第七章 时间,延时和延后工作

在这个阶段，我们已经知道如何编写一个全功能的字符模块。然而，现实世界中的驱动程序需要做的不仅仅是实现控制设备的操作，它们还需要处理诸如时间、内存管理、硬件访问等问题。幸运的是，内核提供了许多设施来简化驱动程序编写者的任务。在接下来的几章中，我们将描述你可以使用的一些内核资源。本章将首先描述如何处理时间问题。处理时间涉及到以下任务，它们的复杂性是递增的：

- 测量时间间隔和比较时间
- 知道当前的时间
- 延迟一定时间的操作
- 安排异步函数在稍后的时间执行

7.1. 测量时间流失Lapses

内核通过定时器中断来跟踪时间的流动。中断在第10章中有详细的介绍。

定时器中断是由系统的计时硬件在规定的间隔内生成的；这个间隔在启动时由内核根据HZ的值进行编程，HZ是在<linux/param.h>或其包含的子平台文件中定义的依赖于架构的值。在分发的内核源代码中，默认值从真实硬件的每秒50到1200次，到软件模拟器的24次。大多数平台每秒运行100或1000次中断；流行的x86 PC默认为1000，尽管在以前的版本中（包括2.4及以前的版本）它曾经是100。作为一般规则，即使你知道HZ的值，你在编程时也永远不应该依赖于那个特定的值。

对于那些希望系统有不同的时钟中断频率的人来说，可以改变HZ的值。如果你在头文件中改变HZ，你需要用新的值重新编译内核和所有模块。如果你愿意为了实现你的目标而支付额外的定时器中断的开销，你可能希望提高HZ以获得更细粒度的异步任务解决方案。实际上，将HZ提高到1000在使用2.4或2.2版本内核的x86工业系统中相当常见。然而，对于当前的版本，处理定时器中断的最佳方法是保持HZ的默认值，因为我们完全信任内核开发者，他们肯定选择了最佳的值。此外，一些内部计算目前只针对HZ在12到1535范围内的值进行（参见<linux/timex.h>和RFC-1589）。

每次定时器中断发生时，内核内部计数器的值都会增加。计数器在系统启动时初始化为0，所以它代表了自上次启动以来的时钟滴答数。计数器是一个64位变量（即使在32位架构上也是如此），称为jiffies_64。然而，驱动程序编写者通常访问jiffies变量，这是一个无符号的长整数，它与jiffies_64或其最低有效位相同。通常更喜欢使用jiffies，因为它更快，而且对64位jiffies_64值的访问在所有架构上都不一定是原子的。

除了低分辨率的内核管理的jiffy机制外，一些CPU平台还具有软件可以读取的高分辨率计数器。尽管它的实际用途在各个平台上有所不同，但有时它是一个非常强大的工具。

7.1.1. 使用 jiffies 计数器

在 `<linux/jiffies.h>` 中，有一个计数器和一些用于读取它的实用函数，尽管你通常只需要包含 `<linux/sched.h>`，它会自动引入 jiffies.h。不用说，jiffies 和 jiffies_64 都应被视为只读。

当你的代码需要记住当前的 jiffies 值时，它可以直接访问这个声明为 volatile 的 unsigned long 变量，这告诉编译器不要优化内存读取。当你的代码需要计算未来的时间戳时，你需要读取当前的计数器，如下例所示：

```c
#include <linux/jiffies.h>

unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies;                          /* 读取当前值 */

stamp_1    = j + HZ;                  /* 未来的1秒 */

stamp_half = j + HZ/2;                /* 半秒后 */

stamp_n    = j + n * HZ / 1000;       /* n毫秒后 */
```

这段代码即使在 jiffies 回绕的情况下也没有问题，只要以正确的方式比较不同的值。尽管在 HZ 为 1000 的 32 位平台上，计数器只在每 50 天回绕一次，但你的代码应该准备好面对这种情况。要比较你缓存的值（如上面的 stamp_1）和当前值，你应该使用以下的宏之一：

```c
#include <linux/jiffies.h>

int time_after(unsigned long a, unsigned long b);

int time_before(unsigned long a, unsigned long b);

int time_after_eq(unsigned long a, unsigned long b);

int time_before_eq(unsigned long a, unsigned long b);
```

第一个宏在 a（作为 jiffies 的快照）表示的时间在 b 之后时返回 true，第二个宏在 a 的时间在 b 之前时返回 true，最后两个宏分别用于比较“在之后或等于”和“在之前或等于”。这段代码通过将值转换为 signed long，然后相减并比较结果来工作。如果你需要以安全的方式知道两个 jiffies 实例之间的差异，你可以使用同样的技巧：`diff = (long)t2 - (long)t1;`。

你可以通过以下方式将 jiffies 差异转换为毫秒： msec = diff * 1000 / HZ; 然而，有时你需要与用户空间程序交换时间表示，这些程序倾向于使用 struct timeval 和 struct timespec 来表示时间值。这两种结构都使用两个数字来表示精确的时间量：在较旧且常用的 struct timeval 中使用秒和微秒，在较新的 struct timespec 中使用秒和纳秒。内核提供了四个辅助函数，用于将以 jiffies 表示的时间值转换为这些结构，并从这些结构转换回来：

```c
#include <linux/time.h>

unsigned long timespec_to_jiffies(struct timespec
*value);

void jiffies_to_timespec(unsigned long jiffies, struct
timespec *value);

unsigned long timeval_to_jiffies(struct timeval *value);

void jiffies_to_timeval(unsigned long jiffies, struct
timeval *value);
```

访问64位的 jiffy 计数并不像访问 jiffies 那样简单。在64位计算机架构上，这两个变量实际上是一体的，但对于32位处理器，访问这个值并不是原子的。这意味着如果在你读取它们时，变量的两个部分都被更新，你可能会读取到错误的值。你极不可能需要读取64位计数器，但如果你确实需要，你会很高兴知道内核提供了一个特定的辅助函数，它为你做了适当的锁定：

```c
#include <linux/jiffies.h>

u64 get_jiffies_64(void);
```

在上述原型中，使用了 u64 类型。这是由 <linux/types.h> 定义的类型之一，表示一个无符号的64位类型。

如果你想知道32位平台是如何同时更新32位和64位计数器的，可以阅读你的平台的链接器脚本（查找一个名字匹配 vmlinux.*lds* 的文件）。在那里，jiffies 符号被定义为根据平台是小端还是大端来访问64位值的最低有效字。实际上，同样的技巧也用于64位平台，以便在同一地址访问 unsigned long 和 u64 变量。

最后，请注意，实际的时钟频率几乎完全对用户空间隐藏。当用户空间程序包含 param.h 时，HZ 宏总是扩展为100，每个报告给用户空间的计数器都相应地进行转换。这适用于 clock(3)、times(2) 和任何相关函数。用户可以获取 HZ 的唯一证据是定

时器中断发生的速度，如 /proc/interrupts 所示。例如，你可以通过将这个计数除以系统在 /proc/uptime 报告的运行时间来获得 HZ。

7.1.2. 处理器特定的寄存器

如果你需要测量非常短的时间间隔，或者你需要极高的精度，你可以使用依赖于特定平台的资源，这是选择精度优先于可移植性。

在现代处理器中，对实证性能数据的迫切需求被大多数CPU设计中指令计时的固有不可预测性所阻碍，这是由于缓存内存、指令调度和分支预测。作为回应，CPU制造商引入了一种计算时钟周期的方法，作为一种简单可靠的测量时间间隔的方式。因此，大多数现代处理器都包含一个计数器寄存器，每个时钟周期都会稳定地增加一次。如今，这个时钟计数器是执行高分辨率计时任务的唯一可靠方式。

具体细节会因平台而异：寄存器可能可以或不能从用户空间读取，可能可以或不能写入，可能是64位或32位宽。在后一种情况下，你必须准备好处理溢出，就像我们处理jiffy计数器一样。对于你的平台，寄存器可能甚至不存在，或者如果CPU缺少这个功能，你正在处理一个特殊用途的计算机，硬件设计师可以在外部设备中实现它。

