

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

operating system

河南大学

# 操作系统

计算机学院

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

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

operating system



# 张 帆

## 教授



**13839965397**



**zhangfan@henu.edu.cn**



**计算机学院 412 房间**



## 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;



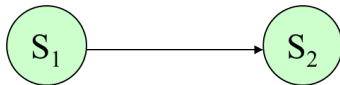
## 利用信号量实现进程互斥

- 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。

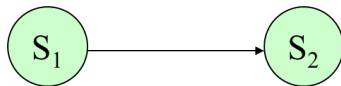


## 利用信号量实现进程同步

- 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

...

$P(S);$

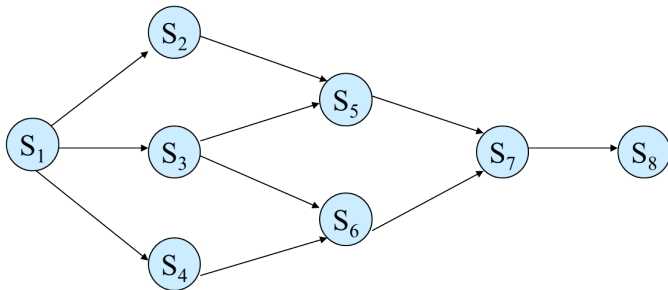
$S2;$

...



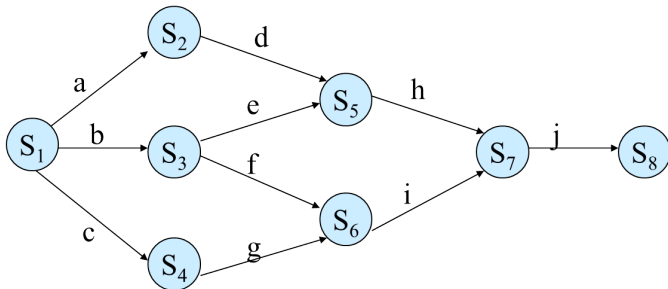
# 利用信号量实现进程同步

例：利用信号量来描述前趋图关系

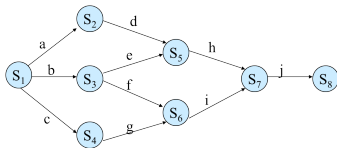


## 利用信号量实现进程同步

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



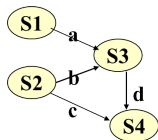
```

1  Struct smaphore a,b,c,d,e,f,g,h,i,j=0,0,0,0,0,0,0,0,0,0
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
  
```



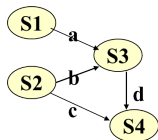
# 利用信号量实现进程同步

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);     P(b);     P(d);
      S3();   V(b);     S3;       S4;
      S4();   V(c);     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

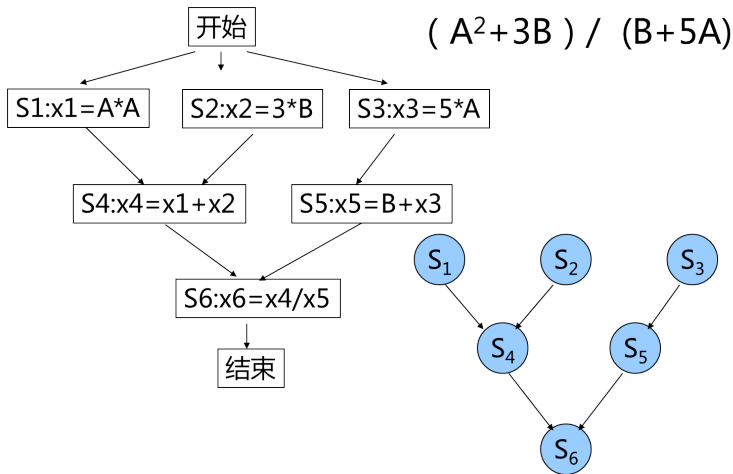
例：已知一个求值公式  $(A^2 + 3B)/(B + 5A)$ ，若 A,B 已赋值，利用信号量来描述该公式求值过程的前趋图。

