内核同步

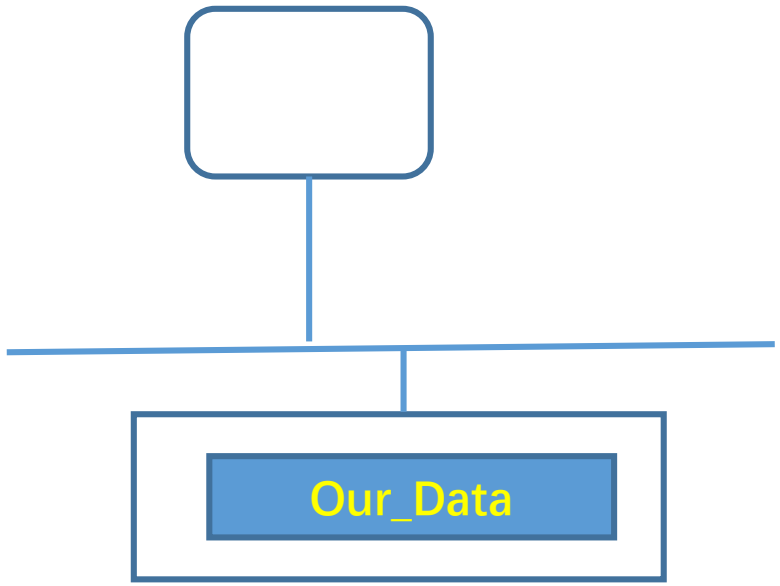
1、变量的互斥访问　　这里有一个数据结构 Our_Data，每次访问这个数据结构的时候，内核控制路径会执行连续的若干条指令。我们必须保证这些指令一次执行完毕，其间不可以让其它内核控制路径访问 Our_Data。

临界区是访问Our_Data的代码片段。

2、同步　　一个操作执行完毕后，才可以执行另一个操作。

一个可能会有很多内核控制路径访问的内核全局变量

UP系统



如果临界区很短 & 临界区不可能有入睡操作，怎样保护?

关中断

实现（不用锁）：

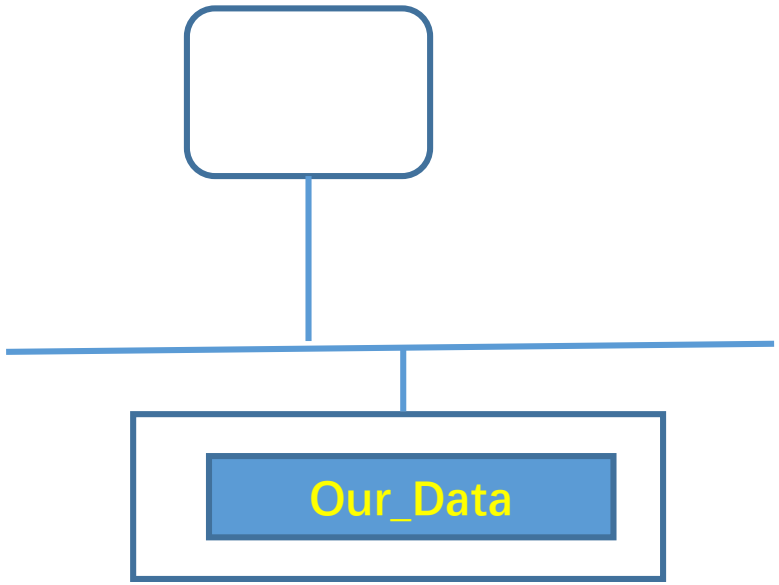
CLI

访问、修改Our_Data；

STI

这是防止中断处理程序插入执行，修改这个数据结构

UP系统



临界区入睡，~?

