東北大學秦皇島分校
Northeastern University at Qinhuangdao

Linux系统与内核分析

--中断和异常

于七龙

目录

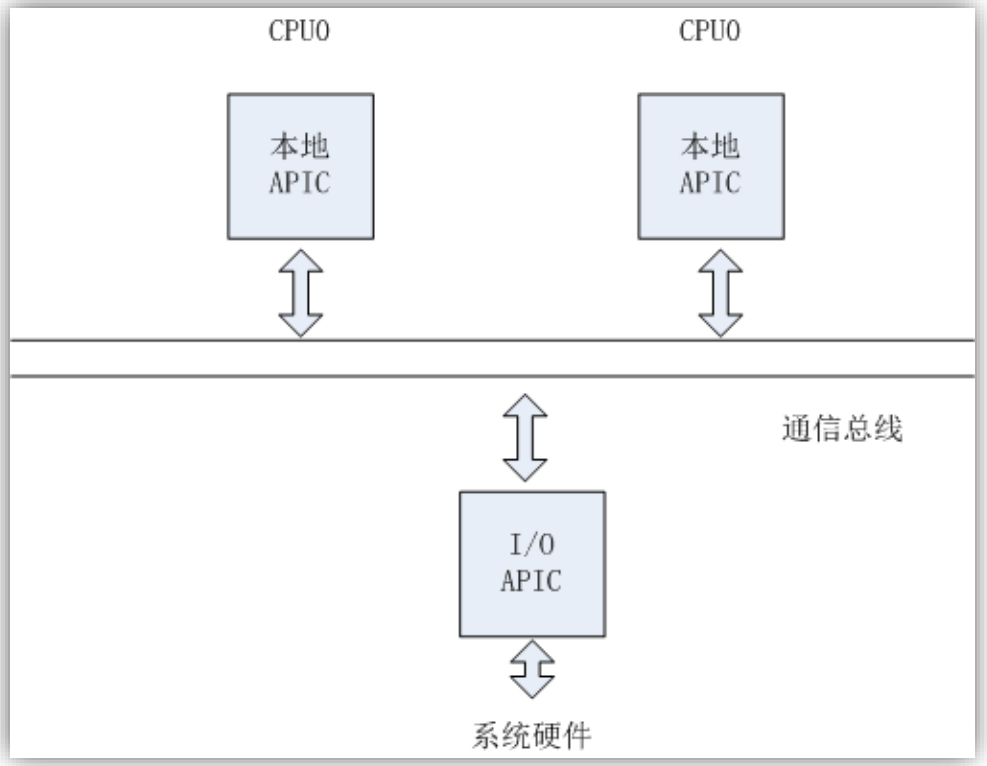
Part 1 中断概述

中断

- 中断指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行
- 中断控制的主要优点是只有在需要中断服务（如I/O服务）时才能得到处理器的响应，而不需要处理器不断地进行查询
  - 中断技术发明之前，系统通过轮询实现

中断的硬件

- 中断响应由中断控制器控制
- APIC：高级可编程中断控制器
  - (Advanced Programmable Interrupt Controller)
  - 每个处理器都有一个本地APIC
  - I/O APIC可选
  - 若I/O没有APIC，则CPU只能核间中断



中断控制器

中断线

- 中断线是与中断控制器相连的通信线，代表中断的来源
  - 中断线数量由中断控制器芯片决定
  - 中断线所对应的编号称为硬中断号
  - 中断线需要申请（IRQ）
    - 计算机外部设备远多于中断线数量，所以只有当设备需要中断的时候才申请占用一个IRQ
    - 中断线可以共享
- 中断处理程序(Interrupt Handler) 存在于中断线中
- 中断服务程序(Interrupt Service Routine) 存在于申请中断线的设备中

