

第 10 章

中断与时钟

本章导读

本章主要讲解 Linux 设备驱动编程中的中断与定时器处理。由于中断服务程序的执行并不存在于进程上下文中，所以要求中断服务程序的时间要尽量短。因此，Linux 在中断处理中引入了顶半部和底半部分离的机制。另外，内核对时钟的处理也采用中断方式，而内核软件定时器最终依赖于时钟中断。

10.1 节讲解中断和定时器的概念及处理流程。

10.2 节讲解 Linux 中断处理程序的架构，以及顶半部、底半部之间的关系。

10.3 节讲解 Linux 中断编程的方法，涉及申请和释放中断、使能和屏蔽中断以及中断底半部 tasklet、工作队列、软中断机制和 `threaded_irq`。

10.4 节讲解多个设备共享同一个中断号时的中断处理过程。

10.5 节和 10.6 节分别讲解 Linux 设备驱动编程中定时器的编程以及内核延时的方法。

10.1 中断与定时器

所谓中断是指 CPU 在执行程序的过程中，出现了某些突发事件急待处理，CPU 必须暂停当前程序的执行，转去处理突发事件，处理完毕后又返回原程序被中断的位置继续执行。

根据中断的来源，中断可分为内部中断和外部中断，内部中断的中断源来自 CPU 内部（软件中断指令、溢出、除法错误等，例如，操作系统从用户态切换到内核态需借助 CPU 内部的软件中断），外部中断的中断源来自 CPU 外部，由外设提出请求。

根据中断是否可以屏蔽，中断可分为可屏蔽中断与不可屏蔽中断（NMI），可屏蔽中断可以通过设置中断控制器寄存器等方法被屏蔽，屏蔽后，该中断不再得到响应，而不可屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，中断可分为向量中断和非向量中断。采用向量中断的 CPU 通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址，进入该入口地址后，再通过软件判断中断标志来识别具体是哪个中断。

也就是说，向量中断由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

一个典型的非向量中断服务程序如代码清单 10.1 所示，它先判断中断源，然后调用不同中断源的中断服务程序。

代码清单 10.1 非向量中断服务程序的典型结构

```

1  irq_handler()
2  {
3      ...
4      int int_src = read_int_status();           /* 读硬件的中断相关寄存器 */
5      switch (int_src) {                         /* 判断中断源 */
6          case DEV_A:
7              dev_a_handler();
8              break;
9          case DEV_B:
10             dev_b_handler();
11             break;
12         ...
13     default:
14         break;
15   }
16   ...
17 }
```

嵌入式系统以及 x86 PC 中大多包含可编程中断控制器（PIC），许多 MCU 内部就集成了 PIC。如在 80386 中，PIC 是两片 i8259A 芯片的级联。通过读写 PIC 的寄存器，程序员可以屏蔽 / 使能某中断及获得中断状态，前者一般通过中断 MASK 寄存器完成，后者一般通过中断 PEND 寄存器完成。

定时器在硬件上也依赖中断来实现，图 10.1 所示为典型的嵌入式微处理器内可编程间隔定时器（PIT）的工作原理，它接收一个时钟输入，当时钟脉冲到来时，将目前计数值增 1 并与预先设置的计数值（计数目标）比较，若相等，证明计数周期满，并产生定时器中断且复位目前计数值。



图 10.1 PIT 定时器的工作原理

在 ARM 多核处理器里最常用的中断控制器是 GIC (Generic Interrupt Controller)，如图 10.2 所示，它支持 3 种类型的中断。

