第二次实例分析

晏悦，任轶慧，徐逸斌，付娆

1. xv6中自旋锁 spinlock 的数据结构以及重要字段的作用

```
1  // Mutual exclusion lock.
2  struct spinlock {
3    uint locked;       // Is the lock held?
4
5    // For debugging:
6    char *name;        // Name of lock.
7    struct cpu *cpu;   // The cpu holding the lock.
8    uint pcs[10];      // The call stack (an array of program counters)
9                       // that locked the lock.
10 };
```

Locked:锁是否被持有？
1:被持有
0：不被持有

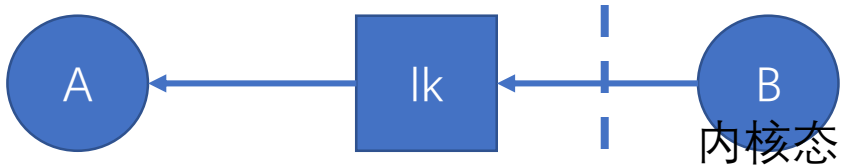
2. acquire 函数获取锁的时候，为什么首先进行要执行 pushcli “关中断”操作?

```
20  // Acquire the lock.
21  // Loops (spins) until the lock is acquired.
22  // Holding a lock for a long time may cause
23  // other CPUs to waste time spinning to acquire it.
24  void
25  acquire(struct spinlock *lk)
26  {
27    pushcli(); // disable interrupts to avoid deadlock.
28    if(holding(lk))
29      panic("acquire");
30
31    // The xchg is atomic.
32    while(xchg(&lk->locked, 1) != 0)
33      ;
34
35    // Tell the C compiler and the processor to not move loa
36    // past this point, to ensure that the critical section'
37    // references happen after the lock is acquired.
38    __sync_synchronize();
39
40    // Record info about lock acquisition for debugging.
41    lk->cpu = mycpu();
42    getcallerpcs(&lk, lk->pcs);
43  }
```

理论上，如果有中断程序可能获得锁lk，则需要在acquire()开始处关中断。否则可能造成死锁。

实例：

a. 应用程序A持有锁lk
b. 应用程序A执行的过程中，出现了中断
c. 系统陷入内核，执行中断处理程序B
d. 中断处理程序B执行的过程中，申请锁lk
e. 但此时锁被应用程序A持有，造成死锁



3. 内联汇编函数 xchg 返回 lk->locked 在 xchg 执行之前的值，考虑两个core上的分别有一个进程先后执行 xchg ，返回值分别为0和1时，这两个进程分别发生了什么？

结合这个例子，说明 xchg指令是原子指令的重要性

```
20  // Acquire the lock.
21  // Loops (spins) until the lock is acquired.
22  // Holding a lock for a long time may cause
23  // other CPUs to waste time spinning to acquire it.
24  void
25  acquire(struct spinlock *lk)
26  {
27    pushcli(); // disable interrupts to avoid deadlock.
28    if(holding(lk))
29      panic("acquire");
30
31    // The xchg is atomic.
32    while(xchg(&lk->locked, 1) != 0)
33      ;
34
35    // Tell the C compiler and the processor to not move load
36    // past this point, to ensure that the critical section's
37    // references happen after the lock is acquired.
38    __sync_synchronize();
39
40    // Record info about lock acquisition for debugging.
41    lk->cpu = mycpu();
42    getcallerpcs(&lk, lk->pcs);
43  }
```

Xchg指令：交换arg1 和 arg2 中的内容，返回交换前arg1的值。

(1)锁被占用: lk->locked == 1, 返回值为1，1写入lk->locked，继续while循环内申请锁；

(2)锁未被占用：lk->locked == 0, 返回值为0，1写入lk->locked，跳出while循环。

如果xchg为原子指令：

(1)A进程申请锁lk，lk->locked==0, 将lk->locked改写为1，返回0；

(2)B进程申请锁lk，lk->locked==1，返回1；

(3)进程A获得锁，进程B继续申请。

3. 内联汇编函数 xchg 返回 lk->locked 在 xchg 执行之前的值，考虑两个core上的分别有一个进程先后执行 xchg ，返回值分别为0和1时，这两个进程分别发生了什么？

