第五章 并发和竞争情况

到目前为止，我们对并发性(concurrency)问题——即系统试图同时做多件事情时会发生什么——并没有给予太多关注。然而，管理并发性是操作系统编程的核心问题之一。并发相关的错误是最容易产生也是最难找到的错误。即使是经验丰富的Linux内核程序员有时也会产生并发相关的错误。

在早期的Linux内核中，并发的来源相对较少。对称多处理（Symmetric multiprocessing，SMP）系统并未得到内核的支持，唯一的并发执行原因是硬件中断的服务。这种方法简单明了，但在一个重视性能、处理器越来越多、并且要求系统快速响应事件的世界中，它已经无法满足需求。为了应对现代硬件和应用的需求，Linux内核已经发展到了许多事情可以同时进行的程度。这种进化大大提高了性能和可扩展性，但同时也显著增加了内核编程的复杂性。设备驱动程序员现在必须从一开始就将并发性考虑到他们的设计中，并且他们必须对内核提供的并发管理设施有深入的理解。

本章的目的是开始创建这种理解。为此，我们介绍了一些可以立即应用到第3章的scull驱动程序的设施。这里介绍的其他设施在一段时间内还没有被使用。但首先，我们来看看我们的简单scull驱动程序可能出现什么问题，以及如何避免这些潜在的问题。

5.1. scull 中的缺陷(Pitfalls)

让我们快速看一下scull内存管理代码的一部分。在写逻辑的深处，scull必须决定它需要的内存是否已经被分配。处理这个任务的代码片段之一是：

```c
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}
```

假设有一刻，两个进程（我们称它们为“A”和“B”）独立地试图写入同一scull设备内的同一偏移量。每个进程都在上面片段的第一行的if测试中同时到达。如果问题的指针是NULL，每个

进程都会决定分配内存，并将结果指针分配给 `dptr->data[s_pos]`。由于两个进程都在分配到同一位置，显然只有一个分配会生效。

当然，会发生的是，最后完成分配的进程会“赢”。如果进程A先分配，它的分配将被进程B覆盖。在那个时候，scull将完全忘记A分配的内存；它只有一个指向B的内存的指针。因此，A分配的内存将被丢弃，永远不会返回给系统。

这个事件序列是一个竞态条件的示例。竞态条件是由于对共享数据的无控制访问造成的。当发生错误的访问模式时，会产生意外的结果。对于这里讨论的竞态条件，结果是内存泄漏。这已经足够糟糕了，但竞态条件通常还可能导致系统崩溃、数据损坏或安全问题。程序员可能会忽视竞态条件，认为它们是极低概率的事件。但是，在计算世界中，百万分之一的事件可能每几秒就会发生，而且后果可能是严重的。

我们将很快从scull中消除竞态条件，但首先我们需要对并发性有更一般的了解。

5.2. 并发和它的管理

在现代的Linux系统中，有许多并发的来源，因此，也有可能的竞态条件。多个用户空间进程正在运行，它们可以以令人惊讶的组合方式访问你的代码。SMP系统可以在不同的处理器上同时执行你的代码。内核代码是可抢占的；你的驱动程序的代码可以在任何时候失去处理器，取而代之的进程也可能在你的驱动程序中运行。设备中断是可以导致你的代码并发执行的异步事件。内核还提供了各种延迟代码执行的机制，如工作队列、任务和定时器，这些机制可以使你的代码在与当前进程无关的方式下在任何时候运行。在现代的热插拔世界中，你的设备可能在你正在使用它的过程中突然消失。

避免竞态条件可能是一项艰巨的任务。在一个任何事情都可能在任何时候发生的世界中，驱动程序员如何避免创建绝对的混乱呢？事实证明，大多数竞态条件可以通过一些思考、内核的并发控制原语和应用一些基本原则来避免。我们先从原则开始，然后详细讨论如何应用它们。

竞态条件是由于对资源的共享访问产生的。当两个执行线程有理由处理同一数据结构（或硬件资源）时，总是存在混乱的可能性。所以，设计驱动程序时要记住的第一个经验法则是尽可能避免共享资源。如果没有并发访问，就不会有竞态条件。因此，精心编写的内核代码应该尽量少的共享。这个想法的最明显应用是避免使用全局变量。如果你把一个资源放在一个多个执行线程可以找到的地方，那么应该有一个强有力的理由这样做。

然而，事实是，这种共享往往是必需的。硬件资源本质上是共享的，软件资源也经常需要对多个线程可用。同时也要记住，全局变量远非共享数据的唯一方式；任何时候你的代码向内核的其他部分传递一个指针，它都可能创建一个新的共享情况。共享是生活的本质。

这是资源共享的硬性规则：任何时候，一个硬件或软件资源被超过一个执行线程共享，并且存在一个线程可能遇到该资源的不一致视图的可能性，你必须显式地管理对该资源的访问。

在上面的scull例子中，进程B的视图是不一致的；它不知道进程A已经为（共享的）设备分配了内存，它执行自己的分配并覆盖了A的工作。在这种情况下，我们必须控制对scull数据结构的访问。我们需要安排事情，使得代码要么看到已经分配的内存，要么知道没有其他人分配或将分配内存。访问管理的常用技术叫做锁定或互斥——确保任何时候只有一个执行线程可以操作一个共享资源。本章的其余部分将主要讨论锁定。

然而，首先，我们必须简要考虑另一个重要的规则。当内核代码创建一个将与内核的任何其他部分共享的对象时，该对象必须继续存在（并正常工作），直到知道没有外部引用存在。scull一旦使其设备可用，就必须准备好处理这些设备的请求。并且scull必须继续能够处理其设备的请求，直到它知道没有引用（如打开的用户空间文件）到这些设备存在。这个规则产生了两个要求：没有对象可以在它处于可以正常工作的状态之前提供给内核，对这样的对象的引用必须被跟踪。在大多数情况下，你会发现内核为你处理了引用计数，但总是有例外。

遵循上述规则需要规划和对细节的仔细关注。你可能会被你没有意识到的资源的并发访问所惊讶。然而，通过一些努力，大多数竞态条件可以在它们咬你——或你的用户之前被避免。

5.3. 信号量和互斥体(Semaphores and Mutexes)

让我们看看如何给scull添加锁。我们的目标是使我们对scull数据结构的操作变得原子性，这意味着就其他执行线程而言，整个操作一次性发生。对于我们的内存泄漏例子，我们需要确保如果一个线程发现一个特定的内存块必须被分配，它有机会在任何其他线程进行该测试之前执行该分配。为此，我们必须设置关键区段(critical sections)：任何给定时间只能由一个线程执行的代码。

并非所有的关键区段都是相同的，所以内核为不同的需求提供了不同的原语(pimitives)。在这种情况下，每次访问scull数据结构都是在进程上下文中作为直接用户请求的结果发生的；不会有从中断处理程序或其他异步上下文中进行的访问。没有特定的延迟（响应时间）要求；应用程序员理解I/O请求通常不会立即得到满足。此外，scull在访问自己的数据结构时并没有持有任何其他关键的系统资源。所有这些意味着，如果scull驱动程序在等待访问数据结构的机会时睡眠，没有人会介意。

“去睡眠”在这个上下文中是一个明确定义的术语。当一个Linux进程到达一个它不能再进一步前进的点时，它会去睡眠（或“阻塞”），让出处理器给其他人，直到将来某个时候它可以再次完成工作。进程经常在等待I/O完成时睡眠。当我们深入到内核时，我们会遇到许多我们不能睡眠的情况。然而，scull的write方法并不是这些情况之一。所以我们可以使用一个可能会导致进程在等待访问关键区段时睡眠的锁定机制。

同样重要的是，我们将执行一个可能会睡眠的操作（使用kmalloc进行内存分配）——所以无论如何，睡眠都是可能的。如果我们的关键区段要正常工作，我们必须使用一个在拥有锁的线程睡眠时工作的锁定原语。并非所有的锁定机制都可以在睡眠是可能的地方使用（我们将在本章后面看到一些不可以的）。然而，对于我们目前的需求，最适合的机制是信号量(semaphore)。

Linux 内核同步（五）：信号量（semaphore）_sema_init-CSDN博客 (https://blog.csdn.net/zhoutaopower/article/details/86614945)

