第4章 同步、通信与死锁

主要内容

- 并发进程
- 临界区管理
- 信号量与PV操作
- 管程
- 进程通信
- 死锁

4.1 并发进程

4.1.1 顺序程序设计

4.1.2 进程的并发性

4.1.3 进程的交互：协作和竞争

4.1.1 顺序程序设计

□进程的顺序性

- 一个进程在顺序处理器上的执行是严格按序的，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
- 程序执行结果的确定性
- 计算过程的可再现性

顺序程序设计的缺点：计算机系统效率不高。

4.1.2 进程的并发性

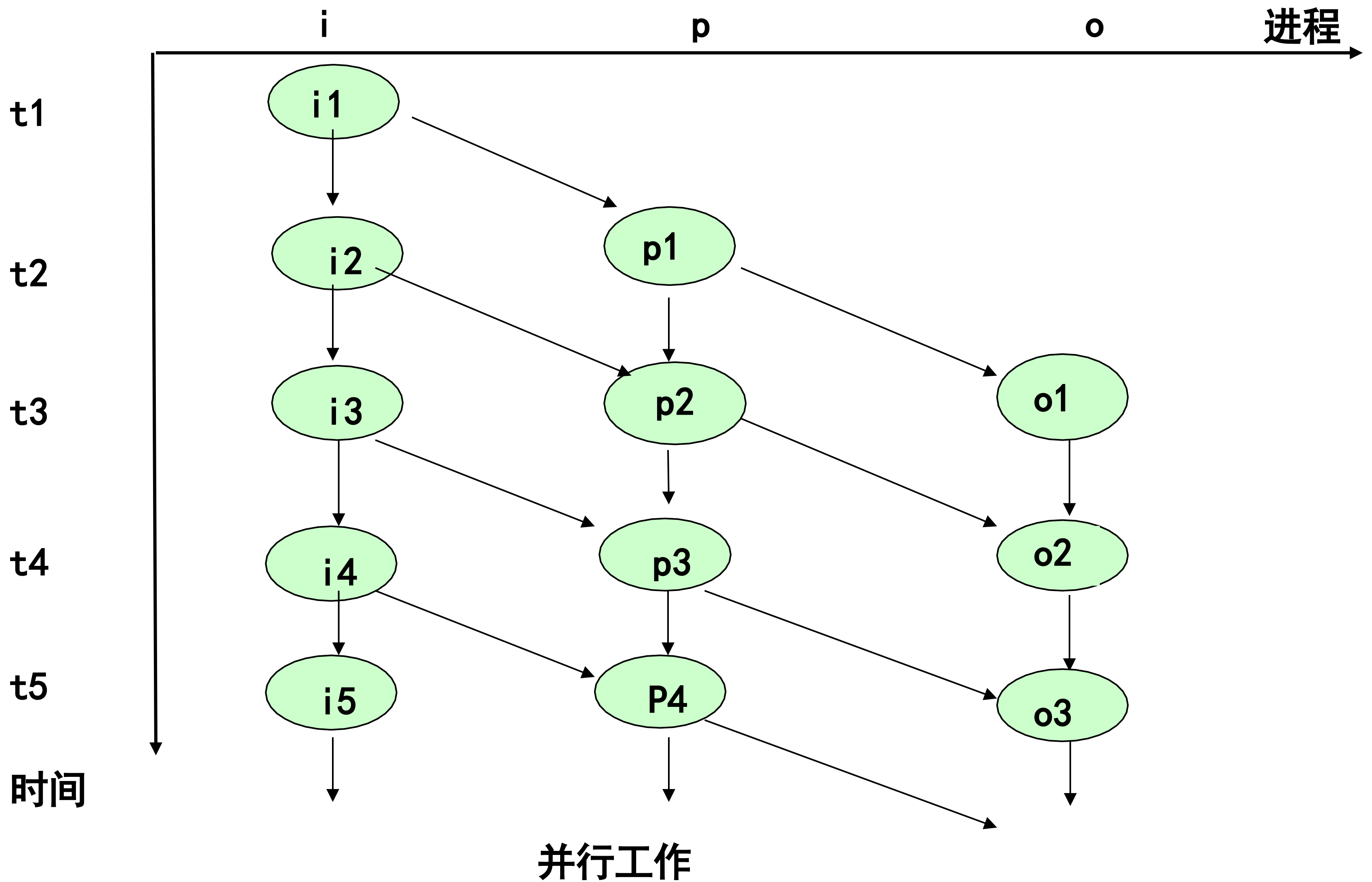
1、并发程序设计

- 进程执行的并发性：一组进程的执行在时间上是重叠的。
- 并发性举例：有两个进程A(a1、a2、a3)和B(b1、b2、b3)并发执行。

  a1、a2、a3、b1、b2、b3 顺序执行

  a1、b1、a2、b2、a3、b3 并发执行
- 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。

并行工作图示



并行工作

并发的实质

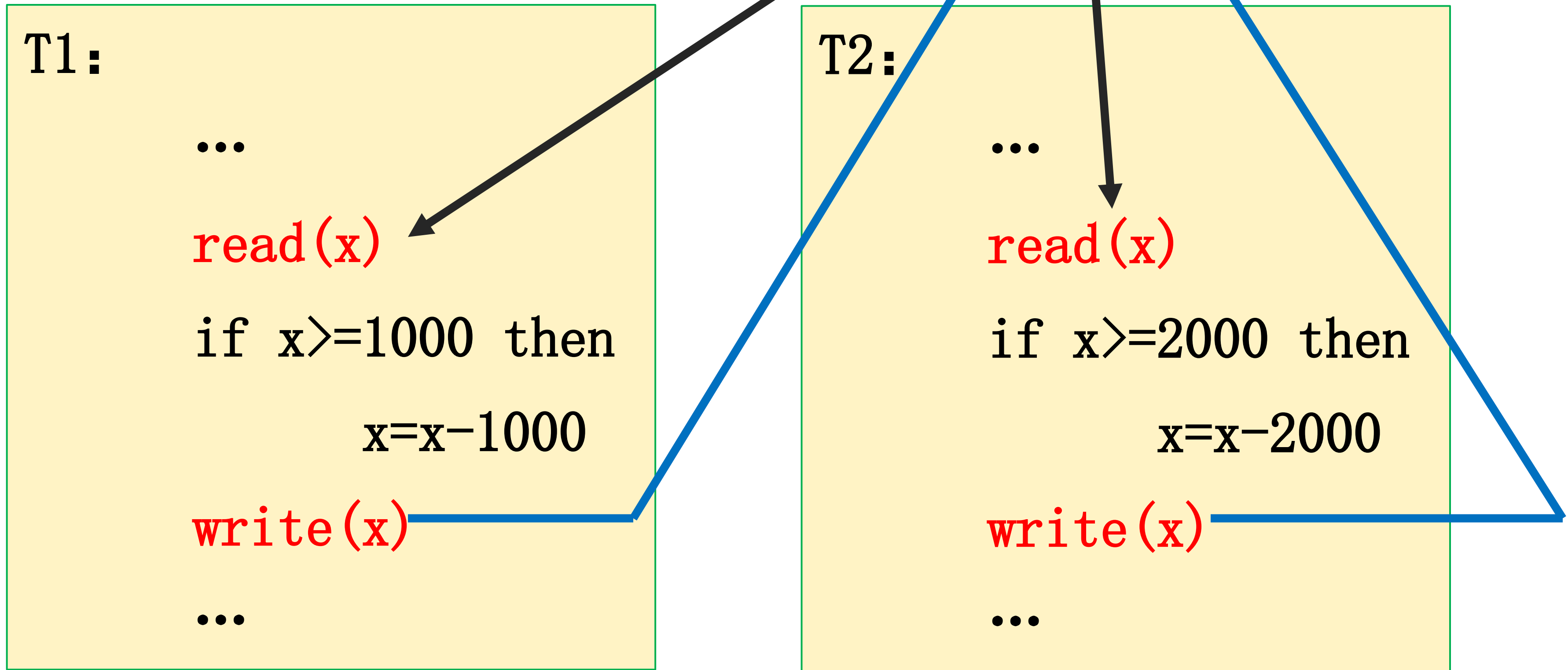
- 并发的实质是一个处理器在几个进程之间的多路复用，并发是对<u>有限的物理资源强制行使多用户共享</u>，消除计算机部件之间的互等现象，以提高系统资源利用率。