无论寄存器是否可以清零，我们强烈建议不要重置它，即使硬件允许。毕竟，你可能不是任何给定时间的计数器的唯一用户；例如，在一些支持SMP的平台上，内核依赖于这样一个计数器在处理器之间同步。由于你总是可以测量值之间的差异，只要那个差异不超过溢出时间，你就可以在不修改其当前值的情况下完成工作，而无需独占寄存器。

最著名的计数器寄存器是TSC（时间戳计数器），它在Pentium和所有之后的CPU设计中被引入到x86处理器中，包括x86_64平台。它是一个64位的寄存器，计数CPU时钟周期；它可以从内核空间和用户空间读取。
在包含了<asm/msr.h>（一个x86特定的头文件，其名称代表“机器特定寄存器”）后，你可以使用以下宏之一：

```c
rdtsc(low32,high32);

rdtscl(low32);

rdtscll(var64);
```

第一个宏原子地将64位值读入两个32位变量；下一个宏（“读取低半部分”）将寄存器的低半部分读入一个32位变量，丢弃高半部分；最后一个宏将64位值读入一个long long变量，因此得名。所有这些宏都将值存储到它们的参数中。

对于TSC的最常见用途，读取计数器的低半部分就足够了。一个1-GHz的CPU每4.2秒只会溢出一次，所以如果你正在对可靠地花费较少时间的时间间隔进行基准测试，你不需要处理多寄存器变量。然而，随着CPU频率的提高和计时要求的增加，你将来可能需要更频繁地读取64位计数器。

作为一个只使用寄存器低半部分的例子，以下几行测量了指令本身的执行：

```c
unsigned long ini, end;

rdtscl(ini); rdtscl(end);

printk("time lapse: %li\n", end - ini);
```

其他一些平台提供了类似的功能，内核头文件提供了一个你可以用来代替rdtsc的架构独立函数。它被称为get_cycles，定义在<asm/timex.h>（被<linux/timex.h>包含）。它的原型是：

```c
#include <linux/timex.h>

cycles_t get_cycles(void);
```

这个函数为每个平台定义，它总是在没有周期计数器寄存器的平台上返回0。cycles_t类型是一个适当的无符号类型，用来保存读取的值。
尽管有独立于架构的函数可用，但我们想借此机会展示一段内联汇编代码的例子。为此，我们为 MIPS 处理器实现了一个与 x86 处理器工作方式相同的 rdtscl 函数。

我们选择 MIPS 作为例子，因为大多数 MIPS 处理器都有一个 32 位的计数器，作为他们内部“协处理器 0”的寄存器 9。要访问这个只能从内核空间读取的寄存器，你可以定义以下宏，它执行一个“从协处理器 0 移动”的汇编指令：

```c
#define rdtscl(dest) \
    __asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

有了这个宏，MIPS 处理器就可以执行与 x86 处理器相同的代码了。

在 gcc 的内联汇编中，通用寄存器的分配由编译器完成。刚才显示的宏使用 %0 作为“参数 0”的占位符，后面指定为“任何用作输出(=)的寄存器(r)”。宏还指出，输出寄存器必须对应 C 表达式 dest。内联汇编的语法非常强大，但有些复杂，尤其是对于每个寄存器能做什么有限制的架构（即 x86 系列）。语法在 gcc 文档中有描述，通常可以在 info 文档树中找到。

这一节中显示的短 C 代码片段已在 K7 类 x86 处理器和 MIPS VR4181（使用刚才描述的宏）上运行。前者报告的时间间隔为 11 个时钟周期，后者只有 2 个时钟周期。这个小数字是预期的，因为 RISC 处理器通常每个时钟周期执行一条指令。

关于时间戳计数器，还有一件值得知道的事：在 SMP 系统中，它们不一定在各个处理器之间同步。为了确保得到一致的值，你应该禁用查询计数器的代码的抢占。

7.2. 获知当前时间

内核代码总是可以通过查看 jiffies 的值来获取当前时间的表示。通常，这个值只表示自上次启动以来的时间对驱动程序来说并不重要，因为它的生命周期限于系统的运行时间。如所示，驱动程序可以使用 jiffies 的当前值来计算事件之间的时间间隔（例如，在输入设备驱动程序中区分双击和单击，或计算超时）。简而言之，当你需要测量时间间隔时，查看 jiffies 几乎总是足够的。如果你需要对短时间间隔进行非常精确的测量，特定于处理器的寄存器可以提供帮助（尽管它们带来了严重的可移植性问题）。

驱动程序很少需要知道以月、日、小时表示的挂钟时间；这种信息通常只有用户程序（如 cron 和 syslogd）才需要。处理现实世界的时间通常最好留给用户空间，那里的 C 库提供了更好的支持；此外，这样的代码通常与策略相关，不应该在内核中。然而，有一个内核函数可以将挂钟时间转换为 jiffies 值：

```c
#include <linux/time.h>

unsigned long mktime (unsigned int year, unsigned int
mon,

                      unsigned int day, unsigned int
hour,

                      unsigned int min, unsigned int
sec);
```

再次强调：在驱动程序中直接处理挂钟时间通常是实施策略的标志，因此应该受到质疑。

虽然你不需要处理人类可读的时间表示，但有时你需要在内核空间处理绝对时间戳。为此，<linux/time.h> 导出了 do_gettimeofday 函数。当调用它时，它会填充一个 struct timeval 指针——与 gettimeofday 系统调用中使用的相同——用秒和微秒值。do_gettimeofday 的原型是：

```c
#include <linux/time.h>

void do_gettimeofday(struct timeval *tv);
```

源代码指出，do_gettimeofday 有“近微秒的分辨率”，因为它询问计时硬件当前 jiffy 已经过去的部分。然而，精度从一个架构到另一个架构是不同的，因为它取决于实际使用的硬件机制。例如，一些 m68knommu 处理器、Sun3 系统和其他 m68k 系统不能提供比 jiffy 更高的分辨率。另一方面，Pentium 系统通过读取本章前面描述的时间戳计数器，提供非常快速和精确的子刻度测量。

当前时间也可以从 xtime 变量（一个 struct timespec 值）获取，尽管粒度是 jiffy。不鼓励直接使用这个变量，因为同时访问所有字段是困难的。因此，内核提供了实用函数 current_kernel_time：

```c
#include <linux/time.h>

struct timespec current_kernel_time(void);
```

在 jit（“即时”）模块中，可以通过各种方式获取当前时间的代码，在 O'Reilly 的 FTP 站点提供的源文件中。jit 创建了一个名为 /proc/currentime 的文件，当读取时，它以 ASCII 返回以下项目：

- 当前的 jiffies 和 jiffies_64 值作为十六进制数字
- 由 do_gettimeofday 返回的当前时间
- 由 current_kernel_time 返回的 timespec

我们选择使用动态的 /proc 文件来将样板代码降到最低——为了返回一点文本信息，没有必要创建一个完整的设备。

只要模块被加载，文件就会连续返回文本行；每个 read 系统调用都会收集并返回一组数据，为了更好的可读性，这些数据被组织在两行中。当你在一个定时器 tick 之内读取多个数据集时，你会看到 do_gettimeofday（它查询硬件）和其他只在定时器 tick 时更新的值之间的差异。

