第 6 章 时间流

至此，我们知道怎样写一个特性比较完全的字符模块了。我们将在后面几章陆续讨论驱动程序可以访问的一些内核资源。本章，我们先来看看内核代码是如何对时间问题进行处理的。该问题包括(按复杂程度排列)：

- 如何获得当前时间
- 如何将操作延迟指定的一段时间
- 如何调度函数到指定的时间后异步执行

内核中的时间间隔

我们首先要涉及的是时钟中断，操作系统通过时钟中断来确定时间间隔。时钟中断的发生频率设定为 **HZ**，**HZ** 是一个与体系结构无关的常数，在文件**<linux/param.h>**中定义。至少从 2.0 版到 2.1.43 版，Alpha 平台上 Linux 定义 HZ 的值为 1024，而其他平台上定义为 100。

当时钟中断发生时，**jiffies** 值就加 1。因此，**jiffies** 值就是自操作系统启动以来的时钟滴答的数目；**jiffies** 在头文件**<linux/sched.h>**中被定义为数据类型为 **unsigned long volatile** (32 位无符号长整型)的变量，因此连续累加一年又四个多月后就会溢出(假定 **HZ**=100，1 个 **jiffies** 等于 1/100 秒，**jiffies** 可记录的最大秒数为 42949672.96 秒，约合 1.38 年)。如果你打算连续运行 Linux 一年又四个多月以上，你最好买台 Alpha，那么，就是跑五亿年也不会溢出了－Alpha 机器上 **jiffies** 有 64 位。我是无法准确地告诉你 **jiffies** 溢出时会发生些什么的，我可没有那么长的时间来等这件事发生。

如果你修改 **HZ** 值后重编译内核，在用户空间你不会注意到有什么不同。尽管 **jiffies** 值以不同的步长增长，但一切似乎还正常。会产生更多的中断，系统开销更大了，但是因为处理器调度得更频繁了，系统会很不稳定。

我在我的PC上测试了一些**jiffies**值：在 100Hz时，系统的响应很慢；100Hz是缺省值；在 1kHz时，系统跑的相当慢，但响应得很快；在 10kHz时，系统极慢；在 50kHz时，系统已经令人无法忍受了。修改中断频率还有副作用，**jiffies**值溢出要花的时间不同了(10kHz的时钟频率下，只要五天)，BogoMips值的计算精度也不同了*。而且还有一些别处都没提及的硬件上的限制。例如，19 是PC上时钟频率的能设的最小值，其他体系结构上也存在着类似的限制。

此外，在使用模块时还要小心。如果你改变了 **HZ** 的定义，你*必须*重新编译和安装你使用的所有模块。内核中一切都与 **HZ** 值有关，包括模块。我是在增加了 **HZ** 值因而无法双击鼠标后，发现到这一点的。

总而言之，时钟中断的最好方法是保留 **HZ** 的缺省值，因为我们可以完全相信内核的开发者们，他们一定已经为我们挑选了最佳值。关于本专题的更多信息可参见头文件 **<linux/timex.h>**。

获取当前时间

内核一般通过 **jiffies** 值来获取当前时间。尽管该数值表示的是自上次系统启动到当前的时间间隔，但因为驱动程序的生命期只限于系统的运行期(uptime)，所以也是可行的。驱动程序利用 **jiffies** 的当前值来计算不同事件间的时间间隔(我在 *kmouse* 模块中就用它来分辨鼠标的单双击)。简而言之，利用 **jiffies** 值来测量时间间隔还是很有效的。

驱动程序一般不需要知道墙上时间，通常只有象 *cron* 和 *at* 这样用户程序才需要墙上时间。需要墙上时间的情形是使用设备驱动程序的特殊情况，此时可以通过用户程序来将墙上时间转换成系统时钟。

如果驱动程序真的需要获取当前时间，可以使用 *do_gettimeofday* 函数。该函数并不返回今天是本周的星期几或类似的信息；它是用微秒值来填充一个指向 **struct timeval** 的指针变量。相应的原型如下：

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

源码中声明的 *do_gettimeofday* 在 Alpha 和 Sparc 之外的体系结构上有“接近微秒级的分辨率”，在 Alpha 和 Sparc 上和 **jiffies** 值的分辨率一样。Sparc 的移植版本在 2.1.34 版的内核中升级了，可以支持更细粒度的时间度量。当前时间也可以通过 **xtime** 变量(类型为 **struct timeval**)获得(但精度差些)；但是，并不鼓励直接使用该变量，因为除非关闭中断，无法原子性地访问 **timeval** 变量的两个域 **tv_sec** 和 **tv_usec**。使用 *do_gettimeofday* 填充的 **timeval** 结构变量会更快些。

令人遗憾的是，1.2 版的 Linux 并未开放 *do_gettimeofday* 函数。如果你要获取当前时间，又希望程序能够向后兼容，你应该使用该函数下面的这个版本：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,46)
/*
 * 内核头文件已经该函数是非静态的。
 * 我们应先用其他名字来实现它，再 #define 它。
 */
extern inline void redo_gettimeofday(struct timeval *tv)
{
 unsigned long flags;
```

---

* 由于中断的开销，时钟频率越高，精度就越差。