3、与时间有关的错误

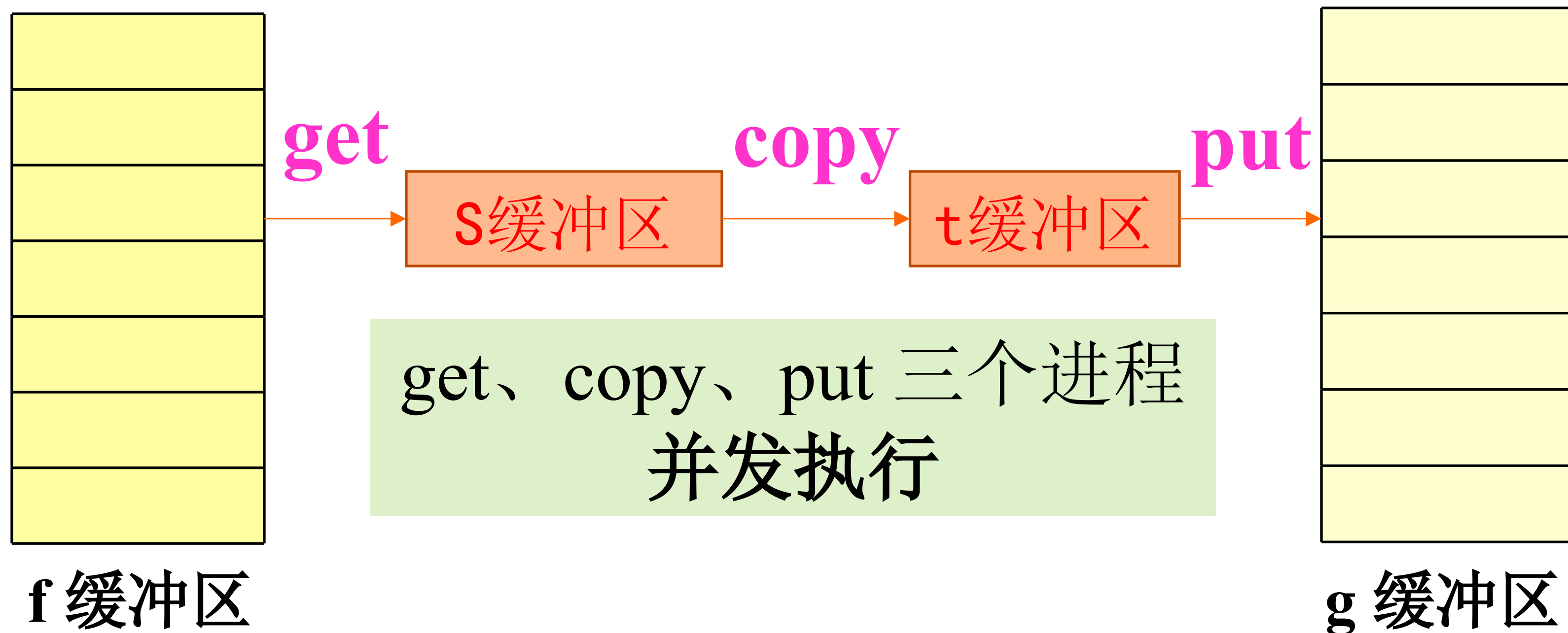
- 对于一组交往的并发进程，执行的相对速度无法相互控制，各种与时间有关的错误就可能出现。
- 与时间有关错误的表现形式：
  - 结果不唯一
  - 永远等待

3、与时间有关的错误（例子1）

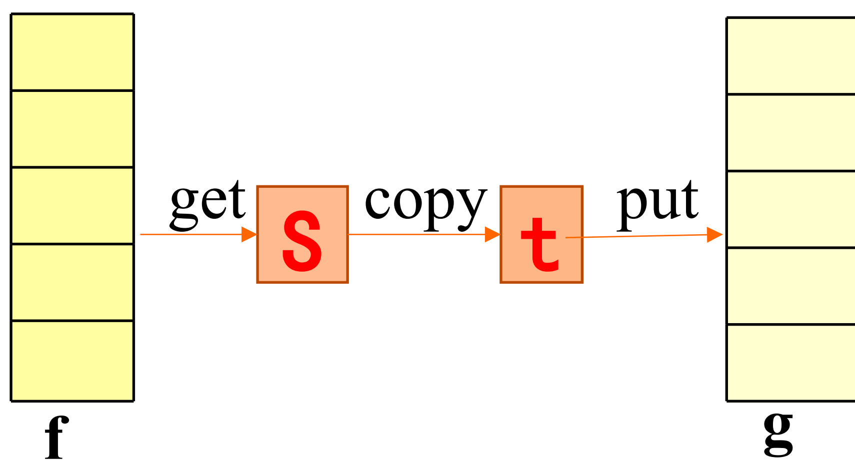
某银行业务系统，某客户的账户有**5000元**，有两个ATM机T1，T2



3、与时间有关的错误（例子2）



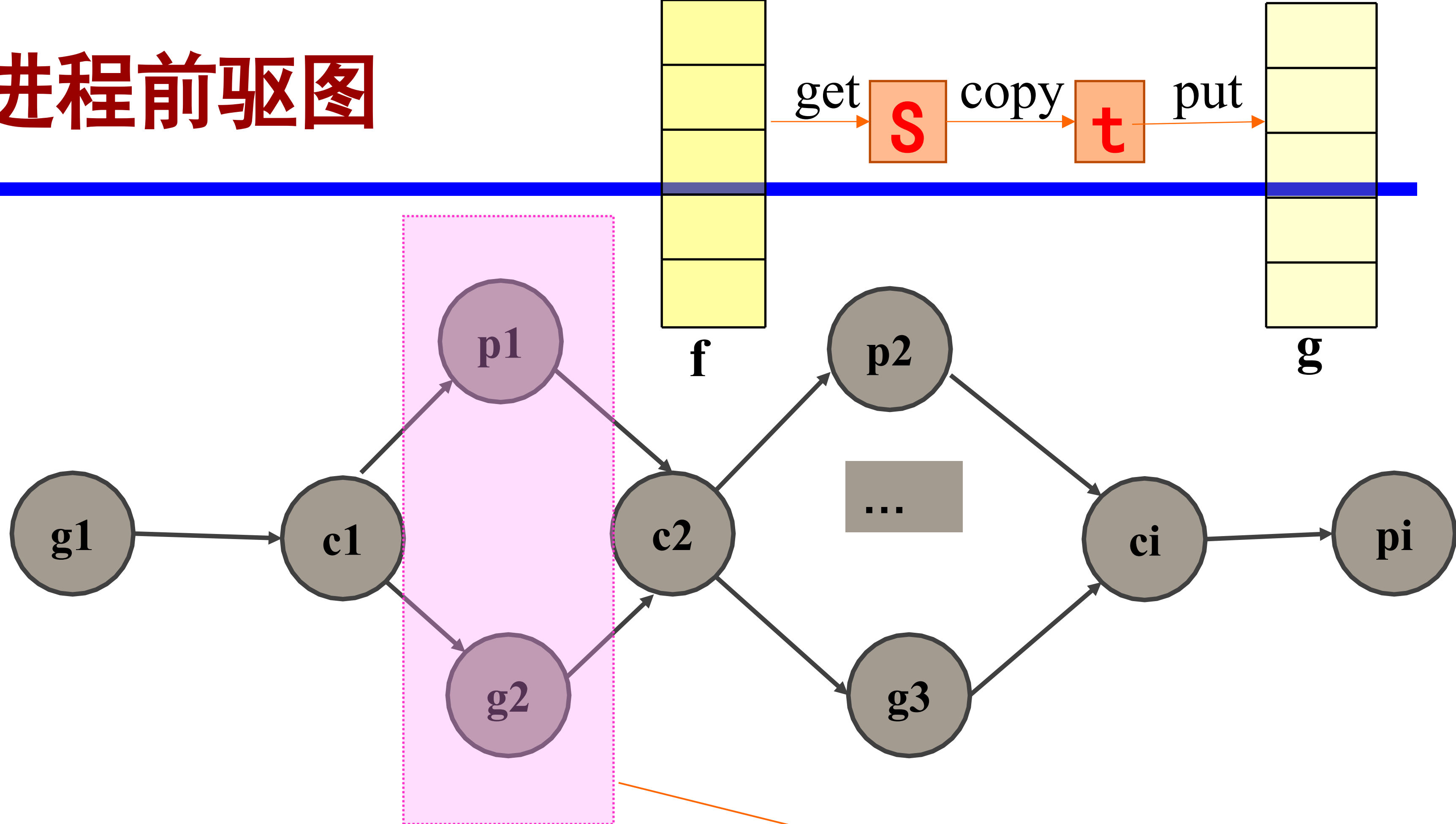
- （1）get把数据读入s，但s中数据还未被copy取走，第二次get过来的进程将s中数据覆盖。
- （2）若put进程先执行，但t中数据未准备好，可能将不需要的数据取走。



