http://m.blog.csdn.net/article/details?id=17597373

上篇博客中（[进程同步之临界区域问题及Peterson算法](http://blog.csdn.net/speedme/article/details/17595821" \t "http://m.blog.csdn.net/article/_blank)），我们对临界区，临界资源，锁机制详细解读了下，留下了一个问题，就是 锁机制 只能判断临界资源是否被占用，所以他解决了互斥问题，但是他不能确定前面的进程是否完成，所以他不能用于同步问题中。下面就为你讲解信号量机制是如何解决这一问题的。

## 1.信号量机制

信号量机制即利用pv操作来对信号量进行处理。

**什么是信号量**？信号量（semaphore）的数据结构为一个值和一个指针，指针指向等待该信号量的下一个进程。信号量的值与相应资源的使用情况有关。

当它的值大于0时，表示当前可用资源的数量；

当它的值小于0时，其绝对值表示等待使用该资源的进程个数。

注意，信号量的值仅能由P，V操作来改变。

     一般来说，信号量S³0时，S表示可用资源的数量。执行一次P操作意味着请求分配一个单位资源，因此S的值减1；当S<0时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去。而执行一个V操作意味着释放一个单位资源，因此S的值加1；若S£0，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。P：S-- V：S++

## 2.PV操作

****什么是PV操作？****

p操作（wait）：申请一个单位资源，进程进入

经典伪代码

wait(S){

while(s<=0) //如果没有资源则会循环等待；

;

S-- ;

}

v操作（signal）：释放一个单位资源，进程出来

signal(S){

S++ ;

}

p操作（wait）：申请一个单位资源，进程进入

v操作（signal）：释放一个单位资源，进程出来

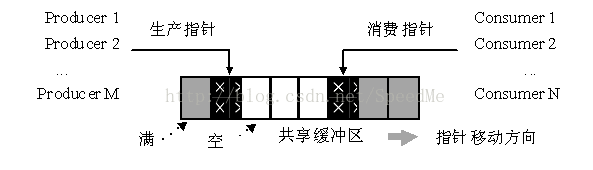
PV操作的意义：我们用信号量及PV操作来实现进程的同步和互斥。PV操作属于进程的低级通信。

使用PV操作实现进程互斥时应该注意的是：  
    （1）每个程序中用户实现互斥的P、V操作必须成对出现，先做P操作，进临界区，后做V操作，出临界区。若有多个分支，要认真检查其成对性。  
    （2）P、V操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。  
    （3）互斥信号量的初值一般为1。

## 3.三个经典同步问题

前面我们讲到信号量机制，下面我们讲解利用信号量及PV操作解决几个经典同步问题。

### a.生产者-消费者（缓冲区问题）



生产者一消费者问题(producer-consumerproblem)是指若干进程通过有限的共享缓冲区交换数据时的缓冲区资源使用问题。假设“生产者”进程不断向共享缓冲区写人数据(即生产数据)，而“消费者”进程不断从共享缓冲区读出数据(即消费数据)；共享缓冲区共有n个；任何时刻只能有一个进程可对共享缓冲区进行操作。所有生产者和消费者之间要协调，以完成对共享缓冲区的操作。

empty：可以生产n个 full：可以消费的数量 mutex：初值=1

生产者进程结构：

do{

wait(empty) ; empty--；

wait(mutex) ;

add nextp to buffer

signal(mutex) ;

signal(full) ;full++；

}while(1) ;

消费者进程结构：

do{

wait(full) ;--；

wait(mutex) ;

remove an item from buffer to nextp

signal(mutex) ;

signal(empty) ;++

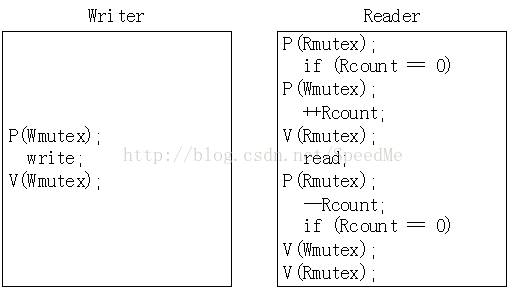
}while(1) ;

我们可把共享缓冲区中的n个缓冲块视为共享资源，生产者写人数据的缓冲块成为消费者可用资源，而消费者读出数据后的缓冲块成为生产者的可用资源。为此，可设置三个信号量：full、empty和mutex。其中：full表示有数据的缓冲块数目，初值是0；empty表示空的缓冲块数初值是n；mutex用于访问缓冲区时的互斥，初值是1。实际上，full和empty间存在如下关系：full + empty = N

