### 第一题：

写一个程序，生成一个从1到1000万的整数数组，然后在程序中起两个线程（采用pthread API）分别计算数组奇数部分和偶数部分的和，并打印出总的和。分别在单核和双核系统上运行该程序，计算加速比（即程序在单核和双核系统上的运行时间比），作业中提供程序源码，及运行命令和结果，并解释加速比是否合理。

程序源码：

#include"stdio.h"

#include"pthread.h"

#include"semaphore.h"

#include<time.h>

#include<stdlib.h>

#include<sys/time.h>

#include<unistd.h>

#define MAX 10

static int datbuf[10] = {0};

static int n;

static int number[10000000];

sem\_t sem1,sem2;

void \*do\_1(void \*pvoid)

{

int i,j;

int sum=0;

j = 1000;

//sem\_wait(&sem1);

while(j>0)

{

sum = 0;

for(i = 1; i < 10000000; i = i + 2)

sum = sum+number[i];

j--;

}

// printf("The sum of odd part is %d\n",sum);

//sem\_post(&sem2);

pthread\_exit(NULL);

}

void \*do\_2(void \*pvoid)

{

int i,sum,j;

j = 0;

//sem\_wait(&sem2);

j = 1000;

while(j>0)

{

sum = 0;

for(i = 0; i < 10000000; i = i + 2)

sum = sum+number[i];

// printf("The sum of even part is %d\n",sum);

j--;

}

//sem\_post(&sem1);

pthread\_exit(NULL);

}

int main()

{

pthread\_t tid1,tid2;

int ret;

int i;

struct timeval tv\_begin, tv\_end;

struct timezone tz;

ret = sem\_init(&sem1,0,1);

ret += sem\_init(&sem2,0,0);

if(ret != 0)

{

printf("Any semaphore inittialization failed\n");

return ret;

}

srand((unsigned int)time(NULL));

for(i = 0; i < 10000000; i++)

{

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

}

pthread\_create(&tid1,NULL,&do\_1,NULL);

pthread\_create(&tid2,NULL,&do\_2,NULL);

// gettimeofday(&tv\_begin,&tz);

pthread\_join(tid1,NULL);

pthread\_join(tid2,NULL);

//gettimeofday(&tv\_end,&tz);

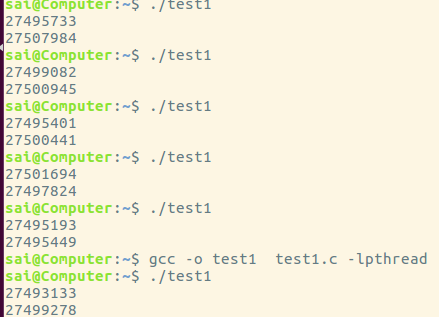
//printf("%.5f\n",(tv\_end.tv\_sec-tv\_begin.tv\_sec));

//printf("%.5f\n",(tv\_end.tv\_usec-tv\_begin.tv\_usec));

return 0;

}

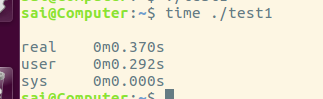
首先，为测试运行结果的正确性，先随机给数组中的数值赋为1到10之间的随机数后打印求和结果，多次重复运行程序结果如图：

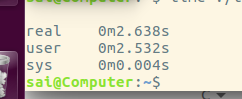


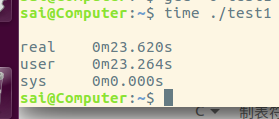
可以看得出两个线程确实在随机的程度上有效工作求和了。

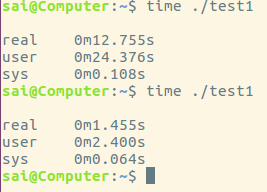
接下来对程序进行修改，使每个进程内部求和的操作循环多次，在外部使用time工具计算main函数总的运行时间。

