第 10 章 对称多处理（SMP）

在全书的讨论过程中，我一直在忽略 SMP 代码，而倾向于把注意力集中在只涉及一个处理器的相对简单的情况。现在已经到了重新访问读者已经熟悉的一些内容的时候了，不过要从一个新的角度来审视它：当内核必须支持多于一个 CPU 的机器时将发生什么？

在一般情况下，使用多于一个 CPU 来完成工作被称为并行处理（parallel processing），它可以被想象成是一段频谱范围，分布式计算（distributed computing）在其中一端，而对称多处理（SMP—symmetric multiprocessing）在另一端。通常，当你沿着该频谱从分布式计算向 SMP 移动时，系统将变得更加紧密耦合——在 CPU 之间共享更多的资源——而且更加均匀。在一个典型的分布式系统中，每个 CPU 通常都至少拥有它自己的高速缓存和 RAM。每个 CPU 还往往拥有自己的磁盘、图形子系统、声卡，监视器等等。

在极端的情形下，分布式系统经常不外乎就是一组普通的计算机，虽然它们可能具有完全不同的体系结构，但是都共同工作在某个网络之上——它们甚至不需要在同一个 LAN 里。读者可能知道的一些有趣的分布式系统包括：Beowulf，它是对相当传统而又极其强大的分布式系统的一个通用术语称谓；SETI@home，它通过利用上百万台计算机来协助搜寻地外生命的证据，以及 distributed.net，它是类似想法的另一个实现，它主要关注于地球上产生的密码的破解。

SMP 是并行处理的一个特殊情况，系统里所有 CPU 都是相同的。举例来说，SMP 就是你共同支配两块 80486 或两块 Pentium（具有相同的时钟速率）处理器，而不是一块 80486 和一块 Pentium，或者一块 Pentium 和一块 PowerPC。在通常的用法中，SMP 也意味着所有 CPU 都是“在相同处境下的”——那就是它们都在同一个计算机里，通过特殊用途的硬件进行彼此通信。

SMP 系统通常是另一种平常的单一（single）计算机——只不过具有两个或更多的 CPU。因此，SMP 系统除了 CPU 以外每样东西只有一个——一块图形卡、一个声音卡，等等之类。诸如 RAM 和磁盘这样以及类似的资源都是为系统的 CPU 们所共享的。（尽管现在 SMP 系统中每个 CPU 都拥有自己的高存缓存的情况已经变得愈发普遍了。）

分布式配置需要很少的或者甚至不需要来自内核的特殊支持；节点之间的协同是依靠用户空间的应用程序或者诸如网络子系统之类未经修改的内核组件来处理的。但是 SMP 在计算机系统内创建了一个不同的硬件配置，并由此需要特殊用途的内核支持。比如，内核必须确保 CPU 在访问它们的共享资源时要相互合作——这是一个读者在 UP 世界中所不曾遇到的问题。

SMP 的逐渐普及主要是因为通过 SMP 所获得的性能的提高要比购买几台独立的机器再把它们组合在一起更加便宜和简单，而且还因为它与等待下一代 CPU 面世相比要快的多。

非对称多 CPU 的配置没有受到广泛支持，这是因为对称配置情况所需的硬件和软件支持通常较为简单。不过，内核代码中平台无关的部分实际上并不特别关心是否 CPU 是相同的——即，是否配置是真正对称的——尽管它也没有进行任何特殊处理以支持非对称配置。例如，在非对称多处理系统中，调度程序应该更愿意在较快的而不是较慢的 CPU 上运行进程，但是 Linux 内核没有对此进行区别。

谚语说得好，“天下没有白吃的午餐”。对于 SMP，为提高的性能所付出的代价就是内核复杂度的增加和协同开销的增加。CPU 必须安排不互相干涉彼此的工作，但是它们又不能在这种协同上花费太多时间以至于它们显著地耗费额外的 CPU 能力。

代码的 SMP 特定部分由于 UP 机器存在的缘故而被单独编译，所以仅仅因为有了 SMP

寄存器是不会使 UP 寄存器慢下来的。这满足两条久经考验的原理：“ 为普遍情况进行优化 ”（UP 机器远比 SMP 机器普遍的多）以及 “ 不为用不着的东西花钱 ”。

并行程序设计概念及其原语

具有两个 CPU 的 SMP 配置可能是最简单的并行配置，但就算是这最简单的配置也揭开了未知问题的新领域——即使要两块相同的 CPU 在一起协调的工作，时常也都像赶着猫去放牧一样困难。幸运的是，至少 30 年前以来，就在这个项目上作了大量和非常熟悉的研究工作。（考虑到第一台电子数字计算机也只是在 50 年前建造的，那这就是一段令人惊讶的相当长的时间了。）在分析对 SMP 的支持是如何影响内核代码之前，对该支持所基于的若干理论性概念进行一番浏览将能够极大的简化这个问题。

注意：并非所有这些信息都是针对 SMP 内核的。一些要讨论的问题甚至是由 UP 内核上的并行程序设计所引起的，既要支持中断也要处理进程之间的交互。因此即使你对 SMP 问题没有特别的兴趣，这部分的讨论也值得一看。

原子操作

在一个并行的环境里，某些动作必须以一种基本的原子方式（atomically）执行——即不可中断。这种操作必须是不可分割的，就象是原子曾经被认为的那样。

作为一个例子，考虑一下引用计数。如果你想要释放你所控制的一份共享资源并要了解是否还有其它（进程）仍在使用它，你就会减少对该共享资源的计数值并把该值与 0 进行对照测试。一个典型的动作顺序可能如下开始：

1. CPU 把当前计数值（假设是 2）装载进它的一个寄存器里。
2. CPU 在它的寄存器里把这个值递减；现在它是 1。
3. CPU 把新值（1）写回内存里。
4. CPU 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。

对于 UP，应不必在此考虑过多（除了某些情况）。但是对于 SMP 就是另一番景象了：如果另一个 CPU 碰巧同时也在作同样的事情应如何处理呢？最坏的情形可能是这样的：

1. CPU A 把当前计数值（2）装载进它的一个寄存器里。
2. CPU B 把当前计数值（2）装载进它的一个寄存器里。
3. CPU A 在它的寄存器里把这个值递减；现在它是 1。
4. CPU B 在它的寄存器里把这个值递减；现在它是 1。
5. CPU A 把新值（1）写回内存里。
6. CPU B 把新值（1）写回内存里。
7. CPU A 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。
8. CPU B 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。

内存里的引用计数值现在应该是 0，然而它却是 1。两个进程都去掉了它们对该共享对象的引用，但是没有一个能够释放它。

这是一个有趣的失败，因为每个 CPU 都作了它所应该做的事情，尽管这样错误的结果还是发生了。当然这个问题就在于 CPU 没有协调它们的动作行为——右手不知道左手正在

干什么。

你会怎样试图在软件中解决这个问题呢？从任何一个 CPU 的观点来看待它——比如说是 CPU A。需要通知 CPU B 它不应使用引用计数值，由于你想要递减该值，所以不管怎样你最好改变某些 CPU B 所能见到的信息——也就是更新共享内存位置。举例来说，你可以为此目的而开辟出某个内存位置，并且对此达成一致：若任何一个 CPU 正试图减少引用计数它就包含一个 1，如果不是它就为 0。使用方法如下：

