第七章 Linux 内核的时钟中断

（By 詹荣开，NUDT）

E-mail:zhanrk@sohu.com
Linux-2.4.0
Version 1.0.0，2003-2-14

摘要：本文主要从内核实现的角度分析了 Linux 2.4.0 内核的时钟中断、内核对时间的表示等。本文是为那些想要了解 Linux I/O 子系统的读者和 Linux 驱动程序开发人员而写的。
关键词：Linux、时钟、定时器

欢迎各位指出文档中的错误与疑问。

前言

时间在一个操作系统内核中占据着重要的地位，它是驱动一个 OS 内核运行的“起博器”。一般说来，内核主要需要两种类型的时间：

1. 在内核运行期间持续记录当前的时间与日期，以便内核对某些对象和事件作时间标记（timestamp，也称为“时间戳”），或供用户通过时间 syscall 进行检索。
2. 维持一个固定周期的定时器，以提醒内核或用户一段时间已经过去了。

PC 机中的时间是有三种时钟硬件提供的，而这些时钟硬件又都基于固定频率的晶体振荡器来提供时钟方波信号输入。这三种时钟硬件是：（1）实时时钟（Real Time Clock，RTC）；（2）可编程间隔定时器（Programmable Interval Timer，PIT）；（3）时间戳计数器（Time Stamp Counter，TSC）。

7．1 时钟硬件

7．1．1 实时时钟 RTC

自从 IBM PC AT 起，所有的 PC 机就都包含了一个叫做实时时钟（RTC）的时钟芯片，以便在 PC 机断电后仍然能够继续保持时间。显然，RTC 是通过主板上的电池来供电的，而不是通过 PC 机电源来供电的，因此当 PC 机关掉电源后，RTC 仍然会继续工作。通常，CMOS RAM 和 RTC 被集成到一块芯片上，因此 RTC 也称作“CMOS Timer”。最常见的 RTC 芯片是 MC146818（Motorola）和 DS12887（maxim），DS12887 完全兼容于 MC146818，并有一定的扩展。本节内容主要基于 MC146818 这一标准的 RTC 芯片。具体内容可以参考 MC146818 的 Datasheet。

7．1．1．1 RTC 寄存器

MC146818 RTC 芯片一共有 64 个寄存器。它们的芯片内部地址编号为 0x00～0x3F（不是 I/O 端口地址），这些寄存器一共可以分为三组：

（1）时钟与日历寄存器组：共有 10 个（0x00~0x09），表示时间、日历的具体信息。在 PC 机中，这些寄存器中的值都是以 BCD 格式来存储的（比如 23dec＝0x23BCD）。

（2）状态和控制寄存器组：共有 4 个（0x0A~0x0D），控制 RTC 芯片的工作方式，并表示当前的状态。

（3）CMOS 配置数据：通用的 CMOS RAM，它们与时间无关，因此我们不关心它。

时钟与日历寄存器组的详细解释如下：

Address	Function
00	Current second for RTC
01	Alarm second
02	Current minute
03	Alarm minute
04	Current hour
05	Alarm hour

06	Current day of　week（01＝Sunday）
07	Current date of month
08	Current month
09	Current year（final two digits，eg：93）

状态寄存器 A（地址 0x0A）的格式如下：

7	6	5	4	3	2	1	0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

图 7－1 RTC 状态寄存器 A 的格式

其中：

（1）bit［7］——UIP 标志（Update in Progress），为 1 表示 RTC 正在更新日历寄存器组中的值，此时日历寄存器组是不可访问的（此时访问它们将得到一个无意义的渐变值）。

（2）bit［6：4］——这三位是“除法器控制位”（divider-control bits），用来定义 RTC 的操作频率。各种可能的值如下：

Divider bits			Time-base frequency	Divider Reset	Operation Mode
DV2	DV1	DV0			
0	0	0	4.194304 MHZ	NO	YES
0	0	1	1.048576 MHZ	NO	YES
0	1	0	32.769 KHZ	NO	YES
1	1	0/1	任何	YES	NO

PC 机通常将 Divider bits 设置成“010”。

（3）bit［3：0］——速率选择位（Rate Selection bits），用于周期性或方波信号输出。

RS bits				4.194304 或 1.048578 MHZ		32.768 KHZ	
RS3	RS2	RS1	RS0	周期性中断	方波	周期性中断	方波
0	0	0	0	None	None	None	None
0	0	0	1	30.517μs	32.768 KHZ	3.90625ms	256 HZ
0	0	1	0	61.035μs	16.384 KHZ		
0	0	1	1	122.070μs	8.192KHZ		
0	1	0	0	244.141μs	4.096KHZ		
0	1	0	1	488.281μs	2.048KHZ		
0	1	1	0	976.562μs	1.024KHZ		
0	1	1	1	1.953125ms	512HZ		
1	0	0	0	3.90625ms	256HZ		
1	0	0	1	7.8125ms	128HZ		
1	0	1	0	15.625ms	64HZ		
1	0	1	1	31.25ms	32HZ		
1	1	0	0	62.5ms	16HZ		
1	1	0	1	125ms	8HZ		

1	1	1	0	250ms	4HZ		
1	1	1	1	500ms	2HZ		

PC 机 BIOS 对其默认的设置值是“0110”。

状态寄存器 B 的格式如下所示：

7	6	5	4	3	2	1	0
SET	PIE	AIE	UIE	SQWE	DM	24/12	DSE

图 7－2 RTC 状态寄存器 B 的格式

各位的含义如下：

（1）bit［7］——SET 标志。为 1 表示 RTC 的所有更新过程都将终止，用户程序随后马上对日历寄存器组中的值进行初始化设置。为 0 表示将允许更新过程继续。

（2）bit［6］——PIE 标志，周期性中断使能标志。

（3）bit［5］——AIE 标志，告警中断使能标志。

（4）bit［4］——UIE 标志，更新结束中断使能标志。

（5）bit［3］——SQWE 标志，方波信号使能标志。

（6）bit［2］——DM 标志，用来控制日历寄存器组的数据模式，0＝BCD，1＝BINARY。BIOS 总是将它设置为 0。

（7）bit［1］——24 / 12 标志，用来控制 hour 寄存器，0 表示 12 小时制，1 表示 24 小时制。PC 机 BIOS 总是将它设置为 1。

（8）bit［0］——DSE 标志。BIOS 总是将它设置为 0。

状态寄存器 C 的格式如下：

7	6	5	4	3	2	1	0
IRQF	PF	AF	UF	0	0	0	0

图 7－3 RTC 状态寄存器 C 的格式

（1）bit［7］——IRQF 标志，中断请求标志，当该位为 1 时，说明寄存器 B 中断请求发生。

（2）bit［6］——PF 标志，周期性中断标志，为 1 表示发生周期性中断请求。

（3）bit［5］——AF 标志，告警中断标志，为 1 表示发生告警中断请求。

（4）bit［4］——UF 标志，更新结束中断标志，为 1 表示发生更新结束中断请求。

状态寄存器 D 的格式如下：

7	6	5	4	3	2	1	0
VRT	0	0	0	0	0	0	0

图 7－4 RTC 状态寄存器 D 的格式

（1）bit［7］——VRT 标志（Valid RAM and Time），为 1 表示 OK，为 0 表示 RTC 已经掉电。

（2）bit［6：0］——总是为 0，未定义。

7．1．1．2 通过 I/O 端口访问 RTC

在 PC 机中可以通过 I/O 端口 0x70 和 0x71 来读写 RTC 芯片中的寄存器。其中，端口 0x70 是 RTC 的寄存器地址索引端口，0x71 是数据端口。

读 RTC 芯片寄存器的步骤是：

```
mov    al, addr
out     70h, al          ; Select reg_addr in RTC chip
jmp $+2                  ; a slight delay to settle thing
in    al, 71h            ;
```

写 RTC 寄存器的步骤如下：

```
mov    al, addr
out    70h, al           ; Select reg_addr in RTC chip
jmp    $+2               ; a slight delay to settle thing
mov    al, value
out    71h, al
```

7．1．2 可编程间隔定时器 PIT

每个 PC 机中都有一个 PIT，以通过 IRQ0 产生周期性的时钟中断信号。当前使用最普遍的是 Intel 8254 PIT 芯片，它的 I/O 端口地址是 0x40~0x43。

Intel 8254 PIT 有 3 个计时通道，每个通道都有其不同的用途：

（1） 通道 0 用来负责更新系统时钟。每当一个时钟滴答过去时，它就会通过 IRQ0 向系统产生一次时钟中断。

（2） 通道 1 通常用于控制 DMAC 对 RAM 的刷新。

（3） 通道 2 被连接到 PC 机的扬声器，以产生方波信号。

每个通道都有一个向下减小的计数器，8254 PIT 的输入时钟信号的频率是 1193181HZ，也即一秒钟输入 1193181 个 clock-cycle。每输入一个 clock-cycle 其时间通道的计数器就向下减 1，一直减到 0 值。因此对于通道 0 而言，当他的计数器减到 0 时，PIT 就向系统产生一次时钟中断，表示一个时钟滴答已经过去了。当各通道的计数器减到 0 时，我们就说该通道处于“Terminal count”状态。

通道计数器的最大值是 10000h，所对应的时钟中断频率是 1193181／（65536）＝18.2HZ，也就是说，此时一秒钟之内将产生 18.2 次时钟中断。

7．1．2．1 PIT 的 I/O 端口

在 i386 平台上，8254 芯片的各寄存器的 I/O 端口地址如下：

Port	Description
40h	Channel 0 counter（read/write）
41h	Channel 1 counter（read/write）
42h	Channel 2 counter（read/write）
43h	PIT control word（write only）

其中，由于通道 0、1、2 的计数器是一个 16 位寄存器，而相应的端口却都是 8 位的，因此读写通道计数器必须进行进行两次 I/O 端口读写操作，分别对应于计数器的高字节和低字节，至于是先读写高字节再读写低字节，还是先读写低字节再读写高字节，则由 PIT 的控制寄存器来决定。8254 PIT 的控制寄存器的格式如下：

7	6	5	4	3	2	1	0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

图 7－5 8254 PIT 的控制寄存器的格式

（1）bit［7：6］——Select Counter，选择对那个计数器进行操作。“00”表示选择 Counter 0，“01”表示选择 Counter 1，“10”表示选择 Counter 2，“11”表示 Read-Back Command（仅对于 8254，对于 8253 无效）。

（2）bit［5：4］——Read/Write/Latch 格式位。“00”表示锁存（Latch）当前计数器的值；“01”只读写计数器的高字节（MSB）；“10”只读写计数器的低字节（LSB）；“11”表示先读写计数器的 LSB，再读写 MSB。

（3）bit［3：1］——Mode bits，控制各通道的工作模式。“000”对应 Mode 0；“001”对应 Mode 1；“010”对应 Mode 2；“011”对应 Mode 3；“100”对应 Mode 4；“101”对应 Mode 5。

（4）bit［0］——控制计数器的存储模式。0 表示以二进制格式存储，1 表示计数器中的值以 BCD 格式存储。

7．1．2．2 PIT 通道的工作模式

PIT 各通道可以工作在下列 6 种模式下：

1. Mode 0：当通道处于“Terminal count”状态时产生中断信号。
2. Mode 1：Hardware retriggerable one-shot。
3. Mode 2：Rate Generator。这种模式典型地被用来产生实时时钟中断。此时通道的信号输出管脚 OUT 初始时被设置为高电平，并以此持续到计数器的值减到 1。然后在接下来的这个 clock-cycle 期间，OUT 管脚将变为低电平，直到计数器的值减到 0。当计数器的值被自动地重新加载后，OUT 管脚又变成高电平，然后重复上述过程。通道 0 通常工作在这个模式下。
4. Mode 3：方波信号发生器。
5. Mode 4：Software triggered strobe。
6. Mode 5：Hardware triggered strobe。

7．1．2．3 锁存计数器（Latch Counter）

当控制寄存器中的 bit［5：4］设置成 0 时，将把当前通道的计数器值锁存。此时通过 I/O 端口可以读到一个稳定的计数器值，因为计数器表面上已经停止向下计数（PIT 芯片内部并没有停止向下计数）。NOTE！一旦发出了锁存命令，就要马上读计数器的值。

7．1．3 时间戳记数器 TSC

从 Pentium 开始，所有的 Intel 80x86 CPU 就都又包含一个 64 位的时间戳记数器（TSC）的寄存器。该寄存器实际上是一个不断增加的计数器，它在 CPU 的每个时钟信号到来时加 1（也即每一个 clock-cycle 输入 CPU 时，该计数器的值就加 1）。

汇编指令 rdtsc 可以用于读取 TSC 的值。利用 CPU 的 TSC，操作系统通常可以得到更为精准的时间度量。假如 clock-cycle 的频率是 400MHZ，那么 TSC 就将每 2.5 纳秒增加一次。

7．2 Linux 内核对 RTC 的编程

MC146818 RTC 芯片（或其他兼容芯片，如 DS12887）可以在 IRQ8 上产生周期性的中断，中断的频率在 2HZ～8192HZ 之间。与 MC146818 RTC 对应的设备驱动程序实现在 include/linux/rtc.h 和 drivers / char/rtc.c 文件中，对应的设备文件是 / dev/rtc（major=10,minor=135，只读字符设备）。因此用户进程可以通过对她进行编程以使得当 RTC 到达某个特定的时间值时激活 IRQ8 线，从而将 RTC 当作一个闹钟来用。

而 Linux 内核对 RTC 的唯一用途就是把 RTC 用作“离线”或“后台”的时间与日期维护器。当 Linux 内核启动时，它从 RTC 中读取时间与日期的基准值。然后再运行期间内核就完全抛开 RTC，从而以软件的形式维护系统的当前时间与日期，并在需要时将时间回写到 RTC 芯片中。

Linux 在 include/linux/mc146818rtc.h 和 include/asm-i386/mc146818rtc.h 头文件中分别定义了 mc146818 RTC 芯片各寄存器的含义以及 RTC 芯片在 i386 平台上的 I/O 端口操作。而通用的 RTC 接口则声明在 include/linux/rtc.h 头文件中。

7．2．1 RTC 芯片的 I/O 端口操作

Linux 在 include/asm-i386/mc146818rtc.h 头文件中定义了 RTC 芯片的 I/O 端口操作。端口 0x70 被称为“RTC 端口 0”，端口 0x71 被称为“RTC 端口 1”，如下所示：

```
#ifndef RTC_PORT
#define RTC_PORT(x)  (0x70 + (x))
#define RTC_ALWAYS_BCD      1     /* RTC operates in binary mode */
#endif
```

显然，RTC_PORT(0)就是指端口 0x70，RTC_PORT(1)就是指 I/O 端口 0x71。

端口 0x70 被用作 RTC 芯片内部寄存器的地址索引端口，而端口 0x71 则被用作 RTC 芯片内部寄存器的数据端口。再读写一个 RTC 寄存器之前，必须先把该寄存器在 RTC 芯片内部的地址索引值写到端口 0x70 中。根据这一点，读写一个 RTC 寄存器的宏定义 CMOS_READ()和 CMOS_WRITE()如下：

```
#define CMOS_READ(addr) ({ \
outb_p((addr),RTC_PORT(0)); \
inb_p(RTC_PORT(1)); \
})
#define CMOS_WRITE(val, addr) ({ \
outb_p((addr),RTC_PORT(0)); \
outb_p((val),RTC_PORT(1)); \
})
```

#define RTC_IRQ 8

在上述宏定义中，参数 addr 是 RTC 寄存器在芯片内部的地址值，取值范围是 0x00~0x3F，参数 val 是待写入寄存器的值。宏 RTC_IRQ 是指 RTC 芯片所连接的中断请求输入线号，通常是 8。

7．2．2 对 RTC 寄存器的定义

Linux 在 include/linux/mc146818rtc.h 这个头文件中定义了 RTC 各寄存器的含义。

（1）寄存器内部地址索引的定义

Linux 内核仅使用 RTC 芯片的时间与日期寄存器组和控制寄存器组，地址为 0x00~0x09 之间的 10 个时间与日期寄存器的定义如下：

```
#define RTC_SECONDS             0
#define RTC_SECONDS_ALARM       1
#define RTC_MINUTES             2
#define RTC_MINUTES_ALARM       3
#define RTC_HOURS           4
#define RTC_HOURS_ALARM         5
/* RTC_*_alarm is always true if 2 MSBs are set */
# define RTC_ALARM_DONT_CARE        0xC0

#define RTC_DAY_OF_WEEK         6
#define RTC_DAY_OF_MONTH 7
#define RTC_MONTH           8
#define RTC_YEAR            9
```

四个控制寄存器的地址定义如下：

```
#define RTC_REG_A           10
#define RTC_REG_B           11
#define RTC_REG_C           12
#define RTC_REG_D           13
```

（2）各控制寄存器的状态位的详细定义

控制寄存器 A（0x0A）主要用于选择 RTC 芯片的工作频率，因此也称为 RTC 频率选择寄存器。因此 Linux 用一个宏别名 RTC_FREQ_SELECT 来表示控制寄存器 A，如下：

#define RTC_FREQ_SELECT   RTC_REG_A

RTC 频率寄存器中的位被分为三组：①bit［7］表示 UIP 标志；②bit［6：4］用于除法器的频率选择；③bit［3：0］用于速率选择。它们的定义如下：

```
# define RTC_UIP        0x80
# define RTC_DIV_CTL            0x70
   /* Periodic intr. / Square wave rate select. 0=none, 1=32.8kHz,... 15=2Hz */
# define RTC_RATE_SELECT     0x0F
```

正如 7.1.1.1 节所介绍的那样，bit［6：4］有 5 中可能的取值，分别为除法器选择不同的工作频率或用于重置除法器，各种可能的取值如下定义所示：

```
    /* divider control: refclock values 4.194 / 1.049 MHz / 32.768 kHz */
```

```
#   define RTC_REF_CLCK_4MHZ   0x00
#   define RTC_REF_CLCK_1MHZ   0x10
#   define RTC_REF_CLCK_32KHZ  0x20
    /* 2 values for divider stage reset, others for "testing purposes only" */
#   define RTC_DIV_RESET1     0x60
#   define RTC_DIV_RESET2     0x70
```

