**主要内容：**

* 原子操作
* 自旋锁
* 读写自旋锁
* 信号量
* 读写信号量
* 互斥体
* 完成变量
* 大内核锁
* 顺序锁
* 禁止抢占
* 顺序和屏障
* 总结

### **1. 原子操作**

原子操作是由编译器来保证的，保证一个线程对数据的操作不会被其他线程打断。

原子操作有2类：

原子整数操作，有32位和64位。头文件分别为<asm/atomic.h>和<asm/atomic64.h>

原子位操作。头文件 <asm/bitops.h>

原子操作的api很简单，参见相应的头文件即可。

原子操作头文件与具体的体系结构有关，比如x86架构的相关头文件在 arch/x86/include/asm/\*.h

### **2. 自旋锁**

原子操作只能用于临界区只有一个变量的情况，实际应用中，临界区的情况要复杂的多。

对于复杂的临界区，linux内核中也提供了多种同步方法，自旋锁就是其中一种。

自旋锁的特点就是当一个线程获取了锁之后，其他试图获取这个锁的线程一直在循环等待获取这个锁，直至锁重新可用。

由于线程实在一直循环的获取这个锁，所以会造成CPU处理时间的浪费，因此最好将自旋锁用于能很快处理完的临界区。

自旋锁方法列表如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| spin\_lock() | 获取指定的自旋锁 |
| spin\_lock\_irq() | 禁止本地中断并获取指定的锁 |
| spin\_lock\_irqsave() | 保存本地中断的当前状态，禁止本地中断，并获取指定的锁 |
| spin\_unlock() | 释放指定的锁 |
| spin\_unlock\_irq() | 释放指定的锁，并激活本地中断 |
| spin\_unlock\_irqstore() | 释放指定的锁，并让本地中断恢复到以前状态 |
| spin\_lock\_init() | 动态初始化指定的spinlock\_t |
| spin\_trylock() | 试图获取指定的锁，如果未获取，则返回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 |

### **3. 读写自旋锁**

读写自旋锁除了和普通自旋锁一样有自旋特性以外，还有以下特点：   
 读锁之间是共享的

写锁之间是互斥的

即一个线程持有了写锁之后，其他线程不能以读或者写的方式持有这个锁

读写锁之间是互斥的

即一个线程持有了读锁之后，其他线程不能以写的方式持有这个锁

**注**：读写锁要分别使用，不能混合使用，否则会造成死锁。

正常的使用方法：

DEFINE\_RWLOCK(mr\_rwlock);

read\_lock(&mr\_rwlock);/\* 临界区(只读).... \*/

read\_unlock(&mr\_rwlock);

write\_lock(&mr\_lock);/\* 临界区(读写)... \*/

write\_unlock(&mr\_lock);

混合使用时：

/\* 获取一个读锁 \*/

read\_lock(&mr\_lock);

/\* 在获取写锁的时候，由于读写锁之间是互斥的，

\* 所以写锁会一直自旋等待读锁的释放，

\* 而此时读锁也在等待写锁获取完成后继续下面的代码。

\* 因此造成了读写锁的互相等待，形成了死锁。

\*/

write\_lock(&mr\_lock);

读写锁相关文件参照 各个体系结构中的 <asm/rwlock.h>

读写锁的相关函数如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| read\_lock() | 获取指定的读锁 |
| read\_lock\_irq() | 禁止本地中断并获得指定读锁 |
| read\_lock\_irqsave() | 存储本地中断的当前状态，禁止本地中断并获得指定读锁 |
| read\_unlock() | 释放指定的读锁 |
| read\_unlock\_irq() | 释放指定的读锁并激活本地中断 |
| read\_unlock\_irqrestore() | 释放指定的读锁并将本地中断恢复到指定前的状态 |
| write\_lock() | 获得指定的写锁 |
| write\_lock\_irq() | 禁止本地中断并获得指定写锁 |
| write\_lock\_irqsave() | 存储本地中断的当前状态，禁止本地中断并获得指定写锁 |
| write\_unlock() | 释放指定的写锁 |
| write\_unlock\_irq() | 释放指定的写锁并激活本地中断 |
| write\_unlock\_irqrestore() | 释放指定的写锁并将本地中断恢复到指定前的状态 |
| write\_trylock() | 试图获得指定的写锁；如果写锁不可用，返回非0值 |
| rwlock\_init() | 初始化指定的rwlock\_t |

