[**Linux内核原子 - spinlock的实现**](http://www.cnblogs.com/Rofael/p/3204482.html)

*spinlock*的数据结构*spinlock\_t*定义在头文件linux/spinlock\_types.h里面：

[复制代码](javascript:void(0);)

20 typedef struct {

21 **raw\_spinlock\_t raw\_lock**;

22 #ifdef CONFIG\_GENERIC\_LOCKBREAK

23 unsigned int break\_lock;

24 #endif

25 #ifdef CONFIG\_DEBUG\_SPINLOCK

26 unsigned int magic, owner\_cpu;

27 void \*owner;

28 #endif

29 #ifdef CONFIG\_DEBUG\_LOCK\_ALLOC

30 struct lockdep\_map dep\_map;

31 #endif

32 } spinlock\_t;

[复制代码](javascript:void(0);)

其中抛开*debug*的数据成员，最核心的成员就是*raw\_lock*，这是一个和处理器架构相关的结构。

比如X86中定义是(arch/x86/include/asm/spinlock\_types.h)：

8 typedef struct raw\_spinlock {

9 unsigned int slock;

10} raw\_spinlock\_t;

ARM中的定义是(arch/arm/include/asm/spinlock\_types.h)：

8 typedef struct {

9 volatile unsigned int lock;

10 } raw\_spinlock\_t;

在单核处理器中，raw\_spinlock\_t被定义为空结构体（linux/spinlock\_types\_up.h)：

25 typedef struct { } raw\_spinlock\_t;

不管是什么体系结构，kernel都是根据结构体里的slock或者lock的值来判断当前的锁是被占用还是空闲，并且做出相应的动作（单核处理器除外）。

在linux/spinlock.h中定义了spinlock操作的API。

spinlock的思想就是在SMP环境中，保护共享的数据结构；也就是CPU-A正在访问（读写）共享数据的期间，其他CPU不能访问同样的共享数据，这样就保证了SMP-safe。每个线程在访问共享数据的之前，都需要获取spin lock,如果锁正被其他线程所占有，那么获取锁的线程则“空转”CPU以等待其他线程释放锁；spin lock相对于信号量这样的锁机制的好处就是，节约了2次context switch的开销，所以如果线程等待锁的时间小于2次context switch的时间，系统性能从spin lock获得的提升就越多。

spin lock除了考虑SMP-safe以外，还要考虑两种伪并发情况，就是中断(interrupt)和抢占(preemption)，就是要保证interrupt-safe和preempt-safe。

如果在中断处理程序中，因为要访问共享变量而使用spin lock，则要避免dead-lock出现。比如，CPU0上线程A获取了锁1，在获取和释放锁之间CPU0上发生软中断进入中断处理程序，中断处理程序也尝试去获取spin lock，但是由于同一CPU0上的lock holder线程A在中断处理程序退出之前无法被调度而释放锁，所以在CPU0上就出现dead-lock；但是如果软中断是发生在其他CPU比如CPU1上，则是没有问题的，因为发现在CPU1上的中断不会中断CPU0上lock holder线程A的执行。所以要保证interrupt-safe，就要在获取锁之前disable**本地**CPU中断。

kernel文档spinlocks.txt里面有相关的描述：

[112](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "112) The reasons you mustn't use these versions if you have interrupts that

[113](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "113) play with the spinlock is that you can get deadlocks:

[114](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "114)

[115](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "115) spin\_lock(&lock);

[116](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "116) ...

[117](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "117) <- interrupt comes in:

[118](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "118) spin\_lock(&lock);

[119](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "119)

[120](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "120) where an interrupt tries to lock an already locked variable. This is ok if

[121](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "121) the other interrupt happens on another CPU, but it is \_not\_ ok if the

[122](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "122) interrupt happens on the same CPU that already holds the lock, because the

[123](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "123) lock will obviously never be released (because the interrupt is waiting

[124](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "124) for the lock, and the lock-holder is interrupted by the interrupt and will

[125](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "125) not continue until the interrupt has been processed).

[126](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "126)

[127](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "127) (This is also the reason why the irq-versions of the spinlocks only need

[128](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "128) to disable the \_local\_ interrupts - it's ok to use spinlocks in interrupts

[129](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "129) on other CPU's, because an interrupt on another CPU doesn't interrupt the

[130](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "130) CPU that holds the lock, so the lock-holder can continue and eventually

[131](http://www.mjmwired.net/kernel/Documentation/spinlocks.txt" \l "131) releases the lock).