寄存器 B 中的各位用于使能 / 禁止 RTC 的各种特性，因此控制寄存器 B（0x0B）也称为“控制寄存器”，Linux 用宏别名 RTC_CONTROL 来表示控制寄存器 B，它与其中的各标志位的定义如下所示：

```
#define RTC_CONTROL     RTC_REG_B
# define RTC_SET 0x80       /* disable updates for clock setting */
# define RTC_PIE 0x40       /* periodic interrupt enable */
# define RTC_AIE 0x20       /* alarm interrupt enable */
# define RTC_UIE 0x10       /* update-finished interrupt enable */
# define RTC_SQWE 0x08          /* enable square-wave output */
# define RTC_DM_BINARY 0x04/* all time/date values are BCD if clear */
# define RTC_24H 0x02       /* 24 hour mode - else hours bit 7 means pm */
# define RTC_DST_EN 0x01/* auto switch DST - works f. USA only */
```

寄存器 C 是 RTC 芯片的中断请求状态寄存器，Linux 用宏别名 RTC_INTR_FLAGS 来表示寄存器 C，它与其中的各标志位的定义如下所示：

```
#define RTC_INTR_FLAGS RTC_REG_C
/* caution - cleared by read */
# define RTC_IRQF 0x80          /* any of the following 3 is active */
# define RTC_PF 0x40
# define RTC_AF 0x20
# define RTC_UF 0x10
```

寄存器 D 仅定义了其最高位 bit［7］，以表示 RTC 芯片是否有效。因此寄存器 D 也称为 RTC 的有效寄存器。Linux 用宏别名 RTC_VALID 来表示寄存器 D，如下：

```
#define RTC_VALID    RTC_REG_D
# define RTC_VRT 0x80           /* valid RAM and time */
```

（3）二进制格式与 BCD 格式的相互转换

由于时间与日期寄存器中的值可能以 BCD 格式存储，也可能以二进制格式存储，因此需要定义二进制格式与 BCD 格式之间的相互转换宏，以方便编程。如下：

```
#ifndef BCD_TO_BIN
#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
#endif

#ifndef BIN_TO_BCD
#define BIN_TO_BCD(val) ((val)=(((val)/10)<<4) + (val)%10)
#endif
```

7．2．3 内核对 RTC 的操作

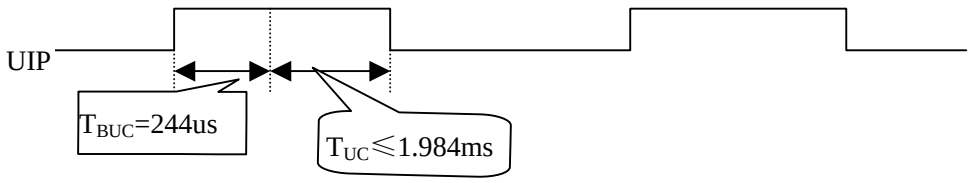
如前所述，Linux 内核与 RTC 进行互操作的时机只有两个：（1）内核在启动时从 RTC 中读取启动时的时间与日期;（2）内核在需要时将时间与日期回写到 RTC 中。为此，Linux 内核在 arch/i386/kernel/time.c 文件中实现了函数 get_cmos_time()来进行对 RTC 的第一种操作。显然，get_cmos_time()函数仅仅在内核启动时被调用一次。而对于第二种操作，Linux 则同样在 arch/i386/kernel/time.c 文件中实现了函数 set_rtc_mmss()，以支持向 RTC 中回写当前时间与日期。下面我们将来分析这二个函数的实现。

在分析 get_cmos_time()函数之前，我们先来看看 RTC 芯片对其时间与日期寄存器组的更新原理。

（1）Update In Progress

当控制寄存器 B 中的 SET 标志位为 0 时，MC146818 芯片每秒都会在芯片内部执行一个“更新周期”（Update Cycle），其作用是增加秒寄存器的值，并检查秒寄存器是否溢出。如果溢出，则增加分钟寄存器的值，如此一致下去直到年寄存器。在“更新周期”期间，时间与日期寄存器组（0x00~0x09）是不可用的，此时如果读取它们的值将得到未定义的值，因为 MC146818 在整个更新周期期间会把时间与日期寄存器组从 CPU 总线上脱离，从而防止软件程序读到一个渐变的数据。

在 MC146818 的输入时钟频率（也即晶体增荡器的频率）为 4.194304MHZ 或 1.048576MHZ 的情况下，“更新周期”需要花费 248us，而对于输入时钟频率为 32.768KHZ 的情况，“更新周期”需要花费 1984us＝1.984ms。控制寄存器 A 中的 UIP 标志位用来表示 MC146818 是否正处于更新周期中，当 UIP 从 0 变为 1 的那个时刻，就表示 MC146818 将在稍后马上就开更新周期。在 UIP 从 0 变到 1 的那个时刻与 MC146818 真正开始 Update Cycle 的那个时刻之间时有一段时间间隔的，通常是 244us。也就是说，在 UIP 从 0 变到 1 的 244us 之后，时间与日期寄存器组中的值才会真正开始改变，而在这之间的 244us 间隔内，它们的值并不会真正改变。如下图所示：



T_{BUC}：Delay Time Before Update Cycle（244us）

T_{UC}：Update Cycle Time（248us 或 1.984ms）

图 7－6 RTC 更新周期中的时间关系

（2）get_cmos_time()函数

该函数只被内核的初始化例程 time_init()和内核的 APM 模块所调用。其源码如下：

```
/* not static: needed by APM */
unsigned long get_cmos_time(void)
{
    unsigned int year, mon, day, hour, min, sec;
    int i;

    /* The Linux interpretation of the CMOS clock register contents:
     * When the Update-In-Progress (UIP) flag goes from 1 to 0, the
     * RTC registers show the second which has precisely just started.
     * Let's hope other operating systems interpret the RTC the same way.
     */
```

```
        /* read RTC exactly on falling edge of update flag */
        for (i = 0 ; i < 1000000 ; i++) /* may take up to 1 second... */
            if (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP)
                break;
        for (i = 0 ; i < 1000000 ; i++) /* must try at least 2.228 ms */
            if (!(CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP))
                break;
        do { /* Isn't this overkill ? UIP above should guarantee consistency */
            sec = CMOS_READ(RTC_SECONDS);
            min = CMOS_READ(RTC_MINUTES);
            hour = CMOS_READ(RTC_HOURS);
            day = CMOS_READ(RTC_DAY_OF_MONTH);
            mon = CMOS_READ(RTC_MONTH);
            year = CMOS_READ(RTC_YEAR);
        } while (sec != CMOS_READ(RTC_SECONDS));
        if (!(CMOS_READ(RTC_CONTROL) & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
          {
            BCD_TO_BIN(sec);
            BCD_TO_BIN(min);
            BCD_TO_BIN(hour);
            BCD_TO_BIN(day);
            BCD_TO_BIN(mon);
            BCD_TO_BIN(year);
          }
        if ((year += 1900) < 1970)
            year += 100;
        return mktime(year, mon, day, hour, min, sec);
    }
```

对该函数的注释如下：

（1）在从 RTC 中读取时间时，由于 RTC 存在 Update Cycle，因此软件发出读操作的时机是很重要的。对此，get_cmos_time()函数通过 UIP 标志位来解决这个问题：第一个 for 循环不停地读取 RTC 频率选择寄存器中的 UIP 标志位，并且只要读到 UIP 的值为 1 就马上退出这个 for 循环。第二个 for 循环同样不停地读取 UIP 标志位，但他只要一读到 UIP 的值为 0 就马上退出这个 for 循环。这两个 for 循环的目的就是要在软件逻辑上同步 RTC 的 Update Cycle，显然第二个 for 循环最大可能需要 2.228ms(T_{BUC}+max(T_{UC})=244us+1984us=2.228ms)

(2)从第二个 for 循环退出后，RTC 的 Update Cycle 已经结束。此时我们就已经把当前时间逻辑定准在 RTC 的当前一秒时间间隔内。也就是说，这是我们就可以开始从 RTC 寄存器中读取当前时间值。但是要注意，读操作应该保证在 244us 内完成（准确地说，读操作要在 RTC 的下一个更新周期开始之前完成，244us 的限制是过分偏执的：－）。所以，get_cmos_time()函数接下来通过 CMOS_READ()宏从 RTC 中依次读取秒、分钟、小时、日期、月份和年分。这里的 do{}while(sec!=CMOS_READ(RTC_SECOND))循环就是用来确保上述 6 个读操作必须在下一个 Update Cycle 开始之前完成。

（3）接下来判定时间的数据格式，PC 机中一般总是使用 BCD 格式的时间，因此需要通过 BCD_TO_BIN()宏把 BCD 格式转换为二进制格式。

（4）接下来对年分进行修正，以将年份转换为“19XX”的格式，如果是 1970 以前的年份，则将其加上 100。

（5）最后调用 mktime()函数将当前时间与日期转换为相对于 1970－01－01 00：00：00 的秒数值，并

将其作为函数返回值返回。

函数 mktime() 定义在 include/linux/time.h 头文件中，它用来根据 Gauss 算法将以 year/mon/day/hour/min/sec（如 1980－12－31 23：59：59）格式表示的时间转换为相对于 1970－01－01 00：00：00 这个 UNIX 时间基准以来的相对秒数。其源码如下：

```
static inline unsigned long
mktime (unsigned int year, unsigned int mon,
    unsigned int day, unsigned int hour,
    unsigned int min, unsigned int sec)
{
    if (0 >= (int) (mon -= 2)) {     /* 1..12 -> 11,12,1..10 */
        mon += 12;          /* Puts Feb last since it has leap day */
        year -= 1;
    }

    return (((
        (unsigned long) (year/4 - year/100 + year/400 + 367*mon/12 + day) +
            year*365 - 719499
        )*24 + hour /* now have hours */
      )*60 + min /* now have minutes */
    )*60 + sec; /* finally seconds */
}
```

（3）set_rtc_mmss()函数

该函数用来更新 RTC 中的时间，它仅有一个参数 nowtime，是以秒数表示的当前时间，其源码如下：

```
static int set_rtc_mmss(unsigned long nowtime)
{
    int retval = 0;
    int real_seconds, real_minutes, cmos_minutes;
    unsigned char save_control, save_freq_select;

    /* gets recalled with irq locally disabled */
    spin_lock(&rtc_lock);
    save_control = CMOS_READ(RTC_CONTROL); /* tell the clock it's being set */
    CMOS_WRITE((save_control|RTC_SET), RTC_CONTROL);

    save_freq_select = CMOS_READ(RTC_FREQ_SELECT); /* stop and reset prescaler */
    CMOS_WRITE((save_freq_select|RTC_DIV_RESET2), RTC_FREQ_SELECT);

    cmos_minutes = CMOS_READ(RTC_MINUTES);
    if (!(save_control & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
        BCD_TO_BIN(cmos_minutes);

    /*
     * since we're only adjusting minutes and seconds,
     * don't interfere with hour overflow. This avoids
```

```
            * messing with unknown time zones but requires your
            * RTC not to be off by more than 15 minutes
            */
           real_seconds = nowtime % 60;
           real_minutes = nowtime / 60;
           if (((abs(real_minutes - cmos_minutes) + 15)/30) & 1)
                real_minutes += 30;             /* correct for half hour time zone */
           real_minutes %= 60;

           if (abs(real_minutes - cmos_minutes) < 30) {
                if (!(save_control & RTC_DM_BINARY) || RTC_ALWAYS_BCD) {
                     BIN_TO_BCD(real_seconds);
                     BIN_TO_BCD(real_minutes);
                }
                CMOS_WRITE(real_seconds,RTC_SECONDS);
                CMOS_WRITE(real_minutes,RTC_MINUTES);
           } else {
                printk(KERN_WARNING
                       "set_rtc_mmss: can't update from %d to %d\n",
                       cmos_minutes, real_minutes);
                retval = -1;
           }

           /* The following flags have to be released exactly in this order,
            * otherwise the DS12887 (popular MC146818A clone with integrated
            * battery and quartz) will not reset the oscillator and will not
            * update precisely 500 ms later. You won't find this mentioned in
            * the Dallas Semiconductor data sheets, but who believes data
            * sheets anyway ...                           -- Markus Kuhn
            */
           CMOS_WRITE(save_control, RTC_CONTROL);
           CMOS_WRITE(save_freq_select, RTC_FREQ_SELECT);
           spin_unlock(&rtc_lock);

           return retval;
      }
```

对该函数的注释如下：

（1）首先对自旋锁 rtc_lock 进行加锁。定义在 arch/i386/kernel/time.c 文件中的全局自旋锁 rtc_lock 用来串行化所有 CPU 对 RTC 的操作。

（2）接下来，在 RTC 控制寄存器中设置 SET 标志位，以便通知 RTC 软件程序随后马上将要更新它的时间与日期。为此先把 RTC_CONTROL 寄存器的当前值读到变量 save_control 中，然后再把值（save_control | RTC_SET）回写到寄存器 RTC_CONTROL 中。

（3）然后，通过 RTC_FREQ_SELECT 寄存器中 bit［6：4］重启 RTC 芯片内部的除法器。为此，类似地先把 RTC_FREQ_SELECT 寄存器的当前值读到变量 save_freq_select 中，然后再把值（save_freq_select ｜ RTC_DIV_RESET2）回写到 RTC_FREQ_SELECT 寄存器中。

（4）接着将 RTC_MINUTES 寄存器的当前值读到变量 cmos_minutes 中，并根据需要将它从 BCD 格

式转化为二进制格式。

（5）从 nowtime 参数中得到当前时间的秒数和分钟数。分别保存到 real_seconds 和 real_minutes 变量。注意，这里对于半小时区的情况要修正分钟数 real_minutes 的值。

（6）然后，在 real_minutes 与 RTC_MINUTES 寄存器的原值 cmos_minutes 二者相差不超过 30 分钟的情况下，将 real_seconds 和 real_minutes 所表示的时间值写到 RTC 的秒寄存器和分钟寄存器中。当然，在回写之前要记得把二进制转换为 BCD 格式。

（7）最后，恢复 RTC_CONTROL 寄存器和 RTC_FREQ_SELECT 寄存器原来的值。这二者的先后次序是：先恢复 RTC_CONTROL 寄存器，再恢复 RTC_FREQ_SELECT 寄存器。然后在解除自旋锁 rtc_lock 后就可以返回了。

最后，需要说明的一点是，set_rtc_mmss()函数尽可能在靠近一秒时间间隔的中间位置（也即 500ms 处）左右被调用。此外，Linux 内核对每一次成功的更新 RTC 时间都留下时间轨迹，它用一个系统全局变量 last_rtc_update 来表示内核最近一次成功地对 RTC 进行更新的时间（单位是秒数）。该变量定义在 arch/i386/kernel/time.c 文件中：

```
/* last time the cmos clock got updated */
static long last_rtc_update;
```

每一次成功地调用 set_rtc_mmss()函数后，内核都会马上将 last_rtc_update 更新为当前时间（具体请见 7.4.3 节）。

7．3 Linux 对时间的表示

通常，操作系统可以使用三种方法来表示系统的当前时间与日期：①最简单的一种方法就是直接用一个 64 位的计数器来对时钟滴答进行计数。②第二种方法就是用一个 32 位计数器来对秒进行计数，同时还用一个 32 位的辅助计数器对时钟滴答计数，之子累积到一秒为止。因为 2^{32} 超过 136 年，因此这种方法直至 22 世纪都可以让系统工作得很好。③第三种方法也是按时钟滴答进行计数，但是是相对于系统启动以来的滴答次数，而不是相对于相对于某个确定的外部时刻；当读外部后备时钟（如 RTC）或用户输入实际时间时，根据当前的滴答次数计算系统当前时间。

UNIX 类操作系统通常都采用第三种方法来维护系统的时间与日期。

7．3．1 基本概念

首先，有必要明确一些 Linux 内核时钟驱动中的基本概念。

（1）时钟周期（clock cycle）的频率：8253 / 8254 PIT 的本质就是对由晶体振荡器产生的时钟周期进行计数，晶体振荡器在 1 秒时间内产生的时钟脉冲个数就是时钟周期的频率。Linux 用宏 CLOCK_TICK_RATE 来表示 8254 PIT 的输入时钟脉冲的频率（在 PC 机中这个值通常是 1193180HZ），该宏定义在 include/asm-i386/timex.h 头文件中：

```
#define CLOCK_TICK_RATE    1193180 /* Underlying HZ */
```

（2）时钟滴答（clock tick）：我们知道，当 PIT 通道 0 的计数器减到 0 值时，它就在 IRQ0 上产生一次时钟中断，也即一次时钟滴答。PIT 通道 0 的计数器的初始值决定了要过多少时钟周期才产生一次时钟中断，因此也就决定了一次时钟滴答的时间间隔长度。

（3）时钟滴答的频率（HZ）：也即 1 秒时间内 PIT 所产生的时钟滴答次数。类似地，这个值也是由 PIT 通道 0 的计数器初值决定的（反过来说，确定了时钟滴答的频率值后也就可以确定 8254 PIT 通道 0 的计数器初值）。Linux 内核用宏 HZ 来表示时钟滴答的频率，而且在不同的平台上 HZ 有不同的定义值。对

于 ALPHA 和 IA62 平台 HZ 的值是 1024，对于 SPARC、MIPS、ARM 和 i386 等平台 HZ 的值都是 100。该宏在 i386 平台上的定义如下（include/asm-i386/param.h）：

```
#ifndef HZ
#define HZ 100
#endif
```

