第二部分：进程管理

第七章: 死 锁

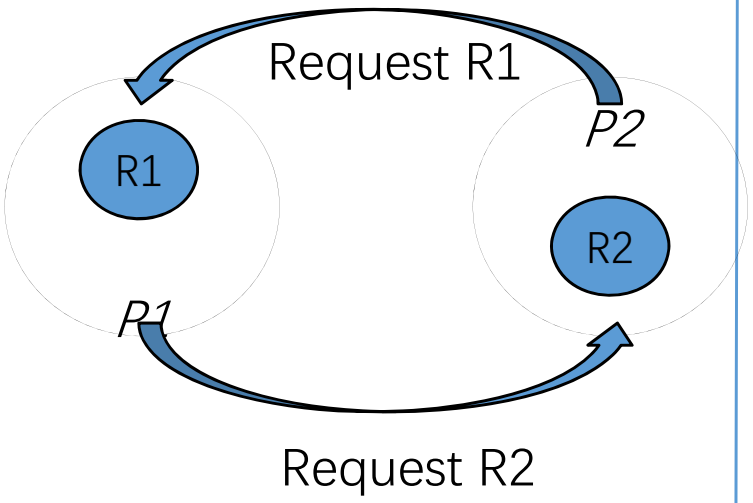
1. 分析死锁问题
2. 介绍预防和避免死锁的方法
3. 死锁检测方法
4. 死锁恢复方法

死锁问题

什么时候可能会发生死锁?

1. 进程占用一个资源
2. 并请求被其他进程所占用的资源

- 系统有 R_1 和 R_2 资源
- 进程 P_1 占用 R_1 并请求使用R_2，P_2 占用 R_2 并请求使用R_1
- 两个进程都无法往下运行



1. 系统模式

- 资源分类，实例的概念
  - 资源类型 $R_1, R_2, \ldots, R_m$, 如 CPU, 内存空间, I/O 设备，文件
  - 每个资源类型R_i 有 W_i 个实例
- 进程按如下顺序使用资源:

- 当一组进程中的每个进程都在等待一个事件的发生，而这一事件只能由这一组进程的另一进程引起，那么这组进程就处于死锁状态

2. 死锁特征-必要条件

1.互斥：一个资源只能由一个进程占用

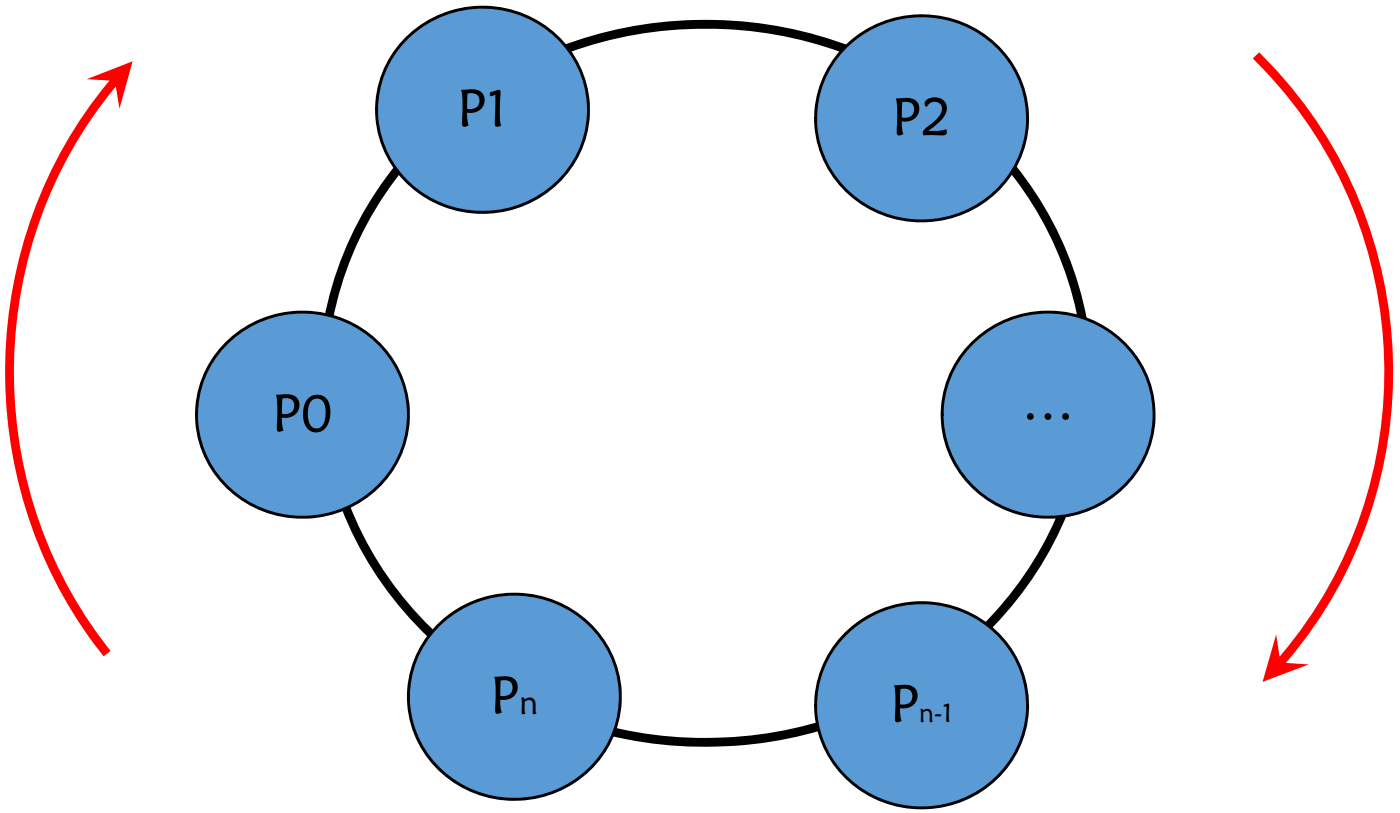
2.占用并等待：一个进程必须占用一个资源，并请求/等待另一个资源

3.非抢占：资源不能被抢占

4.循环等待：有一组等待进程{$P_0, P_1, \cdots, P_n$}，P_0等待的资源被P_1所占用，P_1等待的资源被P_2所占用，…，P_{n-1}等待的资源被P_n所占用，P_n等待的资源被P_0所占用

当以上四个条件都满足时会发生死锁

死锁特征



资源分配图

资源分配图由节点集合 V 和一个边集合 E 组成

1. 节点集合V 分为（1）进程集合P 和（2）资源集合 R

- 进程节点集合： $P = \{P_1, P_2, \cdots, P_n\}$
- 资源节点集合： $R = \{R_1, R_2, \cdots, R_m\}$

