

并发：互斥与同步

阅读
Oyster
easy
32 section
Concurrent
Concurrent
Promise

本章教学目标

- 理解进程的并发性带来的问题
- 理解什么是临界区管理
- 掌握临界区管理的软、硬件方法
- 掌握信号量及PV操作解决互斥与同步问题
- 掌握管程的概念及用法
- 掌握进程间通信的方法

进程的顺序性

- 一个进程在处理器上的执行是严格按序的，一个进程只有当一个操作结束后，才能开始后继操作
- 顺序程序设计是把一个程序设计成一个顺序执行的程序模块，顺序的含义不但指一个程序模块内部，也指两个程序模块之间。

顺序程序设计特点

- 程序执行的顺序性
- 程序环境的封闭性
- 程序执行结果的确定性
- 计算过程的可再现性

进程的并发性(1)

- 进程执行的并发性：一组进程的执行在时间上是重叠的。
- 并发性举例：有两个进程A(a1、a2、a3)和B(b1、b2、b3)并发执行。
 - a1, a2, a3, b1, b2, b3
 - a1, b1, b2, a2, a3, b3
- 从宏观上看，并发性反映一个时间段中几个进程都在同一处理器上，处于运行还未运行结束状态。
- 从微观上看，任一时刻仅有一个进程在处理器上运行。

进程的并发性(2)

- 并发的实质是一个处理器在几个进程之间的多路复用，并发是对有限的物理资源强制行使多用户共享，消除计算机部件之间的互等现象，以提高系统资源利用率。

进程的并发性(3)

- 单处理机系统中，多道程序设计使得不同的进程交错执行
- 多处理机系统中，不同的进程可以同时执行。
- 进程执行的速度不可预测，依赖于其它进程的行为、操作系统处理中断的方式以及操作系统的调度策略。
- 并发性使程序的执行失去了封闭性、顺序性、确定性和可再现性。
测试：再现错误？

Time →



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

并发程序设计

```
while(true){  
    input; P1  
    process; P2  
    output; P3  
}
```



P1
while(true){

input;
 send to P2;

}

程序P1(i)

P2
while(true){

receive from P1; 程序P2(p)
 process;

send to P3; 其实内核在忙?

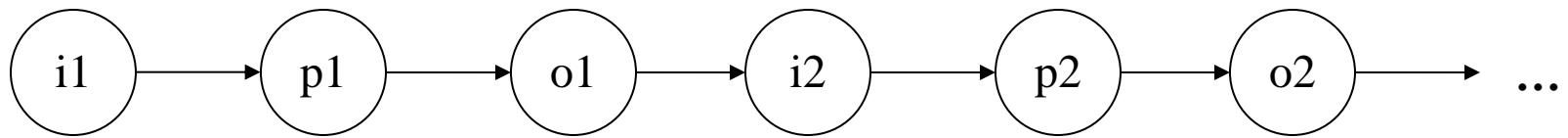
}

P3
while(true){

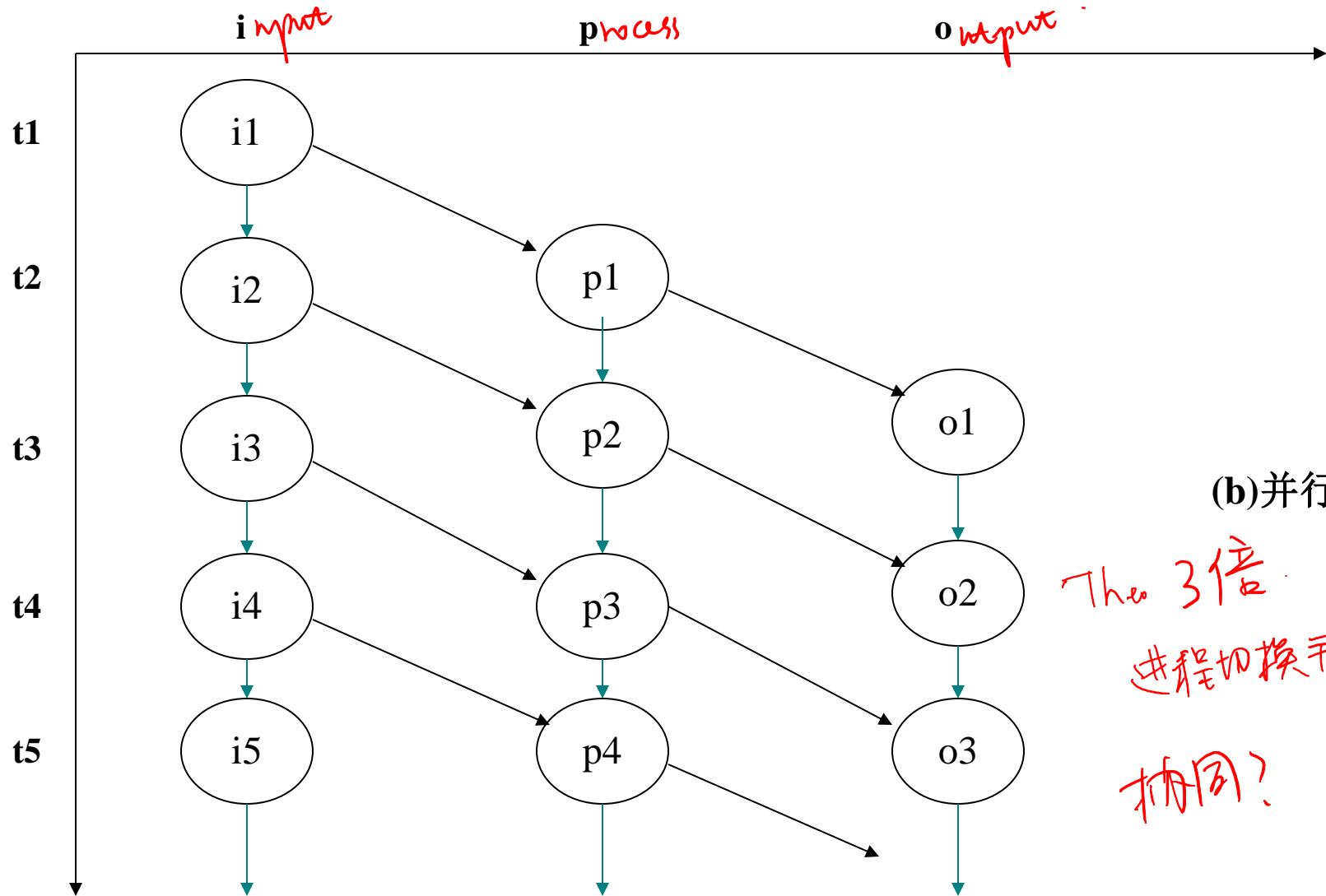
receive from P2;
 output;

}

程序P3(o)



(a)串行工作



并发带来的问题

- 输入依赖之前进程的
输出
- 并发进程可能是无关的，也可能是交互的
 - 无关的并发进程是指它们分别在不同的变量集合上操作
 - 交互的并发进程共享某些变量，之间具有制约关系，有可能产生时间相关的错误。

无关的并发进程

- 无关的并发进程
 - 一组并发进程分别在不同的变量集合上操作
 - 一个进程的执行与其他并发进程的进展无关。
- 并发进程的无关性是进程的执行与时间无关的一个充分条件，~~判断~~ 又称为Bernstein条件。

Bernstein 条件

read

- 设 $R(p_i) = \{a_1, a_2, \dots, a_n\}$, 表示程序 p_i 在执行期间 **引用** 的变量集;
- write*
- $W(p_i) = \{b_1, b_2, \dots, b_n\}$, 表示程序 p_i 在执行期间 **改变** 的变量集;
- 两个进程的程序 p_1 和 p_2 满足 Bernstein 条件是指 引用变量集与改变变量集 交集之并为空集, 即:

$$(R(p_1) \cap W(p_2)) \cup (R(p_2) \cap W(p_1)) \cup (W(p_1) \cap W(p_2)) = \{\}$$

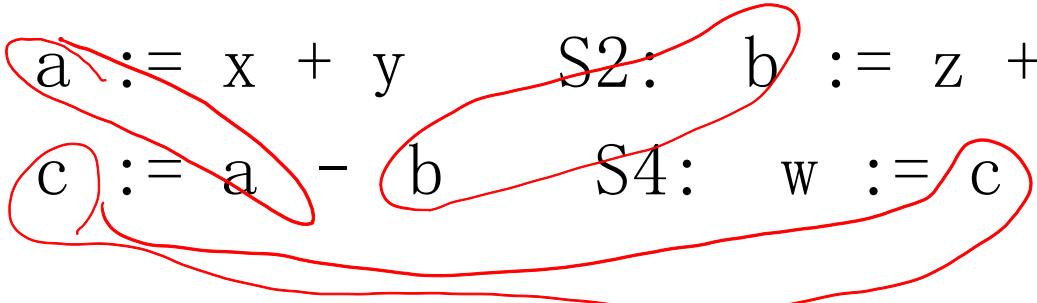
均为空

$R(p_2) \cap R(p_1)$ 先读后读无关

Bernstein条件举例

例如，有如下四条语句：

S1: $a := x + y$ S2: $b := z + 1$
S3: $c := a - b$ S4: $w := c + 1$



于是有： $R(S1) = \{x, y\}$, $R(S2) = \{z\}$,
 $R(S3) = \{a, b\}$, $R(S4) = \{c\}$; $W(S1) = \{a\}$,
 $W(S2) = \{b\}$, $W(S3) = \{c\}$, $W(S4) = \{w\}$ 。

$\checkmark S_1, S_2$

$\checkmark S_1, S_4$

Q: 哪些语句可并发执行? $\checkmark S_2, S_4$

并发程序带来的好处

- 单处理器系统上，可有效利用资源，让处理器和I/O设备、I/O设备和I/O设备同时工作，充分发挥机器部件的并行能力；
✓ 硬件能并行工作仅有了提高效率的可能性，硬部件并行性的实现需要软件技术去利用和发挥，这种软件技术就是并发程序设计。
自己写
- 在多处理器系统上，可让进程在不同处理器上物理地并行工作
- 并发程序设计是多道程序设计的基础，多道程序的实质就是把并发程序设计引入到系统中。

与时间有关的错误

- 并发进程使得进程的执行不可预测，甚至无法再现。
- 进程间的相互影响和制约导致对资源的共享充满了危险，各种 **与时间有关的错误** 就可能出现
 - 结果不唯一
 - 结果与进程执行的速度相关
 - 永远等待
 - 进程相互等待产生死锁，或进程一直得不到资源产生饿死现象

P &
req(A) req(B)
req(B) req(A)

结果不唯一（例1）

开始

生产者

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

三条机器指令

```
register1=counter;
register1=register1+1;
counter=register1;
```

消费者

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

```
register2=counter;
register2=register2-1;
counter=register2;
```

实验台 5

已解
决

| | | |
|-----------|------------------------|---------------|
| T0: 生产者执行 | register1=counter; | [register1=5] |
| T1: 生产者执行 | register1=register1+1; | [register1=6] |
| T2: 生产者执行 | counter = register1; | [counter=6] |
| T3: 消费者执行 | register2=counter; | [register2=6] |
| T4: 消费者执行 | register2=register2-1; | [register2=5] |
| T5: 消费者执行 | counter=register2; | [counter=5] |

执行序列1

| | | |
|-----------|------------------------|-------------------|
| T0: 生产者执行 | register1=counter; | [register1=5] |
| T1: 生产者执行 | register1=register1+1; | [register1=6] |
| T2: 消费者执行 | register2=counter; | [register2=5] 被打断 |
| T3: 消费者执行 | register2=register2-1; | [register2=4] |
| T4: 生产者执行 | counter=register1; | [counter=6] |
| T5: 消费者执行 | counter=register2; | [counter=4] |

执行序列2

| | | |
|-----------|------------------------|---------------|
| T0: 生产者执行 | register1=counter; | [register1=5] |
| T1: 生产者执行 | register1=register1+1; | [register1=6] |
| T2: 消费者执行 | register2=counter; | [register2=5] |
| T3: 消费者执行 | register2=register2-1; | [register2=4] |
| T4: 消费者执行 | counter=register2; | [counter=4] |
| T5: 生产者执行 | counter=register1; | [counter=6] |

