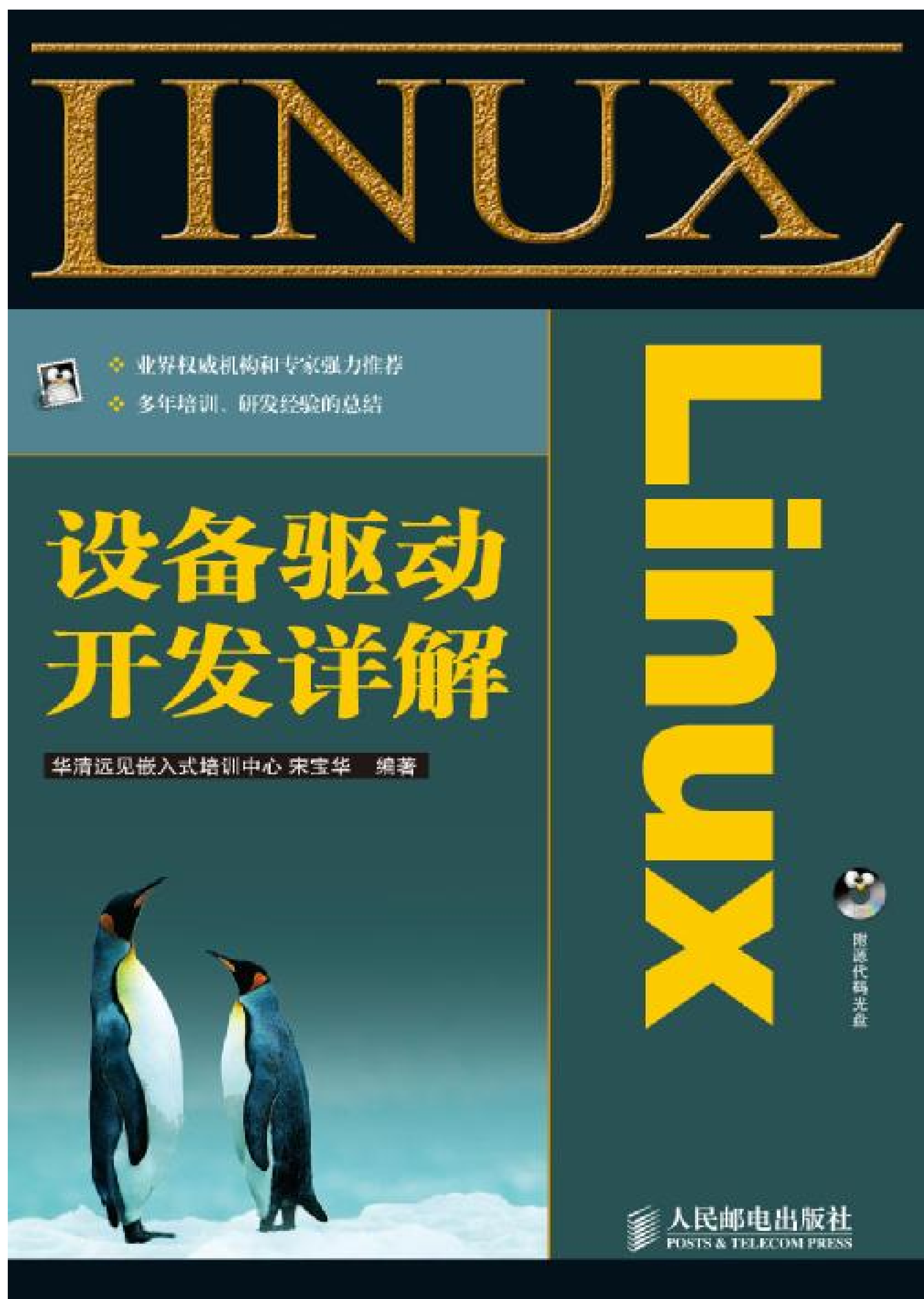
LINUX
业界权威机构和专家强力推荐
多年培训、研发经验的总结
设备驱动开发详解
Linux
华清远见嵌入式培训中心 宋宝华 编著
附源代码光盘
人民邮电出版社
POSTS & TELECOM PRESS

第 7 章 Linux 设备驱动中的并发控制

Linux 设备驱动中必须解决的一个问题是多个进程对共享资源的并发访问，并发访问会导致竞态。

Linux 提供了多种解决竞态问题的方式，这些方式适合不同的应用场景。7.1 节描述了并发和竞态的概念及发生场合。7.2～7.5 节分别讲解了中断屏蔽、原子操作、自旋锁和信号量等并发控制机制。7.6 节讲解增加并发控制后的 globalmem 的设备驱动。

华清远见

7.1 并发与竞态

并发（concurrency）指的是多个执行单元同时、并行被执行，而并发的执行单元对共享资源（硬件资源和软件上的全局变量、静态变量等）的访问则很容易导致竞态（race conditions）。例如，对于 globalmem 设备，假设一个执行单元 A 对其写入 3000 个字符“a”，而另一个执行单元 B 对其写入 4000 个字符“b”，第三个执行单元 C 读取 globalmem 的所有字符。如果执行单元 A、B 的写操作如图 7.1 所示的顺序执行，执行单元 C 的读操作不会有问题。但是，如果执行单元 A、B 如图 7.2 所示的顺序执行，而执行单元 C 又“不合时宜”地读，则会读出 3000 个“b”。

