第五章

内核同步

你可以把内核看作是不断对请求进行响应的服务器，这些请求可能来自在CPU上执行的进程，也可能来自发出中断请求的外部设备。我们用这个类比来强调内核的各个部分并不是严格按照顺序依次执行的，而是采用交错执行的方式。因此，这些请求可能引起竞争条件，而我们必须采用适当的同步机制对这种情况进行控制。有关这些主题的简要介绍可以参看第一章中的“Unix 内核概述”一节。

本章开始部分我们先回顾一下内核请求是何时以交错（interleave）的方式执行以及交错程度如何。然后我们将介绍内核中所实现的基本同步机制，并说明通常情况下如何应用它们。最后，我们给出了几个实际的例子。

内核如何为不同的请求提供服务

为了更好地理解内核代码是如何执行的，我们把内核看作必须满足两种请求的侍者：一种请求来自顾客，另一种请求来自数量有限的几个不同的老板。对不同的请求，侍者采用如下的策略：

1. 老板提出请求时，如果侍者正空闲，则侍者开始为老板服务。
2. 如果老板提出请求时侍者正在为顾客服务，那么侍者停止为顾客服务，开始为老板服务。
3. 如果一个老板提出请求时侍者正在为另一个老板服务，那么侍者停止为第一个老板提供服务，而开始为第二个老板服务，服务完毕再继续为第一个老板服务。

4. 一个老板可能命令侍者停止正在为顾客提供的服务。侍者在完成对老板最近请求的服务之后，可能会暂时不理会原来的顾客而去为新选中的顾客服务。

侍者提供的服务对应于CPU处于内核态时所执行的代码。如果CPU在用户态执行，则侍者被认为处于空闲状态。

老板的请求相当于中断，而顾客的请求相当于用户态进程发出的系统调用或异常。正如我们将在第十章详细描述的，请求内核服务的用户态进程必须发出适当的指令（在80x86上是int $0x80或sysenter指令）。这些指令引起一个异常，它迫使CPU从用户态切换到内核态。在本章的其余部分，我们把系统调用和通常的异常都笼统地表示为“异常”。

细心的读者已经把前三条原则和第四章“中断和异常处理程序的嵌套执行”一节所描述的内核控制路径的嵌套联系起来了。第四条原则与Linux 2.6内核中最有趣的新特点相关，即内核抢占（kernel preemption）。

内核抢占

- 要给内核抢占下一个精确的定义简直太困难了。作为第一种尝试，我们说：如果进程正执行内核函数时，即它在内核态运行时，允许发生内核切换（被替换的进程是正执行内核函数的进程），这个内核就是抢占的。遗憾的是，在Linux中（在所有其他的操作系统中也一样），情况要复杂得多。

  - 无论在抢占内核还是非抢占内核中，运行在内核态的进程都可以自动放弃CPU，比如，其原因可能是，进程由于等待资源而不得不转入睡眠状态。我们将把这种进程切换称为计划性进程切换。但是，抢占式内核在响应引起进程切换的异步事件（例如唤醒高优先权进程的中断处理程序）的方式上与非抢占的内核是有差别的，我们将把这种进程切换称做强制性进程切换。
  - 所有的进程切换都由宏switch_to来完成。在抢占内核和非抢占内核中，当进程执行完某些具有内核功能的线程，而且调度程序被调用后，就发生进程切换。不过，在非抢占内核中，当前进程是不可能被替换的，除非它打算切换到用户态（参见第三章“执行进程切换“一节）。

因此，抢占内核的主要特点是：一个在内核态运行的进程，可能在执行内核函数期间被另外一个进程取代。

让我们举一对实例来说明抢占内核和非抢占内核的区别。

在进程A执行异常处理程序时（肯定是在内核态），一个具有较高优先级的进程B变为可执行状态。这种情况是有可能出现的，例如，发生了中断请求而且相应的处理程序唤

醒了进程B。如果内核是抢占的，就会发生强制性进程切换，让进程B取代进程A。异常处理程序的执行被暂停，直到调度程序再次选择进程A时才恢复它的执行。相反，如果内核是非抢占的，在进程A完成异常处理程序的执行之前是不会发生进程切换的，除非进程A自动放弃CPU。

再看另外一个例子，考虑一个执行异常处理程序的进程已经用完了它的时间配额（参见第七章“scheduler_tick()”函数一节）的情况。如果内核是抢占的，进程可能会立即被取代，但如果内核是非抢占的，进程继续运行直到它执行完异常处理程序或自动放弃CPU。

使内核可抢占的目的是减少用户态进程的分派延迟（*dispatch latency*），即从进程变为可执行状态到它实际开始运行之间的时间间隔。内核抢占对执行及时被调度的任务（如：硬件控制器、环境监视器、电影播放器等等）的进程确实是有好处的，因为它降低了这种进程被另一个运行在内核态的进程延迟的风险。

使Linux 2.6内核具有可抢占的特性无需对支持非抢占的旧内核在设计上做太大的改变，正如在第四章“从中断和异常返回”一节中所描述的，当被current_thread_info()宏所引用的thread_info描述符的preempt_count字段大于0时，就禁止内核抢占。如第四章中的表4-10所示，该字段的编码对应三个不同的计数器，因此它在如下任何一种情况发生时，取值都大于0：

1. 内核正在执行中断服务例程。
2. 可延迟函数被禁止（当内核正在执行软中断或tasklet时经常如此）。
3. 通过把抢占计数器设置为正数而显式地禁用内核抢占。

上面的原则告诉我们：只有当内核正在执行异常处理程序（尤其是系统调用），而且内核抢占没有被显式地禁用时，才可能抢占内核。此外，正如在第四章“从中断和异常返回”一节中所描述的，本地CPU必须打开本地中断，否则无法完成内核抢占。

表5-1中列出了一些简单的宏，它们处理preempt_count字段的抢占计数器。

表5-1：处理抢占计数器子字段的宏

宏	说明
preempt_count()	在thread_info描述符中选择preempt_count字段
preempt_disable()	使抢占计数器的值加1
preempt_enable_no_resched()	使抢占计数器的值减1

表5-1：处理抢占计数器子字段的宏（续）

宏	说明
preempt_enable()	使抢占计数器的值减1，并在thread_info描述符的TIF_NEED_RESCHED标志被置为1的情况下，调用preempt_schedule()
get_cpu()	与preempt_disable()相似，但要返回本地CPU的数量
put_cpu()	与preempt_enable()相同
put_cpu_no_resched()	与preempt_enable_no_resched()相同

preempt_enable()宏递减抢占计数器，然后检查TIF_NEED_RESCHED标志是否被设置（参见第四章中的表4-15）。在这种情况下，进程切换请求是挂起的，因此宏调用preempt_schedule()函数，它本质上执行下面的代码：

```
if (!current_thread_info->preempt_count && !irqs_disabled()) {
    current_thread_info->preempt_count = PREEMPT_ACTIVE;
    schedule();
    current_thread_info->preempt_count = 0;
}
```

该函数检查是否允许本地中断，以及当前进程的preempt_count字段是否为0，如果两个条件都为真，它就调用schedule()选择另外一个进程来运行。因此，内核抢占可能在结束内核控制路径（通常是一个中断处理程序）时发生，也可能在异常处理程序调用preempt_enable()重新允许内核抢占时发生。正如我们将在本章稍后的“禁止和激活可延迟函数”一节所看到的，内核抢占也可能发生在启用可延迟函数的时候。

在结束本节内容时，我们提醒大家注意：内核抢占会引起不容忽视的开销。因此，Linux 2.6独具特色地允许用户在编译内核时通过设置选项来禁用或启用内核抢占。

什么时候同步是必需的

第一章介绍了竞争条件和进程临界区的概念。这些定义同样适用于内核控制路径。在本章，当计算的结果依赖于两个或两个以上的交叉内核控制路径的嵌套方式时，可能出现竞争条件。临界区是一段代码，在其他的内核控制路径能够进入临界区前，进入临界区的内核控制路径必须全部执行完这段代码。

交叉内核控制路径使内核开发者的工作变得复杂了：他们必须特别小心地识别出异常处理程序、中断处理程序、可延迟函数和内核线程中的临界区。一旦临界区被确定，就必须对其采用适当的保护措施，以确保在任意时刻只有一个内核控制路径处于临界区。

例如，假设两个不同的中断处理程序要访问同一个包含了几个相关变量的数据结构，比如一个缓冲区和一个表示缓冲区大小的整型变量。所有影响该数据结构的语句都必须放入一个单独的临界区。如果是单CPU的系统，可以采取访问共享数据结构时关闭中断的方式来实现临界区，因为只有在开中断的情况下，才可能发生内核控制路径的嵌套。

另外，如果相同的数据结构仅被系统调用服务例程所访问，而且系统中只有一个CPU，就可以非常简单地通过在访问共享数据结构时禁用内核抢占功能来实现临界区。

正如你们所预料的，在多处理器系统中，情况要复杂得多。由于许多CPU可能同时执行内核路径，因此内核开发者不能假设只要禁用内核抢占功能，而且中断、异常和软中断处理程序都没有访问过该数据结构，就能保证这个数据结构能够安全地被访问。

我们将在下一节看到内核提供了各种不同的同步技术。内核设计者通过选择最有效的技术解决了所有的同步难题。

什么时候同步是不必要的

前一章所讨论的一些设计上的选择在某种程度上简化了内核控制路径的同步。让我们简单地回忆一下：

- 所有的中断处理程序响应来自PIC的中断并禁用IRQ线。此外，在中断处理程序结束之前，不允许产生相同的中断事件。
- 中断处理程序、软中断和tasklet既不可以被抢占也不能被阻塞，所以它们不可能长时间处于挂起状态。在最坏的情况下，它们的执行将有轻微的延迟，因为在其执行的过程中可能发生其他的中断（内核控制路径的嵌套执行）。
- 执行中断处理的内核控制路径不能被执行可延迟函数或系统调用服务例程的内核控制路径中断。
- 软中断和tasklet不能在一个给定的CPU上交错执行。
- 同一个tasklet不可能同时在几个CPU上执行。

以上的每一种设计选择都可以被看做是一种约束，它能使一些内核函数的编码变得更容易。下面是一些可能简化了的例子：

- 中断处理程序和tasklet不必编写成可重入的函数。
- 仅被软中断和tasklet访问的每CPU变量不需要同步。
- 仅被一种tasklet访问的数据结构不需要同步。

