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=？

线程T1 线程T2

y:=1; 1 x:=1; 5

y:=y+2; 2 x:=x+1; 6

V(S1); P(S1);

z:=y+1; 3 x:=x+y; 7

P(S2); V(S2);

y:=z+y; 4 z:=x+z; 8

注：请分析所有可能的情况，并给出结果与相应执行顺序。

可能的情况，执行顺序有：

1）V（S1），P（S1），P（S2），V（S2）

2）V（S1），P（S1），V（S2），P（S2）

3）V（S1），P（S2），P（S1），V（S1）

4）P（S1），V（S1），P（S2），V（S2）

5）P（S1），V（S1），V（S2），P（S2）

由于S1信号量的存在，语句7必定在1，2执行之后才能执行。由于S2信号量的存在，语句4必定在7执行之后才能执行。

1，2，5，6为不相关语句。3与5，6，7为不相关语句。7与1，2相关，但7必在1，2之后执行。4与8相关。8与3相关。故导致最终值不一样的决定性语句顺序有两处：4与8之间的顺序和3与8之间的顺序。

考虑到4必然在3之后执行，他们之间的顺序有三种：3，4，8；3，8，4；8，3，4。有如下执行顺序：

1）1，2，3，5，6，7，4，8

2）1，2，3，5，6，7，8，4

3）1，2，5，6，7，8，3，4

首先，y = 1 , x = 1,随后y = 3 , x = 2。

1. 顺序：z = 4，x = 2 + 3 = 5，y = 4 + 3 = 7，z = 4 + 5 = 9。最终x = 5，y = 7，z = 9。
2. 顺序：z = 4，x = 2 + 3 = 5，z = 4 + 5 = 9，y = 3 + 9 = 12。最终x = 5，y = 12，z = 9。
3. 顺序：x = 2 + 3 = 5，z = 5 + 2 = 7，z = 3 + 1 = 4，y = 4 + 3 = 7。最终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

顾客取号之后，等待；柜员空闲之后发出服务信号；顾客接收到服务信号后接受服务；服务结束之后柜员发出空闲服务信号。

由于取号并进入等待队列和服务之间并没有同步关系，所以只需要两个信号量，一个控制取号时排队的互斥操作(不可以操作人员一边在拿号，并且顾客也在取号s)，另一个控制等待中的顾客和服务的同步关系

Teller\_Service(int i) //表示第几个服务人员

{

while(1)

{

p(customer\_count);

p(queue\_mutex);

从队头取一个号码；

v(queue\_mutex);

服务；

}

}

Customer\_Service()

{

while(1)

if(customer\_come){

p(queue\_mutex);

取号,入队；

v(queue\_mutex);

v(customer\_count);

}

}

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。

代码：

#include <sys/types.h>

#include <sys/socket.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <errno.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#include <stdlib.h>

//gcc -o a a.c -lpthread

pthread\_mutex\_t clifd\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t clifd\_cond = PTHREAD\_COND\_INITIALIZER;

int number[16] = {0};

int top = 15;

int deadlock = 0;

int length = 7;

int exitn = 0;

void gettask(int a)

{

int x,y;

int i;

while(top > 0)

{

pthread\_mutex\_lock(&clifd\_mutex);

/\*

if(first != 1)

pthread\_cond\_wait(&clifd\_cond, &clifd\_mutex);\*/

if(deadlock != length)

{

deadlock++;

printf("I am thread %u\n",a);

pthread\_cond\_wait(&clifd\_cond, &clifd\_mutex);

deadlock--;

}

printf("I am here! thread = %u \n",a);

if(top == 0)

{

printf("Thread %u is exiting!\n",a);

length--;

pthread\_cond\_signal(&clifd\_cond);

pthread\_mutex\_unlock(&clifd\_mutex);

pthread\_exit(NULL);

}

x = number[0];

y = number[1];

usleep(rand()%90000 + 10000);

for(i = 2; i <= top; i++)

number[i - 1] = number[i];

top--;

number[0] = x + y;

for(i = 0; i <= top; i++)

printf("%d ",number[i]);

printf("\n");

pthread\_mutex\_unlock(&clifd\_mutex);

pthread\_cond\_signal(&clifd\_cond);

sched\_yield();

