



# 计算机与操作系统

## 第六章 并发程序设计

### 6.3 信号量与PV操作

葛季栋  
南京大学软件学院



# 信号量与PV操作



- 1、信号量与PV操作系统的问题背景 ←
- 2、信号量与PV操作的基本原理
- 3、信号量的应用
- 4、小结



# 1、信号量与PV操作系统的问题背景



## ■ 1.1 并发程序设计的基本概念

- (1) 并发程序设计
- (2) 临界资源与临界区
- (3) 同步与互斥

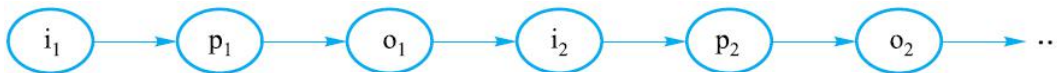
## ■ 1.2 “忙式等待”方法解决临界区调度问题的缺点

## ■ 1.3 操作系统中“并发问题”解决方案的知识框架



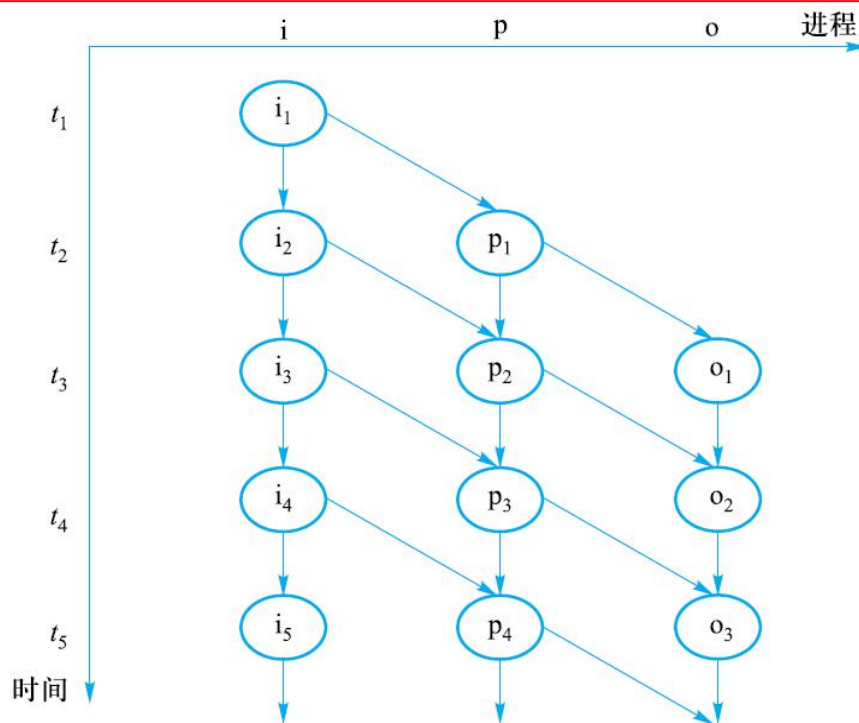
# 1.1 并发程序设计的基本概念

串行



(a) 串行

并发



(b) 并发

图: 串行与并发

优点: 程序控制简单

缺点: 资源利用率低,  
系统吞吐率效率低

引入多道程序设计方法,  
引入多进程并发交替使用CPU的概念

优点: 提高资源利用率,  
提高系统吞吐率

带来的问题:  
并发进程之间  
共享资源的冲突

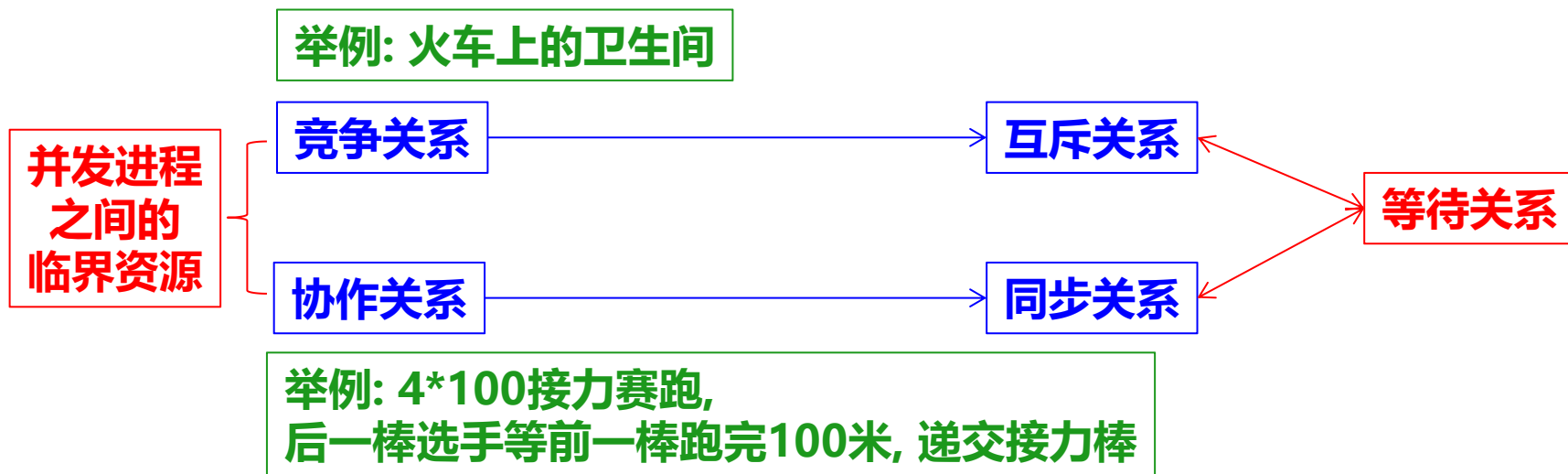


# 1.1 并发程序的基本概念

## 临界资源与临界区、同步与互斥



- **临界资源**：并发进程之间需要互斥使用的共享资源，称为临界资源
  - 举例：火车上的卫生间就是一种互斥使用的共享资源
  - 使用共享变量代表共享资源
  - 并发进程中与共享变量有关的程序段叫“临界区” (critical section)
  - 临界区：并发进程之间控制共享资源使用的程序段
- **并发进程之间的关系**





