计算机与操作系统
第六章 并发程序设计
6.4 管程

葛季栋
南京大学软件学院

6.4 管程

6.4.1 管程和条件变量

6.4.2 管程的实现

6.4.3 管程求解进程的同步与互斥问题

6.4.1管程和条件变量

为什么要引入管程?

- 把分散在各进程中的临界区集中起来进行管理
- 防止进程有意或无意的违法同步操作
- 便于用高级语言来书写程序

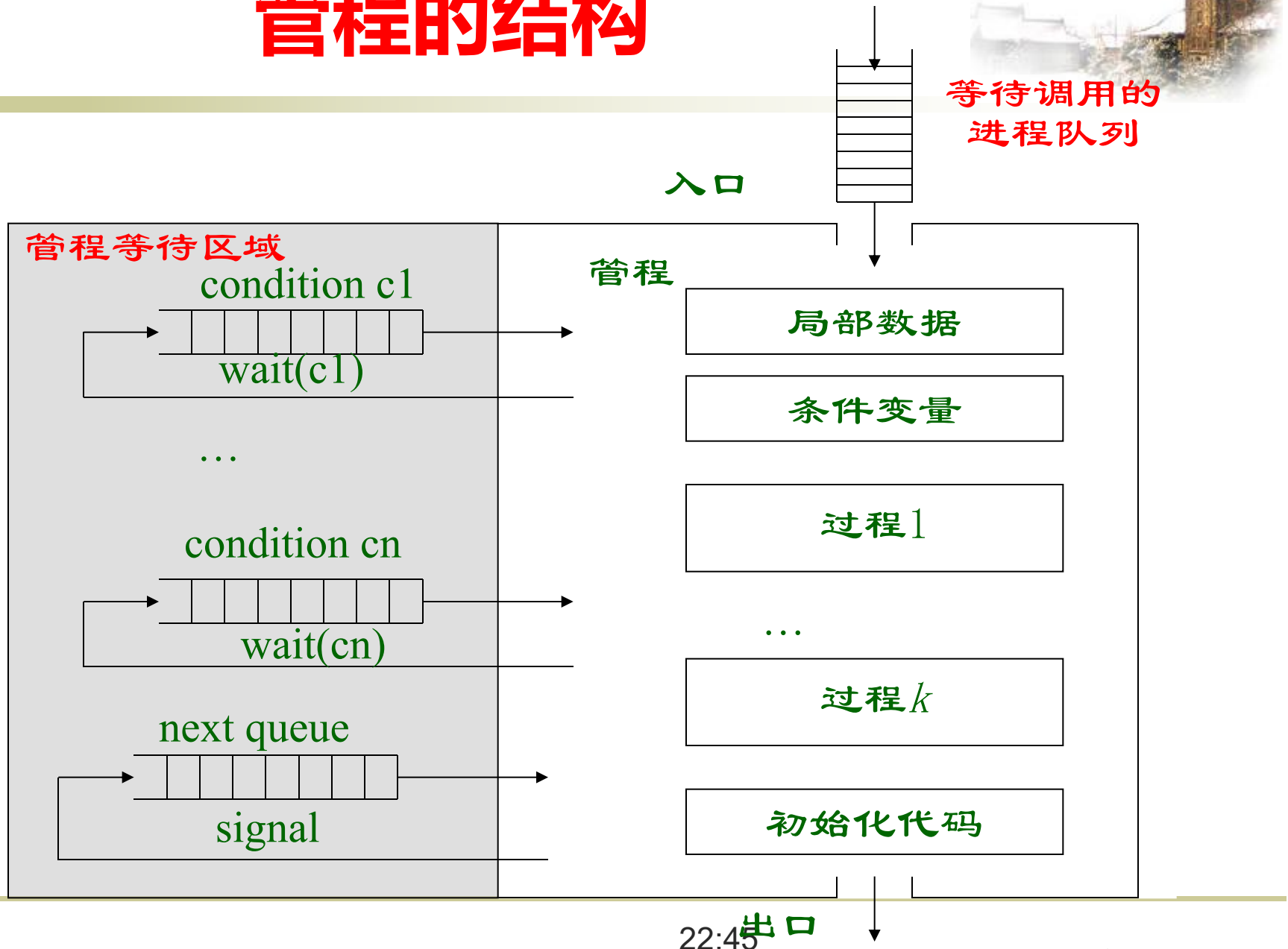
管程定义和属性

- 管程的定义
  - 管程是由局部于自己的若干公共变量及其说明和所有访问这些公共变量的过程所组成的软件模块
- 管程的属性
  - 共享性
  - 安全性
  - 互斥性

