操作系统

Operating system

胡燕

大连理工大学

内容纲要

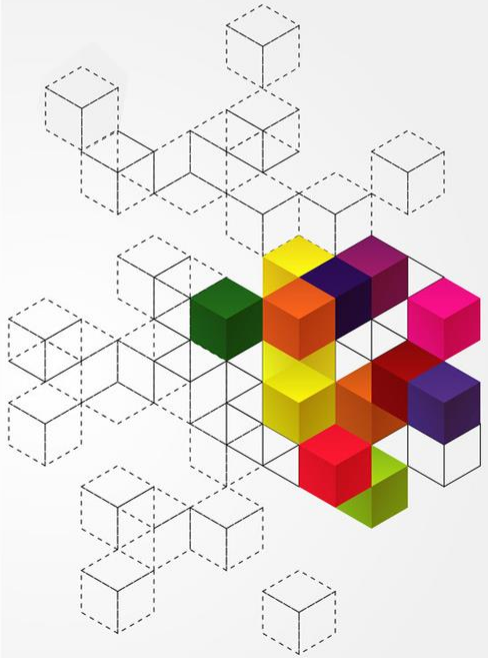
7.4 死锁处理方法

一、死锁处理方法概述

二、死锁预防

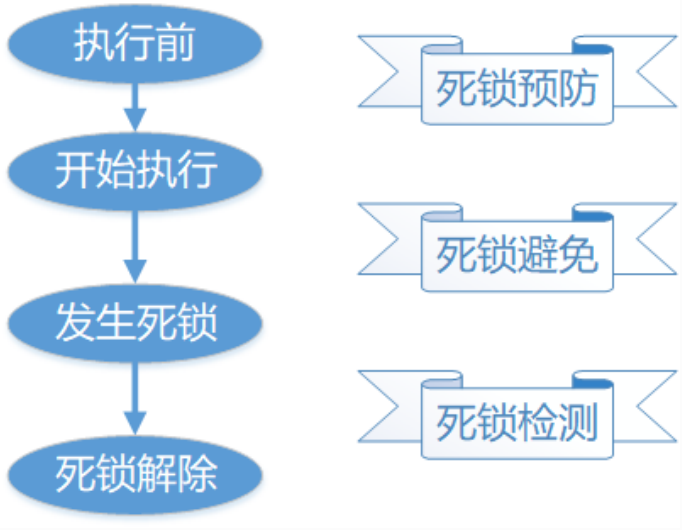
三、死锁避免

四、死锁检测

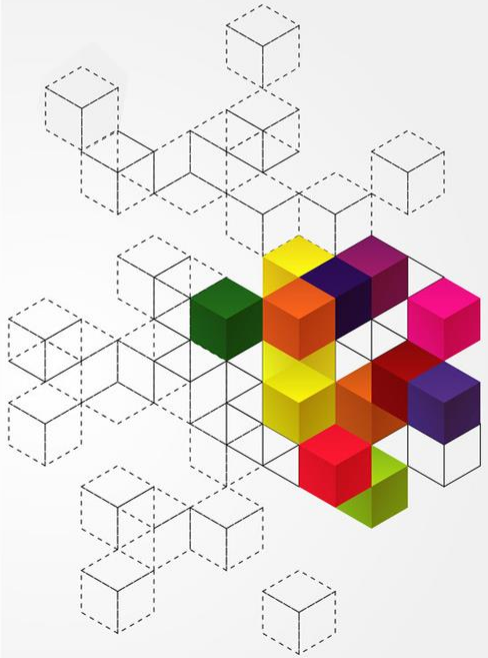


一、死锁处理方法概述

- **死锁处理方法一览图（按处理时机）**



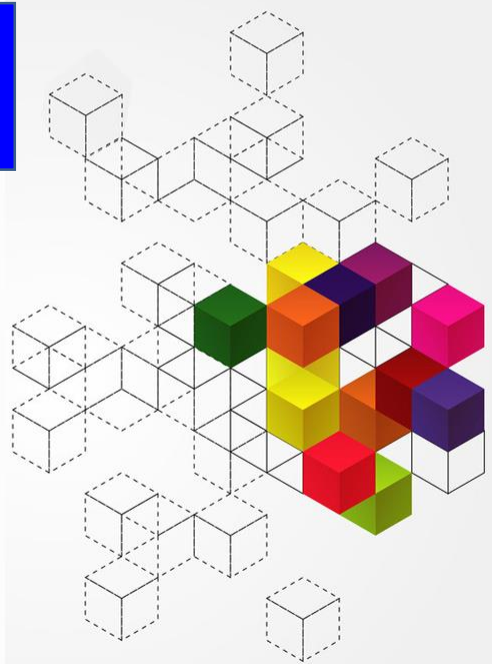
- 死锁预防
- 死锁避免
- 死锁检测（与处理）



二、死锁预防

- **死锁预防思路：事先制定资源使用规则，保证程序按照规则使用资源就必然不会发生死锁**

策略1： Each process tries to acquire all the resources it needs before the process runs



进程： 运行前申请所需全部资源

系统： 对于每个进程的资源申请，如果能够满足，则一次性全部分配；否则，进程等待

破坏“hold-and-wait”条件

缺点：

- 资源利用效率低
- 由于系统很难同时满足多个进程的一次性资源需求，可能出现大量进程等待现象

二、死锁预防

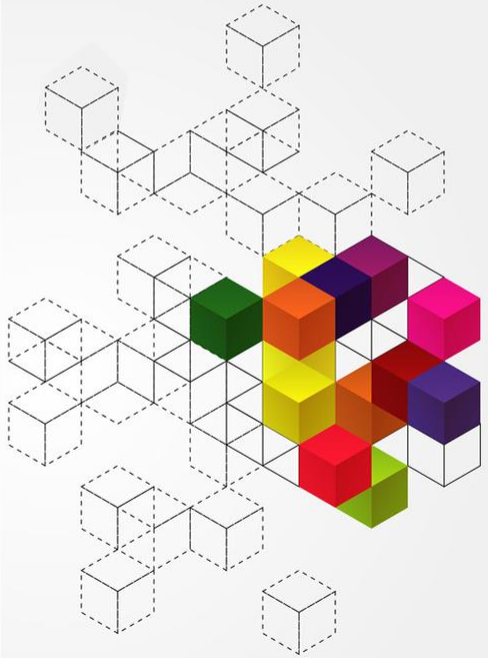
- **死锁预防思路：实现制定资源使用规则，保证程序按照规则使用资源就必然不会发生死锁**

策略2： **对资源进行编号，约定进程按照编号大小顺序对资源进行分配**

资源集：R={r1,r2,...,rn}

函数：F：R→N（为资源定一个级别）

进程pi可以申请资源rj中的实例⇔∀rl,pi占有rl, F(rl)<F(rj)



二、死锁预防

- **死锁预防思路：实现制定资源使用规则，保证程序按照规则使用资源就必然不会发生死锁**

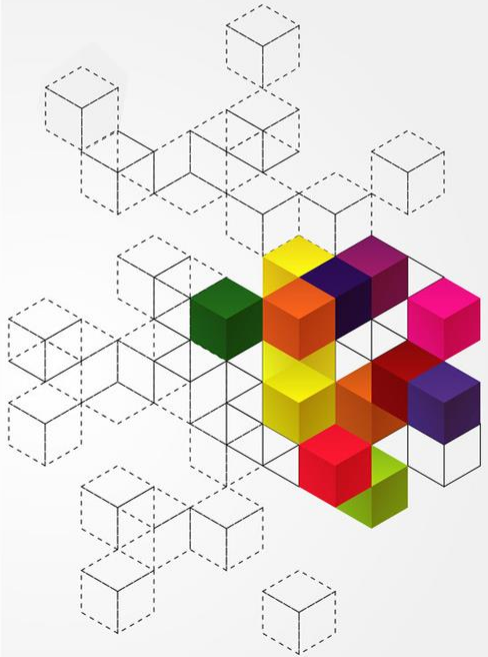
策略2： **对资源进行编号，约定进程按照编号大小顺序对资源进行分配**

例如：R={scanner,tape,printer}

F(scanner)=1; F(tape)=2; F(printer)=3;

要求：进程代码中申请资源必须按照先申请scanner，再申请tape，最后申请printer的原则进行

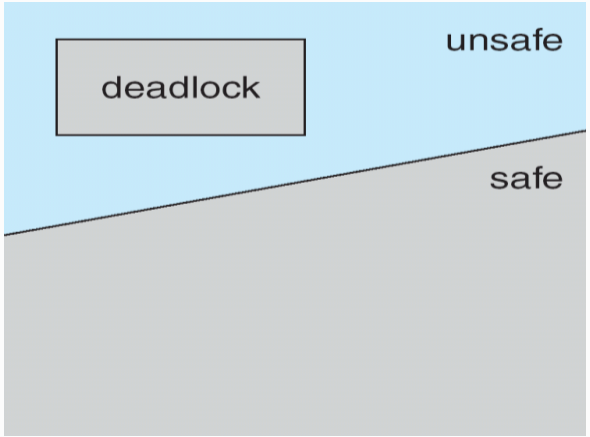
破坏“循环等待”条件：成功预防死锁

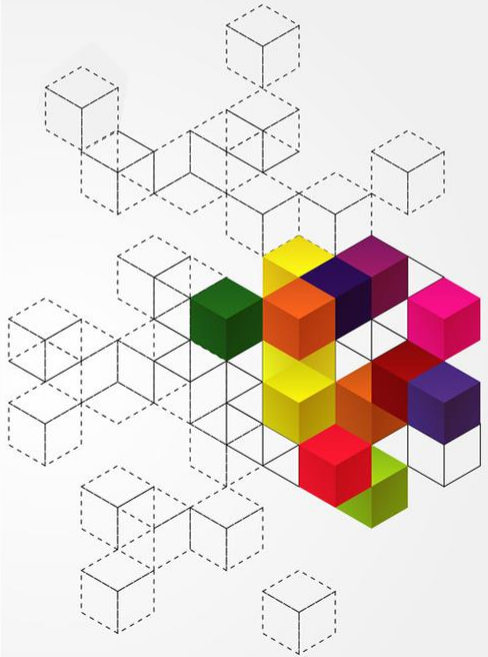


三、死锁避免

2.死锁避免（Deadlock Prevention）

- 保证死锁永远不会进入有死锁风险的状态
- 安全状态：一定不会发生死锁
- 不安全状态：存在发生死锁的可能
- 死锁避免思想：确保系统不会进入不安全状态





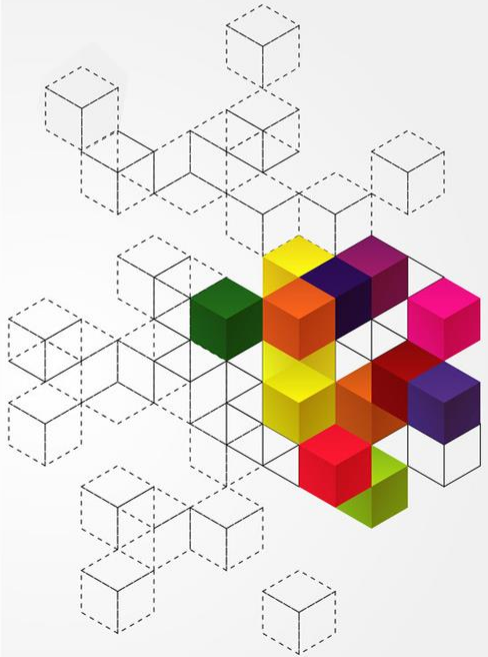
四、死锁检测

3.死锁检测(Deadlock Detection)

