# 2.3同步与互斥

# 进程并发执行的一些问题

在 OS 中引入进程后，一方面可以使系统中的多道程序并发执行，这不仅能有效地改善资源利用率，还可显著地提高系统的吞吐量，但另一方面却使系统变得更加复杂。如果不能采取有效的措施，对多个进程的运行进行妥善的管理，必然会因为这些进程对系统资源的无序争夺给系统造成混乱。致使每次处理的结果存在着不确定性，即显现出其不可再现性。

为保证多个进程能有条不紊地运行，在多道程序系统中，必须引入进程同步机制。在本章中，将详细介绍在单处理机系统中的进程同步机制——硬件同步机制、信号量机制、管程机制等，利用它们来保证程序执行的可再现性。

概念。例如，让系统计算 1+2×3，假设系统产生两个进程：一个是加法进程，一个是乘法进程。要让计算结果是正确的，一定要让加法进程发生在乘法进程之后，但实际上操作系统具有异步性，

### 1. 临界资源

虽然多个进程可以共享系统中的各种资源，但其中许多资源一次只能为一个进程所用，我们将一次仅允许一个进程使用的资源称为临界资源。许多物理设备都属于临界资源，如打印机等。此外，还有许多变量、数据等都可以被若干进程共享，也属于临界资源。

互斥共享资源

对临界资源的访问，必须互斥地进行，在每个进程中，访问临界资源的那段代码称为临界区。为了保证临界资源的正确使用，可把临界资源的访问过程分成 4 个部分：

1）进入区。为了进入临界区使用临界资源，在进入区要检查可否进入临界区，若能进入临界区，则应设置正在访问临界区的标志，以阻止其他进程同时进入临界区。

2）临界区。进程中访问临界资源的那段代码，又称临界段。

3）退出区。将正在访问临界区的标志清除。

4）剩余区。代码中的其余部分。

```
do {
    entry section;              //进入区
    critical section;           //临界区
    exit section;               //退出区
    remainder section;          //剩余区
} while(true)
```

# 两种制约关系

1) 间接相互制约关系

多个程序在并发执行时，由于共享系统资源，如 CPU、I/O 设备等，致使在这些并发执行的程序之间形成相互制约的关系。对于像打印机、磁带机这样的临界资源，必须保证多个进程对之只能互斥地访问，由此，在这些进程间形成了源于对该类资源共享的所谓间接相互制约关系。为了保证这些进程能有序地运行，对于系统中的这类资源，必须由系统实施统一分配，即用户在要使用之前，应先提出申请，而不允许用户进程直接使用。

2) 直接相互制约关系

某些应用程序，为了完成某任务而建立了两个或多个进程。这些进程将为完成同一项任务而相互合作。进程间的直接制约关系就是源于它们之间的相互合作。例如，有两个相互合作的进程——输入进程 A 和计算进程 B，它们之间共享一个缓冲区。进程 A 通过缓冲向进程 B 提供数据。进程 B 从缓冲中取出数据，并对数据进行处理。但如果该缓冲空时，计算进程因不能获得所需数据而被阻塞。一旦进程 A 把数据输入缓冲区后便将进程 B 唤醒；反之，当缓冲区已满时，进程 A 因不能再向缓冲区投放数据而被阻塞，当进程 B 将缓冲区数据取走后便可唤醒 A。

在多道程序环境下，由于存在着上述两类相互制约关系，进程在运行过程中是否能获得处理机运行与以怎样的速度运行，并不能由进程自身所控制，此即进程的异步性。由此会产生对共享变量或数据结构等资源不正确的访问次序，从而造成进程每次执行结果的不一致。这种差错往往与时间有关，故称为“与时间有关的错误”。为了杜绝这种差错，必须对进程的执行次序进行协调，保证诸进程能按序执行。

### 4. 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

(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，因而其它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”状态。

# 实现临界区互斥的方法（软件）

1）算法一：单标志法。该算法设置一个公用整型变量 turn，用于指示被允许进入临界区的进程编号，即若 turn = 0，则允许 $P_0$ 进程进入临界区。该算法可确保每次只允许一个进程进入临界区。但两个进程必须交替进入临界区，若某个进程不再进入临界区，则另一个进程也将无法进入临界区（违背“空闲让进”）。这样很容易造成资源利用不充分。

若 $P_0$ 顺利进入临界区并从临界区离开，则此时临界区是空闲的，但 $P_1$ 并没有进入临界区的打算，turn = 1 一直成立，$P_0$ 就无法再次进入临界区（一直被 while 死循环困住）。

| $P_0$ 进程： | $P_1$ 进程： | |
|---|---|---|
| `while(turn!=0);` | `while(turn!=1);` | //进入区 |
| `critical section;` | `critical section;` | //临界区 |
| `turn=1;` | `turn=0;` | //退出区 |
| `remainder section;` | `remainder section;` | //剩余区 |

- 单标志法：设置一个公共整型变量，表示当前允许进入临界区的进程编号。所有进程按照一定顺序循环轮流使用临界资源。
- 缺点：违背空闲让进，造成资源利用不充分。

**在进入区只做“检查”，不“上锁”**

2）算法二：双标志法先检查。该算法的基本思想是在每个进程访问临界区资源之前，先查看临界资源是否正被访问，若正被访问，该进程需等待；否则，进程才进入自己的临界区。为此，设置一个数据 flag[*i*]，如第 *i* 个元素值为 FALSE，表示 $P_i$ 进程未进入临界区，值为 TRUE，表示 $P_i$ 进程进入临界区。

```
Pi进程:                                  Pj进程:
while(flag[j]);          ①               while(flag[i]);          ②   //进入区
flag[i]=TRUE;            ③               flag[j]=TRUE;            ④   //进入区
critical section;                        critical section;                //临界区

flag[i]=FALSE;                           flag[j]=FALSE;                   //退出区
remainder section;                       remainder section;               //剩余区
```

- 双标志先检查法：设置一个数组表示所有进程是否在使用临界资源，每个进程访问临界资源前都要先扫描数组查看临界资源是否被访问，若正被访问，则等待。否则修改标志位。
- 缺点：违背忙则等待，两个进程可能同时进入临界区。

