Class4 并发控制 互斥

实现互斥为什么困难

Peterson 算法运行在假象的计算机上，这个计算机具有顺序性、可见性、原子性三种特性，遗憾的是，在现代计算机上不成立。

```
x=1                    y=1
load(y)                load(x)
```

这两段代码在现在计算机里，store(x)、store(y)的时候，在访存的时候load(x)、load(y) 就已经开始执行了，在把1存进去之前就可能load出1来。而这与peterson算法要求的顺序性矛盾。

编译执行 sum.c 文件

```
gcc sum.c -O2 -I. -lpthread
objdump -d a.out | less
```

多个线程同行执行循环+1 n次时，得出结果小于n也是有可能的！（Why？）

软件不够，硬件来凑

增加硬件，实现同时读写的指令。sum-atomic.c

x86 的原子操作保证：

- 原子性: load/store 不会被打断
- 顺序：线程 (处理器) 执行的乱序只能不能越过原子操作
- 多处理器之间的可见性：若原子操作 $\blacksquare A$ 发生在 $\blacksquare B$ 之前，则 $\blacksquare A$ 之前的 store 对 $\blacksquare B$ 之后的 load 可见

```
int xchg(volatile int *addr, int newval) {
```

```
  int result;
  result = *addr;
  addr = newval;
  return result;
}
// x86 原子操作xchg
// 内联汇编
int xchg(volatile int *addr, int newval) {
  int result;
  asm volatile ("lock xchg %0, %1"
  : "+m"(*addr), "=a"(result) : "1"(newval) : "cc");
  return result;
}
```

用xchg实现互斥

自旋锁

```
int table = KEY;

void lock() {
retry:
  int got = xchg(&table, NOTE);
  if (got != KEY)
    goto retry;
  assert(got == KEY);
}

void unlock() {
  xchg(&table, KEY)
}
```

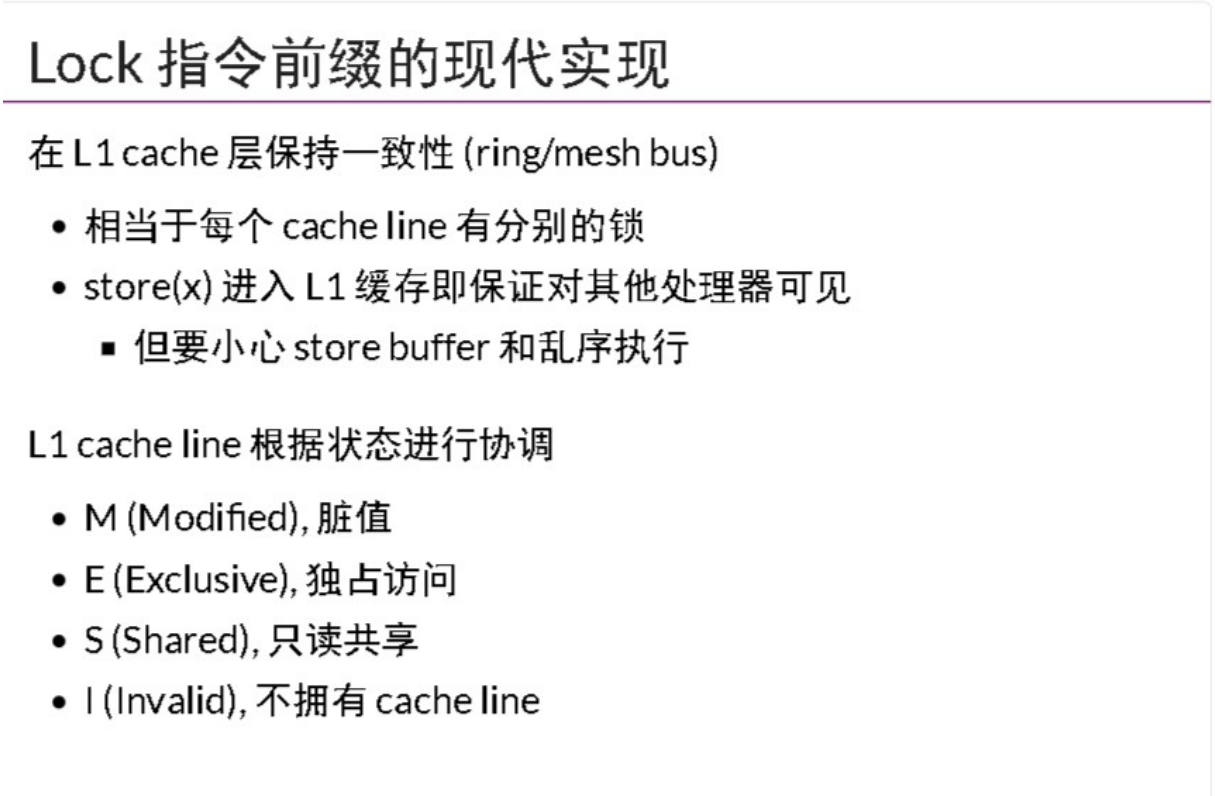
```
int locked = 0;

void lock() {
  while (xchg(&locked, 1)) ;
}

void unlock() {
  xchg(&locked, 0);
}
```

用model-checker检查。运行spinlock.c，测试是否正确实现。

保证了正确性，但是降低了性能。

原子指令的硬件实现

以前的cpu 80486： 锁总线



RISC-V: 另一种原子操作的设计

考虑常见的原子操作：

- atomic test-and-set
  - `reg = load(x); if (reg == XX) { store(x, YY); }`
- lock xchg
  - `reg = load(x); store(x, XX);`
- lock add
  - `t = load(x); t++; store(x, t);`

它们的本质都是：

1. load
2. exec (处理器本地寄存器的运算)
3. store

Load-Reserved/Store-Conditional (LR/SC)

LR: 在内存上标记 reserved (盯上你了)，中断、其他处理器写入都会导致标记消除

```
lr.w rd, (rs1)
  rd = M[rs1]
  reserve M[rs1]
```

SC: 如果“盯上”未被解除，则写入

```
sc.w rd, rs2, (rs1)
  if still reserved:
    M[rs1] = rs2
    rd = 0
  else:
    rd = nonzero
```