第6章 进程同步

6.3 信号量

6.3 信号量

- 前节种种软件方法解决临界区调度问题的缺点:
  - 对不能进入临界区的进程，采用忙式等待测试法，浪费CPU时间。
  - 将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重了用户编程负担。

- 1965年，E．W．Dijkstra提出了新的同步工具:信号量和P、V操作

6.3.1 信号量的定义

- 信号量(semaphore)由两个成员（s，q）组成
  - 其中s是一个具有非负初值的整型变量
  - q是一个初始状态为空的队列。

- 除信号量的初值外，信号量的值仅能由P操作（又称为wait操作）和V操作（又称为signal操作）改变。

P操作

- 设S=（s，q）为一个信号量，P（S）执行时主要完成下述动作：
  - s = s - 1;
  - If (s < 0) {

    设置进程状态为等待;

    将进程放入信号量等待队列q;

    转调度程序;

    }

V操作

- V(S)执行时主要完成下述动作：
  - s = s + 1;
  - If(s≤0){

    将信号量等待队列q中的第一个进程移出;

    设置其状态为就绪状态并插入就绪队列;

    然后再返回原进程继续执行;

    }

几个重要含义

- 信号量中的整型变量s表示系统中某类资源的数目。
- 当s>0时，
  - 该值等于在封锁进程之前对信号量S可施行的P操作数，
  - 等于S所代表的实际还可以使用的资源数
- 当s<0时，
  - 其绝对值等于登记排列在该信号量S队列之中等待的进程个数，
  - 亦即恰好等于对信号量s实施P操作而被封锁起来并进入信号量s队列的进程数

注意

- 通常，P操作意味着请求一个资源，V操作意味着释放一个资源。
  - 在一定条件下，P操作代表使进程阻塞的操作，而V操作代表唤醒阻塞进程的操作。

- P、V操作的原子性要求
  - 即，一个进程在信号量上操作时，不会有别的进程同时修改该信号量。
  - 对于单处理器，可简单地在**封锁中断**的情况下执行
  - 对于多处理器环境，需要**封锁所有处理器的中断**，困难且影响性能，往往用swap()或自旋锁等方式加锁

信号量的另一种描述

```
typedef struct{
   int value;
   struct process *list;
   } semaphore;

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

6.3.2 利用信号量实现互斥

- 设S为两个进程P1、P2实现互斥的信号量
  - S的初值应为1，即可用资源数目为1。
- 只需把临界区置于P（S）和V（S）之间，即可实现两进程的互斥。

进程P1：