1. CPU A 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
2. CPU A 检查它的寄存器里的值并发现它是 0（如果不是，它再次尝试，重复直到该寄存器为 0 为止。）
3. CPU A 把一个 1 写回特殊内存位置。
4. CPU A 访问受保护的引用计数值。
5. CPU A 把一个 0 写回特殊内存位置。

糟糕，令人不安的熟悉情况又出现了。以下所发生的问题仍然无法避免：

1. CPU A 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
2. CPU B 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
3. CPU A 检查它的寄存器里的值并发现它是 0。
4. CPU B 检查它的寄存器里的值并发现它是 0。
5. CPU A 把一个 1 写回特殊内存位置。
6. CPU B 把一个 1 写回特殊内存位置。
7. CPU A 访问受保护的引用计数值。
8. CPU B 访问受保护的引用计数值。
9. CPU A 把一个 0 写回特殊内存位置。
10. CPU B 把一个 0 写回特殊内存位置。

好吧，或许可以再使用一个特殊内存位置来保护被期望保护初始内存位置的那个特殊内存位置……。

面对这一点吧：我们在劫难逃。这种方案只会使问题向后再退一层，而不可能解决它。最后，原子性不可能由软件单独保证——必须要有硬件的特殊帮助。

在 x86 平台上，**lock** 指令正好能够提供这种帮助。（准确地说，**lock** 是一个前缀而非一个单独的指令，不过这种区别和我们的目的没有利害关系。）**lock** 指令用于在随后的指令执行期间锁住内存总线——至少是对目的内存地址。因为 x86 可以在内存里直接减值，而无需明确的先把它读入一个寄存器中，这样对于执行一个减值原子操作来说就是万事俱备了：**lock** 内存总线然后立刻对该内存位置执行 **decl** 操作。

函数 **atomic_dec**（10241 行）正好为 x86 平台完成这样的工作。**LOCK** 宏的 **SMP** 版本在第 10192 行定义并扩展成 **lock** 指令。（在随后的两行定义的 UP 版本完全就是空的——单 CPU 不需要保护自己以防其它 CPU 的干扰，所以锁住内存总线将完全是在浪费时间。）通过把 **LOCK** 宏放在内嵌编译指令的前边，随后的指令就会为 SMP 内核而被锁定。如果 CPU B 在 CPU A 发挥作用时执行了 **atomic_dec** 函数，那么 CPU B 就会自动的等待 CPU A 把锁移开。这样就能够成功了！

这样还只能说是差不多。最初的问题仍然没有被很好的解决。目标不仅是要自动递减引用计数值，而且还要知道结果值是否是 0。现在可以完成原子递减了，可是如果另一个处理器在递减和结果测试之间又“偷偷的”进行了干预，那又怎么办呢？

幸运的是，解决这个部分问题不需要来自 CPU 的特殊目的的帮助。不管加锁还是未锁，x86 的 **decl** 指令总是会在结果为 0 时设置 CPU 的 Zero 标志位，而且这个标志位是 CPU 私有的，所以其它 CPU 的所为是不可能在递减步骤和测试步骤之间影响到这个标志位的。相

应的，**atomic_dec_and_test**（10249 行）如前完成一次加锁的递减，接着依据 CPU 的 Zero 标志位来设置本地变量 c。如果递减之后结果是 0 函数就返回非零值（真）。

如同其它定义在一个文件里的函数一样，**atomic_dec** 和 **atomic_dec_and_test** 都对一个类型为 **atomic_t** 的（10205 行）对象进行操作。就像 **LOCK**，**atomic_t** 对于 UP 和 SMP 也有不同的定义方式——不同之处在于 SMP 情况里引入了 **volatile** 限定词，它指示 gcc 不要对被标记的变量做某种假定（比如，不要假定它可以被安全的保存在一个寄存器里）。

顺便提及一下，读者在这段代码里看到的垃圾代码**__atomic_fool_gcc** 据报告已不再需要了；它曾用于纠正在 gcc 的早期版本下代码生成里的一个故障。

Test-And-Set

经典的并行原语是 test-and-set。test-and-set 操作自动地从一个内存位置读取一个值然后写入一个新值，并把旧值返回。典型的，该位置可以保存 0 或者 1，而且 test-and-set 所写的新值是 1——因此是“设置（set）”。与 test-and-set 对等的是 test-and-clear，它是同样的操作除了写入的是 0 而不是 1。一些 test-and-set 的变体既能写入 1 也可以写入 0，因此 test-and-set 和 test-and-clear 就能够成为一体，只是操作数不同而已。

test-and-set 原语足以实现任何其它并行安全的操作。（实际上，在某些 CPU 上 test-and-set 是唯一被提供的此类原语。）比如，原本 test-and-set 是能够用于前边的例子之中来保护引用计数值的。相似的方法以被尝试——从一个内存位置读取一个值，检查它是否为 0，如果是则写入一个 1，然后继续访问受保护的值。这种尝试的失败并不是因为它在逻辑上是不健全的，而是因为没有可行的方法使其自动完成。假使有了一个原子的 test-and-set，你就可以不通过使用 **lock** 来原子化 **decl** 的方法而顺利通过了。

然而，test-and-set 也有缺点：

- 它是一个低级的原语——在所有与它打交道时，其它原语都必须在它之上被执行。
- 它并不经济——当机器测试该值并发现它已经是 1 了怎么办呢？这个值在内存里不会被搞乱，因为只要用同样的值复写它即可。可事实是它已被设置就意味着其它进程正在访问受保护的对象，所以还不能这样执行。额外需要的逻辑——测试并循环——会浪费 CPU 时钟周期并使得程序变得更大一些（它还会浪费高速缓存里的空间）。

x86 的 **lock** 指令使高级指令更容易执行，但是你也可以在上执行原子 test-and-set 操作。最直接的方式是把 **lock** 和 **btsl** 指令（位 test-and-set）联合起来使用。这种方法要被本章后边介绍的自旋锁（spinlock）所用到。

另一种在 x86 上实现的方法是用它的 **xchg**（exchange）指令，它能够被 x86 自动处理，就好像它的前面有一个 **lock** 指令一样——只要它的一个操作数是在内存里。**xchg** 要比 **lock**/**btsl** 组合更为普遍，因为它可以一次交换 8、16，或者 32 位而不仅仅是 1 位。除了一个在 arch/i386/kernel/entry.S 里的使用之外，内核对 **xchg** 指令的使用都隐藏在 **xchg** 宏（13052 行）之后，而它又是在函数**__xchg**（13061 行）之上实现的。这样是便于在平台相关的代码里内核代码也可以使用 **xchg** 宏；每种平台都提供它自己对于该宏的等价的实现。

有趣的时，**xchg** 宏是另一个宏，**tas**（test-and-set——13054 行）的基础。然而，内核代码的任何一个地方都没有用到这个宏。

内核有时候使用 **xchg** 宏来完成简单的 test-and-set 操作（尽管不必在锁变得可用之前一直循环，如同第 22770 行），并把它用于其它目的（如同第 27427 行）。

信号量

第 9 章中讨论了信号量的基本概念并演示了它们在进程间通信中的用法。内核为达自己的目的有其特有的信号量实现，它们被特别的称为是“ 内核信号量 ”。（在这一章里，未经修饰的名词“ 信号量 ”应被理解为是“ 内核信号量 ”。）第 9 章里所讨论的基本信号量的概念同样适用于内核信号量：允许一个可访问某资源用户的最大数目（最初悬挂在吊钩上钥匙的特定数目），然后规定每个申请资源者都必须先获得一把钥匙才能使用该资源。

