操作系统

Operating system

胡燕

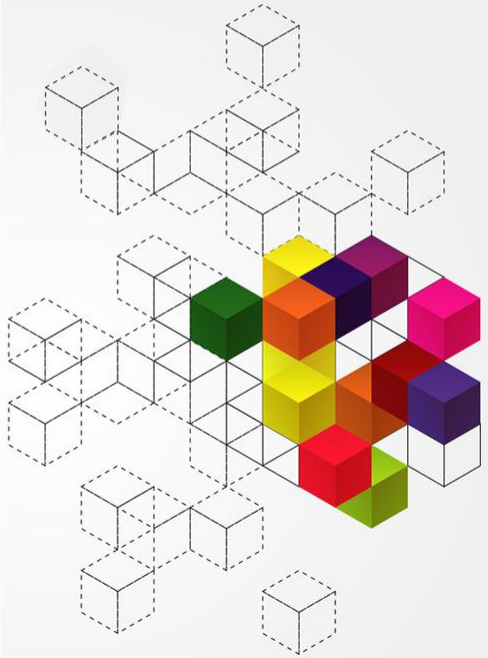
大连理工大学

第6章 进程同步

- **Process Synchronization**

分为三大部分：

- 上：进程同步概念、临界区（互斥）
- 中：信号量
- 下：经典同步问题、管程



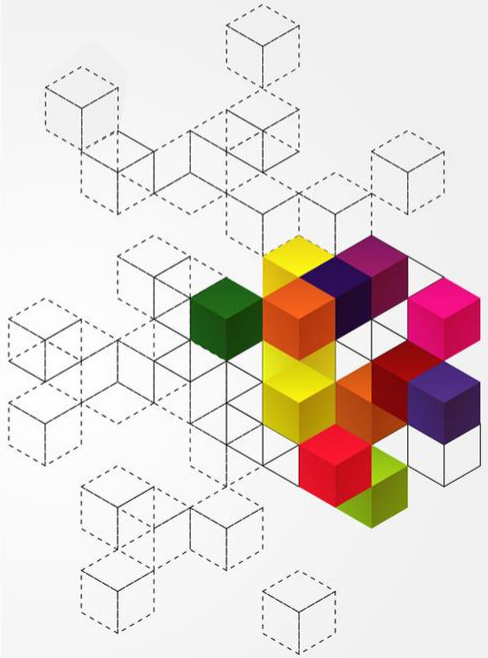
内容纲要

6.1 进程同步基本概念

一、 进程同步概念

二、 进程直接协作

三、 进程间接协作

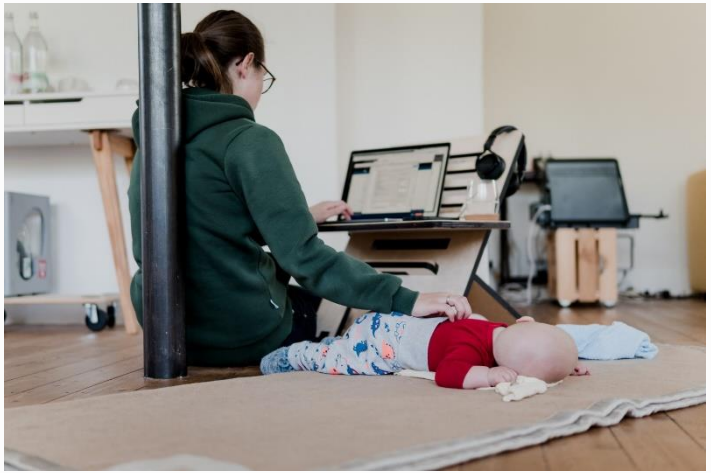


一、进程同步概念

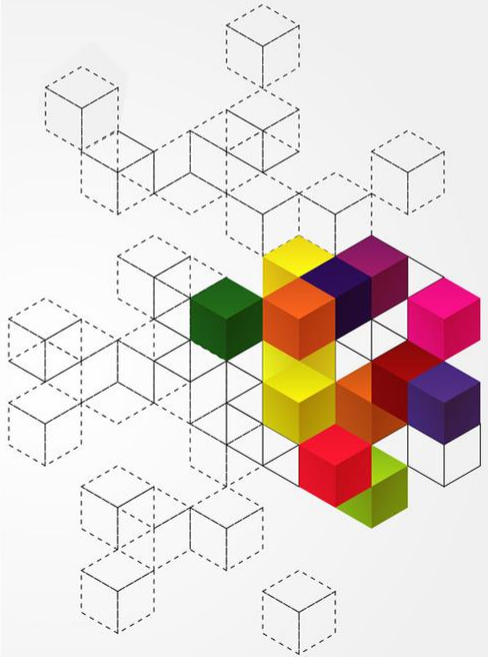
- **Cooperation**

一、进程同步概念

- **现代OS的典型特点：Multitasking**



- 进程（任务）在独立地址空间中各自运行
- 不同进程异步执行

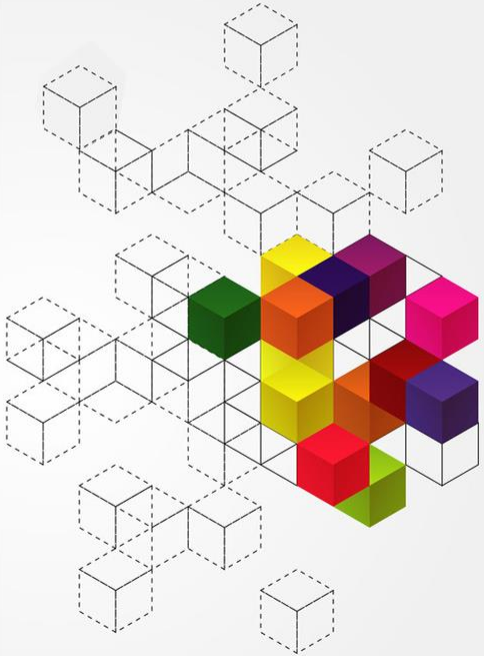


进程间存在协作需求 => Synchronization

一、进程同步概念

- **异步为主，同步为辅**

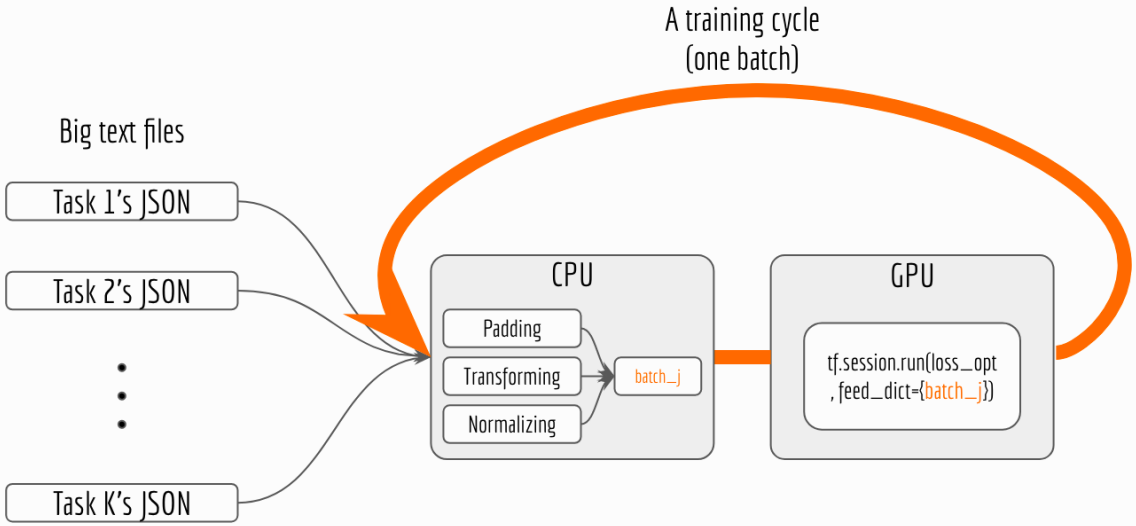
- 当某个进程进行到特定点（如baby睡醒），需要博得另外一个进程关注时，系统必须提供必要的协助

