Linux 内核同步机制简介

2015/7/13
renyl

1 介绍

1) 由于现代 Linux 操作系统是多任务、SMP、抢占式以及中断是异步执行的，导致共享资源容易被并发访问，从而使得访问共享资源的各线程之间互相覆盖共享数据，造成被访问数据处于不一致状态，因此 Linux 提供了同步机制来防止并发访问。

2) 常用的同步机制（如自旋锁）用来保护共享数据使用起来简单有效，但由于 CPU 的处理速度与访问内存的速度差距越来越大，导致获取锁的开销相对于 CPU 的速度在不断的增加。因为这种锁使用了原子操作指令，需要原子地访问内存，即获取锁的开销与访问内存的速度相关。

3) Linux 内核根据对不同共享资源的特性，提供多种同步机制：原子操作、自旋锁、读-写自旋锁、信号量、读-写信号量、完成变量、顺序锁、禁止抢占、内存屏障及 RCU，本文将对其分别进行简要介绍。

2 原子操作（atomic）

2.1 基本原理

1) 所谓原子操作，就是该操作绝不会在执行完毕前被任何其它任务或事件打断，它是最小的执行单位，不可能有比它更小的执行单位。

2) 原子操作通常是内联函数，通过内联汇编指令来实现。

3) 原子操作需要硬件的支持，因此不同的体系结构的实现方式不同。

4) 内核提供了两组原子操作接口：整数操作和位操作。

2.1.2 原子整数操作

1) 原子操作主要用于实现资源计数，很多引用计数就是通过原子操作实现的。

2) 原子类型定义如下：（参看 RHEL6.5GA_x86_64 内核文件：/root/include/linux/types.h）

```
typedef struct {volatile int counter}  atomic_t;
```

3) 针对整数的原子操作只能对 atomic_t 类型的数据进行处理，原因如下：
   a) 让原子函数只接受 atomic_t 类型的操作数，可以确保原子操作只与这种特殊类型一起使用。
   b) 使用 atomic_t 类型确保编译器不对相应的值进行优化，使得原子操作最终接收到正确的内存地址。
   c) 可以屏蔽不同体系结构上实现原子操作的差异。

2.1.2 原子位操作

1) 位操作函数是对普通的内存地址进行操作的，对所操作的数据类型没有要求。
2) 位操作函数有两个参数：一个是位号，一个是指针。第 0 位表示给定地址的最低有效位，第 31 位表示给定地址的最高有效位，第 32 位表示下一个字的最低有效位（32 位系统）。

2.2 相关函数

2.2.1 原子整数

参考 RHEL6.5GA_x86_64 内核文件：/root/arch/x86/include/asm/atomic_32.h

函数	描述
ATOMIC_INIT(int i)	原子地初始化变量为 i
int atomic_read(atomic_t *v)	原子地读取变量 v
void atomic_set(atomic_t * v, int i);	原子地设置变量 v 的值为 i
void atomic_add(int i, atomic_t *v);	原子地给变量 v 加 i
void atomic_sub(int i, atomic_t *v);	原子地给变量 v 减 i
int atomic_sub_and_test(int i, atomic_t *v);	原子地从 v 减 i，等于 0 返回真，否则返回假
void atomic_inc(atomic_t *v);	原子地给变量 v 加 1
void atomic_dec(atomic_t *v);	原子地给变量 v 减 1
int atomic_dec_and_test(atomic_t *v);	原子地从 v 减 1，等于 0 返回真，否则返回假
int atomic_inc_and_test(atomic_t *v);	原子地从 v 加 1，等于 0 返回真，否则返回假
int atomic_add_negative(int i, atomic_t *v);	原子地从 v 加 i，等于负数返回真，否则返回假

2.2.2 原子位

参考 RHEL6.5GA_x86_64 内核文件：/root/include/asm-generic/bitops/atomic.h

函数	描述
void set_bit(int nr, volatile unsigned long *addr)	原子地置位

void clear_bit(int nr, volatile unsigned long *addr)	原子地清除
void change_bit(int nr, volatile unsigned long *addr)	原子地翻转
int test_and_set_bit(int nr, volatile unsigned long *addr)	原子地置位并返回该位原来的值
int test_and_clear_bit(int nr, volatile unsigned long *addr)	原子地清除并返回该位原来的值
int test_and_change_bit(int nr, volatile unsigned long *addr)	原子地翻转并返回该位原来的值

