中断是设备向cpu发送一个信号引起的，cpu会停掉手头的工作去执行这个设备的中断处理函数

1. **安装中断处理例程**

#include <linux/interrupt.h>

// 安装中断例程

int request\_irq(unsigned int irq,

                irqreturn\_t (\*handler)(int, void \*, struct pt\_regs \*),

                unsigned long flags,

                const char \*dev\_name,

                void \*dev\_id);

// 释放中断例程

void free\_irq(unsigned int irq, void \*dev\_id);

Irq：请求的中断号

Handler：安装的处理函数（处理例程）指针

Flags：如你会希望的, 一个与中断管理相关的选项的位掩码

SA\_INTERRUPT

当置位了, 这表示一个"快速"中断处理. 快速处理在当前处理器上禁止中断来执行

SA\_SHIRQ

这个位表示中断可以在设备间共享

SA\_SAMPLE\_RANDOM

这个位表示产生的中断能够有贡给 /dev/random 和 /dev/urandom 使用的加密池.

dev\_name：这个传递给 request\_irq 的字串用在 /proc/interrupts 来显示中断的

拥有者

dev\_id：通常用于保存设备驱动程序所需要的数据的指针，这个值会作为Handler函数的void\*参数，如果中断号是共享的，那么该值还会作为区分当前中断号上的例程的标识，此时，dev\_id不能为null

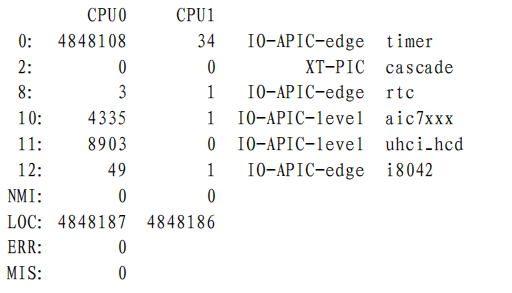
**中断号**

系统中一般有256个中断号

1. 0~31 的向量对应于异常和非屏蔽中断。
2. 32~47 是由I/O 设备引起的中断，分配给屏蔽中断。
3. 48~255 的向量用来标识软中断。

**查看中断号的分配**

cat /proc/interrupts



第一列是 IRQ 号，第二第三列指示该中断号接收了多少中断，最后一列表示使用中断号的设备名

1. **检测中断号**

当我们将设备连接到cpu时，我们如何知道设备使用的中断号是多少呢？

**自动检测**

自动检测是根据设备给出的物理地址判断设备使用的中断号

switch (short\_base)     // 根据设备的物理地址判断中断号

{

case 0x378:

    short\_irq = 7; // 打印机

    break;

case 0x278:

    short\_irq = 2;

    break;

case 0x3bc:

    short\_irq = 5;

    break;

}

**内核帮助下的检测**

示例：

do

{

    unsigned long mask;

    mask = probe\_irq\_on();          // 打开内核检测

    outb\_p(0x10, short\_base + 2);   // 启用设备的中断报告，short\_base为设备的物理地址

    outb\_p(0x00, short\_base);       // 清除设备中断位

    outb\_p(0xFF, short\_base);       // 设置设备的中断位，即设备产生中断

    outb\_p(0x00, short\_base + 2);   // 关闭设备的中断报告

    udelay(5);                      // 延迟5纳秒，以便内核有时间检测

    short\_irq = probe\_irq\_off(mask);    // 获取内核检测到的中断号，并关闭内核检测

    if (short\_irq == 0)

    {

        printk(KERN\_INFO "short: no irq reported by probe\n");

        short\_irq = -1;

    }

} while (short\_irq < 0 && count++ < 5);

内核检测需要在检测期间，设备产生一次中断，如果在检测期间，有多个中断产生，probe\_irq\_off返回负值，上述的示例为重新进行检测，如果在检测期间没有中断，probe\_irq\_off返回0

1. **实现处理例程**

**处理例程需要遵循的规则**

1. 处理例程不能休眠
2. 处理例程不能调用schdule函数（即不能让出cpu）
3. 处理例程不能和用户空间进行数据交换，因为处理例程不是由用户进程调用的

**处理例程的任务**

从设备取走数据，通知等待的进程

清除设备的中断挂起位，使设备又可以接收中断

**处理例程的参数和返回值**

示例：

static irqreturn\_t sample\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    struct sample\_dev \*dev = dev\_id;

    /\* now `dev' points to the right hardware item \*/

    /\* .... \*/

}

Irq：中断号

Dev\_id：在安装中断例程时传给request\_irq的dev\_id参数，该参数我们一般保存为设备的dev结构的指针

Regs： 很少用到，它持有一个处理器的上下文在进入中断状态前的快照

返回值：如果我们函数处理的设备的中断，则返回IRQ\_HANDLED，否则返回IRQ\_NONE

1. **禁止和启用中断**

为什么要禁止中断？

**禁止单个中断**

void disable\_irq(int irq);      // 禁止中断

void enable\_irq(int irq);       // 启用中断

**禁止所有中断**

// 禁止中断

// local\_irq\_save将当前状态保存到flags中，注意, flags 是直接传递, 不是通过指针

void local\_irq\_save(unsigned long flags);

void local\_irq\_disable(void);

// 启用中断

void local\_irq\_restore(unsigned long flags);

void local\_irq\_enable(void);

1. **中断处理前半部和后半部**

在中断处理中，我们希望能尽快处理完成，以免影响其他中断的处理，所有我们将中断处理分为前半部和后半部

前半部任务：

从设备中读取数据，并开启后半部，我们一般使用 处理例程 完成我们前半部工作

后半部任务：

通知进程接收数据

我们通过tasklt或工作队列实现后半部

**使用tasklt实现后半部**

// 声明 tasklt

void short\_do\_tasklet(unsigned long);                   // 后半部

DECLARE\_TASKLET(short\_tasklet, short\_do\_tasklet, 0);    // 第三个参数是要传给short\_do\_tasklet的参数

// 设备的处理例程（前半部）

irqreturn\_t short\_tl\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    // ...

    tasklet\_schedule(&short\_tasklet);       // 调用tasklt（后半部），系统会在合适的时间调用该tasklt

    return IRQ\_HANDLED;

}

**使用工作队列实现后半部**

// 创建一个工作

static struct work\_struct short\_wq;

INIT\_WORK(&short\_wq, (void (\*)(void \*))short\_do\_tasklet, NULL);

// 处理例程

irqreturn\_t short\_wq\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

// ...

    schedule\_work(&short\_wq);   // 将工作队列添加到默认工作队列中

    return IRQ\_HANDLED;

}

1. **中断共享**

设备可的中断号比较少，所以提供了中断共享来解决此问题

在注册中断例程request\_irq时，可以指定共享中断号，共享中断号的dev\_id不能为null

**中断共享的处理例程**

irqreturn\_t short\_sh\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    int value, written;

    struct timeval tv;

    value = inb(short\_base);

    if (!(value & 0x80))            // 检测设备的中断位是否置位

        return IRQ\_NONE;            // 如果未置位，返回未处理

outb(value & 0x7F, short\_base); // 清理设备的中断位

// ...

wake\_up\_interruptible(&short\_queue);    // 唤醒等待进程

    return IRQ\_HANDLED;

}