执行序列3

结果不唯一 (例2)

相关

P1:

```
a = a +1;  
b = b + 1;
```

P2:

```
b = 2 * b;  
a = 2 * a;
```

a =b=1;

线性执行

a=2

b=2;

b=4;

a=4;

a ==b

并发执行

a=a+1; //a=2;

b=2*b; //b=2;

b=b+1; //b=3;

a=2*a; //a=4;

顺序
F-3

a !=b

结果不唯一（例3）

```
void T1(){  
    //按旅客订票要求找到Aj  
    int X1=Aj;  
    if(X1>=1){  
        X1--;  
        Aj=X1;  
        //输出一张票;  
    }  
    else{  
        //输出票已售完  
    }
```

```
void T2(){  
    //按旅客订票要求找到Aj  
    int X2=Aj;  
    if(X2>=1){  
        X2--;  
        Aj=X2;  
        //输出一张票;  
    }  
    else{  
        //输出票已售完  
    }
```

一张票卖给两个人

永远等待

任何时刻都会被中断

```
int X = memory;
```

```
void borrow(int B)
```

```
{
```

```
    while(B>X){
```

进程进入(等待主存资源队列);

```
}
```

$B < X$

```
X=X-B;
```

修改主存分配表，进程获得主存资源；

```
}
```

//修改内存

申请 B 大小的内存

中断

$X > B$

但已经判断过

可能永远

不会被唤醒

```
void return(int B)
```

```
{
```

```
    X=X+B;
```

修改主存分配表；

释放(等待主存资源)的进程；空闲
(无)

```
}
```

两种关系

进程的交互

- 竞争
 - 多个进程之间并不知道其他进程的存在，竞争共享硬设备、存储器、处理器和文件资源等
 - 两个控制问题
 - 死锁问题
 - 饥饿问题
 - 操作系统应该提供支持，合理分配资源，解决资源的竞争问题
 - 进程的互斥访问是解决进程间竞争资源的手段
 - **进程互斥**是指若干进程因相互争夺独占型资源而产生的竞争制约关系
- 协作
 - 一组进程之间相互知道对方的存在，协作完成同一任务。
 - 进程的同步是解决进程间协作关系的手段。
 - **进程同步**是指为完成共同任务的并发进程基于某个条件来协调其活动，因为需要在某些位置上排定执行的先后次序而等待、传递信号或消息所产生的协作制约关系
 - 进程互斥关系是一种特殊的进程同步关系。

不允许多
打断

临界区管理

- 互斥与临界区
- 实现临界区管理的几种尝试
- 实现临界区管理的软件方法
- 实现临界区管理的硬件设施

互斥与临界区(1)

- 并发进程中与共享变量有关的程序段叫“临界区”，共享变量代表的资源叫“临界资源”。
- 与同一变量有关的临界区分散在各进程的程序段中，而各进程的执行速度不可预知。
- 如果保证进程在临界区执行时，不让另一个进程进入临界区，即各进程对共享变量的访问是互斥的，就不会造成与时间有关的错误。

互斥与临界区(2)

- 一次至多一个进程能够进入临界区内执行；
- 如果已有进程在临界区，其他试图进入的进程应等待；
- 进入临界区内的进程应在有限时间内退出，以便让等待进程中的一个进入；
- 临界区调度原则：
互斥使用、有空让进，忙则等待、有限等待，择二而入，算法可行；

```
boolean inside1 = false; // 上否在临界区  
boolean inside2 = false; //
```

Process P1() {

 while(inside2);

 inside1 = true;

 /* critical section */

 inside1 = false;

 /* remainder section */

}



Process P2() {

 while(inside1);

 inside2 = true;

 /* critical section */

 inside2 = false;

 /* remainder section */

}

2

已经进入互斥 (与现实可能不符)

boolean inside1 = false;

boolean inside2 = false;

Process P1() {

 inside1 = true;

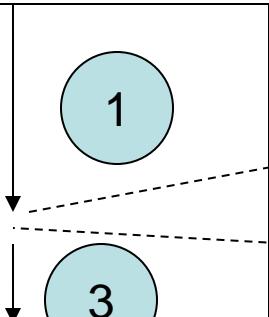
 while(inside2);

 /* critical section */

 inside1 = false;

 /* remainder section */

}



Blocked!

同时阻塞!

无法执行

Process P2() {

 inside2 = true;

 while(inside1);

 /* critical section */

 inside2 = false;

 /* remainder section */

2

Blocked!

临界区管理的尝试

Peterson方法

int turn; //表示现在轮到谁进入

boolean flag[2]; // //表示进程希望进入临界区的意愿

flag[0]=flag[1]=false;

Q: 如何将该算法扩展到n个进程?

Process P0() {

flag[0] = true;

turn = 1;

while (flag[1]&&turn==1) ;

/* critical section */

flag[0] = false;

/* remainder section; */

}

Process P1() {

flag[1] = true;

turn = 0;

while (flag[0]&&turn==0) ;

/* critical section */

flag[1] = false;

/* remainder section; */

}

Peterson方法的正确性

- 互斥性
 - 若P0, P1同时进入临界区，则意味 $\text{flag}[0]=\text{flag}[1]=\text{true}$; 但 turn 只可能取值0或1，因此不可能同时进入临界区
- 有空让进
 - 若P1不准备竞争临界区资源，则 $\text{flag}[1]=\text{false}$, 因此, P0可以进入临界区
- ~~有限等待~~
 - 若P0希望进入临界区而被阻塞，则表明 $\text{turn}=1$, 且 $\text{flag}[1]=\text{true}$; 此时, P1在临界区.
 - 当P1执行完以后, 将设置 $\text{flag}[1]=\text{false}$, 若此时P0想进入临界区, 则可以进入
 - 如果P1执行完 $\text{flag}[1]=\text{false}$ 后, 在P0被调度之前, P1又想进入临界区, 则P1将执行 $\text{flag}[1]=\text{true}$, $\text{turn}=0$, 此时P1将被阻塞, 而P0将得以进入临界区
 - 因此, 进程最多等待另一个进程执行完临界区代码一次即可进入临界区

解决临界区问题的硬件方法

CPU < 中断
程序主动系统调用

- 关中断
- test_and_set指令
- Swap指令

关中断

- 关中断使得当前执行的进程不会被打断，从而**不会发生进程切换**
- 关中断时间过长会影响系统效率
- **不适用多处理器系统**。在一个**CPU**上关中断，并不能防止其他处理器上也执行相同的临界区代码

```
while(true)
{
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

test_and_set

```
boolean test_and_set (boolean x) {  
    if(x==true) {  
        x = false;  
        return true;  
    }  
    else  
        return false;  
}
```

此指令为原子操作，不可分割！

```
boolean lock = true; //资源初始可用
Process Pi( )
{
    while(true){
        /**
         * 若资源可用，则将其设为不可用，并返回true，从而通过测试进入临界区；
         * 若资源不可用，则lock为false，返回false，进程忙等；
        */
        while(test_and_set(lock)==false) ;
        /* critical section */;
        lock = true;
        /* remainder */;
    }
}
```

test_and_set解决临界区管理

- 将临界区与一个布尔变量lock相关联
- lock代表临界资源的状态，可以看成一把锁，lock为true/false表示临界资源可用/不可用
- 初始值为true, 如有进程要进入临界区，则对lock进行测试和设置指令
 - 如果已有进程在临界区，则test_and_set返回false,将被阻止进入临界区
 - 如果没有进程在临界区，则test_and_set返回true, 同时将lock设为false,以阻止其它进程进入临界区
- 当进程离开临界区时，将lock设置为true, 表示临界资源可用
- 能实现进程互斥访问临界区，但是可能会出现进程饿死的情况
 - 如何避免进程饿死?

```

boolean waiting[n];
boolean lock;
lock = true;          //表明临界资源初始可用;
waiting[0...n-1]=false; //当前没有进程在等待
Pi(){
    do{
        waiting[i] = true;           //进程i等待进入临界区;
        boolean key = false;
        while(waiting[i] && !key)   //若当前lock为false, 表明有进程在临界区, 则key为false;
            key = test_and_set(lock); //若当前lock为true, 表明无进程在临界区, 则key为true;
        waiting[i] = false;
        /* critical section */
        j = (i+1)%n;
        while((j!=i) && !waiting[j]) //找到i后面第一个在等待进入临界区的进程j
            j = (j+1)%n;
        if(j == i)                  //若无进程, 则临界区资源标为可用;
            lock = true;
        else
            waiting[j] = false;     /*否则, 并不将临界区资源标识为可用, 而是将进程j等待标识为false, 使得进程j可以进入临界区, 而其它进程则无法进入; */
        /* remainder section */
    }while(true);
}

```



对换指令

```
void swap(boolean a,boolean b){  
    boolean temp;  
    temp = b;  
    b = a;  
    a = temp;  
}
```

例如，Pentium中的XCHG指令

```
boolean lock = false ; //表示资源可用
Process Pi( )
{
```

```
    while(true){
        boolean keyi = true;
```

```
    /*
```

若资源可用，则lock为false，执行swap后keyi为false，通过测试，进入临界区；

若资源不可用，则lock=true，执行swap后keyi仍然为true，进程忙等；