3 自旋锁（spin lock）

3.1 基本原理

1) 自旋锁最多只能被一个可执行线程持有，这样就可以防止多个执行线程同时进入临界区。

2) 如果一个执行线程试图获取一个已经被别的线程持有的自旋锁，那么该线程就会一直进行忙循环—自旋等待锁重新可用，因此自旋锁不应该被长时间持有。

3) 每当使用自旋锁的时候，会使 preempt_count 计数器加 1，释放自旋锁的时候，会使其值减 1。只有当 preemp_count 为 0 时，内核才可以进行抢占。即加锁和解锁分别表示禁止和允许内核抢占。

4) 自旋锁不会导致睡眠，因此可以在中断处理程序中使用。在中断处理程序中使用自旋锁时，一定要在索取锁之前，首先禁止本地中断（当前处理器上的中断请求），否则中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已经被持有自旋锁。这样一来，中断处理程序就会自旋，等待该锁重新可用，但是锁的持有者在这个中断处理程序执行完毕之前都不可能运行，这时候就会发生死锁。

   注：需要关闭的只是当前处理器上的中断。如果中断发生在别的处理器上，即使中断处理程序在同一锁上自旋，也不会妨碍锁的持有者最终释放锁。

5) 自旋锁是不可递归的。一个执行线程试图得到一个已持有的自旋锁，那么它必须自旋等待自己释放这个锁，但是由于处于自旋等待中永远没有机会释放锁，这样死锁发生了。

6) 使用锁的时候要有针对性，锁保护的是数据不是代码。

3.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/include/linux/spinlock.h

函数	描述
spin_lock()	获取指定的自旋锁
spin_lock_irq()	禁止本地中断，并获取指定的自旋锁
spin_lock_irqsave()	保存本地中断的当前状态，禁止本地中断，并获取指定的自旋

	锁
spin_lock()	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，如果获取失败，则立刻返回，不自旋等待
spin_is_locked	如果当前指定的锁已经被持有，则返回真，否则返回假

4 读-写自旋锁（rwlock）

4.1 基本原理

1) 当锁的用途可以明确地分为读取和写入时，如对一个链表可能既要进行更新又要进行检索，这个时候使用读-写自旋锁将非常有效。

2) 读-写自旋锁为读和写分别提供了不同的锁：
   a) 读锁可以同时被多个执行线程持有
   b) 写锁只能被一个执行线程持有，而且此时不能有并发的读操作

3）不能把一个读锁“升级”为写锁，否则会导致死锁发生。如下所示：

```
read_lock(&my_rwlock);
write_lock(&my_rwlock);
```

因为写锁会不断自旋，等待所有的读者释放锁，而自己所持有的读锁永远没有机会释放。

4) 一个线程递归地获得同一读锁也是安全的。

5) 读-写自旋锁倾向于读操作：当读锁被持有时，写操作为了互斥访问只能等待，而读操作可以继续成长地获取锁，等待的写操作在所有读操作释放锁之前是无法获的到锁的。因此，大量的读操作会使写操作处于饥饿状态。

4.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/include/linux/spinlock.h

函数	描述
read_lock()	获得指定的读锁
read_lock_irq()	禁止本地中断，并获得指定的读锁
read_lock_irqsave()	保存本地中断的当前状态，禁止本地中断，并获得指定的读锁
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_irqsave()	释放指定的读锁，并将本地中断恢复到以前的状态
write_lock()	获得指定的写锁
wirte_lock_irq()	禁止本地中断，并获得指定的写锁

write_lock_irqsave()	保存本地中断的当前状态，禁止本地中断，并获得指定的写锁
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_iqrstore()	释放指定的写锁，并将本地中断恢复到以前的状态
write_trylock()	试图获得指定的写锁，如果获取失败，立刻返回，不自旋等待
rw_lock_init()	初始化指定的 rwlock_t
rw_is_locked()	如如果当前指定的锁已经被持有，则返回真，否则返回假

5 信号量（semaphore）

5.1 基本原理

1) 信号量是一种睡眠锁。如果一个进程试图获得一个已经被占用的信号量，信号量将会将其放置一个等待队列，然后让其睡眠。这时处理器能够重新调度别的进程执行，等持有信号量的进程将信号量释放后，处于等待队列中的那个进程将被唤醒，并获得该信号量。