**在进入区先“检查”后“上锁”，退出区“解锁”**

3）算法三：双标志法后检查。算法二先检测对方的进程状态标志，再置自己的标志，由于在检测和放置中可插入另一个进程到达时的检测操作，会造成两个进程在分别检测后同时进入临界区。为此，算法三先将自己的标志设置为 TRUE，再检测对方的状态标志，若对方标志为 TRUE，则进程等待；否则进入临界区。

| $P_i$进程： | $P_j$进程： | |
|---|---|---|
| flag[i]=TRUE; | flag[j]=TRUE; | //进入区 |
| while(flag[j]); | while(flag[i]); | //进入区 |
| critical section; | critical section; | //临界区 |
| flag[i]=FALSE; | flag[j]=FALSE; | //退出区 |
| remainder section; | remainder section; | //剩余区 |

两个进程几乎同时都想进入临界区时，它们分别将自己的标志值 flag 设置为 TRUE，并且同时检测对方的状态（执行 while 语句），发现对方也要进入临界区时，双方互相谦让，结果谁也进不了临界区，从而导致“饥饿”现象。

- 双标志后检查法：设置一个数组表示所有进程是否在使用临界资源，每个进程访问临界资源前都要先将自己的标志位设为True，再扫描数组检测其他进程的标志位。若对方为 True，则等待。
- 缺点：会导致饥饿现象，两个进程可能互相谦让，结果都无法进入临界区。（违背了空闲让进）

在进入区先“加锁”后“检查”，退出区“解锁”

4）算法四：Peterson's Algorithm。为了防止两个进程为进入临界区而无限期等待，又设置了变量 turn，每个进程在先设置自己的标志后再设置 turn 标志。这时，再同时检测另一个进程状态标志和允许进入标志，以便保证两个进程同时要求进入临界区时，只允许一个进程进入临界区。

```
Pi进程:                                  Pj进程:
flag[i]=TRUE;turn=j;                     flag[j]=TRUE;turn=i;          //进入区
while(flag[j]&&turn==j);                 while(flag[i]&&turn==i);      //进入区
critical section;                        critical section;             //临界区
flag[i]=FALSE;                           flag[j]=FALSE;                //退出区
remainder section;                       remainder section;            //剩余区
```

具体如下：考虑进程 $P_i$，一旦设置 flag[$i$] = true，就表示它想要进入临界区，同时 turn = $j$，此时若进程 $P_j$ 已在临界区中，符合进程 $P_i$ 中的 while 循环条件，则 $P_i$ 不能进入临界区。若 $P_j$ 不想要进入临界区，即 flag[$j$] = false，循环条件不符合，则 $P_i$ 可以顺利进入，反之亦然。本算法的基本思想是算法一和算法三的结合。利用 flag 解决临界资源的互斥访问，而利用 turn 解决“饥饿”现象。

- Peterson算法：和单标志法和双标志法类似的，设置一个数组和整型变量turn，每个进程先设置自己的标志，再设置turn。然后检查其他进程的标志和turn，保证只有一个进程才能进入临界区。
- 优点：解决了饥饿现象

缺点：不遵循让权等待

**在进入区“主动争取—主动谦让—检查对方是否想进、己方是否谦让”**

# 实现临界区互斥的方法（硬件）

虽然可以利用软件方法解决诸进程互斥进入临界区的问题，但有一定难度，并且存在很大的局限性，因而现在已很少采用。相应地，目前许多计算机已提供了一些特殊的硬件指令，允许对一个字中的内容进行检测和修正，或者是对两个字的内容进行交换等。可利用这些特殊的指令来解决临界区问题。实际上，在对临界区进行管理时，可以将标志看做一个锁，“锁开”进入，“锁关”等待，初始时锁是打开的。每个要进入临界区的进程必须先对锁进行测试，当锁未开时，则必须等待，直至锁被打开。反之，当锁是打开的时候，则应立即把其锁上，以阻止其它进程进入临界区。显然，为防止多个进程同时测试到锁为打开的情况，测试和关锁操作必须是连续的，不允许分开进行。

# 1.中断屏蔽方法

**1. 关中断**

关中断是实现互斥的最简单的方法之一。在进入锁测试之前关闭中断，直到完成锁测试并上锁之后才能打开中断。这样，进程在临界区执行期间，计算机系统不响应中断，从而不会引发调度，也就不会发生进程或线程切换。由此，保证了对锁的测试和关锁操作的连续性和完整性，有效地保证了互斥。但是，关中断的方法存在许多缺点：① 滥用关中断权力可能导致严重后果；② 关中断时间过长，会影响系统效率，限制了处理器交叉执行程序的能力；③ 关中断方法也不适用于多 CPU 系统，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。

```
…
关中断;
临界区;
开中断;
…
```

- 中断屏蔽方法：在进入临界区前（进入区）关闭中断，不允许进程切换，使得临界区一口气执行完毕，不会被中断 。
- 缺点：关中断限制了处理机交替执行程序的能力，影响执行效率；用户可能滥用关中断的权限。

# 2.TestAndSet指令

TestAndSet 指令：这条指令是原子操作，即执行该代码时不允许被中断。其功能是读出指定标志后把该标志设置为真。指令的功能描述如下：

```
boolean TestAndSet(boolean *lock){
    boolean old;
    old=*lock;
    *lock=true;
    return old;
}
```

可以为每个临界资源设置一个共享布尔变量 lock，表示资源的两种状态：true 表示正被占用，初值为 false。进程在进入临界区之前，利用 TestAndSet 检查标志 lock，若无进程在临界区，则其值为 false，可以进入，关闭临界资源，把 lock 置为 true，使任何进程都不能进入临界区；若有进程在临界区，则循环检查，直到进程退出。利用该指令实现互斥的过程描述如下：

```
while TestAndSet(&lock);
进程的临界区代码段;
lock=false;
进程的其他代码;
```

- TestAndSet指令：该指令是原子操作，不允许中断。使用全局变量lock表示临界资源是否被使用，利用这条指令检查变量lock的值，如果没有进程使用则修改为已被使用（检查，修改一气呵成）

