2. 进程同步的概念

(1) 什么是进程同步

并发进程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通消息称为进程同步。

(2) 进程同步的例子

① 病人就诊

看病活动：

⋮

要病人去化验；

⋮

等化验结果；

⋮

继续诊病；

化验活动：

⋮

需要进行化验？

⋮

进行化验；

开出化验结；

⋮

②誊抄

用卡片输入机将一个文本复写到行式打印机。（输入机：1000卡/分，打印机：600行/分）

- **顺序程序实现方案**

```
while (不空) {
    INPUT;
    OUTPUT;
}
```

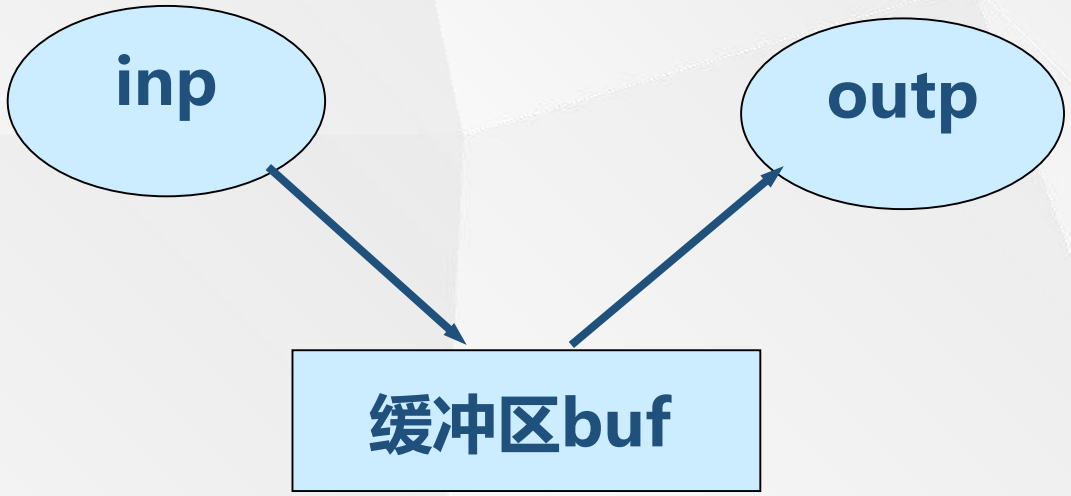
缺点：未能利用卡片输入机与行式打印机的并行操作能力，造成系统效率低。

速度：1/(1/1000+1/600)=375

并发程序实现方案

输入程序inp：
```
while (不空) {
        INPUT;
        写入缓冲区;
}
```

输出程序outp：
```
while (未结束) {
        读缓冲区;
        OUTPUT;
}
```

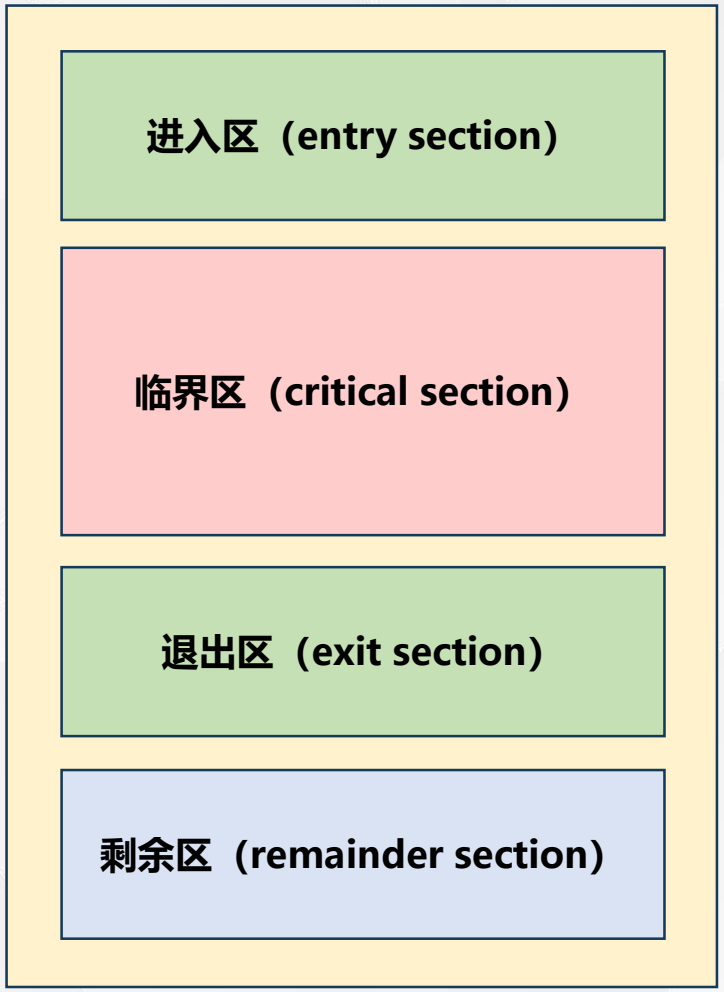


Q：速度？问题？

同步与互斥的关系

- 进程的互斥从本质上来看也是一种特殊的进程同步。
- 但是为便于区别，一般只把有严格的执行顺序要求的进程同步称为同步问题。
- 同步反映的是合作关系；互斥反映的是竞争关系。

如何保证进程互斥地进入临界区



临界区的访问规则：

- **空闲则入**

  没有进程在临界区时，任何进程都可以进入临界区。

- **忙则等待**

  有进程在临界区时，其他进程均不能进入临界区。

- **有限等待**

  等待进入临界区的进程不能无限等待。

- **让权等待（可选）**

  不能进入临界区的进程应释放CPU。

方案一

- **定义共享变量**

  int flag=0; //是否有进程进入临界区

- **进程Pi的代码**

```
do {
        while (flag == 1);
        flag=1;                 //进入区
        critical section        //临界区
        flag=0;                 //退出区
        remainder section  //剩余区
} while(1);
```

- **不满足“忙则等待”**

- **进程Pj的代码**

```
do {
        while (flag == 1);
        flag=1;                 //进入区
        critical section        //临界区
        flag=0;                 //退出区
        remainder section  //剩余区
} while(1);
```

方案二

- **定义共享变量**

  int turn=i;

  //turn==i，表示允许进程i进入临界区

- **进程Pi的代码**

```
do {
        while (turn != i);    //进入区
        critical section      //临界区
        turn=j;               //退出区
        remainder section  //剩余区
} while(1);
```

- **满足“忙则等待”**
- **不能保证“空闲则入”**

- **进程Pj的代码**

```
do {
        while (turn != j);    //进入区
        critical section      //临界区
        turn=i;               //退出区
        remainder section  //剩余区
} while(1);
```

方案三

- **定义共享变量**

```
