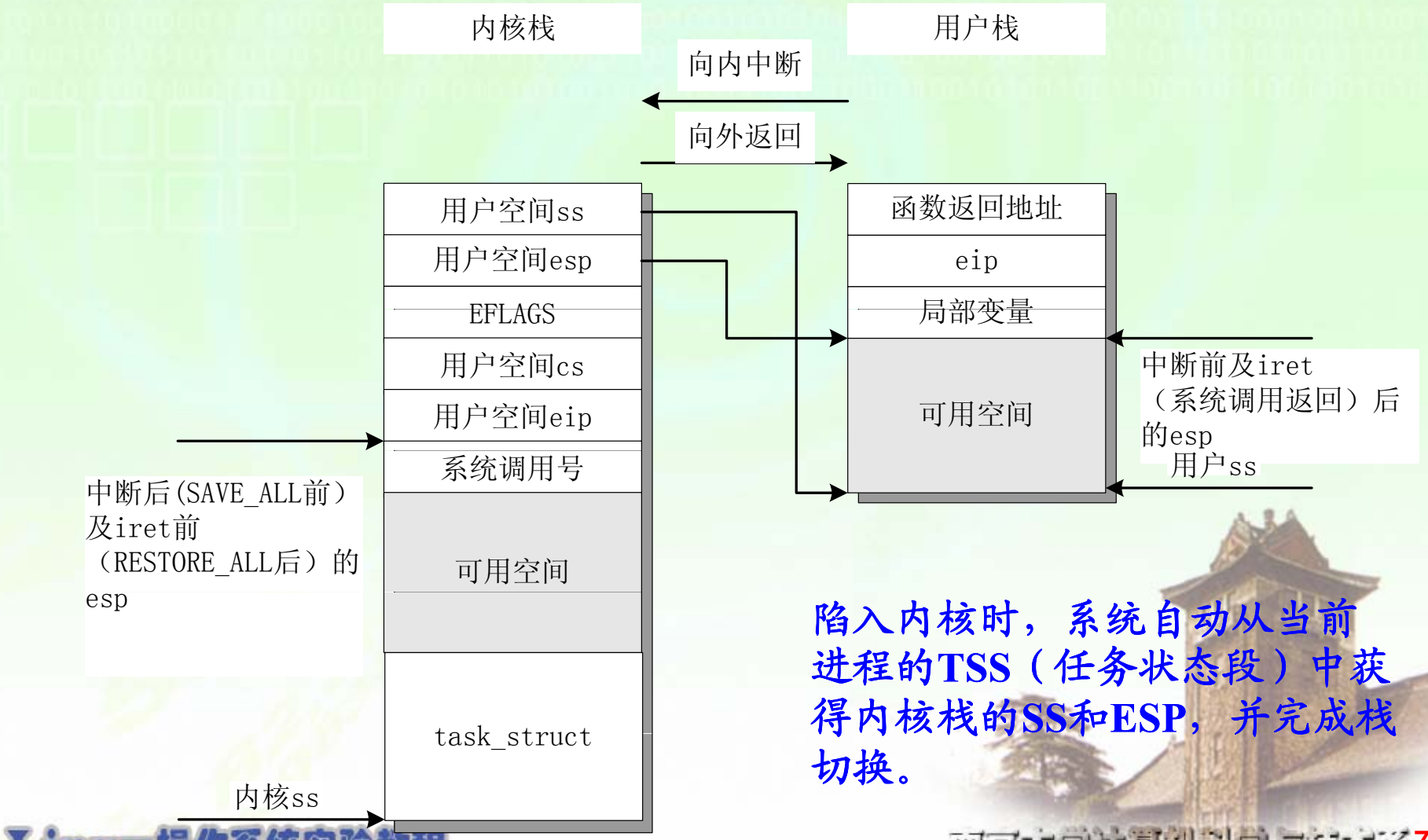
system_call()执行流程



system_call()函数伪代码

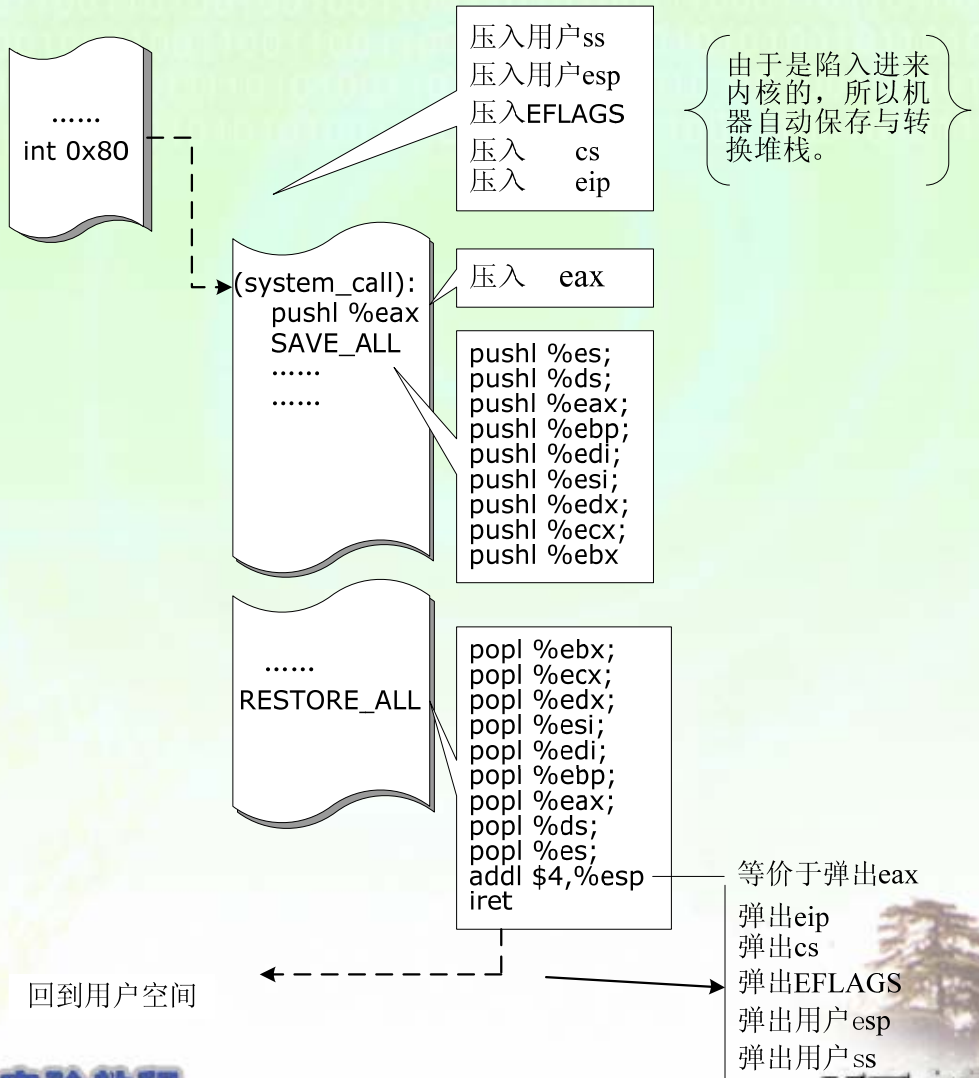
```
ENTRY(system_call)
        pushl %eax                              # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)            # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)                     # save the return value
ENTRY(ret_from_sys_call)
        cli                                     # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
```

系统调用时的内核栈



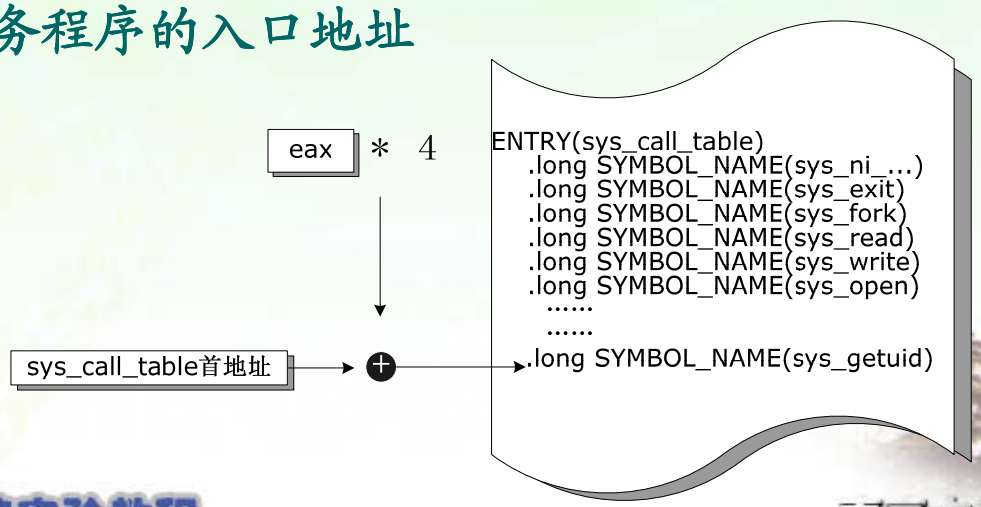
陷入内核时，系统自动从当前进程的TSS（任务状态段）中获得内核栈的SS和ESP，并完成栈切换。

