



第五章 死锁 Deadlock



5.1 死锁的引出

- 多道程序的并发执行可以改善系统的资源，但也可能导致死锁的发生。

日常生活中的死锁例

- 假设一条河上有一座独木桥，若桥两端的人相向而行.....



进程推进顺序不当产生死锁

例 1 设系统有打印机、读卡机各一台，被进程 P 和 Q 共享。两个进程并发执行，按下列次序请求和释放资源：

进程 P

请求读卡机

请求打印机

释放读卡机

释放打印机

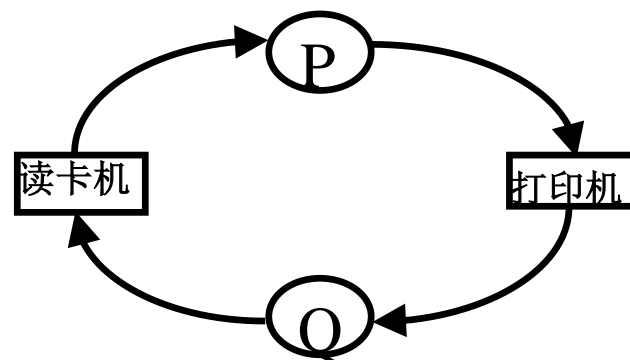
进程 Q

请求打印机

请求读卡机

释放读卡机

释放打印机





PV操作使用不当产生死锁

例2:

进程Q1

.....

P(s1);

P(s2);

使用r1和r2;

V(s1);

V(s2);

进程Q2

.....

P(s2);

P(s1);

使用r1和r2

V(s2);

V(s1);



资源分配不当引起死锁

例3：若系统中有 m 个资源被 n 个进程共享，每个进程都要求 K 个资源，而 $m < n \cdot K$ 时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁。



对临时性资源使用不加限制引起死锁

- 进程通信使用的信箱方式是一种临时性资源，如果对信件的发送和接收不加限制，可能引起死锁。
 - 进程P1等待进程P3的信件S3来到后再向进程P2发送信件S1；
 - P2又要等待P1的信件S1来到后再向P3发送信件S2；
 - 而P3也要等待P2的信件S2来到后才能发出信件S3。
 - 这种情况下形成了循环等待，产生死锁。



死锁的概念

- 操作系统中的死锁
 - 如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生了死锁。



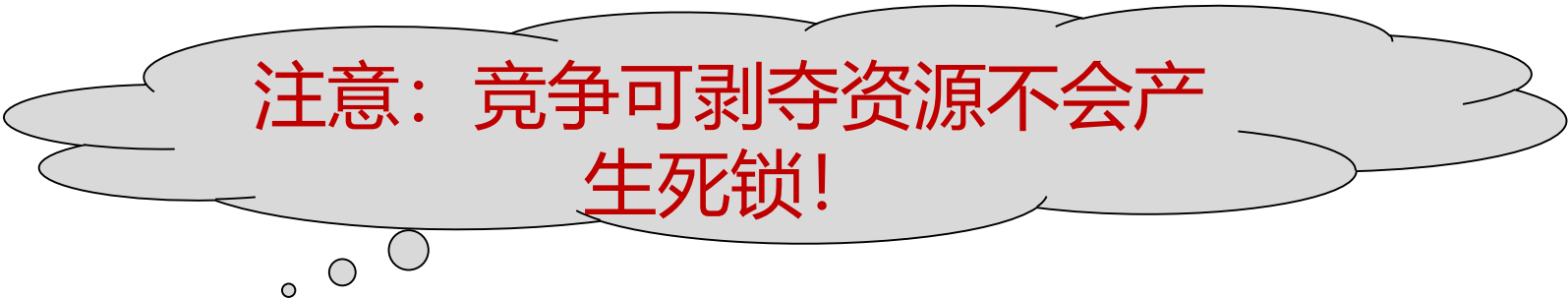
5.2 死锁产生的原因和特征

- 死锁产生的原因是与资源的类型、资源的数量和相应的使用相关
- 资源分类
 - 可剥夺资源与非可剥夺资源
 - 永久性资源和消耗性资源



5.2.1 资源的分类

- 可剥夺资源
 - 指某进程获得这类资源后，该资源可以被其他进程或系统剥夺。如CPU，主存储器。
- 非剥夺资源，又称不可剥夺资源
 - 指系统将这类资源分配给进程后，再不能强行收回，只能在进程使用完后主动释放。如打印机、读卡机。



