



3.5 死锁概述

多个进程竞争资源时进入僵持状态，
若无外力作用，均无法向前推进



3.5.1 资源问题

- 可重用性资源
 - 重复,互斥使用, 如设备
- 消耗性资源
 - 动态产生,如消息
- 可抢占性资源
 - 如处理机, 内存
- 不可抢占性资源
 - 如刻录机

3.5.2 计算机系统中的死锁

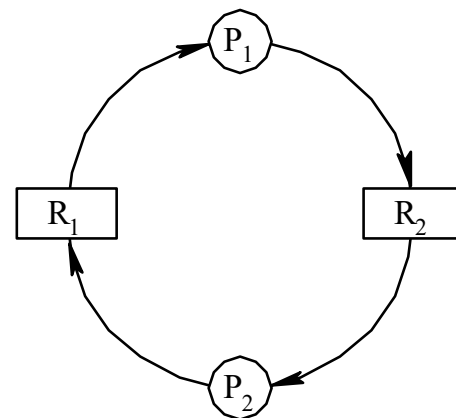
■ 1 竞争资源引起进程死锁

■ (1) 竞争非剥夺资源

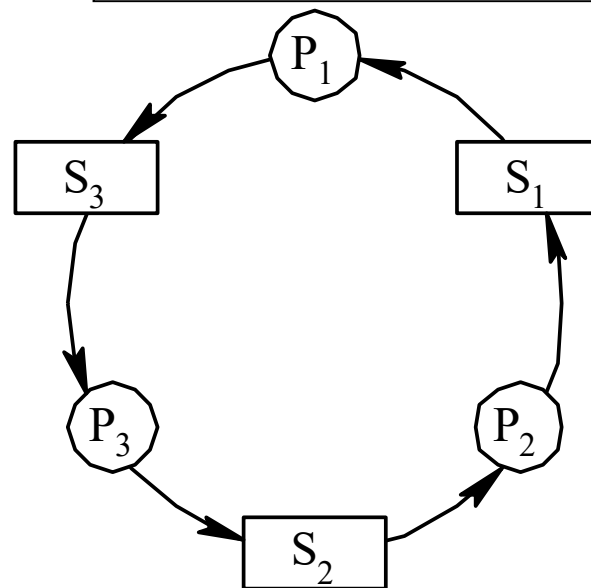
- 打印机，磁带机
- 数量不足时

■ (2) 竞争临时资源

- **s1,s2,s3** 为消息
- **P1: ..Release(s1);Request(s3);**
- **P2: ..Release(s2);Request(s1);**
- **P3: ..Release(s3);Request(s2);**
- **safe**
- **P1: .. Request(s3);Release(s1);**
- **P2: .. Request(s1);Release(s2);**
- **P3: .. Request(s2);Release(s3);**
- **deadlock**



