



# 临界区与锁



中国科学院大学计算机学院  
2025-10-20



中国科学院计算技术研究所  
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES



# 回顾：多线程程序并发修改共享变量

- 一个程序的运行

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;
int loops;

void *worker(void *arg) {
    for (int i=0;i<loops;i++) {
        counter++;
    }
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    printf("Initial value: %d\n", counter);

    pthread_t p1, p2;
    pthread_create(&p1, NULL, worker,NULL);
    pthread_create(&p2, NULL, worker,NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("Final value: %d\n", counter);

    return 0;
}
```



# 回顾：多线程程序并发修改共享变量

- 并发访问控制

- 控制多个线程访问共享资源时的正确性
- 能否把线程访问变成串行？

```
./a.out 1000  
Initial value: 0  
Final value: 2000
```

```
./a.out 10000  
Initial value: 0  
Final value: 20000
```

```
./a.out 100000  
Initial value: 0  
Final value: 115664
```



# 并发访问控制

- 目标
  - 保障多线程/多进程正确地使用**共享资源 – 互斥访问**
  - 共享资源可以是
    - 一个变量
    - 一块缓冲区
    - 一个文件
    - 一个设备
    - .....



# 内容提要

---

- 线程/进程间并发控制
  - 临界区与原子操作
  - 同步机制 – 锁



# 一个并发控制的例子

- 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PID
- 例子：有两个进程同时运行（假定next\_pid = 100）
  - 进程A : PID = 100
  - 进程B : PID = 101
  - next\_pid = 102

```
if ((pid = fork()) == 0) {  
    /* child process */  
    exec("foo"); /* does not return */  
}  
else  
    /* parent */  
    wait(pid); /* wait for child to die */
```

在内核运行

new\_pid = next\_pid++

翻译成机器指令

```
LOAD next_pid Reg1  
STORE Reg1 new_pid  
INC Reg1  
STORE Reg1 next_pid
```

共享变量



# 分配PID过程出现错误

进程A

```
LOAD next_pid Reg1  
STORE Reg1 new_pid
```

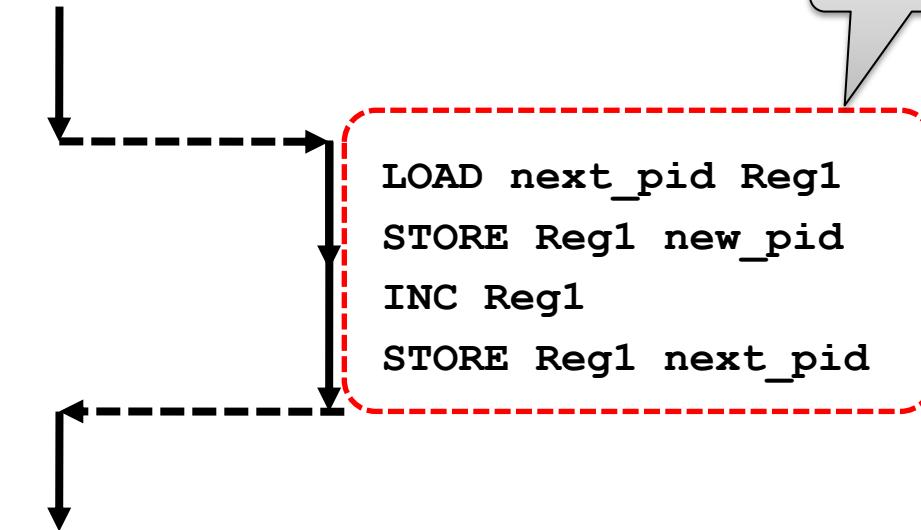
```
INC Reg1  
STORE Reg1 next_pid
```

**new\_pid=100**

进程B

