操作系统

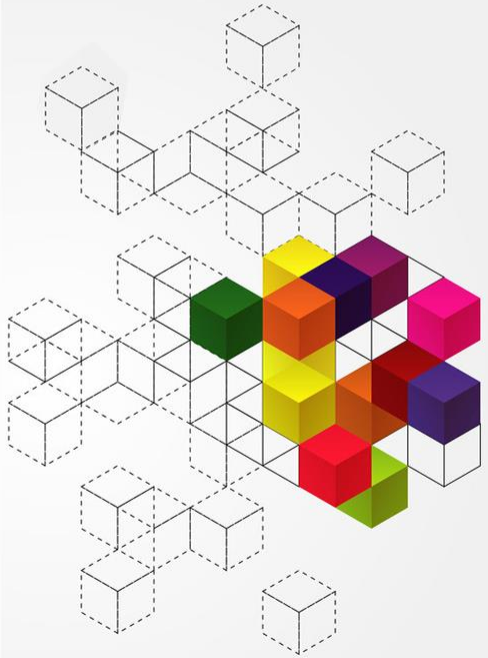
Operating system

胡燕

大连理工大学

一、练习

某寺庙，有小，老和尚若干，由小和尚提水倒入缸供老和尚饮用。水缸可容10桶水，水取自同一井中。水井窄，每次只能容纳1个桶取水。水桶总数为3个。每次从缸中取水，向缸中倒入水仅为1桶，且不可同时进行。试给出有关取井水、入缸水，取缸水的算法。



