

互斥

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组



互斥锁



问题：怎样让 $\text{sum} == 2 * N$?

- 让 $\text{sum}++$ 被“保护起来”就行了

```
1 void thread1() {
2     for (int i = 0; i < N; i++) {
3         lock
4         sum++;
5         unlock
6     }
7 }
8
9 void thread2() {
10    for (int i = 0; i < N; i++) {
11        lock
12        sum++;
13        unlock
14    }
15 }
```

Lock/Unlock的语义

- lock(l) – 上锁
- unlock(l) – 解锁
- 保证lock-unlock(l)之间的区域 – 临界区Critical Section
 - 要么不被执行
 - 要么就不会与其他lock-unlock(l)并发执行
- 如果多个线程同时试图上锁，则只有一个能获得锁

互斥锁的实现 (1)



互斥锁：为什么难实现？

- 如果锁已经有人持有
 - 这好办，等他出来就行了
- 就怕两个人同时.....



实现1：借助机器指令

- 思考题：
 - 这么实现锁是否正确？
 - 这么实现锁有什么缺陷？

```
1 static inline void lock() {  
2     asm volatile ("cli");  
3 }  
4  
5 static inline void unlock() {  
6     asm volatile ("sti");  
7 }
```

实现2: 等待 & 借助一个flag

- 思考题: 这个实现有问题吗?

```
1 typedef struct { int flag; } lock_t;
2
3 void lock(lock_t *mutex) {
4     while (mutex->flag == 1) ;
5     mutex->flag = 1;
6 }
7
8 void unlock(lock_t *mutex) {
9     mutex->flag = 0;
10 }
```


实现3：借助硬件指令

- 硬件为我们提供了“原子”的指令
 - atomic test & set (TAS) / atomic exchange (XCHG)
 - 怎样用tas/xchg实现lock/unlock?

```
1 int tas(int *ptr) {
2     int value = *ptr;
3     if (value == 0) {
4         *ptr = 1;
5     } else {
6         // do nothing
7     }
8     return value;
9 }
```

```
1 int xchg(int *ptr, int newval) {
2     int value = *ptr;
3     *ptr = newval;
4     return value;
5 }
```

```
typedef struct { int flag; } lock_t;
void lock(lock_t *mutex);
void unlock(lock_t *mutex);
```

实现4: xv6 spinlock.c

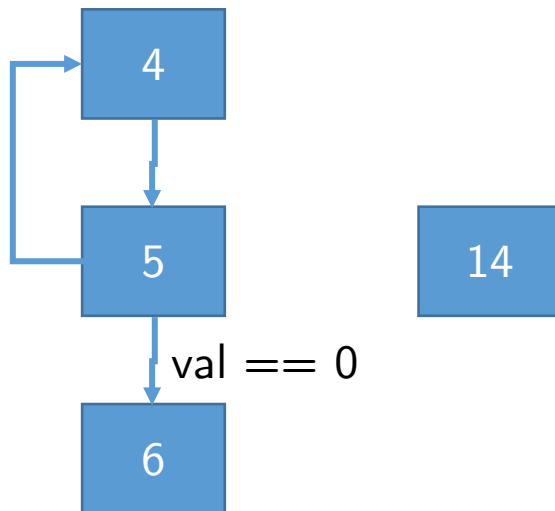
- 代码省略了调试信息
 - pushcli \approx 关闭当前处理器中断; popcli \approx 打开当前处理器中断
 - 为什么它是对的?

```
1 void
2 acquire(struct spinlock *lk) {
3     pushcli();
4     while(xchg(&lk->locked, 1) != 0)
5         ;
6 }
7
8 void
9 release(struct spinlock *lk)
10 {
11     xchg(&lk->locked, 0);
12     popcli();
13 }
```

xv6 spinlock.c 证明

- 回到最初的想法：程序 = 状态机 (model checking)

- 全局状态：lk->locked (0, 1)
- 线程本地的状态：{PC, val}
- 每个线程每次执行一条语句




```
1 void
2 acquire(struct spinlock *lk) {
3     while(1) {
4         int val = xchg(&lk->locked, 1);
5         if (val == 0) {
6             break;
7         }
8     }
9 }
10
11 void
12 release(struct spinlock *lk)
13 {
14     xchg(&lk->locked, 0);
15 }
```

且慢.....

