用信号量解决进程的同步与互斥探讨

2015/05/07 • IT技术 • 882 阅读 • 2 评论 • 临界资源, 操作系统, 进程

原文出处： whatbeg 的博客 欢迎分享原创到伯乐头条

现代操作系统采用多道程序设计机制，多个进程可以并发执行，CPU在进程之间来回切换，共享某些资源，提高了资源的利用率，但这也使得处理并发执行的多个进程之间的冲突和相互制约关系成为了一道难题。如果对并发进程的调度不当，则可能会出现运行结果与切换时间有关的情况，令结果不可再现，影响系统的效率和正确性，严重时还会使系统直接崩溃。就比如你只有一台打印机，有两个进程都需要打印文件，如果直接让他们简单地并发访问打印机，那么你很可能什么都打印不出来或者打印的文件是…anyway，我们需要增加一些机制来控制并发进程间的这种相互制约关系。

进程间通信的很多问题的根本原因是我们不知道进程何时切换。

概念

首先我们了解一下临界资源与临界区的概念：临界资源就是一次只允许一个进程访问的资源，一个进程在使用临界资源的时候，另一个进程是无法访问的，操作系统也不能够中途剥夺正在使用者的使用权利，正所谓“泼出去的女儿嫁出去的水”是也。即临界资源是不可剥夺性资源。那么临界区呢？所谓临界区就是进程中范文临界资源的那段程序代码，注意，是程序代码，不是内存资源了，这就是临界资源与临界区的区别。我们规定临界区的使用原则（也即同步机制应遵循的准则）十六字诀：“空闲让进，忙则等待，有限等待，让权等待” - strling。让我们分别来解释一下：

（1）空闲让进：临界资源空闲时一定要让进程进入，不发生“互斥礼让”行为。

（2）忙则等待：临界资源正在使用时外面的进程等待。

（3）有限等待：进程等待进入临界区的时间是有限的，不会发生“饿死”的情况。

（4）让权等待：进程等待进入临界区是应该放弃CPU的使用。

好了，我们进入下一部分。

进程间通常存在着两种制约关系：直接制约关系和间接制约关系，就是我们通常所说的进程的同步与互斥。顾名思义，一个是合作关系，一个是互斥关系。进程互斥说白了就是“你用的时候别人都不能用，别人用的时候，你也不能去用”，是一种源于资源共享的间接制约关系。进程同步指的是“我们大家利用一些共同的资源区，大家一起合作，完成某些事情，但是我在干某些小事的时候，可能要等到你做完另一些小事”，是一种源于相互合作的直接制约关系。两者区别在于互斥的进程间没有必然的联系，属于竞争者关系，谁竞争到资源（的使用权），谁就使用它，直到使用完才归还。就比如洗衣房的洗衣机这个资源，去洗衣的同学并不需要有必然联系，你们可以互不认识，但是谁竞争到洗衣机的使用权，就可以使用，直到洗完走人。而同步的进程间是有必然联系的，即使竞争到使用权，如果合作者没有发出必要的信息，该进程依然不能执行。就比如排队打水，即使排到你了，如果水箱没水了，你就打不了水，说明你和水箱是有着必然联系的，你得从它里面取水，你们是同步关系，你们合作完成“打水”这个过程。

那么先来讨论如何实现进程的互斥控制。有下列几种方法：严格轮换（每个进程每次都从头执行到尾，效率不高，可能等待很久），屏蔽中断（刚刚进入临界区时就屏蔽中断，刚要出临界区就打开中断），专用机器指令test_and_set,test_and_clear，加锁，软件方法，信号量机制。讲一下加锁和软件方法，加锁方法如下：设置一个锁标志K表示临界资源的状态，K=1表示临界资源正在被使用，K=0表示没有进程在访问临界资源。如果一个进程需要访问临界资源，那么先检查锁标志K：

```
if K == 1, 循环检测，直到K = 0
else if K == 0，设置锁标志为1，进入临界区
```

离开临界区时设置锁标志K为0. 软件方法类似，如爱斯基摩人的小屋协议，爱斯基摩人的小屋很小，每次只能容纳一个人进入，小屋内有一个黑板，上面标志这能够进入临界区的进程。若进程申请进入临界区，则先进入小屋检查黑板标志，如果是自己，那么离开小屋进入临界区，执行完后进入小屋修改黑板标志为其他进程，离开小屋。如果小屋黑板标志不是自己，那么反复进入小屋考察黑板标志是不是自己。这两种方法都实现了互斥访问，但是都违反了四条原则之一：让权等待，都需要不断的循环重复检测标志，霸占了CPU资源，不是很好的方法。

