



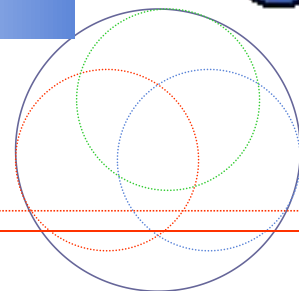
张涛

# Review

进程同步和互斥

临界资源及其访问过程

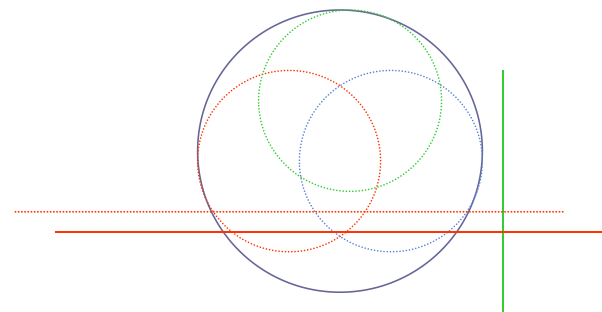
同步机制应遵循的准则



# 本次内容

## 信号量和PV原语操作

## 经典进程同步问题



# 进程同步和互斥间的关系

- **相似处：**进程的互斥实际上是进程同步的一种特殊情况；进程的互斥和同步统称为进程同步。
- **差别：**进程互斥是进程间共享资源的使用权，这种竞争没有固定的必然联系，哪个进程竞争到使用权就归那个进程使用，直到不需要使用时在归还；而进程同步则涉及共享资源的并发进程间有一种必然的联系，当进程必须同步时，即使无进程在使用共享资源时，那么尚未得到同步消息的进程也不能去使用这个资源。

## 3.6.2 信号量(semaphore)和P、V原语

- 信号量机制：由Dijkstra提出的一种解决进程的同步与互斥的工具。
  - 信号量和P、V原语
  - 信号量的使用

## 3.6.2.1 信号量和P、V原语

- 1965年，由荷兰学者Dijkstra提出，是一种卓有成效的进程同步机制。
- 信号量是一个数据结构，它由两个变量构成：整型变量V、指针变量S。
  - 初始化指定一个非负整数值，表示空闲资源总数（又称为“资源信号量”）
  - 若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数
  - 信号量的值只能被P、V操作原语进行改变

