西安科技大学
XI' AN UNIVERSITY OF SCIENCE TECHNOLOGY

第四章 进程的并发和死锁

刘晓建

2018年10月15日

本章概要

第三章介绍了进程的结构、进程的状态转换、进程的控制以及进程的调度，研究了进程与操作系统之间的关系。本章将探讨进程与进程之间的关系：

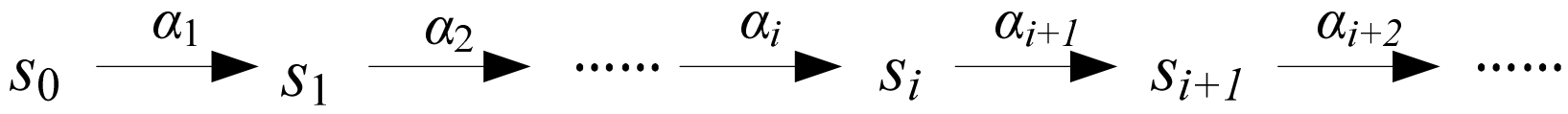
- 进程之间的并发控制：同步与互斥
- 进程之间的通信
- 进程死锁：由于进程之间的并发控制不当有可能导致进程死锁问题发生，如何避免、预防和检测死锁?

4.1 并发问题

程序是指令的一个序列，通常以二进制文件的形式存在，而进程是程序的一次执行过程。程序执行通常用一个无限长的状态序列来描述。

如果从基于状态的观点来看待一个程序执行，那么程序的执行可以被描述成状态迁移的序列，即$<s_0, s_1, \ldots, s_i, s_{i+1}, \ldots>$，从$s_i$到$s_{i+1}$的状态迁移称为一个执行步（Step）。

如果从基于事件的观点来看待一个程序执行，那么程序的执行可以被描述成原子操作的序列$<a_1, a_2, \ldots, a_i, a_{i+1}, \ldots>$，这两种描述方式是等价的。



为了明确表示程序状态与程序执行的操作之间的关系，我们用

$$\{s = s_i\}\ a_{i+1}\{\ s = s_{i+1}\}$$

引入这样的表示之后，上面的程序执行序列就可以表示为：

$$\{s= s_0\}a_1\{\ s= s_1\}a_2\{\ s= s_2\}\cdots\cdots\{s= s_i\}a_{i+1}\{\ s= s_{i+1}\}\cdots\cdots$$

如果一个程序执行的结果仅取决于初始状态s_0，即初始状态确定后，程序的每一个执行步就被完全确定下来，我们把这样的程序称为**确定性程序**。顺序程序是确定性程序。

我们把多个进程可以在同一时间段内同时执行的现象称为程序的**并发性**。对于单处理器而言，这种并发性是“伪并发”，实际上是通过进程的交叉（或交叠）（interleaved或overlapped）执行来实现的。

比如，两个进程A和B，A顺序执行操作<a_1, a_2, a_3>，B顺序执行<β_1, β_2, β_3>。如果这两个进程在单处理器上并发执行，那么可能的交叠执行序列就有很多种：

- 先执行A，后执行B，即 <α_1, α_2, α_3, β_1, β_2, β_3>，
- 先执行B，后执行A，即 <β_1, β_2, β_3, α_1, α_2, α_3>，
- 交叠执行，<α_1, β_1,α_2, β_2, α_3, β_3>
- 交叠执行，<β_1, α_1, α_2, β_2, β_3,α_3>
- ……

当多个进程并发执行时，会不会影响每个进程的执行结果呢？或者说，如何判断每个交叠执行的结果是正确的、一致的？我们分两种情况来考虑：

- 如果并发的进程是**无关的**，即每个进程分别在不同的变量集合上操作，那么一个进程的执行与其它进程的执行无关，这种情况下，任意交叠执行的结果都是一致的、正确的；
- 如果并发的进程是**交互的**，即它们共享某些变量或资源，或者它们的某些操作之间具有特定的顺序关系约束，那么一个进程的执行可能影响其它进程的执行，使得某些交叠执行是一致的、正确的，而另一些交叠执行是错误的、不可接受的。

对于发生交互关系的并发进程，需要对并发进程的操作执行顺序加以合理的控制，否则会出现不正确的结果。我们把这一问题称为交互进程的**并发控制问题**。进程之间交互的方式可以分为三类：

交互方式	特点
进程之间的资源竞争	这些进程通常独立工作，不知道其它进程的存在，进程间没有任何信息交换，不会相互合作，但是它们通过使用相同的资源，从而发生相互作用。我们把进程间的这种交互关系称为**资源竞争或竞争条件**。
进程之间通过共享对象合作	这些进程并不需要确切的知道对方进程的ID，但它们通过共享某些对象，如I/O缓冲区，从而发生相互合作，比如生产者-消费者问题。
进程之间相互通信	对于点对点通信，一个进程知道另一个进程的ID，对于广播通信，接收进程通常知道发送进程的ID。它们通过消息通信从而发生相互作用。

4.2 进程的互斥

【例4-1】假设一个飞机订票系统有两个终端，在每个终端上分别运行订票业务进程P1和P2。

```
cobegin
   P (1) || P(2)
coend
```

```
process P (int i )   (i=1,2)
int x;
begin
      read record to x;
      if x ⩾1 then x :=x-1;
      write x to record;
end
```

临界区

序列1	序列2
{*record*=1}	{*record*=1}
P1::read *record* to *x*	P1:: read *record* to *x*
{P1::*x*=1}	{P1::*x*=1}
P1::*x*:=*x*-1	P2:: read *record* to *x*
{P1::*x*=0}	{P2::*x*=1}
P1::write *x* to *record*	P1::*x*:=*x*-1
{*record*=0}	{P1::*x*=0}
{P1订到票}	P2::*x*:=*x*-1
P2::read *record* to *x*	{P2::*x*=0}
{P2::*x*=0}	P1:: write *x* to *record*
P2::write *x* to *record*	{*record*=0}
{*record*=0}	{P1订到票}
{P2未订到票}	P2:: write *x* to *record*
	{*record*=0}
	{P2订到票}

通过上面例子，可以总结出互斥并发控制问题应满足的条件：

- 一次至多一个进程能够在临界区内；
- 一个在非临界区终止的进程不能影响其它进程；
- 一个进程留在临界区中的时间必须是有限的；
- 不能强迫一个进程无限的等待进入它的临界区。特别的，进入临界区的任一进程不能妨碍正等待进入的其它进程；
- 当没有进程在临界区中时，任何需要进入临界区的进程必须能够立即进入；
- 对相关进程的执行速度和处理器的数目没有任何要求和限制。

解决互斥问题有三类方法：

- 第一类是软件方法，即不需要程序设计语言或操作系统提供任何支持来实施互斥，完全由并发执行的进程通过相互合作来实施互斥。这类方法会增加开销，并存在缺陷，或者说无法真正实现进程的互斥。但是，通过对这些方法的分析，可以更好的理解并发控制的复杂性；
- 第二类方法是通过硬件的支持来实现互斥，涉及到中断禁用、专用机器指令等。该方法的优点是开销小，但是由于其依赖于硬件，因此抽象层次较低，通用性较差；
- 第三类方法是在操作系统或程序设计语言层次，提供相应的支持。这是最常用的实现互斥的方法。下面我们依次介绍这些方法，重点是第三类方法。

算法1

/*P0*/	/*P1*/

……
while(turn !=0) skip;
/*critical section*/
turn:=1;
……
	……
	while(turn !=1) skip;
	/*critical section*/
	turn:=0;
	……

算法1存在的问题是:

- 一个进程依赖于另一个进程对turn变量的设置，因此P0和P1之间产生了交替执行，即要求(P0;P1)*或(P1;P0)*，这显然不满足互斥要求，使得一个进程不能再次进入临界区。

算法2

/*P0*/	/*P1*/
……	……
while(flag[1]) skip;	while(flag[0]) skip;
flag[0]:=1;	flag[1]:=1;
/*critical section*/	/*critical section*/
flag[0]:=0;	flag[1]:=0;
……	……

{flag[0]= 0,flag[1]=0}

P0::test flag[1]

P1::test flag[0]

P0::flag[0]:=1;

P0::enter critical section;

{flag[0]=1,flag[1]=0}

P1::flag[1]:=1;

P1::enter critical section;

{flag[0]= 1,flag[1]=1}

算法2的特点：

- 一个进程只对自己的标记进行设置，避免了算法1的问题
- 但是由于标记的<测试操作-设置操作>不是原子的，因此有可能两个进程都进入临界区

算法3

/*P0*/	/*P1*/

……
flag[0]:=1;
while(flag[1]) skip;
/*critical section*/
flag[0]:=0;
……
	……
	flag[1]:=1;
	while(flag[0]) skip;
	/*critical section*/
	flag[1]:=0;
	……

