中国科学院大学
University of Chinese Academy of Sciences

操作系统课程 **B0911010Y-01**

进程/线程间通信

中国科学院大学计算机与控制学院
中国科学院计算技术研究所
2021-09-27

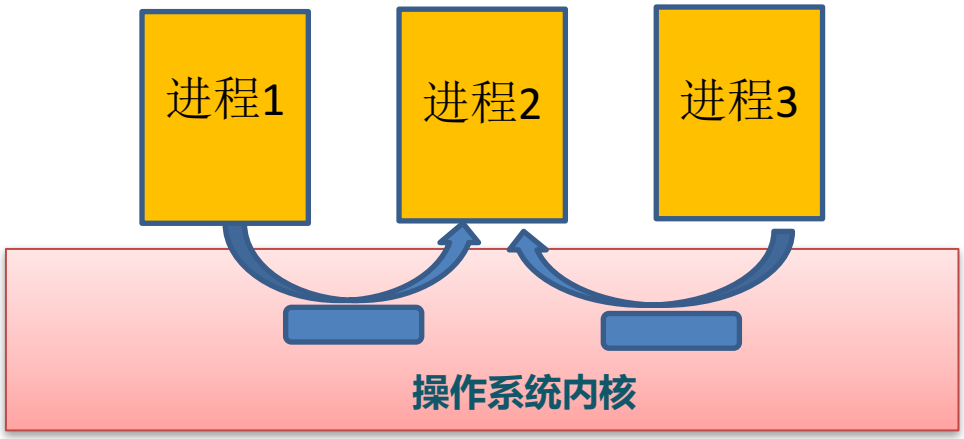
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

