上节课重点：

1. 实时调度（硬实时任务、软实时任务、抢占式算法、非抢占式法、最早截止时间、最低松弛度算法）

2. 死锁：定义、产生原因、必要条件

3. 死锁的处理方法：预防死锁、避免死锁、检测死锁、解除死锁

4. 避免死锁--银行家算法

3.7.2 避免死锁--银行家算法

- 银行家算法（Dijkstra, 1965）问题
  - 在银行中，客户申请贷款的数量是有限的。银行家应尽量满足所有客户的贷款需求。
  - 银行家就好比操作系统，资金就是资源，客户就相当于要申请资源的进程。

- 核心思想：在进程提出资源申请时，先预判此次分配是否会导致系统进入不安全状态。如果会进入不安全状态，就暂时不答应这次请求，让该进程先阻塞等待。

3.7.2 避免死锁--银行家算法

1、银行家算法中的数据结构

设系统中有m类资源，n个进程

（1）可利用资源向量Available。含有m个元素的一维数组，每个元素代表一类可利用的资源数目。

如果Available[j]=k,表示系统中现有Rj类资源k个。

（2）最大需求矩阵Max，是一个n*m的矩阵，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。

如果Max[i,j]=k，则表示进程i，需要Rj类资源的最大数目为k。

3.7.2 避免死锁--银行家算法

（3）分配矩阵Allocation。为n*m的矩阵，它定义了系统中每类资源当前已分配给每个进程的资源数。Allocation[i,j]=K,表示进程i当前已分得Rj类资源的数目为K。

（4）需求矩阵Need。为n*m的矩阵，表示每个进程尚需的各类资源数。Need[i,j]=K, 表示进程i还需要Rj类资源K个，方能完成其任务。

三个矩阵间的关系：Need[i,j]=Max[i,j]-Allocation[i,j]

2、银行家算法的处理步骤

设$Request_i[j]=k$,表示进程Pi需要k个Rj类型的资源

（1）如果$Request_i[j] <= Need[i,j]$,便转向步骤2；否则认为出错，因为它所请求的资源数已超过它所需要的最大值。

（2）如果$Request_i[j] <= Available[j]$,便转向步骤3；

否则，表示尚无足够资源，Pi需等待。

（3）系统试探着把资源分配给进程Pi，并修改下面数据结构中数值

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

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

$Need[i,j]:=Need[i,j]-Request_i[j];$

3.7.2　避免死锁--银行家算法

（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以决定是否完成本次分配。若安全，则正式分配资源给进程Pi；否则，不分配资源，让进程Pi等待。

3、安全性算法

（1）设置两个向量：

工作向量**work**：表示系统可提供给进程继续运行所需的各类资源数目，含有m个元素的一维数组，初始时，work:=Available;

Finish：含n个元素的一维数组，表示系统是否有足够的资源分配给n个进程，使之运行完成。开始时先令Finish[i]:=false (i=1..n);当有足够资源分配给进程i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：

Finish[i]=false;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[j]+ Allocation[i,j] ;

Finish[i]:=true;

goto 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。

银行家算法之例

- 5个进程：P_0~P_4；3类资源：A (10个实例)，B (5个实例)，C (7个实例).
- 假设在 T_0时刻,系统状态如下：

	Allocation			Max			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			

银行家算法之例

- 需要导出Need矩阵。 Need矩阵定义为：Need = Max – Allocation。则得到：

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

- 用安全算法可证明系统目前是安全的，因为存在安全序列：< P_1, P_3, P_4, P_2, P_0>.

银行家算法之例

问：

(1)考察系统在T_0时刻的安全性?

(2)P_1请求资源$Request_1(1, 0, 2)$?

(3)P_4请求资源$Request_4(3, 3, 0)$?

(4)P_0请求资源$Request_0(0, 2, 0)$?