```
    */
```

```
        do swap(keyi, lock) while(keyi);
```

```
        /* critical section */;
```

```
        lock = false;
```

```
        /* remainder */;
```

```
    }
```

```
}
```

- 为每个临界区设置锁变量**lock**, **lock**为**false**表示无进程在临界区内
- 当进程进入临界区时, **lock**设置为**true**, **keyi**变为**false**, 因此进程得以通过测试, 进入临界区
- 当另一个进程希望进入临界区时, **swap(keyi, lock)**的结果是**keyi=lock=true**, 因此, 进程被阻止进入临界区
- 当进程退出临界区时, 将**lock**设为**false**, 从而被阻止进程得以进入临界区
- 同样, 能实现进程互斥访问临界区, 但是可能会出现进程饿死的情况

基于机器指令方法的优缺点

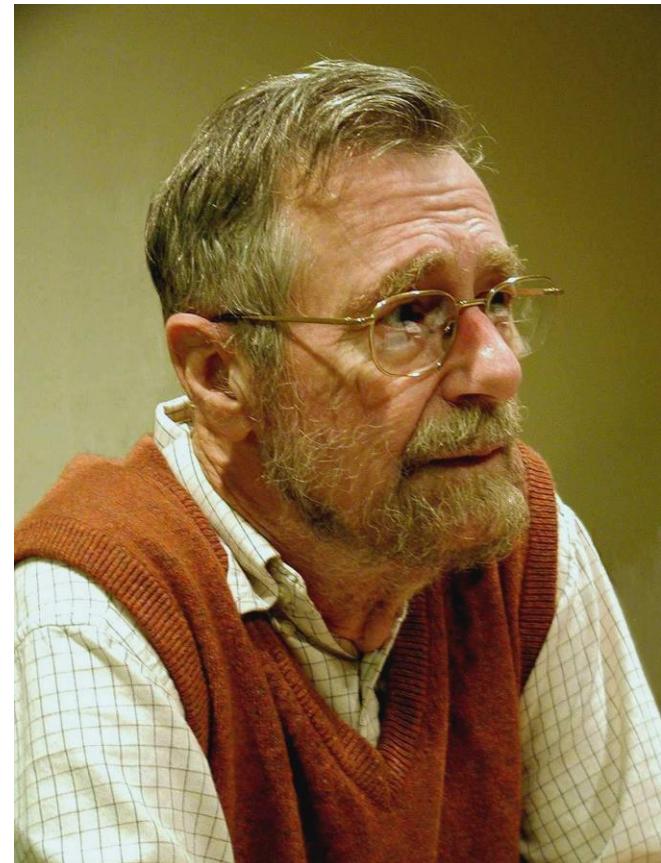
- 优点
 - 适用于任意数目的进程，适用于单处理机系统和多处理机系统，只要共享内存即可；
 - 简单，易于验证；
 - 可以用于支持多个临界区，为每个临界区定义一个变量即可
- 缺点
 - 采用了忙等方式，浪费CPU资源
 - 可能会导致饥饿
 - 不满足有限等待
 - 可能会导致死锁
 - 例如，一个低优先级的进程P1先进入临界区，然后被处理器中断，执行高优先级进程P2，则P2处于忙等，P1因优先级低而不会被处理器调度，从而产生死锁。
 - 实际上，这个例子中临界区是一个资源，而CPU是另一个资源。
 - 只能应用于进程竞争，不能解决进程协作

信号量与PV操作

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

Edsger W. Dijkstra

- 1930.5.11—2002.8.6
- 1972年获得图灵奖
- 成就：
 - 提出“`goto`有害论”；
 - 提出信号量和PV原语；
 - 解决了有趣的“哲学家聚餐”问题；
 - 最短路径算法(SPF)和银行家算法的创造者；
 - 第一个Algol 60编译器的设计者和实现者；
 - THE操作系统的设计者和开发者
- 与D. E. Knuth(*The Art of Computer Programming*的作者)并称为我们这个时代最伟大的计算机科学家的人



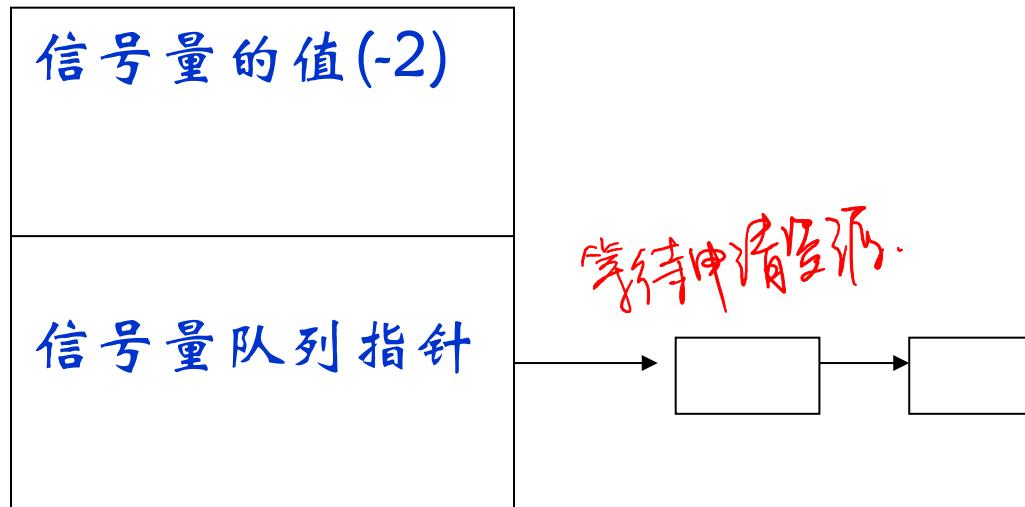
信号量与PV操作

数据结构

- 信号量：一种软件资源
- 原语：内核中执行时 **不可被中断** 的过程
- P操作原语和V操作原语 **原子操作**
- 一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(semaphore)，复杂的进程合作需求都可以通过适当的信号结构得到满足。

信号量

- 操作系统中，信号量表示物理资源的实体，它是一个与队列有关的整型变量。
- 实现时，信号量是一种记录型数据结构，有两个分量：一个是**信号量的值**，另一个是**信号量队列的队列指针**。



信号量

- 记录型信号量的定义

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

其中，`value`为一个整型变量，系统初始化时为其赋值；`list`是等待使用此类资源的进程队列的头指针，初始状态为空队列。

信号量的操作

只有通过

- 对信号量的操作包括两个原子操作, P操作和V操作, 也称为wait()操作和signal()操作

甲请资源

```
void P(semaphore s){  
    s.value --; 可用资源个数  
    if(s.value<0){ 进程甲请不到资源.    >> 结束 ✓  
        place this process in s.list;  
        block this process; 进程阻塞(等待)  
    }  
}
```

释放资源

void V(semaphore s){

list非空 — if(s.value <= 0){

有process在等待 资源

```
    s.value++;  
    if(s.value <= 0){  
        remove a process P from s.list ; 从队列取出.  
        place process P on ready list ; 放入就绪队列.  
    }  
}
```

>> 没有进程在等待. ✓

PV操作的原子性

不可分割(被打断)

- 保证PV操作本身的原子性是典型的临界区问题，即任何时候，只有一个进程能进入P操作或V操作函数的内部
- 可以采用之前介绍过的软件方法或硬件方法保证P操作或V操作本身的原子性
 - Dekker's算法
 - Peterson's算法
 - 关中断(仅限于单处理器)
 - Test_and_set指令
 - swap指令
 - ...

信号量的含义

- 若信号量 $s.value$ 为 正值，则此值等于在 封锁进程之前对信号量 s 可施行的 P 操作数，也就是 s 所代表的实际可用的物理资源数；
- 若信号量 $s.value$ 为 负值，则其绝对值等于 登记排列在 s.list 中的等待进程个数，即恰好等于对信号量 s 实施 P 操作而被阻塞并进入信号量 s 的等待队列的进程数；
- P 操作通常意味着 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 挂起进程的操作，而 V 操作代表 唤醒被挂起进程的操作。

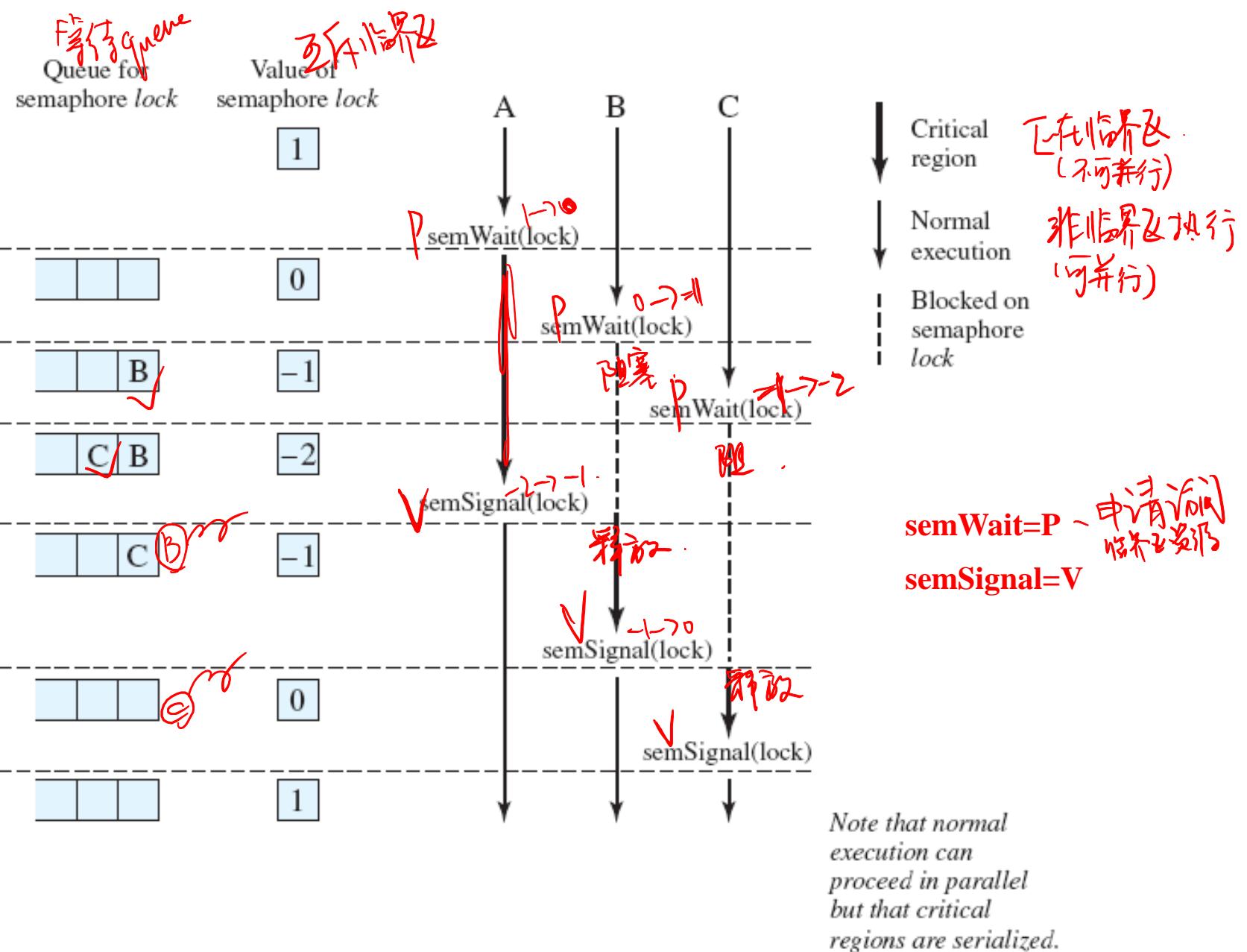


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

(二元信号量)

- 若信号量s的value取值只能为0或1，则称s为二元信号量，其P,V操作定义如下：

```
void P(semaphore s){  
    if(s. value == 1)  
        s. value = 0;  
  
    else {  
        place this process in s.list;  
        block this process;  
    }  
}
```

```
void V(semaphore s){  
    if(s. list is empty)  
        s. value = 1;  
  
    else {  
        remove a process P from s.list ;  
        place process P on ready list ;  
    }  
}
```

信号量实现互斥

```
semaphor mutex; // 定义一个信号量，代表临界区锁  
mutex.value = 1; // 初始值设为1  
  
Process Pi() {  
    P(mutex); // 进入临界区之前执行 P(mutex) 申请锁资源  
    /* critical section */; // 只保护这一段  
    V(mutex); // 退出临界区时执行 V(mutex) 释放锁资源并唤醒可能等待进程;  
    /* remainder section */; // 并发  
}
```

(注) 互斥信号量：初始化用函数（仅一次）

哲学家进餐问题

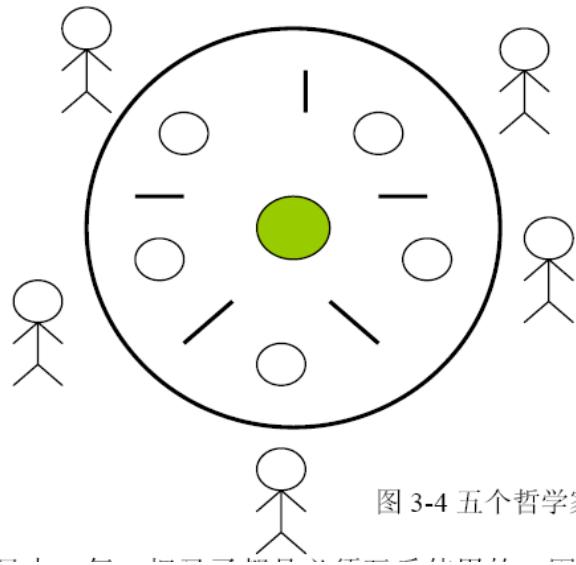


图 3-4 五个哲学家吃通心面问题

问题描述：

5位哲学家围在一张圆桌旁，桌子中央有一盘通心粉，每人面前有一只空盘子，每两人之间有一只筷子；每位哲学家思考，饥饿，然后吃通心面；为了吃面，哲学家必须获得两把叉子，且每人只能直接从紧邻自己的左边或右边去取叉子。

同时刻 2 个哲学家吃

(process)

哲学家进餐问题

- ~~哲~~^哲~~家~~^家
- 每把~~叉子~~都必须~~互斥~~使用，因此，应为每把~~叉子~~设置~~互斥信号量~~ $\text{fork}[i]$ ($i=0,1,2,3,4$)，其初始值均为1
 - 当一位哲学家吃通心面之前必须执行两个P操作，获得自己左边和右边的两把叉子
 - 在吃完通心面后执行两个V操作，放下两把叉子。

