



经典教材《计算机操作系统》**最新版**

# 第4章 进程同步

主讲教师：王申





# 第4章知识导图

第1章 操作系统引论

第2章 进程的描述与控制

第3章 处理机调度与死锁

**第4章 进程同步**

第5章 存储器管理

第6章 虚拟存储器

第7章 输入/输出系统

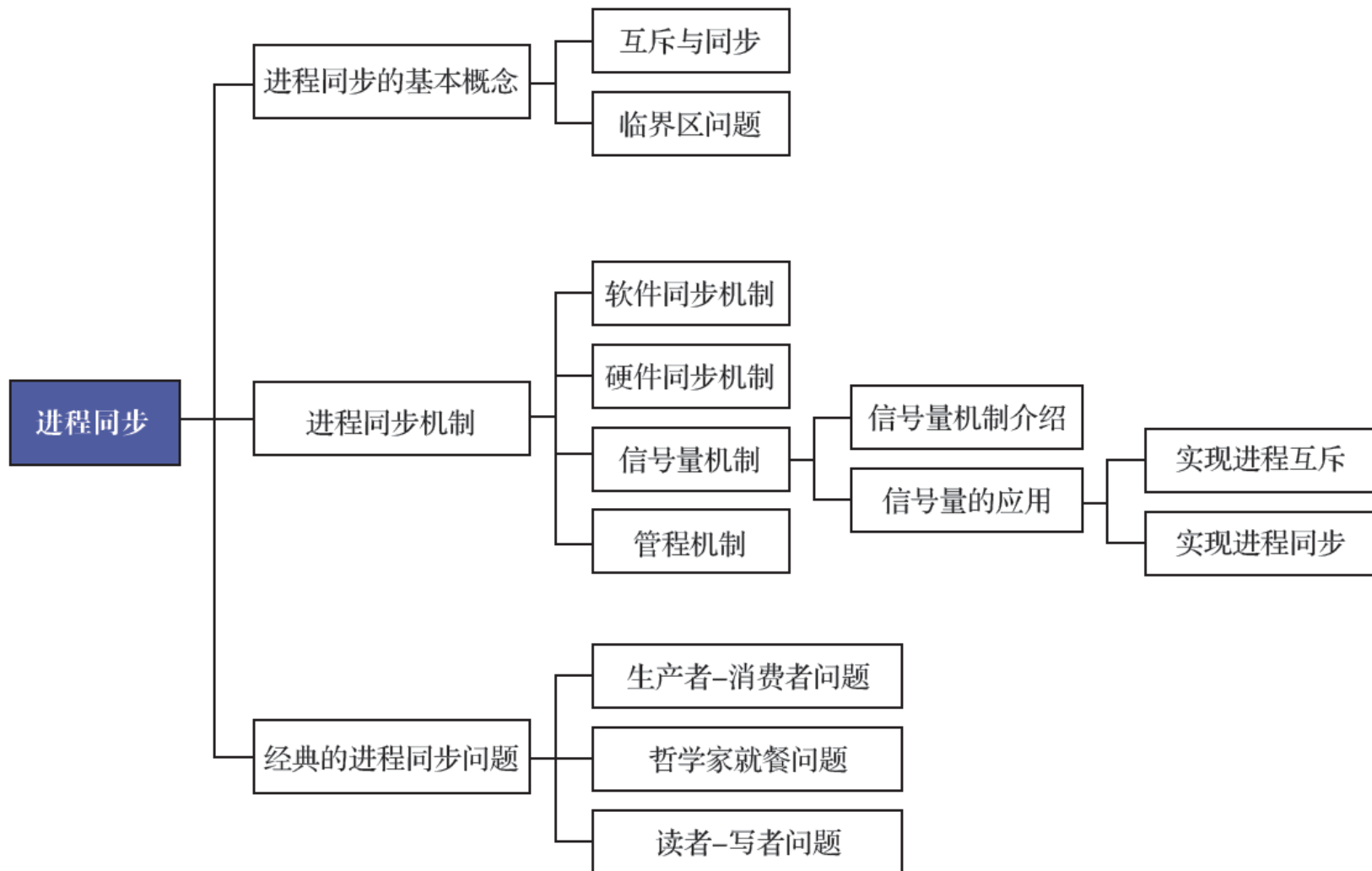
第8章 文件管理

第9章 磁盘存储器管理

第10章 多处理机操作系统

第11章 虚拟化和云计算

第12章 保护和安全





-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  4.7 Linux进程同步机制

## 第4章 进程同步

---



## 主要任务

- 使并发执行的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。



## 进程间的制约关系（竞争&&合作）

- 间接相互制约关系(互斥关系)
  - 进程互斥使用临界资源
- 直接相互制约关系（同步关系）
  - 进程间相互合作

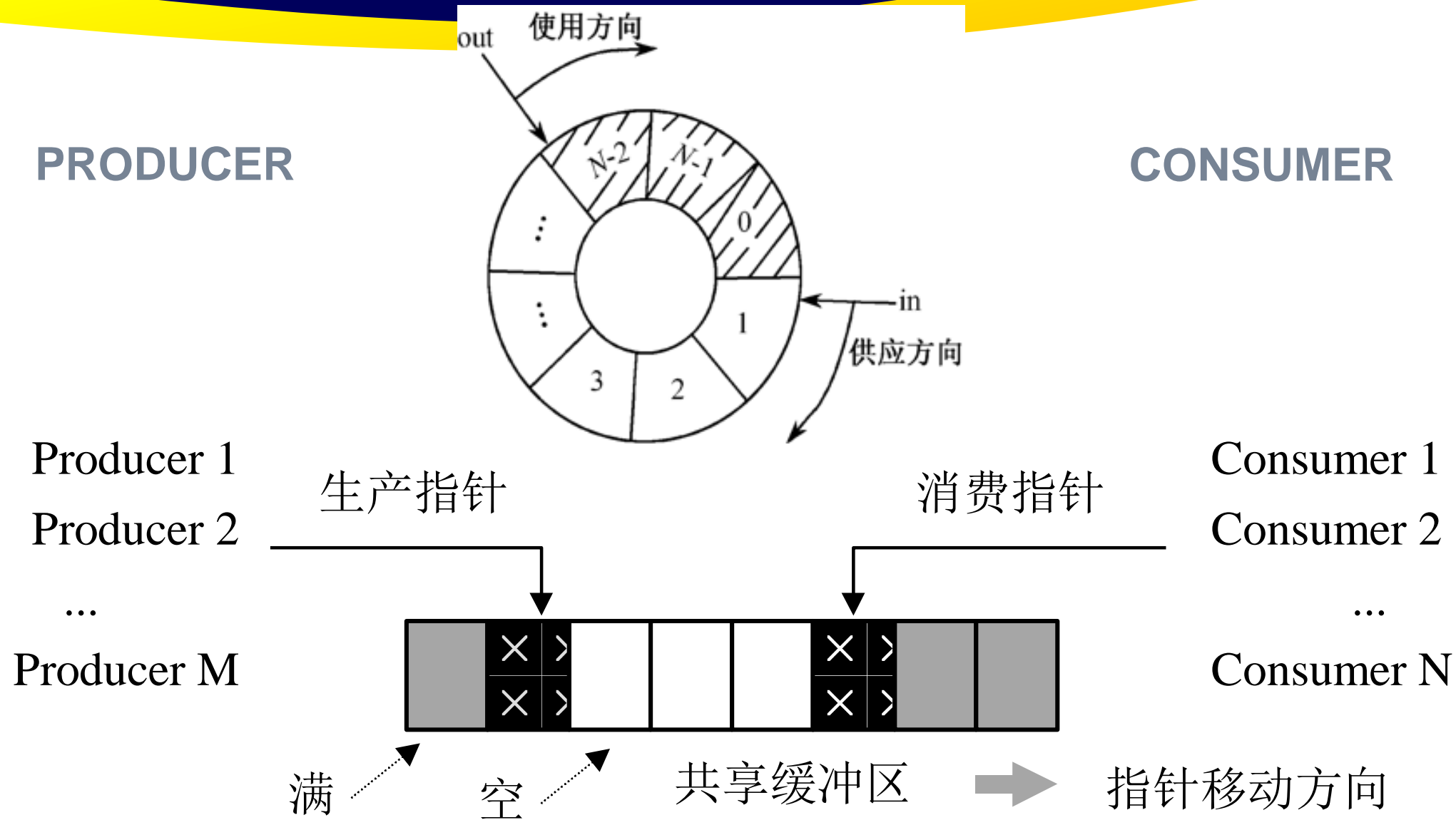


## 临界资源

- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量。
- 诸进程间应采取互斥方式，实现对这种资源的共享。

- 进程并行运行时相互制约 (进程发消息限制另一进程运行), 而**制约关系**归结为**互斥**和**同步**
- **同步**——指两个事件的发生有着**某种时序上的关系**(先, 后)
- **互斥**——资源的使用要**排它使用**, **防止竞争冲突**(不同时使用, 但无先后次序) (作为一种特殊同步)

# OS 数据不一致例子：生产者-消费者问题



**缓冲池buffer:**  
用数组来表示具有n个缓冲区的缓冲池

**输入指针in:**  
指示下一个可投放产品的缓冲区，每当**生产者进程**生产并投放一个产品后，输入指针加1，初值为0



**输出指针out:**  
指示下一个可获取产品的缓冲区，每当**消费者进程**取走一个产品后，输出指针加1，初值为0

**整型变量count:**  
初值为0，表示缓冲区中的产品个数



## 生产者进程: producer()

```
void producer( ) {  
    while(1){  
        produce an item in nextp;  
        ...  
        while (count == n)  
            ; // do nothing  
        // add an item to the buffer  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        count++;  
    }  
}
```





## 消费者进程: consumer ()

```
void consumer() {  
    while(1){  
        while (count == 0)  
            ; // do nothing  
        // remove an item from the buffer  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        count --;  
        consumer the item in nextc;  
        ...  
    }  
}
```

## ■ count++:

- R1=count; 1
- R1=R1+1; 2
- count=R1; 3

## ■ count--:

- R2=count; 4
- R2=R2-1; 5
- count=R2; 6

Count初值为4,并发执行produce()和consume()进程

执行次序	结果	是否正确
123456	4	是
142536	3	否
145263	5	否

**解决方法:** 下列语句必须被原子性地执行

counter++;

counter--;

- 临界区：进程中涉及临界资源的代码段
- 进入区：用于检查是否可以进入临界区的代码段
- 退出区：将临界区正被访问的标志恢复为未被访问标志
- 剩余区：其他代码