访问数据结构前上锁。访问结束开锁。

如果因为等待锁开而入睡，唤醒之后的第一件事：看锁开嘛？不开的话，就再睡。

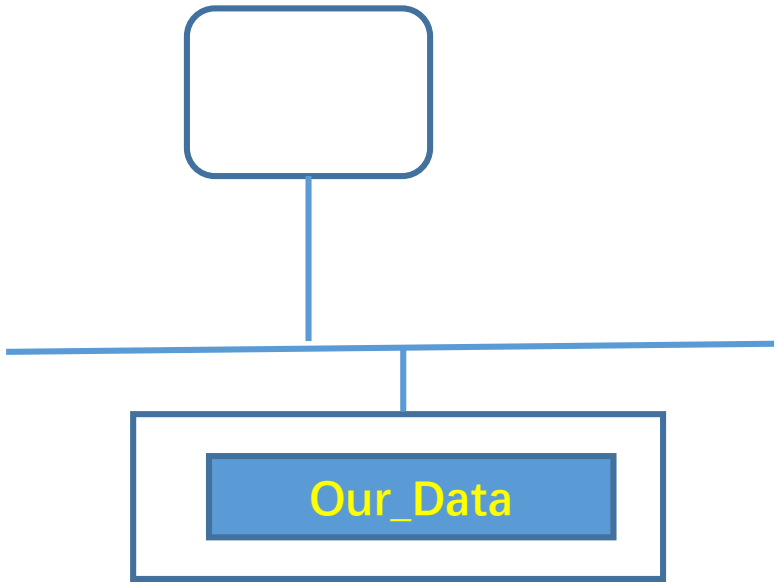
锁的实现（错的）：

数据结构：通常是 flags 里的一个 bit 。

flags是标识资源使用状态的 标识 集合。

```
while（锁bit不是0）
    sleep（&flags，~）；
锁bit = 1;

访问、修改Our_Data；   // 可以睡，其间不可能会有其它进程访问这个数据结构

锁bit = 0;
wakeup（&flags）；
```

UP系统



临界区入睡，~?

访问数据结构前上锁。访问结束开锁。

如果因为等待锁开而入睡，唤醒之后的第一件事：看锁开嘛？不开的话，就再睡。

锁的实现：

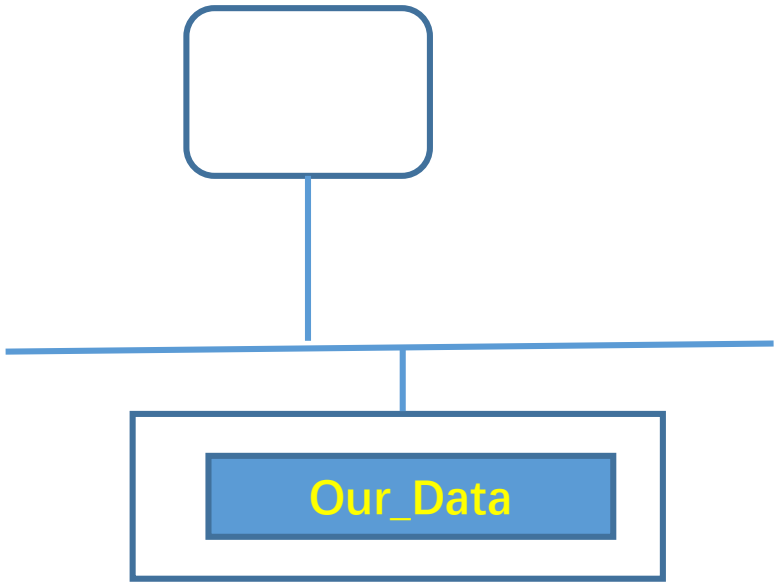
数据结构：通常是 flags 里的一个 bit 。

flags是标识资源使用状态的 标识 集合。

```
CLI;
while（锁bit不是0）
{     STI;   sleep（&flags，~）；   CLI;     }
锁bit = 1;    STI;

访问、修改Our_Data ;    // 可以睡，其间不可能会有其它进程访问这个数据结构

CLI;   锁bit = 0;     STI;
wakeup（&flags）；
```

UP系统

如果可以保证中断处理程序不会访问our data和flags，那就可以锁得松一点



临界区不入睡，怎样保护?

CLI

临界区入睡，~?