中断线

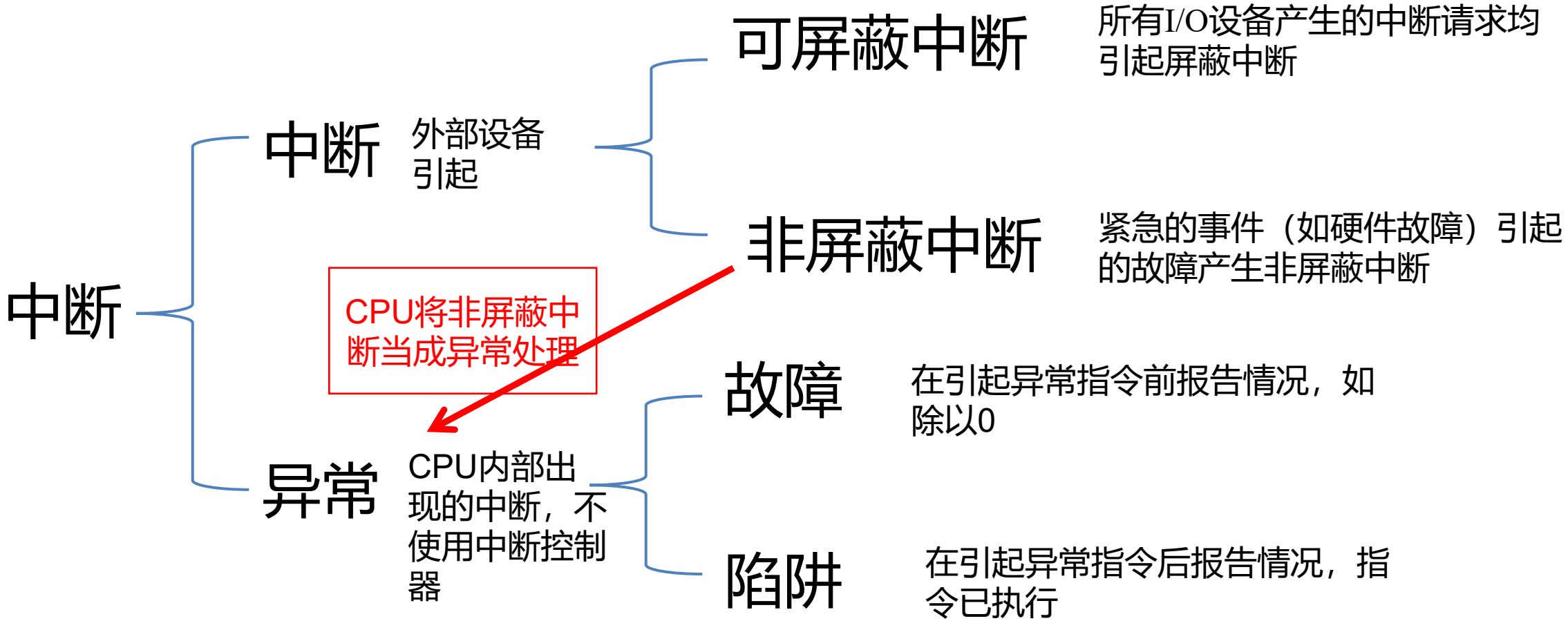
- 中断线数据结构

```
struct irqaction {
  irq_handler_t handler;          //指向具体的中断处理程序
  unsigned long flags;            //标志中断线与中断设备间的关系（如是否可以共享）
  const char *name;               //中断设备名
  void *dev_id ;
  struct irqaction *next;               //有条件的链表
  ……
 };
```

中断向量

- 中断源的编号，X86系列微机支持256种向量编号（0-255），也叫**软中断号**
- Linux对256个中断向量分配如下
  - 中断号定义：(arch\x86\include\asm\irq_vectors.h)
  - 0-31：系统陷阱和异常（一般为非屏蔽中断）
  - 32-127：设备中断（由I/O设备引起的中断，一般为屏蔽中断）
  - 128：即0x80，系统调用
  - 129-INVALIDATE_TLB_VECTOR_START-1（204除外）：设备中断
  - INVALIDATE_TLB_VECTOR_START -255：特殊中断
- Linux中断向量：/proc/interrupts

中断的分类


中断
中断
外部设备引起
可屏蔽中断
所有I/O设备产生的中断请求均引起屏蔽中断
非屏蔽中断
紧急的事件（如硬件故障）引起的故障产生非屏蔽中断
CPU将非屏蔽中断当成异常处理
异常
CPU内部出现的中断，不使用中断控制器
故障
在引起异常指令前报告情况，如除以0
陷阱
在引起异常指令后报告情况，指令已执行

中断描述符表(IDT)

- Linux定义了**中断描述符表**(Interrupt Descriptor Table, IDT)存储**中断向量**与对应**中断处理程序**的入口地址
- x86架构中CPU专门设置了中断描述符表寄存器IDTR
- IDT中每个表项称为门描述符（Gate Descriptor）
- 门是在CPU进行状态切换时提供一种审查机制，也是相应处理程序的入口
  - 中断门（Interrupt Gate）
  - 陷阱门（Trap Gate）
  - 系统门（System Gate）

