第五章　进程调度与切换

在多进程的操作系统中，进程调度是一个全局性、关键性的问题，它对系统的总体设计、系统的的实现、功能设置以及各个方面的性能都有着决定性的影响。根据调度的结果所作的进程切换的速度，也是衡量一个操作系统性能的重要指标。进程调度算法的设计，还对系统的复杂性有着极大的影响，常常会由于实现的复杂程度而在功能与性能方面作出必要的权衡和让步。

进程调度与时间的关系非常密切，因此，本章首先讨论与时间相关的主题，然后才讨论进程的调度，最后介绍了 Linux 中进程是如何进行切换的。

5.1　Linux 时间系统

计算机是以严格精确的时间进行数值运算和和数据处理的，最基本的时间单元是时钟周期，例如取指令、执行指令、存取内存等，但是我们不讨论这些纯硬件的东西，这里要讨论的是操作系统建立的时间系统，这个时间系统是整个操作系统活动的动力。

时间系统是计算机系统非常重要的组成部分，特别是对于 UNIX 类分时系统尤为重要。时间系统通常又被简称为时钟，它的主要任务是维持系统时间并且防止某个进程独占 CPU 及其他资源，也就是驱动进程的调度。本节将详细讲述时钟的来源、在 Linux 中的实现及其重要作用，使读者消除对时钟的神秘感。

5.1.1　时钟硬件

大部分 PC 机中有两个时钟源，他们分别叫做 RTC 和 OS（操作系统）时钟。RTC（Real Time Clock，实时时钟）也叫做 CMOS 时钟，它是 PC 主机板上的一块芯片（或者叫做时钟电路），它靠电池供电，即使系统断电，也可以维持日期和时间。由于它独立于操作系统，所以也被称为硬件时钟，它为整个计算机提供一个计时标准，是最原始最底层的时钟数据。

Linux 只用 RTC 来获得时间和日期，同时，通过作用于/dev/rtc 设备文件，也允许进程对 RTC 编程。内核通过 0x70 和 0x71 I/O 端口存取 RTC。通过执行/sbin/clock 系统程序（它直接作用于这两个 I/O 端口），系统管理员可以配置时钟。

OS 时钟产生于 PC 主板上的定时/计数芯片，由操作系统控制这个芯片的工作，OS 时钟的基本单位就是该芯片的计数周期。在开机时操作系统取得 RTC 中的时间数据来初始化 OS 时钟，然后通过计数芯片的向下计数形成了 OS 时钟，所以 OS 时钟并不是本质意义上的时钟，它更应该被称为一个计数器。OS 时钟只在开机时才有效，而且完全由操作系统控制，所以也

被称为软时钟或系统时钟。下面我们重点描述 OS 时钟的产生。

OS 时钟所用的定时/计数芯片最典型的是 8253/8254 可编程定时/计数芯片，其硬件结构及工作原理在这里不详细讲述，只简单地描述它是怎样维持 OS 时钟的。OS 时钟的物理产生示意图如图 5.1 所示。

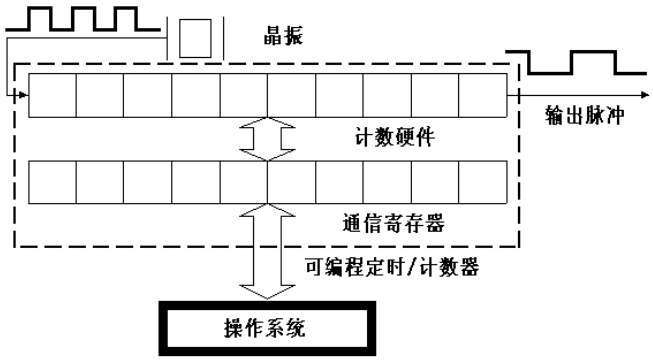


图 5.1　8253/8254 工作示意图

可编程定时/计数器总体上由两部分组成：计数硬件和通信寄存器。通信寄存器包含有控制寄存器、状态寄存器、计数初始值寄存器（16 位）、计数输出寄存器等。通信寄存器在计数硬件和操作系统之间建立联系，用于二者之间的通信，操作系统通过这些寄存器控制计数硬件的工作方式、读取计数硬件的当前状态和计数值等信息。在 Linux 内核初始化时，内核写入控制字和计数初值，这样计数硬件就会按照一定的计数方式对晶振产生的输入脉冲信号（5MHz~100MHz 的频率）进行计数操作：计数器从计数初值开始，每收到一次脉冲信号，计数器减 1，当计数器减至 0 时，就会输出高电平或低电平，然后，如果计数为循环方式（通常为循环计数方式），则重新从计数初值进行计数，从而产生如图 5.1 所示的输出脉冲信号（当然不一定是很规则的方波）。这个输出脉冲是 OS 时钟的硬件基础，之所以这么说，是因为这个输出脉冲将接到中断控制器上 ，产生中断信号，触发后面要讲的时钟中断，由时钟中断服务程序维持 OS 时钟的正常工作，所谓维持，其实就是简单的加 1 及细微的修正操作。这就是 OS 时钟产生的来源。

5.12　时钟运作机制

不同的操作系统，RTC 和 OS 时钟的关系是不同的。RTC 和 OS 时钟之间的关系通常也被称作操作系统的时钟运作机制。

一般来说，RTC 是 OS 时钟的时间基准，操作系统通过读取 RTC 来初始化 OS 时钟，此后二者保持同步运行，共同维持着系统时间。保持同步运行是什么意思呢？就是指操作系统运行过程中，每隔一个固定时间会刷新或校正 RTC 中的信息。

Linux 中的时钟运作机制如图 5.2 所示。OS 时钟和 RTC 之间要通过 BIOS 的连接，是因为传统 PC 机的 BIOS 中固化有对 RTC 进行有关操作的函数，例如 INT 1AH 等中断服务程序，

通常操作系统也直接利用这些函数对 RTC 进行操作,例如从 RTC 中读出有关数据对 OS 时钟初始化、对 RTC 进行更新等。实际上，不通过 BIOS 而直接对 RTC 的有关端口进行操作也是可以的。Linux 中在内核初始化完成后就完全抛弃了 BIOS 中的程序。

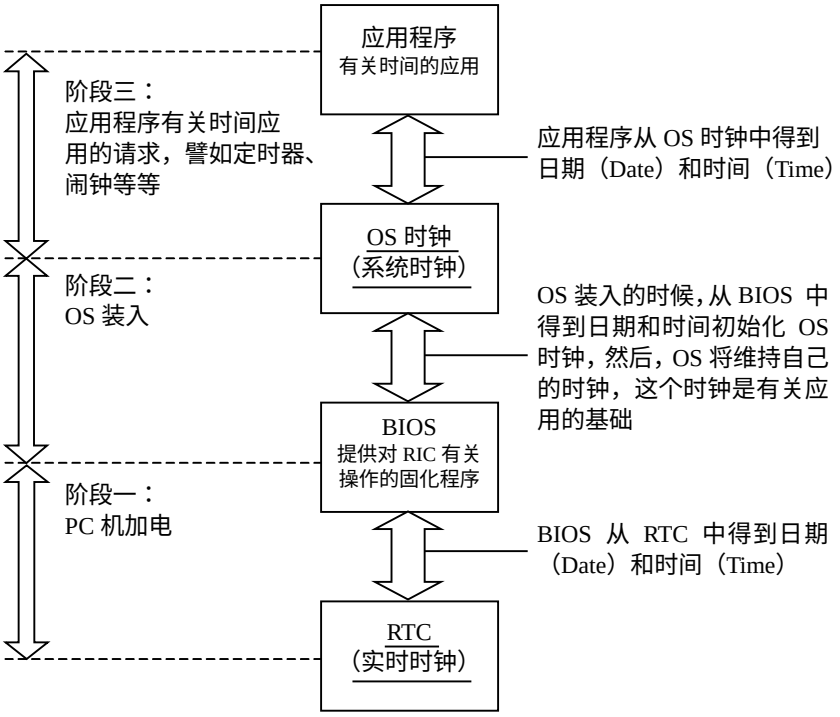


图 5.2　时钟运作机制

我们可以看到，RTC 处于最底层，提供最原始的时钟数据。OS 时钟建立在 RTC 之上，初始化完成后将完全由操作系统控制，和 RTC 脱离关系。操作系统通过 OS 时钟提供给应用程序所有和时间有关的服务。因为 OS 时钟完全是一个软件问题，其所能表达的时间由操作系统的设计者决定，将 OS 时钟定义为整型还是长整型或者大的超乎想像都是设计者的事。