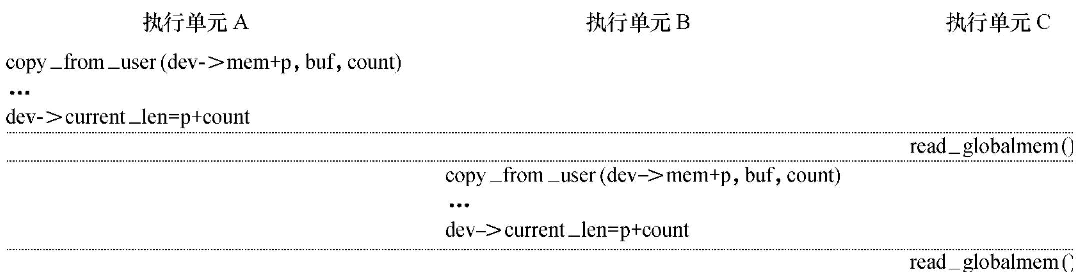

图 7.1　并发执行单元的顺序执行

比图 7.2 更复杂、更混乱的并发大量地存在于设备驱动中，只要并发的多个执行单元存在对共享资源的访问，竞态就可能发生。在 Linux 内核中，主要的竞态发生于如下几种情况。

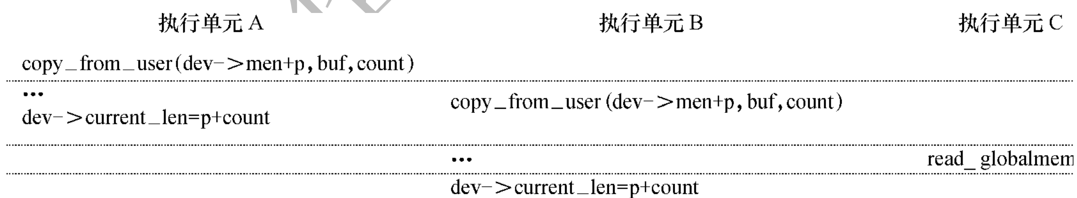

图 7.2　并发执行单元的交错执行

1．对称多处理器（SMP）的多个 CPU

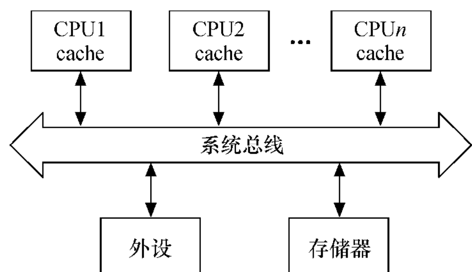

图 7.3　SMP 体系结构

SMP 是一种紧耦合、共享存储的系统模型，其体系结构如图 7.3 所示，它的特点是多个 CPU 使用共同的系统总线，因此可访问共同的外设和储存器。

2．单 CPU 内进程与抢占它的进程

Linux 2.6 内核支持抢占调度，一个进程

嵌入式学院
华清远见旗下品牌

在内核执行的时候可能被另一高优先级进程打断，进程与抢占它的进程访问共享资源的情况类似于 SMP 的多 个 CPU。

3．中断（硬中断、软中断、Tasklet、底半部）与进程之间

中断可以打断正在执行的进程，如果中断处理程序访问进程正在访问的资源，则竞态也会发生。

此外，中断也有可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态。

上述并发的发生情况除了 SMP 是真正的并行以外，其他的都是“宏观并行、微观串行”的，但其引发的实质问题和 SMP 相似。

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。

访问共享资源的代码区域称为临界区（critical sections），临界区需要以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁和信号量等是 Linux 设备驱动中可采用的互斥途径，7.2～7.5 节将进行一一讲解。

7.2 中断屏蔽

在单 CPU 范围内避免竞态的一种简单方法是在进入临界区之前屏蔽系统的中断。CPU 一般都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生，而且，由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也就得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable() //屏蔽中断
...
critical section //临界区
...
local_irq_enable()  //开中断
```

由于 Linux 系统的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。

local_irq_disable()和 local_irq_enable()都只能禁止和使能本 CPU 内的中断，因此，并不能解决 SMP 多 CPU 引发的竞态。因此，单独使用中断屏蔽通常不是一种值得推荐的避免竞态的方法，它适宜与自旋锁联合使用。

与 local_irq_disable()不同的是，local_irq_save（flags）除了进行禁止中断的操作以外，还保存目前 CPU 的中断位信息，local_irq_restore（flags）进行的是与 local_irq_save（flags）相反的操作。

如果只是想禁止中断的底半部，应使用 local_bh_disable()，使能被 local_bh_disable() 禁止的底半部应该调用 local_bh_enable()。

7.3 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

Linux 内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不被打断。位和整型变量原子操作都依赖底层 CPU 的原子操作来实现，因此所有这些函数都与 CPU 架构密切相关。

7.3.1 整型原子操作

1．设置原子变量的值

```
void atomic_set(atomic_t *v, int i); //设置原子变量的值为 i
atomic_t v = ATOMIC_INIT(0); //定义原子变量 v 并初始化为 0
```

2．获取原子变量的值

```
atomic_read(atomic_t *v);  //返回原子变量的值
```

3．原子变量加/减

```
void atomic_add(int i, atomic_t *v); //原子变量增加 i
void atomic_sub(int i, atomic_t *v); //原子变量减少 i
```

4．原子变量自增/自减

```
void atomic_inc(atomic_t *v);    //原子变量增加 1
void atomic_dec(atomic_t *v);    //原子变量减少 1
```

5．操作并测试

```
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);
```

上述操作对原子变量执行自增、自减和减操作后（注意没有加）测试其是否为 0，为 0 则返回 true，否则返回 false。

6．操作并返回

```
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);
```

上述操作对原子变量进行加/减和自增/自减操作，并返回新的值。

华清远见

7.3.2 位原子操作

1．设置位

```
void set_bit(nr, void *addr);
```

上述操作设置 addr 地址的第 nr 位，所谓设置位即将位写为 1。

2．清除位

```
void clear_bit(nr, void *addr);
```

上述操作清除 addr 地址的第 nr 位，所谓清除位即将位写为 0。

3．改变位