注意：竞争可剥夺资源不会产生死锁！



5.2.1 资源的分类

- 永久性资源
 - 可顺序重复使用的资源，如打印机。
- 消耗性资源
 - 由一个进程产生，被另一个进程使用短暂时间后便无用的资源，又称为临时性资源，如消息。



竞争永久性资源和临时性资源
都可能产生死锁。

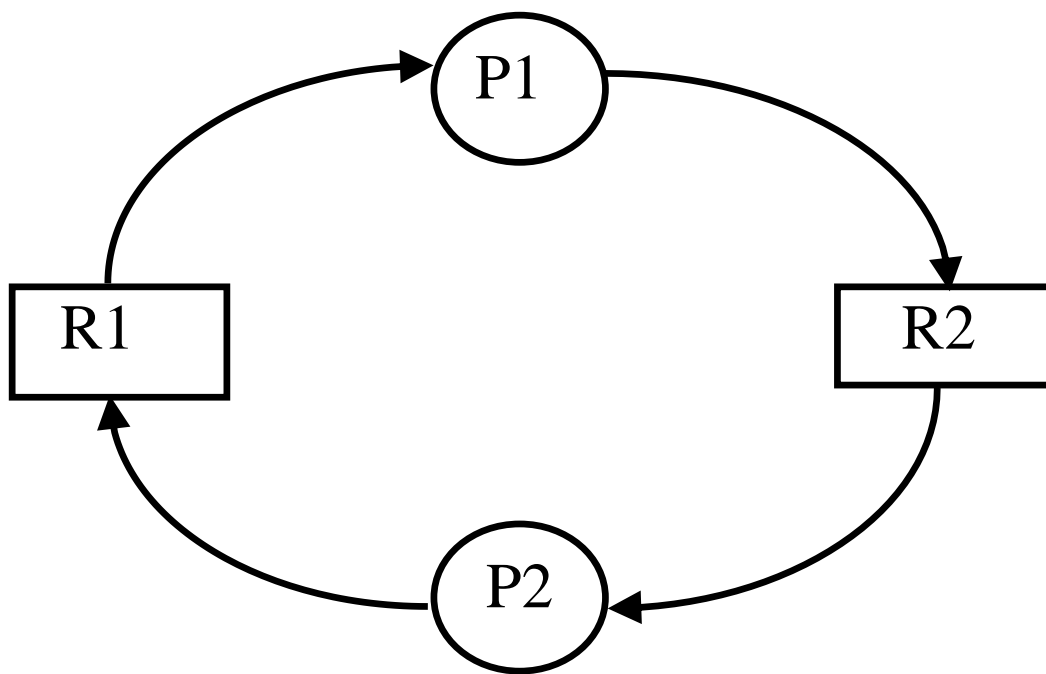


5.2.2 死锁产生的原因

- 死锁产生的原因是：
 - 竞争资源
 - 多个进程竞争资源，而资源又不能同时满足其需求。
 - 进程推进顺序不当
 - 进程申请资源和释放资源的顺序不当。

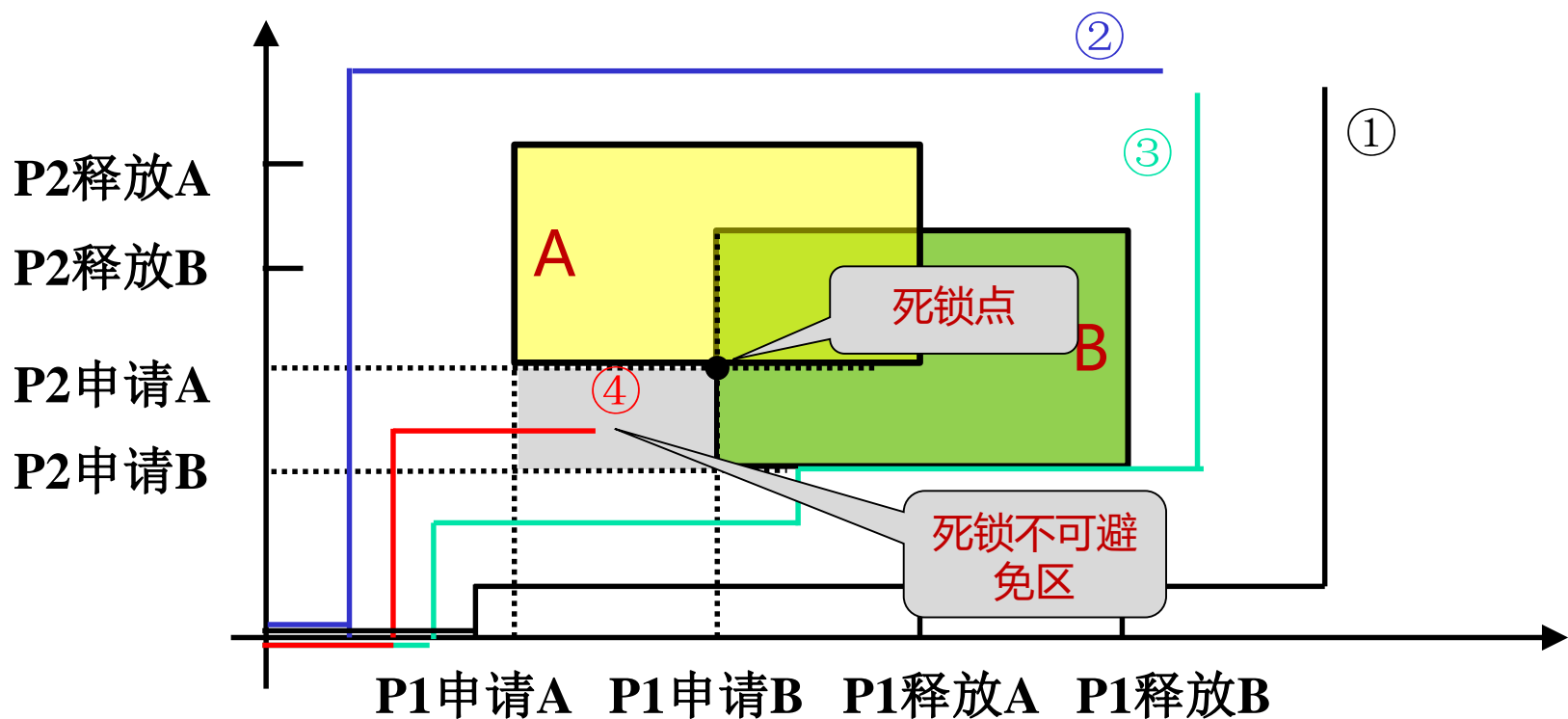
竞争非剥夺资源引起的死锁

- 竞争非剥夺资源例。如，打印机R1和读卡机R2供进程P1和P2共享。



进程推进顺序不当引起的死锁

- 当进程P1、P2共享资源A、B时，若推进顺序合法则不会产生死锁，否则会产生死锁。
- 合法的推进路线：①②③ 不合法的推进线路：④





死锁产生的4个必要条件(Coffman 1971)



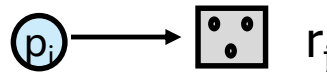
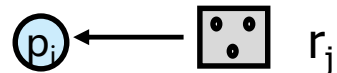
- ① 互斥条件：在一段时间内某资源仅为一个进程所占有。
- ② 请求和保持条件（占有并等待）：又称部分分配条件。当进程因请求资源被阻塞时，已分配资源保持不放。
- ③ 不剥夺条件（非抢占）：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走。
- ④ 循环等待条件：死锁发生时，存在一个进程资源的循环。



