

## 上节课重点：

1. 面包店算法：解决  $n$  个进程的临界区问题

2. 进程互斥的硬件实现方法：

中断屏蔽方法、TestAndSet 指令、Swap 指令

3. 信号量机制 (semaphore)：

整型信号量、

记录型信号量、

AND 型信号量、

信号量集



## 2.4.4 信号量的应用

### 1. 利用信号量实现互斥：

- 为临界资源设置一个互斥信号量 *mutex*，其初值为 1；在每个进程中将临界区代码置于 `wait(mutex)` 和 `signal(mutex)` 原语之间

```
semaphore mutex = 1;
```

```
...
```

```
do{
```

```
...
```

```
wait ( mutex );
```

```
critical section
```

```
signal( mutex );
```

```
remainder section
```

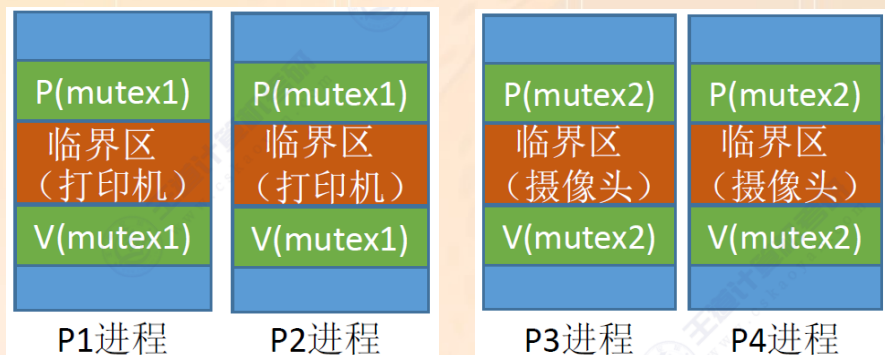
```
}while (true);
```

## 2.4.4 信号量的应用

### 信号量机制实现进程互斥

1. 分析并发进程的关键活动，划分临界区（如：对临界资源打印机的访问就应放在临界区）
2. 设置互斥信号量 `mutex`，初值为 1。
3. 在进入区 `P(mutex)`-- 申请资源
4. 在退出区 `V(mutex)`-- 释放资源

注意：对不同的临界资源需要设置不同的互斥信号量。  
**P、V 操作必须成对出现。**缺少 `P(mutex)` 就不能保证临界资源的互斥访问，等待进程永不被唤醒。



```
/*记录型信号量的定义*/
typedef struct {
    int value;           // 剩余资源数
    struct process *L;   // 等待队列
} semaphore;
```

```
/*信号量机制实现互斥*/
semaphore mutex=1; // 初始化信号量

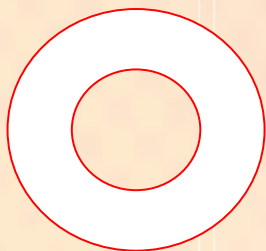
P1(){
    ...
    P(mutex);       // 使用临界资源前需要加锁
    临界区代码段...
    V(mutex);       // 使用临界资源后需要解锁
    ...
}

P2(){
    ...
    P(mutex);
    临界区代码段...
    V(mutex);
    ...
}
```

- **注意：**
  - **wait(mutex) 和 signal(mutex) 必须成对地出现。**
  - **缺 wait(mutex) 将会引起系统混乱，不能保证对临界资源的互斥访问**
  - **缺 signal(mutex) 将会使该临界资源永久不被释放**

# 初始状态

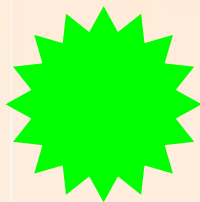
临界区



`mutex := 1`

没有并发进程使用临界区

互斥的进程

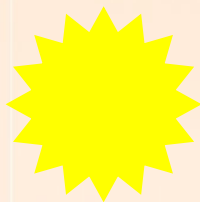
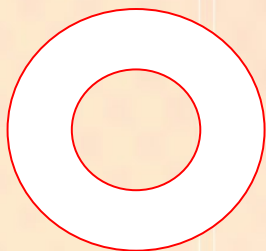




# 一个进程申请临界区

mutex:=1

没有并发进程使用临界区



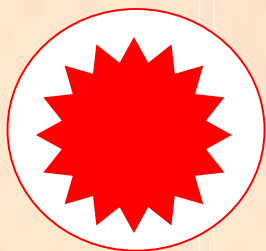
# 申请成功，进程使用临界区

mutex:=0



# 另一个进程也申请临界区

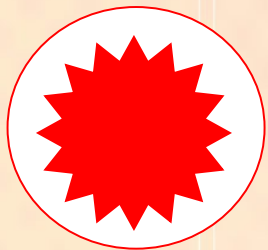
mutex: =0



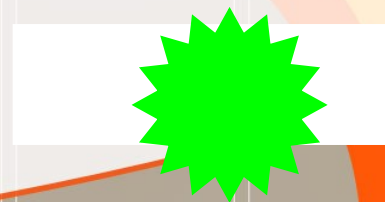


# 申请失败

mutex: **-1**

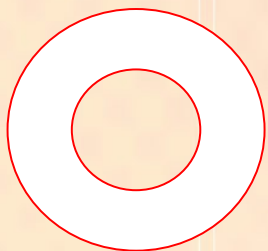


## 阻塞队列



# 释放资源

mutex:=0

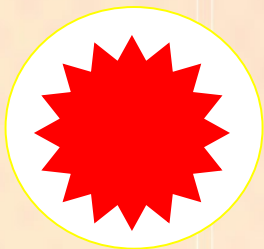


## 阻塞队列



# 释放资源

mutex := 0



## 阻塞队列



## 2. 利用信号量实现同步

**进程同步：**要让各并发进程按要求有序地推进。

```
P1(){  
    代码1;  
    代码2;  
    代码3;  
}
```

```
P2(){  
    代码4;  
    代码5;  
    代码6;  
}
```

比如：P1、P2 并发执行，由于存在异步性，因此二者交替推进的次序是不确定的。

若 P2 的“代码 4”要基于 P1 的“代码 1”和“代码 2”的运行结果才能执行，那么我们就必须保证“代码 4”一定是在“代码 2”之后才会运行。

这就是进程同步的问题，让本来异步并发的进程互相配合，有序推进。

## 信号量机制实现进程同步

用信号量实现进程同步：

1. 分析在什么地方需要实现“同步关系”，即保证“**一前一后**”执行的两个操作（或两句代码）
2. 设置同步信号量  $S$ ，初始为 0
3. 在“前操作”之后执行  $V(S)$
4. 在“后操作”之前执行  $P(S)$

*/\*信号量机制实现同步\*/*

`semaphore S=0; // 初始化同步信号量, 初始值为0`

```
P1(){  
    代码1;  
    代码2;  
    V(S);  
    代码3;  
}
```

释放资源

```
P2(){  
    P(S);  
    代码4;  
    代码5;  
    代码6;  
}
```

若先执行  $V(S)$  操作，则  $S++$  后  $S=1$ 。之后当执行到  $P(S)$  操作时，由于  $S=1$ ，表示有可用资源，会执行  $S--$ ， $S$  的值变回 0。P2 进程不会执行 block 原语，而是继续往下执行代码 4。

若先执行到  $P(S)$  操作，由于  $S=0$ ， $S--$  后  $S=-1$ ，表示此时没有可用资源，因此 P2 操作中会执行 block 原语，主动请求阻塞。

之后当执行完代码 2，继而执行  $V(S)$  操作， $S++$ ，使得  $S$  变回 0。由于此时有进程在该信号量对应的阻塞队列中，因此会在  $V$  操作中执行 wakeup 原语。

保证了 代码 4 一定是在 代码 2 之后执行。



## 信号量机制实现前驱关系

进程 P1 中有句代码 S1，P2 中有句代码 S2，P3 中有句代码 S3.....P6 中有句代码 S6。这些代码要求按照如下前驱图所示的顺序来执行：

其实每一对前驱关系都是一个进程同步问题（需要保证一前一后的操作）  
因此，

1. 要为每一对前驱关系各设置一个同步信号量
2. 在“前操作”之后对相应的同步信号量执行 V 操作
3. 在“后操作”之前对相应的同步信号量执行 P 操作

```
P1() {  
    ...  
    S1;  
    V(a);  
    V(b);  
    ...  
}
```