```
void change_bit(nr, void *addr);
```

上述操作对 addr 地址的第 nr 位进行反置。

4．测试位

```
test_bit(nr, void *addr);
```

上述操作返回 addr 地址的第 nr 位。

5．测试并操作位

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

上述 test_and_xxx_bit（nr, void *addr）操作等同于执行 test_bit（nr, void *addr）后再执行 xxx_bit（nr, void *addr）。

代码清单 7.1 给出了原子变量的使用实例，它用于使设备最多只能被一个进程打开。

代码清单 7.1 使用原子变量使设备只能被一个进程打开

```
1  static atomic_t xxx_available = ATOMIC_INIT(1); /*定义原子变量*/
2
3  static int xxx_open(struct inode *inode, struct file *filp)
4  {
5    ...
6    if (!atomic_dec_and_test(&xxx_available))
7    {
8      atomic_inc(&xxx_available);
9      return - EBUSY; /*已经打开*/
10   }
11   ...
12   return 0; /* 成功 */
13 }
```

```
14
15 static int xxx_release(struct inode *inode, struct file *filp)
16 {
17   atomic_inc(&xxx_available); /* 释放设备 */
18   return 0;
19 }
```

7.4 自旋锁

7.4.1 自旋锁的使用

自旋锁（spin lock）是一种对临界资源进行互斥手访问的典型手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。

如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。

理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区或者标记为“我当前在运行，请稍等一会”或者标记为“我当前不在运行，可以被使用”。如果 A 执行单元首先进入例程，它将持有自旋锁；当 B 执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A 执行单元释放后才能进入。

Linux 系统中与自旋锁相关的操作主要有如下 4 种。

1．定义自旋锁

```
spinlock_t spin;
```

2．初始化自旋锁

```
spin_lock_init(lock)
```

该宏用于动态初始化自旋锁 lock

3．获得自旋锁

```
spin_lock(lock)
```

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放；

```
spin_trylock(lock)
```

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返

回假，实际上不再“在原地打转”；

4．释放自旋锁

```
spin_unlock(lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。

自旋锁一般这样被使用，如下所示：

```
//定义一个自旋锁
spinlock_t lock;
spin_lock_init(&lock);

spin_lock (&lock) ; //获取自旋锁，保护临界区
...//临界区
spin_unlock (&lock) ; //解锁
```

自旋锁主要针对 SMP 或单 CPU 但内核可抢占的情况，对于单 CPU 和内核不支持抢占的系统，自旋锁退化为空操作。在单 CPU 和内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU 系统的行为实际很类似于 SMP 系统，因此，在这样的单 CPU 系统中使用自旋锁仍十分必要。

尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部（BH）的影响。为了防止这种影响，就需要用到自旋锁的衍生。spin_lock()/spin_unlock()是自旋锁机制的基础，它们和关中断 local_irq_ disable()/开中断 local_irq_enable()、关底半部 local_bh_disable()/开底半部 local_bh_enable()、关中断并保存状态字 local_irq_save()/开中断并恢复状态 local_irq_restore()结合就形成了整套自旋锁机制，关系如下所示：

```
spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_unlock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()
```

驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题。

- 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。
- 自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。此外，如果进程获得自旋锁之后再阻塞，也有可能导致死锁的发生。copy_from_user()、copy_to_user()和 kmalloc()等函数都有可能引起阻塞，因此在自旋锁的占用期间不能调用这些函数。

代码清单 7.2 给出了自旋锁的使用实例，它被用于实现使得设备只能被最多一个进程打开。

代码清单 7.2　使用自旋锁使设备只能被一个进程打开

```
1  int xxx_count = 0;/*定义文件打开次数计数*/
2
3  static int xxx_open(struct inode *inode, struct file *filp)
4  {
5    ...
6    spinlock(&xxx_lock);
7    if (xxx_count)/*已经打开*/
8    {
9      spin_unlock(&xxx_lock);
10     return  - EBUSY;
11   }
12   xxx_count++;/*增加使用计数*/
13   spin_unlock(&xxx_lock);
14   ...
15   return 0; /* 成功 */
16 }
17
18 static int xxx_release(struct inode *inode, struct file *filp)
19 {
20   ...
21   spinlock(&xxx_lock);
22   xxx_count--; /*减少使用计数*/
23   spin_unlock(&xxx_lock);
24
25   return 0;
26 }
```

7.4.2　读写自旋锁

自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生锁读写自旋锁（rwlock）可允许读的并发。

读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有一个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。

读写自旋锁涉及的操作如下所示。

1．定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态初始化 */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化 */
```

2．读锁定

```
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);
```

3．读解锁

```
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);
```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。

read_lock_irqsave()、read_lock_irq()和 read_lock_bh()分别是 read_lock()分别与 local_irq_save()、local_irq_disable()和 local_bh_disable()的组合，读解锁函数 read_unlock_irqrestore()、read_unlock_ irq()、read_unlock_bh()的情况与此类似。

4．写锁定

```
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);
```

5．写解锁

```
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);
```

write_lock_irqsave()、write_lock_irq()、write_lock_bh()分别是 write_lock()与 local_irq_save()、local_irq_disable()和 local_bh_disable()的组合，写解锁函数 write_unlock_irqrestore()、write_unlock_irq()、write_unlock_bh()的情况与此类似。

在对共享资源进行读取之前，应该先调用写锁定函数，完成之后应调用写解锁函数。和 spin_trylock()一样，write_trylock()也只是尝试获取读写自旋锁，不管成功失败，都会立即返回。

读写自旋锁一般这样被使用，如下所示：

```
rwlock_t lock;   //定义 rwlock
rwlock_init(&lock); //初始化 rwlock

//读时获取锁
read_lock(&lock);
...  //临界资源
read_unlock(&lock);

//写时获取锁
write_lock_irqsave(&lock, flags);
... //临界资源
write_unlock_irqrestore(&lock, flags);
```

7.4.3 顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元可以在写执行单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。

但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。

如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。这种锁在读写同时进行的概率比较小时，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

顺序锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致 Oops。

在 Linux 内核中，写执行单元涉及如下顺序锁操作。

1．获得顺序锁

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

其中：