中断门

- 类型码：110
- 包含一个中断或异常处理程序的虚拟地址
- 中断门的请求特权级（DPL）为0
- 当控制权通过中断门进入中断处理程序时，处理器IF标识位设为0（IF为CPU中断标志位），即关中断，以避免嵌套中断

陷阱门

- 类型码：111
- DPL为0
- 与中断门相似，只是控制权通过陷阱门后IF标志位不变，即不关中断

系统门

- Linux特有门，用于用户态进程进入内核态的陷阱门
- DPL为3
- 如系统调用通过系统门进入内核

中断相关汇编指令

- 调用过程指令CALL
  - 格式：CALL 过程名
  - 取出CALL指令之后及执行CALL指令之前，使指令指针寄存器EIP指向紧接CALL指令的下一条指令
  - CALL指令先将EIP值压入栈内，在进行控制转移
  - 当遇到RET指令时，栈内信息可使控制权直接回到CALL指令的下一条指令

中断相关汇编指令

- 调用中断过程的指令INT
  - 格式：INT 中断向量
  - EFLAG、CS、EIP等寄存器被压入栈内，控制权被转移到中断向量指定的中断处理程序
  - 中断处理程序结束时，IRET指令把控制权送回被执行中断的地方

中断相关汇编指令

- 中断返回指令IRET
  - 格式：IRET
  - IRET与中断调用过程相反，它将EIP、CS、EFLAGS寄存器内容出栈，并将控制权返回发生中断的地方
  - IRET用于中断处理程序的结束处

中断相关汇编指令

- 加载中断描述符表的指令LIDT
  - 格式：LIDT 48位的伪描述符
  - LIDT将指令中给定的48位伪描述符装入中断描述符寄存器IDTR

中断

- 中断从中断线中来，中断线数量由中断控制器决定；中断向量用于标识不同中断类型，中断线与中断向量是一对多的关系，中断向量通过申请占用中断线；中断描述符表存储中断向量与中断处理程序的入口地址，且由门提供审查机制

Part 2 中断描述符表的初始化

中断描述符表的初始化

- Linux内核在系统初始化阶段需进行大量初始化工作，其中与中断相关的工作有

1. 初始化中断控制器
2. 将中断描述符表的起始地址装入中断寄存器
3. 初始化中断描述符表的每一项

中断描述符表的初始化

- 用户进程通过INT指令发出一个中断请求，其中断向量在0-255之间
- 为了防止用户使用INT指令模拟非法的中断和异常，必须对中断描述符表进行谨慎的初始化
  - 1、将中断门或陷阱门的请求特权DPL域设置为0，如果用户进程发出中断请求，CPU会检查出其当前特权级CPL(3)域所请求的特权级DPL(0)冲突，并拒绝中断
  - 2、针对必须让用户进程使用内核功能（如系统调用）时，即从用户态进入内核态，则通过系统门达到

中断描述符表的初始化

- 中断描述符表的初始化分为两个阶段
  - 内核引导阶段
    - 为IDT分配2K大小的空间（256个中断向量，每个门描述符4字节）
    - 初始化IDT默认值（中断处理程序填充为空）
    - 存储IDT的起始地址到IDTR
  - 内核初始化阶段
    - trap_init：对保留中断向量进行初始化
    - init_IRQ：对其余中断向量的初始化

中断描述符表的初始化

- 内核引导阶段完成中断描述符表寄存器(IDTR)的初始化
  - arch/x86/kenel/head.s

```
#arch/x86/kenel/head.s

call  setup_idt       //初始化IDT
……
lidt idt_descr        //存储IDT的起始地址到IDTR寄存器中
```

中断描述符表的初始化

- 内核初始化阶段中用实际的陷阱和中断处理程序替换第一阶段中空的处理程序
  - trap_init()完成对保留中断向量的初始化

```
#arch/x86/kenel/traps.c

void __init trap_init(void)
{
        ……
        set_intr_gate(X86_TRAP_DE, divide_error);
        set_system_intr_gate(X86_TRAP_OF, &overflow);
        set_intr_gate(X86_TRAP_BR, bounds);
        ……
        set_task_gate(X86_TRAP_DF, GDT_ENTRY_DOUBLEFAULT_TSS);
        set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
        ……
}
```