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);}

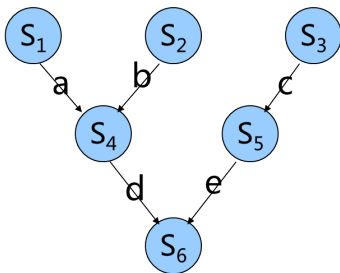
{P(a); P(b); S4;V(d);}

{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

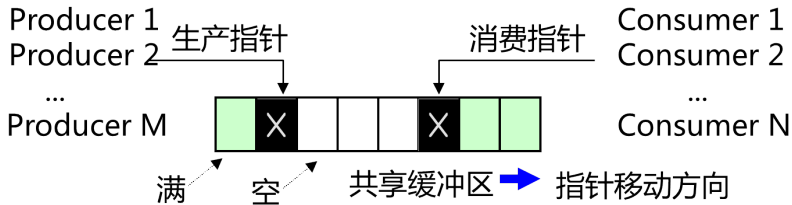


## 生产者—消费者问题

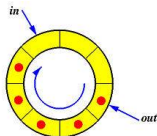
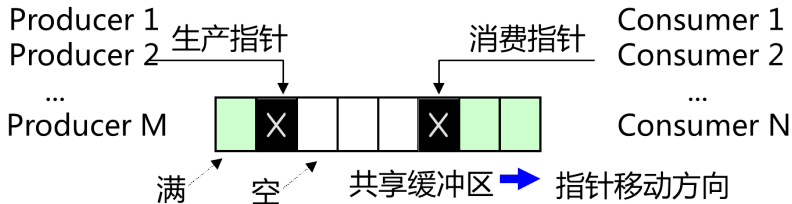
- “生产者—消费者”问题 (producer/consumer problem) 是最著名的进程同步问题。
- 它描述了一组生产者向一组消费者提供产品，它们共享一个**有界缓冲池**，生产者向其中投放产品，消费者从中取得产品。
- 它是许多相互合作进程的抽象，如输入进程与计算进程；计算进程与打印进程等。



# 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)



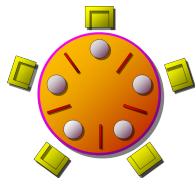
# 生产者—消费者问题

- 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。



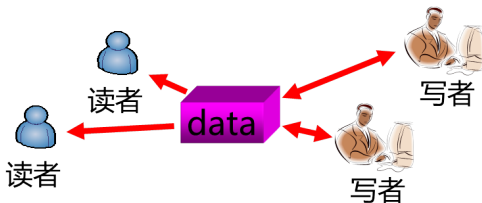
## 哲学家进餐问题的改进解法

- 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、写进程可以继续*/
    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。

解：

```
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
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，每个读者都可视作为一个进程，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

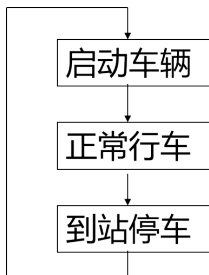
V ( S2 )

V ( S1 )

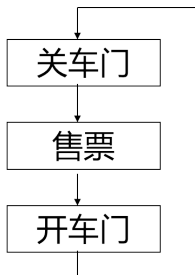


## 经典同步问题例题

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

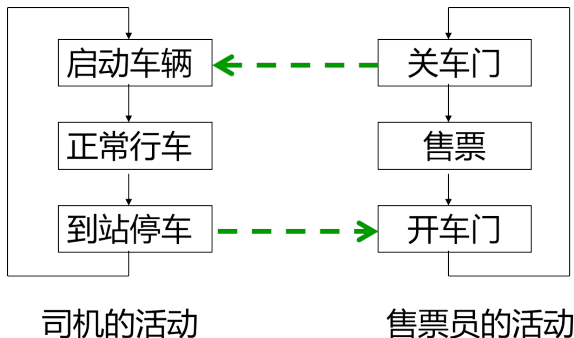


售票员的活动



## 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 )

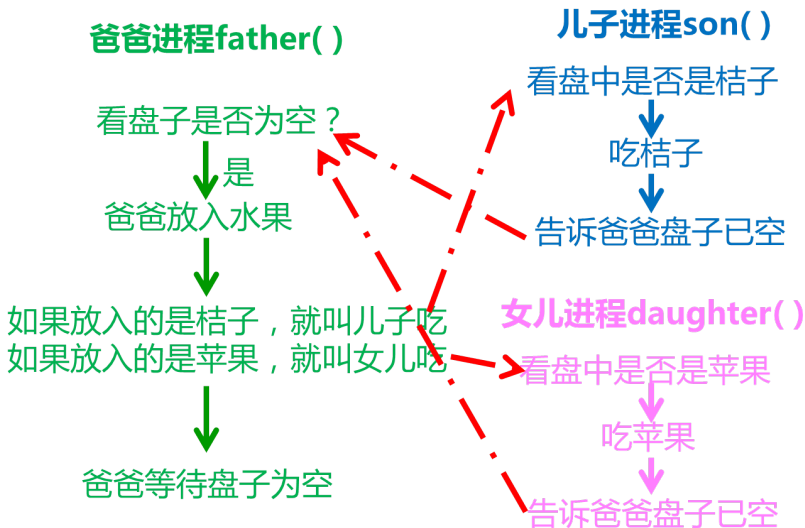


## 经典同步问题例题

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 ;  
}
```

