第6章 并发性：死锁和饥饿

（资源管理）

死锁的原理、预防、避免和检测

管理资源的策略和方法

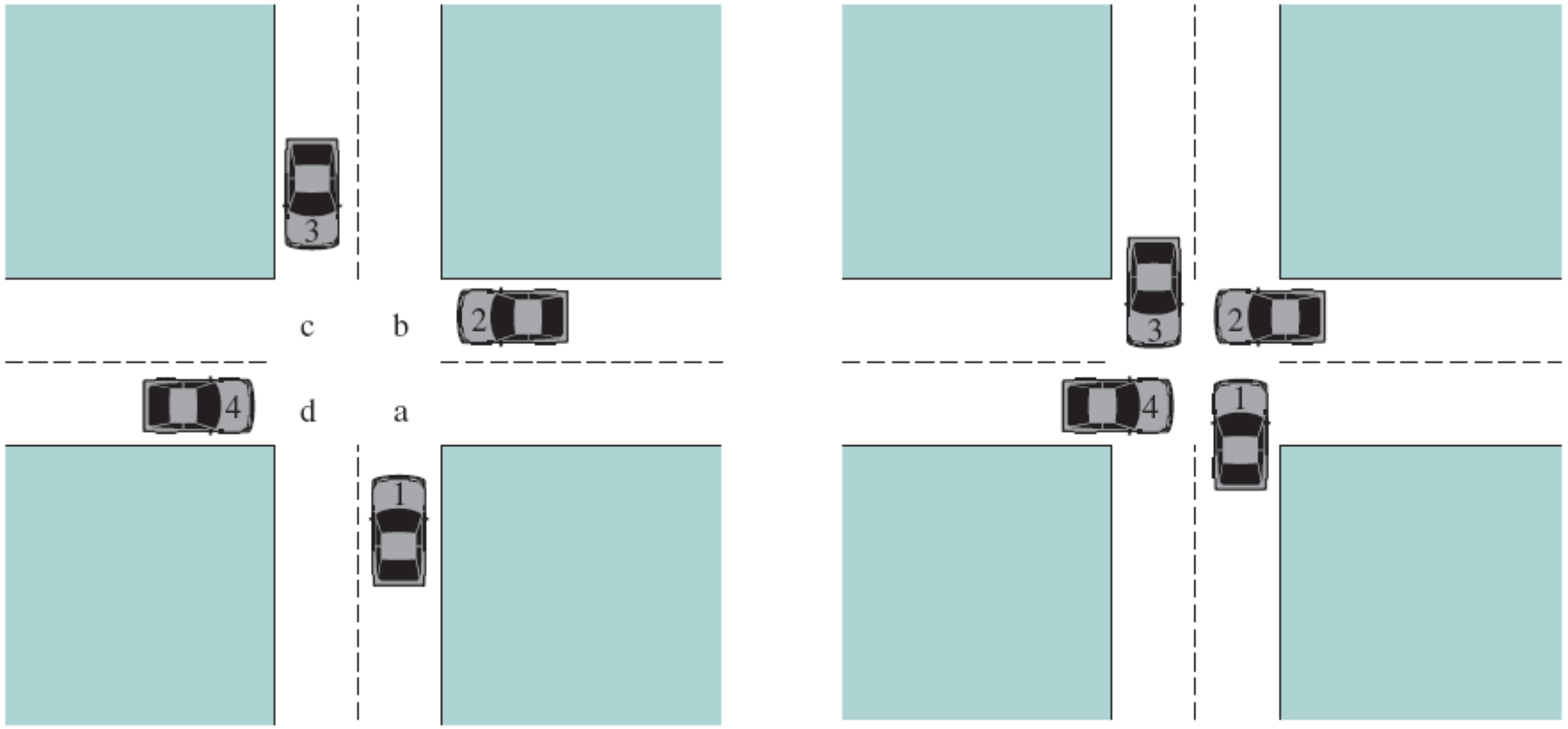
死锁的综合策略

哲学家就餐问题

第一讲 死锁

- 死锁原理
  - 死锁（deadlock）概念
  - 资源、资源分配图与死锁定理
  - 死锁的条件
- 死锁预防
- 死锁避免
  - 进程启动拒绝
  - 资源分配拒绝(银行家算法)
- 死锁检测与恢复
- 综合策略

6.1 死锁



(a) Deadlock possible
可能死锁

(b) Deadlock
死锁

Illustration of Deadlock
死锁的图示

6.1 死锁

6.1.1 死锁的概念

- 死锁(deadlock)：一个进程集合中的每个进程都在等待只能由该集合中的其他进程才能引发的事件（释放占有资源/进行某项操作）
- 死锁是多个进程因竞争资源且推进顺序不合理而造成的一种僵局，若无外力作用，这些进程将永远不能再向前推进

联合进程图(joint progress diagram)

- 记录进程共享资源的历史和分析死锁的工具
- 二个进程的联合进程图是一个二维网格，三个进程的联合进程图是一个三维网格
- 四种区域
  - 安全区域（白色）
  - 敏感区域（fatal region，致命区域）（黑色）
    - 死锁区域（黑色）
    - 不可达区域（彩色）

