LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 10 章 中断与时钟

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，因此，要求中断服务程序的时间尽可能地短。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，内核中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，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，禁止和使能中断，以及中断底半部 tasklet、工作队列、软中断机制等。

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时 CPU 必须暂停执行当前的程序，转去处理突发事件，处理完毕后 CPU 又返回原程序被中断的位置并继续执行。

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。

根据是否可以屏蔽中断分为可屏蔽中断与不屏蔽中断（NMI），可屏蔽中断可以通过屏蔽字被屏蔽，屏蔽后，该中断不再得到响应，而不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，中断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    {
7      case DEV_A:
8        dev_a_handler();
9        break;
10     case DEV_B:
11       dev_b_handler();
12       break;
13     ...
14     default:
15       break;
16   }
```

```
17   ...
18 }
```

由于向量中断使用方便，目前有许多工程师创造了一些在非向量中断的处理器上模拟向量中断的方法。

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 给出了典型的嵌入式微处理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，产生定时器中断并复位目前计数值。

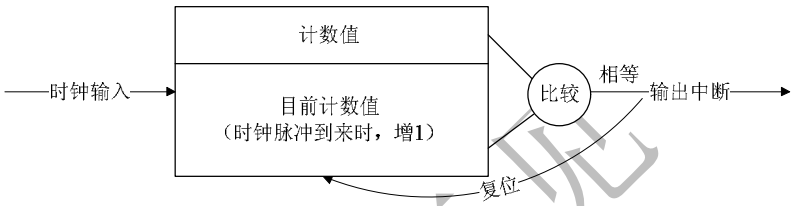

图 10.1　定时器工作原理

10.2 Linux 中断处理程序架构

设备的中断会打断内核中进程的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽可能地短小精悍。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 10.2 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽可能短和中断处理需完成大量工作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）。

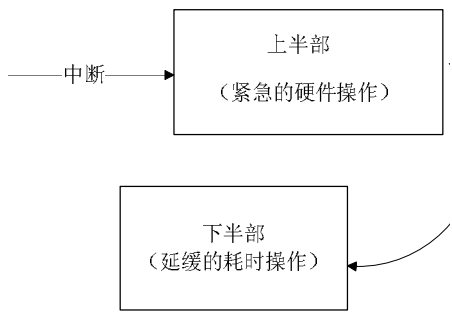

图 10.2　Linux 中断处理机制

顶半部完成尽可能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，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，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。底半部则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 系统的方法，真正的硬件中断服务程序都应该尽可能短。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 系统中，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 系统中，查看/proc/interrupts 文件可以获得系统中断的统计信息，如下所示。在单处理器的系统中，第一列是中断号，第二列是向 CPU0 产生该中断的次数，之后的是对于中断的描述。

```
           CPU0
  0:     135253          XT-PIC          timer
  1:         22          XT-PIC          i8042
  2:          0          XT-PIC          cascade
  8:          1          XT-PIC          rtc
 10:        108         XT-PIC          etho
 11:       3707          XT-PIC          BusLogic BT-958
 12:        313          XT-PIC          i8042
 15:          4         XT-PIC          idel
NMI:          0
ERR           0
```

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，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,
                    void (*handler)(int irq, void *dev_id, struct pt_regs
*regs),
                    unsigned long irqflags,
                    const char * devname,
                    void *dev_id);
```

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，是一个回调函数，中断发生时，系统调用这个函数，dev_id 参数将被传递给它。

irqflags 是中断处理的属性，若设置了 SA_INTERRUPT，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了 SA_SHIRQ，则表示多个设备共享中断，dev_id 在中断共享时会用到，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-INVAL 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。

2．释放 IRQ

与 request_irq()向对应的函数为 free_irq()，free_irq()的原型如下：

```
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。注意，这 3 个函数作用于可编程中断控制器，因此，对系统内的所有 CPU 都生效。

下列两个函数将屏蔽本 CPU 内的所有中断。

```
void local_irq_save(unsigned long 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 被直接传递，而不是通过指针传递。后者直接禁止中断。

与上述两个禁止中断对应的恢复中断的方法如下：

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

以上各 local_开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 系统实现底半部的机制主要有 tasklet、工作队列和软中断。