本章接下来的部分描述在需要同步的时候应该做些什么，也就是：如何避免由于对共享数据的不安全访问导致的数据崩溃。

同步原语

现在我们考察一下在避免共享数据之间的竞争条件时，内核控制路径是如何交错执行的。表5-2列出了Linux内核使用的同步技术。“适用范围”一栏表示同步技术是适用于系统中的所有CPU还是单个CPU。例如，本地中断的禁止只适用于一个CPU（系统中的其他CPU不受影响）；相反，原子操作影响系统中的所有CPU（当访问同一个数据结构时，几个CPU上的原子操作不能交错）。

表5-2：内核使用的各种同步技术

技术	说明	适用范围
每CPU变量	在CPU之间复制数据结构	所有CPU
原子操作	对一个计数器原子地“读－修改－写”的指令	所有CPU
内存屏障	避免指令重新排序	本地CPU或所有CPU
自旋锁	加锁时忙等	所有CPU
信号量	加锁时阻塞等待（睡眠）	所有CPU
顺序锁	基于访问计数器的锁	所有CPU
本地中断的禁止	禁止单个CPU上的中断处理	本地CPU
本地软中断的禁止	禁止单个CPU上的可延迟函数处理	本地CPU
读－拷贝－更新(RCU)	通过指针而不是锁来访问共享数据结构	所有CPU

现在，让我们简要地描述每种同步技术。在后面“对内核数据结构的同步访问”一节，我们会说明如何把这些同步技术组合在一起来有效地保护内核数据结构。

每CPU变量

最好的同步技术是把设计不需要同步的内核放在首位。正如我们将要看到的，事实上每一种显式的同步原语都有不容忽视的性能开销。

最简单也是最重要的同步技术包括把内核变量声明为每CPU变量（*per-cpu variable*）。每CPU变量主要是数据结构的数组，系统的每个CPU对应数组的一个元素。

一个CPU不应该访问与其他CPU对应的数组元素，另外，它可以随意读或修改它自己

的元素而不用担心出现竞争条件，因为它是唯一有资格这么做的CPU。但是，这也意味着每CPU变量基本上只能在特殊情况下使用，也就是当它确定在系统的CPU上的数据在逻辑上是独立的时候。

每CPU的数组元素在主存中被排列以使每个数据结构存放在硬件高速缓存的不同行(参见第二章“硬件高速缓存”一节)，因此，对每CPU数组的并发访问不会导致高速缓存行的窃用和失效（这种操作会带来昂贵的系统开销)。

虽然每CPU变量为来自不同CPU的并发访问提供保护，但对来自异步函数（中断处理程序和可延迟函数）的访问不提供保护，在这种情况下需要另外的同步原语。

此外，在单处理器和多处理器系统中，内核抢占都可能使每CPU变量产生竞争条件。总的原则是内核控制路径应该在禁用抢占的情况下访问每CPU变量。作为一个例子，简单地考虑一下在下面这种情况下会产生什么后果——一个内核控制路径获得了它的每CPU变量本地副本的地址，然后它因被抢占而转移到另外一个CPU上，但仍然引用原来CPU元素的地址。

表5-3列出了内核提供使用每CPU变量的函数和宏。

表5-3：为每CPU变量提供的函数和宏

宏或函数名	说明
DEFINE_PER_CPU(type, name)	静态分配一个每CPU数组，数组名为name,结构类型为type
per_cpu(name, cpu)	为CPU选择一个每CPU数组元素，CPU由参数cpu指定，数组名称为name
__get_cpu_var(name)	选择每CPU数组name的本地CPU元素
get_cpu_var(name)	先禁用内核抢占，然后在每CPU数组name中，为本地CPU选择元素
put_cpu_var(name)	启用内核抢占（不使用name）
alloc_percpu(type)	动态分配type类型数据结构的每CPU数组，并返回它的地址
free_percpu(pointer)	释放被动态分配的每CPU数组，pointer指示其地址
per_cpu_ptr(pointer, cpu)	返回每CPU数组中与参数cpu对应的CPU元素地址，参数pointer给出数组地址

原子操作

若干汇编语言指令具有“读－修改－写”类型——也就是说，它们访问存储器单元两次，第一次读原值，第二次写新值。

假定运行在两个CPU上的两个内核控制路径试图通过执行非原子操作来同时“读－修改－写”同一存储器单元。首先，两个CPU都试图读同一单元，但是存储器仲裁器（对访问RAM芯片的操作进行串行化的硬件电路）插手，只允许其中的一个访问而让另一个延迟。然而，当第一个读操作已经完成后，延迟的CPU从那个存储器单元正好读到同一个（旧）值。然后，两个CPU都试图向那个存储器单元写一新值，总线存储器访问再一次被存储器仲裁器串行化，最终，两个写操作都成功。但是，全局的结果是不对的，因为两个CPU写入同一（新）值。因此，两个交错的“读－修改－写”操作成了一个单独的操作。

避免由于“读－修改－写”指令引起的竞争条件的最容易的办法，就是确保这样的操作在芯片级是原子的。任何一个这样的操作都必须以单个指令执行，中间不能中断，且避免其他的CPU访问同一存储器单元。这些很小的原子操作（*atomic operations*）可以建立在其他更灵活机制的基础之上以创建临界区。

让我们根据那个分类来回顾一下80x86的指令：

- 进行零次或一次对齐内存访问的汇编指令是原子的（注1）。
- 如果在读操作之后、写操作之前没有其他处理器占用内存总线，那么从内存中读取数据、更新数据并把更新后的数据写回内存中的这些“读－修改－写”汇编语言指令（例如`inc`或`dec`）是原子的。当然，在单处理器系统中，永远都不会发生内存总线窃用的情况。
- 操作码前缀是`lock`字节（`0xf0`）的“读－修改－写”汇编语言指令即使在多处理器系统中也是原子的。当控制单元检测到这个前缀时，就“锁定”内存总线，直到这条指令执行完成为止。因此，当加锁的指令执行时，其他处理器不能访问这个内存单元。
- 操作码前缀是一个`rep`字节（`0xf2`，`0xf3`）的汇编语言指令不是原子的，这条指令强行让控制单元多次重复执行相同的指令。控制单元在执行新的循环之前要检查挂起的中断。

注1：当数据项的地址是以字节为单位的整数倍时，数据项在内存中被对齐。例如，一个对齐的短整数的地址必须是2的整数倍，而对齐的整数的地址必须是4的整数配。一般来说，非对齐的内存访问不是原子的。

在你编写C代码程序时，并不能保证编译器会为a=a+1或甚至像a++这样的操作使用一个原子指令。因此，Linux内核提供了一个专门的atomic_t类型（一个原子访问计数器）和一些专门的函数和宏（参见表5-4），这些函数和宏作用于atomic_t类型的变量，并当作单独的、原子的汇编语言指令来使用。在多处理器系统中，每条这样的指令都有一个lock字节的前缀。

表5-4：Linux中的原子操作

函数	说明
atomic_read(v)	返回 *v
atomic_set(v,i)	把 *v 置成 i
atomic_add(i,v)	给 *v 增加 i
atomic_sub(i,v)	从 *v 中减去 i
atomic_sub_and_test(i, v)	从 *v 中减去 i，如果结果为0，则返回1；否则，返回0
atomic_inc(v)	把1加到 *v
atomic_dec(v)	从 *v 减1
atomic_dec_and_test(v)	从 *v 减1，如果结果为0，则返回1；否则，返回0
atomic_inc_and_test(v)	把1加到 *v，如果结果为0，则返回1；否则，返回0
atomic_add_negative(i, v)	把 i 加到 *v，如果结果为负，则返回1；否则，返回0
atomic_inc_return(v)	把1加到 *v，返回 *v 的新值
atomic_dec_return(v)	从 *v 减1，返回 *v 的新值
atomic_add_return(i, v)	把 i 加到 *v，返回 *v 的新值
atomic_sub_return(i, v)	从 *v 减 i，返回 *v 的新值

另一类原子函数操作作用于位掩码（参见表5-5）。

表5-5：Linux中的原子位处理函数

函数	说明
test_bit(nr, addr)	返回 *addr 的第 nr 位的值
set_bit(nr, addr)	设置 *addr 的第 nr 位
clear_bit(nr, addr)	清 *addr 的第 nr 位
change_bit(nr, addr)	转换 *addr 的第 nr 位
test_and_set_bit(nr, addr)	设置 *addr 的第 nr 位，并返回它的原值
test_and_clear_bit(nr, addr)	清 *addr 的第 nr 位，并返回它的原值
test_and_change_bit(nr, addr)	转换 *addr 的第 nr 位，并返回它的原值

表5-5：Linux中的原子位处理函数（续）

函数	说明
atomic_clear_mask(mask, addr)	清mask指定的*addr的所有位
atomic_set_mask(mask, addr)	设置mask指定的*addr的所有位

优化和内存屏障

当使用优化的编译器时，你千万不要认为指令会严格按它们在源代码中出现的顺序执行。例如，编译器可能重新安排汇编语言指令以使寄存器以最优的方式使用。此外，现代CPU通常并行地执行若干条指令，且可能重新安排内存访问。这种重新排序可以极大地加速程序的执行。

然而，当处理同步时，必须避免指令重新排序。如果放在同步原语之后的一条指令在同步原语本身之前执行，事情很快就会变得失控。事实上，所有的同步原语起优化和内存屏障的作用。

优化屏障（*optimization barrier*）原语保证编译程序不会混淆放在原语操作之前的汇编语言指令和放在原语操作之后的汇编语言指令，这些汇编语言指令在C中都有对应的语句。在Linux中，优化屏障就是barrier()宏，它展开为asm volatile("":::"memory")。指令asm告诉编译程序要插入汇编语言片段（这种情况下为空）。volatile关键字禁止编译器把asm指令与程序中的其他指令重新组合。memory关键字强制编译器假定RAM中的所有内存单元已经被汇编语言指令修改；因此，编译器不能使用存放在CPU寄存器中的内存单元的值来优化asm指令前的代码。注意，优化屏障并不保证不使当前CPU把汇编语言指令混在一起执行——这是内存屏障的工作。

内存屏障（*memory barrier*）原语确保，在原语之后的操作开始执行之前，原语之前的操作已经完成。因此，内存屏障类似于防火墙，让任何汇编语言指令都不能通过。

在80x86处理器中，下列种类的汇编语言指令是“串行的”，因为它们起内存屏障的作用：

- 对I/O端口进行操作的所有指令。
- 有lock前缀的所有指令（参见“原子操作”一节）。
- 写控制寄存器、系统寄存器或调试寄存器的所有指令（例如，cli和sti，用于修改eflags寄存器的IF标志的状态）。
- 在Pentium 4微处理器中引入的汇编语言指令lfence、sfence和mfence，它们分别有效地实现读内存屏障、写内存屏障和读－写内存屏障。

