# 第三章 同步与通信

3.1 进程的同步与互斥

3.2 经典进程同步问题

3.3 管程

3.4 进程通信

# 3.1 进程的同步与互斥

## *并发进程间的约束关系*

- 同步关系

- **进程——进程**
- **时间次序上受到某种限制**
- **相互清楚对方的存在及其作用，交换信息**
- **往往指有几个进程共同完成一个任务**

进程之间通过在执行时序上的某种限制而达到相互合作的这种约束关系称为进程的同步—直接相互制约关系

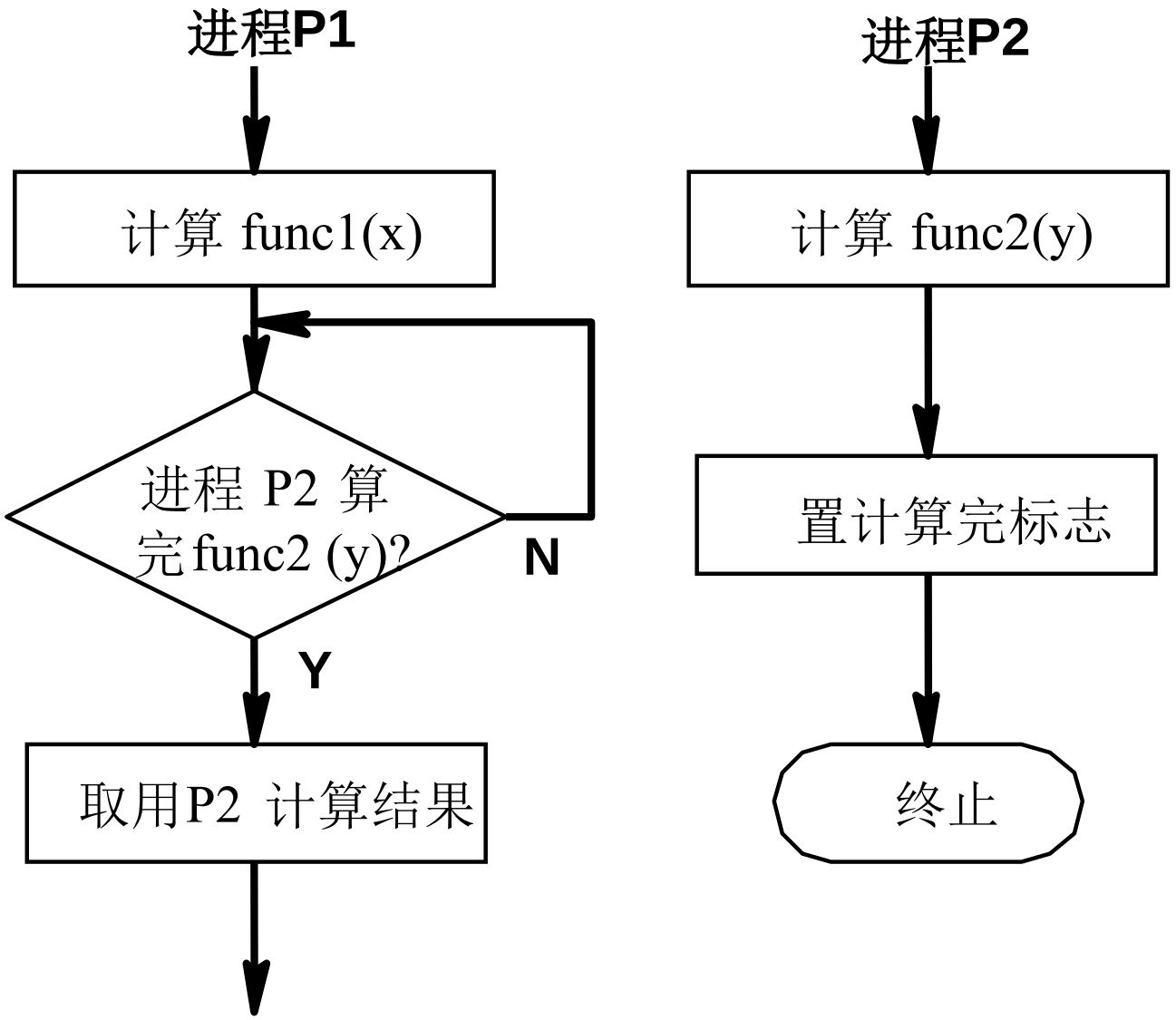
## ●同步问题举例：


A进程只有当缓冲区为空时，才能将数据输入缓冲区，B进程只有当缓冲区有数据时，才能从缓冲区取数进行计算。

又如，有用户作业程序，其形式是：

**Z=func1(x)*func2(y)**

其中**func1(x)**，**func2(y)**均是一个复杂函数，为了加快本题的计算速度，可用两个进程**P1**、**P2**各计算一个函数。进程**P2**计算**func2(y)**，进程P1在计算完**func1(x)**之后，与进程**P2**的计算结果相乘，以获得最终结果**Z**。