可能死锁的进程举例

进程P：

…

获得A

…

获得B

…

释放A

…

释放B

…

进程Q：

…

获得B

…

获得A

…

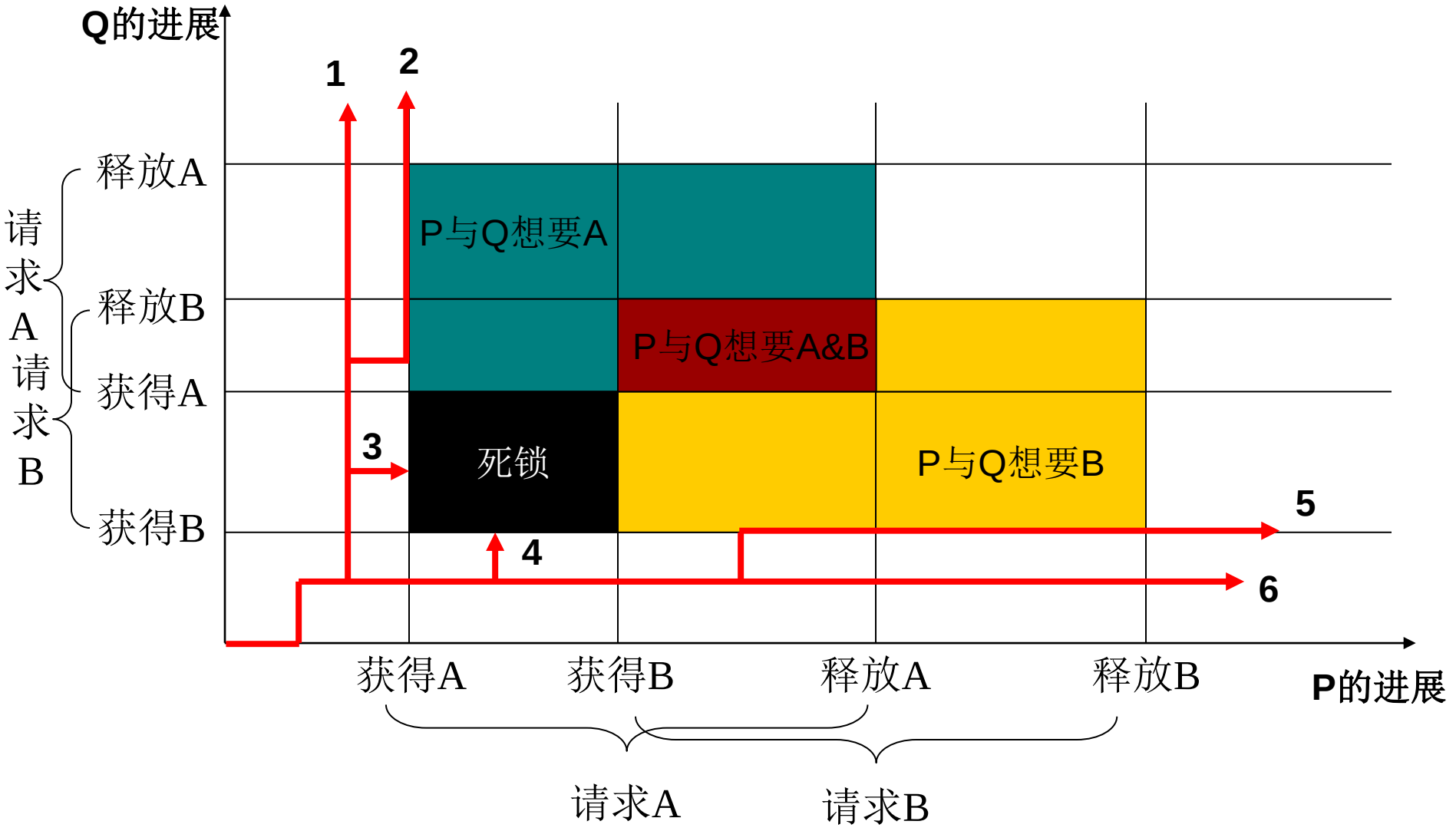
释放B

…

释放A

…

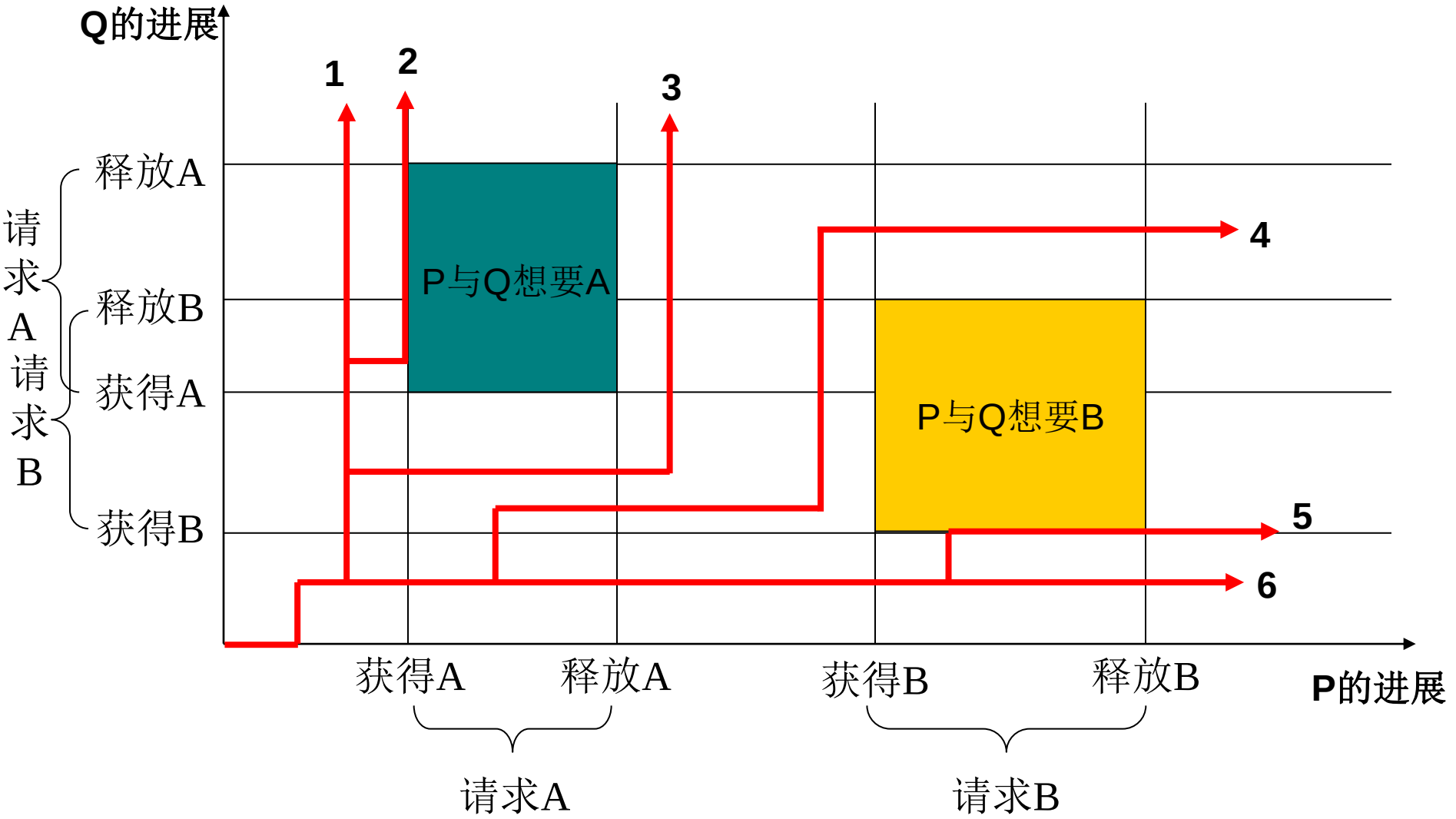
联合进程图——死锁举例


Q的进展
1
2
释放A
请求A
释放B
获得A
请求B
P与Q想要A
P与Q想要A&B
3
死锁
P与Q想要B
5
4
6
获得A
获得B
释放A
释放B
P的进展
请求A
请求B

不会死锁的进程举例

- 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
  - …

联合进程图——无死锁的例子



6.1.1可重用资源

- 资源分类
  - **可重用资源：**（如图书）与可消费资源（如食物）
  - **可剥夺资源：**（如处理机、内存等）与不可剥夺资源（如打印机、表、队列、信号量等）
  - **永久性资源与临时性资源**

6.1.1可重用资源

- 可重用资源（reusable resource）
  - 一次只能供一个进程安全地使用，且不会由于使用而耗尽（消失）的资源
  - 可以获得、使用和释放，能够再次被使用
  - 例子：处理器、I/O通道、主存和辅存、设备，文件、数据库、信号量等数据结构

涉及可重用资源死锁的例子

步骤 Step	进程P操作 Process P Action