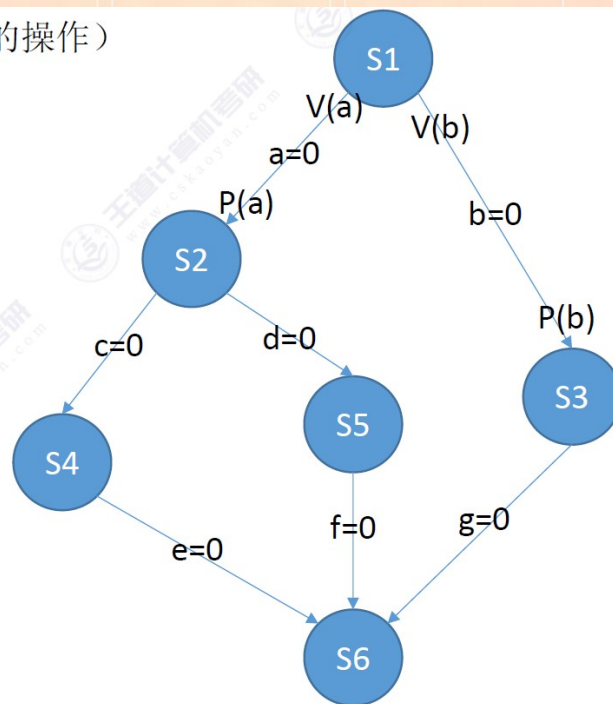
```
P2() {  
    ...  
    P(a);  
    S2;  
    V(c);  
    V(d);  
    ...  
}
```

```
P3() {  
    ...  
    P(b);  
    S3;  
    V(g);  
    ...  
}
```

```
P4() {  
    ...  
    P(c);  
    S4;  
    V(e);  
    ...  
}
```

```
P5() {  
    ...  
    P(d);  
    S5;  
    V(f);  
    ...  
}
```

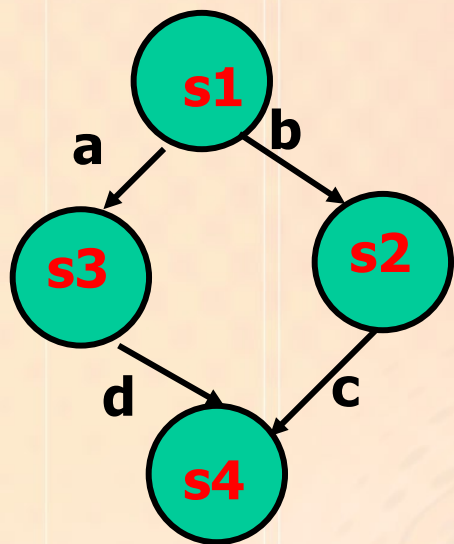
```
P6() {  
    ...  
    P(e);  
    P(f);  
    P(g);  
    S6;  
    ...  
}
```





- 前趋关系：并发执行的进程  $P_1$  和  $P_2$  中，分别有代码  $C_1$  和  $C_2$ ，要求  $C_1$  在  $C_2$  开始前完成；
- 为每个前趋关系设置一个互斥信号量  $S12$ ，其初值为 0
- P1:                      P2:  
    C1;                      wait(s12);  
    signal(s12)              C2 ;

# 进程同步例题



有四个前趋关系，所以定义四个信号量：

**Semaphore a, b, c, d=0;**

**进程 s1:**

```
while (1) {  
    ....  
    signal (a);  
    signal (b);  
}
```

**进程 s2:**

```
while (1) {  
    wait (b);  
    .....  
    signal (c);  
}
```

**进程 s3:**

```
while (1) {  
    wait (a);  
    .....  
    signal (d);  
}
```

**进程 s4:**

```
while (1) {  
    wait (d);  
    wait (c);  
    .....  
}
```



## 2.4.5 管程的基本概念

- 利用信号量实现进程同步，使大量的同步操作分散在各个进程中。使系统管理麻烦，同步操作使用不当会引起死锁。
- 引入新的进程同步工具 - - - 管程 ( Monitors)
- 目的：分离互斥和条件同步的关注

- 管程由四部分组成：
  - 管程的名称
  - 局部于管程的共享变量说明
  - 对该数据结构进行操作的一组过程
  - 对管程中数据设置初值的语句
- 任何管程外的过程都**不能访问**管程内的数据结构。管程相当于围墙，将共享变量和对它进行操作的若干过程围了起来，进程**只要访问临界资源就必须通过管程**。
- 管程每次**只允许一个**进程进入管程，实现了互斥。
- 使用信号量的效率比管程高。
- 管程结构在一些程序设计语言中得到实现。如并发 Pascal 和 Java ， C# 等，它还被作为一个程序库实现。



## • 1. 管程的定义

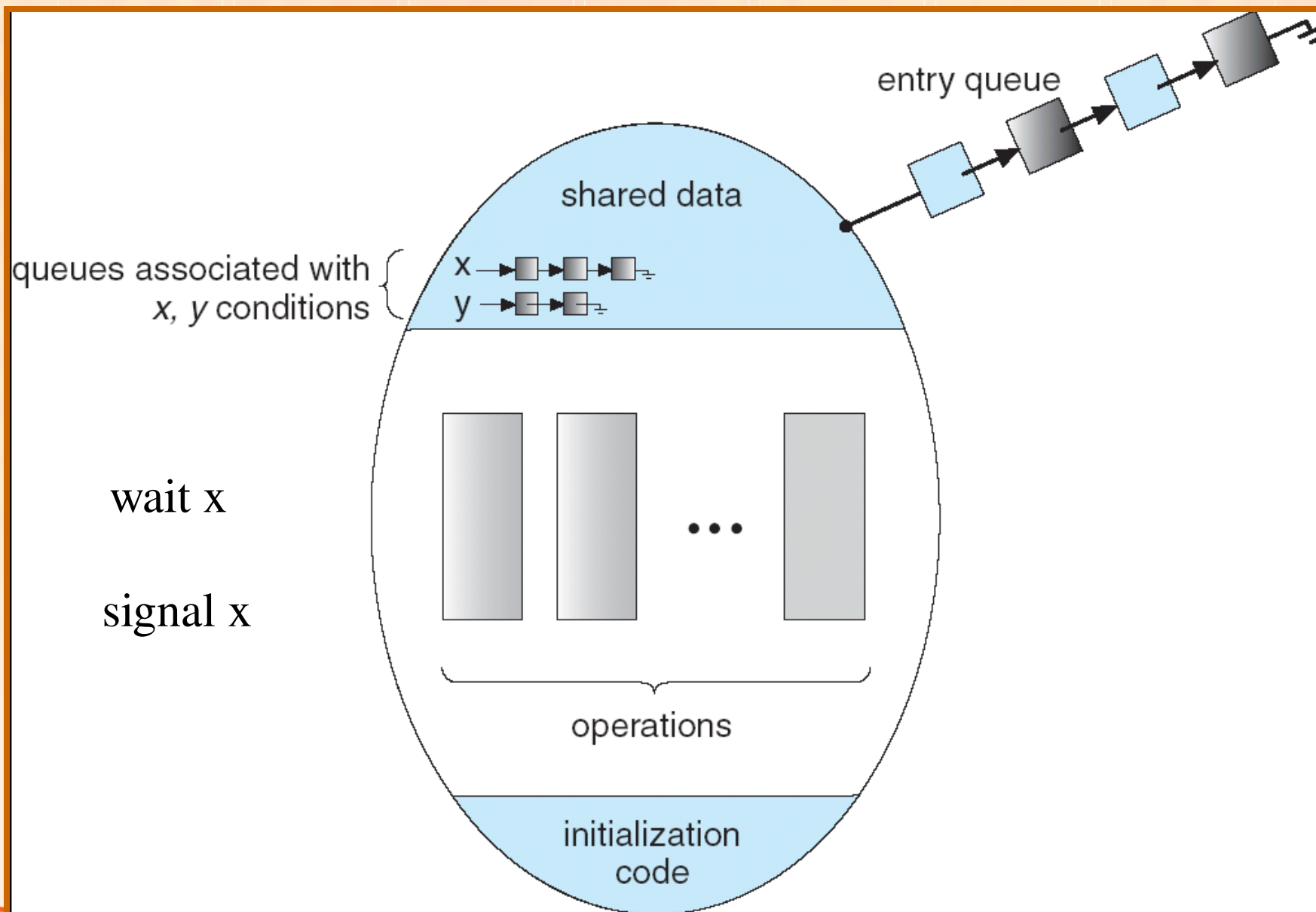
- 系统中各种硬件和软件资源可用抽象数据结构加以描述。
- 如，一台传真机，可用与该资源有关的状态信息 ( busy/free ) 和对它执行的请求和释放操作，以及等待该资源的进程队列来描述。
- 如，一个 FIFO 队列可用队长、队首、队尾以及在该队列上执行的一组操作来描述。
- 当共享资源用共享数据结构表示时，资源管理程序可用对该数据结构进行操作的一组过程来表示。  
如， request、release
- 管程定义了一个数据结构和能为并发进程所执行 ( 在该数据结构上 ) 的一组操作，这组操作能同步进程，改变管程中的数据。