```
P(S);
进程P1的临界区;
V(S);
```

进程P2：

```
P(S);
进程P2的临界区;
V(S);
```

一个老问题……

- 现有订票系统，x为存储某班次飞机剩余票数的存储区域
- A：

```
R1=x;
if (R1>=1) {
    R1--;
    x=R1;
    {出票}
    }
```

B：

```
R2=x;
if (R2>=1){
   R2--;
   x=R2;
   {出票}
   }
```

一个信号量实现互斥的例子

改进订票系统：

```
A:                      B:
P(S);                   P(S);
R1=x;                   R2=x;
if (R1>=1) {            if (R2>=1){
    R1--;                   R2--;
    x=R1;                   x=R2;
    V(S);                    V(S)
    {出票}                  {出票}
    }                       }
    else{                    else{
    V(S)                     V(S)
    {提示无票}              {提示无票}
    }                        }
```

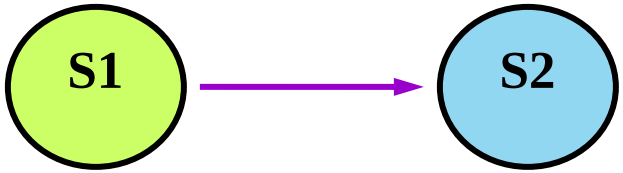
互斥信号量的取值范围

- 若2个进程共享一个临界资源，信号量的取值范围是：
  - 若没有进程使用临界资源　　1
  - 若只有1个进程使用临界资源　　0
  - 若1个进程使用临界资源，另1个进程等待使用临界资源　　-1

- Q：若N个进程共享一个临界资源，其取值范围如何?
- Q：若N个进程共享M个临界资源，信号量初值是多少？其取值范围如何?

6.3.3 利用信号量实现前趋关系

- 前趋关系：并发执行的进程P1和P2，分别有代码S1和S2，要求S1在S2开始前完成



- 设置一个信号量S，其初值为0

进程P1：

S1;

V(S);

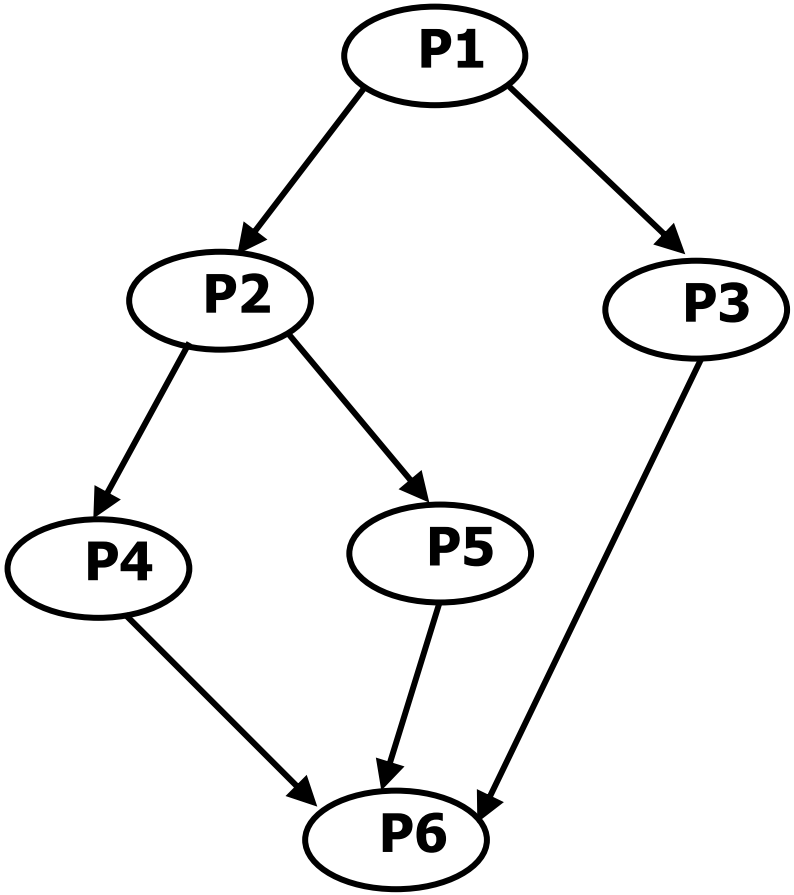
进程P2：

P(S);

S2;

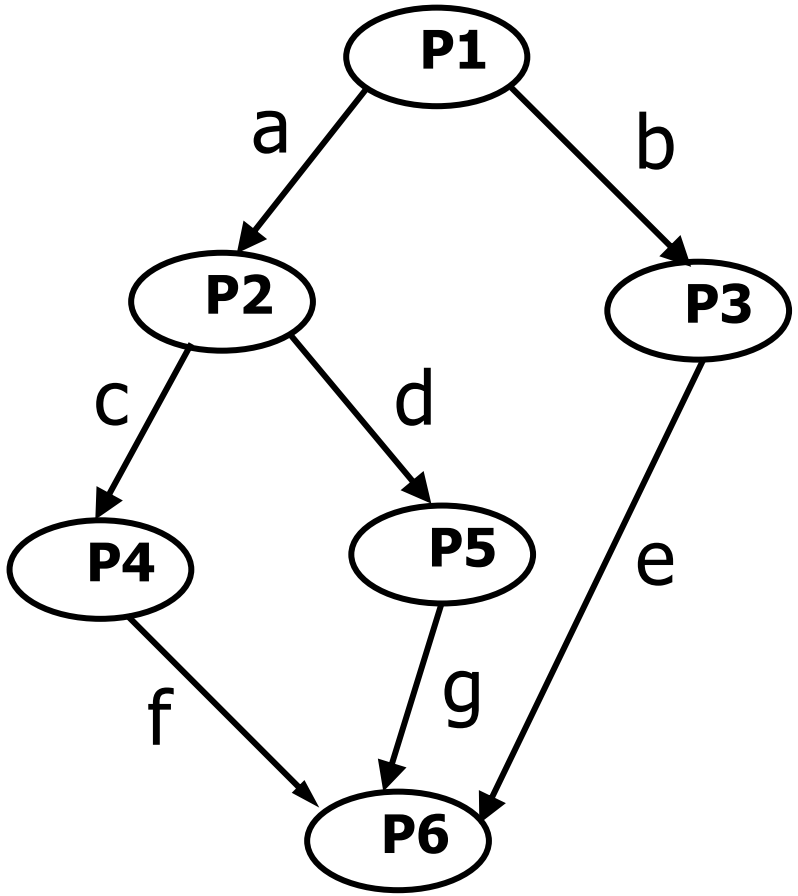
利用信号量实现前趋关系举例

- 例如：P1、P2、P3、P4、P5、P6为一组合作进程，其前驱图如下所示，试用P、V操作完成这六个进程的同步。



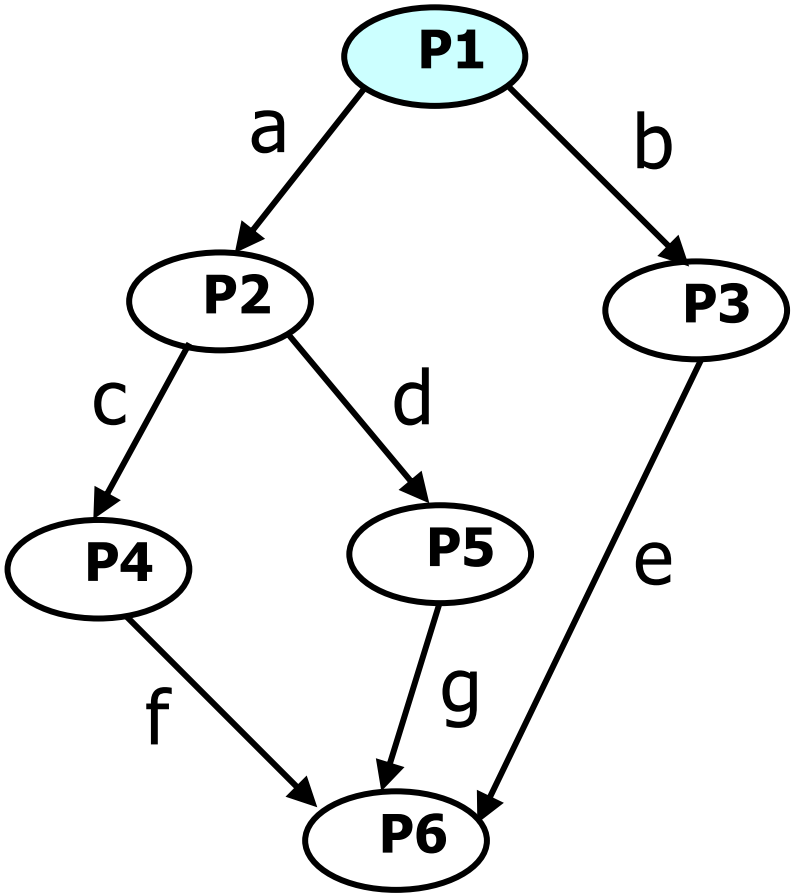
解法1

- 设七个同步信号量a、b、c、d、e、f、g分别表示进程之间的前驱关系，如图所示，**其初值均为0**。这六个进程的同步描述如下：



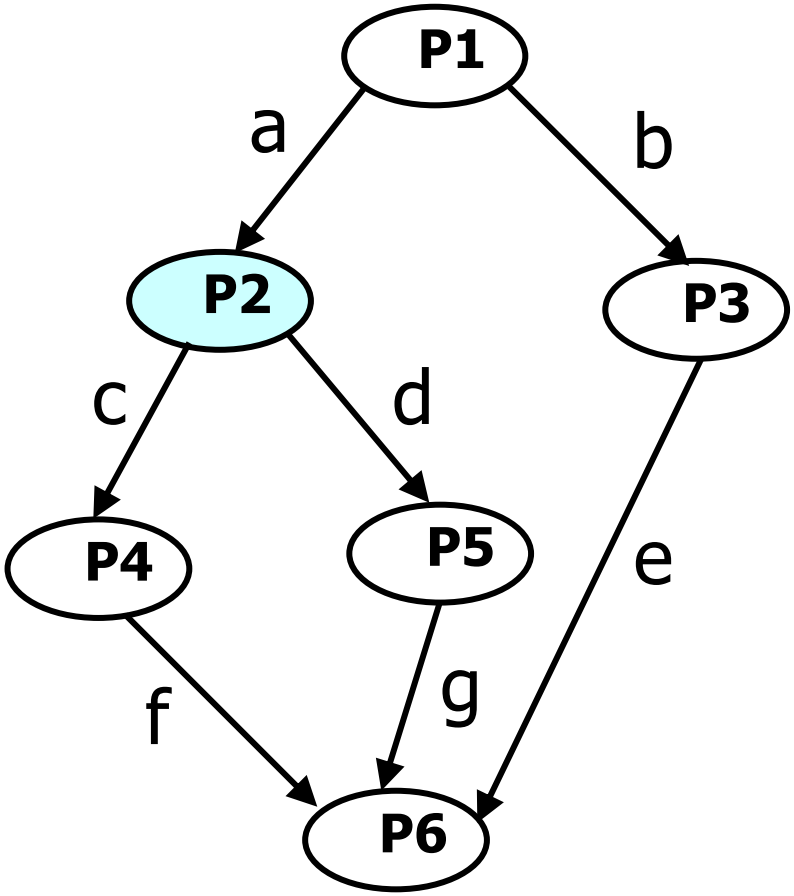
解法1（1）

```
P1()
{
    执行P1的代码;
     v(a);
     v(b);
 }
```



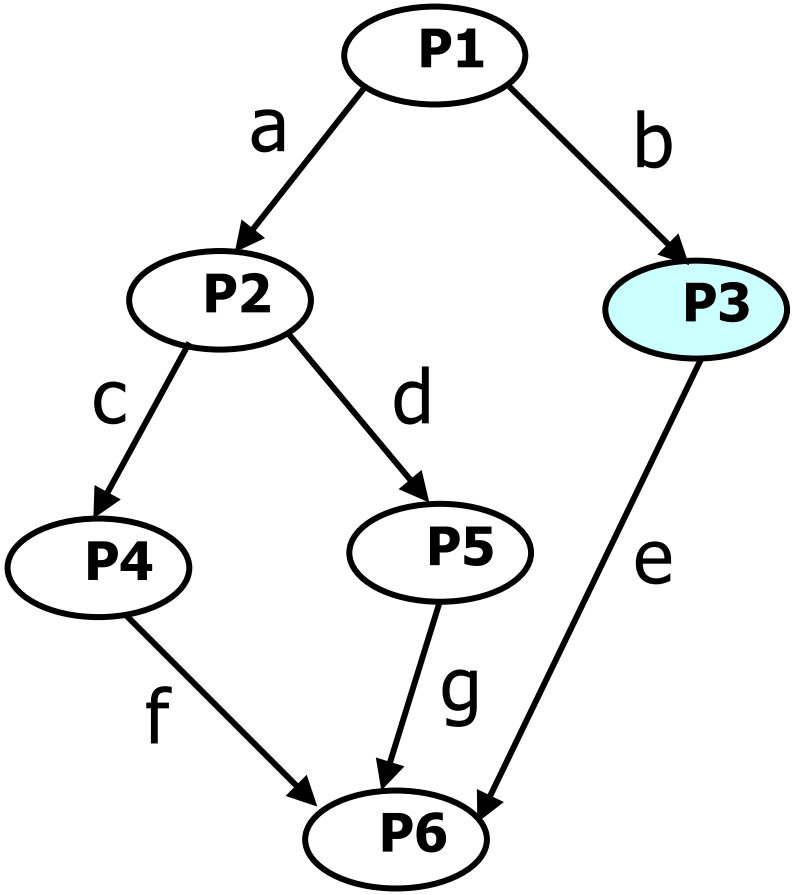
解法1（2）