- 少数专门的汇编语言指令，终止中断处理程序或异常处理程序的iret指令就是其中的一个。

Linux使用六个内存屏障原语，如表5-6所示。这些原语也被当作优化屏障，因为我们必须保证编译程序不在屏障前后移动汇编语言指令。“读内存屏障”仅仅作用于从内存读的指令，而“写内存屏障”仅仅作用于写内存的指令。内存屏障既用于多处理器系统，也用于单处理器系统。当内存屏障应该防止仅出现于多处理器系统上的竞争条件时，就使用smp_xxx()原语；在单处理器系统上，它们什么也不做。其他的内存屏障防止出现在单处理器和多处理器系统上的竞争条件。

表5-6：Linux中的内存屏障

宏	说明
mb()	适用于MP和UP的内存屏障
rmb()	适用于MP和UP的读内存屏障
wmb()	适用于MP和UP的写内存屏障
smp_mb()	仅适用于MP的内存屏障
smp_rmb()	仅适用于MP的读内存屏障
smp_wmb()	仅适用于MP的写内存屏障

内存屏障原语的实现依赖于系统的体系结构。在80x86微处理器上，如果CPU支持lfence汇编语言指令，就把rmb()宏展开为asm volatile("lfence")，否则就展开为asm volatile("lock;addl $0,0(%%esp)":::"memory")。asm指令告诉编译器插入一些汇编语言指令并起优化屏障的作用。lock;addl $0,0(%%esp)汇编指令把0加到栈顶的内存单元；这条指令本身没有价值，但是，lock前缀使得这条指令成为CPU的一个内存屏障。

Intel上的wmb()宏实际上更简单，因为它展开为barrier()。这是因为Intel处理器从不对写内存访问重新排序，因此，没有必要在代码中插入一条串行化汇编指令。不过，这个宏禁止编译器重新组合指令。

注意，在多处理器系统上，在前一节“原子操作”中描述的所有原子操作都起内存屏障的作用，因为它们使用了lock字节。

自旋锁

一种广泛应用的同步技术是加锁（*locking*）。当内核控制路径必须访问共享数据结构或进入临界区时，就需要为自己获取一把“锁”。由锁机制保护的资源非常类似于限制于房间内的资源，当某人进入房间时，就把门锁上。如果内核控制路径希望访问资源，就

试图获取钥匙“打开门”。当且仅当资源空闲时，它才能成功。然后，只要它还想使用这个资源，门就依然锁着。当内核控制路径释放了锁时，门就打开，另一个内核控制路径就可以进入房间。

图5-1显示了锁的使用。5个内核控制路径（P_0，P_1，P_2，P_3和P_4）试图访问两个临界区（C_1和C_2）。内核控制路径P_0正在C_1中，而P_2和P_4正等待进入C_1。同时，P_1正在C_2中，而P_3正在等待进入C_2。注意P_0和P_1可以并行运行。临界区C_3的锁现在打开着，因为没有内核控制路径需要进入C_3。

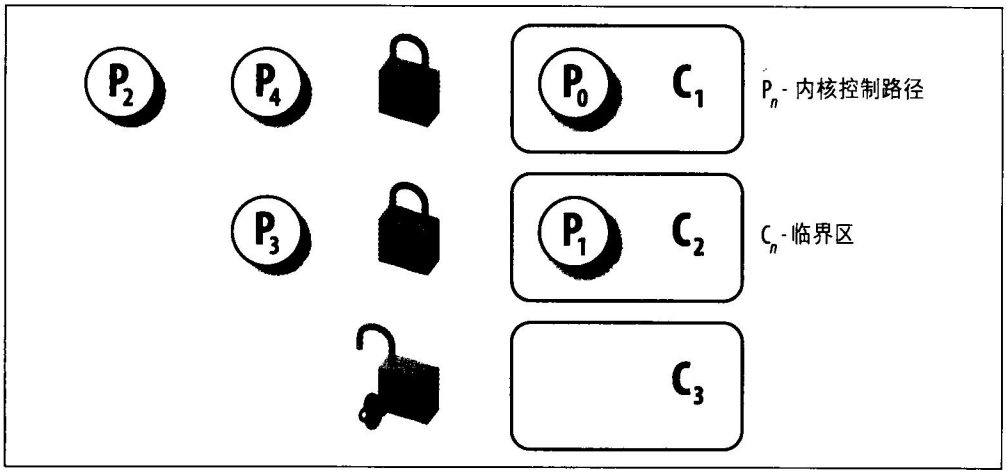

图5-1：用几个锁保护临界区

自旋锁（*spin lock*）是用来在多处理器环境中工作的一种特殊的锁。如果内核控制路径发现自旋锁“开着”，就获取锁并继续自己的执行。相反，如果内核控制路径发现锁由运行在另一个CPU上的内核控制路径“锁着”，就在周围“旋转”，反复执行一条紧凑的循环指令，直到锁被释放。

自旋锁的循环指令表示“忙等”。即使等待的内核控制路径无事可做（除了浪费时间），它也在CPU上保持运行。不过，自旋锁通常非常方便，因为很多内核资源只锁1毫秒的时间片段；所以说，释放CPU和随后又获得CPU都不会消耗多少时间。

一般来说，由自旋锁所保护的每个临界区都是禁止内核抢占的。在单处理器系统上，这种锁本身并不起锁的作用，自旋锁原语仅仅是禁止或启用内核抢占。请注意，在自旋锁忙等期间，内核抢占还是有效的，因此，等待自旋锁释放的进程有可能被更高优先级的进程替代。

在Linux中，每个自旋锁都用`spinlock_t`结构表示，其中包含两个字段：

slock

该字段表示自旋锁的状态：值为1表示“未加锁”状态，而任何负数和0都表示“加锁”状态。

break_lock

表示进程正在忙等自旋锁（只在内核支持SMP和内核抢占的情况下使用该标志）。

表5-7所示的六个宏用于初始化、测试及设置自旋锁。所有这些宏都是基于原子操作的，这样可以保证即使有多个运行在不同CPU上的进程试图同时修改自旋锁，自旋锁也能够被正确地更新（注2）。

表5-7：自旋锁宏

宏	说明
spin_lock_init()	把自旋锁置为1（未锁）
spin_lock()	循环，直到自旋锁变为1（未锁），然后，把自旋锁置为0（锁上）
spin_unlock()	把自旋锁置为1（未锁）
spin_unlock_wait()	等待，直到自旋锁变为1（未锁）
spin_is_locked()	如果自旋锁被置为1（未锁），返回0；否则，返回1
spin_trylock()	把自旋锁置为0（锁上），如果原来锁的值是1，则返回1；否则，返回0

具有内核抢占的spin_lock宏

让我们来详细讨论用于请求自旋琐的spin_lock宏。下面的描述都是针对支持SMP系统的抢占式内核的。该宏获取自旋琐的地址slp作为它的参数，并执行下面的操作：

1. 调用preempt_disable()以禁用内核抢占。
2. 调用函数_raw_spin_trylock()，它对自旋锁的slock字段执行原子性的测试和设置操作。该函数首先执行等价于下列汇编语言片段的一些指令：

```
movb $0, %al
xchgb %al, slp->slock
```

汇编语言指令xchg原子性地交换8位寄存器%al（存0）和slp->slock指示的内存单元的内容。随后，如果存放在自旋锁中的旧值（在xchg指令执行之后存放在%al中）是正数，函数就返回1，否则返回0。

注2： 具有讽刺意味的是，自旋锁是全局的，因此对它本身必须进行保护以防止并发访问。

3. 如果自旋锁中的旧值是正数，宏结束：内核控制路径已经获得自旋锁。

4. 否则，内核控制路径无法获得自旋锁，因此宏必须执行循环一直到在其他CPU上运行的内核控制路径释放自旋锁。调用preempt_enable()递减在第1步递增了的抢占计数器。如果在执行spin_lock宏之前内核抢占被启用，那么其他进程此时可以取代等待自旋锁的进程。

5. 如果break_lock字段等于0，则把它设置为1。通过检测该字段，拥有锁并在其他CPU上运行的进程可以知道是否有其他进程在等待这个锁。如果进程把持某个自旋锁的时间太长，它可以提前释放锁以使等待相同自旋锁的进程能够继续向前运行。

6. 执行等待循环：

```
while (spin_is_locked(slp) && slp->break_lock)
   cpu_relax();
```

   宏cpu_relax()简化为一条pause汇编语言指令。在Pentium 4模型中引入了这条指令以优化自旋锁循环的执行。通过引入一个很短的延迟，加快了紧跟在锁后面的代码的执行并减少了能源消耗。pause与早先的80x86微处理器模型是向后兼容的，因为它对应rep；nop指令，也就是对应空操作。

7. 跳转回到第1步，再次试图获取自旋锁。

非抢占式内核中的spin_lock宏

如果在内核编译时没有选择内核抢占选项，spin_lock宏就与前面描述的spin_lock宏有很大的区别。在这种情况下，宏生成一个汇编语言程序片段，它本质上等价于下面紧凑的忙等待（注3）：

```
1: lock; decb slp->slock
   jns  3f
2: pause
   cmpb $0,slp->slock
   jle 2b
   jmp 1b
3:
```

汇编语言指令decb递减自旋锁的值，该指令是原子的，因为它带有lock字节前缀。随后检测符号标志，如果它被清0，说明自旋锁被设置为1（未锁），因此从标记3处继续正常执行（后缀f表示标签是“向前的”，它在程序的后面出现）。否则，在标签2处（后

注3：忙等待循环的实际实现要稍微复杂些。标号2处的代码（仅在自旋锁忙时被执行）包含在另外的代码段中，以便在大多数情况下（自旋锁已经被释放）不要用不执行的代码填充硬件高速缓存。在我们的讨论中，忽略了这些优化细节。

缀b表示“向后的”标签）执行紧凑循环直到自旋锁出现正值。然后从标签1处开始重新执行，因为不检查其他的处理器是否抢占了锁就继续执行是不安全的。

spin_unlock宏

spin_unlock宏释放以前获得的自旋锁，它本质上执行下列汇编语言指令：

```
movb $1, slp->slock
```

并在随后调用preempt_enable()（如果不支持内核抢占，preempt_enable()什么都不做）。注意，因为现在的80x86微处理器总是原子地执行内存中的只写访问，所以不使用lock字节。

