第十章 中断处理

虽然有些设备可以仅通过它们的 I/O 区域进行控制，但大多数真实的设备比这更复杂一些。设备必须处理外部世界，这通常包括旋转的磁盘、移动的磁带、通往远方的电线等等。许多事情必须在一个与处理器不同且远慢于处理器的时间框架内完成。由于处理器几乎总是不希望等待外部事件，所以必须有一种方式让设备在发生某事时通知处理器。

当然，这种方式就是中断。中断只是硬件在需要处理器注意时可以发送的一个信号。Linux 处理中断的方式与它在用户空间处理信号的方式大致相同。大多数情况下，驱动程序只需要为其设备的中断注册一个处理程序，并在它们到达时正确处理它们。当然，在这个简单的画面下面有一些复杂性；特别是，由于它们的运行方式，中断处理程序在它们可以执行的操作上有些限制。

没有真正的硬件设备来生成它们，就很难演示中断的使用。因此，本章使用的示例代码与并行端口一起工作。这样的端口在现代硬件上开始变得稀缺，但是，幸运的是，大多数人仍然能够得到一个带有可用端口的系统。我们将使用上一章的 short 模块；通过一些小的添加，它可以生成并处理来自并行端口的中断。模块的名称，short，实际上意味着 short int（这是 C，不是吗？），以提醒我们它处理中断。

然而，在我们进入这个话题之前，现在是时候给出一个警告。由于它们的性质，中断处理程序与其他代码并发运行。因此，它们不可避免地引发了并发性和对数据结构和硬件的争用问题。如果你屈服于跳过第 5 章的讨论的诱惑，我们理解。但我们也建议你现在回头再看一遍。当处理中断时，对并发控制技术的深入理解是至关重要的。

10.1. 准备并口Parallel Port

虽然并行接口简单，但它可以触发中断。打印机使用这个功能通知 lp 驱动程序它已经准备好接受缓冲区中的下一个字符。

像大多数设备一样，并行端口在被指示生成中断之前实际上并不生成中断；并行标准规定，设置端口 2（0x37a，0×27a，或其他）的位 4 启用中断报告。short 在模块初始化时执行一个简单的 outb 调用来设置这个位。

一旦启用了中断，每当电气信号在引脚 10（所谓的 ACK 位）从低变为高时，并行接口就会生成一个中断。强制接口生成中断的最简单方法（除了将打印机连接到端口外）是连接并行连接器的引脚 9 和 10。在系统后面的并行端口连接器中插入一段短线就可以创建这个连接。并行端口的引脚图如图 9-1 所示。

引脚 9 是并行数据字节的最高有效位。如果你向 /dev/short0 写入二进制数据，你会生成几个中断。然而，向端口写入 ASCII 文本不会生成任何中断，因为 ASCII 字符集没有设置顶部位的条目。

如果你不愿意将引脚连在一起，但你手头有一台打印机，你可以使用真正的打印机运行示例中断处理程序，如后面所示。然而，请注意，我们介绍的探测函数依赖于引脚 9 和 10 之间的跳线，你需要它来使用我们的代码进行探测实验。

10.2. 安装一个中断处理

你想真正“看到”中断被生成，仅仅写入硬件设备是不够的；系统中必须配置一个软件处理程序。如果 Linux 内核没有被告知期望你的中断，它只会确认并忽略它。

中断线是一种宝贵且经常有限的资源，特别是当它们只有 15 或 16 条时。内核保持了一个中断线的注册表，类似于 I/O 端口的注册表。一个模块在使用中断通道（或 IRQ，用于中断请求）之前应该请求它，并在完成时释放它。在许多情况下，模块也应该能够与其他驱动程序共享中断线，如我们将看到的。上面的函数，
在 `<linux/interrupt.h>` 中声明，实现了中断注册接口。

```c
#include <linux/interrupt.h>

// 请求中断
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
void free_irq(unsigned int irq, void *dev_id);
```

从 request_irq 返回给请求函数的值要么是 0 表示成功，要么是负的错误代码，这是常见的。这个函数返回 -EBUSY 来表示另一个驱动程序已经在使用请求的中断线是很常见的。函数的参数如下：

- `unsigned int irq` 请求的中断号。

- `irqreturn_t (*handler)(int, void *, struct pt_regs *)` 被安装的处理函数的指针。我们将在本章后面讨论这个函数的参数和返回值。

- `unsigned long flags` 如你所料，这是一个与中断管理相关的选项的位掩码（稍后描述）。

- `const char *dev_name` 传递给 request_irq 的字符串用于在 /proc/interrupts 中显示中断的所有者（参见下一节）。

- `void *dev_id` 用于共享中断线的指针。它是一个唯一的标识符，用于释放中断线，也可能被驱动程序用来指向其自己的私有数据区域（用来识别哪个设备正在中断）。如果中断不是共享的，dev_id 可以设置为 NULL，但无论如何使用这个项指向设备结构都是一个好主意。我们将在“实现处理程序”一节中看到 dev_id 的实际用途。

可以在 flags 中设置的位如下：

- `SA_INTERRUPT` 当设置时，这表示一个“快速”中断处理程序。快速处理程序在当前处理器上禁用中断执行（这个主题在“快速和慢速处理程序”一节中介绍）。

- `SA_SHIRQ` 这个位表示中断可以在设备之间共享。共享的概念在“中断共享”一节中概述。

- `SA_SAMPLE_RANDOM` 这个位表示生成的中断可以贡献给 /dev/random 和 /dev/urandom 使用的熵池。这些设备在读取时返回真正的随机数，旨在帮助应用软件选择加密的安全密钥。这样的随机数是从由各种随机事件贡献的熵池中提取的。如果你的设备在真正随机的时间生成中断，你应该设置这个标志。如果，另一方面，你的中断是可预测的（例如，帧抓取器的垂直消隐），那么这个标志就不值得设置——它无论如何都不会贡献给系统熵。那些可能受到攻击者影响的设备不应设置这个标志；例如，网络驱动程序可能会受到来自外部的可预测的数据包定时，因此不应贡献给熵池。有关更多信息，请参见 drivers/char/random.c 中的注释。

中断处理程序可以在驱动程序初始化时或设备首次打开时安装。虽然在模块的初始化函数中安装中断处理程序听起来像是一个好主意，但往往并不是，特别是如果你的设备不共享中断。因为中断线的数量是有限的，你不想浪费它们。你很容易在你的计算机中有更多的设备，而中断的数量却不够。如果一个模块在初始化时请求一个 IRQ，它会阻止

任何其他驱动程序使用中断，即使持有它的设备从未被使用过。另一方面，当设备打开时请求中断，可以允许一些资源共享。

例如，只要你不同时使用这两个设备，就可以在同一个中断上运行一个帧抓取器和一个调制解调器。用户在系统启动时加载特殊设备的模块是很常见的，即使这个设备很少被使用。一个数据采集设备可能使用与第二个串行端口相同的中断。虽然在数据采集期间避免连接到你的互联网服务提供商（ISP）并不太难，但被迫卸载一个模块以使用调制解调器确实很不愉快。

在设备首次打开时，即在硬件被指示生成中断之前，调用 request_irq 是正确的地方。在设备最后一次关闭时，即在硬件被告知不再中断处理器之后，调用 free_irq 是正确的地方。这种技术的缺点是你需要保持每个设备的打开计数，以便知道何时可以禁用中断。

尽管有这个讨论，short 在加载时请求其中断线。这样做是为了让你可以运行测试程序，而不必运行额外的进程来保持设备打开。因此，short 在其初始化函数（short_init）中请求中断，而不是在 short_open 中，就像一个真正的设备驱动程序那样。