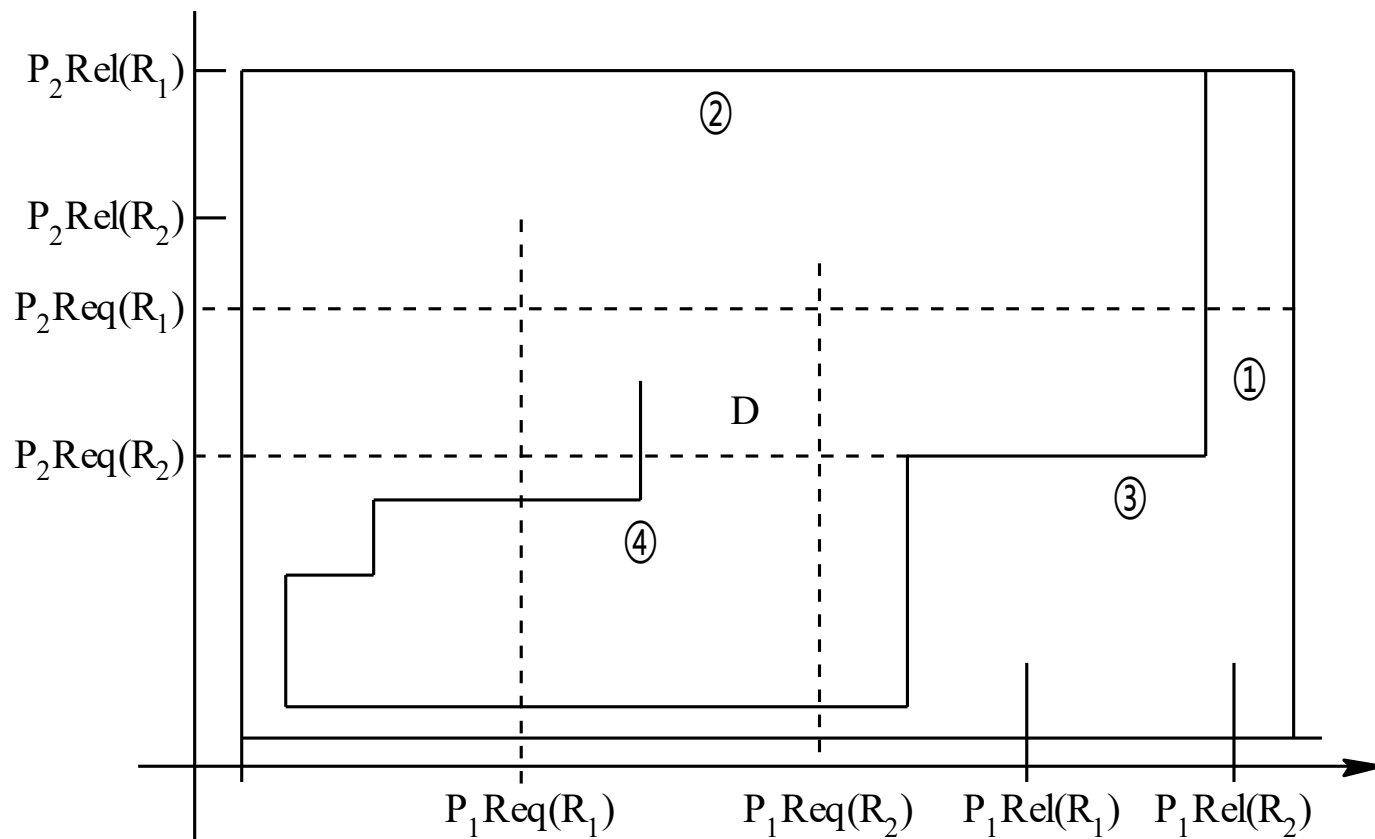
共享I/O设备发生死锁



进程间通信发生死锁

3.5.1 产生死锁的原因

- 2 进程推进顺序不当引起死锁
 - 曲线1, 2, 3均安全
 - 曲线4进入区域D, 不安全



死锁的基本概念

■ 后果

- 参与死锁的所有进程都在等待资源
- 如果死锁发生，会浪费大量系统资源，甚至导致系统崩溃

■ 产生原因

- 并发进程的资源竞争；
 - 申请--分配--使用--释放
 - 系统提供的资源数量少于并发进程所要求的资源数量；资源分配不当；
 - 各进程推进速度不当。
- 如果一组进程中的每一个进程都在等待仅由该组进程中其它进程才能引发的事件,那么该组进程是死锁的

