

# Chap 6 进程同步





# 内容

1. 竞争条件和临界区
2. 信号量
3. 三个经典同步问题
  - ✓ 生产者消费者问题
  - ✓ 读者写者问题
  - ✓ 哲学家就餐问题
4. 管程



# 1、竞争条件和临界区





# 内容

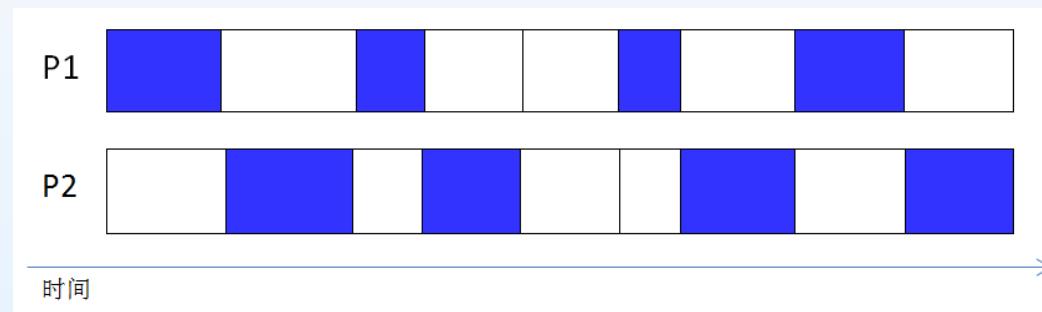
- 数据不一致性
- 有界缓冲问题
- 竞争条件
- 同步和互斥
- 临界资源
- 临界区
- 临界区使用准则





# 数据的不一致性

- 多个进程并发或并行执行
  - 每个进程可在任何时候被中断
  - 仅仅进程的部分代码片段可连续执行



- 共享数据并发/并行访问：数据不一致性
  - 又称不可再现性：同一进程在同一批数据上多次运行的结果不一样
  - 保证并发进程正确执行顺序的机制-**同步(互斥)机制**





# 数据不一致性例子：有界缓冲

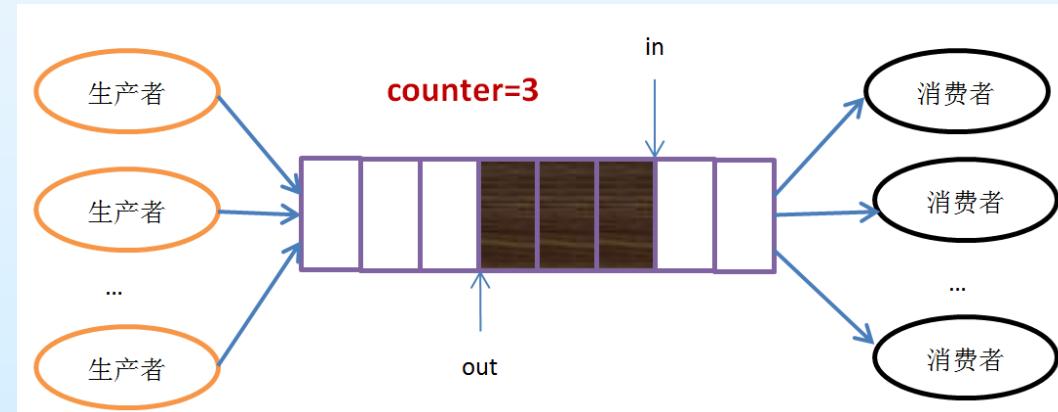
## ■ 例子：n个缓冲区的有界缓冲问题

- 增加变量**counter**, 初始化为0
- 向缓冲区增加一项时, counter加1
- 从缓冲区移去一项时, counter减1

## ■ 数据结构

- Shared data

```
#define BUFFER_SIZE 8  
  
typedef struct {  
    ...  
} item;  
  
item buffer[BUFFER_SIZE];  
  
int in = 0;  int out = 0;  
  
int counter = 0;
```





# 有界缓冲区enter ()

## ■ 生产者进程

```
item nextProduced;  
  
while (1) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```





# 有界缓冲区remove ()

## ■ 消费者进程

```
item nextConsumed;  
  
while (1) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```





# 有界缓冲区

- 语句 “**counter++**”可按如下方式以**机器语言**实现：

(S0)**register1 = counter**

(S1)**register1 = register1 + 1**

(S2)**counter = register1**

- 语句“**counter--**”可按如下方式来实现：

(S3)**register2 = counter**

(S4)**register2 = register2 - 1**

(S5)**counter = register2**

- 如生产者和消费者试图并发地更新缓冲区，汇编语句可能交叉执行
- 交叉取决于生产者和消费者进程如何被调度





语句“**counter++**”可按如下方式以**机器语言**实现：

(S0)**register1 = counter**

(S1)**register1 = register1 + 1**

(S2)**counter = register1**

语句“**counter--**”可按如下方式来实现：

(S3)**register2 = counter**

(S4)**register2 = register2 - 1**

(S5)**counter = register2**

假如当前**counter=2**, 则存在多少种不同的执行结果?





# 有界缓冲区

■ 初时counter = 5:

S0: producer execute **register1 = counter** {register1 = 5}

S1: producer execute **register1 = register1+1** {register1 = 6}

S3: consumer execute **register2 = counter** {register2 = 5}

S4: consumer execute **register2 = register2-1** {register2 = 4}

S2: producer execute **counter = register1** {counter = 6 }

S5: consumer execute **counter = register2** {counter = 4}

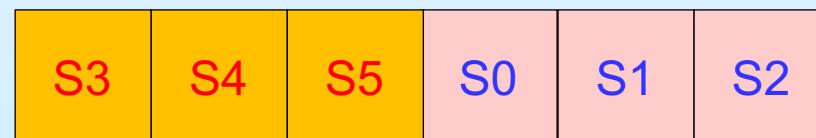




# 有界缓冲区

## ■ 解决方法：

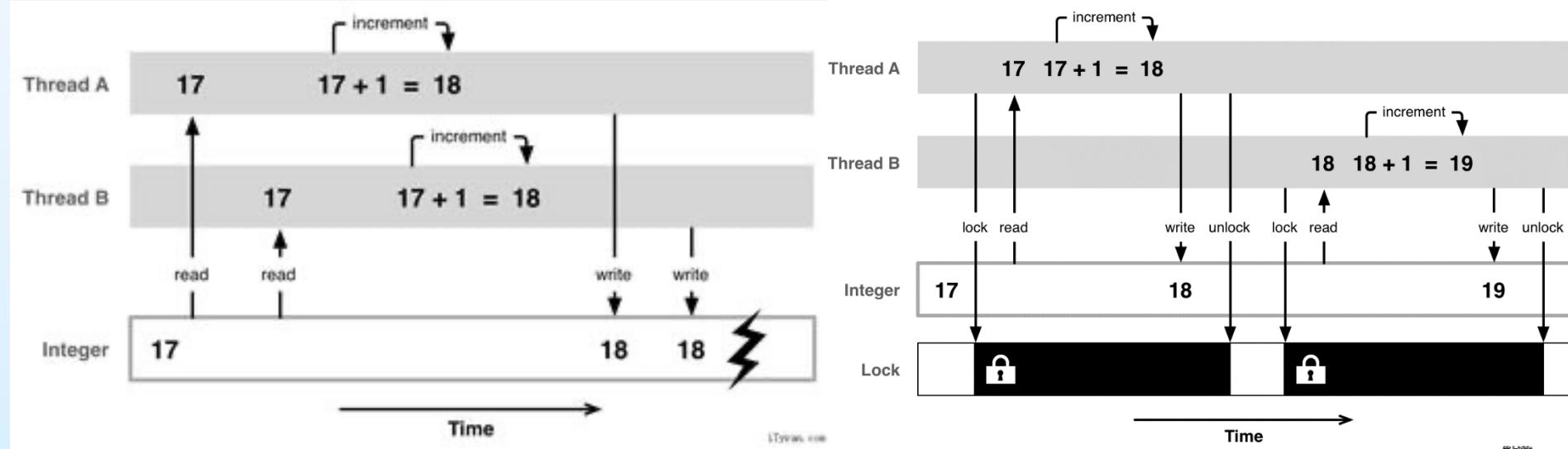
- 规定6个语句的运行次序，把counter++和counter—的语句分别作为一个整体来运行
- 也就是counter++和counter—的三个语句必须分别连续运行，不可中断
- 为此引入原子操作，一个操作在整个执行期间没有中断
- **counter++; counter--;** 包装为原子操作





# 竞争条件 (Race Condition)

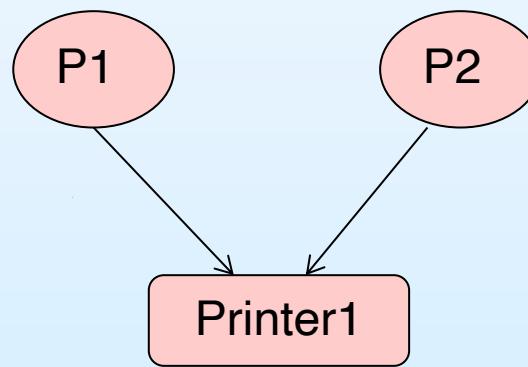
- 竞争条件: 多个进程并发访问和操作同一数据的情况。共享数据的最终结果取决于最后操作的进程
- 为了防止上述竞争条件, **并发进程同步或互斥**



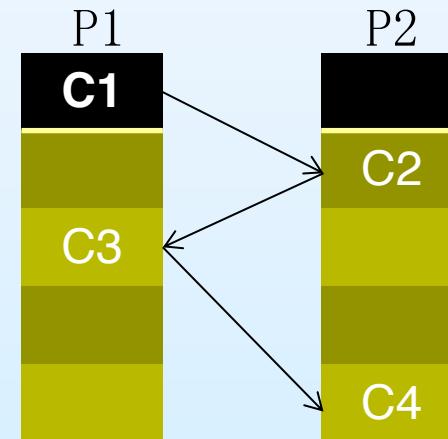


# 同步和互斥

- **同步:** 对多个相关进程在执行次序上进行协调，使并发执行的进程间能有效地共享资源和相互合作，使程序执行具有可再现性，保证数据一致性
- **互斥:** 进程排他性地运行某段代码，任何时候只有一个进程能够运行



