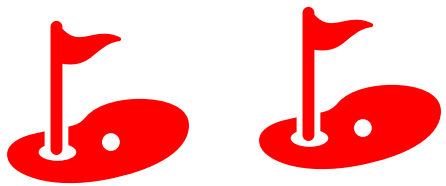
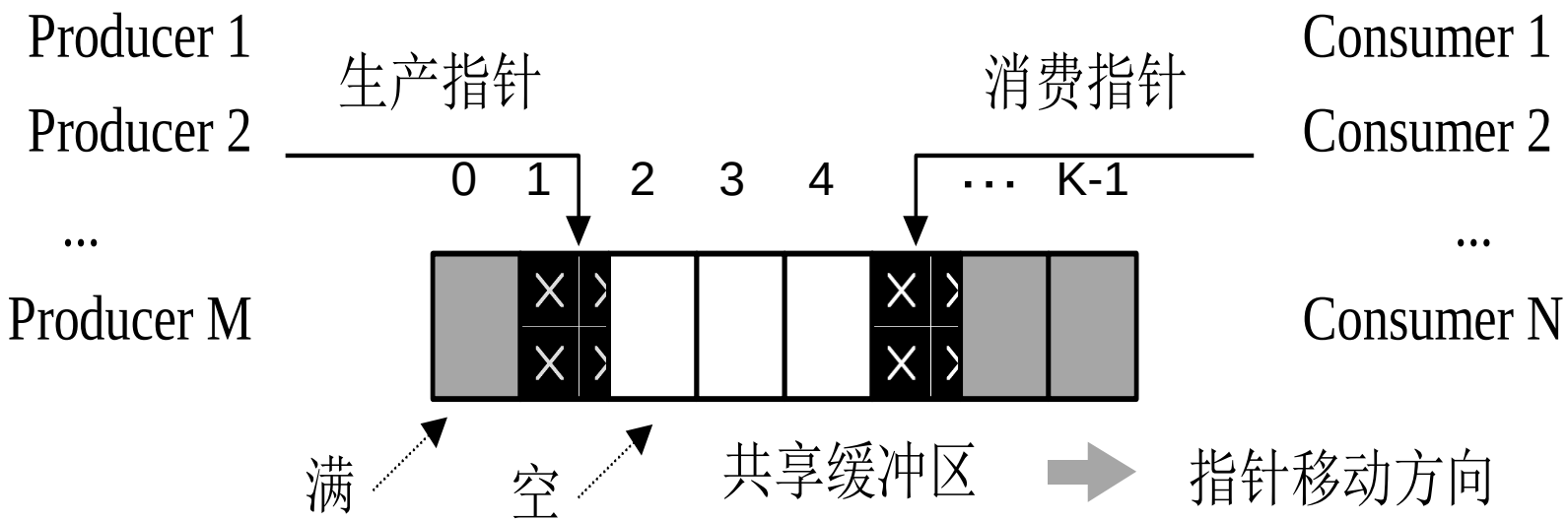


经典同步问题

经典进程同步问题——生产者-消费者问题

■ 问题描述：

若干进程通过有限的共享缓冲区交换数据。其中，"生产者"进程不断写入，而"消费者"进程不断读出；共享缓冲区共有K个；任何时刻只能有一个进程可对共享缓冲区进行操作。



经典进程同步问题——生产者-消费者问题

■分析：

- 确定进程：进程数量及工作内容；
- 确定进程间的关系：
  - 互斥：多个进程间互斥使用同一个缓冲池；
  - 同步：当缓冲池空时，消费者必须阻塞等待；
    当缓冲池满时，生产者必须阻塞等待。
- 设置信号量：
  - mutex：用于访问缓冲池时的互斥，初值是1
  - full：“满缓冲”数目，初值为0；
  - empty：“空缓冲"数目，初值为K。full+empty=K

经典进程同步问题——生产者-消费者问题

■算法描述

```
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，也即非空缓冲区的数量
```

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

```
consumer (){
    while(1){
        P(full);
        P(mutex);
        从缓冲区取出一个产品;
        V(mutex);
        V(empty);
        使用产品;
    }
}
```

消耗一个空闲缓冲区

消耗一个产品（非空缓冲区）

实现互斥是在同一进程中进行一对PV操作

增加一个产品

实现两进程的同步关系，是在其中一个进程中执行P，另一进程中执行V

增加一个空闲缓冲区

经典进程同步问题——生产者-消费者问题

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

```
consumer (){
    while(1){
        P(mutex);
        P(full);
        从缓冲区取出一个产品;
        V(mutex);
        V(empty);
        使用产品;
    }
}
```

若此时缓冲区内已经放满产品，则empty=0，full=n。则生产者进程执行①使mutex变为0，再执行②，由于已没有空闲缓冲区，因此生产者被阻塞。

由于生产者阻塞，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③④①的顺序执行就会发生死锁。

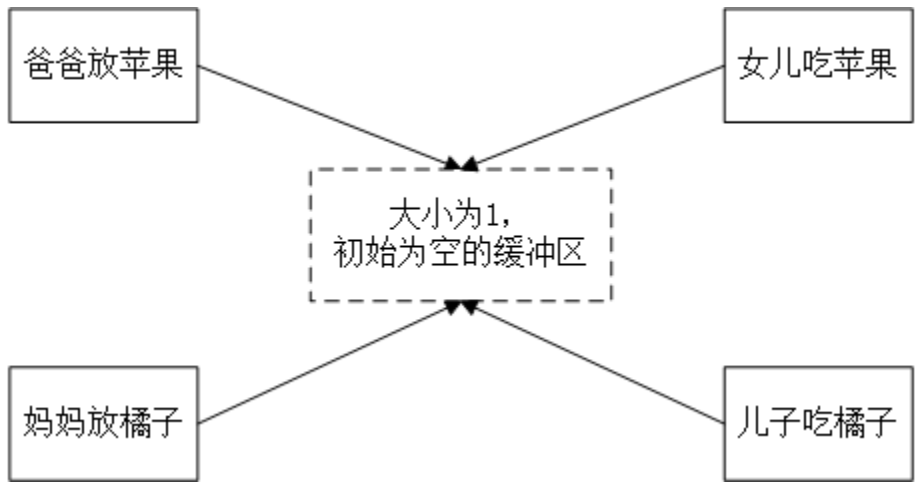
因此，实现互斥的P操作一定要在实现同步的P操作之后。V操作不会导致进程阻塞，因此两个V操作顺序可以交换。

经典进程同步问题——多生产者-多消费者问题

■ 问题描述：

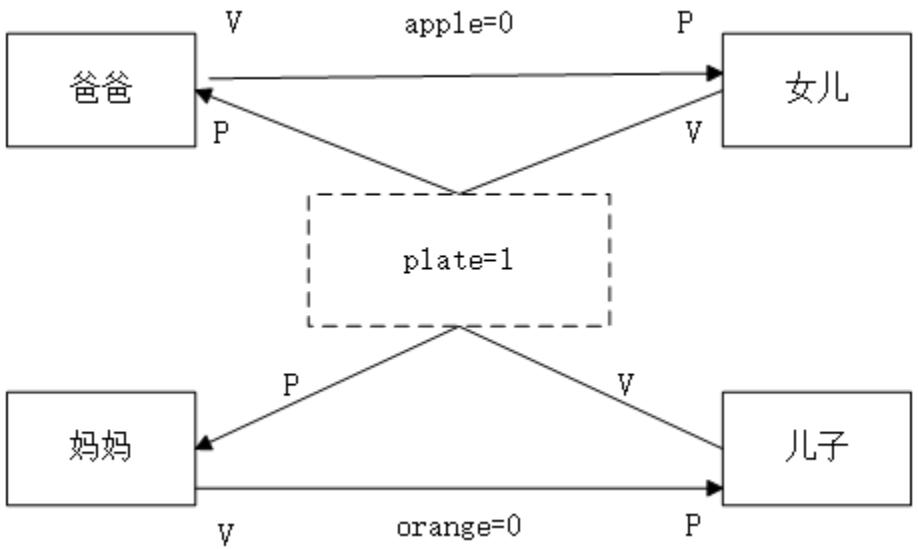
桌子上有一只盘子，每次只能向其中放入一个水果。爸爸专向盘子中放苹果，妈妈专向盘子中放橘子，儿子专等着吃盘子中的橘子，女儿专等着吃盘子中的苹果。只有盘子空时，爸爸或妈妈才可向盘子中放一个水果。仅当盘子中有自己需要的水果时，儿子或女儿可以从盘子中取出水果。

