linux内核避免竞争的机制

中断屏蔽

在单 CPU 范围内避免竞态的一种简单方法是在进入临界区之前屏蔽系统的中断。CPU 一般都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生，而且，由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也就得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable() //屏蔽中断
...
critical section //临界区
...
local_irq_enable() //开中断
```

由于 Linux 系统的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。
`local_irq_disable()`和 `local_irq_enable()`都只能禁止和使能本 CPU 内的中断，因此，并不能解决 SMP 多 CPU 引发的竞态。

因此，**单独使用中断屏蔽通常不是一种值得推荐的避免竞态的方法，它适宜与自旋锁联合使用。**

与 `local_irq_disable()`不同的是，`local_irq_save (flags)` 除了进行禁止中断的操作以外，还保存目前 CPU 的中断位信息，

`local_irq_restore (flags)` 进行的是与 `local_irq_save (flags)` 相反的操作。

如果只是想禁止中断的底半部，应使用 `local_bh_disable()`，使能被 `local_bh_disable()`禁止的底半部应该调用 `local_bh_enable()`。

原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。 Linux 内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不被打断。位和整型变量原子操作都依赖底层CPU 的原子操作来实现，因此所有这些函数都与 CPU 架构密切相关。

整型原子操作

1. 设置原子变量的值
   `void atomic_set(atomic_t *v, int i);` //设置原子变量的值为 i
   `atomic_t v = ATOMIC_INIT(0);` //定义原子变量 v 并初始化为 0
2. 获取原子变量的值
   `atomic_read(atomic_t *v);` //返回原子变量的值

3. 原子变量加/减
   `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
4. 原子变量自增/自减
   `void atomic_inc(atomic_t *v);` //原子变量增加 1
   `void atomic_dec(atomic_t *v);` //原子变量减少 1
5. 操作并测试
   ```
   int atomic_inc_and_test(atomic_t *v);
   int atomic_dec_and_test(atomic_t *v);
   int atomic_sub_and_test(int i, atomic_t *v);
   ```
   上述操作对原子变量执行自增、自减和减操作后（注意没有加）测试其是否为 0，为 0 则返回 true，否则返回 false。
6. 操作并返回
   ```
   int atomic_add_return(int i, atomic_t *v);
   int atomic_sub_return(int i, atomic_t *v);
   int atomic_inc_return(atomic_t *v);
   int atomic_dec_return(atomic_t *v);
   ```
   上述操作对原子变量进行加/减和自增/自减操作，并返回新的值。

位原子操作

1. 设置位
   `void set_bit(nr, void *addr);` 上述操作设置 addr 地址的第 nr 位，所谓设置位即将位写为 1。
2. 清除位
   `void clear_bit(nr, void *addr);` 上述操作清除 addr 地址的第 nr 位，所谓清除位即将位写为 0。
3. 改变位
   `void change_bit(nr, void *addr);` 上述操作对 addr 地址的第 nr 位进行反置。
4. 测试位
   `test_bit(nr, void *addr);` 上述操作返回 addr 地址的第 nr 位。
5. 测试并操作位
   ```
   int test_and_set_bit(nr, void *addr);
   int test_and_clear_bit(nr, void *addr);
   int test_and_change_bit(nr, void *addr);
   ```
   上述 `test_and_xxx_bit (nr, void *addr)` 机制是设置addr指针所指向对象的第nr位，并返回原先nr位上的值。8/27/2022 10:07:55 PM

下面代码给出了原子变量的使用实例，它用于使设备最多只能被一个进程打开。

自旋锁

自旋锁（spin lock）是一种对临界资源进行互斥访问的典型手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。 如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。
理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区或者标记为“我当前在运行，请稍等一

会"或者标记为"我当前不在运行，可以被使用"。如果 A 执行单元首先进入例程，它将持有自旋锁；当 B 执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A 执行单元释放后才能进入。

Linux 系统中与自旋锁相关的操作主要有如下 4 种。

1. 定义自旋锁
   `spinlock_t spin;`

2. 初始化自旋锁
   `spin_lock_init(lock)` 该宏用于动态初始化自旋锁 lock

3. 获得自旋锁
   `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；
   `spin_trylock(lock)` 该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，实际上不再"在原地打转"；

4. 释放自旋锁
   `spin_unlock(lock)`
   该宏释放自旋锁 lock，它与 `spin_trylock` 或 `spin_lock` 配对使用。

自旋锁一般这样被使用，如下所示：