5.1.3　Linux 时间基准

以上我们了解了 RTC（实时时钟、硬件时钟）和 OS 时钟（系统时钟、软时钟）。下面我们具体描述 OS 时钟。

我们知道，OS 时钟是由可编程定时/计数器产生的输出脉冲触发中断而产生的。输出脉冲的周期叫做一个“时钟滴答”，有些书上也把它叫做“时标”。计算机中的时间是以时钟滴答为单位的，每一次时钟滴答，系统时间就会加 1。操作系统根据当前时钟滴答的数目就可以得到以秒或毫秒等为单位的其他时间格式。

不同的操作系统采用不同的“时间基准”。定义“时间基准”的目的是为了简化计算，这样计算机中的时间只要表示为从这个时间基准开始的时钟滴答数就可以了。“时间基准”是由操作系统的设计者规定的。例如 DOS 的时间基准是 1980 年 1 月 1 日，UNIX 和 Minux 的

时间基准是 1970 年 1 月 1 日上午 12 点，Linux 的时间基准是 1970 年 1 月 1 日凌晨 0 点。

5.1.4 Linux 的时间系统

通过上面的时钟运作机制，我们知道了 OS 时钟在 Linux 中的重要地位。OS 时钟记录的时间也就是通常所说的系统时间。系统时间是以“时钟滴答”为单位的，而时钟中断的频率决定了一个时钟滴答的长短，例如每秒有 100 次时钟中断，那么一个时钟滴答的就是 10 毫秒（记为 10ms），相应地，系统时间就会每 10ms 增 1。不同的操作系统对时钟滴答的定义是不同的，例如 DOS 的时钟滴答为 1/18.2s，Minix 的时钟滴答为 1/60s 等。

Linux 中用全局变量 jiffies 表示系统自启动以来的时钟滴答数目。jiffy 是“瞬间、一会儿”的意思，和“时钟滴答”表达的是同一个意思。jiffies 是 jiffy 的复数形式，在 /kernel/time.c 中定义如下：

```
unsigned long volatile jiffies
```

在 jiffies 基础上，Linux 提供了如下适合人们习惯的时间格式，在 /include/linux/time.h 中定义如下：

```
struct timespec {                ／*  这是精度很高的表示*／
     long tv_sec;              ／* 秒 （second） *／
     long tv_nsec;             /* 纳秒：十亿分之一秒（ nanosecond）*／
};

struct timeval  {                ／* 普通精度   *／
     int  tv_sec;              /*  秒    *／
     int  tv_usec;             /* 微秒：百万分之一秒（microsecond）*／
};

struct timezone {                    /*  时区  *／
     int  tz_minuteswest;          ／*  格林尼治时间往西方的时差 *／
     int  tz_dsttime;                 /* 时间修正方式 *／
};
```

tv_sec 表示秒（second），tv_usec 表示微秒（microsecond，百万分之一秒即 10^{-6} 秒），tv_nsec 表示纳秒（nanosecond，十亿分之一秒即 10^{-9} 秒）。定义 tb_usec 和 tv_nsec 的目的是为了适用不同的使用要求，不同的场合根据对时间精度的要求选用这两种表示。

另外，Linux 还定义了用于表示更加符合大众习惯的时间表示：年、月、日。但是万变不离其宗，所有的时间应用都是建立在 jiffies 基础之上的，我们将详细讨论 jiffies 的产生和其作用。简而言之，jiffies 产生于时钟中断！

5.2　时钟中断

5.2.1　时钟中断的产生

前面我们看到，Linux 的 OS 时钟的物理产生原因是可编程定时/计数器产生的输出脉冲，这个脉冲送入 CPU，就可以引发一个中断请求信号，我们就把它叫做时钟中断。

“ 时钟中断 ”是特别重要的一个中断，因为整个操作系统的活动都受到它的激励。系统利用时钟中断维持系统时间、促使环境的切换，以保证所有进程共享 CPU；利用时钟中断进行记帐、监督系统工作以及确定未来的调度优先级等工作。可以说，“ 时钟中断 ”是整个操作系统的脉搏。

时钟中断的物理产生如图 5.3 所示。

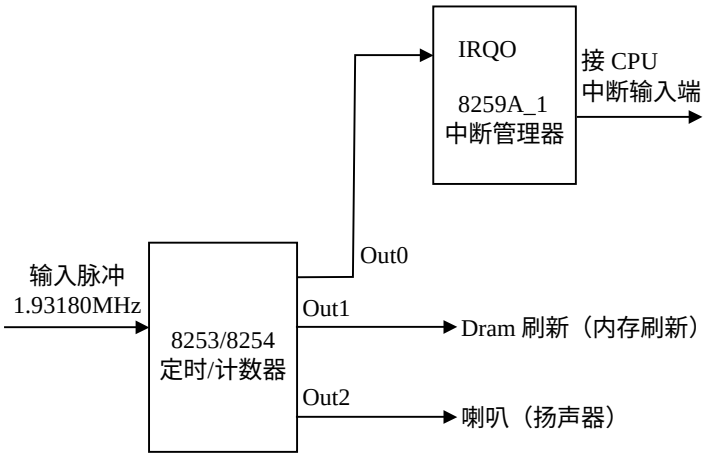


图 5.3　8253 和 8259A 的物理连接方式

操作系统对可编程定时/计数器进行有关初始化，然后定时/计数器就对输入脉冲进行计数（分频），产生的 3 个输出脉冲：Out0、Out1、Out2，它们各有用途，很多有关接口的书都介绍了这个问题，我们只介绍 Out0 上的输出脉冲，这个脉冲信号接到中断控制器 8259A_1 的 0 号管脚，触发一个周期性的中断，我们就把这个中断叫做时钟中断，时钟中断的周期，也就是脉冲信号的周期，我们叫做“ 滴答 ”或“ 时标 ”（tick）。从本质上说，时钟中断只是一个周期性的信号，完全是硬件行为，该信号触发 CPU 去执行一个中断服务程序，但是为了方便，我们就把这个服务程序叫做时钟中断，读者可能早就习惯这种叫法了，我们也不必把一些概念区分得那么详细。

5.2.2 Linux 实现时钟中断的全过程

1．可编程定时/计数器的初始化

IBM PC 中使用的是 8253 或 8254 芯片。有关该芯片的详细知识我们不再详述，只大体介绍以下它的组成和作用，如表 5.1 所示。

表 5.1　　8253/8254 的组成及作用

名称	端口地址	工作方式	产生的输出脉冲的用途
计数器 0	0x40	方式 3	时钟中断，也叫系统时钟

续表

名称	端口地址	工作方式	产生的输出脉冲的用途
计数器 1	0x41	方式 2	动态存储器刷新
计数器 2	0x42	方式 3	扬声器发声
控制寄存器	0x43	/	用于 8253 的初始化，接收控制字

计数器 0 的输出就是图 5.3 中的 Out0,它的频率由操作系统的设计者确定,Linux 对 8253 的初始化程序段如下（在/arch/i386/kernel/i8259.c 的 init_IRQ()函数中）：

```
set_intr_gate (ox20, interrupt[0]) ;
／*在 IDT 的第 0x20 个表项中插入一个中断门。这个门中的段选择符设置成内核代码段的选择符，偏移域设置成 0 号中断处理程序的入口地址。*/
outb_p (0x34,0x43) ;          /* 写计数器 0 的控制字：工作方式 2*/
outb_p (LATCH & 0xff , 0x40) ;   /* 写计数初值 LSB  计数初值低位字节*/
outb (LATCH >> 8 , 0x40) ;   /* 写计数初值 MSB 计数初值高位字节*/
```