结合这个例子，说明 xchg指令是原子指令的重要性

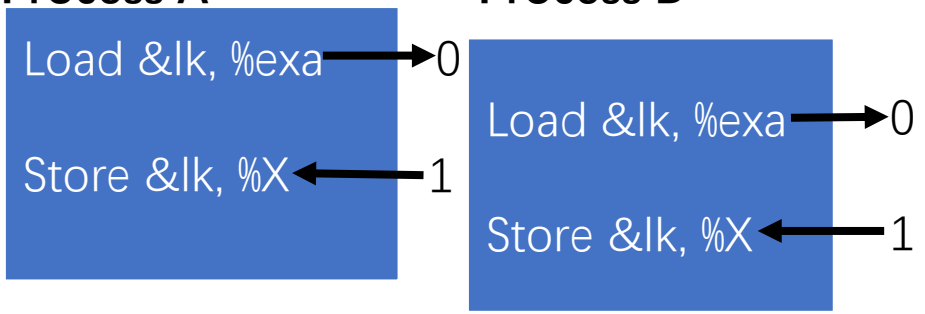
```
20  // Acquire the lock.
21  // Loops (spins) until the lock is acquired.
22  // Holding a lock for a long time may cause
23  // other CPUs to waste time spinning to acquire it.
24  void
25  acquire(struct spinlock *lk)
26  {
27    pushcli(); // disable interrupts to avoid deadlock.
28    if(holding(lk))
29      panic("acquire");
30
31    // The xchg is atomic.
32    while(xchg(&lk->locked, 1) != 0)
33      ;
34
35    // Tell the C compiler and the processor to not move load
36    // past this point, to ensure that the critical section's
37    // references happen after the lock is acquired.
38    __sync_synchronize();
39
40    // Record info about lock acquisition for debugging.
41    lk->cpu = mycpu();
42    getcallerpcs(&lk, lk->pcs);
43  }
```

Xchg指令：交换arg1 和 arg2 中的内容，返回交换前arg1的值。

(1)锁被占用: lk->locked == 1, 返回值为1，1写入lk->locked，继续while循环内申请锁；

(2)锁未被占用：lk->locked == 0, 返回值为0，1写入lk->locked，跳出while循环。

如果xchg不是原子指令：



两个进程都返回1，两个进程都获得锁。

4. 在编译后的xv6源码目录下执行 `objdump -d spinlock.o` ， 观察 `acquire` 函数中， `xchg` 函数和它周围的while循环被编译成了什么汇编代码？试将该汇编代码和C代码对应起来。

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move l
  // past this point, to ensure that the critical sectio
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

```
00000170 <acquire>:
 170:   55                      push   %ebp
 171:   89 e5                   mov    %esp,%ebp
 173:   53                      push   %ebx
 174:   83 ec 14                sub    $0x14,%esp
 177:   e8 fc ff ff ff          call   178 <acquire+0x8>
 17c:   8b 45 08                mov    0x8(%ebp),%eax
 17f:   89 04 24                mov    %eax,(%esp)
 182:   e8 fc ff ff ff          call   183 <acquire+0x13>
 187:   85 c0                   test   %eax,%eax
 189:   75 3c                   jne    1c7 <acquire+0x57>
 18b:   b9 01 00 00 00          mov    $0x1,%ecx
 190:   8b 55 08                mov    0x8(%ebp),%edx
 193:   89 c8                   mov    %ecx,%eax
 195:   f0 87 02                lock xchg %eax,(%edx)
 198:   85 c0                   test   %eax,%eax
 19a:   75 f4                   jne    190 <acquire+0x20>
 19c:   f0 83 0c 24 00          lock orl $0x0,(%esp)
 1a1:   8b 5d 08                mov    0x8(%ebp),%ebx
 1a4:   e8 fc ff ff ff          call   1a5 <acquire+0x35>
 1a9:   89 43 08                mov    %eax,0x8(%ebx)
 1ac:   8b 45 08                mov    0x8(%ebp),%eax
 1af:   83 c0 0c                add    $0xc,%eax
 1b2:   89 44 24 04             mov    %eax,0x4(%esp)
 1b6:   8d 45 08                lea    0x8(%ebp),%eax
 1b9:   89 04 24                mov    %eax,(%esp)
 1bc:   e8 fc ff ff ff          call   1bd <acquire+0x4d>
 1c1:   83 c4 14                add    $0x14,%esp
 1c4:   5b                      pop    %ebx
 1c5:   5d                      pop    %ebp
 1c6:   c3                      ret
```

