内核同步方法（自旋锁：spinlock_t）

一、基础部分

Linux 的的内核最常见的锁是自旋锁。自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个被已经持有（争用）的自旋锁，那么该线程就会一直进行忙循环-旋转-等待锁重新可用要是锁未被争用，请求锁的执行线程就可以立即得到它，继续执行。

在任意时间，自旋锁都可以防止多于一个的执行线程同时进入临界区。同一个锁可以用在多个位置，例如，对于给定数据的所有访问都可以得到保护和同步。

Linux 内核自旋锁基本结构体 spinlock_t 源码如下，有时间大家可以研究分析一下：

include > linux > C spinlock_types.h > spinlock_t

```
61  typedef struct spinlock {
62      union {
63          struct raw_spinlock rlock;
64
65  #ifdef CONFIG_DEBUG_LOCK_ALLOC
66  # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
67          struct {
68              u8 __padding[LOCK_PADSIZE];
69              struct lockdep_map dep_map;
70          };
71  #endif
72      };
73  } spinlock_t;
74
```

二、Spinlock 进阶

内核当发生访问资源冲突的时候，可以有两种锁的解决方案选择：

- 一个是原地等待
- 一个是挂起当前进程，调度其他进程执行（睡眠）

Spinlock 是内核中提供的一种比较常见的锁机制，自旋锁是“原地等待”的方式解决资源冲突的，即，一个线程获取了一个自旋锁后，另外一个线程期望获取该自旋锁，获取不到，只能够原地“打转”（忙等待）。由于自旋锁的这个

忙等待的特性，注定了它使用场景上的限制 ——自旋锁不应该被长时间的持有（消耗 CPU 资源）。**中断上下文要用锁，首选 spinlock**。

三、自旋锁的死锁和解决

自旋锁不可递归，自己等待自己已经获取的锁，会导致死锁。自旋锁可以在中断上下文中使用，但是试想一个场景：一个线程获取了一个锁，但是被中断处理程序打断，中断处理程序也获取了这个锁（但是之前已经被锁住了，无法获取到，只能自旋），中断无法退出，导致线程中后面释放锁的代码无法被执行，导致死锁。（如果确认中断中不会访问和线程中同一个锁，其实无所谓）

A、考虑下面的场景（内核抢占场景）：

（1）进程 A 在某个系统调用过程中访问了共享资源 R

（2）进程 B 在某个系统调用过程中也访问了共享资源 R

会不会造成冲突呢？假设在 A 访问共享资源 R 的过程中发生了中断，中断唤醒了沉睡中的，优先级更高的 B，在中断返回现场的时候，发生进程切换，B 启动执行，并通过系统调用访问了 R，如果没有锁保护，则会出现两个 thread 进入临界区，导致程序执行不正确。OK，我们加上 spin lock 看看如何：A 在进入临界区之前获取了 spin lock，同样的，在 A 访问共享资源 R 的过程中发生了中断，中断唤醒了沉睡中的，优先级更高的 B，B 在访问临界区之前仍然会试图获取 spin lock，这时候由于 A 进程持有 spin lock 而导致 B 进程进入了永久的 spin……怎么破？linux 的 kernel 很简单，在 A 进程**获取 spin lock 的时候，禁止本 CPU 上的抢占**（上面的永久 spin 的场合仅仅在本 CPU 的进程抢占本 CPU 的当前进程这样的场景中发生）。如果 A 和 B 运行在不同的 CPU 上，那么情况会简单一些：A 进程虽然持有 spin lock 而导致 B 进程进入 spin 状态，不过由于运行在不同的 CPU 上，A 进程会持续执行并会很快释放 spin lock，解除 B 进程的 spin 状态

B、再考虑下面的场景（中断上下文场景）：

（1）运行在 CPU0 上的进程 A 在某个系统调用过程中访问了共享资源 R

（2）运行在 CPU1 上的进程 B 在某个系统调用过程中也访问了共享资源 R

（3）外设 P 的中断 handler 中也会访问共享资源 R

在这样的场景下，使用 spin lock 可以保护访问共享资源 R 的临界区吗？我们假设 CPU0 上的进程 A 持有 spin lock 进入临界区，这时候，外设 P 发生了中断事件，并且调度到了 CPU1 上执行，看起来没有什么问题，执行在 CPU1 上的 handler 会稍微等待一会 CPU0 上的进程 A，等它立刻临界区就会释放 spin lock 的，但是，如果外设 P 的中断事件被调度到了 CPU0 上执行会怎么样？CPU0 上的进程 A 在持有 spin lock 的状态下被中断上下文抢占，而抢占它的 CPU0 上的 handler 在进入临界区之前仍然会试图获取 spin lock，悲剧发生了，CPU0 上的 P 外设的中断 handler 永远的进入 spin 状态，这时候，CPU1 上的进程 B 也不可避免在试图持有 spin lock 的时候失败而导致进入 spin 状态。为了解决这样的问题，linux kernel 采用了这样的办法：**如果涉及到中断上下文的访问，spin lock 需要和禁止本 CPU 上的中断联合使用**。

C、再考虑下面的场景（底半部场景）

linux kernel 中提供了丰富的 bottom half 的机制，虽然同属中断上下文，不过还是稍有不同。我们可以把上面的场景简单修改一下：外设 P 不是中断 handler 中访问共享资源 R，而是在的 bottom half 中访问。使用 spin lock+禁止本地中断当然是可以达到保护共享资源的效果，但是使用牛刀来杀鸡似乎有点小题大做，这时候 **disable bottom half** 就 OK 了

D、中断上下文之间的竞争

同一种中断 handler 之间在 uni core 和 multi core 上都不会并行执行，这是 linux kernel 的特性。如果不同中断 handler 需要使用 spin lock 保护共享

资源，对于新的内核（不区分 fast handler 和 slow handler），所有 handler 都是关闭中断的，因此使用 spin lock 不需要关闭中断的配合。

bottom half 又分成 softirq 和 tasklet，同一种 softirq 会在不同的 CPU 上并发执行，**因此如果某个驱动中的 softirq 的 handler 中会访问某个全局变量，对该全局变量是需要使用 spin lock 保护的，不用配合 disable CPU 中断或者 bottom half**。

tasklet 更简单，因为同一种 tasklet 不会多个 CPU 上并发。