p_0	Request (D) 请求
p_1	Lock (D) 加锁
p_2	Request (T) 请求
p_3	Lock (T) 加锁
p_4	Perform function
p_5	Unlock (D) 解锁
p_6	Unlock (T) 解锁

D：文件A

T：文件B

执行功能

步骤 Step	进程Q操作 Process Q Action
q_0	Request (T) 请求
q_1	Lock (T) 加锁
q_2	Request (D) 请求
q_3	Lock (D) 加锁
q_4	Perform function
q_5	Unlock (T) 解锁
q_6	Unlock (D) 解锁

Example of Two Processes Competing for Reusable Resources

两个进程竞争可重用资源的例子

破坏环路等待条件

6.1.2 消耗性资源

- 消耗性资源（consumable resource）
  - 可以创建(生产)并且可以销耗掉的资源
  - 数目没有限制，当一个进程得到一个可消费资源时，这个资源就不再存在了
  - 例子：中断、信号、消息、I/O缓冲区中的信息

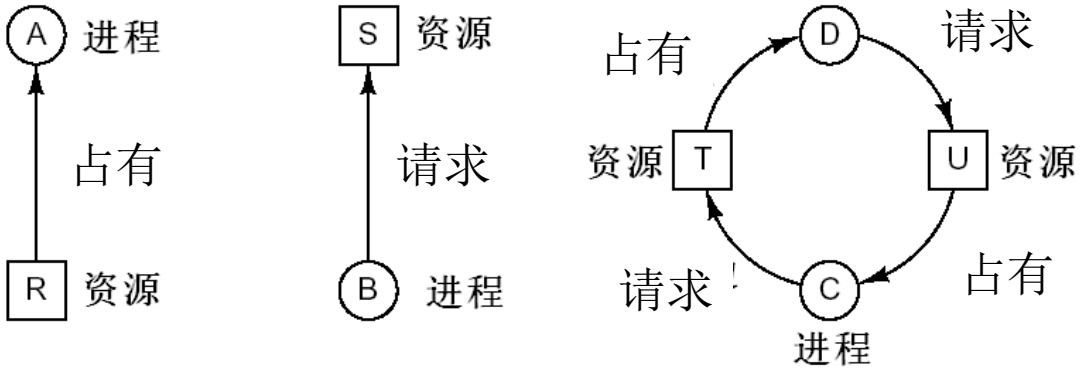
涉及消耗性资源死锁的例子（消息）

```
P1:
…
receive(P2);
…
send(P2, M1);
…
```

```
P2:
…
receive(P1);
…
send(P1, M2);
…
```

