2.4.3　信号量机制

信号量（Semaphores）机制是一种卓有成效的进程同步工具。信号量机制已被广泛应用于单处理机和多处理机系统以及计算机网络中。

信号量机制的发展：

- 整型信号量　“忙等”，未遵循“让权等待”准则。
- 记录型信号量　——后边重点介绍。
- AND型信号量
- 信号量集

} 记录型信号量的扩展。

1. 整型信号量

最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量S，它与一般整型量不同，除初始化外，仅能通过两个标准的原子操作wait(S)和signal(S)来访问，这两个操作一般称为P、V操作。

P为荷兰语的“通过”，V为荷兰语的“释放”

```
wait(S)
        { while (S<=0);              /*do  no-op*/
              S=S--;
        }
```

```
signal(S)
        {
              S=S++;
        }
```

wait(s)和signal(s)是两个原子操作，在执行时是不可中断的。即，当一个进程在修改某信号量时，没有其他进程可以同时对该信号量进行修改。

整型信号量的应用

```
int s =1;
Printer( )
    {
        wait(s);
        print the document on the paper;
        signal(s);
    }
```

2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量S≤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。记录型信号量机制则是一种不存在“忙等”现象的进程同步机制。

但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。

为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表指针list，用于链接上述的所有等待进程。

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

记录型数据结构，在C语言中为结构型数据结构。

wait和signal操作可用C/C++语言描述如下：

```
wait (semaphore  *S){
 S->value -- ;
 if (S->value <0 )
   block (S->list) ;
   /* 让权等待 */
}
```

P(S)

```
signal (semaphore  *S){
 S->value ++ ;
 if (S->value <=0 )
   wakeup (S->list) ;
/*唤醒第一个等待的进程 */
}
```

V(S)

S->value的绝对值表示在该信号量链表中已阻塞进程的数目。

对资源信号量S的每次signal操作，意味着进程释放一个单位资源，使系统中可供分配的该类资源数增加1。若加1后仍是S->value≤0，则表示该信号量链表中仍有等待该资源的进程被阻塞，故还应调用**wakeup**原语，将**S->list**链表中的第一个等待进程唤醒。

P(S)

对资源信号量S的每次wait操作，意味着进程请求一个单位的该类资源，使系统中可供分配的该类资源数减1。若减1后仍是S->value≥0，则表示该类资源已经分配完毕。因此进程应调用**block**原语进行自我阻塞，放弃处理机，并插入到信号量链表**S->list**中。

V(S)

如果S->value的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号了，用于进程互斥。

3. AND型信号量

前面所述的进程互斥问题针对的是多个并发进程仅共享一个临界资源的情况。在有些应用场合，是一个进程往往需要获得两个或更多的共享资源后方能执行其任务。

假定现有两个进程A和B，它们都要求访问共享数据D和E，当然，共享数据都应作为临界资源。可以为这两个数据分别设置两个信号量Dmutex和Emutex来实现互斥，它们的初值都是1。

反例

Process A:	Process B:
wait(Dmutex);	wait(Emutex);
wait(Emutex);	wait(Dmutex);
……	……

死锁

若A和B按以下次序执行wait操作：

Process A：wait(Dmutex); 此时Dmutex=0;

Process B：wait(Emutex); 此时Emutex=0;

Process A：wait(Emutex); 此时Emutex=-1; A阻塞；

Process B：wait(Dmutex); 此时Dmutex=-1; B阻塞；

And型信号量

思想：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完成后再一起释放。只要尚有一个资源未能分配给进程，其他所有可能为之分配的资源，也不分配给它。

即，对若干个临界资源的分配，采取原子操作方式：要么全部分配给进程，要么一个也不分配。

为此，在wait操作中增加了一个“AND”条件，故称为AND同步，或称为同时wait操作，即Swait（simultaneous wait）

And型信号量的P操作

```
Swait(S1,S2,…,Sn)
{
 while(TRUE)
 {
  if S1>=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 counter of this
     process to the beginning of Swait operation
  }
 }
}
```

