本文详细的介绍了 Linux 内核中的同步机制：原子操作、信号量、读写信号量和自旋锁的 API，使用要求以及一些典型示例

## 一、引言

在现代操作系统里，同一时间可能有多个内核执行流在执行，因此内核其实象多进程多线程编程一样也需要一些同步机制来同步各执行单元对共享数据的访问。尤其是在多处理器系统上，更需要一些同步机制来同步不同处理器上的执行单元对共享的数据的访问。

在主流的 Linux 内核中包含了几乎所有现代的操作系统具有的同步机制，这些同步机制包括：原子操作、信号量（semaphore）、读写信号量（rw_semaphore）、spinlock、BKL(Big Kernel Lock)、rwlock、brlock（只包含在 2.4 内核中）、RCU（只包含在 2.6 内核中）和 seqlock（只包含在 2.6 内核中）。

## 二、原子操作

所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就说，它的最小的执行单位，不可能有比它更小的执行单位，因此这里的原子实际是使用了物理学里的物质微粒的概念。

原子操作需要硬件的支持，因此是架构相关的，其 API 和原子类型的定义都定义在内核源码树的 include/asm/atomic.h 文件中，它们都使用汇编语言实现，因为 C 语言并不能实现这样的操作。

原子操作主要用于实现资源计数，很多引用计数(refcnt)就是通过原子操作实现的。原子类型定义如下：

```
typedef struct
        {
volatile int counter;
        }
    atomic t;
```

volatile 修饰字段告诉 gcc 不要对该类型的数据做优化处理，对它的访问都是对内存的访问，而不是对寄存器的访问。

原子操作 API 包括：

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

该函数对原子类型的变量进行原子读操作，它返回原子类型的变量 v 的值。

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

该函数设置原子类型的变量 v 的值为 i。

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

该函数给原子类型的变量 v 增加值 i。

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

该函数从原子类型的变量 v 中减去 i。

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

该函数从原子类型的变量 v 中减去 i，并判断结果是否为 0，如果为 0，返回真，否则返回假。

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

该函数对原子类型变量 v 原子地增加 1。

```
void atomic dec(atomic t *v);
```

该函数对原子类型的变量 v 原子地减 1。

```
int atomic_dec_and_test(atomic_t *v);
```

该函数对原子类型的变量 v 原子地减 1，并判断结果是否为 0，如果为 0，返回真，否则返回假。

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

该函数对原子类型的变量 v 原子地增加 1，并判断结果是否为 0，如果为 0，返回真，否则返回假。

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

该函数对原子类型的变量 v 原子地增加 I，并判断结果是否为负数，如果是，返回真，否则返回假。

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

该函数对原子类型的变量 v 原子地增加 i，并且返回指向 v 的指针。

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

该函数从原子类型的变量 v 中减去 i，并且返回指向 v 的指针。

```
int atomic_inc_return(atomic_t * v);
```

该函数对原子类型的变量 v 原子地增加 1 并且返回指向 v 的指针。

```
int atomic_dec_return(atomic_t * v);
```

该函数对原子类型的变量 v 原子地减 1 并且返回指向 v 的指针。

原子操作通常用于实现资源的引用计数，在 TCP/IP 协议栈的 IP 碎片处理中，就使用了引用计数，碎片队列结构 struct ipq 描述了一个 IP 碎片，字段 refcnt 就是引用计数器，它的类型为 atomic_t，当创建 IP 碎片时（在函数 ip_frag_create 中），使用 atomic_set 函数把它设置为 1，当引用该 IP 碎片

时，就使用函数 atomic_inc 把引用计数加 1。

当不需要引用该 IP 碎片时，就使用函数 ipq_put 来释放该 IP 碎片，ipq_put 使用函数 atomic_dec_and_test 把引用计数减 1 并判断引用计数是否为 0，如果是就释放 IP 碎片。函数 ipq_kill 把 IP 碎片从 ipq 队列中删除，并把该删除的 IP 碎片的引用计数减 1（通过使用函数 atomic_dec 实现）。