读/写自旋锁

读/写自旋锁的引入是为了增加内核的并发能力。只要没有内核控制路径对数据结构进行修改，读/写自旋锁就允许多个内核控制路径同时读同一数据结构。如果一个内核控制路径想对这个结构进行写操作，那么它必须首先获取读/写锁的写锁，写锁授权独占访问这个资源。当然，允许对数据结构并发读可以提高系统性能。

图5-2显示有两个受读/写锁保护的临界区（C_1和C_2）。内核控制路径R_0和R_1正在同时读取C_1中的数据结构，而W_0正等待获取写锁。内核控制路径W_1正对C_2中的数据结构进行写操作，而R_2和W_2分别等待获取读锁和写锁。

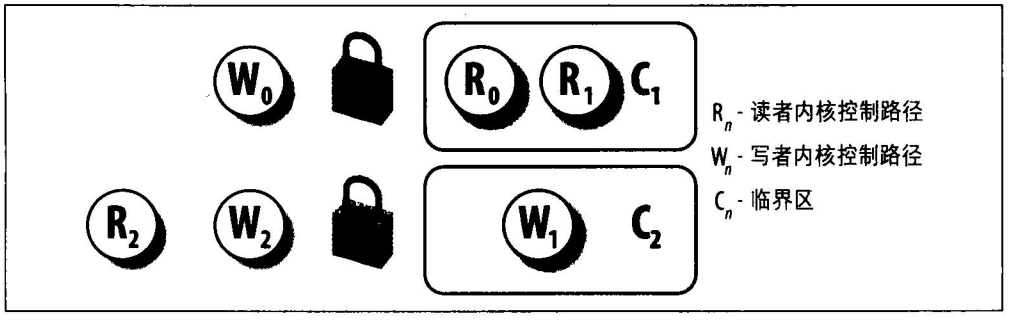

图5-2：读/写自旋锁

每个读/写自旋锁都是一个rwlock_t结构，其lock字段是一个32位的字段，分为两个不同的部分：

- 24位计数器，表示对受保护的数据结构并发地进行读操作的内核控制路径的数目。这个计数器的二进制补码存放在这个字段的0～23位。

- “未锁”标志字段，当没有内核控制路径在读或写时设置该位，否则清0。这个“未锁”标志存放在lock字段的第24位。

注意，如果自旋锁为空（设置了“未锁”标志且无读者），那么lock字段的值为0x01000000；如果写者已经获得自旋锁（“未锁”标志清0且无读者），那么lock字段的值为0x00000000；如果一个、两个或多个进程因为读获取了自旋锁，那么，lock字段的值为0x00ffffff，0x00fffffe等（“未锁”标志清0，读者个数的二进制补码在0～23位上）。与spinlock_t结构一样，rwlock_t结构也包括break_lock字段。

rwlock_init宏把读/写自旋锁的lock字段初始化为0x01000000（“未锁”），把break_lock初始化为0。

为读获取和释放一个锁

read_lock宏，作用于读/写自旋锁的地址rwlp，与前面一节所描述的spin_lock宏非常相似。如果编译内核时选择了内核抢占选项，read_lock宏执行与spin_lock()非常相似的操作，只有一点不同：该宏执行_raw_read_trylock()函数以在第2步有效地获取读/写自旋锁。

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

读/写锁计数器lock字段是通过原子操作来访问的。注意，尽管如此，但整个函数对计数器的操作并不是原子性的。例如，在用if语句完成对计数器值的测试之后并返回1之前，计数器的值可能发生变化。不过，函数能够正常工作：实际上，只有在递减之前计数器的值不为0或负数的情况下，函数才返回1，因为计数器等于0x01000000表示没有任何进程占用锁，等于0x00ffffff表示有一个读者，等于0x00000000表示有一个写者。

如果编译内核时没有选择内核抢占选项，read_lock宏产生下面的汇编语言代码：

```
    movl $rwlp->lock,%eax
    lock; subl $1,(%eax)
    jns 1f
    call _ _read_lock_failed
1:
```

这里，__read_lock_failed()是下列汇编语言函数：

```
__read_lock_failed:
    lock; incl (%eax)
1:  pause
    cmpl $1,(%eax)
    js 1b
    lock; decl (%eax)
    js _ _read_lock_failed
    ret
```

read_lock宏原子地把自旋锁的值减1，由此增加读者的个数。如果递减操作产生一个非负值，就获得自旋锁，否则，调用__read_lock_failed()函数。该函数原子地增加lock字段以取消由read_lock宏执行的递减操作，然后循环，直到lock字段变为正数（大于或等于0）。接下来，__read_lock_failed()又试图获取自旋锁（正好在cmpl指令之后，另一个内核控制路径可能为写获取自旋锁）。

释放读自旋锁是相当简单的，因为read_unlock宏只需使用汇编语言指令简单地增加lock字段的计数器：

```
lock; incl rwlp->lock
```

以减少读者的计数，然后调用preempt_enable()重新启用内核抢占。

为写获取和释放一个锁

write_lock宏实现的方式与spin_lock()和read_lock()相似。例如，如果支持内核抢占，则该函数禁用内核抢占并通过调用_raw_write_trylock()立即获得锁。如果该函数返回0，说明锁已经被占用，因此，该宏像前面章节描述的那样重新启用内核抢占并开始忙等待循环。

_raw_write_trylock()函数描述如下：

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

函数_raw_write_trylock()从读/写自旋锁的值中减去0x01000000，从而清除未上锁标志（第24位）。如果减操作产生0值（没有读者），则获取锁并返回1；否则，函数原子地在自旋锁的值上加0x01000000，以取消减操作。

释放写锁同样非常简单，因为write_unlock宏只需使用汇编语言指令lock; addl

$0x01000000,rwlp把lock字段中的“未锁”标识置位，然后再调用preempt_enable()即可。

顺序锁

当使用读/写自旋锁时，内核控制路径发出的执行read_lock或write_lock操作的请求具有相同的优先权：读者必须等待，直到写操作完成。同样地，写者也必须等待，直到读操作完成。

Linux 2.6中引入了顺序锁（*seqlock*），它与读/写自旋锁非常相似，只是它为写者赋予了较高的优先级：事实上，即使在读者正在读的时候也允许写者继续运行。这种策略的好处是写者永远不会等待（除非另外一个写者正在写），缺点是有些时候读者不得不反复多次读相同的数据直到它获得有效的副本。

每个顺序锁都是包括两个字段的seqlock_t结构：一个类型为spinlock_t的lock字段和一个整型的sequence字段，第二个字段是一个顺序计数器。每个读者都必须在读数据前后两次读顺序计数器，并检查两次读到的值是否相同，如果不相同，说明新的写者已经开始写并增加了顺序计数器，因此暗示读者刚读到的数据是无效的。

通过把SEQLOCK_UNLOCKED赋给变量seqlock_t或执行seqlock_init宏，把seqlock_t变量初始化为“未上锁”。写者通过调用write_seqlock()和write_sequnlock()获取和释放顺序锁。第一个函数获取seqlock_t数据结构中的自旋锁，然后使顺序计数器加1；第二个函数再次增加顺序计数器，然后释放自旋锁。这样可以保证写者在写的过程中，计数器的值是奇数，并且当没有写者在改变数据的时候，计数器的值是偶数。读者执行下面的临界区代码：

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* ...临界区... */
} while (read_seqretry(&seqlock, seq));
```

read_seqbegin()返回顺序锁的当前顺序号；如果局部变量seq的值是奇数（写者在read_seqbegin()函数被调用后，正更新数据结构），或seq的值与顺序锁的顺序计数器的当前值不匹配（当读者正执行临界区代码时，写者开始工作），read_seqretry()就返回1。

注意，当读者进入临界区时，不必禁用内核抢占；另一方面，由于写者获取自旋锁，所以它进入临界区时自动禁用内核抢占。

并不是每一种资源都可以使用顺序锁来保护。一般来说，必须在满足下述条件时才能使用顺序锁：

- 被保护的数据结构不包括被写者修改和被读者间接引用的指针（否则，写者可能在读者的眼鼻下就修改指针）。
- 读者的临界区代码没有副作用（否则，多个读者的操作会与单独的读操作有不同的结果）。

此外，读者的临界区代码应该简短，而且写者应该不常获取顺序锁，否则，反复的读访问会引起严重的开销。在Linux 2.6中，使用顺序锁的典型例子包括保护一些与系统时间处理相关的数据结构（参见第六章）。

读－拷贝－更新（RCU）

读－拷贝－更新(*RCU*)是为了保护在多数情况下被多个CPU读的数据结构而设计的另一种同步技术。RCU允许多个读者和写者并发执行（相对于只允许一个写者执行的顺序锁有了改进）。而且，RCU是不使用锁的，就是说，它不使用被所有CPU共享的锁或计数器，在这一点上与读/写自旋锁和顺序锁（由于高速缓存行窃用和失效而有很高的开销）相比，RCU具有更大的优势。

RCU是如何不使用共享数据结构而令人惊讶地实现多个CPU同步呢？其关键的思想包括限制RCP的范围，如下所述：

1. RCU只保护被动态分配并通过指针引用的数据结构。
2. 在被RCU保护的临界区中，任何内核控制路径都不能睡眠。

当内核控制路径要读取被RCU保护的数据结构时，执行宏`rcu_read_lock()`，它等同于`preempt_disable()`。接下来，读者间接引用该数据结构指针所对应的内存单元并开始读这个数据结构。正如在前面所强调的，读者在完成对数据结构的读操作之前，是不能睡眠的。用等同于`preempt_enable()`的宏`rcu_read_unlock()`标记临界区的结束。

我们可以想象，由于读者几乎不做任何事情来防止竞争条件的出现，所以写者不得不做得更多一些。事实上，当写者要更新数据结构时，它间接引用指针并生成整个数据结构的副本。接下来，写者修改这个副本。一但修改完毕，写者改变指向数据结构的指针，以使它指向被修改后的副本。由于修改指针值的操作是一个原子操作，所以旧副本和新副本对每个读者或写者都是可见的，在数据结构中不会出现数据崩溃。尽管如此，还需要内存屏障来保证：只有在数据结构被修改之后，已更新的指针对其他CPU才是可见的。如果把自旋锁与RCU结合起来以禁止写者的并发执行，就隐含地引入了这样的内存屏障。

然而，使用RCU技术的真正困难在于：写者修改指针时不能立即释放数据结构的旧副本。实际上，写者开始修改时，正在访问数据结构的读者可能还在读旧副本。只有在CPU上的所有（潜在的）读者都执行完宏rcu_read_unlock()之后，才可以释放旧副本。内核要求每个潜在的读者在下面的操作之前执行rcu_read_unlock()宏：

- CPU执行进程切换（参见前面的约束条件2）
- CPU开始在用户态执行
- CPU执行空循环（参见第三章“内核线程”一节）

对上述每种情况，我们说CPU已经经过了静止状态（*quiescent state*）。

写者调用函数call_rcu()来释放数据结构的旧副本。当所有的CPU都通过静止状态之后，call_rcu()接受rcu_head描述符（通常嵌在要被释放的数据结构中）的地址和将要调用的回调函数的地址作为参数。一旦回调函数被执行，它通常释放数据结构的旧副本。

函数call_rcu()把回调函数和其参数的地址存放在rcu_head描述符中，然后把描述符插入回调函数的每CPU（per-CPU）链表中。内核每经过一个时钟滴答（参见第六章“更新本地CPU统计数”一节）就周期性地检查本地CPU是否经过了一个静止状态。如果所有CPU都经过了静止状态，本地tasklet（它的描述符存放在每CPU变量rcu_tasklet中）就执行链表中的所有回调函数。

RCU是Linux 2.6中新加的功能，用在网络层和虚拟文件系统中。

信号量

我们在第一章“同步和临界区”一节引入了信号量。从本质上说，它们实现了一个加锁原语，即让等待者睡眠，直到等待的资源变为空闲。

实际上，Linux提供两种信号量：

- 内核信号量，由内核控制路径使用
- System V IPC信号量，由用户态进程使用

在本节，我们集中讨论内核信号量，而IPC信号量将在第十九章描述。

内核信号量类似于自旋锁，因为当锁关闭着时，它不允许内核控制路径继续进行。然而，当内核控制路径试图获取内核信号量所保护的忙资源时，相应的进程被挂起。只有在资源被释放时，进程才再次变为可运行的。因此，只有可以睡眠的函数才能获取内核信号量；中断处理程序和可延迟函数都不能使用内核信号量。

内核信号量是 struct semaphore 类型的对象，包含下面这些字段：

count

存放 atomic_t 类型的一个值。如果该值大于 0，那么资源就是空闲的，也就是说，该资源现在可以使用。相反，如果 count 等于 0，那么信号量是忙的，但没有进程等待这个被保护的资源。最后，如果 count 为负数，则资源是不可用的，并至少有一个进程等待资源。

wait

存放等待队列链表的地址，当前等待资源的所有睡眠进程都放在这个链表中。当然，如果 count 大于或等于 0，等待队列就为空。

sleepers

存放一个标志，表示是否有一些进程在信号量上睡眠。我们很快就会看到这个字段的作用。

可以用 init_MUTEX() 和 init_MUTEX_LOCKED() 函数来初始化互斥访问所需的信号量：这两个宏分别把 count 字段设置成 1（互斥访问的资源空闲）和 0（对信号量进行初始化的进程当前互斥访问的资源忙）。宏 DECLARE_MUTEX 和 DECLARE_MUTEX_LOCKED 完成同样的功能，但它们也静态分配 semaphore 结构的变量。注意，也可以把信号量中的 count 初始化为任意的正数值 n，在这种情况下，最多有 n 个进程可以并发地访问这个资源。

获取和释放信号量

让我们从如何释放一个信号量来开始讨论，这比获取一个信号量要简单得多。当进程希望释放内核信号量锁时，就调用 up() 函数。这个函数本质上等价于下列汇编语言片段：