P操作

将相关的所有进程移入第一个Si<1的等待队列中，并将该进程的PC指针指向Swait操作开始处

And型信号量的V操作

```
Ssginal(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.
  }
 }
}
```

V操作

把阻塞队列里所有和Si相关的进程送进就绪队列

4. 信号量集

在前面所述的记录型信号量机制中，wait(S)或signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。当一次需要N个单位时，便要进行N次wait(S)操作，这显然是低效的，甚至会增加死锁的概率。

此外，有些情况下，为确保系统的安全性，当所申请的资源数量低于某一下限值时，还必须进行管制，不予以分配。

因此，当进程申请某类临界资源时，在每次分配之前，都必须测试资源的数量，判断是否大于可分配的下限值，决定是否予以分配。

信号量集

【思想】对AND信号量机制加以改进，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Si的测试值不再是1，而是该资源的分配下限值ti，即要求Si≥ti，否则不予分配。
- 一旦允许分配，进程对该资源的需求值为di，即表示资源进行Si=Si-di操作，而不是简单的Si=Si-1操作。
- 由此形成一般化的“信号量集”机制。

以下的程序中，S为信号量，d为需求值，t为下限值。

Swait(S_1,t_1,d_1,…,S_n,t_n,d_n)

if $S_1 \geq t_1$ and … and $S_n \geq t_n$ then

for i=1 to n do

$S_i=S_i-d_i$;

endfor

else

Place the executing process in the waiting queue of the first S_i with $S_i<t_i$ and set its program counter to the beginning of the Swait Operation.

endif

将正在执行的进程移入第一个Si<ti的等待队列中，并将该进程的PC指向Swait操作开始处

P操作

Ssignal(S_1,d_1,…,S_n,d_n)
for i=1 to n do
S_i=S_i+d_i;
Remove all the process waiting in the queue associated with S_i into the ready queue
endfor;

V操作

信号量集的几种特殊情况：

（1）Swait(S,d,d)：信号量集中只有一个信号量S，但允许它每次申请d个资源，当现有资源小于d时，不予分配；

（2）Swait(S,1,1)： 等同于一般的记录型信号量（S>1）或互斥信号量（S=1）；

（3）Swait(S,1,0)：当S≥1时，允许多个进程进入某特定区；当S变为0后，阻止所有进程进入临界区。其功能类似于可控开关。

单选题 1分

设置

（　　）是一种只能进行P操作和V操作的特殊变量

A 调度

B 进程

C 同步

D 信号量

提交

单选题 1分

设置

若P、V操作的信号量S初值为2，当前值为-1，则表示有（　　）等待进程

A 0个

B 1个

C 2个

D 3个

提交

单选题 1分

设置

对于两个并发进程，设互斥信号量为mutex，若mutex=0，则（　　）

A 表示没有进程进入临界区

B 表示有一个进程进入临界区

C 表示有一个进程进入临界区，另一个进程等待进入

D 表示有两个进程进入临界区

提交

单选题 1分

设置

有m个进程共享同一临界资源，若使用信号量机制实现对临界资源的互斥访问，则信号量值的变化范围是（　　）

A [-(m-1)，1]

B [-m，1]

C (-(m-1)，1]

D (-m，1)

提交

单选题 1分

设置

操作系统中，对信号量S的P原语操作定义中，使进程进入相应等待队列等待的条件是（　　）

A　S>0

B　S=0

C　S<0

D　S≤0

提交

2.4.4　信号量的应用

1. 利用信号量实现进程互斥

★为临界资源设置一个互斥信号量mutex，其初值为1；

★各进程访问该资源的临界区置于wait（mutex）和signal（mutex）之间即可。

wait（mutex）——进入区
signal（mutex）——退出区

2.4.4　信号量的应用

mutex的取值为（-1，0，1）：

- **mutex=1：两个进程都未进入临界区；**
- **mutex=0：有一个进程进入临界区运行，另一个如需运行，必须等待，挂入阻塞队列；**
- **mutex=-1：有一个进程正在临界区运行，另一个进程因等待而阻塞在信号量队列中，需要被当前已在临界区运行的进程在退出时唤醒。**

