

第3章 进程之间的并发 控制和死锁



本章主要内容

1. 并发进程的特点
2. 进程之间的低级通信：互斥、同步、信号量和P/V操作、经典IPC问题
3. 管程
4. 进程的高级通信：消息传递，共享内存
5. 死锁：多进程竞争有限资源



3.1 并发进程的特点

(1) 对资源的共享引起的互斥关系

进程之间本来是相互独立的，但由于共享资源而产生了关系。间接制约关系，互斥关系。

(2) 协作完成同一个任务引起的同步关系

一组协作进程要在某些同步点上相互等待发信息后才能继续运行。直接制约关系。同步关系。

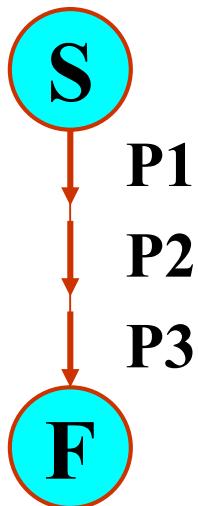
(3) 进程之间的前序关系

由于进程之间的互斥同步关系，使得进程之间具有了前序关系，这些关系决定了各个进程创建和终止的时间。

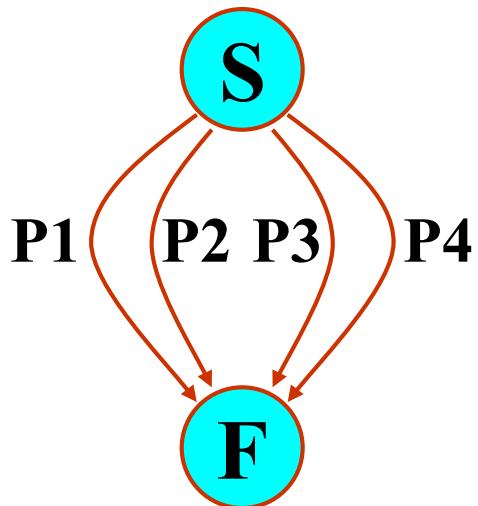
(a) $S(P1, P2, P3)$

(b) $P(P1, P2, P3, P4)$

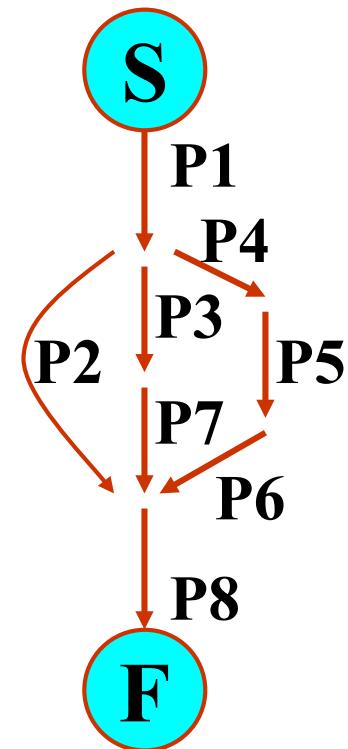
(c) $S(P1, P(P2, S(P3, P7), S(P4, P5, P6)), P8)$



(a)顺序关系



(b)并行关系



(c)一般关系



3.2 进程之间的低级通信

- 进程之间交换信息被称为进程间通信。
- 进程间的低级通信：通过信号量实现进程之间的互斥和同步关系。
- 进程间通信问题——IPC问题

(Inter Process Communication, IPC)



3.2.1 进程之间的互斥

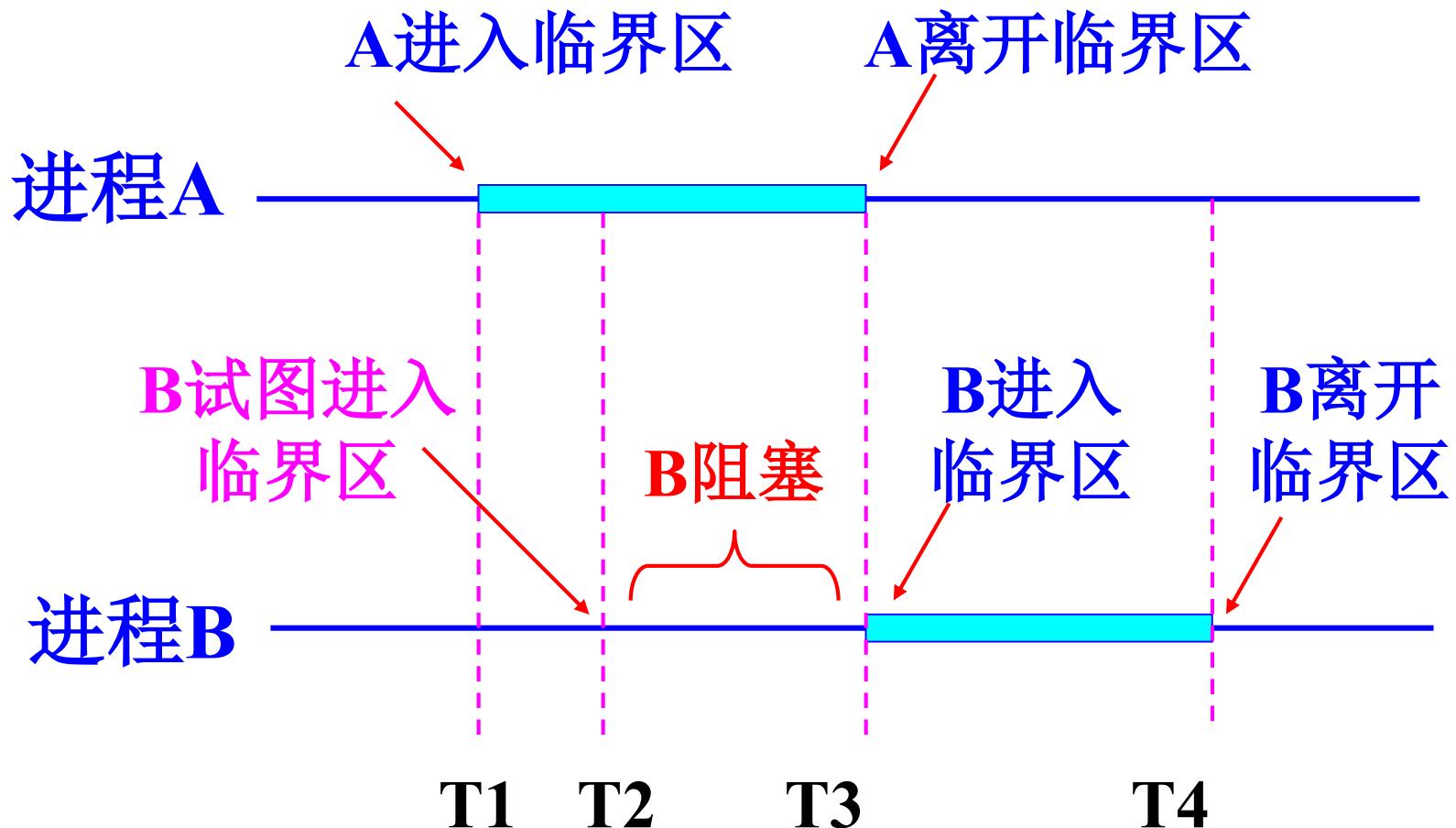
- **共享资源**: ①慢速的硬设备, 如打印机;
②软件资源, 如共享变量、共享文件等。
- **临界资源**: 就是一次仅允许一个进程使用的资源。
- **临界区(critical section)**: 就是并发进程访问临界资源的那段必须互斥执行的程序。



并发进程进入临界区需要遵循的 四个准则

- 不能同时有两个进程在临界区内执行
- 等待进入临界区的进程，应释放处理机后阻塞等待
- 在临界区外运行的进程不可阻止其他进程进入临界区
- 不应使要进入临界区的进程无限期等待在临界区之外

互斥使用；让权等待；有空让进；有限等待



使用临界区的互斥



解决进程之间互斥的方法

- 软件实现方法
- 硬件实现方法



临界区互斥软件实现方法

算法1：设有两个进程Pi和
Pj，共享变量

```
int turn;
```

当turn==i时，进程Pi允许
在临界区内执行。

执行顺序：Pi,Pj,Pi,Pj...

