计算机与操作系统
第六章　并发程序设计

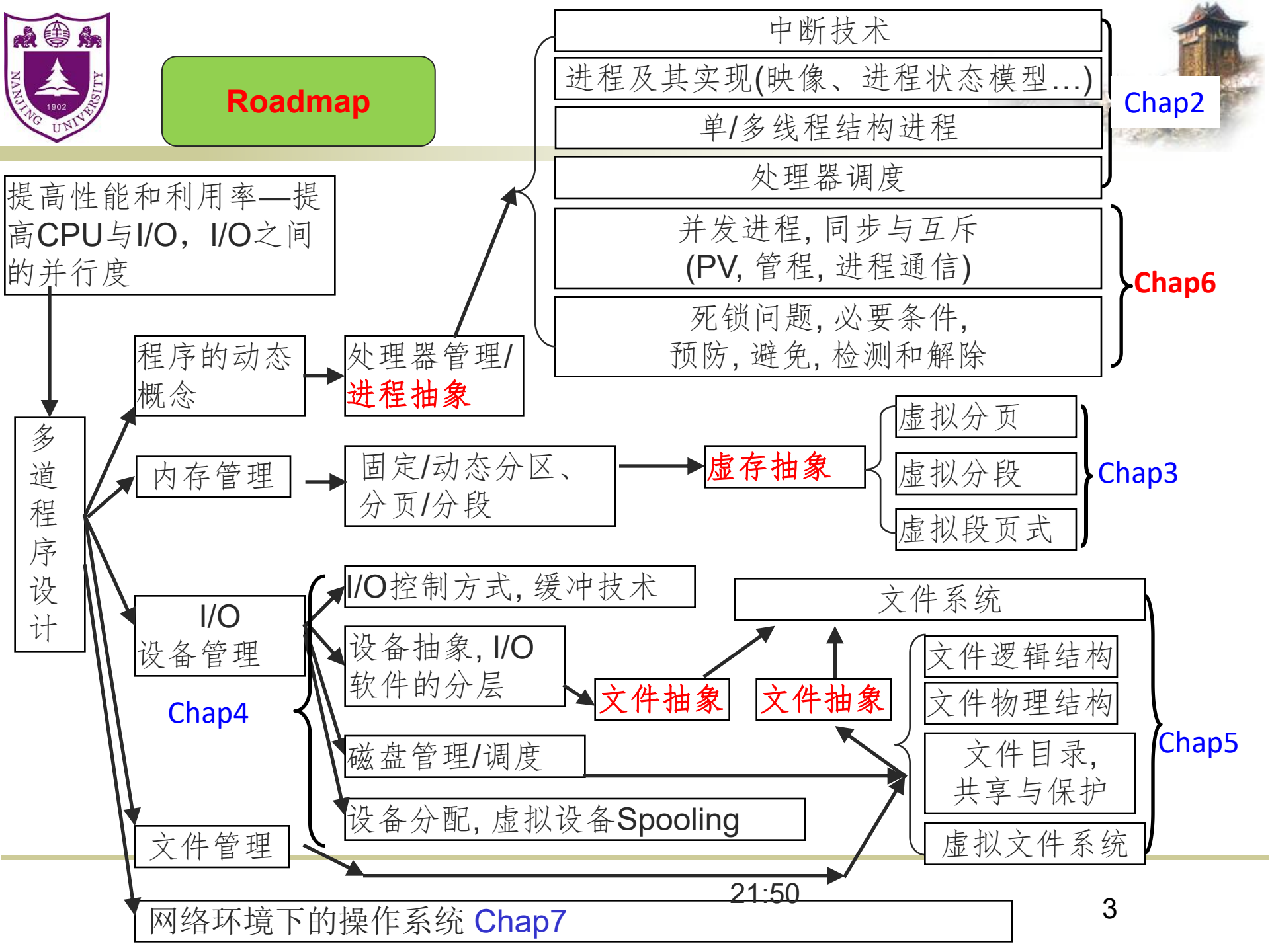
葛季栋

南京大学软件学院

本主题教学目标

1. 了解程序的并发性与并发程序设计
2. 掌握临界区互斥及其解决方案
3. 熟练使用PV进行程序设计
4. 掌握Hoare管程
5. 掌握消息传递

Roadmap


中断技术
进程及其实现(映像、进程状态模型…)
单/多线程结构进程
处理器调度
Chap2
并发进程, 同步与互斥
(PV, 管程, 进程通信)
死锁问题, 必要条件,
预防, 避免, 检测和解除
Chap6
提高性能和利用率—提高CPU与I/O，I/O之间的并行度
多道程序设计
程序的动态概念
处理器管理/进程抽象
内存管理
固定/动态分区、分页/分段
虚存抽象
虚拟分页
虚拟分段
虚拟段页式
Chap3
I/O设备管理
Chap4
I/O控制方式, 缓冲技术
设备抽象, I/O软件的分层
文件抽象
文件抽象
文件系统
文件逻辑结构
文件物理结构
文件目录,
共享与保护
虚拟文件系统
Chap5
磁盘管理/调度
设备分配, 虚拟设备Spooling
文件管理
网络环境下的操作系统 Chap7