中断描述符表的初始化

- 内核初始化阶段中用实际的陷阱和中断处理程序替换第一阶段中空的处理程序
  - init_IRQ完成对其余中断向量的初始化
    - NR_IRQS = NR_VECTORS – FIRST_EXTERNAL_VECTOR
    - NR_VECTORS:256，总的中断向量数
    - FIRST_EXTERNAL_VECTOR:32(0-31)，保留向量数

```
//对NR_IRQS个中断向量进行初始化
for (i = 0; i< NR_IRQS; i++)
{
        int vector = FIRST_EXTERNAL_VECTOR + i;
       //跳过系统调用的中断向量：0x80
       if (vector != SYSCALL_VECTOR)
              set_intr_gate(vector, interrupt[i]);
}
```

中断处理程序的入口地址是一个数组interrupt[]

中断处理程序的形成

- Linux内核对中断处理程序提供了统一的方式形成函数名和函数体
  - 不同的中断处理程序，不仅名字不同，其内容也不同，但此类函数又有很多相同之处
- 中断处理程序的入口地址设置为数组interrupt[]
  - 数组中每个元素为指向中断处理程序的指针

中断处理程序的形成

- 内核对interrupt[]数组的定义如下

```
static void (*interrupt[NR_VECTORS - FIRST_EXTERNAL_VECTOR])(void) = {
        IRQLIST_16(0x2), IRQLIST_16(0x3),
        IRQLIST_16(0x4), IRQLIST_16(0x5), IRQLIST_16(0x6), IRQLIST_16(0x7),
        IRQLIST_16(0x8), IRQLIST_16(0x9), IRQLIST_16(0xa), IRQLIST_16(0xb),
        IRQLIST_16(0xc), IRQLIST_16(0xd), IRQLIST_16(0xe), IRQLIST_16(0xf)
    };
```

  - 数组中每个元素为指向中断处理程序的指针

```
#define IRQLIST_16(x) \
    IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
    IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
    IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
    IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
```

```
#define IRQ(x,y)    IRQ# #x# #y#    #_interrupt
```

```
#define IRQ_NAME(nr)    IRQ_NAME2(IRQ# #nr)
#define IRQ_NAME2(nr)    nr# #_interrupt(void)
```

中断处理程序的形成

- Linux内核对中断处理程序形成流程
  - 数组interrupt[]包含了IRQLIST_16(0x2)-IRQLIST_16(0xf)共14个数组元素
  - IRQLIST_16()通过宏定义了16个IRQ(x,y)，于是形成224(14*16)个函数指针
  - 展开IRQ(x,y)宏，如x=0x2,y=0xf，则得到IRQ0x2f_interrupt
  - 进一步通过IRQ_NAME、IRQ_NAME2宏定义得到函数名
    - 函数名范围是IRQ0x20_interrupt(void)-IRQ0xff_interrupt(void)

Part 3 中断处理

中断处理

- 对于外部中断，需建立中断请求队列，以及执行中断处理程序
- 当CPU执行了当前指令之后，CS和EIP这对寄存器中所包含的内容就是下一条将要执行指令的虚地址，在对下一条指令执行前，CPU先要判断在执行当前指令的过程中是否发生了中断或异常

中断和异常的硬件处理

- 在CPU执行下一条指令之前，若发生异常，CPU则按以下流程进行

1. 确定所发生中断或异常的向量i（在0～255之间）
2. 通过IDTR寄存器找到IDT，读取IDT第i项（或叫第i个门）
3. 按“段”、“门”两级分步进行有效性检查
   - “段”级检查：将CPU的当前特权级CPL（存放与CS寄存器的最低两位）与IDT中第i项段选择符中的DPL比较。若CPL(3)大于DPL(0)，则产生“通用保护”异常，因为中断处理程序的特权级不能低于引起中断的进程的特权级。这种情况发生的可能性不大，因为中断处理程序一般运行在内核态，其特权级为0
   - “门”级检查；把CPL与IDT中第i个门的DPL相比较。若CPL大于DPL，即当前特权级(3)小于门的特权级(0)，CPU就不能“穿过”门，则产生“通用保护”异常，这是为了避免用户应用程序访问特殊的陷阱门或中断门。但这种“门”级检查是针对一般的用户程序，而不包括外部I/O产生的中断或因CPU内部异常而产生的异常，也就是说，如果产生了中断或异常，就免去了“门”级检查
4. 检查是否发生了特权级的变化，若变化，更换堆栈