以下代码请求的中断是 short_irq。变量的实际赋值（即，确定要使用哪个 IRQ）将在后面显示，因为它与当前的讨论无关。short_base 是正在使用的并行接口的基本 I/O 地址；接口的寄存器 2 被写入以启用中断报告。

```c
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 { /* actually enable it -- assume this *is* a
parallel port */

        outb(0x10,short_base+2);

    }

}
```

代码显示正在安装的处理程序是一个快速处理程序（SA_INTERRUPT），不支持中断共享（缺少 SA_SHIRQ），并且不贡献系统熵（也缺少 SA_SAMPLE_RANDOM）。然后 outb 调用启用并行端口的中断报告。

值得一提的是，i386 和 x86_64 架构定义了一个函数用于查询中断线的可用性：

```c
int can_request_irq(unsigned int irq, unsigned long
flags);
```

如果尝试分配给定的中断成功，此函数返回非零值。然而，请注意，可以在调用 can_request_irq 和 request_irq 之间的任何时候更改事情。

10.2.1. /proc 接口

每当硬件中断到达处理器时，内部计数器就会增加，提供了一种检查设备是否按预期工作的方法。报告的中断显示在 /proc/interrupts 中。以下快照是在一个双处理器 Pentium 系统上拍摄的：

```
root@montalcino:/bike/corbet/write/ldd3/src/short# cat /pr
oc/interrupts

           CPU0       CPU1

  0:    4848108         34    IO-APIC-edge  timer

  2:          0          0          XT-PIC  cascade

  8:          3          1    IO-APIC-edge  rtc

 10:       4335          1   IO-APIC-level  aic7xxx

 11:       8903          0   IO-APIC-level  uhci_hcd

 12:         49          1    IO-APIC-edge  i8042

NMI:          0          0

LOC:    4848187    4848186

ERR:          0

MIS:          0
```

第一列是 IRQ 号。你可以从缺失的 IRQs 看出，该文件只显示对应于已安装处理程序的中断。例如，第一个串行端口（使用中断号 4）没有显示，这表明调制解调器没有被使用。实际上，即使调制解调器之前已经被使用过，但在快照时没有被使用，它也不会出现在文件中；串行端口的行为良好，当设备关闭时，它们会释放其中断处理程序。

/proc/interrupts 显示已经传递给系统上每个 CPU 的中断数量。从输出中你可以看到，Linux 内核通常在第一个 CPU 上处理中断，以最大化缓存局部性。最后两列提供了处理中断的可编程中断控制器的信息（驱动程序编写者不需要担心），以及为中断注册处理程序的设备的名称（如在 request_irq 的 dev_name 参数中指定）。

/proc 树包含另一个与中断相关的文件，/proc/stat；有时你会发现一个文件更有用，有时你会更喜欢另一个。/proc/stat 记录了关于系统活动的几个低级统计信息，包括（但不限于）自系统启动以来接收的中断数量。stat 的每一行都以一个文本字符串开始，这是行的关键；我们正在寻找的是 intr 标记。以下（截断的）快照是在前一个快照之后不久拍摄的：

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0
0
```

第一个数字是所有中断的总数，而其他每一个数字都代表一个单独的 IRQ 线，从中断 0 开始。所有的计数都是在系统中所有处理器上求和的。这个快照显示中断号 4 已经被使用了 4907 次，尽管当前没有安装处理程序。如果你正在测试的驱动程序在每个打开和关闭周期中获取和释放中断，你可能会发现 /proc/stat 比 /proc/interrupts 更有用。

两个文件之间的另一个区别是 interrupts 不依赖于架构（除了，也许，最后的几行），而 stat 是；字段的数量取决于内核底层的硬件。可用的中断数量从 SPARC 的 15 个到 IA-64 和其他一些系统的 256 个不等。有趣的是，注意到在 x86 上定义的中断数量目前是 224，而不是你可能期望的 16；这是因为，如在 include/asm-i386/irq.h 中解释的，Linux 使用的是架构限制，而不是实现特定的限制（如旧式 PC 中断控制器的 16 个中断源）。

以下是在 IA-64 系统上拍摄的 /proc/interrupts 的快照。如你所见，除了常见中断源的不同硬件路由外，输出与前面显示的 32 位系统的输出非常相似。

```
           CPU0       CPU1
 27:       1705      34141  IO-SAPIC-level  qla1280
 40:          0          0  SAPIC
perfmon
 43:        913       6960  IO-SAPIC-level  eth0
 47:      26722        146  IO-SAPIC-level  usb-uhci
 64:          3          6  IO-SAPIC-edge   ide0
 80:          4          2  IO-SAPIC-edge   keyboard
 89:          0          0  IO-SAPIC-edge   PS/2 Mouse
239:    5606341    5606052           SAPIC   timer

254:  67575  52815  SAPIC  IPI
NMI:  0  0
ERR:  0
```

10.2.2. 自动检测 IRQ 号

对于驱动程序在初始化时面临的最具挑战性的问题之一可能是如何确定设备将使用哪条 IRQ 线。驱动程序需要这个信息来正确安装处理程序。尽管程序员可以要求用户在加载时指定中断号，但这是一种不好的做法，因为大多数时候用户不知道这个数字，要么是因为他没有配置跳线，要么是因为设备没有跳线。大多数用户希望他们的硬件“就能工作”，并且对中断号这样的问题不感兴趣。所以，自动检测中断号是驱动程序可用性的基本要求。

有时自动检测依赖于一些设备具有默认行为的知识，这些行为很少（如果有的话）改变。在这种情况下，驱动程序可能会假设默认值适用。这正是 short 默认对并行端口的行为。实现很简单，如 short 本身所示：

```c
if (short_irq < 0) /* not yet specified: force the
default on */

    switch(short_base) {

        case 0x378: short_irq = 7; break;

        case 0x278: short_irq = 2; break;

        case 0x3bc: short_irq = 5; break;

    }
```

这段代码根据选择的基础 I/O 地址分配中断号，同时允许用户在加载时用类似以下的方式覆盖默认值：

```c
insmod ./short.ko irq=x
```

short_base 默认为 0x378，所以 short_irq 默认为 7。

一些设备在设计上更先进，简单地“宣布”它们将使用哪个中断。在这种情况下，驱动程序通过从设备的一个 I/O 端口或 PCI 配置空间读取状态字节来获取中断号。当目标设备是有能力告诉驱动程序它将使用哪个中断的设备时，自动检测 IRQ 号只意味着探测设备，无需额外的工作来探测中断。幸运的是，大多数现代硬件都是这样工作的；例如，PCI 标准通过要求外围设备声明它们将使用哪个中断线来解决这个问题。PCI 标准在第 12 章中讨论。

不幸的是，并非每个设备都对程序员友好，自动检测可能需要一些探测。这种技术非常简单：驱动程序告诉设备生成中断，并观察发生了什么。如果一切顺利，只有一条中断线被激活。

尽管在理论上探测很简单，但实际的实现可能不清楚。我们看两种执行任务的方式：调用内核定义的辅助函数和实现我们自己的版本。

10.2.2.1. 内核协助的探测

Linux 内核提供了一个用于探测中断号的低级设施。它只适用于非共享中断，但大多数能够在共享中断模式下工作的硬件提供了更好的方法来找到配置的中断号。该设施由两个函数组成，声明在 <linux/interrupt.h> 中（也描述了探测机制）：

