第 9 章 中断处理

中断是硬件管理的最终资源。众所周知，设备利用中断来通知软件可以对它进行操作了。Linux 为中断处理提供了很好的接口。事实上中断处理的接口如此之好，以至于编写和安装中断处理程序几乎和编写其它核心函数一样容易。但是由于中断处理程序和系统的其它部分是异步运行的，使用时要注意一些事项。

本章的示例代码使用并口来产生中断。因此，如果你想运行测试程序，你必须给你的电烙铁接上电源，即使在上一章的例子程序中你拒绝这样做。

我们用上一章的 short 模块来示范如何使用中断。这个模块的名字，*short*，实际上是指 *short int*(这是 C 语言，不是吗？)，提醒我们它要对中断(*int*errupt)进行处理。

准备并口

虽然我在第 8 章“*硬件管理*”的“使用并口”一节已经提到，并口很简单，但它也会触发中断。打印机就是利用这种能力来通知 *lp* 驱动程序它已准备好接收缓冲区中的下一个字符。

在指定接口这样做之前实际上并不会产生中断；并口标准规定设置 2 号端口的第 4 位(0x37a，0x27a 或其它某个地址)时启动中断报告。*short* 模块在初始化时调用 *outb* 设置该位。

启动中断报告之后，每当引脚 10(所谓的"ACK"位)上的电平从低变高时，并口都会产生一个中断。强迫接口（没有把打印机连到端口上）产生中断的最简单方法是将并行插座的引脚 9 和引脚 10 相连。为此，你需要一个阳性的 25 针 D 型插座和一英寸的电缆线。

引脚 9 是并行数据字节中最重要的一位。如果你往*/dev/short0* 中写入二进制数据，就可以产生几个中断。然而，往端口中写入 ASCII 文本将不会产生中断，因为此时不会设置这个最重要的位。

如果你确实想“看看”产生的中断，那么，仅仅往硬件设备中写是不够的；还必须在系统中配置一个软件处理程序。目前，Linux-x86 和 Linux-Alpha 只是简单的确认，忽略任何在预料之外的中断。

安装中断处理程序

中断信号线是宝贵并且非常有限的资源，当只有 15 或 16 根中断信号线时尤其如此。内核维护了一个类似于 I/O 端口注册表的中断信号线的注册表。一个模块可以申请一个中断通道（或中断请求 IRQ，即 Interrupt ReQuest），并且，处理完以后还可以释放掉它。在 **<linux/sched.h>**头文件中申明的下列函数实现了这个接口：

```
int request_irq(unsigned int irq,
                void (*handler)(int, void*, struct pt_regs *),
                unsigned long flags,
                const char *device, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

注意，1.2 版定义了不同的原型。相关的移植问题可参见本章稍后的“IRQ 处理程序的版本相关性”一节。

通常，申请中断的函数的返回值为 0 时表示成功，或者返回一个负的错误码。函数返回 **-EBUSY** 通知另一个设备驱动程序已经使用了要申请的中断信号线的情况并不常见。函数参数定义如下：

unsigned int irq

该参数为中断号。有时从 Linux 中断号到硬件中断号的映射并不是一对一的。例如，在 *arch/alpha/kernel/irq.c* 文件中可以查看到 Alpha 上的映射。这里，传递给内核函数的参数是 Linux 中断号而不是硬件中断号。

void (*handler)(int,void *,struct pt_regs *)

指向要安装的中断处理函数的指针。

unsigned long flags

如你所想，这是一个与中断管理有关的各种选项的字节掩码。

const char *device

传递给 *request_irq* 的字符串，在*/proc/interrupts* 中用于显示中断的拥有者（参见下一节）。

void *dev_id

这个指针用于共享的中断信号线。它是一个唯一的标志符，更象一个 ClientDate（C++ 中的 this 对象）。设备驱动程序可以自由地任意使用 **dev_id**。除非强制使用中断共享，**dev_id** 通常被置为 **NULL**。在后面的“实现一个处理程序”一节中，我们将看到一个使用 dev_id 的实际例子。

在 **flags** 中可以设置的位是：

SA_INTERRUPT

如果设置该位，就指示这是一个“快速”中断处理程序；如果清除这位，那么它就是一个“慢速”中断处理程序。快速中断处理程序和慢速中断处理程序的概念在下面的“快速和慢速处理程序”一节中会谈到。

SA_SHIRQ

该位表明中断可以在设备间共享。共享的概念在稍后的“中断共享”一节中介绍。

SA_SAMPLE_RANDOM

该位表明产生的中断对*/dev/random* 和*/dev/urandom* 设备要使用的熵池(entropy pool)有贡献。读这些设备返回真正的随机数，它们用来帮助应用软件选取用于加密的安全钥匙。这些随机数是从一个熵池中取得的，各种随机事件都会对系统的熵池(无序度)有贡献。如果你希望设备真正随机地产生中断，你应该置上这个标志。而如果你的中断是可预测的(例如，帧捕捉卡的垂直消隐)，那就不值得设置这个标志位－它对系统的熵池没有任何贡献。更详尽的信息可参见 *drivers/char/random.c* 文件中的注释。

中断处理程序可以在驱动程序初始化时或者在设备第一次打开时安装。虽然在 *init_module* 函数中安装中断处理程序听起来是个好主意，但实际上并非如此。因为中断信号线数量有限，你不会想浪费它们的。你的计算机拥有的设备通常要比中断信号线多。如果一个设备模块在初始化就申请了一个中断，会阻碍其它驱动程序使用这个中断，即便这个设备根本不使用它占用的这个中断。而在打开设备时申请中断，则允许资源有限的共享。

例如，只要你不同时使用帧捕捉卡和调制解调器这两个设备，它们使用同一个中断就是可能的。用户经常在系统启动时装载某个特殊设备的模块，即使这个设备很少使用。数据采集卡可以和第二个串口使用同一个中断。尽管在进行数据采集时避免去连你的 ISP 并不是件难事，但在使用调制解调器前不得不先卸载一个模块太令人不愉快了。

调用 *request_irq* 的正确位置是在设备第一次打开，硬件被指示产生中断*前*的时候。而调用 *free_irq* 的位置是设备最后关闭，硬件被通知不要再中断处理器*后*的时候。该技术的缺点是你必须为每个设备维护一个记录其打开次数的计数器。而如果你在同一个模块中控制两个以上的设备，那么仅仅使用模块计数器那还不够。

尽管我已说了这么多，*short* 却是在装载时申请中断信号线的。我这样做是为了使你在运行测试程序时不必运行其它进程来使设备保持打开的状态。因此，*short* 会象真正的设备那样，在 *init_module* 中而不是 *short_open* 中申请中断。

下面这段代码要申请的中断是 **short_irq**。对这个变量的赋值将在后面再给出，因为它与现在的讨论无关。**short_base** 是使用的并口的 I/O 基地址；写接口的 2 号寄存器打开中断报告。

```
if (short_irq >=0 ) {
    result=request_irq(short_irq, short_interrupt, SA_INTERRUPT, "short", NULL);
    if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
        short_irq=-1;
    }
    else { /*
        outb(0x10, short_base+2);
    }
}
```

这段代码显示安装的处理程序是个快速中断处理程序(**SA_INTERRUPT**)，不支持中断

共享(没有设置 **SA_SHIRQ**)，并且对系统熵池无贡献(没有设置 **SA_SAMPLE_RANDOM**)。然后调用 *outb* 打开并口的中断报告。

/proc 接口

当处理器被硬件中断时，一个内部计数器会被加 1，这为检查设备是否正常工作提供了一个方法。报告的中断显示在文件*/proc/interrupts* 中。下面是我的 486 启动一个半小时(uptime)后该文件的一个快照：

```
 0:     537598    timer
 1:      23070    keyboard
 2:          0    cascade
 3:       7930 +  serial
 5:       4568    NE2000
 7:      15920 + short
13:          0    math error
14:      48163 + ide0
15:       1278 + ide1
```

第一列是 IRQ 中断号。你可以从显示中缺少一些中断推知该文件只会显示已经安装了驱动程序的那些中断。例如，第一个串口(使用中断号 4)没有显示，这表明我现在没有使用调制解调器。实际上，即使我在获取这个快照之前使用过调制解调器，它也不会出现在这个文件中；串口的行为很良好，当设备关闭时会释放它们的中断处理程序。出现在各记录中的加号标志该行中断采用了快速中断处理程序。

/proc 树中还包含了另一个与中断有关的文件，*/proc/stat*；有时你可能会发现一个文件更有用，但有时又更愿意使用另一个。*/proc/stat* 文件记录了关于系统活动的一些底层的统计信息，包括(但不仅限于)只系统启动以来接收到的中断次数。*stat* 文件的每一行都以一个字符串开始，它是该行的关键字；**intr** 标记正是我们要找的。下面的快照是在得到前面那个快照后半分钟获得的：

