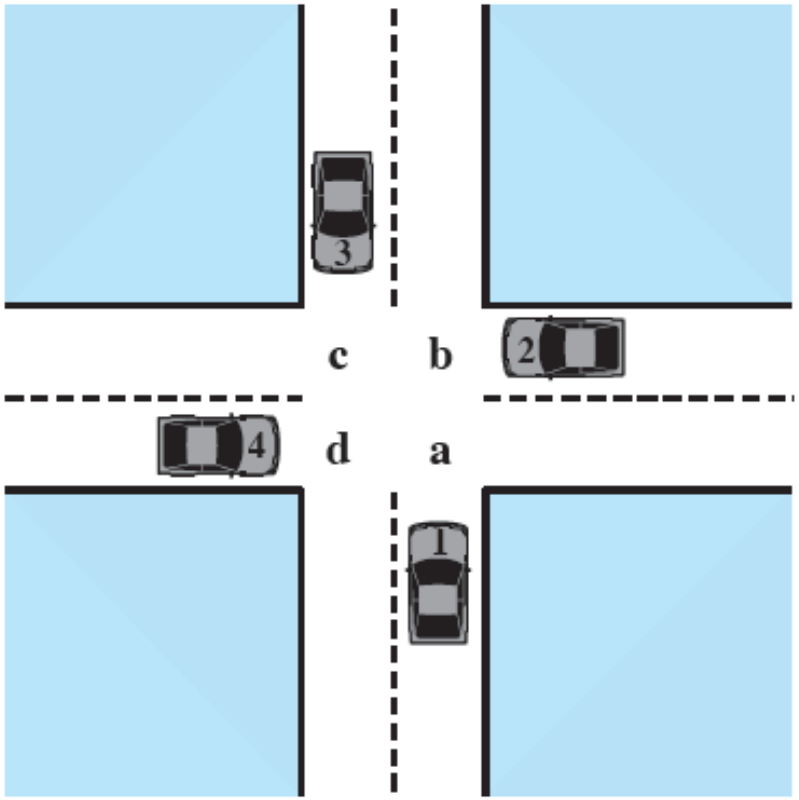
第6章 并发：死锁和饥饿 ★★★★★

- 主要内容
  - 6.1 死锁原理 ★★★★★
  - 6.2 死锁预防 ★★★★★
  - 6.3 死锁避免 ★★★★★
  - 6.4 死锁检测 ★★★★★
  - 6.5 一种综合的死锁策略 ★★★★★
  - 6.6 哲学家就餐问题 ★★★★★
  - 6.7 UNIX的并发机制 ★★★★★
  - 6.8 Linux内核并发机制（略）
  - 6.9 Solaris线程同步原语（略）
  - 6.10 Windows 7并发机制（略）

