**OS第六次作业： 直接发助教邮箱：wth\_os@163.com**

**第一部分**

6.7 Describe how the Swap() instruction can be used to provide mutual exclusion that satisﬁes the bounded-waiting requirement.

代码如下:

void Swap(boolean \*a , boolean \*b){

    boolean temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// 解决无限等待问题，使用队列

int wait[n] =0; // 等待队列

bool look;

do{

    bool key = true;

    wait[i] = 1;     // 当前需要等待

    while(key==true && wait[i] ==1) Swap(&lock,&key)

    //  critical section

    j = (i+1) % n ;

    while( j !=i && wait[i] !=1 ) j = (j+1)%n;

    if(j==i) lock = False;

    else wait[j]=0;

    // remainder section

}while(true)

1. 上述代码新设置了一个 wait 数组， wait = 1表示当前进程在等待， wait =0 表示当前进程不需要等待。
2. 每个进程进入代码之后在 while 前将 wait 设为 1 表明 需要等待。进入等待队列
3. 当当前的 lock为 false是，经过 swap操作，最先到的进程能进入临界并同时将lock设为false.
4. 一般情况下在外面等待的进程 由于 wait[i]=1 并且 key =true 所以只能停在那里等待
5. 当进入临界区的进程执行完之后，就往后循环遍历wait数组。(按照先来后到的顺序),如果有进程在等待(wait=1)就将 wait[j] 设为0
6. 一旦 wait[i]被设为0，该等待的进程可以通过打破第二个条件进入临界区，同时由于lock还是true,所以其它进程进不来
7. 如果没找到说明没有进程在等待，将 lock 设为 false。下一次第一个到达的进程能够进入临界区。
8. 综上实现了 有限等待的互斥访问。

6.8 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

```c++

// 方案 1

class Server{

    count = new Semephore(n); // 计数型信号量

}

Server::create\_connection{

    // 创建连接

    count->P(); // P操作 或者写为 wait(count)

    create\_new\_connection() ; // 创建新的连接

}

Server::release\_connection{

    // 断开连接

    release\_one\_connection();

    count->V(); // V 操作 或者写为 signal(count)

}

// 方案1 可能导致进程卡在 P操作里 换一种写法

Class Server{

    mutex = new Semephore(1); // 互斥变量用来保护对conut的访问

    int count =n;

}

 Server::create\_connection{

    mutex->P(); // 访问 count

    // 临界区

    if(count>0){

        create\_new\_connection();

        count--;

    }

    mutex->V();    // 访问 count 完后曾

}

Server::release\_connection{

     mutex->P()；

    release\_one\_connection();

    count++;

     mutex->V()；

}

本题采用两种方案实现，一种直接使用 计数型信号量进行条件判断，一种则是使用互斥型信号量来控制对临界区的访问。((采用这种方式当server不能建立连接时)不会陷入等待)

6.X 假定一个阅览室最多可同时容纳n个人阅读，读者进入和离开阅览室时，都必须在阅览室门口的一个登记表上登记。假定每次只允许一个人登记和注销。请用PV操作编写读者进程的同步算法，并加以解释

// 阅览室问题

class Room{

    mutex = new Semephore(1);  // 控制对 seat变量的访问

    seat =n ;  //空位

}

Room::get\_in(){

    mutex->P();

    if(seat>0){

        //判断当前是否有座位

        seat--;

        login();   //登记

        reading();

    }

    // 如果没座位就直接退出去了

    mutex->V();

}

Room::get\_out(){

    mutex->P()

    seat++;

    logout(); // 注销

    mutex->V();

}

上述代码可以保证当没有作为的时候，进程不会卡在 get\_in函数不动，其余请看注释

\*6.11 The Sleeping-Barber Problem.A barbershop consists of a waiting room with n chairs and a barber room with one barber chair.

If there are not customers to be served, the barber goes to sleep.

If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.

If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

class BarberRoom{

    chairs   = n ;            //椅子数, 因为要实现没有椅子离开，所以不能用信号量

    mutex = new Semaphore() ; // 控制对 chairs的访问

    customers = new Semaphore(0) ;  // 顾客信号量，一开始没有顾客

    barber = new Semaphore(1) ;  // 理发师信号量

    is\_cut = new Semaphore(0) ;

}

BarberRoom::barber\_room{

    //理发室

    // 如果有顾客

    customers->P();

    barbering()   //理发

    barber->V() ;  //通知新的顾客可以来理发了

}

BarberRoom::waiting\_room{

    //等候室

    mutex->P();

    if(chairs>0){

        chairs--;

        customers->V();  // 顾客增加,同时如果此时理发师在睡觉，会唤醒理发师

        mutex->V();

        barber->P();    //排队理发

        // 起身去理发，把椅子空出来

        mutex->P();

        charirs++;

        mutex->V();

    }else{

        get\_out();

        mutex->V();

    }

}

**第二部分**

6.3 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

1. 忙等的含义是指 : 当有进程在临界区中时，其它进程必须在临界区外等待的现象。
2. 在操作系统中，等待除了忙等还有 死锁(deadlock)。(无限等待), 饥锇（长时间等待甚至可能无限等待）
3. 可以避免忙等，我们增加等待队列，在P()操作( 或者说 waiting 操作)中如果判断信号量的值小于一，那么进入等待队列中，进程进入 waiting 态让出CPU使用权。 在 signal()操作中，当等待队列里有进程的时候将其唤醒，进入ready()态等待被调度。通过这种方式解决忙等问题
4. 在多CPU环境，或者当忙等的时间很短的时候，我们不一定要避免忙等(甚至需要欢迎满等) 因为 切换 PCB的代价比忙等的代价更大。

6.4 Explainwhy spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems

1. 自旋锁，指的是可能引发忙等的信号量，在但CPU中，如果进程发生忙等(自旋)，那么就会浪费 CPU时间(分配的CPU时间中等待), 并且由于是单CPU，所以没有其它进程在执行，该进程在全部的CPU时间中必定都在等待。所以忙等不适合单CPU。

2. 而在多CPU中，虽然发生了忙等，但是其它进程可以在别的CPU上执行，进程执行完成之后，解除对临界区的使用，进程在本次的CPU时间中仍然可以进入临界区工作。同时进程切换需要 contex swap, 需要保存PCB和取出PCB，该代价比多CPU下忙等的代价大(如果切换了进程，可能还需要等待下一次调度。)因此在多CPU中经常使用忙等。

6.9 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

为了解释，给出wait 和 signal的代码如下

wait(semaphore \*S){

    S->value--;

    if(S->value<0){

        add this process to S->list;

        block();

    }

}

signal(semaphore \*S){

    S->value++;

    if(S->value <=0){

        remove a process P from S->list;

        wakeup(P);

    }

}

如图所示 如果不是原子操作。设想以下场景

1. 当前信号量值为1；
2. 有两个进程 P1 和 P2

1.此时 P1执行 wait操作， 实行到 S->value—操作后，发生了中断(可能是时间片用完了)

2.此时 P2 执行 signal 操作，执行 到 S->valuee++ 操作，发生了中断。

3 此时又轮到 P1执行，因为信号量已经被加1，所以P1进入了临界区。

4 P1进入临界区之后，信号量的值已经为1，所以P1之外的信号量也能进入临界区

5 综上 互斥访问无法实现