[引言](#8155-1595230291402)

[原子操作](#8845-1595230291427)

[自旋锁（spinlock)](#8629-1595230291494)

[信号量（Semaphore）](#5016-1595230291660)

[互斥锁（Mutex）](#8683-1595230291750)

[读写信号量（RW\_Semaphore）](#8419-1595230291868)

[读写锁（rw\_lock） 读写自旋锁](#9077-1595230291967)

[顺序锁（seqlock）](#2690-1595230292025)

[RCU（Read-Copy Update）](#2680-1595230292125)

[同步机制之间的比较](#2763-1595230292275)

**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，因此内核其实像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内核中包含了如下这些同步机制包括：

原子操作

信号量（semaphore）

读写信号量（rw\_semaphore）

Spinlock

Mutex

BKL(Big Kernel Lock，只包含在2.4内核中，不讲)

Rwlock

brlock（只包含在2.4内核中，不讲）

RCU（只包含在2.6内核及以后的版本中）

seqlock（只包含在2.6内核及以后的版本中）

**原子操作**

原子操作的概念来源于物理概念中的原子定义，指执行结束前不可分割（即不可打断）的操作，是最小的执行单位。

原子操作与硬件架构强相关，其API具体的定义均位于对应arch目录下的include/asm/atomic.h文件中，通过汇编语言实现，内核源码根目录下的include/asm-generic/atomic.h则抽象封装了API，该API最后分派的实现来自于arch目录下对应的代码。

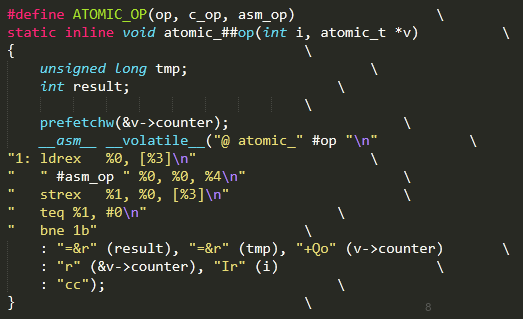
**Structure Definition**

typedef struct { int counter;} atomic\_t;

原子操作主要用于实现资源计数， 许多引用计数(refcnt)就是通过原子操作实现，例如TCP/IP协议栈的IP碎片中，struct ipq中的refcnt字段，类型即为atomic\_t。

**atomic\_add**

原子操作的实现比较简单，以下为例。



**API**

原子操作的API包括如下, 以arm平台为例：

|  |  |  |
| --- | --- | --- |
| 1 | *int atomic\_read(atomic\_t \* v)* | 读操作 |
| 2 | void atomic\_set(atomic\_t\*v,int i) | 设置变量 |
| 3 | void atomic\_add(int i, atomic\_t\*v) | 增加i |
| 4 | void atomic\_sub(int i, atomic\_t\*v) | 减少i |
| 5 | void atomic\_inc(atomic\_t\*v) | 增加1 |
| 6 | void atomic\_dec(atomic\_t\*v) | 减少1 |
| 7 | bool atomic\_inc\_and\_test(atomic\_t\*v) | 加1是否为0 |
| 8 | bool atomic\_dec\_and\_test(atomic\_t\*v) | 减1是否为0 |
| 9 | bool atomic\_add\_negative(int i, atomic\_t\*v) | 加i是否为负 |
| 10 | int atomic\_add\_return(int i, atomic\_t \*v) | 增加i 返回结果，return增加了smp\_mb内存屏障保障数据更新 |
|  | atomic\_add\_return\_acquire | Acquire 该指令先执行，其他能observer到的后执行前面加 smp\_mb |
|  | atomic\_add\_return\_release | 其他observer的先执行，Release 改指令后执行， 指令后smp\_mb |

**自旋锁（spinlock)**

排队锁(拿锁前面都有smp\_mb的内存屏障保护)

A拿锁：排队拿票上厕所拿到的是next，同时next++， owner等于next后持锁成功(第一次next和owner 都是0 直接进去)

B 拿锁：next号码1， next++下一个号码2， owner为0 ，进入WFE低功耗模式等待...

A 去锁: 出门检票owner++，owner为1 与B next相等 ，B拿锁

.....

自旋锁是这样一种同步机制：若自旋锁已被别的执行者保持，调用者就会原地循环等待并检查该锁的持有者是否已经释放锁（即进入自旋状态），若释放则调用者开始持有该锁。***自旋锁持有期间不可被抢占。***

**Structure Definition**

typedef struct { arch\_spinlock\_t raw\_lock; } raw\_spinlock\_t;

typedef struct { union {struct raw\_spinlock rlock;} } spinlock\_t;

typedef struct {

union {

u32 slock;

struct \_\_raw\_tickets {

#ifdef \_\_ARMEB\_\_ //根据字节顺序不同设置安排变量内存布局

u16 next; u16 owner;

#else

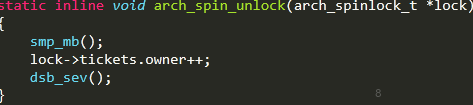
u16 owner; u16 next;

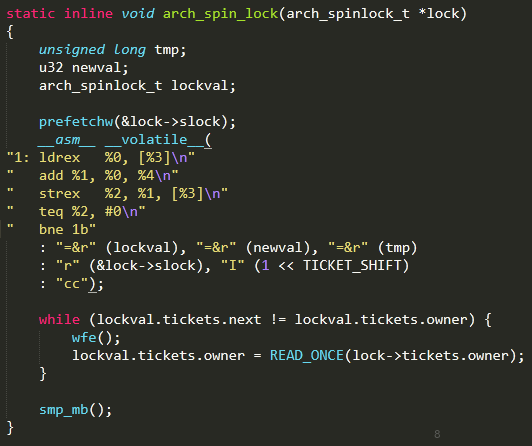
#endif

}; }; } arch\_spinlock\_t;