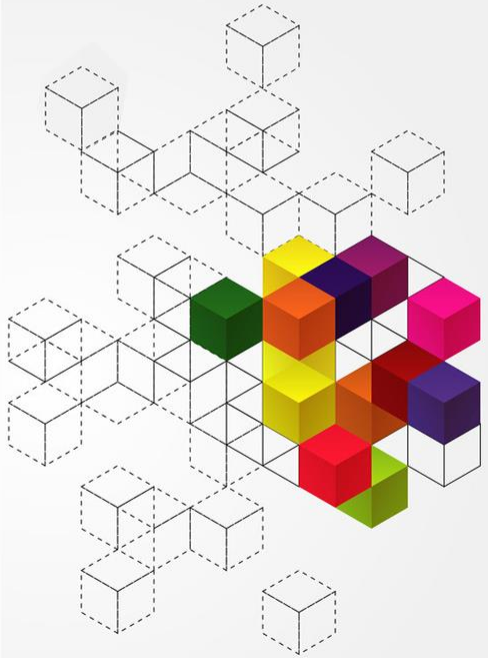


进程异步执行是为了效率，进程同步是为了将多个相关进程联系在一起，保证系统环境正确、和谐

一、进程同步概念

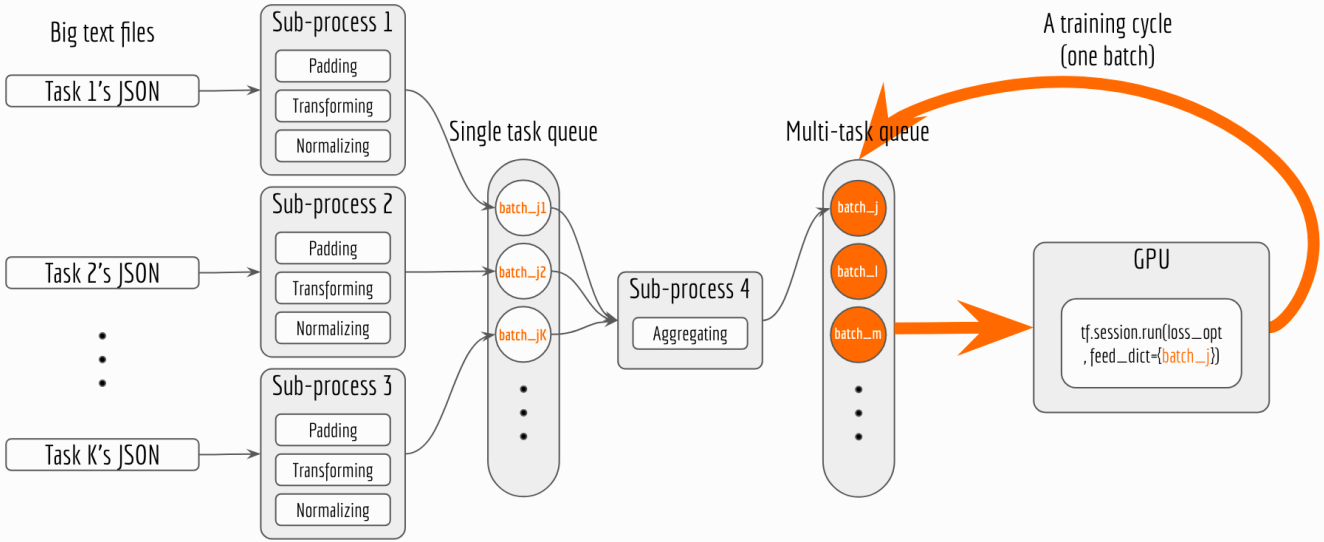
- **Why Process Synchronization?**
- 多进程协同样例：一个基于RNN的训练过程

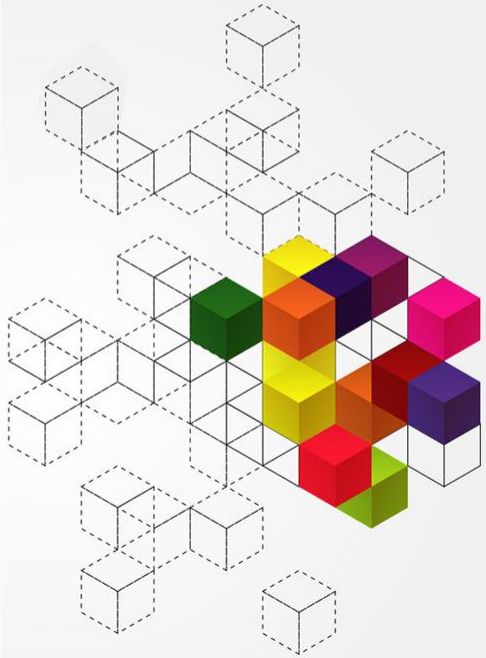




一、进程同步概念

- **Why Process Synchronization?**
- 多进程协同样例：多进程协同





一、进程同步概念

- **Why Process Synchronization?**

现代操作系统上的应用日益复杂

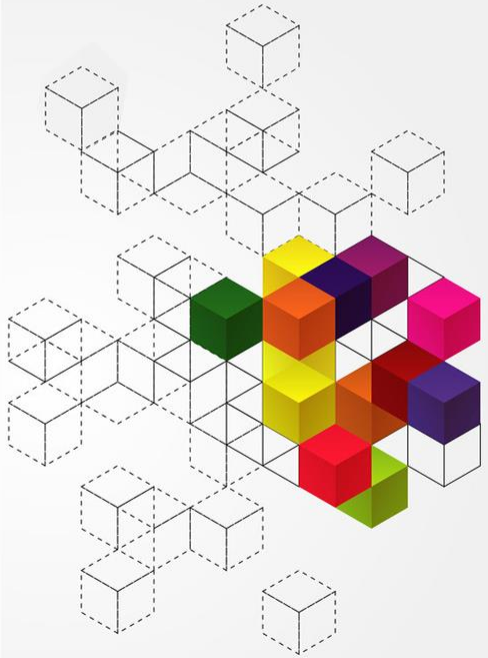
多任务是常态

- 模块化划分后，用多进程、多线程实现，充分利用多核算力
- 任务之间需要传递信息进行协同

任务更频繁涉及I/O：文件I/O，网络I/O等

- 不同任务可能同时访问相同设备

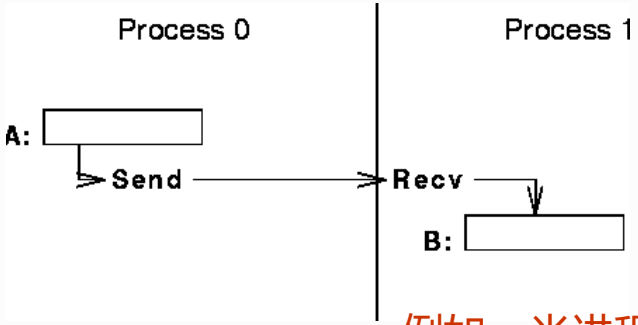
现代应用程序：以异步为基础，同步为辅助



二、进程直接协作

- 进程协作的两种类型:直接协作和间接协作

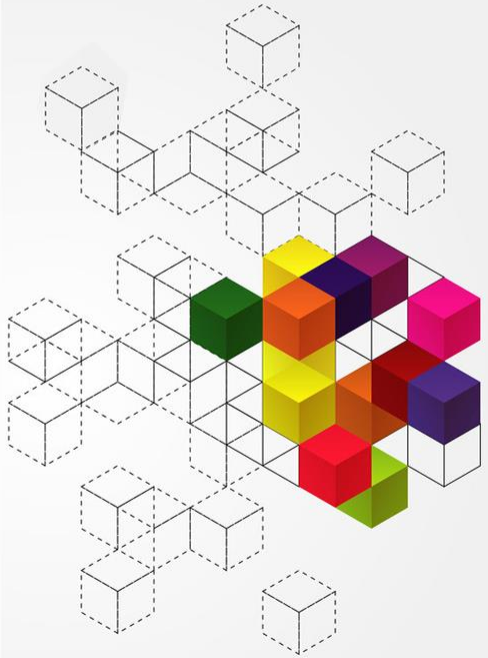
直接协作（Direct Cooperation）



例如：当进程运行到某一点时要求另一伙伴进程为它提供消息

在未获得消息之前，该进程处于等待状态

获得消息后被唤醒进入就绪态



二、进程直接协作

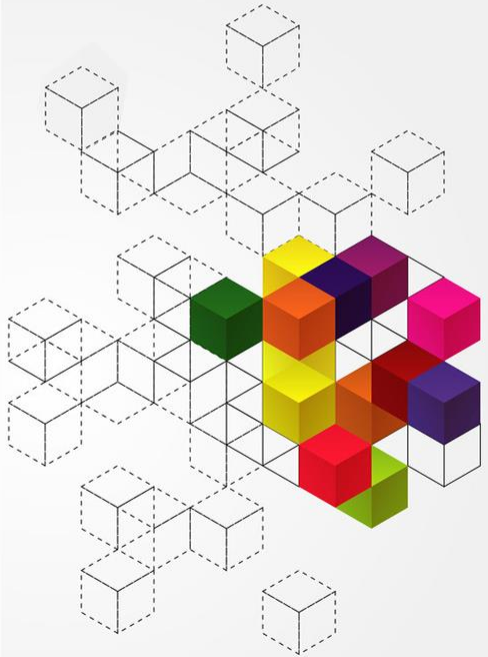
- Direct Cooperation： 示例
  - 司机与售票员之间的协作