```
write_seqlock_irqsave() = loal_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

2．释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中：

```
write_sequnlock_irqrestore() = write_sequnlock() + local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
...//写操作代码块
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与 spinlock 相同。

读执行单元涉及如下顺序锁操作。

1．读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```

读执行单元在对被顺序锁 s1 保护的共享资源进行访问前需要调用该函数，该函数仅返回顺序锁 s1 的当前顺序号。其中：

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

2．重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 s1 保护的共享资源后需要调用该函数来检查，在读访问期间是否有写操作。如果有写操作，读执行单元就需要重新进行读操作。其中：

```
read_seqretry_irqrestore() = read_seqretry() + local_irq_restore()
```

读执行单元使用顺序锁的模式如下：

```
do {
   seqnum = read_seqbegin(&seqlock_a);
//读操作代码块
...
} while (read_seqretry(&seqlock_a, seqnum));
```

7.4.4 读－拷贝－更新

RCU（Read-Copy Update，读－拷贝－更新），它是基于其原理命名的。RCU 并不是新的锁机制，它对于 Linux 内核而言是新的。早在 20 世纪 80 年代就有了这种机制，而在 Linux 系统中，开发 2.5.43 内核时引入该技术，并正式包含在 2.6 内核中。

对于被 RCU 保护的共享数据结构，读执行单元不需要获得任何锁就可以访问它，不使用原子指令，而且在除 Alpha 的所有架构上也不需要内存栅（Memory Barrier），因此不会导致锁竞争、内存延迟以及流水线停滞。不需要锁也使得使用更容易，因为死锁问题就不需要考虑了。

使用 RCU 的写执行单元在访问它前需首先复制一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作的时候。读执行单元没有任何同步开销，而写执行单元的同步开销则取决于使用的写执行单元间的同步机制。

RCU 可以看做读写锁的高性能版本，相比读写锁，RCU 的优点在于既允许多个读执行单元同时访问被保护的数据，又允许多个读执行单元和多个写执行单元同时访问被保护的数据。

但是，RCU 不能替代读写锁，因为如果写比较多时，对读执行单元的性能提高不能弥补写执行单元导致的损失。因为使用 RCU 时，写执行单元之间的同步开销会比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其他写执行单元的修改操作。

Linux 系统中提供的 RCU 操作如下 4 种。

1．读锁定

```
rcu_read_lock()
rcu_read_lock_bh()
```

2．读解锁

```
rcu_read_unlock()
rcu_read_unlock_bh()
```

使用 RCU 进行读的模式如下：

```
rcu_read_lock()
...//读临界区
rcu_read_unlock()
```

其中 rcu_read_lock()和 rcu_read_unlock()实质只是禁止和使能内核的抢占调度，如下所示：

```
#define rcu_read_lock()        preempt_disable()
#define rcu_read_unlock()      preempt_enable()
```

其变种 rcu_read_lock_bh()、rcu_read_unlock_bh()则定义为：

```
#define rcu_read_lock_bh()      local_bh_disable()
#define rcu_read_unlock_bh()    local_bh_enable()
```

3．同步 RCU

```
synchronize_rcu()
```

该函数由 RCU 写执行单元调用，它将阻塞写执行单元，直到所有的读执行单元已经完成读执行单元临界区，写执行单元才可以继续下一步操作。如果有多个 RCU 写执行单元调用该函数，它们将在一个 grace period（即所有的读执行单元已经完成对临界区的访问）之后全部被唤醒。synchronize_rcu()保证所有 CPU 都处理完正在运行的读执行单元临界区。

```
synchronize_kernel()
```

内核代码使用该函数来等待所有 CPU 处于可抢占状态，目前功能等同于 synchronize_rcu()，但现在已经不建议使用，而是使用 synchronize_sched()，该函数用于等待所有 CPU 都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，但不能保证正在运行的软中断处理完毕。

4．挂接回调

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

函数 call_rcu()也由 RCU 写执行单元调用，它不会使写执行单元阻塞，因而可以在中断上下文或软中断中使用。该函数将把函数 func 挂接到 RCU 回调函数链上，然后立即返回。函数 synchronize_rcu()的实现实际上使用了 call_rcu()函数。

```
void fastcall call_rcu_bh(struct rcu_head *head,
                       void (*func)(struct rcu_head *rcu));
```

call_ruc_bh()函数的功能几乎与 call_rcu()完全相同，唯一差别就是它把软中断的完成也当做经历一个 quiescent state（静默状态），因此如果写执行单元使用了该函数，

在进程上下文的读执行单元必须使用 rcu_read_lock_bh()。

每个 CPU 维护两个数据结构 rcu_data 和 rcu_bh_data，它们用于保存回调函数，函数 call_rcu()把回调函数注册到 rcu_data，而 call_rcu_bh()则把回调函数注册到 rcu_bh_data，在每一个数据结构上，回调函数被组成一个链表，先注册的排在前头，后注册的排在末尾。

使用 RCU 时，读执行单元必须提供一个信号给写执行单元以便写执行单元能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测读执行单元的信号，一旦所有的读执行单元都已经发送信号告知它们都不在使用被 RCU 保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。

RCU 还增加了链表操作函数的 RCU 版本，如下所示：

```
static inline void list_add_rcu(struct list_head *new, struct list_head *head);
```

该函数把链表元素 new 插入到 RCU 保护的链表 head 的开头，内存栅保证了在引用这个新插入的链表元素之前，新链表元素的链接指针的修改对所有读执行单元是可见的。

```
static inline void list_add_tail_rcu(struct list_head *new,
    struct list_head *head);
```

该函数类似于 list_add_rcu()，它将把新的链表元素 new 添加到被 RCU 保护的链表的末尾。

```
static inline void list_del_rcu(struct list_head *entry);
```

该函数从 RCU 保护的链表中删除指定的链表元素 entry。