从定义出发， spinlock根本的实现依赖于具体架构实现中slock这个变量，由于spin\_lock是最多locking机制的基础，我们看一看它的实现。

**Lock & Unlock**





由于slock与tickets共享同一块内存(union)，slock 占32位4字节，tickets内部变量next与owner各16位2字节。以大端序为例，slock 高2字节与next共享，低2字节与owner共享，因此arch\_spin\_lock实际上是将tickets.next+1。假设初始时next与owner皆为0，此时next与owner不等，通过wfe指令进入一小段时间等待状态，而后读取新的owner值检查与next是否相等，不等则继续等待，相等则结束等待。

而owner的值由arch\_spin\_unlock控制，即unlock控制何时结束等待。

**Spin\_lock basic API**

|  |  |  |
| --- | --- | --- |
| 1 | spin\_lock\_init | 动态初始化spin\_lock |
| 2 | DEFINE\_SPINLOCK | 声明并初始化spin\_lock |
| 3 | SPIN\_LOCK\_UNLOCKED | 静态初始化spin\_lock |
| 4 | spin\_lock | 获得一个spin\_lock，拿锁前后都包含smp\_mb动作 |
| 5 | spin\_unlock | 释放一个spin\_lock |
| 6 | spin\_trylock | 尝试获得锁，不能则立即返回，不自旋 |
| 7 | spin\_is\_locked | 判断自旋锁是否已被持有 |
| 8 | spin\_can\_lock | 判断自旋锁是否可被持有 |

**Spin\_lock API & irq**

|  |  |  |
| --- | --- | --- |
| 1 | spin\_lock\_irqsave（lock,flags） /spin\_unlock\_irqrestore（lock, flags） | 获得锁时保存标志寄存器至flags变量中，关中断，释放锁反之 |
| 2 | spin\_lock\_irq（lock） /spin\_unlock\_irq（lock） | 获得锁时，关中断，释放锁反之 |
| 3 | spin\_lock\_bh/spin\_unlock\_bh | 获得锁时，关软中断，释放锁反之 |
| 4 | Spin\_trylock\_irqsave | 能够获得锁时保存标志寄存器至flags变量，关中断，否则立刻返回 |
| 5 | spin\_trylock\_irq | 能够获得锁时，关中断，否则立刻返回 |
| 6 | spin\_trylock\_bh | 能够获得锁时，关软中断，否则立刻返回 |

性能上，spin\_lock > spin\_lock\_bh > spin\_lock\_irq > spin\_lock\_irqsave。

安全上，spin\_lock\_irqsave > spin\_lock\_irq > spin\_lock\_bh >spin\_lock。

**Spin\_lock 不同版本的使用**

**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（spin\_lock\_irq）和软中断失效(spin\_lock\_bh)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。**

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，对共享资源的访问最好使用spin\_lock\_bh和spin\_unlock\_bh来保护。

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。

*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。*

**信号量（Semaphore）**

资源数量多时候使用， 单个资源优先mutex，Sem可被外部打断， waitlist中可以timeout移除队例

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，但是它不可能在内核之外使用，因此它与System V的IPC机制信号量完全不同。

信号量是这样一种同步机制：信号量在创建时设置一个初始值count，表示同时可有count个任务可以访问该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，获取信号量的操作为count-1，若当前count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；若当前count为非负数，表示可获得信号量，因而可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count+1实现，如果count为非正数，表明有任务等待，它也唤醒所有等待该信号量的任务。

**Structure Definition**

struct semaphore {

raw\_spinlock\_t lock; // 利用自旋锁同步

unsigned int count; // 用于资源计数

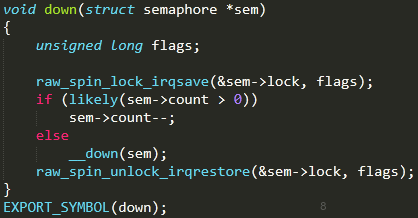
struct list\_head wait\_list; // 等待队列

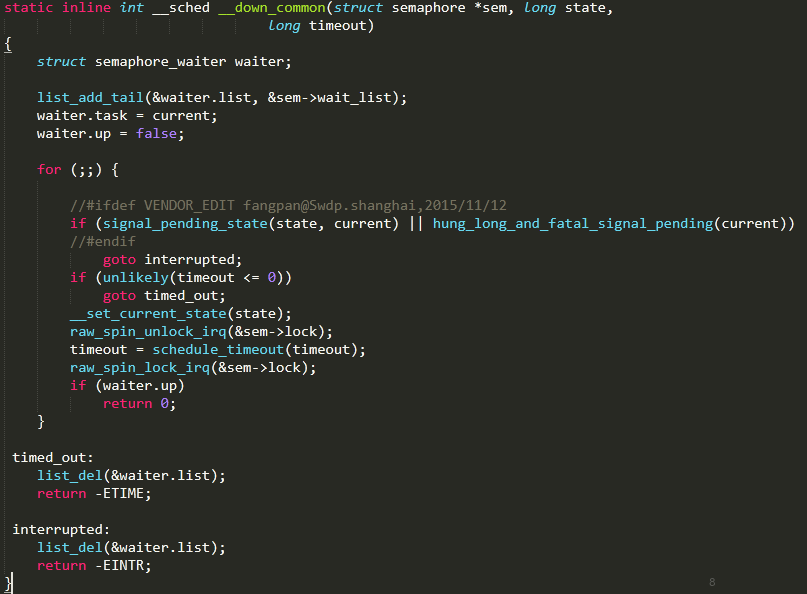
};

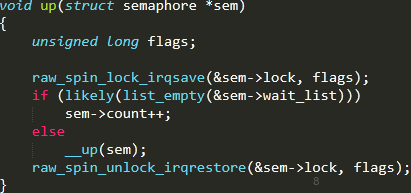
可以发现，信号量是基于spinlock实现的，对其封装以满足高级的功能，例如全局共享资源的配置，并通过等待队列较为灵活的调度。信号量与接下来要讲的mutex都建立在自旋锁实现的执行同步上。