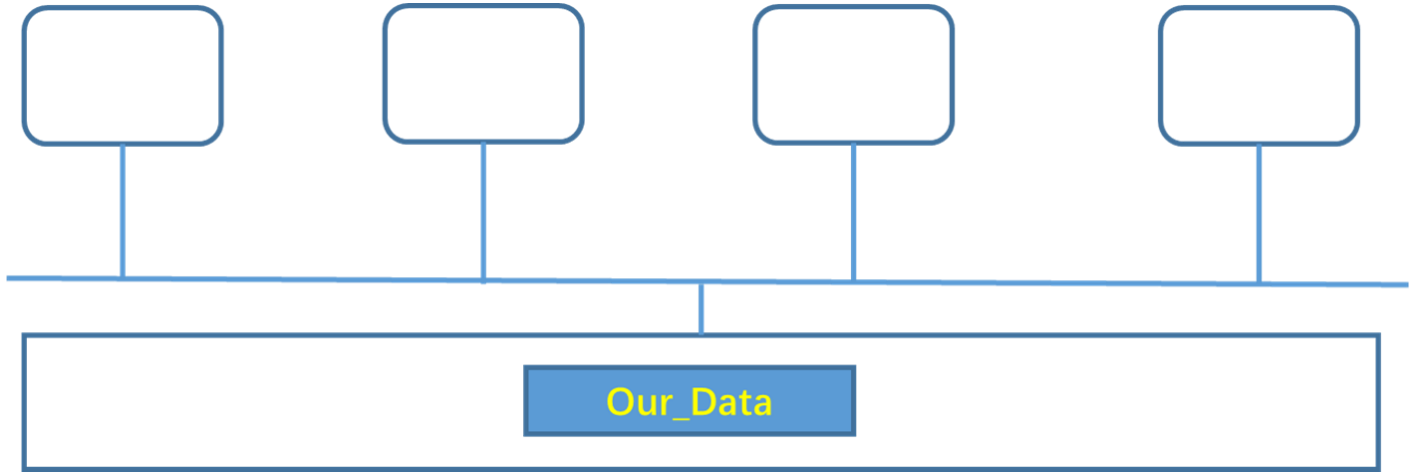
锁的实现：

数据结构：通常是 flags 里的一个 bit 。

flags是标识资源使用状态的 标识 集合。

```
禁内核抢占；                    //preempt_count的抢占计数器++
while（锁bit不是0）    {
    enable内核抢占；            //preempt_count的抢占计数器--
    sleep（&flags，~）；
    禁内核抢占；
}
锁bit = 1;
enable内核抢占；
访问、修改Our_Data ；
锁bit = 0;
wakeup（&flags）；
```

SMP系统



SMP系统中，不同的处理器，竞争内存总线的访问周期。也就是说，正在访问RAM的任何CPU都不能保证能够使用下一个总线周期。

会产生的问题：

1、访问double变量（或没有内存对齐的int）　硬件知识：一个主存周期，CPU get一个int（4字节）

比如，会出现这种情况。假设 Our_Data 是 double 类型的变量。

CPU1 写这个变量需要两个主存周期。 第一个主存周期，写4个bit。 后一个主存周期，写另外4个bit。

假设，在第一个主存周期之后，CPU3获得了主存周期的使用权，读 Our_Data 变量的值。　它读到的 Our_Data 的值如何?

正确的操作是 CPU3 读到 CPU1 写入的完整的数值。如何做到这一点?　锁总线！ 也就是 指令前面带 lock 前缀。

CPU检测到 lock 前缀，会锁住内存总线，直至指令执行完毕。

注意：锁总线期间，所有其它的 CPU 均无法访问内存。系统性能会下降。

2、read-modify-write指令　2个CPU，同时执行 inc value 指令，value的值会错，少 1。

注：

- 局部变量 a 是线程安全的。因为，不存在多个CPU同时访问的可能性

原子操作

- atomic_t类型 整数（内存中，整数边界对齐的整数变量）
- 原子整数操作（指令级的原子操作）

定义一个atomic_c类型的数据方法很平常，你还可以在定义时给它设定初值：

```
atomic_t v;                     /*定义 v */
atomic_t u = ATOMIC_INIT(0);    /*定义 u 并把它初始化为0*/
```

操作也都非常简单：

```
atomic_set(&v,4);   /* v = 4 (原子地)*/
atomic_add(2,&v);   /* v = v + 2 = 6 (原子地) */
atomic_inc(&v);     /* v = v + 1 =7(原子地)*/
```

原子性：赋值操作和算数运算，要么全部执行完，要么根本不执行

可以保证共享整数变量正确性的最轻便的同步操作！　可以使用的时候，不要动锁

原子操作

- 另一组，针对位进行操作的指令，也是原子的

Per-CPU 变量

- 是一个数组。数组中元素的数量与CPU的数量相等。
- 每个CPU以ID标识；使用下标为ID的数组元素。
- 没有不同的CPU访问同一个数组元素的可能性。
- 使用时的注意事项
- 访问Per-CPU 变量期间，禁内核剥夺

锁

如果每个临界区都能像增加变量这样简单就好了，可惜现实总是残酷的。现实世界里，临界区甚至可以跨越多个函数。举个例子，我们经常会碰到这种情况：先得从一个数据结构中移出数据，对其进行格式转换和解析，最后再把它加入到另一个数据结构中。整个执行过程必须是原子的，在数据被更新完毕前，不能有其他代码读取这些数据。显然，简单的原子操作对此无能为力，这就需要使用更为复杂的同步方法——锁来提供保护。

