1. 设有两个优先级相同的进程 T1，T2 如下。令信号量 S1，S2 的初值为 0，已知 z=2，试问 T1，T2 并发运行结束后 x=?y=?z=?

```
步骤    线程 T1 线程 T2
1       y:=1;   x:=1;
2       y:=y+2; x:=x+1;
3       V(S1);  P(S1);
4       z:=y+1; x:=x+y;
5       P(S2);  V(S2);
6       y:=z+y; z:=x+z;
```

首先分析T1和T2两个线程并发运行的要求，其中线程T1的5号指令要在线程T2的5号指令执行完毕后执行，线程T2的3号指令要在线程T1的3号指令执行完毕后执行。

由于两个线程之间的1/2条指令执行互不干扰，所以从第3条指令开始分析：
在此时有$y=3, x=2$。可以观察到对于x、y、z的赋值指令一共有四条，不妨令T1的第i条指令为A_i，T2的第i条指令为B_i。现在进行分类讨论四条指令的顺序：(A_4, A_6, B_4, B_6)

如果仅考虑单一进程的运行顺序，则一共有如下排序顺序：
A_4, A_6, B_4, B_6
A_4, B_4, A_6, B_6
A_4, B_4, B_6, A_6
B_4, B_6, A_4, A_6
B_4, A_4, B_6, A_6
B_4, A_4, A_6, B_6
由于信号量的存在，可以发现B_3会比A_3后执行，A_5会比B_5后执行。因此执行了A_4不可能立即执行A_6，因为B_5没有执行(一旦执行则B_4先执行)。

剩下五种运行顺序的运行结果如下：
A_4, B_4, A_6, B_6 结果为 $x=5, y=7, z=9$
B_4, A_4, A_6, B_6 结果为 $x=5, y=7, z=9$
A_4, B_4, B_6, A_6 结果为 $x=5, y=12, z=9$
B_4, A_4, B_6, A_6 结果为 $x=5, y=12, z=9$
B_4, B_6, A_4, A_6 结果为 $x=5, y=7, z=4$
因此T1，T2并发运行后的结果可能有三种，为：
$x=5, y=7, z=9$
$x=5, y=12, z=9$
$x=5, y=7, z=4$

2. 银行有 n 个柜员,每个顾客进入银行后先取一个号,并且等着叫号,当一个柜员空闲后,就叫下一个号.请使用 PV 操作分别实现:

//顾客取号操作 Customer_Service
//柜员服务操作 Teller_Service

为了明确整个流程，我认为整个银行有k个取号机，n个柜员一人一个叫号机。在此假定下，我给出全流程设计：

互斥资源：取号机，互斥信号量picker
同步资源：等待的客户数量和空闲的柜员数量waiting_num、available。

```
atomic_int picker      = k; //取号机信号量
atomic_int waiting_num = 0; //现在取了号等待的客户数量
atomic_int available   = n; //现在空闲的柜员数
void Customer_Service(){
    P(picker);//使用一台取号机
    V(picker);//使用完毕

    V(waiting_num);//等待叫号
    P(available);//被叫号
}
void Teller_Service(){
    while(SHANGBAN){//上班期间一直等待叫号
        P(waiting_num);//等待客户
        do_service();//叫号成功，开始服务
        V(available);
    }
}
```

3. 多个线程的规约(Reduce)操作是把每个线程的结果按照某种运算(符合交换律和结合律)两两合并直到得到最终结果的过程。
   试设计管程 monitor 实现一个 8 线程规约的过程，随机初始化 16 个整数，每个线程通过调用 monitor.getTask 获得 2 个数，相加后，返回一个数 monitor.putResult ，然后再getTask( ) 直到全部完成退出，最后打印归约过程和结果。
   要求: 为了模拟不均衡性，每个加法操作要加上随机的时间扰动，变动区间 1~10ms。
   提示: 使用 pthread_系列的 cond_wait, cond_signal, mutex 实现管程
   使用 rand( )函数产生随机数，和随机执行时间。

在具体实现中，管程会存需要规约的数，如果一些数因为被其它线程拿走规约，导致管程存储的数少于两个，则会进行wait操作，在putResult阶段执行signal操作唤醒。除此之外，所有有关管程修改的操作都必须上锁。

