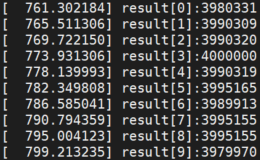
# 并发与竞态

多个执行单元（进程、中断、线程）对共享资源并发访问，可能会发生执行单元之间相互覆盖共享资源的情况，造成被访问资源处于不一致状态，导致竞态（race condition）发生，也就是说程序中就会包含一个bug。竞态问题出现机会小，并且难以追踪和调试，所以需要在程序设计时格外注意。

在示例中创建2个线程每个对result++ 2000000次，如果不进行并发控制，测试结果如下，result并不一定等于我们期望的4000000：



result++的汇编指令如下：

movl result(%rip), %eax

addl $1, %eax

movl %eax, result(%rip)

所以result++的实际机械指令序列为：

* 获取当前result值拷贝到寄存器中
* 将寄存器值加1
* 把新的result值写回到内存

在例程中，我们期望的执行过程如下，如果当前result=7：

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| 获得result 7  增加result 7-》8  写回result 8  ---  ---  --- | ---  ---  ---  获得result 8  增加result 8-》9  写回result 9 |

而实际的执行过程可能如下：

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| 获得result 7  增加result 7-》8  ---  写回result 8  --- | 获得result 7  ---  增加result 7-》8  ---  写回result 8 |

这里result变量就是共享资源，为了防止并发访问，我们需要保证访问和操作共享资源的代码作为一个不可分割的整体去执行就外事大吉了。

现实应用中的并发更加复杂与混乱，只要对共享资源有并发的访问，竞态就可能发生。（1）目前内核支持对称多处理器，支持多处理器意味着代码可以同时运行于两个或多个处理器上，所以两个处理器可能会在同一时刻并发访问共享资源。

（2）2.6版本内核后，已经发展为抢占式内核，所以当前进程与抢占它的进程可能会并发访问共享资源。

（3）中断可以打断正在执行的进程，如果中断服务中访问进程正在访问的资源，竞态也会发生。

总的竞态发生的情况在《Linux设备驱动开发详解》中的图解比较生动，如下:



上述并发的发生除了SMP是真正的并行之外，单核的其实都是“宏观并行，微观串行”，但是引发的实质问题与SMP相似。

# 并发控制

解决竟态问题就是需要保证对共享资源的互斥访问，也就是一个执行单元在访问共享资源时，其他执行单元禁止访问。

## 原子操作

原子操作指的是不能被分割的指令，可以保证指令按照原子的方式执行，也就是说指令执行过程中不会被打断。

如果使用原子操作result++的例子中的竟态就不会发生，因为result++变为一个整体，所以执行过程就会如下：

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| 获得、增加和存储result 7-》8  --- | ---  获得、增加和存储result 8-》9 |

或者

|  |  |
| --- | --- |
| 线程1 | 线程2 |
| ---  获得、增加和存储result 8-》9 | 获得、增加和存储result 7-》8  --- |

无论怎么执行都会得到期望的结果9，原子操作保证了绝对不会并发访问。

内核提供了两组原子操作接口：一组针对整数进行操作，另一组则针对单独的位进行操作。

原子整数操作列表：

|  |  |
| --- | --- |
| atomic\_t v | 定义atomic\_t类型v |
| atomic\_t v = ATOMIC\_INIT(i) | 定义原子变量v并初始化为i |
| atomic\_set(v,i) | 设置原子变量的值为i |
| atomic\_read(v) | 返回原子变量的值 |
| 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 |
| atomic\_inc(v) | 原子变量自增 |
| atomic\_dec(v) | 原子变量自减 |
| atomic\_inc\_and\_test(v) | 原子变量自增后，如果结果是否为0，为0返回true，否则返回false |
| atomic\_dec\_and\_test(v) | 原子变量自减后测试 |
| atomic\_sub\_and\_test(i, v) | 原子变量减i后测试 |
| atomic\_add\_negative(i,v) | 原子变量加i后，如果结果为负，返回true，否则返回false |
| int atomic\_add\_return(int i, atomic\_t \*v) | 原子变量加i并返回新值 |
| int atomic\_sub\_return(int i, atomic\_t \*v) | 原子变量减i并返回新值 |