```
  save_flags(flags);
  cli();
  *tv=xtime;
  restore_flags(flags);
}
#define do_gettimeofday(tv) redo_gettimeofday(tv)
#endif
```

这个版本比实际的版本精度要差些，因为它只使用 **xtime** 结构的当前值，这个值并不会比 **jiffies** 值的粒度更细。但是，它却能在不同的 Linux 平台间移植。“实际的”函数是通过与体系结构相关的代码查询硬件时钟来获得更高的分辨率。

获取当前时间的代码可见于 *jit*("Just In Time")模块，源文件可以从 O'Reilly 公司的 FTP 站点获得。*jit* 模块将创建*/proc/currentime* 文件，它以 ASCII 码的形式返回它读该文件的时间。我选择用动态的*/proc* 文件，是因为这样模块代码量会小些－不值得为返回两行文本而写一整个设备驱动程序。

如果你用 *cat* 命令在一个时钟滴答内多次读该文件，就会发现 **xtime** 和 do_gettimeofday 两者的差异了：

```
morgana%cat /proc/currentime /proc/currentime /proc/currentime
gettime: 846157215.937221
xtime:   846157215.931188
jiffies:   1308094
gettime: 846157215.939950
xtime:   846157215.931188
jiffies:   1308094
gettime: 846157215.942465
xtime:   846157215.941188
jiffies:   1308095
```

延迟执行

使用定时器中断和 **jiffies** 值，时钟滴答的整数倍的时间间隔很容易获得，但更小的时延，程序员必须通过软件循环来获得，这将在本节稍后处介绍。

尽管我将会介绍一些很奇特的技术，但我认为最好先看些简单的延迟实现代码，尽管下面要介绍的第一种实现并不是最好的。

长延迟

如果你想将执行延迟几个时钟滴答或者你对延迟的精度要求不高(比如，你想延迟整数数目的秒数)，最简单的实现(最笨的)如下，也就是*忙等待*：

```
unsigned long j=jiffies+jit_delay*HZ;

while (jiffies<j)
 /* 什么也不做 */
```

这种实现当然要避免*。我在这提到它只是因为有时你可以运行这段代码来更好地理解其他实现(在本章稍后处我会说明如何利用忙等待来做测试)。

还是先看看这段代码是如何工作的。因为内核的头文件中 **jiffies** 被声明为 **volatile** 型变量，每次 C 代码访问它时都会重新读取它，因此该循环可以起到延迟的作用。尽管也是“正确”的实现，但这个忙等待循环在延迟期间会锁住整台计算机；因为调度器不会中断运行在内核空间的进程。而且当前的内核实现为不可重入的，因此内核中的忙等待循环将会锁住一台 SMP 机器的所有处理器。

更糟糕的是，如果在进入循环之前关闭了中断，**jiffies** 值就不会得到更新，那么 while 循环的条件就永真。你将不得不按下大红按钮(指冷启动)。

这种延迟和下面的几种延迟方法都在 *jit* 模块中实现了。由该模块创建的所有*/proc/jit** 文件每次被读时都延迟整整 1 秒。如果你想测试忙等待代码，可以读*/proc/jitbusy* 文件，当该文件的 *read* 方法被调用时它将进入忙等待循环，延迟 1 秒；而象 *dd if=/proc/jitbusy bs=1* 这样的命令每次读一个字符就要延迟 1 秒。

可以想见，读*/proc/jitbusy* 文件会大大影响系统性能，因为此时计算机要到 1 秒后才能运行其他进程。

更好的延迟方法如下，它允许其他进程在延迟的时间间隔内运行，尽管这种方法不能用于实时任务或者其他时间要求很严格的场合：

```
while (jiffies<j)
   schedule();
```

这个例子和下面各例中的变量 j 应是延迟到达时的 **jiffies** 值，在忙等待时一般就是象这样使用的。

这种循环(可以通过读*/proc/jitsched* 文件来测试它)延迟方法还不是最优的。系统可以调度其他任务；当前任务除了释放 CPU 之外不做任何工作，但是它仍在任务队列中。如果它是系统中唯一的可运行的进程，它还会被运行(系统调用调度器，调度器还是同一个进程，此进程又再调用调度器，然后...)。换句话说，机器的负载(系统中运行的进程个数)至少为 1，而 idle 空闲进程(进程为 0，历史性的被称为"swapper")绝不会被运行。尽管这个问题看来无所谓，当系统空闲时运行 idle 空闲进程可以减轻处理器负载，降低处理器温度，延长处理器寿命，如果是手提电脑，电池的寿命也可延长。而且，延迟期间实际上进程是在执行的，因此这段延迟还是记在它的运行时间上的。运行命令 *time cat /proc/jitsched* 就可以发现到这一

* 尤其在SMP机器上要避免，这种实现可能会锁住整个机器。

点。

尽管有些毛病，这种循环延迟还是提供了一种有点“脏” 但比较快的监控驱动程序工作的途径。如果模块中的臭虫(bug)会锁死整个系统，在每个用于调试的 *printk* 语句后都添加一小段延迟，可以保证在处理器碰到令人厌恶的臭虫被锁死之前，所有的打印消息都能进入系统日志(system log)中。如果没有这样的延迟，这些消息能进入内存缓冲区，但在 *klogd* 得到运行前系统可能已经被锁住了。

还有其他更好的获得延迟的方法。在内核态下让进程进入睡眠态的正确方式是设置 **current->timeout** 后睡眠在一个等待队列上。调度器每次运行时都会比较进程的 **timeout** 值和当前的 **jiffies** 值，如果 **timeout** 值小于等于当前时间，那么不管它的等待队列如何进程都会被唤醒。只要没有系统事件唤醒进程使它离开等待队列，那么一旦当前时间达到 timeout 值，调度器就唤醒睡眠进程。

这种延迟实现如下：