到后来，荷兰计算机科学家Dijkstra于1965年提出了解决进程同步与互斥问题的信号量机制，收到了很好的效果，被一直沿用至今，广泛应用与单处理机和多处理机系统以及计算机网络中。信号量机制就是说两个或者多个进程通过他们都可以利用的一个或多个信号来实现准确无误不冲突的并发执行。如果临界资源不够，就会有一个信号表示出来，如果进程此时想访问，那么就会阻塞到一个队列中，等待调度。当临界资源使用完毕，一个进程改变信号，并及时唤醒阻塞的进程，这就实现了进程间的同步和互斥问题。

信号量分为整型信号量，记录型信号量，AND信号量以及信号量集。最初的信号量就是整型信号量，定义信号量为一个整型变量，仅能通过两个原子操作P,V来访问，所谓原子操作就是指一组相联的操作要么不间断地执行，要么不执行。这两个操作又称为wait和signal操作或者down和up操作。之所以叫P,V操作是因为Dijkstra是荷兰人，P指的是荷兰语中的“proberen”，意为“测试”，而V指的是荷兰语中的“verhogen”，意为“增加”。最初P,V操作被描述为：

```
P(S): while (S≤0) {do nothing};
S=S-1;
```

```
V(S): S=S+1;
```

但是这样明显违反了“让权等待的原则”，后来发展为记录型信号量，记录型信号量的数据结构是一个两元组，包含信号量的值value和关于此信号量的阻塞队列Q，value具有非负初值，一般反映了资源的数量，只能由P,V操作改变其值。（还有另一种定义，信号量由value和P组成，value为信号量的值，P为指向PCB队列的指针）。

记录型信号量的P,V操作原语为：

```
P(S): S.value = S.value-1;
 if(S.value < 0)
 block(S,Q);
V(S): S.value = S.value + 1;
 if(S.value <= 0)
 wakeup(S,Q);
```

我们来详细解释一下这两个操作的含义：

首先，P操作，首先将S.value减1，表示该进程需要一个临界资源，如果S.value<0，那么说明原来的S.value <= 0，即已经没有资源可用了，于是将进程阻塞到与信号量S相关的阻塞队列中去，如果S.value<0,那么|S.value|其实就表示阻塞队列的长度，即等待使用资源的进程数量。然后，V操作：首先S.value加1，表示释放一个资源，如果S.value <= 0，那么说明原来的S.value < 0，阻塞队列中是由进程的，于是唤醒该队列中的一个进程。那么，为什么S.value > 0时不唤醒进程呢，很简单，因为阻塞队列中没有进程了。

P操作相当于“等待一个信号”，而V操作相当于“发送一个信号”，在实现同步过程中，V操作相当于发送一个信号说合作者已经完成了某项任务，在实现互斥过程中，V操作相当于发送一个信号说临界资源可用了。实际上，在实现互斥时，P,V操作相当于申请资源和释放资源。

我们将信号量初值设置为1时通常可实现互斥，因为信号量表示资源可用数目，互斥信号量保证只有一个进程访问临界资源，相当于只有一个访问权可用。设置为0或者N时可以用来实现同步。我们后面将会在生产者-消费者问题中看到这点。用P,V操作实现互斥类似于加锁的实现，在临界区之前加P操作，在临界区之后加V操作，即可互斥控制进程进入临界区，访问临界资源。记录型信号量由于引入了阻塞机制，消除了不让权等待的情况，提高了实现的效率。

经典问题

下面通过一些实例详细讲解如何使用信号量机制解决进程同步与互斥问题。先说明一条规律，即：同步与互斥实现的P,V操作虽然都是成对出现，但是互斥的P,V操作出现在同一个进程的程序里，而同步的P,V操作出现在不同进程的程序中。

