

# 第四章 死锁与饥饿

## 4.1 死锁

## 4.2 死锁的预防

## 4.3 死锁的避免

## 4.4 死锁的检测与恢复

## 4.5 死锁的忽略

## 4.6 饥饿



## 引例

- 设系统中只有一台打印机和一台读卡机，它们被进程P和进程Q共用。
- 进程P和Q各自对资源的申请使用情况如下：

P: 申请读卡机  
申请打印机

...

释放读卡机  
释放打印机

Q: 申请打印机  
申请读卡机

...

释放打印机  
释放读卡机



## 考虑下面的执行序列

P: 申请读卡机

Q: 申请打印机

P: 申请打印机

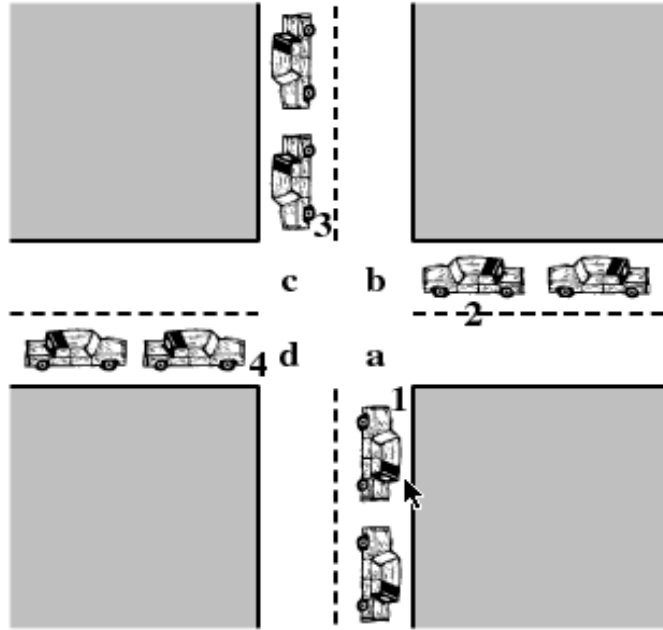
Q: 申请读卡机

...

- 结果：进程P占有读卡机，但未申请到打印机，需要等待；进程Q占有打印机，但等待读卡机。最终，P和Q都无法运行下去，彼此等待对方释放自己所需的资源。



# 交通实例



(a) Deadlock possible



## 独木桥实例

- 过一条独木桥，过桥人都只能向前进并不后退，那么当两个人从桥两端同时上桥，就会在桥中间相遇。这样两个人就只能在桥上相对而立，谁也过不去。



## 4.1 死锁

**死锁：**如是指在一个进程集合中的每一个进程，都在等待只能由该组进程中的其他进程才能引发的事件，从而无限期僵持下去的一种局面。如果没有外力作用，它们都将无法再向前推进。



## 4.1.2 产生死锁的原因

1.竞争资源

2.进程推进顺序不当。



## 1. 竞争资源引起进程死锁

主存、CPU

是否允许被抢占：可抢占性和不可抢占性资源

磁带机、  
打印机





## 2. 进程推进顺序不当引起死锁

- 联合进程图（Joint Progress Diagram）记录共享资源的多个进程的执行进展。

### 进程P

...

申请A（读卡机）

...

申请B

...

释放A

...

释放B

...

### 进程Q

...

申请B（打印机）

...

申请A

...