Pi:

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



算法2：设有两个进程Pi和Pj，共享变量

boolean flag[2];

初值，flag[i]=flag[j]=false

如果flag[i]==true，则该值
表示Pi准备进入临界区。

死锁？

Pi:

do{

flag[i]=true;

while(flag[j]);

临界区

flag[i]=false;

剩余区

}while(1);



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

算法3：

设有两个进程Pi和Pj，共
享两个变量：

```
boolean flag[2];  
int turn;
```

初值，flag[i]=flag[j]=false
turn为i或j都行。

为了进入临界区，进程Pi首先设置flag[i]为true，设
置turn为j，从而表示如果另一个想进入临界区，那么
它能进入。如果两个进程同时试图进入，那么turn会
几乎同时设置成i和j，但只有一个赋值语句的结果会
保持，最终turn值决定了谁能进入临界区。



- 复杂：存在进程间的共享资源。
- 忙等待：消耗处理器的计算时间。



用硬件实现互斥的方法

(1) 关中断 最简单的方法。在进程刚进入临界区后，立即禁止所有中断；在进程要离开之前再打开中断。因为CPU只有在发生时钟中断或其它中断时才会进行进程切换。

关中断(disableInterrupt)

〈critical section〉

开中断(enableInterrupt)



- 优点：简单。
- 缺点：限制了处理机交叉执行程序的能力
 - 把禁止中断的权力交给用户进程是不明智的。
(若用户进程禁止中断之后不再打开中断，其结果将会如何？整个系统可能会因此终止)。
 - 若是多处理机系统，则禁止中断仅仅对执行本指令的那个CPU有效。其他CPU仍将继续运行，并可以访问临界资源。



(2) 使用测试和设置硬件指令

- 锁位变量W：为每个临界资源设置一个，以指示其当前状态。 $W=0$ ，表示资源空闲可用； $W=1$ ，表示资源已被占用。
- testset指令可定义如下：

```
boolean testset(int w){  
    if(w==0){w=1; return TRUE;}  
    else{ return FALSE;}  
}//一条机器指令，其执行不可被中断。
```



```
Const int n=/*进程数 */  
int w;  
void p(int i){  
    while(1){  
        while(!testset(w));  
        <critical section>  
        w=0;  
        <remainder section>  
    } } 
```

```
void main(){  
    w=0;  
    p(1);p(2);...;p(n); } 
```

显然，采用这种加锁语句，由于进程循环测试，白白浪费了CPU的时间。这种现象又叫做“忙等”。



3.2.2 进程之间的同步

同步的原因：一组进程要合作完成一项任务。

[例]两个用户进程共享缓冲区。计算进程将计算结果送入共享缓冲区，打印进程从缓冲区取数据打印。
缓冲区空时不取数据，满时不送数据。

- 由于计算进程与打印进程访问缓冲区的速度不匹配，需要进行同步处理。
- 为了使进程同步，需要引入信号量机制。



3.2.3 信号量和P、V操作

- 1965年，荷兰学者Dijkstra提出的一种同步机制。
- 基本原理：两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位
置停止，直到它接收到一个特定的信号。为了发信号，需要使用一个称作信号量的特殊变量。
- 信号量可协调对共享资源的有效访问。



信号量表示系统共享资源的物理实体

```
typedef struct{ //信号量的类型描述
    int value; //表示该类资源的可用数量
    struct process *list; //等待使用该类资源
    的进程排成队列的队列头指针。
}semaphore, sem;
```

- ❖ 对信号量S的操作只允许执行P、V原语操作



P操作原语： //wait(s) ;

void P (sem &s)

{ **s.value = s.value-1;** //表示申请一个资源（或通过
信号量s接收消息）

if (s.value < 0)

{ **add this process to s.list;**

block();

} //资源用完，调用阻塞原语。“让权等待”

}



V操作原语： // signal(s);

Void V (sem &s) {

 s.value = s.value+1;

 //释放一个资源（或通过信号量s发消息）

 if (s.value <= 0) {

 remove a process P from s.list;

 wakeup();

 }//表示在信号链表中，仍有等待该资源的进程被阻塞。 调用唤醒原语。

}



- 显然，P、V操作的引入，克服了加锁操作的忙等待现象，提高了系统的效率。
- 操作系统正是利用信号量的状态来对进程和资源进行管理的。
- 信号量上的操作：初始化、P操作和V操作。



- 根据用途不同，可以把信号量分为公用信号量和私用信号量。
- 公用信号量（互斥信号量）用于解决进程之间互斥进入临界区。
- 私用信号量（同步信号量）用于解决异步环境下进程之间的同步。



利用信号量实现进程之间的互斥

- 确实临界区。
- 设置一个互斥信号量mutex，初值为1，表示该临界资源空闲。
- 调用P(mutex)申请临界资源。
- 调用V(mutex)释放临界资源。
- 只需把临界区代码置于P(mutex)和 V(mutex)之间，就可实现临界资源的互斥使用了。



- P操作和V操作成对出现。
- 注意P、V操作的顺序。

P1

P2

P(mutex);

临界区

V(mutex);

,

P(mutex);

临界区

V(mutex);

.....



```
int mutex=1;
```

P1:

...

P(mutex);

critical section

V(mutex);

...

P2:

...

P(mutex);

临界区

V(mutex);

...



- 用信号量可以方便地解决n个进程互斥地执行临界区代码的问题。
- 信号量的取值范围: $+1 \sim -(n-1)$ 。
- 信号量值为负时, 说明有一个进程正在临界区执行, 其它的正排在信号量等待队列中等待, 等待的进程数等于信号量值的绝对值。

[例]若P、V操作的信号量初值为1, 当前值为-3, 则表示有3个等待进程。

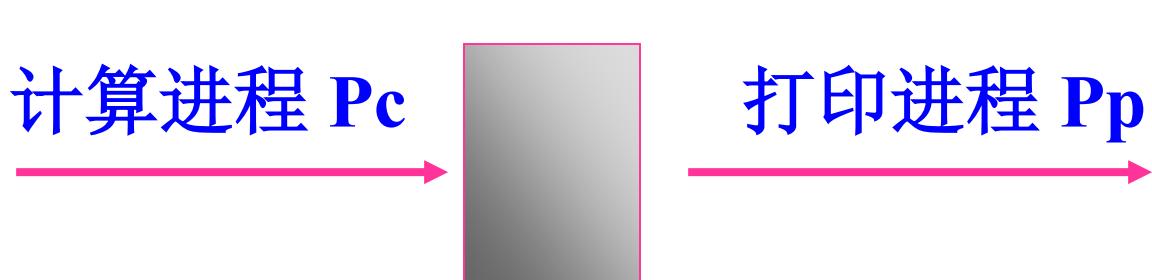


利用信号量实现 进程之间的同步



[例] 用信号量实现计算进程与打印进程之间的同步过程。假定计算进程和打印进程共享一个单缓冲区。为此，引入两个同步信号量s1和s2。

- S1：表示缓冲区是否空，初值为1；
- S2：表示缓冲区中是否有可供打印的计算结果，初值为0。





int s1=1, s2=0;

Pc: ...

computer next data;

P(s1);

add the data to buffer;

V(s2);

...

Pp: ...

P(s2);

take next data from buffer;

V(s1);

print the data;

...

能否用一个同步信号量？



例：医生为某病人诊病，认为需要作某些化验，于是，就为病人开了化验单。病人取样送到化验室，等待化验完毕交回化验结果，然后继续诊病。

分析：有两个进程各自独立地活动，一个是医生为病人诊病，另一个化验室的化验工作。

上述两个合作进程之间的同步关系为：

化验进程只有在收到诊病进程的化验单后才开始工作；而诊病进程只有获得化验结果后才能继续为该病人诊病，并根据化验结果确定医疗方案。



该进程同步，可通过设置两个同步信号量S1、
S2，以及P、V操作
来实现。

其算法的描述为：

```
程序 task3
Main ()
{
    int S1=0; /*表示无化验单*/
    int S2=0; /*表示无化验结果
*/
    parbegin
        labora () ;
        diagnosis () ;
    parend
}
```



Labora ()

```
{  
    while (化验工作未完成)  
    {  
        P (s1) ; /*询问有无化验单，  
                   若无则等*/  
        化验工作；  
        V (s2) ; /*送出化验结果*/  
    }  
}
```

Diagnosis ()

```
{  
    while (看病工作未完成)  
    {  
        看病；  
        V (s1) ; /*送出化验单*/  
        P (s2) ; /*等化验结果*/  
        Diagnosis () ; /*诊断*/  
    }  
}
```