内容提要

- 进程/线程通信的基本概念
- 进程/线程间同步
  - 临界区与原子操作
  - 同步机制

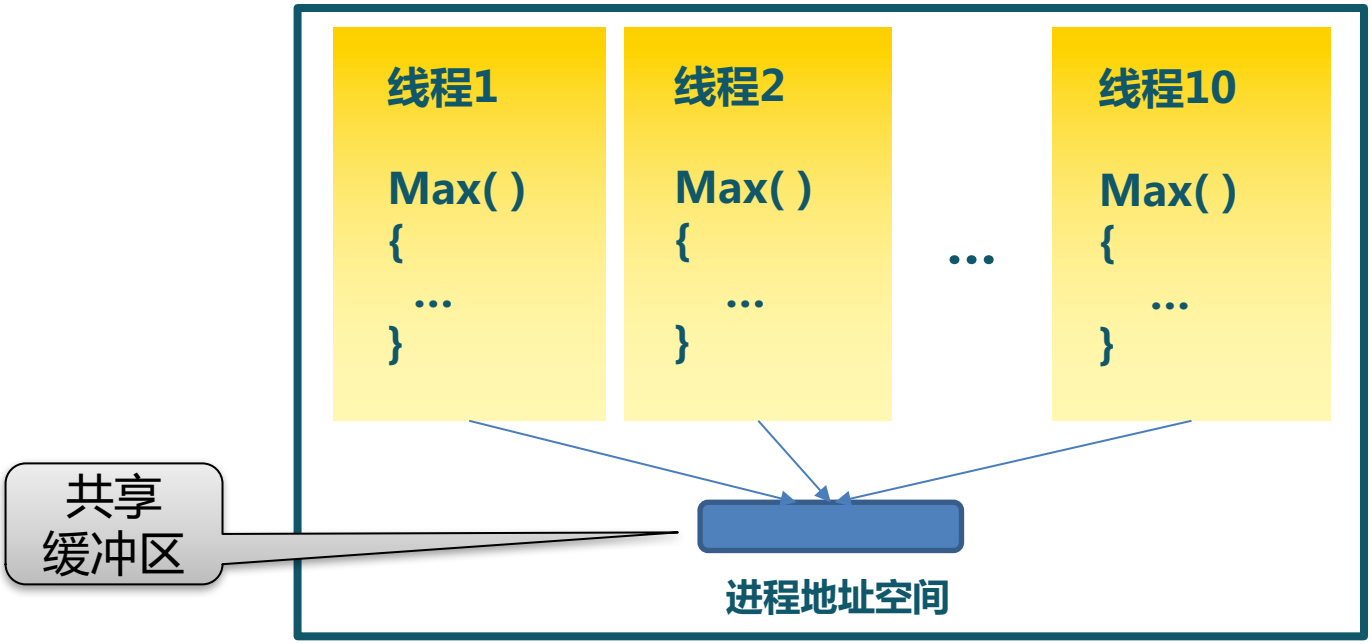
多进程间的通信

- 假设有10个文件，每个文件包含1000万数据，现要统计启动这1个亿数据中的最大值
- 启动10个进程，每个进程负责统计1个文件中的最大值，然后再汇总统计全局最大值
- 问题1：多个进程之间如何传递数据？
  - 文件，Socket，共享内存，管道
  - 多进程之间通信需要通过操作系统内核
- 问题2：一个进程如何通知另一个进程已完成统计？



多线程间的通信

- 多线程实现1亿数据最大值统计
- 问题1：多个线程之间如何传递数据？
  - 同一个地址空间，可直接共享缓冲区
- 问题2：如何保证共享资源的正确访问？例如写在共享缓冲区哪个位置？



回顾：操作系统能做什么？

- 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：操作系统能做什么？

- 并发访问管理
  - 控制多个程序访问相同资源时的正确性
    - 能否把程序访问变成串行？
    - 如何保证程序操作的原子性？
    - 如何让其他程序知道当前程序操作已完成？

```
./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
```

通信的两大作用

- 并发进程/线程之间需要进行数据传输与信息同步
- 数据传输
  - 将单个任务切分成多个子任务并发执行，提高并发度
    - 最大值统计：每个子任务负责统计1个文件中的最大值，最后再汇总统计
  - 数据在不同任务间传输
- 信息同步（互斥访问）
  - 保障多进程/多线程正确地使用共享资源
  - 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