```
LOAD next_pid Reg1  
STORE Reg1 new_pid  
INC Reg1  
STORE Reg1 next_pid
```

临界区



**new\_pid=100**



# 临界区 ( Critical Section )

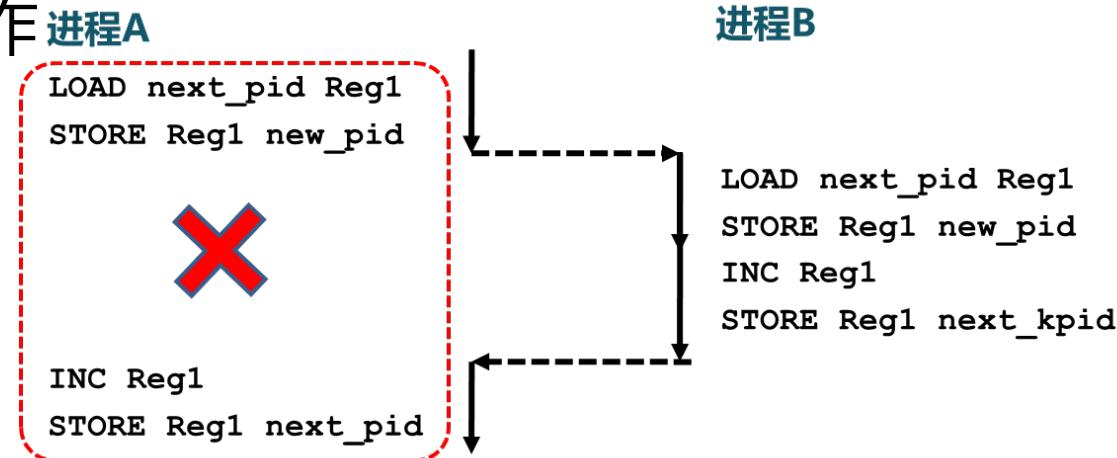
enter section  
critical section  
exit section

- 临界区 ( Critical Section )
  - 线程/进程中访问临界资源（共享资源）的一段需要互斥执行的代码
- 进入临界区
  - 检查可否进入临界区
  - 如可进入，设置相应"正在访问临界区"标志
- 退出临界区
  - 清除“正在访问临界区”标志



# 原子操作 ( Atomic Operation )

- 原子操作是指在执行过程中不存在任何中断的一个或一系列操作
  - 要么操作成功完成
  - 或者操作没有执行
  - 不会出现部分执行的状态
- 对临界区的操作必须是原子操作，**进入临界区的操作也需要是原子操作**



- 操作系统设计了**同步机制**在线程/进程并发访问共享资源时，保证临界区是原子操作



# 内容提要

---

- 线程/进程间并发控制
  - 临界区与原子操作
  - 同步机制 – 锁



# 同步机制

- 基本概念
  - 用于协调多线程/多进程对共享资源进行并发访问的方法，以避免竞态条件和数据不一致的问题
- 竞态条件
  - 在多线程/多进程场景中，由于线程/进程的执行顺序不确定，导致程序结果依赖于线程/进程的调度顺序，从而出现不可预测的结果
  - 使用同步机制来解决竞态条件带来的问题



# 一个例子

- 协调采购