6.1.3 资源分配(resource allocation graph)

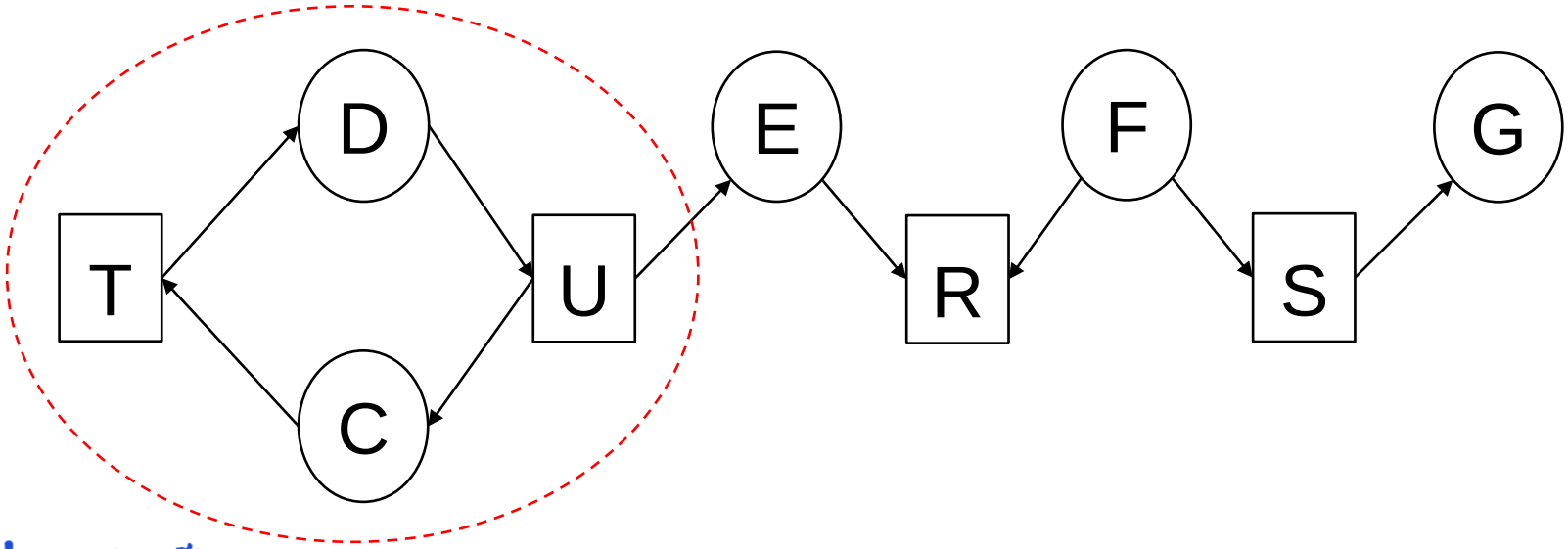
- 用有向图描述的进程死锁模型
- 表示法



  - 资源类（资源的不同类型）：方框
  - 资源实例（存在于每个资源类中）：黑圆点
  - 进程：圆圈中加进程名
  - 占有（分配边）：资源实例指向进程的一条有向边
  - 请求（申请边）：进程指向资源类的一条有向边

死锁定理

- 如果资源分配图中没有环路，则系统中没有死锁，如果图中存在环路则系统中**可能存在**死锁
- 如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件



