第⑦章

中断和中断处理

任何操作系统内核的核心任务，都包含有对连接到计算机上的硬件设备进行有效管理，如硬盘、蓝光碟机、键盘、鼠标、3D 处理器，以及无线电等。而想要管理这些设备，首先要能和它们互通音信才行。众所周知，处理器的速度跟外围硬件设备的速度往往不在一个数量级上，因此，如果内核采取让处理器向硬件发出一个请求，然后专门等待回应的办法，显然差强人意。既然硬件的响应这么慢，那么内核就应该在此期间处理其他事务，等到硬件真正完成了请求的操作之后，再回过头来对它进行处理。

那么到底如何让处理器和这些外部设备能协同工作，且不会降低机器的整体性能呢？轮询（polling）可能会是一种解决办法。它可以让内核定期对设备的状态进行查询，然后做出相应的处理。不过这种方法很可能会让内核做不少无用功，因为无论硬件设备是正在忙碌着完成任务还是已经大功告成，轮询总会周期性地重复执行。更好的办法是由我们来提供一种机制，让硬件在需要的时候再向内核发出信号[⊖]。这就是中断机制。在本章中，我们将先讨论中断，进而讨论内核如何使用所谓的中断处理函数处理对应的中断。

7.1 中断

中断使得硬件得以发出通知给处理器。例如，在你敲击键盘的时候，键盘控制器（控制键盘的硬件设备）会发送一个中断，通知操作系统有键按下。中断本质上是一种特殊的电信号，由硬件设备发向处理器。处理器接收到中断后，会马上向操作系统反映此信号的到来，然后就由操作系统负责处理这些新到来的数据。硬件设备生成中断的时候并不考虑与处理器的时钟同步——换句话说就是中断随时可以产生。因此，内核随时可能因为新到来的中断而被打断。

从物理学的角度看，中断是一种电信号，由硬件设备生成，并直接送入中断控制器的输入引脚中——中断控制器是个简单的电子芯片，其作用是将多路中断管线，采用复用技术只通过一个和处理器相连接的管线与处理器通信。当接收到一个中断后，中断控制器会给处理器发送一个电信号。处理器一经检测到此信号，便中断自己的当前工作转而处理中断。此后，处理器会通知操作系统已经产生中断，这样，操作系统就可以对这个中断进行适当地处理了。

不同的设备对应的中断不同，而每个中断都通过一个唯一的数字标志。因此，来自键盘的中断就有别于来自硬盘的中断，从而使得操作系统能够对中断进行区分，并知道哪个硬件设备产生了哪个中断。这样，操作系统才能给不同的中断提供对应的中断处理程序。

⊖ 变内核主动为硬件主动。——译者注

这些中断值通常被称为中断请求（IRQ）线。每个 IRQ 线都会被关联一个数值量——例如，在经典的 PC 机上，IRQ 0 是时钟中断，而 IRQ 1 是键盘中断。但并非所有的中断号都是这样严格定义的。例如，对于连接在 PCI 总线上的设备而言，中断是动态分配的。而且其他非 PC 的体系结构也具有动态分配可用中断的特性。重点在于特定的中断总是与特定的设备相关联，并且内核要知道这些信息。实际上，硬件发出中断是为了引起内核的关注：嗨，我有新的按键等待处理呢，读取并处理这些调皮鬼吧！

异常

在操作系统中，讨论中断就不能不提及异常。异常与中断不同，它在产生时必须考虑与处理器时钟同步。实际上，异常也常常称为同步中断。在处理器执行到由于编程失误而导致的错误指令（如被 0 除）的时候，或者是在执行期间出现特殊情况（如缺页），必须靠内核来处理的时候，处理器就会产生一个异常。因为许多处理器体系结构处理异常与处理中断的方式类似，因此，内核对它们的处理也很类似。本章对中断（由硬件产生的异步中断）的讨论，大部分也适合于异常（由处理器本身产生的同步中断）。

你已经熟悉一种异常：在第 6 章中你已看到，在 x86 体系结构上如何通过软中断实现系统调用，那就是陷入内核，然后引起一种特殊的异常——系统调用处理程序异常。你会看到，中断的工作方式与之类似，其差异只在于中断是由硬件而不是软件引起的。

7.2 中断处理程序

在响应一个特定中断的时候，内核会执行一个函数，该函数叫做中断处理程序（interrupt handler）或中断服务例程（interrupt service routine，ISR）。产生中断的每个设备都有一个[⊖]相应的中断处理程序。例如，由一个函数专门处理来自系统时钟的中断，而另外一个函数专门处理由键盘产生的中断。一个设备的中断处理程序是它设备驱动程序（driver）的一部分——设备驱动程序是用于对设备进行管理的内核代码。

在 Linux 中，中断处理程序就是普普通通的 C 函数。只不过这些函数必须按照特定的类型声明，以便内核能够以标准的方式传递处理程序的信息，在其他方面，它们与一般的函数别无二致。中断处理程序与其他内核函数的真正区别在于，中断处理程序是被内核调用来响应中断的，而它们运行于我们称之为中断上下文的特殊上下文中（关于中断上下文，我们将在后面讨论）。需要指出的是，中断上下文偶尔也称作原子上下文，因为正如我们看到的，该上下文中的执行代码不可阻塞。不过在本书中我们使用中断上下文这个称谓。

中断可能随时发生，因此中断处理程序也就随时可能执行。所以必须保证中断处理程序能够快速执行，这样才能保证尽可能快地恢复中断代码的执行。因此，尽管对硬件而言，操作系统能迅速对其中断进行服务非常重要；当然对系统的其他部分而言，让中断处理程序在尽可能短的时间内完成运行也同样重要。