访问独占资源-互斥



协调执行次序-同步



## 2、临界资源





# 临界资源和临界区

## ■ Critical resource (临界资源)

- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量
- 如counter（共享变量）和打印机（互斥资源）都是临界资源

## ■ 共享资源

- 一次允许多个进程使用的资源
- 如打开的文件供多个进程读取





■ 以下属于独占资源的是 ( )

- A. 磁盘
- B. 只读变量
- C. 网卡
- D. 打印机





# 临界区

## ■ Critical-Section (临界区)

- 涉及到临界资源的代码段叫临界区
  - ① 临界区是代码片段
  - ② 临界区是进程内的代码
  - ③ 每个进程有一个或多个临界区
  - ④ 临界区的设置方法由程序员确定
- 若能保证各个进程互斥进入具有相同临界资源的临界区，可实现对临界资源的互斥访问





```
item nextConsumed;
```

```
while (1) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;
```

临界资源

**counter--;**

临界区

```
}
```





# 解决临界区要求

## ■ 临界区使用准则

1. 互斥(Mutual Exclusion)准则：假定进程 $P_i$ 在其临界区内执行，其他任何进程将被排斥在自己的临界区之外
  - 有相同临界资源的临界区都需互斥
  - 无相同临界资源的临界区不需互斥
2. 有空让进(Progress)准则：临界区无进程执行，不能无限期地延长下一个要进入临界区进程的等待时间
3. 有限等待(Bounded Waiting)准则：每个进程进入临界区前的等待时间必须有限（不能无限等待）

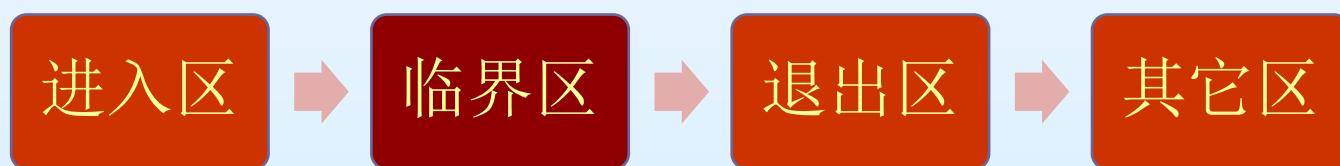




# 访问临界区的方法

## ■ 访问临界区过程

- 在进入区实现**互斥**准则
- 在退出区实现**有空让进**准则
- 每个临界区不能过大，从而实现**有限等待**准则





■ 实现有空让进准则可以在（ ）实现

- A. 进入区
- B. 退出区
- C. 临界区
- D. 其他区



### 3、软件和硬件解决方法





# 解决问题的初始尝试

- 只有2个进程,  $P_0$  和  $P_1$
- $P_i$ 进程的通用结构

```
do {  
    [进入区]  
    critical section  
    [退出区]  
    reminder section  
} while (1);
```

- 进程可以共享一些公共变量来同步它们的行为





# 算法1-轮流进临界区

## ■ 共享变量:

- **int turn;**  
初始 **turn = 0**
- **turn == i**  $\Rightarrow P_i$  能进入临界区
- **j = 1-i**

## ■ 进程 $P_i$

```
do {
    while (turn != i) ;
        临界区
    turn = j;
        剩余区
} while (1);
```

## ■ 满足互斥条件，但不满足有空让进





# 算法2—竞争进入临界区

## ■ 共享变量

- **boolean flag[2];**  
初始 **flag [0] = flag [1] = false.**
- **flag [i] = true**  $\Rightarrow P_i$  准备进入临界区

## ■ 进程 $P_i$

```
do {  
    flag[i] := true;  
    while (flag[j]) ;  
        critical section  
    flag [i] = false;  
        remainder section  
} while (1);
```

## ■ 满足互斥，但不满足有空让进需求，存在死锁





# 算法3- Peterson算法

- 合并算法 1和 2的共享变量
- 进程  $P_i$

```
do {  
    flag [i]:= true;  
    turn = j;  
    while (flag [j] and turn = j) ;  
        critical section  
    flag [i] = false;  
        remainder section  
} while (1);
```

- 满足三个需求；解决了两个进程的临界区问题
- 思考：为什么？





# 硬件同步

- 许多系统采用硬件同步机制来处理临界区
- 基于锁的解决方法
  - 利用锁来保护临界区
- 单处理器：禁止中断
  - 当前运行代码不被中断
  - 不适合多处理器
- 现代操作系统：特殊硬件指令
  - 原子操作（不可中断操作）

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```





# TestAndSet

- 原子地检查和修改字的内容.

```
boolean TestAndSet(boolean &target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```



# Test-and-Set指令实现互斥

- 共享数据:

```
boolean lock = false;
```

- 进程  $P_i$

```
do {
```

```
    while (TestAndSet(lock)) ;
```

critical section

```
    lock = false;
```

remainder section

```
}
```



## 4、信号量





# 内容

- 信号量概念
- 整型信号量
- 记录型信号量
- 同步信号量和互斥信号量
- 信号量使用



# 信号量 (Semaphore)

- 早期采用硬件解决方法，对程序设计人员而言太复杂
- 信号量-软件解决方案
  - 保证两个或多个代码段不被并发调用
  - 在进入关键代码段前，进程必须获取一个信号量，否则不能运行
  - 执行完该关键代码段，必须释放信号量
  - 信号量有值，为正说明它空闲，为负说明其忙碌





# 迪科斯彻(1930-2002)

## ■ 艾兹格·W·迪科斯彻(Edsger Wybe Dijkstra)

- 信号量和**PV**原语发明者—**第六章**
- 解决了“哲学家就餐”问题—**第六章**
- 最短路径算法(**SPF**)和银行家算法的创造者-**第七章**
- 结构程序设计之父
- **THE**操作系统设计者和开发者—**第二章**

## ■ 与D. E. Knuth并称为这个时代最伟大的计算机科学家





# 信号量Semaphore

- 整型信号量  $S$  – 整型变量，大于0表示可以获得信号量，小于等于0表示无法获得信号量
- 提供两个不可分割的[原子操作]访问信号量，wait和signal
  - wait( $S$ ): P( $S$ ): 表示要获得一个信号量。
  - signal( $S$ ): V( $S$ )
  - 如  $S > 0$ : 该进程可以获得一个  $S$  信号量，继续运行；否则  $S \leq 0$ : 无法获得信号量，则无法运行下去

```
wait(S): while S ≤ 0 do no-op;  
          S--;
```

```
signal(S): S++; //释放信号量
```

- 整型信号量的问题：忙等





进程调用**wait (s)** 表示要获得一个信号量。如果s大于等于0，则该进程可以获得一个s信号量，继续运行。

- A. 对
- B. 错





# 记录型信号量：去除忙等的信号量

```
Wait(semaphore *S)
```

```
{  
    S->value--;  
    if (S->value < 0) {  
        add this process to list S->list;  
        block();  
    }  
}
```

记录型信号量定义：

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

```
Signal(semaphore *S) {
```

```
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from list S->list;  
        wakeup(P);  
    }  
}
```





# 记录型信号量

- 记录型信号量是先把信号量的值减1后再判断，而整型信号量时先判断再减1
- 目的是可以知道由于申请该信号量而阻塞的进程的数量
- $S$ 是一个负数时， $|S|$ 表示 $S$ 的等待队列中等待该信号量的进程数目
- 记录型信号量的改进在于通过加入了阻塞和唤醒机制，消除了忙等





# 两种类型信号量

- 计数信号量 – 变化范围没有限制的整型值，又称为同步信号量
- 二值信号量 – 变化范围仅限于0和1的信号量；容易实现，又称为互斥信号量
- 信号量S的使用
  - S必须置一次且只能置一次初值
  - S初值不能为负数
  - 除了初始化，只能通过**执行P、V操作来访问S**





判断：信号量S必须置一次且只能置一次初值，一般情况下，计数信号量的初值是一个整数，二值信号量的值为0。





# 互斥信号量

① **Semaphore \*S;** // 初始化为 1

① **wait(S);** //申请信号量

**CriticalSection()** //临界区

③ **signal(S);** //释放信号量





# 同步信号量

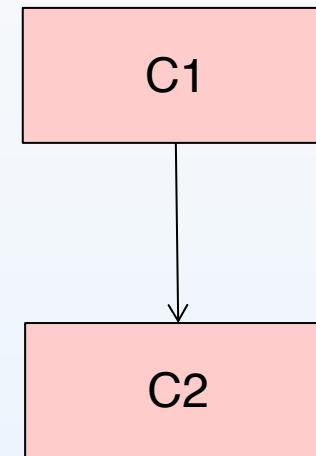
- 实现各种同步问题，位于两个不同进程
- 例子： $P_1$  和  $P_2$  需要  $C_1$  比  $C_2$  先运行  
`semaphore synch=0`

$P_1$ :

```
C1;  
signal(synch);
```

$P_2$ :