一个访问临界资源的循环进程的描述

While(TRUE){

进入区

临界区

退出区

剩余区

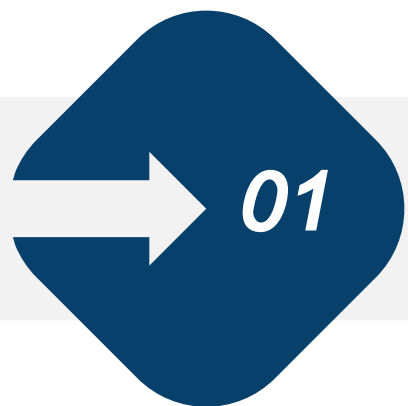
}

临界资源

```
item nextConsumed;
while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
```

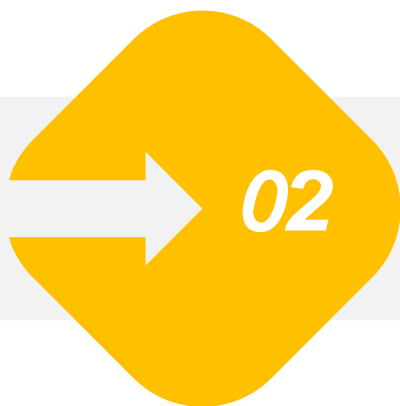
临界区

## 空闲让进



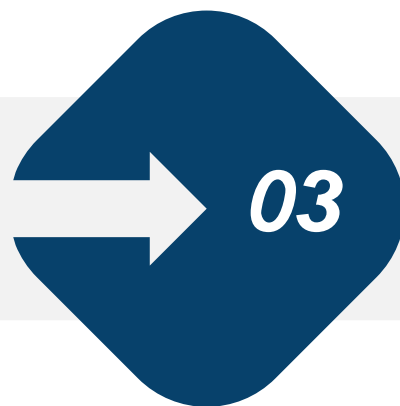
- 当无进程处于临界区，应允许一个请求进入临界区的进程立即进入自己的临界区；

## 忙则等待



- 已有进程处于其临界区，其它试图进入临界区的进程必须等待；

## 有限等待



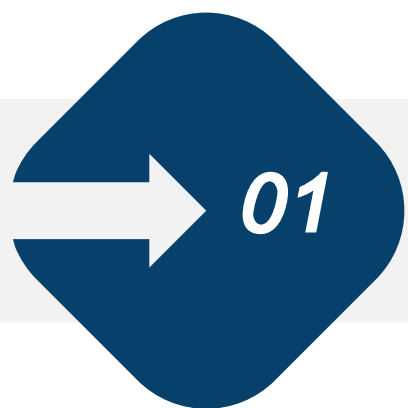
- 等待进入临界区的进程不能“死等”；

## 让权等待



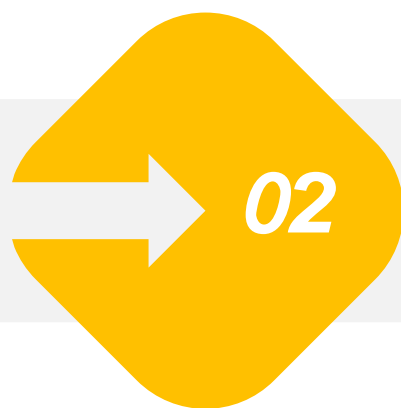
- 不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

## 软件同步机制



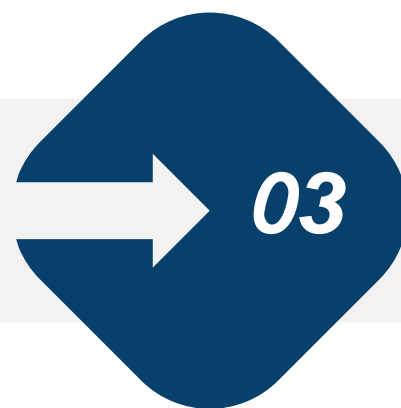
- 使用编程方法解决临界区问题
- 有难度、具有局限性，现在很少采用

## 硬件同步机制



- 使用特殊的硬件指令，可有效实现进程互斥

## 信号量机制



- 一种有效的进程同步机制，已被广泛应用

## 管程机制



- 新的进程同步机制



## 内容导航:



4.1 进程同步的概念



**4.2 软件同步机制**



4.3 硬件同步机制



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题



4.7 Linux进程同步机制

## 第4章 进程同步

---

**共享变量:** `int turn = i;` //代表当前允许进入临界区的进程

### 进程 $P_i$

```
do {  
    while (turn != i);  
    临界区  
    turn = j;  
    剩余区  
} while (1);
```

### 进程 $P_j$

```
do {  
    while (turn != j);  
    临界区  
    turn = i;  
    剩余区  
} while (1);
```

- `turn`初始值为 `i`，即只允许 $P_i$ 进入临界区；若 $P_j$ 先运行，则会一直忙等直到调度发生，切换到 $P_i$ 进程； $P_i$ 可正常访问临界区，访问完后，才允许 $P_j$ 访问。
- 可实现同一时刻最多只允许一个进程访问临界区，且是轮流访问的，缺陷：违背空闲让进原则

**共享变量:** `boolean flag[2];`    `flag [0] = flag [1] = false;` //代表有意愿进入临界区

## 进程 $P_i$

```
do {  
    flag [i] := true;  
    while (flag [j]);  
    临界区  
    flag [i] = false;  
    剩余区  
} while (1);
```

## 进程 $P_j$

```
do {  
    flag [j] := true;  
    while (flag [i]);  
    临界区  
    flag [j] = false;  
    剩余区  
} while (1);
```

- `while(flag [j])`可以理解为 $P_i$ 进临界区之前，先咨询 $P_j$ 是否想要进入它的临界区，如果 $P_j$ 想进的话 $P_i$ 就等待（ $P_i$ 品德高尚）；类似的， $P_j$ 也有类似行为。
- 双方互相谦让的结果是，最终两个进程谁也进不了临界区（饥饿）
- 缺陷：违背了空闲让进和有限等待原则



**共享变量:** `boolean flag[2];`    `flag [0] = flag [1] = false;` //代表是否有意愿进入临界区  
`int turn = 0;` //代表优先让哪个进程进入临界区

### 进程 $P_i$

```
do {  flag [i]:= true;
      turn = j;
      while (flag [j] && turn == j);
      临界区
      flag [i] = false;
      剩余区
    } while (1);
```

### 进程 $P_j$

```
do {  flag [j]:= true;
      turn = i;
      while (flag [i] && turn == i);
      临界区
      flag [j] = false;
      剩余区
    } while (1);
```

解决了两个进程的临界区问题    空闲进入、忙则等待、有限等待、让权等待



## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



**4.3 硬件同步机制**



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题



4.7 Linux进程同步机制

## 第4章 进程同步

---



关中断

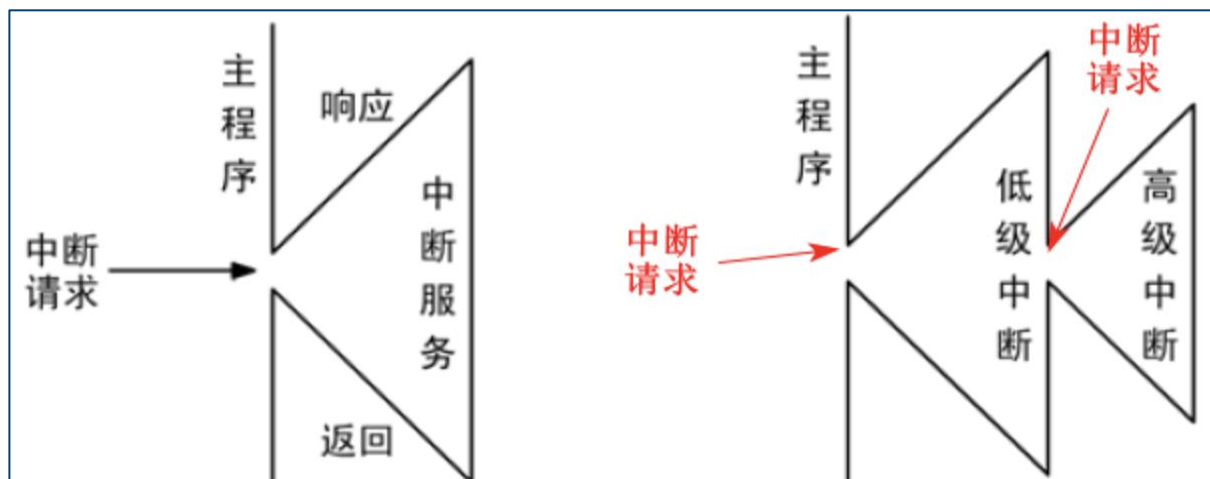


Test-and-Set指令



Swap指令

## 中断回顾



**中断：** CPU中止正在执行的程序，转去处理随机提出的请求，待处理完后，再回到原先被打断的程序继续恢复执行

操作系统在启动时都会注册一个时钟中断，用来实现时间片分片。处理时钟中断的中断处理程序会“引导”处理器进入内核态（包括切换内核栈、调度进程等等）



## 关中断实现互斥

- 进入锁测试**之前关闭中断，完成锁测试并上锁**之后才打开中断
- 中断关掉后→**无进程or线程切换**→**保证对锁测试和关锁操作的完整性**
- **可有效保证互斥**，但存在许多缺点，如：
  1. 影响系统效率，导致系统反应“迟钝”
  2. 不适用于多CPU（关CPU A上的中断，CPU B上进程仍然可以执行相同的临界区代码）

➤ 原子地检查和修改字的内容

```
boolean TS(boolean *lock)
{
    boolean old = *lock;
    *lock = TRUE;
    return old;
}
```

很多计算机都提供该指令，可以看成是一个函数，执行过程不可分割，即原语

**lock 为FALSE，资源空闲**

**lock 为TRUE，资源正在被使用**

➤ 共享数据:

```
boolean lock = FALSE;
```

➤ 进程  $P_i$

```
do {
    while (TS(&lock)) ;
    critical section
    lock = FALSE;
    remainder section
} while(TRUE)
```

- 仓库有1把锁，2把钥匙
- 看门大爷**无法与人流畅交流**
- 只会将手里的物品与来人的物品进行**交换**



### Case A:

1. **仓库里没有人**，你带着一把**钥匙**想访问仓库，此时大爷手里拿着的是**锁**
2. 开锁，大爷把手里的**锁**和你的**钥匙****交换**
3. 你进入仓库**反锁**，执行**临界区代码**
4. 执行完后，把手里的**锁**和大爷的**钥匙****交换**回来



