死锁

死锁的例子

- 1、哲学家吃饭问题

当n个哲学家**同时饥饿**时。。。。死锁!

- 2、PC问题

生产者：先执行　P(s)后执行P(empty)：在有界缓冲**全空**的时候 死锁!

或者

消费者 先执行 P(s)后执行P(full)时：在有界缓冲**全满**时 死锁!

死锁的特点

1、**小概率**事件：同一批进程，同一套资源集合，并发执行多次，只有有限的几次会死锁

2、一旦发生，后果十分严重！会在系统范围内引发**多米诺骨牌**效应，导致更多进程停止，更多资源被占用而不得释放，系统效率大大降低

想办法解决!

1.死锁概念

- 指多个进程因竞争资源而造成的一种僵局，若无外力作用，这些进程都将永远不能再向前推进。
- 即：一组进程中，每个进程都无限等待被该组进程中另一进程所占有的资源，因而永远无法得到的资源，这种现象称为进程死锁，这一组进程就称为死锁进程。

产生死锁的原因

1.系统资源不足（系统能提供的资源集合远小于并发进程主要的资源之和）

2.进程的推进顺序不当（引发小概率事件）

解决办法

从死锁的原因找解决办法

1、资源不足：**增加资源？**

No!

代价大，无止境

2、推进顺序非法，调整顺序？

yes!

代价小，灵活。 **but, how?**

死锁的必要条件

- 互斥：**进程竞争临界资源**
- 保持并请求（部分分配）：进程已**持有**一部分资源，为了顺利运行，还需要申请另一部分资源
- 不可剥夺条件：**已分配给进程的资源，在进程使用完毕主动释放之前不得强制剥夺**
- 环路条件：**系统的资源分配图中出现了环路**

资源分配图

1、节点：进程节点和资源节点

2、有向边：进程指向资源的请求边

资源指向进程的分配边

以两个哲学家吃饭为例，死锁发生的时候：

```
p1
{

p(s1)

p(s2)

吃饭
v(s1)
v(s2)

}
```

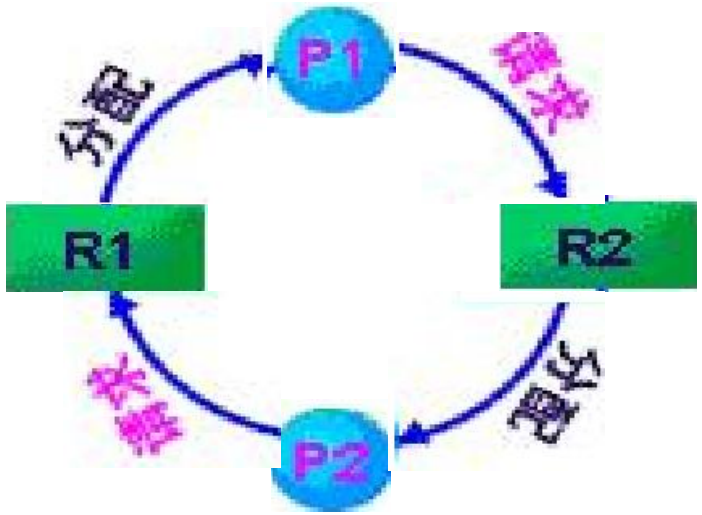
```
p2
{

p(s2)

p(s1)

吃饭
v(s1)
v(s2)

}
```



思路一：死锁预防

定理成立，则定力的逆否定理也成立

通过破坏死锁的四个必要条件之一，使得死锁的必要条件永远不会成立，则死锁不会产生

其中互斥条件一般不破坏，不仅不破坏，还需要采用手段尽量保证。所以后面讨论其他三个条件的破坏方法。

A、破坏“保持并请求条件”：

- 系统要求所有进程要**一次性地申请**在进程整个推进过程中所需要的全部资源。这样该进程在运行期间，将摒弃请求条件，不会再提出资源要求，因而不会发生死锁。（**静态资源分配**法）

