

library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

河南大学

# 操作系统

计算机学院

library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version



library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

# 第2章 进程的描述与控制

## □2.4 进程同步

## □2.5 经典进程的同步问题

# 进程同步

- **进程同步** 是指对多个相关进程在执行次序上进行协调，它的目的是使系统中诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。
- 用来实现同步的机制称为 **同步机制**。如：信号量机制；管程机制。



## 两种形式的制约关系

### ■ 系统中诸进程之间在逻辑上存在着两种制约关系

#### 1 直接制约关系（进程同步）：

为完成同一个任务的诸进程间，因需要协调它们的工作而相互等待、相互交换信息所产生的直接制约关系。

#### 2 间接制约关系（进程互斥）：

进程共享独占型资源而必须互斥执行的间接制约关系。



# 两种形式的制约关系

## ■ 同步与互斥的比较

同 步	互 斥
进程-进程	进程-资源-进程
时间次序上受到某种限制	竞争某一资源时不允许进程同时工作
相互清楚对方的存在及作用，交换信息	不一定清楚其它进程情况
往往指有几个进程共同完成一个任务	往往指多个任务、多个进程间相互制约
例：生产与消费之间，作者与读者之间	例：过独木桥



# 临界资源

- **临界资源**(Critical Resource) 是一次只允许一个进程使用的资源，如打印机、绘图机、变量、数据等。
- 进程之间采取 **互斥方式** 实现对临界资源的共享，从而实现并行政程序的封闭性。
- 引起不可再现性是因为对临界资源没有进行互斥访问。
- 在每个进程中，访问临界资源的那一段代码称为**临界区**（ Critical Section ），简称 CS 区。



## 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**

```
R1=X;  
R1=R1+1;  
X=R1;
```



**B:**

```
R2=X;  
R2=R2+1;  
X=R2;
```



## 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**

```
R1=X;  
R1=R1+1;  
X=R1;
```



**B:**

```
R2=X;  
R2=R2+1;  
X=R2;
```

A 与 B 均对 X 加 1，即 X 加 2。





## 临界资源

例：有两个进程 A 和 B，它们共享一个变量 X，且两个进程按以下方式对变量 X 进行访问和修改，其中 R1 和 R2 为处理机中的两个寄存器。

**A:**                      **R1=X;**  
                             **R1=R1+1;**  
                             **X=R1;**



**B:**                      **R2=X;**  
                             **R2=R2+1;**  
                             **X=R2;**

← 临界区

← 临界区

A 与 B 均对 X 加 1，即 X 加 2。



# 临界资源

如果按另一顺序对变量进行修改

```
A:          R1=X;  
B:          R2=X;  
A:          R1=R1+1;  
            X=R1;  
B:          R2=R2+1;  
            X=R2;
```



# 临界资源

如果按另一顺序对变量进行修改

```
A:      R1=X;  
B:      R2=X;  
A:      R1=R1+1;  
        X=R1;  
B:      R2=R2+1;  
        X=R2;
```

结果 X 只加了 1



## 临界资源

如果按另一顺序对变量进行修改

```
A:      R1=X;  
B:      R2=X;  
A:      R1=R1+1;  
        X=R1;  
B:      R2=R2+1;  
        X=R2;
```

结果 X 只加了 1

**产生错误的原因**：不加控制地访问共享变量 X  
对临界区需要进行保护（互斥访问）



# 临界区

- 为了保证临界资源的正确使用，可以把 **临界资源的访问过程** 分成以下几部分：

- **进入区**：增加在临界区前面的一段代码，用于检查欲访问的临界资源此刻是否被访问。
- **临界区**：进程访问临界资源的那段代码。
- **退出区**：增加在临界区后面的一段代码，用于将临界资源的访问标志恢复为未被访问标志。
- **剩余区**：进程中除了进入区、临界区及退出区之外的其余代码。

进入区

临界区

退出区

剩余区



## 同步机制应遵循的规则

- 1 空闲让进**：当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效地利用临界资源。
- 2 忙则等待**：当已有进程进入临界区时，表明临界资源正在被访问，因而其他试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。
- 3 有限等待**：对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，以免陷入 **死等** 状态。
- 4 让权等待**：当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入 **忙等**。



## 解决临界区（互斥）问题的几类方法

- 软件方法：用编程解决。
  - Dekker 算法
  - Peterson 算法
- 硬件方法：用硬件指令解决。
- **信号量及 P-V 操作**
- 管程



## 解决临界区（互斥）问题的几类方法

- 软件方法：用编程解决。
  - Dekker 算法
  - Peterson 算法
- 硬件方法：用硬件指令解决。
- **信号量及 P-V 操作**
- 管程

同步机制





## 软件方法：Dekker 算法的初步设想

定义全局变量 `turn`，如果 `turn=0`：P0 可以进入 CS；

如果 `turn=1`：P1 可以进入 CS。

```
int turn; /*共享的全局变量*/
```

**P0**

**P1**

...

...

**while (turn!=0) do no\_op;**

**while (turn!=1) do no\_op;**

**<CS>**

**<CS>**

**turn=1;**

**turn=0;**

...

...

## 软件方法：Dekker 算法的初步设想

定义全局变量 `turn`，如果 `turn=0`：P0 可以进入 CS；

如果 `turn=1`：P1 可以进入 CS。

```
int turn; /*共享的全局变量*/
```

**P0**

...

**while (turn!=0) do no\_op;**

**<CS>**

**turn=1;**

...

**P1**

...

**while (turn!=1) do no\_op;**

**<CS>**

**turn=0;**

...

忙等  
busy waiting



## Dekker 算法：初步设想

### ■ 出现的问题

- 进程强制交替进入临界区，容易造成资源利用不充分。
- 当  $turn=0$  时，即使此时 CS 空闲，P1 也必须等待 P0 进入 CS 执行、退出后才能进入 CS，**不符合空闲让进的原则**
- 进程不能进入自己的临界区时，没有立即释放处理机，陷入忙等，**不符合让权等待的原则**。
- 任何进程在 CS 区内、CS 区外失败，其它进程将可能因为等待使用 CS 而无法向前推进。



# Dekker 算法：改进一

使用全局共享数组 flag 标志 CS 状态：

flag[0] 或 flag[1] = true : 表示 P0 或 P1 占用 CS

flag[0] 或 flag[1] = false : 表示 CS 空闲

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

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

**P0**

**P1**

...

...

```
while (flag[1]) do no_op; while (flag[0]) do no_op;
```

```
flag[0]=true;
```

```
flag[1]=true;
```

```
<CS>
```

```
<CS>
```

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...



# Dekker 算法：改进一

## ■ 出现的问题

- 进程在 CS 内失败且相应的  $\text{flag}=\text{ture}$ ，则其它进程永久阻塞。
- **不能实现互斥！** P0 和 P1 可能同时进入临界区。当 P0 执行  $\text{while}(\text{flag}[1])$  并通过以后，在执行  $\text{flag}[0]=\text{true}$  之前，P1 执行  $\text{while}(\text{flag}[0])$ ，这样两个进程同时进入了临界区。



# Dekker 算法：改进一

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

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

**P0**

**P1**

...

...

```
while (flag[1]) do no_op;① while (flag[0]) do no_op;②
```

```
flag[0]=true;③
```

```
flag[1]=ture;④
```

```
<CS>
```

```
<CS>
```

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...

按图中①②③④的次序执行，P0 和 P1 都可进入 CS 区。

不能实现互斥！



# Dekker 算法：改进二，改为先置标志位

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

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

**P0**

**P1**

...

...

```
flag[0]=true;
```

```
flag[1]=ture;
```

```
while (flag[1]) do no_op;
```

```
while (flag[0]) do no_op;
```

```
<CS>
```

```
<CS>
```

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...