```
        movl $sem->count,%ecx
        lock; incl (%ecx)
        jg 1f
        lea %ecx,%eax
        pushl %edx
        pushl %ecx
        call _ _up
        popl %ecx
        popl %edx
1:
```

这里 __up() 是下列 C 函数：

```
__attribute__((regparm(3))) void __up(struct semaphore *sem)
{
    wake_up(&sem->wait);
}
```

up()函数增加*sem信号量count字段的值，然后，检查它的值是否大于0。count的增加及其后jump指令所测试的标志的设置都必须原子地执行；否则，另一个内核控制路径有可能同时访问这个字段的值，这会导致灾难性的后果。如果count大于0，说明没有进程在等待队列上睡眠，因此，什么事也不做。否则，调用__up()函数以唤醒一个睡眠的进程。注意，__up()从eax寄存器接受参数（参见第三章“执行进程切换”一节中对函数__switch_to()的说明）。

相反，当进程希望获取内核信号量锁时，就调用down()函数。down()的实现相当棘手，但本质上等价于下列代码：

```
down:
    movl $sem->count,%ecx
    lock; decl (%ecx);
    jns 1f
    lea %ecx, %eax
    pushl %edx
    pushl %ecx
    call __down
    popl %ecx
    popl %edx
1:
```

这里__down()是下列C函数：