优点：简单且安全

缺点：

1、进程难以一次性地提出全部资源要求；

2、只要有一种资源不能满足该进程的分配要求，其它资源也全部不分配给该进程而让进程等待，能够获得全部资源开始运行的进程数减少，系统并发度降低；

3、某些资源可能进程仅仅最后阶段才使用，或者只使用一个短暂时间，也必须一开始就分配给它独占，造成资源严重浪费。**资源利用率低**

B、破坏“不剥夺条件”：

- 采用这种方法时，不要求进程一次性地提出全部资源要求，进程可以在只满足当前资源要求的情况下运行，在需要新的资源时才提出请求。
- 但是一个已经保持了某些资源的进程，当它再提出新的要求而不能立即满足时，必须释放它占有的所有资源，待以后需要时再重新提出申请。这是一种动态的分配方法，可以减少资源被长时间独占且闲置，因而提高资源利用率。

缺点：进程放弃已经占用但尚未用完的资源可能要付出很大的代价。**动态分配比较复杂**，同时也增加了**系统的开销。进程可能反复申请、撤销资源**

C、破坏“环路等待条件”：

- 系统将所有资源都编上唯一的号，申请资源必须严格按**资源递增的顺序**提出，这样在所形成的资源分配图中，不可能再出现环路。（**有序资源申请法**）
- 优点：提高了资源利用率和系统的吞吐量。
- 缺点：

1、难以照顾所有用户的编程习惯；

2、按规定次序申请资源的方法，可能会限制用户自由的编程思路；

3、为系统中各种类型资源所分配的序号，必须相对稳定，这就限制了添加新类型设备的方便性。

4、 **资源利用率下降**： 资源使用顺序与资源申请顺序不一致，导致资源不能充分利用

一个进程获取资源的规律：

动态执行，按需申请

a. 申请资源

b. 使用资源

c. 释放资源

显然思路一（死锁预防）的办法违背了进程使用资源的规律，强加了一些限制条件（静态分配、有序申请），导致资源利用率下降

思路二 死锁避免

- **死锁避免定义**:在系统运行过程中，对进程发出的每一次**资源申请**进行**动态评估**，并根据评估结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

- 由于在避免死锁的策略中，允许进程动态地**按需**申请资源。因而，系统在进行资源分配之前预先**计算资源分配的安全性**。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

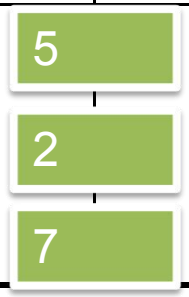
3. 安全状态与不安全状态

安全状态指系统能按某种进程顺序{P_1，…，P_n}来为每个进程P_i($1 \leqslant i \leqslant n$)分配其所需资源，直至最大需求，使每个进程都可顺序完成。若系统不存在这样一个序列，则称系统处于不安全状态。如果存在，则称序列<P_1，…，P_n>为安全序列。

安全状态之例

假定系统中有三个进程P_1、P_2和P_3，共有12台磁带机。进程P_1总共要求10台磁带机，P_2和P_3分别要求4台和9台。假设在T_0时刻，进程P_1、P_2和P_3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