```
struct wait_queue *wait=NULL;

current->timeout=j;
interruptible_sleep_on(&wait);
```

注意要调用 *interruptible_sleep_on* 而不是 *sleep_on*，因为调度器不检查不可中断的进程的 **timeout** 值－这种进程的睡眠即使超时也不被中断。因此，如果你调用 *sleep_on*，就无法中断该睡眠进程。你可以通过读*/proc/jitqueue* 文件来测试上面的代码。

Timeout 域是个很有意思的系统资源。它可以用来实现阻塞的系统调用和计算延迟。如果硬件保证只要不出错就能在确定的时间内给出响应，那么驱动程序就可以在恰当设置 **timeout** 值后进入睡眠。例如，如果你有一个对海量存储的数据传输请求(读或者写)，而磁盘响应该请求比如说要 1 秒。如果你设置了 timeout 值，并且当前时间达到它了，进程于是被唤醒，驱动程序开始处理这个请求。如果你使用这种技术，在进程被正常唤醒之后 timeout 值应被清为零。而如果进程是因为 **timeout** 超时而被唤醒的，调度器会清这个域，驱动程序就不必再做了。

你可能注意到了，如果目的只是插入延迟，这里并没有必要使用等待队列。实际上，如下所示，用 **current->timeout** 而不用等待队列就可以达到目的：

```
current->timeout=j;
current->state=TASK_INTERRUPTIBLE;
schedule();
current->timeout=0;/*  重置 timeout 值*/
```

这段语句是在调用调度器之前先改变进程的状态。进程的状态被标记为 **TASK_INTERRUPTIBLE**(与 **TASk_RUNNING** 相对应)，这保证了该进程在超时前不会被再次运行(但其他系统事件如信号可能会唤醒它)。这种延迟方法在文件*/proc/jitself* 中实现了－这个名字强调了，读进程是“自己进入睡眠的”，而不是通过调用 *sleep_on*。

短延迟

有时驱动程序需要非常短的延迟来和硬件同步。此时，使用 **jiffies** 值就不能达到目的。

这时就要用内核函数*udelay*[*]。它的原型如下：

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
```

该函数在绝大多数体系结构上是作为内联函数编译的，并且使用软件循环将执行延迟指定数量的微秒数。这里要用到 BogoMips 值：*udelay* 利用了整数值 **loops_per_second**，这个值是在启动时计算 BogoMips 时得到的。

udelay 函数只能用于获取较短的时间延迟，因为 **loops_per_second** 值的精度就只有 8 位，所以当计算更长的延迟时会积累下相当大的误差。尽管运行的最大延迟将近 1 秒(因为更长的延迟就要溢出)，推荐的 *udelay* 函数的参数的最大值是取 1000 微秒(1 毫秒)。

要特别注意的是 *udelay* 是个忙等待函数，在延迟的时间段内无法运行其他的任务。源码见头文件**<asm/delay.h>**。

目前内核不支持大于 1 微秒而小于 1 个时钟滴答的延迟，但这不是个问题，因为延迟是给硬件或者人去识别的。百分之一秒的时间间隔对人来说延迟精度足够了，而 1 毫秒对硬件来说延迟时间也足够长。如果你*真的*需要其间的延迟间隔，你只要建立一个连续执行*udelay(1000)*函数的循环。

任务队列

许多驱动程序需要将任务延迟到以后处理，但又不想占用中断。Linux 为此提供了两种方法：任务队列和内核定时器。任务队列的使用很灵活，可以或长或短地延迟任务到以后处理，在写中断处理程序时任务队列非常有用，在第 9 章“*中断处理*”中，我们还将在“下半部处理”一节中继续讨论。内核定时器则用来调度任务在未来某个相对精确的时间执行，将在本章的“内核定时器”一节中讨论。

要使用到任务队列的一个典型情形是，硬件不产生中断，但仍希望提供阻塞的读。此时需要对设备进行轮询，但要小心地不使 CPU 负担过多无谓的操作。将读进程到指定的时间后(例如，使用 **current->timeout** 变量)唤醒并不是个很好的方法，因为每次轮询需要两次上下文切换，而且通常轮询机制在进程上下文之外才可能较好地实现。

类似的情形还有象不时地给简单的硬件设备提供输入。例如，有一个直接连接到并口的步进马达，要求该马达能一步步地移动。在这种情况下，由控制进程通知设备驱动程序进行移动，但实际上移动是在 *write* 返回后才一步步地进行的。