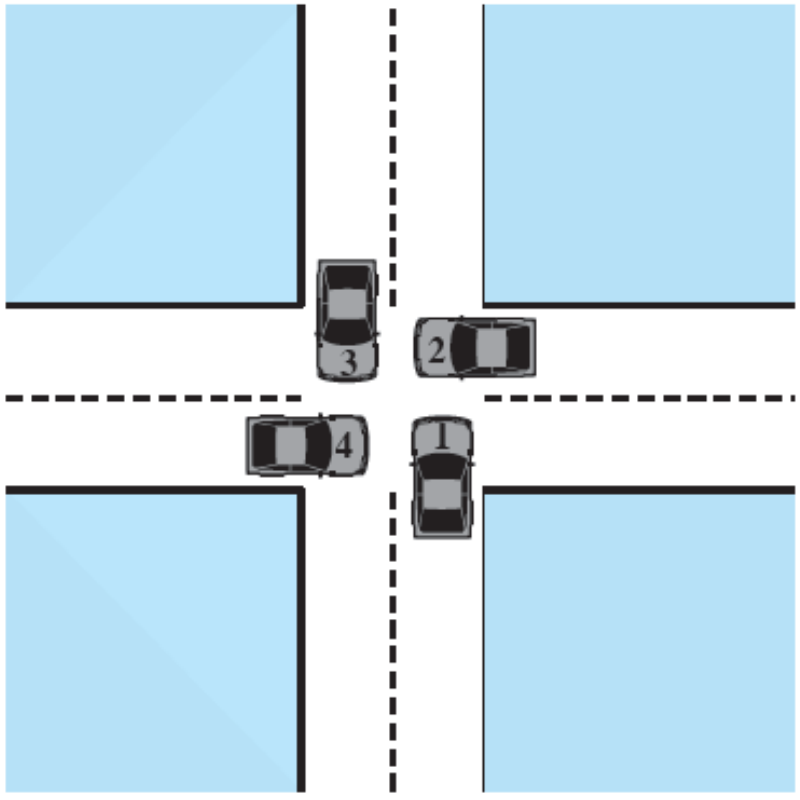
6.1 死锁原理 ★★★★★

- 死锁：一组相互竞争系统资源或进行通信的进程间的永久阻塞。
- 死锁问题没有一种有效的通用解决方案。
- 所有死锁都涉及两个或多个进程之间对资源需求的冲突。

死锁的图示



(a) Deadlock possible



(b) Deadlock

联合进程图

- 进程P,Q， 资源A,B

进程P	进程Q
…	…
获得A	获得B
…	…
获得B	获得A
…	…
释放A	释放B
…	…
释放B	释放A

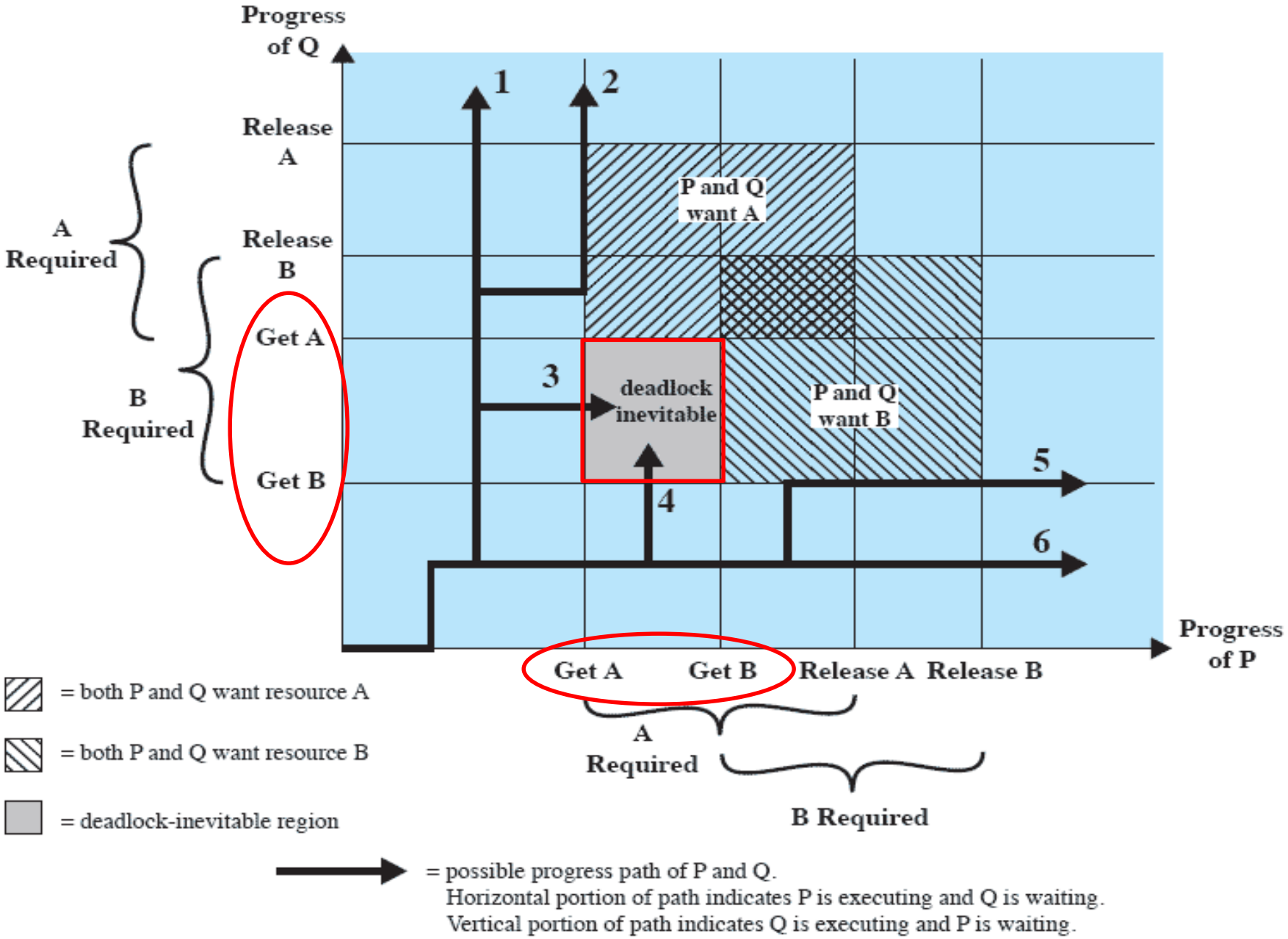


Figure 6.2 Example of Deadlock

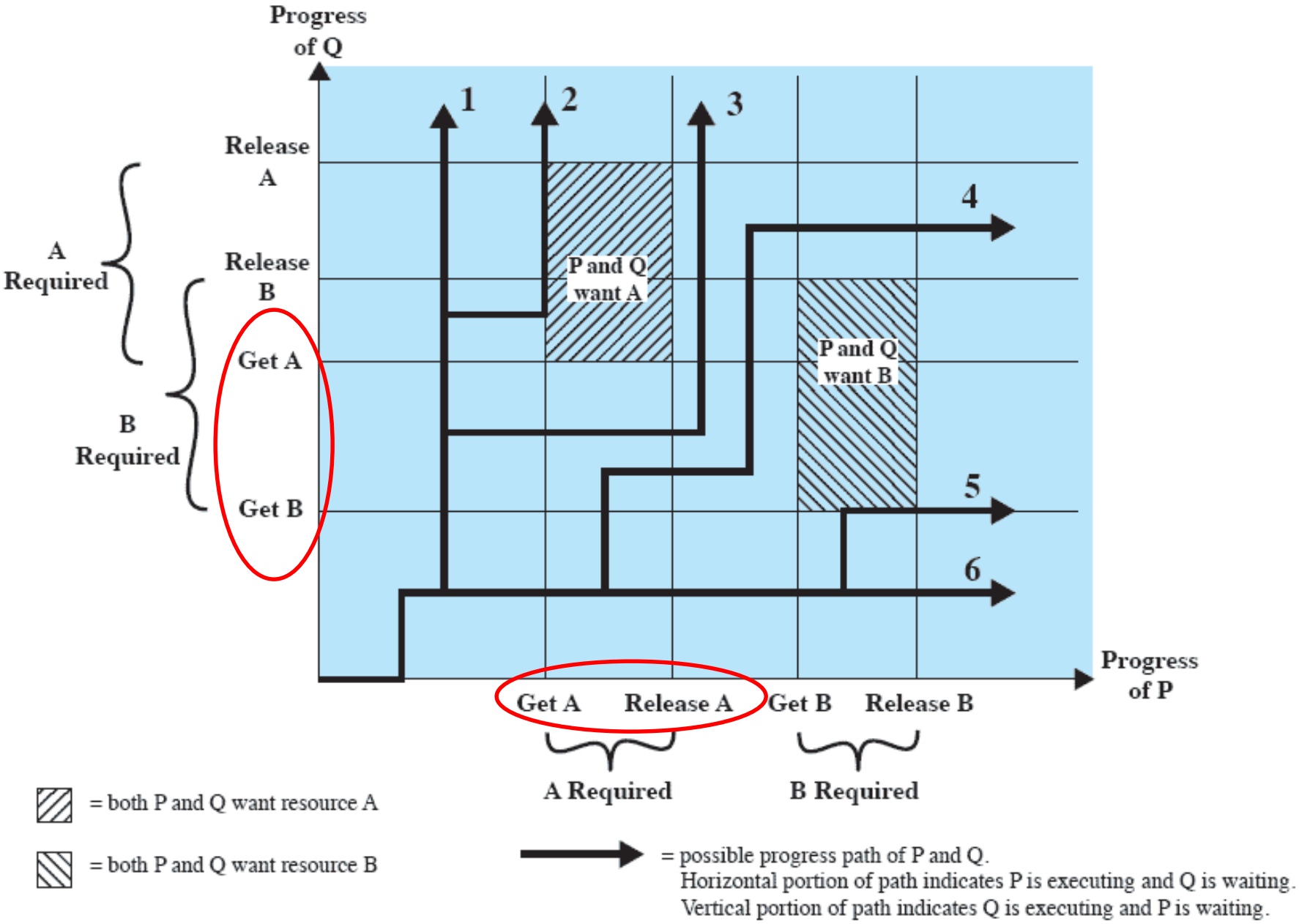


Figure 6.3 Example of No Deadlock [BACO03]