银行家算法之例

- 假设这时进程P_1发出资源请求：

  Request1 = (1,0,2)

- 首先检查：

  Request1 ≤ Need1　(1,0,2) ≤ (1,2,2)

- 再检查 Request1 ≤ Available (1,0,2) ≤　(3,3,2)
  ⇒ 即满足申请资源的条件。

- 假设将资源分配给它，则得到如下的新状态：

银行家算法之例

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	2	3	0
P_1	3	0	2	3	2	2	0	2	0			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

- **执行安全算法可以找到安全序列：**

  $<P_1, P_3, P_4, P_0, P_2>$
- **接下来进程P_4请求资源 (3,3,0) 能够满足它吗?**
- **若是进程P_0请求资源 (0,2,0)能满足它吗?**

银行家算法的特点

- 允许互斥、部分分配和不可抢占，可提高资源利用率；
- 要求事先说明最大资源要求，在现实中很困难。

作业

- P128　　31（第四版）

3.8 死锁的检测和解除

- 如果系统中既不采取预防死锁的措施，也不采取避免死锁的措施，系统就很可能发生死锁。在这种情况下，系统应当提供两个算法：
  - 死锁检测算法：用于检测系统状态，以确定系统中是否发生了死锁。
  - 死锁解除算法：当认定系统中已经发生了死锁，利用该算法可将系统从死锁状态中解脱出来。

3.8 死锁的检测和解除

一、死锁的检测

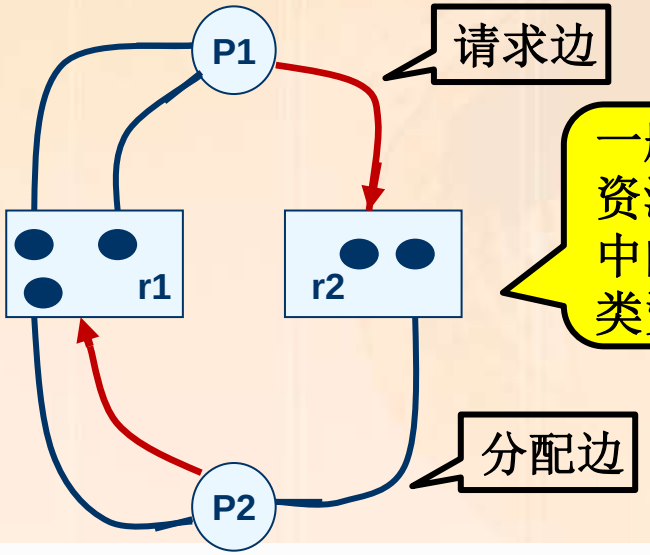
- 为了能对系统是否已经发生了死锁进行检测，必须：
  - 用某种数据结构保存资源的请求和分配信息；
  - 提供一种算法，利用上述信息来检测系统是否已经进入死锁状态。

3.8 死锁的检测和解除

1、资源分配图

数据结构：资源分配图

- 两种节点
  - 进程节点：对应一个进程
  - 资源节点：对应一类资源，一类资源可能有多个
- 两种边
  - 进程节点==》资源结点：表示进程想申请几个资源(每条边代表一个)
  - 资源节点==》进程结点：表示已经为进程分配了几个资源(每条边代表一个)