并发执行分析

当前状态	f	s	t	g	正误
	(3，4，…, m)	2	2	(1, 2)	
执行顺序	假设 g，c，p 为get copy put 的一次循环				
g→c→p	(4，5，…，m)	3	3	(1, 2, 3)	√
g→p→c	(4，5，…，m)	3	3	(1, 2, 2)	×
c→g→p	(4，5，…，m)	3	2	(1, 2, 2)	×
c→p→g	(4，5，…，m)	3	2	(1, 2, 2)	×
p→c→g	(4，5，…，m)	3	2	(1, 2, 2)	×
p→g→c	(4，5，…，m)	3	3	(1, 2, 2)	×

进程前驱图



并发环境下进程间的制约关系

3、与时间有关的错误（例子3 ）

//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"};
}
```

3、与时间有关的错误（例子3 ）

T1、T2并发执行，可能出现如下交叉情况：

T1:X1=Aj;   //X1=m

T2:X2=Aj;   //X2=m

T2:X2--;Aj=X2;{输出一张票};   //Aj=m-1

T1:X1--;Aj=X1;{输出一张票};   //Aj=m-1

同一张票卖给两位旅客

3、与时间有关的错误（例子4 ）

（永远等待）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) {
 while(B>X)   ①
{进程进入等待主存资源队列}; ②
  X=X-B ;
 {修改主存分配表,
  进程获得主存资源}; }
```