6.1.1 可重用资源 ★★★★★

- 一次只能供一个进程安全地使用，并且不会由于使用而耗尽的资源。
  - 处理器、I/O通道、内存、外存、设备、文件、数据库、信号量等。

可重用资源死锁的例子

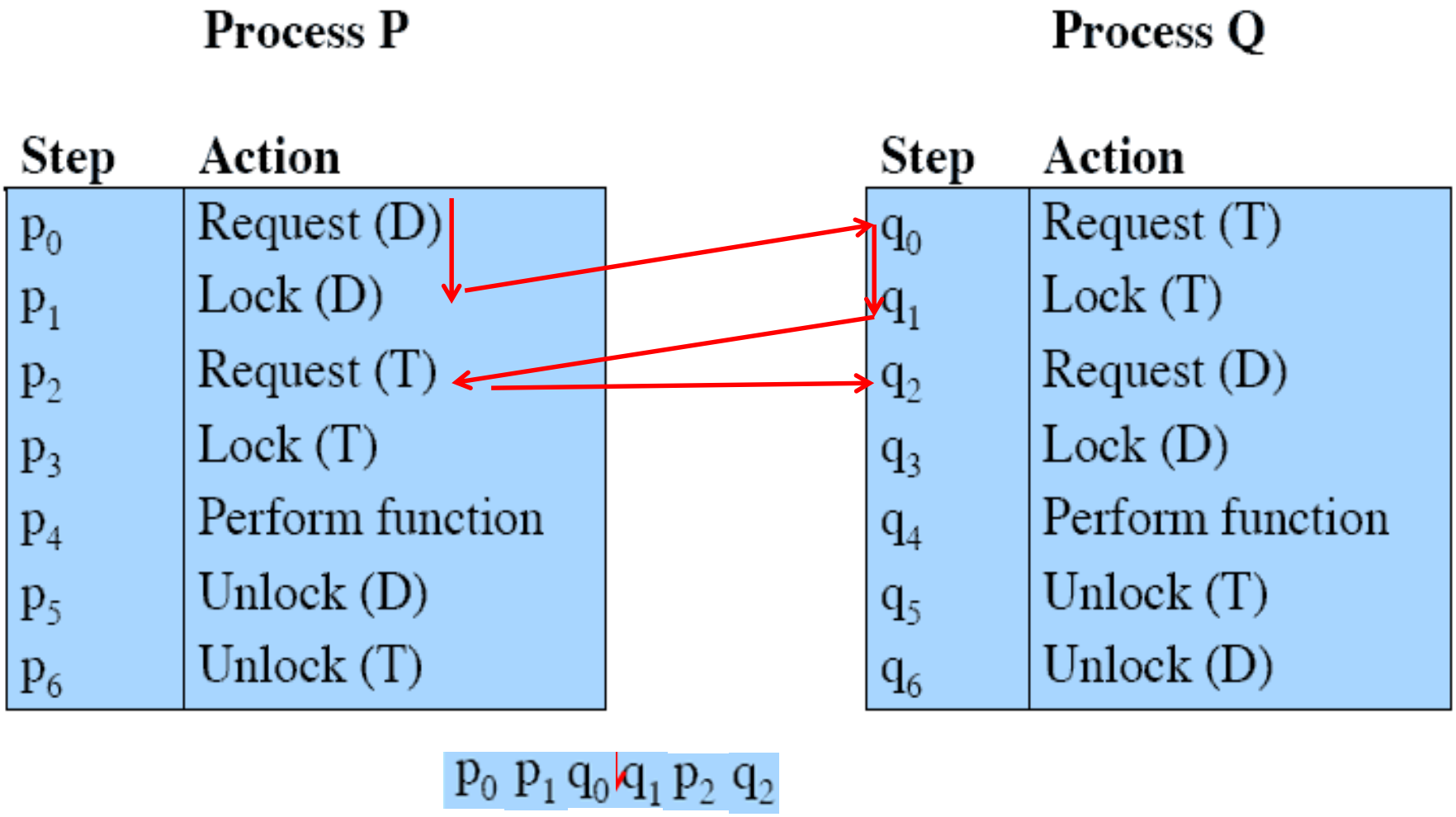
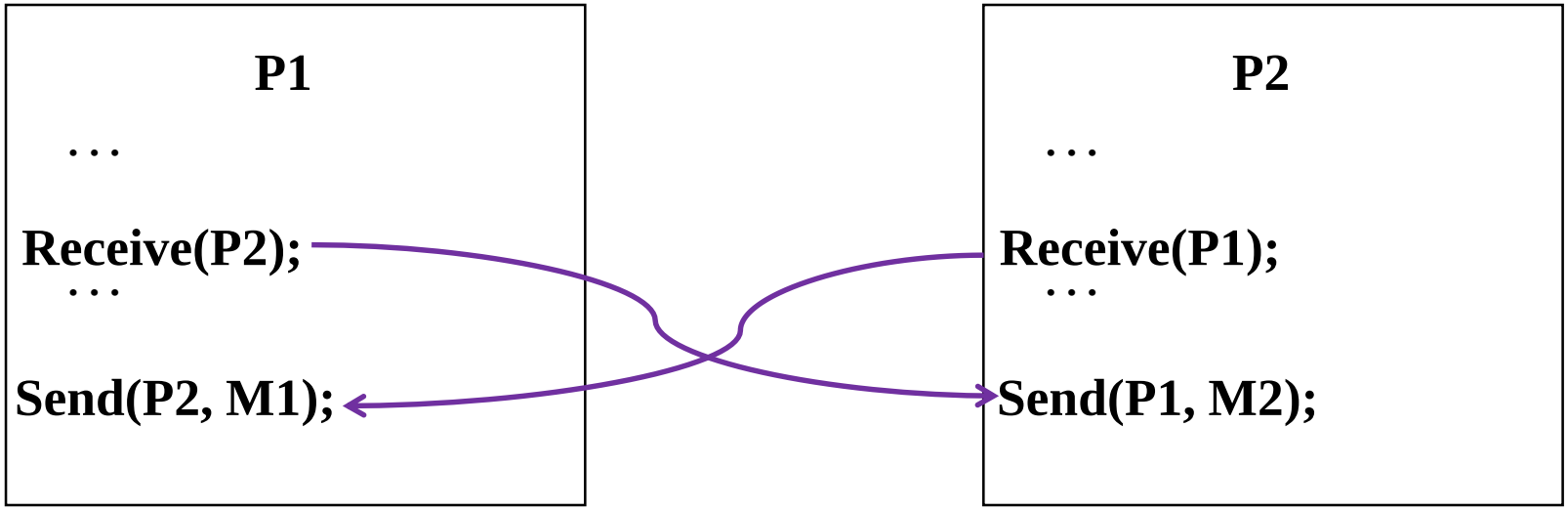