1．tasklet

tasklet 的使用较简单，我们只需要定义 tasklet 及其处理函数并将两者关联，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，如下所示：

```
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, struct pt_regs
*regs)
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      SA_INTERRUPT, "xxx", NULL);
26  ...
27 }
28
29 /*设备驱动模块卸载函数*/
30 void __exit xxx_exit(void)
31 {
32  ...
33  /*释放中断*/
34  free_irq(xxx_irq, xxx_interrupt);
35  ...
36 }
```

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 34 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）调度的 tasklet 函数 xxx_do_tasklet()在适当的时候得到　执行。

上述代码第 12 行显示中断处理程序顶半部的返回类型为 irqreturn_t，它定义为 int，中断处理程序顶半部一般返回 IRQ_HANDLED。

2．工作队列

工作队列的使用方法和 tasklet 非常相似，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(unsigned long); /*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，如下所示：

```
INIT_WORK(&my_wq, (void (*)(void *)) my_wq_func, NULL);
  /*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(unsigned long);
4
5  /*中断处理底半部*/
6  void xxx_do_work(unsigned long)
7  {
8   ...
9  }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
13 {
14  ...
15  schedule_work(&xxx_wq);
16  ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int xxx_init(void)
21 {
22  ...
23  /*申请中断*/
24  result = request_irq(xxx_irq, xxx_interrupt,
25      SA_INTERRUPT, "xxx", NULL);
26  ...
27  /*初始化工作队列*/
28  INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
29  ...
30 }
31
32 /*设备驱动模块卸载函数*/
33 void xxx_exit(void)
34 {
35  ...
36  /*释放中断*/
37  free_irq(xxx_irq, xxx_interrupt);
38  ...
39 }
```

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 28 行）。

尽管 Linux 专家们多建议在设备第一次打开时才申请设备的中断并在最后一次关闭时释放中断以尽量减少中断被这个设备占用的时间，但是，大多数情况下，为求省事，大多数驱动工程师还是将中断申请和释放的工作放在了设备驱动的模块加载和卸载函数中。

3．软中断

软中断是用软件方式模拟硬件中断的概念，实现宏观上的异步执行效果，tasklet 也是基于软中断实现的。

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，软中断通常是硬中断服务程序对内核的中断，而信号则是由内核（或其他进程）对某个进程的中断。

在 Linux 内核中，用 softirq_action 结构体表征一个软中断，这个结构体中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 仍然运行于中断上下文，而工作队列则运行于进程上下文。因此，软中断和 tasklet 处理函数中不能睡眠，而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和 tasklet 底半部机制的函数。

10.3.4 实例：S3C2410 实时钟中断

S3C2410 处理器内部集成了实时钟（RTC）模块，该模块能够在系统断电的情况下由后备电池供电继续工作，其主要功能相对于一个时钟，记录年、月、日、时、分、秒等。S3C2410 的 RTC 可产生两种中断：周期节拍（tick）中断和报警（alarm）中断，前者相当于一个周期性的定时器，后者相当于一个“闹钟”，它在预先设定的时间到来时产生中断。

S3C2410 实时钟设备驱动的 open()函数中，会申请它将要使用的中断，如代码清单 10.4 所示。

代码清单 10.4 S3C2410 实时钟驱动的 open()函数

```
1  /*S3C2410 实时钟驱动的 open()函数 */
2  static int s3c2410_rtc_open(void)
3  {
4    int ret;
5    /*申请 alarm 中断*/
6    ret = request_irq(s3c2410_rtc_alarmno, s3c2410_rtc_alarmirq,
7                      SA_INTERRUPT, "s3c2410-rtc alarm", NULL);
8    /*中断号被占用*/
9    if (ret)
10     printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_alarmno);
11
12   /*申请 tick 中断*/
13     ret  =  request_irq(s3c2410_rtc_tickno,  s3c2410_rtc_tickirq,
SA_INTERRUPT,
14     "s3c2410-rtc tick", NULL);
15   /*中断号被占用*/
16   if (ret)
17   {
18     printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_tickno);
19     goto tick_err;
20   }
21
```

```
    22   return ret;
    23
    24   tick_err: free_irq(s3c2410_rtc_alarmno, NULL); /*释放 alarm 中断*/
    25   return ret;
    26 }
```