- **条件变量**

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时，要使其等待。
- 为了区别不同的等待原因，设置了条件变量和在条件变量上进行操作的两个同步原语 wait, signal。
- 条件变量说明形式为：condition : x , y;
- 同步原语 wait 使调用进程等待，并将它排在相应的等待队列上；signal 唤醒等待队列的队首进程。使用方式为：x.wait , x.signal。

# 管程与条件变量



# 锁和条件变量

- **Lock**
  - **Lock::Acquire()** – 等待直到锁可用，然后抢占锁
  - **Lock::Release()** – 释放锁，唤醒等待者如果有
- **Condition Variable**
  - 允许等待状态进入临界区
    - 允许处于等待（睡眠）的线程进入临界区
    - 某个时刻原子释放锁进入睡眠
  - **Wait() operation**
    - 释放锁，睡眠，重新获得锁返回后
  - **Signal() operation (or broadcast() operation)**
    - 唤醒等待者（或者所有等待者），如果有

# 条件变量实现

- 实现
  - 需要维持每个条件队列
  - 线程等待的条件等待 `signal()`

```
Class Condition {  
    int numWaiting = 0;  
    WaitQueue q;  
}
```

```
Condition::Wait(lock){  
    numWaiting++;  
    Add this thread t to q;  
    release(lock);  
    schedule(); //need mutex  
    require(lock);  
}
```

```
Condition::Signal(){  
    if (numWaiting > 0) {  
        Remove a thread t from q;  
        wakeup(t); //need mutex  
        numWaiting--;  
    }  
}
```

# 管程例子：生产者 - 消费者问题

```
classBoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition notFull, notEmpty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    while (count == n)  
        notFull.Wait(&lock);  
    Add c to the buffer;  
    count++;  
    notEmpty.Signal();  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
    while (count == 0)  
        notEmpty.Wait(&lock);  
    Remove c from buffer;  
    count--;  
    notFull.Signal();  
    lock->Release();  
}
```



## 2.5 经典的进程同步问题

### • 2.5.1 生产者 - 消费者问题

问题描述：系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用。（注：这里的“产品”理解为某种数据，缓冲区为循环缓冲区）

生产者和消费者共享一个初始为空，大小为  $n$  的缓冲区。

只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。

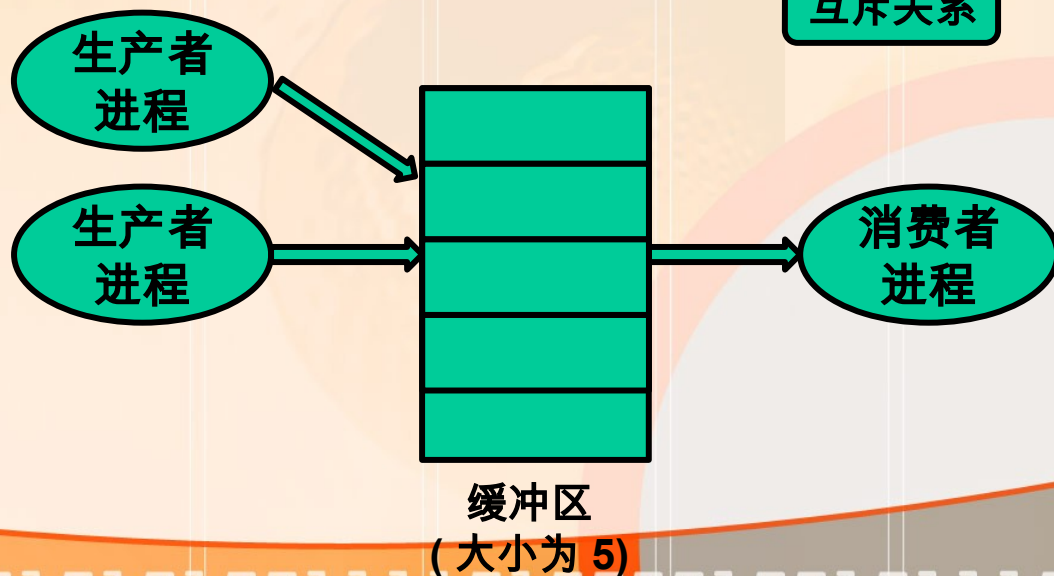
只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。

缓冲区是临界资源，各进程互斥地访问。

缓冲区没满，生产者生产

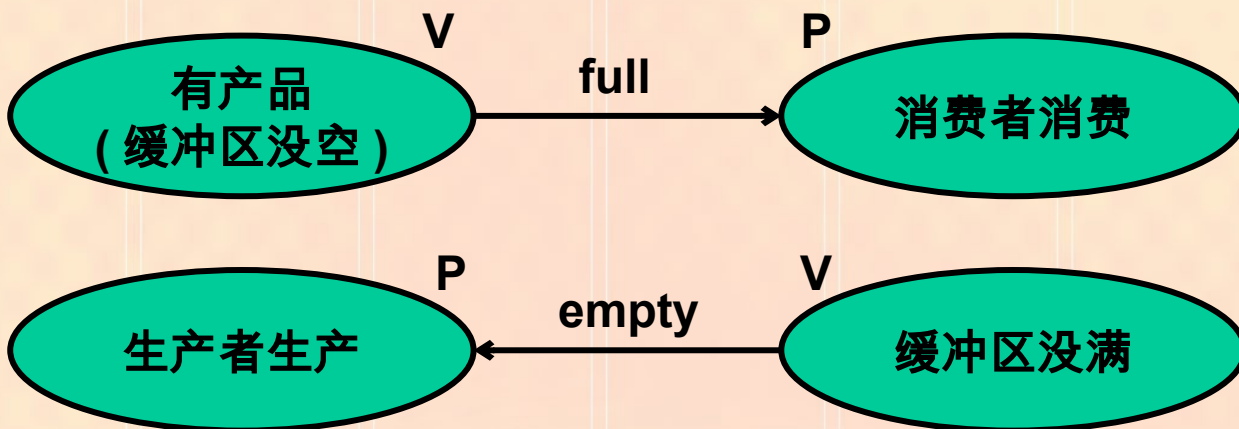
缓冲区没空，消费者消费

互斥关系





## 2.5.1 生产者 - 消费者问题



PV 操作题目分析步骤：

1. 关系分析。找出题目中描述的几个进程，分析它们之间的同步、互斥关系。
2. 整理思路。根据各进程的操作流程确定 P、V 操作的大致顺序。
3. 设置信号量。并根据题目条件确定信号量初值。（互斥信号量初值一般为 1，同步信号量的初始值要看对应资源的初始值是多少）

```
semaphore mutex = 1; // 互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n; // 同步信号量，表示空闲缓冲区的数量
semaphore full = 0. // 同步信号量，表示产品的数量，也是非空缓冲区的数量
```

## 2.5.1 生产者 - 消费者问题

```
semaphore mutex = 1;    // 互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n;     // 同步信号量，表示空闲缓冲区的数量
semaphore full = 0.      // 同步信号量，表示产品的数量，也是非空缓冲区的数量
```

- **full** 是缓冲池“满”数目，初值为 0，**empty** 是缓冲池“空”数目，初值为 N。实际上，**full** 和 **empty** 是同一个含义：  
$$\text{full} + \text{empty} == N$$
- 只要缓冲池未满  $\text{empty} > 0$ ，生产者便可将消息送入缓冲池；
- 只要缓冲池未空  $\text{full} > 0$ ，消费者便可从缓冲池取走一个消息。