通常同步问题分为两类：

- a. 保证一组合作进程按逻辑需要所确定的次序执行；**
- b. 保证共享缓冲区的合作进程的同步。**



生产者和消费者问题

- 生产者和消费者是相互合作进程关系的一种抽象。
- 生产者：当进程释放一个资源时，可把它看成是该资源的生产者，
- 消费者：当进程申请使用一个资源时，可把它看成该资源的消费者。

[例] 上述例子中

- 计算进程：打印数据的生产者；空缓冲的消费者
- 打印进程：打印数据的消费者；空缓冲的生产者

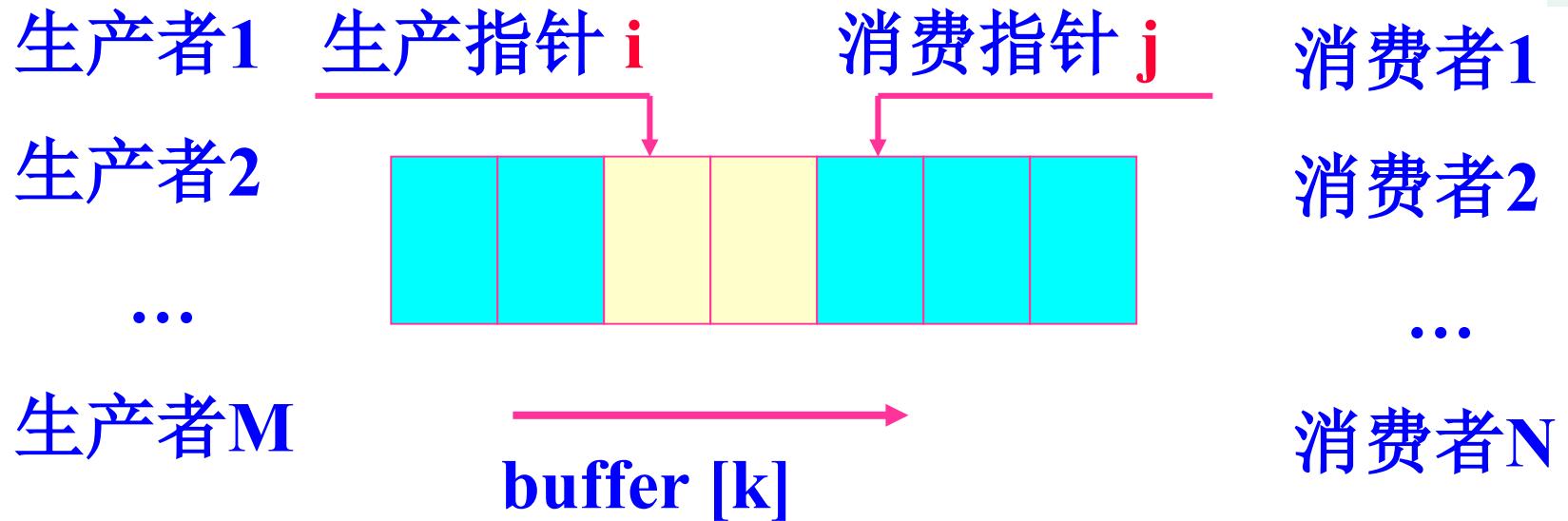


[例]

- 假定有一组生产者和消费者进程，通过一个**有界环形缓冲区**（有 k 个缓冲区）发生联系。生产者向缓冲区放产品，消费者从中取产品。
- 当缓冲区满时，生产者要**等**消费者取走产品后才能向缓冲区放下一个产品；当缓冲区空时，消费者要**等**生产者放一个产品入缓冲区后才能从缓冲区取一个产品。
- 这个环形缓冲区是一个临界资源。互斥使用(只有一个生产者或消费者操作缓冲区)。



- **互斥关系:** 只能有一个进程工作在缓冲区。
- **同步关系1:** 当缓冲区满时，生产者要等消费者取走产品后才能向缓冲区放下一个产品。
- **同步关系2:** 当缓冲区空时，消费者要等生产者放一个产品入缓冲区后才能从缓冲区取一个产品。



- empty: 表示空缓冲区的个数, 初值为k
- full: 有数据的缓冲区个数, 初值为0
- mutex: 互斥访问临界区的信号量, 初值为1



```
int mutex=1, empty=k, full=0, i=0, j=0;
```

```
DataType array[k];
```

Producer:

...

```
produce a product x;
```

```
P(empty); //申请一个空缓冲
```

```
P(mutex); //申请进入缓冲区
```

```
array[i] = x; //放入产品
```

```
i = (i+1)mod k;
```

```
V(full); //有数据的缓冲区个数加1
```

```
V(mutex); //退出缓冲区
```

...



Consumer:

...

```
P(full);      //申请一个产品  
P(mutex);     //申请进入缓冲区  
y = array[j]; //取产品  
j = (j+1)mod k;  
V(empty);    //释放1个空缓冲  
V(mutex);     //退出缓冲区
```

...

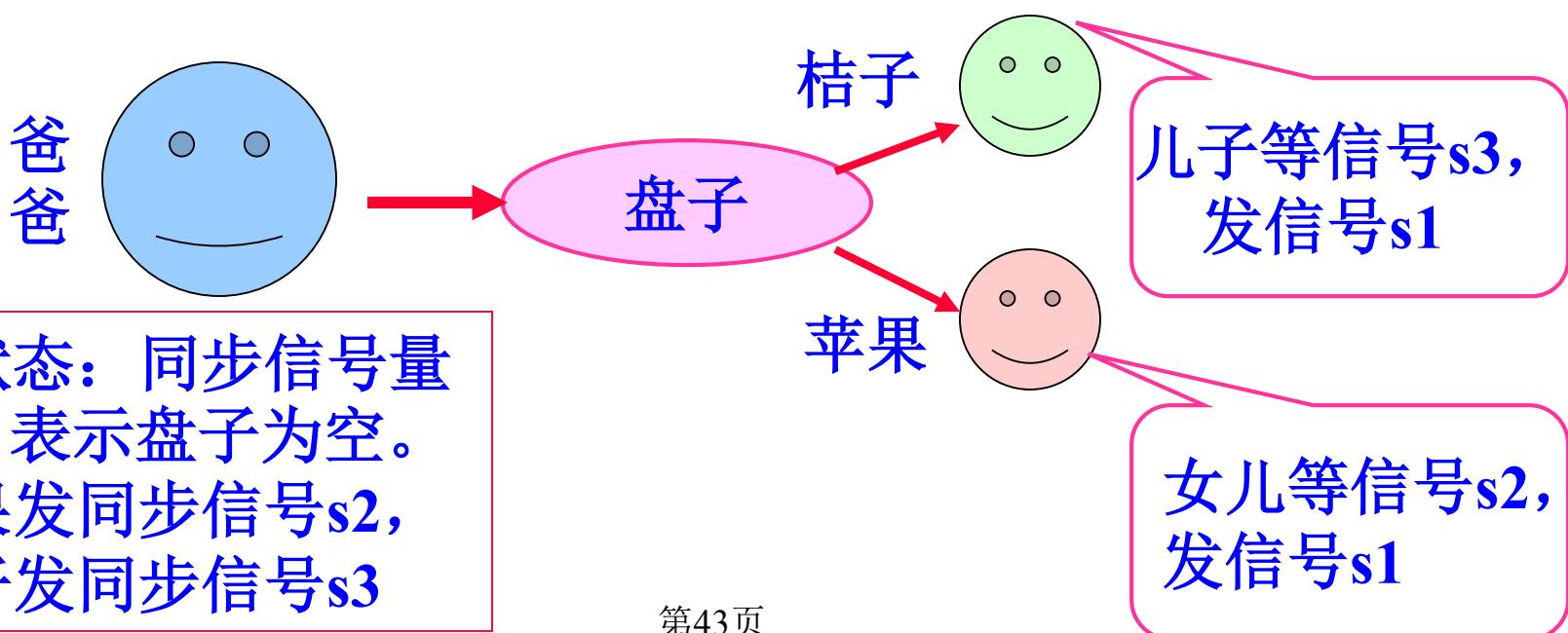


注意P操作的次序

- 若生产者进程中的两个P操作的次序交换。
 - 当缓冲区满时，生成者将在P(empty)上等待，但不释放对缓冲区的互斥使用权。
 - 此后，消费者欲取产品时，由于申请使用缓冲区不成功，它将在P(mutex)上等待。
 - 相互等待就会造成系统发生死锁现象。

[例]

桌上有一空盘，一次只允许放一只水果。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。请用P、V操作实现爸爸、儿子、女儿三个并发进程的同步。