SAVE_ALL()及RESTORE_ALL()



系统调用服务例程调用

- 调用形式
  - call * SYMBOL_NAME(sys_call_table)(,%eax,4)
- 说明
  - eax保存相应系统调用编号，此编号对应系统调用服务例程在系统调用向量表sys_call_table中的编号
  - 由于系统调用向量表sys_call_table每项占4个字节，所以由%eax 乘上4形成偏移地址
  - sys_call_table为基址，基址加上偏移所指向的内容就是相应系统调用服务程序的入口地址



sys_call_table

- 记录所有已注册过的系统调用列表每个有效的系统调用指定一个唯一的系统调用号
- 定义在/linux/include/asm/unistd.h中的entry.s段

```
ENTRY(sys_call_table)
        .long sys_restart_syscall /*  0  */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open            /*  5  */
         ...
        .long sys_mq_unlink
        .long sys_mq_timedsend
        .long sys_mq_timedreceive /*  280  */
        .long sys_mq_notify
        .long sys_mq_getsetattr
```

系统调用编号

- 定义位置：include/asm-i386/unistd.h
  - 有NR_syscalls个表项(通常是256)
    - ✓ 第n个表项对应系统调用号为n的服务例程的入口地址的指针

```
#define _NR_restart_syscall   0
#define _NR_exit              1
#define _NR_fork              2
#define _NR_read              3
#define _NR_write             4
#define _NR_open              5
...
#define _NR_ mq_unlink        278
#define _NR_ mq_timedsend     279
#define _NR_ mq_timedreceive  280
#define _NR_ mq_notify        281
#define _NR_ mq_getsetattr    282
```

sys_ni_syscal()系统调用

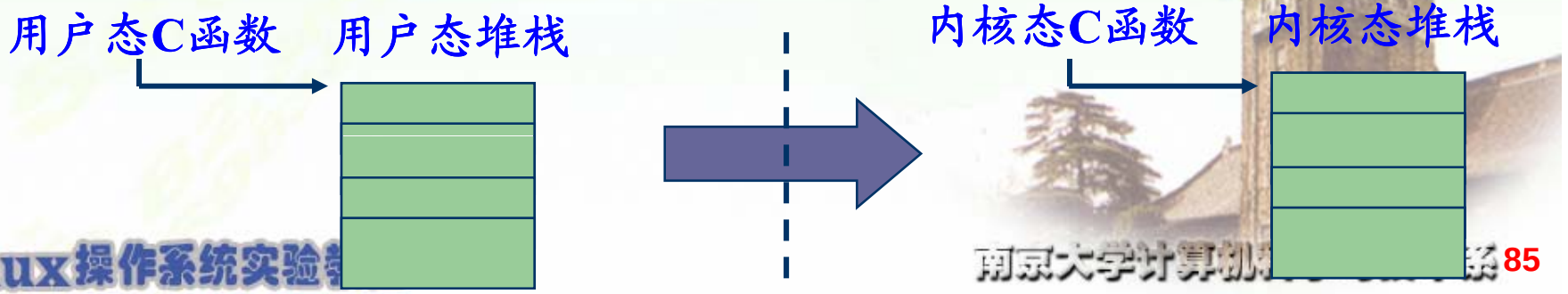
- 只返回-ENOSYS，专门针对无效的系统调用而设置
- 定义形式

```
asmlinkage int sys_ni_syscall(void)
{
 return -ENOSYS；
}
```