int flag[2];
flag[0]=flag[1]=0;
//flag[i]==1，表示进程i已进入临界区
```

- **进程Pi的代码**

```
do {
        while (flag[j] == 1);
        flag[i]=1;              //进入区
        critical section        //临界区
        flag[i]=0;              //退出区
        remainder section  //剩余区
} while(1);
```

- **不满足“忙则等待”**

- **进程Pj的代码**

```
do {
        while (flag[i] == 1);
        flag[j]=1;              //进入区
        critical section        //临界区
        flag[j]=0;              //退出区
        remainder section  //剩余区
} while(1);
```

方案四

- **定义共享变量**

```
int flag[2];
flag[0]=flag[1]=0;
//flag[i]==1，表示进程i想进入临界区
```

- **进程Pi的代码**

```
do {
        flag[i]=1;
        while (flag[j] == 1); //进入区
        critical section          //临界区
        flag[i]=0;                //退出区
        remainder section     //剩余区
} while(1);
```

- **满足“忙则等待”**
- **不满足“空闲则入”**

- **进程Pj的代码**

```
do {
        flag[j]=1;
        while (flag[i] == 1); //进入区
        critical section          //临界区
        flag[j]=0;                //退出区
        remainder section     //剩余区
} while(1);
```

方案五—Peterson算法

- **定义共享变量**

```
int turn;          //表示该哪个进程进入临界区
bool flag[2];      //flag[i]==1，表示进程i想进
                   //入临界区
```

- **进程Pi的代码**

```
do {
        flag[i]=true;
        turn=j;
        while(flag[j] && turn==j); //进入区
        critical section                  //临界区
        flag[i]=false;                    //退出区
        remainder section                 //剩余区
} while(1);
```

- **满足“忙则等待”**
- **满足“空闲则入”**
- **方法复杂，需要忙等待**

- **进程Pj的代码**

```
do {
        flag[j]=true;
        turn=i;
        while(flag[i] && turn==i);//进入区
        critical section                  //临界区
        flag[j]=false;                    //退出区
        remainder section                 //剩余区
} while(1);
```

进程同步机构

華中科技大學
计算机科学与技术学院
School of Computer Science & Technology, HUST

什么是锁?