5. 查阅内联汇编相关文档以及i386手册，结合objdump的输出，解释内联汇编函数 `xchg` (0568~0679)

内联汇编

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write o
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

防止GCC优化

```
00000170 <acquire>:
 170:   55                      push   %ebp
 171:   89 e5                   mov    %esp,%ebp
 173:   53                      push   %ebx
 174:   83 ec 14                sub    $0x14,%esp
 177:   e8 fc ff ff ff          call   178 <acquire+0x8>
 17c:   8b 45 08                mov    0x8(%ebp),%eax
 17f:   89 04 24                mov    %eax,(%esp)
 182:   e8 fc ff ff ff          call   183 <acquire+0x13>
 187:   85 c0                   test   %eax,%eax
 189:   75 3c                   jne    1c7 <acquire+0x57>
 18b:   b9 01 00 00 00          mov    $0x1,%ecx
 190:   8b 55 08                mov    0x8(%ebp),%edx
 193:   89 c8                   mov    %ecx,%eax
 195:   f0 87 02                lock xchg %eax,(%edx)
 198:   85 c0                   test   %eax,%eax
 19a:   75 f4                   jne    190 <acquire+0x20>
 19c:   f0 83 0c 24 00          lock orl $0x0,(%esp)
 1a1:   8b 5d 08                mov    0x8(%ebp),%ebx
 1a4:   e8 fc ff ff ff          call   1a5 <acquire+0x35>
 1a9:   89 43 08                mov    %eax,0x8(%ebx)
 1ac:   8b 45 08                mov    0x8(%ebp),%eax
 1af:   83 c0 0c                add    $0xc,%eax
 1b2:   89 44 24 04             mov    %eax,0x4(%esp)
 1b6:   8d 45 08                lea    0x8(%ebp),%eax
 1b9:   89 04 24                mov    %eax,(%esp)
 1bc:   e8 fc ff ff ff          call   1bd <acquire+0x4d>
 1c1:   83 c4 14                add    $0x14,%esp
 1c4:   5b                      pop    %ebx
 1c5:   5d                      pop    %ebp
 1c6:   c3                      ret
```

5. 查阅内联汇编相关文档以及i386手册，结合objdump的输出，解释内联汇编函数 `xchg` (0568~0679)

内联汇编

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write o
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

输入

```
00000170 <acquire>:
 170:   55                      push   %ebp
 171:   89 e5                   mov    %esp,%ebp
 173:   53                      push   %ebx
 174:   83 ec 14                sub    $0x14,%esp
 177:   e8 fc ff ff ff          call   178 <acquire+0x8>
 17c:   8b 45 08                mov    0x8(%ebp),%eax
 17f:   89 04 24                mov    %eax,(%esp)
 182:   e8 fc ff ff ff          call   183 <acquire+0x13>
 187:   85 c0                   test   %eax,%eax
 189:   75 3c                   jne    1c7 <acquire+0x57>
 18b:   b9 01 00 00 00          mov    $0x1,%ecx
 190:   8b 55 08                mov    0x8(%ebp),%edx
 193:   89 c8                   mov    %ecx,%eax
 195:   f0 87 02                lock xchg %eax,(%edx)
 198:   85 c0                   test   %eax,%eax
 19a:   75 f4                   jne    190 <acquire+0x20>
 19c:   f0 83 0c 24 00          lock orl $0x0,(%esp)
 1a1:   8b 5d 08                mov    0x8(%ebp),%ebx
 1a4:   e8 fc ff ff ff          call   1a5 <acquire+0x35>
 1a9:   89 43 08                mov    %eax,0x8(%ebx)
 1ac:   8b 45 08                mov    0x8(%ebp),%eax
 1af:   83 c0 0c                add    $0xc,%eax
 1b2:   89 44 24 04             mov    %eax,0x4(%esp)
 1b6:   8d 45 08                lea    0x8(%ebp),%eax
 1b9:   89 04 24                mov    %eax,(%esp)
 1bc:   e8 fc ff ff ff          call   1bd <acquire+0x4d>
 1c1:   83 c4 14                add    $0x14,%esp
 1c4:   5b                      pop    %ebx
 1c5:   5d                      pop    %ebp
 1c6:   c3                      ret
```