信号量是计算机科学中一个被理解得很好的概念。在其核心，一个信号量是一个单一的整数值，结合了一对通常被称为P和V的函数。希望进入关键区段的进程会在相关的信号量上调用P；如果信号量的值大于零，那么该值会减少一个，并且进程继续。如果，相反，信号量的值是0（或更小），那么进程必须等待直到有人释放信号量。解锁一个信号量是通过调用V来完成的；这个函数增加信号量的值，并且如果需要的话，唤醒正在等待的进程。

当信号量用于互斥——防止多个进程同时在关键区段内运行——它们的值将被初始设置为1。这样的信号量在任何给定的时间只能被一个进程或线程持有。在这种模式下使用的信号量有时被称为互斥量，这当然是“互斥”（mutual exclusion）的缩写。在Linux内核中发现的几乎所有的信号量都用于互斥。

- 信号量的值代表了可以同时访问某个资源或进入某个代码区段的线程数。当我们将信号量初始化为1时，这意味着在任何给定的时间点，只有一个线程可以访问被该信号量保护的资源或代码区段。

- 当一个线程试图通过调用 `down` 或 `down_interruptible` 函数获取信号量时，如果信号量的值大于0，那么它的值会被减1，线程就可以继续执行。如果信号量的值为0，那么线程就会被阻塞，直到信号量的值大于0。

- 当线程完成对资源或代码区段的访问后，它会通过调用 `up` 函数来释放信号量，这会使信号量的值加1，并唤醒等待该信号量的任何线程。

- 因此，当我们将信号量初始化为1时，我们实际上是在设置一个互斥锁，确保在任何给定的时间点，只有一个线程可以访问被该信号量保护的资源或代码区段。这就是为什么初始化信号量为1可以表示互斥。

- 在Linux内核中，你可以使用信号量来实现互斥。以下是一个简单的示例，展示了如何在Linux内核中使用信号量：

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

struct semaphore sem;

void example_function(void)

{

    /* 初始化信号量为1，表示互斥 */

    sema_init(&sem, 1);

    /* 在关键区段前获取信号量 */

    down(&sem);

    /* 执行关键区段代码... */

    /* 在关键区段后释放信号量 */

    up(&sem);

}
```

- 在这个例子中，`sema_init`函数用于初始化信号量，参数1表示信号量的初始值。`down`函数用于获取信号量，如果信号量的值大于0，那么它会减1并立即返回。如果信号量的值为0，那么`down`函数会阻塞，直到信号量的值大于0。`up`函数用于释放信号量，它会将信号量的值加1，并唤醒等待该信号量的任何进程。

5.3.1. Linux 信号量实现

Linux内核提供了一个符合上述语义的信号量实现，尽管术语有些不同。要使用信号量，内核代码必须包含<asm/semaphore.h>。相关的类型是struct semaphore；实际的信号量可以通过几种方式声明和初始化。一种是直接创建一个信号量，然后用sema_init进行设置：

```
void sema_init(struct semaphore *sem, int val);
```

其中，val是要赋给信号量的初始值。

然而，通常，信号量是以互斥模式使用的。为了使这种常见情况更容易，内核提供了一组辅助函数和宏。因此，可以用以下其中一种方式声明并初始化一个互斥量：

```c
DECLARE_MUTEX(name);

DECLARE_MUTEX_LOCKED(name);
```

这里，结果是一个初始化为1（使用DECLARE_MUTEX）或0（使用DECLARE_MUTEX_LOCKED）的信号量变量（名为name）。在后一种情况下，互斥量开始时处于锁定状态；在任何线程被允许访问之前，必须显式解锁。

如果互斥量必须在运行时初始化（例如，如果它是动态分配的），则使用以下其中一个：

```c
void init_MUTEX(struct semaphore *sem);

void init_MUTEX_LOCKED(struct semaphore *sem);
```

在Linux世界中，P函数被称为down——或者是这个名字的某种变体。这里，“down”指的是该函数减少信号量的值，并且，可能在让调用者睡眠一段时间以等待信号量变得可用之后，授予对受保护资源的访问。down有三个版本：

```c
void down(struct semaphore *sem);

int down_interruptible(struct semaphore *sem);

int down_trylock(struct semaphore *sem);
```

down减少信号量的值并等待需要的时间。down_interruptible也是如此，但操作是可以中断的。可中断版本几乎总是你需要的；它允许在信号量上等待的用户空间进程被用户中断。一般规则是，除非真的没有其他选择，否则你不想使用不可中断的操作。不可中断的操作是创建无法杀死的进程（ps中看到的可怕的“D状态”）和烦扰你的用户的好方法。然而，使用down_interruptible需要一些额外的注意，如果操作被中断，函数返回一个非零值，调用者不持有信号量。正确使用down_interruptible需要始终检查返回值并相应地做出反应。

最后一个版本（down_trylock）永远不会睡眠；如果在调用时信号量不可用，down_trylock立即返回一个非零返回值。

一旦线程成功调用了down的某个版本，就被认为是“持有”信号量（或者说“取出”或“获取”了信号量）。那个线程现在有权访问由信号量保护的临界区。当需要互斥的操作完成后，必须返回信号量。Linux中等同于V的是up：

```c
void up(struct semaphore *sem);
```

一旦调用了up，调用者就不再持有信号量。

正如你所期望的，任何取出信号量的线程都必须通过一次（且只能一次）up调用来释放它。在错误路径中通常需要特别小心；如果在持有信号量时遇到错误，那么在将错误状态返回给调用者之前，必须释放该信号量。忘记释放信号量是一个容易犯的错误；结果（进程在看似无关的地方挂起）可能很难复现和追踪。

5.3.2. 在 scull 中使用信号量

信号量机制为scull提供了一个可以用来避免在访问scull_dev数据结构时出现竞态条件的工具。但是，我们需要正确地使用这个工具。正确使用锁定原语的关键是明确指定要保护的资源，并确保对这些资源的每次访问都使用正确的锁定。在我们的示例驱动程序中，所有感兴趣的内容都包含在scull_dev结构中，所以这是我们锁定机制的逻辑范围。让我们再看一下这个结构：

```c
struct scull_dev {
    struct scull_qset *data;  /* 指向第一个量子集的指针 */
    int quantum;              /* 当前的量子大小 */
    int qset;                 /* 当前的数组大小 */
    unsigned long size;       /* 这里存储的数据量 */
    unsigned int access_key;  /* 由sculluid和scullpriv使用 */
    struct semaphore sem;     /* 互斥信号量 */
    struct cdev cdev;      /* 字符设备结构 */
};
```

在结构的底部有一个名为sem的成员，这当然是我们的信号量。我们选择为每个虚拟scull设备使用一个单独的信号量。使用一个单一的全局信号量也是同样正确的。然而，各种scull设备没有共享的资源，因此没有理由让一个进程等待，而另一个进程正在处理一个不同的scull设备。为每个设备使用一个单独的信号量允许对不同设备的操作并行进行，因此，提高了性能。

在使用之前，必须初始化信号量。scull在加载时在这个循环中执行这个初始化：

```c
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

注意，必须在scull设备对系统的其余部分可用之前初始化信号量。因此，init_MUTEX在scull_setup_cdev之前被调用。如果以相反的顺序执行这些操作，将会创建一个竞态条件，其中信号量可能在准备就绪之前被访问。

接下来，我们必须通过代码，确保没有在不持有信号量的情况下访问scull_dev数据结构。因此，例如，scull_write以这段代码开始：

```c
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
```

注意对down_interruptible返回值的检查；如果它返回非零，操作被中断。在这种情况下，通常的做法是返回-ERESTARTSYS。看到这个返回码后，内核的高层将从头开始重启调用，或者将错误返回给用户。如果你返回-ERESTARTSYS，你必须首先撤销可能已经做出的任何对用户可见的更改，以便在系统调用重试时能够正确执行。如果你不能以这种方式撤销事情，你应该返回-EINTR。

无论scull_write是否能够成功执行其其他任务，都必须释放信号量。如果一切顺利，执行会进入函数的最后几行：

```c
out:
  up(&dev->sem);
  return retval;
```

这段代码释放了信号量，并返回所需的状态。在scull_write中有几个地方可能会出错；这些包括内存分配失败或者在试图从用户空间复制数据时出现故障。在这些情况下，代码执行goto out，确保进行了适当的清理。

