第二章 进程的描述与控制

上节课学习任务整理单：

1.理解OS实现进程控制的方式；

2. 理解原语的概念；

3. 明确进程创建、终止、阻塞和唤醒；

进程控制

进程状态转换

进程控制

07:45

62

第二章 进程的描述与控制

今日学习任务整理单：

1.理解进程同步、临界资源；
2.掌握记录型信号量；
3. 学会运用信号量解决进程同步问题；
4. 明确进程与管程的区别

进程同步

今日学习资料：

1. 雨课堂中的MOOC资源-第二章进程管理-（5）进程同步；
4. 教材2.4和2.5。

今日作业：理解什么是进程同步。

07:45

63

2.4 进程同步

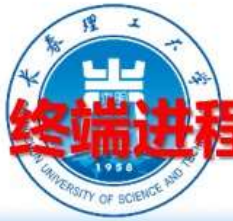
- 本节先要讨论的问题是：

(1) 进程并发执行一定会导致结果不唯一吗？

(2) 在什么情况下会顺利执行

(3) 在什么情况下会出现错误

例1：一个自动图书借阅系统，有两台终端可同时借阅，终端进程描述如下：



终端T1进程：

```
void TT(BOOK *bk)
{  /*BOOK为图书类型*/
  int n;
  n=bk->count;  /*该类图书的数量*/
  if( n>=1)
   {
     n=n-1;
     借出一本书;
     bk->count=n;
   }
}
```

终端T2进程：

```
void TT(BOOK *bk)
{  /*BOOK为图书类型*/
  int n;
  n=bk->count;  /*该类图书的数量*/
  if( n>=1)
   {
     n=n-1;
     借出一本书;
     bk->count=n;
   }
}
```

2.4 进程同步

本节要解决的问题是：

(1) 并发进程之间有什么关系

(2) 进程并发执行相互间会有什么影响

(3) 结果的不可再现性（与时间有关的错误）

2.4.1 进程同步的基本概念

1. 两种形式的制约关系

(1) 间接相互制约关系。

(2) 直接相互制约关系。

2.4.1 进程同步的基本概念

1. 两种形式的制约关系

2. 临界资源(Critical Resouce)

生产者-消费者(producer-consumer)问题是一个著名的进程同步问题。它描述的是：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中； 消费者进程可从一个缓冲区中取走产品去消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