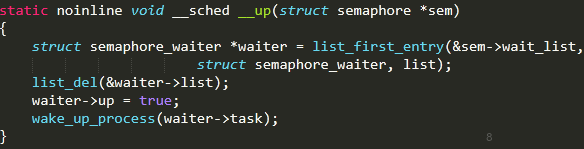
了解了信号量的结构与定义，我们来看看最核心的两个实现down ，up。

**down & up**









**Semaphore API**

|  |  |  |
| --- | --- | --- |
| 1 | DEFINE\_SEMAPHORE(name) | 声明信号量并初始化为1 |
| 2 | void sema\_init (struct semaphore \*sem, int val) | 声明信号量并初始化为val |
| 3 | down | 获得信号量，task不可被中断,除非是致命信号 |
| 4 | down\_interruptible | 获得信号量，task可被中断 |
| 5 | down\_trylock | 能够获得信号量时，coun--，否则立刻返回，不加入waitlist |
| 6 | down\_killable | 获得信号量，task可被kill |
| 7 | up | 释放信号量 |

**互斥锁（Mutex）**

Linux 内核互斥锁是非常常用的同步机制，互斥锁是这样一种同步机制：在互斥锁中同时只能有一个任务可以访问该锁保护的共享资源，且释放锁和获得锁的调用方必须一致。因此在互斥锁中，除了对锁本身进行同步，对调用方（或称持有者）必须也进行同步。当互斥锁无法获得时，task会加入等待队列，直至可获得锁为止。

**Structure Definition**

struct mutex {

atomic\_long\_t owner; // 互斥锁的持有者

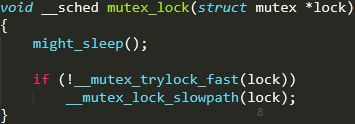
spinlock\_t wait\_lock; // 利用自旋锁同步

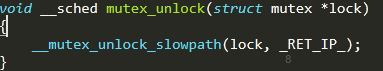
struct list\_head wait\_list; // 等待队列

};

互斥锁从结构上看与信号量十分类似，但将原本的int类型的count计数，改成了atomic\_long\_t的owner以便同步，保证释放者与持有者一致。

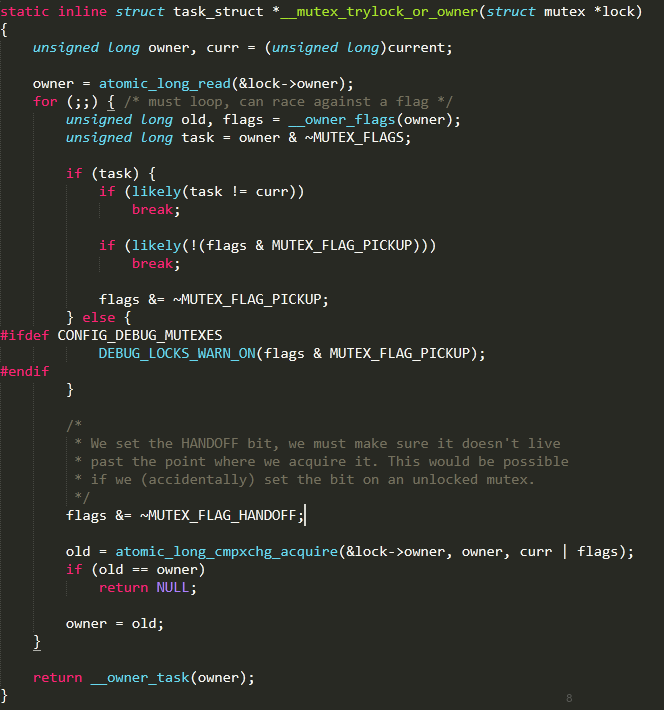
**mutex\_lock & mutex\_unlock**

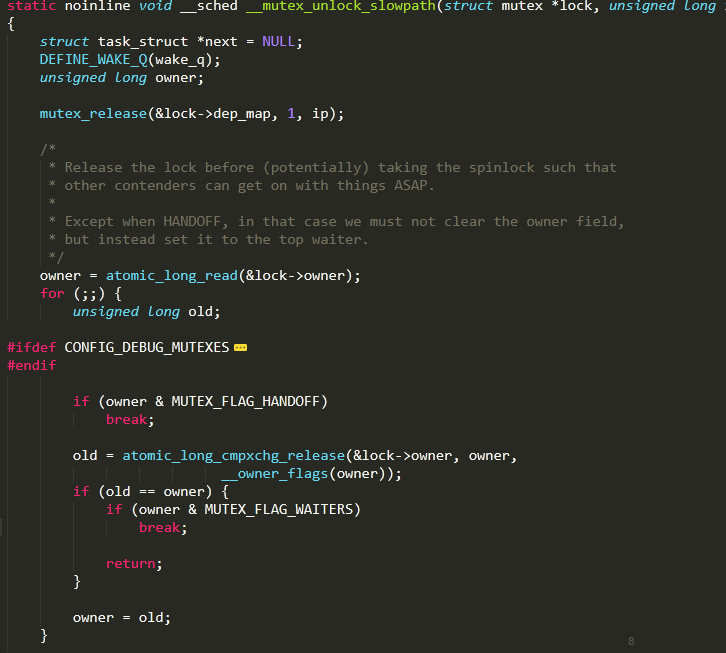




注意might\_sleep指在之后的代码执行中可能会sleep。

由于mutex实现的具体步骤相当复杂，这里选讲比较核心简单的两块。Mutex有关等待队列的处理之后在附录里细讲。





注：当这里的owner实际上是task\_struct的指针，也就是地址，由于task\_struct的地址L1\_cache对齐，因此实际上指针地址后三位为0，因此linux内核利用这三个比特位用于设置mutex的标志位，不影响指针地址的表示也更高效利用了冗余的比特位。