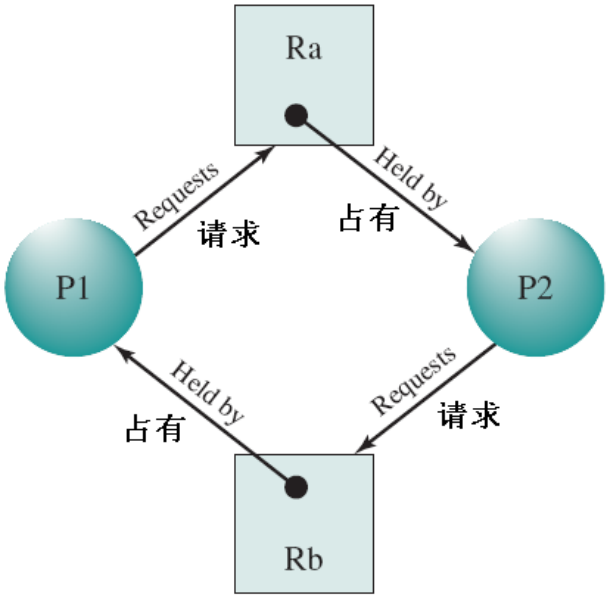
资源分配图例

(a) Resource is requested

资源被请求

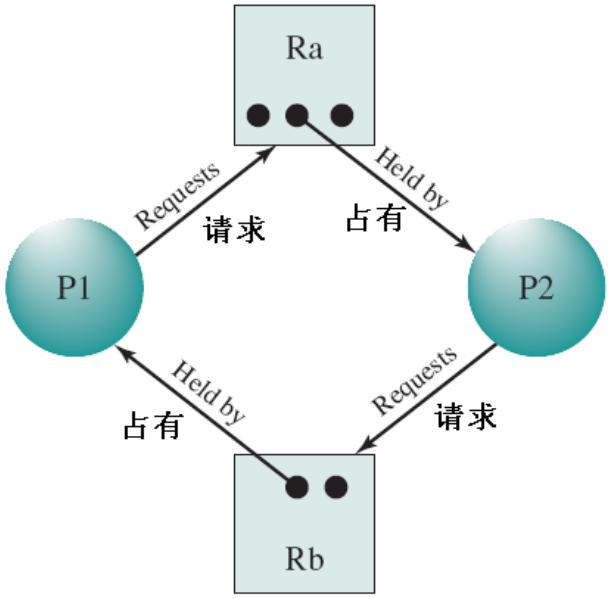
(b) Resource is held

资源被占有



(c) Circular wait

循环等待



(d) No deadlock

无死锁

Examples of Resource Allocation Graphs

资源分配图的例子

6.1.4 死锁的三个必要条件

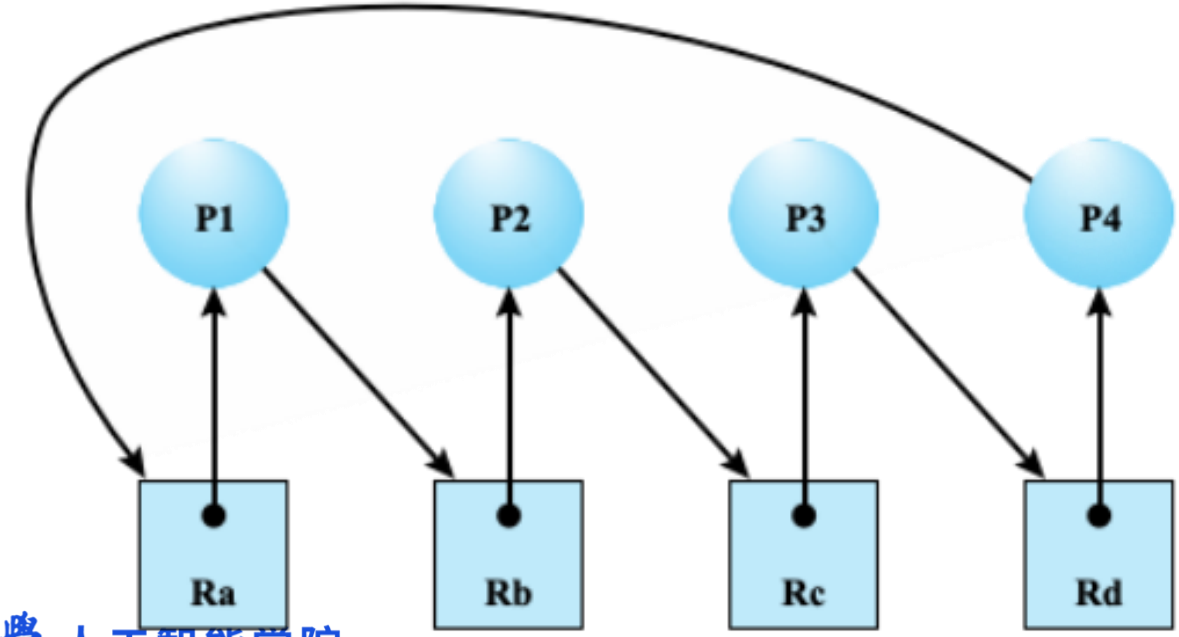
- 互斥
  - 进程对所分配到的资源进行排他性使用：在一段时间内某资源只由一个进程占有
  - 若其他进程要求该资源，要求者需阻塞，直至占有该资源的进程用毕释放

6.1.4 死锁的三个必要条件

- 占有且等待
  - 进程已占有至少一项资源，并提出新的资源要求
  - 若该资源已被其它进程占有，则请求进程阻塞，同时对已经获得的其它资源保持不放
- 不抢占（non-preemption）
  - 进程已获得的资源在未使用完之前不能被抢占，只能在使用后自行释放

死锁的条件

- 环路等待条件
  - 资源分配图中存在环路，即进程集合{P0, P1, P2, …, Pn}(n>=2)中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源



6.2 预防死锁现象发生

- 四类方法
  - **鸵鸟算法**：对死锁视而不见

  - **预防死锁**：通过破坏产生死锁的四个条件中的一个或多个条件，保证不会发生死锁
  - **避免死锁**
  - **检测死锁**

6.2.1 破坏互斥条件

- 方法：不直接操作资源或多个进程，而使用虚拟资源
- 适用条件
  - 资源的固有特性允许多个进程同时使用（如文件允许多个进程同时读）
  - 借助特殊技术允许多个进程同时使用（如打印机借助Spooling技术）
- 缺点
  - 不适用于绝大多数资源

6.2.2 破坏占有且等待条件

- 方法
  - 禁止已拥有资源的进程再申请其他资源
  - 要求所有进程在开始时一次性地申请在整个运行过程所需的全部资源
  - 申请资源时要先释放其占有资源后，再一次性申请所需全部资源

6.2.2 破坏占有且等待条件

- 优点
  - 简单、易于实现、安全
- 缺点
  - 进程延迟运行
  - 资源严重浪费

6.2.3 破坏不抢占条件

- 方法
  - 占有资源的进程提出新的资源请求，而不能立即满足时，必须释放已经占有的所有资源，以后重新申请
  - 操作系统抢占一个进程占有的资源，分配给其他进程
- 适用条件
  - 资源的状态可以很容易地保存和恢复(如CPU)
- 缺点