```

semaphor fork[5];

for( int i = 0;i<5;i++)
    fork[i] = 1;          //初始化互斥

Process philosopher_i(){ //i=0,1,2,3,4;
    while(true)
    {
        think();
        P(fork[i]); //尝试获得左边叉子 ↓↓↓↓
        P(fork[(i+1)%5]); //尝试获得右边叉子 ↓
        eat();
        V(fork[i]); //释放左边叉子
        V(fork[(i+1)%5]); //释放右手叉子
    }
}

```

注：若5个哲学家同时拿起自己左手边（或右手边）的筷子，则所有哲学家将处于等待状态，出现死锁。

死锁的避免将在后续章节介绍。

哲学家进餐中死锁的一种解决方案

- 不允许哲学家拿起一把叉子等待另一把叉子
- 要么两把叉子一起获得，要么一把都不持有

```

Semaphor fork[5], mutex;
int state[5]; /*筷子状态，0表示
               /*空闲,1表示被占用*/
for( int i = 0;i<5;i++){
    fork[i] = 1;
    state[i]=0;
}
mutex=1;

```

```

Process philosopher_i(){ //i=0,1,2,3,4;
while(true)
{
    think();
    P(mutex);
    if(state[i]==0&&state[(i+1)%5]==0){ //两把叉子同时空闲
        state[i]=1;
        state[(i+1)%5]=1;
        P(fork[i]); //尝试获得左边叉子
        P(fork[(i+1)%5]); //尝试获得右边叉子
    }
    V(mutex);
    eat(); ← 没拿到叉子就吃了
    P(mutex);
    state[i]=0;
    state[(i+1)%5]=0;
    V(fork[i]); //释放左边叉子
    V(fork[(i+1)%5]); //释放右手叉子
    V(mutex);
}
}

```

缺乏对一把叉子都未获得的进程阻塞功能

```

Semaphor fork[5], mutex;
int state[5];
for( int i = 0;i<5;i++){
    fork[i] = 1;
    state[i]=0;
}
mutex=1;

```

```

Process philosopher_i(){ //i=0,1,2,3,4;
    while(true)
    {
        think();
        P(mutex);
        if(state[i]==0&&state[(i+1)%5]==0){
            state[i]=1;
            state[(i+1)%5]=1;
            P(fork[i]); //尝试获得左边叉子
            P(fork[(i+1)%5]); //尝试获得右边叉子
            eat();           一次只能有一个
            state[i]=0;
            state[(i+1)%5]=0;
            V(fork[i]); //释放左边叉子
            V(fork[(i+1)%5]); //释放右手叉子
            V(mutex);
        }
    }
}

```

降低了并发性 同一时间仅有一个 哲学家能吃通心面

```
#define THINKING 0
#define HUNGRY    1    状态
#define EATING    2
semaphor s[5]; //用于阻塞哲学家的信号量 一个又饿又取不到时不能吃
semaphore mutex=1;
int state[5]; //哲学家的状态
for(int i=0;i<5;i++){
    state[i]=THINKING; //一开始都在思考
    s[i] = 0;
}
    value .
```

```
void take_fork(int i){
```

P(mutex);

state[i]=HUNGRY;

test(i); //试找左右

V(mutex);

s[i].value $\mapsto 0$

P(s[i]);

0 $\mapsto -1$ 等待

/*若状态不为EATING，则此处进程将阻塞自己；

变为EATING有两种可能：

(1) 自己执行test()时，已经获得两把叉子，此时执行了V(s[i])，再执行P(s[i])不会阻塞；

(2) 邻居放下叉子时发现本进程等待的两把叉子空，且处于饥饿状态，因此决定将叉子都分配给该进程；

*/

}

```
void put_fork(int i){
```

P(mutex);

state[i]=THINKING; //当前进程不再eat，表示归还两把叉子

test((i+1)%5); //检测右边邻居 可能抢饭

test((i+4)%5); //检测左边邻居 可能抢饭

V(mutex);

}

```
void test(int i){
```

自己

右邻居

左邻居

if(state[i]==HUNGRY && state[(i+1)%5]!=EATING && state[(i+4)%5]!=EATING) {

//若左右手的邻居均不在吃，则同时获得两把叉子；

state[i]=EATING; //state[i]=EATING表示同时获得两把叉子

V(s[i]);

s[i].value=0 $\mapsto 1$

不满

}

}

上
第

Philosopher i{

```
while(true){
```

```
    think();
```

```
    take_fork(i);
```

阻塞在 $S[i]$ 信号量 .

```
    eat();
```

```
    put_fork(i);
```

```
}
```

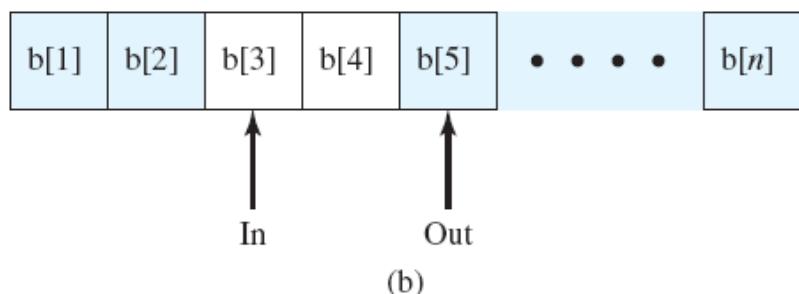
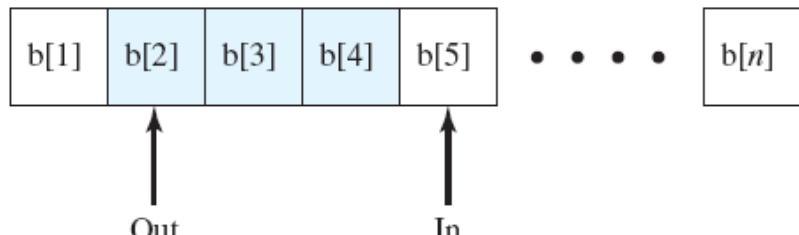
```
}
```

信号量实现同步

- 生产者-消费者问题
- 读者-写者问题
- 理发师问题

生产者—消费者问题

- 有限的大小为n的缓存空间，组织成环状
- p个生产者和q个消费者
- 每个生产者每次生产一件产品并放入缓存，如果缓存已满，则等待消费者消费后再放入；
- 每个消费者每次消费一个产品，如果缓存为空，则等待生产者放入产品



生产者-消费者问题分析

- 互斥要求
 - 禁止两个以上生产者同时将产品放入同一个位置
 - 禁止两个以上的消费者同时消费同一个产品
- 同步要求
 - 仅当缓冲区**产品数**大于0时，消费者才能消费产品，否则消费者必须等待；
 - 当生产者生产了一个产品时，将可用产品数加1，如果有消费者等待，则唤醒一个等待产品的消费者；
生产者 → 消费者
 - 仅当**空闲缓冲区数**大于0时，生产者可以放入产品，否则生产者必须等待；
 - 当消费者消费了一个产品后，将空闲缓冲区数加1，如果有生产者等待，则唤醒一个等待空闲缓冲区的生产者；

```

item B[n];
Semaphore empty; /*可用的空缓冲区个数*/
Semaphore full; /*可用的产品数*/
Semaphore mutex; /*互斥信号量*/
empty = n; full = 0; mutex = 1;
int in = 0; out = 0; /*in为放入缓冲区指针, out为取出缓冲区指针 */
    } > 同步.

```

Process producer_i()

```

while(true){
    item product = produce();

```

P(empty); 空闲缓冲区 70 不能

P(mutex);

B[in]=product; **临界区**

in = (in+1)%n;

V(mutex);

V(full); 生产完之后还有一个唤醒
一个消费者

}

Process consumer_i()

如果将P操作的顺序
交换, 会出现什
么情况?

P(full);

P(mutex);

P(empty);

Item product = B[out];

out = (out+1)%n;

V(mutex);

V(empty);

consume(product);

}

```

item B[n]; //缓冲区
Semaphore empty; /*可用的空缓冲区个数*/
Semaphore full; /*可用的产品数*/
Semaphore pmutex, cmutex; /*互斥信号量*/
empty = n; full = 0; pmutex = 1; cmutex=1;
int in = 0; out = 0; /*in为放入缓冲区指针, out为取出缓冲区指针 */

```

一个生产者放,允许另一个消费者取.

Process producer_i()

```

while(true){
    item product = ...;
    P(empty);
    P(pmutex);
    B[in]=product;
    in = (in+1)%n;
    V(pmutex);
    V(full);
}

```

Process consumer_i()

```

e(true);
    ...;
    if (in == out) {
        V(mutex);
        consume(product);
    }
    else {
        V(cmutex);
        B[out] = product;
        out = (out+1)%n;
        V(cmutex);
    }
}

```

仅当 $in==out$ 时, producer 和 consumer 才共享 buffer 数据,
 但此时,
 要么 $empty=0$, 要么 $full=0$;
 因此, 该方法可行。

在生产者放产品的同时, $V(empty)$;
 消费者可以消费产品,
 而若采用同一 mutex 则不行

读者-写者问题

- 有两组并发进程，读者和写者，共享文件F，要求
 - 允许多个读者同时对文件执行读操作；✓
 - 只允许一个写者对文件执行写操作；
 - 任何写者在完成写操作之前不允许其他读者或写者工作；
 - 写者在执行写操作前，应让已有的写者和读者全部退出。

① 读者写者问题—读者优先

- 当存在读者时，写者将被延迟
 - 且只要有一个读者活跃，随后而来的读者都将被允许访问文件
 - 可能造成写者的饥饿
- 后来的 R 可能先与 W 执行

读者写者问题—读者优先

- 互斥需求
 - 禁止多个写者同时对文件进行写操作
 - 禁止读者和写者同时工作
- 引入写锁
 - 写者必须获得写锁才能进行写操作；
 - 第一个读者也必须获得写锁才能进行读操作，后续读者无需获得写锁可以直接读；
- 同步需求
 - 当最后一个读者结束后，释放写锁，如果存在等待写锁的写者，则唤醒之；
 - 当写者结束后，释放写锁，如果存在等待写锁的用户（其它写者或第一个读者），则唤醒之；

Int readcount = 0; 计算有几个读进程 / 读进程计数器 */ $\xrightarrow{0 \rightarrow 1}$ $\xrightarrow{1 \rightarrow 2}$ ~~保护 readcount~~

Semaphore writelock, mutex; /*writelock为写锁, mutex为互斥信号量*/

Writelock=1; mutex =1 ;

Process reader_i(){

P(mutex); /*保证不同的读者互斥访问 readcount 共享变量; */

readcount++; /*读进程个数加1; */

if(readcount==1) /*当只有一个读进程时, 获得写锁, 阻塞写尝试*/

P(writelock);

V(mutex); /*释放互斥信号量*/

read(); /*如为第一个读进程, 则已经获得写锁, 否则, 表明已有读进程在读, 可以进行读操作 */

P(mutex); /*保证不同的读者互斥访问 readcount 共享变量; */

readcount--; /*读进程个数减1 */

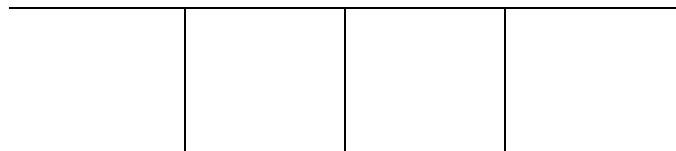
if(readcount==0) /*若读进程个数为0, 则可以唤醒写进程 */

V(writelock); 最后一个读者释放写锁

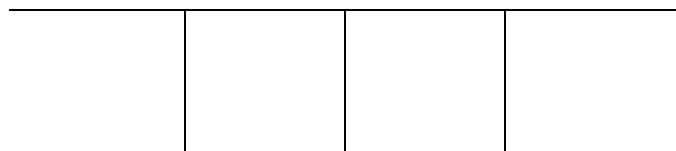
V(mutex); /*释放互斥信号量 */

}