根据 HZ 的值，我们也可以知道一次时钟滴答的具体时间间隔应该是（1000ms / HZ）＝10ms。

（4）时钟滴答的时间间隔：Linux 用全局变量 tick 来表示时钟滴答的时间间隔长度，该变量定义在 kernel/timer.c 文件中，如下：

```
long tick = (1000000 + HZ/2) / HZ;      /* timer interrupt period */
```

tick 变量的单位是微妙（μs），由于在不同平台上宏 HZ 的值会有所不同，因此方程式 tick＝1000000÷HZ 的结果可能会是个小数，因此将其进行四舍五入成一个整数，所以 Linux 将 tick 定义成（1000000＋HZ / 2）/ HZ，其中被除数表达式中的 HZ / 2 的作用就是用来将 tick 值向上圆整成一个整型数。

另外，Linux 还用宏 TICK_SIZE 来作为 tick 变量的引用别名（alias），其定义如下（arch / i386/kernel/time.c）：

```
#define TICK_SIZE tick
```

（5）宏 LATCH：Linux 用宏 LATCH 来定义要写到 PIT 通道 0 的计数器中的值，它表示 PIT 将没隔多少个时钟周期产生一次时钟中断。显然 LATCH 应该由下列公式计算：

LATCH＝（1 秒之内的时钟周期个数）÷（1 秒之内的时钟中断次数）＝（CLOCK_TICK_RATE）÷（HZ）

类似地，上述公式的结果可能会是个小数，应该对其进行四舍五入。所以，Linux 将 LATCH 定义为（include/linux/timex.h）：

```
/* LATCH is used in the interval timer and ftape setup. */
#define LATCH  ((CLOCK_TICK_RATE + HZ/2) / HZ)  /* For divider */
```

类似地，被除数表达式中的 HZ / 2 也是用来将 LATCH 向上圆整成一个整数。

7．3．2 表示系统当前时间的内核数据结构

作为一种 UNIX 类操作系统，Linux 内核显然采用本节一开始所述的第三种方法来表示系统的当前时间。Linux 内核在表示系统当前时间时用到了三个重要的数据结构：

①全局变量 jiffies：这是一个 32 位的无符号整数，用来表示自内核上一次启动以来的时钟滴答次数。每发生一次时钟滴答，内核的时钟中断处理函数 timer_interrupt（）都要将该全局变量 jiffies 加 1。该变量定义在 kernel/timer.c 源文件中，如下所示：

```
unsigned long volatile jiffies;
```

C 语言限定符 volatile 表示 jiffies 是一个易该变的变量，因此编译器将使对该变量的访问从不通过 CPU 内部 cache 来进行。

②全局变量 xtime：它是一个 timeval 结构类型的变量，用来表示当前时间距 UNIX 时间基准 1970－01－01 00：00：00 的相对秒数值。结构 timeval 是 Linux 内核表示时间的一种格式（Linux 内核对时间的表示有多种格式，每种格式都有不同的时间精度），其时间精度是微秒。该结构是内核表示时间时最常用的一种格式，它定义在头文件 include/linux/time.h 中，如下所示：

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

其中，成员 tv_sec 表示当前时间距 UNIX 时间基准的秒数值，而成员 tv_usec 则表示一秒之内的微秒值，且 1000000>tv_usec>＝0。

Linux 内核通过 timeval 结构类型的全局变量 xtime 来维持当前时间，该变量定义在 kernel/timer.c 文件中，如下所示：

```
/* The current time */
volatile struct timeval xtime __attribute__ ((aligned (16)));
```

但是，全局变量 xtime 所维持的当前时间通常是供用户来检索和设置的，而其他内核模块通常很少使用它（其他内核模块用得最多的是 jiffies），因此对 xtime 的更新并不是一项紧迫的任务，所以这一工作通常被延迟到时钟中断的底半部分（bottom half）中来进行。由于 bottom half 的执行时间带有不确定性，因此为了记住内核上一次更新 xtime 是什么时候，Linux 内核定义了一个类似于 jiffies 的全局变量 wall_jiffies，来保存内核上一次更新 xtime 时的 jiffies 值。时钟中断的底半部分每一次更新 xtime 的时侯都会将 wall_jiffies 更新为当时的 jiffies 值。全局变量 wall_jiffies 定义在 kernel/timer.c 文件中：

```
/* jiffies at the most recent update of wall time */
unsigned long wall_jiffies;
```

③全局变量 sys_tz：它是一个 timezone 结构类型的全局变量，表示系统当前的时区信息。结构类型 timezone 定义在 include/linux/time.h 头文件中，如下所示：

```
struct timezone {
    int    tz_minuteswest;     /* minutes west of Greenwich */
    int    tz_dsttime;     /* type of dst correction */
};
```

基于上述结构，Linux 在 kernel/time.c 文件中定义了全局变量 sys_tz 表示系统当前所处的时区信息，如下所示：

```
struct timezone sys_tz;
```

7．3．3 Linux 对 TSC 的编程实现

Linux 用定义在 arch/i386/kernel/time.c 文件中的全局变量 use_tsc 来表示内核是否使用 CPU 的 TSC 寄存器，use_tsc＝1 表示使用 TSC，use_tsc＝0 表示不使用 TSC。该变量的值是在 time_init()初始化函数中被初始化的（详见下一节）。该变量的定义如下：

```
static int use_tsc;
```

宏 cpu_has_tsc 可以确定当前系统的 CPU 是否配置有 TSC 寄存器。此外，宏 CONFIG_X86_TSC 也表示是否存在 TSC 寄存器。

7．3．3．1　读 TSC 寄存器的宏操作

x86 CPU 的 rdtsc 指令将 TSC 寄存器的高 32 位值读到 EDX 寄存器中、低 32 位读到 EAX 寄存器中。Linux 根据不同的需要，在 rdtsc 指令的基础上封装几个高层宏操作，以读取 TSC 寄存器的值。它们均定义在 include/asm-i386/msr.h 头文件中，如下：

```
#define rdtsc(low,high) \
     __asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))

#define rdtscl(low) \
     __asm__ __volatile__ ("rdtsc" : "=a" (low) : : "edx")

#define rdtscll(val) \
     __asm__ __volatile__ ("rdtsc" : "=A" (val))
```

宏 rdtsc（）同时读取 TSC 的 LSB 与 MSB，并分别保存到宏参数 low 和 high 中。宏 rdtscl 则只读取 TSC 寄存器的 LSB，并保存到宏参数 low 中。宏 rdtscll 读取 TSC 的当前 64 位值，并将其保存到宏参数 val 这个 64 位变量中。

7．3．3．2 校准 TSC

与可编程定时器 PIT 相比，用 TSC 寄存器可以获得更精确的时间度量。但是在可以使用 TSC 之前，它必须精确地确定 1 个 TSC 计数值到底代表多长的时间间隔，也即到底要过多长时间间隔 TSC 寄存器才会加 1。Linux 内核用全局变量 fast_gettimeoffset_quotient 来表示这个值，其定义如下（arch/i386/kernel/time.c）：

```
/* Cached *multiplier* to convert TSC counts to microseconds.
 * (see the equation below).
 * Equal to 2^32 * (1 / (clocks per usec) ).
 * Initialized in time_init.
 */
unsigned long fast_gettimeoffset_quotient;
```

根据上述定义的注释我们可以看出，这个变量的值是通过下述公式来计算的：

fast_gettimeoffset_quotient ＝ (2^32) / (每微秒内的时钟周期个数)

定义在 arch/i386/kernel/time.c 文件中的函数 calibrate_tsc() 就是根据上述公式来计算 fast_gettimeoffset_quotient 的值的。显然这个计算过程必须在内核启动时完成，因此，函数 calibrate_tsc() 只被初始化函数 time_init()所调用。

用 TSC 实现高精度的时间服务

在拥有 TSC（TimeStamp Counter）的 x86 CPU 上，Linux 内核可以实现微秒级的高精度定时服务，也即可以确定两次时钟中断之间的某个时刻的微秒级时间值。如下图所示：

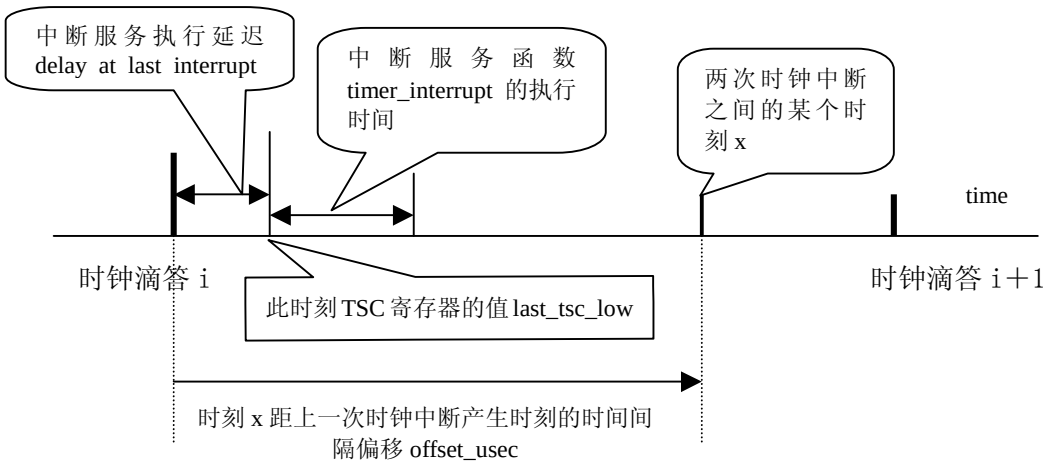


图 7－7 TSC 时间关系

从上图中可以看出，要确定时刻 x 的微秒级时间值，就必须确定时刻 x 距上一次时钟中断产生时刻的时间间隔偏移 offset_usec 的值（以微秒为单位）。为此，内核定义了以下两个变量：

（1）**中断服务执行延迟 delay_at_last_interrupt**：由于从产生时钟中断的那个时刻到内核时钟中断服务函数 timer_interrupt 真正在 CPU 上执行的那个时刻之间是有一段延迟间隔的，因此，Linux 内核用变量 delay_at_last_interrupt 来表示这一段时间延迟间隔，其定义如下（arch/i386/kernel/time.c）：

```
/* Number of usecs that the last interrupt was delayed */
static int delay_at_last_interrupt;
```

关于 delay_at_last_interrupt 的计算步骤我们将在分析 timer_interrupt（）函数时讨论。

（2）**全局变量 last_tsc_low**：它表示中断服务 timer_interrupt 真正在 CPU 上执行时刻的 TSC 寄存器值的低 32 位（LSB）。

显然，通过 delay_at_last_interrupt、last_tsc_low 和时刻 x 处的 TSC 寄存器值，我们就可以完全确定时刻 x 距上一次时钟中断产生时刻的时间间隔偏移 offset_usec 的值。实现在 arch/i386/kernel/time.c 中的函数 do_fast_gettimeoffset()就是这样计算时间间隔偏移的，当然它仅在 CPU 配置有 TSC 寄存器时才被使用，后面我们会详细分析这个函数。

7．4 时钟中断的驱动

如前所述，8253 / 8254 PIT 的通道 0 通常被用来在 IRQ0 上产生周期性的时钟中断。对时钟中断的驱动是绝大数操作系统内核实现 time-keeping 的关键所在。不同的 OS 对时钟驱动的要求也不同，但是一般都包含下列要求内容：

1. 维护系统的当前时间与日期。
2. 防止进程运行时间超出其允许的时间。
3. 对 CPU 的使用情况进行记帐统计。
4. 处理用户进程发出的时间系统调用。
5. 对系统某些部分提供监视定时器。

其中，第一项功能是所有 OS 都必须实现的基础功能，它是 OS 内核的运行基础。通常有三种方法可用来维护系统的时间与日期：(1)最简单的一种方法就是用一个 64 位的计数器来对时钟滴答进行计数。(2)第二种方法就是用一个 32 位计数器来对秒进行计数。用一个 32 位的辅助计数器来对时钟滴答计数直至累计一秒为止。因为 2^{32} 超过 136 年，因此这种方法直至 22 世纪都可以工作得很好。(3）第三种方法也是按滴答进行计数，但却是相对于系统启动以来的滴答次数，而不是相对于一个确定的外部时刻。当读后备时钟（如 RTC）或用户输入实际时间时，根据当前的滴答次数计算系统当前时间。

UNIX 类的 OS 通常都采用第三种方法来维护系统的时间与日期。

7．4．1 Linux 对时钟中断的初始化

Linux 对时钟中断的初始化是分为几个步骤来进行的：（1）首先，由 init_IRQ()函数通过调用 init_ISA_IRQ()函数对中断向量 32～256 所对应的中断向量描述符进行初始化设置。显然，这其中也就把 IRQ0（也即中断向量 32）的中断向量描述符初始化了。(2）然后，init_IRQ()函数设置中断向量 32～256 相对应的中断门。(3）init_IRQ()函数对 PIT 进行初始化编程；(4）sched_init()函数对计数器、时间中断的 Bottom Half 进行初始化。(5）最后，由 time_init()函数对 Linux 内核的时钟中断机制进行初始化。这三个初始化函数都是由 init/main.c 文件中的 start_kernel()函数调用的，如下：

```
asmlinkage void __init start_kernel()
{
    …
    trap_init();
    init_IRQ();
    sched_init();
    time_init();
    softirq_init();
```

```
    …
}
```

(1)init_IRQ()函数对 8254 PIT 的初始化编程

函数 init_IRQ()函数在完成中断门的初始化后，就对 8254 PIT 进行初始化编程设置，设置的步骤如下：（1）设置 8254 PIT 的控制寄存器（端口 0x43）的值为“01100100”，也即选择通道 0、先读写 LSB 再读写 MSB、工作模式 2、二进制存储格式。（2）将宏 LATCH 的值写入通道 0 的计数器中（端口 0x40），注意要先写 LATCH 的 LSB，再写 LATCH 的高字节。其源码如下所示（arch/i386/kernel/i8259.c）：

```
void __init init_IRQ(void)
{
    ……
    /*
     * Set the clock to HZ Hz, we already have a valid
     * vector now:
     */
    outb_p(0x34,0x43);        /* binary, mode 2, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);     /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    ……
}
```

（2）sched_init()对定时器机制和时钟中断的 Bottom Half 的初始化

函数 sched_init()中与时间相关的初始化过程主要有两步：（1）调用 init_timervecs()函数初始化内核定时器机制；（2）调用 init_bh()函数将 BH 向量 TIMER_BH、TQUEUE_BH 和 IMMEDIATE_BH 所对应的 BH 函数分别设置成 timer_bh()、tqueue_bh()和 immediate_bh()函数。如下所示（kernel/sched.c）：

```
void __init sched_init(void)
{
    ……
    init_timervecs();

    init_bh(TIMER_BH, timer_bh);
    init_bh(TQUEUE_BH, tqueue_bh);
    init_bh(IMMEDIATE_BH, immediate_bh);
    ……
}
```

（3）time_init()函数对内核时钟中断机制的初始化

前面两个函数所进行的初始化步骤都是为时间中断机制做好准备而已。在执行完 init_IRQ()函数和 sched_init()函数后，CPU 已经可以为 IRQ0 上的时钟中断进行服务了，因为 IRQ0 所对应的中断门已经被设置好指向中断服务函数 IRQ0x20_interrupt()。但是由于此时中断向量 0x20 的中断向量描述符 irq_desc[0]还是处于初始状态（其 status 成员的值为 IRQ_DISABLED），并未挂接任何具体的中断服务描述符，因此这时 CPU 对 IRQ0 的中断服务并没有任何具体意义，而只是按照规定的流程空跑一趟。但是当 CPU 执行完 time_init()函数后，情形就大不一样了。

函数 time_init()主要做三件事：（1）从 RTC 中获取内核启动时的时间与日期；（2）在 CPU 有 TSC 的情况下校准 TSC，以便为后面使用 TSC 做好准备；（3）在 IRQ0 的中断请求描述符中挂接具体的中断服务描述符。其源码如下所示（arch/i386/kernel/time.c）：

```
void __init time_init(void)
{
        extern int x86_udelay_tsc;

        xtime.tv_sec = get_cmos_time();
        xtime.tv_usec = 0;

/*
 * If we have APM enabled or the CPU clock speed is variable
 * (CPU stops clock on HLT or slows clock to save power)
 * then the TSC timestamps may diverge by up to 1 jiffy from
 * 'real time' but nothing will break.
 * The most frequent case is that the CPU is "woken" from a halt
 * state by the timer interrupt itself, so we get 0 error. In the
 * rare cases where a driver would "wake" the CPU and request a
 * timestamp, the maximum error is < 1 jiffy. But timestamps are
 * still perfectly ordered.
 * Note that the TSC counter will be reset if APM suspends
 * to disk; this won't break the kernel, though, 'cuz we're
 * smart.    See arch/i386/kernel/apm.c.
 */
        /*
         *    Firstly we have to do a CPU check for chips with
         *    a potentially buggy TSC. At this point we haven't run
         *    the ident/bugs checks so we must run this hook as it
         *    may turn off the TSC flag.
         *
         *    NOTE: this doesnt yet handle SMP 486 machines where only
         *    some CPU's have a TSC. Thats never worked and nobody has
         *    moaned if you have the only one in the world - you fix it!
         */

        dodgy_tsc();

        if (cpu_has_tsc) {
                unsigned long tsc_quotient = calibrate_tsc();
                if (tsc_quotient) {
                        fast_gettimeoffset_quotient = tsc_quotient;
                        use_tsc = 1;
                        /*
                         *    We could be more selective here I suspect
                         *    and just enable this for the next intel chips ?
                         */
                        x86_udelay_tsc = 1;
#ifndef do_gettimeoffset
                        do_gettimeoffset = do_fast_gettimeoffset;
```

```
#endif
                do_get_fast_time = do_gettimeofday;

                /* report CPU clock rate in Hz.
                 * The formula is (10^6 * 2^32) / (2^32 * 1 / (clocks/us)) =
                 * clock/second. Our precision is about 100 ppm.
                 */
                {       unsigned long eax=0, edx=1000;
                        __asm__("divl %2"
                                :"=a" (cpu_khz), "=d" (edx)
                                        :"r" (tsc_quotient),
                                        "0" (eax), "1" (edx));
                        printk("Detected %lu.%03lu MHz processor.\n", cpu_khz / 1000, cpu_khz % 1000);
                }
            }
        }

#ifdef CONFIG_VISWS
        printk("Starting Cobalt Timer system clock\n");

        /* Set the countdown value */
        co_cpu_write(CO_CPU_TIMEVAL, CO_TIME_HZ/HZ);

        /* Start the timer */
        co_cpu_write(CO_CPU_CTRL, co_cpu_read(CO_CPU_CTRL) | CO_CTRL_TIMERUN);

        /* Enable (unmask) the timer interrupt */
        co_cpu_write(CO_CPU_CTRL, co_cpu_read(CO_CPU_CTRL) & ~CO_CTRL_TIMEMASK);

        /* Wire cpu IDT entry to s/w handler (and Cobalt APIC to IDT) */
        setup_irq(CO_IRQ_TIMER, &irq0);
#else
        setup_irq(0, &irq0);
#endif
}
```