- 不采取任何预防或避免死锁的措施，在死锁发生时再进行检测并决定如何处理

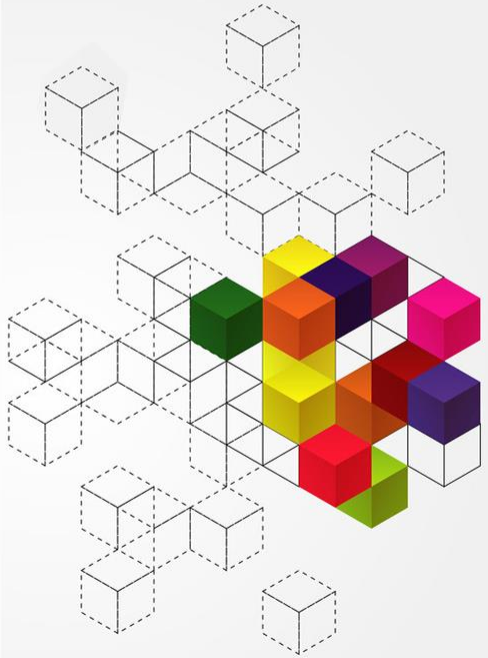
- 解决方案：
- （1）不处理，忽略，鸵鸟机制
- （2）检测，并解除

鸵鸟机制被多数操作系统采用，
原因是死锁检测代价很高



本讲小结

- 死锁处理方法概述
- 死锁预防
- 死锁避免
- 死锁检测



内容纲要

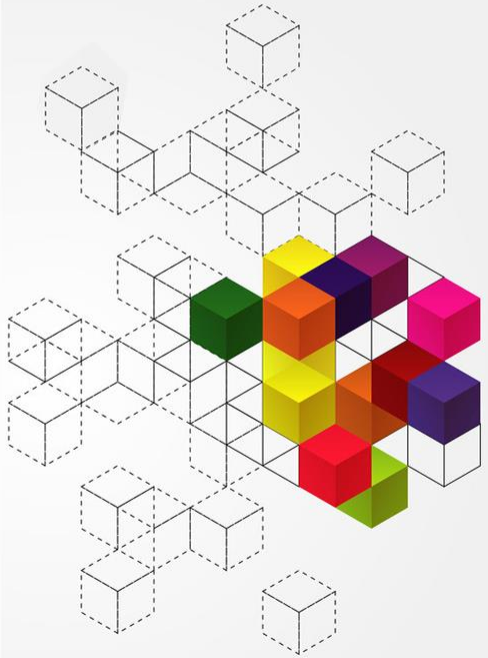
7.5 死锁避免

一、死锁避免概念

二、单资源实例条件下的死锁避免

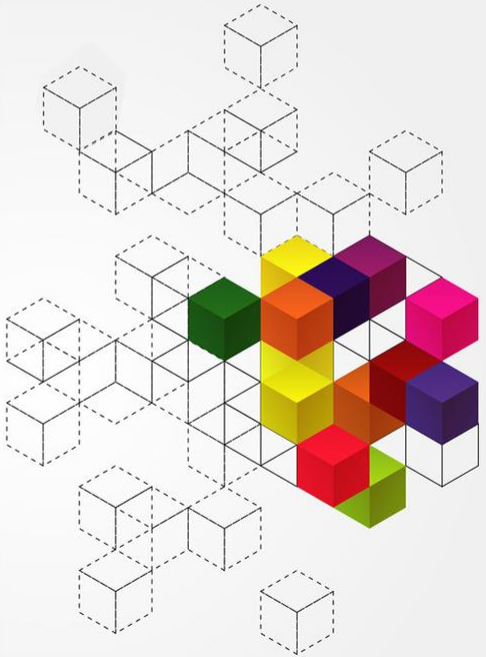
三、多实例的死锁避免算法

四、银行家算法实例解析



一、死锁避免概念

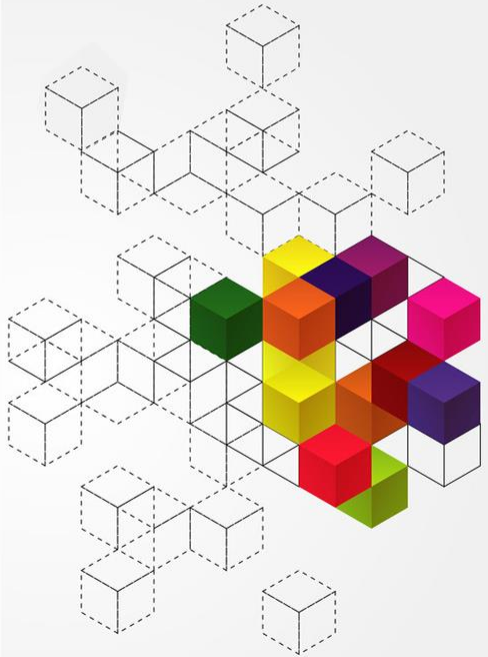
- 死锁避免（Deadlock Avoidance）
  - 在程序运行起来后，对每次资源分配请求进行系统审核，通过拒绝为不安全的资源请求进行分配，来达到避免死锁风险的目的，这类方法被称为死锁避免方法
  - 方法的核心要素：安全性判定
  - 关键概念：安全序列



二、单资源实例条件下的死锁避免

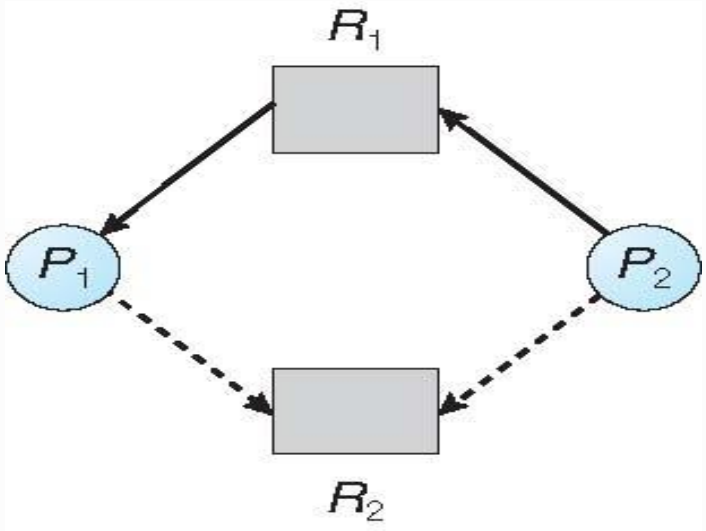
使用资源分配图

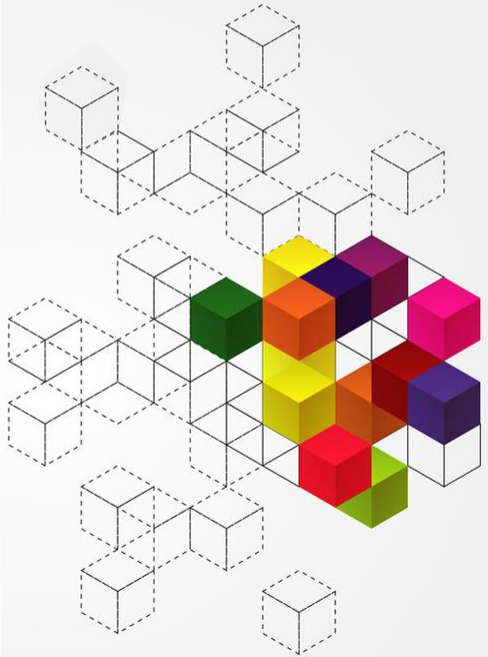
- 在图添加另一种类型的边，表示进程未来可能要提起的申请边，称为Claim edge
- 如果进程已经开始申请资源，那么相应的Claim edge会转变为Request edge



二、单资源实例条件下的死锁避免

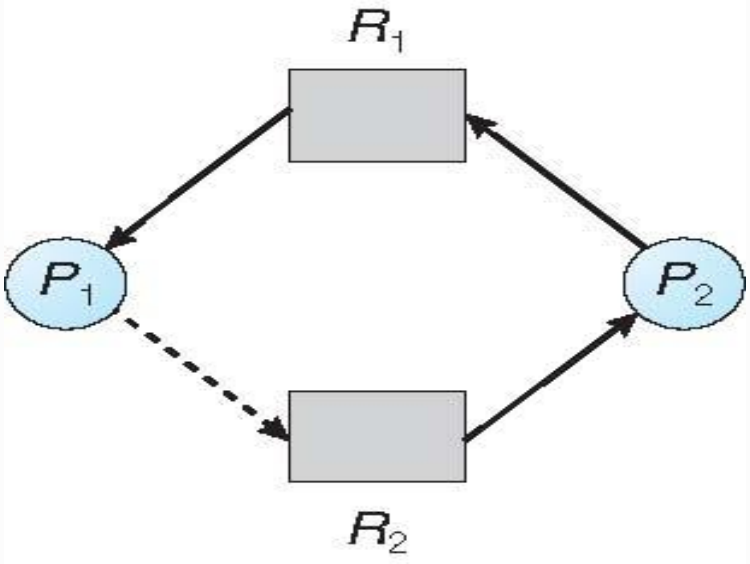
- 资源分配图示例



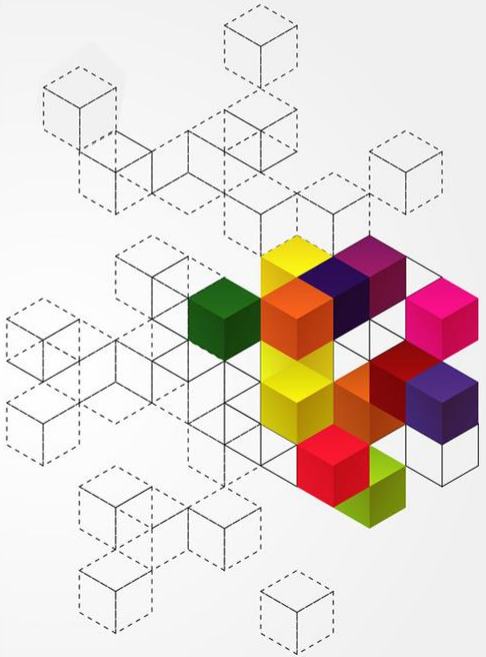


二、单资源实例条件下的死锁避免

- 资源分配图示例



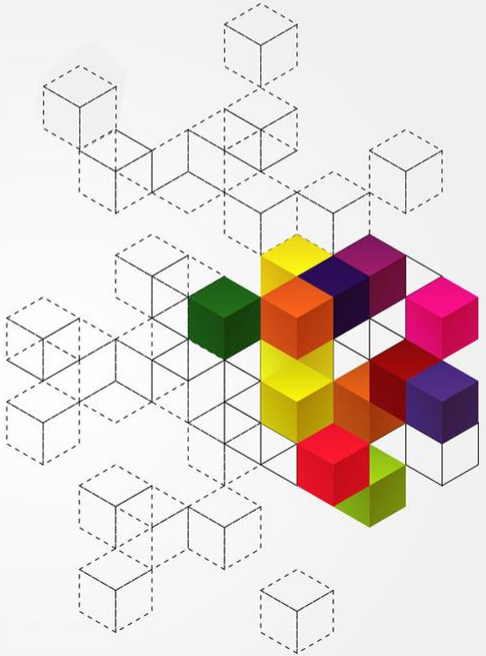
此时，如果允许P1发起对R2的申请
则死锁发生



三、多实例的死锁避免算法

银行家算法

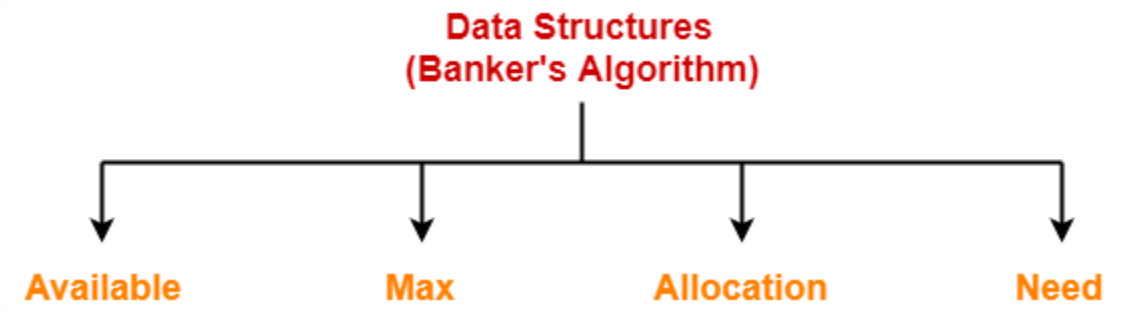
- 多资源实例条件下进行死锁避免，要比单资源实例的情形略复杂
- 主要在检测每次分配是否会使得系统进入不安全状态的方法有所不同