内容纲要

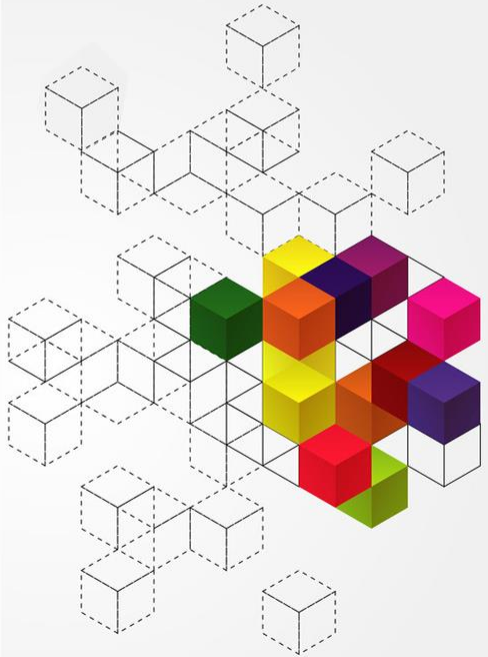
6.7 生产者消费者问题

一、生产者消费者问题简介

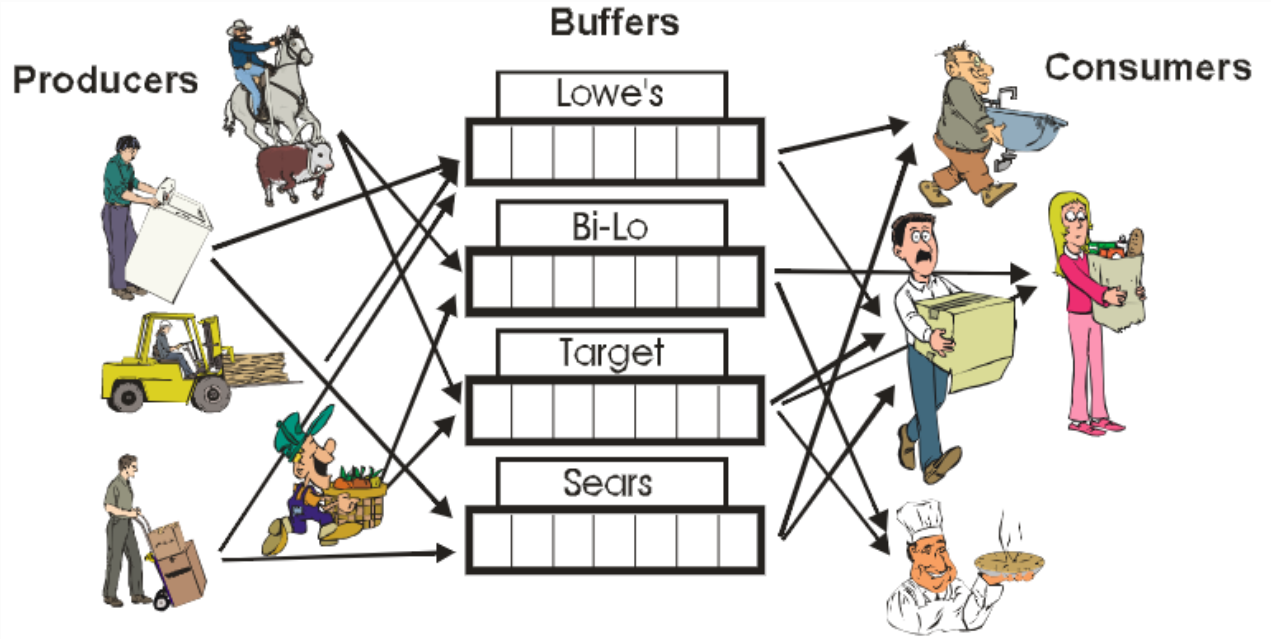
二、生产者消费者问题基础款

三、生产者消费者问题扩展版

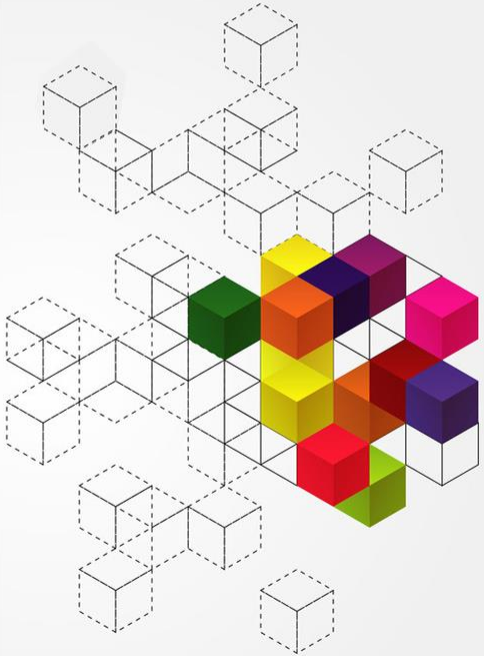
四、生产者消费者问题完整版



一、生产者消费者问题简介

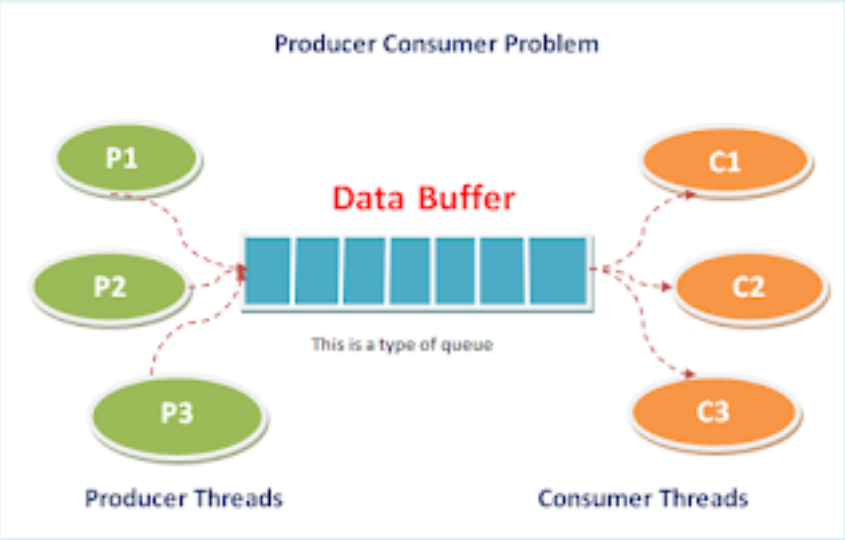


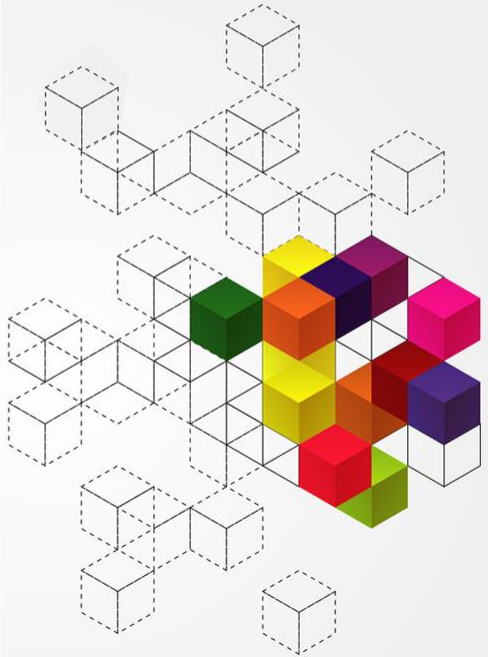
Producers-Stores-Consumers



一、生产者消费者问题简介

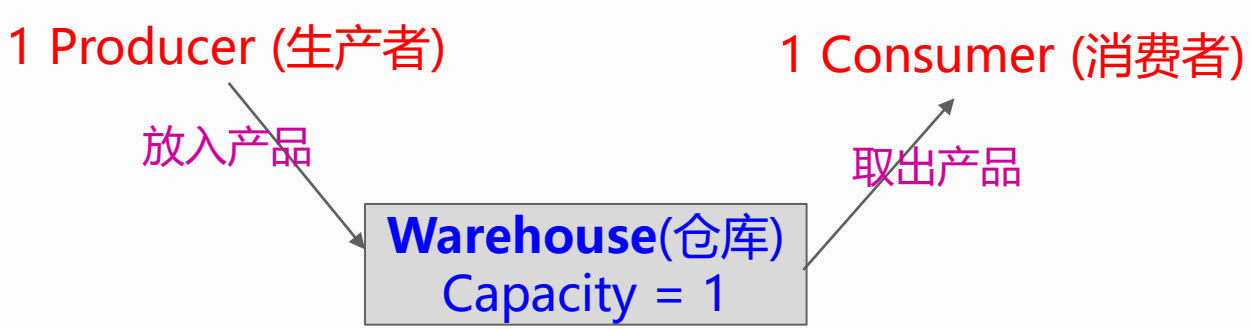
- 生产者消费者问题（Producer Consumer Problem）
  - 又称Bounded Buffer Problem
  - 给定有限大小的缓冲区，生产者将生产出的产品放入缓冲区，消费者从缓冲区取出产品



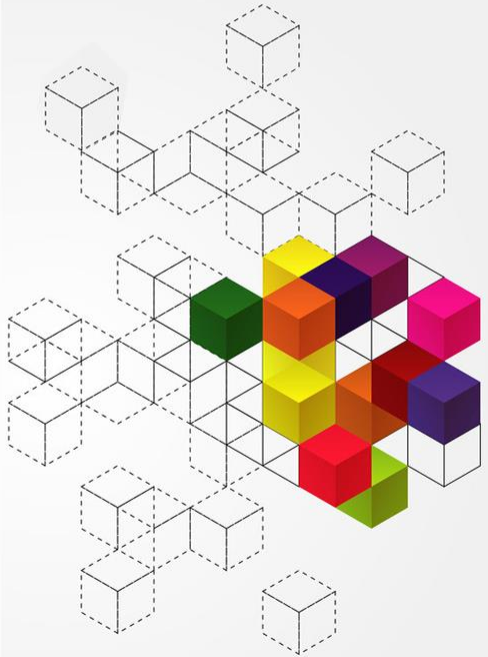


二、生产者消费者问题基础款

- 最简化的版本
  - 1个生产者，1个消费者，共享缓冲区大小=1

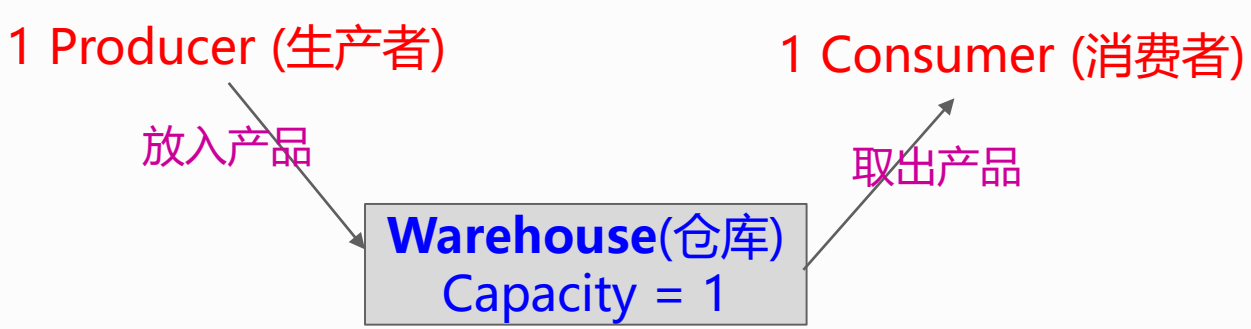


- 进程协作关系分析步骤
  1. 理解问题本质，列出涉及的进程
  2. 分析进程的协作关系
  3. 根据进程协作关系设立信号量，并在程序合理位置通过P/V操作施加并发控制



二、生产者消费者问题基础款

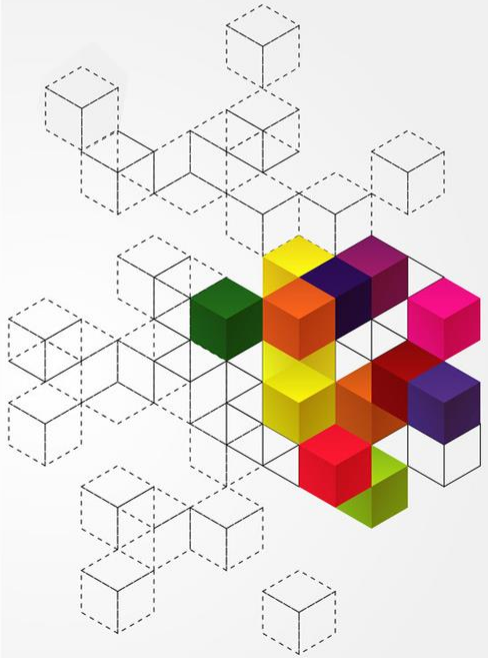
- 最简化的版本
  - 1个生产者，1个消费者，共享缓冲区大小=1



- 步骤1: 列出问题所涉及的进程

```
P:
while (true) {
    生产一个产品;
    将产品放入缓冲区;
};
```

```
C:
while (true) {
    从缓冲区取产品;
    消费产品;
};
```

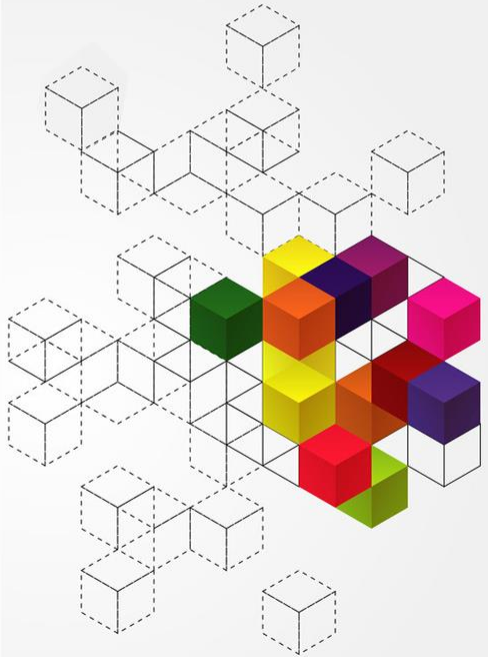


二、生产者消费者问题基础款

- 最简化的版本

```
P:
while (true) {
    生产一个产品;
    将产品放入缓冲区; (put)
};
```

```
C:
while (true) {
    从缓冲区取产品; (get)
    消费产品;
};
```



- 步骤2: 分析进程协作关系
  - 2.1 进程P要执行put操作，要确定缓冲区内有空位存放产品，而除了初始状态之外，这种空位需要进程C的get操作（消费）来创造
  - 2.2 进程C要执行get操作，要确定缓冲区内有存放至少1个产品，产品需要进程P通过生产并通过put操作放入缓冲区

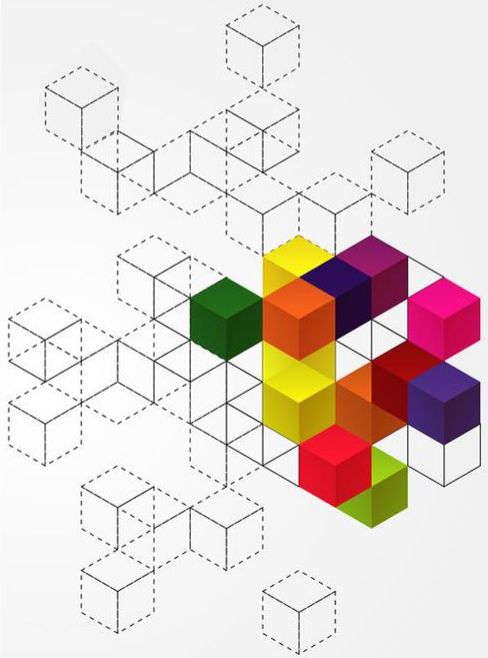
二、生产者消费者问题基础款

- 最简化的版本

```
P:
while (true) {
    生产一个产品;
    将产品放入缓冲区; (put)
};
```

```
C:
while (true) {
    从缓冲区取产品; (get)
    消费产品;
};
```