```
P2 ()
{
    p(a);
    执行P2的代码;
    v(c);
    v(d);
 }
```



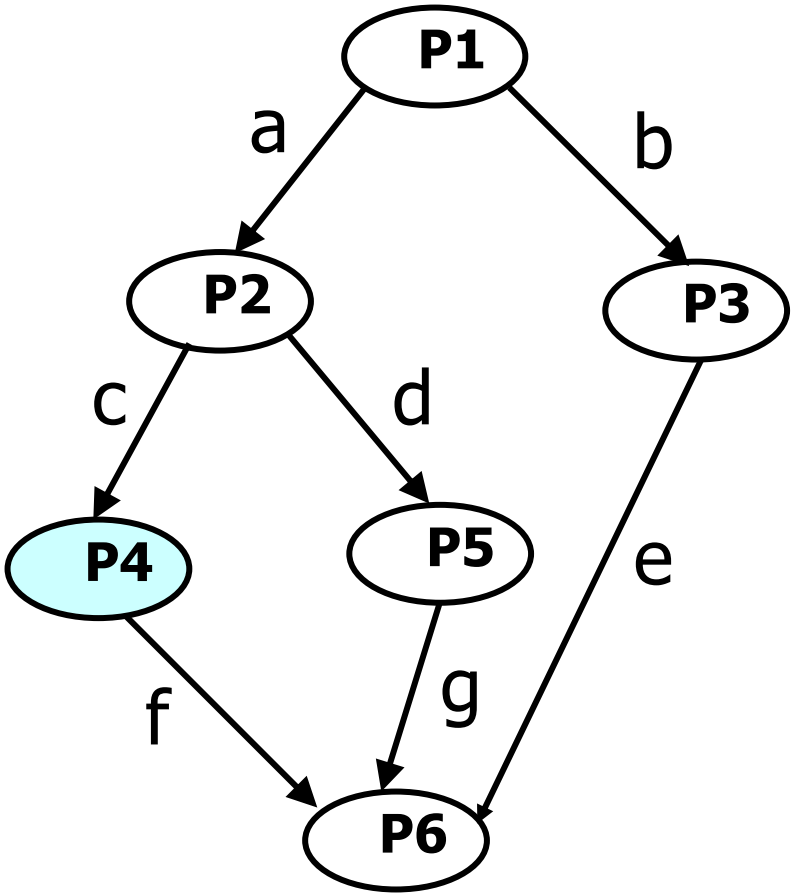
解法1（3）

```
P3 ()
{
    p(b);
    执行P3的代码;
    v(e);
 }
```



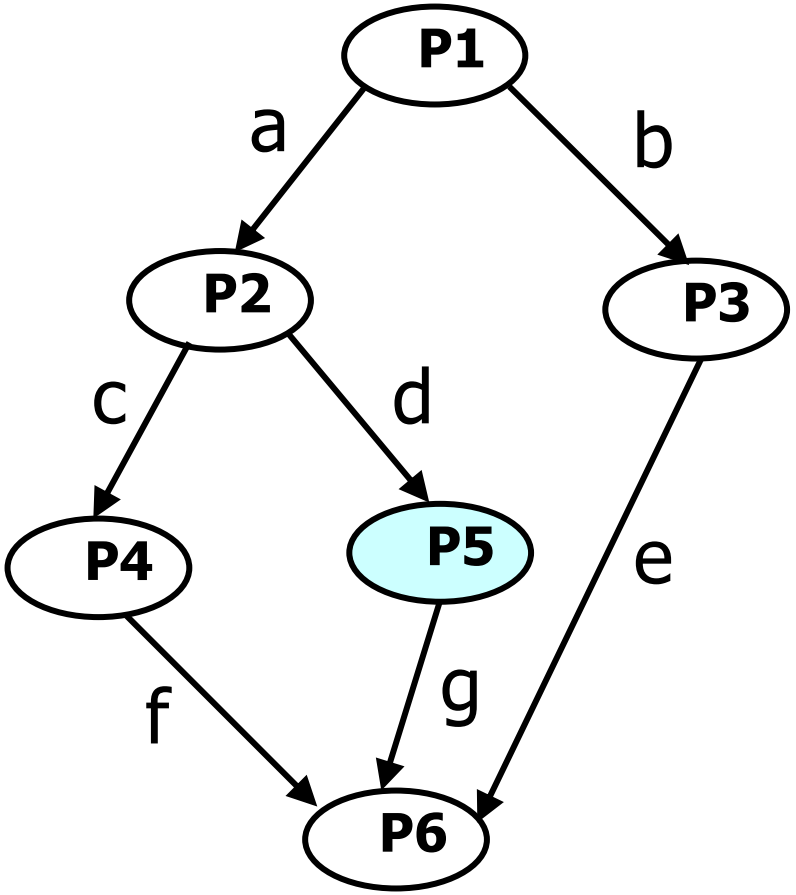
解法1（4）

```
P4 ()
{
    p(c);
    执行P4的代码;
    v(f);
}
```



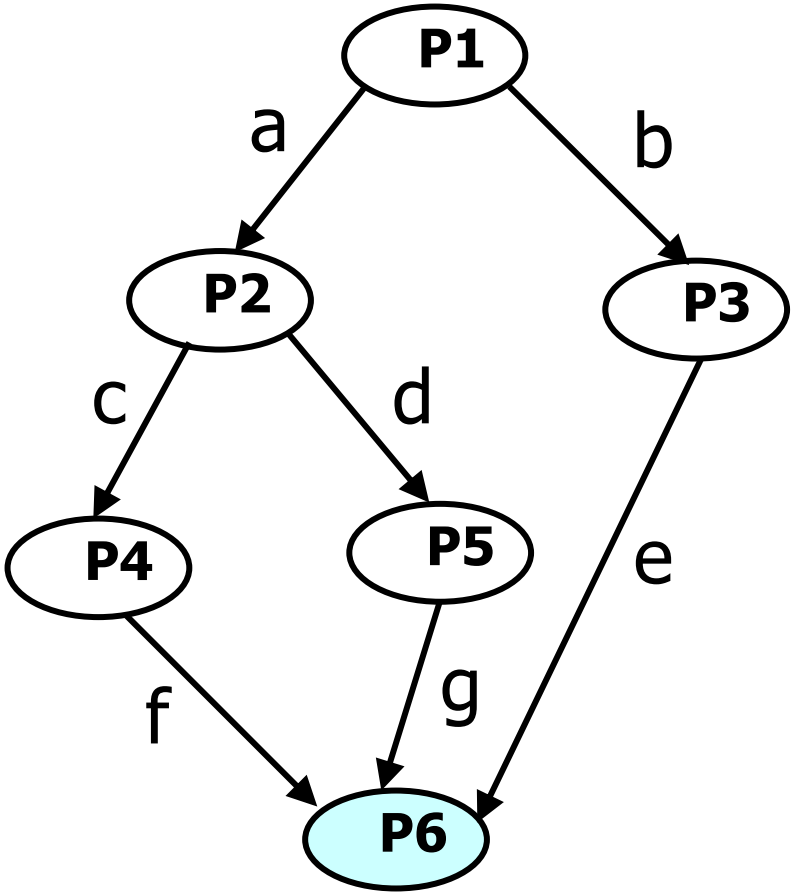
解法1（5）