## Dekker 算法：改进二

### ■ 出现的问题

- 不能实现空闲让进，有限等待。P0 和 P1 可能都进入不了临界区。
- 当 P0 执行了  $\text{flag}[0] = \text{true}$  后，P1 执行了  $\text{flag}[1] = \text{true}$ ，这样两个进程都无法进入临界区（阻塞）。





# Dekker 算法：改进二

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

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

**P0**

**P1**

...

...

```
flag[0]=true; ①
```

```
flag[1]=ture; ②
```

```
while (flag[1]) do no_op; ③ while (flag[0]) do no_op; ④
```

```
<CS>
```

```
<CS>
```

```
flag[0]=false;
```

```
flag[1]=false;
```

...

...

按图中①②③④的次序执行，都无法进入各自的 CS 区。



# 成功的 Dekker 算法

同时使用flag和turn

**Process P0**

```
begin
  flag[0]:=true;
  while (flag[1])
  { if turn=1 then
    begin
      flag[0]:=false;
      while (turn=1) do no-op;
      flag[0]:=true;
    end
  };
  临界区;
  turn = 1;
  flag[0]:=false;
end;
```

While循环体避免  
“改进2”中的死锁



## Peterson 算法

- 代码更简洁
- 设两个全局共享变量：flag[0]、flag[1]，表示临界区状态及哪个进程正在占用 CS。
- 设一个全局共享变量 turn：表示能进入 CS 的进程序号。



# Peterson 算法

```
bool flag[2]; /*共享的全局数组，BOOL类型*/
```

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

**P0**

...

```
flag[0]=true;
```

```
turn=1;
```

```
while (flag[1] and turn==1)
```

```
do no_op;
```

```
<CS>
```

```
flag[0]=false;
```

...

**P1**

...

```
flag[1]=ture;
```

```
turn=0;
```

```
while (flag[0] and turn==0)
```

```
do no_op;
```

```
<CS>
```

```
flag[1]=false;
```

...



# Peterson 算法

- 在进入区先修改后检查
  - 检查对方 flag：如果不在临界区则自己进入（空闲让进）。
  - 否则再检查 turn：turn 保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入（先到先入）。
- 当  $\text{flag}[1]=\text{false}$  或  $\text{turn} = 0$ ，即当进程 1 没有要求进入 CS，或仅允许进程 0 进入 CS 时，P0 进入 CS。



## 硬件方法

- 关中断指令
- TestAndSet 指令
- SWAP 指令



## 中断禁用 (关中断, Interrupt Disabling)

- 如果进程访问临界资源时 (执行临界区代码) 不被中断, 就可以利用它来保证互斥地访问。
- 方法: 使用关中断、开中断原语。
- 过程
  - 关中断
  - 临界区
  - 开中断
  - 剩余区
- 存在问题
  - 代价高: 限制了处理器交替执行各进程的能力。
  - 不能用于多处理器结构: 关中断不能保证互斥。



# TestAndSet 指令 (TS 指令) (测试并设置)

TS指令定义(逻辑)

```
Boolean TestAndSet (int i)
{
    if (i==0)
    {
        i=1;
        return true;
    }
    else
        return false;
}
```

使用TestAndSet实现互斥

```
int lock;           //lock取0或1
...                //0开, 1锁
While(!TestAndSet(lock));
临界区;           //此时lock=1
lock=0;           //开锁
剩余区
...
```

- TS指令管理临界区时，把一个临界区与一个变量lock相连，由于变量lock代表了临界资源的状态，可把它看成一把锁。
- TS指令自动整体执行，不响应任何中断，故可实现进程互斥。





## Swap 指令 ( 交换 )

Swap指令：

```
void Swap (int a, int b)
{
    int temp=a;
    a = b;
    b=temp ;
}
```

使用Swap指令实现互斥

- 每个临界资源设置一个公共变量lock，初值为0(开锁)。
- 进程要使用临界资源时首先把私有变量key置为1。

key=1;

```
do{
    Swap (lock, key);
} while (key);
```

临界区  
lock=0;  
剩余区

0为false  
非0为true



## 互斥与同步解决方法

- 软件方法
  - 实现比较复杂，需要较高的编程技巧。
- 硬件方法
  - 不能实现让权等待。
  - 可能出现死锁。



## 互斥与同步解决方法

- 软件方法
  - 实现比较复杂，需要较高的编程技巧。
- 硬件方法
  - 不能实现让权等待。
  - 可能出现死锁。

有效解决进程同步问题的方法  
**信号量机制**



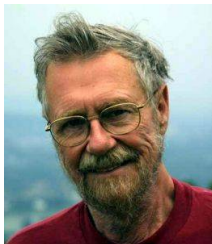
# 信号量机制

- **信号量机制**是荷兰科学家 E. W. Dijkstra 在 1965 年提出的一种同步机制，也称为**P、V 操作**。
- **信号量**
  - 用于表示资源数目或请求使用某一资源的进程个数的整型量。
- 1 整型信号量
- 2 记录型信号量
- 3 AND 信号量集
- 4 一般信号量集



# Dijkstra

- Dijkstra(1930~2002), 荷兰计算机科学家, 计算机先驱之一, 1972 年第七位图灵奖获得者。
- ALGOL 语言的主要贡献者, 提出了结构化程序设计结构, 曾经提出 “goto 有害论”, 提出了信号量和 PV 原语, 解决了 “哲学家聚餐” 问题。
- Dijkstra 的创新思想包括: 结构编程、堆栈、矢量、信号量、同步进程和死锁。



# 信号量机制

- 信号量只能通过**初始化**和**两个标准的原语（P，V 操作）**来访问。
- **初始化**：指定一个整数值，表示空闲资源总数。
- **P 操作也称为 wait 操作，V 操作也称为 signal 操作。**
- 信号量是比锁更高级的资源抽象方式。
- **注意**：P、V 操作应作为一个整体实施，不允许分割。



## 整型信号量

- **整型信号量**：非负整数，用于表示资源数目。除了初始化外，只能通过两个原子操作 wait 和 signal ( P , V ) 来访问。
- wait 和 signal 操作描述：

```
wait(S) : while S ≤ 0 do no-op      // 测试有无可用资源  
          S := S - 1;                // 可用资源数减一  
signal(S) : S := S + 1;
```

- 主要问题：只要  $S \leq 0$ ，wait 操作就不断地测试（**忙等**），因而未做到“让权等待”。



## 记录型信号量

- 为了解决“让权等待”问题，需要引入阻塞队列，信号量值可以取负值——**记录型信号量**。
- 设置一个代表资源数目的整型变量 value（资源信号量）。
- 设置一链表指针 L 用于链接所有等待的进程。





# 记录型信号量的数据结构

```
Type semaphore=record
```

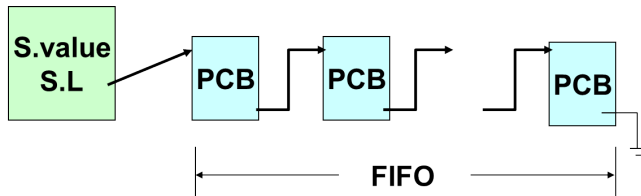
```
    value:integer;
```

```
    L: list of process;
```

```
end
```

1 S.value : 信号量值

2 S.L : 进程等待队列



## s.value 的物理含义

- 执行一次 P(s) 操作，意味着进程请求分配该类资源一个单位的资源。
- 执行一次 V(s) 操作意味着进程释放相应资源一个单位的资源。当值小于等于 0 时，表明有进程被阻塞，需要唤醒。
- 在记录型信号量机制中：
  - $S.value > 0$ ：表示系统中某类资源当前可用的数目。
  - $S.value \leq 0$ ：表示该类资源已分配完。若有进程请求该类资源，则被阻塞，其绝对值表示该信号量链表中因申请该资源而被阻塞进程的数目。

例如：10 个进程，5 台打印机。



## 记录型信号量

记录型信号量 wait 和 signal 操作描述：

```
wait(S) : S.value:=S.value-1;  
          if S.value<0 then block(S.L); // 让权等待  
signal(S): S.value:=S.value+1;  
          if S.value≤0 then wakeup(S.L);
```