```c
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
 1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
 1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
 1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
 1062370899.630161336
```

在上面的屏幕截图中，有两个值得注意的地方。首先，尽管 current_kernel_time 的值以纳秒表示，但只有时钟 tick 的粒度；do_gettimeofday 一致地报告更晚的时间，但不晚于下一个定时器 tick。其次，64 位的 jiffies 计数器在上面 32 位字的最低有效位设置了。这是因为在启动时用来初始化计数器的 INITIAL_JIFFIES 的默认值，会在启动

后几分钟强制低字溢出，以帮助检测与该溢出相关的问题。计数器的这种初始偏差没有影响，因为 jiffies 与挂钟时间无关。在 /proc/uptime 中，内核从计数器中提取运行时间，转换之前会去除初始偏差。

7.3. 延后执行

设备驱动程序经常需要延迟执行一段时间的特定代码，通常是为了让硬件完成某项任务。在这一部分，我们将介绍实现延迟的多种不同技术。每种情况的具体环境决定了哪种技术最好使用；我们会逐一介绍它们，并指出每种的优点和缺点。

需要考虑的一个重要事项是，你需要的延迟与时钟 tick 的比较，考虑到在各种平台上 HZ 的范围。那些可靠地比时钟 tick 长，并且不受其粗粒度影响的延迟，可以使用系统时钟。非常短的延迟通常必须用软件循环来实现。在这两种情况之间存在一个灰色地带。在这一章中，我们使用“长”延迟这个短语来指代多个 jiffy 的延迟，这在某些平台上可能低至几毫秒，但对于 CPU 和内核来说仍然很长。

以下各节通过从各种直观但不适当的解决方案到正确的解决方案的相当长的路径，讨论了不同的延迟。我们选择这条路径，因为它允许更深入地讨论与计时相关的内核问题。如果你急于找到正确的代码，只需浏览这一部分。

- 在 Linux 内核驱动程序中实现延迟执行代码的常见方法有以下几种：

1. **使用 mdelay 或 udelay 函数**：这两个函数分别提供毫秒级和微秒级的延迟。它们是忙等待（busy-waiting）的形式，这意味着它们会阻塞当前的执行直到指定的时间过去。这些函数应该只在需要非常短的延迟时使用，因为它们会阻塞 CPU。

```c
#include <linux/delay.h>

void mdelay(unsigned long msecs);

void udelay(unsigned long usecs);
```

2. **使用 msleep 或 usleep_range 函数**：这些函数提供了一种更有效的方式来实现延迟，它们会将 CPU 交给其他进程使用。msleep 提供毫秒级的延迟，usleep_range 则提供微秒级的延迟。

```c
#include <linux/delay.h>

void msleep(unsigned int msecs);

void usleep_range(unsigned long min, unsigned long max);
```

3. **使用 schedule_timeout 函数**：这个函数可以用来实现 jiffies 级别的延迟。它会将当前进程设置为可中断的睡眠状态，并在指定的 jiffies 数量过去后唤醒它。

```c
#include <linux/sched.h>

long schedule_timeout(long timeout);
```

在选择使用哪种方法时，需要考虑你的延迟需求与时钟 tick 的关系，以及你是否可以接受阻塞 CPU。

7.3.1. 长延时

偶尔，驱动程序需要延迟执行相对较长的时间——超过一个时钟 tick。有几种方法可以实现这种延迟；我们从最简单的技术开始，然后进行更高级的技术。

7.3.1.1. 忙等待

如果你想通过时钟 tick 的倍数来延迟执行，并允许一些松散的值，最简单（尽管不推荐）的实现是一个监视 jiffy 计数器的循环。忙等待的实现通常看起来像下面的代码，其中 j1 是延迟到期时 jiffies 的值：

```c
while (time_before(jiffies, j1))

    cpu_relax( );
```

调用 cpu_relax 调用了一种特定于架构的方式，表示你当前并没有对处理器做太多的事情。在许多系统上，它什么都不做；在对称多线程（“超线程”）系统上，它可能会将核心让给另一个线程。无论如何，只要可能，都应该避免使用这种方法。我们在这里展示它，因为有时你可能想运行这段代码，以更好地理解其他代码的内部结构。

那么，让我们看看这段代码是如何工作的。这个循环保证能工作，因为 jiffies 被内核头文件声明为 volatile，因此，每次一些 C 代码访问它时，都会从内存中获取它。尽管从技术上来说是正确的（即它按设计工作），但这个忙碌的循环严重降低了系统性能。如果你没有为你的内核配置抢占式操作，那么循环在延迟期间完全锁定了处理器；调度器永远不会抢占在内核空间运行的进程，计算机看起来完全死掉，直到达到 j1 的时间。如果你正在运行一个抢占式内核，问题就不那么严重了，因为，除非代码正在持有一个锁，否则可以为其他用途回收一些处理器的时间。然而，在抢占式系统上，忙等待仍然很昂贵。

更糟糕的是，如果在你进入循环时中断恰好被禁用，jiffies 就不会被更新，而 while 条件将永远为真。运行一个抢占式内核也无济于事，你将被迫按下大红按钮。

这种延迟代码的实现，就像下面的那些一样，都在 jit 模块中。模块创建的 /proc/jit* 文件每次你读取一行文本时都会延迟一整秒，每行保证有 20 个字节。如果你想测试忙等待代码，你可以读取 /proc/jitbusy，它对每行返回的内容都忙等待一秒。

建议的读取 /proc/jitbusy 的命令是 dd bs=20 < /proc/jitbusy，也可以选择指定块的数量。文件返回的每一行20字节代表了延迟前后 jiffy 计数器的值。这是在一个否则未加载的计算机上的一个样本运行。

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1686518 1687518
 1687519 1688519
 1688520 1689520
 1689520 1690520
 1690521 1691521
```

一切看起来都很好：延迟正好是一秒（1000 jiffies），下一个读系统调用在前一个结束后立即开始。但是，让我们看看在一个运行着大量 CPU 密集型进程的系统上（和非抢占式内核）会发生什么：

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1911226 1912226
 1913323 1914323
 1919529 1920529
 1925632 1926632
 1931835 1932835
```

在这里，每个读系统调用都延迟了正好一秒，但是内核可能需要超过5秒的时间才能调度 dd 进程，以便它可以发出下一个系统调用。这在多任务系统中是预期的；CPU 时间在所有运行的进程之间共享，CPU 密集型进程的动态优先级被降低。（关于调度策略的讨论超出了本书的范围。）

上面显示的在负载下的测试是在运行 load50 示例程序时进行的。这个程序 fork 出一些什么都不做，但以 CPU 密集型方式做的进程。该程序是本书附带的示例文件的一部分，默认 fork 出 50 个进程，尽管可以在命令行上指定数量。在这一章，以及在本书的其他地方，对加载系统的测试都是在一个否则空闲的计算机上运行 load50 进行的。

如果你在运行一个可抢占的内核时重复这个命令，你会发现在一个否则空闲的 CPU 上没有明显的差异，并且在负载下有以下行为。

```
phon% dd bs=20 count=5 < /proc/jitbusy
 14940680 14942777
 14942778 14945430
 14945431 14948491
 14948492 14951960
 14951961 14955840
```

在这里，一个系统调用结束和下一个开始之间没有显著的延迟，但是单个延迟远远超过一秒：在显示的示例中最多为3.8秒，并且随着时间的推移而增加。这些值表明，该进程在其延迟期间已被中断，调度其他进程。系统调用之间的间隔不是这个进程的唯一调度选项，所以那里看不到特殊的延迟。

7.3.1.2. 让出处理器

正如我们所看到的，忙等待对整个系统造成了重负；我们希望找到一种更好的技术。首先想到的改变是当我们对 CPU 不感兴趣时显式地释放它。这可以通过调用 schedule 函数来实现，该函数在 <linux/sched.h> 中声明：

```c
while (time_before(jiffies, j1)) {
    schedule( );
}
```

我们可以通过读取 /proc/jitsched 来测试这个循环，就像我们在上面读取 /proc/jitbusy 一样。然而，这仍然不是最优的。当前进程除了释放 CPU 外什么也不做，但它仍然在运行队列中。如果它是唯一可运行的进程，它实际上是在运行（它调用调度器，调度器选择相同的进程，然后调用调度器，等等）。换句话说，机器的负载（平均运行进程数）至少是一，空闲任务（进程号为 0，也被称为 swapper，出于历史原因）从不运行。虽然这个问题看起来无关紧要，但当计算机空闲时运行空闲任务可以减轻处理器的工作负担，降低其温度，延长其寿命，如果计算机恰好是你的笔记本电脑，还可以延长电池的使用时间。此外，由于进程在延迟期间实际上是在执行，所以它需要对消耗的所有时间负责。

/proc/jitsched 的行为实际上类似于在抢占式内核下运行 /proc/jitbusy。这是在一个未加载的系统上的一个样本运行：