LATCH（英文意思为：锁存器，即其中锁存了计数器 0 的初值）为计数器 0 的计数初值，在/include/linux/timex.h 中定义如下：

```
#define CLOCK_TICK_RATE 1193180     /* 图 5.3 中的输入脉冲 */
#define LATCH   ( (CLOCK_TICK_RATE + HZ/2)  / HZ)    /* 计数器 0 的计数初值 */
```

CLOCK_TICK_RATE 是整个 8253 的输入脉冲，如图 5.3 中所示为 1.193180MHz，是近似为 1MHz 的方波信号,8253 内部的 3 个计数器都对这个时钟进行计数,进而产生不同的输出信号，用于不同的用途。

HZ 表示计数器 0 的频率，也就是时钟中断或系统时钟的频率，在/include/asm/param.h 中定义如下:

```
#define HZ 100
```

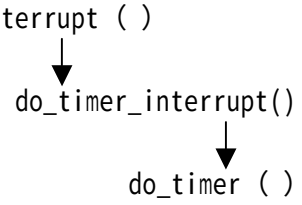
2．与时钟中断相关的函数

下面我们接着介绍时钟中断触发的服务程序，该程序代码比较复杂，分布在不同的源文件中，主要包括如下函数：

- 时钟中断程序：timer_interrupt ( ) ；

- 中断服务通用例程：do_timer_interrupt();
- 时钟函数：do_timer（）；
- 中断安装程序：setup_irq（）；
- 中断返回函数：ret_from_intr（）；

前 3 个函数的调用关系如下：



(1) timer_interrupt（）

这个函数大约每 10ms 被调用一次，实际上，timer_interrupt（）函数是一个封装例程，它真正做的事情并不多，但是，作为一个中断程序,它必须在关中断的情况下执行。如果只考虑单处理机的情况，该函数主要语句就是调用 do_timer_interrupt()函数。

(2) do_timer_interrupt()

do_timer_interrupt()函数有两个主要任务，一个是调用 do_timer（），另一个是维持实时时钟（RTC，每隔一定时间段要回写），其实现代码在/arch/i386/kernel/time.c 中，为了突出主题，笔者对以下函数作了改写，以便于读者理解：

```
static inline void do_timer_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
    do_timer (regs) ; /* 调用时钟函数，将时钟函数等同于时钟中断未尝不可*/

    if (xtime.tv_sec > last_rtc_update + 660)
update_RTC();
/*每隔 11 分钟就更新 RTC 中的时间信息，以使 OS 时钟和 RTC 时钟保持同步，11 分钟即660 秒，xtime.tv_sec 的单位是秒，last_rtc_update 记录的是上次 RTC 更新时的值 */
}
```

其中，xtime 是前面所提到的 timeval 类型，这是一个全局变量。

(3) 时钟函数 do_timer()　(在/kernel/sched.c 中)

```
void do_timer (struct pt_regs * regs)
{
     (* (unsigned long *) &jiffies) ++;  /*更新系统时间，这种写法保证对 jiffies
操作的原子性*/
 update_process_times();
    ++lost_ticks;
    if ( ! user_mode  ( regs )  )
        ++lost_ticks_system;

        mark_bh (TIMER_BH) ;
    if  (tq_timer)
        mark_bh (TQUEUE_BH) ;
}
```

其中，update_process_times()函数与进程调度有关，从函数的名子可以看出，它处理

的是与当前进程与时间有关的变量，例如，要更新当前进程的时间片计数器 counter，如果 counter<=0，则要调用调度程序，要处理进程的所有定时器：实时、虚拟、概况，另外还要做一些统计工作。

与时间有关的事情很多，不能全都让这个函数去完成，这是因为这个函数是在关中断的情况下执行，必须处理完最重要的时间信息后退出，以处理其他事情。那么，与时间相关的其他信息谁去处理，何时处理？这就是由第三章讨论的后半部分去去处理。上面 timer_interrupt()（包括它所调用的函数）所做的事情就是上半部分。

在该函数中还有两个变量 lost_ticks 和 lost_ticks_system,这是用来记录 timer_bh()执行前时钟中断发生的次数。因为时钟中断发生的频率很高(每 10ms 一次),所以在 timer_bh()执行之前，可能已经有时钟中断发生了，而 timer_bh()要提供定时、计费等重要操作，所以为了保证时间计量的准确性，使用了这两个变量。lost_ticks 用来记录 timer_bh()执行前时钟中断发生的次数，如果时钟中断发生时当前进程运行于内核态，则 lost_ticks_system 用来记录 timer_bh()执行前在内核态发生时钟中断的次数，这样可以对当前进程精确计费。

(4) 中断安装程序

从上面的介绍可以看出，时钟中断与进程调度密不可分，因此，一旦开始有时钟中断就可能要进行调度，在系统进行初始化时，所做的大量工作之一就是对时钟进行初始化，其函数 time_init ()的代码在/arch/i386/kernel/time.c 中，对其简写如下：

```
 void __init time_init (void)
  {
xtime.tv_sec=get_cmos_time();
xtime.tv_usec=0;
setup_irq (0, &irq0) ;
}
```

其中的 get_cmos_time()函数就是把当时的实际时间从 CMOS 时钟芯片读入变量 xtime 中，时间精度为秒。而 setup_irq (0, &irq0) 就是时钟中断安装函数，那么 irq0 指的是什么呢，它是一个结构类型 irqaction，其定义及初值如下：

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

setup_irq (0, &irq0) 的代码在/arch/i386/kernel/irq.c 中，其主要功能就是将中断程序连入相应的中断请求队列，以等待中断到来时相应的中断程序被执行。

到现在为止，我们仅仅是把时钟中断程序挂入中断请求队列，什么时候执行，怎样执行，这是一个复杂的过程（参见第三章），为了让读者对时钟中断有一个完整的认识，我们忽略中间过程，而给出一个整体描述。我们将有关函数改写如下，体现时钟中断的大意：

```
do_timer_interrupt ( )          ／＊这是一个伪函数 ＊／
{
     SAVE_ALL                   ／＊保存处理机现场 ＊／
     intr_count += 1;             ／＊ 这段操作不允许被中断 ＊／
     timer_interrupt()            ／＊ 调用时钟中断程序 ＊／
     intr_count -= 1;
     jmp ret_from_intr            /* 中断返回函数 ＊／
}
```

其中，jmp ret_from_intr 是一段汇编代码，也是一个较为复杂的过程，它最终要调用 jmp ret_from_sys_call，即系统调用返回函数，而这个函数与进程的调度又密切相关，因此，

我们重点分析 jmp ret_from_sys_call。

3．系统调用返回函数

系统调用返回函数的源代码在/arch/i386/kernel/entry.S 中

```
ENTRY (ret_from_sys_call)
        cli                 # need_resched and signals atomic test
        cmpl $0,need_resched (%ebx)
        jne reschedule
        cmpl $0,sigpending (%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL

        ALIGN
signal_return:
        sti             # we can get here from an interrupt handler
        testl $ (VM_MASK) ,EFLAGS (%esp)
        movl %esp,%eax
        jne v86_signal_return
        xorl %edx,%edx
        call SYMBOL_NAME (do_signal)
        jmp restore_all

        ALIGN
       v86_signal_return:
        call SYMBOL_NAME (save_v86_state)
        movl %eax,%esp
        xorl %edx,%edx
        call SYMBOL_NAME (do_signal)
        jmp restore_all
 …….
reschedule:
        call SYMBOL_NAME (schedule)     # test
       jmp ret_from_sys_call
```