```
intr 947102 540971 23346 0 8795 4907 4568 0 15920 0 0 0 0 0 0 48317 1278
```

第一个数是总的中断次数，而其它每个数都代表一个中断信号，从 0 号中断开始。上面的快照显示 4 号中断被使用了 4907 次，虽然*当前*它的处理程序没有安装上。如果你测试的驱动程序是在每次打开和关闭设备的循环中获取和释放中断的话，那么你会发现*/proc/stat* 文件要比*/proc/interrupts* 文件更有用。

两个文件另一处不同是 *interrupts* 文件与体系结构无关，而 *stat* 文件则与体系结构有关：其字段的个数取决于内核之下的硬件。可以获取的中断个数在 Sparc 上只有 15 个，而在 Atari(M68k 处理器)上则多达 72 个。

下面的快照给出我的 Alpha 工作站(共有 16 个中断，和 x86 机器一样)上的文件内容：

```
 1:          2   keyboard
 5:       4641   NE2000
15:      22909 + 53c7,8xx

intr    27555 0 2 0 1 1 4642 0 0 0 0 0 0 0 0 0 22909
```

这个输出的最值得注意的地方是不出现时钟中断。在 Alpha 机器上，时钟中断与其它中断到达处理器的方式不同，没有分配 IRQ 中断号。

自动检测中断号

驱动程序初始化时最迫切的问题之一就是如何决定设备要使用哪条中断信号线。驱动程序需要该信息以便安装正确的处理程序。虽然程序员可以要求用户在装载是指定中断号，但这并不好，因为一般用户并不知道中断号，或者是因为他没有配置跳线或者因为该设备根本就没有跳线。自动检测中断号是对驱动程序使用的基本要求。

有时自动检测依赖于一些设备拥有的较少改变的缺省特性。此时，驱动程序可以就假定设备使用了这些缺省值。*short* 在检测并口时就正是这么作的。正如 *short* 的代码中所给出的，实现起来相当简明：

```
if (short_irq<0) /*  尚未指定：强制为缺省的  */
    switch(short_base){
        case 0x378: short_irq=7; break;
        case 0x278: short_irq=2; break;
        case 0x3bc: short_irq=5; break;
}
```

这段代码根据选定的 I/O 地址来分配中断号，但也允许用户在装载驱动程序时通过调用 **insmod short short_irq=x** 来覆盖缺省值。**short_base** 缺省为 0x378，因此 **short_irq** 缺省为 7。

有些设备设计得更为先进，会简单地“声明”它们要使用那个中断。此时，驱动程序可以通过读设备的某个 I/O 端口的一个状态字节来获得中断号。当目标设备能告述设备要使用哪个中断时，那么自动检测中断号就是探测设备，不需要额外工作来探测中断。

值得注意的是，现代的设备能提供自己的中断配置信息。PCI 标准通过要求外围设备声明要使用的中断信号线的方法来解决这个问题。关于 PCI 标准的讨论可参见第 15 章“*外设总线概貌*”。

遗憾的是，不是所有设备都对程序员友好，自动检测可能还是需要一些探测的。技术很简单：驱动程序告诉设备产生中断，然后观察会发生些什么。如果一切正常，那么只有一条中断信号线被激活了。

尽管探测在理论上很简单，实际的实现则并不那么简明。下面我们看看执行该任务的两种方法：调用内核定义的帮助函数和实现我们自己的版本。

核心帮助下的检测

主流的内核版本都提供探测中断号的底层工具。这种工具包括两个函数，都在头文件**<linux/interrupt.h>**中声明(该头文件也描述了探测的机制)：

unsigned long probe_irq_on(void);

这个函数返回尚未分配的中断的位掩码。驱动程序必须保留返回的位掩码以便随后能将它传递给 *probe_irq_off* 函数。调用该函数后，驱动程序要安排相应设备至少产生一次中断。

int probe_irq_off(unsigned long);

在设备已经申请了中断之后，驱动程序要调用这个函数，传递给它的参数是先前调用 *probe_irq_on* 返回的位掩码。*probe_irq_off* 返回“启动探测”后发出的中断次数。如果没有发生任何中断，就返回 0(因此无法探测 0 号中断，但在能支持的所有体系结构上也没有什么定制设备能使用它)。如果产生了多次中断(二义性检测)，*probe_irq_off* 将返回一个负值。

程序员要注意在调用 *probe_irq_on* 后启动设备，并在调用 *probe_irq_off* 后关闭它。此外，在调用 *probe_irq_off* 之后，不要忘了处理你的设备尚未处理的那些中断。

short 模块演示了如何进行这样的探测。如果你在装载模块时指定 **probe=1** 并且并口插座的 9 号和 10 号引脚相连，就会执行下面的代码进行中断信号线的检测。

```
int count=0;
do {
    unsigned long mask;

    mask=probe_irq_on();
    outb_p(0x10, short_base+2); /*  启动中断报告  */
    outb_p(0x00,short_base); /*  清位  */
    outb_p(0xFF, short_base);   /*  置位：中断！ */
    outb_p(0x00, short_base+2); /*  关闭中断报告  */
    short_irq=probe_irq_off(mask);

    if (short_irq==0){ /*  没有探测到中断报告？ */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq=-1;
    }
    /*
     *  如果激活了一个以上的中断，结果就是负的。我们将为中断提供服务(除非是 lpt
     *  端口)并且再次进行循环。最多循环 5 次，然后放弃
     */
} while (short_irq<0 && count++<5);
```

```
if (short_irq<0)
    printk("short: probe failed %i times, giving up\n",count);
```

探测很耗时。尽管 *short* 的探测很快，但象探测帧捕捉卡，就至少需要延迟 20ms(相对处理器时间就太长了)，而探测其它设备可能会更花时间。因此，最好就只在模块初始化时探测中断信号线一次，不管你是在打开设备时(你应该这样做)或者在 *init_module* 中(你无论如何不应该这样做)安装你的中断处理程序的。

值得注意的是，在 Sparc 和 M68k 上，中断探测全无必要，因此也不必实现。探测是种“黑客”行为，象 PCI 这样的成熟的体系结构会提供所有必要的信息。实际上，M68k 和 Sparc 的内核开放给模块桩(stub)的探测函数总是返回 0——每种体系结构都必须定义这些函数，因为它们是由体系结构无关的源文件来开放的。所有其它的体系结构都允许使用上面给出的探测技术。

probe_irq_on 和 *probe_irq_off* 的问题是早期的内核版本并不开放这两个函数。因此，如果你希望写的模块能移植到 1.2 版的内核，你必须自己做中断探测。

DIY(Do It Yourself 自己做)检测

探测也可以有驱动程序自己较容易地实现。如果装载是指定 **probe=2**，*short* 模块将对中断信号线进行 DIY 检测。

实现机制和前面讨论的内核帮助下的检测是一样的：启动所有未被占用的中断，然后等着看会发生些什么。但我们可以利用拥有的对设备的一些知识。通常一个设备可以配置成使用 3 或 4 个中断号中的一个；只需要探测这些中断号，这使我们不必测试所有可能的中断号就可以检测到正确的中断号。

在 *short* 的实现中假定可能的中断号只有 3，5，7 和 9。这些数值实际上是一些并口允许你选取的值的范围。

下面的代码通过测试所有“可能的”中断和会观察发生什么来进行中断探测。**trials** 数组列出所有要尝试的中断号，0 是该列表的结束标志；**trials** 数组用于记录实际上哪个处理程序被驱动程序注册了。

```
int trials[]={3,5,7,9,0};
int tried[]={0,0,0,0,0};
int i,count=0;

/*
*为所有可能的中断信号线安装探测处理程序。记录下结果(0 表示成功，-EBUSY
*表示失败)以便只释放申请的中断
*/
for (i=0; trials[i]; i++)
    tried[i]=request_irq(trials[i], short_probing, SA_INTERRUPT, "short probe", NULL);
```

```
do {
    short_irq=0; /* 尚未取得中断号 */
    outb_p(0x10, short_base+2); /* 启动 */
    outb_p(0x00, short_base);
    outb_p(0xFF, short_base); /* 置位 */
    outb_p(0x10, short_base+2); /* 关闭 */

    /* 处理程序已经设置了这个值 */
    if   (short_irq==0) { /*
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
    * 如果激活了一个以上的中断，结果就是负的。我们将为中断提供服务(除非是 lpt
    * 端口)并且再次进行循环。最多这样做 5 次
    */
} while(short_irq<=0 && count++<5);

/* 循环结束，卸载处理程序 */
for (i=0; trials[i]; i++)
    if (tried[i]==0)
        free_irq(trials[i],NULL);

if (short_irq<0)
    printk("short: probe failed %i times, giving up\n",count);
```

你可能事先不知道“可能的”中断号。此时，你需要探测所有空闲的中断，而不仅是一些 **trials[]**。为了探测所有的中断，你不得不从 0 号中断探测到 **NR_IRQS-1** 号中断，**NR_IRQS** 是在头文件**<asm/irq.h>**中定义的与平台无关的常数。

现在缺的就是探测处理程序自己了。该处理程序的功能就是根据实际接收到的中断号来更新 **short_irq** 变量。**short_irq** 值为 0 意味着“什么也没有”，而负值意味着存在“二义性”。我选取这些值是为了和 *probe_irq_off* 保持一致，并可以在 *short.c* 中使用同样的代码来调用任何一种探测方法。