```
phon% dd bs=22 count=5 < /proc/jitsched
   1760205    1761207
   1761209    1762211
   1762212    1763212
   1763213    1764213
   1764214    1765217
```

值得注意的是，每次读取有时会等待比请求的更多的时钟 tick。随着系统变得越来越忙，这个问题变得越来越严重，驱动程序可能会等待的时间比预期的要长。一旦进程通过 schedule 释放处理器，就不能保证进程会很快得到处理器。因此，以这种方式调用 schedule 不是对驱动程序需求的安全解决方案，而且对整个计算系统来说也是不好

的。如果你在运行 load50 时测试 jitsched，你会看到每行关联的延迟被延长了几秒，因为其他进程在超时到期时正在使用 CPU。

7.3.1.3. 超时

到目前为止，我们看到的次优延迟循环是通过观察 jiffy 计数器而不告诉任何人来工作的。但是，你可能想象，实现延迟的最好方式通常是要求内核为你做这件事。根据你的驱动程序是否在等待其他事件，有两种设置基于 jiffy 的超时的方法。

如果你的驱动程序使用等待队列来等待某个其他事件，但你也希望确保它在一定的时间内运行，它可以使用 wait_event_timeout 或 wait_event_interruptible_timeout：

```c
#include <linux/wait.h>

long wait_event_timeout(wait_queue_head_t q, condition,
long timeout);

long wait_event_interruptible_timeout(wait_queue_head_t
q,

                      condition, long timeout);
```

这些函数在给定的等待队列上睡眠，但在超时（以 jiffies 表示）到期后返回。因此，它们实现了一个有界的睡眠，不会永远进行。注意，超时值表示要等待的 jiffies 数，而不是绝对的时间值。该值由一个有符号的数字表示，因为它有时是一个减法的结果，尽管如果提供的超时是负的，函数会通过一个 printk 语句抱怨。如果超时到期，函数返回 0；如果进程被另一个事件唤醒，它返回剩余的以 jiffies 表示的延迟。返回值永远不会是负的，即使由于系统负载，延迟比预期的要大。

/proc/jitqueue 文件显示了基于 wait_event_interruptible_timeout 的延迟，尽管模块没有事件等待，并使用 0 作为条件：

```c
wait_queue_head_t wait;

init_waitqueue_head (&wait);

wait_event_interruptible_timeout(wait, 0, delay);
```

当读取 /proc/jitqueue 时，观察到的行为几乎是最优的，即使在负载下也是如此：

```
phon% dd bs=22 count=5 < /proc/jitqueue

   2027024    2028024

   2028025    2029025

   2029026    2030026

   2030027    2031027

   2031028    2032028
```

由于读取进程（上面的 dd）在等待超时时不在运行队列中，所以你看不到代码在抢占式内核中运行与否的行为差异。

wait_event_timeout 和 wait_event_interruptible_timeout 是为硬件驱动程序设计的，其中执行可以通过两种方式之一恢复：要么有人在等待队列上调用 wake_up，要么超时到期。这并不适用于 jitqueue，因为没有人在等待队列上调用 wake_up（毕竟，没有其他代码知道它），所以进程总是在超时到期时唤醒。为了适应这种情况，你想延迟执行等待没有特定的事件，内核提供了 schedule_timeout 函数，所以你可以避免声明和使用一个多余的等待队列头：

```c
#include <linux/sched.h>

signed long schedule_timeout(signed long timeout);
```

这里，timeout 是要延迟的 jiffies 数。返回值是 0，除非函数在给定的超时过去之前返回（响应一个信号）。schedule_timeout 要求调用者首先设置当前进程的状态，所以一个典型的调用看起来像这样：

```c
set_current_state(TASK_INTERRUPTIBLE);

schedule_timeout (delay);
```

前面的行（来自 /proc/jitschedto）使进程睡眠，直到给定的时间过去。由于 wait_event_interruptible_timeout 依赖于 schedule_timeout，我们不会去显示 jitschedto 返回的数字，因为它们与 jitqueue 的数字相同。再次值得注意的是，超时到期和你的进程实际被调度执行之间可能会过去额外的时间间隔。

在刚刚展示的示例中，第一行调用 set_current_state 来设置，以便调度程序不会再次运行当前进程，直到超时将其放回 TASK_RUNNING 状态。要实现不可中断的延迟，可以使用 TASK_UNINTERRUPTIBLE。如果你忘记改变当前进程的状态，调用 schedule_timeout 的行为就像调用 schedule（即，jitsched 行为），设置一个未被使用的定时器。

如果你想在不同的系统情况或不同的内核下玩弄这四个 jit 文件，或者尝试其他方式来延迟执行，你可能想在加载模块时通过设置 delay 模块参数来配置延迟的量。

7.3.2. 短延时

当设备驱动程序需要处理其硬件中的延迟时，涉及的延迟通常最多只有几十微秒。在这种情况下，依赖时钟 tick 显然不是解决之道。

内核函数 ndelay、udelay 和 mdelay 非常适合短暂的延迟，分别延迟执行指定的纳秒、微秒或毫秒数。它们的原型是：

```c
#include <linux/delay.h>

void ndelay(unsigned long nsecs);

void udelay(unsigned long usecs);

void mdelay(unsigned long msecs);
```

函数的实际实现在 <asm/delay.h> 中，是特定于架构的，有时会基于一个外部函数。每个架构都实现了 udelay，但其他函数可能会也可能不会被定义；如果它们没有被定义，<linux/delay.h> 提供了基于 udelay 的默认版本。在所有情况下，实现的延迟至少是请求的值，但可能更多；实际上，目前没有平台能够实现纳秒精度，尽管有几个平台提供了亚微秒精度。延迟超过请求的值通常不是问题，因为驱动程序中的短暂延迟通常是为了等待硬件，要求至少等待给定的时间间隔。

udelay（可能也包括 ndelay）的实现使用了一个基于启动时计算的处理器速度的软件循环，使用整数变量 loops_per_jiffy。然而，如果你想查看实际的代码，需要注意的是，x86 的实现由于使用了基于运行代码的 CPU 类型的不同的计时源，所以相当复杂。

为了避免循环计算中的整数溢出，udelay 和 ndelay 对传递给它们的值设定了上限。如果你的模块加载失败并显示一个未解析的符号，__bad_udelay，这意味着你调用了 udelay 并使用了过大的参数。然而，请注意，只有在常量值上才能执行编译时检查，并且并非所有平台都实现了它。作为一般规则，如果你试图延迟几千纳秒，你应该使用 udelay 而不是 ndelay；同样，毫秒级的延迟应该使用 mdelay 而不是更细粒度的函数。

重要的是要记住，这三个延迟函数都是忙等待；在时间间隔内，其他任务不能运行。因此，它们复制了 jitbusy 的行为，尽管在不同的规模上。因此，这些函数只应在没有实际替代方案时使用。

还有另一种实现毫秒（和更长）延迟的方法，不涉及忙等待。文件 <linux/delay.h> 声明了这些函数：