```c
unsigned long probe_irq_on(void);
```

此函数返回未分配中断的位掩码。驱动程序必须保留返回的位掩码，并将其传递给 probe_irq_off。在此调用之后，驱动程序应安排其设备生成至少一个中断。

```c
int probe_irq_off(unsigned long);
```

在设备请求中断后，驱动程序调用此函数，将其参数设置为 probe_irq_on 之前返回的位掩码。probe_irq_off 返回在“probe_on”之后发出的中断的数量。如果没有发生中断，返回 0（因此，不能对 IRQ 0 进行探测，但在任何支持的架构上，没有自定义设备可以使用它）。如果发生了多个中断（模糊检测），probe_irq_off 返回一个负值。

程序员应该注意在调用 probe_irq_on 之后启用设备上的中断，并在调用 probe_irq_off 之前禁用它们。此外，你必须记住在 probe_irq_off 之后处理设备中的挂起中断。

short 模块演示了如何使用这种探测。如果你使用 probe=1 加载模块，只要并行连接器的 9 和 10 脚绑定在一起，就会执行以下代码来检测你的中断线：

```c
int count = 0;

do {

    unsigned long mask;

    mask = probe_irq_on( );

    outb_p(0x10,short_base+2); /* enable reporting */

    outb_p(0x00,short_base);   /* clear the bit */

    outb_p(0xFF,short_base);   /* set the bit: interrupt!
*/

    outb_p(0x00,short_base+2); /* disable reporting */

    udelay(5);  /* give it some time */

    short_irq = probe_irq_off(mask);

    if (short_irq == 0) { /* none of them? */

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

        short_irq = -1;

    }

    /*

     * if more than one line has been activated, the
result is

     * negative. We should service the interrupt (no need
for lpt port)
```

```
         * and loop over again. Loop at most five times, then
    give up

         */

    } while (short_irq < 0 && count++ < 5);

    if (short_irq < 0)

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

注意在调用 probe_irq_off 之前使用 udelay。根据你的处理器的速度，你可能需要等待一段时间，以便给中断时间实际传递。

探测可能是一个长时间的任务。虽然对于 short 来说不是这样，但例如探测一个帧抓取器需要至少延迟 20 毫秒（对于处理器来说是很长的时间），其他设备可能需要更长的时间。因此，最好只在模块初始化时探测一次中断线，无论你是在设备打开时安装处理程序（如你应该的那样），还是在初始化函数中（不推荐）。

有趣的是，注意到在一些平台上（PowerPC、M68k、大多数 MIPS 实现，以及两个 SPARC 版本）探测是不必要的，因此，前面的函数只是空的占位符，有时被称为“无用的 ISA 非理性”。在其他平台上，只为 ISA 设备实现了探测。无论如何，大多数架构都定义了函数（即使它们是空的）以便于移植现有的设备驱动程序。

10.2.2.2. Do-it-yourself 探测

探测也可以在驱动程序本身中实现，而不会带来太多麻烦。必须实现自己探测的驱动程序很少，但看看它是如何工作的可以对过程有一些了解。为此，如果使用 probe=2 加载 short 模块，它将执行自己的 IRQ 线检测。

机制与前面描述的相同：启用所有未使用的中断，然后等待看发生什么。然而，我们可以利用我们对设备的了解。通常，设备可以配置为从三或四个中选择一个 IRQ 号；只探测这些 IRQ 可以使我们检测到正确的一个，而无需测试所有可能的 IRQ。

short 实现假定 3、5、7 和 9 是唯一可能的 IRQ 值。这些数字实际上是一些并行设备允许你选择的值。

以下代码通过测试所有“可能”的中断并查看发生什么来进行探测。trials 数组列出了要尝试的 IRQ，并以 0 作为结束标记；tried 数组用于跟踪此驱动程序实际注册的处理程序。

```c
int trials[] = {3, 5, 7, 9, 0};

int tried[]  = {0, 0, 0, 0, 0};

int i, count = 0;

/*

 * install the probing handler for all possible lines.
Remember

 * the result (0 for success, or -EBUSY) in order to only
free

 * what has been acquired

 */

for (i = 0; trials[i]; i++)

    tried[i] = request_irq(trials[i], short_probing,

             SA_INTERRUPT, "short probe", NULL);

do {

    short_irq = 0; /* none got, yet */

    outb_p(0x10,short_base+2); /* enable */

    outb_p(0x00,short_base);

    outb_p(0xFF,short_base); /* toggle the bit */

    outb_p(0x00,short_base+2); /* disable */

    udelay(5);  /* give it some time */
```

```
    /* the value has been set by the handler */

    if (short_irq == 0) { /* none of them? */

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

    }

    /*

     * If more than one line has been activated, the
result is

     * negative. We should service the interrupt (but the
lpt port

     * doesn't need it) and loop over again. Do it at
most 5 times

     */

} while (short_irq <=0 && count++ < 5);

/* end of loop, uninstall the handler */

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 IRQ 值是什么。在这种情况下，你需要探测所有的空闲中断，而不是限制自己进行几次 trials[]。要探测所有中断，你必须从 IRQ 0 探测到 IRQ NR_IRQS-1，其中 NR_IRQS 在 <asm/irq.h> 中定义，是平台依赖的。

现在我们只缺少探测处理程序本身。处理程序的角色是根据实际接收到的中断来更新 short_irq。short_irq 中的 0 值表示“还没有”，而负值表示“模糊”。这些值被选择是为了与 probe_irq_off 保持一致，并允许相同的代码在 short.c 中调用任何一种探测。

```c
irqreturn_t short_probing(int irq, void *dev_id, struct
pt_regs *regs)

{

    if (short_irq == 0) short_irq = irq;    /* found */

    if (short_irq != irq) short_irq = -irq; /* ambiguous
*/

    return IRQ_HANDLED;