用PV操作实现上述过程。



经典进程同步问题——多生产者-多消费者问题

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）



互斥关系：（mutex = 1）
对缓冲区（盘子）的访问要互斥地进行

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

经典进程同步问题——多生产者-多消费者问题

■算法描述

```
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
```

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

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

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

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

经典进程同步问题——多生产者-多消费者问题

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
```

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

```
doughter (){
    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;
    }
}
```

原因在于：本题中的缓冲区大小为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…

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

分析：刚开始，儿子进程先上处理机运行，则：父亲P(plate)，女儿进程被(apple)，唤醒，其他进程即使apple)，访问盘子，V(plate)，等待盘子入临界区）->……

经典进程同步问题——多生产者-多消费者问题

```
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 = 2; //盘子中还可以放多少个水果
```

如果盘子（缓冲区）容量为2

```
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);
    }
}
```

```
doughter (){
    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;
    }
}
```

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。

经典进程同步问题——吸烟者问题

■ 问题描述：

假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但是要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草、第二个拥有纸、第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者进程一个信号告诉完成了，供应者就会放另外两种材料再桌上，这个过程一直重复（让三个抽烟者轮流地抽烟）

经典进程同步问题——吸烟者问题

假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但是要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草、第二个拥有纸、第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者进程一个信号告诉完成了，供应者就会放另外两种材料再桌上，这个过程一直重复（让三个抽烟者轮流地抽烟）

本质上这题也属于“生产者-消费者”问题，更详细的说应该是“可生产多种产品的单生产者-多消费者”。

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）

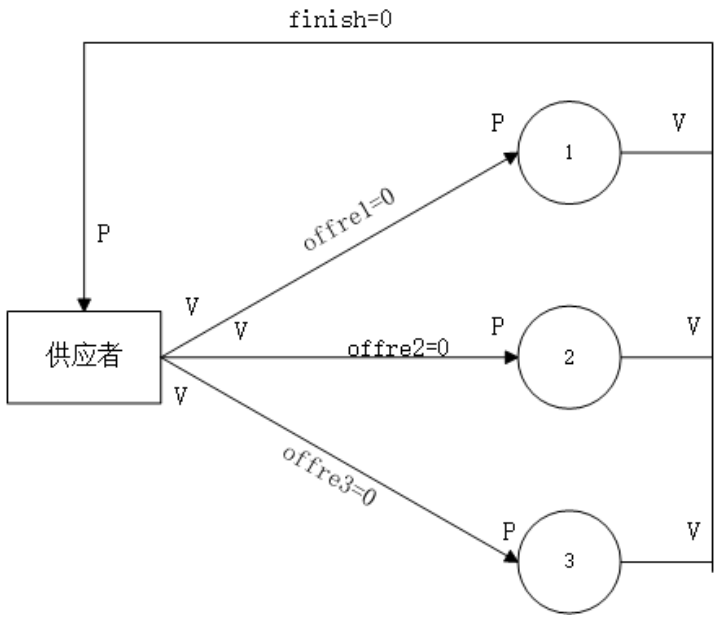
桌子可以抽象为容量为1的缓冲区，要互斥访问

组合一：纸+胶水
组合二：烟草+胶水
组合三：烟草+纸

同步关系（从事件的角度来分析）：
桌上有组合一 -> 第一个抽烟者取走东西
桌上有组合二 -> 第二个抽烟者取走东西
桌上有组合三 -> 第三个抽烟者取走东西
发出完成信号 -> 供应者将下一个组合放到桌上
PV操作顺序：“前V后P”

经典进程同步问题——吸烟者问题

假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但是要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草、第二个拥有纸、第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者进程一个信号告诉完成了，供应者就会放另外两种材料再桌上，这个过程一直重复（让三个抽烟者轮流地抽烟）



桌上有组合一 -> 第一个抽烟者取走东西
桌上有组合二 -> 第二个抽烟者取走东西
桌上有组合三 -> 第三个抽烟者取走东西
发出完成信号 -> 供应者将下一个组合放到桌上

经典进程同步问题——多生产者-多消费者问题

■算法描述

```
provider (){
    while(1){
        if(i==0) {
            将组合一放桌上;
            V(offer1);
        } else if(i==1){
            将组合二放桌上;
            V(offer2);
        } else if(i==2){
            将组合三放桌上;
            V(offer3);
        }
        i = (i+1)%3;
        P(finish);
    }
}
```

```
semaphore offer1 = 0; //桌上组合一的数量
semaphore offer2 = 0; //桌上组合二的数量
semaphore offer3 = 0; //桌上组合三的数量
semaphore finish = 0; //抽烟是否完成
int i = 0; //用于实现“三个抽烟者轮流抽烟”
```

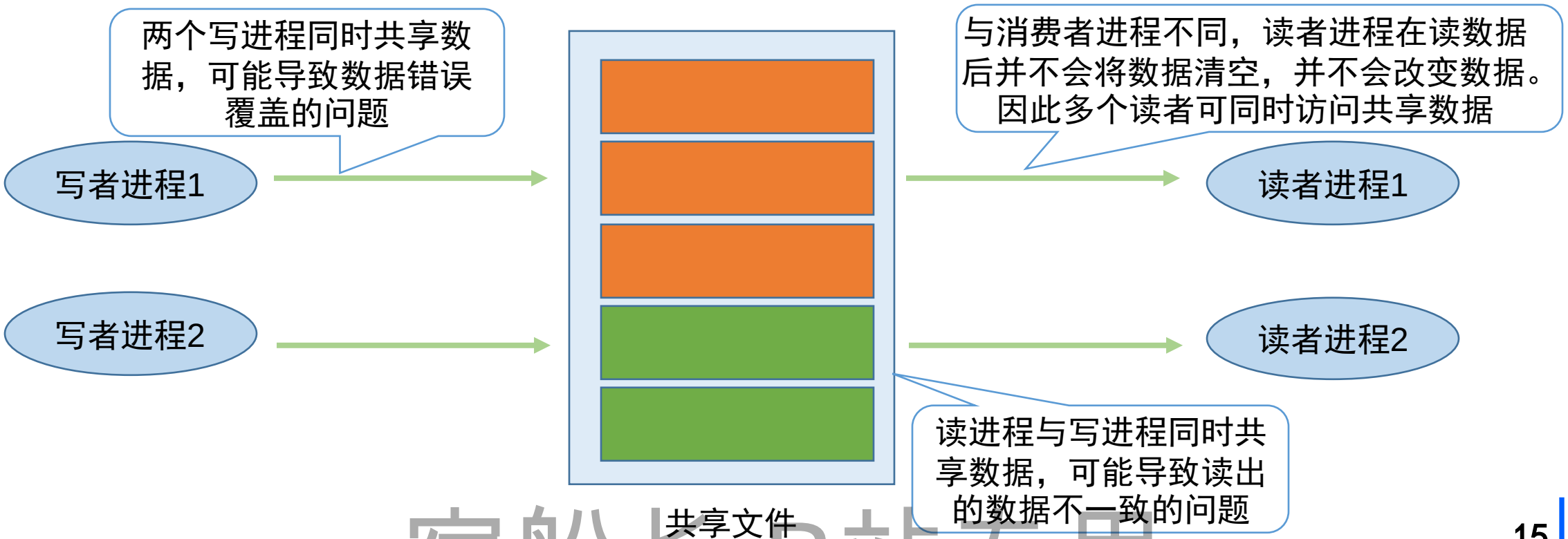
```
smoker1 (){
    while(1){
        P(offer1);
        从桌上拿走组合一; 卷烟; 抽掉;
        V(finish);
    }
}
```