3.5.2 产生死锁的必要条件

■ 1 互斥条件

- 资源不能同时被两个或以上进程共享

■ 2 不剥夺条件

- 进程已经获得的资源在未使用完毕之前，不可被其它进程强行剥夺，而只能自己释放。

■ 3 部分分配（请求和保持）

- 每次申请所需要的一部分资源，在等待新资源的同时，继续占有已经分配到的资源

■ 4 循环等待

- 存在进程循环链，链中每一进程已获得的资源同时被下一个进程所请求。

3.5.4 处理死锁的基本方法

- 对资源的请求和分配进行控制
- 预防
 - 设置限制条件，破坏产生死锁的必要条件
 - 容易实现，降低资源利用率和系统吞吐量
- 避免（动态预防）
 - 不事先限制
 - 资源分配时，系统预测资源使用情况，防止系统进入不安全状态，避免死锁发生
 - 实现难度大
- 检测和恢复
 - 允许死锁发生
 - 检测到死锁发生的位置和原因，通过外力破坏死锁发生的必要条件，使得并发进程从死锁状态恢复过来。



3.6 预防死锁

3.6 死锁预防

- 破坏“请求和保持”条件（打破部分分配条件）
 - 静态分配资源：进程运行前预先分配其运行所全部资源。
 - 优点：
 - 简单，容易实现，安全
 - 缺点
 - 进程资源需求可能无法事先确定；资源很难满足要求
 - 进程生命周期中对资源一直占用，资源浪费
 - 降低了并发性
 - 改进：细分进程任务为子任务，为子任务请求一次性分配资源
- 破坏“不可抢占”条件
 - 进程提出新的资源请求如不能满足则放弃已有资源
 - 降低系统效率，增长周转时间
 - 对某些类型的资源不适用
- 破坏“环路等待”条件
 - 资源分类按顺序排列，各进程有序申请资源
 - 缺点
 - 限制了进程对资源的请求，可能会降低系统效率；
 - 排序需系统开销。

3.7 避免死锁

■ 避免死锁

- 允许进程动态申请资源，分配之前考察分配的安全性

■ 1 安全状态

- 系统能按某种进程顺序(P_1, P_2, \dots, P_n)(称 $\langle P_1, P_2, \dots, P_n \rangle$ 序列为安全序列)，来为每个进程 P_i 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态
- 不安全状态的系统可能会进入死锁

■ 2 安全状态实例

- 假定系统中三个进程P1、P2和P3；
- 共有12台磁带机；进程P1总共要求10台磁带机，P2和P3分别要求4台和9台。
- 假设在70时刻，进程P1、P2和P3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配。
- 安全序列
 - 2→ p2 →4 , 则5台可用
 - 5→ p1 →10 ,
 - →p3

进 程	最 大 需 求	已 分 配	可 用
P₁	10	5	3
P₂	4	2	
P₃	9	2	

■ 3 由安全状态向不安全状态的转换

■ 无法找到一个安全序列

■ 例：P3请求1台，为其分配，

■ 此时，在P2完成后只能释放出4台，既不能满足P1尚需5台的要求，也不能满足P3尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。

■ 故：应该不满足P3的请求。

进 程	最 大 需 求	已 分 配	可 用
P ₁	10	5	2
P ₂	4	2	
P ₃	9	3	

4 利用银行家算法避免死锁

■ 银行家算法

- 假定一个银行家拥有资金，数量为 Σ ，被 N 个客户共享。银行家对客户提出下列约束条件：
 - 每个客户必须预先说明自己所要求的最大资金量；
 - 每个客户每次提出部分资金量申请和获得分配；
 - 如果银行满足了客户对资金的最大需求量，那么，客户在资金运作后，应在有限时间内全部归还银行。
- 银行家将保证做到：
 - 若一个客户所要求的最大资金量不超过 Σ ，则银行一定接纳该客户，并可处理他的资金需求；
 - 银行在收到一个客户的资金申请时，可能因资金不足而让客户等待，但保证在有限时间内让客户获得资金。

4 利用银行家算法避免死锁

■ 数据结构

- n 个进程， m 类个资源
- (1) 可利用资源向量**Available**。
 - 含有 m 个元素的数组
 - 每一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**Available** [j] = K ，则表示系统中现有第 j 类资源 K 个。
- (2) 最大需求矩阵**Max**。
 - $n \times m$ 矩阵
 - 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**Max** [i, j] = K ，则表示进程 i 需要第 j 类资源的最大数目为 K 。
- (3) 分配矩阵**Allocation**。
 - $n \times m$ 矩阵
 - 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
 - **Allocation** [i, j] = K ，则表示进程 i 当前已分得第 j 类资源的数目为 K 。
- (4) 需求矩阵**Need**
 - $n \times m$ 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 还需要第 j 类资源 K 个，方能完成其任务。
- **Need** [i, j] = **Max** [i, j] - **Allocation** [i, j]