  - 处理边界错误，0号系统调用就是用此特殊的服务程序
  - 用来替换旧的已淘汰了的系统调用
  - 用于将要扩展的系统调用

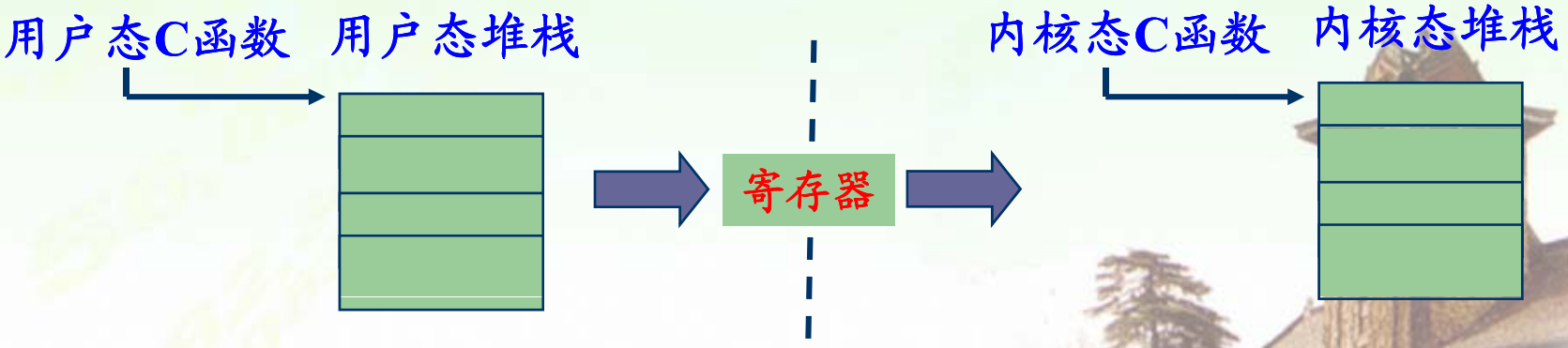
参数传递

- 系统调用也需要输入输出参数，如
  - 系统调用编号
  - 用户态进程地址空间的变量的地址
  - 甚至是包含指向用户态函数的指针的数据结构的地址
- 系统调用参数传递特点
  - 普通C函数的参数传递是通过把参数值写入堆栈(用户态堆栈或内核态堆栈)来实现的
  - 系统调用是一种特殊函数，由用户态进入内核态，所以既不能使用用户态的堆栈，也不能直接使用内核态堆栈