```
wait(synch);  
C2;
```





判断：互斥信号量用于临界区时，`wait`和`signal`操作一般位于同一个进程中，在临界区前执行`wait`操作。





# 死锁和饥饿

- 死锁 – 两个或多个进程无限期地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.
- S和Q是两个初值为1的信号量

$P_0$	$P_1$
$P(S);$	$P(Q);$
$P(Q);$	$P(S);$
$\vdots$	$\vdots$
$V(S);$	$V(Q);$
$V(Q)$	$V(S);$

- 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.



## 5、经典同步问题





# 经典同步问题

## ■ 生产者-消费者问题

- 共享有限缓冲区

## ■ 读者写者问题

- 数据读写操作

## ■ 哲学家就餐问题

- 资源竞争





# 单缓存生产者-消费者解决方案

P:

Repeat

    生产一个产品；

    wait(empty);

    送产品到缓冲区；

    signal(full);

Until false

C:

Repeat

    wait(full);

    从缓冲区中取产品；

    signal(empty);

    消费产品；

Until false

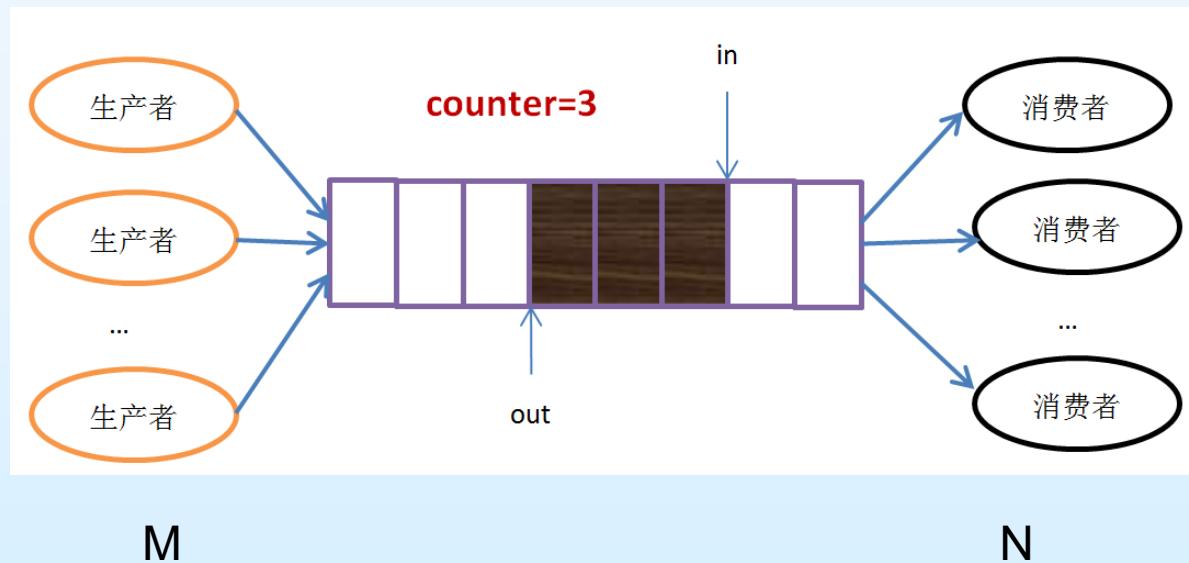
信号量empty初值为1，full初值为0





# 问题描述

- 生产者 ( $M$ 个)：生产产品，并放入缓冲区
- 消费者 ( $N$ 个)：从缓冲区取产品消费
- 问题本质：如何实现生产者和消费者之间的同步和互斥





# 生产者消费者流程

生产者:

{

...  
生产一个产品  
...

...  
把产品放入指定缓冲区  
...

}

消费者:

{

...  
...  
从指定缓冲区取出产品  
...

...  
...  
消费取出的产品  
...

}





# 互斥分析基本方法

查找临界资源



划分临界区



定义互斥信号量并赋初值



在临界区前的进入区加wait操作；退出区  
加signal操作





# 生产者消费者的互斥分析

## ■ 临界资源

### ● 生产者

✓ 把产品放入指定缓冲区

✓ **in**:所有的生产者对in指针需要互斥

✓ **counter**: 所有生产者消费者进程对counter互斥

### ● 消费者

✓ 从指定缓冲区取出产品

✓ **out**:所有的消费者对out指针需要互斥

✓ **counter**: 所有生产者消费者进程对counter互斥

```
buffer[in] = nextProduced;  
in = (in + 1) % BUFFER_SIZE;  
counter++;
```

```
nextConsumed = buffer[out];  
out = (out + 1) % BUFFER_SIZE;  
counter--;
```



# 划分临界区

生产者:

{

...

生产一个产品

...

临界区

把产品放入指定缓冲区

...

}

消费者:

{

...

从指定缓冲区取出产品

...

消费取出的产品

...

}

临界区





- 多个生产者由于共享变量**in**需要互斥访问生产者的临界区
- 多个消费者由于共享变量**out**需要互斥访问消费者的临界区
- 生产者和消费者由于共享变量**counter**需要互斥访问生产者和消费者的临界区





# 增加互斥机制

```
semaphore m;    m= 1;
```

生产者：

```
{
```

...

生产一个产品

...

**wait(m);**

临界区

...

把产品放入指定缓冲区

...

**signal(m);**

```
}
```

消费者：

```
{
```

...

**wait(m);**

临界区

从指定缓冲区取出产品

...

**signal(m);**

...

消费取出的产品

...