# 信号量定义与声明

信号量定义：

```
struct semaphore
{
    int value;
    pointer_PCB queue;
}
```

信号量声明：

```
semaphore s;
```

- 必须置一次且只能置一次初值
- 初值应该大于等于零，不能为负数
- 只能执行P、V操作

# 信号量的物理意义

$S > 0$  表示有  $S$  个资源可用

$S = 0$  表示无资源可用

$S < 0$  则  $|S|$  表示  $S$  等待队列中的进程个数

进程等待队列 `s.queue` 是阻塞在该信号量的各个进程标识

$P(S)$ : 表示申请一个资源

$V(S)$ : 表示释放一个资源



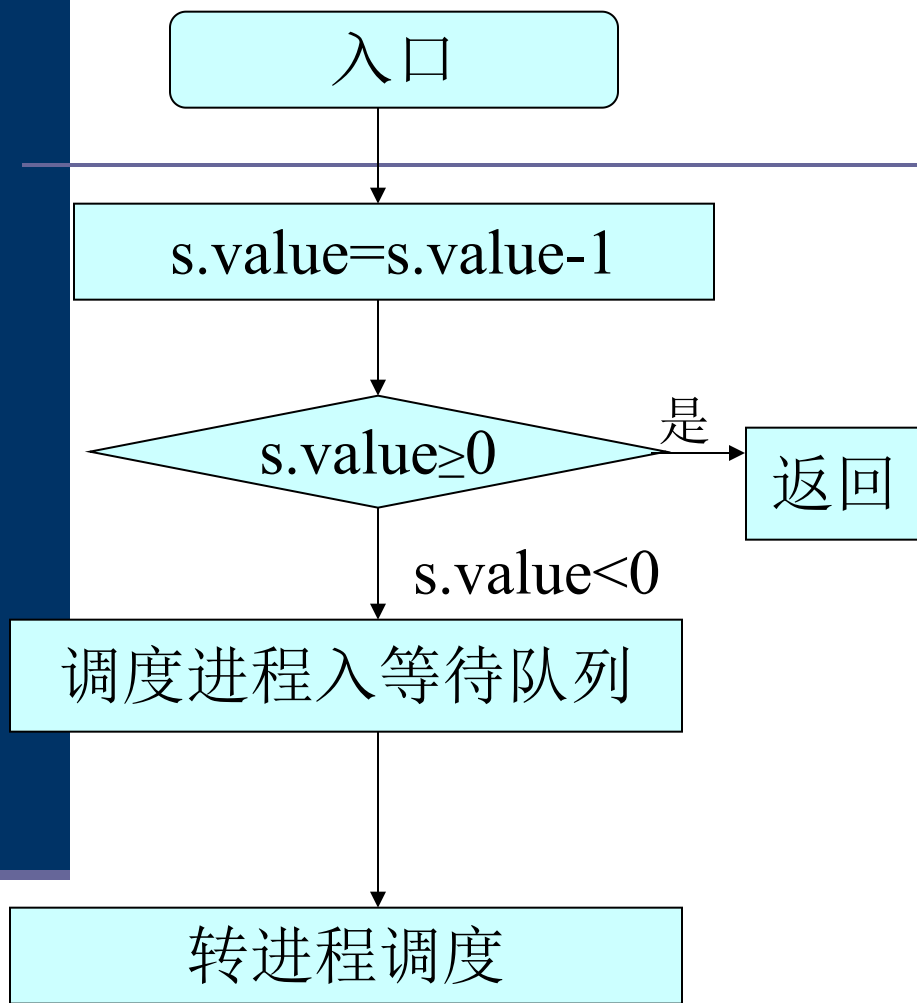
# P原语wait(s)

```
P(s)
{
    s.value = s.value - 1 ; //表示申请一个资源;
    if (s.value < 0)        //表示没有空闲资源;

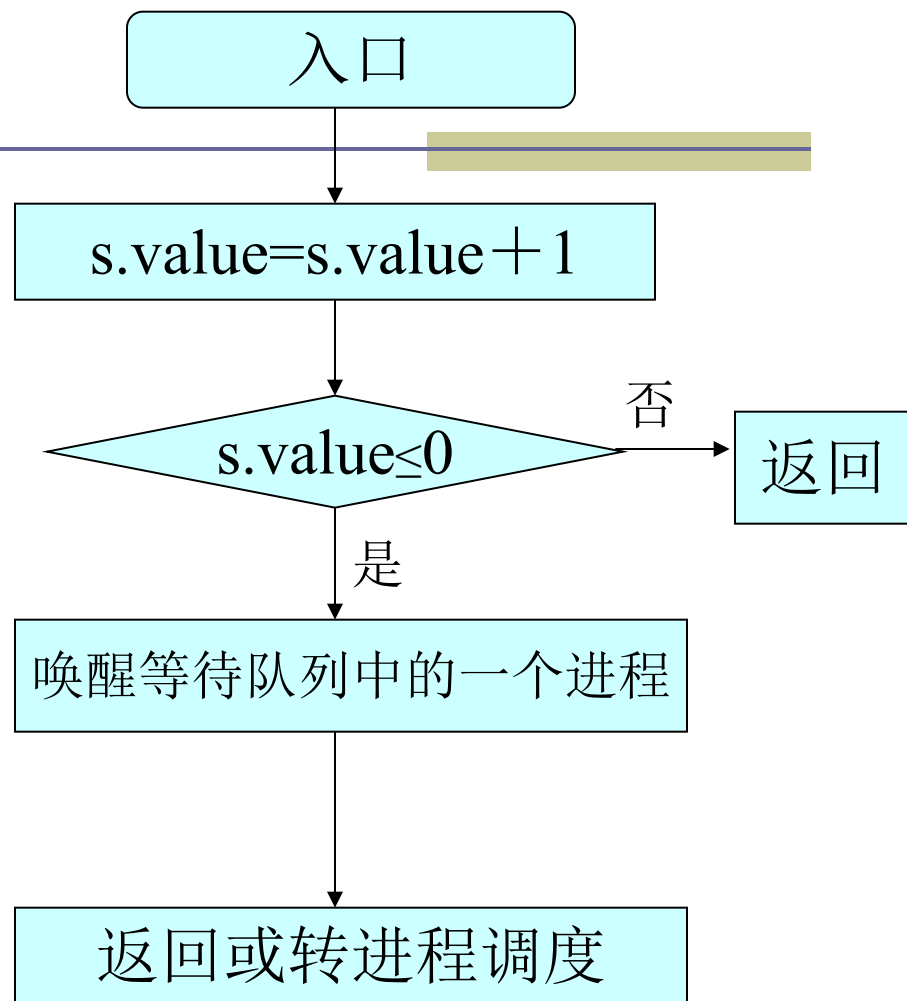
    {
        该进程状态置为等待状态
        将该进程的PCB插入相应的等待队列末尾s.queue;
    }
}
```