第六章 并发程序设计

6.1 并发进程

6.2 临界区管理

6.3 信号量与PV操作

6.4 管程

6.5 进程通信

计算机与操作系统
第六章　并发程序设计
6.1　并发进程

葛季栋

南京大学软件学院

6.1 并发进程

6.1.1 顺序程序设计

6.1.2 进程的并发性

6.1.3 进程的交互：竞争和协作

6.1.1 顺序程序设计

- 一个进程在处理器上的顺序执行是严格按序的，一个进程只有当一个操作结束后，才能开始后继操作
- 顺序程序设计是把一个程序设计成一个顺序执行的程序模块，顺序的含义不但指一个程序模块内部，也指两个程序模块之间

顺序程序设计特点

- 程序执行的顺序性
- 程序环境的封闭性
- 执行结果的确定性
- 计算过程的可再现性

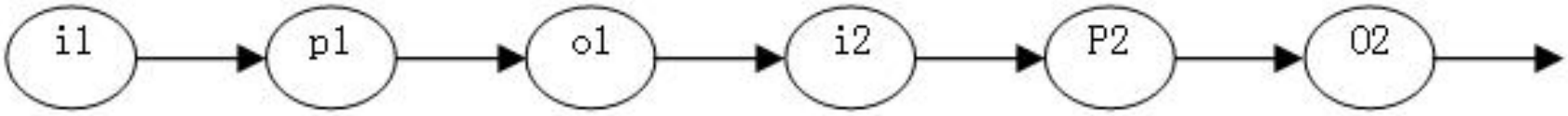
6.1.2 进程的并发性

- 进程的并发性(Concurrency)是指一组进程的执行在时间上是重叠的
- 例如：有两个进程A(a1、a2、a3)和B(b1、b2、b3)并发执行，若允许进程交叉执行，如执行操作序列为a1，b1，a2，b2，a3，b3或a1，a2，b1，b2，b3，a3等，则说进程A和B的执行是并发的
- 从宏观上看，并发性反映一个时间段中几个进程都在同一处理器上，处于运行还未运行结束状态
- 从微观上看，任一时刻仅有一个进程在处理器上运行

并发程序设计

- 使一个程序分成若干个可同时执行的程序模块的方法称**并发程序设计**(concurrent programming)，每个程序模块和它执行时所处理的数据就组成一个进程