```
__attribute__((regparm(3))) void __down(struct semaphore * sem)
{
    DECLARE_WAITQUEUE(wait, current);
    unsigned long flags;
    current->state = TASK_UNINTERRUPTIBLE;
    spin_lock_irqsave(&sem->wait.lock, flags);
    add_wait_queue_exclusive_locked(&sem->wait, &wait);
    sem->sleepers++;
    for (;;) {
        if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
            sem->sleepers = 0;
            break;
        }
        sem->sleepers = 1;
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        schedule();
        spin_lock_irqsave(&sem->wait.lock, flags);
        current->state = TASK_UNINTERRUPTIBLE;
    }
    remove_wait_queue_locked(&sem->wait, &wait);
    wake_up_locked(&sem->wait);
    spin_unlock_irqrestore(&sem->wait.lock, flags);
    current->state = TASK_RUNNING;
}
```

down()函数减少*sem信号量的count字段的值，然后检查该值是否为负。该值的减少和检查过程都必须是原子的。如果count大于或等于0，当前进程获得资源并继续正常执行。否则，count为负，当前进程必须挂起。把一些寄存器的内容保存在栈中，然后调用__down()。

从本质上说，__down()函数把当前进程的状态从TASK_RUNNING改变为TASK_UNINTERRUPTIBLE，并把进程放在信号量的等待队列。该函数在访问信号量结构的字段之前，要获得用来保护信号量等待队列的sem->wait.lock自旋锁（参见第三章"如何组织进程"），并禁止本地中断。通常当插入和删除元素时，等待队列函数根据需要获取和释放等待队列的自旋锁。函数__down()也用等待队列自旋锁来保护信号量数据结构的其他字段，以使在其他CPU上运行的进程不能读或修改这些字段。最后，__down()使用等待队列函数的"_locked"版本，它假设在调用等待队列函数之前已经获得了自旋锁。

__down()函数的主要任务是挂起当前进程，直到信号量被释放。然而，要实现这种想法是并不容易。为了容易地理解代码，要牢记如果没有进程在信号量等待队列上睡眠，则信号量的sleepers字段通常被置为0，否则被置为1。让我们通过考虑几种典型的情况来解释代码。

MUTEX 信号量打开(count 等于 1, sleepers 等于 0)

down宏仅仅把count字段置为0，并跳到主程序的下一条指令；因此，__down()函数根本就不执行。

MUTEX 信号量关闭，没有睡眠进程 (count 等于 0, sleepers 等于 0)

down宏减count并将count字段置为 -1且sleepers字段置为0来调用__down()函数。在循环体的每次循环中，该函数检查count字段是否为负。（因为当调用atomic_add_negative()函数时，sleepers等于0，因此atomic_add_negative()不改变count字段。）

- 如果count字段为负，__down()就调用schedule()挂起当前进程。count字段仍然置为-1，而sleepers字段置为1。随后，进程在这个循环内恢复自己的运行并又进行测试。
- 如果count字段不为负，则把sleepers置为0，并从循环退出。__down()试图唤醒信号量等待队列中的另一个进程（但在我们的情景中，队列现在为空），并终止保持的信号量。在退出时，count字段和sleepers字段都置为0，这表示信号量关闭且没有进程等待信号量。

MUTEX 信号量关闭，有其他睡眠进程 (count 等于 -1, sleepers 等于 1)

down宏减count并将count字段置为 -2且sleepers字段置为1来调用__down()函数。该函数暂时把sleepers置为2，然后通过把sleepers-1加到count来取

消由down宏执行的减操作。同时，该函数检查count是否依然为负(在__down()进入临界区之前，持有信号量的进程可能正好释放了信号量)。

- 如果count字段为负，__down()函数把sleepers重新设置为1，并调用schedule()挂起当前进程。count字段还是置为－1，而sleepers字段置为1。
- 如果count字段不为负，__down()函数把sleepers置为0，试图唤醒信号量等待队列上的另一个进程，并退出持有的信号量。在退出时，count字段置为0且sleepers字段置为0。这两个字段的值看起来错了，因为还有其他的进程在睡眠。然而，考虑一下在等待队列上的另一个进程已经被唤醒。这个进程进行循环体的另一个次循环；atomic_add_negative()函数从count中减去1，count重新变为-1；此外，唤醒的进程在重新回去睡眠之前，把sleepers重置为1。

可以很容易地验证，代码在所有的情况下都正确地工作。考虑一下，__down()中的wake_up()函数至多唤醒一个进程，因为等待队列中的睡眠进程是互斥的(参见第三章“如何组织进程”一节)。

只有异常处理程序，特别是系统调用服务例程，才可以调用down()函数。中断处理程序或可延迟的函数不必调用down()，因为当信号量忙时，这个函数挂起进程。由于这个原因，Linux提供了down_trylock()函数，前面提及的异步函数可以安全地使用down_trylock()函数。该函数和down()函数除了对资源繁忙情况的处理有所不同外，其他都是相同的。在资源繁忙时，该函数会立即返回，而不是让进程去睡眠。

系统中还定义了一个略有不同的函数，即down_interruptible()。该函数广泛地用在设备驱动程序中，因为如果进程接收了一个信号但在信号量上被阻塞，就允许进程放弃“down”操作。如果睡眠进程在获得需要的资源之前被一个信号唤醒，那么该函数就会增加信号量的count字段的值并返回-EINTR。另一方面，如果down_interruptible()正常结束并得到了需要的资源，就返回0。因此，在返回值是-EINTR时，设备驱动程序可以放弃I/O操作。

最后，因为进程通常发现信号量处于打开状态，因此，就可以优化信号量函数。尤其是，如果信号量等待队列为空，up()函数就不执行跳转指令；同样，如果信号量是打开的，down()函数就不执行跳转指令。信号量实现的复杂性是由于极力在执行流的主分支上避免费时的指令而造成的。

读/写信号量

读/写信号量类似于前面“读/写自旋锁”一节描述的读/写自旋锁，有一点不同：在信号量再次变为打开之前，等待进程挂起而不是自旋。

很多内核控制路径为读可以并发地获取读/写信号量。但是，任何写者内核控制路径必须有对被保护资源的互斥访问。因此，只有在没有内核控制路径为读访问或写访问持有信号量时，才可以为写获取信号量。读/写信号量可以提高内核中的并发度，并改善了整个系统的性能。

内核以严格的FIFO顺序处理等待读/写信号量的所有进程。如果读者或写者进程发现信号量关闭，这些进程就被插入到信号量等待队列链表的末尾。当信号量被释放时，就检查处于等待队列链表第一个位置的进程。第一个进程常被唤醒。如果是一个写者进程，等待队列上其他的进程就继续睡眠。如果是一个读者进程，那么紧跟第一个进程的其他所有读者进程也被唤醒并获得锁。不过，在写者进程之后排队的读者进程继续睡眠。

每个读/写信号量都是由rw_semaphore结构描述的，它包含下列字段：

count
: 存放两个16位的计数器。其中最高16位计数器以二进制补码形式存放非等待写者进程的总数（0或1）和等待的写内核控制路径数。最低16位计数器存放非等待的读者和写者进程的总数。

wait_list
: 指向等待进程的链表。链表中的每个元素都是一个rwsem_waiter结构，该结构包含一个指针和一个标志，指针指向睡眠进程的描述符，标志表示进程是为读需要信号量还是为写需要信号量。

wait_lock
: 一个自旋锁，用于保护等待队列链表和rw_semaphore结构本身。

init_rwsem()函数初始化rw_semaphore结构，即把count字段置为0，wait_lock自旋锁置为未锁，而把wait_list置为空链表。

down_read()和down_write()函数分别为读或写获取读/写信号量。同样，up_read()和up_write()函数为读或写释放以前获取的读/写信号量。down_read_trylock()和down_write_trylock()函数分别类似于down_read()和down_write()函数，但是，在信号量忙的情况下，它们不阻塞进程。最后，函数downgrade_write()自动把写锁转换成读锁。这5个函数的实现代码比较长，但因为它与普通信号量的实现类似，所以容易理解，我们就不再对它们进行说明。

补充原语

Linux 2.6还使用了另一种类似于信号量的原语：补充（*completion*）。引入这种原语是为了解决多处理器系统上发生的一种微妙的竞争条件，当进程A分配了一个临时信号量

变量，把它初始化为关闭的MUTEX，并把其地址传递给进程B，然后在A之上调用down()，进程A打算一但被唤醒就撤消该信号量。随后，运行在不同CPU上的进程B在同一信号量上调用up()。然而，up()和 down()的目前实现还允许这两个函数在同一个信号量上并发执行。因此，进程A可以被唤醒并撤销临时信号量，而进程B还在运行up()函数。结果，up()可能试图访问一个不存在的数据结构。

当然，也可以改变up()和down()的实现以禁止在同一信号量上并发执行。然而，这种改变可能需要另外的指令，这对于频繁使用的函数来说不是什么好事。

补充是专门设计来解决以上问题的同步原语。completion数据结构包含一个等待队列头和一个标志：

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

与up()对应的函数叫做complete()。complete()接收completion数据结构的地址作为参数，在补充等待队列的自旋锁上调用spin_lock_irqsave()，递增done字段，唤醒在wait等待队列上睡眠的互斥进程，最后调用spin_unlock_irqrestore()。

与down()对应的函数叫做wait_for_completion()。wait_for_completion()接收completion数据结构的地址作为参数，并检查done标志的值。如果该标志的值大于0，wait_for_completion()就终止，因为这说明complete()已经在另一个CPU上运行。否则，wait_for_completion()把current作为一个互斥进程加到等待队列的末尾，并把current置为TASK_UNINTERRUPTIBLE状态让其睡眠。一旦current被唤醒，该函数就把current从等待队列中删除，然后，函数检查done标志的值：如果等于0函数就结束，否则，再次挂起当前进程。与complete()函数中的情形一样，wait_for_completion()使用补充等待队列中的自旋锁。

补充原语和信号量之间的真正差别在于如何使用等待队列中包含的自旋锁。在补充原语中，自旋锁用来确保complete()和wait_for_completion()不会并发执行。在信号量中，自旋锁用于避免并发执行的down()函数弄乱信号量的数据结构。

禁止本地中断

确保一组内核语句被当作一个临界区处理的主要机制之一就是中断禁止。即使当硬件设备产生了一个IRQ信号时，中断禁止也让内核控制路径继续执行，因此，这就提供了一种有效的方式，确保中断处理程序访问的数据结构也受到保护。然而，禁止本地中断并不保护运行在另一个CPU上的中断处理程序对数据结构的并发访问，因此，在多处理器

系统上，禁止本地中断经常与自旋锁结合使用（参见后面“对内核数据结构的同步访问”一节）。

宏local_irq_disable()使用cli汇编语言指令关闭本地CPU上的中断，宏local_irq_enable()使用sti汇编语言指令打开被关闭的中断。正如在第四章“IRQ和中断”一节中说明的，汇编语言指令cli和sti分别清除和设置eflags控制寄存器的IF标志。如果eflags寄存器的IF标志被清0，宏irqs_disabled()产生等于1的值；如果IF标志被设置，该宏也产生为1的值。

当内核进入临界区时，通过把eflags寄存器的IF标志清0关闭中断。但是，内核经常不能在临界区的末尾简单地再次设置这个标志。中断可以以嵌套的方式执行，所以内核不必知道当前控制路径被执行之前IF标志的值究竟是什么。在这种情况下，控制路径必须保存先前赋给该标志的值，并在执行结束时恢复它。

保存和恢复eflags的内容是分别通过宏local_irq_save和local_irq_restore来实现的。local_irq_save宏把eflags寄存器的内容拷贝到一个局部变量中，随后用cli汇编语言指令把IF标志清0。在临界区的末尾，宏local_irq_restore恢复eflags原来的内容，因此，只是在这个控制路径发出cli汇编语言指令之前，中断被激活的情况下，中断才处于打开状态。

禁止和激活可延迟函数

在第四章的“软中断”一节，我们说明了可延迟函数可能在不可预知的时间执行（实际上是在硬件中断处理程序结束时）。因此，必须保护可延迟函数访问的数据结构使其避免竞争条件。

禁止可延迟函数在一个CPU上执行的一种简单方式就是禁止在那个CPU上的中断。因为没有中断处理程序被激活，因此，软中断操作就不能异步地开始。

然而，我们在下一节会看到，内核有时需要只禁止可延迟函数而不禁止中断。通过操纵当前thread_info描述符preempt_count字段中存放的软中断计数器，可以在本地CPU上激活或禁止可延迟函数。

回忆一下，如果软中断计数器是正数，do_softirq()函数就不会执行软中断，而且，因为tasklet在软中断之前被执行，把这个计数器设置为大于0的值，由此禁止了在给定CPU上的所有可延迟函数和软中断的执行。

宏local_bh_disable给本地CPU的软中断计数器加1，而函数local_bh_enable()从本地CPU的软中断计数器中减掉1。内核因此能使用几个嵌套的local_bh_disable调

用，只有宏local_bh_enable与第一个local_bh_disable调用相匹配，可延迟函数才再次被激活。

递减软中断计数器之后，local_bh_enable()执行两个重要的操作以有助于保证适时地执行长时间等待的线程：

1. 检查本地CPU的preempt_count字段中硬中断计数器和软中断计数器，如果这两个计数器的值都等于0而且有挂起的软中断要执行，就调用do_softirq()来激活这些软中断（见第四章“软中断”一节）。
2. 检查本地CPU的TIF_NEED_RESCHED标志是否被设置，如果是，说明进程切换请求是挂起的，因此调用preempt_schedule()函数（参见本章前面的“内核抢占”一节）。

对内核数据结构的同步访问

可以使用前面所述的同步原语保护共享数据结构避免竞争条件。当然，系统性能可能随所选择同步原语种类的不同而有很大变化。通常情况下，内核开发者采用下述由经验得到的法则：*把系统中的并发度保持在尽可能高的程度*。

系统中的并发度又取决于两个主要因素：

- 同时运转的I/O设备数
- 进行有效工作的CPU数

为了使I/O吞吐量最大化，应该使中断禁止保持在很短的时间。正如第四章的“IRQ和中断”一节描述的那样，当中断被禁止时，由I/O设备产生的IRQ被PIC暂时忽略，因此，就没有新的活动在这种设备上开始。

为了有效地利用CPU，应该尽可能避免使用基于自旋锁的同步原语。当一个CPU执行紧指令循环等待自旋锁打开时，是在浪费宝贵的机器周期。就像我们前面所描述的，更糟糕的是：由于自旋锁对硬件高速缓存的影响而使其对系统的整体性能产生不利影响。

让我们举例说明在下列两种情况下，既可以维持较高的并发度，也可以达到同步。

- 共享的数据结构是一个单独的整数值，可以把它声明为atomic_t类型并使用原子操作对其更新。原子操作比自旋锁和中断禁止都快，只有在几个内核控制路径同时访问这个数据结构时速度才会慢下来。
- 把一个元素插入到共享链表的操作决不是原子的，因为这至少涉及两个指针赋值。

不过，内核有时并不用锁或禁止中断就可以执行这种插入操作。我们把这种操作的工作机制作为例子来进行说明。考虑一种情况，系统调用服务例程（参见第十章的“系统调用处理程序及服务例程）把新元素插入到一个简单链表中，而中断处理程序或可延迟函数异步地查看该链表。

在C语言中，插入是通过下列指针赋值实现的：