代码描述：

```
semaphore  mutex=1;
  Pa( ){
  while(1){
    wait(mutex);
    临界区；
    signal(mutex);
    剩余区：
  }
}
```

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

代码描述：

```
semaphore  mutex=1;
 Pa( ){
 while(1){
   wait(mutex);
   临界区；
   signal(mutex);
```

```
Pb( ){
 while(1){
   wait(mutex);
   临界区；
   signal(mutex);
```

利用信号量实现进程互斥时必须注意，wait(mutex)和signal(mutex)必须成对出现。

- 缺少wait(mutex)将会导致系统混乱，不能保证对临界资源的互斥访问；
- 缺少signal(mutex)将会使临界资源永远不被释放，从而使因等待该资源而阻塞的进程不能被唤醒。

【举例】某交通路口设置了一个自动计数系统，该系统由“观察者”进程和“报告者”进程组成。观察者进程能识别卡车，并对通过的卡车计数；报告者进程定时将观察者的计数值打印输出，每次打印后把计数值清“0”。两个进程的并发执行可完成对每小时中卡车流量的统计。

```
struct semaphore S ； int count = 0 ； S.value = 1 ;
```

```
process observer
{  while (condition){
     observe a lorry;
     wait（S）;
     count = count + 1;
     signal（S）;
}}
```

临界区

```
process reporter
{while (condition){
     wait（S）;
     print（count）;
     count = 0;
     signal（S）;
}
```

临界区

【举例】某交通路口设置了一个自动计数系统，该系统由“观察者”进程和“报告者”进程组成。观察者进程能识别卡车，并对通过的卡车计数；报告者进程定时将观察者的计数值打印输出，每次打印后把计数值清“0”。两个进程的并发执行可完成对每小时中卡车流量的统计。

struct semaphore S； int count = 0； S.value = 1；

process observer　　　　　　　　**process reporter**

临
界
区

第1步：搞清楚谁是临界资源

计数值count

第2步：搞清楚哪些是临界区

访问count的语句，包括累加、打印和清零

signal（S）；　　　　　　　　**signal（S）；**

} }　　　　　　　　**}**

2. 利用信号量实现前趋关系（同步关系）

设有两个并发执行的进程P_1和P_2。P_1中有语句S_1；P_2中有语句S_2。我们希望在S_1执行后再执行S_2。为实现这种前趋关系，需要进行如下操作：

使进程P_1和P_2共享一个公用信号量S，并赋予其初值为0，将signal(S)操作放在语句A后面，而在B语句前面插入wait(S)操作，即

在进程P_1中，用A → signal(S)的顺序执行

在进程P_2中，用wait(S) → B的顺序执行

由于S被初始化为0，这样，若P2先执行必定阻塞，只有在进程P1执行完A和signal(S)操作后使S增为1时，P2进程方能成功执行语句B。

一次同步执行：

一次性

```
semaphore s1;
s1.value=0;
P0( ){
    A ;
    Signal(s1);
}
```

```
P1( ){
    Wait(s1);
    B ;
}
```

