Note

并发互斥

1.peterson算法

解决双线程互斥的算法。

```
//Pi进程
flag[i] = True;
turn = j;
while(flag[j] && turn == j);
critical section;
flag[i] = False;
remainder section;
//Pj进程
flag[j] = True;
turn = i;
while(flag[i] && turn == i);
critical section;
flag[j] = False;
remainder section;
```

但是操作系统需要更简单粗暴的方式去解决问题，所以引入锁的概念。

2.锁

假设硬件能为我们提供一个原子操作，这个原子操作能够原子性地完成“读+写”操作，那么我们可以用这个原子操作来实现锁。 比如：x86提供了lock前缀。

(1).自旋锁

自旋锁使用硬件提供的原子指令 `xchg` 来实现。

`xchg` 指令是硬件提供的一个原子指令， 它地效果是将 `eax` 寄存器的值和 `mem` 地址处的值进行交换，然后将 `mem` 地址处的值放到 `eax` 寄存器中。

自旋锁的实现如下：

```
int table = YES;

void lock(){
retry:
   int got = xchg(&table, NO);
   if(got == NO)
      goto retry;
   assert(table == YES);
}

void unlock(){
   xchg(&table, YES);
}
```

(2).硬件上如何实现原子指令

与多核CPU的内存共享有关：在共享的mem上上一个锁即可（lock 指令前缀）。在x86（Bus Lock 80486）中一读到 `lock` 就去总线上锁，直到这个指令执行完毕，才会解锁。这样就保证了在这个指令执行期间，其他CPU不能访问这个内存地址。（那个时候L1 cache在主板上）

但是今天这个时代每个CPU都有自己的L1,L2 cache， 当两个CPU都具有相同的L1 cache时，就会出现问题。所以需要MESI协议来解决这个问题。

RISCV上原子指令的实现：

原子操作本质上是：

- 1.load
- 2.execute
- 3.store
  Load-Reserved/Store-Conditional(LR/SC):

LR:

```
lr.w rd, rs1
   rd = M[rs1]
   reserved M[rs1]
```

中断、其它处理器写入都会导致标记消失。 SC:

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

使用LR/SC以及不断retry可以实现各种原子操作，可以检测并发冲突。

(3).自旋锁的问题

自旋锁的问题在于，当一个线程在等待锁的时候，它会一直占用CPU，这样会导致CPU的资源浪费。所以我们需要一种更好的方式来实现锁。

1、自旋会触发处理器之间的缓存同步。

2、除了进入临界区的线程，其它处理器上的线程都在空转，争抢锁的处理器越多，利用率越低。

3、获得自旋锁的线程可能被操作系统切出去（因为操作系统并不感知线程在做什么），这样实现100%的资源浪费。

自旋锁的使用场景：

1、临界区几乎不拥堵，即临界区的执行时间远远小于自旋锁的等待时间。 2、持有自旋锁时禁止执行流切换。

所以自旋锁的真实使用场景：操作系统内核中的并发数据结构（短临界区）。

(4).互斥锁（Mutex）