南京大学
NANJING UNIVERSITY

计算机与操作系统

第十一讲 并发程序设计

南京大学软件学院

本主题教学目标

1. 了解程序的并发性与并发程序设计
2. 掌握临界区互斥及其解决方案
3. 熟练使用PV进行程序设计
4. 掌握Hoare管程
5. 掌握消息传递

南京大学
NANJING UNIVERSITY

第十一讲 并发程序设计

11.1 并发进程

11.2 临界区管理

11.3 信号量与PV操作

11.4 管程

11.5 进程通信

南京大学
NANJING UNIVERSITY

11.1 并发进程

11.1.1 顺序程序设计

11.1.2 进程的并发性

11.1.3 进程的交互：竞争和协作

11.1.1 顺序程序设计

* 一个进程在处理器上的顺序执行是严格按序的，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
* 执行结果的确定性
* 计算过程的可再现性

11.1.2 进程的并发性

* 进程的并发性(Concurrency)是指一组进程的执行在时间上是重叠的
* 例如：有两个进程A(a1、a2、a3)和B(b1、b2、b3)并发执行，若允许进程交叉执行，如执行操作序列为a1，b1，a2，b2，a3，b3或a1，b1，a2，b2，b3，a3等，则说进程A和B的执行是并发的
* 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

南京大学
NANJING UNIVERSITY

并发程序设计

* 使一个程序分成若干个可同时执行的程序模块的方法称**并发程序设计**(concurrent programming)，每个程序模块和它执行时所处理的数据就组成一个进程

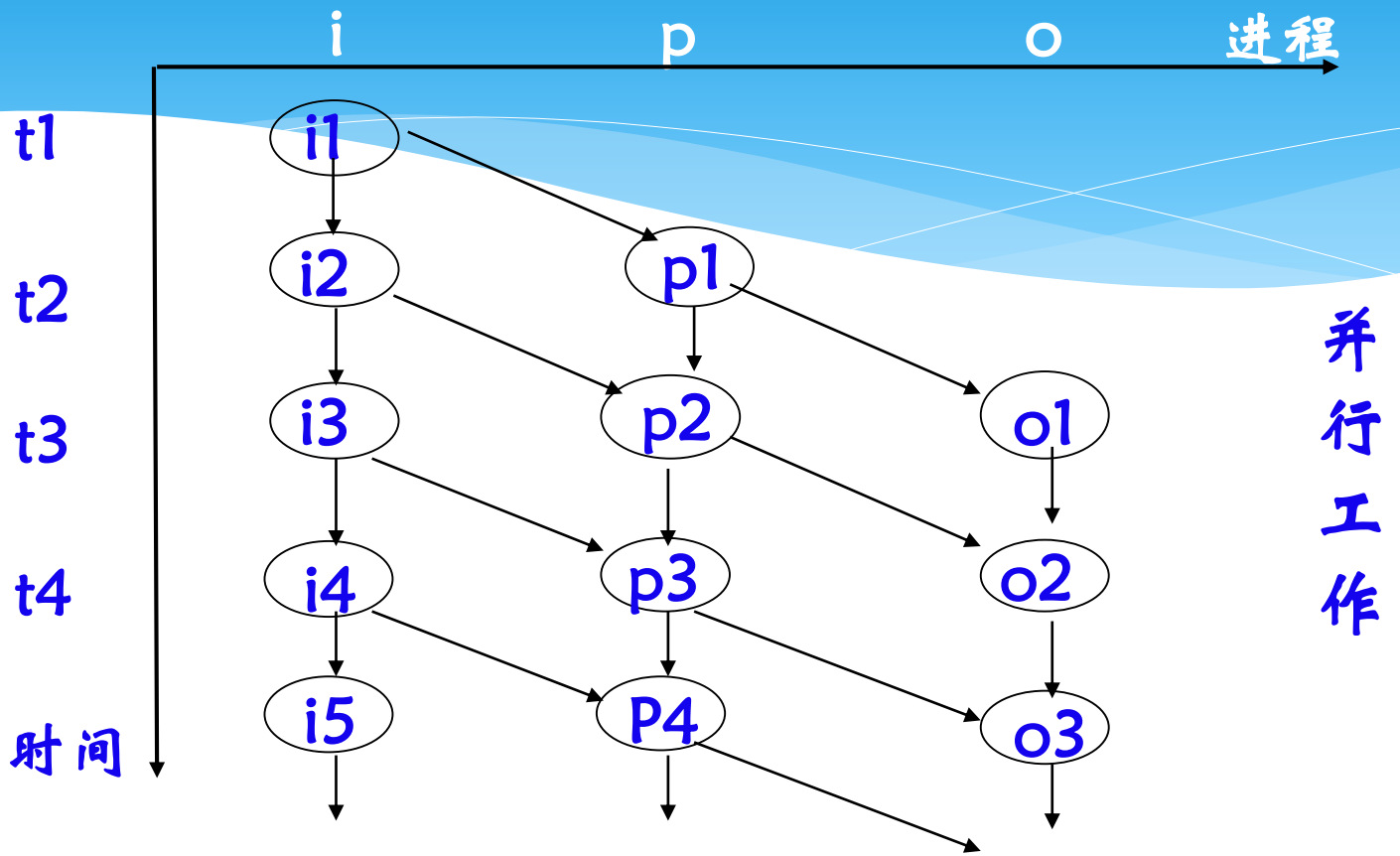
8

进程的并发性(1)

(a) 串行工作

* 由于程序是按照while(TRUE) {input，process，output}串行地输入-处理-输出来编制的，所以该程序只能顺序地执行，这时系统的效率相当低
* 如果把求解这个问题的程序分成三部分：
  * i: while(TRUE) {input，send}
  * p: while(TRUE) {receive，process，send}
  * o: while(TRUE) {receive，output}

进程的并发性(2)



小程序1：循环执行，读入字符，将读入字符送缓冲区1

小程序2：循环执行，处理缓冲区1中的字符，把计算结果送缓冲区2

小程序3：循环执行，取出缓冲区2中的计算结果并写到磁带上

进程的并发性(3)

* 每一部分是一个小程序，它们可并发执行，并会产生制约关系，其中send和receive操作用于小程序之间通过通信机制解决制约关系，以便协调一致地工作

南京大学
NANJING UNIVERSITY

并发进程的分类

* 并发进程分类：无关的，交互的
* 无关的并发进程：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，又称为Bernstein条件
* 交互的并发进程：不满足Bernstein条件，一个进程的执行可能影响其他并发进程的结果

南京大学 NANJING UNIVERSITY

Bernstein条件

* $R(p_i)=\{ai1,ai2,\ldots ain\}$，程序pi在执行期间引用的变量集
* $W(pi)=\{bi1,bi2,\ldots bim\}$，程序pi在执行期间改变的变量集
* 若两个进程的程序p1和p2能满足Bernstein条件，即满足：

  $(R(p1)\cap W(p2)\cup R(p2)\cap W(p1)\cup W(p1)\cap W(p2)=\varnothing$

  则并发进程的执行与时间无关

Bernstein条件举例

* 例如，有如下分属四个进程中的四条语句：

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}

S1和S2可并发执行，满足Bernstein条件

其他语句并发执行可能会产生与时间有关的错误

与时间有关的错误

* 对于一组交互的并发进程，执行的相对速度无法相互控制，各种与时间有关的错误就可能出现。
* 与时间有关错误的表现形式：
  * 结果不唯一
  * 永远等待

机票问题

//飞机票售票问题

```
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
 {输出信息"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
   {输出信息"票已售完"};
}
```

南京大学
NANJING UNIVERSITY

机票问题

* 此时出现把同一张票卖给两个旅客的情况，两个旅客可能各自都买到一张同天同次航班的机票，可是，Aj的值实际上只减去1，造成余票数不正确。特别是，当某次航班只有一张余票时，可能把一张票同时售给两位旅客

主存管理问题

申请和归还主存资源问题

int X=memory; //memory为初始主存容量

void borrow(int B) {
while(B>X)
{进程进入等待主存资源队列};
X=X-B ;
{修改主存分配表，进程获得主存资源};
}
	void return(int B) {
	X=X+B;
	{修改主存分配表};
	{释放等主存资源进程};
	}

主存管理问题

* 由于borrow和return共享代表主存物理资源的临界变量X，对并发执行不加限制会导致错误，例如，一个进程调用borrow申请主存，在执行比较B和X大小的指令后，发现B>X，但在执行{进程进入等待主存资源队列}前，另一个进程调用return抢先执行，归还所借全部主存资源；这时，由于前一个进程还未成为等待者，return中的{释放等主存资源进程}相当于空操作，以后当调用borrow的应用进程被置成{等主存资源}时，可能已经没有其他进程再来归还主存，从而，申请资源的进程处于永远等待状态