⊖ 中断处理程序通常不是和特定设备关联，而是和特定中断关联的，也就是说，如果一个设备可以产生多种不同的中断，那么该设备就可以对应多个中断处理程序，相应的，该设备的驱动程序也就需要准备多个这样的函数。

最起码的，中断处理程序要负责通知硬件设备中断已被接收：嗨，硬件，我听到你了，现在回去工作吧！但是中断处理程序往往还要完成大量其他的工具。例如，我们可以考虑一下网络设备的中断处理程序面临的挑战。该处理程序除了要对硬件应答，还要把来自硬件的网络数据包拷贝到内存，对其进行处理后再交给合适的协议栈或应用程序。显而易见，这种工作量不会太小，尤其对于如今的千兆比特和万兆比特以太网卡而言。

7.3 上半部与下半部的对比

又想中断处理程序运行得快，又想中断处理程序完成的工作量多，这两个目的显然有所抵触。鉴于两个目的之间存在此消彼长的矛盾关系，所以我们一般把中断处理切为两个部分或两半。中断处理程序是上半部（top half）——接收到一个中断，它就立即开始执行，但只做有严格时限的工作，例如对接收的中断进行应答或复位硬件，这些工作都是在所有中断被禁止的情况下完成的。能够被允许稍后完成的工作会推迟到下半部（bottom half）去。此后，在合适的时机，下半部会被开中断执行。Linux 提供了实现下半部的各种机制，第 8 章会讨论这些机制。

让我们考察一下上半部和下半部分割的例子，还是以我们的老朋友——网卡作为实例。当网卡接收来自网络的数据包时，需要通知内核数据包到了。网卡需要立即完成这件事，从而优化网络的吞吐量和传输周期，以避免超时。因此，网卡立即发出中断：嗨，内核，我这里有最新数据包了。内核通过执行网卡已注册的中断处理程序来做出应答。

中断开始执行，通知硬件，拷贝最新的网络数据包到内存，然后读取网卡更多的数据包。这些都是重要、紧迫而又与硬件相关的工作。内核通常需要快速的拷贝网络数据包到系统内存，因为网卡上接收网络数据包的缓存大小固定，而且相比系统内存也要小得多。所以上述拷贝动作一旦被延迟，必然造成缓存溢出——进入的网络包占满了网卡的缓存，后续的入包只能被丢弃。当网络数据包被拷贝到系统内存后，中断的任务算是完成了，这时它将控制权交还给系统被中断前原先运行的程序。处理和操作数据包的其他工作在随后的下半部中进行。本章，我们考察上半部；第 8 章，我们关注下半部。