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



# 方案一

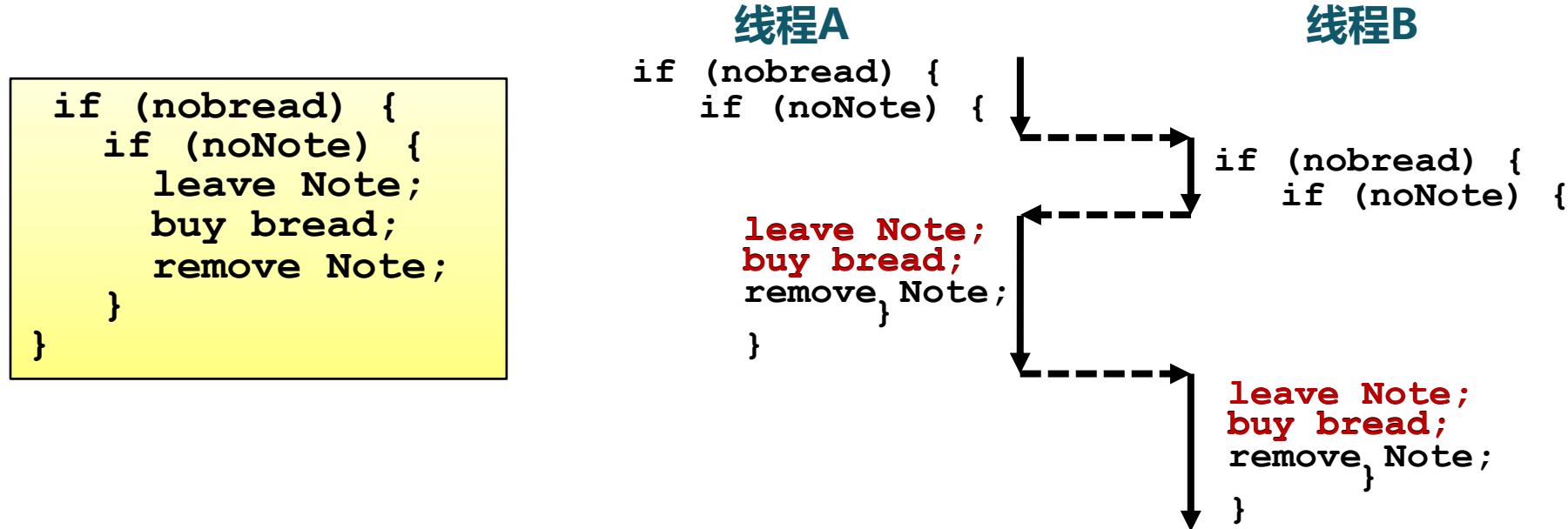
- 方案描述
  - 在冰箱上设置一个锁和钥匙
  - 去买面包之前锁住冰箱并且拿走钥匙
- 临界区：
  - 进入临界区：锁住冰箱，拿走钥匙
  - 临界区操作：检查有无面包;若无，则购买面包，放入冰箱
  - 退出临界区：解锁冰箱，放回钥匙
  - 能实现原子操作？
- 缺点
  - 进入临界区时锁住的资源太大：冰箱中还有其他食品时，别人无法取到
  - 冰箱锁了，忘拿钥匙。 . 。



# 方案二

- 方案描述

- 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





# 方案二分析

- 基本思路：A与B通过**便签**约定使用冰箱
- 临界区
  - 进入临界区：查看别人是否留便签，若没有便签，则留便签
  - 临界区操作：检查有无面包;若无,则购买面包，放入冰箱
  - 退出临界区：拿走便签
  - 能实现原子操作么？
- 问题
  - 检查面包和便签后，帖便签前，有其他人检查面包和便签，导致购买太多面包

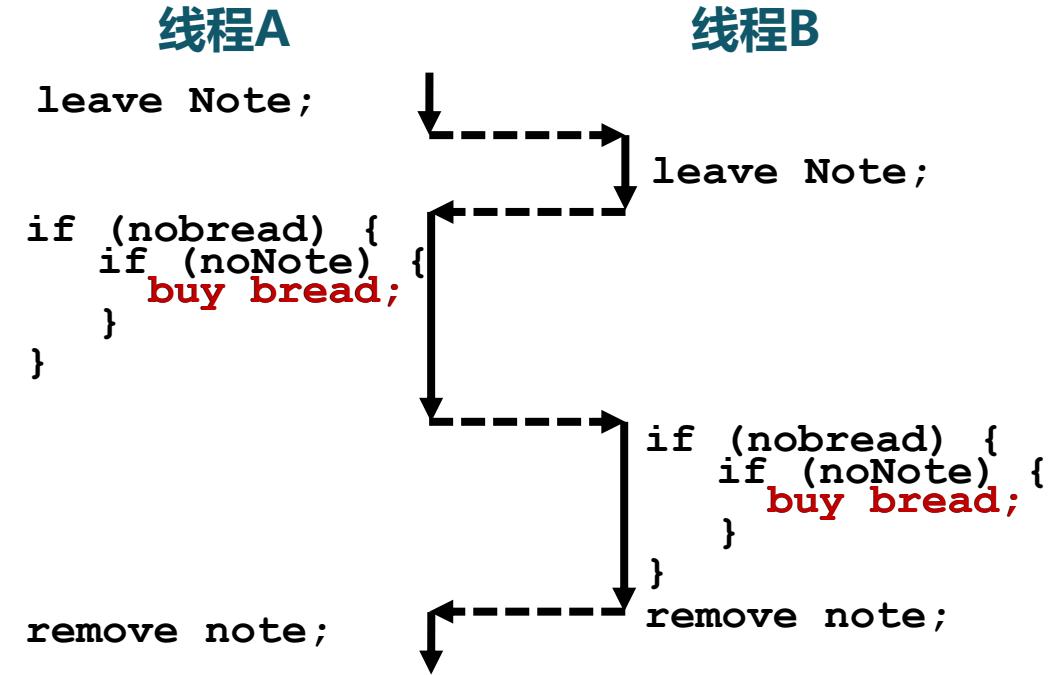
```
if (nobread) {  
    if (noNote) {  
        leave Note;  
        buy bread;  
        remove Note;  
    }  
}
```



# 方案三

- 方案描述
  - 先留便签，后检查面包和其他人留的便签
- 问题
  - 可能没有人买面包