司机 P_1

```
While(true)
{
    启动车辆;
    正常运行;
    到站停车;
}
```

售票员 P_2

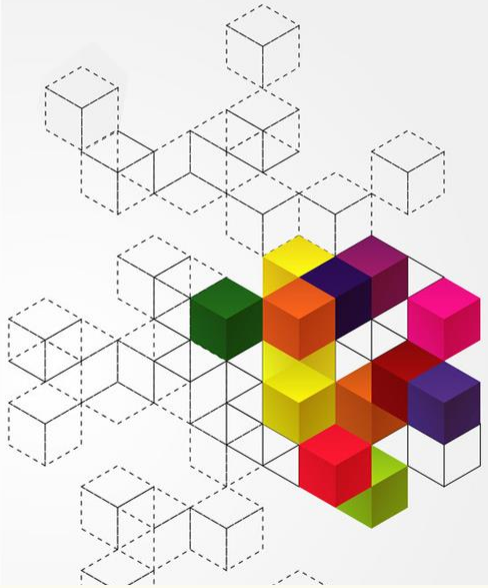
```
While(true)
{
    关门;
    售票;
    开门;
}
```



主观题 5分

设置

试分析，司机与售票员之间的协作关系，并用箭头在合适的位置加以表示。



司机 P_1

```
While(true)
{
    启动车辆;
    正常运行;
    到站停车;
}
```

售票员 P_2

```
While(true)
{
    关门;
    售票;
    开门;
}
```


作答

三、进程间接协作

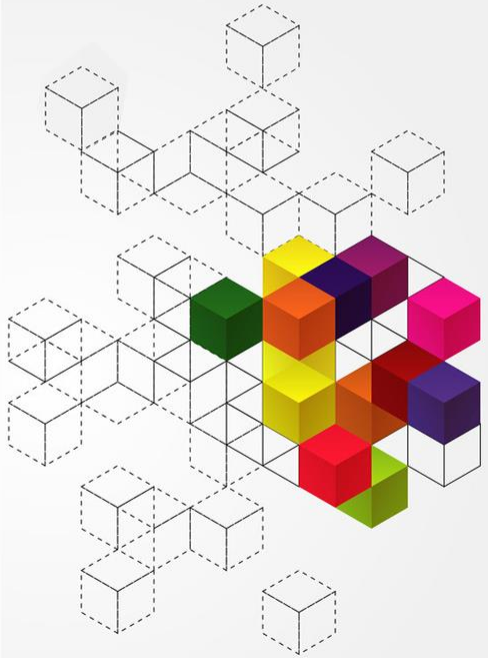
- 进程协作的两种类型:直接协作和间接协作

直接协作（Direct Cooperation）

间接协作（Indirect Cooperation）

进程之间间接作用

- 多个进程竞争使用共享数据（资源）
- 间接作用不会强制合作进程之间遵循特定的先后顺序
- 间接合作进程必须保证对共享数据的一致性访问



三、进程间接协作

- 进程之间间接作用示例

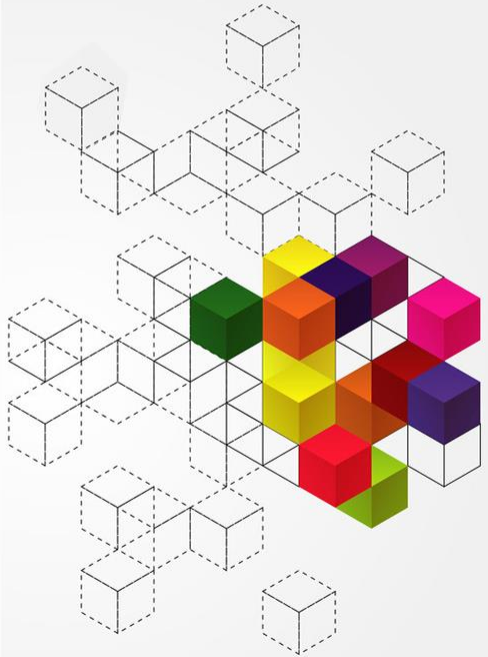
P_1:

```
if(x≥100){
    x-=100;
}
```

P_2:

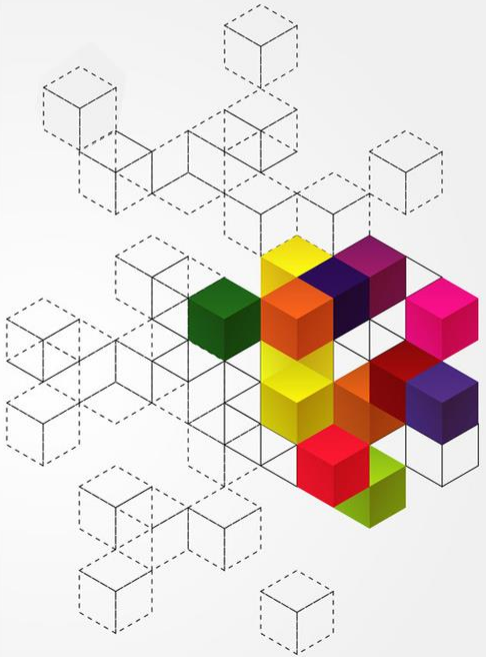
```
if(x≥100){
    x-=100;
}
```

x为共享变量，初值=100



本讲小结

- 进程同步概念
- 进程直接协作
- 进程间接协作



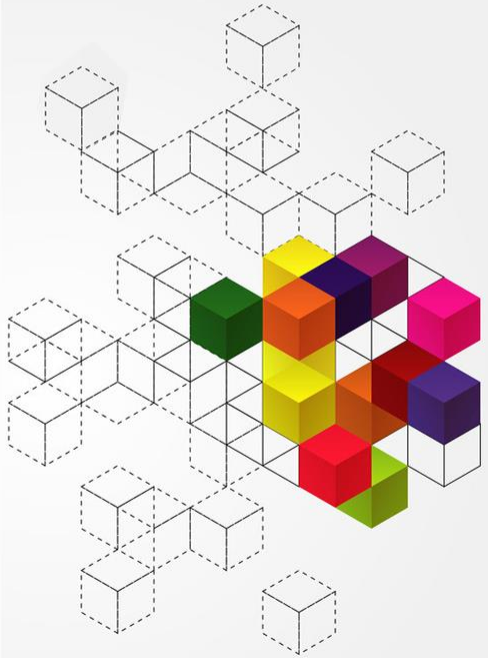
内容纲要

6.2 临界区问题

一、 互斥概念

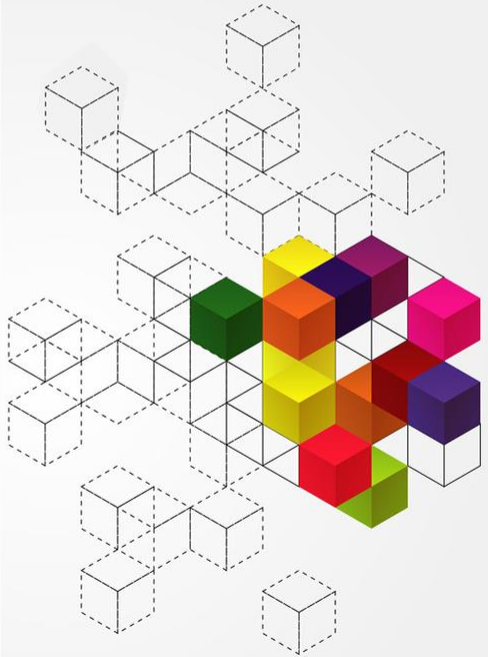
二、 临界区

三、 临界资源



一、互斥概念

- **互斥是进程间接协作的关键性质**
  - 间接进程协作下的进程推进的无序性，导致对其处理的难度相对较大
  - 对共享数据的并发访问可能导致数据不一致
  - 为了保证数据一致性，需保证协作进程按照特定的操作执行顺序进行



一、互斥概念

- **为什么需要分析间接协作性质（示例）**
  - 不同的进程推进顺序，导致不同的结果（不一致）

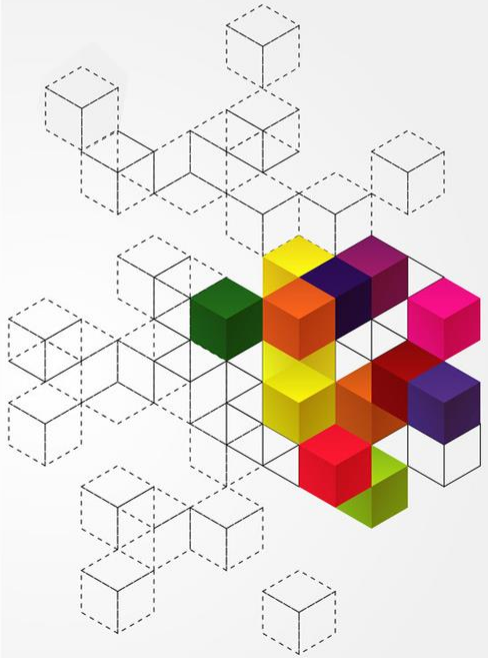
P1:

```
if(x>=100){
    x-=100;
}
```

P2:

```
if(x>100){
    x-=100;
}
```

进程P1	进程P1
if(x>=100);	if(x>=100);
进程P2	进程P1
If(x>=100)	X-=100;
进程P1	进程P2
X-=100;	If(x>=100)
进程P2	进程P2
X-=100;	X-=100;



二、临界区

何谓临界区?