这一段汇编代码就是前面我们所说的“从系统调用返回函数”ret_from_sys_call，它是从中断、异常及系统调用返回时的通用接口。这段代码主体就是 ret_from_sys_call 函数，其执行过程中要调用其他一些函数（实际上是一段代码，不是真正的函数），在此我们列出相关的几个函数。

(1) ret_from_sys_call：主体。

(2) reschedule：检测是否需要重新调度。

(3) signal_return：处理当前进程接收到的信号。

(4) v86_signal_return：处理虚拟 86 模式下当前进程接收到的信号。

(5) RESTORE_ALL：我们把这个函数叫做彻底返回函数，因为执行该函数之后，就返回到当前进程的地址空间中去了。

可以看到 ret_from_sys_call 的主要作用有：检测调度标志 need_resched，决定是否要执行调度程序；处理当前进程的信号；恢复当前进程的环境使之继续执行。

最后我们再次从总体上浏览一下时钟中断：

每个时钟滴答，时钟中断得到执行。时钟中断执行的频率很高：100 次/秒，时钟中断的主要工作是处理和时间有关的所有信息、决定是否执行调度程序以及处理下半部分。和时间有关的所有信息包括系统时间、进程的时间片、延时、使用 CPU 的时间、各种定时器，进程更新后的时间片为进程调度提供依据，然后在时钟中断返回时决定是否要执行调度程序。下半部分处理程序是 Linux 提供的一种机制，它使一部分工作推迟执行。时钟中断要绝对保证维持系统时间的准确性，而下半部分这种机制的提供不但保证了这种准确性，还大幅提高了系统性能。

5.3 Linux 的调度程序——Schedule（）

进程的合理调度是一个非常复杂的工作，它取决于可执行程序的类型（实时或普通）、调度的策略及操作系统所追求的目标，幸运的是，Linux 的调度程序比较简单。

5.3.1 基本原理

从前面我们可以看到，进程运行需要各种各样的系统资源，如内存、文件、打印机和最宝贵的 CPU 等，所以说，调度的实质就是资源的分配。系统通过不同的调度算法（Scheduling Algorithm）来实现这种资源的分配。通常来说，选择什么样的调度算法取决于资源分配的策略（Scheduling Policy），我们不准备在这里详细说明各种调度算法，只说明与 Linux 调度相关的几种算法及这些算法的原理。

一个好的调度算法应当考虑以下几个方面。

（1）公平：保证每个进程得到合理的 CPU 时间。

（2）高效：使 CPU 保持忙碌状态，即总是有进程在 CPU 上运行。

（3）响应时间：使交互用户的响应时间尽可能短。

（4）周转时间：使批处理用户等待输出的时间尽可能短。

（5）吞吐量：使单位时间内处理的进程数量尽可能多。

很显然，这 5 个目标不可能同时达到，所以，不同的操作系统会在这几个方面中作出相应的取舍，从而确定自己的调度算法，例如 UNIX 采用动态优先数调度、5.3BSD 采用多级反馈队列调度、Windows 采用抢先多任务调度等。

下面来了解一下主要的调度算法及其基本原理。

1．时间片轮转调度算法

时间片（Time Slice）就是分配给进程运行的一段时间。

在分时系统中，为了保证人机交互的及时性，系统使每个进程依次地按时间片轮流的方式执行，此时即应采用时间片轮转法进行调度。在通常的轮转法中，系统将所有的可运行（即就绪）进程按先来先服务的原则，排成一个队列，每次调度时把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms 不等。当执行的时间片用完时，系统发

出信号，通知调度程序，调度程序便据此信号来停止该进程的执行，并将它送到运行队列的末尾，等待下一次执行。然后，把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证运行队列中的所有进程，在一个给定的时间（人所能接受的等待时间）内，均能获得一时间片的处理机执行时间。

2．优先权调度算法

为了照顾到紧迫型进程在进入系统后便能获得优先处理，引入了最高优先权调度算法。当将该算法用于进程调度时，系统将把处理机分配给运行队列中优先权最高的进程，这时，又可进一步把该算法分成两种方式。

（1）非抢占式优先权算法（又称不可剥夺调度，Nonpreemptive Scheduling）

在这种方式下，系统一旦将处理机（CPU）分配给运行队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可将处理机分配给另一个优先权高的进程。这种调度算法主要用于批处理系统中，也可用于某些对实时性要求不严的实时系统中。

（2）抢占式优先权调度算法（又称可剥夺调度，Preemptive Scheduling）

该算法的本质就是系统中当前运行的进程永远是可运行进程中优先权最高的那个。

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但是只要一出现了另一个优先权更高的进程时，调度程序就暂停原最高优先权进程的执行，而将处理机分配给新出现的优先权最高的进程，即剥夺当前进程的运行。因此，在采用这种调度算法时，每当出现一新的可运行进程，就将它和当前运行进程进行优先权比较，如果高于当前进程，将触发进程调度。

这种方式的优先权调度算法，能更好的满足紧迫进程的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。Linux 也采用这种调度算法。

3．多级反馈队列调度

这是时下最时髦的一种调度算法。其本质是：综合了时间片轮转调度和抢占式优先权调度的优点，即：优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片。

4．实时调度

最后我们来看一下实时系统中的调度。什么叫实时系统，就是系统对外部事件有求必应、尽快响应。在实时系统中存在有若干个实时进程或任务，它们用来反应或控制某个（些）外部事件，往往带有某种程度的紧迫性，因而对实时系统中的进程调度有某些特殊要求。

在实时系统中，广泛采用抢占调度方式，特别是对于那些要求严格的实时系统。因为这种调度方式既具有较大的灵活性，又能获得很小的调度延迟；但是这种调度方式也比较复杂。

我们大致了解以上的调度方式以后，下面具体来看 Linux 中的调度程序，这里要说明的是，Linux 的调度程序并不复杂，但这并不影响 Linux 调度程序的高效性！

5.3.2 Linux 进程调度时机

调度程序虽然特别重要，但它不过是一个存在于内核空间中的函数而已，并不神秘。Linux 的调度程序是一个叫 Schedule()的函数，这个函数被调用的频率很高，由它来决定是否要进行进程的切换，如果要切换的话，切换到哪个进程等。我们先来看在什么情况下要执行调度程序，我们把这种情况叫做调度时机。

Linux 调度时机主要有。

(1) 进程状态转换的时刻：进程终止、进程睡眠；

(2) 当前进程的时间片用完时（current->counter=0）；

(3) 设备驱动程序；

(4) 进程从中断、异常及系统调用返回到用户态时。

时机 1，进程要调用 sleep()或 exit()等函数进行状态转换，这些函数会主动调用调度程序进行进程调度。

时机 2，由于进程的时间片是由时钟中断来更新的，因此，这种情况和时机 4 是一样的。

时机 3，当设备驱动程序执行长而重复的任务时，直接调用调度程序。在每次反复循环中，驱动程序都检查 need_resched 的值，如果必要，则调用调度程序 schedule()主动放弃 CPU。

时机 4，如前所述，不管是从中断、异常还是系统调用返回，最终都调用 ret_from_sys_call()，由这个函数进行调度标志的检测，如果必要，则调用调用调度程序。那么，为什么从系统调用返回时要调用调度程序呢？这当然是从效率考虑。从系统调用返回意味着要离开内核态而返回到用户态，而状态的转换要花费一定的时间，因此，在返回到用户态前，系统把在内核态该处理的事全部做完。

对于直接执行调度程序的时机，我们不讨论，因为后面我们将描述调度程序的工作过程。前面我们讨论了时钟中断，知道了时钟中断的重要作用，下面我们就简单看一下每个时钟中断发生时内核要做的工作，首先对这个最频繁的调度时机有一个大体了解，然后再详细讨论调度程序的具体工作过程。

每个时钟中断（timer interrupt）发生时，由 3 个函数协同工作，共同完成进程的选择和切换，它们是：schedule()、do_timer()及 ret_form_sys_call()。我们先来解释一下这 3 个函数。