算法3的问题：

•<设置操作-测试操作>不是原子的，因此有可能两个进程都不能进入临界区

算法4

```
/*P0*/                                  /*P1*/
……                                    ……
flag[0]:=1;                           flag[1]:=1;
while(flag[1]) {                      while(flag[0]) {
  flag[0]:=0;                           flag[1]:=0;
  /*delay*/                             /*delay*/
  flag[0]:=1;                           flag[1]:=1;
}                                     }
/*critical section*/                  /*critical section*/
flag[0]:=0;                           flag[1]:=0;
……                                    ……
```

当另一进程在临界区时，当前进程首先礼让，然后在发起进入请求

算法4存在的问题：

- P0和P1相互谦让，使得两个进程都不能进入临界区

Peterson算法

```
int flag[2];
int turn;
```

```
void main( ){
   flag[0]:=0;
   flag[1]:=0;
   parbegin(P0, P1);
}
```

```
void P0( ){
   while(1){
      flag[0]:=1;
      turn:=1;
      while(flag[1] && turn=1) skip;
      /*critical region*/
      flag[0]:=0;
      ……
   }
}
```

```
void P1( ){
      while(1){
         flag[1]:=1;
         turn:=0;
         while(flag[0] && turn=0) skip;
         /*critical region*/
         flag[1]:=0;
         ……
      }
}
```

Peterson算法的正确性证明：

- 可重复进入：当一个进程没有进入临界区时的企图时，另一个进程可以多次重复进入，克服了算法1存在的问题；
- 互斥：当一个进程正在临界区时，另一个进程不能进入临界区；克服算法3存在的问题；
- 不存在相互谦让问题；克服算法4的问题；
- 当两个进程同时竞争进入临界区时，只允许其中一个进入。究竟哪个进程可以进入，取决于操作系统的调度。

Peterson算法

```
int flag[2];
int turn;
```

```
void main( ){
   flag[0]:=0;
   flag[1]:=0;
   parbegin(P0, P1);
}
```

```
void P0( ){
   while(1){
      flag[0]:=1;
      turn:=1;
      while(flag[1] && turn=1) skip;
      /*critical region*/
      flag[0]:=0;
      ……
   }
}
```

可以发生任意交叠

```
void P1( ){
      while(1){
         flag[1]:=1;
         turn:=0;
         while(flag[0] && turn=0) skip;
         /*critical region*/
         flag[1]:=0;
         ……
      }
}
```

4.2.3 解决互斥问题的硬件方法——中断禁用

在进程互斥问题中，如果能够保证一个进程进入临界区后，其执行是原子的，即不能被其它进程所中断，直到该进程退出临界区，那么自然就可以保证进程的互斥执行。这种能力可以通过系统内核启用或禁用中断来实现。一个进程可以通过下面的方法实现互斥：

```
while(1){
    ……
    /*禁用中断*/;
    /*临界区*/;
    /*启用中断*/;
    ……
}
```

```
const int n:=/*进程的个数*/;
int bolt;
void P(int i){
   while(1){
     while(compare&swap(bolt,0,1)=1) skip;
     /*临界区*/
     bolt:=0;
     ……
   }
}
main(){
   bolt:=0;
   parbegin(P(1),P(2),…,P(n));
}
```

专用机器指令

软件和硬件方法存在的缺点：

- 使用了忙等待（Busy waiting）。因此，当一个进程在等待进入临界区时，它会继续消耗处理器时间；
- 可能发生饥饿现象。当一个进程离开临界区，并且有多个进程正在等待时，选择哪一个等待进程是任意的，因此，某些进程可能被无限期的拒绝进入。
- 可能发生死锁。考虑单处理器中的下列情况：进程P1执行专门指令并进入临界区，然后P1可能被中断并把处理器让渡给具有更高优先级的P2。P2在试图进入临界区前，需要执行专门指令，并忙等在循环上。由于P2的优先级高于P1，其执行权不会被P1所抢占，因此P1永远不会得到调度执行，而P2也将永远忙等在循环上，于是死锁就发生了。

4.2.4 信号量和P、V操作

为了支持进程并发，我们希望并发控制机制满足下面三个需求：

（1）能够让一个进程阻塞或等待（waiting）在某个条件上。

（2）当一个进程所等待的条件发生变化时，该进程能够得到通知。

（3）能够被多个进程共享的条件变量。

1965年，荷兰计算机科学家Dijkstra首次提出了一个由操作系统提供的并发控制机制——**信号量**（Semaphore）。信号量实际上是一个抽象数据类型，包括一个由操作系统管理和维护的整数变量count、一个阻塞进程队列queue，以及作用在信号量上的一组操作。

P操作和V操作可表示为如下两个过程：

```
struct semaphore{
    int count;
    queuetype queue;
};
Procedure P(semaphore s){
    s.count:=s.count-1;
    if(s.count<0) Wait(s);
}
Procedure V(semaphore s){
    s.count:=s.count+1;
    if(s.count≤0) Release(s);
}
```

其中，Wait(s)和Release(s)是由操作系统提供的两个原语。

- Wait(s)：操作系统把调用P(s)的进程设置为阻塞态，并把它加入到信号量s的阻塞队列s.queue中，然后退出。
- Release(s)：操作系统从信号量s的阻塞队列s.queue中，按照某种策略挑选一个进程，把它从队列中移出，并把它从阻塞态变换为就绪态，加入就绪队列，然后退出；
- s.count的初值可以是任意整数值。

对于信号量，还需要注意如下几点：

•P、V操作必须是原子的。*如何保证P、V操作的原子性呢？*

•信号量s是由操作系统创建、管理和维护的一种系统资源，并通过唯一的ID进行管理和识别。当一个进程需要对信号量进行操作时，首先必须向操作系统申请获取该信号量。

•调用P操作可能使调用进程阻塞在条件s<0上。

•调用一次V操作可能使阻塞在s上的一个进程被唤醒（或被通知），从而进入就绪状态，但不一定能够马上得到调度执行，这取决于调度策略。

• 当存在多个进程阻塞在信号量上时，V操作究竟该选择哪一个进程解除阻塞呢？这取决于操作系统的调度策略。

•信号量必须在互斥的进程（或线程）间共享？线程间共享可以通过信号量句柄（全局变量）来实现，那么进程间如何共享信号量呢？（要通过named semaphore）

•信号量具有安全属性。

•信号量具有内核级别的持久性（kernel-persistence）。

4.2.5 使用信号量解决互斥问题

设有*n*个进程，用P(i)（*i*=1,2,…,*n*）来表示。每个进程进入临界区前执行P操作，退出临界区后执行V操作。

```
const int n:=/*进程数*/;
semaphore s:=1;
void P(int i){
   while(1){
      P(s);
      /*临界区*/
      V(s);
      /*其它代码*/
   }
}
void main( ){
    parbegin (P(1), P(2),…, P(n));
}
```

【例4-3】设一民航航班售票系统有n个售票处。每个售票处通过终端访问系统中的票务数据库。假定数据库中一些单元x_k(k=1,2,…)分别存放航班现有票数。设P1，P2，…，Pn分别表示各售票处的处理进程，R1，R2，…，Rn表示各进程执行时所用的工作单元。

```
1  semaphore s:=1;
2  void main( ){
3      parbegin (P(1), P(2), …, P(n));
4  }
5
6  void P(int i){
7      while(1){
8        /*按旅客订票要求找到 xk */;
9        P(s);
10       Ri=xk;
11       if Ri≥1 then {
12           Ri:=Ri-1;
13           xk:=Ri;
14           V(s);
15           /*输出一张票*/
16       }
17       else{
18           V(s);
19           /*输出“票已售完”*/
20       }
```