释放B

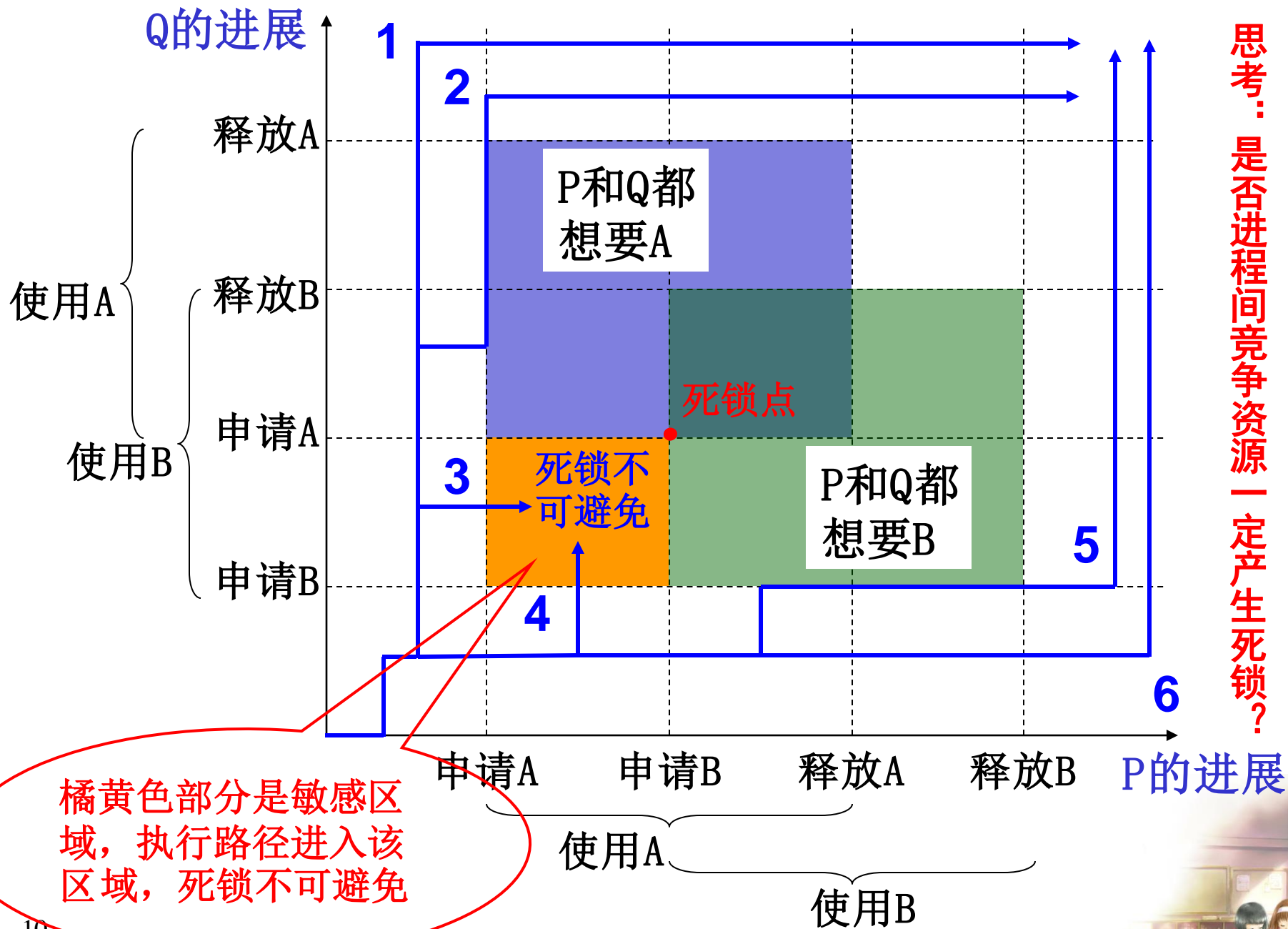
...

释放A

...

特点：一个进程互斥使用的资源不只一个。每个进程都需要独占两个资源一段时间。





### 进程P

...

获得A

...

释放A

...

获得B

...

释放B

...

### 进程Q

...

获得B

...

获得A

...