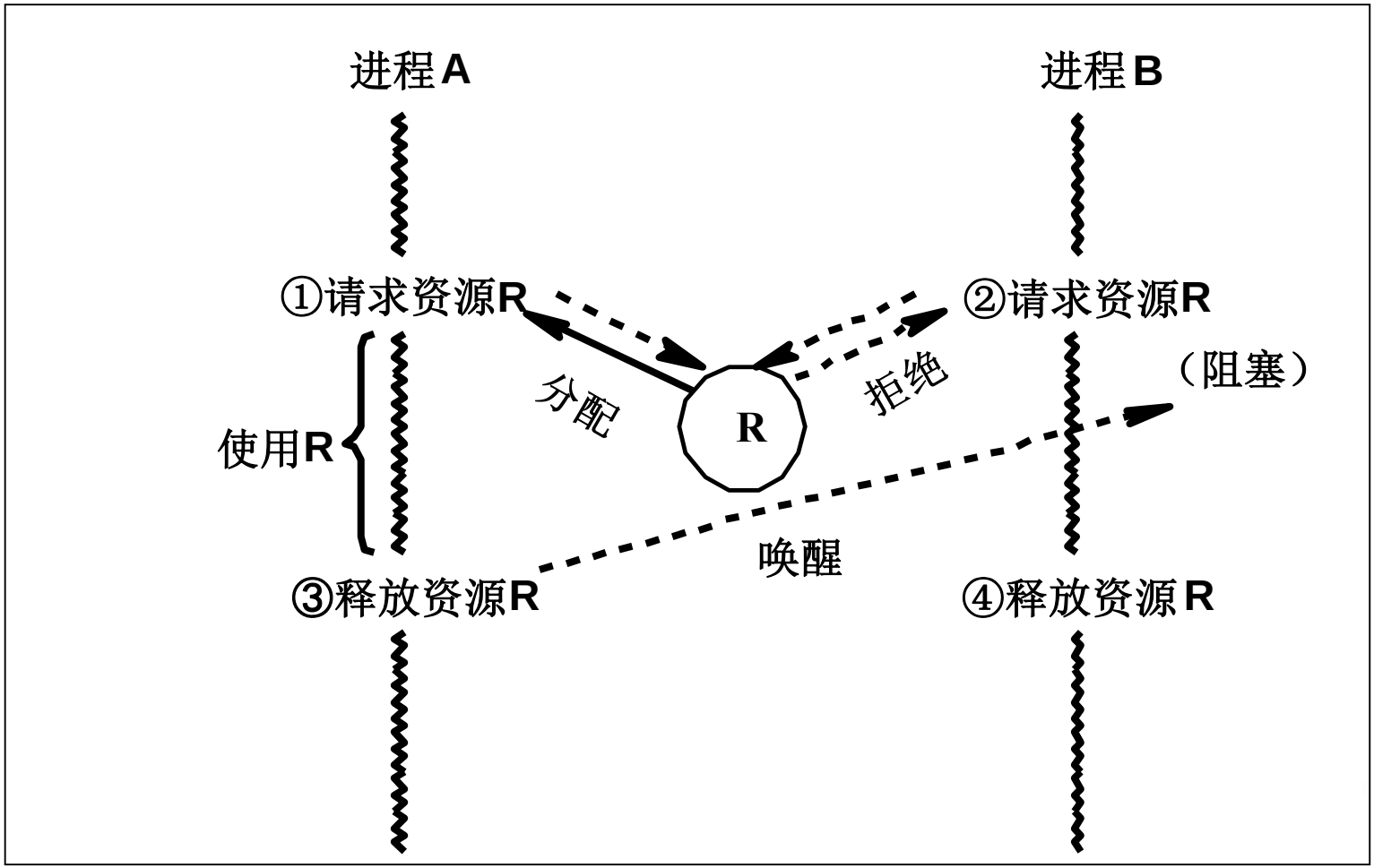

进程P1
计算 func1(x)
进程 P2 算
完 func2 (y)?
N
Y
取用P2 计算结果
进程P2
计算 func2(y)
置计算完标志
终止

## 互斥关系

- 进程—资源—进程
- 竞争到某一物理资源时不允许其它进程工作
- 相互之间不一定清楚其它进程情况
- 往往指多个任务多个进程间通讯制约故更广泛

进程之间彼此无关，但是由于竞争使用同一共享资源而产生了相互约束的关系。这种因共享资源而产生的制约关系称为进程的互斥—间接相互制约关系

## ●互斥问题举例：



# 3.1.1 基本概念

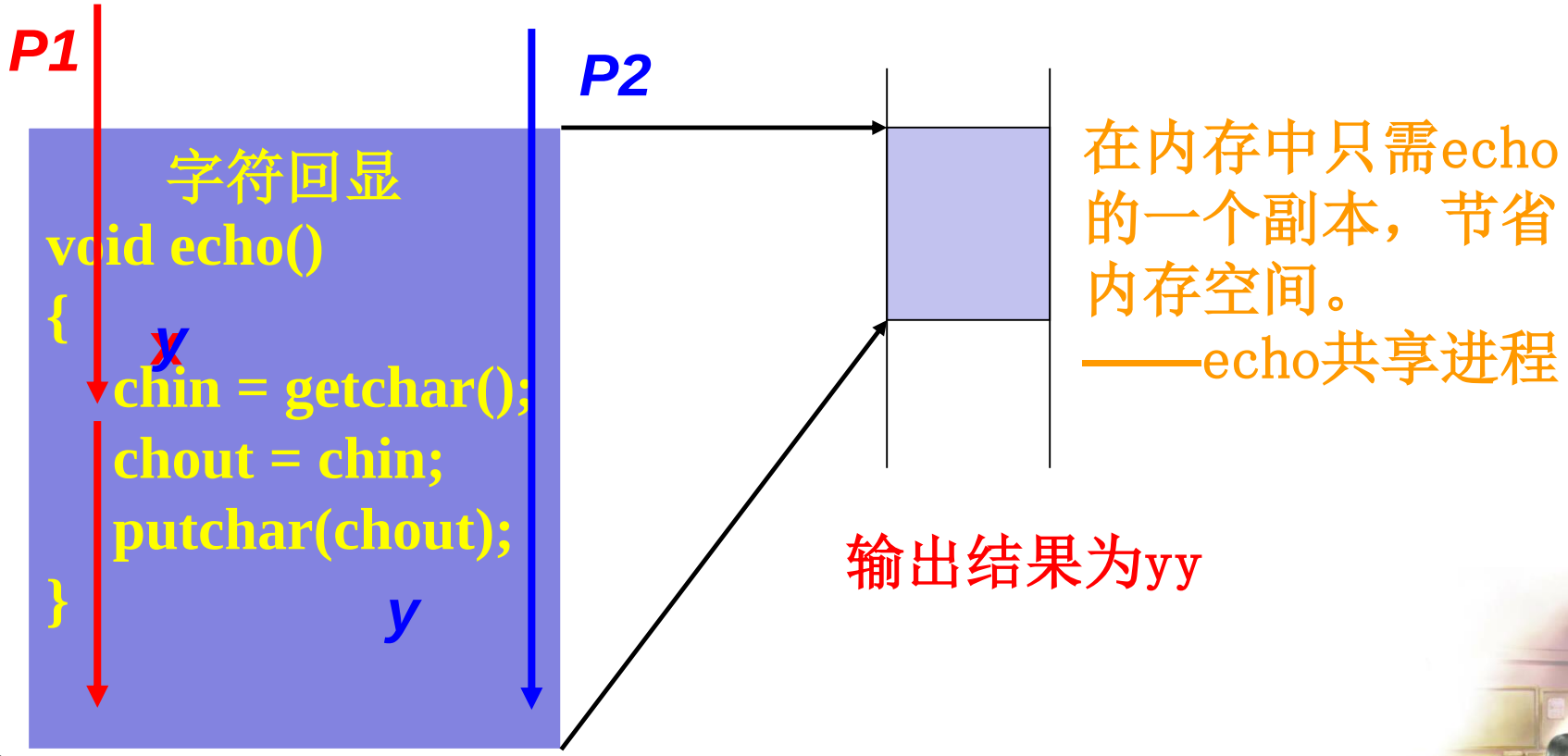
## 1. 什么是临界资源

凡是以互斥方式使用的共享资源都称为临界资源。临界资源具有一次只允许一个进程使用的属性。

- 计算机的许多硬件资源都被处理成临界资源，如打印机、磁带机等。例如，若打印机允许若干用户同时打印，则打印结果会混在一起，不易分辨，给用户带来许多不便。
- 系统中还有许多软资源，如共享变量、表格、队列、栈等也被处理成临界资源，以避免多个进程对它们访问时出现问题。