2. 边集合E

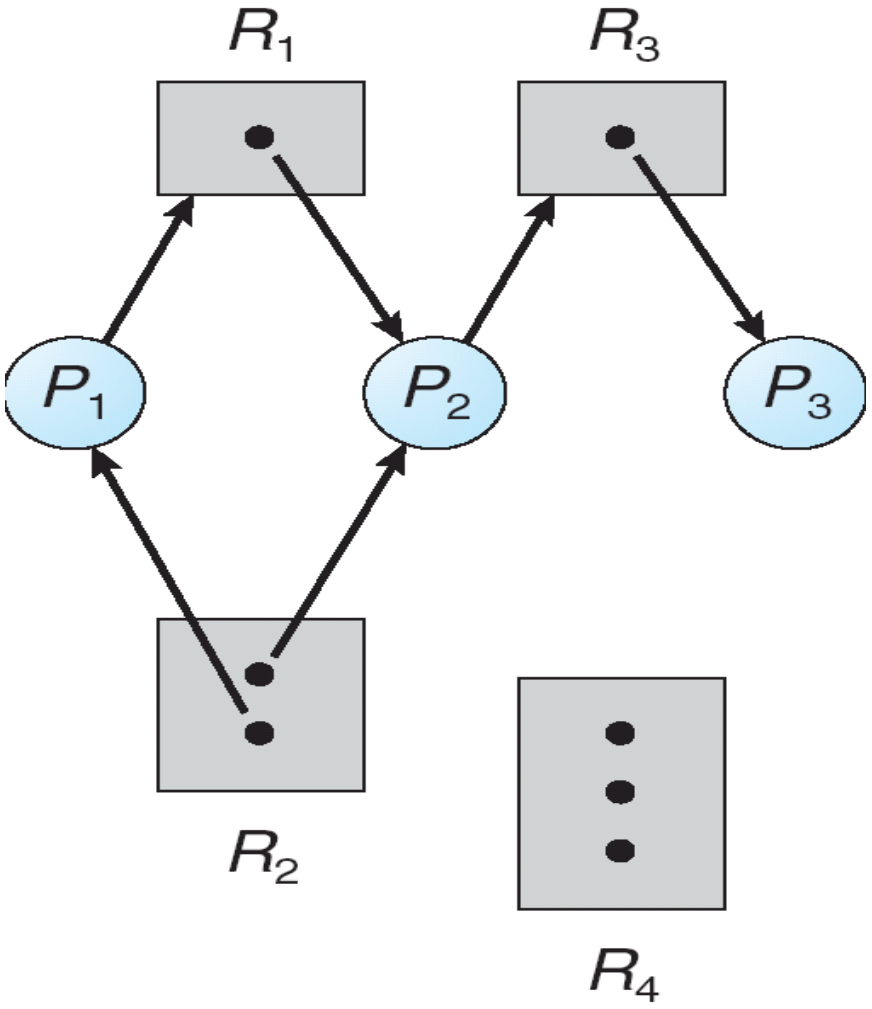
- $P_i \rightarrow R_j$

： 表示进程 Pi已经申请使用资源类型 R_j 的一个实例

- $R_j \rightarrow P_i$

： 表示资源类型 R_j 的一个实例已经分配给进程P_i

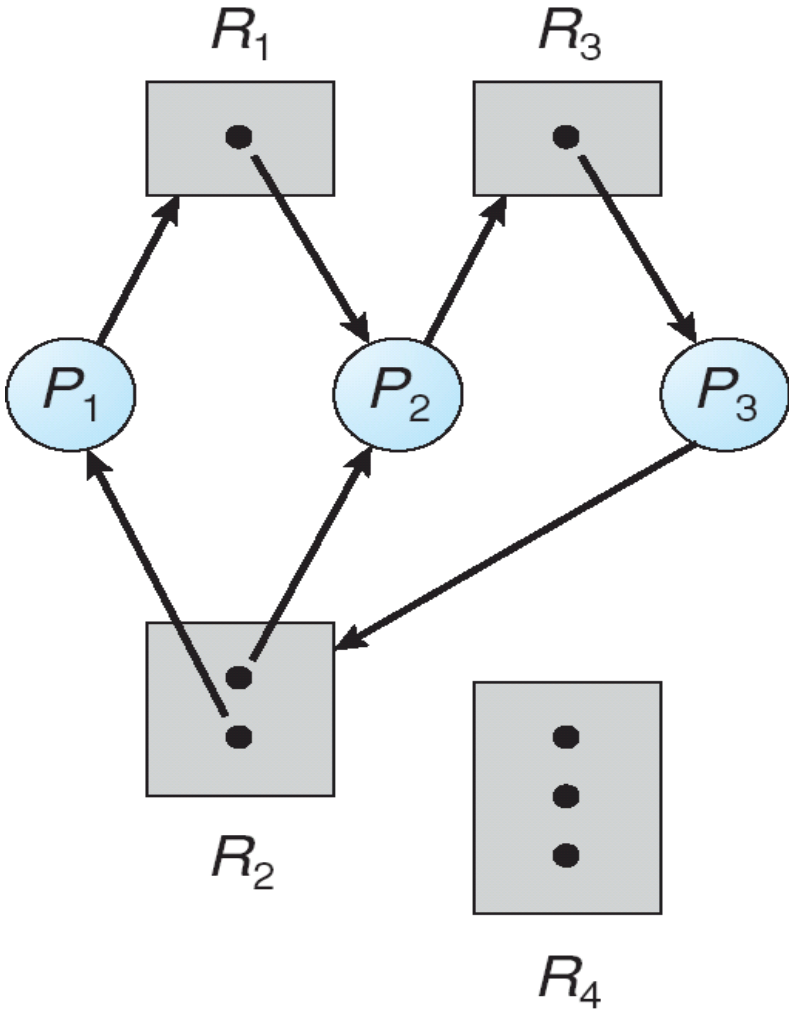
资源分配图例子



- R1→P2
- R2→P1, R2→P2
- R3→P3

- P1→R1
- P2→R3

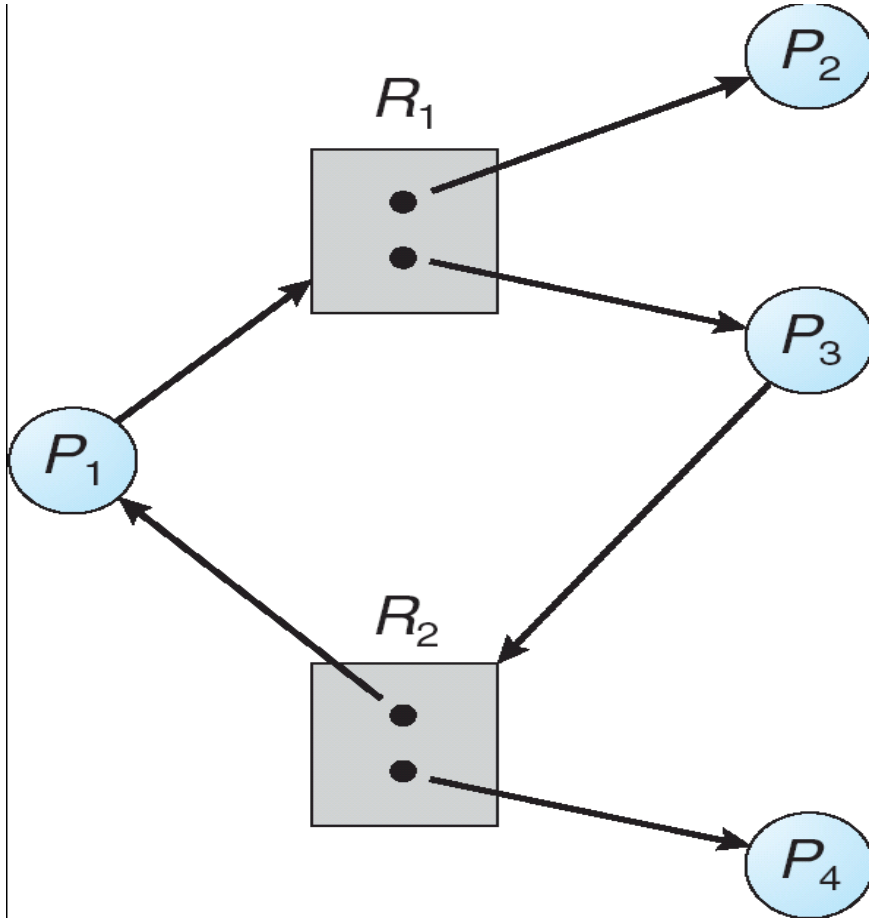
存在死锁的资源分配图



由于P3申请资源R2，导致资源的请求形成了环。

形成环就会发生死锁吗?