到目前为止，你大概应该已经发现信号量如何能够被建立在 test-and-set 之上并成为二元（“ 唯一钥匙 ”）信号量，或者在像 **atomic_dec_and_test** 这样的函数之上成为计数信号量的过程。内核正好就完成着这样的工作：它用整数代表信号量并使用函数 **down**（11644 行）和 **up**（11714 行）以及其它一些函数来递减和递增该整数。读者将看到，用于减少和增加整数的底层代码和 **atomic_dec_and_test** 及其它类似函数所使用的代码是一样的。

作为相关历史事件的提示，第一位规范信号量概念的研究者，Edsger Dijistra 是荷兰人，所以信号量的基础操作就用荷兰语命名为：Proberen 和 Verhogen，常缩写成 P 和 V。这对术语被翻译成“ 测试（test）”（检查是否还有一把钥匙可用，若是就取走）和“ 递增（increment）”（把一个钥匙放回到吊钩之上）。那些词首字母正是在前一章中所引入的术语“ 获得（procure）”和“ 交出（vacate）”的来源。Linux 内核打破了这个传统，用操作 **down** 和 **up** 的称呼取代了它们。

内核用一个非常简单的类型来代表信号量：定义在 11609 行的 **struct semaphore**。他只有三个成员：

- **count**——跟踪仍然可用的钥匙数目。如果是 0，钥匙就被取完了；如果是负数，钥匙被取完而且还有其它申请者在等待它。另外，如果 **count** 是 0 或负数，那么其它申请者的数目就等于 **count** 的绝对值。

  Sema_init 宏（11637 行）允许 **count** 被初始化为任何值，所以内核信号量可以是二元的（初始化 **count** 为 1）也可以是计数型的（赋予它某个更大的初始值）。所有内核信号量代码都完全支持二元和计数型信号量，前者可作为后者的一个特例。不过在实践中 **count** 总是被初始化为 1，这样内核信号量也总是二元类型的。尽管如此，没有什么能够阻止一个开发者将来增加一个新的计数信号量。

  要顺便提及的是，把 **count** 初始化为正值而且用递减它来表明你需要一个信号量的方法并没有什么神秘之处。你也可以用一个负值（或者是 0）来初始化计数值然后增加它，或者遵循其它的方案。使用正的数字只是内核所采用的办法，而这碰巧和我们头脑中的吊钩上的钥匙模型吻合得相当好。的确，正如你将看到的那样，内核锁采用的是另一种方式工作——它被初始化为负值，并在进程需要它时进行增加。
- **waking**——在 **up** 操作期间及之后被暂时使用；如果 **up** 正在释放信号量则它被设置为 1，否则是 0。
- **wait**——因为要等待这个信号量再次变为可用而不得不被挂起的进程队列。

down

11644：**down** 操作递减信号量计数值。你可能会认为它与概念里的实现一样简单，不过实际上远不是这样简单。

11648：减少信号量计数值——要确保对 SMP 这是自动完成的。对于 SMP 来说（当然也适于 UP），除了被访问的整数是在一个不同类型的 **struct** 之内以外，这同在 **atomic_dec_and_test** 中所完成的工作本质上是相同的。

读者可能会怀疑 **count** 是否会下溢。它不会：进程总是在递减 **count** 之后进入休眠，所以一个给定的进程一次只能获得一个信号量，而且 **int** 具有的负值要比进程的数目多的多。

11652：如果符号位被设置，信号量就是负值。这意味着甚至它在被递减之前就是 0 或者负值了，这样进程无法得到该信号量并因此而应该休眠一直到它变成可用。接下来的几行代码十分巧妙地完成了这一点。如果符号位被设置则执行 **js** 跳转（即若 **decl** 的结果是负的它就跳转），**2f** 标识出跳转的目的地。**2f** 并非十六进制值——它是特殊的 GNU 汇编程序语法：**2** 表示跳转到本地符号“ 2 ”，**f** 表示向前搜索这个符号。（**2b** 将表示向后搜索最近的本地符号“ 2 ”。）这个本地符号在第 11655 行。

11653：分支转移没有执行，所以进程得到了信号量。虽然看起来不是这样，但是这实际已经到达 **down** 的末尾。稍后将对此进行解释。

11654：**down** 的技巧在于指令**.section** 紧跟在跳转目标的前面，它表示把随后的代码汇编到内核的一个单独的段中——该段被称为**.text.lock**。这个段将在内存中被分配并标识为可执行的。这一点是由跟在段名之后的 **ax** 标志字符串来指定的——注意这个 **ax** 与 x86 的 AX 寄存器无关。

这样的结果是，汇编程序将把 11655 和 11656 行的指令从 **down** 所在的段里转移到可执行内核的一个不同的段里。所以这些行生成的目标代码与其前边的行所生成的代码从物理上不是连续的。这就是为什么说 11653 行是 **down** 的结尾的原因。

11655：当信号量无法得到时跳转到的这一目的行。**Pushl $1b** 并不是要把十六进制值 1b 压入栈中——如果要执行那种工作应该使用 **pushl $0x1b**（也可以写成是不带**$**的）。正确的解释是，这个 **1b** 和前边见到的 **2f** 一样都是 GNU 汇编程序语法——它指向一个指令的地址；在此情形中，它是向后搜索时碰到的第一个本地标识“ 1 ”的地址。所以，这条指令是把 11653 行代码的地址压入栈中；这个地址将成为返回地址，以便在随后的跳转操作之后，执行过程还能返回到 **down** 的末尾。

11656：开始跳转到**__down_failed**（不包括在本书之内）。这个函数在栈里保存几个寄存器并调用后边要介绍的**__down**（26932 行）来完成等待信号量的工作。一旦**__down** 返回了，**__down_failed** 就返回到 **down**，而它也随之返回。一直到进程获得了信号量**__down** 才会返回；最终结果就是只要 **down** 返回，进程就得到信号量了，而不管它是立刻还是经过等待后获得的它。

11657：伪汇编程序指令**.previous** 的作用未在正式文档中说明，但是它的意思肯定是还原到以前的段中，结束 11654 行里的伪指令**.section** 的作用效果。

down_interruptible

11664：**down_interruptible** 函数被用于进程想要获得信号量但也愿意在等待它时被信号中断的情况。这个函数与 **down** 的实现非常相似，不过有两个区别将在随后的两段里进行解释。

11666：第一个区别是 **down_interruptible** 函数返回一个 **int** 值来指示是否它获得了信号量或者被一个信号所打断。在前一种情况里返回值（在 **result** 里）是 0，在后一种情况里它是负值。这部分上是由 11675 行代码完成的，如果函数未经等待获得了信号量则该行把 **result** 设置为 0。

11679：第二个区别是 **down_interruptible** 函数跳转到**__down_failed_interruptible**（不包括在本书之内）而不是**__down_failed**。因循**__down_failed** 建立起来的模式，**__down_failed_interruptible** 只是调整几个寄存器并调用将在随后进行研究的**__down_interruptible** 函数（26942 行）。要注意的是 11676 行为**__down_failed_**

interruptible 设置的返回目标跟在 **xorl** 之后，**xorl** 用于在信号量可以被立刻获得的情况中把 **result** 归 0。**down_interruptible** 函数的返回值再被复制进 **result** 中。

down_trylock

11687：除了调用**__down_failed_trylock** 函数（当然还要调用 26961 行的**__down_trylock** 函数，我们将在后面对它进行检查）之外，**down_trylock** 函数和 **down_interruptible** 函数相同。因此，在这里不必对 **down_trylock** 函数进行更多解释。