Swap 指令：该指令的功能是交换两个字（字节）的内容。其功能描述如下：

```
Swap(boolean *a, boolean *b){
    boolean temp;
    Temp=*a;
    *a=*b;
    *b=temp;
}
```

注意：以上对 TestAndSet 和 Swap 指令的描述仅是功能实现，而并非软件实现的定义。事实上，它们是由硬件逻辑直接实现的，不会被中断。

用 Swap 指令可以简单有效地实现互斥，为每个临界资源设置一个共享布尔变量 lock，初值为 false；在每个进程中再设置一个局部布尔变量 key，用于与 lock 交换信息。在进入临界区前，先利用 Swap 指令交换 lock 与 key 的内容，然后检查 key 的状态；有进程在临界区时，重复交换和检查过程，直到进程退出。其处理过程描述如下：

```
key=true;
while(key!=false)
    Swap(&lock, &key);
进程的临界区代码段;
lock=false;
进程的其他代码;
```

- Swap指令：该指令是原子操作，不允许中断。使用全局变量lock表示临界资源是否被使用，通过交换局部变量和lock变量的值来判断临界资源是否被使用，并进行修改。

注意：以上对 TestAndSet 和 Swap 指令的描述仅是功能实现，而并非软件实现的定义。事实上，它们是由硬件逻辑直接实现的，不会被中断。

可以理解为底层的逻辑电路

硬件方法的优点：适用于任意数目的进程，而不管是单处理机还是多处理机；简单、容易验证其正确性。可以支持进程内有多个临界区，只需为每个临界区设立一个布尔变量。

硬件方法的缺点：进程等待进入临界区时要耗费处理机时间，不能实现让权等待。从等待进程中随机选择一个进入临界区，有的进程可能一直选不上，从而导致“饥饿”现象。

不能够及时让出处理机，即一直处于while循环中，即没有及时进入阻塞态

# 信号量

信号量机制是一种功能较强的机制，可用来解决互斥与同步问题，它只能被两个标准的原语wait(S)和 signal(S)访问，也可记为“P 操作”和“V 操作”。

原语是指完成某种功能且不被分割、不被中断执行的操作序列，通常可由硬件来实现。例如，前述的 Test-and-Set 和 Swap 指令就是由硬件实现的原子操作。原语功能的不被中断执行特性在单处理机上可由软件通过屏蔽中断方法实现。原语之所以不能被中断执行，是因为原语对变量的操作过程若被打断，可能会去运行另一个对同一变量的操作过程，从而出现临界段问题。

# 整型信号量

整型信号量被定义为一个用于表示资源数目的整型量 S，wait 和 signal 操作可描述为

```
wait(S){
    while(S<=0);
    S=S-1;
}
signal(S){
    S=S+1;
}
```

在整型信号量机制中的 wait 操作，只要信号量 $S \leq 0$，就会不断地测试。因此，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。

### 2. 记录型信号量

记录型信号量机制是一种不存在“忙等”现象的进程同步机制。除了需要一个用于代表资源数目的整型变量 value 外，再增加一个进程链表 L，用于链接所有等待该资源的进程。记录型信号量得名于采用了记录型的数据结构。记录型信号量可描述为

```
typedef struct{
    int value;
    struct process *L;
} semaphore;
```

相应的 wait(S)和 signal(S)的操作如下：

```
void wait(semaphore S){        //相当于申请资源
    S.value--;
    if(S.value<0){
        add this process to S.L;
        block(S.L);
    }
}
```

wait 操作，S.value--表示进程请求一个该类资源，当 S.value < 0 时，表示该类资源已分配完毕，因此进程应调用 block 原语，进行自我阻塞，放弃处理机，并插入该类资源的等待队列 S.L，可见该机制遵循了“让权等待”的准则。

```
void signal(semaphore S){     //相当于释放资源
    S.value ++;
    if(S.value<=0){
        remove a process P from S.L;
        wakeup(P);
    }
}
```

signal 操作，表示进程释放一个资源，使系统中可供分配的该类资源数增 1，因此有 S.value ++。若加 1 后仍是 S.value ≤ 0，则表示在 S.L 中仍有等待该资源的进程被阻塞，因此还应调用 wakeup 原语，将 S.L 中的第一个等待进程唤醒。

wait(S)和 signal(S)访问，也可记为“P 操作”和“V 操作”。

P是荷兰语proberen缩写 意思为“尝试”
V是荷兰语verhogen的缩写 意思为“增加”

# 利用信号量实现互斥

假设系统有两台打印机，内存中有四个并发执行的进程都要使用打印机资源。它们之间具有间接制约关系，也就是互斥。

- S.value标识资源的数目，则把S.value初始值设为2，即S.value=2，而对于S.L这个阻塞队列初始化为空。



```
Print(){
    ...
    P(S);
    使用打印机;
    V(S);
    ...
}
```

进程P1

```
Print(){
    ...
    P(S);
    使用打印机;
    V(S);
    ...
}
```

进程P2

```
Print(){
    ...
    P(S);
    使用打印机;
    V(S);
    ...
}
```

进程P2

```
Print(){
    ...
    P(S);
    使用打印机;
    V(S);
    ...
}
```

进程P2

对信号量 S 的一次 P 操作意味着进程请求一个单位的该类资源，因此需要执行 S.value--，表示资源数减1。
当S.value < 0 时表示该类资源已分配完毕，因此进程应调用 block 原语进行自我阻塞（当前运行的进程从运行态到阻塞态），主动放弃处理机，并插入该类资源的等待队列 S.L 中。
可见，该机制遵循了“让权等待”原则，不会出现“忙等”现象。

```
Print(){
    ...
    P(S); // 进入区 加锁
    使用打印机; // 临界区 访问临界资源
    V(S); // 退出去 解锁
    ...
}
```

对信号量 S 的一次 V 操作意味着进程释放一个单位的该类资源，因此需要执行 S.value++，表示资源数加1。
若加1后仍是 S.value <= 0，表示依然有进程在等待该类资源，因此应调用 wakeup 原语唤醒等待队列中的第一个进程（被唤醒进程从阻塞态到就绪态）。