- 进程/线程通信的基本概念
- 进程/线程间同步
  - 临界区与原子操作
  - 同步机制

一个同步的例子

- 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

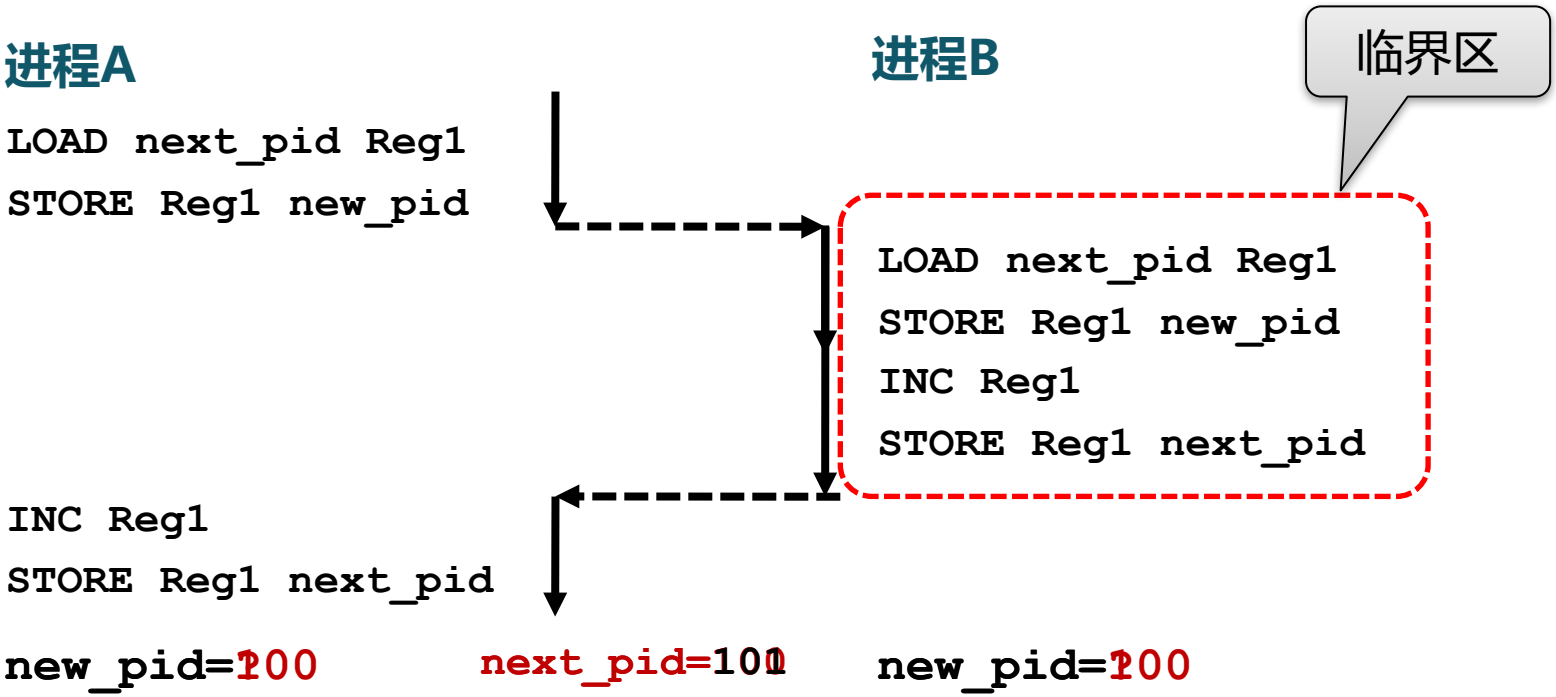
```
new_pid = next_pid++
```

共享变量

翻译成机器指令

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

分配PID过程出现错误



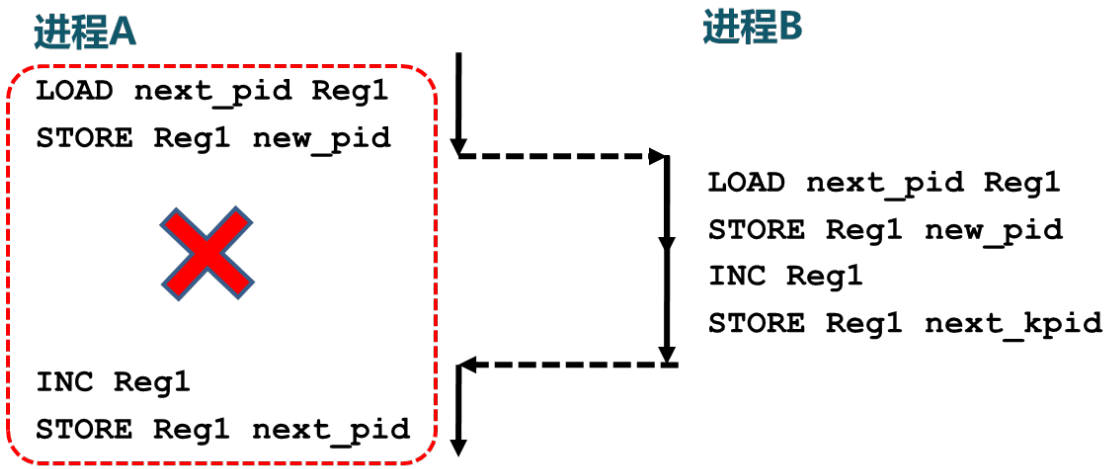
临界区（Critical Section）

```
entry section
    critical section
exit section
```

- 临界区（Cirtical Section）
  - 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的一段代码
  - 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）

- 原子操作是指一次不存在任何中断或失败的操作
  - 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