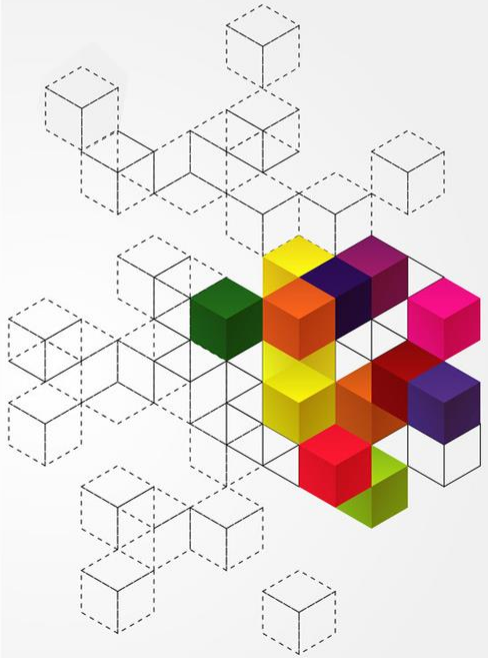


三、多实例的死锁避免算法

银行家算法的基本数据结构

- Available: array[1..m]of integer;　//系统可用资源
- Max: array[1..n,1..m]of integer;　//进程最大需求
- Allocation: array[1..n,1..m]of integer; //当前分配
- Need: array[1..n,1..m]of integer;　//尚需资源
- Request: array[1..n,1..m]of integer;　//当前请求

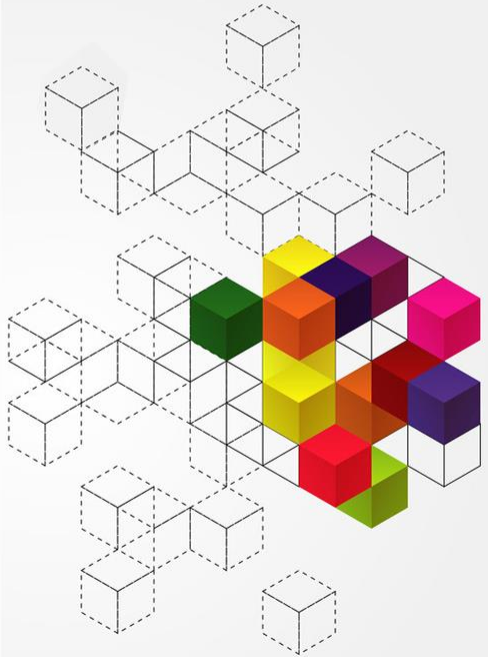




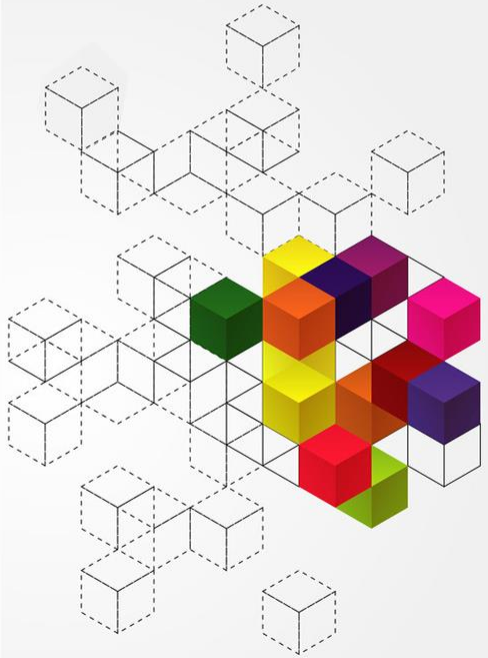
三、多实例的死锁避免算法

银行家算法的初始化

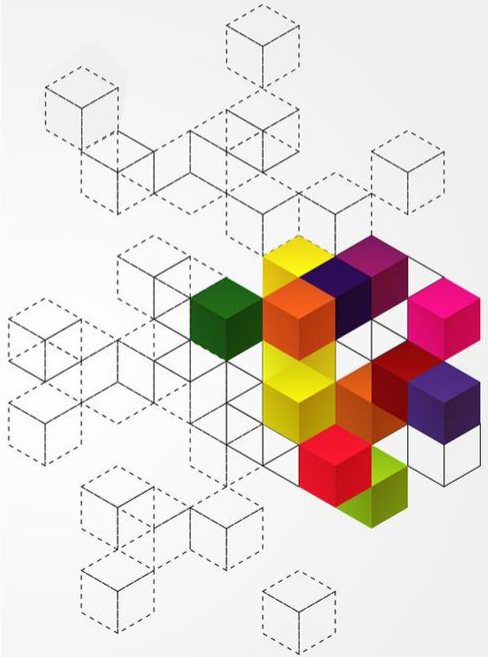
- 初始化
- For each process Pi
- Need[i]=Max[i]
- Allocation[i] = 0
- Finish[i] = false



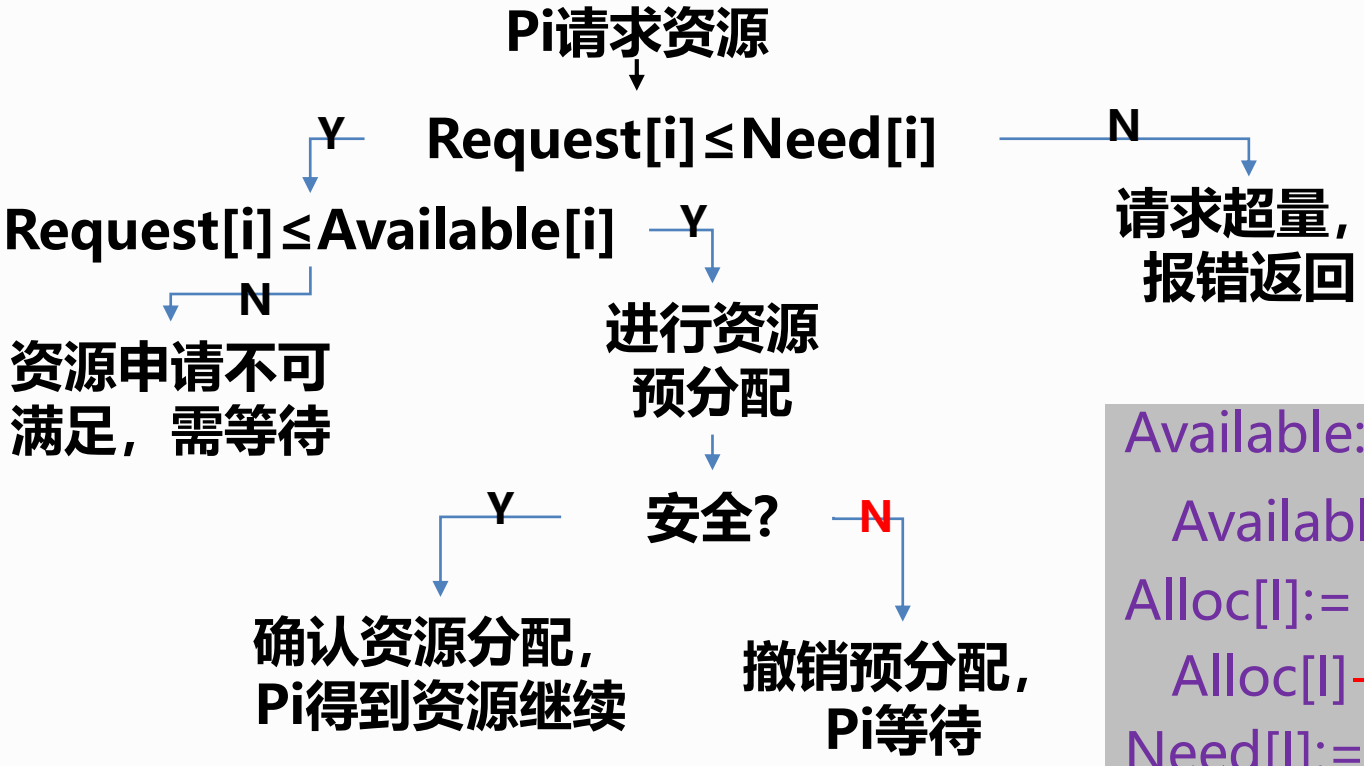
三、多实例的死锁避免算法



三、多实例的死锁避免算法



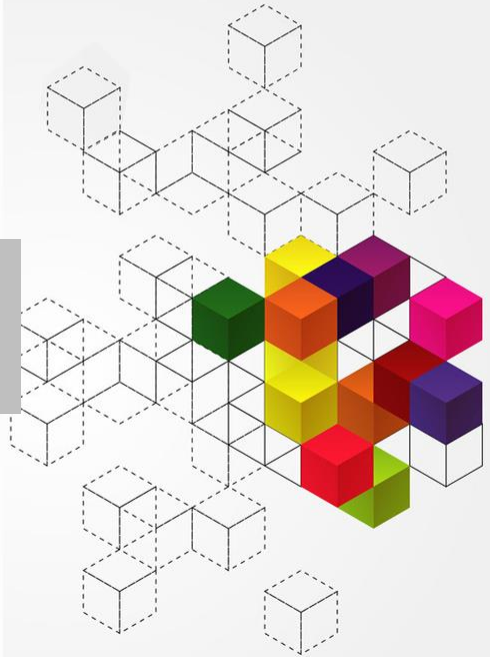
三、多实例的死锁避免算法




```
Available:=
    Available+Req[I]
Alloc[I]:=
    Alloc[I]-Req[I]
Need[I]:=
    Need[I]+Req[I]
```

三、多实例的死锁避免算法

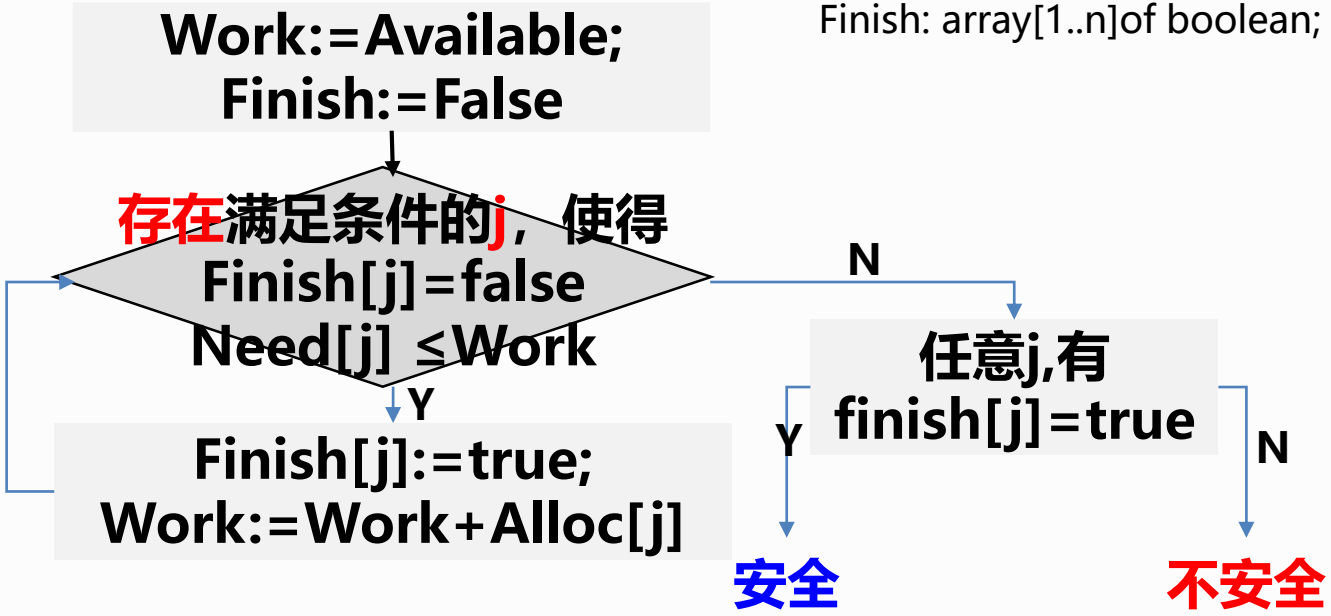
问题2: 如何判定预分配后的系统状态是否安全?