然后就是preempt-safe。

**spin\_lock\_init**

spin\_lock\_init的实现是一个宏，对spinlock\_t类型的lock做一个初始化。

80 # define \_\_SPIN\_LOCK\_UNLOCKED(lockname) \

81 (spinlock\_t) { .raw\_lock = \_\_RAW\_SPIN\_LOCK\_UNLOCKED, \

82 SPIN\_DEP\_MAP\_INIT(lockname) }

[94](http://lxr.linux.no/linux+v2.6.32.61/include/linux/spinlock_types.h" \l "L94)#define [SPIN\_LOCK\_UNLOCKED](http://lxr.linux.no/linux+v2.6.32.61/+code=SPIN_LOCK_UNLOCKED) [\_\_SPIN\_LOCK\_UNLOCKED](http://lxr.linux.no/linux+v2.6.32.61/+code=__SPIN_LOCK_UNLOCKED)([old\_style\_spin\_init](http://lxr.linux.no/linux+v2.6.32.61/+code=old_style_spin_init))

[104](http://lxr.linux.no/linux+v2.6.32.61/include/linux/spinlock.h" \l "L104)# define [spin\_lock\_init](http://lxr.linux.no/linux+v2.6.32.61/+code=spin_lock_init)([lock](http://lxr.linux.no/linux+v2.6.32.61/+code=lock)) \

[105](http://lxr.linux.no/linux+v2.6.32.61/include/linux/spinlock.h" \l "L105) do { \*([lock](http://lxr.linux.no/linux+v2.6.32.61/+code=lock)) = [SPIN\_LOCK\_UNLOCKED](http://lxr.linux.no/linux+v2.6.32.61/+code=SPIN_LOCK_UNLOCKED); } while (0)

其中raw\_lock被初始化为宏\_\_RAW\_SPIN\_LOCK\_UNLOCKED，很明显，这个宏也会是体系结构相关的，在X86它被定义为：

12 #define \_\_RAW\_SPIN\_LOCK\_UNLOCKED { 0 }

也就是说，在X86中如果无符号整型变量slock的值是0，为则UNLOCKED的状态。

**spin\_lock**

如果能确定被保护的共享变量在*interrupt*中是不会被访问的，那么可以忽略*interrupt-safe*,用简单也更有效率的*spin\_lock*。

在*UP*的环境中，*spin\_lock*的实现是没有*lock*操作的，*spin\_lock*仅仅保证在线程在临界区中（也就是*spin\_lock*和*spin\_unlock*之前的section）是不会被抢占的preempt的。

*UP*版*spin\_lock*的实现是在linux/spinlock\_api\_up.h中：

[复制代码](javascript:void(0);)

51#define \_spin\_lock(lock) \_\_LOCK(lock)

21/\*

22 \* In the UP-nondebug case there's no real locking going on, so the

23 \* only thing we have to do is to keep the preempt counts and irq

24 \* flags straight, to suppress compiler warnings of unused lock

25 \* variables, and to add the proper checker annotations:

26 \*/

27#define \_\_LOCK(lock) \

28 do { preempt\_disable(); \_\_acquire(lock); (void)(lock); } while (0)

[复制代码](javascript:void(0);)

preempt\_disable()禁止在临界区中线程被抢占。

(void)(lock)是避免编译器的报警。

*smp*版的spin\_lock的实现在spinlock\_api\_smp.h中：

[复制代码](javascript:void(0);)

333 static inline void \_\_spin\_lock(spinlock\_t \*lock)

334 {

335 preempt\_disable();

336 spin\_acquire(&lock->dep\_map, 0, 0, \_RET\_IP\_);

337 LOCK\_CONTENDED(lock, \_raw\_spin\_trylock, \_raw\_spin\_lock);

338 }

393 #define LOCK\_CONTENDED(\_lock, try, lock) \

394 lock(\_lock)

[复制代码](javascript:void(0);)

也就是说，核心的操作就是\_raw\_spin\_lock(\_lock)。

160# define \_raw\_spin\_lock(lock) \_\_raw\_spin\_lock(&(lock)->raw\_lock)

