上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L6. 死锁

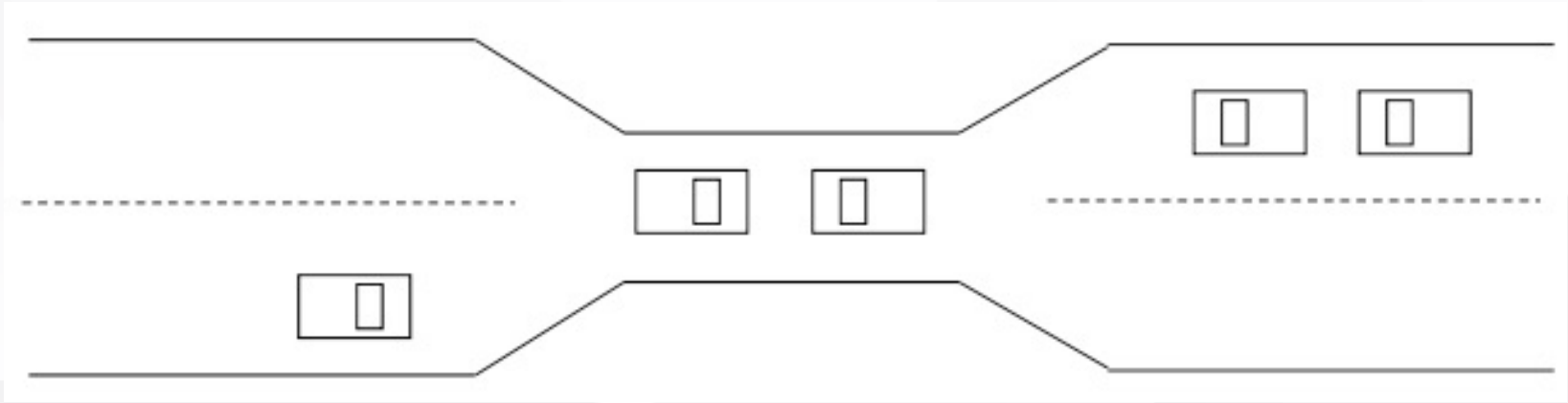
宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

背景

- 桥的每一部分可以看作一个资源
- 当两辆车同时从不同方向上桥时，就会发生死锁



死锁问题

- 系统中有两个磁盘，进程P1和P2分别拥有一个，并需要另一个
- 信号量S和Q，均初始化为1

P_0	P_1
① wait (S);	② wait (Q);
③ wait (Q);	④ wait (S);

- 死锁：一系列被阻塞的进程持有部分资源，同时需要另一部分资源，这些资源被在该系列中的进程所持有
- 当死锁时，进程永远不能完成，系统资源被阻碍使用，以致于阻止了其他作业开始执行

系统模型

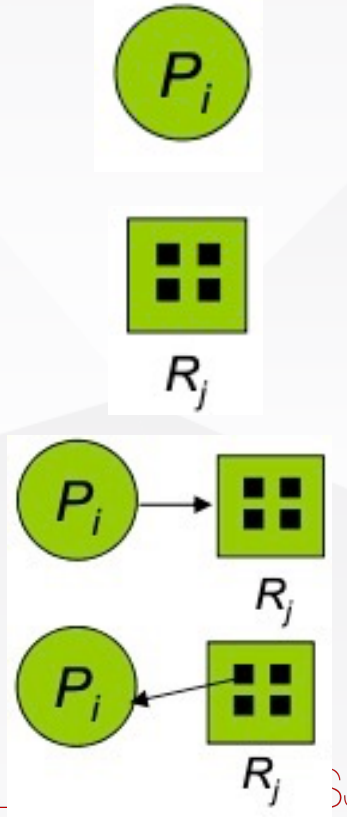
- 进程：P1、P2、P3...
- 系统资源：R1、R2、R3...
  - CPU、内存、文件、I/O设备
- 每种资源类型有W个实例
  - 如果一个系统有两个CPU，那么资源类型CPU 就有两个实例
- 进程只能按如下顺序使用资源
  - 申请
  - 使用
  - 释放

