第7章　死锁

第7章 死锁

7.1 死锁的引出

7.2 死锁产生的原因和必要条件

7.3 处理死锁的基本方法

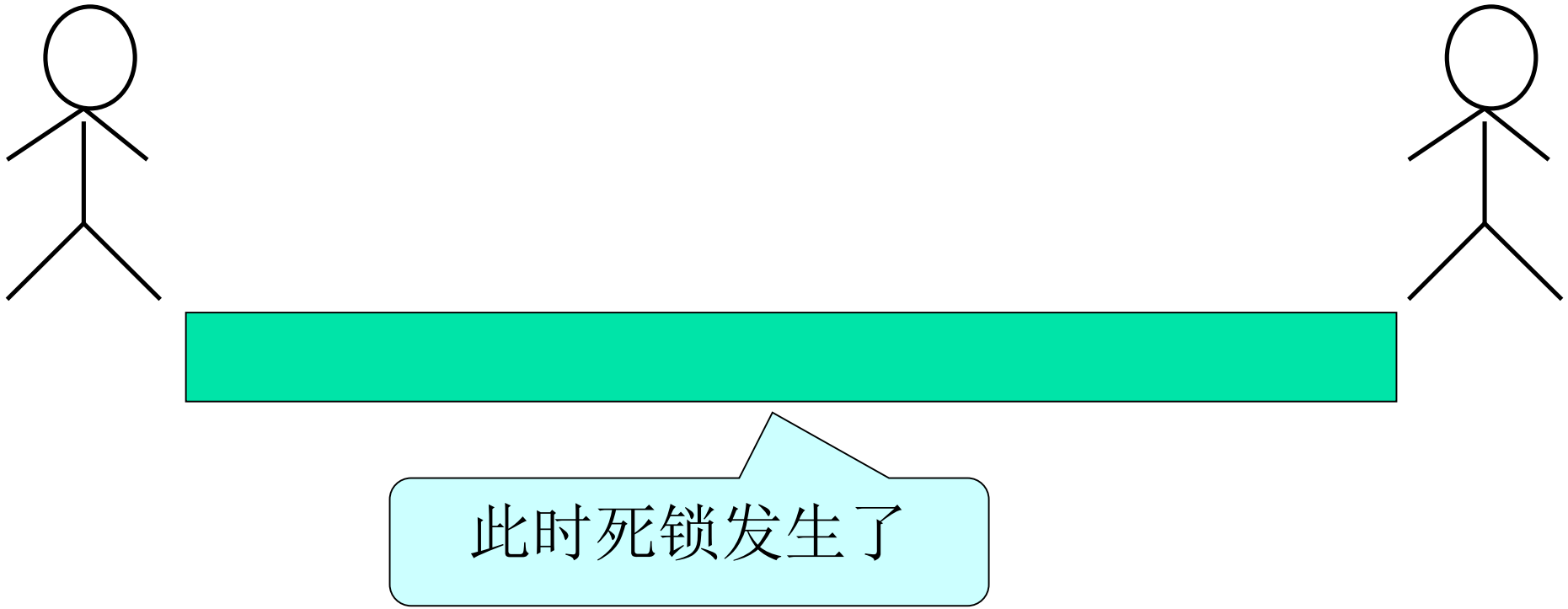
- 7.3.1预防死锁
- 7.3.2 避免死锁
- 7.3.3 死锁检测和解除
- 7.3.4 综合方法

7.1 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

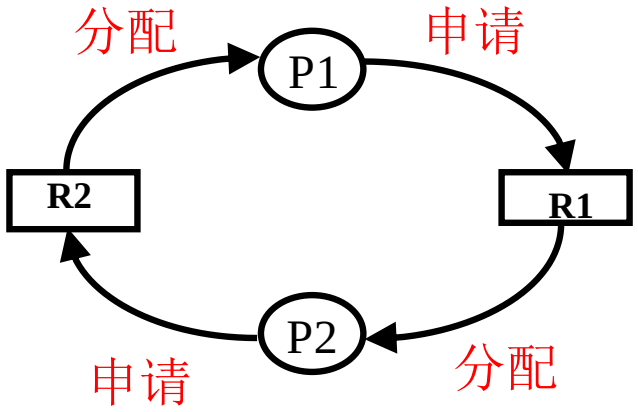
- 假设一条河上有一座独木桥，若桥两端的人相向而行……



进程推进顺序不当产生死锁

- 例1:设系统有打印机R1、读卡机R2各一台，被进程P1和P2共享。两个进程并发执行，按下列次序请求和释放资源：

进程P1	进程P2
请求读卡机R2	请求打印机R1
请求打印机R1	请求读卡机R2
释放读卡机R2	释放读卡机R2
释放打印机R1	释放打印机R1



PV操作使用不当产生死锁

例2:

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);

操作系统中的死锁概念

- 死锁是指多个进程因竞争系统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向前推进。

- 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。

7.2 死锁产生的原因和必要条件

- 死锁产生的原因是与资源的使用相关。下面介绍资源分类。
  - 可剥夺资源与非可剥夺资源
  - 永久性资源和消耗性资源

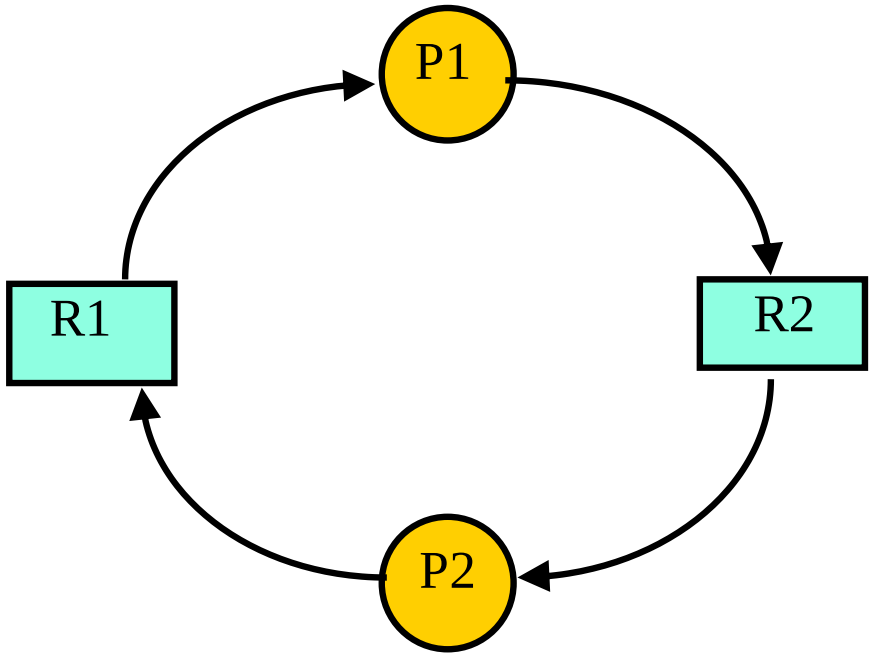
可剥夺和非剥夺资源

- 可剥夺资源是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，存储器。
- 非剥夺资源又称不可剥夺资源，是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！

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。



永久性资源和消耗性资源

- 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。

竞争消耗性资源引起的死锁

- 竞争非剥夺资源例。
  - 如：进程P1等待进程P3的消息S3来到后再向进程P2发送消息S1；P2又要等待P1的消息S1来到后再向P3发送消息S2；而P3也要等待P2的消息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。

死锁的4个必要条件（Coffman 1971）

- 互斥条件：在一段时间内某资源仅为一个进程所占有。
- 不剥夺条件：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。
- 请求和保持条件：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。
- 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
- 死锁与部分进程及资源相关
- 通常死锁至少涉及两个进程

7.3 处理死锁的基本方法

- 用于处理死锁的方法主要有：
  - 忽略死锁。大多数操作系统的做法；死锁产生的原因主要是资源使用方法的不当，属用户级错误
  - 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。
  - 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。
  - 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。

7.3.1 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
    - 互斥条件
    - 不可剥夺条件
    - 请求和保持条件
    - 循环等待条件
- 特点：较易实现，广泛使用，但限制较严，资源利用率低。

破坏条件1：互斥

- 互斥是设备本身固有的属性，此条件不能破坏。

破坏条件2：不可剥夺

- 若一个进程占有资源，申请另一个不可以立即分配的资源，那么已获得的资源可被抢占

- 局限：通常用于状态可以保存和恢复的资源
  - CPU寄存器、内存
  - 不适用于打印机、磁盘驱动器等

破坏条件3：请求和保持

- 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。（静态资源分配法）
- 特点
  - 简单、安全且易于实现；
  - 资源利用率低，进程延迟运行。
- 要求进程在没有资源时才可申请资源，若继续申请资源，必须释放已分配的所有资源

破坏条件4：循环等待

- 层次分配策略：资源被分成多个层次，且要求进程按照层次递增的顺序来申请资源
  - 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
  - 当进程要申请同一类型的多个资源实例，必须一起申请。
  - 那么，当进程得到某一层的一个资源后，它想申请同层或较低层的另一个资源时，必须先释已占用的那个资源，…
- 也称为有序资源分配法