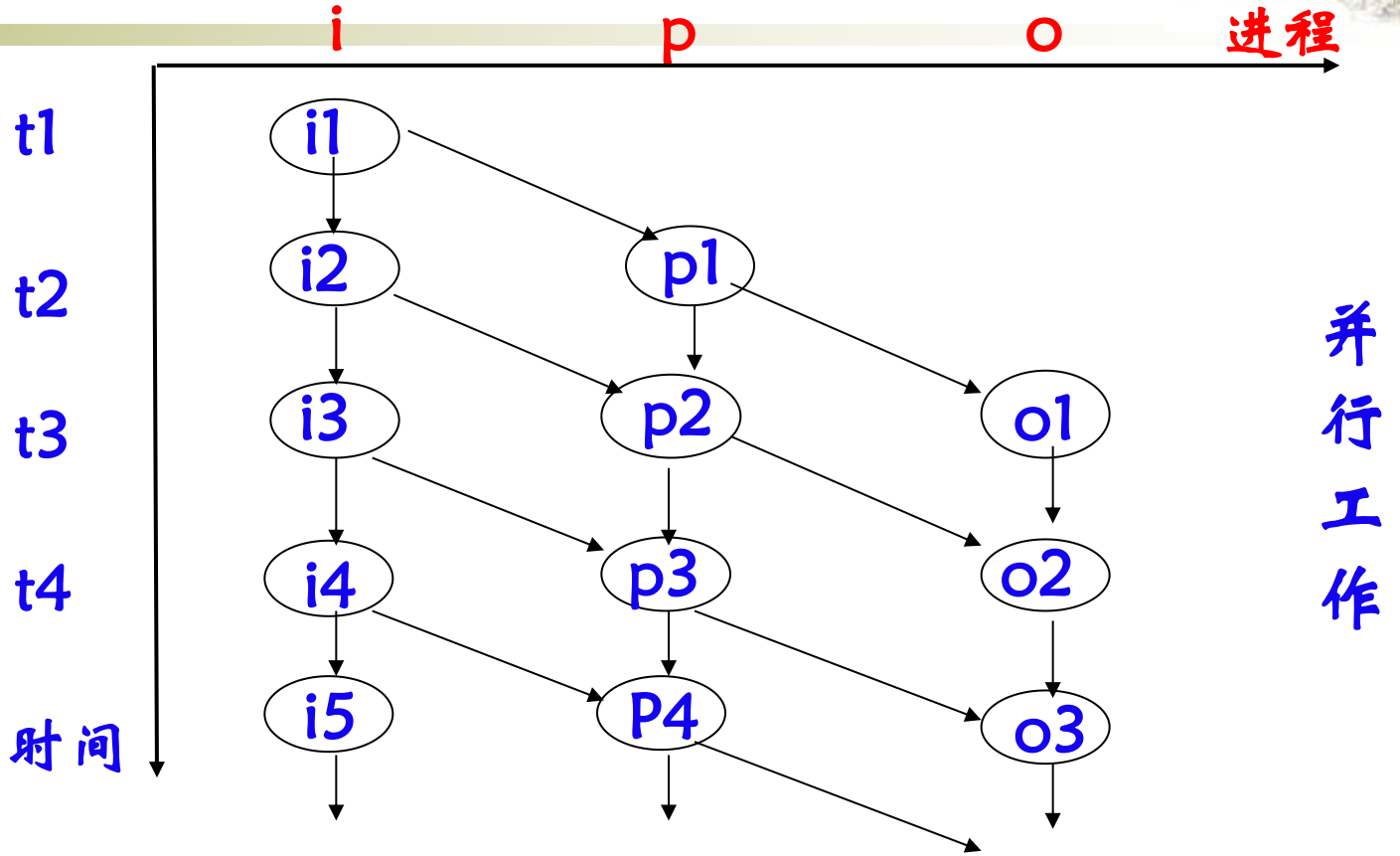
进程的并发性(1)



(a) 串行工作

- 由于程序是按照while(TRUE) {input，process，output}串行地输入 - 处理 - 输出来编制的，所以该程序只能顺序地执行，这时系统的效率相当低
- 如果把求解这个问题的程序分成三部分：
  - i: while(TRUE) {input，send}
  - p: while(TRUE) {receive，process，send}
  - o: while(TRUE) {receive，output}

进程的并发性(2)



小程序1：循环执行，读入字符，将读入字符送缓冲区1

小程序2：循环执行，处理缓冲区1中的字符，把计算结果送缓冲区2

小程序3：循环执行，取出缓冲区2中的计算结果并写到磁带上

进程的并发性(3)

- 每一部分是一个小程序，它们可并发执行，并会产生制约关系，其中send和receive操作用于小程序之间通过通信机制解决制约关系，以便协调一致地工作

并发进程的分类

- 并发进程分类：无关的，交互的
- 无关的并发进程：一个进程的执行与其他并发进程的进展无关
  - 并发进程的无关性是进程的执行与时间无关的一个充分条件，又称为Bernstein条件
- 交互的并发进程：不满足Bernstein条件，一个进程的执行可能影响其他并发进程的结果

Bernstein条件

- $R(p_i)=\{ai1,ai2,\ldots ain\}$，程序pi在执行期间引用的变量集
- $W(pi)=\{bi1,bi2,\ldots bim\}$，程序pi在执行期间改变的变量集
- 若两个进程的程序p1和p2能满足Bernstein条件，即满足：

$$(R(p1)\cap W(p2))\cup(R(p2)\cap W(p1))\cup(W(p1)\cap W(p2))=\varnothing$$

则并发进程的执行与时间无关

Bernstein条件举例

例如，有如下分属四个进程中的四条语句：

S1: a := x + y

S2: b := z + 1

S3: c := a – b

S4: w := c + 1

于是有：R(S1)={x,y}，R(S2)={z}，R(S3)={a,b}，R(S4)={c}；　W(S1)={a}，W(S2)={b}，W(S3)={c}，W(S4)={w}