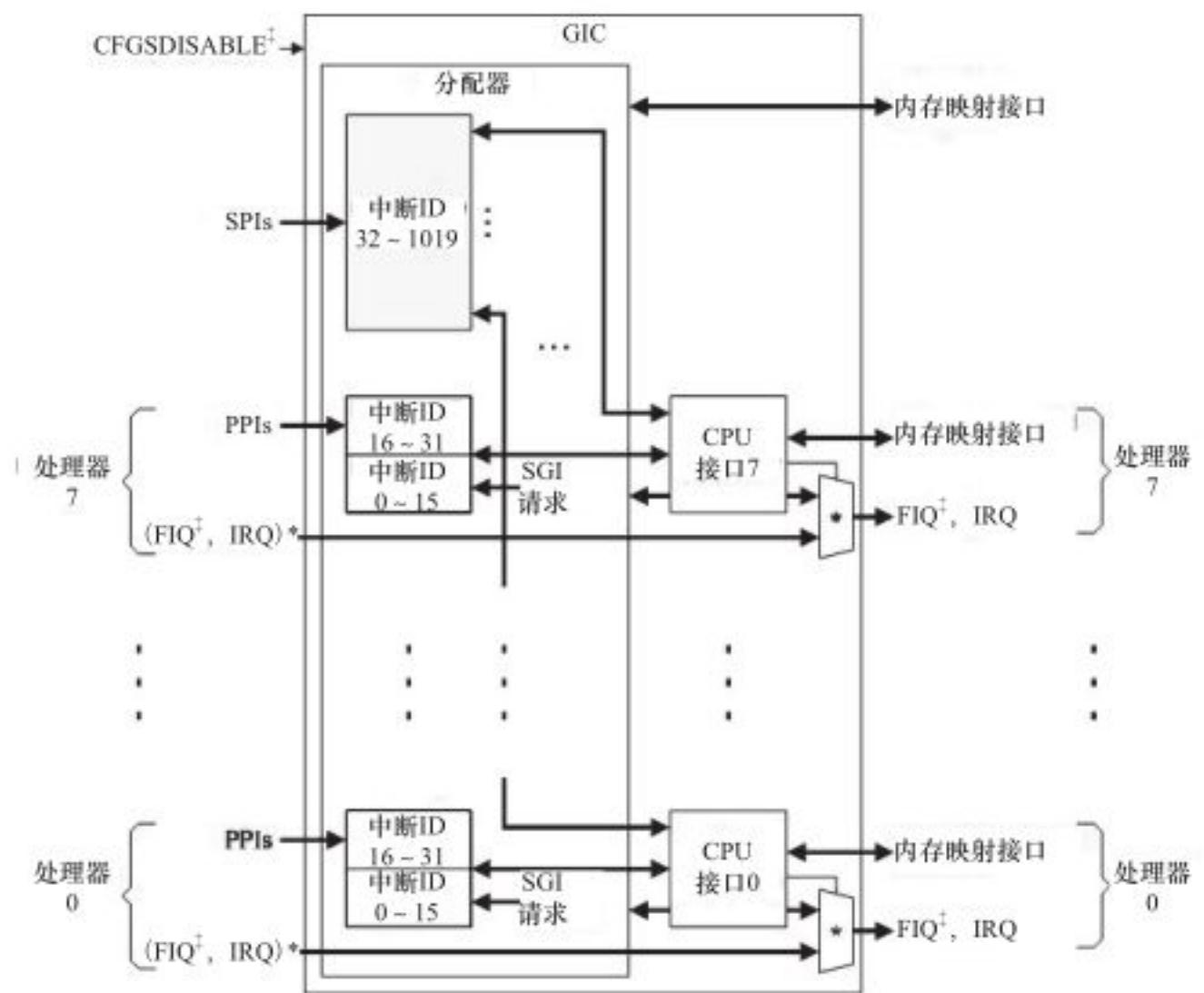


图 10.2 ARM 多核处理器里的 GIC

SGI (Software Generated Interrupt)：软件产生的中断，可以用于多核的核间通信，一个 CPU 可以通过写 GIC 的寄存器给另外一个 CPU 产生中断。多核调度用的 IPI_WAKEUP、IPI_TIMER、IPI_RESCHEDULE、IPI_CALL_FUNC、IPI_CALL_FUNC_SINGLE、IPI_CPU_STOP、IPI_IRQ_WORK、IPI_COMPLETION 都是由 SGI 产生的。

PPI (Private Peripheral Interrupt)：某个 CPU 私有外设的中断，这类外设的中断只能发给绑定的那个 CPU。

SPI (Shared Peripheral Interrupt)：共享外设的中断，这类外设的中断可以路由到任何一个 CPU。

对于 SPI 类型的中断，内核可以通过如下 API 设定中断触发的 CPU 核：

```
extern int irq_set_affinity (unsigned int irq, const struct cpumask *m);
```

在 ARM Linux 默认情况下，中断都是在 CPU0 上产生的，比如，我们可以通过如下代码把中断 `irq` 设定到 CPU i 上去：

```
irq_set_affinity(irq, cpumask_of(i));
```

10.2 Linux 中断处理程序架构

设备的中断会打断内核进程中的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽量短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大量的耗时处理。

图 10.3 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽量短和中断处理需完成的工作尽量大之间找到一个平衡点，Linux 将中断处理程序分解为两个半部：顶半部（Top Half）和底半部（Bottom Half）。

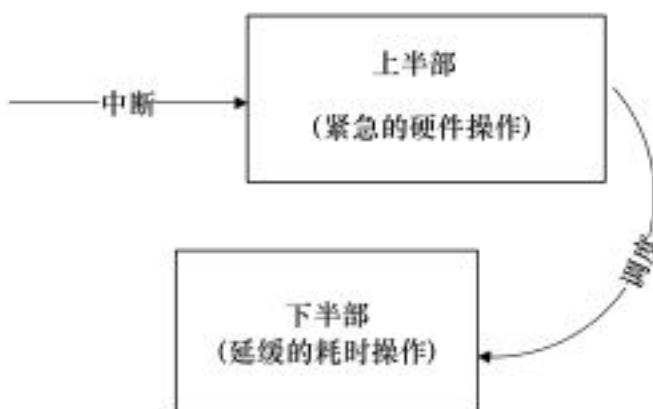


图 10.3 Linux 中断处理机制

顶半部用于完成尽量少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态，并在清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，从而可以服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，需用它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部相对来说并不是非常紧急的，而且相对比较耗时，不在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，僵化地认为 Linux 设备驱动中的中断处理一定要分两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。



其他操作系统中对中断的处理也采用了类似于 Linux 的方法，真正的硬件中断服务程序都应该尽量短。因此，许多操作系统都提供了中断上下文和非中断上下文相结合的机制，将中断的耗时工作保留到非中断上下文去执行。例如，在 VxWorks 中，网络设备包接收中断到来后，中断服务程序会通过 `netJobAdd()` 函数将耗时的包接收和上传工作交给 `tNetTask` 任务去执行。

在 Linux 中，查看 /proc/interrupts 文件可以获得系统中中断的统计信息，并能统计出每一个中断号上的中断在每个 CPU 上发生的次数，具体如图 10.4 所示。

	CPU0	CPU1	
0:	119	0	IO-APIC-edge timer
1:	14444	0	IO-APIC-edge i8042
8:	0	0	IO-APIC-edge rtc0
9:	0	0	IO-APIC-fasteoi acpi
12:	2483	0	IO-APIC-edge i8042
14:	0	0	IO-APIC-edge ata_pmix
15:	11679	0	IO-APIC-edge ata_pmix
19:	14749	0	IO-APIC 19-fasteoi ehci_hcd:usb1, eth0
20:	17379	0	IO-APIC 20-fasteoi vboxguest
21:	105402	0	IO-APIC 21-fasteoi 0000:00:0d.0
22:	30	0	IO-APIC 22-fasteoi ohci_hcd:usb2
NMI:	0	0	Non-maskable interrupts
LOC:	564521	478738	Local timer interrupts
SPU:	0	0	Spurious interrupts
PMI:	0	0	Performance monitoring interrupts
IWI:	0	0	IRQ work interrupts
RTR:	0	0	APIC ICR read retries
RES:	263390	382214	Rescheduling interrupts
CAL:	52	50	Function call interrupts
TLB:	830	1201	TLB shootdowns
TRM:	0	0	Thermal event interrupts
THR:	0	0	Threshold APIC interrupts
MCE:	0	0	Machine check exceptions
MCP:	40	40	Machine check polls
ERR:	0	0	
MIS:	0	0	

图 10.4 Linux 中的中断统计信息

10.3 Linux 中断编程

10.3.1 申请和释放中断

在 Linux 设备驱动中，使用中断的设备需要申请和释放对应的中断，并分别使用内核提供的 request_irq() 和 free_irq() 函数。

1. 申请 irq

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
                const char *name, void *dev);
```

irq 是要申请的硬件中断号。

handler 是向系统登记的中断处理函数（顶半部），是一个回调函数，中断发生时，系统调用这个函数，dev 参数将被传递给它。

irqflags 是中断处理的属性，可以指定中断的触发方式以及处理方式。在触发方式方

面，可以是 IRQF_TRIGGER_RISING、IRQF_TRIGGER_FALLING、IRQF_TRIGGER_HIGH、IRQF_TRIGGER_LOW 等。在处理方式方面，若设置了 IRQF_SHARED，则表示多个设备共享中断，dev 是要传递给中断服务程序的私有数据，一般设置为这个设备的设备结构体或者 NULL。

request_irq() 返回 0 表示成功，返回 -EINVAL 表示中断号无效或处理函数指针为 NULL，返回 -EBUSY 表示中断已经被占用且不能共享。