- 1  $S.value > 0$  : 表示系统中某类资源当前可用的数目。
- 2  $S.value \leq 0$  : 其绝对值表示该信号量链表中因申请该资源而被阻塞进程的数目。



## P 操作过程描述

```
P(S) {  
    lockout interrupt ; //关中断  
    S=S-1 ;             //可用资源数减1  
    if(S<0) {  
        status= "blocked" ; //状态=阻塞  
        Insert(Q); //插入相应阻塞队列Q中  
        unlock interrupt;    //开中断  
        Scheduler;          //进程调度  
    }  
    else unlock interrupt ;   //开中断  
}
```

让权等待



## V 操作过程描述

```
V(S) {  
    lockout interrupt;           //关中断  
    S=S+1;  
    If( S≤0) {  
        A = Remove(Q);  
        //从相应阻塞队列Q中取出队首进程A  
        status(A) = "ready" ; //A状态 = 就绪  
        Insert(A, RL);          //A插入就绪队列RL  
        length(RL) = length(RL)+1;  
        //就绪队列RL长度加1  
    }  
    unlock interrupt             //开中断  
}
```



## AND 型信号量

- 上述的进程互斥问题，是针对各进程之间只共享一个临界资源而言的。
- 在有些应用场合，是一个进程需要先获得两个或更多的共享资源后方能执行，但这种情况可能发生死锁。
- 假定现有两个进程 A 和 B，他们都要求访问共享数据 D 和 E。可为这两个数据分别设置用于互斥的信号量 Dmutex 和 Emutex，并令它们的初值都是 1。

process A	process B
wait(Dmutex) ;	wait(Emutex) ;
wait(Emutex) ;	wait(Dmutex) ;



## AND 型信号量

- 若进程 A 和 B 按下述次序交替执行 wait 操作：

process A: wait(Dmutex) ; 于是 Dmutex=0

process B: wait(Emutex) ; 于是 Emutex=0

process A: wait(Emutex) ; 于是 Emutex=-1 **A 阻塞**

process B: wait(Dmutex) ; 于是 Dmutex=-1 **B 阻塞**

- 最后，进程 A 和 B 处于僵持状态。在无外力作用下，两者都将无法从僵持状态中解脱出来。我们称此时的进程 A 和 B 已进入**死锁**状态。

为避免死锁，可以采用**AND 型信号量**。



# AND 型信号量

## ■ AND 型信号量的基本思想

- 将进程在整个运行过程中所需要的所有临界资源，一次性全部分配给进程，待进程使用完后再一起释放。
- 只要有一个资源未能分配给进程，其它所有可能分配的资源也不分配给该进程。从而可避免死锁发生。

## ■ AND 型信号量集 P 原语为 Swait(Simultaneous Wait)，V 原语为 Ssignal(Simultaneous Signal)。

- 在 Swait 时，各个信号量的次序并不重要，虽然会影响进程归入哪个阻塞队列，但是因为是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。





# AND 型信号量

```
1 Swait(S1; S2; ...; Sn)    // P 原语 ;
2 {
3     if (S1>=1 && S2>=1 && ... && Sn>=1)
4         {                    // 满足全部资源要求才进行减 1 操作
5             for (i=1; i<=n; i++)
6                 Si--;
7         }
8     else
9         {调度进程进入第一个小于
10          1 的信号量的等待队列 Si.L;
11         }
12 }
```



## 一般信号量集

- **一般信号量集**是同时需要多种资源、每种占用的数目不同、且可分配资源还存在一个临界值时的信号量处理。
- 一般信号量集的基本思路就是在 AND 型信号量集的基础上扩充，在一次原语操作中完成所有的资源申请。
- 进程对信号量  $S_i$  的测试值为  $t_i$ （表示信号量的判断条件，要求当资源数量低于  $t_i$  时，便不予分配），占用值为  $d_i$ （表示资源的申请量，即  $S_i = S_i - d_i$  和  $S_i = S_i + d_i$ ）。
- 一般信号集的特点
  - 一次可分配多个某种临界资源，不需执行多次 P 操作。
  - 每次分配前都测试该种资源数目是否大于测试值。



# 一般信号量集

```
Swait(S1, t1, d1; ...; Sn, tn, dn);
Ssignal(S1, d1; ...; Sn, dn);    // 释放时不必考虑  $t_i$ 
```

- 一般信号量集可以用于各种情况的资源分配和释放。下面是几种特殊的情况：
  - Swait(S, d, d)：表示每次申请d个资源，当资源数量少于d个时，便不予分配。
  - Swait(S, 1, 1)：蜕化为一般的记录型信号量 ( $S > 1$  时) 或互斥信号量 ( $S = 1$ )。
  - Swait(S, 1, 0)：可作为一个可控开关( $S \geq 1$  时，允许多个进程进入某特定区； $S = 0$  时禁止任何进程进入)。



# 一般信号量集

$S_i$ : 可用资源数     $t_i$ : 阈值     $d_i$ : 申请资源数

```
1  Swait( $S_1$ ,  $t_1$ ,  $d_1$ ; ...,  $S_n$ ,  $t_n$ ,  $d_n$ )
2  if  $S_1 \geq t_1$  and ... and  $S_n \geq t_n$  then
3      for  $i:=1$  to  $n$  do
4           $S_i := S_i - d_i$ ;
5      end
6  else
7      Place the executing process in the waiting queue.
8  end
9
10 Ssignal( $S_1$ ,  $d_1$ ; ...,  $S_n$ ,  $d_n$ )
11 for  $i:=1$  to  $n$  do
12      $S_i := S_i + d_i$ ;
13     Remove the process waiting in the queue associated with  $S_i$ .
14 end
```



## 课堂练习

- 1 若 P、V 操作的信号量 S 初值为 2，当前值为 -1，则表示有（ ）个阻塞进程。
  - A.0 个    B.1 个    C.2 个    D.3 个
- 2 若有三个进程共享一个程序段，且每次最多允许两个进程进入该程序段，则信号量的初值应置为（ ）。
  - A.3    B.1    C.2    D.0
- 3 如果有 4 个进程共享同一程序段，每次允许 3 个进程进入该程序段，若用 PV 操作作为同步机制，则信号量的取值范围是（ ）。
  - A. 4 3 2 1 -1    B. 2 1 0 -1 -2
  - C. 3 2 1 0 -1    D. 2 1 0 -2 -3



## 课堂练习

- 1 若 P、V 操作的信号量 S 初值为 2，当前值为 -1，则表示有 ( ) 个阻塞进程。
  - A.0 个    B.1 个    C.2 个    D.3 个
- 2 若有三个进程共享一个程序段，且每次最多允许两个进程进入该程序段，则信号量的初值应置为 ( )。
  - A.3    B.1    C.2    D.0
- 3 如果有 4 个进程共享同一程序段，每次允许 3 个进程进入该程序段，若用 PV 操作作为同步机制，则信号量的取值范围是 ( )。
  - A. 4 3 2 1 -1    B. 2 1 0 -1 -2
  - C. 3 2 1 0 -1    D. 2 1 0 -2 -3

1B    2C    3C



## 课堂练习

- 4 有  $m$  个进程共享同一临界资源（每次只允许一个进程访问该临界资源），若使用信号量机制实现对临界资源的互斥访问，则信号量值的变化范围是（ ）。
- 5 设系统有  $n$  ( $n > 2$ ) 个进程，且当前不在执行进程调度程序，试考虑下述 4 种情况，不可能发生的是（ ）。
- A. 没有运行进程，有 2 个就绪进程， $n-2$  个进程阻塞
  - B. 有 1 个运行进程，没有就绪进程， $n-1$  个进程阻塞
  - C. 有 1 个运行进程，有 1 个就绪进程， $n-2$  个进程阻塞
  - D. 有 1 个运行进程， $n-1$  个就绪进程，没有进程阻塞