进程中访问共享资源（如共享变量、共享数据结构等）的代码

临界区示例: 网上银行取款

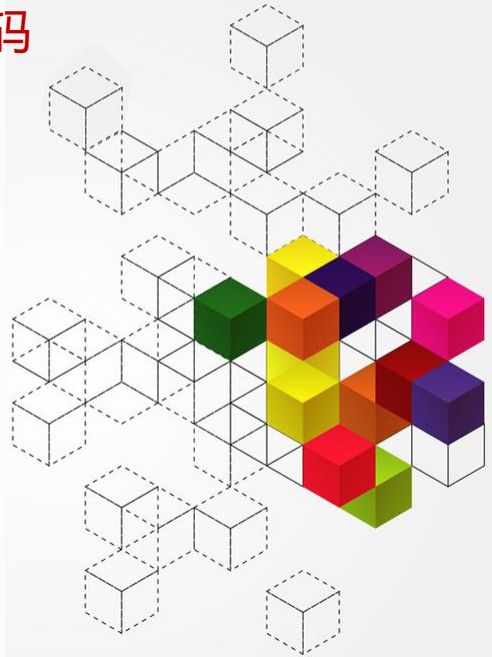
P_1:

```
while (true)
{
   if(x>=100){
      x-=100;
   }
}
```

P_2:

```
while (true)
{
   if(x>=100){
      x-=100;
   }
}
```

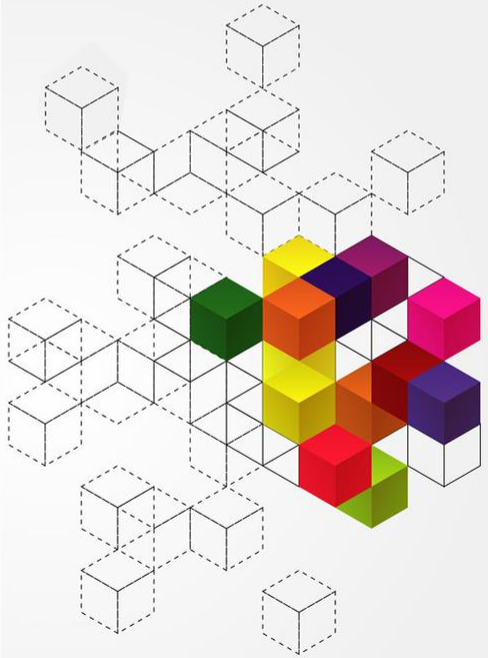
红框区域： **Critical Section**
(Access to shared variable x)



二、临界区

临界区一般表示

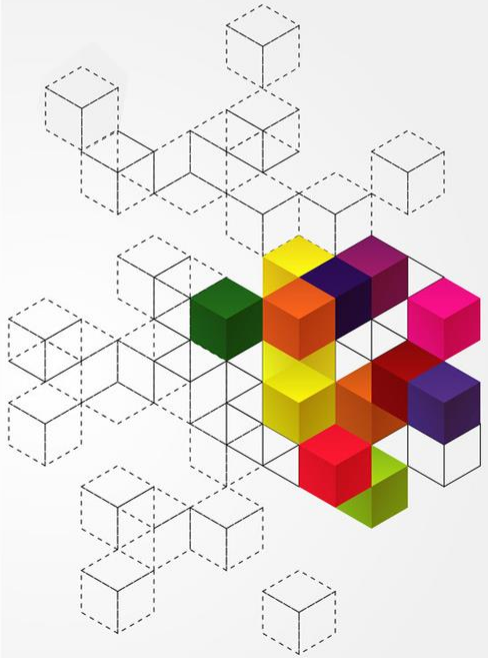
```
While (true)
{
    进入区;   //CS Entry
    CS;
    退出区;   //CS Exit
    剩余区;
}
```