中断请求队列的建立

- 由于硬件条件的限制，很多硬件设备共享一条中断线；为方便处理，Linux为每条中断线设置了一个中断请求队列

```
#include/linux/interrupt.h

struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);  //指向具体I/O设备中的中断处理程序
  unsigned long flags;              //标志中断线与中断设备间的关系(如是否可共享、关中断等)
  const char *name;          // I/O设备名
  void *dev_id ;                          // 设备ID
  struct irqaction *next;                 //共享中断线的设备链表，前提是中断线可共享
... ...
};
```

注册中断服务程序

- 初始化IDT表之后，必须通过 request_irq() 函数将相应的中断服务程序挂入中断请求队列，即对其进行注册

```
int request_irq(
    unsigned int irq,                    //要分配的中断号
    void (*handler)(int, void *, struct pt_regs *),    //实际中断服务程序的函数指针
    unsigned long irqflags,              //与struct irqaction中irqflags相同
    const char * devname,                // I/O设备名
    void *dev_id                         //设备ID，无需共享 设为null
)
```

注册中断服务程序

- 在驱动程序初始化或设备第一次使用时，首先要调用request_irq()函数，以申请使用参数中指明的中断号irq，以及注册要挂入到中断请求队列中的中断服务例程
  - 假定一个程序要对 / dev/fd0/（第一个软盘对应的设备）设备进行访问，通常将IRQ6分配给软盘控制器，给定这个中断号6，软盘驱动程序就可发出下列请求，以将其中断服务例程挂入中断请求队列

```
request_irq(
        6,
        floppy_interrupt,
        IRQF_DISABLED | SA_SAMPLE_RANDOM,        //中断执行时禁止中断，允许根据中断发生时间产生随机数
        "floppy",
        NULL
);
```

注销中断服务程序

- 关闭设备或卸载驱动程序时，需调用free_irq()函数注销相应的中断处理服务程序，并释放中断线
  - 如果中断线不共享，则函数删除处理程序的同时将禁用该中断线
  - 如果中断线共享，则仅删除dev_id所对应的服务程序，直到删除了最后一个服务程序时才禁用该中断线

```
void free_irq(unsigned int irq，void*dev_id)
如:
free_irq(6，NULL)
```

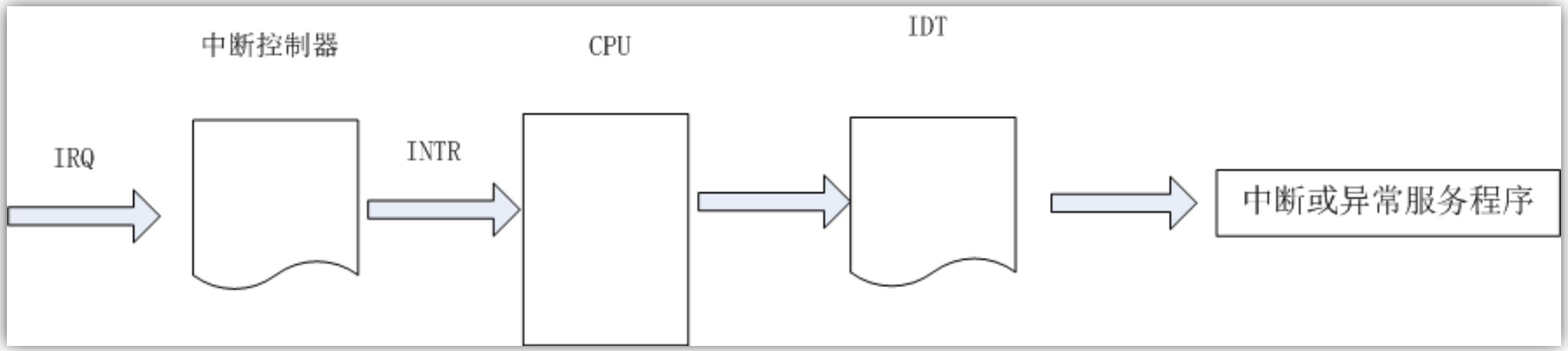
中断处理程序执行流程

- 中断处理程序执行流程

1. CPU从中断控制器的一个端口取得中断向量i
2. 根据i从中断描述符表IDT中找到相应的中断门
3. 从中断门获得中断处理程序的入口地址IRQn_interrupt
4. 判断是否要进行堆栈切换
5. 调用do_IRQ()对所接收的中断进行应答，并禁止这条中断线
6. 调用handle_IRQ_event()处理中断中断线上的所有中断服务例程
7. 当处理所有外设中断请求的函数do_IRQ()执行时，内核栈顶包含的就是do_IRQ()的返回地址，这个地址指向ret_from_intr
8. 从中断返回时，CPU要调用恢复中断现场的宏RESTORE_ALL，彻底从中断返回