```
static inline void list_replace_rcu(struct list_head *old, struct list_head *new);
```

该函数是 RCU 新添加的函数，并不存在非 RCU 版本。它使用新的链表元素 new 取代旧的链表元素 old，内存栅保证在引用新的链表元素之前，它对链接指针的修正对所有读执行单元是可见的。

```
list_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的链表 head，只要在读执行单元临界区使用该函数，它就可以安全地和其他_rcu 链表操作函数并发运行如 list_add_rcu()。

```
list_for_each_safe_rcu(pos, n, head)
```

该宏类似于 list_for_each_rcu，不同之处在于它允许安全地删除当前链表元素 pos。

```
list_for_each_entry_rcu(pos, head, member)
```

该宏类似于 list_for_each_rcu，不同之处在于它用于遍历指定类型的数据结构链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。

```
static inline void hlist_del_rcu(struct hlist_node *n)
```

它从由 RCU 保护的哈希链表中移走链表元素 n。

```
static inline void hlist_add_head_rcu(struct hlist_node *n,
    struct hlist_head *h);
```

该函数用于把链表元素 n 插入到被 RCU 保护的哈希链表的开头，但同时允许读执行单元对该哈希链表的遍历。内存栅确保在引用新链表元素之前，它对指针的修改对所有读执行单元可见。

```
hlist_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的哈希链表 head，只要在读端临界区使用该函数，它就可以安全地和其他_rcu 哈希链表操作函数（如 hlist_add_rcu）并发运行。

```
hlist_for_each_entry_rcu(tpos, pos, head, member)
```

类似于 hlist_for_each_rcu()，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。

目前，RCU 的使用在内核中已经非常普遍，内核中大量原先使用读写锁的代码被 RCU 替换，下面的表单左右两列平行地分别给出使用读写锁和 RCU 实现链表元素读、删除、添加和修改的代码。

<table>
<tr>
<td>

```
/*
  原先的 audit_filter_task 函数,读链
表前获得读写锁 */
static enum audit_state
  audit_filter_task(struct
task_struct
  *tsk)
{
  struct audit_entry *e;
  enum audit_state state;
  read_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e,    &audit_
tsklist,
    list)
  {
    if   (audit_filter_rules(tsk,
&e
      ->rule, NULL, &state))
    {
      read_unlock(&auditsc_lock);
      return state;
    }
  }
  read_unlock(&auditsc_lock);
  return AUDIT_BUILD_CONTEXT;
}


/* 原先的 audit_del_rule，删除链表元
素前获得读写锁 */
static          inline          int
audit_del_rule(struct
  audit_rule *rule, struct
```

</td>
<td>

```
/*
修改后的 audit_filter_task 函数，采用 RCU
*/
static enum audit_state
  audit_filter_task(struct
task_struct
  *tsk)
{
  struct audit_entry *e;
  enum audit_state state;
  rcu_read_lock();
  /* 遍历链表 */
  list_for_each_entry_rcu(e,
    &audit_tsklist, list)
  {
    if (audit_filter_rules(tsk, &e
      ->rule, NULL, &state))
    {
      rcu_read_unlock();
      return state;
    }
  }
  rcu_read_unlock();
  return AUDIT_BUILD_CONTEXT;
}

/*  修 改 后 的  audit_del_rule ， 使 用
list_del_rcu 删除链表元素，并调用 call_rcu 注
册回调函数 */
static          inline          int
audit_del_rule(struct
  audit_rule *rule, struct
  list_head*list)
{
  struct audit_entry *e;

  /* 遍历链表 */
  list_for_each_entry(e, list, list)
  {
    if (!audit_compare_rule(rule, &e
      ->rule))
    {
     list_del_rcu(&e->list);


call_rcu(&e->rcu,audit_free_rule,e);
      return 0;
    }
```

</td>
</tr>
</table>

```
  list_head*list)
{
  struct audit_entry *e;
  write_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e,    list,
list)
  {
    if (!audit_compare_rule(rule,
&e
      ->rule))
    {
      list_del(&e->list); //删除链
表元素

write_unlock(&auditsc_lock);
      return 0;
    }
  }
  write_unlock(&auditsc_lock);
  return  - EFAULT;
}

/* 原先的 audit_add_rule，添加链表元
素前获得读写锁 */
static   inline   int   audit_add_
rule(struct
  audit_entry *entry, struct
  list_head*list)
{
  write_lock(&auditsc_lock);
  if  (entry->rule.flags  &AUDIT_
PREPEND)
  {
    entry->rule.flags   &=   ~AUDIT_
PREPEND;
    list_add(&entry->list, list);
  }
  else
  {
```

```
  }

  return  - EFAULT;
}

/* 修改后的 audit_add_rule，在添加链表元素
时使用 list_add_xxx 函数 */
static            inline            int
audit_add_rule(struct
  audit_entry *entry, struct
  list_head*list)
{

  if              (entry->rule.flags
&AUDIT_PREPEND)
  {
    entry->rule.flags    &=    ~AUDIT_
PREPEND;
    list_add_rcu(&entry->list, list);
  }
  else
  {
    list_add_tail_rcu(&entry->list,
      list);
  }
  return 0;
}

/* 修 改 后 的 audit_upd_rule ， 使 用 list_
replace_
rcu 修改链表元素，并用 call_rcu 注册回调函数
*/
static            inline            int
audit_upd_rule(struct
  audit_rule *rule, struct list_head
  *list, __u32 newaction, __u32
  newfield_count)
{
  struct audit_entry *e;
  struct audit_newentry *ne;

  /* 遍历链表 */
  list_for_each_entry(e, list, list)
  {
    if (!audit_compare_rule(rule, &e
      ->rule))
    {
      ne = kmalloc(sizeof(*entry),
        GFP_ATOMIC);//分配新元素内存
      if (ne == NULL)
        return  - ENOMEM;
      audit_copy_rule(&ne->rule, &e
        ->rule);//写前拷贝
      ne->rule.action = newaction;
      ne->rule.file_count =
        newfield_count;
      list_replace_rcu(e, ne);
```