11.1.3 进程的交互：竞争与协作

* 进程之间存在两种基本关系：竞争关系和协作关系
* **第一种是竞争关系**，一个进程的执行可能影响到同其竞争资源的其他进程，如果两个进程要访问同一资源，那么，一个进程通过操作系统分配得到该资源，另一个将不得不等待
* **第二种是协作关系**，某些进程为完成同一任务需要分工协作，由于合作的每一个进程都是独立地以不可预知的速度推进，这就需要相互协作的进程在某些协调点上协调各自的工作。当合作进程中的一个到达协调点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，直到其他合作进程发来协调信号或消息后方被唤醒并继续执行

竞争关系带来的问题

* 资源竞争的两个控制问题：
* 一个是死锁(Deadlock)问题：一组进程如果都获得了部分资源，还想要得到其他进程所占有的资源，最终所有的进程将陷入死锁
* 一个是饥饿(Starvation) 问题：一个进程由于其他进程总是优先于它而被无限期拖延
* 操作系统需要保证诸进程能互斥地访问临界资源，既要解决饥饿问题，又要解决死锁问题

竞争关系：死锁

* 死锁：一组进程因争夺资源陷入永远等待的状态
* P0和P1两个进程，均需要使用S和Q两类资源，每类资源数为1

P0	P1
申请 (S);	申请 (Q);
申请 (Q);	申请 (S);
...	...
释放 (S);	释放 (Q);
释放 (Q);	释放 (S);

竞争关系：进程的互斥

* 进程的互斥(mutual exclusion)是解决进程间竞争关系(间接制约关系)的手段。进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源

协作关系：进程的同步

* 进程的同步(Synchronization)是解决进程间协作关系(直接制约关系)的手段。进程同步指两个以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号，当一个进程没有得到来自于另一个进程的消息或信号时则需等待，直到消息或信号到达才被唤醒

进程的交互：竞争与协作

* 进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，是对进程使用资源次序上的一种协调

南京大学
NANJING UNIVERSITY

11.2 临界区管理

南京大学
NANJING UNIVERSITY

11.2 临界区管理

11.2.1 互斥与临界区

11.2.2 临界区管理的尝试

11.2.3 实现临界区管理的硬件设施

南京大学
NANJING UNIVERSITY

11.2.1 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”(critical section)，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
* 竞争条件(race condition)
* 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)

* 临界区调度原则(Dijkstra, 1965)：
  * 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

南京大学
NANJING UNIVERSITY

互斥与临界区(3)

```
    /*  Process 1 */
void P1
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

```
    /*  Process 2 */
void P2
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

。 。 。

```
    /*  Process n */
void Pn
{
  while (true)
  {
    /* preceding code */
   entercritical (Ra);
    /* critical section */
   exitcritical (Ra);
    /* following code */
  }
}
```

11.2.2临界区管理的尝试(1)

```
bool inside1=false; //P1不在其临界区内
bool inside2=false; //P2不在其临界区内
cobegin /*cobegin和coend表示括号中的进程是一组并发进程*/
```

process P1( ) {
while(inside2);//等待
inside1=true;
{临界区};
inside1=false;
}
	process P2( ) {
	while(inside1);//等待
	inside2=true;
	{临界区};
	inside2=false;
	}

```
coend
```

思考：该算法存在的问题

临界区管理的尝试(2)

* 存在的问题：两个进程可能都进去
* 进程P1(P2)测试inside2(insidel)与随后置insidel(inside2)之间，P2(P1)可能发现insidel(inside2)有值false，于是它将置inside2(insidel)为true，并且与进程P1(P2)同时进入临界区

临界区管理的尝试(3)

```
bool inside1=false; //P1不在其临界区内
bool inside2=false; //P2不在其临界区内
cobegin
```

process P1( ) {	process P2( ) {
inside1=true;	inside2=true;
while(inside2);//等待	while(inside1);//等待
{临界区};	{临界区};
inside1=false;	inside2=false;
}	}

```
coend
```

思考：该算法存在的问题

临界区管理的尝试(4)

* 存在的问题：两个进程都进不去
* 延迟进程P1(P2)对inside2(insidel)的测试，先置insidel(inside2)为true，用以封锁P2(P1)，但是，有可能每个进程都把自己的标志置成true，从而出现死循环，这时没有进程能在有限时间内进入临界区，造成永远等待。

11.2.3 实现临界区管理的硬件设施

* (1) 关中断
* (2) 测试并建立指令
* (3) 对换指令

南京大学
NANJING UNIVERSITY

(1) 关中断

* 实现互斥的最简单方法
* 关中断适用场合
* 关中断方法的缺点

(2)测试并建立指令(1)

TS指令的处理过程

```
bool TS(bool &x) {
 if(x) {
   x=false;
   return true;
  }
 else
  return false;
   }
```

(2)测试并建立指令(2)

```
  //TS指令实现进程互斥
bool s=true;
cobegin
process Pi( ) {//i=1,2,...,n
        while(!TS(s));        //上锁
        {临界区};
        s=true;               //开锁
}
coend
```

南京大学
NANJING UNIVERSITY

对换指令(1)

```
void SWAP(bool &a, bool &b) {
   bool temp=a;
   a=b;
   b=temp;
   }
```

对换指令(2)

```
//对换指令实现进程互斥
bool lock=false;
cobegin
Process Pi( ){  //i=1,2,...,n
   bool keyi=true;
    do {
      SWAP(keyi,lock);
    }while(keyi);              //上锁
    {临界区};
    SWAP(keyi,lock);     //开锁
}
coend
```

南京大学
NANJING UNIVERSITY

11.3 信号量与PV操作

11.3.1 信号量与PV操作

11.3.2 信号量实现互斥

11.3.3 经典问题求解

11.3.1 信号量与PV操作

* 前面方法解决临界区调度问题的缺点:
  * (1)对不能进入临界区的进程，采用忙式等待测试法，浪费CPU时间
  * (2)将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重用户编程负担

11.3.1 信号量与PV操作

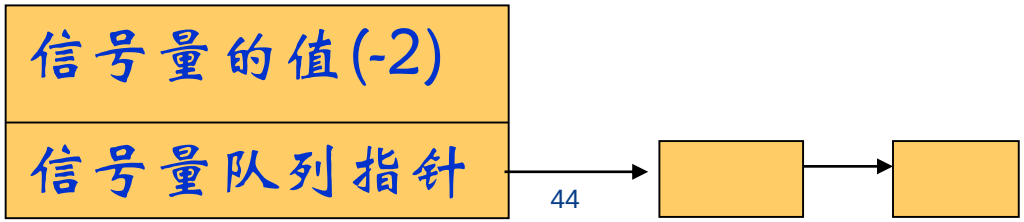
* 1965年E.W.Dijkstra提出了新的同步工具--信号量和P、V操作原语(荷兰语中“检测(Proberen)”和“增量(Verhogen)”的头字母)
* 一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(semaphore)，复杂的进程协作需求都可以通过适当的信号结构得到满足

Edsger W. Dijkstra: Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9): 569 (1965)

信号量与PV操作

设s为一个记录型数据结构，一个分量为整型量value，另一个为信号量队列queue，P和V操作原语定义：

* P(s)：将信号量s减去1，若结果小于0，则调用P(s)的进程被置成等待信号量s的状态
* V(s)：将信号量s加1，若结果不大于0，则释放(唤醒)一个等待信号量s的进程，使其转换为就绪态