原子位操作列表：

|  |  |
| --- | --- |
| set\_bit(nr,p) | 原子地设置p的对象的第nr位 |
| clear\_bit(nr,p) | 原子地清空p的对象的第nr位 |
| change\_bit(nr,p) | 原子地翻转p的对象的第nr位 |
| test\_and\_set\_bit(nr,p) | 原子地设置p的对象的第nr位，并返回原来的值 |
| test\_and\_clear\_bit(nr,p) | 原子地清空p的对象的第nr位，并返回原来的值 |
| test\_and\_change\_bit(nr,p) | 原子地翻转p的对象的第nr位，并返回原来的值 |
| find\_first\_zero\_bit(p,sz) | 原子地查找第一个未被设置的位 |
| find\_next\_zero\_bit(p,sz,off) | 原子地查找从offset第一个未被设置的位 |
| find\_first\_bit(p,sz) | 原子地查找第一个被设置的位 |
| find\_next\_bit(p,sz,off) | 原子地查找从offset第一个被设置的位 |

## 自旋锁

现实中的临界区往往不只是一个变量的加加减减，比如进行链表操作时，原子操作是无能为力的。

自旋锁是内核中最常见的锁。自旋锁最多只能被一个执行单元所持有，如果一个执行单元试图获取一个已经被持有的自旋锁（争用），那么程序将会一直进行忙等待-等待锁重新可用，如果锁空闲，则执行单元可立即获得继续程序执行。

如果临界区域是一个房间的话，自旋锁就是房间的钥匙，如果有人进房间前发现门口有钥匙，则开门拿着钥匙进入房间，此时其他人来了发现没有钥匙则会原地等待不断的检查钥匙回没回来，直到第一个人将钥匙返还，才能拿着钥匙进入房间。这里的钥匙就保证了进入房间的人最多只有一个。

### 自旋锁的使用

spinlock\_t lock

spin\_lock\_init(&lock);

spin\_lock(&lock)；

/\*临界区\*/

spin\_unlock(&lock)；

使用自旋锁可以保证临界区不受别的CPU或者本CPU进程抢占的影响，但是还是会受到硬中断以及软中断的影响，这时就需要用到如下函数，这些函数可以保证进程在自旋锁使用过程中不会被中断影响（毕竟已经将中断关闭了）。

spin\_lock\_irq() = spin\_lock()+local\_irq\_disable()（关中断）

spin\_unlock\_irq() = spin\_unlock()+local\_irq\_enable()（开中断）

spin\_lock\_irqsave() = spin\_lock()+local\_irq\_save()（关中断并保存中断状态）

spin\_unlock\_irqrestore() = spin\_unlock()+local\_irq\_restore()（开中断并恢复中断状态）

spin\_lock\_bh() = spin\_lock()+local\_bh\_disable()(关软中断)

spin\_unlock\_bh() = spin\_unlock()+local\_bh\_enable()(开软中断)

这里注意spin\_lock\_irq()与spin\_unlock\_irq()，spin\_unlock\_irq()会开启中断无论在spin\_lock\_irq()关中断前是否被激活。所以spin\_lock\_irqsave()与spin\_unlock\_irqrestore()更加实用。

在自旋锁使用时，对于多进程访问同一片临界区时，需要调用spin\_lock()/spin\_unlock()暂时禁止本核内抢占，注意并不会禁止另一个核的抢占，而对于进程和中断访问同一片临界区的情况，进程中需要调用spin\_lock\_irqsave()/spin\_unlock\_irqrestore(),暂时禁止本核内抢占以及中断的发生，中断中需要调用spin\_lock()/spin\_unlock()。此后如果CPU0的进程获得锁后，CPU0不会发生抢占与中断，而CPU1无论是进程上下文还是中断上下文，想获得同一自旋锁，会忙等待，这就避免了一切核内与核间并发的可能性。