```
}
```





# 同步分析

找出需要同步的代码片段（关键代码）



分析这些代码片段的执行次序



增加同步信号量并赋初始值



在关键代码前后加wait和signal操作

同步分析较为困难





# 生产者消费者的同步分析

## ■ 两者需要协同的部分

- 生产者：把产品放入指定缓冲区（关键代码C1）
- 消费者：从满缓冲区取出一个产品（关键代码C2）

## ■ 三种运行次序（不同条件下不同运行次序）

- 所有缓冲区空时：



- 所有缓冲区满时：



- 缓冲区有空也有满时：





# 算法描述：生产者

## ■ 生产者

...

生产一个产品

...

1) 判断是否能获得一个空缓冲区，如果不能则阻塞

**C1:**把产品放入指定缓冲区

临界区

2) 满缓冲区数量加1，如果有消费者由于等消费产品而被阻塞，则唤醒该消费者

同步：通知





# 算法描述：消费者

## ■ 消费者

同步：判断

1) 判断是否能获得一个满缓冲区，如果不能则阻塞

C2:从满缓冲取出一个产品

临界区

2) 空缓冲区数量加1，如果有生产者由于等空缓冲区而阻塞，则唤醒该生产者

同步：通知





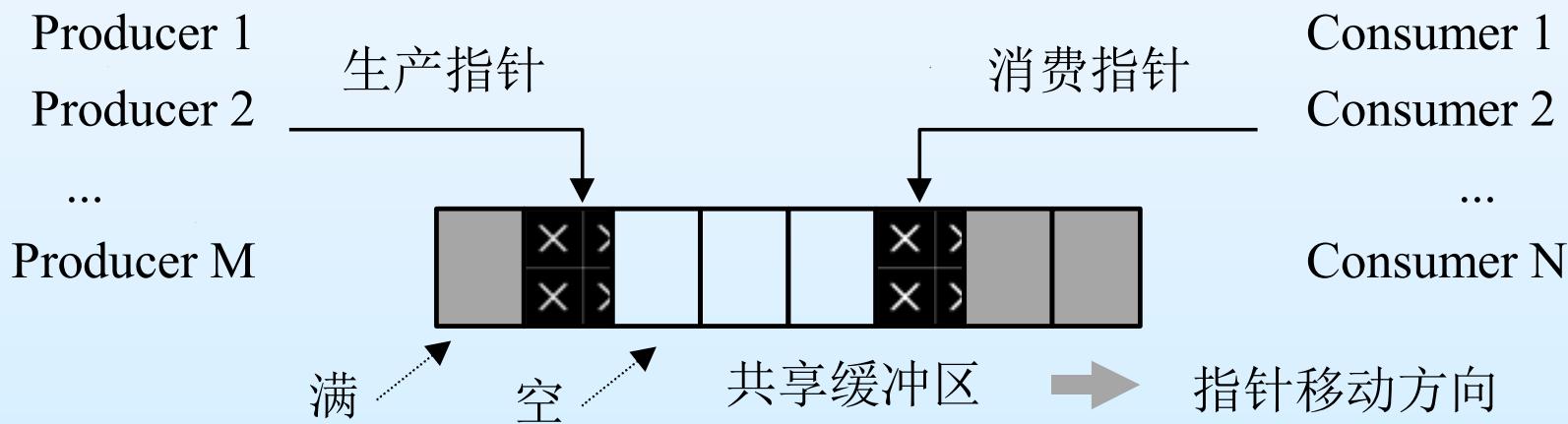
# 同步信号量定义

## ■ 共享数据

**semaphore full, empty, m;** //full:满缓冲区数量 empty:空缓冲区数量

初始化:

**full = 0;**      **empty = N;**    **m= 1;**





# 解决方法

生产者:

{

...  
生产一个产品

...

**wait(empty);**

wait(m);

...

**C1:** 把产品放入指定缓冲区

...

signal(m);

**signal(full);**

}

当empty大于0时，  
表示有空缓冲区，  
继续执行；否则，  
表示无空缓冲区，  
当前生产者阻塞。

消费者:

{

...  
**wait(full);**

wait(m);

...

**C2:** 从指定缓冲区取出产品

...

signal(m);

**signal(empty);**

...

消费取出的产品

...

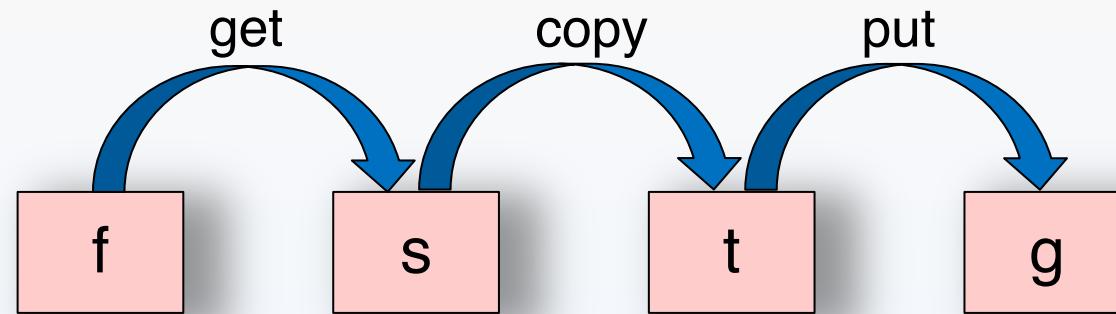
}

当full大于0时，表  
示有满缓冲区，继  
续执行；否则，表  
示无满缓冲区，当  
前消费者阻塞。

把full值加1，如  
果有消费者等在full的  
队列上，则唤醒该  
消费者。

把empty值加1，如  
果有生产者等在empty的队列上，  
则唤醒该生产者。





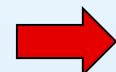
把copy进程分为2个：

fulls=0,emptyS=1,fullt=0,emptyt=1;

get和copy1:

wait(emptyS)  
get()  
signal(fulls)

wait(fulls)  
copy1()  
signal(emptyS)



copy2和put:

wait(emptyt)  
copy2()  
signal(fullt)

把copy进程重新拼接起来：

fulls=0,emptyS=1,fullt=0,emptyt=1;

Get进程:

wait(emptyS)  
get()  
signal(fulls)

Copy进程:

wait(emptyt)  
wait(fulls)  
copy()  
signal(fullt)  
signal(emptyS)

put进程:

wait(fullt)  
put()  
signal(emptyt)





# 读者写者问题

- 问题描述
- 同步互斥分析
- 解决方法





# 读者写者问题

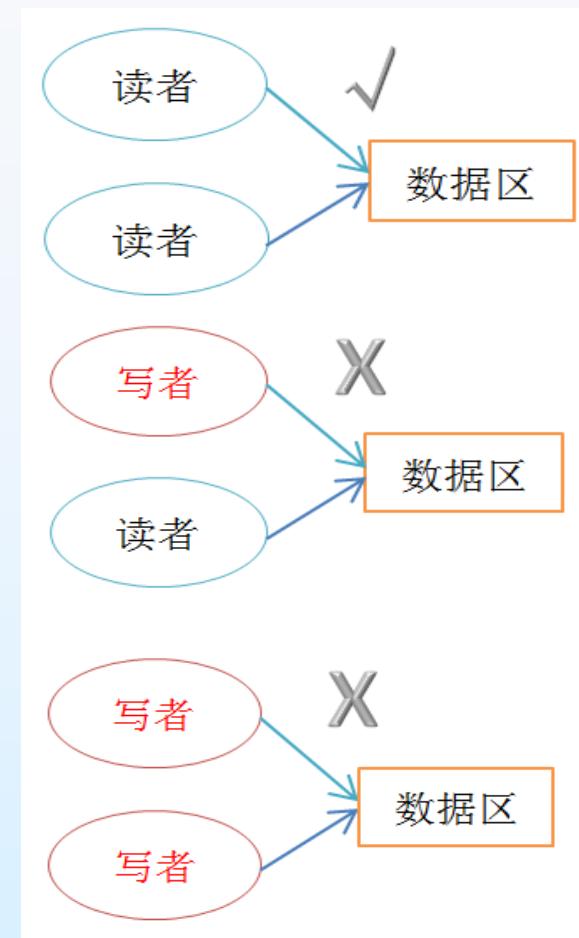
## ■ 两组并发进程

- 读者和写者
- 共享一组数据区进行读写

## ■ 读写操作要求

- 允许多个读者同时读
- 不允许读者、写者同时读写
- 不允许多个写者同时写

## ■ 例子：对文件的读写操作





- 写者1：写入“苹果5元/斤”
- 写者2：写入“樱桃30元/斤”





# 读者优先

如果读者来：

- 1 ) 无读者、写者，新读者可以读，并阻止写者对数据区的写。
- 2 ) 有写者等，但有其它读者正在读，则新读者也可以读，体现了读者优先。
- 3 ) 有写者写，新读者必须等

如果写者来：

- 1 ) 无读者，新写者可以写，并阻止其他的读者和写者。
- 2 ) 有读者，新写者等待。
- 3 ) 有其它写者，新写者等待。





# 解决方法

Semaphore \*W; W->value=1;

## ■ Readers

.....

**wait(W);**

读

**signal(W);**

.....

## ■ Writers

.....

**wait(W);**

写

**signal(W);**

.....

这种互斥模式实现了写者之间，读者和写者之间的互斥，符合读者优先的要求。

但是，这种模式要求读者之间也要互斥，违背了“有写者在等，但有其他读者在读时，则新读者可以进入数据读”的要求。当一个读者获得信号量W进入数据进行读操作时，后续读者无法继续进入数据区读取，不能实现共享读。





# 修改思路

❖ Readers

第一个读者

.....