}
```

处理程序的参数将在后面描述。知道 irq 是正在处理的中断应该足以理解刚刚显示的函数。

10.2.3. 快速和慢速处理

早期版本的 Linux 内核非常努力地区分“快速”和“慢速”中断。快速中断是那些可以非常快速处理的中断，而处理慢速中断需要显著更长的时间。慢速中断可能对处理器的需求足够大，因此在处理它们时重新启用中断是值得的。否则，需要快速处理的任务可能会被延迟太久。

在现代内核中，快速和慢速中断之间的大多数差异已经消失。只剩下一个：快速中断（那些使用 SA_INTERRUPT 标志请求的中断）在当前处理器上执行时，所有其他中断都被禁用。注意，其他处理器仍然可以处理中断，尽管你永远不会看到两个处理器同时处理同一个 IRQ。

那么，你的驱动程序应该使用哪种类型的中断呢？在现代系统上，SA_INTERRUPT 只打算在一些特定情况下使用，如定时器中断。除非你有充分的理由在禁用其他中断的情况下运行你的中断处理程序，否则你不应使用 SA_INTERRUPT。

这个描述应该能满足大多数读者，尽管有一些对硬件有兴趣并且对她的计算机有一些经验的人可能对深入了解感兴趣。如果你不关心内部细节，你可以跳到下一节。

10.2.3.1. x86上中断处理的内幕

这个描述是从 2.6 内核中的 arch/i386/kernel/irq.c、arch/i386/kernel/apic.c、arch/i386/kernel/entry.S、arch/i386/kernel/i8259.c 和 include/asm-i386/hw_irq.h 中推断出来的；尽管一般概念仍然相同，但硬件细节在其他平台上有所不同。

最低级别的中断处理可以在 entry.S 中找到，这是一个汇编语言文件，处理许多机器级别的工作。通过一些汇编技巧和一些宏，一些代码被分配给每一个可能的中断。在每种情况下，代码将中断号推入堆栈，并跳转到一个公共段，该段调用在 irq.c 中定义的 do_IRQ。

do_IRQ 的第一件事是确认中断，以便中断控制器可以处理其他事情。然后，它为给定的 IRQ 号获取一个自旋锁，从而防止任何其他 CPU 处理此 IRQ。它清除了一些状态位（包括我们稍后将看到的一个叫做 IRQ_WAITING 的位），然后查找这个特定 IRQ 的处理程序。如果没有处理程序，就没有什么可做的；释放自旋锁，处理任何挂起的软件中断，然后 do_IRQ 返回。

然而，通常，如果一个设备正在中断，至少有一个处理程序为其 IRQ 注册。实际调用处理程序的是 handle_IRQ_event 函数。如果处理程序是慢速的（没有设置 SA_INTERRUPT），则在硬件中重新启用中断，并调用处理程序。然后就是清理，运行软件中断，然后回到正常工作。由于中断的结果，“正常工作”可能已经改变（例如，处理程序可以唤醒一个进程），所以从中断返回的最后一件事可能是处理器的重新调度。

通过为每个当前缺少处理程序的 IRQ 设置 IRQ_WAITING 状态位来探测 IRQ。当中断发生时，do_IRQ 清除该位，然后返回，因为没有注册处理程序。当驱动程序调用 probe_irq_off 时，只需要搜索不再设置 IRQ_WAITING 的 IRQ。

10.2.4. 实现一个Handler

到目前为止，我们已经学会了注册一个中断处理程序，但还不会编写一个。实际上，处理程序没有什么特别的——它是普通的 C 代码。

唯一的特殊之处在于处理程序在中断时间运行，因此在它可以做什么上有一些限制。这些限制与我们在内核计时器中看到的相同。处理程序不能将数据传输到用户空间或从用户空间传输，因为它不在进程的上下文中执行。处理程序也不能做任何会睡眠的事情，如调用 wait_event，使用除 GFP_ATOMIC 之外的任何东西分配内存，或锁定一个信号量。最后，处理程序不能调用 schedule。

中断处理程序的角色是向其设备反馈中断接收情况，并根据正在服务的中断的含义读取或写入数据。第一步通常包括在接口板上清除一个位；大多数硬件设备在其“中断挂起”位被清除之前不会生成其他中断。根据你的硬件如何工作，这一步可能需要在最后而不是首先执行；这里没有一刀切的规则。一些设备不需要这一步，因为它们没有“中断挂起”位；这样的设备是少数，尽管并行端口就是其中之一。因此，short 不需要清除这样的位。

中断处理程序的一个典型任务是唤醒在设备上睡眠的进程，如果中断信号了它们正在等待的事件，如新数据的到来。坚持使用帧捕获器示例，一个进程可以通过连续读取设备来获取一系列图像；在读取每个帧之前，read 调用都会阻塞，而中断处理程序会在每个新帧到达时立即唤醒进程。这假设捕获器中断处理器以信号成功到达每个新帧。

程序员应该小心编写一个在最小的时间内执行的例程，无论它是快速还是慢速的处理程序。如果需要进行长时间的计算，最好的方法是使用 tasklet 或 workqueue 在更安全的时间安排计算（我们将在“上半部和下半部”一节中看到如何以这种方式推迟工作）。

我们在 short 中的示例代码通过调用 do_gettimeofday 并将当前时间打印到一个页面大小的循环缓冲区来响应中断。然后它唤醒任何正在读取的进程，因为现在有数据可以读取了。

```c
irqreturn_t short_interrupt(int irq, void *dev_id, struct
pt_regs *regs)

{

    struct timeval tv;

    int written;

    do_gettimeofday(&tv);

        /* Write a 16 byte record. Assume PAGE_SIZE is a
multiple of 16 */

    written = sprintf((char *)short_head,"%08u.%06u\n",

            (int)(tv.tv_sec % 100000000), (int)
(tv.tv_usec));

    BUG_ON(written != 16);

    short_incr_bp(&short_head, written);

    wake_up_interruptible(&short_queue); /* awake any
reading process */

    return IRQ_HANDLED;

}
```

这段代码虽然简单，但代表了中断处理程序的典型工作。它反过来调用short_incr_bp，定义如下：

```c
static inline void short_incr_bp(volatile unsigned long
*index, int delta)

{

    unsigned long new = *index + delta;

    barrier( );  /* Don't optimize these two together */

    *index = (new >= (short_buffer + PAGE_SIZE)) ?
short_buffer : new;

}
```

这个函数被精心编写，以将一个指针包装到循环缓冲区中，而永远不会暴露出一个错误的值。barrier 调用在那里阻止编译器优化函数的其他两行。没有 barrier，编译器可能决定优化出 new 变量并直接赋值给 `* index`。这种优化在包装的情况下可能会暴露出 index 的错误值。通过小心防止不一致的值对其他线程可见，我们可以在不使用锁的情况下安全地操作循环缓冲区指针。

用于读取在中断时间填充的缓冲区的设备文件是 /dev/shortint。这个设备特殊文件，连同 /dev/shortprint，没有在第 9 章中介绍，因为它的使用是特定于中断处理的。/dev/shortint 的内部特别适用于中断生成和报告。写入设备每两个字节生成一个中断；读取设备给出每个中断报告的时间。

如果你将并行连接器的 9 和 10 针连接在一起，你可以通过提高并行数据字节的高位来生成中断。这可以通过向 /dev/short0 写入二进制数据或向 /dev/shortint 写入任何东西来实现。

以下代码实现了 /dev/shortint 的读和写：

```c
ssize_t short_i_read (struct file *filp, char __user
*buf, size_t count,

     loff_t *f_pos)

{

    int count0;

    DEFINE_WAIT(wait);

    while (short_head == short_tail) {

        prepare_to_wait(&short_queue, &wait,
TASK_INTERRUPTIBLE);

        if (short_head == short_tail)

            schedule( );

        finish_wait(&short_queue, &wait);

        if (signal_pending (current))  /* a signal
arrived */

            return -ERESTARTSYS; /* tell the fs layer to
handle it */

    }

    /* count0 is the number of readable data bytes */

    count0 = short_head - short_tail;

    if (count0 < 0) /* wrapped */

        count0 = short_head + PAGE_SIZE - short_tail;
```

```
    if (count0 < count) count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))

        return -EFAULT;

    short_incr_bp (&short_tail, count);

    return count;

}

ssize_t short_i_write (struct file *filp, const char
__user *buf, size_t count,

        loff_t *f_pos)

{

    int written = 0, odd = *f_pos & 1;

    unsigned long port = short_base; /* output to the
parallel data latch */

    void *address = (void *) short_base;

    if (use_mem) {

        while (written < count)

            iowrite8(0xff * ((++written + odd) & 1),
address);

    } else {

        while (written < count)

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

```
        }

        *f_pos += count;

        return written;

    }
```

另一个设备特殊文件，/dev/shortprint，使用并行端口驱动打印机；如果你想避免连接 D-25 连接器的 9 和 10 针，你可以使用它。shortprint 的写实现使用一个循环缓冲区来存储要打印的数据，而读实现就是刚刚显示的（所以你可以读取你的打印机吃掉每个字符所花费的时间）。