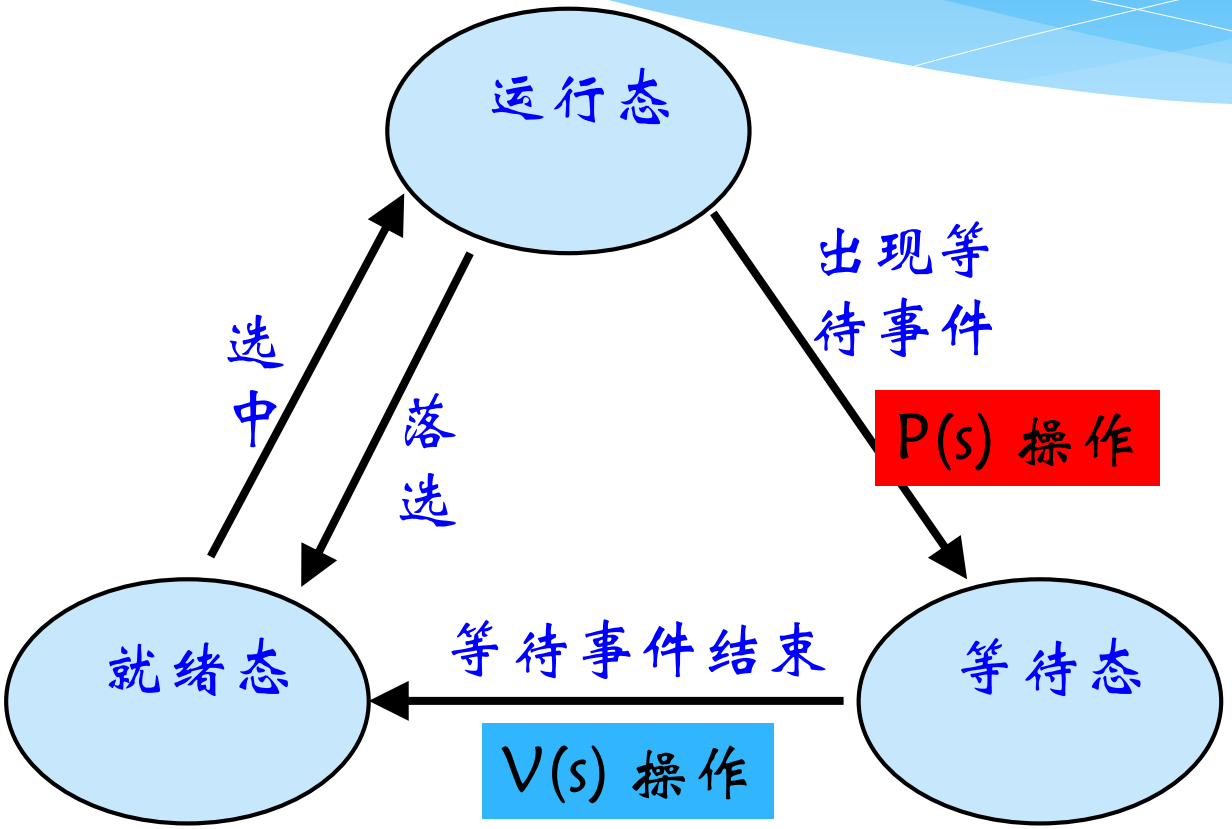


信号量与PV操作

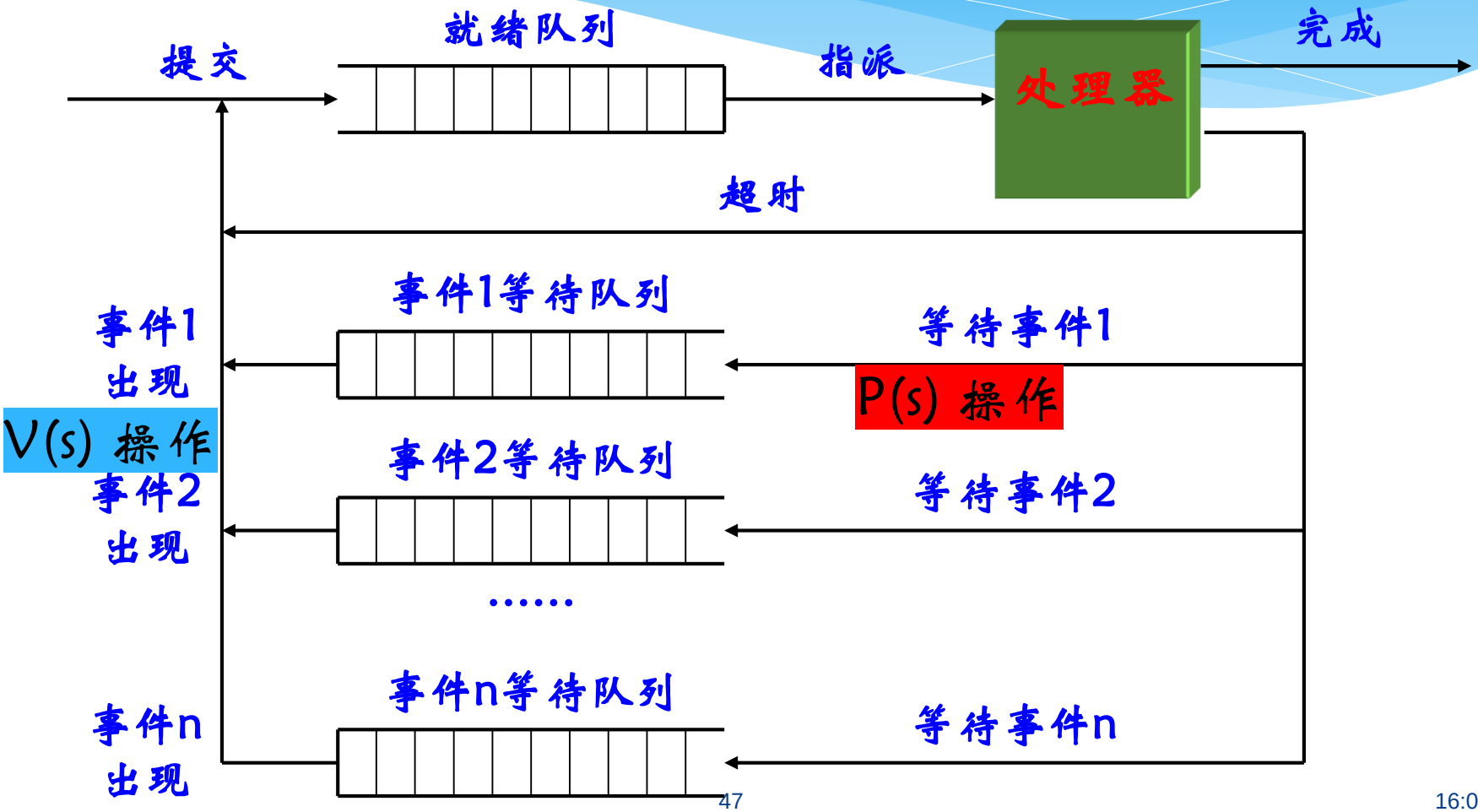
```
struct  semaphore
  { int count; QueueType queue; }
void P(semaphore s); // also named wait
{
  s.count - -;
  if (s.count < 0)  { place this process in s.queue; block this process }
};

void V(semaphore s); // also named signal
{
    s.count ++;
    if (s.count <= 0)
       { remove a process from s.queue; convert it to ready state; }
};
```