- `down_interruptible`和`down`都是用于获取信号量的函数，但它们在处理中断时的行为不同。
- `down`函数会阻塞调用线程，直到信号量可用。如果在等待期间收到中断，`down`函数将忽略它并继续等待。
- 相反，`down_interruptible`函数在等待信号量时，如果收到中断，会立即返回。这对于需要响应信号（如用户按下Ctrl+C）的程序来说是有用的。如果在等待期间收到中断，`down_interruptible`会返回一个非零值（通常是-ERESTARTSYS），这可以用来检查是否发生了中断。
- 总的来说，`down_interruptible`提供了一种机制，允许线程在等待信号量时响应中断，而`down`则会忽略任何中断，直到信号量可用。

5.3.3. 读写信号量

信号量为所有调用者执行互斥，无论每个线程可能想做什么。然而，许多任务可以分解为两种不同类型的工作：只需要读取受保护的数据结构的任务，和必须进行更改的任务。只要没有人试图做任何改变，通常可以允许多个并发读者。这样做可以显著优化性能；只读任务可以并行完成他们的工作，而不必等待其他读者退出临界区。

Linux内核为这种情况提供了一种特殊类型的信号量，称为rwsem（或"读/写信号量"）。在驱动程序中使用rwsems相对较少，但它们偶尔是有用的。

使用rwsems的代码必须包含<linux/rwsem.h>。读/写信号量的相关数据类型是struct rw_semaphore；rwsem必须在运行时明确初始化：

```c
void init_rwsem(struct rw_semaphore *sem);
```

新初始化的rwsem可供下一个来的任务（读者或写者）使用。需要只读访问的代码的接口是：

```c
void down_read(struct rw_semaphore *sem);

int down_read_trylock(struct rw_semaphore *sem);

void up_read(struct rw_semaphore *sem);
```

调用down_read提供对受保护资源的只读访问，可能与其他读者并发。注意，down_read可能会使调用进程进入不可中断的睡眠。如果读访问不可用，down_read_trylock不会等待；如果访问被授予，它返回非零，否则返回0。注意，down_read_trylock的约定与大多数内核函数的约定不同，其中成功是由返回值0表示的。用down_read获取的rwsem最终必须用up_read释放。

down_write、down_write_trylock和up_write的行为都与他们的读者对应物一样，当然，他们提供的是写访问。如果你有一个情况，需要一个写锁进行快速更改，然后是一个更长的只读访问期，你可以使用downgrade_write在你完成更改后允许其他读者进入。

```c
void down_write(struct rw_semaphore *sem);

int down_write_trylock(struct rw_semaphore *sem);

void up_write(struct rw_semaphore *sem);

void downgrade_write(struct rw_semaphore *sem);
```

down_write提供对受保护资源的写访问，可能会阻止其他读者和写者。down_write_trylock如果写访问不可用，它不会等待；如果访问被授予，它返回非零，否则返回0。用down_write获取的rwsem最终必须用up_write释放。downgrade_write将写锁降级为读锁，允许其他读者并发访问。

5.4. Completions 机制

在内核编程中，一种常见的模式是在当前线程之外启动一些活动，然后等待这些活动完成。这些活动可以是创建新的内核线程或用户空间进程，向现有进程发出请求，或者进行某种基于硬件的操作。在这种情况下，你可能会想到使用信号量来同步这两个任务，代码可能如下：

```c
struct semaphore sem;

init_MUTEX_LOCKED(&sem);

start_external_task(&sem);

down(&sem);
```

外部任务在完成工作后可以调用 `up(&sem)`。

然而，事实证明，信号量并不是这种情况下的最佳工具。在正常使用中，试图锁定信号量的代码几乎总是能找到可用的信号量；如果信号量的争用很大，性能就会受到影响，需要重新审视锁定方案。因此，信号量已经针对“可用”情况进行了大量优化。但是，当用于上述方式通知任务完成时，调用 down 的线程几乎总是需要等待；性能将相应地受到影响。如果将信号量声明为自动变量，那么在这种使用方式下，信号量也可能会遇到（困难的）竞态条件。在某些情况下，信号量可能在调用 up 的进程完成使用之前就消失了。

- 首先，信号量是一种用于多线程编程中的同步机制，它用于解决资源的争用问题。当一个线程试图锁定（down）一个信号量时，如果信号量可用，那么这个线程就可以继续执行；如果信号量不可用（即已经被其他线程锁定），那么这个线程就会被阻塞，直到信号量变得可用。

- 然而，这段话指出，信号量并不是在所有情况下都是最佳的同步工具。在大多数正常使用中，试图锁定信号量的代码几乎总是能找到可用的信号量，因此信号量的设计和优化主要是针对这种“可用”的情况。但是，如果用信号量来通知任务完成，那么调用 down 的线程几乎总是需要等待，因为在任务完成之前，信号量是不可用的。这就导致了性能问题，因为线程需要花费时间等待，而不是执行有用的工作。

- 此外，如果将信号量声明为自动变量（即在函数内部声明的变量），那么可能会遇到竞态条件。竞态条件是指在多线程环境中，由于线程的调度和执行顺序不确定，导致程序的行为也不确定。在这种情况下，信号量可能在调用 up 的进程完成使用之前就消失

了，因为自动变量在函数返回时会被销毁。这就导致了信号量的使用者可能会尝试访问一个已经不存在的信号量，从而导致错误。

这些问题促使人们在2.4.7内核中添加了“完成completion”接口。完成是一种轻量级机制，只有一个任务：让一个线程告诉另一个线程工作已经完成。要使用完成，你的代码必须包含 `<linux/completion.h>`。可以用以下方式创建一个完成：

```c
DECLARE_COMPLETION(my_completion);
```

或者，如果完成必须动态创建和初始化：

```c
struct completion my_completion;

/* ... */

init_completion(&my_completion);
```

等待完成只是简单地调用：

```c
void wait_for_completion(struct completion *c);
```

注意，这个函数执行的是不可中断的等待。如果你的代码调用 `wait_for_completion`，但没有人完成任务，结果将是一个无法杀死的进程。

另一方面，实际的完成事件可以通过调用以下其中一个来发出信号：

```c
void complete(struct completion *c);

void complete_all(struct completion *c);
```

如果有多个线程等待同一个完成事件，这两个函数的行为是不同的。 `complete` 只唤醒一个等待的线程，而 `complete_all` 允许所有线程继续。在大多数情况下，只有一个等待者，这两个函数将产生相同的结果。

完成通常是一次性的设备；使用一次然后丢弃。然而，如果采取适当的措施，是可以重用完成结构的。如果没有使用 `complete_all`，只要没有关于正在发出的事件的歧义，就可以无问题地重用完成结构。但是，如果你使用了 `complete_all`，你必须在重用它之前重新初始化完成结构。可以使用以下宏来快速执行这个重新初始化：

```c
INIT_COMPLETION(struct completion c);
```

作为完成如何使用的一个例子，考虑一下包含在示例源代码中的完整模块。这个模块定义了一个具有简单语义的设备：任何试图从设备读取的进程都会等待（使用 `wait_for_completion`），直到其他进程写入设备。实现这种行为的代码是：

```c
DECLARE_COMPLETION(comp);

ssize_t complete_read (struct file *filp, char __user *buf,
size_t count, loff_t *pos)

{

    printk(KERN_DEBUG "process %i (%s) going to sleep\n",

            current->pid, current->comm);

    wait_for_completion(&comp);

    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid,
current->comm);

    return 0; /* EOF */

}

ssize_t complete_write (struct file *filp, const char __user
*buf, size_t count,

        loff_t *pos)

{

    printk(KERN_DEBUG "process %i (%s) awakening the
readers...\n",

            current->pid, current->comm);

    complete(&comp);

    return count; /* succeed, to avoid retrial */

}
```

可以同时有多个进程“从这个设备读取”。每次写入设备都会导致恰好一个读操作完成，但是无法知道会是哪一个。

完成机制的典型用途是在模块退出时的内核线程终止。在原型案例中，驱动程序的一些内部工作是由一个在 `while (1)` 循环中的内核线程执行的。当模块准备清理时，退出函数告诉线程退出，然后等待完成。为此，内核包含了一个特定的函数供线程使用：

```c
void complete_and_exit(struct completion *c, long retval);
```

5.5. 自旋锁(Spinlocks)