## 三、信号量（semaphore）

Linux 内核的信号量在概念和原理上与用户态的 System V 的 IPC 机制信号量是一样的，但是它绝不可能在内核之外使用，因此它与 System V 的 IPC 机制信号量毫不相干。

信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初始值为 1 就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。

一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减 1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。

当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加 1 实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

信号量的 API 有：

```
DECLARE_MUTEX(name)
```

该宏声明一个信号量 name 并初始化它的值为 0，即声明一个互斥锁。

```
DECLARE_MUTEX_LOCKED(name)
```

该宏声明一个互斥锁 name，但把它的初始值设置为 0，即锁在创建时就处在已锁状态。因此对于这种锁，一般是先释放后获得。

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

该函用于数初始化设置信号量的初值，它设置信号量 sem 的值为 val。

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

该函数用于初始化一个互斥锁，即它把信号量 sem 的值设置为 1。

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

该函数也用于初始化一个互斥锁，但它把信号量 sem 的值设置为 0，即一开始就处在已锁状态。

```
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（包括 IRQ 上下文和 softirq 上下文）使用该函数。该函数将把 sem 的值减 1，如果信号量 sem 的值非负，就直接返回，否则调用者将被挂起，直到别的任务释放该信号量才能继续运行。

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

该函数功能与 down 类似，不同之处为，down 不会被信号（signal）打断，但 down_interruptible 能被信号打断，因此该函数有返回值来区分是正常返回还是被信号中断，如果返回 0，表示获得信号量正常返回，如果被信号打断，返回-EINTR。

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

该函数试着获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，表示不能获得信号量 sem，返回值为非 0 值。因此，它不会导致调用者睡眠，可以在中断上下文使用。

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

该函数释放信号量 sem，即把 sem 的值加 1，如果 sem 的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者。

信号量在绝大部分情况下作为互斥锁使用，下面以 console 驱动系统为例说明信号量的使用。

在内核源码树的 kernel/printk.c 中，使用宏 DECLARE_MUTEX 声明了一个互斥锁 console_sem，它用于保护 console 驱动列表 console_drivers 以及同步对整个 console 驱动系统的访问。

其中定义了函数 acquire_console_sem 来获得互斥锁 console_sem，定义了 release_console_sem 来释放互斥锁 console_sem，定义了函数 try_acquire_console_sem 来尽力得到互斥锁 console_sem。这三个函数实际上是分别对函数 down，up 和 down_trylock 的简单包装。

需要访问 console_drivers 驱动列表时就需要使用 acquire_console_sem 来保护 console_drivers 列表，当访问完该列表后，就调用 release_console_sem 释放信号量 console_sem。

函数 console_unblank，console_device，console_stop，console_start，register_console 和 unregister_console 都需要访问 console_drivers，因此它们都使用函数对 acquire_console_sem 和 release_console_sem 来对 console_drivers 进行保护。

**四、读写信号量（rw_semaphore）**

读写信号量对访问者进行了细分，或者为读者，或者为写者，读者在保持读写信号量期间只能对该读写信号量保护的共享资源进行读访问，如果一个任务除了需要读，可能还需要写，那么它必须被归类为写者，它在对共享资源访问之前必须先获得写者身份，写者在发现自己不需要写访问的情况下可以降级为读者。读写信号量同时拥有的读者数不受限制，也就说可以有任意多个读者同时拥有一个读写信号量。

如果一个读写信号量当前没有被写者拥有并且也没有写者等待读者释放信号量，那么任何读者都可以成功获得该读写信号量；否则，读者必须被挂起直到写者释放该信号量。如果一个读写信号量当前没有被读者或写者拥有并且也没有写者等待该信号量，那么一个写者可以成功获得该读写信号量，否则写者将被挂起，直到没有任何访问者。因此，写者是排他性的，独占性的。

读写信号量有两种实现，一种是通用的，不依赖于硬件架构，因此，增加新的架构不需要重新实现它，但缺点是性能低，获得和释放读写信号量的开销大；另一种是架构相关的，因此性能高，获取

和释放读写信号量的开销小，但增加新的架构需要重新实现。在内核配置时，可以通过选项去控制使用哪一种实现。

读写信号量的相关 API 有：