## 课堂练习

- 4 有  $m$  个进程共享同一临界资源（每次只允许一个进程访问该临界资源），若使用信号量机制实现对临界资源的互斥访问，则信号量值的变化范围是（ ）。
- 5 设系统有  $n$  ( $n > 2$ ) 个进程，且当前不在执行进程调度程序，试考虑下述 4 种情况，不可能发生的是（ ）。
- A. 没有运行进程，有 2 个就绪进程， $n-2$  个进程阻塞
  - B. 有 1 个运行进程，没有就绪进程， $n-1$  个进程阻塞
  - C. 有 1 个运行进程，有 1 个就绪进程， $n-2$  个进程阻塞
  - D. 有 1 个运行进程， $n-1$  个就绪进程，没有进程阻塞

41 至  $-(m-1)$  5A





## 课堂练习

6 对于两个并发进程，设互斥信号量为 mutex。当 mutex=0 时，则（ ）。

- A. 表示没有进程进入临界区
- B. 表示有一个进程进入临界区
- C. 表示有一个进程进入临界区，另一个进程等待进入
- D. 表示有两个进程进入临界区



## 课堂练习

6 对于两个并发进程，设互斥信号量为 mutex。当 mutex=0 时，则（ ）。

- A. 表示没有进程进入临界区
- B. 表示有一个进程进入临界区
- C. 表示有一个进程进入临界区，另一个进程等待进入
- D. 表示有两个进程进入临界区

6B

A:mutex=1

B:mutex=0

C:mutex=-1

D: 错误



library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

河南大学

# 操作系统

计算机学院

library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version



library computing CPU  
mainframe task web apps  
desktop task program  
applications OS  
user interface allocation  
file system resources  
storage data development  
security real-time memory  
command processor web server  
process

operating system

instruction multi-tasking  
networking software  
hardware computer  
graphical interface  
GUI management  
interrupt phone  
control virtual memory  
input output  
device driver  
multi-user  
game console  
supercomputer  
services microcomputer  
version

## 第 2 章

# 进程的描述与控制



**1** 2.4 进程同步

**2** 2.5 经典进程同步问题

**3** 经典同步问题例题

**4** 管程机制

**5** 2.6 进程通信

**6** 本章作业



## 1 2.4 进程同步

## 2 2.5 经典进程同步问题

## 3 经典同步问题例题

## 4 管程机制

## 5 2.6 进程通信

## 6 本章作业



# 信号量的应用

## 信号量的应用

- 利用信号量实现进程互斥
- 利用信号量实现进程同步（前趋关系）



## 利用信号量实现进程互斥

- 利用信号量可以方便地解决临界区问题（进程互斥）。
- 为临界资源设置一互斥信号量 mutex，初值为 1。
- 实现进程 P1 和 P2 互斥的描述：

进程

P1

...

wait(mutex);

critical section;

signal(mutex);

...

进程

P2

...

wait(mutex);

critical section;

signal(mutex);

...





## 利用信号量实现进程互斥

例：进程 A 与进程 B 共享同一文件 file，设此文件要求互斥使用，则可将 file 作为临界资源，有关 file 的使用程序段分别为临界区 CSA 和 CSB。



## 利用信号量实现进程互斥

例：进程 A 与进程 B 共享同一文件 file，设此文件要求互斥使用，则可将 file 作为临界资源，有关 file 的使用程序段分别为临界区 CSA 和 CSB。

解：`semaphore mutex=1;`



## 利用信号量实现进程互斥

例：进程 A 与进程 B 共享同一文件 file，设此文件要求互斥使用，则可将 file 作为临界资源，有关 file 的使用程序段分别为临界区 CSA 和 CSB。

解：semaphore mutex=1;

PA:

L1 : P(mutex);

CSA;

V(mutex);

remainder of process A;

GOTO L1;

PB:

L2 : P(mutex);

CSB;

V(mutex);

remainder of process B;

GOTO L2;



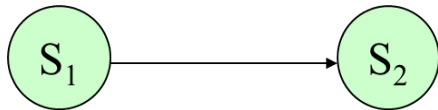
## 利用信号量实现进程互斥

- 为使多个进程能互斥地访问某临界资源，只须为该资源**设置一个互斥信号量 mutex，并设其初始值为 1**，然后将各进程访问该资源的临界区 CS 置于 wait(mutex) 和 signal(mutex) 操作之间即可。
- 在进程互斥中使用 P、V 操作，须**在同一程序段成对出现**同一信号量的 P、V 操作，否则会造成系统瘫痪。



## 利用信号量实现进程同步

- 信号量可用来描述程序或语句间的前趋关系。
- 设有两个并发进程 A 和 B，A 中有语句 S1，B 中有语句 S2，要求 S1 执行后再执行 S2。



# 利用信号量实现进程同步



- 为实现这种前趋关系，须使进程 A 和 B 共享一个公用信号量 S，并赋予初值为 0。

process A

...  
S1;  
V(S);  
...

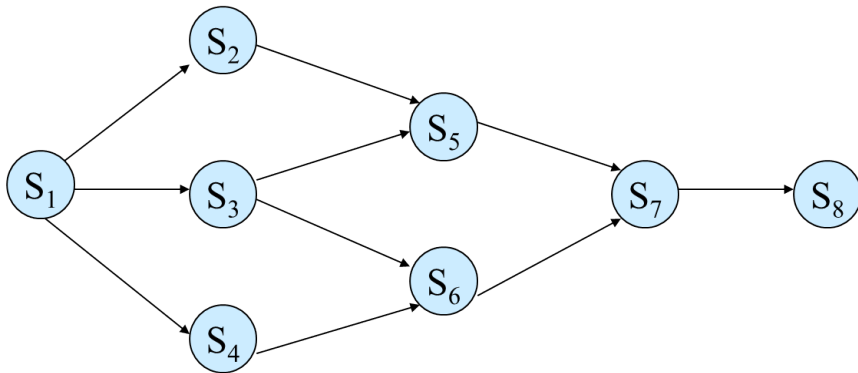
process B

...  
P(S);  
S2;  
...



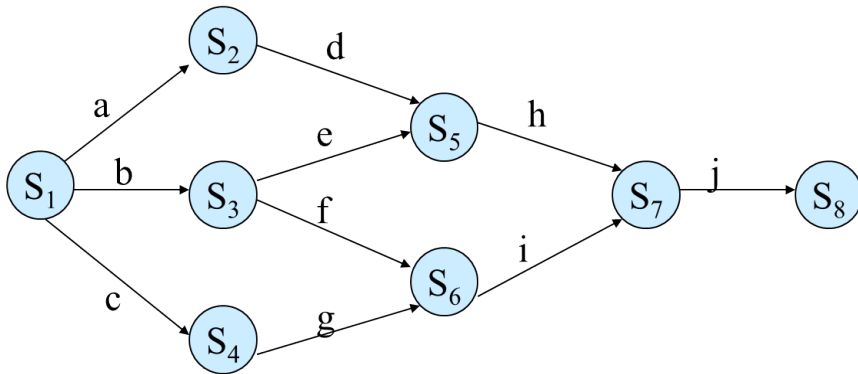
# 利用信号量实现进程同步

例：利用信号量来描述前趋图关系



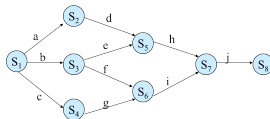
## 利用信号量实现进程同步

这是一个具有 8 个结点的前趋图。图中的前趋图中共有 10 条有向边，可设 10 个信号量，初值均为 0；有 8 个结点，可设计成 8 个并发进程，具体描述如下：





# 利用信号量实现进程同步

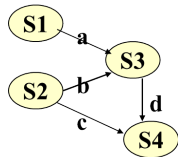


```

1  Struct smaphore a,b,c,d,e,f,g,h,i,j=0,0,0,0,0,0,0,0,0,0,0
2  cobegin
3      {S1;V(a);V(b);V(c);}
4      {P(a);S2;V(d);}
5      {P(b);S3;V(e);V(f);}
6      {P(c);S4;V(g);}
7      {P(d);P(e);S5;V(h);}
8      {P(f);P(g);S6;V(i)}
9      {P(h);P(i);S7;V(j);}
10     {P(j);S8;}
11 coend
  