Figure 6.4 Example of Two Processes Competing for Reusable Resources

6.1.2 可消耗资源 ★★★★★

- 可以被创建和销毁的资源。
  - 中断、信号、消息、I/O缓冲区中的信息等。
- 可消耗资源死锁的例子



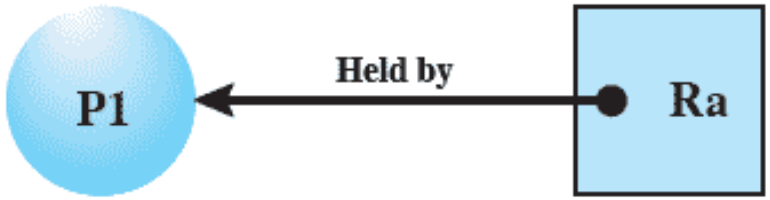
没有一个可以解决**所有类型**死锁的有效策略！！！

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

6.1.3 资源分配图 ★★★★★

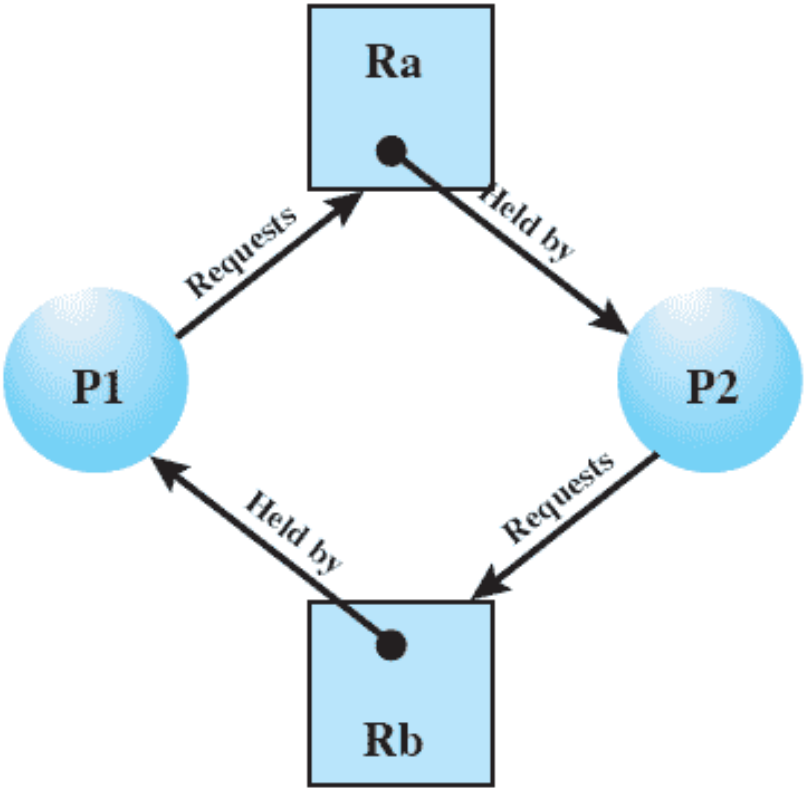
(a) Resouce is requested



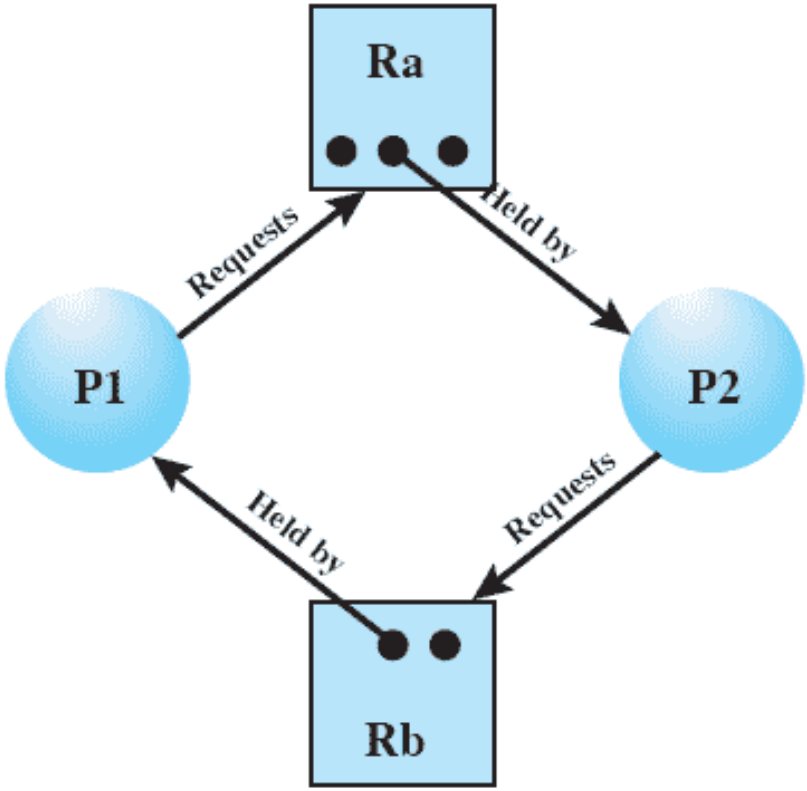
(b) Resource is held

进程是圆形，资源是方形！

资源分配图



(c) Circular wait



(d) No deadlock

资源分配图

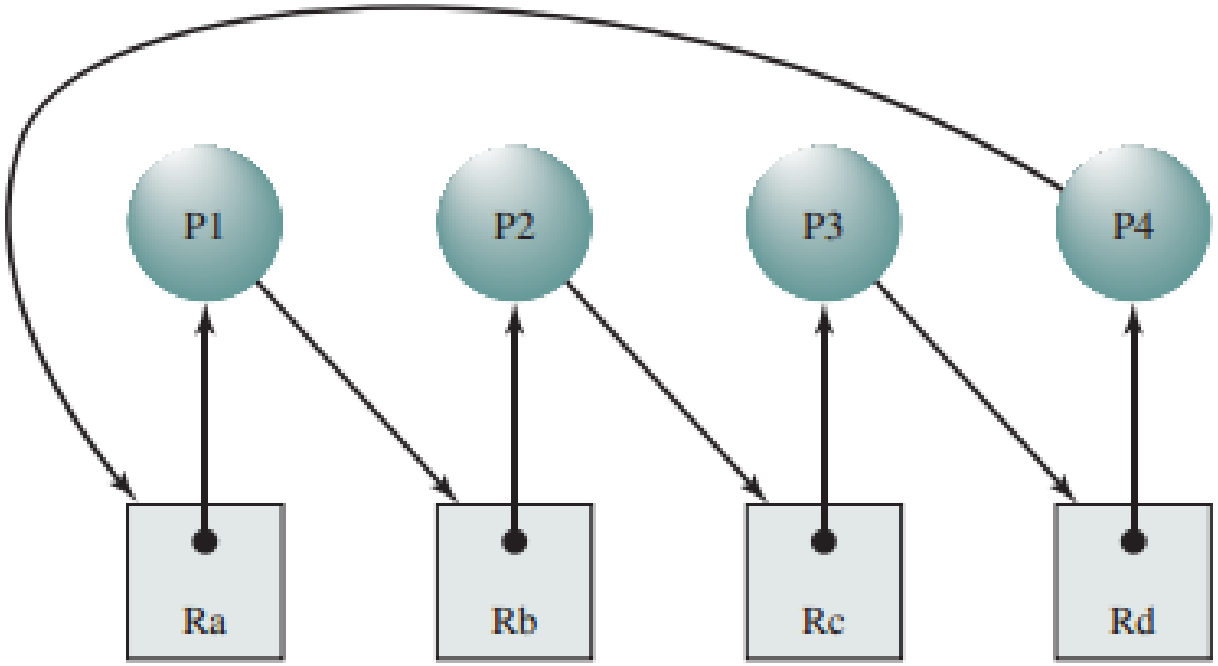


Figure 6.6 Resource Allocation Graph for Figure 6.1b

6.1.4 死锁的条件 ★★★★★

- 互斥
  - 一次只有一个进程可以使用一个资源。
- 占有且等待
  - 当一个进程等待其他进程时，继续占有已经分配的资源。
- 不可抢占
  - 不能强行抢占进程已占有的资源。
- 循环等待
  - 存在一个封闭的进程链，使得每个进程至少占有此链中下一个进程所需要的一个资源。

处理死锁的方法

- 死锁预防
- 死锁避免
- 死锁检测与恢复

6.2 死锁预防 ★★★★★

- 试图设计一种系统来排除发生死锁的可能性。
  - 间接的死锁预防方法

    预防前三个条件
  - 直接的死锁预防方法

    预防第四个条件

1、互斥

- 该条件不可能被禁止

2、占有且等待

- 可要求进程一次性地请求所有需要的资源，并且阻塞进程直到所有请求都同时满足。
- 存在的问题：
  - 一个进程可能被阻塞很长时间，已等待满足其所有的资源请求。
  - 分配给一个进程的资源可能有相当长的一段时间不会被使用。
  - 一个进程可能事先并不知道它所需要的全部资源。

3、不可抢占

- 如果占有某些资源的进程进一步申请资源时被拒绝，则该进程必须释放它最初占有的资源。
- 如果进程A请求当前被进程B占有的一个资源，则操作系统可以抢占进程B，要求它释放资源。

4、循环等待

- 定义资源类型的线性顺序。如果一个进程已经分配到了R类型的资源，那么它接下来请求的资源只能是那些排在R类型之后的资源类型。
- 存在的问题：
  - 会导致进程执行速度变慢；
  - 可能在没有必要的情况下拒绝资源访问。

6.3 死锁避免 ★★★★★

- 如果一个进程的请求会导致死锁，则不启动此进程；
- 如果一个进程增加的资源请求会导致死锁，则不允许此分配。

6.3.1 进程启动拒绝—数据结构

系统有n个进程和m种不同类型的资源：

- Resource=R=(R_1, R_2, ..., R_m)
  - 系统中每种资源的总量
- Available=V=(V_1, V_2, ...V_m)
  - 未分配给进程的每种资源的总量
- $$Claim=C=\begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$$
  - C_{ij}表示进程i对资源j的需求

数据结构

- $$Allocation=A=\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$$
  - A_{ij}表示当前分配给进程i的资源j

关系式

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$，对所有j

2. $C_{ij} \leqslant R_j$，对所有i, j

3. $A_{ij} \leqslant C_{ij}$，对所有i, j

死锁避免策略

- 仅当 $R_j \geqslant C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ （对所有j）时，才启动一个新进程P_{n+1}。
- 即只有所有当前进程的最大请求量加上新的进程请求可以满足时，才会启动该进程。

6.3.2 资源分配拒绝

- 主要思想：动态的检测资源分配状态以确保循环等待条件不可能成立。
- 银行家算法
  - 银行家拥有一笔周转资金
  - 客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款
  - 银行家应谨慎的贷款，防止出现坏帐
- 用银行家算法避免死锁
  - 操作系统（银行家）
  - 操作系统管理的资源(周转资金)
  - 进程（要求贷款的客户）

