第3章 同步、通信与死锁

知识要点

- 掌握
  - 程序的顺序执行与并发执行；
  - 进程互斥：临界区、临界资源、临界区管理的实现方法；
  - 进程同步：采用同步机制（信号量）解决同步问题；
  - 通信机制分类和实现原理；
  - 死锁：定义、引发原因、死锁必要条件、死锁处理方法。
- 了解
  - 管程：概念、特性、结构、条件变量和实现

3.1并发进程

- 顺序程序设计
- 并发程序设计
- 进程的交互：协作和竞争

进程的顺序性

- 计算机的信息处理
  - 处理机逐条的一次只执行一条指令
  - 主存储块一次只访问一个字或字节
  - 外设一次只能传送一个数据块
- 程序的顺序执行：一个程序由若干个程序段组成，而程序段在顺序处理器上的执行是严格按序的，只有当前一个操作结束后，后继操作才能开始。
- 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：把一个程序设计成一个顺序执行的程序模块。

顺序程序设计特点

- 程序执行的顺序性：处理机严格按照程序所规定的顺序执行，即每个操作必须在下一个操作开始之前结束。
- 程序环境的封闭性：程序一旦开始执行，其计算结果不受外界的影响，当程序的初始条件给定之后，其后的状态只能由程序本身确定，即只有本程序才能改变它。
- 执行结果的确定性：程序执行结果与它的执行速度无关。
- 计算过程的可再现性：程序执行的结果与初始条件有关，而与执行时间无关。即只要程序的初始条件相同，不论什么时间执行，它的执行结果是相同的。

进程的并发性(1)

- 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，可交替进行。
- 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。

进程的并发性(2)

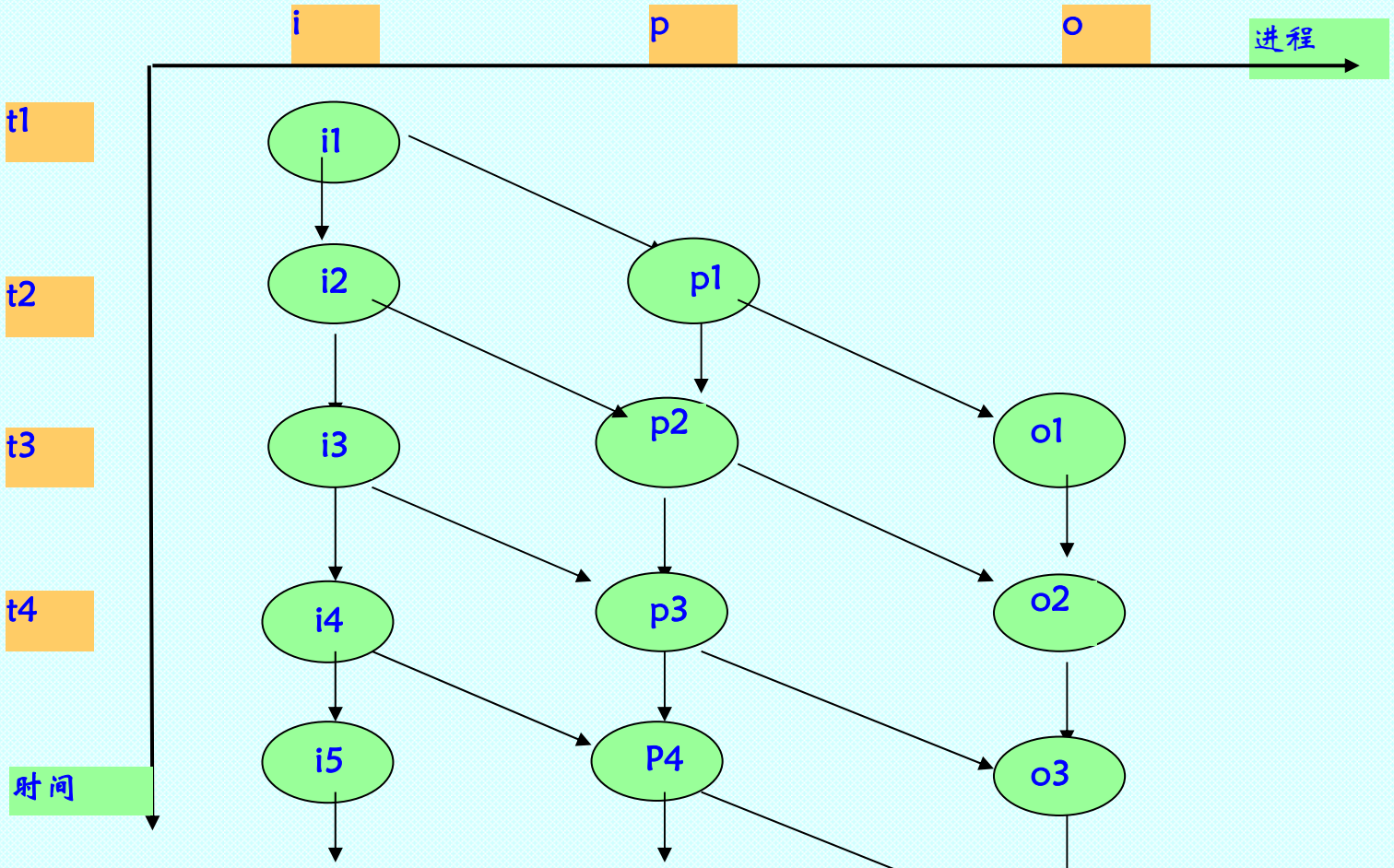
程序并发执行

- 小程序 1：循环执行，“读入”数据块，将数据送入缓冲区 1；
- 小程序 2：循环执行，“加工”缓冲区 1 中的数据，把计算结果送缓冲区 2；
- 小程序 3：循环执行，“输出”缓冲区 2 中的计算结果，让数据打印输出。

进程的并发性(3)
观察程序并发执行现象

- (1) 小程序操作数据块有先后次序，按数据块序号从小到大处理。
- (2) 小程序可以并发执行的，例如，“输入”i 3、“加工”p 2 与“输出”o 1 可并行操作。这样 CPU、输入设备和输出设备可并行工作。
- (3) 小程序并发执行时，相互之间会产生制约关系，原因是小程序1与 2 共享资源--缓冲区1，小程序2与 3共享资源--缓冲区2，出现了竞争共享资源问题。

进程的并发性(4)

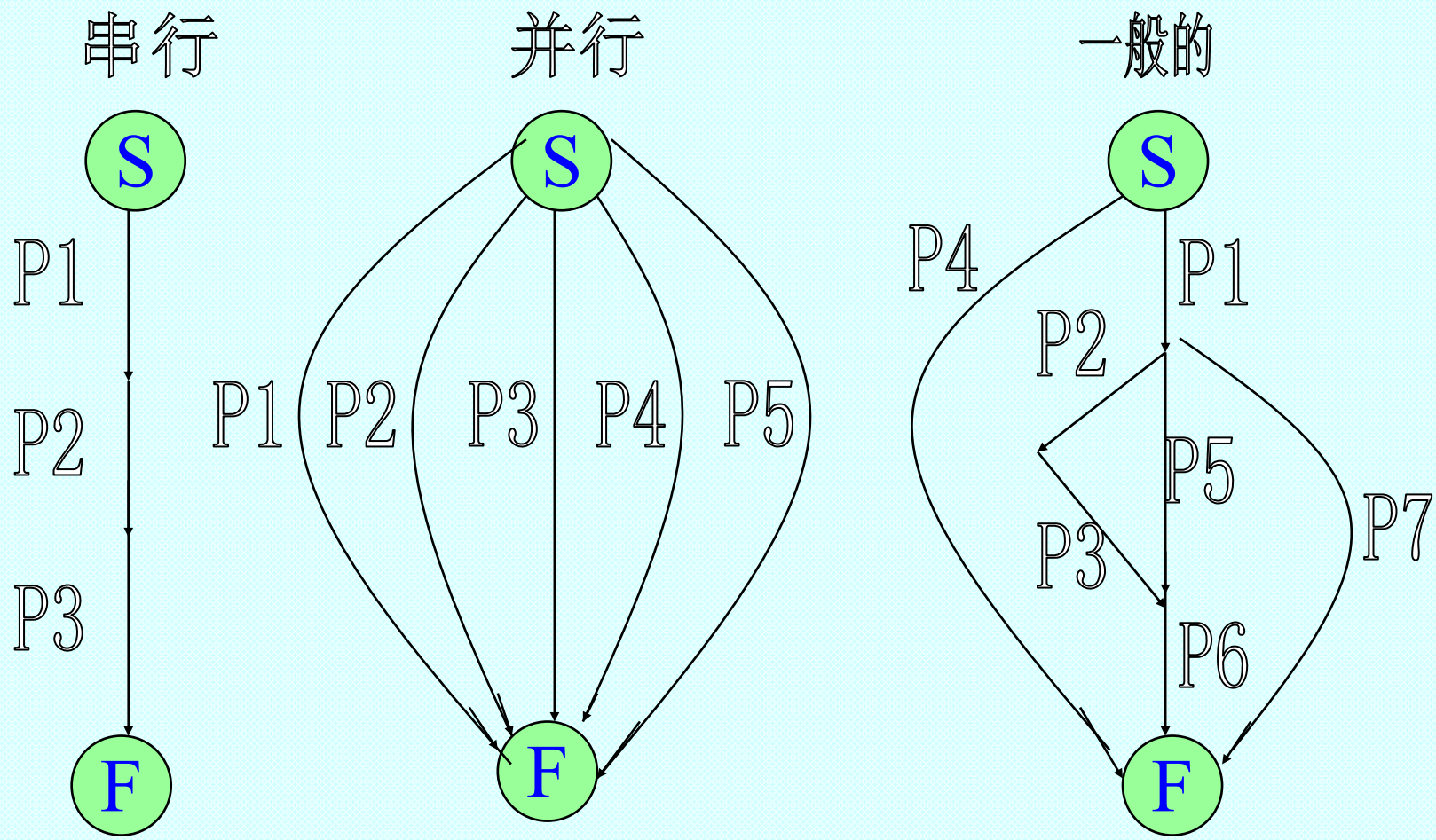


并发工作

进程的并发性(4)

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

程序并行性的表示之一：有向图



程序并行性的表示之二：并行语言

- 并行语言：并行PASCAL，CSP/K语言，MODULA语言，扩充的Ada等.
- 并行语句记号:

  cobegin

  S1;S2;S3;...;SN

  coend;

  - Si(i=1,2,3,...,n)表示n个语句（程序段），这n个语句用cobegin和coend括起来表示这n个语句是可以并发执行的。co是concurrent的头两个字符。
  - 有的书用parbegin和parend表示。
  - **Si:** 简单语句，复合语句，并行语句。
  - 编译程序为每个并行语句Si设置一个进程。

程序并行性表示举例

假设有一个程序由S_0～S_{n+1}个语句，

其中 S_1～S_n语句是并发执行的。

程序如下：

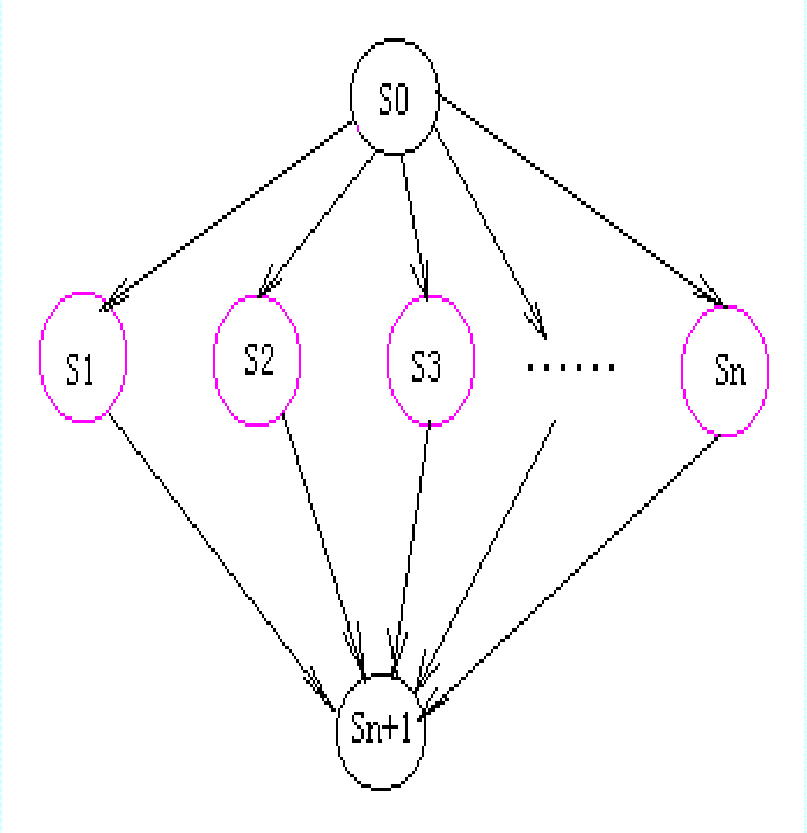
S_0;

cobegin

S_1;S_2;S_3;...;S_N

coend;

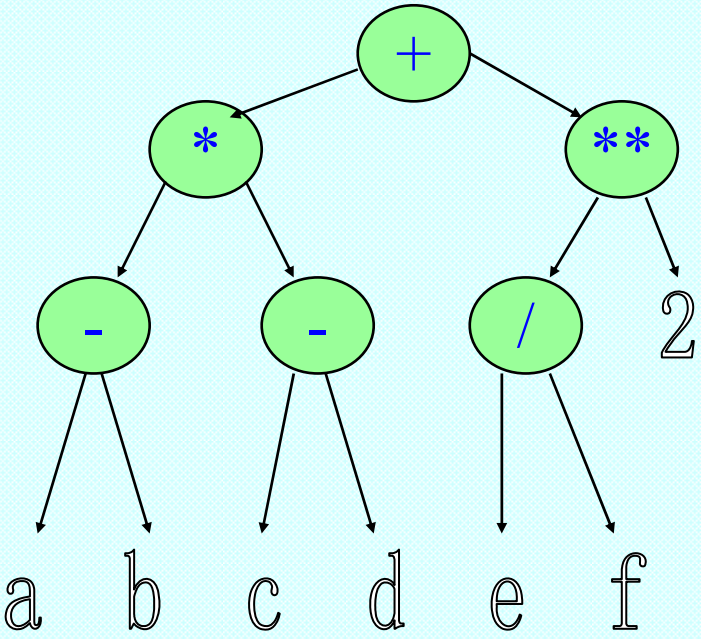
S_{n+1};