结果如下：

单核，进程内部运行1次。

单核，进程内部运行100次。由此可见，单次运行时程序有0.35s左右的时间用于进行main函数中的其他操作，剩余0.023秒左右用来执行其他进程。

单核，运行1000次。基本可以确定，两个进程各执行一次所耗cpu时间为0.023秒。下面我们测试双核：

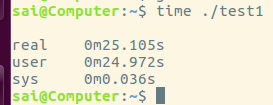
双核，运行1000次。

双核，运行100次。

由此看来，双核两个进程各执行一次所耗的真实时间约为0.0125秒，所耗用户时间约为0.023秒。

用户时间之所以与真实时间不一致，是因为它所体现的就是程序按单内核全部运行下来所需要的时间。在这里，其大小近似等于真实时间的两倍。

由此，加速比等于。考虑到部分语句不可以并行操作，再加之调研得到平均双核的效率较单核提升60%左右，该加速比较为合理。

另外，我对源代码中灰色部分进行了修改，去除了注释号，即加入sem\_wait和sem\_post，控制信号量，这样处理后程序变难以并行运行。事实证明也是如此，再次调用双核运行该程序，所用时间与单核相差无几。

### 第二题：

写一个两线程程序,两线程同时向一个数组分别写入1000万以内的奇数和偶数，过程中共用一个偏移量，代码逻辑如下所示。写完后打印出数组相邻两个数的最大绝对差值以及数组各元素的总和（注意：使用uint64）。

int MAX=10000000;

index = 0

//thread1

for(i=0;i<MAX;i+=2) {

data[index] = i; //even ( i+1 for thread 2)

index++;

}

//thread2

for(i=0;i<MAX;i+=2) {

data[index] = i+1; //odd

index++;

}

请分别按下列方法完成一个不会丢失数据的程序：

1) 请用Peterson算法实现上述功能。

2) 学习了解 pthread\_mutex\_lock/unlock()函数的功能，并实现上述程序功能。

3) 学习了解 atomic\_add()( \_sync\_fetch\_and\_add（）for gcc 4.1+) 函数，实现上述程序功能。

作业要求：

1. 作业需提供上述三种方法的源代码，运行命令及结果（即，数组相邻两个数的最大绝对差值，数组各元素的总和）
2. 作业中需说明程序中的临界区。同时，方法1和方法2 实现时，每次进入临界区之后，执行一百次操作后离开临界区。方法3不做要求。
3. 请找一个双核系统测试，作业中需提供三种方法中完成数组写入时各自所需的执行时间，不用提供计算绝对差值和数组总和的时间。
4. Peterson算法

代码：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<unistd.h>

#define true 1

#define false 0

typedef int bool;

bool flag[2];

int turn;

int index;

#define MAX 10000000

int data[MAX] = {0};

unsigned long sum()

{

int i;

unsigned long sum = 0;

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

sum = sum + data[i];

return sum;

}

int sub()

{

int i,j;

int sub, ex\_sub;

ex\_sub = data[1] - data[0];

if(ex\_sub < 0)

sub = -ex\_sub;

else sub = ex\_sub;

for(i = 1; i < MAX-1; i++)

{

ex\_sub = data[i+1] - data[i];

if(ex\_sub > sub || ex\_sub + sub < 0)

sub = (ex\_sub>0)?ex\_sub:(-ex\_sub);

}

return sub;

}

void procedure0()

{

int odd = 1;

int i;

while(true)

{

flag[0] = true;

turn = 1;

while(flag[1] && turn == 1)//退出while循环的条件就是，要么另一个线程

//不想要使用关键区，要么此线程拥有访问权限。

{

}

//critical section

if(index < MAX)

for(i = 0; i < 100; i++)

{

data[index] = odd;

odd = odd + 2;

index = index + 1;

//printf("%d\n",index);

}

else {

printf("The sum is %lu\n",sum());

printf("The largest sub is %d\n",sub());

flag[0] = false;

pthread\_exit(NULL);}

flag[0] = false;

}

}

void procedure1()

{

int even = 0;

int i;

while(true)

{

flag[1] = true;

turn = 0;

while(flag[0] && turn == 0)

{

}

//critical section

if(index < MAX)

for(i = 0; i < 100; i++)

{

data[index] = even;

even = even + 2;

index = index + 1;

//printf("%d\n",index);

}

else

{

printf("The sum is %lu\n",sum());

printf("The largest sub is %d\n",sub());

flag[1] = false;

pthread\_exit(NULL);}

flag[1] = false;

}

}

void main()

{

pthread\_t t1,t2;

flag[0] = flag[1] = false;

int err;

turn = 0;

err = pthread\_create(&t1,NULL,(void\*)procedure0,NULL);