锁（保护）的是共享数据结构。
进程在临界区里操作共享数据结构：

开锁

//持有锁的进程，是操作~的唯一进程
临界区

解锁

被保护的数据结构

member1
member2
member3
member4
member5
……

把锁加在被它保护的数据结构里

锁　锁变量
member1
member2
member3
member4
member5
……

二值锁

- 二值锁：只有开关，这两种状态。持有锁的只可能是一个线程。
  二值锁用来实现被保护数据结构的互斥访问。
- 非二值的锁：　　可能会有多个线程同时持有锁。

自旋锁和信号量

- 自旋锁，适用于SMP。是一种简单轻便的锁。持锁的是1个进程。在此期间，其它CPU上的进程试图获得自旋锁不成功时，会忙等开锁。

- 信号量，既适用于单处理器系统，又适用于SMP。

- 自旋锁，适用于加锁时间很短的场合。
  - 自旋锁使用时的注意事项：使用前禁内核抢占，临界区不能入睡。

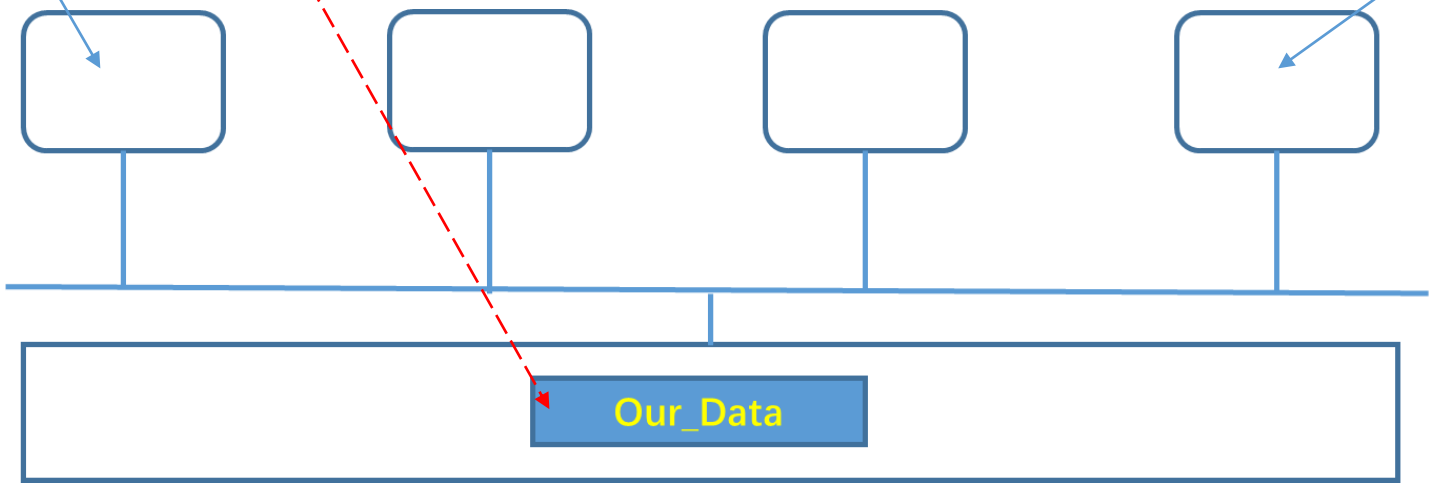
- 信号量，适用于加锁时间很长。特别是，加锁其间进程有可能入睡的场合。

自旋锁　　基本实现

初始化的状态：　　spin_lock_t mr_lock = SPIN_LOCK_UNLOCKED;

```
spin_lock (&mr_lock);    //上锁，本CPU禁中断

临界区     //  4个CPU 只有我可以访问 Our_Data

spin_unlock (&mr_lock);    //开锁，本CPU开中断
```

```
spin_lock (&mr_lock);

临界区

spin_unlock (&mr_lock);
```