**wait(W);**

❖ Writers

.....

**wait(W);**

读

写

最后离开的读者

**signal(W);**

**signal(W);**

.....

.....





# 修改方法

增加一个读者计数器rc，设置初始值为0：

## ❖ Readers

.....

**rc++;**

**if (rc==1) wait(W);**

读

第一个读者

**rc--;**

**if (rc==0) signal(W);**

.....

## ❖ Writers

.....

**wait(W);**

写

**signal(W);**

.....





# 修改方法

再增加一个互斥信号量M，设置初始值为1；

## ❖ Readers

.....

**wait(M);**

rc++;

if (rc==1) wait(W);

**signal(M);**

读

**wait(M);**

rc--;

If (rc==0) signal(W);

**signal(M);**

.....

临界区

## ❖ Writers

.....

**wait(W);**

写

**signal(W);**

.....





判断：

- 1、在读者写者问题中，不允许多个读者同时读，也不允许读者、写者同时读写。
- 2、所有读者只有在执行**wait(M)**操作时才有可能被阻塞。





# 思考

第二类读者写者问题：写者优先

条件：

- 1 ) 多个读者可以同时进行读
- 2 ) 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 3 ) 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

■ 如何用PV操作实现？





# 哲学家就餐问题

- 哲学家就餐问题描述
- 存在死锁的解决方法
- 两种无死锁的解决方法
- 对信号量机制的总结





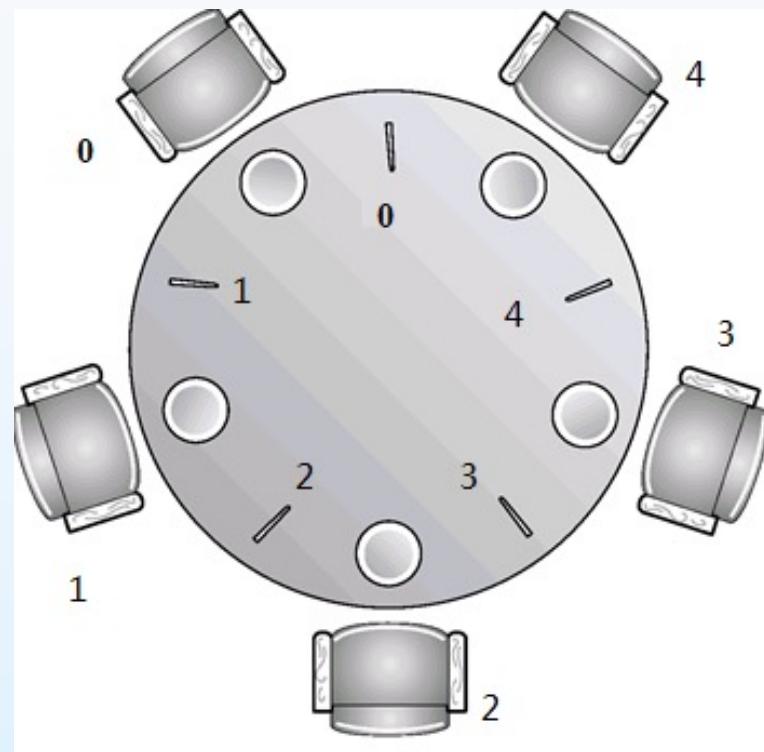
# 哲学家就餐问题

## ■ 问题描述

- 5个哲学家
- 5根筷子
- 每个哲学家左右各有一根筷子
- 每个哲学家只有拿起左右两个筷子才能吃饭

## ■ 多个进程共享资源竞争的问题

- 把5根筷子看作5个互斥信号量，任意一个哲学家只有拿起左右两根筷子，也就是获得左右两个信号量才能吃饭
- 吃完饭，这些就应该放下左右两根筷子，也就是释放左右两个信号量





# 解决方法

■ semaphore \*chopStick[5]; //初始值为1

■ 哲学家  $i$ :

.....

```
wait(chopStick[i]);          //拿左边筷子  
wait(chopStick[(i + 1) % 5]); //拿右边筷子
```

吃饭

```
signal(chopStick[i]);        //放下左边筷子  
signal(chopStick[(i + 1) % 5]); //放下右边筷子
```

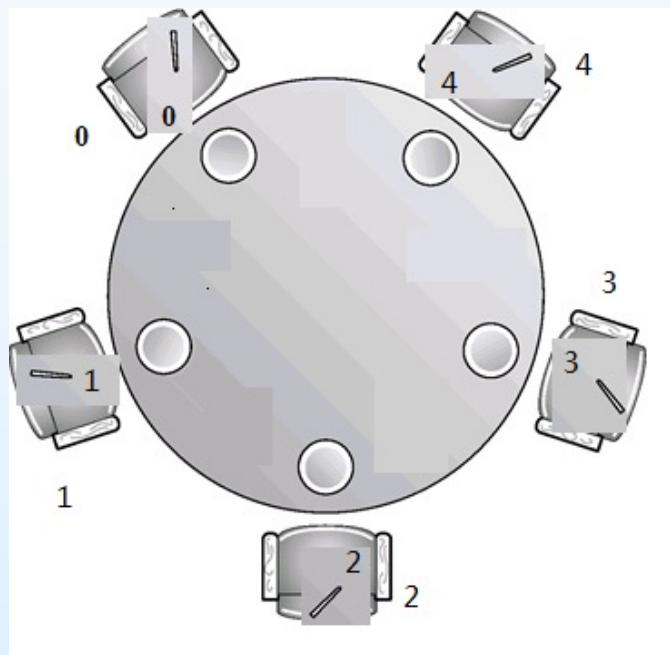
.....





# 存在问题：死锁

- 每个哲学家同时执行**wait(chopStick[i])**，拿起左边筷子，导致死锁



这时，5个哲学家对筷子存在循环等待，从而导致他们都无法吃饭，形成了死锁

这种死锁导致进程无法推进、资源无法使用，是必须解决的。

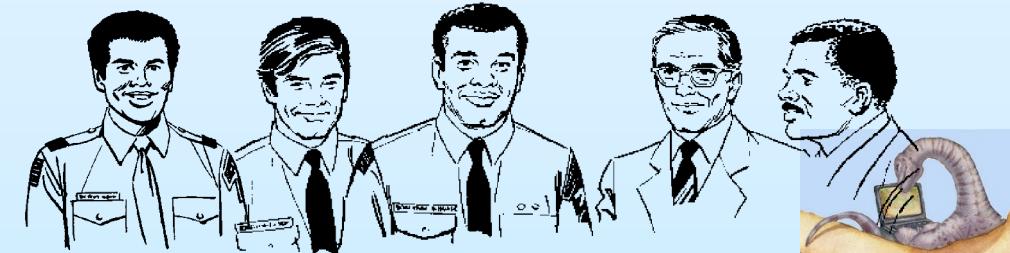
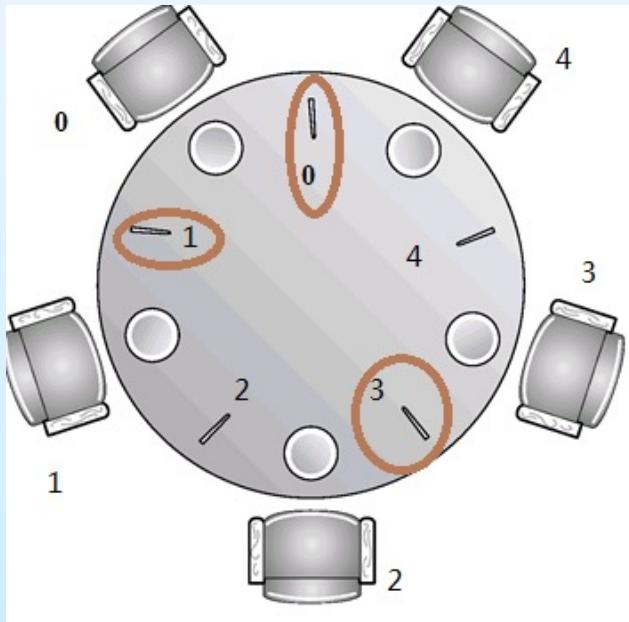




# 哲学家就餐问题

为防止死锁发生可采取的措施：

- 方法1：最多允许4个哲学家同时坐在桌子周围（必然有一个人可以拿起左右两根筷子）
- 方法2：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（筷子要一起拿）
- 方法3：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





# 方法1-最多4个哲学家入座

- **semaphore \*chopStick[5];** //初始值为1
- **semaphore \*seat;** //初始值为4
  
- 哲学家 *i*:
  - .....
  - wait(seat);** //看看4个座位是否有空
  - wait (chopStick[i]);** //拿左边筷子
  - wait (chopStick[(i + 1) % 5]);** //拿右边筷子
  - 吃饭
  - signal(chopStick[i]);** //放下左边筷子
  - signal (chopStick[(i + 1) % 5]);** //放下右边筷子
  - signal (seat);** //释放占据的位置
  - .....





## 方法2 – 同时拿筷子

- 两根筷子都空闲，则该哲学家可以拿起两根筷子吃饭
- 否则，只要有一根筷子在被其他哲学家使用，那么两根筷子都无法拿到
- 为了避免死锁，所以把哲学家分为三种状态，思考，饥饿，进食，并且一次拿到两只筷子，否则不拿。
- 哲学家分为3个状态：
  - int \*state={Thinking, hungry, eating};
- 设置5个信号量，对应5个哲学家
  - semaphore \*ph[5]; //初始值为0





## 方法2-同时拿筷子