实现复杂、代价大，反复申请/释放资源、

系统开销大、降低系统吞吐量

6.2.4 破坏环路等待条件

- 方法
  - 要求每个进程任何时刻只能占有一个资源，如果要申请第二个则必须先释放第一个（不现实）
  - 对所有资源按类型进行线性排队，进程申请资源必须严格按资源序号递增的顺序（可避免循环等待）
- 缺点

排序困难，类型序号的安排只能考虑一般作业的情况，限制了用户简单、自主地编程

易造成资源的浪费（会不必要地拒绝对资源的访问）

可能低效（会使进程的执行速度变慢）

资源按序分配方法说明

- 例子：某多进程应用系统，有4个数据库文件db1、db2、db3和db4，每个进程都要打开其中几个文件，完成事务处理。
  - 每个进程，打开文件的顺序严格按按资源序号递增的顺序db1->db2->db3->db4，即必须先打开db1才能打开后面的文件，如此类推
  - 如果进程1要用到db1、db2和db4，则必须先打开db1,再打开db2、最后打开db4，而非随意打开

其他内容

- 死锁避免
  - 拒绝启动进程策略
  - 拒绝资源请求策略——银行家算法
- 死锁检测和恢复
- 综合策略

6.3 死锁避免(deadlock avoidance)

- 不需事先采取限制措施破坏产生死锁的必要条件，而是在**资源的动态分配**过程中，采用某种策略防止系统进入不安全状态，从而避免发生死锁
- 两种死锁避免的方法：
  - 不启动其资源请求会导致死锁的进程
  - 不允许会导致死锁的进程资源请求

6.3.1 进程启动拒绝模型

n个进程和m种资源

系统资源(Resource)总量向量**R**=(R_1,R_2, … , R_m)， R_j为第j种资源的总数

系统当前可用(aVailable)资源总量向量**V**=(V_1,V_2, … , V_m)， V_j为第j种资源的剩余数

进程-资源需求(Claim)矩阵**C**，C_{ij}=进程i对资源j的请求数

进程-资源分配(Allocation)矩阵**A**，A_{ij}=当前已经分配给进程i的资源j数

进程启动拒绝模型

- 系统正常状态下，模型有如下性质：
  - $R_j=V_j+(A_{1j}+A_{2j}+\ldots+A_{nj})$，对所有的$j$=1,2,…,$m$
  - $C_{ij}\le R_j$，对所有的i=1,2,…,n，j=1,2,…,m
  - $A_{ij}\le C_{ij}$，对所有的i=1,2,…,n，j=1,2,…,m

$$\begin{pmatrix} C_{11},C_{12},\ \ldots\ ,C_{1m} \\ C_{21},C_{22},\ \ldots\ ,C_{2m} \\ \ldots \\ C_{n1},C_{n2},\ \ldots\ ,C_{nm} \end{pmatrix} \quad \begin{pmatrix} A_{11},A_{12},\ \ldots\ ,A_{1m} \\ A_{21},A_{22},\ \ldots\ ,A_{2m} \\ \ldots \\ A_{n1},A_{n2},\ \ldots\ ,A_{nm} \end{pmatrix}$$

进程启动拒绝模型

- 进程启动拒绝策略（新进程P_{n+1}）
  - 若$R_j \geq C_{n+1,j} + (C_{1j} + C_{2j} + \ldots + C_{nj})$， $j=1,2,\ldots,m$，则允许启动进程P_{n+1}
  - 否则系统不启动进程P_{n+1}
  - 不能保证资源的最优使用率，但能保证系统现有进程不会发生死锁

6.3.2 资源分配拒绝

Dijkstra于1965年提出资源分配拒绝策略（银行家算法）

- 模型
  - 系统状态：由资源总量向量**R**=(R_1,R_2, … , R_m)、系统可用资源总量向量**V**=(V_1,V_2, … , V_m)、进程-资源需求矩阵**C**和进程-资源分配矩阵**A**表示
  - **安全状态**：至少存在一个执行时序，使当前所有进程都能运行到结束状态
  - **不安全状态**：不存在一个执行时序，使当前所有进程都能运行到结束状态

安全状态举例

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
0	1	1

可用向量V

(a) 初始状态

安全状态举例（续）

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

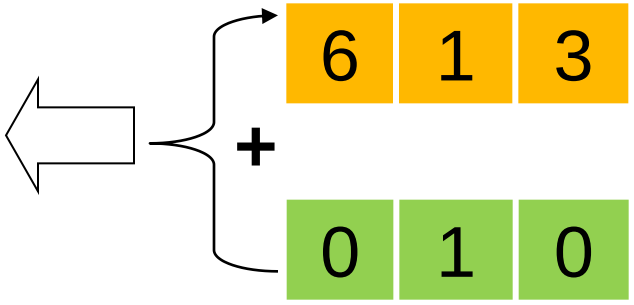
矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
6	2	3