**Mutex 的改进**

最初的互斥锁仅支持睡眠等待，然而经过漫长时间的改进，如今的互斥锁已经可以支持自旋等待，通过MCS锁机制实现。在内核中可以选择配置以支持，CONFIG\_MUTEX\_SPIN\_ON\_OWNER。

struct mutex {

atomic\_long\_t owner; // 互斥锁的持有者

spinlock\_t wait\_lock; // 利用自旋锁同步

#ifdef CONFIG\_MUTEX\_SPIN\_ON\_OWNER

struct optimistic\_spin\_queue osq; /\* Spinner MCS lock \*/

#endif

struct list\_head wait\_list; // 等待队列

};

如上是4.9内核中mutex中常用有效的字段，目前最常用的算法是OSQ算法。自旋等待机制的核心原理是当发现持有者正在临界区执行并且没有其他优先级高的进程要被调度（need\_resched）时，那么mutex当前所在进程认为该持有者很快会离开临界区并释放锁，此时mutex选择自旋等待，短时间的自旋等待显然比睡眠-唤醒开销小一些。

在实现上Mutex保证了同一时间只有一个进程自旋等待持有者释放锁。

MCS 的实现较为复杂，具体可参考一些内核书籍。MCS机制保证了不会存在多个cpu争用锁的情况，从而避免了多个CPU的cacheline颠簸从而降低系统性能的问题。经过改进后，mutex的性能有了相当大的提高，相对信号量的实现要高效得多。因此我们尽量选用mutex。

**Mutex 的使用条件**

Mutex虽然高效，灵活，但存在若干限制条件，需要牢记:

同一时刻只有一条内核路径可以持有锁

只有锁持有者可以解锁

不允许递归加锁解锁

进程持有mutex时不可退出

Mutex 可能导致睡眠阻塞，不可用于中断处理与下半部使用

**Mutex API**

|  |  |  |
| --- | --- | --- |
| 1 | DEFINE\_MUTEX(name) | 静态声明互斥量并初始化解锁状态 |
| 2 | mutex\_init (mutex） | 动态声明互斥量并初始化解锁状态 |
| 3 | void mutex\_destroy(struct mutex \*lock) | 销毁该互斥量 |
| 4 | bool mutex\_is\_locked(struct mutex \*lock) | 判断互斥量是否被锁住 |
| 5 | mutex\_lock | 获得锁，task除致命信号不可中断 |
| 6 | mutex\_unlock | 解锁 |
| 7 | mutex\_trylock | 尝试获得锁，不能加锁则立刻返回 |
| 8 | mutex\_lock\_interruptible | 获得锁，task可以被中断 |
| 9 | mutex\_lock\_killable | 获得锁，task可以被致命信号中断 |
| 10 | mutex\_lock\_io | 获得锁，在该task等待锁时，它会被调度器标记为io等待状态 |

**读写信号量（RW\_Semaphore）**

读写信号量与信号量有相似也有不同，它是如下一种同步机制：读写信号量将访问者分为读者或者写者，读者在持有读写信号量期间只能对该信号量保护的共享资源进行读访问，而只要一个任务需要写，它就被归类为写者，其进行访问之前必先获得写者身份，在其不需写访问时可降级为读者。读写信号量可同时拥有不受限的读者数，写者是排他性的，独占性的，而读者不排他。若读写信号量未被写者持有或者等待，读者就可以获得读写信号量，否则必须等待直到写者释放读写信号量为止；若读写信号量没有被读者或写者持有，也没用写者等待，写者可以获得该读写信号量，否则等待至信号量全部释放（没有其他访问者）为止。

**Structure Definition**

struct rw\_semaphore {

atomic\_long\_t count; // 信号量持有者数目

struct list\_head wait\_list;

raw\_spinlock\_t wait\_lock;

#ifdef CONFIG\_RWSEM\_SPIN\_ON\_OWNER

struct optimistic\_spin\_queue osq;

struct task\_struct \*owner;

#endif

#ifdef CONFIG\_RWSEM\_PRIO\_AWARE

long m\_count;

#endif

};

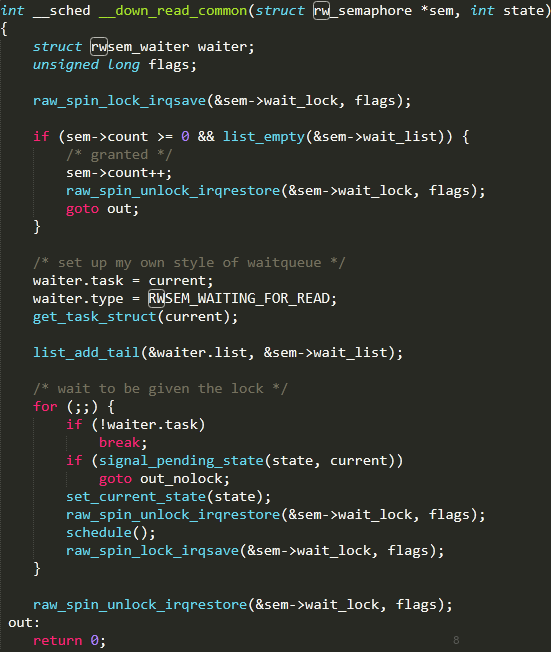
若从上述结构定义看，最关键的前三个字段与mutex、信号量十分相似不再赘述，后面的字段在附录中介绍MCS与OSQ会讲到。

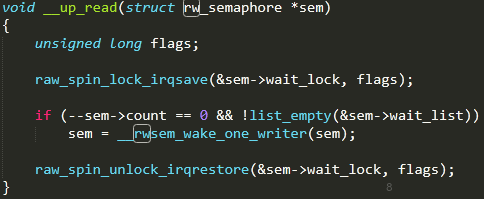
**读写信号量与信号量之间的关系**

读写信号量可能会引起进程阻塞,但是它允许N个读执行单元同时访问共享资源,而最多只允许有一个写执行单元访问共享资源;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。信号量不允许任何操作之间有并发,即:读操作与读操作之间、读操作与写操作之间、写操作与写操作之间,都不允许并发;而读写信号量则只允许读操作与读操作之间的并发,但不允许读操作与写操作之间的并发,也不允许写操作与写操作之间的并发。