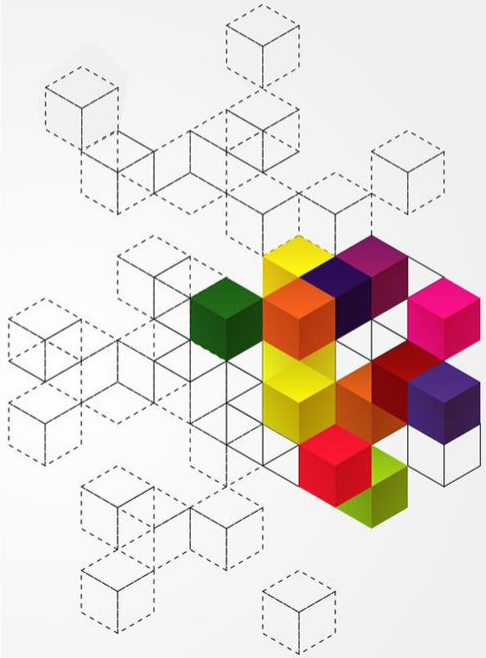


三、多实例的死锁避免算法

银行家算法的安全性判定流程

临时变量：
Work: array[1..m] of integer;
Finish: array[1..n]of boolean;





四、银行家算法实例解析

R={A(10),B(5),C(7)}：3类资源

P={p0,p1,p2,p3,p4}：5个进程

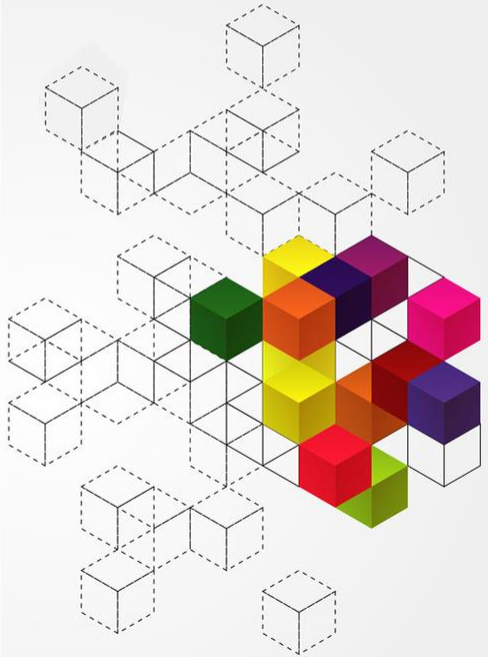
	Max			Alloc			Need			Available			Work			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	
P0:	7	5	3	0	1	0	7	4	3	3	3	2				
p1:	3	2	2	2	0	0	1	2	2							
p2:	9	0	2	3	0	2	6	0	0							
p3:	2	2	2	2	1	1	0	1	1							
p4:	4	3	3	0	0	2	4	3	1							

问题1：当前状态是否安全?

结论：初始状态安全

根据银行家算法的安全性判定流程，找到安全序列

<p1,p3,p4,p2,p0>



四、银行家算法实例解析

R={A(10),B(5),C(7)}：3类资源

P={p0,p1,p2,p3,p4}：5个进程

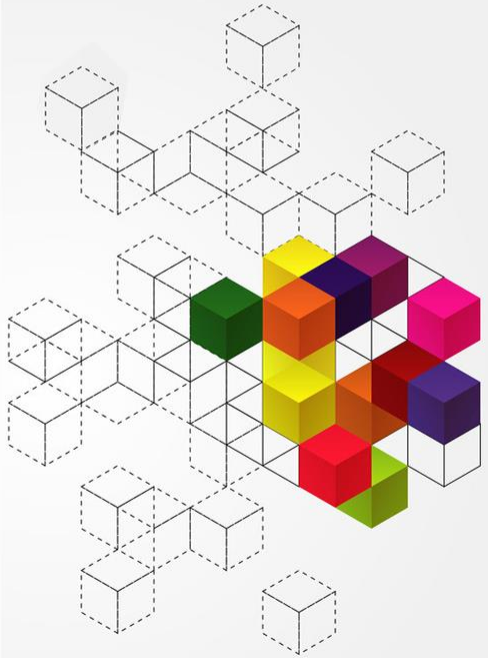
	Max			Alloc			Need			Available			Work			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	
P0:	7	5	3	0	1	0	7	4	3	3	3	2				
p1:	3	2	2	2	0	0	1	2	2							
p2:	9	0	2	3	0	2	6	0	0							
p3:	2	2	2	2	1	1	0	1	1							
p4:	4	3	3	0	0	2	4	3	1							

问题2：

进程P1提出资源申请

Request[1]=(1,0,2)

是否可以满足之



四、银行家算法实例解析

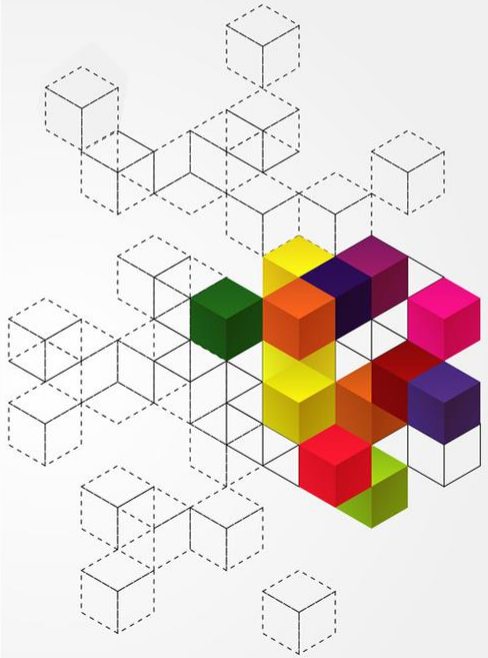
问题2：进程P1提出资源申请Request[1]=(1,0,2)是否可以满足之

步骤2.1：对req[1]进行预分配，资源状态发生变化

	Max	Alloc	Need	Available	Work	Finish
	A B C	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			

步骤2.2：判断预分配后，资源状态是否安全

找到安全进程序列：<p1,p3,p4,p0,p2> ➡ **安全**



四、银行家算法实例解析

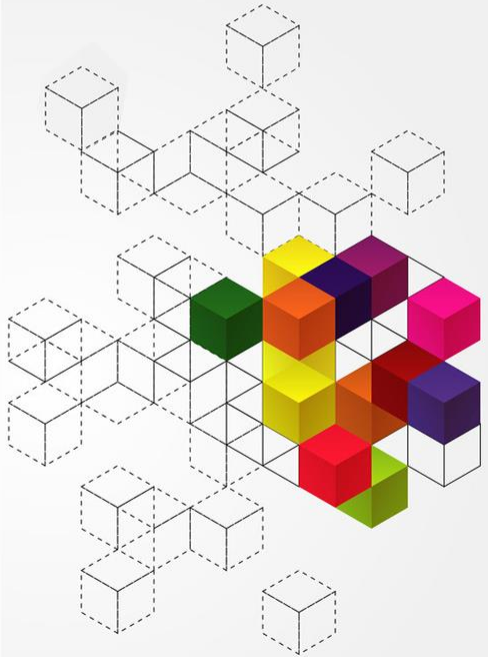
问题2：进程P1提出资源申请Request[1]=(1,0,2)是否可以满足之

步骤2.3：预分配状态安全，允许为P1完成此次分配

	Max	Alloc	Need	Available	Work	Finish
	A B C	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			

问题3：
进程P4此后提出
request[4]=(3,3,0)
能否满足?

结论：资源不够，P4等待

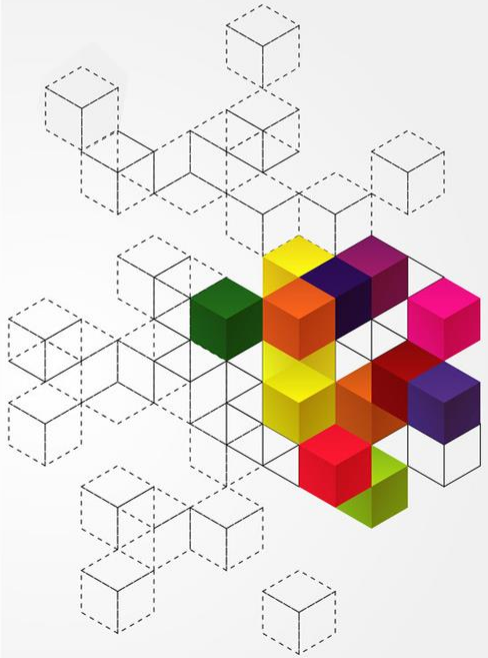


四、银行家算法实例解析

	Max	Alloc	Need	Available	Work	Finish
	A B C	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			

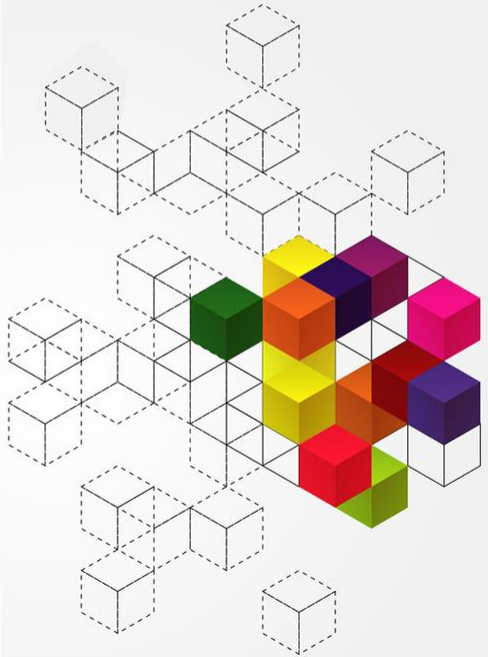
问题4：
进程P0此后提出
request[0]=(0,2,0)
能否满足?

