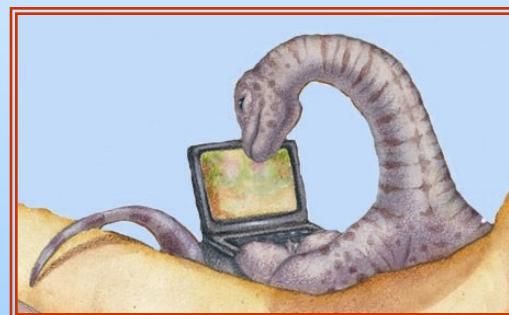


第7章 死锁





内容

1. 死锁概念
2. 死锁预防
3. 死锁避免
4. 死锁检测和恢复



1、死锁概念



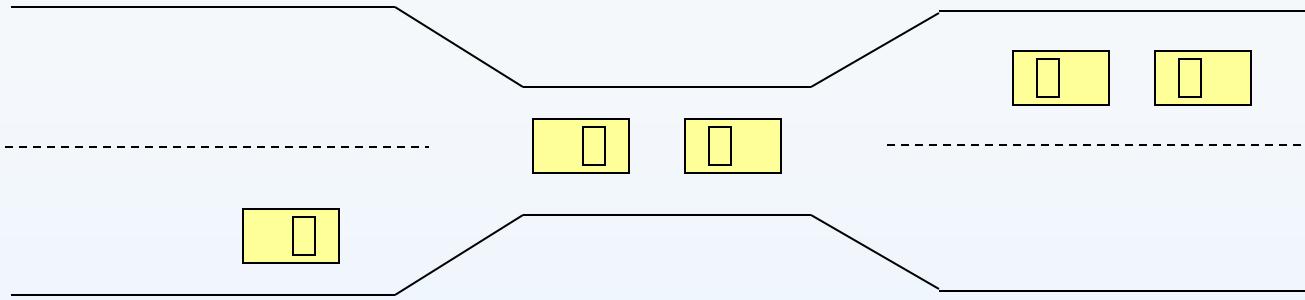
死锁概念
资源分配图
死锁处理方法







过桥例子



- 只能一个方向通行
- 桥的每一个部分都可以看成资源
- 如果死锁发生，它可以由一辆车返回而解决，抢占资源并回退
- 如果死锁发生，可能很多车都不得不返回
- 有可能产生饥饿





死锁问题

- 在多道程序环境下，一组处于等待状态的进程，其中每一个进程都持有资源，并且等待着由这个组中其他进程所持有的资源，那么该组等待进程有可能再也无法改变其状态，这种情况称为死锁。
- 所有死锁进程如无外力的介入，都无法往前推进
- 引起死锁的主要原因
 - 竞争互斥资源
 - 进程推进不当
- 例如
 - 系统有两个磁带设备
 - 进程P1和P2各占有一个磁带设备并且实际需要两个磁带
 - 此时P1和P2进程均死锁



互斥资源





死锁例子

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```





死锁的特性

- **互斥**: 至少有一个资源必须处于非共享模式，即一次只有一个进程可以使用
 - **占有并等待**: 一个至少持有一个资源的进程等待获得额外的由其他进程所持有的资源
 - **非抢占**: 资源不能被抢占，即一个资源只有当持有它的进程完成任务后自动释放
 - **循环等待**: 一组等待资源的进程 $\{P_0, P_1, \dots, P_n\}$ 之间存在循环等待现象，即 P_0 等待 P_1 占有的资源， P_1 等待 P_2 占有的资源， \dots ， P_{n-1} 等待 P_n 占有的资源， P_n 等待 P_0 占有的资源
- 四个条件同时出现，死锁将会发生（必要条件）





系统模型

- 资源类型 CPU周期, 内存空间, 文件, I/O设备
 - 形式化表示为: R_1, R_2, \dots, R_m
 - 每一种资源 R_i 有 W_i 种实例
-
- 每一个进程通过如下顺序来使用资源
 - 申请资源
 - ▶ 动态申请资源: 在进程运行过程中申请资源 (常用方法)
 - ▶ 静态申请资源: 在进程运行前一次申请所有资源
 - 使用资源
 - 释放资源