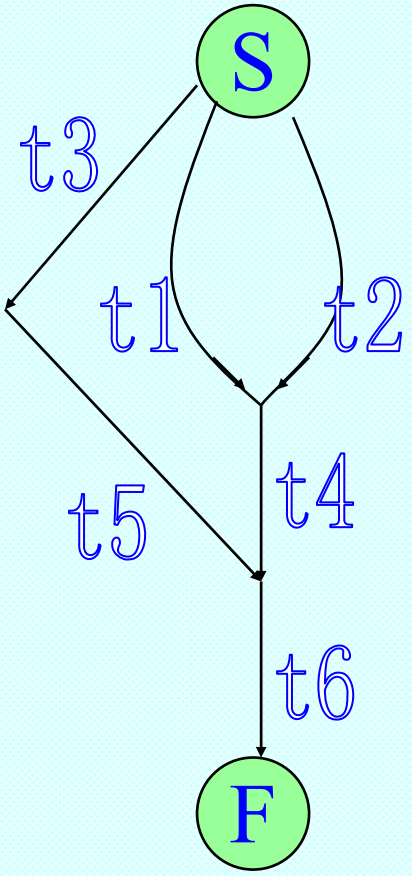
程序并行性表示举例

算数表达式求值:

(a-b)*(c-d)+(e/f)2**




```
BEGIN
COBEGIN
t1=a-b
t2=c-d
t3=e/f
COEND
COBEGIN
t4=t1*t2
t5=t3**2
COEND
t6=t4+t5
END
```



无关的并发进程

- 并发进程分类
  - 无关的并发进程
  - 交往的并发进程
- 无关的并发进程：一组并发进程分别在不同的变量集合上操作，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。

Bernstein条件

- R(pi)={a1,a2,…an}，程序pi在执行期间引用的变量集；
- W(pi)={b1,b2,…bm}，程序pi在执行期间改变的变量集；
- 若两个程序的变量集交集之和为空集：R(p1)∩W(p2)∪ R(p2)∩W(p1)∪ W(p1)∩W(p2)={ }；
- 则并发进程的执行与时间无关。

Bernstein条件举例

- 例如，有如下四条语句：

  S1: a := x + y　S2: b := z + 1

  S3: c := a – b　S4: z := c + a

- 于是有：R(S1)={x,y} ,R(S2)={z}，

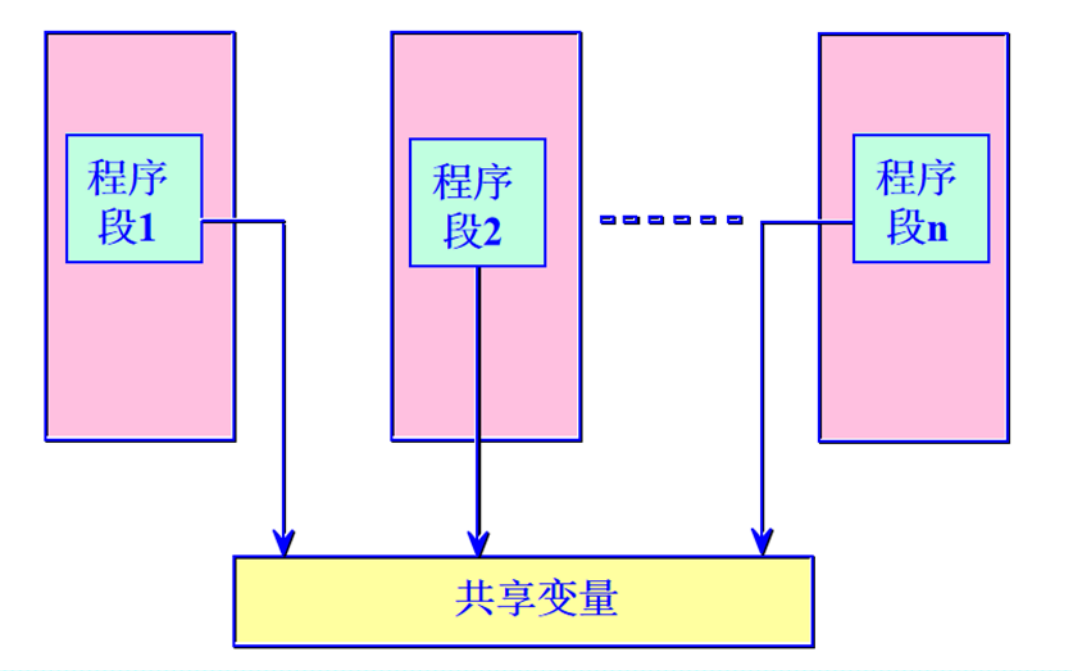
  R(S3)={a,b}，R(S4)={c,a}；W(S1)={a}，

  W(S2)={b}，W(S3)={c}，W(S4)={z}。

- 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。

交往的并发进程

- 交往的并发进程：一组并发进程共享某些变量，一个进程的执行可能影响其他并发进程的结果。



并发程序设计的优点

- 对于单处理器系统，可让处理器和各I/O设备同时工作，发挥硬部件的并行能力。
- 对于多处理器系统，可让各进程在不同处理器上物理地并行，加快计算速度。
- 简化程序设计任务。

采用并发程序设计的目的

- 充分发挥硬件的并行性，提高系统效率。硬件能并行工作仅有了提高效率的可能性，硬部件并行性的实现需要软件技术去利用和发挥，这种软件技术就是并发程序设计。
- 并发程序设计是多道程序设计的基础，多道程序的实质就是把并发程序设计引入到系统中。

与时间有关的错误

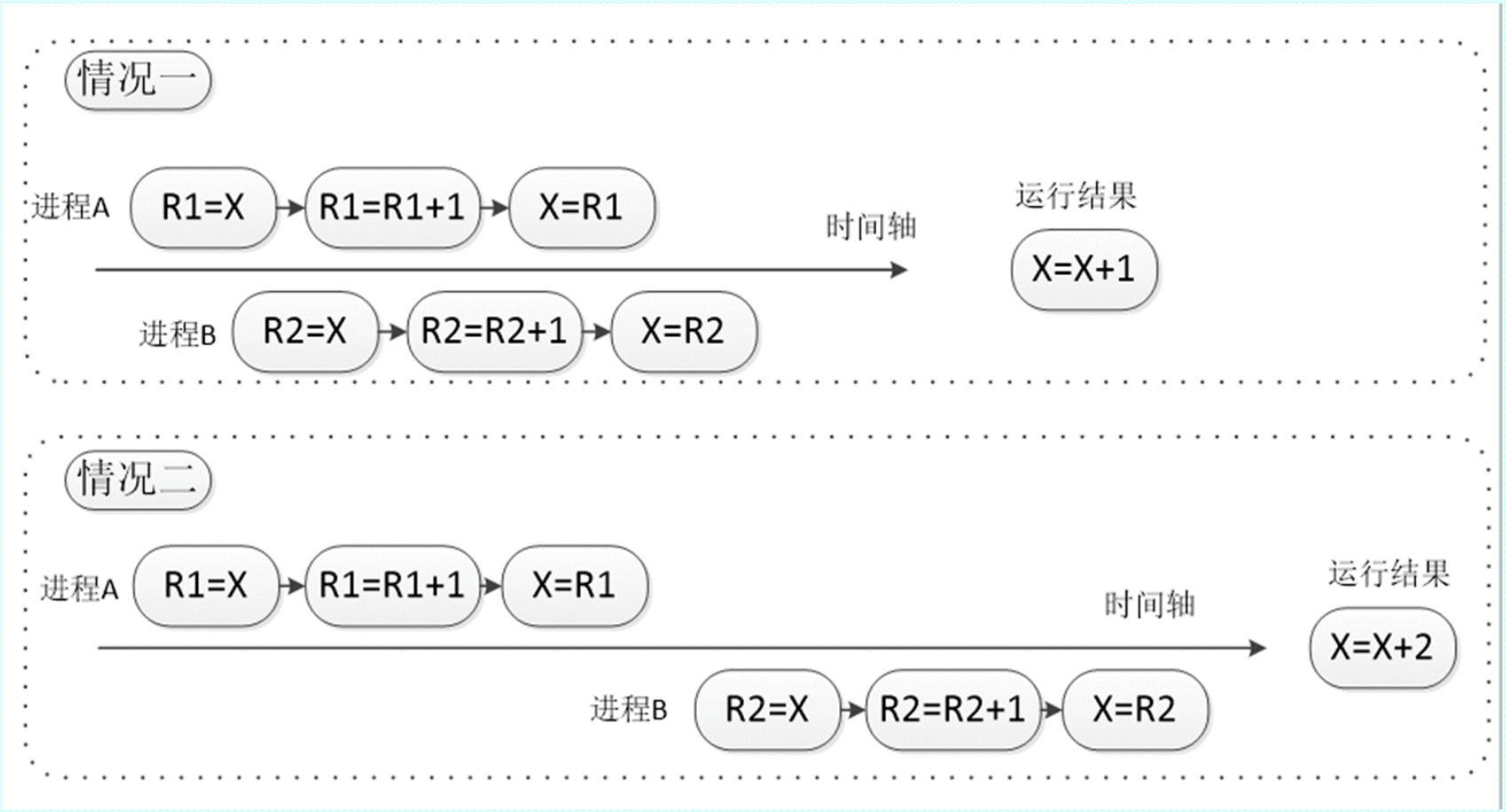
- 对于一组交往的并发进程，执行的相对速度无法相互控制，各种与时间有关的错误就可能出现。
- 与时间有关错误有两种表现形式：
  - 结果不唯一
  - 永远等待

（结果不唯一）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 "*/;
}
```

（结果不唯一）飞机票售票问题



（永远等待）内存资源管理问题

申请和归还内存资源问题

int X=memory;　　//memory为初始内存容量

```
void borrow(int B) {
   if(B>X)
   {/*进程进入等待内存资源队列*/; }
   X=X-B ;
  /*修改内存分配表，进程获得内存资源*/;
}
```

```
void return(int B) {
   X=X+B;
    /*修改内存分配表*/;
    /*释放等内存资源进程*/;
}
```

进程的交往：竞争与协作(1)

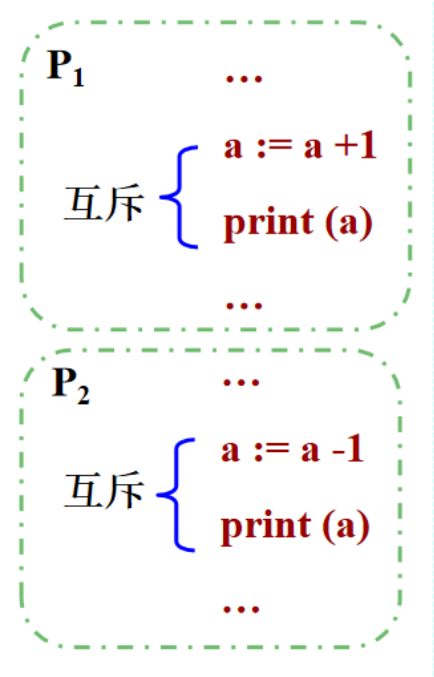
第一种是竞争关系

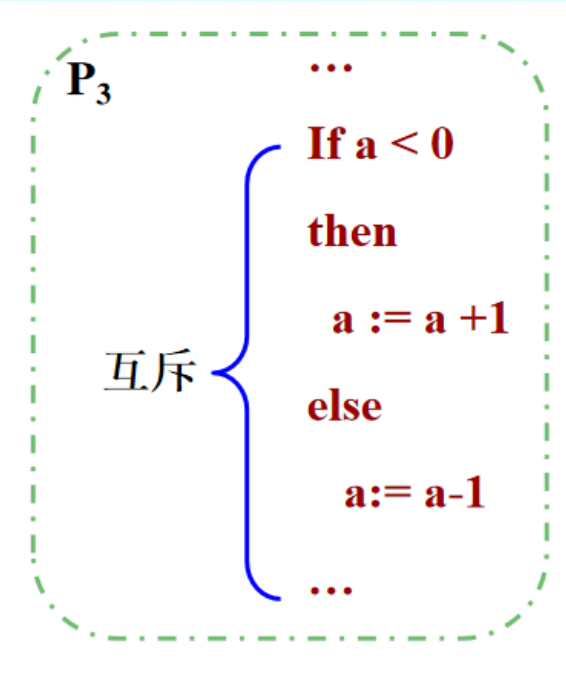
- 竞争关系： 指原本不存在逻辑关系的诸进程因共享资源而产生的交互和制约关系。这是一种间接制约，又称互斥关系。
- 资源竞争的两个控制问题：
  - 死锁(Deadlock)问题：进程因争夺资源陷入永久等待的状态；
  - 饥饿(Starvation) 问题：进程对资源的使用被无限期拖延或超过等待时间的上届。
- 既要解决饥饿问题，又要解决死锁问题。

进程的交往：竞争与协作(2)

进程互斥(Mutual Exclusion)

- 进程互斥：指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。





进程的交往：竞争与协作(3)

第二种是协作关系

- 协作关系：指某些进程为完成同一任务需要分工协作而产生的关系。
- 进程同步：指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。
- 如果一个进程的执行依赖于协作进程的消息或信号，当该进程没有得到来自于协作进程的消息或信号时需等待，直到消息或信号到达才被唤醒。

进程的交往：竞争与协作(4)

- 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。

3.2 临界区管理

- 互斥与临界区
- 实现临界区管理的几种尝试
- 实现临界区管理的软件方法
- 实现临界区管理的硬件设施

3.2.1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。

互斥与临界区(2)

- 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。
- 临界区调度原则：
  - 互斥使用、有空让进，
  - 忙则等待、有限等待，
  - 择一而入、算法可行。

互斥与临界区(2)

实现各进程互斥进入临界区

进程须在临界区前面增加一段用于进行上述检查的代码，称为进入区(entry section)。在临界区后面加上一段称为退出区(exit section)的代码

```
While (1)
{
    进入区代码;
    临界区代码;
    退出区代码;
    其余代码 ;
}
```

进入区

临界区

退出区

剩余区

3.2.2临界区管理的尝试(1)

```
bool inside1=false;      /*P1不在其临界区内*/
bool inside2=false;      /*P2不在其临界区内*/
cobegin
process P1( ) {                    process P2( ) {
   while(inside2);/*等待*/            while(inside1);  /*等待*/
   inside1=true;                      inside2=true;
   /*临界区*/                         /*临界区*/;
  inside1=false;                     inside2=false;
 }                                  }
coend
```

双标志、先检查，后表态。　　违背互斥性原则

临界区管理的尝试(2)