对该函数的注解如下：

（1）调用函数 get_cmos_time()从 RTC 中得到系统启动时的时间与日期，它返回的是当前时间相对于 1970－01－01 00：00：00 这个 UNIX 时间基准的秒数值。因此这个秒数值就被保存在系统全局变量 xtime 的 tv_sec 成员中。而 xtime 的另一个成员 tv_usec 则被初始化为 0。

（2）通过 dodgy_tsc()函数检测 CPU 是否存在时间戳记数器 BUG（I know nothing about it：－）

（3）通过宏 cpu_has_tsc 来确定系统中 CPU 是否存在 TSC 计数器。如果存在 TSC，那么内核就可以用 TSC 来获得更为精确的时间。为了能够用 TSC 来修正内核时间。这里必须作一些初始化工作：①调用 calibrate_tsc()来确定 TSC 的每一次计数真正代表多长的时间间隔（单位为 us），也即一个时钟周期的真正时间间隔长度。②将 calibrate_tsc()函数所返回的值保存在全局变量 fast_gettimeoffset_quotient 中，该变量被用来快速地计算时间偏差；同时还将另一个全局变量 use_tsc 设置为 1，表示内核可以使用 TSC。这两个变量都定义在 arch/i386/kernel/time.c 文件中，如下：

```
/* Cached *multiplier* to convert TSC counts to microseconds.
 * (see the equation below).
 * Equal to 2^32 * (1 / (clocks per usec) ).
 * Initialized in time_init.
 */
unsigned long fast_gettimeoffset_quotient;
……
static int use_tsc;
```

③接下来，将系统全局变量 x86_udelay_tsc 设置为 1，表示可以通过 TSC 来实现微妙级的精确延时。该变量定义在 arch/i386/lib/delay.c 文件中。④将函数指针 do_gettimeoffset 强制性地指向函数 do_fast_gettimeoffset()（与之对应的是 do_slow_gettimeoffset()函数），从而使内核在计算时间偏差时可以用 TSC 这种快速的方法来进行。⑤将函数指针 do_get_fast_time 指向函数 do_gettimeofday()，从而可以让其他内核模块通过 do_gettimeofday()函数来获得更精准的当前时间。⑥计算并报告根据 TSC 所算得的 CPU 时钟频率。

（4）不考虑 CONFIG_VISWS 的情况，因此 time_init()的最后一个步骤就是调用 setup_irq()函数来为 IRQ0 挂接具体的中断服务描述符 irq0。全局变量 irq0 是时钟中断请求的中断服务描述符，其定义如下（arch/i386/kernel/time.c）：

```
static struct irqaction irq0  = { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
```

显然，函数 timer_interrupt()将成为时钟中断的服务程序（ISR），而 SA_INTERRUPT 标志也指定了 timer_interrupt()函数将是在 CPU 关中断的条件下执行的。结构 irq0 中的 next 指针被设置为 NULL，因此 IRQ0 所对应的中断服务队列中只有 irq0 这唯一的一个元素，且 IRQ0 不允许中断共享。

7．4．2 时钟中断服务例程 timer_interrupt()

中断服务描述符 irq0 一旦被钩挂到 IRQ0 的中断服务队列中去后，Linux 内核就可以通过 irq0->handler 函数指针所指向的 timer_interrupt()函数对时钟中断请求进行真正的服务，而不是向前面所说的那样只是让 CPU“空跑”一趟。此时，Linux 内核可以说是真正的“跳动”起来了。

在本节一开始所述的对时钟中断驱动的 5 项要求中，通常只有第一项（即 timekeeping）是最为迫切的，因此必须在时钟中断服务例程中完成。而其余的几个要求可以稍缓，因此可以放在时钟中断的 Bottom Half 中去执行。这样，Linux 内核就是 timer_interrupt()函数的执行时间尽可能的短，因为它是在 CPU 关中断的条件下执行的。

函数 timer_interrupt()的源码如下（arch/i386/kernel/time.c）：

```
/*
 * This is the same as the above, except we _also_ save the current
 * Time Stamp Counter value at the time of the timer interrupt, so that
 * we later on can estimate the time of day more exactly.
 */
static void timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int count;

        /*
         * Here we are in the timer irq handler. We just have irqs locally
         * disabled but we don't know if the timer_bh is running on the other
```

```
         * CPU. We need to avoid to SMP race with it. NOTE: we don' t need
         * the irq version of write_lock because as just said we have irq
         * locally disabled. -arca
         */
        write_lock(&xtime_lock);

        if (use_tsc)
        {
            /*
             * It is important that these two operations happen almost at
             * the same time. We do the RDTSC stuff first, since it's
             * faster. To avoid any inconsistencies, we need interrupts
             * disabled locally.
             */

            /*
             * Interrupts are just disabled locally since the timer irq
             * has the SA_INTERRUPT flag set. -arca
             */

            /* read Pentium cycle counter */

            rdtscl(last_tsc_low);

            spin_lock(&i8253_lock);
            outb_p(0x00, 0x43);     /* latch the count ASAP */

            count = inb_p(0x40);    /* read the latched count */
            count |= inb(0x40) << 8;
            spin_unlock(&i8253_lock);

            count = ((LATCH-1) - count) * TICK_SIZE;
            delay_at_last_interrupt = (count + LATCH/2) / LATCH;
        }

        do_timer_interrupt(irq, NULL, regs);

        write_unlock(&xtime_lock);

    }
```

对该函数的注释如下：

（1）由于函数执行期间要访问全局时间变量 xtime，因此一开就对自旋锁 xtime_lock 进行加锁。

（2）如果内核使用 CPU 的 TSC 寄存器（use_tsc 变量非 0），那么通过 TSC 寄存器来计算从时间中断的产生到 timer_interrupt（）函数真正在 CPU 上执行这之间的时间延迟：

- 调用宏 rdtscl()将 64 位的 TSC 寄存器值中的低 32 位（LSB）读到变量 last_tsc_low 中，以供 do_fast_gettimeoffset()函数计算时间偏差之用。这一步的实质就是将 CPU TSC 寄存器的值更新到

内核对 TSC 的缓存变量 last_tsc_low 中。

- 通过读 8254 PIT 的通道 0 的计数器的当前值来计算时间延迟，为此：首先，对自旋锁 i8253_lock 进行加锁。自旋锁 i8253_lock 的作用就是用来串行化对 8254 PIT 的读写访问。其次，向 8254 的控制寄存器（端口 0x43）中写入值 0x00，以便对通道 0 的计数器进行锁存。最后，通过端口 0x40 将通道 0 的计数器的当前值读到局部变量 count 中，并解锁 i8253_lock。
- 显然，从时间中断的产生到 timer_interrupt()函数真正执行这段时间内，以一共流逝了（（LATCH-1）-count）个时钟周期，因此这个延时长度可以用如下公式计算：

  delay_at_last_interrupt＝（（（LATCH-1）－count）÷LATCH）＊TICK_SIZE

  显然，上述公式的结果是个小数，应对其进行四舍五入，为此，Linux 用下述表达式来计算 delay_at_last_interrupt 变量的值：

  （（（LATCH-1）-count）＊TICK_SIZE＋LATCH/2）／LATCH

  上述被除数表达式中的 LATCH／2 就是用来将结果向上圆整成整数的。

（3）在计算出时间延迟后，最后调用函数 do_timer_interrupt()执行真正的时钟服务。

函数 do_timer_interrupt()的源码如下（arch/i386/kernel/time.c）：

```
/*
 * timer_interrupt() needs to keep up the real-time clock,
 * as well as call the "do_timer()" routine every clocktick
 */
static inline void do_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    。。。。。。
    do_timer(regs);
    。。。。。。。
    /*
     * If we have an externally synchronized Linux clock, then update
     * CMOS clock accordingly every ~11 minutes. Set_rtc_mmss() has to be
     * called as close as possible to 500 ms before the new second starts.
     */
    if ((time_status & STA_UNSYNC) == 0 &&
        xtime.tv_sec > last_rtc_update + 660 &&
        xtime.tv_usec >= 500000 - ((unsigned) tick) / 2 &&
        xtime.tv_usec <= 500000 + ((unsigned) tick) / 2) {
        if (set_rtc_mmss(xtime.tv_sec) == 0)
            last_rtc_update = xtime.tv_sec;
        else
            last_rtc_update = xtime.tv_sec - 600; /* do it again in 60 s */
    }
    ……
}
```

上述代码中省略了许多与 SMP 相关的代码，因为我们不关心 SMP。从上述代码我们可以看出，do_timer_interrupt()函数主要作两件事：

（1）调用 do_timer()函数。

（2）判断是否需要更新 CMOS 时钟（即 RTC）中的时间。Linux 仅在下列三个条件同时成立时才更新 CMOS 时钟：①系统全局时间状态变量 time_status 中没有设置 STA_UNSYNC 标志，也即说明 Linux 有一个外部同步时钟。实际上全局时间状态变量 time_status 仅在一种情况下会被清除 STA_SYNC 标志，那

就是执行 adjtimex()系统调用时（这个 syscall 与 NTP 有关）。②自从上次 CMOS 时钟更新已经过去了 11 分钟。全局变量 last_rtc_update 保存着上次更新 CMOS 时钟的时间。③由于 RTC 存在 Update Cycle，因此最好在一秒时间间隔的中间位置 500ms 左右调用 set_rtc_mmss()函数来更新 CMOS 时钟。因此 Linux 规定仅当全局变量 xtime 的微秒数 tv_usec 在 500000±(tick/2)微秒范围范围之内时，才调用 set_rtc_mmss()函数。如果上述条件均成立，那就调用 set_rtc_mmss()将当前时间 xtime.tv_sec 更新回写到 RTC 中。

如果上面是的 set_rtc_mmss()函数返回 0 值，则表明更新成功。于是就将“最近一次 RTC 更新时间”变量 last_rtc_update 更新为当前时间 xtime.tv_sec。如果返回非 0 值，说明更新失败，于是就让 last_rtc_update＝xtime.tv_sec-600（相当于 last_rtc_update+=60），以便在在 60 秒之后再次对 RTC 进行更新。

函数 do_timer()实现在 kernel/timer.c 文件中，其源码如下：

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
#ifndef CONFIG_SMP
    /* SMP process accounting uses the local APIC timer */

    update_process_times(user_mode(regs));
#endif
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

该函数的核心是完成三个任务：

（1）将表示自系统启动以来的时钟滴答计数变量 jiffies 加 1。

（2）调用 update_process_times()函数更新当前进程的时间统计信息。注意，该函数的参数原型是“int user_tick”，如果本次时钟中断（即时钟滴答）发生时 CPU 正处于用户态下执行，则 user_tick 参数应该为 1；否则如果本次时钟中断发生时 CPU 正处于核心态下执行时，则 user_tick 参数应改为 0。所以这里我们以宏 user_mode(regs)来作为 update_process_times()函数的调用参数。该宏定义在 include/asm-i386/ptrace.h 头文件中，它根据 regs 指针所指向的核心堆栈寄存器结构来判断 CPU 进入中断服务之前是处于用户态下还是处于核心态下。如下所示：

```
#ifdef __KERNEL__
#define user_mode(regs) ((VM_MASK & (regs)->eflags) || (3 & (regs)->xcs))
……
#endif
```

（3）调用 mark_bh()函数激活时钟中断的 Bottom Half 向量 TIMER_BH 和 TQUEUE_BH（注意，TQUEUE_BH 仅在任务队列 tq_timer 不为空的情况下才会被激活）。

至此，内核对时钟中断的服务流程宣告结束，下面我们详细分析一下 update_process_times()函数的实现。

7．4．3 更新时间记帐信息——CPU 分时的实现

函数 update_process_times()被用来在发生时钟中断时更新当前进程以及内核中与时间相关的统计信

息，并根据这些信息作出相应的动作，比如：重新进行调度，向当前进程发出信号等。该函数仅有一个参数 user_tick，取值为 1 或 0，其含义在前面已经叙述过。

该函数的源代码如下（kernel/timer.c）：

```
/*
 * Called from the timer interrupt handler to charge one tick to the current
 * process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id(), system = user_tick ^ 1;

    update_one_process(p, user_tick, system, cpu);
    if (p->pid) {
        if (--p->counter <= 0) {
            p->counter = 0;
            p->need_resched = 1;
        }
        if (p->nice > 0)
            kstat.per_cpu_nice[cpu] += user_tick;
        else
            kstat.per_cpu_user[cpu] += user_tick;
        kstat.per_cpu_system[cpu] += system;
    } else if (local_bh_count(cpu) || local_irq_count(cpu) > 1)
        kstat.per_cpu_system[cpu] += system;
}
```

（1）首先，用 smp_processor_id()宏得到当前进程的 CPU ID。

（2）然后，让局部变量 system＝user_tick^1，表示当发生时钟中断时 CPU 是否正处于核心态下。因此，如果 user_tick=1，则 system=0；如果 user_tick＝0，则 system=1。

（3）调用 update_one_process()函数来更新当前进程的 task_struct 结构中的所有与时间相关的统计信息以及成员变量。该函数还会视需要向当前进程发送相应的信号（signal）。

（4）如果当前进程的 PID 非 0，则执行下列步骤来决定是否重新进行调度，并更新内核时间统计信息：

- 将当前进程的可运行时间片长度（由 task_struct 结构中的 counter 成员表示，其单位是时钟滴答次数）减 1。如果减到 0 值，则说明当前进程已经用完了系统分配给它的的运行时间片，因此必须重新进行调度。于是将当前进程的 task_struct 结构中的 need_resched 成员变量设置为 1，表示需要重新执行调度。
- 如果当前进程的 task_struct 结构中的 nice 成员值大于 0，那么将内核全局统计信息变量 kstat 中的 per_cpu_nice［cpu］值将上 user_tick。否则就将 user_tick 值加到内核全局统计信息变量 kstat 中的 per_cpu_user［cpu］成员上。
- 将 system 变量值加到内核全局统计信息 kstat.per_cpu_system［cpu］上。

（5）否则，就判断当前 CPU 在服务时钟中断前是否处于 softirq 软中断服务的执行中，或则正在服务一次低优先级别的硬件中断中。如果是这样的话，则将 system 变量的值加到内核全局统计信息 kstat.per_cpu.system［cpu］上。

- update_one_process()函数

实现在 kernel/timer.c 文件中的 update_one_process()函数用来在时钟中断发生时更新一个进程的

task_struc 结构中的时间统计信息。其源码如下（kernel/timer.c）：

```
void update_one_process(struct task_struct *p, unsigned long user,
                        unsigned long system, int cpu)
{
    p->per_cpu_utime[cpu] += user;
    p->per_cpu_stime[cpu] += system;
    do_process_times(p, user, system);
    do_it_virt(p, user);
    do_it_prof(p);
}
```

注释如下：

（1）由于在一个进程的整个生命期（Lifetime）中，它可能会在不同的 CPU 上执行，也即一个进程可能一开始在 CPU1 上执行，当它用完在 CPU1 上的运行时间片后，它可能又会被调度到 CPU2 上去执行。另外，当进程在某个 CPU 上执行时，它可能又会在用户态和内核态下分别各执行一段时间。所以为了统计这些事件信息，进程 task_struct 结构中的 per_cpu_utime［NR_CPUS］数组就表示该进程在各 CPU 的用户台下执行的累计时间长度，per_cpu_stime［NR_CPUS］数组就表示该进程在各 CPU 的核心态下执行的累计时间长度；它们都以时钟滴答次数为单位。

所以，update_one_process()函数的第一个步骤就是更新进程在当前 CPU 上的用户态执行时间统计 per_cpu_utime［cpu］和核心态执行时间统计 per_cpu_stime［cpu］。

（2）调用 do_process_times()函数更新当前进程的总时间统计信息。

（3）调用 do_it_virt()函数为当前进程的 ITIMER_VIRTUAL 软件定时器更新时间间隔。

（4）调用 do_it_prof（）函数为当前进程的 ITIMER_PROF 软件定时器更新时间间隔。

- **do_process_times()函数**

函数 do_process_times()将更新指定进程的总时间统计信息。每个进程 task_struct 结构中都有一个成员 times，它是一个 tms 结构类型（include/linux/times.h）：

```
struct tms {
    clock_t tms_utime;    /* 本进程在用户台下的执行时间总和 */
    clock_t tms_stime;    /* 本进程在核心态下的执行时间总和 */
    clock_t tms_cutime;   /* 所有子进程在用户态下的执行时间总和 */
    clock_t tms_cstime;   /* 所有子进程在核心态下的执行时间总和 */
};
```

上述结构的所有成员都以时钟滴答次数为单位。

函数 do_process_times()的源码如下（kernel/timer.c）：

```
static inline void do_process_times(struct task_struct *p,
    unsigned long user, unsigned long system)
{
    unsigned long psecs;

    psecs = (p->times.tms_utime += user);
    psecs += (p->times.tms_stime += system);
    if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_cur) {
        /* Send SIGXCPU every second.. */
        if (!(psecs % HZ))
            send_sig(SIGXCPU, p, 1);
```

```
            /* and SIGKILL when we go over max.. */
            if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_max)
                send_sig(SIGKILL, p, 1);
        }
    }
```