释放B

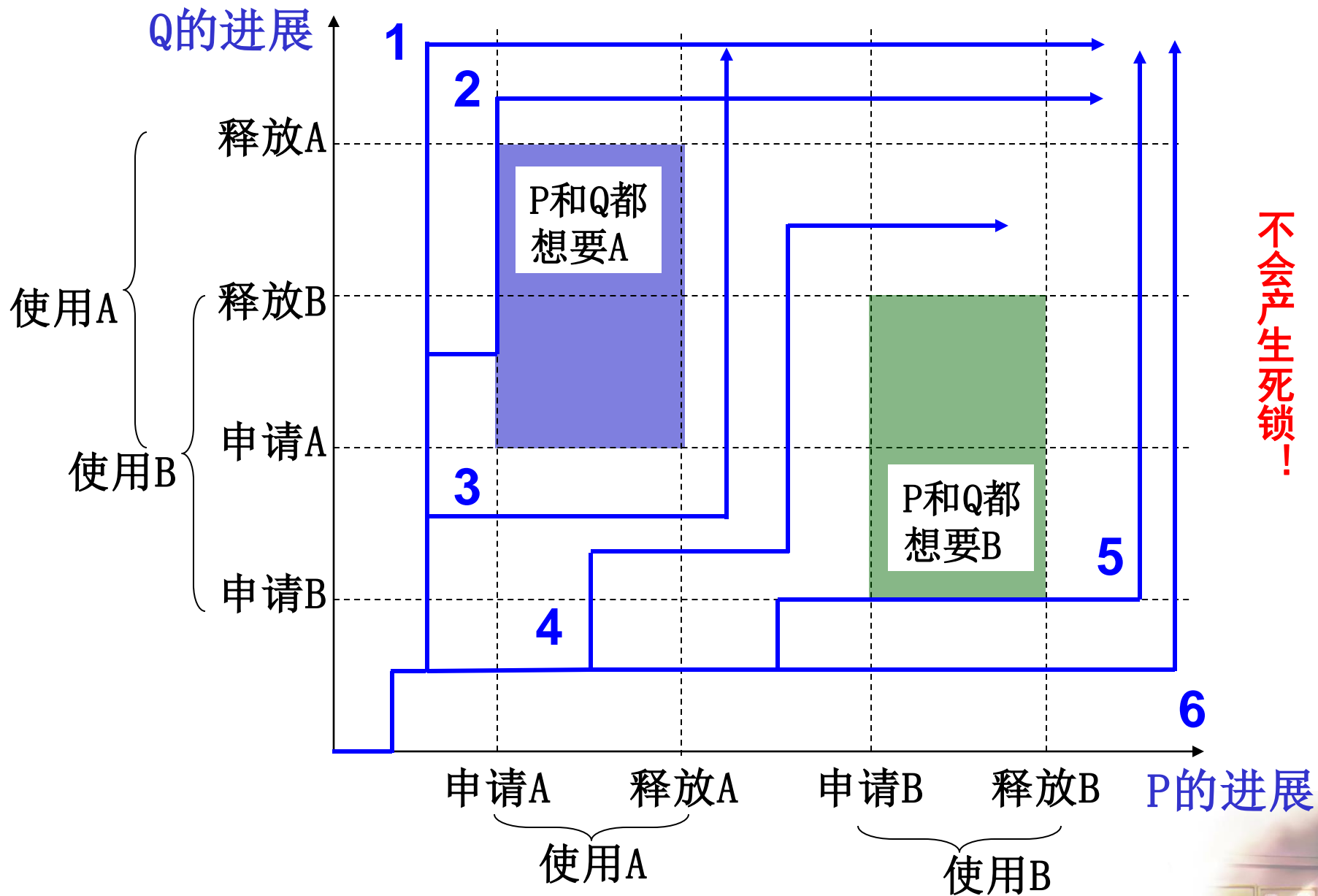
...

释放A

...