```
int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                     unsigned long irqflags, const char *devname, void *dev_id);
```

此函数与 request_irq() 的区别是 devm_ 开头的 API 申请的是内核“managed”的资源，一般不需要在出错处理和 remove() 接口里再显式的释放。有点类似 Java 的垃圾回收机制。比如，对于 at86rf230 驱动，如下的补丁中改用 devm_request_irq() 后就删除了 free_irq()，该补丁对应的内核 commit ID 是 652355c5。

```
--- a/drivers/net/ieee802154/at86rf230.c
+++ b/drivers/net/ieee802154/at86rf230.c
@@ -1190,24 +1190,22 @@ static int at86rf230_probe(struct spi_device *spi)
     if (rc)
         goto err_hw_init;

-    rc = request_irq(spi->irq, irq_handler, IRQF_SHARED,
-                     dev_name(&spi->dev), lp);
+    rc = devm_request_irq(&spi->dev, spi->irq, irq_handler, IRQF_SHARED,
+                         dev_name(&spi->dev), lp);
     if (rc)
         goto err_hw_init;

     /* Read irq status register to reset irq line */
     rc = at86rf230_read_subreg(lp, RG_IRQ_STATUS, 0xff, 0, &status);
     if (rc)
-        goto err_irq;
+        goto err_hw_init;

     rc = ieee802154_register_device(lp->dev);
     if (rc)
-        goto err_irq;
+        goto err_hw_init;

     return rc;

-err_irq:
-    free_irq(spi->irq, lp);
 err_hw_init:
     flush_work(&lp->irqwork);
     spi_set_drvdata(spi, NULL);
@@ -1232,7 +1230,6 @@ static int at86rf230_remove(struct spi_device *spi)
```

```

at86rf230_write_subreg(lp, SR_IRQ_MASK, 0);
ieee802154_unregister_device(lp->dev);

- free_irq(spi->irq, lp);
flush_work(&lp->irqwork);

if (gpio_is_valid(pdata->slp_tr))

```

顶半部 handler 的类型 irq_handler_t 定义为：

```

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

```

2. 释放 irq

与 request_irq() 相对应的函数为 free_irq(), free_irq() 的原型为：

```
void free_irq(unsigned int irq, void *dev_id);
```

free_irq() 中参数的定义与 request_irq() 相同。

10.3.2 使能和屏蔽中断

下列 3 个函数用于屏蔽一个中断源：

```

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

```

disable_irq_nosync() 与 disable_irq() 的区别在于前者立即返回，而后者等待目前的中断处理完成。由于 disable_irq() 会等待指定的中断被处理完，因此如果在 n 号中断的顶半部调用 disable_irq(n)，会引起系统的死锁，这种情况下，只能调用 disable_irq_nosync(n)。

下列两个函数（或宏，具体实现依赖于 CPU 的体系结构）将屏蔽本 CPU 内的所有中断：

```

#define local_irq_save(flags) ...
void local_irq_disable(void);

```

前者会将目前的中断状态保留在 flags 中（注意 flags 为 unsigned long 类型，被直接传递，而不是通过指针），后者直接禁止中断而不保存状态。

与上述两个禁止中断对应的恢复中断的函数（或宏）是：

```

#define local_irq_restore(flags) ...
void local_irq_enable(void);

```

以上各以 local_ 开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 实现底半部的机制主要有 tasklet、工作队列、软中断和线程化 irq。

1. tasklet

tasklet 的使用较简单，它的执行上下文是软中断，执行时机通常是顶半部返回的时候。我们只需要定义 tasklet 及其处理函数，并将两者关联则可，例如：

```
void my_tasklet_func(unsigned long); /* 定义一个处理函数 */
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
/* 定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data) 函数相关联 */
```

代码 DECLARE_TASKLET(my_tasklet, my_tasklet_func, data) 实现了定义名称为 my_tasklet 的 tasklet，并将其与 my_tasklet_func() 这个函数绑定，而传入这个函数的参数为 data。

在需要调度 tasklet 的时候引用一个 tasklet_schedule() 函数就能使系统在适当的时候进行调度运行：

```
tasklet_schedule(&my_tasklet);
```

使用 tasklet 作为底半部处理中断的设备驱动程序模板如代码清单 10.2 所示（仅包含与中断相关的部分）。

代码清单 10.2 tasklet 使用模板

```
1 /* 定义 tasklet 和底半部函数并将它们关联 */
2 void xxx_do_tasklet(unsigned long);
3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
5 /* 中断处理底半部 */
6 void xxx_do_tasklet(unsigned long)
7 {
8     ...
9 }
10
11 /* 中断处理顶半部 */
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {
14     ...
15     tasklet_schedule(&xxx_tasklet);
16     ...
17 }
18
19 /* 设备驱动模块加载函数 */
20 int __init xxx_init(void)
21 {
22     ...
23     /* 申请中断 */
24     result = request_irq(xxx_irq, xxx_interrupt,
25             0, "xxx", NULL);
26     ...
27     return IRQ_HANDLED;
28 }
```

```

29
30 /* 设备驱动模块卸载函数 */
31 void __exit xxx_exit(void)
32 {
33 ...
34 /* 释放中断 */
35 free_irq(xxx_irq, xxx_interrupt);
36 ...
37 }

```

上述程序在模块加载函数中申请中断（第 24 ~ 25 行），并在模块卸载函数中释放它（第 35 行）。对应于 `xxx_irq` 的中断处理程序被设置为 `xxx_interrupt()` 函数，在这个函数中，第 15 行的 `tasklet_schedule(&xxx_tasklet)` 调度被定义的 tasklet 函数 `xxx_do_tasklet()` 在适当的时候执行。

2. 工作队列

工作队列的使用方法和 tasklet 非常相似，但是工作队列的执行上下文是内核线程，因此可以调度和睡眠。下面的代码用于定义一个工作队列和一个底半部执行函数：

```

struct work_struct my_wq;           /* 定义一个工作队列 */
void my_wq_func(struct work_struct *work); /* 定义一个处理函数 */

```

通过 `INIT_WORK()` 可以初始化这个工作队列并将工作队列与处理函数绑定：

```

INIT_WORK(&my_wq, my_wq_func);
/* 初始化工作队列并将其与处理函数绑定 */

```

与 `tasklet_schedule()` 对应的用于调度工作队列执行的函数为 `schedule_work()`，如：

```

schedule_work(&my_wq);           /* 调度工作队列执行 */

```

与代码清单 10.2 对应的使用工作队列处理中断底半部的设备驱动程序模板如代码清单 10.3 所示（仅包含与中断相关的部分）。

代码清单 10.3 工作队列使用模板