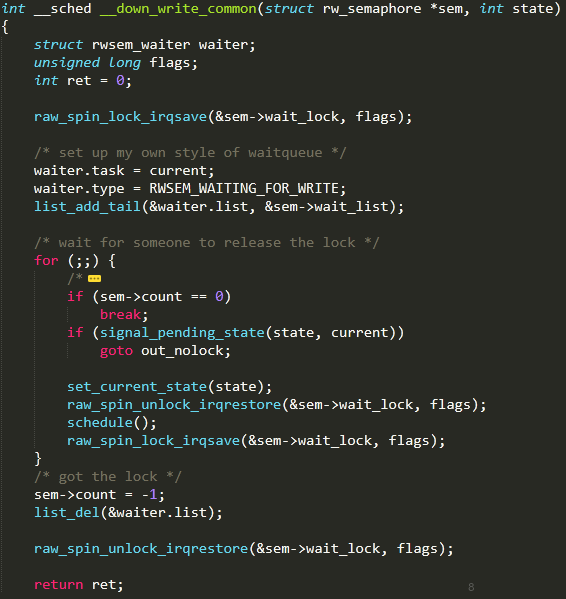
因此读写信号量比较适合读多写少的情况，可良好地利用读者并发的特性。

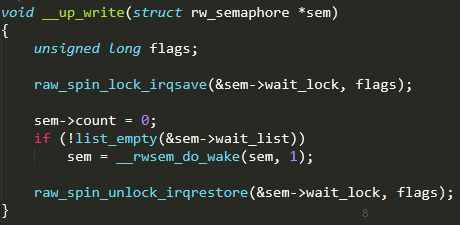
**\_\_down\_read & \_\_up\_read**





**\_\_down\_write & \_\_up\_write**





**读写信号量的改进**

读写信号量同mutex一样均引入了OSQ lock机制实现自旋等待。

**RW\_Semaphore API**

|  |  |  |
| --- | --- | --- |
| 1 | DECLARE\_RWSEM(name) | 静态声明并初始化读写信号量 |
| 2 | Init\_rwsem (sem） | 动态声明并初始化读写信号量 |
| 3 | down\_read(sem) | 读者获得读写信号量 |
| 4 | down\_read\_trylock(sem) | 读者尝试获得读写信号量，不能获得则立刻返回 |
| 5 | down\_write(sem) | 写者获得读写信号量 |
| 6 | down\_write\_trylock(sem) | 写者尝试获得读写信号量，不能获得则立刻返回 |
| 7 | up\_write(sem) | 写者释放读写信号量 |
| 8 | up\_read(sem) | 读者释放读写信号量 |
| 9 | downgrade\_write(sem) | 写者身份降级为读者 |

**读写锁（rw\_lock） 读写自旋锁**

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。

在读写锁保持期间也是抢占失效的。如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

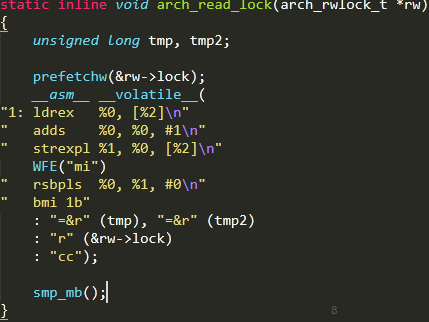
**Structure Definition**

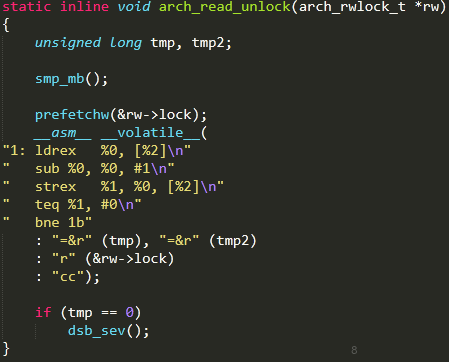
typedef struct { arch\_rwlock\_t raw\_lock;} rwlock\_t;

typedef struct { u32 lock;} arch\_rwlock\_t;

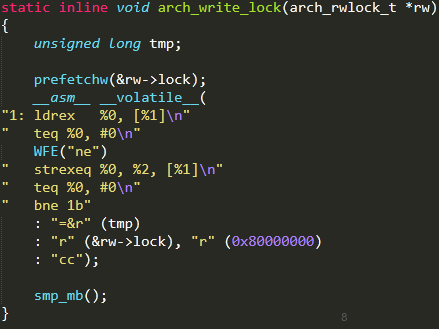
从结构上看，读写锁与自旋锁基本相似，实际上二者的实现也十分相似，二者的关系可以类比读写信号量与信号量的关系。

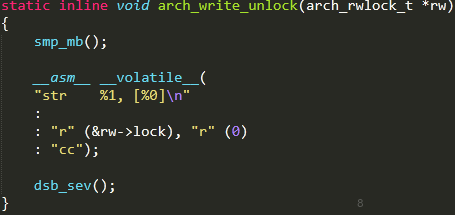
**arch\_read\_lock & arch\_read\_unlock**