## 1.2 “忙式等待”方法 解决临界区调度问题的缺点



### ■ 临界区管理的简单方法（忙式等待/反复测试）

- (1) 关中断
- (2) 测试并建立指令
- (3) 对换指令
- (4) Peterson算法

### ■ 存在的问题

- (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

### ■ 通用的解决方案：信号量与PV操作



# 1.3 操作系统中“并发问题” 解决方案的知识框架



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<br>↑<br>自底向上<br>↓<br>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

低级通信原语

忙式等待



# 信号量与PV操作



- 1、信号量与PV操作系统的问题背景
- 2、信号量与PV操作的基本原理 ←
- 3、信号量的应用





## 2、信号量与PV操作

- 2.1 发明人Dijkstra简介
- 2.2 信号量与PV操作的数据结构与原语操作
- 2.3 信号量与PV操作管理临界资源的的举例  
( 火车上的卫生间 )
- 2.4 信号量与进程状态转换模型及其队列模型
- 2.5 信号量与PV操作的推论



## 2.1 发明人Dijkstra简介



### ■ 信号量的原始文献:

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



### ■ 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“检测(Proberen)”和“增量(Verhogen)”的首字母

### ■ 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

### ■ Dijkstra ( 1930年5月11日~2002年8月6日 )

- 1962年，Edsger Dijkstra在TH Eindhoven任全职教授
- 1972年，Dijkstra获得ACM图灵奖
- 1984年，Dijkstra任UTAustin教授



## 2.1 发明人Dijkstra简介

### ■ 对于计算机学科有不朽的奠基性贡献

- algorithm design //最短路径算法
- programming languages //Algol60, Pascal语言的前身
- program design, structured Programming //结构化程序设计
- operating systems //信号量&PV操作, 哲学家就餐, 银行家算法
- distributed processing
- formal specification and verification
- design of mathematical arguments

基础性的通用方法，  
对硬件无特殊要求



### ■ 南大软件所必读专著《Notes on structured programming》

计算机学科的  
奠基性工作

1965: 信号量与PV操作

操作系统的同步与互斥

1968: goto语句有害

结构化程序设计方法学

程序设计脱离原始形态

### ■ Dijkstra之于计算机科学，就相当于欧拉之于近代数学

- 拉普拉斯说: 读一读欧拉，他是所有（数学）人的老师
- 读一读Dijkstra，他是计算机科学CSer和软件工程SEer的老师

奠基性工作的意义：CS专业的程序设计由艺术变为科学，  
结构化方法是一种科学方法论



## 2.1 发明人Dijkstra简介



- 奥斯汀大学永久主页 <http://www.cs.utexas.edu/users/EWD/>


# Edsger W. Dijkstra

1930–2002

Search transcriptions:   [Advanced](#)

[search.](#)

[Home](#)  
[Search](#)  
Numerical  
EWD Index:  
[00xx](#)  
[01xx](#)  
[02xx](#)  
[03xx](#)  
[04xx](#)  
[05xx](#)  
[06xx](#)  
[07xx](#)  
[08xx](#)  
[09xx](#)  
[10xx](#)  
[11xx](#)  
[12xx](#)  
[13xx](#)  
[BibTeX index](#)  
[MC Reports](#)



©2002 Hamilton Richards

(photo)

Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are

- algorithm design
- programming languages
- program design
- operating systems
- distributed processing
- formal specification and verification
- design of mathematical arguments

In addition, Dijkstra was intensely interested in teaching, and in the relationships between academic computing science and the software

文献目录



## 2.1 发明人Dijkstra简介 学术挚友 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- )

- 剑桥大学，微软研究院剑桥分院
- 获1980年图灵奖

### ■ 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.



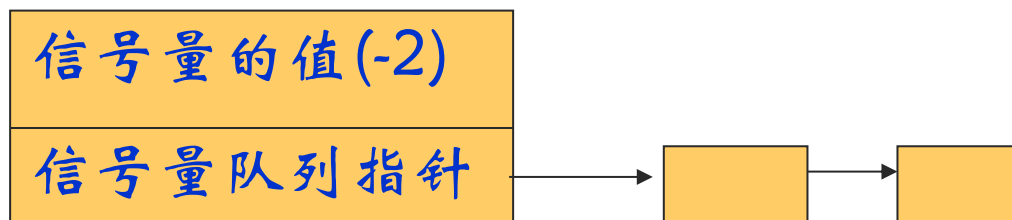
## 2.2 信号量与PV操作的数据结构与原语操作

### (1) 信号量数据结构定义



- 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
  - **原语** : CPU处于内核态, 在关中断环境下执行的一段指令序列

**原子性：不被中断，确保安全且完整执行这段指令序列**



**强调：对于信号量，只允许使用P和V原语操作访问信号量，不能直接对信号量的整型值做读写操作，也不能直接对信号量的队列做任何其他操作**





## 2.2 信号量与PV操作的数据结构与原语操作

### (2) PV原语操作含义及其伪代码



```
typedef struct semaphore {
```

#### 基本数据结构定义

```
    int value;                /* 信号量值 */
    struct pcb * list;        /* 信号量队列指针 */
}
```

```
void P(semaphore s) {
```