资源分配图

一个顶点的集合**V**和边的集合**E**

■ V 被分为两个部分

- $P = \{P_1, P_2, \dots, P_n\}$, 含有系统中全部的进程

- $R = \{R_1, R_2, \dots, R_m\}$, 含有系统中全部的资源

■ 申请边：有向边 $P_i \rightarrow R_j$, 表示进程 P_i 申请了资源 R_j 的一个实例

■ 分配边：有向边 $R_j \rightarrow P_i$, 表示资源 R_j 的一个实例分配给进程 P_i





资源分配图

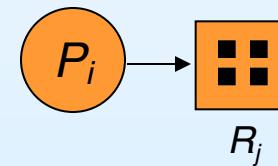
- 进程



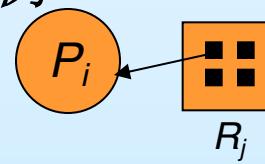
- 有四个实例的资源类型



- P_i 请求一个 R_j 的实例

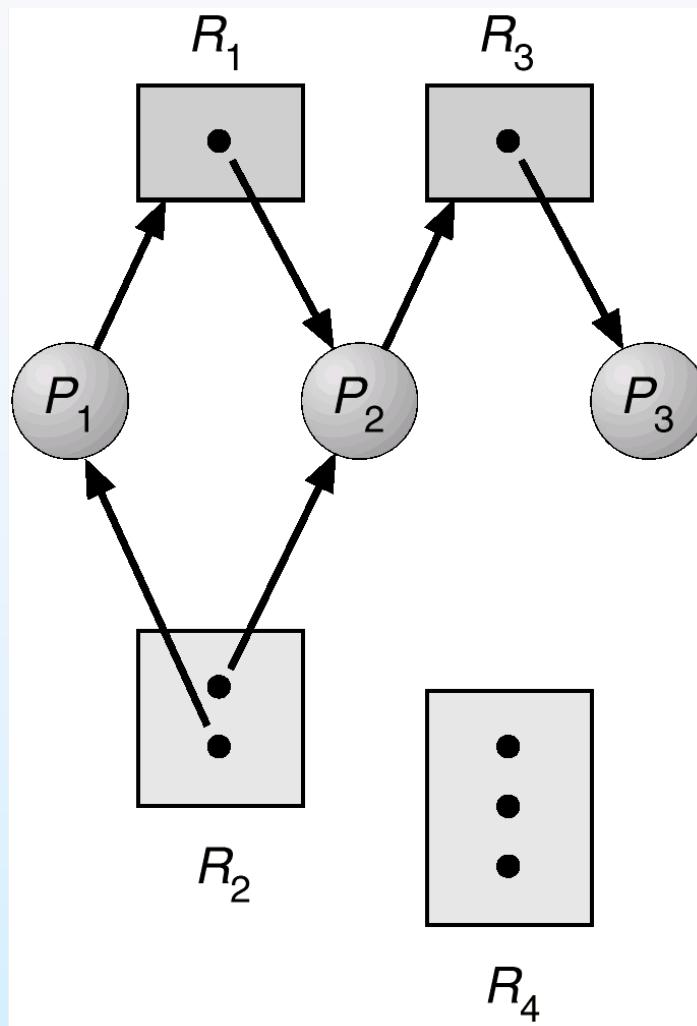


- P_i 持有一个 R_j 的实例





资源分配图的例子



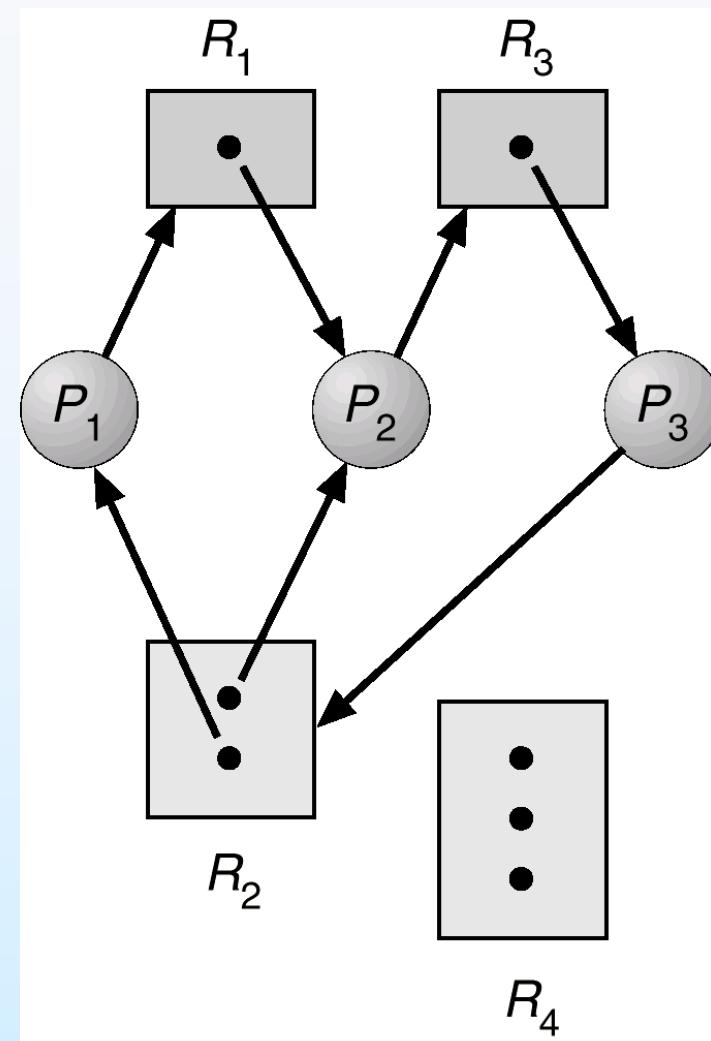
如果分配图没有环，那么系统就没有进程死锁；

如果分配图有环，那么可能存在死锁。



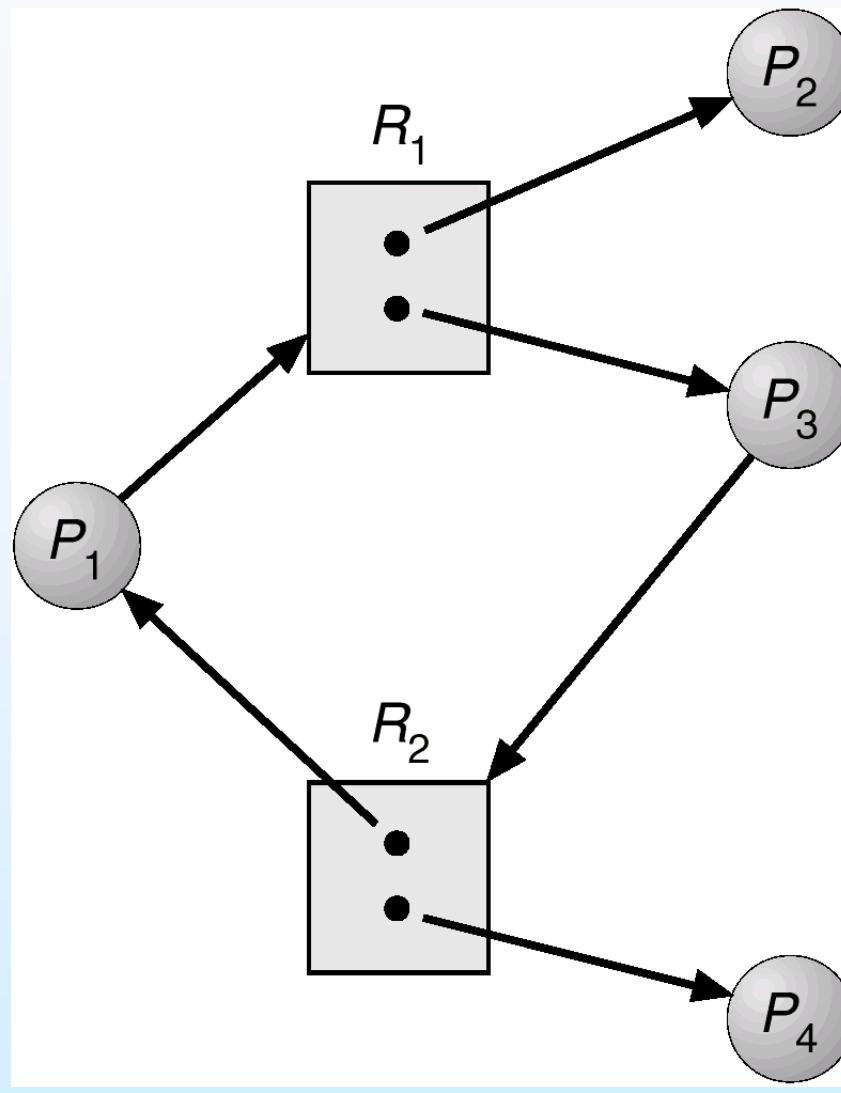


有环有死锁的资源分配图





有环但没有死锁的资源分配图





结论

- 如果资源分配图没有环，那么系统就不处于死锁状态
- 如果图有环，那么系统可能处于死锁状态⇒