# 2.5.1 生产者 - 消费者问题

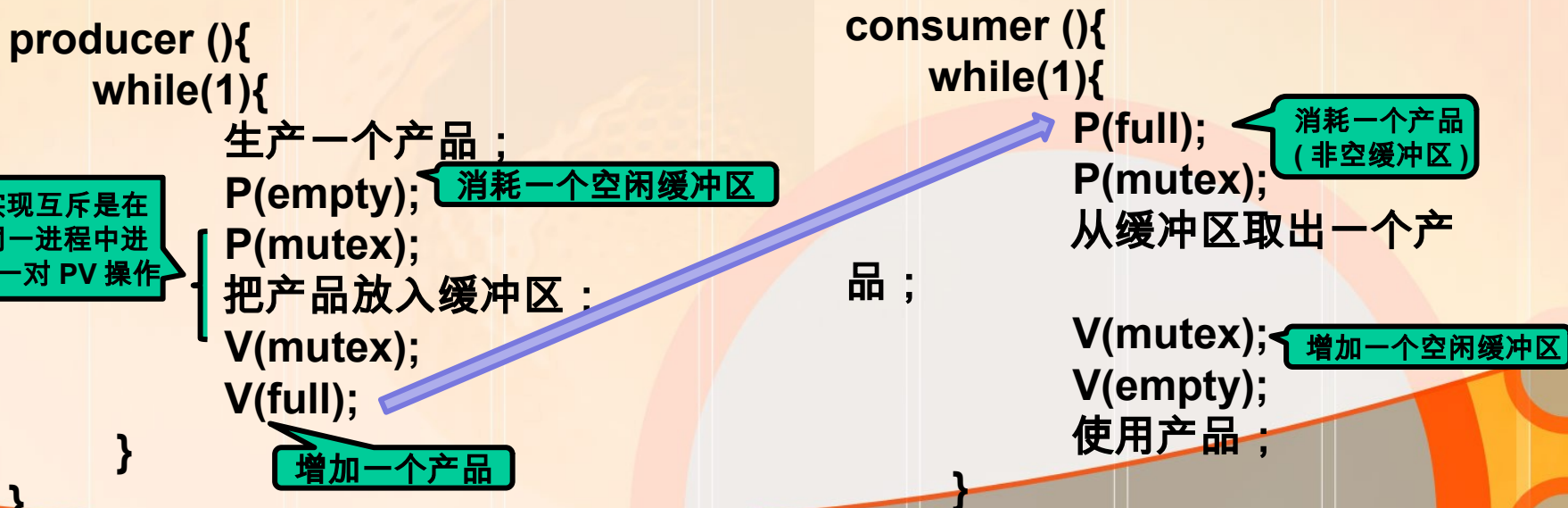
## 如何实现

生产者、消费者共享一个**初始为空、大小为 n 的缓冲区**。  
只有**缓冲区没满**时，生产者才能把产品放入缓冲区，否则**必须等待**。  
只有**缓冲区不空**时，消费者才能从中取出产品，否则**必须等待**。  
缓冲区是临界资源，各进程必须**互斥地访问**。

semaphore mutex = 1;   // 互斥信号量，实现对缓冲区的互斥访问

semaphore empty = n;   // 同步信号量，表示空闲缓冲区的数量

semaphore full = 0;    // 同步信号量，表示产品的数量，也是非空缓冲区的数量



## 2.5.1 生产者 - 消费者问题

**生产者 :**

```
while ( 1 ) {  
    生产商品 x;  
    wait (empty);  
    buffer[in]=x;  
    in=(in+1)%N;  
    signal (full);  
}
```

**消费者 :**

```
while (1) {  
    wait (full);  
    nextc=buffer[out];  
    out=(out+1)%N;  
    signal (empty);  
}
```

## 2.5.1 生产者 - 消费者问题

问题扩充：若生产者与消费者变成多对多关系，  
我们要做什么变动？

增加互斥信号量：mutex=1;

生产者：

```
生产商品 x;  
wait (empty);  
wait (mutex);  
buffer[in]=x;  
in=(in+1)%N;  
signal (mutex);  
signal (full);
```

消费者：

```
wait (full);  
wait (mutex);  
nextc=buffer[out];  
out=(out+1)%N;  
signal (mutex);  
signal (empty);
```



## 2.5.1 生产者 - 消费者问题

- 注意：
  - 每个程序中互斥的 `wait(mutex)` 和 `signal(mutex)` 必须成对出现。
  - 对资源信号量 `empty` 和 `full` 的 `wait`、`signal` 操作成对出现，但它们**分别处于不同的程序中**。例如 `wait` 在计算进程中，而 `signal` 则在打印进程中，计算进程若因执行 `wait` 而阻塞，则以后将由打印进程将它唤醒。
  - 每个程序中的 `wait` 操作顺序不能颠倒。应先执行对**资源信号量**的 `wait` 操作，然后再执行对互斥信号量的 `wait` 操作，否则可能引起进程死锁。



## 2.5.1 生产者 - 消费者问题

思考：能否改变相邻 P、V 操作的顺序？

```
semaphore mutex = 1; // 互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n; // 同步信号量，表示空闲缓冲区的数量
semaphore full = 0; // 同步信号量，表示产品的数量，也是非空缓冲区的数量
```

```
producer(){
    while(1){
        生产一个产品;
        P(mutex); ①
        P(empty); ②
        把产品放入缓冲区;

        V(mutex);
        V(full);
    }
```

mutex 的 P  
操作在前

```
consumer(){
    while(1){
        P(full); ③
        P(mutex); ④
        从缓冲区取出一个产
        品;

        V(mutex);
        V(empty);
        使用产品;
    }
```

能否放到 PV  
操作之间

若此时 empty 里面已经放满产品，则 empty=0，full=n。

则生产者进程执行①使得 mutex=0，接着再执行②，由于已没有空闲缓冲区，因此生产者会被阻塞在②，它需要等 empty 资源。

由于生产者进程阻塞，因此切换回消费者进程。消费者进程执行③，由于 mutex 为 0，即生产者还没释放对临界资源的“锁”，因此消费者进程也被阻塞。

这就造成了生产者等待消费者释放空闲缓冲区，而消费者又等待生产者释放临界区的情况。生产者和消费者循环等待被对方唤醒，出现“死锁”。

同样的，若缓冲区中没有产品，即 full=0，empty=n。按照③④①的顺序执行也会发生死锁。

因此，实现互斥的 P 操作一定要在实现同步的 P 操作之后。

## 2.5.1 生产者 - 消费者问题

### 问题回顾与重要考点

PV 操作题目的解题思路：

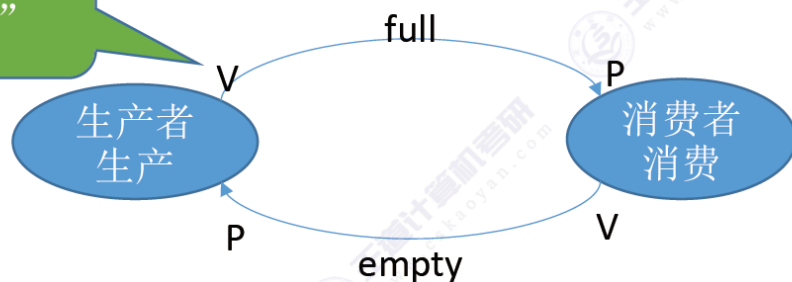
1. 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系。
2. 整理思路。根据各进程的操作流程确定P、V操作的大致顺序。
3. 设置信号量。设置需要的信号量，并根据题目条件确定信号量初值。（互斥信号量初值一般为1，同步信号量的初始值要看对应资源的初始值是多少）

生产者消费者问题是一个互斥、同步的综合问题。

对于初学者来说最难的是发现题目中隐含的两对同步关系。

有时候是消费者需要等待生产者生产，有时候是生产者要等待消费者消费，这是两个不同的“一前一后问题”，因此也需要设置两个同步信号量。

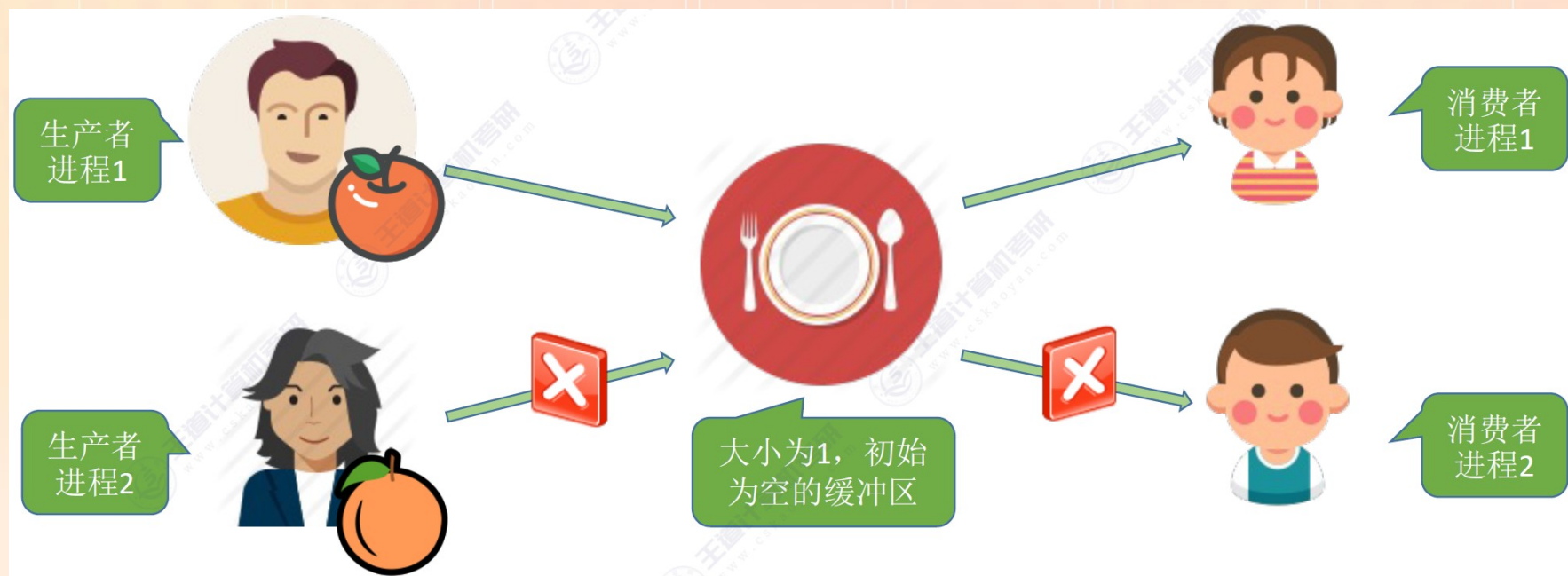
实现“一前一后”  
需要“前V后P”