```
smoker2 (){
    while(1){
        P(offer2);
        从桌上拿走组合二; 卷烟; 抽掉;
        V(finish);
    }
}
```

```
smoker3 (){
    while(1){
        P(offer3);
        从桌上拿走组合三; 卷烟; 抽掉;
        V(finish);
    }
}
```

经典进程同步问题——读者-写者问题

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。



经典进程同步问题——读者-写者问题

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。

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）

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。

经典进程同步问题——读者-写者问题

■算法描述

```
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”
    }
}
```

```
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);
    }
}
```

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。

经典进程同步问题——读者-写者问题

```
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

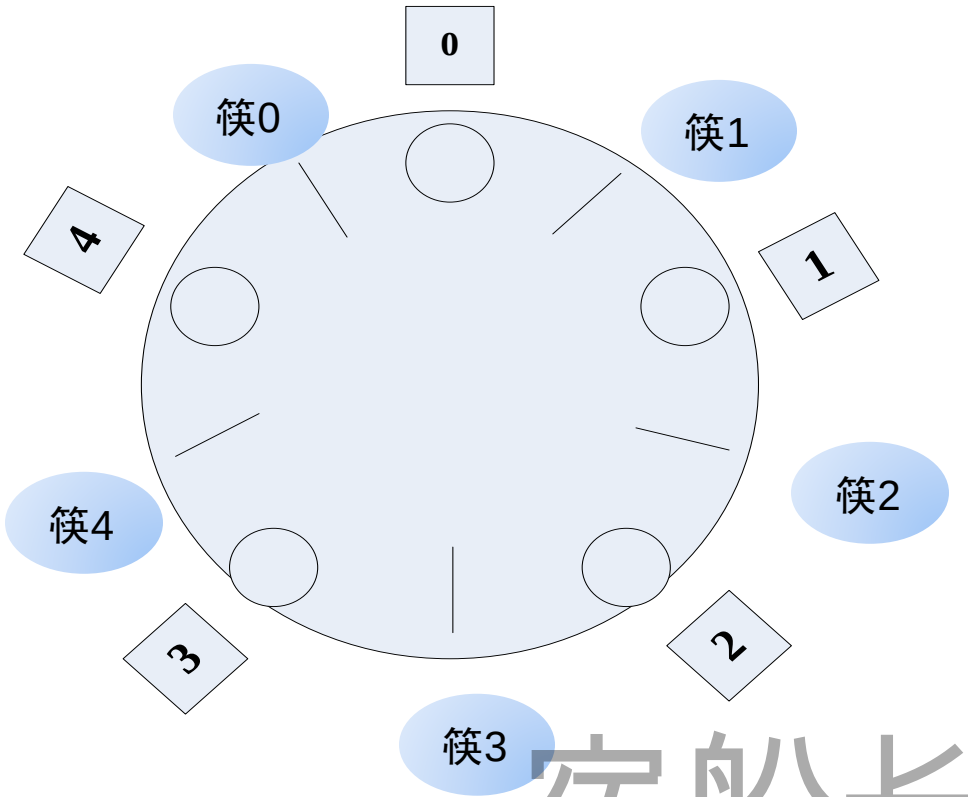
结论：在这种算法中，连续进入的多个读者可以同时读文件；写者和其他进程不能同时访问文件；写者不会饥饿，但也并不是真正的“写优先”，而是相对公平的先来先服务原则。
有的书上把这种算法称为“读写公平法”。

```
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);
    }
}
```

经典进程同步问题——哲学家进餐问题

一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭。哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左、右两根筷子（一根一根地拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。



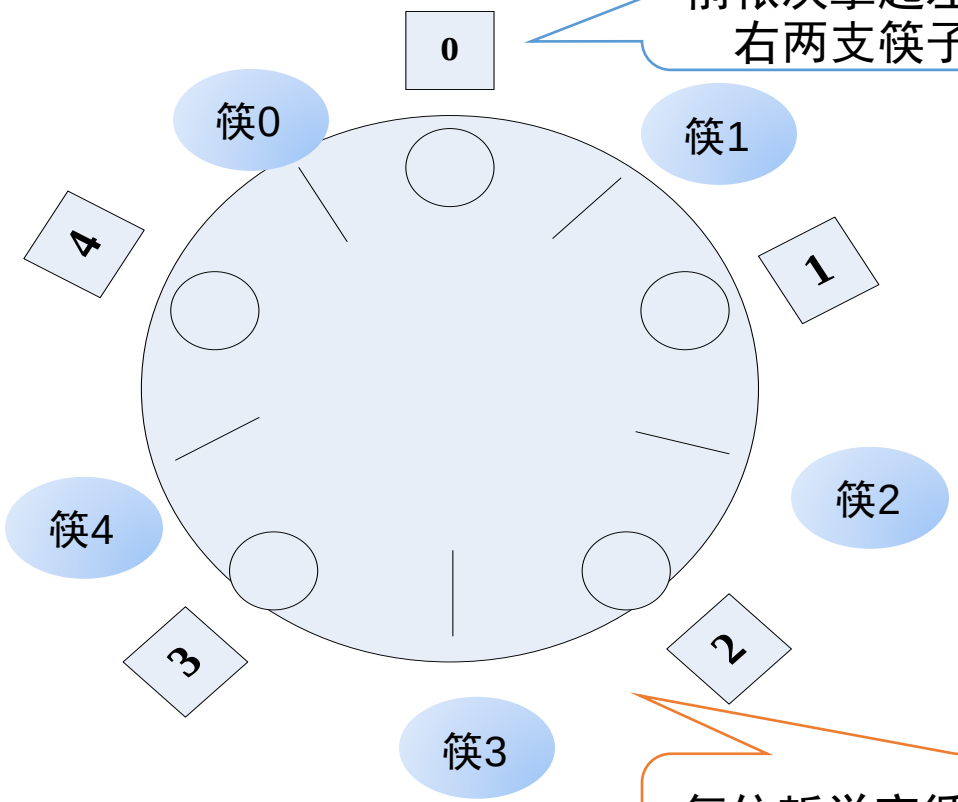
1. 关系分析。系统中有5个哲学家进程，5位哲学家与左右邻居对其中间筷子的访问是互斥关系。

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事，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现象，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。