\_\_raw\_spin\_lock的是一个和arch相关的实现了，在内核2.5.24上的X86平台上（asm-x86/spinlock\_64.h):

[复制代码](javascript:void(0);)

25static inline void \_\_raw\_spin\_lock(raw\_spinlock\_t \*lock)

26{

27 asm volatile(

28 "\n1:\t"

29 LOCK\_PREFIX " ; decl %0\n\t"

30 "jns 2f\n"

31 "3:\n"

32 "rep;nop\n\t"

33 "cmpl $0,%0\n\t"

34 "jle 3b\n\t"

35 "jmp 1b\n"

36 "2:\t" : "=m" (lock->slock) : : "memory");

37}

[复制代码](javascript:void(0);)

实际上，spinlock的实现就是检查lock->slock的值来判断锁的free or busy状态，所以不同的CPU对锁进行的decl或者incl指令必须是原子的，否则会出现多个CPU同时认为锁是free而进入临界区或者所有CPU都认为锁是busy而dead-lock的后果；在x86平台上，LOCK\_PREFIX用前缀保证对lock->slock的原子性。

LOCK\_PREFIX的实现可以参见 <http://wenku.baidu.com/view/13dbbe1fb7360b4c2e3f642b.html>。

'rep;nop'是什么指令呢？我们反汇编看看：

[复制代码](javascript:void(0);)

#include <stdio.h>

static inline void rep\_nop(void)

{

asm volatile("rep; nop" ::: "memory");

}

int main(void)

{

rep\_nop();

return 0;

}

[yzhang2@sles10sp3 ~]$gcc -c asm.c

[yzhang2@sles10sp3 ~]$objdump -s -d asm.o

0000000000000010 <rep\_nop>:

10: 55 push %rbp

11: 48 89 e5 mov %rsp,%rbp

14: f3 90 **pause**

16: c9 leaveq

17: c3 retq

[复制代码](javascript:void(0);)

原来'rep;nop'指令被解释为了'*pause*'指令。

这段代码的具体逻辑就是：

decl %0就是将lock->slock减1，如果锁是空闲的，计算结果是0，根据‘jns 2f'则跳转到2，退出函数获取锁；如果锁是被占有的，结果是负数，则运行'rep;nop'，然后再次比较lock->slock，如果其大于0，说明锁已经被释放，跳到1重新尝试获得锁；否则继续等待。

但是这样的自旋锁不能保证获取锁的fairness,所以在2.6.25以后引入了FIFO ticket spinlock

[复制代码](javascript:void(0);)

109 static \_\_always\_inline void \_\_ticket\_spin\_lock(raw\_spinlock\_t \*lock)

110 {

111 int inc = 0x00010000;

112 int tmp;

113

114 asm volatile(LOCK\_PREFIX "xaddl %0, %1\n"

115 "movzwl %w0, %2\n\t"

116 "shrl $16, %0\n\t"

117 "1:\t"

118 "cmpl %0, %2\n\t"

119 "je 2f\n\t"

120 "rep ; nop\n\t"

121 "movzwl %1, %2\n\t"

122 /\* don't need lfence here, because loads are in-order \*/

123 "jmp 1b\n"

124 "2:"

125 : "+r" (inc), "+m" (lock->slock), "=&r" (tmp)

126 :

127 : "memory", "cc");

128 }

[复制代码](javascript:void(0);)

在执行完xaddl以后，%0的值是(lock->slock),%1的值是（inc+lock->slock),也就是把slock值的next域加1。

movzwl是将(lock->slock)的owner字段赋给%2，shrl是将(lock->slock)的next字段赋给%0

cmpl就是比较slock的next和owner字段，如果相等，则代表获得了锁；不相等，则进入忙等待(rep;nop)，然后通过movzwl %1 %2更新owner字段，因为spin\_unlock的操作就是把owner字段加1，然后返回到1重新比较，如果这时owner==next，则获得锁。

这样，各个cpu指定了自己的next字段，然后他们就能按照顺序保证了cpu获取锁的fairness。

**传统自旋锁的实现与不足**

Linux 内核自旋锁的底层数据结构 raw\_spinlock\_t 定义如下：

**清单 1. raw\_spinlock\_t 数据结构**

typedef struct {

unsigned int slock;