```

1 /* 定义工作队列和关联函数 */
2 struct work_struct xxx_wq;
3 void xxx_do_work(struct work_struct *work);
4
5 /* 中断处理底半部 */
6 void xxx_do_work(struct work_struct *work)
7 {
8 ...
9 }
10
11 /* 中断处理顶半部 */
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {

```

```

14 ...
15 schedule_work(&xxx_wq);
16 ...
17 return IRQ_HANDLED;
18 }
19
20 /* 设备驱动模块加载函数 */
21 int xxx_init(void)
22 {
23 ...
24 /* 申请中断 */
25 result = request_irq(xxx_irq, xxx_interrupt,
26 0, "xxx", NULL);
27 ...
28 /* 初始化工作队列 */
29 INIT_WORK(&xxx_wq, xxx_do_work);
30 ...
31 }
32
33 /* 设备驱动模块卸载函数 */
34 void xxx_exit(void)
35 {
36 ...
37 /* 释放中断 */
38 free_irq(xxx_irq, xxx_interrupt);
39 ...
40 }

```

与代码清单 10.2 不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 29 行）。

工作队列早期的实现是在每个 CPU 核上创建一个 worker 内核线程，所有在这个核上调度的工作都在该 worker 线程中执行，其并发性显然差强人意。在 Linux 2.6.36 以后，转而实现了“Concurrency-managed workqueues”，简称 cmwq，cmwq 会自动维护工作队列的线程池以提高并发性，同时保持了 API 的向后兼容。

3. 软中断

软中断（Softirq）也是一种传统的底半部处理机制，它的执行时机通常是顶半部返回的时候，tasklet 是基于软中断实现的，因此也运行于软中断上下文。

在 Linux 内核中，用 softirq_action 结构体表征一个软中断，这个结构体包含软中断处理函数指针和传递给该函数的参数。使用 open_softirq() 函数可以注册软中断对应的处理函数，而 raise_softirq() 函数可以触发一个软中断。

软中断和 tasklet 运行于软中断上下文，仍然属于原子上下文的一种，而工作队列则运行于进程上下文。因此，在软中断和 tasklet 处理函数中不允许睡眠，而在工作队列处理函数中允许睡眠。

local_bh_disable() 和 local_bh_enable() 是内核中用于禁止和使能软中断及 tasklet 底半部

机制的函数。

内核中采用 softirq 的地方包括 HI_SOFTIRQ、TIMER_SOFTIRQ、NET_TX_SOFTIRQ、NET_RX_SOFTIRQ、SCSI_SOFTIRQ、TASKLET_SOFTIRQ 等，一般来说，驱动的编写者不会也不宜直接使用 softirq。

第 9 章异步通知所基于的信号也类似于中断，现在，总结一下硬中断、软中断和信号的区别：硬中断是外部设备对 CPU 的中断，软中断是中断底半部的一种处理机制，而信号则是由内核（或其他进程）对某个进程的中断。在涉及系统调用的场合，人们也常说通过软中断（例如 ARM 为 swi）陷入内核，此时软中断的概念是指由软件指令引发的中断，和我们这个地方说的 softirq 是两个完全不同的概念，一个是 software，一个是 soft。

需要特别说明的是，软中断以及基于软中断的 tasklet 如果在某段时间内大量出现的话，内核会把后续软中断放入 ksoftirqd 内核线程中执行。总的来说，中断优先级高于软中断，软中断又高于任何一个线程。软中断适度线程化，可以缓解高负载情况下系统的响应。

4. threaded_irq

在内核中，除了可以通过 request_irq()、devm_request_irq() 申请中断以外，还可以通过 request_threaded_irq() 和 devm_request_threaded_irq() 申请。这两个函数的原型为：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn,
                         unsigned long flags, const char *name, void *dev);
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                             irq_handler_t handler, irq_handler_t thread_fn,
                             unsigned long irqflags, const char *devname,
                             void *dev_id);
```

由此可见，它们比 request_irq()、devm_request_irq() 多了一个参数 thread_fn。用这两个 API 申请中断的时候，内核会为相应的中断号分配一个对应的内核线程。注意这个线程只针对这个中断号，如果其他中断也通过 request_threaded_irq() 申请，自然会得到新的内核线程。

参数 handler 对应的函数执行于中断上下文，thread_fn 参数对应的函数则执行于内核线程。如果 handler 结束的时候，返回值是 IRQ_WAKE_THREAD，内核会调度对应线程执行 thread_fn 对应的函数。

request_threaded_irq() 和 devm_request_threaded_irq() 支持在 irqflags 中设置 IRQF_ONESHOT 标记，这样内核会自动帮助我们在中断上下文中屏蔽对应的中断号，而在内核调度 thread_fn 执行后，重新使能该中断号。对于我们无法在上半部清除中断的情况，IRQF_ONESHOT 特别有用，避免了中断服务程序一退出，中断就洪泛的情况。

handler 参数可以设置为 NULL，这种情况下，内核会用默认的 irq_default_primary_handler() 代替 handler，并会使用 IRQF_ONESHOT 标记。irq_default_primary_handler() 定义为：

```
/*
 * Default primary interrupt handler for threaded interrupts. Is
```

```

* assigned as primary handler when request_threaded_irq is called
* with handler == NULL. Useful for oneshot interrupts.
*/
static irqreturn_t irq_default_primary_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

```

10.3.4 实例：GPIO 按键的中断

`drivers/input/keyboard/gpio_keys.c` 是一个放之四海皆准的 GPIO 按键驱动，为了让该驱动在特定的电路板上工作，通常只需要修改 `arch/arm/mach-xxx` 下的板文件或者修改 device tree 对应的 dts。该驱动会为每个 GPIO 申请中断，在 `gpio_keys_setup_key()` 函数中进行。注意最后一个参数 `bdata`，会被传入中断服务程序。

代码清单 10.4 GPIO 按键驱动中断申请

```

1 static int gpio_keys_setup_key(struct platform_device *pdev,
2                                struct input_dev *input,
3                                struct gpio_button_data *bdata,
4                                const struct gpio_keys_button *button)
5 {
6     ...
7
8     error = request_any_context_irq(bdata->irq, isr, irqflags, desc, bdata);
9     if (error < 0) {
10         dev_err(dev, "Unable to claim irq %d; error %d\n",
11                 bdata->irq, error);
12         goto fail;
13     }
14     ...
15 }

```

第 8 行的 `request_any_context_irq()` 会根据 GPIO 控制器本身的“上级”中断是否为 `threaded_irq` 来决定采用 `request_irq()` 还是 `request_threaded_irq()`。一组 GPIO（如 32 个 GPIO）虽然每个都提供一个中断，并且都有中断号，但是在硬件上一组 GPIO 通常是嵌套在上一级的中断控制器上的一个中断。