三、临界资源

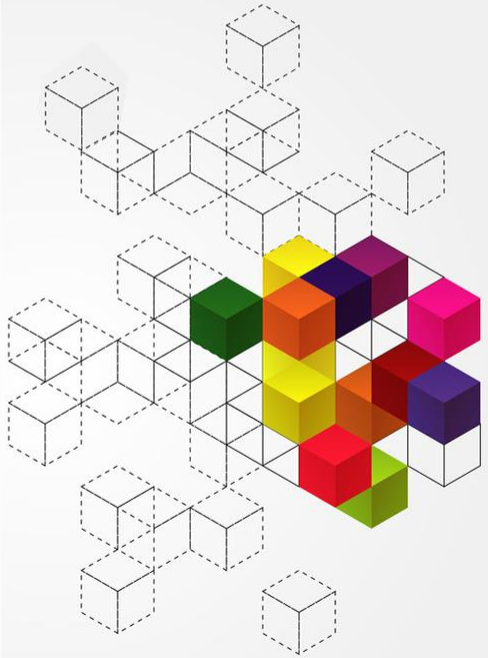
- **典型的临界资源**

临界资源类型	临界资源实例
硬件资源	打印机、磁带机
软件资源	消息缓冲队列、共享变量、 全局数组、共享缓冲区



本讲小结

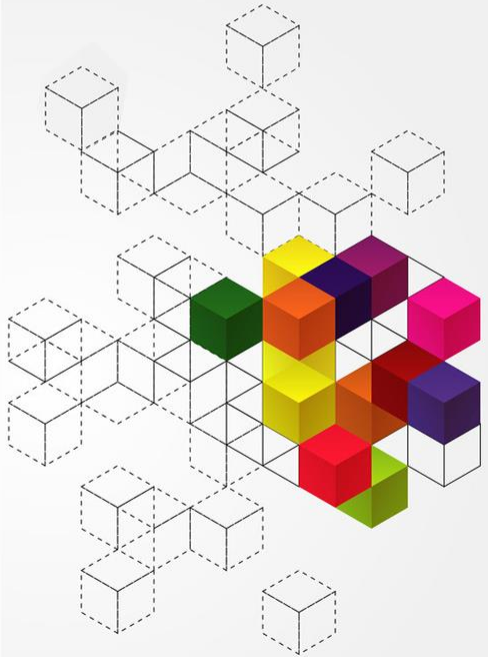
- 互斥概念
- 临界区
- 临界资源



临界区正确实现

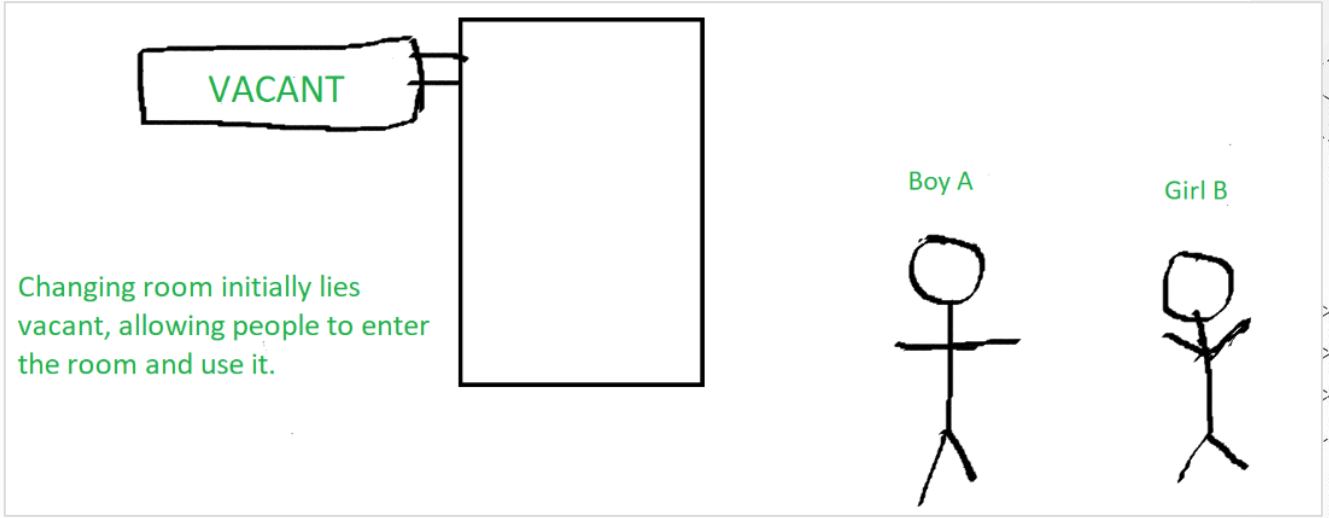
临界区正确实现的基本要求

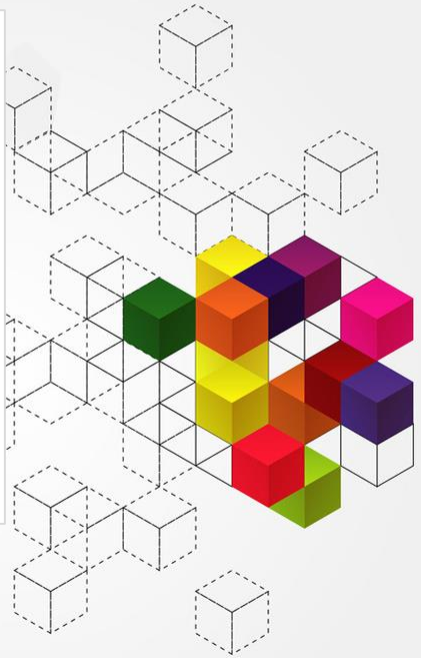
- 互斥（忙则等待）
- 空闲让进
- 优先等待



临界区正确实现

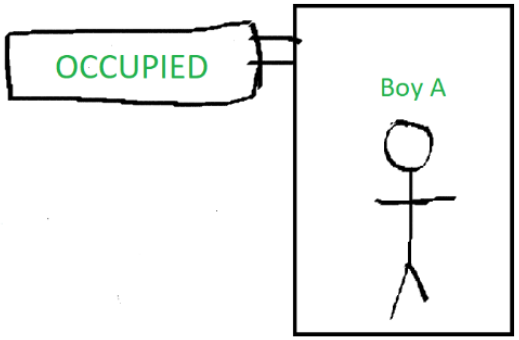
- 进程之间间接作用示例：商场更衣室





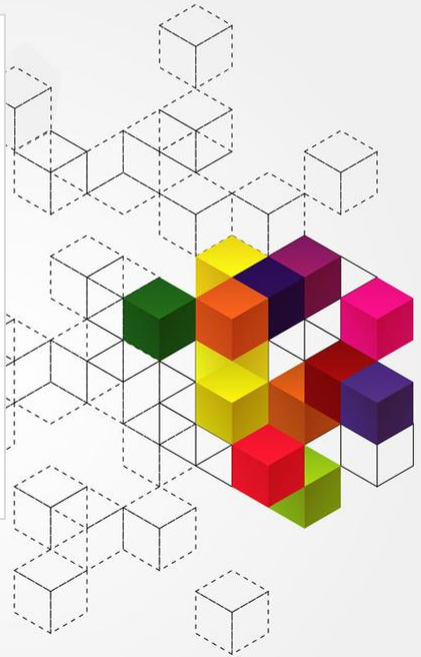
临界区正确实现

- 进程之间间接作用示例：商场更衣室



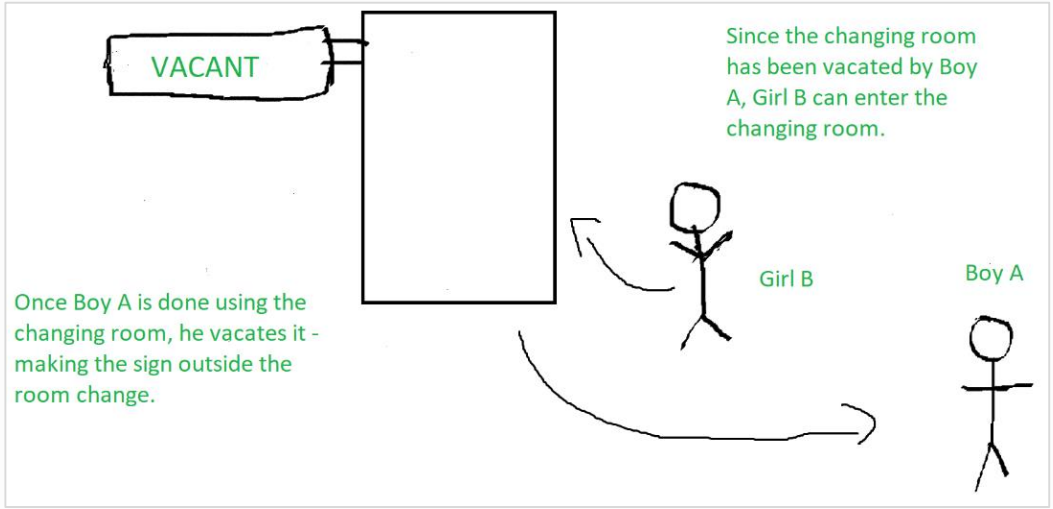
Since Boy A is inside the changing room, the sign on it prevents the others from entering the room.

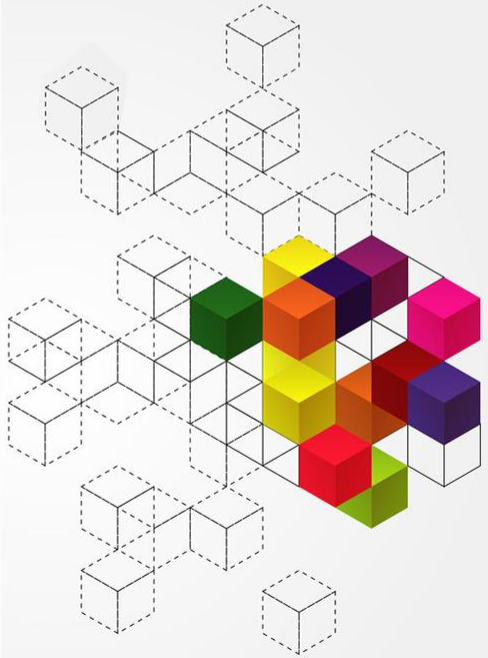
Girl B has to wait outside the changing room till Boy A comes out.



临界区正确实现

- 进程之间间接作用示例：商场更衣室





内容纲要

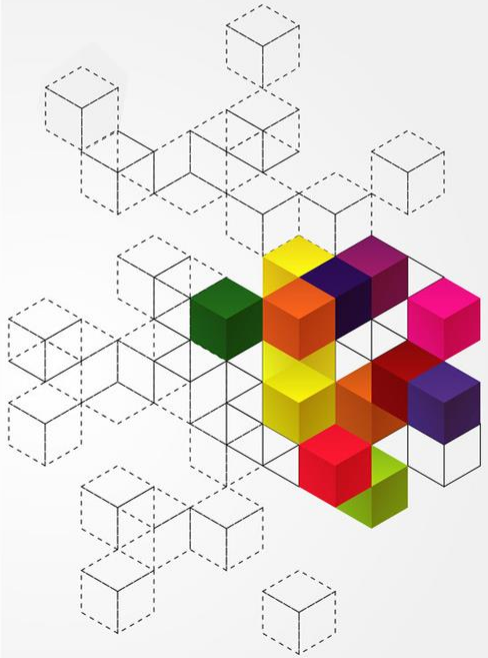
6.3 临界区问题软件解法

一、 两进程解法-尝试1

二、 两进程解法-尝试2

三、 Peterson算法

四、 多进程软件解法



一、两进程解法-尝试1

两进程P0,P1

Try1: 使用一个共享变量turn，初值为0(或1)

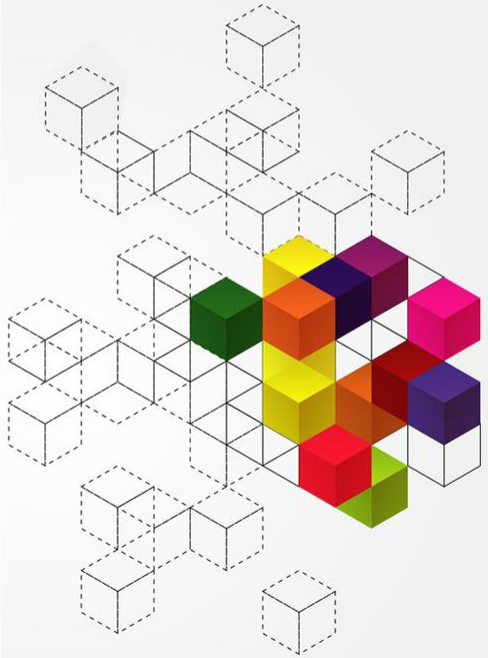
i=0或1
j=1-i

```
do{
   <进入区>
   <临界区>
    <退出区>

    <剩余区>
}while(1);
```

进程Pi:

```
do{
    while(turn!=i);
     <临界区>
    turn=j;

    <剩余区>
}while(1);
```

问题：如果P0循环体执行速度快于P1，可能会出现P0处于turn=1，但是P1实质是在剩余区
（P0空闲不让进）



二、两进程解法-尝试2

两进程P0,P1

进程Pi：

```
do{
    while(turn!=i);
    <临界区>
    turn=j;
    <剩余区>
}while(1);
```