- 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

内容提要

- 进程/线程通信的基本概念
- 进程/线程间同步
  - 临界区与原子操作
  - 同步机制

同步机制设计三步走

- 第一步：识别出共享资源与使用者
- 第二步：设计合适的同步机制
- 第三步：验证临界区是否符合原子操作

- 一个生活中的例子：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

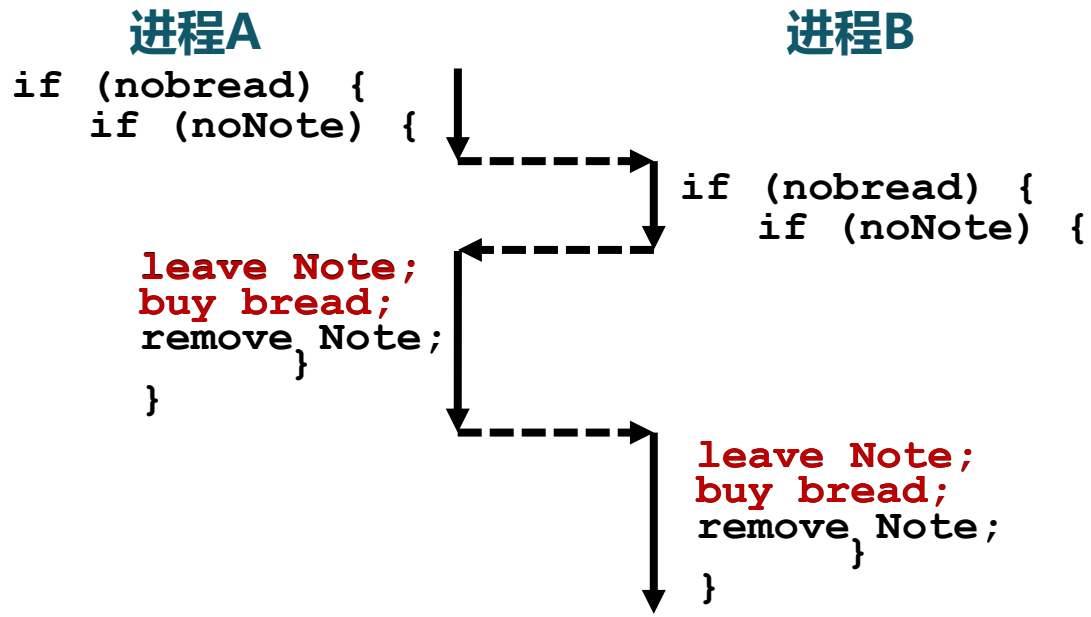
方案一

- 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
- 共享资源与使用者：A与B都能使用冰箱
- 临界区：
  - 进入临界区：锁住冰箱拿走钥匙
  - 临界区操作：去商店买面包
  - 退出临界区：打开冰箱，放入面包，放回钥匙
  - 原子操作？
- 缺点
  - 临界区太大：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

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



方案二分析

- 共享资源与使用者：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：购买面包
  - 退出临界区：拿走便签
  - 原子操作？

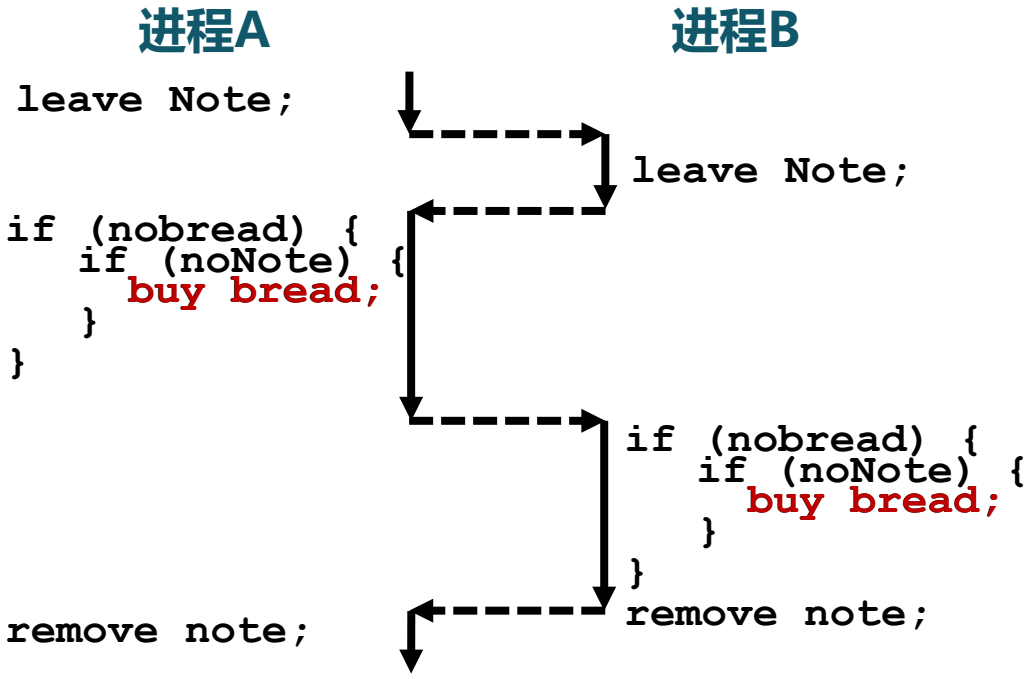
```
if (nobread) {
   if (noNote) {
      leave Note;
      buy bread;
      remove Note;
   }
}
```