```
void short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq == 0) short_irq = irq;       /* 找到 */
    if (short_irq != irq) short_irq = -irq; /* 有二义性 */
}
```

处理程序的参数稍后会介绍。知道参数**irq**是要处理的中断号就足以理解上面的函数了。

快速和慢速中断处理

你已经看到，我为 short 的中断处理程序设置了 **SA_INTERRUPT** 标志位，因此是请求安装一个快速中断处理程序。现在到解释什么是“快速”和“慢速”的时候了。实际上，不是所有的体系结构都支持快速和慢速中断处理程序两种实现的。例如，Alpha 和 Sparc 的移植版本，快速和慢速处理程序是一样处理的。2.1.37 版和其后的 Intel 移植版本也消除了两者的差别，因为现代处理器的可以获得的处理能力使得我们不必再区分出快速和慢速两种中断。

这两种中断处理程序的主要差别就在于，快速中断处理程序保证中断的原子处理，而慢速中断处理程序则不保证(这种差别在最新的中断处理的实现也保留了)。也就是说，“开启中断”处理器标志位(IF)在运行快速中断处理程序时是关闭的，因此在服务该中断时不允许被中断。而调用慢速中断处理时，内核启动微处理器的中断报告，因此在运行慢速中断处理程序时其它中断仍可以得到服务。

在调用实际的中断处理程序之前，不管是快速还是慢速中断处理程序，内核都要执行一项任务，关闭刚才发出报告的那个中断信号线。这对程序员是个好消息－中断服务例程不必是可重入的。但另一方面，即使是慢速中断处理程序也要实现得运行的尽可能快，以免丢失后面到达的中断。

当处理程序还在处理上一个中断时，如果设备又发出新的中断，新的中断会永远丢失。中断控制器并不缓存被屏蔽的中断，但是处理器会进行缓存－一旦发出 *sti* 指令，待处理的中断就会得到服务。*sti* 函数是“置中断标志位”处理器指令(是在第 2 章“*编写和运行模块*”的“ISA 内存”一节引入的)。

总结快速和慢速两种执行环境如下：

- 快速中断处理程序运行时微处理器关闭了中断报告，中断控制器禁止了被服务这个中断。但处理程序可以通过调用 *sti* 来启动处理器的中断报告。

- 慢速处理程序运行时启动了处理器的中断报告，但中断控制器也禁止了被服务这个中断。

但快速和慢速中断处理程序还有另一处不同：内核带来的额外开销。慢速中断处理程序*之所以*慢是因为内核带来的一些管理开销造成的。这意味着较频繁的中断最好由快速中断处理程序为之提供服务。至于 *short*，当把大文件拷贝到*/dev/short0* 时每秒会产生上千次中断。因此我选择使用了一个快速中断处理程序来控制添加给系统的开销。这种分别在更新的 2.1 版的内核中已经得到统一；这个开销现在加到了所有的中断处理程序上。

帧捕捉卡是使用慢速中断处理程序的一个好的候选者。它每秒只中断处理器 50 到 60 次，选择使用慢速处理程序将帧数据从接口卡拷贝到物理内存就不会阻塞住其它的系统中断，例如那些由串口或定时器服务产生的中断。

x86 平台上中断处理的内幕

下面的描述是根据 2.0.x 版本的内核中的两个文件 *arch/i386/kernel/irq.c* 和 *include/asm-i386/irq.h* 推断的；虽然基本概念是相同的，但是具体的硬件细节与平台有关，并且在 2.1 开发版本中有些修改。

最底层的中断处理是在头文件 *irq.h* 中的声明为宏的一些汇编代码，这些宏在文件 *irq.h* 中被扩展。为每个中断声明了三种处理函数：慢速，快速和伪(bad)处理函数。

“伪”处理程序，它最小，是当没有为中断安装C语言的处理程序时的汇编入口点。它将中断转交给适当的PIC(Programmable Interrupt Controller，可编程的中断控制器)设备[*]的同时禁止它，以避免由于伪中断而进一步浪费处理器时间。在驱动程序处理完中断信号后调用 *free_irq*时又会重新安装伪处理程序。伪处理程序不会将*/proc/stat*中的计数器加 1。

值得注意的是，在 x86 和 Alpha 上的自动探测都是依赖于伪处理程序的这种行为。*probe_irq_on* 启动所有的伪中断，而不安装处理程序；*probe_irq_off* 只是简单地检查自调用 *probe_irq_on* 以来那些中断被禁止了。如果你想验证这一点，可以在装载 *short* 时指定 **probe=1**(内核帮助下的检测)，此时可观察到中断计数器没有加 1，而如果装载时指定 **probe=2**(DIY 检测)则会将它们加 1。

慢速中断的汇编入口点会将所有寄存器保存到堆栈中，并将数据段(DS 和 ES 处理器寄存器)指向核心地址空间(处理器已经设置了 CS 寄存器)。然后代码将将中断转交给 PIC，禁止在相同的中断信号线上触发新的中断，并发出一条 sti 指令(set interrupt flag，置中断标志位)。注意处理器在对中断进行服务时会自动清除该标志位。接着慢速中断处理程序就将中断号和指向处理器寄存器的一个指针传递给 *do_IRQ*，这是一个 C 函数，由它来调用相应的 C 语言处理程序。驱动程序传递给中断处理程序参数 **struct pt_regs ***是一个指向存放着各个寄存器的堆栈的指针。

do_IRQ 结束后，会发出 *cli* 指令，打开 PIC 中指定的中断，并调用 *ret_from_sys_call*。最后这个入口点(*arch/i386/kernel/entry.S*)从堆栈中恢复所有的寄存器，处理所有待处理的下半部处理程序(参见本章的“下半部”一节)，并且，如果需要的话，重新调度处理器。

快入口点不同的是，在跳转到 C 代码之前并不调用 *sti* 指令，并且在调用 *do_fast_IRQ* 前并不保存所有的机器寄存器。当驱动程序中的处理程序被调用时，**regs** 参数是 **NULL**(空指针，因为寄存器没有保存到堆栈中)并且中断仍被屏蔽。

最后，快速中断处理程序会重新打开 8259 芯片上的所有中断，恢复先前保存的所有寄存器，并且不经过 *ret_from_sys_call* 就返回了。待处理的下半部处理程序也不运行。

2.1.34 前的所有内核版本中，这两种处理程序在将控制转移给 C 代码前都会将 **intr_count** 变量加 1(参见第 6 章“*时间流*”的“任务队列的特性”一节)。

---

[*] PC机通常就已经有了两个中断控制芯片，叫做 8259 芯片(主从片)。而可编程的中断控制器设备已经不存在了，但现代的芯片组中也实现了相同的功能。

实现中断处理程序

至此，我们学习了如何注册一个中断处理程序，但还并没有真正编写这样的一个处理程序。实际上，处理程序并没有什么特别的－就是普通的 C 代码。

唯一特别的地方就是处理程序是在中断时间内运行的，因此它的行为要受些限制。这些限制和我们在任务队列中看到的差不多。处理程序不能向用户空间发送或接受数据，因为它不在任何进程的上下文中执行。快速中断处理程序，可以认为是原子地执行的，当访问共享的数据项时并不需要避免竞争条件。而慢速处理程序不是原子的，因为在运行慢速处理程序时也能为其它处理程序提供服务。

中断处理程序的功能就是将有关中断接收的信息反馈给设备，并根据要服务的中断的不同含义相应地对数据进行读写。第一步通常要先清除接口卡上的一个位；大部分硬件设备在它们的“中断待处理”位被清除前是不会产生任何中断的。一些设备就不需要这一步，因为它们没有“中断待处理”位；这样的设备比较少，但并口却是其中之一。因此，*short* 不需要清除这样的位。

中断处理程序的典型任务是唤醒在设备上睡眠的那些进程——如果中断向这些进程发出了信号，指示它们等待的事件已经发生，比如，新数据到达了。

还举老的帧捕获卡的例子，进程可以通过连续地对设备读来获取一系列的图像；每读一帧后 *read* 调用都被阻塞，而新的帧一到达中断处理程序都会唤醒该进程。这假定了捕获卡会中断处理器来发出信号通知每一帧的成功到达。

不论是快速还是慢速中断处理程序，程序员都要注意处理例程的执行时间必须尽可能短。如果要进行长时间的计算，最好的方法是使用任务队列，将计算调度到安全时间内进行(参见第 6 章的“任务队列”一节)。这也是需要下半部处理的一个原因(参见本章稍后的“下半部”)。

short 中的范例代码使用中断来调用 *do_gettimeofday* 并把当前时间打印到大小为一页的循环缓冲区。然后它唤醒所有的读进程(实际上由于 *short* 使用快速中断处理程序，这些读进程只会在下一个慢速中断处理程序结束时或下一个时钟滴答时醒来)。

```
void short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    do_gettimeofday(&tv);

    /*  写一个 16 个字节的记录。假设  PAGE_SIZE 是 16 的倍数  */
    short_head += sprintf((char *)short_head,"%08u.%06u\n",
                          (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    if (short_head == short_buffer + PAGE_SIZE)
        short_head = short_buffer; /*  绕回来  */
```