中断处理程序执行流程

- 中断处理程序执行流程



- 获得中断处理程序的入口

中断处理程序执行流程

- 中断处理程序执行流程

- 获得中断处理程序的入口

中断处理程序的执行

- IRQn_interrupt

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

- common_interrupt

```
common_interrupt:
        SAVE_ALL
        TRACE_IRQS_OFF
        movl %esp, %eax
        call do_IRQ
        jmp_ret_from_intr
```

中断处理程序的执行

- do_IRQ()
  - do_IRQ() - handle_irq() - handle_IRQ_event()函数循环调用请求队列中的中断服务程序

```
irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action) {
        ......
        do {
                ret = action->handler(irq, action->dev_id);        //指向注册的中断处理函数
                action = action -> next;
                  ......
        } while (action);
        .....
}
```

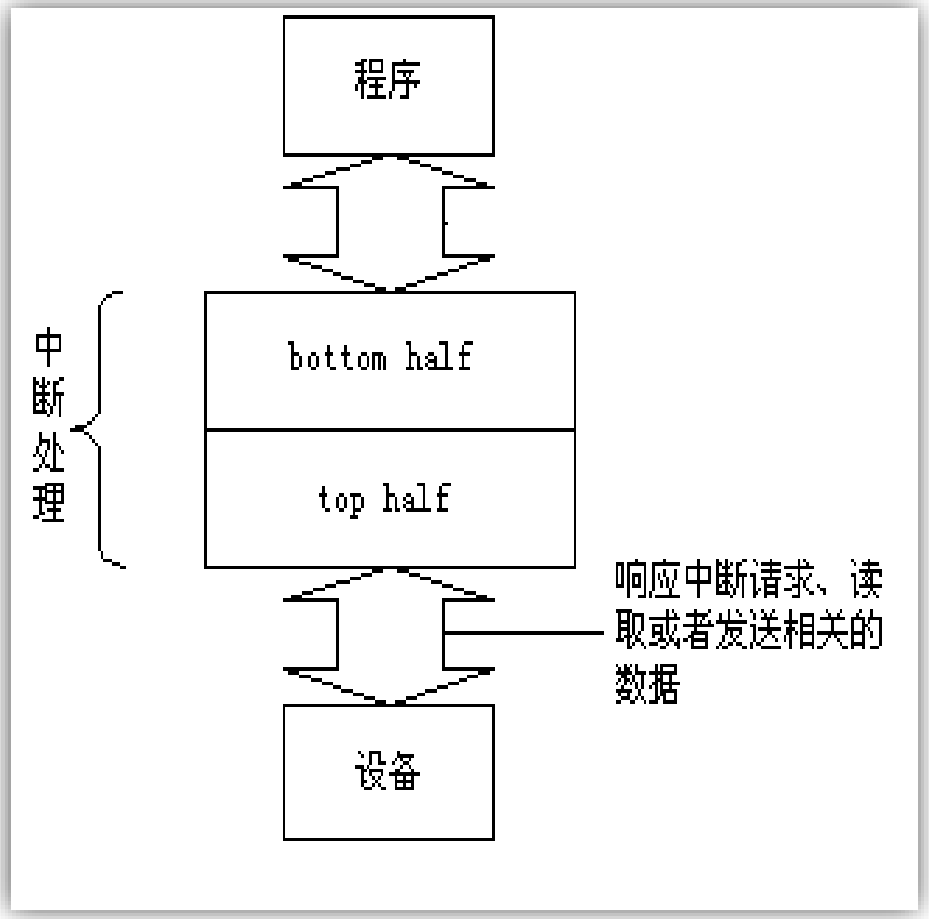
  - 中断服务程序都是在关中断的条件下执行

Part 4 中断的下半部处理机制

中断处理

- 硬件中断处理程序以异步方式执行，它会打断其他重要的代码执行，为了避免被打断的程序停止时间过长，硬件中断处理程序必须尽快执行完成
- 中断服务程序都是在关中断的条件下执行(CPU在穿过中断门时自动关闭中断)，避免嵌套使中断控制复杂化
- 系统不能长时间关中断运行，否则可能导致中断丢失