```
P5 ()
{
    p(d);
   执行P5的代码;
    v(g);
}
```



解法1（6）

```
P6 ()
{
      p(e);
      p(f);
      p(g);
     执行P6的代码;
}
```

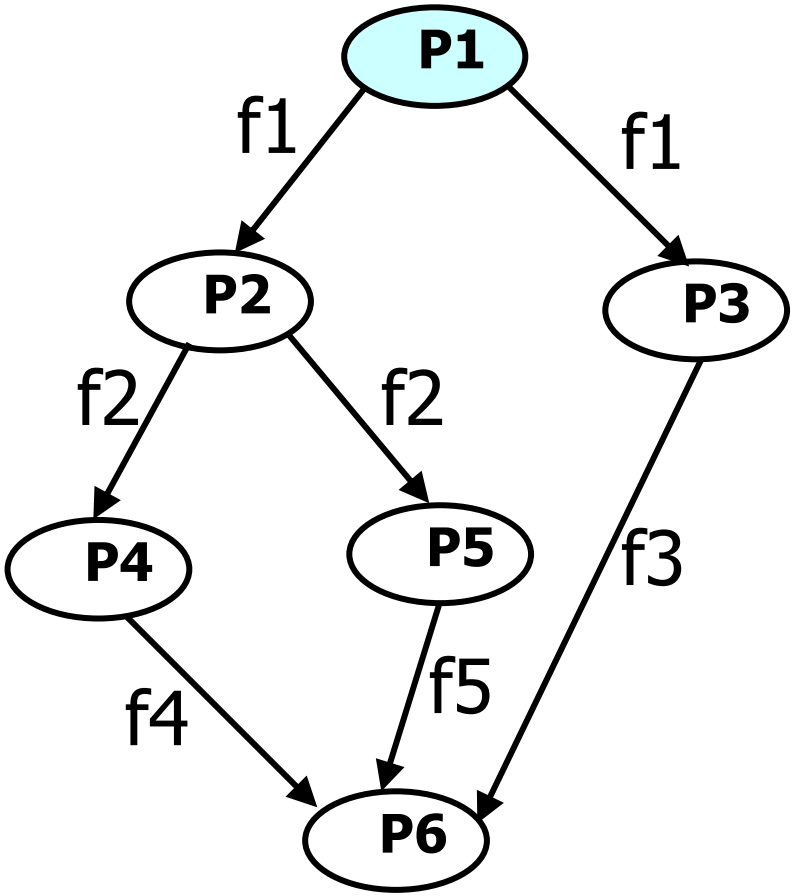


解法2

- 提示：设五个同步信号量f1、f2、f3、f4、f5分别表示进程P1、P2、P3、P4、P5是否执行完成，其初值**均为0**。这六个进程的同步描述如何?

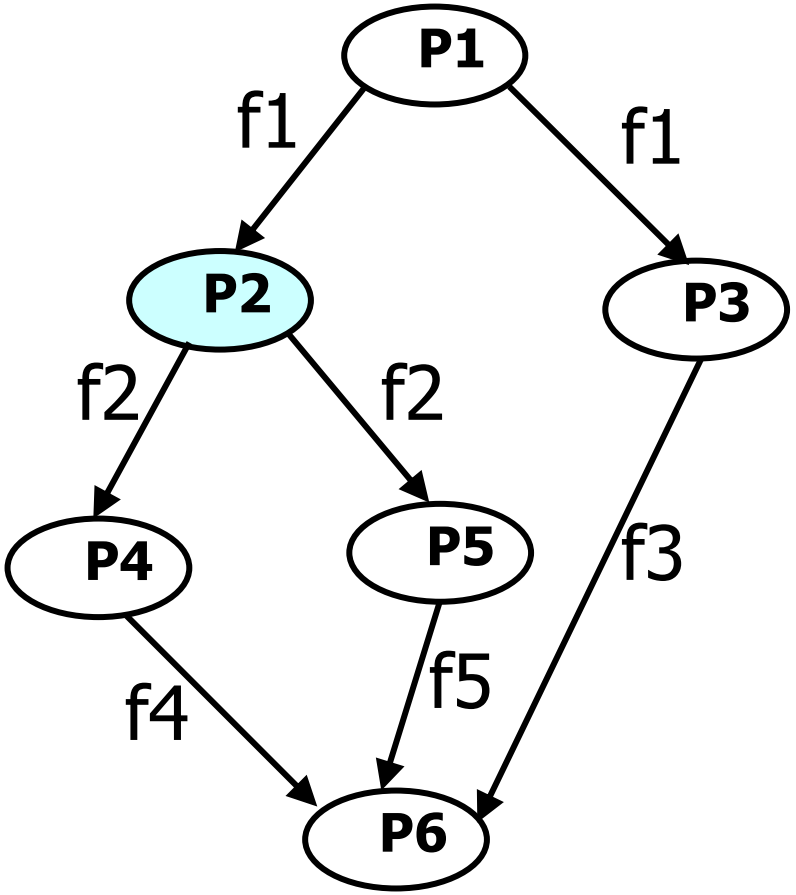
解法2（1）

```
P1 ()
{
    执行P1的代码;
    v(f1);
    v(f1);
 }
```



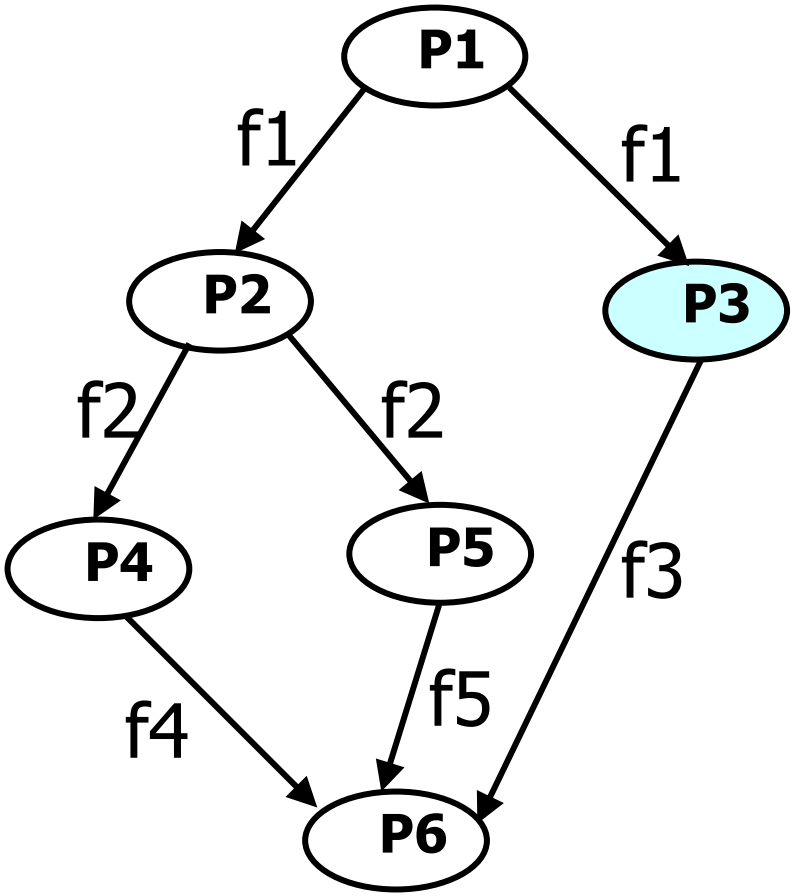
解法2（2）

```
P2 ()
{
     p(f1);
    执行P2的代码;
     v(f2);
     v(f2);
 }
```



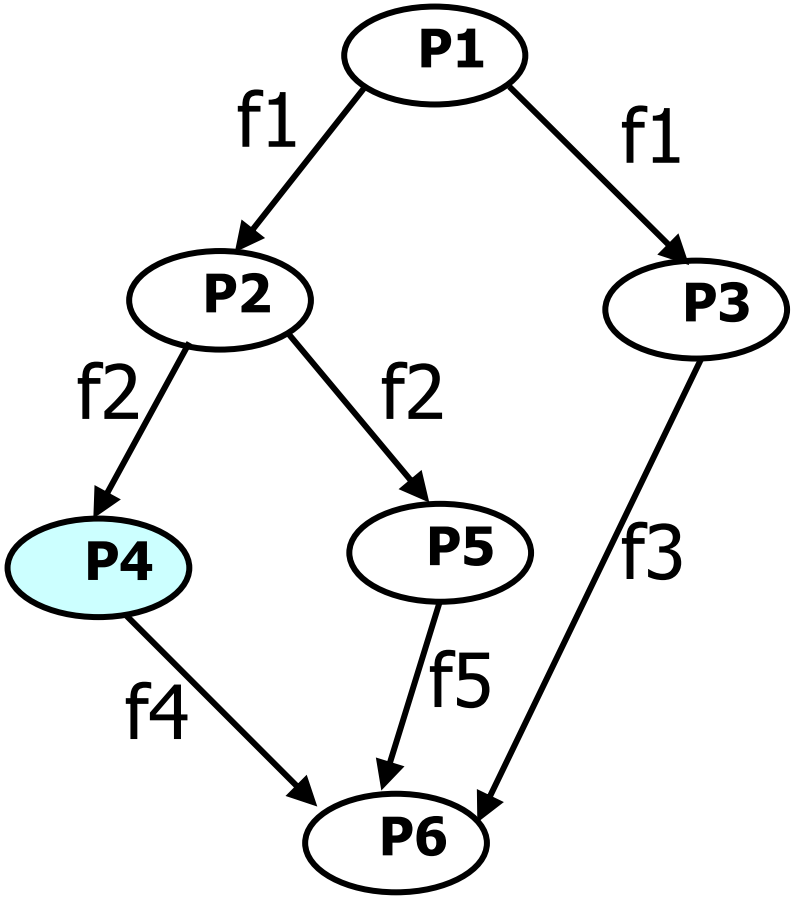
解法2（3）

```
P3 ()
{
    p(f1);
   执行P3的代码;
    v(f3);
 }
```



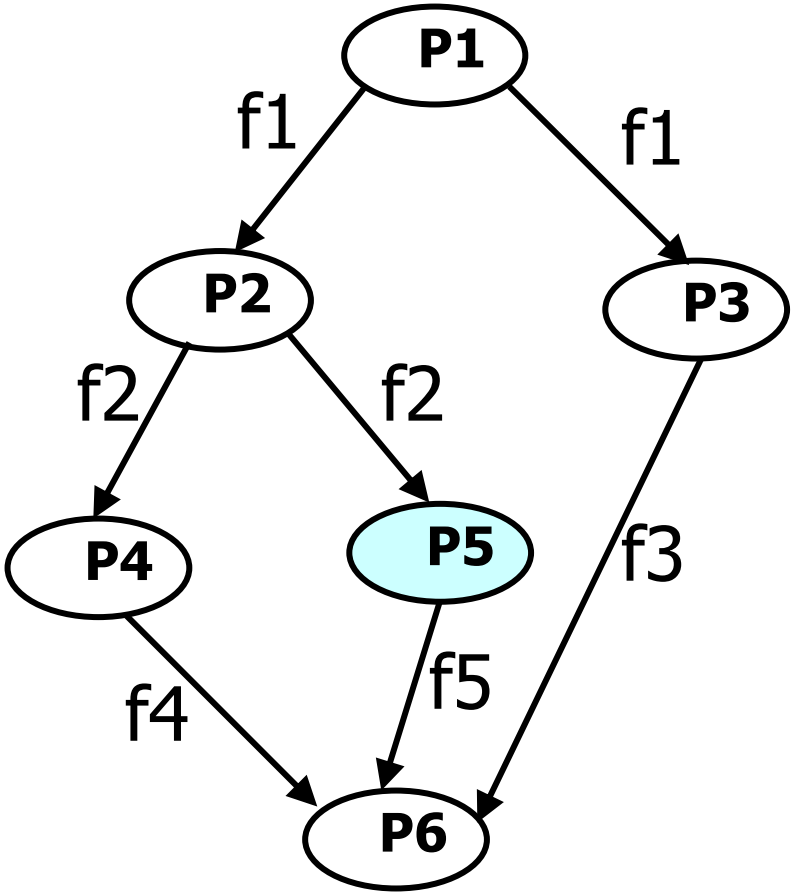
解法2（4）

```
P4 ()
{
    p(f2);
   执行P4的代码;
    v(f4);
}
```



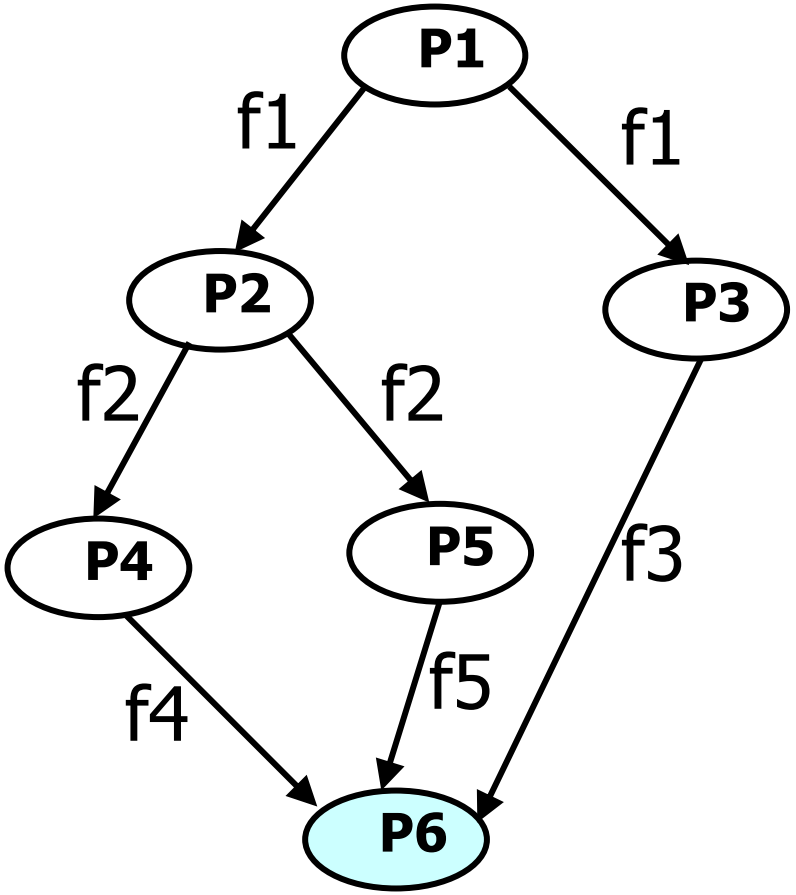
解法2（5）

```
P5 ()
{
     p(f2);
    执行P5的代码;
     v(f5);
}
```



解法2（6）

```
P6 ()
{
    p(f3);
    p(f4);
    p(f5);
    执行P6的代码;
}
```



关于信号量的注意事项

- P和V必须成对使用，且不能乱序，不能遗漏。
  - P和V乱序/用混
    - V(mutex) …. P (mutex)
    - P(mutex) …. P (mutex)
    - V(mutex) … V (mutex)
  - 遗漏 P (mutex) 或者V (mutex) (or both)
  - ……
- 不正确使用信号量可能导致各种问题。

不正确使用信号量可能导致的问题

- 饥饿：无限期地阻塞。进程可能永远无法从它等待的信号量队列中移去

- 死锁：两个或多个进程无限等待一个事件的发生，而该事件正是由其中的一个等待进程引起的

例如：两个进程竞争使用两个不同的临界资源。

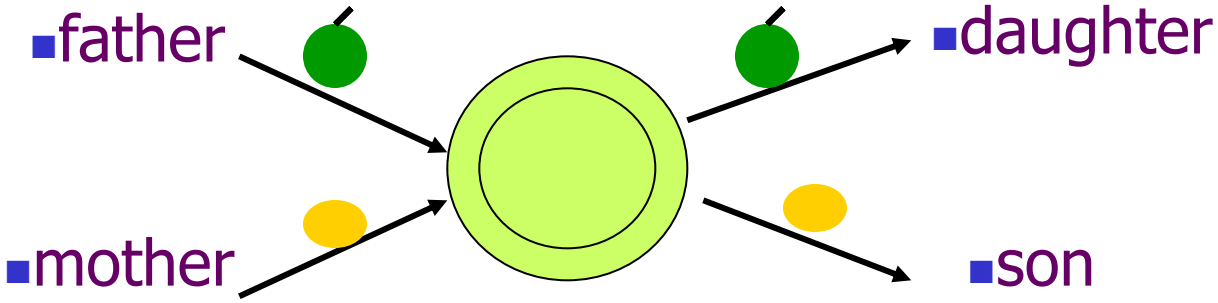
设S和Q是两个初值为1的信号量

P_0:
```
P(S);
P(Q);
 …
V(S);
V(Q)
```

P_1:
```
P(Q);
P(S);
 …
V(Q);
V(S);
```

课堂练习1：苹果桔子问题

- 桌上有一只盘子，每次只能放入一只水果；爸爸专向盘子中放苹果，妈妈专向盘子中放桔子，一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子里的苹果 。



苹果桔子问题(续)