- 竞争资源，未必产生死锁。
- 是否产生死锁，还取决于动态执行和应用程序细节。





### 4.1.3 产生死锁的必要条件

**(1)互斥条件：**进程对分配到的资源进行排它性使用。


**(2)请求和保持条件：**进程已经保持了至少一个资源，但又提出了新的资源要求，而该资源又被其他进程占有，请求进程阻塞，但对已经获得的资源不放。


**(3)不剥夺（不可抢占）条件：**进程已获得的资源，使用完之前不能被剥夺，只能用完自己释放。


**(4)环路等待条件：**发生死锁时，必然存在进程——资源的环形链。



## 4.1.4 处理死锁的基本方法

**(1) 预防死锁：** 设置某些限制条件，破坏四个必要条件中的一个或几个。**优点：** 容易实现。**缺点：** 系统资源利用率和吞吐量降低。 

**(2) 避免死锁：** 在资源的动态分配过程用某种方法防止系统进入不安全状态。**优点：** 较弱限制条件可获得较高系统资源利用率和吞吐量。**缺点：** 有一定实现难度。 

**(3) 检测和解除死锁：** 预先不采取任何限制，也不检查系统是否已进入不安全区，通过设置检测机构，检测出死锁后设法解除它。如撤消或挂起一些进程，回收一些资源。 

14 **(4) 忽略死锁：** 认为发生死锁的可能性很低，完全忽略死锁。 



## 4.2 死锁的预防

### 1. 破坏“请求和保持”条件

系统要求所有进程一次性申请所需的全部资源，只要有一种资源要求不能满足，即使是已有的其它各资源，也全部不分配给该进程，而让其等待。

**优点：**简单、易于实现且很安全。

**缺点：**资源严重浪费；进程延迟运行。



### 2. 破坏“不可抢占”条件

进程在需要资源时才提出请求，一个已经保持了某些资源的进程，再提出新的资源要求而不能立即得到满足时，必须释放已经保持的所有资源。

**优点：**摒弃了“不可抢占”条件。

**缺点：**实现复杂，代价大；延长了进程的周转时间，增加系统开销，降低系统吞吐量。





### 3. 破坏“循环等待”条件

系统将所有资源按类型进行线性排队，并赋予不同的序号。所有进程对资源的请求必须严格按资源序号递增的次序提出。

**优点：**资源利用率和系统吞吐量较前两者改善。

**缺点：**限制了新设备类型的增加；进程使用各类资源的顺序与系统规定的顺序不同而造成资源的浪费；限制了用户简单、自主地编程。

## 4.3 死锁的避免

### 4.3.1 安全状态与不安全状态

#### 1. 安全状态

所谓**安全状态**，是指系统能按某种进程顺序( $P_1, P_2, \dots, P_n$ )(称  $\langle P_1, P_2, \dots, P_n \rangle$  序列为**安全序列**)，来为每个进程 $P_i$ 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地地完成。如果系统**无法**找到这样一个安全序列，则称系统处于**不安全状态**。

并非所有不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进而进入死锁状态。



### 2. 安全状态之例

假定系统中三个进程 $P_1$ 、 $P_2$ 和 $P_3$ ，共有8台磁带机。进程 $P_1$ 总共要求6台打印机， $P_2$ 和 $P_3$ 分别要求4台和7台。假设在 $T_0$ 时刻，进程 $P_1$ 、 $P_2$ 和 $P_3$ 已分别获得2台、2台和1台磁带机，尚有3台空闲未分配，如下表所示：

|       | 最大需求 | 已分配 | 尚需 | 可用资源数 |
|-------|------|-----|----|-------|
| $P_1$ | 6    | 2   | 4  | 3     |
| $P_2$ | 4    | 2   | 2  |       |
| $P_3$ | 7    | 1   | 6  |       |



### 3. 由安全状态向不安全状态的转换

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在 $T_0$ 时刻以后， $P_1$ 又请求2台磁带机，若此时系统把剩余3台中的2台分配给 $P_1$ ，则系统便进入不安全状态。

|       | 最大需求 | 已分配 | 尚需 | 可用资源数 |
|-------|------|-----|----|-------|
| $P_1$ | 6    | 4   | 2  | 1     |
| $P_2$ | 4    | 2   | 2  |       |
| $P_3$ | 7    | 1   | 6  |       |



## 4.3.2 利用银行家算法避免死锁

- 银行家算法思想——死锁避免策略：
  - 该策略用于确保系统中进程和资源总是处于安全状态。
  - 当进程请求一组资源时，**假设**同意该请求，从而改变了系统的状态，然后确定其结果是否还处于安全状态。
    - 如果是，同意该请求。
    - 如果不是，阻塞该进程直到同意该请求后系统状态仍是安全的。



- 系统：  $n$  个进程、  $m$  种资源
- **Resource**= $R=(R_1, R_2, \dots, R_m)$  系统中每种资源的总量
- **Available**= $V=(V_1, V_2, \dots, V_m)$  系统中可用资源数目
- **Claim**= $C = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$   $C_{ij}$ =进程*i*对资源*j*的最大需求  
**Max**
- **Allocation**= $A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}$

$$R_j = V_j + \sum_{i=1}^n A_{ij}$$

$$C_{ij} \leq R_j$$

$$A_{ij} \leq C_{ij}$$

$A_{ij}$ =系统分配给进程*i*的资源*j*的情况



## 1. 银行家算法举例

- 假定系统中有五个进程 {P1, P2, P3, P4} 和三类资源 {A, B, C}，各种资源的数量分别为9、3、6，在 $T_0$ 时刻的资源分配情况如下表所示。

|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 1         | 1 | 2 |
| P2 | 3   | 1 | 4 | 2          | 1 | 1 | 1    | 0 | 3 |           |   |   |
| P3 | 6   | 1 | 3 | 5          | 1 | 1 | 1    | 0 | 2 |           |   |   |
| P4 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |



- 问：

(1)该系统是否处于安全状态？

(2)P1提出资源请求Request (2, 2, 1)

(3)P2提出资源请求Request (1, 0, 1)

(4)P3提出资源请求Request (1, 0, 1)