---

[*] u表示希腊字母"mu"(μ)，它代表“微”。

快速完成这类不固定的任务的恰当方法是注册任务在未来执行。内核提供了对任务“队列”的支持，任务可以累积到队列上一块“运行”。你可以声明你自己的任务队列并且随意地操纵它，或者也可以将你的任务注册到预定义的任务队列中去，由内核来运行它。

下面一节将先概述任务队列，然后介绍预定义的任务队列，这让你可以开始进行一些有趣的测试(如果出错也可能挂起系统)，最后介绍如何运行你自己的任务队列。

任务队列的特性

任务队列是任务的一张列表，每个任务用一个函数指针和一个参数表示。任务运行时，它接受一个 **void *** 类型的参数，返回值类型为 **void**。而参数指针 data 可用来将一个数据结构传入函数，或者可以被忽略。队列本身是结构(任务)的列表，为声明和操纵它们的内核模块所拥有。这些模块全权负责这些数据结构的分配和释放；为此一般使用静态的数据结构。

队列元素由下面这个结构来描述，这段代码是直接从头文件**<linux/tqueue.h>**拷贝下来的：

```
struct tq_struct {
        struct tq_struct *next;         /*  激活的 bh 的链接表  */
        unsigned long sync;               /*  必须初始化为零  */
        void (*routine)(void *);        /*  调用的函数  */
        void *data;                       /*  传递给函数的参数  */
};
```

第一行注释中的 bh 指的是*下半部处理程序(bottom-half)*。下半部处理程序是“中断处理处理程序的下半部”；我们将在第 9 章的“下半部处理程序”一节介绍中断时详细讨论。

任务队列是处理异步事件的重要资源，而且绝大多数的中断处理程序将它们的任务延迟到任务队列被处理时执行。另外，有些任务队列*是*下半部处理程序，通过调用 *do_bottom_half* 函数来处理。本章并不要求你理解下半部处理，但必要时我也会涉及到。

上面的数据结构中最重要的字段是 **routine** 和 **data**。将要延迟的任务插入队列，必须先设置好结构的这些字段，并把 **next** 和 **sync** 两个字段清零。结构中的 **sync** 标志位用于避免同一任务被插入多次，这会破坏 **next** 指针。一旦任务被排入队列，该数据结构就被认为为内核“拥有”了，不能再被修改。

与任务队列有关的其他数据结构还有 **task_queue**，目前它实现为指向 **tq_struct** 结构的指针；如果将来需要扩充 **task_queue**，只要用 **typedef** 将该指针定义为其他符号就可以了。

下面的列表汇总了所有可以对 **tq_struct** 结构进行的操作；所有的函数都是内联的。

void queue_task(struct tq_struct *task, task_queue *list);

正如该函数的名字，本函数用于将任务排进队列中。它关闭了中断，避免了竞争，因此

可以被模块中任一函数调用。

void queue_task_irq(struct tq_struct *task, task_queue *list);

与前者类似，但本函数只能由不可重入的函数调用(象中断处理程序，所以本函数的名字带上了 irq)。它比 *queue_task* 函数要快一些，因为它在排队前不关闭中断。如果你在一个可重入的函数内调用本函数，由于没有屏蔽资源竞争，是很危险的。但是，本函数排除了“运行时排队”的情形(也即将任务插入正在运行的那个任务的位置上)。

void queue_task_irq_off(struct tq_struct *task, task_queue *list);

本函数只能在中断已关闭的情况下调用。它比前两个函数要快，但没有防止象“并发排队”和“运行时排队”这样的资源竞争。

void run_task_queue(struct tq_struct *task, task_queue *list);

run_task_queue 函数用于运行累积在队列上的任务。除非你要声明和维护自己的任务队列，否则不必调用本函数。

2.1.30 版的内核已经不提供 *queue_task_irq* 和 *queue_task_irq_off* 这两个函数了，被认为得不偿失。详情见第 17 章“*最近的发展*” 的“任务队列”一节。

在研讨任务队列的细节之前，最好还是先介绍一下内部的一些实现细节。任务队列与相应的系统调用是异步执行的；这种异步执行特别需要注意，必须先介绍一下。

任务队列要在*安全的时间内*运行。这里安全的意思是在执行时没有什么特别严格的要求。因为在处理任务队列时允许硬件中断，任务代码也不要求执行的非常快。但队列中的函数执行得也不能太慢，毕竟在整个处理任务队列的期间，只有硬件中断才能被系统处理。

另一个与任务队列有关的概念是*中断时间*。在Linux中，中断时间是个软件上的概念，取决于内核的全局变量**intr_count**。任一时候该变量都记录了正在执行的中断处理程序被嵌套的层数*。

一般的计算流程中，当处理器允许某个进程时，**intr_count** 值为 0。当 **intr_count** 不为零时，，执行的代码就与系统的其他部分是异步的了。这些异步代码可以是硬件中断的处理或者是“软件中断”－与任何进程都无关的一个任务，我们称它在“中断时间内”运行。这种异步代码是不允许做某些操作的；特别的，它不能使当前进程进入睡眠，因为 current 指针的值与正在运行的软件中断代码无关。