（中断处理程序不会访问到our_data，才可以这样用。否则，系统会被锁死）

禁中断，同时使用自旋锁

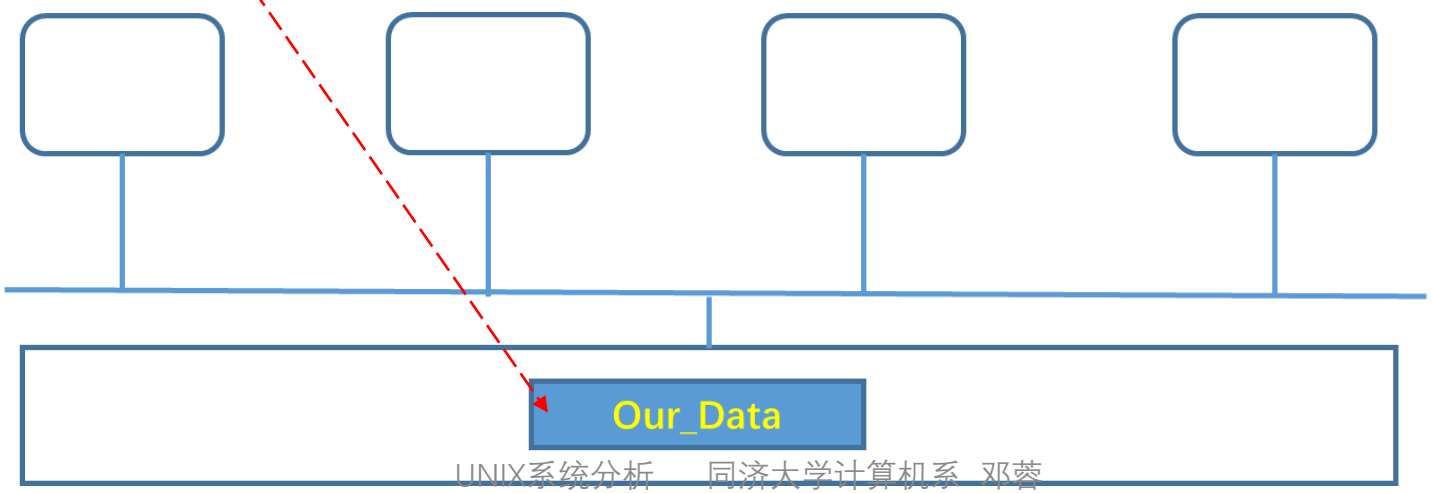
防止中断上半段破坏变量的值

初始化的状态：　　spin_lock_t mr_lock = SPIN_LOCK_UNLOCKED;

```
spin_lock_irqsave(&mr_lock, flags);

临界区

spin_unlock_irqrestore(&mr_lock, flags);
```

//把当前中断状态保存在flags里，关中断，
再去获得自旋锁

//用 flags 恢复 中断的先前状态，
释放自旋锁



只要 Our_Data 有可能被中断上半段访问，就应该用这段代码互斥。

关中，防止本CPU执行的中断处理程序访问这个共享变量。

自旋锁，防其它CPU~

禁软中断，同时使用自旋锁

防止软中断破坏变量的值

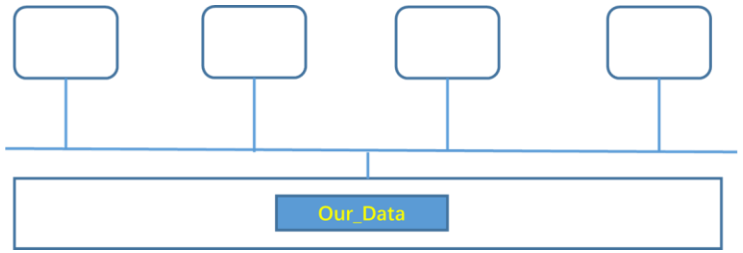
初始化的状态：　　spin_lock_t mr_lock = SPIN_LOCK_UNLOCKED;

```
spin_lock_bh(&mr_lock);

临界区

spin_unlock_bh(&mr_lock);
```

```
local_bh_disable( );
spin_lock (&mr_lock);
```

```
spin_unlock (&mr_lock);
local_bh_enable( );
```

```
void local_bh_disable(void)
{
    struct thread_info *t = current_thread_info();
    t->preempt_count += SOFTIRQ_OFFSET;
}
```




```
void local_bh_enable(void)
{
    struct thread_info *t = current_thread_info();
    t->preempt_count -= SOFTIRQ_OFFSET;
    if (unlikely(!t->preempt_count && softirq_pending(smp_processor_id())))
    do_softirq();
}
```

软中断代码保护共享变量

- 和其它软中断共享的变量
  - 用自旋锁，不用禁下半部（因为，单CPU上软中断是串行执行的）

```
spin_lock (&mr_lock);
```

- 和上半部共享的变量
  - 关中断　防上半部
  - 自旋锁　防其它CPU

```
spin_lock_irqsave(&mr_lock, flags);
```