- 仓库有1把锁，2把钥匙
- 看门大爷**无法与人流畅交流**
- 只会将手里的物品与来人的物品进行**交换**



### Case B:

1. **仓库里有人**，你带着一把**钥匙**想访问仓库，此时大爷手里拿着的是**钥匙**
2. 你不停的将手里的**钥匙**和大爷手里的**钥匙**进行**交换**（无用功，直到里面的人出来，才可以访问仓库）





➤ 原子地交换两个变量

```
void swap(boolean *a, boolean *b) {  
    boolean temp;  
    temp= *a;  
    *a = *b;  
    *b = temp;  
}
```

➤ 共享数据 (初始化为 FALSE):

```
boolean lock;
```

➤ 进程 Pi

```
do {  
    key = TRUE;  
    do {  
        swap(&lock, &key);  
    }while (key != FALSE)  
    临界区  
    swap(&lock, &key)或lock = FALSE;  
    剩余区  
}while(TRUE)
```

此时key = FALSE;



- 利用上述硬件可以有效地实现进程互斥
- 但是，当临界资源被访问时，其他访问进程必须不断进行测试（忙等待状态）
- 不符合“让权等待”原则，浪费CPU时间
- 难以应用并解决复杂进程的同步问题



## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



**4.4 信号量机制**



4.5 管程机制



4.6 经典进程的同步问题



4.7 Linux进程同步机制

## 第4章 进程同步

---

1965年，由荷兰学者迪科斯彻Dijkstra提出（P、V分别代表荷兰语的Proberen (test)和Verhogen (increment))，是一种卓有成效的进程同步机制。

### 信号量-软件解决方案

- 保证两个或多个代码段不被并发调用
- 在进入关键代码段前，进程必须获取一个信号量，否则不能运行
- 执行完该关键代码段，必须释放信号量
- 信号量有值，为正说明它空闲，为负说明其忙碌

### 类型

- 整型信号量
- 记录型信号量
- AND型信号量
- 信号量集



艾兹格·W·迪科斯彻(Edsger Wybe Dijkstra)

- 信号量和PV原语发明者
- 解决了“哲学家就餐”问题
- 最短路径算法(SPF)和**银行家算法**的创造者
- 结构程序设计之父
- **THE操作系统**设计者和开发者



与高德纳 (Donald E. Knuth) 并称为这个时代最伟大的计算机科学家 (《计算机程序设计的艺术》→排版系统TEX→\$3125.36 )



- 停车场有3个车位
- 且有一个自动管理系统可以统计车辆资源

### Case A:

1. 有停车位 ( $S > 0$ ) , 你可以直接开车进去

### Case B:

1. 无停车位 ( $S \leq 0$ ) , 对不起, 你只能等
2. 等到其他车开走, 让出空位

车位是临界资源, 每辆车好比一个进程, 统计的空闲车位数就是信号量



- 信号量**S**-整型变量
- 提供两个不可分割的[原子操作]访问信号量
  - wait(S): //wait 原语, 相当于进入区*  
while  $s \leq 0$  ; //无车位就一直等  
s:=s-1; //有车位, 直接占用
  - signal(S): //singal原语, 相当于退出区*  
s:=s+1; //腾退车位,
- wait(S)又称为P(S)
- signal(S)又称为V(S)
- **缺点**: 进程忙等 (没有遵循“让权等待”准则)

P0 进程:

...

wait(S);

使用打印机资源

signal(S)

...

P1 进程:

...

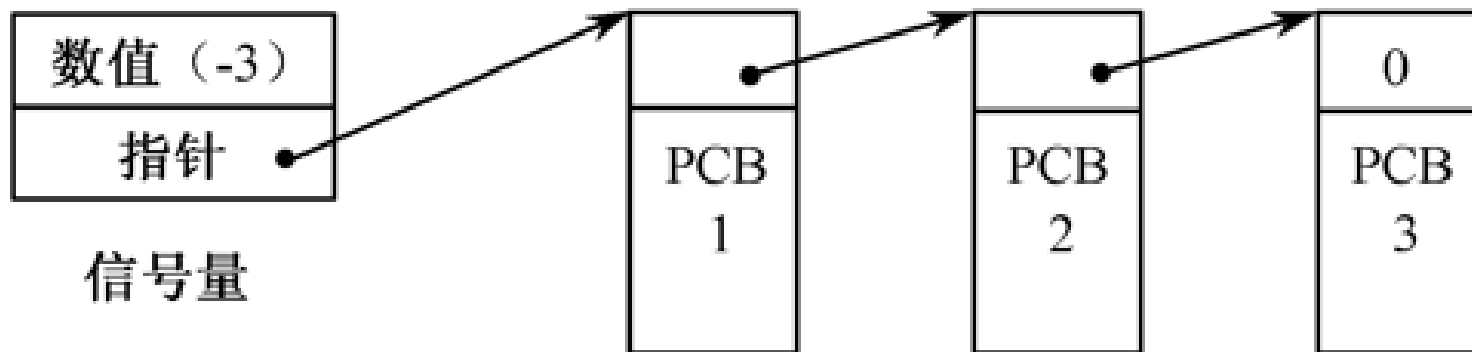
wait(S);

使用打印机资源

signal(S)

...

每个信号量S除一个**整数值**S.value外，还有一个**进程等待队列**S.list，存放阻塞在该信号量的各个进程PCB



```
typedef struct {
    int value;
    struct process_control_block *list;
} semaphore;
```



每个信号量S除一个**整数值**S.value外, 还有一个**进程等待队列**S.list, 存放阻塞在该信号量的各个进程PCB

- 信号量只能通过**初始化**和**两个标准的原语PV**来访问 - - 作为OS核心代码执行, 不受进程调度的打断
- **初始化**指定一个非负整数值, 表示**空闲资源总数** (又称为"资源信号量") -
  - 若为非负值表示**当前的空闲资源数**, 若为负值其绝对值表示**当前等待临界区的进程数**

```
typedef struct {  
    int value;  
    struct process_control_block *list;  
} semaphore;
```

```
wait(semaphores *S)                //请求一个单位的资源
{
    S->value --;                    //资源减少一个
    if (S->value<0) block(S->list); //进程自我阻塞, 放弃处理机, 队尾
}

signal(semaphores *S)               //释放一个单位资源
{
    S->value++;                     //资源增加一个
    if (S->value<=0) wakeup(S->list); //唤醒等待队列中的第一个进程
}
```

AND型信号量同步的基本思想：将进程在整个运行过程中需要的所有资源，一次性全部分配给进程，待进程使用完后再一起释放。

---

对若干个临界资源的分配，采用原子操作（要么不分配，要么全部分配给一个进程，死锁理论）。

---

在wait(S)操作中增加了一个“AND”条件，故称之为AND同步，或同时wait(S)操作，即**Swait(Simultaneous wait)**。

```

Swait(S1, S2, ..., Sn) {
    while (TRUE) {
        if (Si >= 1 && ... && Sn >= 1) {
            for (i = 1; i <= n; i++) Si--;
            break;
        }
        else {
            place the process in the waiting
            queue associated with the first Si found
            with Si < 1, and set the program count of
            this process to the beginning of Swait
            operation
        }
    }
}

```

```

Signal(S1, S2, ..., Sn) {
    while (TRUE) {
        for (i = 1; i <= n; i++) {
            Si++;

            Remove all the process
            waiting in the queue associated
            with Si into the ready queue.
        }
    }
}

```





在记录型信号量中，wait或signal仅能对某类临界资源进行一个单位的申请和释放，当需要对N个单位进行操作时，需要N次wait/signal操作，效率低下




**扩充AND信号量**：对进程所申请的所有资源以及每类资源不同的资源需求量，在一次P、V原语操作中完成申请或释放

- 进程对信号量 $S_i$ 的测试值是该资源的分配下限值 $t_i$ ，即要求 $S_i \geq t_i$ ，否则不予分配。一旦允许分配，进程对该资源的需求值为 $d_i$ ，即表示资源占用量，进行 $S_i = S_i - d_i$ 操作
- $Swait(S1, t1, d1, ..., Sn, tn, dn)$
- $Ssignal(S1, d1, ..., Sn, dn)$

 利用信号量实现进程互斥  
➤ 设置互斥信号量

 利用信号量实现进程同步  
➤ 设置同步信号量

 利用信号量实现前趋关系

```
//设置互斥型信号量
semaphore mutex;
mutex=1; // 初始化为 1

while(1)
{
    wait(mutex);
    临界区;
    signal(mutex);
    剩余区;
}
```

**Case A:**  $\text{mutex} = 1$

- 两个进程都没有进入需要互斥访问的临界区

**Case B:**  $\text{mutex} = 0$

- 有一个进程进入临界区运行，临界区外无进程等待

**Case C:**  $\text{mutex} = -1$

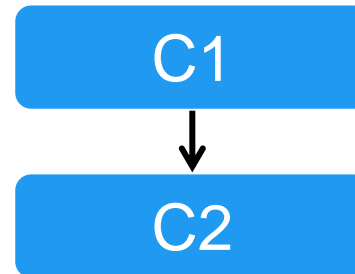
- 有一个进程进入临界区运行，另一个进程因等待而被阻塞在信号量队列中，需已在临界区的进程退出时唤醒

wait和signal必须成对出现

协作进程除了需要互斥的方位资源，还需要相互制约和传递消息，以同步它们之间的运行，利用信号量同样可以达到目的。

- 实现各种同步问题（让各并发进程按要求有序地推进）
- 例子：P1和 P2 需要代码段 C1 比C2先运行

**semaphores s=0; //主要用于传递消息**



```
P1()
{
    C1;
    signal(s);
    ...
}
```

```
P2()
{
    ...
    wait(s);
    C2;
}
```