DOWN_VAR

26900：这是作为**__down** 和**_down_interruptible** 共同代码因子的三个宏中的第一个。它只是声明了几个变量。

DOWN_HEAD

26904：这个宏使任务 **tsk**（被 **DOWN_VAR** 所声明）转移到 **task_state** 给出的状态，然后把 **tsk** 添加到等待信号量的任务队列。最后，它开始一个无限循环，在此循环期间当**__down** 和**__down_interruptible** 准备退出时将使用 **break** 语句结束该循环。

DOWN_TAIL

26926：这个宏完成循环收尾工作，把 **tsk** 设置回 **task_state** 的状态，为再次尝试获得信号量做准备。

26929：循环已经退出；**tsk** 已或者得到了信号量或者被一个信号中断了（仅适于**__down_interruptible**）。无论哪一种方式，任务已准备再次运行而不再等待该信号量了，因此它被转移回 **TASK_RUNNING** 并从信号量的等待队列里被注销。

__down

26932：**__down** 和**__down_interruptible** 遵循以下模式：

1. 用 **DOWN_VAR** 声明所需的本地变量，随后可能还有补充的本地变量声明。
2. 以 **DOWN_HEAD** 开始进入无穷循环。
3. 在循环体内完成函数特定的（function-specific）工作。
4. 重新调度。
5. 以 **DOWN_TAIL** 结束。注意对 **schedule** 的调用（26686 行，在第 7 章里讨论过）可以被移进 **DOWN_TAIL** 宏中。
6. 完成任何函数特定的收尾工作。

我将只对函数特定的步骤（第 3 和第 6 步）进行讨论。

26936：**__down** 的循环体调用 **waking_non_zero**（未包括），它自动检查 **sem->waking** 来判断是否进程正被 **up** 唤醒。如果是这样，它将 **waking** 归零并返回 1（这仍然是同一个原子操作的一部分）；如果不是，它返回 0。因此，它返回的值指示了是否进程获得了信号量。如果它获得了值，循环就退出，接着函数也将返回。否则，进程将继续等待。

顺便要说明的是，观察一下**__down** 尝试获得信号量是在调用 **schedule** 之前。如果信号量的计数值已知为负值时，为什么不用另一种相反的方式来实现它呢？实际上它对于第一遍循环之后的任何一遍重复都是没有影响的，但是去掉一次没有必要的检查可以稍微加快第一遍循环的速度。如果需要为此提出什么特别的理由的话，那

可能就是因为自从信号量第一次被检查之后的几个微秒内它就应该可以被释放（可能是在另一个处理器上），而且额外获取标志要比一次额外调度所付出的代价少得多。因此**__down** 可能还可以在重新调度之前做一次快速检查。

__down_interruptible

26942：**__down_interruptible** 除了允许被信号中断以外，它和**__down** 在本质上是一样的。

26948：所以，当获取信号量时对 **waking_non_zero_interruptible**（未包括）进行调用。如果它没能得到信号量就返回 0，如果得到就返回 1，或者如果它被一个信号所中断就返回**–EINTR**。在第一种情况下，循环继续。

26958：否则，**__down_interruptible** 退出，如果它得到信号量就返回 0（不是 1），或者假如被中断则返回**–EINTR**。

__down_trylock

26961：有时在不能立刻获得信号量的情况下，内核也需要继续运行。所以，**__down_trylock** 不在循环之内。它仅仅调用 **waking_nonzero_trylock**（未包括），该函数夺取信号量，如果失败就递增该信号量的 **count**（因为内核不打算继续等待下去）然后返回。

up

11714：我们已经详尽的分析了内核尝试获得信号量时的情况，也讨论了它失败时的情况。现在是考察另一面的时候了：当释放一个信号量时将发生什么。这一部分相对简单。

11721：原子性地递增信号量的计数值。

11722：如果结果小于等于 0，就有某个进程正在等待被唤醒。up 向前跳转到 11725 行。

11724：**up** 采用了 **down** 里同样的技巧：这一行进入了内核的单独的一段，而不是在 **up** 本身的段内。**up** 的末尾的地址被压入栈然后 **up** 跳转到**__up_wakeup**（未包括）。这里完成如同**__down_failed** 一样的寄存器操作并调用下边要讨论的**__up** 函数。

__up

26877：**__up** 函数负责唤醒所有等待该信号量的进程。

26897：调用 **wake_one_more**（未包括在本书中），该函数检查是否有进程在等待该信号量，如果有，就增加 **waking** 成员来通知它们可以尝试获取它了。

26880：利用 **wake_up** 宏（16612 行），它只是调用**__wake_up** 函数（26829 行）来唤醒所有等待进程。

__wake_up

26829：正如在第 2 章中所讨论的那样，**__wake_up** 函数唤醒所有传递给它的在等待队列上的进程，假如它们处于被 **mode** 所隐含的状态之一的话。当从 **wake_up** 被调用时，函数唤醒所有处于 **TASK_UNINTERRUPTIBLE** 或 **TASK_INTERRUPTIBLE** 状态的进程；当从 **wake_up_interruptible**（16614 行）被调用时，它只唤醒处于 **TASK_INTERRUPTIBLE** 状态的任务。

26842：进程用 **wake_up_process**（26356 行）被唤醒，该函数曾在以前提到过，它将在本章随后进行详细介绍。

现在所感兴趣的是唤醒所有进程后的结果。因为**__wake_up** 唤醒所有队列里的进程，而不仅仅是队列里的第一个，所以它们都要竞争信号量——在 SMP 里，它们可以精确的同

时做这件事。通常，获胜者将首先获得 CPU。这个进程将是拥有最大“ goodness ”的进程（回忆一下第 7 章中 26338 行对 **goodness** 的讨论）。 这一点意义非常重大，因为拥有更高优先权的进程应该首先被给予继续其工作的机会。（这对于实时进程尤其重要。）

这种方案的不足之处是有发生饥饿（starvation）的危险，这发生在一个进程永远不能得到它赖以继续运行的资源时。这里可能会发生饥饿现象：假如两个进程反复竞争同一个信号量，而第一个进程总是有比第二个更高的优先权，那么第二个进程将永远不会得到 CPU。这种场景同它应该的运行方式存在一定差距——设想一个是实时进程而另一个以 20 的 niceness 运行。我们可以通过只唤醒队列里第一个进程的方法来避免这种饥饿的危险，可是那样又将意味着有时候会耽误从各个方面来说都更有资格的进程对 CPU 的使用。

以前对此没有讨论过，可是 Linux 的调度程序在适当的环境下也能够使得 CPU 的一个进程被彻底饿死。这不完全是一件坏事——只是一种设计决策而已——而且至少应用于通篇内核代码的原则是一致的，这就很好。还要注意的是使用前边讨论过的其它机制，饥饿现象也同样会发生。例如说，test-and-set 原语就是和内核信号量一样的潜在饥饿根源。