经典进程同步问题——哲学家进餐问题

每个哲学家吃饭前依次拿起左、右两支筷子



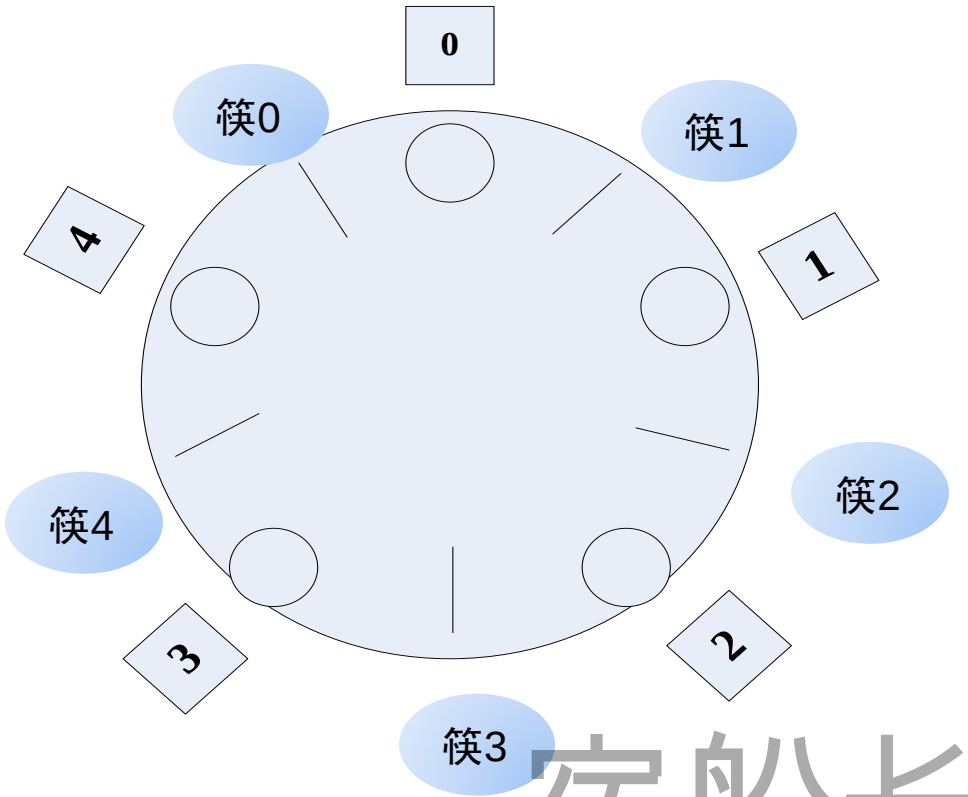

```
semaphore chopstick[5]={1,1,1,1,1};
Pi (){ //i号哲学家的进程
    while(1){
        P(chopst          //拿左
        P(chopsti      1)%5]); //拿右
        吃饭…
        V(chopst          //放左
        V(chop      [(    %5]); //放右
        思考…
    }
}
```

每位哲学家循环等待右边的人放下筷子（阻塞）。发生“死锁”

如果5个哲学家并发地拿起了自己左手边的筷子…

经典进程同步问题——哲学家进餐问题

一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭。哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左、右两根筷子（一根一根地拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。



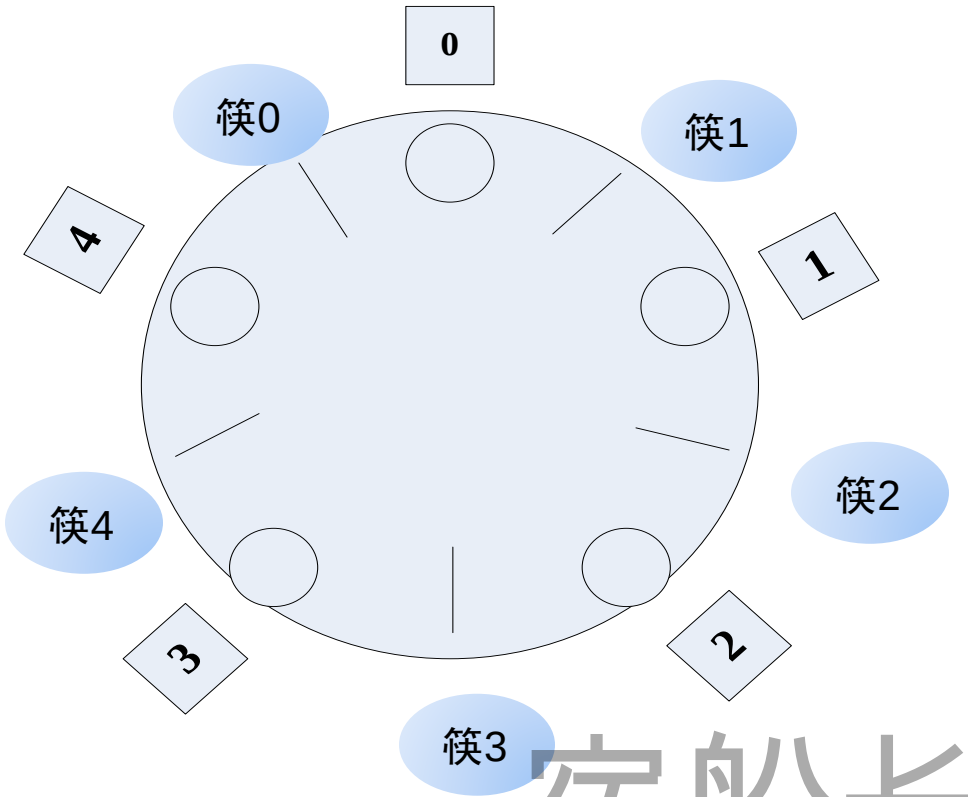
Q：如何防止死锁的发生呢？

A：①可以对哲学家进程施加一些限制条件，比如最多允许四个哲学家同时进餐。这样可以保证至少有一个哲学家是可以拿到左右两只筷子的

②要求奇数号哲学家先拿左边的筷子，然后再拿右边的筷子，而偶数号哲学家刚好相反。用这种方法可以保证如果相邻的两个奇偶号哲学家都想吃饭，那么只会有其中一个可以拿起第一只筷子，另一个会直接阻塞。这就避免了占有一支后再等待另一只的情况。

经典进程同步问题——哲学家进餐问题

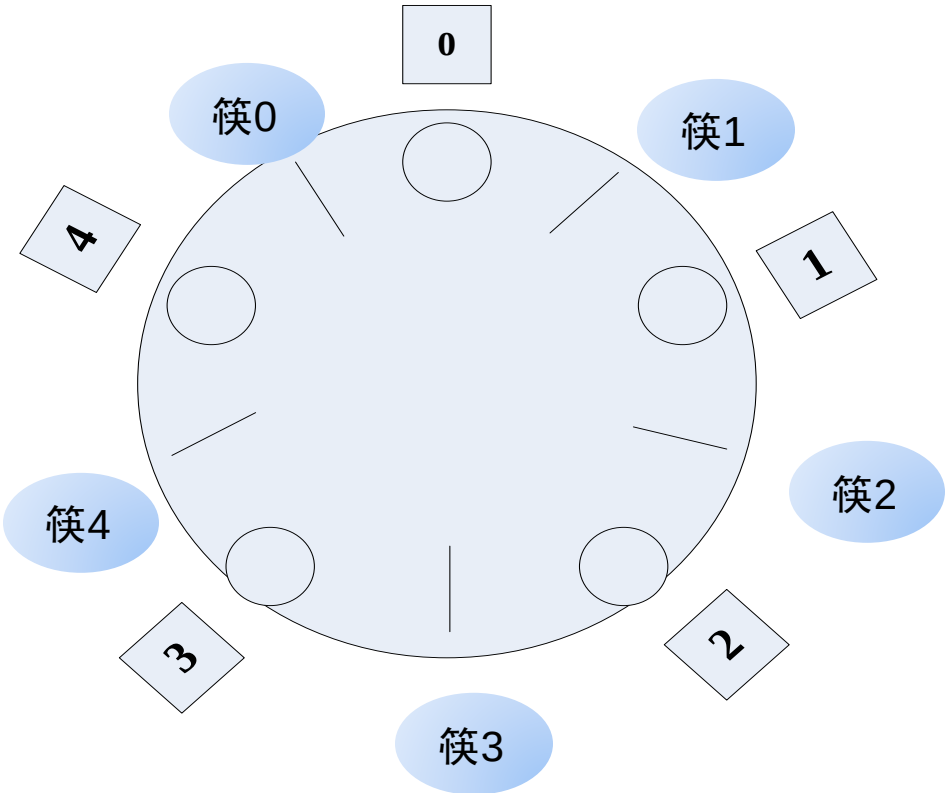
一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭。哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左、右两根筷子（一根一根地拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。