例2:生产者—消费者问题

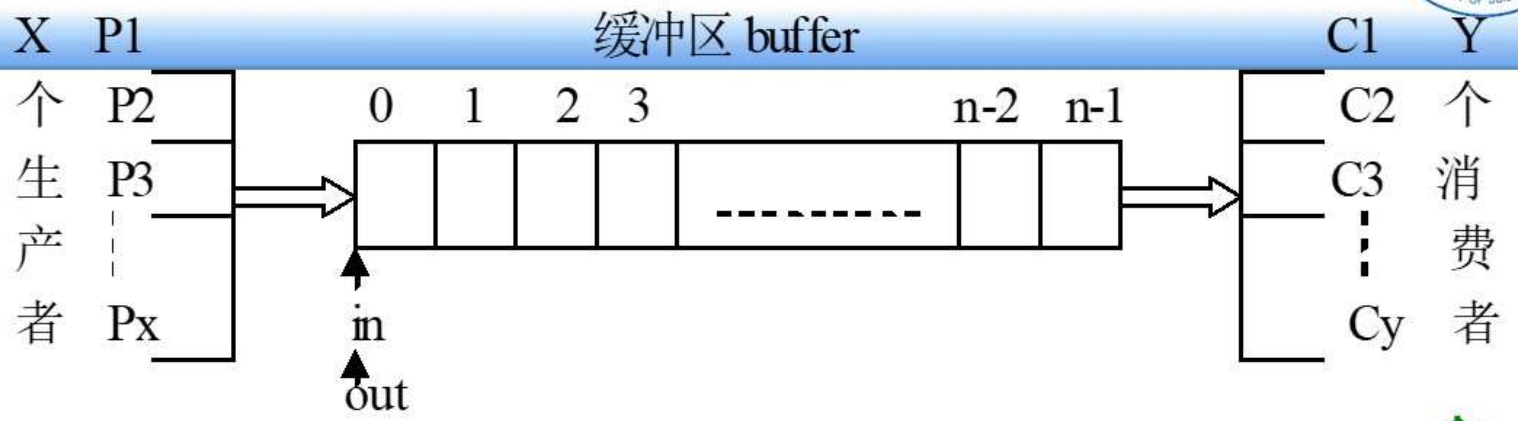
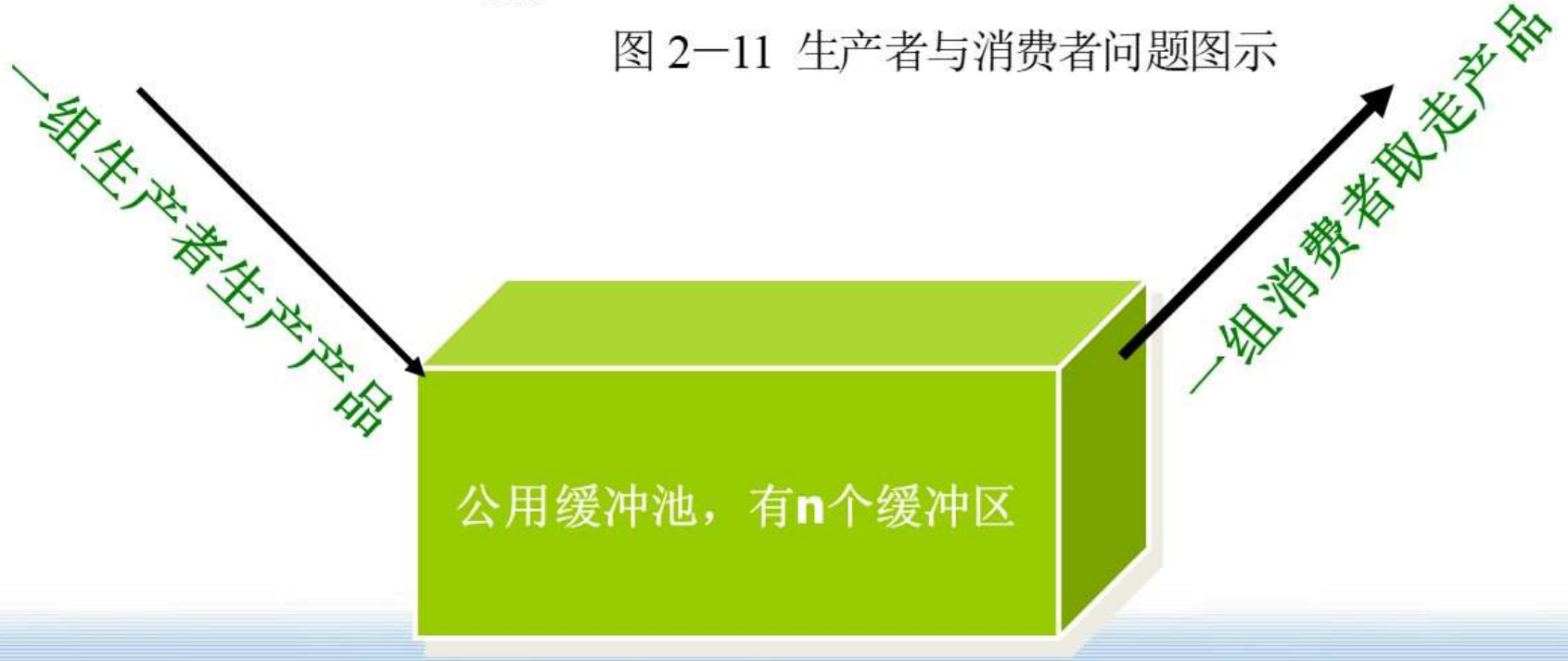


图 2—11 生产者与消费者问题图示



```
int    in=out=counter=0;
item   buffer[n];
```

```
void producer()
{ while(1)
   {
     produce an item in nextp;
     ……
     while(counter== n);
     buffer[in]= nextp;
     in= (in+1)% n;
     counter++;
   }
}
```

```
void consumer()
{ while(1)
   {
     while( counter= = 0);
     nextc= buffer[out];
     out = (out+1) % n;
     counter--;
     consumer the item in nextc;
     ……
     }
}
```