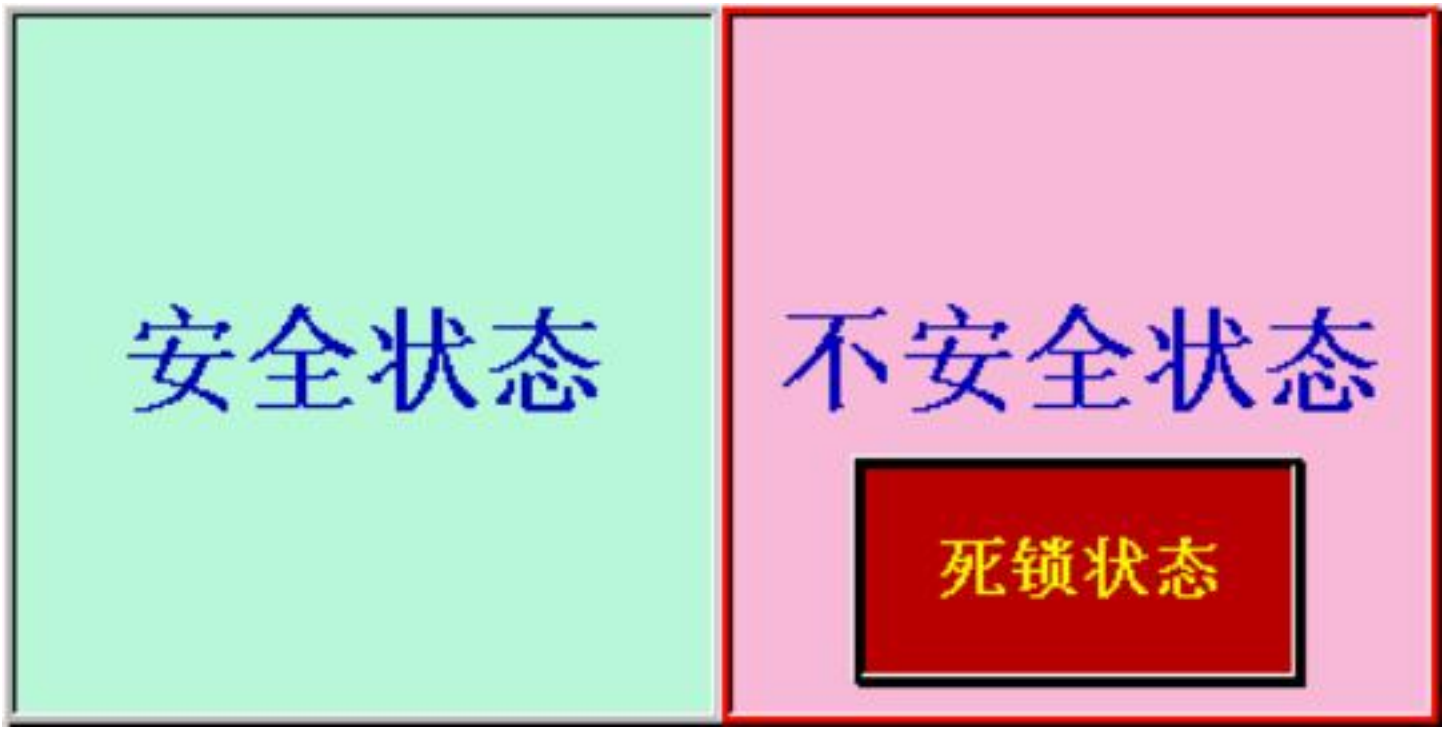
进 程	最 大 需 求	已 分 配	可 用
P_1	10	5	3
P_2	4	2	
P_3	9	2	



问：T_0时刻是否安全?

安全状态与不安全状态

不安全状态:不存在一个安全序列，不安全状态不一定导致死锁



4.利用银行家算法避免死锁

1)银行家算法中的数据结构

(1) **可利用资源向量Available**。这是一个含有*m*个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available［j］=K，则表示系统中现有R_j类资源*K*个。

avilable = [4，3，7]表示一共有三类资源，每一类的可用数量分别是4个，3个，7个

(2) **最大需求矩阵Max**。这是一个$n \times m$的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果Max［i，j］=K，则表示进程i需要R_j类资源的最大数目为K。

max =

3	3	7
2	0	1
5	1	4
6	4	2

allocation =

2	0	1
1	0	0
1	0	1
2	2	2

need =

1	3	6
1	0	1
4	1	3
4	2	0

(3) **分配矩阵Allocation**。这也是一个$n \times m$的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i，j］=K，则表示进程i当前已分得R_j类资源的数目为K。

(4) 需求矩阵Need。这也是一个$n \times m$的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i，j］=K，则表示进程i还需要Rj类资源K个，方能完成其任务。

Need［i，j］=Max［i，j］-Allocation［i，j］

2)银行家算法

设**Request$_i$**是进程P_i的请求向量，是一个m元向量，例如$Request_i$=（2，0，1），表示进程P_i需要*2个R1，0个R2，1个R3*资源。当P_i发出资源请求后，系统按下述步骤进行检查：

(1) 如果$Request_i$［j］≤Need［i，j］，便转向步骤2；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值(判断资源申请的合法性，不能超过最大需求)。

(2) 如果$Request_i$［j］≤Available［j］，便转向步骤(3)；否则，表示尚无足够资源，P_i须等待（判断操作系统进行资源分配的能力）。

(3) 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：

Available［j］：=Available［j］-$Request_i$［j］；

Allocation［i，j］：=Allocation［i，j］+$Request_i$［j］；

Need［i，j］：=Need［i，j］-$Request_i$［j］；

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程P_i，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程P_i等待。

3)安全性算法

(1) 设置两个向量:

① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有*m*个元素，在执行安全算法开始时，Work：=Available;

② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］：=false; 当有足够资源分配给进程时，再令Finish［i］：=true。

(2) 从进程集合中找到一个能满足下述条件的进程：① Finish［i］=false; ② Need［i，j］≤Work［j］； 若找到，执行步骤(3)， 否则，执行步骤(4)。