- 步骤3: 设立信号量，通过P/V操作施加并发控制
  - 信号量empty=1
  - 信号量full=0
  - 信号量的P/V操作应该加到进程合理的位置

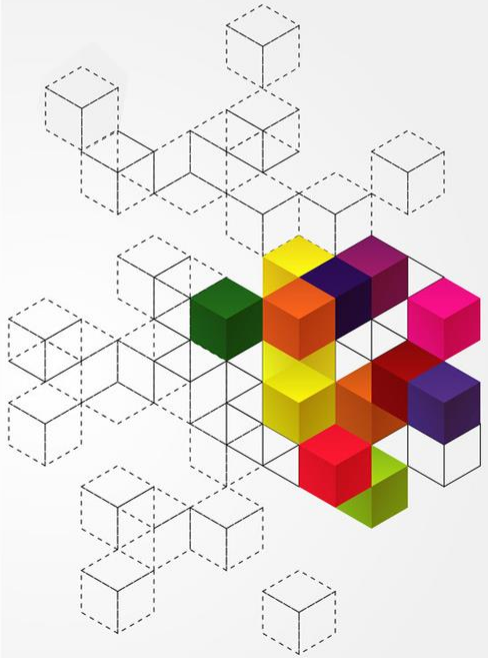


二、生产者消费者问题基础款

- 最简化的版本
  - 基于信号量的解决方案

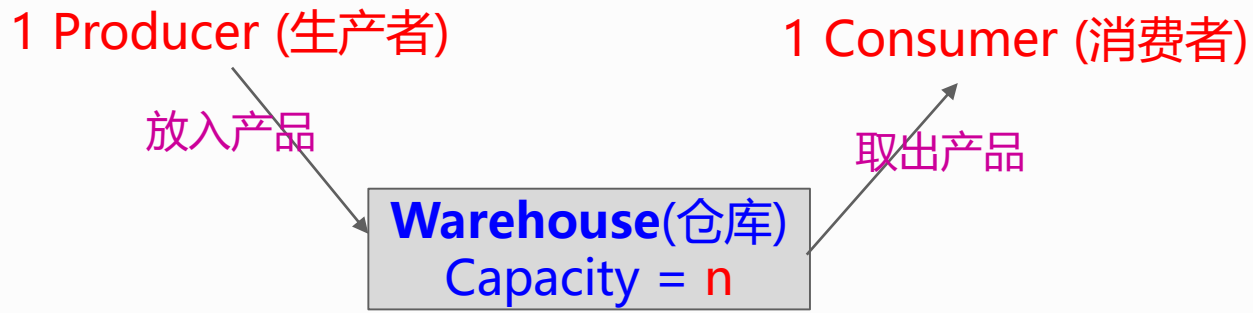
```
int empty = 1;

P:
while (true) {
    生产一个产品;
    P(empty) ;
    送产品到缓冲区;
    V(full);
};
```

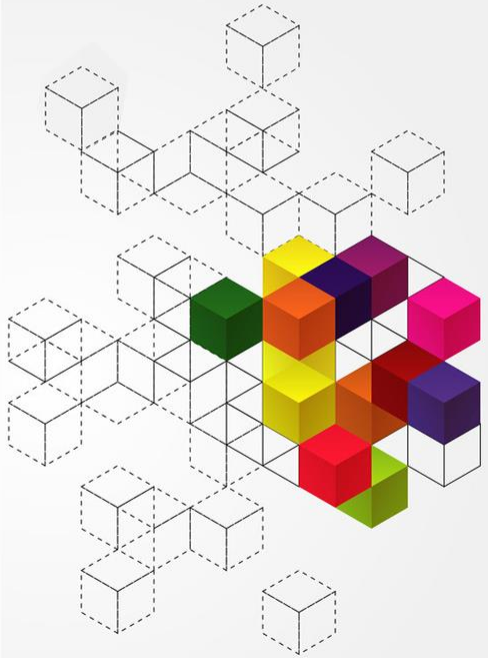
```
int full = 0;

C:
while (true) {
    P(full);
    从缓冲区取产品;
    V(empty);
    消费产品;
};
```



三、生产者消费者问题扩展版

- **扩展版：缓冲区大小 1 -> n**



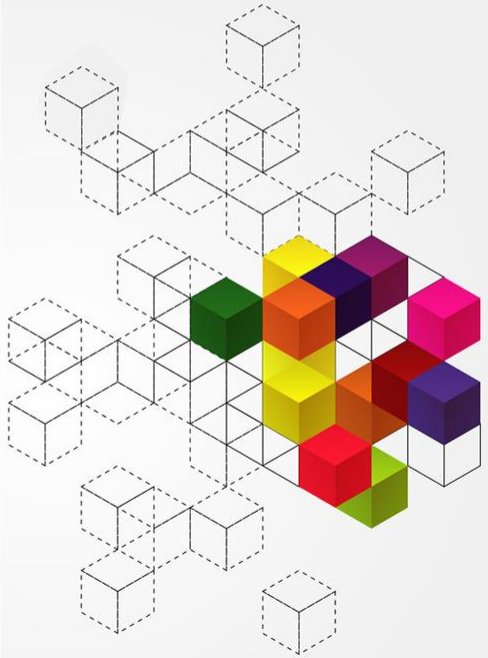


三、生产者消费者问题扩展版

- **扩展版：缓冲区大小 1 -> n**

```
int empty = n;
P:
    i = 0;
    while (true)  {
      生产产品;
      P(empty);
      往Buffer [i]放产品;
      i = (i+1) % n;
      V(full);
    };
```

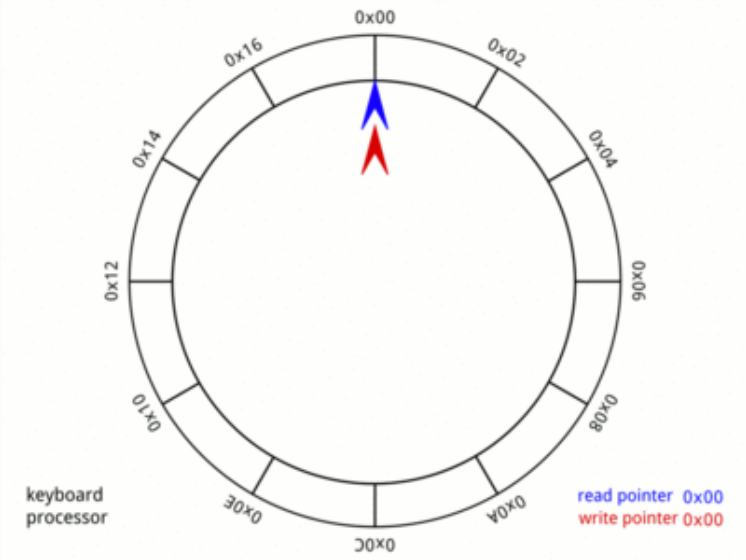
```
int full = 0;
C:
    j =  0;
    while (true)  {
        P(full);
        从Buffer[j]取产品;
        j = (j+1) % n;
        V(empty);
        消费产品;
    };
```



讨论：对环形缓冲的两条伪码语句，需不需要加互斥锁？为什么？

- **问题模式（1:1:n）**
  - **1 producer, 1 consumer, warehouse with capacity n(buffer size=n)**

Solution:

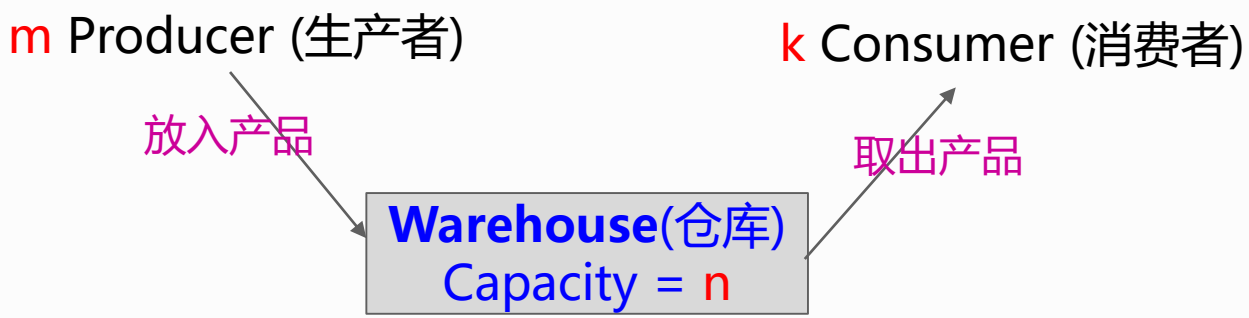


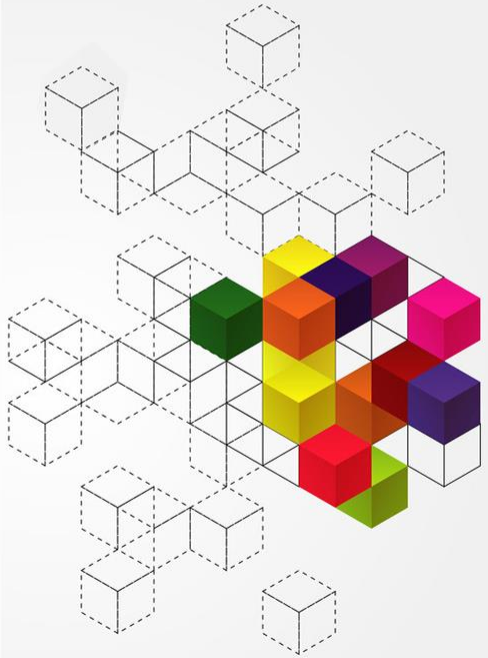

```
P：
        i = 0;
        while (true)  {
          生产产品;
          P(empty);
          往Buffer [i]放产品;
         i = (i+1) % n;
          V(full);
    };
```

```
C：
    j =  0;
    while (true)  {
        P(full);
        从Buffer[j]取产品;
        j = (j+1) % n;
        V(empty);
        消费产品;
    };
```