局部变量nextp，用于暂时存放每次刚生产出来的产品；而在消费者进程中，则使用一个局部变量nextc，用于存放每次要消费的产品。

2.4 进程同步

虽然上面的生产者程序和消费者程序，在分别看时都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时，就会出现差错，问题就在于这两个进程共享变量counter。生产者对它做加1操作，消费者对它做减1操作，这两个操作在用机器语言实现时，常可用下面的形式描述：

register1= counter;	register2= counter;
register1= register1+1;	register2= register2-1;
counter= register1;	counter= register2;

假设：counter的当前值是5。如果生产者进程先执行左列的三条机器语言语句，然后消费者进程再执行右列的三条语句，则最后共享变量counter的值仍为5；反之，如果让消费者进程先执行右列的三条语句，然后再让生产者进程执行左列的三条语句，counter值也还是5，但是，如果按下述顺序执行：

register 1 = counter; (register 1=5)

register 1 = register 1+1; (register 1=6)

register 2 = counter; (register 2=5)

register 2 = register 2-1; (register 2=4)

counter= register 1; (counter=6)

counter = register 2; (counter=4)

2.4.1 进程同步的基本概念

1. 两种形式的制约关系
2. 临界资源(Critical Resouce)
3. 临界区(critical section)

可把一个访问临界资源的循环进程描述如下：

```
while(1)
{
        entry section
        critical section;
        exit section
        remainder section;
}
```

2.4.1 进程同步的基本概念

1. 两种形式的制约关系
2. 临界资源(Critical Resouce)
3. 临界区(critical section)
4. 同步机制应遵循的规则

(1) 空闲让进。

(2) 忙则等待。

(3) 有限等待。

(4) 让权等待。

2.4.2 硬件同步机制

为防止多个进程同时测试到锁为打开的状态，测试与关锁操作必须是连续的，不允许分开进行。

1. 关中断

锁测试之前关闭中断，上锁后再打开中断。这样进程在临界区内，不响应中断。

关中断的缺点：

(1) 滥用关中断后果严重（不能及时响应）；
(2) 关中断时间长，影响系统效率；
(3)该方法不适应多CPU系统。

2.4.2 硬件同步机制

2. 利用test-and-set指令实现互斥（测试并建立）

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

利用TS指令实现互斥：

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

*lock=FALSE； 锁开，资源空闲
*lock=TRUE; 锁关，资源忙碌

2.4.2 硬件同步机制

3. 利用swap指令实现互斥

对换指令描述如下：

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

用对换指令实现互斥，先为每个临界资源设置一个全局的布尔变量lock初值为FALSE，如下：

```
int key;
.....
do{
      key=TRUE;
      do{
          swap(&lock,&key);
        }while(key!=FALSE);
      critical section
      lock=FALSE;
      remainder section;
  }while(TRUE);
```

2.4.3 信号量机制

1. 整型信号量

最初由Dijkstra把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作一直被分别称为P、V操作。

wait操作可描述为：

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

signal操作可描述为：

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

2.4.3 信号量机制

2. 记录型信号量

在整型信号量机制中的wait操作，是使进程处“忙等”的状态。而记录型信号量机制，则采取了“让权等待”的策略，为管理多个进程等待访问同一临界资源的情况，还应增加一个进程链表list，用于链接所有的等待进程。value用于代表资源数目的整型变量。