PV操作与进程状态转换模型



PV操作与进程状态队列模型



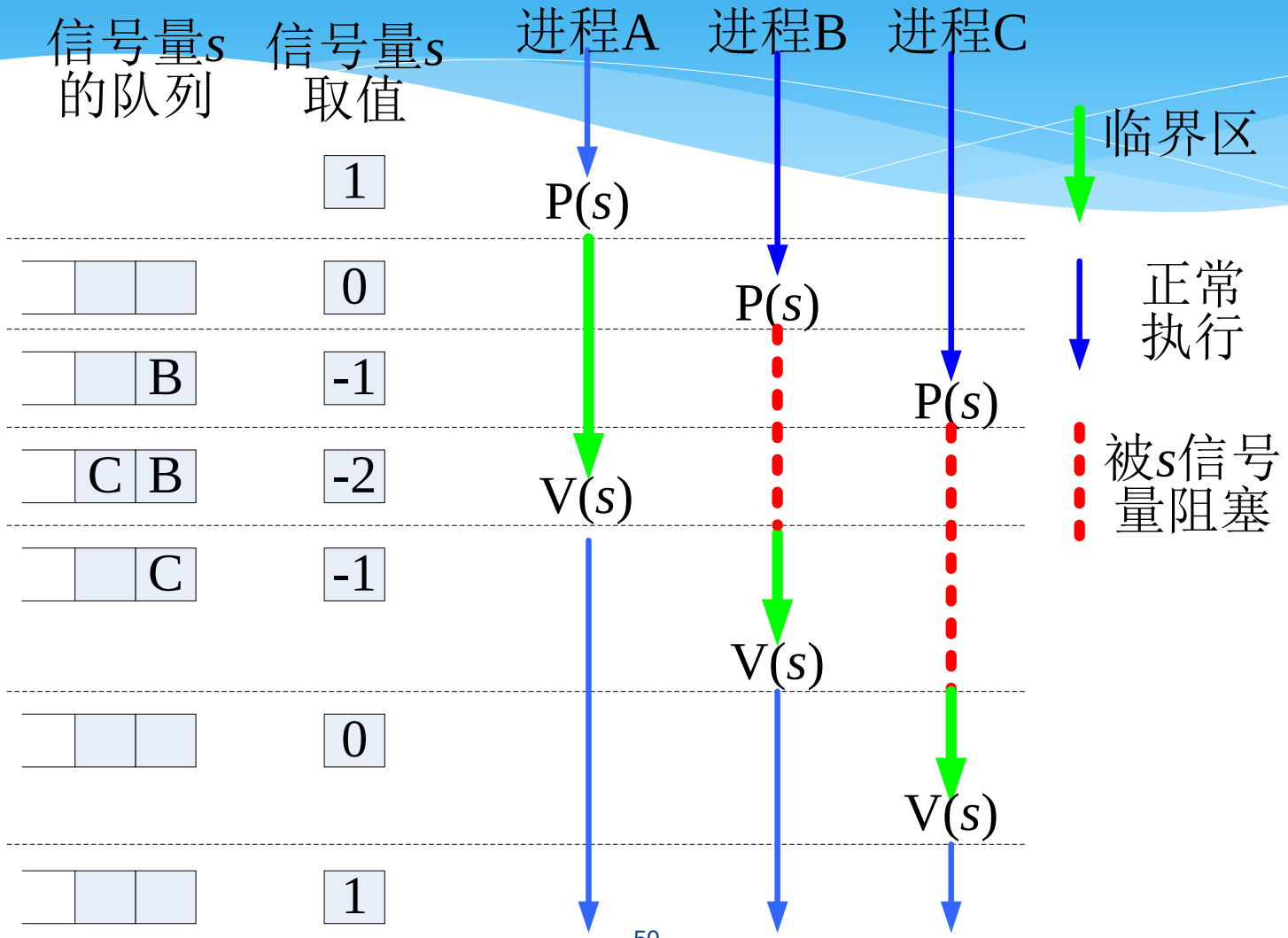
信号量

* 推论1：若信号量s为正值，则该值等于在封锁进程之前对信号量s可施行的P操作次数、亦等于s所代表的实际还可以使用的物理资源数
* 推论2：若信号量s为负值，则其绝对值等于登记排列在该信号量s队列之中等待的进程个数、亦即恰好等于对信号量s实施P操作而被封锁起来并进入信号量s队列的进程数
* 推论3：通常，P操作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操作代表阻塞进程操作，而V操作代表唤醒被阻塞进程的操作

11.3.2 信号量实现互斥

```
semaphore s=1;
cobegin
  Process Pi       /* i=1,…,n */
  {
      …
      P(s);
      /* critical region */
      V(s);
      …
  };
coend
```

进程访问受信号量保护的共享数据



南京大学 NANJING UNIVERSITY

11.3.3 经典问题求解 (信号量与PV操作)

* 互斥问题
  * (1) 飞机票问题
  * (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

求解飞机票问题(1)

```
Var A : ARRAY[1..m] of integer;
  mutex : semaphore;
  mutex:= 1;
cobegin
process Pi
    var Xi:integer;
begin
    L1:
      按旅客定票要求找到A[j];
      P(mutex);
      Xi := A[j];
      if Xi>=1  then
      begin
        Xi:=Xi-1;A[j]:=Xi;
        V(mutex); {输出一张票};
      end;
      else begin
        V(mutex); {输出“票已售完”};
      end;
    goto L1;
 end;
coend
```

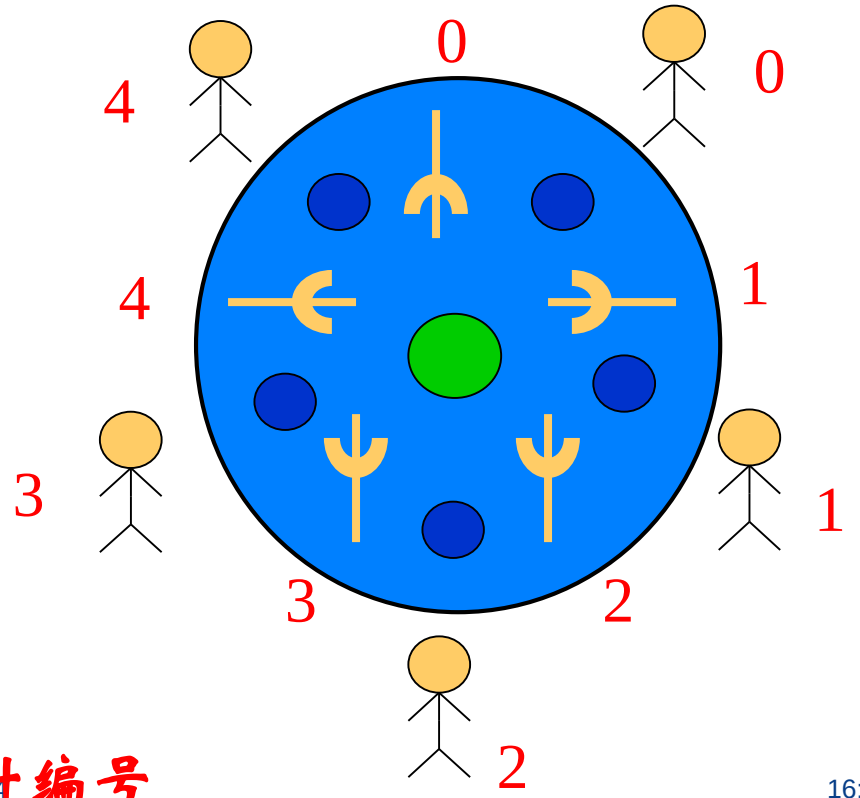
求解飞机票问题(2)

```
Var A : ARRAY[1..m] of integer;
  s : ARRAY[1..m] of semaphore;
  s[j] := 1;
cobegin
process Pi
   var Xi:integer;
begin
   L1:
     按旅客定票要求找到A[j];
     P(s[j]);
     Xi := A[j];
   if Xi>=1  then
   begin
     Xi:=Xi-1;A[j]:=Xi;
     V(s[j]); {输出一张票};
   end;
   else begin
     V(s[j]); {输出“票已售完”};
   end;
   goto L1;
end;
coend
```

南京大学
NANJING UNIVERSITY

哲学家就餐问题(1)

* 有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子



哲学家顺时针编号

16:03

哲学家就餐问题(2)

```
semaphore fork[5];
for (int i=0;i<5;i++)
fork[i]=1;
cobegin
process philosopher_i( ) {  //i= 0,1,2,3,4
  while(true) {
    think( );
    P(fork[i]);                    //先取右手的叉子
    P(fork[(i+1)%5]);              //再取左手的叉子
       eat( );
    V(fork[i]);
    V(fork[(i+1)%5]);
     }
}
coend
```

存在什么问题？

死锁！

有若干种办法可避免这类死锁

上述解法可能出现永远等待，有若干种办法可避免死锁

- 至多允许四个哲学家同时取叉子
- 奇数号先取左手边的叉子，偶数号先取右手边的叉子
- 每个哲学家取到手边的两把叉子才吃，否则一把叉子也不取

哲学家就餐问题的其他正确解

```
   semaphore fork[5];
    for (int i=0;i<5;i++)
          fork[i]= 1;
    semaphore room=4; //增加一个侍者
cobegin
process philosopher_i( ){/*i=0,1,2,3 */
          while(true) {
          think( );
          P(room); //控制最多允许4为哲学家取叉子
          P(fork[i];
          P(fork[(i+1)%5] ) ;
            eat( );
          V(fork[i]);
          V(fork([i+ 1] % 5);
          V(room);
     }
 }
coend
```

哲学家就餐问题的其他正确解

```
void philosopher (int i)
{  if  i  mod 2＝＝0 then
   {
          P(fork[i]);          //偶数哲学家先右手
          P(fork[(i+1) mod 5 ]); //后左手
                eat();
          V(fork[i]);
          V (fork[(i+1) mod 5]);
   }
   else
    {
          P (fork[(i+1) mod 5 ]); //奇数哲学家，先左手
          P (fork[i]);  //后右手
                eat();
          V(fork[(i+1) mod 5]);
          V(fork[i]);
   }
}
```