```

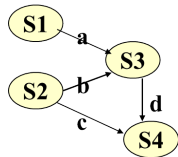
## 利用信号量实现进程同步

例：设 S1,S2,S3,S4 为一组合作进程，其前趋图如图所示，用 P、V 操作实现其同步。



# 利用信号量实现进程同步

例：设 S1,S2,S3,S4 为一组合作进程，其前趋图如图所示，用 P、V 操作实现其同步。



```

var a,b,c,d:semaphore:=0,0,0,0;
main()
{
  cobegin
    S1()    S2()    S3()    S4()
    {      {      {      {
      S1;   S2;   P(a);   P(c);
      S2(); V(a); V(b);   P(b); P(d);
      S3();      V(c);   S3;   S4;
      S4();      }      V(d); }
    }
  coend
}
  
```



## 利用信号量实现进程同步

例：已知一个求值公式  $(A^2 + 3B)/(B + 5A)$ ，若 A,B 已赋值，利用信号量来描述该公式求值过程的前趋图。



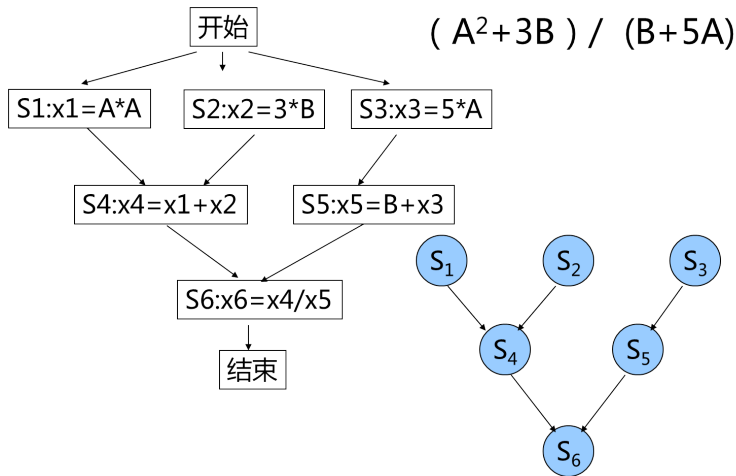
## 利用信号量实现进程同步

例：已知一个求值公式  $(A^2 + 3B)/(B + 5A)$ ，若 A,B 已赋值，利用信号量来描述该公式求值过程的前趋图。

在该公式的求值过程中，有些运算分量的执行是可以并发执行的。  
为了描述方便，可设置一些中间变量保存中间结果，并给每个语句命名。



# 利用信号量实现进程同步



# 利用信号量实现进程同步

Struct smaphore a,b,c,d,e=0,0,0,0,0

cobegin

{S1;V(a);}

{S2;V(b);}

{S3;V(c);}

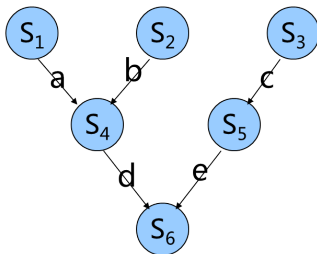
{P(a); P(b); S4;V(d);}

{P(c);S5;V(e);}

{P(d);P(e);S6;}

coend

$$(A^2 + 3B) / (B + 5A)$$



## wait、signal 操作小结

- **wait、signal 操作必须成对出现**，有一个 wait 操作就一定有一个 signal 操作。
  - 当为互斥操作时，它们同处于同一进程。
  - 当为同步操作时，则不在同一进程中出现。
- **如果两个 wait 操作相邻，那么它们的顺序至关重要**，而两个相邻的 signal 操作的顺序无关紧要。
- 一个同步 wait 操作与一个互斥 wait 操作在一起时，**同步 wait 操作在互斥 wait 操作前**。
- 优点：简单，而且表达能力强（用 wait、signal 操作可解决任何同步互斥问题）。
- 缺点：不够安全，**P、V 操作使用不当会出现死锁**。





1 2.4 进程同步

2 2.5 经典进程同步问题

3 经典同步问题例题

4 管程机制

5 2.6 进程通信

6 本章作业



## 经典进程同步问题

在多道程序环境下，进程同步问题十分重要，出现一系列经典的进程同步问题，其中有代表性有：

### 经典进程同步问题

- 生产者—消费者问题
- 哲学家进餐问题
- 读者—写者问题

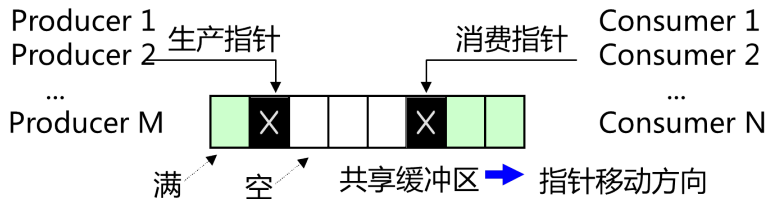


## 生产者—消费者问题

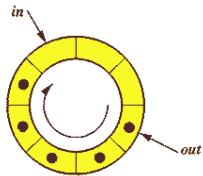
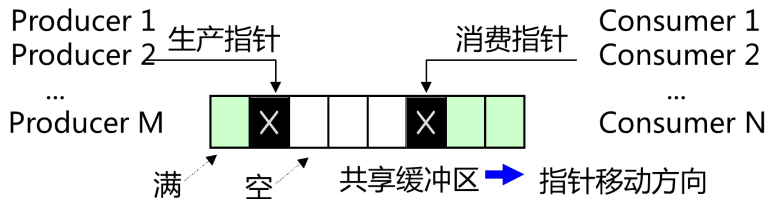
- “生产者—消费者”问题 (producer/consumer problem) 是最著名的进程同步问题。
- 它描述了一组生产者向一组消费者提供产品，它们共享一个有界缓冲池，生产者向其中投放产品，消费者从中取得产品。
- 它是许多相互合作进程的抽象，如输入进程与计算进程；计算进程与打印进程等。



# 生产者—消费者问题



# 生产者—消费者问题



## 生产者—消费者问题

- 设置两个资源信号量及一个互斥信号量。
- 资源信号量 `empty` : 说明空缓冲区的数目, 其初值为有界缓冲池的大小  $n$ 。
- 资源信号量 `full` : 说明满缓冲区的数目 (即产品数目), 其初值为 0。  $full + empty = n$ 。
- 互斥信号量 `mutex`: 说明该有界缓冲池是一个临界资源, 必须互斥使用, 其初值为 1。



# 生产者—消费者问题

## “生产者—消费者”问题的同步算法描述

- semaphore **full**=0; /\* 表示满缓冲区的数目 \*/
- semaphore **empty**=n; /\* 表示空缓冲区的数目 \*/
- semaphore **mutex**=1; /\* 表示对缓冲区进程操作的互斥信号量 \*/



## 生产者—消费者问题

```
Var mutex,empty,full: semaphore:=1,n,0 ;  
    buffer:array[0,...,n-1] of item ;  
    in,out: integer:=0,0 ;  
begin  
    parbegin  
        producer: begin  
            repeat  
                producer an item nextp ;  
                wait(empty) ;  
                wait(mutex) ;  
                buffer(in):=nextp ;  
                in:=(in+1) mod n ;  
                signal(mutex) ;  
                signal(full) ;  
            until false ;  
        end
```





## 生产者—消费者问题

```
consumer: begin
    repeat
        wait(full) ;
        wait(mutex) ;
        nextc:=buffer(out) ;
        out:=(out+1) mod n ;
        signal(mutex) ;
        signal(empty) ;
        consumer the item in nextc ;
    until false ;
end
parend
end
```