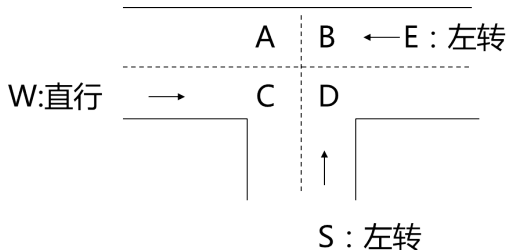
daughter()

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



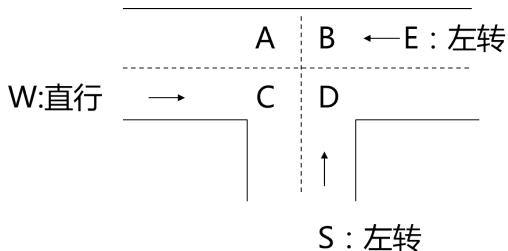
# 经典同步问题例题

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:

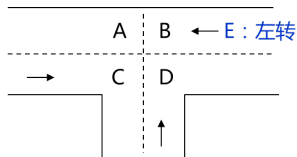
P(Sb);  
驶入B ;

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



## 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



# 管程机制

- 本为了避免凡要使用临界资源的进程都自备同步操作 wait(s) 和 signal(s)，将同步操作的机制和临界资源结合到一起，形成**管程**。
- **管程 (Monitors)**
  - 管程相当于围墙，把共享变量和对它进行操作的若干个过程围了起来，所有进程要访问临界资源时，都必须经过管程才能进入。
  - 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，也便于程序正确性验证。



# 管程的基本概念

- **基本思想**: 把访问某种临界资源的所有进程的同步操作都集中起来, 构成一个所谓的“秘书”进程(管程), 凡是访问临界资源的进程, 都需要报告“秘书”, 由秘书来实现诸进程的同步。
- **管程的定义**: 一个数据结构和在该数据结构上能被并发进程所执行的一组操作, 这组操作能使进程同步和改变管程中的数据。
- **管程的组成**: ①管程的名称; ②局部于管程内部的共享数据结构说明; ③对该数据结构进行操作的一组过程; ④对局部于管程内部的共享数据设置初始值的语句。



## 管程的基本概念

- 管程中的共享变量在管程外部是不可见的，不能被任何外部过程访问。管程相当于围墙，把共享变量和对它进行操作的若干过程围了起来。
- 外部只能通过调用管程中所说明的外部过程（函数）来间接地访问管程中的共享变量。
- 一个进程通过调用管程内的一个过程而进入管程。
- 管程通常是用来管理资源的，因而在管程中设有进程等待队列以及相应的等待及唤醒操作。
- 管程每次只允许一个进程执行，从而实现了进程的互斥。



# 管程的基本概念

```

TYPE <管程名> = MONITOR
  <管程变量说明>;
  define <( 能被其他模块引用的 ) 过程名列表>;
  use <( 要调用的本模块外定义的 ) 过程名列表>;
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
    end;
  .....
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
    end;
  begin
    <管程的局部数据初始化语句>;
  end;

```



# 管程的基本概念

- **互斥**：管程规定每次只准许一个进程执行，从而实现了进程互斥，保证了管程共享变量的数据完整性。
  - 任一时刻管程中只能有一个活跃进程，其它想进入管程的进程阻塞。
  - 进入管程的互斥由编译器负责，写管程的人无需关心。
- **同步**：通过条件变量及两个相关操作 wait 和 signal 实现。
  - 条件变量 x：表示等待原因。
  - x.wait：执行该操作的进程阻塞，直至被其它进程唤醒。
  - x.signal：唤醒 x 等待队列中的一个进程。如果 x 等待队列中无进程阻塞，该操作不产生任何作用。