初始状态：同步信号量
 $s1=1$ ，表示盘子为空。
放苹果发同步信号s2，
放桔子发同步信号s3



Father:

```
while(1) {  
    p(s1);  
    if(放入的是苹果)v(s2);  
    else v(s3);  
}
```

Daughter:

```
while(1) {  
    p(s2);  
    从盘中取出苹果;  
    v(s1);  
}
```

Son:

```
while(1) {  
    p(s3);  
    从盘中取出桔子;  
    v(s1);  
}
```



读者和写者问题

读/写问题：有一个多进程共享的数据区，这个数据区可以是一个文件或者主存的一块空间。有一些只读取这个数据区的进程（reader）和一些只往数据区中写数据的进程（writer）。此外还必须满足以下条件：

1. 任意多的读进程可以同时读这个数据区；
2. 一次只有一个写进程可以往数据区中写；
3. 若一个写进程正在写，禁止任何进程读。



信号量的设置

- 写互斥信号量wmutex：实现读写互斥和写写互斥地访问共享文件，初值为1。
- 计数器readcount：记录同时读的读者数，初值为0。
- 读互斥信号量rmutex：使读者互斥地访问共享变量readcount，初值为1。



```
int rmutex=1,wmutex=1,readcount=0;
```

Reader:

```
{ P(rmutex); //互斥访问readcount  
if readcount=0 then P(wmutex);
```

```
    readcount++;
```

```
    V(rmutex);
```

读文件；

```
    P(rmutex);
```

```
    readcount= readcount-1;
```

```
    if readcount=0 then V(wmutex);
```

```
    V(rmutex);
```

若规定仅允许
5个进程同时
读，怎样修改
程序？



Writer: ...

P(wmutex);

写文件；

V(wmutex);

...

[解析]

- 允许多个读进程同时读。当没有读进程正在读时，第一个试图读的读进程需要通过**P(wmutex)**实现读写互斥；当至少已经有一个读进程在读时，随后的读进程无需等待，可以直接进入。



理发师问题

- 有一个理发师、一把理发椅和n把供等候理发的顾客坐的椅子。如果没有顾客，则理发师坐在椅子上睡觉，当有一个顾客到来时，必须唤醒理发师，请求理发；如果理发师正在理发，又有顾客到来时，只要有空椅子，他就坐下来等待，如果没有空椅子，他就离开。请为理发师和顾客各编写一段程序来描述他们的同步问题。（3-18）



理发师问题

- 设两个信号量: (1)用s1制约理发师, 初值为0, 表示有0个顾客; (2)用s2制约顾客, 表示可用椅子数, 初值为n。

顾客:

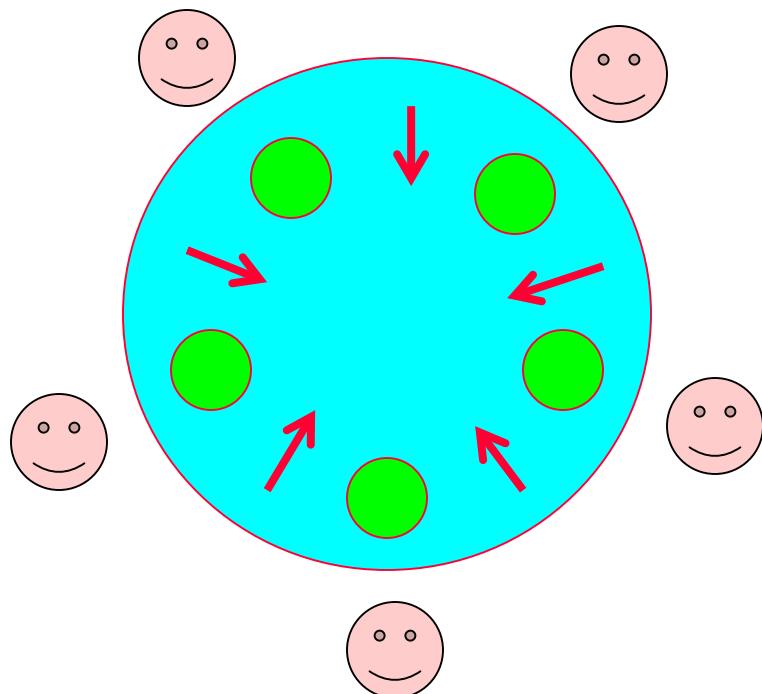
$P(s2)$; 申请椅子
 $V(s1)$; 发信号
坐椅子等理发;

理发师:

$P(s1)$; 查是否有顾客
给一名顾客理发;
 $V(s2)$; 让顾客离开

哲学家进餐问题

- 这是一个典型的同步问题，是一大类并发控制问题的例子。
- 假设有5个哲学家，花费一生的时光思考和吃饭。在桌子上放着5把叉子。一个哲学家要分两次去取其左边和右边的叉子。若得到两把叉子，就开始吃饭；吃完放下两把叉子。





哲学家进餐问题

```
int fork[0]=fork[1]=...=fork[4]=1;
```

第*i*个哲学家所执行的程序：

```
do{
```

```
{ P(mutex);  
  P(fork[i]);  
  P(fork[(i+1)mod5]);  
  V(mutex);
```

吃饭

```
  V(fork[i]);  
  V(fork[(i+1)mod5]);
```

```
} while(1);
```



信息量使用的局限性

- 代码开发难度大。
- 容易出错（如死锁问题）。



3.3 管程

- 管程是管理进程间同步的机制，它保证进程互斥地访问共享变量，并且提供了一个方便的阻塞和唤醒进程的机构。
- 管程比信号量好控制。



管程的定义

- **基本思想：**是将共享变量及对共享变量能够进行的所有操作集中在一个模块中。
- **管程**是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块。
- **管程保证：**一次只有一个进程执行管程中的代码。从而提供互斥机制，保证管程数据的一致性。



管程的组成:

Monitor monitor-name {

..... 局部于该管程的共享变量的说明

define; 本管程内定义的过程名

use; 引用的外部模块的说明

..... 本管程内的定义的各过程（函数体）

..... 为本管程内的共享变量赋初值

}



临界资源的互斥使用

```
Monitor mutexshow {  
    boolean busy=false; //临界资源是否可用标志  
    condition nonbusy; //等待队列的条件变量  
    define request, release; //管程中的过程说明  
    use wait, signal; //引用外部模块  
}
```

调用wait()的进程会阻塞在条件变量nonbusy的等待队列上。调用signal()会启动一个阻塞进程，若无阻塞进程则signal()不起作用。



procedure request() //申请临界资源的过程

{

if busy then wait(nonbusy);
 busy=true;

}

资源忙则在nonbusy
等待队列上等待，
并立即退出该管程。
申请成功，置资源
已经占用标志

procedure release() //释放临界资源的过程

{

busy=false;
 signal(nonbusy);

}

设置资源已经空闲标志
唤醒nonbusy上的等待者



生产者和消费者问题

Monitor prod_conshow

{ char buffer[n] ; 环形缓冲区

int k=0; 缓冲区中的产品个数

int nextempty=0, nextfull=0; 送/取产品的指针

condition nonempty,nonfull;

define put, get; 管程中定义的过程说明

use wait(), signal(); 引用外部模块的过程说明

}



```
procedure put(product) { //向缓冲区送产品  
    if k=n then wait(nonfull); //缓冲区满等待  
    buffer[nextempty] = product;  
    k=k+1;  
    nextempty=(nextempty+1)mod n;  
    signal(nonempty); //唤醒等待取产品消费者  
}
```



```
procedure get(product) {      //从缓冲区取产品
    if k=0 then wait(nonempty); //缓冲区空消费者
    等待
    goods = buffer[nextfull];
    k=k-1;
    nextfull = (nextfull+1) mod n;
    signal(nonfull); //唤醒等待送产品的生产者
}
```



producer:

```
{ char item;  
    produce an item;  
    prod_conshow.put(item);  
}
```

生产者
进程

互斥执行
管程中的
代码

consumer:

```
{ char item;  
    prod_conshow.get(item);  
    consume an item;  
}
```

消费者
进程



局限性

- 管程是编程语言的组成部分，编译器知道它们的特殊性，因此可以采用与其他过程调用不同的方法来处理对管程的调用，对其操作做出互斥安排。
- 进程只能通过管程中的过程来访问管程中的数据。
- C、Pascal以及多数其他语言都没有管程。
- Java有管程。