```
DECLARE_RWSEM(name)
```

该宏声明一个读写信号量 name 并对其进行初始化。

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

该函数对读写信号量 sem 进行初始化。

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

读者调用该函数来得到读写信号量 sem。该函数会导致调用者睡眠，因此只能在进程上下文使用。

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

该函数类似于 down_read，只是它不会导致调用者睡眠。它尽力得到读写信号量 sem，如果能够立即得到，它就得到该读写信号量，并且返回 1，否则表示不能立刻得到该信号量，返回 0。因此，它也可以在中断上下文使用。

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

写者使用该函数来得到读写信号量 sem，它也会导致调用者睡眠，因此只能在进程上下文使用。

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

该函数类似于 down_write，只是它不会导致调用者睡眠。该函数尽力得到读写信号量，如果能够立刻获得，就获得该读写信号量并且返回 1，否则表示无法立刻获得，返回 0。它可以在中断上下文使用。

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

读者使用该函数释放读写信号量 sem。它与 down_read 或 down_read_trylock 配对使用。如果 down_read_trylock 返回 0，不需要调用 up_read 来释放读写信号量，因为根本就没有获得信号量。

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

写者调用该函数释放信号量 sem。它与 down_write 或 down_write_trylock 配对使用。如果 down_write_trylock 返回 0，不需要调用 up_write，因为返回 0 表示没有获得该读写信号量。

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

该函数用于把写者降级为读者，这有时是必要的。因为写者是排他性的，因此在写者保持读写信号量期间，任何读者或写者都将无法访问该读写信号量保护的共享资源，对于那些当前条件下不需要写访问的写者，降级为读者将，使得等待访问的读者能够立刻访问，从而增加了并发性，提高了效率。

读写信号量适于在读多写少的情况下使用，在 linux 内核中对进程的内存映像描述结构的访问就使用了读写信号量进行保护。

在 Linux 中，每一个进程都用一个类型为 task_t 或 struct task_struct 的结构来描述，该结构的类型为 struct mm_struct 的字段 mm 描述了进程的内存映像，特别是 mm_struct 结构的 mmap 字段维护了整个进程的内存块列表，该列表将在进程生存期间被大量地遍利或修改。

因此 mm_struct 结构就有一个字段 mmap_sem 来对 mmap 的访问进行保护，mmap_sem 就是一个读写信号量，在 proc 文件系统里有很多进程内存使用情况的接口，通过它们能够查看某一进程的内存使用情况，命令 free、ps 和 top 都是通过 proc 来得到内存使用信息的，proc 接口就使用 down_read 和 up_read 来读取进程的 mmap 信息。

当进程动态地分配或释放内存时，需要修改 mmap 来反映分配或释放后的内存映像，因此动态内存分配或释放操作需要以写者身份获得读写信号量 mmap_sem 来对 mmap 进行更新。系统调用 brk 和 munmap 就使用了 down_write 和 up_write 来保护对 mmap 的访问。

**五、自旋锁（spinlock）**

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保

持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用（_trylock 的变种能够在中断上下文使用），而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。

如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共巷资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或 SMP 的情况下才真正需要，在单 CPU 且不可抢占的内核下，自旋锁的所有操作都是空操作。

跟互斥锁一样，一个执行单元要想访问被自旋锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。

无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。

自旋锁的 API 有：