```
leave Note;  
if (nobread) {  
    if (noNote) {  
        buy bread;  
    }  
}  
remove note;
```





# 方案四

- 方案描述
  - 两个人采用不同的处理流程
  - 正确！
- 问题
  - A和B的代码不同，扩展性差
  - A处于忙等状态

必须假设A先启动

线程A

```
leave note_1;  
while(note_2) {  
    do nothing;  
}  
if(no bread) {  
    buy bread;  
}  
remove note_1;
```

如果没有便签2,那么A可以去买面包,否则等待B离开

线程B

```
leave note_2;  
if(no note_1) {  
    if(no bread) {  
        buy bread;  
    }  
}  
remove note_2;
```

如果没有便签1,那么B可以去买面包,否则B离开并且移除便签2



# 方案小结

- 问题抽象
  - 多线程/进程并发访问共享资源，如何保证正确性？
- 方案要求
  - 进入临界区必须是原子操作
  - 不同线程/进程的代码一致
- 锁
  - Lock.Acquire()
    - 若没有线程/进程拿锁，则可以获得锁；否则一直等待
    - 多个线程/进程等待同一个锁时，当锁释放后，只有一个能够获得锁
  - Lock.Release()
    - 解锁并唤醒等待中的线程/进程（状态：阻塞 → 就绪）

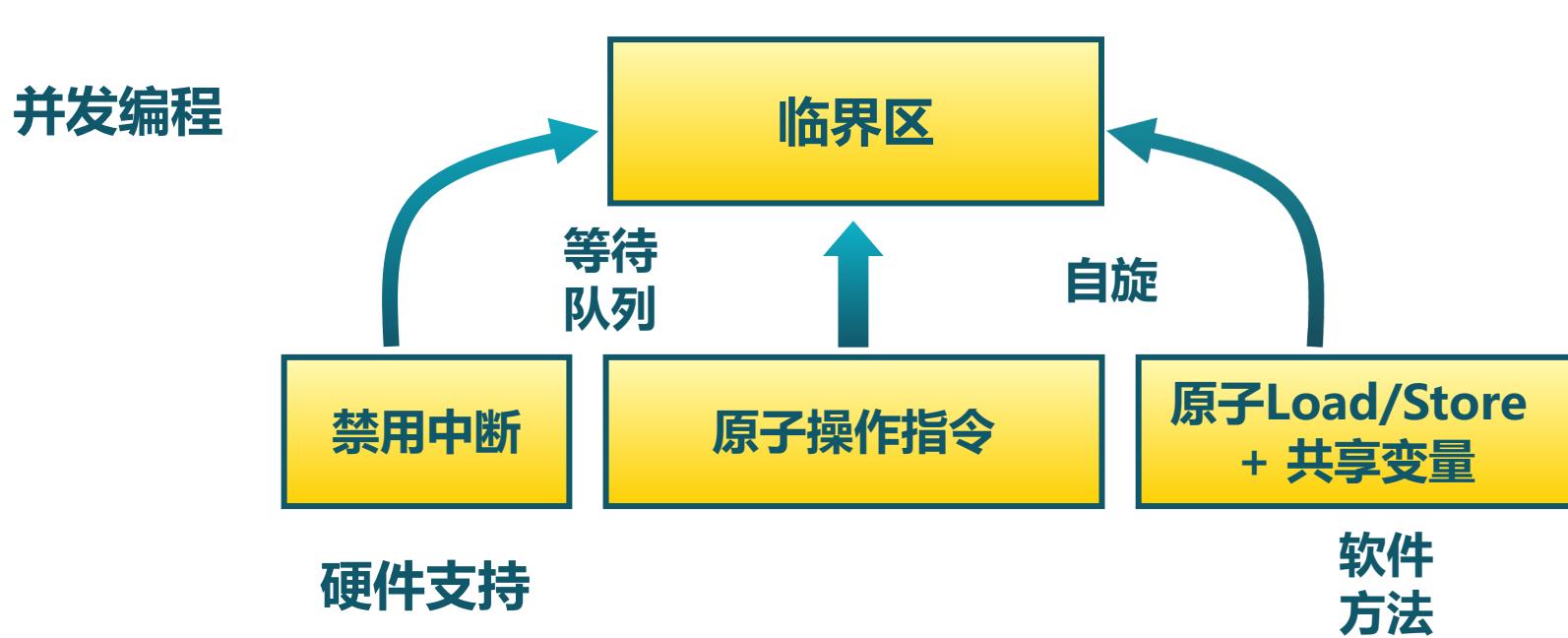
```
breadlock.Acquire();    进入临界区
if (nobread) {
    buy bread;          临界区
}
breadlock.Release();    退出临界区
```

- 注意：锁自身的操作必须是原子操作！



# 锁的实现方法

- 基于软件的方法
- 原子操作指令
- 禁用硬件中断





# 基于软件的方法

- 两个线程，T0和T1
- 线程可通过共享一些共有变量来协调它们的行为

```
do {  
    enter section 进入临界区  
    critical section  
    exit section 退出临界区  
    remainder section  
} while (1);
```



# 方案一

- 共享变量

```
int turn = 0;  
turn == i // 表示允许线程Ti进入临界区
```

- 线程Ti的代码

```
do {  
    while (turn != i) ;  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

- 满足“忙则等待”，但是不满足“空闲则入”

- T1执行，while判断条件成立，进入等待；但T0一直没有被调度执行，导致T1也无法进入临界区执行。而此时，临界区并没有被访问。

**忙则等待**：当线程无法获取锁时，则等待，直到锁可用

**空闲则入**：当锁是空闲时，则线程可以立即获得锁进入临界区



## 方案二

- 共享变量 test & set之间不能被打断

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

- 线程Ti的代码

```
do {  
    while (flag[j] == 1) ;  
    flag[i] = 1;  
    critical section  
    flag[i] = 0;  
    remainder section  
} while (1);
```

while循环结束 ( test ) 后会被打断

- 不满足“忙则等待”，正确性有问题



# 方案三

- 共享变量

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

- 线程Ti的代码

```
do {
    flag[i] = 1;
    while (flag[j] == 1) ;
    critical section
    flag[i] = 0;
    remainder section
} while (1);
```

两个线程同时置  
flag，但没有线程  
通过test

- 满足“忙则等待”，但是不满足“空闲则入”



# Dekker's 算法

## 线程Ti 的代码

flag : 是否想要进入临界区 ; turn : 哪个线程有权限进入

```
flag[0] := false; flag[1] := false; turn := 0;//or1  
  
do {  
    flag[i] = true;  
    while flag[j] == true {  
        if turn != i {  
            flag[i] := false  
            while turn != i { }  
            flag[i] := true  
        }  
    }  
    CRITICAL SECTION  
    turn := j  
    flag[i] = false;  
    REMAINDER SECTION  
} while (true);
```