# example：程序A和B都要调用echo函数

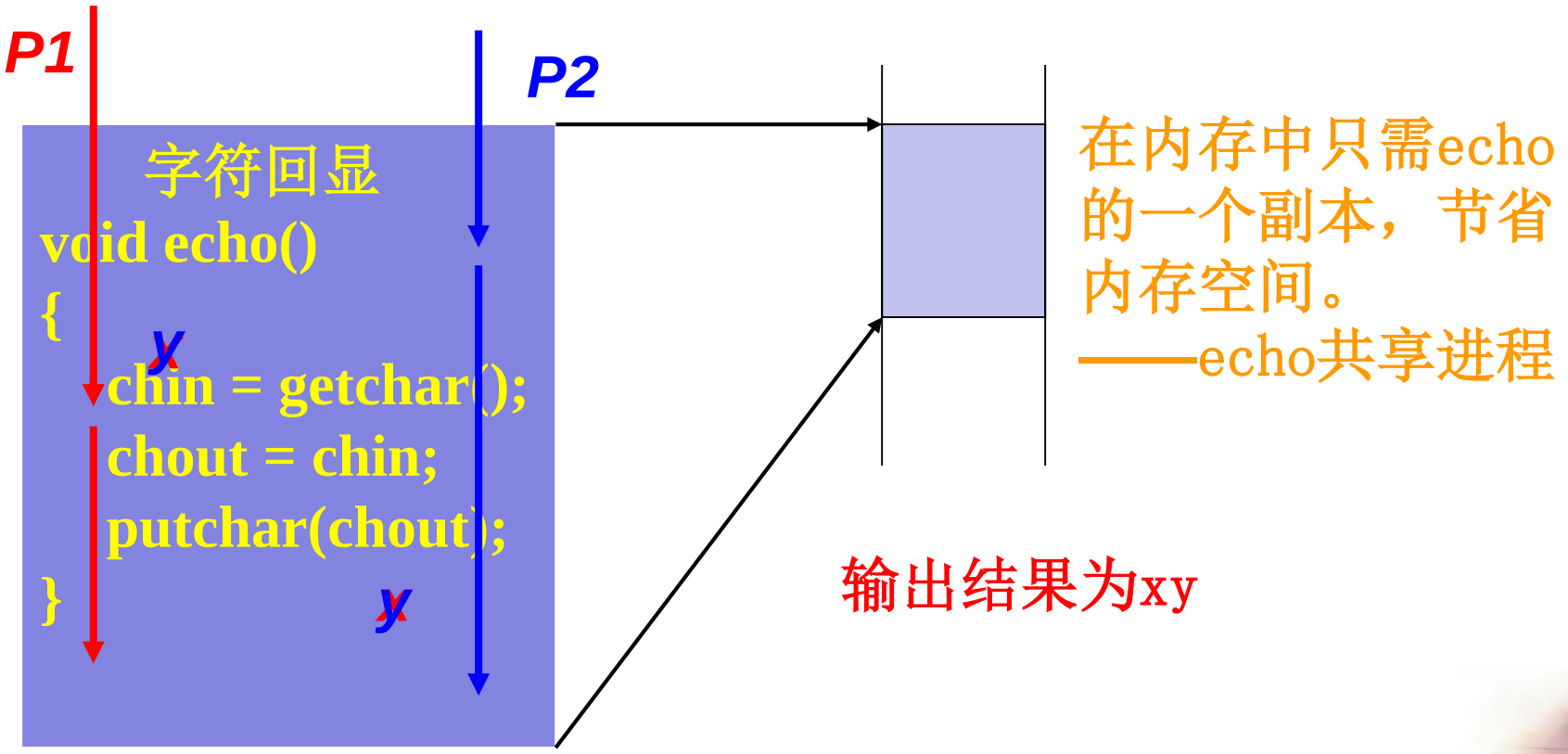
- 进程并发执行的相对执行速度是不可预测的。



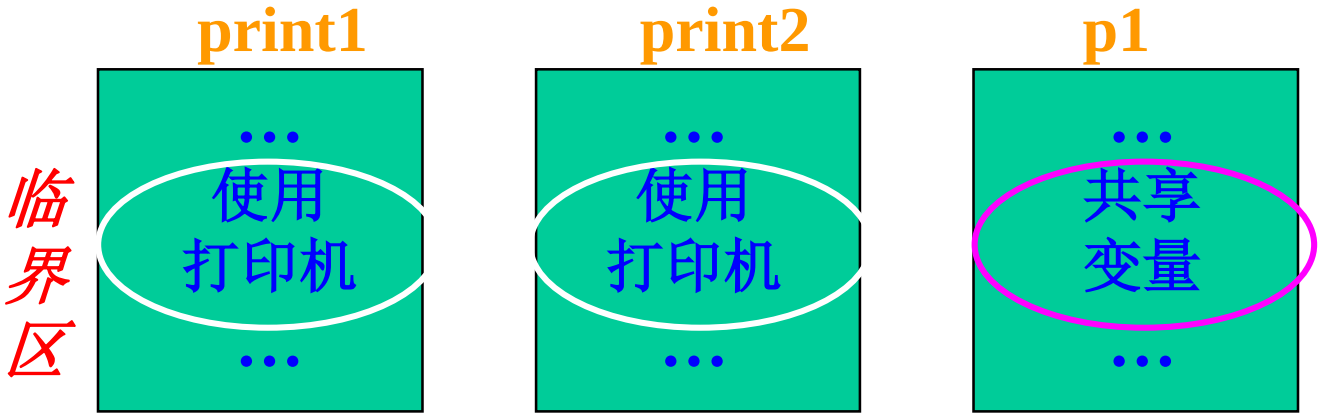
# 启发

- example中问题的根本是访问共享的全局变量造成的，有何启发？
- 既然对echo的交替执行会引发执行结果的不同，由此想到对访问该变量的代码要进行控制。
- 解决方法：一次只允许一个进程进入echo，并且只有在echo过程运行结束后，另外一个进程才能进入执行。

- **example：程序A和B都要调用echo函数**



# 2. 临界区（critical section）




- 每个进程中，访问临界资源的那段代码称为临界区。
  - 临界区是一段代码。
  - 这些程序段分散在不同的进程中。
- 临界区可按不同的共享资源划分成不同的集合。如：{print1,print2}
  - 同属一个集合中的临界区不允许并发执行，必须互斥。

- 如何保证同一集合的临界区不会并发执行呢？
  - 在进入临界区之前先检测是否能够进入。如果有其它进程已进入临界区使用临界资源，则必须等待。否则，允许进入，同时设置标志表明有进程正在临界区内。 ——进入区 entercritical
  - 退出临界区后，要修改标志为无进程使用该临界资源。 ——退出区 exitcritical
  - 互斥机制**entercritical**、**exitcritical**有多种实现方法：加锁、信号量、管程等。

# 临界区的代码构成

```
void P1
{     while(true)
    { …    //preceding code
          entercritical
      …    //critical  section
          exitcritical
       …  //following code
    }
}
```

进入区：申请进入临界区。
进入前，先测试临界资源是否被占用；未占用，则进入，同时设被占用标志。已占用，则不能进入。

临界区：访问临界资源

退出区：释放临界资源，
修改资源标志为未占用。

# 小结

- 受到制约的这些进程，不管是受到直接制约还是受到间接制约，执行时都要依次执行。
- 或者说并发执行的同时在受到制约的语句部分要顺序执行。
- 所不同的是，受到直接制约的要求有明确的执行次序。同步 Synchronization
- 受到间接制约的进程，谁先执行都可以，但一旦一个先执行，其余需要等待。互斥 Mutual Exclusion
- 同步的概念比互斥要强一些。

进程的同步机制就是在进程异步运行时，在执行时序上施加某些限制，使其对共享资源的操作与时间无关。

## 3、同步机制应遵循的互斥准则

- 空闲让进　无进程处于临界区内时，可让一个申请进入该临界区的进程进入。
- 忙则等待　临界区内有进程时，申请进入临界区的进程必须等待。
- 有限等待　进程进入临界区的请求，必须在有限的时间内满足。
- 让权等待　等待进入临界区的进程，必须立即释放CPU。

- 互斥的实现
  - 软件方法：忙等待，浪费资源。
  - 硬件方法。
  - 信号量、管程等。
- **Busy Waiting 忙等待**
  - 进程一直在检测是否能进入临界区，在没有进入之前，一直在做一些检测工作。

# 3.1.2 硬件同步机制

## 1.利用测试与设置(Test_and_Set)指令

```
boolean TS(int i)
{   if(i==0)
          { i=1;
            return true;}
    else { return  false;}
}
```

```
int lock=0;
```

```
do{ while (!TS(lock))do skip;
      critical section;
      lock=0;
      remainder section;
    } while (TRUE);
```

## 2. 利用交换(Exchange)指令

```
void Exchange(int register,int memory)
{   int temp;
    temp=memory;
    memory=register;
    register=temp;
}
```

```
do{  int  key=1;
       while (key==1)
                  Exchange(key,lock);
       critical section;
       lock=0;
    } while (TRUE);
```

# 3. 使用硬件指令的优缺点：

- 优点：

  (1) 适用于单处理机或共享内存的多处理机上的任何数目的进程。

  (2) 非常简单且易于证明。

  (3) 可用于支持多临界区。

- 缺点：

  (1) 使用了“忙等待”，不符合“让权等待”原则，造成处理机时间的浪费。

  (2) 可能饥饿。当一个进程离开一个临界区并且有多个进程正在等待时，选择哪一个等待进程是任意的。

  (3) 可能死锁。

# 3.1.3 信号量机制

## 1.什么是信号量

并发进程间的相互制约关系从本质上说是由于争夺和共享资源而产生的。将资源抽象为信号量（Semaphore），在信号量基础上引入同步操作原语： P操作、V操作。

## 2. 整型信号量

最初由Dijkstra把信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作被分别称为P、V操作。

wait和signal操作可描述为：

*违背“让权等待”原则*

定义：Semaphore: S;

wait(S): while S < =0 do no-op

S=S-1;

signal(S): S=S+1;

# 3. 记录型信号量

```
定义：
    struct  semaphore{
        int  count;                  信号量值
        queueType  *queue ;          信号量等待队列指针
    };
```

其中：

- 信号量值—表示某种资源的数量。
- 等待队列指针—当信号量值为负时，表示该类资源已分配完，等待该类资源的进程排在等待队列中。L为指向该信号量等待队列的指针。

定义：Semaphore: S；

a. P操作（wait原语）

每执行一次P操作，申请分配一个单位的资源。

P(S) — 对信号量S进行P操作。

① S.count = S.count - 1；

② 若 S.count≥ 0 则进程继续执行。

若 S.count < 0 则进程阻塞，并进入等待队列(L)。

b. V操作（Signal原语）

V(S) — 对信号量S进行V操作，释放一个单位的资源。

① S.count= S.count + 1；

② 若 S.count> 0 则进程继续执行。

若 S.count≤ 0 则唤醒S等待队列中的一个进程，使之转为就绪状态。



## *P、V操作的算法描述*

**P 操作：**

```
P（S）{
   semaphore：S；
   S.count= S.count--；
   if (S.count < 0) block（S.queue）；
  }
```

**V 操作：**

```
V（s）{
   semaphore：S；
   S.count= S.count++；
   if (S.count< =0)  wakeup（S.queue）；
   }
```



说明：

① S.count> 0时，其值表示某类资源可用数量。

S.count≤0时，其绝对值表示在信号量队列中等待该资源的进程数。

② P、V操作有严格不可分割性；执行过程不允许中断；

③ P、V操作成对出现。

？问题？

如何使用P、V操作实现同步机制？

考虑：
- 如何控制互斥地使用临界资源？
- 如何控制进程并发执行的时序？

实现同步机制基本思想是：加锁、解锁

# 4. 信号量的应用

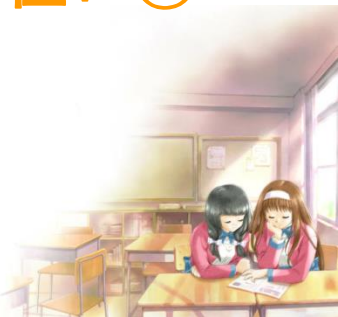
## (1) 用于实现互斥

设 mutex —公共互斥信号量 初值： mutex = 1

**利用P、V操作实现互斥的模型**

| 进程P1 | | 进程P2 | |
|---|---|---|---|
| P（mutex）； | 值0 ① | P（mutex）； | 值-1 ② |
| 进入P1临界区； | | 进入P2临界区； | |
| V（mutex）； | 值0 ③ | V（mutex）； | 值1 ④ |