# V原语signal(s)

```
V(s)
{
    s.value = s.value + 1;    //表示释放一个资源;
    if (s.value <= 0)        //表示有进程处于阻塞状态;
    {
        唤醒相应等待队列s.queue中等待的一个进程
        改变其状态为就绪态, 并将其插入就绪队列
    }
}
```



P原语操作功能流程图



V原语操作功能流程图

# 对信号量操作的物理意义

- 操作系统对信号量只能通过**初始化**和**两个标准的原语**来访问，对信号量的操作只有**三种**原子操作
  - **初始化**: 通常将信号量的值初始化为非负整数
  - **P操作(wait操作)**
    - 使信号量的值减1(申请一个单位的资源 ( $s.value--$ ))
    - 如果使信号量的值变成负数, 则执行P操作的进程被阻塞(当 $s.value < 0$ 时, 资源已分配完毕, 进程自己阻塞在S的队列上----让权等待)
  - **V操作(signal操作)**
    - 使信号量的值加1(释放一个单位资源 ( $s.value++$ ))
    - 如果信号量的值不是正数, 则使一个因执行V操作被阻塞的进程解除阻塞(若 $s.value \leq 0$ , 则唤醒一个等待进程)

# 信号量及P、V操作讨论

P.V操作必须成对出现，有一个P操作就一定有一个V操作

当为互斥操作时，它们同处于同一进程

当为同步操作时，则不在同一进程中出现

如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要：

一个同步P操作与一个互斥P操作在一起时同步P操作在互斥P操作前

而两个V操作无关紧要

# P、V操作的优缺点

## 优点：

简单，而且表达能力强（用P、V操作可解决任何同步互斥问题）

## 缺点：

不够安全，P、V操作使用不当会出现死锁；  
遇到复杂同步互斥问题时实现复杂

# 利用信号量实现互斥

```
P(mutex);
```

```
critical section
```

```
V(mutex);
```

```
remainder section
```

- 在互斥问题中，对信号量mutex必须设置一次初值，初值必须为1
- 在每个进程中将临界区代码置于P和V原语之间,P、V原语操作应该分别紧靠临界区的头部和尾部，从而提高进程的并发度
- Mutex的取值为： $1, 0, -1, -2, \dots, -(n-1)$
- P、V操作必须成对出现，而且它们同处于同一个进程中

**例 1 :** 设进程A和进程B, 它们都要求进入临界段CS, 下面的设计就可以满足进程的互斥要求:

信号量  $S = 1;$                       \* 定义信号量并确定初值\*/

进程A:

⋮  
P (S) ;  
CS1;  
V (S) ;  
⋮

进程B:

⋮  
P (S) ;  
CS2;  
V (S) ;  
⋮



**例2:** 设有M个进程都要以独享的方式用到某一种资源, 且一次只申请一个资源, 该种资源的数目为N。 实现方法如下:

信号量       $S = N;$

进程  $P_i$ :

$\vdots$

$P(S);$

$CS_i;$

$V(S);$

$\vdots$

# 利用信号量实现同步

**例3:** 设有进程A和B, 要求进程A的输出结果成为进程B的输入信号, 也就是说进程B必须在进程A执行完毕后才能执行。实现方法如下:

信号量  $S = 0$ ;

进程A:  
:  
V (S) ;

进程B:  
:  
P (S) ;  
:



**例4:** 设有进程A、B、C, 要求进程A、C先于进程B运行  
(见图)。实现方法如下:

信号量       $S1 = 0, S2 = 0;$

进程A:      进程C:

⋮

⋮

V (S1) ;      V (S2) ;