用变量w代表某种资源的状态，w称为“锁” 。

上锁操作和开锁操作

- 检测w的值 (是0还是1);
- 如果w的值为1，继续检测;
- 如果w的值为0，将锁位置1 (表示占用资源)，进入临界区执行。 **(此为上锁操作)**

- 临界资源使用完毕，将锁位置0。 **(此为开锁操作)**

上锁原语和开锁原语

① 上锁算法Lock

输入：锁变量w

输出：无

```
{
    while (w==1);/* 测试锁位的值*/
    w=1;          /*上锁*/
}
```

Q：这个算法有什么缺点?

② 开锁算法Unlock

输入：锁变量w

输出：无

```
{
        w=0；/*开锁*/
}
```

利用等待和唤醒原语实现上锁和解锁

(1) 上锁算法Lock

```
while (w!=0) {
    将当前进程送入等待队列;
    进程状态变为等待态;
    转进程调度;
}
w=1;
```

(2) 开锁算法Unlock

```
if (w等待队列不为空)
    唤醒等待队列中的进程;
w=0;
```

用锁实现进程互斥


```
main()
{
    lock w=0;        /* 互斥锁 */
    cobegin
      pa( );
      pb( );
    coend
}
```

```
pa()
{
    ⋮
    lock(w) ;
    csa ;
    unlock(w)
    ⋮
}
```

```
pb()
{
    ⋮
    lock(w) ;
    csb ;
    unlock(w)
    ⋮
}
```

信号灯

- 1965年由荷兰学者Dijkstra提出的进程同步机制，其基本思想是用一种新的变量类型——信号灯（semaphore）来表示当前资源的可用数量。
- 信号灯是一个确定的二元组（s，q），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。操作系统利用信号灯对并发进程和共享资源进行控制和管理。
  - **变量值s ≥ 0 时，表示绿灯，进程执行；**
  - **变量值s < 0 时，表示红灯，进程停止执行。**