所以自旋锁发生自旋是多核在访问临界资源时发生的，而单核的自旋锁则是通过禁止核内抢占与中断来保护临界资源的。

自旋锁方法列表：

|  |  |
| --- | --- |
| DEFINE\_SPINLOCK(lock） | 定义并初始化自旋锁 |
| spinlock\_t lock | 定义自旋锁 |
| spin\_lock\_init(); | 动态初始化锁 |
| spin\_lock() | 获取指定锁 |
| spin\_lock\_irq() | 禁止本地中断并获取指定锁 |
| spin\_lock\_irqsave() | 保存本地中断的当前状态，禁止本地中断，并获取指定锁 |
| spin\_lock\_bh() | 禁止本地软中断并获取指定锁 |
| spin\_unlock() | 释放指定锁 |
| spin\_unlock\_irq() | 释放指定锁并激活本地中断 |
| spin\_unlock\_irqrestore() | 释放指定锁并让本地中断恢复到以前状态 |
| spin\_unlock\_bh() | 释放指定锁并激活本地软中断 |
| spin\_trylock() | 试图获得指定的锁，如果未获取，则返回非0 |
| spin\_is\_locked() | 如果指定的锁当前正在被获取，则返回非0，否则返回0 |

### 自旋锁注意事项

1） 线程在请求一个被争用的自旋锁时，在锁重新可用前自旋-忙等待，CPU在等待自旋锁时不做任何有用的工作，所以占用锁的时间需要尽可能短，如果占用锁的时间较长，使用自旋锁会减低系统性能。

2）死锁。几种死锁情况：

* 一个线程试图去获得一个自己已经持有的锁，它将一直等待锁的释放，所以这个锁永远也不会有机会得到释放，导致死锁。

获得锁

再次试图获得锁（注意是试图，而且将会一直试图下去）

释放锁（永远也不可能执行到了）

* 多个线程多个锁，线程互相等待其他线程持有的锁，可能会导致死多。

线程1 线程2

获得锁A 获得锁B

试图获得锁B 试图获得锁A

释放锁A 释放锁B（永远也不会执行到了）

* 锁未被正确释放，也会导致死锁发生。

3）自旋锁期间不能调用可能引起进程调度的函数，如copy\_from\_user(),copy\_to\_user()，kmalloc()和msleep()等函数。kmalloc()在当前进程少内存时会睡眠等待页;copy\_from\_user()/copy\_to\_user()会在包含用户数据的页被换出到硬盘上而不是物理内存上时睡眠，直到缺页处理程序将该页从硬盘重新换回物理内存。如果锁期间引起调度，内核会直接崩溃：

BUG: scheduling while atomic: test/1266/0x00000002

4）单核情况下编程，也应该认为自己的CPU是多核的。在单CPU的情况下，如果中断和进程之间有临界区，进程中调用spin\_lock\_irqsave()是安全的，此时在中断中不调用spin\_lock()也不会产生问题，因为spin\_lock\_irqsave()保证了这个CPU的中断服务程序不可能执行。但是程序移植到多核平台上时，spin\_lock\_irqsave()并不能屏蔽另一个核的中断，此时另一个产生的中断就可能与这个核的进程间造成竟态的发生。所以无论如何，中断与进程可能访问同一临界区时，中断服务中都应该调用spin\_lock()。

### 读写自旋锁

自旋锁并不关心锁定的临界区内到底是读还是写，也就是说即便是多个执行单元同时读取临界资源也会被锁住，但其实多个执行单元并发读取临界资源时并不会发生问题。读写自旋锁允许读的并发，而最多只有一个写操作执行，并且写操作不能有并发的读操作。

#### 读写自旋锁的使用

DEFINE\_RWLOCK(rwclock)

read\_lock(&rwclock)

/\*临界区 只读\*/

read\_unlock(&rwclock)

write\_lock(&rwclock)