**易错点：**实现互斥和实现同步的两个P操作的先后顺序（死锁问题）

# 引申 -- 多生产者 - 消费者问题

- 问题描述：桌子上有一只盘子，每次只能向其中放入一个水果。爸爸专门向盘子中放入苹果，妈妈专门向盘子中放入橘子，儿子专等着吃盘子中的橘子，女儿专等着吃盘子中的苹果。只有盘子空时，爸爸妈妈才能向盘子中放入一个水果。仅当盘子中有自己需要的水果时，儿子或女儿可以从盘子中取出水果。用 PV 操作实现上述过程。





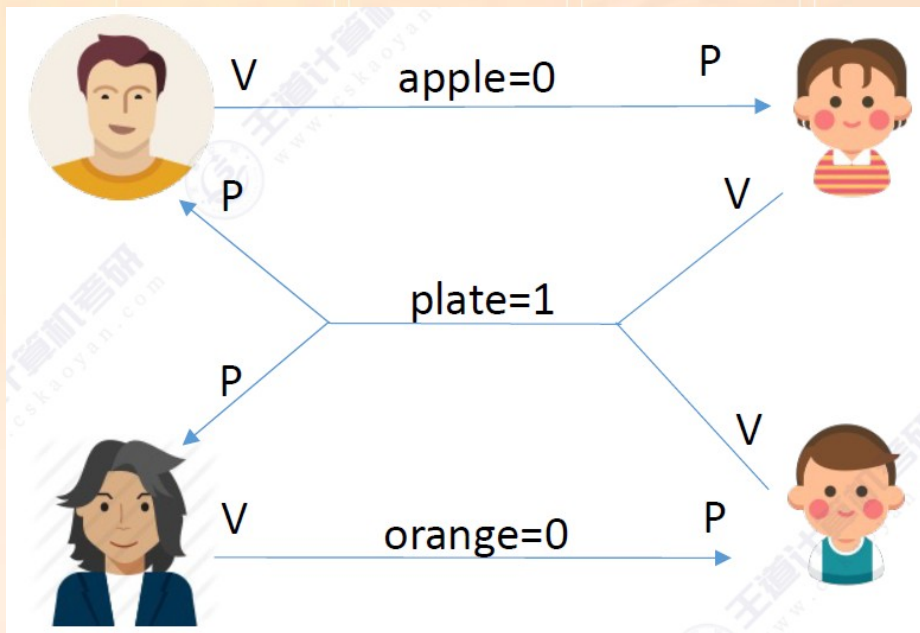
# 引申 -- 多生产者 - 消费者问题

互斥：在临界区前后分别 PV  
同步：先进行 V 操作，再进行 P 操作

## • 问题分析：

1. 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系。
2. 整理思路。根据各进程的操作流程确定 P、V 操作的大致顺序。
3. 设置信号量。设置需要的信号量，并根据题目条件确定信号量初值。（互斥信号量初值一般为 1，同步信号量的初始值要看对应资源的初始值是多少）

**同步信号量：**apple、orange，初始值都为 0. plate，初始值为 1，表示盘子是否为空。



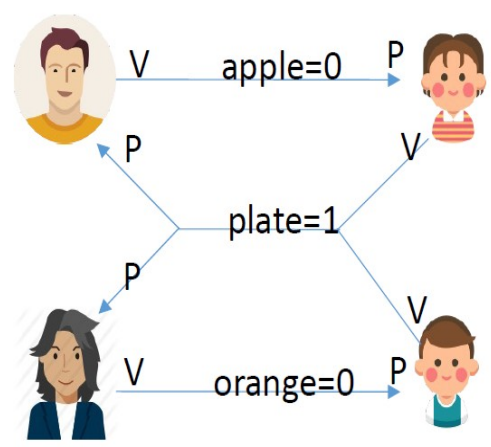
**互斥关系：**  
对缓冲区（盘子）的访问需要互斥地进行

- 同步关系（一前一后）：**
1. 父亲将苹果放入盘子后，女儿才能取苹果。
  2. 母亲将橘子放入盘子后，儿子才能取橘子。
  3. 只有**盘子为空**时，**父亲或母亲**才能放入水果。

“盘子为空”这个事件可以由儿子或女儿触发，事件发生后才允许父亲或母亲放水果

# 引申 -- 多生产者 - 消费者问题

## 如何实现



问题：可不可以不用互斥信号量？

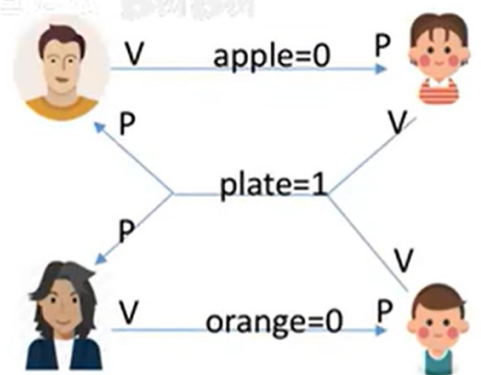
```
semaphore mutex = 1; //实现互斥访问盘子（缓冲区）
semaphore apple = 0; //盘子中有几个苹果
semaphore orange = 0; //盘子中有几个橘子
semaphore plate = 1; //盘子中还可以放多少个水果
```

<pre>dad () {     while(1) {         准备一个苹果;         P(plate);         P(mutex);         把苹果放入盘子;         V(mutex);         V(apple);     } }</pre>	<pre>mom () {     while(1) {         准备一个橘子;         P(plate);         P(mutex);         把橘子放入盘子;         V(mutex);         V(orange);     } }</pre>	<pre>daughter () {     while(1) {         P(apple);         P(mutex);         从盘中取出苹果;         V(mutex);         V(plate);         吃掉苹果;     } }</pre>	<pre>son () {     while(1) {         P(orange);         P(mutex);         从盘中取出橘子;         V(mutex);         V(plate);         吃掉橘子;     } }</pre>
---	--	--	--





# 引申 -- 多生产者 - 消费者问题



### 如何实现

```
semaphore mutex = 1; //实现互斥访问盘子（缓冲区）
semaphore apple = 0; //盘子中有几个苹果
semaphore orange = 0; //盘子中有几个橘子
semaphore plate = 1; //盘子中还可以放多少个水果
```

结论：即使不设置专门的互斥变量mutex，也不会出现多个进程同时访问盘子的现象

```
dad () {
    while(1) {
        准备一个苹果;
        P(plate);
        把苹果放入盘子;
        V(apple);
    }
}

mom () {
    while(1) {
        准备一个橘子;
        P(plate);
        把橘子放入盘子;
        V(orange);
    }
}

daughter () {
    while(1) {
        P(apple);
        从盘中取出苹果;
        V(plate);
        吃掉苹果;
    }
}

son () {
    while(1) {
        P(orange);
        从盘中取出橘子;
        V(plate);
        吃掉橘子;
    }
}
```

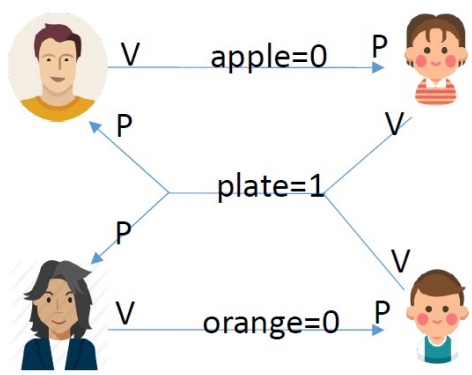
分析：刚开始，儿子、女儿进程即使上处机运行，则：父亲 P(plate)，可以访问盘果 V(apple)，女儿进程被唤醒，其他进程子) == 》女儿 P(apple)，访问盘子， V(盘子 (其他进程暂时无法进入临界区) == )

原因在于：本题缓冲区的大小为 1，在任何时刻，apple、orange、plate 三个同步信号量中最多只有一个是 1。因此在任何时刻，最多只有一个进程的 P 操作不会被阻塞，并顺利地进入临界区。（如果盘子容量为 2 呢）