无论如何，在实际中，饥饿是非常少见的——它只是一个有趣的理论案例。

Spinlocks

这一章里最后一个重要的并行程序设计原语是自旋锁（spinlock）。自旋锁的思想就是在一个密封的循环里坚持反复尝试夺取一个资源（一把锁）直到成功为止。这通常是通过在类似 test-and-set 操作之上进行循环来实现的——即，旋转（spinning）——一直到获得该锁。

如果这听起来好像是一个二元信号量，那是因为它就是一个二元信号量。自旋锁和二元信号量唯一的概念区别就是你不必循环等待一个信号量——你可以夺取信号量，也可以在不能立刻得到它时放弃申请。因此，自旋锁原本是可以通过在信号量代码外再包裹一层循环来实现的。不过，因为自旋锁是信号量的一个受限特例，它们有更高效的实现方法。

自旋锁变量——其中的一位被测试和设置——总是 **spinlock_t** 类型（12785 行）。只有 **spinlock_t** 的最低位被使用；如果锁可用，则它是 0，如果被取走，则它是 1。在一个声明里，自旋锁被初始化为值 **SPIN_LOCK_UNLOCKED**（12789 行）；它也可以用 **spin_lock_init** 函数（12791 行）来初始化。这两者都把 **spinlock_t** 的 **lock** 成员设置成 0——也就是未锁状态。

注意 12795 行代码简洁地对公平性进行了考虑并最后抛弃了它——公平是饥饿的背面，正如我们前面已经介绍过的（使得一个 CPU 或进程饥饿应被认为是“ 不公平的 ”）。

自旋锁的加锁和解锁宏建立在 **spin_lock_string** 和 **sping_unlock_string** 函数之上，所以这一小节只对 **spin_lock_string** 和 **sping_unlock_string** 函数进行详述。其它宏如果有的话只是增加了 IRQ 加锁和解锁。

spin_lock_string

12805：这个宏的代码对于所有自旋锁加锁的宏都是相同的。它也被用于 x86 专用的 **lock_kernel** 和 **unlock_kernel** 版本之中（它们不在本书之列，不过其常规版本则是包括的——参见 10174 和 10182 行）。

12807：尝试测试和设置自旋锁的最低位，这要把内存总线锁住以便对于任何其它对同一个自旋锁的访问来说这个操作都是原子的。

12808：如果成功了，控制流程就继续向下运行；否则，**spin_lock_string** 函数向前跳转到第 12810 行（**btsl** 把这一位的原值放入 CPU 的进位标志位（Carry flag），这正是这里使

用 **jc** 的原因）。同样的技巧我们已经看到过三次了：跳转目标放在内核的单独一段中。

12811：在封闭的循环里不停地检测循环锁的最低位。注意 **btsl** 和 **testb** 以不同方式解释它们第一个操作数——对于 **btsl**，它是一个位状态（bit position），而对于 **testb**，它是一个位屏蔽（bitmask）。因此，12811 行在测试 **spin_lock_string** 曾在 12807 行已经试图设置（但失败了）的同一位，尽管一个使用**$0** 而另一个使用**$1**。

12813：该位被清除了，所以 **spin_lock_string** 应该再次夺取它。函数调转回第 12806 行。这个代码可以只用加上 **lock** 前缀的两条代码加以简化：

```
1: lock ; btsl $0, %0
   jc 1b
```

不过，使用这个简化版本的话，系统性能将明显受到损害，这因为每次循环重复内存总线都要被加锁。内核使用的版本虽然长一些，但是它可以使其它 CPU 运行的更有效，这是由于该版本只有在它有充分理由相信能够获得锁的时候才会锁住内存总线。

spin_unlock_string

12816：并不很重要：只是重新设置了自旋锁的锁定位（lock bit）。

读/写自旋锁

自旋锁的一个特殊情况就是读/写自旋锁。这里的思想是这样的：在某些情况中，我们想要允许某个对象有多个读者，但是当有一个写者正在写入这个对象时，则不允许它再有其它读者或者写者。

遵循基于 **spinlock_t** 的自旋锁的同样模式，读/写自旋锁是用 **rwlock_t**（12853 行）来代表的，它可以在有 **RW_LOCK_UNLOCKED**（12858 行）的声明里被初始化。与 **rwlock_t** 一起工作的最低级的宏是 **read_lock**、**read_unlock**、**write_lock**，以及 **write_unlock**，它们在本小节中进行描述。很明显，那些跟随在这些宏之后并建立在它们之上的宏，自然要在你理解了最初的这四个宏之后在去接触。

正如第 12860 行注释中所声明的，当写锁（write lock）被占有时，**rwlock_t** 的 **lock** 成员是负值。当既没有读者也没有写者时它为 0，当只有读者而没有写者时它是正值——在这种情况下，**lock** 将对读者的数目进行计数。

read_lock

12867：开始于 **rwlock_t** 的 **lock** 成员的自动递增。这是推测性的操作——它可以被撤销。

12868：如果它在增量之后为负，表示某个进程占用了写锁——或者至少是某个进程正试图得到它。**read_lock** 向前跳到第 12870 行（注意，在一个不同的内核段里）。否则，没有写者退出（尽管还有可能有，或者也有可能没有其它读者——这并不重要），所以可以继续执行读锁定（read-locked）代码。

12870：一个写者出现了。**read_lock** 取消第 12867 行增值操作的影响。

12871：循环等待 **rwlock_t** 的 **lock** 变为 0 或正值。

12873：跳回到第 12866 行再次尝试。

read_unlock

12878：不太复杂：只是递减该计数值。

write_lock

12883：表示出有一个进程需要写锁：检测并设置 **lock** 的符号位并保证 **lock** 的值是负的。

12884：如果符号位已经被设置，则另外有进程占有了写锁；**write_lock** 向前跳转到第 12889 行（同以前一样，那是在一个不同的内核段里）。

12885：没有别的进程正试图获得该写锁，可是读者仍可以退出。因为符号位被设置了，读者不能获得读锁，但是 **write_lock** 仍然必须等待正在退出的读者完全离开。它通过检查低端的 31 位中是否任何一位被设置过开始，这可以表示 **lock** 以前曾是正值。如果没有，则 **lock** 在符号位反转之前曾是 0，这意味着没有读者；因而，这对于写者的继续工作是很安全的，所以控制流程就可以继续向下运行了。不过，如果低端 31 位中任何一位被设置过了，也就是说有读者了，这样 **write_lock** 就会向前跳转到第 12888 行等到它们结束。

12888：该进程是仅有的写者，但是有若干读者。**write_lock** 会暂时清除符号位（这个宏稍后将再次操纵它）。有趣的是，对符号位进行这样的胡乱操作并不会影响读者操纵 **lock** 的正确性。考虑作为示例的下列顺序事件：

1. 两个读者增加了 **lock**；**lock** 用十六进制表示现在是 0x00000002。
2. 一个即将成为写者的进程设置了符号位；**lock** 现在是 0x80000002。
3. 读者中的一个离开；**lock** 现在是 0x80000001。
4. 写者看到剩余的位不全部是 0——仍然有读者存在。这样它根本没有写锁，因此它就清除符号位；**lock** 现在是 0x00000001。