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

   }
```
②

3、与时间有关的错误（例子4）

若对borrow和return的并发执行不加限制将会导致错误，例如：

Borrow:while(B>X);

Return:X=X+B;{修改主存分配表};{释放等待主存资源的进程};

此时，因为borrow还没有进入等待队列，因此，return的释放操作是空操作，当borrow进入等待队列时，可能没有进程再来归还，处于永远等待状态。

3.1.3 进程的交互：竞争与协作(1)

第一种是竞争关系

- **进程互斥**：若干个进程因相互争夺独占型资源时所产生的竞争制约关系。
- 资源竞争的两个控制问题：

  一个是死锁(Deadlock)问题，

  一个是饥饿(Starvation) 问题

  既要解决饥饿问题，又要解决死锁问题。

进程的交往：竞争与协作(2)

第二种是协作关系

进程同步：为完成共同任务的并发进程基于某个条件来协调它们的活动，因为需要在某些位置上排定执行的先后次序而等待、传递信号或消息所产生的协作制约关系。

- 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。

4.2 临界区管理

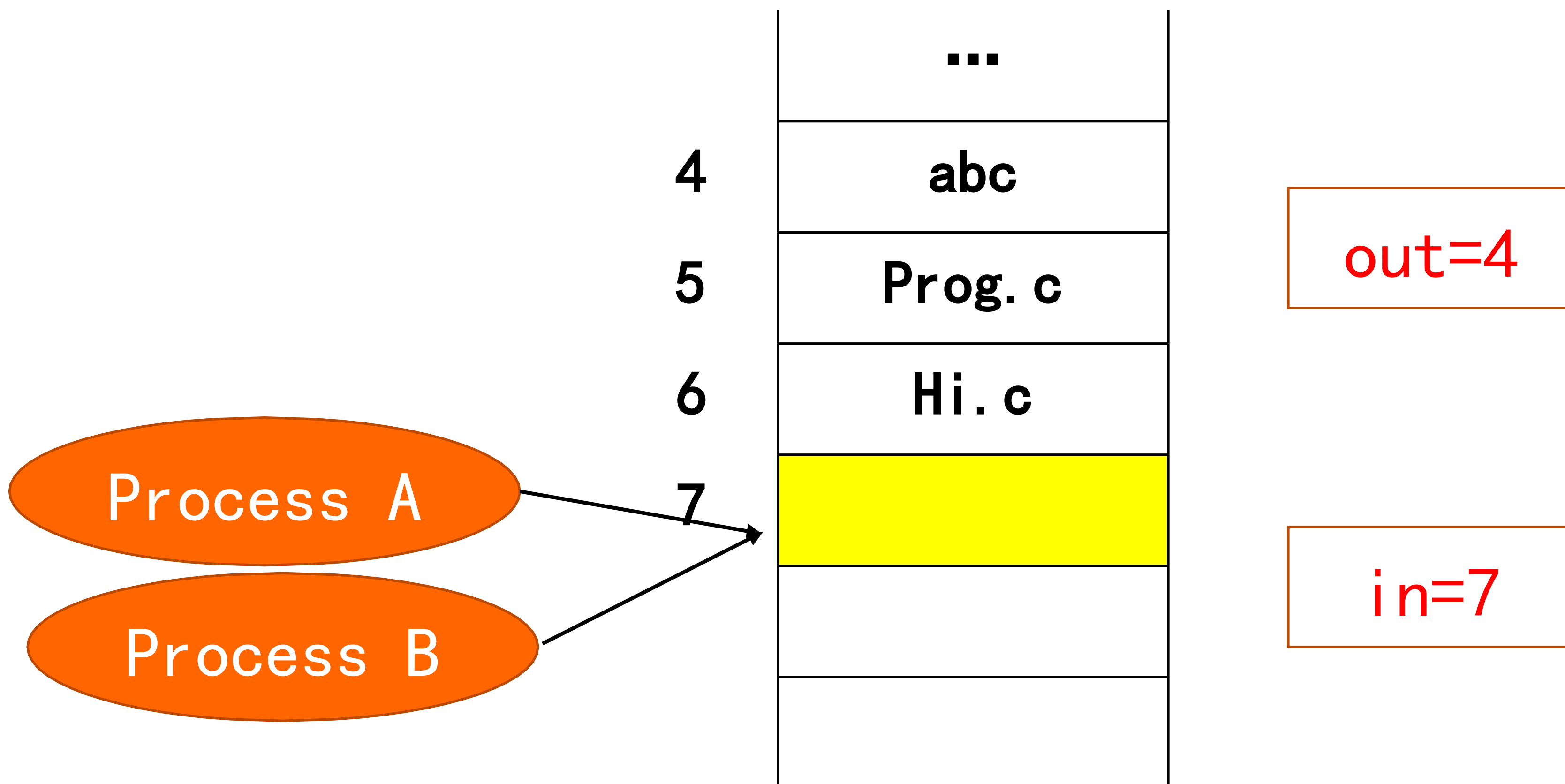
4.2.1　互斥与临界区

4.2.2　实现临界区管理的几种尝试

4.2.3　实现临界区管理的软件方法

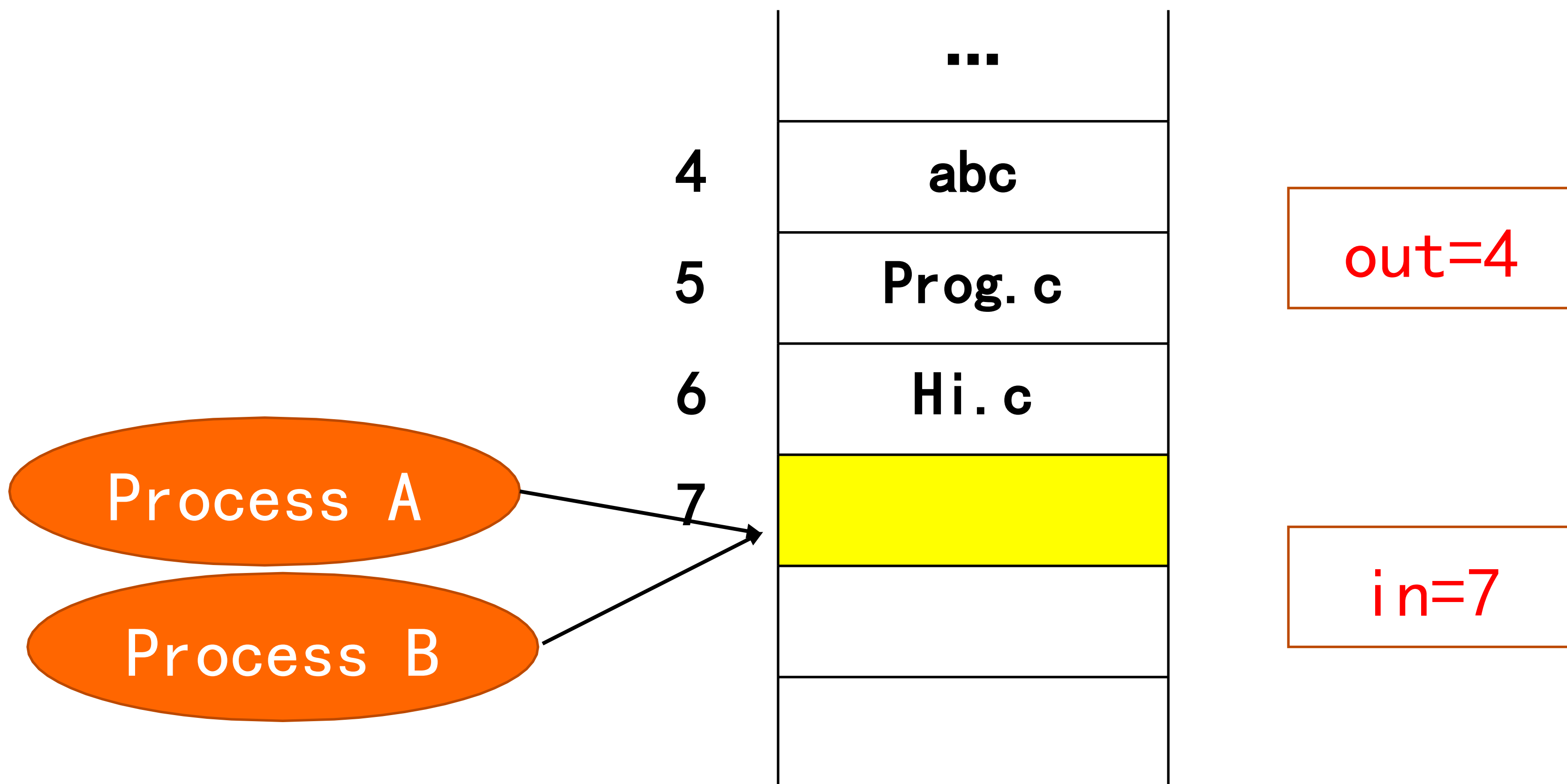
4.2.4　实现临界区管理的硬件设施

竞争条件（空闲区域7）



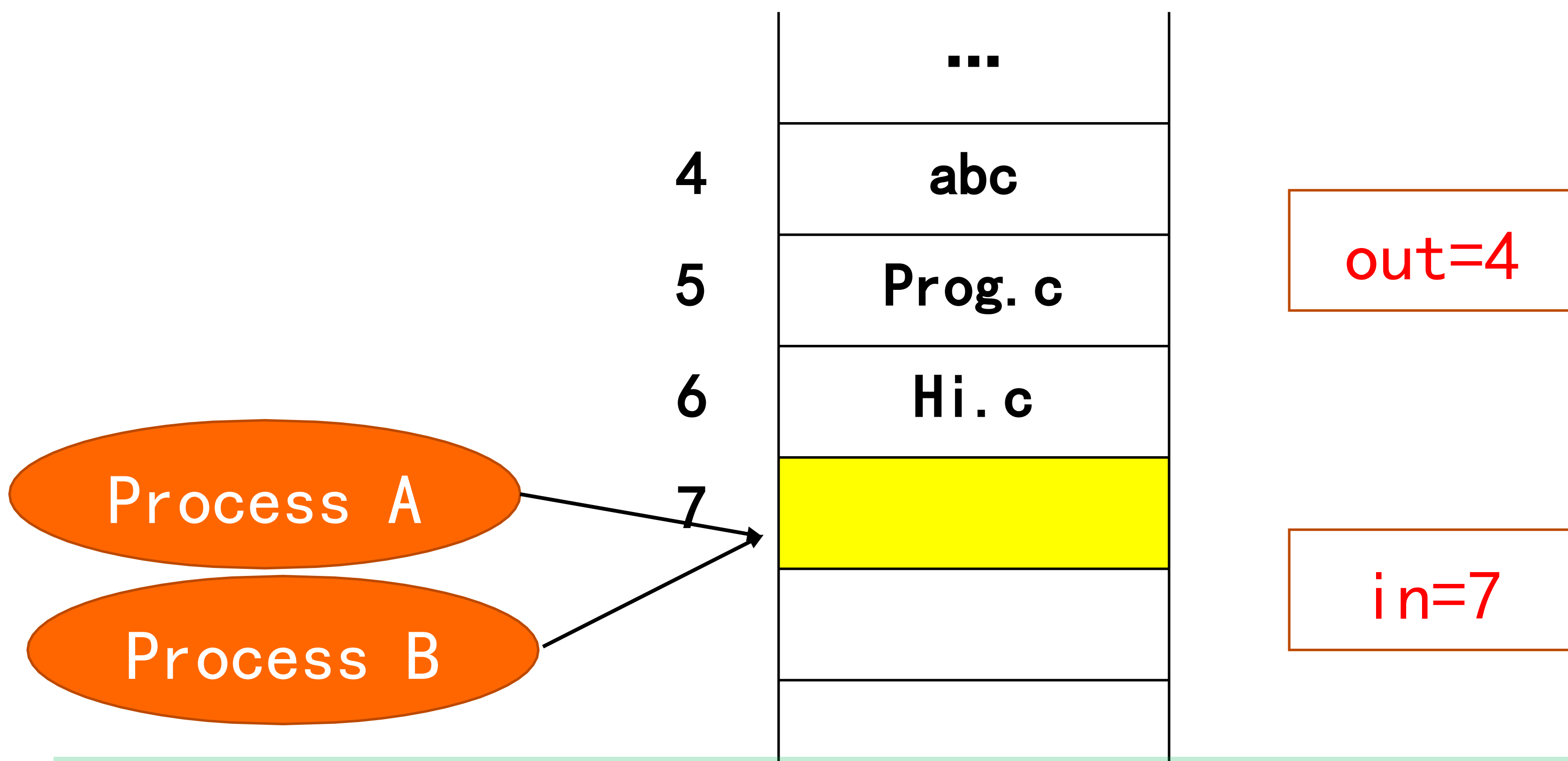
进程A 和B 都申请把要打印的文件名放入 7 ， 然后打印

竞争条件（空闲区域7）



进程A 读取到in=7，将文件名放入7，即将 修改in时，进程A被切换下CPU，进程B上CPU。