```
//定义一个自旋锁
spinlock_t lock;
spin_lock_init(&lock);

spin_lock (&lock) ; //获取自旋锁，保护临界区
......//临界区
spin_unlock (&lock) ; //解锁
```

自旋锁主要针对 SMP 或单 CPU 但内核可抢占的情况，对于单 CPU 但是内核不支持抢占的系统，自旋锁退化为空操作。在单CPU内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU 系统的行为实际很类似于 SMP系统，因此，在这样的单 CPU 系统中使用自旋锁仍十分必要。

尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部（BH）的影响。为了防止这种影响，就需要用到自旋锁的衍生。`spin_lock()/spin_unlock()`是自旋锁机制的基础，它们和关中断 `local_irq_ disable()`/开中断 `local_irq_enable()`、关底半部`local_bh_disable()`/开底半部 `local_bh_enable()`、关中断并保存状态字 `local_irq_save()`/开中断并恢复状态 `local_irq_restore()`结合就形成了整套自旋锁机制，关系如下所示：

```
spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_unlock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()
```

驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题:

1. 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。
2. 自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。此外，如果进程获得自旋锁之后再阻塞，也有可能导致死锁的发生。`copy_from_user()`、`copy_to_user()`和 `kmalloc()`等函数都有可能引起阻塞，因此在自旋锁的占用期间不能调用这些函数。

下面代码给出了自旋锁的使用实例，它被用于实现使得设备只能被最多一个进程打开。

```
1  int xxx_count = 0;/*定义文件打开次数计数*/
2
3  static int xxx_open(struct inode *inode, struct file *filp)
4  {
5    ...
6    spinlock(&xxx_lock);
7    if (xxx_count)/*已经打开*/
8    {
9      spin_unlock(&xxx_lock);
10     return  - EBUSY;
11   }
12   xxx_count++;/*增加使用计数*/
13   spin_unlock(&xxx_lock);
14   ...
15   return 0; /* 成功 */
16 }
17
18 static int xxx_release(struct inode *inode, struct file *filp)
19 {
20   ...
21   spinlock(&xxx_lock);
22   xxx_count--; /*减少使用计数*/
23   spin_unlock(&xxx_lock);
24
25   return 0;
26 }
```

读写自旋锁

自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生锁读写自旋锁（rwlock）可允许读的并发。 读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有一个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。 读写自旋锁涉及的操作如下所示。

1. 定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态初始化 */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化 */
```

1. 读锁定

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

3. 读解锁

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。
read_lock_irqsave()、read_lock_irq()和 read_lock_bh()分别是 read_lock()分别与 local_irq_save() 、 local_irq_disable() 和 local_bh_disable() 的 组合，读解锁函数 read_unlock_irqrestore()、read_unlock_ irq()、read_unlock_bh()的情况与此类似。
4. 写锁定

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

5. 写解锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

其中： write_lock_irqsave() 、 write_lock_irq() 、 write_lock_bh() 分 别 是 write_lock() 与 local_irq_save() 、 local_irq_disable() 和 local_bh_disable() 的组合，写解锁函数 write_unlock_irqrestore()、write_unlock_irq()、write_unlock_bh()的情况与此类似。

在对共享资源进行读取之前，应该先调用写锁定函数，完成之后应调用写解锁函数。和 spin_trylock()一样，write_trylock()也只是尝试获取读写自旋锁，不管成功失败，都会立即返回。 读写自旋锁一般这样被使用，如下所示：

```
rwlock_t lock;   //定义 rwlock
rwlock_init(&lock); //初始化 rwlock

//读时获取锁
read_lock(&lock);
...  //临界资源
read_unlock(&lock);

//写时获取锁
write_lock_irqsave(&lock, flags);
... //临界资源
write_unlock_irqrestore(&lock, flags);
```

顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元可以在写执行单元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，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。 但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。 如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。这种锁在读写同时进行的概率比较小时，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。
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，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致 Oops。 在 Linux 内核中，写执行单元涉及如下顺序锁操作。

1. 获得顺序锁

```
void write_seqlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
write_seqlock_irqsave(lock, flags)
write_seqlock_irq(lock)
write_seqlock_bh(lock)
```

其中：

```
write_seqlock_irqsave() = loal_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

2. 释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中：

```
write_sequnlock_irqrestore() = write_sequnlock() + local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
......
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与 spinlock 相同。 读执行单元涉及如下顺序锁操作。
1．读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```

读执行单元在对被顺序锁 s1 保护的共享资源进行访问前需要调用该函数，该函数仅返回顺序锁 s1 的当前顺序号。其中：

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

2．重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 s1 保护的共享资源后需要调用该函数来检查，在读访问期间是否有写操作。如果有写操作，读执行单元就需要重新进行读操作。其中：

```
read_seqretry_irqrestore() = read_seqretry() + local_irq_restore()
```

读执行单元使用顺序锁的模式如下：

```
do {
seqnum = read_seqbegin(&seqlock_a);
//读操作代码块
...
} while (read_seqretry(&seqlock_a, seqnum));
```

读 - 拷贝 - 更新

RCU（Read-Copy Update，读 - 拷贝 - 更新），它是基于其原理命名的。RCU 并不是新的锁机制，它对于 Linux 内核而言是新的。早在 20 世纪 80 年代就有了这种机制，而在 Linux 系统中，开发 2.5.43 内核时引入该技术，并正式包含在 2.6 内核中。 对于被 RCU 保护的共享数据结构，读执行单元不需要获得任何锁就可以访问它，不使用原子指令，而且在除 Alpha 的所有架构上也不需要内存栅（Memory Barrier），因此不会导致锁竞争、内存延迟以及流水线停滞。不需要锁也使得使用更容易，因为死锁问题就不需要考虑了。
使用 RCU 的写执行单元在访问它前需首先复制一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作的时候。读执行单元没有任何同步开销，而写执行单元的同步开销则取决于使用的写执行单元间的同步机制。
RCU 可以看做读写锁的高性能版本，相比读写锁，RCU 的优点在于既允许多个读执行单元同时访问被保护的数据，又允许多个读执行单元和多个写执行单元同时访问被保护的数据。 但是，RCU 不能替代读写锁，因为如果写比较多时，对读执行单元的性能提高不能弥补写执行单元导致的损失。因为使用 RCU 时，写执行单元之间的同步开销会比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其他写执行单元的修改操作。
Linux 系统中提供的 RCU 操作如下 4 种。

1. 读锁定

```
rcu_read_lock()
rcu_read_lock_bh()
```

2. 读解锁

```
rcu_read_unlock()
rcu_read_unlock_bh()
```

使用 RCU 进行读的模式如下：

```
rcu_read_lock()
...//读临界区
```

```
rcu_read_unlock()
```

其中 rcu_read_lock()和 rcu_read_unlock()实质只是禁止和使能内核的抢占调度，如 下所示：

```
#define rcu_read_lock() preempt_disable()
#define rcu_read_unlock() preempt_enable()
```

其变种 rcu_read_lock_bh()、rcu_read_unlock_bh()则定义为：

```
#define rcu_read_lock_bh() local_bh_disable()
#define rcu_read_unlock_bh() local_bh_enable()
```

3. 同步 RCU

   synchronize_rcu()

   该函数由 RCU 写执行单元调用，它将阻塞写执行单元，直到所有的读执行单元已经完成读执行单元临界区，写执行单元才可以继续下一步操作。如果有多个 RCU写执行单元调用该函数，它们将在一个 grace period（即所有的读执行单元已经完成对临界区的访问）之后全部被唤醒。synchronize_rcu()保证所有 CPU 都处理完正在运行的读执行单元临界区。

   synchronize_kernel()