S3C2410 实时钟设备驱动的 release()函数中，会释放它将要使用的中断，如代码清单 10.5 所示。

代码清单 10.5　S3C2410 实时钟驱动的 release()函数

```
    1 static void s3c2410_rtc_release(void)
    2 {
    3     s3c2410_rtc_setpie(0);
    4     /* 释放中断 */
    5     free_irq(s3c2410_rtc_alarmno, NULL);
    6     free_irq(s3c2410_rtc_tickno, NULL);
    7 }
```

S3C2410 实时钟驱动的中断处理比较简单，不需要分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6　S3C2410 实时钟驱动的中断处理程序

```
    1  /*中断处理*/
    2  static irqreturn_t s3c2410_rtc_alarmirq(int irq, void *id, struct
pt_regs *r)
    3  {
    4   rtc_update(1, RTC_AF | RTC_IRQF);
    5   return IRQ_HANDLED;
    6  }
    7
    8  static irqreturn_t s3c2410_rtc_tickirq(int irq, void *id, struct
pt_regs *r)
    9  {
    10   rtc_update(1, RTC_PF | RTC_IRQF);
    11   return IRQ_HANDLED;
    12 }
```

上述代码中调用的 rtc_update() 函数定义于 linux-2.6.16\arch\arm\common\Rtctime.c 文件中，被各种实时钟共享，如代码清单 10.7 所示。

代码清单 10.7　实时钟更新的 rtc_update()函数

```
    1  void rtc_update(unsigned long num, unsigned long events)
    2  {
    3   spin_lock(&rtc_lock);
    4   rtc_irq_data = (rtc_irq_data + (num << 8)) | events;
    5   spin_unlock(&rtc_lock);
    6
    7   wake_up_interruptible(&rtc_wait);/*唤醒等待队列*/
    8   kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);/*释放信号*/
    9  }
    10 EXPORT_SYMBOL(rtc_update);
```

上述中断处理程序并没有底半部（或者说没有严格意义上的 tasklet、工作队列或软中断底半部），实际上，它只是唤醒一个等待队列 rtc_wait，而这个等待队列的唤醒也将导致一个阻塞的进程被执行（这个阻塞的进程可看做底半部）。现在我们看到，等待队列可以作为中断处理程序顶半部和进程同步的一种良好机制。但是，任何情况下，都不能在顶半部等待一个等待队列，而只能唤醒。

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，PCI 设备即是如此。Linux 2.6 支持这种中断共享。下面是中断共享的使用方法。

- 共享中断的多个设备在申请中断时都应该使用 SA_SHIRQ 标志，而且一个设备以 SA_SHIRQ 申请某中断成功的前提是之前申请该中断的所有设备也都以 SA_SHIRQ 标志申请该中断。
- 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是可传入的最佳参数。
- 在中断到来时，所有共享此中断的中断处理程序都会被执行，在中断处理程序顶半部中，应迅速地根据硬件寄存器中的信息比照传入的 dev_id 参数判断是否是本设备的中断，若不是，应迅速返回，如图 10.3 所示。

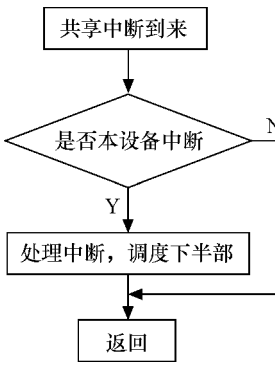

图 10.3 共享中断的处理

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 共享中断编程的模板

```
1  /*中断处理顶半部*/
2  irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
3  {
4   ...
5   int status = read_int_status();/*获知中断源*/
6   if(!is_myint(dev_id,status))/*判断是否是本设备中断*/
7   {
8   return IRQ_NONE;  /*立即返回*/
9   }
10  ...
11  return IRQ_HANDLED;
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      SA_SHIRQ, "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，时钟中断处理程序执行 update_process_timers()函数，该函数调用 run_local_timers()函数，这个函数处理 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    struct list_head entry; //定时器列表
3    unsigned long expires; //定时器到期时间
4    void (*function)(unsigned long); //定时器处理函数
5    unsigned long data; //作为参数被传入定时器处理函数
6    struct timer_base_s *base;
7 };
```

当定时器期满后，其中第 5 行的 function()成员将被执行，而第 4 行的 data 成员则是传入其中的参数，第 3 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

```
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，这个宏的定义如下所示：

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

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