 - 如果每一种资源类型只有一个实例，那么死锁一定发生
 - 如果一种资源类型有多个实例，则可能死锁



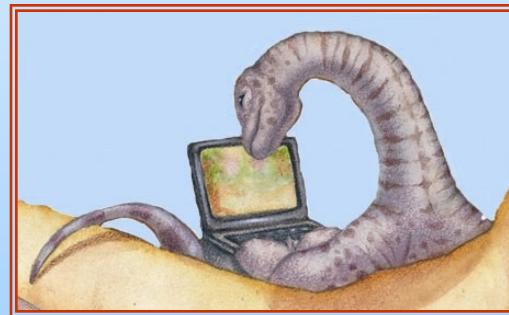


处理死锁的方法

1. 使用协议来预防或避免死锁，确保系统永远不会进入死锁状态
 - 死锁预防
 - 死锁避免
2. 允许系统进入死锁状态，然后检测它，并加以恢复系统
 - 死锁检测
 - 死锁恢复
3. 忽略这个问题，假装系统中从未出现过死锁。
 - 这个方法被大部分的操作系统采用，包括**UNIX**、**Window**
 - 需要应用程序的开发人员自己来处理死锁
 - 虚拟化技术，虚拟为共享设备



2、死锁预防





死锁的预防

- 死锁预防，Deadlock prevention，是一组方法，只要确保至少一个必要条件不成立，就能预防死锁。
- 互斥：
 - 非共享资源（互斥资源），必须要有互斥条件
 - 共享资源，不涉及死锁
 - 现代操作系统中的虚拟化技术，将互斥资源改造为共享资源
 - 如果系统中存在互斥资源，不能改变这个条件来预防死锁





死锁的预防

- 占有并等待：必须保证进程申请资源的时候没有占有其他资源
 - 静态分配策略
 - 协议1：要求进程在执行前一次性申请全部的资源
 - 协议2：进程只有没有占有资源时才可以分配资源
 - 利用率低，可能出现饥饿





死锁的预防

■ 非抢占：

- 如果一个进程的申请没有实现，它要释放所有占有的资源（被抢占）
- 被抢占（先占）的资源放入进程等待资源列表中