管程的形式

```
type 管程名=monitor{
  局部变量说明;
  条件变量说明;
  初始化语句;
define 管程内定义的，管程外可调用的过程或函数名列表;
use 管程外定义的，管程内将调用的过程或函数名列表;
过程名/函数名(形式参数表){
        <过程/函数体>;
  }
            …
过程名/函数名(形式参数表){
        <过程/函数体>;
  }
}
```

管程的结构


等待调用的
进程队列
入口
管程等待区域
condition c1
wait(c1)
...
condition cn
wait(cn)
next queue
signal
管程
局部数据
条件变量
过程1
...
过程k
初始化代码
出口

管程的条件变量

- 条件变量-是出现在管程内的一种数据结构，且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
- wait( )-阻塞调用进程并释放管程，直到另一个进程在该条件变量上执行signal( )
- signal( )-如果存在其他进程由于对条件变量执行wait( )而被阻塞，便释放之；如果没有进程在等待，那么，信号不被保存

管程问题讨论

- 使用signal释放等待进程时，可能出现两个进程同时停留在管程内。解决方法：
  - 执行signal的进程等待，直到被释放进程退出管程或等待另一个条件变量
  - 被释放进程等待，直到执行signal的进程退出管程或等待另一个条件
- 霍尔(Hoare, 1974)采用第一种办法
- 汉森(Hansen)选择两者的折衷，规定管程中的过程所执行的signal操作是过程体的最后一个操作

6.4.2 管程的实现 (Hoare方法)

- 霍尔方法使用P和V操作原语来实现对管程中过程的互斥调用，及实现对共享资源互斥使用的管理
- 不要求signal操作是过程体的最后一个操作，且wait和signal操作可被设计成可以中断的过程

C. A. R. Hoare

Dijkstra与Hoare的学术友谊堪比马克思与恩格斯
研究兴趣相同，学术理想相同
共同的目标：解决程序的正确性问题和易理解性问题

- C. A. R. Hoare（1934- ）
  - 昵称 Tony Hoare
  - 剑桥大学，微软研究院剑桥分院
  - 获1980年图灵奖
  - 曾经被英国女王赐予爵士爵位
- 学术贡献
  - 操作系统: 信号量与PV操作 --> Hoare管程
  - 算法: QuickSort
  - 进程代数: Communicating Sequential Processes,
  - 程序验证: Hoare Logic, etc.

Hoare管程数据结构(1)

1. mutex

- 对每个管程，使用用于管程中过程互斥调用的信号量mutex (初值为1)
- 进程调用管程中的任何过程时，应执行P(mutex)；进程退出管程时，需要判断是否有进程在next信号量等待，如果有(即next_count>0)，则通过V(next)唤醒一个发出signal的进程，否则应执行V(mutex)开放管程，以便让其他调用者进入
- 为了使进程在等待资源期间，其他进程能进入管程，故在wait操作中也必须执行V(mutex)，否则会妨碍其他进程进入管程，导致无法释放资源

Hoare管程数据结构(2)

2. next和next-count

- 对每个管程，引入信号量next(初值为0)，凡发出signal操作的进程应该用P(next)阻塞自己，直到被释放进程退出管程或产生其他等待条件
- 进程在退出管程的过程前，须检查是否有别的进程在信号量next上等待，若有，则用V(next)唤醒它。next-count(初值为0)，用来记录在next上等待的进程个数

Hoare管程数据结构(3)

3. x-sem和 x-count

- 引入信号量x-sem(初值为0)，申请资源得不到满足时，执行P(x-sem)阻塞。由于释放资源时，需要知道是否有别的进程在等待资源，用计数器x-count(初值为0)记录等待资源的进程数
- 执行signal操作时，应让等待资源的诸进程中的某个进程立即恢复运行，而不让其他进程抢先进入管程，这可以用V(x-sem)来实现

Hoare管程数据结构(4)

每个管程定义如下数据结构：

```
typedef struct InterfaceModule { //InterfaceModule是结构体名字
    semaphore mutex;  //进程调用管程过程前使用的互斥信号量
    semaphore next;   //发出signal的进程阻塞自己的信号量
    int next_count;   //在next上等待的进程数
};
mutex=1; next=0; next_count=0;  //初始化语句
```

Hoare管程的enter( )和leave( )操作

```
void enter(InterfaceModule &IM) {
    P(IM.mutex); //判断有否发出过signal的进程?
}
```

```
void leave(InterfaceModule &IM) {
   if (IM.next_count>0)
       V(IM.next);  //有就释放一个发出过signal的进程
   else
          V(IM.mutex); //否则开放管程
}
```