四、生产者消费者问题完整版

- **完整版：**
  - m个生产者，k个消费者，缓冲区大小=n



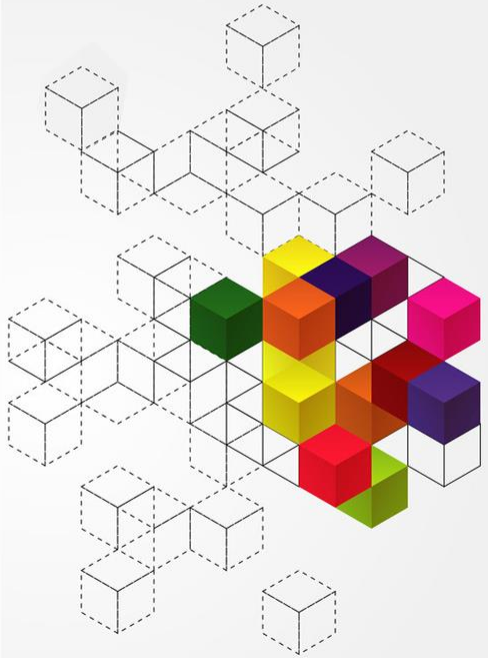


四、生产者消费者问题完整版

- **完整版：**
  - m个生产者，k个消费者，缓冲区大小=n

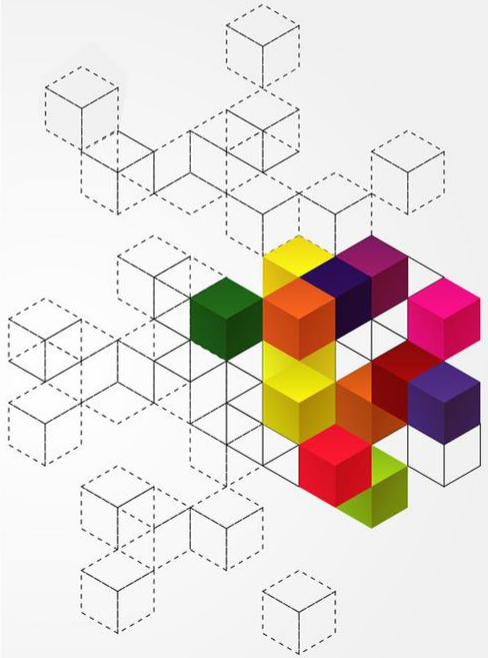
```
P:
    while (true) {
      生产产品;
      P(empty);
      P(mutex);
      往Buffer [i]放产品;
      i = (i+1) % n;
      V(mutex);
      V(full);
    };
```

```
C:
while (true) {
    P(full);
    P(mutex);
    从Buffer[j]取产品;
    j = (j+1) % n;
    V(mutex);
    V(empty);
    消费产品;
};
```



本讲小结

- 生产者消费者问题简介
- 基于信号量的生产者问题解法



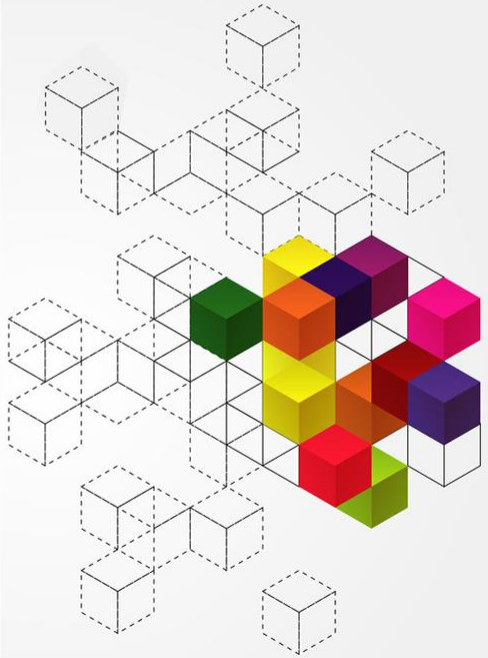
内容纲要

6.8 读者写者问题

一、读者写者问题简介

二、进程协作关系分析

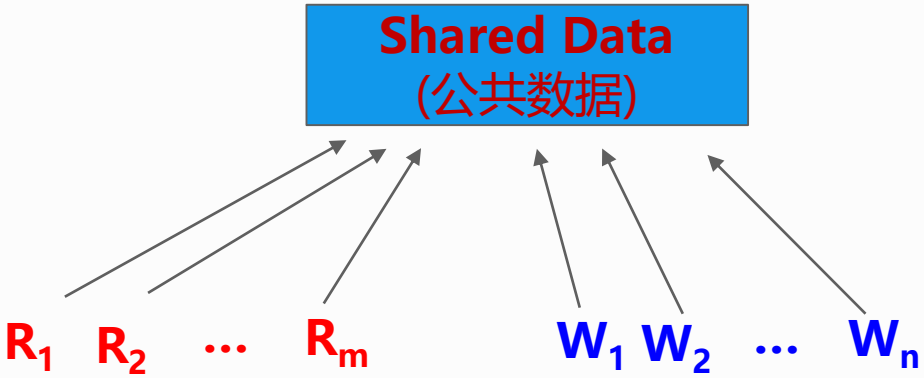
三、读者写者问题的同步解法

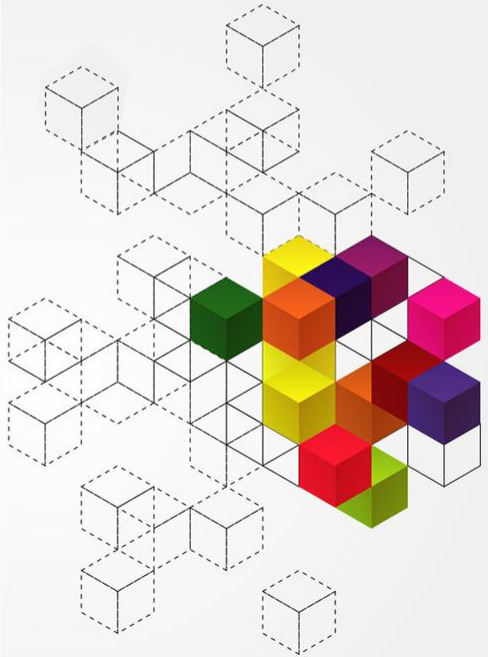


一、读者写者问题简介

一、读者写者问题简介

- **Reader-Writers Problems**
  - 公共数据
  - Reader
  - Writer



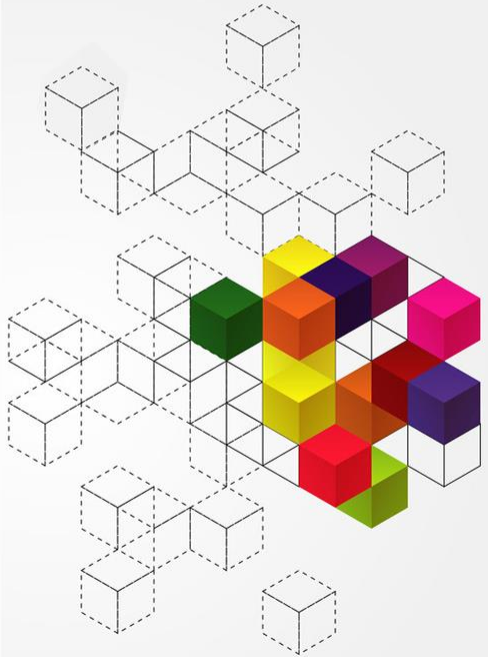


二、进程协作关系分析

步骤1：读者-写者进程表示

```
R:
    while (true) {
        read();
    }
```

```
W:
while (true) {
    write();
}
```