# Peterson算法

- 满足线程Ti和Tj之间互斥的经典的软件解决方法  
( 1981年 )

- 共享变量

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

- 进入临界区代码

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

- 退出临界区代码

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



# Peterson算法实现

## 线程Ti 的代码

```
do {  
    flag[i] = true;  
    turn = j;  
    while ( flag[j] && turn == j);  
  
        CRITICAL SECTION  
  
        flag[i] = false;  
  
        REMAINDER SECTION  
  
    } while (true);
```



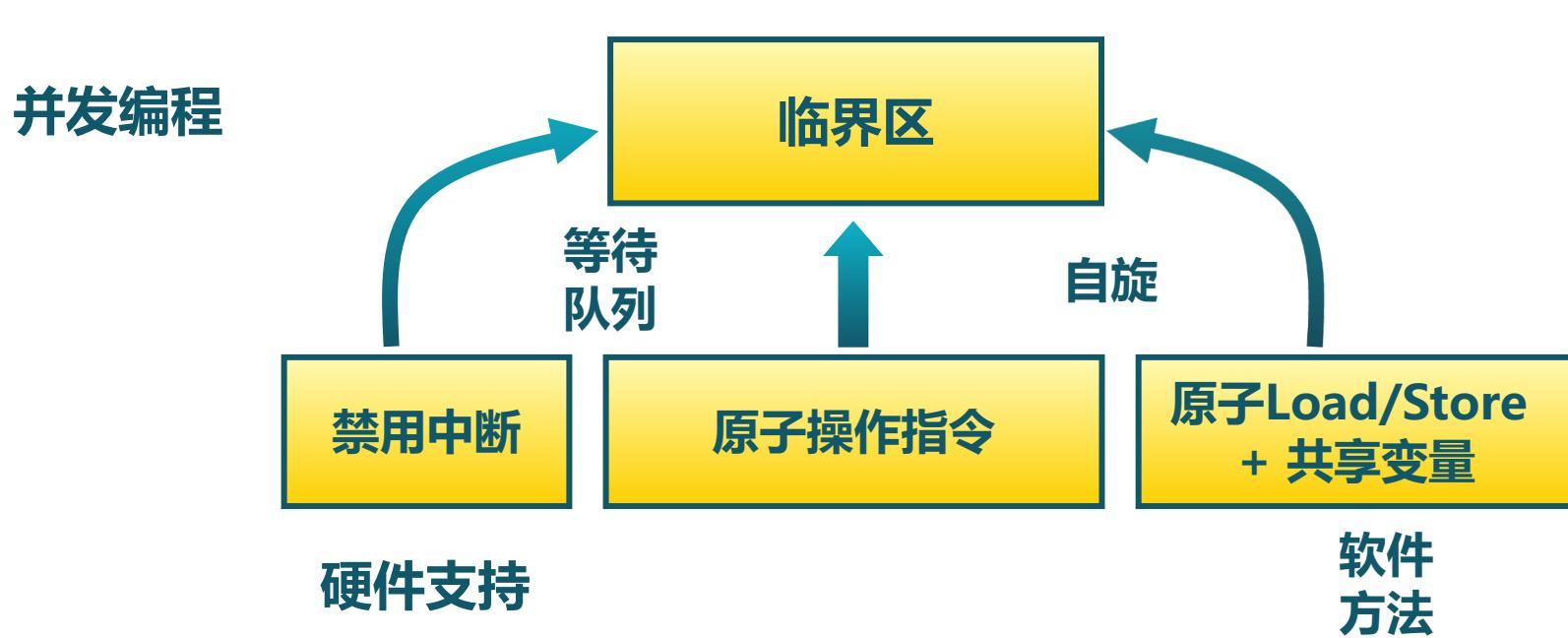
# 软件方法的分析

- 复杂
  - 需要算法和共享变量来保证锁本身操作（进入临界区）的原子性
    - 交替使用多个标记位（flag和turn），巧妙安排标记位的检查顺序，保证互斥
- 需要“忙等待”
  - 浪费CPU时间
- 适用于单核场景



# 锁的实现方法

- 基于软件的方法
- 原子操作指令
- 禁用硬件中断





# 原子操作指令

- 现代CPU都提供一些特殊的原子操作指令
- 测试和置位 ( Test-and-Set , TAS/TS ) 指令
  - 从内存单元中读取值
  - 测试该值是否为1或0，然后返回真或假（返回内存单元原值）
  - 内存单元值设置为1

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```



# 使用TAS指令实现锁

- 示例

```
class Lock {  
    int value = 0;  
}
```

```
Lock::Acquire () {  
    while (test-and-set(value))  
        ; //spin  
}
```

```
Lock::Release () {  
    value = 0;  
}
```

如果锁未被占用，那么TAS指令读取0并将值设置为1

► 锁被设置为忙，跳出循环

如果锁被占用，那么TAS指令读取1并将值设置为1

► 不改变锁的状态并且需要循环

► 线程在等待的时候消耗CPU时间



# 原子操作指令

- 交换指令 ( exchange )
  - 交换\*a和\*b的值，返回\*a原值

```
exchange (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