3.4 进程的高级通信

低级通信的优点：速度快。

低级通信的缺点：

1. 传送信息量小且效率低。每次通信只能传递一个单位的信息量。
2. P、V操作使用不当时，易导致死锁。
3. 当程序非正常撤离时，查找只做了P操作而未做V操作的进程是很困难的。



- **高级通信：**是指进程采用系统提供的多种通信方式来实现通信。如消息缓冲、信箱、管道、共享主存区等。
- **发送进程和接收进程的消息通信方式：**
 - 非阻塞发送，阻塞接收
 - 非阻塞发送，非阻塞接收
 - 阻塞发送，阻塞接收



消息通信方式

① 非阻塞发送，阻塞接收

发送进程发完消息后继续前进。接收进程阻塞等待接收消息。

[例] 服务器上的多个服务进程平时处于阻塞状态，一旦有客户请求服务的消息到达，系统便唤醒相应的服务进程，去完成用户所需要的服务。



消息通信方式

② 非阻塞发送，非阻塞接收

发送进程发完消息后继续前进。接收进程有消息则接收，无消息则继续前进。

这是目前多用户系统广泛采用的信件通信方式。

③ 阻塞发送，阻塞接收

发送进程发完消息后，阻塞等待接收进程的回答消息。接收进程阻塞等待接收消息，并向发送进程发送回答消息。双向通信。



3.4.1 消息缓冲通信

实现方法：

- 系统设置一个消息缓冲池，其中每个缓冲区可以存放一个消息。
- 每当进程欲发送消息时，向系统申请一个缓冲区，将消息存入缓冲区，然后把该缓冲区链接到接收进程的消息队列上。
- 消息队列通常放在接收进程的进程控制块中。
- 属于直接通信方式。



(1) 消息缓冲区的类型

```
struct message_buffer{  
    xx  sender;          /* 发送进程标识符 */  
    xx  size;            /* 消息长度 */  
    xx  text;             /*消息正文 */  
    struct message_buffer *next;  
    /*指向下一个消息缓冲区的指针 */  
}
```



(2) PCB中有关通信的数据项描述

```
struct PCB {  
    ...  
    mq;          //消息队列队首指针  
    mutex;       //消息队列互斥信号量  
    sm;          //消息队列同步信号量  
    ...  
}
```

消息队列通常放在进程控制块中。

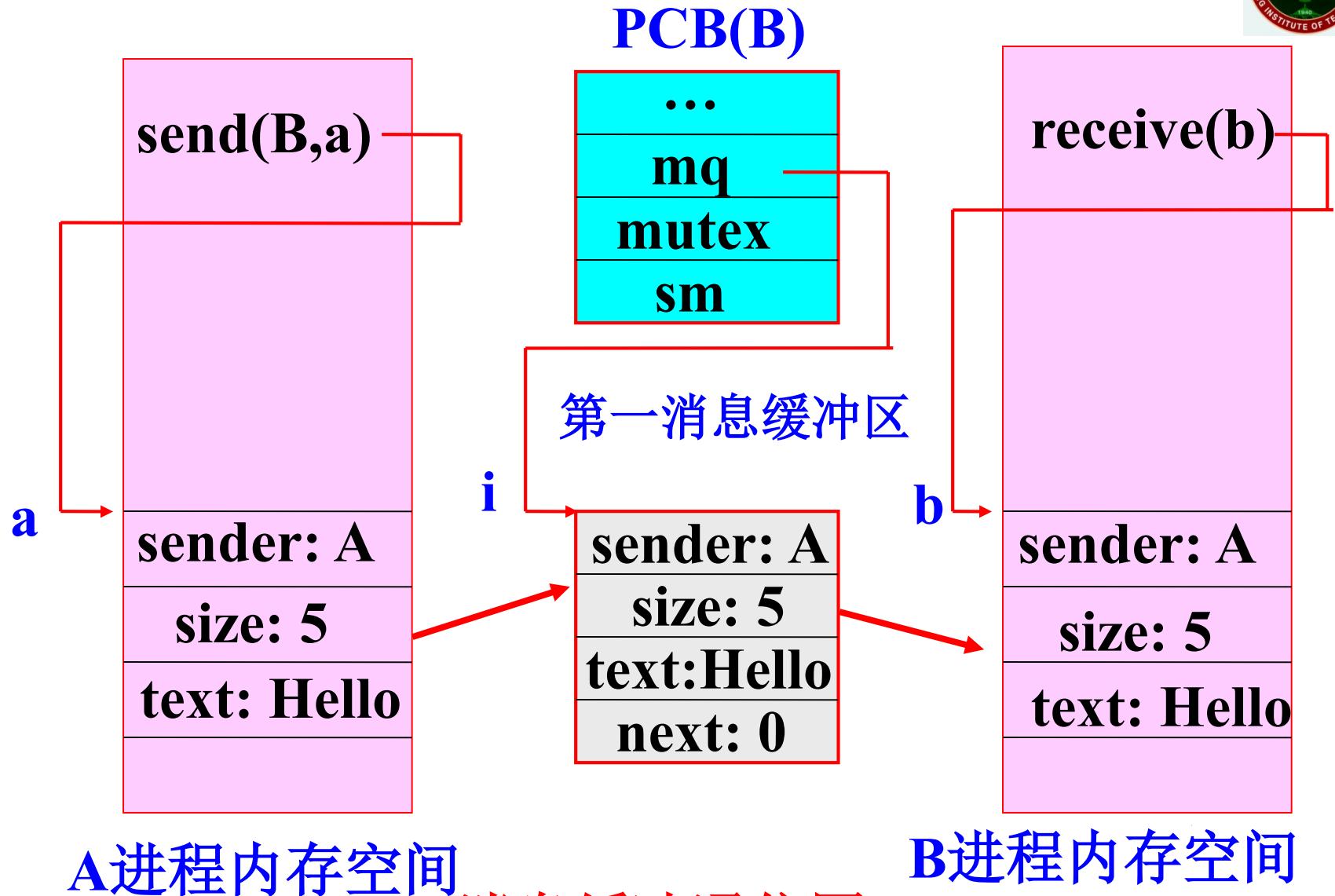


(3) 发送原语、接收原语

send (接收者, 被发送消息始址)

receive (发送者, 接收区始址)

- **发送者**先在自己的地址空间形成一个**消息发送区**, 将消息写入其中, 然后调用发送原语。
- **发送原语**: 从系统缓冲区申请一个**消息缓冲区**, 将消息从发送区传入其中, 然后挂到接收进程的消息队列上。
- **接收原语**: 将消息接收到自己的**接收区**。





```
send(receiver, a){ //发送原语  
    getbuf(a.size, i);  
    //据a区消息长度来申请一缓冲区i  
    i.sender=a.sender;    i.size=a.size;  
    i.text=a.text;    i.next=0;  
    getid(PCB set, receiver, j);  
    //获得接收进程的内部标识符j  
    P(j.mutex);  
    insert(j.mq, i); //将i挂在接收进程j的消息队列mq上  
    // (属于临界资源)。  
    V(j.mutex);  
    V(j.sm); //消息队列同步信号量sm  
}
```



```
receive(b){ //接收原语
    j=get caller's internal name; //内部标识符
    P(j.sm); //等消息
    P(j. mutex);
    remove(j.mq, i); //从自己的消息缓冲队列mq中摘
    下第一个消息缓冲区i。
    V(j.mutex);
    b.sender=i.sender;
    b.size=i.size;
    b.text=i.text;
}
```



信箱通信原语

- 发送进程将消息发送到中间媒介——信箱，接收进程从中取得消息。
- 间接通信方式

发送原语: `send(A, Msg)`

将一个消息Msg发送到信箱A

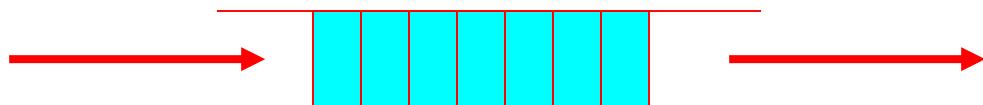
接收原语: `receive(A, Msg)`

从信箱A中接收一个消息Msg



3.4.2 其他通信机制

(1) 管道通信



- 是指用于连接一个读进程和一个写进程的共享文件，又称pipe文件。
- 是通过操作系统管理的核心缓冲区。先进先出。
- 命令方式的管道通信；程序方式的管道通信。