⋮

⋮

进程B:      ⋮

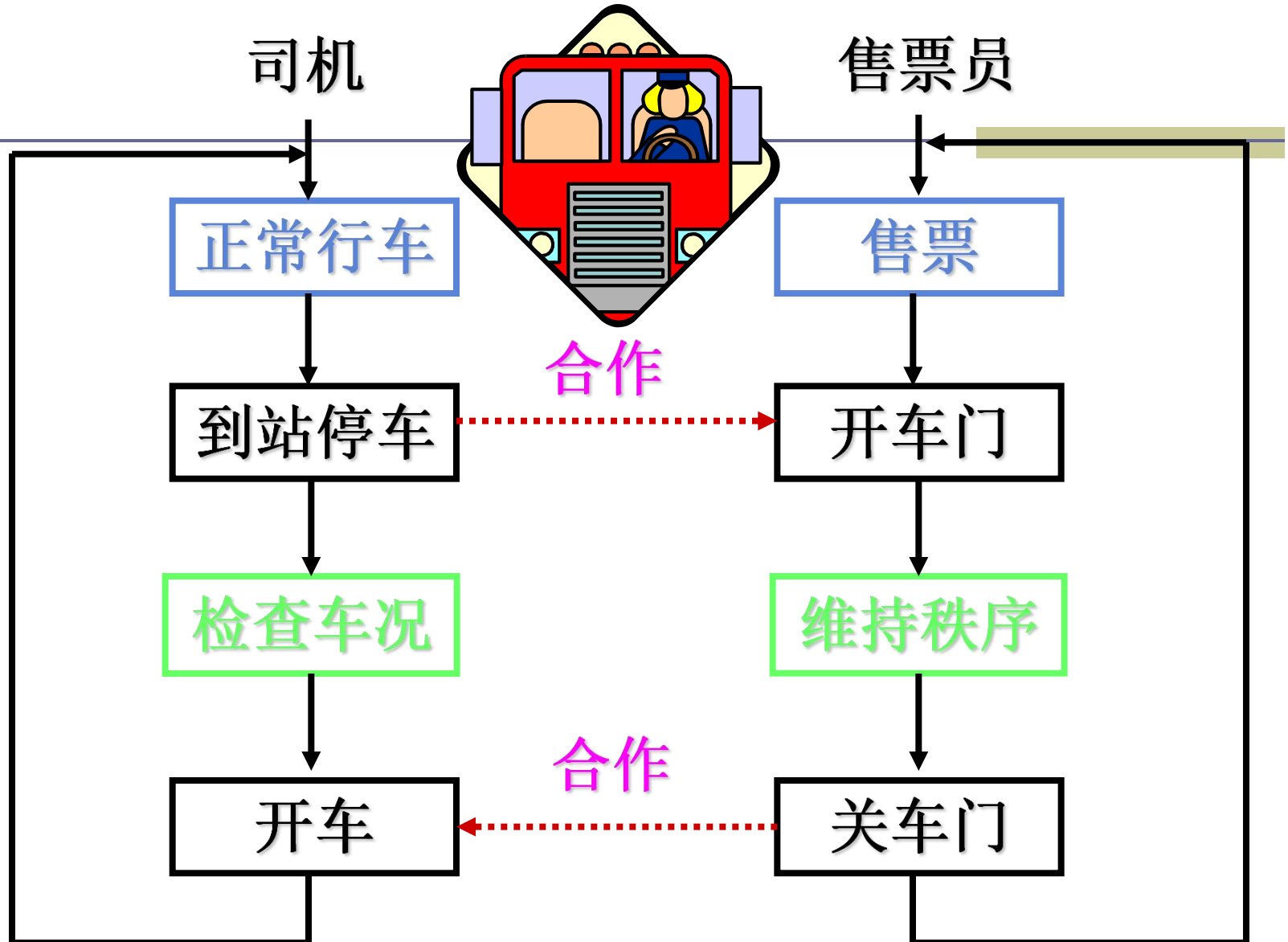
P (S1) ;

P (S2) ;