死锁特征

- 如果一个系统中以下四个条件同时成立，就能引起死锁
  - 互斥：至少有一个资源必须处于非共享模式，即一次只能有一个进程可使用
  - 占有并等待：一个进程应占有至少一个资源，并等待另一个资源，而该资源为其他进程所占有
  - 非抢占：资源不能被抢占
  - 循环等待：有一组等待进程（P0-Pn），P0等待的资源被P1占有，P1等待的资源被P2占有，P2等待的资源被P3占有，...

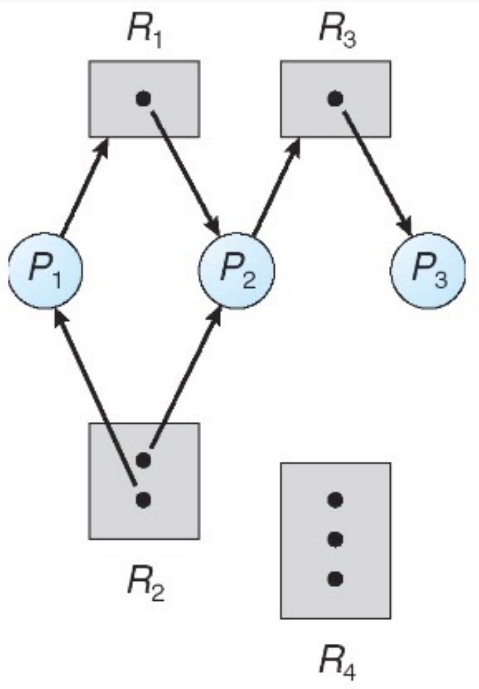
资源分配图

- 系统资源分配图的有向图可以更精确地描述死锁
  - 一组节点集合V和边集合E
  - 节点集合V被分为两类
    - P = {P1, P2, ..., Pn} 进程
    - R = {R1, R2, ..., Rm} 资源
      - 矩形表示资源类型，由于资源类型Rj可能有多个实例，所以矩形内的点点数量表示实例数量
  - 边集合E被分为两类
    - 申请边 Pi->Rj
    - 分配边 Rj->Pi



资源分配图 实例

- P={P1,P2,P3}
- R={R1,R2,R3,R4}
- 资源实例
  - W1=W3=1
  - W2=2
  - W4=3



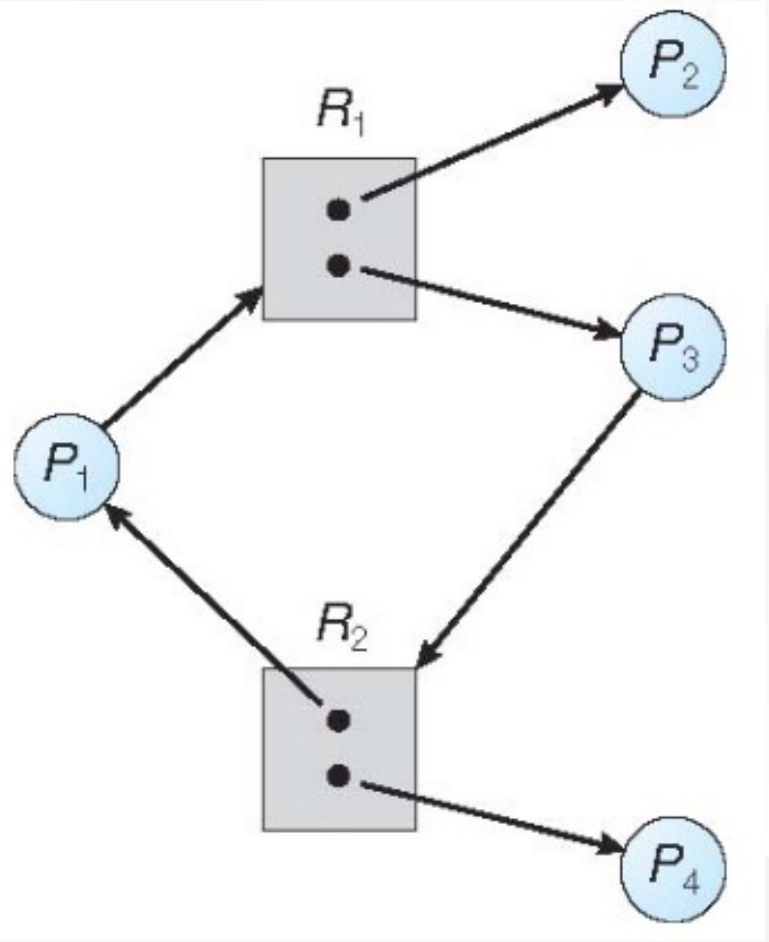
- E = {P1->R1, P2->R3, R1->P2, R2->P2, R2->P1, R3->P3}
- 根据资源分配图的定义，可以证明：如果分配图没有环，那么系统就没有进程死锁。如果分配图有环，那么**可能**存在死锁