1、安全状态

若存在一个安全序列，则系统处于安全状态。例：

	最大需求	当前占有
P0	10	5
P1	4	2
P2	9	2

说明：某系统有12台磁带驱动器

P0：最多要求10台

P1：最多要求4台

P2：最多要求9台

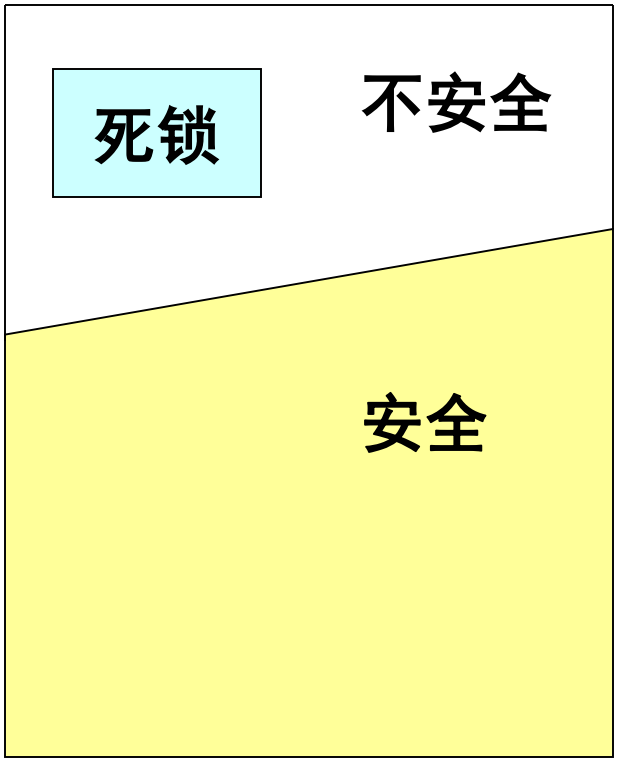
P1 分配2；用完释放4；则系统剩余5

P0 分配5；用完释放10；则系统剩余10

P2 分配7；用完释放9；则系统剩余12

存在安全序列<P1，P0，P2>

安全、不安全和死锁状态空间



结论：

- 安全状态不是死锁状态
- 死锁状态是不安全状态
- 不是所有不安全状态都是死锁状态

2、银行家算法

- 数据结构

- Available：Available[j]=k
  - 资源类型R_j 现有k个实例
- Claim： Claim[i,j]=k
  - 进程P_i最多可申请k个R_j的实例
- Allocation：Allocation[i,j]=k
  - 进程P_i现在已经分配了k个R_j的实例
- Need：Need[i,j]=k
  - 进程P_i还可能申请k个R_j的实例
  - Need[i,j] = Claim[i,j] - Allocation[i,j]

符号说明

- X<=Y
  - （X和Y是长度为n的向量），当且仅当对所有i=1,2,...,n，X[i]<=Y[i]
- $Allocation_i$
  - 表示分配给进程P_i的资源（将Allocation每行作为向量）
- Need同Allocation

安全性算法

- 用于确定计算机系统是否处于安全状态

1) 设Work和Finish分别是长度为m和n的向量，初始化
   Work:=Available，Finish[i]=false(i=1,2,...,n)
2) 查找 i 使其满足
   a. Finish[i] = false
   b. $Need_i$ <=Work
      若没有这样的 i 存在，转到4）。
3) Work := Work + $Allocation_i$
   Finish[i] := true
   返回到2）
4) 如果对所有 i，Finish[i] = true，则系统处于安全状态

资源请求算法

- 设$Request_i$ 为进程P_i的请求向量

1) 如果$Request_i$ <= $Need_i$ ，那么转到第2）步。否则，产生出错条件，因为进程已超过了其请求。

2) 如果$Request_i$ <=Available，那么转到第3）步。否则，P_i等待，因为没有可用资源。

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 – $Request_i$;
   - $Allocation_i$:= $Allocation_i$ + $Request_i$;
   - $Need_i$:= $Need_i$ – $Request_i$;

4) 调用安全性算法确定新状态是否安全
   - 安全—操作完成且进程P_i分配到其所需要的资源
   - 不安全—进程P_i必须等待，并将数据结构恢复到原状态（即 3）的逆操作）

银行家算法举例

说明：

- 5个进程P_0...P_4，
- 3种资源类型A、B、C，且实例个数分别为10、5、7
- T0时刻状态如图所示

	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
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			

✓ 问题：

1) T0时刻是否为安全状态？若是，给出安全序列。

2) 若在T0时刻进程P1请求资源（1，0，2），是否能实施分配？为什么？

3) 在（2）的基础上，若进程P4请求资源（3，3，0），是否能实施分配？为什么？

4) 在（3）的基础上，若进程P0请求资源（ 0，2，0），是否能实施分配？为什么？

转换（Need）

	Claim			Allocation			Need			Available		
	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			

① T0时刻是否为安全状态？

	Work			Need			Allocation			Work+allocation			finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	3	3	2	1	2	2	2	0	0	5	3	2	True
P3	5	3	2	0	1	1	2	1	1	7	4	3	True
P4	7	4	3	4	3	1	0	0	2	7	4	5	True
P2	7	4	5	6	0	0	3	0	2	10	4	7	True
P0	10	4	7	7	4	3	0	1	0	10	5	7	True

T0时刻是安全的，存在安全序列<P1，P3，P4，P2，P0>

② P1请求资源Request(1，0，2)，执行银行家算法：

1> Request（1，0，2）<=Need（1，2，2）；

2> Request（1，0，2）<=Available（3，3，2）；

3> 系统试探为P1分配资源后，资源情况是：

	Claim			Allocation			Need			Available		
	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			

4> 执行安全性算法，得到安全序列 <*P1*, *P3*, *P4*, *P0*, *P2*>：

	Work			Need			Allocation			Work+allocation			finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	2	3	0	0	2	0	3	0	2	5	3	2	True
P3	5	3	2	0	1	1	2	1	1	7	4	3	True
P4	7	4	3	4	3	1	0	0	2	7	4	5	True
P0	7	4	5	7	4	3	0	1	0	7	5	5	True
P2	7	5	5	6	0	0	3	0	2	10	5	7	True

③ P4请求资源Request（3，3，0），执行银行家算法:

1> Request(3, 3, 0) <= Need(4, 3, 1);

2> Request(3, 3, 0) >=Available(2, 3, 0), P4 等待.

④ P0请求资源Request（0，2，0），执行银行家算法:

1> Request（0，2，0）<= Need（7，4，3）；

2> Request（0，2，0）<= Available（2，3，0）；

3> 系统试探为P0分配资源后，资源情况是:

	Claim			Allocation			Need			Available		
	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			

4> 再进行安全性检查，Available(2, 1, 0)不能满足任何进程，进入不安全状态，恢复旧数据结构， P0等待。

练习

说明：

- 5个进程P_1...P_5，
- 3种资源类型A、B、C，且实例个数分别为17、5、20
- T0时刻状态如图所示

	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
P_1	2	1	2	5	5	9	2	3	3
P_2	4	0	2	5	3	6			
P_3	4	0	5	4	0	11			
P_4	2	0	4	4	2	5			
P_5	3	1	4	4	2	4			

✓ 问题：

1) T0时刻是否为安全状态？若是，给出安全序列

2) 若在T0时刻进程P2请求资源（0，3，4），是否能实施分配？为什么？

3) 在（2）的基础上，若进程P4请求资源（2，0，1），是否能实施分配？为什么？

4) 在（3）的基础上，若进程P1请求资源（ 0，2，0），是否能实施分配？为什么？

死锁避免的优缺点

- 优点
  - 不需要死锁预防中的抢占和回滚进程
  - 比死锁预防的限制少
- 缺点
  - 必须事先声明每个进程请求的最大资源。
  - 进程必须是无关的，其执行的顺序必须没有任何同步要求的限制。
  - 分配的资源数目必须是固定的。
  - 在占有资源时，进程不能退出。

6.4 死锁检测 ★★★★★

- 死锁检测策略不限制资源访问或约束进程行为。
- 系统周期性地执行检测算法，检测循环等待条件是否成立。

6.4.1 死锁检测算法

- 检测时机
  - 每个资源请求发生时
  - 隔一段时间
- 每次资源请求时检测死锁
  - 优点：可以尽早地检测死锁情况
  - 缺点：频繁的检查会耗费相当多的处理器时间

死锁检测的常用算法

- 新定义一个请求矩阵Q

1. 标记Allocaiton矩阵中一行全为零的进程。
2. 初始化一个临时向量W，令W等于Available向量。
3. 查找下标i，使进程i当前未标记且Q的第i行小于等于W，如果找不到这样的行，终止算法。
4. 如果找到这样的行，标记进程i，并把Allocation矩阵中的相应行加到W中，返回步骤3.