```
lock = UNLOCKED //UNLOCK is 0, LOCKED is 1
```

```
Lock::Acquire() {
    while (exchange(lock, LOCKED))
        ; //spin
}
Lock::Release() {
    lock = UNLOCKED;
}
```



# 原子操作指令

- Load linked 和 Conditional store (LL-SC)
  - 在一条指令中读一个值(Load linked)
  - 做一些操作
  - Store 时，检查 load linked 之后，值是否被修改过。  
如果没有，则成功修改；否则，从头再来
- Fetch-and-Add 或 Fetch-and-Op
  - 用于大型共享内存多处理器系统的原子指令



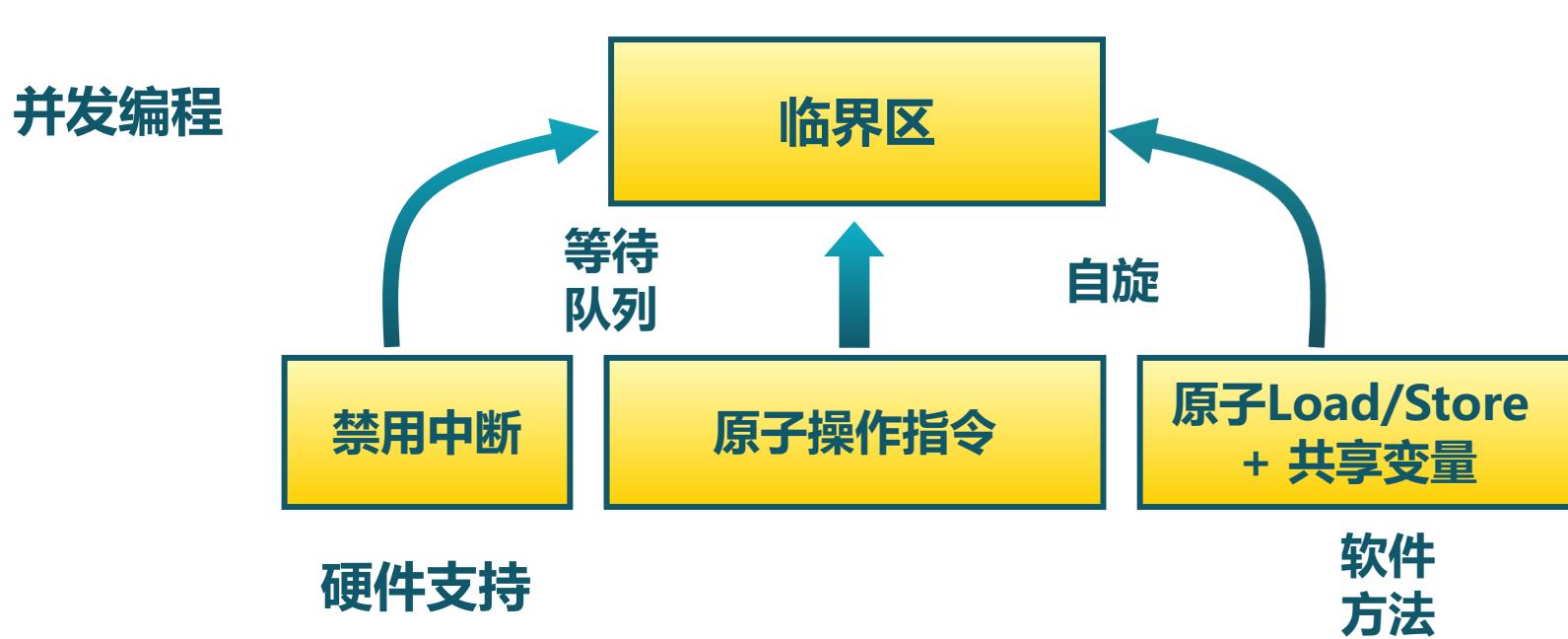
# 原子操作指令

- 优点
  - 单核、多核场景都适用
    - 支持内存总线锁，可以在加锁时，防止其他核对锁变量的访问
    - 利用多核缓存一致性，确保修改后的锁变量值在其他处理核中失效
  - 实现简单，性能好



# 锁的实现方法

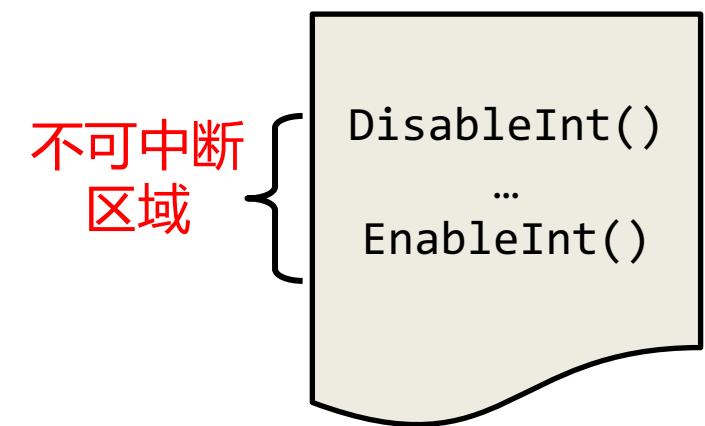
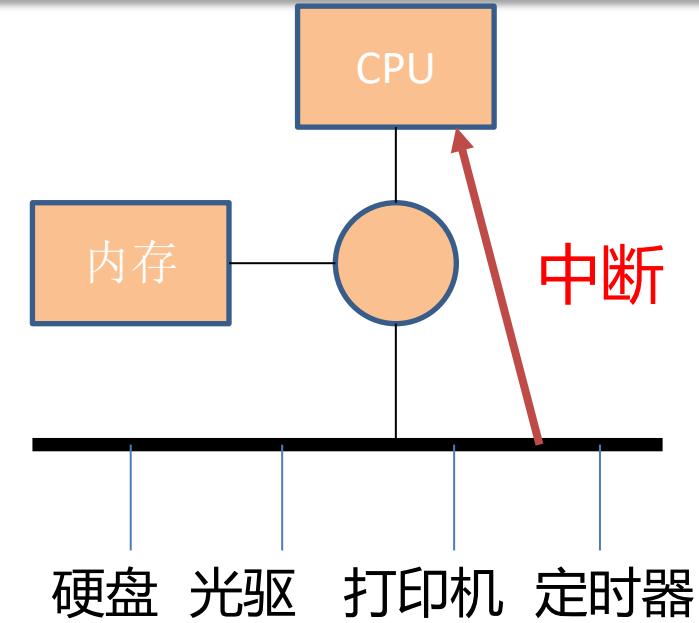
- 基于软件的方法
- 原子操作指令
- 禁用硬件中断





# 禁用中断实现互斥

- 基于中断
  - 可以实现抢占式 CPU 调度，并进行线程/进程切换
- 通过在 acquire 和 release 之间禁止上下文切换来提供互斥
- 禁用中断以屏蔽外部事件
  - 引入不可中断的代码区域
  - 实现串行执行临界区
  - **延迟**处理外部事件





# 禁用中断的简单方法