4 利用银行家算法避免死锁

■ 银行家算法

- 设 **Request_i** 是进程 **P_i** 的请求向量，如果 **Request_i [j] = K**，表示进程 **P_i** 需要 **K** 个第 **j** 类型的资源。当 **P_i** 发出资源请求后，系统按下述步骤进行检查：
 - **(1)** 如果对所有 **j**, **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。
 - **(2)** 如果对所有 **j**, **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。

4 利用银行家算法避免死锁

■ 银行家算法

- (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。

4 利用银行家算法避免死锁

■ 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 如果对所有 j , 如果 **Need [i,j] ≤ Work [j]** 执行步骤(3)
- 找不到则执行步骤(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:

- 对所有 j **Work [j] :=Work [j] +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 利用银行家算法避免死锁

■ 银行家算法实例 1

- 资源的数量分别为**10**、**5**、**7**

	Max			Allocation			Need (=Max- Allocation)			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0	1	0	7	4	3	3	3	2
p1	3	2	2	2	0	0	1	2	2			
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			

4 利用银行家算法避免死锁

■ 银行家算法实例 1

■ 为p1分配: **Work > Need**

	Work			Need			Allocation			Work+ Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
p1	3	3	2	1	2	2	2	0	0	5	3	2	ture
p3	5	3	2	0	1	1	2	1	1	7	4	3	ture
p4	7	4	3	4	3	1	0	0	2	7	4	5	ture
p2	7	4	5	6	0	0	3	0	2	10	4	7	ture
p0	10	4	7	7	4	3	0	1	0	10	5	7	ture

4 利用银行家算法避免死锁

银行家算法实例 2

- P1申请: Request1(1,0,2)
 - Request1(1,0,2) < Need1(1,2,2);
 - Request1(1,0,2) < Available(3,3,2);

	Work			Need			Allocaiton			Work+ Allocaiton			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
p1	3	3	2	1	2	2	2	0	0	5	3	2	ture
	/	/	/	/	/	/	/	/	/				
	2	3	0	0	2	0	3	0	2				
p3	5	3	2	0	1	1	2	1	1	7	4	3	ture
p4	7	4	3	4	3	1	0	0	2	7	4	5	ture
p0	7	4	5	7	4	3	0	1	0	7	5	5	ture
p2	7	5	5	6	0	0	3	0	2	10	5	7	ture

4 利用银行家算法避免死锁

■ 银行家算法实例 3

■ P4申请: Request4(3,3,0)

- Request4(3,3,0) < Need4(4,3,1);
- 但, Request4(3,3,0) > Available(2,3,0);
- 故P4申请不能得到满足。

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0	1	0	7	4	3	2	3	0
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			

4 利用银行家算法避免死锁

■ 银行家算法实例 4

- **P0申请: Request₀(0,2,0)**
 - **Request₀(0,2,0) < Need₀(7,4,3);**
 - **Request₀(0,2,0) < Available(2,3,0);**
 - 分配后剩余无法满足任何进程

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0/ 0	1/ 3	0/ 0	7/ 7	4/ 2	3/ 3	2 / 2	3 / 1	0 / 0
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			