```
void father()
{
    while(true)
    {   削一个苹果;
        P(empty);
        把苹果放入盘子;
        V(apple);
     }
}
void mother()
{
    while(true)
    {   剥一个桔子;
        P(empty);
        把桔子放入盘子;
        V(orange);
    }
}
```

```
void daughter()
{
     while(true)
     {  P(apple);
        从盘子中取苹果;
        V(empty);
        吃苹果;
      }
}
void son()
{
    while(true)
    {   P(orange);
        从盘子中取桔子;
        V(empty);
        吃桔子;
     }
}
```

课堂练习2：司机和售票员问题

- 司机的活动

```
P1：while(true)
    {
      启动车辆;
      正常行车;
      到站停车;
    }
```

- 售票员的活动

```
P2：while(true)
    {
      关门;
       售票;
      开门;
    }
```

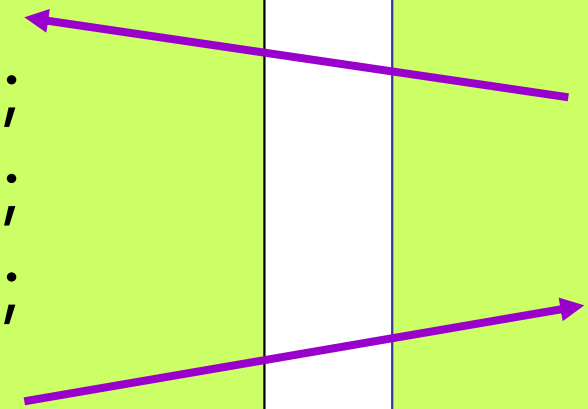
- 设置两个信号量 start和open，初值均为0；

◆ 司机的活动

```
P1：while(true)
    {
      P(start);
      启动车辆;
      正常行车;
      到站停车;
      V(open);
    }
```

◆ 售票员的活动

```
P2：while(true)
    {
      关门;
      V(start);
      售票;
      P(open);
      开门;
    }
```



第6章 进程同步

6.4 经典的进程同步问题

6.4 经典进程同步问题

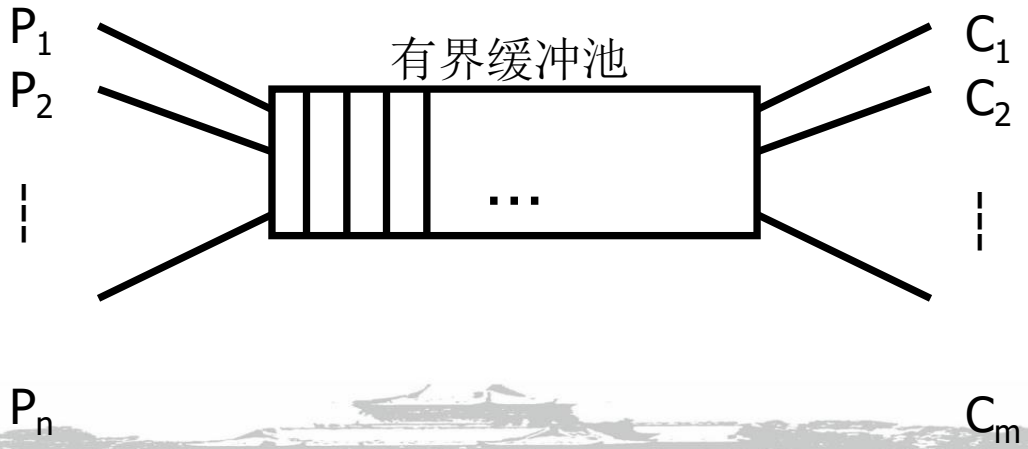
- 多道程序环境中的进程同步是一个非常有趣的问题，吸引了很多学者研究，从而产生了一系列经典进程同步问题。

1. 生产者—消费者问题

- 一个经典的同步问题：生产者与消费者
  - 著名的生产者--消费者问题是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。
  - 在操作系统中，生产者进程可以是计算进程、发送进程；而消费者进程可以是打印进程、接收进程等等。
  - 解决好生产者--消费者问题就解决好了一类并发进程的同步问题。

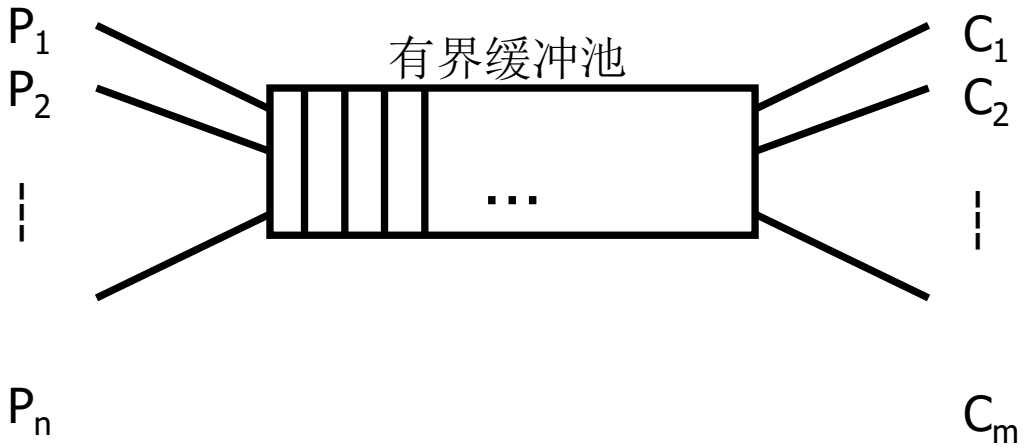
生产者—消费者问题

- 问题描述（有界缓冲区问题）
  - 有n个生产者和m个消费者，连接在一个有k个单位缓冲区的有界缓冲上。
  - 其中，生产者进程P_i和消费者进程C_j都是并发进程。
  - 只要缓冲区未满，生产者P_i生产的产品就可投入缓冲区;
  - 只要缓冲区不空，消费者进程C_j就可从缓冲区取走并消耗产品。



生产者—消费者问题

- 同步关系有：
  - 当缓冲池满时生产者进程需等待；
  - 当缓冲池空时消费者进程需等待；
  - 诸进程应互斥使用缓冲池。



生产者—消费者问题

- 设置两个同步信号量empty、full，其初值分别为n、0。
- 有界缓冲池是一个临界资源，还需要设置一个互斥信号量mutex，其初值为1。

- 生产者—消费者问题的同步描述如下：

算法描述

- 生产者

```
生产一个产品;
  P(empty);
  P(mutex);
将一个产品送入缓冲区;
  V(mutex);
  V(full);
```

- 消费者