```
    wake_up_interruptible(&short_queue); /* 唤醒所有的读进程 */
}
```

这段代码，尽管简单，却给出了一个中断处理程序的典型工作流程。

用来读取在中断时间里填满的缓冲区的节点是*/dev/shortint*。这是唯一的没有在第 8 章中介绍的 *short* 设备节点。*/dev/shortint* 内部的实现为中断产生和报告作了特别的处理。每向设备写入一个字节都会产生一个中断；而读设备时则给出每次中断报告的时间。

如果你将并口插座的第 9 和第 10 引脚相连，那么拉高并行数据字节的最高位就可以产生中断。这可以通过向*/dev/short0* 写二进制数据或者向*/dv/shortint*[*]写入任意数据来实现。

下面的代码是*/dev/shortint* 的 *read* 和 *write* 的实现：

```
read_write_t short_i_read (struct inode *inode, struct file *filp,
                           char *buf, count_t count)
{
    int count0;

    while (short_head == short_tail) {
        interruptible_sleep_on(&short_queue);
        if (current->signal & ~current->blocked) /* 有信号到达 */
          return -ERESTARTSYS; /* 通知 fs 层去处理它 */
        /* 否则，再次循环 */
    }
    /* count0 是可以读进来的数据字节个数 */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    memcpy_tofs(buf, (char *)short_tail, count);
    short_tail += count;
    if (short_tail == short_buffer + PAGE_SIZE)
        short_tail = short_buffer;
    return count;
}

read_write_t short_i_write (struct inode *inode, struct file *filp,
                            const char *buf, count_t count)
{
    int written = 0, odd = filp->f_pos & 1;
    unsigned port = short_base; /* 输出到并口数据锁存器 */
```

[*] *shortint*设备是通过交替地向并口写入 0x00 和 0xff来实现的。

```
        while (written < count)
            outb(0xff * ((++written + odd) & 1), port);

        filp->f_pos += count;
        return written;
    }
```

使用参数

虽然 *short* 中不对参数进行处理，但还是有三个参数被传给了中断处理函数：**irq**，**dev_id** 和 **regs**。下面我们看看每个参数的意义。

当用一个处理程序来同时对若干个设备进行处理并且使用不同的中断信号线，那么中断号**(int irq)**就很有用了。例如，立体视频系统就使用了两个中断来支持两个帧捕捉卡。驱动程序必须能检测两个设备，并且安装一个处理程序来对两个中断进行处理。驱动程序就可以使用 **irq** 参数来通知处理程序是哪个设备发出了中断。

例如，如果驱动程序声明了一个设备结构的数组 **hwinfo**，每个元素都有一个 **irq** 域，那么下面的代码可以在中断到达时选取出正确的设备。这段代码的设备前缀是 **cx**。

```
    static void cx_interrupt(int irq)
    {
        /* “Cxg_Board”是硬件信息的数据类型 */
        Cxg_Board *board; int i;

        for (i=0, board=hwinfo; i>cxg_boards; board++,i++)
            if   (board->irq==irq)
                break;

        /* 现在'board' 指向了正确的硬件描述 */
        /* .... */
    }
```

第二个参数，**void *dev_id**，是一种 ClientData；是传递给 *request_irq* 函数的一个 **void *** 类型的指针，并且当中断发生时这个设备 ID 还会作为参数传回给处理程序。参数 **dev_id** 是在 1.3.70 版的 Linux 中引入以处理共享中断，但即使不共享它也很有用。

假定我们例子中的设备是象下面这样注册它的中断的(这里 **board->irq** 是要申请的中断，**board** 是 ClientData)

```
    static void cx_open(struct inode *inode, struct file *filp)
    {
        Cxg_Board *board=hwinfo+MINOR(inode->i_rdev);
        Request_irq(board->irq, cx_interrupt, 0, "cx100", board /* dev_id */);
```

```
    /* .... */
    return 0;
}
```

这样处理程序的代码就可以缩减如下：

```
static void cx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    Cxg_Board *board=dev_id;

    /*  现在'board'  指向了正确的硬件项  */
    /* .... */
}
```

最后一个参数，**struct pt_regs *regs**，很少使用。它存放着在处理器进入中断代码前的一个处理器上下文的快照。这些寄存器可用于监控和调试，实际上 *show_regs* 函数(它是按下 **RightAlt-PrScr** 键时由键盘中断启动的调试函数－第 4 章“*调试技术*”的“系统挂起”一节)就是使用它们来实现监控和调试的。

打开和禁止中断

有时驱动程序要打开和禁止它相应 IRQ 信号的中断报告。内核为此提供了两个函数，都在头文件**<asm/irq.h>**中声明：