资源分配图 实例

- 根据资源分配图的定义，可以证明：如果分配图没有环，那么系统就没有进程死锁。如果分配图有环，那么**可能**存在死锁
- 如果每个资源类型刚好有一个实例，那么有环就意味着已经出现死锁。如果环上的每个类型只有一个实例，那么就出现了死锁 。环上的进程就死锁。在这种情况下，图中的环就是死锁存在的充分且必要条件
- 如果每个资源类型有多个实例，那么有环并不意味着已经出现了死锁。在这种情况下， 图中的环就是死锁存在的必要条件而不是充分条件

资源分配图 实例

- 有一个环
  - P1->R1->P3->R2->P1
- 但没有死锁，因为进程P4可能释放资源R2的实例
- 总结：如果资源分配图没有环，那么系统就不处于死锁状态。如果有环，那么系统有可能处于死锁



死锁处理方法

- 处理死锁问题有三种方法
  - 通过协议来预防或避免死锁，确保系统不会进入死锁状态
  - 可以允许系统进入死锁状态，然后检测它，并加以恢复
  - 可以忽视这个问题，认为死锁不可能在系统内发生
- 死锁预防：确保至少有一个必要条件不成立，通过限制申请资源
- 死锁避免：通过事先得到有关进程申请资源和使用资源的额外信息，确定进程是否应该等待
- 死锁检测+恢复

死锁预防

- 互斥条件：必须成立，计算机中常有资源是非共享的。若资源为共享的，则不会参与死锁，例如只读文件，如果有多个进程试图打开同一个只读文件，它们可以同时访问。一般不能通过否定互斥条件来预防死锁，因为有的资源本身就是非共享的
- 持有且等待： 应确保当每个进程申请一个资源时，它不能占有其他资源
  - 每个进程在执行前申请并获得所有资源
  - 允许进程仅在没有资源时才可申请资源
  - 缺点：资源利用率低、可能发生饥饿

死锁预防

- 无抢占
  - 如果一个进程持有资源并申请另一个不能立即分配的资源，那么它现在分配的资源都可被抢占
  - 这些资源都被隐式释放了，被抢占资源添加到进程等待的资源列表上
  - 只有当进程获得其原有资源和申请的新资源时，它才可以重新执行
  - 这个协议通常用于状态可以保存和恢复的资源，如CPU寄存器和内存。它一般不适用于其他资源