可用向量V



(b) P2运行到完成

安全状态举例（续）

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

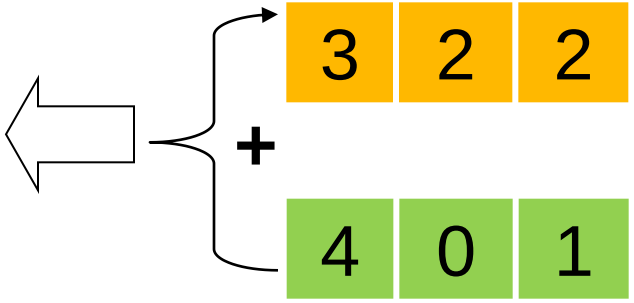
矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
7	2	3

可用向量V



(c) P1运行到完成

安全状态举例（续）

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

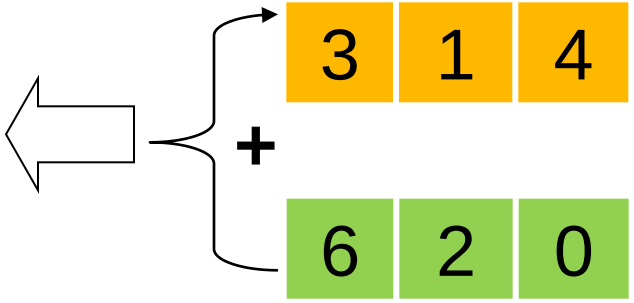
矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
9	3	4

可用向量V



(d) P3运行到完成

安全状态举例（续）

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

需求矩阵C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

分配矩阵A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	0

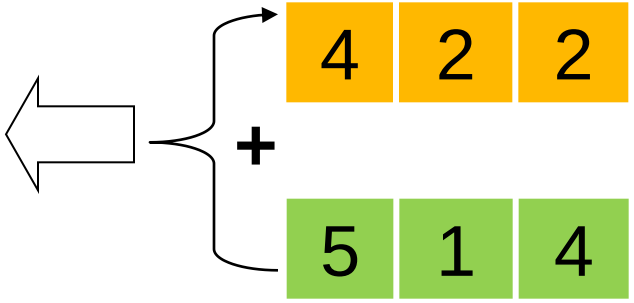
矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
9	3	6

可用向量V



(e) P4运行到完成

不安全状态举例

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
1	1	2

可用向量V

(a) 初始状态

不安全状态举例（续）

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

需求矩阵C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

分配矩阵A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

矩阵C-A

R1	R2	R3
9	3	6

资源向量R

R1	R2	R3
0	1	1

可用向量V

(b) P1请求1个R1&1个R3

资源分配拒绝策略

- 只要系统处于安全状态，必定不会进入死锁状态
- 不安全状态不一定是死锁状态，但不能保证不会进入死锁状态
- 死锁避免的实质：如何避免系统进入不安全状态
  - 如果一个新进程的资源请求会导致不安全状态， 则拒绝启动这个进程
  - 如果满足一个进程新提出的一项资源请求后，会导致不安全状态，则拒绝分配资源给这个进程

银行家算法

全局数据结构

```
struct state {
   int resource[m];
   int available[m];
   int claim[n][m];
   int alloc[n][m];
}S, X;
```

资源分配算法

```
if (S.alloc[i][*]+request(*)>claim[i,*])
    return ERROR; /*总申请量>需求 → 出错*/
else if (request(*)>S.available[*]) /*请求>可用*/
    return SUSPEND_PROCESS; /*挂起进程*/
else {/*定义临时新状态*/
     X.alloc[i][*]= S.alloc[i][*]+ request(*);
     X.available[*]=S.available[*]- request(*); }
if safe(X) {/*安全：定义新状态*/
   S.alloc[i][*]= X.alloc[i][*];
   S.available[*]=X.available[*];}
else/*不安全：不分配资源*/
   return SUSPEND_PROCESS; /*挂起进程*/
```

死锁避免分析

- 优势
  - 比死锁预防限制少
  - 无须死锁检测中的资源剥夺和进程重启
- 不足
  - 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的，即它们执行的顺序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

银行家算法-安全状态判定

```
boolean safe(state s) {
   int curruntavai[m];
   process_rest[number of processes];
   curruntavai[*]=s.available[*];
   rest={all process};
   possible=true;
   while (possible) {
        <寻找一进程Pk使：claim[k][*]-alloc[k][*]<=curruntavai[*]; >
         if (found) {
              curruntavai[*] = curruntavai[*]+alloc[k][*];
              rest=rest-{Pk};
          } else possible=false;
    }
     return (rest==null);
}
```

6.4 死锁检测

- 没有任何预先限制措施（死锁预防）
- 资源分配时不检查系统是否会进入不安全状态（死锁避免）
- 系统可能出现死锁
- 系统检测是否出现死锁
- 死锁发生时进行恢复