信号量是一种实现互斥的有用工具，但它并不是内核提供的唯一工具。相反，大多数锁定是通过一种叫做自旋锁的机制实现的。与信号量不同，自旋锁可以在不能休眠的代码中使用，比如中断处理程序。当正确使用时，自旋锁通常比信号量提供更高的性能。然而，它们对使用带来了一组不同的约束。

自旋锁的概念很简单。自旋锁是一种只有两个值的互斥设备：“锁定”和“未锁定”。它通常实现为整数值中的一个单独的位。希望获取特定锁的代码会测试相关的位。如果锁可用，就设置“锁定”位，然后代码继续进入关键区域。如果锁被其他人占用，代码就会进入一个紧密的循环，反复检查锁，直到它变得可用。这个循环就是自旋锁的“自旋”部分。

当然，自旋锁的真实实现比上述描述要复杂一些。“测试和设置test and set”操作必须以原子方式进行，以便只有一个线程可以获取锁，即使在任何给定时间都有几个线程在自旋。还必须小心避免在超线程处理器上发生死锁——这些芯片实现了多个虚拟CPU，共享一个处理器核心和缓存。因此，实际的自旋锁实现对于Linux支持的每一种架构都是不同的。然而，核心概念在所有系统上都是相同的，当有自旋锁的争用时，等待的处理器执行一个紧密的循环，不做任何有用的工作。

自旋锁本质上是为多处理器系统设计的，尽管运行抢占式内核的单处理器工作站在并发性方面表现得像SMP。如果非抢占式单处理器系统曾经在锁上自旋，它会永远自旋；没有其他线程能够获取CPU来释放锁。因此，没有启用抢占的单处理器系统上的自旋锁操作被优化为什么都不做，除了改变IRQ屏蔽状态的那些操作。由于抢占，即使你从不期望你的代码在SMP系统上运行，你仍然需要实现适当的锁定。

5.5.1. 自旋锁 API 简介

使用自旋锁原语所需的包含文件是 <linux/spinlock.h>。实际的锁的类型是 spinlock_t。像任何其他数据结构一样，自旋锁必须被初始化。这个初始化可以在编译时如下进行：

```c
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者在运行时使用：

```c
void spin_lock_init(spinlock_t *lock);
```

在进入关键区域之前，你的代码必须用以下方式获取必要的锁：

```c
void spin_lock(spinlock_t *lock);
```

注意，所有的自旋锁等待都是不可中断的。一旦你调用 spin_lock，你将一直自旋，直到锁变得可用。

要释放你已经获取的锁，将它传递给：

```c
void spin_unlock(spinlock_t *lock);
```

还有许多其他的自旋锁函数，我们将很快看到它们。但是，它们都没有偏离上面列出的函数所展示的核心思想。除了锁定和释放它，你可以对锁做的事情非常少。然而，有一些关于你必须如何使用自旋锁的规则。在深入了解完整的自旋锁接口之前，我们将花一点时间来看看这些规则。

5.5.2. 自旋锁和原子上下文(Spinlocks and Atomic Context)

假设你的驱动程序获取了一个自旋锁，并在其关键区域内进行操作。在中间的某个地方，你的驱动程序失去了处理器。也许它调用了一个函数（比如 copy_from_user）使进程进入睡眠。或者，可能是内核抢占发生，一个更高优先级的进程将你的代码推到一边。你的代码现在正在持有一个它在可预见的未来都不会释放的锁。如果其他线程试图获取同一个锁，最好的情况是，它会在处理器中等待（自旋）很长时间。最糟糕的情况是，系统可能完全死锁。

大多数读者都会同意，最好避免这种情况。因此，适用于自旋锁的核心规则是，任何代码在持有自旋锁时，必须是原子的。它不能睡眠；实际上，除了服务中断（有时甚至不能这样

做），它不能出于任何原因放弃处理器。

内核抢占的情况由自旋锁代码本身处理。任何时候内核代码持有一个自旋锁，相关处理器上的抢占就被禁用。即使是单处理器系统也必须以这种方式禁用抢占，以避免竞争条件。这就是为什么即使你从不期望你的代码在多处理器机器上运行，你仍然需要实现适当的锁定。

避免在持有锁的情况下睡眠可能更困难；许多内核函数可以睡眠，而这种行为并不总是有良好的文档记录。复制数据到用户空间或从用户空间复制数据是一个明显的例子：在复制可以进行之前，可能需要从磁盘中换入所需的用户空间页面，而这个操作显然需要睡眠。几乎任何必须分配内存的操作都可以睡眠；除非明确告知不要这样做，否则 kmalloc 可以决定放弃处理器，并等待更多的内存变得可用。睡眠可能会在令人惊讶的地方发生；编写在自旋锁下执行的代码需要注意你调用的每一个函数。

- 在操作系统中，"睡眠"是指进程因为等待某个条件（如等待I/O操作完成，等待某个资源变得可用，等待某个锁被释放等）而暂停执行的状态。当进程处于睡眠状态时，它不会消耗CPU资源。
- 当进程的等待条件得到满足时（如I/O操作完成，资源变得可用，锁被释放等），操作系统会将进程从睡眠状态唤醒，使其恢复执行。
- 在多任务操作系统中，进程的睡眠和唤醒机制是非常重要的，它使得系统可以在多个进程之间共享有限的系统资源，如CPU时间，内存，磁盘等。
- 当你的代码持有一个自旋锁并在关键区域内执行时，你需要确保这段代码不会引起任何可能导致进程睡眠的操作。因为如果进程在持有自旋锁的情况下进入睡眠，那么其他试图获取该自旋锁的进程将会被阻塞，这可能导致系统性能下降甚至死锁。
- 这里提到的一些可能导致进程睡眠的操作包括：

1. 复制数据到用户空间或从用户空间复制数据：这可能需要从磁盘中换入所需的用户空间页面，而这个操作可能需要进程睡眠等待。
2. 分配内存：例如，kmalloc函数在内存不足时可能会选择让进程睡眠，等待更多的内存变得可用

这是另一个场景：你的驱动程序正在执行，并刚刚取出一个控制对其设备访问的锁。在持有锁的情况下，设备发出一个中断，导致你的中断处理程序运行。在访问设备之前，中断处理程序也必须获取锁。在中断处理程序中取出一个自旋锁是合法的；这就是自旋锁操作不会睡眠的原因之一。但是，如果中断例程在与最初取出锁的代码相同的处理器中执行会发生什么？当中断处理程序在自旋时，非中断代码将无法运行以释放锁。那个处理器将永远自旋。

- 在这个场景中，你的驱动程序已经获取了一个锁，然后设备发出一个中断。这个中断触发了一个中断处理程序，这个程序也需要获取同一个锁才能访问设备。

- 如果这个中断处理程序在同一个处理器上运行，并试图获取已经被驱动程序持有的锁，那么就会发生死锁。因为中断处理程序会一直等待（自旋）直到锁被释放，但是因为中断处理程序正在运行，所以驱动程序无法继续执行以释放锁。这就导致了处理器陷入了一个无法退出的自旋状态，也就是死锁。

- 这就是为什么在持有自旋锁的情况下，需要禁用中断，以防止这种死锁的发生。

- 自旋锁和睡眠是操作系统中两种不同的概念，它们在处理资源竞争和进程调度时有不同的作用。

1. 自旋锁（Spinlock）：自旋锁是一种同步机制，用于保护临界区，防止多个进程同时访问共享资源。当一个进程尝试获取已经被其他进程持有的自旋锁时，它会进入一个忙等（busy-wait）或称为"自旋"的状态，不断地检查锁是否已经被释放，而不是立即释放CPU。
2. 睡眠（Sleep）：睡眠是进程在等待某个条件（如等待I/O操作完成，等待某个资源变得可用，等待某个锁被释放等）而进入的状态。当进程处于睡眠状态时，它会释放CPU，让出CPU给其他进程使用。当等待的条件得到满足时，进程会被唤醒，恢复执行。

- 主要的区别在于，自旋锁在等待锁释放时会持续占用CPU（即使它什么也没做），而进程在睡眠状态时会释放CPU。因此，自旋锁适用于等待时间非常短的情况，而当预计等待时间较长时，应该使用可以让进程进入睡眠的锁机制（如信号量或互斥体）。

避免这个陷阱需要在持有自旋锁的情况下禁用中断（只在本地 CPU 上）。有一些自旋锁函数的变体会为你禁用中断（我们将在下一节中看到它们）。然而，对中断的完整讨论必须等到第10章。

使用自旋锁的最后一个重要规则是，自旋锁必须始终保持最短的时间。你持有锁的时间越长，其他处理器可能需要自旋等待你释放它的时间就越长，而且它必须自旋的机会也更大。长时间持有锁也会阻止当前处理器进行调度，这意味着一个更高优先级的进程——它真的应该能够得到 CPU——可能必须等待。内核开发者在 2.5 开发系列中投入了大量的努力来减少内核延迟（一个进程可能需要等待被调度的时间）。一个编写得不好的驱动程序只需持有一个锁过长时间就可以抹去所有这些进展。为了避免造成这种问题，务必保持你的锁持有时间短。

5.5.3. 自旋锁函数

我们已经看过两个操作自旋锁的函数，spin_lock 和 spin_unlock。然而，还有一些其他的函数，它们的名字和目的类似。我们现在将介绍全部的函数。这个讨论会引导我们进入一些我们还不能完全覆盖的领域；对自旋锁API的完全理解需要理解中断处理和相关概念。

实际上，有四个函数可以锁定一个自旋锁：