这样，读和写可以任何顺序交错尝试操作而不会影响结果的正确程度。

12889：循环等待计数值降到 0——也就是等待所有读者退出。实际上，0 除了表示所有读者已离开之外，它还表示着没有其它进程获得了写锁。

12891：所有读者和写者都结束了操作；**write_lock** 又从头开始，并再次获得写锁。

write_unlock

12896：不太重要：只是重置符号位。

APICs 和 CPU-To-CPU 通信

Intel 多处理规范的核心就是高级可编程中断控制器（Advanced Programmable Interrupt Controllers——APICs）的使用。CPU 通过彼此发送中断来完成它们之间的通信。通过给中断附加动作（actions），不同的 CPU 可以在某种程度上彼此进行控制。每个 CPU 有自己的 APIC（成为那个 CPU 的本地 APIC），并且还有一个 I/O APIC 来处理由 I/O 设备引起的中断。在普通的多处理器系统中，I/O APIC 取代了第 6 章里提到的中断控制器芯片组的作用。

这里有几个示例性的函数来让你了解其工作方式的风格。

smp_send_reschedule

5019： 这个函数只有一行，其作用将在本章随后进行说明，它仅仅是给其 ID 以参数形式给出了的目标 CPU 发送一个中断。函数用 CPU ID 和 **RESCHEDULE_VECTOR** 向量调用 **send_IPI_single** 函数（4937 行）。**RESCHEDULE_VECTOR** 与其它 CPU 中断向量是一起在第 1723 行开始的一个定义块中被定义的。

send_IPI_single

4937： **send_IPI_single** 函数发送一个 IPI——那是 Intel 对处理器间中断（interprocessor interrupt）的称呼——给指定的目的 CPU。在这一行，内核以相当低级的方式与发送 CPU 的本地 APIC 对话。

4949： 得到中断命令寄存器（ICR）高半段的内容——本地 APIC 就是通过这个寄存器进行编程的——不过它的目的信息段要被设置为 **dest**。尽管**__prepare_ICR2**（4885 行）里使用了“ 2 ”，CPU 实际上只有一个 ICR 而不是两个。但是它是一个 64 位寄存器，内核更愿意把它看作是两个 32 位寄存器——在内核代码里，“ ICR ”表示这个寄存器的低端 32 位，所以“ ICR2 ”就表示高端 32 位。我们想要设置的的目的信息段就在高端 32 位，即 ICR2 里。

4950： 把修改过的信息写回 ICR。现在 ICR 知道了目的 CPU。

4953： 调用**__prepare_ICR**（4874 行）来设置我们想要发送给目的 CPU 的中断向量。（注意没有什么措施能够保证目的 CPU 不是当前 CPU——ICR 完全能够发送一个 IPI 给它自己的 CPU。尽管这样，我还是没有找到有任何理由要这样做。）

4957： 通过往 ICR 里写入新的配置来发送中断。

SMP 支持如何影响内核

既然读者已经学习了能够成功支持 SMP 的若干原语，那么就让我们来纵览一下内核的 SMP 支持吧。本章剩余的部分将局限于对分布在内核之中的那些具有代表性的 SMP 代码进行讨论。

对调度的影响

schedule（26686 行）正是内核的调度函数，它已在第 7 章中全面地介绍过了。**schedule** 的 SMP 版本与 UP 的相比有两个主要区别：

- 在 **schedule** 里从第 26780 开始的一段代码要计算某些其它地方所需的信息。
- 在 SMP 和 UP 上都要发生的对**__schedule_tail** 的调用（26638 行）实际上在 UP 上并无作用，因为**__schedule_tail** 完全是为 SMP 所写的代码，所以从实用的角度来说它就是 SMP 所特有的。

schedule

26784：获取当前时间，也就是自从机器开机后时钟流逝的周期数。这很像是检查 **jiffies**，不过是以 CPU 周期而不是以时钟滴答作为计时方法的——显然，这要精确得多。

26785：计算自从 **schedule** 上一次在此 CPU 上进行调度后过去了多长时间，并且为下一次的计算而记录下当前周期计数。（**schedule_data** 是每个 CPU **aligned_data** 数组的一部分，它在 26628 行定义。）

26790：进程的 **avg_slice** 成员（16342 行）记录该进程在其生命周期里占有 CPU 的平均时间。可是这并不是简单的平均——它是加权平均，进程近期的活动远比很久以前的活动权值大。（因为真实计算机的计算是有穷的，“ 很久以前 ”的部分在足够远以后，将逐渐趋近于 0。）这将在 **reschedule_idle** 中（26221 行，下文讨论）被用来决定是否把进程调入另一个 CPU 中。因此，在 UP 的情况下它是无需而且也不会被计算的。

26797：记录哪一个 CPU 将运行 **next**（它将在当前的 CPU 上被执行），并引发它的 **has_cpu** 标志位。

26803：如果上下文环境发生了切换，**schedule** 记录失去 CPU 的进程——这将在下文的 **__schedule_tail** 中被使用到。

__schedule_tail

26654：如果失去 CPU 的任务已经改变了状态（这一点在前边的注释里解释过了），它将被标记以便今后的重新调度。

26664：因为内核已经调度出了这个进程，它就不再拥有 CPU 了——这样的事实也将被记录。

reschedule_idle

26221：当已经不在运行队列里的进程被唤醒时，**wake_up_process** 将调用 **reschedule_idle**，进程是作为 **p** 而被传递进 **reschedule_idle** 中的。这个函数试图把新近唤醒的进程在一个不同的 CPU 上进行调度——即一个空闲的 CPU 上。

26225：这个函数的第一部分在 SMP 和 UP 场合中都是适用的。它将使高优先级的进程得到占用 CPU 的机会，同时它也会为那些处于饥饿状态的进程争取同样的机会。如果该进程是实时的或者它的动态优先级确实比当前占有 CPU 进程的动态优先级要高某个量级（强制选定的），该进程就会被标记为重新调度以便它能够争取占用 CPU。

26263：现在来到 SMP 部分，它仅仅适用于在上述测试中失败了的那些进程——虽然这种现象经常发生。**reschedule_idle** 必须确定是否要在另一个 CPU 上尝试运行该进程。
正如在对 **schedule** 的讨论中所提到的那样，一个进程的 **avg_slice** 成员是它对 CPU 使用的加权平均值；因此，它说明了假如该进程继续运行的话是否它可能要控制 CPU 一段相对来说较长的时间。

26264：这个 **if** 条件判断的第二个子句使用 **related** 宏（就在本函数之上的第 26218 行）来测试是否 CPU 都在控制着——或想要控制——内核锁。如果是这样，那么不管它们生存于何处，都将不大可能同时运行，这样把进程发送到另一个 CPU 上将不会全面提高并行的效能。因此，假如这条子句或者前一条子句被满足，函数将不会考虑使进程在另一 CPU 上进行调度并简单的返回。