若最后有未标记的进程时，存在死锁，每个未标记的进程都是死锁的。

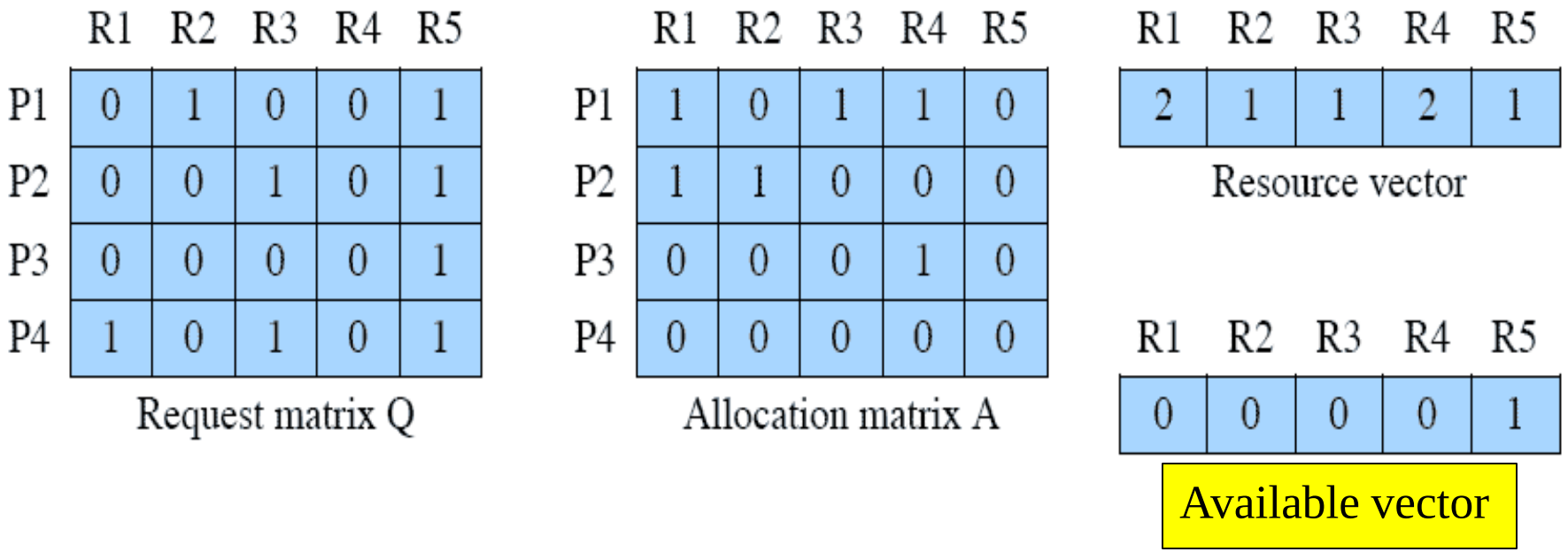
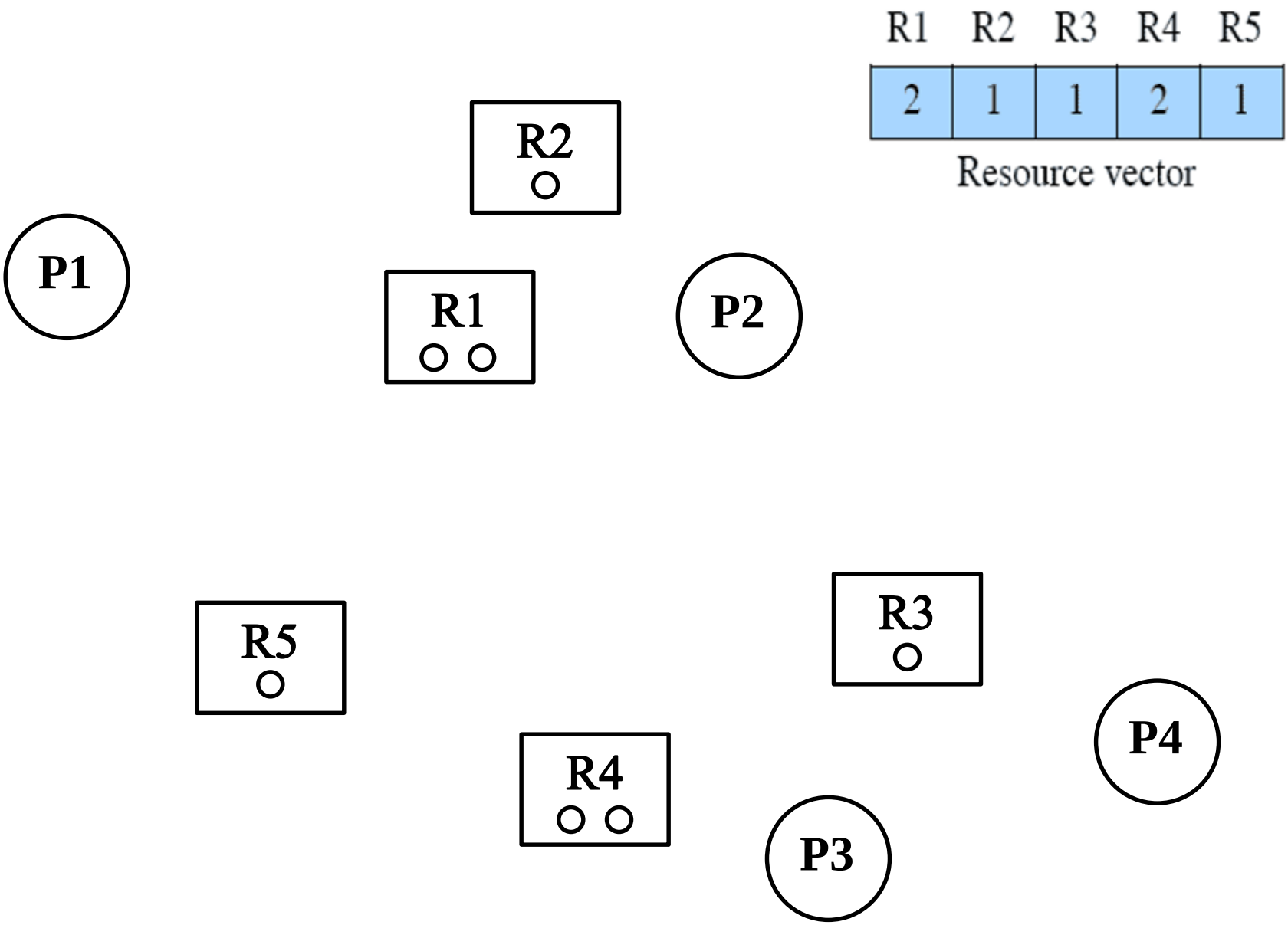
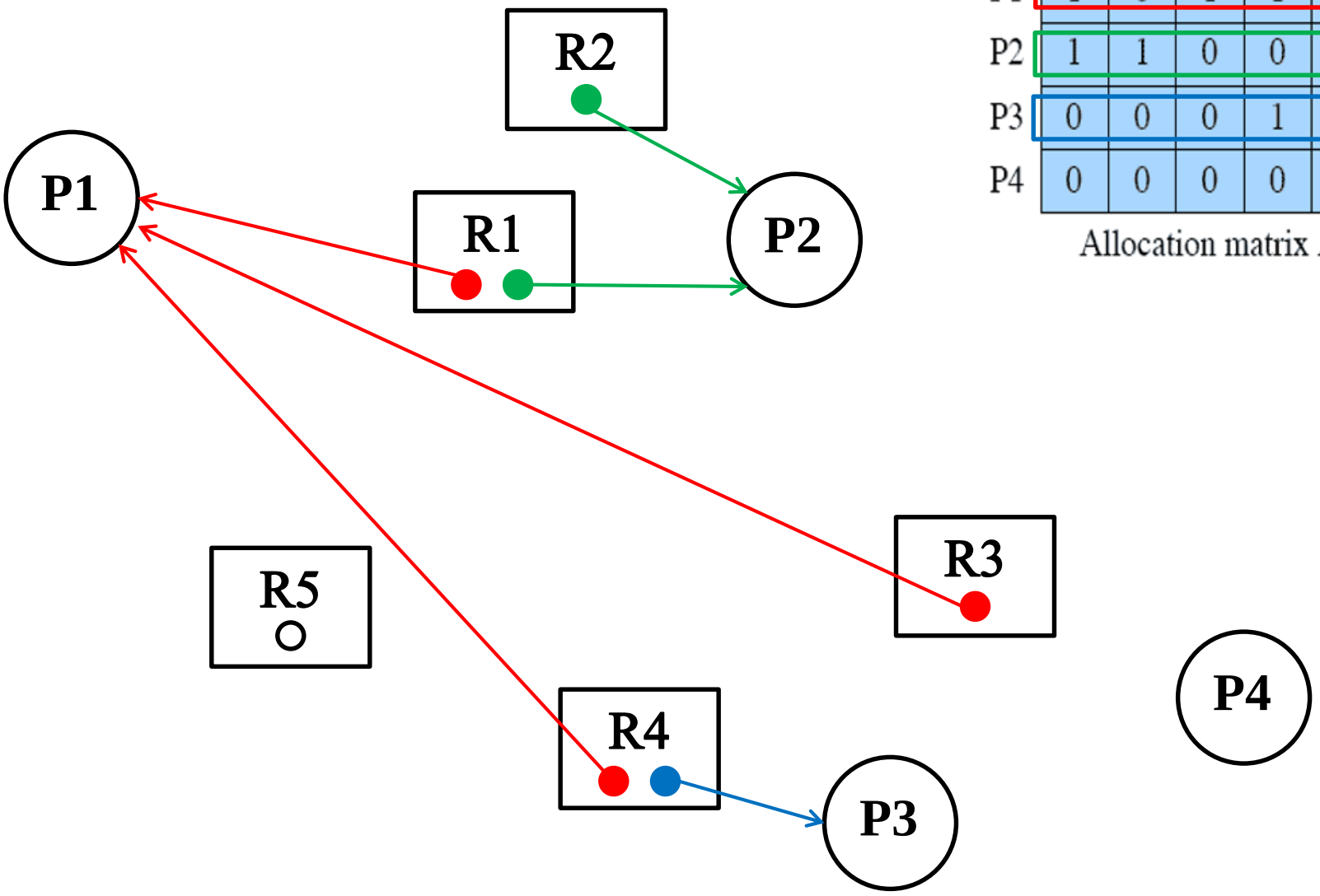