S1和S2可并发执行，满足Bernstein条件

其他语句并发执行可能会产生与时间有关的错误

与时间有关的错误

- 对于一组交互的并发进程，执行的相对速度无法相互控制，各种与时间有关的错误就可能出现。
- 与时间有关错误的表现形式：
  - 结果不唯一
  - 永远等待

(1) 机票问题

//飞机票售票问题

```
void T1( ) {
{按旅客订票要求找到Aj};
 int X1=Aj;
 if(X1>=1) {
    X1--;
      Aj=X1;
     {输出一张票};
  }
  else
 {输出信息"票已售完"};
}
```

```
void T2( ) {
{按旅客订票要求找到Aj};
 int X2=Aj;
 if(X2>=1) {
    X2--;
   Aj=X2;
    {输出一张票};
  }
 else
   {输出信息"票已售完"};
    }
```

(1) 机票问题

* 此时出现把同一张票卖给两个旅客的情况，两个旅客可能各自都买到一张同天同次航班的机票，可是，Aj的值实际上只减去1，造成余票数不正确。特别是，当某次航班只有一张余票时，可能把一张票同时售给两位旅客

主存管理问题

申请和归还主存资源问题

int X=memory; //memory为初始主存容量

```
void borrow(int B) {
 (1) while(B>X)
 (5) {进程进入等待主存资源队列};
  X=X-B ;
 {修改主存分配表，进程获得主存资源};
}
```

```
void return(int B) {
    (2)X=X+B;
    (3){修改主存分配表};
    (4) {释放等主存资源进程};
    }
```

主存管理问题

* 由于borrow和return共享代表主存物理资源的临界变量X，对并发执行不加限制会导致错误，例如，一个进程调用borrow申请主存，在执行比较B和X大小的指令后，发现B>X，但在执行{进程进入等待主存资源队列}前，另一个进程调用return抢先执行，归还所借全部主存资源；这时，由于前一个进程还未成为等待者，return中的{释放等主存资源进程}相当于空操作，以后当调用borrow的应用进程被置成{等主存资源}时，可能已经没有其他进程再来归还主存，从而，申请资源的进程处于永远等待状态

6.1.3 进程的交互: 竞争与协作

- 进程之间存在两种基本关系：竞争关系和协作关系
- **第一种是竞争关系**，一个进程的执行可能影响到同其竞争资源的其他进程，如果两个进程要访问同一资源，那么，一个进程通过操作系统分配得到该资源，另一个将不得不等待
- **第二种是协作关系**，某些进程为完成同一任务需要分工协作，由于合作的每一个进程都是独立地以不可预知的速度推进，这就需要相互协作的进程在某些协调点上协调各自的工作。当合作进程中的一个到达协调点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，直到其他合作进程发来协调信号或消息后方被唤醒并继续执行

竞争关系带来的问题

- 资源竞争的两个控制问题：
- 一个是死锁(Deadlock)问题：一组进程如果都获得了部分资源，还想要得到其他进程所占有的资源，最终所有的进程将陷入死锁
- 一个是饥饿(Starvation) 问题：一个进程由于其他进程总是优先于它而被无限期拖延
- 操作系统需要保证诸进程能互斥地访问临界资源，既要解决饥饿问题，又要解决死锁问题

竞争关系: 可能产生死锁

- 死锁：一组进程因争夺资源陷入永远等待的状态
- P0 和P1两个进程，均需要使用 S 和 Q 两类资源，每类资源数为1

P0	P1
申请 (S);	申请 (Q);
申请 (Q);	申请 (S);
...	...
释放 (S);	释放 (Q);
释放 (Q);	释放 (S);

竞争关系: 进程的互斥

- 进程的互斥(mutual exclusion)是解决进程间竞争关系(间接制约关系)的手段。进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源

协作关系: 进程的同步

- 进程的同步(Synchronization)是解决进程间协作关系(直接制约关系)的手段。进程同步指两个以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号，当一个进程没有得到来自于另一个进程的消息或信号时则需等待，直到消息或信号到达才被唤醒

进程的交互: 竞争与协作

- 进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，是对进程使用资源次序上的一种协调

计算机与操作系统
第六章 并发程序设计
6.2 临界区管理

葛季栋

南京大学软件学院

6.2 临界区管理

6.2.1 互斥与临界区

6.2.2 临界区管理的尝试

6.2.3 实现临界区管理的硬件设施

6.2.1 互斥与临界区(1)

- 并发进程中与共享变量有关的程序段叫“临界区”(critical section)，共享变量代表的资源叫“临界资源”
- 与同一变量有关的临界区分散在各进程的程序段中，而各进程的执行速度不可预见
- 竞争条件(race condition)
- 如果保证进程在临界区执行时，不让另一个进程进入临界区，即各进程对共享变量的访问是互斥的，就不会造成与时间有关的错误

6.2.1 互斥与临界区(2)