死锁预防

- 循环等待
  - 对所有资源类型进行完全排序，而且要求每个进程按递增顺序来申请资源
  - 要求对资源排序，开销
  - 常用于嵌入式操作系统，资源类型不多的情况
- 如何证明？反证法
  - 假设有一个循环等待，对于进程P0-Pn，其中Pi等待资源Ri，而Ri被进程Pi+1所占有；Pi+1等待资源Ri+1，而占有资源Ri
  - 可以得知：R0<R1<...<Ri<Ri+1<...Rn<R0
  - 由于R0不可能小于R0，显然不可能有循环等待

死锁避免

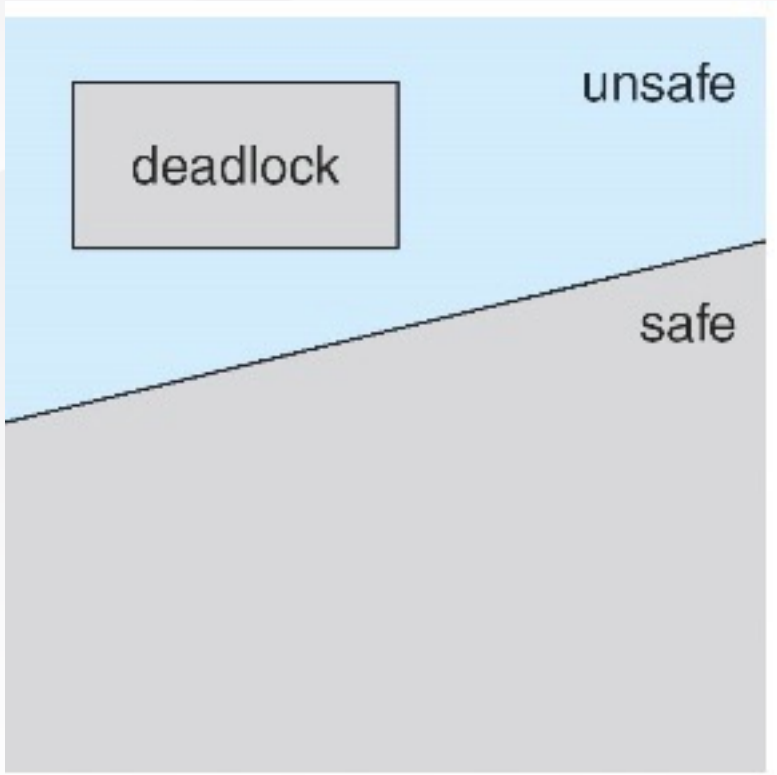
- 死锁预防的方法是通过限制资源申请的方法来预防死锁，导致设备利用率、系统吞吐量低
- 死锁避免：获得额外信息，包括进程需要申请的资源、顺序等，系统从而可以决定，在每次请求时进程是否应等待以避免可能的死锁
  - 每个进程应声明可能需要的每种资源的最大数量
  - 死锁避免算法动态检查资源分配状态，以确保循环等待条件不能成立
  - 资源分配状态包括可用的资源、已分配的资源及进程的最大需求

安全状态

- **如果系统能按一定顺序为每个进程分配资源（不超过它的最大需求），仍然避免死锁，那么系统的状态就是安全的**
- 只有存在一个安全序列<P1,P2,...,Pn>，系统才处于安全状态。对于每个进程Pi，Pi仍然可以申请的资源数小于当前可用资源+所有进程Pj（j<i）所占有的资源
  - 进程Pi所需要的资源即使不能立即可用，Pi可以等待，直到所有Pj释放资源
  - 当Pj完成，Pi可以得到所需要的资源，完成任务，最后终止
  - 当Pi终止时，Pi+1可得到它所需要的资源，如此进行