按照避免死锁的要求，系统能否满足上述资源请求？





(1) 判断 $T_0$ 时刻的安全性

- 判断 $T_0$ 时刻是否是安全状态？

|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 1         | 1 | 2 |
| P2 | 3   | 1 | 4 | 2          | 1 | 1 | 1    | 0 | 3 |           |   |   |
| P3 | 6   | 1 | 3 | 5          | 1 | 1 | 1    | 0 | 2 |           |   |   |
| P4 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |

- 利用安全性算法检测。



## 第四章 死锁与饥饿

|    | <b>Max</b> |   |   | <b>Allocation</b> |   |   | <b>Need</b> |   |   | <b>Available</b> |   |   |
|----|------------|---|---|-------------------|---|---|-------------|---|---|------------------|---|---|
|    | A          | B | C | A                 | B | C | A           | B | C | A                | B | C |
| P1 | 3          | 2 | 2 | 1                 | 0 | 0 | 2           | 2 | 2 | 1                | 1 | 2 |
| P2 | 3          | 1 | 4 | 2                 | 1 | 1 | 1           | 0 | 3 |                  |   |   |
| P3 | 6          | 1 | 3 | 5                 | 1 | 1 | 1           | 0 | 2 |                  |   |   |
| P4 | 4          | 2 | 2 | 0                 | 0 | 2 | 4           | 2 | 0 |                  |   |   |

(1)T0  
时刻  
是安  
全的

|    | <b>Work</b> |   |   | <b>Need</b> |   |   | <b>Allocation</b> |   |   | <b>Work+ Allocation</b> |   |   | <b>Finish</b> |
|----|-------------|---|---|-------------|---|---|-------------------|---|---|-------------------------|---|---|---------------|
|    | A           | B | C | A           | B | C | A                 | B | C | A                       | B | C |               |
| P3 | 1           | 1 | 2 | 1           | 0 | 2 | 5                 | 1 | 1 | 6                       | 2 | 3 | true          |
| P1 | 6           | 2 | 3 | 2           | 2 | 2 | 1                 | 0 | 0 | 7                       | 2 | 3 | true          |
| P2 | 7           | 2 | 3 | 1           | 0 | 3 | 2                 | 1 | 1 | 9                       | 3 | 4 | true          |
| P4 | 9           | 3 | 4 | 4           | 2 | 0 | 0                 | 0 | 2 | 9                       | 3 | 6 | true          |



## (2) P1提出请求Request (2, 2, 1) ?

|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 1         | 1 | 2 |
| P2 | 3   | 1 | 4 | 2          | 1 | 1 | 1    | 0 | 3 |           |   |   |
| P3 | 6   | 1 | 3 | 5          | 1 | 1 | 1    | 0 | 2 |           |   |   |
| P4 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |

- ①  $\text{Request}_1(2, 2, 1) \leq \text{Need}_1(2, 2, 2)$
- ②  $\text{Request}_1(2, 2, 1) \not\leq \text{Available}(1, 1, 2)$
- 即系统没有足够的可用资源供其使用。因此，进程P1的资源请求无法满足，故进程P1产生阻塞。



## (3) P2提出请求Request (1, 0, 1) ?

|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 0         | 1 | 1 |
| P2 | 3   | 1 | 4 | 3          | 1 | 2 | 0    | 0 | 2 |           |   |   |
| P2 | 6   | 1 | 3 | 5          | 1 | 1 | 1    | 0 | 2 |           |   |   |
| P3 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |

- ①  $\text{Request}_2(1, 0, 1) \leq \text{Need}_2(1, 0, 3)$
- ②  $\text{Request}_2(1, 0, 1) \leq \text{Available}(1, 1, 2)$
- ③  $\text{Available} = (1, 1, 2) - (1, 0, 1) = (0, 1, 1)$   
 $\text{Need}_2 = (1, 0, 3) - (1, 0, 1) = (0, 0, 2)$   
 $\text{Allocation}_2 = (2, 1, 1) + (1, 0, 1) = (3, 1, 2)$
- 利用安全性算法检测状态是否安全。



- 在进行安全性算法检查时发现，可用资源  $Available[0, 1, 1]$  已不能满足任何进程的需要，故系统进入不安全状态。因此，进程  $P_2$  的资源请求无法满足，进程  $P_2$  产生阻塞。
- (4)  $P_3$  提出资源请求  $Request(1, 0, 1)$  能满足吗？