回顾

1、信号量是一个整数计数值，取值范围可以是任意整数。 T/F

2、信号量是位于用户空间的一项资源。T/F

3、P(s)操作一定使调用进程阻塞。T/F

4、V(s)操作一定会唤醒一个阻塞在s上的进程。T/F

5、使用忙等实现互斥有可能使系统发生死锁。T/F

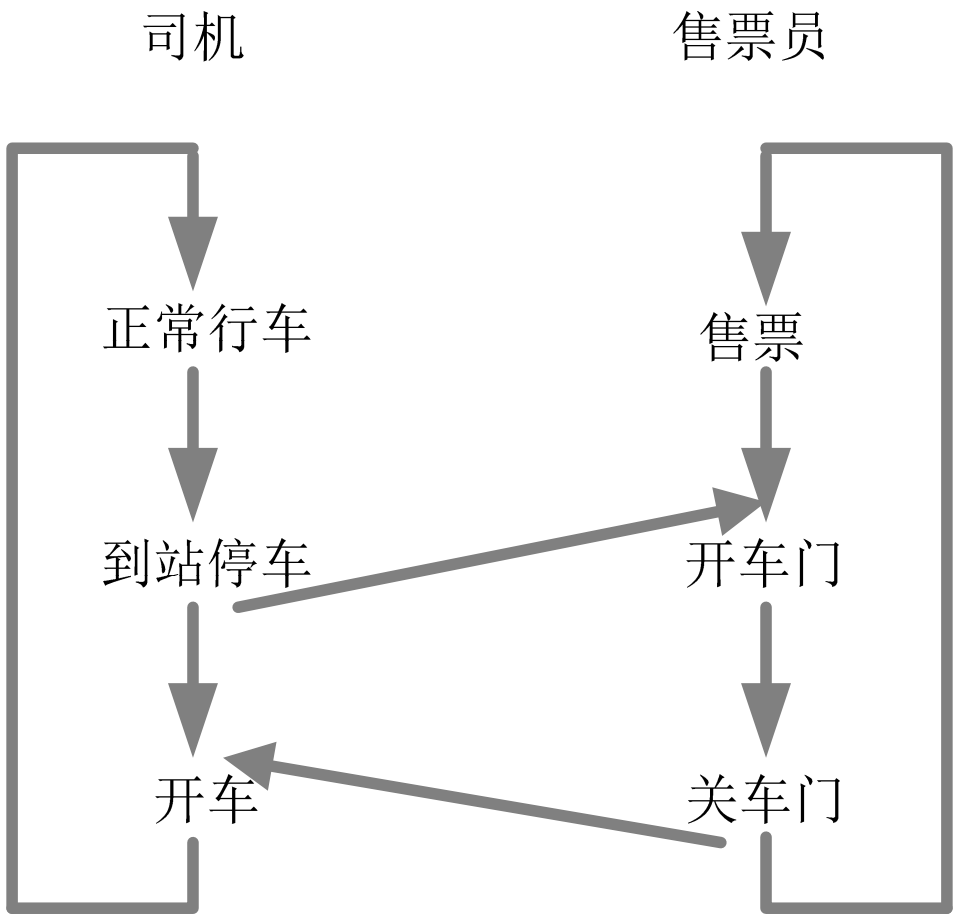
6、使用信号量实现互斥的设计模式是：每个进程在进入临界区之前，调用_____操作，在退出临界区之前调用_____操作，信号量的初值是_____。

秋色　X-T2 F5.6 1/70s ISO400 -67/100

崇仁 *X-T2 F2.0 1/40s ISO1600*

雨后和平门 *X-T2 F2.0 1/140s ISO1600*

§4.3 进程的同步



【例4-4】使用信号量实现司机和售票员进程的同步。

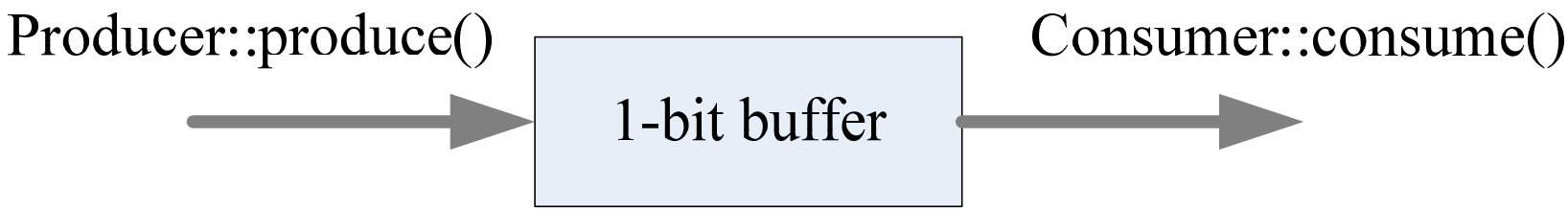
```
semaphore BusStop:=0;
semaphore DoorClose:=0;
void main( ){
  parbegin (driver( ), seller( ));
}
void driver( ){
    while(1){
        正常行车;
        到站停车;
        V(BusStop);          //signal bus stop
        P(DoorClose);        //wait for door close
        开车;
    }
}
void seller( ){
    while(1){
        售票;
        P(BusStop);      //wait for bus stop
        开车门;
        关车门;
        V(DoorClose);    //signal door close
    }
}
```

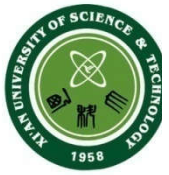
有两个同步点：

1、“到站停车”-> “开车门”

2、“关车门”->“开车”

【例4-5】（1-bit buffer问题）设有一个1位的缓冲区，每次只能放置一个产品。一个生产者不断生产产品并放入缓冲区，一个消费者不断从缓冲区中取出产品来消费。显然，由于缓冲区一次只能放置一个产品，于是限制了生产和消费的步调，即生产者生产一个产品后必须等待消费者消费之后才能进行下一轮生产，而消费者也必须等待生产者生产一个产品后才能消费。请用P、V操作实现这一同步过程。

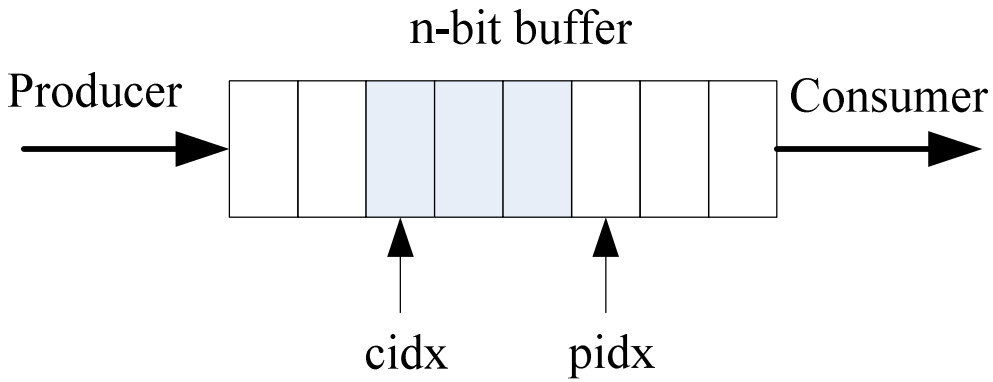




```
1   int buffer;
2   semaphore empty:=1;      //表示缓冲区中空位的个数
3   semaphore full:=0;       //表示缓冲区中产品的个数
4   void main( ){
5       parbegin (Producer( ), Consumer( ));
6   }
7   void Producer( ){
8       while(1){
9           //produce a message m;
10          P(empty);
11          buffer:=m;
12          V(full);
13      }
14  }
15
16  void Consumer( ){
17      while(1){
18          P(full);
19          m:=buffer;
20          V(empty);
21          //consume message m
22      }
23  }
24
25
```

4.4.1生产者-消费者问题

【例4-6】（1个生产者-1个消费者的n-bit buffer问题）设有一个n位的缓冲区，最多只能放置n个产品。一个生产者不断生产产品并放入缓冲区，一个消费者不断从缓冲区中取出产品来消费。由于缓冲区的容量有限，于是生产者和消费者的速度必须协调。请用信号量实现生产者和消费者的同步过程。



该问题中涉及互斥和同步并发控制问题：

- 生产者和消费者访问buffer时必须互斥；
- 当缓冲区为空时，消费者必须等待生产者；当缓冲区满时，生产者必须等待消费者。

为了实现互斥，需要一个初值为1的信号量mutex；为了实现同步，需要两个信号量empty和full，empty表示缓冲区中空位的个数，初始值为n；full表示缓冲区中产品的个数，初始值为0。

full>0 → Consume( )

empty>0 → Produce( )

```
int buffer[n];
semaphore mutex:=1;
semaphore empty:=n;  /*开始时空位个数为n*/
semaphore full:=0;    /*开始时产品个数为0*/
void main( ){
    parbegin (Producer( ), Consumer( ));
}
```

```
void Producer( ){
    int pidx:=0;
    while(1){
       produce a product;
       P(empty);
       P(mutex);
       buffer[pidx]:=product;
       pidx:=(pidx+1) mod n;
       V(full);
       V(mutex);
     }
}
```

```
void Consumer( ){
    int cidx:=0;
    int product;
    while(1){
        P(full);
        P(mutex);
         product:=buffer[cidx];
         cidx:=(cidx+1) mod n;
         V(empty);
         V(mutex);
         consume product;
    }
}
```

```
int buffer[n];
semaphore mutex:=1;
semaphore empty:=n;  /*开始时空位个数为n*/
semaphore full:=0;    /*开始时产品个数为0*/
void main( ){
    parbegin (Producer( ), Consumer( ));
}
```

```
void Producer( ){
    int pidx:=0;
    while(1){
      produce a product;
      P(empty);
      P(mutex);
      buffer[pidx]:=product;
      pidx:=(pidx+1) mod n;
      V(full);
      V(mutex);
    }
}
```

能否交换次序?

```
void Consumer( ){
    int cidx:=0;
    int product;
    while(1){
      P(full);
      P(mutex);
      product:=buffer[cidx];
      cidx:=(cidx+1) mod n;
      V(empty);
      V(mutex);
      consume product;
    }
}
```

```
int buffer[n];
semaphore mutex:=1;
semaphore empty:=n;  /*开始时空位个数为n*/
semaphore full:=0;    /*开始时产品个数为0*/
void main( ){
    parbegin (Producer( ), Consumer( ));
}
```