- 问题
  - 可能会购买太多面包
  - 检查面包和便签后帖便签前，有其他人检查面包和便签

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

进程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

进程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

方案五

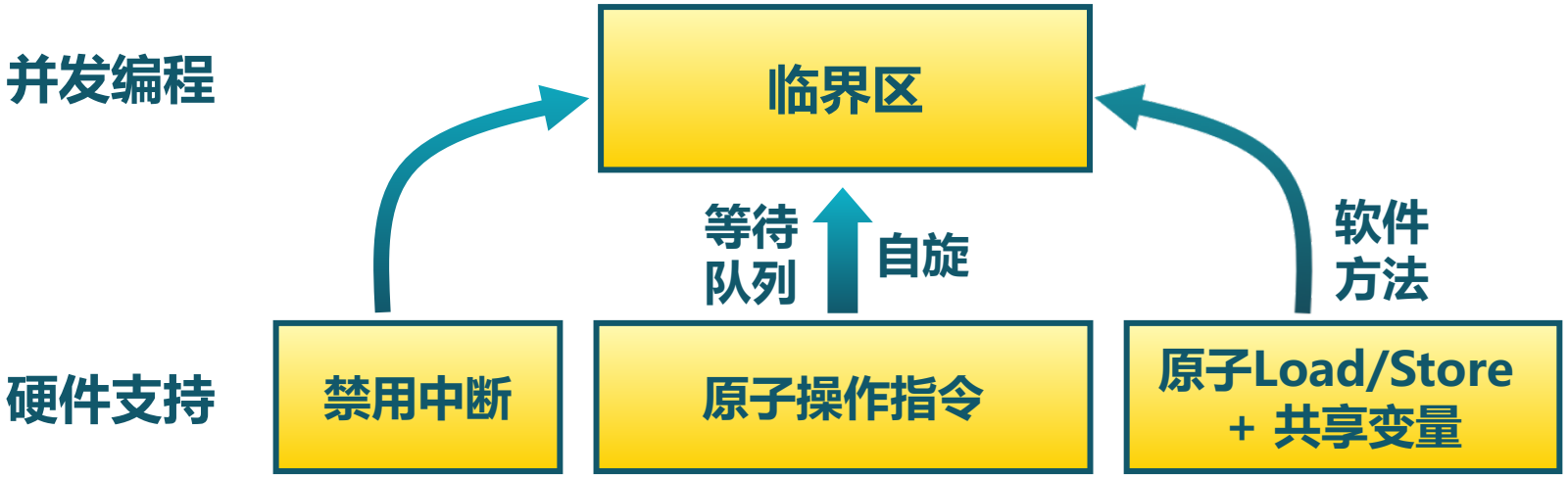
- 方案描述
  - 利用同步机制：锁(lock)
  - Lock.Acquire()
    - 在锁被释放前一直等待，然后获得锁
    - 如果两人都在等待同一个锁，并且同时发现锁被释放了，那么只有一个能够获得锁
  - Lock.Release()
    - 解锁并唤醒任何等待中的进程

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

- **关键**：锁的操作必须是原子操作！

临界区的保障

- 基于软件
- 硬件中断
- 原子操作指令与互斥锁



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同步它们的行为

```
do {
   enter section 进入区
          critical section
   exit section  退出区
          remainder section
} while (1);
```