```c
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long
flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
```

我们已经看过 `spin_lock` 是如何工作的。 `spin_lock_irqsave` 在获取自旋锁之前禁用中断（只在本地处理器上），之前的中断状态存储在 flags 中。如果你完全确定没有其他东西可能已经在你的处理器上禁用了中断（换句话说，你确定当你释放你的自旋锁时，你应该启用中断），你可以使用 `spin_lock_irq`，而不必跟踪 flags。最后， `spin_lock_bh` 在获取锁之前禁用软件中断，但保持硬件中断启用。

如果你有一个自旋锁，可以被运行在（硬件或软件）中断上下文的代码获取，你必须使用一种禁用中断的 spin_lock 形式。否则，可能会导致系统死锁，迟早会发生。如果你不在硬件中断处理器中访问你的锁，但你通过软件中断做到了这一点（例如，在一个任务中运行的代码，这是第7章的一个主题），你可以使用 spin_lock_bh 来安全地避免死锁，同时仍然允许服务硬件中断。

释放自旋锁也有四种方式；你使用的方式必须与你用来获取锁的函数相对应：

```c
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long
flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

每个 spin_unlock 变体都会撤销相应的 spin_lock 函数所做的工作。传递给 spin_unlock_irqrestore 的 flags 参数必须是传递给 spin_lock_irqsave 的同一变量。你也必须在同一函数中调用 spin_lock_irqsave 和 spin_unlock_irqrestore；否则，你的代码可能在某些架构上出问题。

还有一组非阻塞的自旋锁操作：

```c
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

这些函数在成功时返回非零（获取了锁），否则返回0。 没有禁用中断的“尝试”版本。

5.5.4. 读/写自旋锁

内核提供了一种读写形式的自旋锁，这与我们在本章前面看到的读写信号量非常相似。这种锁允许任意数量的读者同时进入临界区，但是写者必须独占访问。读写锁的类型为rwlock_t，定义在<linux/spinlock.h>中。它们可以通过两种方式声明和初始化：

```c
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态方式 */

rwlock_t my_rwlock;

rwlock_init(&my_rwlock);  /* 动态方式 */
```

现在，我们应该对可用的函数列表有所了解。对于读者，有以下函数可用：

```c
void read_lock(rwlock_t *lock);

void read_lock_irqsave(rwlock_t *lock, unsigned long flags);

void read_lock_irq(rwlock_t *lock);

void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);

void read_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);

void read_unlock_irq(rwlock_t *lock);

void read_unlock_bh(rwlock_t *lock);
```

有趣的是，没有read_trylock函数。

对于写访问，函数类似：

```c
void write_lock(rwlock_t *lock);

void write_lock_irqsave(rwlock_t *lock, unsigned long
flags);

void write_lock_irq(rwlock_t *lock);

void write_lock_bh(rwlock_t *lock);

int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);

void write_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);

void write_unlock_irq(rwlock_t *lock);