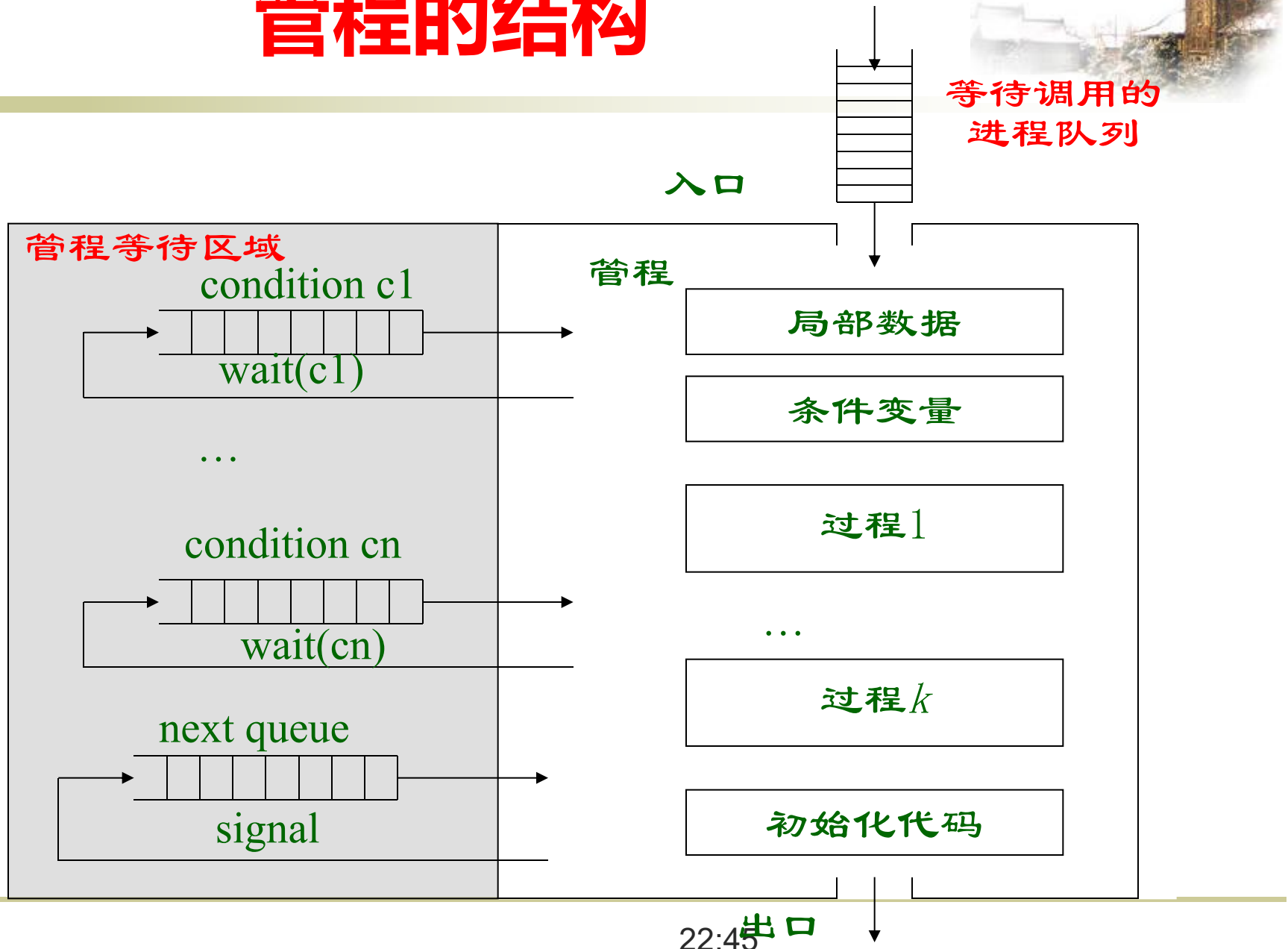
Hoare管程的wait( )操作

```
void wait(semaphore &x_sem,int
              &x_count,InterfaceModule &IM) {
  x_count++; //等资源进程个数加1，x_count初始化为0
  if (IM.next_count>0) //判断是否有发出过signal的进程
      V(IM.next); //有就释放一个
  else
      V(IM.mutex); //否则开放管程
  P(x_sem); //等资源进程阻塞自己，x_sem初始化为0
  x_count--; //等资源进程个数减1
}
```