存在环，但是没有死锁的资源分配图



P2或P4释放资源，可以打破环

得出的结论

1. 如果分配图没有环，就没有死锁
2. 如果分配图有环，就有可能发生死锁

2.1：如果每个资源类型只有一个实例，就肯定会发生死锁

2.2：如果每个资源类型有多个实例，就有可能处于死锁

3. 死锁处理方法

1. 预防或避免（prevention or avoidance）
   - 预防死锁：确保至少一个必要条件不成立
   - 避免死锁：利用事先得到进程申请资源和使用资源的额外信息，判断每当发生资源请求时是否会发生死锁
2. 发生死锁，检测并恢复
   - 确定死锁是否确实发生， 并提供算法从死锁中恢复
3. 忽视死锁问题

3.1.1 预防死锁

(1) 互斥

- 不可共享（如打印机）的资源必须要确保互斥
- 可共享的资源（如只读文件）不要求互斥访问

通常不能通过否定互斥条件来预防死锁

(2) 占有并等待

- 确保当一个进程请求一个资源时，它不能占有其他资源。实现的方法如下

A. 每个进程在执行前申请并获得所有资源

B. 进程只有在不占用资源时，允许进程申请资源

3.1.1 预防死锁

举例：

将数据从DVD驱动器复制到磁盘文件，并对磁盘文件进行排序，再将结果打印到打印机上

A. 一开始就同时申请资源DVD驱动器、磁盘文件、打印机

B. 一开始只申请DVD驱动器和磁盘文件，复制结束后释放资源；释放资源后，在申请磁盘文件和打印机

让互斥和占用并等待不成立、两种方法的缺点是资源利用率低和可能发生饥饿

3.1.1 预防死锁

(3) 非抢占

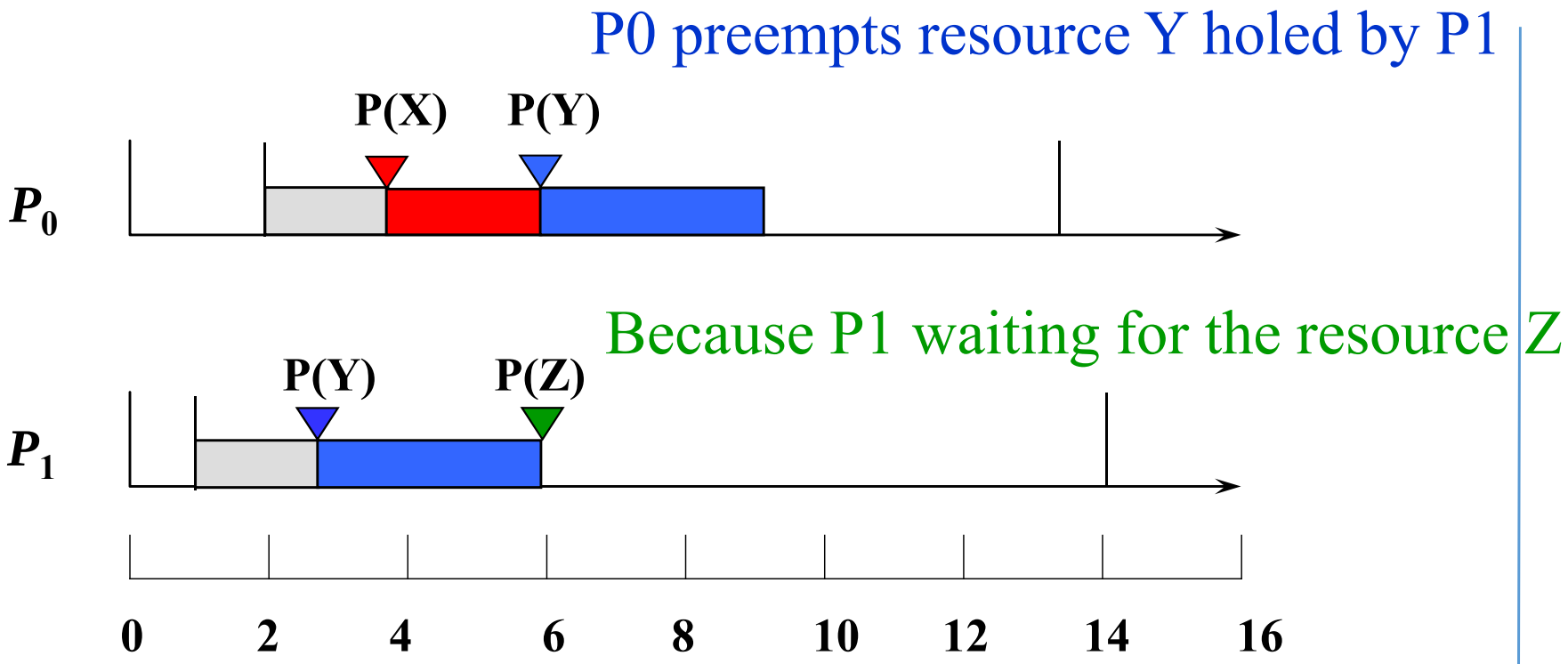
为了确保这一条件不成立，可以是使用如下协议：
如果一个进程占有资源并申请另一个不能分配的资源，那么其现已分配的资源可被抢占

如果一个进程申请一些资源，那么首先检查它们是否可用

1. 如果可用，就分配
2. 如果不可用，检查这些资源是否已分配给其他等待额外资源的进程，如果是，那么就抢占；
3. 如果不可用，且也没有被其他等待进程占有，那么就等待