# 利用信号量实现同步

假设有两个进程P1和P2，P1进程会向缓冲区里写入数据，而P2进程需要读入缓冲区的数据才能继续运行。因此两者之间具有直接制约关系，也就是同步 。

1. 先初始化信号量的值，初始时缓冲区为空，即资源数目为0，S.value=0，并且S.L阻塞队列初始化为空


```
semaphore S=0;
P1(){
    write data;
    V(S);
}
```

```
P2(){
    P(S);
    read data;
}
```

- 情况1 P2先执行
- 情况2 P1先执行

```
semaphore S=0;
```

等价于

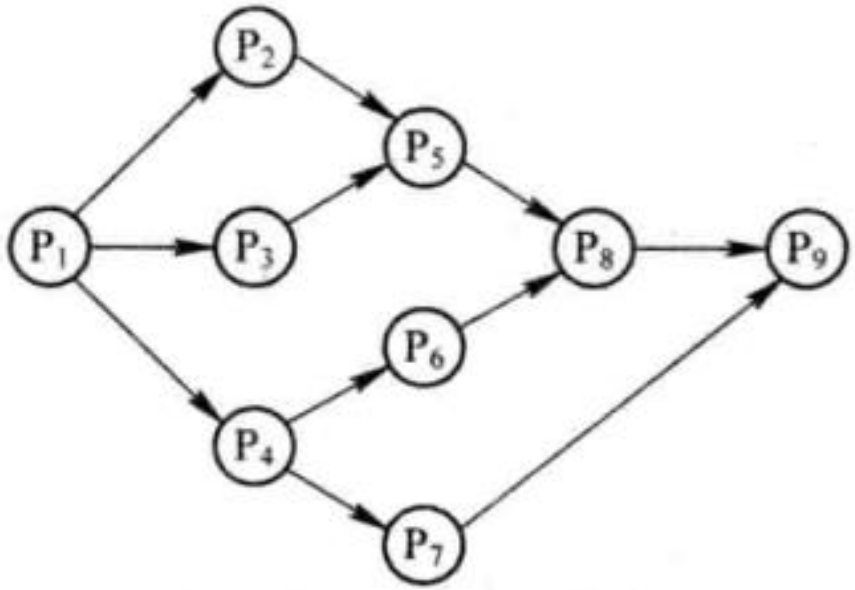
semaphore S;

S.value = 0;

### 2.1.1 前趋图

为了能更好地描述程序的顺序和并发执行情况，我们先介绍用于描述程序执行先后顺序的前趋图。所谓前趋图(Precedence Graph)，是指一个有向无循环图，可记为 DAG(Directed Acyclic Graph)，它用于描述进程之间执行的先后顺序。图中的每个结点可用来表示一个进程或程序段，乃至一条语句，结点间的有向边则表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)。

进程(或程序)之间的前趋关系可用“→”来表示，如果进程 $P_i$ 和 $P_j$ 存在着前趋关系，可表示为$(P_i, P_j) \in \rightarrow$，也可写成 $P_i \rightarrow P_j$，表示在 $P_j$ 开始执行之前 $P_i$ 必须完成。此时称 $P_i$ 是 $P_j$ 的直接前趋，而称 $P_j$ 是 $P_i$ 的直接后继。在前趋图中，把没有前趋的结点称为初始结点(Initial Node)，把没有后继的结点称为终止结点(Final Node)。此外，每个结点还具有一个重量(Weight)，用于表示该结点所含有的程序量或程序的执行时间。在图 2-1(a)所示的前趋图中，存在着如下前趋关系：




(a) 具有九个结点的前趋图


(b) 具有循环的前趋图

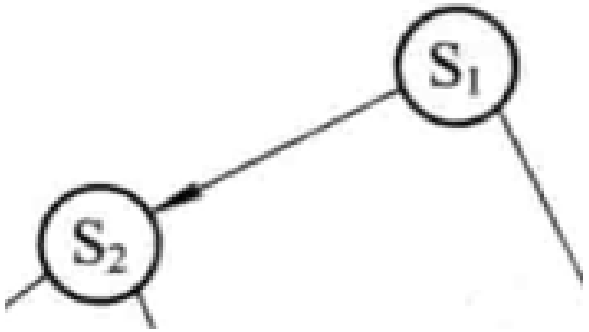
图 2-1　前趋图

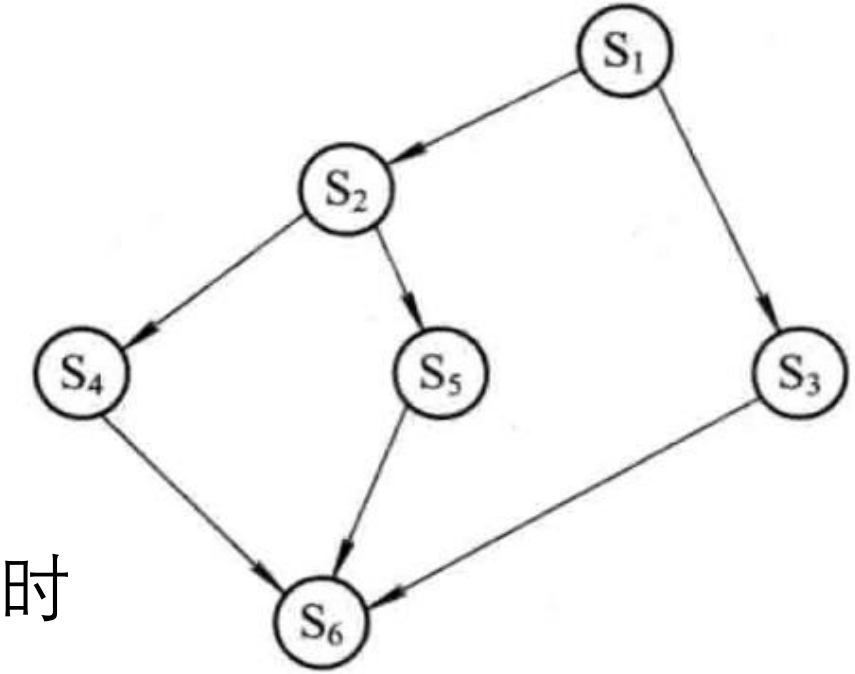
$P_1 \rightarrow P_2$，$P_1 \rightarrow P_3$，$P_1 \rightarrow P_4$，$P_2 \rightarrow P_5$，$P_3 \rightarrow P_5$，$P_4 \rightarrow P_6$，$P_4 \rightarrow P_7$，$P_5 \rightarrow P_8$，$P_6 \rightarrow P_8$，$P_7 \rightarrow P_9$，$P_8 \rightarrow P_9$