先运行的进程

后运行的进程

思考：为什么是一次性的？为什么不能如果循环重复？

循环同步执行：

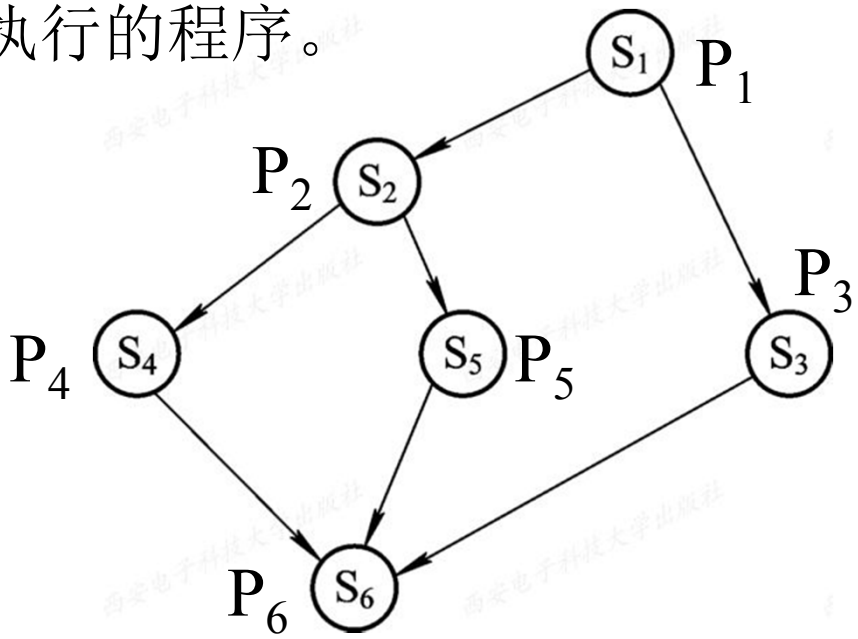
```
struct semaphore s1,s2;
  s1.value=0;
  s2.value=1;

P0( ){
   wait(s2) ;
   A;
   Signal(s1);
}
```

按照一定顺序循环运行

```
P1( ){
   wait(s1);
   B;
   signal(s2) ;
}
```

我们可以利用信号量按照语句间的前趋关系（见下图），写出一个更为复杂的可并发执行的程序。



图中表示：

- 进程P1中有语句S1；
- 进程P2中有语句S2；

……

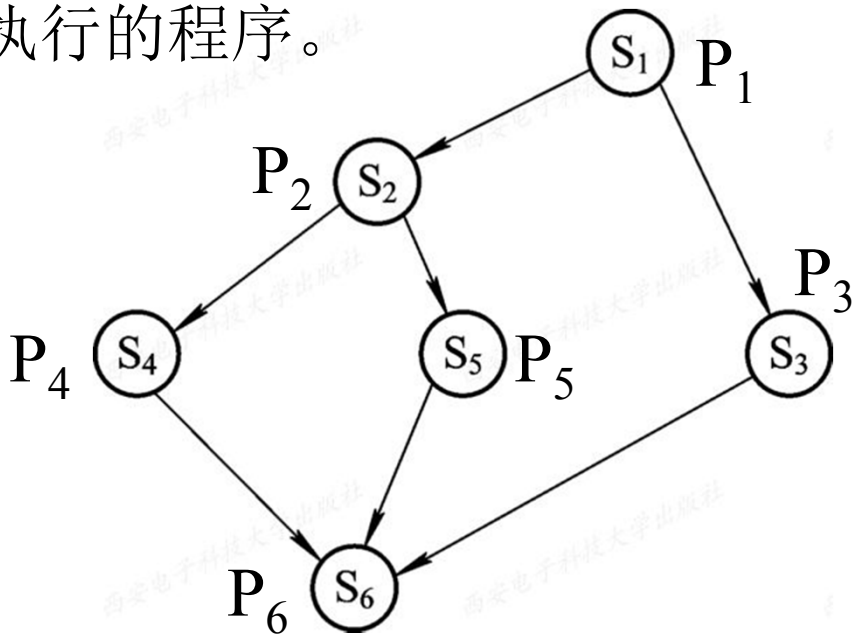
- 语句S1执行后才能执行语句S2和语句S3；
- 语句S2执行后才能执行语句S4和S5；
- 语句S3、S4和S5执行后，才能执行语句S6。