   内核代码使用该函数来等待所有 CPU 处于可抢占状态，目前功能等同于synchronize_rcu()，但现在已经不建议使用，而是使用 synchronize_sched()，该函数用于等待所有 CPU 都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，但不能保证正在运行的软中断处理完毕。

4. 挂接回调

   void fastcall call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu));

   函数 call_rcu()也由 RCU 写执行单元调用，它不会使写执行单元阻塞，因而可以在中断上下文或软中断中使用。该函数将把函数 func 挂接到 RCU 回调函数链上，然后立即返回。函数 synchronize_rcu()的实现实际上使用了 call_rcu()函数。 void fastcall call_rcu_bh(struct rcu_head *head, void (*func)(struct rcu_head *rcu)); call_ruc_bh()函数的功能几乎与 call_rcu()完全相同，唯一差别就是它把软中断的完成也当做经历一个 quiescent state（静默状态），因此如果写执行单元使用了该函数， 在进程上下文的读执行单元必须使用 rcu_read_lock_bh()。 每个 CPU 维护两个数据结构 rcu_data 和 rcu_bh_data，它们用于保存回调函数，函数 call_rcu()把回调函数注册到 rcu_data，而 call_rcu_bh()则把回调函数注册到rcu_bh_data，在每一个数据结构上，回调函数被组成一个链表，先注册的排在前头，后注册的排在末尾。

   使用 RCU 时，读执行单元必须提供一个信号给写执行单元以便写执行单元能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测读执行单元的信号，一旦所有的读执行单元都已经发送信号告知它们都不在使用被 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 还增加了链表操作函数的 RCU 版本，如下所示：

   static inline void list_add_rcu(struct list_head *new, struct list_head *head);

   该函数把链表元素 new 插入到 RCU 保护的链表 head 的开头，内存栅保证了在引用这个新插入的链表元素之前，新链表元素的链接指针的修改对所有读执行单元是可见的。

   static inline void list_add_tail_rcu(struct list_head *new, struct list_head *head);

   该函数类似于 list_add_rcu()，它将把新的链表元素 new 添加到被 RCU 保护的链表的末尾。

`static inline void list_del_rcu(struct list_head *entry);`
该函数从 RCU 保护的链表中删除指定的链表元素 entry。
`static inline void list_replace_rcu(struct list_head *old, struct list_head *new);`
该函数是 RCU 新添加的函数，并不存在非 RCU 版本。它使用新的链表元素 new取代旧的链表元素 old，内存栅保证在引用新的链表元素之前，它对链接指针的修正对所有读执行单元是可见的。
`list_for_each_rcu(pos, head)`
该宏用于遍历由 RCU 保护的链表 head，只要在读执行单元临界区使用该函数，它就可以安全地和其他 _rcu 链表操作函数并发运行如 list_add_rcu()。
`list_for_each_safe_rcu(pos, n, head)`
该宏类似于 list_for_each_rcu，不同之处在于它允许安全地删除当前链表元素 pos。
`list_for_each_entry_rcu(pos, head, member)`
该宏类似于 list_for_each_rcu，不同之处在于它用于遍历指定类型的数据结构链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。
`static inline void hlist_del_rcu(struct hlist_node *n)` 它从由 RCU 保护的哈希链表中移走链表元素 n。
`static inline void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);`
该函数用于把链表元素 n 插入到被 RCU 保护的哈希链表的开头，但同时允许 读执行单元对该哈希链表的遍历。内存栅确保在引用新链表元素之前，它对指针的修改对所有读执行单元可见。
`hlist_for_each_rcu(pos, head)`
该宏用于遍历由 RCU 保护的哈希链表 head，只要在读端临界区使用该函数，它就可以安全地和其他_rcu 哈希链表操作函数（如 hlist_add_rcu）并发运行。
`hlist_for_each_entry_rcu(tpos, pos, head, member)`
类似于 hlist_for_each_rcu()，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。 目前，RCU 的使用在内核中已经非常普遍.