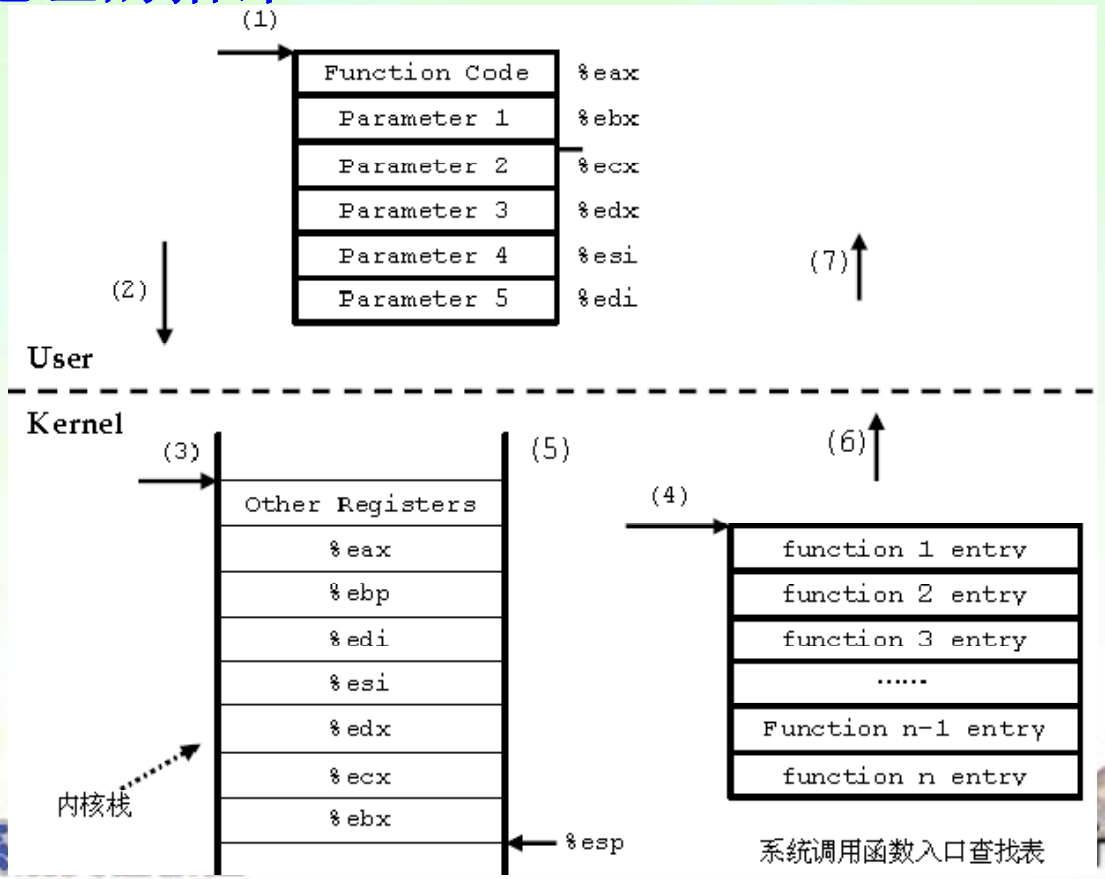
系统调用参数传递方式

- 在执行int $0x80汇编指令之前，系统调用的参数被写入CPU的寄存器
- 然后，在进入内核态调用系统调用服务例程之前，内核再把存放在CPU寄存器中的参数拷贝到内核态堆栈中
- 因为毕竟服务例程是C函数，它还是要到堆栈中去寻找参数的



x86系统中参数传递方法

- 一般支持6个参数，按次序存放在%eax，%ebx，%ecx，%edx，%esi，%edi中
- 超过6个参数需要用单独的寄存器存放指向所有参数的用户空间地址的指针



参数验证

- 在内核打算满足用户的请求之前，检查参数的合法性与正确性
  - 与I/O相关的系统调用需检查文件描述符的有效性
  - 与进程相关的函数必须检查PID的有效性比如前面的
  - 检查用户提供的指针的合法性
    - ✓ 指向的内存区域属于用户空间，不能哄骗内核去读内核空间数据
    - ✓ 指向的区域在进程的地址空间，不能哄骗内核去读其他进程的数据
    - ✓ 如果是读/写，该内存应标记为相应的读/写权限

与地址相关的参数验证

- 判断标准
  - 检查它是否在这个进程的地址空间之内
- 两种验证方法
  - 验证这个线性地址是否属于进程的地址空间
    - ✓ 费时
    - ✓ 大多数情况下，不必要
  - 仅仅验证这个线性地址小于PAGE_OFFSET
    - ✓ 高效
    - ✓ 可以在后续的执行过程中，很自然的捕获到出错的情况
    - ✓ 从linux2.2开始执行此种检查

对用户地址参数的粗略验证

- 在内核中，可以访问到所有的内存
- 要防止用户将一个内核地址作为参数传递给内核，这将导致它借用内核代码来读写任意内存
- 检查方法

```
verify_area(int type, const void * addr, unsigned long size)
```

- 最高地址：addr+size-1
- 是否超出3G边界
  - ✓对于用户进程：不大于3G
- 是否超出当前进程的地址边界
  - ✓对于内核线程：可以使用整个4G

用户空间与内核空间数据访问的函数

Function	Action
get_user _ _get_user	Reads an integer value from user space (1, 2, or 4 bytes)
put_user _ _put_user	Writes an integer value to user space (1, 2, or 4 bytes)
copy_from_user _ _copy_from_user	Copies a block of arbitrary size from user space
copy_to_user _ _copy_to_user	Copies a block of arbitrary size to user space
strncpy_from_user _ _strncpy_from_user	Copies a null-terminated string from user space
strlen_user strnlen_user	Returns the length of a null-terminated string in user space
clear_user _ _clear_user	Fills a memory area in user space with zeros