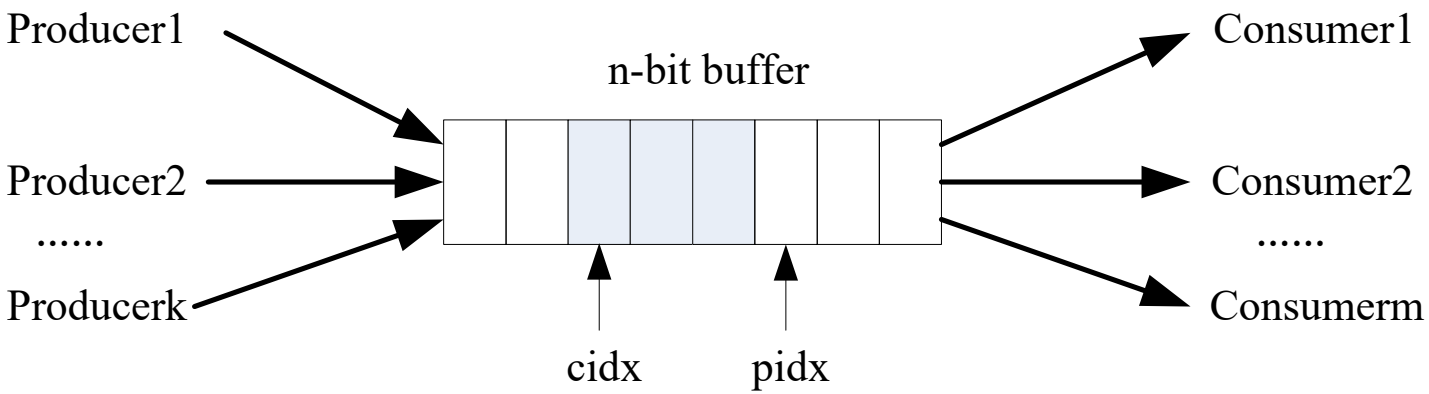
```
void Producer( ){
    int pidx:=0;
    while(1){
        produce a product;
        P(empty);
        P(mutex);
        buffer[pidx]:=product;
        pidx:=(pidx+1) mod n;
        V(full);
        V(mutex);
    }
}
```

能否去掉mutex?

```
void Consumer( ){
    int cidx:=0;
    int product;
    while(1){
        P(full);
        P(mutex);
        product:=buffer[cidx];
        cidx:=(cidx+1) mod n;
        V(empty);
        V(mutex);
        consume product;
    }
}
```

为什么pidx和cidx不可能指向同一buffer单元呢?

【例4-7】（k个生产者-m个消费者-n-bit buffer问题）设有一个n位的缓冲区，最多只能放置n个产品。多个生产者不断生产产品并放入缓冲区，多个消费者不断从缓冲区中取出产品来消费。请用信号量实现生产者和消费者的同步过程。



```
int buffer[n];
semaphore mutex:=1;
semaphore empty:=n;        /*开始时空位个数为n*/
semaphore full:=0;         /*开始时产品个数为0*/
int pidx:=0, cidx:=0;
void main( ){
    parbegin (Producer(1), …,Producer(k),      Consumer(1), …, Consumer(m));
}
```

```
void Producer(int i){    /*i=1,2,…,k*/
    while(1){
        produce a product;
        P(empty);
        P(mutex);
        buffer[pidx]:=product;
        pidx:=(pidx+1) mod n;
        V(full);
        V(mutex);
    }
}
```

```
void Consumer( int j){
    int product;
    while(1){
        P(full);
        P(mutex);
        product:=buffer[cidx];
        cidx:=(cidx+1) mod n;
        V(empty);
        V(mutex);
        consume product;
    }
}
```

4.4.2 读者-写者问题

【例4-8】有一个被多个进程共享的数据区，这个数据区可以是一个文件、一块内存空间或者是一组寄存器。有一些进程（Reader）只读取这个数据区中的数据，一些进程（Writer）只向数据区中写数据。此外，还必须满足以下条件：

- 任意多个读者进程可以同时读这个文件；
- 一次只能有一个写者进程可以写文件；
- 如果一个写进程正在写文件，禁止任何进程读该文件。

关键是要解决“**选择性互斥**”（Selective mutual exclusion）问题，即当存在一个读者进程正在读文件时，只互斥写进程；当存在一个写者进程正在写文件时，互斥所有其它进程。

为了实现选择性互斥，设计一个初值为0的变量nr，表示正在读取文件的读者进程的个数。当一个读者进程企图读取文件时，首先将计数nr递增，然后需要判断nr的值：

- 如果nr >1，表示已经有若干读者进程正在读取，于是该读者进程允许直接进入读取；
- 如果nr=1，说明这是第一个读者进程，但是可能有一个写者进程正在写入，因此读者需要与写者进行互斥。

```
semaphore rw:=1;     /*读者与写者互斥，或者写者与写者互斥*/
semaphore mutex:=1;
int nr:=0;           /*readers’ counter*/
void main( ){
    parbegin(Reader(1),…,Reader(m), Writer(1),…,Writer(n));
}
```

```
void Reader(int i){      /*i=1,2,…,m*/
    P(mutex);
    nr++;
    if(nr=1) then P(rw);
    V(mutex);
    read file;
    P(mutex);
    nr--;
    if(nr=0) then V(rw);
    V(mutex);
}
```

```
void Writer(int j){      /*j=1,2,…,n*/
    P(rw);
    write file;
    V(rw);
}
```

由于nr是读者进程的共享变量，因此对nr进行读写操作时，必须将它作为临界区互斥起来。

举一个反例。假如没有第6、9行的P、V操作，那么有可能出现这种交叠序列，于是就出现了读者和写者同时操作文件的情形，这显然是违背设计要求的。

{nr=0}

Reader1::nr++;

{nr=1}

Reader2::nr++;

{nr=2}

Reader1::if (nr=1);

Reader1::read file;

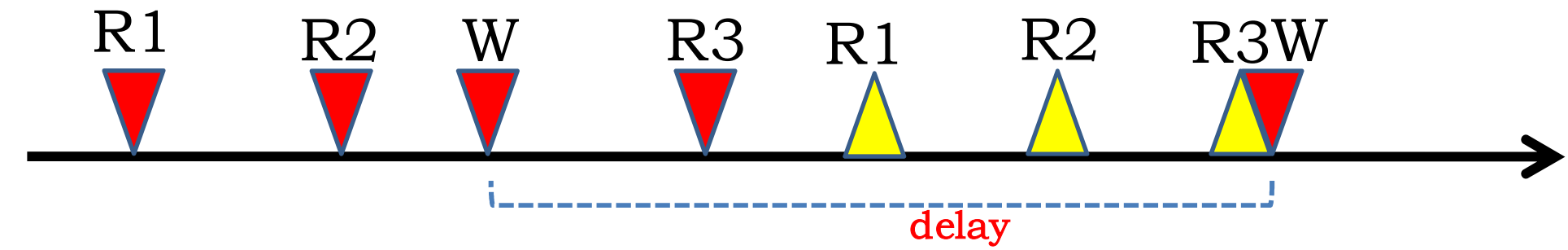
Writer1::**P**(rw);

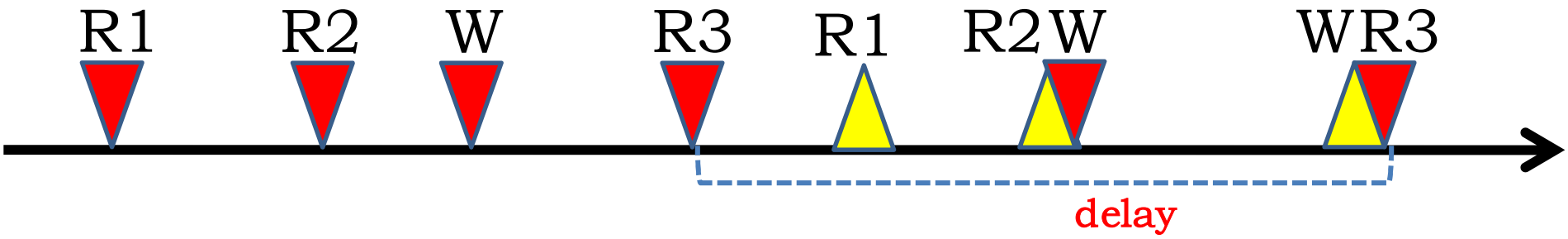
Writer1::write file

容易证明算法满足如下几个性质：

- 当nr>0时，其它读者进程可以允许直接进入；
- 当nr>0时，任何一个写者进程都不允许进入；
- 当一个写者进程进入并写入文件时，不允许任何读者进程和其它写者进程进入；
- 当一个读者进程等待进入时，其它试图进入的读者进程均需等待进入。