/\*临界区 读写\*/

write\_unlock(&rwclock)

同样如果想避免中断影响，还需延伸的函数，如read\_lock\_irqsave(&rwclock) //获取读锁并关中断并保存中断状态，其他使用类似。

读写自旋锁方法列表

|  |  |
| --- | --- |
| DEFINE\_RWLOCK(rwclock) | 定义并初始化读写自旋锁 |
| rwlock\_t lock | 定义读写自旋锁 |
| rwlock\_init() | 动态初始化读写自旋锁 |
| read\_lock() | 获取读锁 |
| read\_lock\_irq() | 禁止本地中断并获取读锁 |
| read\_lock\_irqsave() | 存储本地中断状态，禁止本地中断并获取读锁 |
| read\_unlock() | 释放读锁 |
| read\_unlock\_irq() | 释放读锁并激活本地中断 |
| read\_unlock\_irqrestore() | 释放读锁并将本地中断恢复到原本的状态 |
| write\_lock() | 获取写锁 |
| write\_lock\_irq() | 禁止本地中断并获取写锁 |
| write\_lock\_irqsave() | 存储本地中断状态，禁止本地中断并获取写锁 |
| write\_unlock() | 释放写锁 |
| write\_unlock\_irq() | 释放写锁并激活本地中断 |
| write\_unlock\_irqrestore() | 释放写锁并将本地中断恢复到原本的状态 |
| Write\_trylock() | 试图获得写锁，如果写锁不可用，返回非0值 |

#### 读写自旋锁注意事项

1）多个读者可以安全的获取同一读锁，但是读锁不可随便修改为写锁。

这么使用是没有问题的

read\_lock(&lock)

read\_lock(&lock)

…

但是如下就会造成死锁，因为写锁会一直等待读操作释放锁

read\_lock(&lock)

write\_lock(&lock)

2）如果读写操作不能清晰的分开的话，不要使用读写自旋锁，使用普通自旋锁即可，如果在读锁期间有写操作就可能导致竟态的发生，毕竟多个读者可以获取同一把锁。

3）程序优化，如果中断中只有读操作没有写操作，那么进程中只需调用read\_lock(),无需调用read\_lock\_irqsave()，因为读操作期间被只有读的中断打断，并不会造成竟态。但是还是需要用read\_lock\_irqsave()防止有写操作的中断抢占造成死锁（如读锁还没有释放时被中断打断，进入中断后在获取写锁时会自旋---一直等待读锁释放，而读操作又需要中断返回才能继续，死锁就会发生）。

4）读写自旋锁照顾读比照顾写多一点，毕竟读锁是可以递归获取的，而写操作只能等到所有读操作完成才能获取。所以，如果有大量读者就可能引起写者饥饿（读一直可以成功占有锁，而写一直要等待所有读锁的释放，可能会造成写锁一直不能获取）。

### 顺序锁

顺序锁是对读写锁的一种优化，使用顺序锁，读操作不会被写操作阻塞，也就是说读操作在写操作对被顺序锁保护的共享资源写操作时仍然可以继续读，而不必等待写操作完成，写操作也无需等待所有读执行单元完成读操作。但是两个写操作仍然是互斥的，如果一个写操作正在进行，其他写操作会自旋，直到顺序锁的释放。

尽管顺序锁队读写不进行互斥，但是如果读操作期间，写操作发生，读操作必须重新进行，以保证完成的数据是完整与正确的，所以，这种情况下读操作可能需要读多次才能读取到有效的数据。

#### 顺序锁的使用

seqlock\_init(&seq\_lock);

write\_seqlock(&seq\_lock);

/\*临界期 写操作\*/

write\_sequnlock(&seq\_lock);

do {

seq = read\_seqbegin(&seq\_lock);

/\*临界区 读操作\*/

} while (read\_seqretry(&seq\_lock, seq));

