Linux 内核的同步机制

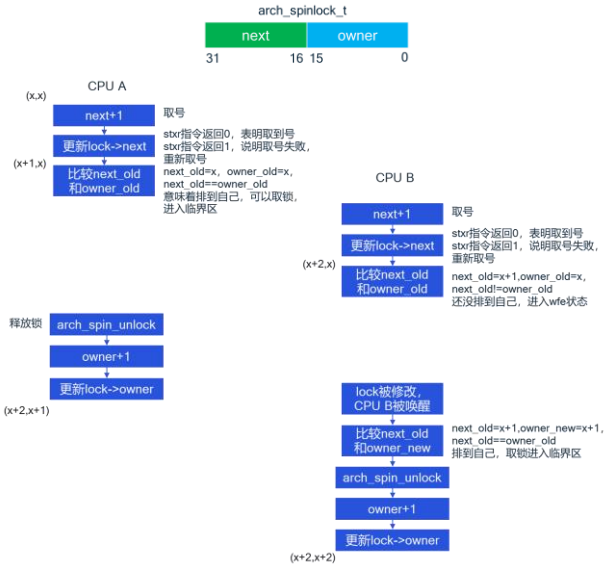
--自旋锁 spinlock

一、取号自旋锁

1.1 基本原理

针对 int/long 型变量的同步问题，使用原子操作 atomic_add/sub 或 atomic64_add/sub 就可以保证数据安全。但是更多情况下需要对一段代码进行保护，这种存在竞争情况的代码叫临界区。在进入临界区之前，为防止数据被其他线程修改，需要加锁来保证只有自己可以执行临界区的代码。

取号自旋锁的原理类似银行取号办业务，每个人都要先取号，只有当叫到自己的号才能办理。锁的结构体 arch_spinlock_t 包含 owner 和 next 两个各 16 位长度的元素，next 是当前系统中最后来尝试取锁的号，owner 是持锁的号。当 CPU A 尝试来取锁的时候会先保存当前自旋锁的值，把其中的 next 值加 1(取号)保存下来，next_1=next+1，然后更新自旋锁的值—lock->next=next+1。比较取号之前的锁中的 next 和 owner 值，如果相等，表明排到了自己，拿到锁，进入临界区。如果这时候 CPU B 也来尝试取锁，同样的先取号--把锁中的 next 值加 1 保存下来，next_2 =next+2，然后更新锁的值—lock->next=next+2。这时候比较更新之前的锁的 next 值（next_1=next+1）比 owner 大 1，表明锁被占用，调用 wfe 指令进入低功耗状态等待锁释放。当 CPU A 释放锁的时候，把 owner 加 1，同时 CPU B 会从低功耗中唤醒（为什么可以被唤醒后面介绍），比较取锁时候的 next 值（next+1）和当前 owner 值(owner+1)，两者相等，拿到锁，进入临界区。



ticket spinlock 原理图

1.2 wfe 机制和释放锁

对于没拿到自旋锁的 CPU，会处于忙等的状态，一直等到锁被释放才会再次尝试取锁。CPU 在这段时间内既然做不了其他事情，对于 ARM 架构的 CPU 可以调用 wfe 指令通过 WFE（wait for event）机制进入低功耗状态。WFE 机制的工作原理依赖于 event register 寄存器，包括 local event register 和 global event register。分别使用 sevl 和 sev 指令可以设置 local event register 和 global event register。当 local event register 为 1 时，wfe 指令仅仅是清零 local event register，不会让 CPU 进入低功耗状态；当 local event register 为 0 时，wfe 指令才会让 CPU 进入低功耗状态。另外，由于 event register 的初始值是不确定的，因此一般在调用 wfe 指令试图让 CPU 进入低功耗之前会先调用 sevl 指令设置 local event register 为 1，这样 wfe 指令可以清零 local event register，再次调用 wfe 的时候就可以成功进入低功耗。