信号量 睡眠锁

门和钥匙的例子。当某个人到了门前，他抓取钥匙，然后进入房间。最大的差异在于当另一个人到了门前，但无法得到钥匙时会发生什么情况。在这种情况下，这家伙不是在徘徊等待，而是把自己的名字写在一个列表中，然后打盹去了。当里面的人离开房间时，就在门口查看一下列表。如果列表上有名字，他就对第一个名字仔细检查，并在胸部给他一拳，叫醒他，让他进入房间。在这种方式中，钥匙（相当于信号量）继续确保一次只有一个人（相当于执行线程）进入房间（相当于临界区）。如果房间被占用，那么，那个人不是徘徊等待，而是把自己的名字写在列表（相当于等待队列）上，然后打一会儿盹（相当于在等待队列上阻塞，并且入睡），让处理器离开这里到别的地方执行代码。这就比自旋锁提供了更好的处理器利用率，因为没有把时间花费在忙等上，但是，信号量比自旋锁有更大的开销。生活总是一分为二的。

我们可以从信号量的睡眠特性得出一些有意思的结论：

- 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况。
- 相反，锁被短时间持有时，使用信号量就不太适宜了。因为睡眠、维护等待队列以及唤醒所花费的开销可能比锁被占用的全部时间还要长。
- 由于执行线程在锁被争用时会睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文中是不能进行调度的。
- 你可以在持有信号量时去睡眠（当然你也可能并不需要睡眠），因为当其他进程试图获得同一信号量时不会因此而死锁（因为该进程也只是去睡眠而已，而你最终会继续执行的）。
- 在你占用信号量的同时不能占用自旋锁。因为在你等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

自旋锁保护的临界区，不能用P操作

- **中断处理程序不能用信号量保护共享变量，只能用自旋锁。 这是因为，不允许中断处理程序入睡**

完成变量（类似于条件变量）

- 内核同步工具。保证一个操作完成之后，另一个操作才能开始。

完成变量由结构completion表示，定义在<linux/completion.h>中。通过以下宏静态地创建完成变量并初始化它：

```
DECLARE_COMPLETION(mr_comp);
```

通过 init_completion()动态创建并初始化完成变量。

在一个指定的完成变量上，需要等待的任务调用wait_for_completion()来等待特定事件。当特定事件发生后，产生事件的任务调用complete()来发送信号唤醒正在等待的任务。表9-7列出了完成变量的方法。

表9-7 完成变量方法

方 法	描 述
init_completion(struct completion *)	初始化指定的动态创建的完成变量
wait_for_completion(struct completion *)	等待指定的完成变量接受信号
complete(struct completion *)	发信号唤醒任何等待任务

使用完成变量的例子可以参考kernel/sched.c和kernel/fork.c。完成变量的通常用法是，将完成变量作为数据结构中的一项动态创建，而完成数据结构初始化工作的内核代码将调用wait_for_completion()进行等待。初始化完成后，初始化函数调用completion()唤醒在等待的内核任务。

内核动态生成数据结构，并且完成这些数据结构的初始化工作。

在初始化没有结束之前，其它模块不可以使用新近创建的数据结构。

禁止抢占

内核何时可以被抢占？

Bits	Description
07	Preemption counter (max value = 255)
815	Softirq counter (max value = 255).
1627	Hardirq counter (max value = 4096)
28	PREEMPT_ACTIVE flag

- 开中断
- &现运行进程 thread_info 中的 preempt_count 等于0。

内核不可以被抢占

1. 内核正在执行中断服务例程。
2. 内核正在执行软中断或 tasklet
3. 通过把抢占计数器设置为正数而显式地禁用内核抢占
4. 关中断

操作抢占计数器，禁止内核抢占的情形

1、内核在使用自旋锁的时候，会禁止内核抢占。

2、内核访问 per CPU variable 的时候，不需要用自旋锁锁住其它处理器上的内核代码。但需要保证，访问这些变量的过程中不可以放弃CPU。也就是，不能够入睡，禁止抢占。

内核抢占的第1个时刻：系统调用离开临界区

```
preempt_enable( )
{
        ……preempt_count --;
        if ( TIF_NEED_RESCHED == 0 )
                return;
        if (!current_thread_info->preempt_count && !irqs_disabled()) {
            current_thread_info->preempt_count = PREEMPT_ACTIVE;
            schedule();
            current_thread_info->preempt_count = 0;
        }
}
```