5. 查阅内联汇编相关文档以及i386手册，结合objdump的输出，解释内联汇编函数 xchg (0568~0679)

内联汇编

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write o
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

两个输出值
‘+’可读可输出，‘=’输出
‘m’内存，‘a’寄存器%exa

```
00000170 <acquire>:
 170:   55                      push   %ebp
 171:   89 e5                   mov    %esp,%ebp
 173:   53                      push   %ebx
 174:   83 ec 14                sub    $0x14,%esp
 177:   e8 fc ff ff ff          call   178 <acquire+0x8>
 17c:   8b 45 08                mov    0x8(%ebp),%eax
 17f:   89 04 24                mov    %eax,(%esp)
 182:   e8 fc ff ff ff          call   183 <acquire+0x13>
 187:   85 c0                   test   %eax,%eax
 189:   75 3c                   jne    1c7 <acquire+0x57>
 18b:   b9 01 00 00 00          mov    $0x1,%ecx
 190:   8b 55 08                mov    0x8(%ebp),%edx
 193:   89 c8                   mov    %ecx,%eax
 195:   f0 87 02                lock xchg %eax,(%edx)
 198:   85 c0                   test   %eax,%eax
 19a:   75 f4                   jne    190 <acquire+0x20>
 19c:   f0 83 0c 24 00          lock orl $0x0,(%esp)
 1a1:   8b 5d 08                mov    0x8(%ebp),%ebx
 1a4:   e8 fc ff ff ff          call   1a5 <acquire+0x35>
 1a9:   89 43 08                mov    %eax,0x8(%ebx)
 1ac:   8b 45 08                mov    0x8(%ebp),%eax
 1af:   83 c0 0c                add    $0xc,%eax
 1b2:   89 44 24 04             mov    %eax,0x4(%esp)
 1b6:   8d 45 08                lea    0x8(%ebp),%eax
 1b9:   89 04 24                mov    %eax,(%esp)
 1bc:   e8 fc ff ff ff          call   1bd <acquire+0x4d>
 1c1:   83 c4 14                add    $0x14,%esp
 1c4:   5b                      pop    %ebx
 1c5:   5d                      pop    %ebp
 1c6:   c3                      ret
```

5. 查阅内联汇编相关文档以及i386手册，结合objdump的输出，解释内联汇编函数 `xchg` (0568~0679)

内联汇编

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write o
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

“Lock”指令前缀
保证了这条指令对总线和缓存的独占权，
保证了这条指令的执行过程中不会有其他CPU
或同CPU内的指令访问缓存和内存

```
00000170 <acquire>:
 170:   55                      push   %ebp
 171:   89 e5                   mov    %esp,%ebp
 173:   53                      push   %ebx
 174:   83 ec 14                sub    $0x14,%esp
 177:   e8 fc ff ff ff          call   178 <acquire+0x8>
 17c:   8b 45 08                mov    0x8(%ebp),%eax
 17f:   89 04 24                mov    %eax,(%esp)
 182:   e8 fc ff ff ff          call   183 <acquire+0x13>
 187:   85 c0                   test   %eax,%eax
 189:   75 3c                   jne    1c7 <acquire+0x57>
 18b:   b9 01 00 00 00          mov    $0x1,%ecx
 190:   8b 55 08                mov    0x8(%ebp),%edx
 193:   89 c8                   mov    %ecx,%eax
 195:   f0 87 02                lock xchg %eax,(%edx)
 198:   85 c0                   test   %eax,%eax
 19a:   75 f4                   jne    190 <acquire+0x20>
 19c:   f0 83 0c 24 00          lock orl $0x0,(%esp)
 1a1:   8b 5d 08                mov    0x8(%ebp),%ebx
 1a4:   e8 fc ff ff ff          call   1a5 <acquire+0x35>
 1a9:   89 43 08                mov    %eax,0x8(%ebx)
 1ac:   8b 45 08                mov    0x8(%ebp),%eax
 1af:   83 c0 0c                add    $0xc,%eax
 1b2:   89 44 24 04             mov    %eax,0x4(%esp)
 1b6:   8d 45 08                lea    0x8(%ebp),%eax
 1b9:   89 04 24                mov    %eax,(%esp)
 1bc:   e8 fc ff ff ff          call   1bd <acquire+0x4d>
 1c1:   83 c4 14                add    $0x14,%esp
 1c4:   5b                      pop    %ebx
 1c5:   5d                      pop    %ebp
 1c6:   c3                      ret
```