```
void disable_irq(int irq);
void enable_irq(int irq);
```

调用其中任一函数都会更新 PIC 中对指定的 **irq** 的掩码。

实际上，当中断被禁止后，那么即使硬件急需处理，处理器也得不到报告。例如，“x86 上中断处理的内幕”一节中就介绍了“伪”处理程序在 x86 上的实现就禁止了它收到的所有中断。

但是，为什么我们要禁止中断呢？还是举并口的例子，我们看看 *plip*(并行 IP)网络接口。*plip* 设备使用裸的(bare-bones)并口来传输数据。因为只能从并口读出 5 个位，它们就被解释为四个数据位和一个时钟/握手信号。当发起者(即发送数据包的那个接口)送出数据包的第一个位时，时钟信号会升高，接收方接口就会中断处理器。然后 *plip* 处理程序被调用来处理新到达的数据。

在设备被激活后，开始数据传输，使用握手信号将新数据按时钟周期传送给接收接口(这可能不是最好的实现方法，但只有这样才能和其它使用并口的数据包驱动程序兼容)。如果接收接口为接收每个字节(8 个位)都要处理两次中断，那性能必然不可忍受。因此驱动程序在接收数据包时要禁止中断。

同样的，因为从接收方到发送方的握手信号用于确认数据的接收，发送接口也要在发送数据包时禁止它的中断信号。

但要注意的是，因为处理程序本身无法打开和禁止中断信号。存在这个限制是因为，如上所述，内核在调用处理程序前会禁止中断，而在处理程序结束后又会重新打开它。但打开和禁止中断仍可以做到，只要在下半部处理程序中作就可以了(参见下一节)。

最后值得注意的是，在 Sparc 实现中，*disable_irq* 和 *enable_irq* 都被定义为指针而不是函数。这个小技巧允许内核在启动检测你是在运行哪种 Sparc 时对指针进行相应的赋值(Sun4c 和 Sun4m 的中断硬件不相同)。而所有的 Linux 系统上，不管使不使用这种小技巧，函数在 C 语言中的语义都相同，这就避免了编写那些冗长无味的条件编译代码。

下半部

中断处理的一个主要问题是如何在处理程序中完成比较耗时的任务。Linux 解决这个问题的方法是将中断处理程序划分成两个部分：所谓的“上半部”是你通过 *request_irq* 函数注册的处理例程，而“下半部”(bottom half，简称为“bh”)则是由上半部调度到以后在更安全的时间内执行的那部分例程。

但是下半部有什么用呢？

上半部和下半部处理程序最大的不同就在于在执行 bh 是所有的中断都是打开的－所以说它是在“更安全”时间内运行。典型的情况是，上半部处理程序将设备数据存放进一个设备指定的缓冲区，再标记它的下半部，然后退出；这样处理得就非常快。由 bh 将新到的数据再分派给各个进程，必要时再唤醒它们。这种设置允许上半部处理程序在下半部还在运行时就能为新的中断提供服务。但另一方面，在上半部处理程序结束前如果有新的数据到了，由于中断控制器禁止了中断信号，这些数据仍会丢失掉。

所有实际的中断处理程序都作了这样的划分。例如，当网络接口卡报告新的数据包到达了，处理程序只是取得数据并将它推进协议层中；对数据包的实际处理是在下半部中完成的。

这使我们想起了任务队列；实际上，任务队列就是从下半部的一个较老的实现演变而来的。甚至 1.0 版的内核也有下半部，而任务队列则还未引入。

与动态的任务队列不同，下半部的个数有限，并由内核预定义了；这和老的内核定时器有些类似。下半部的静态特性并不是个问题，因为有些下半部可以通过运行任务队列演变为动态对象。在头文件**<linux/interrupt.h>**中，你可以看到下半部的一张列表；它们的最有意思的一部分将在下面讨论。

下半部的设计

下半部由一个函数指针数组和一个位掩码组成－这就是为什么它们不超过 32 个的原因。当内核准备处理异步事件时，它就调用 *do_bottom_half*。我们已经在前面看到，从系统

调用返回和退出慢速处理程序时，内核都是这样做的；而这两类事件都发生得很频繁。而决定使用掩码主要出于性能的考虑：检查掩码只要一条机器指令，开销最小。

当某段代码需要调度运行下半部处理时，只要调用 *mark_bh* 即可，该函数设置了掩码变量的一个位以将相应的函数进入执行队列。下半部可以由中断处理程序或其它函数来调度。执行下半部时，它会自动去除标记。

标记下半部的函数是在头文件**<linux/interrupt.h>**中定义的：

void mark_bh(int nr);

这里，**nr** 是激活的 bh 的“数目”。这个数是在头文件**<linux/interupt.h>**中定义的一个符号常数，它标记位掩码中要设置哪个位。每个下半部 bh 相应的处理函数由拥有它的那个驱动程序提供。例如，当调用 **mark_bh(KEYBOARD_BH)**时，要调度执行的函数是 *kbd_bh*，它是键盘驱动程序的一部分。

因为下半部是静态对象，模块化的驱动程序无法注册*自己的*下半部。目前还不支持下半部的动态分配，可能将来也不会支持，因为此时可以使用立即队列。

本节其余部分将列出一些有意思的下半部：

IMMEDIATE_BH

对设备驱动程序来说这是最重要的 bh。被调度执行的函数处理任务队列 **tq_immediate**。没有下半部的驱动程序(例如一个定制模块)可以通过使用立即队列来取得和立即 bh 同样的效果。将任务等记到队列中后，驱动程序必须标记 bh 以使得它的代码真正得到执行；具体做法可参见第 6 章的“立即队列”一节。

TQUEUE_BH

*如果*任务等记在 tq_timer 队列中，那么每次时钟滴答都会激活这个 bh。实际上，驱动程序可以使用 tq_timer 来实现自己的下半部；定时器队列是在第 6 章(“定时器队列”一节中)引入的一种下半部，但并不必为它调用 *mark_bh*。**TQUEUE_BH** 总是在 **IMMEDIATE_BH** *后*执行的。

NET_BH

网络驱动程序通过标记这个队列来将事件通知上面的网络层。bh 本身是网络层的一部分，模块无法访问。我们将在第 14 章“*网络设备驱动程序*”的“中断驱动的操作”一节中熟悉它的使用。

CONSOLE_BH

控制台是在下半部中进行终端tty切换的。这个操作要包含进程控制。例如，在X Window系统和字符模式间切换就是由X 服务器控制的。而且，如果键盘驱动程序请求控制台的切换，那么控制台切换不能在中断时进行。也不能在进程向控制台写的时候进行。使用bh就能满足这些要求，因为驱动程序可以任意禁止下半部；如果发生了前面情况，在

写控制台时禁止**console_bh**即可*。

TIMER_BH

这个 bh 由 *do_timer* 函数标记；*do_timer* 函数管理着时钟滴答。这个 bh 要执行的函数正是驱动内核定时器的那个函数。因此不使用 *add_timer* 的驱动程序是无法使用这种功能的。

其余的下半部是有特定的内核驱动程序使用的。没有为模块提供入口点，即使有入口也没什么意义。

bh 一旦被激活，当在 *return_from_sys_call* 中调用 *do_bottom_half*(*kernel/softirq.c*)时它就会得到执行。当进程退出系统调用或慢速中断处理程序退出时都会执行 *return_from_sys_call* 过程。快速中断处理程序退出时就不会执行下半部；如果驱动程序需要快速执行它的下半部。它必须注册一个慢速处理程序。

时钟滴答总要执行 *ret_from_sys_call* 的；因此，如果快速中断处理程序标记了一个 bh，实际的处理函数最多 10ms 后就会被执行(Alpha 上则小于 1ms，它时钟滴答的频率是 1024Hz)。

下半部运行后，如果设置了 **need_resched** 变量，就会调用调度器；各种 *wake_up* 函数都会设置这个变量。因此，上半部可以将任何与被唤醒的进程有关的任务放到下半部去做－这些任务马上就会被调度。例如，当 *telnet* 数据包到达网络时就是这样的。*net_bh* 唤醒 *telnetd*，并且调度器马上给它处理器时间，因此没有额外的延迟。

编写下半部

下半部代码是在安全时间内运行的——比上半部处理程序运行时安全。但是，也有些注意事项，因为 bh 还是在“中断时间”内处理的。intr_count 不为 0，因为下半部是在进程上下文之外执行的。因此，第 6 章的“任务队列的特性”一节中列出的各种限制也适用于在下半部中执行的代码。

下半部的主要问题是它们通常要与上半部中断处理程序共享数据结构，因此要避免竞争条件。这意味着要暂时禁止中断或者使用锁的技术。

从前面“下半部的设计”一节中给出的下半部列表可以明显看出，新编写的实现了下半部的驱动程序应该通过使用立即队列来将它的代码挂在 **IMMEDIATE_BH** 上。如果你的驱动程序很关键，那么甚至可以拥有从内核里分配的 bh 号。但这样的驱动程序比较少，因此我就不详细介绍了。共有三个函数可用于管理自己私有的下半部：*init_bh*，*enable_bh* 和 *disable_bh*。如果你有兴趣的话，可以在内核源码找到它们。

实际上，使用立即队列和管理自己拥有的下半部并无区别－立即队列也是一种下半部。当标记了 **IMMEDIATE_BH** 后，处理下半部的函数实际上就是去处理立即队列。如果你的

* 随后就会介绍，使用自己的下半部的驱动程序可以调用*disable_bh*函数。

中断处理函数将它的 bh 处理函数排进 **tq_immediate** 队列并且标记了下半部，那么队列中的这个任务会正确地被执行。因为所有最新的内核都可以将相同的任务多次排队而不破坏任务队列，因此每次运行上半部处理函数时都可以将下半部排队。稍后我们会看到这种做法。

需要特殊配置的驱动程序——需要多个下半部或不能简单地用 **tq_immediate** 来设置——可以使用定制的任务队列。中断处理函数将任务排进自己的队列中，当它准备运行这些任务时，就将一个简单的对队列进行处理的函数插入立即队列。详情可参见第 6 章的“运行自己的任务队列”一节。

下面让我们看看 *short* 的实现。装载时如果指定 **bh=1**，那么模块就会安装一个使用了下半部的中断处理函数。

short 是这样对中断处理进行划分的：上半部(中断处理函数)将当前时间保存到一个循环缓冲区中并调度下半部。而 bh 将累积的各个时间值打印到一个字符缓冲区，然后唤醒所有的读进程。

最后上半部非常简单：

```
void short_bh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday(tv_head);
    tv_head++;

    if (tv_head == (tv_data + NR_TIMEVAL) )
        tv_head = tv_data; /* wrap */

    /* 将 bh 排队。即使被多次排队也没有关系 */
    queue_task_irq_off(&short_task, &tq_immediate);
    mark_bh(IMMEDIATE_BH);

    short_bh_count++; /* 记录一个新的中断到了 */
}
```

正如我们所料，这段代码调用 *queue_task* 而不会检查任务是否已被排进队列。但在 Linux 1.2 中不能这么做，并且如果你是用 1.2 版的头文件来编译 *short*，那么它会使用不同的处理函数，仅当 **short_bh_count** 为 0 时该函数才会将任务排队。

然后，下半部处理剩下的工作。它也记录下在调度下半部前上半部被激活的次数(savecount)。如果上半部是一个“慢速”处理函数，那么这个数总为 1，因为如上所述，当慢速处理函数退出时，总会运行待处理的下半部。

```
void short_bottom_half(void *unused)
{
    int savecount = short_bh_count;
```

```
    short_bh_count = 0; /*  我们已经从队列中删去*/
    /*
    *  下半部读入由上半部填充的 tv 数组，并将它打印入循环的字符缓冲区，该缓冲区是
    *  由读进程处理的
    */

    /*  首先写入在这个 bh  前发生的中断的次数*/

    short_head += sprintf((char *)short_head,"bh after %6i\n",savecount);
    if (short_head == short_buffer + PAGE_SIZE)
    short_head = short_buffer; /*  绕回来  */

    /*
    *然后，写入时间值。每次写 16 个字节。因此与 PAGE_SIZE 是对齐的
    */

    do {
        short_head += sprintf((char *)short_head,"%08u.%06u\n",
                                    (int)(tv_tail->tv_sec % 100000000),
                                    (int)(tv_tail->tv_usec));
        if (short_head == short_buffer + PAGE_SIZE)
            short_head = short_buffer; /*  绕回来  */

        tv_tail++;
        if (tv_tail == (tv_data + NR_TIMEVAL) )
            tv_tail = tv_data; /*  绕回来  */

    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /*  唤醒所有读进程  */
}
```

在我的老式的计算上机运行时给出的时间值表明，使用下半部，两个中断间的时间间隔从 53ms 减少到了 27ms，因为上半部处理函数作的工作更少些。但处理中断的总的工作量不变，更快的上半部的优点是禁止中断的时间较短。但这对 *short* 不是个问题，因为只有在中断处理函数结束后才会重新调用产生中断的 *write* 函数(因为 *short* 采用的是快速中断处理函数)，但对真正的硬件中断来说，这个时间还是很有关系的。

下面是当装载 *short* 时指定 **bh=1** 你可能看到的输出结果：