Hoare管程的signal( )操作

```
void signal(semaphore &x_sem,int
          &x_count,InterfaceModule &IM) {
 if(x_count>0) { //判断是否有等待资源的进程
   IM.next_count++; //发出signal进程个数加1
   V(x_sem);    //释放一个等资源的进程
   P(IM.next);   //发出signal进程阻塞自己
   IM.next_count--; //发出signal进程个数减1
    }
}
```

管程的结构


等待调用的
进程队列
入口
管程等待区域
condition c1
wait(c1)
...
condition cn
wait(cn)
next queue
signal
管程
局部数据
条件变量
过程1
...
过程k
初始化代码
出口

```
typedef struct InterfaceModule { //InterfaceModule是结构体的名字
        semaphore mutex;          //进程调用管程过程前使用的互斥信号量
        semaphore next;           //发出signal的进程挂起自己的信号量
        int next_count; };          //在next上等待的进程数
    mutex=1;next=0;next_count=0; //初始化语句
```

```
void enter(InterfaceModule &IM) {
  P(IM.mutex); //判断有否发出过signal的进程
}
```

```
void wait(semaphore &x_sem, int
&x_count,InterfaceModule &IM) {
  x_count++; //等资源进程个数加1,
  if (IM.next_count>0)
       //判断是否有发出过signal的进程
     V(IM.next); //有就释放一个
  else
     V(IM.mutex); //否则开放管程
  P(x_sem); //等资源进程阻塞自己,
            //x_sem初始化为0
  x_count--; } //等资源进程个数减1
```

```
void leave(InterfaceModule &IM) {
   if (IM.next_count>0)
        V(IM.next);
      //有就释放一个发出过signal的进程
   else V(IM.mutex); } //否则开放管程
```

```
void signal(semaphore &x_sem,int
        &x_count,InterfaceModule &IM)
{ if(x_count>0) { //判断等待进程
  IM.next_count++;
        //发出signal进程个数加1
   V(x_sem);  //释放一个等资源的进程
   P(IM.next); //发出signal进程阻塞自己
   IM.next_count--; }
        //发出signal进程个数减1
}
```

if *urgentcount* $>$ 0 **then** *V*(*urgent*) **else** *V*(*mutex*)

condcount := *condcount* + 1;
if *urgentcount* $>$ 0 **then** *V*(*urgent*) **else** *V*(*mutex*);
P(*condsem*);
comment This will always wait;
condcount := *condcount* − 1

The signal operation may be coded:

urgentcount := *urgentcount* + 1;
if *condcount* $>$ 0 **then** {*V*(*condsem*); *P*(*urgent*)};
urgentcount := *urgentcount* − 1

6.4.3 管程求解进程同步与互斥问题

- 互斥问题
  - (1)读者写者问题
  - (2)哲学家就餐问题
- 同步问题
  - (1) 生产者-消费者问题
  - (2) 苹果-桔子问题

霍尔管程求解读者写者问题

```
TYPE  read-write=monitor
    int rc,wc;
      semaphore R,W;R=0;W=0;
      int R_count,W_count;
      rc=0; wc=0;
InterfaceModule IM;
DEFINE start_read, end_read, start_write, end_write;
USE  wait,signal,enter,leave;
```

霍尔管程求解读者写者问题(2)

```
void start_read( ){
  enter(IM);
   if(wc>0) wait(R,R_count,IM);
                        rc++;
            signal(R, R_count, IM);
  leave(IM);   }
```

```
void start_write( ) {
  enter(IM);
       wc++;
       if(rc>0 ‖ wc>1)  wait(W,W_count,IM);
  leave(IM);
}
```

```
void end_read( ) {
  enter(IM);
       rc--;
       if(rc==0)
          signal(W,W_count,IM);
  leave(IM);  }
```

```
void end_write( ) {
  enter(IM);
       wc-- ;
       if(wc>0) signal(W,W_count,IM);
       else signal(R, R_count, IM);
  leave(IM);    }
```

```
process P1( ) {
  ……
   read-write.start_read( );
    {read};
   read-write.end_read( );
  …… }
```

```
process P2( ) {
  ……
   read-write.start_write( );
     {write};
   rear-write.end_write( );
  …… }
```