安全状态 非安全状态 死锁

- 安全状态不是死锁状态
- 非安全状态可能导致死锁
- 死锁状态是非安全状态
- **死锁避免算法：保证避免系统处于非安全状态**



安全状态 非安全状态

- 安全序列：？

	最大需求	当前持有	需要
P0	10	5	5
P1	4	2	2
P2	9	2	7

当前空闲
3

安全状态 非安全状态

- 安全序列：P1

	最大需求	当前持有	需要
P0	10	5	5
P1	4	4	0
P2	9	2	7

当前空闲
1

安全状态 非安全状态

- 安全序列：P1

	最大需求	当前持有	需要
P0	10	5	5
P1	4	---	---
P2	9	2	7

当前空闲
5

安全状态 非安全状态

- 安全序列：P1->P0

	最大需求	当前持有	需要
P0	10	10	0
P1	4	---	---
P2	9	2	7

当前空闲
0

安全状态 非安全状态

- 安全序列：P1->P0

	最大需求	当前持有	需要
P0	10	---	---
P1	4	---	---
P2	9	2	7

当前空闲
10

安全状态 非安全状态

- 安全序列：P1->P0->P2

	最大需求	当前持有	需要
P0	10	---	---
P1	4	---	---
P2	9	9	0

当前空闲
3

安全状态 非安全状态

- 安全序列：P1->P0->P2

	最大需求	当前持有	需要
P0	10	---	---
P1	4	---	---
P2	9	---	---

当前空闲
12

安全状态 非安全状态

- 安全序列：？

	最大需求	当前持有	需要
P0	10	5	5
P1	4	2	2
P2	9	3	6

当前空闲
2

安全状态 非安全状态

- 安全序列：P1

	最大需求	当前持有	需要
P0	10	5	5
P1	4	---	---
P2	9	3	6

当前空闲
4

安全状态 非安全状态

- 安全序列：P1->?

	最大需求	当前持有	需要
P0	10	5	5
P1	4	---	---
P2	9	3	6

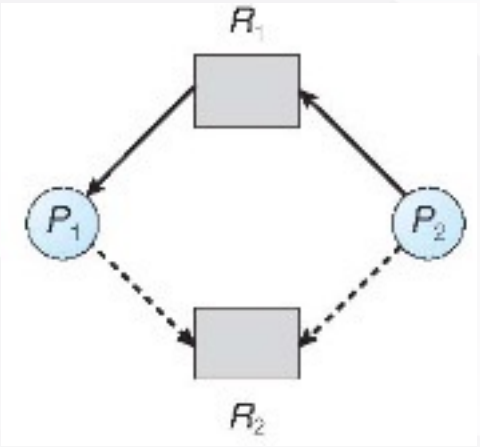
当前空闲
4

死锁避免算法

- 死锁避免算法的核心
  - 每当进程请求资源时，仅当分配使系统处于安全状态时，才会授予请求
- 两个算法
  - 资源分配图算法
    - 应用于每个资源只有一个实例
  - **银行家算法**
    - 应用于每个资源有多个实例