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

err = pthread\_create(&t2,NULL,(void\*)procedure1,NULL);

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

pthread\_join(t1,NULL);

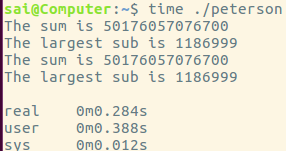
pthread\_join(t2,NULL);

return ;

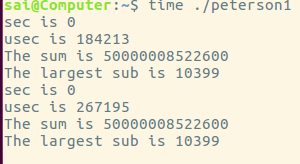
}

值得注意的时，在测试程序时发现，全局变量在多线程内共享，但在多进程内不共享。通过循环打印index的值可以发现这个规律。

程序中的临界区用灰色加深。

双核单独运行一次的程序打印和时间结果。由于最后需要将两个线程杀死，故先退出的线程需要在退出前将另一线程的flag设置为false，否则会陷入死循环。

运行结果表示，真实时间与用户时间相差并非两倍。这从代码结构中可以看出，由于临界区的存在，使线程在执行临界区的指令时，另一个线程需要空等，这样导致了时间的不可并行性。

对代码略作修改，在创建线程处新增了一个gettimeofday变量tv\_begin，在求和前新增gettimeofday变量tv\_end，计算两者之差，得到数组赋值阶段总用时0.18秒左右。同时也可以给出求和加比较相邻绝对值最大的算法耗时0.08秒左右。

1. 互斥锁算法

代码：

#include <pthread.h>

#include <stdio.h>

#define NLOOP 100

#define MAX 10000000

static pthread\_mutex\_t counter\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

static int counter;

void \*fodd(void \*);

void \*feven(void \*);

int odd = 1;

int even = 0;

int index = 0;

int data[MAX] = {0};

int main()

{

int i;

unsigned long sum;

pthread\_t tidA, tidB;

pthread\_create(&tidA, NULL, fodd, NULL);

pthread\_create(&tidB, NULL, feven, NULL);

sum = 0;

/\*wait for both threads to terminate\*/

pthread\_join(tidA, NULL);

pthread\_join(tidB, NULL);

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

sum = sum + data[i];

printf("The sum is %lu\n",sum);

return 0;

}

void \*fodd(void \*arg)

{

int i, val;

while(index < MAX)

{

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

for (i=0; i<NLOOP; i++) {

data[index] = odd;

odd = odd + 2;

index++;

}

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

}

return NULL;

}

void \*feven(void \*arg)

{

int i, val;

while(index < MAX)

{

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

for (i=0; i<NLOOP; i++) {

data[index] = even;

even = even + 2;

index++;

}

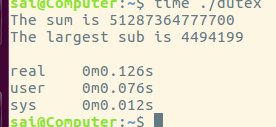
pthread\_mutex\_unlock(&counter\_mutex);

}

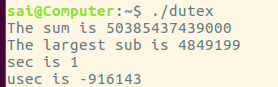
return NULL;

}

代码中灰色部分为临界区。

双核，单独运行程序一次的结果。互斥锁的实现大大节省了算法在调度上的耗费。

但是，我们可以看到，即使是在双核处理器中，系统所使用的真实时间仍比用户时间要长。这是因为由于互斥锁的阻塞功能，实际上在某一时刻只有一个进程在运行，另一个进程处于被阻塞的情况。这与Peterson算法不同的是，算法将其中一个进程陷入while的自旋，实际上该部分也占用了总时间。这里体现了互斥锁算法的优越性，在双核运行上总会有一个核其实属于未使用状态，可以用来处理其他的进程。

类似的，在线程创建和结束处加入gettimeofday的变量，测得总计运行时间为1-0.91 = 0.09秒。这一部分的处理，近似总时间为之前使用Peterson算法的一半，于是猜想在双核系统运行下，由于另一个线程始终处于被阻塞的状态，所以双核处理器都用于处理同一个线程，并且赋值逻辑比较简单，数据之间依赖性低，其速度得到将近两倍的加速也是可行的。