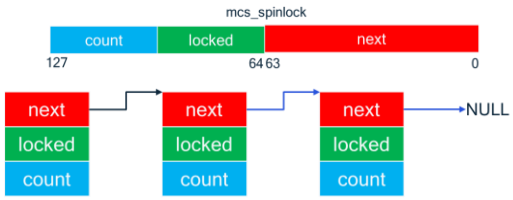
当 CPU 通过 wfe 指令进入低功耗状态后，以下事件可以唤醒 CPU。1）其他 CPU 调用 sev 指令设置 global event register 为 1。2）如果中断没有被屏蔽，IRQ、FIQ 也可以唤醒 CPU。3）可以清零内存全局监视器的（global monitor）事件，参考原子操作中的内存独占监视器的状态机迁移，可以知道在其他 CPU 上对 exclusive 状态的内存执行 store/store-exclusive 指令可以清零 global monitor。在 spinlock 的代码中也是使用的这种方式。在 CPU B 获取锁失败调用 wfe 之前会先调用 ldaxrh 指令将 lock->owner 地址区域设置 global monitor，当 CPU A 释放锁的时候会调用 stlrh 指令更新 lock->owner，同时会清零 global monitor，从而唤醒 CPU B。从这里也可以看到，CPU A 真正 spin 在这里等待的是 lock->owner 的值是否更新。

1.3 取号自旋锁的缺陷

取号自旋锁有效解决了多个 CPU 无序争抢锁的问题，大家按照先来后到的顺序有序取锁，除了取号的时候会存在竞争，这是基于原子操作的原理所无法避免的，多个 CPU 同时过来抢号的情况本来就很少见。但是取号自旋锁并不是没有缺陷，从 wfe 的机制可以看出来，只要 lock->owner 被更新，所有等锁的 CPU 都会从低功耗状态醒来，然后判断有没有排到自己，然而最终只会有一个 CPU 是真正能拿到锁的，也就是说其实只需要唤醒他就可以，这样所有等锁的 CPU 全部唤醒一方面浪费了系统资源。同时由于在取锁之前会把 lock 加载到 L1 cache，根据 MESI 协议此时那些等锁的 CPU 会因为内存中 lock 的值被修改而将 cacheline 标记为 invalid。如果 arm 想应用在服务器领域，系统核心数较多，多个 CPU 在等锁的情况下，系统的性能会受到较大的拖累。其中一种解决方案是 ARMV8.1 开始引入了原子加指令（ldadd/stadd）和 CAS 指令，也就是读-加法-写操作用

同一个指令实现，省去了读数据到 L1 cache 的操作，也就不存在 invalid cacheline 的过程。另外从软件层面，理想的情况应该是轮到哪个 CPU 取锁的时候只唤醒他就可以了，其他等锁的 CPU 应该毫不知情，静静地睡眠。很自然的可以想到使用包含链表的结构体来组织各个前来取锁的 CPU，一个 CPU 释放锁之后直接通知下一个来取锁就可以了，其他 CPU 并不会从低功耗状态被唤醒，为此 linux 引入了 mcs(两个人名的缩写 Mellor-Crummey and Scott)自旋锁。

二、mcs 自旋锁

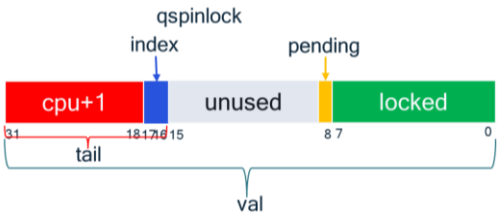


mcs 自旋锁数据结构

Mcs_spinlock 实际上就是一个链表，每个链表节点的 locked 值表明是否可以去拿锁，next 指向下一个链表节点。在取锁的时候，如果链表中只有自己一个节点，说明锁空闲，成功拿到锁，并且把节点置为 NULL。如果链表不为空，把自己添加到链表尾部，然后调用 wfe，进入低功耗状态，直到本节点的 locked 值被置 1。在释放锁的时候，会把下一个节点的 locked 值置 1，从而唤醒下一个等锁的 CPU。这样有效的解决了取号自旋锁在释放的时候会唤醒所有等锁的 CPU 的问题。但是 mcs_spinlock 的结构体占用了 16 字节，而 linux 内核中现有的自旋锁的结构体只占用 4 字节，大量的内核代码中的 spinlock 如果替换的话内存的消耗会显著增加，显然是不现实的。另外有些对结构体大小有严格要求的结构体（比如 struct page），如果替换的话改动会比较大。实际上在内核中搜索 mcs_spin_lock 接口会发现仅仅有定义而没有调用的地方，也就是说没有得到真正的使用。其实 mcs spinlock 是隐藏在 qspinlock 中的，qspinlock 在 mcs spinlock 的基础上，适当改造之后实现了对原有代码的兼容。