竞争条件（空闲区域7）



进程B 读取到in=7，将文件名放入7，修改in=8
结果：进程A中文件未能打印，因为被进程B覆盖了。

4.2.1 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”， 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知。
- 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。

竞争互斥

由于各个进程要求使用共享资源（变量、文件）

- 而这些资源需要排他性使用，各进程之间竞争使用这些资源——这一关系称为进程互斥

临界资源

- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量

临界区（互斥区）

- 各个进程对某个临界资源（共享变量）实施操作的程序片段

互斥与临界区(2)

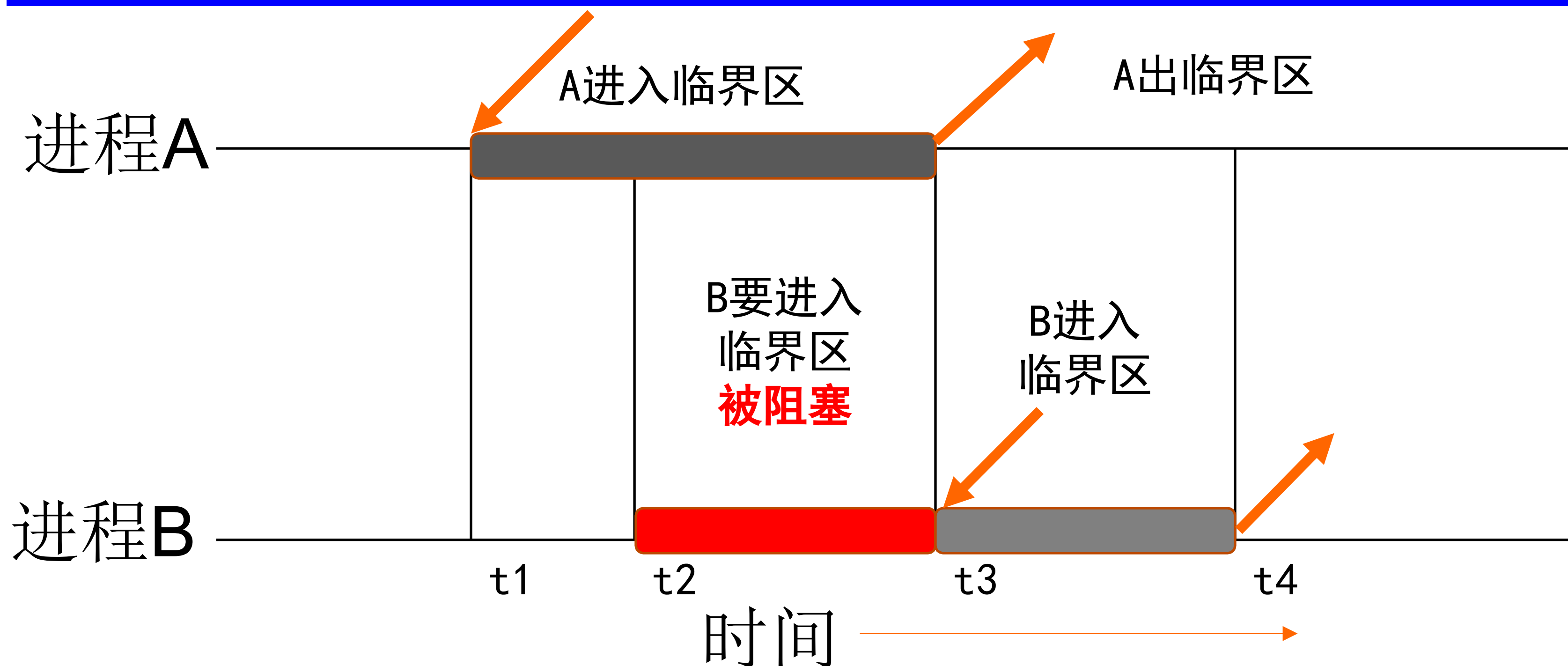
临界区调度原则：

- 一次至多一个进程能够进入临界区内执行；玘
- 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。

临界区调度原则总结：

- 互斥使用、有空让进；
- 忙则等待、有限等待；
- 择一而入，算法可行。

临界区（互斥区）的使用原则



（1）若没有进程在临界区，想进入临界区的进程可进入

（2）不允许两个进程同时处于其临界区

（3）临界区外运行的进程不能阻塞其他进程进入临界区

（4）不得使进程无限期等待进入临界区

实现进程互斥的方法

- 软件方案

Dekker解法、Peterson解法

- 硬件方案

屏蔽中断、TSL（XCHG）指令

软件方法1

- free:临界区空闲标志

true:有进程在临界区; false:无进程在临界区

- 初值: free为false