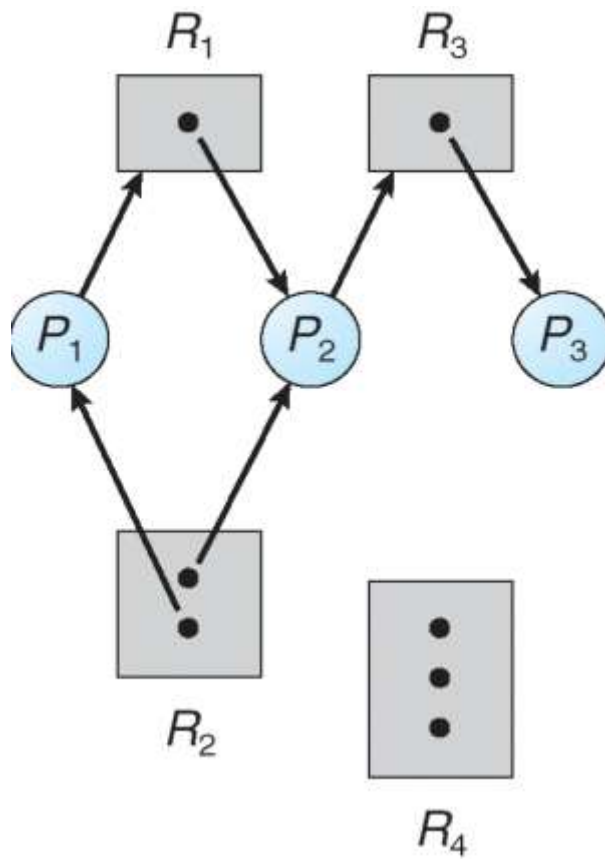
注意

- 死锁是因资源竞争造成的僵局
- 通常死锁至少涉及两个进程
- 死锁与部分进程及资源相关

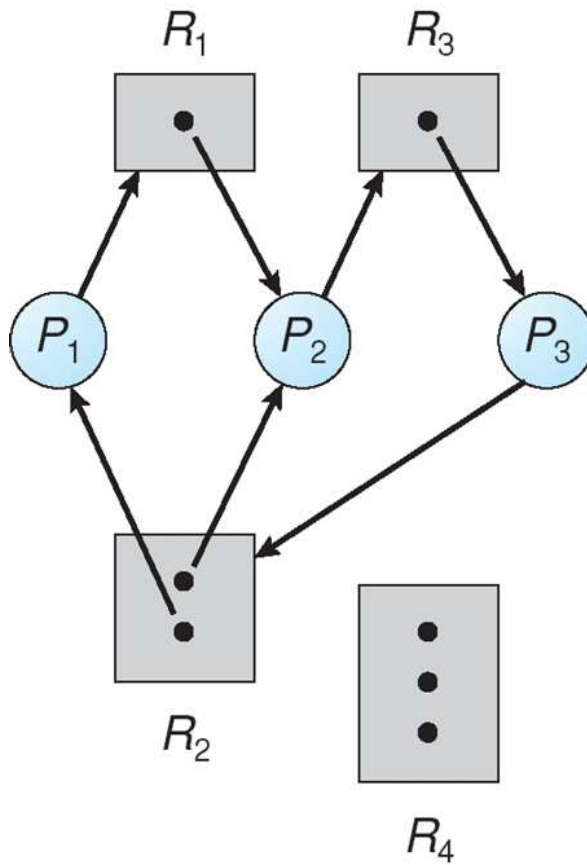
5.2.3 资源分配图

- 系统死锁可利用资源分配图描述。
 - 资源分配图又称“进程——资源”图，由一组结点N和一组边E所构成：
 - N被分成两个互斥的子集：进程结点子集 $P = \{p_1, p_2, \dots, p_n\}$ ，资源结点子集 $R = \{r_1, r_2, \dots, r_m\}$ 。
 - 用圆圈代表一个进程 
 - 用方框代表一类资源，方框中的一个点代表一类资源中的一个资源 
 - E是边集，它连接着P中的一个结点和R中的一个结点
 - $e = \langle p_i, r_j \rangle$ 是资源请求边 
 - $e = \langle r_j, p_i \rangle$ 是资源分配边 

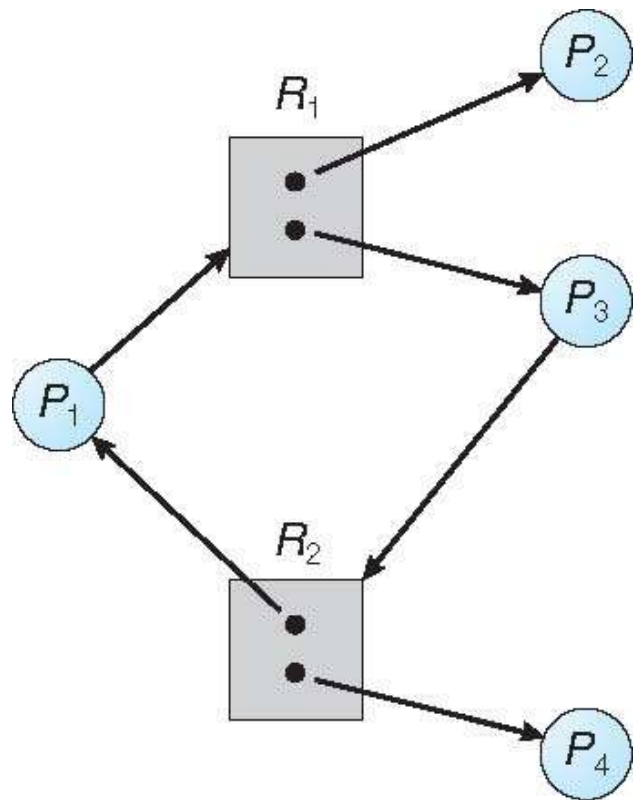
资源分配图例



存在死锁的资源图例



具有环且未死锁的资源分配图





基本事实

- 如果分配图没有环 \Rightarrow **NO deadlock!**
- 如果分配图包含环 \Rightarrow
 - 如果每个资源类型, 只包含一个资源实例, 则**死锁**
 - 如果每个资源有多个资源实例, 则只是存在**死锁的可能**, 不一定会死锁



5.3 处理死锁的基本方法

- 用于处理死锁的方法主要有：

- ① 忽略死锁。这种处理方式又称鸵鸟算法，指像鸵鸟一样对死锁视而不见。被大多数OS采用，因为死锁出现概率低，忽略死锁代价小。
- ② 预防死锁：设置某些限制条件，通过破坏死锁产生的四个必要条件之一来预防死锁。
- ③ 避免死锁：在资源的动态分配过程中，用某种方法来防止系统进入不安全状态。
- ④ 检测死锁及解除：系统定期检测是否出现死锁，若出现则解除死锁。



5.4 处理死锁的方法1：预防死锁

■ 预防死锁

- 通过破坏产生死锁的四个必要条件中的一个或几个条件，来防止发生死锁。
- 考虑破坏必要条件的可能：
 - 互斥条件
 - 请求和保持条件
 - 不可剥夺条件
 - 循环等待条件



5.4.1 互斥条件

- 破坏条件1：互斥条件
 - 使资源可同时访问，而非互斥使用
 - 如可重入程序、只读数据、时钟等
 - 但互斥对一些资源是固有的属性
 - 如可写文件、互斥锁，此条件往往不能破坏



5.4.2 请求和保持条件

- 破坏条件2：请求和保持条件
 - 思路：当每个进程申请一个资源时（可能成功或失败），它不能占有其他资源。
 - 方法一：要求进程一次申请它所需的全部资源，若有足够的资源则分配给进程，否则不分配资源，进程等待。这种方法称为**静态资源分配法**。
 - 方法二：允许进程仅在没有资源时才可申请资源。一个进程申请资源并使用，但是在申请更多资源时，**应释放已经分配的所有资源**。
 - 特点：
 - 简单且易于实现；
 - 但资源利用率低，进程延迟运行，可能发生饥饿。



5.4.3 不可剥夺条件

■ 破坏条件3：不可剥夺条件

- 对一个已获得某些资源的进程，若新的资源请求得不到满足，则它已占有的资源都可以被抢占。即这些资源都被**隐式释放**了。

- 例：进程A已经占有了资源a，并计划申请资源b，此时进程B也处于等待其他资源c的状态，如果：

- ① 资源b可用，则分配给进程A

- ② 资源b不可用，则检查资源b是否已经分配给进程B，如果：

- ① 资源b被进程B所占有，则抢占资源b

- ② 资源b不被任何一个处于等待资源的进程占有，则资源a也可被其他进程抢占。

- 负面：这种释放有可能造成已有工作的失效，重新申请和释放会带来新的系统开销

- 适用范围：常用于状态易于保存和恢复的资源，如CPU寄存器和内存资源，对于打印机、互斥信号量等不可使用



5.4.4 循环等待条件

- 破坏条件4：循环等待条件
 - 层次分配策略
 - 资源被分成多个层次
 - 当进程得到某一层的一个资源后，它只能再申请较高层次的资源
 - 当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源
 - 当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源
 - 也称为有序资源分配法

5.4.4 循环等待条件

- 层次策略的变种：按序分配策略

- 把系统的所有资源排一个顺序

- 如系统若共有 n 个进程,共有 m 个资源, 用 r_i 表示第 i 个资源, 于是这 m 个资源是:

$$r_1, r_2, \dots, r_m$$

- 规定:

- 进程不得在占用资源 $r_i (1 \leq i \leq m)$ 后再申请 $r_j (j < i)$
 - 即, 只能申请编号之后的资源, 而不许申请编号之前的资源, 从而避免资源申请的环路问题。