void write_unlock_bh(rwlock_t *lock);
```

读写锁可以像rwsems一样使读者饥饿。这种行为很少成为问题；然而，如果锁竞争足够激烈以至于导致饥饿，那么性能本来就很差。

- 读写锁（RWLock）与互斥锁（Mutex）相比，主要有以下优势：

1. **并发性**：读写锁允许多个读者同时访问资源，这在读操作远多于写操作的情况下可以提高系统的并发性能。而互斥锁在同一时间只允许一个线程访问资源。
2. **灵活性**：读写锁提供了更细粒度的控制，可以根据需要选择读锁还是写锁。而互斥锁只提供了一种类型的锁。

- 然而，使用读写锁也有一些需要注意的地方：

1. **复杂性**：读写锁的使用和管理通常比互斥锁更复杂。
2. **写饥饿**：如果读操作非常频繁，可能会导致写操作长时间得不到执行，这被称为写饥饿。
3. **性能开销**：读写锁的性能开销通常比互斥锁要大，特别是在高竞争的情况下。

- 因此，是否选择使用读写锁还是互斥锁，需要根据具体的应用场景和需求来决定。

5.6. 锁陷阱(Locking Traps)

多年的锁定经验（这些经验早于Linux）已经表明，锁定可能很难做对。管理并发性本质上是一项棘手的任务，有很多可能出错的地方。在这一部分，我们快速看一下可能出错的地方。

5.6.1. 模糊的规则(Ambiguous)

如上所述，一个合适的锁定方案需要清晰和明确的规则。当你创建一个可以并发访问的资源时，你应该定义哪个锁将控制那个访问。锁定应该在开始时就明确规划好；在后期进行改造可能会很困难。在开始时花费的时间通常在调试时会得到丰厚的回报。

在编写代码时，你无疑会遇到几个函数，它们都需要访问由特定锁保护的结构。在这一点上，你必须小心：如果一个函数获取了一个锁，然后调用另一个也试图获取锁的函数，你的代码就会死锁。无论是信号量还是自旋锁，都不允许锁的持有者第二次获取锁；如果你试图这样做，事情就会停滞不前。

为了使你的锁定工作正常，你必须写一些函数，假设它们的调用者已经获取了相关的锁。通常，只有你的内部，静态函数可以这样写；从外部调用的函数必须明确处理锁定。当你编写假设锁定的内部函数时，为自己（和任何其他使用你的代码的人）做个好事，明确记录这些假设。几个月后回来，想弄清楚是否需要持有一个锁来调用特定的函数可能会很困难。

在scull的情况下，设计决策是要求所有直接从系统调用调用的函数获取应用于被访问的设备结构的信号量。所有内部函数，只从其他scull函数调用，然后可以假设信号量已经被正确获取。

5.6.2. 加锁顺序规则

在拥有大量锁的系统中（内核正在变成这样的系统），代码需要同时持有多个锁的情况并不少见。如果需要使用两个不同的资源进行某种计算，每个资源都有自己的锁，那么通常没有其他选择，只能获取两个锁。

然而，获取多个锁可能是危险的。如果你有两个锁，我们称之为Lock1和Lock2，代码需要同时获取这两个锁，你就有可能出现死锁。只需想象一下，一个线程锁定Lock1，而另一个线程同时获取Lock2。然后每个线程都试图获取它没有的那个。两个线程都会死锁。

解决这个问题的方法通常很简单：当需要获取多个锁时，应始终按相同的顺序获取。只要遵循这个约定，就可以避免上述描述的简单死锁。然而，遵循锁定顺序规则可能比说起来容易。这样的规则实际上很少写在任何地方。通常你能做的最好的就是看看其他代码是怎么做的。

有几个经验法则可以帮助。如果你必须获取一个属于你的代码的本地锁（比如设备锁）以及一个属于内核更中心部分的锁，那么首先获取你的锁。如果你有信号量和自旋锁的组合，你

当然必须先获取信号量；在持有自旋锁的情况下调用down（可能会睡眠）是一个严重的错误。但最重要的是，尽量避免需要多个锁的情况。

5.6.3. 细 -粗- 粒度加锁

首个支持多处理器系统的Linux内核是2.0版本，它只包含一个自旋锁。这个大内核锁将整个内核变成了一个大的临界区；任何给定时间内，只有一个CPU能执行内核代码。这个锁足够好地解决了并发问题，使得内核开发者能够解决支持SMP所涉及的所有其他问题。但是，它的扩展性并不好。即使是一个双处理器系统，也可能花费大量的时间只是等待大内核锁。一个四处理器系统的性能甚至无法接近四台独立机器的性能。

因此，后续的内核版本引入了更细粒度的锁定。在2.2版本中，一个自旋锁控制了对块I/O子系统的访问；另一个用于网络等。一个现代的内核可以包含数千个锁，每个锁保护一个小资源。这种细粒度的锁定对于可扩展性是有好处的；它允许每个处理器在不与其他处理器争用锁的情况下，处理其特定的任务。很少有人怀念大内核锁。

然而，细粒度的锁定是有代价的。在一个包含数千个锁的内核中，要知道你需要哪些锁——以及你应该以什么顺序获取它们——以执行特定的操作，可能会非常困难。记住，锁定错误可能非常难以发现；更多的锁提供了更多的机会，让真正恶劣的锁定错误悄悄溜进内核。细粒度的锁定可能会带来一种复杂性，从长期来看，可能对内核的可维护性产生大的不利影响。

在设备驱动程序中的锁定通常相对直接；你可以有一个覆盖你做的所有事情的锁，或者你可以为你管理的每个设备创建一个锁。作为一般规则，除非你有真正的理由相信争用可能是一个问题，否则你应该从相对粗糙的锁定开始。抵制过早优化的冲动；真正的性能限制通常会出现在意想不到的地方。

如果你怀疑锁争用正在影响性能，你可能会发现lock-meter工具很有用。这个补丁（可在 http://oss.sgi.com/projects/lockmeter/ (vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 获取）对内核进行了仪器化，以测量在锁中等待的时间。通过查看报告，你能够快速确定锁争用是否真的是问题。

5.7. 加锁的各种选择

Linux内核提供了一些强大的锁定原语，可以用来防止内核自己跌倒。但是，正如我们所看到的，锁定方案的设计和实现并非没有陷阱。通常没有其他选择，只能使用信号量和自旋锁；它们可能是正确完成任务的唯一方式。然而，也有一些情况，可以在不需要完全锁定的情况下设置原子访问。本节将探讨其他的处理方式。

1. **原子操作**：原子操作是一种在多线程环境中安全执行的操作，它在执行过程中不会被其他线程中断。Linux内核提供了一组原子操作函数，如

`atomic_add()`, `atomic_sub()`, `atomic_inc()` 等。

2. **读-复制-更新（RCU）**：RCU是一种同步机制，它允许读操作在没有锁的情况下并发进行。RCU通过在更新数据结构时创建和维护其副本来实现这一点。
3. **顺序锁（Seqlock）**：顺序锁是一种特殊类型的锁，它允许多个读者和一个写者并发访问数据。顺序锁通过维护一个表示锁状态的序列号来实现这一点。
4. **位锁（Bitlock）**：位锁是一种轻量级的锁，它只使用一个位来表示锁的状态。位锁适用于保护小的、独立的数据结构。

5.7.1. 不加锁算法(Lock-Free Algorithms)

有时候，你可以重新设计你的算法，以避免完全需要锁定。如果只有一个写入者，许多读/写情况通常可以以这种方式工作。如果写入者确保数据结构的视图，如被读取者所看到的，始终是一致的，那么可能可以创建一个无锁的数据结构。

对于无锁生产者/消费者任务通常很有用的一种数据结构是循环缓冲区。这个算法涉及到生产者将数据放入数组的一端，而消费者从另一端移除数据。当到达数组的末尾时，生产者会回到开始处。因此，一个循环缓冲区需要一个数组和两个索引值，以跟踪下一个新值应该去哪里，以及下一个应该从缓冲区中移除哪个值。

当谨慎实现时，如果没有多个生产者或消费者，循环缓冲区不需要任何锁定。生产者是唯一被允许修改写入索引和它指向的数组位置的线程。只要写入者在更新写入索引之前将新值存入缓冲区，读取者就会始终看到一致的视图。反过来，读取者是唯一可以访问读取索引和它指向的值的线程。只要小心确保这两个指针不会相互超越，生产者和消费者就可以同时访问缓冲区，没有竞态条件。

图5-1显示了几种填充状态的循环缓冲区。这个缓冲区被定义为，当读取和写入指针相等时，表示为空条件，而当写入指针紧跟在读取指针后面时（注意考虑到环绕！），表示为满条件。当谨慎编程时，这个缓冲区可以无锁使用。

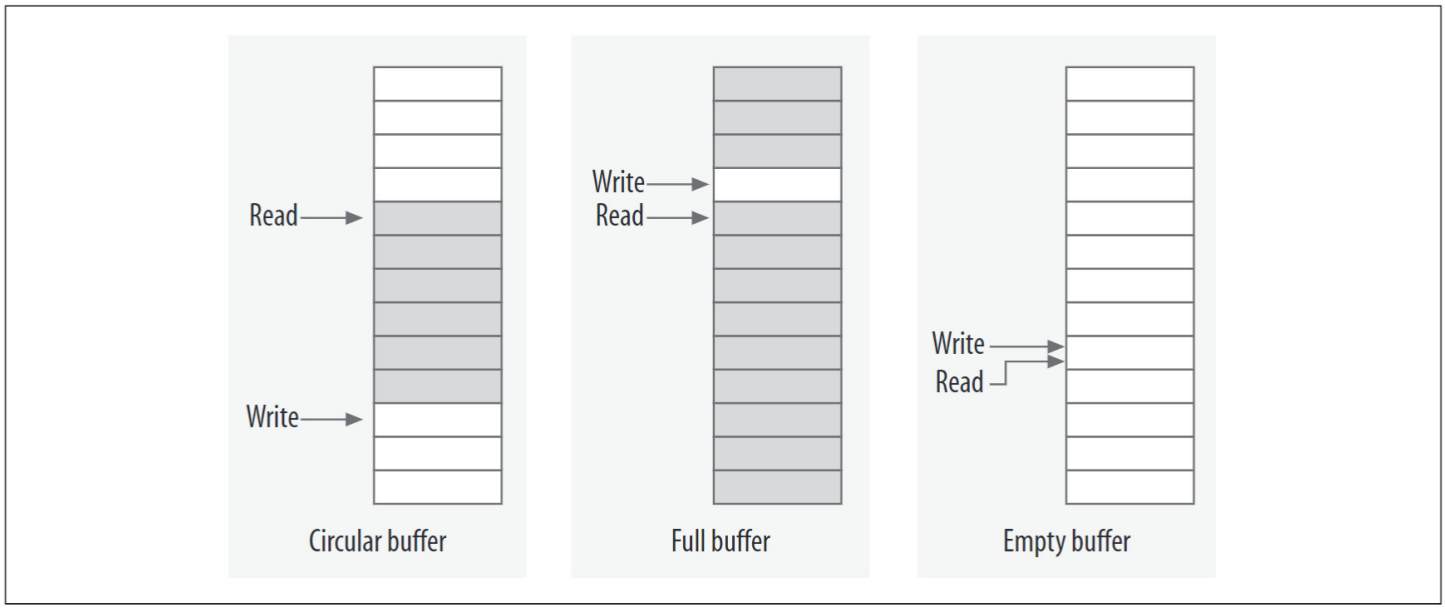

Figure 5-1. A circular buffer

循环缓冲区在设备驱动程序中出现的频率相当高。特别是网络适配器，经常使用循环缓冲区来交换数据（包）。注意，从2.6.10开始，内核中有一个通用的循环缓冲区实现；参见<linux/kfifo.h>了解如何使用它。

5.7.2. 原子变量(Atomic Variables)

有时，一个共享资源是一个简单的整数值。假设你的驱动程序维护一个共享变量n_op，它表示当前有多少设备操作正在进行。通常，即使是像：

`n_op++;`

这样的简单操作也需要锁定。有些处理器可能会以原子方式执行这种增量，但你不能依赖它。但是，对于一个简单的整数值来说，完全的锁定机制似乎过于复杂。对于这样的情况，内核提供了一个名为`atomic_t`的原子整数类型，定义在`<asm/atomic.h>`中。

一个`atomic_t`在所有支持的架构上都持有一个int值。然而，由于这种类型在某些处理器上的工作方式，可能无法使用完整的整数范围；因此，你不应该指望一个atomic_t能够持有超过24位。以下操作被定义为该类型，并且保证对于一个SMP计算机的所有处理器都是原子的。这些操作非常快，因为它们尽可能地编译为一条机器指令。

以下是一些原子操作函数的中文解释：