中断的上半部与下半部



- 中断的分割

中断的上半部与下半部

- 内核把中断处理分为两部分：上半部（top half）和下半部（bottom half），上半部内核立即执行，而下半部留着稍后处理
  - 如：当数据块到达网口时，中断控制器接收到该中断请求信号，内核只是简单的标志数据到达，然后让处理器恢复到以前运行的状态，其余的处理稍后再进行
  - 一个快速的“上半部”来处理硬件发出的请求，它必须在一个新的中断产生之前终止
  - 下半部运行时是允许中断请求的，而上半部运行时是关中断的，即通过引入下半部，实现了中断的同步

中断的上半部与下半部

- 上半部通常完成整个中断处理任务中的一小部分，如响应中断表明中断已经被软件接收，以及硬件中断处理完成时发送EOI信号给中断控制器等
- 中断处理任务中的计算任务，如数据复制、数据封装和转发及计算时间较长的数据处理等，均可放到中断下半部来执行
- Linux内核无严格规则约束何种任务应该放到下半部，一般为驱动开发者自定，中断任务的划分直接影响系统性能
- 中断的下半部执行无确切时间点，一般是从硬件中断返回后某一个时间点内被执行，下半部执行的关键是开中断环境，允许响应所有的中断

中断的下半部机制

- 中断常见下半部处理机制有3种
  - 软中断(SoftIRQ)
  - 小任务机制(Tasklet)
  - 工作队列机制

软中断

- 软中断于Linux内核2.3引入
- 软中断是预留给系统中对时间最为严格和最重要的下半部使用，目前驱动中只有块设备和网络子系统使用了软中断
- 系统静态定义了若干软中断类型，且内核开发者不希望用户扩充新的软中断类型，如有需要，建议使用小任务和工作队列机制

软中断

- 系统静态定义了若干软中断类型

```
#include/linux/interrupt.h
enum{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,        /* Preferable RCU should always be the last softirq */
        NR_SOFTIRQS
};
```

软中断

◆ 软中断类型

- HI_SOFTIRQ=0:优先级为0，最高优先级的软中断类型
- TIMER_SOFTIRQ：优先级为1，用于定时器的软中断
- NET_TX_SOFTIRQ：优先级为2，用于发送网络数据包的软中断
- NET_RX_SOFTIRQ：优先级为3，用于接收网络数据包的软中断
- BLOCK_SOFTIRQ：优先级为4，用于块设备的软中断
- BLOCK_IOPOLL_SOFTIRQ：优先级为5，用于块设备的软中断
- TASKLET_SOFTIRQ：优先级为6，用于tasklet机制的软中断
- SCHED_SOFTIRQ：优先级为7，进程调度及负载均衡
- HRTIMER_SOFTIRQ：优先级为8，高精度定时器
- RCU_SOFTIRQ：优先级为9，RCU服务的软中断

小任务机制

- 小任务机制是为要推迟执行的函数进行组织的一种机制，推迟的事情由tasklet_handler实现，至于何时执行，由小任务机制封装后交给内核去处理

- 小任务相关操作
  - 详见程序举例：tasklet_test.c

```
[ 3300.367685] mark!!
[ 3300.367692] tasklet_handler is running!
```

程序运行结果

工作队列

- 工作队列是另外一种将工作推后执行的形式
- 工作队列可以把工作推后，交由一个内核线程执行
- 工作队列允许被重新调度甚至睡眠
  - 如果推后执行的任务需要睡眠，选择工作队列
  - 若推后执行的任务不需睡眠，则选择小任务
  - 若需要用一个可以重新调度的实体来执行下半部的处理，也应使用工作队列

工作队列

- 工作队列数据结构

```
struct  work_struct{
    unsigned long pending;              // 这个工作正在等待处理吗?
    struct list_head entry;             //工作的链表
    void (*func) (void *);              // 要执行的函数
    void *data;                          // 传递给函数的参数
    void *wq_data;                       // 内部使用
    struct timer_list timer;             // 延迟的工作队列所用到的定时器
};
```

工作队列

- 小任务相关操作
  - 详见程序举例：work_test.c

```
[ 4604.633436] mark! ! !
[ 4604.633441] work_handler is running!
```