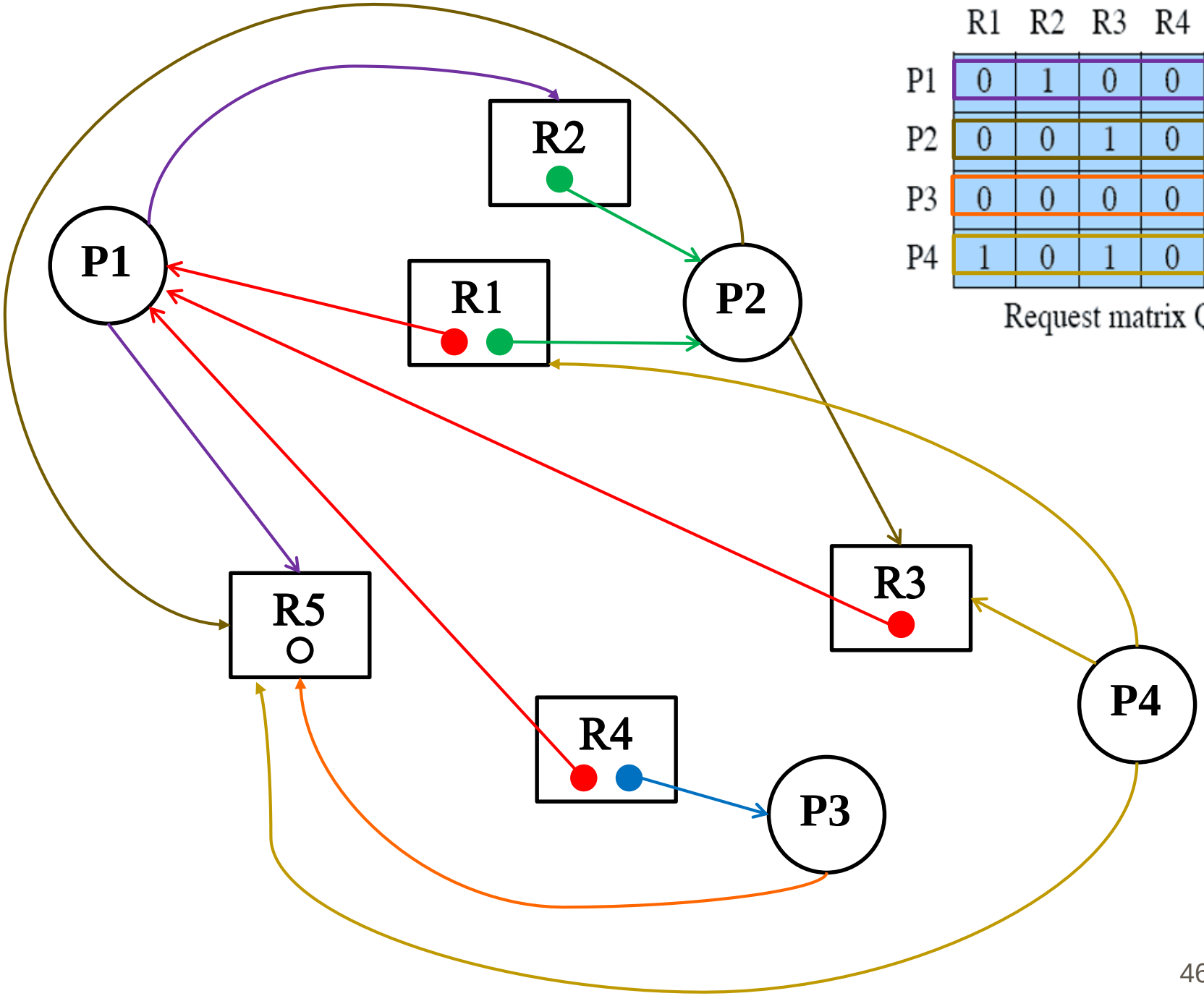
Figure 6.10 Example for Deadlock Detection