问题1：生产者-消费者问题

经典的同步互斥问题，也称作“有界缓冲区问题”。具体表现为：

1.两个进程对同一个内存资源进行操作，一个是生产者，一个是消费者。

2.生产者往共享内存资源填充数据，如果区域满，则等待消费者消费数据。

3.消费者从共享内存资源取数据，如果区域空，则等待生产者填充数据。

4.生产者的填充数据行为和消费者的消费数据行为不可在同一时间发生。

生产者-消费者之间的同步关系表现为缓冲区空，则消费者需要等待生产者往里填充数据，缓冲区满则生产者需要等待消费者消费。两者共同完成数据的转移或传送。生产者-消费者之间的互斥关系表现为生产者往缓冲区里填充数据的时候，消费者无法进行消费，需要等待生产者完成工作，反之亦然。

既然了解了互斥与同步关系，那么我们就来设置信号量：

由于有互斥关系，所以我们应该设置一个互斥量mutex控制两者不能同时操作缓冲区。此外，为了控制同步关系，我们设置两个信号量empty和full来表示缓冲区的空槽数目和满槽数目，即有数据的缓冲区单元的个数。mutex初值为1，empty初值为n，即缓冲区容量，代表初始没有任何数据，有n个空的单元，类似的，full初值为0.

下面进行生产者-消费者行为设计：

```
void Productor() {
 while(1) {

//制造数据
 P(&empty);
 P(&mutex);

//填充数据
 V(&mutex);
 V(&full);
 }
}
void Consumer() {
 while(1) {
 P(&full);
 P(&mutex);

//消费数据
 V(&mutex);
 V(&empty);
 }
}
```

这样我们的分析也就完成了，http://www.cnblogs.com/whatbeg/p/4419979.html 这篇文章里有我用Windows API实现的用信号量实现生产者-消费者问题。

下面，问题来了，我们的生产者和消费者里面都有两个P,两个V操作，那么两个P操作可否调换顺序呢？V操作呢？想一想。

答案是P操作不可对换，V操作可以。为什么呢？想象一下这种情况，生产者执行P(mutex)把互斥量锁住，然后再P(empty),此时empty < 0，锁住，无法继续生产，等待消费者消费，消费者倒是也想消费，可是mutex被锁住了啊，于是两个人就等啊等，就成了等待戈多了。。但是V操作是可以随意调换的，因为V操作是解锁和唤醒，不会因为它锁住什么。

问题2：读者-写者问题

第二个经典问题是读者-写着问题，它为数据库的访问建立了一个模型。规则如下：

1.一个进程在读的时候，其他进程也可以读。

2. 一个进程在读/写的时候，其他进程不能进行写/读。

3. 一个进程在写的时候，其他进程不能写。

我们来分析他们的关系，首先，这个问题没有明显的同步关系，因为在这个问题里，读和写并不要合作完成某些事情。但是是有互斥关系的，写者和写者，写者和读者是有互斥关系的，我们需要设置一个mutex来控制其访问，但是单纯一个信号量的话会出现读者和读者的互斥也出现了，因为我们可能有多个读者，所以我们设置一个变量ReadCount表示读者的数量，好，这个时候，对于ReadCount又要实现多个读者对他的互斥访问，所以还要设置一个RC_mutex。这样就好了。然后是行为设计：

```
void Reader() {
 while(1) {
 P(&RC_mutex);
 rc = rc + 1;
 if(rc == 1) P(&mutex);
//如果是第一个读者，那么限制写者的访问
 V(&RC_mutex);

//读数据
 P(&RC_mutex);
 rc = rc - 1;
 if(rc == 0) V(&mutex);
//如果是最后一个读者，那么释放以供写者或读者访问
 V(&RC_mutex);
 }
}
void Writer() {
 while(1) {
 P(&mutex);

//写数据
 V(&mutex);
 }
}
```

其实，这个方法是有一定问题的，只要趁前面的读者还没读完的时候新一个读者进来，这样一直保持，那么写者会一直得不到机会，导致饿死。有一种解决方法就是在一个写者到达时，如果后面还有新的读者进来，那么先挂起那些读者，先执行写者，但是这样的话并发度和效率又会降到很低。有人提出了一种写者优先的解法，有点不好理解，这里给出实现：

```
//写者优先的读者-写者问题解法
Semaphore x = y = z = 1;
//x控制ReadCount的互斥访问，y控制WriteCount的互斥访问
Semaphore rsem = wsem = 1;
//rsem,wsem分别表示对读和写的互斥控制
int ReadCount = WriteCount = 0;
void Reader() {
 P(z);
//z保证写跳过读，做到写优先
 P(rsem);
//控制对读的访问，如果有写者，那么此处不成功
 P(x);
//对RC的互斥控制
 ReadCount++;
 if(ReadCount == 1) P(wsem);
//第一个读者出现后，锁住不让写
 V(x);
 V(rsem);
//释放读的访问，以使其他读者进入
 V(z);

//读数据...
 P(x);
 ReadCount--;
 if(ReadCount == 0) V(wsem);
//如果是最后一个读者，释放对写的信号
 V(x);
}
```

```
29  void Writer() {
30   P(y);
31   WriteCount++;
32   if(WriteCount == 1) P(rsem);
      V(y);
      P(wsem);

     //写数据...
      V(wsem);
      P(y);
      WriteCount--;
      if(WriteCount == 0) V(rsem);
      V(y);
     }
```