⋮



# 用P.V操作解决司机与售票员的问题



# 解

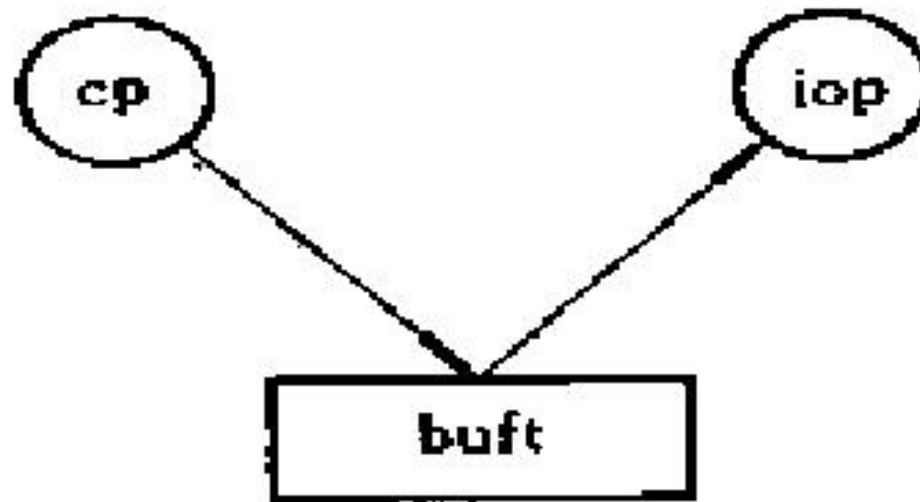
- 设有两个信号量S1, S2, 初值均为0。

```
司机进程:  
while(1)  
{  
    P(S1)  
    启动车辆  
  
    正常驾驶  
  
    到站停车  
    V(S2)  
} ...
```

```
售票员进程:  
while(1)  
{  
    关门  
    V(S1)  
  
    售票  
  
    P(S2)  
    开门  
} ...
```

# 共享缓冲区的进程的同步

- 设某计算进程 C P 和打印进程 I O P 共用一个单缓冲区，C P 进程负责不断地计算数据并送入缓冲区 T 中，I O P 进程负责不断地从缓冲区 T 中取出数据去打印。



## 解

- 为此设有两个信号量  $Sa=0$ ,  $Sb=1$ ,  $Sa$  表示缓冲区中是否有数据,  $Sb$  表示缓冲区中是否有空位置。

```
cp ( )
```

```
{ while (计算未完成)
{
```

```
    得到一个计算结果;
```

```
    p (Sb);
```

```
    将数送到缓冲区中;
```

```
    v (Sa);
```

```
}
```

```
}
```

```
iop ( )
```

```
{ while (打印工作未完成)
{
```

```
    p (Sa);
```

```
    从缓冲区中取一数;
```

```
    v (Sb);
```

```
    从打印机上输出;
```

```
}
```

```
}
```