步骤2：进程协作关系分析

- 读操作可以同时进行（R-R，共享读）
- 读操作和写操作不可以同时进行（R-W，互斥）
- 写操作和写操作不可以同时进行（W-W，互斥）

三、同步解法

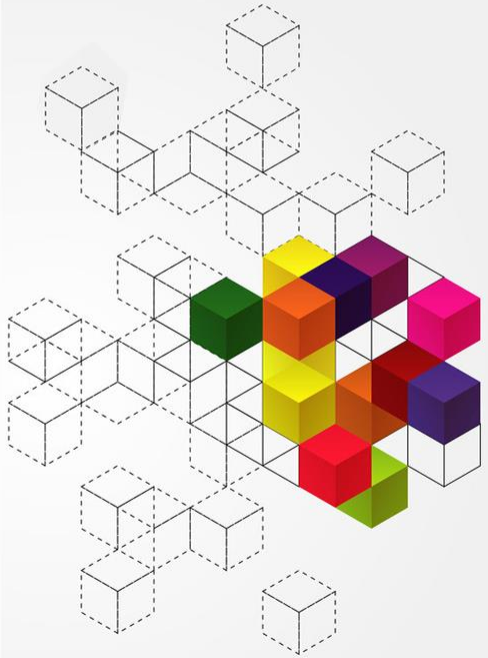
- Try1: 使用信号量mutex对读写操作进行互斥保护

```
R:
    while (true)  {
        P(mutex);
        read();
        V(mutex);
    }
```

```
W:
while (true)  {
    P(mutex);
    write();
    V(mutex);
}
```

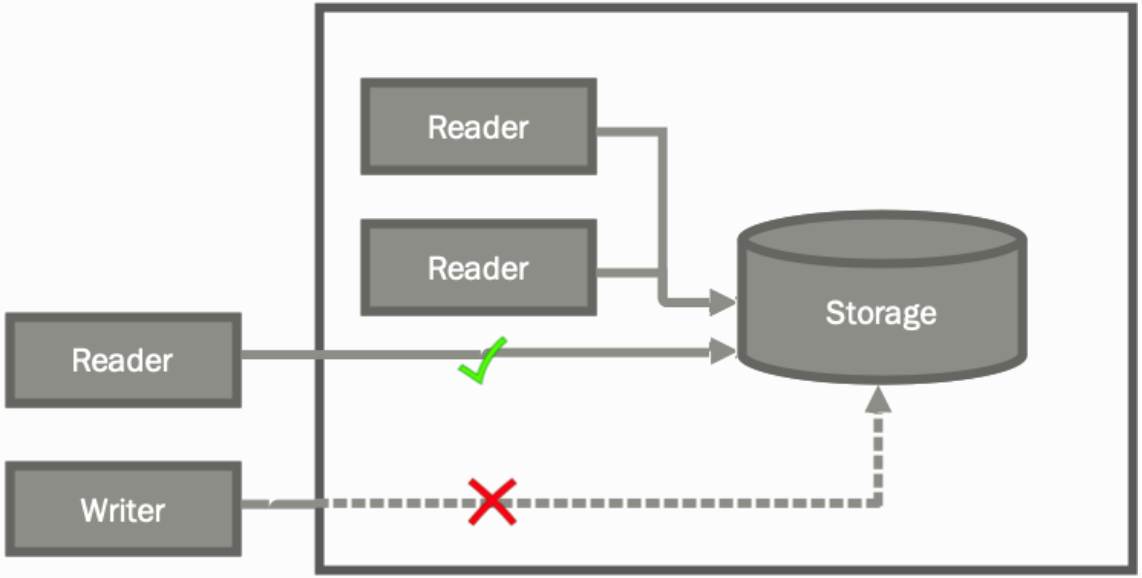
- mutex初值为1

- 是否正确解决问题?

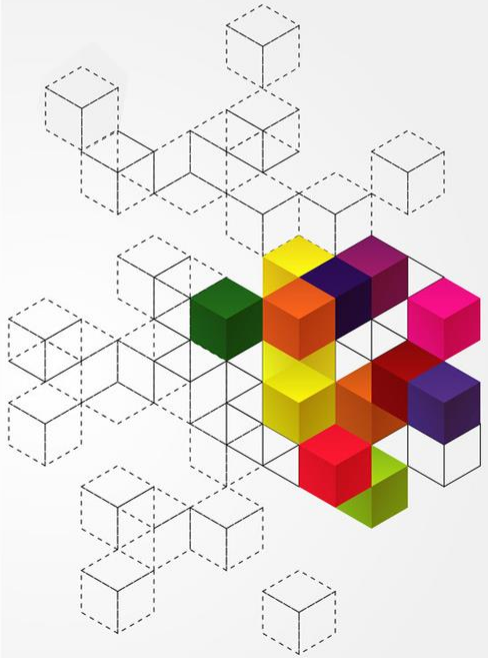


三、同步解法

- Try1: 使用信号量mutex对读写操作进行互斥保护
  对读者之间施加了互斥，过于严苛



- 对try1进行改进，重点考虑保证R-R共享



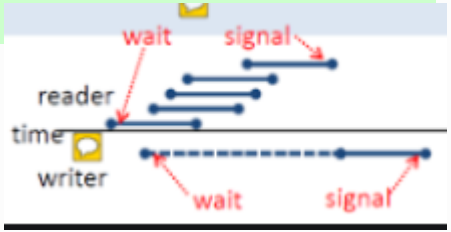
三、同步解法

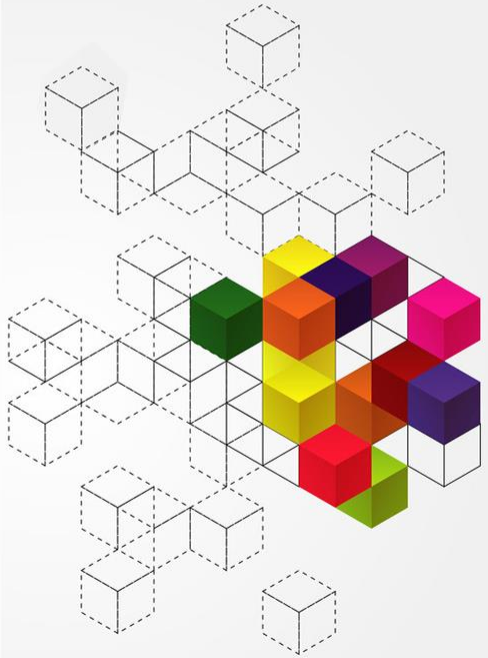
- Try2: 通过引入引用计数保证读者共享

```
Reader:
  while (true)  {
     P(r_mutex);
     r_cnt++;
     if(r_cnt==1)
        P(mutex);
     V(r_mutex);
     read();
     P(r_mutex);
     r_cnt- -;
     if(r_cnt==0)
        V(mutex);
     V(r_mutex);
  }
```

```
Writer:
while (true)  {
    P(mutex);
    write();
    V(mutex);
};
```

问题：Writer Starvation

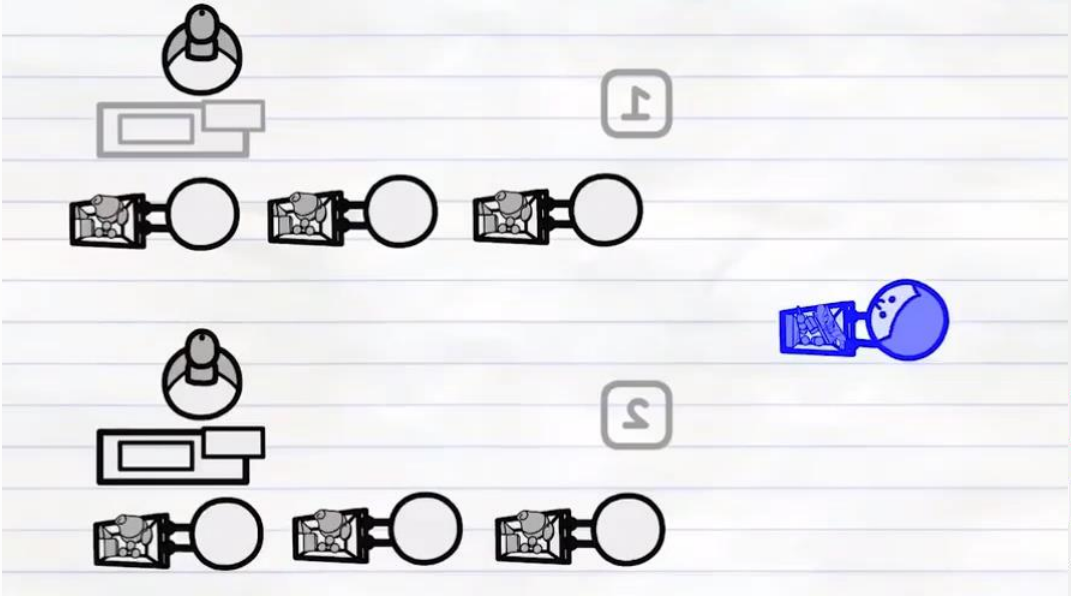




- 尝试2

```
Reader：
 while (true)  {
    P(r_mutex);
    r_cnt++;
    if(r_cnt==1)
        P(mutex);
    V(r_mutex);
    read();
    P(r_mutex);
    r_cnt- -;
    if(r_cnt==0)
        V(mutex);
    V(r_mutex);
}
```

```
Writer：
while (true)  {
    P(mutex);
    write();
    V(mutex);
};
```