## (4) P3提出请求Request (1, 0, 1) ?

|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 0         | 1 | 1 |
| P2 | 3   | 1 | 4 | 2          | 1 | 1 | 1    | 0 | 3 |           |   |   |
| P2 | 6   | 1 | 3 | 6          | 1 | 2 | 0    | 0 | 1 |           |   |   |
| P3 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |

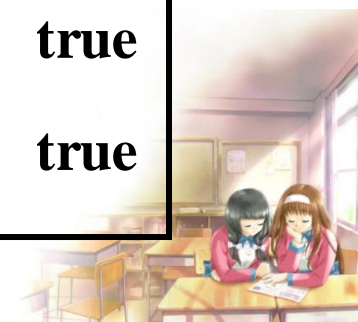
- ①  $\text{Request}_3(1, 0, 1) \leq \text{Need}_3(1, 0, 2)$
- ②  $\text{Request}_3(1, 0, 1) \leq \text{Available}(1, 1, 2)$
- ③  $\text{Available} = (1, 1, 2) - (1, 0, 1) = (0, 1, 1)$   
 $\text{Need}_3 = (1, 0, 2) - (1, 0, 1) = (0, 0, 1)$   
 $\text{Allocation}_3 = (5, 1, 1) + (1, 0, 1) = (6, 1, 2)$
- 利用安全性算法检测状态是否安全。



|    | Max |   |   | Allocation |   |   | Need |   |   | Available |   |   |
|----|-----|---|---|------------|---|---|------|---|---|-----------|---|---|
|    | A   | B | C | A          | B | C | A    | B | C | A         | B | C |
| P1 | 3   | 2 | 2 | 1          | 0 | 0 | 2    | 2 | 2 | 0         | 1 | 1 |
| P2 | 3   | 1 | 4 | 2          | 1 | 1 | 1    | 0 | 3 |           |   |   |
| P2 | 6   | 1 | 3 | 6          | 1 | 2 | 0    | 0 | 1 |           |   |   |
| P3 | 4   | 2 | 2 | 0          | 0 | 2 | 4    | 2 | 0 |           |   |   |

系统处于安全状态

|    | Work |   |   | Need |   |   | Allocation |   |   | Work+ Allocation |   |   | Finish |
|----|------|---|---|------|---|---|------------|---|---|------------------|---|---|--------|
|    | A    | B | C | A    | B | C | A          | B | C | A                | B | C |        |
| P3 | 0    | 1 | 1 | 0    | 0 | 1 | 6          | 1 | 2 | 6                | 2 | 3 | true   |
| P1 | 6    | 2 | 3 | 2    | 2 | 2 | 1          | 0 | 0 | 7                | 2 | 3 | true   |
| P2 | 7    | 2 | 3 | 1    | 0 | 3 | 2          | 1 | 1 | 9                | 3 | 4 | true   |
| P4 | 9    | 3 | 4 | 4    | 2 | 0 | 0          | 0 | 2 | 9                | 3 | 6 | true   |



- 进行安全性算法检查，可以找到一个安全序列 {P3, P1, P2, P4}。因此，系统处于安全状态，此时可以将进程P3所请求的资源分配给它。





### 2. 银行家算法中的数据结构

(1) **可利用资源向量Available**。这是一个含有 $m$ 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 $\text{Available}[j] = K$ ，则表示系统中现有 $R_j$ 类资源 $K$ 个。



(2) **最大需求矩阵Max**。这是一个 $n \times m$ 的矩阵，它定义了系统中 $n$ 个进程中的每一个进程对 $m$ 类资源的最大需求。如果 $\text{Max} [i,j] = K$ ，则表示进程 $i$ 需要 $R_j$ 类资源的最大数目为 $K$ 。

(3) **分配矩阵Allocation**。这也是一个 $n \times m$ 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 $\text{Allocation} [i,j] = K$ ，则表示进程 $i$ 当前已分得 $R_j$ 类资源的数目为 $K$ 。

(4) **需求矩阵Need**。这也是一个 $n \times m$ 的矩阵，用以表示每一个进程尚需的各类资源数。如果 $\text{Need} [i,j] = K$ ，则表示进程 $i$ 还需要 $R_j$ 类资源 $K$ 个，方能完成其任务。

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



### 3. 银行家算法

设 $\text{Request}_i$ 是进程 $P_i$ 的请求向量，如果 $\text{Request}_i[j] = K$ ，表示进程 $P_i$ 需要 $K$ 个 $R_j$ 类型的资源。当 $P_i$ 发出资源请求后，系统按下述步骤进行检查：