## 【思考题】

- 桌上有一空盘，最多允许存放一只水果。爸爸可向盘中放一个苹果或放一个桔子，儿子专等吃盘中的桔子，女儿专等吃苹果。

试用P、V操作实现爸爸、儿子、女儿三个并发进程的同步。

提示：设置一个信号量表示可否向盘中放水果，一个信号量表示可否取桔子，一个信号量表示可否取苹果。



# 解

设置三个信号量S,So,Sa , 初值分别为1, 0, 0。分别表示可否向盘中放水果, 可否取桔子, 可否取苹果。

Father()

```
{ while(1)
```

```
{ p(S);
```

将水果放入盘中;

```
if(是桔子)v(So);
```

```
else v(Sa);
```

```
}
```

```
}
```

Son()

```
{ while(1)
```

```
{ p(So)
```

取桔子

```
v(S);
```

吃桔子;

```
}
```

```
}
```

Daughter()

```
{ while(1)
```

```
{ p(Sa)
```

取苹果

```
v(S);
```

吃苹果;

```
}
```

```
}
```

## 3.6.3 经典进程同步问题

- 生产者－消费者问题

the Producer-Consumer Problem

- 读者/写者问题

Readers and Writers Problem

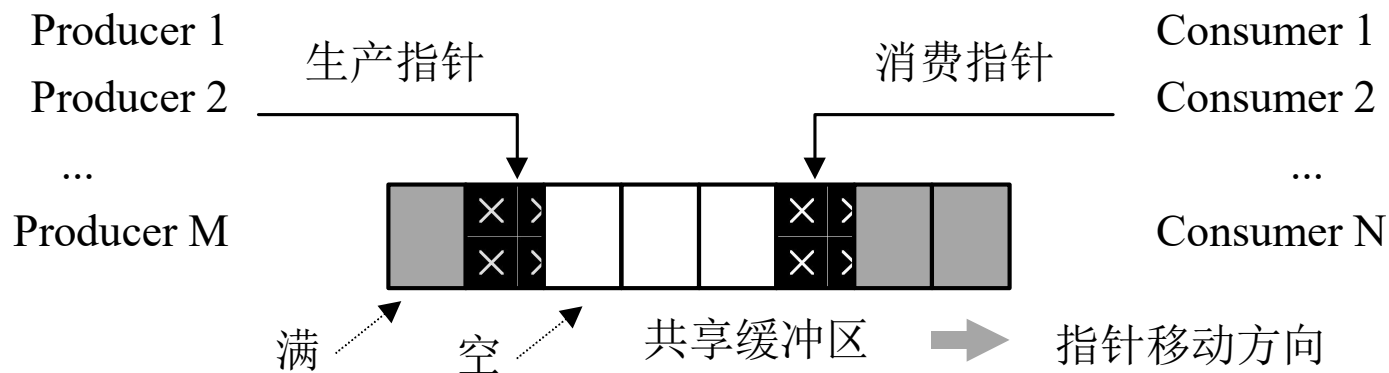
- 哲学家进餐问题

the Dining Philosophers Problem

## 3.6.3.1 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；
- “消费者”进程不断读出；
- 共享缓冲区共有N个；
- 任何时刻只能有一个进程可对共享缓冲区进行操作。



# 问题分析

- 为解决生产者消费者问题，应该设两个同步信号量，一个说明**空缓冲区的数目**，用**S1**表示，初值为有界缓冲区的大小N，另一个说明**已用缓冲区的数目**，用**S2**表示，初值为0。
- 由于在此问题中有M个生产者和N个消费者，它们在执行生产活动和消费活动中要对有界缓冲区进行操作。由于有界缓冲区是一个**临界资源**，必须互斥使用，所以，另外还需要设置一个**互斥信号量mutex**，其初值为1。

Producer

P(empty);
P(mutex);    //进入区
one unit --> buffer;
V(mutex);
V(full);      //退出区

Consumer

P(full);
P(mutex);    //进入区
one unit <-- buffer;
V(mutex);
V(empty);    //退出区

◆ 问题的解：设信号量

◆ full是“满”数目，初值为0，

◆ empty是“空”数目，初值为N。实际上，full和empty是同一个含义： $full + empty == N$

◆ mutex用于访问缓冲区时的互斥，初值是1

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

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

# 分析P操作的顺序很重要

假定执行顺序如下

Producer:

P(empty);

P(mutex);

one unit-->buf;

V(mutex);

V(full);

Consumer:

P(mutex);

P(full); //进入区

one unit<--buf;

V(mutex);

V(empty); //退出区


分析：当full=0, mutex = 1时，执行顺序：

**Consumer.P(mutex) ; Consumer.P(full);**

// C阻塞，等待Producer 发出的full信号

**Producer.P(empty) ; Producer.P(mutex) ;**

// P阻塞，等待Consumer发出的empty信号

 发生死锁

## 【思考题】

- 有一个仓库，可以存放A和B两种产品，要求：
  - (1) 每次只能存入一种产品 (A或B)
  - (2)  $-N < A\text{产品数量} - B\text{产品数量} < M$ 。
- 其中，N和M是正整数。试用P、V操作描述产品A与B的入库过程。
- **提示：**设两个同步信号量Sa、Sb
  - Sa表示允许A产品比B产品多入库的数量
  - Sb表示允许B产品比A产品多入库的数量

## Answer:

设两个信号量Sa、Sb，初值分别为M-1，N-1

Sa表示允许A产品比B产品多入库的数量

Sb表示允许B产品比A产品多入库的数量

设互斥信号量mutex，初值为1。

A产品入库进程：

```
i = 0;
while (1)
{
    生产产品;
    P(Sa);
    P(mutex);
    A产品入库
    V(mutex);
    V(Sb);
};
```

B产品入库进程：

```
j = 0;
while (1)
{
    P(Sb);
    P(mutex);
    B产品入库
    V(mutex);
    V(Sa);
    消费产品;
};
```



## 3.6.3.2 读者/写者问题

- **问题描述：**对共享资源的读写操作，有两组并发进程——读者和写者，共享一组数据区
- **要求：**
  - 允许多个读者同时执行读操作
  - 不允许读者、写者同时操作
  - 不允许多个写者同时操作