#### P操作原语

```
    s.value -- ;              /* 信号量值减 1 */
    if(s.value < 0)           /* 若信号量值小于 0, 执行 P 操作的进程调用
                                sleep(s.list) 阻塞自己, 被置成等待信号量 s 状态
                                并移入 s 信号量队列, 转向进程调度程序 */
        sleep(s.list);
}
```

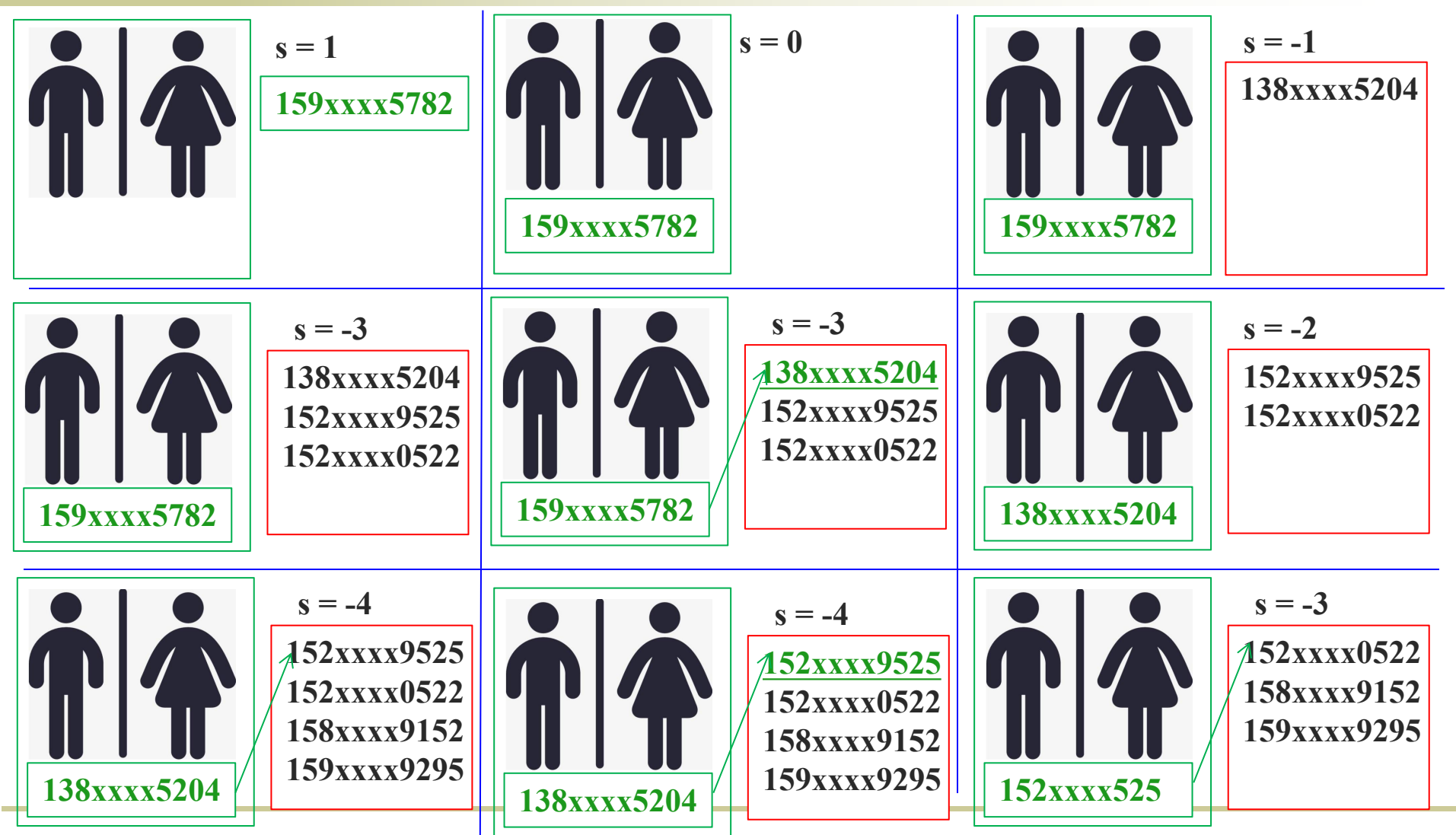
```
void V(semaphore s) {
```

#### V操作原语

```
    s.value ++ ;             /* 信号量值加 1 */
    if(s.value <= 0)         /* 若信号量值小于等于 0, 则调用 wakeup(s.list) 从信
                                号量 s 队列中释放一个等待信号量 s 的进程并转换成
                                就绪态, 进程则继续执行 */
        wakeup(s.list);
}
```