方案一

- 共享变量

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

- 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进入临界区执行完成后

方案二

- 共享变量

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

- 线程Ti的代码

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

- 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是否在临界区
```

- 线程Ti的代码

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

- 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[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之间互斥的经典的基于软件的解决方法（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

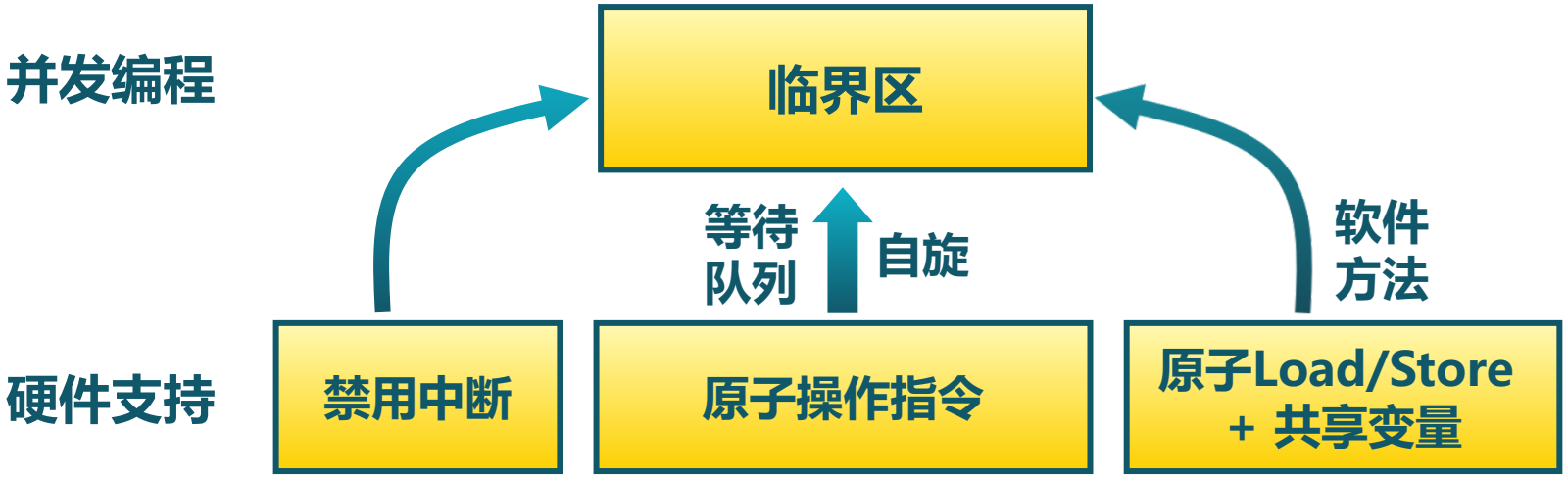
```
do {
       flag[i] = true;
       turn = j;
       while ( flag[j] && turn == j);

             CRITICAL SECTION

       flag[i] = false;