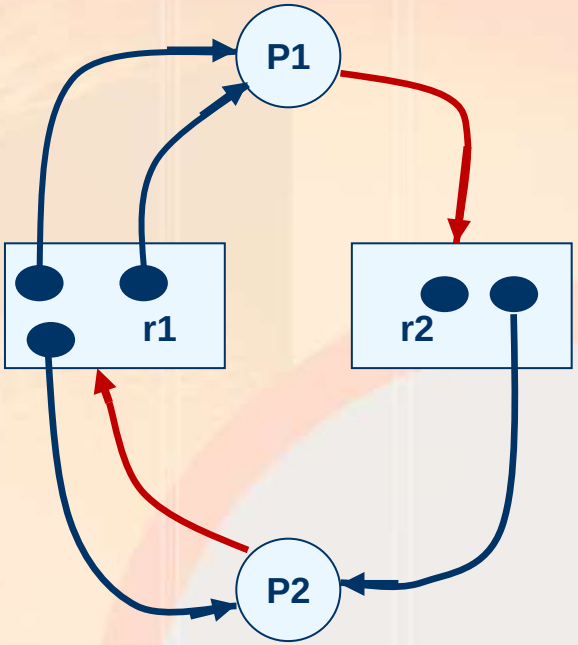
Tips: 学习过数据结构--图的同学可以动手尝试一下，如何定义这个数据结构。

2、死锁的检测

为了能对系统是否已经发生死锁进行检测，必须：

① 用某种数据结构来保存资源的请求和分配信息；

② 提供一种算法，利用上述信息来检测系统是否已经进入死锁状态。



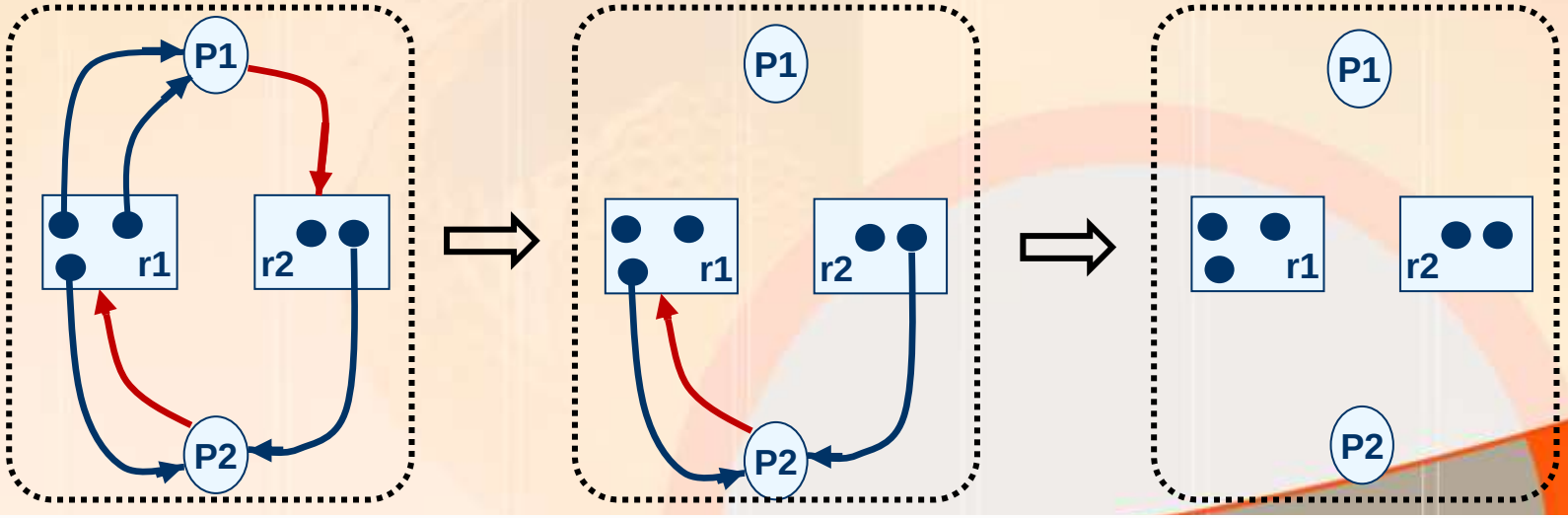
3.8 死锁的检测和解除

我们可以利用资源分配图加以简化的方法，来检测系统处于某状态时是否为死锁状态。简化方法如下：

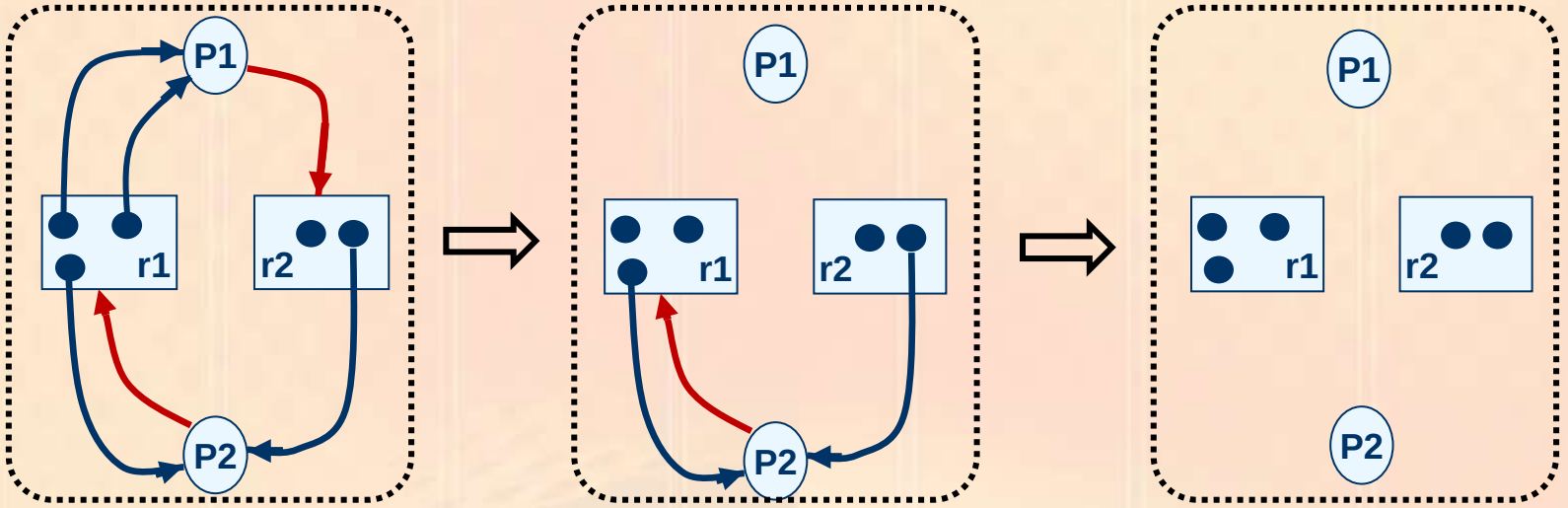
状态1：如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。