*设先执行进程P1*



## 利用信号量实现进程互斥的进程描述为：

定义公共的互斥信号量：mutex　初值：　mutex = 1

执行过程中mutex的值，在 1，0，-1 之间变化。

```
semaphore  mutex =1;
 {
 process  1:
      do{
          P (mutex);
          critical  section;
          V (mutex);
           remainder  section;
         } while  true;
```

```
process 2:
      do{
          P (mutex);
          critical  section;
          V (mutex);
           remainder  section;
          } while true;
 }
```

为了实现进程互斥地进入临界区，只须把临界区CS置于P (mutex)和V (mutex)之间。



## (2) 用于实现同步

例如，有两个并发进程P1、P2，共享一个公共信号量s，初值为0。P1执行的程序中有一条S1语句，P2执行的程序中有一条S2语句。而且，只有当P1执行完S1语句后，P2才能开始执行S2语句。

假设semaphore S=0； // 信号量初值为0；

```
void P1（）{    ……
                S1;
              signal（S）;
               ……  }
void P2（）{  ……
              wait（S）;
               S2;
              ……  }
void main（）{  parbegin（P1(),P2()）; }
```

# 3.2 经典进程同步问题

## 3.2.1 生产者—消费者问题

a. 一个生产者，一个消费者，一个缓冲区，生产者生产消息，并将此消息提供给消费者进程去消费。

单一资源

生产者关心？

消费者关心？

信号量empty，full

生产者进程：

生产一个产品；
P（empty）；
将产品 放入缓冲区；
V（ full）；

消费者进程：

P（full）；
从缓冲区取产品；
V（ empty）；
消费产品；

empty的初值为1， full的初值为0

## b. 一个生产者，一个消费者，n个缓冲区。

empty的初值为n，full的初值为0

有限资源

生产者进程：

```
in=0;
do{
  生产一个产品；
  P（empty）；
  往buffer[in] 放产品；
  in=（in+1）% n;
  V（ full）；
}while(1);
```

消费者进程：

```
out=0;
do{
  P（full）；
  从buffer[out]取产品；
  out=（out+1） % n;
  V（ empty）；
  消费产品；
}while(1);
```

c．一个生产者，一个消费者，∞缓冲区。

full的初值为0

无限资源

生产者进程：

```
in=0;
生产一个产品；
往buffer[in] 放产品;
in=in+1;
V（ full）；
```

消费者进程：

```
out=0;
P（full）；
从buffer[out]取产品；
out=out+1;
消费产品;
```

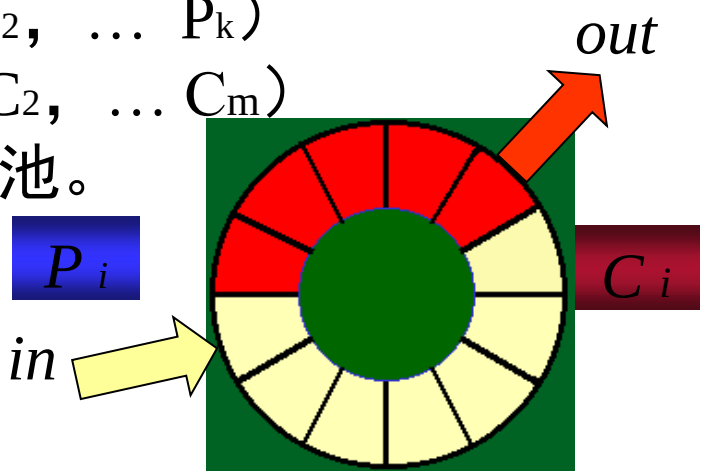
# 一般的生产者—消费者问题

**问题？**

一组生产者进程 $P_i$（$P_1$，$P_2$，… $P_k$）

一组消费者进程 $C_i$（$C_1$，$C_2$，… $C_m$）

互斥使用由n个缓冲区组成的缓冲池。




**分析**

## 1.同步关系：

- 当缓冲池放满产品时生产者必须等待。

  定义生产者进程同步信号量： empty-表示空闲缓冲区数。

  $0 \le empty \le n$　　empty 初值为 *n* ；

- 当缓冲池空时，消费者进程必须等待。

  定义消费者进程同步信号量：full-表示有产品的缓冲区数。

  $0 \le full \le n$　　full 初值为 *0* ；

## 2.定义 in ,out分别表示首空缓冲区序号及首满缓冲区序号。

semaphore empty，full=n , 0

message buffer[n];

in, out = 0,0

*思考：*

*两个生产者交替执行，可不可以？*

生产者进程：

```
生产一个产品 m ；
      ⋮
P（empty）；

将产品 m放入缓冲区；
in =（in +1） % n ；

V（ full）；
```

消费者进程：

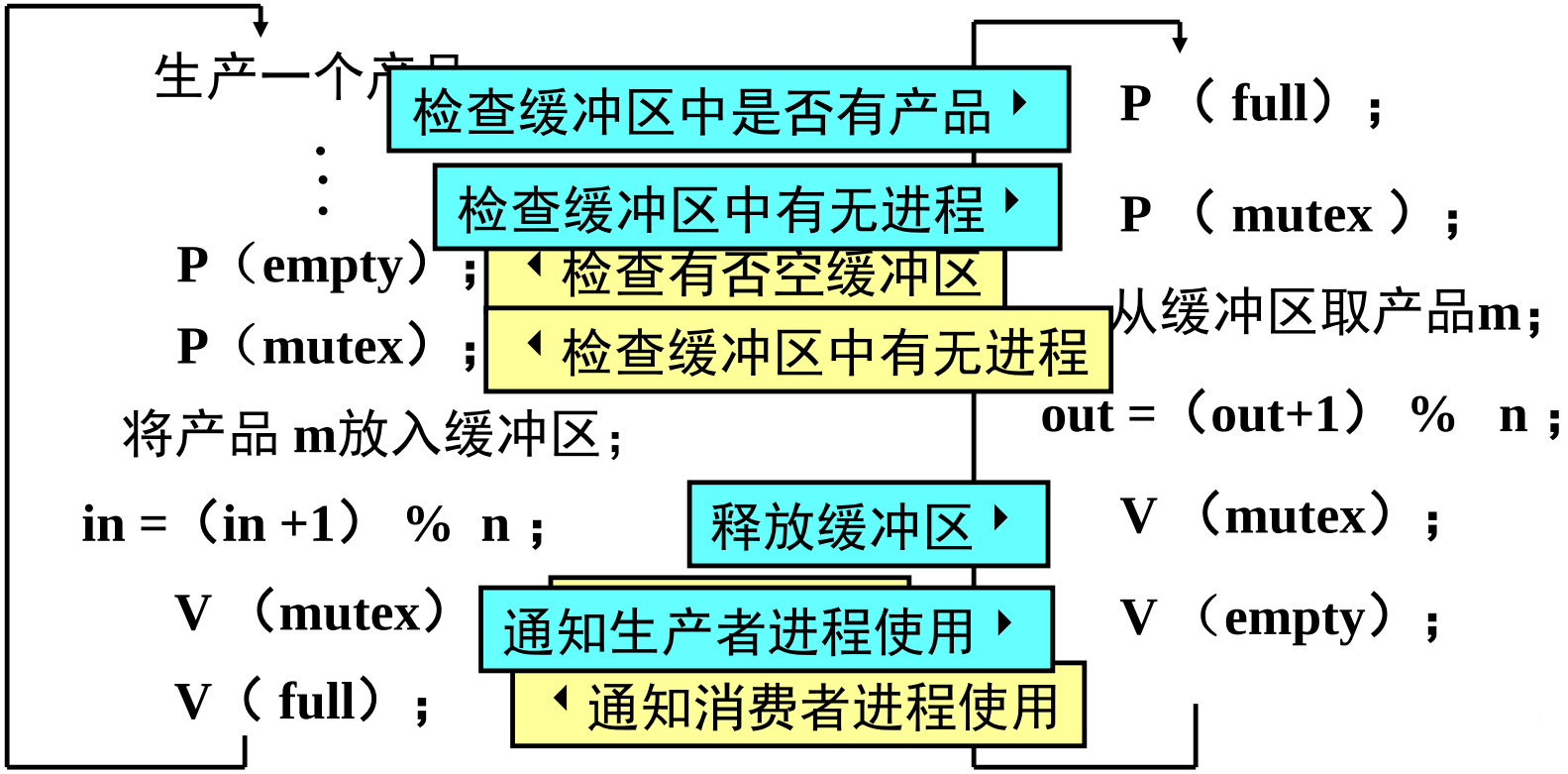
```
P （ full）；

从缓冲区取产品m；
out =（out+1） % n ；

V （empty）；
```