问题3：哲学家就餐问题

哲学家就餐问题描述如下：

有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，进餐完毕，放下筷子又继续思考。

约束条件
(1)只有拿到两只筷子时，哲学家才能吃饭。
(2)如果筷子已被别人拿走，则必须等别人吃完之后才能拿到筷子。
(3)任一哲学家在自己未拿到两只筷子吃饭前，不会放下手中拿到的筷子。
(4)用完之后将筷子返回原处

分析：筷子是临界资源，每次只被一个哲学家拿到，这是互斥关系。如果筷子被拿走，那么需要等待，这是同步关系。

容易想到一种错误的解法，所以设置一个信号量表示一只筷子，有5只筷子，所以设置5个信号量，哲学家每次饥饿时先试图拿左边的筷子，再试图拿右边的筷子，拿不到则等待，拿到了就进餐，最后逐个放下筷子。这种情况可能会产生死锁，因为我们不知道进程何时切换（这也是很多IPC问题的根本原因），如果5个哲学家同时饥饿，同时试图拿起左边的筷子，也很幸运地都拿到了，那么他们拿右边的筷子的时候都会拿不到，而根据第三个约束条件，都不会放下筷子，这就产生了死锁。《现代操作系统》中记载的一种解法是仅当一个哲学家左右的筷子都可用时，才拿起筷子，将“试图获取两个筷子”作为临界资源，用一个互斥量mutex实现对其的互斥控制，然后用n个变量记录哲学家的状态（饥饿，进餐，思考<可有可无，因为除了前两者以外只会思考>），然后用一个同步信号量数组，每个信号量对应一个哲学家，来保证哲学家得不到自己所需筷子的时候阻塞。算法如下：

```
#define N           5                   /* 哲学家数目 */
#define LEFT        (i+N-1)%N           /* i 的左邻居编号 */
#define RIGHT       (i+1)%N             /* i 的右邻居编号 */
#define THINKING    0                   /* 哲学家在思考 */
#define HUNGRY      1                   /* 哲学家试图拿起叉子 */
#define EATING      2                   /* 哲学家进餐 */
typedef int semaphore;                  /* 信号量是一种特殊的整型数据 */
int state[N];                           /* 数组用来跟踪记录每位哲学家的状态 */
semaphore mutex = 1;                    /* 临界区的互斥 */
semaphore s[N];                         /* 每个哲学家一个信号量 */

void philosopher(int i)                 /* i：哲学家编号，从0到N-1 */
{
    while (TRUE) {                      /* 无限循环 */
    }
}

void take_forks(int i)                  /* i：哲学家编号，从0到N-1 */
{
    down(&mutex);                       /* 进入临界区 */
    state[i] = HUNGRY;                  /* 记录哲学家i处于饥饿的状态 */
    test(i);                            /* 尝试获取2把叉子 */
    up(&mutex);                         /* 离开临界区 */
    down(&s[i]);                        /* 如果得不到需要的叉子则阻塞 */
}

void put_forks(i)                       /* i：哲学家编号，从0到N-1 */
{
    down(&mutex);                       /* 进入临界区 */
    state[i] = THINKING;                /* 哲学家已经就餐完毕 */
    test(LEFT);                         /* 检查左边的邻居现在可以吃吗 */
    test(RIGHT);                        /* 检查右边的邻居现在可以吃吗 */
    up(&mutex);                         /* 离开临界区 */
}

void test(i)                            /* i：哲学家编号，从0到N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```


还有一种解法是让奇数号与偶数号的哲学家拿筷子的先后顺序不同，以破坏环路等待条件。还可以只允许4个哲学家同时进餐（4个人都拿起一只筷子的时候，第5个人不能再拿筷子，这样就会空出一只筷子）

例题分析

至此，我们已经可以总结出一点用信号量解决同步互斥问题的基本规律和一般步骤：

（1）分析各进程间的制约关系，从而得出同步与互斥关系

（2）根据（1）中的分析，设置信号量

（3）编写伪代码，实施P,V操作

同步：多个进程在执行次序上的协调，相互等待消息

互斥：对临界资源的使用

要注意的是，虽然P,V操作在每一个进程中都是成对出现的，但不一定是针对一个信号量。互斥信号量的P,V操作总是出现在一个进程中的临界区的前后，而同步信号量的P,V操作总是出现在具有同步关系的两个进程中，需要等待消息的一方执行P操作，发出消息的一方执行V操作。

下面通过诸多例题来熟悉，掌握及训练用信号量解决同步与互斥问题的一般方法。

问题4：放水果问题

桌上有一空盘，最多允许存放一只水果。爸爸可向盘中放一个苹果或放一个桔子。

儿子专等吃盘中的桔子，女儿专等吃苹果。

试用P、V操作实现爸爸、儿子、女儿三个并发进程的同步。

分析：临界资源是盘子，放的时候不能取，取的时候不能放，取的时候不能再取。同步关系：爸爸与盘子为空，儿子与盘中有桔，女儿与盘中有苹果。

所以设置一个mutex互斥信号量来控制对盘子的访问，用empty，orange，apple分别代表以上同步关系。程序如下：