```c
void msleep(unsigned int millisecs);

unsigned long msleep_interruptible(unsigned int
millisecs);

void ssleep(unsigned int seconds)
```

前两个函数使调用进程睡眠给定的毫秒数。调用 msleep 是不可中断的；你可以确保进程至少睡眠给定的毫秒数。如果你的驱动程序正在等待队列上，你希望唤醒能打断睡眠，使用 msleep_interruptible。msleep_interruptible 的返回值通常是 0；然而，如果进程提前被唤醒，返回值是原始请求的睡眠期间剩余的毫秒数。调用 ssleep 使进程进入给定秒数的不可中断睡眠。

总的来说，如果你可以容忍比请求更长的延迟，你应该使用 schedule_timeout、msleep 或 ssleep。

7.4. 内核定时器

每当你需要安排一个稍后发生的动作，而不阻塞当前进程直到那个时间到来，内核定时器就是你的工具。这些定时器用于在未来的某个特定时间基于时钟 tick 来安排执行一个函数，可以用于各种任务；例如，当硬件不能触发中断时，通过定期检查其状态来轮询设备。

内核定时器的其他典型用途包括关闭软盘马达或完成另一个冗长的关闭操作。在这种情况下，延迟从 close 返回会对应用程序造成不必要的（并且令人惊讶的）成本。

最后，内核本身在几种情况下使用定时器，包括实现 schedule_timeout。

内核定时器是一个数据结构，指示内核在用户定义的时间执行一个用户定义的函数，带有一个用户定义的参数。实现位于 `<linux/timer.h>` 和 kernel/timer.c 中，并在“内核定时器的实现”一节中详细描述。

几乎可以肯定的是，计划运行的函数不会在注册它们的进程执行时运行。相反，它们是异步运行的。到目前为止，我们在示例驱动程序中做的所有事情都是在执行系统调用的进程的上下文中运行的。然而，当一个定时器运行时，安排它的进程可能正在睡眠，正在另一个处理器上执行，或者可能已经完全退出。

这种异步执行类似于硬件中断发生时的情况（这在第10章中详细讨论）。实际上，内核定时器是作为“软件中断”的结果运行的。在这种原子上下文中运行时，你的代码受到一些约束。定时器函数必须是原子的，就像我们在第5章的“自旋锁和原子上下文”一节中讨论的那样，但由于缺乏进程上下文，还有一些额外的问题。我们现在将介绍这些约束；它们将在后面的章节中的几个地方再次出现。重复是必要的，因为必须严格遵守原子上下文的规则，否则系统将陷入深深的麻烦。

许多操作需要进程的上下文才能执行。当你在进程上下文之外（即，在中断上下文中），你必须遵守以下规则：

- 不允许访问用户空间。因为没有进程上下文，所以没有通向任何特定进程关联的用户空间的路径。
- 在原子模式下，current 指针没有意义，不能使用，因为相关代码与被中断的进程没有关系。
- 不允许进行睡眠或调度。原子代码不能调用 schedule 或 wait_event 的形式，也不能调用任何可能睡眠的其他函数。例如，调用 kmalloc(..., GFP_KERNEL) 是违反规则的。信号量也不能使用，因为它们可以睡眠。

内核代码可以通过调用函数 in_interrupt() 来判断它是否在中断上下文中运行，该函数不接受参数，如果处理器当前在中断上下文中运行（硬件中断或软件中断），则返回非零值。

与 in_interrupt() 相关的一个函数是 in_atomic()。当不允许调度时，它的返回值是非零的；这包括硬件和软件中断上下文，以及持有自旋锁的任何时间。在后一种情况下，current 可能是有效的，但是禁止访问用户空间，因为它可能导致调度发生。每当你使用 in_interrupt() 时，你应该真正考虑是否 in_atomic() 是你实际意味着的。这两个函数都在 `<asm/hardirq.h>` 中声明。

内核定时器的另一个重要特性是，一个任务可以重新注册自己在稍后的时间运行。这是可能的，因为每个 timer_list 结构在运行之前都从活动定时器的列表中解除链接，因此，可以立即在其他地方重新链接。尽管反复调度同一个任务可能看起来是一个无意义的操作，但有时它是有用的。例如，它可以用来实现设备的轮询。

还值得知道的是，在一个 SMP 系统中，定时器函数是由注册它的同一个 CPU 执行的，以尽可能实现更好的缓存局部性。因此，重新注册自己的定时器总是在同一个 CPU 上运行。

然而，定时器的一个重要特性不应被忘记，那就是它们是竞争条件的潜在来源，即使在单处理器系统上也是如此。这是它们与其他代码异步的直接结果。因此，任何由定时器

函数访问的数据结构都应该被保护免受并发访问，要么通过成为原子类型（在第5章的“原子变量”一节中讨论），要么通过使用自旋锁（在第5章中讨论）。

7.4.1. 定时器 API

内核为驱动程序提供了一些函数来声明、注册和移除内核定时器。以下摘录显示了基本的构建块：

```c
#include <linux/timer.h>

struct timer_list {

        /* ... */

        unsigned long expires;

        void (*function)(unsigned long);

        unsigned long data;

};

void init_timer(struct timer_list *timer);

struct timer_list TIMER_INITIALIZER(_function, _expires,
_data);

void add_timer(struct timer_list * timer);

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

数据结构包含的字段比显示的要多，但这三个是意味着从定时器代码本身之外访问的。expires 字段表示定时器预期运行的 jiffies 值；在那个时候，函数 function 被调用，data 作为参数。如果你需要在参数中传递多个项目，你可以将它们捆绑为一个单一的数据结构，并传递一个转换为 unsigned long 的指针，这在所有支持的架构上都是安全的做法，并且在内存管理中非常常见（如第15章所讨论）。expires 值不是一个

jiffies_64 项目，因为定时器不预期在很远的未来到期，而且64位操作在32位平台上很慢。

在使用之前，必须初始化结构。这一步确保所有字段都被正确设置，包括对调用者不透明的那些字段。可以通过调用 init_timer 或将 TIMER_INITIALIZER 分配给静态结构来执行初始化，根据你的需要。初始化后，你可以在调用 add_timer 之前更改三个公共字段。要在定时器到期之前禁用已注册的定时器，调用 del_timer

jit 模块包含一个样本文件 `/proc/jitimer`（表示“刚好在定时器”），它返回一行头部行和六行数据行。数据行代表代码正在运行的当前环境；第一行是由读文件操作生成的，其他的由定时器生成。以下输出是在编译内核时记录的：

```
phon% cat /proc/jitimer

   time    delta  inirq    pid    cpu command

 33565837     0      0      1269    0   cat

 33565847    10      1      1271    0   sh

 33565857    10      1      1273    0   cpp0

 33565867    10      1      1273    0   cpp0

 33565877    10      1      1274    0   cc1

 33565887    10      1      1274    0   cc1
```

在这个输出中，time 字段是代码运行时 jiffies 的值，delta 是自上一行以来 jiffies 的变化，inirq 是 in_interrupt 返回的布尔值，pid 和 command 指的是当前进程，cpu 是正在使用的 CPU 的数量（在单处理器系统上总是 0）。

如果你在系统未加载时读取 `/proc/jitimer`，你会发现定时器的上下文是进程 0，即空闲任务，主要出于历史原因被称为“swapper”。

用于生成 `/proc/jitimer` 数据的定时器默认每 10 jiffies 运行一次，但你可以通过在加载模块时设置 tdelay（定时器延迟）参数来更改该值。

以下代码摘录显示了与 jitimer 定时器相关的 jit 部分。当一个进程试图读取我们的文件时，我们如下设置定时器：

```c
unsigned long j = jiffies;

/* fill the data for our timer function */

data->prevjiffies = j;

data->buf = buf2;

data->loops = JIT_ASYNC_LOOPS;

/* register the timer */

data->timer.data = (unsigned long)data;

data->timer.function = jit_timer_fn;

data->timer.expires = j + tdelay; /* parameter */

add_timer(&data->timer);

/* wait for the buffer to fill */

wait_event_interruptible(data->wait, !data->loops);
```

实际的定时器函数如下所示：

```c
void jit_timer_fn(unsigned long arg)

{

    struct jit_data *data = (struct jit_data *)arg;

    unsigned long j = jiffies;

    data->buf += sprintf(data->buf, "%9li  %3li     %i
%6i   %i   %s\n",

                  j, j - data->prevjiffies, in_interrupt(
) ? 1 : 0,

                  current->pid, smp_processor_id( ),
current->comm);

    if (--data->loops) {

        data->timer.expires += tdelay;

        data->prevjiffies = j;

        add_timer(&data->timer);

    } else {

        wake_up_interruptible(&data->wait);

    }

}
```

- 在这段代码中，我们首先获取当前的 jiffies 值，并将其存储在 `data->prevjiffies` 中。然后，我们设置定时器的数据、函数和到期时间，并将定时器添加到定时器系统。