26267：否则，**reschedule_idle_slow**（接下来讨论）被调用以决定是否进程应当被删除。

reschedule_idle_slow

26157：正如注释中所说明的，**reschedule_idle_slow** 试图找出一个空闲 CPU 来贮存 **p**。这个算法是基于如下观察结果的，即 **task** 数组的前 n 项是系统的空闲进程，机器的 n 个 CPU 中每个都对应一个这样的空闲进程。这些空闲进程当（且仅当）对应 CPU 上没有其它进程需要处理器时才会运行。如果可能，函数通常是用 **hlt** 指令使 CPU 进入低功耗的“睡眠”状态。
因此，如果有空闲 CPU 存在的话，对任务数组的前 n 个进程进行循环是找出一个空闲 CPU 所必须的。**reschedule_idle_slow** 函数只需简单的查询每个空闲进程是否此刻正在运行着；如果是这样，它所在的 CPU 就一定是空闲的，这就为进程 **p** 提供了一个很好的候选地点来运行。
当然，这个被选中的明显空闲的 CPU 完全有可能只是暂时空闲而且必定会被一堆拥有更高优先级的，CPU 绑定的进程所充满，这些进程可能在一纳秒后就会被唤醒并在该 CPU 上运行。所以，这并不是完美的解决方法，可是从统计的角度来说它已经

相当好了——要记住，像这样的选择是很符合调度程序“ 快餐店式（quick-and-dirty）”的处理方式的。

26180：建立本地变量。**best_cpu** 是此时正在运行的 CPU；它是“ 最佳 ”的 CPU，因为 **p** 在其上会避免缓冲区溢出或其它的开销麻烦。**this_cpu** 是运行 **reschedule_idle_slow** 的 CPU。

26182：**idle** 和 **tsk** 将沿 **task** 数组进行遍历，**target_tsk** 将是所找到的最后一个正在运行的空闲进程（或者假如没有空闲进程它就为 **NULL**）。

26183：**i** 从 **smp_num_cpus**（前边被叫作 n）开始并且在每一次循环后都递减。

26189：假如这个空闲进程的 **has_cpu** 标志被设置，它就正在它的 CPU 上运行着（我们将称这样的 CPU 为“ 目标（target）CPU ”）。如果该标志没有被设置，那么目标 CPU 就正被某个其它进程占用着；因而，它也就不是空闲的，这样 **reschedule_idle_slow** 将不会把 **p** 发送到那里。刚刚提及问题的反面在这里出现了：现在仅因为 CPU 不空闲并不能表示它所有的进程都不会死亡而使其空闲下来。可是 **reschedule_idle_slow** 无法知道这种情形，所以它最好还是假定目标 CPU 将要被占用一段时间。无论如何，这都是可能的，就算并非如此，某个其它的进程也将很快会被调度到另一个空闲 CPU 上运行。

26190：不过假如 CPU 目标就是当前 CPU，它就会被跳过。这看来很怪，不过无论怎样这都是“ 不可能发生 ”的情况：一个空闲进程的 **counter** 是负值，在第 26226 行的测试将早已阻止这个函数执行到这一步了。

26192：找到一个可用的空闲 CPU；相关的空闲进程被保存在 **target_tsk** 中。

既然已找到了空闲 CPU，为什么现在不中断循环呢？这是因为继续循环可能会发现 **p** 当前所在的处理器也是空闲的，在两个 CPU 都空闲时，维持在当前处理器上运行要比把它送往另一个好一些。

26193：这一步 **reschedule_idle_slow** 检查是否 **p** 所在的处理器空闲。如果刚才找到的空闲 CPU 就是 **p** 所在的，函数将向前跳转到 **send** 标记处（26203 行）来在那个 CPU 上对 **p** 进行调度。

26199：函数已经转向另一个 CPU；它要递减。

26204 如果循环遍历了所有空闲的 CPU，该 CPU 的空闲任务就被标记为重新调度并且 **smp_send_reschedule**（26205 行）会给那个 CPU 发送一个 IPI 以便它可以重新对其进程进行调度。

正如读者所见到的，**reschedule_idle_slow** 是 CPU 之间协调无需在 UP 系统中所进行的工作的典范示例。对于 UP 机器来说，询问进程应占有哪一个 CPU 和询问它是否应拥有系统的唯一的一个 CPU 或根本不应该占有 CPU 是等价的。SMP 机器必须花费一些代价来决定系统中哪一个 CPU 是该进程的最佳栖身之所。当然，换来的速度极大提高使得这些额外的努力还是相当合算的。

release

22951：**release** 中非 SMP 特有的部分在第 7 章中已经介绍过了——在这里，一个僵进程（zombie）将被送往坟墓，而且其 **struct task_struct** 将被释放。

22960：查看是否该进程拥有一个 CPU。（拥有它的 CPU 可能还没有清除这个标志；但是它马上就将执行这个操作。）如果没有，**release** 退出循环并像往常一样接着释放 **struct task_ struct** 结构体。

22966：否则，**release** 等待进程的 **has_cpu** 标志被清除。当它被清除后，**release** 再次进行尝试。这种貌似奇特的情况——某进程正被删除，然而它仍占有 CPU——确实少见，

不过并非不可能。进程可能已经在一个 CPU 上被杀死，而且这个 CPU 还没来得及清除 **has_cpu** 标志，但是它的父进程已经正在从另一个 CPU 对它进行释放了。

smp_local_timer_interrupt

对于 UP 专有的 **update_process_times** 函数（27382 行）来说，这个函数就是它在 SMP 上的对应。该函数能够完成 **update_process_times** 所完成的所有任务——更新进程和内核在 CPU 使用方面的统计值——以及其它的一些操作。与众不同的地方在于拥有这个特性的 SMP 版本并没有被添加到一个 UP 函数中去，而是采用了一个具有同样功能，但却完全分离的功能程序。在浏览了函数之后，我们就能够很容易的知道这是为什么了——它与 UP 版本差别甚大到以至于试图将二者融为一体都将是无意义的。**smp_local_timer_interrupt** 可从两个地方进行调用：

- 从 **smp_apic_timer_interrupt**（5118 行）调用，它用于 SMP 的时钟中断。这是通过使用在第 1856 行定义的 **BUILD_SMP_TIMER_INTERRUPT** 宏于第 919 行建立起来的。
- 从第 5776 行通常的 UP 时钟中断函数里进行调用。只有当在 UP 机器上运行 SMP 内核时此种调用方式才会发生。

smp_local_timer_interrupt

5059： **prof_counter**（4610 行）用于跟踪到更新进程和内核统计值之前内核应该等待多长时间；如果该计数器还没有到达 0，控制流程会有效地跳转到函数的末尾。正如代码中所证明的，**prof_counter** 项目从 1 开始递减计数，除非由根（root）来增加这个值，因此在缺省情况下每次时钟滴答都要完成此项工作。然后，**prof_counter[cpu]** 从 **prof_multiplier[cpu]**处被重新初始化。

明显的这是一个优化的过程：每次时钟滴答都在这个 **if** 语句块里完成所有工作将相当的缓慢，所以我们可能想到以牺牲一些精确度的代价将工作分批完成。因为乘法器是可调的，所以你可以指定你所需要的速度频率来放松对准确度的要求。

然而，关于这段代码我总感到有些困惑：确定无疑的是，当 **prof_multiplier[cpu]**耗尽时，统计值应该被更新，就像 **prof_multiplier[cpu]**的计数流逝一样——既然它们已经如此。（除了 **prof_multiplier[cpu]**本身刚刚被改变时，不过这已经偏离了这里讨论的主题。）与此不同的是，这里代码表现出来的就好像只经过了一次滴答计数。或许其用意是为了以后能把记录下来的滴答数目和 **prof_multiplier[cpu]**在某个地方相乘，不过现在并没有这样实现。

5068： 当时钟中断被触发时假如系统正在用户模式运行，**smp_local_timer_interrupt** 会假定全部滴答都是在用户模式里流逝的；否则，它将假定全部滴答是在系统模式里流逝的。