(3) 当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］：=Work［i］+Allocation［i，j］；

Finish［i］：=True;

go to step 2;

(4) 如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

4) 银行家算法之例

假定系统中有五个进程 {P_0, P_1, P_2, P_3, P_4} 和三类资源 {A, B, C}，各种资源的数量分别为10、5、7，在T_0时刻的资源分配情况如图 3-16 所示。

available = (10,5,7) - (7,2,5) = (3,3,2)

	Max(A B C)	Allocate(A B C)	Need(A B C)	work 3,3,2	finish
P0	7, 5, 3	0, 1, 0	7, 4, 3	7,5,3	T 3
P1	3, 2 , 2	2, 0, 0	1, 2, 2	5,3,2	T 1
P2	9, 0, 2	3, 0, 2	6, 0, 0	10,5,5	T 4
P3	2, 2, 2	2, 1, 1	0, 1, 1	7,4,3	T 2
P4	4, 3, 3	0, 0, 2	4, 3, 1	10,5,7	T 5
	25,12,12	7, 2, 5			

P1,P3,P0,P2,p4为所求安全序列！ 安全!

P_1请求资源：P_1发出请求向量$Request_{1=}$(1，0，2),

(1)请求合法性(1,0,2) < (1,2,2) 请求合法!

(2)os能力判断（1，0,2 ）< (3,3,2) OS有能力满足进程请求

（3）试分配

	Max(A B C)	Allocate(A B C)	Need(A B C)	work 2，3，0	finish
P0	7，5，3	0，1，0	7，4，3	10,5,7	T 5
P1	3，2，2	3，0，2	0，2，0	5,3,2	T 1
P2	9，0，2	3，0，2	6，0，0	10,4,7	T 4
P3	2，2，2	2，1，1	0，1，1	7,4,5	T 3
P4	4，3，3	0，0，2	4，3，1	5,3,4	T 2

P1,p4,p3p2,p0为所求安全序列！ 系统状态是安全的!

(3) P_4请求资源：P_4发出请求向量$Request_4=(3, 3, 0)$，系统按银行家算法进行检查：

① $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$；请求合法!

② $Request_4(3, 3, 0) \not\leq Available(2, 3, 0)$，

OS没有能力满足进程的本次资源申请!

os拒绝P4请求!

（4） P0请求资源：P0发出请求向量Request0=(0，2，0),

(1) $Request_0(0, 2, 0) \leq Need_0(7, 4, 3)$; 请求合法!

(2)os能力判断$Request_0(0, 2, 0) \leq Available(2, 3, 0)$ OS有能力满足进程请求

（3）试分配

	Max(A B C)	Allocate(A B C)	Need(A B C)	work	finish
				2，1，0	
P0	7，5，3	0，3，0	7，2，3	No	
P1	3，2，2	3，0，2	0，2，0	No	
P2	9，0，2	3，0，2	6，0，0	No	
P3	2，2，2	2，1，1	0，1，1	No	
P4	4，3，3	0，0，2	4，3，1	No	

死也找不到安全序列！ 系统状态是不安全的!

(4) OS拒绝进程P0的请求，数据回退到试分配前的状态!

	Max(A B C)	Allocate(A B C)	Need(A B C)	work	finish
				2, 1, 0	
P0	7, 5, 3	0, 3, 0	7, 2, 3		
P1	3, 2 , 2	3, 0, 2	0, 2, 0		
P2	9, 0, 2	3, 0, 2	6, 0, 0		
P3	2, 2, 2	2, 1, 1	0, 1, 1		
P4	4, 3, 3	0, 0, 2	4, 3, 1		

（5）思考： P_0 请求改为 $Request_0(0, 1, 0)$，系统是否能分配资源？

合法性请求 -----OS能力检查------试分配------判断

	Max(A B C)	Allocate(A B C)	Need(A B C)	work	finish
				2, 3, 0	
P0	7, 5, 3	0, 1, 0	7, 4, 3		
P1	3, 2 , 2	3, 0, 2	0, 2, 0		
P2	9, 0, 2	3, 0, 2	6, 0, 0		
P3	2, 2, 2	2, 1, 1	0, 1, 1		
P4	4, 3, 3	0, 0, 2	4, 3, 1		