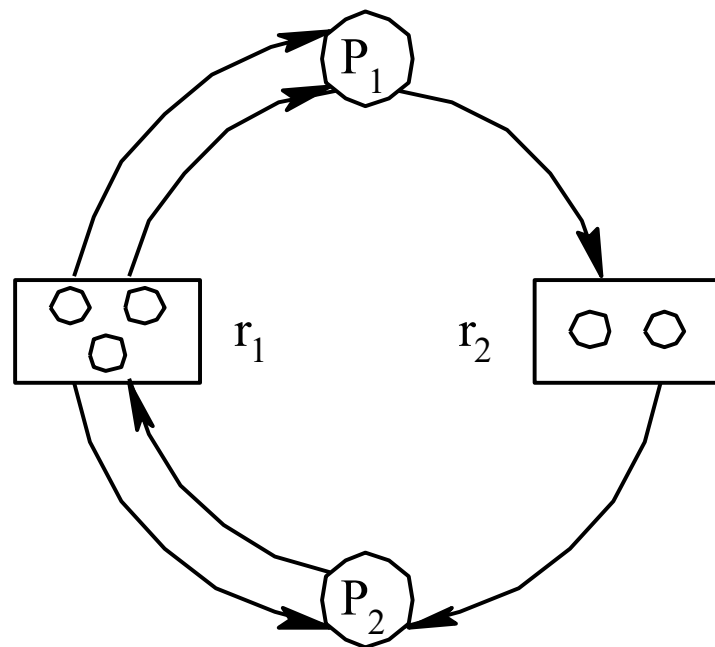


3.8 死锁的检测和解除

3.8.1 死锁的检测

■ 1. 资源分配图(Resource Allocation Graph)

- $G = (N, E)$ 结点 N , 边 E ;
- 结点 N 分为: 进程结点 P 、资源结点 R ;
- 凡属于 E 中的一个边 $e \in E$, 都连接着 P 中的一个结点和 R 中的一个结点
- $e = \{p_i, r_j\}$ 是资源请求边, 由进程 p_i 指向资源 r_j , 它表示进程 p_i 请求一个单位的 r_j 资源。
- $e = \{r_j, p_i\}$ 是资源分配边, 由资源 r_j 指向进程 p_i , 它表示把一个单位的资源 r_j 分配给进程 p_i 。