**Case A:**  $s = 0$

➤ C1/C2均未执行

**Case B:**  $s = 1$

➤ C1已经执行, C2可以执行

**Case C:**  $s = -1$

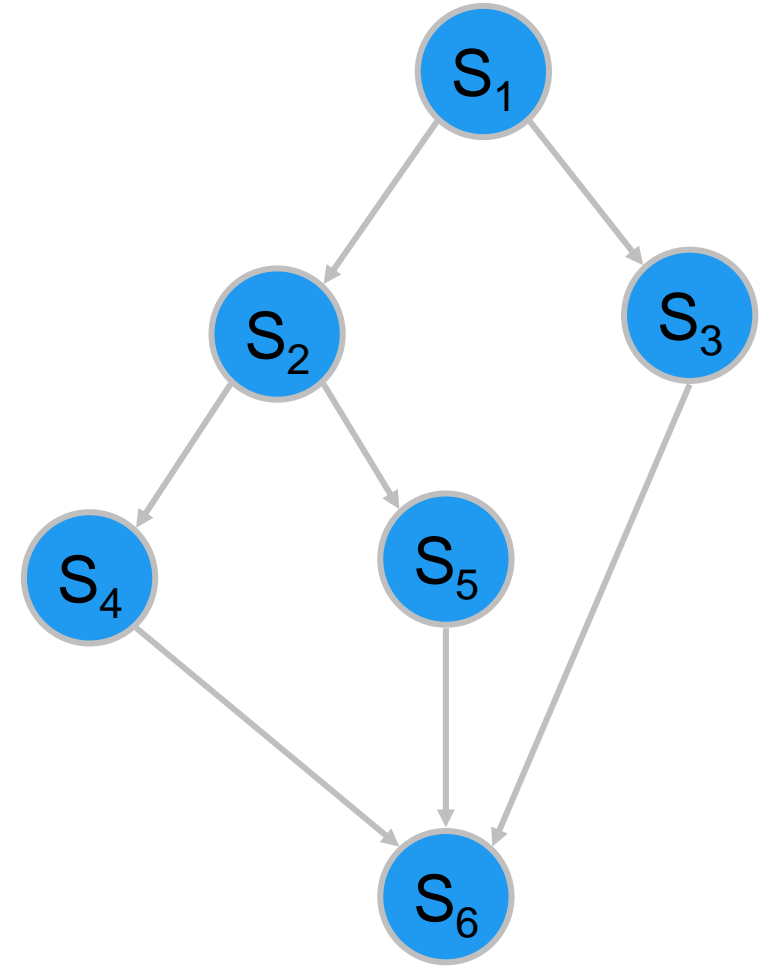
➤ P2进程尝试先执行C2, 结果被扔进了阻塞队列。  
等到C1执行完毕, 此时s的值会自加, 并且会  
wakeup P2进程

```
P1()
{
    C1;
    signal(s);
    ...
}
```

```
P2()
{
    ...
    wait(s);
    C2;
}
```

其实每一对前驱关系都是一个进程同步问题(需要保证一前一后的操作)

```
main(){  
    Semaphore a,b,c,d,e,f,g;  
    a.value=0;b.value=0;c.value=0;  
    d.value=0;e.value=0;f.value=0;g.value=0;  
    cobegin  
        { S1;signal(a);signal(b); }  
        { wait(a);S2;signal(c) ;signal(d);}  
        { wait(b);S3;signal(e); }  
        { wait(c);S4;signal(f); }  
        { wait(d);S5;signal(g); }  
        { wait(e);wait(f);wait(g);S6; }  
    coend  
}
```





## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



4.4 信号量机制



**4.5 管程机制**



4.6 经典进程的同步问题



4.7 Linux进程同步机制

## 第4章 进程同步

---



## 问题

- 需要程序员实现，编程困难
- 维护困难
- 容易出错
  - wait/signal位置错
  - wait/signal不配对



## 解决方法

- 由**编程语言**解决同步互斥问题
- 管程 (1970s, Hoare和Hansen)

信号量：**分散式**  
管 程：**集中式**



## 发明ATM管程



在计算机银行和客户之间担任中介

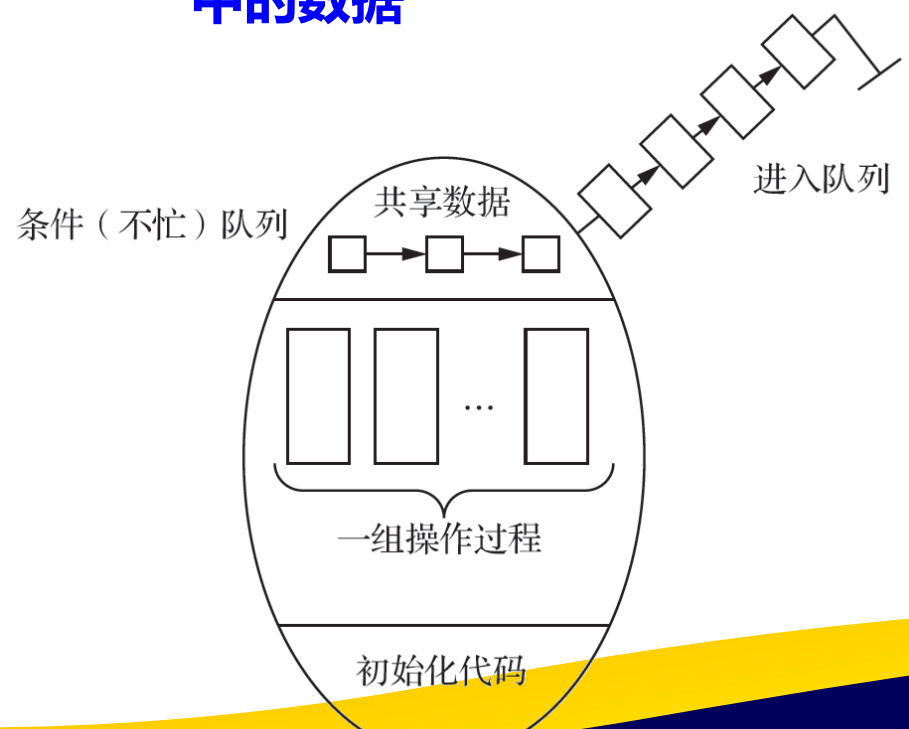


- 去计算机银行取钱，只有一个空窗口
- 人→进程，钱→计算机的共享资源，一般是硬件设备或一群共享变量
- 每个人都拥向窗口，场面混乱不堪

- ATM管程封装了钱和对外开放了一些存取钱的操作；
- 封闭屋子里，一次只服务一人（让进程互斥使用）；
- ATM屋里有人时，其他需依次排队使用；
- 一个人不能一直使用ATM，故定义需要定义condition进行约束。

## 管程 (Monitors) 定义

- 一个管程定义了一个**数据结构**和能为并发进程所执行（在该数据结构上）的**一组操作**，这组操作能**同步进程**和**改变管程中的数据**



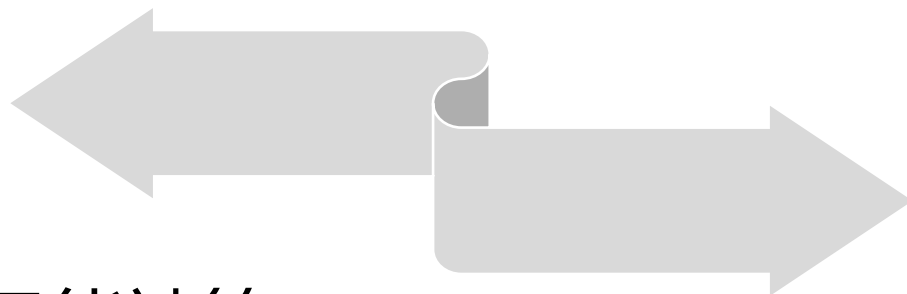
## 语法描述如下:

```
Monitor monitor_name {                               /*名程管*/
    share variable declarations;                       /*明说量变享共*/
    cond declarations;                                /*明说量变件条*/
    public:                                             /*程过的用调程进被能*/
        void P1(.....) {.....} /*程过的作操构结据数对*/
        void P2(.....) {.....}
        .....
        void (.....) {.....}
        .....
    {
        initialization code;                          /*体主程管*/
        .....                                          /*码代化始初*/
    }
}
```



## 互斥

- 管程中的变量只能被管程中的操作访问
- 任何时候只有一个进程在管程中操作
- 类似临界区
- 由编译器完成



## 同步

- 条件变量
- 唤醒和阻塞操作



condition x, y;



条件变量的操作

➤ 阻塞操作: wait

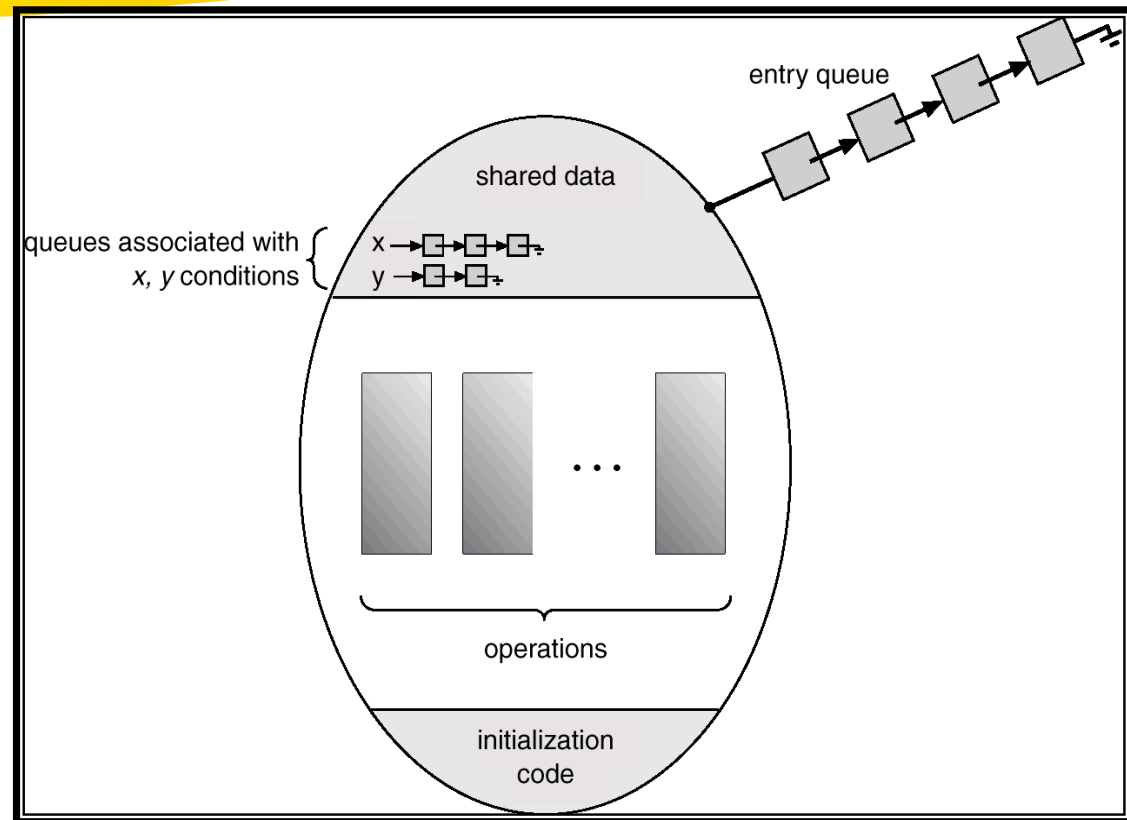
➤ 唤醒操作: signal



x.wait(): 进程阻塞直到另外一个进程调用x.signal()



x.signal(): 唤醒另外一个进程  
(不自加)