三、qspinlock

为了与原有的 spinlock 兼容，qspinlock 结构体同样是 4 字节长度，但是把这 4 字节分成了三段，bit 0 ~ bit 7 是 locked 成员，表明锁当前是否被占用；bit 8 ~ bit 15 是 pending 成员，其实真正用到的是 bit 8，表明当前锁为 pending 状态，一般第二个取锁的 CPU 才会置 pending 位为 1；bit 16 ~ bit 31 是 tail 成员，当取锁的 CPU 大于 2 个的时候，会将此时取锁的 CPU 号做编码之后保存在这里。



qspinlock 的数据结构

下图是 queued_spin_lock 的总体流程图，总结一下锁状态的切换过程：

1）lock.val 为 0，表示锁空闲，可以直接获取到；

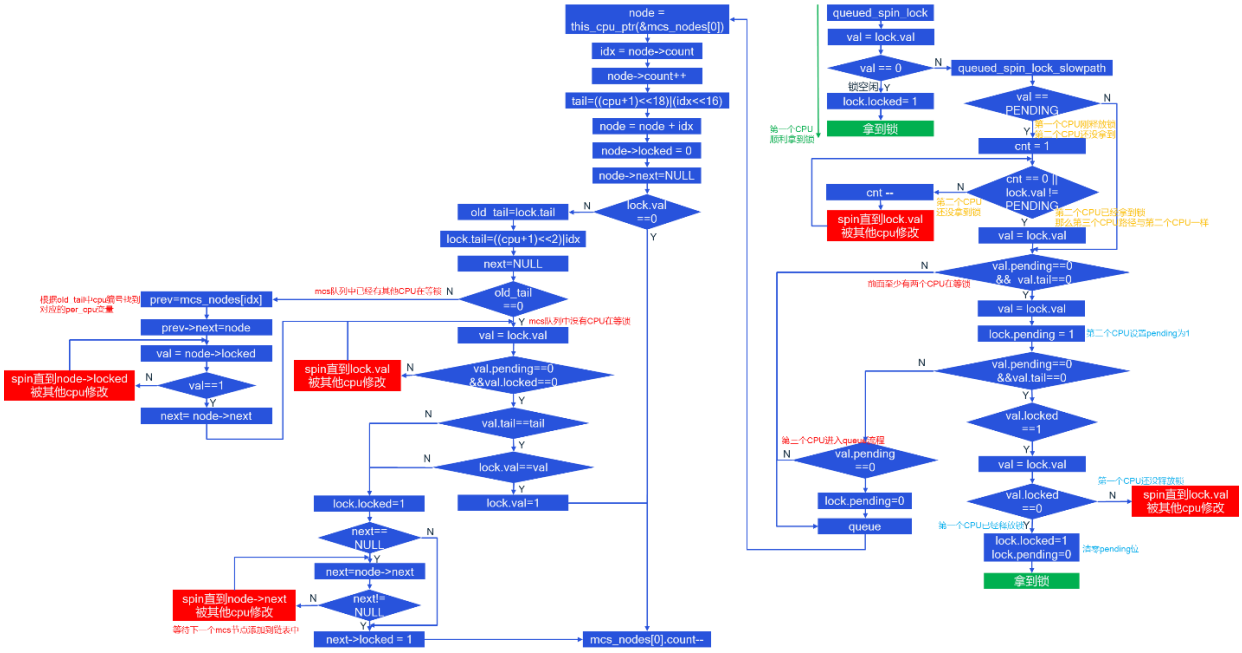
2）拿到锁之后要设置 lock.locked 为 1 表明锁被占用，释放锁要设置 lock.locked 为 0 表明其他 CPU 可以占用锁；