# 引申 -- 多生产者 - 消费者问题

## 如何实现



如果盘子（缓冲区）容量为2

问盘子（缓冲区）

```
semaphore mutex = 1;
semaphore apple = 0;
semaphore orange = 0;
semaphore plate = 2;
```

//盘子中有几个苹果  
//盘子中有几个橘子  
//盘子中还可以放多少个水果

<pre>dad () {     while (1) {         准备一个苹果;         P(plate);         把苹果放入盘子;         V(apple);     } }</pre>	<pre>mom () {     while (1) {         准备一个橘子;         P(plate);         把橘子放入盘子;         V(orange);     } }</pre>	<pre>daughter () {     while (1) {         P(apple);         从盘中取出苹果;         V(plate);         吃掉苹果;     } }</pre>	<pre>son () {     while (1) {         P(orange);         从盘中取出橘子;         V(plate);         吃掉橘子;     } }</pre>
--	---	---	---

父亲 P(plate)，可以访问盘子→母亲 P(plate)，可以访问盘子→父亲在往盘子里放苹果，同时母亲也可以往盘子里放橘子。于是就出现了两个进程同时访问缓冲区的情况，有可能导致两个进程写入缓冲区的数据相互覆盖的情况。因此，如果缓冲区大小大于1，就必须专门设置一个互斥信号量 **mutex** 来保证互斥访问缓冲区。



# 引申 -- 多生产者 - 消费者问题

## 知识回顾与重要考点

总结：在生产者-消费者问题中，如果缓冲区大小为1，那么有可能不需要设置互斥信号量就可以实现互斥访问缓冲区的功能。当然，这不是绝对的，要具体问题具体分析。

建议：在考试中如果来不及仔细分析，可以加上互斥信号量，保证各进程一定会互斥地访问缓冲区。但需要注意的是，实现互斥的P操作一定要在实现同步的P操作之后，否则可能引起“死锁”。

PV 操作题目的解题思路：

1. 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系。
2. 整理思路。根据各进程的操作流程确定P、V操作的大致顺序。
3. 设置信号量。设置需要的信号量，并根据题目条件确定信号量初值。（互斥信号量初值一般为1，同步信号量的初始值要看对应资源的初始值是多少）



# 引申 -- 多生产者 - 消费者问题

## 知识回顾与重要考点

解决“多生产者-多消费者问题”的关键在于理清复杂的同步关系。

在分析同步问题（一前一后问题）的时候不能从单个进程行为的角度来分析，要把“一前一后”发生的事看做是两种“事件”的前后关系。

比如，如果从单个进程行为的角度来考虑的话，我们会有以下结论：

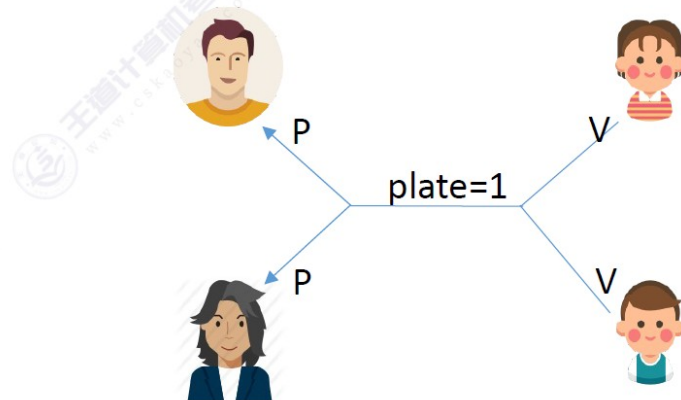
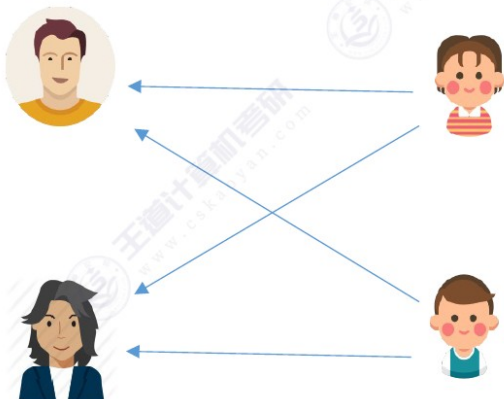
如果盘子里装有苹果，那么一定要女儿取走苹果后父亲或母亲才能再放入水果

如果盘子里装有橘子，那么一定要儿子取走橘子后父亲或母亲才能再放入水果

这么看是否就意味着要设置四个同步信号量分别实现这四个“一前一后”的关系了？

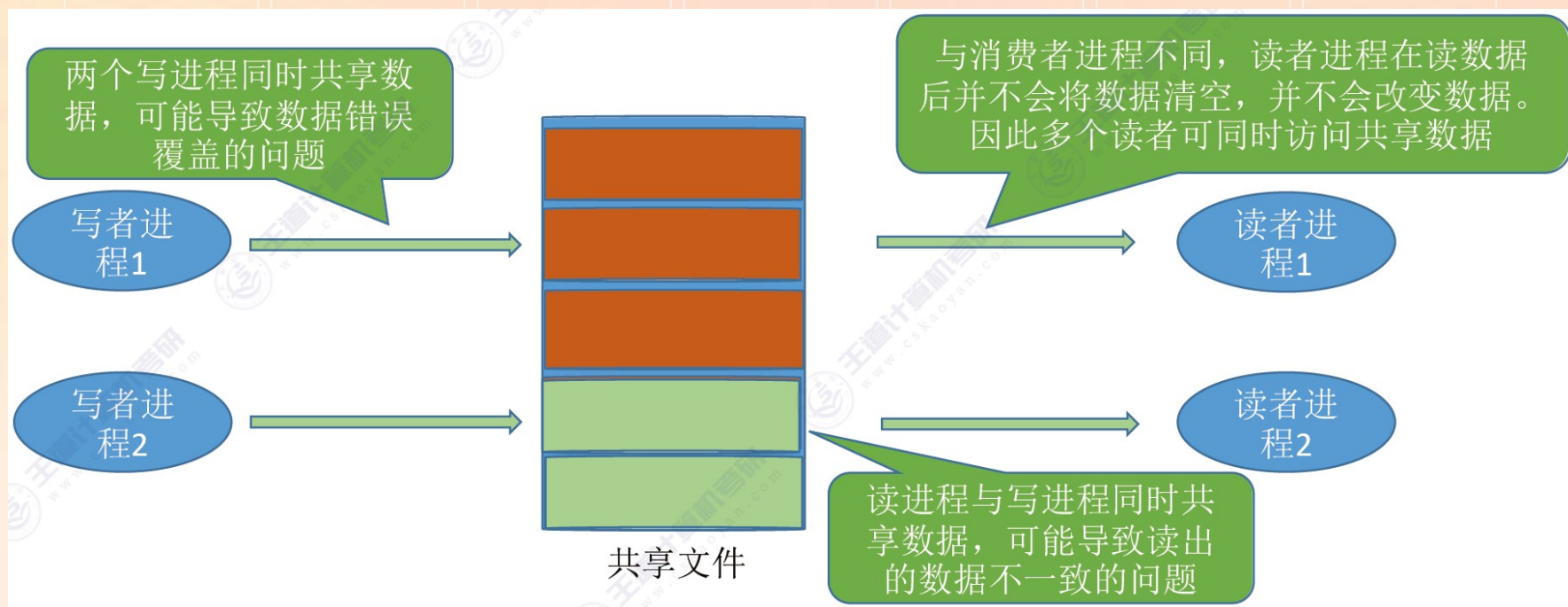
正确的分析方法应该从“事件”的角度来考虑，我们可以把上述四对“进程行为的前后关系”抽象为一对“事件的前后关系”

盘子变空事件→放入水果事件。“盘子变空事件”既可由儿子引发，也可由女儿引发；“放水果事件”既可能是父亲执行，也可能是母亲执行。这样的话，就可以用一个同步信号量解决问题了



## 2.5.2 读者 - 写者问题 (the readers-writers problem)

- 有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用。但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：①允许多个读者可以同时~~对文件执行读操作~~；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出。





## 2.5.2 读者 - 写者问题 (the readers-writers problem)

- 一个数据文件或记录可被多个进程共享。其中，有些进程要求读；而另一些进程要求行写或修改。
- 只要求读的进程称为“Reader 进程”，其它进程称为“Writer 进程”。
- 任一时刻“写者”最多只允许一个，而“读者”则允许多个——“读 - 写”互斥，“写 - 写”互斥，“读 - 读”允许。
- 所谓读者 - 写者问题是指保证一个 Writer 进程必须与其它进程互斥地访问共享对象的同步问题。