(2) 共享存储区（最快捷）

诸进程为了相互交换大量数据，在主存中划出一块共享存储区，并将共享存储区连到各自的地址空间，通过读或写共享存储区中的数据来实现通信。



3.5 死锁

在计算机系统中有很多独占性的资源，在任一时刻只能被一个进程使用。如打印机、磁带机。

[例]一个计算机系统，有4台磁带机和2个并发执行的进程。某一时刻，每一进程都已占有2台磁带机，还要再请求一台才能完成它们的任务。这时，由于再无空闲的磁带机，两个进程就处于永远的等待状态，系统产生了死锁。



3.5.1 死锁的定义和产生的必要条件

1. 资源的特性

- 可抢占资源：当资源从占用进程剥夺走时，对进程不产生什么破坏性的影响。如主存、CPU。
- 不可抢占资源（临界资源）：一旦分配，不能强收回，只能由其自动释放。如打印机、磁带机。
- 死锁涉及的是不可抢占资源。



进程使用资源的顺序：
请求资源， 使用资源， 释放资源。

2. 死锁的定义

一组进程是死锁的，是指这一组中的每个进程都正在等待该组中的其他进程所占用的资源时，可能引起的一种错误现象。



3. 死锁产生的必要条件

- 1) 互斥条件。独占性的资源。
- 2) 保持和等待条件。进程因请求资源而阻塞时，对已经获得的资源保持不放。
- 3) 不剥夺条件。已分配给进程的资源不能被剥夺，只能由进程自己释放。
- 4) 循环等待条件。存在一个进程循环链，链中每个进程都在等待链中的下一个进程所占用的资源。



4. 死锁产生的原因

- 系统资源配置不足，引起进程竞争资源。
- 并发进程请求资源的随机性，包括所请求资源的类别和数量。
- 各并发进程在系统中异步向前推进，造成进程推进顺序的不合理性。

产生死锁的根本原因：是对独占资源的共享，并发执行进程的同步关系不当。



3.5.2 解决死锁的方法

- ① 鸵鸟算法。忽略死锁。
- ② 死锁的预防。通过破坏产生死锁的四个必要条件中的一个或几个，来防止发生死锁。
- ③ 死锁的避免。是在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免发生死锁。
- ④ 死锁的检测和恢复。允许死锁发生，通过设置检测机构，及时检测出死锁的发生，然后采取适当措施清除死锁。



1. 鸵鸟算法

- 最简单的解决方法是鸵鸟算法：把头埋进沙子里，假装毫无问题。
- 如果死锁平均每5年发生一次，而每个月系统都会因硬件故障、编译器错误或者操作系统故障而崩溃一次，那么大多数工程师不会花费代价去防止系统死锁。
- **UNIX, Linux, Windows都参用该策略。**



[例]

UNIX系统允许创建的进程总数是由**进程表**的项数决定的。如果进程表中已经无空闲的PCB，则创建子进程操作(**fork**)失败，只能等待一段时间之后再试。

假定UNIX系统有100个PCB项，**10个进程**正在运行，每个需要创建12个子进程。在每个进程已经创建9个进程后，原来的10个进程和新创建的90个子进程已用完了进程表。这样，原来的10个进程现在都处于创建子进程的无限循环中——死锁。



2. 死锁的预防

- 因产生死锁需四个必要条件。若能破坏其中的一个或几个条件，则不产生死锁。
- 但到目前为止，还没有一个有效办法来预防死锁。



(1) 破坏互斥条件

资源的互斥使用条件是由资源本身性质决定的，不能破坏。

如果采用 spooling 技术，借助磁盘空间，就可以将一台独享设备改造成多台设备，以满足多个进程的共享需求。如打印机。

实际中，不是所有设备都能采用 spooling 技术的。即使采用了该技术，由于多个进程竞争磁盘空间，磁盘空间的不足，仍可能导致死锁。



(2) 破坏保持和请求条件

让进程在开始运行前，就获得所需的全部资源。若系统不能满足，则该进程等待。

属静态分配，资源利用率很低。

许多进程在开始运行之前，不能精确提出所用资源数量。



(3) 破坏非剥夺条件

当一个进程已占有某些资源，又申请新的资源而得不到满足时，则在进入阻塞状态前强行使其释放已经占有的资源。以后运行时，再重新申请。

显然也不行，因为保护进程放弃资源时的现场以及之后的恢复现场，系统要付出很高的代价。



(4) 破坏循环等待条件

将系统全部资源按类进行全局编号排序。进程对资源的请求必须按照资源的序号递增顺序进行。这样，就不会出现进程循环等待资源，预防死锁。

但找到能满足所有进程要求的资源编号是不可能的。



3. 死锁的避免

基本思想：允许进程动态地申请资源，一次申请一部分资源。系统在进行资源分配之前，先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，便将资源分配给进程；否则，进程等待。

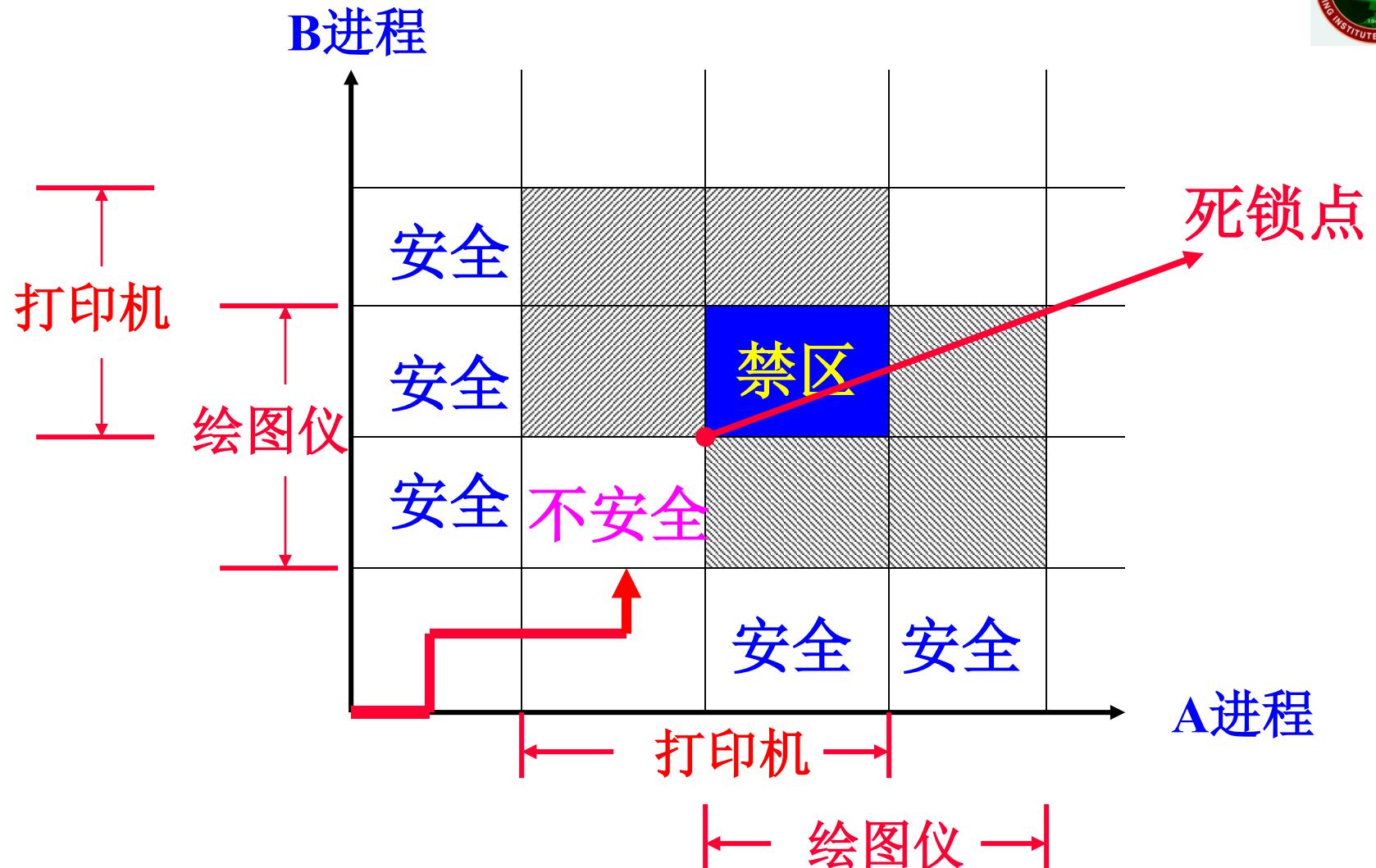
安全状态：是指系统能按某种顺序，如 $P_1, P_2, P_3, \dots, P_n$ (安全序列)，来为每个进程分配其所需资源，直至最大需求，使每个进程都可顺序完成。