```
void test(int i);  
{  
    if (state[i] == hungry) &&      //是否饿了  
        (state[(i+4)%5]!=eating) && //左边哲学家是否在吃饭  
        (state[(i+1)%5]!=eating)    //右边哲学家是否在吃饭  
    {  
        state[i]=eating;          //设置哲学家状态为eating  
        signal(ph[i]);            //ph[i]设置为1  
    }  
}
```





## 方法2-同时拿筷子

哲学家i: 0 ~ 4

思考中.....

state[i]=hungry ;

**wait(m);**

test(i);

**signal(m);**

wait(ph[i]);

拿起左边筷子

拿起右边筷子

吃饭.....

放下左边筷子

放下右边筷子

state[i]=thinking;

test((i+4)%5);

test((i+1)%5);

.....

**semaphore \*m; //初始值为1**





# 信号量S和PV操作的讨论

## ■ 理解信号量的物理含义

- $S > 0$ ，表示有 $S$ 个资源可用
- $S = 0$ ，表示无资源可用
- $S < 0$ ，则  $|S|$  表示 $S$ 队列中的等待进程个数

## ■ 理解wait和signal这两个原子操作

- $\text{wait}(S)$ ，表示申请一个资源
- $\text{signal}(S)$ ，表示释放一个资源

## ■ 注意信号量初值

- 互斥信号量初值一般为1
- 同步信号量初值一般为0-N的整数





# 信号量S和PV操作的讨论

## ■ 信号量使用中注意的问题

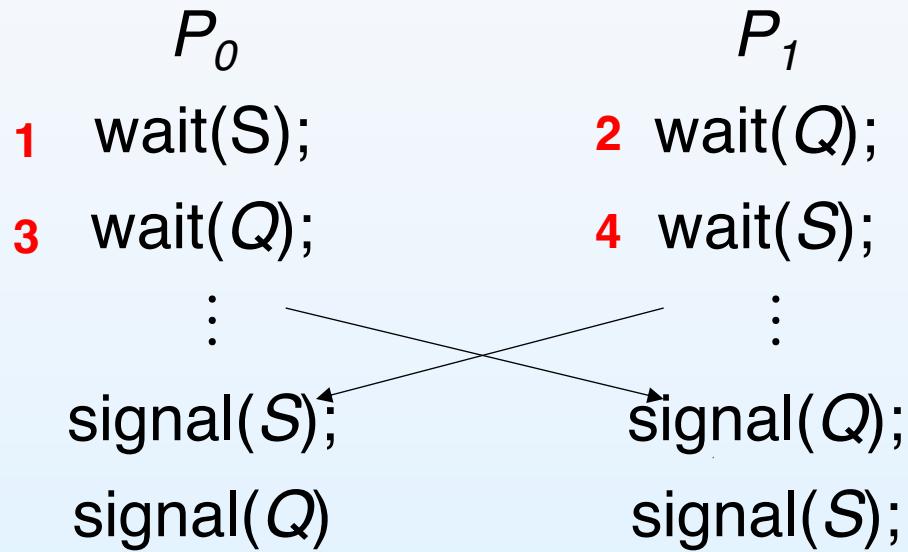
- P.V操作必须成对出现，有一个P操作就一定有一个V操作
- 当为互斥操作时，它们同处于同一进程
- 当为同步操作时，则不在同一进程中出现
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要,如果使用不当会造成死锁
- 同步和互斥P操作在一起时，同步P操作在互斥P操作前，而两个V操作无关紧要





# 死锁：信号量使用不当

- S和Q是两个初值为1的二值信号量



- 死锁

- 两个或多个进程因等待对方资源而无限等待的情况，如果没有外力干预，则进程将永远无法运行下去





# 同步问题例子1

## 用P.V操作解决司机与售票员的问题

**司机进程：**

REPEAT

启动车辆

正常驾驶

到站停车

UNTIL ...

**售票员进程：**

REPEAT

关门

售票

开门

UNTIL ...





## 同步问题例子2

■ 桌上有一空盘，允许存放一个水果。爸爸可向盘中放苹果 (Apple)，也可向盘中放橙子(Orange)，儿子专等吃盘中的橙子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一个水果供儿子或女儿取用，请用P、V原语实现爸爸、儿子、女儿3个并发进程的同步。

■ 分析：

- 爸爸、儿子、女儿共用一个盘子，且盘中一次只能放一个水果
- 当盘子为空时，爸爸可将一个水果放入盘中。若放入盘中的是橙子，则允许儿子吃，女儿必须等待；若放入盘中的是苹果，则允许女儿吃，儿子必须等待。
- 生产者——消费者问题的一种变形。这里，生产者放入缓冲区的产品有两类，消费者也有两类，每类消费者只消费其中固定的一类产品。





# 同步描述

```
int S=1;
int SA=0;
int SO=0;
main()
{
    cobegin
        father();
        son();
        daughter();
    coend
}
```

```
    father()
    {
        while(1)
        {
            P(S); // 盘子是否空
            将水果放入盘中;
            if(放入的是橙子)
                V(SO);
            else
                V(SA)
        }
    }
}
```





## 同步描述

```
son()
{
    while(1)
    {
        P(SO); // 盘子中有无橙子
        从盘中取出橙子;
        V(S);
        吃橙子;
    }
}
```

```
daughter()
{
    while(1)
    {
        P(SA); // 盘子中有无苹果
        从盘中取出苹果;
        V(S);
        吃苹果;
    }
}
```





## 同步问题例子3

- 桌上有一空盘，允许存放一个水果。爸爸可向盘中放苹果（Apple）。儿子和女儿各吃一半(不能一人 吃全部)。请用P、V原语实现爸爸、儿子、女儿3个并发进程的同步。





## 同步问题例子4

- 桌上有一空盘，允许存放一个水果。爸爸可向盘中放苹果(**Apple**)，妈妈可向盘中放橙子(**Orange**)，儿子专等吃盘中的橙子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一个水果供儿子或女儿取用，请用P、V原语实现爸爸、妈妈、儿子、女儿4个并发进程的同步。





## 同步问题例子5

- 桌上有一空盘，允许存放2个不同水果（不允许存放2个相同水果）。爸爸可向盘中放苹果(**Apple**)，妈妈可向盘中放橙子(**Orange**)。等盘子满后，儿子吃盘中的半个橙子和半个苹果，女儿吃盘中的半个橙子和半个苹果。不允许儿子或女儿一人吃掉全部水果。用P、V原语实现爸爸、妈妈、儿子、女儿4个并发进程的同步。



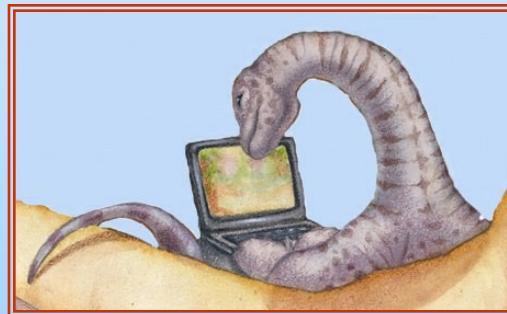


# 信号量同步的缺点

- 同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）
- 易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；
- 不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；
- 正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误；



# 6、管程





# 内容

- 信号量机制的问题
- 管程的概念
- 引入条件变量的管程
- 管程的实现方法
- 哲学家就餐问题的例子
- 常用操作系统的同步机制



# 信号量机制的问题

## ■ 优点

- 程序效率高、编程灵活

## ■ 问题

- 需要程序员实现，编程困难
- 维护困难、容易出错

✓ wait/signal位置错

✓ wait/signal不配对

信号量：分散式  
管 程：集中式

## ■ 解决方法

- 管程（1970s, Hoare和Hansen）
- 由**编程语言**解决同步互斥问题，而不是程序员





# 管程Monitors

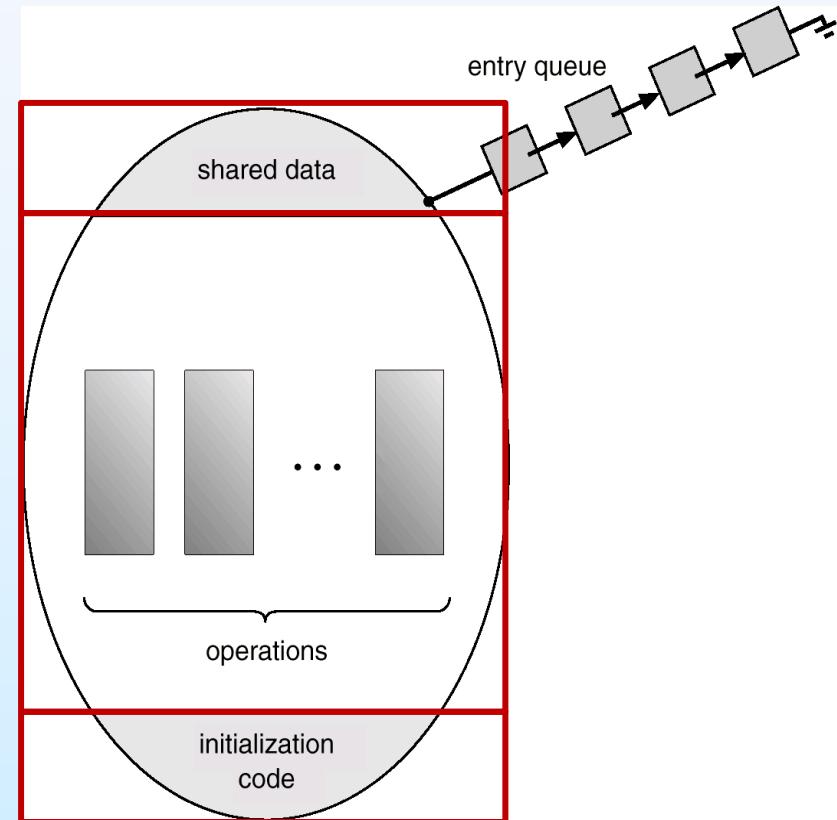
## Hansen的管程定义

一个管程定义了一个**数据结构**和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据

- 高级同步构建类型
- 管程是对提供线程安全机制的高度抽象
- 任一时刻在管程中只有一个线程能运行