- 进程在重新得到旧的资源的时候可以重新开始





死锁的预防

■ 循环等待：

- 对所有的资源类型排序进行总排序，并且要求进程按照递增顺序申请资源
 - 设 $R=\{R_1, R_2, \dots, R_3\}$ 为资源类型的集合，为每个资源类型分配唯一整数来允许比较这两个资源，以确定其顺序。
 - 为预防死锁，每个进程只按递增顺序申请资源，即一个进程开始可以申请任何类型资源 R_i 的实例，
 - 当且仅当 R_j 的值大于 R_i 的值时，进程才可以申请 R_j 的实例。
- 设计一个完全排序方法并不能防止死锁，而要靠应用程序员来按照顺序编写程序
- 各资源类型的序号确定也是至关重要的，要按照系统内部资源使用的正常顺序定义。

资源类型 R 的集合	数值
磁带驱动器	1
磁盘驱动器	5
打印机	12



3、死锁避免





死锁预防算法

- 操作 通过限制资源申请的方法来预防死锁
- 副作用 降低了设备使用率和系统吞吐量

- 避免死锁的另一个方法：获得以后如何申请资源的附加信息来决定是否分配资源
- 每次申请要求系统考虑现有可用资源、现已分配给每个进程的资源和每个进程将来申请与释放的资源，以决定申请是否满足，从而避免死锁发生的可能性。





死锁避免

需要系统有一些额外的信息

- 一个简单而有效的模型要求每一个进程声明它所需要的资源的最大数
- 死锁避免算法动态检查资源分配状态以确保不会出现循环等待的情况
- 资源分配状态是由 可用的与已分配的资源数，和 进程所需的最大资源量 决定的。