`request_any_context_irq()` 也有一个变体是 `devm_request_any_context_irq()`。

在 GPIO 按键驱动的 `remove_key()` 函数中，会释放 GPIO 对应的中断，如代码清单 10.5 所示。

代码清单 10.5 GPIO 按键驱动中断释放

```

1 static void gpio_remove_key(struct gpio_button_data *bdata)
2 {
3     free_irq(bdata->irq, bdata);

```

```

4     if (bdata->timer_debounce)
5         del_timer_sync(&bdata->timer);
6     cancel_work_sync(&bdata->work);
7     if (gpio_is_valid(bdata->button->gpio))
8         gpio_free(bdata->button->gpio);
9 }

```

GPIO 按键驱动的中断处理比较简单，没有明确地分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6 GPIO 按键驱动中断处理程序

```

1 static irqreturn_t gpio_keys_gpio_isr(int irq, void *dev_id)
2 {
3     struct gpio_button_data *bdata = dev_id;
4
5     BUG_ON(irq != bdata->irq);
6
7     if (bdata->button->wakeup)
8         pm_stay_awake(bdata->input->dev.parent);
9     if (bdata->timer_debounce)
10        mod_timer(&bdata->timer,
11                   jiffies + msecs_to_jiffies(bdata->timer_debounce));
12    else
13        schedule_work(&bdata->work);
14
15    return IRQ_HANDLED;
16 }

```

第 3 行直接从 dev_id 取出了 bdata，这就是对应的那个 GPIO 键的数据结构，之后根据情况启动 timer 以进行 debounce 或者直接调度工作队列去汇报按键事件。在 GPIO 按键驱动初始化的时候，通过 INIT_WORK(&bdata->work, gpio_keys_gpio_work_func) 初始化了 bdata->work，对应的处理函数是 gpio_keys_gpio_work_func()，如代码清单 10.7 所示。

代码清单 10.7 GPIO 按键驱动的工作队列底半部

```

1 static void gpio_keys_gpio_work_func(struct work_struct *work)
2 {
3     struct gpio_button_data *bdata =
4         container_of(work, struct gpio_button_data, work);
5
6     gpio_keys_gpio_report_event(bdata);
7
8     if (bdata->button->wakeup)
9         pm_relax(bdata->input->dev.parent);
10 }

```

观察其中的第 3 ~ 4 行，它通过 container_of() 再次从 work_struct 反向解析出了 bdata。

原因是 work_struct 本身在定义时，就嵌入在 gpio_button_data 结构体内。读者朋友们应该掌握 Linux 的这种可以到处获取一个结构体指针的技巧，它实际上类似于面向对象里面的“this”指针。

```
struct gpio_button_data {
    const struct gpio_keys_button *button;
    struct input_dev *input;
    struct timer_list timer;
    struct work_struct work;
    unsigned int timer_debounce; /* in msecs */
    unsigned int irq;
    spinlock_t lock;
    bool disabled;
    bool key_pressed;
};
```

10.4 中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，Linux 支持这种中断共享。下面是中断共享的使用方法。

- 1) 共享中断的多个设备在申请中断时，都应该使用 IRQF_SHARED 标志，而且一个设备以 IRQF_SHARED 申请某中断成功的前提是该中断未被申请，或该中断虽然被申请了，但是之前申请该中断的所有设备也都以 IRQF_SHARED 标志申请该中断。

- 2) 尽管内核模块可访问的全局地址都可以作为 request_irq(..., void *dev_id) 的最后一个参数 dev_id，但是设备结构体指针显然是可传入的最佳参数。

- 3) 在中断到来时，会遍历执行共享此中断的所有中断处理程序，直到某一个函数返回 IRQ_HANDLED。在中断处理程序顶半部中，应根据硬件寄存器中的信息比照传入的 dev_id 参数迅速地判断是否为本设备的中断，若不是，应迅速返回 IRQ_NONE，如图 10.5 所示。

代码清单 10.8 给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

代码清单 10.8 共享中断编程模板

```
1 /* 中断处理顶半部 */
2 irqreturn_t xxx_interrupt(int irq, void *dev_id)
3 {
4     ...
5     int status = read_int_status(); /* 获知中断源 */
```

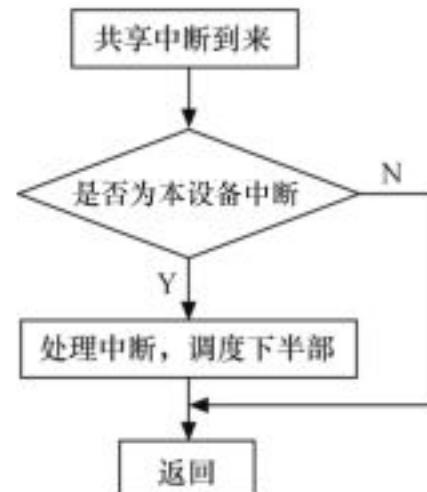


图 10.5 共享中断的处理

```

6   if(!is_myint(dev_id,status))           /* 判断是否为本设备中断 */
7       return IRQ_NONE;                  /* 不是本设备中断，立即返回 */
8
9   /* 是本设备中断，进行处理 */
10  ...
11  return IRQ_HANDLED;                 /* 返回 IRQ_HANDLED 表明中断已被处理 */
12 }
13
14 /* 设备驱动模块加载函数 */
15 int xxx_init(void)
16 {
17     ...
18     /* 申请共享中断 */
19     result = request_irq(sh_irq, xxx_interrupt,
20             IRQF_SHARED, "xxx", xxx_dev);
21     ...
22 }
23
24 /* 设备驱动模块卸载函数 */
25 void xxx_exit(void)
26 {
27     ...
28     /* 释放中断 */
29     free_irq(xxx_irq, xxx_interrupt);
30     ...
31 }

```

10.5 内核定时器

10.5.1 内核定时器编程

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后检测各定时器是否到期，到期后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序会唤起 TIMER_SOFTIRQ 软中断，运行当前处理器上到期的所有定时器。

在 Linux 设备驱动编程中，可以利用 Linux 内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动工程师在多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux 内核所提供的用于操作定时器的数据结构和函数如下。

1. timer_list

在 Linux 内核中，timer_list 结构体的一个实例对应一个定时器，如代码清单 10.9 所示。

代码清单 10.9 timer_list 结构体

```

1 struct timer_list {
2     /*
3      * All fields that change during normal runtime grouped to the

```

```

4         * same cacheline
5         */
6         struct list_head entry;
7         unsigned long expires;
8         struct tvec_base *base;
9
10        void (*function)(unsigned long);
11        unsigned long data;
12
13        int slack;
14
15 #ifdef CONFIG_TIMER_STATS
16         int start_pid;
17         void *start_site;
18         char start_comm[16];
19#endif
20 #ifdef CONFIG_LOCKDEP
21         struct lockdep_map lockdep_map;
22#endif
23    };

```

当定时器期满后，其中第 10 行的 function() 成员将被执行，而第 11 行的 data 成员则是传入其中的参数，第 7 行的 expires 则是定时器到期的时间 (jiffies)。

如下代码定义一个名为 my_timer 的定时器：

```
struct timer_list my_timer;
```

2. 初始化定时器

init_timer 是一个宏，它的原型等价于：

```
void init_timer(struct timer_list * timer);
```

上述 init_timer() 函数初始化 timer_list 的 entry 的 next 为 NULL，并给 base 指针赋值。

TIMER_INITIALIZER(_function, _expires, _data) 宏用于赋值定时器结构体的 function、expires、data 和 base 成员，这个宏等价于：

```
#define TIMER_INITIALIZER(_function, _expires, _data) \
    .entry = { .prev = TIMER_ENTRY_STATIC }, \
    .function = (_function), \
    .expires = (_expires), \
    .data = (_data), \
    .base = &boot_tvec_bases,
```

DEFINE_TIMER(_name, _function, _expires, _data) 宏是定义并初始化定时器成员的“快捷方式”，这个宏定义为：

```
#define DEFINE_TIMER(_name, _function, _expires, _data)
```

```
struct timer_list _name =
    TIMER_INITIALIZER(_function, _expires, _data)
```

此外，`setup_timer()`也可用于初始化定时器并赋值其成员，其源代码为：

```
#define __setup_timer(_timer, _fn, _data, _flags)
    do {
        __init_timer((_timer), (_flags));
        (_timer)->function = (_fn);
        (_timer)->data = (_data);
    } while (0)
```

3. 增加定时器

```
void add_timer(struct timer_list * timer);
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4. 删除定时器

```
int del_timer(struct timer_list * timer);
```

上述函数用于删除定时器。

`del_timer_sync()`是`del_timer()`的同步版，在删除一个定时器时需等待其被处理完，因此该函数的调用不能发生在中断上下文中。

5. 修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

上述函数用于修改定时器的到期时间，在新的被传入的`expires`到来后才会执行定时器函数。

代码清单 10.10 给出了一个完整的内核定时器使用模板，在大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 10.10 内核定时器使用模板

```
1 /* xxx 设备结构体 */
2 struct xxx_dev {
3     struct cdev cdev;
4     ...
5     timer_list xxx_timer;           /* 设备要使用的定时器 */
6 };
7
8 /* xxx 驱动中的某函数 */
9 xxx_func1(...)
10 {
11     struct xxx_dev *dev = filp->private_data;
12     ...
13     /* 初始化定时器 */
14     init_timer(&dev->xxx_timer);
```

```

15     dev->xxx_timer.function = &xxx_do_timer;
16     dev->xxx_timer.data = (unsigned long)dev;
17             /* 设备结构体指针作为定时器处理函数参数 */
18     dev->xxx_timer.expires = jiffies + delay;
19     /* 添加(注册)定时器 */
20     add_timer(&dev->xxx_timer);
21     ...
22 }
23
24 /* xxx 驱动中的某函数 */
25 xxx_func2(...)
26 {
27 ...
28     /* 删除定时器 */
29     del_timer(&dev->xxx_timer);
30     ...
31 }
32
33 /* 定时器处理函数 */
34 static void xxx_do_timer(unsigned long arg)
35 {
36     struct xxx_device *dev = (struct xxx_device *)arg;
37     ...
38     /* 调度定时器再执行 */
39     dev->xxx_timer.expires = jiffies + delay;
40     add_timer(&dev->xxx_timer);
41     ...
42 }

```

从代码清单第 18、39 行可以看出，定时器的到期时间往往是在目前 jiffies 的基础上添加一个时延，若为 Hz，则表示延迟 1s。

在定时器处理函数中，在完成相应的工作后，往往会延后 expires 并将定时器再次添加到内核定时器链表中，以便定时器能再次被触发。

此外，Linux 内核支持 tickless 和 NO_HZ 模式后，内核也包含对 hrtimer（高精度定时器）的支持，它可以支持到微秒级别的精度。内核也定义了 hrtimer 结构体，hrtimer_set_expires()、hrtimer_start_expires()、hrtimer_forward_now()、hrtimer_restart() 等类似的 API 来完成 hrtimer 的设置、时间推移以及到期回调。我们可以从 sound/soc/fsl/imx-pcm-fiq.c 中提取出一个使用范例，如代码清单 10.11 所示。

代码清单 10.11 内核高精度定时器 (hrtimer) 使用模板

```

1 static enum hrtimer_restart snd_hrtimer_callback(struct hrtimer *hrt)
2 {
3     ...
4
5     hrtimer_forward_now(hrt, ns_to_ktime(iprtd->poll_time_ns));
6

```

```

7         return HRTIMER_RESTART;
8     }
9
10    static int snd_imx_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
11    {
12        struct snd_pcm_runtime *runtime = substream->runtime;
13        struct imx_pcm_runtime_data *iprtd = runtime->private_data;
14
15        switch (cmd) {
16        case SNDDRV_PCM_TRIGGER_START:
17        case SNDDRV_PCM_TRIGGER_RESUME:
18        case SNDDRV_PCM_TRIGGER_PAUSE_RELEASE:
19            ...
20            hrtimer_start(&iprtd->hrt, ns_to_ktime(iprtd->poll_time_ns),
21                          HRTIMER_MODE_REL);
22            ...
23    }
24
25    static int snd_imx_open(struct snd_pcm_substream *substream)
26    {
27        ...
28        hrtimer_init(&iprtd->hrt, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
29        iprtd->hrt.function = snd_hrtimer_callback;
30
31        ...
32        return 0;
33    }
34    static int snd_imx_close(struct snd_pcm_substream *substream)
35    {
36        ...
37        hrtimer_cancel(&iprtd->hrt);
38        ...
39    }

```

第 28 ~ 29 行在声卡打开的时候通过 `hrtimer_init()` 初始化了 `hrtimer`，并指定回调函数为 `snd_hrtimer_callback()`；在启动播放（第 15 ~ 21 行 `SNDDRV_PCM_TRIGGER_START`）等时刻通过 `hrtimer_start()` 启动了 `hrtimer`；`iprtd->poll_time_ns` 纳秒后，时间到 `snd_hrtimer_callback()` 函数在中断上下文被执行，它紧接着又通过 `hrtimer_forward_now()` 把 `hrtimer` 的时间前移了 `iprtd->poll_time_ns` 纳秒，这样周而复始；直到声卡被关闭，第 37 行又调用了 `hrtimer_cancel()` 取消在 open 时初始化的 `hrtimer`。

10.5.2 内核中延迟的工作 `delayed_work`

对于周期性的任务，除了定时器以外，在 Linux 内核中还可以利用一套封装得很好的快捷机制，其本质是利用工作队列和定时器实现，这套快捷机制就是 `delayed_work`，`delayed_work` 结构体的定义如代码清单 10.12 所示。

代码清单 10.12 delayed_work 结构体

```

1 struct delayed_work {
2     struct work_struct work;
3     struct timer_list timer;
4
5     /* target workqueue and CPU ->timer uses to queue ->work */
6     struct workqueue_struct *wq;
7     int cpu;
8 };

```

我们可以通过如下函数调度一个 delayed_work 在指定的延时后执行：

```
int schedule_delayed_work(struct delayed_work *work, unsigned long delay);
```

当指定的 delay 到来时，delayed_work 结构体中的 work 成员 work_func_t 类型成员 func() 会被执行。work_func_t 类型定义为：

```
typedef void (*work_func_t)(struct work_struct *work);
```

其中，delay 参数的单位是 jiffies，因此一种常见的用法如下：

```
schedule_delayed_work(&work, msecs_to_jiffies(poll_interval));
```

msecs_to_jiffies() 用于将毫秒转化为 jiffies。

如果要周期性地执行任务，通常会在 delayed_work 的工作函数中再次调用 schedule_delayed_work()，周而复始。

如下函数用来取消 delayed_work：

```
int cancel_delayed_work(struct delayed_work *work);
int cancel_delayed_work_sync(struct delayed_work *work);
```

10.5.3 实例：秒字符设备

下面我们编写一个字符设备“second”（即“秒”）的驱动，它在被打开的时候初始化一个定时器并将其添加到内核定时器链表中，每秒输出一次当前的 jiffies（为此，定时器处理函数中每次都要修改新的 expires），整个程序如代码清单 10.13 所示。

代码清单 10.13 使用内核定时器的 second 字符设备驱动

```

1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/mmc.h>
4 #include <linux/init.h>
5 #include <linux/cdev.h>
6 #include <linux/slab.h>
7 #include <linux/uaccess.h>
8
9 #define SECOND_MAJOR 248

```

```

10
11 static int second_major = SECOND_MAJOR;
12 module_param(second_major, int, S_IRUGO);
13
14 struct second_dev {
15     struct cdev cdev;
16     atomic_t counter;
17     struct timer_list s_timer;
18 };
19
20 static struct second_dev *second_devp;
21
22 static void second_timer_handler(unsigned long arg)
23 {
24     mod_timer(&second_devp->s_timer, jiffies + HZ);      /* 触发下一次定时 */
25     atomic_inc(&second_devp->counter);                      /* 增加秒计数 */
26
27     printk(KERN_INFO "current jiffies is %ld\n", jiffies);
28 }
29
30 static int second_open(struct inode *inode, struct file *filp)
31 {
32     init_timer(&second_devp->s_timer);
33     second_devp->s_timer.function = &second_timer_handler;
34     second_devp->s_timer.expires = jiffies + HZ;
35
36     add_timer(&second_devp->s_timer);
37
38     atomic_set(&second_devp->counter, 0);                  /* 初始化秒计数为 0 */
39
40     return 0;
41 }
42
43 static int second_release(struct inode *inode, struct file *filp)
44 {
45     del_timer(&second_devp->s_timer);
46
47     return 0;
48 }
49
50 static ssize_t second_read(struct file *filp, char __user *buf, size_t count,
51   loff_t * ppos)
52 {
53     int counter;
54
55     counter = atomic_read(&second_devp->counter);
56     if (put_user(counter, (int *)buf))                         /* 复制 counter 到 userspace */
57         return -EFAULT;
58     else
59         return sizeof(unsigned int);
60 }

```

```
61
62 static const struct file_operations second_fops = {
63     .owner = THIS_MODULE,
64     .open = second_open,
65     .release = second_release,
66     .read = second_read,
67 };
68
69 static void second_setup_cdev(struct second_dev *dev, int index)
70 {
71     int err, devno = MKDEV(second_major, index);
72
73     cdev_init(&dev->cdev, &second_fops);
74     dev->cdev.owner = THIS_MODULE;
75     err = cdev_add(&dev->cdev, devno, 1);
76     if (err)
77         printk(KERN_ERR "Failed to add second device\n");
78 }
79
80 static int __init second_init(void)
81 {
82     int ret;
83     dev_t devno = MKDEV(second_major, 0);
84
85     if (second_major)
86         ret = register_chrdev_region(devno, 1, "second");
87     else {
88         ret = alloc_chrdev_region(&devno, 0, 1, "second");
89         second_major = MAJOR(devno);
90     }
91     if (ret < 0)
92         return ret;
93
94     second_devp = kzalloc(sizeof(*second_devp), GFP_KERNEL);
95     if (!second_devp) {
96         ret = -ENOMEM;
97         goto fail_malloc;
98     }
99
100    second_setup_cdev(second_devp, 0);
101
102    return 0;
103
104 fail_malloc:
105     unregister_chrdev_region(devno, 1);
106     return ret;
107 }
108 module_init(second_init);
109
110 static void __exit second_exit(void)
```

```

111 {
112     cdev_del(&second_devp->cdev);
113     kfree(second_devp);
114     unregister_chrdev_region(MKDEV(second_major, 0), 1);
115 }
116 module_exit(second_exit);
117
118 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
119 MODULE_LICENSE("GPL v2");

```

在 second 的 open() 函数中，将启动定时器，此后每 1s 会再次运行定时器处理函数，在 second 的 release() 函数中，定时器被删除。

second_dev 结构体中的原子变量 counter 用于秒计数，每次在定时器处理函数中调用的 atomic_inc() 会令其原子性地增 1，second 的 read() 函数会将这个值返回给用户空间。

本书配套的 Ubuntu 中 /home/baohua/develop/training/kernel/drivers/second/ 包含了 second 设备驱动以及 second_test.c 用户空间测试程序，运行 make 命令编译得到 second.ko 和 second_test，加载 second.ko 内核模块并创建 /dev/second 设备文件节点：

```
# mknod /dev/second c 248 0
```

代码清单 10.14 给出了 second_test.c 这个应用程序，它打开 /dev/second，其后不断地读取自 /dev/second 设备文件打开以后经历的秒数。

代码清单 10.14 second 设备用户空间测试程序