2) 由于进程在获取信号量失败时，不是进行自旋等待，而是进行睡眠，这样就比自旋锁提供了更好的处理器利用率。但是信号量比自旋锁有更大的开销，因为睡眠会引起进程上下文切换，从而导致产生开销。

3) 信号量不同于自旋锁，它不会禁止内核抢占，所以持有信号量的代码可以被抢占。

4) 信号量在创建时需要设置一个初始值，表示同时可以有多少个执行线程获取该信号量，若初始值为 1 该信号量就变成互斥锁，即同时只有一个执行线程可以获取该信号量。

5) 根据信号量的睡眠特性可以得出如下一些信息：
   a) 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况。
   b) 由于进程在锁被争用时会睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文中是不能进行睡眠的。
   c) 进程在占用信号量的同时不能占用自旋锁，因为在等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

5.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：
/root/kernel/semaphore.c 、/root/include/linux/semaphore.h

函数	描述
void sema_init(struct semaphore *sem, int val)	以指定的计数值初始化动态创建的信号量
void down(struct semaphore *sem)	试图获得指定的信号量，如果信号量已被别

	的进程持有，则进入不可中断睡眠状态
int down_interruptible(struct semaphore *sem)	试图获得指定的信号量，如果信号量已被别的进程持有，则进入可中断睡眠状态
int down_trylock(struct semaphore *sem)	以试图获得指定的信号量，如果信号量已被别的进程持有，则立刻返回非 0 值
void up(struct semaphore *sem)	释放指定的信号量，如果睡眠队列不为空，则唤醒其中一个进程。

6 读-写信号量（rwsemaphore）

6.1 基本原理

1) 读-写信号量与读-写自旋锁类似，对访问者进行了划分：读者和写者。

2) 所有的读-写信号量都是互斥信号量。读-写信号量可以同时被多个读者持有，但只能同时被一个写者持有且此时不能有并发的读者。

3) 写者在不需写访问的时候可以“降级”为读者。

6.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：
/root/kernel/rwsem.c、/root/include/linux/rwsem-spinlock.h

函数	描述
init_rwsem(sem)	对读-写信号量 sem 进行初始化
void down_read(struct rw_semaphore *sem)	读者获取读-写信号量 sem，获取失败进入睡眠状态
int down_read_trylock(struct rw_semaphore *sem)	读者获取读-写信号量 sem，获取失败立刻返回 0
void down_write(struct rw_semaphore *sem)	写者获取读-写信号量 sem，获取失败进入睡眠状态
int down_write_trylock(struct rw_semaphore *sem)	写者获取读-写信号量 sem，获取失败立刻返回 0
void up_read(struct rw_semaphore *sem)	读者释放读-写信号量 sem
void up_write(struct rw_semaphore *sem)	写者释放读-写信号量 sem
void downgrade_write(struct rw_semaphore *sem)	把写者“降级”为读者

7 完成变量（completion）

7.1 基本原理

1) 如果内核中一个任务需要发出信号通知另一个任务发生了某个特定的事件，利用完成变量使得两个任务同步。如 vfork 系统调用中，当子进程退出时，会使用完成变量唤

醒父进程。

2) 完成变量仅仅提供了替代信号量的一个简单解决方法。

7.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件： /root/include/linux/completion.h、/root/kernel/sched.c

函数	描述
void init_completion(struct completion *)	初始化动态创建的完成变量
void complete(struct completion *)	发送信号唤醒等待的任务
void wait_for_completion(struct completion *)	等待指定的完成变量发送信号

8 顺序锁（sequence lock）

8.1 基本原理

1) 顺序锁是对读-写锁的一种优化，这种锁主要依靠一个序列计数器实现。

2) 每当写者获取该锁时，使序列计数器加 1，释放该锁时，也会使序列计数器再加 1。
注：写者获取顺序锁的功能等同于“spin_lock 同时使序列计数器加 1”。

3) 读者绝不会被写者阻塞。读者在读取共享数据前后都会读取序列计数器，如果序列计数器的值一样，说明读者在读取共享数据的过程中没有被写者打断。如果序列计数器的值发生了变化，说明读者读操作期间有写者进行操作，那么读者必须重新读取数据，以便确保该数据是完整的。读操作如下所示:

```
do {
   seq=read_seqbegin(&my_seq_lock);
   /*read share data*/
}while (read_seqretry(&my_seq_lock,seq));
```