3.1.1 预防死锁：非抢占

- 有两个进程 P0 and P1 ，有三个资源 X，Y，Z
- P1占有资源Y，等待资源Z
- P0占有资源X，申请资源Y



3.1.1 预防死锁

(4) 循环等待

- 为每个资源类型分配一个唯一的整数
- 每个进程按递增顺序申请资源

例子

- 磁带驱动器： 1
- 打印机： 12
- 磁盘驱动器： 5

3.1.2 避免死锁

避免死锁指的是确保系统不进入不安全状态

避免死锁是动态(dynamic)的方法，它根据进程申请资源的附加信息决定是否申请资源

需要掌握的附加信息包括：

1. 当前可用资源
2. 已分配给每个进程的资源
3. 每个进程将来要申请或释放的资源
4. 每个进程可能申请的每种资源类型实例的需求
5. 。。。

3.1.2 避免死锁算法

(1) 资源分配图算法

：适用于每个资源具有单个实例时

(2) 银行家算法

：适用于每个资源具有多个实例时

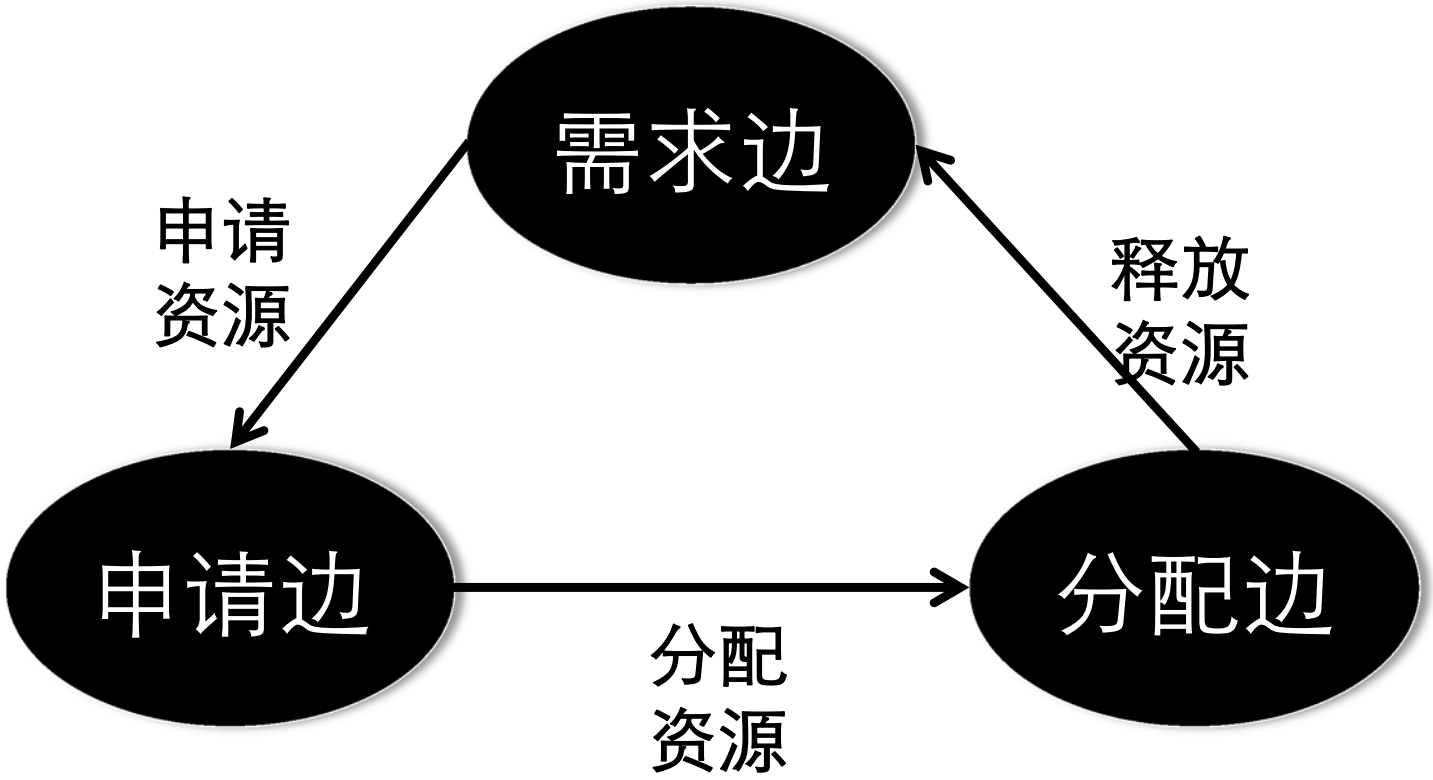
（1）资源分配图算法

引入了新的类型边叫需求边

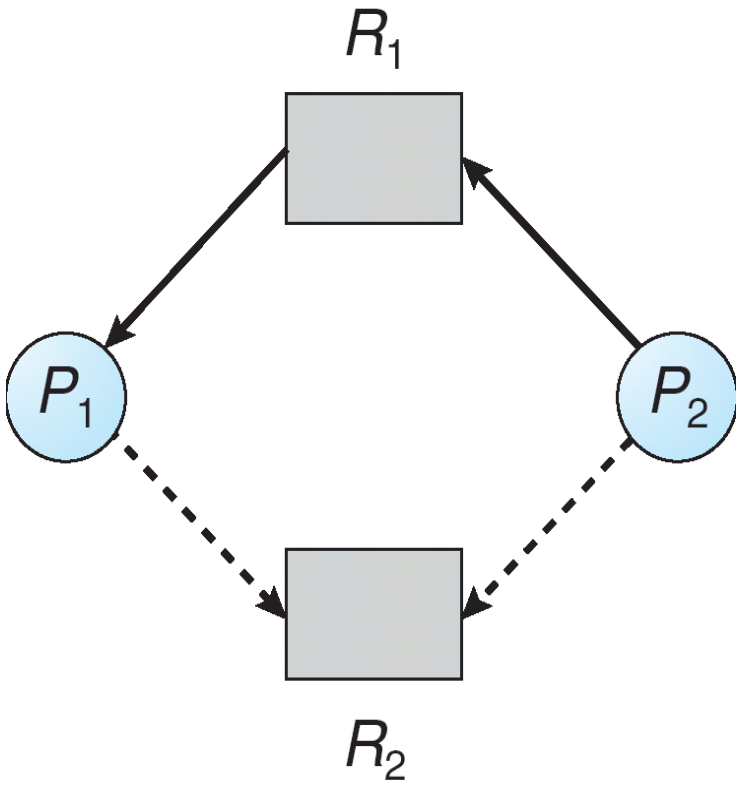
1. 需求边用虚线 Pi---> Rj 表示进程 Pi可能在将来某个时刻申请资源 Rj。
2. 当进程Pi 申请资源Rj时，需求边变成申请边（虚线变成实线）