```
Semaphore mutex = 1;
Semaphore empty = 1, orange = apple = 0;
father:
 while(1) {
 P(empty);
 P(mutex);

//放入水果
 V(mutex)
 if(放入桔子) V(orange);
 else V(apple);
 }
son:
 while(1) {
 P(orange)
 P(mutex)

//取桔子
 V(mutex);
 V(empty);
 }

daughter:
 while(1) {
 P(apple)
 P(mutex)

//取苹果
 V(mutex);
 V(empty);
 }
```

问题5：读文件问题

四个进程A、B、C、D都要读一个共享文件F，系统允许多个进程同时读文件F。但限制是进程A和进程C不能同时读文件F，进程B和进程D也不能同时读文件F。为了使这四个进程并发执行时能按系统要求使用文件，现用P、V操作进行管理。

分析：互斥关系：A和C读文件时互斥，B和D读文件时互斥，没有同步关系。

所以设置两个互斥信号量：AC_mutex,BD_mutex即可。伪代码如下：

```
Semaphore AC_mutex = BD_mutex = 1;
A:
 while(1) {
 P(AC_mutex);

//read F
 V(AC_mutex);
 }
B:
 while(1) {
 P(BD_mutex);

//read F
 V(BD_mutex);
 }
C:
 while(1) {
 P(AC_mutex);

//read F
 V(AC_mutex);
 }
D:
 while(1) {
 P(BD_mutex);

//read F
 V(BD_mutex);
 }
```

问题6：阅览室问题 / 图书馆问题

有一阅览室，读者进入时必须先在一张登记表上进行登记，该表为每一座位列一表目，包括座号和读者姓名。读者离开时要消掉登记信号
，阅览室中共有100个座位。用PV操作控制这个过程。

分析：

由于每个读者都会进行一样的操作：登记->进入->阅读->撤销登记->离开，所以建立一个读者模型即可。

临界资源有：座位，登记表

读者间有座位和登记表的互斥关系，所以设信号量empty表示空座位的数量，初始为100，mutex表示对登记表的互斥访问，初始为1。

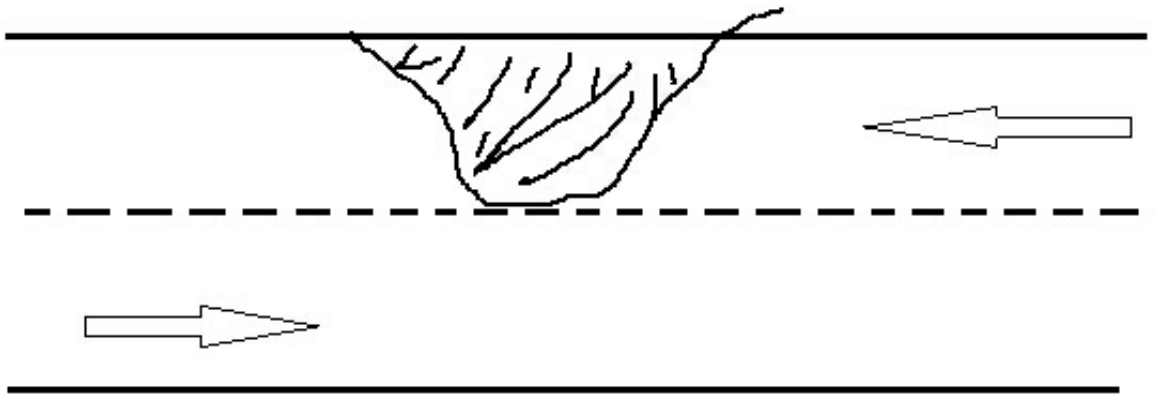
P,V操作如下：