典型的例子是退出系统调用时要执行的代码。如果因为某个原因此时还有任务需要得到执行，内核可以一退出系统调用就处理它。这是个“软件中断”，**intr_count** 值在处理这个待执行的任务之前会先加 1。由于主线指令流被中断了，该函数算是在“中断时间”内被处理的。

当 **intr_count** 非零时，不能激活调度器。这也就意味着不允许调用 **kmalloc(GFP_KERNEL)**。在中断时间内，只能进行原子性的分配(见第 7 章“*掌握内存*”

* 2.1.34 版的内核不再使用**intr_count**变量。详情见第 17 章的“中断管理”一节。

的“优先权参数”一节)，而原子性的分配较“普通的”分配更容易失败。

如果运行在中断时间的代码调用了调度器，类似“Aiee: scheduling in interrupt”这样的错误信息和以 16 进制显示的调用点处的地址会打印到控制台上。2.1.37 之后的版本，oops 消息也会打印出来，通过分析寄存器的值可以进行调试。在中断时间内如果试图非原子性地按优先权分配内存，也会显示包括着调用者的调用点处地址的错误信息。

预定义的任务队列

延迟任务执行的简单方法是使用内核维护的任务队列。这种队列有下面描述的四种，但驱动程序只能用前三种。任务队列的定义在头文件**<linux/queue.h>**中，你的驱动程序代码要将它包含(include)进来。

tq_scheduler 队列

当调度器被运行时该队列就会被处理。因为此时调度器在被调度出的进程的上下文中运行，所以该队列中的任务几乎可以做任何事；它们不会在中断时运行。

tq_timer 队列

该队列由定时器队列处理程序(timer tick)运行。因为该处理程序(见函数 *do_timer*)是在中断时间运行的，该队列中的所有任务就也是在中断时间内运行的了。

tq_immediate 队列

立即队列在系统调用返回时或调度器运行时尽快得到处理的(不管两种情况谁先发生了)。该队列是在中断时间内得到处理的。

tq_disk 队列

1.2 版的内核不再提供这种任务队列了，内存管理例程内部使用，模块不能使用。

使用任务队列的一个设备驱动程序的执行流程可见图 6-1。该图演示了设备驱动程序是如何在中断处理程序中将一个函数插入 **tq_scheduler** 队列中的。

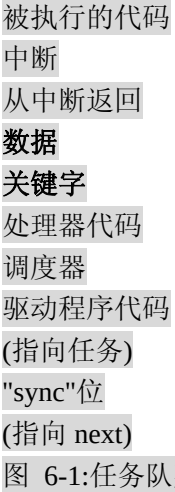

图 6-1:任务队列使用的执行流程

示例程序是如何工作的

延迟计算的示例程序是 *jiq*(Just In Queue)模块，本节中抽取了它的部分源码。该模块创建*/proc* 文件，可以用 *dd* 或者其他工具来读；这与 *jit* 模块很相似。该示例程序使用了动态*/proc* 文件因此不能在 Linux1.2 上运行。

读*jiq*文件的进程进入睡眠状态直到缓冲区满*。缓冲区由不断运行的任务队列来填充。任务队列的每遍运行都将在要填充的缓冲区中添加一个字符串；该字符串记录了当前时间(jiffies值)，该遍的current进程和intr_count值。

该/proc 文件最好是用 dd count=1 命令一次性地读进来；如果你用 *cat* 命令，*read* 方法要被多次调用，输出结果会有重迭，详情可见第 4 章“*调试技术*”的“使用/proc 文件系统”一节。

填充缓冲区的代码都在 *jiq_print* 函数中，任务队列的每遍运行都要调用它。打印函数没什么意思，不在这里列出；我们还是来看看插入队列的任务的初始化代码：

```
struct tq_struct jiq_task; /*  全局变量；初始化为零  */

/*  该行在 init_module()中  */
jiq_task.routine = jiq_print;
jiq_task.data = (void *)&jiq_data;
```

这里没必要清零 **jiq_task** 结构变量的 **sync** 域和 **next** 域，因为静态变量已由编译器初始化为零了。

调度器队列

最容易使用的任务队列是 **tq_scheduler** 队列，因为该队列中的任务不会在中断时间内运行，因此少了很多限制。

/proc/jiqsched 文件是使用 **tq_scheduler** 队列的示例文件。该文件的 *read* 函数以如下的方式将任务 jiq_task 放进 **tq_scheduler** 队列中：

```
/*
 *  使用调度器队列的例子    -- /proc/jiqsched
 */
int jiq_read_sched(char *buf, char **start, off_t offset, int len, int unused)
{
      jiq_data.len = 0;                   /*  还未打印，长度为 0 */
      jiq_data.buf = buf;                 /*  打印到这个缓冲区中  */
```

* */proc*文件的缓冲区是内存中的一页：4KB或 8KB。

```
        jiq_data.jiffies = jiffies;         /*  开始时间  */

        /* jiq_print 会调用  queue_task()  使自己重新进入  jiq_data.queue 队列  */
        jiq_data.queue = &tq_scheduler;

        queue_task(&jiq_task, &tq_scheduler); /*  准备运行*/
        interruptible_sleep_on(&jiq_wait);       /*  进入睡眠队列只到任务完成  */

        return jiq_data.len;
}
```