注释如下：

（1）根据参数 user 更新指定进程 task_struct 结构中的 times.tms_utime 值。根据参数 system 更新指定进程 task_struct 结构中的 times.tms_stime 值。

（2）将更新后的 times.tms_utime 值与 times.tms_stime 值的和保存到局部变量 psecs 中，因此 psecs 就表示了指定进程 p 到目前为止已经运行的总时间长度（以时钟滴答次数计）。如果这一总运行时间长超过进程 P 的资源限额，那就每隔 1 秒给进程发送一个信号 SIGXCPU；如果运行时间长度超过了进程资源限额的最大值，那就发送一个 SIGKILL 信号杀死该进程。

- **do_it_virt()函数**

每个进程都有一个用户态执行时间的 itimer 软件定时器。进程任务结构 task_struct 中的 it_virt_value 成员是这个软件定时器的时间计数器。当进程在用户态下执行时，每一次时钟滴答都使计数器 it_virt_value 减 1，当减到 0 时内核向进程发送 SIGVTALRM 信号，并重置初值。初值保存在进程的 task_struct 结构的 it_virt_incr 成员中。

函数 do_it_virt()的源码如下（kernel/timer.c）：

```
static inline void do_it_virt(struct task_struct * p, unsigned long ticks)
{
    unsigned long it_virt = p->it_virt_value;

    if (it_virt) {
        it_virt -= ticks;
        if (!it_virt) {
            it_virt = p->it_virt_incr;
            send_sig(SIGVTALRM, p, 1);
        }
        p->it_virt_value = it_virt;
    }
}
```

- do_it_prof（）函数

类似地，每个进程也都有一个 itimer 软件定时器 ITIMER_PROF。进程 task_struct 中的 it_prof_value 成员就是这个定时器的时间计数器。不管进程是在用户态下还是在内核态下运行，每个时钟滴答都使 it_prof_value 减 1。当减到 0 时内核就向进程发送 SIGPROF 信号，并重置初值。初值保存在进程 task_struct 结构中的 it_prof_incr 成员中。

函数 do_it_prof()就是用来完成上述功能的，其源码如下（kernel/timer.c）：

```
static inline void do_it_prof(struct task_struct *p)
{
    unsigned long it_prof = p->it_prof_value;

    if (it_prof) {
        if (--it_prof == 0) {
            it_prof = p->it_prof_incr;
```

```
                send_sig(SIGPROF, p, 1);
            }
            p->it_prof_value = it_prof;
        }
    }
```

7．5 时钟中断的 Bottom Half

与时钟中断相关的 Bottom Half 向两主要有两个：TIMER_BH 和 TQUEUE_BH。与 TIMER_BH 相对应的 BH 函数是 timer_bh()，与 TQUEUE_BH 对应的函数是 tqueue_bh()。它们均实现在 kernel/timer.c 文件中。

7．5．1 TQUEUE_BH 向量

TQUEUE_BH 的作用是用来运行 tq_timer 这个任务队列中的任务。因此 do_timer()函数仅仅在 tq_timer 任务队列不为空的情况才激活 TQUEUE_BH 向量。函数 tqueue_bh()的实现非常简单，它只是简单地调用 run_task_queue()函数来运行任务队列 tq_timer。如下所示：

```
void tqueue_bh(void)
{
    run_task_queue(&tq_timer);
}
```

任务对列 tq_timer 也是定义在 kernel/timer.c 文件中，如下所示：

```
DECLARE_TASK_QUEUE(tq_timer);
```

7．5．2 TIMER_BH 向量

TIMER_BH 这个 Bottom Half 向量是 Linux 内核时钟中断驱动的一个重要辅助部分。内核在每一次对时钟中断的服务快要结束时，都会无条件地激活一个 TIMER_BH 向量，以使得内核在稍后一段延迟后执行相应的 BH 函数——timer_bh()。该任务的源码如下：

```
void timer_bh(void)
{
    update_times();
    run_timer_list();
}
```

从上述源码可以看出，内核在时钟中断驱动的底半部分主要有两个任务：（1）调用 update_times()函数来更新系统全局时间 xtime；（2）调用 run_timer_list()函数来执行定时器。关于定时器我们将在下一节讨论。本节我们主要讨论 TIMER_BH 的第一个任务——对内核时间 xtime 的更新。

我们都知道，内核局部时间 xtime 是用来供用户程序通过时间 syscall 来检索或设置当前系统时间的，而内核代码在大多数情况下都引用 jiffies 变量，而很少使用 xtime（偶尔也会有引用 xtime 的情况，比如更新 inode 的时间标记）。因此，对于时钟中断服务程序 timer_interrupt（）而言，jiffies 变量的更新是最紧迫

的，而 xtime 的更新则可以延迟到中断服务的底半部分来进行。

由于 Bottom Half 机制在执行时间具有某些不确定性，因此在 timer_bh()函数得到真正执行之前，期间可能又会有几次时钟中断发生。这样就会造成时钟滴答的丢失现象。为了处理这种情况，Linux 内核使用了一个辅助全局变量 wall_jiffies，来表示上一次更新 xtime 时的 jiffies 值。其定义如下（kernel/timer.c）：

```
/* jiffies at the most recent update of wall time */
unsigned long wall_jiffies;
```

而 timer_bh()函数真正执行时的 jiffies 值与 wall_jiffies 的差就是在 timer_bh()真正执行之前所发生的时钟中断次数。

函数 update_times()的源码如下（kernel/timer.c）：

```
static inline void update_times(void)
{
        unsigned long ticks;

        /*
         * update_times() is run from the raw timer_bh handler so we
         * just know that the irqs are locally enabled and so we don't
         * need to save/restore the flags of the local CPU here. -arca
         */
        write_lock_irq(&xtime_lock);

        ticks = jiffies - wall_jiffies;
        if (ticks) {
                wall_jiffies += ticks;
                update_wall_time(ticks);
        }
        write_unlock_irq(&xtime_lock);
        calc_load(ticks);
}
```

（1）首先，根据 jiffies 和 wall_jiffies 的差值计算在此之前一共发生了几次时钟滴答，并将这个值保存到局部变量 ticks 中。并在 ticks 值大于 0 的情况下(ticks 大于等于 1，一般情况下为 1)：①更新 wall_jiffies 为 jiffies 变量的当前值（wall_jiffies＋＝ticks 等价于 wall_jiffies＝jiffies)。②以参数 ticks 调用 update_wall_time()函数去真正地更新全局时间 xtime。

（2）调用 calc_load()函数去计算系统负载情况。这里我们不去深究它。

函数 update_wall_time()函数根据参数 ticks 所指定的时钟滴答次数相应地更新内核全局时间变量 xtime。其源码如下（kernel/timer.c）：

```
/*
 * Using a loop looks inefficient, but "ticks" is
 * usually just one (we shouldn't be losing ticks,
 * we're doing this this way mainly for interrupt
 * latency reasons, not because we think we'll
 * have lots of lost timer ticks
 */
static void update_wall_time(unsigned long ticks)
{
        do {
```

```
            ticks--;
            update_wall_time_one_tick();
        } while (ticks);

        if (xtime.tv_usec >= 1000000) {
            xtime.tv_usec -= 1000000;
            xtime.tv_sec++;
            second_overflow();
        }
    }
```

对该函数的注释如下：

（1）首先，用一个 do{}循环来根据参数 ticks 的值一次一次调用 update_wall_time_one_tick()函数来为一次时钟滴答更新 xtime 中的 tv_usec 成员。

（2）根据需要调整 xtime 中的秒数成员 tv_usec 和微秒数成员 tv_usec。如果微秒数成员 tv_usec 的值超过 10^6，则说明已经过了一秒钟。因此将 tv_usec 的值减去 1000000，并将秒数成员 tv_sec 的值加 1，然后调用 second_overflow（）函数来处理微秒数成员溢出的情况。

函数 update_wall_time_one_tick（）用来更新一次时钟滴答对系统全局时间 xtime 的影响。由于 tick 全局变量表示了一次时钟滴答的时间间隔长度（以 us 为单位），因此该函数的实现中最核心的代码就是将 xtime 的 tv_usec 成员增加 tick 微秒。这里我们不去关心函数实现中与 NTP（Network Time Protocol）和系统调用 adjtimex（）的相关部分。其源码如下（kernel/timer.c）：

```
/* in the NTP reference this is called "hardclock()" */
static void update_wall_time_one_tick(void)
{
    if ( (time_adjust_step = time_adjust) != 0 ) {
        /* We are doing an adjtime thing.
         *
         * Prepare time_adjust_step to be within bounds.
         * Note that a positive time_adjust means we want the clock
         * to run faster.
         *
         * Limit the amount of the step to be in the range
         * -tickadj .. +tickadj
         */
         if (time_adjust > tickadj)
        time_adjust_step = tickadj;
         else if (time_adjust < -tickadj)
        time_adjust_step = -tickadj;

        /* Reduce by this step the amount of time left  */
        time_adjust -= time_adjust_step;
    }
    xtime.tv_usec += tick + time_adjust_step;
    /*
     * Advance the phase, once it gets to one microsecond, then
     * advance the tick more.
```

```
         */
        time_phase += time_adj;
        if (time_phase <= -FINEUSEC) {
            long ltemp = -time_phase >> SHIFT_SCALE;
            time_phase += ltemp << SHIFT_SCALE;
            xtime.tv_usec -= ltemp;
        }
        else if (time_phase >= FINEUSEC) {
            long ltemp = time_phase >> SHIFT_SCALE;
            time_phase -= ltemp << SHIFT_SCALE;
            xtime.tv_usec += ltemp;
        }
    }
```

7．6 内核定时器机制

Linux 内核 2.4 版中去掉了老版本内核中的静态定时器机制，而只留下动态定时器。相应地在 timer_bh()函数中也不再通过 run_old_timers()函数来运行老式的静态定时器。动态定时器与静态定时器这二个概念是相对于 Linux 内核定时器机制的可扩展功能而言的，动态定时器是指内核的定时器队列是可以动态变化的，然而就定时器本身而言，二者并无本质的区别。考虑到静态定时器机制的能力有限，因此 Linux 内核 2.4 版中完全去掉了以前的静态定时器机制。

7．6．1 Linux 内核对定时器的描述

Linux 在 include/linux/timer.h 头文件中定义了数据结构 timer_list 来描述一个内核定时器：

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

各数据成员的含义如下：

（1）双向链表元素 list：用来将多个定时器连接成一条双向循环队列。

（2）expires：指定定时器到期的时间，这个时间被表示成自系统启动以来的时钟滴答计数（也即时钟节拍数）。当一个定时器的 expires 值小于或等于 jiffies 变量时，我们就说这个定时器已经超时或到期了。在初始化一个定时器后，通常把它的 expires 域设置成当前 expires 变量的当前值加上某个时间间隔值（以时钟滴答次数计）。

（3）函数指针 function：指向一个可执行函数。当定时器到期时，内核就执行 function 所指定的函数。而 data 域则被内核用作 function 函数的调用参数。

内核函数 init_timer()用来初始化一个定时器。实际上，这个初始化函数仅仅将结构中的 list 成员初始化为空。如下所示（include/linux/timer.h）：

```
static inline void init_timer(struct timer_list * timer)
{
    timer->list.next = timer->list.prev = NULL;
}
```

由于定时器通常被连接在一个双向循环队列中等待执行（此时我们说定时器处于 pending 状态）。因此函数 time_pending()就可以用 list 成员是否为空来判断一个定时器是否处于 pending 状态。如下所示（include/linux/timer.h）：

```
static inline int timer_pending (const struct timer_list * timer)
{
    return timer->list.next != NULL;
}
```

- **时间比较操作**

在定时器应用中经常需要比较两个时间值，以确定 timer 是否超时，所以 Linux 内核在 timer.h 头文件中定义了 4 个时间关系比较操作宏。这里我们说时刻 a 在时刻 b 之后，就意味着时间值 a≥b。Linux 强烈推荐用户使用它所定义的下列 4 个时间比较操作宏（include/linux/timer.h）：

```
#define time_after(a,b)         ((long)(b) - (long)(a) < 0)
#define time_before(a,b)time_after(b,a)

#define time_after_eq(a,b)      ((long)(a) - (long)(b) >= 0)
#define time_before_eq(a,b)  time_after_eq(b,a)
```

7．6．2 动态内核定时器机制的原理

Linux 是怎样为其内核定时器机制提供动态扩展能力的呢？其关键就在于“定时器向量”的概念。所谓“定时器向量”就是指这样一条双向循环定时器队列（对列中的每一个元素都是一个 timer_list 结构）：对列中的所有定时器都在同一个时刻到期，也即对列中的每一个 timer_list 结构都具有相同的 expires 值。显然，可以用一个 timer_list 结构类型的指针来表示一个定时器向量。

显然，定时器 expires 成员的值与 jiffies 变量的差值决定了一个定时器将在多长时间后到期。在 32 位系统中，这个时间差值的最大值应该是 0xffffffff。因此如果是基于“定时器向量”基本定义，内核将至少要维护 0xffffffff 个 timer_list 结构类型的指针，这显然是不现实的。

另一方面，从内核本身这个角度看，它所关心的定时器显然不是那些已经过期而被执行过的定时器（这些定时器完全可以被丢弃），也不是那些要经过很长时间才会到期的定时器，而是那些当前已经到期或者马上就要到期的定时器（注意！时间间隔是以滴答次数为计数单位的）。

基于上述考虑，并假定一个定时器要经过 interval 个时钟滴答后才到期（interval＝expires－jiffies），则 Linux 采用了下列思想来实现其动态内核定时器机制：对于那些 0≤interval≤255 的定时器，Linux 严格按照定时器向量的基本语义来组织这些定时器，也即 Linux 内核最关心那些在接下来的 255 个时钟节拍内就要到期的定时器，因此将它们按照各自不同的 expires 值组织成 256 个定时器向量。而对于那些 256≤interval≤0xffffffff 的定时器，由于他们离到期还有一段时间，因此内核并不关心他们，而是将它们以一种扩展的定时器向量语义（或称为“松散的定时器向量语义”）进行组织。所谓“松散的定时器向量语义”就是指：各定时器的 expires 值可以互不相同的一个定时器队列。

具体的组织方案可以分为两大部分：

（1）对于内核最关心的、interval 值在［0，255］之间的前 256 个定时器向量，内核是这样组织它们的：这 256 个定时器向量被组织在一起组成一个定时器向量数组，并作为数据结构 timer_vec_root 的

一部分，该数据结构定义在 kernel/timer.c 文件中，如下述代码段所示：

```
/*
 * Event timer code
 */
#define TVN_BITS 6
#define TVR_BITS 8
#define TVN_SIZE (1 << TVN_BITS)
#define TVR_SIZE (1 << TVR_BITS)
#define TVN_MASK (TVN_SIZE - 1)
#define TVR_MASK (TVR_SIZE - 1)

struct timer_vec {
    int index;
    struct list_head vec[TVN_SIZE];
};

struct timer_vec_root {
    int index;
    struct list_head vec[TVR_SIZE];
};

static struct timer_vec tv5;
static struct timer_vec tv4;
static struct timer_vec tv3;
static struct timer_vec tv2;
static struct timer_vec_root tv1;

static struct timer_vec * const tvecs[] = {
    (struct timer_vec *)&tv1, &tv2, &tv3, &tv4, &tv5
};

#define NOOF_TVECS (sizeof(tvecs) / sizeof(tvecs[0]))
```

基于数据结构 timer_vec_root，Linux 定义了一个全局变量 tv1，以表示内核所关心的前 256 个定时器向量。这样内核在处理是否有到期定时器时，它就只从定时器向量数组 tv1.vec［256］中的某个定时器向量内进行扫描。而 tv1 的 index 字段则指定当前正在扫描定时器向量数组 tv1.vec［256］中的哪一个定时器向量，也即该数组的索引，其初值为 0，最大值为 255（以 256 为模）。每个时钟节拍时 index 字段都会加 1。显然，index 字段所指定的定时器向量 tv1.vec［index］中包含了当前时钟节拍内已经到期的所有动态定时器。而定时器向量 tv1.vec［index＋k］则包含了接下来第 k 个时钟节拍时刻将到期的所有动态定时器。当 index 值又重新变为 0 时，就意味着内核已经扫描了 tv1 变量中的所有 256 个定时器向量。在这种情况下就必须将那些以松散定时器向量语义来组织的定时器向量补充到 tv1 中来。

（2）而对于内核不关心的、interval 值在［0xff，0xffffffff］之间的定时器，它们的到期紧迫程度也随其 interval 值的不同而不同。显然 interval 值越小，定时器紧迫程度也越高。因此在将它们以松散定时器向量进行组织时也应该区别对待。通常，定时器的 interval 值越小，它所处的定时器向量的松散度也就越低（也即向量中的各定时器的 expires 值相差越小）；而 interval 值越大，它所处的定时器向量的松散度也就越大（也即向量中的各定时器的 expires 值相差越大）。

内核规定，对于那些满足条件：0x100≤interval≤0x3fff 的定时器，只要表达式（interval>>8）具

有相同值的定时器都将被组织在同一个松散定时器向量中。因此，为组织所有满足条件 0x100≤interval≤0x3fff 的定时器，就需要 2^6=64 个松散定时器向量。同样地，为方便起见，这 64 个松散定时器向量也放在一起形成数组，并作为数据结构 timer_vec 的一部分。基于数据结构 timer_vec，Linux 定义了全局变量 tv2，来表示这 64 条松散定时器向量。如上述代码段所示。

对于那些满足条件 0x4000≤interval≤0xfffff 的定时器，只要表达式（interval>>8＋6）的值相同的定时器都将被放在同一个松散定时器向量中。同样，要组织所有满足条件 0x4000≤interval≤0xfffff 的定时器，也需要 2^6=64 个松散定时器向量。类似地，这 64 个松散定时器向量也可以用一个 timer_vec 结构来描述，相应地 Linux 定义了 tv3 全局变量来表示这 64 个松散定时器向量。