结论：预分配后发现不安全，不能分配



本讲小结

- 死锁避免概念
- 单资源实例条件下的死锁预防
- 多实例的死锁避免算法
- 银行家算法示例解析



内容纲要

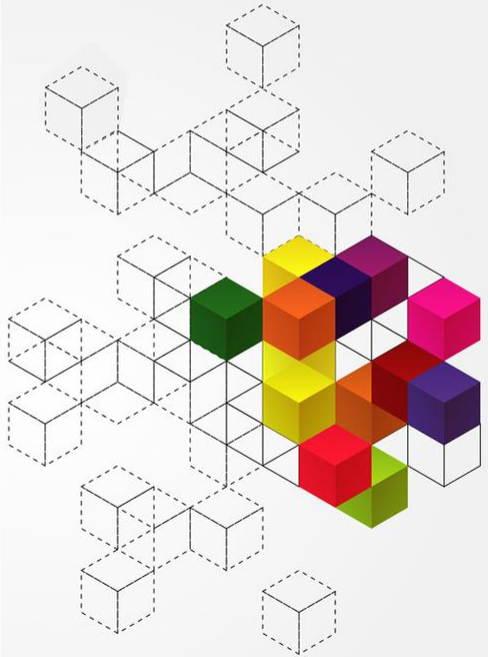
7.6 死锁检测

一、单资源实例条件下的死锁检测

二、多资源实例条件下的死锁检测

三、死锁检测算法的代价分析

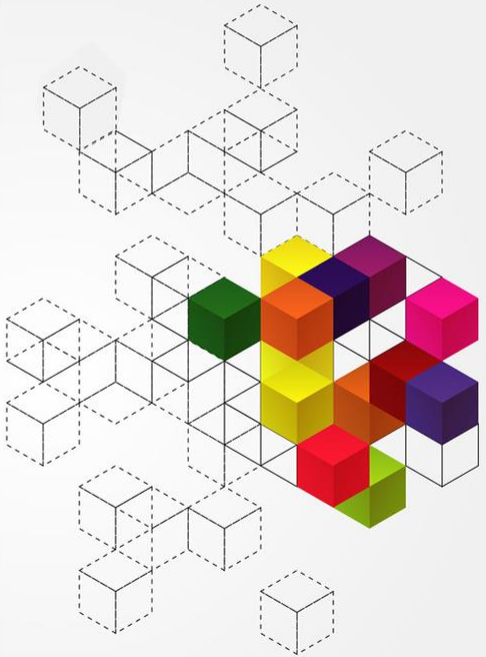
四、死锁检测示例



一、单资源实例条件下的死锁检测

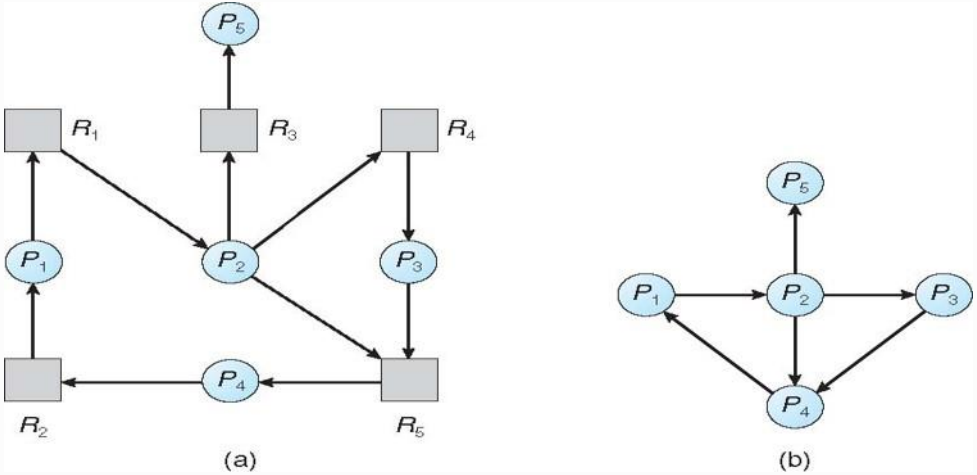
- **死锁检测**
  - 在系统运转过程中，定期进行死锁检测，判断是否有死锁发生
  - 需要设计专门算法来完成死锁检测任务

  - 从资源分配的角度开始考虑，死锁检测算法也要考虑单资源实例与多资源实例这两种不同情况

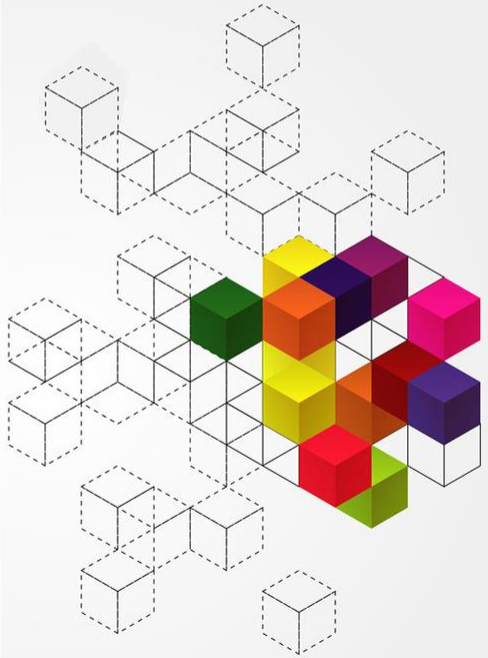


一、单资源实例条件下的死锁检测

- **基本方法**
  - 以资源分配图为基础，构建进程等待图

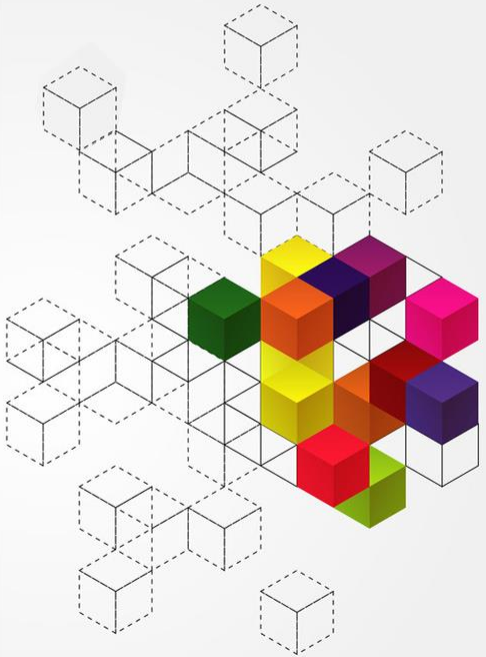


(a)资源分配图　　(b)进程等待图



一、单资源实例条件下的死锁检测

- **基本方法**
  - 以资源分配图为基础，构建进程等待图
  - 实现等待图中的环路检测
  - 定期启动对进程等待图的环路检测，如果发现图中有环，那么报告死锁发生

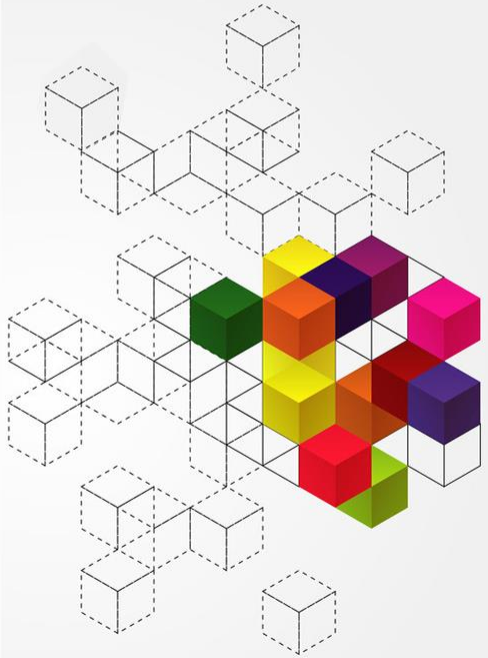


二、多资源实例条件下的死锁检测

基本数据结构（与银行家算法类似）

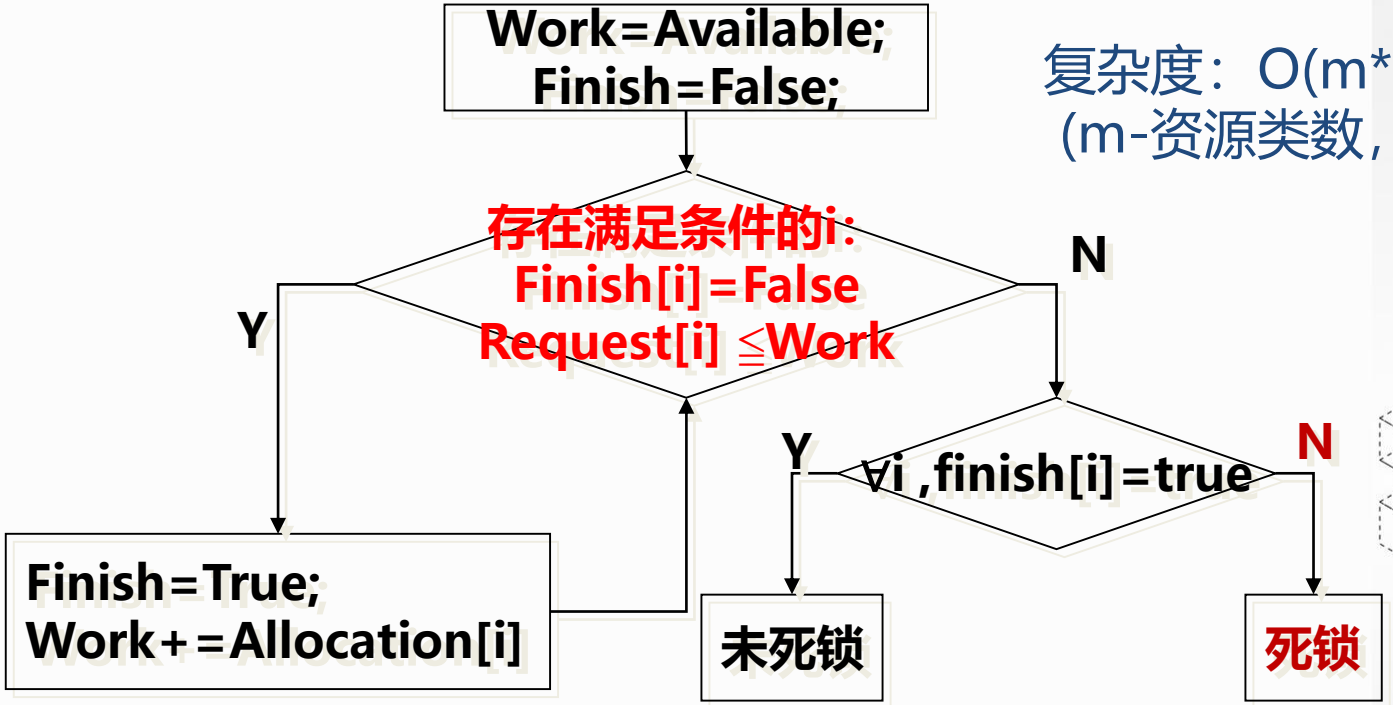
- Available:每类资源当前可用数量。是一个长为m的数组
- Allocation:每个进程当前已分配资源数量.
- Request: n x m 的进程资源请求矩阵

If Request [i][j] = k, then process Pi is requesting k more instances of resource type Rj.

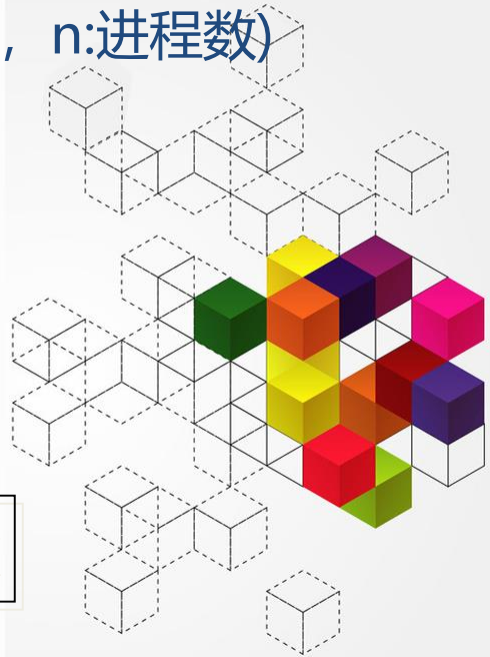


二、多资源实例条件下的死锁检测

死锁检测算法流程



复杂度：O(m*n^2)
(m-资源类数，n:进程数)



复杂度： O(m*n^2)，其中m-资源类数，n:进程数

四、死锁检测示例

多实例死锁检测

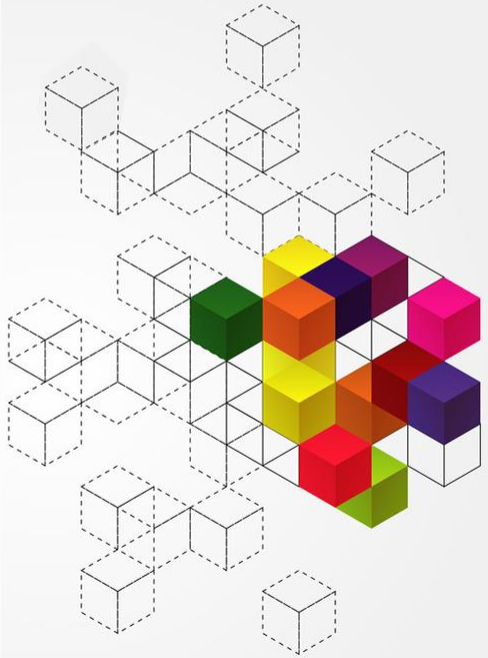
R={A(7),B(2),C(6)};