Deficiencies: Starvation of Writers (写者饥饿)

三、同步解法

- Try3: 引入额外的rw_mutex用于R-W竞争

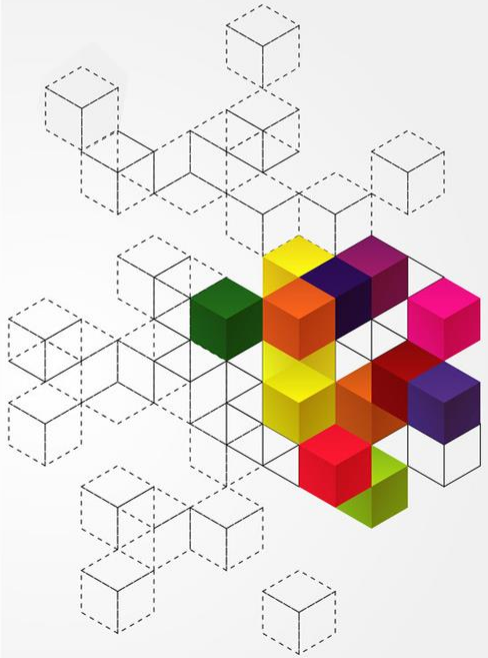
  Final Solution

R:

```
while (true)  {
    P(rw_mutex);
    P(r_mutex);
    r_cnt++;
    if(r_cnt==1)
     P(mutex);
    V(r_mutex);
    V(rw_mutex);
    read();
    P(r_mutex);
    r_cnt- -;
    if(r_cnt==0) V(mutex);
    V(r_mutex);
}
```

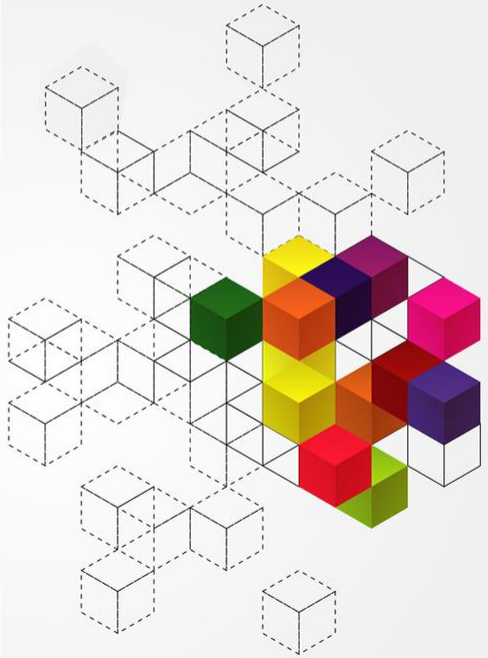
W:

```
while (true)  {
    P(rw_mutex);
    P(mutex);
    write();
    V(mutex);
    V(rw_mutex);
};
```



本讲小结

- 读者写者问题同步分析
- 基于信号量的读者写者问题同步解法



信号量应用问题求解步骤

- 进程结构表示
- 同步关系分析
- 确定信号量
- 明确信号量初值
- 使用P/V操作实施同步控制

P/V操作安排

生产者进程:

P(empty)

⋮

V(full)

消费者进程:

P(full)

⋮

V(empty)

同步

读者进程:

P(rw_mutex)

⋮

if r_count==1
P(mutex)

V(rw_mutex)

if r_count==0
V(mutex)

写者进程:

P(rw_mutex)

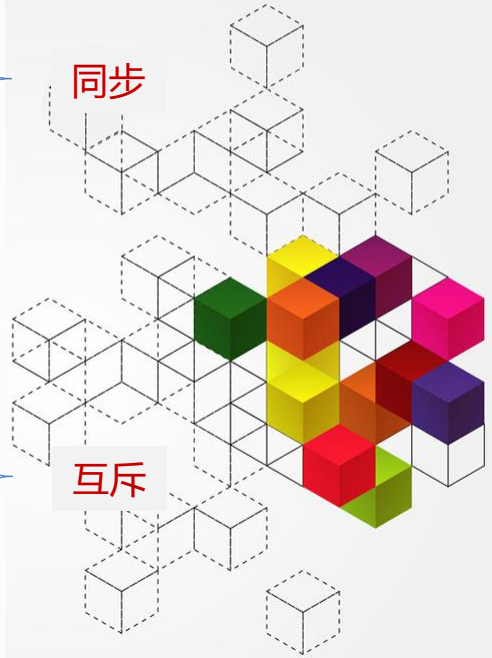
P(mutex)

⋮

V(mutex)

V(rw_mutex)

