中山大学计算机学院2023 年春季超级计算原理与实践 MPI 编程作业本科生实验报告

课程名称：超级计算机原理与操作

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

一、实验题目

Pthread编程

二、实验内容

1.欧拉公式

并非所有和π有关的研究都旨在提高计算它的准确度。1735年，欧拉解决了巴塞尔问题，建立了所有平方数的倒数和π的关系:

$$ Π/6=\sum_{i=1}^{∞} 1/i^2 $$ 请使用pthread中的semaphore计算的值。

可以参考课件中的方法，在参考代码中提供了运行所需的主函数，也提供了串行代码供同学们参考；请同学们将并行的代码补充完整，需要补充的部分见注释PLEASE ADD THE RIGHTCODES部分。实验报告中请展示相应的运算结果，并分析加速比随n变化的关系。

2。生产者消费者问题

有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，所有生产者和消费者都是异步方式运行的，但它们必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经装满产品且尚未被取走的缓冲区中投放产品。在本题中，我们只考虑一个生产者一个消费者的情况，具体流程如图:

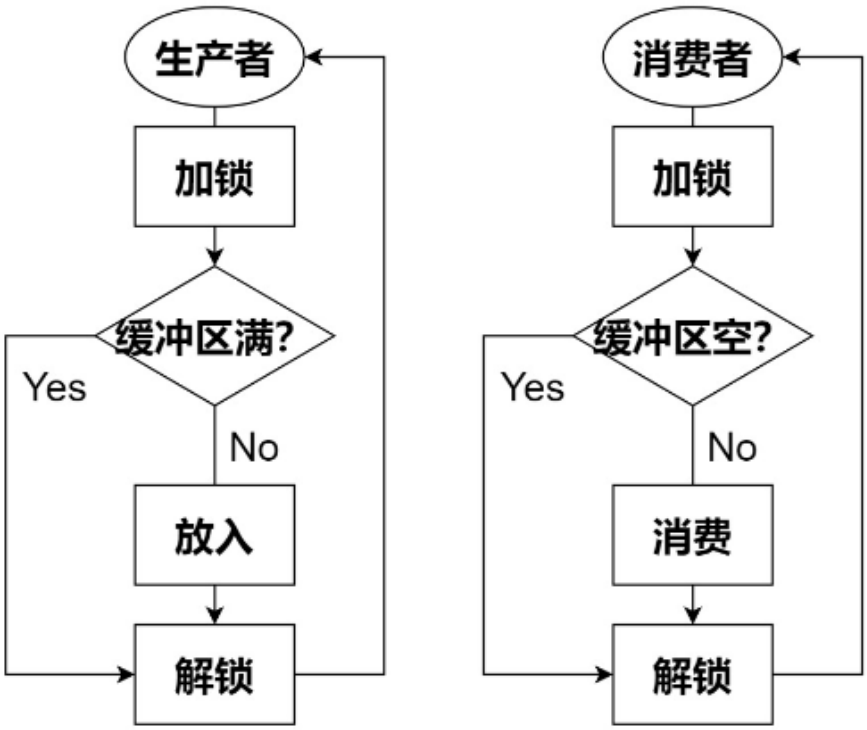


Figure 1:一把锁实现的生产者消费者队列

1.在参考代码中提供了运行所需要的主函数，请同学们将代码补充完整。

2.参考代码只使用了一把锁来实现生产者消费者队列，但是这存在一个问题，极端情况下，生产者每次都加锁成功，那缓冲区会满，产品无法放入缓冲区。消费者会饥饿，因为他一直无法获得锁，请考虑如何解决饥饿问题。实验报告中请展示相应的实验过程和运算结果。

3.线程池

线程池(thread pool)：是一种线程的使用模式。在实际系统中，频繁的创建销毁线程会带来过多的调度开销，从而影响整体性能。线程池维护着多个线程，等待着监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利用，还能防止过分调度。

在本习题中，我们采用任务队列的模式来实现一个线程池：主线程负责将相应的任务放入任务队列中，工作线程负责从任务队列中取出相应的任务进行处理，如果任务队列为空，则取不到任务的工作线程将进入挂起状态。具体代码见