Q：如何防止死锁的发生呢？

A：③仅当一个哲学家左右两支筷子都可用时才允许他抓起筷子。

经典进程同步问题——哲学家进餐问题



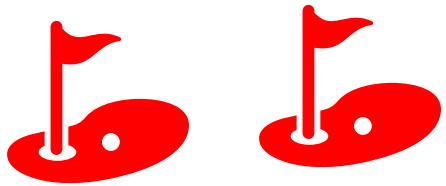
Q：如何防止死锁的发生呢？

A：③仅

他抓起筷子

各哲学家拿筷子这件事必须互斥的执行。这就保证了即使一个哲学家在拿筷子拿到一半时被阻塞，也不会有别的哲学家会继续尝试拿筷子。这样的话，当前正在吃饭的哲学家放下筷子后，被阻塞的哲学家就可以获得等待的筷子了。

```
semaphor
semaphor
Pi  (){ //i号哲      的进程
    while(1){
        P(mutex);
        P(chopstick[i]); //拿左
        P(chopstick[(i+1)%5]); //拿右
        V(mutex);
        吃饭…
        V(chopstick[i]); //放左
        V(chopstick[(i+1)%5]); //放右
        思考…
    }
}
```



死锁问题

死锁——死锁、饥饿、死循环的区别

死锁：各进程互相等待对方手里的资源，导致各进程都阻塞，无法向前推进的现象。
饥饿：由于长期得不到想要的资源，某进程无法向前推进的现象。比如：在短进程优先（SPF）算法中，若有源源不断的短进程到来，则长进程将一直得不到处理机，从而发生长进程“饥饿”。
死循环：某进程执行过程中一直跳不出某个循环的现象。有时是因为程序逻辑bug导致的，有时是程序员故意设计的。

	共同点	区别
死锁	都是进程无法顺利向前推进的现象（故意设计的死循环除外）	死锁一定是“循环等待对方手里的资源”导致的，因此如果有死锁现象，那至少有两个或两个以上的进程同时发生死锁。另外，发生死锁的进程一定处于阻塞态。
饥饿		可能只有一个进程发生饥饿。发生饥饿的进程既可能是阻塞态(如长期得不到需要的I/O设备)，也可能是就绪态(长期得不到处理机)
死循环		可能只有一个进程发生死循环。死循环的进程可以上处理机运行（可以是运行态），只不过无法像期待的那样顺利推进。死锁和饥饿问题是由于操作系统分配资源的策略不合理导致的，而死循环是由代码逻辑的错误导致的。死锁和饥饿是管理者（操作系统）的问题，死循环是被管理者的问题。

死锁——死锁产生的必要条件

产生死锁必须同时满足一下四个条件，只要其中任一条件不成立，死锁就不会发生。、

互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁（如哲学家的筷子、打印机设备）。像内存、扬声器这样可以同时让多个进程使用的资源是不会导致死锁的（因为进程不用阻塞等待这种资源）。
不剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。
请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。
循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

注意！发生死锁时一定有循环等待，但是发生循环等待时未必死锁（循环等待是死锁的必要不充分条件）
如果同类资源数大于1，则即使有循环等待，也未必发生死锁。但如果系统中每类资源都只有一个，那循环等待就是死锁的充分必要条件了。

死锁——什么时候会发生死锁

1. 对系统资源的竞争。各进程对不可剥夺的资源（如打印机）的竞争可能引起死锁，对可剥夺的资源（CPU）的竞争是不会引起死锁的。

2. 进程推进顺序非法。请求和释放资源的顺序不当，也同样会导致死锁。例如，并发执行的进程P1、P2分别申请并占有了资源R1、R2，之后进程P1又紧接着申请资源R2，而进程P2又申请资源R1，两者会因为申请的资源被对方占有而阻塞，从而发生死锁。

3. 信号量的使用不当也会造成死锁。如生产者-消费者问题中，如果实现互斥的P操作在实现同步的P操作之前，就有可能导致死锁。（可以把互斥信号量、同步信号量也看做是一种抽象的系统资源）

总之，对不可剥夺资源的不合理分配，可能导致死锁。

死锁——死锁的处理策略

1. 预防死锁。破坏死锁产生的四个必要条件中的一个或几个。

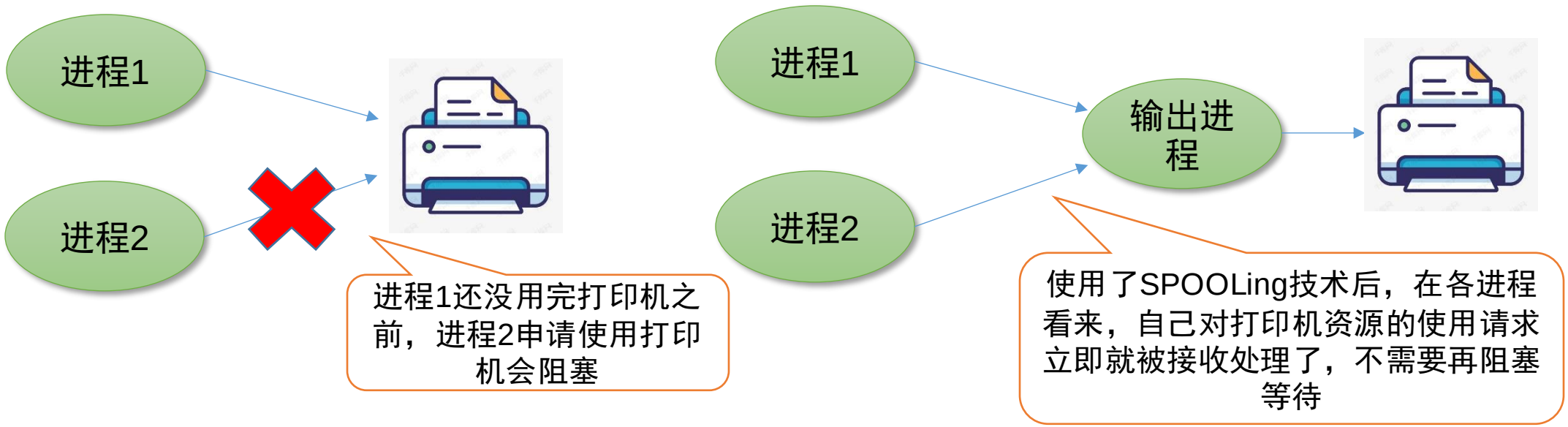
2. 避免死锁。用某种方法防止系统进入不安全状态，从而避免死锁（银行家算法）

3. 死锁的检测和解除。允许死锁的发生，不过操作系统会负责检测出死锁的发生，然后采取某种措施解除死锁。

死锁的处理策略——预防死锁（破坏互斥条件）

互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁。

如果把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态。比如：SPOOLing技术。操作系统可以采用SPOOLing 技术把独占设备在逻辑上改造成共享设备。比如，用SPOOLing技术将打印机改造为共享设备…



该策略的缺点：并不是所有的资源都可以改造成可共享使用的资源。并且为了系统安全，很多地方还必须保护这种互斥性。因此，很多时候都无法破坏互斥条件。

死锁的处理策略——预防死锁（破坏不剥夺条件）

不剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。

破坏不剥夺条件：
方案一：当某个进程请求新的资源得不到满足时，它必须立即释放保持的所有资源，待以后需要时再重新申请。也就是说，即使某些资源尚未使用完，也需要主动释放，从而破坏了不可剥夺条件。

方案二：当某个进程需要的资源被其他进程所占有的时候，可以由操作系统协助，将想要的资源强行剥夺。这种方式一般需要考虑各进程的优先级（比如：剥夺调度方式，就是将处理机资源强行剥夺给优先级更高的进程使用）