安全状态

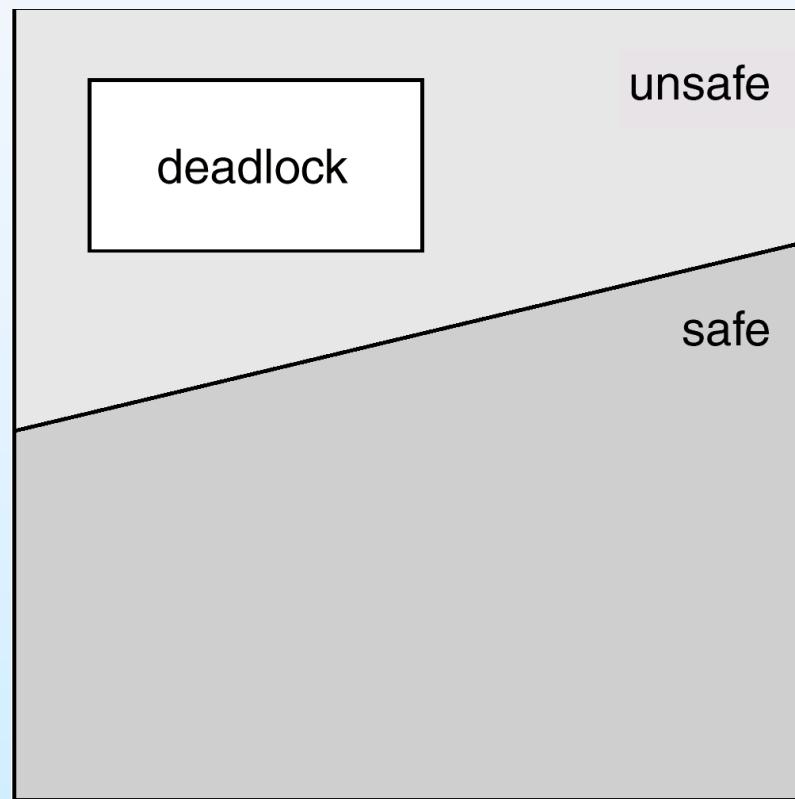
- 如果系统能按照某个顺序为每个进程分配资源并能避免死锁，那么系统状态是安全的。
- 如果存在一个安全序列，系统处于安全状态，否则系统处于不安全状态。
- 进程序列 $\langle P_1, P_2, \dots, P_n \rangle$ 是安全的，如果每一个进程 P_i 所申请的可以被满足的资源数加上其他进程所持有的资源数小于系统总数
 - 如果 P_i 需要的资源不能马上获得，那么 P_i 等待直到所有的 P_{i-1} 进程结束。
 - 当 P_{i-1} 结束后， P_i 获得所需的资源，执行、返回资源、结束。
 - 当 P_i 结束后， P_{i+1} 获得所需的资源执行，依此类推。