- “读者优先”策略：只要有一个读者进程正在读取文件，其它读者进程可以不加控制的进入读取，使得等待进入的写者进程迟迟不能进入，必须等待所有读者进程都退出之后才能进入。
- “写者优先”策略：希望写者优先，即当有读者进程在读取文件时，如果有写者进程请求写入，那么后来的读者进程必须被拒绝进入，待已经进入的读者完成读操作之后，立即让写者进入，只有当无写者工作时，才让读者进入读取。





```
1  semaphore sn:=n;        /*n 为读者进程的个数*/
2  semaphore mutex:=1;
3
4  void Reader(int i){
5      P(mutex);
6      P(sn);
7      V(mutex);                    /*{mutex=1 /\ 0≤sn<n}*/
8      read file;
9      V(sn);
10 }
11
12 void Writer(int j){
13     P(mutex);                    /*{mutex=0}*/
14     for(i=1, i≤n, i++) P(sn);    /*{mutex=0 /\ sn=0}*/
15     write file;
16     for(i=1, i≤n, i++) V(sn);
17     V(mutex);                    /*{mutex=1 /\ sn=n}*/
18 }
19
20 void main( ){
21     parbegin(Reader(1),…,Reader(m), Writer(1),…,Writer(n));
22 }
```

理解这段程序的关键是要证明程序满足这样一条性质：

> 当有k $(1 \leqslant k \leqslant n)$个读者进程正在读取文件时，一个写者进程请求写入文件，那么它必须等待这k个读者退出后才能进入，而且后续试图进入的读者进程必须等待，直到该写者进程退出。

下面简要证明这个性质。

当有k个读者进程正在读取文件时，两个信号量的状态为{mutext=1/\sn=n-k}。这时如果一个写者进程请求写入，它将执行第14行的for循环，经过n-k轮循环后，sn=0，于是写者进程阻塞在第n-k+1轮循环的P(sn)上，直到k个读者进程都执行完第9行的V(sn)之后，14行的循环才能退出，于是写者进程进入第15行代码执行文件写入。这时信号量的状态为{mutex=0/\sn=0}。从写者进程执行第13行的P(mutex)操作，一直到写入文件完毕这段时间内，所有试图进入的读者和写者进程均需等待，无法进入，直到第16行和17行执行完毕时，sn的状态恢复为n，mutex得到释放为止，这时的信号量状态为{mutex=1 /\ sn=n }。此状态允许等待的读者或写者进程进入。

4.6 死锁（Deadlock）

死锁是系统中多个进程并发执行时，由于资源占有和请求所引起的一种进程永远被阻塞的现象。通常认为死锁是由并发设计不当引起的，是设计过程中应当予以避免的一种负面现象。

在验证一个并发程序的正确性时，无死锁（deadlock freedom）通常是程序最基本的安全性需求之一。

- 作业

P144　12，　15，17 (a) ，20

下周四交作业

练习

问题描述：

1、图书馆阅览室最多能够容纳200名学生，若有更多学生想进入阅览室，必须等到阅览室中有同学退出之后才能进入。

2、每个学生在进入阅览室之前要刷卡，在退出阅览室之前也要刷卡，假定刷卡机只有一台，任一时刻只能有一名同学刷卡。

3、阅览室有一名管理员。早到的同学必须等管理员开门之后才能进入，管理员必须等到所有同学都退出之后才能关门。

请你用信号量实现上述问题。

问题分析

互斥问题：

（1）阅览室最多只能容纳200名学生，可以看做容量为200的临界区；

（2）刷卡机最多只能被一名同学使用，可以看做容量为1的临界区；

同步问题：

（1）早到的同学要等待管理员开门，发生一次同步；

（2）管理员要等待同学离开才能关门，发生第二次同步。

管理员进程（1个）：

```
Manager( ){

   while(1){

       openDoor();
       manage();
       closeDoor();

   }
}
```

学生进程（n个）：

```
Student(i){

   while(1){

       checkin();
       reading();
       checkout();
       rest();
   }
}
```

```
Semaphore  room:=200;
Semaphore  checker:=1;
```

管理员进程（1个）：

```
Manager( ){

   while(1){

       openDoor();
       manage();

       closeDoor();

   }
}
```

学生进程（n个）：

```
Student(i){

   while(1){

       P(room);
       P(checker);
       checkin();
       V(checker);
       reading();
       P(checker);
       checkout();
       V(checker);
       V(room);
       rest();
   }
}
```

```
Semaphore  room:=200;
Semaphore  checker:=1;
Semaphore  open:=0;
Semaphore  close:=0;
```

管理员进程（1个）：

```
Manager( ){

   while(1){

       openDoor();
       V(open);
       manage();

       P(close);
       closeDoor();

   }
}
```

学生进程（n个）：

```
Student(i){

   while(1){
       P(open);
       P(room);
       P(checker);
       checkin();
       V(checker);
       reading();
       P(checker);
       checkout();
       V(checker);
       V(room);
       V(close);
       rest();
   }
}
```

```
Semaphore  room:=200;
Semaphore  checker:=1;
Semaphore  open:=0;
Semaphore  close:=0;
int nr:=0    //阅览室中人数
```

管理员进程（1个）：

```
Manager( ){

   while(1){

      openDoor();
      V(open);
      manage();

      P(close);
      closeDoor();

   }
}
```

学生进程（n个）：

```
Student(i){

   while(1){
      if(nr=0)
         P(open);
      P(room);
       nr++;
      P(checker);
      checkin();
      V(checker);
      reading();
      P(checker);
      checkout();
      V(checker);
      V(room);
       nr--;
       if (nr=1)
          V(close);
      rest();
   }
}
```

```
Semaphore  room:=200;
Semaphore  checker:=1;
Semaphore  open:=0;
Semaphore  close:=0;
int nr:=0    //阅览室中人数
int wr:=0    //等在门外人数
```

管理员进程（1个）：

```
Manager( ){
   while(1){
      openDoor();
      V(open);
      manage();
      P(close);
      closeDoor();
   }
}
```

学生进程（n个）：

```
Student(i){
   while(1){
      if (nr=0)
         P(open);
      P(room);
       nr++;
      P(checker);
      checkin();
      V(checker);
      reading();
      P(checker);
      checkout();
      V(checker);
      V(room);
       nr--;
       if (nr=1)
```

4.6.1 死锁的定义

【例4-12】竞争资源产生死锁。

设系统有打印机、读卡机各一台，它们被进程P和Q共享。两个进程并发执行，它们按下列次序请求和释放资源：

进程P：	进程Q：
请求读卡机	请求打印机
请求打印机	请求读卡机
……	……
释放读卡机	释放读卡机
释放打印机	释放打印机

【例4-13】P、V操作使用不当产生死锁。

设进程P1和P2共享两个资源r1和r2。信号量s1和s2分别用来控制资源r1和r2的互斥访问。假定两个进程使用资源的方式如下：

进程P1：	进程P2：
P(s1);	P(s2);
P(s2);	P(s1);
使用r1和r2	使用r1和r2
V(s1);	V(s2);
V(s2);	V(s1);

- **死锁**：一组进程处于**死锁状态**是指：该组中每一个进程都在等待被另一个进程所占有的、不能抢占的资源。
- 注意：
  - 死锁是系统的一个状态而不是进程的状态。进程只具有就绪、运行和阻塞等基本状态，死锁状态与进程的阻塞状态有关，但不等同于阻塞状态。
  - 有时，系统中的进程能够运行（即处于运行态），但仍然会发生死锁。
  - 死锁的发生通常是有条件的。通常情况下，死锁只有在某些条件下才会发生，因此死锁的发现、再现和检测通常较为困难。

死锁是由并发执行的进程对共享资源的占有和请求所造成的，因此在讨论死锁问题时，我们首先对资源的占有和请求方式，以及程序的特性做一定假设，在此基础上才能准确的理解死锁现象：