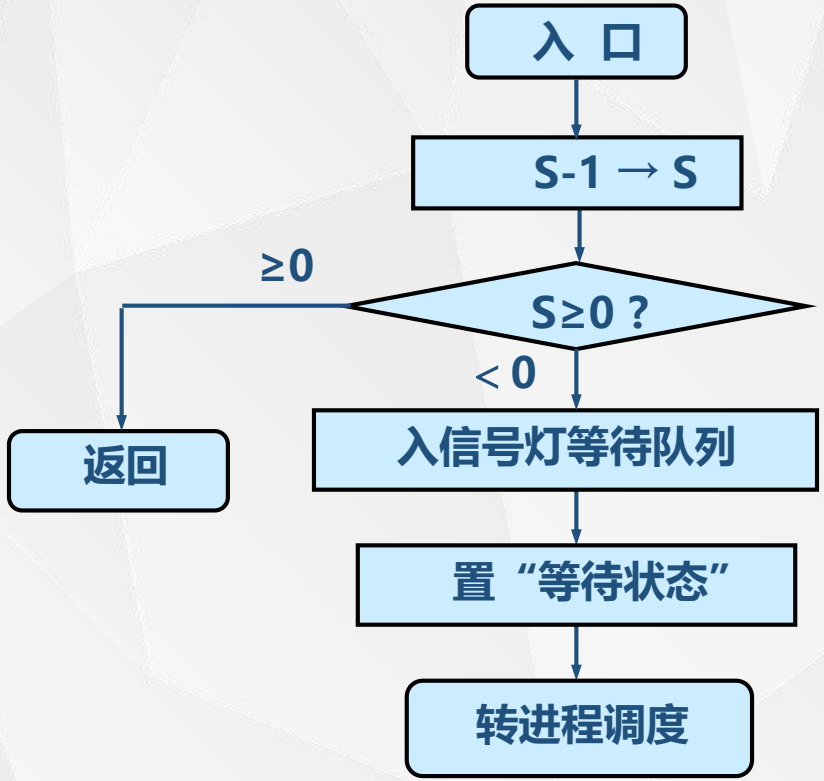
注意：创建信号灯时应说明信号灯的意义和s的初值，且初值绝不能为负值。

- P和V是对信号灯变量可以执行的操作，分别代表荷兰语的test(proberen)和increment(verhogen)。

P操作

对信号灯s的P操作记为**P(s)**。

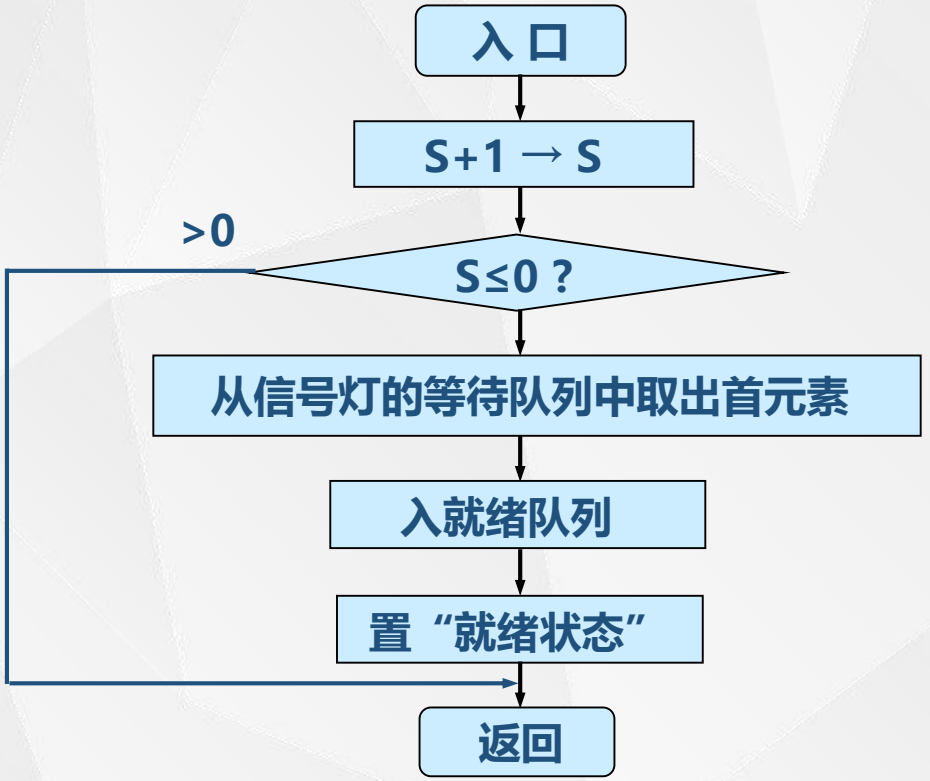
P(s)是一个不可分割的原语操作，即取信号灯值减1，若相减结果为负，则调用P(s)的进程被阻，并插入到该信号灯的等待队列中，否则该进程继续执行。



V操作

对信号灯s的V操作记为**V(s)**。

V(s)是一个不可分割的原语操作，即取信号灯值加1，若相加结果大于零，进程继续执行，否则，唤醒在该信号灯等待队列上的第一个进程。



用信号灯实现进程互斥


```
main( )
{
    semaphore mutex=1;        /* 互斥信号灯 */
    cobegin
        pa( );
        pb( );
    coend
}
pa( )                          pb( )
{                              {
        ⋮                              ⋮
    p(mutex);                      p(mutex);
       csa ;                          csb ;
    v(mutex);                      v(mutex);
        ⋮                              ⋮
}                              }
```