```
monitor monitor-name
{
    // variable declarations
    public entry p1(...) {...}
    public entry p2(...) {...}

    ...
    Initialization_code(...){...}
}
```





# 管程功能

## ■ 互斥

- 管程中的变量只能被管程中的操作访问
- 任何时候只有一个进程在管程中操作
- 类似临界区
- 由编译器完成

## ■ 同步

- 条件变量
- 唤醒和阻塞操作



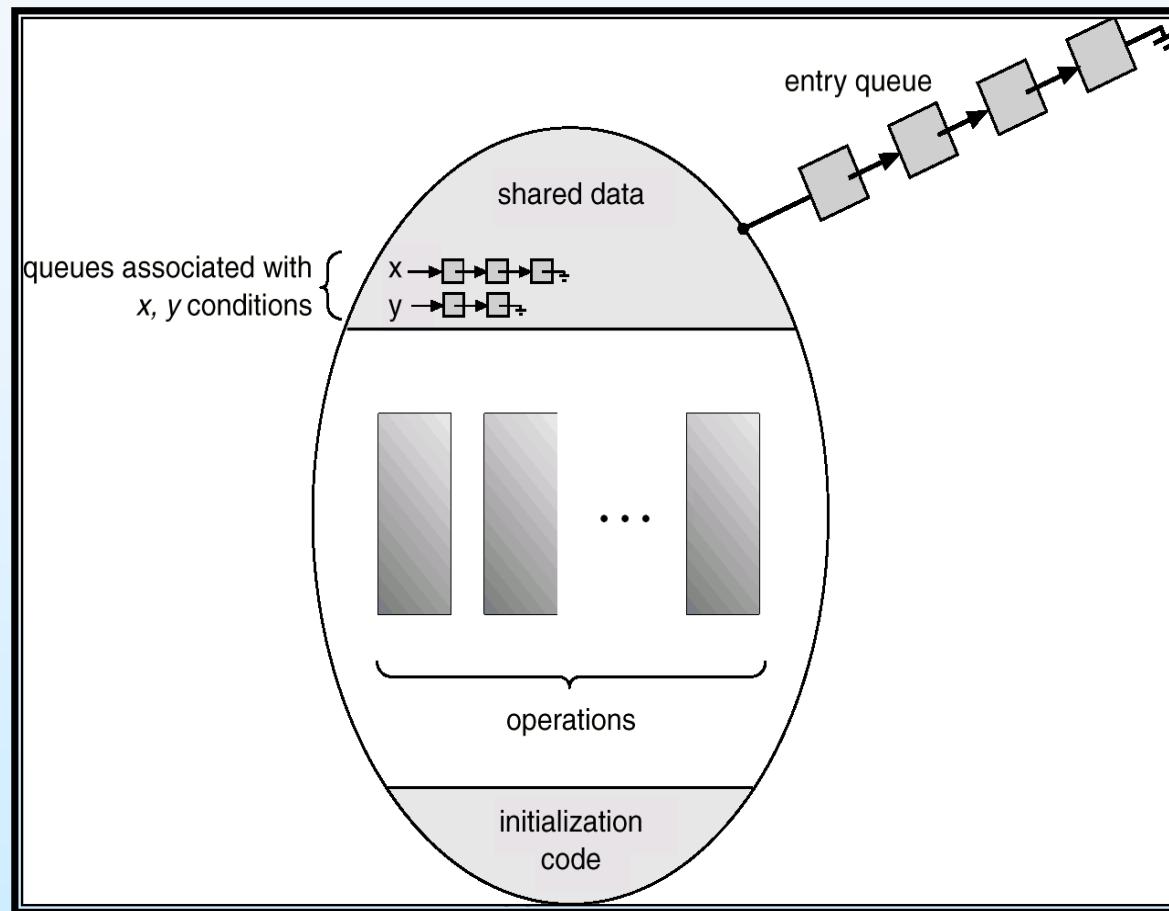


# 条件变量

- **condition x, y;**

- 条件变量的操作

- `x.wait()`: 进程阻塞直到另外一个进程调用`x.signal()`
- `x.signal()`: 唤醒另外一个进程





- 判断：在管程中加入条件变量，以及基于条件变量的阻塞和唤醒操作可以为进程提供同步机制。





# 条件变量问题

- 管程内可能存在不止1个进程
  - 如：进程P调用signal操作唤醒进程Q后，此时管程内有P和Q两个进程
- 存在的可能
  - P等待直到Q离开管程（Hoare）
  - Q等待直到P离开管程（Lampson & RedII, MESA语言）
  - P的signal操作是P在管程内的最后一个语句，然后Q开始运行（Hansen，并行Pascal）



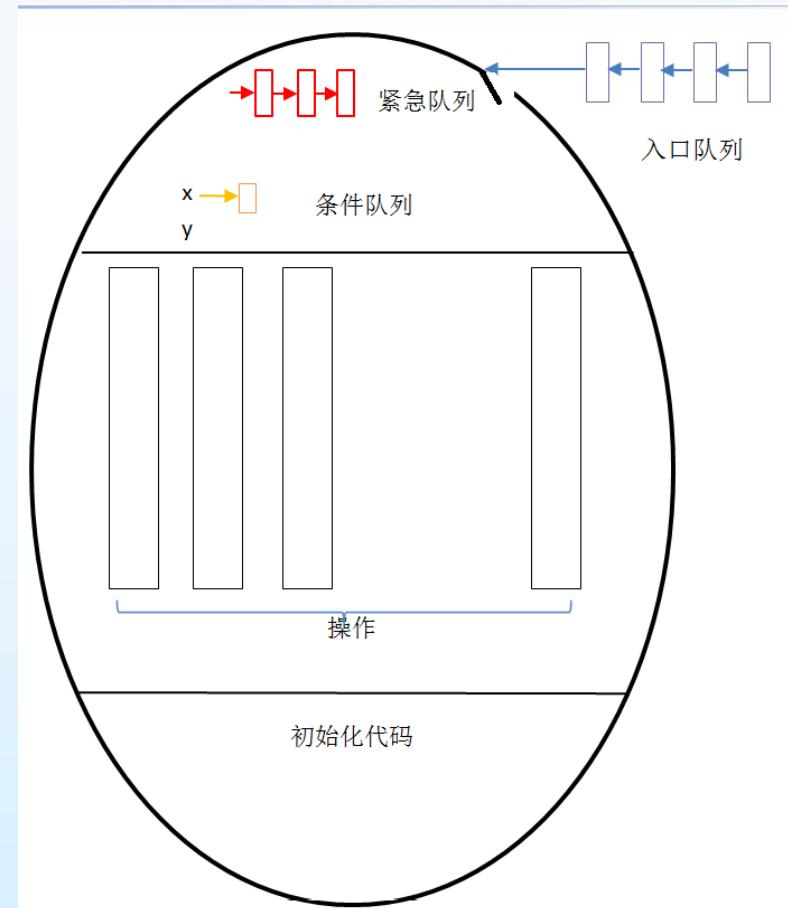
# Hoare管程

## ■ 进程互斥进入管程

- 如果有进程在管程内运行，管程外的进程等待
- **入口队列**：等待进入管程的进程队列

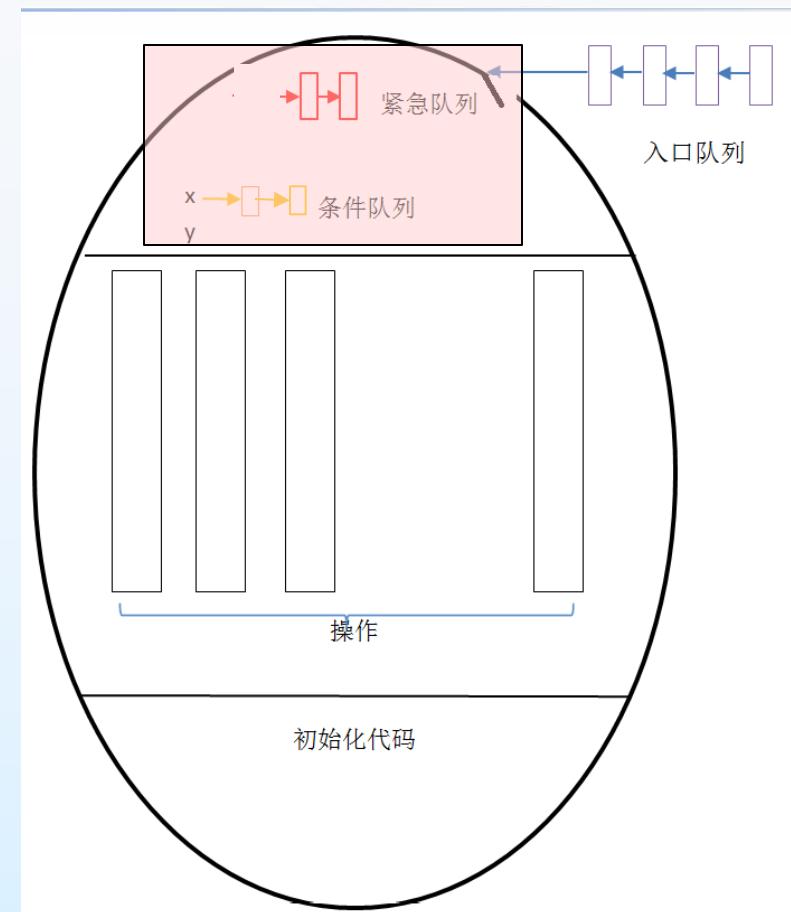
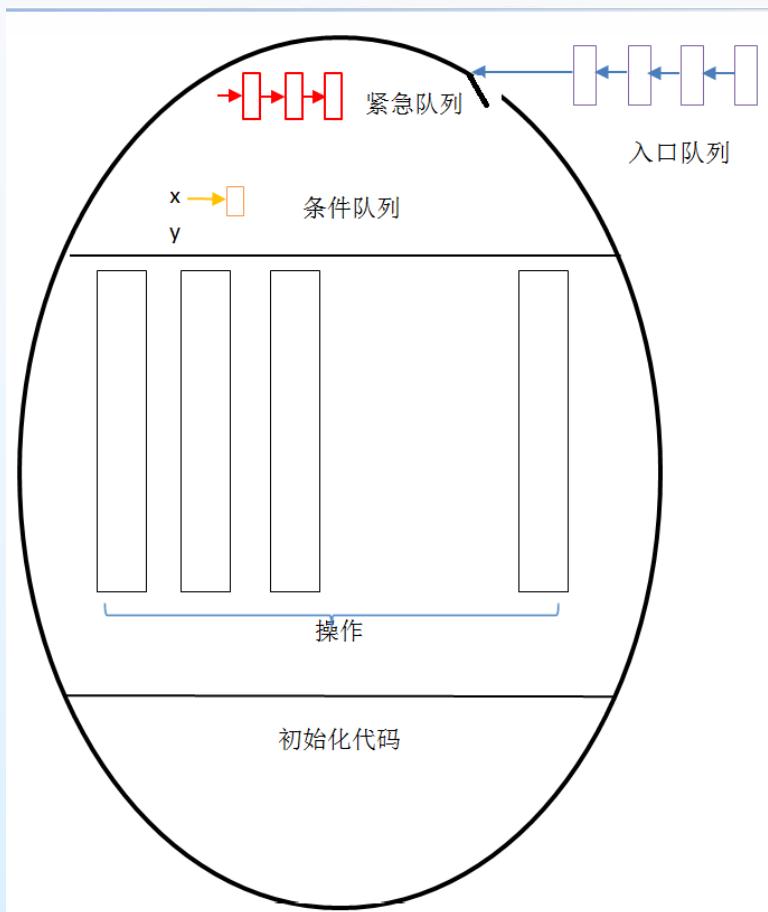
## ■ 管程内进程P唤醒Q后

- P等待，Q运行
- P加入**紧急队列**
- 紧急队列的优先级高于入口队列



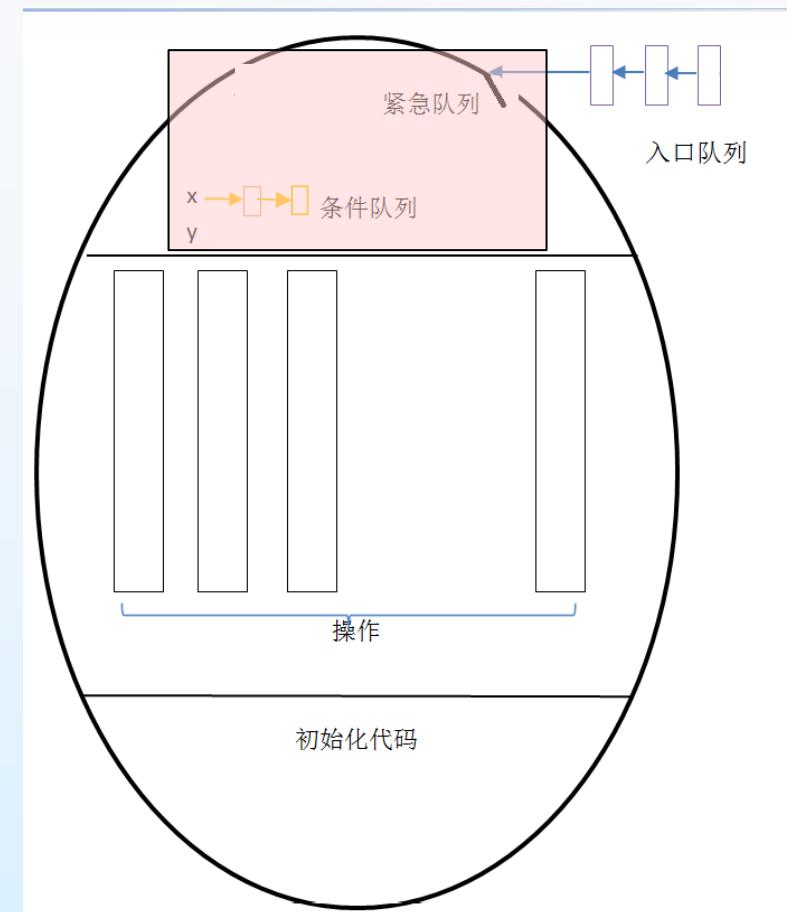
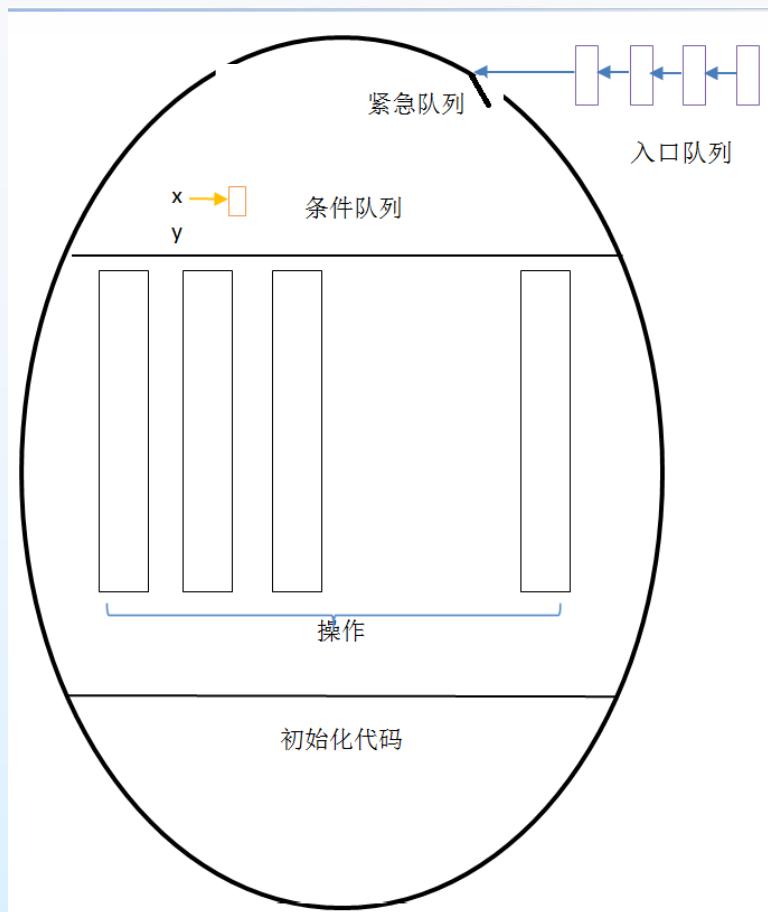


# 紧急队列非空





# 紧急队列空





# Hoare管程

## ■ condition x;

### ■ x.wait()

- 紧急队列非空：唤醒第一个等待进程
- 紧急队列空：释放管程控制权，允许入口队列进程进入管程
- 执行该操作进程进入x的条件队列

### ■ x.signal()

- x的条件队列空：空操作，执行该操作进程继续运行
- x的条件队列非空：唤醒该条件队列的第一个等待进程，执行该操作进程进入就绪队列





# 哲学家就餐Hoare管程解决方案

monitor DP

```
{  
    enum { THINKING; HUNGRY, EATING } state [5] ;  
    condition self [5];  
  
    initialization_code() {  
        for (int i = 0; i < 5; i++)  
            state[i] = THINKING;  
    }  
}
```

共享变量

```
void pickup (int i) {  
    state[i] = HUNGRY;  
    test(i);  
    if (state[i] != EATING)  
        self[i].wait();  
}
```

初始化