基本事实

- 如果一个系统在安全状态，就没有死锁
- 如果一个系统不是处于安全状态，就有可能死锁
- 避免 \Rightarrow 确保系统永远不会进入死锁状态





避免算法

- 单实例资源
 - 资源分配图法

- 多实例资源
 - 银行家算法





资源配置图边转换

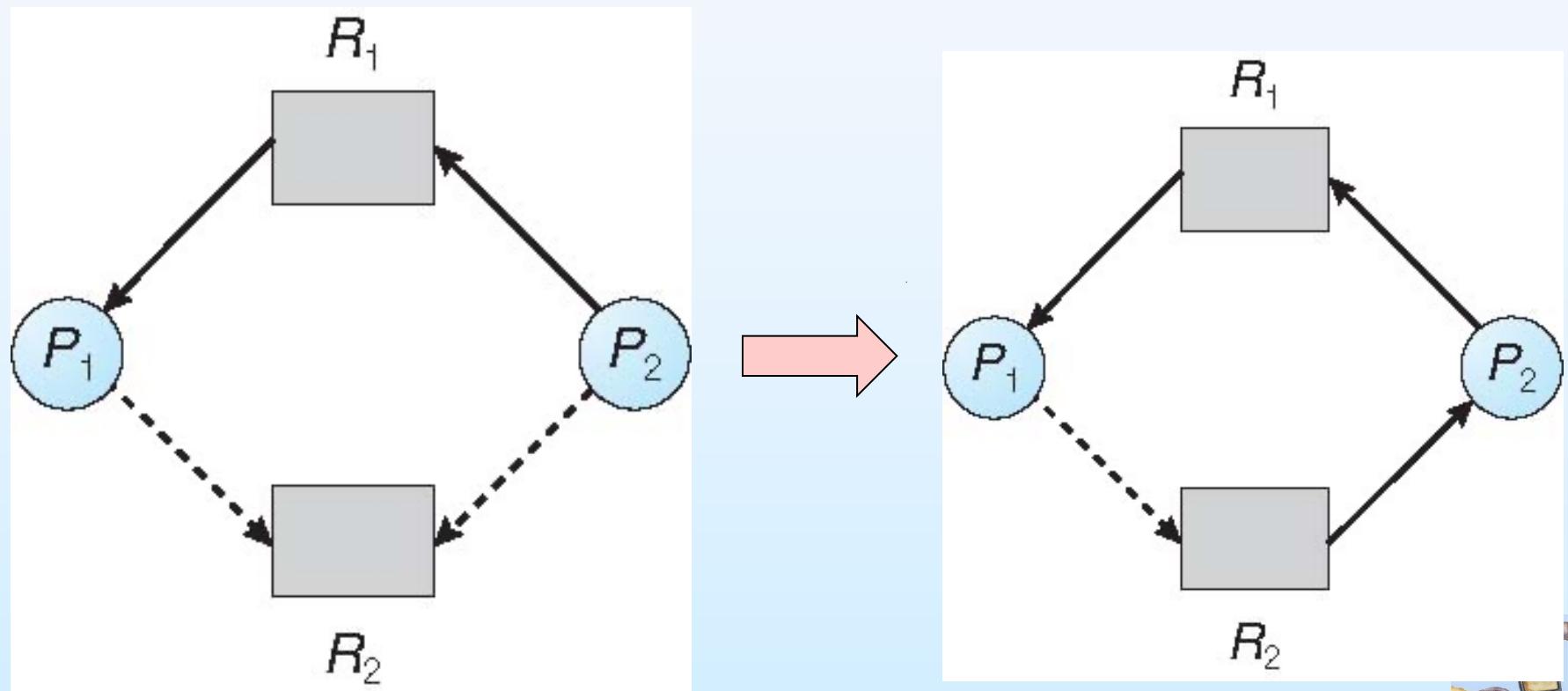
- 需求边 $P_i \rightarrow R_j$: P_i 可能以后需要申请 R_j 资源，用虚线表示
- P_i 申请 R_j 资源，需求边转换为请求边
- 请求边在资源分配后转换为分配边
- 资源释放后，分配边转换为需求边





资源分配图算法

- 假设 P_i 申请资源 R_j
- 请求能满足的前提是：把请求边转换为分配边后不会导致环存在





银行家算法

- 多个实例
- 每一个进程必须事先声明使用各类资源的最大量，不能超过系统资源的总和
- 当一个进程请求资源，它可能要等待，系统需要确定资源分配是否会使系统处于安全状态
- 当一个进程得到所有的资源，它必须在有限的时间释放它们





银行家算法的数据结构

- *Available*: 长度为 m 的向量。如果 $\text{available}[j]=k$, 那么资源 R_j 有 k 个实例有效
- *Max*: $n \times m$ 矩阵。如果 $\text{Max}[i,j]=k$, 那么进程 P_i 可以最多请求资源 R_j 的 k 个实例
- *Allocation*: $n \times m$ 矩阵。如果 $\text{Allocation}[i,j]=k$, 那么进程 P_j 当前分配了 k 个资源 R_j 的实例
- *Need*: $n \times m$ 矩阵。如果 $\text{Need}[i,j]=k$, 那么进程 P_j 还需要 k 个资源 R_j 的实例

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j].$$

n 为进程的数目, m 为资源类型的数目





安全算法

1. 让Work和Finish作为长度为m和n的向量初始化:

$Work := Available$

$Finish[i] = false \text{ for } i - 1, 3, \dots, n.$

2. 查找i

(a) $Finish[i] = false$

(b) $Need_i \leq Work$

If no such i exists, go to step 4.