(1) 如果 $\text{Request}_i[j] \leq \text{Need}[i,j]$ ，便转向步骤2；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果 $\text{Request}_i[j] \leq \text{Available}[j]$ ，便转向步骤(3)；否则，表示尚无足够资源， $P_i$ 须等待。



(3) 系统试探着把资源分配给进程 $P_i$ ，并修改下面数据结构中的数值：

$$\text{Available } [j] = \text{Available } [j] - \text{Request}_i [j] ;$$

$$\text{Allocation } [i,j] = \text{Allocation } [i,j] + \text{Request}_i [j] ;$$

$$\text{Need } [i,j] = \text{Need } [i,j] - \text{Request}_i [j] ;$$

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程 $P_i$ ，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程 $P_i$ 等待。



### 4. 安全性算法

(1) 设置两个向量：① 工作向量**Work**：它表示系统可提供给进程继续运行所需的各类资源数目，它含有 $m$ 个元素，在执行安全算法开始时，**Work=Available**；② **Finish**：它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做**Finish [i] =false**；当有足够资源分配给进程时，再令**Finish [i] =true**。



(2) 从进程集合中找到一个能满足下述条件的进程:

①  $\text{Finish}[i] = \text{false}$ ; ②  $\text{Need}[i,j] \leq \text{Work}[j]$ ; 若找到, 执行步骤(3), 否则, 执行步骤(4)。

(3) 当进程 $P_i$ 获得资源后, 可顺利执行, 直至完成, 并释放出分配给它的资源, 故应执行:

$\text{Work}[j] = \text{Work}[j] + \text{Allocation}[i,j]$ ;

$\text{Finish}[i] = \text{true}$ ;

go to step 2;

(4) 如果所有进程的 $\text{Finish}[i] = \text{true}$ 都满足, 则表示系统处于安全状态; 否则, 系统处于不安全状态。



## 总结

- 预防死锁
  - 避免死锁
- } 事先施加限制条件，以防死锁发生。
- 两者的区别：
    - 预防死锁：破坏死锁的必要条件，施加的条件比较严格，可能会影响到进程的并发执行。
    - 避免死锁：资源动态分配，施加的限制条件较弱一些，有利于进程的并发执行。





## 总结

- 死锁避免策略并不能确切的预测死锁，仅仅是预料死锁的可能性并确保永远不会出现这种可能性。
- 死锁避免比死锁预防限制少，但使用中也有许多限制：
  - 必须事先声明每个进程请求的最大资源。
  - 考虑的进程必须是无关系的，执行的顺序必须没有任何同步的要求。
  - 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。