3）锁被占用，但是前面没有其他 CPU 在等锁的时候，要设置 lock.pending 为 1，等到拿到锁要将 lock.pending 清零；

4）锁被占用，且 lock.pending 也被置 1，那么本 CPU 会进入 mcs 队列中等锁，将 CPU 号编码到 lock.tail 中；

5）锁被占用，且 lock.tail 不为 0，表明 mcs 队列中已经有 CPU 在排队，需要把本 CPU 添加到 mcs node 链表中；

6）mcs 队列头部的 CPU 拿到锁的时候会设置下一个 mcs 节点的 locked 值为 1，唤醒下一个 CPU，提示它已经到队列头部，下一个就轮到它拿到锁了。



总体流程图

下面来看具体的场景，为了描述方便，我们用（tail，pending，locked）的格式来说

明锁的状态，例如（0，0，1）表示 tail=0，pending=0，locked=1。

1）当 lock->val 的值为 0，即锁的状态是（0，0，0）的时候，表明没有被占用，如果这时候 CPU A 来取锁，可以成功拿到锁，同时会设置 locked 值为 1，锁的状态变为（0，0，1），表明锁已经被占用。

2）CPU B 来取锁，由于 lock->val 不为 0，会进入慢速取锁流程，设置 pending 值为 1，锁的状态变为（0，1，1）。由于 locked 值为 1，会调用 wfe 进入低功耗状态，直到 lock->val 的值被修改才会被唤醒。

3）CPU N 来取锁，同样的因为 lock->val 不为 0，会进入慢速取锁流程。首先判断 pending 和 tail 中是否都是 0，由于此时 pending 为 1，CPU N 会进入 queue 流程，接下来就是 mcs spinlock 的实现。在系统初始化的时候，qspinlock 定义了 per-cpu 数组 mcs_spinlock mcs_node[4]，在 queue 流程中本次需要的 mcs_spinlock 节点只需要从本地 cpu 的 mcs_node 数组中获取就可以了，省去了内存分配的过程，提高效率。为什么 mcs_node 数组有四个元素，因为在 linux 中有四种上下文，task，irq，softirq 和 nmi，而每一种上下文最多获取一次 spinlock，原因下文会说。其中 nmi 是基于 x86 架构的，arm 其实只支持前三种。另外 mcs_node[0].count 用来记录当前 CPU 有几个上下文在取锁，每来一次就会加 1，释放锁之后会减 1。再来回顾前文说的 qspinlock 把 32 位 lock->val 分成三个域，bit 16 ~ bit 31 是 tail 域，其中的 bit 16 ~ bit 17 两个 bit 是 mcs_nodes[0]->count，也就是当前 mcs spinlock 节点在 mcs_nodes 数组中的索引值；bit 18 ~ bit 31 是 CPU 编号加 1。为什么要加 1？因为 CPU 编号是从 0 开始的，如果此时 CPU0 没有其他上下文在等锁，那么 idx 也是 0，那么 tail 的值就是 0，而 tail 为 0 表示在 mcs spinlock 中还没有节点，显然是矛盾的。CPU N 先根据 idx 将 mcs_node[n]赋值给当前的 mcs spinlock 节点，设置 node->locked=0，node->next=NULL。之后把 idx 和 cpu+1 更新到 lock->tail 中。此时锁的状态转变为（n，1，1）。接下来会判断 locked 和 pending 的值是否均为 0，由于 locked 和 pending 都为 1，CPU N 会调用 wfe 进入低功耗状态，直到 locked 或者 pending 被其他 CPU 修改才会重新唤醒。

4）CPU K 来取锁，同样的因为 lock->val 不为 0，会进入慢速取锁流程。首先判断 pending 和 tail 中是否都是 0，由于此时 pending 和 locked 都为 1，CPU K 会进入 queue 流程。CPU N 和 CPU K 一样会把 idx 和 CPU+1 的值写入 lock->tail，此时锁的状态转变为（k，1，1）。此外，还会判断 lock->tail 是否为 0，由于 CPU N 此前已经设置了 lock->tail 的值为 n，表明 mcs spinlock 中已经有节点，CPU K 会把自己的 mcs spinlock 节点添加到 mcs 链表尾部。这里就有一个问题，怎么找到前一个 mcs spinlock 节点？注意锁状态的切换过程，由于每个取锁的 CPU 会把 CPU 编号加 1 保存在 lock->val 的 bit 17 ~ bit 31，所以在本 CPU 更新 lock->tail 之前要把原来的值保存下来，从里面解码出上一个取锁的