### **4. 信号量**

信号量也是一种锁，和自旋锁不同的是，线程获取不到信号量的时候，不会像自旋锁一样循环的去试图获取锁，而是进入睡眠，直至有信号量释放出来时，才会唤醒睡眠的线程，进入临界区执行。

由于使用信号量时，线程会睡眠，所以等待的过程不会占用CPU时间。所以信号量适用于等待时间较长的临界区。

信号量相关函数参照: <linux/semaphore.h> 实现方法参照：kernel/semaphore.c

使用信号量的方法如下：

/\* 定义并声明一个信号量，名字为mr\_sem，用于信号量计数 \*/

static DECLARE\_MUTEX(mr\_sem);

/\* 试图获取信号量....， 信号未获取成功时，进入睡眠

\* 此时，线程状态为 TASK\_INTERRUPTIBLE

\*/

/\* 这里也可以用：

\* down(&mr\_sem);

\* 这个方法把线程状态置为 TASK\_UNINTERRUPTIBLE 后睡眠

\* 一般用的比较多的是down\_interruptible()方法，因为以 TASK\_UNINTERRUPTIBLE 方式睡眠无法被信号唤醒。

\*/

down\_interruptible(&mr\_sem);

/\* 临界区 ... \*/

/\* 释放给定的信号量 \*/

up(&mr\_sem);

信号量方法如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| sema\_init(struct semaphore \*, int) | 以指定的计数值初始化动态创建的信号量 |
| init\_MUTEX(struct semaphore \*) | 以计数值1初始化动态创建的信号量 |
| init\_MUTEX\_LOCKED(struct semaphore \*) | 以计数值0初始化动态创建的信号量（初始为加锁状态） |
| down\_interruptible(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则进入可中断睡眠状态 |
| down(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则进入不可中断睡眠状态 |
| down\_trylock(struct semaphore \*) | 以试图获得指定的信号量，如果信号量已被争用，则立即返回非0值 |
| up(struct semaphore \*) | 以释放指定的信号量，如果睡眠队列不空，则唤醒其中一个任务 |

信号量结构体具体如下：

/\* Please don't access any members of this structure directly \*/

struct semaphore {

spinlock\_t lock;

unsigned int count;

struct list\_head wait\_list;

};

可以发现信号量结构体中有个自旋锁，这个自旋锁的作用是保证信号量的down和up等操作不会被中断处理程序打断。

### **5. 读写信号量**

读写信号量和信号量之间的关系 与 读写自旋锁和普通自旋锁之间的关系 差不多。

读写信号量都是二值信号量，即计数值最大为1，增加读者时，计数器不变，增加写者，计数器才减一。

也就是说读写信号量保护的临界区，最多只有一个写者，但可以有多个读者。

读写信号量的相关内容参见：<asm/rwsem.h> 具体实现与硬件体系结构有关。

### **6. 互斥体**

互斥体也是一种可以睡眠的锁，相当于二值信号量，只是提供的API更加简单，使用的场景也更严格一些，如下所示：

1. mutex的计数值只能为1，也就是最多只允许一个线程访问临界区
2. 在同一个上下文中上锁和解锁
3. 不能递归的上锁和解锁
4. 持有个mutex时，进程不能退出
5. mutex不能在中断或者下半部中使用，也就是mutex只能在进程上下文中使用
6. mutex只能通过官方API来管理，不能自己写代码操作它

在面对**互斥体和信号量的选择**时，只要满足互斥体的使用场景就尽量优先使用互斥体。

在面对**互斥体和自旋锁的选择**时，参见下表：

|  |  |
| --- | --- |
| **需求** | **建议的加锁方法** |
| 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 |
| 中断上下文中加锁 | 使用自旋锁 |
| 持有锁需要睡眠 | 使用互斥体 |