```
Acquire()  
{  
    disable interrupts;  
}
```

```
Release()  
{  
    enable interrupts;  
}
```

Acquire()

关键区？

Release()

- 有什么问题吗？
  - 锁整个冰箱

多核不适用，  
临界区设计不好  
时效率低



# 再次尝试

- 使用锁变量

```
Acquire(lock)
{
    disable interrupts;
    while (lock.value != FREE);
    lock.value = BUSY;
    enable interrupts;
}
```

```
Release(lock)
{
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}
```

- 有什么问题吗？
  - 忙等阶段无法响应中断，单核场景下导致一直忙等



# 再次尝试

- 使用锁变量，并且只在对锁变量进行测试和赋值时通过中断实现互斥

```
Acquire(lock)
{
    disable interrupts;
    while (lock.value != FREE) {
        enable interrupts;
        disable interrupts;
    }
    lock.value = BUSY;
    enable interrupts;
}
```

```
Release(lock)
{
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}
```

- 仍然有概率导致“一直忙等”



# 再次尝试，并引入队列.....

```
Acquire(lock)
```

```
{  
    disable interrupts;  
    while (lock.value == BUSY) {  
        add TCB to wait queue q;  
        Yield();  
    }  
    lock.value = BUSY;  
    enable interrupts;  
}
```

```
Release(lock)
```

```
{  
    disable interrupts;  
    if (q is not empty) {  
        remove thread t from q  
        Wakeup(t);  
    }  
    lock.value = FREE;  
    enable interrupts;  
}
```

- 不再忙等，进入wait queue
- 通过yield放弃CPU控制权
- 何时重新启用中断？



# 再次尝试，并引入队列.....

```
Acquire(lock)
```

```
{  
    disable interrupts;  
    while (lock.value == BUSY) {  
        enable interrupts;  
        add TCB to wait queue q;  
        Yield();  
        disable interrupts;  
    }  
    lock.value = BUSY;  
    enable interrupts;  
}
```

```
Release(lock)
```

```
{  
    disable interrupts;  
    if (q is not empty) {  
        remove thread t from q  
        Wakeup(t);  
    }  
    lock.value = FREE;  
    enable interrupts;  
}
```

在该线程被阻塞前另一个线程释放锁，会导致该线程无法被唤醒

- 进入wait queue前启用中断，问题？
  - TCB入wait queue，谁来唤醒 → 唤醒丢失问题



# 再次尝试，并引入队列.....