```
new->next = list_element->next;
list_element->next = new;
```

在汇编语言中，插入简化为两个连续的原子指令。第一条指令建立new元素的next指针，但不修改链表。因此，如果中断处理程序在第一条指令和第二条指令执行的中间查看这个链表，看到的就是没有新元素的链表。如果该处理程序在第二条指令执行后查看链表，就会看到有新元素的链表。关键是，在任一种情况下，链表都是一致的且处于未损坏状态。然而，只有在中断处理程序不修改链表的情况下才能确保这种完整性。如果修改了链表，那么在new元素内刚刚设置的next指针就可能变为无效的。

然而，开发者必须确保两个赋值操作的顺序不被编译器或CPU控制单元搅乱；否则，如果中断处理程序在两个赋值之间中断了系统调用服务例程，处理程序就会看到一个损坏的链表。因此，就需要一个写内存屏障原语：

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

在自旋锁、信号量及中断禁止之间选择

遗憾的是，对大多数内核数据结构的访问模式非常复杂，远不像上例所示的那样简单，于是，迫使内核开发者使用信号量、自旋锁、中断禁止和软中断禁止。一般来说，同步原语的选取取决于访问数据结构的内核控制路径的种类，如表5-8所示。记住，只要内核控制路径获得自旋锁（还有读/写锁、顺序锁或RCU“读锁”），就禁用本地中断或本地软中断，自动禁用内核抢占。

表5-8：内核控制路径访问的数据结构所需要的保护

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常	信号量	无
中断	本地中断禁止	自旋锁
可延迟函数	无	无或自旋锁（参看表5-9）
异常与中断	本地中断禁止	自旋锁

表5-8：内核控制路径访问的数据结构所需要的保护（续）

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常与可延迟函数	本地软中断禁止	自旋锁
中断与可延迟函数	本地中断禁止	自旋锁
异常、中断与可延迟函数	本地中断禁止	自旋锁

保护异常所访问的数据结构

当一个数据结构仅由异常处理程序访问时，竞争条件通常是易于理解也易于避免的。最常见的产生同步问题的异常就是系统调用服务例程（参看第十章的“系统调用处理程序及服务例程”一节），在这种情况下，CPU运行在内核态而为用户态程序提供服务。因此，仅由异常访问的数据结构通常表示一种资源，可以分配给一个或多个进程。

竞争条件可以通过信号量避免，因为信号量原语允许进程睡眠到资源变为可用。注意，信号量工作方式在单处理器系统和多处理器系统上完全相同。

内核抢占不会引起太大的问题。如果一个拥有信号量的进程是可以被抢占的，运行在同一个CPU上的新进程就可能试图获得这个信号量。在这种情况下，让新进程处于睡眠状态，而且原来拥有信号量的进程最终会释放信号量。只有在访问每CPU变量的情况下，必须显式地禁用内核抢占，就像在本章前面“每CPU变量”一节中所描述的那样。

保护中断所访问的数据结构

假定一个数据结构仅被中断处理程序的“上半部分”访问。我们在第四章的“中断处理”一节了解到每个中断处理程序都相对自己串行地执行——也就是说，中断处理程序本身不能同时多次运行。因此，访问数据结构就无需任何同步原语。

但是，如果多个中断处理程序访问一个数据结构，情况就有所不同了。一个处理程序可以中断另一个处理程序，不同的中断处理程序可以在多处理器系统上同时运行。没有同步，共享的数据结构就很容易被破坏。

在单处理器系统上，必须通过在中断处理程序的所有临界区上禁止中断来避免竞争条件。只能用这种方式进行同步，因为其他的同步原语都不能完成这件事。信号量能够阻塞进程，因此，不能用在中断处理程序上。另一个方面，自旋锁可能使系统冻结：如果访问数据结构的处理程序被中断，它就不能释放锁；因此，新的中断处理程序在自旋锁的紧循环上保持等待。

同样，多处理器系统的要求甚至更加苛刻。不能简单地通过禁止本地中断来避免竞争条

不过，内核有时并不用锁或禁止中断就可以执行这种插入操作。我们把这种操作的工作机制作为例子来进行说明。考虑一种情况，系统调用服务例程（参见第十章的“系统调用处理程序及服务例程）把新元素插入到一个简单链表中，而中断处理程序或可延迟函数异步地查看该链表。

在C语言中，插入是通过下列指针赋值实现的：

```
new->next = list_element->next;
list_element->next = new;
```

在汇编语言中，插入简化为两个连续的原子指令。第一条指令建立new元素的next指针，但不修改链表。因此，如果中断处理程序在第一条指令和第二条指令执行的中间查看这个链表，看到的就是没有新元素的链表。如果该处理程序在第二条指令执行后查看链表，就会看到有新元素的链表。关键是，在任一种情况下，链表都是一致的且处于未损坏状态。然而，只有在中断处理程序不修改链表的情况下才能确保这种完整性。如果修改了链表，那么在new元素内刚刚设置的next指针就可能变为无效的。

然而，开发者必须确保两个赋值操作的顺序不被编译器或CPU控制单元搅乱；否则，如果中断处理程序在两个赋值之间中断了系统调用服务例程，处理程序就会看到一个损坏的链表。因此，就需要一个写内存屏障原语：

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

在自旋锁、信号量及中断禁止之间选择

遗憾的是，对大多数内核数据结构的访问模式非常复杂，远不像上例所示的那样简单，于是，迫使内核开发者使用信号量、自旋锁、中断禁止和软中断禁止。一般来说，同步原语的选取取决于访问数据结构的内核控制路径的种类，如表5-8所示。记住，只要内核控制路径获得自旋锁（还有读/写锁、顺序锁或RCU“读锁”），就禁用本地中断或本地软中断，自动禁用内核抢占。

表5-8：内核控制路径访问的数据结构所需要的保护

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常	信号量	无
中断	本地中断禁止	自旋锁
可延迟函数	无	无或自旋锁（参看表5-9）
异常与中断	本地中断禁止	自旋锁

表5-8：内核控制路径访问的数据结构所需要的保护（续）

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常与可延迟函数	本地软中断禁止	自旋锁
中断与可延迟函数	本地中断禁止	自旋锁
异常、中断与可延迟函数	本地中断禁止	自旋锁

保护异常所访问的数据结构

当一个数据结构仅由异常处理程序访问时，竞争条件通常是易于理解也易于避免的。最常见的产生同步问题的异常就是系统调用服务例程（参看第十章的“系统调用处理程序及服务例程”一节），在这种情况下，CPU运行在内核态而为用户态程序提供服务。因此，仅由异常访问的数据结构通常表示一种资源，可以分配给一个或多个进程。

竞争条件可以通过信号量避免，因为信号量原语允许进程睡眠到资源变为可用。注意，信号量工作方式在单处理器系统和多处理器系统上完全相同。

内核抢占不会引起太大的问题。如果一个拥有信号量的进程是可以被抢占的，运行在同一个CPU上的新进程就可能试图获得这个信号量。在这种情况下，让新进程处于睡眠状态，而且原来拥有信号量的进程最终会释放信号量。只有在访问每CPU变量的情况下，必须显式地禁用内核抢占，就像在本章前面“每CPU变量”一节中所描述的那样。

保护中断所访问的数据结构

假定一个数据结构仅被中断处理程序的“上半部分”访问。我们在第四章的“中断处理”一节了解到每个中断处理程序都相对自己串行地执行——也就是说，中断处理程序本身不能同时多次运行。因此，访问数据结构就无需任何同步原语。

但是，如果多个中断处理程序访问一个数据结构，情况就有所不同了。一个处理程序可以中断另一个处理程序，不同的中断处理程序可以在多处理器系统上同时运行。没有同步，共享的数据结构就很容易被破坏。

在单处理器系统上，必须通过在中断处理程序的所有临界区上禁止中断来避免竞争条件。只能用这种方式进行同步，因为其他的同步原语都不能完成这件事。信号量能够阻塞进程，因此，不能用在中断处理程序上。另一个方面，自旋锁可能使系统冻结：如果访问数据结构的处理程序被中断，它就不能释放锁；因此，新的中断处理程序在自旋锁的紧循环上保持等待。

同样，多处理器系统的要求甚至更加苛刻。不能简单地通过禁止本地中断来避免竞争条

件。事实上，即使在一个CPU上禁止了中断，中断处理程序还可以在其他CPU上执行。避免竞争条件最简单的方法是禁止本地中断(以便运行在同一个CPU上的其他中断处理程序不会造成干扰)，并获取保护数据结构的自旋锁或读/写自旋锁。注意，这些附加的自旋锁不能冻结系统，因为即使中断处理程序发现锁被关闭，在另一个CPU上拥有锁的中断处理程序最终也会释放这个锁。

Linux内核使用了几个宏，把本地中断激活/禁止与自旋锁结合起来。表5-9描述了其中的所有宏。在单处理器系统上，这些宏仅激活或禁止本地中断和内核抢占。

表5-9：与中断相关的自旋锁宏

宏	说明
spin_lock_irq(l)	local_irq_disable();spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l);local_irq_enable()
spin_lock_bh(l)	local_bh_disable();spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l);local_bh_enable()
spin_lock_irqsave(l,f)	local_irq_save(f);spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l);local_irq_restore(f)
read_lock_irq(l)	local_irq_disable();read_lock(l)
read_unlock_irq(l)	read_unlock(l);local_irq_enable()
read_lock_bh(l)	local_bh_disable();read_lock(l)
read_unlock_bh(l)	read_unlock(l);local_bh_enable()
write_lock_irq(l)	local_irq_disable();write_lock(l)
write_unlock_irq(l)	write_unlock(l);local_irq_enable()
write_lock_bh(l)	local_bh_disable();write_lock(l)
write_unlock_bh(l)	write_unlock(l);local_bh_enable()
read_lock_irqsave(l,f)	local_irq_save(f);read_lock(l)
read_unlock_irqrestore(l,f)	read_unlock(l);local_irq_restore(f)
write_lock_irqsave(l,f)	local_irq_save(f);write_lock(l)
write_unlock_irqrestore(l,f)	write_unlock(l);local_irq_restore(f)
read_seqbegin_irqsave(l,f)	local_irq_save(f);read_seqbegin(l)
read_seqretry_irqrestore(l,v,f)	read_seqretry(l,v);local_irq_restore(f)
write_seqlock_irqsave(l,f)	local_irq_save(f);write_seqlock(l)
write_sequnlock_irqrestore(l,f)	write_sequnlock(l);local_irq_restore(f)
write_seqlock_irq(l)	local_irq_disable();write_seqlock(l)
write_sequnlock_irq(l)	write_sequnlock(l);local_irq_enable()
write_seqlock_bh(l)	local_bh_disable();write_seqlock(l)
write_sequnlock_bh(l)	write_sequnlock(l);local_bh_enable()

保护可延迟函数所访问的数据结构

只被可延迟函数访问的数据结构需要哪种保护呢？这主要取决于可延迟函数的种类。在第四章“软中断及tasklet”一节，我们说明了软中断和ttasklet本质上有不同的并发度。

首先，在单处理器系统上不存在竞争条件。这是因为可延迟函数的执行总是在一个CPU上串行进行——也就是说，一个可延迟函数不会被另一个可延迟函数中断。因此，根本不需要同步原语。

相反，在多处理器系统上，竞争条件的确存在，因为几个可延迟函数可以并发运行。表5-10列出了所有可能的情况。

表5-10：在SMP上可延迟函数访问的数据结构所需的保护

访问数据结构的可延迟函数	保护
软中断	自旋锁
一个tasklet	无
多个tasklet	自旋锁

由软中断访问的数据结构必须受到保护，通常使用自旋锁进行保护，因为同一个软中断可以在两个或多个CPU上并发运行。相反，仅由一种tasklet访问的数据结构不需要保护，因为同种tasklet不能并发运行。但是，如果数据结构被几种tasklet访问，那么，就必须对数据结构进行保护。

保护由异常和中断访问的数据结构

让我们现在考虑一下由异常处理程序（例如系统调用服务例程）和中断处理程序访问的数据结构。

在单处理器系统上，竞争条件的防止是相当简单的，因为中断处理程序不是可重入的且不能被异常中断。只要内核以本地中断禁止访问数据结构，内核在访问数据结构的过程中就不会被中断。不过，如果数据结构正好是被一种中断处理程序访问，那么，中断处理程序不用禁止本地中断就可以自由地访问数据结构。

在多处理器系统上，我们必须关注异常和中断在其他CPU上的并发执行。本地中断禁止还必须外加自旋锁，强制并发的内核控制路径进行等待，直到访问数据结构的处理程序完成自己的工作。

有时，用信号量代替自旋锁可能更好。因为中断处理程序不能被挂起，它们必须用紧循环和`down_trylock()`函数获得信号量；对这些中断处理程序来说，信号量起的作用本

质上与自旋锁一样。另一方面，系统调用服务例程可以在信号量忙时挂起调用进程。对大部分系统调用而言，这是所期望的行为。在这种情况下，信号量比自旋锁更好，因为信号量使系统具有更高的并发度。

保护由异常和可延迟函数访问的数据结构

异常和可延迟函数都访问的数据结构与异常和中断处理程序访问的数据结构处理方式类似。事实上，可延迟函数本质上是由中断的出现激活的，而可延迟函数执行时不可能产生异常。因此，把本地中断禁止与自旋锁结合起来就足够了。

实际上，这更加充分：异常处理程序可以通过使用local_bh_disable()宏简单地禁止可延迟函数，而不禁止本地中断（参看第四章的“软中断”一节）。仅禁止可延迟函数比禁止中断更可取，因为中断还可以继续在CPU上得到服务。在每个CPU上可延迟函数的执行都被串行化，因此，不存在竞争条件。

同样，在多处理器系统上，要用自旋锁确保任何时候只有一个内核控制路径访问数据结构。

保护由中断和可延迟函数访问的数据结构

这种情况类似于中断和异常处理程序访问的数据结构。当可延迟函数运行时可能产生中断，但是，可延迟函数不能阻止中断处理程序。因此，必须通过在可延迟函数执行期间禁用本地中断来避免竞争条件。不过，中断处理程序可以随意访问被可延迟函数访问的数据结构而不用关中断，前提是没有其他的中断处理程序访问这个数据结构。

在多处理器系统上，还是需要自旋锁禁止对多个CPU上数据结构的并发访问。

保护由异常、中断和可延迟函数访问的数据结构

类似于前面的情况，禁止本地中断和获取自旋锁几乎总是避免竞争条件所必需的。注意，没有必要显式地禁止可延迟函数，因为当中断处理程序终止执行时，可延迟函数才能被实质激活；因此，禁止本地中断就足够了。

避免竞争条件的实例

人们总是期望内核开发者确定和解决由内核控制路径的交错执行所引起的同步问题。但是，避免竞争条件是一项艰巨的任务，因为这需要对内核的各个成分如何相互作用有一个清楚的理解。为了直观地认识内核内部到底是什么样子，需要提及本章所定义同步原语的几种典型用法。

引用计数器

引用计数器广泛地用在内核中以避免由于资源的并发分配和释放而产生的竞争条件。引用计数器（*reference counter*）只不过是一个atomic_t计数器，与特定的资源，如内存页、模块或文件相关。当内核控制路径开始使用资源时就原子地减少计数器的值，当内核控制路径使用完资源时就原子地增加计数器。当引用计数器变为0时，说明该资源未被使用，如果必要，就释放该资源。

大内核锁

在早期的Linux内核版本中，大内核锁（*big kernel block*，也叫全局内核锁或BKL）被广泛使用。在2.0版本中，这个锁是相对粗粒度的自旋锁，确保每次只有一个进程能运行在内核态。2.2和2.4内核具有极大的灵活性，不再依赖一个单独的自旋锁，而是由许多不同的自旋锁保护大量的内核数据结构。在Linux 2.6版本的内核中，用大内核锁来保护旧的代码（绝大多数是与VFS和几个文件系统相关的函数）。

从内核版本2.6.11开始，用一个叫做kernel_sem的信号量来实现大内核锁（在较早的2.6版本中，大内核锁是通过自旋锁来实现的）。但是，大内核锁比简单的信号量要复杂一些。

每个进程描述符都含有lock_depth字段，这个字段允许同一个进程几次获取大内核锁。因此，对大内核锁两次连续的请求不挂起处理器（相对于普通自旋锁）。如果进程未获得过锁，则这个字段的值为-1；否则，这个字段的值加1，表示已经请求了多少次锁。lock_depth字段对中断处理程序、异常处理程序及可延迟函数获取大内核锁都是至关重要的。如果没有这个字段，那么，在当前进程已经拥有大内核锁的情况下，任何试图获得这个锁的异步函数都可能产生死锁。

lock_kernel()和unlock_kernel()内核函数用来获得和释放大内核锁。前一个函数等价于：