哲学家就餐问题的其他正确解

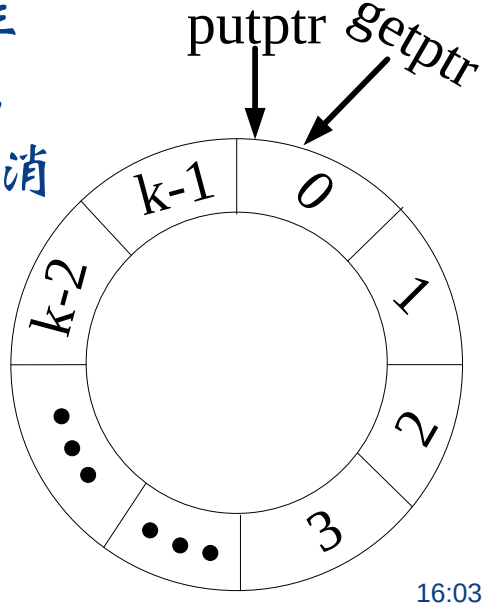
```
semaphore fork[5];
for (int i=0;i<5;i++)
        fork[i]= 1;
cobegin
process philosopher_i( ){/*i=0,1,2,3 */
        while(true) {
        think( );
        P(fork[i];//先取右手的叉子    /*i=4,P(fork[0])*/
        P(fork[(i+1)%5] ) ; //再取左手的叉子 /*i=4,P(fork[4])*/
          eat( );
        V(fork[i]);
        V(fork([i+ 1] % 5);
    }
 }
coend
```

先取左手的叉子

再取右手的叉子

生产者/消费者问题

* 有n个生产者和m个消费者，连接在一个有k个单位缓冲区的有界缓冲上。其中，生产者进程Producer_i和消费者进程Consumer_j都是并发进程，只要缓冲区未满，生产者Producer_i生产的产品就可投入缓冲区；只要缓冲区不空，消费者进程Consumer_j就可从缓冲区取走并消耗产品
* 可能情形：
  * n=1, m=1, k=1
  * n=1, m=1, k>1
  * n>1, m>1, k>1



生产者/消费者问题

B: integer	

Process producer
begin
L1:
produce a product;
B:=product;
goto L1;
end;
	Process consumer
	begin
	L2:
	product:=B;
	consume a product;
	goto L2;
	end;

一个生产者/一个消费者/一个缓冲单元

```
B: integer;
sput: semaphore;  /* 可以使用的空缓冲区数 */
sget: semaphore;  /* 缓冲区可以使用的产品数 */
sput:=1;          /* 缓冲区内允许放入一件产品 */
sput:=0;          /* 缓冲区内没有产品 */
```

```
Process producer
Begin
    L1:
    produce a product;
    P(sput);
    B:=product;
    V(sget);
    goto L1;
end;
```

```
Process consumer
begin
    L2:
    P(sget);
    product:=B;
    V(sput);
    consume a product;
    goto L2;
end;
```

一个生产者/一个消费者/多个缓单元

```
B : ARRAY[0..k-1] of integer;
sput: semaphore；          /* 可以使用的空缓冲区数 */
sget: semaphore;           /* 缓冲区内可以使用的产品数 */
sput := k;                 /* 缓冲区内允许放入k件产品 */
sget := 0;                 /* 缓冲区内没有产品 */
putptr, getptr : integer; putptr:=0; getptr:=0;
```

```
process producer
begin
   L1: produce a product；
     P(sput);
     B[putptr] := product;
     putptr :=(putptr+1) mod k;
     V(sget);
   goto L1;
end;
```

```
process consumer
begin
   L2: P(sget);
     Product:= B[getptr]；
     getptr:=(getptr+1) mod k;
     V(sput);
     consume a product；
   goto L2;
end;
```

多个生产者/多个消费者/多个缓冲单元

```
B : ARRAY[0..k-1] OF integer;
sput: semaphore；          /* 可以使用的空缓冲区数 */
sget: semaphore;          /* 缓冲区内可以使用的产品数 */
sput := k;                /* 缓冲区内允许放入k件产品 */
sget := 0;                /* 缓冲区内没有产品 */
putptr, getptr : integer;
putptr := 0; getptr := 0;
s: semaphore；            /* 互斥使用putptr, getptr */
s := 1；
```

```
process Producer_i
begin
   L1:produce a product；
      P(sput);
      P(s);
      B[putptr] := product;
      putptr :=(putptr+1) mod k;
      V(s);
      V(sget);
   goto L1;
end;
```

```
process Consumer_ j
begin
   L2:P(sget);
      P(s);
      Product:= B[getptr]；
      getptr:=(getptr+1) mod k;
      V(s);
      V(sput);
      consume a product；
   goto L2;
end;
```

多个生产者/多个消费者/多个缓冲单元

```
B : ARRAY[0..k-1] OF integer;
sput: semaphore；          /* 可以使用的空缓冲区数 */
sget: semaphore;          /* 缓冲区内可以使用的产品数 */
sput := k;                /* 缓冲区内允许放入k件产品 */
sget := 0;                /* 缓冲区内没有产品 */
putptr, getptr : integer;
putptr := 0; getptr := 0;
s: semaphore；            /* 互斥使用putptr, getptr */
s := 1；
```

```
process Producer_i
begin
   L1:produce a product；
      P(sput);
      P(s1);
      B[putptr] := product;
      putptr :=(putptr+1) mod k;
      V(s1);
      V(sget);
   goto L1;
end;
```

```
process Consumer_ j
begin
   L2:P(sget);
      P(s2);
      Product:= B[getptr]；
      getptr:=(getptr+1) mod k;
      V(s2);
      V(sput);
      consume a product；
   goto L2;
end;
```

苹果-桔子问题

爸爸

放苹果

妈妈

放桔子

盘子

取桔子

儿子

取苹果

女儿

苹果-桔子问题

```
plate : integer;
sp:semaphore;          /* 盘子里可以放几个水果 */
sg1:semaphore;         /* 盘子里有桔子 */
sg2:semaphore;         /* 盘子里有苹果 */
sp := 1;               /* 盘子里允许放入一个水果*/
sg1 := 0;              /* 盘子里没有桔子 */
sg2 := 0;              /* 盘子里没有苹果*/
```

```
process father
begin
   L1: 削一个苹果;
       P(sp);
       把苹果放入plate;
       V(sg2);
  goto L1;
end;
```

```
process son
begin
   L3: P(sg1);
       从plate中取桔子;
       V(sp);
       吃桔子;
   goto L3;
end;
```

```
process mother
begin
    L2: 剥一个桔子;
       P(sp);
       把桔子放入plate;
       V(sg1);
    goto L2;
end;
```

```
process daughter
begin
    L4: P(sg2);
       从plate中取苹果;
       V(sp);
       吃苹果;
   goto L4;
end;
```

习题
(信号量与PV操作)

* 读者写者问题
* 睡眠的理发师问题
* 农夫猎人问题
* 银行业务问题
* 缓冲区管理
* 售票问题
* 吸烟者问题

读者/写者问题

* 读者与写者问题(reader-writer problem) (Courtois, 1971)也是一个经典的并发程序设计问题。有两组并发进程：读者和写者，共享一个文件F，要求：
* (1)允许多个读者可同时对文件执行读操作
* (2)只允许一个写者往文件中写信息
* (3)任一写者在完成写操作之前不允许其他读者或写者工作
* (4)写者执行写操作前，应让已有的写者和读者全部退出
* 使用PV操作求解该问题

睡眠的理发师问题

* 理发店理有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子
* 如果没有顾客，理发师便在理发椅上睡觉
* 一个顾客到来时，它必须叫醒理发师
* 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开
* 使用PV操作求解该问题

农夫猎人问题

* 有一个铁笼子，每次只能放入一个动物。猎手向笼中放入老虎，农夫向笼中放入羊；动物园等待取笼中的老虎，饭店等待取笼中的羊。请用P、V操作原语写出同步执行的程序

南京大学
NANJING UNIVERSITY

银行业务问题

* 某大型银行办理人民币储蓄业务，由n个储蓄员负责。每个顾客进入银行后先至取号机取一个号，并且在等待区找到空沙发坐下等着叫号。取号机给出的号码依次递增，并假定有足够多的空沙发容纳顾客。当一个储蓄员空闲下来，就叫下一个号。请用信号量和P，V操作正确编写储蓄员进程和顾客进程的程序

缓冲区管理

* 有n个进程将字符逐个读入到一个容量为80的缓冲区中(n>1)，当缓冲区满后，由输出进程Q负责一次性取走这80个字符。这种过程循环往复，请用信号量和P、V操作写出n个读入进程(P1， P2，...Pn)和输出进程Q能正确工作的的动作序列

售票问题

* 汽车司机与售票员之间必须协同工作，一方面只有售票员把车门关好了司机才能开车，因此，售票员关好门应通知司机开车，然后售票员进行售票。另一方面，只有当汽车已经停下，售票员才能开门上下客，故司机停车后应该通知售票员。假定某辆公共汽车上有一名司机与两名售票员，汽车当前正在始发站停车上客，试用信号量与P、V操作写出他们的同步算法