6. acquire 和 release函数中的__sync_synchronize() 语句的作用是什么？在汇编程序中是如何体现的？

- 设置一个内存屏障（memory barrier），防止编译器reorder的时候把临界区的访存指令移到锁操作之前
- 该语句被编译为lock orl $0x0, (%esp)指令，lock指令前缀…

```
45  // Release the lock.
46  void
47  release(struct spinlock *lk)
48  {
49    if(!holding(lk))
50      panic("release");
51
52    lk->pcs[0] = 0;
53    lk->cpu = 0;
54
55    // Tell the C compiler and the processor to not move loads or stores
56    // past this point, to ensure that all the stores in the critical
57    // section are visible to other cores before the lock is released.
58    // Both the C compiler and the hardware may re-order loads and
59    // stores; __sync_synchronize() tells them both not to.
60    __sync_synchronize();
61
62    // Release the lock, equivalent to lk->locked = 0.
63    // This code can't use a C assignment, since it might
64    // not be atomic. A real OS would use C atomics here.
65    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
66
67    popcli();
68  }
```

```
195  00000210 <release>:
196   210:   55                      push   %ebp
197   211:   89 e5                   mov    %esp,%ebp
198   213:   53                      push   %ebx
199   214:   83 ec 10                sub    $0x10,%esp
200   217:   8b 5d 08                mov    0x8(%ebp),%ebx
201   21a:   53                      push   %ebx
202   21b:   e8 fc ff ff ff          call   21c <release+0xc>
203   220:   83 c4 10                add    $0x10,%esp
204   223:   85 c0                   test   %eax,%eax
205   225:   74 22                   je     249 <release+0x39>
206   227:   c7 43 0c 00 00 00 00    movl   $0x0,0xc(%ebx)
207   22e:   c7 43 08 00 00 00 00    movl   $0x0,0x8(%ebx)
208   235:   f0 83 0c 24 00          lock orl $0x0,(%esp)
209   23a:   c7 03 00 00 00 00       movl   $0x0,(%ebx)
210   240:   8b 5d fc                mov    -0x4(%ebp),%ebx
211   243:   c9                      leave
212   244:   e9 77 fe ff ff          jmp    c0 <popcli>
213   249:   83 ec 0c                sub    $0xc,%esp
214   24c:   68 26 00 00 00          push   $0x26
215   251:   e8 fc ff ff ff          call   252 <release+0x42>
216
```

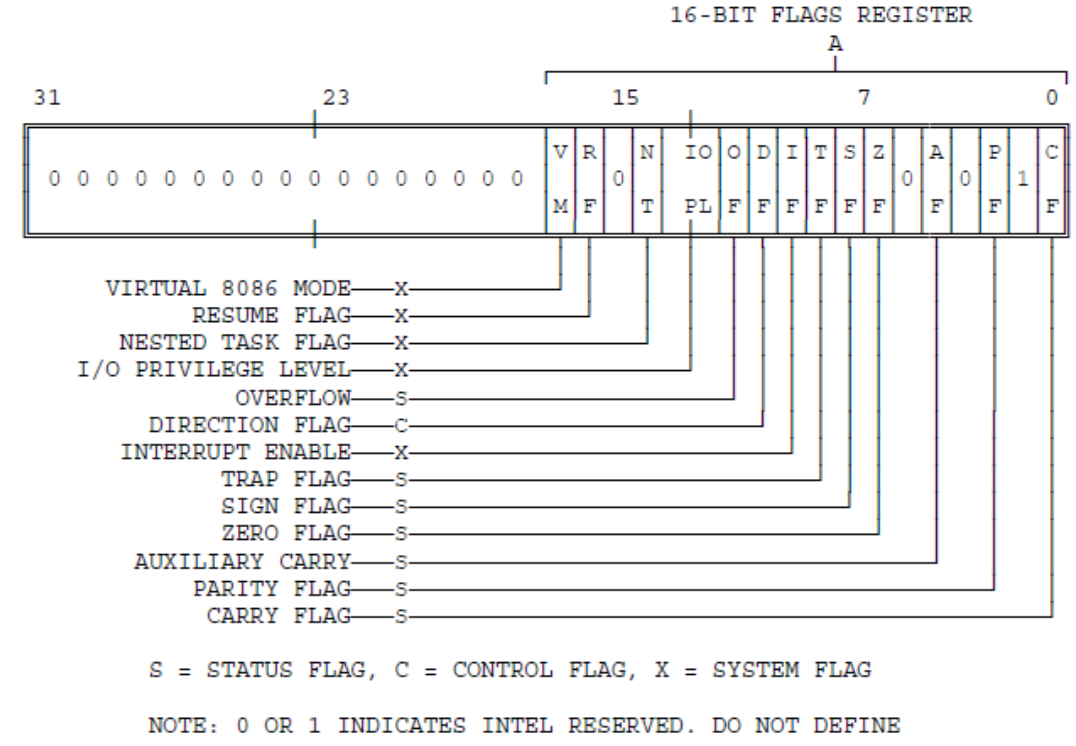
7. 查阅i386手册，了解pushcli函数中的EFLAGS和FL_IF分别是什么？sti和cli内联汇编函数是如何影响前者的？