为了支持打印机操作，中断处理程序已经从刚刚显示的稍微修改了一下，增加了如果有更多数据要传输，就向打印机发送下一个数据字节的能力。

10.2.5. Handler的参数和返回值

虽然 short 忽略了它们，但是有三个参数被传递给中断处理程序：irq、dev_id 和 regs。让我们看看每个的作用。

中断号（int irq）作为你可能在日志消息中打印的信息是有用的。第二个参数，`void *dev_id`，是一种客户端数据；一个 `void *` 参数被传递给 request_irq，当中断发生时，这个相同的指针被作为一个参数传回给处理程序。你通常在 dev_id 中传递一个指向你的设备数据结构的指针，所以一个管理同一设备的多个实例的驱动程序在中断处理程序中不需要任何额外的代码来找出哪个设备负责当前的中断事件。

在中断处理程序中使用参数的典型用法如下：

```c
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 */

    /* .... */

}
```

与此处理程序相关的典型打开代码如下：

```c
static void sample_open(struct inode *inode, struct file
*filp)

{

    struct sample_dev *dev = hwinfo + MINOR(inode-
>i_rdev);

    request_irq(dev->irq, sample_interrupt,

                0 /* flags */, "sample", dev /* dev_id
*/);

    /*....*/

    return 0;

}
```

最后一个参数，`struct pt_regs *regs`，很少使用。它保存了处理器进入中断代码之前的处理器上下文的快照。这些寄存器可以用于监控和调试；它们通常不需要用于常规的设备驱动程序任务。

中断处理程序应返回一个值，表示是否实际上有一个中断需要处理。如果处理程序发现它的设备确实需要注意，它应返回 IRQ_HANDLED；否则返回值应为 IRQ_NONE。你也可以使用这个宏来生成返回值：

```c
IRQ_RETVAL(handled)
```

其中 handled 是非零的，如果你能够处理中断。返回值被内核用来检测和抑制伪中断。如果你的设备没有办法告诉你它是否真的中断，你应该返回 IRQ_HANDLED。

10.2.6. 使能和禁止中断

有时候，设备驱动程序必须阻止中断的传递一段（希望是短暂的）时间（我们在第5章的“自旋锁”部分看到了这样的情况）。通常，必须在持有自旋锁的同时阻止中断，以避免系统死锁。有一些不涉及自旋锁的禁用中断的方法。但在我们讨论它们之前，请注意，禁用中断应该是一个相对罕见的活动，即使在设备驱动程序中，这种技术也绝不应该被用作驱动程序内部的互斥机制。

10.2.6.1. 禁止单个中断

有时候（但很少！）驱动程序需要禁用特定中断线的中断传递。内核为此目的提供了三个函数，都在 <asm/irq.h> 中声明。这些函数是内核 API 的一部分，所以我们描述它们，但在大多数驱动程序中不鼓励使用它们。除其他外，你不能禁用共享中断线，而在现代系统中，共享中断是常态。话虽如此，这就是它们：

```c
void disable_irq(int irq);

void disable_irq_nosync(int irq);

void enable_irq(int irq);
```

调用这些函数中的任何一个可能会更新可编程中断控制器（PIC）中指定 irq 的掩码，从而禁用或启用所有处理器上的指定 IRQ。这些函数的调用可以嵌套——如果连续两次调用 disable_irq，那么在 IRQ 真正重新启用之前，需要两次调用 enable_irq。可以从中断处理程序中调用这些函数，但在处理自己的 IRQ 时启用它通常不是好的做法。

disable_irq 不仅禁用给定的中断，而且还等待当前正在执行的中断处理程序（如果有的话）完成。请注意，如果调用 disable_irq 的线程持有任何中断处理程序需要的资源（如自旋锁），系统可能会死锁。disable_irq_nosync 与 disable_irq 的不同之处在于它立即返回。因此，使用 disable_irq_nosync 虽然稍快一些，但可能让你的驱动程序面临竞争条件。

但为什么要禁用一个中断呢？坚持使用并行端口，让我们看看 plip 网络接口。一个 plip 设备使用基本的并行端口来传输数据。由于只有五个位可以从并行连接器中读取，所以它们被解释为四个数据位和一个时钟/握手信号。当数据包的前四位由发起者（发送数据包的接口）传输时，时钟线被拉高，导致接收接口中断处理器。然后调用 plip 处理程序来处理新到达的数据。

在设备被警告后，数据传输继续，使用握手线向接收接口时钟新数据（这可能不是最好的实现，但对于与使用并行端口的其他数据包驱动程序的兼容性是必要的）。如果接收接口必须处理每接收一个字节的两个中断，性能将无法忍受。因此，驱动程序在接收数据包期间禁用中断；相反，使用轮询和延迟循环来引入数据。同样，因为接收器到发射器的握手线用于确认数据接收，所以在数据包传输期间，发射接口禁用其 IRQ 线。

10.2.6.2. 禁止所有中断

如果你需要禁用所有中断怎么办？在 2.6 内核中，可以使用以下两个函数中的任何一个关闭当前处理器上的所有中断处理（这些函数在 <asm/system.h> 中定义）：

```c
void local_irq_save(unsigned long flags);

void local_irq_disable(void);
```

调用 local_irq_save 在保存当前中断状态到 flags 后禁用当前处理器上的中断传递。注意 flags 是直接传递的，而不是通过指针；这是因为 local_irq_save() 实际上是作为宏实现的，而不是函数。local_irq_disable 在不保存状态的情况下关闭本地中断传递；只有当你知道中断没有在其他地方被禁用时，才应该使用这个版本。

打开中断是通过以下方式完成的：

```c
void local_irq_restore(unsigned long flags);

void local_irq_enable(void);
```

第一个版本恢复了由 local_irq_save 存储到 flags 中的状态，而 local_irq_enable 无条件地启用中断。与 disable_irq 不同，local_irq_disable 不跟踪多次调用。如果调用链中的多个函数可能需要禁用中断，应使用 local_irq_save。

在 2.6 内核中，没有办法在整个系统中全局禁用所有中断。内核开发者已经决定，关闭所有中断的代价太高，而且无论如何都不需要这种能力。如果你正在使用一个旧的驱动程序，它调用了如 cli 和 sti 这样的函数，你需要更新它以使用适当的锁定，才能在 2.6 下工作。

10.3. 前和后半部Top and Bottom Halves

处理中断的主要问题之一是如何在处理程序中执行长时间的任务。通常，必须对设备中断做出大量的工作，但中断处理程序需要快速完成，不能长时间阻塞中断。这两个需求（工作和速度）相互冲突，让驱动程序编写者陷入困境。

Linux（以及许多其他系统）通过将中断处理程序分为两部分来解决这个问题。所谓的上半部分是实际响应中断的例程——你在 request_irq 中注册的那个。下半部分是由上半部分安排在稍后、更安全的时间执行的例程。上半部分处理程序和下半部分的主要区别在于，所有的中断在执行下半部分时都是启用的——这就是为什么它在更安全的时间运行。在典型的情况下，上半部分将设备数据保存到设备特定的缓冲区，安排其下半部分，然后退出：这个操作非常快。然后下半部分执行其他所需的工作，如唤醒进程，启动另一个 I/O 操作等。这种设置允许上半部分在下半部分仍在工作时服务新的中断。

几乎每一个严肃的中断处理程序都是这样分割的。例如，当网络接口报告新数据包到达时，处理程序只是检索数据并将其推送到协议层；实际的数据包处理是在下半部分执行的。

Linux 内核有两种不同的机制可以用来实现下半部分处理，这两种机制都在第7章中介绍过。Tasklet 通常是下半部分处理的首选机制；它们非常快，但所有的 tasklet 代码都必须是原子的。tasklet 的替代方案是工作队列，它们可能有更高的延迟，但是允许睡眠。