注：读者实际没有任何获取锁和释放锁的开销，只是获取序列计数器。

4) 顺序锁中写者不会阻塞读者，读者也不会阻塞写者，但是写者和写者之间仍然是互斥的。

5) 顺序锁有一个限制，它要求被保护的共享资源不含有指。因为写者可能使得指针失效，但读者如果正要访问该指针，那么将会导致系统崩溃。

6) 顺序锁是一种对写者更有利的锁。因为只要没有其他写者，写锁总是能够被成功获得，而写者在操作期间，使得读者必须不断循环地进行读操作，知道写者释放锁。

8.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/include/linux/seqlock.h

函数	描述
unsigned read_seqbegin(const seqlock_t *)	读者在在访问共享数据之前调用的函数
int read_seqretry(const seqlock_t *, unsigned)	读者在在访问共享数据之后调用的函数
void write_seqlock(seqlock_t *)	写者获取顺序锁，如果获取失败，自旋等待
void write_sequnlock(seqlock_t *)	写者释放顺序锁
int write_tryseqlock(seqlock_t *)	写者获取顺序锁，如果获取失败，立刻返回

9 禁止抢占（disable preempt）

9.1 基本原理

1) 使用自旋锁可以禁止内核抢占，但是若共享数据对每个处理器是唯一的，那么可以通过使用 preempt_disable()函数来禁止内核抢占，而不必使用自旋锁。
注：使用禁止内核抢占比使用自旋锁开销小。

2) preempt_disable()可以嵌套调用。preempt_disable()调用多少次就需要调用 preemt_enabled()多少次才能使内核抢占重新启用。

3) premmpt_disable()和 premmpt_enable()会对抢占计数器 preemp_count（低 8 位）进行操作。抢占计数器 preempt_count 分为 4 段：

bit32-24（区域 A）	bit23-16（区域 B）	bit15-8（区域 C）	bit7-0（区域 D）
其它	硬中断计数	软中断计数	普通抢占

只有当 preempt_count=0，才表示内核可以抢占。

9.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/include/linux/preempt.h

函数	描述
preempt_disable()	增加抢占计数值，从而禁止内核抢占
preempt_enable()	减少抢占计数值，若值为 0 表示内核抢占重新启用
preempt_enable_no_resched()	减少抢占计数值，若值为 0 表示内核抢占重新启用，但不检查被挂起的需要调度的任务
preempt_count()	返回抢占计数值

10 内存屏障（memory barrier）

10.1 基本原理

1) 内存屏障主要解决两个问题：单处理器下的乱序问题和多处理器下的内存同步问题。

2) CPU 乱序的由来：
   a) CPU 采用流水线来执行指令，一个指令的执行被分成：取指、译码、访存、执行、写回等多个阶段。
   b) 流水线是并行的。多条指令可以同时存在于流水线中，同时被执行，只要 CPU 内部相应的处理部件未被占满即可，比如说 CPU 有一个加法器和一个除法器，那么一条加法指令和一条除法指令就可能同时处于“执行”阶段。
   c) 在 X86 平台下，CPU 总是顺序的从内存里面取指令，然后将其顺序的放入指令流水线，但是由于指令执行时的各种条件，指令与指令之间的互相影响，可能导致顺序放入流水线的指令，最终乱序执行完成。比如，一条加法指令原本出现在一个除法指令的后面，但是由于除法指令的执行时间很长，在它执行完之前，加法指令可能已经执行完了。

3) CPU 乱序的特点：CPU 的乱序并不是任意的乱序，而是以保证上下文因果关系为前提的。
   a） 如下的代码：

```
b = c / d;
a = 1;
```

   有可能在 b 中存储新值之前就在 a 中存储 1 了。

   b) 如下的代码：

```
b = c / d;
a = b + 1;
```

   由于 a=b+1 这条指令的执行结果依赖于前一条指令 b=c/d 的执行结果，那么在 b=c/d 执行完毕之前，a=b+1 将被在“执行”阶段阻塞。

4) 编译器的乱序：作为编译优化的一种手段，是真正对指令顺序做了调整，但是编译器的乱序也必须保证程序上下文的因果关系不发生改变。
   如下所示：

```
b = c / d;
a = f(b);;
e++;
```