**arch\_write\_lock & arch\_write\_unlock**





从这里可以看出，读写锁的实现上以及功能上，相当于针对自旋锁对于读多写少的场景提高并发度，设计原理与读写信号量十分类似。

**RW\_Lock API**

|  |  |  |
| --- | --- | --- |
| 1 | rwlock\_init(lock) | 静态声明并初始化读写锁 |
| 2 | read\_lock (lock） | 获得读者锁 |
| 3 | read\_unlock(lock) | 获得写者锁 |
| 4 | read\_trylock(lock) | 读者尝试获得读者锁，不能获得则立刻返回 |
| 5 | write\_lock(lock) | 获得写者锁 |
| 6 | write\_unlock(lock) | 释放写者锁 |
| 7 | write\_trylock（lock） | 尝试获得写者锁，不能获得则立刻返回 |
| 8 | read\_lock\_irqsave(lock， flags) /read\_unlock\_irqrestore(lock， flags) | 获得读者锁并关中断，将标志寄存器内容写入flags, 解锁反之 |
| 9 | read\_lock\_irq/read\_unlock\_irq | 获得读者锁并关中断， 解锁反之 |
| 10 | read\_lock\_bh/read\_unlock\_bh | 获得读者锁并关软中断，解锁反之 |
| 11 | write\_lock\_irqsave(lock， flags) /write\_unlock\_irqrestore(lock， flags) | 获得写者锁并关中断，将标志寄存器内容写入flags, 解锁反之 |
| 12 | write\_lock\_irq/write\_unlock\_irq | 获得写者锁并关中断， 解锁反之 |
| 13 | write\_lock\_bh/write\_unlock\_bh | 获得写者锁并关软中断， 解锁反之 |
| 14 | write\_trylock\_irqsave | 将标志寄存器内容写入flags并尝试获得写者锁，成功则关中断，失败则恢复寄存器内容立刻返回 |

**顺序锁（seqlock）**

顺序锁是对读写锁的一种优化:读者绝不会被写者阻塞，也就说，读者可以在写者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，不必等待写者完成写操作，写者也不需要等待所有读者完成读操作才去进行写操作。但是，写者与写者之间仍然是互斥的。写操作的优先级大于读操作。

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，因为写者可能使得指针失效，但读者如果正要访问该指针，将导致OOPs。如果读者在读操作期间，写者已经发生了写操作，那么，读者必须重新读取数据，以便确保得到的数据是完整的。顺序锁适用于读多写少的情况。

这种锁对于读写同时进行的概率比较小的情况，性能是非常好的，而且它允许读写同时进行，更大地提高了并发性。顺序锁的一个典型的应用在于时钟系统。

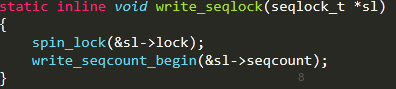
**Structure Definition**

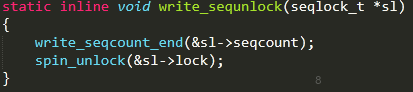
typedef struct seqcount { unsigned sequence;} seqcount\_t;

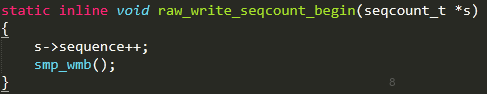
typedef struct { seqcount\_t seqcount; spinlock\_t lock;} seqlock\_t;

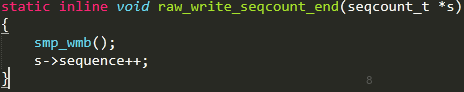
从结构上看，也是依赖于自旋锁的，seqcount用于同步写者访问的顺序以更新读者访问,自旋锁的作用在于实现写操作之间的互斥，读者访问不受限制。

**write\_seqlock & write\_sequnlock**

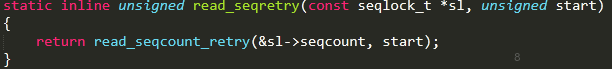


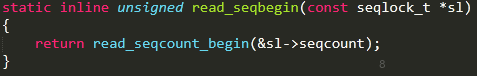


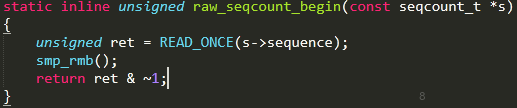




**read\_seqretry & read\_seqbegin**







read\_seqcount\_begin返回当前seqlock的seqcount, 在读完后，需调用read\_seqretry查看读者读完后的seqcount是否与读之前一致，一致则结束，不一致则说明有写操作正在或已经执行，需要重新读一次以更新数据。另外read\_seqbegin返回的是lock.seqcount/2,实际上是写操作发生的次数。

**seqlock API**

|  |  |  |
| --- | --- | --- |
| 1 | seqlock\_init(lock) /DEFINE\_SEQLOCK | 声明并初始化顺序锁 |
| 2 | write\_seqlock | 写者获得顺序锁 |
| 3 | write\_sequnlock | 写者释放顺序锁 |
| 4 | read\_seqbegin | 读取seqlock当前的写操作发生次数 |
| 5 | read\_seqretry | 判断是否与当前seqlock的seqcount一致 |
| 6 | read\_seqlock\_excl /read\_sequnlock\_excl | 一般读者无需获得锁，但内核也提供了读者获得锁/释放锁的接口。 |

其他\_irqsave,\_irq,\_bh版本均是与其他锁类似的。

**RCU（Read-Copy Update）**

RCU是读写锁的高性能版本，既允许多个读者同时访问被保护的数据，又允许多个读者和多个写者同时访问被保护的数据（注意：是否可以有多个写者并行访问取决于写者之间使用的同步机制），读者没有任何同步开销，而写者的同步开销则取决于使用的写者间同步机制。

对于被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机(所有引用该数据的CPU都退出对共享数据的操作时)把指向原来数据的指针重新指向新的被修改的数据。有一个专门的垃圾收集器探测读者的信号，一旦所有读者都已发送信号告知它们不在使用被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不能替代读写锁，当写比较多时，对读者的性能提高不能弥补写者导致的损失。且RCU是一种另类的实现，也不能代替顺序锁，具体原因之后讲解。

**RCU Basic API**

|  |  |  |
| --- | --- | --- |
| 1 | rcu\_read\_lock | 读者在读取由RCU保护的共享数据时使用该函数标记它进入读端临界区。 |
| 2 | rcu\_read\_unlock | 与rcu\_read\_lock配套使用，以标记读者退出临界区。 |
| 3 | synchronize\_rcu | 同步RCU, 即所有的读者已经完成读端临界区，写者才可以继续下一步操作。由于该函数将阻塞写者，只能在进程上下文中使用。 |
| 4 | call\_rcu | 该函数将把回调函数 func 注册到 RCU回调函数链上，然后立即返回。。 |
| 5 | rcu\_assign\_pointer | 用于RCU指针赋值 |
| 6 | rcu\_dereference | 用于RCU指针取值 |
| 7 | list\_add\_rcu | 向RCU注册一个链表结构 |
| 8 | list\_del\_rcu | 从RCU移除一个链表结构 |