## 2.3 信号量与PV操作管理临界资源的的举例 ( 火车上的卫生间 )



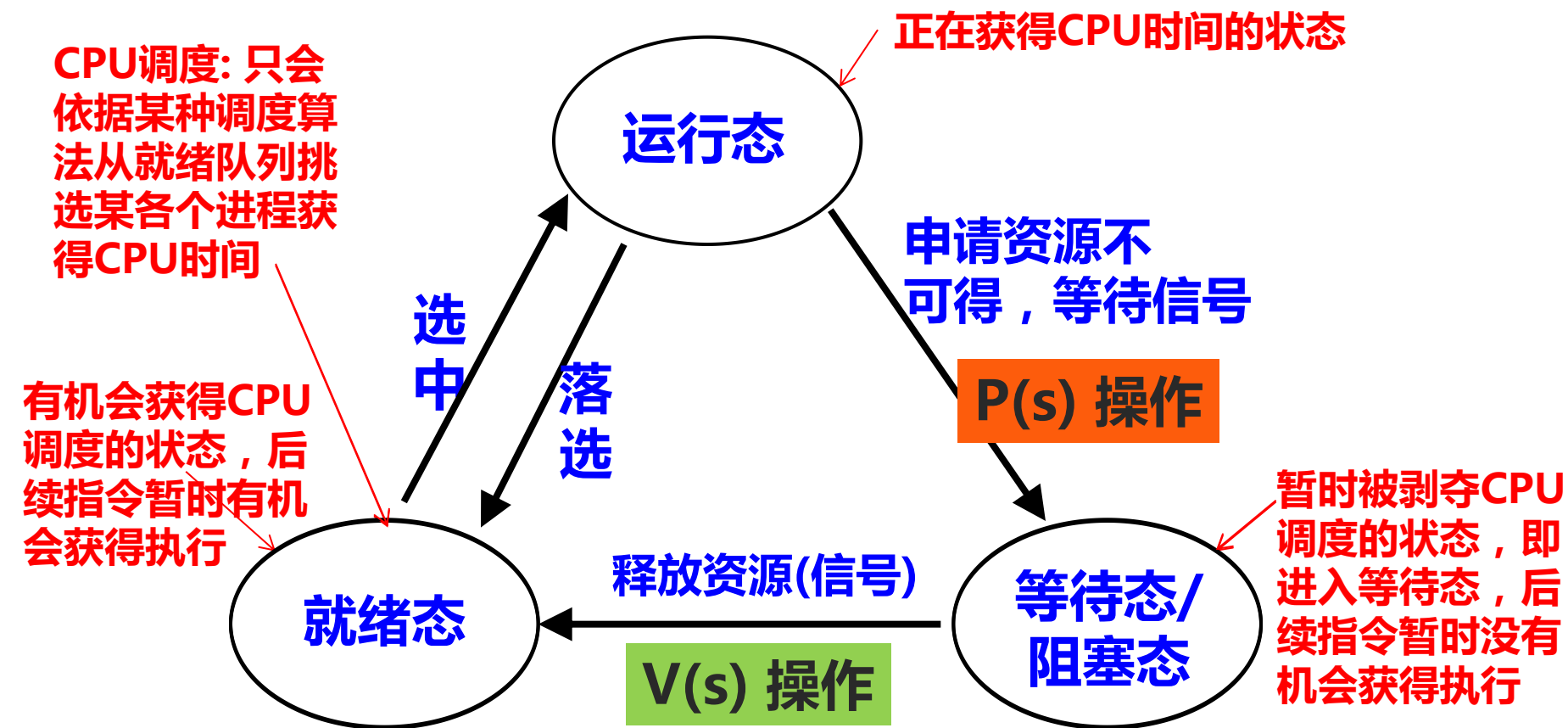




## 2.4 信号量与进程状态转换模型及其队列模型(1)



### 信号量与进程状态转换模型

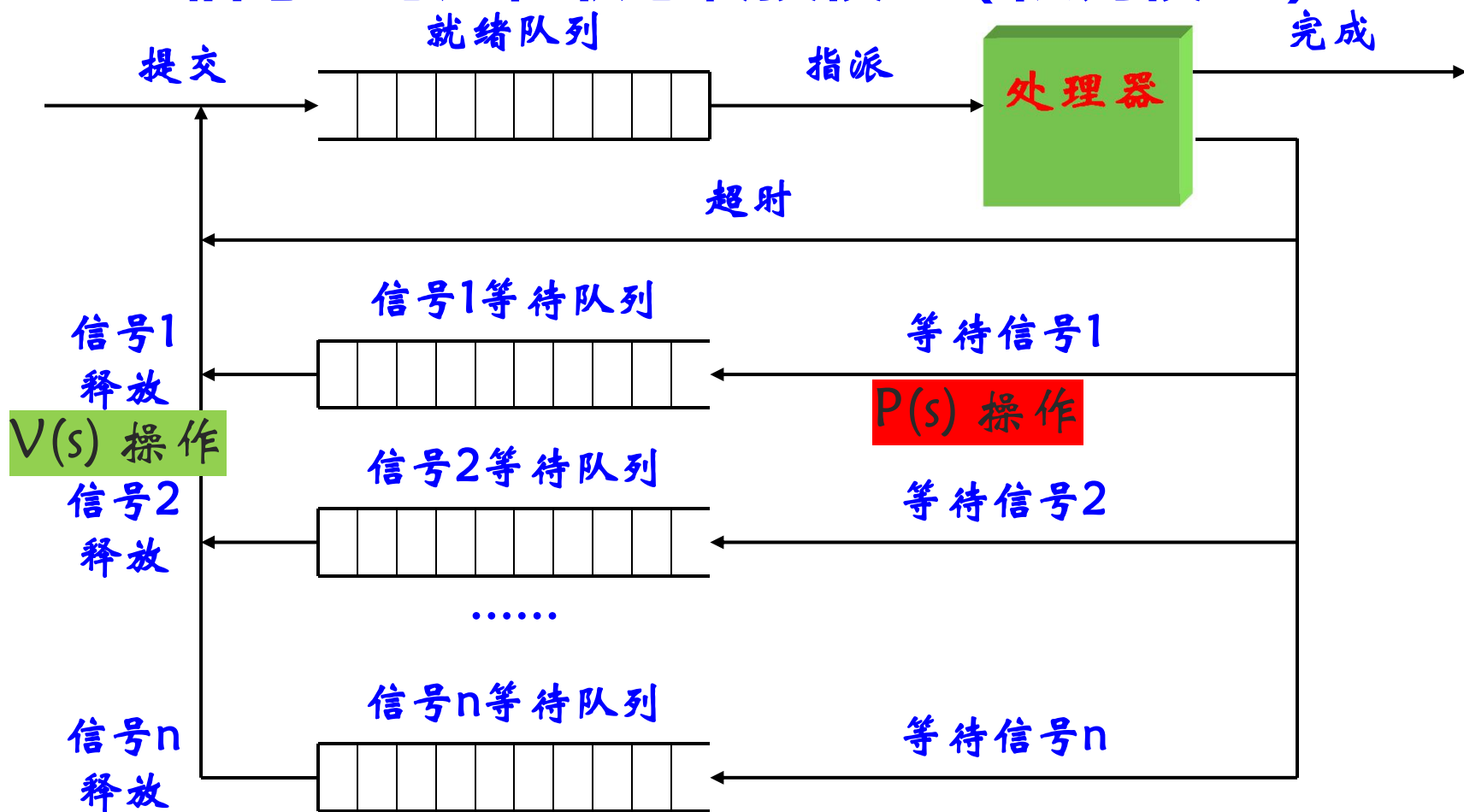




## 2.4 信号量与进程状态转换模型及其队列模型(2)



### 信号量与进程状态转换模型（队列模型）





## 2.5 信号量与PV操作的推论

- **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



# 信号量与PV操作



- 1、信号量与PV操作系统的问题背景
- 2、信号量与PV操作的基本原理
- 3、信号量的应用 ←



## 3.1 信号量程序设计的一般结构

**semaphore s=1;**

**cobegin**

**Process Pi**     **/\* i=1,...,n \*/**

**{**

...

申请进入临界区

**P(s);**

**/\* critical region 临界区 \*/**

**V(s);**

申请退出临界区

...

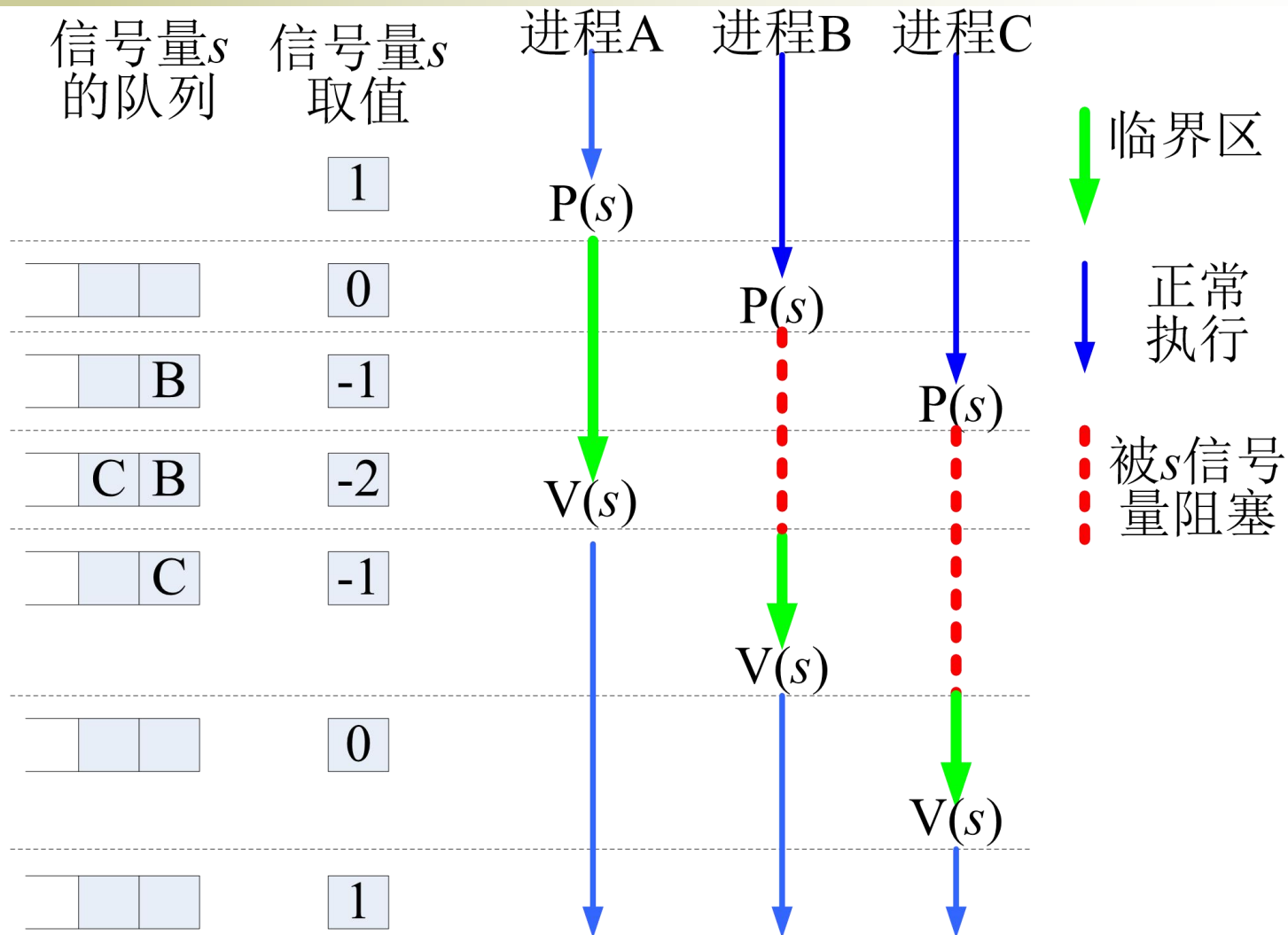
**};**

**coend**

说明: 在表达纯粹互斥关系时信号量初值为1,  
且同一个信号量的P操作和V操作处于同一个进程之中。  
但是这种情形不适用于同步关系。



# 3.2 并发程序演算举例: 信号量与PV操作控制并发进程之间的临界资源





## 3.3 求解互斥问题

### (1) 飞机票问题



```
Var A : ARRAY[1..m] of integer;  
mutex : semaphore;  
mutex:= 1;  
cobegin  
process Pi  
  var Xi:integer;  
begin  
  L1:  
    按旅客订票要求找到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
```

```
Var A : ARRAY[1..m] of integer;  
s : ARRAY[1..m] of semaphore;  
s[j] := 1;  
cobegin  
process Pi  
  var Xi:integer;  
begin  
  L1:  
    按旅客订票要求找到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
```