```
    list_add_tail(&entry->list,
list);
  }
  write_unlock(&auditsc_lock);
  return 0;
}

/*原先的 audit_upd_rule 函数，在修改链表元素前获得读写锁 */
static inline int audit_upd_rule(struct
  audit_rule *rule, struct list_head
  *list, __u32 newaction, __u32
  newfield_count)
{
  struct audit_entry *e;
  struct audit_newentry *ne;
  write_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e, list, list)
  {
    if (!audit_compare_rule(rule, &e
      ->rule))
    {
      e->rule.action = newaction;
      e->rule.file_count =
        newfield_count;
write_unlock(&auditsc_lock);
      return 0;
    }
  }
  write_unlock(&auditsc_lock);
  return - EFAULT;
}
```

```
call_rcu(&e->rcu,audit_free_rule,e);
      return 0;
    }
  }

  return - EFAULT;
}
```

7.5 信号量

7.5.1 信号量的使用

信号量（semaphore）是用于保护临界区的一种常用方法，它的使用方式和自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。

Linux 系统中与信号量相关的操作主要有如下 4 种。

1．定义信号量

下列代码定义名称为 sem 的信号量。

```
struct semaphore sem;
```

2．初始化信号量

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

该函数初始化信号量，并设置信号量 sem 的值为 val。尽管信号量可以被初始化为大于 1 的值从而成为一个计数信号量，但是它通常不被这样使用。

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

该函数用于初始化一个用于互斥的信号量，它把信号量 sem 的值设置为 1，等同于 sema_init (struct semaphore *sem, 1)。

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

该函数也用于初始化一个信号量，但它把信号量 sem 的值设置为 0，等同于 sema_init (struct semaphore *sem, 0)。

此外，下面两个宏是定义并初始化信号量的“快捷方式”。

```
DECLARE_MUTEX(name)
DECLARE_MUTEX_LOCKED(name)
```

前者定义一个名为 name 的信号量并初始化为 1，后者定义一个名为 name 的信号量并初始化为 0。

3．获得信号量

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

该函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文使用。

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

该函数功能与 down()类似，不同之处为，因为 down()而进入睡眠状态的进程不能被信号打断，而因为 down_interruptible()而进入睡眠状态的进程能被信号打断，信号也会导致该函数返回，这时候函数的返回值非 0。

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

该函数尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文使用。

在使用 down_interruptible()获取信号量时，对返回值一般会进行检查，如果非 0，通常立即返回-ERESTARTSYS，如：

```
if (down_interruptible(&sem))
{
return  - ERESTARTSYS;
}
```

4．释放信号量

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

该函数释放信号量 sem，唤醒等待者。

信号量一般这样被使用，如下所示：

```
//定义信号量
DECLARE_MUTEX(mount_sem);
down(&mount_sem);//获取信号量，保护临界区
...
critical section //临界区
...
up(&mount_sem);//释放信号量
```

Linux 自旋锁和信号量所采用的"获取锁—访问临界区—释放锁"的方式存在于几乎所有的多任务操作系统之中。

代码清单 7.3 给出了使用信号量实现设备只能被一个进程打开的例子，等同于代码清单 7.1 和代码清单 7.2。

代码清单 7.3　使用信号量实现设备只能被一个进程打开

```
1  static DECLARE_MUTEX(xxx_lock);//定义互斥锁
2
3  static int xxx_open(struct inode *inode, struct file *filp)
4  {
5    ...
6    if (down_trylock(&xxx_lock))  //获得打开锁
7       return  - EBUSY;  //设备忙
8    ...
9    return 0; /* 成功 */
10 }
11
12 static int xxx_release(struct inode *inode, struct file *filp)
13 {
14   up(&xxx_lock);  //释放打开锁
15   return 0;
16 }
```

7.5.2　信号量用于同步

如果信号量被初始化为 0，则它可以用于同步，同步意味着一个执行单元的继续执行需等待另一执行单元完成某事，保证执行的先后顺序。如图 7.4 所示，执行单元 A 执行代码区域 b 之前，必须等待执行单元 B 执行完代码单元 c，信号量可辅助这一同步过程。

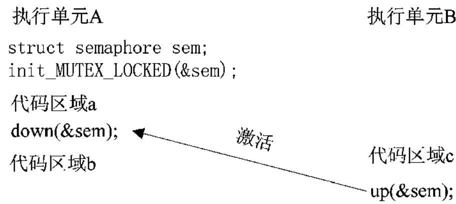

图 7.4　信号量用于同步

7.5.3　完成量用于同步

Linux 系统提供了一种比信号量更好的同步机制，即完成量（completion，关于这个名词，至今没有好的翻译，笔者将其译为“完成量”），它用于一个执行单元等待另一个执行单元执行完某事。

Linux 系统中与 completion 相关的操作主要有以下 4 种。

1．定义完成量

下列代码定义名为 my_completion 的完成量。

```
struct completion my_completion;
```

2．初始化 completion

下列代码初始化 my_completion 这个完成量。