```
morgana%echo 1122334455 > /dev/shortint; cat /dev/shortint
bh after          5
50588804.876653
```

50588804.876693
50588804.876720
50588804.876747
50588804.876774

共享中断

PC 机一个众所周知的“特性”就是不能将不同的设备挂到同一个中断信号线上。但是，Linux 打破了这一点。甚至我的 ISA 硬件手册－一本没提到 Linux 的书－也说“最多只有一个设备”可以挂到中断信号线上，除非硬件设备设计的不好，电信号上并无这样的限制。问题在于软件。

Linux 软件对共享的支持是为 PCI 设备做的，但也可用于 ISA 卡。不必说，非 PC 平台和总线也支持共享。

为了开发能处理共享中断信号的驱动程序，必须考虑一些细节。下面会讨论到，使用共享中断的驱动程序不能使用本章描述的一些特性。但最好尽可能对共享中断提供支持，因为这样对最终用户来说比较方便。

安装共享的处理程序

和已经拥有的中断一样，要与它共享的中断也是通过 *request_irq* 函数来安装的，但它们有两处不同：

- 申请共享中断时，必须在 **flags** 参数中指定 **SA_SHIRQ** 位
- **dev_id** 参数*必须*是唯一的。任何指向模块的地址空间的指针都可以，当然 **dev_id** 一定不能设为 **NULL**。

内核为每个中断维护了一张共享处理函数的列表，并且这些处理函数的 **dev_id** 各不相同，就象是驱动程序的签名。如果两个驱动程序都将 **NULL** 注册为它们对同一个中断的签名，那么在卸载时会混淆起来，当中断到达时内核就会出现 oops 消息。我第一次测试共享中断时就发生过这种事情(当时我只是想着“一定要将 **SA_SHIRQ** 位加到这两个驱动程序上”)。

满足这些条件之后，如果中断信号线空闲或者下面两个条件同时得到满足，那么 *request_irq* 就会成功：

- 前面注册的处理函数的 **flags** 参数指定了 **SA_SHIRQ** 位。
- 新的和老的处理函数同为快速处理函数，或者同为慢速处理函数。

需要满足这些要求的原因很明显：快速和慢速处理函数处于不同的环境，不能互相混淆。

类似的，你也不能与已经安装为不共享的中断处理函数共享相同的中断。但关于快速和慢速处理函数的限制对最近的 2.1 版的内核来说是不必要的，因为两种处理函数已经合并了。

当两个或两个以上的驱动程序共享同一根中断信号线，而硬件又通过这根信号线中断了处理器时，内核激活这个中断注册的所有处理函数，并将自己的 **dev_id** 传递给它们。因此，共享处理函数必须能够识别出它对应于哪个中断。

如果你在申请中断信号之前需要探测你的设备的话，内核无法提供帮助。没有共享中断的探测函数。仅当使用的中断信号线空闲时，标准的探测机制才能奏效；但如果被其它的具有共享特性的驱动程序占用的话，那么即使你的程序已经可以正常工作了，探测也会失败。

那么，唯一的可以用来探测共享中断信号的技术就是 DIY 探测。驱动程序必须为所有可能的中断信号线申请共享处理函数，然后观察中断在何处报告。这里和前面介绍的 DIY 的探测之间的差别在于，此时探测处理函数必须检查是否真的发生了中断，因为为响应共享中断信号线上的其它设备的中断它可能已经被调用过了。

释放处理函数同样是通过执行 *release_irq* 来实现的。这里 **dev_id** 参数用于从该中断的共享处理函数列表中正确地选出要释放的那个处理函数。这就是 **dev_id** 指针必须唯一的原因。

使用共享处理程序的驱动程序时还要小心：不能使用 *enable_irq* 和 *disable_irq*。如果它使用了这两个函数，共享中断信号线的其它设备就无法正常工作了。一般地，程序员必须牢记他的驱动程序并不独占这个中断，因此它的行为必须比独占中断信号线时更“社会化”些。

运行处理函数

如上所述，当内核接收到中断时，所有注册过的处理函数都会被激活。共享中断处理程序必须能将需要处理的中断和其它设备产生的中断区分开来。

装载 *short* 时指定 **shared=1** 将安装下面的处理程序而不是缺省的处理程序：

```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value;
    struct timeval tv;

    /*  如果不是 short，立即返回  */
    value = inb(short_base);
    if (!(value & 0x80)) return;

    /*  清除中断位  */
    outb(value & 0x7F, short_base);

    /*  其余不变  */
```

```
        do_gettimeofday(&tv);
        short_head += sprintf((char *)short_head,"%08u.%06u\n",
                                    (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
        if (short_head == short_buffer + PAGE_SIZE)
            short_head = short_buffer; /*  绕回来  */

        wake_up_interruptible(&short_queue); /*  唤醒所有读进程  */
    }
```

解释如下。因为并口没有 “待处理的中断”位可供检查，为此处理函数使用了 **ACK** 位。如果该位为高，报告的中断就是送给 *short* 的，并且处理函数将清除该位。

处理函数是通过将并口的数据端口的高位清零来清除中断位的－*short* 假定并口的 9 和 10 引脚是连在一起的。如果与 short 共享同一中断的设备产生了一个中断，*short* 会知道它的信号线并未激活，因此什么也不会做。

显然，*真正的*驱动程序做的工作会更多些；特别的，它要使用 **dev_id** 参数来得到自己的硬件结构。

特性完全的驱动程序可能会将工作划分为上半部和下半部，但这很容易添加，对实现共享的代码并无太大影响。

/proc 接口

系统中安装的共享中断处理程序不会影响*/proc/stat* 文件(该文件甚至并不知道处理程序的存在)。但是，*/proc/interrupts* 文件会有些变化。

为同一个中断号安装的处理程序会出现在*/proc/interrupts* 文件的同一行上。下面的快照取自我的计算机，是在我将 *short* 和我的帧捕捉卡装载为共享中断处理程序之后：

```
 0:       1153617    timer
 1:         13637    keyboard
 2:             0    cascade
 3:         14697 +  serial
 5:        190762    NE2000
 7:          2094 +  short, + cx100
13:             0    math error
14:         47995 +  ide0
15:         12207 +  ide1
```

这里共享中断信号是 IRQ7 号中断；激活的处理程序列在同一行，用逗号隔开。显然内核是无法区分 *short* 中断和捕捉卡(*cx100*)中断的。

中断驱动的 I/O

如果和处理的硬件间的数据传输因为某些原因会被延迟的话，那么驱动程序的写函数必须实现缓冲。数据缓冲可以将数据的发送和接收与 *write* 及 *read* 系统调用分离开来，提高系统的整体性能。

一个好的缓冲机制是“中断驱动的 I/O”，它在中断时间内填充一个输入缓冲区并由读设备的进程将其取空；或由写设备的进程来填充一个输入缓冲区并在中断时间内将其取空。

中断驱动的数据传输要正确进行，要求硬件必须安下面的语义产生中断：

- 对输入而言，当新数据到达，系统处理器准备读取它时，设备就中断处理器。实际执行的动作取决于设备是否使用了 I/O 端口，内存映射或者 DMA。

- 对输出而言，当设备准备好接收新数据或对成功的数据传输进行确认时都会发出中断。内存映射和能进行 DMA 的设备通常是通过产生中断来通知系统它们的对缓冲区的处理已经结束。

read 或 *write* 调用时间和实际的数据到达时间之间的关系是在第 5 章“*字符设备驱动程序的扩展操作*”的“阻塞型和非阻塞型操作”一节中介绍的。中断驱动的 I/O 引入了共享数据项的并发进程间的同步问题，因此所有这些问题都与竞争条件有关。

竞争条件

当变量或其它数据项在中断时间内被修改时，由于竞争条件的存在，驱动程序的操作就有可能造成它们的不一致。当操作不是原子地执行时，竞争条件就会发生，但在执行时仍假定数据会保持一致性。因此“竞争”是在非原子性的操作和其它可能被同时执行的代码之间发生的。典型的，竞争条件会在三种情况下发生：在函数内隐式地调用 *schedule*，阻塞操作和由中断代码或系统调用访问共享数据。最后一种情况发生得最频繁，因此我们在这一章处理竞争条件。

处理竞争条件是编程时最麻烦的一部分，因为相关的臭虫满足的条件很苛刻，不容易再现，很难分辨出中断代码和驱动程序的方法间是否存在竞争条件。程序员必须极为小心地避免数据或元数据的冲突。

一般用于避免竞争条件的技术是在驱动程序的方法中实现的，这些方法必须保证当数据项受到没有预料到的修改时得到正确的处理。但另一方面，中断处理函数并不需要特别的处理，因为相对设备的方法，它的操作是原子性的。

可以使用不同的技术来防止数据冲突，我下面将介绍最常用的一些技术。我不给出完整的代码，因为各种情况下最好的实现代码取决于被驱动的设备的操作模式以及程序员的不同爱好。

最常用的防止数据被并发地访问的方法有：

- 使用循环缓冲区和避免使用共享变量。
- 在访问共享变量的方法里暂时禁止中断。
- 使用锁变量，它是原子地增加和减少的。

当访问可能在中断时间内被修改了的变量时，不论你选用的是哪种方法，都必须决定如何进行处理。这样的变量可以声明为 **volatile** 的，来阻止编译器对该值的访问进行优化(例如，它阻止编译器在整个函数的运行期内将这个值放进一个寄存器中)。但是，使用 **volatile** 变量后，编译器产生的代码会很糟糕，因此你可能会转向使用 *cli* 和 *sti*。Linux 实现这些函数时使用了 *gcc* 的制导来保证在中断标志位被修改之前处理器处于安全状态。

使用循环缓冲区

使用循环缓冲区是处理并发访问问题的一种有效方法：当然最好的处理方法还是不允许并发访问。

循环缓冲区使用了一种被称为“生产者和消费者”的算法－一个进程将数据放进缓冲区中，另一个则将它取出来。如果只有一个生产者和一个消费者，那就避免了并发访问。在 *short* 模块中有两个生产者和消费者的例子。其中一个情形是，读进程等待消费在中断时间里生产的数据；而另一个情形是，下半部消费上半部生产的数据。

共有两个指针用于对循环缓冲区进行寻址：**head** 和 **tail**。**head** 是数据的写入位置，由数据的生产者更新。数据从 **tail** 处读出，它是由消费者更新的。正如我上面提到的，如果数据是在中断时间内写的，那么多次访问 **head** 多次时就必须小心。你必须将 **head** 定义成 **volatile** 的或者在进入竞争条件前将中断禁止。

循环缓冲区在填满前工作的很好。如果缓冲区满了，就可能出问题，但你可以有多种不同的解决方法可供选择。*short* 中的实现就是简单地丢弃数据；并不检查溢出，如果 **head** 超过了 **tail**，那么整个缓冲区中的数据都丢失了。其它的实现还有丢弃最后那个数据项；覆盖缓冲区的 **tail**，*printk* 是这么实现的(参见第 4 章的“消息是如何记录的”一节)；或者阻塞生产者，*scullpipe* 是这么实现的；或者分配一个临时的附加的缓冲区作为主力缓冲区的候补。最好的解决方案取决于数据的重要性和其它一些具体情况下的问题，所以我就不在这讨论了。

虽然循环缓冲区看来解决了并发访问的问题，但当 *read* 函数进入睡眠时仍有出现竞争条件的可能。下面的代码给出 *short* 中这个问题出现的位置：