- schedule()：进程调度函数，由它来完成进程的选择（调度）。
- do_timer()：暂且称之为时钟函数，该函数在时钟中断服务程序中被调用，是时钟中断服务程序的主要组成部分，该函数被调用的频率就是时钟中断的频率即每秒钟 100 次（简称 100 赫兹或 100Hz）；
- ret_from_sys_call()：系统调用返回函数。当一个系统调用或中断完成时，该函数被调用，用于处理一些收尾工作，例如信号处理、核心任务等。

这 3 个函数是如何协调工作的呢？

前面我们讲过，时钟中断是一个中断服务程序，它的主要组成部分就是时钟函数 do_timer()，由这个函数完成系统时间的更新、进程时间片的更新等工作，更新后的进程时间片 counter 作为调度的主要依据。

在时钟中断返回时，要调用函数 ret_from_sys_call()，前面我们已经讨论过这个函数，在这个函数中有如下几行：

```
cmpl $0, _need_resched
jne reschedule
     ……
restore_all:
     RESTORE_ALL

reschedule:
     call SYMBOL_NAME (schedule)
     jmp ret_from_sys_call
```

这几行的意思很明显：检测 need_resched 标志，如果此标志为非 0，那么就转到 reschedule 处调用调度程序 schedule()进行进程的选择。调度程序 schedule()会根据具体的标准在运行队列中选择下一个应该运行的进程。当从调度程序返回时，如果发现又有调度标志被设置，则又调用调度程序，直到调度标志为 0，这时，从调度程序返回时由 RESTORE_ALL 恢复被选定进程的环境，返回到被选定进程的用户空间，使之得到运行。

以上就是时钟中断这个最频繁的调度时机。讨论这个的主要目的使读者对时机 4 有个大致的了解。

最后要说明的是，系统调用返回函数 ret_from_sys_call()是从系统调用、异常及中断返回函数通常要调用的函数，但并不是非得调用，对于那些要经常被响应的和要被尽快处理的中断请求信号，为了减少系统开销，处理完成后并不调用 ret_from_sys_call()（因为很明显，从这些中断处理程序返回到的用户空间肯定是那个被中断的进程，无需重新选择），并且，它们作的工作要尽可能少，因为响应的频率太高了。

Linux 调度程序和其他的 UNIX 调度程序不同，尤其是在“nice level”优先级的处理上，与优先权调度（priority 高的进程最先运行）不同，Linux 用的是时间片轮转调度（Round Robing），但同时又保证了高优先级的进程运行得既快、时间又长（both sooner and longer）。而标准的 UNIX 调度程序都用到了多级进程队列。大多数的实现都用到了二级优先队列：一个标准队列和一个实时（“real time”）队列。一般情况下，如果实时队列中的进程未被阻塞，它们都要在标准队列中的进程之前被执行，并且，每个队列中，“nice level”高的进程先被执行。

总体上，Linux 调度序程在交互性方面表现很出色，当然了，这是以牺牲一部分“吞吐量”为代价的。

5.3.3　进程调度的依据

调度程序运行时，要在所有处于可运行状态的进程之中选择最值得运行的进程投入运行。选择进程的依据是什么呢？在每个进程的 task_struct 结构中有如下 5 项：

need_resched、nice、counter、policy 及 rt_priority

(1) need_resched: 在调度时机到来时，检测这个域的值，如果为 1，则调用 schedule()。

(2) counter: 进程处于运行状态时所剩余的时钟滴答数，每次时钟中断到来时，这个值就减 1。当这个域的值变得越来越小，直至为 0 时，就把 need_resched 域置 1，因此，也

把这个域叫做进程的“动态优先级”。

(3) nice: 进程的“静态优先级”，这个域决定 counter 的初值。只有通过 nice()、POSIX.1b sched_setparam() 或 5.4BSD/SVR4 setpriority()系统调用才能改变进程的静态优先级。

(4) rt_priority: 实时进程的优先级

(5) policy: 从整体上区分实时进程和普通进程，因为实时进程和普通进程的调度是不同的，它们两者之间，实时进程应该先于普通进程而运行，可以通过系统调用 sched_setscheduler（ ）来改变调度的策略。对于同一类型的不同进程，采用不同的标准来选择进程。对于普通进程，选择进程的主要依据为 counter 和 nice。对于实时进程，Linux 采用了两种调度策略，即 FIFO（先来先服务调度）和 RR（时间片轮转调度）。因为实时进程具有一定程度的紧迫性，所以衡量一个实时进程是否应该运行，Linux 采用了一个比较固定的标准。实时进程的 counter 只是用来表示该进程的剩余滴答数，并不作为衡量它是否值得运行的标准，这和普通进程是有区别的。

这里再次说明，与其他操作系统一样，Linux 的时间单位也是“时钟滴答”，只是不同的操作系统对一个时钟滴答的定义不同而已(Linux 设计者将一个“时钟滴答”定义为 10ms)。在这里，我们把 counter 叫做进程的时间片，但实际上它仅仅是时钟滴答的个数，例如，若 counter 为 5，则分配给该进程的时间片就为 5 个时钟滴答，也就是 5*10ms=50ms，实际上，Linux 2.4 中给进程初始时间片的大小就是 50ms

5.3.4 进程可运行程度的衡量

函数 goodness()就是用来衡量一个处于可运行状态的进程值得运行的程度。该函数综合使用了上面我们提到的 5 项，给每个处于可运行状态的进程赋予一个权值（weight），调度程序以这个权值作为选择进程的唯一依据。函数主体如下（为了便于理解，笔者对函数做了一些改写和简化，只考虑单处理机的情况）：

```
static inline int goodness (struct task_struct * p, struct mm_struct *this_mm)
{    int weight;      ／＊ 权值，作为衡量进程是否运行的唯一依据 ＊

       weight=-1;
       if  (p->policy&SCHED_YIELD)
        goto out;  /*如果该进程愿意“礼让（yield）”，则让其权值为－1 ＊／
      switch (p->policy)
     {
         /* 实时进程*/
         case SCHED_FIFO:
         case SCHED_RR:
              weight = 1000 + p->rt_priority;

         /* 普通进程 */
         case SCHED_OTHER:
             {     weight = p->counter;
               if (!weight)
                 goto out
```

```
                /* 做细微的调整*/
                if  (p->mm=this_mm||!p->mm)
                          weight = weight+1;
                  weight+=20-p->nice;
                }
    }
out:
    return weight;   /*返回权值*/
}
```

其中，在 sched.h 中对调度策略定义如下：

```
#define SCHED_OTHER             0
#define SCHED_FIFO              1
#define SCHED_RR                2
    #define SCHED_YIELD             0x10
```

这个函数比较很简单。首先，根据 policy 区分实时进程和普通进程。实时进程的权值取决于其实时优先级，其至少是 1000，与 conter 和 nice 无关。普通进程的权值需特别说明如下两点。

（1）为什么进行细微的调整？如果 p->mm 为空，则意味着该进程无用户空间（例如内核线程），则无需切换到用户空间。如果 p->mm=this_mm，则说明该进程的用户空间就是当前进程的用户空间，该进程完全有可能再次得到运行。对于以上两种情况，都给其权值加 1，算是对它们小小的“奖励”。

（2）进程的优先级 nice 是从早期 UNIX 沿用下来的负向优先级，其数值标志“谦让”的程度，其值越大，就表示其越“谦让”，也就是优先级越低，其取值范围为－20～＋19，因此，（20-p->nice）的取值范围就是 0～40。可以看出，普通进程的权值不仅考虑了其剩余的时间片，还考虑了其优先级，优先级越高，其权值越大。