•任意一个进程要求资源的最大数量不超过系统能提供的最大量。

•如果一个进程在执行中所提出的资源要求能够得到满足，那么它一定能在有限的时间内结束。

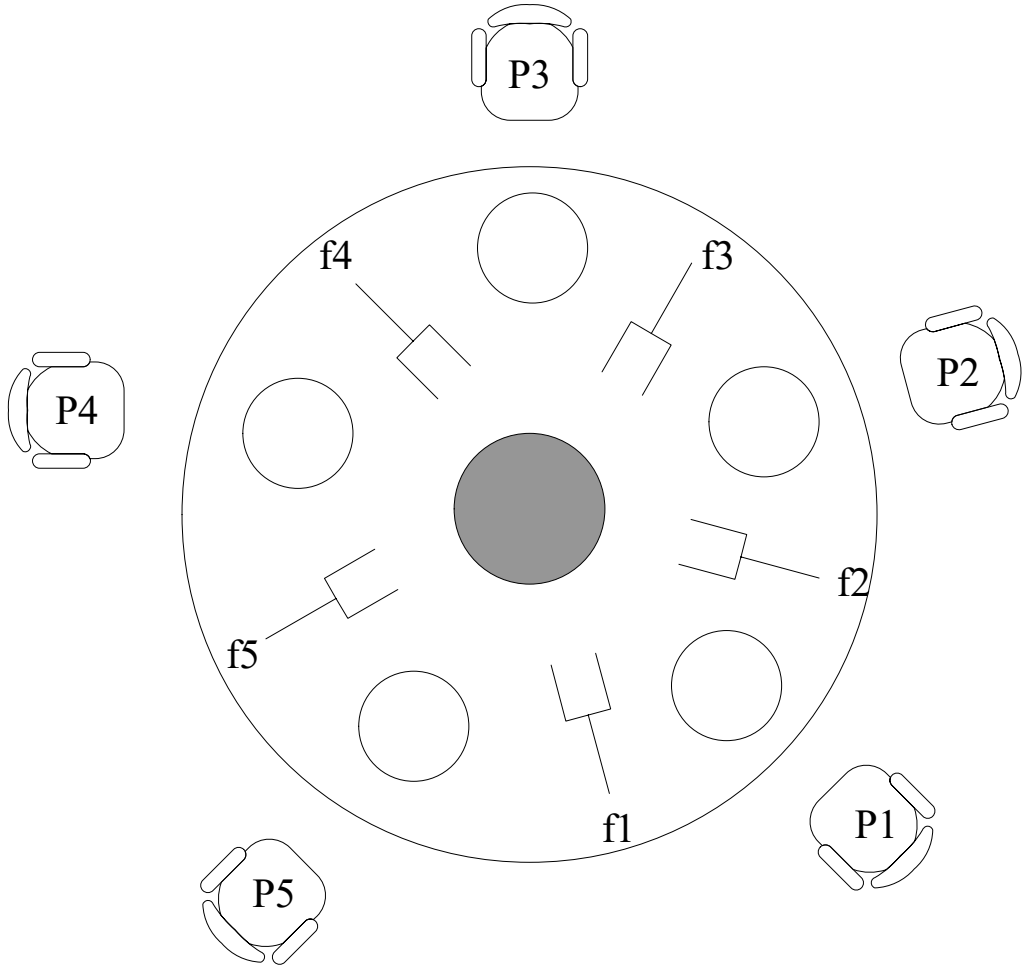
•一个资源在任何时刻最多只为一个进程所占有。

•一个进程一次申请一个资源，且只在申请资源得不到满足时才处于等待状态。换言之，其它一些等待状态，如人工干预、等待外围设备传输结束等，在没有故障的条件下，可以在有限长的时间内结束，不会产生死锁。因此，这里不考虑这种等待。

•一个进程结束时，释放它占有的全部资源。

•系统具有有限个进程和资源。

4.6.2 哲学家就餐问题



```
semaphore fork[5]:={1};
int i;
void P (int i){
    think( );
    P(fork[i]);
    Pick up f[i];
    P(fork[(i+1) mod 5]);
    Pick up f[(i+1) mod 5];
    eat();
    Put down f[i];
    Put down f[(i+1) mod 5]
    V(fork[i]);
    V(fork[(i+1) mod 5]);
}

void main( ){
    parbegin(P(1), P(2), …, P(5));
}
```

P1::**P**(fork[1]);pick up f[1];

P2::**P**(fork[2]);pick up f[2];

P3::**P**(fork[3]);pick up f[3];

P4::**P**(fork[4]);pick up f[4];

P5::**P**(fork[5]);pick up f[5];

P1::**P**(fork[2]);

{P1阻塞}

P2::**P**(fork[3]);

{P2阻塞}

P3::**P**(fork[4]);

{P3阻塞}

P4::**P**(fork[5]);

{P4阻塞}

P5::**P**(fork[1]);

{P5阻塞}

{系统死锁}

为了确保任何调度情况下都不会发生死锁，

问题1：在进程数和每个进程所需资源数一定的情况下，系统需要提供最少多少个资源?

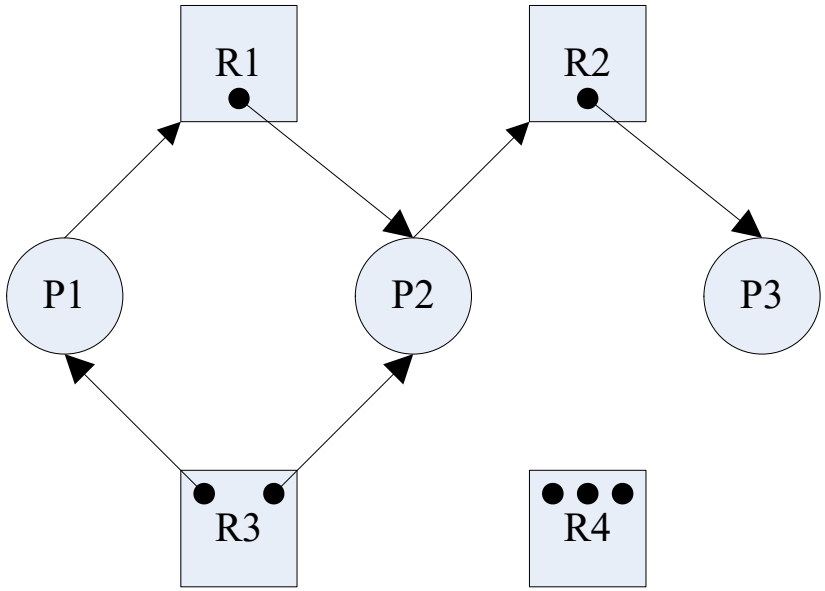
问题2：在系统资源总数和每个进程所需资源数一定的情况下，最多允许的进程个数为多少?

设系统中同类资源的个数为m，由n个进程互斥使用，每个进程对该类资源的最大需求量为k。为了保证在任何调度情况下，系统都不会发生死锁，那么m、n和k必须满足如下条件：

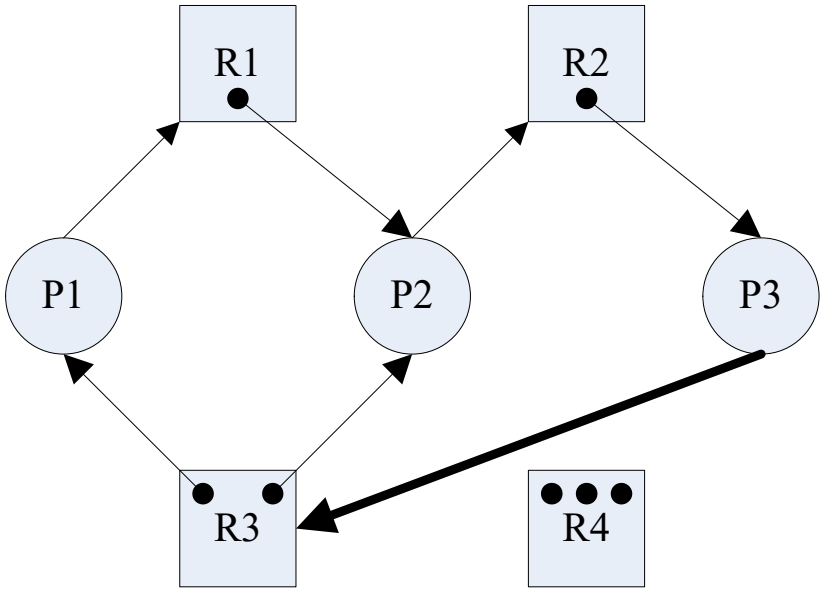
$$n\times(k-1)+1\leqslant m$$

4.6.3 死锁的描述

用资源分配图（resource allocation graph）描述系统资源和进程的关系，每个资源和进程用节点来表示。资源节点中，一个圆点表示资源的一个实例。从进程节点指向资源节点的边表示进程请求该资源，但是还没有得到授权。从资源节点中的一个圆点到进程的边表示资源请求已经被授权，或者该资源已经被进程所占有。