状态2：如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。

状态3：相应地，这些被激活地进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程...



3.8 死锁的检测和解除

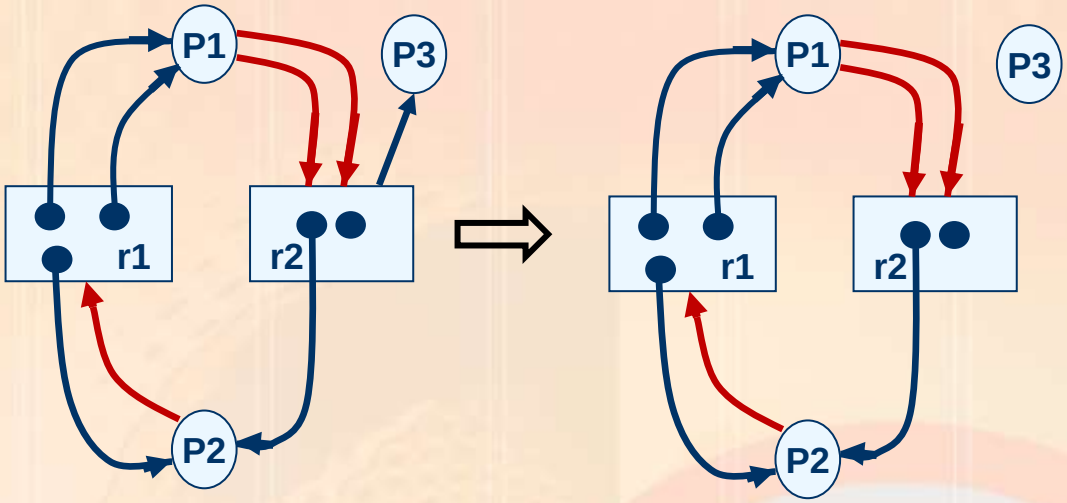


如果按照上述过程分析，最终能消除所有边，就称这个图是可完全简化的。此时一定没有发生死锁(相当于能找到一个安全序列)。

如果最终不能消除所有边，那么此时就是发生了死锁。

3.8 死锁的检测和解除

- 最终证明：S状态为死锁状态的充分条件是当且仅当S状态的资源分配图是不可完全简化的。〈死锁定理〉



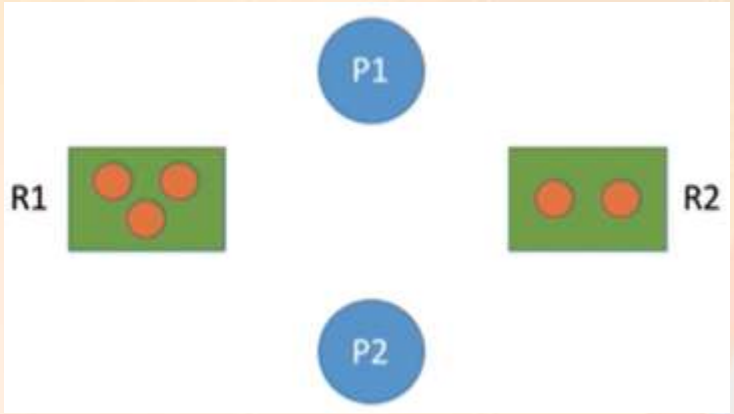
3.8 死锁的检测和解除

检测死锁算法总结：

检测死锁的算法：

1）在资源分配图中，找出既不阻塞又不是孤点的进程 Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。如下图中，R1没有空闲资源，R2有一个空闲资源。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配变，使之称为孤立的结点。在下图中，P1 是满足这一条件的进程结点，于是将P1的所有边消去。