互斥



提升途径

- 利用Linux环境的sysvipc对象，如共享内存区、信号量等，编程实现典型的同步问题

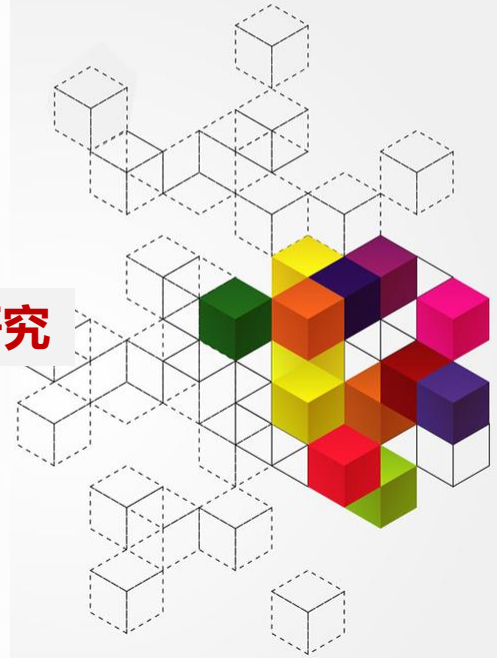
实验检验能力提升

- 分析Linux中的信号量实现

提升系统分析能力

- 了解并发问题的分析与测试方法
  - 阅读文献，了解分布式系统中的并发问题

了解并发控制相关研究



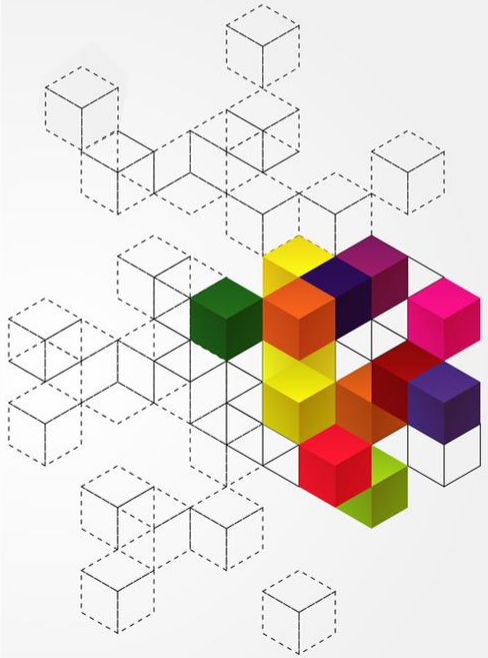
内容纲要

6.9 管程

一、管程概念

二、管程的三种语义

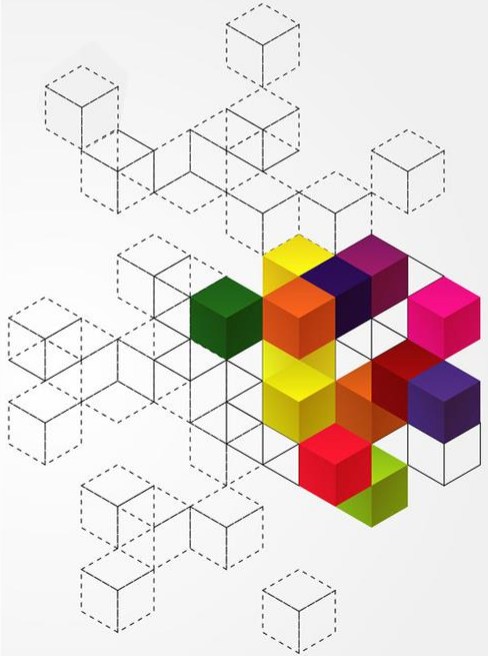
三、管程应用示例



一、管程概念

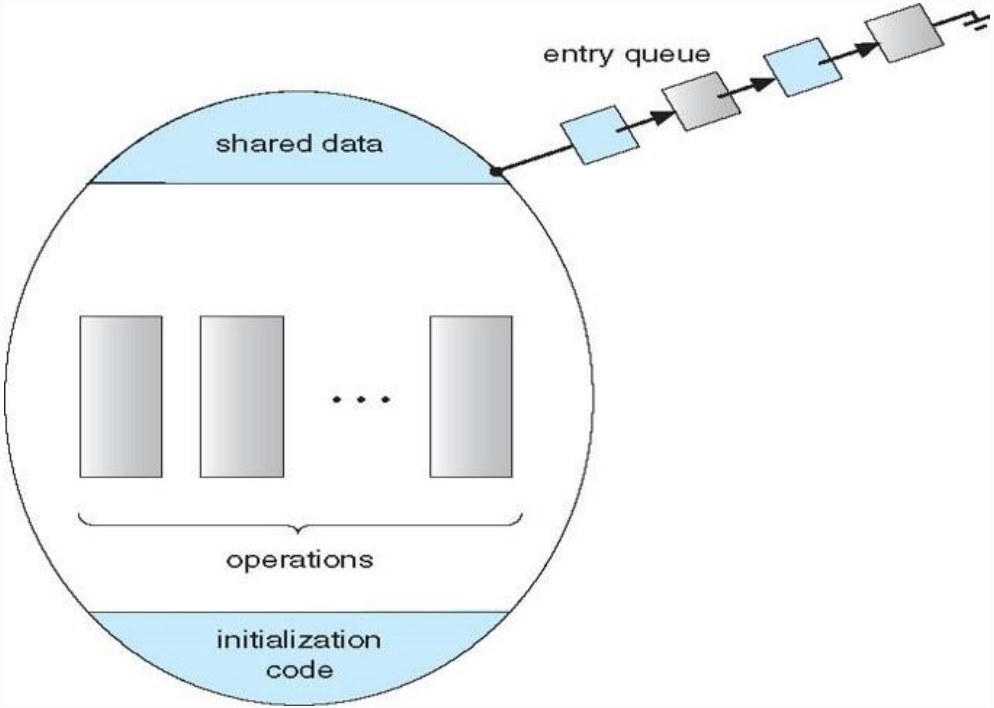
管程的基本思想

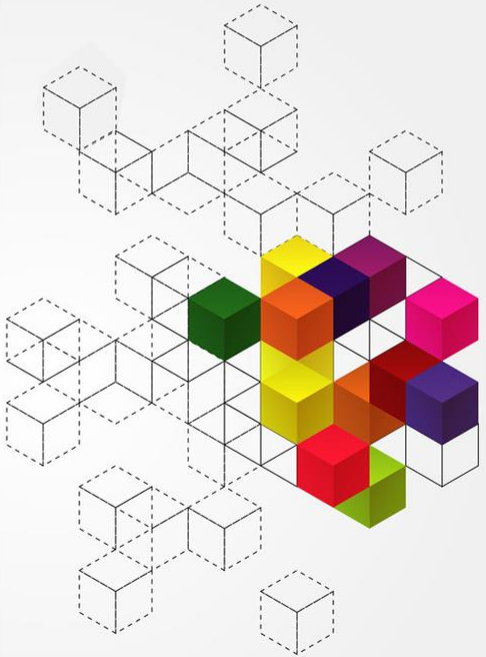
- 把分散在各进程中的临界区集中起来进行管理，将系统中的临界资源用数据结构表示
- 建立一个“秘书”程序来管理对临界资源的访问。“秘书”每次仅允许一个进程来访，如此，既便于对临界资源的管理，又实现对资源的互斥访问。“秘书”就是后来的管程。
- 利用管程，对资源的管理可借助数据结构及在其上实施的若干过程来进行；对共享资源的申请和释放，通过过程在数据结构上的操作来实现
- 管程被请求和释放资源的进程所调用