总结：死锁避免的特点

优点：

缺点：

判断：是否存在资源利用率下降问题？

思路三：检测并解除

既然死锁是小概率事件，无论死锁预防还是死锁避免都会导致资源利用率下降。

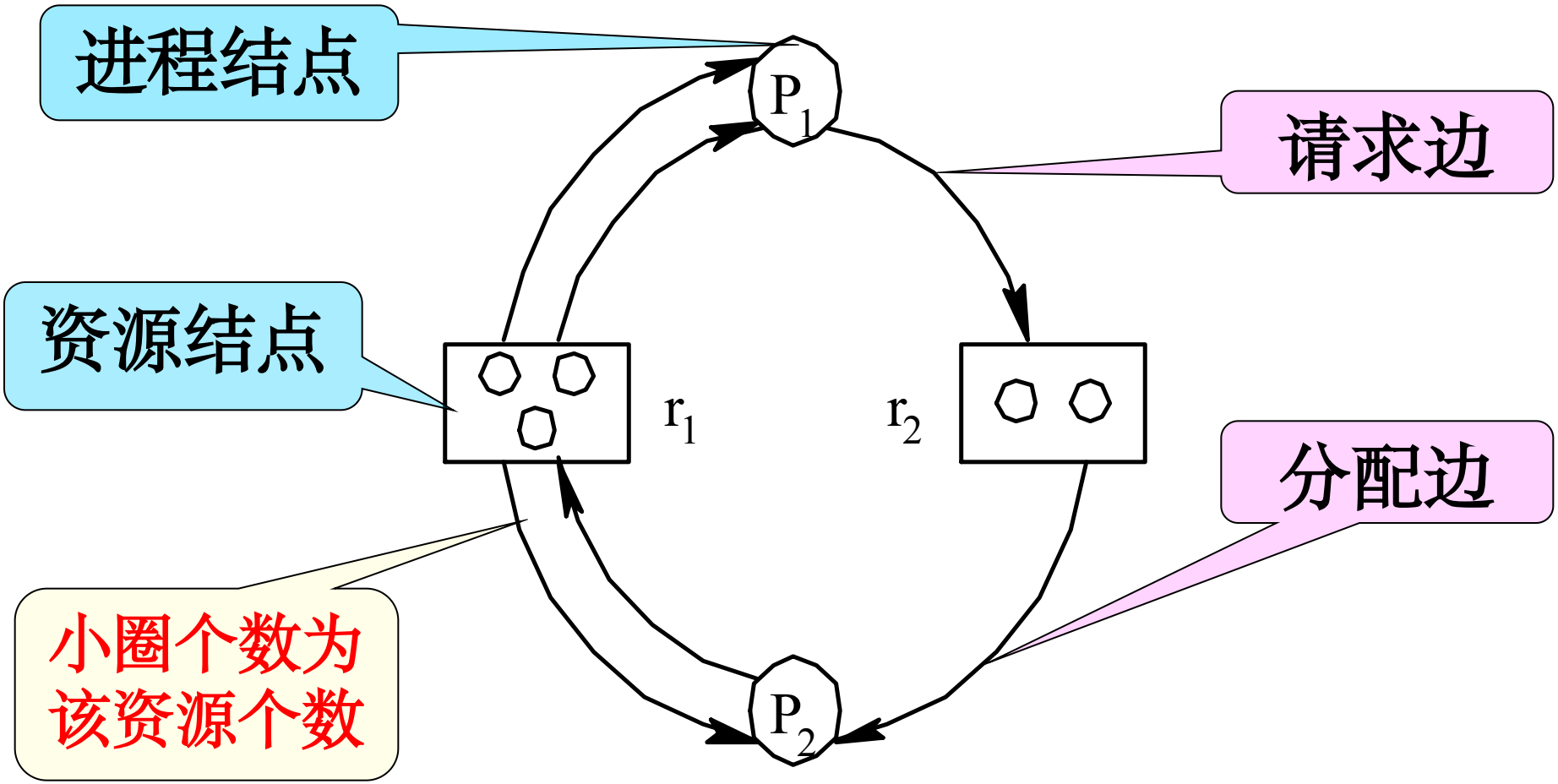
干脆破罐破摔，对进程的资源申请和操作系统的资源分配不做任何限制。允许死锁发生。

对系统进行周期性检测，一旦检测到系统中发生了死锁，外力解除。

死锁定理（死锁充分条件）