算法规则：只有在将申请边变成分配边而不会导致资源分配图形成环时，才允许申请资源

资源分配图算法

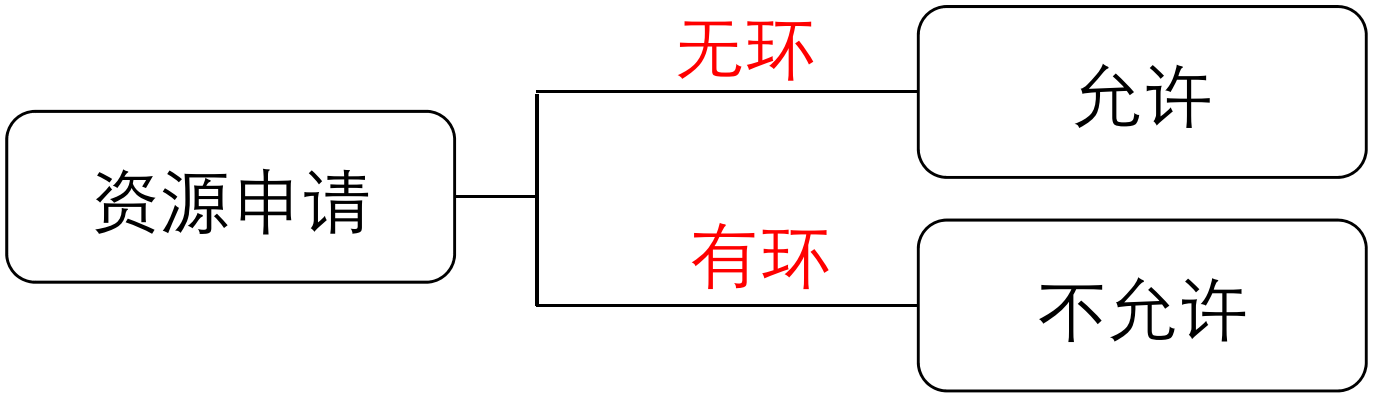


避免死锁的资源分配图

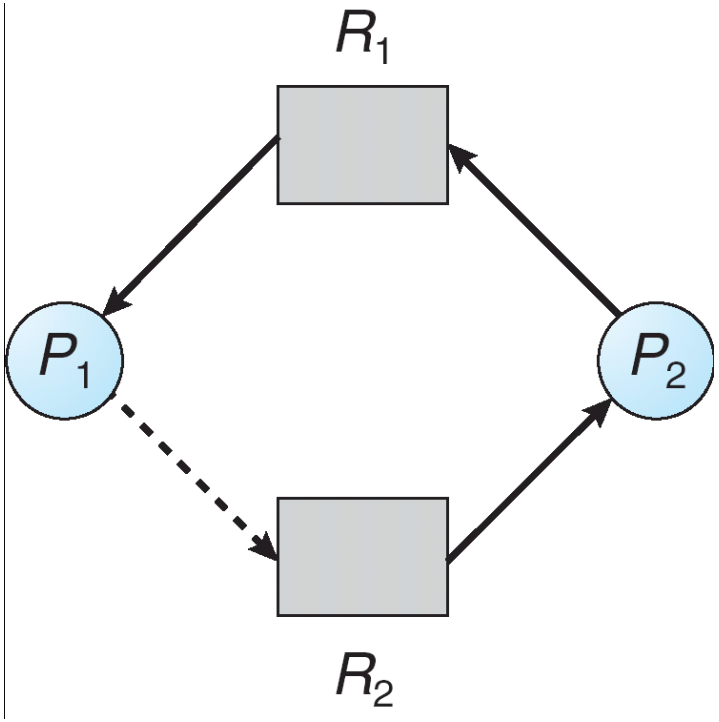


资源分配图算法

- 假设进程 P_i 申请资源 R_j
- 对资源的申请，把申请边变成分配边后，如果没有环就允许申请，如有环就不允许申请



死锁避免的不安全状态



(2) 银行家算法

- 适用于多资源、多实例；当新的进程进入系统时，其可能需要的每种类型资源实例的最大数量，不能超过系统能分配的资源总和。
- 当用户申请一组资源时，系统必须确定这些资源的分配是否仍会使系统处于安全状态
- 需要的数据：
  1. 进程个数
  2. 资源类型的种类
  3. 每个资源的现有实例的数量
  4. 每个进程的资源需求
  5. 现已分配的各种资源类型的实例数量
  6. 在某个时刻，每个进程还需要的资源数量

银行家算法的数据结构

n 进程数,
m 资源类型数.

1. Available(vector向量): 表示可分配的资源数
   - Available[j] = k 表示资源 R_j 的可用资源数是 k个
2. Max（n x m 矩阵）：表示资源最大需求数
   - Max[i,j] = k 表示进程 P_i 可能请求的资源 R_j 的实例个数是 k.
3. Allocation（n x m 矩阵）：表示占有的资源数
   - Allocation[i,j] = k 表示 P_i 已经占有的资源 R_j. 实例的个数是 k.
4. Need（n x m 矩阵）：为完成任务可能仍然需要的资源
   - Need[i, j] = k, 表示进程 P_i 可能需要的资源 R_j 实例数是 k.

银行家算法

（1）安全性算法

：确定计算机系统是否处于安全状态的算法

（2）资源请求算法

：判断是否可安全允许请求的算法

每次进程请求资源的时候，运行资源请求检测算法，确认是否允许请求。***Resource Request Algorithm:***

```
IF  ( Request i <= Need i)  //检测资源的请求是否合法
    IF (Request i <= Available i){ //检测可用资源能否满足请求
            Available i = Available i – Request i ;
            Allocation i = Allocation i + Request i ;
            Need i = Need i – Request i ;
            do Safety Check Algorithm; //检测分配资源后是否安全
    ELSE
            waiting;
    End IF
ELSE
    error message;
End IF
```

Safety Check Algorithm:

Work = Available;
For all *i*, Finish[i] = false;
For all *i* do
    IF (Finish[i] == false && Need *i*<= Work)
        Work = Work + Allocation *i*
        Finish[i] = true
    End IF
End For
IF for all *i*, Finish[i] == true
    Then the system is safety
End IF

（1）安全性算法

STEP 1: Work 和 Finish 分别为长度 m 和 n的向量, 分别初始化为:

Work = Available //可分配的资源数

Finish [i] = false for i = 0, 1, …, n- 1.

STEP 2: 查找 i 使其满足:

(a) Finish [i] = false

(b) $Need_i \leq Work$

如果没有满足以上条件的 i, 那么就跳转到 STEP 4.

STEP 3:

$Work = Work + Allocation_i$

Finish[i] = true

返回到 STEP 2.

STEP 4: 如果对所有 i, Finish[i] == true 那么系统处于安全状态，如果不是就处于不安全状态

银行家算法

- Work = Available;
- For all i, Finish[i] = false;

For all *i* do

if (Finish[i] == false && $Need_i \leq Work$)