- 该问题不仅需要同步，还需要互斥。
- 互斥：
  - 缓冲区是临界资源，每个进程要互斥使用缓冲区。
  - 定义公共互斥信号量**mutex**，初值为1。

semaphore empty， full=n , 0;
semaphore mutex=1;
message buffer[n];
in, out = 0,0;

生产者进程：

消费者进程：



**生产者进程：**

```
生产一个产品 m ；
   ⋮
P（empty）；
P（mutex）；
将产品 m放入缓冲区；
in =（in +1） % n ；
V （mutex）；
V（ full）；
```

**消费者进程：**

```
P （ full）；
P （ mutex ）；
从缓冲区取产品m；
out =（out+1） % n ；
V （mutex）；
V （empty）；
```

**问题**

1. 能否交换两个 P操作？能否交换两个V 操作？为什么？
2. 如果缺少操作：P（empty）或 P ( full），结果如何？



思考下面的情况：
缓冲区全空，消费者先执行

```
const int n= /*buffer size*/
semaphore empty=n;
semaphore full=0;        semaphore mutex=1;

void producer()
{
        while(true)
        {  生产一个产品
           Wait(mutex);    0-1=-1
           Wait(empty);
           将产品放入缓冲区；
           in=(in+1)%n:
           Signal(full);
           Signal(mutex);
        }
}

void consumer()
{    while(true)
     {
          Wait(mutex);   1-1=0
          Wait(full);    0-1=-1
          从缓冲区中取产品；
          out=(out+1)%n;
          Signal(mutex);
          Signal(empty);
          消费该产品；
     }
}

void main()
{   parbegin( producer,consumer );
}
```

# 结论

- 进程中若有多个**Wait**，要求同步信号量的**Wait**操作在前，互斥信号量的**Wait**操作在后，以免引起死锁。
- 进程中若有多个**Signal**，其操作顺序无所谓。

# 同步与互斥的解题思路

- ①分清哪些是互斥问题（互斥访问临界资源的），哪些是同步问题（具有前后执行顺序要求的）。
- ②对互斥问题要设置互斥信号量，不管有互斥关系的进程有几个或几类，通常只设置一个互斥信号量，且初值为1，代表一次只允许一个进程对临界资源访问。
- ③对同步问题要设置同步信号量，通常同步信号量的个数与参与同步的进程种类有关，即同步关系涉及几类进程，就有几个同步信号量。同步信号量表示该进程是否可以开始或该进程是否已经结束。
- ④在每个进程中用于实现互斥的PV操作必须成对出现；用于实现同步的PV操作也必须成对出现，但可以分别出现在不同的进程中；在某个进程中如果同时存在互斥与同步的P操作，则其顺序不能颠倒，必须先执行对同步信号量的P操作，再执行对互斥信号量的P操作，但V操作的顺序没有严格要求。



# 3.2.2 读者—写者问题

## 问题描述

有一个数据区（可以是一个文件、一块内存空间或是一组寄存器）被多个用户共享，其中从里面读数据的进程被称为读者，向里面写数据的进程被称为写者。

规定：读者对数据区是只读的，而且允许多个读者同时读；写者对数据区是只写的，当一个写者正在向数据区写信息的时候，不允许其他用户使用。即保证一个写者进程必须与其他进程互斥地访问共享对象。

# 问题分析

- 读者可以同时读。
- 读写互斥。
- 写写互斥。

考虑下面的进程到达序列：R1R2W1R3R4…Rn

第一类：“读者优先”

第二类：“写者优先”

## 第一类：“读者优先”

```
semaphore wsem=1;

void reader()
{ while(true)
  {
    readunit();

  }
}

void writer()
{ while(true)
    { wait(wsem);
      writeunit();
      signal(wsem);
    }
}
```

## 第一类：“读者优先”

```
semaphore wsem=1;

void reader()
{ while(true)
  { wait(wsem);
    readunit();
    signal(wsem);
  }
}

void writer()
{ while(true)
    { wait(wsem);
      writeunit();
      signal(wsem);
    }
}
```

## 第一类：“读者优先”

```
semaphore wsem=1;
int readcount=0 ;

void reader()
{ while(true)
  {
    readcount++;
    if readcount==1 wait(wsem);

    readunit();

    readcount--;
    if readcount==0 signal(wsem);

  }  }
```

```
void writer()
{ while(true)
  { wait(wsem);
    writeunit();
    signal(wsem);
  }
}
```

## 第一类：“读者优先”

```
semaphore wsem=1;
int readcount=0 ;

void reader()
{ while(true)
  {
    readcount++;
    if readcount==1 wait(wsem);

    readunit();

    readcount--;
    if readcount==0 signal(wsem);

  }   }
```

```
void writer()
{ while(true)
   { wait(wsem);
     writeunit();
     signal(wsem);
    }
  }
```

```
semaphore wsem=1;  rsem=1;
int readcount=0 ;

void reader()
{ while(true)
  { wait(rsem);
    readcount++;
    if readcount==1 wait(wsem);
    signal(rsem);
    readunit();
    wait(rsem);
    readcount--;
    if readcount==0 signal(wsem);
    signal(rsem);
  }  }
```

读者申请使用文件

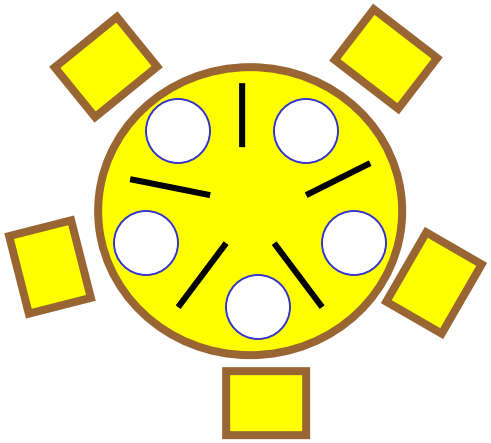
阻塞写者进程

读者释放文件
让其他读者进入

读者退出

```
void writer()
{ while(true)
   { wait(wsem);
     writeunit();
     signal(wsem);
   }
}
```

# 3.2.3 哲学家进餐问题



**分析：为每支筷子设置一个信号量，一个哲学家， 通过在相应的信号量上执行P操作，抓起一支筷子，通过执行V操作放下一支筷子。**

**5个信号量构成数组：**

```
semaphore   chopstick [5];
```

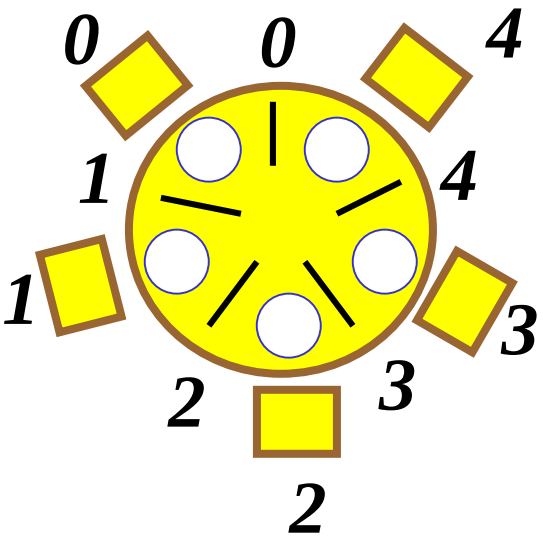
**同时拿起筷子，出现死锁!**

第i个哲学家的活动可描述为：

```
do{
    P（ chopstick [i]）；                          抓起左边筷子；
    P（ chopstick [（i+1） % 5 ] ）；              抓起右边筷子；
            :
            Eat ;                                  吃饭；
    V(chopstick [i]）；                            放下左边筷子；
    V( chopstick [（i+1） % 5 ] ）；               放下左边筷子；
            :
        Think ;                                    思考；
            :
}while (true);
```



# 解决办法

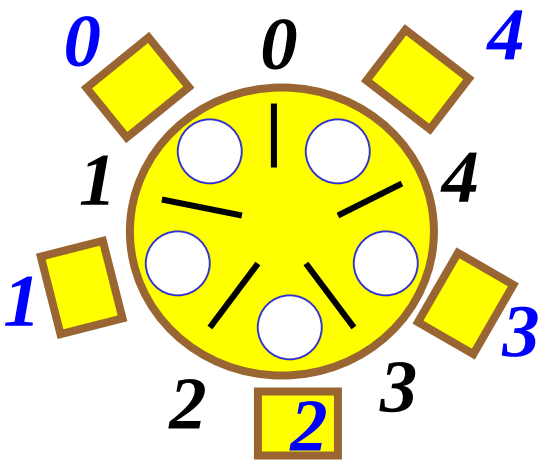



- (1)规定奇数号哲学家先拿他左边的叉子，然后再去拿右边的叉子；而偶数号哲学家则相反。按此规定，0、 1号哲学家竞争1号叉子；2、3号哲学家竞争3号叉子。
- (2)至多有4个哲学家同时拿叉子。