} raw\_spinlock\_t;

slock 虽然被定义为无符号整数，但是实际上被当作有符号整数使用。slock 值为 1 代表锁未被占用，值为 0 或负数代表锁被占用。初始化时 slock 被置为 1。

线程通过宏 spin\_lock 申请自旋锁。如果不考虑内核抢占，则 spin\_lock 调用 \_\_raw\_spin\_lock 函数，代码如下所示：

**清单 2. \_\_raw\_spin\_lock 函数**

static inline void \_\_raw\_spin\_lock(raw\_spinlock\_t \*lock)

{

asm volatile("\n1:\t"

LOCK\_PREFIX " ; decb %0\n\t"

"jns 3f\n"

"2:\t"

"rep;nop\n\t"

"cmpb $0,%0\n\t"

"jle 2b\n\t"

"jmp 1b\n"

"3:\n\t"

: "+m" (lock->slock) : : "memory");

}

1. LOCK\_PREFIX 的定义如下：

**清单 3. LOCK\_PREFIX宏**

#ifdef CONFIG\_SMP

#define LOCK\_PREFIX \

".section .smp\_locks,\"a\"\n" \

\_ASM\_ALIGN "\n" \

\_ASM\_PTR "661f\n" /\* address \*/ \

".previous\n" \

"661:\n\tlock; "

#else /\* ! CONFIG\_SMP \*/

#define LOCK\_PREFIX ""

#endif

在多处理器环境中 LOCK\_PREFIX 实际被定义为 “lock”前缀。

x86 处理器使用“lock”前缀的方式提供了在指令执行期间对总线加锁的手段。芯片上有一条引线 LOCK，如果在一条汇编指令(ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG)前加上“lock” 前缀，经过汇编后的机器代码就使得处理器执行该指令时把引线 LOCK 的电位拉低，从而把总线锁住，这样其它处理器或使用DMA的外设暂时无法通过同一总线访问内存。

从 P6 处理器开始，如果指令访问的内存区域已经存在于处理器的内部缓存中，则“lock” 前缀并不将引线 LOCK 的电位拉低，而是锁住本处理器的内部缓存，然后依靠缓存一致性协议保证操作的原子性。

1. decb 汇编指令将 slock 的值减 1。由于“减 1”是“读-改-写”操作，不是原子操作，可能会被同时申请锁的其它处理器上的线程干扰，所以必须加上“lock”前缀。
2. jns 汇编指令检查 EFLAGS 寄存器的 SF(符号)位，如果为 0，说明 slock 原来的值为 1，则线程获得锁，然后跳到标签 3 的位置结束本次函数调用。如果 SF 位为 1，说明 slock 原来的值为 0 或负数，锁已被占用。那么线程转到标签 2 处不断测试 slock 与 0 的大小关系，假如 slock 小于或等于 0，跳转到标签 2 的位置继续忙等待；假如 slock 大于 0，说明锁已被释放，则跳转到标签 1 的位置重新申请锁。

线程通过宏 spin\_unlock 释放自旋锁，该宏调用 \_\_raw\_spin\_unlock 函数：

**清单 4. \_\_raw\_spin\_unlock函数**

static inline void \_\_raw\_spin\_unlock(raw\_spinlock\_t \*lock)

{

asm volatile("movb $1,%0" : "+m" (lock->slock) :: "memory");

}

可见 \_\_raw\_spin\_unlock 函数仅仅执行一条汇编指令：将 slock 置为 1。

尽管拥有使用简单方便、性能好的优点，自旋锁也存在自身的不足：

1. 由于传统自旋锁无序竞争的本质特点，内核执行线程无法保证何时可以取到锁，某些执行线程可能需要等待很长时间，导致“不公平”问题的产生。这有两方面的原因：
   1. 随着处理器个数的不断增加，自旋锁的竞争也在加剧，自然导致更长的等待时间。
   2. 释放自旋锁时的重置操作将无效化所有其它正在忙等待的处理器的缓存，那么在处理器拓扑结构中临近自旋锁拥有者的处理器可能会更快地刷新缓存，因而增大获得自旋锁的机率。
2. 由于每个申请自旋锁的处理器均在全局变量 slock 上忙等待，系统总线将因为处理器间的缓存同步而导致繁重的流量，从而降低了系统整体的性能。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-spinlock/#ibm-pcon)