7.4 注册中断处理程序

中断处理程序是管理硬件的驱动程序的组成部分。每一设备都有相关的驱动程序，如果设备使用中断（大部分设备如此），那么相应的驱动程序就注册一个中断处理程序。

驱动程序可以通过 request_irq() 函数注册一个中断处理程序（它被声明在文件 <linux/interrupt.h> 中），并且激活给定的中断线，以处理中断：

```
/* request_irq: 分配一条给定的中断线 */
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

第一个参数 irq 表示要分配的中断号。对某些设备，如传统 PC 设备上的系统时钟或键盘，这个值通常是预先确定的。而对于大多数其他设备来说，这个值要么是可以通过探测获取，要么可以通过编程动态确定。

第二个参数 handler 是一个指针，指向处理这个中断的实际中断处理程序。只要操作系统一接收到中断，该函数就被调用。

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

注意 handler 函数的原型，它接受两个参数，并有一个类型为 irqreturn_t 的返回值。我们将在本章随后的部分讨论这个函数。

7.4.1 中断处理程序标志

第三个参数 flags 可以为 0，也可能是下列一个或多个标志的位掩码。其定义在文件 <linux/interrupt.h>。在这些标志中最重要的是：

- IRQF_DISABLED——该标志被设置后，意味着内核在处理中断处理程序本身期间，要禁止所有的其他中断。如果不设置，中断处理程序可以与除本身外的其他任何中断同时运行。多数中断处理程序是不会去设置该位的，因为禁止所有中断是一种野蛮行为。这种用法留给希望快速执行的轻量级中断。这一标志是 SA_INTERRUPT 标志的当前表现形式，在过去的中断中用以区分“快速”和“慢速”中断。
- IRQF_SAMPLE_RANDOM——此标志表明这个设备产生的中断对内核熵池（entropy pool）有贡献。内核熵池负责提供从各种随机事件导出的真正的随机数。如果指定了该标志，那么来自该设备的中断间隔时间就会作为熵填充到熵池。如果你的设备以预知的速率产生中断（如系统定时器），或者可能受外部攻击者（如联网设备）的影响，那么就不要设置这个标志。相反，有其他很多硬件产生中断的速率是不可预知的，所以都能成为一种较好的熵源。
- IRQF_TIMER —— 该标志是特别为系统定时器的中断处理而准备的。
- IRQF_SHARED——此标志表明可以在多个中断处理程序之间共享中断线。在同一个给定线上注册的每个处理程序必须指定这个标志；否则，在每条线上只能有一个处理程序。有关共享中断处理程序的更多信息将在下面的内容中提供。

第四个参数 name 是与中断相关的设备的 ASCII 文本表示。例如，PC 机上键盘中断对应的这个值为“keyboard”。这些名字会被 /proc/irq 和 /proc/interrupts 文件使用，以便与用户通信，稍后我们将对此进行简短讨论。

第五个参数 dev 用于共享中断线。当一个中断处理程序需要释放时（稍后讨论），dev 将提供唯一的标志信息（cookie），以便从共享中断线的诸多中断处理程序中删除指定的那一个。如果没有这个参数，那么内核不可能知道在给定的中断线上到底要删除哪一个处理程序。如果无须共享中断线，那么将该参数赋为空值（NULL）就可以了，但是，如果中断线是被共享的，那么就必须传递唯一的信息（除非设备又旧又破且位于 ISA 总线上，那么就必须支持共享中断）。另

外，内核每次调用中断处理程序时，都会把这个指针传递给它[⊖]。实践中往往会通过它传递驱动程序的设备结构：这个指针是唯一的，而且有可能在中断处理程序内被用到。

request_irq() 成功执行会返回 0。如果返回非 0 值，就表示有错误发生，在这种情况下，指定的中断处理程序不会被注册。最常见的错误是 -EBUSY，它表示给定的中断线已经在使用（或者当前用户或者你没有指定 IRQF_SHARED）。

注意，request_irq() 函数可能会睡眠，因此，不能在中断上下文或其他不允许阻塞的代码中调用该函数。天真地在睡眠不安全的上下文中调用 request_irq() 函数，是一种常见错误。造成这种错误的部分原因是为什么 request_irq() 会引起堵塞——这确实让人费解。在注册的过程中，内核需要在 /proc/irq 文件中创建一个与中断对应的项。函数 proc_ mkdir() 就是用来创建这个新的 procfs 项的。proc_mkdir() 通过调用函数 proc_create() 对这个新的 profs 项进行设置，而 proc_create() 会调用函数 kmalloc() 来请求分配内存。我们在第 12 章中将会看到，函数 kmalloc() 是可以睡眠的。看清楚了，你的程序就是跑到那里小憩去了！

7.4.2 一个中断例子

在一个驱动程序中请求一个中断线，并在通过 request_irq() 安装中断处理程序：

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

在这个例子中，irqn 是请求的中断线；my_interrupt 是中断处理程序；我们通过标志设置中断线可以共享；设备命名为“my_device”；最后是传递 my_dev 变量给 dev 形参。如果请求失败，那么这段代码将打印出一个错误并返回。如果调用返回 0，则说明处理程序已经成功安装。此后，处理程序就会在响应该中断时被调用。有一点很重要，初始化硬件和注册中断处理程序的顺序必须正确，以防止中断处理程序在设备初始化完成之前就开始执行。

7.4.3 释放中断处理程序

卸载驱动程序时，需要注销相应的中断处理程序，并释放中断线。上述动作需要调用：

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

如果指定的中断线不是共享的，那么，该函数删除处理程序的同时将禁用这条中断线。如果中断线是共享的，则仅删除 dev 所对应的处理程序，而这条中断线本身只有在删除了最后一个处理程序时才会被禁用。由此可以看出为什么唯一的 dev 如此重要。对于共享的中断线，需要一个唯一的信息来区分其上面的多个处理程序，并让 free_irq() 仅仅删除指定的处理程序。不管在哪

⊖ 中断处理程序都是预先在内核进行注册的回调函数（callback function），而不同的函数位于不同的驱动程序中，所以在这些函数共享同一个中断线时，内核必须准确地为它们创造执行环境，此时就可以通过这个指针将有用的环境信息传递给它们了。——译者注

种情况下（共享或不共享），如果 dev 非空，它都必须与需要删除的处理程序相匹配。必须从进程上下文中调用 free_irq()。

表 7-1 给出了终端处理函数的注册和注销函数。

表 7-1 中断注册方法表

函 数	描 述
request_irq()	在给定的中断线上注册一给定的中断处理程序
free_irq()	如果在给定的中断线上没有中断处理程序，则注销响应的处理程序，并禁用其中断线

7.5 编写中断处理程序

以下是一个中断处理程序声明：

