

```
多处理器中并发访问临界区,防止内核抢占造成的竞争。
                   原理
                         底层通过控制原子变量的值,CPU会循环读取,直到ok
                         SMP场景,当一个CPU持有自旋锁后,其他CPU只能忙等待
                   场景
                         锁不能导致睡眠 ( 硬中断上下文 )
                         临界区执行快
                                   让CPU忙等待,抢占调度占时禁止
                                          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_lock
                                          spin_unlock来保护,不同的软中断能够同时在不同的CPU上运行。
                                   场景
                                           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
                                     spin+禁止中断 屏蔽中断包括硬中断和软中断,只能屏蔽当前CPU
                                            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。
                         spin_lock_irq
                                            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
                                     场景
                   API
                                           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
                                           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
                                        spin+屏蔽中断并保存当前中断状态,只能屏蔽当前CPU
                         spin_lock_irqsave
                                               如果可以不确信在对共享资源访问前中断是否使能,那么使用spin_lock_irgsave和
                                        场景
                                              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
                                     spin+屏蔽软中断
                                            如果只在进程上下文和软中断上下文访问临界资源,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
自旋锁 spin_lock
                         spin_lock_bh
                                            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
                                     场景
                                           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。
                           一般要求在临界区执行要快,否则影响系统性能
                           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
                           自旋锁可以导致死锁,如果一个CPU获取自旋锁,如果再次获取会导致死锁,一般都是中断导致,所以
                           在单CPU中要禁止中断。
                   注意点
                           有的操作系统里,提供不顺带关中断的spinlock,这种spinlock是危险的,因为会
                           发生死锁(任务A take了spinlock,发生中断,中断里面又take了spinlock,失败
                           了,死锁了),这种情况下理论上是可以产生调度的,但因为这种spinlock关了抢
                           占,别的任务又不能跑,这个核实际上是浪费了。所以我见到的大部分spinlock,
                           都是顺带关了中断的
                             __raw_spin_lock
                                          __raw_spin_lock
                   内核实现
                            MCS Spinlock
                                           MCS Spinlock 申请操作描述如下
                            qspinlock
                                      queued_spin_lock_slowpath
                         https://blog.csdn.net/edison0716/article/details/5410430
                   2、writer thread的操作
                  对于writer thread,获取seqlock操作如下:
                   (1)获取锁(例如spin lock),该锁确保临界区只有一个writer进入。
                   (2) sequence counter加一
                  释放seqlock操作如下
                  (1)释放锁,允许其他writer thread进入临界区。
                   (2)sequence counter加一(注意:不是减一哦,sequence counter是一个不
                  断累加的counter)
                  由上面的操作可知,如果临界区没有任何的writer thread,那么sequence
                  counter是偶数(sequence counter初始化为0),如果临界区有一个writer
                  thread(当然,也只能有一个),那么sequence counter是奇数。
                  3、reader thread的操作如下:
                   (1)获取sequence counter的值,如果是偶数,可以进入临界区,如果是奇数
                   那么等待writer离开临界区(sequence counter变成偶数)。进入临界区时候的
                  sequence counter的值我们称之old sequence counter。