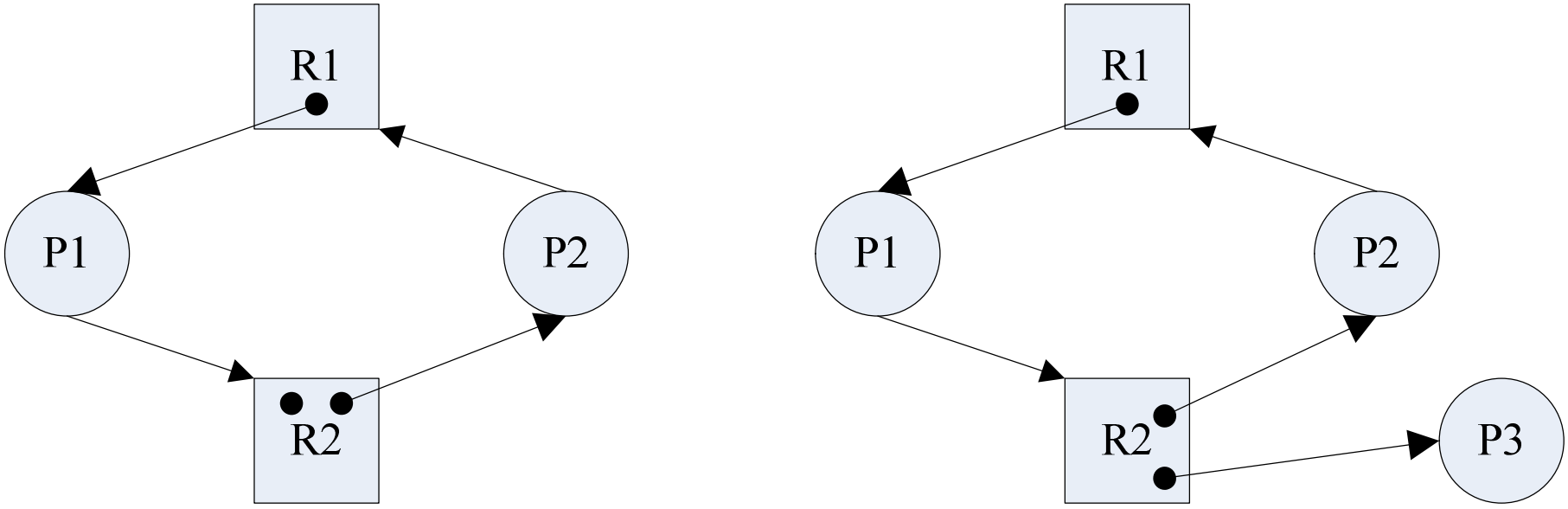


a)



b)

如果资源分配图中不存在环路，那么系统一定无死锁；如果资源分配图中存在环路，那么系统**可能**死锁。



资源分配图中存在环路是死锁的**必要条件**，即如果系统发生了死锁，那么资源分配图中一定存在环路，或者说，如果资源分配图中不存在环路，那么系统一定无死锁。但是如果资源分配图中存在环路，那么不能保证系统一定死锁。

死锁的发生有如下必要条件：

1. **互斥**。一个资源一次只能被一个进程所使用，如果有其它进程请求该资源，那么请求进程必须等待，直到该资源被释放。
2. **占有且等待**。一个进程请求资源得不到满足而等待时，不释放已占有的资源。
3. **不可抢占**。一个进程不能强行从另一个进程那里抢夺资源，即已被占用的资源，只能由占用进程自己来释放。
4. **循环等待**。在资源分配图中，存在一个循环等待链，其中每一个进程分别等待它的前一个进程所持有的资源。

4.7 死锁的处理

死锁是一种不期望发生的系统状态，应当予以必要的处理。处理死锁的方法有三个层次：预防、避免和检测。

- 死锁预防（deadlock prevention）是指采用某种策略来消除条件1至条件4中的一个条件的出现，使得死锁的条件不再成立，从而保证死锁不会发生；
- 死锁避免（deadlock avoidance）允许前三个必要条件，但是基于当前的资源分配状态，选择接受或拒绝当前资源分配请求，确保系统状态不会到达死锁点；
- 死锁检测（deadlock detection）则不限制资源访问或约束进程行为，只要有可能，尽量授予进程请求的资源。操作系统周期性地执行一个算法检测死锁发生，若发生死锁，则把系统状态恢复到死锁前某个一致或正确的状态上。

4.7.1 死锁预防

静态分配（预分配）

所谓静态分配是指一个进程必须在执行前就申请到它所需全部资源，并且直到它所要的资源都得到满足之后才开始执行。采用静态分配后，进程在执行中不再申请资源，因而不会出现占有了某些资源同时等待另一些资源的情况，即破坏了第二个条件的出现。静态分配策略实现简单，因而被许多操作系统采用，例如OS/360。

顺序分配策略

循环等待条件可以通过定义申请资源的顺序来消除。该方法的基本思想是对系统的全部资源加以全局编号，然后规定进程申请资源时，必须按照编号的特定顺序来申请。

4.7.2 死锁避免

一个死锁避免方法是**银行家算法**，最初是由Dijkstra在1965年提出的。Dijkstra把系统比作一个银行家，它占有有限资金（资源）。银行家不可能满足所有借款人（进程）的最大需求量总和，但可以满足一部分借款人的借款需求。待这些人的借款归还后，又可把这笔资金借给他人。这样，当一借款人提出借款要求时，银行家就要进行计算，以决定是否借给他，看是否会造成银行家的资金被借光而使资金无法周转（死锁）。

考虑一个具有n个进程和m种不同类型资源的系统。为了描述系统的资源分配状态，需要引入如下几个向量和矩阵。

Resource$=(R_1,R_2,\ldots,R_m)$	系统中每种资源的总量
Available$=(V_1,V_2,\ldots,V_m)$	剩余的每种资源的总量。初始时 ***Available=Resource***
Claim$=\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	进程最大资源需求矩阵。C_{ij}表示进程 i 对资源 j 的最大需求。显然，必须满足 $C_{1j}+C_{2j}+\ldots+C_{nj} \le R_j$
Allocate$=\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	资源分配矩阵。A_{ij}表示当前分配给进程 i 的资源 j 的数目。显然，对任意 i 和 j，必须有 $A_{ij} \le C_{ij}$
Need=Claim-Allocate	每个进程仍需请求的资源矩阵。

假设系统中有4个进程（P1、P2、P3和P4）和3类资源。

设*Resouce*和*Claim*为：

Resource=(9, 3, 6)　*Claim* = $\begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix}$

假设系统运行到一定阶段时，资源分配状态为：

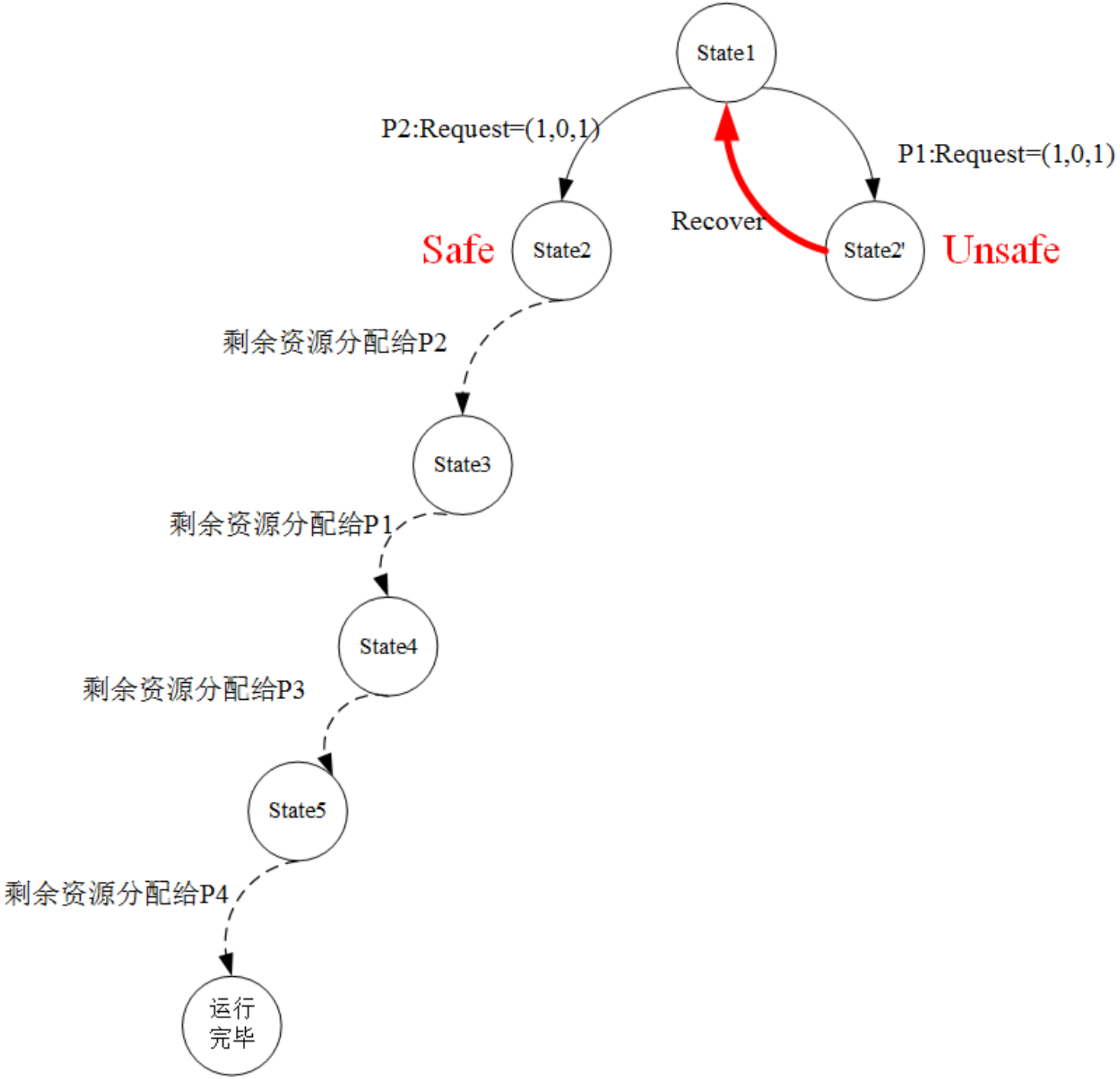
State1：*Allocate*= $\begin{pmatrix} 1 & 0 & 0 \\ 5 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$，则 *Need*= $\begin{pmatrix} 2 & 2 & 2 \\ 1 & 0 & 2 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix}$，*Available*=(1, 1, 2)