Work = Work + $Allocation_i$

Finish[i] = true

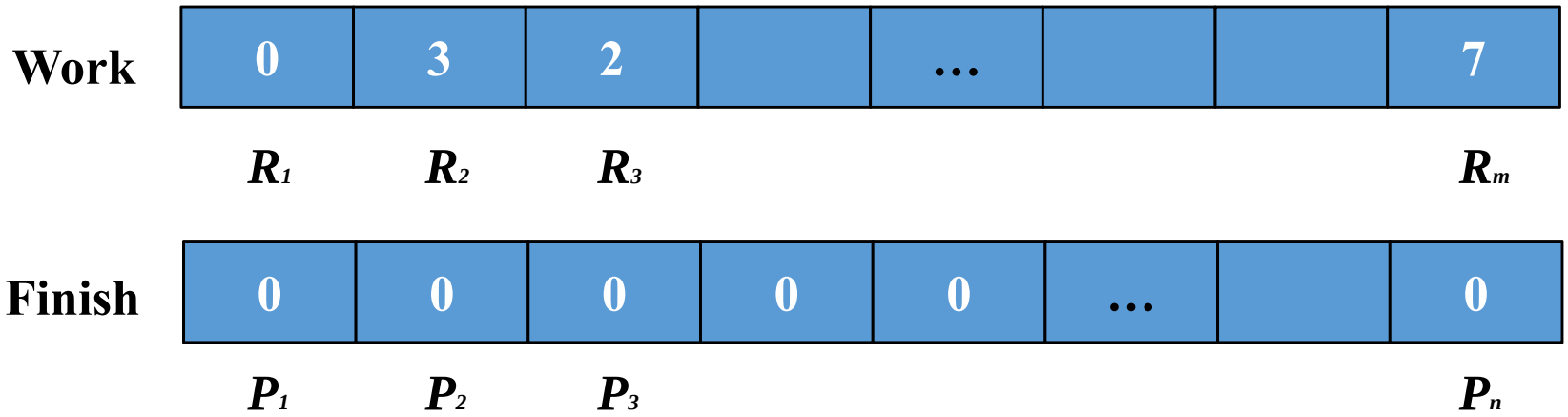
End for

IF for all i, Finish[i] == true

Then the system is safety

End IF

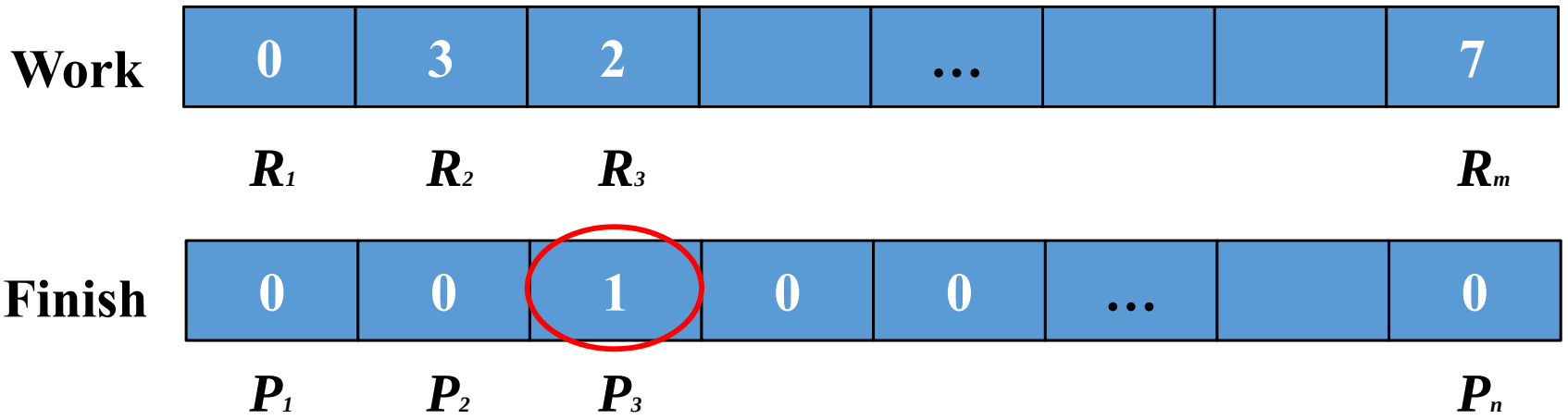
（1）安全性算法



- 查找满足 $Need_i \leq Work$ 的 i

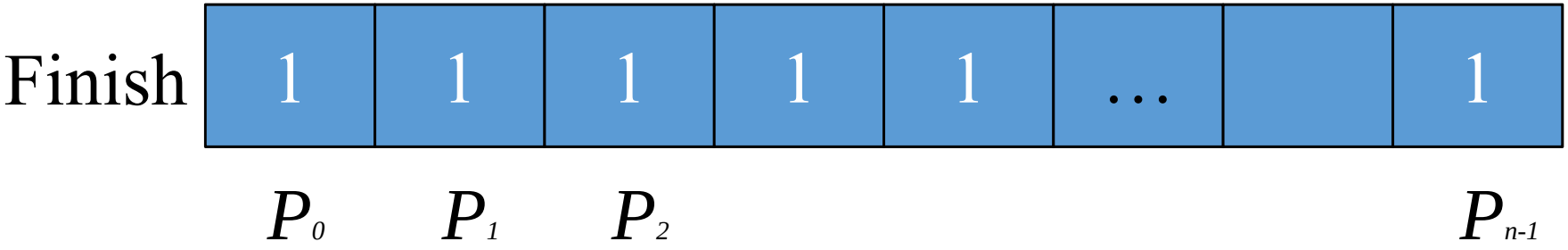
- 假设 P_3 满足以上条件（需要资源 R_2 2个实例）

（1）安全性算法



$$Work = Work + Allocation_i$$

（1）安全性算法



即系统处于安全状态

（2）资源请求算法

设 Request i 为进程 P i 的请求向量，当进程 P i 作出资源请求时，采取如下操作:

STEP 1.

如果 $Request_i \le Need_i$, 那么转到 STEP 2. 否则，产生出错条件，这是因为进程Pi 已超过了其最大请求。

STEP 2.

如果 $Request_i \le Available$, 那么转到 STEP 3. 否则 P_i 必须等待，这是因为没有可用资源。

（2）资源请求算法

STEP 3.

假定系统可以分配给进程 P_i 所请求的资源，并按如下方式修改状态:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

在现阶段，进行安全性检查

1. 如果所产生的资源分配状态是安全的，那么 Pi 可分配到其所需要资源.

2. 如果所产生的资源分配状态是不安全的，那么 Pi必须等待并恢复到原来资源分配状态

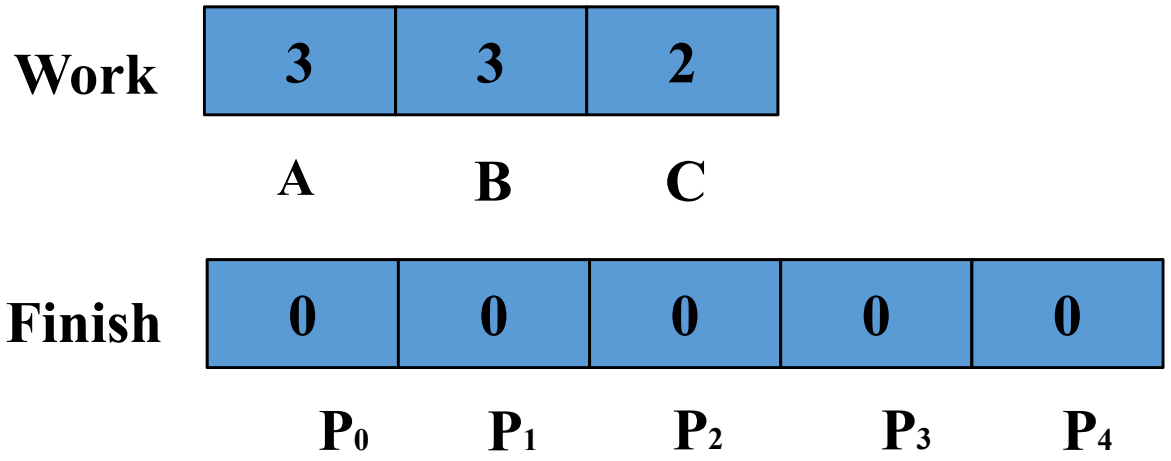
银行家算法举例（安全性算法）

- 5个进程： $P_0\ P_1\ P_2\ P_3\ P_4$;
- 3 个资源类型: A (10个实例), B (5个实例),C (7个实例).
- 当前，资源分配状态如下:

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

用安全性算法检查安全状态，发现安全运行顺序为
【P_1, P_3, P_4, P_0, P_2】

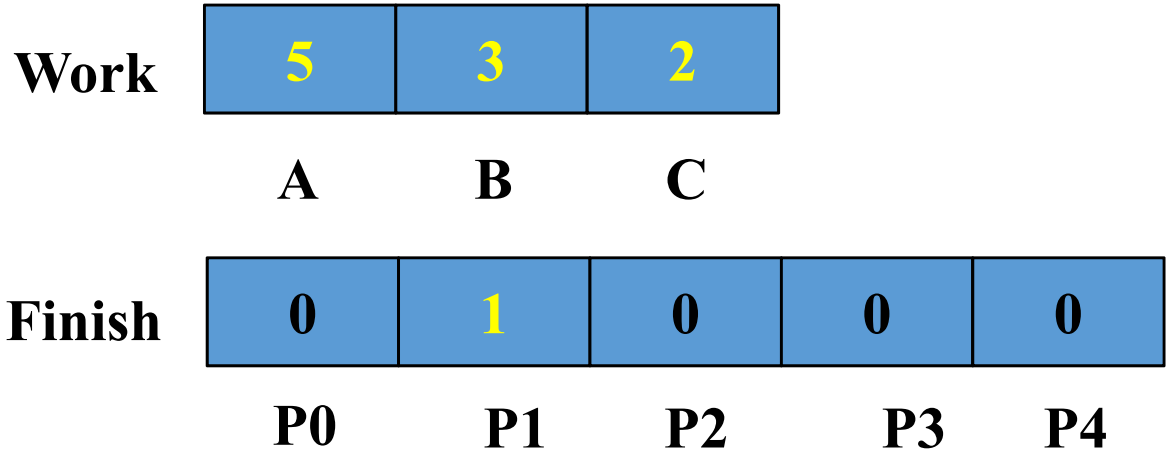
银行家算法举例（安全性算法）



查找满足如下条件的i：
(a) Finish [i] = false
(b) $Need_i \leq Work$

	Allocation	Need
	A B C	A B C
P0	0 1 0	7 4 3
P1	2 0 0	1 2 2
P2	3 0 2	6 0 0
P3	2 1 1	0 1 1
P4	0 0 2	4 3 1