返回值

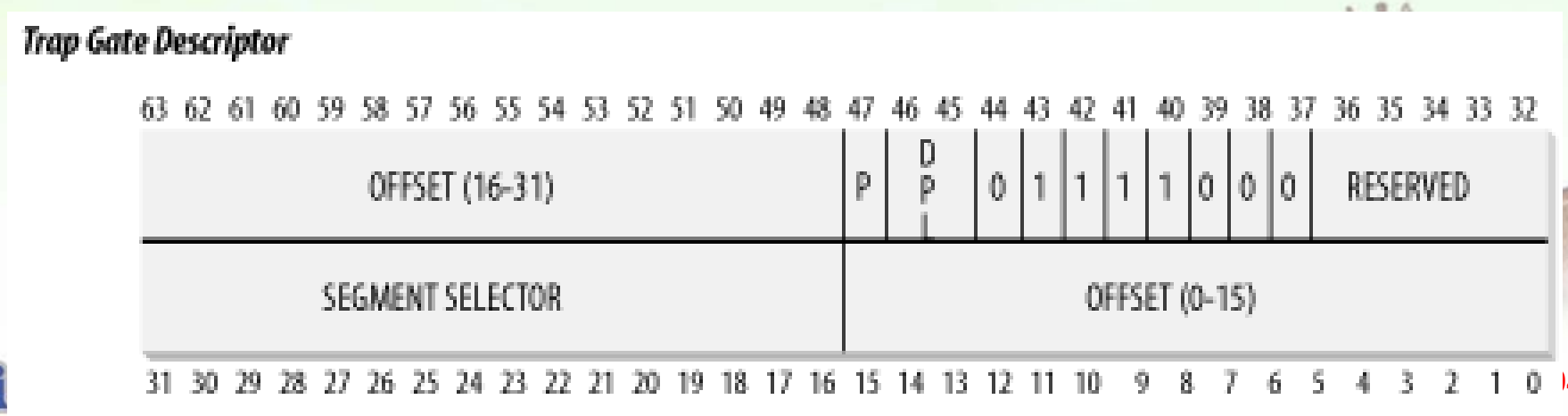
- 返回值存放在%eax中，在执行“return”指令时，由编译器自动完成的
  - 函数的返回值可为void或32-bit 类型(integer)
- 返回值与封装例程返回值的约定是不同的
  - 正数或0表示系统调用成功结束
  - 负数表示一个出错条件，此时这个负值将要存放在errno变量中返回给应用程序
  - 内核没有设置或使用errno变量，封装例程在系统调用返回取得返回值之后设置这个变量

errno变量

- 功能
  - 保存系统调用返回的错误码
  - 如果一个系统调用失败，可以读出errno的值来确定问题所在
  - errno不同数值所代表的错误消息定义在errno.h中，可通过perror()显示相应错误信息
- 说明
  - errno的值只在函数发生错误时设置，如果函数不发生错误，errno的值就无定义，并不会被置为0。
  - 在处理errno前最好先把它的值存入另一个变量，因为在错误处理过程中，即使像printf()这样的函数出错时也会改变errno的值。

系统调用初始化

- 初始化函数：trap_init()
- 调用语句：set_system_gate(0x80, $system_call);
  - 建立IDT表中向量128对应的表，将下列值存入该系统门描述符的相应字段
    - ✓ segment selector：核代码段__KERNEL_CS的段选择符
    - ✓ offset：指向system_call()异常处理程序的入口地址
    - ✓ type：置为15。表示这个异常是一个陷阱，相应的处理程序不禁止可屏蔽中断
    - ✓ DPL(描述符特权级)：置为3，允许用户态进程访问这个门，即在用户程序中使用int $0x80是合法的



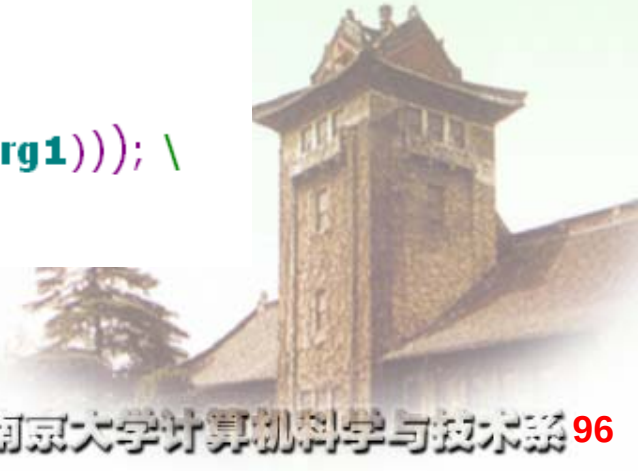
封装例程

- 系统调用可由用户态进程或内核线程使用，但内核线程是不能使用库函数的，为简化处理，linux定义了七个内核封装宏
  - _syscall0(type,name)
  - _syscall1(type,name,type1,arg1)
  - _syscall2(type,name,type1,arg1,type2,arg2)
  - _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
  - _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
  - _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)
  - _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)
- 说明
  - 形成相应的系统调用函数原型，供在程序中调用
  - 数字和typeN,argN的数目一样多
    - ✓ _syscall后面跟的数字指明展开后形成函数的参数的个数