github:https://github.com/Monaco12138/Threadpool-2023-DCS244-homework4

请根据要求填充完成空缺的部分，实现一个简单的线程池。

三、代码实现以及结果分析

1.实验一 ex1_pi_mutex

(1)代码实现

```c
/* File:     ex1_pi_mutex.c
 *
 * Purpose:  Estimate (pi^2)/6 using series
 *
 *              (pi^2)/6 = [1/(1^2) + 1/(2^2) + 1/(3^2) + 1/(4^2) . . . ]
 *
 *           It uses a semaphore to protect the critical section.
 *
 * Compile:  gcc -g -Wall -o ex1_pi_mutex ex1_pi_mutex.c -lpthread
 *           timer.h needs to be available
 * Run:      ./ex1_pi_mutex <number of threads> <n>
 *      or   ./ex1_pi_mutex  (using defalut)
 *           n is the number of terms of the Maclaurin series to use
 *           n should be evenly divisible by the number of threads
 *
 * Input:    none
 * Output:   The estimate of pi using multiple threads, one thread, and the
 *           value computed by the math library arctan function
 *           Also elapsed times for the multithreaded and singlethreaded
 *           computations.
 *
 * Notes:
 *    1.  The radius of convergence for the series is only 1.  So the
 *        series converges quite slowly.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/time.h>

#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}


const int MAX_THREADS = 1024;

long thread_count;
long  long n;
long double sum;

sem_t sem;

void* Thread_sum(void* rank);

/* Only executed by main thread */
void Get_args(int argc, char* argv[]);
void Usage(char* prog_name);
```

```
double Serial_pi(long long n);

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
   double start, finish, elapsed;

   /* please choose terms 'n', and the threads 'thread_count' here. */
   n = 100000000;
   thread_count = 4;

   /* You can also get number of threads from command line */
   //Get_args(argc, argv);

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   sem_init(&sem, 0, 1);
   sum = 0.0;
   GET_TIME(start);
    for (thread = 0; thread < thread_count; thread++)
       pthread_create(&thread_handles[thread], NULL,
           Thread_sum, (void*)thread);

    for (thread = 0; thread < thread_count; thread++)
       pthread_join(thread_handles[thread], NULL);
    GET_TIME(finish);
    elapsed = finish - start;
    // sum = 4.0*sum;
       printf("With n = %lld terms,\n", n);
       printf("   Our estimate of pi = %.15Lf\n", sum);
       printf("The elapsed time is %e seconds\n", elapsed);
       GET_TIME(start);
       sum = Serial_pi(n);
       GET_TIME(finish);
       elapsed = finish - start;
       printf("   Single thread est  = %.15Lf\n", sum);
       printf("The elapsed time is %e seconds\n", elapsed);
       printf("                     pi = %.15lf\n", (4.0*atan(1.0))*
(4.0*atan(1.0))/6 );
        sem_destroy(&sem);
          free(thread_handles);
          return 0;
}  /* main */

void* Thread_sum(void* rank) {
    long long  my_rank = (long long) rank;//获取自己的线程号
    long double my_sum = 0.0;//线程计算的和

    /*****************************************************************/



    long long local_n=n/thread_count;//此线程计算的个数
    long long start=1+(my_rank)*local_n;//此线程计算范围的开始
    long long end=(my_rank+1)*local_n;//此线程计算范围的结束
```

```
    //对本线程计算的范围求和
    for(long long i=start;i<=end;i++){

    my_sum += 1.0 / (i*i);

    }
   sem_wait(&sem);//sem_wait可以用来阻塞当前线程，直到信号量的值大于0，解除阻塞,解除阻塞后，sem的值-1，表示公共资源被执行减少了
   sum += my_sum;//全局变量sum加上本线程计算的和
   sem_post(&sem);//当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不再阻塞


   /*****************************************************************/

    return NULL;
}  /* Thread_sum */

double Serial_pi(long long n) {
                long double sum = 0.0;
                long long i;

                for ( i = 1; i <= n; i++ ) {
                   sum += 1.0 / (i*i);
                }
                return sum;

}  /* Serial */

void Get_args(int argc, char* argv[]) {
                  if (argc != 3) Usage(argv[0]);
                  thread_count = strtol(argv[1], NULL, 10);
                  if (thread_count <= 0 || thread_count > MAX_THREADS)
Usage(argv[0]);
                  n = strtoll(argv[2], NULL, 10);
                  if (n <= 0) Usage(argv[0]);
                }  /* Get_args */


void Usage(char* prog_name) {
                     fprintf(stderr, "usage: %s <number of threads> <n>\n",
prog_name);
                     fprintf(stderr, "   n is the number of terms and should be
>= 1\n");
                     fprintf(stderr, "   n should be evenly divisible by the
number of threads\n");
                     exit(0);
}  /* Usage */
```

实验原理

本实验利用$$ Π/6=\sum_{i=1}^{∞} 1/i^2 $$计算Π/6

(1)首先初始化信号量，同一时间只能由一个线程加入互斥区

```
sem_init(&sem, 0, 1);
```

(2)接着调用pthread_create创造thread_count多个线程，每个线程执行Thread_sum

```
for (thread = 0; thread < thread_count; thread++)
       pthread_create(&thread_handles[thread], NULL,Thread_sum, (void*)thread);
```