```
Process writer_i()
{
    P(writelock); /*尝试获得写锁 ;*/
    write();          /*写操作 */
    V(writelock); /*释放写锁 */
}
```



mutex队列



writelock队列

写锁被取了

- (1) 当系统中已有进程在读或写时, 写进程阻塞在writelock队列
- (2) 当系统中已有进程在读时, 后续读进程不会被阻塞:
- (3) 当系统中已有进程在写时, 第一个读进程阻塞在writelock队列①, 后续读进程阻塞在mutex队列; ②

② 读者写者问题—写者优先

- 当一个写进程声明想进行写操作时，不允许新的读者访问文件，即所有之后发生的读请求将在该写操作之后进行
- 当后续写者到达时，只要系统中有写者在工作，则后续写者也将优先于系统中已到达的读者被服务

```
Int readcount = 0; /*读者计数器*/  
int writecount=0; /*写者计数器*/  
semaphore writelock = 1; /*写锁*/  
semaphore rdcntmutex = 1; /*读者计数器的互斥信号量*/  
semaphore wrtcntmutex = 1; /*写者计数器的互斥信号量*/  
semaphore queue = 1; /*排队信号量，用于实现写者优先*/
```

写进来以后，把谁谁在后面？？？

Process writer_i0 {

P(wrtcntmutex); /*获取写者计数器的互斥信号量*/

if(writecount==0)

P(queue);

堵读者

/*若为第一个写者，则尝试获取排队信号量，从而后续读者进程将被阻塞在queue队列，而后续写者将被允许进入*/

writecount++; /*写者计数器加1*/

V(wrtcntmutex); /*释放写者计数器互斥信号量 */

P(writelock); /*获得写锁，当存在读者时等待现有读者完成；当存在写者时，
等待写者完成；未获得写锁的写者将被阻塞在该信号量队列；
*/

/* write(); */

V(writelock); /*释放写锁*/

P(wrtcntmutex); /*获得写者计数器互斥信号量 */

writecount--; /*写者计数器减1*/

if(writecount==0) **最后一个写者**

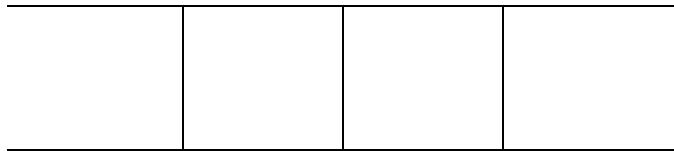
V(queue); /*如果不存在写者，则从queue等待队列中释放一个读者*/

V(wrtcntmutex); /*释放写计数器互斥信号量*/

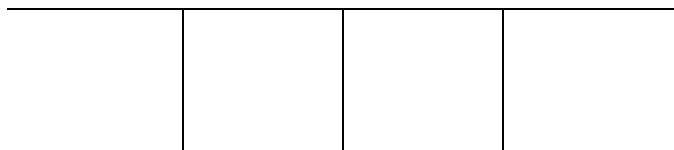
}

Process reader_i()

```
P(queue); /*尝试获得排队信号量，若已有写者，则读者将被阻塞在该信号量*/ 成功表明系统无写者  
P(rcdcntmutex); /*获取读计数器互斥信号量*/  
readcount++; /*读者计数器加1*/  
if(readcount==1)  
    P(writelock); /*若为第一个读者，则获取写锁，以防止后续写者和读者同时工作； */  
V(rcdcntmutex); /*释放读者计数器互斥信号量*/  
V(queue); /*释放queue信号量，按先来先到的方式选择queue信号量上等待的读者或写者*/  
/* read(); */ 读之前要释放 queue -  
P(rcdcntmutex); /*获取读者计数器互斥信号量*/  
readcount--; /*读者计数器减1 */  
if(readcount==0)  
    V(writelock); /*若为最后一个读者，则释放写锁*/  
V(rcdcntmutex); /*释放读者计数器互斥信号量*/  
}
```



queue队列



writelock队列

- (1) 当系统中存在写者时, 所有之后来的读者都被阻塞在queue队列;
- (2) 当读者进程先于写者到达, 若在开始读之前写者到达, 则写者被临时阻塞在queue队列 (一旦读者开始读, 则写者将被移到writelock队列) ;
- (3) 若在读者开始读之后写者到达, 则写者被阻塞在writelock队列 ✓
- (4) 当系统中已有进程在写时, 所有的写者进程将阻塞在writelock队列

理发师问题

- 理发店里有一位理发师、一把理发椅和n把供等候理发的顾客休憩的椅子；
- 如果没有顾客，理发师便在理发椅上睡觉，当有顾客到来时，他唤醒理发师；
- 如果理发师正在理发时又有新的顾客到来，那么如果还有空椅子，顾客就坐下来等待，否则就离开理发店。

Semaphore customer = 0; /* 等候理发的顾客数，用于阻塞理发师进程，初始值为0 */

Semaphore barbers = 0; /* 正在等候顾客的理发师数, 用于阻塞顾客进程，初始值为0 */

Semaphore mutex = 1; /* 互斥信号量 */

int waiting = 0; /* 等待理发的顾客坐的椅子数 */

int CHAIR = N; /* 为顾客准备的椅子数 */

```
process barber() {
    while(true) {
        P(customers); /* 判断是否有顾客,若没有则理发师等待*/
        P(mutex);      /*若有顾客, 获取互斥信号量*/
        waiting--;    /*等待人数减1*/
        V(barbers);   /*准备为顾客理发,解除一个阻塞顾客*/
        V(mutex);     /*释放互斥信号量*/
        cut_hair();   /*理发*/
    }
}
```

```
process customer() {
    P(mutex);      /*获取互斥信号量*/
    if(waiting<CHAIRS){ /*若等待人数小于N*/
        waiting++;    /*等待人数加1*/
        V(customers); /* 唤醒理发师*/
        V(mutex);     /*释放互斥信号量*/
        P(barbers);   /*若理发师忙, 则等待;否则, 申请并占用理发师资源*/
        get_haircut(); /*占用理发师资源, 可以理发*/
    }
    else
        V(mutex);     /*释放互斥信号量,人满, 顾客离开*/
}
```

Posix 信号量

- 信号量包含在头文件 <semaphore.h> 中
- 无名信号量
 - 可用于共享内存情况，如各个线程之间的互斥与同步
- 命名信号量
 - 通常用于不共享内存的情况下，如不共享内存的进程之间的互斥与同步

Posix 无名信号量

- 初始化:

[输出]
int sem_init(sem_t *sem, int pshared, unsigned value); 赋初值

- 销毁:

Int sem_destroy(sem_t *sem);

- PV操作

P操作: int sem_wait(sem_t *sem);

V操作: int sem_post(sem_t *sem);

Posix 命名信号量

- 创建或打开一个命名信号量：

sem_t *sem_open(const char *name, int oflag);

与文件关联

- 关闭命名信号量：

int sem_close(sem_t *sem);

关闭信号量并不能将信号量从系统中删除

- 删除信号量：

int sem_unlink(const char *name);

解除连接

管程

每个会产生死锁

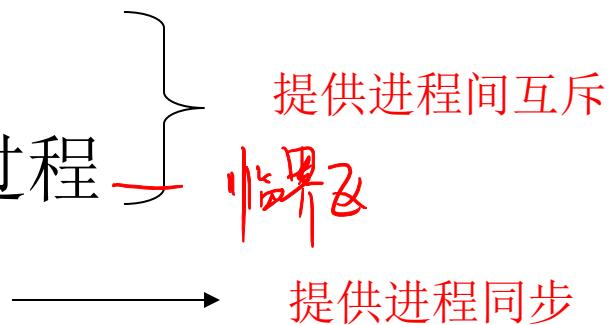
- 信号量提供了实现进程间互斥和协同的强有力工具。但是，信号量分散在程序各个地方，采用信号量编写正确的程序难度较大。
- 管程：
 - 把分散在各个进程中的临界区集中起来管理，并**把共享资源用数据结构抽象地表示出来**。
 - 由于临界区是访问共享资源的代码段，建立一个管程来管理到来的访问。
 - 管程每次只让一个进程来访**，这样既便于对共享资源进行管理，又能实现互斥访问。

管程的表示

类/结构体

- 管程是软件模块，包括：

- 局部于自己的共享变量
- 一组用于访问共享变量的过程
- 条件变量



管程

- 管程实现互斥
 - 通过防止对一个资源的并发访问，达到实现临界区的效果，提供互斥机制
 - 管程实现同步
 - 需要引入同步工具使得进程在资源不能满足而无法继续运行时被阻塞，同时还需开放管程控制权
 - 采用**条件变量**，让等待的进程临时放弃管程控制权，然后在适当时刻再尝试检测管程内的状态变化
- B₁ 已经进入管程但阻塞
- B₂ 唤醒B₁??

条件变量

- 条件变量是出现在管程内的一种数据结构，只有在管程中才能被访问，它对管程内的所有过程是全局的，只能通过两个原语操作来控制它
 - 条件变量的原语操作
 - cwait(x)
 阻塞 放入等待队列
 • 在条件变量x上挂起调用进程并释放管程，直到另一个进程在条件变量上执行csignal(x) 同一个条件变量上
 - csignal(x)
 • 如果有其它进程因对条件变量x执行cwait(x)而被挂起，便释放其中的一个等待进程；如果没有进程在等待，则csignal()操作没有任何效果，即x变量的状态没有改变 与pv不同
- 注：与信号量中的V操作有区别，V操作总是会改变信号量的状态

没有值

关于csignal()

(P及并行) ×

唤醒队列

- 假设进程P执行csignal(x)，且存在进程Q等待条件变量x，则如果阻塞进程Q被允许立即执行，则P必须等待，否则，P、Q将同时访问管程，违反了管程的互斥访问性。
- 但概念上讲，P和Q都应该可以继续执行
- 两种方案
 - P等待，直至Q离开管程或因等待另一个条件变量而开放管程
 - Q等待，直至P离开管程或因等待另一个条件变量而开放管程

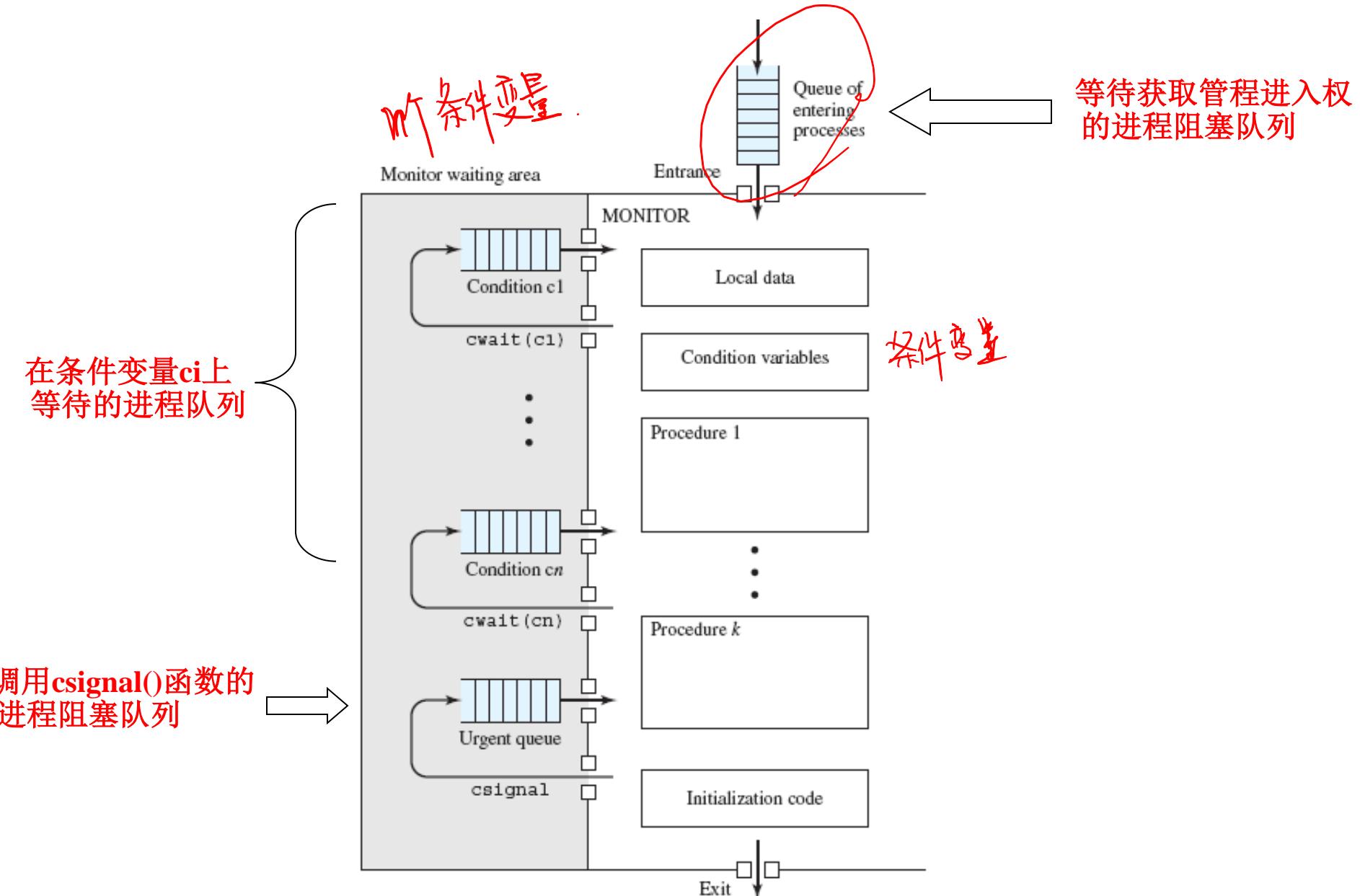


Figure 5.15 Structure of a Monitor

/* 管程解决生产者-消费者问题 */

monitor boundedbuffer{

/* shared variables */

char buffer[N];

int nextin=0, nextout=0; 下标

int count=0; 可用产品数

/* condition variables */

condition notfull, notempty;

→ 等待队列

void append(char x) 生产