同样为了解决中断的影响，在可能与中断同时访问临界资源时，需要用到延伸的函数。

顺序锁方法列表：

|  |  |
| --- | --- |
| seqlock\_init() | 动态初始化顺序锁 |
| write\_seqlock() | 获取写锁 |
| write\_seqlock\_irq () | 禁止本地中断并获取写锁 |
| write\_seqlock\_irqsave() | 存储本地中断状态，禁止本地中断并获取写锁 |
| write\_sequnlock() | 释放写锁 |
| write\_sequnlock\_irq () | 释放写锁并激活本地中断 |
| write\_sequnlock\_irqrestore() | 试图获得写锁，如果写锁不可用，返回非0值 |
| read\_seqbegin() | 读开始 |
| read\_seqretry() | 重读 |

#### 顺序锁的选择

顺序锁对写更有利，只要没有其他写操作，写锁总是被成功获得，读不会影响到写，写操作期间会使得读操作不断循环，直到所有写操作不再持有锁。如下情况顺序锁是一个很好的选择：

（1）共性资源有很多读者，很少写者

（2）希望写优先于读，而且不允许读者让写者饥饿（回顾一下读写自旋锁注意事项第4条）

jiffies就用到了顺序锁，jiffies记录了系统启动以来总的时钟中断次数，get\_jiffies\_64()可以读取：

u64 get\_jiffies\_64(void)

{

unsigned long seq;

u64 ret;

do {

seq = read\_seqbegin(&jiffies\_lock);

ret = jiffies\_64;

} while (read\_seqretry(&jiffies\_lock, seq));

return ret;

}

定时器会更新jiffies值

void xtime\_update(unsigned long ticks)

{

write\_seqlock(&jiffies\_lock);

jiffies\_64 += ticks;

write\_sequnlock(&jiffies\_lock);

}

这个简直把顺序锁的优势体现的淋漓尽致，试想如果此时用的是读写自旋锁，如果读者很多，岂不是jiffies值就很难更新了。

### 读-复制-更新

RCU（READ-COPY-UPDATE），不同于自旋锁，使用RCU的读并没有锁，几乎是直接读取，而RCU的写操作在访问共享资源时首先复制一个副本，然后对副本进行更改，最后使用一个回调机制在所有引用该数据的CPU都退出对共享数据读操作的时候把指向原来数据的指针重新指向新的被修改的数据。

RCU可以看做读写锁的高性能版本，相对读写锁，RCU的优点在于既允许多个读操作同时访问共享资源，又允许多个读操作和多个写同时访问共享资源，根本无需担心死锁问题。但是RCU不能替代读写锁，因为如果写比较多时，对读操作的性能提升不能弥补写操作同步所带来的开销。

#### RCU的使用

struct foo{

int a;

int b;

};

struct foo \*gp = NULL

/\*读操作\*/

rcu\_read\_lock()//并不是真正的锁，仅标识临界区开始位置，表明临界区内不能阻塞与休眠，也不能让写者进行数据的替换

/\*临界区\*/

struct foo \*p = rcu\_dereference(gp)

if(p != NULL){

do\_something\_with(p->a,p->b)

}

rcu\_read\_unlock()

/\*写操作\*/

struct foo \*p = kmalloc(sizeof(gp),GFP\_KERNEL);

p->a = 1;

p->b = 2;

rcu\_assign\_pointer(gp,p);

读操作中的rcu\_dereference()可以看做订阅了gp，而写操作中的rcu\_assign\_pointer ()可以看做发布了gp，他能保证读操作不会出现读到新的指针但是旧的内存的情况。试想如果不使用rcu\_assign\_pointer(gp,p)，而是直接gp=p,编译器的优化导致可能先执行指针传递(gp=p)再进行指针的赋值(p->a = 1, p->b = 2)。

rcu\_dereference()在arm、x86等架构上其实就是直接赋值，所以直接改为p= gp也没有问题，但是alpha的优化可能会预先读取p->a，再执行p= gp，所以rcu\_dereference()调用的smp\_read\_barrier\_depends()如果架构为alpha加了内存屏障，它要求后面的读操作如果依赖前面的读操作，则前面的读操作需要首先完成，也就是想读取p->a，就必须读取p，想读取p，就需要读取gp。所以使用rcu\_dereference()肯定是没错的。