应当注意，前趋图中是不允许有循环的，否则必然会产生不可能实现的前趋关系。如图 2-1(b)所示的前趋关系中就存在着循环。它一方面要求在 S3 开始执行之前，S2 必须完成，另一方面又要求在 S2 开始执行之前，S3 必须完成。显然，这种关系是不可能实现的。

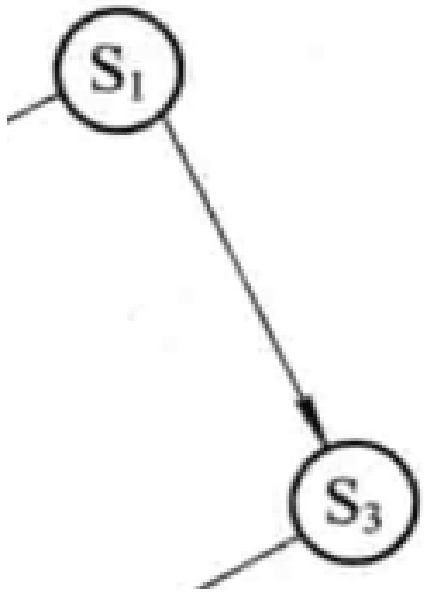
# 信号量实现前趋关系（前趋图）（前驱关系）

```
semaphore a=0;
S1(){
    ...
    ...
    V(a);
}

S2(){
    P(a);
    ...
    ...
}
```

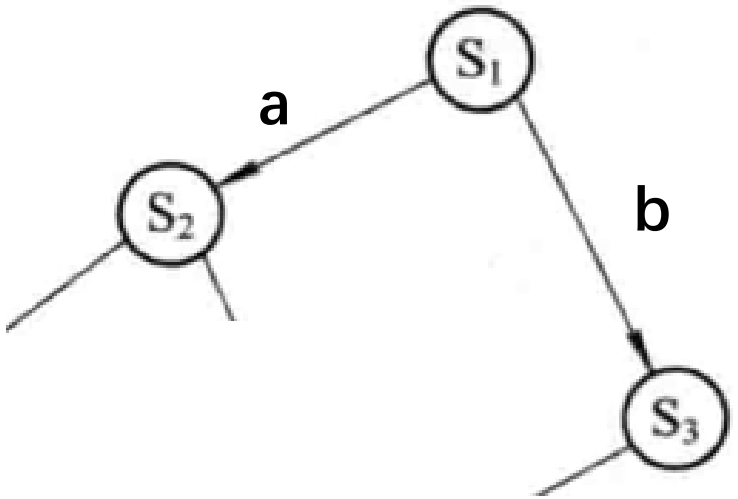



当S1()执行到V(a)这一行时
有两种情况：



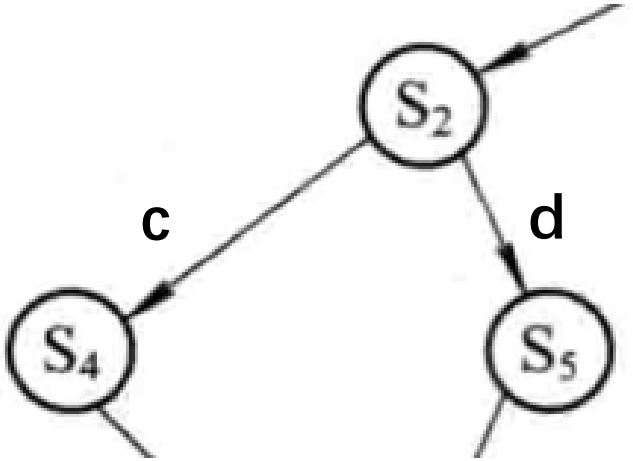

```
semaphore b=0;
S1(){
    ...
    ...
    V(b);
}

S3(){
    P(b);
    ...
    ...
}
```




```
semaphore b=0,a=0;
S1(){
    ...
    ...
    V(b);
    V(a);
}
```

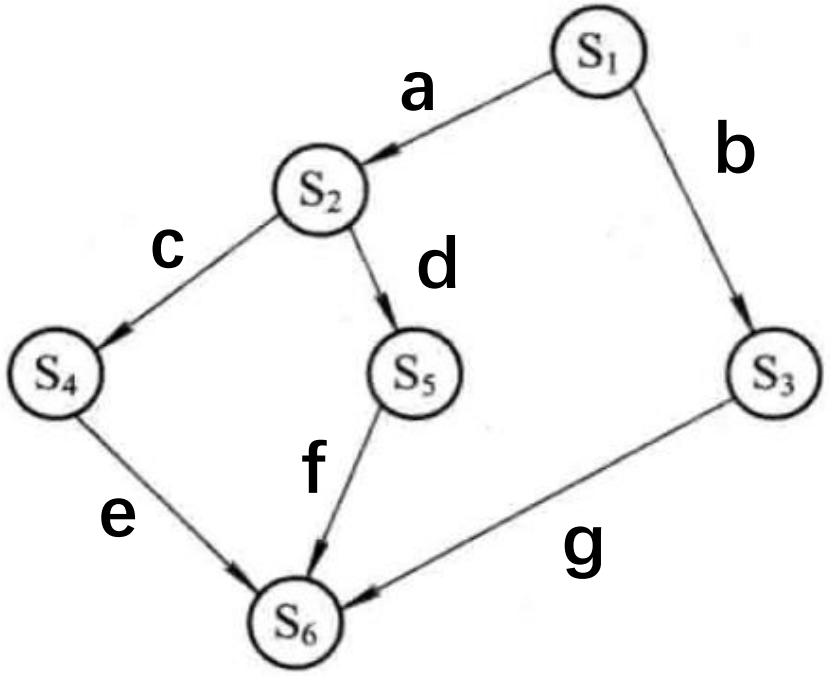
```
S2(){
    P(a);
    ...
    ...
}
```

```
S3(){
    P(b);
    ...
    ...
}
```




```
semaphore c=0, d=0;
S2(){
    P(a);
    ...
    ...
    V(c);
    V(d);
}
```

```
S4(){
    P(c);
    ...
    ...
}

S5(){
    P(d);
    ...
    ...
}
```