## 3.3 求解互斥问题

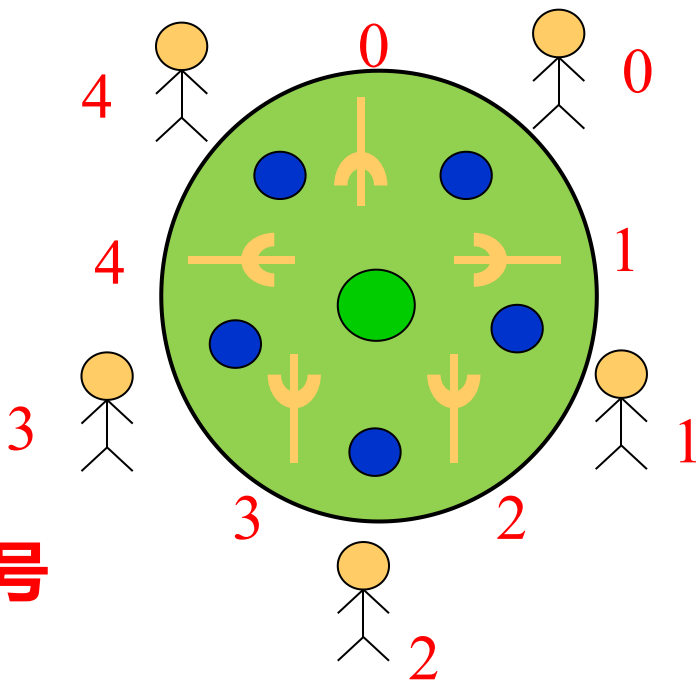
### (1) 哲学家就餐问题(a)



**问题描述：**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

**Dijkstra最早设计了哲学家就餐问题**

**哲学家顺时针编号**







## 3.3 求解互斥问题

### (2) 哲学家就餐问题(b)



```
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]);
```

```
    P(fork[(i+1)%5]);
```

```
    eat();
```

```
    V(fork[i]);
```

```
    V(fork[(i+1)%5]);
```

```
}
```

```
}
```

```
coend
```

存在什么问题?

死锁!

//先取右手的叉子

//再取左手的叉子



## 3.3 求解互斥问题

### (2) 哲学家就餐问题(c)



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 (C. A. R. Hoare方案)
- 奇数号先取左手边的叉子，偶数号先取右手边的叉子



## 3.3 求解互斥问题

### (2) 哲学家就餐问题(d)



```
semaphore fork[5];
for (int i=0;i<5;i++)
    fork[i]= 1;
semaphore room=4;
//增加一个侍者，设想有两个房间1号房间是会议室，2号房间是餐厅
cobegin
process philosopher_i() { /*i=0,1,2,3, 4 */
    while(true) {
        think();
        P(room); //控制最多允许4位哲学家进入2号房间餐厅取叉子
        P(fork[i]);
        P(fork[(i+1)%5] );
        eat();
        V(fork[i]);
        V(fork[(i+1) % 5]);
        V(room);
    }
}
coend
```



## 3.3 求解互斥问题

### (2) 哲学家就餐问题(e)



```
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]);
    }
}
```



## 3.4 求解同步问题

### (1) 生产者与消费者(a)



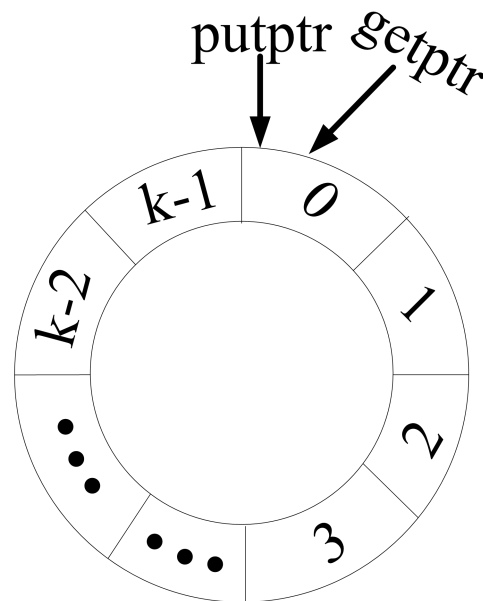
问题描述：有 $n$ 个生产者和 $m$ 个消费者，连接在一个有 $k$ 个单位缓冲区的有界缓冲上。其中，生产者进程 $Producer\_i$ 和消费者进程 $Consumer\_j$ 都是并发进程，只要缓冲区未满，生产者 $Producer\_i$ 生产的产品就可投入缓冲区；只要缓冲区不空，消费者进程 $Consumer\_j$ 就可从缓冲区取走并消耗产品

可能情形：

$n=1, m=1, k=1$

$n=1, m=1, k>1$

$n>1, m>1, k>1$





## 3.4 求解同步问题

### (1) 生产者与消费者(b)



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;