读读*/proc/jiqsched* 文件很有意思，因为它显示调度器在何时运行－**jiffies** 值表明调度器激活的时间。如果系统中有些正在占用 CPU 的进程，那么队列中各任务的运行间会有些延迟；因为调度器要在若干时钟滴答后才会抢先那些进程。打开这个文件会花上好几秒钟，因为它长达 100 行(在 Alpha 机器上是 200 行)。

测试这些情形最简单方法是跑一个执行空循环的进程。*load50* 是个增加机器负载的程序，它在用户空间执行 50 个并发的忙循环；你可以在示例程序中找到它的源码(*misc-progs/load50.c*)。当在系统中运行 *load50* 程序时，*head* 命令将从*/proc/jiqsched* 文件中抽取类似如下的信息：

```
time      delta       intr_count pid command
1643733     0             0       701 head
1643747    14             0       658 load50
1643747     0             0         3 kswapd
1643755     8             0       655 load50
1643761     6             0       666 load50
1643764     3             0       650 load50
1643767     3             0       661 load50
1643769     2             0       659 load50
1643769     0             0         6 loadmonitor
```

注意到调度队列是在进入 *schedule* 过程后就执行的，因此 **current** 进程就是刚刚被调度出去的进程。这就是为什么*/proc/jiqsched* 文件的第一行总是读该文件的那个进程；它正进入睡眠状态，就要被调出。还可以发现，*kswapd* 和 *loadmonitor*(这是我在我的系统上运行的一个程序)的执行时间都少于 1 个时钟滴答，而 *load50* 是在它的时间片耗尽后被抢先，这离它获得处理器有好几个时钟滴答。

当系统中实际上没有任何进程在运行时，current 进程总是空闲(idle)任务(0 号进程，历史性的被称为"swapper")，任务队列或者是不断地运行或者是每隔 1 个时钟滴答运行一次。如果处理器不能进入“暂停”("halted")状态，调度器和任务队列就将不断运行；如果处理器能被 0 号进程暂停(halt)，它们每隔 1 个时钟滴答才运行一次。暂停的(halted)处理器只能由中断唤醒。当中断发生时，空闲进程运行调度器(及相应的队列)。下面显示了在没有负载的系统运行运行命令 *head /proc/jiqsched* 得到的结果：

time	delta	intr_count	pid	command
1704475	0	0	730	head
1704476	1	0	0	swapper
1704477	1	0	0	swapper
1704478	1	0	0	swapper
1704478	0	0	6	loadmonitor
1704479	1	0	0	swapper
1704480	1	0	0	swapper
1704481	1	0	0	swapper
1704482	1	0	0	swapper

定时器队列

定时器队列的使用方法和调度器队列差不多。和调度器队列不同的是，定时器队列是在中断时间内执行的。另外，该队列一定在下一个时钟滴答被运行，因此就与系统负载无关了。下面是在我的系统在跑编译程序时运行命令 *head /proc/jiqtimer* 输出的结果：

time	delta	intr_count	pid	command
1760712	1	1	945	cc1
1760713	1	1	945	cc1
1760714	1	1	945	cc1
1760715	1	1	946	as
1760716	1	1	946	as
1760717	1	1	946	as
1760718	1	1	946	as
1760719	1	1	946	as
1760720	1	1	946	as

当前的任务队列实现有一个特性，一个任务可以将自己重新插回到它原先所在的队列。例如，定时器队列中的任务可以在运行时将自己插回到定时器队列中去，从而在下一个时钟滴答又再次被运行。在处理任务队列之前，由于是先用 **NULL** 指针替换队列的头指针，因此才可能进行不断的重新调度。这种实现可追溯到 1.3.70 版的内核。在早期的版本中(象 1.2.13 版)，重调度是不可能的，因为内核在运行队列前不会先整理它。在 1.2 版的 Linux 中试图重新调度任务会因进入死循环(tight loop)而挂起系统。是否具有重新调度的能力是任务队列实现上 1.2.13 版和 2.0.x 各版本间唯一的一处差别。

尽管一遍遍地重新调度同一个任务看起来似乎没什么意义，但有时这也有些用处。例如，我的计算机就是在到达目的地之前让任务在定时器队列上不断地重新调度自己，来实现步进马达的一步步移动的。其他的例子还有 *jiq* 模块，该模块中的打印函数都又重新调度了自己来显示对任务队列一遍扫描的结果。

立即队列

最后一个可由模块代码使用的预定义队列是立即队列。它和下半部中断处理程序工作机制相似，因此必须用 **mark_bh(IMMEDIATE_BH)**标记该处理程序是活跃的。出于高效的目的，只有被标记为活跃的下半部处理程序才会被执行。注意必须在调用 *queue_task 后*才能标记下半部处理程序，否则会带来竞争。详情见第 9 章的“下半部处理”一节。