(1) 进程--资源轨迹

下图中：

- 有两个进程A、B和两个资源（打印机和绘图仪）
- 水平坐标表示进程A执行的指令序列； 垂直坐标表示进程B执行的指令序列。
- 两个进程不能同时进入阴影区域



一个单处理器系统的进程资源轨迹图



(2) 银行家算法

- 最具代表性的避免死锁的算法是Dijkstra的银行家算法，是在1965年提出的。
- 利用了上面图中介绍的避免进程进入不安全区的原理。



[例]

银行家算法是用来模拟一个小城镇的银行家为一批顾客贷款的问题。

有四个顾客：A，B，C，D，每个顾客提出的最大贷款数量分别为6、5、4、7(以千美元为单位)。银行家知道不是所有顾客都马上需要其全部贷款(22)。因此，他只保留10个单位数量(而不是全部22个单位)为这些顾客服务。

在这个模型中，顾客是进程，现金是资源，而银行家就是操作系统。



| 顾 客 | 拥 有 量 | 最大 需求 |
|--------|-------------|----------|
| A | 0 | 6 |
| B | 0 | 5 |
| C | 0 | 4 |
| D | 0 | 7 |

系统拥有量:10

(a)初始状态

| 顾 客 | 拥 有 量 | 最大 需求 |
|--------|-------------|----------|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

当前剩余量:2

(b)安全状态

| 顾 客 | 拥 有 量 | 最大 需求 |
|--------|-------------|----------|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

当前剩余量:1

(c) 不安全状态

C 得到全部贷款后，很快将其全部贷款还清



银行家算法

已分配资源

剩余请求资源

| 进程 | 磁带机 | 绘图机 | 打印机 | 光盘驱动器 |
|----|-----|-----|-----|-------|
| P1 | 3 | 0 | 1 | 1 |
| P2 | 0 | 1 | 0 | 0 |
| P3 | 1 | 1 | 1 | 0 |
| P4 | 1 | 1 | 0 | 1 |
| P5 | 0 | 0 | 0 | 0 |

| 进程 | 磁带机 | 绘图机 | 打印机 | 光盘驱动器 |
|----|-----|-----|-----|-------|
| P1 | 1 | 1 | 0 | 0 |
| P2 | 0 | 1 | 1 | 2 |
| P3 | 3 | 1 | 0 | 0 |
| P4 | 0 | 0 | 1 | 0 |
| P5 | 2 | 1 | 1 | 0 |



[例]

系统拥有资源向量: $E = (6, 3, 4, 2)$

最大需求
矩阵

$$\begin{pmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} \quad 5 \times 4$$

已分配矩阵
进程i

$$\begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

j类资源

剩余请求
矩阵R

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix}$$

剩余资源向量: $A = (1 \ 0 \ 2 \ 0)$



检查一个状态是否安全

1. 检查剩余请求矩阵R是否有一行，其剩余请求向量小于等于系统剩余资源向量A。若不存在这样的行，系统将会死锁。
2. 若找到这样一行，则可以假设它获得所需的资源并运行结束，将该进程标记为终止，并将其资源加到向量A上。
3. 重复以上两步，直到所有进程都标记为终止；或者直到死锁发生，出现不安全状态。



上述例子分析

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

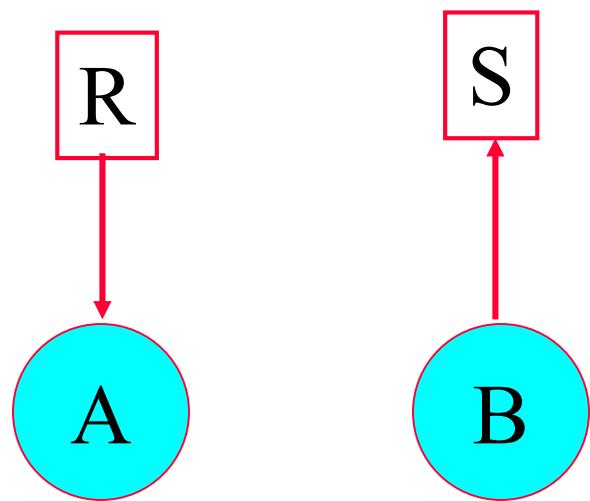
- 当前系统安全。因为根据剩余请求矩阵R，可以找到一个进程完成序列 P4, P1, P2, P3, P5。
- 之后，假定进程P2请求一个第3类资源，能否满足？
检查分配后系统是否仍处于安全状态。分配后
 $A=(1,0,1,0)$ ，还能找到一个进程完成序列 P4, P1, P5,
P2, P3，因此进程P2的请求可以满足。
- 在满足P2后，P5也请求一台打印机，不能满足。



4. 死锁的检测和恢复

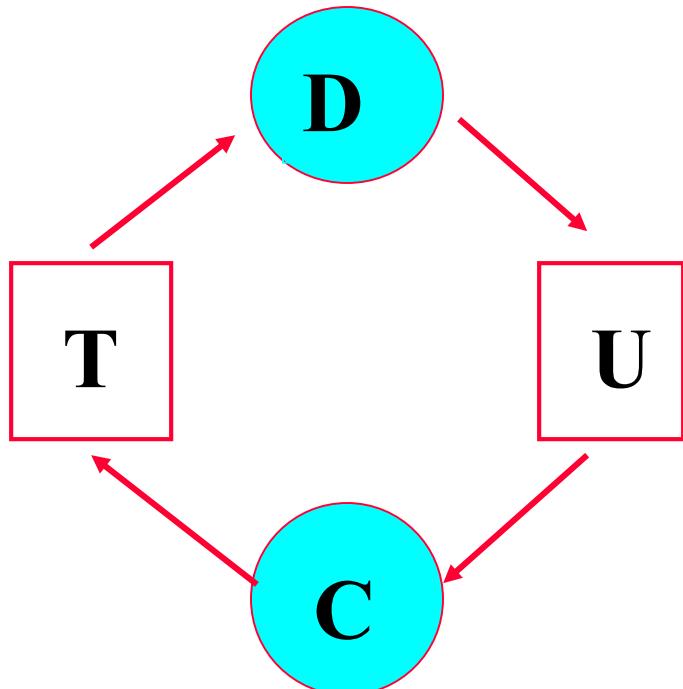
死锁的检测和恢复技术：是指定期启动一个软件检测系统的状态，若发现有死锁存在，则采取措施恢复之。

(1) 死锁的检测—用进程资源图检测死锁
检查系统中由进程和资源构成的有向图是否包含一个或多个环路，若是，则存在死锁，否则不存在。



(a) 进程A获得了
一个资源

(b) 进程B
正在请求一
个资源



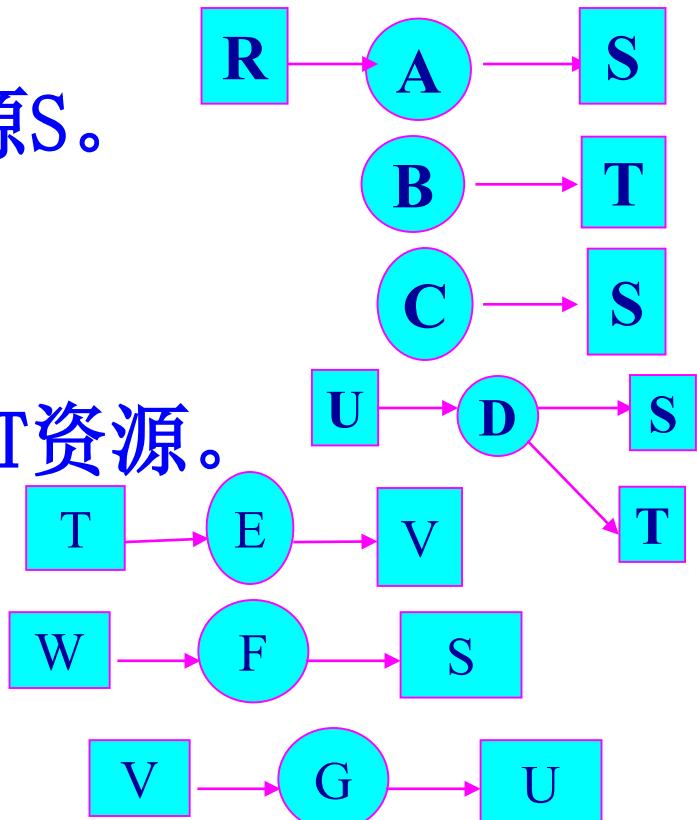
(c)进程C 和D已处
于死锁状态

圆圈代表进程，方块代表资源



[例] 假定一个系统有七个进程: A~G; 六个资源: R~W。

- 进程A保持资源R, 请求资源S。
- 进程B请求资源T。
- 进程C请求S资源。
- 进程D保持资源U, 请求S和T资源。
- 进程E保持T, 请求V资源。
- 进程F保持W, 请求S资源。
- 进程G保持V, 请求U资源。