```
P:
    ……
    while(free);
    free=true;
    临界区
    free=false;
    ……
```

CPU

1

```
Q:
    ……
    while(free);
    free=true;
    临界区
    free=false;
    ……
```

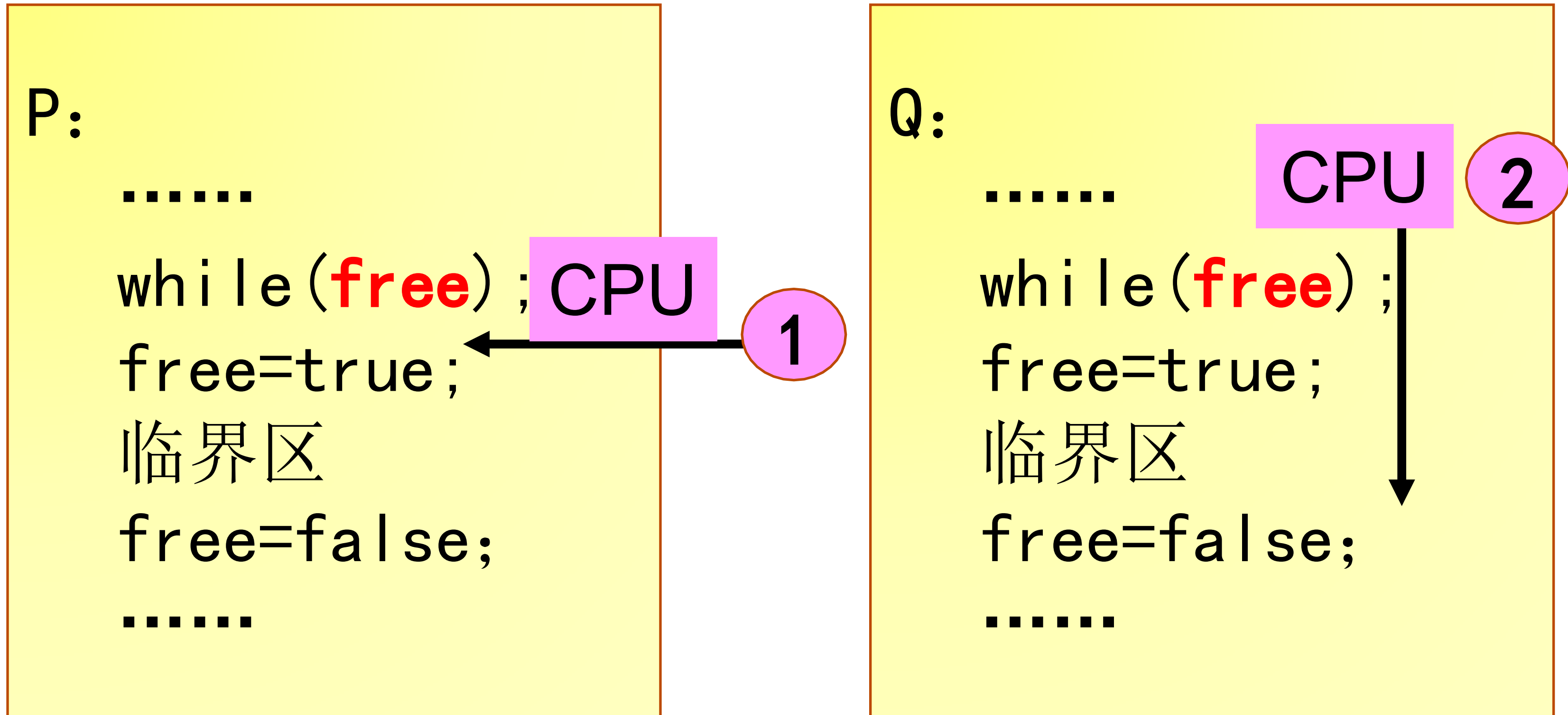
Step1: P先上CPU

软件方法1

- free:临界区空闲标志

true:有进程在临界区；false:无进程在临界区

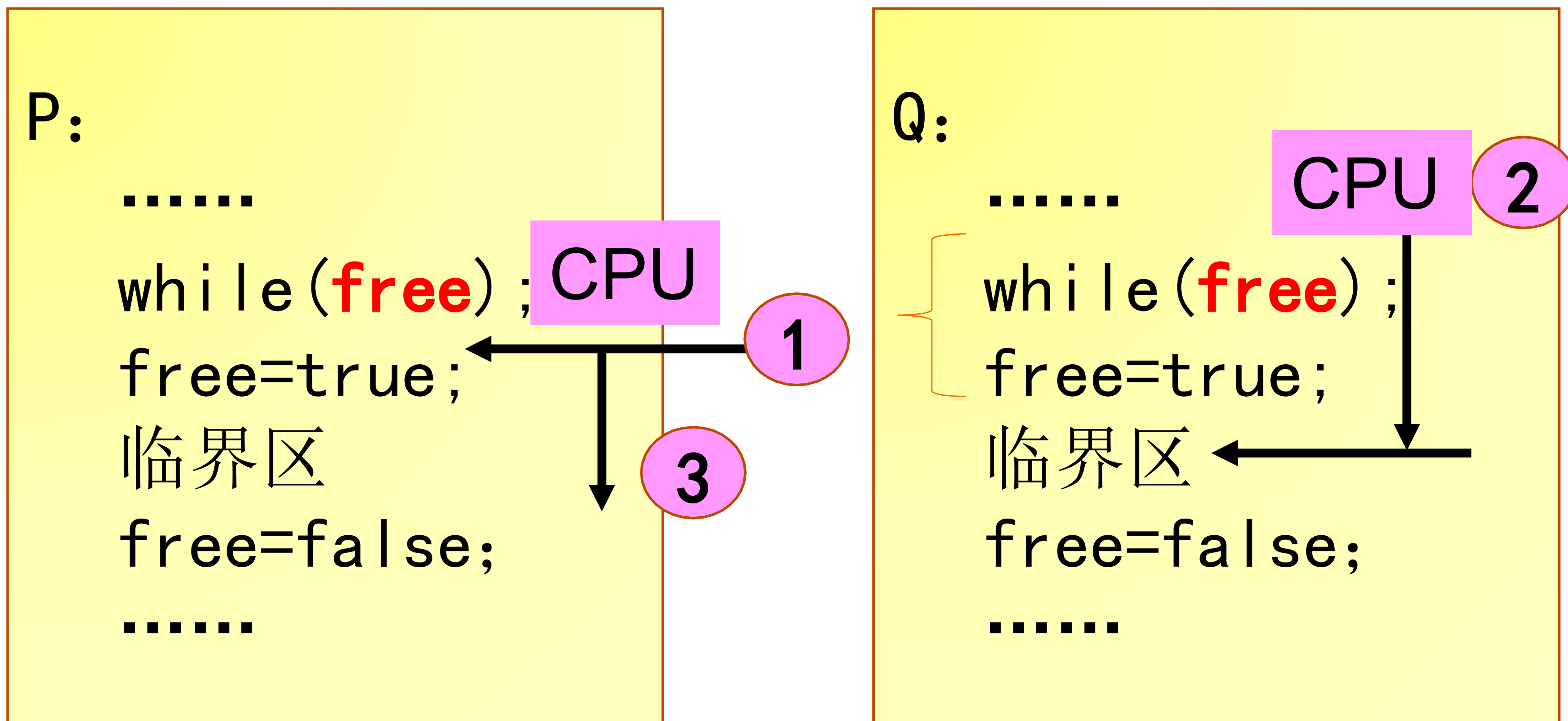
- 初值：free为false



Step2: P下CPU， Q上CPU

软件方法1

- free:临界区空闲标志
  true:有进程在临界区；false:无进程在临界区
- 初值：free为false



Step3: Q下CPU，P上CPU；**此时两个进程都在临界区！该方法有问题。**

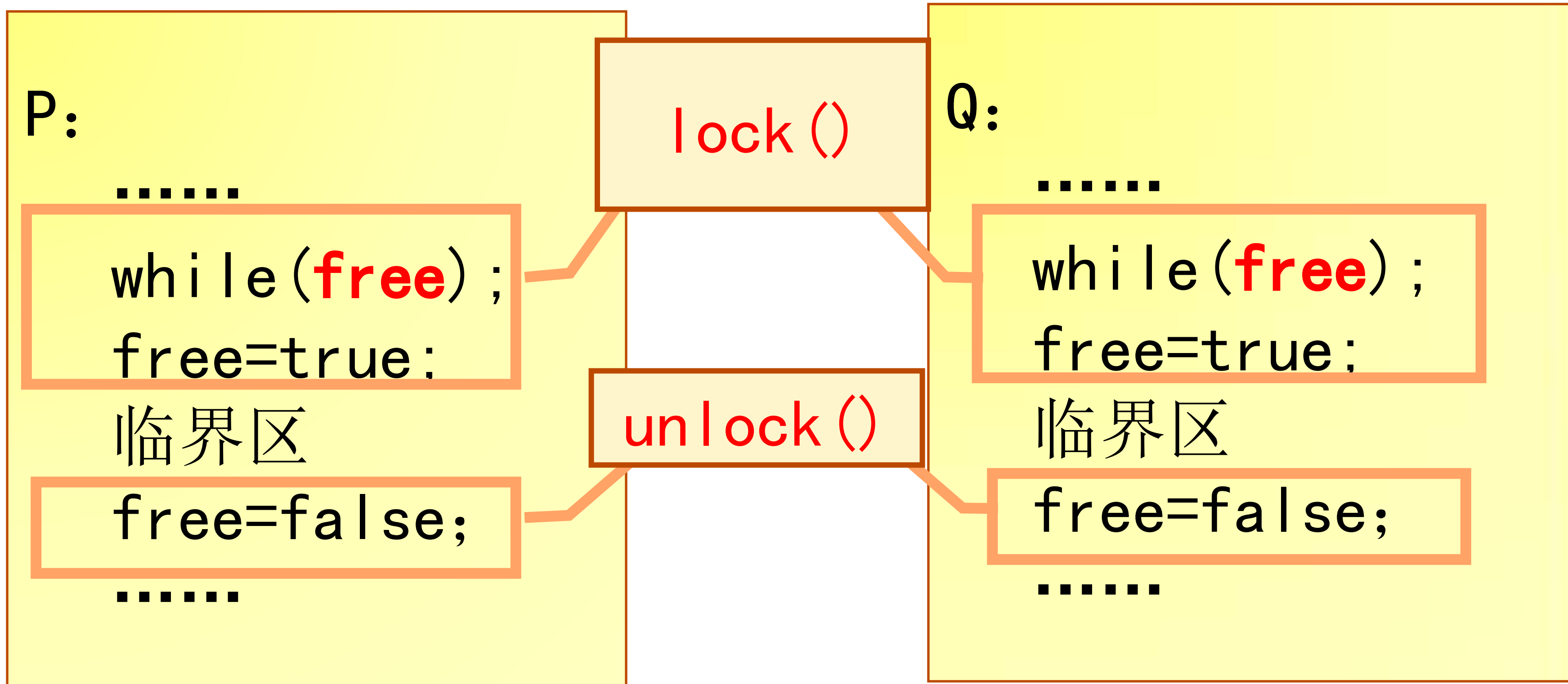
软件方法1

- free:临界区空闲标志

true:有进程在临界区；false:无进程在临界区

- 初值：free为false

改进方法:设置原语



软件方法2

- turn：谁进临界区的标志

true:P进程进临界区；false:Q进程进临界区

- 初值任意