**排队自旋锁的设计原理**

传统自旋锁的“不公平”问题在锁竞争激烈的服务器系统中尤为严重，因此 Linux 内核开发者 Nick Piggin 在 Linux 内核 2.6.25 版本中引入了排队自旋锁：通过保存执行线程申请锁的顺序信息来解决“不公平”问题。大家可能有这样的经历：去银行处理业务的时候，银行一般都有一个排队机，供客户抽号；如果客户抽的号正在柜台上被叫，则可以直接去办理业务；如果抽的号比柜台上正办理业务的号大，那就要等待。Nick Piggin的排队自旋锁就是这个道理：将spinlock的整形变量当成两部分来看，分别是next和owner(当CPU<256的时候，仅使用变量的低16位，此16位的高8位为next，低8位为owner；当cpu>256的时候，变量的32位全被使用，高16位为next，低16位为owner)；锁初始状态为0，即next=owner=0；需要获取锁时，spin\_lock()内部会首先返回当前的next值，并同时将其加1，然后判断返回next是否等于owner，相等的话，锁获取成功，否则循环判断(判断返回next是否等于owner)；锁释放时，spin\_unlock()会将owner的值加1，这样第一个要求锁的CPU就有能力在第一时间获取锁，其它竞争者只能排队等候了，That's it！完全公平了，不存在恶意争抢的问题了，但这就完美了吗？

排队自旋锁仍然使用原有的 raw\_spinlock\_t 数据结构，但是赋予 slock 域新的含义。为了保存顺序信息，slock 域被分成两部分，分别保存锁持有者和未来锁申请者的票据序号(Ticket Number)，如下图所示：

**图 1. Next 和 Owner 域**

Ticket Number

如果处理器个数不超过 256，则 Owner 域为 slock 的 0-7 位，Next 域为 slock 的 8-15 位，slock 的高 16 位不使用；如果处理器个数超过 256，则 Owner 和 Next 域均为 16 位，其中 Owner 域为 slock 的低 16 位。可见排队自旋锁最多支持 216=65536 个处理器。

只有 Next 域与 Owner 域相等时，才表明锁处于未使用状态（此时也无人申请该锁）。排队自旋锁初始化时 slock 被置为 0，即 Owner 和 Next 置为 0。内核执行线程申请自旋锁时，原子地将 Next 域加 1，并将原值返回作为自己的票据序号。如果返回的票据序号等于申请时的 Owner 值，说明自旋锁处于未使用状态，则直接获得锁；否则，该线程忙等待检查 Owner 域是否等于自己持有的票据序号，一旦相等，则表明锁轮到自己获取。线程释放锁时，原子地将 Owner 域加 1 即可，下一个线程将会发现这一变化，从忙等待状态中退出。线程将严格地按照申请顺序依次获取排队自旋锁，从而完全解决了“不公平”问题。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-spinlock/#ibm-pcon)

**排队自旋锁的实现**

排队自旋锁没有改变原有自旋锁的调用接口，该 API 是以 C 语言宏的形式提供给开发人员。下表列出 6 个主要的 API 和相对应的底层实现函数：

**表 1. 排队自旋锁 API**

| **宏** | **底层实现函数** | **描述** |
| --- | --- | --- |
| **spin\_lock\_init** | 无 | 将锁置为初始未使用状态(值为 0) |
| **spin\_lock** | \_\_raw\_spin\_lock | 忙等待直到 Owner 域等于本地票据序号 |
| **spin\_unlock** | \_\_raw\_spin\_unlock | Owner 域加 1，将锁传给后续等待线程 |
| **spin\_unlock\_wait** | \_\_raw\_spin\_unlock\_wait | 不申请锁，忙等待直到锁处于未使用状态 |
| **spin\_is\_locked** | \_\_raw\_spin\_is\_locked | 测试锁是否处于使用状态 |
| **spin\_trylock** | \_\_raw\_spin\_trylock | 如果锁处于未使用状态，获得锁；否则直接返回 |