对于那些满足条件 0x100000≤interval≤0x3ffffff 的定时器，只要表达式（interval>>8＋6＋6）的值相同的定时器都将被放在同一个松散定时器向量中。同样，要组织所有满足条件 0x100000≤interval≤0x3ffffff 的定时器，也需要 2^6=64 个松散定时器向量。类似地，这 64 个松散定时器向量也可以用一个 timer_vec 结构来描述，相应地 Linux 定义了 tv4 全局变量来表示这 64 个松散定时器向量。

对于那些满足条件 0x4000000≤interval≤0xffffffff 的定时器，只要表达式（interval>>8＋6＋6＋6）的值相同的定时器都将被放在同一个松散定时器向量中。同样，要组织所有满足条件 0x4000000≤interval≤0xffffffff 的定时器，也需要 2^6=64 个松散定时器向量。类似地，这 64 个松散定时器向量也可以用一个 timer_vec 结构来描述，相应地 Linux 定义了 tv5 全局变量来表示这 64 个松散定时器向量。

最后，为了引用方便，Linux 定义了一个指针数组 tvecs []，来分别指向 tv1、tv2、…、tv5 结构变量。如上述代码所示。

整个内核定时器机制的总体结构如下图 7－8 所示：

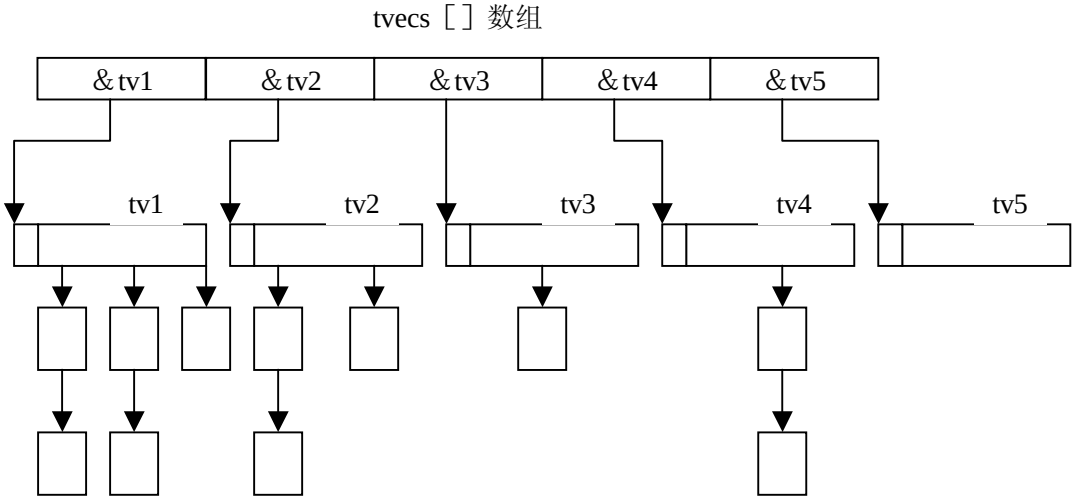


图 7-8 内核动态定时器机制的结构

7．6．3 内核动态定时器机制的实现

在内核动态定时器机制的实现中，有三个操作时非常重要的：（1）将一个定时器插入到它应该所处的定时器向量中。（2）定时器的迁移，也即将一个定时器从它原来所处的定时器向量迁移到另一个定时器向量中。（3）扫描并执行当前已经到期的定时器。

7．6．3．1 动态定时器机制的初始化

函数 init_timervecs()实现对动态定时器机制的初始化。该函数仅被 sched_init()初始化例程所调用。动态定时器机制初始化过程的主要任务就是将 tv1、tv2、…、tv5 这 5 个结构变量中的定时器向量指针数组 vec［］初始化为 NULL。如下所示（kernel/timer.c）：

```
void init_timervecs (void)
{
    int i;

    for (i = 0; i < TVN_SIZE; i++) {
        INIT_LIST_HEAD(tv5.vec + i);
        INIT_LIST_HEAD(tv4.vec + i);
        INIT_LIST_HEAD(tv3.vec + i);
        INIT_LIST_HEAD(tv2.vec + i);
    }
    for (i = 0; i < TVR_SIZE; i++)
        INIT_LIST_HEAD(tv1.vec + i);
}
```

上述函数中的宏 TVN_SIZE 是指 timer_vec 结构类型中的定时器向量指针数组 vec［］的大小，值为 64。宏 TVR_SIZE 是指 timer_vec_root 结构类型中的定时器向量数组 vec［］的大小，值为 256。

7．6．3．2 动态定时器的时钟滴答基准 timer_jiffies

由于动态定时器是在时钟中断的 Bottom Half 中被执行的，而从 TIMER_BH 向量被激活到其 timer_bh()函数真正执行这段时间内可能会有几次时钟中断发生。因此内核必须记住上一次运行定时器机制是什么时候，也即内核必须保存上一次运行定时器机制时的 jiffies 值。为此，Linux 在 kernel/timer.c 文件中定义了全局变量 timer_jiffies 来表示上一次运行定时器机制时的 jiffies 值。该变量的定义如下所示：

```
static unsigned long timer_jiffies;
```

7．6．3．3 对内核动态定时器链表的保护

由于内核动态定时器链表是一种系统全局共享资源，为了实现对它的互斥访问，Linux 定义了专门的自旋锁 timerlist_lock 来保护。任何想要访问动态定时器链表的代码段都首先必须先持有该自旋锁，并且在访问结束后释放该自旋锁。其定义如下（kernel/timer.c）：

```
/* Initialize both explicitly - let's try to have them in the same cache line */
spinlock_t timerlist_lock = SPIN_LOCK_UNLOCKED;
```

7．6．3．4 将一个定时器插入到链表中

函数 add_timer()用来将参数 timer 指针所指向的定时器插入到一个合适的定时器链表中。它首先调用

timer_pending()函数判断所指定的定时器是否已经位于在某个定时器向量中等待执行。如果是，则不进行任何操作，只是打印一条内核告警信息就返回了；如果不是，则调用 internal_add_timer()函数完成实际的插入操作。其源码如下（kernel/timer.c）：

```
void add_timer(struct timer_list *timer)
{
	unsigned long flags;

	spin_lock_irqsave(&timerlist_lock, flags);
	if (timer_pending(timer))
		goto bug;
	internal_add_timer(timer);
	spin_unlock_irqrestore(&timerlist_lock, flags);
	return;
bug:
	spin_unlock_irqrestore(&timerlist_lock, flags);
	printk("bug: kernel timer added twice at %p.\n",
			__builtin_return_address(0));
}
```

函数 internal_add_timer()用于将一个不处于任何定时器向量中的定时器插入到它应该所处的定时器向量中去（根据定时器的 expires 值来决定）。如下所示（kernel/timer.c）：

```
static inline void internal_add_timer(struct timer_list *timer)
{
	/*
	 * must be cli-ed when calling this
	 */
	unsigned long expires = timer->expires;
	unsigned long idx = expires - timer_jiffies;
	struct list_head * vec;

	if (idx < TVR_SIZE) {
		int i = expires & TVR_MASK;
		vec = tv1.vec + i;
	} else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
		int i = (expires >> TVR_BITS) & TVN_MASK;
		vec = tv2.vec + i;
	} else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
		int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
		vec =    tv3.vec + i;
	} else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
		int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
		vec = tv4.vec + i;
	} else if ((signed long) idx < 0) {
		/* can happen if you add a timer with expires == jiffies,
		 * or you set a timer to go off in the past
		 */
```

```
            vec = tv1.vec + tv1.index;
        } else if (idx <= 0xffffffffUL) {
            int i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
            vec = tv5.vec + i;
        } else {
            /* Can only get here on architectures with 64-bit jiffies */
            INIT_LIST_HEAD(&timer->list);
            return;
        }
        /*
         * Timers are FIFO!
         */
        list_add(&timer->list, vec->prev);
    }
```

对该函数的注释如下：

（1）首先，计算定时器的 expires 值与 timer_jiffies 的插值（注意！这里应该使用动态定时器自己的时间基准），这个差值就表示这个定时器相对于上一次运行定时器机制的那个时刻还需要多长时间间隔才到期。局部变量 idx 保存这个差值。

（2）根据 idx 的值确定这个定时器应被插入到哪一个定时器向量中。其具体的确定方法我们在 7.6.2 节已经说过了，这里不再详述。最后，定时器向量的头部指针 vec 表示这个定时器应该所处的定时器向量链表头部。

（3）最后，调用 list_add()函数将定时器插入到 vec 指针所指向的定时器队列的尾部。

7．6．3．5 修改一个定时器的 expires 值

当一个定时器已经被插入到内核动态定时器链表中后，我们还可以修改该定时器的 expires 值。函数 mod_timer()实现这一点。如下所示（kernel/timer.c）：

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

    spin_lock_irqsave(&timerlist_lock, flags);
    timer->expires = expires;
    ret = detach_timer(timer);
    internal_add_timer(timer);
    spin_unlock_irqrestore(&timerlist_lock, flags);
    return ret;
}
```

该函数首先根据参数 expires 值更新定时器的 expires 成员。然后调用 detach_timer()函数将该定时器从它原来所属的链表中删除。最后调用 internal_add_timer()函数将该定时器根据它新的 expires 值重新插入到相应的链表中。

函数 detach_timer()首先调用 timer_pending()来判断指定的定时器是否已经处于某个链表中，如果定时

器原来就不处于任何链表中，则 detach_timer()函数什么也不做，直接返回 0 值，表示失败。否则，就调用 list_del()函数将定时器从它原来所处的链表中摘除。如下所示（kernel/timer.c）：

```
static inline int detach_timer (struct timer_list *timer)
{
    if (!timer_pending(timer))
        return 0;
    list_del(&timer->list);
    return 1;
}
```

7．6．3．6 删除一个定时器

函数 del_timer()用来将一个定时器从相应的内核定时器队列中删除。该函数实际上是对 detach_timer()函数的高层封装。如下所示（kernel/timer.c）：

```
int del_timer(struct timer_list * timer)
{
    int ret;
    unsigned long flags;

    spin_lock_irqsave(&timerlist_lock, flags);
    ret = detach_timer(timer);
    timer->list.next = timer->list.prev = NULL;
    spin_unlock_irqrestore(&timerlist_lock, flags);
    return ret;
}
```

7．6．3．7 定时器迁移操作

由于一个定时器的 interval 值会随着时间的不断流逝（即 jiffies 值的不断增大）而不断变小，因此那些原本到期紧迫程度较低的定时器会随着 jiffies 值的不断增大而成为既将马上到期的定时器。比如定时器向量 tv2.vec[0]中的定时器在经过 256 个时钟滴答后会成为未来 256 个时钟滴答内会到期的定时器。因此，定时器在内核动态定时器链表中的位置也应相应地随着改变。改变的规则是：当 tv1.index 重新变为 0 时（意味着 tv1 中的 256 个定时器向量都已被内核扫描一遍了，从而使 tv1 中的 256 个定时器向量变为空），则用 tv2.vec［index］定时器向量中的定时器去填充 tv1，同时使 tv2.index 加 1（它以 64 为模）。当 tv2.index 重新变为 0（意味着 tv2 中的 64 个定时器向量都已经被全部填充到 tv1 中去了，从而使得 tv2 变为空），则用 tv3.vec［index］定时器向量中的定时器去填充 tv2。如此一直类推下去，直到 tv5。

函数 cascade_timers()完成这种定时器迁移操作，该函数只有一个 timer_vec 结构类型指针的参数 tv。这个函数将把定时器向量 tv->vec［tv->index］中的所有定时器重新填充到上一层定时器向量中去。如下所示（kernel/timer.c）：

```
static inline void cascade_timers(struct timer_vec *tv)
{
    /* cascade all the timers from tv up one level */
    struct list_head *head, *curr, *next;
```

```
        head = tv->vec + tv->index;
        curr = head->next;
        /*
         * We are removing _all_ timers from the list, so we don't  have to
         * detach them individually, just clear the list afterwards.
         */
        while (curr != head) {
            struct timer_list *tmp;

            tmp = list_entry(curr, struct timer_list, list);
            next = curr->next;
            list_del(curr); // not needed
            internal_add_timer(tmp);
            curr = next;
        }
        INIT_LIST_HEAD(head);
        tv->index = (tv->index + 1) & TVN_MASK;
    }
```

对该函数的注释如下：

（1）首先，用指针 head 指向定时器头部向量头部的 list_head 结构。指针 curr 指向定时器向量中的第一个定时器。

（2）然后，用一个 while{}循环来遍历定时器向量 tv->vec［tv->index］。由于定时器向量是一个双向循环队列，因此循环的终止条件是 curr=head。对于每一个被扫描的定时器，循环体都先调用 list_del()函数将当前定时器从链表中摘除，然后调用 internal_add_timer()函数重新确定该定时器应该被放到哪个定时器向量中去。

（3）当从 while{}循环退出后，定时器向量 tv->vec［tv->index］中所有的定时器都已被迁移到其它地方（到它们该呆的地方：－），因此它本身就成为一个空队列。这里我们显示地调用 INIT_LIST_HEAD()宏来将定时器向量的表头结构初始化为空。

（4）最后，将 tv->index 值加 1，当然它是以 64 为模。

7．6．4．8 扫描并执行当前已经到期的定时器

函数 run_timer_list()完成这个功能。如前所述，该函数是被 timer_bh()函数所调用的，因此内核定时器是在时钟中断的 Bottom Half 中被执行的。记住这一点非常重要。全局变量 timer_jiffies 表示了内核上一次执行 run_timer_list()函数的时间，因此 jiffies 与 timer_jiffies 的差值就表示了自从上一次处理定时器以来，期间一共发生了多少次时钟中断，显然 run_timer_list()函数必须为期间所发生的每一次时钟中断补上定时器服务。该函数的源码如下（kernel/timer.c）：

```
static inline void run_timer_list(void)
{
    spin_lock_irq(&timerlist_lock);
    while ((long)(jiffies - timer_jiffies) >= 0) {
        struct list_head *head, *curr;
        if (!tv1.index) {
```

```
                int n = 1;
                do {
                    cascade_timers(tvecs[n]);
                } while (tvecs[n]->index == 1 && ++n < NOOF_TVECS);
            }
repeat:
            head = tv1.vec + tv1.index;
            curr = head->next;
            if (curr != head) {
                struct timer_list *timer;
                void (*fn)(unsigned long);
                unsigned long data;

                timer = list_entry(curr, struct timer_list, list);
                fn = timer->function;
                data= timer->data;

                detach_timer(timer);
                timer->list.next = timer->list.prev = NULL;
                timer_enter(timer);
                spin_unlock_irq(&timerlist_lock);
                fn(data);
                spin_lock_irq(&timerlist_lock);
                timer_exit();
                goto repeat;
            }
            ++timer_jiffies;
            tv1.index = (tv1.index + 1) & TVR_MASK;
        }
        spin_unlock_irq(&timerlist_lock);
    }
```

函数 run_timer_list()的执行过程主要就是用一个大 while{}循环来为时钟中断执行定时器服务，每一次循环服务一次时钟中断。因此一共要执行（jiffies－timer_jiffies＋1）次循环。循环体所执行的服务步骤如下：

（1）首先，判断 tv1.index 是否为 0，如果为 0 则需要从 tv2 中补充定时器到 tv1 中来。但 tv2 也可能为空而需要从 tv3 中补充定时器，因此用一个 do{}while 循环来调用 cascade_timer()函数来依次视需要从 tv2 中补充 tv1，从 tv3 中补充 tv2、…、从 tv5 中补充 tv4。显然如果 tvi.index=0（2≤i≤5），则对于 tvi 执行 cascade_timers()函数后，tvi.index 肯定为 1。反过来讲，如果对 tvi 执行过 cascade_timers()函数后 tvi.index 不等于 1，那么可以肯定在未对 tvi 执行 cascade_timers()函数之前，tvi.index 值肯定不为 0，因此这时 tvi 不需要从 tv(i+1)中补充定时器，这时就可以终止 do{}while 循环。

（2）接下来，就要执行定时器向量 tv1.vec［tv1.index］中的所有到期定时器。因此这里用一个 goto repeat 循环从头到尾依次扫描整个定时器对列。由于在执行定时器的关联函数时并不需要关 CPU 中断，所以在用 detach_timer()函数将当前定时器从对列中摘除后，就可以调用 spin_unlock_irq()函数进行解锁和开中断，然后在执行完当前定时器的关联函数后重新用 spin_lock_irq（）函数加锁和关中断。

（3）当执行完定时器向量 tv1.vec[tv1.index]中的所有到期定时器后，tv1.vec［tv1.index］应该是个空队列。至此这一次定时器服务也就宣告结束。

（4）最后，将 timer_jiffies 值加 1，将 tv1.index 值加 1，当然它的模是 256。然后，回到 while 循环开

始下一次定时器服务。

7．7 进程间隔定时器 itimer

所谓“间隔定时器（Interval Timer，简称 itimer）就是指定时器采用“间隔”值（interval）来作为计时方式，当定时器启动后，间隔值 interval 将不断减小。当 interval 值减到 0 时，我们就说该间隔定时器到期。与上一节所说的内核动态定时器相比，二者最大的区别在于定时器的计时方式不同。内核定时器是通过它的到期时刻 expires 值来计时的，当全局变量 jiffies 值大于或等于内核动态定时器的 expires 值时，我们说内核内核定时器到期。而间隔定时器则实际上是通过一个不断减小的计数器来计时的。虽然这两种定时器并不相同，但却也是相互联系的。假如我们每个时钟节拍都使间隔定时器的间隔计数器减 1，那么在这种情形下间隔定时器实际上就是内核动态定时器（下面我们会看到进程的真实间隔定时器就是这样通过内核定时器来实现的）。

间隔定时器主要被应用在用户进程上。每个 Linux 进程都有三个相互关联的间隔定时器。其各自的间隔计数器都定义在进程的 task_struct 结构中，如下所示（include/linux/sched.h）：