```
  P(full);
  P(mutex);
从缓冲区中取一个产品;
  V(mutex);
  V(empty);
消费一个产品;
```

- 注意：无论在生产者进程还是在消费者进程中，P操作的次序都不能颠倒，否则将可能造成死锁。

颠倒生产者进程中的P操作

- 当mutex=1，full=n，empty=0，且调度到生产者执行时

生产一个产品；

P(mutex);

P(empty);

将一个产品送入缓冲区；

V(mutex);

V(full);

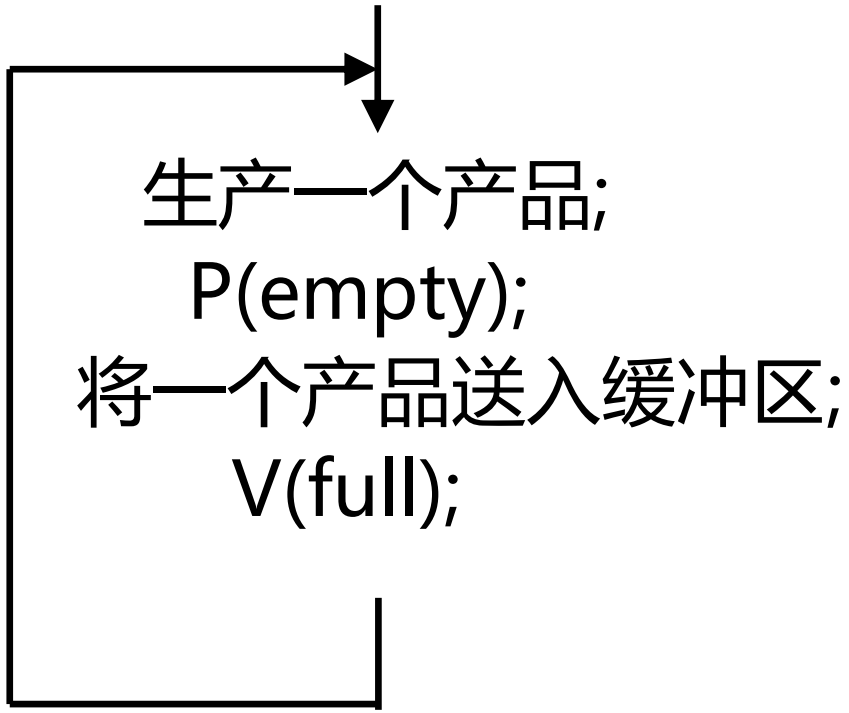
	mutex	full	empty
←	1	n	0
←	0	n	0
←	0	n	-1

- 可见，生产者执行P(empty)后将阻塞，而此时消费者、或其它生产者能否执行呢？

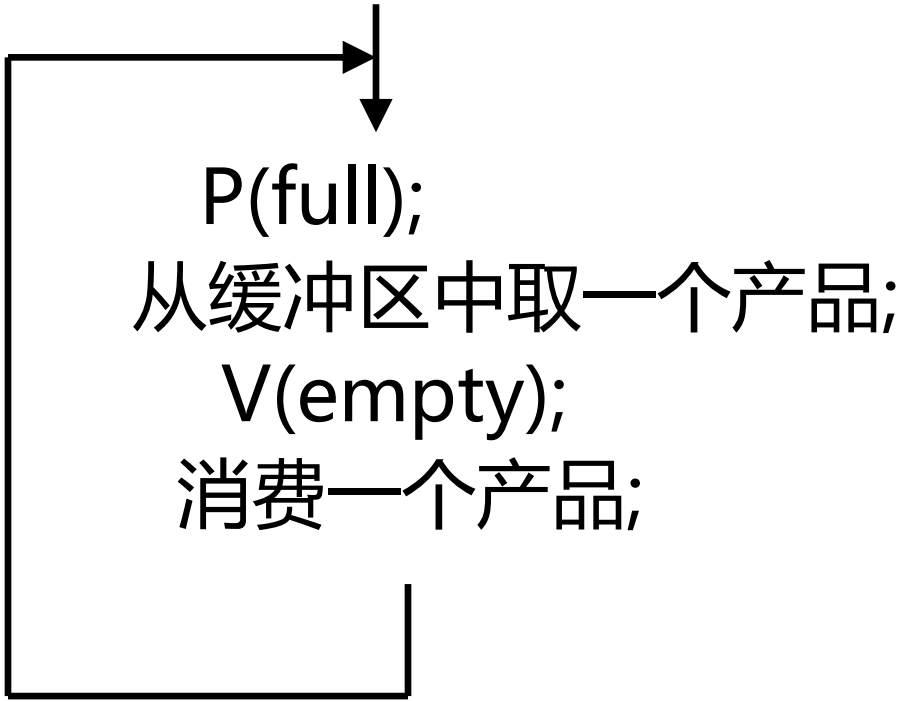
一个简化：只有一个缓冲区

- 设置两个同步信号量empty、full，其初值分别为1、0

- **生产者**



- **消费者**



2.读者—写者问题

- 一个数据对象（如文件或记录）可以被多个并发进程所共享

- 其中有些进程只要求读数据对象的内容——读者
- 而另一些进程则要求修改或写数据对象的内容——写者

- 允许多个读者进程同时读此数据对象

- 但是一个写者进程不能与其他进程（不管是写者进程还是读者进程）同时访问此数据对象

读者—写者问题分类

- 读者优先：
  - 当存在一个读者时候，读者允许进入，写者要等待，而且可能会因为不断有读者到达而长时间等待
- 写者优先：
  - 当写者提出存取共享对象的要求后，已经开始读的进程让其读完，但不允许新读者进入。
  - 写者结束时，判断是否有写者等，有则优先写者
- 读写者公平：
  - 按提出请求的顺序先来先服务
  - 读者前面无写者等待时，可进入
  - 读者前面有写者等待时，等待

用信号量解决读者优先问题

- 为解决读者优先问题，应设置两个信号量和一个共享变量：
  - 互斥信号量rmutex，用于使读进程互斥地访问共享变量readcount，其初值为1；
  - 共享变量readcount，用于记录当前正在读数据集的读进程数目，初值为0。
  - 写互斥信号量wmutex，用于实现写进程与读进程的互斥以及写进程与写进程的互斥，其初值为1；

算法描述

- 读者

```
P(rmutex);
if (readcount==0) P(wmutex);
readcount++;
V(rmutex);
读数据集;
P(rmutex) ;
readcount--;
if (readcount==0) V(wmutex);
V(rmutex);
```

- 写者

```
P(wmutex);
写数据集;
V(wmutex);
```

对读者写者问题的理解

- 请注意对信号量rmutex意义的理解。
- rmutex是一个互斥信号量，用于使读进程互斥地访问共享变量readcount。该信号量并不表示读进程的数目，表示读进程数目的是共享变量readcount。

- 问题首次讨论于：
  - P.J. Courtois, F. Heymans. Concurrent Control with "Readers" and "Writers”. Communications of the ACM. 1971.

写者优先

- 应该满足的要求
  - 多个读者可以同时进行读；
  - 写者必须互斥
    - 只允许一个写者写，不能读者写者同时进行
  - 写者优先于读者
    - 一旦有写者，则后续读者必须等待，在唤醒时优先考虑写者，直到最后一个写者完成
    - 对于已经开始的读者让其读完，对于未开始的读者让其等待
    - 如果一堆读者排在写者前面，应该按照排队顺序工作，or写者能插队吗？

写者优先解法1

- 变量int readcount=0，writecount = 0
- 设置4个信号量: mutexReadCount=1, mutexWriteCount=1, r=1, w=1

```
Reader()
{
    P(r);
    P(mutexReadCount);
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(mutexReadCount);
    V(r);
    reading is performed….
    P(mutexReadCount);
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

```
Writer()
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);
    writing is performed…
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```

这个对吗?

- 变量int readcount=0， writecount = 0
- 设置4个信号量: mutexReadCount=1, mutexWriteCount=1, r=1, w=1

```
Reader()
{
    P(mutexReadCount);
    P(r);                          ← ③ r2 阻塞
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(r);
    V(mutexReadCount);
    reading is performed....       ← ① r1
    P(mutexReadCount);             ← ④ r1 阻塞
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

颠倒了两个P操作的顺序

```
Writer()
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);                          ← ② w1 阻塞
    writing is performed...
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```

写者优先解法2

- 变量int readcount=0， writecount = 0。
- 设置5个信号量: mutexReadCount, mutexWriteCount, mutexPriority, r, w， 初值均为1.