   由于 a=f(b)这条指令必定要等待 b=c/d 这条指令的执行结果，a=f(b)会阻塞在“执行”阶段，从而占用流水线资源。

   编译器优化可能对上述代码进行如下排序：

```
b = c / d;
e++;
a = f(b);
```

这样的话，由于指令 b=c/d 和指令 a=f(b)间隔开了，很有可能 a=f(b)指令在进入 CPU 的时候，b=c/d 已经执行完毕了，这样 a=f(b)就不会在流水线中阻塞了。

5) 乱序的后果：有了“保证程序上下文因果关系”这一前提，一般情况下是不会有问题的，但是有些逻辑程序，单纯从上下文中看不出它们的因果关系，这样乱序后就会出现问题。

a) 如下的代码：

```
*addr = 5;
val = *data;
```

从表面上看，addr 和 data 没有任何联系，可以放心的乱序执行。但是如果这是在某设备驱动中，这两个变量可能对应到设备的地址端口和数据端口，并且这个设备规定读写设备上的某个寄存器时，先将寄存器编号设置到地址端口，然后就可以通过对数据端口的读写而操作对应的寄存器。这样的话，一乱序执行就可能造成错误。

b) 如下的代码：

（CPU1）线程 1	（CPU2）线程 2
obj->ready = 1; obj->data = 100;- - -	- - if (obj->read) { do_something(obj->data); }

线程 1 给标记位 ready 置 1 和设置 data，线程 2 根据标记位 ready 的值来判断是否读取 data 然后做处理。由于 date 和 ready 没有逻辑关系，那么乱序很可能发生，结果执行流程可能为这样：线程 1 设置 data，线程 2 判断 ready 的值为假，不读取 data 进行处理。

6) 内存屏障主要有：读屏障、数据相关读屏障、写屏障、通用屏障和编译器屏障。

a) 读屏障：它用于保证读操作有序，屏障之前的读操作一定会先于屏障之后的读操作完成，写操作不受影响，且同属于屏障某一侧的读操作不受影响。举个例子，如下所示：

```
a = b;        //读操作
a2 = b2;     //读操作
i = 1;        //写操作
rmb();     //读屏障
c = d;     //读操作
j = 2;     //写操作
```

语句 c=d 一定后于 a=b 以及 a2=b2 执行，屏障同一侧的 a=b 和 a2=b2 的执行顺序可以重新排序，写操作 i=1 和 j=2 的执行顺序也可以重新排序。

b) 数据相关度屏障：它与读屏障类似，但是屏障之后的读操作与屏障之前的读操作是数据相关的。其速度比读屏障速度要快。举个例子，如下所示：

```
pp = p;
read_barrier_depends(); //数据相关读屏障
b = *pp;      //注意：这里是*pp 不是 pp，否则就不需要屏障来保证执行顺序
```

由于 b = *pp，所以 b 和 pp 是数据相关的，使用数据相关度屏障可以保证执行

顺序。

c) 写屏障：它用于保证写操作有序，屏障之前的写操作一定会先于屏障之后的写操作完成，读操作不受影响，且同属于屏障某一侧的写操作不受影响。

d) 通用屏障：它用于保证读和写操作有序，屏障之前的读和写操作一定会先于屏障之后的读和写操作完成，且同属于屏障某一侧的读和写操作不受影响。

e) 编译器屏障：用于限制编译器的指令重新排序。读屏障、写屏障、通用屏障都隐含了编译器屏障的功能。

7) 多处理器的屏障：一个处理器（CPU—a）对内存的写操作并不是直接就在内存上生效的，而是要先经过自身的 Cache。另一处理器（CPU—b）如果要读取相应内存上的新值，需要等 CPU-a 的 Cache 同步到内存，然后 CPU-b 的 Cache 再从内存同步这个新值。而如果需要同步的值不止一个的话，就会存在顺序问题。如下所示：

CPU-a	CPU-b

a = 3;
wmb();
b = 4;
-
-
	-
	-
	c = b;
	rmb();
	d = a;

由于使用了 wmb()保证了 CPU-a 不发生乱序，同时也保证屏障之前的 Cache 消息先于屏障之后的消息被发出去（发送给 CPU-b）。CPU-b 接收到 CPU-a 发送的 Cache 消息，但是可能会先处理 b=4 的消息（更新 b 的数据），导致最后 c=4，而 d 并不等于 3。使用 rmb()可以确保 CPU-b 按顺序更新接收到的消息，从而使得 c=4 和 d=3。