```
spin_lock_init(x)
```

该宏用于初始化自旋锁 x。自旋锁在真正使用前必须先初始化。该宏用于动态初始化。

```
DEFINE_SPINLOCK(x)
```

该宏声明一个自旋锁 x 并初始化它。该宏在 2.6.11 中第一次被定义，在先前的内核中并没有该宏。

```
SPIN_LOCK_UNLOCKED
```

该宏用于静态初始化一个自旋锁。

```
DEFINE_SPINLOCK(x)等同于 spinlock_t x =
SPIN_LOCK_UNLOCKED
spin_is_locked(x)
```

该宏用于判断自旋锁 x 是否已经被某执行单元保持（即被锁），如果是，返回真，否则返回假。

```
spin_unlock_wait(x)
```

该宏用于等待自旋锁 x 变得没有被任何执行单元保持，如果没有任何执行单元保持该自旋锁，该宏立即返回，否则将循环在那里，直到该自旋锁被保持者释放。

```
spin_trylock(lock)
```

该宏尽力获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待 lock 被释放。

```
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，这时，它获得锁并返回。总之，只有它获得锁才返回。

```
spin_lock_irqsave(lock, flags)
```

该宏获得自旋锁的同时把标志寄存器的值保存到变量 flags 中并失效本地中断。

```
spin_lock_irq(lock)
```

该宏类似于 spin_lock_irqsave，只是该宏不保存标志寄存器的值。

```
spin_lock_bh(lock)
```

该宏在得到自旋锁的同时失效本地软中断。

```
spin_unlock(lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。如果 spin_trylock 返回假，表明没有获得自旋锁，因此不必使用 spin_unlock 释放。

```
spin_unlock_irqrestore(lock, flags)
```

该宏释放自旋锁 lock 的同时，也恢复标志寄存器的值为变量 flags 保存的值。它与 spin_lock_irqsave 配对使用。

```
spin_unlock_irq(lock)
```

该宏释放自旋锁 lock 的同时，也使能本地中断。它与 spin_lock_irq 配对应用。

```
spin_unlock_bh(lock)
```

该宏释放自旋锁 lock 的同时，也使能本地的软中断。它与 spin_lock_bh 配对使用。

```
spin_trylock_irqsave(lock, flags)
```

该宏如果获得自旋锁 lock，它也将保存标志寄存器的值到变量 flags 中，并且失效本地中断，如果没有获得锁，它什么也不做。

因此如果能够立即获得锁，它等同于 spin_lock_irqsave，如果不能获得锁，它等同于 spin_trylock。如果该宏获得自旋锁 lock，那需要使用 spin_unlock_irqrestore 来释放。

```
spin_trylock_irq(lock)
```

该宏类似于 spin_trylock_irqsave，只是该宏不保存标志寄存器。如果该宏获得自旋锁 lock，需要使用 spin_unlock_irq 来释放。

```
spin_trylock_bh(lock)
```

该宏如果获得了自旋锁，它也将失效本地软中断。如果得不到锁，它什么也不做。因此，如果得到了锁，它等同于 spin_lock_bh，如果得不到锁，它等同于 spin_trylock。如果该宏得到了自旋锁，需要使用 spin_unlock_bh 来释放。