- 在定时器函数 `jit_timer_fn` 中，我们首先获取当前的 jiffies 值，并将其与上一次的 jiffies 值进行比较，然后将结果以及其他相关信息写入到缓冲区中。如果还有更多的循环需要执行，我们就更新定时器的到期时间并重新添加定时器。否则，我们就唤醒等待队列中的进程。

定时器 API 包含的函数比上面介绍的要多一些。以下集合完成了内核提供的列表：

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

更新定时器的到期时间，这是使用超时定时器的常见任务（再次，关闭软盘马达的定时器就是一个典型的例子）。mod_timer 也可以在非活动定时器上调用，你通常使用 add_timer。

```c
int del_timer_sync(struct timer_list *timer);
```

像 del_timer 一样工作，但也保证当它返回时，定时器函数不在任何 CPU 上运行。del_timer_sync 用于避免 SMP 系统上的竞态条件，并且在 UP 内核中与 del_timer 相同。在大多数情况下，应优先使用此函数而不是 del_timer。如果从非原子上下文调用此函数，它可以睡眠，但在其他情况下会忙等待。在持有锁的同时非常小心地调用 del_timer_sync；如果定时器函数试图获取相同的锁，系统可能会死锁。如果定时器函数重新注册自己，调用者必须首先确保这种重新注册不会发生；这通常是通过设置一个“正在关闭”的标志来完成的，定时器函数会检查这个标志。

```c
int timer_pending(const struct timer_list * timer);
```

返回 true 或 false 来指示定时器当前是否被调度运行，通过读取结构的一个不透明字段。

7.4.2. 内核定时器的实现

虽然你在使用内核定时器时不需要知道它们是如何实现的，但实现方式很有趣，看一下它的内部结构是值得的。定时器的实现设计是为了满足以下要求和假设：

- 定时器管理必须尽可能轻量级。
- 随着活动定时器数量的增加，设计应具有良好的扩展性。
- 大多数定时器在几秒或最多几分钟内到期，而长时间延迟的定时器相当罕见。
- 定时器应在注册它的同一 CPU 上运行。

内核开发人员设计的解决方案基于每个 CPU 的数据结构。timer_list 结构在其 base 字段中包含指向该数据结构的指针。如果 base 为 NULL，定时器不会被调度运行；否则，指针告诉哪个数据结构（因此，哪个 CPU）运行它。每个 CPU 的数据项在第 8 章的“每个 CPU 的变量”部分中有描述。

每当内核代码注册定时器（通过 add_timer 或 mod_timer），操作最终由 internal_add_timer（在 kernel/timer.c 中）执行，它又将新定时器添加到与当前 CPU 关联的“级联表”中的双向链接定时器列表。

级联表的工作方式如下：如果定时器在接下来的 0 到 255 jiffies 内到期，它会被添加到 256 个专门用于短期定时器的列表之一，使用 expires 字段的最低有效位。如果它在未来（但在 16,384 jiffies 之前）到期，它会被添加到基于 expires 字段的第 9-14 位的 64 个列表之一。对于更远到期的定时器，对第 15-20 位、21-26 位和 27-31 位使用相同的技巧。对于 expire 字段指向更远未来的定时器（这只能在 64 位平台上发生），它们会使用延迟值 0xffffffff 进行哈希，并且过去的 expires 的定时器被安排在下一个定时器 tick 运行。（在高负载情况下，有时可能会注册已经过期的定时器，特别是如果你运行一个可抢占的内核。）

当 `__run_timers` 被触发时，它执行当前定时器 tick 的所有待处理定时器。如果 jiffies 当前是 256 的倍数，该函数还会将下一级定时器列表中的一个重新哈希到 256 个短期列表中，可能还会根据 jiffies 的位表示级联一个或多个其他级别。

虽然这种方法乍看之下非常复杂，但无论是少量的定时器还是大量的定时器，它的性能都非常好。管理每个活动定时器所需的时间与已注册的定时器数量无关，仅限于对其 expires 字段的二进制表示进行几个逻辑操作。与这种实现相关的唯一成本是 512 个列表头的内存（256 个短期列表和 4 组 64 个更多的列表）——即，4 KB 的存储空间。

函数 `__run_timers`，如 /proc/jitimer 所示，是在原子上下文中运行的。除了我们已经描述的限制外，这还带来了一个有趣的特性：定时器在恰好正确的时间到期，即使你没有运行一个可抢占的内核，CPU 也在内核空间忙碌。当你在后台读取 /proc/jitbusy 并在前台读取 /proc/jitimer 时，你可以看到发生了什么。尽管系统似乎被忙等待的系统调用牢牢锁定，但内核定时器仍然工作正常。

然而，要记住，内核定时器远非完美，因为它受到硬件中断引起的抖动和其他工件的影响，以及其他定时器和其他异步任务。虽然与简单的数字 I/O 关联的定时器对于运行步进电机或其他业余电子设备等简单任务可能足够，但通常不适合在工业环境中的生产系统。对于这样的任务，你最可能需要使用实时内核扩展。

7.5. Tasklets 机制

与定时问题相关的另一个内核设施是 tasklet 机制。它主要用于中断管理（我们将在第 10 章再次看到它。）

Tasklet 在某些方面类似于内核定时器。它们总是在中断时间运行，总是在调度它们的同一 CPU 上运行，并且它们接收一个无符号长参数。然而，与内核定时器不同，你不能要求在特定时间执行函数。通过调度一个 tasklet，你只是要求它在内核选择的稍后时间执行。这种行为对于中断处理程序特别有用，其中硬件中断必须尽快管理，但大部分数据管理可以安全地延迟到稍后时间。实际上，一个 tasklet，就像一个内核定时器一样，在“软中断”的上下文中（以原子模式）执行，这是一个内核机制，它在启用硬件中断的情况下执行异步任务。

Tasklet 作为一个数据结构存在，必须在使用前初始化。可以通过调用特定函数或使用某些宏声明结构来进行初始化：

```c
#include <linux/interrupt.h>

struct tasklet_struct {
        /* ... */
        void (*func)(unsigned long);
        unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
        void (*func)(unsigned long), unsigned long data);

DECLARE_TASKLET(name, func, data);

DECLARE_TASKLET_DISABLED(name, func, data);
```

Tasklet 提供了一些有趣的特性：

- 一个 tasklet 可以被禁用并在稍后重新启用；它不会被执行，直到它被启用的次数与它被禁用的次数一样多。
- 就像定时器一样，一个 tasklet 可以重新注册自己。
- 一个 tasklet 可以被调度以正常优先级或高优先级执行。后者组总是先执行。
- 如果系统没有承受重负载，tasklet 可能会立即运行，但永远不会晚于下一个定时器 tick。
- 一个 tasklet 可以与其他 tasklet 并发，但严格地与自身序列化——同一个 tasklet 永远不会在多于一个处理器上同时运行。此外，如前所述，一个 tasklet 总是在调度它的同一 CPU 上运行。

jit 模块包括两个文件，/proc/jitasklet 和 /proc/jitasklethi，它们返回与在“内核定时器”部分介绍的 /proc/jitimer 相同的数据。当你读取其中一个文件时，你会得到一个头部和六行数据。第一行数据描述了调用进程的上下文，其他行描述了 tasklet 程序的连续运行的上下文。这是在编译内核时的一个样本运行：

```
phon% cat /proc/jitasklet

    time    delta  inirq    pid    cpu command

 6076139     0      0       4370    0   cat

 6076140     1      1       4368    0   cc1

 6076141     1      1       4368    0   cc1

 6076141     0      1          2    0   ksoftirqd/0

 6076141     0      1          2    0   ksoftirqd/0

 6076141     0      1          2    0   ksoftirqd/0
```

如上述数据所证实，只要 CPU 忙于运行一个进程，tasklet 就会在下一个定时器 tick 时运行，但是当 CPU 否则空闲时，它会立即运行。内核提供了一组 ksoftirqd 内核线程，每个 CPU 一个，只是为了运行“软中断”处理程序，如 tasklet_action 函数。因此，tasklet 的最后三次运行发生在与 CPU 0 关联的 ksoftirqd 内核线程的上下文中。jitasklethi 实现使用了一个高优先级的 tasklet，将在即将到来的函数列表中解释。