2）进程 Pi 所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在下图中，P2 就满足这样的条件。根据 1）中的方法进行一系列简化后，若能消去途中所有的边，则称该图是可完全简化的。



既然检测到了死锁，那就要想办法解除。

3. 死锁检测算法

- 1）检测算法中的数据结构
  - 设系统中有n个进程，m类资源
  - Available:长度为 m 的向量，表示各种类型资源的可用实例数
  - Allocation:为 n × m 矩阵，表示目前已分配给各个进程的各种资源的数量
  - Request:为 n × m 矩阵，表示目前各个进程请求资源的情况。若Request[i,j] = k, 表示进程 P_i 正在请求 k 个类型为R_j的资源。

2）算法描述

(1) 设 Work 和 Finish **分别是长度为 m 和 n 的向量，各自初始化为：**

(a) Work = Available

(b) 对于i = 1,2, ..., n，如果 $Allocation_i \neq 0$，那么 Finish[i] = false；否则 Finish[i] = true.

(2) **查找这样的下标 i ，使其满足：**

(a) Finish[i] == false

(b) $Request_i \leq Work$

如果不存在这样的 i ，转去执行第4步

(3) Work = Work + $Allocation_i$
Finish[i] = true
转去执行第2步 。

(4) 如果**对某个i** （ 1 ≤ i ≤ n ），若Finish[i] == false，**那么系统死锁。 而且，如果** Finish[i] == false，**那么进程 P_i 正处于死锁状态。**

该算法需要$O(m \times n^2)$ 操作来检查系统是否处于死锁状态

- 设系统有5个进程P_0 - P_4；3类资源：A (7 个), B (2 个), C (6 个).
- 假设在 T_0 时刻：

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- 序列： <P_0, P_2, P_3, P_1, P_4> 表明对所有的i，*Finish*[*i*] = true。即表明系统现在不处于死锁状态。

- **假设现在进程P_2 请求一个C类资源，则系统状态变成：**

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- **现在系统处于什么状态?**

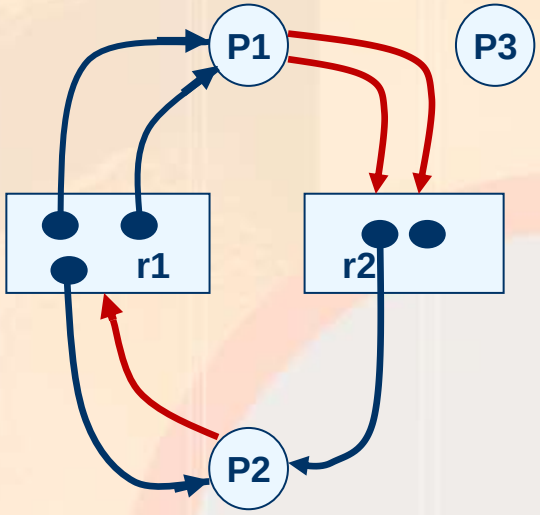
- 虽然现在能够收回进程 P_0 所拥有的资源，但是即便这样的话，目前可用资源还是不足以满足其余任何一个进程对资源的需求。
- 死锁将会发生，进程P_1, P_2, P_3 和P_4都会死锁。

- 4.死锁检测算法的应用
  - 何时、以什么样的频率调用检测算法，取决于以下两个因素：
    - 可能发生死锁的频率是多少?
    - 将影响多少进程（重新执行全部或部分代码）?
  - 一旦资源请求不能立刻响应，调用检测算法（开销大）
  - 固定间隔调用检测算法，例如每小时1次或CPU利用率<40%。