```
init_completion(&my_completion);
```

对 my_completion 的定义和初始化可以通过如下快捷方式实现。

```
DECLARE_COMPLETION(my_completion);
```

3．等待完成量

下列函数用于等待一个 completion 被唤醒。

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

4．唤醒完成量

下面两个函数用于唤醒完成量。

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

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

前者只唤醒一个等待的执行单元，后者释放所有等待同一完成量的执行单元。

图 7.5 描述了使用完成量实现的同步功能。

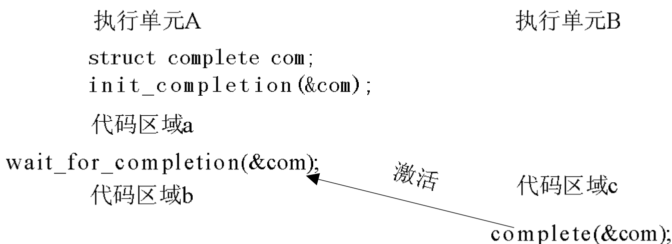

图 7.5 完成量用于同步

7.5.4 自旋锁 vs 信号量

自旋锁和信号量都是解决互斥问题的基本手段，面对特定的情况，应该如何进行选择呢？选择的依据是临界区的性质和系统的特点。

从严格意义上说，信号量和自旋锁属于不同层次的互斥手段，前者的实现依赖于后者。在信号量本身的实现上，为了保证信号量结构存取的原子性，在多 CPU 中需要自旋锁来互斥。

信号量是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份，代表进程来争夺资源的。如果竞争失败，会发生进程上下文切换，当前进程进入睡眠状态，CPU 将运行其他进程。鉴于进程上下文切换的开销也很大，因此，只有当进程占用资源时间较长时，用信号量才是较好的选择。

当所要保护的临界区访问时间比较短时，用自旋锁是非常方便的，因为它节省上下文切换的时间。但是 CPU 得不到自旋锁会在那里空转直到其他执行单元解锁为止，所以要求锁不能在临界区里长时间停留，否则会降低系统的效率。

由此，可以总结出自旋锁和信号量选用的 3 项原则。

- 当锁不能被获取时，使用信号量的开销是进程上下文切换时间 Tsw，使用自旋锁的开销是等待获取自旋锁（由临界区执行时间决定）Tcs，若 Tcs 比较小，应使用自旋锁，若 Tcs 很大，应使用信号量。
- 信号量所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就会发生。
- 信号量存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在信号量和自旋锁之间只能选择自旋锁。当然，如果一定要使用信号量，则只能通过 down_trylock()方式进行，不能获取就立即返回以避免阻塞。

华清远见

7.5.5 读写信号量

读写信号量与信号量的关系与读写自旋锁和自旋锁的关系类似，读写信号量可能引起进程阻塞，但它可允许 *N* 个读执行单元同时访问共享资源，而最多只能有一个写执行单元。因此，读写信号量是一种相对放宽条件的粒度稍大于信号量的互斥机制。

读写自旋锁涉及的操作包括如下 5 种。

1．定义和初始化读写信号量

```
struct rw_semaphore my_rws;  /*定义读写信号量*/
void init_rwsem(struct rw_semaphore *sem);  /*初始化读写信号量*/
```

2．读信号量获取

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

3．读信号量释放

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

4．写信号量获取

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
```

5．写信号量释放

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

读写信号量一般这样被使用，如下所示：

```
rw_semaphore rw_sem;   //定义读写信号量
init_rwsem(&rw_sem); //初始化读写信号量

//读时获取信号量
down_read(&rw_sem);
...  //临界资源
up_read(&rw_sem);

//写时获取信号量
down_write(&rw_sem);
... //临界资源
up_write(&rw_sem);
```

7.6 互斥体

尽管信号量已经可以实现互斥的功能，而且包含 DECLARE_MUTEX()、init_MUTEX ()等定义信号量的宏或函数，从名字上看就体现出了互斥体的概念，但是 mutex 在 Linux 内核中还是真实地存在的。

下面代码定义名为 my_mutex 的互斥体并初始化它。

```
struct mutex my_mutex;
```

```
mutex_init(&my_mutex);
```

下面的两个函数用于获取互斥体。

```
void fastcall mutex_lock(struct mutex *lock);
int fastcall mutex_lock_interruptible(struct mutex *lock);
int fastcall mutex_trylock(struct mutex *lock);
```

mutex_lock()与 mutex_lock_interruptible()的区别和 down()与 down_trylock()的区别完全一致，前者引起的睡眠不能被信号打断，而后者可以。mutex_trylock()用于尝试获得 mutex，获取不到 mutex 时不会引起进程睡眠。

下列函数用于释放互斥体。

```
void fastcall mutex_unlock(struct mutex *lock);
```

mutex 的使用方法和信号量用于互斥的场合完全一样，如下所示：

```
struct mutex my_mutex;  //定义 mutex
mutex_init(&my_mutex);  //初始化 mutex

mutex_lock(&my_mutex);  //获取 mutex
...//临界资源
mutex_unlock(&my_mutex); //释放 mutex
```

7.7 增加并发控制后的 globalmem 驱动

在 globalmem()的读写函数中，由于要调用 copy_from_user()、copy_to_user()这些可能导致阻塞的函数，因此不能使用自旋锁，宜使用信号量。

驱动工程师习惯将某设备所使用的自旋锁、信号量等辅助手段也放在设备结构中，因此，可如代码清单 7.4 那样修改 globalmem_dev 结构体的定义，并在模块初始化函数中初始化这个信号量，如代码清单 7.5 所示。

代码清单 7.4　增加并发控制后的 globalmem 设备结构体

```
1 struct globalmem_dev
2 {
3   struct cdev cdev; /*cdev 结构体*/
4   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
5   struct semaphore sem; /*并发控制用的信号量*/
6 };
```

代码清单 7.5　增加并发控制后的 globalmem 设备驱动模块加载函数

```
1  int globalmem_init(void)
2  {
3    int result;
4    dev_t devno = MKDEV(globalmem_major, 0);
```

```
5
6   /* 申请设备号*/
7   if (globalmem_major)
8     result = register_chrdev_region(devno, 1, "globalmem");
9   else  /* 动态申请设备号 */
10  {
11    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
12    globalmem_major = MAJOR(devno);
13  }
14  if (result < 0)
15    return result;
16
17  /* 动态申请设备结构体的内存*/
18      globalmem_devp   =   kmalloc(sizeof(struct   globalmem_dev),
GFP_KERNEL);
19  if (!globalmem_devp)    /*申请失败*/
20  {
21    result =  - ENOMEM;
22    goto fail_malloc;
23  }
24  memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
25
26  globalmem_setup_cdev(globalmem_devp, 0);
27  init_MUTEX(&globalfifo_devp->sem);   /*初始化信号量*/
28  return 0;
29
30  fail_malloc: unregister_chrdev_region(devno, 1);
31  return result;
32 }
```