```
bool inside1=false; //P1不在其临界区内
bool inside2=false; //P2不在其临界区内
cobegin
process P1( ) {
    inside1=true;
    while(inside2);//等待
    {临界区};
    inside1=false;
}
process P2( ) {
    inside2=true;
    while(inside1);//等待
    {临界区};
    inside2=false;
}
coend
```

双标志，先表态，后查看　　违背有空让进

3.2.3 实现临界区的软件算法

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

Peterson算法

```
cobegin
process P0( ) {
  inside[0]=true;
  turn=1;
  while(inside[1]&&turn==1);
    /*临界区*/;
    inside[0]=false;
 }
process P1( ) {
   inside[1]=true;
   turn=0;
    while(inside[0]&&turn==0);
    {临界区};
     inside[1]=false;
 }
coend
```

实现临界区管理的硬件设施

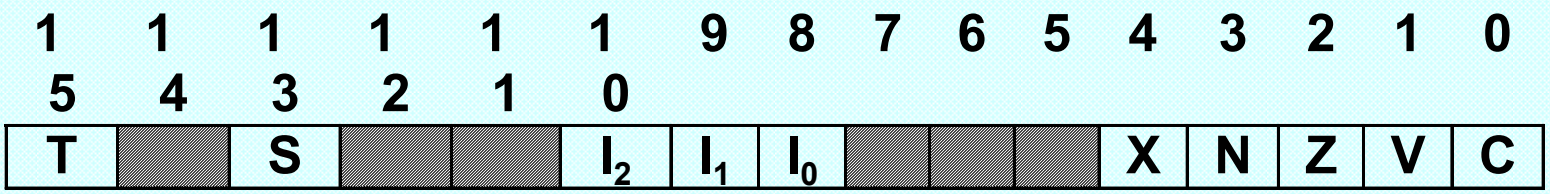
- 关中断
- 硬件指令
  - 测试并设置指令
  - 对换指令

关中断

- 禁止一切中断发生。
- 单CPU中，引起进程切换的唯一原因是中断，故单CPU下可行。
- 缺点：
  - 代价高，影响并发性
  - 不安全，将禁止一切中断权利给了普通用户。
  - 局限性：不适合多CPU，一个进程只能禁止本CPU的中断。

关中断

```
While (1)
{
    屏蔽中断响应;
    临界区代码;
    恢复中断响应;
    其余代码 ;
}
```

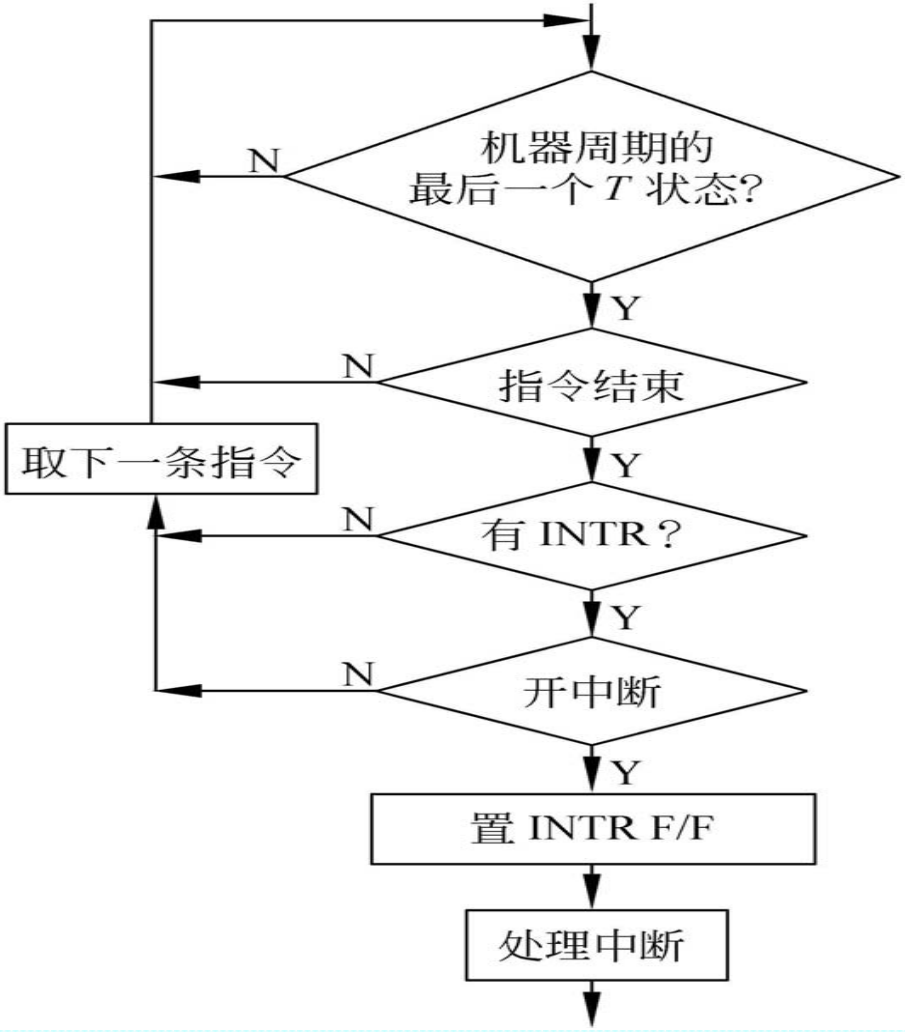


微处理器M68000的程序状态字

I0 – I2：三位中断屏蔽位

硬件指令方法

- 思路：一条机器硬件指令完成读写两个操作。
- 手段：执行硬件指令的CPU封锁内存总线，以禁止其他CPU在该指令完成前访问内存。
- 硬件指令（微指令）的执行过程中不响应中断。CPU在微指令结束时才去检测是否有中断信号。



测试并设置指令(1)

- TS指令的处理过程

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

- TS指令管理临界区时，可把一个临区与一个布尔变量s相连，由于变量s代表了临界资源的状态，可把它看成一把锁。

测试并设置指令(2)

- /*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
```

对换指令(1)

- SWAP指令的处理过程

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

对换指令(2)

- /*对换指令实现进程互斥*/

```
bool lock=false;
cobegin
Process Pi( ){  //i=1,2,...,n
    bool keyi=true;8
      do {
        SWAP(keyi,lock);
      }while(keyi);                /*上锁*/
      /*临界区*/;
      SWAP(keyi,lock);      /*开锁*/
}
coend
```

硬件指令方法的缺点

- 忙等待：上述硬件指令虽然可以有效的保证进程间互斥，但是进程在临界段中执行时，其他想进入临界段的进程必须不断地检测布尔变量lock的值，这就造成了处理机时的浪费，通常称这种情况为“忙等待”。
- 饥饿：由于采用随机从等待队列中选取进程，会出现有的进程一直处于等待。
- 需CPU支持。

硬件指令方法的优点

- 不但适用于单处理器情况，而且适用于共享主存的SMP多处理器情况（即对称多处理器）；
- 方法简单，行而有效；
- 可以被使用于多重临界段情况，每个临界段可以定义自己的共享变量。

3.3 信号量与PV操作

- 同步和同步机制
- 信号量与PV操作
- 信号量实现互斥
- 信号量解决五个哲学家就餐问题
- 信号量解决生产者-消费者问题
- 记录型信号量解决读者-写者问题
- 记录型信号量解决睡眠理发师问题

3.3.1 同步和同步机制

- 著名的生产者--消费者问题是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。
- 在操作系统中，生产者进程可以是计算进程、发送进程；而消费者进程可以是打印进程、接收进程等等。
- 解决好生产者--消费者问题就解决好了一类并发进程的同步问题。

生产者--消费者问题表述

- 有界缓冲问题
- 有n个生产者和m个消费者，连接在一个有k个单位缓冲区的有界缓冲上。其中，pi和cj都是并发进程，只要缓冲区未满，生产者pi生产的产品就可投入缓冲区；只要缓冲区不空，消费者进程cj就可从缓冲区取走并消耗产品。

生产者-消费者问题算法描述(1)

- int k;
- typedef anyitem item; /*item类型*/
- item buffer[k];
- int in=0,out=0,counter=0;

生产者-消费者问题算法描述(2)

```
process producer(void) {
    while (true) {              /*无限循环*/
        {produce an item in nextp};/*生产一个产品*/
        if (counter==k)   /*缓冲满时，生产者睡眠*/
            sleep(producer);
        buffer[in]=nextp; /*一个产品放入缓冲区*/
        in=(in+1)%k;      /*指针推进*/
        counter++;        /*缓冲内产品数加1*/
        if(counter==1)    /*缓冲为空，加进一件产品*/
            wakeup(consumer); /*并唤醒消费者*/
    }
}
```

生产者-消费者问题算法描述(3)

```
process consumer(void) {
  while (true) {            /*无限循环*/
    if (counter==0)         /*缓冲区空，消费者睡眠*/
      sleep(consumer);
    nextc=buffer[out];/*取一个产品到nextc */
    out=(out+1)%k;  /*指针推进*/
    counter--;              /*取走一个产品，计数减1 */
    if(counter==k-1)  /*缓冲满了，取走一件产品并唤*/
      wakeup(producer);  /*醒生产者*/
    {consume the item in nextc};/*消耗产品*/
  }
}
```

生产者-消费者问题的算法描述(4)

- 生产者和消费者进程对counter的交替执行会使其结果不唯一
- 生产者和消费者进程的交替执行会导致进程永远等待

3.3.2信号量与PV操作(1)

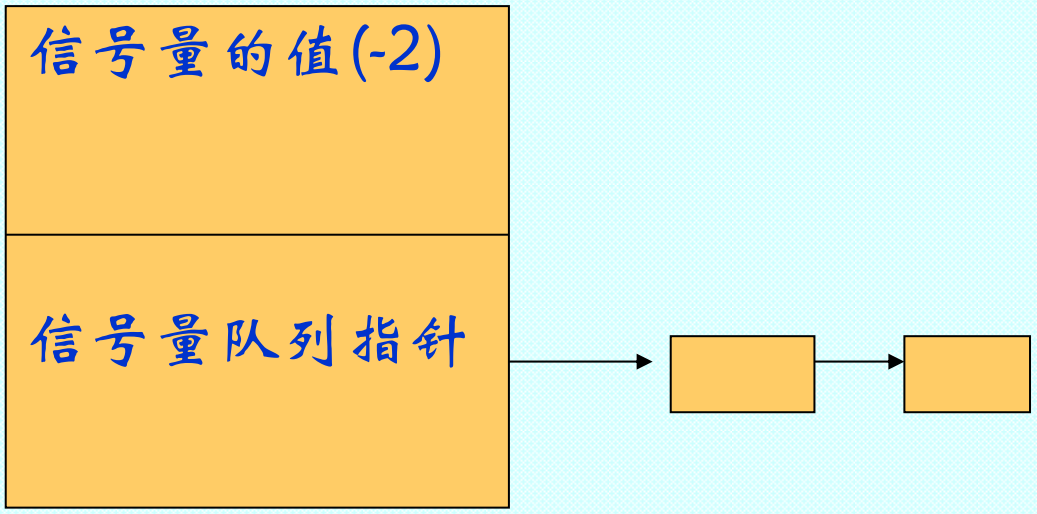
- 前面方法解决临界区调度问题的缺点:
  - 对不能进入临界区的进程，采用忙式等待测试法，浪费CPU时间。
  - 将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重用户编程负担。
- 1965年E.W.Dijkstra提出了新的同步工具--信号量和P、V操作。

信号量与PV操作(2)

- 信号量：一种软件资源，除初始化外，仅能由两个同步原语对其进行操作的整型变量。
- 原语：内核中执行时不可被中断的过程；
  - P操作原语
  - V操作原语
- 一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(semaphore)，复杂的进程合作需求都可以通过适当的信号结构得到满足。

信号量与PV操作(3)

- 操作系统中，信号量表示物理资源的实体，它是一个与队列有关的整型变量。
- 实现时，信号量是一种记录型数据结构，有两个分量：一个是信号量的值，另一个是信号量队列的队列指针。



信号量分类

- 信号量按其取值分为
  - 二元信号量：信号量的值仅允许取0或1，主要用于互斥变量。
  - 一般信号量：信号量取值允许为非负整数，主要用于进程间的一般同步。

一般信号量(1)

- 设s为一个记录型数据结构，一个分量为整形量value，另一个为信号量队列queue，P和V操作原语定义：
  - P(s)：将信号量s减去1，若结果小于0，则调用P(s)的进程被置成等待信号量s的状态。
  - V(s)：将信号量s加1，若结果不大于0，则释放一个等待信号量s的进程。

一般信号量(2)

```
typedef struct semaphore {
        int value;              /*信号量值*/
        struct pcb *list;    /*信号量队列指针*/
};
void P(semaphore &s) {
         s.value--;
         if(s.value<0)  sleep(s.list);
}
void V(semaphore &s) {
        s.value++;
        if(s.value<=0)    wakeup(s.list);
}
```

二元信号量(1)

- 设s为一个记录型数据结构，一个分量为value，它仅能取值0和1，另一个分量为信号量队列queue，把二元信号量上的P、V操作记为BP和BV，BP和BV操作原语的定义如下：

二元信号量(2)

```
typedef struct binary_semaphore {
  int value;                    /*value取值0 or 1*/
  struct pcb *list;  };
void BP(binary_semaphore &s) {
        if(s.value==1)
             s.value=0;
        else
            sleep(s.list);
}
void BV(binary_semaphore &s) {
        if(s.list is empty( ))
             s.value=1;
        else
            wakeup(s.list);
}
```

信号量的物理意义

- 信号量S的初值表示可用资源数
- 当S>0时，S的值表示还剩可用资源数
- 当S<=0时，表示已无资源可分配，其绝对值表示此时在等待队列中等待分配资源的进程数
- Wait操作：申请资源，若S>0，意味着有资源可以申请，操作后，S=S-1意味着资源减少
- Signal操作：释放资源，执行Signal操作之后，S=S+1,意味着资源数增加