6.4.1 死锁检测时机与算法

- 检测时机
  - 资源申请时（早期检测，算法相对简单，但处理器时间消耗大）
  - 周期性检测，或死锁“好像”已经发生（如CPU使用率降低到一定阈值时）

6.4.1 死锁检测时机与算法

- 检测方法
  - 单个资源实例：检测资源分配图中是否存在环路（依据——死锁定理）
  - 多个资源实例：类似银行家算法的安全检查

6.4.2 恢复

- **抢占法**

连续抢占资源直到不再存在死锁

- **回退法**

每个死锁进程回滚(rollback)到某些检查点(checkpoint),并且重新启动所有进程（死锁可能重现）

6.4.2 恢复

- **杀死进程法**
  - 杀死所有死锁进程（最常用）
  - 或连续杀死死锁进程直到不再存在死锁
  - 或杀死一个非死锁进程

选择杀死死锁进程或抢占资源时

要考虑代价！

操作系统检测、预防和避免死锁的方法

原则	资源分配策略	不同的方案	主要优点	主要缺点
预防	保守，预提交(undercommits)资源	一次性请求所有资源	■对执行单一突发行为的进程有效 ■不需抢占	■低效 ■延时进程的初始化 ■进程必须知道未来的资源请求
		抢占	■便于用于状态易保存和恢复的资源	■过多的不必要抢占
		资源排序	■通过编译时检测可实施 ■由于问题已在系统设计时解决，不需运行时再计算	■不允许增加资源请求
避免	位于检测和预防中间	操纵以发现至少一条安全路径	■不需抢占	■操作系统须知道未来的资源请求 ■进程可能被长期阻塞
检测	非常自由；尽可能地准许请求的资源	周期性地调用以测试死锁	■不会延时进程的初始化 ■易于在线处理	■丢失固有抢占

6.5 一种综合的死锁策略

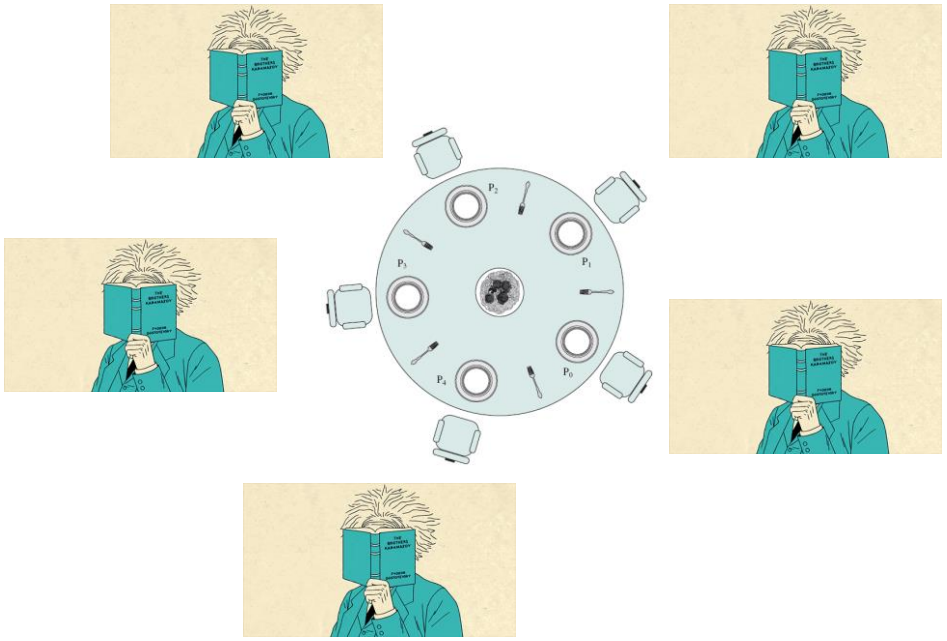
- 把资源分为不同资源类
  - 可交换空间：进程交换所用外存中的存储块
  - 进程资源：可分配的设备，如磁带设备和文件
  - 内存：可按页或段分配
  - 内部资源：I/O通道

6.5 一种综合的死锁策略

- 为预防在资源类之间由于循环等待产生死锁，可使用**线性排序策略**
- 在一个资源类中，使用该类资源最适合的算法

6.6 哲学家就餐问题

- 5个哲学家，每天思考、吃意大利面
- 必须用2把叉子吃面，但每人只有1把叉子
- 饥饿：某些哲学家长期无法用餐



6.6.1 使用信号量的解决方案A

```
semaphore fork[5]={1,1,1,1,1};
int i;
void philosopher(int i) {
   while (true)  {
      think(); // 非临界区
      P(fork[i]);
      P(fork[(i+1) mod 5]);
      eat();
      V(fork[(i+1) mod 5]);
      V(fork[i]);
    }
}
```

```
void main() {
    parbegin(
       philosopher(0),
       philosopher(1),
       philosopher(2),
       philosopher(3),
       philosopher(4)
     )
}
```

可能产生死锁

导致全部饥饿

使用信号量的解决方案B

```
semaphore fork[5]={1,1,1,1,1};
semaphore room=4;
int i;
void philosopher(int i) {
   while (true)  {
      think();
      P(room);
      P(fork[i]);
      P(fork[(i+1) mod 5]);
      eat();
      V(fork[(i+1) mod 5]);
      V(fork[i]);
      V(room); }
}
```

```
void main() {
    parbegin(
       philosopher(0),
       philosopher(1),
       philosopher(2),
       philosopher(3),
       philosopher(4)
     )
}
```

每次只允许四个人进餐，

不会产生死锁和饥饿

6.6.2 使用管程的解决方案

```
monitor dining_controller;
cond ForkReady[5];
boolean fork[5]={true};
void get_forks(int pid) {
   int left=pid;
   int right=(++pid) mod 5;
   if (!fork[left])
        cwait(ForkReady[left]);
    fork[left]=false;
    if (!fork[right])
        cwait(ForkReady[right]);
    fork[right]=false;
}
```

```
void release_forks(int pid) {
   int left=pid;
   int right=(++pid) mod 5;
   if (empty(ForkReady[left])
        fork[left]=true;
   else
        csignal(ForkReady[left]);
   if (empty(ForkReady[right])
        fork[right]=true;
   else
        csignal(ForkReady[right]);
   }
}
```

条件变量表示等待叉子的队列

问答