## 3.4 求解同步问题

### 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 */  
sget:=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;
```



## 3.4 求解同步问题

### 一个生产者/一个消费者/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;

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;





## 3.4 求解同步问题

### 一个生产者/一个消费者/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 */
s1:= 1; s2:= 1;
```

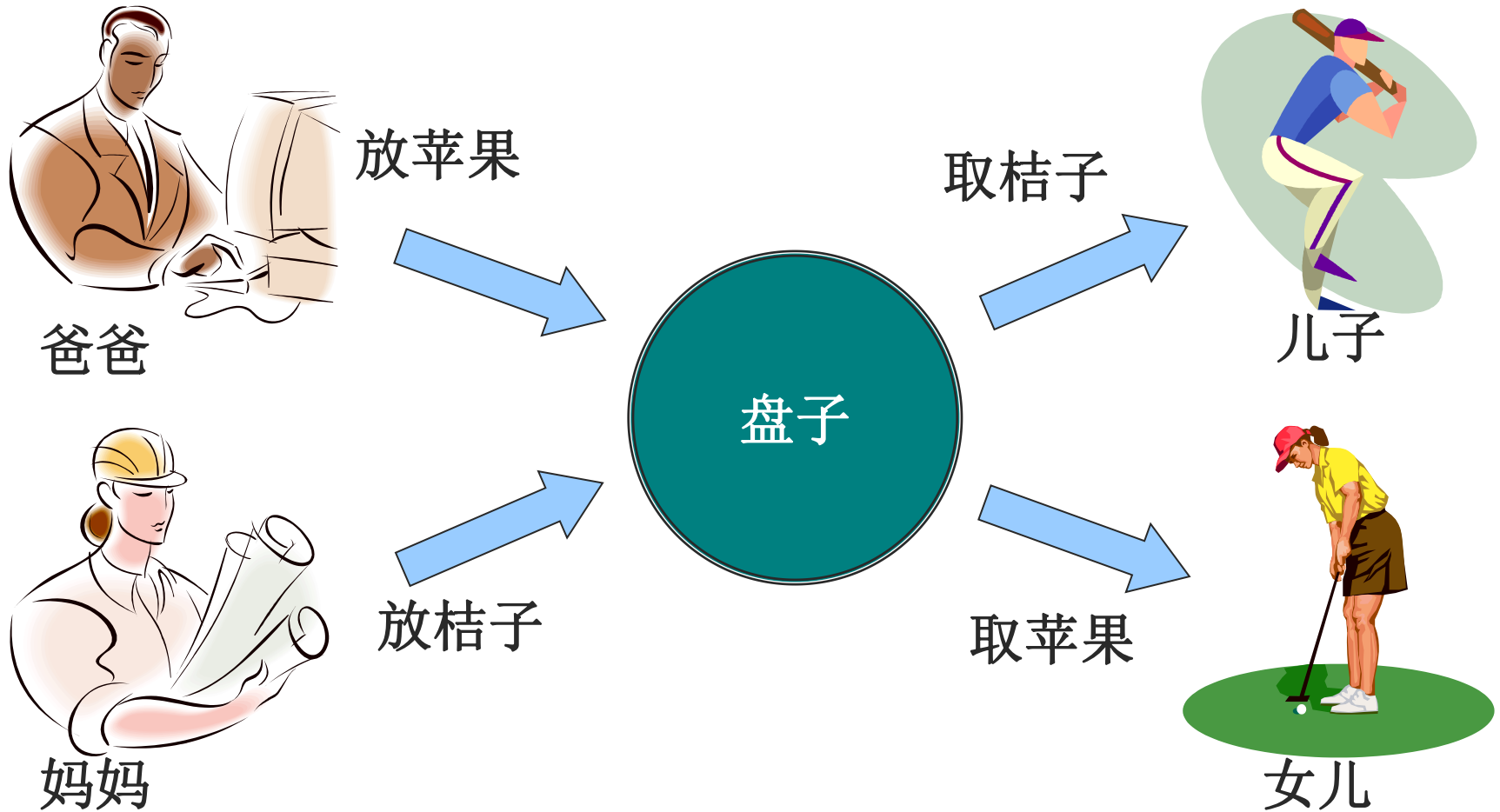
```
process Producer_i
begin
  L1:produce a product;
  P(sput);
  { P(s1);
    B[putptr] := product;
    putptr :=(putptr+1) mod k;
    V(s1);
    V(sget);
  }
  goto L1;
end;
```

```
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;
```