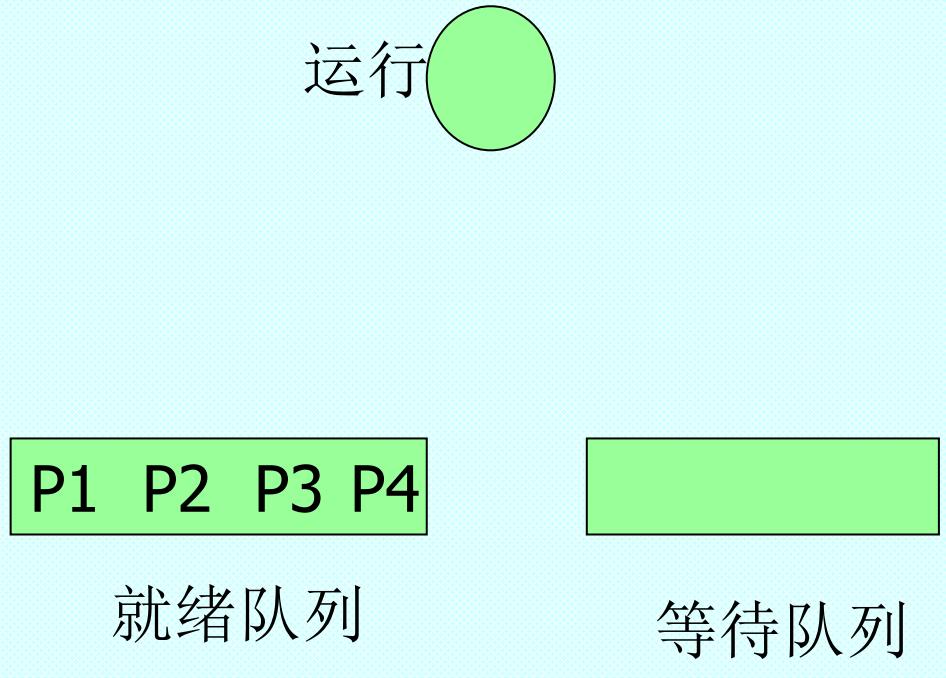
信号量的物理意义

- 信号量的变化范围：

  设可用资源数为m，进程数为n

  - 阻塞等待： -(n-m)<=S <=m

例：4个进程共享2台打印机

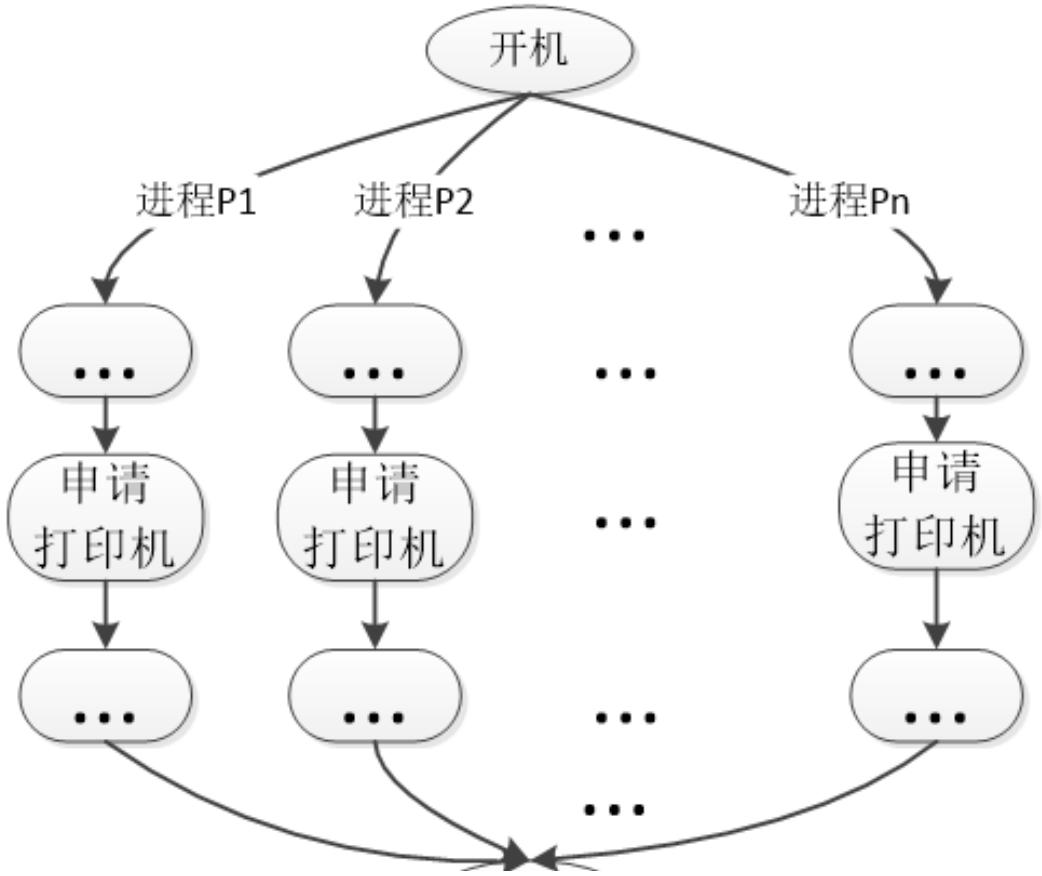


S: 2	
1	P1:wait(s)
0	p2:wait(s)
-1	p3:wait(s)
-2	p4:wait(s)
-1	p1:signal(s)
0	p2:signal(s)
1	p3:signal(s)
2	p4:signal(s)

3.3.3信号量实现互斥

- 描述：多个进程共享临界资源，并且对资源的访问是互斥的，

进程P1、P2、P3、。。。、Pn关于“打印机”互斥



信号量实现互斥

用信号量及P、V操作来描述上图

1、说明进程的互斥关系

进程P1、P2、。。。。、Pn可并行执行，他们的运行时序时随机的、异步的。但对于X的操作只能互斥、排他的进行。

2、设置信号灯，说明含义、初值。

①当s = m 表示系统有m个临界资源(打印机)可用(空闲状态)；

②当s <=0 表示系统没有临界资源(打印机)可用（占用状态）；

此时s的绝对值表示由于申请临界资源(打印机)而处于等待状态的进程数。

信号量实现互斥

进程P1
```
P1()
{
    …
  P(s);
申请打印机;
  V(s);
   …
}
```

进程P2
```
P2()
{
    …
  P(s);
申请打印机;
  V(s);
   …
}
```

。 。 。

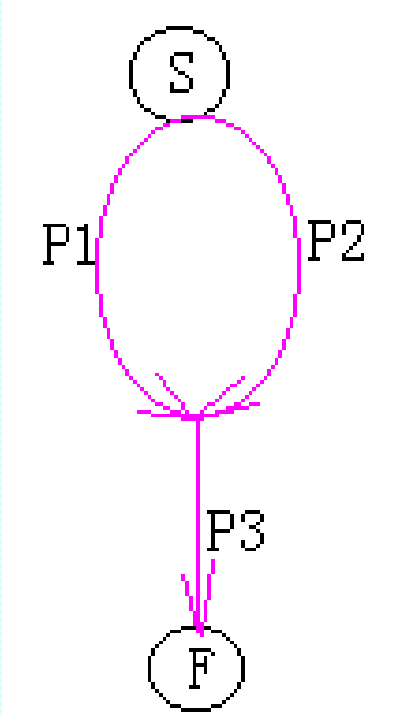
进程Pn
```
Pn()
{
    …
  P(s);
申请打印机;
  V(s);
   …
}
```

主程序描述
```
    main()
{  int s=m;
    cobegin{
          P1;
          P2;
        。 。 。
          Pn;
    }coend;
}
```

信号量实现互斥

- 为临界资源设置一个互斥信号量mutex，其初值为1；在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间。
- 必须成对使用P和V原语：遗漏P原语则不能保证互斥访问，遗漏V原语则不能在使用临界资源之后将其释放（给其他等待的进程）；
- P、V原语不能次序错误、重复或遗漏。

利用信号量来实现同步



用信号量及P、V操作来描述左图

1、说明进程的同步关系

进程P1、P2可并行执行，P3的执行必须等待P1、P2都完成后才能开始执行。

2、设置信号灯，说明含义、初值。

s13 = 0 表示进程P1尚未执行完成；

s23 = 0 表示进程P2尚未执行完成；

利用信号量来实现同步

- 程序描述

```
main()
{  int s13=0;s23=0;
   cobegin
        p1;
        p2;
        p3;
   coend;
}
```

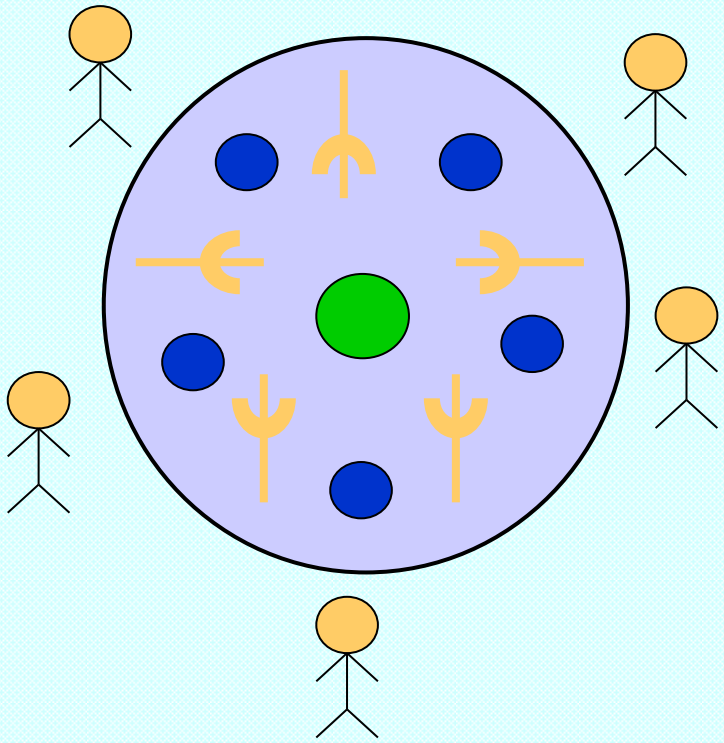
```
P1()
{
    …
    V(s13);
}
P2()
{
    ….
    V(s23);
}
```

```
P3()
{
    …
    P(s13);
    P(s23);
    ….
}
```

3.3.4 信号量解决哲学家就餐问题(1)

- 有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘于，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。

信号量解决哲学家就餐问题(2)



哲学家吃通心面问题(3)

```
semaphore fork[5];
for (int i=0;i<5;i++)
fork[i]=1;
cobegin
        process philosopher_i( ) {/*i= 0,1,2,3,4*/
          while(true) {
                think( );
                P(fork[i]);
                P(fork[(i+1)%5]);
                eat( );
                V(fork[i]);
                V(fork[(i+1)%5]);
          }
        }
coend
```

有若干种办法可避免这类死锁

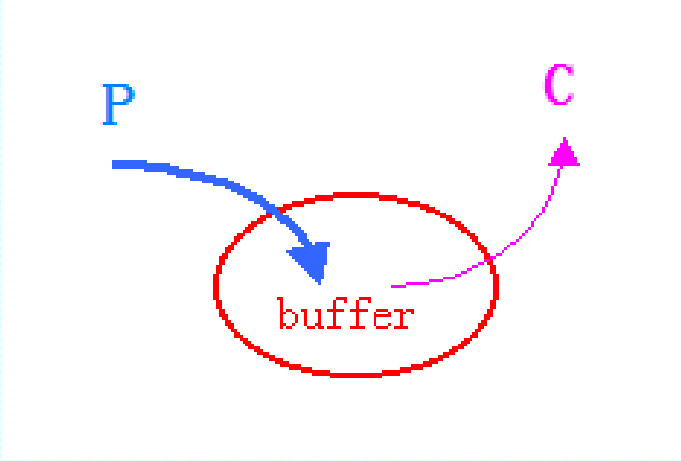
- 上述解法可能出现永远等待，有若干种办法可避免死锁：
  - 至多允许四个哲学家同时吃；
  - 奇数号先取左手边的叉子，偶数号先取右手边的叉子；
  - 每个哲学家取到手边的两把叉子才吃，否则一把叉子也不取。

哲学家问题的一种正确解

```
semaphore fork[5];
 for (int i=0;i<5;i++)
        fork[i]= 1;
cobegin
process philosopher_i( ){/*i=0,1,2,3 */
        while(true) {
        think( );
        P(fork[i]);                  /*i=4,P(fork[0])*/
        P(fork[(i+1)%5] );  /*i=4,P(fork[4])*/
        eat( );
        V(fork[i]);
        V(fork([i+ 1] % 5);
   }
 }
coend
```

3.3.5信号量解决生产者-消费者问题

- ①一个生产者、一个消费者共享一个缓冲区
- ②多个生产者、多个消费者共享多个缓冲区

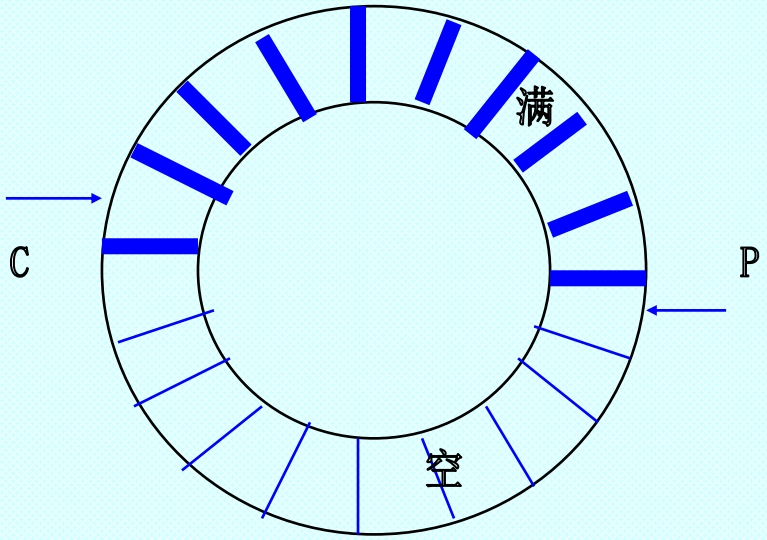


一个生产者/消费者共享一个缓冲区

```
int B;
semaphore empty;  /*可以使用的空缓冲区数*/
semaphore full;       /*缓冲区内可以使用的产品数*/
empty=1;              /*缓冲区内允许放入一件产品*/
full=0;               /*缓冲区内没有产品*/
cobegin
process   producer(){                    process   consumer(){
    while(true){                             while(true) {
        produce( );                              P(full);
        P(empty);                                take from B;
        append to B;                             V(empty);
        V(full);                                 consume( );
    }                                        }
}                                        }
coend
```