在访问 globalmem_dev 中的共享资源时，需先获取这个信号量，访问完成后，随即释放这个信号量。驱动中新的 globalmem 读、写操作如代码清单 7.6 所示。

代码清单 7.6　增加并发控制后的 globalmem 读写操作

```
1 /*增加并发控制后的 globalmem 读函数*/
2 static ssize_t globalmem_read(struct file *filp, char __user *buf,
size_t size,
3   loff_t *ppos)
4 {
5   unsigned long p =  *ppos;
6   unsigned int count = size;
7   int ret = 0;
```

```
    8   struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指
针*/
    9
    10   /*分析和获取有效的写长度*/
    11   if (p >= GLOBALMEM_SIZE)
    12     return count ?  - ENXIO: 0;
    13   if (count > GLOBALMEM_SIZE - p)
    14     count = GLOBALMEM_SIZE - p;
    15
    16   if (down_interruptible(&dev->sem)) //获得信号量
    17   {
    18     return  - ERESTARTSYS;
    19   }
    20
    21   /*内核空间->用户空间*/
    22   if (copy_to_user(buf, (void*)(dev->mem + p), count))
    23   {
    24     ret =  - EFAULT;
    25   }
    26   else
    27   {
    28     *ppos += count;
    29     ret = count;
    30
    31     printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
    32   }
    33   up(&dev->sem); //释放信号量
    34
    35   return ret;
    36 }
    37
    38 /*增加并发控制后的globalmem写函数*/
    39 static ssize_t globalmem_write(struct file *filp, const char __user
*buf,
    40   size_t size, loff_t *ppos)
    41 {
    42   unsigned long p =  *ppos;
    43   unsigned int count = size;
    44   int ret = 0;
    45   struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指
针*/
    46
    47   /*分析和获取有效的写长度*/
    48   if (p >= GLOBALMEM_SIZE)
```

```
49      return count ? - ENXIO: 0;
50    if (count > GLOBALMEM_SIZE - p)
51      count = GLOBALMEM_SIZE - p;
52
53    if (down_interruptible(&dev->sem)) //获得信号量 54
54    {
55      return - ERESTARTSYS;
56    }
57    /*用户空间->内核空间*/
58    if (copy_from_user(dev->mem + p, buf, count))
59      ret = - EFAULT;
60    else
61    {
62      *ppos += count;
63      ret = count;
64
65      printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
66    }
67    up(&dev->sem); //释放信号量
68    return ret;
69  }
```

代码第 16～19 和第 53～56 行用于获取信号量，如果 down_interruptible()返回值非 0，则意味着其在获得信号量之前已被打断，这时写函数返回-ERESTARTSYS。代码第 33 和第 67 行用于在对临界资源访问结束后释放信号量。

除了 globalmem 的读写操作之外，如果在读写的同时，另一执行单元执行 MEM_CLEAR IO 控制命令，也会导致全局内存的混乱，因此，globalmem_ioctl()函数也需被重写，如代码清单 7.7 所示。

代码清单 7.7　增加并发控制后的 globalmem 设备驱动 ioctl()函数

```
1  static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned
2    int cmd, unsigned long arg)
3  {
4    struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
5
6    switch (cmd)
7    {
8      case MEM_CLEAR:
9        if (down_interruptible(&dev->sem))//获得信号量
10       {
11         return - ERESTARTSYS;
12       }
13       memset(dev->mem, 0, GLOBALMEM_SIZE);
14       up(&dev->sem); //释放信号量
15
16       printk(KERN_INFO "globalmem is set to zero\n");
17       break;
18
19     default:
20       return - EINVAL;
21   }
22   return 0;
23 }
```

下面是增加并发控制后的 globalmem 驱动针对代码清单 6.16 未考虑并发问题

globalmem 驱动的 patch。

```
28a30
>   struct semaphore sem; /*并发控制用的信号量*/
53a58,61
>   if (down_interruptible(&dev->sem))
>   {
>      return  - ERESTARTSYS;
>   }
54a63,64
>   up(&dev->sem); //释放信号量
>
78a89,93
>   if (down_interruptible(&dev->sem))
>   {
>     return  - ERESTARTSYS;
>   }
>
90a106
>   up(&dev->sem); //释放信号量
109a126,129
>   if (down_interruptible(&dev->sem))//获得信号量
>   {
>     return  - ERESTARTSYS;
>   }
120c140
<
---
>   up(&dev->sem); //释放信号量
216a237
>   init_MUTEX(&globalfifo_devp->sem);   /*初始化信号量*/
```

7.8 总结

并发和竞态广泛存在，中断屏蔽、原子操作、自旋锁和信号量都是解决并发问题的机制。中断屏蔽很少单独被使用，原子操作只能针对整数进行，因此自旋锁和信号量应用最为广泛。

自旋锁会导致死循环，锁定期间不允许阻塞，因此要求锁定的临界区小。信号量允许临界区阻塞，可以适用于临界区大的情况。

读写自旋锁和读写信号量分别是放宽了条件的自旋锁和信号量，它们允许多个执行单元对共享资源的并发读。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm

嵌入式学院 华清远见旗下品牌

· 课程内容：http://www.embedu.org/courses/course1.htm

· 项目实战：http://www.embedu.org/courses/project.htm

· 出版教材：http://www.embedu.org/courses/course3.htm

· 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

· 嵌入式 Linux 应用开发班：

http://www.farsight.com.cn/courses/TS-LinuxApp.htm

· 嵌入式 Linux 系统开发班：

http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

· 嵌入式 Linux 驱动开发班：

http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

嵌入式学院
华清远见旗下品牌