P={p0,p1,p2,p3,p4}

	Alloc	Request	Available	Work	Finish
	A B C	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			

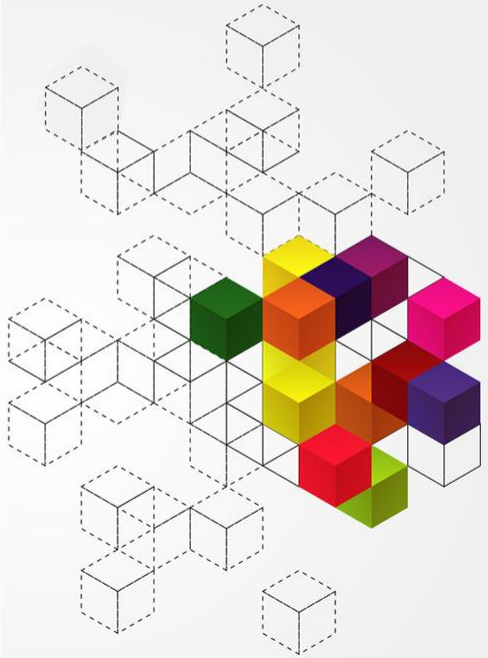
可以找到序列，使得进程能够按序完成

→ **结论：未死锁**



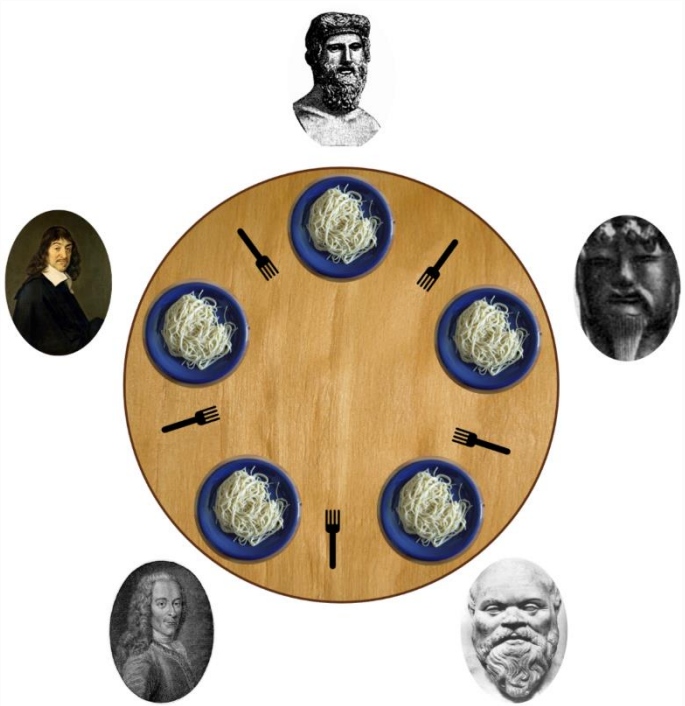
本讲小结

- 死锁检测算法
- 死锁检测算法代价分析
- 死锁检测方法示例



E、哲学家就餐问题

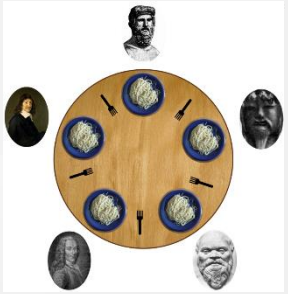
- 五位哲学家就餐



E、哲学家就餐问题-管程实现

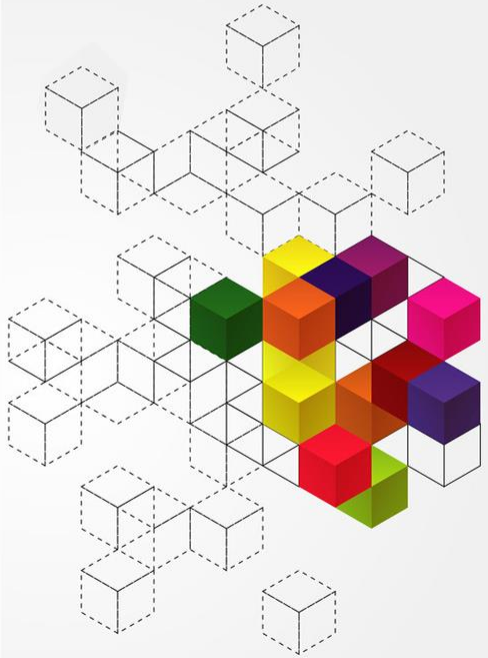
```
monitor DiningPhilosophers
  {
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
   }

      void putdown (int i) {
         state[i] = THINKING;
                // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
       }
```

```
   void test (int i) {
         if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
                  self[i].signal () ;
          }
    }

      initialization_code() {
         for (int i = 0; i < 5; i++)
         state[i] = THINKING;
   }
}
```



E1、死锁练习

见慕课堂练习1

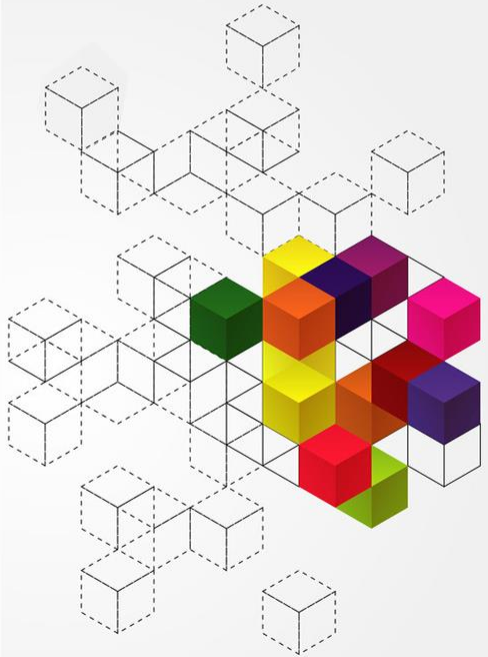


信号量练习题讲解1：

- 设有一个可以装A、B两种物品的仓库，其容量无限大，但要求仓库中A、B两种物品的数量满足下述不等式:

  $-M \leq$ A物品数量 - B物品数量 $\leq N$

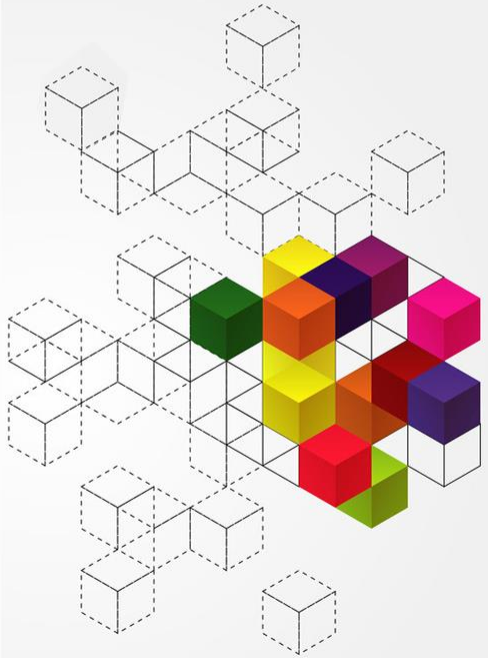
- 其中M和N为正整数。 试用信号量和PV操作描述A、B两种物品的入库过程。



```
semaphore a=n;
semaphore b=m;
```

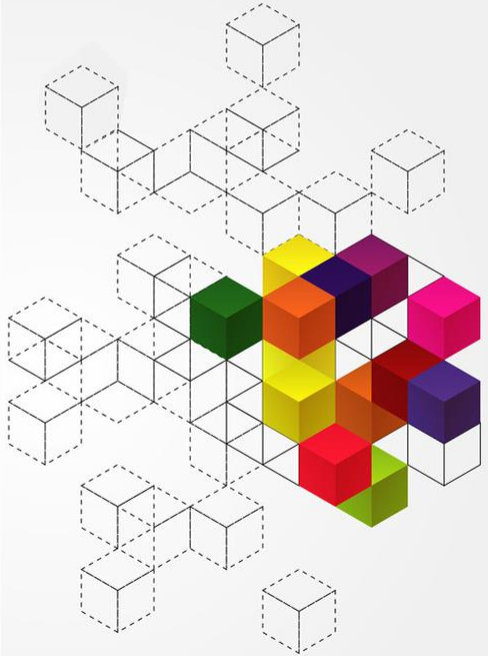
```
A物品入库：
process A()
{
while(1){
   P(a);
   A物品入库;
   V(b);
 }
}
```

```
B物品入库：
process B()
{
  while(1){
     P(b);
     B物品入库;
     V(a);
  }
}
```



信号量练习题讲解1：

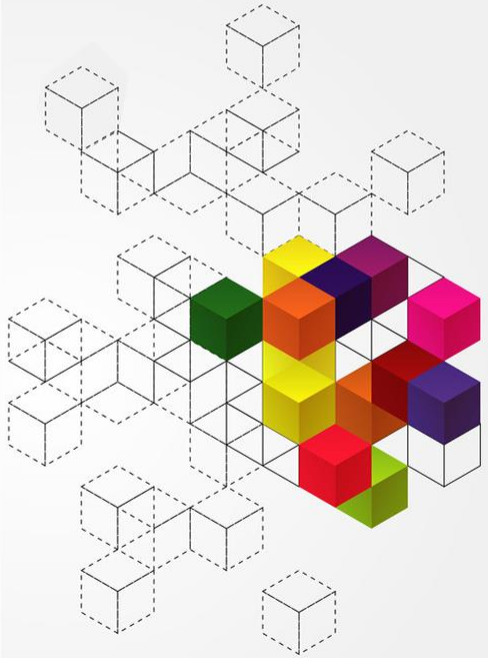
一座小桥(最多只能承重两个人)横跨南北两岸，任意时刻同一方向只允许一人过桥，南侧桥段和北侧桥段较窄只能通过一人，桥中央一处宽敞，允许两个人通过或歇息。试用信号量和PV操作写出南、北两岸过桥的同步算法。



同步关系分析：

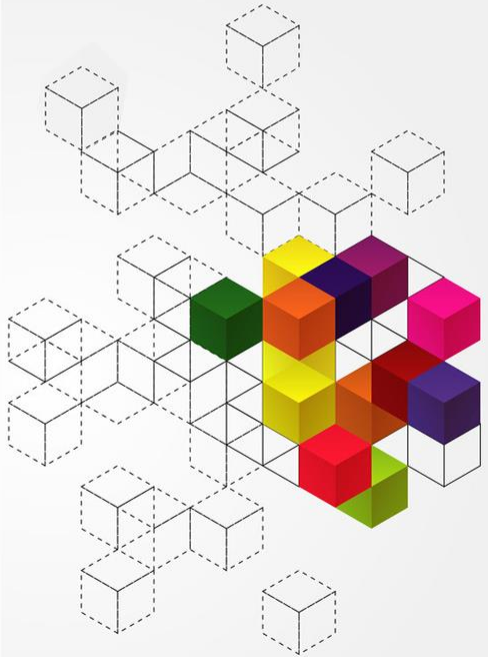
- 桥上可能没有人，也可能有一人，也可能有两人。
- 共需要四个信号量，load1来控制桥上可向南人数，初值为1； load2来控制桥上可向北人数，初值为1；north用来控制北段桥的使用，初值为1，用于对北段桥互斥；south用来控制南段桥的使用，初值为1，用于对南段桥互斥。

```
semaphore load1=1,load2 =1;
semaphore north=1;
semaphore south=1;

  tosouth(){
    P(load1);
    P(north);
    过北段桥;
    到桥中间;
    V(north);
    P(south);
    过南段桥;
    到达南岸
    V(south);
    V(load1);
  }

tonorth(){
  P(load2);
  P(south);
  过南段桥;
  到桥中间
  V(south);
  P(north);
  过北段桥;
  到达北岸
  V(north);
  V(load2);
}
```