1. \_\_sync\_fetch\_and\_add()函数

该函数从属于一个函数族。假设一个全局变量count在多个线程内被引用，其作用是把count++这样的操作变成一个原子操作，这样可以保证在进程切换时对count的操作不会因为切换进程而错误。因为普通的++操作不是一步完成的，不具备原子性，现将其原子化后可保证其正确性。

type \_\_sync\_fetch\_and\_add (type \*ptr, type value);  
type \_\_sync\_fetch\_and\_sub (type \*ptr, type value);  
type \_\_sync\_fetch\_and\_or (type \*ptr, type value);  
type \_\_sync\_fetch\_and\_and (type \*ptr, type value);  
type \_\_sync\_fetch\_and\_xor (type \*ptr, type value);  
type \_\_sync\_fetch\_and\_nand (type \*ptr, type value);  
type \_\_sync\_add\_and\_fetch (type \*ptr, type value);  
type \_\_sync\_sub\_and\_fetch (type \*ptr, type value);  
type \_\_sync\_or\_and\_fetch (type \*ptr, type value);  
type \_\_sync\_and\_and\_fetch (type \*ptr, type value);  
type \_\_sync\_xor\_and\_fetch (type \*ptr, type value);  
type \_\_sync\_nand\_and\_fetch (type \*ptr, type value);

以上为原子操作族的12个函数。各自实现了操作后返回和返回后操作的原子操作。

之所以采用这些函数可以避免对全局变量修改的错误，是因为当对全局变量进行修改操作时，如果在修改到一半的情况下出现线程调度，很有可能导致操作重叠（比如说count当前等于10，在做完+1操作之后还没来得及写回就跳转走了，这样到下一个线程中从内存取出count的值还是10，随后+1操作，写回，跳转回原线程后，由于最后只剩一步写回，就出现了在两个+1操作后，count的值一直为11的错误），而将++操作原子化之后就不会出现这样的问题。

代码：

#define \_GNU\_SOURCE

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

#include <linux/unistd.h>

#include <sys/syscall.h>

#include <errno.h>

#include<linux/types.h>

#include<time.h>

#include<sys/time.h>

#define INC\_TO 100 // one million...

#define MAX 10000000

int data[MAX] = {0};

unsigned long sum()

{

int i;

unsigned long sum = 0;

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

sum = sum + data[i];

return sum;

}

int sub()

{

int i,j;

int sub, ex\_sub;

ex\_sub = data[1] - data[0];

if(ex\_sub < 0)

sub = -ex\_sub;

else sub = ex\_sub;

for(i = 1; i < MAX-1; i++)

{

ex\_sub = data[i+1] - data[i];

if(ex\_sub > sub || ex\_sub + sub < 0)

sub = (ex\_sub>0)?ex\_sub:(-ex\_sub);

}

return sub;

}

int global\_int = 0;

int odd = 1;

int even = 0;

pthread\_mutex\_t count\_lock = PTHREAD\_MUTEX\_INITIALIZER;

void \*thread\_routine( void \*arg )

{

int i;

int proc\_num = (int)(long)arg;

while(global\_int < MAX-1)

{

// global\_int++;

data[\_\_sync\_fetch\_and\_add( &global\_int, 1 )] = \_\_sync\_fetch\_and\_add(&odd, 2 );

}

return NULL;

}

void \*thread\_routine2( void \*arg )

{

int i;

int proc\_num = (int)(long)arg;

while(global\_int < MAX-1)

{

// global\_int++;

data[\_\_sync\_fetch\_and\_add( &global\_int, 1 )] = \_\_sync\_fetch\_and\_add(&even, 2 );

}

return NULL;

}

int main()

{

int procs = 0;

int i;

pthread\_t \*thrs;

// Getting number of CPUs

procs = (int)sysconf( \_SC\_NPROCESSORS\_ONLN );

if (procs < 0)

{

perror( "sysconf" );

return -1;

}

thrs = malloc( sizeof( pthread\_t ) \* procs );

if (thrs == NULL)

{

perror( "malloc" );

return -1;

}

printf( "Starting %d threads...\n", procs );

i = 0;

if (pthread\_create( &thrs[0], NULL, thread\_routine,

(void \*)(long)i ))

{

perror( "pthread\_create" );

procs = i;

}

i = 1;

if (pthread\_create( &thrs[1], NULL, thread\_routine2,

(void \*)(long)i ))

{

perror( "pthread\_create" );

procs = i;

}

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

pthread\_join( thrs[i], NULL );

free( thrs );