霍尔管程求解哲学家就餐问题

```
type dining_philosophers=monitor
  enum {thinking, hungry, eating} state[5];
  semaphore self[5];   int self_count[5];   InterfaceModule IM;
  for (int i=0;i<5;i++)  state[i]=thinking; //初始化，i为进程号
 define pickup, putdown;
 use enter, leave, wait, signal
```

```
void pickup(int i) { //i=0,1,...,4
    enter(IM);
        state[i]=hungry;
        test(i);
        if(state[i]!=eating)
            wait(self[i],self_count[i],IM);
    leave(IM);
}
```

```
void putdown(int i)
{    //i=0,1,2,..,4
    enter(IM);
        state[i]=thinking;
        test((i-1)%5);
        test((i+1)%5);
    leave(IM);
}
```

```
void test(int k)  {      //k=0,1,...,4
    if((state[(k-1)%5]!=eating)&&(state[k]==hungry)
            &&(state[(k+1)%5]!=eating)) {
                state[k]=eating; signal(self[k],self_count[k],IM);
            }
    }
}
```

霍尔管程求解生产者消费者问题

```
type producer_consumer=monitor
item B[k]; int in, out; //B[k]表示缓冲单元， in, out是存取指针
int count;    //缓冲中产品数
semaphore notfull, notempty; //条件变量
int notfull_count, notempty_count; InterfaceModule IM;
define append,take;
use enter,leave,wait,signal;
```

```
void append(item x) {
  enter(IM);
    if(count==k)   //缓冲已满
   wait(notfull,notfull_count,IM);
      B[in]=x;
      in=(in+1)%k;
      count++; //增加一个产品
signal(notempty,notempty_count,IM);
              //唤醒等待消费者
  leave(IM);
}
```

```
void take(item &x) {
  enter(IM);
    if(count==0) //缓冲已空
wait(notempty,notempty_count,IM);
      x=B[out];
      out=(out+1)%k;
      count--;      //减少一个产品
  signal(notfull,notfull_count,IM);
        //唤醒等待生产者
  leave(IM);
}
```

```
process producer_i( ) { //i=1,…,n
    item x;
          produce(x);
    producer_consumer.append(x)
}
```

```
process consumer_j( ) { //j=1,…m
          item x;
    producer_consumer.take(x);
          consume(x);
}
```

霍尔管程求解苹果桔子问题

- 桌上有一只盘子，每次只能放入一只水果。爸爸专向盘子中放苹果(apple)，妈妈专向盘子中放桔子(orange)，一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子里的苹果。使用Hoare管程求解该问题

```
type FMSD=MONITOR
enum FRUIT {apple,orange} plate;  bool full;
semaphore SP, SS, SD;  int SP_count, SS_count, SD_count;
full=false; InterfaceModule IM; DEFINE put,get;
USE enter,leave,wait,signal;
```

```
void put(FRUIT fruit) { // fruit:
apple or orange
   enter(IM);
      if(full)
        wait(SP,SP_count,IM);
           full=true;
           plate=fruit;
      if(fruit==orange)
         signal(SS,SS_count,IM);
      else   signal(SD,SD_count,IM);
   leave(IM); }
```

```
void get(FRUIT fruit, FRUIT &x) {
    enter(IM);
        if (!full ‖ plate != fruit) {
            if (fruit==orange)
                wait(SS,SS_count,IM);
            else wait(SD,SD_count,IM);
            }
            x=plate;
              full=false;
            signal(SP,SP_count,IM);
        leave(IM);              }
```

```
process father( ){
     {准备好苹果};
     FMSD.put(apple);       }
```

```
process son( ){
      FMSD.get(orange, x);
      {吃取到的桔子};          }
```

```
process mother( ){
     {准备好桔子};
     FMSD.put(orange);     }
```

```
process daughter( ){
       FMSD.get(apple, x);
       {吃取到的苹果};          }
```

表 3-1　操作系统并发问题解决方案

原语类型	采用策略	同步机制	适用场合	方向
高级通信原语	采用消息传递、共享内存、共享文件策略	消息队列、共享内存、管道通信	解决并发进程通信、同步和互斥问题，适用于面向语句的高级程序设计	上↑自底向上↓底
低级通信原语	采用阻塞/唤醒 + 集中临界区（1 次测试）策略	管程	解决并发进程同步和互斥问题，不能传递消息，适用于面向语句的高级程序设计	
	采用阻塞/唤醒 + 分散临界区（1 次测试）策略	信号量和 PV 操作	解决并发进程同步和互斥问题，不能传递消息，适用于面向指令的低级程序设计	
	采用忙式等待（反复测试）策略	关中断、对换、测试并建立、peterson 算法、dekker 算法	解决并发进程互斥问题，不能传递消息，适用于面向指令的低级程序设计	

只需要测一次

忙式等待