- 一个顶点表示一个进程，有多少个顶点，就有多少个进程（函数）
- 一条边代表了一个约束（制约，一个先后关系），即需要一个信号量来控制其先后关系。

```
semaphore a=0,b=0,c=0,d=0,e=0,f=0,g=0;
```

```
P1() {
  ...
  S1;
  V(a);
  V(b);
  ...
}
```

```
P2() {
  ...
  P(a);
  S2;
  V(c);
  V(d);
  ...
}
```

```
P3() {
  ...
  P(b);
  S3;
  V(g);
  ...
}
```

```
P4(){
  ...
  P(c);
  S4;
  V(e);
  ...
}
```

```
P5(){
  ...
  P(d);
  S5;
  V(f);
  ...
}
```

```
P6() {
  ...
  P(e);
  P(f);
  P(g);
  S6;
  ...
}
```

可以把信号量理解为一种信息，表示当前进程的直接前驱是否执行完毕（对应信号量值为0/1）

### 6．分析进程同步和互斥问题的方法步骤

1）关系分析。找出问题中的进程数，并分析它们之间的同步和互斥关系。同步、互斥、前驱关系直接按照上面例子中的经典范式改写。

2）整理思路。找出解决问题的关键点，并根据做过的题目找出求解的思路。根据进程的操作流程确定 P 操作、V 操作的大致顺序。

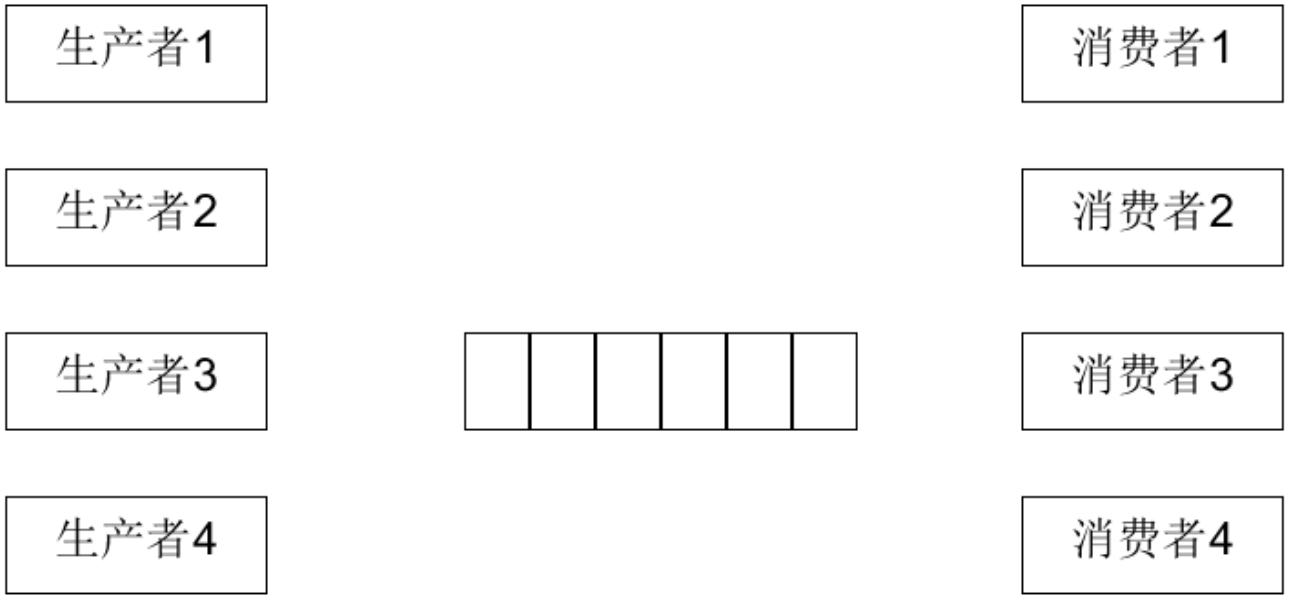
3）设置信号量。根据上面的两步，设置需要的信号量，确定初值，完善整理。

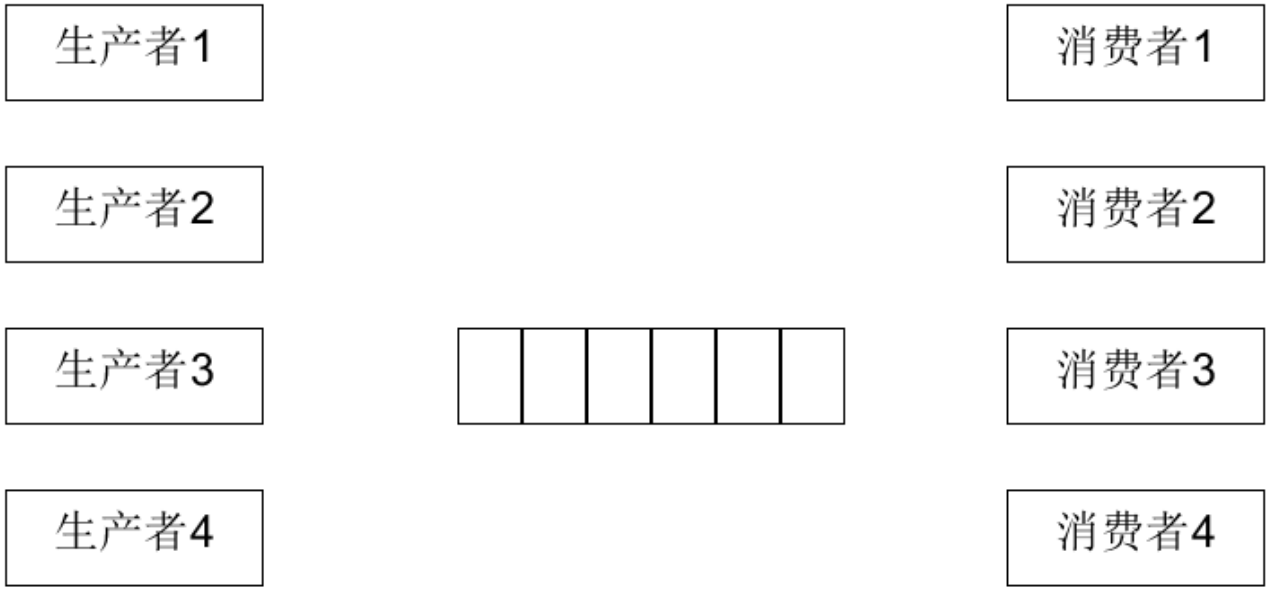
这是一个比较直观的同步问题，以 $S_2$ 为例，它是 $S_1$ 的后继，所以要用到 $S_1$ 的资源，在前面的简单总结中我们说过，在同步问题中，要用到某种资源，就要在行为（题中统一抽象成 L）前面 P 这种资源一下。$S_2$ 是 $S_4$，$S_5$ 的前驱，给 $S_4$，$S_5$ 提供资源，所以要在 L 行为后面 V 由 $S_4$ 和 $S_5$ 代表的资源一下。