接下来的讨论再次使用 short 驱动程序。当使用模块选项加载时，可以告诉 short 以顶部/底部半模式进行中断处理，使用 tasklet 或 workqueue 处理程序。在这种情况下，上半部分执行得很快；它只是记住当前的时间并安排下半部分处理。然后下半部分负责编码这个时间并唤醒可能正在等待数据的用户进程。

10.3.1. Tasklet 实现

记住，tasklet 是一种特殊的函数，可以安排在软件中断上下文中运行，在系统确定的安全时间。它们可以被安排运行多次，但 tasklet 的调度不是累积的；tasklet 只运行一次，即使在启动之前反复请求，也只运行一次。没有任何 tasklet 会与自身并行运行，因为它们只运行一次，但在 SMP 系统上，tasklet 可以与其他 tasklet 并行运行。因此，如果你的驱动程序有多个 tasklet，它们必须使用某种锁定机制来避免相互冲突。

tasklet 也保证在首次调度它们的函数的同一 CPU 上运行。因此，中断处理程序可以确保在处理程序完成之前，tasklet 不会开始执行。然而，当 tasklet 运行时，另一个中断肯定可以被传递，所以可能仍然需要在 tasklet 和中断处理程序之间进行锁定。

tasklet 必须使用 DECLARE_TASKLET 宏声明：

```c
DECLARE_TASKLET(name, function, data);
```

name 是要给 tasklet 的名称，function 是被调用来执行 tasklet 的函数（它接受一个 unsigned long 参数并返回 void），data 是要传递给 tasklet 函数的 unsigned long 值。

short 驱动程序如下声明其 tasklet：

```c
void short_do_tasklet(unsigned long);

DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

函数 tasklet_schedule 用于安排一个 tasklet 运行。如果 short 加载了 tasklet=1，它会安装一个不同的中断处理程序，保存数据并安排 tasklet，如下所示：

```c
irqreturn_t short_tl_interrupt(int irq, void *dev_id,
struct pt_regs *regs)

{

    do_gettimeofday((struct timeval *) tv_head); /* cast
to stop 'volatile' warning

*/

    short_incr_tv(&tv_head);

    tasklet_schedule(&short_tasklet);

    short_wq_count++; /* record that an interrupt arrived
*/

    return IRQ_HANDLED;

}
```

实际的 tasklet 例程，short_do_tasklet，将在系统方便的时候执行（可以这么说）。如前所述，这个例程执行了处理中断的大部分工作；它看起来像这样：

```c
void short_do_tasklet (unsigned long unused)

{

    int savecount = short_wq_count, written;

    short_wq_count = 0; /* we have already been removed
from the queue */

    /*

     * The bottom half reads the tv array, filled by the
top half.

     * and prints it to the circular text buffer, which
is then consumed

     * by reading processes

     */

    /* First write the number of interrupts that occurred
before this bh */

    written = sprintf((char *)short_head,"bh after
%6i\n",savecount);

    short_incr_bp(&short_head, written);

    /*

     * Then, write the time values. Write exactly 16
bytes at a time,

     * so it aligns with PAGE_SIZE

     */
```

```
    do {

        written = sprintf((char
*)short_head,"%08u.%06u\n",

                (int)(tv_tail->tv_sec % 100000000),

                (int)(tv_tail->tv_usec));

        short_incr_bp(&short_head, written);

        short_incr_tv(&tv_tail);

    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /* awake any
reading process */

}
```

这个 tasklet 在其他事情中，记录了自上次调用以来有多少中断到达。像 short 这样的设备可以在短时间内产生很多中断，所以在下半部分执行之前有几个到达是很常见的。驱动程序必须始终准备好这种可能性，并且必须能够从上半部分留下的信息中确定有多少工作要做。

10.3.2. 工作队列

回想一下，工作队列在特殊的工作进程上下文中在未来的某个时间调用一个函数。由于工作队列函数在进程上下文中运行，所以如果需要的话，它可以睡眠。然而，你不能从工作队列中复制数据到用户空间，除非你使用我们在第15章中演示的高级技术；工作进程无法访问任何其他进程的地址空间。

如果 short 驱动程序加载时 wq 选项设置为非零值，它会使用工作队列进行其底半部分处理。它使用系统默认的工作队列，所以不需要特殊的设置代码；如果你的驱动程序有特殊的延迟要求（或者可能在工作队列函数中长时间睡眠），你可能想要创建你自己的专用工作队列。我们确实需要一个 work_struct 结构，它的声明和初始化如下：

```c
static struct work_struct short_wq;

/* this line is in short_init( ) */

INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet,
NULL);
```

我们的工作函数是 short_do_tasklet，我们已经在前一节中看到了。

当使用工作队列时，short 建立了另一个中断处理程序，如下所示：

```c
irqreturn_t short_wq_interrupt(int irq, void *dev_id,
struct pt_regs *regs)

{

    /* Grab the current time information. */

    do_gettimeofday((struct timeval *) tv_head);

    short_incr_tv(&tv_head);

    /* Queue the bh. Don't worry about multiple
enqueueing */

    schedule_work(&short_wq);

    short_wq_count++; /* record that an interrupt arrived
*/

    return IRQ_HANDLED;

}
```

如你所见，中断处理程序看起来非常像 tasklet 版本，只是它调用 schedule_work 来安排底半部分处理。

10.4. 中断共享

IRQ冲突的概念几乎与PC架构同义。在过去，PC上的IRQ线路不能服务于多于一个的设备，而且它们从来都不够用。结果是，沮丧的用户经常花费大量时间打开他们的电脑机箱，试图找到一种方法让所有的外设都能良好地一起工作。

当然，现代硬件已经被设计为允许共享中断；PCI总线就要求这样做。因此，Linux内核支持所有总线上的中断共享，即使是那些（如ISA总线）传统上不支持共享的总线。对于2.6内核的设备驱动程序，如果目标硬件可以支持那种操作模式，应该编写为与共享中断一起工作。幸运的是，大多数时候，使用共享中断是很容易的。

10.4.1. 安装一个共享的Handler

共享中断通过 request_irq 安装，就像非共享中断一样，但有两个区别：

- 在请求中断时，必须在 flags 参数中指定 SA_SHIRQ 位。
- dev_id 参数必须是唯一的。任何指向模块地址空间的指针都可以，但 dev_id 绝对不能设置为 NULL。

内核保持与中断关联的共享处理程序的列表，dev_id 可以被认为是区分它们的签名。如果两个驱动程序在同一个中断上注册 NULL 作为他们的签名，那么在卸载时可能会混淆，导致中断到达时内核崩溃。因此，当注册共享中断时，如果传递了 NULL dev_id，现代内核会大声抱怨。当请求一个共享中断时，如果满足以下条件之一，request_irq 就会成功：

- 中断线是空闲的。
- 已经为该线注册的所有处理程序也指定了 IRQ 是共享的。

每当两个或更多的驱动程序共享一个中断线，并且硬件在该线上中断处理器时，内核会调用为该中断注册的每个处理程序，将其自己的 dev_id 传递给每个处理程序。因此，一个共享处理程序必须能够识别自己的中断，并且当自己的设备没有中断时应该快速退出。当你的处理程序被调用并发现设备没有中断时，一定要返回 IRQ_NONE。

如果你需要在请求 IRQ 线之前探测你的设备，内核无法帮助你。没有可用的探测函数用于共享处理程序。如果正在使用的线是空闲的，标准的探测机制就会工作，但如果线已经被另一个具有共享能力的驱动程序占用，即使你的驱动程序可以完美工作，探测也会

失败。幸运的是，大多数为中断共享设计的硬件也能告诉处理器它正在使用哪个中断，从而消除了显式探测的需要。

使用 free_irq 以正常方式释放处理程序。这里的 dev_id 参数用于从中断的共享处理程序列表中选择正确的处理程序进行释放。这就是为什么 dev_id 指针必须是唯一的。

使用共享处理程序的驱动程序需要注意一件事：它不能玩弄 enable_irq 或 disable_irq。如果它这样做，对于共享线的其他设备可能会出现问题；即使短时间禁用另一个设备的中断也可能为该设备及其用户创建问题性的延迟。通常，程序员必须记住他的驱动程序并不拥有 IRQ，其行为应该比拥有中断线的情况更“社交”。

10.4.2. 运行Handler

如前所述，当内核接收到一个中断时，所有注册的处理程序都会被调用。一个共享处理程序必须能够区分它需要处理的中断和其他设备生成的中断。

使用选项 shared=1 加载 short 会安装以下处理程序，而不是默认的处理程序：