- 不难证明, 按这种策略分配资源时系统不会发生死锁。

为什么层次资源分配法可以防止死锁

■ 反证法:

设时刻 t_1 , 进程 P_1 处于等资源 r_{k1} 状态, 则 r_{k1} 必为另一进程占有, 假定是 P_2 所占用, 所以一定在某个时刻 t_2 , 进程 P_2 占有资源 r_{k1} 而处于永远等待资源 r_{k2} 状态。如此推下去, 系统只有有限个进程, 必有某个 n , 在时刻 t_n 时, 进程 P_n 永远等待资源 r_{kn} , 而 r_{kn} 必为前面某进程 P_i 占用($i < n$)。按照按序分配策略, 当 P_2 占用了 r_{k1} 后再申请 r_{k2} 必有: $k1 < k2$

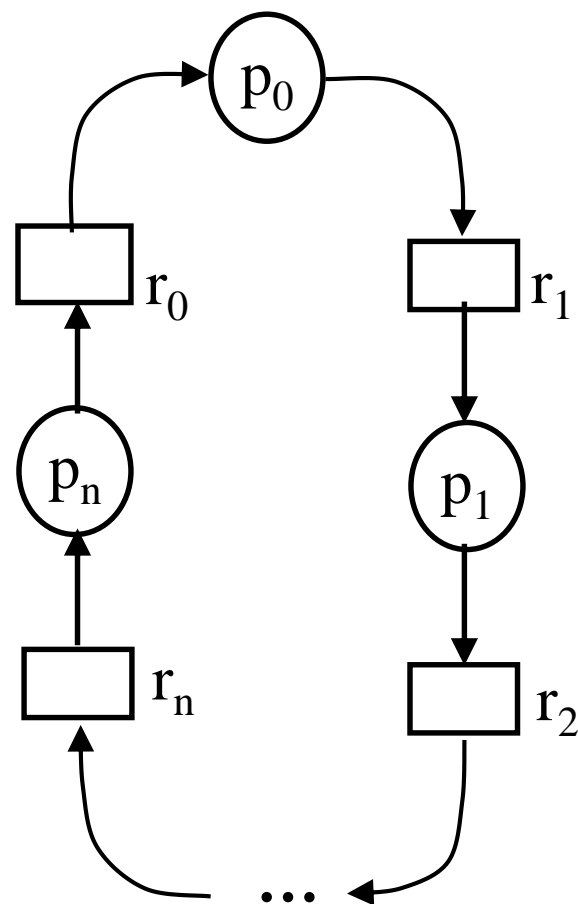
依此类推, 可得:

$$k2 < k3 < \dots < ki < \dots < kn$$

但由于进程 P_i 占有了 r_{kn} 却要申请 r_{ki} , 那么, 必定有:

$$kn < ki$$

这就产生了矛盾。所以层次分配策略可以防止死锁。





5.4.5 预防死锁的特点

■ 预防死锁

- 通过破坏产生死锁的四个必要条件中的一个或几个条件，来防止发生死锁。
- 特点：
 - 较易实现，广泛使用
 - 但限制较严，影响了系统的并发性，导致资源利用率低



5.5 处理死锁方法2：避免死锁

■ 死锁的避免

- 允许系统中存在前3个必要条件，通过合适的资源分配算法，确保不会出现第四个必要条件，从而避免死锁。
- 不是对进程随意强加规则，而是在**资源的动态分配**过程中实施
- 用某种方法**防止系统进入不安全状态**，从而避免死锁的发生
- 决策依据：已分配资源情况，当前申请资源情况，以及将来资源的申请与释放情况
- 决策结果：
 - 如果一个进程当前请求的资源会导致死锁，系统就拒绝启动这个进程
 - 如果一个资源分配会导致下一步死锁，系统就拒绝本次分配

5.5.1 安全状态

- **思路**：允许进程动态地申请资源，系统在进行资源分配之前，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，便将资源分配给进程，否则进程等待。
- **安全状态**
 - 是指系统能按某种顺序如 $\langle P_1, P_2, \dots, P_n \rangle$ 来为每个进程分配其所需的资源，直至最大需求，使每个进程都可以顺利完成，则称此时的系统状态为**安全状态**，称序列 $\langle P_1, P_2, \dots, P_n \rangle$ 为安全序列。



不安全状态

- 若某一时刻系统中**不存在一个安全序列**，则称此时的系统状态为**不安全状态**。
- 进入不安全状态后，便**可能**进入死锁状态；
 - 不是所有的不安全状态都能导致死锁，因为不安全状态有可能转变为安全状态
- 因此避免死锁的**本质**是使系统不进入不安全状态，而是**始终保持**在安全状态。