```c
void atomic_set(atomic_t *v, int i);
```

将原子变量v设置为整数值i。你也可以使用ATOMIC_INIT宏在编译时初始化原子值。

```c
atomic_t v = ATOMIC_INIT(0);
```

在编译时初始化原子变量v为0。

```c
int atomic_read(atomic_t *v);
```

返回v的当前值。

```c
void atomic_add(int i, atomic_t *v);
```

将i加到由v指向的原子变量。返回值是void，因为返回新值有额外的成本，而大多数时候没有必要知道它。

```c
void atomic_sub(int i, atomic_t *v);
```

从* v中减去i。

```c
void atomic_inc(atomic_t *v);

void atomic_dec(atomic_t *v);
```

增加或减少一个原子变量。

```c
int atomic_inc_and_test(atomic_t *v);

int atomic_dec_and_test(atomic_t *v);

int atomic_sub_and_test(int i, atomic_t *v);
```

执行指定的操作并测试结果；如果操作后，原子值为0，那么返回值为真；否则，为假。注意，没有atomic_add_and_test。

```c
int atomic_add_negative(int i, atomic_t *v);
```

将整数变量i加到v。如果结果为负，则返回值为真，否则为假。

```c
int atomic_add_return(int i, atomic_t *v);

int atomic_sub_return(int i, atomic_t *v);

int atomic_inc_return(atomic_t *v);

int atomic_dec_return(atomic_t *v);
```

行为就像atomic_add和它的朋友们一样，只是它们将原子变量的新值返回给调用者。

如前所述，atomic_t数据项只能通过这些函数进行访问。如果你将一个原子项传递给一个期望整数参数的函数，你将得到一个编译器错误。

你还应该记住，atomic_t值只在问题的数量真正是原子的时候才有效。需要多个atomic_t变量的操作仍然需要其他类型的锁定。

- 虽然atomic_t类型的变量可以保证单个操作是原子的，但是如果你需要执行涉及多个atomic_t变量的复合操作，那么这些操作就不再是原子的。
  考虑以下代码：

```c
atomic_sub(amount, &first_atomic);

atomic_add(amount, &second_atomic);
```

有一段时间，amount已经从第一个原子值中减去，但还没有加到第二个原子值中。如果这种情况可能为可能在两个操作之间运行的代码造成麻烦，必须使用某种形式的锁定。

5.7.3. 位操作(Bit Operations)

`atomic_t`类型非常适合执行整数运算。然而，当你需要以原子方式操作单个位时，它的效果就不那么好了。为此，内核提供了一组可以原子地修改或测试单个位的函数。因为整个操作在一个步骤中完成，所以没有中断（或其他处理器）可以干扰。

原子位操作非常快，因为它们使用单个机器指令执行操作，只有在底层平台可以做到的时候才会禁用中断。这些函数依赖于架构，并在<asm/bitops.h>中声明。即使在SMP计算机上，它们也保证是原子的，对于保持处理器间的一致性非常有用。

不幸的是，这些函数中的数据类型也依赖于架构。`nr`参数（描述要操作的位）通常定义为`int`，但在一些架构中是`unsigned long`。要修改的地址通常是指向`unsigned long`的指针，但是一些架构使用`void *`。

可用的位操作有：

- `void set_bit(nr, void *addr);` 设置由addr指向的数据项中的位号nr。
- `void clear_bit(nr, void *addr);` 清除位于addr处的unsigned long数据项中的指定位。其语义与set_bit相同。
- `void change_bit(nr, void *addr);` 切换位。
- `test_bit(nr, void *addr);` 这个函数是唯一不需要原子的位操作；它只返回位的当前值。
- `int test_and_set_bit(nr, void *addr);`
- `int test_and_clear_bit(nr, void *addr);`
- `int test_and_change_bit(nr, void *addr);` 这些函数的行为像前面列出的那样是原子的，除此之外，它们还返回位的前一个值。

当这些函数用于访问和修改共享标志时，你不需要做任何事情，只需要调用它们；它们以原子方式执行操作。使用位操作管理控制对共享变量的访问的锁变量，另一方面，稍微复杂一些，值得一个例子。大多数现代代码不以这种方式使用位操作，但是像下面这样的代码仍然存在于内核中。

需要访问共享数据项的代码段试图使用`test_and_set_bit`或`test_and_clear_bit`原子地获取锁。通常的实现如下所示；它假设锁位于地址`addr`的位`nr`。它还假设当锁空闲时位为0，当锁忙时位为非零。

```c
/* 尝试设置锁 */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();
/* 执行你的工作 */
/* 释放锁，并检查... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong(); /* 已经释放：错误 */
```

如果你阅读内核源码，你会发现有像这个例子一样的代码。然而，在新代码中使用自旋锁要好得多；自旋锁经过了良好的调试，它们处理了像中断和内核抢占这样的问题，其他阅读你的代码的人不必努力理解你在做什么。

5.7.4. seqlock 锁

2.6版本的内核包含了一些新的机制，旨在提供快速的、无锁的访问共享资源的方式。Seqlocks适用于需要保护的资源小、简单、访问频繁，且写访问很少但必须快速的情况。本质上，它们通过允许读者自由访问资源，但要求读者检查与写者的冲突，并在发生这种冲突时重试访问。Seqlocks通常不能用于保护涉及指针的数据结构，因为读者可能在跟踪一个无效的指针，而写者正在改变数据结构。

Seqlocks在<linux/seqlock.h>中定义。初始化seqlock（其类型为seqlock_t）有两种常见的方法：

```c
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;

seqlock_init(&lock2);
```

读取访问是通过在进入临界区时获取一个（无符号）整数序列值来工作的。在退出时，将该序列值与当前值进行比较；如果有不匹配，必须重试读取访问。因此，读者代码有如下形式：

```c
unsigned int seq;

do {

    seq = read_seqbegin(&the_lock);

    /* 做你需要做的事情 */

} while read_seqretry(&the_lock, seq);
```

这种锁通常用于保护需要多个一致值的简单计算。如果计算结束时的测试显示并发写入发生了，结果可以简单地被丢弃并重新计算。

如果你的seqlock可能会从中断处理程序中访问，你应该使用IRQ-safe版本：

```c
unsigned int read_seqbegin_irqsave(seqlock_t *lock,
                                   unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int
seq,
                             unsigned long flags);
```

写者必须获取一个独占锁才能进入由seqlock保护的临界区。要做到这一点，调用：

```c
void write_seqlock(seqlock_t *lock);
```

写锁是用自旋锁实现的，所以所有常见的约束都适用。调用以下函数来释放锁：

```c
void write_sequnlock(seqlock_t *lock);
```

由于使用自旋锁来控制写访问，所以所有常见的变体都可用：

```c
void write_seqlock_irqsave(seqlock_t *lock, unsigned long
flags);

void write_seqlock_irq(seqlock_t *lock);

void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned
long flags);

void write_sequnlock_irq(seqlock_t *lock);

void write_sequnlock_bh(seqlock_t *lock);
```

还有一个`write_tryseqlock`，如果能够获取锁，它会返回非零值。

5.7.5. 读取-拷贝-更新

读-复制-更新（Read-Copy-Update，RCU）是一种高级的互斥排斥方案，在适当的条件下可以产生高性能。在驱动程序中使用它是罕见的，但并非未知，所以在这里进行一个快速的概述是值得的。对RCU算法的全部细节感兴趣的人可以在其创建者发布的白皮书中找到它们 http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html (%22http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html%22)

RCU对它可以保护的数据结构类型施加了许多约束。它针对读取常见和写入稀少的情况进行了优化。被保护的资源应通过指针访问，所有对这些资源的引用只能由原子代码持有。当需要改变数据结构时，写线程创建一个副本，改变副本，然后将相关指针指向新版本——因此，算法的名称。当内核确定没有对旧版本的引用时，它可以被释放。

作为RCU在实际使用中的一个例子，考虑网络路由表。每个出站数据包都需要检查路由表，以确定应使用哪个接口。检查是快速的，一旦内核找到了目标接口，它就不再需要路由表条目。RCU允许路由查找在没有锁定的情况下进行，带来显著的性能优势。内核中的Starmode无线电IP驱动程序也使用RCU来跟踪其设备列表。

使用RCU的代码应包含<linux/rcupdate.h>。

在读取方面，使用RCU保护的数据结构的代码应该用rcu_read_lock和rcu_read_unlock的调用来包围其引用。因此，RCU代码往往看起来像这样：