# 管程的基本概念

- **条件变量**：在管程机制中，当某进程通过管程请求临界资源未能满足时，管程便调用 wait 原语使该进程等待，但等待的原因可能有多个，为了加以区别，在 P、V 操作前，引入条件变量加以说明。
- 条件变量的定义格式：Var x,y: condition
- 对条件变量执行的两种操作
  - Wait 操作如 x.wait 用来将执行进程挂到与条件变量 x 相应的等待队列上。
  - Signal 操作如 x.signal 用来唤醒与条件变量 x 相应的等待队列上的一个进程。





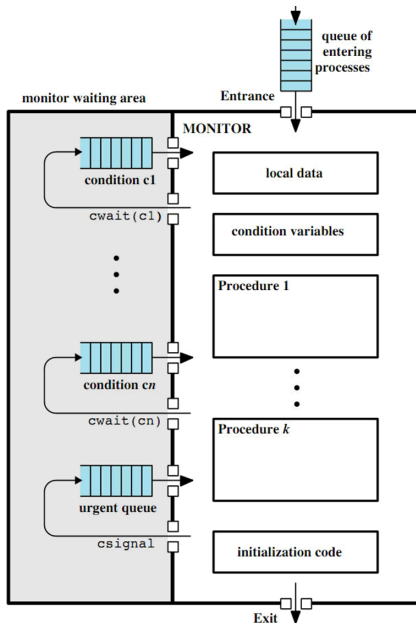
## 管程的基本概念

- 当一个进入管程的进程执行唤醒操作时（如进程 P 唤醒进程 Q），管程结构不允许被唤醒的进程重新执行（管程内只能有一个进程）。这种情况有两种可能的解决办法：
  - P 等待 Q 继续，直到 Q 等待或退出；（Hoare 采用）
  - Q 等待 P 继续，直到 P 等待或退出；（Lampson 和 Redell 采用）
- 折衷方案：Hansan 规定 signal 操作为过程体的最后一个操作，P 执行 signal 操作后马上退出管程，这样被唤醒的进程 Q 可以立即恢复执行。P 进入 Urgent 队列，下次优先进入管程。



## 管程示意图

Urgent队列上的进程比入口等待队列中的进程具有更高的优先级。



# 用管程机制解决生产者—消费者问题

## 1、建立Producer-Consumer(PC)管程

Type **PC=monitor**

```

    Var in,out,count:integer;
    buffer:array[0,...,n-1] of item;
    full, empty: condition;
    put(item);          /* 过程*/
    get(item);          /* 过程*/
begin                  /* 初始化代码*/
    in:=out:=0;
    count:=0;          /* 已放产品的缓冲区数目*/
end
  
```

管程中的两个条件变量：

(1) full 当缓冲区中全满，该变量为真。

(2) empty 当缓冲区中全空，该变量为真。



# 管程的基本概念

## ■ put(item) 过程

生产者利用此过程将自己的产品（消息）放到缓冲池中，若发现缓冲池已满 ( $\text{count} \geq n$ )，则等待。

## ■ get(item) 过程

消费者利用此过程将缓冲池中的产品取走，若发现缓冲池已空 ( $\text{count} \leq 0$ )，则等待。



# 用管程机制解决生产者—消费者问题

Procedure entry **put(item)**

begin

if  $\text{count} \geq n$  then **full.wait**; /\*缓冲池全满时等待\*/

$\text{buffer}(\text{in}) := \text{nextp}$ ;

$\text{in} := (\text{in} + 1) \bmod n$ ;

$\text{count} := \text{count} + 1$ ;

if **empty.queue** then **empty.signal**;

/\* 如果有empty.queue , 则唤醒一个因empty等待的进程\*/

end



# 用管程机制解决生产者—消费者问题

Procedure entry **get(item)**

begin

if  $\text{count} \leq 0$  then **empty.wait**; /\*缓冲池全空时等待\*/

nextc:=buffer(out);

out:=(out+1) mod n;

count:=count-1;

if full.queue then **full.signal**;

/\* 如果有full.queue , 则唤醒一个因full等待的进程\*/

end



# 用管程机制解决生产者—消费者问题

## 使用管程

```
Producer: begin
    repeat
        produce an item in nextp;
        PC.put(item);
    until false
end
Consumer: begin
    repeat
        PC.get(item);
        consume the item in nextc;
    until false
end
```