```
while (short_head==short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

执行这个语句时，新数据有可能在**while**条件被测试是否为真*后*和进程进入睡眠*前*到达。中断中携带的信息就无法被进程及时读取；因此即使此时 **head != tail** 进程也将进入睡眠，直到下一项数据到达时它才会被唤醒。

我并没有为 *short* 实现正确的锁，因为 *short_read* 的源码在第 8 章的“驱动程序样例”一节中就包括了，当时还没有讨论到这一点。而且，short 处理的数据也不值得我们为它这么做。

尽管 *short* 收集的数据并不重要，而且在连续的两条指令时间间隔内发生中断的可能性小到可以忽略，但是有些时候你还是不能在还有待处理的数据时冒险地进入睡眠。

但这个问题一般来说还是值得对它进行特别的处理的，我们将它留到本章后面的“无竞争地进入睡眠”一节，那里我将会更详细地进行讨论。

值得注意的是，循环缓冲区只能处理生产者和消费者的情形。程序员必须经常地通过更复杂的数据结构来解决并发访问的问题。生产者/消费者的情形实际上是这些问题中最简单的一种；其它的数据结构，比如象链接表，就不能简单地使用循环缓冲区的实现方案。

禁止中断

获得对共享数据独占访问的通用方法是调用 *cli* 来禁止处理器的中断报告。当数据项(例如链接表)在中断时间内要被修改*并且*是被生存于正常的计算流中的函数修改时，那么随后的函数在访问这些数据前就必须先禁止中断。

这种情况下，竞争条件会发生在读共享数据项的指令和使用刚获得与数据有关的信息的指令之间。例如，如果链接表在中断时间内被修改过了，那么下面的循环在读这个表时就可能会失败。

```
for (ptr=listHead; ptr; ptr=ptr->next)
    /* do somthing */;
```

在 **ptr** 已经被读取后但在使用它之前，一个中断可能会改变了 **ptr** 的值。如果发生了这种情况，你一使用 **ptr** 就会有问题，因为这个指针当前的值与链接表已经没有关系了。

一个可能的解决的方法就是在整个关键循环期间都将中断禁止。虽然禁止中断的代码早在第 2 章的“ISA 内存”一节中就已经引入了，但仍值得在这里再重复一遍：

```
unsigned long flags;
save_flags(flags);
cli();
/*  临界区代码  */
restore_flags(flags);
```

实际上，在驱动程序的方法中，可以就用简单的 *cli/sti* 对来替代，因为你可以认为当进程进入系统调用时中断会被打开。但是，在要被其它代码所调用的代码中，你不得不使用更安全的 *save_flags/restore_flags* 解决方法，因为此时无法确定中断标志位(IF) 当前的值。

使用锁变量

共享数据变量的第三种方法是使用使用原子指令进行访问的锁。当两个无关的实体(比如象中断处理程序和 read 系统调用，或者是 SMP 对称多处理器计算机中的两个处理器)需要并发地对共享的数据项进行访问时，它们必须先申请锁。如果得不到锁，它就必须等待。

Linux 内核开放了两套函数来对锁进行处理：位操作和对“原子性”数据类型的访问。

位操作

经常的，我们要使有单个位的锁变量或者要在中断时间内更新设备状态位－而进程可能正在访问它们。内核为此提供了一套原子地修改和测试位的函数。因为整个操作是单步完成的，因此不会介入任何中断。

原子性的位操作运行的很快，因为它们通常不禁止中断，使用单条机器指令来完成相应操作。这些函数与体系结构相关，在头文件**<asm/bitops.h>**中声明。即使在 SMP 机器上它们也能保证是原子的，因此是推荐的保持处理器间一致性的方式。

不幸的是，这些函数的数据类型也是体系结构相关的。**nr** 参数和返回值在 Alpha 上是 **unsigned long** 类型，而在其它体系结构上是 **int** 类型。下面的列表描述了 1.2 到 2.1.37 各版的位操作形式。但该列表在 2.1.38 版中有了改变，详情可参见第 17 章“*近期发展*”的“位操作”一节。

set_bit(nr, void *addr);

这个函数用于设置 **addr** 指向的数据项的第 **nr** 个位。该函数作用在一个 **unsigned long** 上，即使 **addr** 指向 **void**。返回的是该位原先的取值－0 或非零。

clear_bit(nr, void *addr);

这个函数用于清除 **addr** 指向的 **unsigned long** 数据中的指定位。它的语义和 *set_bit* 类似。

change_bit(nr, void *addr);

这个函数用于切换指定位，其它方面和前面的 *set_bit* 和 *clear_bit* 函数类似。

test_bit(nr, void *addr);

这个函数是唯一一个不必是原子的为操作；它只是简单地返回该位当前的值。

当这些函数用于访问和修改共享的位时，你只要调用它们即可。而使用位操作来管理控制共享变量访问的锁变量，则更复杂些，需要举一个例子。

要访问共享数据项的代码段可以使用 *set_bit* 或 *clear_bit* 来试着原子地获取锁。通常是象下面的代码段这样实现的；假定锁位于地址 **addr** 的第 **nr** 位上。并且假定当锁空闲时该位为 0，锁忙时该位非零。

```
/*  试着设置锁  */
while (set_bit(nr,addr)!=0)
      wait_for_a_while();

/*  做你的工作  */

/*  释放锁，并检查... */
if (clear_bit(nr,addr)==0)
      something_wnt_wrong(); /*  已经被释放了：出错  */
```

这种访问共享数据的方式的毛病是竞争双方都必须要等待。如果其中一方是中断处理程序，那么这一点就较难保证了。

原子性的整数操作

内核程序员经常需要在中断处理程序和其它函数间共享整数变量。我们刚才已经看到对位的原子访问还不足以保证一切都能运行正常(对前面的例子来说，如果一方是一个中断处理函数的话，那就必须使用 *cli*)。

实际上，防止竞争条件的需要是如此迫切，以致于内核的开发者为这个问题专门实现了一个头文件：**<asm/atomic.h>**。这个头文件比较新，Linux 1.2 中就没有提供。因此，需要向后兼容的驱动程序是不能使用的。

atomic.h 中提供的函数比刚才介绍的那些位操作功能更强大。*atomic.h* 中定义了一种新的数据类型，**atomic_t**，只能通过原子操作来访问它。

atomic_t 目前在所有支持的体系结构上都被定义为 **int**。下面的操作是为这个数据类型所定义的，能保证 SMP 机器上的所有处理器是原子地对它进行访问。这些操作都非常快，因为它们都尽可能编译成单条的机器指令。

void atomic_add(atomic_t i, atomic_t *v);

将 **v** 指向的原子变量加上 **i**。返回值是 **void** 类型，大部分时候没有必要知道新值。网络部分的代码使用这个函数来更新套接字在内存使用上的统计信息。

void atomic_sub(atomic_t i, atomic_t *v);