多个生产者-消费者共享多个缓冲区

- 共享资源
  - 缓冲池：K个缓冲区
  - P：一组生产者共用的指向空缓冲区头的指针；
  - C：一组消费者共用的指向满缓冲区头的指针。
- 互斥操作：
  - 分配空缓冲区和移动指针in;
  - 释放满缓冲区和移动指针out;



多个生产者-消费者共享多个缓冲区

- 信号量设置

  3个信号量

  - Empty：表示空的缓冲区数，初值为k；
  - Full：表示满的缓冲区数，初值为0；
  - Mutex：分配缓冲区的互斥信号量，初值为1。

多个生产者-消费者共享多个缓冲区

```
item B[k];
semaphore empty;     empty=k; /*可以使用的空缓冲区数*/
semaphore full; full=0;       /*缓冲区内可以使用的产品数*/
semaphore mutex;     mutex=1; /*互斥信号量*/
int in=0;                     /*放入缓冲区指针*/
int out=0;                    /*取出缓冲区指针*/
cobegin
process producer_i ( ){                process consumer_j ( ){
    while(true) {                          while(true) {
        produce( );                        P(full);
        P(empty);                          P(mutex);
        P(mutex);                          take( ) from B[out];
        append to B[in];                   out=(out+1)%k;
        in=(in+1)%k;                       V(mutex);
        V(mutex);                          V(empty);
        V(full);                           consume( );
    }                                      }
}                                      }
coend
```

多个生产者-消费者共享多个缓冲区

特点

- P次序不能颠倒，否则会出现死锁
  - 当Empty＝0，Full＝k时，

    Producer：P(mutex)-> Producer :P(Empty)->

    consumer :P(mutex)-> consumer :wait(Full)
  - 当E＝n，F＝0时，

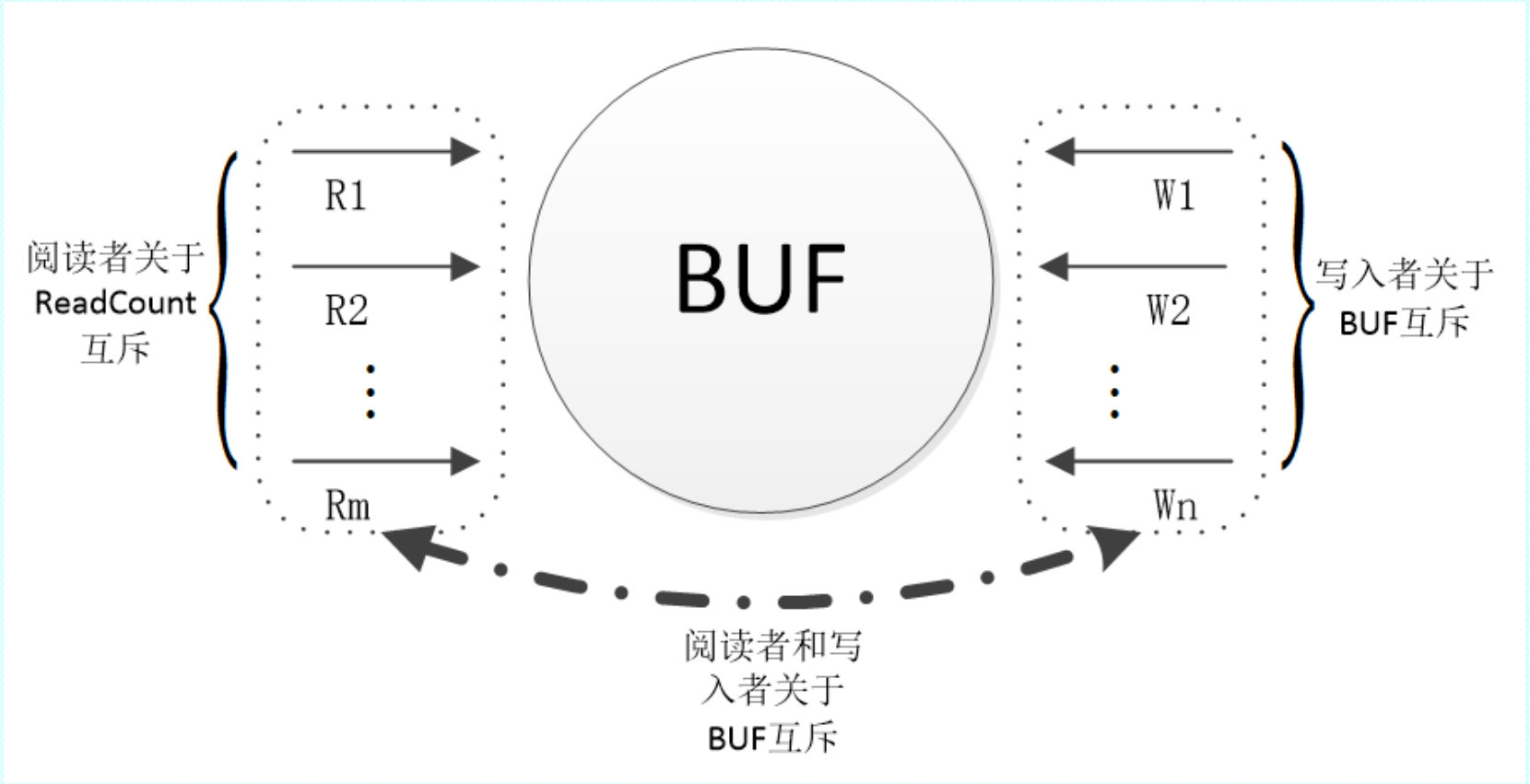
    consumer：P(mutex)-> consumer :P(Full)->

    Producer :P(mutex)-> Producer :P(Empty)
  - 生产者和消费者的缓冲指针in、out不能同时移动。即缓冲分配不能同时进行。
- 可改进：将两个互斥信号量来分别控制对指针in、out的操作。

3.3.6 信号量解决读者-写者问题(1)

- 有两组并发进程：读者和写者，共享一个文件F，要求：
  - 允许多个读者同时执行读操作；
  - 只允许一个写者执行写操作；
  - 任一写者在完成写操作之前不允许其它读者或写者工作；
  - 写者执行写操作前，应让已有的写者和读者全部退出。

信号量解决读者写者问题(2)



信号量解决读者写者问题(3)

- int readcount=0;/*读进程计数*/
- semaphore writeblock,mutex;
- writeblock=1;mutex=1;

信号量解决读者写者问题(4)

cobegin

```
process reader_i( ){
    P(mutex);
    readcount++;
    if(readcount==1)
        P(writeblock);
    V(mutex);
    /*读文件*/;
    P(mutex);
    readcount--;
    if(readcount==0)
        V(writeblock);
    V(mutex);
}
```

```
process writer_j( ){
    P(writeblock);
    /*写文件*/;
    V(writeblock);
}
```

coend

3.3.7信号量解决睡眠理发师问题(1)

- 理发店理有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子；
- 如果没有顾客，理发师便在理发椅上睡觉
- 一个顾客到来时，它必须叫醒理发师；
- 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。

信号量解决理发师问题(2)

- int waiting=0; /*等候理发顾客坐的椅子数*/
- int CHAIRS=N; /*为顾客准备的椅子数*/
- semaphore customers,barbers,mutex;
- customers=0;barbers=0;mutex=1;

信号量解决理发师问题(3)

```
cobegin
process barber( ) {
    while(true) {
        P(customers);/*有顾客吗？若无顾客，理发师睡眠*/
        P(mutex);          /*若有顾客时，进入临界区*/
        waiting--;         /*等候顾客数少一个*/
        V(barbers);        /*理发师准备为顾客理发*/
        V(mutex);          /*退出临界区*/
        cut_hair();        /*理发师正在理发(非临界区)*/
    }
}
```

信号量解决理发师问题(4)

```
process customer_i( ) {
        P(mutex);                  //进入临界区
        if(waiting<CHAIRS) {    //有空椅子吗
            waiting++;          //等候顾客数加1
            V(customers);       //唤醒理发师
            V(mutex);           //退出临界区
            P(barbers);         //理发师忙，顾客坐下等待
            get_haircut();      //否则顾客坐下理发
        }
        else
            V(mutex);           //人满了，走吧!
}
coend
```

3.4 管程

- 管程的基本概念
- 条件变量
- 使用管程解决生产者-消费者问题

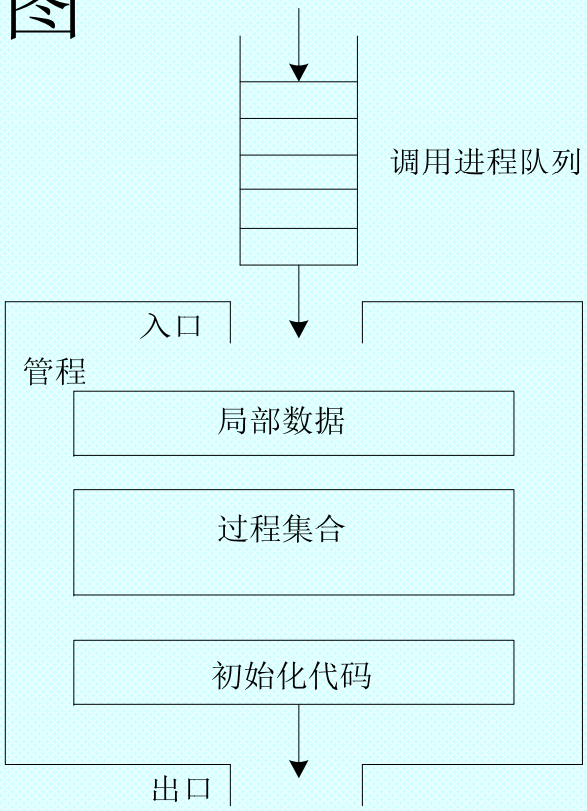
3.4.1 管程的基本概念

- 信号量机制功能强大，是一种既方便、又有效的进程同步机制，但是在使用过程中对信号量的操作分散在各个进程中，不易控制。
- 管程定义

  管程是由过程、变量及数据结构等组成，它们共同构成了一个特殊的模块或软件包。进程可以在任意时刻调用管程中的过程，但不允许使用管程外的过程来访问管程内的数据结构。
- 管程中在任一时刻只能有一个活跃进程，这一特性可以帮助管程有效地实现互斥。

3.4.1 管程的基本概念

- 管程示意图



管程的基本概念

- 管程与进程的区别：

(1) 虽然二者都定义了数据结构，但进程定义的是私有数据结构PCB，而管程定义的是公共数据结构，如条件变量等；

(2) 二者都存在对各自数据结构上的操作，但进程是顺序执行的，而管程则是进行同步操作和初始化操作；

(3) 进程是为了保证系统并发而设计的，管程则是为了解决共享资源的互斥；

(4) 进程通过调用管程中的过程来进行共享数据结构的操作，该过程就表现为进程的子程序，因此进程是主动的，管程是被动的；

(5) 进程间可并发，管程作为子程序不能与其调用者并发；

(6) 进程具有动态性和生命周期，而管程只是一个资源管理模块，供进程调用。

条件变量

- 在管程中，除了完成互斥外，还要保证能将无法继续运行的进程正确阻塞。因此，需要引入条件变量(condition variable)以及对其的一对wait、signal操作。

条件变量

- wait操作的含义：

  x.wait表示正在调用管程的进程因x条件需要被阻塞或挂起，该进程在执行wait操作时将自己插入x条件的等待队列中，并释放管程。此时其它进程可以使用该管程完成自身工作，当x条件产生变化时，系统调度程序将选择等待队列中的一个进程继续执行。

条件变量

- signal操作的含义：

  x.signal表示正在调用管程的进程发现x条件发生了变化，则使用signal操作唤醒一个因x条件被阻塞或挂起的进程。

- 当一个管程中的过程发现自身无法运行下去(如生产者进程发现缓冲池满)时，它将对某个条件变量(如full)执行wait操作。该操作会阻塞调用进程自身，并将其它在管程外等待的进程调入管程。

3.4.2 使用管程解决生产者-消费者问题

- 使用管程机制可以很好地解决生产者-消费者问题。此时需要首先建立一个管程ProducerConsumer，其中包含两个过程insert(item)和consumer(item)。

3.4.2 使用管程解决生产者-消费者问题

- 伪代码描述如下：

```
monitor ProducerConsumer
    condition full,empty;
    int count;
    void insert(int item)
    {
        if (count==N) wait(full);
        insert(item);
        count=count+1;
        if (count==1) signal(empty);
    }
    int remover()
    {
        if (count==0) wait(empty);
        remove=remove_item;
        count=count-1;
        if (count==N-1) signal(full);
    }
    count=0;
end monitor
```

3.4.2 使用管程解决生产者-消费者问题

- 伪代码描述如下: (续)

```
void producer()
{
        while (true)
        {
                item=produce_item;
                ProducerConsumer.insert(item);
        }
}
void consumer()
{
        while (true)
        {
                item=ProducerConsumer.remove;
                consume(item)
        }
}
```

3.5 进程通信

- 进程通信的概念
- 进程通信的方式
- 消息传递系统
- 消息缓冲队列通信机制
- 管道通信方式

3.5、进程通信

- 进程通信：指进程之间的信息交换。
  - 作业—>若干个可并行执行的进程—>协同完成一个工作（同步）—>进程通信（在进程之间交换一定数量的信息）。
- 进程通信方式：
  - 低级通信原语：交换信息量较少。如互斥，同步机构。
  - 高级通信原语：交换信息量较多。如直接通信，间接通信。
- 高级通信原语：
  - 直接通信：一个进程直接发送消息给接受者进程；

    Send(P, Msg);　　Receive(P, Msg);
  - 间接通信：进程通过一个“信箱”来传递消息。

    Send(A, Msg);　　Receive(A, Msg);

直接通信

- 要求发送进程和接收进程都以显示的方式提供对方的标识符。通常系统提供两条通信原语。
- 原语send（P，消息）：把一个消息发送给进程P
- 原语receive（Q，消息）：从进程Q接收一个消息

消息队列

通常一个进程可以与多个进程通信，即可以向多个进程发送消息，也可以接收来自多个进程的消息。为了便于进程接收和处理这些消息，一般采用消息队列通信机制，将消息组织成消息队列，用链指针链接起来，头指针放在进程的PCB中。