对于链表而言，linux内核增加了专门的RCU保护的链表操作API：

list\_add\_rcu(struct list\_head \*new, struct list\_head \*head);//new插入RCU保护的链表的开头

list\_add\_tail\_rcu(struct list\_head \*new,struct list\_head \*head); new插入RCU保护的链表的末尾

list\_del\_rcu(struct list\_head \*entry);//从RCU保护的链表中删除指定的链表元素entry

list\_replace\_rcu(struct list\_head \*old,struct list\_head \*new);// RCU保护的链表中new替代oled

list\_for\_each\_entry\_rcu(pos, head, member);//遍历RCU保护的链表

1. struct el{
2. struct list\_head lp;
3. long key;
4. spinlock\_t mutex;
5. int data;
6. }
7. DEFINE\_RWLOCK(listmutex);
8. LIST\_HEAD(head);
9. int search(long key,int \*result)
10. {
11. struct el \*p;
12. read\_lock(&listmutex);
13. list\_for\_each\_entry(p,
14. &head,lp){
15. if(p->key == key){
16. \*result = p->data;
17. read\_unlock(&listmutex);
18. return 1;
19. }
20. }
21. read\_unlock(&listmutex);
22. return 0;
23. }
24. int delete(long key)
25. {
26. struct el \*p;
27. write\_lock(&listmutex);
28. list\_for\_each\_entry(p,
29. &head,lp){
30. if(p->key == key){
31. list\_del(&p->lp);
32. write\_unlock(&listmutex);
33. kfree(p);
34. return 1;
35. }
36. }
37. write\_unlock(&listmutex);
38. return 0;
39. }
40. struct el{
41. struct list\_head lp;
42. long key;
43. spinlock\_t mutex;
44. int data;
45. }
46. DEFINE\_SPINLOCK(listmutex);
47. LIST\_HEAD(head);
48. int search(long key,int \*result)
49. {
50. struct el \*p;
51. rcu\_read\_lock();
52. list\_for\_each\_entry\_rcu(p,
53. &head,lp){
54. if(p->key == key){
55. \*result = p->data;
56. rcu\_read\_unlock();
57. return 1;
58. }
59. }
60. rcu\_read\_unlock();
61. return 0;
62. }
63. int delete(long key)
64. {
65. struct el \*p;
66. spin\_lock(&listmutex);
67. list\_for\_each\_entry\_rcu(p,
68. &head,lp){
69. if(p->key == key){
70. list\_del\_rcu(&p->lp);
71. spin\_unlock(&listmutex);
72. synchronize\_rcu();
73. kfree(p);
74. return 1;
75. }
76. }
77. spin\_unlock(&listmutex);
78. return 0;
79. }

上面是《Linux设备驱动开发详解》中分别使用读写锁和RCU进行链表操作的例子，由上可以看出读写锁可以很方便的升级为RCU。synchronize\_rcu()将阻塞写执行单元，直到所有的RCU读临界操作完成，如果不调用此函数直接kfree(p)，可能导致读操作还在使用p，而写操作已经free，极可能导致系统崩溃。在写操作中spin\_lock()是为了保护链表，如果进程A执行到30行被进程B抢占，进程B同样执行delete，等进程B执行完毕回到进程A，进程A继续执行再删除链表的时候就会导致程序出错。

RCU适用于读操作远远大于写操作的场景，使用它取代读写锁可以获得更高的性能，但是延后的内存释放将占用一些内存，造成额外的内存开销。另外因为RCU读者与写者并行执行，所以RCU读者可能会访问到旧数据，所以对于无法容忍旧数据以及不止一个写者的情况，写者需要自旋锁或者其他锁保证写操作之间的同步。