3. $Work := Work + Allocation_i$

$Finish[i] := true$

go to step 2.

4. 如果对所有*i*的 $Finish[i] = true$, 则系统处在安全状态。





是否满足进程 P_i 的资源请求算法

$Request_i$ = 进程 P_i 的资源请求向量. 如果 $Request_i[m] = k$ 则进程 P_i 想要资源类型为 R_m 的 k 个实例

算法如下:

1. 如果 $Request_i \leq Need_i$, 转 step 2. 否则报错, 因为进程请求超出了其声明的最大值
2. 如果 $Request_i \leq Available$, 转 step 3. 否则 P_i 必须等待, 因为资源不可用.
3. 假设通过修改下列状态来分配请求的资源给进程 P_i :
 $Available := Available - Request_i;$
 $Allocation_i := Allocation_i + Request_i;$
 $Need_i := Need_i - Request_i;$

- 如果系统安全 \Rightarrow 将资源分配给 P_i .
- 如果系统不安全 $\Rightarrow P_i$ 必须等待, 恢复原有的资源分配状态





银行家算法的例子

- 5个进程 P_0 到 P_4 ; 3个资源类型A(10个实例) , B (5个实例) , C (7个实例)
- 时刻 T_n 的快照:

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

系统处在安全的状态，因为序列 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 满足了安全的标准





例子续: P_1 request (1,0,2)

- 检查 $\text{Request} \leq \text{Available}$ (即, $(1,0,2) \leq (3,3,2)$ 为真)

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- 执行安全算法表明序列 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 满足要求
- 思考:
 - P_4 的请求(3,3,0)是否可以通过?
 - P_0 的请求(0,2,0)是否可以通过?





银行家算法

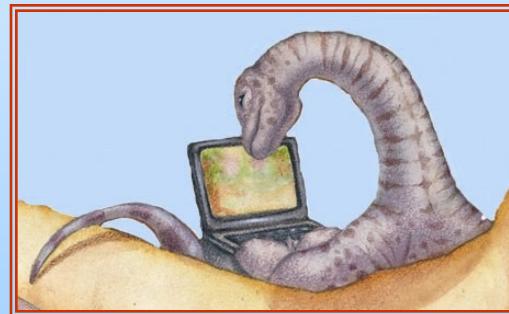
- 5个进程 P_0 到 P_4 ; 3个资源类型A(10个实例) , B (5个实例) , C (7个实例)
- 时刻 T_n 的快照: 系统是否是安全的? 如果是安全的, 请写出安全序列。

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
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

- P_1 的请求(1,0,2)是否可以通过?
- P_4 的请求(3,3,0)是否可以通过?
- P_0 的请求(0,2,0)是否可以通过?