```
struct task_struct {
    ……
    unsigned long it_real_value, it_prof_value, it_virt_value;
    unsigned long it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list real_timer;
    ……
}
```

（1）真实间隔定时器（ITIMER_REAL）：这种间隔定时器在启动后，不管进程是否运行，每个时钟滴答都将其间隔计数器减 1。当减到 0 值时，内核向进程发送 SIGALRM 信号。结构类型 task_struct 中的成员 it_real_incr 则表示真实间隔定时器的间隔计数器的初始值，而成员 it_real_value 则表示真实间隔定时器的间隔计数器的当前值。由于这种间隔定时器本质上与上一节的内核定时器时一样的，因此 Linux 实际上是通过 real_timer 这个内嵌在 task_struct 结构中的内核动态定时器来实现真实间隔定时器 ITIMER_REAL 的。

（2）虚拟间隔定时器 ITIMER_VIRT：也称为进程的用户态间隔定时器。结构类型 task_struct 中成员 it_virt_incr 和 it_virt_value 分别表示虚拟间隔定时器的间隔计数器的初始值和当前值，二者均以时钟滴答次数位计数单位。当虚拟间隔定时器启动后，只有当进程在用户态下运行时，一次时钟滴答才能使间隔计数器当前值 it_virt_value 减 1。当减到 0 值时，内核向进程发送 SIGVTALRM 信号（虚拟闹钟信号），并将 it_virt_value 重置为初值 it_virt_incr。具体请见 7.4.3 节中的 do_it_virt()函数的实现。

（3）PROF 间隔定时器 ITIMER_PROF：进程的 task_struct 结构中的 it_prof_value 和 it_prof_incr 成员分别表示 PROF 间隔定时器的间隔计数器的当前值和初始值（均以时钟滴答为单位）。当一个进程的 PROF 间隔定时器启动后，**则只要该进程处于运行中**，而不管是在用户态或核心态下执行，每个时钟滴答都使间隔计数器 it_prof_value 值减 1。当减到 0 值时，内核向进程发送 SIGPROF 信号，并将 it_prof_value 重置为初值 it_prof_incr。具体请见 7.4.3 节的 do_it_prof()函数。

Linux 在 include/linux/time.h 头文件中为上述三种进程间隔定时器定义了索引标识，如下所示：

```
#define    ITIMER_REAL     0
#define    ITIMER_VIRTUAL      1
#define    ITIMER_PROF     2
```

7．7．1 数据结构 itimerval

虽然，在内核中间隔定时器的间隔计数器是以时钟滴答次数为单位，但是让用户以时钟滴答为单位来指定间隔定时器的间隔计数器的初值显然是不太方便的，因为用户习惯的时间单位是秒、毫秒或微秒等。所以 Linux 定义了数据结构 itimerval 来让用户以秒或微秒为单位指定间隔定时器的时间间隔值。其定义如下（include/linux/time.h）：

```
struct      itimerval {
    struct      timeval it_interval;  /* timer interval */
    struct      timeval it_value;     /* current value */
};
```

其中，it_interval 成员表示间隔计数器的初始值，而 it_value 成员表示间隔计数器的当前值。这两个成员都是 timeval 结构类型的变量，因此其精度可以达到微秒级。

- **timeval 与 jiffies 之间的相互转换**

由于间隔定时器的间隔计数器的内部表示方式与外部表现方式互不相同，因此有必要实现以微秒为单位的 timeval 结构和为时钟滴答次数单位的 jiffies 之间的相互转换。为此，Linux 在 kernel/itimer.c 中实现了两个函数实现二者的互相转换——tvtojiffies()函数和 jiffiestotv()函数。它们的源码如下：

```
static unsigned long tvtojiffies(struct timeval *value)
{
    unsigned long sec = (unsigned) value->tv_sec;
    unsigned long usec = (unsigned) value->tv_usec;

    if (sec > (ULONG_MAX / HZ))
        return ULONG_MAX;
    usec += 1000000 / HZ - 1;
    usec /= 1000000 / HZ;
    return HZ*sec+usec;
}

static void jiffiestotv(unsigned long jiffies, struct timeval *value)
{
    value->tv_usec = (jiffies % HZ) * (1000000 / HZ);
    value->tv_sec = jiffies / HZ;
}
```

7．7．2 真实间隔定时器 ITIMER_REAL 的底层运行机制

间隔定时器 ITIMER_VIRT 和 ITIMER_PROF 的底层运行机制是分别通过函数 do_it_virt（）函数和 do_it_prof（）函数来实现的，这里就不再重述（可以参见 7.4.3 节）。

由于间隔定时器 ITIMER_REAL 本质上与内核动态定时器并无区别。因此内核实际上是通过内核动态定时器来实现进程的 ITIMER_REAL 间隔定时器的。为此，task_struct 结构中专门设立一个 timer_list 结构类型的成员变量 real_timer。动态定时器 real_timer 的函数指针 function 总是被 task_struct 结构的初始化宏 INIT_TASK 设置为指向函数 it_real_fn()。如下所示（include/linux/sched.h）：

```
#define    INIT_TASK(tsk)    \
```

```
        ……
        real_timer：  {
            function：  it_real_fn \
        } \
        ……
    }
```

而 real_timer 链表元素 list 和 data 成员总是被进程创建时分别初始化为空和进程 task_struct 结构的地址，如下所示（kernel/fork.c）：

```
    int do_fork(……)
    {
        ……
        p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
        p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
        init_timer(&p->real_timer);
        p->real_timer.data = (unsigned long)p;
        ……
    }
```

当用户通过 setitimer()系统调用来设置进程的 ITIMER_REAL 间隔定时器时，it_real_incr 被设置成非零值，于是该系统调用相应地设置好 real_timer.expires 值，然后进程的 real_timer 定时器就被加入到内核动态定时器链表中，这样该进程的 ITIMER_REAL 间隔定时器就被启动了。当 real_timer 定时器到期时，它的关联函数 it_real_fn()将被执行。注意！所有进程的 real_timer 定时器的 function 函数指针都指向 it_real_fn()这同一个函数，因此 it_real_fn()函数必须通过其参数来识别是哪一个进程，为此它将 unsigned long 类型的参数 p 解释为进程 task_struct 结构的地址。该函数的源码如下（kernel/itimer.c）：

```
    void it_real_fn(unsigned long __data)
    {
        struct task_struct * p = (struct task_struct *) __data;
        unsigned long interval;

        send_sig(SIGALRM, p, 1);
        interval = p->it_real_incr;
        if (interval) {
            if (interval > (unsigned long) LONG_MAX)
                interval = LONG_MAX;
            p->real_timer.expires = jiffies + interval;
            add_timer(&p->real_timer);
        }
    }
```

函数 it_real_fn()的执行过程大致如下：

（1）首先将参数 p 通过强制类型转换解释为进程的 task_struct 结构类型的指针。

（2）向进程发送 SIGALRM 信号。

（3）在进程的 it_real_incr 非 0 的情况下继续启动 real_timer 定时器。首先，计算 real_timer 定时器的 expires 值为（jiffies＋it_real_incr）。然后，调用 add_timer()函数将 real_timer 加入到内核动态定时器链表中。

7．7．3 itimer 定时器的系统调用

与 itimer 定时器相关的 syscall 有两个：getitimer()和 setitimer()。其中，getitimer()用于查询调用进程的三个间隔定时器的信息，而 setitimer()则用来设置调用进程的三个间隔定时器。这两个 syscall 都是现在 kernel/itimer.c 文件中。

7．7．3．1 getitimer()系统调用的实现

函数 sys_getitimer()有两个参数：（1）which，指定查询调用进程的哪一个间隔定时器，其取值可以是 ITIMER_REAL、ITIMER_VIRT 和 ITIMER_PROF 三者之一。(2)value 指针，指向用户空间中的一个 itimerval 结构，用于接收查询结果。该函数的源码如下：

```
/* SMP: Only we modify our itimer values. */
asmlinkage long sys_getitimer(int which, struct itimerval *value)
{
    int error = -EFAULT;
    struct itimerval get_buffer;

    if (value) {
        error = do_getitimer(which, &get_buffer);
        if (!error &&
            copy_to_user(value, &get_buffer, sizeof(get_buffer)))
            error = -EFAULT;
    }
    return error;
}
```

显然，sys_getitimer()函数主要通过 do_getitimer()函数来查询当前进程的间隔定时器信息，并将查询结果保存在内核空间的结构变量 get_buffer 中。然后，调用 copy_to_usr()宏将 get_buffer 中结果拷贝到用户空间缓冲区中。

函数 do_getitimer()的源码如下（kernel/itimer.c）：

```
int do_getitimer(int which, struct itimerval *value)
{
    register unsigned long val, interval;

    switch (which) {
    case ITIMER_REAL:
        interval = current->it_real_incr;
        val = 0;
        /*
         * FIXME! This needs to be atomic, in case the kernel timer happens!
         */
        if (timer_pending(&current->real_timer)) {
            val = current->real_timer.expires - jiffies;

            /* look out for negative/zero itimer.. */
```

```
                if ((long) val <= 0)
                    val = 1;
            }
            break;
        case ITIMER_VIRTUAL:
            val = current->it_virt_value;
            interval = current->it_virt_incr;
            break;
        case ITIMER_PROF:
            val = current->it_prof_value;
            interval = current->it_prof_incr;
            break;
        default:
            return(-EINVAL);
        }
        jiffiestotv(val, &value->it_value);
        jiffiestotv(interval, &value->it_interval);
        return 0;
    }
```

查询的过程如下：

（1）首先，用局部变量 val 和 interval 分别表示待查询间隔定时器的间隔计数器的当前值和初始值。

（2）如果 which＝ITIMER_REAL，则查询当前进程的 ITIMER_REAL 间隔定时器。于是从 current->it_real_incr 中得到 ITIMER_REAL 间隔定时器的间隔计数器的初始值，并将其保存到 interval 局部变量中。而对于间隔计数器的当前值，由于 ITITMER_REAL 间隔定时器是通过 real_timer 这个内核动态定时器来实现的，因此不能通过 current->it_real_value 来获得 ITIMER_REAL 间隔定时器的间隔计数器的当前值，而必须通过 real_timer 来得到这个值。为此先用 timer_pending()函数来判断 current->real_timer 是否已被起动。如果未启动，则说明 ITIMER_REAL 间隔定时器也未启动，因此其间隔计数器的当前值肯定是 0。因此将 val 变量简单地置 0 就可以了。如果已经启动，则间隔计数器的当前值应该等于（timer_real.expires －jiffies）。

（3）如果 which＝ITIMER_VIRT，则查询当前进程的 ITIMER_VIRT 间隔定时器。于是简单地将计数器初值 it_virt_incr 和当前值 it_virt_value 分别保存到局部变量 interval 和 val 中。

（4）如果 which＝ITIMER_PROF，则查询当前进程的 ITIMER_PROF 间隔定时器。于是简单地将计数器初值 it_prof_incr 和当前值 it_prof_value 分别保存到局部变量 interval 和 val 中。

（5）最后，通过转换函数 jiffiestotv()将 val 和 interval 转换成 timeval 格式的时间值，并保存到 value->it_value 和 value->it_interval 中，作为查询结果返回。

7．7．3．2 setitimer()系统调用的实现

函数 sys_setitimer()不仅设置调用进程的指定间隔定时器，而且还返回该间隔定时器的原有信息。它有三个参数：（1）which，含义与 sys_getitimer()中的参数相同。（2）输入参数 value，指向用户空间中的一个 itimerval 结构，含有待设置的新值。（3）输出参数 ovalue，指向用户空间中的一个 itimerval 结构，用于接收间隔定时器的原有信息。

该函数的源码如下（kernel/itimer.c）：

```
/* SMP: Again, only we play with our itimers, and signals are SMP safe
```

```
 *          now so that is not an issue at all anymore.
 */
asmlinkage long sys_setitimer(int which, struct itimerval *value,
                              struct itimerval *ovalue)
{
    struct itimerval set_buffer, get_buffer;
    int error;

    if (value) {
        if(copy_from_user(&set_buffer, value, sizeof(set_buffer)))
            return -EFAULT;
    } else
        memset((char *) &set_buffer, 0, sizeof(set_buffer));

    error = do_setitimer(which, &set_buffer, ovalue ? &get_buffer : 0);
    if (error || !ovalue)
        return error;

    if (copy_to_user(ovalue, &get_buffer, sizeof(get_buffer)))
        return -EFAULT;
    return 0;
}
```

对该函数的注释如下：

（1）在输入参数指针 value 非空的情况下，调用 copy_from_user()宏将用户空间中的待设置信息拷贝到内核空间中的 set_buffer 结构变量中。如果 value 指针为空，则简单地将 set_buffer 结构变量全部置 0。

（2）调用 do_setitimer()函数完成实际的设置操作。如果输出参数 ovalue 指针有效，则以内核变量 get_buffer 的地址作为 do_setitimer()函数的第三那个调用参数，这样当 do_setitimer()函数返回时，get_buffer 结构变量中就将含有当前进程的指定间隔定时器的原来信息。Do_setitimer()函数返回 0 值表示成功，非 0 值表示失败。

（3）在 do_setitimer()函数返回非 0 值的情况下，或者 ovalue 指针为空的情况下（不需要输出间隔定时器的原有信息），函数就可以直接返回了。

（4）如果 ovalue 指针非空，调用 copy_to_user()宏将 get_buffer()结构变量中值拷贝到 ovalue 所指向的用户空间中去，以便让用户得到指定间隔定时器的原有信息值。

函数 do_setitimer()的源码如下（kernel/itimer.c）：

```
int do_setitimer(int which, struct itimerval *value, struct itimerval *ovalue)
{
    register unsigned long i, j;
    int k;

    i = tvtojiffies(&value->it_interval);
    j = tvtojiffies(&value->it_value);
    if (ovalue && (k = do_getitimer(which, ovalue)) < 0)
        return k;
    switch (which) {
        case ITIMER_REAL:
```

```
                del_timer_sync(&current->real_timer);
                current->it_real_value = j;
                current->it_real_incr = i;
                if (!j)
                    break;
                if (j > (unsigned long) LONG_MAX)
                    j = LONG_MAX;
                i = j + jiffies;
                current->real_timer.expires = i;
                add_timer(&current->real_timer);
                break;
            case ITIMER_VIRTUAL:
                if (j)
                    j++;
                current->it_virt_value = j;
                current->it_virt_incr = i;
                break;
            case ITIMER_PROF:
                if (j)
                    j++;
                current->it_prof_value = j;
                current->it_prof_incr = i;
                break;
            default:
                return -EINVAL;
        }
        return 0;
    }
```

对该函数的注释如下：

（1）首先调用 tvtojiffies（）函数将 timeval 格式的初始值和当前值转换成以时钟滴答为单位的时间值。并分别保存在局部变量 i 和 j 中。

（2）如果 ovalue 指针非空，则调用 do_getitimer()函数查询指定间隔定时器的原来信息。如果 do_getitimer()函数返回负值，说明出错。因此就要直接返回错误值。否则继续向下执行开始真正地设置指定的间隔定时器。

（3）如果 which=ITITMER_REAL，表示设置 ITIMER_REAL 间隔定时器。（a）调用 del_timer_sync()函数（该函数在单 CPU 系统中就是 del_timer()函数）将当前进程的 real_timer 定时器从内核动态定时器链表中删除。(b)将 it_real_incr 和 it_real_value 分别设置为局部变量 i 和 j。(c)如果 j=0，说明不必启动 real_timer 定时器，因此执行 break 语句退出 switch…case 控制结构，而直接返回。（d）将 real_timer 的 expires 成员设置成（jiffies＋当前值 j），然后调用 add_timer()函数将当前进程的 real_timer 定时器加入到内核动态定时器链表中，从而启动该定时器。

（4）如果 which=ITIMER_VIRT，则简单地用局部变量 i 和 j 的值分别更新 it_virt_incr 和 it_virt_value 就可以了。

（5）如果 which=ITIMER_PROF，则简单地用局部变量 i 和 j 的值分别更新 it_prof_incr 和 it_prof_value 就可以了。

（6）最后，返回 0 值表示成功。

7．7．3．3 alarm 系统调用

系统调用 alarm 可以让调用进程在指定的秒数间隔后收到一个 SIGALRM 信号。它只有一个参数 seconds，指定以秒数计的定时间隔。函数 sys_alarm()的源码如下（kernel/timer.c）：

```
/*
 * For backwards compatibility?  This can be done in libc so Alpha
 * and all newer ports shouldn't need it.
 */
asmlinkage unsigned long sys_alarm(unsigned int seconds)
{
    struct itimerval it_new, it_old;
    unsigned int oldalarm;

    it_new.it_interval.tv_sec = it_new.it_interval.tv_usec = 0;
    it_new.it_value.tv_sec = seconds;
    it_new.it_value.tv_usec = 0;
    do_setitimer(ITIMER_REAL, &it_new, &it_old);
    oldalarm = it_old.it_value.tv_sec;
    /* ehhh.. We can't return 0 if we have an alarm pending.. */
    /* And we'd better return too much than too little anyway */
    if (it_old.it_value.tv_usec)
        oldalarm++;
    return oldalarm;
}
```

这个系统调用实际上就是启动进程的 ITIMER_REAL 间隔定时器。因此它完全可放到用户空间的 C 函数库（比如 libc 和 glibc）中来实现。但是为了保此内核的向后兼容性，2.4.0 版的内核仍然将这个 syscall 放在内核空间中来实现。函数 sys_alarm（）的实现过程如下：

（1)根据参数 seconds 的值构造一个 itimerval 结构变量 it_new。注意!由于 alarm 启动的 ITIMER_REAL 间隔定时器是一次性而不是循环重复的，因此 it_new 变量中的 it_interval 成员一定要设置为 0。

（2）调用函数 do_setitimer()函数以新构造的定时器 it_new 来启动当前进程的 ITIMER_REAL 定时器，同时将该间隔定时器的原定时间隔保存到局部变量 it_old 中。

（3）返回值 oldalarm 表示以秒数计的 ITIMER_REAL 间隔定时器的原定时间隔值。因此先把 it_old.it_value.tv_sec 赋给 oldalarm，并且在 it_old.it_value.tv_usec 非 0 的情况下，将 oldalarm 的值加 1（也即不足 1 秒补足 1 秒）。

7．8 时间系统调用的实现

本节讲述与时间相关的 syscall，这些系统调用主要用来供用户进程向内核检索当前时间与日期，因此他们是内核的时间服务接口。主要的时间系统调用共有 5 个：time、stime 和 gettimeofday、settimeofday，以及与网络时间协议 NTP 相关的 adjtimex 系统调用。这里我们不关心 NTP，因此仅分析前 4 个时间系统调用。前 4 个时间系统调用可以分为两组：（1）time 和 stime 是一组；（2）gettimeofday 和 settimeofday 是一组。

7．8．1 系统调用 time 和 stime

系统调用 time（）用于获取以秒数表示的系统当前时间（即内核全局时间变量 xtime 中的 tv_sec 成员的值）。它只有一个参数——整型指针 tloc，指向用户空间中的一个整数，用来接收返回的当前时间值。函数 sys_time（）的源码如下（kernel/time.c）：