```
P:
    ......
    while(not turn);
    临界区
    turn=false;
    ......
```

```
Q:
    ......
    while(turn);
    临界区
    turn=ture;
    ......
```

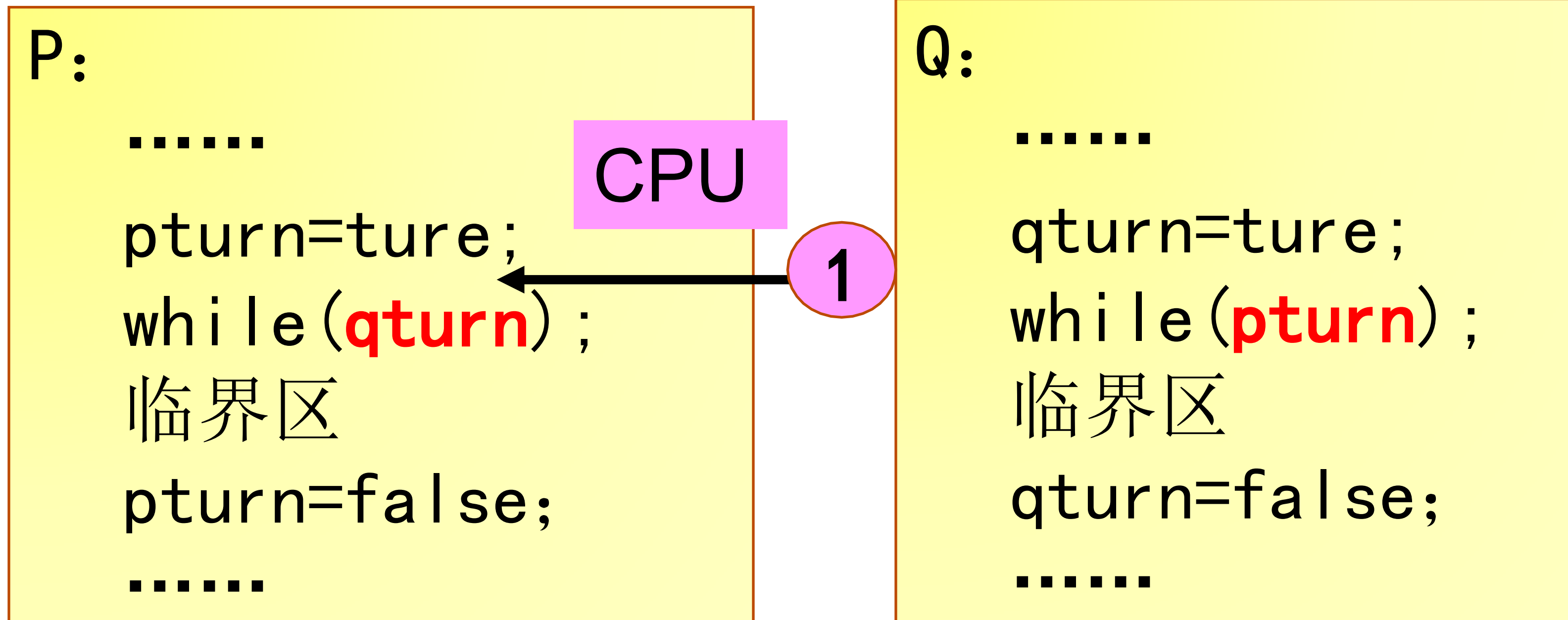
若P想进临界区，由于turn=false；进不了；

同时Q进程始终不准备进临界区，即使临界区一直没有进程，但P一直无法进入临界区

该方法，违反了使用临界区的原则

软件方法3

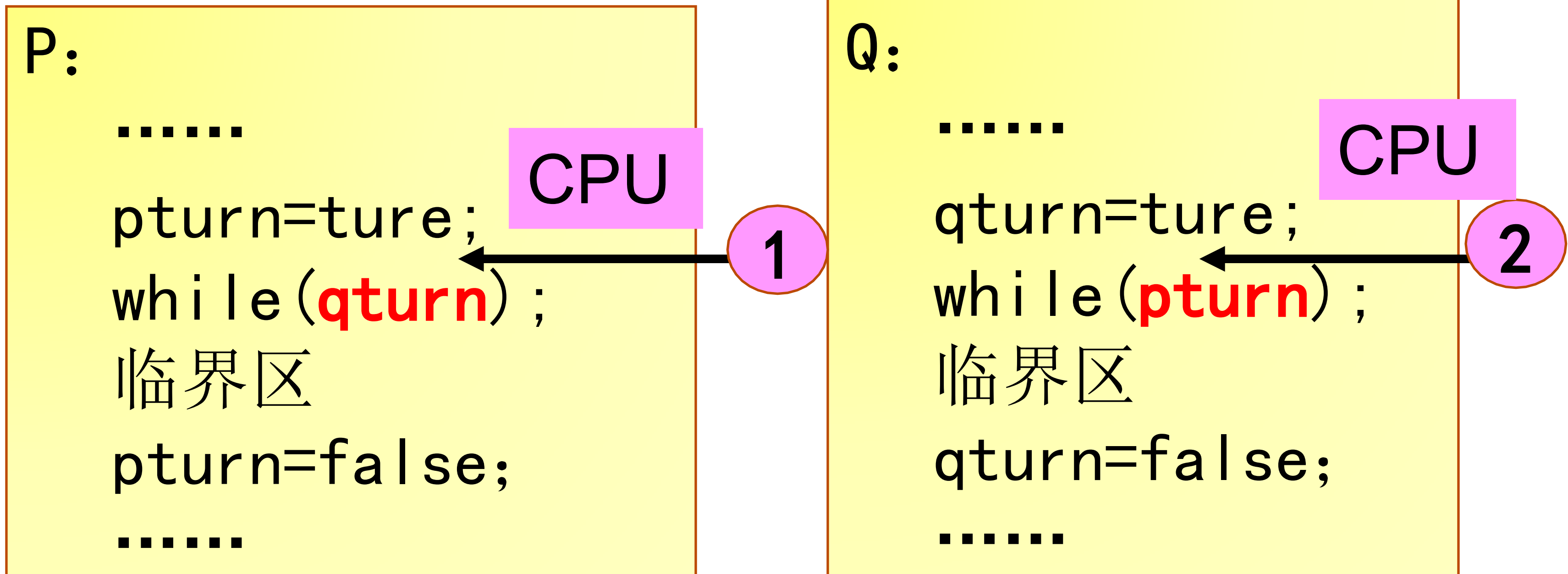
- pturn,qturn的初值为false
- P进入临界区的条件：pturn ∧ not qturn（与）
- Q进入临界区的条件：not pturn ∧ qturn （与）



Step1: p执行到pturn=ture，被撤下CPU

软件方法3

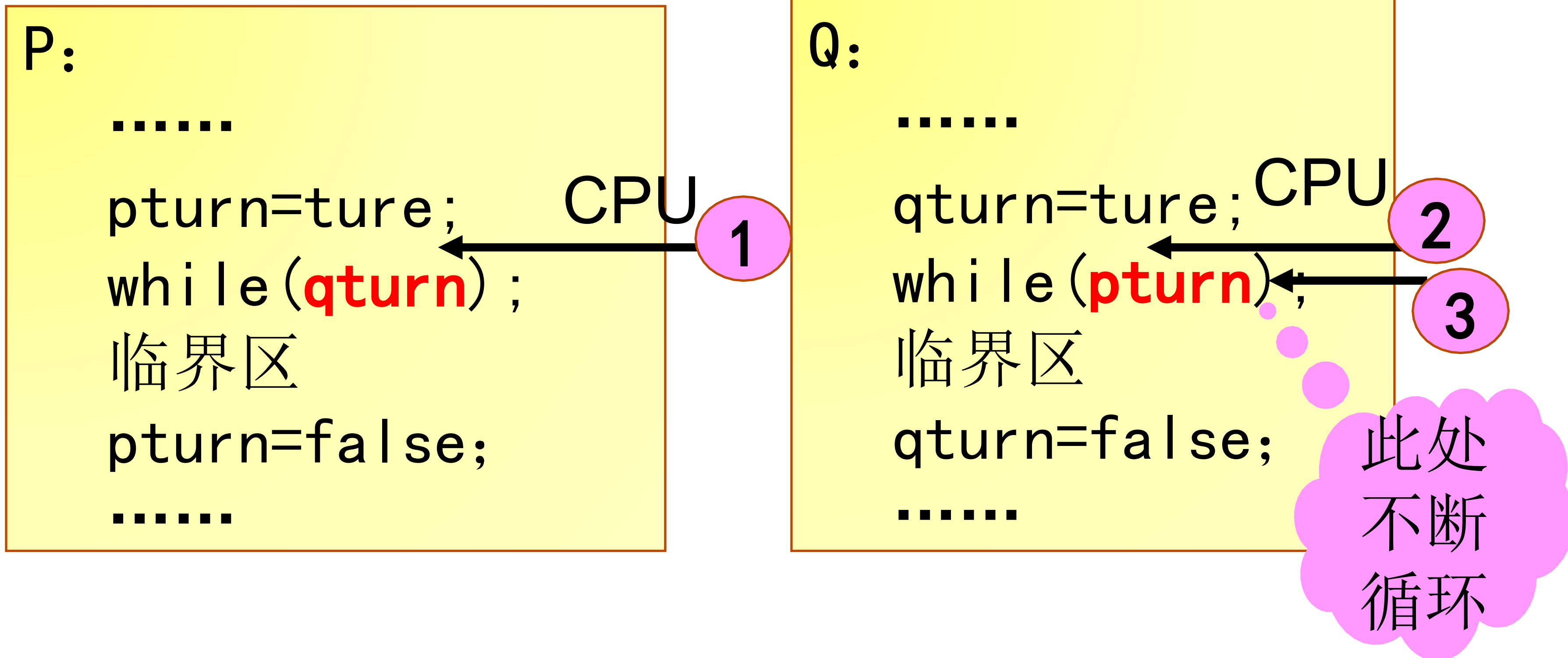
- pturn,qturn的初值为false
- P进入临界区的条件：pturn^ not qturn
- Q进入临界区的条件：not pturn^qturn



Step2: Q进程上CPU执行到qturn=ture

软件方法3

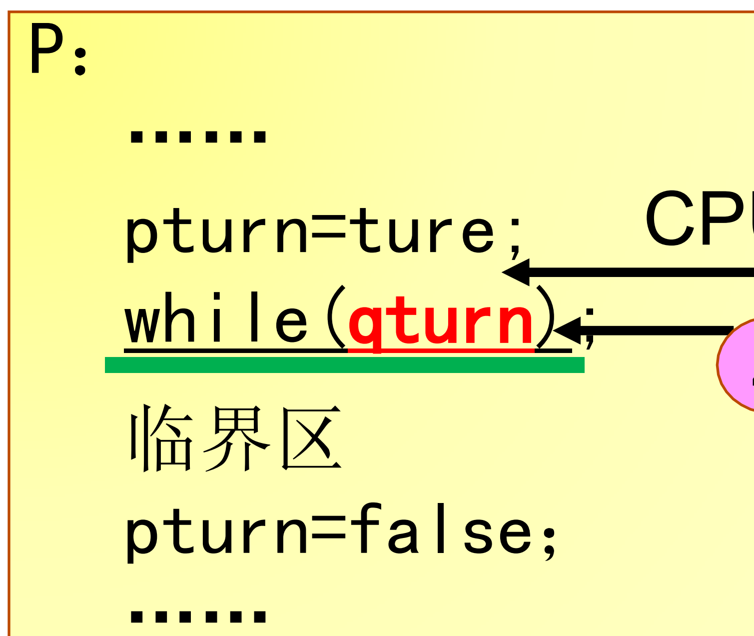
- pturn,qturn的初值为false
- P进入临界区的条件：pturn^ not qturn
- Q进入临界区的条件：not pturn^qturn

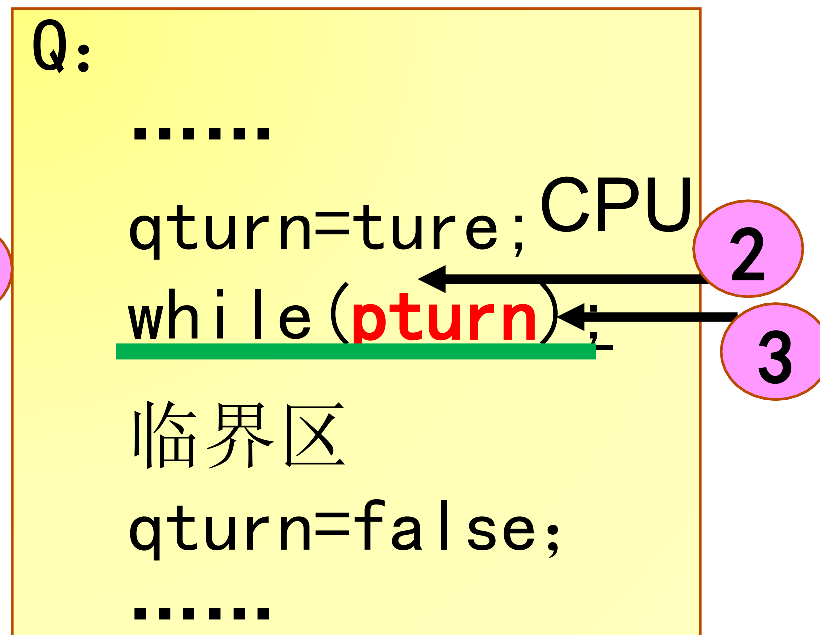


Step3: Q进程上一直测试pturn，<u>无法通过，始终不能进入临界区</u>

软件方法3

- pturn,qturn的初值为false
- P进入临界区的条件：pturn^ not qturn
- Q进入临界区的条件：not pturn^qturn





Step3: 时间片结束，Q进程被撤下CPU，P进程上CPU，一直测试qturn，无法通过，始终不能进入临界区

- P进程和Q进程都无法进入空闲着的临界区（该方法不满足临界区的使用原则）

软件方法4 dekker 算法 对解法3的改进

1965年第一个 用软件的方法解决了临界区保护问题

引入turn变量：turn=1 P上cpu；turn=2 Q上cpu