一人在操作ATM时被电话打断，打断原因记为x，并执行x.wait，让其离开ATM机，去接电话的队列中等待。等打完电话，即调用x.signal后，可回去继续操作ATM了（正在操作ATM的人操作完后，才能重新进去）





问题：管程内可能存在不止  
1个进程（互斥性被破坏）

- 例如：进程P调用signal操作唤醒进程Q后，在那一瞬间，管程内有两个进程。









存在的可能处理方式：

- P等待，直到Q离开管程或等待另一条件（Hoare）。
- Q等待，直到P离开管程或等待另一条件（Hansen）。

- **管程相当于对临界资源的操作进行封装**：当进程要对资源进行操作时，只需调用管程中的方法即可，而不用进程自己担心同步和互斥的问题，管程的内部有一套机制进行同步与互斥
- **管程中每次只允许一个进程进入管程**
- 当调用管程的进程因为某原因阻塞或者挂起时，把这个**原因定义为一个条件变量x**
- x.wait操作就是把自己放到一个队列上，这个队列上的进程都是因为x原因而阻塞的
- x.signal操作就是让在x阻塞队列上的一个进程重新启动。



## 内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  **4.6 经典进程的同步问题**
-  4.7 Linux进程同步机制

# 第4章 进程同步

---



## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题 ➡



4.7 Linux进程同步机制

## 第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ 4.6.2 哲学家进餐问题

■ 4.6.3 读者-写者问题

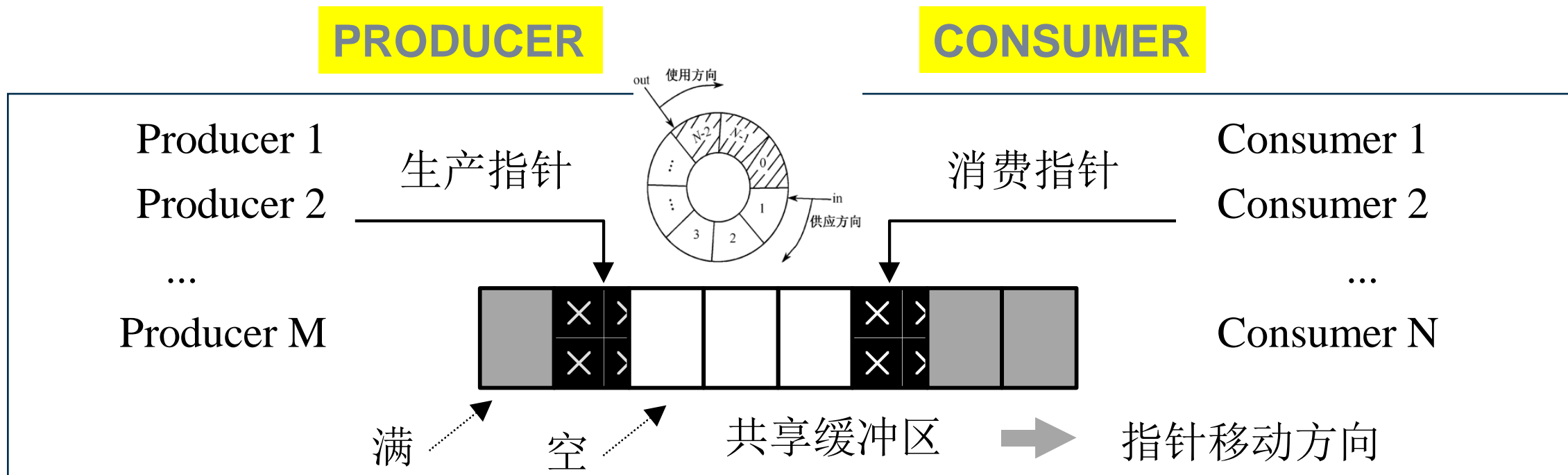
生产者-消费者问题是相互合作进程关系的一种抽象

利用记录型信号量实现：

- 假定，在生产者和消费者之间的**公用缓冲池**中，具有**n个缓冲区**，可利用互斥信号量mutex使诸进程实现对缓冲池的互斥使用；
- 利用**资源信号量**empty和full分别表示缓冲池中**空缓冲区**和**满缓冲区的数量**。
- 又假定这些生产者和消费者相互等效，只要缓冲池未**满**，生产者便可将消息送入缓冲池；只要缓冲池未**空**，消费者便可从缓冲池取走一个消息

其它解决方案：AND信号集、管程

- 生产者 (M个) : 生产产品, 并放入缓冲区
- 消费者 (N个) : 从缓冲区取产品消费
- 问题: 如何实现生产者和消费者之间的同步和互斥?



生产者:

```
{  
    ...  
    生产一个产品  
    ...  
  
    ...  
    把产品放入指定缓冲区  
    ...  
}
```

消费者:

```
{  
    ...  
  
    ...  
    从指定缓冲区取出产品  
    ...  
  
    ...  
    消费取出的产品  
    ...  
}
```

## 临界资源判断



## 生产者

- 把产品放入指定缓冲区
- **in**: 所有的生产者对in指针需要互斥
- **counter**: 所有生产者消费者进程对counter互斥

```
buffer[in] = nextp;  
in = (in + 1) % N;  
counter++;
```



## 消费者

- 从指定缓冲区取出产品
- **out**: 所有的消费者对out指针需要互斥
- **counter**: 所有生产者消费者进程对counter互斥

```
nextc = buffer[out];  
out = (out + 1) % N;  
counter--;
```



生产者:

{

...

生产一个产品

...

...

把产品放入指定缓冲区

...

}

消费者:

{

...

...

从指定缓冲区取出产品

...

...

消费取出的产品

...

}

临界区

临界区

```
semaphore *m;    m->vaule = 1;
```

生产者:

```
{
```

```
...
```

```
生产一个产品
```

```
...
```

```
wait(m);
```

```
...
```

临界区

```
把产品放入指定缓冲区
```

```
...
```

```
signal(m);
```

```
}
```

消费者:

```
{
```

```
...
```

```
wait(m);
```

```
...
```

```
从指定缓冲区取出产品
```

```
...
```

```
signal(m);
```

```
...
```

```
消费取出的产品
```

```
...
```

```
}
```

临界区



## 两者需要协同的部分

- **生产者**：把产品放入指定缓冲区（关键代码C1）
- **消费者**：从满缓冲区取出一个产品（关键代码C2）



## 三种运行次序（不同条件下不同运行次序）

- 所有缓冲区（缓冲池）空时，谁先运行？
- 所有缓冲区（缓冲池）满时，谁先运行？
- 缓冲区（缓冲池）有空也有满时，谁先运行？





## 生产者

...

生产一个产品

...

同步：判断

1) 判断是否能获得一个空缓冲区，如果不能则阻塞

**C1:**把产品放入指定缓冲区

临界区

2) 满缓冲区数量加1，如果有消费者由于等消费产品而被阻塞，则唤醒该消费者

同步：通知



## 消费者

同步：判断

1) 判断是否能获得一个满缓冲区（非空缓冲池），如果不能则阻塞

从满缓冲取出一个产品

临界区

2) 空缓冲区数量加1，如果有生产者由于等空缓冲区而阻塞，则唤醒该生产者

同步：通知



## 共享数据

semaphore **\*full**, **\*empty**, \*m; //full:满缓冲区数量 empty: 空缓冲区数量

初始化:

**full->value = 0;**      **empty->vaule = N;** m->vaule = 1;

---

m:互斥型信号量 (实现各个进程对缓冲池的互斥访问)  
full和empty都是同步型信号量 (同步空缓冲区和满缓冲区的数量)

生产者:

```
{  
    ...  
    生产一个产品  
    ...  
    wait(empty);  
    wait(m);  
    ...  
    C1: 把产品放入指定缓冲区  
    ...  
    signal(m);  
    signal(full);  
}
```

当empty大于0时, 表示有空缓冲区, 继续执行; 否则, 表示无空缓冲区, 当前生产者阻塞。

把full值加1, 如果有消费者等在full的队列上, 则唤醒该消费者

消费者:

```
{  
    ...  
    wait(full);  
    wait(m);  
    ...  
    C2: 从指定缓冲区取出产品  
    ...  
    signal(m);  
    signal(empty);  
    ...  
    消费取出的产品  
    ...  
}
```

当full大于0时, 表示有满缓冲区, 继续执行; 否则, 表示无满缓冲区, 当前消费者阻塞。

把empty值加1, 如果有生产者等在empty的队列上, 则唤醒该生产者。

生产者:

{

...

生产一个产品

...

**wait(m);**

**wait(empty);**

...