## 信号量

前面提到自旋锁期间不允许睡眠，而且自旋锁希望占用锁的时间越短越好。信号量是一种睡眠锁。如果有一个任务试图获得一个已经被占用的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器重获自由可以去执行其他代码，当信号量被释放后，处于等待队列的任务被唤醒，并获得信号量。所以信号量与自旋锁的主要区别就是当锁被占用时，信号量会睡眠，自旋锁会忙等。

还是在自旋锁中讲到的房间与钥匙的例子。当某人来到门前拿着钥匙进入房间，另一个人来到门前后发现没有钥匙时，不像自旋锁那样在门口打转，而是把自己的名字写到一个列表中，然后去睡觉。当房间里面的人离开时后，查看一下列表，如果列表上有名字那么就叫醒他，让他进入房间。所以信号量在保证只有一个执行单元进入临界区，而且比自旋锁提供了更好的CPU利用率，因为CPU不会忙等待，但是信号量的开销是很大的，毕竟维护等待队列以及唤醒等操作花费的时间可能比锁占用的时间还要长，那这个时候为什么不用自旋锁呢。

### 信号量的使用

down()操作会将信号量计数值减1，如果结果大于等于0则获得信号量并进入临界区，如果为负数，任务会被放入等待队列。临界区操作完毕后，up()操作会释放信号量，增加信号量计数值。如果此时该信号量的等待队列不为空，那么处于队列中的等待任务会被唤醒并获得信号量。

down\_ interruptible ()试图获得信号量，如果信号量不可用，那么它将进程状态置为TASK\_INTERRUPTIBLE后睡眠，这种状态意味着可被信号唤醒，而down()会将进程状态置为TASK\_UNINTERRUPTIBLE后睡眠，也就是不可被信号打断，这种情况进程就会死等信号量不能被信号打断，所以down\_ interruptible ()更合理和适用。

struct semaphore sem;

sema\_init(&sem,count);//count为信号量初值

if(down\_interruptible(&sem)){

return –ERESTARTSYS;//接收到信号，但是并没有获取到信号量

}

/\*临界区\*/

up(&sem)

信号量方法列表

|  |  |
| --- | --- |
| sema\_init() | 以指定的计数值初始化信号量 |
| down\_interruptible() | 试图获得指定的信号量，如果信号量被争用，进入可中断睡眠状态 |
| down() | 试图获得指定的信号量，如果信号量被争用，进入不可中断睡眠状态 |
| down\_trylock() | 试图获得指定的信号量，如果信号量被争用，则返回非0值 |
| up() | 释放信号量，并睡眠队列不空时唤醒其中一个任务 |

信号量的处理过程如下：



## 互斥体

互斥体是可以睡眠的互斥锁。目前内核倾向于使用mutex作为互斥手段（4.4版本内核中mutex\_init()有2473 个，而sema\_init()只有144个，2.6版本内核中mutex\_init()有1347 个，而sema\_init()只有140个），因为目前更多的都是在使用信号量计数为1的信号量，此时功能与互斥体一致。互斥体使用更加简单，那为什么不顺应潮流呢。

### 互斥体的使用

DEFINE\_MUTEX(mutex)

mutex\_lock(&mutex);

/\*临界区\*/

mutex\_unlock(&mutex);

mutex方法列表：

|  |  |
| --- | --- |
| DEFINE\_MUTEX(mutex) | 定义并初始化mutex |
| mutex\_init(&mutex) | 动态初始化mutex |
| mutex\_lock(&mutex) | 上锁，如果锁不可用则睡眠，不能被信号打断 |
| mutex\_unlock(&mutex) | 解锁 |
| mutex\_lock\_ interruptible(&mutex) | 上锁，如果锁不可用则睡眠，能被信号打断(更加适用) |
| mutex\_trylock(&mutex) | 试图获取锁，获取不到不会引进进程睡眠，成功返回1，失败返回0 |
| mutex\_is\_locked(&mutex) | 如果锁被争用，返回1，否则返回0 |