# 生产者—消费者问题

## 生产者

P(empty) ;	
P(mutex) ;	//进入区
放入产品 ;	
V(mutex) ;	
V(full) ;	//退出区

## 消费者

P(full) ;	
P(mutex) ;	//进入区
取出产品 ;	
V(mutex) ;	
V(empty) ;	//退出区



# 生产者—消费者问题

## 生产者

P(empty) ;
P(mutex) ;     //进入区
放入产品 ;
V(mutex) ;
V(full) ;        //退出区

## 消费者

P(full) ;
P(mutex) ;     //进入区
取出产品 ;
V(mutex) ;
V(empty) ;      //退出区

进程中 P 操作的次序可换吗？



## 生产者—消费者问题

P操作的顺序不当会产生死锁。例如假定执行顺序如下

Producer :

P(empty);

P(mutex);

one unit → buf;

V(mutex);

V(full);

Consumer :

P(mutex);

P(full); //进入区

buf → one unit;

V(mutex);

V(empty); //退出区



# 生产者—消费者问题

P操作的顺序不当会产生死锁。例如假定执行顺序如下

Producer :

P(empty);

P(mutex);

one unit → buf;

V(mutex);

V(full);

Consumer :

P(mutex);

P(full); //进入区

buf → one unit;

V(mutex);

V(empty); //退出区

分析：当full=0, mutex = 1时，执行顺序：

Consumer.P(mutex); Consumer.P(full);

// Consumer阻塞，等待Producer发出的full信号

Producer.P(empty); Producer.P(mutex);

// Producer阻塞，无法进入缓冲区



# 生产者—消费者问题

P操作的顺序不当会产生死锁。例如假定执行顺序如下

Producer :


```
P(empty);
P(mutex);
one unit → buf;
V(mutex);
V(full);
```

Consumer :

```
P(mutex);
P(full);      //进入区
buf → one unit;
V(mutex);
V(empty);     //退出区
```

分析：当full=0, mutex = 1时，执行顺序：

```
Consumer.P(mutex); Consumer.P(full);
```

// Consumer阻塞，等待Producer发出的full信号  发生死锁

```
Producer.P(empty); Producer.P(mutex);
```

// Producer阻塞，无法进入缓冲区

可以考虑采用 AND 信号量集：Swait(empty, mutex)



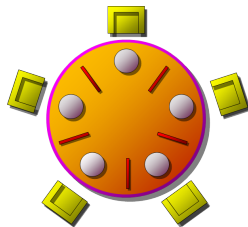
## 生产者—消费者问题

- 生产者—消费者问题是一个同步问题
  - 消费者想要取产品，有界缓冲区中至少有一个单元是满的。
  - 生产者想要放产品，有界缓冲区中至少有一个是空的。
- 它是一个互斥问题：有界缓冲区是临界资源，因此各生产者进程和各消费者进程必须互斥执行。
- 互斥信号量的 P,V 操作在每一程序中必须成对出现。资源信号量 (full,empty) 也必须成对出现，但可分别处于不同的程序中。
- 多个 P 操作顺序不能颠倒。先执行资源信号量的 P 操作，再执行互斥信号量的 P 操作，否则可能引起死锁。



## “哲学家进餐”问题

- 有五个哲学家，他们的生活方式是交替地进行思考和进餐。他们共用一张圆桌，分别坐在五张椅子上。
- 在圆桌上有五个碗和五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进餐完毕，放下筷子又继续思考。



哲学家进餐问题可看作是并发进程并发执行时处理共享资源的一个有代表性的问题。





## “哲学家进餐”问题

```
Var chopstick: array[0 , ... , 4] of semaphore=1 ;
```

```
/*5支筷子分别设置为初始值为1的互斥信号量*/
```

第*i*个哲学家的活动

```
repeat
    wait(chopstick[i]) ;
    wait(chopstick[(i+1) mod 5]) ;
    eat ;
    signal(chopstick[i]) ;
    signal(chopstick[(i+1) mod 5]) ;
    think ;
until false ;
```



## “哲学家进餐”问题

- 此算法可以保证不会有相邻的两位哲学家同时进餐。
- 若五位哲学家同时饥饿而各自拿起了左边的筷子，这使五个信号量 chopstick 均为 0，当他们试图去拿起右边的筷子时，都将因无筷子而无限期地等待下去，即可能会引起死锁。



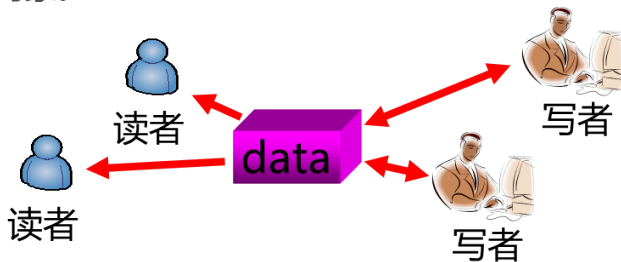
## 哲学家进餐问题的改进解法

- 方法一：至多只允许四位哲学家同时去拿左筷子，最终能保证至少有一位哲学家能进餐，并在用完后释放两只筷子供他人使用。
- 方法二：仅当哲学家的左右手筷子都拿起时才允许进餐。
- 方法三：规定奇数号哲学家先拿左筷子再拿右筷子，而偶数号哲学家相反。



## 读者—写者问题

- 问题描述：一个数据对象（数据文件或记录）可被多个进程共享。其中，reader 进程要求读，writer 进程要求写或修改。
- 允许多个 reader 进程同时读共享数据，但不允许一个 writer 进程与其它的 reader 进程或 writer 进程同时访问，即 writer 进程必须与其它进程互斥访问共享对象。



## 第一类：读者优先

- 当一个读者正在读数据时，另一个读者也需要读数据，应允许第二个读者进入，同理第三个及后续读者都应允许进入。
- 现在假设一个写者到来，由于写操作是排他的，所以它不能访问数据，需要阻塞等待。如果一直有新的读者陆续到来，写者的写操作将被严重推迟。
- 该方法称为“**读者优先**”，即一旦有读者正在读数据，只有当全部读者退出，才允许写者进入写数据。



## 第一类：读者优先

### ■ 新读者：

- 如果无读者、写者，新读者可以读。
- 如果有写者等待，但有其它读者正在读，则新读者也可以读。
- 如果有写者写，新读者等待。

### ■ 新写者：

- 如果无读者，新写者可以写。
- 如果有读者，新写者等待。
- 如果有其它写者，新写者等待。



## 第一类：读者优先

- 设置一个共享变量和两个互斥信号量。
- 共享变量 `Readcount`：记录当前正在读数据集的读进程数目，初值为 0。
- 读互斥信号量 `Rmutex`：表示读进程互斥地访问共享变量 `Readcount`，初值为 1。
- 写互斥信号量 `Wmutex`：表示写进程与其它进程（读、写）互斥地访问数据集，初值为 1。



# 第一类：读者优先

```

Var rmutex,wmutex: semaphore:=1,1 ; Readcount: integer:=0 ;
begin
  parbegin
    Reader: begin
      repeat
        wait(rmutex) ;    /*各位读者互斥访问readcount*/
        if readcount=0 then wait(wmutex) ; /*第一位读者阻止写者*/
        readcount:=readcount+1 ;
        signal(rmutex) ;
        ... perform read operation ; ...
        wait(rmutex) ;
        readcount:=readcount-1 ;
        if readcount=0 then signal(wmutex) ; /*末位读者允许写者*/
        signal(rmutex) ;
      until false ;
    end
  end

```

Readcount是记录有多少读者正在读，是临界资源。





## 第一类：读者优先

```
Writer: begin
        repeat
            wait(wmutex) ;
            ... perform write operation ... ;
            signal(wmutex) ;
        until false ;
    end
parend
end
```



## 第二类：写者优先

- 为了防止“读者优先”可能导致的写者饥饿，可以考虑写者优先。
- **写者优先**：当共享数据区被读者占用时，后续紧邻到达的读者可以继续进入，若这时有一个写者到来并阻塞等待，则写者后续到来的读者全部阻塞等待。
- 即只要有一个写者申请写数据，则不再允许新的读者进程进入读数据。这样，写者只需等待先于它到达的读者完成其读数据的任务，而不用等待其后到达的读者。