```
Semaphore mutex = 1, empty = 100;
Reader():
While(true) {
 P(empty)
//申请空座位
 P(mutex)
//申请登记表

//登记
 V(mutex)
//释放登记表

//进入阅读
 P(mutex)
//申请登记表

//撤销登记
 V(mutex)
```

```
//释放登记表
 V(empty)
//释放座位
}
```

问题7：单行道问题

一段双向行驶的公路，由于山体滑坡，一小段路的一般车道被阻隔，该段每次只能容纳一辆车通过，一个方向的多个车辆可以紧接着通过，试用P,V操作控制此过程。



单行道问题

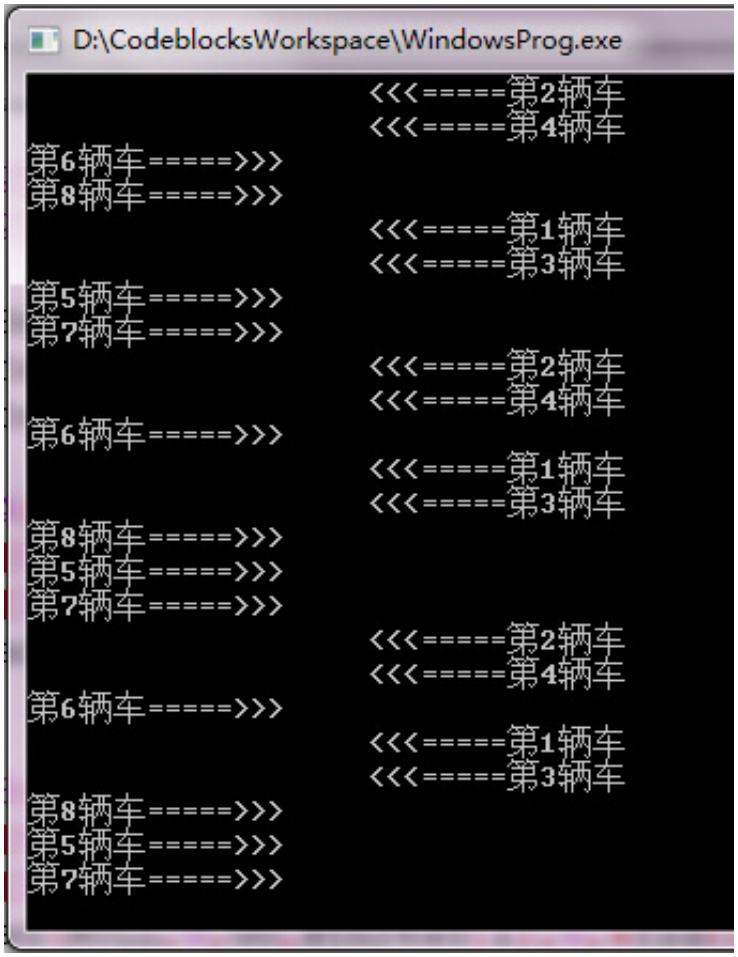
分析：

临界资源为一半被阻隔的一小段区域，所以需要一个mutex来控制每个方向第一辆车通过该路段的访问，类似于读者-写者问题，车辆从两边通过相当于两个读者，我们设立两个计数器A和B分别代表两个方向的汽车数量，还要设置两个信号量A_mutex和B_mutex来实现对计数器的互斥访问。于是程序如下（PV操作包含其中）：