有关数据结构

- 消息队列

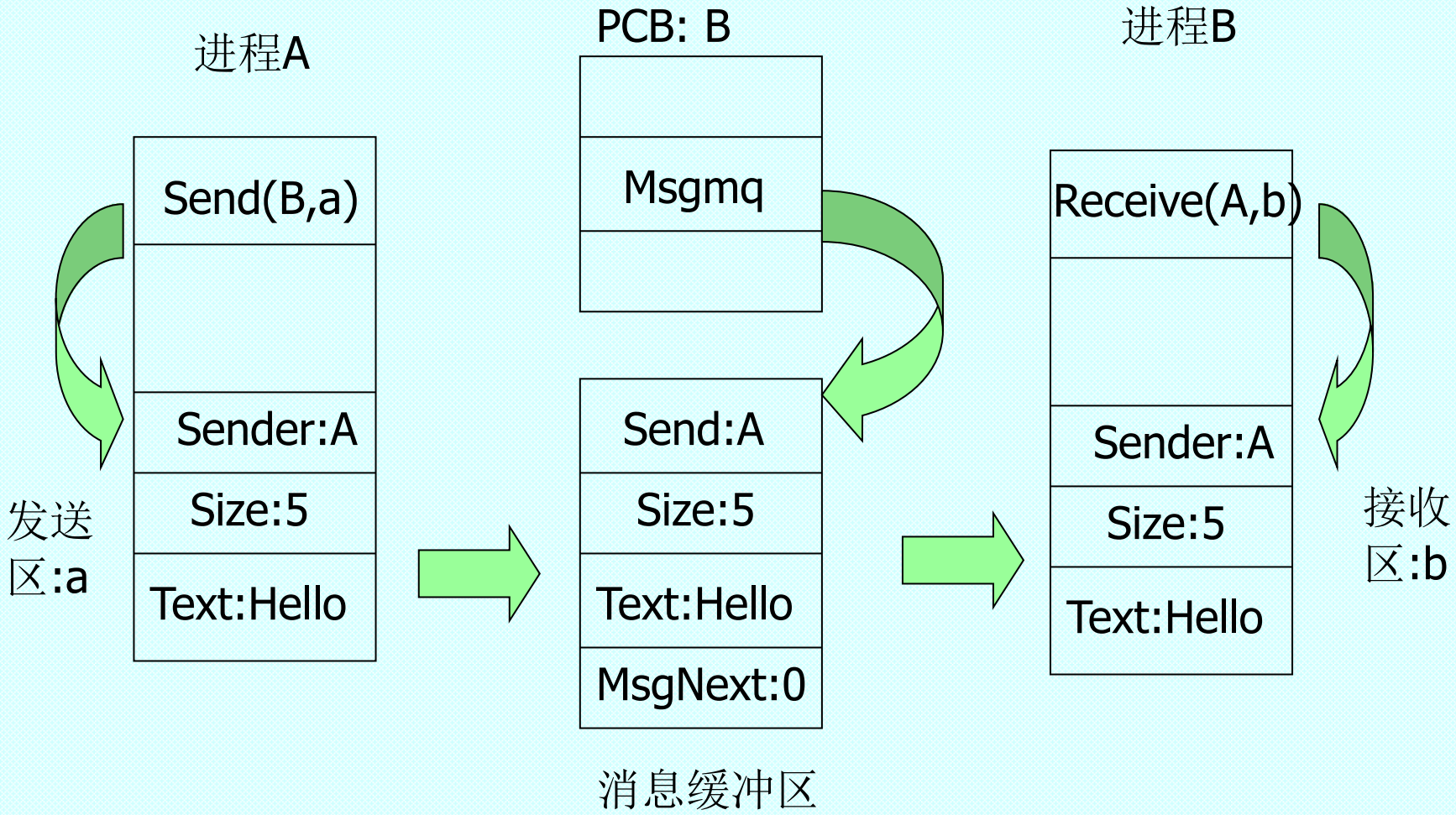
```
type Msg = record
    MsgSend;
      MsgSize;
      MsgText;
      MsgNext;
end
```

- PCB中部分数据

```
type PCB = record
          ……
      Msgmq;    首指针
      MsgMutex;  互斥信号量
      MsgSm;     资源信号量
          ……
end
```

发送和接收过程

- 发送进程在自己地址空间设置一发送区，将发送的消息正文，发送者进程标示符，消息长度填入其中，然后调用发送原语。
- 发送原语根据发送区的消息长度，申请一缓冲区，将发送区的消息复制到缓冲区中。并获得接收进程的内部标识符，然后将缓冲区挂在接收进程的消息队列上。
- 接收进程调用接收原语，从自己的消息队列中摘下消息队列中的消息，并将其中的数据复制到指定的消息接收区。


进程A
Send(B,a)
Sender:A
Size:5
Text:Hello
发送区:a
PCB: B
Msgmq
Send:A
Size:5
Text:Hello
MsgNext:0
消息缓冲区
进程B
Receive(A,b)
Sender:A
Size:5
Text:Hello
接收区:b

同步机制

- 信号量：
  - 互斥信号量（mutex）：对消息对列指针的操作
  - 等待信号量（swait）：消息资源数

发送时：

wait(mutex);

将消息链入队列；

signal（mutex）；

signal（swait）；

接收时：

wait（swait）；

wait(mutex);

从队列中摘取消息；

signal（mutex）；

间接通信

- 进程间发送或接收消息通过一个信箱来进行，消息可以被理解成信件
- 原语send（A，信件）：把一封信件（消息）传送到信箱A
- 原语receive（A，信件）：从信箱A接收一封信件（消息）

间接通信的实现

- 信箱是存放信件的存储区域，每个信箱可以分成信箱特征和信箱体两部分。信箱特征指出信箱容量、信件格式、指针等；信箱体用来存放信件
- 发送信件：如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则发送信件者被置成等待信箱状态
- 接收信件：如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者，否则接收信件者被置成等待信箱中信件的状态

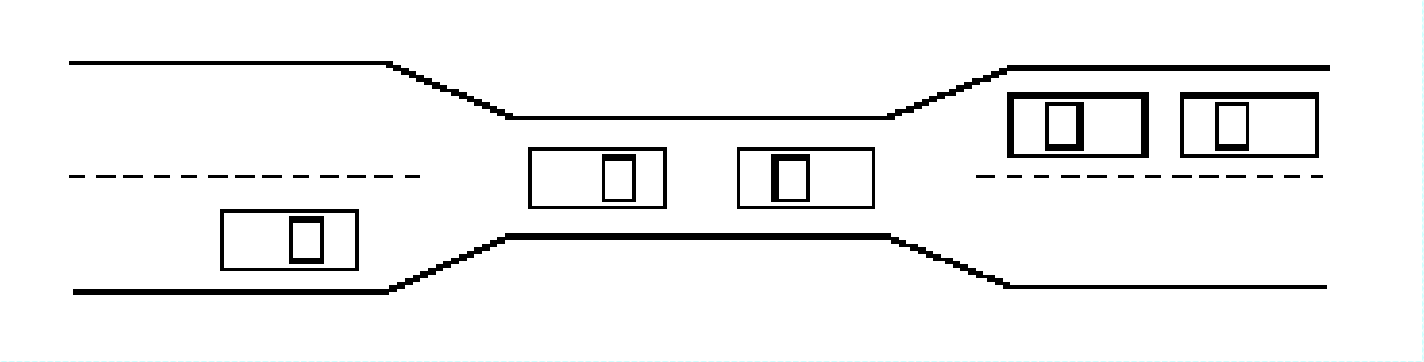
3.7 死锁

- 死锁产生
- 死锁防止
- 死锁避免
- 死锁检测和解除

3.7.1 死锁问题的提出

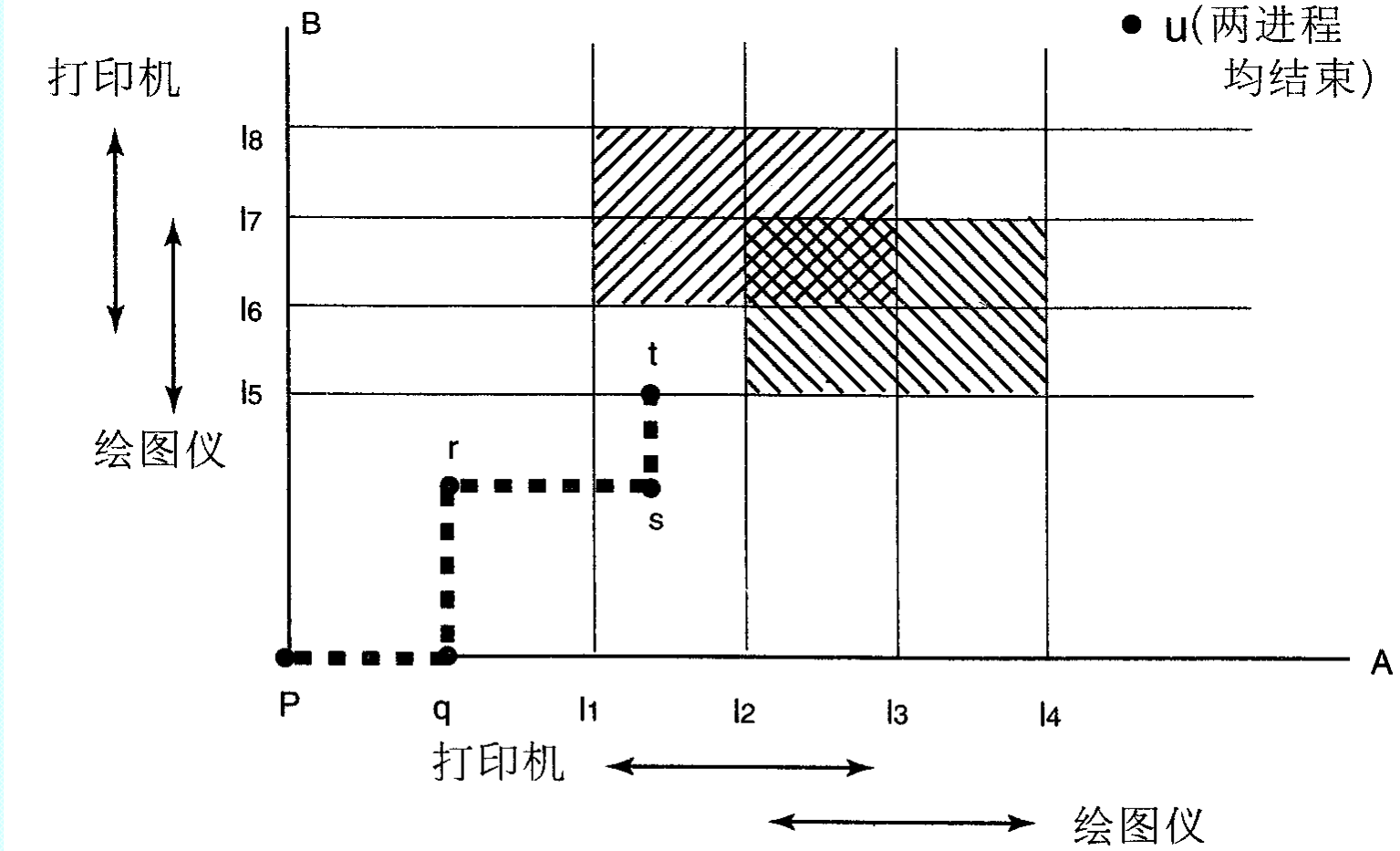
- 死锁定义：死锁是指系统中的一组进程，由于竞争系统资源或由于彼此通信而永远阻塞，称这些进程处于死锁状态。
- 死锁的产生是与资源分配策略和并发进程执行的速度有关