有了衡量进程是否应该运行的标准，选择进程就是轻而易举的事情了，“弱肉强食”，谁的权值大谁就先运行。

根据进程调度的依据，调度程序就可以控制系统中的所有处于可运行状态的进程并在它们之间进行选择。

5.3.5　进程调度的实现

调度程序在内核中就是一个函数，为了讨论方便，我们同样对其进行了简化，略去对 SMP 的实现部分。

```
asmlinkage void schedule (void)
{

  struct task_struct *prev, *next, *p; ／* prev 表示调度之前的进程，
     next 表示调度之后的进程 *／
    struct list_head *tmp;
    int this_cpu, c;

      if  (!current->active_mm)  BUG();/*如果当前进程的 active_mm 为空，出错*／
need_resched_back:
          prev = current;          ／*让 prev 成为当前进程 *／
```

```
            this_cpu = prev->processor;

    if (in_interrupt()) {／*如果 schedule 是在中断服务程序内部执行，
    就说明发生了错误*／
            printk ("Scheduling in interrupt\n") ;
               BUG();
          }
       release_kernel_lock (prev, this_cpu) ; /*释放全局内核锁，
    并开 this_cpu 的中断*／
          spin_lock_irq (&runqueue_lock) ; ／*锁住运行队列，并且同时关中断*/
           if (prev->policy == SCHED_RR) ／*将一个时间片用完的 SCHED_RR 实时
                 goto move_rr_last;       进程放到队列的末尾 *／
     move_rr_back:
           switch (prev->state) {      ／*根据 prev 的状态做相应的处理*／
                 case TASK_INTERRUPTIBLE:  /*此状态表明该进程可以被信号中断*/
                          if (signal_pending (prev) ) { /*如果该进程有未处理的
    信号，则让其变为可运行状态*/
                                    prev->state = TASK_RUNNING;
                                     break;
                          }
                   default:      ／*如果为可中断的等待状态或僵死状态*／
                          del_from_runqueue (prev) ; ／*从运行队列中删除*／
                  case TASK_RUNNING:;／*如果为可运行状态，继续处理*／
            }
            prev->need_resched = 0;

         ／*下面是调度程序的正文 *／
    repeat_schedule:     ／*真正开始选择值得运行的进程*／
           next = idle_task (this_cpu) ; /*缺省选择空闲进程*/
       c = -1000;
         if (prev->state == TASK_RUNNING)
              goto still_running;
    still_running_back:
        list_for_each (tmp, &runqueue_head) { ／*遍历运行队列*／
           p = list_entry (tmp, struct task_struct, run_list) ;
     if ( can_schedule ( p, this_cpu ) ) { ／* 单 CPU 中，该函数总返回 1* ／
int weight = goodness (p, this_cpu, prev->active_mm) ;
                  if (weight > c)
                      c = weight, next = p;
              }
         }

    ／* 如果 c 为 0，说明运行队列中所有进程的权值都为 0，也就是分配给各个进程的
         时间片都已用完，需重新计算各个进程的时间片 *／
        if (!c) {
                 struct task_struct *p;
                 spin_unlock_irq (&runqueue_lock) ;／*锁住运行队列*／
                  read_lock (&tasklist_lock) ; ／* 锁住进程的双向链表*／
                 for_each_task (p)              ／* 对系统中的每个进程*／
                 p->counter = (p->counter >> 1) + NICE_TO_TICKS (p->nice) ;
                 read_unlock (&tasklist_lock) ;
                     spin_lock_irq (&runqueue_lock) ;
```

```
                goto repeat_schedule;
          }


      spin_unlock_irq (&runqueue_lock) ;／*对运行队列解锁，并开中断*／

        if  (prev == next)  {      /*如果选中的进程就是原来的进程*/
             prev->policy &= ~SCHED_YIELD;
                goto same_process;
       }

           ／* 下面开始进行进程切换*／
         kstat.context_swtch++; ／*统计上下文切换的次数*／

          {
                struct mm_struct *mm = next->mm;
                struct mm_struct *oldmm = prev->active_mm;
               if  (!mm)  {  ／*如果是内核线程，则借用 prev 的地址空间*／
                       if  (next->active_mm)  BUG();
                       next->active_mm = oldmm;

               } else { ／*如果是一般进程，则切换到 next 的用户空间*／
                        if  (next->active_mm != mm)  BUG();
                        switch_mm (oldmm, mm, next, this_cpu) ;
               }

              if  (!prev->mm)  { ／*如果切换出去的是内核线程*／
                    prev->active_mm = NULL;／*归还它所借用的地址空间*／
                      mmdrop (oldmm) ;      ／*mm_struct 中的共享计数减 1*／
                }
          }

          switch_to (prev, next, prev) ; ／*进程的真正切换，即堆栈的切换*／
          __schedule_tail (prev) ;  ／*置 prev->policy 的 SCHED_YIELD 为 0 *／

    same_process:
          reacquire_kernel_lock (current) ;／*针对 SMP*／
          if  (current->need_resched)      ／*如果调度标志被置位*／
                goto need_resched_back; ／*重新开始调度*／
          return;
    }
```

以上就是调度程序的主要内容，为了对该程序形成一个清晰的思路，我们对其再给出进一步的解释。

- 如果当前进程既没有自己的地址空间,也没有向别的进程借用地址空间,那肯定出错。另外，如果 schedule()在中断服务程序内部执行,那也出错。
- 对当前进程做相关处理，为选择下一个进程做好准备。当前进程就是正在运行着的进程，可是，当进入 schedule()时，其状态却不一定是 TASK_RUNNIG，例如，在 exit()系统调用中，当前进程的状态可能已被改为 TASK_ZOMBE；又例如，在 wait4()系统调用中，当前进程的状态可能被置为 TASK_INTERRUPTIBLE。因此，如果当前进程处于这些状态中的一种，就

要把它从运行队列中删除。

- 从运行队列中选择最值得运行的进程，也就是权值最大的进程。
- 如果已经选择的进程其权值为 0，说明运行队列中所有进程的时间片都用完了（队列中肯定没有实时进程，因为其最小权值为 1000），因此，重新计算所有进程的时间片，其中宏操作 NICE_TO_TICKS 就是把优先级 nice 转换为时钟滴答。
- 进程地址空间的切换。如果新进程有自己的用户空间，也就是说，如果 next->mm 与 next->active_mm 相同，那么，switch_mm（ ）函数就把该进程从内核空间切换到用户空间，也就是加载 next 的页目录。如果新进程无用户空间（next->mm 为空），也就是说，如果它是一个内核线程，那它就要在内核空间运行，因此，需要借用前一个进程（prev）的地址空间，因为所有进程的内核空间都是共享的，因此，这种借用是有效的。
- 用宏 switch_to()进行真正的进程切换，后面将详细描述。

5.4 进程切换

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换，任务切换，或上下文切换。Intel 在 i386 系统结构的设计中考虑到了进程（任务）的管理和调度，并从硬件上支持任务之间的切换。

5.4.1 硬件支持

Intel i386 体系结构包括了一个特殊的段类型，叫任务状态段（TSS），如图 5.4 所示。每个任务包含有它自己最小长度为 104 字节的 TSS 段，在/include/ i386/processor.h 中定义为 tss_struct 结构：