```
static irqreturn_t intr_handler(int irq, void *dev)
```

注意，它的类型与 request_irq() 参数中 handler 所要求的参数类型相匹配。第一个参数 irq 就是这个处理程序要响应的中断的中断号。如今，这个参数已经没有太大用处了，可能只是在打印日志信息时会用到。而在 2.0 版以前的 Linux 内核中，由于没有 dev 这个参数，必须通过 irq 才能区分使用相同驱动程序，因而也使用相同的中断处理程序的多个设备。例如，具有多个相同类型硬盘驱动控制器的计算机。

第二个参数 dev 是一个通用指针，它与在中断处理程序注册时传递给 request_irq() 的参数 dev 必须一致。如果该值有唯一确定性（这样做是为了能支持共享），那么它就相当于一个 cookie，可以用来区分共享同一中断处理程序的多个设备。另外 dev 也可能指向中断处理程序使用的一个数据结构。因为对每个设备而言，设备结构都是唯一的，而且可能在中断处理程序中也用得到，因此，它也通常被看做 dev。

中断处理程序的返回值是一个特殊类型：irqreturn_t。中断处理程序可能返回两个特殊的值：IRQ_NONE 和 IRQ_HANDLED。当中断处理程序检测到一个中断，但该中断对应的设备并不是在注册处理函数期间指定的产生源时，返回 IRQ_NONE；当中断处理程序被正确调用，且确实是它所对应的设备产生了中断时，返回 IRQ_HANDLED。另外，也可以使用宏 IRQ_RETVAL(val)。如果 val 为非 0 值，那么该宏返回 IRQ_HANDLED；否则，返回 IRQ_NONE。利用这些特殊的值，内核可以知道设备发出的是否是一种虚假的（未请求）中断。如果给定中断线上所有中断处理程序返回的都是 IRQ_NONE，那么，内核就可以检测到出了问题。注意，irqreturn_t 这个返回类型实际上就是一个 int 型。之所以使用这些特殊值是为了与早期的内核保持兼容——2.6 版之前的内核并不支持这种特性，中断处理程序只需返回 void 就行了。如果要在 2.4 或更早的内核上使用这样的驱动程序，只需简单地将 typedef irqreturn_t 改为 void，屏蔽掉此特性，并给 no-ops 定义不同的返回值，其他用不着做什么大的修改。中断处理程序通常会标记为 static，因为它从来不会被别的文件中的代码直接调用。

中断处理程序扮演什么样的角色要取决于产生中断的设备和该设备为什么要发送中断。即使其他什么工作也不做，绝大部分的中断处理程序至少需要知道产生中断的设备，告诉它已经收到中断了。对于复杂一些的设备，可能还需要在中断处理程序中发送和接收数据，以及执行一

些扩充的工作。如前所述，应尽可能将扩充的工作推给下半部处理程序，这点将在第 8 章中进行讨论。

重入和中断处理程序

Linux 中的中断处理程序是无须重入的。当一个给定的中断处理程序正在执行时，相应的中断线在所有处理器上都会被屏蔽掉，以防止在同一中断线上接收另一个新的中断。通常情况下，所有其他的中断都是打开的，所以这些不同中断线上的其他中断都能被处理，但当前中断线总是被禁止的。由此可以看出，同一个中断处理程序绝对不会被同时调用以处理嵌套的中断。这极大地简化了中断处理程序的编写。

7.5.1 共享的中断处理程序

共享的处理程序与非共享的处理程序在注册和运行方式上比较相似，但差异主要有以下三处：

- request_irq() 的参数 flags 必须设置 IRQF_SHARED 标志。
- 对于每个注册的中断处理程序来说，dev 参数必须唯一。指向任一设备结构的指针就可以满足这一要求；通常会选择设备结构，因为它是唯一的，而且中断处理程序可能会用到它。不能给共享的处理程序传递 NULL 值。
- 中断处理程序必须能够区分它的设备是否真的产生了中断。这既需要硬件的支持，也需要处理程序中有相关的处理逻辑。如果硬件不支持这一功能，那中断处理程序肯定会束手无策，它根本没法知道到底是与它对应的设备发出了这个中断，还是共享这条中断线的其他设备发出了这个中断。

所有共享中断线的驱动程序都必须满足以上要求。只要有任何一个设备没有按规则进行共享，那么中断线就无法共享了。指定 IRQF_SHARED 标志以调用 request_irq() 时，只有在以下两种情况下才可能成功：中断线当前未被注册，或者在该线上的所有已注册处理程序都指定了 IRQF_SHARED。注意，在这一点上 2.6 版与以前的内核是不同的，共享的处理程序可以混用 IRQF_DISABLED。

内核接收一个中断后，它将依次调用在该中断线上注册的每一个处理程序。因此，一个处理程序必须知道它是否应该为这个中断负责。如果与它相关的设备并没有产生中断，那么处理程序应该立即退出。这需要硬件设备提供状态寄存器（或类似机制），以便中断处理程序进行检查。毫无疑问，大多数硬件都提供这种功能。

7.5.2 中断处理程序实例

让我们考察一个实际的中断处理程序，它来自 real-time clock（RTC）驱动程序，可以在 drivers/char/rtc.c 中找到。很多机器（包括 PC）都可以找到 RTC。它是一个从系统定时器中独立出来的设备，用于设置系统时钟，提供报警器（alarm）或周期性的定时器。对大多数体系结构而言，系统时钟的设置，通常只需要向某个特定的寄存器或 I/O 地址写入想要的时间就可以了。然而报警器或周期性定时器通常就得靠中断来实现。这种中断与生活中的闹铃差不多：中断发出

时，报警器或定时器就会启动。

RTC 驱动程序装载时，rtc_init() 函数会被调用，对这个驱动程序进行初始化。它的职责之一就是注册中断处理程序：

```
/* 对 rtc_irq 注册 rtc_interrupt */
if (request_irq(rtc_irq, rtc_interrupt, IRQF_SHARED, "rtc", (void *)&rtc_port)) {
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
        return -EIO;
}
```

从中我们看到，中断号由 rtc_irq 指定。这个变量用于为给定体系结构指定 RTC 中断。例如，在 PC 上，RTC 位于 IRQ 8。第二个参数是我们的中断处理程序 rtc_interrupt——它将与其他中断处理程序共享中断线，因为它设置了 IRQF_SHARED 标志。由第四个参数我们看出，驱动程序的名称为“rtc”。因为这个设备允许共享中断线，所以它给 dev 型参传递了一个面向每个设备的实参值。

最后要展示的是处理程序本身：

```
static irqreturn_t rtc_interrupt(int irq, void *dev)
{
    /*
     * 可以是报警器中断、更新完成的中断或周期性中断
     * 我们把状态保存在 rtc_irq_data 的低字节中，
     * 而把从最后一次读取之后所接收的中断号保存在其余字节中
     */
    spin_lock (&rtc_lock);

    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

    if (rtc_status & RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

    spin_unlock (&rtc_lock);