C语言中信号量定义

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

【P(S)、V(S)原语操作的算法描述】

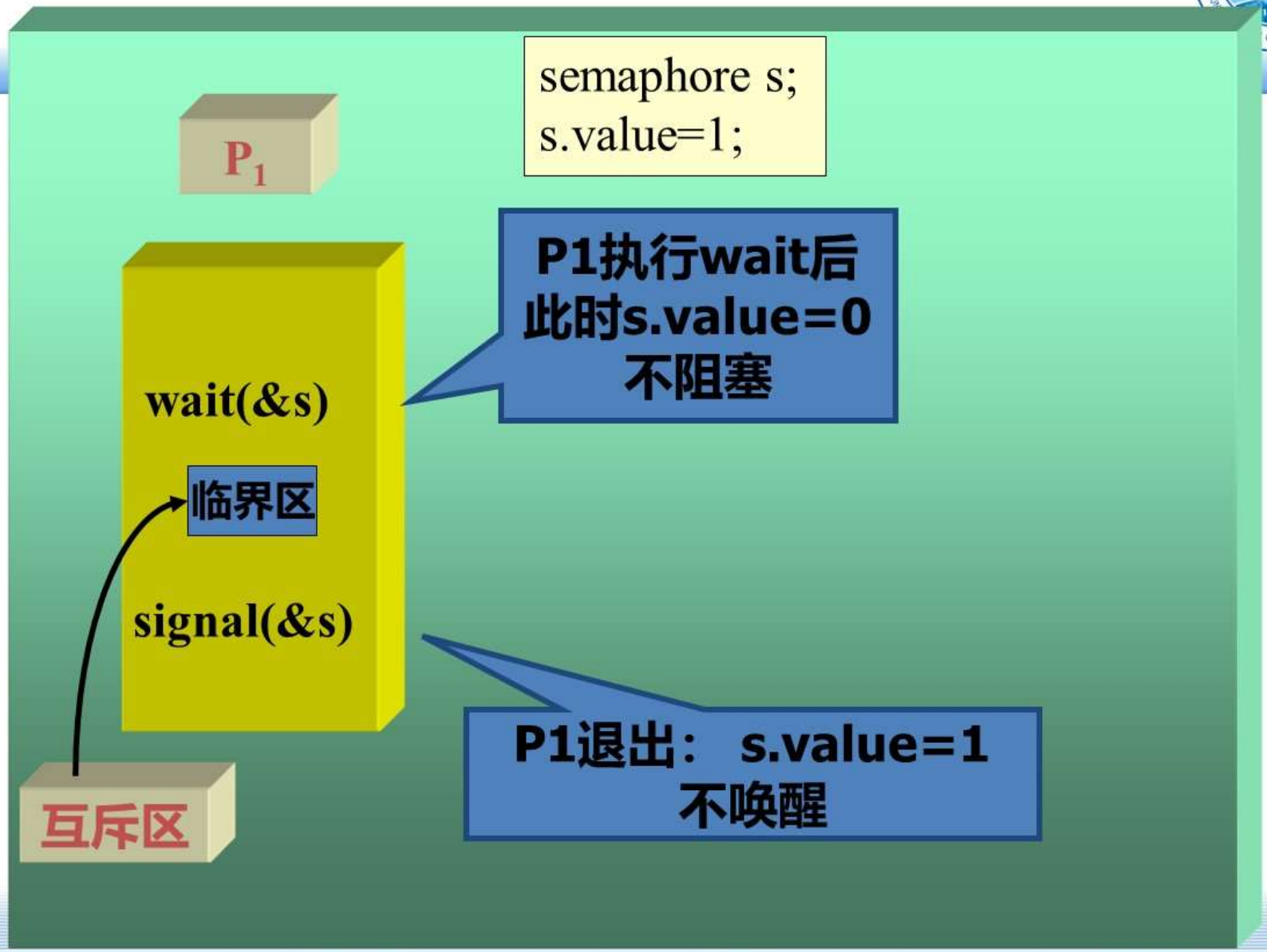
```
Wait(semaphore *s)
{
  s->value--;
  if  (s->value<0) block(s->list);
}
```

```
Signal(semaphore *s)
{
  s->value++;
  if (s->value<=0) wakeup(s->list);
}
```