```
#include <Windows.h>
#include <stdio.h>
#define N 100
#define TRUE 1
typedef int Semaphore;
Semaphore A = 0, B = 0;
HANDLE mutex,A_mutex,B_mutex;
void down(HANDLE handle) {
 WaitForSingleObject(handle, INFINITE);
}
void up(HANDLE handle) {
 ReleaseSemaphore(handle, 1, NULL);
}
DWORD WINAPI Come(LPVOID v) {
int item;
while(TRUE) {
down(A_mutex);
 if(A == 0) down(mutex);
 A = A+1;
 up(A_mutex);

//自东向西通过该路段
 printf(" <<<=====第%s辆车\n",(char *)v);
 down(A_mutex);
 A = A-1;
 if(A == 0) up(mutex);
 up(A_mutex);
Sleep(2000);
 }
 return 1;
}
```

```
DWORD WINAPI Go(LPVOID v) {
while(TRUE) {
down(B_mutex);
 if(B == 0) down(mutex);
 B = B+1;
 up(B_mutex);

//自西向东通过该路段
 printf("第%s辆车=====>>>\n",(char *)v);
 down(B_mutex);
 B = B-1;
 if(B == 0) up(mutex);
 up(B_mutex);
Sleep(2000);
 }
 return 1;
}
int main()
{
 DWORD Tid;
 char AThread[12][10];
 char BThread[12][10];
mutex = CreateSemaphore(NULL, 1, 1, NULL);
 A_mutex = CreateSemaphore(NULL, 1, 1, NULL);
 B_mutex = CreateSemaphore(NULL, 1, 1, NULL);
for(int i=0;i<4;i++) {
 AThread[i][0] = i+1+'0';
 AThread[i][1] = '';
 CreateThread(NULL,0,Come,AThread[i],0,&Tid);
 }
for(int i=4;i<8;i++) {
 BThread[i][0] = i+1+'0';
 BThread[i][1] = '';
 CreateThread(NULL,0,Go,BThread[i],0,&Tid);
 }
Sleep(20000);
 return 0;
}
```

运行结果：



从其中可以看出，车辆正常交替顺序通过该路段。数字重复出现是因为线程被重复地调度执行。

问题8：理发师问题

理发店理有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子 如果没有顾客，理发师便在理发椅上睡觉。 一个顾客到来时，它必
须叫醒理发师 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。用PV操作管理该过程。

分析：

法1：首先设置一个count表示等待的人数（包括理发椅上的那个人），初值为0，以供后来者判断是否应该离开。同时对count的访问要保证互斥，所以设置mutex信号量来保证互斥，初值为1。

临界资源：凳子，理发椅。 分别设置waitchair,barchair信号量，初值分别为n和1，表示临界资源数量。

同步关系：顾客和理发师之间有同步关系，用ready和done信号量来表示，初值均为0，ready表示顾客有没有准备好，done表示理发师是否完成一次理发。

注意：并非每一个进程都需要while(1)无限循环，比如此例，顾客剪完一次头发就走了，不可能马上再来剪，而以前的生产者-消费者不同，他们都是可以不断生产消费的。

写出P,V操作如下：

```
1 Semaphore waitchair = n;
2 Semaphore barchair = 1;
3 Semaphore ready = done = 0;
```

```
int count = 0;
Semaphore mutex = 1;
barber:
 while(1) {
 P(ready);
 理发
 V(done);
 }
consumer:
 P(mutex);
 if(count <= n) {
 count = count + 1;
 V(mutex);
 }
 else {
 V(mutex);
 离开
 }
 P(waitchair);
 P(barchair);
 V(waitchair);
//离开等待椅去理发椅需要释放等待椅!
 V(ready);
//准备好了
 P(done);
//等待理发完成
 V(barchair);
 P(mutex);
 count = count - 1;
 V(mutex);
 离开
```

法2：将凳子和理发椅看做同一种资源，因为只要理发椅空就一定会有人凑上去，所以相当于每个位置都是理发椅，理发师只需要去每个有人的座位理发即可。

还是设置count表示正在理发店中的人数，以便决定后来者是否离开。

同步关系仍用ready和done来表示。

算法：

```
Semaphore ready = done = 0;
int count = 0;
Semaphore mutex = 1;
barber:
 while(1) {
 P(ready);
 理发
 V(done);
 }
consumer:
 P(mutex);
 if(count <= n) {
 count = count + 1;
 V(mutex);
 }
 else {
 V(mutex);
 离开
 }
 V(ready);
//准备好了
 P(done);
//等待理发完成
 P(mutex);
//也可由理发师来做count-1的操作
 count = count - 1;
 V(mutex);
 离开
```