```
depth = current->lock_depth + 1;
if (depth == 0)
    down(&kernel_sem);
current->lock_depth = depth;
```

而后者等价于：

```
if (--current->lock_depth < 0)
    up(&kernel_sem);
```

注意，lock_kernel()和unlock_kernel()函数的if语句不需要原子地执行，因为lock_depth不是全局变量——这是每个CPU在自己当前进程描述符中访问的一个字段。在

if语句内的本地中断也不会引起竞争条件。即使新内核控制路径调用了lock_kernel()，它在终止前也必须释放大内核锁。

足以令人吃惊的是，允许一个持有大内核锁的进程调用schedule()，从而放弃CPU！不过，schedule()函数检查被替换进程的lock_depth字段，如果它的值是0或者正数，就自动释放kernel_sem信号量（参见第七章“schedule()函数”一节）。因此，不会有显式调用schedule()的进程在进程切换前后都保持大内核锁。但是，当schedule()函数再次选择这个进程来执行的时候，将为该进程重新获得大内核锁。

然而，如果一个持有大内核锁的进程被另一个进程抢占，情况就有所不同了。一直到内核版本2.6.10还没有出现这种情况，因为获取自旋锁时会自动禁用内核抢占。但是，现在大内核锁的实现是基于信号量的，而且不会由于获得它而自动禁用内核抢占。实际上，在被大内核锁保护的临界区内允许内核抢占是改变大内核锁实现的主要原因。其次，这对于系统的响应时间会产生有益的影响。

当一个持有大内核锁的进程被抢占时，schedule()一定不能释放信号量，因为在临界区内执行代码的进程没有主动触发进程切换。所以，如果释放大内核锁，那么另外一个进程就可能获得它，并破坏由被抢占的进程所访问的数据结构。

为了避免被抢占的进程失去大内核锁，preempt_schedule_irq()临时把进程的lock_depth字段设置为-1（参见第四章“从中断和异常返回”）。观察这个字段的值，schedule()假定被替换的进程不拥有kernel_sem信号量，也就不释放它。结果，被抢占的进程就一直拥有kernel_sem信号量。一旦这个进程再次被调度程序选中，preempt_schedule_irq()函数就恢复lock_depth字段原来的值，并让进程在被大内核锁保护的临界区中继续执行。

内存描述符读/写信号量

mm_struct类型的每个内存描述符在mmap_sem字段中都包含了自己的信号量（参见第九章的“内存描述符”一节）。由于几个轻量级进程之间可以共享一个内存描述符，因此，信号量保护这个描述符以避免可能产生的竞争条件。

例如，让我们假设内核必须为某个进程创建或扩展一个内存区。为了做到这一点，内核调用do_mmap()函数分配一个新的vm_area_struct数据结构。在分配的过程中，如果没有可用的空闲内存，而共享同一内存描述符的另外一个进程可能在运行，那么当前进程可能被挂起。如果没有信号量，那么需要访问内存描述符的第二个进程的任何操作（例如，由于写时复制而产生的缺页）都可能会导致严重的数据崩溃。

这种信号量是作为读/写信号量来实现的，因为一些内核函数，如缺页异常处理程序（参见第九章的“缺页异常处理程序”一节）只需要扫描内存描述符。

slab 高速缓存链表的信号量

slab高速缓存描述符链表(参见第八章的"高速缓存描述符"一节)是通过cache_chain_sem信号量保护的，这个信号量允许互斥地访问和修改该链表。

当 kmem_cache_create()在链表中增加一个新元素，而 kmem_cache_shrink()和 kmem_cache_reap()顺序地扫描整个链表时，可能产生竞争条件。然而，在处理中断时，这些函数从不被调用，在访问链表时它们也从不阻塞。由于内核是支持抢占的，因此这种信号量在多处理器系统和单处理器系统中都会起作用。

索引节点的信号量

正如我们将在第十二章的"索引节点对象"一节中看到的，Linux 把磁盘文件的信息存放在一种叫做索引节点（inode）的内存对象中。相应的数据结构也包括有自己的信号量，存放在 i_sem 字段中。

在文件系统的处理过程中会出现很多竞争条件。实际上，磁盘上的每个文件都是所有用户共有的一种资源，因为所有进程都（可能）会存取文件的内容、修改文件名或文件位置、删除或复制文件等等。例如，让我们假设一个进程在显示某个目录所包含的文件。由于每个磁盘操作都可能会阻塞，因此即使在单处理器系统中，当第一个进程正在执行显示操作的过程中，其他进程也可能存取同一目录并修改它的内容。或者，两个不同的进程可能同时修改同一目录。所有这些竞争条件都可以通过用索引节点信号量保护目录文件来避免。

只要一个程序使用了两个或多个信号量，就存在死锁的可能，因为两个不同的控制路径可能互相死等着释放信号量。一般来说，Linux在信号量请求上很少会发生死锁问题，因为每个内核控制路径通常一次只需要获得一个信号量。然而，在有些情况下，内核必须获得两个或更多的信号量锁。索引节点信号量倾向于这种情况，例如，在 rename()系统调用的服务例程中就会发生这种情况。在这种情况下，操作涉及两个不同的索引节点，因此，必须采用两个信号量。为了避免这样的死锁，信号量的请求按预先确定的地址顺序进行。