5073： 用 **irq_enter**（1792 行）来夺取全局 IRQ 锁。这是我们要分批处理这项工作的另一个原因：并不需要在每次时钟滴答时都要得到全局 IRQ 锁，这有可能成为 CPU 之间争夺的一个重要根源，实际中函数是以较低的频度来争取该锁的。因此，函数不经常夺取这个锁，可是一旦它获得了锁，就不会再使其被锁。在此我们又一次以准确度的代价换来了这种效率上的提高。

5074： 不用为保存空闲进程的统计值而操心。这样做只会浪费 CPU 的周期。总之，内核会跟踪系统处于空闲的总共时间，对空闲进程的更多细节进行统计价值不大（比如我

们知道它们总是在系统模式下执行的，所以就没有必要再明确计算它们的系统时间了）。

5075： **update_process_times** 和 **smp_local_timer_interrupt** 在这一点上是一致的：它们都调用 **update_process_times** 来完成对单进程 CPU 使用统计的更新工作。

5077： 减少进程的 **counter**（它的动态优先级），如果它被耗尽就重新调度该进程。

5082： 更新内核的统计数字。如在 **update_process_times** 中一样，用户时间既可以用内核的“ 最优时间 ”也可以用常规的用户时间来计算，这要取决于进程的优先级是否低于 **DEF_PRIORITY**。

5094： 重新初始化 CPU 的 **prof_counter** 并释放全局 IRQ 锁。该工作必须要以这种顺序完成，当然——若以相反的方式，则可能在 **prof_counter** 被重新初始化之前发生又一次时钟中断。

lock_kernel 和 unlock_kernel

这两个函数也有专门适应于 x86 平台的版本；但是在这里只介绍通用版本。

lock_kernel

10174：这个函数相当简单，它获得全局内核锁——在任何一对 **lock_kernel/unlock_kernel** 函数里至多可以有一个 CPU。显然这在 UP 机器上是一个空操作（no-op）。

10176：进程的 **lock_depth** 成员初始为–1（参见 24040 行）。在它小于 0 时（若小于 0 则恒为-1），进程不拥有内核锁；当大于或等于 0 时，进程得到内核锁。
这样，单个进程可以调用 **lock_kernel**，然后在运行到 **unlock_kernel** 之前可能又将调用另一个要使用 **lock_kernel** 的函数。在这种情况中，进程将立刻被赋予内核锁——而这正是我们所期望的。
其结果是，一旦增加进程的 **lock_depth** 就会使 **lock_depth** 为 0，那么进程以前就是没有锁的。所以，函数在此情形下获得 **kernel_flag** 自旋锁（3587 行）。

unlock_kernel

10182：同样的，如果丢弃内核锁就会使 **lock_depth** 低于 0 值，进程退出它所进入的最后一对 **lock_kernel/unlock_kernel** 函数。此时，**kernel_flag** 自旋锁一定要被解锁以便其它进程可以给内核加锁。通过测试结果的符号位（即使用“ <0 ”而不是“ == -1 ”）可以使 gcc 生成更高效的代码，除此之外，这还可能有利于内核在面对不配对的 **lock_kernel/unlock_kernel** 时可正确执行（或者不能，这取决于具体情况）。

softirq_trylock

你可能能够回忆起在第 6 章的讨论中，**softirq_trylock** 的作用是保证对于其它程序段来说下半部分代码（bottom half）是原子操作——也就是说，保证在任何特定时段的整个系统范围之内至多只有一个下半部分代码在运行。对于 UP 来说这相当容易：内核只不过需要检查或者还要设置一下标志位就可以了。不过对于 SMP 来说自然没有这样简单。

softirq_trylock

12528：测试并设置（tests-and-sets）**global_bh_count** 的第 0 位。尽管读者可能会从 **global_bh_count** 的名字上得到另外一种看法，实际它总是 0 或者 1 的——这样的考虑是适当的，因为至多运行一个下半部分程序代码。不管怎样，如果 **global_bh_count** 已经是 1 了，那么就已经有一个下半部分代码在运行着，因此控制流程就跳转到函数末尾。

12529：如果还可得到 **global_bh_lock**，那么下半部分代码就能够在这个 CPU 上运行。这种情况与 UP 机器上使用的双锁系统非常类似。

12533：**softirq_trylock** 无法获取 **global_bh_lock**，因此它的工作失败了。

cli 和 sti

正如在第 6 章中解释过的，**cli** 和 **sti** 分别用于禁止和启用中断。对于 UP 这简化为单个 **cli** 或 **sti** 指令。而在 SMP 情况下，这就很不够了，我们不仅需要禁止本地 CPU 还要暂时避免其它 CPU 处理 IRQ。因此对于 SMP，宏就变成了对**__global_cli** 和**__global_sti** 函数的调用。

__global_cli

1220： 把 CPU 的 EFLAGS 寄存器复制到本地变量 **flags** 里。

1221： x86 系统里的中断使能标志在 EFLAGS 寄存器的第 9 位——在第 1205 行解释了 EFLAG_IF_SHIFT 的定义。它被用来检测是否已经禁止了中断，这样就不再需要去禁止它们了。

1223： 禁止这个 CPU 的中断。

1224： 如果该 CPU 没有正在对 IRQ 进行处理，**__global_cli** 就调用 **get_irqlock**（1184 行）来获得全局 IRQ 锁。如果 CPU 已经在对 IRQ 进行处理了，那么正如我们马上要看到的，它已经拥有了该全局 IRQ 锁。
现在本 CPU 已经禁止了中断，而且它也拥有了全局 IRQ 锁，这样任务就完成了。

__global_sti

1233： 如果 CPU 没有正在对 IRQ 进行处理，**__global_sti** 就在**__global_cli** 中通过 **release_irqlock**（10752 行）调用来实现对全局 IRQ 锁的释放工作。如果 CPU 已经在对 IRQ 进行处理了，那么它已经拥有了该全局 IRQ 锁，正如在接下来的部分中将要解释的那样，这个锁将在其它地方被释放掉。

1235： 再次允许在本 CPU 上进行中断。

irq_enter 和 irq_exit

第 6 章中顺便提及了这两个函数的 UP 版本。包含在一对 **irq_enter/irq_exit** 之中的代码段都是原子操作，这不仅对于其它这样的代码区域是原子的，而且对于 **cli/sti** 宏对来说也是如此。

irq_enter

1794： 调用 **hardirq_enter**（10761 行）自动为本 CPU 增加全局 IRQ 计数和本地 IRQ 计数。这个函数记录了 CPU 正在处理一个 IRQ 的情况。

1795： 执行循环直到这个 CPU 得到全局 IRQ 锁为止。这就是为什么我要在前面说明如果 CPU 正在处理 IRQ，那么它就已经获得了全局 IRQ 锁的原因：到这个函数退出时，这两个特性都将被加强。对于内核代码来说，把这两个特性分离出去并没有太大的意义——它可以直接调用 **hardirq_enter**，而且也不用去争夺全局 IRQ 锁。函数只是没有这样作而已。

irq_exit

1802： 这个函数转向 **hardirq_enter** 的相反函数 **hardirq_exit**（10767 行）。顺便要提及的是，对 **irq_enter** 和 **irq_exit** 来说其 **irq** 参数都被忽略了——至少在 x86 平台上如此。