8) 多处理器下屏障的误区：内存屏障保证的是“一个 CPU 的多个操作的顺序”（被另一个 CPU 所观察到的顺序），而不保证“两个 CPU 的操作顺序”。如下所示：

CPU-a	CPU-b

a = 5;
wmb();
b = 6;
	j = b;
	rmb();
	I = a;

i 不一定等于 5，因为两个 CPU 的执行操作没有关联，不知谁先执行，因此内存屏障保证的是一个 CPU 的多个操作顺序。

10.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/arch/x86/include/asm/system.h

函数	描述
rmb()	读屏障
read_barrier_depends()	数据相关读屏障（在 x86 平台仅是空操作）
wmb()	写屏障
mb()	通用屏障
smp_rmb()	在 SMP 上提供 rmb()功能，在 UP 上提供 barrier()功能
smp_read_barrier_depends()	在 SMP 上提供提供 read_barrier_depends()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()	阻止编译器重新排序

11 RCU（read-copy-update）

11.1 基本原理

1) RCU 的适用范围：RCU 针对经常发生读取而很少写入的情形做了优化，因此适用于读者多而写者少的情况。

2) RCU 的使用：
   a) 读者不需要获取任何锁就可以访问共享资源，即读者不需要与其它读者或写者保持同步。
   b) 写者与写者之间需要保持同步，且写者必须要等它之前的读者全部都退出之后才能释放之前的资源。

3) RCU 的机制：
   a) RCU 保护的是指针。因为指针赋值是一条单指令（原子操作），因此更改指针指向不需要考虑它的同步，只需要考虑 Cache 的影响。
   b) 写者对共享资源进行操作时，会对共享资源进行拷贝，然后对拷贝的数据进行修改，修改完成之后，最后会使用一个回调（callback）机制在适当的时机（之前的读者全部退出之后，这一时机也称为 grace period）把指向原来数据的指针重新指向新的被修改的数据。

4) RCU 的规定：
   a) 读者在临界区内（调用 rcu_read_lock（可以嵌套调用）和 rcu_read_unlock 期间）不能发生进程上下文切换（因为写者需要等待读者完成,否则写者进程也会一直被阻塞），但是允许中断发生。如下所示：（参考 RHEL6.5GA_x86_64 内核文件 root/include/linux/rcutree.h、root/include/linux/rcupdate.h

```
static inline void rcu_read_lock(void)
{
     __rcu_read_lock();
     __acquire(RCU);              //选择编译函数
    rcu_read_acquire();
}

static inline void __rcu_read_lock(void)
{
    preempt_disable();
 }

static inline void rcu_read_unlock(void)
{
```

```
    rcu_read_release();
     __release(RCU);
    __rcu_read_unlock();
 }
static inline void __rcu_read_unlock(void)
{
      preempt_enable();
}
```

由代码可知：

i. rcu_read_lock()和 rcu_read_unlock()只是禁止和启用内核抢占，读者访问共享资源时并不获取任何锁。

ii. 中断允许发生是因为即使发生了中断，也不会导致进程发生上下文切换。

b) 写者在执行完更新共享数据后，需要调用一个回调函数等待所有的读者对老指针的引用结束之后释放老指针。由于读者在临界区中不能发生进程上下文切换，当系统中所有处理器都发生了一次进程上下文切换（称为 quiescent state），所有的读者肯定结束了对老指针的引用。

c) 释放老指针，linux 提供了两种方法：call_rcu 和 synchronize_rcu

i. call_rcu:是一种异步方式，可用在中断上下文中。

ii. synchronize_rcu:是一种同步方式，在所有处理器发生一次进程切换前，将会在一个等待队列中睡眠，不能用在中断上下文中。

11.2 相关函数

参考 RHEL6.5GA_x86_64 内核文件：/root/include/linux/rcupdate.h

函数	描述
void rcu_read_lock(void)	读者访问共享数据前调用，标记读者进入读端临界区
void rcu_read_unlock(void)	读者访问共享数据后调用，标记读者退出读端临界区
void synchronize_rcu(void)	阻塞式，等待所有 CPU 发生一次上下文切换后释放老指针
void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu))	非阻塞式，等待所有 CPU 发生一次上下文切换后释放老指针