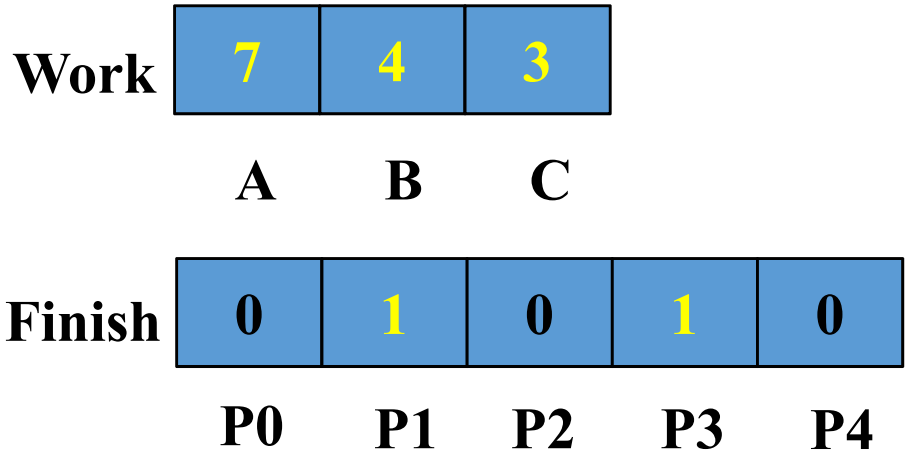
银行家算法举例（安全性算法）



发现 P1 满足以上条件，则 Finish[1] 设置为 1，Work = Work + Allocationi；
Work = 【3,3,2】 + 【2,0,0】
继续往下查找，

	Allocation	Need
	A B C	A B C
P0	0 1 0	7 4 3
P1	2 0 0	1 2 2
P2	3 0 2	6 0 0
P3	2 1 1	0 1 1
P4	0 0 2	4 3 1

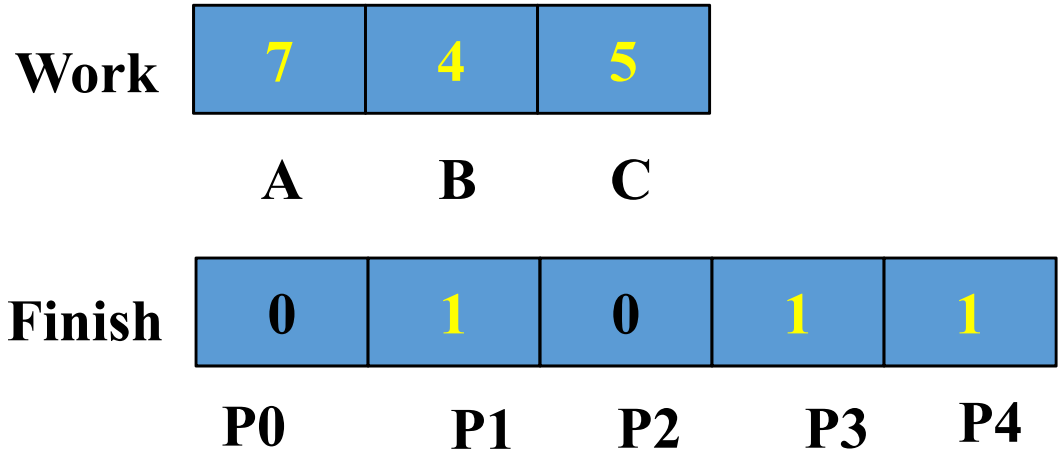
银行家算法举例（安全性算法）



发现 P3 满足条件，则 Finish[3] 设置为 1，
Work = 【5,3,2】 + 【2,1,1】 = 【7,4,3】；
继续往下查找，

	Allocation	Need
	A B C	A B C
P0	0 1 0	7 4 3
P1	2 0 0	1 2 2
P2	3 0 2	6 0 0
P3	2 1 1	0 1 1
P4	0 0 2	4 3 1

银行家算法举例（安全性算法）



发现 P4 满足条件，则 Finish[4]设置为 1，
Work = 【7,4,3】 + 【0,0,2】 = 【7,4,5】；
继续往下查找，发现P0，P2 满足条件，则该状态是安全状态，安全顺序为【P1,P3,P4,P0,P2】。

	Allocation	Need
	A B C	A B C
P0	0 1 0	7 4 3
P1	2 0 0	1 2 2
P2	3 0 2	6 0 0
P3	2 1 1	0 1 1
P4	0 0 2	4 3 1

（资源请求算法）-当P_1请求资源(1,0,2)

检查 $Request_i \le Available$，因$(1,0,2) \le (3,3,2)$，进程P1的请求满足条件，会产生如下新状态:

1. Available = Available - Requesti;
2. $Allocation_i = Allocation_i + Request_i$;
3. $Need_i = Need_i - Request_i$;

	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	2 3 0
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

（资源请求算法）-当P_1请求资源(1,0,2)

通过运行安全性算法确定是否是安全状态，结果发现是安全状态，并【P_1,P_3,P_4,P_0,P_2】为安全顺序。则允许P1的（1,0,2）的请求。

问：

- 是否能允许P_4 的请求(3,3,0)?
- 是否能允许P_0 的请求(0,2,0)?