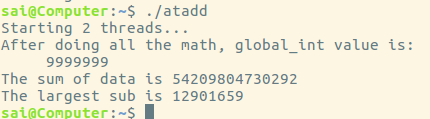
printf( "After doing all the math, global\_int value is: %d\n", global\_int );

printf("The sum of data is %lu\n",sum());

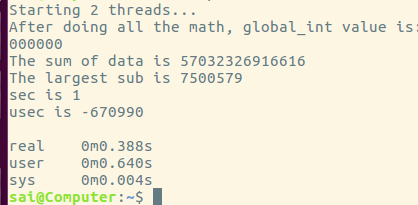
printf("The largest sub is %d\n",sub());

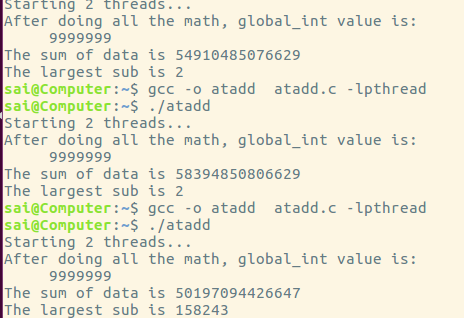
return 0;

}

双核，进行单步执行。

从打印的global\_int（对应index）的值为9999999来看，原子操作保证了index的正确自增操作。

添加gettimeofday函数后，测量得到写入数组的耗时约为0.33秒。这个从数值上来看比使用锁的算法耗时要长，可能与\_\_sync函数本身调用过程之间存在阻塞有关。在文档末尾我会单独提供针对互斥锁锁定的count++与调用\_\_sync\_fetch\_and\_add函数实现的自增操作的时间对比。

发现了一个有趣的现象，当我设置检查两个相邻数组元素的最大差值时，如果检查范围设置为前1000个和前10000个时，sub最大只为2。但是如果调整到100000时，差距就会突然增长。怀疑这与时间片的长度有关。

附：对比使用互斥锁的单步自增操作与使用\_\_sync\_fetch\_and\_add()的时间差异。

代码：

#define \_GNU\_SOURCE

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <sched.h>

#include <linux/unistd.h>

#include <sys/syscall.h>

#include <errno.h>

#include<linux/types.h>

#include<time.h>

#include<sys/time.h>

#define INC\_TO 1000000 // one million...

\_\_u64 rdtsc()

{

\_\_u32 lo,hi;

\_\_asm\_\_ \_\_volatile\_\_

(

"rdtsc":"=a"(lo),"=d"(hi)

);

return (\_\_u64)hi<<32|lo;

}

int global\_int = 0;

pthread\_mutex\_t count\_lock = PTHREAD\_MUTEX\_INITIALIZER;

pid\_t gettid( void )

{

return syscall( \_\_NR\_gettid );

}

void \*thread\_routine( void \*arg )

{

int i;

int proc\_num = (int)(long)arg;

\_\_u64 begin, end;

struct timeval tv\_begin,tv\_end;

\_\_u64 timeinterval;

cpu\_set\_t set;

CPU\_ZERO( &set );

CPU\_SET( proc\_num, &set );

if (sched\_setaffinity( gettid(), sizeof( cpu\_set\_t ), &set ))

{

perror( "sched\_setaffinity" );

return NULL;

}

begin = rdtsc();

gettimeofday(&tv\_begin,NULL);

for (i = 0; i < INC\_TO; i++)

{

// global\_int++;

\_\_sync\_fetch\_and\_add( &global\_int, 1 );

}

gettimeofday(&tv\_end,NULL);

end = rdtsc();

timeinterval =(tv\_end.tv\_sec - tv\_begin.tv\_sec)\*1000000 +(tv\_end.tv\_usec - tv\_begin.tv\_usec);

fprintf(stderr,"proc\_num :%d,\_\_sync\_fetch\_and\_add cost %llu CPU cycle,cost %llu us\n", proc\_num,end-begin,timeinterval);

return NULL;

}

void \*thread\_routine2( void \*arg )

{

int i;

int proc\_num = (int)(long)arg;

\_\_u64 begin, end;

struct timeval tv\_begin,tv\_end;

\_\_u64 timeinterval;

cpu\_set\_t set;