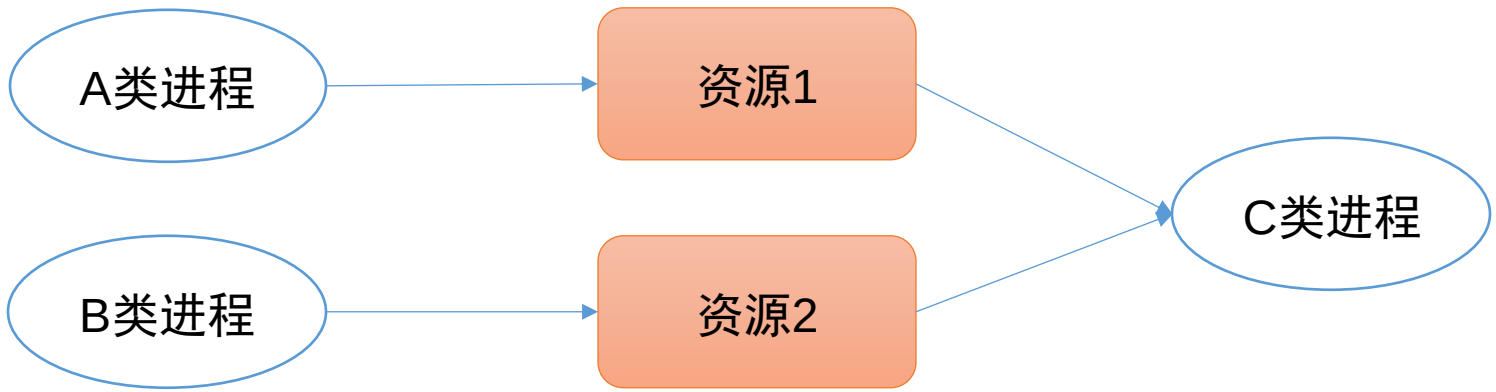
该策略的缺点：
1. 实现起来比较复杂。
2. 释放已获得的资源可能造成前一阶段工作的失效。因此这种方法一般只适用于易保存和恢复状态的资源，如CPU。
3. 反复地申请和释放资源会增加系统开销，降低系统吞吐量。
4. 若采用方案一，意味着只要暂时得不到某个资源，之前获得的那些资源就都需要放弃，以后再重新申请。如果一直发生这样的情况，就会导致进程饥饿。

死锁的处理策略——预防死锁（破坏请求和保持条件）

请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。

可以采用静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源了。

该策略实现起来简单，但也有明显的缺点：
有些资源可能只需要用很短的时间，因此如果进程的整个运行期间都一直保持着所有资源，就会造成严重的资源浪费，资源利用率极低。另外，该策略也有可能导致某些进程饥饿。



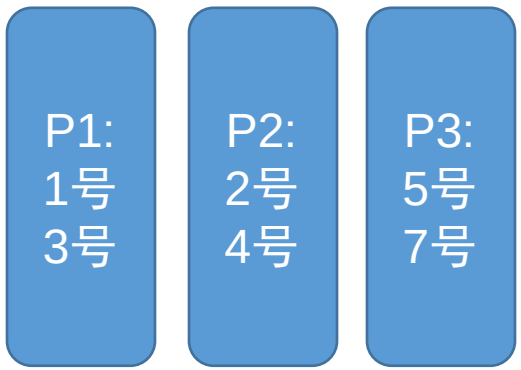
死锁的处理策略——预防死锁（破坏循环等待条件）

循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

可采用顺序资源分配法。首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源（即编号相同的资源）一次申请完。

原理分析：一个进程只有已占有小编号的资源时，才有资格申请更大编号的资源。按此规则，已持有大编号资源的进程不可能逆向地回来申请小编号的资源，从而就不会产生循环等待的现象。

假设系统中共有10个资源，编号为1，2，…… 10



在任何一个时刻，总有一个进程拥有的资源编号是最大的，那这个进程申请之后的资源必然畅通无阻。因此，不可能出现所有进程都阻塞的死锁现象

该策略的缺点：
1. 不方便增加新的设备，因为可能需要重新分配所有的编号；
2. 进程实际使用资源的顺序可能和编号递增顺序不一致，会导致资源浪费；
3. 必须按规定次序申请资源，用户编程麻烦。

死锁的处理策略——避免死锁（系统安全状态）

安全状态定义

设系统中有n个进程，若存在一个进程序列<P_1，P_2，…，P_n>。使得进程P_i（i=1, 2，…，n）以后还需要的资源可以通过系统现有空闲资源加上所有P_j（j<i）已占有的资源来满足，则称此时系统处于安全状态，进程序列<P_1，P_2，…，P_n>称为安全序列，因为各进程至少可以按照安全序列中的顺序依次执行完成。

如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

死锁的处理策略——避免死锁（系统安全状态）

安全状态举例：

假设某系统共有15台磁带机和三个进程P_0、P_1、P_2，各进程对磁带机的最大需求数量、T_0时刻已经分配到的磁带机数量、还需要的磁带机数量以及系统剩余的可用磁带机数量如下表所示：

进程	最大需求	已分配数量	还需要的数量	剩余可用数量
P_0	12	6	6	4
P_1	5	2	3	
P_2	10	3	7	

由安全状态向不安全状态的转换：

如果不按照安全顺序分配资源，则系统可能由安全状态进入不安全状态。

死锁的处理策略——避免死锁（银行家算法）

■ 基本思想：

Dijkstra E.W 于1968年提出银行家算法：

它的模型基于一个小城镇的银行家，该算法可描述如下：假定一个银行家拥有资金，数量为Σ，被N个客户共享。银行家对客户提出下列约束条件：

1. 每个客户必须预先说明自己所要求的最大资金量；
2. 每个客户每次提出部分资金量申请；
3. 如果银行满足了某客户对资金的最大需求量，那么，客户在资金运作后，应在有限时间内全部归还银行。

死锁的处理策略——避免死锁（银行家算法）

■ 数据结构：

① 可利用资源向量Available[m]。

其中的每一个元素代表一类可利用的资源数目

Available[j]=K，则表示系统中现有R_j类资源K个。

② 最大需求矩阵Max[1..n,1..m]

该矩阵定义了系统中n个进程对m类资源的最大需求。

Max[i,j]=K，则表示进程i需要R_j类资源的最大数目为K。

③ 分配矩阵Allocation[1..n,1..m]

该矩阵表示系统中每个进程当前已分配到的每类资源数量。

Allocation[i,j]=K，表示进程i当前已分得R_j类资源的数目为K。

④ 需求矩阵Need[1..n,1..m]

该矩阵表示每个进程尚需的各类资源数。Need[i,j]=K，则表示进程i还需要R_j类资源K个，方能完成其任务。

Need[i,j]=Max[i,j]-Allocation[i,j]

⑤ 请求向量Requesti[m];

某进程提出的资源请求向量。

死锁的处理策略——避免死锁（银行家算法）

■ 算法描述

当进程P_i提出资源申请Requesti[m]时，系统执行下列步骤：

(1) 若Request[i]≤Need[i],转(2)；否则错误返回；

(2) 若Request[i]≤Available，转(3)；否则，表示尚无足够资源，P_i须等待；

(3)系统尝试把资源分配给进程P_i，并修改以下数据结构：

Available:= Available-Request[i];

Allocation[i]:= Allocation[i]+Request[i];

Need[i]:= Need[i]-Request[i];

(4) 执行安全性算法：

检查此次资源分配后，系统是否处于安全状态。若安全，则将资源分配给进程P_i，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程P_i等待。

死锁的处理策略——避免死锁（银行家算法）

■安全性算法描述

当进程P_i提出资源申请Requesti[m]时，系统执行下列步骤：

① 设置两个临时向量：

- 工作向量Work：表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work=Available;
- Finish[n]：它表示系统是否有足够的资源分配给进程，使之运行完成。