下面介绍其中 3 个底层函数的实现细节，假定处理器个数不超过 256。

1. \_\_raw\_spin\_is\_locked

**清单 5. \_\_raw\_spin\_is\_locked 函数**

static inline int \_\_raw\_spin\_is\_locked(raw\_spinlock\_t \*lock)

{

int tmp = \*(volatile signed int \*)(&(lock)->slock);

return (((tmp >> 8) & 0xff) != (tmp & 0xff));

}

* 1. 此函数判断 Next 和 Owner 域是否相等，如果相等，说明自旋锁处于未使用状态，返回 0；否则返回1。
  2. tmp 这种复杂的赋值操作是为了直接从内存中取值，避免处理器缓存的影响。

1. \_\_raw\_spin\_lock

**清单 6. \_\_raw\_spin\_lock 函数**

static inline void \_\_raw\_spin\_lock(raw\_spinlock\_t \*lock)

{

short inc = 0x0100;

\_\_asm\_\_ \_\_volatile\_\_ (

LOCK\_PREFIX "xaddw %w0, %1\n"

"1:\t"

"cmpb %h0, %b0\n\t"

"je 2f\n\t"

"rep ; nop\n\t"

"movb %1, %b0\n\t"

/\* don't need lfence here, because loads are in-order \*/

"jmp 1b\n"

"2:"

:"+Q" (inc), "+m" (lock->slock)

:

:"memory", "cc");

}

* 1. LOCK\_PREFIX 宏在前文中已经介绍过，就是“lock”前缀。
  2. xaddw 汇编指令将 slock 和 inc 的值交换，然后把这两个值相加后的和存到 slock 中。也就是说，该指令执行完毕后，inc 存有原来的 slock 值作为票据序号，而 slock 的 Next 域被加 1。
  3. comb 比较 inc 变量的高位和低位字节是否相等，如果相等，表明锁处于未使用状态，直接跳转到标签 2 的位置退出函数。
  4. 如果锁处于使用状态，则不停地将当前的 slock 的 Owner 域复制到 inc 的低字节处(movb 指令)，然后重复 c 步骤。不过此时 inc 变量的高位和低位字节相等表明轮到自己获取了自旋锁。

1. \_\_raw\_spin\_unlock

**清单 7. \_\_raw\_spin\_unlock 函数**

static inline void \_\_raw\_spin\_unlock(raw\_spinlock\_t \*lock)

{

\_\_asm\_\_ \_\_volatile\_\_(

UNLOCK\_LOCK\_PREFIX "incb %0"

:"+m" (lock->slock)

:

:"memory", "cc");

}

* 1. 在 IA32 体系结构下，如果使用 PPro SMP 系统或者启用了 X86\_OOSTORE，则 UNLOCK\_LOCK\_PREFIX 被定义为“lock”前缀；否则被定义为空。
  2. incb 指令将 slock 最低位字节也就是 Owner 域加 1。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-spinlock/#ibm-pcon)

**Windows 操作系统的排队自旋锁(Queued Spinlock)介绍**

排队自旋锁并不是一个新想法，某些操作系统早已采用了类似概念，只是实现方式有所差别。例如在 Windows 操作系统中排队自旋锁被称为 Queued Spinlock。

Queued Spinlock 的工作方式如下：每个处理器上的执行线程都有一个本地的标志，通过该标志，所有使用该锁的处理器（锁拥有者和等待者）被组织成一个单向队列。当一个处理器想要获得一个已被其它处理器持有的 Queued Spinlock 时，它把自己的标志放在该 Queued Spinlock 的单向队列的末尾。如果当前锁持有者释放了自旋锁，则它将该锁移交到队列中位于自己之后的第一个处理器。同时，如果一个处理器正在忙等待 Queued Spinlock，它并不是检查该锁自身的状态，而是检查针对自己的标志；在队列中位于该处理器之前的处理器释放自旋锁时会设置这一标志，以表明轮到这个正在等待的处理器了。

与 Linux 的排队自旋锁相比，Queued Spinlock 的设计更为复杂，但是 Queued Spinlock 拥有自己的优势：

1. 忙等待 Queued Spinlock 的每个处理器在针对该处理器的标志上旋转，而不是在全局的自旋锁上测试旋转，因此处理器之间的同步比 Linux 的排队自旋锁少得多。
2. Queued Spinlock 拥有真实的队列结构，因此便于扩充更高级的功能。