                   (2)进入临界区,读取数据
                   (3)获取sequence counter的值,如果等于
                  总结一下seqlock的特点:临界区只允许一个writer thread进入,在没有writer thread的情况下
                  reader thread可以随意进入,也就是说reader不会阻挡reader。在临界区只有有reader thread的情况
                  下, writer thread可以立刻执行, 不会等待。
                  1) read操作比较频繁
                  2) write操作较少,但是性能要求高,不希望被reader thread阻挡(之所以要求write操作较少主要是
                  考虑read side的性能)
                  3)数据类型比较简单,但是数据的访问又无法利用原子操作来保护。
                   seqlock这种锁机制是倾向writer thread,也就是说,除非有其他的writer thread进入了临界区,否则
                  它会长驱直入,无论有多少的reader thread都不能阻挡writer的脚步。writer thread这么霸道,reader
                  肿么办?对于seqlock , reader这一侧需要进行数据访问的过程中检测是否有并发的writer thread操
                  作,如果检测到并发的writer,那么重新read。通过不断的retry,直到reader thread在临界区的时
                  候,没有任何的writer thread插入即可。这样的设计对reader而言不是很公平,特别是如果writer
                  thread负荷比较重的时候,reader thread可能会retry多次,从而导致reader thread这一侧性能的下
seqlock
                     在kernel中,jiffies_64保存了从系统启动以来的tick数目,对该数据的访问(以及其他jiffies相关数
                     据)需要持有jiffies_lock这个seq lock
                                          u64 get_jiffies_64(void)
                                             seq = read_seqbegin(&jiffies_lock);
                                            ret = jiffies_64;
                                           } while (read_seqretry(&jiffies_lock, seq));
            API示例
                    1、reader side代码如下
                                                static void tick_do_update_jiffies64(ktime_t now)
                                                 write_seqlock(&jiffies_lock);
                                                临界区会修改jiffies_64等相关变量,具体代码略
                                                 write_sequnlock(&jiffies_lock);
                    2、writer side代码如下
                        通过底层架构相关原子操作实现
                 原理
                        可以同时读,但读写,写写互斥
                 场景
                        针对读多写少的场景
读写锁 rwlock
                 注意点
                         可以配合中断使能与关闭使用
                 API
               RCU(Read-Copy Update),顾名思义就是读-拷贝修改,它是基于其原理命名的。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
         原理
               RCU 技术的核心是写操作分为写和更新两步,允许读操作在任何时候无阻碍的运行,换句话说,就是
                通过延迟写来提高同步性能
               是一个改进的rwlock,适合读多写少的场景
         场景
                 延迟释放数据结构,复制数据结构
                 读者的CPU不能被阻塞(不能上下文切换)
                 多个写者之间需要其他同步机制
         注意点
RCU
                        延后的删除或释放将占用一些内存,尤其是对嵌入式系统,这可能是非常昂贵的内存开销。此外,写者
                        的开销比较大,尤其是对于那些无法容忍旧数据的情况以及不只一个写者的情况,写者需要spinlock或
                      其他的锁机制来与其他写者同步
                https://cloud.tencent.com/developer/article/1006204
               子主题
         实现
               子主题
               子主题
           原理
                  基于底层原子操作实现互斥访问
           场景
                  在进程上下文中互斥访问变量
                   会导致其他进程或者线程堵塞
mutex
                   不能用在中断上下文
                   同一时间只能有一个任务持有互斥锁,而且只有这个任务可以对互斥锁进行解锁。互斥锁不能进行递归
                   锁定或解锁。一个互斥锁对象必须通过其API初始化,而不能使用memset或复制初始化。一个任务在
                   持有互斥锁的时候是不能结束的。互斥锁所使用的内存区域是不能被释放的。使用中的互斥锁是不能被
                   重新初始化的。并且互斥锁不能用于中断上下文
                   互斥锁比当前的内核信号量选项更快
                    基于底层原子API实现,实现多对一访问
                    生产者和消费者问题,队列访问
              场景
                    进程间访问共享资源
信号量sem
                      只有可以睡眠的函数才能获取内核信号量;中断处理程序和可延迟函数(软中断)都不能使用内核
              注意点
                     会导致堵塞的进程睡眠
                      一个线程占用,其他线程释放。
                      completion是一种轻量级同步机制,一个线程告诉另一个线程某个工作已完成
                      采用等待队列唤醒等待线程,自旋锁控制done变量
```

场景

注意点

completion

需要同步等待场景

咱们可以总结下自旋锁的特点:

单处理器非抢占内核下:自旋锁会在编译时被忽略;

● 单处理器抢占内核下:自旋锁仅仅当作一个设置内核抢占的开关;

● 多处理器下:此时才能完全发挥出自旋锁的作用,自旋锁在内核中主要用来防止