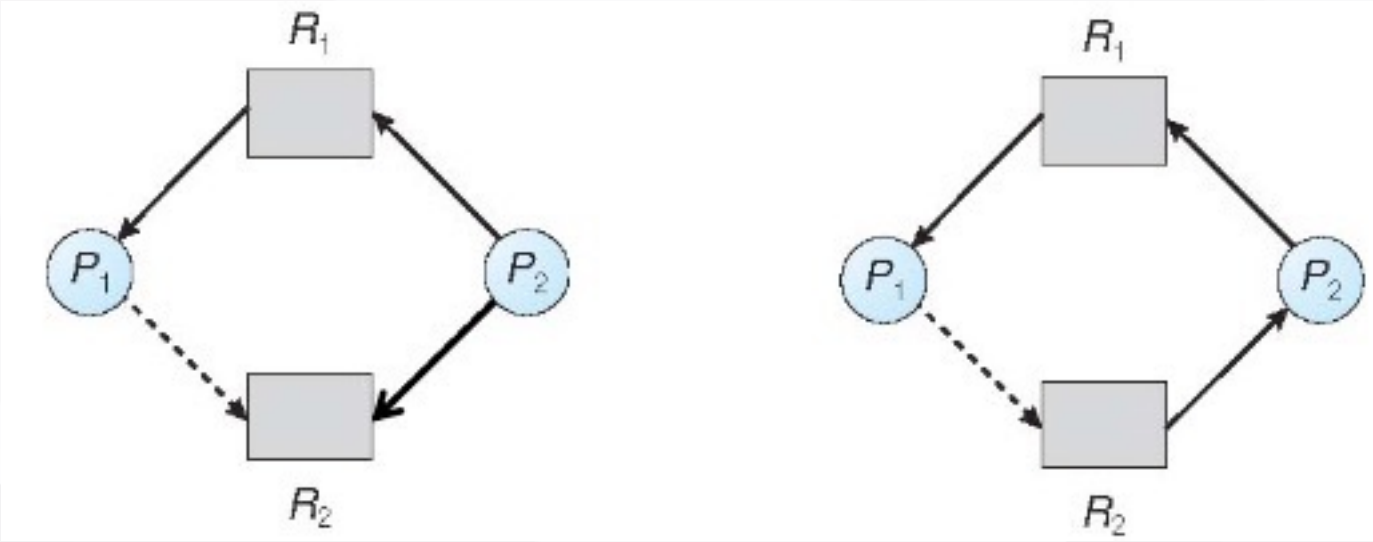
资源分配图算法

- 引入新的需求边 Pi->Rj，在图中用虚线表示，代表进程Pi可能在将来某时刻申请资源Rj
- 当进程Pi申请资源Rj时，需求边变成了申请边；当进程Pi释放Rj时，分配边变成了需求边
- 系统资源的需求应事先说明
  - 当进程Pi开始执行时，它所有需求边应处于资源分配图内



资源分配图算法

- 当进程Pi请求资源Rj
- 只有在申请边Pi->Rj变成分配边Rj->Pi并且不会导致资源分配图形成环时，才能允许申请



我们是否可以允许P2申请R2?

会成环，不允许，P2需要等待

银行家算法

- 前提条件
  - 适用于每个资源类型拥有多个实例的系统
  - 每个进程应声明可能需要的资源实例的最大数量
  - 当进程请求一个资源，就不得不等待
  - 当进程获得所有资源就必须在一段有限时间内释放它们
- 银行家算法试图寻找允许每个进程获得最大资源并结束的进程请求的一个理想执行序列，来决定状态是否安全
- 不存在这满足条件的执行序列的状态都是不安全的

银行家算法数据结构

- n=进程数量，m=资源类型数量
- Max（总需求量）：nxm矩阵。如果Max[i,j]=k，表示进程Pi最多需要资源类型Rj的k个实例
- Available（剩余空闲量）：长度为m的向量。如果Available[j]=k，有k个类型Rj的资源实例可用
- Allocation（已分配量）：nxm矩阵。如果Allocation[i,j]=k，则Pi当前分配了k个Rj的实例
- Need（未来需要量）：nxm矩阵。如果Need[i,j]=k，则Pi可能需要k个Rj的实例
- Need[i,j]=Max[i,j]-Allocation[i,j]

安全状态判断算法

1. Work和Finish分别是长度m和n的向量
   - 初始化：
     - Work=Available //当前资源剩余空闲量
     - Finish[i]=false for i=1,2,3…,n //线程i没结束
2. 找这样的i： //找出Need比Work小的进程i
   - Finish[i]=false
   - Need[i]<=Work
   - 没有找到这样的i，转到4

安全状态判断算法

3. 进程i结束，转到2
   - Work=Work+Allocation[i]　　（回收资源）
   - Finish[i]=true
4. 如果Finish[i]=true for all i，
   - 该系统是安全状态
   - 否则是非安全状态，进程i需要等待

银行家算法

- Request是进程Pi请求的资源量

1. 如果Request[i]<=Need[i]，转到步骤2；否则，提出错误条件
2. 如果Request[i]<=Available，转到步骤3；否则，进程Pi必须等待
3. 假装给Pi分配它需要的资源，并更新参数
    - Available= Available-Request[i]
    - Allocation[i]=Allocation[i]+Request[i]
    - Need[i]= Need[i]- Request[i]

- 调用安全算法
    - 如果返回safe，将资源分配给Pi；否则，Pi必须等待

银行家算法 实例

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）
- T0时刻：

	最大需求	当前持有	需要	当前空闲
	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	

系统是否处于安全状态?