**C1:** 把产品放入指定缓冲区

...

signal(m);

signal(full);

}

自己已阻塞，无法释放资源，但又期待其他进程获得此资源后唤醒自己，

- 缓冲区全满时 ( $\text{empty}=0$ )，若一生产者进程先执行 **wait(m)**，并获得成功。当再执行 **wait(empty)** 操作时，将因无空缓冲区 → 失败 → 阻塞状态，它期待消费者执行  $\text{signal}(\text{empty})$  来唤醒自己
- 在此之前，它不可能执行  $\text{signal}(m)$  操作，从而使企图通过  $\text{wait}(m)$  进入自己的临界区的其他生产者和所有的消费者进程全部进入阻塞状态，从而引起系统死锁
- 一进程一定要先具备修改的资格  $\text{wait}(\text{empty})$ ，才能去修改  $\text{wait}(m)$



生产者:

```
{  
    ...  
    生产一个产品  
    ...  
    wait(empty);  
    wait(m);  
    ...  
    C1: 把产品放入指定缓冲区  
    ...  
    signal(m);  
    signal(full);  
}
```

一个同步P操作与一个互斥P操作在一起时, 同步P操作在互斥P操作前, 否则死锁

消费者:

```
{  
    ...  
    wait(full);  
    wait(m);  
    ...  
    C2: 从指定缓冲区取出产品  
    ...  
    signal(m);  
    signal(empty);  
    ...  
    消费取出的产品  
    ...  
}
```

```
int in=0,out=0;  
item buffer[n];  
semaphore mutex=1,empty=n,full=0;
```

```
void producer() {  
    do {  
        produce an item nextp;  
        ...  
        Swait(empty, mutex);  
        buffer[in]= nextp;  
        in = (in+1) % n;  
        Ssignal(mutex,full);  
    }while(TRUE);  
}
```

```
void consumer() {  
    do {  
        Swait(full,mutex);  
        nextc=buffer[out];  
        out= (out+1) % n;  
        Ssignal(mutex,empty);  
        consume the item in nextc;  
        ...  
    }while(TRUE); }
```

```
Monitor producerconsumer {  
    item buffer[N];  
    int in,out;  
    condition notfull,notempty;  
    int count;  
    public:  
    void put(item x) {  
        if (count>=N) cwait(notfull);  
        buffer[in] = x;  
        in = (in+1) % N;  
        count++;  
        csignal(notempty);  
    }  
}
```

生产进程阻塞，等待notfull条件成立

唤醒等待notempty条件的阻塞进程（消费进程因空而阻塞）

```
void get(item x) {  
    if (count<=0) cwait(notempty);  
    x = buffer[out];  
    out = (out+1) % N;  
    count--;  
    csignal(notfull);  
}  
  
{  
    in=0;out=0;count=0;  
}  
} PC;
```

消费阻塞，等待notempty条件成立

唤醒等待notfull条件的阻塞进程（生产进程因满而阻塞）

在利用管程方法来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为Proclucer-Consumer，或简称为PC。其中包括两个过程：

- (1) **put(x)过程**：生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当 $\text{count} \geq n$ 时，表示缓冲池已满，生产者须等待。
- (2) **get(x)过程**：消费者利用该过程从缓冲池中取出一个产品，当 $\text{count} \leq 0$ 时，表示缓冲池中已无可取用的产品，消费者应等待。

在利用管程方法来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为Proclucer-Consumer，或简称为PC。

对于条件变量notfull和notempty，有以下两个过程可对它进行操作：

- (1) **cwait (conditional wait)**：当管程被一个进程占用时，其他进程调用该过程时会阻塞，并且被会挂在相应的条件上
- (2) **csignal (conditional signal)**：唤醒在cwait上执行后阻塞在条件上的进程  
(如果有多个选择一个进行唤醒；如果没有，则直接返回)

```
void producer() {  
    item x;  
    while(TRUE) {  
        .....  
        produce an item in nextp;  
        PC.put(x);  
    }  
}  
  
void consumer() {  
    item x;
```

```
while(TRUE) {  
    PC.get(x);  
    consume the item in nextc;  
    .....  
}  
}  
  
void main() {  
    cobegin  
        producer(); consumer();  
    coend  
}
```



## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题 ➡



4.7 Linux进程同步机制

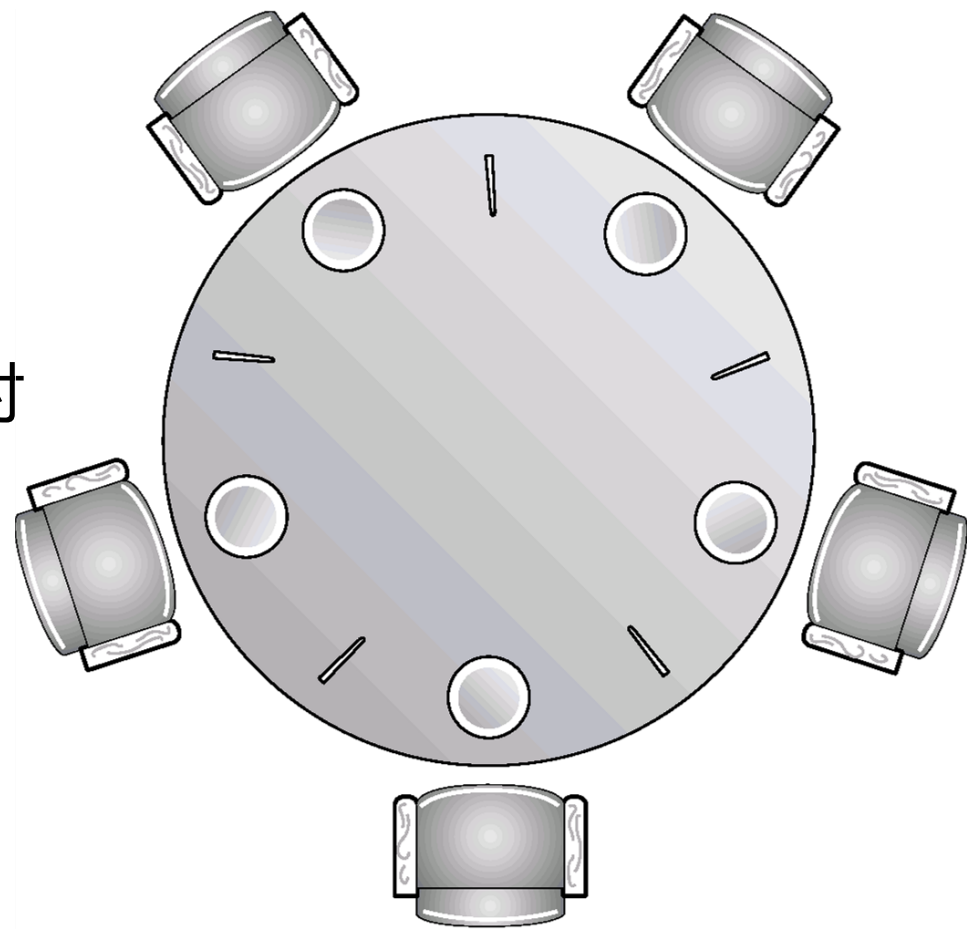
## 第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ **4.6.2 哲学家进餐问题**

■ 4.6.3 读者-写者问题

- 五个哲学家的生活方式：思考和进餐  
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  
平时哲学家思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在拿到两支筷子时才能进餐  
进餐毕，放下筷子又继续思考
- 解决方案：
  - ❑ 记录型信号量；
  - ❑ AND信号量集、管程。





Semaphore **chopstick**[5] = {1,1,1,1,1};

Philosopher i:

```
do {  
    wait(chopStick[i]); // get left chopstick  
    wait(chopStick[(i + 1) % 5]); // get right chopstick  
    ...  
    // eat for awhile  
    ...  
    signal(chopStick[i]); //return left chopstick  
    signal(chopStick[(i + 1) % 5]); // return right chopstick  
    ...  
    // think for awhile  
    ...  
} while (true)
```

**可能引起死锁**，如五个哲学家同时饥饿而各自拿起左筷子时，会使信号量chopstick均为0；因此他们试图去拿右筷子时，无法拿到而无限期等待。

**解决方法：**

- ① 最多允许4个哲学家同时坐在桌子周围
- ② 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子。
- ③ 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之

```
semaphore chopstick chopstick[5]={1,1,1,1,1};  
do {  
    ...  
    //think  
    ...  
    Sswait(chopstick[(i+1)%5], chopstick[i]);  
    ...  
    //eat  
    ...  
    Ssignal(chopstick[(i+1)%5], chopstick[i]);  
} while[TRUE];
```



```
monitor dp{  
    enum { thinking, hungry, eating} state [5];  
    condition self [5];  
    initialization_code() {  
        for (int i = 0; i < 5; i++)  
            state[i] = thinking;  
    }  
    void pickup (int i) {  
        state[i] = hungry;  
        test(i);  
        if (state[i] != eating) self[i].wait();  
    }  
}
```

当自己饥饿，但又  
拿不到筷子时阻塞  
自己

```
void putdown (int i) {  
    state[i] = thinking;  
    // test left and right neighbors  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```

当邻桌都没有在吃，自己又饿了，才将自己设置为eating状态

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != eating) &&(state[i] == hungry)  
        &&(state[(i + 1) % 5] != eating) ) {  
        state[i] = eating ;  
        self[i].signal() ;  
    }  
}
```

哲学家i的活动可描述为：

```
do {  
    dp.pickup (i);  
    ...  
    eat  
    ...  
    dp.putdown (i);  
} while[TRUE];
```

思考：该方案确保了相邻的哲学家不会同时进餐，且不会死锁，但是自己可能被饿死，该如何解决



## 内容导航:



4.1 进程同步的概念



4.2 软件同步机制



4.3 硬件同步机制



4.4 信号量机制



4.5 管程机制



4.6 经典进程的同步问题 ➡



4.7 Linux进程同步机制

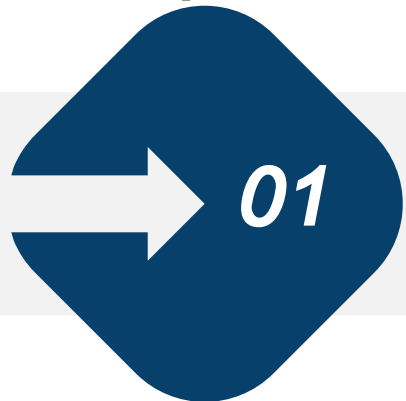
## 第4章 进程同步

■ 4.6.1 生产者-消费者问题

■ 4.6.2 哲学家进餐问题

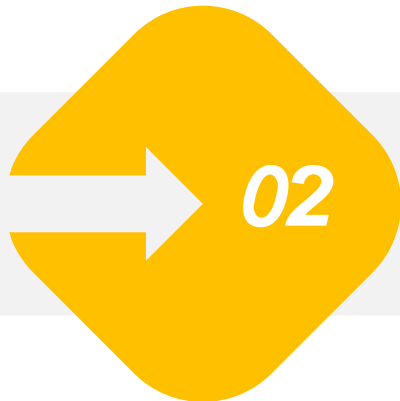
■ **4.6.3 读者-写者问题**

有两组并发  
进程：



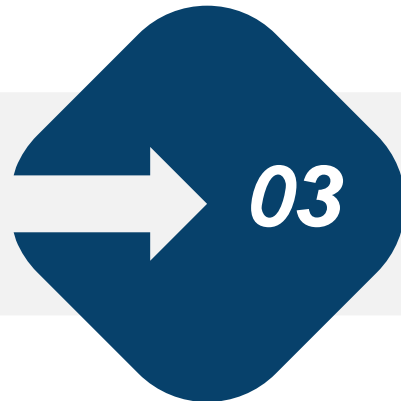
- 读者和写者，共享一组数据区。

要求：



- 允许多个读者同时执行读操作；
- 不允许读者、写者同时操作；
- 不允许多个写者同时操作。

分类：



- 读者优先(第一类读者写者问题)
- 写者优先(第二类读者写者问题)