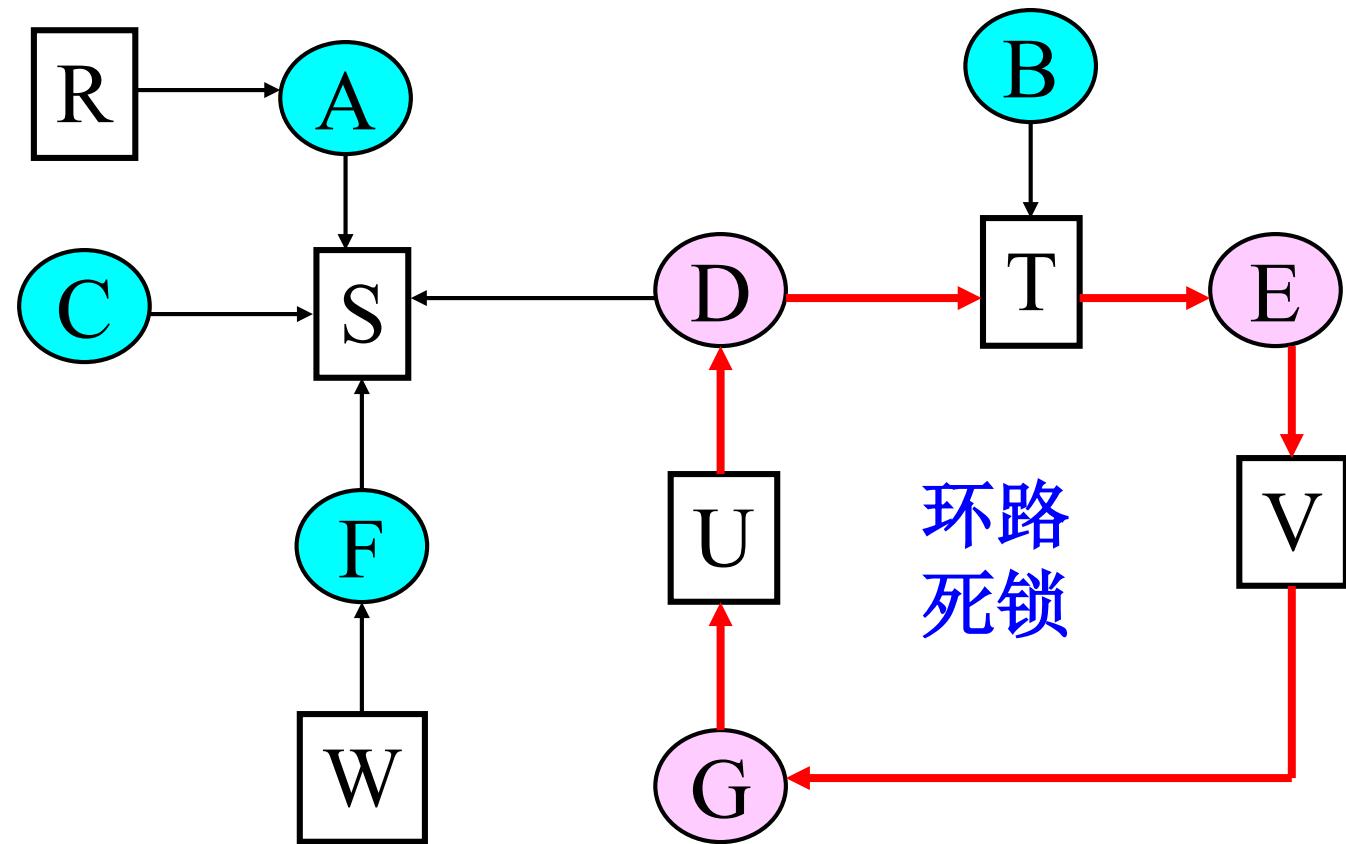


图3.13 进程资源图

存在环路: $D \rightarrow T \rightarrow E \rightarrow V \rightarrow G \rightarrow U \rightarrow D$



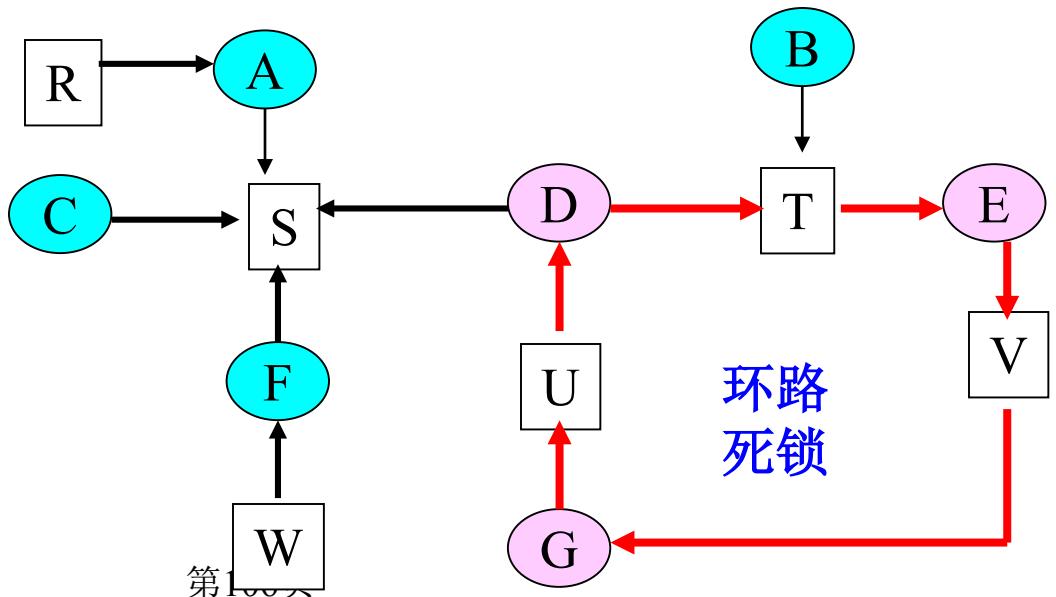
一个简单的死锁检测算法

- 需要一个数据结构L，用来记录各检查点的情况。执行过程中，标记已检测过的弧。
- 类似于有向图的遍历。
- 死锁检测算法如下：



- ① 对图中的每个节点N，以N为起始节点，执行：
- ② 将L初始化为空表，以表示所有弧都未标记过。
- ③ 将当前节点加到L的末端，检查这个节点在表中是否出现过。如果是，这个图包含一个环路，算法终止。如果没有，转④。
- ④ 由这个节点再看，是否有未标记的引出弧。如果有，转⑤；否则转⑥。
- ⑤ 任意选择一个未标记的引出弧并标记它。然后，将引出弧所到节点作为新的当前节点，转③。
- ⑥ 若所有从这个节点引出的弧都已标记，则返回到前一个节点，如果这个节点是最初开始的节点，这个图没有包含环路，算法终止；若不是初始节点，再以该节点作为当前节点，转④。

[例] 从B开始。由B跟踪引出弧一直到T，得 $L = [B, T, E, V, G, U, D, T]$ ，由表中看出，T出现两次，因此，该图包含环路，停止算法的执行。由于存在环路，故存在死锁。死锁进程为D、E、G。





(2) 死锁的恢复

- ① 故障终止一些进程
 - a. 故障终止所有死锁进程。简单。
 - b. 一次终止一个死锁进程，直到死锁解除为止。
- ② 资源剥夺：暂时从当前占有者夺走一部分资源给另一些进程，直到死锁恢复。
 - a. 从一个进程取走资源给另一个进程使用。
 - b. 将一死锁进程滚回到获得资源之前的执行点（检查点）。为进程设置检查点是指将进程在该点的执行状态信息写到一个文件中，便于以后从该检查点启动进程执行。



3.6 小结

1. 进程之间的低级通信
2. 管程
3. 进程之间的高级通信
4. 死锁