- 临界区调度原则(Dijkstra, 1965)：
  - 一次至多一个进程能够进入临界区内执行
  - 如果已有进程在临界区，其他试图进入的进程应等待
  - 进入临界区内的进程应在有限时间内退出，以便让等待进程中的一个进入

6.2.1 互斥与临界区(3)

```
    /*  Process 1 */
void P1
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

```
    /*  Process 2 */
void P2
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

。 。 。

```
    /*  Process n */
void Pn
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

6.2.2 临界区管理的尝试 (1)

如果并发执行轨迹是①③④②,
导致两个进程都被阻塞, 均不能进入临界区

P1和PP2进程都满足进入条件

```
bool inside1=false; //P1不在其临界区内
bool inside2=false; //P2不在其临界区内
cobegin /*cobegin和coend表示括号中的进程是一组并发进程*/
```

process P1( ) {	process P2( ) {
while(inside2);//等待①	while(inside1);//等待③
inside1=true; ②	inside2=true; ④
{临界区};	{临界区};
inside1=false;	inside2=false;
}	}

```
coend
```

思考: 该算法存在的问题

6.2.2 临界区管理的尝试 (2)

如果并发执行轨迹是①③④②，
导致两个进程都被阻塞，均不能进入临界区

P1和P2进程都被阻塞在While循环

```
bool inside1=false; //P1不在其临界区
bool inside2=false; //P2不在其临界区内
cobegin
```

process P1( ) {	process P2( ) {
inside1=true; ①	inside2=true; ③
while(inside2);//等待 ②	while(inside1);//等待 ④
{临界区};	{临界区};
inside1=false;	inside2=false;
}	}

```
coend
```

思考：该算法存在的问题

实现临界区的软件算法Peterson算法

```
bool inside[2];
inside[0]=false;
inside[1]=false;
enum {0,1} turn;
cobegin
```

假设并发时的可能执行顺序(部分举例)：

情况1: P0启动①②(turn=1)->跳至P1④⑤(turn=0)⑥(持续循环)
->跳至P0③(循环不成立)->P0进入临界区

情况2: ①->跳至P1④⑤(turn=0)⑥(循环成立)->
跳至P0②(turn=1)③(循环成立)->
跳至P1⑥(循环不成立)->P0进入临界区

```
process P0( ) {
 inside[0]=true; ①
 turn=1; ②
 while(inside[1]&&turn==1);③
        {临界区};
        inside[0]=false;
}
```

```
process P1( ) {
inside[1]=true; ④
  turn=0; ⑤
  while(inside[0]&&turn==0);⑥
       {临界区};
  inside[1]=false;
}
```

```
coend
```

实现临界区的软件算法Peterson算法

```
enum {0,1} turn;
cobegin
process P0( ) {                    process P1( ) {
 turn=1;                           turn=0;
 while(turn==1);                   while(turn==0);
    {临界区};                              {临界区};
 turn=0;                           turn=1;
 }                                 }
coend
```

P0中执行了turn=1, 暂时进不去，等P1中执行turn=0, P0可以进去，P0使用完临界区，退出临界区的时候，将turn=0(好像是多余的), 此时P1还是进不去，要等p0执行turn=1，使得P1有机会进入临界区，之后，P1退出临界区的时候，turn=1，P0暂时进不去，等在P1中执行turn=0，P0可以再次进入临界区，因此，P0和P1使用临界区的次序变成了完全一比一的交替方式，这只能是临界区互斥使用的一个特例，不能满足临界区互斥使用的完全随机性。

6.2.3 实现临界区管理的硬件设施

- (1) 关中断
- (2) 测试并建立指令
- (3) 对换指令

(1) 关中断

- 实现互斥的最简单方法
- 关中断适用场合
- 关中断方法的缺点

(2)测试并建立指令

TS指令的处理过程

```
bool TS(bool &x) {
    if(x) {
       x=false;
       return true;
      }
   else
      return false;
}
```

```
//TS指令实现进程互斥
bool s=true;
cobegin
process Pi( ) {//i=1,2,...,n
        while(!TS(s)); //上锁
        {临界区};
        s=true;          //开锁
}
coend
```

(3) 对换指令

```
void SWAP(bool &a, bool &b) {
        bool temp=a;
        a=b;
        b=temp;
}
```

```
//对换指令实现进程互斥
bool lock=false;
cobegin
Process Pi( ){ //i=1,2,...,n
   bool keyi=true;
    do {
      SWAP(keyi,lock);
    }while(keyi);          //上锁
    {临界区};
    SWAP(keyi,lock);       //开锁
}
coend
```