安全状态例

- T0时刻，系统资源状态如下：

进程	最大需求	已分配	需要	可用
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

这里存在一个安全序列吗？



安全状态例

- T0时刻，系统资源状态如下：

进程	最大需求	已分配	需要	可用
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

这时可用资源能满足P2的需要，P2获得运行需要的所有资源并能顺利运行结束。



P2运行结束的系统资源状态

进程	最大需求	已分配	需要	可用
P1	10	5	5	5
P2	4			
P3	9	2	7	

- 这时可用资源能满足P1的需要，P1获得运行需要的所有资源并能顺利运行结束。



P2、P1运行结束的系统资源状态

进程	最大需求	已分配	需要	可用
P1	10			10
P2	4			
P3	9	2	7	

- 这时可用资源能满足P3的需要，P3获得运行需要的所有资源并能顺利运行结束。
- 因此存在一个安全序列<P2、P1、P3>，系统状态安全。



由安全状态向不安全状态转换

- 若在T0之后，采取的是将1个资源分配给P3，则系统进入了不安全状态：

进程	最大需求	已分配	需要	可用
P1	10	5	5	2
P2	4	2	2	
P3	9	3	6	



由不安全状态向安全状态的转换

- 若在T1之后，P3会先主动释放一个资源，则系统进入了安全状态：

进程	最大需求	已分配	需要	可用
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

- **但是这种转换，并不是总能实现的**



死锁避免的算法

- 每个资源类型只有一个实例
 - 使用资源分配图机制
- 每个资源类型有多个实例
 - 使用银行家算法

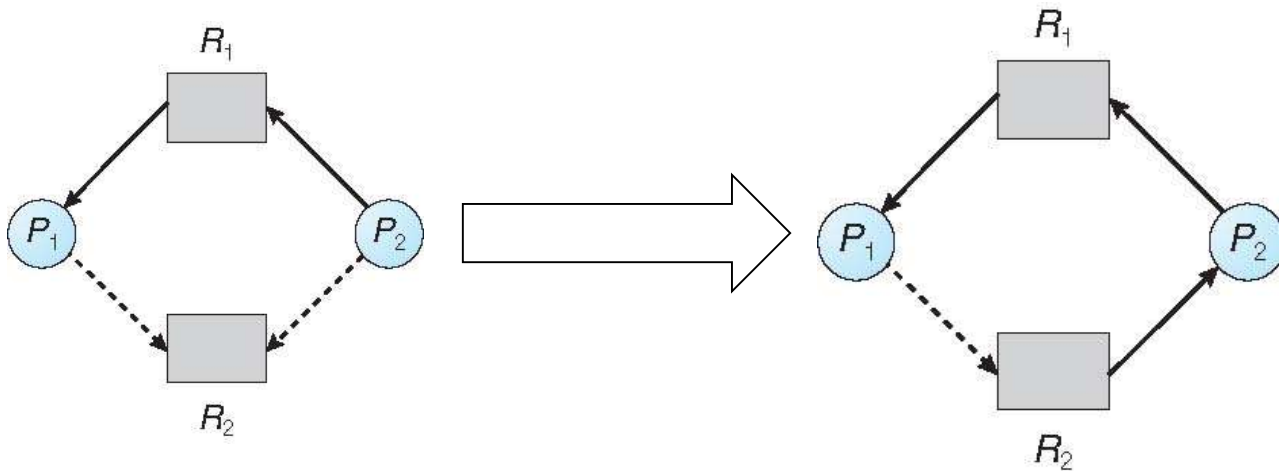


5.5.2 资源分配图机制

- 需求边(Claim edge): $P_i \rightarrow R_j$
 - 进程 P_i 可能在将来某个时候申请资源 R_j
 - 有向边为虚线
- 需求边可转化为请求边(Request edge)
 - 当进程请求该资源时
- 请求边可转化为分配边 (Assignment edge)
 - 当资源被分配给该进程
- 分配边可转化为需求边
 - 当资源被进程释放
- 系统中资源的需求, 要事先声明
 - 在进程开始执行前, 所有需求边应处于资源分配图中

资源分配图算法

- 假设 P_i 请求一个资源 R_j
 - 该请求被允许，仅当将请求边转换为分配边后，并不会造成资源分配图的环路
 - 即需要检测图中是否存在环
 - 例：当 P_2 申请资源 R_2





5.5.3 银行家算法

- 对于多类实例资源，最具代表性的死锁避免算法是Dijkstra的银行家算法。
- 背景：
 - 类比于银行业务：顾客类比于进程，顾客想借钱，钱为资源，银行为OS
 - 银行可借出的钱有限，每个顾客都有一定的银行信用额度
 - 顾客可以选择借一部分，但不能保证顾客在借走大量贷款后一定能偿还，除非他能获取全部贷款要求
 - 如果银行存在风险，没有足够的资金提供更多贷款让顾客偿还，则银行家就拒绝贷款给顾客
- 核心思想
 - 检查资源分配后是否会导致系统进入不安全状态
 - 手段：模拟分配资源，然后检查是否满足安全状态



基本数据结构

(1) 可用资源向量Available

- 假定系统中有 n 个进程 P_1 、 P_2 、...、 P_n ， m 类资源 R_1 、 R_2 、...、 R_m ，银行家算法
- 可利用资源向量Available是一个含有 m 个元素的数组，其中每一个元素代表一类资源的空闲资源数目。
- 如果 $Available(j) = k$ ，表示系统中现有空闲的 R_j 类资源 k 个。

(2) 最大需求矩阵Max

- 最大需求矩阵Max是一个 $n \times m$ 的矩阵，定义了系统中每个进程对 m 类资源的最大需求数目。
- 如果 $Max(i, j) = k$ ，表示进程 P_i 需要 R_j 类资源的最大数目为 k 。



基本数据结构Cont.

(3) 分配矩阵Allocation

- 分配矩阵Allocation是一个 $n \times m$ 的矩阵，定义了系统中每一类资源当前已分配给每一个进程的资源数目。
- 如果 $\text{Allocation}(i, j) = k$ ，表示进程 P_i 当前已分到 R_j 类资源的数目为 k 。
- Allocation_i 表示进程 P_i 的分配向量，由矩阵Allocation的第 i 行构成。

(4) 需求矩阵Need

- 需求矩阵Need是一个 $n \times m$ 的矩阵，它定义了系统中每一个进程还需要的各类资源数目。
- 如果 $\text{Need}(i, j) = k$ ，表示进程 P_i 还需要 R_j 类资源 k 个。 Need_i 表示进程 P_i 的需求向量，由矩阵Need的第 i 行构成。
- 三个矩阵间的关系：
$$\text{Need}(i, j) = \text{Max}(i, j) - \text{Allocation}(i, j)$$

资源请求算法

- 设 $Request_i$ 是进程 P_i 的请求向量， $Request_i(j) = k$ 表示进程 P_i 请求分配 R_j 类资源 k 个。
- 当 P_i 发出资源请求后，系统按下述步骤进行检查：
 - ① 如果 $Request_i \leq Need_i$ ，则转向步骤2；否则生成出错条件，此时进程请求超出了他的需求。
 - ② 如果 $Request_i \leq Available$ ，则转向步骤3；否则进程 P_i 转入等待。
 - ③ 试分配并修改数据结构：
 - ① $Available = Available - Request_i$ ；
 - ② $Allocation_i = Allocation_i + Request_i$ ；
 - ③ $Need_i = Need_i - Request_i$ ；
 - ④ 系统执行安全性算法，检查此次资源分配后得到的新状态是否安全。若安全，才正式分配；否则，试分配作废，让进程 P_i 等待。



安全性算法(1)

① 设置两个向量

- Work: 表示系统可提供给进程继续运行的各类空闲资源数目, 含有m个元素, 执行安全性算法开始时, 初始化 $Work = Available$ 。
- Finish: 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, $Finish(i) = false$; 当有足够资源分配给进程 P_i 时, 则令 $Finish(i) = true$ 。

② 从进程集合中找到一个能满足下述条件的进程i:

- $Finish(i) == false$;
- $Need_i \leq Work$;
- 如找到则执行步骤3; 否则执行步骤4。

安全性算法(2)

- ③ 当进程 P_i 获得资源后，可顺利执行直到完成，并释放出分配给它的资源，故应执行：
- $Work = Work + Allocation_i$;
 - $Finish(i) = true$;
 - Goto step 2 ;
- ④ 若所有进程的 $Finish(i)$ 都为true，则表示系统处于安全状态；否则，系统处于不安全状态

一个事实：这里的安全路径可能有多条！

一个新问题：是否存在，有分叉的安全路径情况下，安全性算法搜索到了一条不安全路径呢？

可以证明：只要存在一个序列不是安全序列，那么任意路径都不是安全序列。

只要有一个序列是安全序列，那么在算法进行过程中出现的任何分叉点所构成的其它序列就都是安全序列。

银行家算法例

- 假定系统中有5个进程 P0、P1、P2、P3、P4和三种类型的资源A、B、C，数量分别为12、5、9，在T0时刻的资源分配情况如下所示。

资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	8	5	3	1	1	0	7	4	3	3	3	2
P1	3	2	3	2	0	1	1	2	2			
P2	9	0	3	3	0	3	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	5	3	3	1	0	2	4	3	1			

T₀时刻的安全性检查

资源情况 进程	Work			Need			Alloc			Work+Alloc			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	3	3	2	1	2	2	2	0	1	5	3	3	true
P3	5	3	3	0	1	1	2	1	1	7	4	4	true
P4	7	4	4	4	3	1	1	0	2	8	4	6	true
P2	8	4	6	6	0	0	3	0	3	11	4	9	true
P0	11	4	9	7	4	3	1	1	0	12	5	9	true