    /*
     * 现在执行其余的操作
     */
    spin_lock(&rtc_task_lock);
    if (rtc_callback)
            rtc_callback->func(rtc_callback->private_data);
    spin_unlock(&rtc_task_lock);
    wake_up_interruptible(&rtc_wait);

    kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

    return IRQ_HANDLED;
}
```

只要计算机一接收到 RTC 中断，就会调用这个函数。首先要注意的是使用了自旋锁——第一次调用是为了保证 rtc_irq_data 不被 SMP 机器上的其他处理器同时访问，第二次调用避免 rtc_

callback 出现相同的情况。锁机制在第 10 章中进行讨论。

rtc_irq_data 变量是无符号长整数，存放有关 RTC 的信息，每次中断时都会更新以反映中断的状态。

接下来，如果设置了 RTC 周期性定时器，就要通过函数 mod_timer() 对其更新。定时器在第 11 章进行讨论。

代码的最后一部分——处于注释“现在执行其余的操作”下，会执行一个可能被预先设置好的回调函数。RTC 驱动程序允许注册一个回调函数，并在每个 RTC 中断到来时执行。

最后，这个函数会返回 IRQ_HANDLED，表明已经正确地完成了对此设备的操作。因为这个中断处理程序不支持共享，而且 RTC 也没有什么用来测试虚假中断的机制，所以该处理程序总是返回 IRQ_HANDLED。

7.6 中断上下文

当执行一个中断处理程序时，内核处于中断上下文（interrput context）中。让我们先回忆一下进程上下文。进程上下文是一种内核所处的操作模式，此时内核代表进程执行——例如，执行系统调用或运行内核线程。在进程上下文中，可以通过 current 宏关联当前进程。此外，因为进程是以进程上下文的形式连接到内核中的，因此，进程上下文可以睡眠，也可以调用调度程序。

与之相反，中断上下文和进程并没有什么瓜葛。与 current 宏也是不相干的（尽管它会指向被中断的进程）。因为没有后备进程，所以中断上下文不可以睡眠，否则又怎能再对它重新调度呢？因此，不能从中断上下文中调用某些函数。如果一个函数睡眠，就不能在你的中断处理程序中使用它——这是对什么样的函数可以在中断处理程序中使用的限制。

中断上下文具有较为严格的时间限制，因为它打断了其他代码。中断上下文中的代码应当迅速、简洁，尽量不要使用循环去处理繁重的工作。有一点非常重要，请永远牢记：中断处理程序打断了其他的代码（甚至可能是打断了在其他中断线上的另一中断处理程序）。正是因为这种异步执行的特性，所以所有的中断处理程序必须尽可能的迅速、简洁。尽量把工作从中断处理程序中分离出来，放在下半部来执行，因为下半部可以在更合适的时间运行。

中断处理程序栈的设置是一个配置选项。曾经，中断处理程序并不具有自己的栈。相反，它们共享所中断进程的内核栈[⊖]。内核栈的大小是两页，具体地说，在 32 位体系结构上是 8KB，在 64 位体系结构上是 16KB。因为在这种设置中，中断处理程序共享别人的堆栈，所以它们在栈中获取空间时必须非常节约。当然，内核栈本来就很有限，因此，所有的内核代码都应该谨慎利用它。

在 2.6 版早期的内核中，增加了一个选项，把栈的大小从两页减到一页，也就是在 32 位的系统上只提供 4KB 的栈。这就减轻了内存的压力，因为系统中每个进程原先都需要两页连续，且不可换出的内核内存。为了应对栈大小的减少，中断处理程序拥有了自己的栈，每个处理器一个，大小为一页。这个栈就称为中断栈，尽管中断栈的大小是原先共享栈的一半，但平均可用栈

⊖ 总得有一个进程在运行着。当没有进程可调度时，空任务运行。

空间大得多，因为中断处理程序把这一整页占为己有。

你的中断处理程序不必关心栈如何设置，或者内核栈的大小是多少。总而言之，尽量节约内核栈空间。

7.7 中断处理机制的实现

中断处理系统在 Linux 中的实现是非常依赖于体系结构的，想必你对此不会感到特别惊讶。实现依赖于处理器、所使用的中断控制器的类型、体系结构的设计及机器本身。

图 7-1 是中断从硬件到内核的路由。设备产生中断，通过总线把电信号发送给中断控制器。如果中断线是激活的（它们是允许被屏蔽的），那么中断控制器就会把中断发往处理器。在大多数体系结构中，这个工作就是通过电信号给处理器的特定管脚发送一个信号。除非在处理器上禁止该中断，否则，处理器会立即停止它正在做的事，关闭中断系统，然后跳到内存中预定义的位置开始执行那里的代码。这个预定义的位置是由内核设置的，是中断处理程序的入口点。

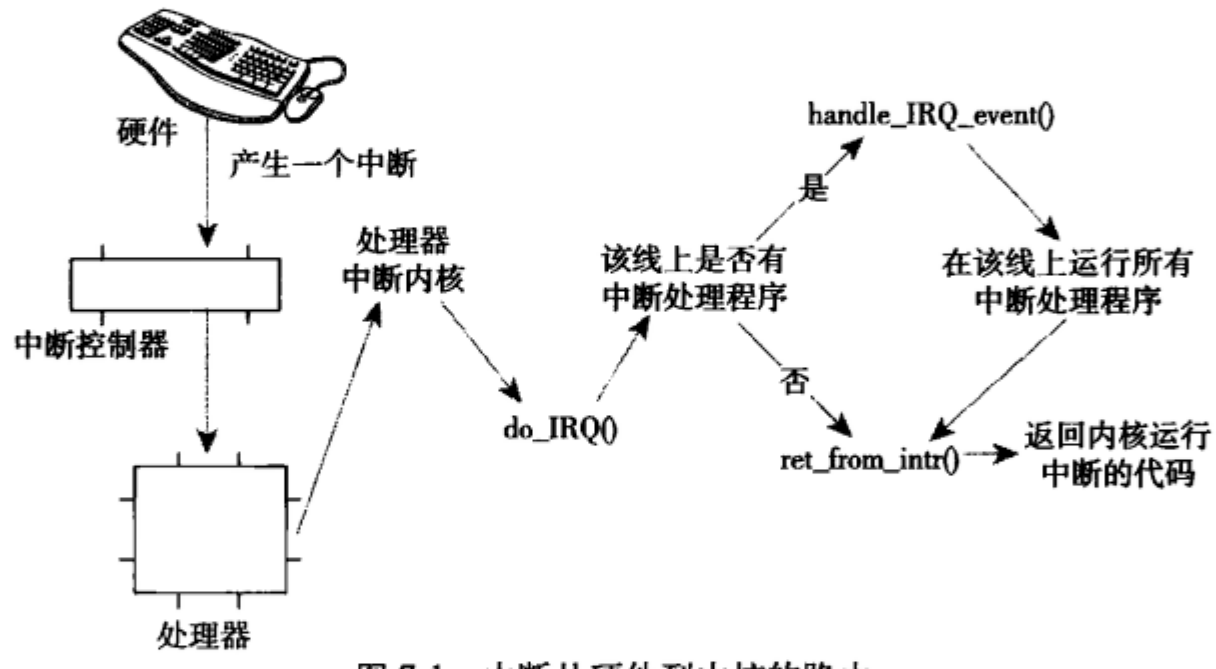

图 7-1 中断从硬件到内核的路由

在内核中，中断的旅程开始于预定义入口点，这类似于系统调用通过预定义的异常句柄进入内核。对于每条中断线，处理器都会跳到对应的一个唯一的位置。这样，内核就可知道所接收中断的 IRQ 号了。初始入口点只是在栈中保存这个号，并存放当前寄存器的值（这些值属于被中断的任务）；然后，内核调用函数 do_IRQ()。从这里开始，大多数中断处理代码是用 C 编写的——但它们依然与体系结构相关。

do_IRQ() 的声明如下：