# 3.4 求解同步问题

## (2) 苹果-桔子问题





## 3.4 求解同步问题

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

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

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



# 信号量与PV操作

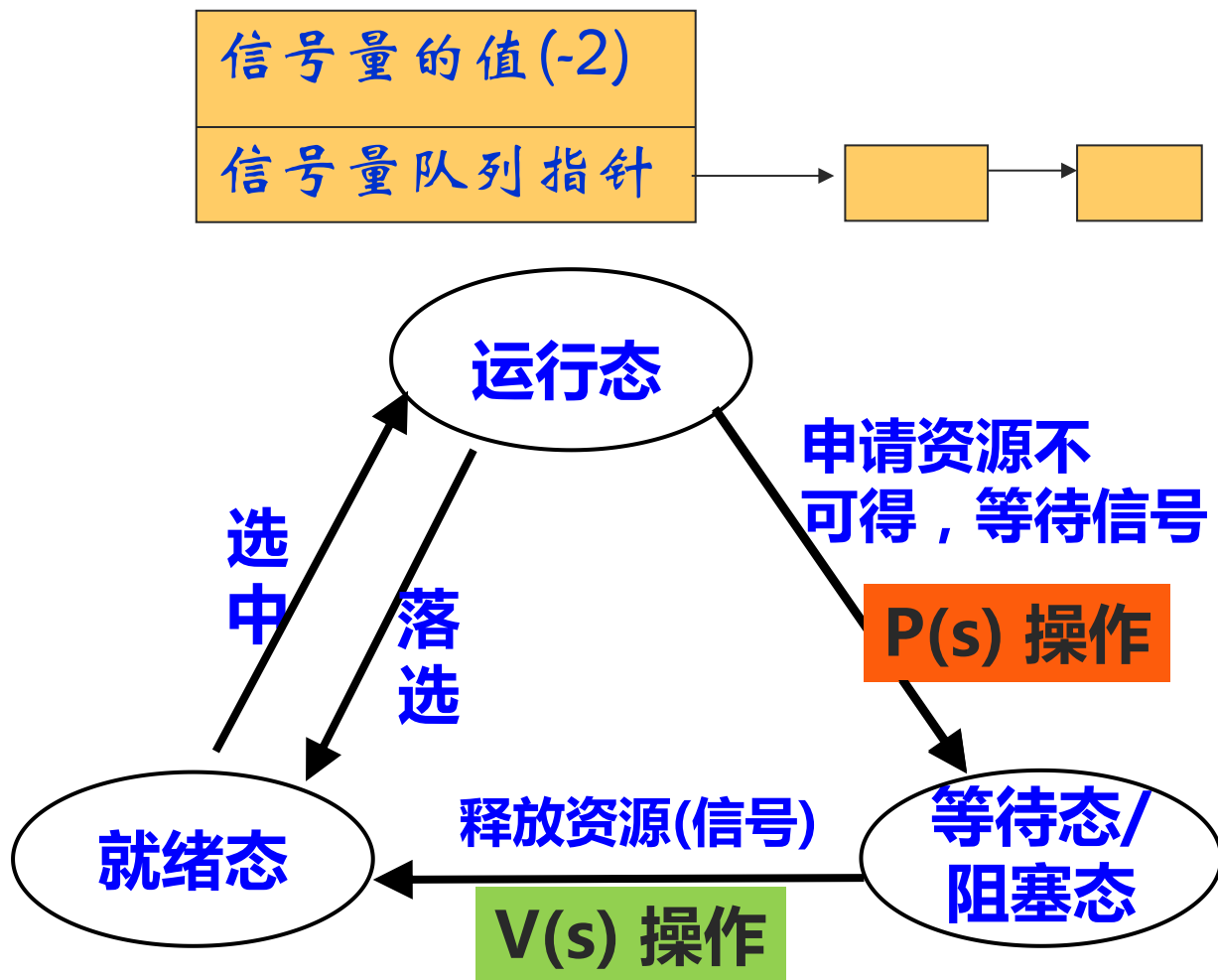


- 1、信号量与PV操作系统的问题背景
- 2、信号量与PV操作的基本原理
- 3、信号量的应用
- 4、小结 ←



## 4、小结

# 1965年Dijkstra发明的同步控制的编程方法





谢谢！



# 信号量与PV操作原著（仅一页）



## Solution of a Problem in Concurrent Programming Control

E. W. DIJKSTRA

*Technological University, Eindhoven, The Netherlands*

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

### Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

### The Problem

To begin, consider  $N$  computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these  $N$  cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the  $N$  computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the  $N$  computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each

computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

### The Solution

The common store consists of:

"Boolean array  $b$ ,  $c[1:N]$ ; integer  $k$ "

The integer  $k$  will satisfy  $1 \leq k \leq N$ ,  $b[i]$  and  $c[i]$  will only be set by the  $i$ th computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of  $k$  is immaterial.

The program for the  $i$ th computer ( $1 \leq i \leq N$ ) is:

```
"integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
    go to Li1
end
else
Li4: begin c[i] := false;
    for j := 1 step 1 until N do
        if j ≠ i and not c[j] then go to Li1
    end;
    critical section;
    c[i] := true; b[i] := true;
    remainder of the cycle in which stopping is allowed;
    go to Li0"
```

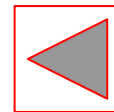
### The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement  $Li4$  without jumping back to  $Li1$ , i.e., finding all other  $c$ 's **true** after having set its own  $c$  to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to  $Li1$ ) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the  $k$ th computer is not among the looping ones,  $b[k]$  will be **true** and the looping ones will all find  $k \neq i$ . As a result one or more of them will find in  $Li3$  the Boolean  $b[k]$  **true** and therefore one or more will decide to assign " $k := i$ ". After the first assignment " $k := i$ ",  $b[k]$  becomes **false** and no new computers can decide again to assign a new value to  $k$ . When all decided assignments to  $k$  have been performed,  $k$  will point to one of the looping computers and will not change its value for the time being, i.e., until  $b[k]$  becomes **true**, viz., until the  $k$ th computer has completed its critical section. As soon as the value of  $k$  does not change any more, the  $k$ th computer will wait (via the compound statement  $Li4$ ) until all other  $c$ 's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their  $c$  **true**, as they will find  $k \neq i$ . And this, the author believes, completes the proof.

Dijkstra风格：  
惜墨如金，逻辑严谨  
行文优雅，笔锋犀利  
语不惊人，死不休





## 3.3 求解互斥问题

### (2) 哲学家就餐问题(b)



```
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
```

4号哲学家先取左手的叉子

4号哲学家, 第二步取右手的叉子





## 3.3 求解互斥问题

### (2) 哲学家就餐问题(c)



- 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 (C. A. R. Hoare方案)
  - 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 (第五版教材, Page 188, AND型信号量)



## 3.4 求解同步问题

### 一个生产者/一个消费者/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;
```