Bits	Description
07	Preemption counter (max value = 255)
815	Softirq counter (max value = 255).
1627	Hardirq counter (max value = 4096)
28	PREEMPT_ACTIVE flag

内核抢占的第2个时刻：中断控制路径结束

中断内核控制路径：
入口时，硬中断计数器++

- 中断处理程序结束时（如果没有其它中断~）
- 软中断结束时

Bits	Description
07	Preemption counter (max value = 255)
815	Softirq counter (max value = 255).
1627	Hardirq counter (max value = 4096)
28	PREEMPT_ACTIVE flag

硬中断计数器--（或，软中断计数器--）；
……

```
if ( TIF_NEED_RESCHED == 0 )
        return;
if (!current_thread_info->preempt_count && !irqs_disabled()) {
    current_thread_info->preempt_count = PREEMPT_ACTIVE;
    schedule();
    current_thread_info->preempt_count = 0;
}
```

自旋锁的实现（看重点，先不管内核抢占）

spin_lock( )

```
1: lock; decb slp->slock
   jns  3f  //如果符号位不为1就跳转
2: pause
   cmpb $0,slp->slock
   jle 2b   //小于等于0就跳转
   jmp 1b
3:
```

spin_unlock( )

```
slp->slock = 1;
```

对pause指令，如何可以提高系统性能的解释
https://blog.csdn.net/maray/article/details/8757030

抢占式内核中自旋锁的用法

- 禁内核抢占
- spin_lock( )
- 临界区
- spin_unlock( )
- enable内核抢占

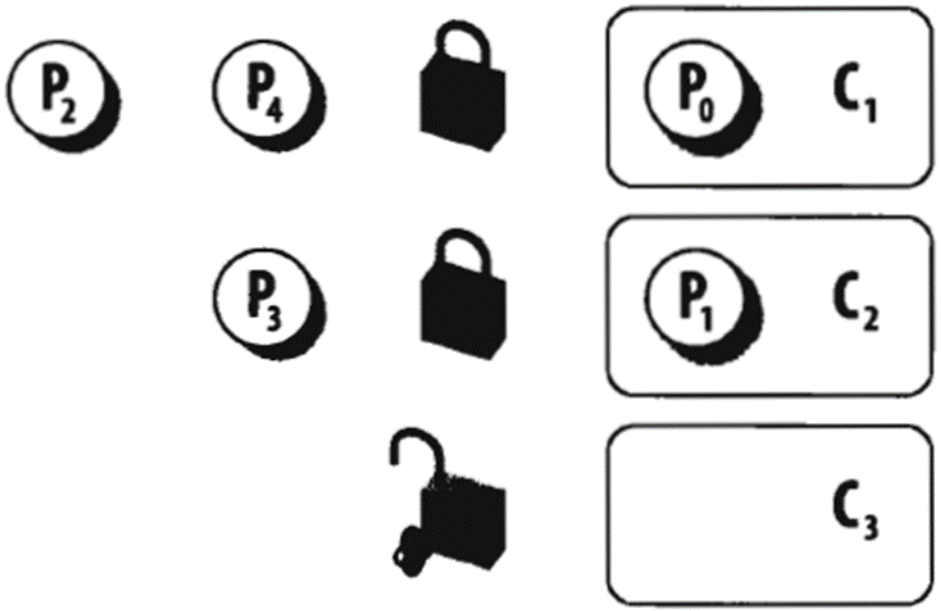
- 持有自旋锁的进程，不能放弃CPU
- 如果一个持有自旋锁的进程放弃了CPU。设下次该进程T秒之后恢复运行。那最坏的情况是： 废了3*T秒的计算能力（4核系统）