进程通过调用管程内的  
一个过程而进入管程



# 管程和进程区别

- 数据结构：进程定义的是私有数据结构 PCB，管程定义的是公共数据结构，如等待队列。
- 设置进程的目的在于实现系统的并发性，而管程的设置则是解决共享资源的互斥使用问题。
- 工作方式：管程为被动工作方式，进程则为主动工作方式。
- 进程能并发执行，管程则不能与其调用者并发。
- 进程具有动态性，由“创建”而诞生，由“撤销”而消亡，而管程则是操作系统中的一个资源管理模块，供进程调用。





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



# 进程通信

- **进程通信**：进程间的信息交换。
- 低级通信
  - 实例：信号量机制
  - 效率低，交换信息量少
  - 通信对用户不透明
- 高级通信
  - 效率高
  - 通信实现细节对用户透明



# 进程通信的类型

## 高级进程通信机制

- 1 共享存储器系统 (Shared Memory System)
- 2 消息传递系统 (Message Passing System)
  - 直接通信方式：消息缓冲通信
  - 间接通信方式：又称为信箱通信方式
- 3 管道 (Pipe) 通信：又称为共享文件通信
- 4 客户机 - 服务器系统 (Client-Server System)



# 共享存储器系统

## 1 基于共享数据结构的通信方式

- 系统提供了一种共享数据结构，适用于少量通信。
- 低效，不透明（数据结构的设置以及对进程间同步的处理，都必须由程序员来处理）。
- 例如 producer-consumer 中的缓冲区

## 2 基于共享存储区的通信方式

- 系统提供共享存储区。
- 通信过程：
  - ①向系统申请一个或多个分区。
  - ②对获得的分区进行读写。
- 特点：高效，速度快。



# 消息传递系统

- 消息传递系统 (Message Passing System)
  - 直接通信方式：消息缓冲通信
  - 间接通信方式：又称为信箱通信方式
- 信息单位：消息（报文）
- 实现：一组通信命令（原语），具有透明性

msgsender	消息发送者
msgreceiver	消息接收者
msgnext	下一个消息的链指针
msgsize	整个消息的字节数

**消息头**

msgtext	消息正文
---------	------

**消息正文**



## 直接通信方式

### ■ 原语

send ( Receiver, message ) 如接收者阻塞, 则唤醒。

receive ( Sender, message ) 无消息则阻塞。

### ■ 用直接通信方式解决生产者-消费者问题:

```
repeat
....
Produce an item in
nextp;
....
send(consumer,nextp);
until false;
```

```
repeat
....
Receive(producer,nextc);
....
consume the item in nextc;
until false
```



# 发送原语 send

## 发送原语

```

procedure send(receiver, a)
begin
    getbuf(a.size, i);           //根据a.size申请缓存区
    i.sender:=a.sender;         //将发送区a信息复制到消息缓存区i中
    i.size:=a.size;
    i.text:=a.text;
    i.next:=0;
    getid(PCB set, receiver.j); //取接收进程内部标识
    wait(j.mutex);              //消息队列互斥信号量，mutex初值为1
    insert(j.mq, i);            //将缓存区插入消息队列mq
    signal(j.mutex);
    signal(j.sm);               //消息队列资源信号量，sm初值为0
end

```



## 接收原语 receive

### 接收原语

```
procedure receive(b)
begin
    j:=internal name;    //j接受进程内部标识符
    wait(j.sm);           //sm消息队列资源信号量
    wait(j.mutex);        //mutex消息队列互斥信号量
    remove(j.mq, i);      //mq消息队列指针
    signal(j.mutex);
    b.sender:=i.sender;   //将消息缓存区i信息复制到接收区b中
    b.size:=i.size;
    b.text:=i.text;
end
```





## 间接通信方式

- 写进程  $\Rightarrow$  信箱（中间实体）  $\Rightarrow$  读进程
- 原语  
    Send(mailbox,message)  
    Receive(mailbox,message)
- 信箱的分类：私用信箱、公用信箱、共享信箱
- 发送—接收进程之间的关系：
  - 一对一关系
  - 多对一关系
  - 一对多关系
  - 多对多关系：公用信箱。