3.1.3 死锁检测

当系统已进入死锁的状态下，需要提供如下算法：

1. 检测算法：确定系统是否进入死锁
2. 恢复算法：从死锁状态中恢复

从如下两个方面分别考虑这个问题：

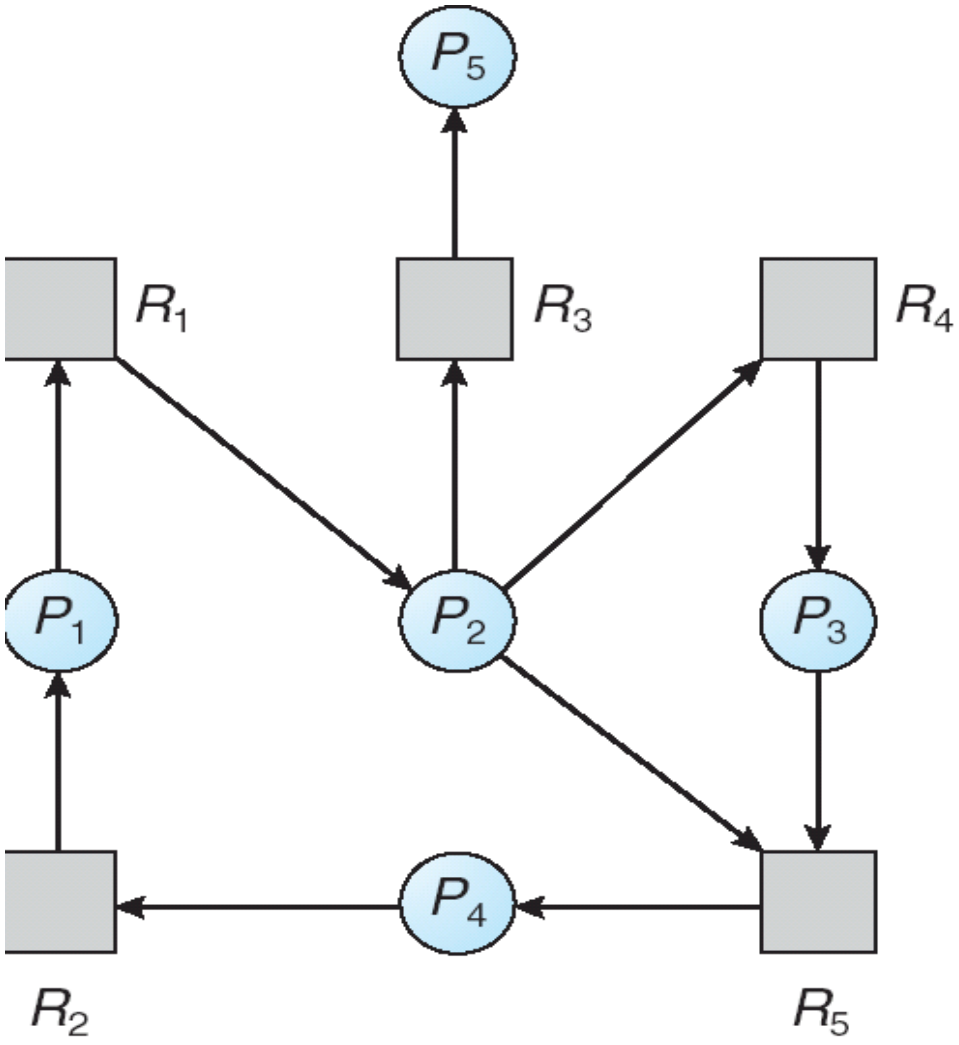
第一个方面，每个资源类型有单个实例

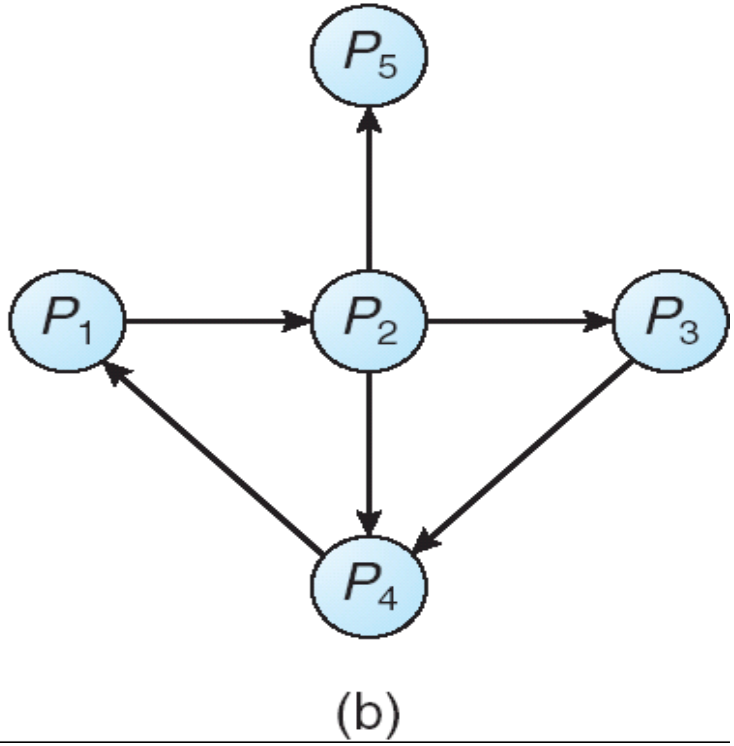
另一个方面，每个资源类型有多个实例

情况1：每种资源类型只有单个实例

- 检测算法：用等待图，它是资源分配图的一个变种
  - 每个节点是进程.
  - $P_i \rightarrow P_j$ 意味着进程 P_i 等待进程 P_j 释放一个 P_i 所需的资源
- 如等待图中有环，系统中存在死锁
- 该算法需维持等待图，并周期性的调用在图中进行搜索的算法

资源分配图和对应的等待图





资源分配图

对应的等待图

情况 2：每种资源类型可有多个实例

1. Available：长度为 m 的向量，表示各种资源的可用实例
2. Allocation：n x m 矩阵，表示当前每个进程资源分配情况
3. Request：n x m 矩阵，表示当前每个进程的资源请求情况

检测算法

STEP 1.

向量Work 和 Finish 的长度分别为 m 和 n，并初始化为:

(a) Work = Available

(b) 对所有 i = 1,2, …, n, 如已经分配资源 $Allocation_i \neq 0$, 则 Finish[i] = false，否则 Finish[i] = true.

STEP 2.

查找满足以下条件的进程 i

(a) Finish[i] == false

(b) $Request_i \leq Work$

如没有，则跳转到 STEP 4.

检测算法

STEP 3.

Work = Work + $Allocation_i$

Finish[i] = true， 并跳转到 STEP 2.

STEP 4.

如果 Finish[i] == false 的 i 存在，则系统处于死锁状态

检测算法

Work = Available

For all *i* do ,

IF $Allocation_i \neq 0$, Finish[i] = false;

ELSE Finish[i] = true;

End For

For all *i* do ,

IF Finish[i] == false && $Request_i \leq Work$

$Work = Work + Allocation_i$;

Finish[i] = true;

End IF

If there is a *i*, Finish[i] == false

then the system is deadlock;

End For

检测算法

STEP 3.

Work = Work + $Allocation_i$

Finish[i] = true， 并跳转到 STEP 2.

STEP 4.

如果 Finish[i] == false 的 i 存在，则系统处于死锁状态

检测算法举例

- 5 个进程： P_0 - P_4; 3 资源类型: A(7个实例), B(2个实例), C (6 个实例).
- 当前资源分配状态:

	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	

现处于安全状态，安全顺序为【P_0, P_2, P_3, P_4, P_1】。

检测算法举例

- 安全顺序【P_0, P_2, P_3, P_4, P_1】

Finish[i]	false	false	false	false	false
	P0	P1	P2	P3	P4

	Alloc.	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	0 1 0
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Run P_0

	Alloc.	Request	Available
	A B C	A B C	A B C
P0	0 0 0	0 0 0	
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	3 1 3
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Run P_2

检测算法举例

	Alloc.	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	5 2 4
P3	2 11	1 0 0	
P4	0 0 2	0 0 2	

Run P_3

	Alloc.	Request	Available
	A B C	A B C	A B C
P0	0 0 0	0 0 0	
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	5 2 6
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Run P_4

检测算法举例

Finish[i]	true	true	true	true	true
	P0	P1	P2	P3	P4

	Alloc.	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	7 2 6
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	

Run P_1

检测算法举例

- 当前，如进程 P_2 请求资源 C 的一个实例，则处于死锁。

	Alloc.	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 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Run P_2

检测算法的应用

- 何时调用算法取决于:
  1. 死锁发生的频率
  2. 死锁发生时，有多少进程会受影响

- 每次资源请求调用检测算法
- 每次资源请求不被允许时调用检测算法

4. 死锁恢复：进程终止

1. 终止所有死锁进程
2. 一次只终止一个进程直到取消死锁循环为止,但要考虑
   - 进程的优先级
   - 进程已计算了多久，进程在完成指定任务之前还需要多久
   - 进程使用了多少类型的资源
   - 进程需要多少资源以完成
   - 多少资源需要被终止
   - 进程是交互的还是批处理的

4. 死锁恢复: 资源抢占

通过抢占资源以取消死锁，逐步从进程中抢占资源给其他进程使用，直到死锁被打破为止。

- 选择一个牺牲品： 抢占哪些资源和哪个进程
  - 但需要考虑饥饿： 避免同一个进程总成为牺牲品
- 回滚（Rollback）： 必须把不能正常运行的进程，回滚到某个安全状态，以便重启进程

Q&A