封装例程伪代码

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_##name)); \
__syscall_return(type,__res); \
}

#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
```

. . .



封装例程举例

- write系统调用的宏代码
  - _syscall3(int,write,int, fd, const char *, buf, unsigned int, count)

```
int write(int fd,const char * buf,unsigned int count)
{
    long __res;
    asm("int $0x80"
        : "=a" (__res)
        : "0" (__NR_write),"b" ((long)fd),
          "c" ((long)buf),"d" ((long)count));
    if ((unsigned long)__res >= (unsigned long)-125) {
        errno = -__res;
        __res = -1;
    }
    return (int) __res;
}
```

封装例程举例

❖write系统调用编译后生成的汇编代码

```
write:
      pushl %ebx                ; push ebx into stack
      movl 8(%esp), %ebx        ; put first parameter in ebx
      movl 12(%esp), %ecx       ; put second parameter in ecx
      movl 16(%esp), %edx       ; put third parameter in edx
      movl $4, %eax             ; put __NR_write in eax
      int $0x80                 ; invoke system call
      cmpl $-126, %eax          ; check return code
      jbe .L1                   ; if no error, jump
      negl %eax                 ; complement the value of eax
      movl %eax, errno          ; put result in errno
      movl $-1, %eax            ; set eax to -1
.L1:  popl %ebx                 ; pop ebx from stack
      ret                       ; return to calling program
```

常见系统调用的封装例程

```
static inline _syscall0(int,pause)
static inline _syscall0(int,sync)
static inline _syscall0(pid_t,setsid)
static inline _syscall3(int,write,int,fd,const char *,buf,off_t,count)
static inline _syscall3(int,read,int,fd,char *,buf,off_t,count)
static inline _syscall3(off_t,lseek,int,fd,off_t,offset,int,count)
static inline _syscall1(int,dup,int,fd)
static inline _syscall3(int,execve,const char *,file,char **,argv,char **,envp)
static inline _syscall3(int,open,const char *,file,int,flag,int,mode)
static inline _syscall1(int,close,int,fd)
static inline _syscall1(int,_exit,int,exitcode)
static inline _syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)
static inline _syscall1(int,delete_module,const char *,name)
```

系统调用添加步骤

- 基本步骤
  - 确定功能、形态
    - ✓必须功能明确单一、不提倡多用途系统调用
    - ✓确定参数、返回值及错误码
  - 在系统调用表中添加一个表项
    - ✓位于entry.s的ENTRY（sys_call_table）
  - 将系统调用号定义到<asm/unistd.h>中
  - 编译到内核映象（不能编译成模块）
    - ✓把实现代码放入kernel/下的一个相关文件
  - 重新编译内核，启动新内核
  - 封装系统调用例程，支持用户空间的访问

添加一个系统调用

- 系统调用名
  - mysyscall
- 功能
  - 调用这个mysyscall，使用户的uid等于0

添加一个系统调用

- 内核中实现该系统调用的程序的名字sys_mysyscall
  - 改写arch/i386/kernel/entry.S
  - 系统调用号为236（原因？）

```
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)
            ……
        .long SYMBOL_NAME(sys_ni_syscall)  /*235*/
        .long SYMBOL_NAME(sys_mysyscall)
        .rept NR_syscalls-(.-sys_call_table)/4
        .long SYMBOL_NAME(sys_ni_syscall)
        .endr
```

添加一个系统调用

- 在系统调用表中添加一个表项
  - 改写/usr/include/asm/unistd.h
  - 系统调用的编号名字 __NR_mysyscall

```
#define __NR_llistxattr        233
#define __NR_flistxattr        234
#define __NR_removexattr       235
#define __NR_mysyscall         236
#define __NR_fremovexattr      237
```

添加一个系统调用

❖把一小段程序添加在kernel/sys.c

```
asmlinkage int sys_mysyscall(void)
{
  current->uid = current->euid = current->suid =
  current->fsuid = 0;
        return 0;
}
```

asmlinkage使得编译器不通过寄存器(x=0)而
使用堆栈传递参数
所有系统调用都使用这个限定词

添加一个系统调用

- 编写一段测试程序检验实验结果