```c
irqreturn_t short_sh_interrupt(int irq, void *dev_id,
struct pt_regs *regs)

{

    int value, written;

    struct timeval tv;

    /* 如果不是 short，立即返回 */

    value = inb(short_base);

    if (!(value & 0x80))

        return IRQ_NONE;

    /* 清除中断位 */

    outb(value & 0x7F, short_base);

    /* 其余部分不变 */

    do_gettimeofday(&tv);

    written = sprintf((char *)short_head,"%08u.%06u\n",

            (int)(tv.tv_sec % 100000000), (int)
(tv.tv_usec));

    short_incr_bp(&short_head, written);

    wake_up_interruptible(&short_queue); /* 唤醒任何正在读
取的进程 */

    return IRQ_HANDLED;

}
```

这里需要解释一下。由于并行端口没有“中断待处理”位可以检查，所以处理程序使用 ACK 位来达到这个目的。如果该位为高，那么正在报告的中断就是为 short 的，处理程序清除该位。

处理程序通过将并行接口的数据端口的高位清零来重置该位——short 假设引脚 9 和 10 是连在一起的。如果与 short 共享 IRQ 的其他设备生成一个中断，short 看到自己的线仍然是非活动的，就什么也不做。

当然，一个全功能的驱动程序可能将工作分为上半部分和下半部分，但这很容易添加，并且不会对实现共享的代码产生任何影响。一个真正的驱动程序也可能使用 dev_id 参数来确定可能有许多设备中的哪一个可能正在中断。

请注意，如果你正在使用打印机（而不是跳线）来测试 short 的中断管理，这个共享处理程序可能不会按照广告中的方式工作，因为打印机协议不允许共享，驱动程序无法知道中断是来自打印机的。

10.4.3. /proc 接口和共享中断

在系统中安装共享处理程序不会影响 /proc/stat，它甚至不知道处理程序。然而，/proc/interrupts 会有些许变化。

所有为同一个中断号安装的处理程序都会出现在 /proc/interrupts 的同一行。以下输出（来自一个 x86_64 系统）显示了如何显示共享中断处理程序：

```
           CPU0
  0:  892335412          XT-PIC  timer
  1:     453971          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:          0          XT-PIC  libata, ehci_hcd
  8:          0          XT-PIC  rtc
  9:          0          XT-PIC  acpi
 10:   11365067          XT-
PIC  ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx, EMU10K1
 11:    4391962          XT-PIC  uhci_hcd, uhci_hcd
 12:        224          XT-PIC  i8042
 14:    2787721          XT-PIC  ide0
 15:     203048          XT-PIC  ide1
NMI:      41234
LOC:  892193503
ERR:        102
MIS:          0
```

这个系统有几个共享的中断线。IRQ 5 用于串行ATA和USB 2.0控制器；IRQ 10 有几个设备，包括一个IDE控制器，两个USB控制器，一个以太网接口和一个声卡；IRQ 11 也被两个USB控制器使用。

10.5. 中断驱动 I/O

无论何时，只要由于任何原因可能会延迟到管理硬件的数据传输，驱动程序编写者都应该实现缓冲。数据缓冲区有助于将数据传输和接收与写和读系统调用分离，整体系统性能得到提升。

一个好的缓冲机制会导致中断驱动的I/O，其中输入缓冲区在中断时间被填充，并由读取设备的进程清空；输出缓冲区由写入设备的进程填充，并在中断时间被清空。一个中断驱动输出的例子是 /dev/shortprint 的实现。

为了成功进行中断驱动的数据传输，硬件应该能够生成具有以下语义的中断：

- 对于输入，设备在新数据到达并准备好由系统处理器检索时中断处理器。实际要执行的操作取决于设备是否使用I/O端口，内存映射，或DMA。
- 对于输出，设备在准备好接受新数据或确认成功的数据传输时发送一个中断。内存映射和DMA能力的设备通常生成中断，告诉系统它们已经完成了缓冲区的处理。

在第6章的“阻塞和非阻塞操作”一节中介绍了读或写与数据实际到达之间的时间关系。

10.5.1. 一个写缓存例子

我们已经提到了几次 shortprint 驱动程序；现在是时候真正看一看了。这个模块实现了一个非常简单的，面向输出的并行端口驱动程序；然而，它足以使打印文件成为可能。如果你选择测试这个驱动程序，然而，记住你必须传递给打印机一个它能理解的格式的文件；并非所有的打印机在给定一串任意数据时都能很好地响应。

shortprint 驱动程序维护了一个一页的循环输出缓冲区。当用户空间进程向设备写入数据时，该数据被送入缓冲区，但是 write 方法实际上并不执行任何 I/O。相反，shortp_write 的核心看起来像这样：

```c
while (written < count) {
    /* Hang out until some buffer space is available. */
    space = shortp_out_space( );
    if (space <= 0) {
        if (wait_event_interruptible(shortp_out_queue,
                    (space = shortp_out_space( )) > 0))
            goto out;
    }
    /* Move data into the buffer. */
    if ((space + written) > count)
        space = count - written;
    if (copy_from_user((char *) shortp_out_head, buf,
space)) {
        up(&shortp_out_sem);
        return -EFAULT;
    }
    shortp_incr_out_bp(&shortp_out_head, space);
    buf += space;
    written += space;
    /* If no output is active, make it active. */
```

```
        spin_lock_irqsave(&shortp_out_lock, flags);
        if (! shortp_output_active)
            shortp_start_output( );
        spin_unlock_irqrestore(&shortp_out_lock, flags);
    }
    out:
    *f_pos += written;
```

一个信号量（shortp_out_sem）控制对循环缓冲区的访问；shortp_write 在上述代码片段之前获取该信号量。在持有信号量的同时，它试图将数据送入循环缓冲区。函数 shortp_out_space 返回可用的连续空间的数量（所以不需要担心缓冲区包裹）；如果该数量为0，驱动程序等待直到有一些空间被释放。然后它将尽可能多的数据复制到缓冲区。

一旦有数据要输出，shortp_write 必须确保数据被写入设备。实际的写入是通过工作队列函数完成的；shortp_write 必须启动该函数，如果它还没有运行的话。在获取一个控制对输出缓冲区消费者端使用的变量的访问的单独的自旋锁（包括 shortp_output_active）后，如果需要，它调用 shortp_start_output。然后就只是记录有多少数据被“写入”到缓冲区并返回。

启动输出过程的函数看起来像这样：