# 经典同步问题

## 1. 生产者-消费者问题

问题描述：一组生产者进程和一组消费者进程共享一个初始为空、大小为 $n$ 的缓冲区，只有缓冲区没满时，生产者才能把消息放入缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或一个消费者从中取出消息。






```
semaphore mutext=1;
semaphore empty_num = n;
semaphore full_num=0;
P(){
produce data;
P(empty_num);
P(mutex);
put data;
V(mutex);
V(full_num);
}

C(){
P(full_num);
P(mutex);
move data;
V(mutex);
V(empty_num);
}
```

```
1  typedef struct {
2      int *buf;          /* Buffer array */
3      int n;             /* Maximum number of slots */
4      int front;         /* buf[(front+1)%n] is first item */
5      int rear;          /* buf[rear%n] is last item */
6      sem_t mutex;       /* Protects accesses to buf */
7      sem_t slots;       /* Counts available slots */
8      sem_t items;       /* Counts available items */
9  } sbuf_t;
```

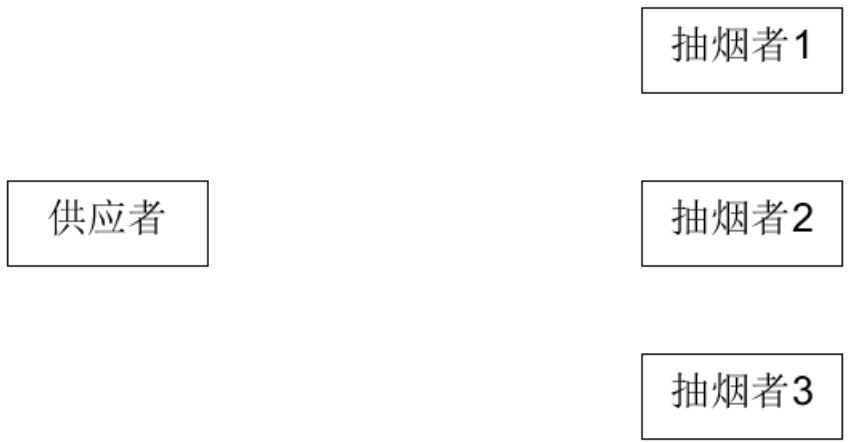
```
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                          /* Buffer holds max of n items */
    sp->front = sp->rear = 0;           /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1);         /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n);         /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0);         /* Initially, buf has zero data items */
}
```

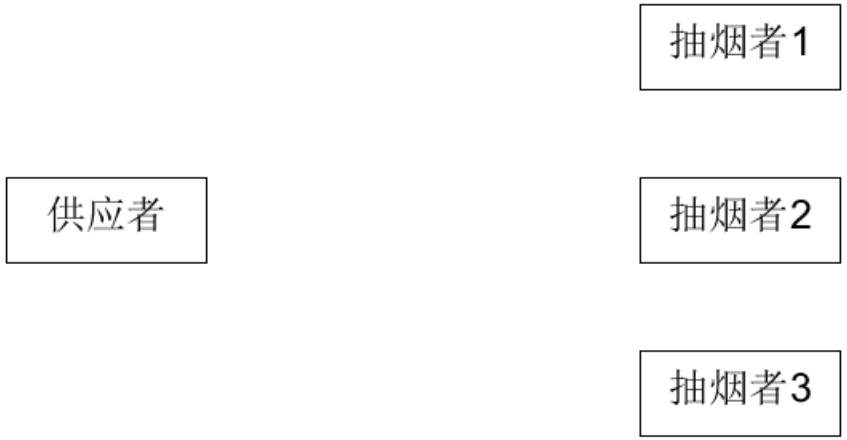
```
/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;   /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}

/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

## 4．吸烟者问题

问题描述：假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草，第二个拥有纸，第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放到桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者一个信号告诉已完成，此时供应者就会将另外两种材料放到桌上，如此重复（让三个抽烟者轮流地抽烟）。






```
int count=0;
semaphore s1=0,s2=0,s3=0;
semaphore a=0;
P(){
count++;
if (count%3=1)
        V(s1);
else if (count%3=2)
        V(s2);
else
        V(s3);
P(a);
}

Smoke1(){
P(s1);
卷烟;
抽烟;
V(a);
}
```

# 管程

在信号量机制中，每个要访问临界资源的进程都必须自备同步的 PV 操作，大量分散的同步操作给系统管理带来了麻烦，且容易因同步操作不当而导致系统死锁。于是，便产生了一种新的进程同步工具——管程。管程的特性保证了进程互斥，无须程序员自己实现互斥，从而降低了死锁发生的可能性。同时管程提供了条件变量，可以让程序员灵活地实现进程同步。

### 1．管程的定义

系统中的各种硬件资源和软件资源，均可用数据结构抽象地描述其资源特性，即用少量信息和对资源所执行的操作来表征该资源，而忽略它们的内部结构和实现细节。

利用共享数据结构抽象地表示系统中的共享资源，而把对该数据结构实施的操作定义为一组过程。进程对共享资源的申请、释放等操作，都通过这组过程来实现，这组过程还可以根据资源情况，或接受或阻塞进程的访问，确保每次仅有一个进程使用共享资源，这样就可以统一管理对共享资源的所有访问，实现进程互斥。这个代表共享资源的数据结构，以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序，称为管程（monitor）。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。

由上述定义可知，管程由 4 部分组成：

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（或函数）；

④对局部于管程内部的共享数据设置初始值的语句。

管程的定义描述举例如下：

