

Linux内核中的互斥

wheelz

March 28, 2006

需要澄清的是，互斥手段的选择，不是根据临界区的大小，而是根据临界区的性质，以及有哪些部分的代码，即哪些内核执行路径来争夺。

从严格意义上说，semaphore和spinlock_XXX属于不同层次的互斥手段，前者的实现有赖于后者，这有点象HTTP和TCP的关系，都是协议，但层次是不同的。

先说semaphore，它是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份，代表进程来争夺资源的。如果竞争不上，会有context switch，进程可以去sleep，但CPU不会停，会接着运行其他的执行路径。从概念上说，这和单CPU或多CPU没有直接的关系，只是在semaphore本身的实现上，为了保证semaphore结构存取的原子性，在多CPU中需要spinlock来互斥。

在内核中，更多的是要保持内核各个执行路径之间的数据访问互斥，这是最基本的互斥问题，即保持数据修改的原子性。semaphore的实现，也要依赖这个。在单CPU中，主要是中断和bottom_half的问题，因此，开关中断就可以了。在多CPU中，又加上了其他CPU的干扰，因此需要spinlock来帮助。这两个部分结合起来，就形成了spinlock_XXX。它的特点是，一旦CPU进入了spinlock_XXX，它就不会干别的，而是一直空转，直到锁定成功为止。因此，这就决定了被spinlock_XXX锁住的临界区不能停，更不能context switch，要存取完数据后赶快出来，以便其他的在空转的执行路径能够获得spinlock。这也是spinlock的原则所在。如果当前执行路径一定要进行context switch，那就要在schedule()之前释放spinlock，否则，容易死锁。因为在中断和bh中，没有context，无法进行context switch，只能空转等待spinlock，你context switch走了，谁知道猴年马月才能回来。

因为spinlock的原意和目的就是保证数据修改的原子性，因此也没有理由在spinlock锁住的临界区中停留。

spinlock_XXX有很多形式，有

```
spin_lock()/spin_unlock(),
spin_lock_irq()/spin_unlock_irq(),
spin_lock_irqsave()/spin_unlock_irqrestore()
spin_lock_bh()/spin_unlock_bh()
```

```
local_irq_disable()/local_irq_enable
local_bh_disable()/local_bh_enable
```

那么，在什么情况下具体用哪个呢？这要看是在什么内核执行路径中，以及要与哪些内核执行路径相互斥。我们知道，内核中的执行路径主要有：

1. 用户进程的内核态，此时有进程context，主要是代表进程在执行系统调用等。
2. 中断或者异常或者自陷等，从概念上说，此时没有进程context，不能进行 context switch。
3. bottom_half，从概念上说，此时也没有进程context。
4. 同时，相同的执行路径还可能在其他的CPU上运行。

这样，考虑这四个方面的因素，通过判断我们要互斥的数据会被这四个因素中的哪几个来存取，就可以决定具体使用哪种形式的spinlock。

1. 如果只要和其他CPU互斥，
就要用spin_lock和spin_unlock
2. 如果要和irq及其他CPU互斥，
就要用spin_lock_irq和spin_unlock_irq
3. 如果既要和irq及其他CPU互斥，又要保存EFLAG的状态，
就要用spin_lock_irqsave和spin_unlock_irqrestore
4. 如果要和bh及其他CPU互斥，
就要用spin_lock_bh和spin_unlock_bh
5. 如果不需要和其他CPU互斥，只要和irq互斥，
则用local_irq_disable和local_irq_enable

6. 如果不需要和其他CPU互斥，只要和bh互斥，
则用local_bh_disable和local_bh_enable

值得指出的是，对同一个数据的互斥，在不同的内核执行路径中，所用的形式有可能不同(见下面的例子)。

举一个例子。在中断部分中有一个irq_desc_t类型的结构数组变量irq_desc[]，该数组每个成员对应一个irq的描述结构，里面有该irq的响应函数等。在irq_desc_t结构中有一个spinlock，用来保证存取(修改)的互斥。

对于具体一个irq成员，irq_desc[irq]，对其存取的内核执行路径有两个，一是在设置该irq的响应函数时(setup_irq)，这通常发生在module的初始化阶段，或系统的初始化阶段；二是在中断响应函数中(do_IRQ)。代码如下：

```
int setup_irq(unsigned int irq, struct irqaction * new)
{
    int shared = 0;
    unsigned long flags;
    struct irqaction *old, **p;
    irq_desc_t *desc = irq_desc + irq;

    /*
     * Some drivers like serial.c use request_irq() heavily,
     * so we have to be careful not to interfere with a
     * running system.
     */
    if (new->flags & SA_SAMPLE_RANDOM) {
        /*
         * This function might sleep, we want to call it first,
         * outside of the atomic block.
         * Yes, this might clear the entropy pool if the wrong
         * driver is attempted to be loaded, without actually
         * installing a new handler, but is this really a problem,
         * only the sysadmin is able to do this.
         */
        rand_initialize_irq(irq);
    }

    /*
     * The following block of code has to be executed atomically
     */
    [1] spin_lock_irqsave(&desc->lock, flags);
```

```

p = &desc->action;
if ((old = *p) != NULL) {
    /* Can't share interrupts unless both agree to */
    if (!(old->flags & new->flags & SA_SHIRQ)) {
[2]         spin_unlock_irqrestore(&desc->lock,flags);
            return -EBUSY;
        }

        /* add new interrupt at end of irq queue */
        do {
            p = &old->next;
            old = *p;
        } while (old);
        shared = 1;
    }

    *p = new;

    if (!shared) {
        desc->depth = 0;
        desc->status &= ~(IRQ_DISABLED | IRQ_AUTODETECT | IRQ_WAITING);
        desc->handler->startup(irq);
    }
[3] spin_unlock_irqrestore(&desc->lock,flags);

    register_irq_proc(irq);
    return 0;
}

asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
    /*
     * We ack quickly, we don't want the irq controller
     * thinking we're snobs just because some other CPU has
     * disabled global interrupts (we have already done the
     * INT_ACK cycles, it's too late to try to pretend to the
     * controller that we aren't taking the interrupt).
     *
     * 0 return value means that this irq is already being
     * handled by some other CPU. (or is disabled)
     */
}