开始时先做**Finish［i］=false;** 当有足够资源分配给进程时， 再令**Finish［i］=true**。

②从进程集合中找到一个能满足下述条件的进程：

1） Finish［i］=false； 2）Need［i,j］≤Work［j］；

若找到， 执行步骤③， 否则，执行步骤④；

③ 当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］= Work［j］+Allocation［i,j］；

Finish［i］= true;

go to step ②;

④ 如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态

死锁的处理策略——避免死锁（银行家算法）

例1： 假定系统中有五个进程{P_0, P_1, P_2, P_3, P_4}和三类资源{A, B, C}，各种资源的数量分别为10、5、7，在T_0时刻的资源分配情况如图所示。

资源 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
P_1	3 2 2	2 0 0	1 2 2	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

（1） T_0时刻的安全性；

（2）P1请求资源：Request1（1，0，2）；

死锁的处理策略——避免死锁（银行家算法）

资源 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
P_1	3 2 2	2 0 0	1 2 2	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

（1） T_0时刻的安全性；

资源 / 进程	Work	Need	Allocation	Work+Allocation	Finish
	A B C	A B C	A B C	A B C	
P_1	3 3 2	1 2 2	2 0 0	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0	10 4 7	7 4 3	0 1 0	10 5 7	true

由于T_0时刻存在安全序列{P_1, P_3, P_4, P_2, P_0},故此时系统是安全的。

死锁的处理策略——避免死锁（银行家算法）

（2）P1请求资源：Request1（1，0，2）；

资源 / 情况 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	2, 3, 0
P_1	3 2 2	3, 0, 2	0, 2, 0	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

（2）当P_1请求资源：$Request_1$（1，0，2）时：

① $Request_1$（1，0，2）≦ $Need_1$（1，2，2）

② $Request_1$（1，0，2）≦ Available（3，3，2）

③试分配资源后，修改数据结构。

④对试分配后状态进行安全性检查：

死锁的处理策略——避免死锁（银行家算法）

（2）P1请求资源：Request1（1，0，2）；

资源情况 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	2 3 0
P_1	3 2 2	3 0 2	0 2 0	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

资源情况 / 进程	Work	Need	Allocation	Work+Allocation	Finish
	A B C	A B C	A B C	A B C	
P_1	2 3 0	0 2 0	3 0 2	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0	10 4 7	7 4 3	0 1 0	10 5 7	true

由于此时存在安全序列{P_1, P_3, P_4, P_2, P_0},故系统是安全的，可为P_1分配上述资源。

死锁的处理策略——避免死锁（银行家算法）

（3）当P4请求资源：Request4（3，3，0）时：

资源 / 情况 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	2 3 0
P_1	3 2 2	3 0 2	0 2 0	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

① $Request_4$（3，3，0） ≦ $Need_4$（4，3，1）成立；

② $Request_4$（3，3，0） ≦ Available（2，3，0）不成立，故让P_4等待。

死锁的处理策略——避免死锁（银行家算法）

4）当P0请求资源：Request0（0，2，0）时：

资源 / 情况 / 进程	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0, 3, 0	7, 2, 3	2, 1, 0
P_1	3 2 2	3 0 2	0 2 0	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

① Request0（0，2，0） ≦ Need0 （7，4，3）成立；

② Request0（0，2，0） ≦ Available（2，3，0）成立；

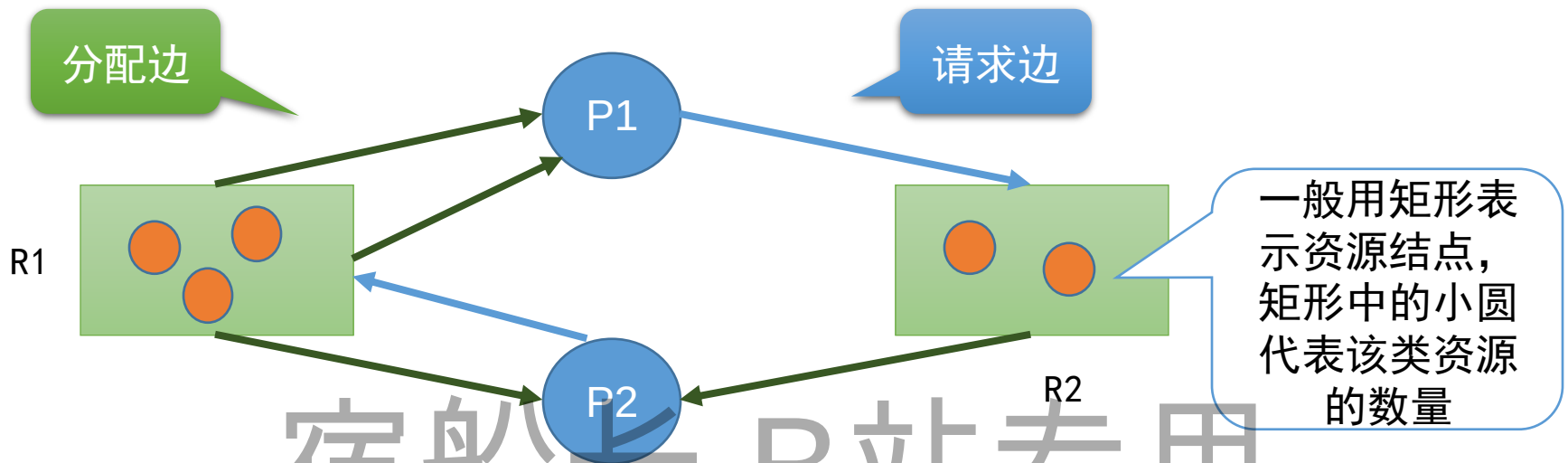
③试分配资源后，修改数据结构。

④对试分配后的状态进行安全性检查：由于Available（2,1,0）已不能满足任何进程的需要，故系统进入不安全状态，所以不能为P0分配资源，而应恢复原来的状态，让P0等待。

死锁的处理策略——检测和解除（死锁的检测）

为了能对系统是否已发生了死锁进行检测，必须：
①用某种数据结构来保存资源的请求和分配信息；
②提供一种算法，利用上述信息来检测系统是否已进入死锁状态。

资源分配图
- 两种结点
  - 进程结点：对应一个进程
  - 资源结点：对应一类资源，一类资源可能有多个
- 两种边
  - 进程结点——> 资源结点:表示进程想申请几个资源（每条边代表一个）
  - 资源节点——> 进程结点：表示已经为进程分配了几个资源（每条边代表一个）

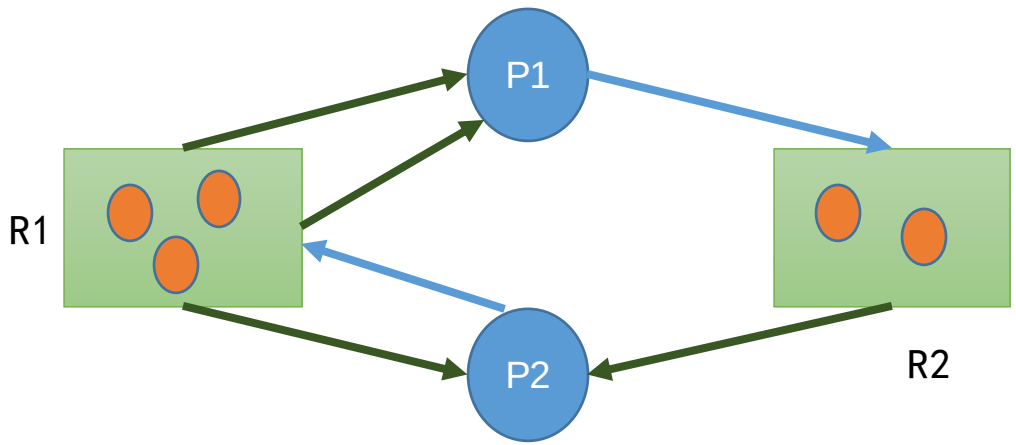


死锁的处理策略——检测和解除（死锁的检测）

为了能对系统是否已发生了死锁进行检测，必须：
①用某种数据结构来保存资源的请求和分配信息；
②提供一种算法，利用上述信息来检测系统是否已进入死锁状态。

如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。
如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。相应的，这些被激活的进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程…