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，例如，系统若共有n个进程,共有m个资源，用ri表示第i个资源，于是这m个资源是：

  r1,r2……,rm

- 规定：进程不得在占用资源ri(1≤i≤m)后再申请rj(j<i)，即只能申请编号之后的资源，而不许可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。

按序分配有什么缺点？

为什么层次资源分配法可以防死锁

- **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，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

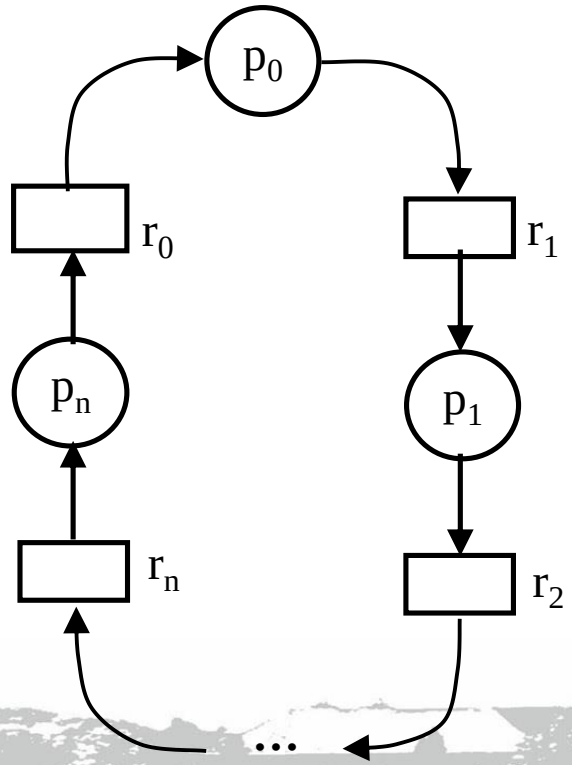
依此类推，可得：

k2 < k3 < …<ki<… < 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。



死锁预防总结

- 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、吞吐量低

7.3.2 死锁避免

- 死锁避免
  - 允许进程动态地申请资源
  - 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。
    - 系统需要掌握进程后续如何申请资源的附加信息；对资源的分配需要动态考虑可用资源、已分配资源、将来申请和释放的资源；动态检测资源分配状态，确保循环等待不成立，**防止系统进入不安全状态**
- 特点：以较弱的限制获得较高的利用率，但实现有一定难度。

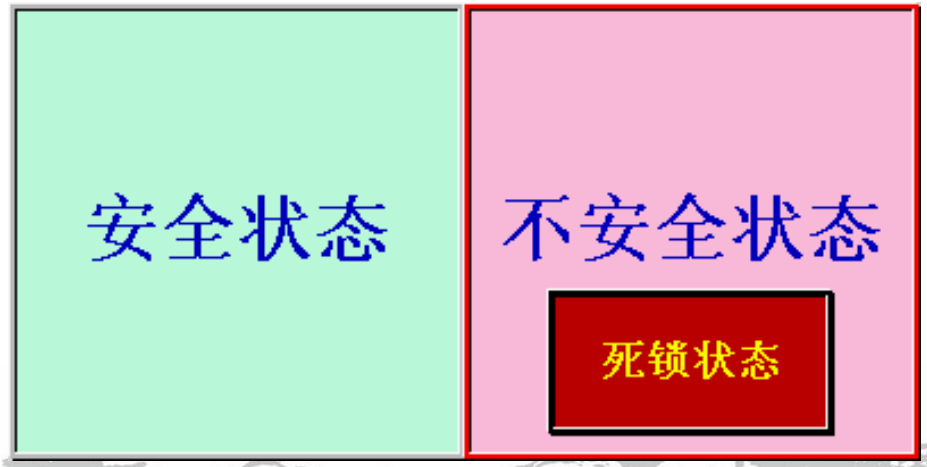
安全状态

- 安全状态
  - 系统能按某种顺序，如< P1、P2… 、Pn>，来为每个进程分配其所需的资源，直至最大需求，
  - 按此顺序，每个进程都可以顺利完成，
  - 此时的系统状态为安全状态，称序列< P1、P2、…、Pn >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；
- 因此，避免死锁的本质是使系统不进入不安全状态。