**RCU 临界区管理**

之前的同步机制中，均是利用锁或原子操作实现的，一个锁管理一个临界区，并通过加锁解锁控制进程进入或者离开临界区。一个程序中可以存在若干的临界区，因此可以对应存在若干把锁分别管理，这是之前所有锁机制的基础。

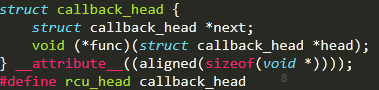
然而RCU并不基于锁机制实现，RCU字段是耦合在进程描述符中的，是一种与系统强耦合的同步机制，一个进程只有一个RCU，该RCU负责管理进程内所有的临界区，进程通过调用rcu\_read\_lock与rcu\_read\_unlock标记读者临界区，通过rcu\_assign\_pointer、list\_add\_rcu将数据纳入保护区，当写者copy出新数据时在读者全部退出临界区后，将新数据指针更新，旧数据将在垃圾收集器的检查中被释放，但存在延迟。

**RCU 限制条件**

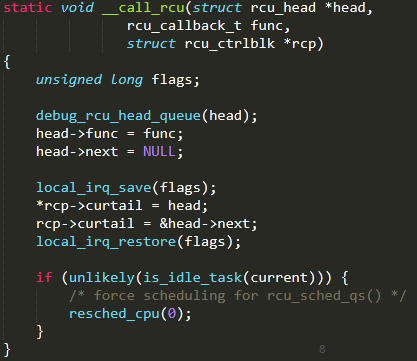
RCU只保护动态分配并通过指针引用的数据结构

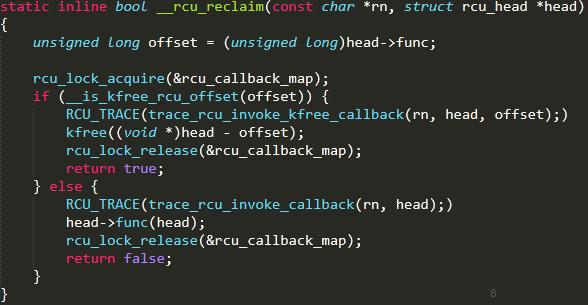
在被RCU保护的临界区中，任何内核路径都不能睡眠（经典）

**RCU callback的实现**

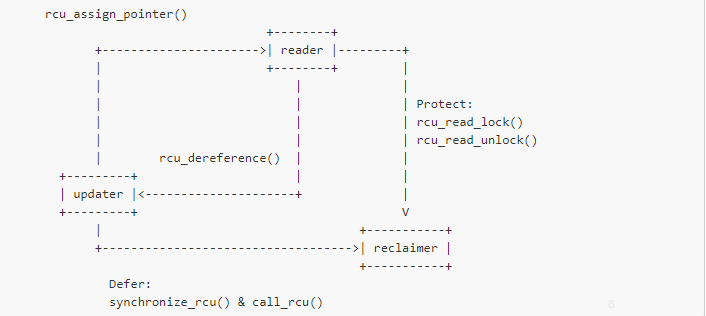


rcu\_head 是RCU回调函数的关键结构。此外，回调机制主要涉及两个基本函数\_\_call\_rcu(用于注册), \_\_rcu\_reclaim（用于调用）。





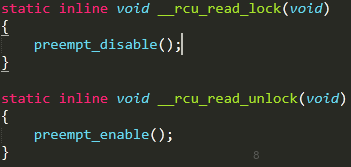
实际上，synchronize\_rcu在等待读者全数退出临界区时，也通过call\_rcu注册了回调函数。



相对麻烦的是回收阶段，RCU通过一个垃圾收集器检查需要回收的旧数据并调用回调函数释放，准确的说调用rcu\_check\_callbacks检查是否有需要执行的回调函数，而后调用rcu\_process\_callbacks执行必要的rcu 回调函数。

那么问题来了，谁去调用rcu\_check\_callbacks函数呢？时钟系统，每当时间片消耗完或者出现时钟中断，时钟系统都将调用rcu\_check\_callbacks进行及时检查处理，避免过量的旧数据垃圾造成内存浪费。

**RCU read**



RCU read之所以禁止抢占，主要是由于写者必须等待读者完全执行完退出临界区方能修改数据指针。一旦读者被抢占，那么其退出临界区的过程将会阻塞，进而阻塞写者，这对性能是一种不小的开销。但是现在的linux 内核版本中提供了可抢占的版本，只是对抢占深度做了把控。

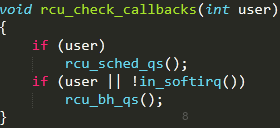
**RCU Synchronize**

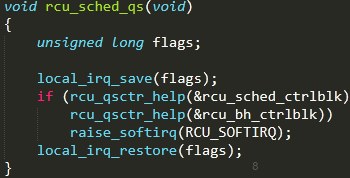
**可是RCU是如何获知所有读者已经离开临界区？**RCU read实现中并没有设置字段标记进出临界区，RCU是通过什么判断的呢？既然RCU read过程不可抢占，那么换言之，若所有 CPU 都已经过一次上下文切换，则所有前置 reader 的临界区必定全部退出。