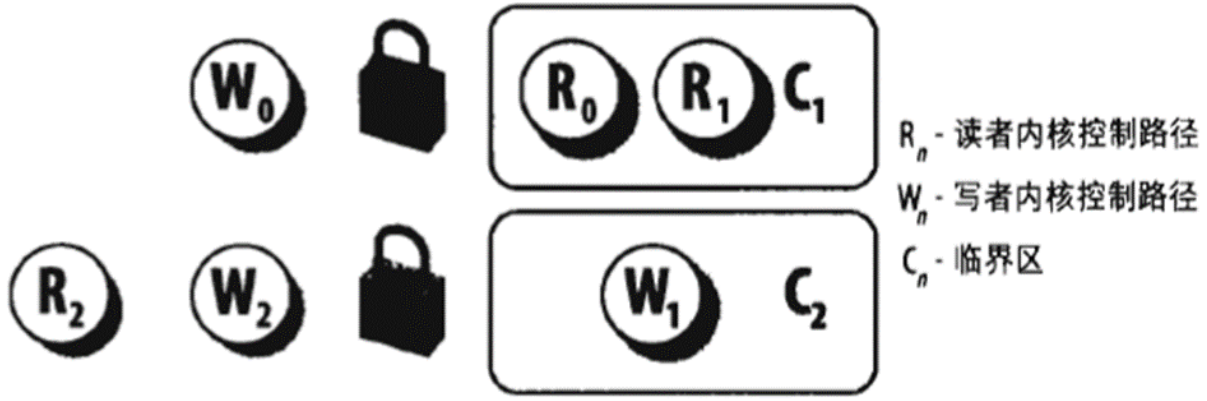
9.3 读-写自旋锁

有时，锁的用途可以明确地分为读取和写入。例如，对一个链表可能既要更新又要检索。当更新（写入）链表时，不能有其他代码并发地写链表或从链表中读取数据，写操作要求完全互斥。另一方面，当对其检索（读取）链表时，只要其他程序不对链表进行写操作就行了。只要没有写操作，多个并发的读操作都是安全的。任务链表的存取模式（在第3章中讨论过）就非常类似于这种情况，它就是通过读——写自旋锁获得保护的。

当对某个数据结构的操作可以像这样被划分为读/写两种类别时，类似读/写锁这样的机制就很有用了。为此，Linux提供了专门的读——写自旋锁。这种自旋锁为读和写分别提供了不同的锁。一个或多个读任务可以并发的持有读者锁；相反，用于写的锁最多只能被一个写任务持有，而且此时不能有并发的读操作。有时把读/写锁叫做共享/排斥锁，或者并发/排斥锁，因为这种锁以共享（对读者而言）和排斥（对写者而言）的形式获得使用。

普通自旋锁 和 读-写锁





读锁，共享锁（并发锁）
写锁，互斥锁

读/写自旋锁的使用方法类似于普通自旋锁，它们通过下面的方法初始化：

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;
```

然后，在读者的代码分支中使用如下函数：

```
read_lock(&mr_rwlock);
/*临界区(只读)*/
read_unlock(&mr_rwlock);
```

最后，在写者的代码分支中使用如下函数：

```
write_lock(&mr_rwlock);
/* 临界区(读写)…*/
write_unlock(&mr_lock);
```

通常情况下，读锁和写锁会位于完全分割开的代码分支中，如上例所示。
注意，不能把一个读锁“升级”为写锁。这种代码：

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

将会带来死锁，因为写锁会不断自旋，等待所有的读者释放锁，其中也包括它自己。所以当确实需要写操作时，要在一开始就请求写锁。如果写和读不能清晰地分开的话，那么使用一般的自旋锁就行了，不要使用读-写自旋锁。

seqlock （顺序锁，又称序列锁）

- 数据结构 seqlock_t，with
  - spinlock_t lock
  - sequence； 顺序锁保护的数据结构，内容的版本号

- 写，上锁封锁下一个写者，不封锁读者

- write_seqlock( ) //上锁后，sequence++
- 修改数据结构的临界区
- write_sequnlock( ) // sequence++，解锁

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* ...临界区... */
} while (read_seqretry(&seqlock, seq));
```

- 读数据结构不上锁

read_seqbegin()返回顺序锁的当前顺序号；如果局部变量seq的值是奇数（写者在read_seqbegin()函数被调用后，正更新数据结构），或seq的值与顺序锁的顺序计数器的当前值不匹配（当读者正执行临界区代码时，写者开始工作），read_seqretry()就返回1。

RCU锁（读-拷贝-更新）

- **不锁，不用计数器！！** 也能够保证数据结构不被破坏
- 随时读，随时写。 不管多少个读者写者并发，都没事
- 假设，我们有一个内核数据结构kernel_foo。指向这个数据结构的内核指针是foo_finger

读kernel_foo(读者)

- rcu_read_lock ( )； 禁内核抢占
- 临界区读
- rcu_read_unlock ( )

指针赋值是原子操作。所有进程操作的都是完整的数据结构新旧版本不同而已！

写者

- local_finger = new （struct kernel_foo）
- 用左边的代码把foo_finger指向的数据结构的值读到local_finger指向的新的空间。
- 随 便 怎 样 修 改
- 满意了， 就赋值

  foo_finger = local_finger