```
unsigned int do_IRQ(struct pt_regs regs)
```

因为 C 的调用惯例是要把函数参数放在栈的顶部，因此 pt_regs 结构包含原始寄存器的值，这些值是以前在汇编入口例程中保存在栈中的。中断的值也会得以保存，所以，do_IRQ() 可以将它提取出来。

计算出中断号后，do_IRQ() 对所接收的中断进行应答，禁止这条线上的中断传递。在普通的 PC 机上，这些操作是由 mask_and_ack_8259A() 来完成的。

接下来，do_IRQ() 需要确保在这条中断线上有一个有效的处理程序，而且这个程序已经启动，但是当前并没有执行。如果是这样的话，do_IRQ() 就调用 handle_IRQ_event() 来运行为这条中断线所安装的中断处理程序。 handle_IRQ_event() 方法被定义在文件 kernel/irq/handler.c 中。

```
/**
 * handle_IRQ_event - irq action chain handler
 * @irq:        the interrupt number
 * @action:     the interrupt action chain for this irq
 *
 * Handles the action chain of an irq event
 */
irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action)
{
        irqreturn_t ret, retval = IRQ_NONE;
        unsigned int status = 0;

        if (!(action->flags & IRQF_DISABLED))
                local_irq_enable_in_hardirq();

        do {
                trace_irq_handler_entry(irq, action);
                ret = action->handler(irq, action->dev_id);
                trace_irq_handler_exit(irq, action, ret);
switch (ret) {
case IRQ_WAKE_THREAD:
                /*
                 * 把返回值设置为已处理，以便可疑的检查不再触发
                 */
                ret = IRQ_HANDLED;

                /*
                 * 捕获返回值为 WAKE_THREAD 的驱动程序，但是并不创建一个线程函数
                 */
                if (unlikely(!action->thread_fn)) {
                        warn_no_thread(irq, action);
                        break;
                }

                /*
                 * 为这次中断唤醒处理线程。万一线程崩溃且被杀死，我们仅仅假装已经处理了该中
                   断。上述的硬件中断（hardirq）处理程序已经禁止设备中断，因此杜绝 irq 产生
                 */
                if (likely(!test_bit(IRQTF_DIED,
                                     &action->thread_flags))) {
                        set_bit(IRQTF_RUNTHREAD, &action->thread_flags);
                        wake_up_process(action->thread);
                }

                /* Fall through to add to randomness */
case IRQ_HANDLED:
```

```
                status |= action->flags;
                break;

        default:
                break;
        }

        retval |= ret;
        action = action->next;
    } while (action);

    if (status & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    local_irq_disable();

    return retval;
}
```

首先，因为处理器禁止中断，这里要把它们打开，就必须在处理程序注册期间指定 IRQF_DISABLED 标志。回想一下，IRQF_DISABLED 表示处理程序必须在中断禁止的情况下运行。接下来，每个潜在的处理程序在循环中依次执行。如果这条线不是共享的，第一次执行后就退出循环。否则，所有的处理程序都要被执行。之后，如果在注册期间指定了 IRQF_SAMPLE_RANDOM 标志，则还要调用函数 add_interrupt_randomness()。这个函数使用中断间隔时间为随机数产生器产生熵。最后，再将中断禁止（do_IRQ() 期望中断一直是禁止的），函数返回。回到 do_IRQ()，该函数做清理工作并返回到初始入口点，然后再从这个入口点跳到函数 ret_from_intr()。

ret_from_intr() 例程类似于初始入口代码，以汇编语言编写。这个例程检查重新调度是否正在挂起（回想一下第 4 章，这意味着设置了 need_resched）。如果重新调度正在挂起，而且内核正在返回用户空间（也就是说，中断了用户进程），那么，schedule() 被调用。如果内核正在返回内核空间（也就是说，中断了内核本身），只有在 preempt_count 为 0 时，schedule() 才会被调用，否则，抢占内核便是不安全的。在 schedule() 返回之后，或者如果没有挂起的工作，那么，原来的寄存器被恢复，内核恢复到曾经中断的点。

在 x86 上，初始的汇编例程位于 arch/x86/kernel/entry_64.S（文件 entry_32.S 对应 32 位的 x86 体系架构），C 方法位于 arch/x86/kernel/irq.c。其他所支持的结构与此类似。

7.8 /proc/interrupts

procfs 是一个虚拟文件系统，它只存在于内核内存，一般安装于 /proc 目录。在 procfs 中读写文件都要调用内核函数，这些函数模拟从真实文件中读或写。与此相关的例子是 /proc/interrupts 文件，该文件存放的是系统中与中断相关的统计信息。下面是从单处理器 PC 上输出的信息：