- 当系统的资源分配图是**不可完全简化**时，系统中发生了死锁。

可完全简化：通过简化方法，最终能够将资源分配图进行简化，最终**所有的节点**都变成**孤立的节点**（即所有的边全部消去）


进程结点
资源结点
小圈个数为
该资源个数
请求边
分配边
P1
P2
r1
r2

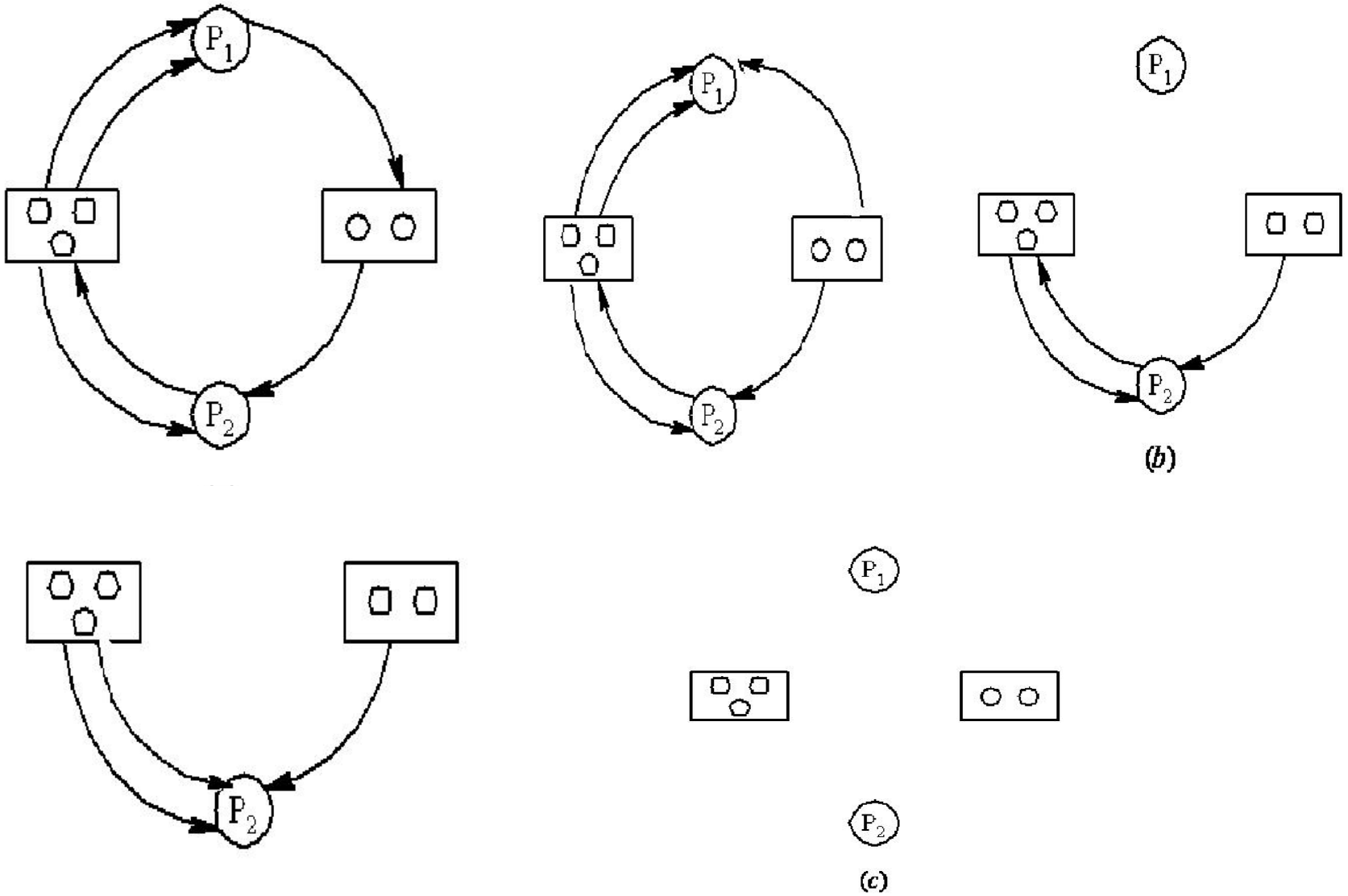
简化方法

重复如下两个过程

1、从资源分配图中任选一个①非孤立②不阻塞的进程，如果其请求边可以得到满足，将请求边改为分配边。