立即队列是系统处理得最快的队列－在 **intr_count** 变量加 1 之后马上执行。该队列执行得如此“立即”以致于当你重新注册你的任务之后，它一返回就立即重新运行。该队列一遍遍执行直到队列为空。只要看看*/proc/jiqimmed* 文件，就会明白它执行得如此只快的原因，在整个读过程中它完全控制了 CPU。

立即队列是由调度器执行的或者是在一个进程从系统调用返回时被执行的。值得注意的是，调度器(至少对于 2.0 版的内核)并不总会一直处理立即队列到它为空；只有从系统调用返回时才会这么做。这可以从下面的示例输出看出来－只有 *jiqimmed* 文件的第一行是当前进程 *head*，而下面各行都不是了。

```
  time      delta      intr_count pid command
1975640     0             1      1060 head
1975641     1             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
1975641     0             1         0 swapper
```

显然该队列不能用于延迟任务的执行－它是个“立即”队列。相反，它的目的是使任务尽快地得以执行，但是要在“安全的时间”内。这对中断处理非常有用，因为它提供在实际的中断处理程序之外执行处理程序代码的一处入口。

尽管*/proc/jiqimmed* 将任务重新注册到立即队列中，但这种技术在实际的实现代码中并不鼓励；象这种不肯合作的行为会在整个不断重等记的过程中独占住处理器，那还不如将整个任务一次性地完成。

运行自己定制的工作队列

声明新的任务队列不困难。驱动程序可以随意地声明任意多的新任务队列；这些队列的使用和 **tq_scheduler** 队列差不多。

与预定义队列不同的，内核不会自动处理定制的任务队列。定制的任务队列要由程序员自己维护。

下面的宏声明一个定制队列，在你需要任务队列声明处这个宏会被扩展：

```
DECLARE_TASK_QUEUE(tq_custom);
```

声明完队列，就可以调用下面的函数对任务进行排队。上面的宏和下面的调用相匹配：

```
queue_task(&custom_task,&tq_custom);
```

然后就可以调用下面的函数运行 **tq_custom** 队列了：

```
run_task_queue(&tq_custom);
```

如果现在你想测试你定制的任务队列，你可以在预定义的队列中注册一个函数来处理这个队列。尽管看起来象绕了弯路，但其实并非如此。如果你希望累积任务以便同时得到执行，尽管需要用另一个队列来决定这个“同时”，定制的任务队列还是非常有用的。

内核定时器

内核中最终的计时资源还是定时器。定时器用于调度函数(定时器处理程序)在未来某个特定时间执行。与任务队列不同，你可以指定你的函数在未来*何时*被调用，但你不能确定任务队列中的任务何时执行。另外，内核定时器与任务队列相似的，内核定时器注册的处理函数只执行一次－定时器不是循环执行的。

有时你要执行的操作不在任何进程上下文内，比如关闭软驱马达和中止某个耗时的关闭操作。在这些情况下，延迟从 *close* 调用的返回对应用程序不公平。而且这时也没有必要使用任务队列，因为队中的任务在估算时间的同时还要不断重新注册自己。

这里用定时器就更方便。你注册你的处理函数一次，当定时器超时后内核就调用它一次。这种处理一般较适合由内核完成，但有时驱动程序也需要，就象软驱马达的例子。

Linux 使用了两种定时器，所谓的“旧定时器”和新定时器。在介绍如何使用更好的新定时器前，我先简要介绍一下旧定时器。新定时器，实际上并不新；它们在 1.0 版之前的 Linux 里就已经引入了。

旧定时器包括 32 个静态的定时器。它们的存在只是出于兼容性的考虑(因为替换旧定时器需要修改和测试大量的驱动程序代码)。

旧定时器的数据结构包括一个标明活动的定时器的位屏蔽码和定时器数组，数组的每个成员又包括一个处理程序和该定时器的超时值。旧的定时器结构的主要问题在于，每个需要定时器来延迟操作的设备都要静态地分配给一个定时器。

这种实现在几年前是可以接受的，当时支持的设备(因此需要的定时器)还很有限，但对当前的 Linux 版本就不够了。

我不再介绍如何使用旧的定时器；我在这里提到它们只是为了满足那些好奇的读者。

新的定时器列表

新的定时器被组织成双向链接表。这意味着你加入任意多的定时器。定时器包括它的timeout(超时)值(单位是 jiffies)和超时时调用的函数。定时器处理程序需要一个参数，该参数和处理程序函数指针本身一起存放在一个数据结构中。

定时器列表的数据结构如下，抽取自头文件**<linux/timer.h>**:

```
struct timer_list {
        struct timer_list *next;   /*不要直接修改它 */
        struct timer_list *prev; /*不要直接修改它 */
        unsigned long expires; /* timeout 超时值，以 jiffies 值为单位 */
        unsigned long data; /* 传递给定时器处理程序的参数 */
        void (*function)(unsigned long); /* 超时时调用的定时器处理程序 */
};
```

可以看到，定时器的实现和任务队列有所不同，尽管两个表的表项有些相似。这两个数据结构是由两个编程者几乎在同时创建的，因此不大一样；它们并没有相互复制。所以，定时器处理程序的形参是 **unsigned long** 类型，而不是 **void *** 类型，而定时器处理程序的名字是 function 而非 routine。