```
spin_can_lock(lock)
```

该宏用于判断自旋锁 lock 是否能够被锁，它实际是 spin_is_locked 取反。如果 lock 没有被锁，它返回真，否则，返回假。该宏在 2.6.11 中第一次被定义，在先前的内核中并没有该宏。

获得自旋锁和释放自旋锁有好几个版本，因此让读者知道在什么样的情况下使用什么版本的获得和释放锁的宏是非常必要的。

如果被保护的共享资源只在进程上下文访问和软中断上下文访问，那么当在进程上下文访问共享资源时，可能被软中断打断，从而可能进入软中断上下文来对被保护的共享资源访问，因此对于这种情况，对共享资源的访问必须使用 spin_lock_bh 和 spin_unlock_bh 来保护。

当然使用 spin_lock_irq 和 spin_unlock_irq 以及 spin_lock_irqsave 和 spin_unlock_irqrestore 也可以，它们失效了本地硬中断，失效硬中断隐式地也失效了软中断。但是使用 spin_lock_bh 和 spin_unlock_bh 是最恰当的，它比其他两个快。

如果被保护的共享资源只在进程上下文和 tasklet 或 timer 上下文访问，那么应该使用与上面情况相同的获得和释放锁的宏，因为 tasklet 和 timer 是用软中断实现的。

如果被保护的共享资源只在一个 tasklet 或 timer 上下文访问，那么不需要任何自旋锁保护，因为同一个 tasklet 或 timer 只能在一个 CPU 上运行，即使是在 SMP 环境下也是如此。实际上 tasklet 在调用 tasklet_schedule 标记其需要被调度时已经把该 tasklet 绑定到当前 CPU，因此同一个 tasklet 决不可能同时在其他 CPU 上运行。

timer 也是在其被使用 add_timer 添加到 timer 队列中时已经被帮定到当前 CPU，所以同一个 timer 绝不可能运行在其他 CPU 上。当然同一个 tasklet 有两个实例同时运行在同一个 CPU 就更不可能了。

如果被保护的共享资源只在两个或多个 tasklet 或 timer 上下文访问，那么对共享资源的访问仅需

要用 spin_lock 和 spin_unlock 来保护，不必使用_bh 版本，因为当 tasklet 或 timer 运行时，不可能有其他 tasklet 或 timer 在当前 CPU 上运行。

如果被保护的共享资源只在一个软中断（tasklet 和 timer 除外）上下文访问，那么这个共享资源需要用 spin_lock 和 spin_unlock 来保护，因为同样的软中断可以同时在不同的 CPU 上运行。

如果被保护的共享资源在两个或多个软中断上下文访问，那么这个共享资源当然更需要用 spin_lock 和 spin_unlock 来保护，不同的软中断能够同时在不同的 CPU 上运行。

如果被保护的共享资源在软中断（包括 tasklet 和 timer）或进程上下文和硬中断上下文访问，那么在软中断或进程上下文访问期间，可能被硬中断打断，从而进入硬中断上下文对共享资源进行访问，因此，在进程或软中断上下文需要使用 spin_lock_irq 和 spin_unlock_irq 来保护对共享资源的访问。

而在中断处理句柄中使用什么版本，需依情况而定，如果只有一个中断处理句柄访问该共享资源，那么在中断处理句柄中仅需要 spin_lock 和 spin_unlock 来保护对共享资源的访问就可以了。

因为在执行中断处理句柄期间，不可能被同一 CPU 上的软中断或进程打断。但是如果有不同的中断处理句柄访问该共享资源，那么需要在中断处理句柄中使用 spin_lock_irq 和 spin_unlock_irq 来保护对共享资源的访问。

在使用 spin_lock_irq 和 spin_unlock_irq 的情况下，完全可以用 spin_lock_irqsave 和 spin_unlock_irqrestore 取代，那具体应该使用哪一个也需要依情况而定，如果可以确信在对共享资源访问前中断是使能的，那么使用 spin_lock_irq 更好一些。

因为它比 spin_lock_irqsave 要快一些，但是如果你不能确定是否中断使能，那么使用 spin_lock_irqsave 和 spin_unlock_irqrestore 更好，因为它将恢复访问共享资源前的中断标志而不是直接使能中断。

当然，有些情况下需要在访问共享资源时必须中断失效，而访问完后必须中断使能，这样的情形使用 spin_lock_irq 和 spin_unlock_irq 最好。

需要特别提醒读者，spin_lock 用于阻止在不同 CPU 上的执行单元对共享资源的同时访问以及不

同进程上下文互相抢占导致的对共享资源的非同步访问，而中断失效和软中断失效却是为了阻止在同一 CPU 上软中断或中断对共享资源的非同步访问。