在 jit 中实现 /proc/jitasklet 和 /proc/jitasklethi 的实际代码与实现 /proc/jitimer 的代码几乎相同，但它使用的是 tasklet 调用而不是定时器调用。以下列表详细列出了在 tasklet 结构被初始化后的内核接口：

```c
void tasklet_disable(struct tasklet_struct *t);
```

此函数禁用给定的 tasklet。tasklet 仍然可以使用 tasklet_schedule 进行调度，但其执行被推迟到 tasklet 再次被启用。如果 tasklet 当前正在运行，此函数会忙等待，直到 tasklet 退出；因此，在调用 tasklet_disable 后，你可以确定 tasklet 在系统的任何地方都没有运行。

```c
void tasklet_disable_nosync(struct tasklet_struct *t);
```

禁用 tasklet，但不等待任何当前正在运行的函数退出。当它返回时，tasklet 被禁用并且在重新启用之前不会被调度，但是当函数返回时，它可能仍在另一个 CPU 上运行。

```c
void tasklet_enable(struct tasklet_struct *t);
```

启用之前被禁用的 tasklet。如果 tasklet 已经被调度，它将很快运行。对 tasklet_enable 的调用必须与每次对 tasklet_disable 的调用相匹配，因为内核会跟踪每个 tasklet 的“禁用计数”。

```c
void tasklet_schedule(struct tasklet_struct *t);
```

调度 tasklet 执行。如果一个 tasklet 在有机会运行之前再次被调度，它只运行一次。然而，如果它在运行时被调度，它在完成后再次运行；这确保了在处理其他事件时发生的事件得到应有的关注。这种行为也允许一个 tasklet 重新调度自己。

```c
void tasklet_hi_schedule(struct tasklet_struct *t);
```

以更高的优先级调度 tasklet 执行。当软中断处理程序运行时，它在处理其他软中断任务（包括“正常” tasklet）之前处理高优先级的 tasklet。理想情况下，只有具有低延迟要求的任务（如填充音频缓冲区）应该使用这个函数，以避免其他软中断处理程序引入的额外延迟。实际上，/proc/jitasklethi 与 /proc/jitasklet 没有人类可见的区别。

```c
void tasklet_kill(struct tasklet_struct *t);
```

此函数确保 tasklet 不再被调度运行；通常在设备被关闭或模块被移除时调用。如果 tasklet 被调度运行，函数会等待它执行。如果 tasklet 重新调度自己，你必须在调用 tasklet_kill 之前阻止它重新调度自己，就像 del_timer_sync 一样。

Tasklet 在 kernel/softirq.c 中实现。两个 tasklet 列表（正常和高优先级）被声明为每 CPU 数据结构，使用与内核定时器相同的 CPU 亲和性机制。在 tasklet 管理中使用的数据结构是一个简单的链表，因为 tasklet 没有内核定时器的排序要求。

7.6. 工作队列

工作队列（Workqueues）在表面上类似于 tasklets；它们允许内核代码请求在未来某个时间调用一个函数。然而，两者之间有一些显著的区别，包括：

- Tasklets 在软件中断上下文中运行，因此所有的 tasklet 代码必须是原子的。相反，工作队列函数在特殊内核进程的上下文中运行；因此，它们有更多的灵活性。特别是，工作队列函数可以睡眠。
- Tasklets 总是在它们最初提交的处理器上运行。工作队列默认也是这样工作的。
- 内核代码可以请求延迟工作队列函数的执行一个明确的间隔。

两者之间的关键区别是，tasklets 快速执行，执行时间短，且在原子模式下，而工作队列函数可能有更高的延迟，但不需要是原子的。每种机制都有适合的情况。

工作队列有一个类型为 struct workqueue_struct，定义在 <linux/workqueue.h> 中。在使用前，必须使用以下两个函数之一显式创建工作队列：

```c
struct workqueue_struct *create_workqueue(const char
*name);

struct workqueue_struct
*create_singlethread_workqueue(const char *name);
```

每个工作队列都有一个或多个专用进程（“内核线程”），它们运行提交到队列的函数。如果你使用 create_workqueue，你会得到一个工作队列，它对系统上的每个处理器都有一个专用线程。在许多情况下，所有这些线程都是过度的；如果一个工作线程就足够了，那么使用 create_singlethread_workqueue 创建工作队列。

要将任务提交给工作队列，你需要填充一个 work_struct 结构。这可以在编译时如下进行：

```c
DECLARE_WORK(name, void (*function)(void *), void *data);
```

其中，name 是要声明的结构的名称，function 是要从工作队列中调用的函数，data 是要传递给该函数的值。如果你需要在运行时设置 work_struct 结构，使用以下两个宏：

```c
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK 对初始化结构做了更彻底的工作；你应该在第一次设置该结构时使用它。PREPARE_WORK 做了几乎相同的工作，但它没有初始化用于将 work_struct 结构链接到工作队列的指针。如果有可能该结构当前已被提交到一个工作队列，并且你需要更改该结构，那么使用 PREPARE_WORK 而不是 INIT_WORK。

有两个函数用于将工作提交到工作队列：

```c
int queue_work(struct workqueue_struct *queue, struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);
```

两者都将工作添加到给定的队列。然而，如果使用 queue_delayed_work，实际的工作直到至少过去 delay 个 jiffies 后才执行。这些函数的返回值是非零的，如果工作成功添加到队列；零结果意味着这个 work_struct 结构已经在队列中等待，并没有第二次添加。

在未来的某个时间，将使用给定的数据值调用工作函数。该函数将在工作线程的上下文中运行，所以如果需要，它可以睡眠——尽管你应该意识到这种睡眠可能如何影响提交到同一工作队列的任何其他任务。然而，函数不能做的是访问用户空间。由于它在内核线程内部运行，简单地说就是没有用户空间可以访问。

如果你需要取消一个待处理的工作队列条目，你可以调用：

```c
int cancel_delayed_work(struct work_struct *work);
```

如果在开始执行之前取消了条目，返回值为非零。内核保证在调用 cancel_delayed_work 后不会启动给定条目的执行。然而，如果 cancel_delayed_work 返回 0，条目可能已经在另一个处理器上运行，并且可能在调用 cancel_delayed_work 后仍在运行。为了绝对确定在 cancel_delayed_work 返回 0 后，工作函数在系统的任何地方都没有运行，你必须在那个调用后跟一个调用：

```c
void flush_workqueue(struct workqueue_struct *queue);
```

在 flush_workqueue 返回后，提交到调用之前的任何工作函数都不在系统的任何地方运行。

当你完成一个工作队列，你可以用以下方法去掉它：

```c
void destroy_workqueue(struct workqueue_struct *queue);
```

7.6.1. 共享队列

在许多情况下，设备驱动程序不需要自己的工作队列。如果你只偶尔向队列提交任务，那么简单地使用内核提供的共享的默认工作队列可能会更有效率。然而，如果你使用这个队列，你必须意识到你将与其他人共享它。在其他方面，这意味着你不应该长时间垄断队列（没有长时间的睡眠），并且你的任务可能需要更长的时间才能在处理器中轮到。

jiq（“just in queue”）模块导出两个文件，演示了共享工作队列的使用。它们使用一个 work_struct 结构，这样设置：

```c
static struct work_struct jiq_work;

    /* this line is in jiq_init( ) */

    INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

当一个进程读取 /proc/jiqwq 时，模块启动一系列通过共享工作队列的旅程，没有延迟。它使用的函数是：

```c
int schedule_work(struct work_struct *work);
```

注意，当使用共享队列时，使用的是不同的函数；它只需要一个 work_struct 结构作为参数。jiq 中的实际代码看起来像这样：

```c
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);

schedule_work(&jiq_work);

schedule( );

finish_wait(&jiq_wait, &wait);
```