C语言中信号量定义

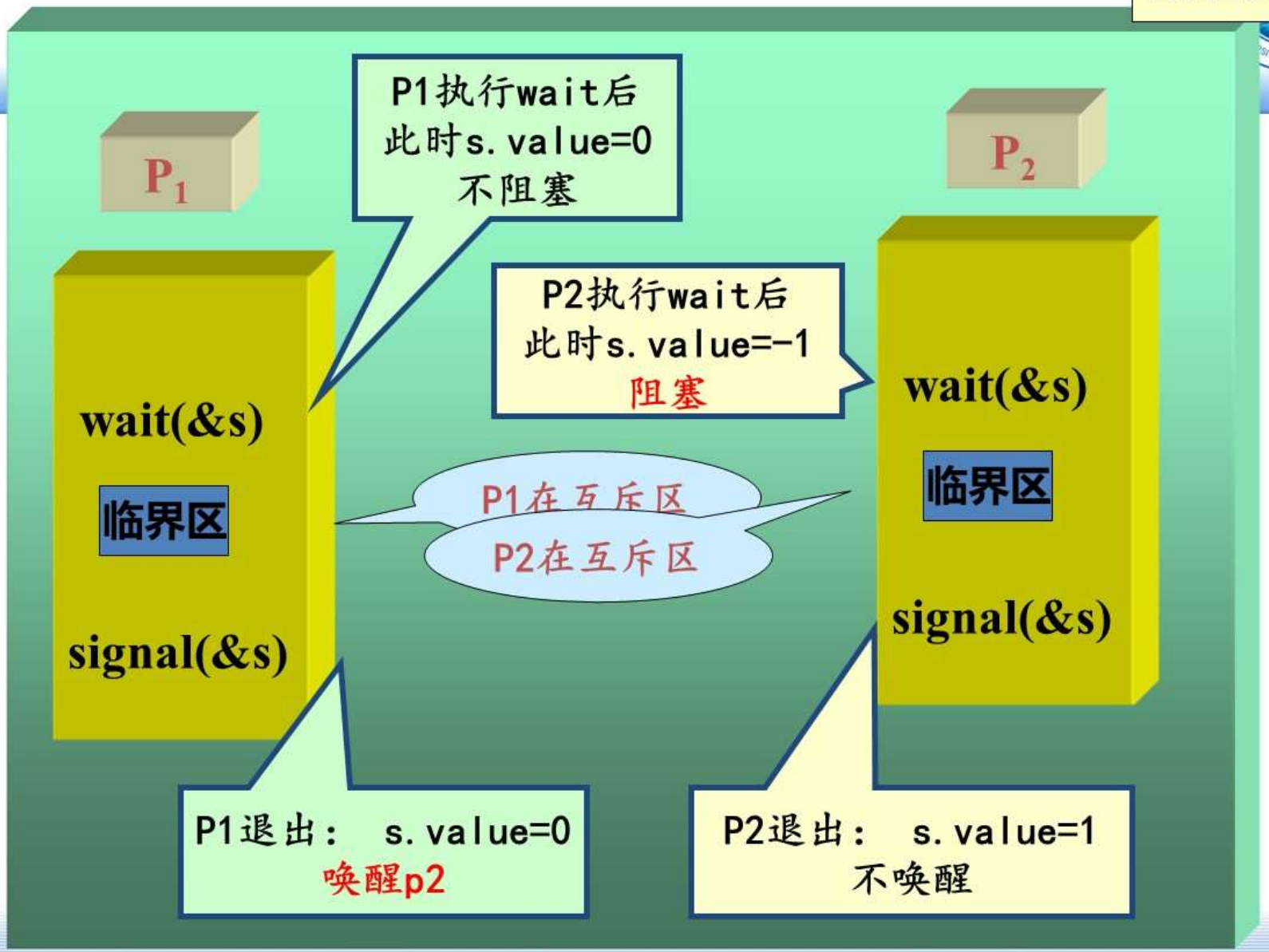
```
typedef struct
{
  int value;
  struct PCB *list;
} semaphore;
```

一个进程互斥进入临界区的情况



二个进程互斥进入临界区的情况

semaphore s;
s.value=1;



3. AND型信号量

若进程同时竞争2个临界资源d、e，描述如下：

```
process A                    process A:
wait(&d);                    wait(&d);
wait(&e);                    wait(&e);
临界区                       临界区
signal(&d)                   signal(&d)
signal(&e)                   signal(&e)
```