```
void putdown (int i) {  
    state[i] = THINKING;  
    // test left and right neighbors  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != EATING)  
        &&(state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[i] = EATING ;  
        self[i].signal() ;  
    }  
}
```

操作





# 哲学家就餐问题的管程解决方案

- 每个哲学家按照以下的顺序轮流调用操作 `pickup()` 和 `putdown()`

`dp.pickup (i)`

EAT

`dp.putdown (i)`



## 7、同步实例





# Windows同步机制

- 事件 (Event)
  - 通过通知操作的方式来保持线程的同步
- 临界区 (CriticalSection)
- 互斥量 (Mutex)
- 自旋锁 (Spinlock)
- 信号量 (Semaphore)
  
- 在单处理器系统中，使用中断屏蔽来保护全局资源的访问
- 在多处理器系统中使用自旋锁
- 互斥量是可以命名的，可以跨进程使用；进程内部使用，可以选择临界区
- 也提供调度对象，这些对象类似于互斥、信号量等
- 调度对象也可以提供事件来同步 (**events**)
  - 事件的使用与条件变量类似





# 事件

- 事件（Event）对象可通过通知操作的方式来保持同步，可实现不同进程中的线程同步操作（跨进程）
- 事件是WIN32中最灵活的线程间同步机制
- 事件存在两种状态：
  - 激发状态（Signaled or True）
  - 未激发状态(Unsignaled or False)
- 事件可分为两类：
  - 手动设置：  
这种对象只可能用程序手动设置，在需要该事件或者事件发生时，采用**SetEvent**及**ResetEvent**来进行设置。
  - 自动恢复：  
一旦事件发生并被处理后，自动恢复到没有事件状态，不需要再次设置。



# 事件 API

## ■ 创建事件：

```
HANDLE CreateEvent(  
    LPSECURITY_ATTRIBUTES lpEventAttributes,  
    // SECURITY_ATTRIBUTES结构指针，可为NULL  
    BOOL bManualReset,  
    // 手动/自动  
    // TRUE: 在WaitForSingleObject后必须手动调用ResetEvent清除信号  
    // FALSE: 在WaitForSingleObject后，系统自动清除事件信号  
    BOOL bInitialState, //初始状态  
    LPCTSTR lpName //事件的名称  
)
```

## ■ 打开事件

```
HANDLE hEvent=OpenEvent(EVENT_ALL_ACCESS,true,“MyEvent”);
```

## ■ 其它

ResetEvent()      SetEvent()      WaitForMultipleObjects()





# 事件例子

## ■ 三个线程：

- 主线程
- 读线程
- 写线程
- 读线程必须在写线程操作结束后才能进行读；
- 主线程必须等读和写线程结束后才能结束



# 代码

```
include "stdafx.h"
#include <windows.h>
#include <process.h>
#include <iostream.h>
#include <fstream.h>
HANDLE evRead,evFinish;

void ReadThread(LPVOID param) {
    WaitForSingleObject(evRead, INFINITE);
    cout<<"Reading"<<endl;
SetEvent(evFinish);
}

void WriteThread(LPVOID param) {
    cout<<"Writing"<<endl;
SetEvent(evRead);
}
```

```
int main(int argc, char* argv[])
{
    evRead=
CreateEvent(NULL, FALSE, FALSE, NULL);
    evFinish=
CreateEvent(NULL, FALSE, FALSE, NULL);
    _beginthread(ReadThread, 0, NULL);
    _beginthread(WriteThread, 0, NULL);
WaitForSingleObject(evFinish, INFINITE);
    cout<<"End."<<endl;

    return 0;
}
```





# 临界区

- 临界区：保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问
- 临界区包含两个操作原语：
  - EnterCriticalSection () : 进入临界区
  - LeaveCriticalSection () : 离开临界区
- 临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程
- 用法：

.....

```
EnterCriticalSection (& gCriticalSection) ;  
//do something (CriticalSection)  
LeaveCriticalSection(& gCriticalSection);
```

.....





# 临界区API

- 创建临界区：在进程中分配一个全局CRITICAL\_SECTION数据结构：  
`CRITICAL_SECTION gCriticalSection;`
- 使用临界区：使用临界区之前，必须调用InitializeCriticalSection函数初始化：  
`VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 进入临界区：调用EnterCriticalSection函数进入临界区：  
`VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 离开临界区：调用LeaveCriticalSection函数退出了临界区：  
`VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`
- 删除临界区：调用DeleteCriticalSection函数删除临界区：  
`VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);`





# 临界区—例子

- 一个银行系统中两个线程执行取款操作，一个使用ATM机，另一个使用存折在柜台取款。如果不加于控制，会使得账户余额为负数。





# 代码

```
#include "stdafx.h"
#include <windows.h>
#include <process.h>
#include <iostream.h>
#include <fstream.h>

int total =1000;
HANDLE evFin[2];
CRITICAL_SECTION cs; //声明临界区对象cs

void WithDrawThread1(LPVOID param) {
    EnterCriticalSection(&cs);
    if ((total-900) >= 0) {
        total-=900;
        cout<<"Withdraw $900." <<endl;
    } else {
        cout<<"No enough money!"<<endl;
    }
    LeaveCriticalSection(&cs);
    SetEvent(evFin[0]);
}
```

```
void WithDrawThread2(LPVOID param) {
    EnterCriticalSection(&cs);
    if ((total-700) >= 0) {
        total-=700;
        cout<<"Withdraw $700." <<endl;
    } else {
        cout<<"No enough money!"<<endl;
    }
    LeaveCriticalSection(&cs);
    SetEvent(evFin[1]);
}
```





```
int main (int argc, char* argv[]) {  
    evFin[0]=CreateEvent(NULL, FALSE, FALSE, NULL);  
    evFin[0]=CreateEvent(NULL, FALSE, FALSE, NULL);  
  
    InitializeCriticalSection(&cs); //初始化临界区  
  
    _beginthread(WithDrawThread1,0,NULL);  
    _beginthread(WithDrawThread2,0,NULL);  
    WaitForMultipleObjects(2,evFin,TRUE,INFINITE);  
  
    DeleteCriticalSection(&cs); //删除临界区  
  
    cout<<total<<endl;  
    return 0;  
}
```





# 信号量

- 信号量是维护0到指定最大值之间的同步对象
- 信号量状态在其计数大于0时是有信号的，而其计数是0时是无信号的
- 信号量对象在控制上可以支持有限数量共享资源的访问





# 信号量—创建和撤销

## ■ 创建信号量：

```
HANDLE CreateSemaphore (
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount, //开始时可供使用的资源数
    LONG lMaximumCount, //最大资源数
    PCTSTR pszName);
```

## ■ 撤销信号量：

```
BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount, //信号量的当前资源数增加
    LPLONG lpPreviousCount
);
```

## ■ 打开信号量

```
HANDLE OpenSemaphore (
    DWORD dwAccess,
    BOOL bInheritHandle,
    PCTSTR pszName
);
```





# 生产者-消费者例子

```
#include "stdafx.h"
#include <windows.h>
#include <iostream.h>
#define THREAD_INSTANCE_NUMBER 3
#define PRODUCT_NUMBER 2
#define MAX_ITEMS 2

// 定义信号量
HANDLE m_S_Full; // Semaphore
HANDLE m_S_Empty; // Semaphore
HANDLE m_E_Mutex; // Event

// 假设仓库最多容纳MAX_ITEMS个物品，开始仓库为空
DWORD IDP[THREAD_INSTANCE_NUMBER];
DWORD IDC[THREAD_INSTANCE_NUMBER];
HANDLE hp[THREAD_INSTANCE_NUMBER];
HANDLE hc[THREAD_INSTANCE_NUMBER];
```





# 生产者-消费者例子

```
int main(int argc, char* argv[]){
    int i;
    //初始计数为M
    m_S_Full = CreateSemaphore(NULL, 0, MAX_ITEMS, NULL);
    //初始计数为0
    m_S_Empty = CreateSemaphore(NULL, MAX_ITEMS, MAX_ITEMS, NULL);
    //自动类型， 初始状态为信号态
    m_E_Mutex = CreateEvent(NULL, FALSE, TRUE, NULL);

    for (i=0;i<THREAD_INSTANCE_NUMBER;i++)
    {
        hp[i]=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadProducer,(void*)&IDP[i],0,&(IDP[i]));
        hc[i]=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadConsumer,(void*)&IDC[i],0,&(IDC[i]));}

    WaitForMultipleObjects(THREAD_INSTANCE_NUMBER,hp,true,INFINITE);
    WaitForMultipleObjects(THREAD_INSTANCE_NUMBER,hc,true,INFINITE);
    cout<<"Close the mutex handle!"<<endl;
    return 0;
}
```





# 生产者-消费者例子

```
int counter=0;
void ThreadProducer(void *pData) {
    int j;
    int ThreadNumTemp=(*(int *) pData);

    for (j=0;j<PRODUCT_NUMBER;j++) {
        WaitForSingleObject(m_S_Empty, INFINITE);
        WaitForSingleObject(m_E_Mutex, INFINITE);

        // OK now, put product
        counter++;
        cout<<"ThreadProducer:<<ThreadNumTemp<<" puts a porduct."<<endl;
        cout<<"ThreadProducer:<<ThreadNumTemp<<" counter=<<counter<<endl;

        // relase consumer's semaphore
        ReleaseSemaphore(m_S_Full, 1, NULL);
        // set event to signal
        SetEvent(m_E_Mutex);
    }
}
```





# 生产者-消费者例子

```
void ThreadConsumer(void *pData) {  
    int j;  
    int ThreadNumTemp=(*(int *) pData);  
  
    for (j=0;j<PRODUCT_NUMBER;j++) {  
        WaitForSingleObject(m_S_Full, INFINITE);  
        WaitForSingleObject(m_E_Mutex, INFINITE);  
  
        // OK now, get a product  
        counter--;  
        cout<<"ThreadConsumer:<<ThreadNumTemp<<" gets a product."<<endl;  
        cout<<"ThreadConsumer:<<ThreadNumTemp<<" counter="<<counter<<endl;  
        // release producer's semaphore  
        ReleaseSemaphore(m_S_Empty, 1, NULL);  
        // set event to signal  
        SetEvent(m_E_Mutex);  
    }  
}
```





# Linux同步机制

- 使用禁止中断来实现短的临界区
- 自旋锁（**spinlock**）
  - 不会引起调用者阻塞
- 互斥锁（**Mutex**）
- 条件变量（**Condition Variable**）
- 信号量（**Semaphore**）





# Solaris同步

■ Solaris 实现了多种锁来支持多任务、多线程（包括实时线程）和多进程处理。

- 适应互斥量：保护那些为较短代码段所访问的数据
- 条件变量
- 信号量
- 读-写锁：用于保护经常访问但通常是只读访问的数据
- 十字转门：用于安排等待获取适应互斥和读写锁的线程链表





# Pthreads同步

- Pthreads API 操作系统无关

- 提供了:

- 互斥锁
- 条件变量

- 不可移植的扩展包括:

- 读写锁
- 自旋锁