CPU 和 idx，再根据 per-cpu 接口调用 per_cpu_ptr 读取上一个 CPU 的 mcs_nodes[idx_n]。添加完 mcs spinlock 之后 CPU K 会判断自己的 mcs spinlock 节点的 locked 值是否为 1，由于初始化的值为 0，所以 CPU K 会调用 wfe 进入低功耗状态，直到自己的 mcs spinlock 节点的 locked 值被更新才会被唤醒。之后更多的 CPU 来取锁都和 CPU K 的流程是一样的。从第四个来取锁的 CPU 即 CPUK 开始，就仅仅在自己的 mcs_spinlock->locked 上自旋，lock->val 的值变化不会唤醒这些等锁的 CPU。

再看解锁的过程，解锁本身的代码很简单，仅仅是把 lock->locked 设置为 0。具体的解锁对各 CPU 的影响过程如下：

1）CPU A 执行完临界区的代码之后释放锁，把 lock->locked 清零，这时候 CPU B 和 CPU N 会被唤醒，锁的状态变为（k，1，0）。CPU B 终于等到了 locked 值变为 0，设置 locked 为 1，pending 为 0，拿到锁，锁定状态变为(k,0,1)，进入临界区。CPU N 虽然也被唤醒了，但是他等待的是 locked 和 pending 都为 0 才能拿到锁，所以会再次进入低功耗状态。

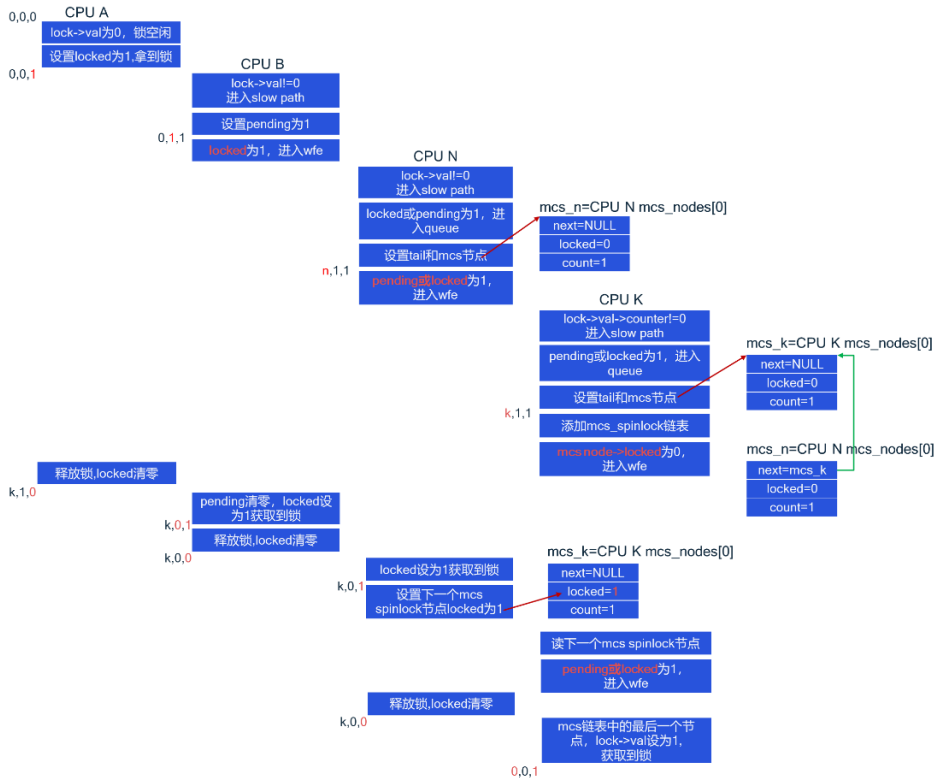
2）CPU B 释放锁的时候，同样把 locked 值清零，这时候 CPN 再次被唤醒，锁的状态变为（k，0，0）。这次 CPU N 等到了 locked 和 pending 均为 0，会设置锁的 locked 为 1，此时锁的状态变为（k，0，1）。