## 第二类：写者优先

- **写者优先**：一旦有写者到达，后续的读者必须等待，无论是否有读者在读。
- 增加一个信号量  $s$ ，初值为 1，用来实现读写互斥，使写者请求发生后的读者等待。

### 第二类：写者优先

- `int readcount = 0; /* 定义读者计数器 */`
- `semaphore rmutex = 1; /* 读者计数器互斥信号量 */`
- `semaphore wmutex = 1; /* 写互斥信号量 */`
- `semaphore  $s = 1$ ; /* 读写互斥信号量 */`



## 第二类：写者优先

### Reader()

```
{  
    P(s);           /*读写互斥*/  
    P(rmutex);  
    Readcount++;  
    if(readcount==1) P(wmutex);  
    V(rmutex);  
    V(s);  
    /*先释放s, 使其它读、写进程可以进入*/  
    read;  
    P(rmutex);  
    readcount--;  
    if(readcount==0) V(wmutex);  
    V(rmutex);  
};
```

### Writer()

```
{  
    P(s);  
    P(wmutex);  
    /*等待前面所有读者读完*/  
    write;  
    V(wmutex);  
    V(s);  
};
```



## 读者-写者问题的变形

例：进程 A、进程 B 共享文件 file，共享要求是：A、B 可同时读 file，或者同时写，但不允许一个在写 file，一个去读 file。即不允许读、写操作同时进行。



## 读者-写者问题的变形

例：进程 A、进程 B 共享文件 file，共享要求是：A、B 可同时读 file，或者同时写，但不允许一个在写 file，一个去读 file。即不允许读、写操作同时进行。

```
semaphore s=1 ; //读写互斥信号量  
int readcount=0 ; //读者数目计数器  
int writecount=0 ; //写者数目计数器  
semaphore rmutex=1 ; //读计数器信号量  
semaphore wmutex=1 ; //写计数器信号量
```



## 读者-写者问题的变形

```

读操作：reader( )
{
    P(rmutex) ;           //计数器互斥操作
    if(readcount==0) P(s) ; //第一个读者执行读写互斥
    readcount++ ;          //计数器加1
    V(rmutex) ;           //退出计数器操作
    read file ;           //读文件
    P(rmutex) ;           //读完修改计数器
    readcount-- ;          //计数器减1
    if(readcount==0) V(s) ; //最后一个读者释放读写互斥
    V(rmutex) ;           //退出计数器操作
}

```



## 读者-写者问题的变形

```
写操作：writer( )
{
    P(wmutex) ;           //计数器互斥操作
    If(writecount==0) P(s) ; //第一个写者应执行读写互斥
    writecount++ ;         //计数器加1
    V(wmutex) ;           //退出计数器操作
    write file ;           //写文件
    P(wmutex) ;           //写完成修改计数器
    writecount-- ;         //计数器减1
    if(writecount==0) V(s) ; //最后一个写者释放读写互斥
    V(wmutex);             //退出计数器操作
}
```





## 利用信号量集解决读者—写者问题

- 假设最多只允许  $RN$  个读者同时读。

### 利用信号量集解决读者—写者问题

- 引入一个信号量  $L$ ，并赋予其初值为  $RN$ 。
- 通过执行  $\text{Swait}(L, 1, 1)$  操作，来控制读者的数目。
- 每当有一个读者进入时，就要先执行  $\text{Swait}(L, 1, 1)$  操作，使  $L$  的值减 1。当有  $RN$  个读者进入读后， $L$  便减为 0，第  $RN+1$  个读者要进入读时，必然会因  $\text{Swait}(L, 1, 1)$  操作失败而阻塞。



# 利用信号量集解决读者—写者问题

```

Var RN integer ;
L, mutex: semaphore:=RN,1           //mutex是读写互斥
信号量
begin
  parbegin
    reader: begin
      repeat
        Swait(mutex,1,0) ; //可控开关
        Swait(L,1,1) ;
        perform read operation ;
        Ssignal(L,1) ;
      until false ;
    end
  end

```



# 利用信号量集解决读者—写者问题

读者优先

```

writer: begin
    repeat
        Swait( mutex, 1, 1 ; L, RN, 0 ) ;
        //AND型信号量，要么全分配，要么全不分配。
        //如果写者进程开始写，则后来的读者进程都无法进入。
        //只有L=RN，即还没有读者进入或读者全部离开时才允许写。
        perform write operation ;
        Ssignal(mutex,1) ;
    until false ;
end
parend
end
  
```



# 利用信号量集解决读者—写者问题

写者优先

```

writer: begin
    repeat
        Swait( mutex, 1, 1 );
        Swait(L, RN, 0 );
        //如果写者进程到来，则后来的读者进程都无法进入。
        //只有L=RN，即还没有读者进入或读者全部离开时才允许写。
        perform write operation ;
        Ssignal(mutex,1) ;
    until false ;
end
parend
end
  
```



## 利用信号量集解决读者—写者问题

- `Swait(mutex, 1, 0)` 语句起着开关的作用。只要无 writer 进程进入写，`mutex=1`，reader 进程就都可以进入读。但只要一旦有 writer 进程进入写时，其 `mutex=0`，则任何 reader 进程就都无法进入读。
- `Swait(mutex, 1, 1 ; L, RN, 0)` 语句表示仅当既无 writer 进程在写 (`mutex=1`)，又无 reader 进程在读 (`L=RN`) 时，writer 进程才能进入临界区写。



1 2.4 进程同步

2 2.5 经典进程同步问题

3 经典同步问题例题

4 管程机制

5 2.6 进程通信

6 本章作业



## 经典同步问题例题

例：若有一售票厅只能容纳 300 人，当少于 300 人时可以进入；否则，需在外等候。若将每一个购票者作为一个进程，请用 P、V 操作编程，并写出信号量的初值。



## 经典同步问题例题

例：若有一售票厅只能容纳 300 人，当少于 300 人时可以进入；否则，需在外等候。若将每一个购票者作为一个进程，请用 P、V 操作编程，并写出信号量的初值。

解：购票者进程  $P_i$  ( $i=1, 2, 3, \dots$ )。设信号量  $S$ ，初值  $=300$ 。

$P(S)$ ;

进入售票厅;

购票;

退出售票厅;

$V(S)$ ;





## 经典同步问题例题

例：有一阅览室，读者进入时必须先在一张表上进行登记。该表为每一座位列出一个表目（包括座号、姓名、阅览时间），读者离开时要撤消登记信息。阅览室有 100 个座位。

①为描述读者的动作，应编写几个程序，应设置几个进程？程序和进程之间的对应关系如何？

②试用 P、V 操作描述这些进程间的同步关系。



## 经典同步问题例题

- 在本题中，每个读者都可视为一个进程，有多少读者就有多少进程。这些进程称为读者进程，设为  $P_i$  ( $i=0, 1, \dots$ )。读者进程  $P_i$  执行的程序包括：登记、阅览和撤消。每个读者进程的活动都相同，所以其程序也相同。进程和程序的关系是各读者进程共用同一程序。
- 在读者进程所执行的程序中，对登记和撤消都需互斥执行，设一个信号量，其初值为 1，而对进入阅览室的读者人数也需要控制，设一个信号量，其初值为 100。



# 经典同步问题例题

信号量S1，的初值为100，  
代表座位数目；

信号量S2，初值为1，控  
制对登记表的互斥访问。

读者进程Pi (  $i=0,1,2,\dots$  )

P ( S1 )

P ( S2 )

登记

V ( S2 )

阅览

P ( S2 )

撤消登记

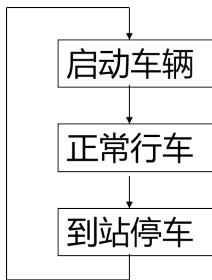
V ( S2 )

V ( S1 )