P1执行完应通知P2、P3；
P2得到通知后才开始执行；
P2执行完应通知P4、P5；
……

我们可以利用信号量按照语句间的前趋关系（见下图），写出一个更为复杂的可并发执行的程序。

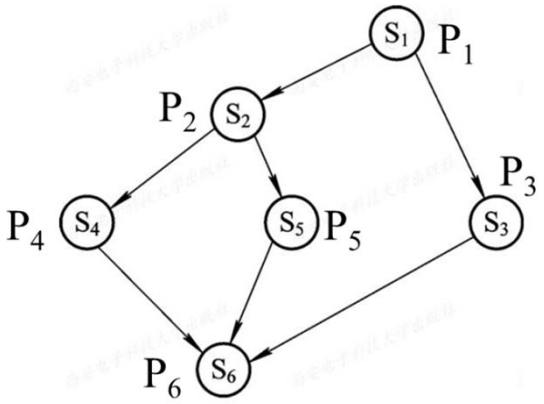
对应于每一对前趋关系，应分别设置信号量，如信号量a用于$S_1 \rightarrow S_2$的前趋关系。

$S_1 \rightarrow S_2$：a
$S_1 \rightarrow S_3$：b
$S_2 \rightarrow S_4$：c
$S_2 \rightarrow S_5$：d
$S_3 \rightarrow S_6$：e
$S_4 \rightarrow S_6$：f
$S_5 \rightarrow S_6$：g



P1执行完应通知P2、P3；
P2得到通知后才开始执行；
P2执行完应通知P4、P5；
……

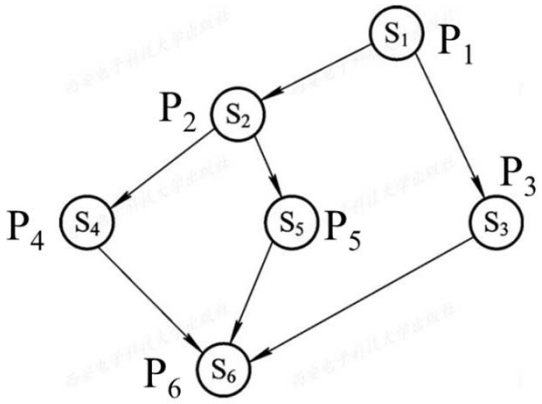
```
struct semaphore a，
   a.value = b.value
= c.value = d.value
= e.value = f.value
= g.value = 0;
```



初始化

```
   begin  /*begin表示并发执行开始*/
   P1：{ S1；signal（a）；signal（b）；  }
   P2：{ wait（a）；S2；signal（c）；signal（d）；  }
   P3：{ wait（b）；S3；signal（e）；  }
   P4：{ wait（c）；S4；signal（f）；  }
   P5：{ wait（d）；S5；signal（g）；  }
   P6：{ wait（e）；wait（f）；wait（g）；S6；  }
   end   /*用end表示并发执行结束 */
```

struct semaphore a，
a.value = b.value
= c.value = d.value
= e.value = f.value
= g.value = 0;



$S_1 \rightarrow S_2$: a
$S_1 \rightarrow S_3$: b
$S_2 \rightarrow S_4$: c
$S_2 \rightarrow S_5$: d
$S_3 \rightarrow S_6$: e
$S_4 \rightarrow S_6$: f
$S_5 \rightarrow S_6$: g

```
begin /*begin表示并发执行开始*/
P1: { S1; signal（a）; signal（b）; }
P2: { wait（a）; S2; signal（c）; signal（d）; }
P3: { wait（b）; S3; signal（e）; }
P4: { wait（c）; S4; signal（f）; }
P5: { wait（d）; S5; signal（g）; }
P6: { wait（e）; wait（f）; wait（g）; S6; }
end  /*用end表示并发执行结束 */
```