3）CPU N 进入临界区之前会设置下一个 mcs spinlock 节点（即 CPU K 的节点）的 locked 值为 1。CPU K 的 mcs_spinlock->locked 被置 1 后就结束了在 mcs spinlock 队列中的自旋，表明已经排到了队首，接下来只需要等待 lock->pending 和 lock->locked 都为 0 就可以拿到锁。

4）CPU N 执行完临界区的代码之后释放锁，设置 locked 为 0，锁的状态变为（k，0，0）。由于此时 mcs 队列中没有其他节点，CPU K 清零 tail 值，并设置 locked 值为 1，锁的状态变为（0，0，1），拿到锁进入临界区。

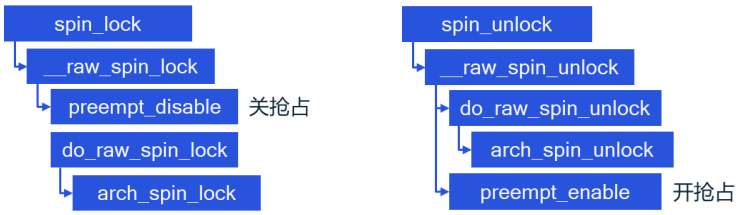
可以看到，在 mcs spinlock 的等待队列中，从第二个节点开始，每个 CPU 会被唤醒两次，第一次唤醒是在前一个 mcs 节点的 CPU 拿到锁的时候设置本 mcs 节点的 locked 值为 1，表明自己排到了 mcs spinlock 的队列头部。接下来该 CPU 会进入低功耗，直到 lock->locked 和 lock->pending 都为 0。第二次唤醒是上一个 CPU 释放锁的时候清零 lock->locked，这时才可以拿到锁。

下图描述了上述的加锁和解锁的流程，锁的状态变化用红色标出。



四、spinlock 相关的 API

前面介绍的是基于 ARM64 的架构层的 spinlock 的实现 arch_spin_lock，在内核编程中实际使用的接口到 ARCH 层还有很多工作要做。我们针对 SMP 系统从最基本的 API spin_lock/spin_unlock 开始研究。



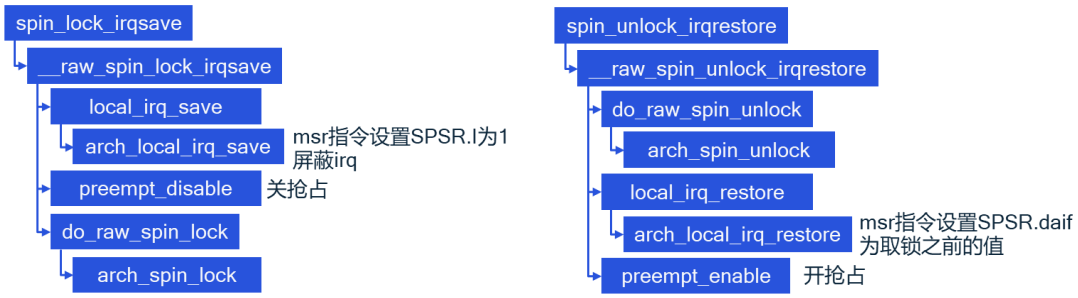
spin_lock/spin_unlock

在调用 arch_spin_lock 取锁之前会关抢占(thread_info->preempt_count+1)，不允许等锁线程所在的 CPU 上发生线程切换，这是为了保证其他 CPU 释放锁之后当前等锁的线程可以尽快拿到锁。这也是为什么说等自旋锁的线程处于忙等状态的原因，该 CPU 上其他线程此时没有机会得到调度。同时也暗含了拿到 spinlock 之后的临界区代码要尽快执行完，避免对系统性能造成影响。在调用 arch_spin_unlock 释放锁之后会把 thread_info->preempt_count-1，由于 spinlock 是可以嵌套的，当线程持有的所有锁都释放的时候 preempt_count 值为 0，此时如果线程的 thread_info 中的 flag TIF_NEED_RESCHED 被设置为 1，说明需要让出 CPU，此时会发生线程切换，该线程所