R1 R2 R3 R4 R5
2 1 1 2 1
Resource vector
R2
P1
R1
P2
R5
R3
P4
R4
P3



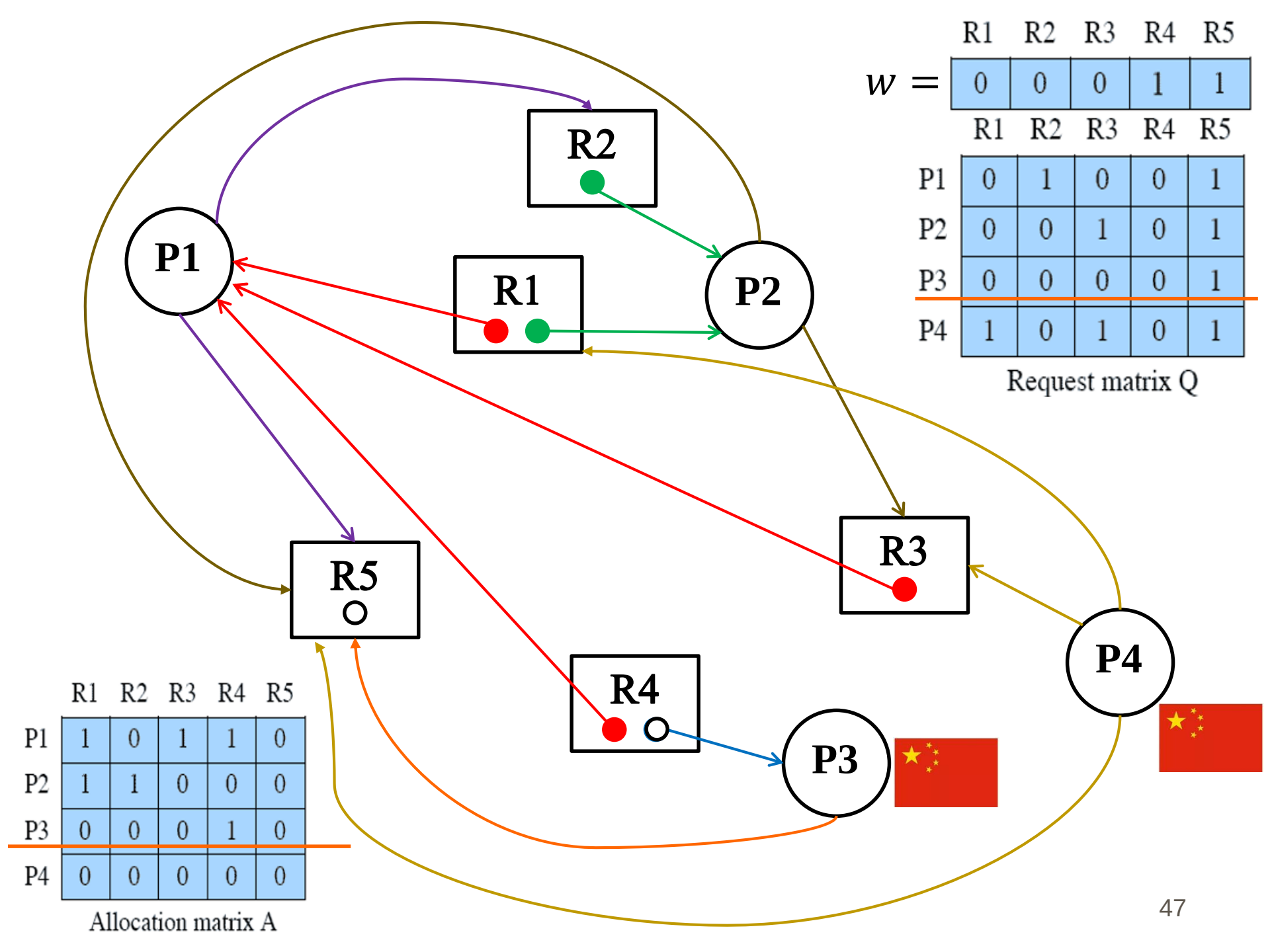
	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A



	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q



$w =$

R1	R2	R3	R4	R5
0	0	0	1	1

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

6.4.2 恢复

1. 取消所有死锁进程。
2. 把每个死锁进程回滚到某些检查点，并重新启动所有进程。
3. 连续取消死锁进程直到不再存在死锁。选择取消进程的顺序基于某种最小代价原则。每次取消后，必须重新调用检测算法，以测试是否仍存在死锁。
4. 连续抢占资源直到不再存在死锁。同3。

6.5 一种综合的死锁策略 ★★★★★

- 可交换空间
  - 通过要求一次性分配所有请求的资源来预防死锁
  - 死锁避免
- 进程资源
  - 死锁避免
  - 资源排序的死锁预防
- 内存
  - 基于抢占的预防
- 内部资源
  - 基于资源排序的预防

6.6 哲学家就餐问题 ★★★★★

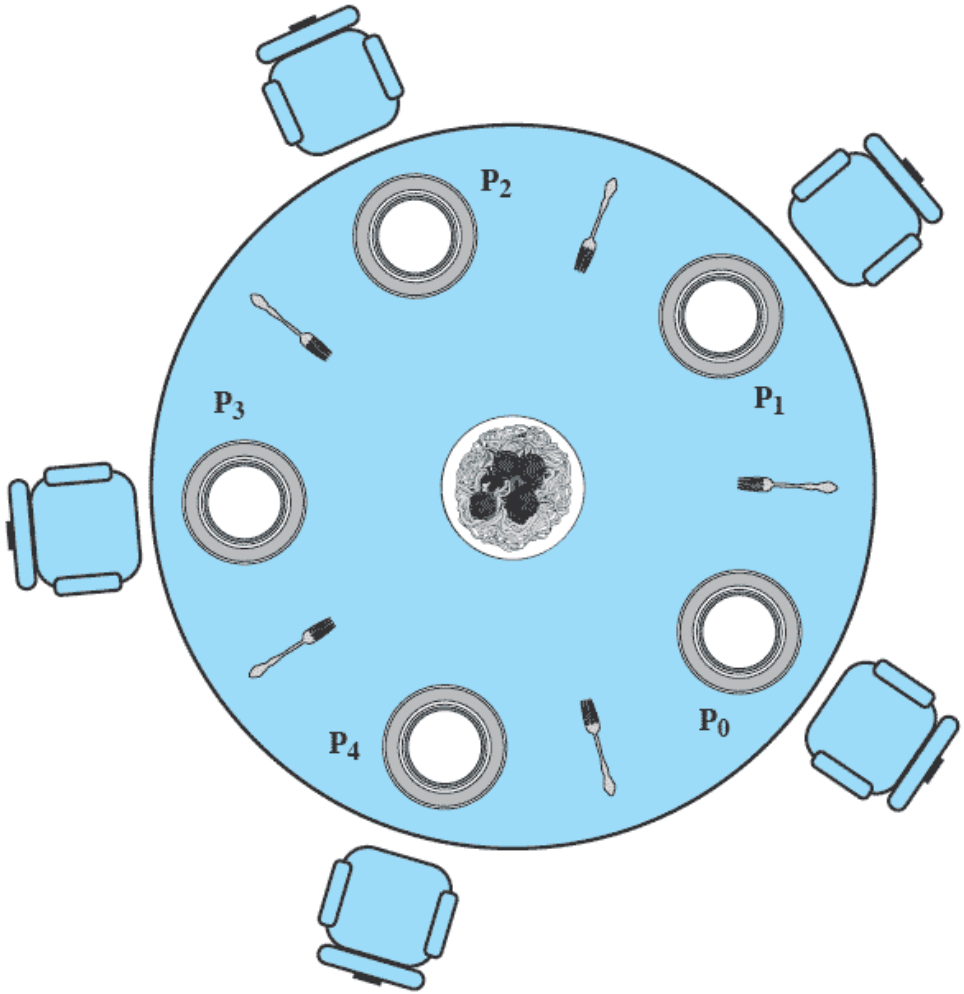


Figure 6.11 Dining Arrangement for Philosophers

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
```

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
```

6.7 UNIX的并发机制 ★★★★★

- 管道
- 消息
- 共享内存
- 信号量
- 信号

提问

1. 银行家算法属于避免死锁的一种方法。

A. 正确　　B. 错误

2. 剥夺资源法属于（）。

A. 死锁检测　　B. 死锁预防　　C. 死锁避免　　D. 互斥

3. 互斥是发生死锁的必要条件之一，因此可以通过破坏这个条件来避免死锁。

A. 正确　　B. 错误

4. 不安全状态是指进程发生了死锁。

A. 正确　　B. 错误

5. 进程启动拒绝策略属于（）。

A. 死锁检测　　B. 死锁预防　　C. 死锁避免　　D. 互斥

6. 下面属于可消耗资源的是（）。

A. 中断　　B. I/O通道　　C. 打印机　　D. 磁盘

作业

- 复习题 6.3， 6.7
- 习题 6.5， 6.6， 6.15