```
{  
    if (count == N) cwait(notfull); /*若缓冲已满，则阻塞自己，等待notfull条件产生*/  
    buffer[nextin] = x;  
    nextin = (nextin+1) % N;  
    count++;  
    /*通知了，唤醒在不空上等待的进程*/  
    csignal(notempty); /*如有进程等待notempty条件，则唤醒其中的一个*/  
}
```

void take(char& x) 消费

```
{  
    if(count == 0) cwait(notempty); /*如果缓冲为空，则阻塞自己，等待notempty条件产生*/  
    x = buffer[nextout];  
    nextout = (nextout+1) % N;  
    count --;  
    /*通知了，唤醒在不满上等待的进程*/  
    csignal(notfull); /*如果有进程等待notfull条件，则唤醒其中的一个*/  
}
```

boundedbuffer bd;

共享缓冲区

由线程封装好！

process producer()

{

char x;

while(true){

 x = produce();

 bd.append(x);

}

}

process consumer()

{

char x;

while(true){

 bd.take(x);

 consume(x);

}

}

```

/*管程解决哲学家进餐问题*/
monitor dp{
    enum{THINKING, HUNGRY, EATING} state[5];
    condition self[5]; //用于阻塞哲学家i
    void pickup(int i){
        state[i] = HUNGRY;
        test(i);
        if(state[i]!=EATING) //state[i]变为EATING,表示其占用两把叉子，否则一把都不占用;
            cwait(self[i]); //一把叉子都不占用时，在自身条件变量上阻塞自己
    }
    void putdown(int i){
        state[i] = THINKING;
        test((i+4)%5); //看邻居节点是否具备EATING的条件;
        test((i+1)%5); //看
    }
}

void test(int i){
    if((state[(i+4)%5]!=EATING)&&(state[i]==HUNGRY)&&(state[(i+1)%5]!=EATING)){
        state[i]=EATING; //当i想拿起两把叉子，且邻居都没有处于EATING状态时,
       拿起两把叉子;
        csignal(self[i]); //唤醒一个在self[i]条件变量等待的哲学家
    }
}
initialization_code(){
    for(int i=0;i<5;i++)
        state[i]=THINKING;
}
}

```

没有互斥信号量了，
边风向

邻居不在吃自己
90

```
Philosophy_i(){  
    while(true)  
    {  
        think();  
        dp.pickup(i);  
        eat();  
        dp.putdown(i);  
    }  
}
```

Java中的管程

- 互斥访问
 - **synchronized**关键词 *加在方法前*
 - java为每个对象都设置一个monitor *不同对象不同管程*
 - Monitor确保了关联对象中**synchronized方法的互斥访问**, 即同一个对象, 同一时刻只有一个线程可以执行其上的synchronized方法
 - 若是在同一个类的不同对象上执行相同的synchronized方法, 则可以并发执行
- 同步:
 - **wait()**: *开放管程, 在给定对象上等待;*
 - **notify()**: 在给定对象的等待队列中唤醒一个线程;
 - **notifyAll()**: 唤醒给定对象等待队列中的所有线程, *所有被唤醒的线程竞争管程的拥有权, 即从条件变量的等待队列移到了管程入口的等待队列。一旦获得管程拥有权, 从wait()操作之后开始执行。*
 - 只有在获得管程对象所有权时, 才能执行wait(), signal(), signalAll()操作, 否则会抛出IllegalMonitorStateException *进入synchronized语句才行*

```

class Counter
{
    private int count = 0;    共享对象
    public void increment(){
        int n = count;    (在方法里, 成程有自己的栈空间) 局部变量
        count = n+1;
    }
}

```

如果两个线程共享一个对象Counter counter, 且都希望执行counter.increment()方法, 则会导致运行结果不唯一。

| Thread 1 | Thread 2 | Count |
|----------------------|----------------------|-------|
| counter.increment(); | --- | 0 |
| n = count; // 0 | --- | 0 |
| --- | counter.increment(); | 0 |
| --- | n = count; // 0 | 0 |
| --- | count = n + 1; // 1 | 1 |
| count = n + 1; // 1 | --- | 1 |

```

class Counter { 有线程
    private int count = 0;
    public void synchronized increment() {
        int n = count;
        count = n+1;
    }
}

```

| Thread 1 | Thread 2 | Count |
|---|---|-------|
| counter.increment(); | --- | 0 |
| (acquires the monitor) 线程1占有 | --- | 0 |
| n = count; // 0 | --- | 0 |
| --- | counter.increment(); | 0 |
| --- | (can't acquire monitor) | 0 |
| count = n + 1; // 1 | ---(blocked) 在管程内部阻塞 | 1 |
| (releases the monitor) | ---(blocked) | 1 |
| --- | (acquires the monitor) | 1 |
| --- | n = count; // 1 | 1 |
| --- | count = n + 1; // 2 | 2 |
| --- | (releases the monitor) | 2 |

```

class Buffer {
    private char [] buffer;
    private int count = 0, in = 0, out = 0;
    Buffer(int size) {
        buffer = new char[size];
    }
    public synchronized void Put(char c) {
        while(count == buffer.length) { //如果改为if会如何?
            try {
                wait(); // this.wait()
            } catch (InterruptedException e) { }
            finally { }
        }
        System.out.println("Producing " + c + " ...");
        buffer[in] = c;
        in = (in + 1) % buffer.length;
        count++;
        this.notify(); //等待别的生产者同一个可能导致
    }
    public synchronized char Get() {
        while(count == 0) { //调用wait();
            try {
            } catch (InterruptedException e) { }
            finally { }
        }
        char c = buffer[out];
        out = (out + 1) % buffer.length;
        count--;
        System.out.println("Consuming " + c + " ...");
        this.notify(); //被唤醒的生产者从wait()开始执行，将覆盖
        return c; //未消费产品！
    }
}

```

读写同一个变量时必须添加 sync 语句

- 如果while改为if，则考虑下述情形：
 - 假如count=n;
 - 则可能存在多个生产者阻塞在Buffer类对象上
 - 当一个消费者消费了一个产品后，唤醒某个生产者
 - 该生产者将产品放入后，调用notify()操作，则会唤醒其它等待的生产者
 - 被唤醒的生产者从wait()调用后开始执行，将覆盖未消费产品！

synchronized (对象)

进程间通信

- 信号通信(signal) **回调函数?**
- 管道通信(pipe)
- 消息传递(message passing)
- 信号量(semaphore)
- 共享主存(shared memory)

信号通信

- 信号是一种软中断，通过发送指定信号来通知进程某个**异步事件**的发生，迫使进程执行信号处理函数
- 信号通信不能传输数据，能力较弱
- 信号通信机制
 - `sigaction()` 安装**信号处理程序**
 - 信号发送
 - 用户按键：如`ctrl+C`
 - 信号发送函数：如`kill()` **发信号给程序**
 - 信号检测和响应
 - 若收到信号，则转入信号处理程序

```

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#define PROMPT "do you want to terminate the program"
char *prompt=PROMPT;

/*按下ctrl+C的信号处理函数 */
void ctrl_c_handler(int signo){
    write(STDERR_FILENO, prompt, strlen(prompt));
}

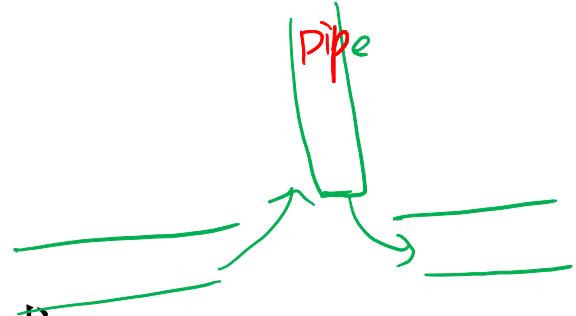
int main(){
    struct sigaction act;
    act.sa_handler=ctrl_c_op;      //设置信号处理函数;自己定义
    sigemptyset(&act.sa_mask);   //安装进单
    act.sa_flags=0;
    if(sigaction(SIGINT, &act, NULL)<0){ //安装信号处理函数;
        fprintf(stderr, "Install Signal Action Error\n");
        exit(1);
    }
    while(1);                  //循环， 等待用户按下ctrl+c;
}

```

改变 ctrl+c 信号的处理方式

buffer管道
(生产消费者)

管道通信



- 管道是UNIX的传统进程通信方式
- 管道是连接读写进程的一个特殊文件
 - 发送进程将管道文件视作输出文件
 - 接收进程将管道文件视作输入文件
- 管道的实质是一个共享文件, 可借助于文件系统的机制来实现
- 但是, 读写进程之间的协作单靠文件系统无法解决, 必须做到:
 - 进程正在使用管道写入或读取数据时, 另一个进程必须等待;
 - 发送者和接收者必须知道对方是否存在;
 - 发送信息和接收信息之间要实现正确的同步关系

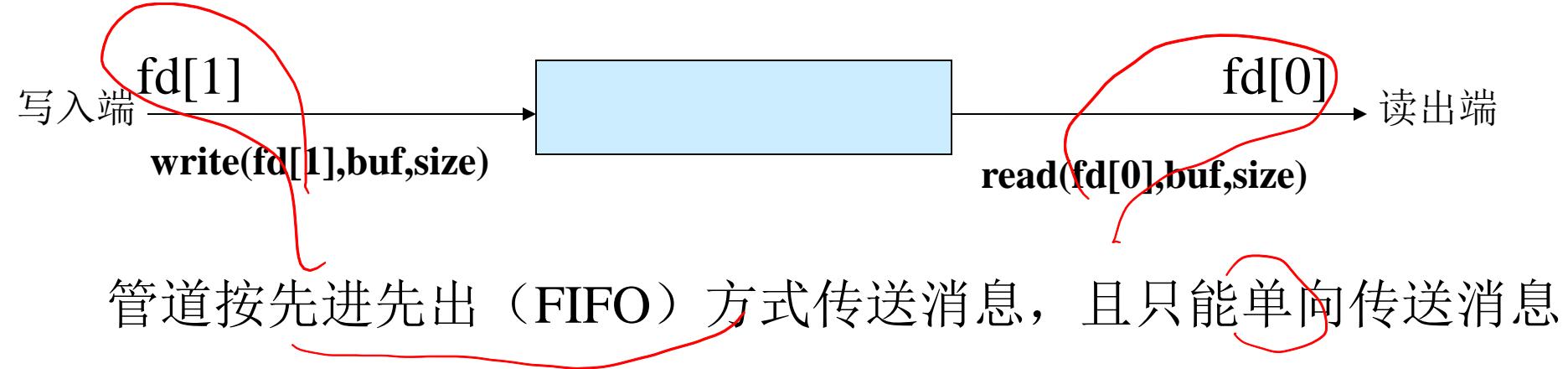
管道通信

- 匿名管道
- 命名管道

匿名管道

- 只能用于具有亲缘关系的进程间通信，如父子进程或兄弟进程；
✓
 - 匿名管道对于通信两端的进程而言，就是一个文件，可以使用一般的文件I/O函数，如**read()**, **write()**;
 - 一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区末尾，每次都从缓冲区的头部读出数据。
 - 系统调用原型：**int pipe(int fd[2]);**
 - 返回0，表示成功
 - 返回 -1， 表示失败
- 2个文件描述符 (返回)
一个读一写

pipe (fd)



管道文件读写操作的同步与互斥

- 对管道文件进行读写操作过程中，要对发送进程和接收进程实施正确的同步与互斥以确保通信的正确性：
 - 接收进程：当接收进程读pipe时，若发现pipe为空，则进入等待状态。一旦有发送进程对该pipe执行写操作时唤醒等待进程。
 - 发送进程：当发送进程在写pipe时，总是先按pipe文件的当前长度设置，如果pipe文件长度已经到规定长度，但仍有一部分信息没有写入，则系统使要求写pipe的进程进入睡眠状态，当读pipe进程读走了全部信息时，系统再唤醒待写的进程。它将余下部分信息继续送入pipe中。

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>
#define MAX_LINE 180
int main()
{
    int thePipe[2], ret;
    char buf[MAX_LINE+1];
    const char *testbuf = "a test string";
    if(pipe(thePipe)==-1)          /*创建匿名管道*/
        if(fork()==0){           /*子进程*/
            ret = read(thePipe[0], buf, MAX_LINE); /*从管道中读数据*/
            buf[ret]=0;           收尾
            printf("Child read %s\n", buf);
        }
        else{                   /*父进程*/
            ret = write(thePipe[1], testbuf, strlen(testbuf)); /*向管道写入数据*/
            ret = wait(NULL);   /*等待子进程结束*/
        }
    }
    close(thePipe[0]);
    close(thePipe[1]);
    return 0;
}