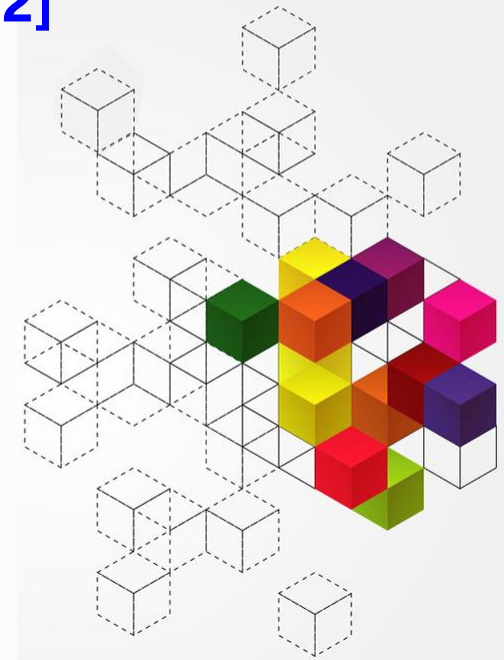
单个共享变量turn显得不够用

i=0或1
j=1-i

Try2：用共享布尔数组flag[2]来替代共享变量turn

进程Pi：

```
do{
    flag[i]=true;
    While(flag[j]);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```



二、两进程解法-尝试2

两进程P0,P1

Try2解决方案

进程Pi：

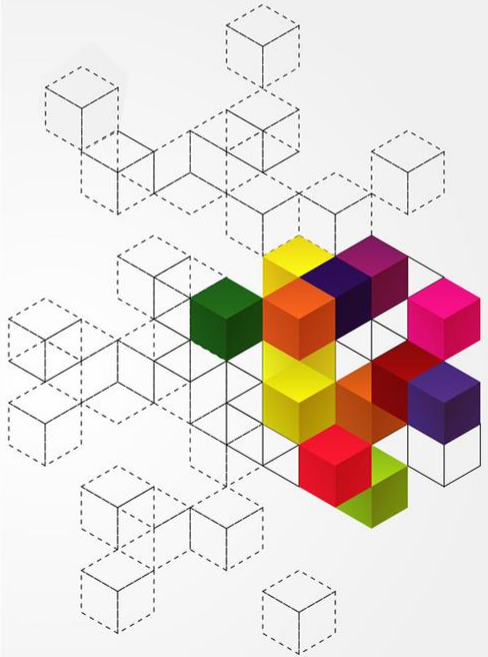
```
do{
    flag[i]=true;
    While(flag[j]);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

i=0或1

j=1-i

问题：考虑到进程P0、P1执行时的异步并发特性，可能会出现P0和P1的准入标志flag[0],flag[1]均为true，使得进程P0，P1陷入死循环的问题

（P0，P1空闲不让进）



三、两进程解法-Peterson算法

两进程P0,P1

Try2解决方案

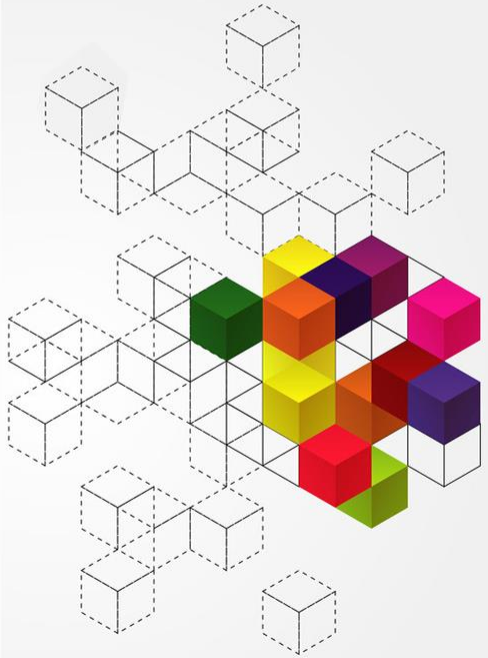
进程Pi：

```
do{
    flag[i]=true;
    while(flag[j]);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

i=0或1
j=1-i

如果解决了死循环问题，两进程互斥的问题即可解决

Peterson给出的方案：
共享变量turn + 数组flag[2]



三、两进程解法-Peterson算法

两进程P0,P1

Try2解决方案

进程Pi：

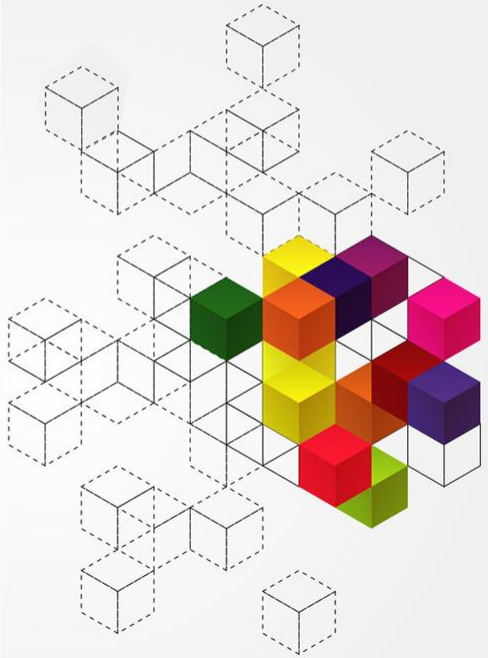
```
do{
    flag[i]=true;
    while(flag[j]);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

i=0或1
j=1-i

Try3 (Peterson算法)

进程Pi：

```
do{
    flag[i]=true;
    turn=j;
    while(flag[j]
        &&turn==j);
    <临界区>
    Flag[i]=false;
    <剩余区>
}while(1);
```

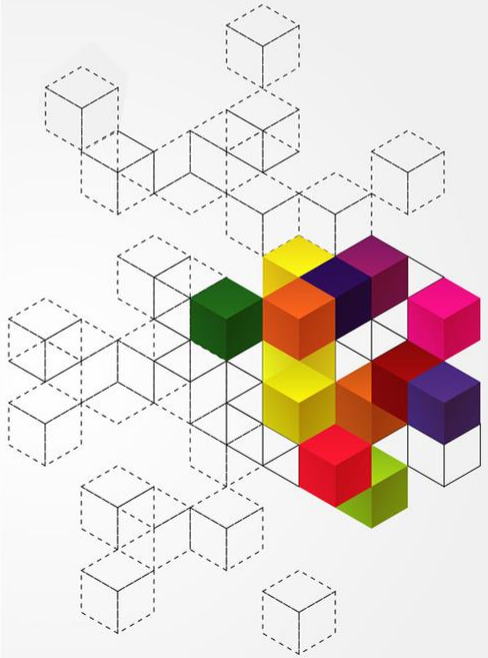


四、多进程软件解法

多进程互斥（面包店算法）

```
do{
    choosing[i]=true;
    number[i]=max(number[0],…,number[n-1])+1;
    choosing[i]=false;

    for(j=0;j<n;j++){
        while(choosing[j]);
        while((number[j]!=0) &&(number[j],j)<(number[i],i));
    }
    临界区
    number[i]=0;
    剩余区
}while(1);
```

取时间戳



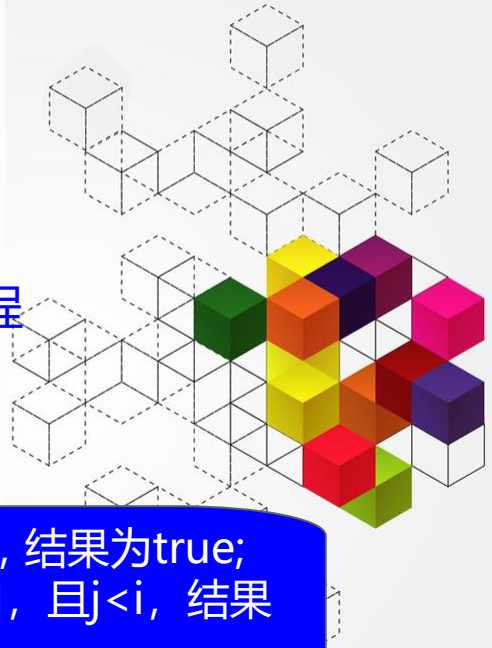
四、多进程软件解法

多进程互斥（面包店算法）

```
do{
    choosing[i]=true;
    number[i]=max(number[0],…,number[n-1])+1;
    choosing[i]=false;

     for(j=0;j<n;j++){
         while(choosing[j]);
         while((number[j]!=0) &&(number[j],j)<(number[i],i));
     }
     临界区
     number[i]=0;
     剩余区
}while(1);
```