- 从上述分析得知，T₀时刻存在着一个安全序列< P1、P3、P4、P2、P0 >，故系统是安全的，T₀是安全的



当P1请求资源

- P1发出请求向量 $\text{Request}_1(1, 0, 2)$ ，系统按银行家算法进行检查：
 - 1) $\text{Request}_1(1, 0, 2) \leq \text{Need}_1(1, 2, 2)$
 - 2) $\text{Request}_1(1, 0, 2) \leq \text{Available}(3, 3, 2)$
 - 3) 系统先假定可为P1分配资源，并修改Available、 Allocation_1 、 Need_1 向量，由此形成的资源变化情况如下所示。

为P1试分配资源后

资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	8	5	3	1	1	0	7	4	3	2	3	0
P1	3	2	3	3	0	3	0	2	0			
P2	9	0	3	3	0	3	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	5	3	3	1	0	2	4	3	1			

P1申请资源后的安全性检查

4) 再利用安全性算法检查此时系统是否安全，可得如下所示的安全性分析。

资源情况 进程	Work			Need			Alloc			Work+Alloc			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	2	3	0	0	2	0	3	0	3	5	3	3	true
P3	5	3	3	0	1	1	2	1	1	7	4	4	true
P4	7	4	4	4	3	1	1	0	2	8	4	6	true
P2	8	4	6	6	0	0	3	0	3	11	4	9	true
P0	11	4	9	7	4	3	1	1	0	12	5	9	true

- 从上述分析得知，可以找到安全序列<P1、P3、P4、P2、P0>，系统安全，可以分配。



当P4请求资源

- P4发出请求向量 $\text{Request}_4(3, 3, 0)$ ，系统按银行家算法进行检查：
 - 1) $\text{Request}_4(3, 3, 0) \leq \text{Need}_4(4, 3, 1)$
 - 2) $\text{Request}_4(3, 3, 0) > \text{Available}(2, 3, 0)$ ，让P4等待。



P0请求资源

- P0发出请求向量 $\text{Request}_0(0, 2, 0)$ ，系统按银行家算法进行检查：
 - 1) $\text{Request}_0(0, 2, 0) \leq \text{Need}_0(7, 4, 3)$
 - 2) $\text{Request}_0(0, 2, 0) \leq \text{Available}(2, 3, 0)$
 - 3) 系统先假定可为P0分配资源，并修改有关数据，如下所示。

为P0试分配资源后

资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	8	5	3	1	3	0	7	2	3	2	1	0
P1	3	2	3	3	0	3	0	2	0			
P2	9	0	3	3	0	3	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	5	3	3	1	0	2	4	3	1			

- 4) 再利用安全性算法检查此时系统是否安全。从上表中可以看出，可用资源Available (2, 1, 0) 已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。



安全性算法的思考

- **一个事实**：这里的安全路径可能有多条！
- **一个新问题**：注意，这个算法是没有回溯搜索的。是否存在，有分叉的安全路径情况下，安全性算法搜索到了一条不安全路径呢？
- **可以证明**：只要存在一个序列不是安全序列，那么任意路径都不是安全序列。只要有一个序列是安全序列，那么在算法进行过程中出现的任何分叉点所构成的其它序列就都是安全序列



5.5.4 避免死锁的特点

- 通过对资源分配进行动态的决策，从而避免环路等待条件
- 特点：
 - 有利于系统并发能力，以较弱的限制获得较高的利用率
 - 但实现有一定难度



5.6 处理死锁方法3:死锁的检测与解除

■ 基本思想

- 对资源的分配不施加限制，也不采取死锁避免措施，系统定时的运行“死锁检测”程序
- 判断系统内是否已经出现死锁，如果系统出现死锁，则采取某种措施解除死锁。

■ 特点：

- 死锁检测和解除可使系统获得较高的利用率
- 需要确定何时运行检测算法，执行频率如何



5.6.1 依据资源分配图来判定死锁

- 资源分配图与死锁状态的关系：
 - 如果资源分配图中无环路，则此时系统没有发生死锁
 - 如果资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生死锁，此时，环路是系统死锁的充分必要条件，环路中的进程就是死锁进程
 - 如果资源分配图中有环路，且涉及的资源类中包含多个资源，则环路的存在只是产生死锁的必要条件，而非充分条件，系统未必会发生死锁。
 - 如何求解死锁的充分条件？ **死锁定理**