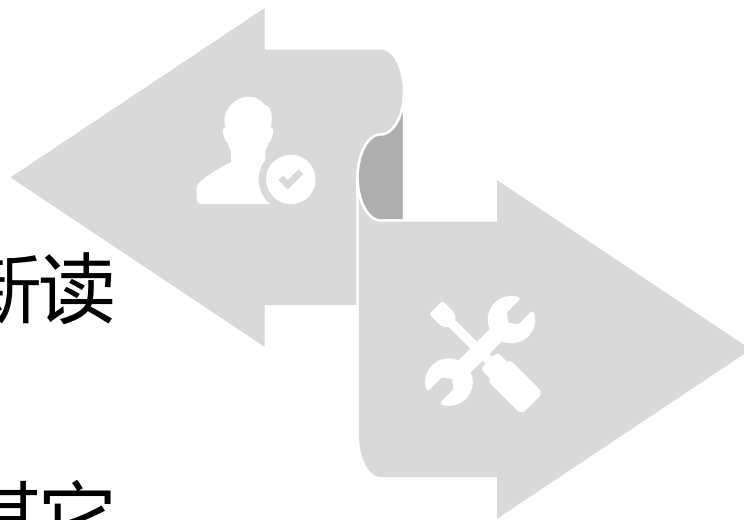
解决方案：



- 记录型信号量
- 信号量集

如果读者来:

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



如果写者来:

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



初始化:

semaphore mutex=1, w=1;

int readcount=0;

void **writer**() {

do {

wait(w);

...

**写**

...

signal(w);

}

} while(TRUE)

读写进程间的互斥  
信号量

是否有读者在读,  
有则将自己阻塞

自己写完, 释放资源,  
让读者读

void **reader**() {

do {

wait(mutex);

if (readcount==0) wait(w);

readcount++;

signal(mutex);

...

**读**

...

wait(mutex);

readcount--;

if (readcount==0) signal(w);

signal(mutex);

}while(TRUE);

}

# 读者优先(第一类读者写者问题)

初始化:

```
semaphore mutex=1, w=1;  
int readcount=0;
```

```
void writer() {  
    do {  
        wait(w);  
        ...  
        写  
        ...  
        signal(w);  
    }  
} while(TRUE)
```

读者进程个数的互斥信号量

读者进程个数 (读者进程内的临界资源)

```
void reader() {  
    do {
```

读前

```
        wait(mutex);  
        if (readcount==0) wait(w);  
        readcount++;  
        signal(mutex);
```

...  
读

读后

```
        ...  
        wait(mutex);  
        readcount--;  
        if (readcount==0) signal(w);  
        signal(mutex);  
    } while(TRUE);  
}
```

当读者个数为0, 测试是否有写者在写, 若有, 将自己阻塞

没有读者, 释放资源, 允许写者进行写操作



## 问题描述

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）



如何用PV操作实现？（思考题）

```
void reader() {  
    do {  
        Swait(L,1,1)  
        Swait(mx,1,0);  
        ...  
        perform read operation;  
        ...  
        Ssignal(L,1);  
    } while(TRUE);  
}
```

---

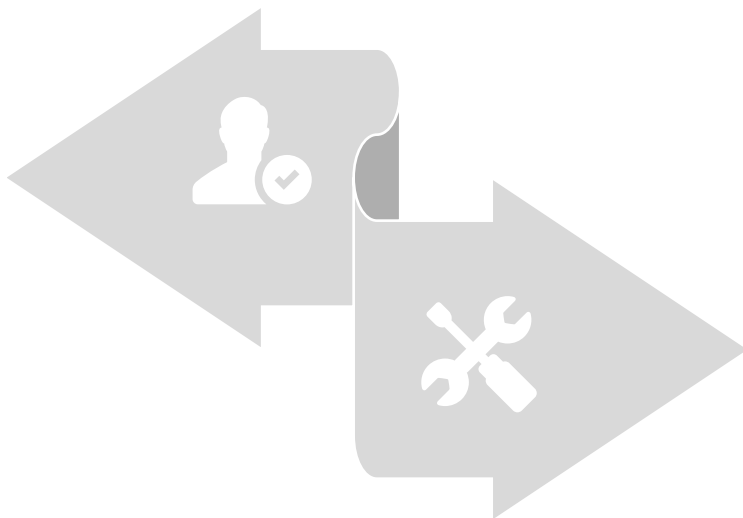
---

```
void writer() {  
    do {  
        Swait(mx,1,1; L,RN,0);  
        perform write operation;  
        Ssignal(mx,1);  
    }while(TRUE);  
}
```

```
int RN;  
semaphore L=RN, mx=1;
```

## 信号量的使用:

- 信号量必须置一次且只能置一次初值，初值不能为负数
- 除了初始化，只能通过执行P、V操作来访问信号量



## 使用中存在的问题

- 死锁
- 饥饿



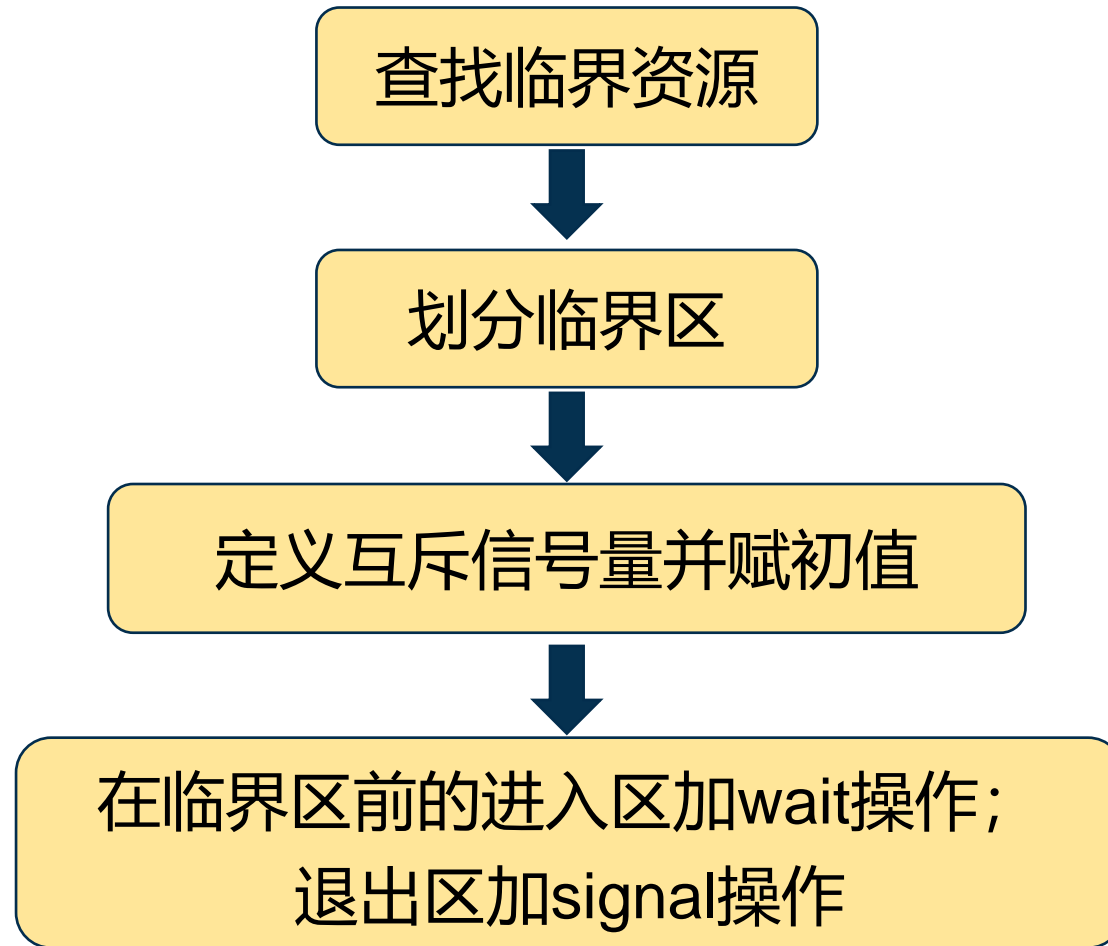
**死锁**：两个或多个进程无限期地等待一个事件的发生，而该事件正是由其中的一个等待进程引起的。

例如：S和Q是两个初值为1的信号量

$P_0$	$P_1$
P(S);	P(Q);
P(Q);	P(S);
⋮	⋮
V(S);	V(Q);
V(Q)	V(S);



**饥饿**：无限期地阻塞，进程可能永远无法从它等待的信号量队列中移去（只涉及一个进程）。



①找出需要同步的代码片段（关键代码）

②分析这些代码片段的执行次序

③增加同步信号量并赋初始值

④在关键代码前后加wait和signal操作

同步分析较为困难！





## 信号量的物理含义

- $S > 0$  表示有  $S$  个资源可用;
- $S = 0$  表示无资源可用;
- $S < 0$  则  $|S|$  表示  $S$  等待队列中的进程个数。
- $P(S)$ : 表示申请一个资源;
- $V(S)$  表示释放一个资源。信号量的初值应该大于等于 0



## PV操作的使用

- P.V 操作必须成对出现, 有一个 P 操作就一定有一个 V 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  $P(S_1)$  和  $P(S_2)$  两个操作在一起, 那么 P 操作的顺序至关重要, 一个同步 P 操作与一个互斥 P 操作在一起时, 同步 P 操作在互斥 P 操作前, 否则死锁
- 而两个 V 操作无关紧要

同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）

不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；



易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；

**正确性难以保证：**操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误；



## 内容导航:

-  4.1 进程同步的概念
-  4.2 软件同步机制
-  4.3 硬件同步机制
-  4.4 信号量机制
-  4.5 管程机制
-  4.6 经典进程的同步问题
-  **4.7 Linux进程同步机制**

# 第4章 进程同步

---

## Linux并发的主要来源

- **中断处理**：当进程在访问某个临界资源的时候发生了中断。随后进入中断处理程序，假设在中断处理程序中也访问了该临界资源。尽管不是严格意义上的并发，可是也会造成了对该资源的竞态。
- **内核态抢占**：当进程在访问某个临界资源的时候发生内核态抢占，随后进入了高优先级的进程。假设该进程也访问了同一临界资源，那么就会造成进程与进程之间的并发。
- **多处理器的并发**：多处理器系统上的进程与进程之间是严格意义上的并发，每一个处理器都能够独自调度执行一个进程。在同一时刻有多个进程在同一时候执行。



