互斥：基础
Mutual Exclusion (Mutex): Basics

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

一些关于并发的术语

- 临界区(critical section): 访问共享资源的一段代码，资源通常是一个变量或数据结构。
- 竞态条件(race condition)：出现在多个执行线程大致同时进入临界区时，它们都试图更新共享的数据结构，导致非预期的结果。
  - 当程序含有一个或多个竞态条件，程序的输出会因运行而异，具体取决于哪些线程在何时运行，因此结果是不确定的(non-deterministic)。

如何设计机制使得临界区避免出现竞态条件是“并发”的重要主题！

安全性(Safety)和活性(Liveness)

对于一个**运行**程序(尤其是并发程序)的两个重要属性(by Leslie Lamport):

- 安全性: “**没有坏事**发生”
  - 安全性要求执行中的**任何**有限步骤内都保持这个性质。
    - 如：执行过程中不可出现除0错误
- 活性: “**好事终将**发生”:
  - 活性要求只要在**最终**能满足要求即可，一个隐含的要求是执行中不能发生“不可挽回”的步骤！
    - 如：执行最终停止（执行中间出现一个无限循环就“不可挽回”）

Theorem: Every property defined on an execution of a program is a combination of a safety property and a liveness property
by Alpern and Schneider, 1987

临界区的解决方案需满足的条件

- **互斥(Mutual Exclusion)**: 临界区内最多只能有一个线程(Safety)
- **行进(Progress)**：如果当前临界区内没有线程，并且有线程想要进入临界区，那么最终某个想要进入临界区的线程会进入该临界区(liveness)
- **有界等待(bounded waiting)**: 如果某个线程想要进入临界区，那么其等待的期限有限（期间其他线程进入该临界区次数有上限），不可一直排队等待(Fairness/No starvation)
  - 如果这个上限没有被指定，那么这就是一个Liveness property，其最终会进入！
  - 如果这个上限被指定具体数字，那么这就是一个Safety property，因为任何一次执行的有限步骤内，其等待进入的次数只要超过这个上限就发生了“坏事”！

临界区的解决方案需满足的条件

- 除此之外，还有一个临界区解决方案所需要关心的，那就是性能（performance）
  - 也就是这个方案的开销：**进入**和**退出**该临界区的两个操作应该相对于在该临界区内做的计算而言尽可能的小。
  - 该性质刻画的是一个程序的all the runs的属性！
- 经验法则(Rule of thumb): 当设计并发算法时，优先考虑安全性！(but don't forget liveness and performance).

回归问题

- 对于下面的临界区代码，首先我们要关心的就是互斥：保证每刻最多只有一个线程访问这个临界区

```c
#define N 100000000
#define NUMBER_OF_THREADS 2

long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        sum++;
    }
}

int main() {
    pthread_t threads[NUMBER_OF_THREADS];
    pthread_create(&threads[0], NULL, T_sum,  NULL);
    pthread_create(&threads[1], NULL, T_sum,  NULL);
    for (int i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i],
NULL);}

    printf("sum = %ld\n", sum);
    printf("2*n = %ld\n", 2L * N);
}
```

访问了共享资源，为临界区代码

```c
int x = 0, y = 0;

void T1() {
  x = 1; int t = y; // Store(x); Load(y)
  __sync_synchronize(); // a barrier
  printf("%d", t);
}

void T2() {
  y = 1; int t = x; // Store(y); Load(x)
  __sync_synchronize();
  printf("%d", t);
}
```

思考：printf代码是临界区代码吗？printf线程安全吗？🤔

锁(Locks)

锁(Locks)

- 锁是一个变量，其保存了锁在某一时刻的状态。它要么是可用的（available，或unlocked，或free），表示没有线程持有锁，要么是被占用的（acquired，或locked，或held），表示有一个线程持有锁，正处于临界区。

- 其提供两个配对操作：

  - **lock()/acquire()**: 调用lock()或acquire()尝试获取锁，如果没有其他线程持有锁，该线程会获得锁，进入临界区，否则(该锁已经被持有了)不会返回(该线程会卡在那里)

  - **unlock()/release()**: 调用unlock()或release()，锁就变成可用了（可被获得），之前如果有因获得锁操作没成功卡在那的线程，那么其中一个会进入临界区。

Locks

- 锁为程序员提供了最小程度的调度控制，通过给临界区加锁，保证临界区内只有一个活跃变量（**互斥**）。

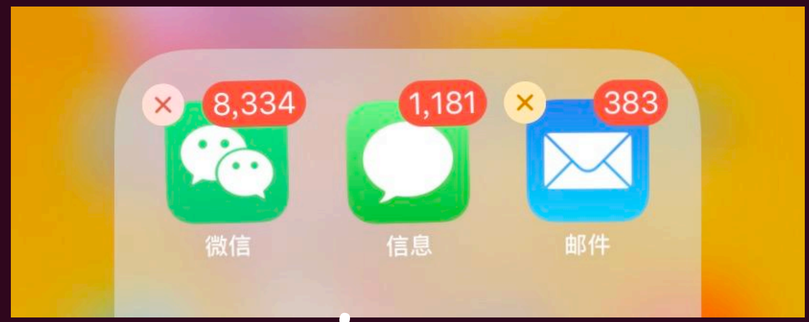
```
long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        lock();
        sum++;
        unlock();
    }
}
```

如何实现这样的锁?

```
int x = 0, y = 0;

void T1() {
  lock();
  x = 1; int t = y; // Store(x); Load(y)
  unlock();
  __sync_synchronize(); // a barrier
  printf("%d", t);
}

void T2() {
  lock();
  y = 1; int t = x; // Store(y); Load(x)
  unlock();
  __sync_synchronize();
  printf("%d", t);
}
```

尝试1:关中断

- 能否使当前程序状态机独占计算机系统?
  - 单处理器系统中 “其他任何事”：仅有中断!
- 一条指令就可以实现原子性（lock() -> disable interrupt）
  - x86: cli 清除 eflags 中的 IF bit
  - RISC-V: 清除 mstatus 的 MIE bit
  - ARM64: msr daifset, #3
- unlock()就是再次打开打开中断



题外话：NMI (Non-Maskable Interrupts)

- 不是所有中断都是可以被屏蔽的，处理器有不可屏蔽中断 (Non-Maskable Interrupts)
  - 主要是为了一些不可处理或通知一些不可恢复的错误
    - 如内部芯片系统错误、系统数据损坏等
  - 如果操作系统的中断处理程序崩掉了（死循环），然后此时又在关中断，那么利用NMI可以监控这个错误，并给出处理（如重启计算机）
    - 可行方法：设置硬件定时触发NMI，操作系统（正常运行下）定时复位定时器以使得NMI不被触发，但操作系统一旦不正常了，timeout会触发NMI

尝试1的问题

- 临界区的代码死循环了⟶整个系统也卡死了
- 中断关闭时间过长会导致很多其他重要的外界响应丢失（比如错过了磁盘I/O的完成事件）
- 关中断是特权指令，用户态的应用是无法执行的，只有操作系统有这个权限！(x86可以看看cs寄存器的最低的两位)
  - 事实上，在操作系统内核的实现中，关闭中断是一个常见的操作
- 多处理器无效（单处理器是可行的）
  - 每个处理器有独立的寄存器组
  - 中断是每个处理器内部状态

尝试2: 通过软件(Lock标志)

- 一个简单的想法：使用一个标志🚩来表达此时锁的状态，比如：为1就是已被占用，为0就是可用的。对应的lock() 和 unlock()

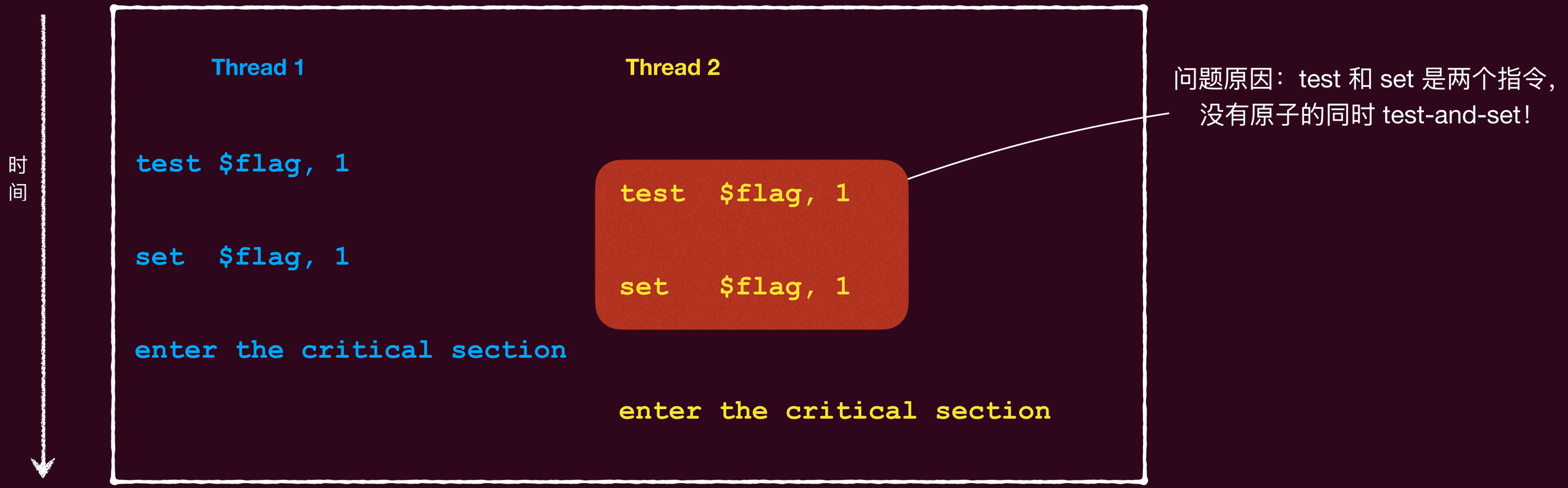
```
int flag = 0;

void lock(){
    while (flag == 1); //自旋的观测flag
    flag = 1; //设置
}

void unlock(){
    flag = 0;
}
```

这里假设 load一个变量并测试它是原子的（flag == 1），
并假设设置一个变量也是原子的 （flag = 1）、（flag = 0）

注： 这两个假设在单处理器时代是合理的设定，但在现代计算系统已经不正确了

尝试2问题

- 这个方法不过是把共享变量存在的竞态条件转移到了锁的这个状态变量上而已
  - 完全存在两个线程同时发现flag为0，然后都进入临界区的可能！ (not safe)



尝试3: 互斥的test！

- 如果每个线程都是采用同一个判定：test flag == 1，那么完全存在所有的线程都可能得到同样的True

  - 但是如果我们给每个线程的判定条件都不一样，且判定条件不可能同时对每个为True呢？ —> 一个线程回答“True”意味着其他线程回答“False”，在逻辑上就达成了互斥！

```
#for thread 1

void lock(){
    while (flag == 0); //自旋的观测flag
}

void unlock(){
    flag = 0;  //设置
}
```

任何时刻判定结果不同！

注：这是在内存顺序一致性模型下才成立

```
#for thread 2

void lock(){
    while (flag == 1);
}

void unlock(){
    flag = 1;
}
```

unlock会给别人机会，自己下一次lock判定就会false，只有别的进程unlock，自己下一次lock才能判定为true, 所以这个方法是让每个线程“严格轮转的”。

尝试3问题

- 这个方法存在一个问题：一个线程能否得到一个锁完全依赖于另外一个线程是否先进入了临界区

```
#for thread 1

void lock(){
    while (flag == 0); //自旋的观测flag
}
void unlock(){
    flag = 0;  //设置
}

void task(){
    while(1){
        lock();
        critical_section_code();
        unlock();
    }
}
```

如果thread 2始终不进入临界区，那么thread 1最多进入一次临界区之后就无法再进入了，违背了[liveness]性质，因为目前没有线程在临界区，而thread 1想进入临界区，但永远无法再进入，因此无法“行进”！

Peterson算法

- 20世纪60年代，Dijkstra向他的朋友们提出了并发问题，他的数学家朋友Theodorus Jozef Dekker想出了一个解决方法，该方法是第一个被证明是正确的“互斥”锁的设计算法

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions. The solution presented here has an extremely simple structure and, as shown later, is easy to prove correct.

“Myths about the mutual exclusion problem”, by Gary L. Peterson, 1981

感受一下其复杂程度

Peterson算法

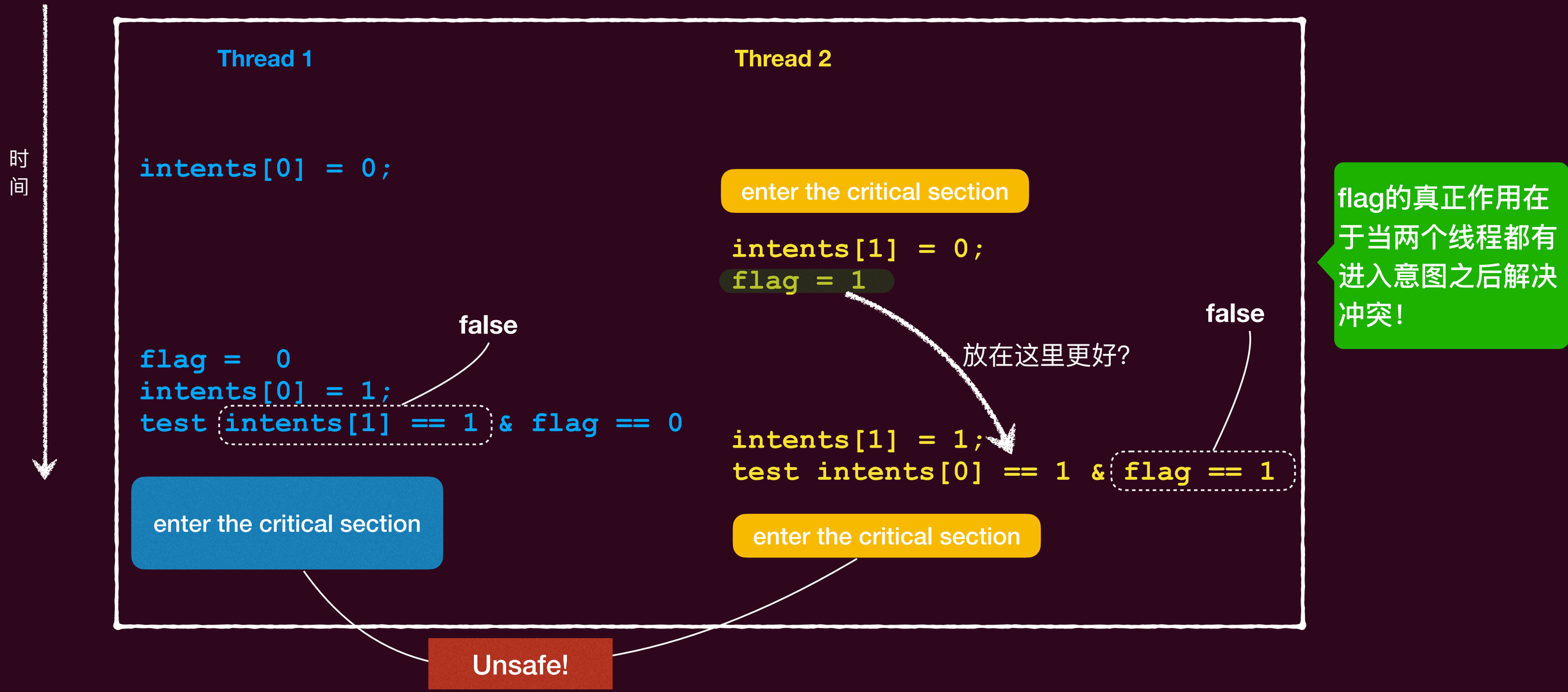
- 1981年, Peterson 发现了一个更加简单的算法
  - 主要思想就是之前尝试3中的test的改进：除了测试flag之外，还看看是否有别的线程要进入临界区（这个部分可以避免之前尝试出现的可能没有“行进”问题）
    - 因此，除了有一个全局变量来进行 flag == i 判定，还得有一个全局变量 intents 来记录是否有其他线程要进入临界区
  - Peterson原始算法只能在两个线程上工作，不过该算法很容易拓展到*N*个线程上（当然，*N*必须提前知道）。

Peterson算法

- Peterson想法的初步实现，下面这个实现对吗?

```
int intents[2] = {0, 0}; //进入意图
int flag = 0;  // 当前是谁可以获得锁 (thread 0 or 1)

int self = i; //当前的线程ID, 0,  1中的一个

void lock() {
    intents[self] = 1;  // 标记自己想要进入临界区
    while (  (intents[(self + 1)%2] == 1)  //别人想不想要进来
          && (flag == self) ); // 如果别人也想进来，就再看看是否当前“轮转”到自己了
}

void unlock() {
  intents[self] = 0; //标记自己不再想要进入临界区
  flag = self; //轮转到下一位
}
```

Peterson算法


时间
Thread 1
Thread 2
intents[0] = 0;
enter the critical section
intents[1] = 0;
flag = 1
false
flag = 0
intents[0] = 1;
test intents[1] == 1 & flag == 0
放在这里更好?
false
intents[1] = 1;
test intents[0] == 1 & flag == 1
enter the critical section
enter the critical section
Unsafe!
flag的真正作用在于当两个线程都有进入意图之后解决冲突!

Peterson算法

- Peterson的一个正确实现

```
int intents[2] = {0, 0}; //进入意图
int flag = 0;  // 当前是谁可以获得锁（thread 0 or 1）

int self = i; //当前的线程ID，0，  1中的一个

void lock() {
    intents[self] = 1;  // 标记自己想要进入临界区
    flag = self; //轮转到下一位
    while (  (intents[(self + 1)%2] == 1)  //别人想不想要进来
          && (flag == self) ); // 如果别人也想进来，就再看看是否当前“轮转”到自己了
}

void unlock() {
  intents[self] = 0; //标记自己不再想要进入临界区
}
```

证明Peterson算法的正确性

- Lemma 1: 当一个线程T_i在调用lock()后和在离开临界区之前: intents[self] = 1
- Proof: trivial

```
int intents[2] = {0, 0}; //进入意图
int flag = 0;  // 当前是谁可以获得锁（thread 0 or 1）

int self = i; //当前的线程ID，0，  1中的一个

void lock() {
    intents[self] = 1;  // 标记自己想要进入临界区
    flag = self; //轮转到下一位
    while (  (intents[(self + 1)%2] == 1)  //别人想不想要进来
          && (flag == self) ); // 如果别人也想进来，就再看看是否当前“轮转”到自己了
}

void unlock() {
  intents[self] = 0; //标记自己不再想要进入临界区
}
```

只有调用unlock()才用

证明Peterson算法的正确性

- Lemma 2: [安全性]Peterson算法能够保持互斥性：
- Proof: 为了方便起见，一个状态记为：$[t, h, k, f_0, f_1]$
  - t当前的flag的值
  - h是当前线程T_1的语句的index
  - k是当前线程T_2的语句的index
  - f_0是intents[0]
  - f_1是intents[1]

T_i works as follows:

```
do{
1.    intents[i] = 1;
2.    flag = i;
3.    while (  (intents[(i + 1)%2] == 1)
           && (flag == i) );
4.    critical_section();
5.    intents[i] = 0;
6.    reminder_section();
}while(1);
```

证明Peterson算法的正确性

T_i works as follows:

```
do{
 1.   intents[i] = 1;
 2.   flag = i;
 3.   while (  (intents[(i + 1)%2] == 1)
           && (flag == i) );
 4.   critical_section();
 5.   intents[i] = 0;
 6.   reminder_section();
}while(1);
```

- Lemma 2: [安全性]Peterson算法能够保持互斥性：
- Proof:
- 不失一般性，我们假设状态[0, 4, 4, 1, 1]发生了，意味着T_1和T_2都进入了临界区，由于最后flag = 0，最后进入临界区的线程的判定语句一定为 (intents[(i + 1)%2] == 1) && (0 == i)

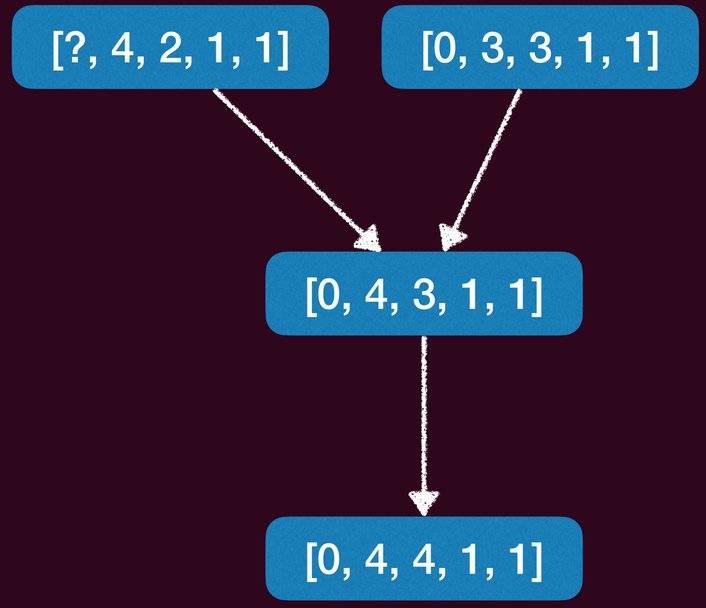
根据Lemma1
这部分是true

那这个只能是False了！，因此最后进入的是T_2

- 因此，[0, 4, 4, 1, 1]的前一个状态是[0, 4, 3, 1, 1]，即T_2一定是从状态[0, 4, 3, 1, 1]之后进入了临界区。

证明Peterson算法的正确性

- 状态[0,4,3,1,1]有两个可能的前驱
  - 一个前驱是[0,3,3,1,1]： T_1也等待在判定语句
    - 然而如果是[0,3,3,1,1]，T_1的loop条件是False, 其无法进入临界区
  - 另一个前驱是[?, 4, 2, 1,1]：T_1早早进入了临界区，T_2再往后退一步
    - 然而基于[?,4,2,1,1]，T_2的语句2会将flag变为1，不会有下一步的flag为0!
- 由于这两个可能都矛盾了，因此假设不成立，由于一般性，因此，Peterson算法是互斥的!



证明Peterson算法的正确性

- Peterson算法能够保持“行进”吗？显然可以！

```
int intents[2] = {0, 0}; //进入意图
int flag = 0;  // 当前是谁可以获得锁（thread 0 or 1）

int self = i; //当前的线程ID，0， 1中的一个

void lock() {
    intents[self] = 1;  // 标记自己想要进入临界区
    flag = self; //轮转到下一位
    while ( (intents[(self + 1)%2] == 1) //别人想不想要进来
          && (flag == self) ); // 如果别人也想进来，就再看看是否当前“轮转”到自己了
}

void unlock() {
  intents[self] = 0; //标记自己不再想要进入临界区
}
```

没有其他线程要进入，这个地方直接返回False

证明Peterson算法的正确性

- Peterson算法能够保持“有界等待”吗?
- 对于只有两个线程的情况下，这是显然的，因为一个线程只要表达出要进入临界区的意愿，最多只要等一轮

```
int intents[2] = {0, 0}; //进入意图
int flag = 0;  // 当前是谁可以获得锁 (thread 0 or 1)

int self = i; //当前的线程ID, 0,  1中的一个

void lock() {
    intents[self] = 1;  // 标记自己想要进入临界区
    flag = self; //轮转到下一位
    while (  (intents[(self + 1)%2] == 1)  //别人想不想要进来
          && (flag == self) ); // 如果别人也想进来，就再看看是否当前“轮转”到自己了
}

void unlock() {
  intents[self] = 0; //标记自己不再想要进入临界区
}
```

在一个线程已经在等待进入的情况下，另一个线程无法连续进入两次

证明的繁琐

- Peterson算法已经有点复杂了，能否自动化证明?
  - 如果在原算法上稍微改一下（比如调整语句的顺序，删除一条语句），它还是正确的吗?
  - 当然，你还是可以手动再次证明，但有点低效了，对于这类问题，有更好的自动化证明方法：模型检验(model checking)!
  - Model checking is a method for formally verifying finite-state systems
    - 比如你可以用我们提供的mosaic

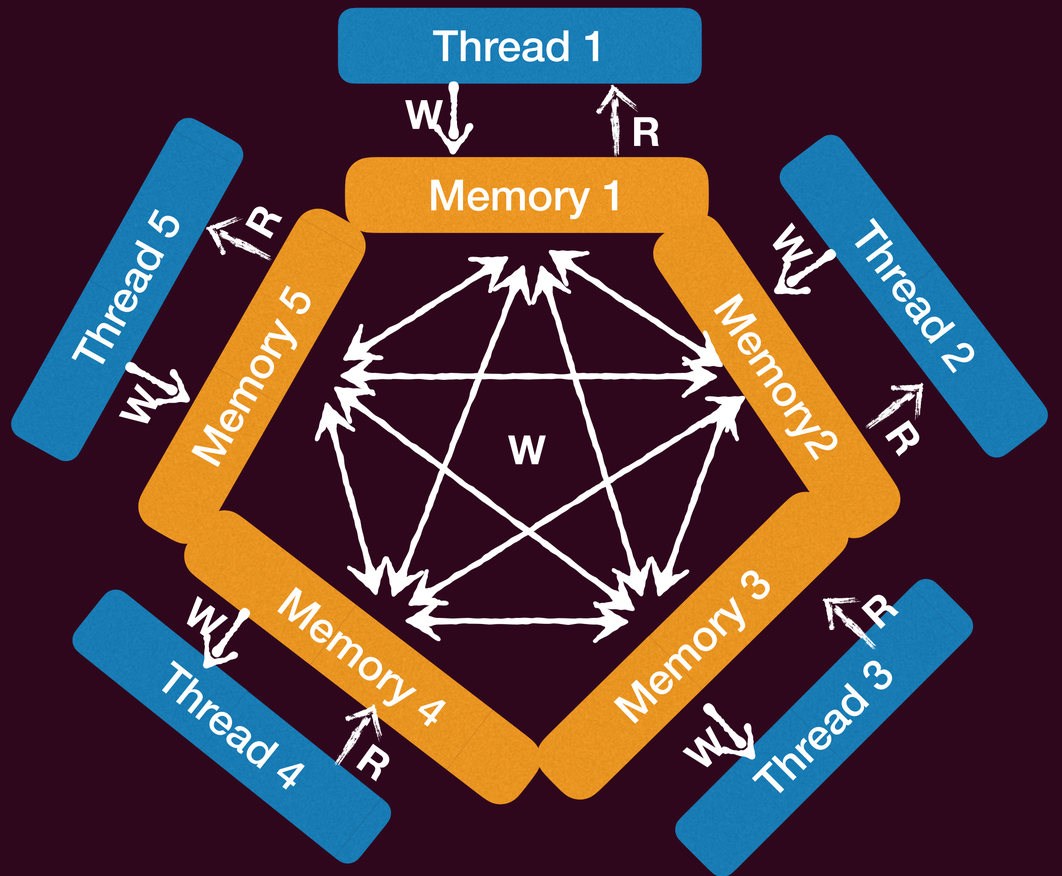
```
python3 mosaic.py -c peterson.py | grep \"cs\" | sort | uniq
```

*Model checking问题

- 两个线程的model checking的遍历空间已经很大了
- 100个线程呢?
- 不指定上限的线程数呢?
- 学术界已经很多关于model checking的研究都在试图挑战这些问题，使得这个工具更加实用，比如：
  - Model checking for programming languages using VeriSoft
  - Finding and reproducing Heisenbugs in concurrent programs
  - Using model checking to find serious file system errors
  - VSync: Push-button verification and optimization for synchronization primitives on weak memory models

回到现实计算机架构

- 现实的load和store都不是原子的，甚至两个线程同一刻读的flag都不一致！因此现实架构上的peterson算法其实是错误的



如何写出正确的peterson?

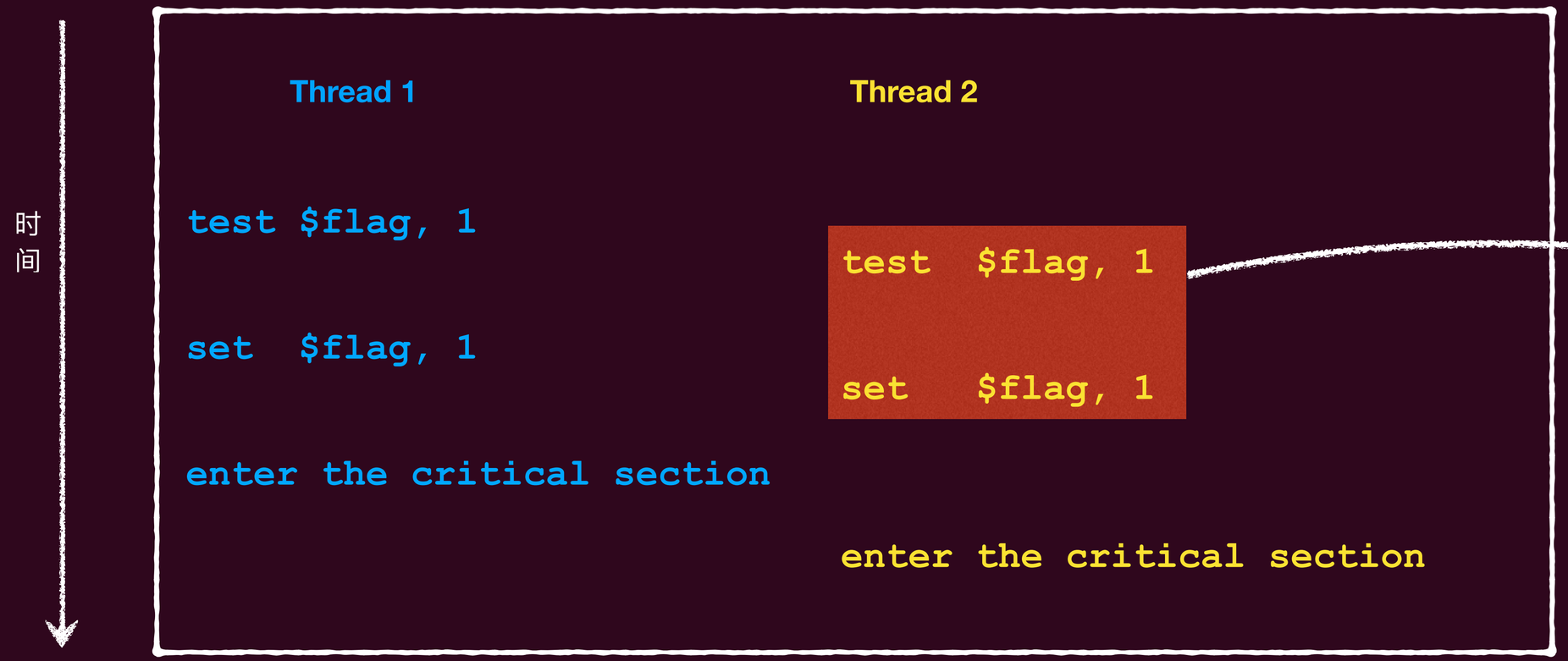
- 在现代计算机体系架构下，由于多处理器以及乱序指令流的存在，需要硬件的支持，比如：内存屏障 (Memory Barrier)
- gcc编译器可以将__sync_synchronize()编译为相应指令架构的内存屏障
  - x86: mfence (lock)
  - Arm: dmb ish
  - Mips: sync
  - Risc-v: fence rw, rw

但问题是：这些barrier就够了吗?

硬件支持的锁

回归初心

```
int flag = 0;

void lock(){
    while (flag == 1); //自旋的观测flag
    flag = 1; //设置
}

void unlock(){
    flag = 0;
}
```



如果这两个指令可以原子化就好了，软件方法如Peterson算法进行了大费周章的逻辑才实现了互斥，并且在现代计算机体系架构下还是错的，如果有硬件能够支持这两个操作的原子化，一切都变得简单！

原子的Test-And-Set (TAS) 指令

- 很多硬件架构都提供了原子指令可以来实现Test-And-Set Lock (TSL)，比如X86平台下，利用lock前缀加上cmpxchg：

```
cmpxchg src, dst:
if (dst == %ax) {
    dst = src;
    ZF = 1;
} else {
    %ax = dst
    ZF = 0;
}
```

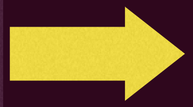
Compare-and-swap

```
int flag; // 需要被原子的test 和 set的flag

int expected = 0;

// expected = old value of flag, and, if flag != 0 then nothing else if flag == 0 then flag = 1
// equal to expected = __sync_val_compare_and_swap(&flag, 0, 1);
asm volatile (
        "lock cmpxchg %2, %1"
        : "+a" (expected) //Value for comparison. x86 uses eax/rax.
        : "m" (flag),     // Memory location
          "r" (1)         // Value(1) to be written if flag == expected
       : "memory", "cc"
 );
```

自旋锁(Spin Lock)的实现

- 有了TSA的硬件支持，我们的之前的锁可以实现为如下（x86）：

```
int flag = 0;

void lock(){
    while (flag == 1);
    flag = 1;
}

void unlock(){
    flag = 0;
}
```



```
int flag = 0;

void lock() {
int expected;
do {
        expected = 0;
        asm volatile (
           "lock cmpxchg %2, %1"
           : "+a" (expected)
           : "m" (flag),
             "r" (1)
           : "memory", "cc"
        );
    } while (expected != 0);
}
```

```
void unlock() {
  asm volatile (
       "mov %1, %0"
       : "=m" (flag)
       : "r" (0)
       : "memory"
    );
}
```

硬件原子操作

- 事实上，所有X86 CPU 都具有锁定一个特定内存地址的能力，当这个特定内存地址被锁定后，它就可以阻止其他的系统总线读取或修改这个内存地址
  - 通过Lock 指令前缀+具体指令，就可以在执行这个具体指令时“锁住”！

```
#这些具体指令都可以加Lock前缀，其中X开头的指令即使不加Lock前缀同样是原子的
 (according to IA-32 Intel® Architecture Software Developer’s Manual)

ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC,
NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG
```

*自旋锁的可以通过除(Compare-and-swap)其他形式实现

- 比如：Load linked/Store conditional (LL/SC): arm、mips、risc-v都提供了相应的支持
  - L returns the value of a memory location
  - A subsequent SC to that memory location succeeds only if that location has not been updated since LL

阅读材料

- [OSTEP] 第25、26、27章