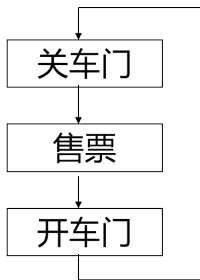


## 经典同步问题例题

例：设公共汽车上，司机和售票员的活动如下，在汽车不断地到站、停车、行驶过程中，这两个活动有什么同步关系？用信号量和 P、V 操作实现它们的同步。



司机的活动

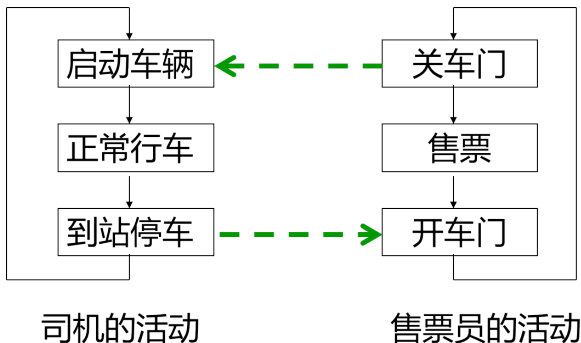


售票员的活动



## 经典同步问题例题

例：设公共汽车上，司机和售票员的活动如下，在汽车不断地到站、停车、行驶过程中，这两个活动有什么同步关系？用信号量和 P、V 操作实现它们的同步。



## 经典同步问题例题

本题中，应设置两个信号量：S1、S2。S1 表示是否允许司机启动汽车（由“售票员关门”来控制），其初值为 0；S2 表示是否允许售票员开门（由“司机停车”来控制），其初值为 0。



## 经典同步问题例题

本题中，应设置两个信号量：S1、S2。S1 表示是否允许司机启动汽车（由“售票员关门”来控制），其初值为 0；S2 表示是否允许售票员开门（由“司机停车”来控制），其初值为 0。

```
Semaphore S1=0;  
Semaphore S2=0;  
Main()  
{  
    parbegin  
        Driver();  
        Conductor();  
    parend  
}
```

```
Driver()  
{  
    P(S1);  
    启动车辆;  
    正常行车;  
    到站停车;  
    V(S2);  
}
```

```
Conductor()  
{  
    关车门;  
    V(S1);  
    售票;  
    P(S2);  
    开车门;  
    上下乘客;  
}
```



## 经典同步问题例题

例：有一只铁笼子，每次只能放入一只动物。猎手向笼中放入老虎，农民向笼中放入猪，动物园等待取笼中的老虎，饭店等待取笼中的猪，试用 P、V 操作写出能同步执行的程序。





## 经典同步问题例题

例：有一只铁笼子，每次只能放入一只动物。猎手向笼中放入老虎，农民向笼中放入猪，动物园等待取笼中的老虎，饭店等待取笼中的猪，试用 P、V 操作写出能同步执行的程序。

这个问题实际上可看作是两个生产者和两个消费者共享了一个仅能存放一件产品的缓冲器。生产者各自生产不同的产品，消费者各自取自己需要的产品。



# 经典同步问题例题

需设三个信号量，S代表笼子的使用情况，初始值为1；S1代表笼中是否是老虎，初值为0；S2代表笼中是否是猪，初值为0。

猎手进程

P ( S )

放入老虎

V ( S1 )

农民进程

P ( S )

放入猪

V ( S2 )

动物园进程

P ( S1 )

买老虎

V ( S )

饭店进程

P ( S2 )

买猪

V ( S )

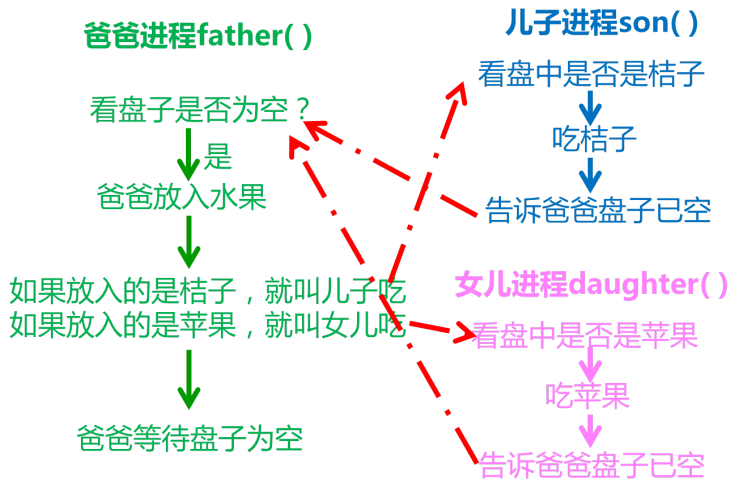


## 经典同步问题例题

例：桌上有一空盘，允许存放一个水果。爸爸可向盘中放苹果，也可向盘中放桔子。儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定盘空时一次只能放一只水果供吃者取用，请用 P、V 原语实现爸爸、儿子、女儿三个并发进程的同步。



# 经典同步问题例题



## 经典同步问题例题

- 本题实际上是生产者—消费者问题的一种变形。这里，生产者放入缓冲区的产品有两类（苹果和桔子），消费者也有两类（儿子和女儿），每类消费者只消费其中固定的一类产品。
- 此题应设三个信号量  $S$ 、 $S_o$ 、 $S_a$ 。
  - 信号量  $S$  表示盘子是否为空，初值为 1。
  - 信号量  $S_o$  表示盘中是否是桔子，初值为 0。
  - 信号量  $S_a$  表示盘中是否是苹果，初值为 0。



# 经典同步问题例题

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

```
father()
{
    P(S);
    将水果放入盘中 ;
    if (放入的是桔子) V(So) ;
    else V(Sa);
}
```



# 经典同步问题例题

son()

```
{  
    P(So);  
    从盘中取出桔子 ;  
    V(S);  
    吃桔子 ;  
}
```

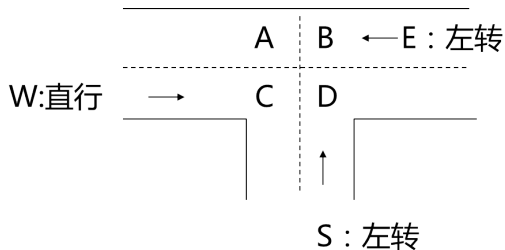
daughter()

```
{  
    P(Sa);  
    从盘中取出苹果 ;  
    V(S);  
    吃苹果 ;  
}
```



# 经典同步问题例题

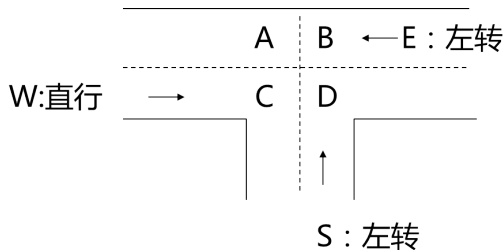
例：用 P、V 原语实现三辆汽车 S、E、W 过丁字路口的过程（不考虑红绿灯）。





# 经典同步问题例题

例：用 P、V 原语实现三辆汽车 S、E、W 过丁字路口的过程（不考虑红绿灯）。



解：设临界资源：A B C D

Var Sa, Sb, Sc, Sd: semaphore (1,1,1,1)



# 经典同步问题例题

Procedure S:

P(Sd);  
驶入D ;

P(Sb);  
驶入B;  
V(Sd);

P(Sa);  
驶入A ;  
V(Sb);

驶出A;  
V(Sa)

Procedure E:

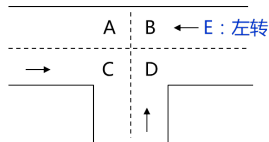
P(Sb);  
驶入B ;

P(Sa);  
驶入A;  
V(Sb);

P(Sc);  
驶入C ;  
V(Sa);

驶出C;  
V(Sc);

W:直行



Procedure W:

P(Sc);  
驶入C;

P(Sd);  
驶入D ;  
V(Sc);

驶出D ;  
V(Sd);

S: 左转