```
#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，其源代码如下：

```
static inline void setup_timer(struct timer_list * timer,
                    void (*function)(unsigned long),
                    unsigned long data)
{
     timer->function = function;
     timer->data = data;
     init_timer(timer);
}
```

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，主要在多处理器系统中使用，如果编译内核时不支持 SMP，del_timer_sync()和 del_timer()等价。

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，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  {
4   struct cdev cdev;
5   ...
6   timer_list xxx_timer;/*设备要使用的定时器*/
7  };
8
9  /*xxx 驱动中的某函数*/
10 xxx_func1(...)
11 {
12  struct xxx_dev *dev = filp->private_data;
13  ...
14  /*初始化定时器*/
15  init_timer(&dev->xxx_timer);
16  dev->xxx_timer.function = &xxx_do_timer;
17  dev->xxx_timer.data = (unsigned long)dev;
18                        /*设备结构体指针作为定时器处理函数参数*/
19  dev->xxx_timer.expires = jiffies + delay;
20  /*添加（注册）定时器*/
21  add_timer(&dev->xxx_timer);
22  ...
23 }
24
25 /*xxx 驱动中的某函数*/
26 xxx_func2(…)
27 {
28  ...
29  /*删除定时器*/
30  del_timer (&dev->xxx_timer);
31  ...
32 }
33
34 /*定时器处理函数*/
35 static void xxx_do_timer(unsigned long arg)
```

```
36 {
37  struct xxx_device *dev = (struct xxx_device *)(arg);
38  ...
39  /*调度定时器再执行*/
40  dev->xxx_timer.expires = jiffies + delay;
41  add_timer(&dev->xxx_timer);
42  ...
43 }
```

从代码清单第 19、40 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，在做完相应的工作后，往往会延后 expires 并将定时器再次添加到内核定时器链表，以便定时器能再次被触发。

10.5.2 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，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 10.11。

代码清单 10.11 使用内核定时器的 second 字符设备驱动

```
1   #include ...
2
3   #define SECOND_MAJOR 252    /*预设的 second 的主设备号*/
4
5   static int second_major = SECOND_MAJOR;
6
7   /*second 设备结构体*/
8   struct second_dev
9   {
10    struct cdev cdev; /*cdev 结构体*/
11    atomic_t counter;/* 一共经历了多少秒？ */
12    struct timer_list s_timer; /*设备要使用的定时器*/
13  };
14
15  struct second_dev *second_devp; /*设备结构体指针*/
16
17  /*定时器处理函数*/
18  static void second_timer_handle(unsigned long arg)
19  {
20    mod_timer(&second_devp->s_timer,jiffies + HZ);
21    atomic_inc(&second_devp->counter);
22
```

```
23    printk(KERN_NOTICE "current jiffies is %ld\n", jiffies);
24  }
25
26  /*文件打开函数*/
27  int second_open(struct inode *inode, struct file *filp)
28  {
29    /*初始化定时器*/
30    init_timer(&second_devp->s_timer);
31    second_devp->s_timer.function = &second_timer_handle;
32    second_devp->s_timer.expires = jiffies + HZ;
33
34    add_timer(&second_devp->s_timer); /*添加（注册）定时器*/
35
36    atomic_set(&second_devp->counter,0); //计数清零
37
38    return 0;
39  }
40  /*文件释放函数*/
41  int second_release(struct inode *inode, struct file *filp)
42  {
43    del_timer(&second_devp->s_timer);
44
45    return 0;
46  }
47
48  /*globalfifo 读函数*/
49  static ssize_t second_read(struct file *filp, char __user *buf,
size_t count,
50    loff_t *ppos)
51  {
52    int counter;
53
54    counter = atomic_read(&second_devp->counter);
55    if(put_user(counter, (int*)buf))
56      return - EFAULT;
57    else
58      return sizeof(unsigned int);
59  }
60
61  /*文件操作结构体*/
```