```

1 #include ...
2
3 main()
4 {
5     int fd;
6     int counter = 0;
7     int old_counter = 0;
8
9     /* 打开 /dev/second 设备文件 */
10    fd = open("/dev/second", O_RDONLY);
11    if (fd != -1) {
13        while (1) {
15            read(fd, &counter, sizeof(unsigned int)); /* 读目前经历的秒数 */
16            if(counter!=old_counter) {
18                printf("seconds after open /dev/second :%d\n",counter);
19                old_counter = counter;
20            }
21        }
22    } else {
25        printf("Device open failure\n");
26    }
27 }

```

运行 second_test 后，内核将不断地输出目前的 jiffies 值：

```
[13935.122093] current jiffies is 13635122
[13936.124441] current jiffies is 13636124
[13937.126078] current jiffies is 13637126
[13952.832648] current jiffies is 13652832
[13953.834078] current jiffies is 13653834
[13954.836090] current jiffies is 13654836
[13955.838389] current jiffies is 13655838
[13956.840453] current jiffies is 13656840
...
```

从上述内核的打印消息也可以看出，本书配套 Ubuntu 上的每秒 jiffies 大概走 1000 次。而应用程序将不断输出自 /dev/second 打开以后经历的秒数：

```
# ./second_test
seconds after open /dev/second :1
seconds after open /dev/second :2
seconds after open /dev/second :3
seconds after open /dev/second :4
seconds after open /dev/second :5
...

```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了下列 3 个函数以分别进行纳秒、微秒和毫秒延迟：

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待，它根据 CPU 频率进行一定次数的循环。有时候，人们在软件中进行下面的延迟：

```
void delay(unsigned int time)
{
    while(time--);
}
```

ndelay()、udelay() 和 mdelay() 函数的实现方式原理与此类似。内核在启动时，会运行一个延迟循环校准（Delay Loop Calibration），计算出 lpj（Loops Per Jiffy），内核启动时会打印如下类似信息：

```
Calibrating delay loop... 530.84 BogoMIPS (lpj=1327104)
```

如果我们直接在 bootloader 传递给内核的 bootargs 中设置 lpj=1327104，则可以省掉这个

校准的过程，节省约百毫秒级的开机时间。

毫秒时延（以及更大的秒时延）已经比较大了，在内核中，最好不要直接使用 mdelay() 函数，这将耗费 CPU 资源，对于毫秒级以上时延，内核提供了下述函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠参数指定的时间为 millisecs，msleep()、ssleep() 不能被打断，而 msleep_interruptible() 则可以被打断。



受系统 Hz 以及进程调度的影响，msleep() 类似函数的精度是有限的。

10.6.2 长延迟

在内核中进行延迟的一个很直观的方法是比较当前的 jiffies 和目标 jiffies（设置为当前 jiffies 加上时间间隔的 jiffies），直到未来的 jiffies 达到目标 jiffies。代码清单 10.15 给出了使用忙等待先延迟 100 个 jiffies 再延迟 2s 的实例。

代码清单 10.15 忙等待时延实例

```
1 /* 延迟 100 个 jiffies */
2 unsigned long delay = jiffies + 100;
3 while(time_before(jiffies, delay));
4
5 /* 再延迟 2s */
6 unsigned long delay = jiffies + 2*Hz;
7 while(time_before(jiffies, delay));
```

与 time_before() 对应的还有一个 time_after()，它们在内核中定义为（实际上只是将传入的未来时间 jiffies 和被调用时的 jiffies 进行一个简单的比较）：

```
#define time_after(a,b)          \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)(b) - (long)(a) < 0))
#define time_before(a,b)   time_after(b,a)
```

为了防止在 time_before() 和 time_after() 的比较过程中编译器对 jiffies 的优化，内核将其定义为 volatile 变量，这将保证每次都会重新读取这个变量。因此 volatile 更多的作用还是避免这种读合并。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，睡着延迟是在等待的时间到来之前进程处于睡眠

状态，CPU 资源被其他进程使用。schedule_timeout() 可以使当前任务休眠至指定的 jiffies 之后再重新被调度执行，msleep() 和 msleep_interruptible() 在本质上都是依靠包含了 schedule_timeout() 的 schedule_timeout_uninterruptible() 和 schedule_timeout_interruptible() 来实现的，如代码清单 10.16 所示。

代码清单 10.16 schedule_timeout() 的使用

```

1 void msleep(unsigned int msecs)
2 {
3     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5     while (timeout)
6         timeout = schedule_timeout_uninterruptible(timeout);
7 }
8
9 unsigned long msleep_interruptible(unsigned int msecs)
10 {
11     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13     while (timeout && !signal_pending(current))
14         timeout = schedule_timeout_interruptible(timeout);
15     return jiffies_to_msecs(timeout);
16 }
```

实际上，schedule_timeout() 的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒与参数对应的进程。

代码清单 10.16 中第 6 行和第 14 行分别调用 schedule_timeout_uninterruptible() 和 schedule_timeout_interruptible()，这两个函数的区别在于前者在调用 schedule_timeout() 之前置进程状态为 TASK_INTERRUPTIBLE，后者置进程状态为 TASK_UNINTERRUPTIBLE，如代码清单 10.17 所示。

代码清单 10.17 schedule_timeout_interruptible() 和 schedule_timeout_uninterruptible()

```

1 signed long __sched schedule_timeout_interruptible(signed long timeout)
2 {
3     __set_current_state(TASK_INTERRUPTIBLE);
4     return schedule_timeout(timeout);
5 }
6
7 signed long __sched schedule_timeout_uninterruptible(signed long timeout)
8 {
9     __set_current_state(TASK_UNINTERRUPTIBLE);
10    return schedule_timeout(timeout);
11 }
```

另外，下面两个函数可以将当前进程添加到等待队列中，从而在等待队列上睡眠。当超

时发生时，进程将被唤醒（后者可以在超时前被打断）：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned long timeout);
```

10.7 总结

Linux 的中断处理分为两个半部，顶半部处理紧急的硬件操作，底半部处理不紧急的耗时操作。tasklet 和工作队列都是调度中断底半部的良好机制，tasklet 基于软中断实现。内核定时器也依靠软中断实现。

内核中的延时可以采用忙等待或睡眠等待，为了充分利用 CPU 资源，使系统有更好的吞吐性能，在对延迟时间的要求并不是很精确的情况下，睡眠等待通常是值得推荐的，而 `ndelay()`、`udelay()` 忙等待机制在驱动中通常是为了配合硬件上的短时延迟要求。