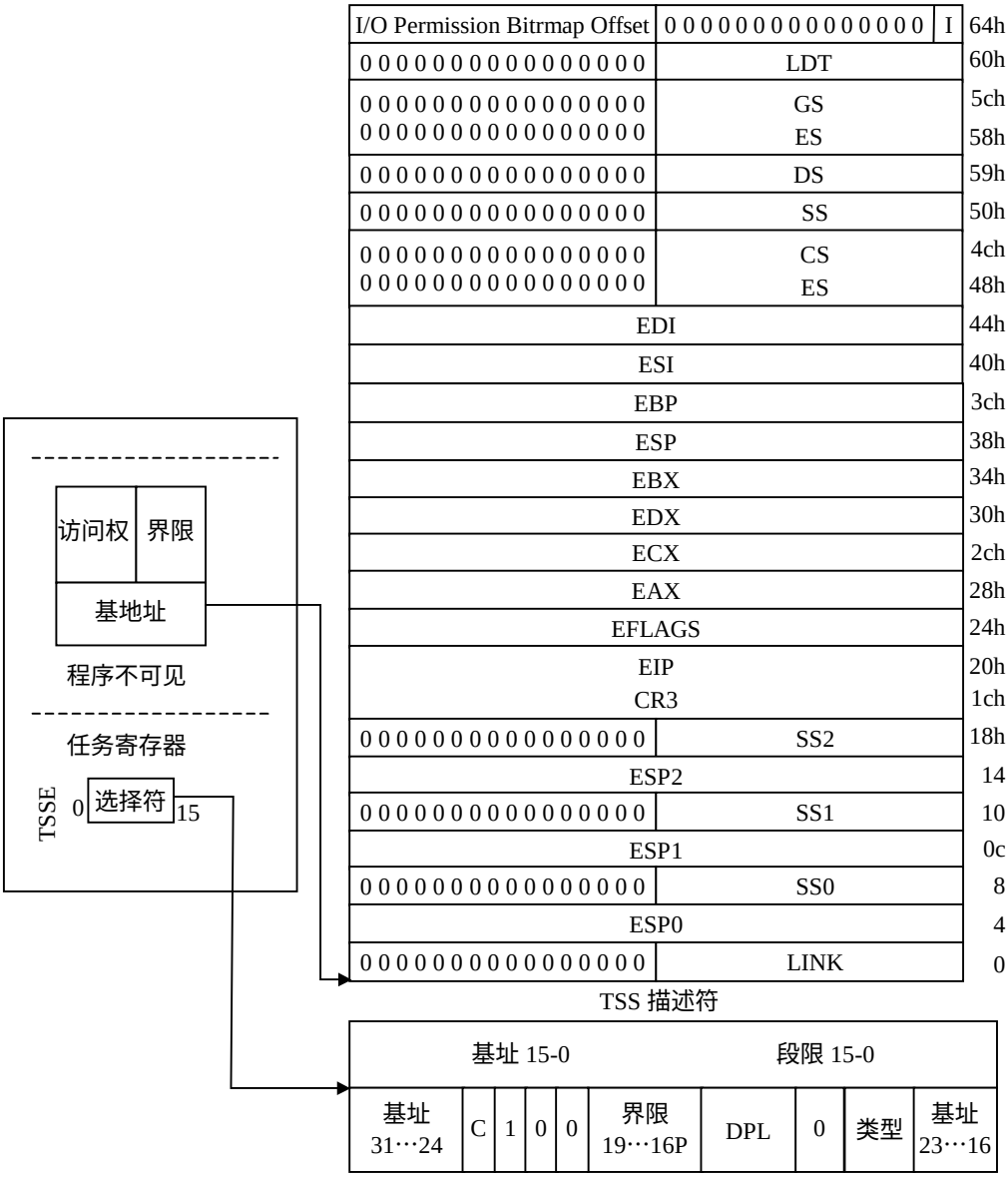


图 5.4　Intel i386 任务状态段及 TR 寄存器

```
struct tss_struct {
        unsigned short  back_link,__blh;
        unsigned long   esp0;
        unsigned short  ss0,__ss0h;／*0 级堆栈指针，即 Linux 中的内核级 *／
        unsigned long   esp1;
        unsigned short  ss1,__ss1h; ／* 1 级堆栈指针，未用*／
        unsigned long   esp2;
        unsigned short  ss2,__ss2h; ／* 2 级堆栈指针，未用*／
        unsigned long   __cr3;
        unsigned long   eip;
        unsigned long   eflags;
```

```
        unsigned long   eax,ecx,edx,ebx;
        unsigned long   esp;
        unsigned long   ebp;
        unsigned long   esi;
        unsigned long   edi;
        unsigned short  es, __esh;
        unsigned short  cs, __csh;
        unsigned short  ss, __ssh;
        unsigned short  ds, __dsh;
        unsigned short  fs, __fsh;
        unsigned short  gs, __gsh;
        unsigned short  ldt, __ldth;
        unsigned short  trace, bitmap;
        unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
         /*
        * pads the TSS to be cacheline-aligned  (size is 0x100)
         */
        unsigned long __cacheline_filler[5];
};
```

每个 TSS 有它自己 8 字节的任务段描述符（Task State Segment Descriptor ，简称 TSSD)。这个描述符包括指向 TSS 起始地址的 32 位基地址域，20 位界限域，界限域值不能小于十进制 104（由 TSS 段的最小长度决定)。TSS 描述符存放在 GDT 中，它是 GDT 中的一个表项。

后面将会看到，Linux 在进程切换时，只用到 TSS 中少量的信息，因此 Linux 内核定义了另外一个数据结构，这就是 thread_struct 结构：

```
struct thread_struct {
        unsigned long   esp0;
        unsigned long   eip;
        unsigned long   esp;
        unsigned long   fs;
        unsigned long   gs;
   /* Hardware debugging registers */
         unsigned long   debugreg[8];  /* %%db0-7 debug registers */
  /* fault info */
         unsigned long   cr2, trap_no, error_code;
   /* floating point info */
         union i387_union        i387;
  /* virtual 86 mode info */
        struct vm86_struct      * vm86_info;
         unsigned long           screen_bitmap;
        unsigned long           v86flags, v86mask, v86mode, saved_esp0;
/* IO permissions */
         int             ioperm;
        unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
};
```

用这个数据结构来保存 cr2 寄存器、浮点寄存器、调试寄存器及指定给 Intel 80x86 处理器的其他各种各样的信息。需要位图是因为 ioperm（ ） 及 iopl（ ）系统调用可以允许用户态的进程直接访问特殊的 I/O 端口。尤其是，如果把 eflag 寄存器中的 IOPL 域设置

为 3，就允许用户态的进程访问对应的 I/O 访问权位图位为 0 的任何一个 I/O 端口。

那么，进程到底是怎样进行切换的？

从第三章我们知道，在中断描述符表（IDT）中，除中断门、陷阱门和调用门外，还有一种“ 任务门 ”。任务门中包含有 TSS 段的选择符。当 CPU 因中断而穿过一个任务门时，就会将任务门中的段选择符自动装入 TR 寄存器，使 TR 指向新的 TSS，并完成任务切换。CPU 还可以通过 JMP 或 CALL 指令实现任务切换，当跳转或调用的目标段（代码段）实际上指向 GDT 表中的一个 TSS 描述符项时，就会引起一次任务切换。

Intel 的这种设计确实很周到，也为任务切换提供了一个非常简洁的机制。但是，由于 i386 的系统结构基本上是 CISC 的，通过 JMP 指令或 CALL（或中断）完成任务的过程实际上是“ 复杂指令 ”的执行过程，其执行过程长达 300 多个 CPU 周期（一个 POP 指令占 12 个 CPU 周期），因此，Linux 内核并不完全使用 i386 CPU 提供的任务切换机制。

由于 i386 CPU 要求软件设置 TR 及 TSS，Linux 内核只不过“ 走过场 ”地设置 TR 及 TSS，以满足 CPU 的要求。但是，内核并不使用任务门，也不使用 JMP 或 CALL 指令实施任务切换。内核只是在初始化阶段设置 TR，使之指向一个 TSS，从此以后再不改变 TR 的内容了。也就是说，每个 CPU（如果有多个 CPU）在初始化以后的全部运行过程中永远使用那个初始的 TSS。同时，内核也不完全依靠 TSS 保存每个进程切换时的寄存器副本，而是将这些寄存器副本保存在各个进程自己的内核栈中（参见上一章 task_struct 结构的存放）。

这样以来，TSS 中的绝大部分内容就失去了原来的意义。那么，当进行任务切换时，怎样自动更换堆栈？我们知道，新任务的内核栈指针（SS0 和 ESP0）应当取自当前任务的 TSS，可是，Linux 中并不是每个任务就有一个 TSS，而是每个 CPU 只有一个 TSS。Intel 原来的意图是让 TR 的内容（即 TSS）随着任务的切换而走马灯似地换，而在 Linux 内核中却成了只更换 TSS 中的 SS0 和 ESP0，而不更换 TSS 本身，也就是根本不更换 TR 的内容。这是因为，改变 TSS 中 SS0 和 ESP0 所化的开销比通过装入 TR 以更换一个 TSS 要小得多。因此，在 Linux 内核中，TSS 并不是属于某个进程的资源，而是全局性的公共资源。在多处理机的情况下，尽管内核中确实有多个 TSS，但是每个 CPU 仍旧只有一个 TSS。