```

```

int irq = regs.orig_eax & 0xff; /* high bits used in ret_from_code */
int cpu = smp_processor_id();
irq_desc_t *desc = irq_desc + irq;
struct irqaction * action;
unsigned int status;

kstat.irqs[cpu][irq]++;
[4] spin_lock(&desc->lock);
desc->handler->ack(irq);
/*
    REPLAY is when Linux resends an IRQ that was dropped earlier
    WAITING is used by probe to mark irqs that are being tested
*/
status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
status |= IRQ_PENDING; /* we _want_ to handle it */

/*
    * If the IRQ is disabled for whatever reason, we cannot
    * use the action we have.
    */
action = NULL;
if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS))) {
    action = desc->action;
    status &= ~IRQ_PENDING; /* we commit to handling */
    status |= IRQ_INPROGRESS; /* we are handling it */
}
desc->status = status;

/*
    * If there is no IRQ handler or it was disabled, exit early.
    * Since we set PENDING, if another processor is handling
    * a different instance of this same irq, the other processor
    * will take care of it.
    */
if (!action)
    goto out;

/*
    * Edge triggered interrupts need to remember
    * pending events.
    * This applies to any hw interrupts that allow a second

```

```

    * instance of the same irq to arrive while we are in do_IRQ
    * or in the handler. But the code here only handles the _second_
    * instance of the irq, not the third or fourth. So it is mostly
    * useful for irq hardware that does not mask cleanly in an
    * SMP environment.
    */
for (;;) {
[5]     spin_unlock(&desc->lock);
        handle_IRQ_event(irq, &regs, action);
[6]     spin_lock(&desc->lock);

        if (!(desc->status & IRQ_PENDING))
            break;
        desc->status &= ~IRQ_PENDING;
    }
    desc->status &= ~IRQ_INPROGRESS;
out:
    /*
     * The ->end() handler has to deal with interrupts which got
     * disabled while the handler was running.
     */
    desc->handler->end(irq);
[7]    spin_unlock(&desc->lock);

    if (softirq_pending(cpu))
        do_softirq();
    return 1;
}

```

在`setup_irq()`中，因为其他CPU可能同时在运行`setup_irq()`，或者在运行`setup_irq()`时，本地irq中断来了，要执行`do_IRQ()`以修改`desc->status`。为了同时防止来自其他CPU和本地irq中断的干扰，如[1][2][3]处所示，使用了`spin_lock_irqsave/spin_unlock_irqrestore()`

而在`do_IRQ()`中，因为`do_IRQ()`本身是在中断中，而且此时还没有开中断，本CPU中没有 什么可以中断其运行，其他CPU则有可能在运行`setup_irq()`，或者也在中断中，但这二者 对本地`do_IRQ()`的影响没有区别，都是来自其他CPU的干扰，因此只需要用`spin_lock/spin_unlock`，如[4][5][6][7]处所示。值得注意的是[5]处，先释放该spinlock，再调用具体的响应函数。

再举个例子：

```
static void tasklet_hi_action(struct softirq_action *a)
```

```

{
    int cpu = smp_processor_id();
    struct tasklet_struct *list;

[8]    local_irq_disable();
        list = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = NULL;
[9]    local_irq_enable();

        while (list) {
            struct tasklet_struct *t = list;

            list = list->next;

            if (tasklet_trylock(t)) {
                if (!atomic_read(&t->count)) {
                    if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                        BUG();
                    t->func(t->data);
                    tasklet_unlock(t);
                    continue;
                }
                tasklet_unlock(t);
            }

[10]        local_irq_disable();
            t->next = tasklet_hi_vec[cpu].list;
            tasklet_hi_vec[cpu].list = t;
            __cpu_raise_softirq(cpu, HI_SOFTIRQ);
[11]        local_irq_enable();
        }
}

```

这里，对`tasklet_hi_vec[cpu]`的修改，不存在CPU之间的竞争，因为每个CPU有各自独立的数据，所以只要防止irq的干扰，用`local_irq_disable/local_irq_enable`可，如[8][9][10][11]处所示。