死锁

若进程A和B按下述次序交替执行wait操作：

process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

process A: wait(Emutex); 于是Emutex=-1 A阻塞

process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

2.4 进程同步

AND同步机制的基本思想是：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，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。亦即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。 由死锁理论可知，这样就可避免上述死锁情况的发生。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)定义如下：

```
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 count of this process to the beginning
of Swait operation }
    }
}
```

```
Ssignal(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.
        }
    }
}
```

4. 信号量集

若进程请求资源的数量超过1，并且当资源数量低于某一下限时将不予分配，这时就要扩展信号量的表示：

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

s_i为i类资源信号量（其值即系统中i类资源的可用数量）；t_i为资源的下限值；d_i为进程请求的i类资源数量。显然$t_i > d_i$

2.4.3 信号量机制

一般“信号量集”的几种特殊情况：

(1) Swait(S, d, d)。 此时在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，当现有资源数少于d时，不予分配。

(2) Swait(S, 1, 1)。 此时的信号量集已蜕化为一般的记录型信号量(S＞1时)或互斥信号量(S=1时)。

(3) Swait(S, 1, 0)。这是一种很特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

2.4.4 信号量的应用

1. 利用信号量实现进程互斥

Semaphore mutex = 1;

```
process1()
{
  while(1)
  {
    wait(mutex);
    critical section;
    signal(mutex);
    remainder seetion;
    }
}
```

```
process2()
{
  while(1)
  {
    wait(mutex);
    critical section;
    signal(mutex);
    remainder seetion;
    }
}
```

2. 利用信号量实现前趋关系

p1(){S_1; signal(a); signal(b); }
P2(){wait(a); S_2; signal(c); signal(d); }
P3(){wait(b); S_3; signal(g); }
P4(){wait(c); S_4; signal(e); }
P5(){wait(d); S_5; signal(f); }
P6(){wait(e); wait(f); wait(g); S_6; }

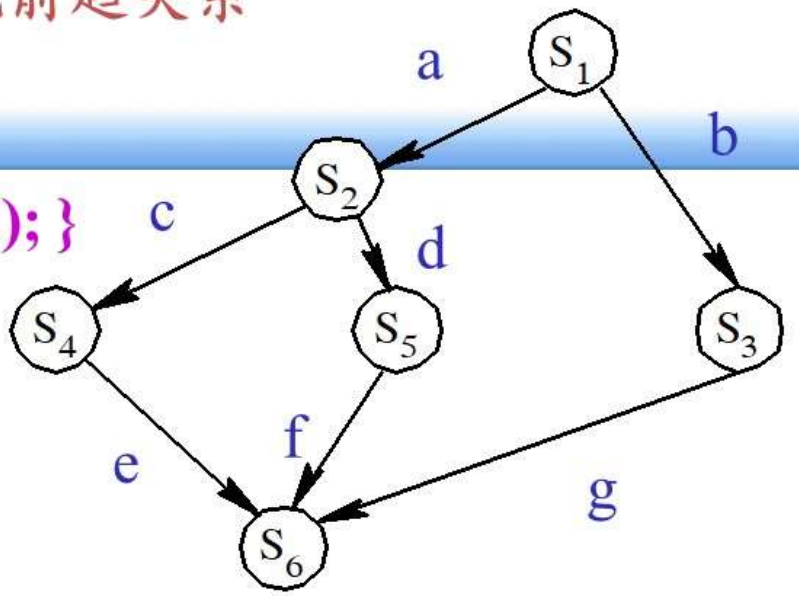


图 2-14 前趋图举例

```
main()
{
  semaphore a,b,c,d,e,f,g;
  a.value= b.value= c.value= d.value= 0;
  e.value= f.value= g.value= 0;
  cobegin
    p1(); p2(); p3(); p4(); p5(); p6();
  coend
}
```

2.4 进程同步

课堂练习题：

- 1、有两个用户进程A和B，在运行过程中都要使用系统中的一台打印机输出计算结果。
- 试说明A、B两进程之间存在什么样的制约关系?
- 为保证这两个进程能正确地打印出各自的结果，请用信号量和P、V操作写出各自的有关申请、使用打印机的代码。要求给出信号量的含义和初值。