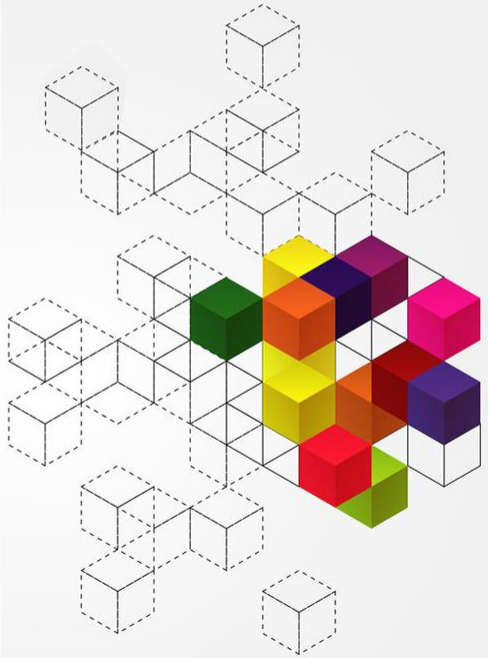
看有没有更“老”的进程

当number[i]<number[j], 结果为true;
当number[j]==number[i]，且j<i，结果为true



本讲小结

- 两进程解法-尝试1
- 两进程解法-尝试2
- 两进程解法-Peterson算法
- 多进程软件解法



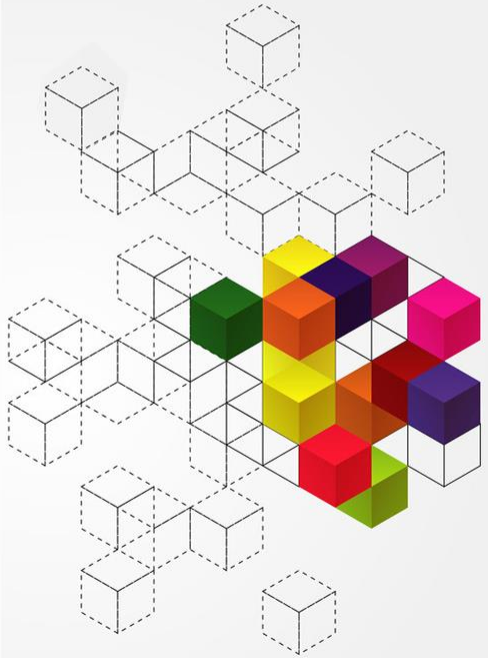
内容纲要

6.4 互斥的硬件实现

一、 引入互斥硬件解法的意义

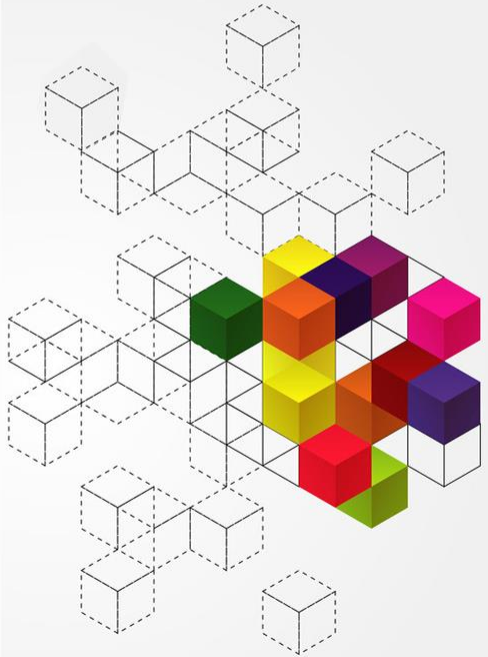
二、 互斥硬件解法的思路

三、 基于加锁操作的互斥实现



一、引入互斥硬件解法的意义

- **互斥，用软件方法实现代价非常大**
  - 一般性的多进程解法（面包店算法），其进入区保护代码的实现成本很高，无法实际应用
- 互斥实现的最大难题
  - 判断进入临界区条件，同时设置本进程已经进入临界区的状态，这两个操作之间可能存在时间窗



二、互斥硬件解法的思路

解决互斥问题的最直观实现

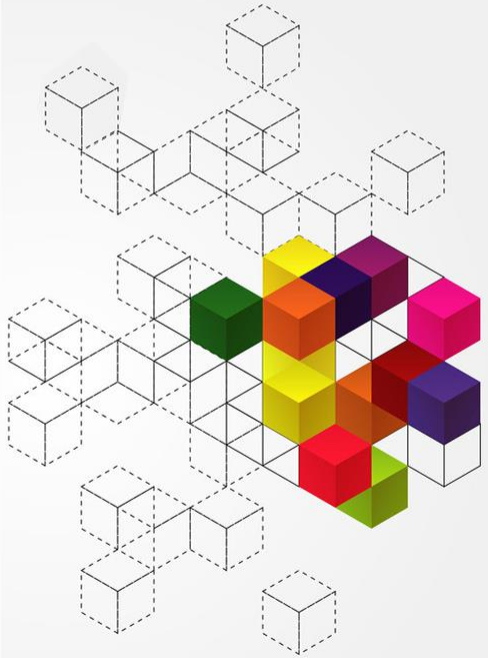
```
lock = 0
```

进程A

```
While(lock != 0)
    NULL;
lock = 1;
Critical_region();
lock = 0;
```

进程B

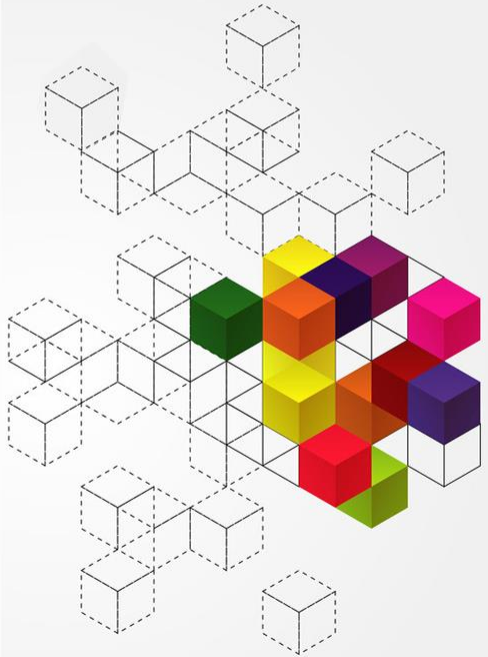
```
While(lock != 0)
    NULL;
lock = 1;
Critical_region();
lock = 0;
```



二、互斥硬件解法的思路

互斥问题的直观加锁解决方案的问题

- 在检查临界区是否被上锁（while循环），以及获准进入后上锁(lock=1)之间存在一个时间窗



二、互斥硬件解法的思路

- 互斥问题的直观加锁解决方案的问题（问题调度）

lock = 0

进程A

While(lock != 0);

进程B

While(lock != 0);

进程A

lock = 1;

Critical Region();

进程B

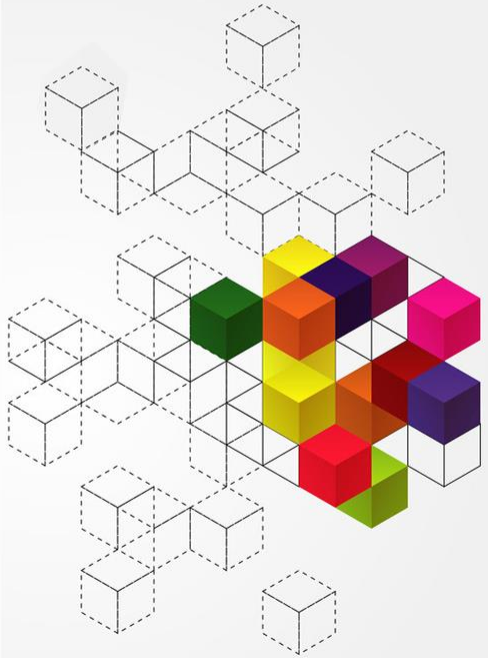
lock = 1;

Critical_Region();

发生进程调度，互斥的进程开始运行

发生进程调度，A并不知道B也进入临界区

发生进程调度，B并不知道A也进入临界区



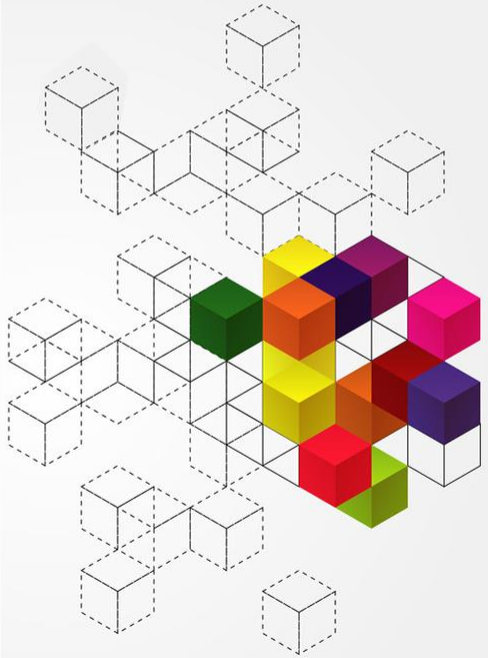
二、互斥硬件解法的思路

互斥问题的直观加锁解决方案的问题

```
While(lock!=0);
Lock=1;
```

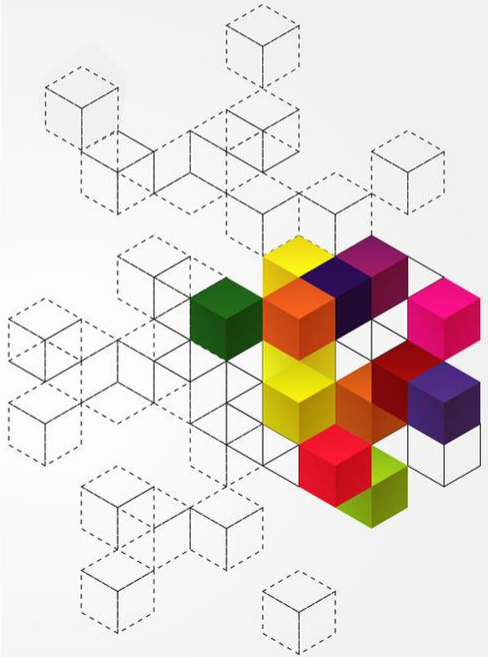
X 其他任务的指令

- 保证检查lock状态的while循环、lock置1的操作一起执行，不需被中断。
- 这种执行期间不会被中断的指令序列，称为原子操作（Atomic Operation）