# 解决方法一

```
void Philosophers (int i)
{   while(true)
    {     think( );
          if( i%2= =0)        {   Wait( chopstick[i] );
                                  Wait( chopstick[(i+1) mod 5]);
                              }
          else                {   Wait( chopstick[(i+1) mod 5]);
                                  Wait( chopstick[i]);
                              }
         eat( );
         Signal(chopstick[i]);
        Signal(chopstick[(i+1) mod 5]);
    }
}
```

```
/*program diningphilosophers*/
semaphore chopstick[5]={1,1,1,1,1};     int i;
semaphore room=4;
void Philosophers (int i)
{    while(true)
     {      think( );
            Wait( room );
            Wait( chopstick[i]
            Wait( chopstick[(i+1) mod 5]
            eat( );
            Signal( chopstick[i] );
            Signal( chopstick[(i+1) mod 5] );
            Signal( room );
      }
}
void main( )
{   parbegin(Philosophers(0), Philosophers(1), Philosopher(2),
    Philosopher(3), Philosopher(4));   }
```




解决方法二

# 3.3 管程

**Dijkstra(1971)：“秘书”进程**
**Hansan和Hoare(1973)：管程**

**管程的基本思想：**
——加围墙，统管

将分散的各同类临界区集中起来，为每一类共享资源设置一个“管程”，统一控制和管理各进程对该资源的访问，即任何一个进程要访问共享资源，必须通过管程。

# 3.3.1 管程的基本概念

管程（monitor）是更高级的同步机制，能够进一步实现复杂的并发性问题。

## 1.为什么引入管程

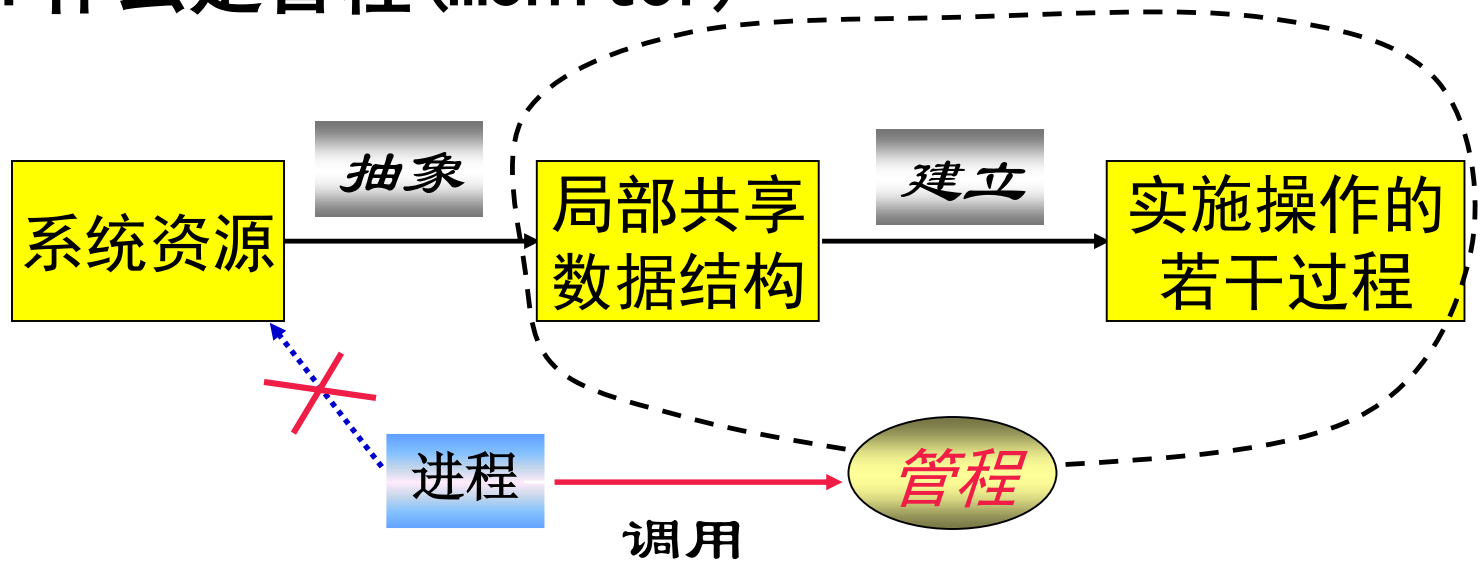
采用PV同步机制来编写并发程序，对临界区的执行分散在各个进程中，缺点：

(1) 易读性差。因为要了解对于一组共享变量及信号量的操作是否正确，就必须通读整个系统或者并发程序。

(2) 不利于修改和维护。因为程序的局部性很差，所以任一组变量或一段代码的修改都可能影响大局。

(3) 正确性难以保证。因为操作系统或并发程序通常很大，要保证这样一个复杂的系统没有逻辑错误是很难的。

# 2. 什么是管程(monitor)




## 管程的组成

- 管程名
- 局部于管程的共享变量说明
- 对该数据结构实施操作的若干函数
- 对局部于管程的数据设置初始值的语句

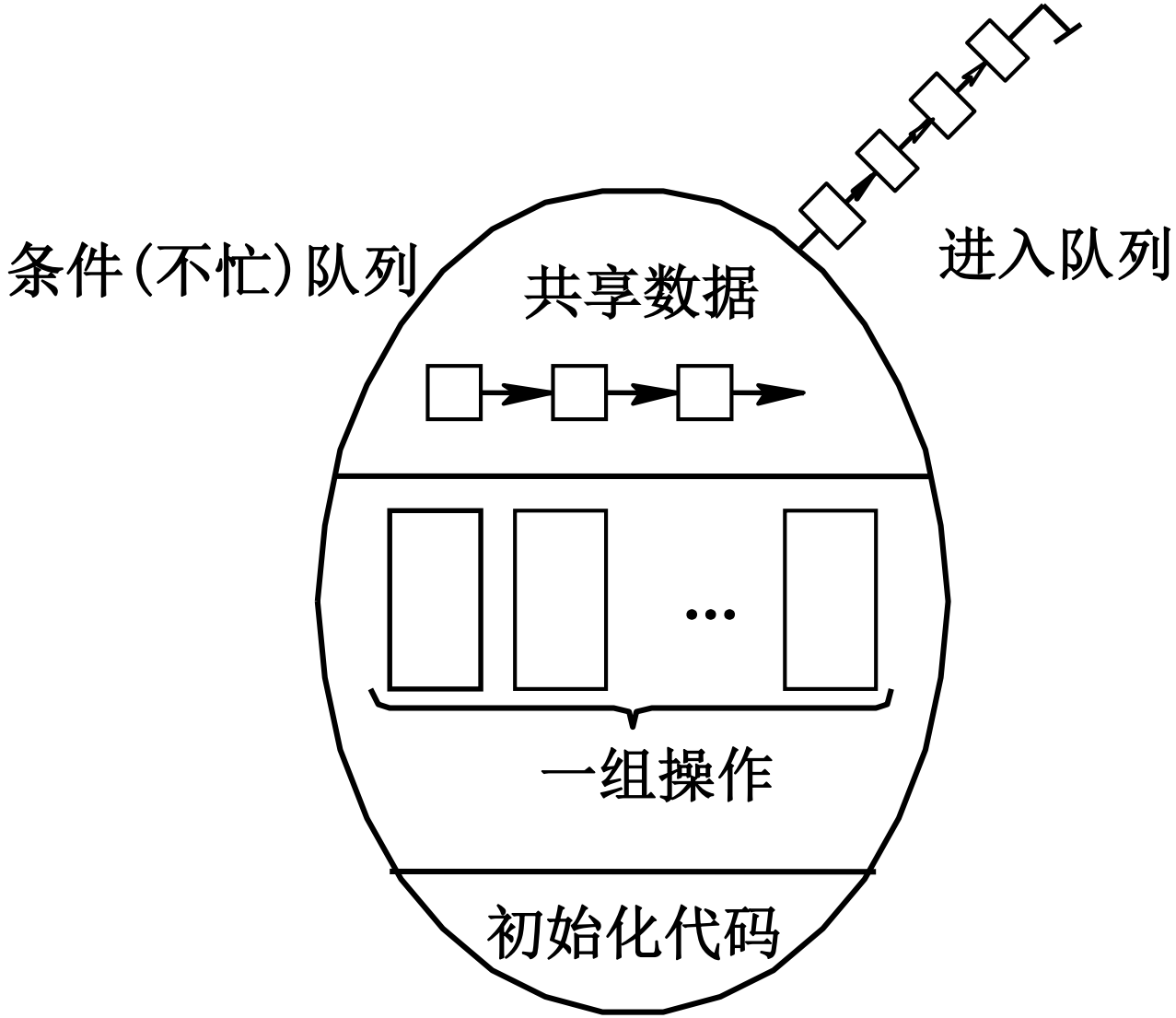



图 3-3　管程的示意图

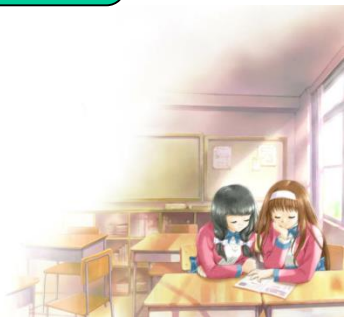
## 3.管程最主要的特点：

（1）只能通过管程中的过程而不能用其他外部过程访问其局部数据变量。

（2）进程通过调用管程的过程而进入管程。

（3）每一时刻只能有一个进程在管程中执行，任何其他调用管程的进程将被挂起直至管程可用。

*可以实现互斥*



# 4.条件变量

假设一个进程调用了管程，当它在执行过程中发现条件不满足时，应将该进程阻塞。这就需要一种机制，使得该进程不仅被阻塞，而且能释放这个管程，以便其他进程可以进入。

为区别不同的等待原因，引入条件变量(condition)；将wait原语和signal原语修改为：

〈条件变量名〉.wait()

〈条件变量名〉.signal()

## condition c;

- 在管程中执行的某进程发现条件未能满足时，发送wait使该进程阻塞。
  - c.wait()：等待操作，用来将执行进程挂到与条件变量c相应的等待队列上。
- 当管程中执行的另一进程发现条件c满足了，则发送signal通知c条件队列，条件已改变。
  - c.signal()：发信号操作，用来恢复与X相应的等待队列上的一个进程。若没有等待进程，则c.signal不起任何作用，什么都不做。

讨论：

如果有进程Q处于阻塞状态， 当进程P执行了X.signal操作后，怎样决定由哪个进程执行，哪个等待，可采用下述两种方式之一进行处理：

(1) P等待，直至Q离开管程或等待另一条件。（Hoare,进程切换过于频繁）

(2) Q等待，直至P离开管程或等待另一条件。（效率高，但它导致逻辑关系不清晰）

(3)规定唤醒为管程中的最后一个可执行的操作。（Hansan,是(1) 的特例）

# 5. 利用管程实现进程同步

步骤：

✶ 为进程建立管程，其格式为：

```
typedef monitor monitor_name
 variable declarations

   p1( ... );
       {...};
        .
        .
        .
   pn( ... );
       {...};

{initialization  code}
```

定义管程名、局部变量

一组操作

设置变量初值

✸ 在并发进程中调用管程

# 管程与信号量的对比

- **c.wait()/c.signal()与Wait()/Signal()的区别：**
  - 使用**c.wait()**会挂起当前进程。而执行**Wait()**未必阻塞。
  - 如果没有挂起进程，**c.signal()**什么都不做。