一、管程概念

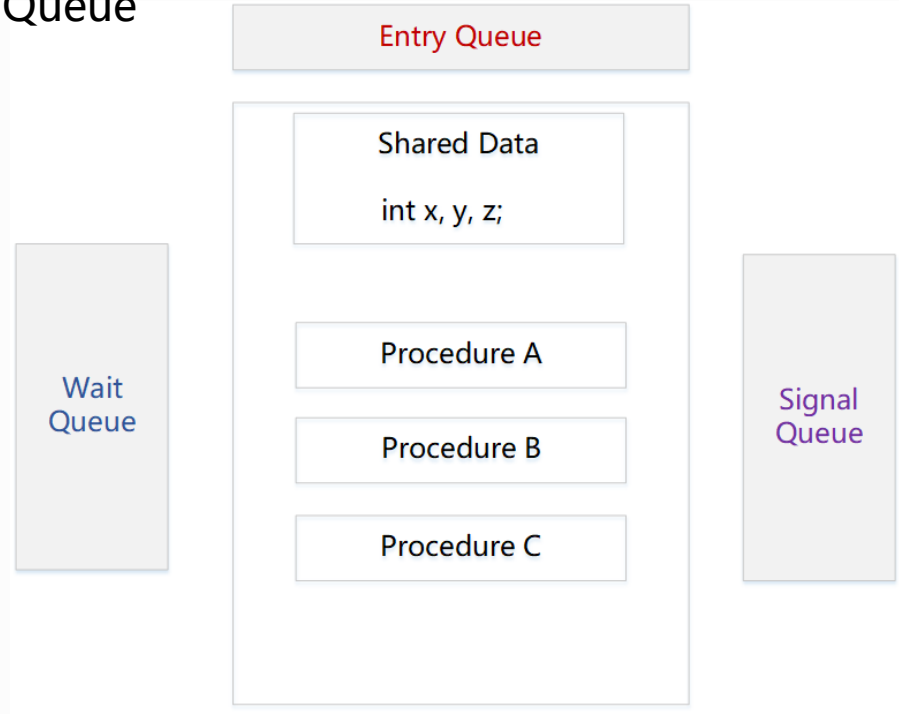
- 管程构成示意图

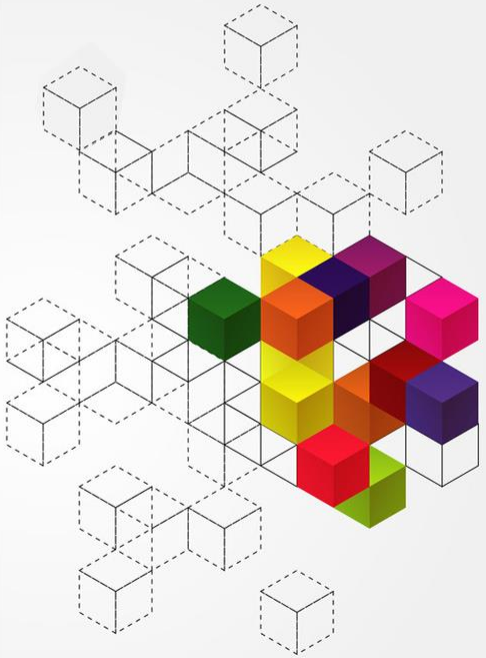




一、管程概念

- 管程实现中两个关键队列
  - Wait Queue
  - Signal Queue

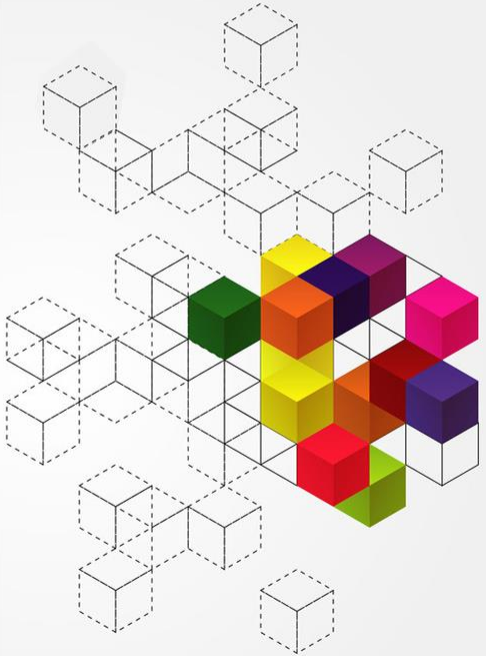




二、管程的三种语义

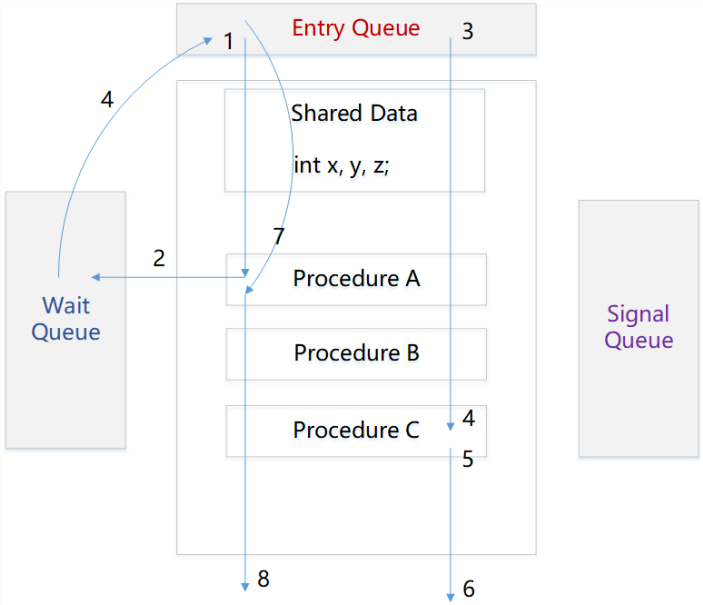
管程的三种不同实现语义：

Mesa语义
Hoare语义
Brinch Hanson语义

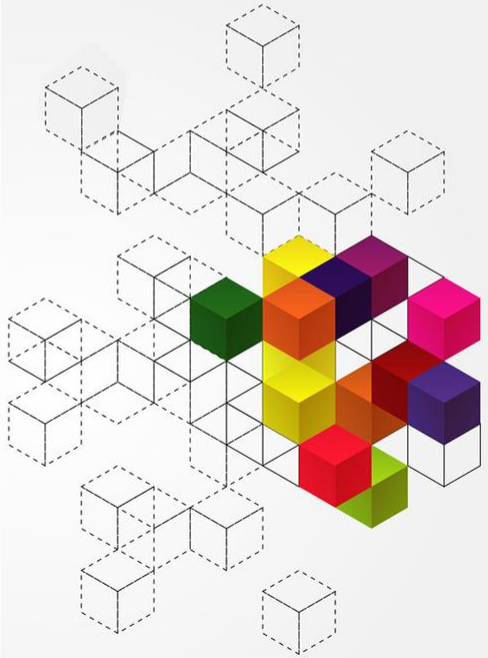


二、管程的三种语义

Mesa Semantics：

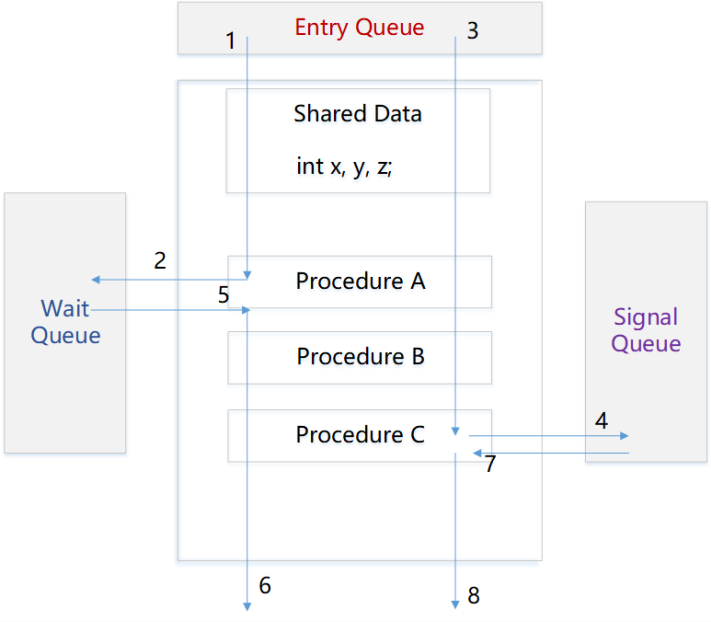


1. 线程A进入管程
2. 线程A等待某个资源
3. 线程B进入管程
4. 线程B释放线程A等待的资源，线程A被转到Entry Queue，而线程B继续执行
5. 线程B继续在管程内执行
6. 线程B离开管程
7. 线程A重新进入管程
8. 线程A离开管程
9. 其他线程可以继续进入管程

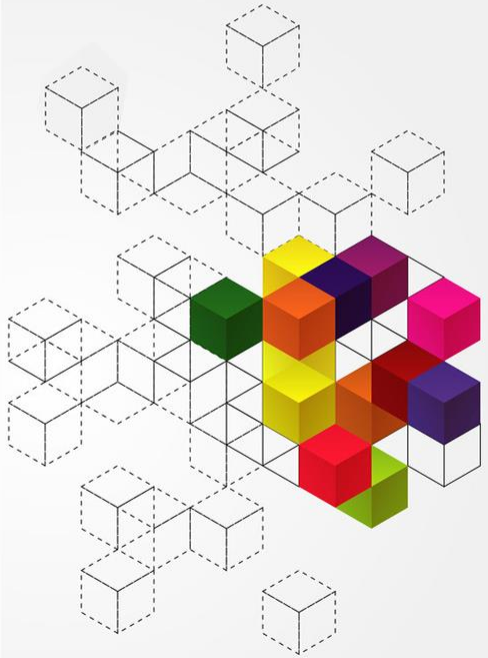


二、管程的三种语义

Hoare Semantics：

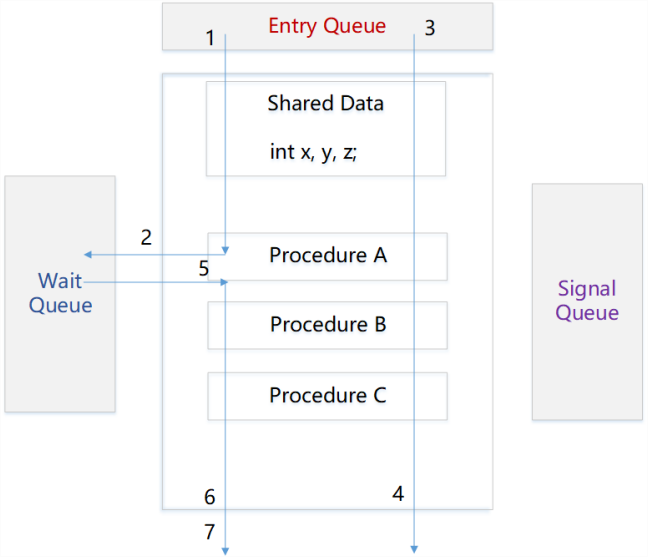


1. 线程A进入管程
2. 线程A等待某个资源
3. 线程B进入管程
4. 线程B释放线程A等待的资源，唤醒线程A，而线程B进入Signal Queue
5. 线程A重新进入管程继续执行
6. 线程A离开管程
7. 线程B重新进入管程
8. 线程B离开管程
9. 其他线程可以继续进入管程

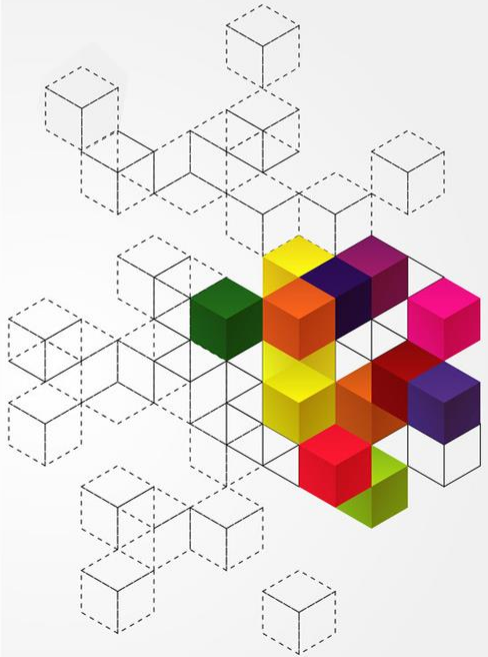


二、管程的三种语义

Brinch Hanson Semantics:



1. 线程A进入管程
2. 线程A等待某个资源
3. 线程B进入管程
4. 线程B发资源已释放信号给线程A，随后线程B离开管程
5. 线程A重新进入管程继续执行
6. 线程A离开管程
7. 其他线程可以继续进入管程



三、管程应用示例

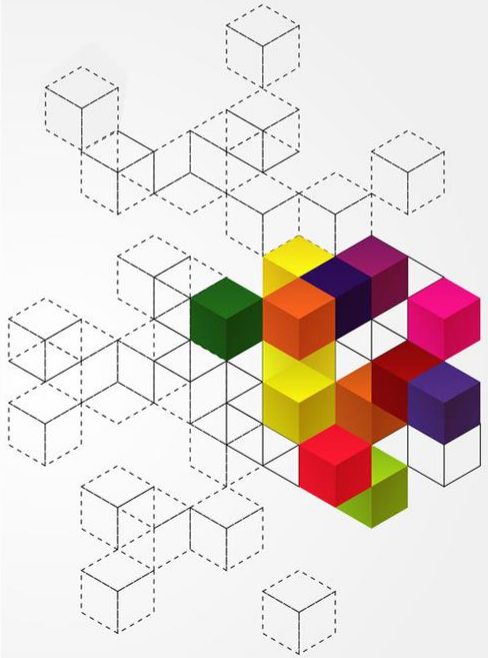
```
monitor PC {
    condition : full, empty;
    int : count = 0;

    entry put {
        if (count==max) wait (full);
        insert item
        count = count+1;
        if (count==1) signal (empty);
    }


    entry get {
        if (count==0) wait (empty);
        remove item
        count = count-1;
        if (count==max-1) signal (full);
    }

}
```

producer process

```
while (TRUE) {
    produce item
    PC.put;

}
```

consumer process

```
while (true) {
    PC.get;
    consume item
}
```



本讲小结

- 管程概念
- 管程的实现语义
- 管程应用示例