南京大学
NANJING UNIVERSITY

吸烟者问题

* 一个经典同步问题：吸烟者问题(patil，1971)。三个吸烟者在一个房间内，还有一个香烟供应者。为了制造并抽掉香烟，每个吸烟者需要三样东西：烟草、纸和火柴，供应者有丰富货物提供。三个吸烟者中，第一个有自己的烟草，第二个有自己的纸和第三个有自己的火柴。供应者随机地将两样东西放在桌子上，允许一个吸烟者进行对健康不利的吸烟。当吸烟者完成吸烟后唤醒供应者，供应者再把两样东西放在桌子上，唤醒另一个吸烟者。试用信号量和P、V操作求解该问题

南京大学
NANJING UNIVERSITY

11.4 管程

南京大学
NANJING UNIVERSITY

11.4 管程

11.4.1 管程和条件变量

11.4.2 管程的实现

11.4.3 管程求解进程的同步与互斥问题

南京大学
NANJING UNIVERSITY

11.4.1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

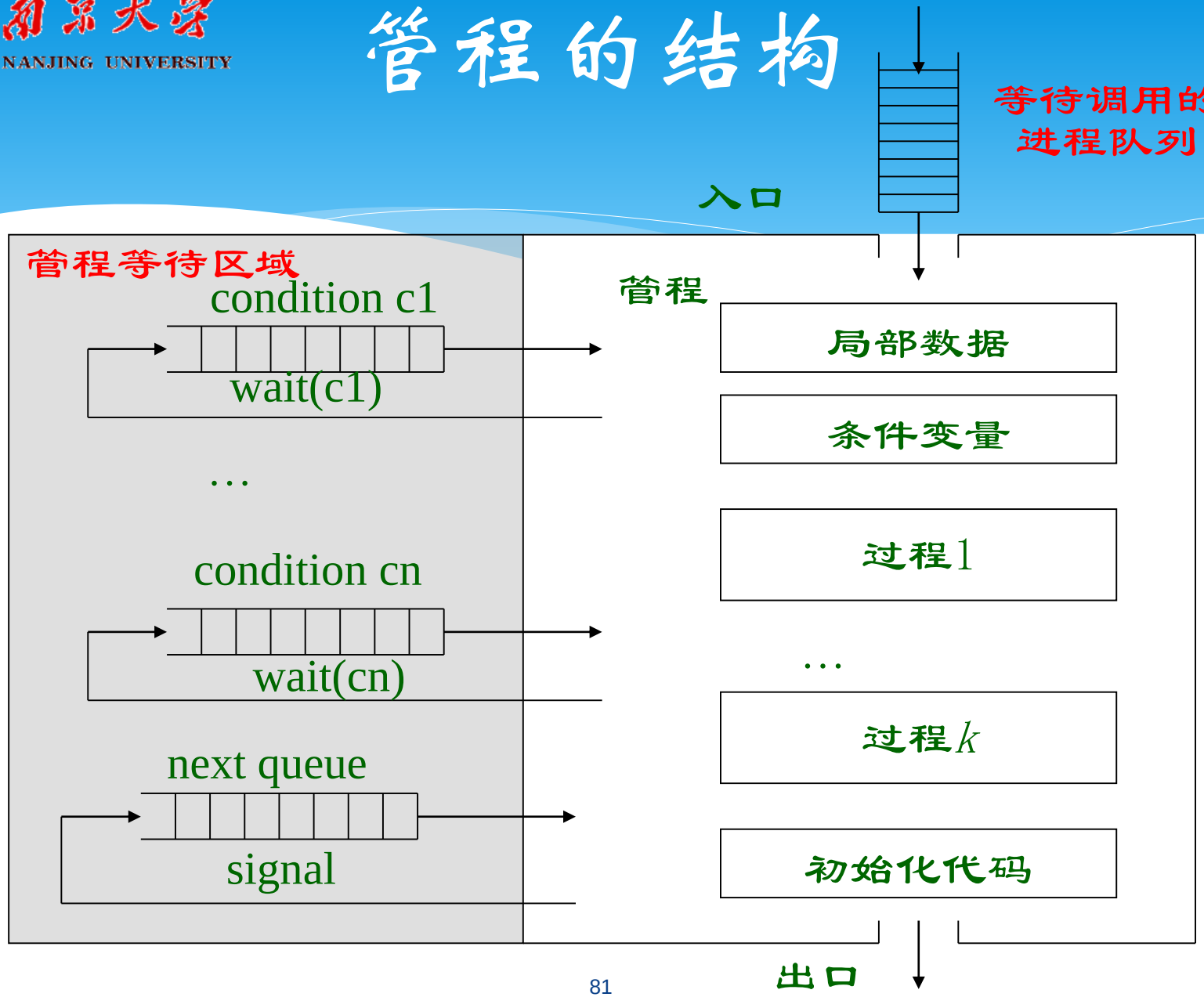
管程定义和属性

* 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

南京大学
NANJING UNIVERSITY

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
  * 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

南京大学 NANJING UNIVERSITY

11.4.2 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

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

南京大学
NANJING UNIVERSITY

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
```

南京大学
NANJING UNIVERSITY

Hoare管程的enter( )和leave( )操作

```
* void enter(InterfaceModule &IM) {
*     P(IM.mutex); //判有否发出过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
    }
}
```

```
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
}
```

11.4.3 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, 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,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);
}
```