2、如果一个进程所有的边都是分配边，则消去其所有分配边。

简化例子



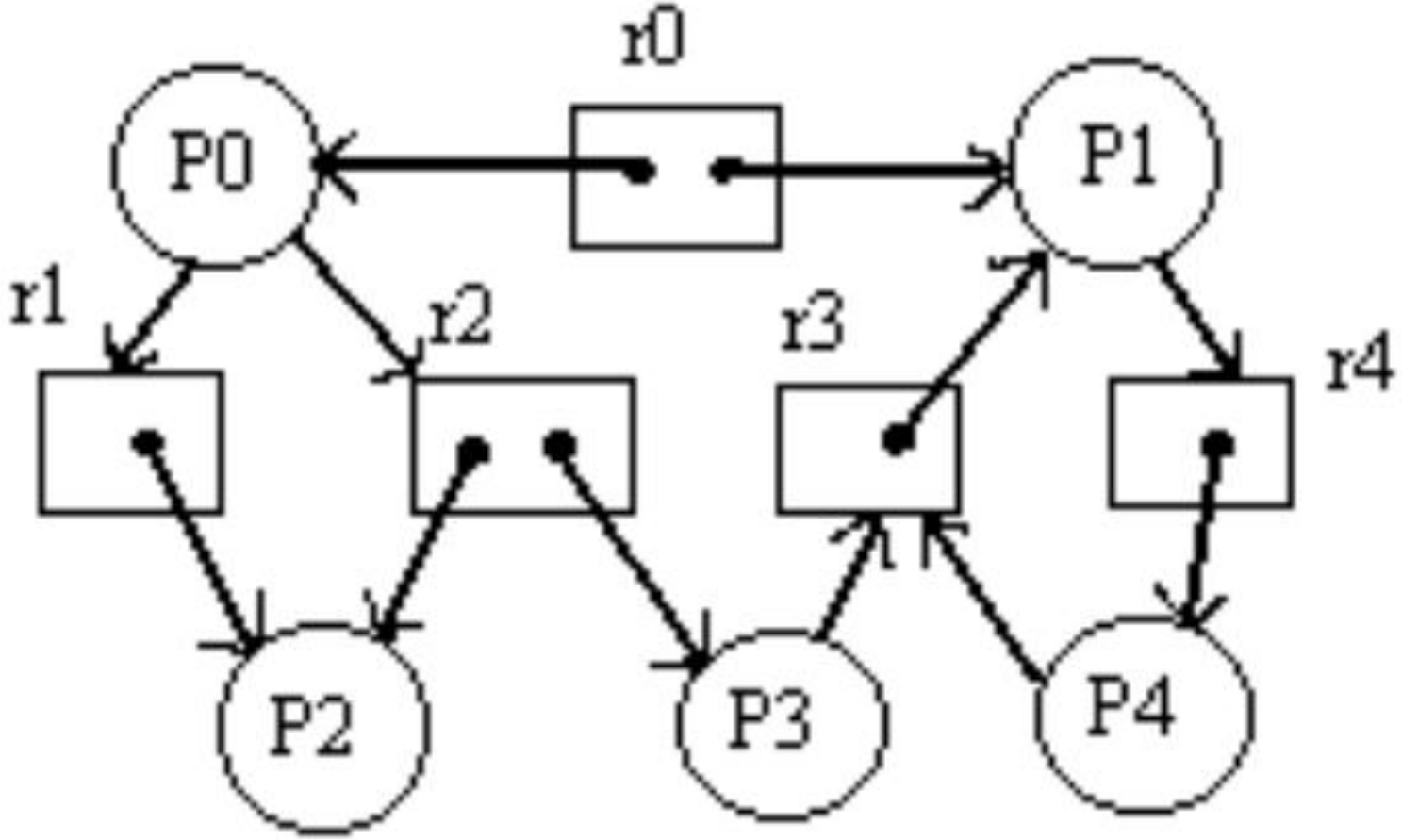
思考

对系统进行周期性检测，一旦检测到系统中发生了死锁，外力解除。

1、周期的选择对自锁的检测和解除有影响吗?

2、可以采用哪些外力方法解除?

3、检测并解除的方法有何优缺点?


r0
P0
P1
r1
r2
r3
r4
P2
P3
P4

谢谢观赏！

Thanks！