```
monitor Demo{ //①定义一个名称为“Demo”的管程
    //②定义共享数据结构，对应系统中的某种共享资源
    共享数据结构 S;
    //④对共享数据结构初始化的语句
    init_code(){
        S=5;                    //初始资源数等于 5
    }
    //③过程 1：申请一个资源
    take_away(){
        对共享数据结构 x 的一系列处理;
        S--;                    //可用资源数-1
        ...
    }
    //③过程 2：归还一个资源
    give_back(){
        对共享数据结构 x 的一系列处理;
        S++;                    //可用资源数+1
        ...
    }
```

```
Monitor Printer_monitor{
    Printer_struct S; // 共享数据结构
    ...
    init_code(){
        ...
        S.count = 2;
        ...
    }

    take_away(){
        S.count--;
        ...
    }

    give_back(){
        S.count++;
        ...
    }

}
```

```
P1(){
    Printer_monitor.take_away();
    ...
    使用打印机
    Printer_struct.give_back();
}

P2(){
    Printer_monitor.take_away();
    ...
    使用打印机
    Printer_struct.give_back();
}

P3(){
    Printer_monitor.take_away();
    ...
    使用打印机
    Printer_struct.give_back();
}
```

# 进程和管程的不同

管程和进程不同：① 虽然二者都定义了数据结构，但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，如消息队列等；② 二者都存在对各自数据结构上的操作，但进程是由顺序程序执行有关操作，而管程主要是进行同步操作和初始化操作；③ 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；④ 进程通过调用管程中的过程对共享数据结构实行操作，该过程就如通常的子程序一样被调用，因而管程为被动工作方式，进程则为主动工作方式；⑤ 进程之间能并发执行，而管程则不能与其调用者并发；⑥ 进程具有动态性，由“创建”而诞生，由“撤消”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。

# 练习

- 整理实现临界区互斥的基本方法
- 整理进程和管程的不同
- 独立写出生产者消费者的PV代码
- 独立写出吸烟者问题的PV代码

1. 什么是前趋图？为什么要引入前趋图？
2. 试画出下面四条语句的前趋图：

S1: a = x+y;

S2: b = z+1;

S3: c = a-b;

S4: w = c+1;

# 真题

- 2018.11 信号量最大值的计算 填空
- 2019.11 临界区个数计算 填空

# 2016.7

7. 进程之间存在哪几种相互制约关系？各是什么原因引起的？下面活动分别属于哪种制约关系？

（1）若干同学去图书馆借书

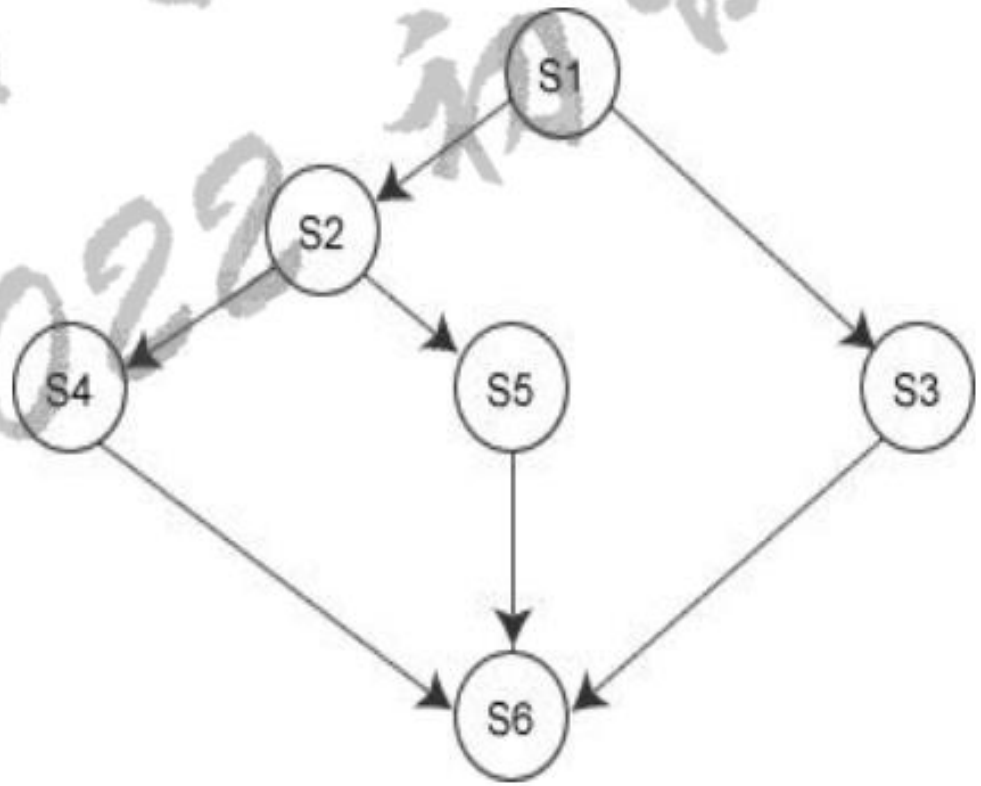
（2）两队举行篮球比赛

（3）流水线生产的各道工序

（4）商品生产和消费

# 2022年真题

1.用信号量实现下列前驱关系



| 考点 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 甘特图 | | | | | | | | | | 10 | |
| 三级调度 | | | | 13 | 8 | 9 | 20 | 20 | 1 | 11 | |
| 同步与互斥 | 6 | 5 | | | 10 | 11 | 19 | 18 | 2 | 9 | 4 |