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，又将1个资源分配给了P3，

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	

则系统进入了不安全状态。

银行家算法

- 最具代表性的死锁避免算法（Dijkstra提出）
- 以银行借贷分配策略为基础，判断并保证系统处于安全状态
  - 客户在第一次申请贷款时，声明所需最大资金量，在满足所有贷款要求并完成项目时，及时归还
  - 在客户贷款数量不超过银行拥有的最大值时，银行家尽量满足客户需要
  - 类比
    - 银行家与操作系统；
    - 资金与资源；
    - 客户与申请资源的进程

银行家算法

- 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，银行家算法中使用的数据结构如下：

1）可用资源向量Available

- 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×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。

3）分配矩阵Allocation

- 分配矩阵Allocation是一个n×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×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：

$$Need(i，j) = Max(i，j) - Allocation(i，j)$$

银行家算法

- 银行家算法的核心思想
  - 资源分配后是否会导致系统处于不安全状态
- 银行家算法的主要步骤
  - 预分配资源，然后检查是否满足安全状态

- 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：

银行家算法描述

1）如果$Request_i \leq Need_i$，则转向步骤2；否则出错。

2）如果$Request_i \leq Available$，则转向步骤3；否则P_i等待。

3）试分配并修改数据结构：

$Available = Available - Request_i$；

$Allocation_i = Allocation_i + Request_i$；

$Need_i = Need_i - Request_i$；

4）系统执行安全性算法，检查此次资源分配是否安全。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，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，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；
  - $Need_i \leq Work$；
  - 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。

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			

T_0时刻的安全性

- 利用安全性算法对T_0时刻的资源分配情况进行分析，可得如下所示的T_0时刻的安全性分析。

T_0时刻的安全性检查

Need0：7,4,3　Need1：1,2,2　Need2：6,0,0　Need3：0,1,1　Need4：4,3,1
Alloc0：1,1,0　Alloc1：2,0,1　Alloc2：3,0,3　Alloc3：2,1,1　Alloc4：1,0,2
Avail 332　13420

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

T_0时刻是安全的

- 从上述分析得知，T_0时刻存在着一个安全序列< P1、P3、P4、P2、P0 >，故系统是安全的。

P1请求资源

- P1发出请求向量$Request_1$ (1，0，2)，系统按银行家算法进行检查：
  - 1）$Request_1(1，0，2) \leq Need_1(1，2，2)$
  - 2）$Request_1(1，0，2) \leq Available(3，3，2)$
  - 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

- 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	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

Need0：7,4,3　Need1：0,2,0　Need2：6,0,0　Need3：0,1,1　Need4：4,3,1
Alloc0：1,1,0　Alloc1：3,0,3　Alloc2：3,0,3　Alloc3：2,1,1　Alloc4：1,0,2
Avail 230　13420

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

可以为P1分配资源

- 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。

P4继续请求资源

- P4发出请求向量$Request_4(3，3，0)$，系统按银行家算法进行检查：
  - 1）$Request_4(3，3，0) \leq Need_4(4，3，1)$
  - 2）$Request_4(3，3，0) > Available(2，3，0)$，让P4等待。

P0继续请求资源

- P0发出请求向量$Request_0$ (0，2，0)，系统按银行家算法进行检查：
  - 1）$Request_0(0，2，0) \leq Need_0(7，4，3)$
  - 2）$Request_0(0，2，0) \leq Available(2，3，0)$
  - 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

- 4）再利用安全性算法检查此时系统是否安全。从下表中可以看出，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。

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			

7.3.3 死锁的检测及解除

- 如果系统既不采用死锁预防算法也不采用死锁避免算法，那么可能会出现死锁。
- 为此，系统必须通过检测机构及时地检测出死锁的发生，然后采取某种措施解除死锁。

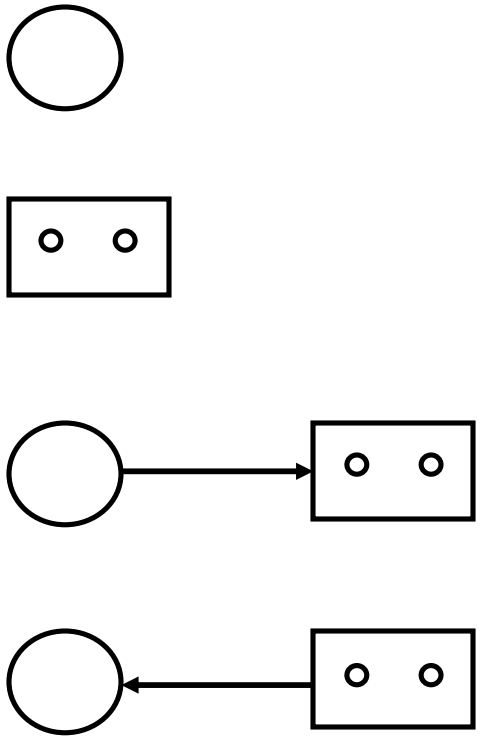
- 特点：死锁检测和解除可使系统获得较高的利用率，但是实现难度最大。
  - 维护信息和执行检测算法的开销，死锁恢复带来的损失