在的 CPU 上其他线程得到调度。

有的时候中断处理函数和内核代码会共用一些变量，如果线程拿到锁之后，该 CPU 上接收到中断，那么该线程就会切换到中断上下文中。如果中断处理函数中也试图去取这个锁，就会由于锁被占用而进入低功耗状态，而拿到锁的线程由于处于中断上下文中，它本身的代码永远不会释放锁，形成死锁。这种情况就需要另外一个 API spin_lock_irqsave/spin_unlock_irqrestore，在取锁之前先将本地 CPU 的中断屏蔽掉，不允许接收中断。需要注意的是 local_irq_save()只对本地 CPU 执行关中断操作，这包含两层意思：一是如果其他 CPU 上接收到了中断，那么这些 CPU 上的中断处理函数，也有可能试图去获取一个被本 CPU 上运行的线程占有的 spinlock。不过没有关系，因为此时中断处理函数和持锁线程运行在不同的 CPU 上，等到线程释放了这个 spinlock，中断处理函数就有机会获取到锁，不至于造成死锁。第二层意思是只要中断没有绑定到被屏蔽中断的 CPU，该中断就可以被其他 CPU 接收并正常处理，不至于出现中断被 pending 的情况。

另外还有一组 API 用于在 spinlock 中屏蔽中断：spin_lock_irq/spin_unlock_irq。与前述 API 的区别在于，获取锁的时候，spin_lock_irqsave 会把当前 SPSR 寄存器的 daif 域记录下来；释放锁的时候，spin_unlock_irqrestore 再把之前的 daif 的值恢复到 SPSR 寄存器中。而 spin_lock_irq 不会保存寄存器的值，释放锁之后的 SPSR.daif 中 I 位的值会被清零。为什么要有保存 daif 的操作呢?因为 spinlock 是可以嵌套调用的，如果调用某一 spin_lock_irq 之后，临界区的代码又调用了 spin_lock_irq 对另一个临界区加锁，那么当第二个 spinlock 调用 spin_unlock_irq 释放锁的时候中断会被 enable，前面一个 spinlock 屏蔽中断的操作就失效了。如果第一个 spinlock 临界区处于某个中断处理函数路径中，那么就会出现前述所说的死锁的情况。所以第二个 spinlock 应该调用 spin_lock_irqsave 就可以避免这种情形。



spin_lock_irqsave/spin_unlock_irqrestore

如果中断处理函数不会和线程共享变量，是不是就可以直接使用 spin_lock()呢？也不一定，这是由于中断还有下半部机制，也就是在中断处理函数中可能会产生 softirq 来继续执行一些比较耗时的操作，而中断函数本身退出执行。如果中断下半部用的是

workqueue，就需要 kworker 来执行，和进程上下文是一样的，只需要调用 spin_lock 就可以了。如果中断下半部采用的是 tasklet 或者 softirq，在中断上半部退出之后会调用 raise_softirq 产生软中断，设置本 CPU 的 irq_stat.__softirq_pending 为 1，表明有 softirq 需要处理。之后会唤醒 ksoftirqd，但是由于 spin_lock 接口关闭了抢占，所以 ksoftirqd 并不会调度，似乎与进程上下文的情形类似，不会形成死锁。然而 spin_lock 接口并没有屏蔽中断，如果此时该 CPU 又接收到一个中断，在中断退出的过程中会检测 irq_stat.__softirq_pending 是否为 1，如果是 1，就会直接在线程调用栈中调用软中断处理函数。如果软中断处理函数中与线程有共享的数据需要操作，就会形成死锁。针对这种情形，可以调用 spin_lock_bh/spin_unlock_bh 这组 API 把 preempt_count 的 SOFTIRQ 域加 1，伪造出处于中断上下文中。这样在 irq_exit 中检测到处于中断上下文，就不会执行 softirq。

Q & A:

1. 为什么加 spinlock 的临界区中不能睡眠，即调用 schedule()主动放弃 CPU。
   如果切换到其他 CPU，需要获取这个 spinlock 的话就极容易形成死锁或拖慢系统性能。

   同理，中断上下文中不允许睡眠不是技术上不能实现，而是出于性能考虑。中断处理函数一般是快速执行且紧急的任务，如果有耗时或不紧急的操作需要使用中断下半部机制 workqueue。