```

父子进程通信

fork() 创建独立地址空间
进程

Linux 匿名管道进程通信例子

命名管道

文件系统是可见的

- 可用于任何可以访问命名文件的两个进程之间的通信
- 系统调用原型：
– int mkfifo(const char *filename, mode_t mode);
– mode与文件创建系统调用open()中的mode含义一致
- 命名管道操作
– open(): 打开管道 ✓
– read(): 读管道 ✓
– write(): 写管道 ✓
– 命名管道是先进先出的，不支持lseek等文件定位操作

命名管道

- 实例
 - 服务器进程接收客户进程的请求，并打印出客户的进程标识符和发送的命令
 - 客户进程接收用户输入的命令，并向服务器进程发送

```
/*fifo.h*/
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#define FIFO_PATH "/home/stu/fifo_abc" //命名管道的名字
struct fifo_cmd {
    pid_t pid;           //存放发送进程的进程标识符
    char cmd[100];       //发送进程发送给服务器的命令
};
```

与文件关系

```

/*命名管道通信服务器端 server.c*/
#include "fifo.h.h"
int main(void) {
    int fd,err,n;
    struct fifo_cmd cmd;
    if((err = mkfifo(FIFO_PATH, 0666)) < 0){  

        if(err != EEXIST) {  

            perror("mkfifo fail");  

            exit(-1);  

        }  

    }  

    if((fd = open(FIFO_PATH, O_RDONLY)) < 0){  

        perror("open fail");  

        exit(-1);  

    }  

    while(1){  

        if((n = read(fd, &cmd, sizeof(cmd))) < 0) {  

            perror("read fail");  

            exit(-1);  

        }  

        if(n > 0){  

            printf("command from process %d: %s\n", cmd.pid, cmd.cmd);  

        }
        sleep(1);
    }
}

```

```

/*命名管道通信客户端 client.c*/
#include "fifo.h.h"
int main(int argc, char* argv[]) {
    int fd;
    struct fifo_cmd cmd;
    if((fd = open(FIFO_PATH, O_WRONLY)) < 0) {
        perror("open fail");
        exit(-1);
    }
    cmd.pid = getpid();
    while(1) {
        printf("%Please enter a command string: ");
        fgets(cmd.cmd, sizeof(cmd.cmd), stdin);
        cmd.cmd[strlen(cmd.cmd) - 1] = 0; // 截尾
        if(write(fd, &cmd, sizeof(cmd)) < 0) {
            perror("write fail");
            exit(-1);
        }
    }
}

```

同一机器通讯

2个管道

消息传递

- 信号量及PV操作属于低级通信方式，不便于使用，局限性大
- 消息传递属于高级通信方式，由操作系统隐藏实现细节，便于使用。
- 消息传递需要操作系统的参与
 - 进程之间地址空间隔离
 - 适合于少量数据传递
- 无需应用程序显式地避免冲突访问



消息传递

- 两条原语

Send(destination, message)

可以不指定到个人
(灵活)

Receive(source, message)

- 寻址

- 直接寻址

- send 直接给出目标进程的标识

- Receive 可以指定发送进程的标识，也可以不指定。

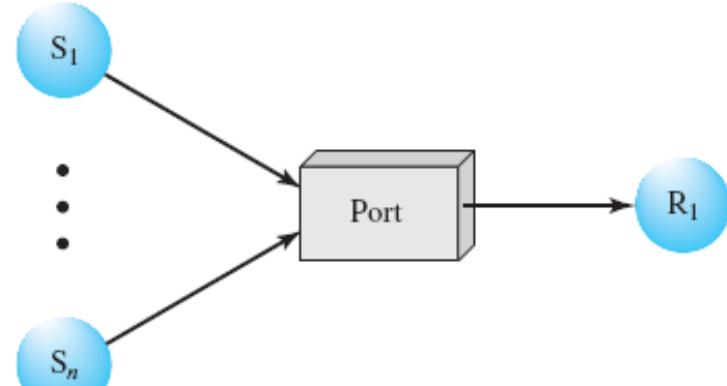
- 间接寻址

- 消息发送到共享的mailbox中

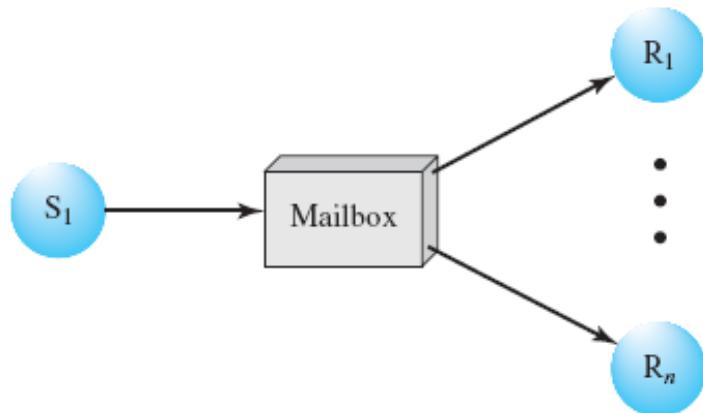
- 可以实现多种通信，较为灵活



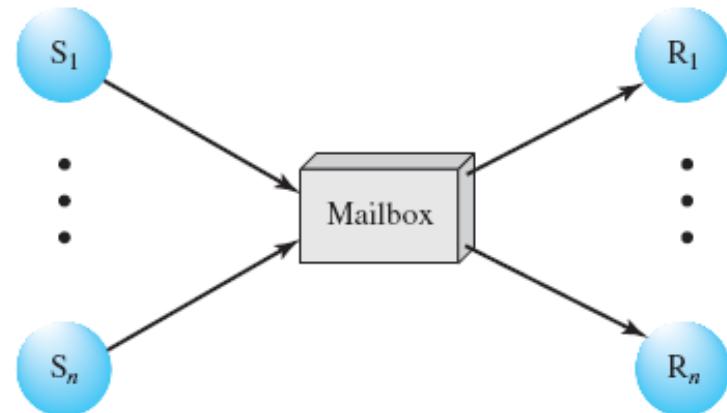
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

Figure 5.18 Indirect Process Communication

1对多

生产-消费

基于间接寻址的消息传递

- 发送信件: `send(A, message)`
 - 若指定信箱A未满，则将信件送入信箱中由指针所指示的位置，并释放一个**等待信件**的接收者；
 - 否则，发送信件者被置成等待信箱的状态；
- 接收信件: `receive(A, message)`
 - 若指定信箱A中有信件，则取出一封信件，并释放一个**等待信箱**的发送者；
 - 否则，接收信件者被置成等待信件的状态

消息传递实现互斥

```
/* program mutualexclusion */
const int n = /* number of processes */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);          //从信箱中接收信件，若无信件，则等待;
        /* critical section */;
        send (box, msg);            //处理完临界区代码，向信箱发送消息，唤醒一个等待信箱进程;
        /* remainder */;
    }
}

void main()
{
    create_mailbox (box);         //创建信箱
    send (box, null);             //向信箱中发送一封内容为空的信件
    cobegin(P(1), P(2), ..., P(n)); //进程开始并发执行
}
```

消息传递机制实现进程同步

```
const int capacity;
int i;
void producer(){
    message pmsg;
    while(true){
        receive(mayproduce, pmsg); //取到队列生产 . . . . . notfull
        pmsg = produce();
        send(mayconsume, pmsg);
    }
}
void consumer(){
    message cmsg;
    while(true){
        receive(mayconsume, cmsg); // . . . . . notempty
        consume(cmsg);
        send(mayproduce, null);
    }
}
void main(){
    create_mailbox(mayproduce);
    create_mailbox(mayconsume);
    for(i = 0; i<capacity; i++) send(mayproduce, null); //最多生产capacity个产品
    parbegin(producer, consumer);
}
```

System V消息队列

- 创建消息队列
 - ~~int msgget(key_t key, int flag);~~
 - 若调用成功，返回消息队列标识
- 向消息队列发送消息
 - ~~int msgsnd(int msgid, struct msgbuf *msgp, size_t size, int flag);~~
- 从消息队列接收一个消息
 - ~~ssize_t msgrcv(int msgid, void *msgp, size_t size, long msgtype, int flag);~~
- 在消息队列上执行指定的操作 *comand*
 - ~~int msgctl(int msgid, int cmd, struct msgid_ds *buf);~~
- 参数说明
 - msgid为msgget返回的消息队列标识
 - msgp为指向消息缓冲区的指针，用于暂时存储发送和接收的消息
 - msgsize指消息的大小
 - msgtype指消息类型

并不能直接设置

System V消息队列

- 在System V的IPC机制(消息队列、共享内存、信号量)实现中，需要用ftok函数创建一个key_t类型的值，并作为相应的标识；
- `key_t ftok(char *fname, int id);`
- 一般的UNIX实现中，是将文件的索引节点号取出，前面加上子序号id得到key_t的返回值。
 - `ls -i` 命令可以查看一个节点的inode号

```
/*msgqueue.h*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <error.h>
#define BUFFER 10
#define MSG_PATH "/home/stu"
#define PRIVATE_KEY 10001
struct msgbuf{
    long mtype;
    char buf[BUFFER];
};
```

```

/*sender.c*/
#include "msgqueue.h"
int main(){
    key_t msgkey;
    int msgid;
    struct msghdr msg; // 保留下 receiver 上下文
    msgkey = ftok(MSG_PATH, PRIVATE_KEY); // 消息队列
    if(msgkey == -1) { perror("ftok error"); exit(-1); }
    msgid = msgget(msgkey, IPC_CREAT|0666);
    if(msgid == -1) { perror("msgget error"); exit(-1); }
    int i = 0;
    while(i<10){
        memset(msg.buf, 0, BUFFER);
        sprintf(msg.buf, "buf_0x%x", i);
        msg.mtype=1001;
        if( msgsnd(msgid, &msg, sizeof(struct msghdr), 0) <0 ) { perror("msgsnd"); exit(-1); }
        i++;
        sleep(1);
    }
    sleep(30);
    if( msgctl(msgid, IPC_RMID, 0) ==-1){ perror("msgctl error"); exit(-1); }
    return 0;
}