```
           CPU0
 0:    3602371    XT-PIC    timer
 1:    3048       XT-PIC    i8042
 2:    0          XT-PIC    cascade
 4:    2689466    XT-PIC    uhci-hcd, eth0
 5:    0          XT-PIC    EMU10K1
 12:   85077      XT-PIC    uhci-hcd
 15:   24571      XT-PIC    aic7xxx
NMI:   0
LOC:   3602236
ERR:   0
```

第 1 列是中断线。在这个系统中，现有的中断号为 0 ～ 2、4、5、12 及 15。这里没有显示没有安装处理程序的中断线。第 2 列是一个接收中断数目的计数器。事实上，系统中的每个处理器都存在这样的列，但是，这个机器只有一个处理器。我们看到，时钟中断已接收 3602371 次中断[⊖]，这里，声卡（EMU10K1）没有接收一次中断（这表示机器启动以来还没有使用它）。第 3 列是处理这个中断的中断控制器。XT-PIC 对应于标准的 PC 可编程中断控制器。在具有 I/O APIC 的系统上，大多数中断会列出 IO-APIC-level 或 IO-APIC-edge，作为自己的中断控制器。最后一列是与这个中断相关的设备名字。这个名字是通过参数 devname 提供给函数 request_irq() 的，前面已讨论过了。如果中断是共享的（例子中的 4 号中断就是这种情况），则这条中断线上注册的所有设备都会列出来。

对于想深入探究 procfs 内部的人来说，procfs 代码位于 fs/proc 中。不必惊讶，提供 /proc/interrupts 的函数是与体系结构相关的，叫做 show_interrupts()。

7.9 中断控制

Linux 内核提供了一组接口用于操作机器上的中断状态。这些接口为我们提供了能够禁止当前处理器的中断系统，或屏蔽掉整个机器的一条中断线的能力，这些例程都是与体系结构相关的，可以在 <asm/system.h> 和 <asm/irq.h> 中找到。本章稍后给出的表 7-2 是接口的完整列表。

一般来说，控制中断系统的原因归根结底是需要提供同步。通过禁止中断，可以确保某个中断处理程序不会抢占当前的代码。此外，禁止中断还可以禁止内核抢占。然而，不管是禁止中断还是禁止内核抢占，都没有提供任何保护机制来防止来自其他处理器的并发访问。Linux 支持多处理器，因此，内核代码一般都需要获取某种锁，防止来自其他处理器对共享数据的并发访问。获取这些锁的同时也伴随着禁止本地中断。锁提供保护机制，防止来自其他处理器的并发访问，而禁止中断提供保护机制，则是防止来自其他中断处理程序的并发访问。第 9 章和第 10 章着重讨论同步的各种问题及其对策。因此，必须理解内核中断的控制接口。

7.9.1 禁止和激活中断

用于禁止当前处理器（仅仅是当前处理器）上的本地中断，随后又激活它们的语句为：

⊖ 作为一个练习，读过第 11 章后，你能在知道时钟产生的中断次数的情况下说出系统已经工作了多久了吗（根据 HZ 值）？知道时钟中断发生了多少次吗？

```
local_irq_disable();
/* 禁止中断 */
local_irq_enable();
```

这两个函数通常以单个汇编指令来实现（当然，这依赖于体系结构）。实际上，在x86中，local_irq_discable()仅仅是cli指令，而local_irq_enable()只不过是sti指令。cli和sti分别是对clear和set允许中断（allow interrupt）标志的汇编调用。换句话说，在发出中断的处理器上，它们将禁止和激活中断的传递。

如果在调用local_irq_discable()例程之前已经禁止了中断，那么该例程往往会带来潜在的危险；同样相应的local_irq_enable()例程也存在潜在危险，因为它将无条件地激活中断，尽管这些中断可能在开始时就是关闭的。所以我们需要一种机制把中断恢复到以前的状态而不是简单地禁止或激活。内核普遍关心这点是因为，内核中一个给定的代码路径既可以在中断激活的情况下达到，也可以在中断禁止的情况下达到，这取决于具体的调用链。例如，想象一下前面的代码片段是一个大函数的组成部分。这个函数被另外两个函数调用：其中一个函数禁止中断，而另一个函数不禁止中断。因为随着内核的不断增长，要想知道到达这个函数的所有代码路径将变得越来越困难，因此，在禁止中断之前保存中断系统的状态会更加安全一些。相反，在准备激活中断时，只需把中断恢复到它们原来的状态。