资源分配图

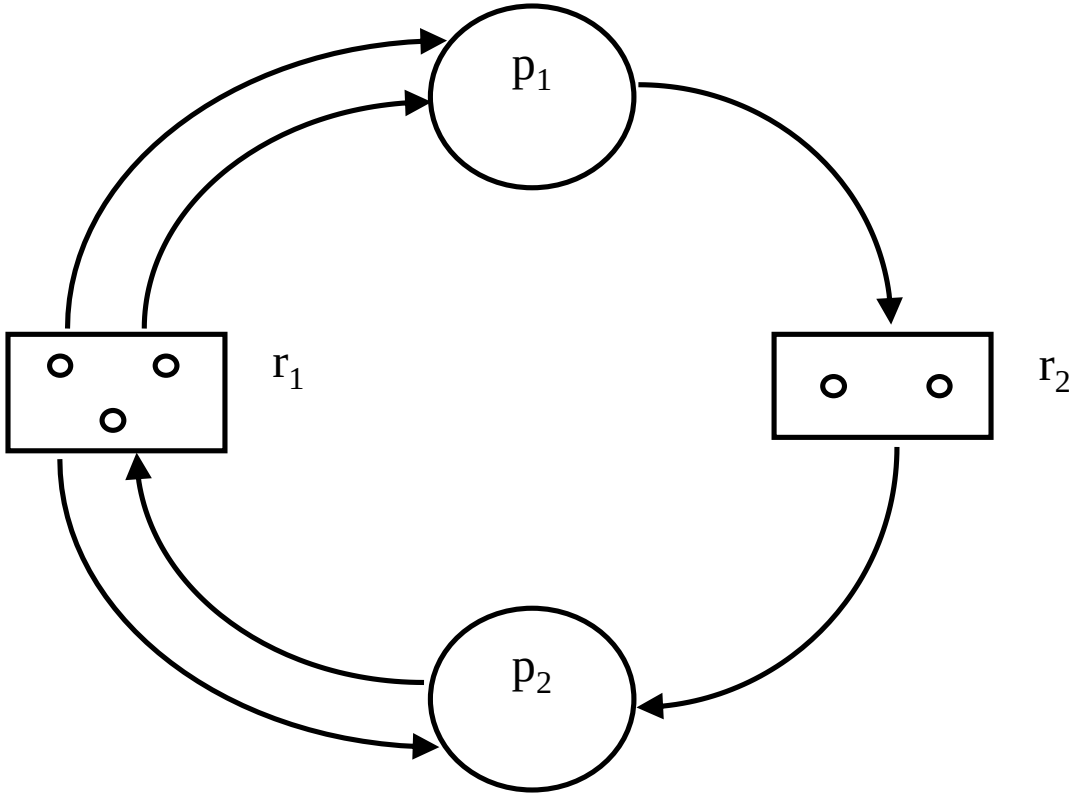
- 系统死锁可以利用资源分配图描述
  - 资源分配图又称进程—资源图，由一组结点N和一组边E所构成的有向图：
    - N被分成两个互斥的子集：进程结点子集P = {p1，p2，…，pn}，资源结点子集R = {r1，r2，…，rm}。
    - E是边集，它连接着P中的一个结点和R中的一个结点，e = <pi，rj>是资源请求边，e = <rj，pi>是资源分配边。

资源分配图 -- 图例

- 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实例；
- 从进程指向资源的有向边代表进程**请求一个**该类资源；
- 从资源指向进程的有向边代表该类资源中有**一个已经分配**给该进程。



资源分配图 — 举例



死锁判定法则

- 将资源分配图简化可以检测系统状态S是否为死锁状态，方法如下：
  - 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
    - **即从进程集合中找到一个有边与之相连，且资源申请数量不大于系统已有空闲资源数量的进程**
    - **即该进程节点的出边+被申请资源节点的出边≤被申请资源的数量**
  - 进程p_i获得了它所需要的全部资源，能运行完成，然后释放所有资源。
    - 这相当于消去p_i的所有请求边和分配边，使之成为孤立结点。