二、死锁的解除

当发现进程死锁时，便应立即把它们从死锁状态中解脱出来。常采用的方法是：

1、资源剥夺法：挂起(暂时放到外存上)某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。

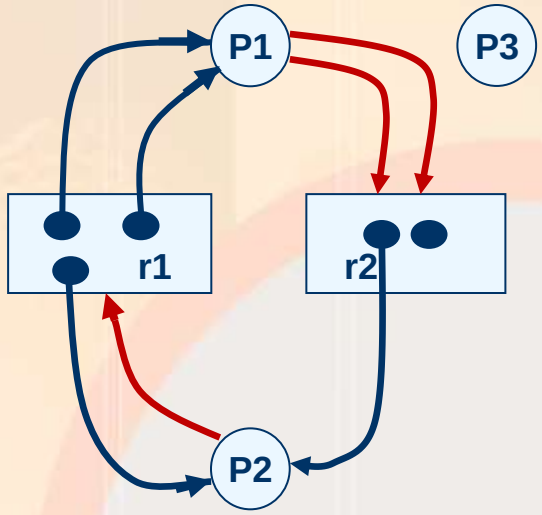


剥夺资源法

- **选择一个牺牲者**
  - **代价最小**
- **后退（也称回滚-Rollback）**
  - **退回到安全状态，在此重新启动进程**
  - 完全回滚：中止进程后重新开始
- **饿死**
  - **如果仅仅是基于代价来选择进程的话，某些进程可能会饿死。因此“代价”还应增加一个因素：做牺牲品的次数。**

2、撤销进程法：强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。

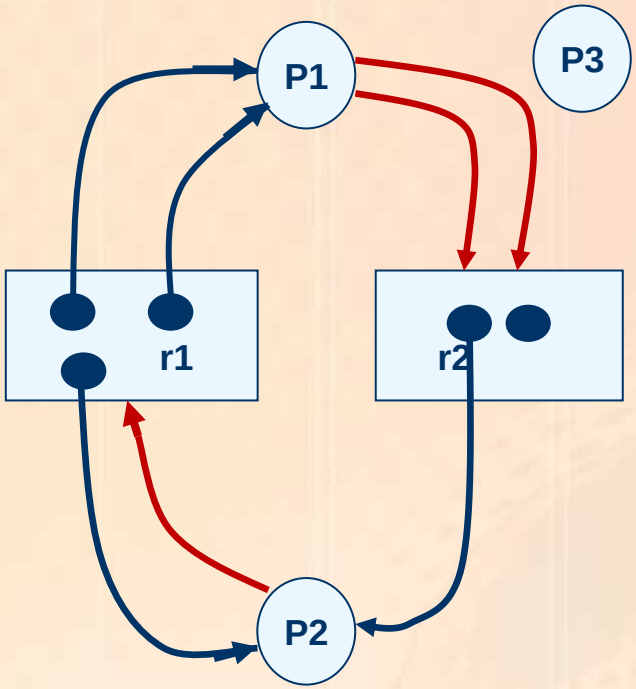
3、进程回退法：让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。



撤销进程法

- 1. 终止所有死锁的进程。代价大。易实现。
- 2. 一次只终止一个进程，直到消除环路为止。
  - 哪一个进程应该被终止?
    - 进程的优先权.
    - 进程已经执行了多久，离完成计算任务还有多长时间。
    - 进程使用了多少资源。
    - 进程还需要多少资源才能完成.
    - 有多少进程需要终止.
    - 是交互进程还是批处理进程

3.8 死锁的检测和解除



如何决定“对谁动手”

1. 进程优先级
2. 已执行多长时间
3. 还要多久完成
4. 进程已经使用了多少资源
5. 进程是交互式的还是批处理式的

例：在著名的“哲学家就餐问题”中，当5位哲学家同时饿了，同时取得左手的筷子，再同时申请右手的筷子，此时发生死锁。请简述如何用死锁预防、死锁避免、死锁检测与恢复的方法解决该死锁问题。

答：

- 死锁预防：
  - 破坏占有等待条件：每位哲学家拿筷子时必须左右筷子同时申请，即采用资源一次性分配法，若缺一则不可获得资源分配。
  - 破坏非剥夺条件：第1位哲学家拿筷子时，若左右邻居中有等待资源者，则可强行抢夺其已经占有的筷子。
  - 破坏循环等待：对5根筷子顺序编号，即1，2，3，4，5，每位哲学家只能按由小到大的顺序申请(资源顺序分配法)。
- 死锁避免：每次分配筷子时，执行银行家算法，确保系统不会进入不安全状态。
- 死锁检测与恢复：对筷子的申请和分配不加限制，但定期（例如每隔5秒）检测是否已发生死锁，是则剥夺其中一位哲学家的筷子分配给其他人.