E2、进程管理模块-小结

进程同步部分问卷结果	
管程这一部分，上课好像还没怎么讲就跳过了，感觉十分的不熟悉。	就哲学家就餐问题的管程解决方案加深理解
调度算法 信号量	信号量部分已经增加复习，调度算法部分增加习题讲解
关于p操作和v操作，以及进程的划分	找准问题中活动的主体对象以及它的活动



调度算法

- 先来先服务
  - 只考虑每个作业的等待时间
- 最短作业优先
  - 只考虑每个作业的执行时间
- 剩余时间最短者优先
- 轮转法
  - 等待时间与执行时间成比例。时间片长度q值的选择。
- 多级反馈轮转法
  - 划分不同队列，给予不同的优先级和时间片。
- 优先级法
  - 静态优先级
  - 动态优先级

性能衡量指标：周转时间、带权周转时间、响应时间

问题：调度算法

进程	就绪时间	服务时间	等待时间	响应比
P3	4	4	9-4=5	1+5/4=2.25
P4	6	5	9-6=3	1+3/5=1.60
P5	8	2	9-8=1	1+1/2=1.50

进程	就绪时间	服务时间	等待时间	响应比
P4	6	5	13-6=7	1+7/5=2.40
P5	8	2	13-8=5	1+5/2=3.50

	服务时间
	3
	6
	4
P4	5
P5	2

响应比R = (W+T)/T = 1+ W/T

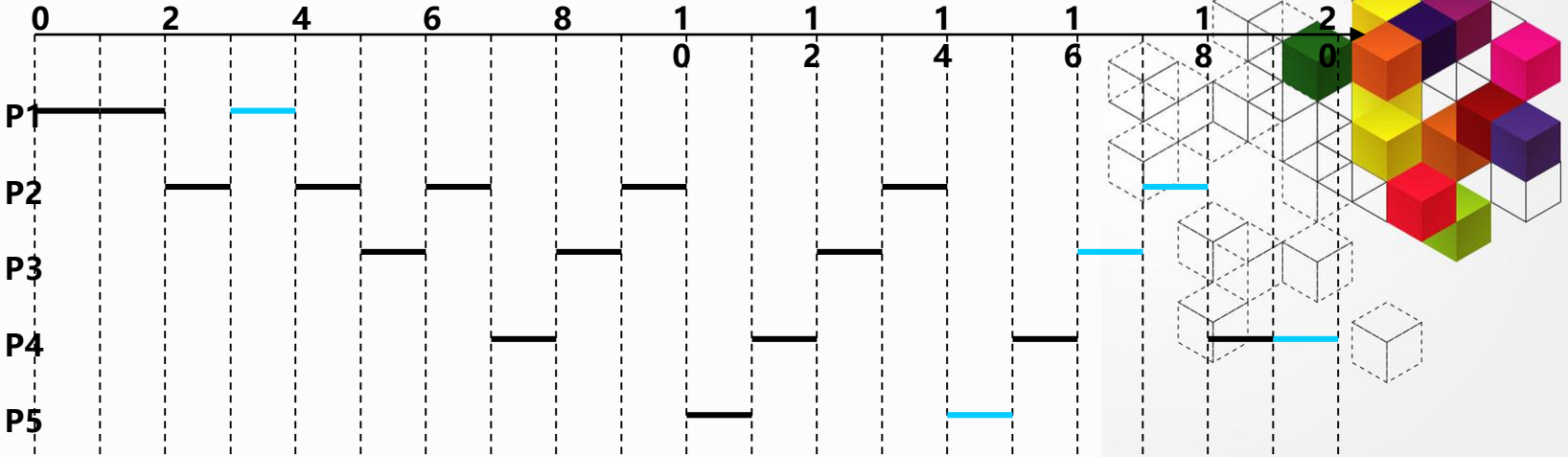
关键点：系统的队列调度模型，即从就绪

调度算法调度顺序图

进程	就绪时间	服务时间
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2

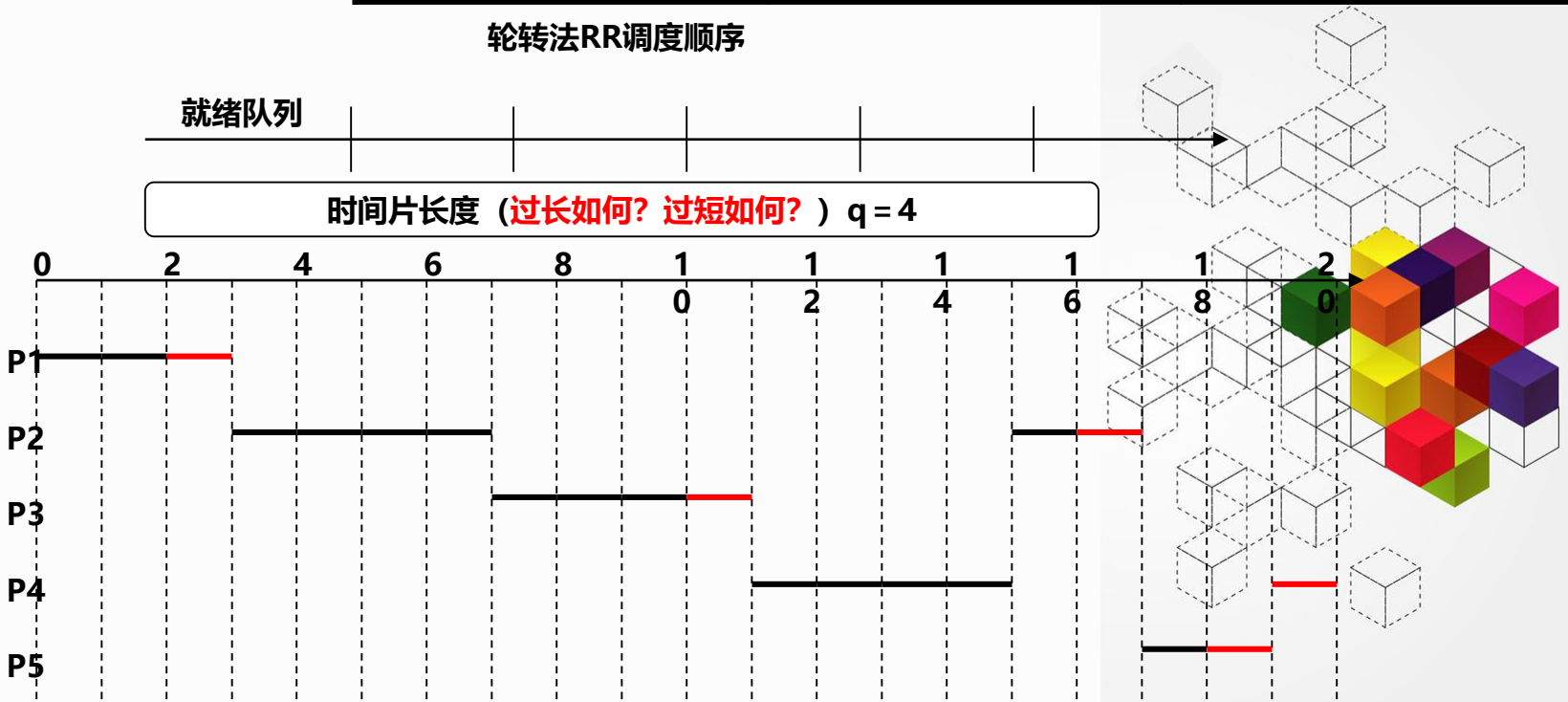
轮转法RR调度顺序

时间片长度（过长如何？过短如何？）q =1和q = 4和q=6



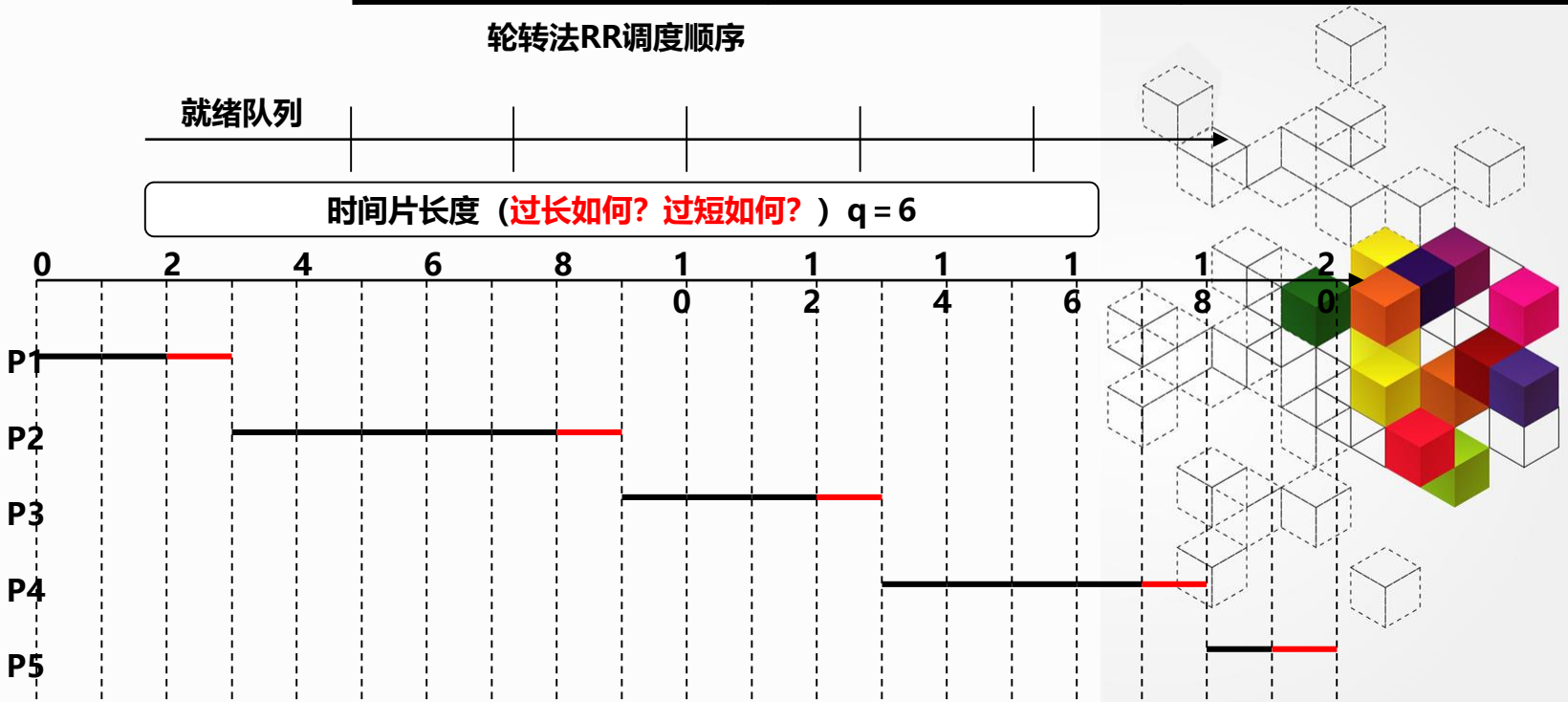
进程	就绪时间	服务时间
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2