使用安全算法

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	5 3 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	

安全序列：P1

使用安全算法

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	7 4 3
P2	9 0 2	3 0 2	6 0 0	
P4	4 3 3	0 0 2	4 3 1	

安全序列：P1->P3

使用安全算法

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
				7 5 3
P2	9 0 2	3 0 2	6 0 0	
P4	4 3 3	0 0 2	4 3 1	

安全序列：P1->P3->P0

使用安全算法

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
				10 5 5
P4	4 3 3	0 0 2	4 3 1	

安全序列：P1->P3->P0->P2

使用安全算法

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
				10 5 7

安全序列：P1->P3->P0->P2->P4

银行家算法 实例 P1请求（1 0 2）

- Original Available（3 3 2）
- 检查发现P1请求Request<=Available（1 0 2）< （3 3 2）
- 执行安全算法，找到安全序列< P1->P3->P0->P2->P4 >

银行家算法 实例 P1请求（1 0 2）

	最大需求	当前持有	需要	当前空闲
	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	

执行安全算法，找到安全序列< P1->P3->P0->P2->P4 >

	最大需求	当前持有	需要	当前空闲
	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	

银行家算法 实例 P0继续请求（0 2 0）

- 检查发现Request<=Available（0 2 0）< （2 3 0）

	最大需求	当前持有	需要	当前空闲
	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	

无法找到安全序列

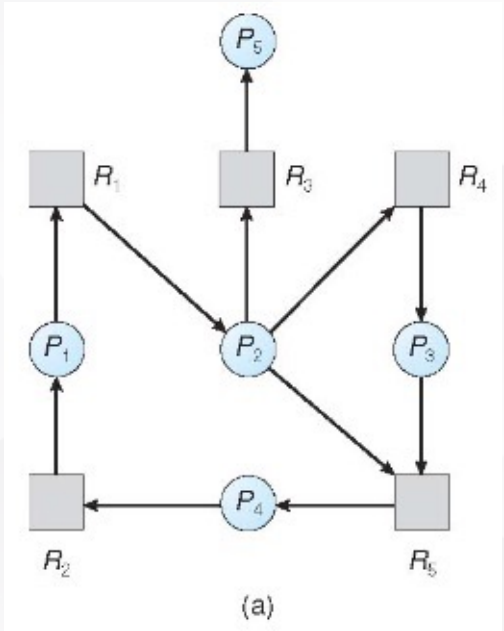
	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 3 0	7 2 3	2 1 0
P1	3 2 2	3 0 2	0 2 0	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

死锁检测

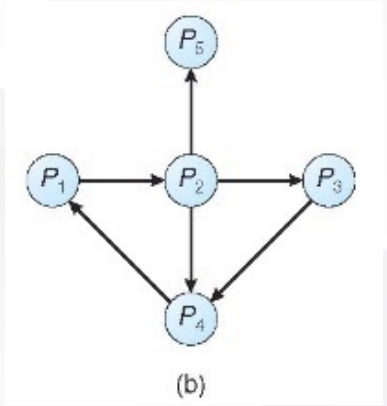
- 不使用死锁预防算法和死锁避免算法，系统可能出现死锁
- **允许系统出现死锁**
  - 死锁检测算法
  - 死锁恢复算法
- 两种情况
  - 单个实例的资源
  - 多个实例的资源