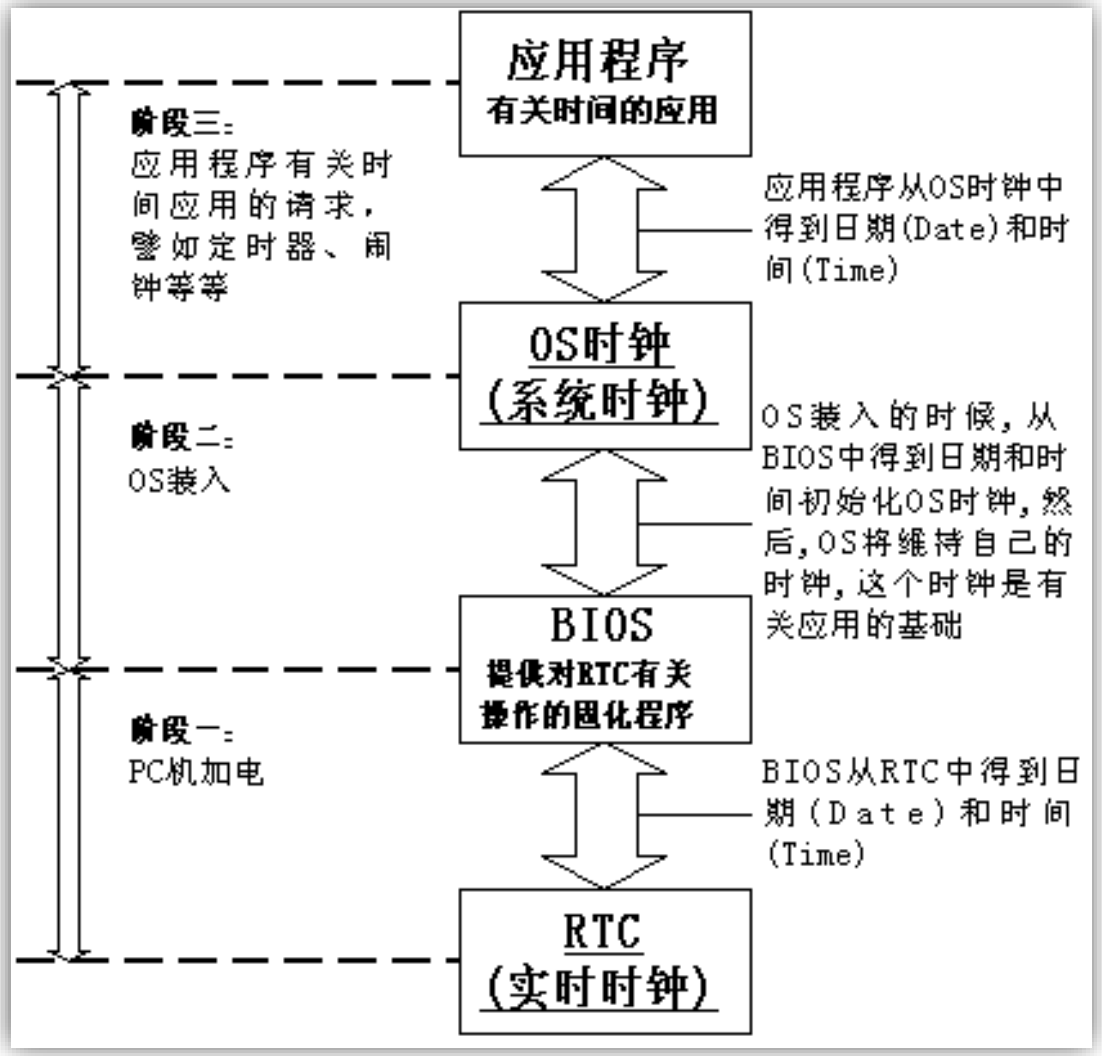
程序运行结果

Part 5 时钟中断

时钟硬件

- 大部分PC机中有两个时钟源，分别是实时时钟（RTC）和 操作系统（OS）时钟
  - 实时时钟也叫硬件时钟，它靠电池供电，即使系统断电，也可以维持日期和时间
  - OS时钟完全是一个软件问题，操作系统通过读取RTC来初始化OS时钟，此后二者保持同步运行，共同维持着系统时间

时钟运作机制



- Linux时钟运作机制

时钟运作机制

- 操作系统的“时间基准” 由设计者决定
  - Linux的时间基准是1970年1月1日凌晨0点
  - DOS时间基准是1980年1月1日
  - UNIX时间基准是1970年1月1日上午12点

时钟运作机制

- Linux中OS时钟以“时钟节拍”为单位
  - Linux中用全局变量jiffies表示系统自启动以来的时钟节拍数目
  - jiffies是记录着从电脑开机到现在总共的时钟中断次数
  - Linux内核从2.5版内核开始把频率从100调高到1000，即调整后每秒钟中断1000次（1ms/次）