死锁判定法则

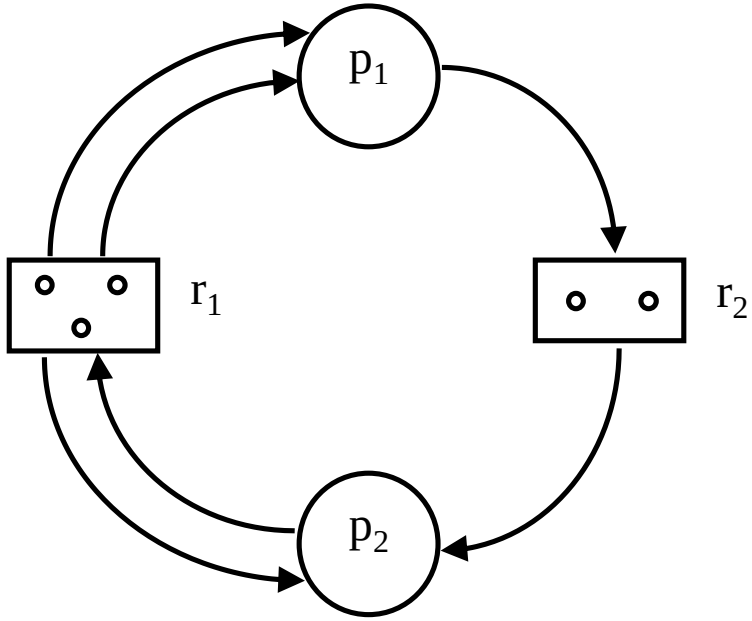
② 进程p_i释放资源后，可以唤醒因等待这些资源而阻塞的进程，从而可能使原来阻塞的进程变为非阻塞进程。

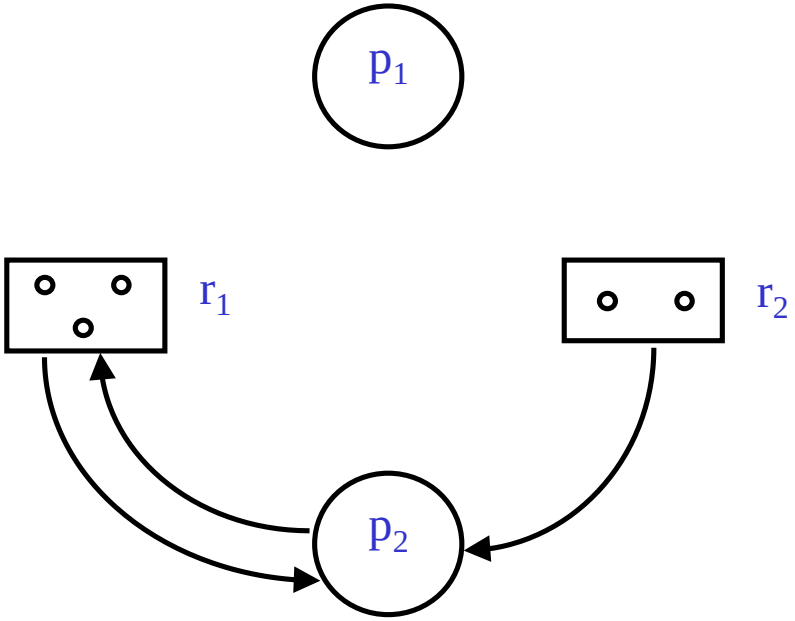
③ 在进行一系列化简后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是**可完全简化的**；若不能使该图完全化简，则称该图是**不可完全简化的**。

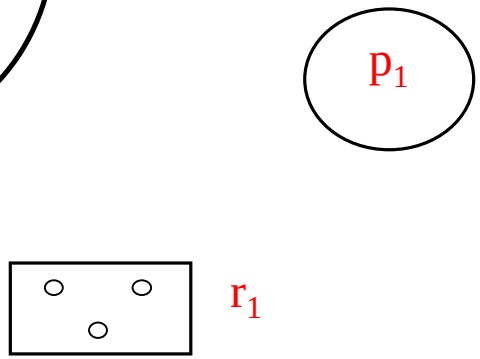
死锁定理

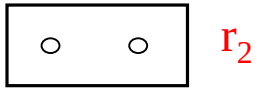
- **可以证明**：所有的简化顺序将得到相同的不可简化图。
- S为死锁状态的条件是当且仅当S状态的资源分配图是不可完全简化的。该条件称为死锁定理。