资源分配图的简化

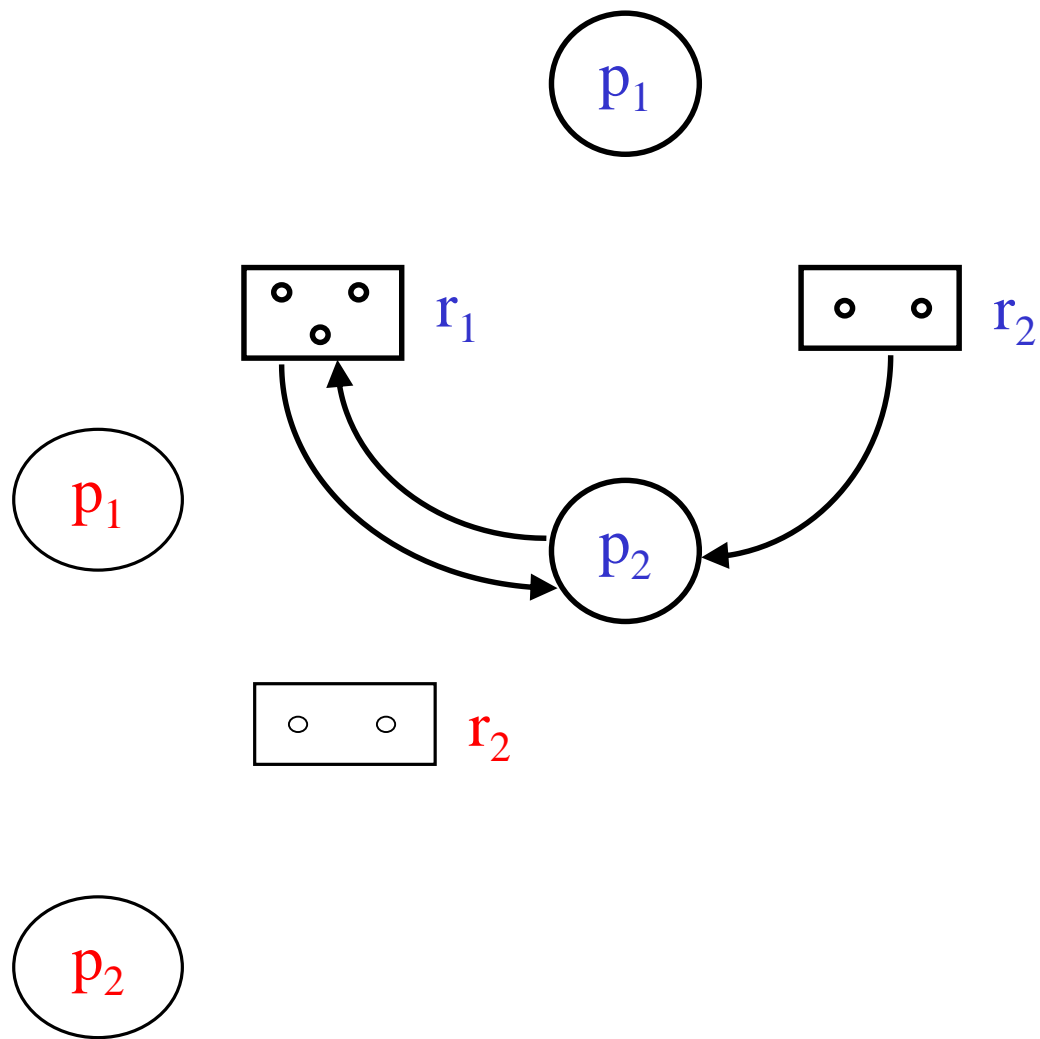
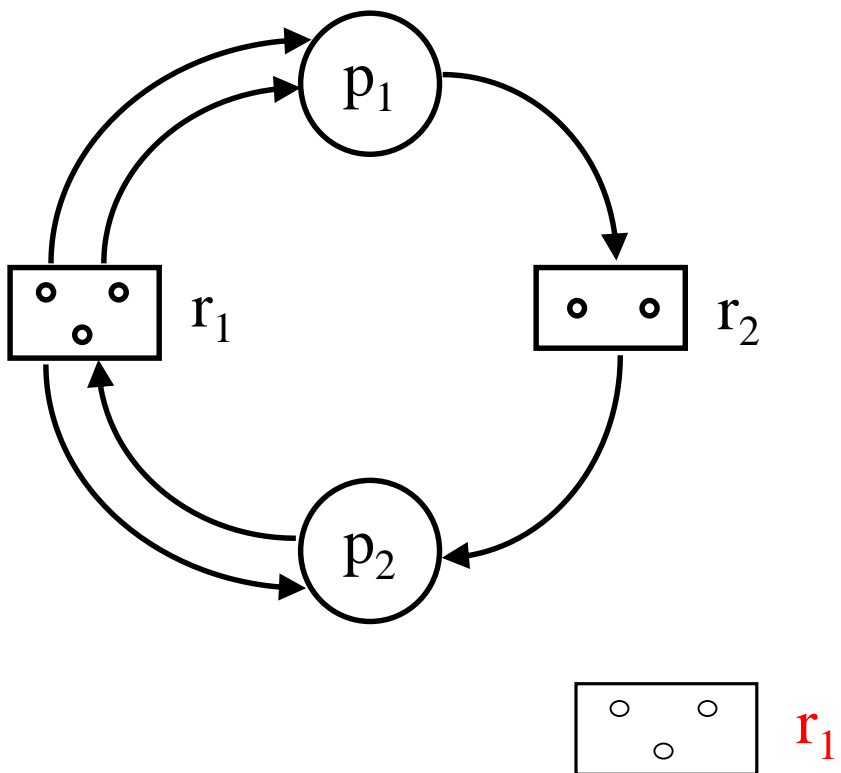
- ① 在资源分配图中，找出一个**既不阻塞又非孤立的进程结点** p_i
 - 非孤立：指该结点是一个有边与之相连的
 - 非阻塞：该结点没有因为资源请求与分配而导致等待
 - 即资源申请数量不大于系统已有空闲资源数量的进程
 - 亦即该进程节点的出边+被申请资源节点的出边 \leq 被申请资源的数量
- ② 当进程 p_i 获得了它所需要的全部资源，则能运行完成，然后释放所有资源
 - 即相当于消去 p_i 的所有请求边和分配边，使之成为孤立结点。
 - 进程 p_i 释放资源后，可以唤醒因等待这些资源而阻塞的进程，从而可能使原来阻塞的进程变为非阻塞进程。
- ③ 在进行一系列化简后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是**可完全简化的**；若不能使该图完全化简，则称该图是**不可完全简化的**。



死锁定理

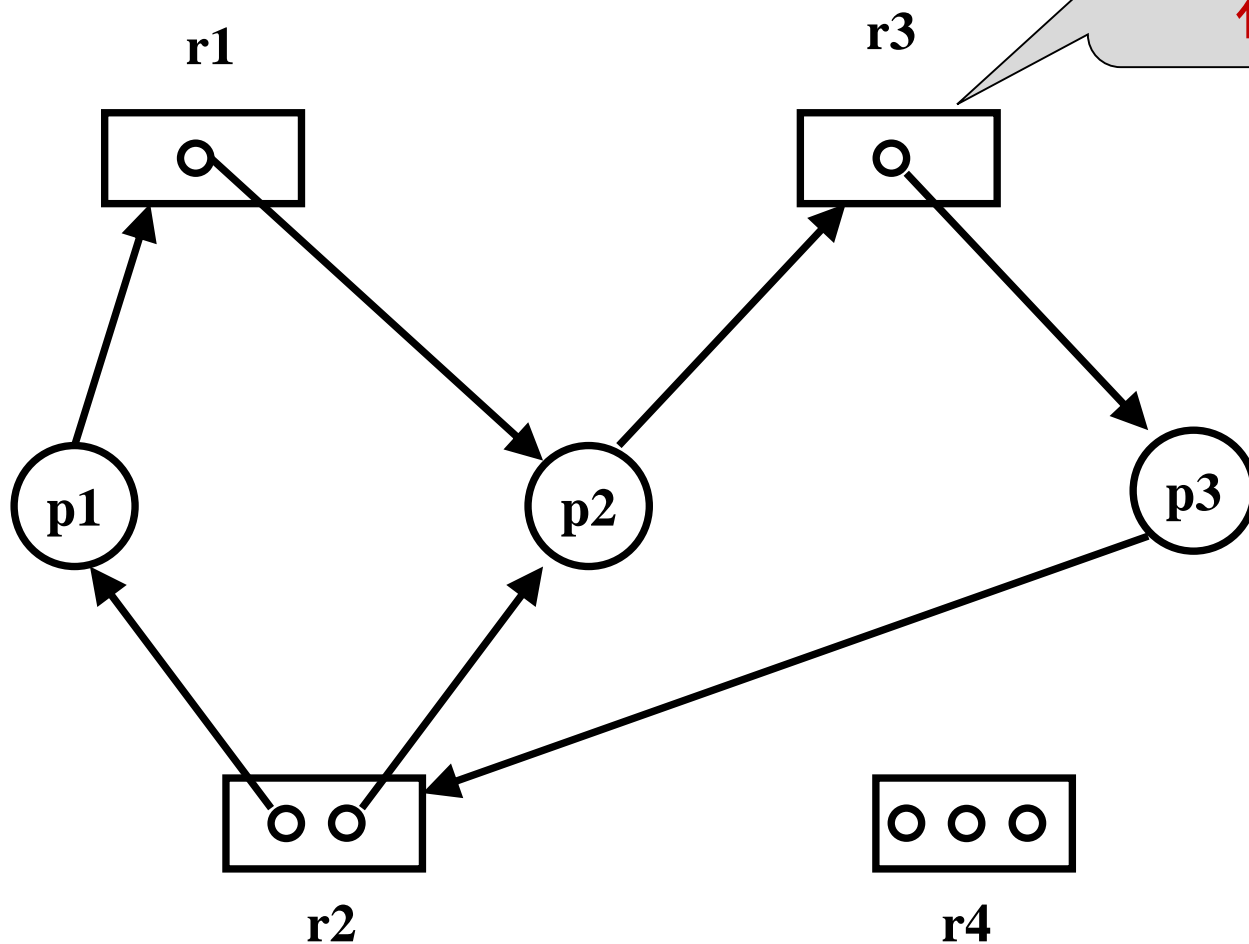
- 可以证明，所有的简化顺序将得到相同的不可简化图。
- 死锁定理：S为死锁状态的条件，当且仅当S状态的资源分配图是不可完全简化的。

资源分配图简化例



资源分配图简化例2

- 下图是否存在死锁?