## 2.5.2 读者 - 写者问题 (the readers-writers problem)

- 有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用。但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：①允许多个读者可以同时~~对文件执行读操作~~；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出。
- 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系。
  - 整理思路。根据各进程的操作流程确定 P、V 操作的大致顺序。
  - 设置信号量。设置需要的信号量，并根据题目条件确定信号量初值。（互斥信号量初值一般为 1，同步信号量的初始值要看对应资源的初始值是多少）

两类进程：写进程、读进程

互斥关系：写进程 -- 写进程，写进程 -- 读进程。读进程与读进程不存在互斥问题

## 2.5.2 读者 - 写者问题 (the readers-writers problem)

```
semaphore rw=1;    // 用于实现对共享文件的互斥访问
int count = 0;    // 记录当前有几个读进程正在访问文件
semaphore mutex = 1; // 用于保证对 count 变量的互斥访问
```

```
writer() {
    while(1){
        P(rw) ;    // 写之前“加锁”
        写文件
        V(rw);    // 写完了“解锁”
    }
}
```

思考：若两个读进程并发执行，则 count=0 时两个进程也许都能满足 if 条件，都会执行 P(rw)，从而使第二个读进程阻塞的情况。

如何解决：出现上述问题的原因在于对 count 变量的检查和赋值无法一气呵成，因此可以设置另一个互斥信号量来保证各读进程对 count 的访问是互斥的。

```
reader() {
    while(1){
        P(mutex); // 各进程互斥访问 count
        if(count==0) // 由第一个读进程负责
            P(rw) ; // 读之前“加锁”
        count++; // 访问文件的读进程数 +1
        V(mutex);

        读文件 ...

        P(mutex); // 各进程互斥访问 count
        count--; // 访问文件的读进程数 -1
        if(count==0) // 由最后一个读进程负责
            V(rw) ; // 读完了“解锁”
        V(mutex);
    }
}
```

## 2.5.2 读者 - 写者问题 (the readers-writers problem)

### 如何实现

```
semaphore rw=1;      //用于实现对共享文件的互斥访问
int count = 0;        //记录当前有几个读进程在访问文件
semaphore mutex = 1; //用于保证对count变量的互斥访问
```

```
writer () {
    while(1) {
        P(rw);    //写之前“加锁”
        写文件...
        V(rw);    //写完了“解锁”
    }
}
```

思考：若两个读进程并发执行，则 `count=0` 时两个进程也许都能满足 `if` 条件，都会执行 `P(rw)`，从而使第二个读进程阻塞的情况。  
如何解决：出现上述问题的原因在于对 `count` 变量的检查和赋值无法一气呵成，因此可以设置另一个互斥信号量来保证各读进程对 `count` 的访问是互斥的。

```
reader () {
    while(1) {
        P(mutex); //各读进程互斥访问count
        if(count==0) //由第一个读进程负责
            P(rw); //读之前“加锁”
        count++; //访问文件的读进程数+1
        V(mutex);
        读文件...
        P(mutex); //各读进程互斥访问count
        count--; //访问文件的读进程数-1
        if(count==0) //由最后一个读进程负责
            V(rw); //读完了“解锁”
        V(mutex);
    }
}
```

潜在的问题：只要有读进程还在读，写进程就要一直阻塞等待，可能“饿死”。因此，这种算法中，读进程是优先的



## 2.5.2 读者 - 写者问题 (the readers-writers problem)

### 如何实现

```
semaphore rw=1;      //用于实现对共享文件的互斥访问
int count = 0;        //记录当前有几个读进程在访问文件
semaphore mutex = 1;  //用于保证对count变量的互斥访问
semaphore w = 1;      //用于实现“写优先”
```

分析以下并发执行 P(w) 的情况:

读者1→读者2

写者1→写者2

写者1→读者1

读者1→写者1→读者2

写者1→读者1→写者2

结论: 在这种算法中, 连续进入的多个读者可以同时读文件; 写者和其他进程不能同时访问文件; 写者不会饥饿, 但也并不是真正的“写优先”, 而是相对公平的先来先服务原则。

有的书上把这种算法称为“读写公平法”。

```
writer () {
    while (1) {
        P(w);
        P(rw);
        写文件...
        V(rw);
        V(w);
    }
}
```

```
reader () {
    while (1) {
        P(w);
        P(mutex);
        if (count == 0)
            P(rw);
        count++;
        V(mutex);
        V(w);
        读文件...
        P(mutex);
        count--;
        if (count == 0)
            V(rw);
        V(mutex);
    }
}
```



## 2.5.2 读者 - 写者问题 (the readers-writers problem)

读者-写者问题为我们解决复杂的互斥问题提供了一个参考思路。

其**核心思想**在于设置了一个**计数器 count** 用来记录当前正在访问共享文件的读进程数。我们可以用 **count** 的值来判断当前进入的进程是否是第一个/最后一个读进程，从而做出不同的处理。

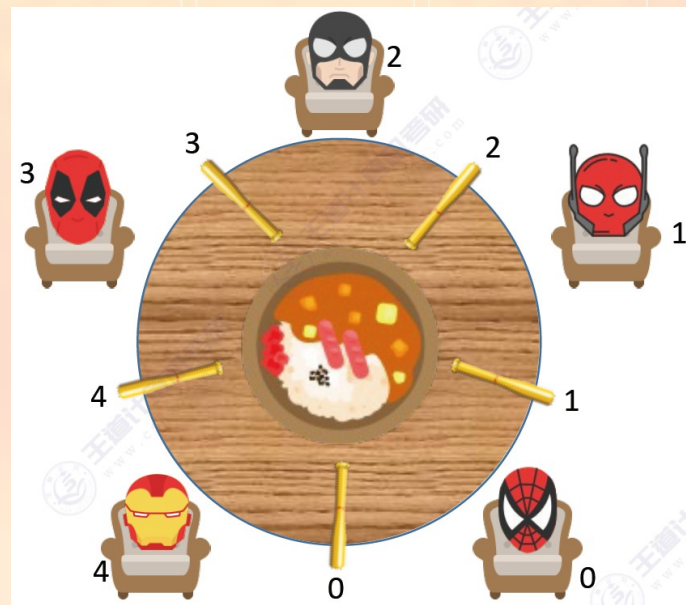
另外，对 **count** 变量的检查和赋值不能一气呵成导致了一些错误，如果**需要实现“一气呵成”**，自然应该想到用**互斥信号量**。

最后，还要认真体会我们是如何解决“写进程饥饿”问题的。

绝大多数的考研PV操作大题都可以用之前介绍的几种生产者-消费者问题的思想来解决，如果遇到更复杂的问题，可以想想能否用读者写者问题的这几个思想来解决。

## 2.5.3 哲学家就餐问题

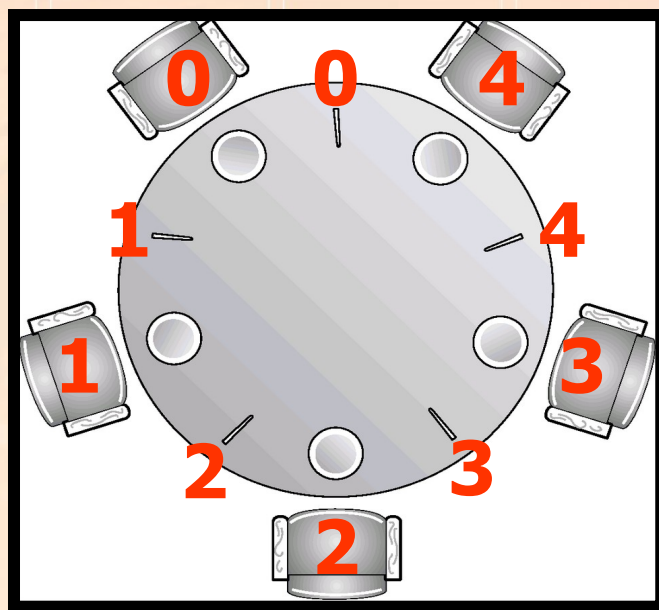
- 问题描述：（由 Dijkstra 首先提出并解决）
  - 5 个哲学家围绕一张圆桌而坐，桌子上放着 5 支筷子，每两个哲学家之间放一支；
  - 哲学家的动作包括思考和进餐：
    - 进餐时需要同时拿起他左边和右边的两支筷子；
    - 思考时则同时将两支筷子放回原处。



1. 关系分析。系统中有 5 个哲学家进程，5 位哲学家与左右邻居对其中间筷子的访问是互斥关系。
2. 整理思路。这个问题中只有互斥关系，但与之前遇到的问题不同的是，每个哲学家进程需要同时持有两个临界资源才能开始吃饭。如何避免临界资源分配不当造成的死锁问题，是哲学家问题的精髓。
3. 信号量设置。定义互斥信号量数组 `chopstick[5]={1,1,1,1,1}` 用于实现对 5 个筷子的互斥访问。并对哲学家按照 0~4 编号，哲学家  $i$  左边的筷子编号为  $i$ ，右边的筷子编号为  $(i+1)\%5$ 。