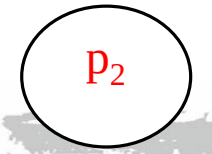
资源分配图简化例





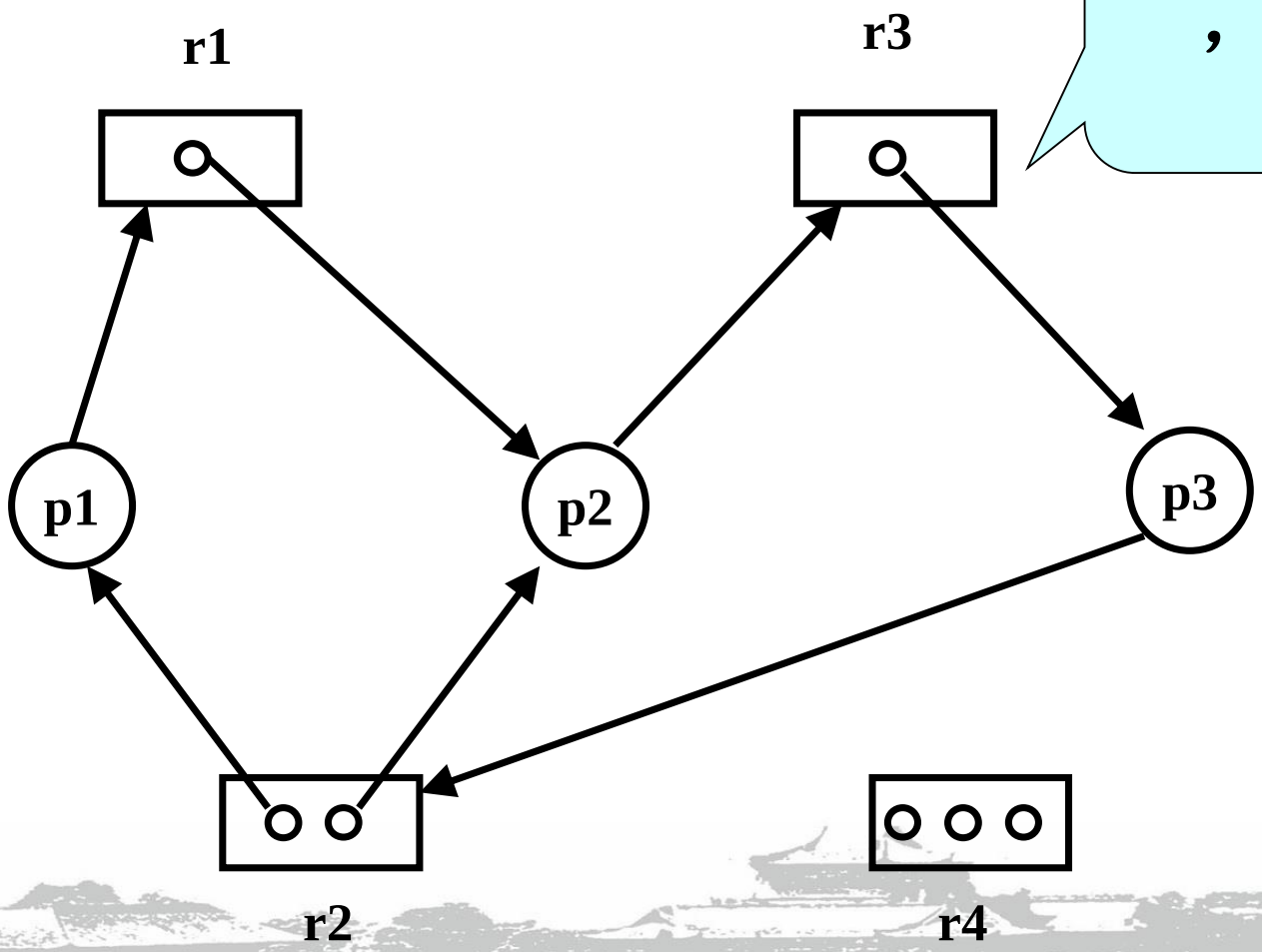






资源分配图简化例2

- 下图是否存在死锁?