互斥体头文件：<linux/mutex.h>

常用的互斥体方法如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| mutex\_lock(struct mutex \*) | 为指定的mutex上锁，如果锁不可用则睡眠 |
| mutex\_unlock(struct mutex \*) | 为指定的mutex解锁 |
| mutex\_trylock(struct mutex \*) | 试图获取指定的mutex，如果成功则返回1；否则锁被获取，返回0 |
| mutex\_is\_locked(struct mutex \*) | 如果锁已被争用，则返回1；否则返回0 |

### **7. 完成变量**

完成变量的机制类似于信号量，

比如一个线程A进入临界区之后，另一个线程B会在完成变量上等待，线程A完成了任务出了临界区之后，使用完成变量来唤醒线程B。

完成变量的头文件：<linux/completion.h>

完成变量的API也很简单：

|  |  |
| --- | --- |
| **方法** | **描述** |
| init\_completion(struct completion \*) | 初始化指定的动态创建的完成变量 |
| wait\_for\_completion(struct completion \*) | 等待指定的完成变量接受信号 |
| complete(struct completion \*) | 发信号唤醒任何等待任务 |

使用完成变量的例子可以参考：kernel/sched.c 和 kernel/fork.c

一般在2个任务需要简单同步的情况下，可以考虑使用完成变量。

### **8. 大内核锁**

大内核锁已经不再使用，只存在与一些遗留的代码中。

### **9. 顺序锁**

顺序锁为读写共享数据提供了一种简单的实现机制。

之前提到的读写自旋锁和读写信号量，在读锁被获取之后，写锁是不能再被获取的，

也就是说，必须等所有的读锁释放后，才能对临界区进行写入操作。

顺序锁则与之不同，读锁被获取的情况下，写锁仍然可以被获取。

使用顺序锁的读操作在读之前和读之后都会检查顺序锁的序列值，如果前后值不符，则说明在读的过程中有写的操作发生，

那么读操作会重新执行一次，直至读前后的序列值是一样的。

do

{

/\* 读之前获取 顺序锁foo 的序列值 \*/

seq = read\_seqbegin(&foo);

...

} while(read\_seqretry(&foo, seq)); /\* 顺序锁foo此时的序列值!=seq 时返回true，反之返回false \*/

顺序锁优先保证写锁的可用，所以适用于那些读者很多，写者很少，且写优于读的场景。

顺序锁的使用例子可以参考：kernel/timer.c和kernel/time/tick-common.c文件

### **10. 禁止抢占**

其实使用自旋锁已经可以防止内核抢占了，但是有时候仅仅需要禁止内核抢占，不需要像自旋锁那样连中断都屏蔽掉。

这时候就需要使用禁止内核抢占的方法了：

|  |  |
| --- | --- |
| **方法** | **描述** |
| preempt\_disable() | 增加抢占计数值，从而禁止内核抢占 |
| preempt\_enable() | 减少抢占计算，并当该值降为0时检查和执行被挂起的需调度的任务 |
| preempt\_enable\_no\_resched() | 激活内核抢占但不再检查任何被挂起的需调度的任务 |
| preempt\_count() | 返回抢占计数 |

这里的preempt\_disable()和preempt\_enable()是可以嵌套调用的，disable和enable的次数最终应该是一样的。

禁止抢占的头文件参见：<linux/preempt.h>

### **11. 顺序和屏障**

对于一段代码，编译器或者处理器在编译和执行时可能会对执行顺序进行一些优化，从而使得代码的执行顺序和我们写的代码有些区别。

一般情况下，这没有什么问题，但是在并发条件下，可能会出现取得的值与预期不一致的情况

比如下面的代码：

/\*

\* 线程A和线程B共享的变量 a和b

\* 初始值 a=1, b=2

\*/

int a = 1, b = 2;

/\*

\* 假设线程A 中对 a和b的操作

\*/

void Thread\_A()

{

a = 5;

b = 4;

}

/\*

\* 假设线程B 中对 a和b的操作

\*/

void Thread\_B()

{

if (b == 4)

printf("a = %d\n", a);