我们主要分析以下两种：

rcu\_check\_callbacks

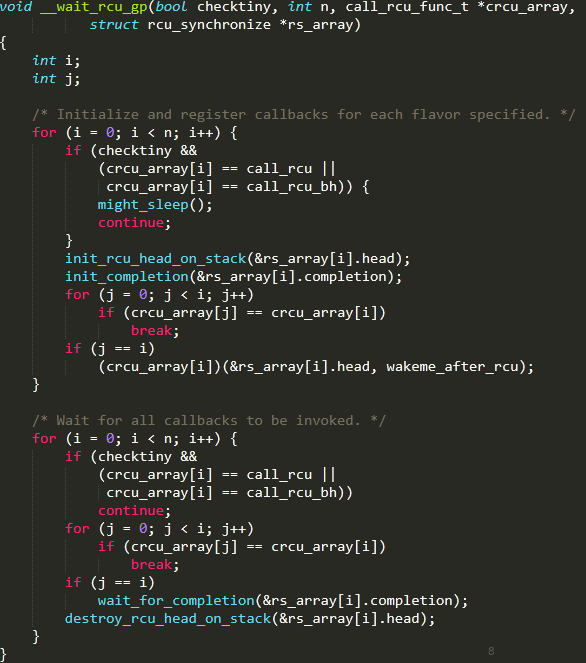
synchronize\_rcu



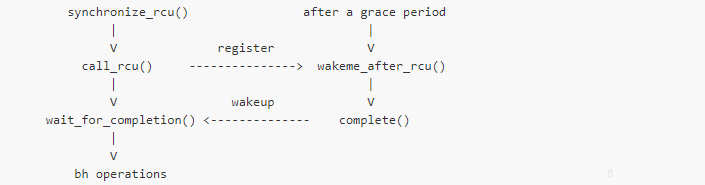


user其实在调用中真实的传入是user\_tick,值为1指用户时间，0指系统时间，由于RCU必须在内核态执行，因此user为1说明必然不处于lock~unlock的时段，很有可能已经发生过rcu\_read,因此发送一个RCU\_SOFTIRQ软中断，调用rcu\_process\_callbacks。

synchronize\_rcu的核心是wait\_rcu\_gp函数。

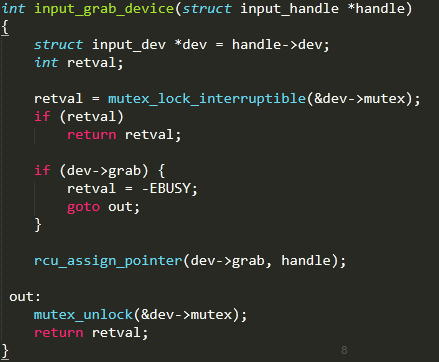


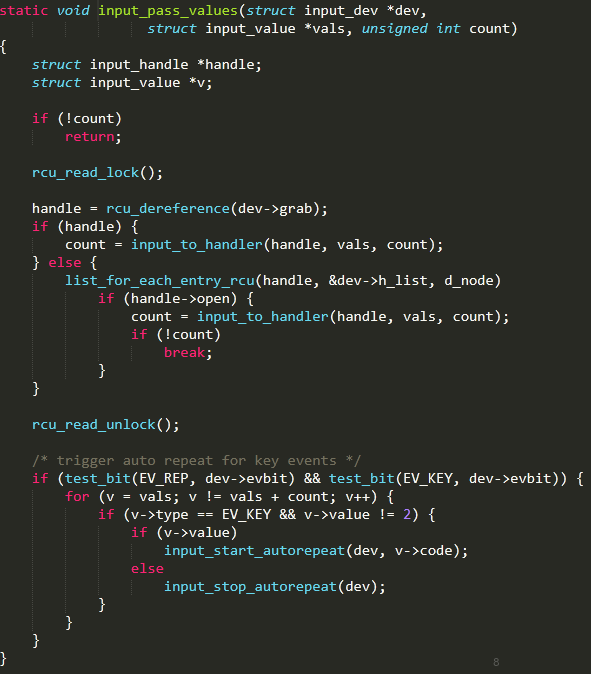
如下图：

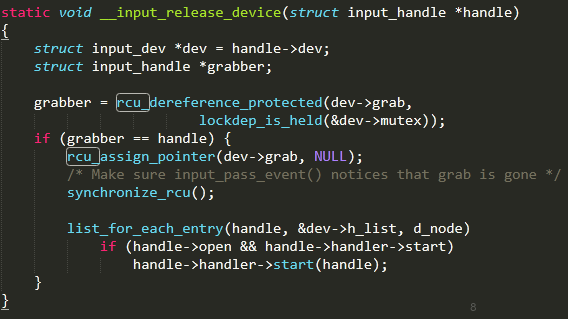


**RCU Example**

Input.c 中的使用为例。







**同步机制之间的比较**

|  |  |  |  |
| --- | --- | --- | --- |
| 机制 | 等待机制 | 优缺 | 适用于 |
| 原子操作 | 无，ldrex与strex实现内存独占访问 | 性能相当高， 场景受限 | 资源计数 |
| 自旋锁 | 忙等待， 唯一持有 | 多处理器下性能优异， 临界区时间长会浪费 | 中断上下文 |
| 信号量 | 睡眠等待（阻塞）， 多数持有 | 相对灵活，适用于复杂情况，但是时耗长 | 情况复杂且耗时长的场景，比如内核与用户空间的交互 |
| 互斥锁 | 睡眠等待（阻塞）， 优先自旋等待 唯一持有 | 较信号量，高效，适用于复杂场景，但存在若干限制条件 | 满足使用条件下，mutex优先于信号量 |
| 读写信号量 | 睡眠等待（阻塞）， 优先自旋等待， 读者多数持有， 写者唯一持有 | 读写优化的信号量，对于读多写少的情形性能大大提升 | 类似信号量，更适用于读多写少的场景 |
| 读写锁 | 忙等待， 读者多数持有， 写者唯一持有 | 读写优化的自旋锁，读者阻塞写者，对于读多写少的情形性能大大提升，但写者可能受阻塞降低性能 | 类似自旋锁，更适用于读多写少的场景 |
| 顺序锁 | 忙等待 | 写优先读的锁，对于读写同时比较少的情况拥有高性能表现 | 不允许读者阻塞写者，写者少于读者的情形 |
| RCU | 具体见九节 | 在绝大部分为读而只有极少部分为写的情况下，它是非常高效的，但延后释放内存会造成内存开销，写者阻塞比较严重 | 读多写少的情况下，对内存消耗不敏感 |