实现代码：

```c
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdatomic.h>
#define TASKNUM (16)
typedef struct STA{
    int sum;
    struct STA * nxt;
}STA;
int GetTop(STA *S){
    if(S->sum == 0){
        return 0;
    }
    else{
        return S->nxt->sum;
    }
}
void Pop(STA *S){
    if(S->nxt == NULL)return;
    S->nxt = S->nxt->nxt;
    (S->sum) --;
}
void Ins(STA *S,int sum){
    STA * now = (STA *)malloc(sizeof(STA));
    now->sum = sum;
    now->nxt = S->nxt;
    S->nxt   = now;
    (S->sum)++;
}
typedef struct {
    STA *S;
    pthread_mutex_t mutex;
    pthread_cond_t  cond;
    int remind;
}Monitor_NODE;
void Moniter_init(Monitor_NODE * Monitor){
    pthread_mutex_init(&(Monitor->mutex),NULL);
    pthread_cond_init (&(Monitor->cond) ,NULL);
    Monitor->remind = TASKNUM;

    Monitor->S = (STA *)malloc(sizeof(STA));

    Monitor->S->nxt = NULL;
    Monitor->S->sum = 0;

    for(int i = 0; i<TASKNUM; i++){
```

```
        int x = rand()%20;
        Ins(Monitor->S,x);
    }
}

typedef struct {
    int first,second,a,b;
}RETNODE;


RETNODE Monitor_getTASK(Monitor_NODE * Monitor){
    RETNODE ret;
    pthread_mutex_lock(&(Monitor->mutex));
    if((Monitor->remind) == 1){
        pthread_mutex_unlock(&(Monitor->mutex));
        ret.first = 0;
        ret.second= 0;
        ret.a=ret.b=0;
        return ret;
    }
    (Monitor->remind)--;
    while(Monitor->S->sum < 2){
        pthread_cond_wait(&(Monitor->cond),&(Monitor->mutex));
    }
    int a = GetTop(Monitor->S);
    Pop(Monitor->S);
    int b = GetTop(Monitor->S);
    Pop(Monitor->S);
    pthread_mutex_unlock(&(Monitor->mutex));
    ret.first = 0;
    ret.second= 1;
    ret.a     = a;
    ret.b     = b;
    return ret;
}

void Monitor_putResult(Monitor_NODE * Monitor,int x){
    pthread_mutex_lock(&(Monitor->mutex));
    Ins(Monitor->S,x);
    if(Monitor->S->sum == 2){
        pthread_cond_signal(&(Monitor->cond));
    }
    pthread_mutex_unlock(&(Monitor->mutex));
}

void *worker(void *arg){
    long * argnow = arg;
    long id = argnow[0];
    Monitor_NODE * Monitor = argnow[1];
    while(1){
        RETNODE now = Monitor_getTASK(Monitor);
```

```
        if(now.second == 0){
            printf("thread %d finished\n",id);
            return NULL;
        }
        else{
            int a = now.a;
            int b = now.b;
            usleep(rand()%10);
            int c = a + b;
            printf("thread %d %d+%d=%d\n",id,a,b,c);
            Monitor_putResult(Monitor,c);
        }
    }
}
pthread_t thread[9];
atomic_long arg[8][3];
int main(){
    srand(time(0));
    Monitor_NODE* Monitor;
    Monitor = malloc(sizeof(Monitor_NODE));
    Moniter_init(Monitor);
    STA * now = Monitor->S;
    int Target = 0;
    for(now=now->nxt;now;now=now->nxt){
        printf("%d ",now->sum);
        Target += now->sum;
    }
    puts("");
    puts("============== TASK begin ==============");

    for(atomic_long i=0;i<8;i++){
        arg[i][0] = i;
        arg[i][1] = Monitor;
        pthread_create(&(thread[i]),NULL,worker,arg[i]);
        // usleep(1);
    }

    for(int i=0;i<8;i++){
        pthread_join((thread[i]),NULL);
    }
    now = Monitor->S;
    printf("Reduce sum = %d Target = %d\n",now->nxt->sum,Target);
    return 0;
}
```

运行效果：

```
12 10 4 2 11 15 9 14 7 14 10 19 14 13 13 14
============== TASK begin ==============
thread 0 12+10=22
thread 1 4+2=6
thread 4 14+10=24
thread 2 14+7=21
thread 3 11+15=26
thread 0 22+9=31
thread 2 21+13=34
thread 4 24+14=38
thread 3 26+13=39
thread 1 6+19=25
thread 0 31+14=45
thread 2 39+34=73
thread 2 finished
thread 3 finished
thread 5 38+25=63
thread 4 finished
thread 0 finished
thread 1 45+73=118
thread 5 finished
thread 1 finished
thread 6 118+63=181
thread 6 finished
thread 7 finished
Reduce sum = 181 Target = 181
```