信号灯可能的取值

两个并发进程，互斥信号灯的值仅取1、0和-1三个值。

- 1：表示没有进程进入临界区；
- 0：表示有一个进程进入临界区；
- -1：表示一个进程进入临界区，另一个进程等待进入。

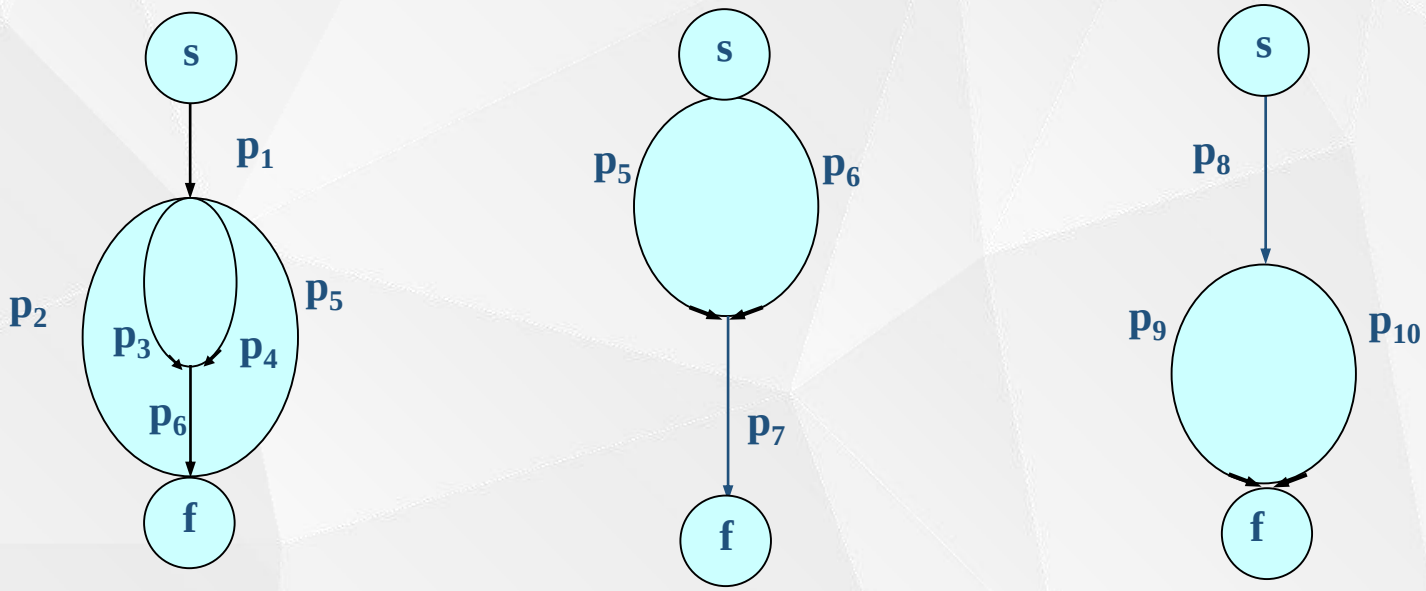
讨论：若有n个并发进程需要互斥

- 互斥信号灯s的取值范围是什么?
- 若s=1，代表什么含义?
- 若s=0，代表什么含义?
- 若s=-k，代表什么含义?

使用信号灯的注意事项

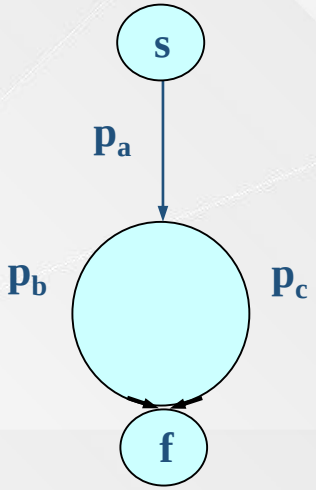
- 信号灯的**初值不能为负值**。
- 必须**成对使用**P和V操作：遗漏P操作则不能保证互斥访问，遗漏V操作则不能在使用临界资源之后将其释放（给其他等待的进程）。
- P、V操作**不能次序错误、重复或遗漏**。

利用信号灯实现同步——合作进程的执行次序



进程流图

例：pa、pb、pc为一组合作进程，其进程流图如图所示，试用信号灯实现这三个进程的同步。



分析任务的同步关系

任务启动后 p_a先执行，当它结束后，p_b、p_c可以开始执行，p_b、p_c 都执行完毕后，任务终止。

信号灯设置

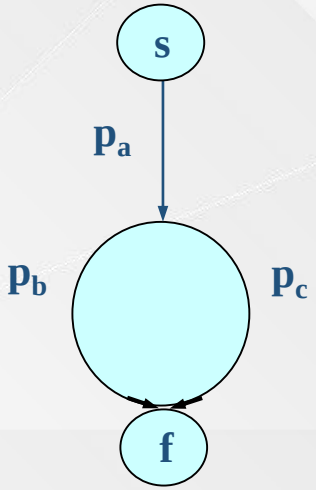
设两个同步信号灯s_b、s_c分别表示进程p_b和p_c能否开始执行，其初值均为0。