```
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 */
```

这些方法至少部分要以宏的形式实现，因此表面上flags参数(这些参数必须定义为unsigned long类型)是以值传递的。该参数包含具体体系结构的数据，也就是包含中断系统的状态。至少有一种体系结构把栈信息与值相结合（SPARC），因此flags不能传递给另一个函数（特别是它必须驻留在同一栈帧中）。基于这个原因，对local_irq_save()和对local_irq_restore()的调用必须在同一个函数中进行。

前面的所有函数既可以在中断中调用，也可以在进程上下文中调用。

不再使用全局的cli()

以前的内核中提供了一种“能够禁止系统中所有处理器上的中断”方法。而且，如果另一个处理器调用这个方法，那么它就不得不等待，直到中断重新被激活才能继续执行。这个函数就是cli()，相应的激活中断函数为sti()——虽然适用于所有体系结构，但完全以x86为中心。这些接口在2.5版本开发期间被取消了，相应地，所有的中断同步现在必须结合使用本地中断控制和自旋锁（在第9章中进行讨论）。这就意味着，为了确保对共享数据的互斥访问，以前代码仅仅需要通过全局禁止中断达到互斥，而现在则需要多做些工作了。

以前，驱动程序编写者可能假定在他们的中断处理程序中，任何访问共享数据地方都可以使用cli()提供互斥访问。cli()调用将确保没有其他的中断处理程序（因而只有它们特定的处理程序）会运行。此外，如果另一个处理器进入了cli保护区，那么它不可能继续运行，直到原来的处理器退出它们的cli()保护区，并调用了sti()后才能继续运行。

取消全局 cli() 有不少优点。首先，强制驱动程序编写者实现真正的加锁。要知道具有特定目的细粒度锁比全局锁要快许多，而且也完全吻合 cli() 的使用初衷。其次，这也使得很多代码更具流线型，避免了代码的成簇布局。所以由此得到的中断系统更简单也更易于理解。

7.9.2 禁止指定中断线

在前面的内容中，我们看到了禁止整个处理器上所有中断的函数。在某些情况下，只禁止整个系统中一条特定的中断线就够了。这就是所谓的屏蔽掉（masking out）一条中断线。作为例子，你可能想在对中断的状态操作之前禁止设备中断的传递。为此，Linux 提供了四个接口：

```
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);
```

前两个函数禁止中断控制器上指定的中断线，即禁止给定中断向系统中所有处理器的传递。另外，函数只有在当前正在执行的所有处理程序完成后，disable_irq() 才能返回。因此，调用者不仅要确保不在指定线上传递新的中断，同时还要确保所有已经开始执行的处理程序已全部退出。函数 disable_irq_nosync() 不会等待当前中断处理程序执行完毕。

函数 synchronize_riq() 等待一个特定的中断处理程序的退出。如果该处理程序正在执行，那么该函数必须退出后才能返回。

对这些函数的调用可以嵌套。但要记住在一条指定的中断线上，对 disable_irq() 或 disable_irq_nosync() 的每次调用，都需要相应地调用一次 enable_irq()。只有在对 enable_irq() 完成最后一次调用后，才真正重新激活了中断线。例如，如果 disable_irq() 被调用了两次，那么直到第二次调用 enable_irq() 后，才能真正地激活中断线。

所有这三个函数可以从中断或进程上下文中调用，而且不会睡眠。但如果从中断上下文中调用，就要特别小心！例如，当你正在处理一条中断线时，并不想激活它（回想当某个处理程序的中断线正在被处理时，它被屏蔽掉）。

禁止多个中断处理程序共享的中断线是不合适的。禁止中断线也就禁止了这条线上所有设备的中断传递。因此，用于新设备的驱动程序应该倾向于不使用这些接口[⊖]。根据规范，PCI 设备必须支持中断线共享，因此，它们根本不应该使用这些接口。所以，disable_irq() 及其相关函数在老式传统设备（如 PC 并口）的驱动程序中更容易被找到。

7.9.3 中断系统的状态

通常有必要了解中断系统的状态（如中断是禁止的还是激活的），或者你当前是否正处于中断上下文的执行状态中。

宏 irqs_disable() 定义在 <asm/system.h> 中。如果本地处理器上的中断系统被禁止，则它返

⊖ 很多老式设备，尤其是 ISA 设备，不提供方法检测它们是否产生了中断。因为这一点，ISA 的中断线常常不能共享。由于 PCI 规范要求中断共享，因此，现代基于 PCI 的设备支持中断共享。在当代计算机中，几乎所有的中断线都可以共享。

回非 0；否则返回 0。

在 <linux/hardirq.h> 中定义的两个宏提供一个用来检查内核的当前上下文的接口，它们是：

```
in_interrupt()
in_irq()
```

第一个宏最有用：如果内核处于任何类型的中断处理中，它返回非 0，说明内核此刻正在执行中断处理程序，或者正在执行下半部处理程序。宏 in_irq() 只有在内核确实正在执行中断处理程序时才返回非 0。

通常情况下，你要检查自己是否处于进程上下文中。也就是说，你希望确保自己不在中断上下文中。这种情况很常见，因为代码要做一些像睡眠这样只能从进程上下文中做的事。如果 in_interrupt() 返回 0，则此刻内核处于进程上下文中。

是的，名字有点混淆，但可以对它们的含义稍加区别。表 7-2 是中断控制方法和其描述的摘要。

表 7-2 中断控制方法的列表

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

7.10 小结

本章介绍了中断，它是一种由设备使用的硬件资源异步向处理器发信号。实际上，中断就是由硬件来打断操作系统。

大多数现代硬件都通过中断与操作系统通信。对给定硬件进行管理的驱动程序注册中断处理程序，是为了响应并处理来自相关硬件的中断。中断过程所做的工作包括应答并重新设置硬件，从设备拷贝数据到内存以及反之，处理硬件请求，并发送新的硬件请求。

内核提供的接口包括注册和注销中断处理程序、禁止中断、屏蔽中断线以及检查中断系统的状态。表 7-2 提供了这些函数的概述。

因为中断打断了其他代码的执行（进程，内核本身，甚至其他中断处理程序），它们必须赶快执行完。但通常是还有很多工作要做。为了在大量的工作与必须快速执行之间求得一种平衡，内核把处理中断的工作分为两半。中断处理程序，也就是上半部在本章讨论。现在，让我们了解下半部。