3.8.1 死锁的检测

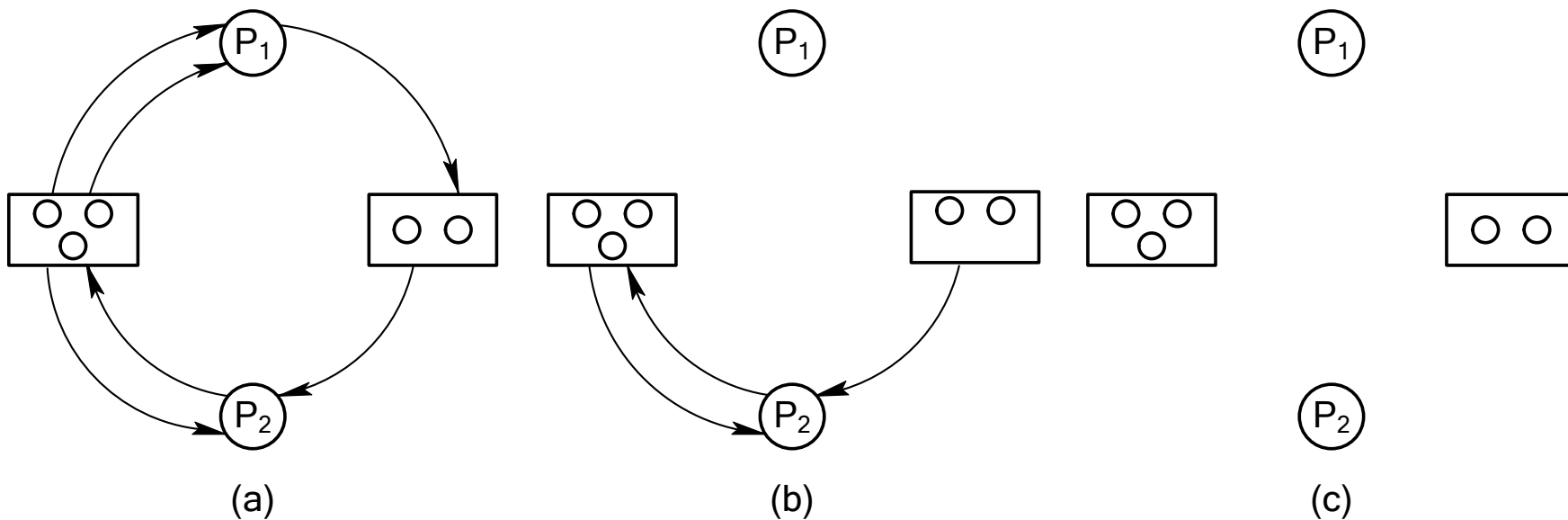
■ 2 死锁定理

- **(1)** 在资源分配图中，找到既不阻塞又非独立的进程结点 \mathbf{pi} 。此进程可顺利结束，释放资源。即可消去 \mathbf{pi} 相连的请求边和分配边，使其称为孤立结点。
- **(2)**依次进行。
- **(3)**若所有进程都能成为孤立结点，则称该图为可完全简化。否则为不可完全简化。
- 死锁状态的充分条件是：当且仅当该状态的资源分配图是不可完全简化的。

3.8.1 死锁的检测

■ 2 死锁定理

■ 资源分配图的简化



3.8.1 死锁的检测

■ 3. 死锁检测中的数据结构

- (1) 可利用资源向量**Available**，它表示了**m**类资源中每一类资源的可用数目。
- (2) 把不占用资源的进程**P_i**(向量**Allocation:=0**)记入**L**表中，即 **$L = \{P_i\} \cup L$** 。
- (3) 从不再**L**中进程集合中找到一个进程**P_i**, **Request_i ≤ Work**的进程，做如下处理：
 - ① 将其资源分配图简化，释放出资源，增加工作向量 **Work:=Work+Allocation_i**。
 - ② 将对应进程**P_i**记入**L**表中。重复 (3)
- (4) 若不能把所有进程都记入**L**表中，便表明系统状态**S**的资源分配图是不可完全简化的。因此，该系统状态将发生死锁。

3.8.1 死锁的检测

■ 3. 死锁检测中的数据结构

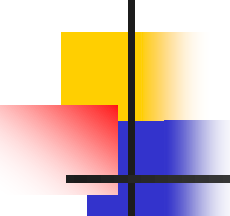
(参照银行家算法)

```
Work = Available;
L = { Pi | Allocationi = 0 ∧ Requesti = 0 }
finish = false;
while (finish == false)
    finish = true;
    for (all Pi ! ∈ L) {
        if (Requesti ≤ Work) {
            Work = Work + Allocationi;
            L = { Pi } ∪ L;
            finish = false;
        }
    }
}
deadlock = ! (L == { p1, p2, ..., pn });
```



3.8.2 死锁的解除

- **(1) 剥夺资源**
- **(2) 撤消进程**
 - 尽可能使撤销的进程数目少或代价小
- 死锁不是经常出现的，因而检测和恢复可行。



■ 死锁的一些问题

■ 通信死锁

■ 超时机制

■ 活锁

■ 没有阻塞，不能执行

■ 死锁与饥饿

```
void process_A(void) {  
    acquire_lock(&resource_1);  
    while (try_lock(&resource_2) == FAIL) {  
        release_lock(&resource_1);  
        wait_fixed_time();  
        acquire_lock(&resource_1);  
    }  
    use_both_resources( );  
    release_lock(&resource_2);  
    release_lock(&resource_1);  
}
```

```
void process_A(void) {  
    acquire_lock(&resource_2);  
    while (try_lock(&resource_1) == FAIL) {  
        release_lock(&resource_2);  
        wait_fixed_time();  
        acquire_lock(&resource_2);  
    }  
    use_both_resources( );  
    release_lock(&resource_1);  
    release_lock(&resource_2);  
}
```