“读-写”互斥，“写-写”互斥，“读-读”允许

# 问题分析：第一类，读者优先

如果读者来：

- 1) 无读者、写者，新读者可以读
- 2) 有写者等，但有其它读者正在读，新读者也可以读
- 3) 有写者写，新读者等

如果写者来：

- 1) 无读者，新写者可以写
- 2) 有读者，新写者等待
- 3) 有其它写者，新写者等待

# 第一类读者写者问题的解法

- 设有两个信号量 $w=1$ ,  $\text{mutex}=1$
- 另设一个全局变量 $\text{readcount} = 0$
- $w$ 用于读者和写者、写者和写者之间的互斥
- $\text{readcount}$ 表示正在读的读者数目
- $\text{mutex}$ 用于对 $\text{readcount}$  这个临界资源的互斥访问

读者：

while (1)

{

P(mutex);

readcount ++;

if (readcount==1) P (w);

V(mutex);

读

P(mutex);

readcount --;

if (readcount==0) V(w);

V(mutex);

};

写者：

while (1)

{

P(w);

写

V(w);

};

## 【思考题】写优先

- 修改以上读者写者问题的算法，使之对写者优先，即一旦有写者到达，后续的读者必须等待，无论是否有读者在读。
- **提示：**增加一个信号量，用于在写者到达后封锁后续的读者

解：增加一个信号量S，初值为1

读者：

```
while (1)
{
    P(s);
    P(mutex);
    readcount ++;
    if (readcount==1) P (w);
    V(mutex);
    V(s);
    读
    P(mutex);
    readcount --;
    if (readcount==0) V(w);
    V(mutex);
};
```

OS Lec9

写者：

```
while (1)
{
    P(s);
    P(w);
    写
    V(w);
    V(s);
};
```

# 读者-写者问题(Readers and Writers Problem)

```
typedef int semaphore;  
semaphore mutex = 1;  
semaphore db = 1;  
int rc = 0;
```

```
void reader(void)  
{
```

```
    while (TRUE) {  
        down(&mutex);  
        rc = rc + 1;  
        if (rc == 1) down(&db);  
        up(&mutex);  
        read_data_base();  
        down(&mutex);  
        rc = rc - 1;  
        if (rc == 0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }
```

```
}
```

```
void writer(void)  
{
```

```
    while (TRUE) {  
        think_up_data();  
        down(&db);  
        write_data_base();  
        up(&db);  
    }
```

```
}
```

```
/* use your imagination */  
/* controls access to 'rc' */  
/* controls access to the database */  
/* # of processes reading or wanting to */
```

```
/* repeat forever */  
/* get exclusive access to 'rc' */  
/* one reader more now */  
/* if this is the first reader ... */  
/* release exclusive access to 'rc' */  
/* access the data */  
/* get exclusive access to 'rc' */  
/* one reader fewer now */  
/* if this is the last reader ... */  
/* release exclusive access to 'rc' */  
/* noncritical region */
```

```
/* repeat forever */  
/* noncritical region */  
/* get exclusive access */  
/* update the data */  
/* release exclusive access */
```

# 3.6.3.3 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。
- **问题：**如何保证哲学家们的动作有序进行？如：
  - 不出现相邻者同时要求进餐；
  - 不出现有人永远拿不到筷子；





# 解

设fork[5]为5个信号量，初值为均1

Philosopheri:

while (1)

{

思考;

P(fork[i]);

P(fork[(i+1) % 5]);

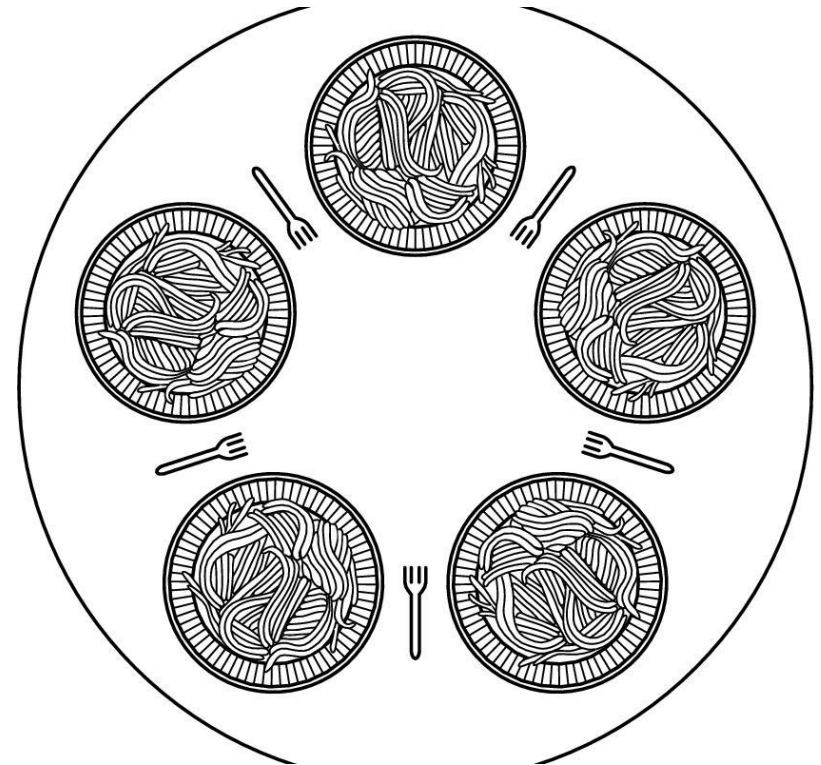
进食;