4、死锁检测和恢复





死锁检测和恢复

- 允许进入死锁状态
- 检测死锁
- 恢复策略





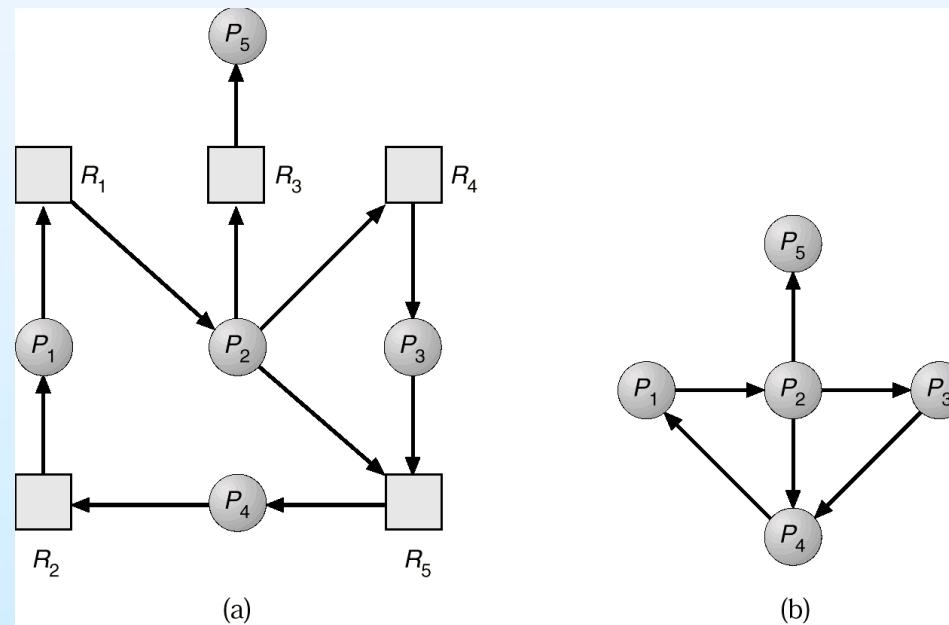
每一种资源类型只有一个实例

■ 维护等待图

- 节点是进程
- $P_i \rightarrow P_j$ 表明 P_i 在等待 P_j 所持有的资源

■ 定期调用算法来检查是否有环

- 一个检查图中是否有环的算法需要 n^2 的操作来进行， n 为图中的节点数，即进程数





一个资源类型的多个实例

- *Available* : 一个向量的长度 m 代表每一种资源类型有效的数目
- *Allocation*: 一个 $n \times m$ 的矩阵定义了当前分配的每一种资源类型的实例数目
- *Request*: 一个 $n \times m$ 的矩阵表明了当前的进程请求。如果 $\text{Request}[i,j]=k$, 那么进程 P_i 请求 k 个资源 R_j 的实例





检测算法

1. 让Work和Finish作为长度为m和n的向量初始化
 - (a) $Work := Available$
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] := \text{false}$; otherwise, $Finish[i] := \text{true}$.
2. 找到满足下列条件的下标i
 - (a) $Finish[i] = \text{false}$
 - (b) $Request_i \leq Work$
如果没有这样的i存在, 转4
3. $Work := Work + Allocation_i$
 $Finish[i] := \text{true}$
转 2.
4. 如果有一些 i , $1 \leq i \leq n$, $Finish[i] = \text{false}$, 则系统处在死锁状态。而且, 如果 $Finish[i] = \text{false}$, 则进程 P_i 是死锁的。

算法需要 $m \times n^2$ 次操作来判断是否系统处于死锁状态





检测算法的例子

- 五个进程 P_0 到 P_4 ,三个资源类型A (7个实例) , B (2个实例) ,C (6个实例)
- 时刻 T_n 的状态

	<i>Allocation</i>			<i>Request</i>			<i>Available</i>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
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			

- 对所有 i , 序列 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ 将导致 $Finish[i] = \text{true}$ 。





例子（续）

- P2请求一个额外的C实例

Request

	A	B	C
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- 系统的状态？

- 可以归还 P_0 所有的资源，但是资源不够完成其他进程的请求
- 死锁存在，包括进程 P_1, P_2, P_3 和 P_4





检测算法的用法

- 何时及多长时间的调用取决于
 - 死锁可能发生的频度
 - 有多少进程可能需要被回滚
 - ▶ 每一个独立的环需要一个
- 每个请求都调用
 - 极端情况下，每次请求不能立即满足时，调用死锁检测算法
- 不太高的频率
 - 每小时一次，或者CPU使用率低于40%时





从死锁中恢复

■ 发现死锁后的措施

- 人工处理
- 自动恢复

■ 自动恢复

- 进程终止
- 抢占资源





从死锁中恢复：进程终止

- 中断所有的死锁进程
- 一次中断一个进程直到死锁环消失
- 应该选择怎样的中断顺序？
 - 进程的优先级
 - 进程需要计算多长时间，以及需要多长时间结束
 - 进程使用的资源
 - 进程完成还需要多少资源
 - 多少个进程需要被中断
 - 进程是交互的还是批处理





从死锁中恢复：抢占资源

- 把抢占的资源分配给其他进程使用，直到死锁循环被打破为止
 1. 选择一个牺牲品：最小化代价
 - ▶ 包含所拥有的资源数、运行到现在的时间等。
 2. 回退：返回到安全的状态，然后重新开始进程
 3. 饥饿：同样进程的可能总是被选中
 - ▶ 在代价因素中加入回滚次数





思考

1. 为什么系统处于不安全状态，但不一定发生死锁？
2. 死锁预防和死锁避免哪一个系统开销大？
3. 死锁的进程中，有一个满足需求执行完成且释放了所有资源，剩下的进程一定能解除死锁？
4. 资源分配图中有环就一定死锁吗？
5. 资源实例数目大于进程所需的资源数目，一定不会发生死锁？