```
62  static const struct file_operations second_fops =
63  {
64    .owner = THIS_MODULE,
65    .open = second_open,
66    .release = second_release,
67    .read = second_read,
68  };
69
70  /*初始化并注册 cdev*/
71  static void second_setup_cdev(struct second_dev *dev, int index)
72  {
73    int err, devno = MKDEV(second_major, index);
74
75    cdev_init(&dev->cdev, &second_fops);
76    dev->cdev.owner = THIS_MODULE;
77    dev->cdev.ops = &second_fops;
78    err = cdev_add(&dev->cdev, devno, 1);
79    if (err)
80      printk(KERN_NOTICE "Error %d adding LED%d", err, index);
81  }
82
83  /*设备驱动模块加载函数*/
84  int second_init(void)
85  {
86    int ret;
87    dev_t devno = MKDEV(second_major, 0);
88
89    /* 申请设备号*/
90    if (second_major)
91      ret = register_chrdev_region(devno, 1, "second");
92    else  /* 动态申请设备号 */
93    {
94      ret = alloc_chrdev_region(&devno, 0, 1, "second");
95      second_major = MAJOR(devno);
96    }
97    if (ret < 0)
98      return ret;
99    /* 动态申请设备结构体的内存*/
100   second_devp = kmalloc(sizeof(struct second_dev), GFP_KERNEL);
101   if (!second_devp)    /*申请失败*/
```

```
    102  {
    103    ret =  - ENOMEM;
    104    goto fail_malloc;
    105  }
    106
    107  memset(second_devp, 0, sizeof(struct second_dev));
    108
    109  second_setup_cdev(second_devp, 0);
    110
    111  return 0;
    112
    113  fail_malloc: unregister_chrdev_region(devno, 1);
    114 }
    115
    116 /*模块卸载函数*/
    117 void second_exit(void)
    118 {
    119  cdev_del(&second_devp->cdev);   /*注销 cdev*/
    120  kfree(second_devp);     /*释放设备结构体内存*/
    121  unregister_chrdev_region(MKDEV(second_major, 0), 1); /*释放设备
号*/
    122 }
    123
    124 MODULE_AUTHOR("Song Baohua");
    125 MODULE_LICENSE("Dual BSD/GPL");
    126
    127 module_param(second_major, int, S_IRUGO);
    128
    129 module_init(second_init);
    130 module_exit(second_exit);
```

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将被 atomic_inc()调用原子的增 1，second 的 read()函数会将这个值返回给用户空间。

编译驱动，加载该内核模块并创建“/dev/second”设备文件结点后，使用代码清单 10.12 的应用程序打开“/dev/second”。

代码清单 10.12 的应用程序 second_test 会不断地读取自打开“/dev/second”设备文件以来经历的秒数。

代码清单 10.12　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)
12   {
13     while (1)
14     {
15       read(fd,&counter, sizeof(unsigned int));//读目前经历的秒数
16       if(counter!=old_counter)
17       {
18           printf("seconds after open /dev/second :%d\n",counter);
19           old_counter = counter;
20       }
21     }
22   }
23   else
24   {
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，如下所示：

```
current jiffies is 17216
current jiffies is 17316
current jiffies is 17416
current jiffies is 17516
current jiffies is 17616
current jiffies is 17716
current jiffies is 17816
current jiffies is 17916
current jiffies is 18016
current jiffies is 18116
current jiffies is 18216
current jiffies is 18316
```

而应用程序将不断输出自打开/dec/second 以来经历的秒数，如下所示：

```
[root@localhost driver_study]# ./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
seconds after open /dev/second :6
seconds after open /dev/second :7
seconds after open /dev/second :8
seconds after open /dev/second :9
```

```
seconds after open /dev/second :10
```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了如下 3 个函数分别进行纳秒、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，可以在软件中进行这样的延迟：

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

ndelay()、udelay()和 mdelay()函数的实现方式机理与此类似。

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，这将无谓地耗费 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，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

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 10.13 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 10.13 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 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，这将保证它每次都被重新读取。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，随着延迟在等待的时间到来之间进程处于睡眠状态，CPU 资源被其他进程使用。schedule_timeout()可以使当前任务睡眠指定的 jiffies 之后重新被调度执行，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()实现的，如代码清单 10.14 所示。

代码清单 10.14 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参数对应的进程。

代码清单 10.14 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 10.15 所示。

代码清单 10.15 schedule_timeout_interruptible()和 schedule_timeout_interruptible()

```
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），如下所示：

```
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。

内核中的延时是忙等待或者睡眠等待，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。

- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm