

# 背景

---

## 并发进程的正确性

- 独立进程
  - 不和其他进程共享资源或状态
  - 确定性=>输入状态决定结果
  - 可重现=>能够重现起始条件
  - 调度顺序不重要
- 并发进程
  - 在多个进程间有资源共享
  - 不确定性
  - 不可重现
- 并发进程的正确性
  - 执行过程是不确定性和不可重现的
  - 程序错误可能是间歇性发生的

## 进程并发执行的好处

- 进程需要与计算机中的其他进程和设备进行协作
- 好处 1: 共享资源
  - 多个用户使用同一台计算机
  - 银行账号存款余额在多台 ATM 机操作
  - 机器人上的嵌入式系统协调手臂和手的动作
- 好处 2: 加速
  - I/O 操作和 CPU 计算可以重叠 (并行)
  - 程序可划分成多个模块放在多个处理器上并行执行
- 好处 3: 模块化
  - 将大程序分解成小程序
  - 以编译为例, gcc 会调用 cpp, cc1, cc2, as, ld
  - 使系统易于复用和扩展

## 并发创建新进程时的标识分配

- 程序可以调用函数 `fork()` 来创建一个新的进程
  - 操作系统需要分配一个新的并且唯一的进程 ID
  - 在内核中, 这个系统调用会运行

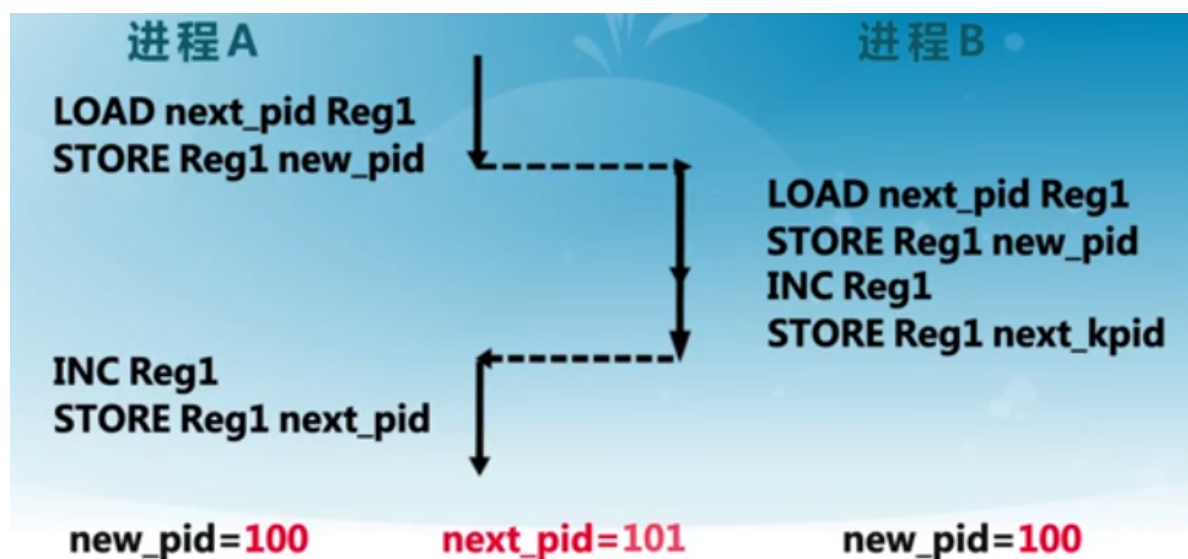
```
1 | new_pid = next_pid++;
```

- 翻译成机器指令

```
1 | LOAD next_pid Reg1
2 | STORE Reg1 new_pid
3 | INC Reg1
4 | STORE Reg1 next_pid
```

- 两个进程并发执行时的预期结果（假定 next\_pid = 100）
  - 一个进程得到的 ID 应该是 100
  - 另一个进程的 ID 应该是 101
  - next\_pid 应该增加到 102

## 新进程分配标识中的可能错误



会出现并发错误

该图例说明在进程 A 执行到 STORE 的时候，进程 B 抢占资源也开始执行代码

导致二者访问到同一个 `new_pid` 变量，从而获得了相同的进程 ID

那么这就会导致两个不同进程得到相同的进程 ID 的错误

## 原子操作 (Atomic Operation)

- 原子操作是指一次不存在任何中断或失败的操作
  - 要么操作成功执行
  - 或者操作没有执行
  - 不会出现部分执行的状态
- 操作系统需要利用同步机制在并发执行的同时，保证一些操作是原子操作

## 现实生活中的同步问题

- 操作系统和现实生活中的问题类比
  - 利用现实生活问题帮助理解操作系统同步问题
  - 同时注意，计算机与人的差异
- 例如：家庭采购协调

时 间	A	B
3:00	查看冰箱，没有面包了	
3:05	离开家去商店	
3:10	到达商店	查看冰箱，没有面包了
3:15	购买面包	离开家去商店
3:20	到家，把面包放进冰箱	到达商店
3:25		购买面包
3:30		到家，把面包放进冰箱

## 家庭采购协调问题分析

- 如何保证家庭采购协调的成功和高效
  - 有人去买
    - 需要采购时，有人去买面包
  - 最多只有一个人去买面包
- 可能的解决方法
  - 在冰箱上设置一个**锁和钥匙 (lock&key)**
  - 去买面包之前锁住冰箱并且拿走钥匙
- 加锁导致的新问题
  - 冰箱中还有其他食品时，别人无法取到

## 方案一

- 使用**便签**来避免购买太多面包
  - 购买之前留下一张便签
  - 买完后移除该便签
  - 别人看到便签时，就不去购买面包

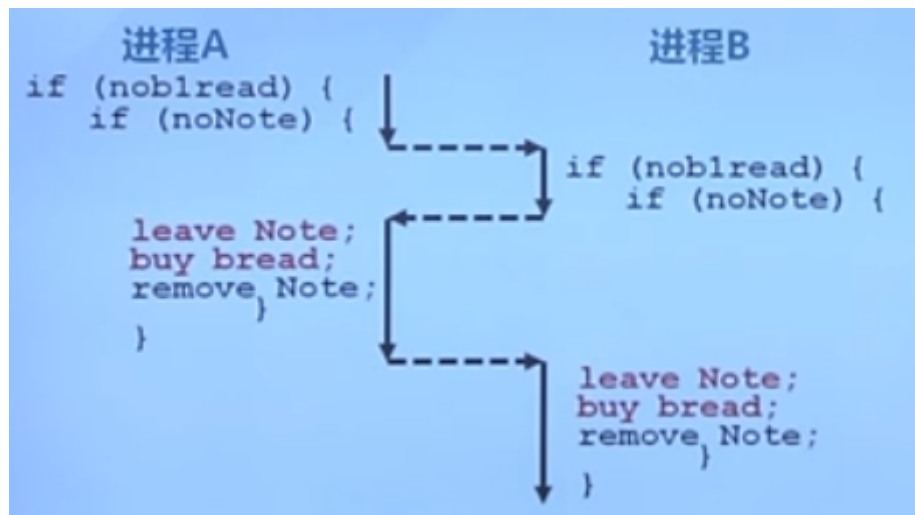
```

1  if (nobread) {
2      if (noNote) {
3          leave Note;
4          buy bread;
5          remove Note;
6      }
7  }
```

- 有效吗？

## 方案一分析

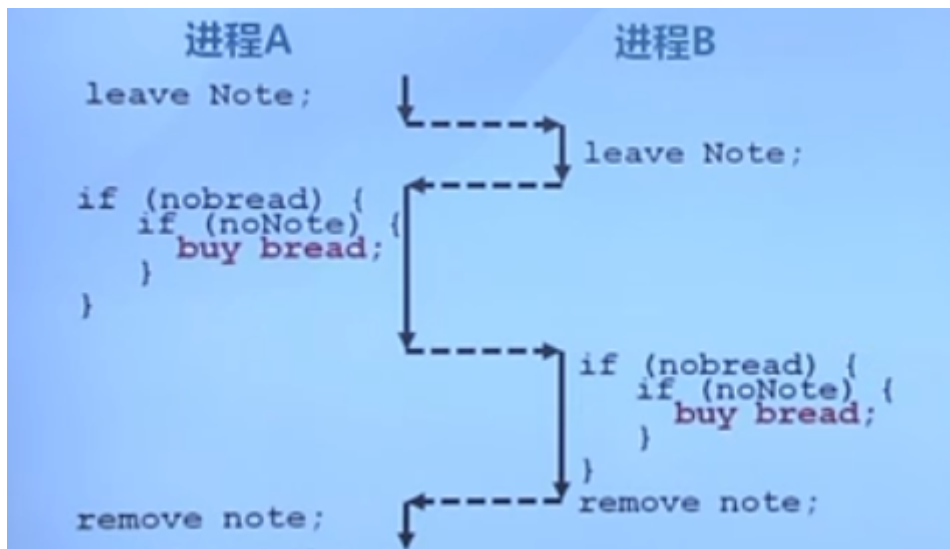
- 偶尔会购买太多面包
  - 检查面包和便签后贴便签前，有其他人检查面包和便签



- 解决方案只是间歇性地失败
  - 问题难以调试
  - 必须考虑调度器所做的事情

## 方案二

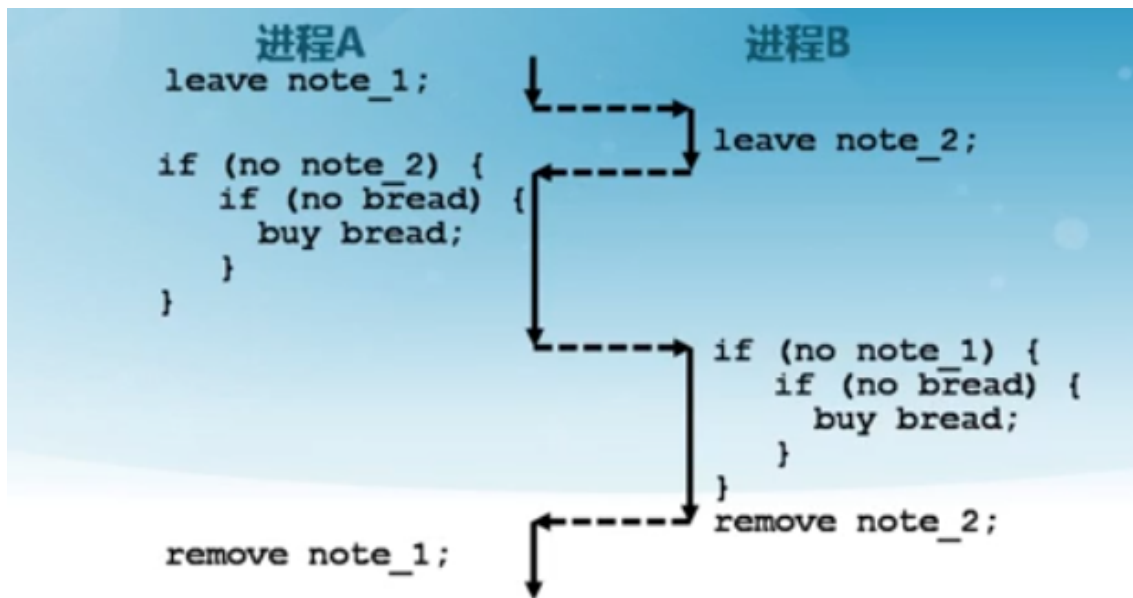
- 先留便签，后检查面包和便签



- 会发生什么？
  - 不会有人买面包

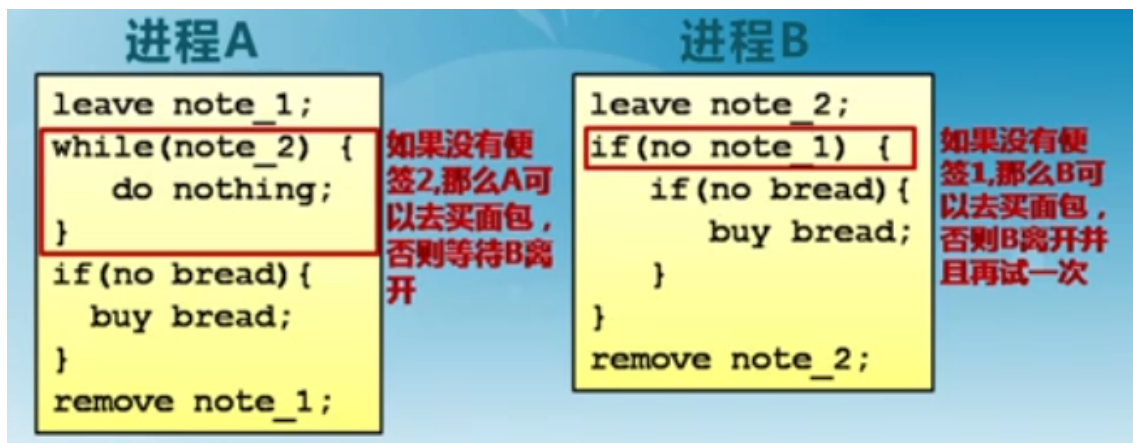
## 方案三

- 为便签增加标记，以区别不同人的便签
  - 现在可在检查之前留便签
- 会发生什么？
  - 可能导致没有人去买面包
    - 每个人都认为另外一个去买面包



## 方案四

- 两个人采用不同的处理流程



- 现在有效吗？
  - 枚举所有可能后，可以确认它是有效的
- 这种解决方案你满足？

## 方案四分析

- 它有效，但太复杂
  - 很难验证它的有效性
- A 和 B 的代码不同
  - 每个进程的代码也会略有不同
  - 如果进程更多，怎么办？
- 当 A 在等待时，它不能做其他事
  - 忙等待 (busy-waiting)
- 有更好的方法吗？

## 方案五

- 利用两个原子操作实现一个锁 (lock)
  - Lock.Acquire()
    - 在锁被释放前一直等待，然后获得锁
    - 如果两个线程都在等待同一个锁，并且同时发现锁被释放了，那么只有一个能够获得锁
  - Lock.Release()
    - 解锁并唤醒任何等待中的进程
- 基于原子锁的解决方法

```
1 breadlock.Acquire(); // 进入临界区
2 if (nobread) {
3     buy bread;        // 临界区
4 }
5 breadlock.Release(); // 退出临界区
```

## 进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥 (mutual exclusion)
  - 一个进程占用资源，其它进程不能使用
- 死锁 (deadlock)
  - 多个进程各占用部分资源，形成循环等待
- 饥饿 (starvation)
  - 其他进程可能轮流占用资源，一个进程一直得不到资源

## 练习题

- 临界资源是**互斥共享资源**
- 共享变量是指**可被多个进程访问**的变量

## 临界区和禁用硬件中断同步方法

# 临界区 (Critical Section)

```
1 entry section
2 critical section
3 exit section
4 remainder section
```

- 临界区 (critical section)
  - 进程中访问临界资源的一段需要互斥执行的代码
- 进入区 (entry section)
  - 检查可否进入临界区的一段代码
  - 如可进入, 设置相应“正在访问临界区”标志
- 退出区 (exit section)
  - 清除“正在访问临界区”标志
- 剩余区 (remainder section)
  - 代码中的其余部分

## 临界区的访问规则

- 空闲则入
  - 没有进程在临界区时, 任何进程可进入
- 忙则等待
  - 有进程在临界区时, 其他进程均不能进入临界区
- 有限等待
  - 等待进入临界区的进程不无限期待
- 让权等待 (可选)
  - 不能进入临界区的进程, 应释放 CPU (如转换到阻塞状态)

## 临界区的实现方法

- 禁用中断
- 软件方法
- 更高级的抽象方法
  - 原子操作指令
- 不同的临界区实现机制的比较
  - 性能: 并发级别

## 方法 1: 禁用硬件中断

- 没有中断, 没有上下文切换, 因此没有并发
  - 硬件将中断处理延迟到中断被启用之后
  - 现代计算机体系结构都提供指令来实现禁用中断

```
1 local_irq_save(unsigned long flags);
2 critical section
3 local_irq_restore(unsigned long flags);
```

- 进入临界区
  - 禁止所有中断，并保存标志
- 离开临界区
  - 使能所有中断，并恢复标志

## 缺点

- 禁用中断后，进程无法被停止
  - 整个系统都会为此停下来
  - 可能导致其他进程处于饥饿状态
- 临界区可能很长
  - 无法确定响应中断所需的时间（可能存在硬件影响）
- 要小心使用

## 练习题

- 要想进程互斥地进入各自的同类资源的临界区，需要在进程间互斥地使用临界资源

## 基于软件的同步方法

---

两个线程，T0 和 T1

线程 Ti 的代码

```
1  do {
2      enter section // 进入区
3      critical section
4      exit section // 退出区
5      reminder section
6  } while(1);
```

线程可通过共享一些共有变量来同步它们的行为

## 第一次尝试

- 共享变量

```
1  int turn = 0;
2  turn == i // 表示允许进入临界区的线程
```

- 线程 Ti 的代码

```
1  do {
2      while (turn != i) ;
3      critical section
4      turn = j;
5      reminder section
6  } while(1);
```

- 满足“忙则等待”，但是有时不满足“空闲则入”
  - Ti 不在临界区，Tj 想要继续运行，但是必须等待 Ti 进入过临界区后



## 第二次尝试

- 共享变量

```
1 int flag[2];
2 flag[0] = flag[1] = 0;
3 flag[i] == 1; // 表示线程 Ti 是否在临界区
```

- 线程 Ti 的代码

```
1 do {
2     while (flag[j] != 1) ;
3     flag[i] = 1;
4     critical section
5     flag[i] = 0;
6     remainder section
7 } while(1);
```

- 不满足“忙则等待”

## 第三次尝试

- 共享变量

```
1 int flag[2];
2 flag[0] = flag[1] = 0;
3 flag[i] == 1; // 表示线程 Ti 想要进入临界区
```

- 线程 Ti 的代码

```
1 do {
2     flag[i] = 1;
3     while (flag[j] == 1) ;
4     critical section
5     flag[i] = 0;
6     remainder section
7 } while(1);
```

- 满足“忙则等待”，但是不满足“空闲则入”

## Peterson 算法

- 满足线程 Ti 和 Tj 之间互斥的经典的基于软件的解决方法 (1981 年) turn
- 共享变量

```
1 int turn; // 表示该谁进入临界区
2 boolean flag[]; // 表示进程是否准备好进入临界区
```

- 进入区代码

```
1 flag[i] = true;
2 turn = j;
3 while (flag[j] && turn == j)
```

- 退出区代码

```
1 flag[i] = false;
```

线程  $T_i$  的代码

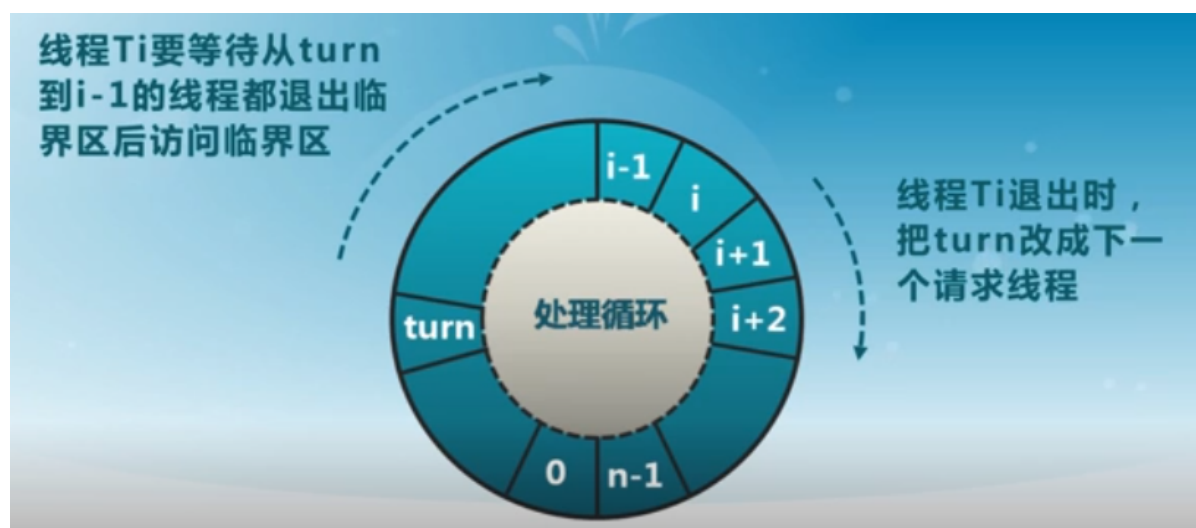
```
1 do {
2     flag[i] = true;
3     turn = j;
4     while (flag[j] && turn == j) ;
5     critical section
6     flag[i] = false;
7     reminder section
8 } while(true);
```

## Dekkers 算法

线程  $T_i$  的代码

```
1 flag[0] := false; flag[1] := false; turn := 0; // or 1
2 do {
3     flag[i] = true;
4     while flag[j] == true {
5         if turn ≠ i {
6             flag[i] := false
7             while turn ≠ i { }
8             flag[i] := true
9         }
10    }
11    critical section
12    turn := j
13    flag[i] = false;
14    reminder section
15 } while(true);
```

## N 线程的软件方法 (Eisenberg 和 McGuire)



- 复杂
  - 需要两个进程间的共享数据项

- 需要忙等待
  - 浪费 CPU 时间

## 高级抽象的同步方法

- 硬件提供了一些同步原语
  - 中断禁用，原子操作指令等
- 操作系统提供更高级的编程抽象来简化进程同步
  - 例如：锁、信号量
  - 用硬件原语来构建

## 锁 (lock)

- 锁是一个抽象的数据结构
  - 一个二进制变量（锁定/解锁）
  - Lock::Acquire()  
锁被释放前一直等待，然后得到锁
  - Lock::Release()  
释放锁，唤醒任何等待的进程
- 使用锁来控制临界区访问

```
1 lock_next_pid->Acquire();
2 new_pid = next_pid++;
3 lock_next_pid->Release();
```

## 原子操作指令

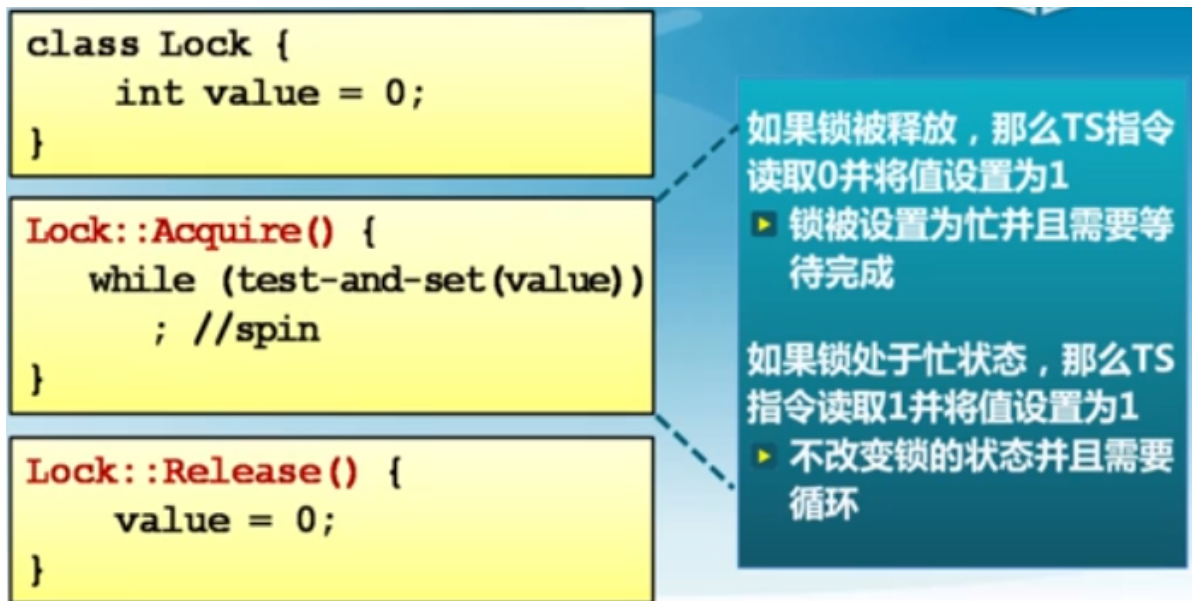
- 现代 CPU 体系结构都提供一些特殊的原子操作指令
- 测试和置位 (Test-and-set) 指令
  - 从内存单元中读取值
  - 测试该值是否为 1（然后返回真或假）
  - 内存单元值设置为 1

```
1 boolean TestAndSet (boolean *target) {
2     boolean rv = *target;
3     *target = true;
4     return rv;
5 }
```

- 交换指令 (exchange)
  - 交换内存中的两个值

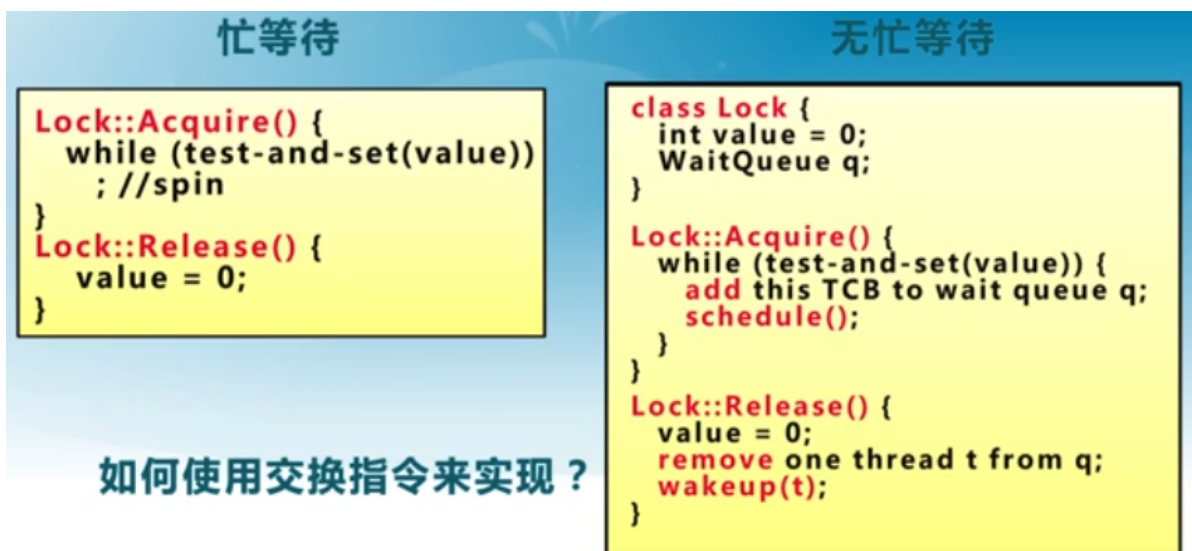
```
1 void Exchange (boolean *a, boolean *b) {
2     boolean temp = *a;
3     *a = *b;
4     *b = temp;
5 }
```

## 使用 TS 指令实现自旋锁 (spinlock)



- 线程在等待的时候消耗 CPU 时间

## 无忙等待锁



## 原子操作指令锁的特征

- 优点
  - 适用于单处理器或者共享主存的多处理器中任意数量的进程同步
  - 简单并且容易证明
  - 支持多临界区
- 缺点
  - 忙等待消耗处理器时间
  - 可能导致饥饿
    - 进程离开临界区时有多个等待进程的情况
  - 死锁
    - 拥有临界区的低优先级进程
    - 请求访问临界区的高优先级进程获得处理器并等待临界区

## 同步方法总结

- 锁是一种高级的同步抽象方法
  - 互斥可以使用锁来实现
  - 需要硬件支持
- 常用的三种同步实现方法
  - 禁用中断（仅限于单处理器）
  - 软件方法（复杂）
  - 原子操作指令（单处理器或多处理器均可）

## 练习题

- 一个进程由阻塞队列进入就绪队列，可能发生了一个**进程释放一种资源**的情况