二、互斥硬件解法的思路

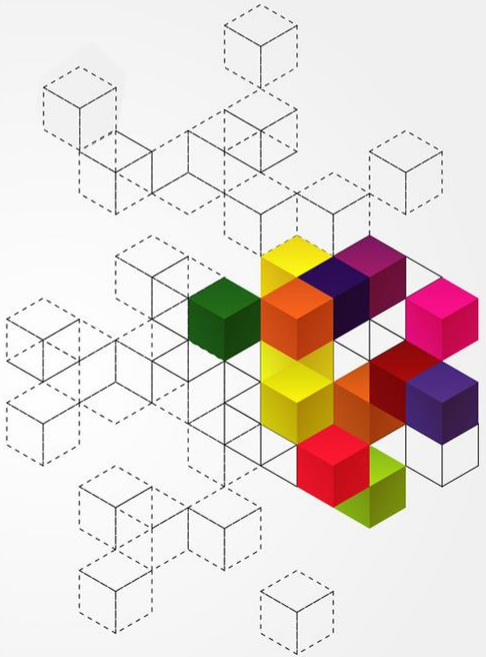
- 最终实现思路归结为：提供TestAndSet指令
  - 使用TestAndSet来处理上述临界区问题
  - TestAndSet(lock)指令语义 ：
    - lock=0, 则将lock置1，返回0
    - lock=1,则直接返回1
  - TestAndSet实例： x86上的xchg指令



二、互斥硬件解法的思路

- 终于实现

“躲进厕所锁上门，我把全世界的人都锁在外面”



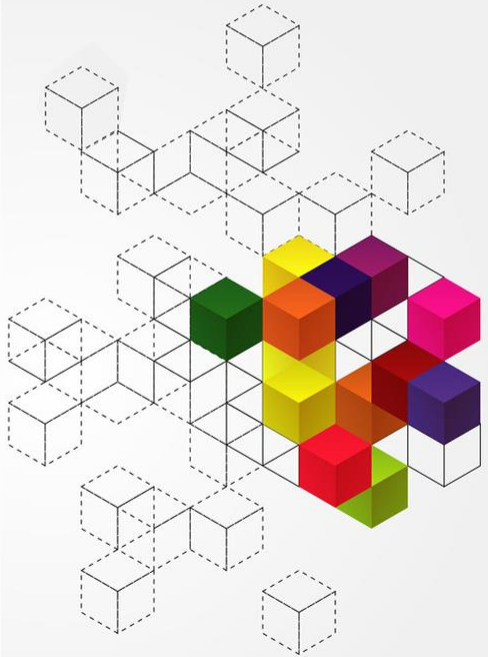
三、基于加锁操作的互斥实现

临界区通过基于TestAndSet原子操作实现的锁对象保护，方式如下

```
do{
    while(TestAndSet(lock));
    临界区
    lock = false;
    剩余区;
}while(1);
```

满足临界区条件(1)和(2):
-互斥
-空闲让进

不满足临界区条件（3）：非有限等待



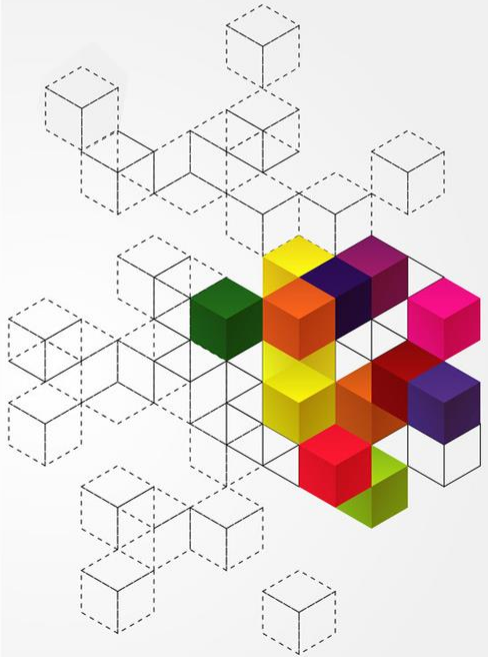
三、基于加锁操作的互斥实现

```
do{
    waiting[i] = true
    key = true;
    while(waiting[i]&&key)
        key=TestAndSet(lock);
    waiting[i]=false;

    临界区

    j=(i+1)%n;
    while(j!=i && !waiting[j])
        j = (j+1)%n;
    if(j==i)
        lock = false;
    else
        waiting[j] = false;
     剩余区
}while(1);
```

表示进程Pi处于
等待获取锁的状态

如果进程Pi抢到了锁，
记录key=false

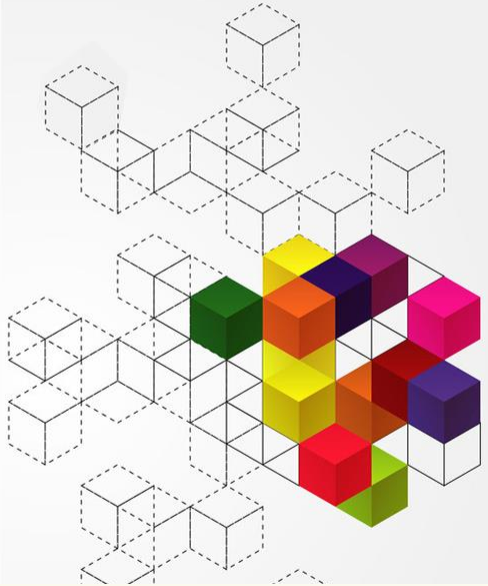
进程Pi处于
等待获取锁的状态



主观题 5分

设置

请从你的日常生活经验中，发掘涉及互斥的场景，并加以生动描述。

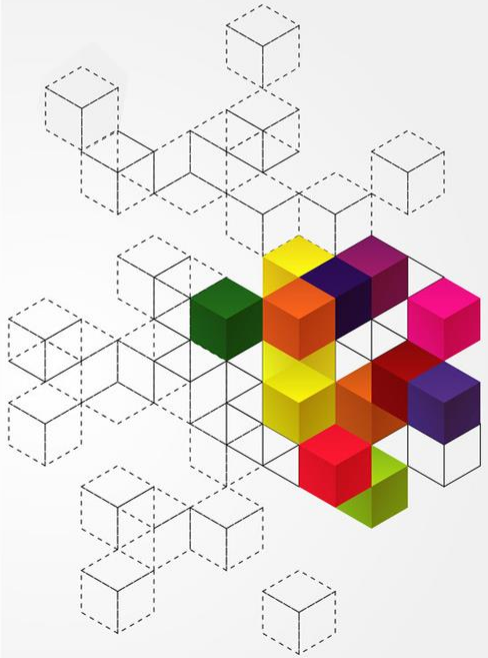


正常使用主观题需2.0以上版本雨课堂

作答

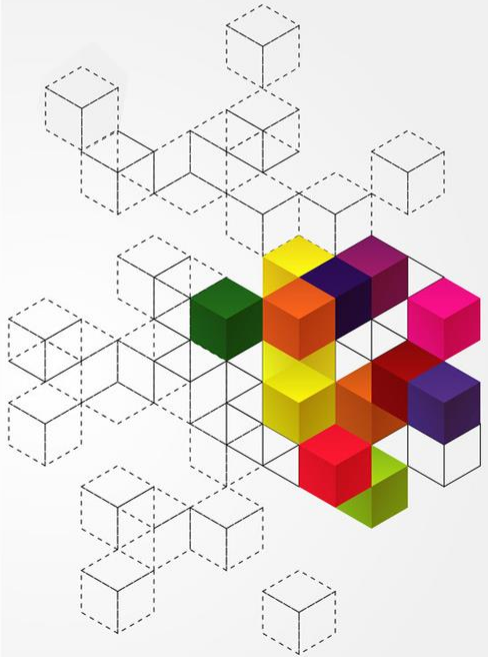
本讲小结

- 引入互斥硬件解法的意义
- 互斥硬件解法的思路
- 基于加锁操作的互斥实现



L08 课后思考题

1.旅客在购票网站上购买同一家航空公司的机票。这个过程中涉及的临界区问题及其互斥解决方案，请用伪码加以表示。



L08 课后思考题

2.编写存在临界区的多线程代码，实际体验互斥问题。