```
Acquire(lock)
```

```
{  
    disable interrupts;  
    while (lock.value == BUSY) {  
        add TCB to wait queue q;  
        enable interrupts;  
        Yield();  
        disable interrupts;  
    }  
    lock.value = BUSY;  
    enable interrupts;  
}
```

```
Release(lock)
```

```
{  
    disable interrupts;  
    if (q is not empty) {  
        remove thread t from q  
        Wakeup(t);  
    }  
    lock.value = FREE;  
    enable interrupts;  
}
```

- yield前启用中断



# 基于中断实现锁

- 执行临界区时禁止中断
  - 临界区可能很长，系统将长时间无法响应中断，可能导致外部中断事件丢失
- 进入临界区时禁用中断
  - 细粒度控制中断的禁用和启用
- 主要适用于单核系统
  - 多核系统中其他核上的线程仍然可以访问共享资源
  - 检查并设置操作使用原子指令操作

有了原子指令，是否不用关中断的方法了？



# 自旋锁和互斥锁

## 自旋锁：忙等

```
Lock::Acquire() {
    while (test-and-set(value))
        ; //spin
}
Lock::Release() {
    value = 0;
}
```

## 互斥锁：进阻塞队列

```
class Lock {
    int value = 0;
    WaitQueue q;
}

Lock::Acquire() {
    while (test-and-set(value)) {
        add this TCB to wait queue q;
        schedule();
    }
}

Lock::Release() {
    value = 0;
    remove one thread t from q;
    wakeup(t);
}
```

### 自旋锁

- 临界区很小，快速释放
- 通常用于内核，并关中断

### 互斥锁

- 临界区可大可小
- 用户态请求锁，获取不到时阻塞（通过系统调用进内核）



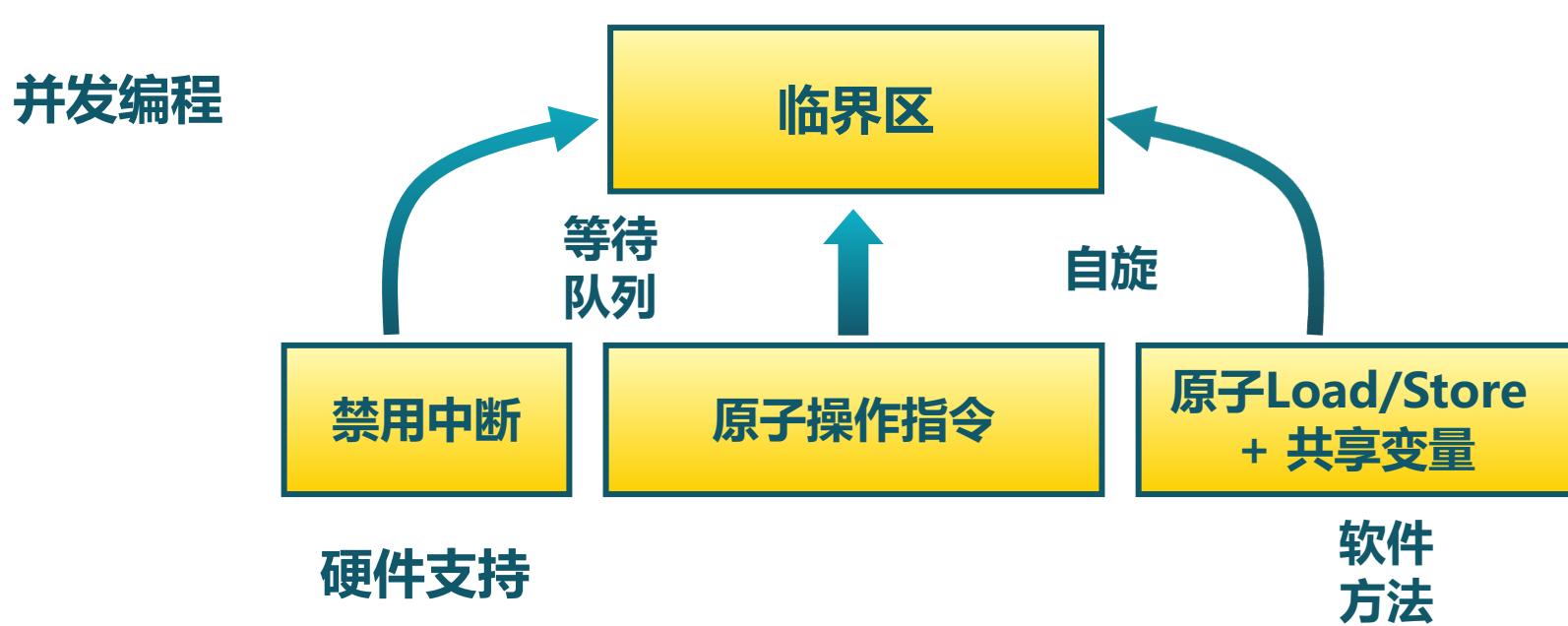
# 扩展了解

- `pthread_mutex_lock`/`pthread_mutex_unlock`
- `futex`系统调用



# 总结

- 进程/线程并发控制
  - 临界区
  - 进入临界区的操作须为原子操作





# 总结

- 基于软件方法实现锁
  - 实现复杂，依赖并发控制算法
  - Dekker和Peterson算法
- 基于原子操作指令实现锁
  - 支持多核场景
- 基于关中断实现锁
  - 粒度过大会导致系统无法及时响应中断
  - 细粒度控制中断启停
  - 用于单核场景
- 自旋锁与互斥锁