交通中的死锁



- 桥只能单向行驶
- 桥可以看成一种资源
- 如果死锁发生，可以通过一辆车倒退解决（即释放已占有的资源）
- 可能需要多辆车倒退
- 可能发生饥饿现象

计算机系统中的死锁



死锁产生的原因

- 进程竞争资源，而资源不足

  当系统中供多个进程所共享的资源不足以同时满足进程的需要时，就可能引起进程对资源的竞争而产生死锁。

- 进程推进顺序不合适

  在进程运行过程中，当请求和释放资源的顺序不当时，可能会导致进程死锁。

- 如系统有打印机5台，它们有N个进程竞争使用，每个进程需要同时使用2台打印机，则N取哪些值时，系统不会死锁？

分析：

N=1时，系统资源数大于进程要求

N=2时，系统资源数大于进程要求

N=3时，系统资源数小于进程要求，最坏情形是先每个进程分配1个资源，此时剩余2个资源，只要分配给任何一个进程，该进程就可以完成，从而释放所有资源。

N=4时，系统资源数小于进程要求，最坏情形是先每个进程分配1个资源，此时剩余1个资源，只要分配给任何一个进程，该进程就可以完成，从而释放所有资源。

N=5时，当每个进程分配一个打印机，系统已无剩余资源，每个进程都没有获得需要的资源数，不能完成，也不能释放其所占资源，死锁。

- 设系统某类资源有m个，有n个进程，每个进程需要K个该资源，则当满足

  nk<= m+(n-1)时，系统不会引起死锁。

死锁的必要条件

资源的分类

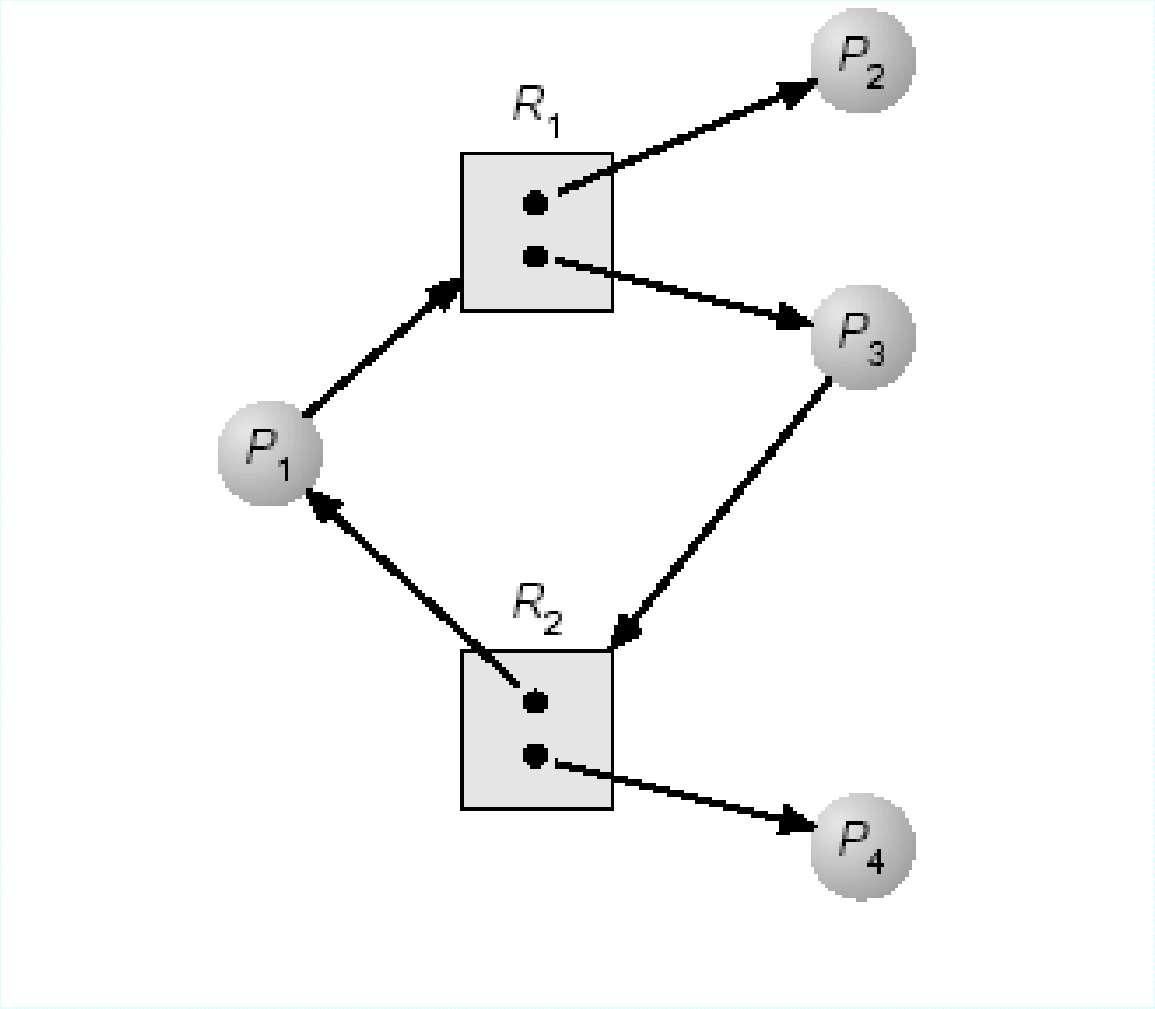
- 根据资源是否可抢占
  - 可抢占资源：指资源占有者进程虽然仍需要使用资源，但系统可以根据某原则强行将该资源剥夺，分配给其他进程。
  - 不可抢占资源：指资源一旦被进程占有，只有当进程不再使用而主动释放资源外，其他进程不得强行抢占其资源。
- 根据资源使用方式
  - 共享资源：指资源同时可以为多个进程共同使用。
  - 独享资源：指资源同一时刻只能为一个进程单独使用
- 进程因竞争独享、不可抢占资源而发生死锁。

死锁的必要条件

- 互斥条件：一个资源一次只能被一个进程所使用。
- 不可抢占条件：一个资源仅能被占有它的进程所释放，而不能被其他的进程强行抢占。
- 部分分配条件：一个进程已占有分给它的资源，但仍然要求其他资源。
- 循环等待条件：在系统中存在一个由若干个进程形成的环形请求链，其中的每一个进程均占有若干种资源中的某一种，同时还要求下一个进程所占有的资源。

进程-资源分配图

- 约定Pi→Rj为请求边，表示进程Pi申请资源类Rj中的一个资源得不到满足而处于等待Rj类资源的状态，该有向边从进程开始指到方框的边缘，表示进程Pi申请Rj类中的一个资源。
- Rj→Pi为分配边，表示Rj类中的一个资源已被进程Pi占用，由于已把一个具体的资源分给了进程Pi，故该有向边从方框内的某个黑圆点出发指向进程。



进程-资源分配图

处理死锁的基本方法

- 处理死锁有三种基本方法：
  - 死锁的预防
  - 死锁的避免
  - 死锁的检测和恢复

处理死锁的基本方法

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置（从机制上使死锁条件不成立）	一次请求所有资源<条件 3>	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新起动 不便灵活申请新资源
		资源剥夺<条件 2>	适用于状态可以保存和恢复的资源	
		资源按序申请<条件 4>	可以在编译时（而不必在运行时）就进行检查	
避免 Avoidance	是“预防”和“检测”的折衷（在运行时判断是否可能死锁）	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

3.7.2 死锁防止

- 死锁的预防主要通过设置某些限制条件，破坏死锁产生的必要条件，以达到不产生死锁的目的。
- 破坏第一个条件（互斥条件）：使资源可同时访问而不是互斥使用，
- 破坏第二个条件（部分分配条件）：采用预先静态分配法
- 破坏第三个条件（不剥夺条件）：采用剥夺式调度方法
- 破坏第四个条件（循环等待条件）：采用层次分配策略

预先静态分配法

- 实质：破坏部分分配条件
- 实现：在进程开始运行之前，一次分配给其所需的全部资源，若系统不能满足，则进程阻塞，直到系统满足其要求。
- 优点：简单，易于实现且安全
- 缺点：
  - 资源严重浪费。
  - 延迟运行。

有序资源使用法

- 实质：破坏循环等待条件
- 实现：
  - 把资源分类，按序排列，每种资源分配唯一序号；
  - 进程对资源的请求必须严格按资源序号的递增次序申请；
  - 经常用的普通的资源低序号，贵重少用的高序号。
- 优点：基于动态分配，资源利用率提高
- 缺点：
  - 由于资源序号必须相对稳定，限制了新设备类型的增加；
  - 存在资源申请次序与实际使用次序不一致，导致利用率不高。

例如：进程PA，使用资源的顺序是R1，R2；

进程PB，使用资源的顺序是R2，R1；

若采用动态分配有可能形成环路条件，造成死锁。

采用有序资源分配法：R1的编号为1，R2的编号为2；

PA：申请次序应是：R1，R2

PB：申请次序应是：R1，R2

这样就破坏了环路条件，避免了死锁的发生

3.7.3 死锁的避免

- 在分配资源时判断是否会出现死锁，如不会死锁，则分配资源。
- 系统的两种状态。
  - 安全状态：

    指系统在资源分配中能找到某种分配顺序（如：p1,p2,p3,……,pn），按该顺序来为每个进程分配其所需资源，直至最大需求，可使每个进程都能完成，则称系统处于安全状态，该分配顺序成为安全序列。
  - 不安全状态：

    指系统在资源分配中能不能找到某种分配顺序（如：p1,p2,p3,……,pn），可使每个进程都能完成，则称系统处于不安全状态。

银行家算法

- 银行家算法
  - 银行家拥有一笔周转资金
  - 客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款
  - 银行家应谨慎的贷款，防止出现坏帐
- 用银行家算法避免死锁
  - 操作系统（银行家）
  - 操作系统管理的资源(周转资金)
  - 进程（要求贷款的客户）

系统安全性定义

- 系统安全性定义：在时刻T0系统是安全的，仅当存在一个进程序列P1,..,Pn，对进程Pk满足公式：

$$Need[k,i] \leq Available[i] + \sum Allocation[j,i]$$

$$k=1,\ldots,n;$$

$$i=1,\ldots,m;$$

银行家算法基本思想

- 系统中的所有进程进入进程集合,
- 在安全状态下系统收到进程的资源请求后, 先把资源试探性分配给它。
- 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较，在进程集合中找到剩余资源能满足最大需求量的进程, 从而, 保证这个进程运行完毕并归还全部资源。
- 把这个进程从集合中去掉，系统的剩余资源更多了, 反复执行上述步骤。
- 最后, 检查进程集合, 若为空表明本次申请可行, 系统处于安全状态, 可实施本次分配; 否则, 有进程执行不完，系统处于不安全状态, 本次资源分配暂不实施, 让申请进程等待。

单资源银行家算法

- 对每一个请求进行检查，检查如果满足它，是否会导致不安全状态。若是，则不满足该请求；否则便满足
- 检查状态是否安全的方法：

  （1）计算所有客户离最大需求的距离；

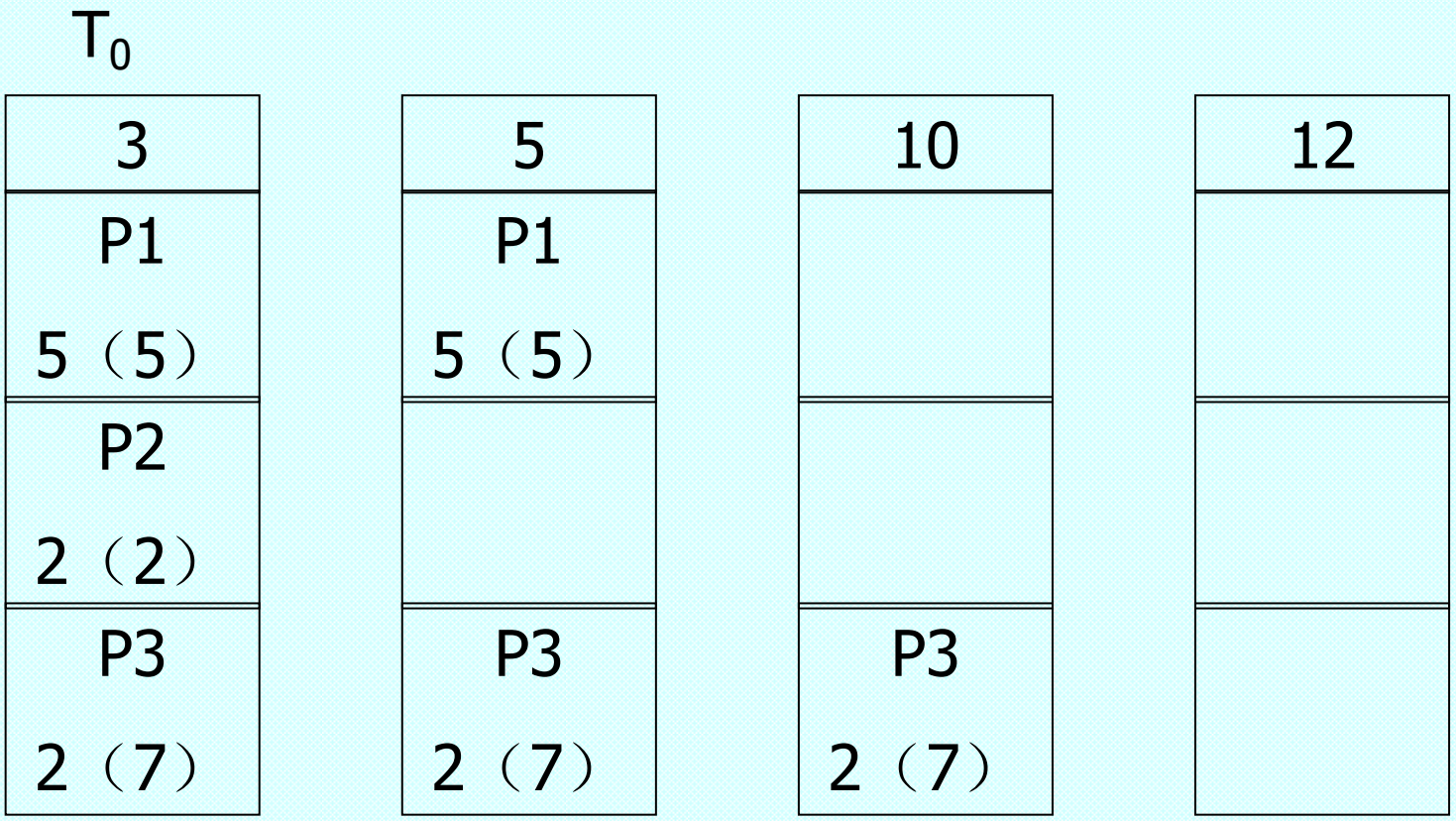
  （2）检查系统资源状况，获得可以被满足的客户；

  （3）挑选一个客户，一般挑选距离最短的客户，假设满足该客户，则可以收回该客户所分配的所有资源；

  （4）计算剩余客户客户离最大需求的距离；

  （5）返回（2），如此反复下去。如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。

例：系统有3个进程（P1、P2和P3），共有12台打印机，P1总共要求10台打印机，P2和P3分别要求4台和9台，在T_0时刻，进程P1、P2和P3已分别获得5台、2台和2台打印机，问此时，系统是否安全？



安全序列：P2,P1,P3

多资源银行家算法

- 实际系统中可能有多种资源，每类资源有不同的个数
- 多资源银行家算法中定义了
  - 分配矩阵
  - 请求矩阵
  - 请求向量
  - 可用资源向量（剩余资源向量）

多资源银行家算法

进程	磁带机	绘图仪	打印机	CD-ROM
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

已分配的资源

进程	磁带机	绘图仪	打印机	CD-ROM
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

仍需要的资源

E = (6342)
P = (5322)
A = (1020)

总资源E、已分配资源P、剩余资源A

银行家算法资源分配步骤(1)

- （1）如果 Request [i, *] ≦Need[i,*]，转步骤（2）；否则，进程申请量超过最大需求量，出错处理。
- （2）如果 Request [i, *] ≦Available [*]，转步骤（3）；否则，申请量超过当前系统拥有的可分配量，进程P i 等待 。
- （3）系统对 P i进程请求资源进行试探性分配，执行：

  Allocation [i, *] = Allocation [i, *] + Request [I,*]

  Available [*] = Available [*] − Request [i,*]

  Need [i,*] =Need [i,*] - Request [i, *]
- （4）转向（5）执行安全性测试算法，如果返回安全状态则承认试分配，否则，抛弃试分配，进程 P i等待，并执行；

  Allocation [i, *] = Allocation [i, *] - request [i,*]

  Available [*] = Available [*] + Request [i,*]

  Need [i,*] =Need [i,*] + Request [i,*]

银行家算法资源分配步骤(2)

- （5）安全性测试算法
  - ① 定义工作向量 Work[i]、布尔型标志 possible 和进程集合 rest;
  - ② 执行初始化操作：让全部进程进入 rest 集合，并让：

    Work [*] =Available [*] , possible =true ;
  - ③ 保持 possible =true，从进程集合 rest 中找出满足下列条件的进程 Pk ：

    Need [k, *] ≤ Work [*]
  - ④ 如果不存在，则转向⑤；如果找到，则释放进程 Pk 所占用的资源，并执行以下操作 ：

    Work [*] = Work [*] + Allocation [k, *]