同步描述

p_a:
⋮
v(s_b);
v(s_c);

p_b:
p(s_b);
⋮
⋮

p_c:
p(s_c);
⋮
⋮



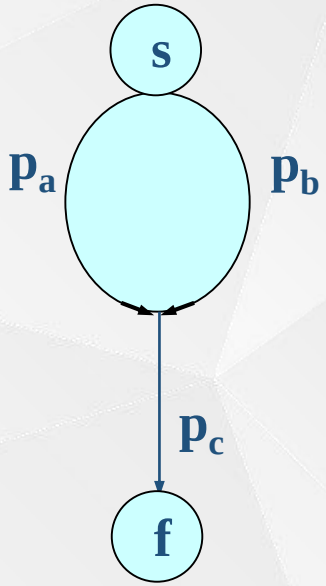
讨论：能否用一个信号灯实现?

```
main( ) {
    semaphore s_b=0;   /*p_b进程能否开始执行*/
    semaphore s_c=0;   /*p_c进程能否开始执行*/
    cobegin
        p_a( );
        p_b( );
        p_c( );
    coend
}
```

```
p_a( ){
    ⋮
  v(s_b);
  v(s_c);
}
```

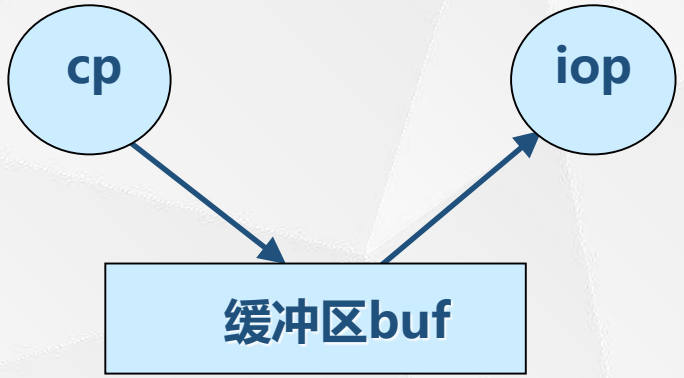
```
p_b( ){
    p(s_b);
    ⋮
    ⋮
}
```

```
p_c( ){
    p(s_c);
    ⋮
    ⋮
}
```


s
p_a
p_b
p_c
f

利用信号灯实现同步——共享缓冲区的合作进程的同步

计算进程 cp和打印进程 iop共用一个缓冲区，为了完成正确的计算与打印，试用信号量的p、v操作实现这两个进程的同步。



两个进程的任务

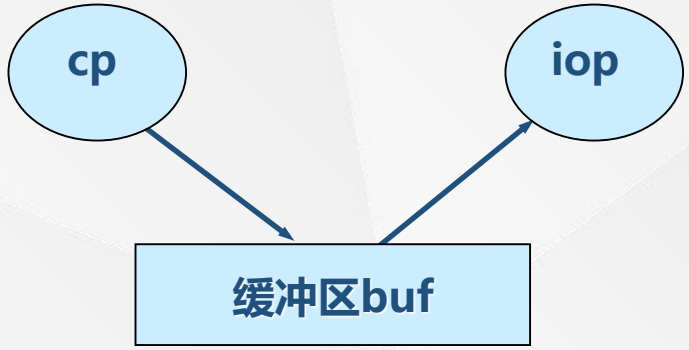
- 计算进程cp经过计算，将计算结果送入buf。
- 打印进程iop把buf中的数据取出，然后打印。

任务的同步关系

- cp进程把计算结果送入buf后，iop进程才能从buf中取出结果去打印，否则必须等待。
- iop进程把buf中的数据取出后，cp进程才能把下一个计算结果数据送入buf中，否则必须等待。

信号量如何设置?

- s_{full}：表示缓冲区是否有可打印的数据，其初值为0。
- s_{empty}：表示缓冲区是否空闲，其初值为1。



信号灯s_{full}和s_{empty}如何使用?

讨论：两个信号量可能的取值及其所表示的状态。

```
main( )
{
      semaphore s_full =0;      /*表示缓冲区中是否有数据*/
      semaphore s_empty =1;     /*表示缓冲区是否为空*/
      cobegin
            cp( ); iop( );
      coend
}
cp( )
{
      while(计算未完成)
      {
            得到一个计算结果;
              p(s_empty);
            将数送到缓冲区中;
              v(s_full);
      }
 }
```

```
iop( )
{
      while(打印工作未完成)
      {
              p(s_full);
              从缓冲区中取一数;
              v(s_empty);
            从打印机上输出;
      }
}
```