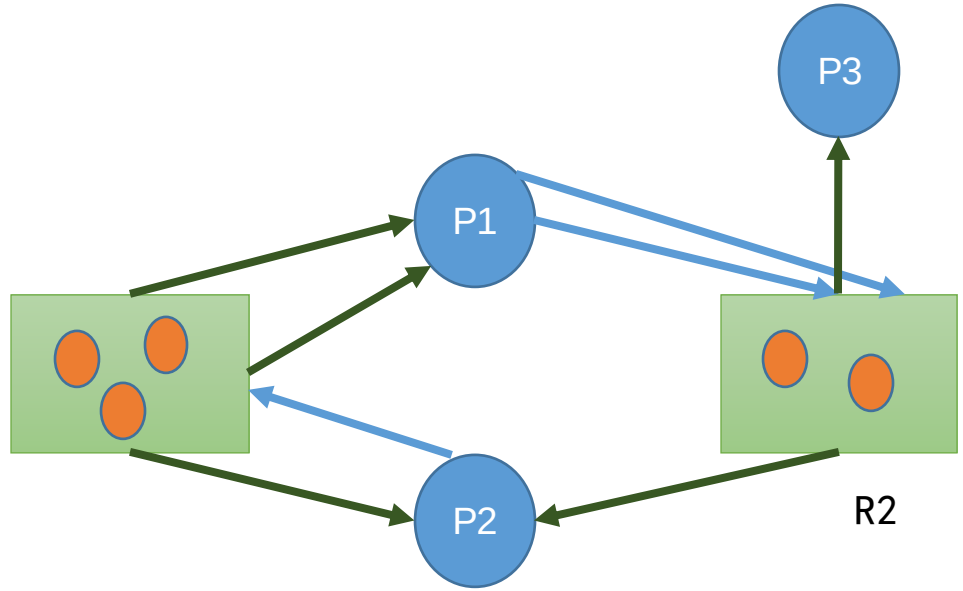
如果按上述过程分析，最终能消除所有边，就称这个图是*可完全简化的*。此时一定没有发生死锁（相当于能找到一个安全序列）



死锁的处理策略——检测和解除（死锁的检测）

为了能对系统是否已发生了死锁进行检测，必须：
①用某种数据结构来保存资源的请求和分配信息；
②提供一种算法，利用上述信息来检测系统是否已进入死锁状态。

如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。
如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。相应的，这些被激活的进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程…



如果按上述过程分析，最终能消除所有边，就称这个图是*可完全简化的*。此时一定没有发生死锁（相当于能找到一个安全序列）

如果最终不能消除所有边，那么此时就是发生了死锁。

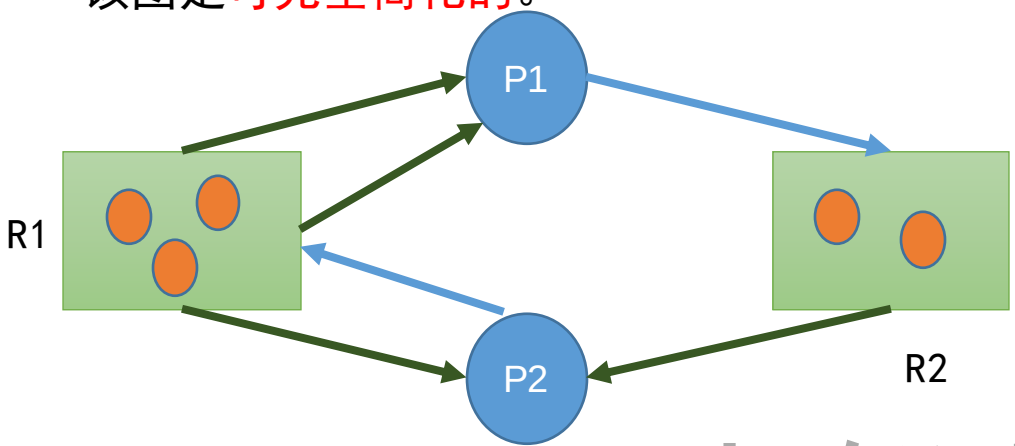
最终还连着边的那些进程就是处于死锁状态的进程。

死锁的处理策略——检测和解除（死锁的检测）

检测死锁的算法：

1）在资源分配图中，找出既不阻塞又不是孤点的进程Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。如下图中，R1没有空闲资源，R2有一个空闲资源。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配变，使之称为孤立的结点。在下图中，P1是满足这一条件的进程结点，于是将P1的所有边消去。

2）进程Pi所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在下图中，P2就满足这样的条件。根据1）中的方法进行一系列简化后，若能消去途中所有的边，则称该图是可完全简化的。



死锁定理：如果某时刻系统的资源分配图是不可完全简化的，那么此时系统死锁

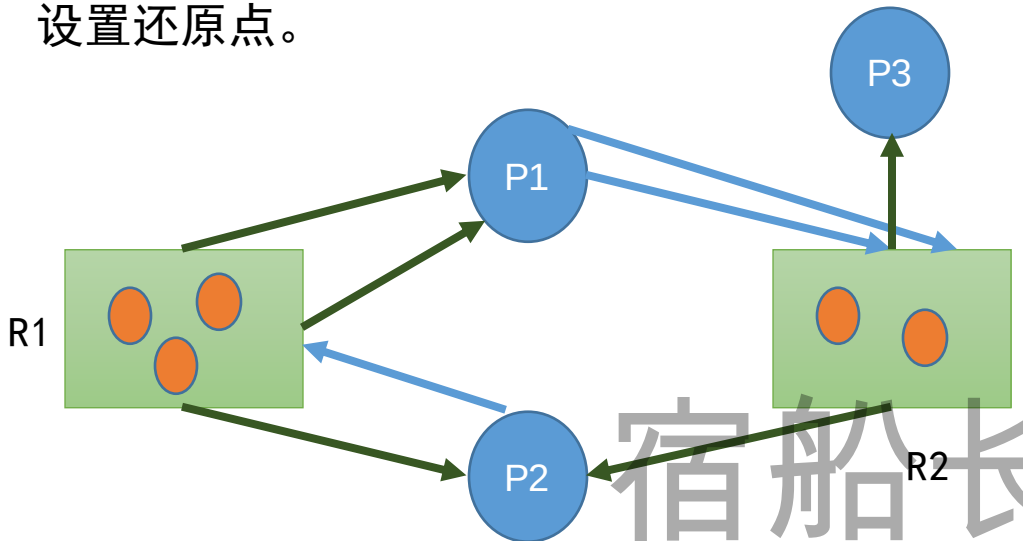
死锁的处理策略——检测和解除（死锁的解除）

一旦检测出死锁的发生，就应该立即解除死锁。

补充：并不是系统中所有的进程都是死锁状态，用死锁检测算法化简资源分配图后，还连着边的那些进程就是死锁进程

解除死锁的主要方法有：

1. 资源剥夺法。挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。
2. 撤销进程法（或称终止进程法）。强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。
3. 进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。



如何决定“对谁动手”

1. 进程优先级
2. 已执行多长时间
3. 还要多久能完成
4. 进程已经使用了多少资源
5. 进程是交互式的还是批处理式的

拜拜

宿船长 B站专用