```
Reader()
{
    P(mutexPriority) ;
    P(r);
    P(mutexReadCount);
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(mutexReadCount);
    V(r);
    V(mutexPriority);
    reading is performed….
    P(mutexReadCount);
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

为什么要增加了一个信号量

```
Writer()
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);
    writing is performed…
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```

读写公平的解法

- 变量int readcount=0
- 设置3个信号量: wmutex=1, rmutex=1，w=1

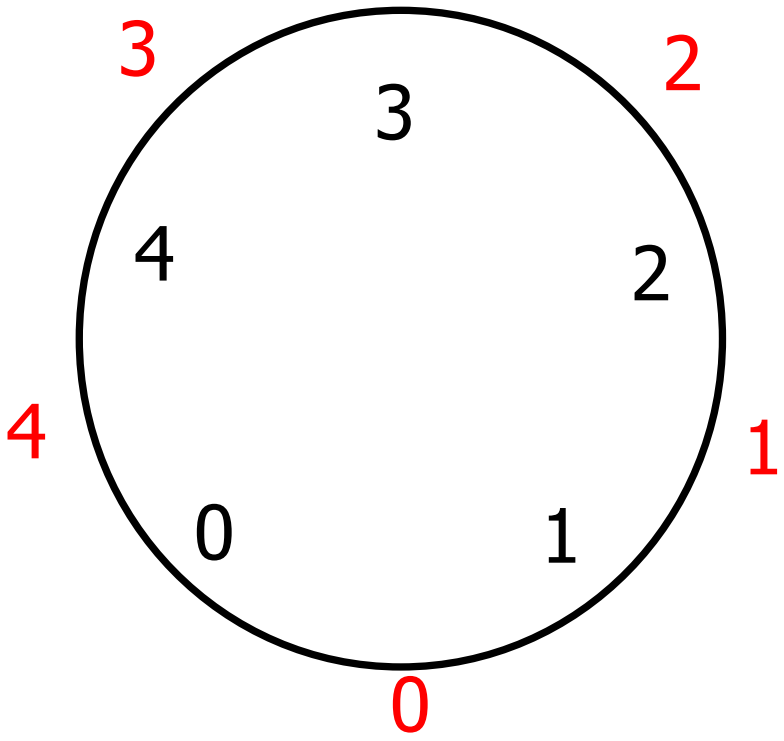
```
Reader()
{
        P(w)
        P(rmutex);
        //若为第一个读者，互斥写者
        if (readcount==0) P(wmutex);
        readcount++;
        V(rmutex);
        V(w)
        读数据集;
        P(rmutex) ;
        readcount--;
        //最后一个读者，则唤醒写者
        if (readcount==0) V(wmutex);
        V(rmutex);
}
```

```
Writer()
{
    P(w)
    P(wmutex);
    写数据集;
    V(wmutex);
    V(w);
}
```

3. 哲学家进餐问题

- 哲学家进餐问题描述：
  - 有五个哲学家，他们的生活方式是交替地进行思考和进餐，
  - 哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有一盆米饭和五支筷子，
  - 平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，
  - 进餐完毕，放下筷子又继续思考。

用信号量解决哲学家进餐问题



- 用五支筷子的信号量构成信号量数组:
  - semaphore chopstick[5] ;
  - 所有信号量初值为1,

某哲学家活动的简单描述

?

```
P(chopstick[i]); //拿左边筷子
P(chopstick[(i+1) %  5]); //拿右边筷子
    进餐;
V(chopstick[i]); //放下左边筷子
V(chopstick[(i+1) % 5]); //放下右边筷子
    思考;
```

算法存在的问题

- 上述算法有可能引起死锁。
  - 当五个哲学家同时感觉饥饿，且同时拿起自己左边的筷子…

算法存在的问题（cont.）

- 对于这样的死锁问题有如下办法解决：
  - 至多允许四个哲学家同时进餐。
  - 仅当左、右两支筷子均可用时，才允许拿起筷子进餐。
  - 奇数号哲学家先拿左边筷子再拿右边筷子，偶数号哲学家相反。
  - 按照一定顺序获取资源，按照相反顺序释放资源。破坏环路 {资源分级法}
    - 取筷子原则：每个哲学家总先拿起左右两边编号较低的筷子，再拿编号较高的。
    - 放筷子规则：用完餐后，他总是先放下编号较高的筷子，再放下编号较低的。
  - ……

4.睡眠的理发师问题

- 理发店里有一位理发师，一把理发椅和N把供等候理发顾客坐的椅子。
- 如果没有顾客，理发师睡眠，当一个顾客到来时叫醒理发师;
- 若理发师正在理发时又有顾客来，那么有空椅子就坐下，否则离开理发店。

用信号量解决睡眠的理发师问题

- 为解决睡眠的理发师问题，应使用三个信号量：
  - customers记录等候理发的顾客数（不包括正在理发的顾客）;初值为0。无顾客理发师在此阻塞
  - barbers记录正在等候理发师的顾客数;初值为0。无理发师顾客在此排队
  - mutex用于互斥访问count。初值为1
- 一个变量：
  - 变量count记录等候的顾客数，它是customers的一份拷贝。之所以使用count是因为无法读取信号量的当前值。

算法描述

```
Baber(){
    while(true){
    //是否有等候的顾客,若无顾客，理发师睡眠
        p(customers);
    //若有顾客，则进入临界区
        p(mutex);
    //等待中的顾客数减1
        count--;
    //理发师发出通知，已经准备好为顾客理发
        v(barbers);
    //退出临界区
        v(mutex);
        cut_hair();
    }
}
```

```
Customer-i (){
    p(mutex);
     //判断是否有空椅子
     if(count <N)
     {
          //等待顾客数加1
           count++;
           //唤醒理发师
           v(customers);
           //退出临界区
           v(mutex);
        //等待理发师准备好，如果理发师忙，
    顾客等待
           p(barbers);
           get_haircut();
    }
    else
    //无空椅子人满了，顾客离开
    v(mutex);
}
```

讨论：

- 1.为什么在baber中，p(mutex)不可以在p(customer)之前出现?
- 2.为什么在baber中， v(barbers) 要放在v(mutex)之前?
- 3.为什么在customer中，v(customers)要放到v(mutex)之前?

讨论1：

为什么在baber中，P(mutex)不可以在p(customer)之前出现？

```
Baber(){
    while(true){
        p(mutex);
        p(customers);   ← ① 阻塞
        count--;
        v(barbers);
        v(mutex);
        cut_hair();
    }
}
```

```
Customer-i (){
    p(mutex);      ← ②      阻塞...
    if(count <N)
    {
        count++;
        v(customers);
        v(mutex);
        p(barbers);
        get_haircut();
    }
    else
        v(mutex);
```

后继的顾客也都在这里阻塞，发生死锁

讨论2：

在baber中，v(babers) 放在v(mutex)之后会怎样?

```
Baber(){
    while(true){
        p(customers);
        p(mutex);
        count--;
        v(mutex);      ← ①
        v(barbers);
        cut_hair();
    }
}
```

```
Customer-i (){
    p(mutex);
     if(count <N)
     {
        count++;
        v(customers);
        v(mutex);
        p(barbers);    ← ②  阻塞...
        get_haircut();
    }
    else
        v(mutex);
```

如果后继很多顾客持续到达，在理发师继续执行前执行到②，都在这里阻塞，发生饥饿

讨论3：

在customer中，v(customers)放在v(mutex)之后会怎样?

```
Baber(){
    while(true){
        p(customers);
        p(mutex);
        count--;
        v(barbers);
        v(mutex);
        cut_hair();
    }
}
```

```
Customer-i (){
    p(mutex);
     if(count <N)
     {
        count++;
        v(mutex);
        v(customers);
        p(barbers);
        get_haircut();
    }
    else
        v(mutex);
```

顾客数超过N个后，直接离开，如果顾客持续达到，干扰了唤醒理发师的工作

利用信号量解决同步问题的思路

- 理清同步与互斥关系
  - 哪些资源及对象需要互斥访问
  - 哪些资源的访问顺序对进程调度有制约关系
  - 同步信号量要表示出资源的等待条件及数目
  - P操作内包含等待；V操作内包含唤醒
  - 依多个访问顺序约束，同类资源可设置多个信号量
    - 生产者消费者问题中的empty和full
  - 一定要注意互斥量与同步信号量的顺序
    - 同步P优先于互斥P
  - 信号量的操作只能为PV，切记不要直接取值和赋值
    - 可设置副本，如理发师问题中的count 和customers