# 3.3.2 用管程解决生产者-消费者问题

- 定义一个管程PC，管理有界缓冲区。
- 管程中有两个条件变量：
  - notfull：缓冲区未全满，变量值为true；
  - notempty：缓冲区未全空，变量值为true。
- 为该管程定义两个过程：
  - 放产品 append(char x)；
  - 取产品 take(char x)；

## 用管程解决

```
typedef  monitor  producer-consumer
    int nextin,nextout,count;
    char buffer[N];
    condition  notfull, notempty;
      append(char x)
      {
       if  (count==N)  notfull.wait();
        buffer[nextin]=x;
        nextin =(nextin+1) % N;
        count =count++;
        notempty.signal();
       }
```

## 用管程解决

```
 take(char x)
{
      if  (count==0)   notempty.wait();
      x=buffer[nextout];
      nextout =(nextout+1) ％ N;
      count =count--;
      notfull.signal();
}
{nextin=nextout=0; count=0 }
```

## 用管程解决

在利用管程解决生产者-消费者问题时， 其中的生产者和消费者可描述为：

```
Producer :
        do
         {produce an item in x;
          PC.append(x);
         }while  (true);
Consumer:
        do
         {PC.take(x);
           consume the item in x;
         }while  (true);
```

# 管程与信号量的对比

- c.wait()/c.signal()与Wait()/Signal()的区别：
  - 使用c.wait()会挂起当前进程。而执行Wait()未必阻塞。
  - 如果没有挂起进程，c.signal()什么都不做。
- 解决同步互斥问题时：
  - 使用信号量：实现同步和互斥都是程序员的责任。
  - 使用管程：管程本身的互斥机制使得生产者、消费者不能同时访问缓冲区。但实现同步需要程序员使用c.wait()、c.signal()实现。
- 与信号量相比，管程的优势在于所有的同步机制都限制在管程内部，因此不仅易于验证同步的正确性，而且易于检测错误。

# 3.4 进程通信

进程通信（communication）是指进程间的数据交换。显然，进程同步是一种简单的通信方式，但交换的信息量少，通信效率低，称为低级通信方式。

本节讨论的是高效传递大量信息的高级通信方式。

讨论以下问题

进程通信的类型

消息传递通信的实现方式

消息缓冲队列通信机制

# 进程通信类型

## 3.4.1共享存储器系统(Shared-Memory System)

在内存中开辟一块共享存储区，供进程之间交换信息。

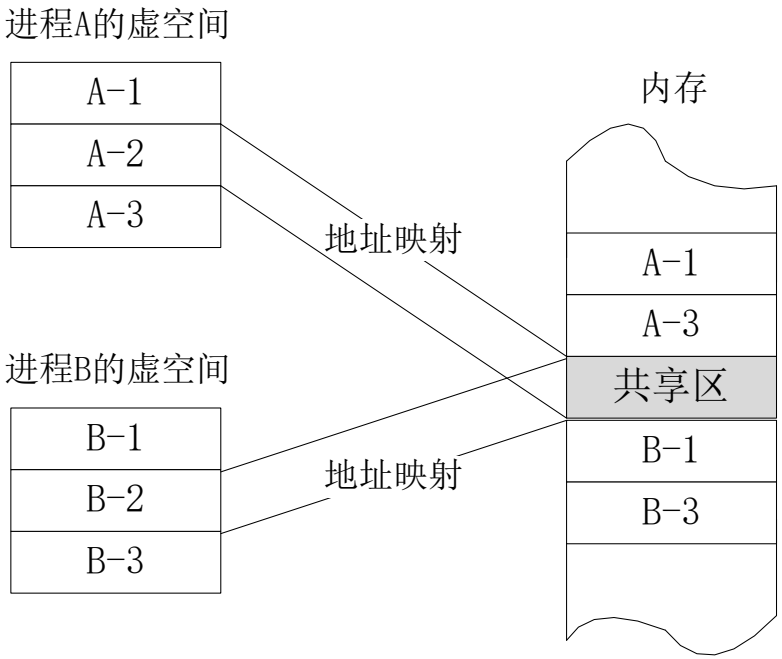



图3-4 地址空间映射示意图

## 3.4.2 管道(Pipe)通信（能有效地传送大量数据）

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：① 互斥 ② 同步③ 状态测试。确定对方是否存在，只有确定了对方已存在时，才能进行通信。

## 3.4.3 消息传递系统(Message passing system)

是目前单机系统、多机系统及计算机网络中的主要进程通信机制。

进程间的数据交换以格式化的消息为单位。

因实现方式的不同，分为：

1. 直接通信方式：消息缓冲队列
2. 间接通信方式：信箱通信

# 1.直接消息传递系统（直接通信方式）

(1) 直接通信原语：

Send(Receiver, message); 发送一个消息给接收进程；

Receive(Sender, message); 接收Sender发来的消息；

(2) 利用直接通信原语，来解决生产者-消费者问题。

```
do{   …
      produce an item in nextp
         …
      send(consumer, nextp);
    }while  true;

do{ receive(producer, nextc);
         …
      consume the item in nextc;
     }while  true;
```

## 2.信箱通信（间接通信方式）

### (1) 信箱的结构

- 信箱头

  存放信箱标识符、信箱的拥有者、信箱口令、信箱的空格数等。

- 信箱体

  由若干个可以存放消息（或消息头）的信箱格组成。

## (2) 信箱通信原语

- 信箱的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。
- 消息的发送和接收。

  **Send(mailbox, message)**：将一个消息发送到指定信箱；

  **Receive(mailbox, message)**：从指定信箱中接收一个消息；

## (3) 信箱的类型

- 私用邮箱
- 公用邮箱
- 共享邮箱

# (4) 用信箱通信来解决生产者-消费者问题

```
const int capacity=/*buffering capacity*/
null=/*empty message*/
int i;
void producer(  )
{   message pmsg;
    while(true) {    receive(producemail,null);
                     pmsg=produce( );
                     send(consumemail,pmsg);
                 }
 }
```