- 还记得上次黑人问号的例子吗？我们的假设成立吗？

```
for (int i = 0; i < N; i++) {  
    lock(&lk);  
  
    sum++;  
    unlock(&lk);  
}
```



```
13 void thread1() {  
14     x = 1;  
15     read(y); // y = 0  
16 }  
17  
18 void thread2() {  
19     y = 1;  
20     read(x); // x = 0  
21 }
```

- lock/unlock兼做串行化(serialization)
 - x86的原子操作xchg包含了串行化
 - x86还提供了LFENCE/SFENCE指令完成串行化
 - asm volatile (" ::: "memory"); 阻止编译器调换内存访问顺序

小结

- 互斥: lock/unlock保护的区域

```
1 void
2 acquire(struct spinlock *lk) {
3     while(1) {
4         int val = xchg(&lk->locked, 1);
5         if (val == 0) {
6             break;
7         }
8     }
9 }
10
11 void
12 release(struct spinlock *lk)
13 {
14     xchg(&lk->locked, 0);
15 }
```

号外: Consensus Number

- lock/unlock里的等待循环非常浪费, 但也无法避免
 - $CN = k$ 只能实现 k 个线程wait-free的consensus
 - 永远不能用CN小的原语在wait-free的前提下实现CN大的
 - load / store无法实现 $O(1)$ 的test & set

Consensus Number	并发对象
1	load / store
2	test & set, xchg, ...
...	...
$2n - 2$	n 变量同时读/写
...	...
∞	compare & swap

互斥锁的实现(2)

如果不给硬件的.....支持呢？

- 以下程序无法正确实现互斥

```
while (locked) ;  
locked = 1;  
// critical section  
locked = 0;
```


两个线程的例子：失败的尝试1

- 能保证互斥(假设寄存器读/写不被乱序)
 - 在不并发的时候正确
 - 在并发的时候可能卡死

```
flag[0] = 1;  
while (flag[1]) ;  
// critical section  
flag[0] = 0;
```

```
flag[1] = 1;  
while (flag[0]) ;  
// critical section  
flag[1] = 0;
```



两个线程的例子：失败的尝试2

- 能保证互斥
 - 在并发的时候能做对了
 - 但不并发的时候会卡死

```
victim = 0;  
while (victim == 0) ;  
// critical section
```

```
victim = 1;  
while (victim == 1) ;  
// critical section
```

Peterson's Algorithm

- 文明礼让：我有兴趣，你先来

```
flag[0] = 1; // 我有兴趣
victim = 0; // 你先来
while (flag[1] && victim == 0) ;
// critical section
flag[0] = 0; // 我失去兴趣了
```

```
flag[1] = 1; // 我有兴趣
victim = 1; // 你先来
while (flag[0] && victim == 1) ;
// critical section
flag[1] = 0; // 我失去兴趣了
```

- 证明
 - 首先不并发的时候是对的
 - 并发的时候.....呢？

互斥锁的实现 (3)



不想自旋？

- 刚才的实现都包含“自旋” (spinning)
 - lock = while (锁被别人占着) 等待
 - 实现互斥，自旋不可避免
- 能否避免自旋？
 - 操作系统中有很多线程可以运行
 - 如果暂时当前线程获取失败，可以让其他线程运行啊！

与操作系统交互

- 如果xchg失败，就通过系统调用告诉操作系统
 - 方法1：把锁操作实现成系统调用
 - 方法2：告诉操作系统我在spin，先让别人跑(yield)
 - 方法3：告诉操作系统我要睡觉啦！等到锁释放的时候叫我
 - 操作系统因为在内核中执行，只要把线程状态设置一下，然后调度其他线程/进程执行就行了
- Linux提供了futex系统调用
 - To reiterate, **bare futexes are not intended as an easy to use abstraction for end-users**. Implementors are expected to be assembly literate and to have read the sources of the futex userspace library referenced below.

在操作系统内部实现互斥锁

- 这是单处理器的版本
- 多处理器的版本呢?

```
void mutex_lock(mutex_t *mutex) {  
    cli();  
    if (mutex->locked == 1) {  
        insert(mutex->queue, current);  
        current->status = BLOCK;  
        _yield();  
    }  
    mutex->locked = 1;  
    sti();  
}
```

```
void mutex_unlock(mutex_t *mutex) {  
    cli();  
    mutex->locked = 0;  
    if (!empty(mutex->queue)) {  
        thread_t *thread = pop(mutex->queue);  
        thread->status = RUNNABLE;  
    }  
    sti();  
}
```

小结：互斥



实现原子性(Atomicity)

- 互斥锁能保证一段代码原子执行不与其他线程并发
- 是一种帮助我们理解并发程序行为的抽象
 - 人类根本无法理解并发程序的行为，必须借助抽象降低难度.....

```
flag[0] = 1; // 我有兴趣  
victim = 0; // 你先来  
while (flag[1] && victim == 0) ;  
// critical section  
flag[0] = 0; // 我失去趣了
```

```
flag[1] = 1; // 我有兴趣  
victim = 1; // 你先来  
while (flag[0] && victim == 1) ;  
// critical section  
flag[1] = 0; // 我失去兴趣了
```