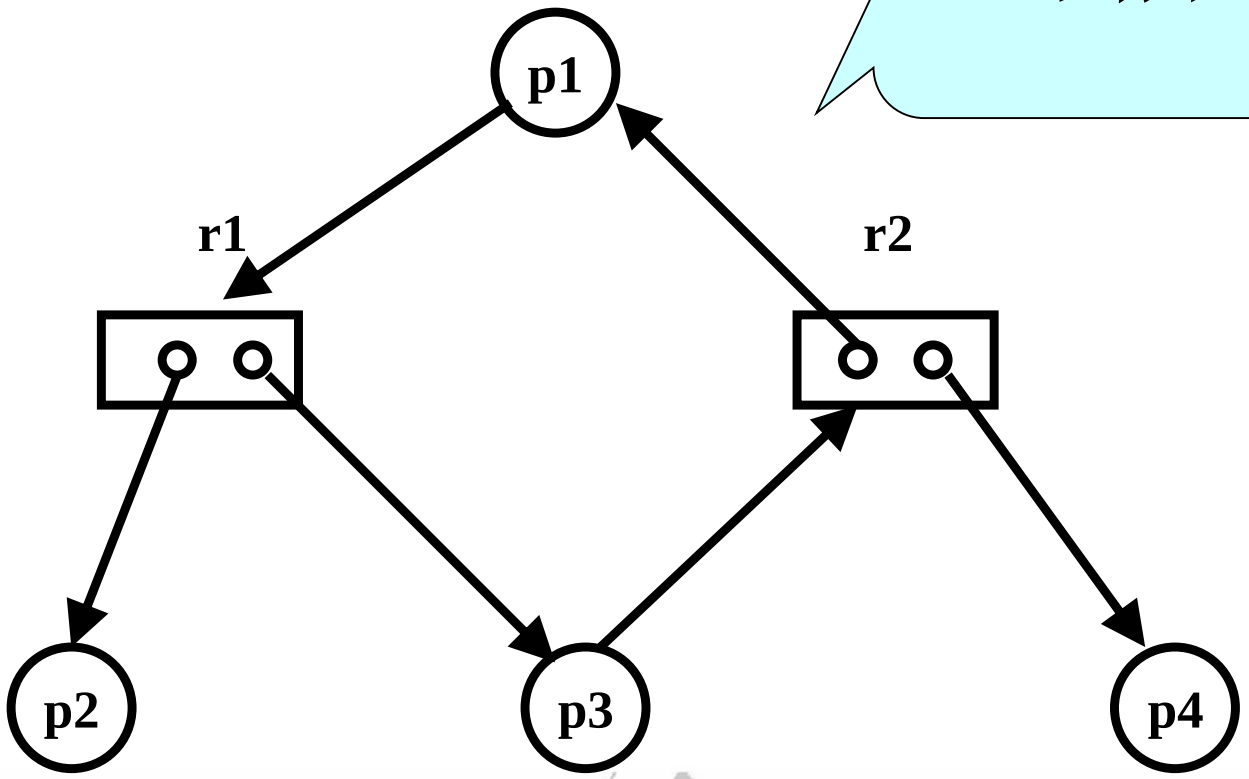


此图不可完全简化，存在死锁。

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。



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。
  - $Resquest_i$ 表示进程i对每一类资源的请求
- 分配矩阵Allocation：表示每个进程当前已分配的资源数目。
  - $Allocation_i$ 表示进程i的每一类资源分配情况
- 工作向量Work：表示系统当前可提供资源数。
- 进程集合L：记录当前已不占用资源的进程。

死锁检测的算法

① 初始化，设置两个向量Work和Finish并置初值：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。

why?

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？

死锁检测举例

- 考虑这样一个系统，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7、2、6。假定在T_0时刻的资源分配情况如下所示。

资源情况 / 进程	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

死锁检测举例（续）

- T0时刻是否死锁?
- 若现在P2又请求资源类型C的一个实例，系统是否死锁?

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。
  - 资源剥夺法：剥夺陷于死锁进程占用资源，但不撤销进程，直至死锁解除。
  - 进程回退：保存检查点，所有进程回退，直到足够解除死锁。

7.3.4 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。
- 综合解决的办法是：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。

综合方法例

- 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：可分配的设备和文件。采用死锁避免法。
  - 交换空间：采用静态分配法。

课堂练习

- 1、考虑一个共150个某类互斥资源的系统，如下分配三个进程，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，以确定以下每个请求是否安全。如果安全，找出安全序列；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个
  - （2）P4进程到达，P4最大需求60，最初请求35个

课堂练习

2、某系统有三类非剥夺性资源，其中r_1类有2个、r_2类有2个、r_3类有4个；当前三个进程（P_1、P_2、P_3）对资源的占用和请求情况如下表，判断当前是否发生死锁。

进程	占用情况			请求情况		
	r_1	r_2	r_3	r_1	r_2	r_3
P_1	1	0	2	1	0	0
P_2	0	0	2	0	1	0
P_3	0	2	0	2	0	1

作业

V9 教材 P228

1. 7.1
2. 7.7
3. 7.8
4. 7.13