V(fork[i]);

V(fork[(i+1) % 5]);

}

OSLec9



# 分析

以上解法会出现死锁。为防止死锁发生可采取的措施：

- 最多允许4个哲学家同时坐在桌子周围
- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子  
(√)
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之



```
#define N 5
```

```
void philosopher(int i)  
{
```

```
    while (TRUE) {
```

```
        think( );
```

```
        take_fork(i);
```

```
        take_fork((i+1) % N);
```

```
        eat( );
```

```
        put_fork(i);
```

```
        put_fork((i+1) % N);
```

```
    }
```

```
}
```

```
/* number of philosophers */
```

```
/* i: philosopher number, from 0 to 4 */
```

```
/* philosopher is thinking */
```

```
/* take left fork */
```

```
/* take right fork; % is modulo operator */
```

```
/* yum-yum, spaghetti */
```

```
/* put left fork back on the table */
```

```
/* put right fork back on the table */
```

## A nonsolution to the dining philosophers problem

```

#define N          5                /* number of philosophers */
#define LEFT      (i+N-1)%N        /* number of i's left neighbor */
#define RIGHT     (i+1)%N          /* number of i's right neighbor */
#define THINKING  0                /* philosopher is thinking */
#define HUNGRY    1                /* philosopher is trying to get forks */
#define EATING    2                /* philosopher is eating */
typedef int semaphore;             /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}

```

## Solution to dining philosophers problem (part 1)

```

void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                       /* enter critical region */
    state[i] = HUNGRY;                                  /* record fact that philosopher i is hungry */
    test(i);                                           /* try to acquire 2 forks */
    up(&mutex);                                         /* exit critical region */
    down(&s[i]);                                       /* block if forks were not acquired */
}

void put_forks(i)                                     /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                       /* enter critical region */
    state[i] = THINKING;                               /* philosopher has finished eating */
    test(LEFT);                                       /* see if left neighbor can now eat */
    test(RIGHT);                                      /* see if right neighbor can now eat */
    up(&mutex);                                         /* exit critical region */
}

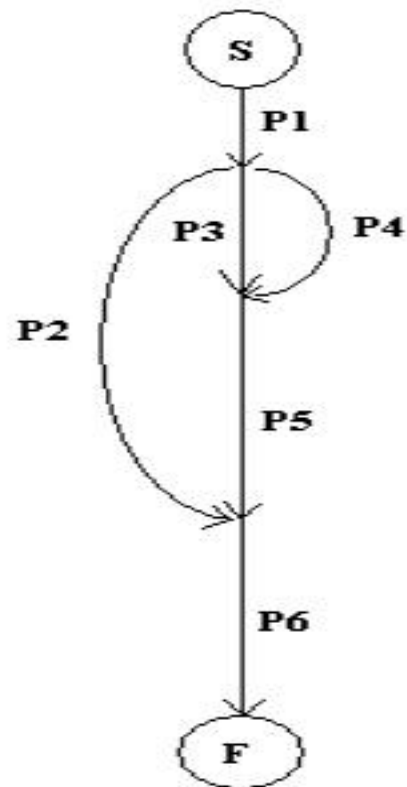
void test(i)                                          /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

## Solution to dining philosophers problem (part 2)

## 【思考题】

如图，试用信号量实现这6个进程的同步



# What you need to do?

---

- 复习课本3.6节的内容
- 课后作业：习题10、13、14、23、25、26、32

See you next time!