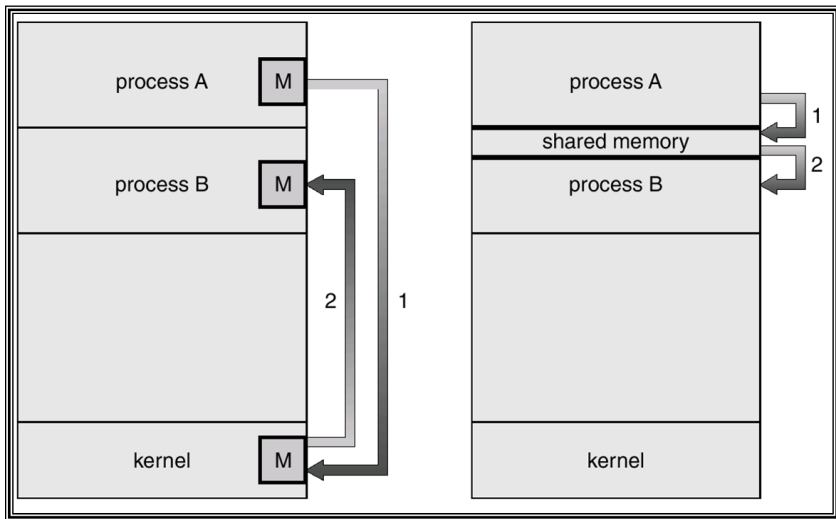
## 间接通信方式

### ■ 用间接通信方式解决生产者-消费者问题：

semaphore full=0;	/*满缓冲区数量*/
semaphore empty=N;	/*空缓冲区数量*/
send(mailbox, message)	receive(mailbox, message)
{	{
P(empty);	P(full);
选择空缓冲区E;	选择满缓冲区F;
将消息msg放入E中;	把F中的消息取出放msg中;
置E的标志为满 ( $E \rightarrow F$ );	置F标志为空 ( $F \rightarrow E$ );
V(full);	V(empty);
}	}



# 消息传递与共享内存



# 管道通信

- **管道**: 连接读进程和写进程间通信的**共享文件**。
- 功能: 大量的数据发收。
- 管道通信必须提供三方面的协调能力:
  - 1 共确定对方是否存在。
  - 2 互斥: 互斥访问管道。
  - 3 同步: 写满或读空管道后, 进程睡眠等待, 直到其它进程唤醒。



# UNIX 管道通信

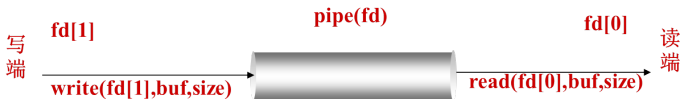
- 利用系统调用 `pipe()`，建立同步通信管道。

```
int pipe (int fd[2]);
```

//函数参数为一个长度为2的整型数组，是两个文件描述符。

//`fd[1]`为写入端，`fd[0]`为读出端。

//返回值，0为调用成功，-1为调用失败。



# UNIX 管道通信

- pipe 是进程间共享的。由系统调用生成的两个文件符也是进程间共享的。其中规定 `fd[0]` 文件符只能进行读操作，`fd[1]` 文件符只能进行写操作。
- 发送进程利用系统调用 `write(fd[1],buf,size)`，把 `buf` 中长度为 `size` 字符的消息送入管道入口 `fd[1]`。
- 接收进程则使用系统调用 `read(fd[0],buf,size)` 从管道出口 `fd[0]` 读出 `size` 字符的消息置入 `buf` 中。
- 管道按 FIFO 方式传送消息，且只能单向传送消息。



# UNIX 管道通信

例：用 C 语言编写一个程序，建立一个 pipe，同时父进程生成一个子进程，子进程向 pipe 中写入一字符串，父进程从 pipe 中读出该字符串。

本题结合了进程控制与进程通信。



```
main()
{   int x,fd[2];
    char buf[30],s[30];
    pipe(fd);                      //创建管道
    while ((x=fork())!=-1);        //创建子进程失败时，循环
    if (x==0)                      //在子进程中
    {
        sprintf(buf," this is an example\n" );
        write(fd[1],buf,30);      //把buf中字符写入管道
        exit(0);
    }
    else                          //在父进程中
    {
        wait(0);                 //等待子进程结束
        read(fd[0],s,30);        //父进程读管道中字符
        printf( "%s" ,s);
    }
}
```





```
main()
{
    int x,fd[2];
    char buf[30],s[30];
    pipe(fd);                                //创建管道
    while ((x=fork())!=-1);                  //创建子进程失败时，循环
    if (x==0)                                //在子进程中
    {
        sprintf(buf," this is an example\n" );
        write(fd[1],buf,30);                //把buf中字符写入管道
        exit(0);
    }
    else                                     //在父进程中
    {
        wait(0);                            //等待子进程结束
        read(fd[0],s,30);                   //父进程读管道中字符
        printf( "%s" ,s);
    }
}
```