             REMAINDER SECTION

   } while (true);
```

基于软件的解决方法的分析

- 复杂
  - 需要算法和两个进程/线程之间的共享数据项来保证锁的原子性
- 需要“忙等待”
  - 浪费CPU时间

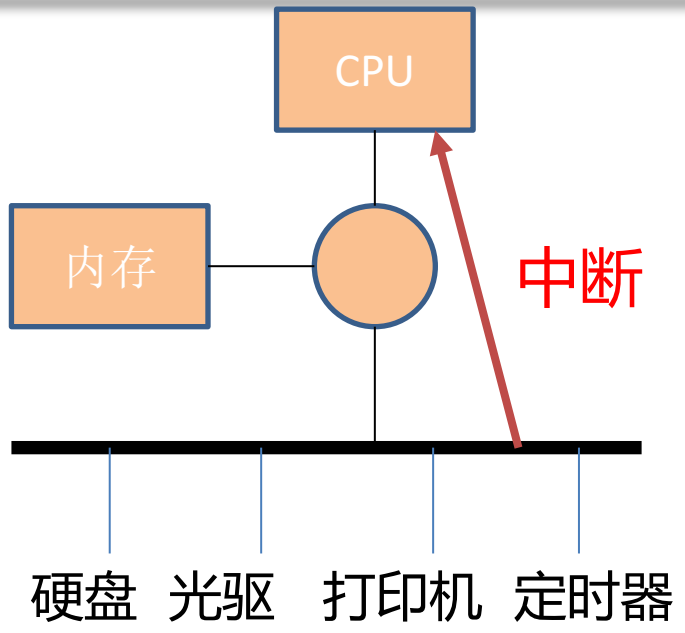
临界区的保障

- 基于软件
- 硬件中断
- 原子操作指令与互斥锁



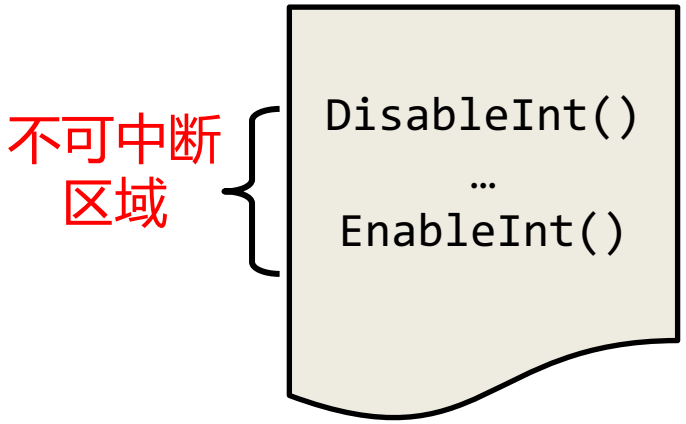
禁用中断实现互斥

- 使用中断
  - 可以实现抢占式 CPU调度
  - 两种类型的事件能引起切换：
    - 内部事件，放弃 CPU 控制权
    - 外部事件，使得 CPU 重新调度



- 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
  - 大多数时候用串行思维
  - **延迟**处理外部事件



禁用中断的简单方法

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

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

- 有什么问题吗？
  - 锁整个冰箱

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，可能导致永久等待

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;
}
```

- 仍然可能导致“永远等待”

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;
}
```

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

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;
}
```

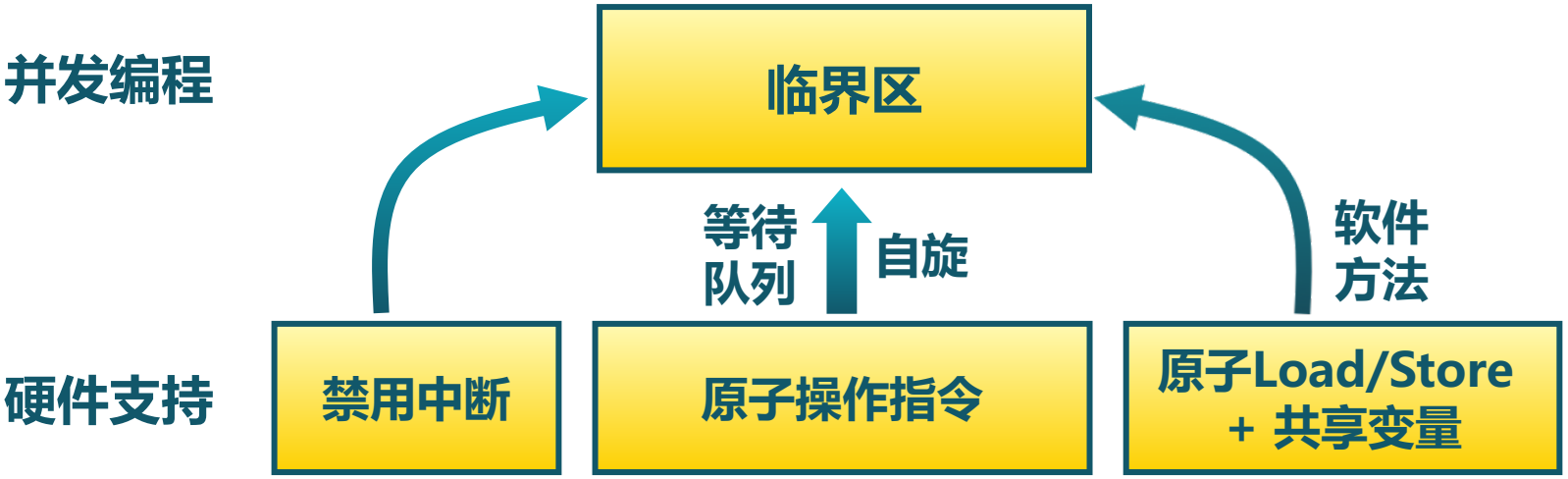
- yield前启用中断

中断的缺点

- 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
  - 长时间无法响应中断

临界区的保障

- 基于软件
- 硬件中断
- 原子操作指令与互斥锁



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，然后返回真或假
  - 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;
}
```

如果锁被释放，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

如果锁处于忙状态，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

无忙等待锁

忙等待

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

无忙等待