```
P: ……
    pturn=ture;
    while(qturn){
        if(turn==2){
            pturn=false;
            while(turn==2);
            pturn=turn;
        }
    }
    临界区
    turn=2;
    pturn=false;
    ……
```

```
Q: ……
    qturn=ture;
    while(pturn){
        if(turn==2){
            qturn=false;
            while(turn==1);
            qturn=turn;
        }
    }
    临界区
    turn=1;
    qturn=false;
    ……
```

P:

```
    ......
    pturn=ture;        CPU ← 1
    while(qturn){
        if(turn==2){
            pturn=false;
            while(turn==2);
            pturn=turn;
        }
    }
    临界区
    turn=2;
    pturn=false;
    ......
```

Q:

```
    ......
    qturn=ture;        CPU ← 2
    while(pturn){
        if(turn==1){
            qturn=false;
            while(turn==1);
            qturn=turn;
        }
    }
    临界区
    turn=2;
    qturn=false;
    ......
```

Step1: P上CPU

Step2：P被撤下CPU，Q上CPU

P:

```
    ……
    pturn=ture;
    while(qturn){
        if(turn==2){
            pturn=false;
            while(turn==2);
            pturn=turn;
        }
    }
    临界区
    turn=2;
    pturn=false;
    ……
```

Q:

```
    ……
    qturn=ture;
    while(pturn){
        if(turn==1){
            qturn=false;
            while(turn==1);
            qturn=turn;
        }
    }
    临界区
    turn=2;
    qturn=false;
    ……
```


此时pturn=ture， qturn=true; 根据turn的值决定谁进CPU

```
P:
    ......
    pturn=ture;      ← CPU ①
    while(qturn){
        if(turn==2){
            pturn=false;
            while(turn==2);
            pturn=turn;
        }
    }
    临界区
    turn=2;
    pturn=false;
    ......
```

```
Q:
    ......
    qturn=ture;      ← CPU ②
    while(pturn){
        if(turn==1){                ③
            qturn=false;
            while(turn==1);         ④
            qturn=turn;
        }
    }
    临界区
    turn=2;
    qturn=false;
    ......
```

Step3:根据turn的值，判断谁进临界区，Q将自己的qturn设为false，让P进入临界区，并用while一直查询turn是否=1，即**轮到Q进入临界区**

软件方法5 peterson算法

```
进程i:
...
enter_region(i);//进程想进临界区调用该函数，看是否能
                //安全进入，i是进程号;
临界区;
lever_region（i） ;//进程离开临界区调用该函数,
```

Peterson算法解决了互斥访问的问题，而且克服了（Dekker算法）强制轮流法的缺点，可以完全正常的工作

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/397016052146006200