## (4) 用信箱通信来解决生产者-消费者问题

```
void consumer(  )
{  message cmsg;
   while(true)  {  receive(consumemail,cmsg);
                   consume(cmsg);
                   send(producemail,null); }
  }
void main( )
{    create_mailbox(producemail);
     create_mailbox(consumemail);
     for(int i=1;i<=capacity;i++)   send(producemail,null);
    parbegin(producer(), consumer());
 }
```

## 3.消息缓冲队列通信机制

消息缓冲区结构：

- 发送者进程标识符（sender）
- 消息长度（size）
- 消息正文（text）
- 指向下一个消息缓冲区的指针（next）

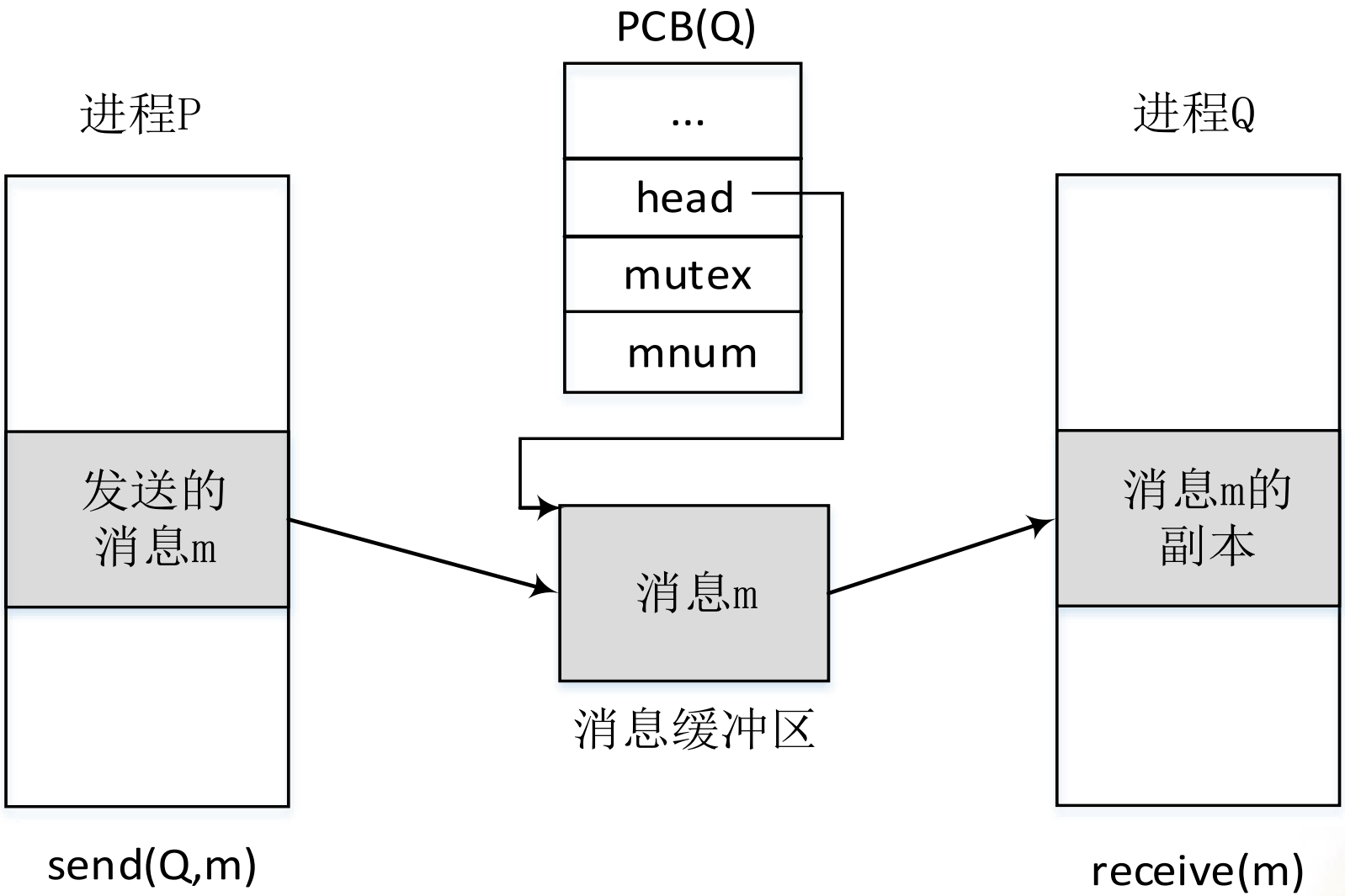



图3-5　消息缓冲队列通信示例

# 本章小结

- 临界资源和临界区，同步机制应遵循的准则；
- 信号量机制；
- 整型信号量；
- 记录型信号量（重点考查内容）；
- 理解经典进程同步问题中的几个算法，P操作顺序问题；
- 了解管程概念；
- 用管程实现进程的同步；
- 进程通信类型，直接通信和间接通信；
- 消息传递系统中的进程同步方式；
- 了解消息缓冲队列；



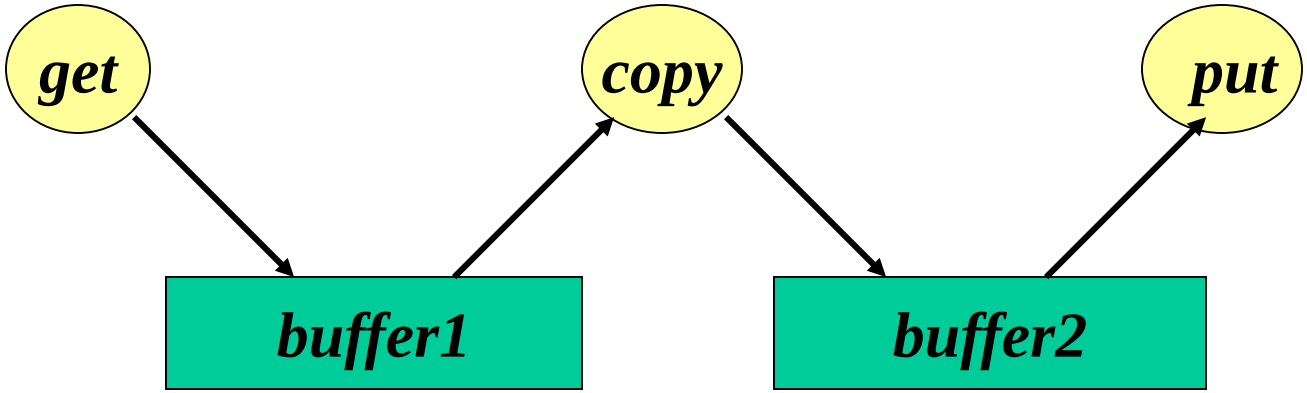
# 第三章 典型问题分析与解答

1.有一个阅览室，共有100个座位，读者进入时必须先在一张登记表上登记，该表为每一位列一表目，包括座号和读者姓名等，读者离开时要消掉登记的信息，试问：

① 为描述读者的动作，应编写几个程序，设置几个进程？

② 试用PV操作描述读者进程之间的同步关系。

2. 有三个进程 ，进程 get 从输入设备上不断读数据，并存入buffer1；进程 copy 不断将buffer1的内容复制到缓冲区buffer2，进程put 则不断将buffer2的内容在打印机上输出。三个进程并发执行，协调工作。写出该三个进程并发执行的同步模型。



3.请用信号量解决以下的“过独木桥”问题：同一方向的行人可连续过桥，当某一方向有人过桥时，另一方向的行人必须等待；当某一方向无人过桥时，另一方向的行人可以过桥。

## 作业题

- 有一个充分大的池子，两个人分别向池中扔球，甲扔红球，乙扔蓝球，一次扔一个，开始时池中有红、蓝球各一个，要求池中球满足条件：

$$1 \leqq \frac{\text{红球数}}{\text{蓝球数}} \leqq 2$$

用P、V操作描述两个进程。

## 作业题

- 有一个充分大的池子，三个人分别向池中扔球，甲扔红球，乙扔蓝球，丙扔绿球，一次扔一个，开始时池中有红、蓝、绿球各一个，要求池中球同时满足条件：

$$\begin{cases} 1 \le \dfrac{\text{红球数}}{\text{蓝球数}} \le 2 \\ \text{蓝球数} \le \text{绿球数} \le \text{红球数} + \text{篮球数} \end{cases}$$

用P、V操作描述三个进程。