```
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);
}
```

原子操作指令

- 交换指令（exchange）
  - 交换寄存器与内存

```
 void Exchange (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

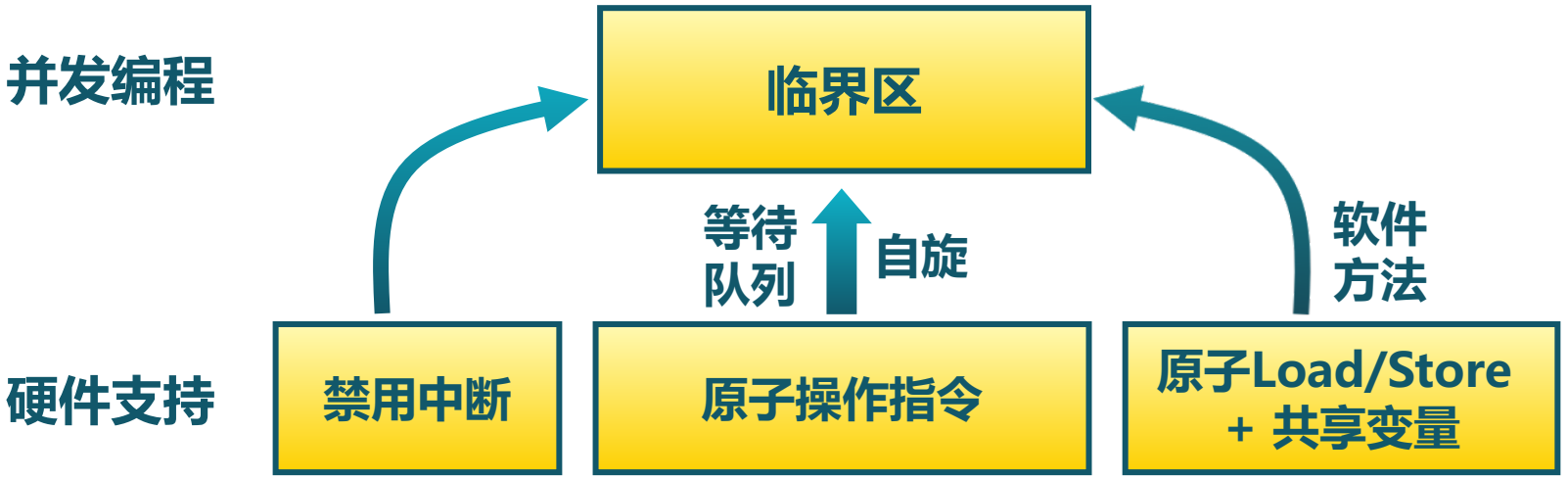
- Load linked 和 Conditional store (LL-SC)
  - 在一条指令中读一个值(Load linked)
  - 做一些操作
  - Store 时，检查 load linked 之后，值是否被修改过。如果没有，则成功修改；否则，从头再来
- Fetch-and-Add 或 Fetch-and-Op
  - 用于大型共享内存多处理器系统的原子指令

优缺点

- 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

总结

- 进程/线程通信
  - 数据传输
  - 信息同步
- 进程/线程同步
  - 临界区
  - 进入临界区的操作须为原子操作



总结

- 基于软件方法的同步机制
  - 实现复杂，依赖一定算法：Dekkers和Peterson算法
- 基于关中断的同步机制
  - 简单直接
  - 可能长时间无法响应中断，导致别的进程饥饿
  - 只能用于单核处理器
- 基于原子操作指令的同步机制
  - 等待时自旋或进入等待队列
  - 支持线程数比处理器数目多