每种资源类型只有单个实例

- 将资源分配图简化，变成等待图
  - 将所有资源节点删除，合并适当边，则可以得到等待图
  - Pi->Pj，表示Pi在等待Pj释放一个Pi所需要的资源
- 周期性地调用算法，搜索等待图中是否存在环，当且仅当在等待图中出现环，系统死锁



资源分配图



对应的等待图

每种资源类型有多个实例

- 类似于安全状态判断算法
- 数据结构
  - Available：长度为m的向量，表示各种资源的可用实例数量
  - Allocation：nxm矩阵，表示每个进程的每种资源的当前分配数量
  - Request： nxm矩阵，表示每个进程的每种资源的当前请求

死锁检测算法

1. Work和Finish分别是长度m和n的向量
   - 初始化：
     - Work=Available //当前资源剩余空闲量
     - 对于i=1,2,3…,n，如果Allocation[i]不为0，则Finish[i]=false；否则Finish[i]=true
2. 找这样的i：
   - Finish[i]=false
   - Request[i]<=Work
   - 没有找到这样的i，转到4

死锁检测算法

3. 进程i结束，转到2
   - Work=Work+Allocation[i]
   - Finish[i]=true
4. 如果对某个进程i， Finish[i]=false，则系统死锁

算法复杂度：$O(m*n^2)$
开销大！
用于系统调试

死锁检测算法 实例1

- 5个进程：P0-P4
- 3个资源类型：A（7个实例），B（2个实例），C（6个实例）
- T0时刻：

	当前持有 (Allocation)	需要 (Request)	当前空闲 (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	

序列< P0, P2, P3, P1, P4 >可以使系统处于安全状态

死锁检测算法 实例2

- 5个进程：P0-P4
- 3个资源类型：A（7个实例），B（2个实例），C（6个实例）
- P2需要一个资源C的实例

	当前持有 (Allocation)	需要 (Request)	当前空闲 (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 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

无法保持安全状态，存在死锁，包括P1、P2、P3、P4

死锁检测算法使用

- 何时、使用什么样的频率来检测依赖于：
  - 死锁多久可能会发生？
  - 多少进程需要被回滚？
- 如果在任一时间点调用检测算法，那么资源图可能有多个环。通常不能确定哪个进程“造成”了死锁
- 死锁检测的计算开销相当大，一般用于开发阶段，判断系统是否正确

死锁恢复算法

- 进程终止
  - 终止所有死锁进程
  - 一次终止一个进程，直到消除死锁循环为止
- 资源抢占：通过资源抢占来消除死锁，我们不断地抢占一些进程的资源以便给其他进程使用，直到死锁循环被打破为止

进程终止

- 终止所有死锁进程：代价大，计算结果都要放弃，数据重新加载
- 一次终止一个进程，直到消除死锁循环为止：代价大，每次终止一个进程，就要调用死锁检测算法，以确定是否仍有进程处于死锁；确定哪个进程该被终止也很难，需要考虑多种因素：
  - 进程的优先级是什么?
  - 进程已计算了多久？在完成指定任务之前还要计算多久?
  - 进程使用了多少数量的何种类型的资源(例如，这些资源是否容易抢占)？
  - 进程需要多少资源才能完成?
  - 需要终止多少进程?
  - 进程是交互的还是批处理的

资源抢占

- 用抢占来处理死锁，需要解决三个问题
  - 选择牺牲进程：抢占哪些资源和哪些进程？
    - 要考虑代价，确定抢占顺序，使代价最小
  - 回滚：如果从一个进程那里抢占了一个资源，需要将其回滚。回滚到什么状态？完全回滚是最简单的，但开销最大；或回滚到足够打破死锁，但这要求系统维护相关运行进程的更多状态信息
  - 饥饿：如何确保不会发生饥饿？即如何保证资源不会总是从同一个进程中被抢占。如果一个系统是基于代价来选择牺牲进程，那么同一进程可能总是被选为牺牲的。应确保一个进程只能有限次数地被选为牺牲进程