}

由于编译器或者处理器的优化，线程A中的赋值顺序可能是b先赋值后，a才被赋值。

所以如果线程A中 b=4; 执行完，a=5; 还没有执行的时候，线程B开始执行，那么线程B打印的是a的初始值1。

这就与我们预期的不一致了，我们预期的是a在b之前赋值，所以线程B要么不打印内容，如果打印的话，a的值应该是5。

在某些并发情况下，为了保证代码的执行顺序，引入了一系列屏障方法来阻止编译器和处理器的优化。

|  |  |
| --- | --- |
| **方法** | **描述** |
| rmb() | 阻止跨越屏障的载入动作发生重排序 |
| read\_barrier\_depends() | 阻止跨越屏障的具有数据依赖关系的载入动作重排序 |
| wmb() | 阻止跨越屏障的存储动作发生重排序 |
| mb() | 阻止跨越屏障的载入和存储动作重新排序 |
| smp\_rmb() | 在SMP上提供rmb()功能，在UP上提供barrier()功能 |
| smp\_read\_barrier\_depends() | 在SMP上提供read\_barrier\_depends()功能，在UP上提供barrier()功能 |
| smp\_wmb() | 在SMP上提供wmb()功能，在UP上提供barrier()功能 |
| smp\_mb() | 在SMP上提供mb()功能，在UP上提供barrier()功能 |
| barrier() | 阻止编译器跨越屏障对载入或存储操作进行优化 |

为了使得上面的小例子能正确执行，用上表中的函数修改线程A的函数即可：

/\*

\* 假设线程A 中对 a和b的操作

\*/

void Thread\_A()

{

a = 5;

mb();

/\*

\* mb()保证在对b进行载入和存储值(值就是4)的操作之前

\* mb()代码之前的所有载入和存储值的操作全部完成(即 a = 5;已经完成)

\* 只要保证a的赋值在b的赋值之前进行，那么线程B的执行结果就和预期一样了

\*/

b = 4;

}

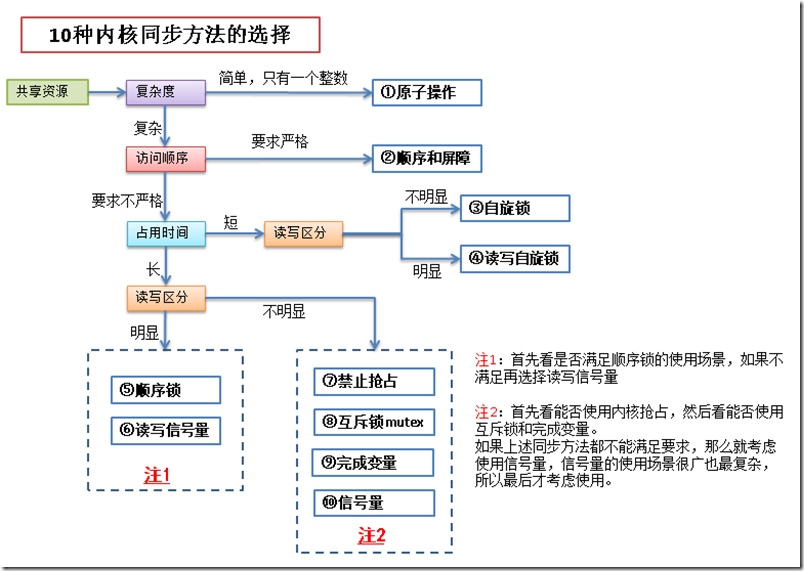
### **12. 总结**

本节讨论了大约11种内核同步方法，除了大内核锁已经不再推荐使用之外，其他各种锁都有其适用的场景。

了解了各种同步方法的适用场景，才能正确的使用它们，使我们的代码在安全的保障下达到最优的性能。

同步的目的就是为了保障数据的安全，其实就是保障各个线程之间共享资源的安全，下面根据共享资源的情况来讨论一下10种同步方法的选择。

10种同步方法在图中分别用蓝色框标出。

[](https://images0.cnblogs.com/blog/83005/201305/01111554-c232cb4333a446fca92bbe6c03077863.png)