- EFLAGS：32位的标志位寄存器
- FL_IF：xv6自定义的宏，与EFLAGS相与可以得到中断位IF

```
1   // This file contains definitions for the
2   // x86 memory management unit (MMU).
3
4   // Eflags register
5   #define FL_IF           0x00000200      // Interrupt Enable
6
```

```
104 void
105 pushcli(void)
106 {
107   int eflags;
108
109   eflags = readeflags();
110   cli();
111   if(mycpu()->ncli == 0)
112     mycpu()->intena = eflags & FL_IF;
113   mycpu()->ncli += 1;
114 }
```

Figure 2-8. EFLAGS Register



7. 查阅i386手册，了解pushcli函数中的EFLAGS和FL_IF分别是什么？sti和cli内联汇编函数是如何影响前者的？

- STI：设置 EFLAGS 寄存器中的中断标志 (IF)。设置 IF 标志之后，处理器在执行下一个指令之后，开始响应可屏蔽的外部中断。（例如STI+RET）
- CLI：如果CPL小于等于IOPL，清除EFLAGS寄存器中的IF标志，否则不清除，EFLAGS的其他标志位不会受到影响。

8. mycpu()->intena变量的作用是什么？

- 表示在调用pushcli函数之前是否允许中断
- pushcli和popcli的调用过程…

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

```
void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
```

```
void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

Linux 锁相关代码分析(x86版本)

- 1. x86实现原子指令
- 方法: x86提供了附加的lock前缀，使带**lock前缀**的读修改写指令能原子性执行。
- 实现原理: 带lock前缀的指令在操作时会**锁住总线**，使自身的执行即使在多处理器间也是原子性执行的。
- API使用: 原子性操作是线程间同步的基础，linux专门定义了一种只进行原子操作的类型atomic_t，并提供相关的原子读写调用API。

atomic_t数据结构与API

- typedef struct {
- 	volatile int counter;
- } atomic_t;
- 原子类型其实是int类型，只是禁止寄存器对其暂存
- static inline int atomic_read(const atomic_t *v)
- {
- 	return v->counter;
- }
- static inline void atomic_set(atomic_t *v, int i)
- {
- 	v->counter = i;
- }
- 单独读或者写, 在x86下是原子性的

原子加, 原子减

```
static inline void atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
                 : "+m" (v->counter)
                 : "ir" (i));
}

static inline void atomic_sub(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "subl %1,%0"
                 : "+m" (v->counter)
                 : "ir" (i));
}
```

LOCK_PREFIX
原子指令前缀

counter值加上i 再存回counter

linux FIFO ticket-based spinlock

- 1. 普通自旋锁的缺陷: 由于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。
- 2. 改进方法: 先申请锁的排一个号码, 当锁释放时候,然后按照号码来调度号码小的任务(类似去银行办理服务, 假设只有一个窗口,每个人有排队号码)
- 3. 数据结构: next 与 owner 域, 均初始化为0
- next: 等待服务的任务数目,
- owner: 已经服务完成的任务数目

next:　等待服务的任务数目,
owner: 已经服务完成的任务数目

```
acquire{
    tmp = fetch_and_add(next)
    while( tmp  != owner )
        ;
    }

release{
  atomic_add(owner);
}
```

next++,
返回原来值
新来了一个服务

当前完成任务不等于等待服务任务数, 忙等待(直到排队号到了)

服务完成任务数++