轮转法RR调度顺序



进程	就绪时间	服务时间
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2

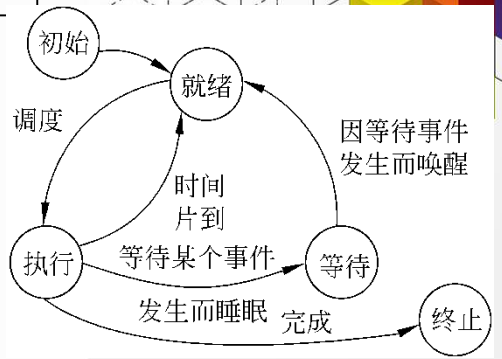
轮转法RR调度顺序



E2、进程管理模块-小结

- 程序的执行
  - 顺序执行
    - 顺序性
    - 封闭性
    - 可再现性
  - 并发执行
    - 间断性
    - 失去封闭性
    - 不可再现性

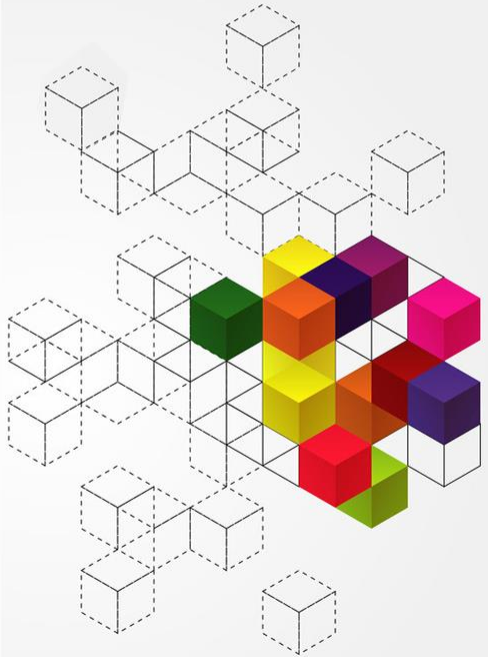
- 进程
  - 特征：动态、并发、独立、异步和结构性
  - 静态描述：PCB、程序段、数据集合
  - 基本状态



E2、进程管理模块-小结

进程控制原语

- 在进程的生命周期中，操作系统需要通过多种操作对其施加控制。常用的进程控制原语有哪些？各完成什么操作？
  - 进程创建原语
  - 进程终止原语
  - 进程阻塞原语
  - 进程唤醒原语
  - 进程挂起原语
  - 进程激活原语



单选题 1分

设置

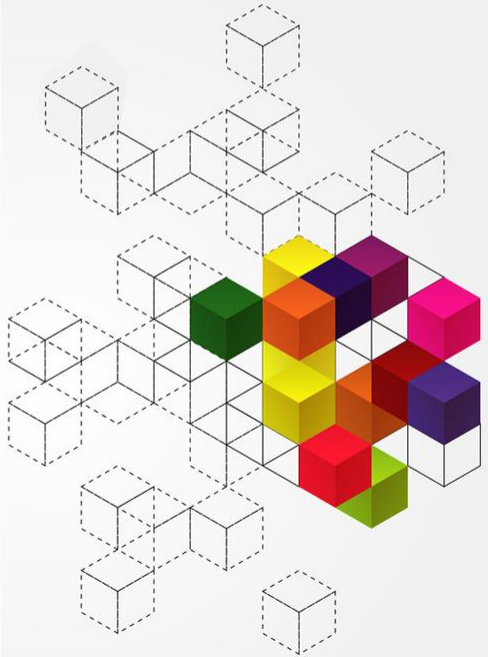
并发进程指的是（ ）。

A 并行执行的进程

B 先后串行执行的进程

C 同时执行的进程

D 不可中断的进程



提交

单选题 1分

设置

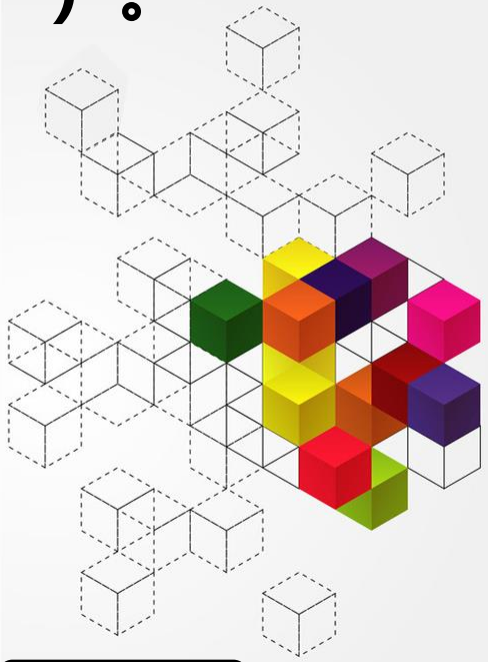
进程上下文中会包含以下的项，除了（ ）。

A 用户打开文件表

B PCB

C 中断向量

D 核心栈



提交

单选题 1分

设置

此题未设置答案，请点击右侧设置按钮

进程和程序的一个本质区别是（ ）。

A 进程是动态的，程序是静态的

B 进程储存在内存，程序储存在外存

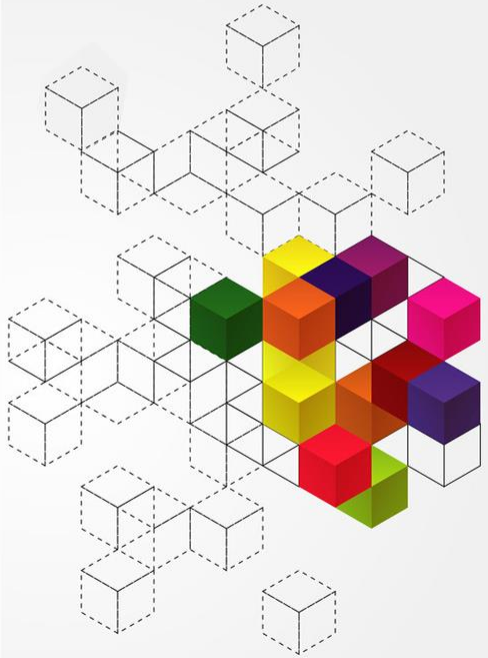
C 进程在一个文件中，程序在多个文件中

D 进程分时使用CPU，程序独占CPU

提交

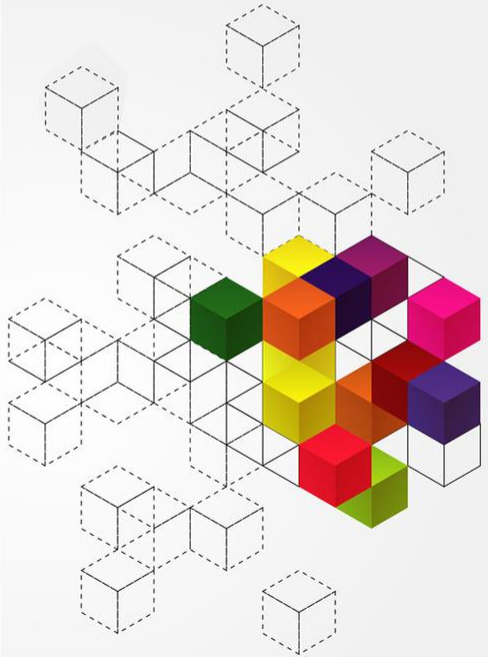
E2、进程管理模块-小结

讨论：进程状态之间转换的原因。



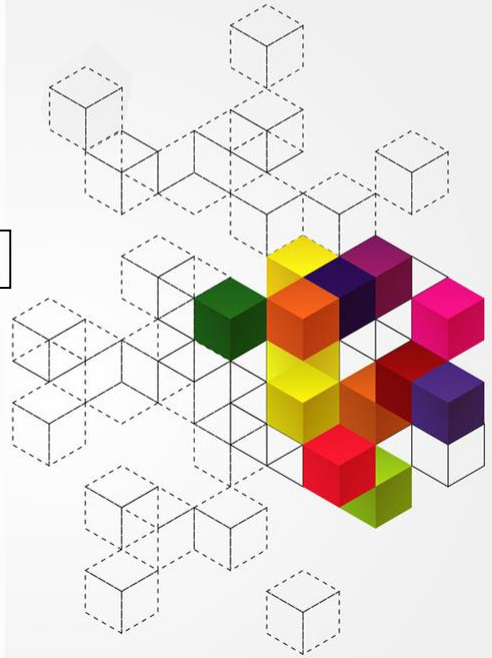
E2、进程管理模块-小结

讨论：当进程发生状态转移时，操作系统对进程实施哪些操作？



E2、进程管理模块-小结

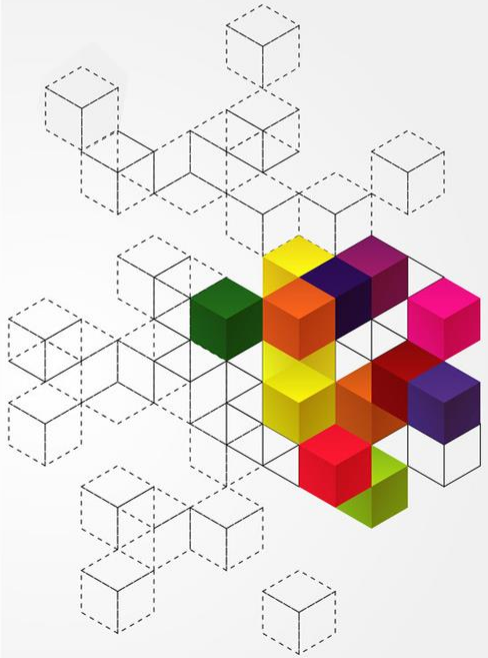
- 线程
  - 引入线程的目的
    - 提高执行效率和节省资源
    - 减少进程的创建和切换开销
  - 线程和进程的比较
    - 调度
    - 并发性
    - 拥有资源
    - 系统开销
  - 线程的基本类型
    - 用户级
    - 核心级



E2、进程管理模块-小结

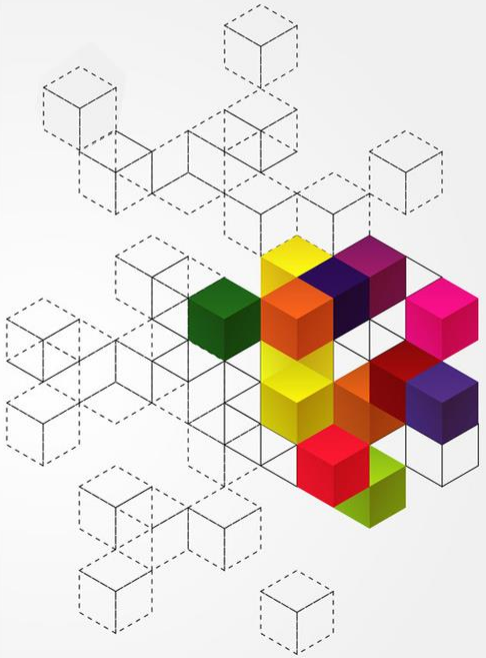
问题1：程序、进程和线程的本质区别是?

- 程序：静态。
- 进程：动态；资源分配的基本单位。
- 线程：动态；调度和分派的基本单位



E2、进程管理模块-小结

- 请简述进程和线程的区别和联系。
  - 调度
  - 并发性
  - 拥有资源
  - 系统开销



E2、进程管理模块-小结

问题3：线程2种基本类型

- 用户级线程和核心级线程有何区别?

① 线程的调度与切换时间

用户级线程的切换通常发生在一个应用进程的多个线程之间，无须通过中断进行OS的内核，且切换规则也简单，因此其切换速度特别快。而核心级线程的切换速度相对较慢。

② 系统调用

用户级线程调用阻塞型系统调用时，内核不知道用户级线程的存在，只是当作是整个进程行为，使进程等待并调度另一个进程执行，在内核完成系统调用而返回时，进程才能继续执行。而核心级线程则以线程为单位进行调度，当线程调度系统调用时，内核将其作为线程的行为，因此阻塞该线程，可以调度该进程中的其他线程执行。

③ 线程执行时间

如果用户设置了用户级线程，系统调用是以进程为单位进行的，但随着进程中线程数目的增加，每个线程得到的执行时间就少。而如果设置的是核心级线程，则调度以线程为单位，因此可以获得良好的执行时间。