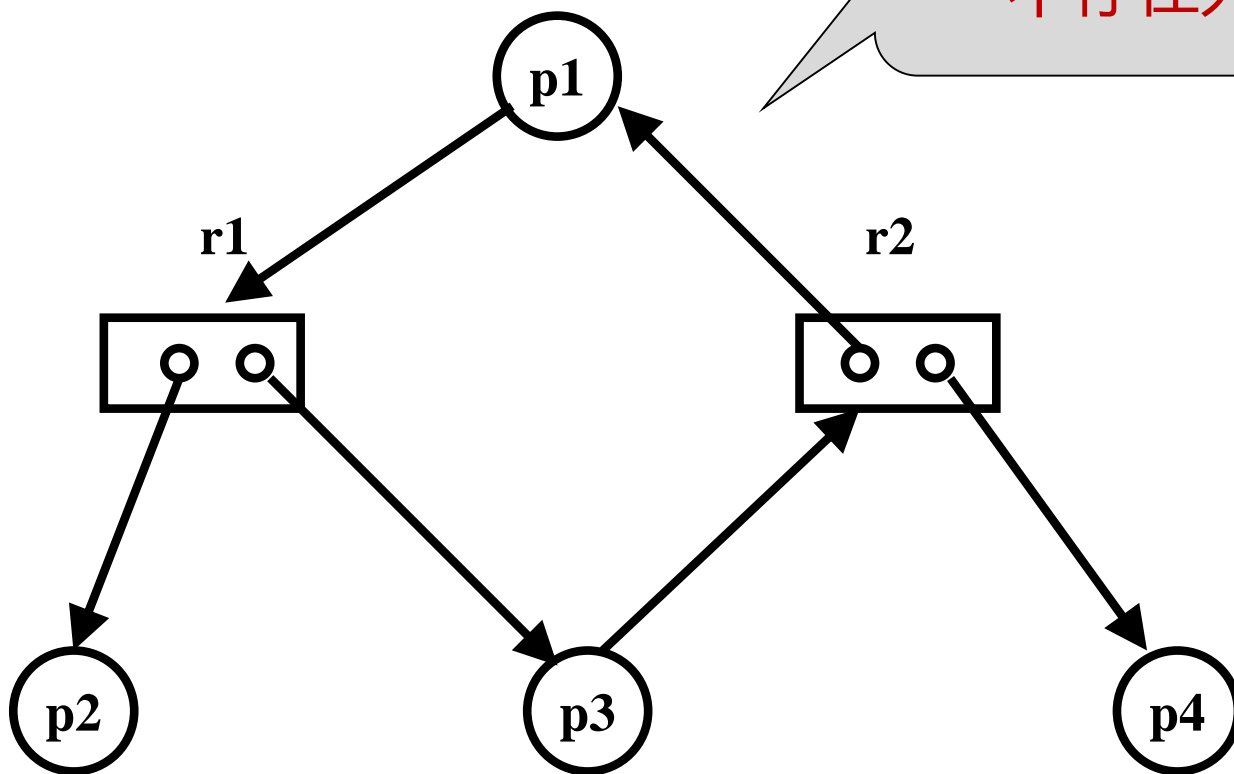


此图不可完全简化,
存在死锁。

资源分配图简化例3

- 下图是否存在死锁?

此图可以完全简化,
不存在死锁。





5.6.2 死锁的检测算法

- 类似银行家算法中安全性测试
- 可利用资源向量Available：表示m类资源中每类资源的可用数目。
- 请求矩阵Request：表示每个进程当前对各类资源的请求数目。
- 分配矩阵Allocation：表示每个进程当前已分配的资源数目。
- 工作向量Work：表示系统当前可提供资源数，长度为m
- 完成向量Finish：表示系统是否有足够的资源分配给进程，长度为进程数量

死锁检测的算法

$Allocation_i$ 表示进程 i 的每一类资源分配情况

$Request_i$ 表示进程 i 对每一类资源的请求

■ 并依然满足以下条件:

- $Request_i \leq Need_i$
- $Request_i \leq Available$

① 初始化, $Work = Available$; 对于所有的进程, 如果 $Allocation_i$ 为0, 则 $Finish[i] = true$, 否则为 $false$

② 寻找进程 i , 满足

- a. $Finish[i] == false$
- b. $Request_i \leq Work$
- c. 如果没有这样的 i , 则转第4步

③ 尝试回收资源

- a. $Work = Work + Allocation_i$
- b. $Finish[i] = true$
- c. 转第2步

④ 如果存在某个 i ($0 \leq i < n$), $Finish[i] == false$, 则系统死锁, 相应的进程 P_i 死锁。

这个算法和银行家算法中安全算法的共同点和区别点是什么?



5.6.3 死锁解除

- 一旦检测出系统中出现了死锁，就应将陷入死锁的进程从死锁状态中解脱出来，常用的死锁解除方法有：
 - 系统重启法：结束进程执行，重新启动系统
 - 进程终止：终止进程
 - 进程撤销法：撤消全部死锁进程，使系统恢复到正常状态
 - 逐步撤销法：按照某种顺序逐个撤消死锁进程，直到有足够的资源供其他未被撤消的进程使用，消除死锁状态为止。
 - 当逐步撤销时，如何选取终止哪些进程？计算代价
 - 资源抢占：剥夺陷于死锁进程占用资源，但不撤销进程，直至死锁解除。
 - 选择牺牲进程：计算代价
 - 回滚：将牺牲进程回滚到安全状态，完全回滚 or 回滚到打破死锁
 - 饥饿：如何保证资源不会总从同一个进程中被抢占？



处理死锁的综合方法

- 单独使用处理死锁的某种方法不能全面解决OS中遇到的所有死锁问题。综合解决的办法：
 - 将系统中的资源按层次分为若干类，对每一类资源使用最适合它的办法解决死锁问题。即使发生死锁，一个死锁环也只包含某一层次的资源，因此整个系统不会受控于死锁。
- 如：将系统的资源分为四个层次：
 - 内部资源：由系统本身使用，如PCB，采用有序资源分配法。
 - 主存资源：采用资源剥夺法。
 - 作业资源：可分配的设备 and 文件。采用死锁避免法。
 - 交换空间：采用静态分配法。



作业

1. V9, 7.1
2. V9, 7.8
3. V9, 7.9
4. 考虑一个有150个存储单元的系统，如下分配给三个进程：
 - P1最大需求70，已经占有25
 - P2最大需求60，已经占有40
 - P3最大需求60，已经占有45使用银行家算法，以确定下面的每个请求是否安全。如果安全，给出安全序列，如果不安全，给出结果分配情况
 - (1) P4进程到达，P4最大需求60，最初请求25
 - (2) P4进程到达，P4最大需求60，最初请求35
5. 简化图2、3，怎么简化的，画出过程。