```

```

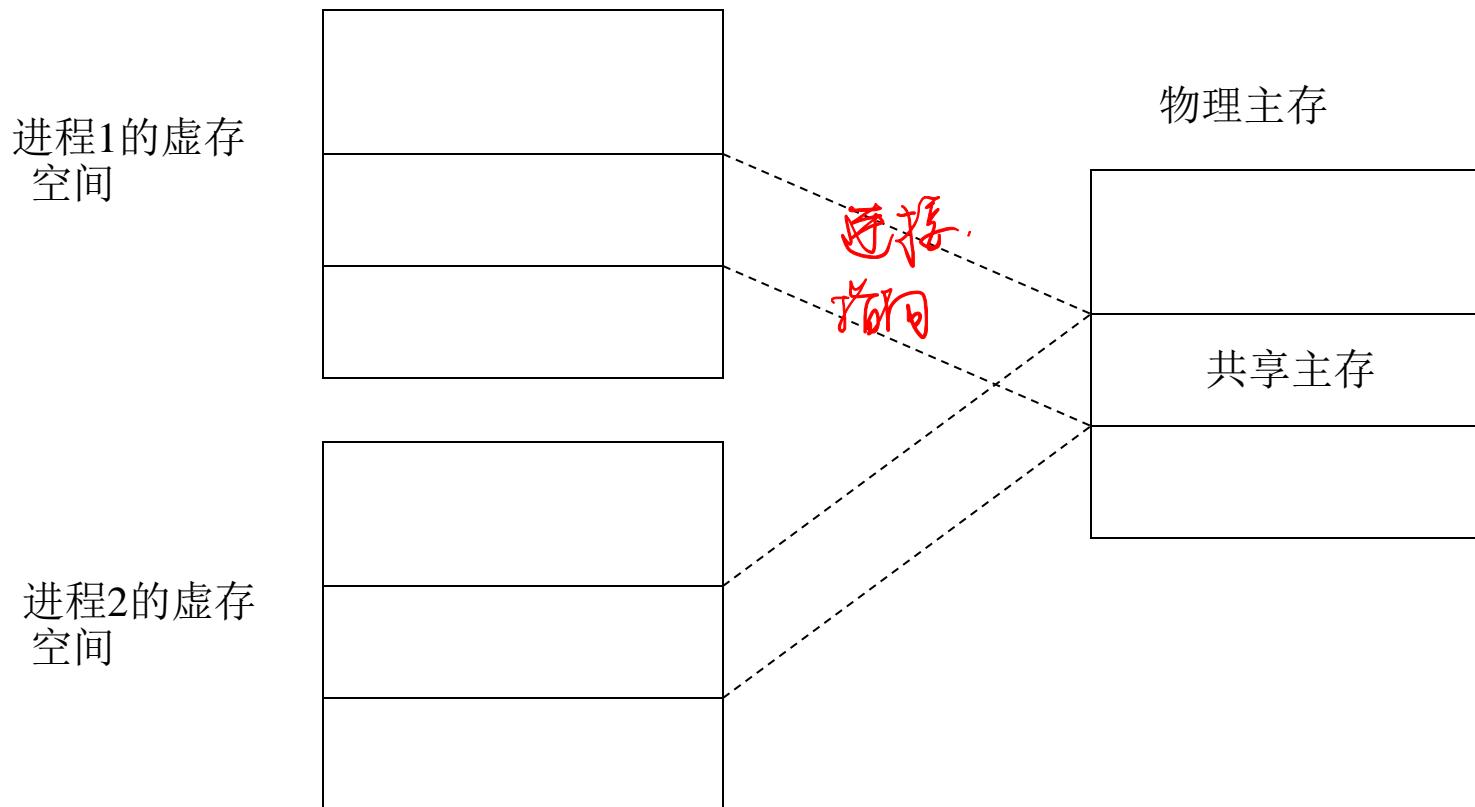
/*receiver.c*/
#include "msgqueue.h"
int main(){
    key_t msgkey;
    int msgid;
    struct msghdr msg;
    msgkey = ftok(MSG_PATH, PRIVATE_KEY);
    if(msgkey == -1){ perror("ftok error"); exit(-1); }
    msgid = msgget(msgkey, IPC_EXCL|0666) ;
    if(msgid== -1){ perror("msgget error"); exit(-1); }
    int i = 0;
    while(i<10){
        memset(msg.buf, 0, BUFFER);
        msg.mtype=1001;
        if( msgrcv(msgid, &msg, sizeof(struct msghdr), msg.mtype, 0) == -1){
            perror("msgrecv error");
            exit(-1);
        }
        printf("msg.buf = %s\n", msg.buf);
        i++;
        sleep(2);
    }
    return 0;
}

```

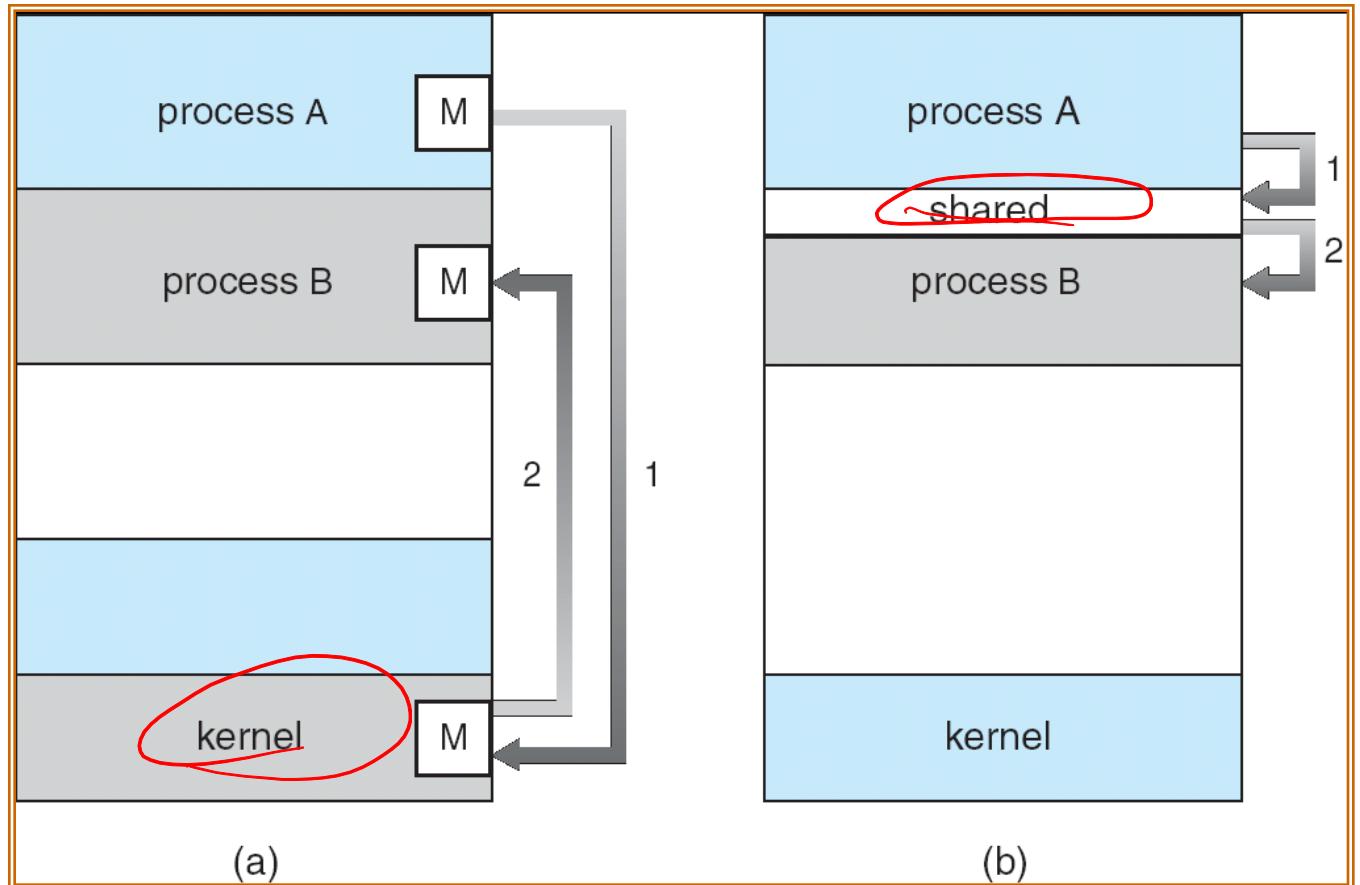
共享内存通信机制

- 建立一块共享内存区域
- 进程可以读/写共享内存的数据
- 适用于大量数据传输，速度快
- 需要应用程序显式地避免冲突

消息队列在内核空间



消息传递 vs 共享主存



消息传递模型



共享内存模型



共享内存解决生产者消费者问题

- 类似于vfork()系统调用，子进程和父进程共享内存
- 定义共享数据区

```
#define BUFFER_SIZE 10
Typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0; /*用于指出下一个空缓存区位置*/
int out = 0; /*用于指出第一个物品的位置*/
int numOfItems = 0;
```

生产者进程

✗

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

消费者进程



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

注：程序未考虑生产者/消费者进程并发访问共享内存

System V共享内存API实例

shared memory

- `shmget()`创建共享内存
- `shmat()`将共享内存连接到进程
attach
- `Shmdt()`将共享内存与进程解除连接
detach
- `Shmctl()`删除共享内存

```
/*shmtest.h*/  
  
#include <sys/IPC.h>  
  
#include <sys/shm.h>  
  
#include <sys/types.h>  
  
#include <unistd.h>  
  
typedef struct{  
  
    char name[4];  
  
    int age;  
  
} people;
```

```
/*testwrite.c*/
#include "shmtest.h"
void main(int argc, char** argv) {
    int shm_id,i;
    key_t key;
    char temp;
    people *p_map;
    char* name = "/dev/shm/myshm2";
    key = ftok(name,0);           //将共享内存地址映射到key上
    if(key == -1) {
        perror("ftok error");
        return;
    }
    shm_id=shmget(key,4096, IPC_CREAT); //create the shared memory;
    if(shm_id == -1) {
        perror("shmget error");
        return;
    }
    p_map=(people*)shmat(shm_id,NULL,0); //attach it to this process
    temp='a';
    for(i = 0;i<10;i++) {
        temp+=1;
        memcpy((*(p_map+i)).name,&temp,1); //写操作
        (*(p_map+i)).age=20+i;
    }
    if( shmdt(p_map) == -1) //detach the shared memory
        perror(" detach error ");
}
```

```

/*testread.c*/
#include "shmtest.h"
void main(int argc, char** argv) {
    int shm_id,i;
    key_t key;
    people *p_map;
    char* name = "/dev/shm/myshm2";
    key = ftok(name,0);
    if(key == -1) {
        perror("ftok error");
        return;
    }
    shm_id = shmget(key,4096,IPC_CREAT); //create the shared memory;
    if(shm_id == -1) {
        perror("shmget error");
        return;
    }
    p_map = (people*)shmat(shm_id,NULL,0);
    for(i = 0;i<10;i++) {
        printf( "name:%s\n",*(p_map+i).name );
        printf( "age %d\n",*(p_map+i).age );
    }
    if(shmdt(p_map) == -1)
        perror(" detach error ");
}

```

本章小结

- 进程的并发性使得进程的执行环境不再封闭，有可能造成与时间有关的错误，如结果不唯一、永远等待；
- 进程的关系包括竞争和协作两种关系，分别可以用进程互斥和同步加以解决；
- 软件方法、硬件方法可以实现进程互斥，但有其局限性；
- 信号量和PV操作能有效解决进行的互斥和同步；
- 管程是一种软件模块，用于将分散的信号量操作集中起来，能降低编程的复杂度；
- 进程间通信的方法包括信号通信、信号量通信、管道通信、消息传递通信、共享内存通信