3. 利用信号量控制使用资源进程的数量（资源管控）

系统中有5台打印机可以使用，请使用进程同步机制使得最多可以有5个进程可以同时使用打印机，多于5个进程使用时，要对新申请进程进行阻塞。

程序描述如下：

```
struct semaphore S;
S.value=5 ;
Printer{
    wait(S);
        print the document on the paper;
    signal(S);
}
```

2.4.5　管程机制

自学

习题1

在一个盒子里，混装了数量相等的黑白围棋子。现在用自动分拣系统把黑子、白子分开，设分拣系统有两个进程P1和P2，其中P1拣白子，P2拣黑子。规定当一个进程拣了一子后，必须让另一个进程去拣。用信号量和PV操作协调两进程的活动。

灵魂3问：

- 互斥问题or同步问题or资源管控问题？ 同步
- 需要几个信号量？ 2个，需要循环执行
- 初值如何设置？ 分别设置为0和1

解答1

```
struct semaphore S1, S2;
S1.value=1; S2.value=0;

process P1( ){
      while(true){
         P(S1);
         拣黑子( );
         V(S2);
      }
   }
```

P(S1)即wait(S1);
V(S1)即signal(S1)

```
process P2( ){
    while(true){
       P(S2);
       拣白子( );
       V(S1);
    }
}
```

习题2

某控制系统中，数据采集进程负责把采集到的数据放到一缓冲区中；分析进程负责把数据从缓冲区中取出进行分析，试用信号量实现两者之间的同步。

灵魂3问：

- 互斥问题or同步问题or资源管控问题？ 同步
- 需要几个信号量？ 2个，需要循环执行
- 初值如何设置？ 分别设置为0和1

解答2

```
Struct semaphore S1, S2;
S1.value = S2.value=0;

process P1( ){
    while(true){
        采集数据( );
        P(S1);
        放缓冲区( );
        V(S2);
    }
}

process P2( ){
    while(true){
        P(S2);
        缓冲区中取数据( );
        V(S1);
        分析( );
    }
}
```

习题3

图书馆规定，每位进入图书馆的读者要在登记表上登记，退出时要在登记表上注销。

（1）用信号量实现读者之间的互斥登记和注销；

（2）图书馆共有100个座位，当图书馆中没有空座位时，后到的读者在图书馆要等待（阻塞）。

灵魂3问：

互斥和资源管控

- 互斥问题or同步问题or资源管控问题？
- 需要几个信号量？

2个，1个互斥，1个资源管控

- 初值如何设置？

分别设置为1和100

解答3

```
Struct semaphore S, mutex;
s.value=100;            /*100个座位资源*/
mutex.value=1;          /*互斥信号量*/
reader( ){
    P(s);
    P(mutex);
    进入登记;
    V(mutex);
    读书;
    P(mutex);
    退出登记;
    V(mutex);
    V(s);
}
```

习题4

一家四人父、母、儿子、女儿围桌而坐；桌上有一个水果盘； 当水果盘空时，父亲可以放香蕉或者母亲可以放苹果，但盘中已有水果时，就不能放，父母等待。当盘中有香蕉时，女儿可吃香蕉，否则，女儿等待；当盘中有苹果时，儿子可吃，否则，儿子等待。

习题5

在公共汽车上，司机和售票员的活动分别是：

司机的活动：　　启动车辆
　　正常运行
　　到站停车

售票员的活动：　　关车门
　　售票
　　开车门

在汽车不断的到站，停车，行驶过程中，司机和售票员
的活动有什么同步关系？用信号量和Ｐ，Ｖ操作实现。

习题6

有一个超市，最多可容纳N个人进入购物，当N个顾客满员时，后到的顾客在超市外等待；超市中只有一个收银员。可以把顾客和收银员看作两类进程，两类进程间存在同步关系。写出用P、V操作实现的两类进程的算法。