### b.作者读者问题

读者一写者问题(readers-writersproblem)是指多个进程对一个共享资源进行读写操作的问题。

假设“读者”进程可对共享资源进行读操作，“写者”进程可对共享资源进行写操作；任一时刻“写者”最多只允许一个，而“读者”则允许多个。即对共享资源的读写操作限制关系包括：“读—写，互斥、“写一写”互斥和“读—读”允许。



信号量Wmutex表示“允许写”，初值是1；公共变量int Rcount表示“正在读”的进程数，初值是0；信号量Rmutex表示对Rcount的互斥操作，初值是1。

如果读进程==0，将要进行读操作，先占有写进程，不让其进行写操作，然后进行读操作。

在这个例子中，我们可见到临界资源访问过程的嵌套使用。在读者算法中，进入区和退出区又分别嵌套了一个临界资源访问过程。

### c.哲学家进餐问题

(1) 在什么情况下5 个哲学家全部吃不上饭？   
考虑两种实现的方式，如下：   
A.   
算法描述：

void philosopher(int i) /\*i：哲学家编号，从0 到4\*/

{

while (TRUE) {

think( ); /\*哲学家正在思考\*/

take\_fork(i); /\*取左侧的筷子\*/

take\_fork((i+1) % N); /\*取右侧筷子；％为取模运算\*/

eat( ); /\*吃饭\*/

put\_fork(i); /\*把左侧筷子放回桌子\*/

put\_fork((i+1) % N); /\*把右侧筷子放回桌子\*/

}

}

分析：假如所有的哲学家都同时拿起左侧筷子，看到右侧筷子不可用，又都放下左侧筷子，   
等一会儿，又同时拿起左侧筷子，如此这般，永远重复。对于这种情况，即所有的程序都在   
无限期地运行，但是都无法取得任何进展，即出现饥饿，所有哲学家都吃不上饭。   
B．   
算法描述：   
规定在拿到左侧的筷子后，先检查右面的筷子是否可用。如果不可用，则先放下左侧筷子，   
等一段时间再重复整个过程。   
分析：当出现以下情形，在某一个瞬间，所有的哲学家都同时启动这个算法，拿起左侧的筷   
子，而看到右侧筷子不可用，又都放下左侧筷子，等一会儿，又同时拿起左侧筷子……如此   
这样永远重复下去。对于这种情况，所有的程序都在运行，但却无法取得进展，即出现饥饿，   
所有的哲学家都吃不上饭。   
(2) 描述一种没有人饿死（永远拿不到筷子）算法。   
考虑了四种实现的方式（A、B、C、D）：   
A．原理：至多只允许四个哲学家同时进餐,以保证至少有一个哲学家能够进餐,最终总会释   
放出他所使用过的两支筷子,从而可使更多的哲学家进餐。以下将room 作为信号量，只允 许4 个哲学家同时进入餐厅就餐，这样就能保证至少有一个哲学家可以就餐，而申请进入 餐厅的哲学家进入room 的等待队列，根据FIFO 的原则，总会进入到餐厅就餐，因此不会 出现饿死和死锁的现象。   
伪码： room=4

semaphore chopstick[5]={1，1，1，1，1};

semaphore room=4;

void philosopher(int i)

{

while(true)

{

think();

wait(room); //请求进入房间进餐

wait(chopstick[i]); //请求左手边的筷子

wait(chopstick[(i+1)%5]); //请求右手边的筷子

eat();

signal(chopstick[(i+1)%5]); //释放右手边的筷子

signal(chopstick[i]); //释放左手边的筷子

signal(room); //退出房间释放信号量room

}

}

B．原理：仅当哲学家的左右两支筷子都可用时,才允许他拿起筷子进餐。   
方法1：利用AND 型信号量机制实现：根据课程讲述，在一个原语中，将一段代码同时需要的多个临界资源，要么全部分配给它，要么一个都不分配，因此不会出现死锁的情形。当某些资源不够时阻塞调用进程;由于等待队列的存在，使得对资源的请求满足FIFO 的要求， 因此不会出现饥饿的情形。   
伪码：

semaphore chopstick[5]={1，1，1，1，1};

void philosopher(int I)

{

while(true)

{

think();

Swait(chopstick[(I+1)]%5,chopstick[I]);

eat();