实际的工作函数打印出一行，就像 jit 模块一样，然后，如果需要，将 work_struct 结构重新提交到工作队列。这是 jiq_print_wq 的全部内容：

```c
static void jiq_print_wq(void *ptr)

{

    struct clientdata *data = (struct clientdata *) ptr;

    if (! jiq_print (ptr))

        return;

    if (data->delay)

        schedule_delayed_work(&jiq_work, data->delay);

    else

        schedule_work(&jiq_work);

}
```

如果用户正在读取延迟设备（/proc/jiqwqdelay），工作函数将自己以延迟模式重新提交，使用 schedule_delayed_work：

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

这两个设备的输出看起来像这样：

% cat /proc/jiqwq

```
     time  delta preempt   pid cpu command

   1113043     0       0     7   1 events/1

   1113043     0       0     7   1 events/1

   1113043     0       0     7   1 events/1

   1113043     0       0     7   1 events/1

   1113043     0       0     7   1 events/1

 % cat /proc/jiqwqdelay

     time  delta preempt   pid cpu command

   1122066     1       0     6   0 events/0

   1122067     1       0     6   0 events/0

   1122068     1       0     6   0 events/0

   1122069     1       0     6   0 events/0

   1122070     1       0     6   0 events/0
```

当读取 /proc/jiqwq 时，每行打印之间没有明显的延迟。相反，当读取 /proc/jiqwqdelay 时，每行之间恰好有一个 jiffy 的延迟。在任何情况下，我们都看到打印的相同进程名；这是实现共享工作队列的内核线程的名称。CPU 数字在斜杠后打印；我们永远不知道在读取 /proc 文件时哪个 CPU 会运行，但工作函数之后总是在同一个处理器上运行。

如果你需要取消提交到共享队列的工作条目，你可以使用 cancel_delayed_work，如上所述。然而，刷新共享工作队列需要一个单独的函数：

```c
void flush_scheduled_work(void);
```

由于你不知道谁可能还在使用这个队列，你永远不知道 flush_scheduled_work 可能需要多长时间才能返回。

7.7. 快速参考

本章介绍了下面的符号.

7.7.1. 时间管理

```c
#include <linux/param.h>
HZ
```

HZ 符号指定了每秒产生的时钟嘀哒的数目.

```c
#include <linux/jiffies.h>
volatile unsigned long jiffies;
u64 jiffies_64;
```

jiffies_64 变量每个时钟嘀哒时被递增; 因此, 它是每秒递增 HZ 次. 内核代码几乎常常引用 jiffies, 它在 64-位平台和 jiffies_64 相同并且在 32-位平台是它低有效的一半.

```c
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

这些布尔表达式以一种安全的方式比较 jiffies, 没有万一计数器溢出的问题和不需要存取 jiffies_64.

```c
u64 get_jiffies_64(void);
```

获取 jiffies_64 而没有竞争条件.

```c
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec
*value);
void jiffies_to_timespec(unsigned long jiffies, struct
timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct
timeval *value);
```

在 jiffies 和其他表示之间转换时间表示.

```c
#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var32);
```

x86-特定的宏定义来读取时戳计数器. 它们作为 2 半 32-位来读取, 只读低一半, 或者全部读到一个 long long 变量.

```c
#include <linux/timex.h>
cycles_t get_cycles(void);
```

以平台独立的方式返回时戳计数器. 如果 CPU 没提供时戳特性, 返回 0.

```c
#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
```

返回自 Epoch 以来的秒数, 基于 6 个 unsigned int 参数.

```c
void do_gettimeofday(struct timeval *tv);
```

返回当前时间, 作为自 Epoch 以来的秒数和微秒数, 用硬件能提供的最好的精度. 在大部分的平台这个解决方法是一个微秒或者更好, 尽管一些平台只提供 jiffies 精度.

```c
struct timespec current_kernel_time(void);
```

返回当前时间, 以一个 jiffy 的精度.

7.7.2. 延迟

```c
#include <linux/wait.h>
long wait_event_interruptible_timeout(wait_queue_head_t
*q, condition, signed long timeout);
```

使当前进程在等待队列进入睡眠, 安装一个以 jiffies 表达的超时值. 使用 schedule_timeout(下面) 给不可中断睡眠.

```c
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

调用调度器, 在确保当前进程在超时到的时候被唤醒后. 调用者首先必须调用 set_curret_state 来使自己进入一个可中断的或者不可中断的睡眠状态.

```c
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

引入一个整数纳秒, 微秒和毫秒的延迟. 获得的延迟至少是请求的值, 但是可能更多. 每个函数的参数必须不超过一个平台特定的限制(常常是几千).

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

使进程进入睡眠给定的毫秒数(或者秒, 如果使 ssleep).

7.7.3. 内核定时器

```c
#include <asm/hardirq.h>
int in_interrupt(void);
int in_atomic(void);
```

返回一个布尔值告知是否调用代码在中断上下文或者原子上下文执行. 中断上下文是在一个进程上下文之外, 或者在硬件或者软件中断处理中. 原子上下文是当你不能调度一个

中断上下文或者一个持有一个自旋锁的进程的上下文.

```c
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
struct timer_list TIMER_INITIALIZER(_function, _expires,
_data);
```

这个函数和静态的定时器结构的声明是初始化一个 timer_list 数据结构的 2 个方法.

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

注册定时器结构来在当前 CPU 上运行.

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

改变一个已经被调度的定时器结构的超时时间. 它也能作为一个 add_timer 的替代.

```c
int timer_pending(struct timer_list * timer);
```

宏定义, 返回一个布尔值说明是否这个定时器结构已经被注册运行.

```c
void del_timer(struct timer_list * timer);
void del_timer_sync(struct timer_list * timer);
```

从激活的定时器链表中去除一个定时器. 后者保证这定时器当前没有在另一个 CPU 上运行.

7.7.4. Tasklets 机制

```c
#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
void tasklet_init(struct tasklet_struct *t, void (*func)
(unsigned long), unsigned long data);
```

前 2 个宏定义声明一个 tasklet 结构, 而 tasklet_init 函数初始化一个已经通过分配或其他方式获得的 tasklet 结构. 第 2 个 DECLARE 宏标识这个 tasklet 为禁止的.

```c
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

禁止和使能一个 tasklet. 每个禁止必须配对一个使能(你可以禁止这个 tasklet 即便它已经被禁止). 函数 tasklet_disable 等待 tasklet 终止如果它在另一个 CPU 上运行. 这个非同步版本不采用这个额外的步骤.

```c
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

调度一个 tasklet 运行, 或者作为一个"正常" tasklet 或者一个高优先级的. 当软中断被执行, 高优先级 tasklets 被首先处理, 而正常 tasklet 最后执行.

```c
void tasklet_kill(struct tasklet_struct *t);
```

从激活的链表中去掉 tasklet, 如果它被调度执行. 如同 tasklet_disable, 这个函数可能在 SMP 系统中阻塞等待 tasklet 终止, 如果它当前在另一个 CPU 上运行.

7.7.5. 工作队列

```c
#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;
```

这些结构分别表示一个工作队列和一个工作入口.

```c
struct workqueue_struct *create_workqueue(const char
*name);
struct workqueue_struct
*create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);
```

创建和销毁工作队列的函数. 一个对 create_workqueue 的调用创建一个有一个工作者线程在系统中每个处理器上的队列; 相反, create_singlethread_workqueue 创建一个有一个单个工作者进程的工作队列.

```c
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void
*), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)
(void *), void *data);
```

声明和初始化工作队列入口的宏.

```c
int queue_work(struct workqueue_struct *queue, struct
work_struct *work);int queue_delayed_work(struct
workqueue_struct *queue, struct work_struct *work,
unsigned long delay);
```

从一个工作队列对工作进行排队执行的函数.

```c
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
```

使用 cancel_delayed_work 来从一个工作队列中去除入口; flush_workqueue 确保没有工作队列入口在系统中任何地方运行.

```c
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work,
unsigned long delay);
void flush_scheduled_work(void);
```

使用共享队列的函数.