```c
struct my_stuff *stuff;

rcu_read_lock( );

stuff = find_the_stuff(args...);

do_something_with(stuff);

rcu_read_unlock( );
```

`rcu_read_lock`调用是快速的；它禁用内核抢占，但不等待任何东西。在持有读取“锁”时执行的代码必须是原子的。在调用`rcu_read_unlock`之后，不得使用对受保护资源的引用。

需要改变受保护结构的代码必须执行几个步骤。第一部分很简单；它分配一个新的结构，如果需要，从旧的复制数据，然后替换读代码看到的指针。在这一点上，对于读取方来说，更改已经完成；任何进入临界区的代码都看到数据的新版本。

剩下的就是释放旧版本。问题当然是，运行在其他处理器上的代码可能仍然有对旧数据的引用，所以不能立即释放。相反，写代码必须等待，直到它知道不可能存在这样的引用。由于所有持有对这个数据结构的引用的代码必须（按规则）是原子的，我们知道一旦系统上的每个处理器都至少被调度一次，所有的引用都必须消失。所以这就是RCU所做的；它设置一个回调，等待所有处理器都已调度；然后运行该回调来执行清理工作。

改变一个RCU保护的数据结构的代码必须通过分配一个struct rcu_head来获取其清理回调，尽管它不需要以任何方式初始化该结构。通常，该结构只是简单地嵌入在被RCU保护的更大的资源中。在对该资源的更改完成后，应该调用：

```c
void call_rcu(struct rcu_head *head, void (*func)(void
*arg), void *arg);
```

当可以安全地释放资源时，调用给定的func；它传递给call_rcu传递的相同的arg。通常，func需要做的唯一事情就是调用kfree。

我们在这里看到的完整的RCU接口比我们在这里看到的更复杂；例如，它包括用于处理受保护的链表的实用函数。查看相关的头文件以获取完整的故事。

5.8. 快速参考

本章介绍了一组用于管理并发的重要符号。这些符号中最重要的在这里进行了总结：

```c
#include <asm/semaphore.h>
```

这个包含文件定义了信号量和对它们的操作。

```c
DECLARE_MUTEX(name);

DECLARE_MUTEX_LOCKED(name);
```

这两个宏用于声明和初始化用于互斥模式的信号量。

```c
void init_MUTEX(struct semaphore *sem);

void init_MUTEX_LOCKED(struct semaphore *sem);
```

这两个函数可以用于在运行时初始化一个信号量。

```c
void down(struct semaphore *sem);

int down_interruptible(struct semaphore *sem);

int down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);
```

锁定和解锁一个信号量。如果需要，down会将调用进程置于不可中断的睡眠状态；down_interruptible可以被信号中断。down_trylock不会睡眠；相反，如果信号量不可用，它会立即返回。锁定信号量的代码最终必须用up解锁它。

```c
struct rw_semaphore;
```

信号量的读/写版本和初始化它的函数。

```c
init_rwsem(struct rw_semaphore *sem);
```

初始化读/写信号量的函数。

```c
void down_read(struct rw_semaphore *sem);

int down_read_trylock(struct rw_semaphore *sem);
```

```
void up_read(struct rw_semaphore *sem);
```

获取和释放读/写信号量读取权限的函数。

```
void down_write(struct rw_semaphore *sem)

int down_write_trylock(struct rw_semaphore *sem)

void up_write(struct rw_semaphore *sem)

void downgrade_write(struct rw_semaphore *sem)
```

管理读/写信号量写入权限的函数。

```
#include <linux/completion.h>

DECLARE_COMPLETION(name);

init_completion(struct completion *c);

INIT_COMPLETION(struct completion c);
```

描述Linux完成机制的包含文件，以及初始化完成的常规方法。
INIT_COMPLETION只应用于重新初始化已经使用过的完成。

```
void wait_for_completion(struct completion *c);
```

等待一个完成事件被信号。

```
void complete(struct completion *c);

void complete_all(struct completion *c);
```

信号一个完成事件。complete最多唤醒一个等待的线程，而complete_all唤醒所有等待者。

```
void complete_and_exit(struct completion *c, long retval);
```

通过调用complete和为当前线程调用exit来信号一个完成事件。

```
#include <linux/spinlock.h>
```

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;

spin_lock_init(spinlock_t *lock);
```

定义spinlock接口的包含文件和初始化锁的两种方式。

```
void spin_lock(spinlock_t *lock);

void spin_lock_irqsave(spinlock_t *lock, unsigned long
flags);

void spin_lock_irq(spinlock_t *lock);

void spin_lock_bh(spinlock_t *lock);
```

锁定spinlock的各种方式，可能禁用中断。

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

上述函数的非旋转版本；如果无法获取锁，则返回0，否则返回非零。

```
void spin_unlock(spinlock_t *lock);

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long
flags);

void spin_unlock_irq(spinlock_t *lock);

void spin_unlock_bh(spinlock_t *lock);
```

释放spinlock的对应方式。

```
rwlock_t lock = RW_LOCK_UNLOCKED

rwlock_init(rwlock_t *lock);
```

初始化读/写锁的两种方式。

```
void read_lock(rwlock_t *lock);
```

```
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);

void read_lock_irq(rwlock_t *lock);

void read_lock_bh(rwlock_t *lock);
```

获取读/写锁读取权限的函数。

```
void read_unlock(rwlock_t *lock);

void read_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);

void read_unlock_irq(rwlock_t *lock);

void read_unlock_bh(rwlock_t *lock);
```

释放读/写spinlock读取权限的函数。

```
void write_lock(rwlock_t *lock);

void write_lock_irqsave(rwlock_t *lock, unsigned long
flags);

void write_lock_irq(rwlock_t *lock);

void write_lock_bh(rwlock_t *lock);
```

获取读/写锁写入权限的函数。

```
void write_unlock(rwlock_t *lock);

void write_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);

void write_unlock_irq(rwlock_t *lock);

void write_unlock_bh(rwlock_t *lock);
```

释放读/写spinlock写入权限的函数。

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

原子访问整数变量。atomic_t变量只能通过这些函数访问。

```
#include <asm/bitops.h>

void set_bit(nr, void *addr);

void clear_bit(nr, void *addr);

void change_bit(nr, void *addr);

test_bit(nr, void *addr);

int test_and_set_bit(nr, void *addr);

int test_and_clear_bit(nr, void *addr);

int test_and_change_bit(nr, void *addr);
```

原子访问位值；它们可以用于标志或锁变量。使用这些函数可以防止与并发访问位相关的任何竞态条件。

```
#include <linux/seqlock.h>
```

```
seqlock_t lock = SEQLOCK_UNLOCKED;

seqlock_init(seqlock_t *lock);
```

定义seqlocks的包含文件和初始化它们的两种方式。

```
unsigned int read_seqbegin(seqlock_t *lock);

unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned
long flags);

int read_seqretry(seqlock_t *lock, unsigned int seq);

int read_seqretry_irqrestore(seqlock_t *lock, unsigned int
seq, unsigned long flags);
```

获取对seqlock保护的资源的读取权限的函数。

```
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long
flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
int write_tryseqlock(seqlock_t *lock);
```

获取对seqlock保护的资源的写入权限的函数。

```
void write_sequnlock(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned
long flags);

void write_sequnlock_irq(seqlock_t *lock);

void write_sequnlock_bh(seqlock_t *lock);
```

释放对seqlock保护的资源的写入权限的函数。

```
#include <linux/rcupdate.h>
```

使用读-复制-更新（RCU）机制所需的包含文件。

```
void rcu_read_lock;

void rcu_read_unlock;
```

获取对RCU保护的资源的原子读取权限的宏。

```
void call_rcu(struct rcu_head *head, void (*func)(void
*arg), void *arg);
```

在所有处理器都已调度并且可以安全释放RCU保护的资源后，安排回调运行。