Ssignal(chopstick[(I+1)]%5,chopstick[I]);

}

}

方法2：利用信号量的保护机制实现。通过信号量mutex对eat（）之前的取左侧和右侧筷子的操作进行保护，使之成为一个原子操作，这样可以防止死锁的出现。   
伪码：

semaphore mutex = 1 ;

semaphore chopstick[5]={1，1，1，1，1};

void philosopher(int I)

{

while(true)

{

think();

wait(mutex);

wait(chopstick[(I+1)]%5);

wait(chopstick[I]);

signal(mutex);

eat();

signal(chopstick[(I+1)]%5);

signal(chopstick[I]);

}

}

C． 原理：规定奇数号的哲学家先拿起他左边的筷子,然后再去拿他右边的筷子;而偶数号的哲学家则相反.按此规定,将是1,2号哲学家竞争1号筷子,3,4号哲学家竞争3号筷子.即五个哲学家都竞争奇数号筷子,获得后,再去竞争偶数号筷子,最后总会有一个哲学家能获得两支筷子而进餐。而申请不到的哲学家进入阻塞等待队列，根FIFO原则，则先申请的哲学家会较先可以吃饭，因此不会出现饿死的哲学家。   
伪码：

semaphore chopstick[5]={1，1，1，1，1};

void philosopher(int i)

{

while(true)

{ //如果拿不到第一个筷子，就拿不到第二个

think();

if(i%2 == 0) //偶数哲学家，先右后左。

{

wait (chopstick[ i + 1 ] mod 5) ;

wait (chopstick[ i]) ;

eat();

signal (chopstick[ i + 1 ] mod 5) ;

signal (chopstick[ i]) ;

}

Else //奇数哲学家，先左后右

{

wait (chopstick[ i]) ;

wait (chopstick[ i + 1 ] mod 5) ;

eat();

signal (chopstick[ i]) ;

signal (chopstick[ i + 1 ] mod 5) ;

}

}

}

D．利用管程机制实现（最终该实现是失败的，见以下分析）：   
原理：不是对每只筷子设置信号量，而是对每个哲学家设置信号量。test()函数有以下作 用：   
a. 如果当前处理的哲学家处于饥饿状态且两侧哲学家不在吃饭状态，则当前哲学家通过 test()函数试图进入吃饭状态。   
b. 如果通过test()进入吃饭状态不成功，那么当前哲学家就在该信号量阻塞等待，直到其他的哲学家进程通过test()将该哲学家的状态设置为EATING。   
c. 当一个哲学家进程调用put\_forks()放下筷子的时候，会通过test()测试它的邻居， 如果邻居处于饥饿状态，且该邻居的邻居不在吃饭状态，则该邻居进入吃饭状态。   
由上所述,该算法不会出现死锁，因为一个哲学家只有在两个邻座都不在进餐时，才允 许转换到进餐状态。   
该算法会出现某个哲学家适终无法吃饭的情况，即当该哲学家的左右两个哲学家交替处在吃饭的状态的时候，则该哲学家始终无法进入吃饭的状态，因此不满足题目的要求。 但是该算法能够实现对于任意多位哲学家的情况都能获得最大的并行度，因此具有重要的意义。   
伪码：

#define N 5 /\* 哲学家人数\*/

#define LEFT (i-1+N)%N /\* i的左邻号码 \*/

#define RIGHT (i+1)%N /\* i的右邻号码 \*/

typedef enum { THINKING, HUNGRY, EATING } phil\_state; /\*哲学家状态\*/

monitor dp /\*管程\*/

{

phil\_state state[N];

semaphore mutex =1;

semaphore s[N]; /\*每个哲学家一个信号量，初始值为0\*/

void test(int i)

{

if ( state[i] == HUNGRY &&state[LEFT(i)] != EATING && state[RIGHT(i)] != EATING )

{

state[i] = EATING;

V(s[i]);

}

}

void get\_forks(int i)

{

P(mutex);

state[i] = HUNGRY;

test(i); /\*试图得到两支筷子\*/

V(mutex);

P(s[i]); /\*得不到筷子则阻塞\*/

}

void put\_forks(int i)

{

P(mutex);

state[i]= THINKING;

test(LEFT(i)); /\*看左邻是否进餐\*/

test(RIGHT(i)); /\*看右邻是否进餐\*/

V(mutex);

}

}

哲学家进程如下：

void philosopher(int process)

{

while(true)

{

think();

get\_forks(process);

eat();

put\_forks(process);

}

}