5.4.2　进程切换

前面所介绍的 schedule()中调用了 switch_to 宏，这个宏实现了进程之间的真正切换，其代码存放于 include/ i386/system.h：

```
1   #define switch_to (prev,next,last)  do {                                    \
2         asm volatile ("pushl %%esi\n\t"                                        \
3                       "pushl %%edi\n\t"                                       \
4                       "pushl %%ebp\n\t"                                       \
5                       "movl %%esp,%0\n\t"        /* save ESP */               \
6                       "movl %3,%%esp\n\t"        /* restore ESP */            \
7                       "movl $1f,%1\n\t"          /* save EIP */               \
8                       "pushl %4\n\t"             /* restore EIP */            \
9                       "jmp __switch_to\n"                                     \
10                      "1:\t"                                                  \
11                      "popl %%ebp\n\t"                                        \
12                      "popl %%edi\n\t"                                        \
```

```
13                      "popl %%esi\n\t"                                    \
14                       :"=m" (prev->thread.esp) ,"=m" (prev->thread.eip) ,  \
15                        "=b" (last)                                         \
16                       :"m" (next->thread.esp) ,"m" (next->thread.eip) ,    \
17                        "a" (prev) , "d" (next) ,                           \
18                        "b" (prev) ) ;                                      \
19 } while (0)
```

switch_to 宏是用嵌入式汇编写成，比较难理解，为描述方便起见，我们给代码编了行号，在此我们给出具体的解释。

- thread 的类型为前面介绍的 thread_struct 结构。
- 输出参数有 3 个，表示这段代码执行后有 3 项数据会有变化，它们与变量及寄存器的对应关系如下：

0%与 prev->thread.esp 对应，1%与 prev->thread.eip 对应，这两个参数都存放在内存，而 2%与 ebx 寄存器对应，同时说明 last 参数存放在 ebx 寄存器中。

- 输入参数有 5 个，其对应关系如下：

3%与 next->thread.esp 对应，4%与 next->thread.eip 对应，这两个参数都存放在内存，而 5%、6%和 7%分别与 eax、edx 及 ebx 相对应，同时说明 prev、next 以及 prev 这 3 个参数分别放在这 3 个寄存器中。表 5.1 列出了这几种对应关系。

- 第 2～4 行就是在当前进程 prev 的内核栈中保存 esi、edi 及 ebp 寄存器的内容。
- 第 5 行将 prev 的内核堆栈指针 ebp 存入 prev->thread.esp 中。
- 第 6 行把将要运行进程 next 的内核栈指针 next->thread.esp 置入 esp 寄存器中。从现在开始，内核对 next 的内核栈进行操作，因此，这条指令执行从 prev 到 next 真正的上下文切换，因为进程描述符的地址与其内核栈的地址紧紧地联系在一起（参见第四章），因此，改变内核栈就意味着改变当前进程。如果此处引用 current，那就已经指向 next 的 task_struct 结构了。从这个意义上说，进程的切换在这一行指令执行完以后就已经完成。但是，构成一个进程的另一个要素是程序的执行，这方面的切换尚未完成。

表 5.1　　输入/输出参数与变量及寄存器的对应关系

参数类型	参数名	内存变量	寄存器	函数参数
输出参数	0%	prev->thread.esp		
	1%	prev->thread.eip		
	2%		ebx	last
输入参数	3%	next->thread.esp		
	4%	next->thread.eip		
	5%		eax	prev
	6%		edx	next
	7%		ebx	prev

- 第 7 行将标号“1”所在的地址，也就是第一条 popl 指令（第 11 行）所在的地址保

存在 prev->thread.eip 中，这个地址就是 prev 下一次被调度运行而切入时的“返回”地址。

- 第 8 行将 next->thread.eip 压入 next 的内核栈。那么，next->thread.eip 究竟指向那个地址？实际上，它就是 next 上一次被调离时通过第 7 行保存的地址，也就是第 11 行 popl 指令的地址。因为，每个进程被调离时都要执行这里的第 7 行，这就决定了每个进程（除了新创建的进程）在受到调度而恢复执行时都从这里的第 11 行开始。
- 第 9 行通过 jump 指令（而不是 call 指令）转入一个函数__switch_to()。这个函数的具体实现将在下面介绍。当 CPU 执行到__switch_to()函数的 ret 指令时，最后进入堆栈的 next->thread.eip 就变成了返回地址，这就是标号“1”的地址。
- 第 11～13 行恢复 next 上次被调离时推进堆栈的内容。从现在开始，next 进程就成为当前进程而真正开始执行。

下面我们来讨论__switch_to()函数。

在调用__switch_to()函数之前，对其定义了 fastcall：

```
extern void FASTCALL (__switch_to (struct task_struct *prev, struct task_struct *next) ) ;
```

fastcall 对函数的调用不同于一般函数的调用，因为__switch_to()从寄存器(如表 5.1)取参数，而不像一般函数那样从堆栈取参数，也就是说，通过寄存器 eax 和 edx 把 prev 和 next 参数传递给__switch_to()函数。

```
void __switch_to (struct task_struct *prev_p, struct task_struct *next_p)
{
        struct thread_struct *prev = &prev_p->thread,
                             *next = &next_p->thread;
        struct tss_struct *tss = init_tss + smp_processor_id();

      unlazy_fpu (prev_p) ;／* 如果数学处理器工作，则保存其寄存器的值*／

   /* 将 TSS 中的内核级（0 级）堆栈指针换成 next->esp0，这就是 next 进程在内核
      栈的指针*/

      tss->esp0 = next->esp0;

     ／＊ 保存 fs 和 gs，但无需保存 es 和 ds，因为当处于内核时，内核段
总是保持不变＊／

       asm volatile ("movl %%fs,%0":"=m"  (* (int *) &prev->fs) ) ;
       asm volatile ("movl %%gs,%0":"=m"  (* (int *) &prev->gs) ) ;

     ／＊恢复 next 进程的 fs 和 gs ＊／

         loadsegment (fs, next->fs) ;
        loadsegment (gs, next->gs) ;

／＊ 如果 next 挂起时使用了调试寄存器，则装载 0～7 个寄存器中的 6 个寄存器，其中第 4、5 个寄存器没有使用 ＊／

        if  (next->debugreg[7]) {
                loaddebug (next, 0) ;
                loaddebug (next, 1) ;
```

```
                loaddebug (next, 2) ;
                loaddebug (next, 3) ;
                 /* no 4 and 5 */
                loaddebug (next, 6) ;
                loaddebug (next, 7) ;
        }

         if  (prev->ioperm || next->ioperm)  {
                if  (next->ioperm)  {

/*把 next 进程的 I/O 操作权限位图拷贝到 TSS 中 */
                      memcpy (tss->io_bitmap, next->io_bitmap,
IO_BITMAP_SIZE*sizeof (unsigned long) ) ;

/* 把 io_bitmap 在 tss 中的偏移量赋给 tss->bitmap */
                        tss->bitmap = IO_BITMAP_OFFSET;
                } else

/*如果一个进程要使用 I/O 指令，但是，若位图的偏移量超出 TSS 的范围，
就会产生一个可控制的 SIGSEGV 信号。第一次对 sys_ioperm()的调用会
建立起适当的位图  */

                      tss->bitmap = INVALID_IO_BITMAP_OFFSET;
         }
}
```

从上面的描述我们看到，尽管 Intel 本身为操作系统中的进程（任务）切换提供了硬件支持，但是 Linux 内核的设计者并没有完全采用这种思想，而是用软件实现了进程切换，而且，软件实现比硬件实现的效率更高，灵活性更大。