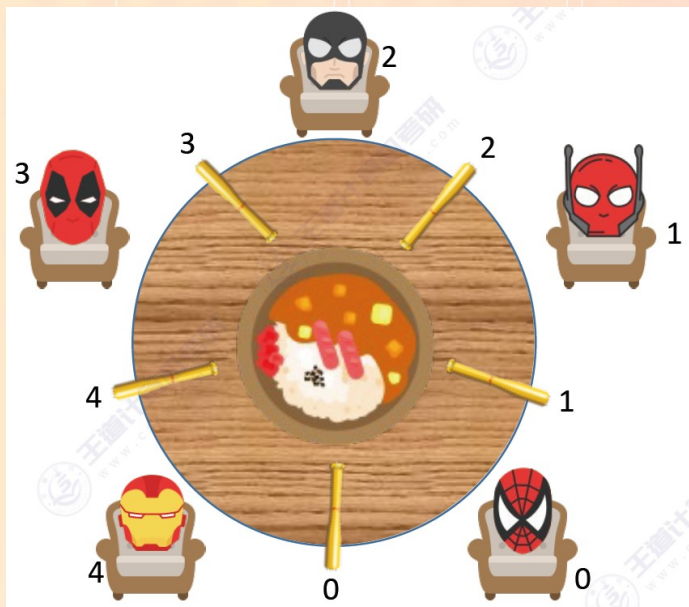
## 2.5.3 哲学家进餐问题

- 哲学家逆时针编号 0-4，筷子也相应编号
- 定义互斥信号量数组 `chopstick[5]`，对应 5 支筷子，初值均为 1。





## 2.5.3 哲学家进餐问题



每位哲学家循环等待右边的人放下筷子 (阻塞)。发生“死锁”。

第  $i$  个哲学家进程 :

```
do {
```

```
    wait (chopstick [i] )  
    wait (chopstick [(i+1) % 5])
```

```
    ...  
    eat
```

```
    ...  
    signal (chopstick [i] );  
    signal (chopstick[(i+1) %
```

```
5]);
```

```
    ...  
    think  
    ...
```

```
} while (1);
```

如果 5 个哲学家并发地拿起了自己左手边的筷子 ....

问题：所有哲学家拿起了左边的筷子，全部封锁在右边筷子上——死锁。

解决方法：

1 ) 桌子前同时最多允许 4 个哲学家准备进餐。

```
semaphore chopstick[5]={1 , 1 , 1 , 1 , 1};
semaphore room=4;
void philosopher(int i)
{
    while(true)
    {
        think();
        wait(room); // 请求进餐
        wait(chopstick[i]); // 请求左手边的筷子
        wait(chopstick[(i+1)%5]); // 请求右手边的筷子
        eat();
        signal(chopstick[(i+1)%5]); // 释放右手边的筷子
        signal(chopstick[i]); // 释放左手边的筷子
        signal(room); // 退出进餐释放信号量 room
    }
}
```



2 ) 仅当两双筷子都可用时，才允许哲学家拿筷子。

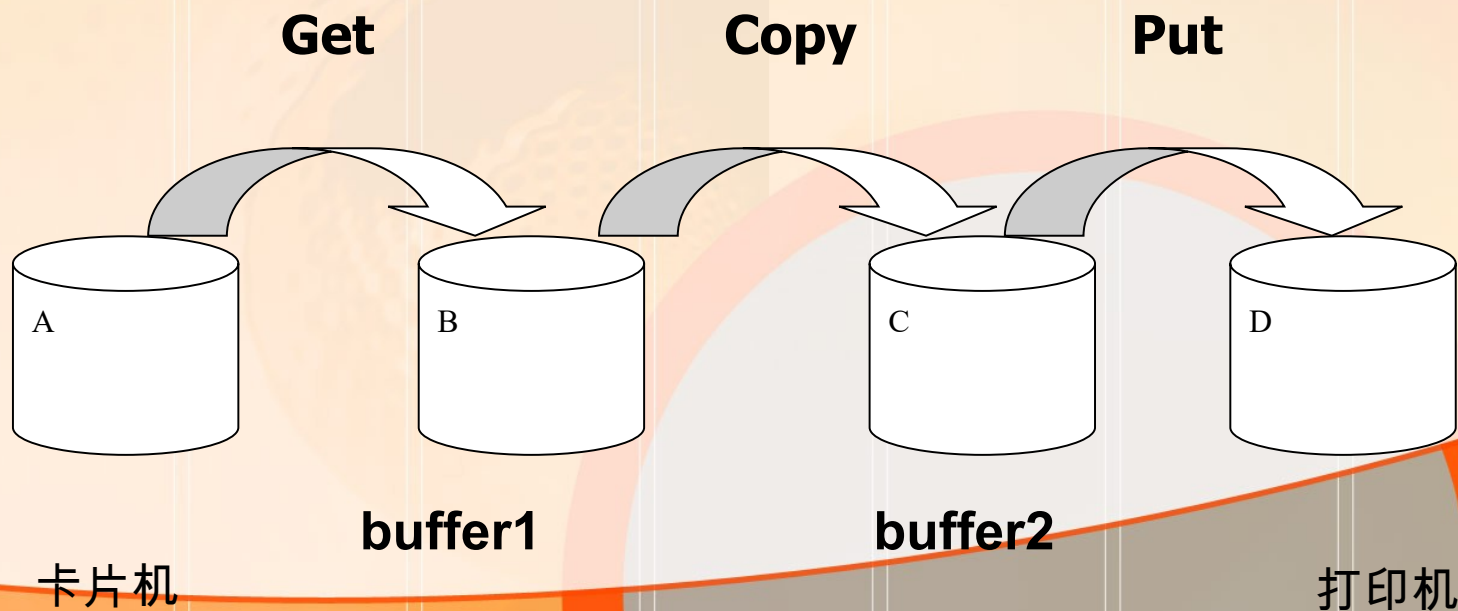
```
semaphore chopstick[5]={1 , 1 , 1 , 1 , 1};  
void philosopher(int i)  
{  
    while(true)  
    {  
        think();  
        Swait(chopstick[(i+1)]%5,chopstick[i]);  
        eat();  
        Ssignal(chopstick[(i+1)]%5,chopstick[i]);  
    }  
}
```

3 ) 奇数哲学家先拿左手筷子再拿右手筷子。偶数哲学家则反过来。

```
semaphore chopstick[5]={1 , 1 , 1 , 1 , 1};  
void philosopher(int i)  
{  
    while(true)  
    {  
        think();  
        if(i%2 == 0) // 偶数哲学家，先右后左  
        {  
            wait (chopstick[ i + 1 ] mod 5) ;  
            wait (chopstick[ i]) ;  
            eat();  
            signal (chopstick[ i + 1 ] mod 5) ;  
            signal (chopstick[ i]) ;  
        }  
        else // 奇数哲学家，先左后右  
        {  
            wait (chopstick[ i]) ;  
            wait (chopstick[ i + 1 ] mod 5) ;  
            eat();  
            signal (chopstick[ i]) ;  
            signal (chopstick[ i + 1 ] mod 5) ;  
        }  
    }  
}
```

# 作业

- 1. 从读卡机上读进 N 张卡片，然后复制一份，要求复制出来的卡片与读进的卡片完全一样。该任务由三个进程 get，copy 和 put 及两个缓冲区 buffer1 和 buffer2 完成。进程 get 的功能是把一张卡片的信息从读卡机读入 buffer1；进程 copy 是把 buffer1 中的信息复制到 buffer2；进程 put 的功能是取出 buffer2 中的信息并从打印机上打印。
- 要求说明每个信号量是用于解决什么互斥、什么同步关系的。



- 2. 办公室有一个文件格，专门存放整包的 A3 和 A4 打印纸，所有人都可以使用该文件格。每次放纸时只能放入一包 A3 或 A4 纸，每次取纸时只能取一包 A3 或 A4 纸，放纸和取纸不能同时进行。初始时文件格是空的。由于文件格容量为 10，因此要求 A3 纸和 A4 纸的包数总和小于 10。下面分别给出了放纸和取纸过程的文字描述，请用 P、V 操作分别描述放纸和取纸过程，并说明所定义的信号量的初值和含义。

放纸进程描述：

若文件格已满则等待；

if ( 放 A3 纸 )

    若文件格有空位则放 A3 纸；

else

    若文件格有空位则放 A4 纸；

离开；

取纸进程描述：

if ( 取 A3 纸 ) {

    若无 A3 纸则等待；

    若有 A3 纸则取纸；

}

else {

    若无 A4 纸则等待；

    若有 A4 纸则取纸；

}

离开；