```c
static void shortp_start_output(void)
{
    if (shortp_output_active) /* Should never happen */
        return;
    /* Set up our 'missed interrupt' timer */
    shortp_output_active = 1;
    shortp_timer.expires = jiffies + TIMEOUT;
    add_timer(&shortp_timer);
    /*  And get the process going. */
    queue_work(shortp_workqueue, &shortp_work);
}
```

处理硬件的现实是，你偶尔会丢失设备的一个中断。当这种情况发生时，你真的不希望你的驱动程序停止，直到系统重启；这不是一种用户友好的做事方式。更好的做法是意识到一个中断已经被错过，收拾残局，然后继续。为此，每当 shortprint 向设备输出数据时，它都会设置一个内核定时器。如果定时器到期，我们可能已经错过了一个中断。我们稍后会看到定时器函数，但是，目前，让我们坚持主要的输出功能。这是在我们的工作队列函数中实现的，如你在上面可以看到，它在这里被调度。该函数的核心看起来像这样：

```c
spin_lock_irqsave(&shortp_out_lock, flags);

/* Have we written everything? */

if (shortp_out_head = = shortp_out_tail) { /* empty */

    shortp_output_active = 0;

    wake_up_interruptible(&shortp_empty_queue);

    del_timer(&shortp_timer);

}

/* Nope, write another byte */

else

    shortp_do_write( );

/* If somebody's waiting, maybe wake them up. */

if (((PAGE_SIZE + shortp_out_tail - shortp_out_head) %
PAGE_SIZE) > SP_MIN_SPACE)

{

wake_up_interruptible(&shortp_out_queue);

}

spin_unlock_irqrestore(&shortp_out_lock, flags);
```

由于我们正在处理输出端的共享变量，我们必须获取自旋锁。然后我们查看是否还有更多的数据要发送出去；如果没有，我们注意到输出不再活动，删除定时器，并唤醒任何

可能一直在等待队列完全空出的人（这种等待是在设备关闭时进行的）。如果还有数据要写入，我们调用 shortp_do_write 实际向硬件发送一个字节。

然后，由于我们可能已经在输出缓冲区中释放了空间，我们考虑唤醒任何等待向该缓冲区添加更多数据的进程。然而，我们并不无条件地执行这种唤醒；相反，我们等待直到有一定数量的空间可用。每次我们从缓冲区中取出一个字节时，唤醒一个写入者是没有意义的；唤醒进程，调度它运行，然后让它重新睡眠的成本太高了。相反，我们应该等到该进程能够一次性将大量数据移动到缓冲区中。这种技术在缓冲，中断驱动的驱动程序中很常见。

为了完整性，这里是实际将数据写入端口的代码：

```c
static void shortp_do_write(void)
{
    unsigned char cr = inb(shortp_base + SP_CONTROL);
    /* Something happened; reset the timer */
    mod_timer(&shortp_timer, jiffies + TIMEOUT);
    /* Strobe a byte out to the device */
    outb_p(*shortp_out_tail, shortp_base+SP_DATA);
    shortp_incr_out_bp(&shortp_out_tail, 1);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

在这里，我们重置定时器以反映我们已经取得了一些进展，向设备输出一个字节，并更新循环缓冲区指针。

工作队列函数不直接重新提交自己，所以只有一个字节会被写入设备。在某个时候，打印机会以它的慢速方式消耗这个字节并准备好下一个字节；然后它会中断处理器。在 shortprint 中使用的中断处理程序简短而简单：

```c
static irqreturn_t shortp_interrupt(int irq, void
*dev_id, struct pt_regs *regs)

{

    if (! shortp_output_active)

        return IRQ_NONE;

        /* Remember the time, and farm off the rest to
the workqueue function */

    do_gettimeofday(&shortp_tv);

    queue_work(shortp_workqueue, &shortp_work);

    return IRQ_HANDLED;

}
```

由于并行端口不需要显式的中断确认，所以中断处理程序真正需要做的就是告诉内核再次运行工作队列函数。

如果中断永远不来怎么办？我们到目前为止看到的驱动程序代码将简单地停止。为了防止这种情况发生，我们在几页前设置了一个定时器。当该定时器到期时执行的函数是：

```c
static void shortp_timeout(unsigned long unused)

{

    unsigned long flags;

    unsigned char status;

    if (! shortp_output_active)

        return;

    spin_lock_irqsave(&shortp_out_lock, flags);

    status = inb(shortp_base + SP_STATUS);

    /* If the printer is still busy we just reset the
timer */

    if ((status & SP_SR_BUSY) == 0 || (status &
SP_SR_ACK)) {

        shortp_timer.expires = jiffies + TIMEOUT;

        add_timer(&shortp_timer);

        spin_unlock_irqrestore(&shortp_out_lock, flags);

        return;

    }

    /* Otherwise we must have dropped an interrupt. */

    spin_unlock_irqrestore(&shortp_out_lock, flags);

    shortp_interrupt(shortp_irq, NULL, NULL);
```

```
}
```

如果不应该有输出活动，定时器函数只是返回；这阻止了在关闭时定时器重新提交自己。然后，在获取锁之后，我们查询端口的状态；如果它声称自己忙，那么它只是还没有打断我们，所以我们重置定时器并返回。打印机有时可能需要很长时间才能准备好；考虑一下在每个人都离开的长周末期间用完纸的打印机。在这种情况下，除了耐心等待直到某些事情发生之外，别无他法。

然而，如果打印机声称自己已经准备好了，我们一定是错过了它的中断。在这种情况下，我们只是手动调用我们的中断处理程序，让输出过程再次运行。

shortprint 驱动程序不支持从端口读取；相反，它的行为像 shortint 一样，返回中断定时信息。然而，我们已经看到的中断驱动读取方法的实现将非常相似。设备的数据将被读入驱动程序缓冲区；只有当缓冲区中积累了大量数据，满足了完整的读取请求，或者发生了某种超时，才会将其复制到用户空间。

10.6. 快速参考

本章中介绍了这些关于中断管理的符号:

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

调用这个注册和注销一个中断处理.

```
#include <linux/irq.h.h>
int can_request_irq(unsigned int irq, unsigned long
flags);
```

这个函数, 在 i386 和 x86_64 体系上有, 返回一个非零值如果一个分配给定中断线的企图成功.

```
#include <asm/signal.h>
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM
```

给 request_irq 的标志. SA_INTERRUPT 请求安装一个快速处理者(相反是一个慢速的). SA_SHIRQ 安装一个共享的处理者, 并且第 3 个 flag 声称中断时戳可用来产生系统熵.

```
/proc/interrupts
/proc/stat
```

报告硬件中断和安装的处理者的文件系统节点.

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

驱动使用的函数, 当它不得不探测来决定哪个中断线被设备在使用. probe_irq_on 的结果必须传回给 probe_irq_off 在中断产生之后. probe_irq_off 的返回值是被探测的中断号.

```
IRQ_NONE
IRQ_HANDLED
IRQ_RETVAL(int x)
```

从一个中断处理返回的可能值, 指示是否一个来自设备的真正的中断出现了.

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

驱动可以使能和禁止中断报告. 如果硬件试图在中断禁止时产生一个中断, 这个中断永远丢失了. 一个使用一个共享处理者的驱动必须不使用这个函数.

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

使用 local_irq_save 来禁止本地处理器的中断并且记住它们之前的状态. flags 可以被传递给 local_irq_restore 来恢复之前的中断状态.

```
void local_irq_disable(void);
void local_irq_enable(void);
```

在当前处理器熵无条件禁止和使能中断的函数.