南京大学
NANJING UNIVERSITY

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};          }
```

南京大学
NANJING UNIVERSITY

11.5 进程通信

南京大学 NANJING UNIVERSITY

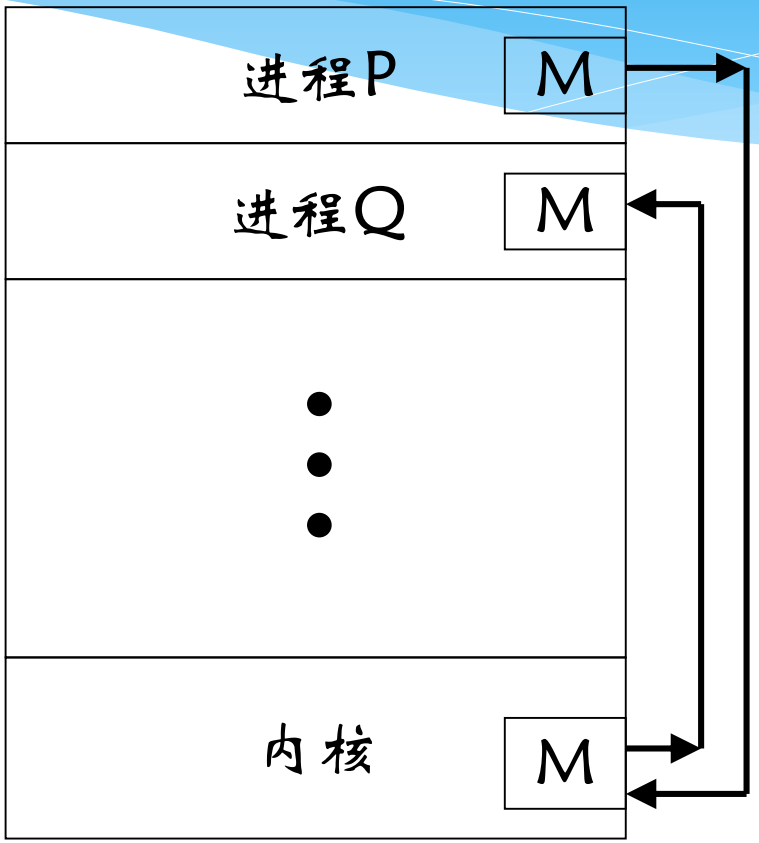
11.5 进程通信 (消息传递)

* 当进程互相交互时，必须满足两个基本要求：同步和通信
  * 为实施互斥，进程间需要同步
  * 为了协作，进程间需要交换信息
* 消息传递提供了这些功能，最典型的消息传递原语
  * send 发送消息的原语
  * receive 接收消息的原语

直接通信

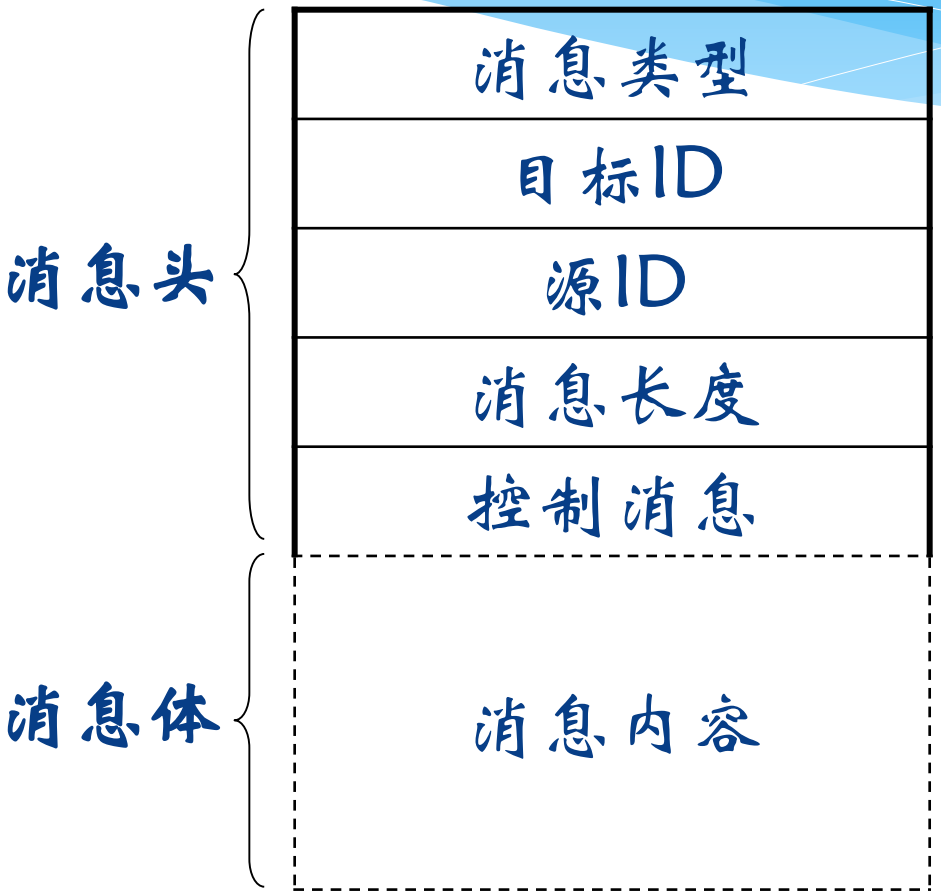
* 对称直接寻址，发送进程和接收进程必须命名对方以便通信，原语send() 和 receive()定义如下：
  * send(P, messsage) 发送消息到进程P
  * receive(Q, message) 接收来自进程Q的消息
* 非对称直接寻址，只要发送者命名接收者，而接收者不需要命名发送者，send()和 receive()定义如下：
  * send(P, messsage) 发送消息到进程P
  * receive(id, message) 接收来自任何进程的消息，变量id置成与其通信的进程名称

直接通信



消息传递：进程P向Q发送消息

消息格式



间接通信

* 消息不是直接从发送者发送到接收者，而是发送到由临时保存这些消息的队列组成的一个共享数据结构，这些队列通常成为信箱(mailbox)
* 一个进程给合适的信箱发送消息，另一进程从信箱中获得消息
* 间接通信的send()和 receive()定义如下：
  * send(A,message)：把一封信件(消息)传送到信箱A
  * receive(A,message)：从信箱A接收一封信件(消息)

间接通信

* 消息不是直接从发送者发送到接收者，而是发送到由临时保存这些消息的队列组成的一个共享数据结构，这些队列通常成为信箱(mailbox)
* 一个进程给合适的信箱发送消息，另一进程从信箱中获得消息
* 间接通信的send()和 receive()定义如下：
  * send(A,message)：把一封信件(消息)传送到信箱A
  * receive(A,message)：从信箱A接收一封信件(消息)

间接通信

* 信箱可以分成信箱头和信箱体两部分，信箱头指出信箱容量、信件格式、存放信件位置的指针等；信箱体用来存放信件，信箱体分成若干个区，每个区可容纳一封信
* “发送”和“接收”两条原语的功能为：
* 发送信件：如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则，发送信件者被置成等待信箱状态
* 接收信件：如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者，否则，接收信件者被置成等待信箱中信件的状态

send/receive原语的算法描述

```
type box=record
     size: integer;                    /*信箱大小*/
     count: integer;                   /*现有信件数*/
     letter: array[1..n] of message;   /*信箱中的信件*/
     S1,S2: semaphore;                 /*等信箱和等信件信号量*/
end
```

```
procedure send(varB:box,M:message)
   var i:integer;
   begin
    if B.count=B.size  then W(B.s1);
       i:=B.count+1;
       B.letter[i]:=M;
       B.count:=i;
       R(B.S2)
    end;
```

```
procedure receive(varB:box, x:message)
   var i:integer;
   begin
    if B.count=0  then W(B.s2);
       B.count:=B.count-1;
       x:=B.letter[1];
       if B.count ≠0 then
      for i=1 to B.count do
          B.letter[i]:=B.letter[i+1];
       R(B.S1);
  end;
```

R()和W()是让进程入队和出队的两个过程

消息传递求解生产者消费者问题

```
creat-mailbox(producer);          //创建信箱
creat-mailbox(consumer);
```

```
void producer_i( ) {  //i=1,… ,n
    message pmsg;
    while(true) {
       pmsg = produce();
       send(consumer, pmsg);
    }
}
```

```
void consumer_j( ) { //j=1,… ,m
    message cmsg;
    while(true) {
     receive (consumer, cmsg);
     consume(csmg);
    }
}
```