假设在此状态下，P2请求一个R1资源和一个R3资源，即*Request*=(1, 0, 1)。如果假定同意该请求，则资源分配状态变为：

$$\textbf{State2：}\ \textit{Allocate}=\begin{pmatrix}1&0&0\\6&1&2\\2&1&1\\0&0&2\end{pmatrix},\ \text{则}\ \textit{Need}=\begin{pmatrix}2&2&2\\0&0&1\\1&0&3\\4&2&0\end{pmatrix},\ \textit{Available}=(0,1,1)$$

如果State2是安全的，则接受该请求，如果不是安全的，则拒绝该请求。

那么下面就来考察State2是否安全，即考察从该状态出发，是否存在一个资源分配序列使系统中的所有进程资源需求都能满足，进而运转下去。


State1
P2:Request=(1,0,1)
P1:Request=(1,0,1)
Safe
State2
Recover
State2'
Unsafe
剩余资源分配给P2
State3
剩余资源分配给P1
State4
剩余资源分配给P3
State5
剩余资源分配给P4
运行
完毕

银行家算法：

设 $Request_i$ 是进程 P_i 的资源请求向量，$Request_i[j]=k$ 表示进程 P_i 请求 k 个资源 j 的实例。在当前资源分配状态 *Allocate* 下，当进程 P_i 发出一个资源请求时，银行家算法的步骤如下：

1. 如果 $Request_i \le Need_i$，那么执行步骤 2。否则，产生一个错误条件，因为进程的资源请求已经超过了它的资源需求总量。
2. 如果 $Request_i \le Available$，执行步骤 3。否则，P_i 必须等待，因为目前剩余资源不能满足它的需求。
3. 系统尝试接受该请求，并实施资源分配，那么资源分配状态改变为：

$$Allocate_i := Allocate_i + Request_i$$

$$Need_i := Need_i - Request_i$$

$$Available = Available - Request_i$$

判断新资源状态 *Allocate* 是否安全。如果新资源状态是安全的，则实际实施资源分配；如果新资源状态是不安全的，则 P_i 必须延迟请求，并且把系统状态恢复到原来状态上。

（1）设***Work***[*m*]和***Finish***[*n*]分别是长度为*m*和*n*的向量。开始时，

Work :=***Available***，***Finish***[*i*] :=false，*i*=1,2,…,*n*。

（2）在***Need***矩阵中找一个*i*，使得

$$\boldsymbol{Finish}[i]=\text{false} \wedge \boldsymbol{Need}_i \leq \boldsymbol{Work}$$

如果这样的*i*不存在，执行步骤4。

（3）***Work*** := ***Work***+***Allocate***$_i$

Finish[*i*] :=true

返回步骤2。

（4）如果对于所有的*i*=1,2,…,*n*，都有***Finish***[*i*]=true，说明在状态***Allocate***下，存在一条资源分配序列，使系统运行完毕，则状态***Allocate***是安全状态；否则，***Allocate***状态是不安全状态。

（5）如果在第2步中，存在多个*i*满足条件，无论选择哪一个都不会影响判断结果。换句话说，该算法在状态树上搜索时，选择的任意一条路径都会得到同样的结果，不需要回溯。

- 评价：

银行家算法比起资源静态分配法，资源利用率提高了，又避免了死锁。但它有这样几个不足：

- 对资源分配过于保守，没有考虑到进程获得资源后，虽然未达到其最大需求量，也可能把它释放；
- 算法计算量较大。每次申请都要经过计算已决定是否同意分配；
- 必须事先知道进程对资源的最大需求量，这往往是不实际的。

4.7.3 死锁检测

死锁检测的基本思想是，不限制资源访问请求或约束进程行为，只要有可能，请求的资源就被授予给进程。操作系统周期性的执行一个算法检测死锁是否发生。若检测到死锁，则设法加以解除。

死锁检测算法

下面描述一个常见的死锁检测算法，它使用了上一节介绍的资源分配矩阵*Allocate*，未分配的资源向量*Available*，此外还需要定义一个资源请求矩阵*Request*，其中$Request_{ij}$表示进程 i 请求资源*j*的数量。算法的工作过程与前面的银行家算法有些类似，但是其目的是要检测当前状态是否会导致死锁。

死锁检测算法

1. 设 ***Work***[m]和 ***Finish***[n]分别是长度为 m 和 n 的向量。开始时，***Work***:=***Available***，对于 i=1,2,…,n，如果***Allocate***$_i$≠**0**，那么 ***Finish***[i] :=false；否则 ***Finish***[i] :=true。
2. 在 ***Request*** 矩阵中找一个 i，使得

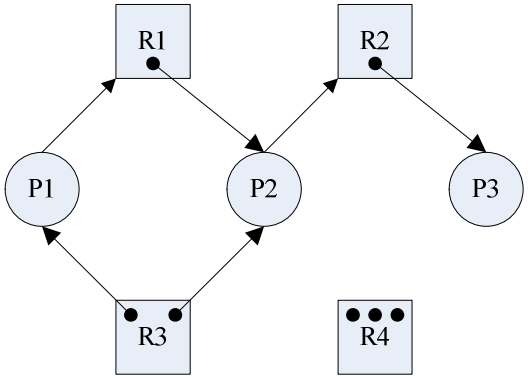
$$\textbf{\textit{Finish}}[i]=\text{false} \wedge \textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$$

如果找不到这样的 i，则执行步骤 4。

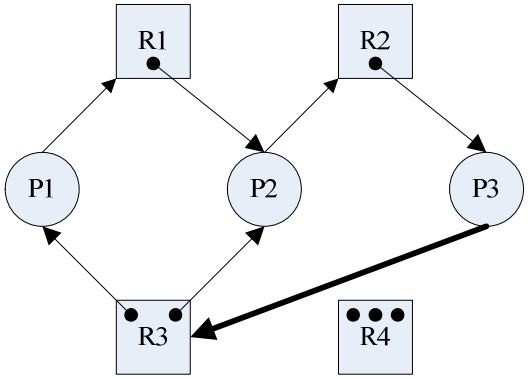
3. ***Work*** := ***Work*** + ***Allocate***$_i$

   Finish[i] :=true

   返回步骤 2。
4. 如果存在某个 i，$1 \leq i \leq n$，***Finish***[i] =false，说明剩余资源无法满足进程 P_i 的资源需求，那么系统状态是死锁的，而且 P_i 是死锁进程；否则，说明该状态不是死锁状态。
5. 如果在第 2 步，存在多个 i 满足条件，无论选择哪一个都不会影响判断结果。



a)



b)

	初始状态	State1	State2	State3
Allocate	$\begin{pmatrix}0&0&1&0\\1&0&1&0\\0&1&0&0\end{pmatrix}$	$\begin{pmatrix}0&0&1&0\\1&0&1&0\\-&-&-&-\end{pmatrix}$	$\begin{pmatrix}0&0&1&0\\-&-&-&-\\-&-&-&-\end{pmatrix}$	$\begin{pmatrix}-&-&-&-\\-&-&-&-\\-&-&-&-\end{pmatrix}$
Work	(0,0,0,3)	(0,1,0,3)	(1,1,1,3)	(1,1,2,3)
Request	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\end{pmatrix}$	$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\end{pmatrix}$
Finish	Finish[*i*]=false, *i* =1,2,3	Finish[3]=true Finish[*i*]=false,*i*=1,2	Finish[3]=true, Finish[2]=true, Finish[1]=false	Finish[i]=true,i=1,2,3

死锁的恢复

一旦检测到死锁，就需要某种策略把系统恢复到死锁前的一个正确状态上去。下面按复杂度递增的顺序列出可能的方法：

- 取消所有的死锁进程，这是操作系统最常用的方法；
- 把每个死锁进程回滚到前面定义的某些检查点（Checkpoint），并且从这些检查点重新执行所有进程。这要求在系统中构造回滚和重启机制。该方法的风险是原来的死锁可能再次发生。但是，并发进程的不确定性通常能够保证不会发生这种情况；
- 连续取消死锁进程，直到不再存在死锁。选择取消进程的顺序基于某种最小代价原则。在每次取消后，必须重新调用检测算法，以测试是否仍存在死锁；
- 连续抢占资源，直到不再存在死锁。和前面取消进程的策略一样，需要使用一种基于代价的选择方法，并且需要在每次抢占后重新调用检测算法。一个资源被抢占的进程，必须回滚到获得这个资源之前的某一状态。

- 习题

P143　2、12、17（b）、19