    把Pk从进程集合中去掉 ，即 rest = rest－ {Pk} ，再转向③ ；
  - ⑤ 置 possible =false，停止执行本算法;
  - ⑥最后，查看进程集合 rest，若其为空集则返回安全标记；否则，返回不安全标记 。

实例说明系统所处的安全或不安全状态(1)

- 如果系统中共有五个进程和A、B、C三类资源;
- A类资源共有10个,B类资源共有5个,C类资源共有7个。
- 在时刻T0,系统目前资源分配情况如下：

实例说明系统所处的安全或不安全状态(2)

process	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

实例说明系统所处的安全或不安全状态(3)

每个进程目前还需资源为Need[k,i]

process	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

实例说明系统所处的安全或不安全状态(4)

- 可以断言目前系统处于安全状态，因为，T0时刻序列{P1，P3，P4，P2，P0}能满足安全性条件。

实例说明系统所处的安全或不安全状态(5)

资源 / 进程	work			Need			allocation			work+allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	3	2	1	2	2	2	0	0	5	3	2	TRUE
P_3	5	3	2	0	1	1	2	1	1	7	4	3	TRUE
P_4	7	4	3	4	3	1	0	0	2	7	4	5	TRUE
P_2	7	4	5	6	0	0	3	0	2	10	4	7	TRUE
P_0	10	4	7	7	4	3	0	1	0	10	5	7	TRUE

实例说明系统所处的安全或不安全状态(6)

- 进程P1申请资源request1=(1,0,2），检查request1≤Available,比较(1,0,2) ≤(3,3,2),结果满足条件,试分配,得到新状态：

process	allocation			Need			available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

实例说明系统所处的安全或不安全状态(7)

- 判定新状态是否安全?可执行安全性测试算法，找到一个进程序列{P1, P3, P4, P0, P2}能满足安全性条件, 可正式把资源分配给进程P1;

资源 / 进程	work			Need			allocation			work+allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	2	3	0	0	2	0	3	0	2	5	3	2	TRUE
P_3	5	3	2	0	1	1	2	1	1	7	4	3	TRUE
P_4	7	4	3	4	3	1	0	0	2	7	4	5	TRUE
P_0	7	4	5	7	4	3	0	1	0	7	5	5	TRUE
P_2	7	5	5	6	0	0	3	0	2	10	5	7	TRUE

实例说明系统所处的安全或不安全状态(8)

- 系统若处在下面状态中，进程P4请求资源(3,3,0)，由于可用资源不足，申请被系统拒绝；此时，系统能满足进程P0的资源请求(0,2,0)；但可看出系统已处于不安全状态了。

资源 进程	allocation	Need	available
	A B C	A B C	A B C
P0	0 3 0	7 2 3	2 1 0
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

3.7.4 死锁的检测和恢复

- 基本思想：通过系统设置的检测机构，实际地检查系统中是否存在死锁，并精确地标定出与死锁有关的进程和资源。
- 实现：在OS中保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。
- 检测算法主要是检查是否有循环等待
- 不采取任何限制措施，也不检查系统是否会进入不安全状态，允许系统在运行中发生死锁。

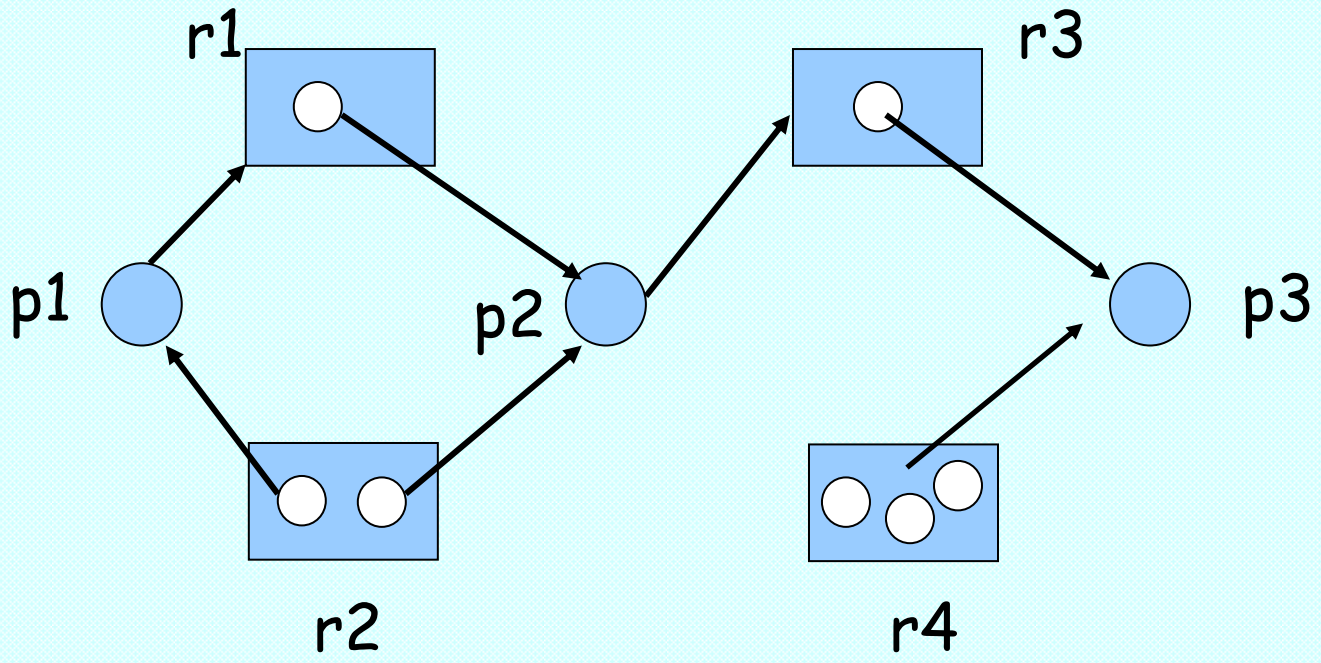
死锁定理

- (1)如果进程-资源分配图中无环路，则此时系统没有发生死锁。
- (2)如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程。
- (3)如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件。
  - 如果能在进程-资源分配图中消去此进程的所有请求边和分配边，成为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的。
  - 系统为死锁状态的充分条件是：当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理。

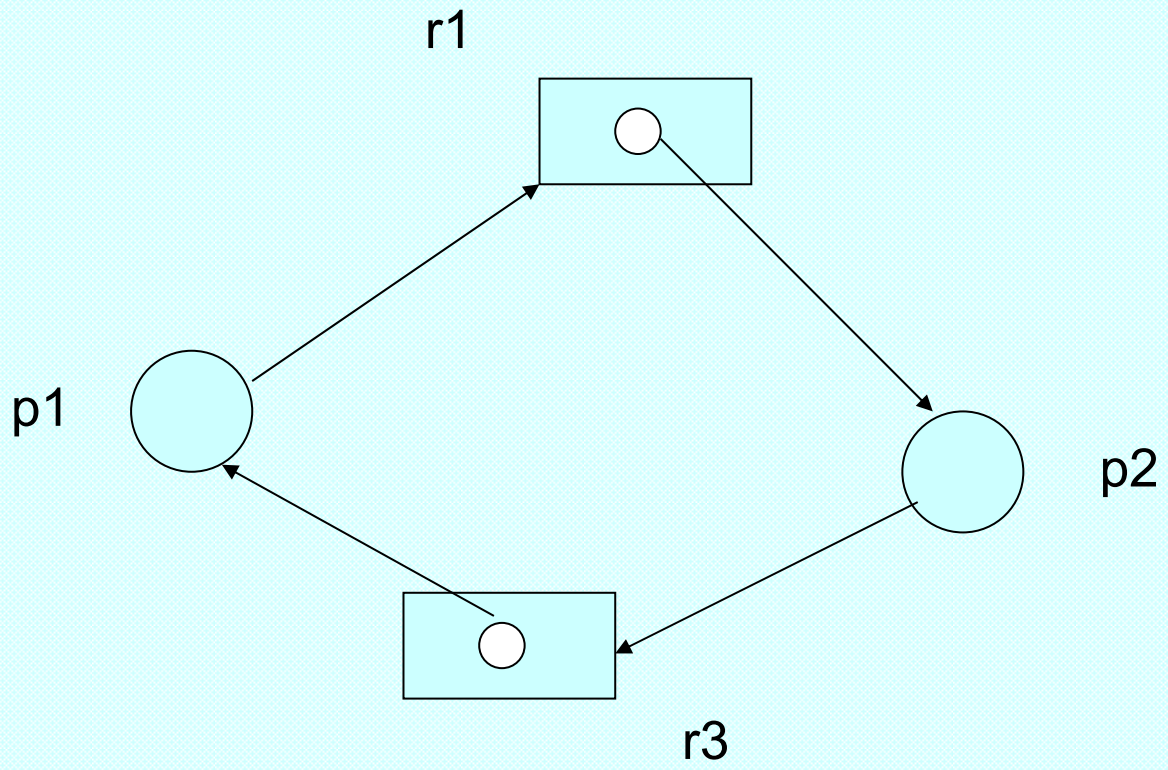
例子（无环路，无死锁）

例1. P={p1,p2,p3}, R={r1(1),r2(2),r3(1),r4(3)}

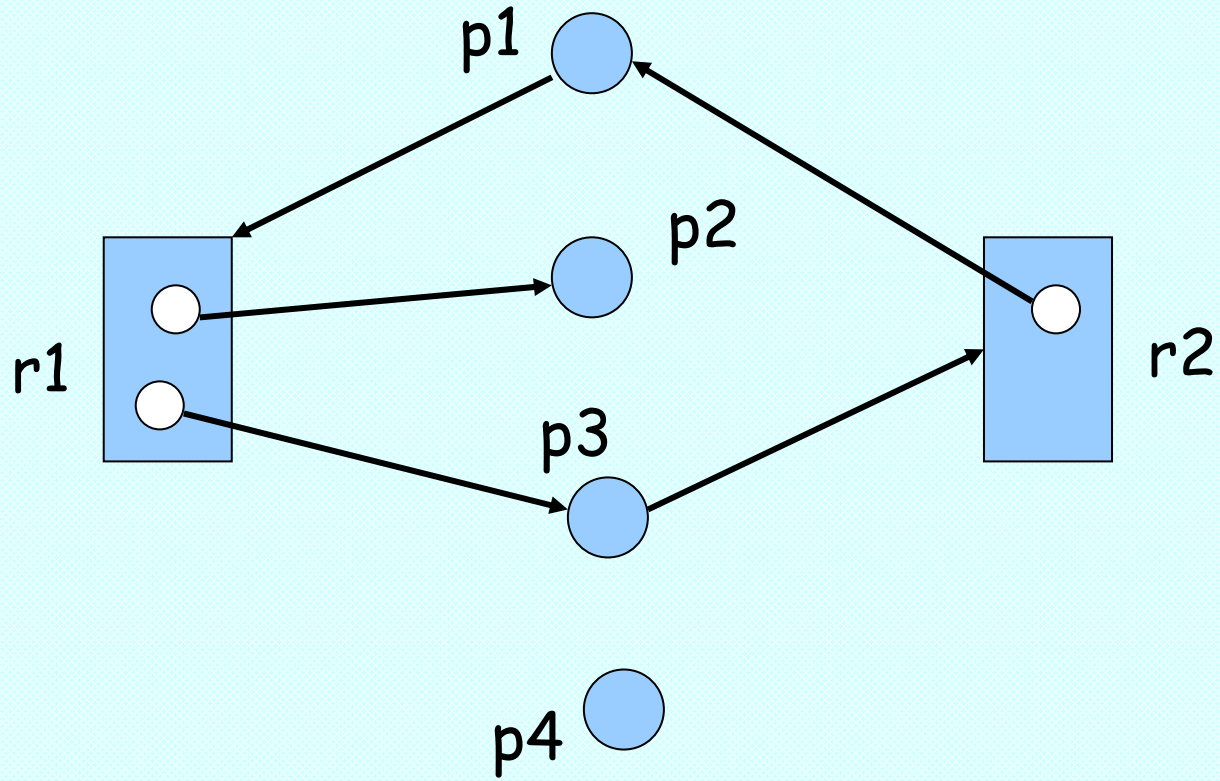
E={(p1,r1),(p2,r3),(r1,p2),(r2,p1),(r2,p2),(r3,p3)}



例子（有环路，一个资源实体，有死锁）



例子（有环路，无死锁）



死锁的检测和解除方法(1)

- (1) 借助于死锁的安全性测试算法来实现。死锁检测算法与死锁避免算法是类似的，不同在于前者考虑了检查每个进程还需要的所有资源能否满足要求；而后者则仅要根据进程的当前申请资源量来判断系统是否进入了不安全状态。

死锁检测和解除方法(2)

- 一种具体死锁检测方法，检测算法步骤如下：
  - （１）Available [m]是长度为 m 的向量，说明每类资源中可供分配的资源数目 。
  - （２）Allocation[n,m] 是 n × m 矩阵，说明已分配给每个进程的每类资源数目 。
  - （３）Request [n,m]是 n × m 矩阵，说明当前每个进程对每类资源的申请数目 。
  - （４）Work [m]是长度为 m 的工作向量 。
  - （５）finish [n]是长度为 n 的布尔型工作向量 。

死锁检测和解除方法(3)

- 令 k ＝ 1 ， 2 ， …， n，死锁检测算法步骤如下：
  - (1) Work［ * ］＝ Available ［ * ］；
  - (2) 如果 Allocation［k，*］≠ 0 ，令 finish［ k］＝ false ； 否则 finish［k］ ＝true ；
  - (3) 寻找一个 k ， 应满足条件：
    （finish［k］＝＝ false）&&（Request［k ， *］ ≤ Work［ * ］）
    若找不到这样的 k ， 则转向步骤（ 5 ） ；
  - (4) 修改 Work［*］＝ Work［*］＋Allocation［k，*］， finish［ k］＝ true ， 然后， 转向步 骤（ 3 ）；
  - (5) 如果存在 k（ 1 ≤k≤n），finish［k］＝false，则系统处于死锁状态，并且 finish［k］＝false 的Pk 是处于死锁的进程 。

死锁的解除

- 与死锁的检测配套使用
- 当发现进程死锁时，采用一定的策略，将系统从死锁状态中恢复。
- 常用的方法：
  - 撤销进程并剥夺资源。
  - 使用挂起和解除挂起机构