}

pthread\_mutex\_lock(&clifd\_mutex);

printf("Thread %u is exiting!\n",a);

length--;

pthread\_cond\_signal(&clifd\_cond);

pthread\_mutex\_unlock(&clifd\_mutex);

pthread\_exit(NULL);

}

void thread()

{

int a;

a = pthread\_self();

gettask(a);

}

void main()

{

pthread\_t t1,t2,t3,t4,t5,t6,t7,t8;

int err;

int i;

err = pthread\_create(&t1,NULL,(void\*)thread,NULL);

if(err != 0) exit(-1);

err = pthread\_create(&t2,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t3,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t4,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t5,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t6,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t7,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

err = pthread\_create(&t8,NULL,(void\*)thread,NULL);

if(err != 0 ) exit(-1);

srand((unsigned int)time(0));

for(i = 0; i < 16; i++)

{

number[i] = rand()%10 + 1;

printf("%d ",number[i]);

}

printf("\n");

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

pthread\_join(t3,NULL);

pthread\_join(t4,NULL);

pthread\_join(t5,NULL);

pthread\_join(t6,NULL);

pthread\_join(t7,NULL);

pthread\_join(t8,NULL);

return ;

}

分析：

cond\_wait的使用方式：

pthread\_mutex\_lock(&qlock);    /\*lock\*/

pthread\_cond\_wait(&qready, &qlock); /\*block-->unlock-->wait() return-->lock\*/

pthread\_mutex\_unlock(&qlock); /\*unlock\*/

qlock是线程间互斥操作的互斥锁，qready是条件变量。

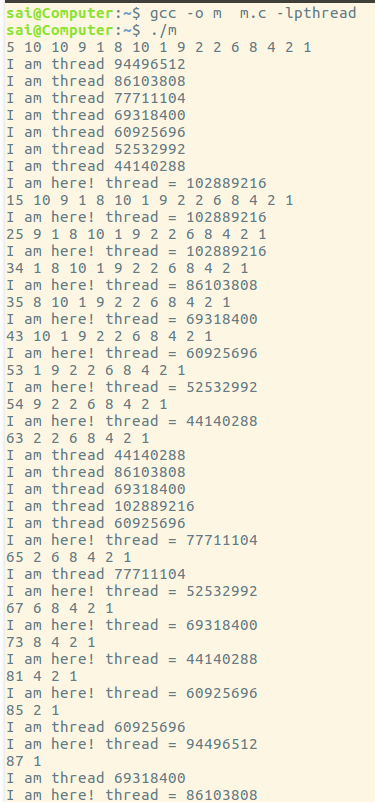
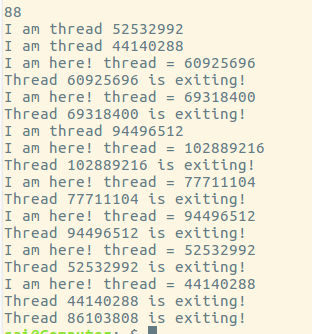
pthread\_cond\_wait的操作中蕴含着三步操作：释放锁，阻塞任务，获得锁。

运行逻辑如下：在配合while使用的线程中，为避免多线程访问临界区，线程先获得锁，然后判断while条件，执行pthread\_cond\_wait：1）释放锁 2）阻塞任务 3）获得锁 ，循环到不满足while条件为止，随后执行临界区的操作，最后释放锁，执行pthread\_cond\_signal。

另外，可能存在有的线程还在被阻塞，而已经有线程满足条件退出的情况。该情况下，可能出现signal的数量不足，导致不能释放所有的被阻塞线程，从而出现永久性阻塞的情况。解决办法：1）使用pthread\_cond\_broadcast函数，对所有被阻塞线程播报。

2）设置一个退出变量，每有一个进程退出就减一，如果变量不为0，就反复执行pthread\_cond\_signal，直到变量的值降为0。

运行结果：



每个线程在运行的过程中将处理后的数组打印出来，并且在wait之前和signal获得锁之后打印自己的线程号。退出的时候也打印自己的线程号。为避免任务在一个线程上运行完，使用了sched\_yield函数，在每一次计算结束循环之后进行一次线程调度。