定时器的 timeout 值是个"jiffy"值，当 **jiffies** 值大于等于 **timer->expires** 时，就要运行 **timer->function** 函数。Timeout 值是个绝对数值；它与当前时间无关，不需要更新。

一初始化完 **timer_list** 结构，*add_timer* 函数就将它插入一张有序表中，该表每秒钟会被查询 100 次左右(尽管时钟滴答的频率有时要比这高，但这样节省 CPU 时间)。

简单说，操作定时器的有如下函数：

void init_timer(struct timer_list *timer);

该内联函数用来初始化新定时器队列结构。目前，它只将 **prev** 和 **next** 指针清零。建议程序员使用该函数来初始化定时器而不要显式地修改对结构内的指针，以保证向前兼容。

void add_timer(struct timer_list * timer);

该函数将定时器插入挂起的定时器的全局队列。有意思的是，内核定时器最初的实现和现在的实现不同；在 1.2 版的内核中，*add_timer* 函数认为 **timer->expires** 值是相对于当前的 jiffy 值的，所以它在将结构插入全局列表前会先将 jiffies 值加到 timer->expires 上。这个不兼容在示例源代码文件 *sysdep.h* 中得到处理。

int del_timer(struct timer_list *timer);

如果需要在定时器超时前将它从列表中删除，应调用 *del_timer* 函数。但当定时器超时是，系统会自动地将它从列表中删除。

使用定时器的一个例子是 *jiq* 示例模块。*/proc/jittimer* 文件使用一个定时器来产生两行数据；print 函数和前面任务队列用的是同一个。第一行数据是调用 *read* 产生的，而第二行是 100 jiffies 后定时器处理函数打印出的。

/proc/jittimer 文件的代码如下：

```
struct timer_list jiq_timer;

void jiq_timedout(unsigned long ptr)
{
    jiq_print((void *)ptr);                 /*  打印一行数据  */
    wake_up_interruptible(&jiq_wait);   /*  唤醒进程  */
}

int jiq_read_run_timer(char *buf, char **start, off_t offset,
                       int len, int unused)
{

    jiq_data.len = 0;               /*  准备传递给 jiq_print()函数的各个参数  */
    jiq_data.buf = buf;
    jiq_data.jiffies = jiffies;
    jiq_data.queue = NULL;          /*  不会重新进入队列  */

    init_timer(&jiq_timer);                 /*  初始化定时器结构  */
    jiq_timer.function = jiq_timedout;
    jiq_timer.data = (unsigned long)&jiq_data;
    jiq_timer.expires = jiffies + HZ; /* 1 秒  */

    jiq_print(&jiq_data);     /*  打印并进入睡眠  */
    add_timer(&jiq_timer);
    interruptible_sleep_on(&jiq_wait);
    return jiq_data.len;
}
```

运行命令 **head /proc/jitimer** 得到如下输出结果：

```
   time       delta      intr_count    pid    command
2121704         0           0          1092    head
2121804       100           1             0    swapper
```

很明显地，从第 2 行的 **intr_count** 变量的值可以发现定时器程序是在“中断时”运行的。

可能看起来有点奇怪的是，定时器总是可以正确地超时，即使处理器正在执行系统调用。我在前面曾提到，运行在内核态的进程不会被调出；但时钟中断是个例外，它与当前进程无关。你可以试试在前台同时读*/proc/jitbusy* 文件和*/proc/jittimer* 文件，这时尽管看起来系统似乎被忙等待的系统调用给锁死住了，但定时器队列和内核定时器还是能不断得到处理的。

快速参考

本章引入如下符号：

#include <linux/param.h>
HZ

HZ 符号指出每秒钟产生的时钟滴答数。

volatile unsigned long jiffies

jiffies 变量每个时钟滴答后加 1；因此它每秒增加 1 个 **HZ**。

#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);

该函数返回当前时间。1.2 版的内核并不提供。

#include <linux/delay.h>
void udelay(unsigned long usecs);

udelay 函数延迟整数数目的微秒数，但不应超过 1 毫秒。

#include <linux/tqueue.h>
void queue_task(struct tq_struct *task, task_queue *list);
void queue_task_irq();
void queue_task_irq_off();

这些函数注册延迟执行的任务。第一个函数，*queue_task*，总是可以被调用；第二个函数只能在不可重入的函数内被调用，而最后一个函数只有在关闭中断后才能被调用。新近的内核只提供第一种函数接口了(见第 17 章的“任务队列”一节)。

void run_task_queue(task_queue *list);

该函数运行任务队列。

task_queue tq_immediate, tq_timer, tq_scheduler;

这些预定义的任务队列在每个时钟滴答后并在内核调度新的进程前尽快地分别得到执行。

#include <linux/timer.h>
void init_timer(struct timer_list *timer);

该函数初始化新分配的定时器队列。

void add_timer(struct timer_list * timer);

该函数将定时器插入待处理的定时器的全局队列。

int del_timer(struct timer_list *timer);

del_timer 函数将定时器从挂起的定时器队列中删除。如果队列中有该定时器，*del_timer* 返回 1，否则返回 0。