```
#include <linux/unistd.h>
_syscall0(int,mysyscall)        /* 注意这里没有分号 */
int main()
{
    mysyscall();
    printf("em…, this is my uid: %d. \n", getuid());
}
```

主要内容

- 背景知识
  - 中断机制
  - 系统调用
- 实验内容
  - 记录系统调用的使用次数

记录系统调用的使用次数

❖实验说明

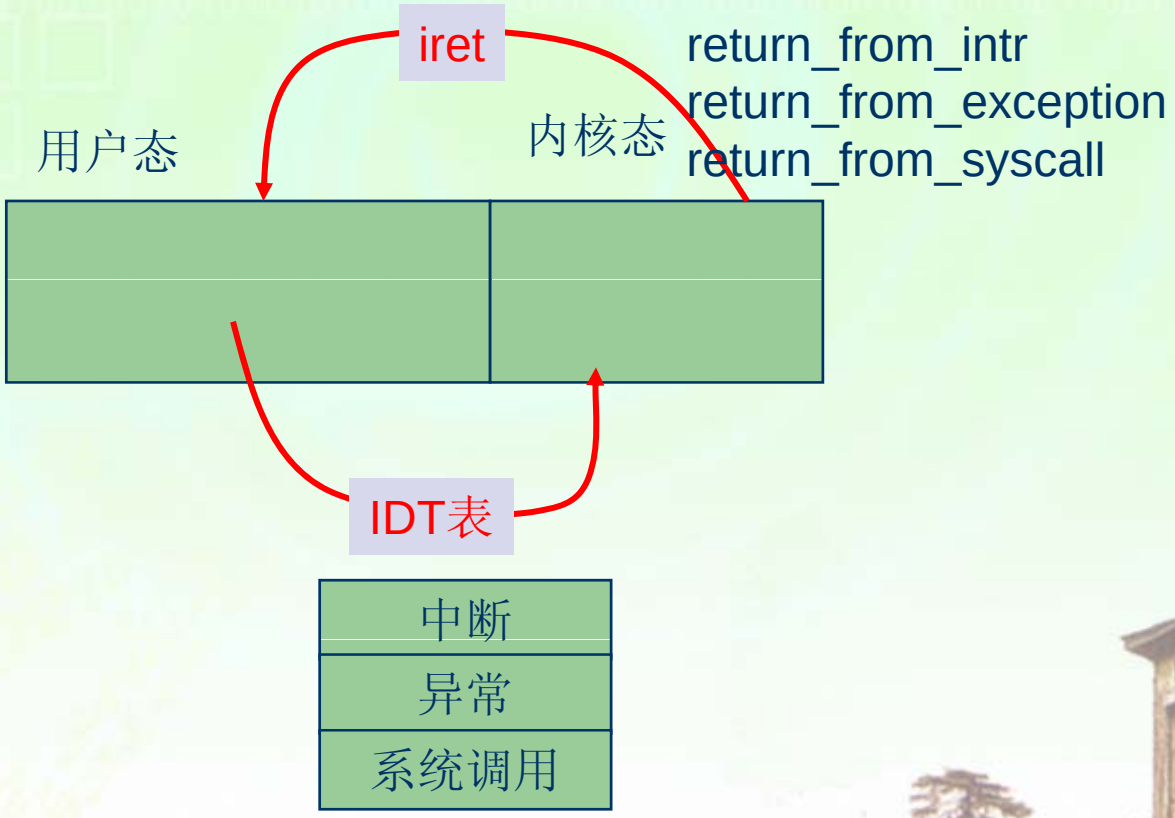
- 通过修改system_call( )，使得内核能够记录每个系统调用被使用的次数
- 为使应用进程能够查询到这些数据，本实验要求提供一个系统调用，供应用进程查询某个特定系统调用被使用的次数

记录系统调用的使用次数

❖解决方案：处理步骤

- 编写系统调用的内核函数，该内核函数在内核中的标准名称应该是在函数名前面加上“sys_”标志
- 连接新的系统调用
  - ✓先将新系统调用的源代码添加到／usr／Linux／kernel／sys.c文件中
  - ✓再通知内核的其余部分：内核增加了新的系统调用的内核函数
  - ✓需要重新编辑／usr／src／Linux／include／asm／unistd.h和／usr／src／Linux／arch／asm／kernel／entry.S两个文件
- 编译新的Linux内核，获得内核映象文件bzImage。
- 启动新内核的操作系统
- 尝试使用新系统调用：用Linux提供的预处理宏指令“syscall n( )”对该系统调用进行封装

中断与系统调用小结


iret
return_from_intr
return_from_exception
return_from_syscall
用户态
内核态
IDT表
中断
异常
系统调用

第13章 中断与系统调用