从***v** 里减去 **i**。在最新的 2.1 版的内核中这两个函数的参数 **i** 都声明成 **int** 类型，但这种改变主要是出于美观的需要，并不对源代码造成影响。

void atomic_inc(atomic_t *v);

void atomic_dec(atomic_t *v);

对原子变量加减 1。

int atomic_dec_and_test(atomic_t *v);

该函数是在 1.3.84 版的内核里加入的，用于跟踪引用计数。仅当变量***v** 在减 1 后取值为 0 时返回值为 0。

如上所述，只能使用上面这些函数来访问 **atomic_t** 类型的数据。如果你将原子数据项传递给了一个要求参数类型为整型的函数，编译时就会得到警告。不用说，可以读取原子数据项的当前值并将它强制转换成其它数据类型。

无竞争地进入睡眠

在讨论进入睡眠的问题中我们曾忽略了一个竞争条件。这个问题实际上要比中断驱动的 I/O 问题更普遍，而有效的解决方案需要对 *sleep_on* 的实现内幕有些了解。

这种特别的竞争条件发生在检查进入睡眠的条件和对 *sleep_on* 的实际调用之间。下面的测试代码和前面使用的代码是一样的，但我觉得还是值得再在这里列出：

```
while (short_head==short_tail){
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

如果要安全地进行比较和进入睡眠，你必须*先*禁止中断报告，*然后*测试条件并进入睡眠。因此，比较中被测试的变量不会被修改。内核允许进程在发出 *cli* 指令后就进入睡眠。而在将进程插入它的等待队列之后，在调用 *shcedule* 之前，内核只要简单地重新打开中断报告就可以了。

这里给出的例子代码使用了 **while** 循环，由该循环来进行信号处理。如果有阻塞的信号向进程发出报告，*interruptible_sleep_on* 就返回，再次进行 **while** 语句中的测试。

下面是一种可能的实现：

```
while (short_head==short_tail){
    cli();
    if (short_head==short_tail)
        interuptible_sleep_on(&short_queue);
    sti();
    /* ... 信号解码 .... */
}
```

如果中断是在 *cli* 后发生的，那么这个中断在当前进程进入睡眠前都会处于待处理状态。而当中断最终报告给处理器时，进程已经进入了睡眠，可以被安全地唤醒。

在这个例子中，我可以使用 *cli/sti*，是因为设计的这段范例代码存在于 *read* 方法内的；否则我们必须使用更为安全的 *save_flags*，*cli*，和 *restore_flags* 函数。

如果在进入睡眠之前你不想禁止中断，那么还有另一种方法来完成与上面相同的任务(Linus 非常喜欢用这种方法)。但是，如果你愿意的话你可以跳过下面的讨论，因为下面的讨论的确有点太细了。

该方法的基本想法是，进程可以把自己排进等待队列，声明自己的状态为睡眠状态，*然后*执行它的测试代码。

典型的实现如下：

```
struct wait_queue wait = {current, NULL};

add_wait(&short_queue, &wait);
current->state=TASK_INTERRUPTIBLE;
while (short_head==short_tail){
    schedule();
    /* ... 信号解码 ... */
}
remove_wait_queue(&short_queue, &wait);
```

这段代码看起来有点象将 *sleep_on* 的内部实现展开了。显式地声明了 **wait** 变量，因为需要用它来使进程进入睡眠；这一切是在第 5 章的“等待队列”一节中解释的，但这个例子中引入了一些新的符号。

current->state

这个字段是给调度器用的提示。调度器被激活后，它将通过观察所有进程的 **state** 字段来决定接着作些什么。所有进程都可以任意修改自己的 **state** 字段，但在调度器运行之前这种改变还不会生效。

#include <linux/sched.h>
TASK_RUNNING
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE

这些符号名代表了 **current->state** 最经常取的一些值。**TASK_RUNNING** 表示进程正在运行，其它两个表示进程正在睡眠。

void add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __ remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

这些函数用于从等待队列中插入和删除进程。wait 参数必须指向进程堆栈所在的页(临

时变量)。以下划线开头的函数运行的更快些，但它们在禁止中断后才能被调用(例如，在快速中断处理程序内)。

有了这些背景知识，下面让我们看看当中断到达时会发生什么。此时处理程序将调用 **wake_up_interruptible(&short_queue)**；对 Linux 而言，这意味着“将 **state** 置为 **TASK_RUNNING**”。因此，如果在 **while** 条件和 *schedule* 调用间有中断报告的话，该任务的 state 字段将会又被标记为 **TASK_RUNNING** 的，因此不会丢失数据。

而如果进程仍是“可中断的”(**TASK_INTERRUPTIBLE**)，*schedule* 将保持它的睡眠状态。

值得注意的是，*wake_up* 系统调用并不会将进程从等待队列中删去。是由 *sleep_on* 来对等待队列进行进程的添加和删除的。因此程序代码必须显式地调用 *add_wait_queue* 和 *remove_wait_queue*，因为这种情况下不再使用 *sleep_on* 了。

中断处理的版本相关性

不是所有本章引入的代码都能向后兼容地移植到 Linux 1.2 上的。在此我将列出主要的差异并对如何处理这些差异提出建议。实际上，*short* 在 2.0.x 和 1.2.13 版的内核上都编译和运行得很好。

request_irq 函数的不同原型

我在这一整章中使用的给 *request_irq* 函数传递参数的方式都是到 1.3.70 版的内核才引入的，因为是到这个版本才出现了共享中断处理程序的。

更早的内核版本并不需要 **dev_id** 参数，原型也相对简单些：

```
int request_irq(unsigned int irq,
                void (*handler)(int, struct pt_regs *),
                unsigned long flags, const char *device);
```

只要使用下面的宏定义(注意早期的版本中 *free_irq* 也没有 **dev_id** 参数)，新的语义可以很容易地强加在旧原型上：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,70)
    /*  预处理器必须能处理递归的定义  */
#   define request_irq(irq,fun,fla,nam,dev) request_irq(irq,fun,fla,nam)
#   define free_irq(irq,dev)                free_irq(irq)
#endif
```

这些宏只是简单地丢弃额外的 **dev** 参数。

处理函数原型上的差异通过显式的**#if/#else/#endif** 语句得到很好的处理。如果你使用了

dev_id 指针，旧内核的条件分支可以将它申明为 **NULL** 变量，这样处理函数体就可以对 **NULL** 设备指针进行处理了。

short 模块中的一个例子可以作为这种想法的范例：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,70)
void short_sh_interrupt(int irq, struct pt_regs *regs)
{
  void *dev_id = NULL;
#else
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
#endif
```

探测中断信号线

内核到 1.3.30 版开始开放探测函数。如果你希望你的驱动程序能移植到旧的内核上，你将不得不实现 DIY 检测。实际上早在 1.2 版的内核，这些函数就已经存在了，只是模块化的驱动程序无法使用罢了。

这样，移植中断处理函数就没有什么其它的问题了。

快速参考

本章引入了下面这些与中断管理有关的符号：

#include <linux/sched.h>
int request_irq(unsigned int irq, void (*handler)());
unsigned long flags, const char *device, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);

这些系统调用用于注册和注销中断处理程序。低于 2.0 版的内核不提供 **dev_id** 参数。

SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM

这些是 *request_irq* 函数的各种选项。**SA_INTERRUPT** 请求安装快速中断处理程序(相对于慢速处理函数)。**SA_SHIRQ** 安装共享中断处理函数，而第三种选项表明产生的中断的时间戳对系统熵池(entropy pool)有贡献。

/proc/interrupts
/proc/stat

这些文件系统节点用于报告关于硬件中断和安装的处理函数的信息。

unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);

当驱动程序需要探测设备使用哪根中断信号线时，可以使用这些函数。在中断产生之后，*probe_irq_on* 的返回值必须传回给 *probe_irq_off*。*probe_irq_off* 的返回值就是检测到的中断号。

void disable_irq(int irq);
void enable_irq(int irq);

驱动程序可以启动和禁止中断报告。禁止中断后，硬件产生的中断都将丢失。在上半部处理程序中调用这些函数则没有任何效果。而使用共享中断处理程序的驱动程序决不能使用这些函数。

#include <linux/interrupt.h>
void mark_bh(int nr);

这些函数用于标记要执行的下半部。

#include <asm/bitops.h>
set_bit(nr, void *addr);
clear_bit(nr, void *addr);
change_bit(nr, void *addr);
test_bit(nr, void *addr);

这些函数用于原子性地访问位的值；它们可作用于标志位和锁变量。使用这些函数避免了所有与对位的并发访问有关的竞争条件。

#include <asm/atomic.h>
typedef int atomic_t;
void atomic_add(atomic_t i, atomic_t *v);
void atomic_sub(atomic_t i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);

这些函数用于原子地访问整数变量。如果想让编译时不出现警告信息，必须只使用这些函数来访问 **atomic_t** 类型的变量。

#include <linux/sched.h>
TASK_RUNNING
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE

这些是 **current->state** 最经常取的一些值。它们是给 *schedule* 用的提示。

void add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __ remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

这些是使用等待队列的最底层的函数。打头的下划线标志该函数是底层的函数，使用后两个函数时处理器必须已经禁止了中断报告。