## 4.4 死锁的检测与解除

### 4.4.1 死锁的检测

#### 1. 资源分配图(Resource Allocation Graph)

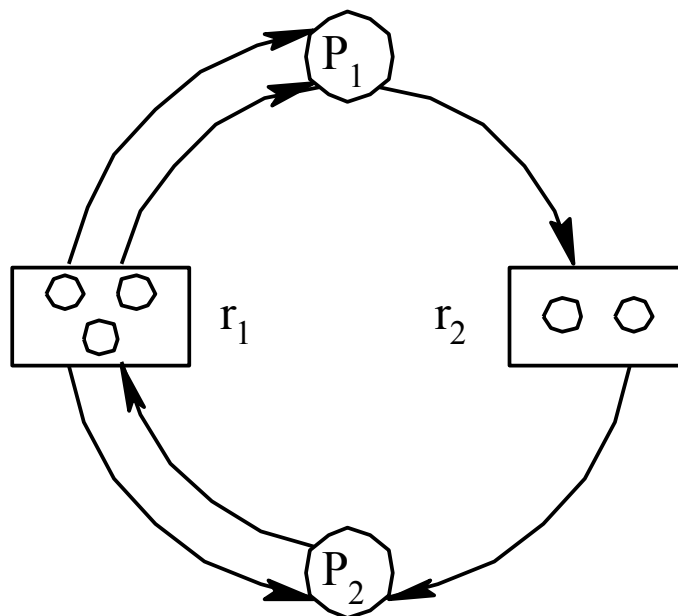


图 3-21 每类资源有多个时的情况



2. 死锁定理：S为死锁状态当且仅当S状态的资源分配图是不可完全简化的。

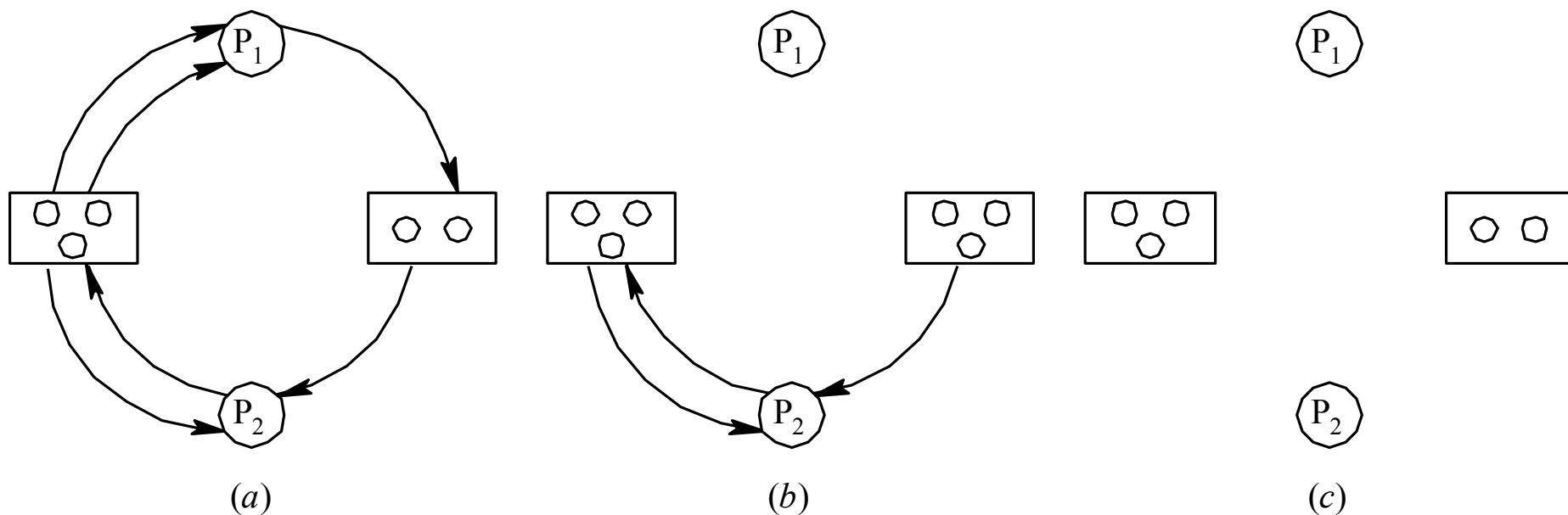


图 3-22 资源分配图的简化



### 3. 死锁检测算法

(1) 可利用资源向量Available，它表示了 $m$ 类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量Allocation=0)记入L表中，即 $L_i \cup L$ 。

(3) 从进程集合中找到一个 $\text{Request}_i \leq \text{Work}$ 的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量 $\text{Work} = \text{Work} + \text{Allocation}_i$ 。② 将它记入L表中。



(4) 若不能把所有进程都记入L表中, 便表明系统状态S的资源分配图是不可完全简化的。 因此, 该系统状态将发生死锁。  $Work = Available;$

$L = \{L_i | Allocation_i = 0 \cap Request_i = 0\}$

for( $i=1$ ;  $L_i \notin L$ ;  $i++$ )

{

if  $Request_i \leq Work$

{

$Work = Work + Allocation_i;$

$L_i \cup L;$

}

}

$deadlock = \bigcap (L = \{p_1, p_2, \dots, p_n\});$



## 4.4.2 死锁的解除

(1) 剥夺资源。

(2) 撤消进程。



## 4.5 死锁的忽略

鸵鸟算法：效率和正确性之间进行折中

鸵鸟算法用于表示对发生的死锁问题视而不见，它是处理死锁最简单的一种方法。如Unix和Windows。

一般来说，用户宁愿忍受系统偶然性故障带来的损失，也不愿因经常性进行死锁处理而牺牲系统的性能。



## 4.6 饥饿

饥饿：是指一个可以运行的进程尽管能继续执行，但被调度无限期地忽视，不能被调度执行的情况。

饥饿与死锁的不同：

死锁的进程都必定处于阻塞态，而饥饿进程不一定被阻塞，只是被无限期地拖延，未调度执行，即饥饿进程可以在就绪态。



## 第四章 小结

- 死锁原因，必要条件，处理方法；
- 死锁的预防；
- 安全状态；安全序列；
- 银行家算法（结合具体题目），理解方法；
- 死锁检测与解除；
- 死锁的忽略