this is an example



# 客户机 - 服务器系统通信机制

- 在网络环境下，客户机 - 服务器系统的通信机制是目前主流的通信实现机制。

## 客户机 - 服务器系统通信机制实现方式

- 1 套接字
- 2 远程过程调用
- 3 远程方法调用



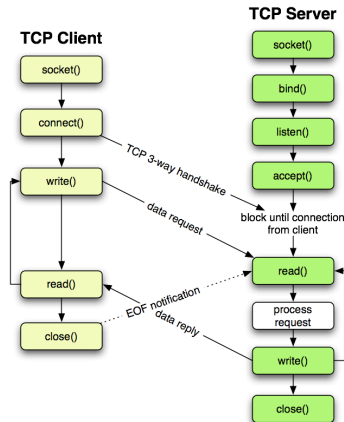
# 套接字

- **套接字** (Socket) 是对网络中进程之间进行双向通信的端点的抽象。
- 一个套接字就是网络中进程通信的一端。
- 套接字是一个通信标识类型的数据结构，包含许多选项，由操作系统内核管理。
- 套接字可以通过网络应用程序的编程接口 API 实现。
- 套接字的优点：
  - 不仅适用于一台计算机内部的通信，也适用于网络通信。
  - 确保了通信链路的唯一性，有利于数据的并发传输。
  - 隐藏了通信的细节。



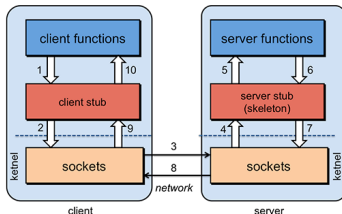
# 套接字

- 套接字是为客户机 - 服务器模型而设计的，通常套接字包括两类。
- **基于文件型**。通信进程运行于同一台机器中，套接字关联到一个特殊文件，通信双方通过读写这个文件进行通信，类似于管道。
- **基于网络型**。通信双方运行于不同主机的网络环境下。



# 远程过程调用和远程方法调用

- **远程过程调用**：(Remote Procedure Call, RPC) 是一个计算机通信协议。允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。如果涉及的软件采用面向对象编程，那么远程过程调用亦可称作**远程方法调用**。
- 为了能以相同的方式完成本地过程调用和远程过程调用，在客户/服务器上各增设一个客户存根(stub)和服务器存根。



## 远程过程调用和远程方法调用

- 当客户机上的某进程需要调用服务器上的一个过程时，发出一条 RPC 命令给客户存根 (stub)，委托它作为自己的代理。
- 客户存根收到 RPC 命令后，执行 send 原语请求内核把消息发送到服务器去，在发送消息执行后，客户存根调用 receive 原语阻塞自己直到服务器发来应答。
- 消息到达服务器后，内核把消息传送给服务器存根。
- 服务器存根拆包从消息中取出参数，然后以一般方式调用服务器进程，该进程执行相应的过程调用并返回结果。
- 过程调用完毕后，服务器存根调用 send 原语请求内核把消息发回给调用者，服务器存根回到 receive 状态，等待下一条消息。
- 内核把消息送给客户存根，客户存根检查并拆开消息包，把取出的结果返回给调用进程。



## 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

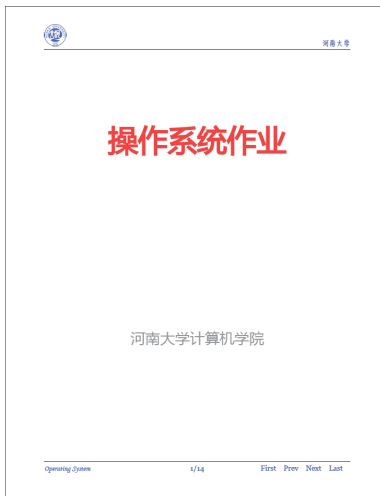
## 4 管程机制

## 5 2.6 进程通信

## 6 本章作业



# 本章作业



作业





# THE END

*School of Computer & Information Engineering*

*Henan University*

*Kaifeng, Henan Province*

*475001*

*China*