CPU\_ZERO( &set );

CPU\_SET( proc\_num, &set );

if (sched\_setaffinity( gettid(), sizeof( cpu\_set\_t ), &set ))

{

perror( "sched\_setaffinity" );

return NULL;

}

begin = rdtsc();

gettimeofday(&tv\_begin,NULL);

for(i = 0;i<INC\_TO;i++)

{

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

global\_int++;

pthread\_mutex\_unlock(&count\_lock);

}

gettimeofday(&tv\_end,NULL);

end = rdtsc();

timeinterval =(tv\_end.tv\_sec - tv\_begin.tv\_sec)\*1000000 +(tv\_end.tv\_usec - tv\_begin.tv\_usec);

fprintf(stderr,"proc\_num :%d,pthread lock cost %llu CPU cycle,cost %llu us\n",proc\_num,end-begin ,timeinterval);

return NULL;

}

int main()

{

int procs = 0;

int i;

pthread\_t \*thrs;

// Getting number of CPUs

procs = (int)sysconf( \_SC\_NPROCESSORS\_ONLN );

if (procs < 0)

{

perror( "sysconf" );

return -1;

}

thrs = malloc( sizeof( pthread\_t ) \* procs );

if (thrs == NULL)

{

perror( "malloc" );

return -1;

}

printf( "Starting %d threads...\n", procs );

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

{

if (pthread\_create( &thrs[i], NULL, thread\_routine2,

(void \*)(long)i ))

{

perror( "pthread\_create" );

procs = i;

break;

}

}

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

pthread\_join( thrs[i], NULL );

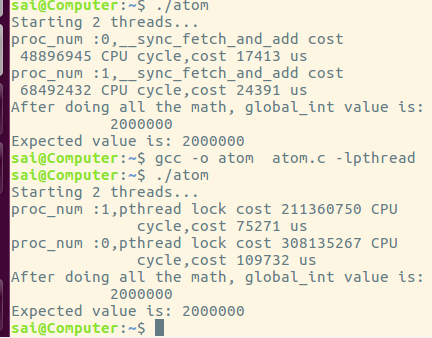
free( thrs );

printf( "After doing all the math, global\_int value is: %d\n", global\_int );

printf( "Expected value is: %d\n", INC\_TO \* procs );

return 0;

}

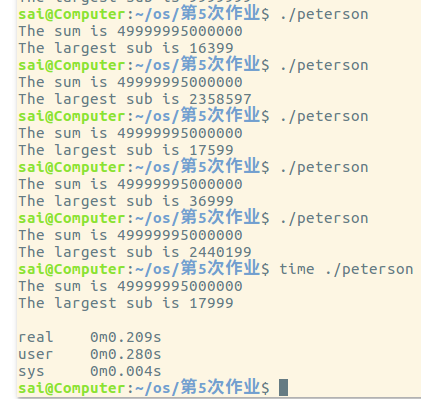
测试结果如图。

使用\_\_sync\_fetch\_and\_add一次的开销在0.02us左右，而使用互斥锁完成自增操作的开销在0.1us左右。二者之间的效率在5倍左右的量级上。

所以，使用单步操作全局变量时，无锁化的执行效率更高。但是，如果临界区的代码量较大，需要考虑\_\_\_sync函数族对程序的阻塞效果。

附：

在交作业之前发现所写程序和提供的示例程序实现的功能略有不同。示例程序中，i的值有上限10000000的约束，故当一个线程达到约束条件后将不会再执行，线程将被退出。而我之前写的版本是只有对index的约束，两个线程抢占是无约束的。这导致每一次赋给数组的值无论是顺序还是大小都是随机的，这样导致了总和随着程序每一次执行都在改变的情况。对临界区内运算的判断条件做了一点补充，得到了一下的运行结果：



可以看到，每次独立运行程序给出的总和是相同的，并且从相邻两个数的最大绝对值差不同可以看出，数组的赋值顺序并不同。该程序实现了目标。

事实上，修改后程序与原先程序的差别只差一个逻辑判断和计算总和打印判断，故在运行时长上应该只有一次计算加打印的时间差。从左图的real time中也可以反映出这一点。

发现问题的时候已经没有时间把之前的分析全部修改了，望助教大大网开一面QAQ