```
asmlinkage long sys_time(int * tloc)
{
    int i;

    /* SMP: This is fairly trivial. We grab CURRENT_TIME and
       stuff it to user space. No side effects */
    i = CURRENT_TIME;
    if (tloc) {
        if (put_user(i,tloc))
            i = -EFAULT;
    }
    return i;
}
```

注释如下：

（1）首先，函数调用 CURRENT_TIME 宏来得到以秒数表示的内核当前时间值，并将该值保存在局部变量 i 中。宏 CURRENT_TIME 定义在 include/linux/sched.h 头文件中，它实际上就是内核全局时间变量 xtime 中的 tv_sec 成员。如下所示：

```
#define CURRENT_TIME    (xtime.tv_sec)
```

（2）然后，在参数指针 tloc 非空的情况下将 i 的值通过 put_user()宏传递到有 tloc 所指向的用户空间中去，以作为函数的输出结果。

（3）最后，将局部变量 I 的值——也即也秒数表示的系统当前时间值作为返回值返回。

系统调用 stime()与系统调用 time()刚好相反，它可以让用户设置系统的当前时间（以秒数为单位）。它同样也只有一个参数——整型指针 tptr，指向用户空间中待设置的时间秒数值。函数 sys_stime()的源码如下（kernel/time.c）：

```
asmlinkage long sys_stime(int * tptr)
{
    int value;

    if (!capable(CAP_SYS_TIME))
        return -EPERM;
    if (get_user(value, tptr))
        return -EFAULT;
    write_lock_irq(&xtime_lock);
    xtime.tv_sec = value;
    xtime.tv_usec = 0;
    time_adjust = 0;    /* stop active adjtime() */
    time_status |= STA_UNSYNC;
    time_maxerror = NTP_PHASE_LIMIT;
    time_esterror = NTP_PHASE_LIMIT;
```

```
        write_unlock_irq(&xtime_lock);
        return 0;
    }
```

注释如下：

（1）首先检查调用进程的权限，显然，只有 root 用户才能有权限修改系统时间。

（2）调用 get_user()宏将 tptr 指针所指向的用户空间中的时间秒数值拷贝到内核空间中来，并保存到局部变量 value 中。

（3）将局部变量 value 的值更新到全局时间变量 xtime 的 tv_sec 成员中，并将 xtime 的 tv_usec 成员清零。

（4）在相应地重置其它状态变量后，函数就可以返回了（返回值 0 表示成功）。

7．8．2 系统调用 gettimeofday

这个 syscall 用来供用户获取 timeval 格式的当前时间信息（精确度为微秒级），以及系统的当前时区信息（timezone）。结构类型 timeval 的指针参数 tv 指向接受时间信息的用户空间缓冲区，参数 tz 是一个 timezone 结构类型的指针，指向接收时区信息的用户空间缓冲区。这两个参数均为输出参数，返回值 0 表示成功，返回负值表示出错。函数 sys_gettimeofday()的源码如下（kernel/time.c）：

```
asmlinkage long sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    if (tv) {
        struct timeval ktv;
        do_gettimeofday(&ktv);
        if (copy_to_user(tv, &ktv, sizeof(ktv)))
            return -EFAULT;
    }
    if (tz) {
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
    }
    return 0;
}
```

显然，函数的实现主要分成两个大的方面：

（1）如果 tv 指针有效，则说明用户要以 timeval 格式来检索系统当前时间。为此，先调用 do_gettimeofday()函数来检索系统当前时间并保存到局部变量 ktv 中。然后再调用 copy_to_user（）宏将保存在内核空间中的当前时间信息拷贝到由参数指针 tv 所指向的用户空间缓冲区中。

（2）如果 tz 指针有效，则说明用户要检索当前时区信息，因此调用 copy_to_user()宏将全局变量 sys_tz 中的时区信息拷贝到参数指针 tz 所指向的用户空间缓冲区中。

（3）最后，返回 0 表示成功。

函数 do_gettimeofday()的源码如下（arch/i386/kernel/time.c）：

```
/*
 * This version of gettimeofday has microsecond resolution
 * and better than microsecond precision on fast x86 machines with TSC.
 */
```

```
void do_gettimeofday(struct timeval *tv)
{
    unsigned long flags;
    unsigned long usec, sec;

    read_lock_irqsave(&xtime_lock, flags);
    usec = do_gettimeoffset();
    {
        unsigned long lost = jiffies - wall_jiffies;
        if (lost)
            usec += lost * (1000000 / HZ);
    }
    sec = xtime.tv_sec;
    usec += xtime.tv_usec;
    read_unlock_irqrestore(&xtime_lock, flags);

    while (usec >= 1000000) {
        usec -= 1000000;
        sec++;
    }

    tv->tv_sec = sec;
    tv->tv_usec = usec;
}
```

该函数的完成实际的当前时间检索工作。由于 gettimeofday()系统调用要求时间精度要达到微秒级，因此 do_gettimeofday()函数不能简单地返回 xtime 中的值即可，而必须精确地确定自从时钟驱动的 Bottom Half 上一次更新 xtime 的那个时刻（由 wall_jiffies 变量表示，参见 7.3 节）到 do_gettimeofday()函数的当前执行时刻之间的具体时间间隔长度,以便精确地修正 xtime 的值.如下图 7-9 所示：

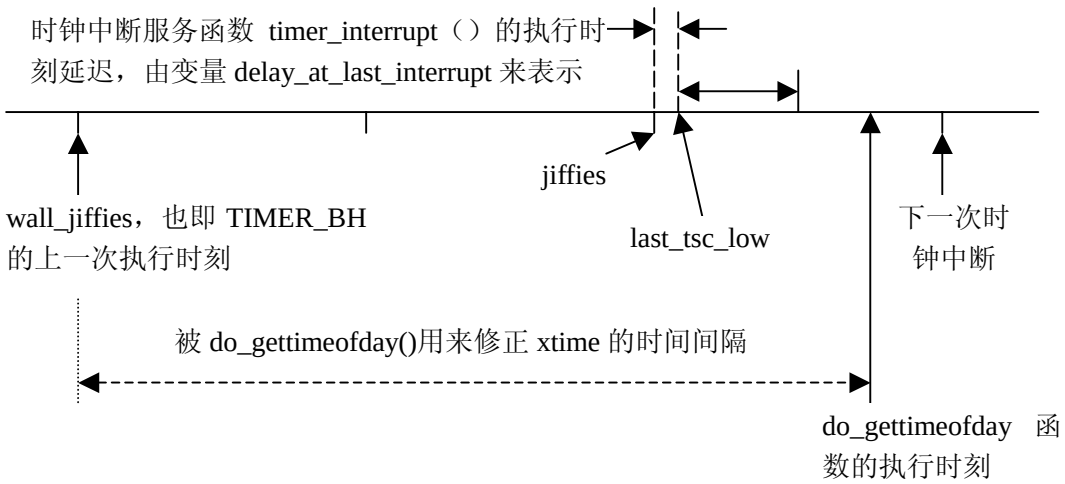


图 7－9 时钟中断过程中的时间关系

假定被 do_gettimeofday()用来修正 xtime 的时间间隔为 fixed_usec，而从 wall_jiffies 到 jiffies 之间的时间间隔是 lost_usec，而从 jiffies 到 do_gettimeofday()函数的执行时刻的时间间隔是 offset_usec。则下列三个等式成立：

fixed_usec＝（lost_usec＋offset_usec）

lost_usec＝（jiffies－wall_jiffies）＊TICK_SIZE＝（jiffies－wall_jiffies）＊（1000000／HZ）

由于全局变量 last_tsc_low 表示上一次时钟中断服务函数 timer_interrupt()执行时刻的 CPU TSC 寄存器的值，因此我们可以用 X86 CPU 的 TSC 寄存器来计算 offset_usec 的值。也即：

offset_usec=delay_at_last_interrupt＋（current_tsc_low－last_tsc_low）＊fast_gettimeoffset_quotient

其中，delay_at_last_interrupt 是从上一次发生时钟中断到 timer_interrupt()服务函数真正执行时刻之间的时间延迟间隔。每一次 timer_interrupt()被执行时都会计算这一间隔，并利用 TSC 的当前值更新 last_tsc_low 变量（可以参见 7.4 节）。假定 current_tsc_low 是 do_gettimeofday()函数执行时刻 TSC 的当前值，全局变量 fast_gettimeoffset_quotient 则表示 TSC 寄存器每增加 1 所代表的时间间隔值，它是由 time_init()函数所计算的。

根据上述原理分析，do_gettimeofday()函数的执行步骤如下：

（1）调用函数 do_gettimeoffset()计算从上一次时钟中断发生到执行 do_gettimeofday()函数的当前时刻之间的时间间隔 offset_usec。

（2）通过 wall_jiffies 和 jiffies 计算 lost_usec 的值。

（3）然后，令 sec=xtime.tv_sec，usec=xtime.tv_usec+lost_usec+offset_usec。显然，sec 表示系统当前时间在秒数量级上的值，而 usec 表示系统当前时间在微秒量级上的值。

（4）用一个 while{}循环来判断 usec 是否已经溢出而超过 10^6us＝1 秒。如果溢出，则将 usec 减去 10^6us 并相应地将 sec 增加 1，直到 usec 不溢出为止。

（5）最后，用 sec 和 usec 分别更新参数指针所指向的 timeval 结构变量。至此，整个查询过程结束。

函数 do_gettimeoffset()根据 CPU 是否配置有 TSC 寄存器这一条件分别有不同的实现。其定义如下（arch/i386/kernel/time.c）：

```
#ifndef    CONFIG_X86_TSC
static unsigned long do_slow_gettimeoffset(void)
{
      ……
}
static unsigned long (*do_gettimeoffset)(void)    = do_slow_gettimeoffset;
#else
#define do_gettimeoffset()      do_fast_gettimeoffset()
#endif
```

显然，在配置有 TSC 寄存器的 i386 平台上，do_gettimeoffset（）函数实际上就是 do_fast_gettimeoffset()函数。它通过 TSC 寄存器来计算 do_fast_gettimeoffset()函数被执行的时刻到上一次时钟中断发生时的时间间隔值。其源码如下（arch/i386/kernel/time.c）：

```
static inline unsigned long do_fast_gettimeoffset(void)
{
      register unsigned long eax, edx;

      /* Read the Time Stamp Counter */

      rdtsc(eax,edx);

      /* .. relative to previous jiffy (32 bits is enough) */
      eax -= last_tsc_low;       /* tsc_low delta */

      /*
            * Time offset = (tsc_low delta) * fast_gettimeoffset_quotient
```

```
                *                    = (tsc_low delta) * (usecs_per_clock)
                *                    = (tsc_low delta) * (usecs_per_jiffy / clocks_per_jiffy)
         *
         * Using a mull instead of a divl saves up to 31 clock cycles
         * in the critical path.
                */

        __asm__("mull %2"
                :"=a" (eax), "=d" (edx)
                :"rm" (fast_gettimeoffset_quotient),
                 "0" (eax));

        /* our adjusted time offset in microseconds */
        return delay_at_last_interrupt + edx;
    }
```

对该函数的注释如下：

（1）先调用 rdtsc()函数读取当前时刻 TSC 寄存器的值，并将其高 32 位保存在 edx 局部变量中，低 32 位保存在局部变量 eax 中。

（2）让局部变量 eax＝Δ_{tsc_low}＝eax－last_tsc_low；也即计算当前时刻的 TSC 值与上一次时钟中断服务函数 timer_interrupt()执行时的 TSC 值之间的差值。

（3）显然，从上一次 timer_interrupt() 到当前时刻的时间间隔就是（Δ_{tsc_low} * fast_gettimeoffset_quotient）。因此用一条 mul 指令来计算这个乘法表达式的值。

（4）返回值 delay_at_last_interrupt＋（Δ_{tsc_low} * fast_gettimeoffset_quotient）就是从上一次时钟中断发生时到当前时刻之间的时间偏移间隔值。

7．8．3 系统调用 settimeofday

这个系统调用与 gettimeofday（）刚好相反，它供用户设置当前时间以及当前时间信息。它也有两个参数：（1）参数指针 tv，指向含有待设置时间信息的用户空间缓冲区；（2）参数指针 tz，指向含有待设置时区信息的用户空间缓冲区。函数 sys_settimeofday()的源码如下（kernel/time.c）：

```
asmlinkage long sys_settimeofday(struct timeval *tv, struct timezone *tz)
{
    struct timeval  new_tv;
    struct timezone new_tz;

    if (tv) {
        if (copy_from_user(&new_tv, tv, sizeof(*tv)))
            return -EFAULT;
    }
    if (tz) {
        if (copy_from_user(&new_tz, tz, sizeof(*tz)))
            return -EFAULT;
    }
```

```
    return do_sys_settimeofday(tv ? &new_tv : NULL, tz ? &new_tz : NULL);
}
```

函数首先调用 copy_from_user（）宏将保存在用户空间中的待设置时间信息和时区信息拷贝到内核空间中来，并保存到局部变量 new_tv 和 new_tz 中。然后，调用 do_sys_settimeofday()函数完成实际的时间设置和时区设置操作。

函数 do_sys_settimeofday()的源码如下（kernel/time.c）：

```
int do_sys_settimeofday(struct timeval *tv, struct timezone *tz)
{
    static int firsttime = 1;

    if (!capable(CAP_SYS_TIME))
        return -EPERM;

    if (tz) {
        /* SMP safe, global irq locking makes it work. */
        sys_tz = *tz;
        if (firsttime) {
            firsttime = 0;
            if (!tv)
                warp_clock();
        }
    }
    if (tv)
    {
        /* SMP safe, again the code in arch/foo/time.c should
         * globally block out interrupts when it runs.
         */
        do_settimeofday(tv);
    }
    return 0;
}
```

该函数的执行过程如下：

（1）首先，检查调用进程是否有相应的权限。如果没有，则返回错误值－EPERM。

（2）如果执政 tz 有效，则用 tz 所指向的新时区信息更新全局变量 sys_tz。并且如果是第一次设置时区信息，则在 tv 指针不为空的情况下调用 wrap_clock()函数来调整 xtime 中的秒数值。函数 wrap_clock()的源码如下（kernel/time.c）：

```
inline static void warp_clock(void)
{
    write_lock_irq(&xtime_lock);
    xtime.tv_sec += sys_tz.tz_minuteswest * 60;
    write_unlock_irq(&xtime_lock);
}
```

（3）如果参数 tv 指针有效，则根据 tv 所指向的新时间信息调用 do_settimeofday()函数来更新内核的当前时间 xtime。

（4）最后，返回 0 值表示成功。

函数 do_settimeofday()执行刚好与 do_gettimeofday()相反的操作。这是因为全局变量 xtime 所表示的时

间是与 wall_jiffies 相对应的那一个时刻。因此，必须从参数指针 tv 所指向的新时间中减去时间间隔 fixed_usec（其含义见 7.8.2 节）。函数源码如下（arch/i386/kernel/time.c）：

```
void do_settimeofday(struct timeval *tv)
{
    write_lock_irq(&xtime_lock);
    /*
     * This is revolting. We need to set "xtime" correctly. However, the
     * value in this location is the value at the most recent update of
     * wall time.    Discover what correction gettimeofday() would have
     * made, and then undo it!
     */
    tv->tv_usec -= do_gettimeoffset();
    tv->tv_usec -= (jiffies - wall_jiffies) * (1000000 / HZ);

    while (tv->tv_usec < 0) {
        tv->tv_usec += 1000000;
        tv->tv_sec--;
    }

    xtime = *tv;
    time_adjust = 0;            /* stop active adjtime() */
    time_status |= STA_UNSYNC;
    time_maxerror = NTP_PHASE_LIMIT;
    time_esterror = NTP_PHASE_LIMIT;
    write_unlock_irq(&xtime_lock);
}
```

该函数的执行步骤如下：

（1）调用 do_gettimeoffset()函数计算上一次时钟中断发生时刻到当前时刻之间的时间间隔值。

（2）通过 wall_jiffies 与 jiffies 计算二者之间的时间间隔 lost_usec。

（3）从 tv->tv_usec 中减去 fixed_usec，即：tv->tv_usec－＝（lost_usec＋offset_usec）。

（4）用一个 while{}循环根据 tv->tv_usec 是否小于 0 来调整 tv 结构变量。如果 tv->tv_usec 小于 0，则将 tv->tv_usec 加上 10^6us，并相应地将 tv->tv_sec 减 1。直到 tv->tv_usec 不小于 0 为止。

（5）用修正后的时间 tv 来更新内核全局时间变量 xtime。

（6）最后，重置其它时间状态变量。

至此，我们已经完全分析了整个 Linux 内核的时钟机制！