### 互斥体注意事项

（1）任何时刻只会有一个任务可以持有muxte

（2）给mutex上锁的进程必须负责给其解锁

（3）不能递归的上锁与解锁

（4）mutex不能在中断中使用，因为其可能引起睡眠，睡眠就会引起调度，而中断中禁止调度

### 信号量的用武之地

（1）mutex任何时刻只会有一个任务持有，而信号量支持多个任务持有（计数值大于1）

（2）信号量可用于同步，而mutex无能为力,如下使用信号量就可保证B一定会等待A的信号量，使用mutex无法决定进程A/B的执行顺序

进程A 进程B

--- down(&sem)

up(&sem) ---

### 互斥体和自旋锁的选取

（1）由于争用信号量的进程在等待锁的时候会睡眠，所以信号量适合锁会被长时间持有的情况，相反如果所被持有的时间较短，自旋锁更加试用

（2）由于信号量争用时会睡眠，睡眠会产生内核调度，所以只能在进程上下文中才能获取信号量，因为在中断中时不能进行调度的。

自旋锁与互斥体选择

|  |  |
| --- | --- |
| 需求 | 加锁方法 |
| 低开销加锁 | 优先使用自旋锁 |
| 短期锁定 | 优先使用自旋锁 |
| 长期加锁 | 优先使用互斥体 |
| 中断上下文加锁 | 只能使用自旋锁 |
| 持有锁需要睡眠 | 只能使用互斥体 |

## 完成量

完成量用于两个任务的同步，用于一个任务等待另一个任务执行完某事。回忆一下信号量的用武之地，感觉信号量的作用又被蚕食了一点。

### 完成量使用

完成量方法列表：

|  |  |
| --- | --- |
| DECLARE\_COMPLETION(comp) | 竟态创建完成量并初始化 |
| init\_completion() | 动态初始化完成量 |
| wait\_for\_ completion() | 等待一个完成量被唤醒 |
| completion() | 唤醒完成量，只唤醒一个等待的任务 |
| Completion\_all() | 唤醒完成量，唤醒所有等待此完成量的任务 |

完成量同步流程如下，进程B就会等待进程A的唤醒：

进程A 进程B

--- wait\_for\_ completion(&done)

completion(done) ---

# 总结

## 并发控制方法选择

|  |  |  |
| --- | --- | --- |
| 并发控制方法 | 适用范围 | 注意事项 |
| 原子操作 | 整数或者位操作保护 |  |
| 自旋锁 | 保护多进程或者进程与中断间共享资源 | 死锁；锁期间不能睡眠 |
| 读写自旋锁 | 读写操作能清晰的分开，提高自旋锁性能 | 死锁；锁期间不能睡眠；读比写多很多时，可能导致写饥饿 |
| 顺序锁 | 对读写锁优化，希望写优先于读 | 死锁；锁期间不能睡眠；适用于读多写少时 |
| RCU | 无需担心死锁；读多写少时的优化； | RCU读者可能会访问到旧数据；写者需要自旋锁或者其他锁保证写操作之间的同步 |
| 信号量 | 同步；控制进程顺序 | 死锁；不能用于中断中 |
| 互斥体 | 互斥操作 | 死锁；不能用于中断中 |
| 完成量 | 同步 |  |

## 什么数据需要加锁？

大多数内核数据结构都需要加锁。如果有两个及以上执行单元可以访问同一数据，就需要加锁。如下情况都需要加锁：

（1）全局变量。除了当前线程，其他线程也会访问；

（2）数据在不同进程进程中共享；数据在进程与中断之间共享；

（3）数据在两个不同的中断服务中共享；

（4）进程被抢占，被调度的程序也会访问同一数据；

（5）进程睡眠，被调度的程序也会访问同一数据；

# 示例说明

示例通过更改模块挂载参数来更改并发控制方法，insmod concurrency\_control.ko type=0，type=0时不进行并发控制，type=1时使用自旋锁，type=2时使用互斥体。可以看到不进行并发控制时我们不能得到我们想要的结果，如果使用了自旋锁或者互斥体则能得到想要的结果。