```
cobegin
     producer_i( );
     consumer_j( );
 coend
```

有关消息传递问题的若干问题

* 关于信箱容量问题
* 关于多进程与信箱相连的信件接收问题
* 关于信箱的所有权问题
  * 信箱为操作系统所有是指由操作系统统一设置信箱，归系统所有，供相互通信的进程共享，消息缓冲机制就是一个著名的例子
* 关于信件的格式问题和其他有关问题
* 关于通信进程的同步问题

有关消息传递问题的若干问题

* 消息缓冲是在1973年由P.B.Hansan提出的一种进程间高级通信设施，并在RC4000系统中实现
* 消息缓冲通信的基本思想是：由操作系统统一管理一组用于通信的消息缓冲存储区，每一个消息缓冲存储区可存放一个消息(信件)。当一个进程要发送消息时，先在自己的消息发送区里生成待发送的消息，包括：接收进程名、消息长度、消息正文等。然后，向系统申请一个消息缓冲区，把消息从发送区复制到消息缓冲区中，注意在复制过程中系统会将接收进程名换成发送进程名，以便接收者识别。随后该消息缓冲区被挂到接收消息的进程的消息队列上，供接收者在需要时从消息队列中摘下并复制到消息接收区去使用，同时释放消息缓冲区。

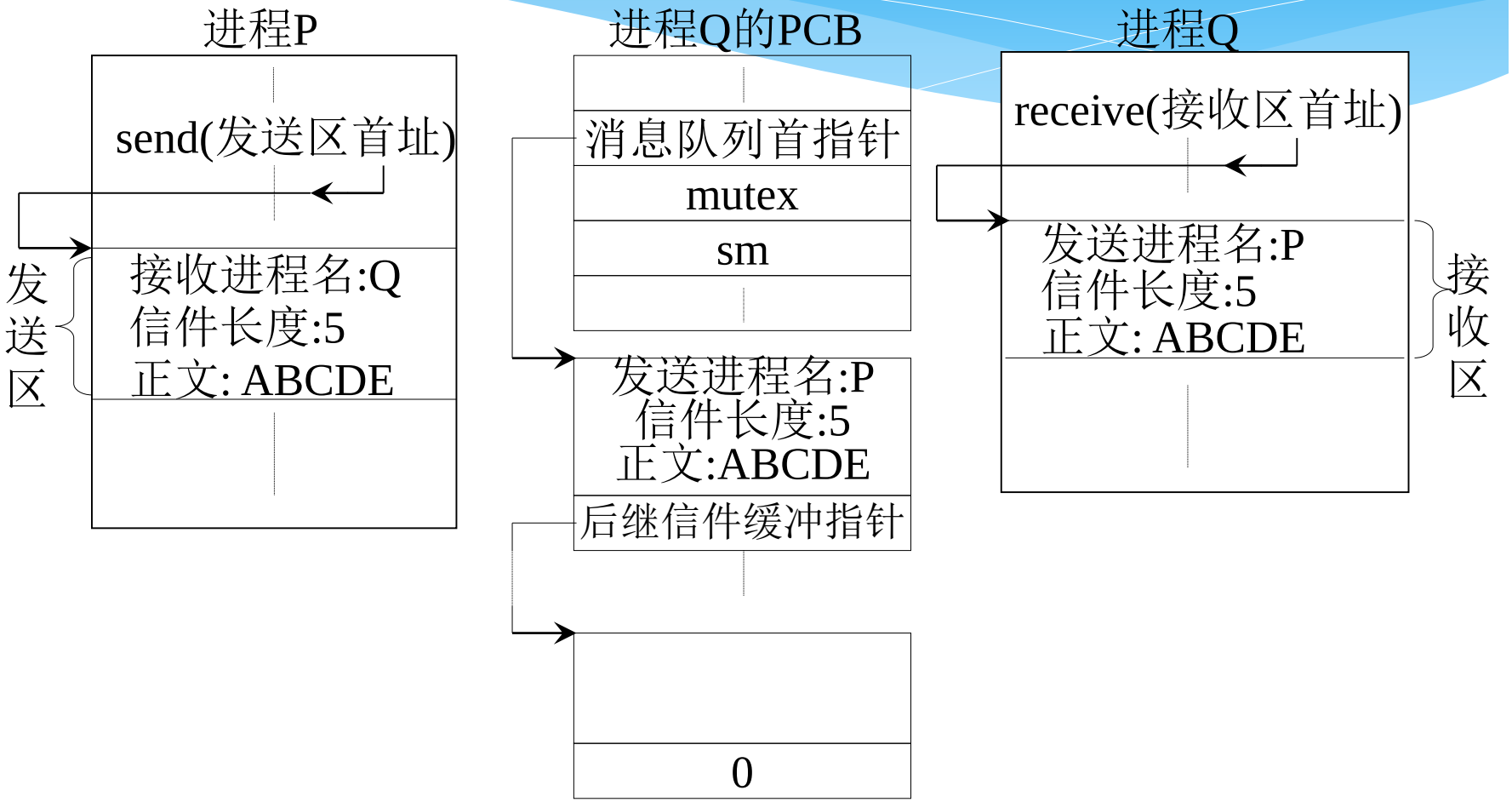
消息缓冲通信

* 消息缓冲通信涉及的数据结构：
  * sender：发送消息的进程名或标识符
  * size：发送的消息长度
  * text：发送的消息正文
  * next-ptr：指向下一个消息缓冲区的指针
* 在进程的PCB中涉及通信的数据结构：
  * mptr：消息队列队首指针
  * mutex：消息队列互斥信号量，初值为1
  * sm：表示接收进程消息队列上消息的个数，初值为0，是控制收发进程同步的信号量

消息缓冲通信

* 发送原语和接收原语的实现如下：
* 发送原语Send：申请一个消息缓冲区，把发送区内容复制到这个缓冲区中；找到接收进程的PCB，执行互斥操作P(mutex)；把缓冲区挂到接收进程消息队列的尾部，执行V(sm)、即消息数加1；执行V(mutex)
* 接收原语Receive：执行P(sm)查看有否信件；执行互斥操作P(mutex)，从消息队列中摘下第一个消息，执行V(mutex)；把消息缓冲区内容复制到接收区，释放消息缓冲区

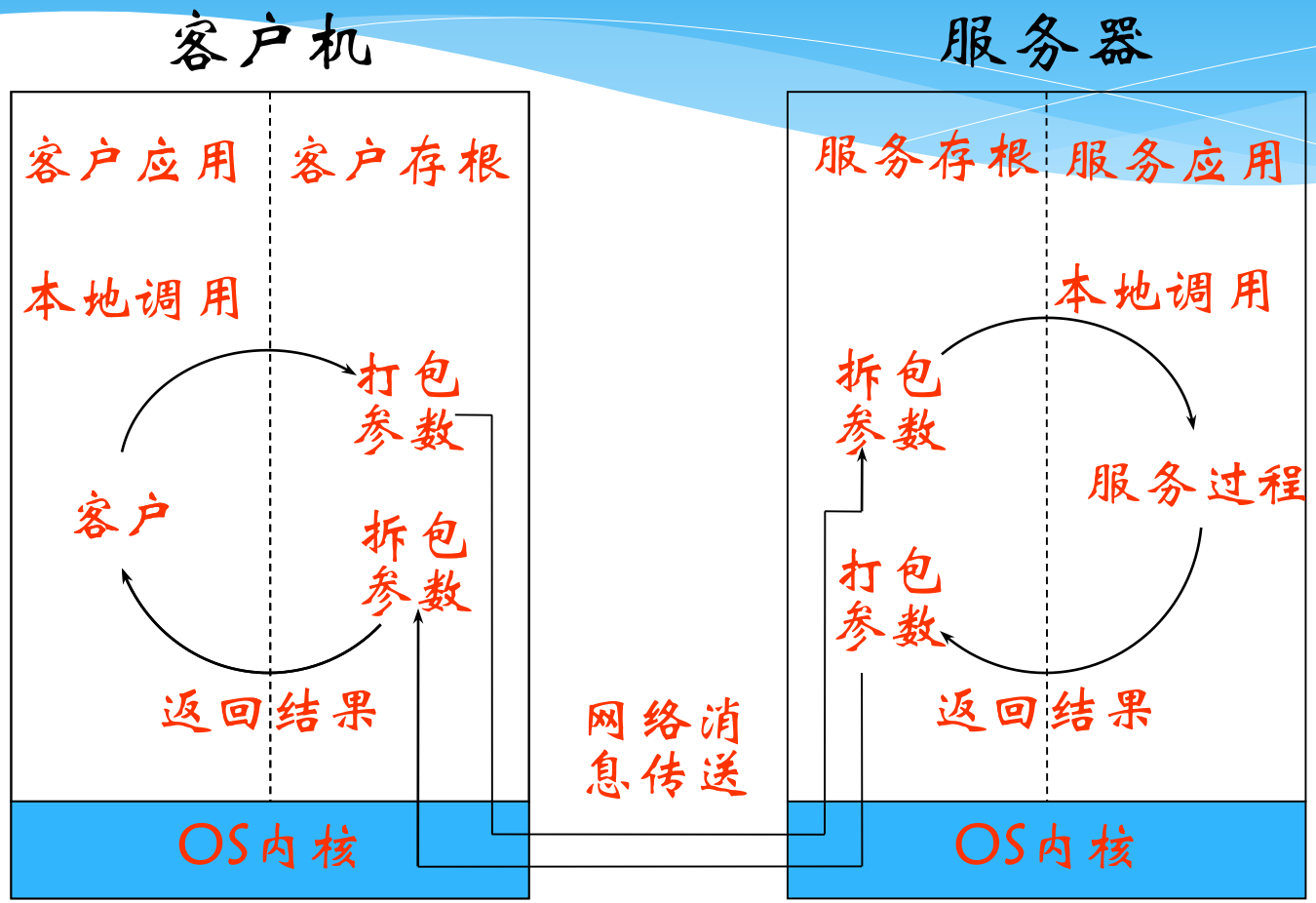
消息缓冲通信


进程P
send(发送区首址)
发送区
接收进程名:Q
信件长度:5
正文: ABCDE
进程Q的PCB
消息队列首指针
mutex
sm
发送进程名:P
信件长度:5
正文:ABCDE
后继信件缓冲指针
0
进程Q
receive(接收区首址)
发送进程名:P
信件长度:5
正文: ABCDE
接收区

管道和套接字

* 管道(pipeline)是Unix和C的传统通信方式
* 套接字(socket)起源于Unix BSD版本，目前已经被Unix和Windows操作系统广泛采用，并支持TCP/IP协议，即支持本机的进程间通信，也支持网络级的进程间通信
* 管道和套接字都是基于信箱的消息传递方式的一种变体，它们与传统的信箱方式等价，区别在于没有预先设定消息的边界。换言之，如果一个进程发送10条100字节的消息，而另一个进程接收1000个字节，那么接收者将一次获得10条消息

远程过程调用 (RPC, Remote Procedure Call)



RPC执行步骤 (1)

(1) 客户进程以普通方式调用客户存根

(2) 客户存根组织RPC消息并执行Send，激活内核程序

(3) 内核把消息通过网络发送到远地内核

(4) 远地内核把消息送到服务器存根

(5) 服务器存根取出消息中参数后调用服务器过程

RPC执行步骤 (2)

(6) 服务器过程执行完后把结果返回至服务器存根

(7) 服务器存根进程将它打包并激活内核程序

(8) 服务器内核把消息通过网络发送至客户机内核

(9) 客户内核把消息交给客户存根

(10) 客户存根从消息中取出结果返回给客户进程

(11) 客户进程获得控制权并得到了过程调用的结果

南京大学
NANJING UNIVERSITY

本主题教学目标

1. 了解程序的并发性与并发程序设计
2. 掌握临界区互斥及其解决方案
3. 熟练使用PV进行程序设计
4. 掌握Hoare管程
5. 掌握消息传递