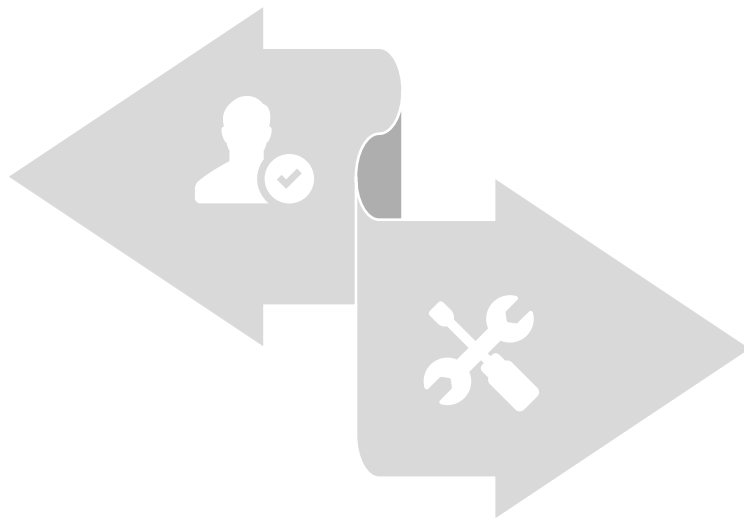
# 为什么需要Linux同步机制

进程是操作系统调度的实体，系统具备了并发运行多个进程的能力，但也导致了各个进程之间的资源竞争和共享。另外，因为中断、异常机制的引入，以及内核态抢占都导致了这些内核运行路径（进程）以交错的方式运行。

对于这些交错路径运行的内核路径，如不采取必要的同步措施。将会对一些重要数据结构进行交错访问和改动。从而导致这些数据结构状态的不一致，进而导致系统崩溃。因此。**为了确保系统高效稳定有序地运行，linux必需要采用同步机制。**

## Linux并发的主要来源:

- 中断处理、内核态抢占、多处理器的并发。



## 同步方法:

- 原子操作
- 自旋锁 (spin lock)
  - 不会引起调用者阻塞
- 信号量 (Semaphore)
- 互斥锁 (Mutex)
- 禁止中断 (单处理器不可抢占系统)

## 同步方法:

### ■ 原子操作

绝不会在执行完毕前被任何其他任务或事件打断，也就说，它的最小的执行单位，不可能有比它更小的执行单位。

原子操作需要硬件的支持，因此是架构相关的，其API和原子类型的定义都定义在内核源码树的include/asm/atomic.h文件中，它们都使用汇编语言实现，因为C语言并不能实现这样的操作。

原子操作主要用于实现资源计数，很多引用计数(refcnt)就是通过原子操作实现的。例如，Linux内核通过提供atomic\_t类型提供了一系列的原子操作。

## 同步方法：

### ■ 自旋锁 (spin lock)

一个执行单元要想访问被锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取锁时已经有保持者，那么获取锁操作将 **“自旋”** (**忙等待状态**) 在那里，直到该自旋锁的保持者释放了锁。

**自旋锁不会引起调用者睡眠（自我阻塞）**。这是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁（自我阻塞将引起上下文切换，效率低）。



## 同步方法:

### ■ 信号量 (semaphore)

**信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源。**

一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。

当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

除了一般信号量外，Linux系统还有读写信号量 (rw\_semaphore) 。



## 同步方法:

### ■ 互斥锁 ( Mutex )

初始值为1的信号量就变成互斥信号量/互斥锁 (Mutex) , 即同时只能有一个任务可以访问信号量保护的共享资源。

## 同步方法:

### ■ 禁止中断 (针对单处理器不可抢占系统)

对于一个微处理器核，所以禁止中断是实现临界代码段的有效手段。例如，在  $\mu\text{C}/\text{OS-II}$  等嵌入式系统中，可以使用禁止中断保护临界代码段。（临界代码段一般较短，如果太长，会影响整个系统任务的调度）

Linux也有类似的机制，它保证内核控制路径可以继续执行，其访问的数据结构不会被中断处理程序破坏

## 同步方法：

- 原子操作
- 自旋锁 (spin lock)
  - 不会引起调用者阻塞
- 信号量 (Semaphore)
- 互斥锁 (Mutex)
- 禁止中断 (单处理器不可抢占系统)



# 学而时习之（第4章总结）

第1章 操作系统引论

第2章 进程的描述与控制

第3章 处理机调度与死锁

**第4章 进程同步**

第5章 存储器管理

第6章 虚拟存储器

第7章 输入/输出系统

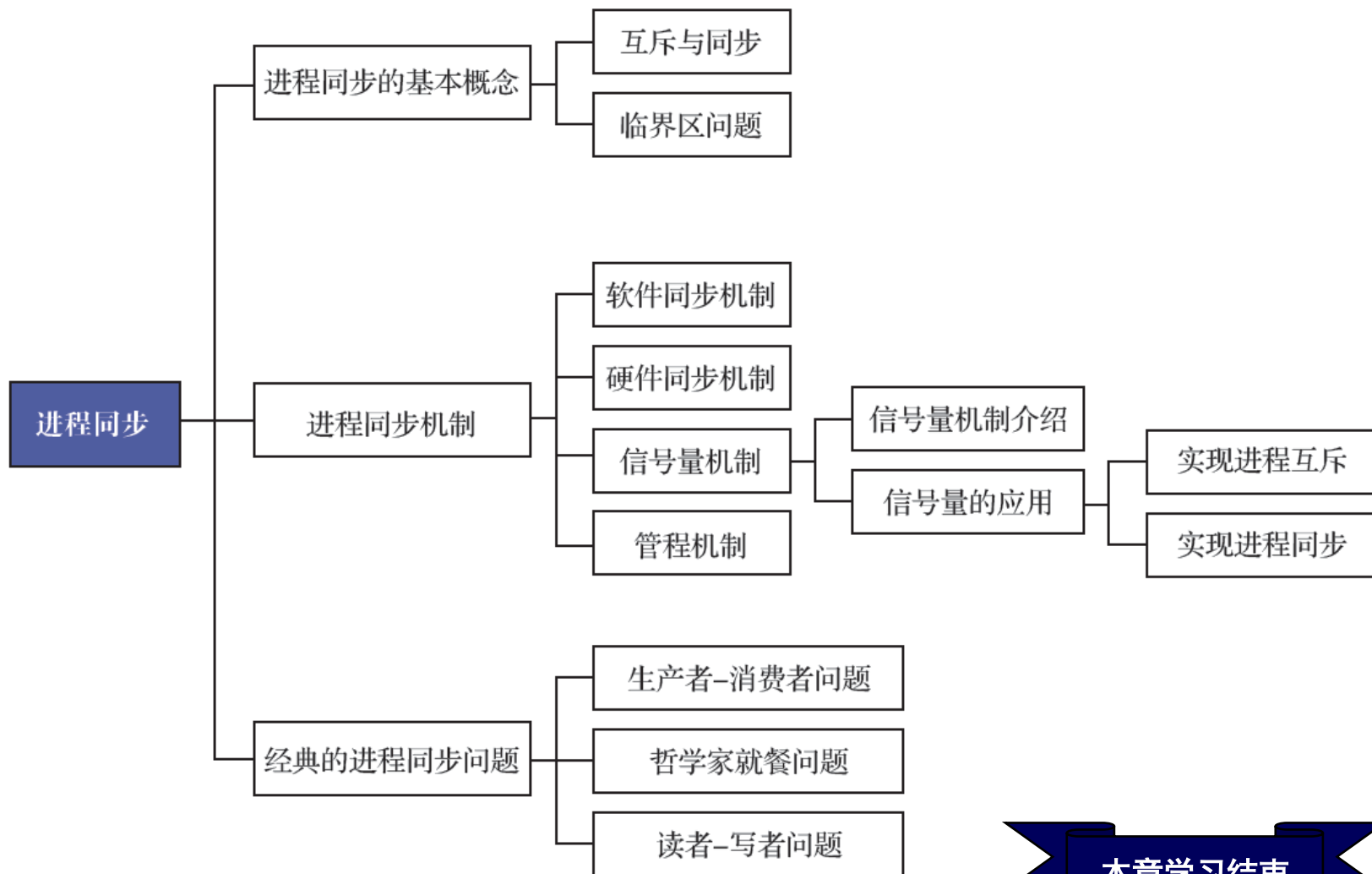
第8章 文件管理

第9章 磁盘存储器管理

第10章 多处理机操作系统

第11章 虚拟化和云计算

第12章 保护和安全



本章学习结束

简答题

1

2

3

4

5

6

7

8

9

10

11

12

计算题

13

14

15

综合应用题

16

17

18

19

20

21

标黄色为本次作业

## Windows XP系统宣布“退役”，国产操作系统迎来新机遇

2001年10月25日，由微软公司研发的Windows XP系统正式发布。

Windows XP系统是一款面向个人计算机和平板电脑的操纵系统，具有界面华丽、兼容性好、功能丰富、管理方便等特点，集成了微软公司的防火墙技术，支持在第一时间为用户所要升级的硬件提供相关程序下载等功能。相比于Windows系列之前的操作系统，Windows XP系统的用户体验得到了极大程度的提升。这也就引出了该操作系统名称中“XP”的由来，其即“experience”（体验）一词的缩写。

2014年4月8日，这款功能多、体验好，在各个领域经过13年广泛应用的操作系统正式宣布“退役”，即微软公司终止对Windows XP系统继续提供技术支持。



## Windows XP系统宣布“退役”，国产操作系统迎来新机遇

该事件实际上对我国工薪阶层以及行动迟缓的单位机关造成了一定的影响，但是同时也给国产操作系统的发展带来了新机遇。虽说当时我国还没有真正意义上自己的操作系统（没有自己的操作系统，就得继续受制于人），但是，计算机技术对国人来说具有独到优势（在计算机相关技术国际知名企业中有大量国人就业），而且我国的整体经济状况呈良好发展态势，因此，无论从哪个层面讲，我国都有必要投入人力、物力来研发自己的操作系统。

我们坚信，在众多知识分子与科技人才的共同努力下，我国必定能够早日研发成功自主可控的国产操作系统，从此不再受制于人！