(3)在Thread_sum也就是我们编写的函数中，我们获取每个线程的线程号，根据线程号分好计算的范围，每个线程计算该线程负责范围的和，通过sem_wait(&sem)和sem_post(&sem)一个时刻只能一个线程加入互斥区域，并执行全局变量sum加上本线程计算的和。

```
void* Thread_sum(void* rank) {
    long long  my_rank = (long long) rank;//获取自己的线程号
    long double my_sum = 0.0;//线程计算的和

    /*****************************************************************/



    long long local_n=n/thread_count;//此线程计算的个数
    long long start=1+(my_rank)*local_n;//此线程计算范围的开始
    long long end=(my_rank+1)*local_n;//此线程计算范围的结束
    //对本线程计算的范围求和
    for(long long i=start;i<=end;i++){

    my_sum += 1.0 / (i*i);

    }
   sem_wait(&sem);//sem_wait可以用来阻塞当前线程，直到信号量的值大于0，解除阻塞,解除阻塞后，sem的值-1，表示公共资源被执行减少了
   sum += my_sum;//全局变量sum加上本线程计算的和
   sem_post(&sem);//当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不再阻塞


   /*****************************************************************/

    return NULL;
}  /* Thread_sum */
```

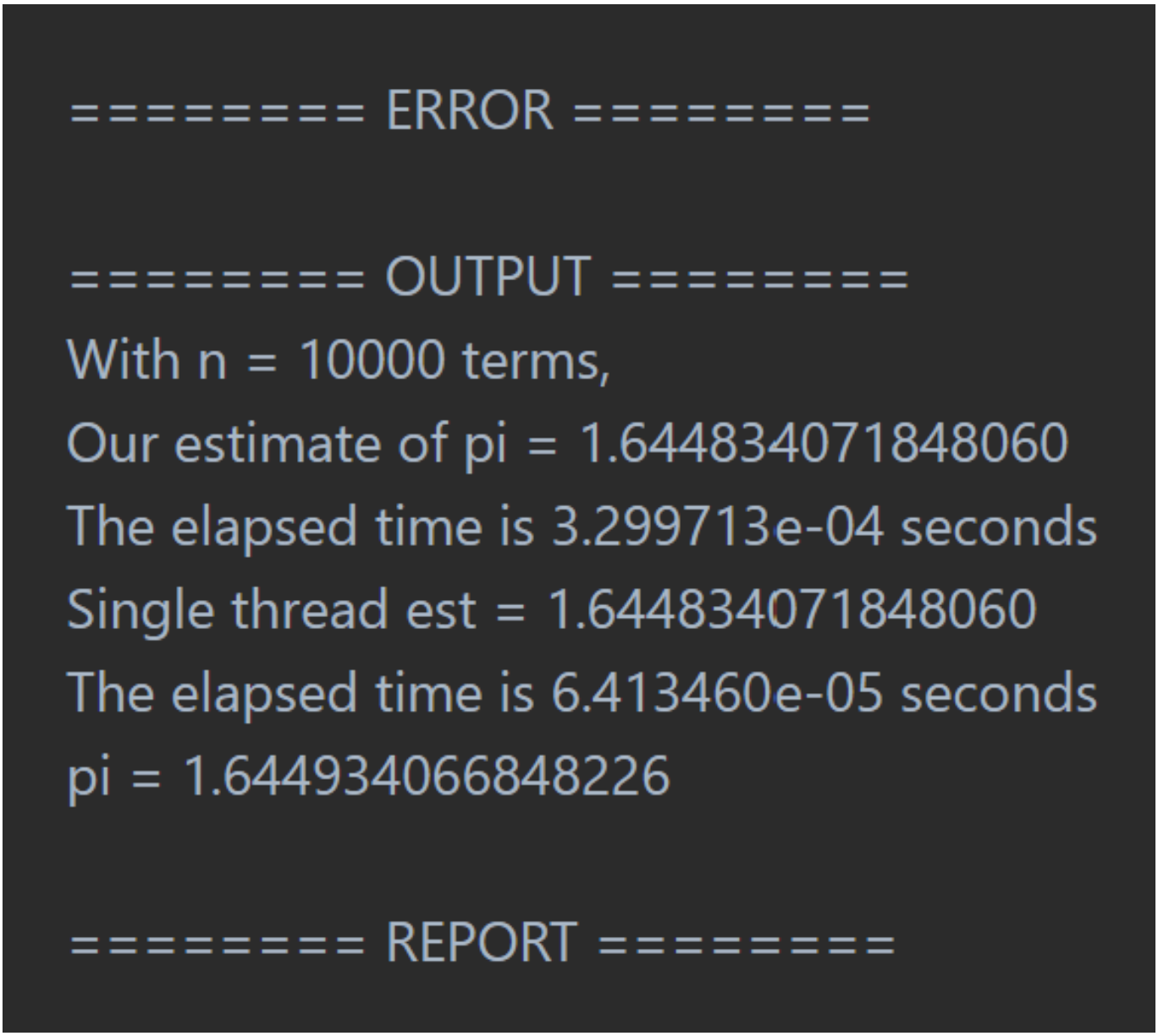
(4)输出多线程计算的时间和估计值

(5)再串行计算sum

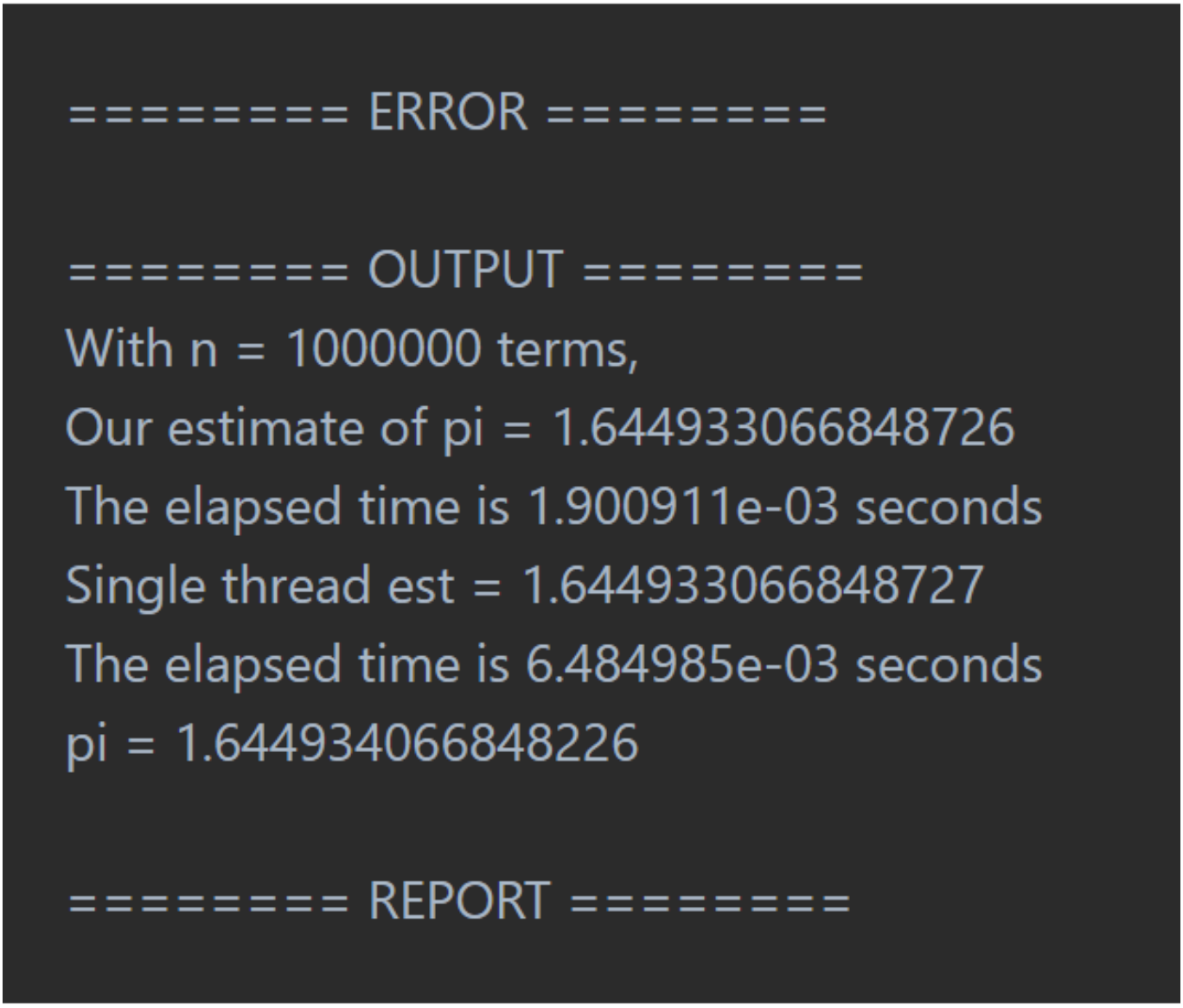
(6)输出串行计算的时间和估计值

运行结果

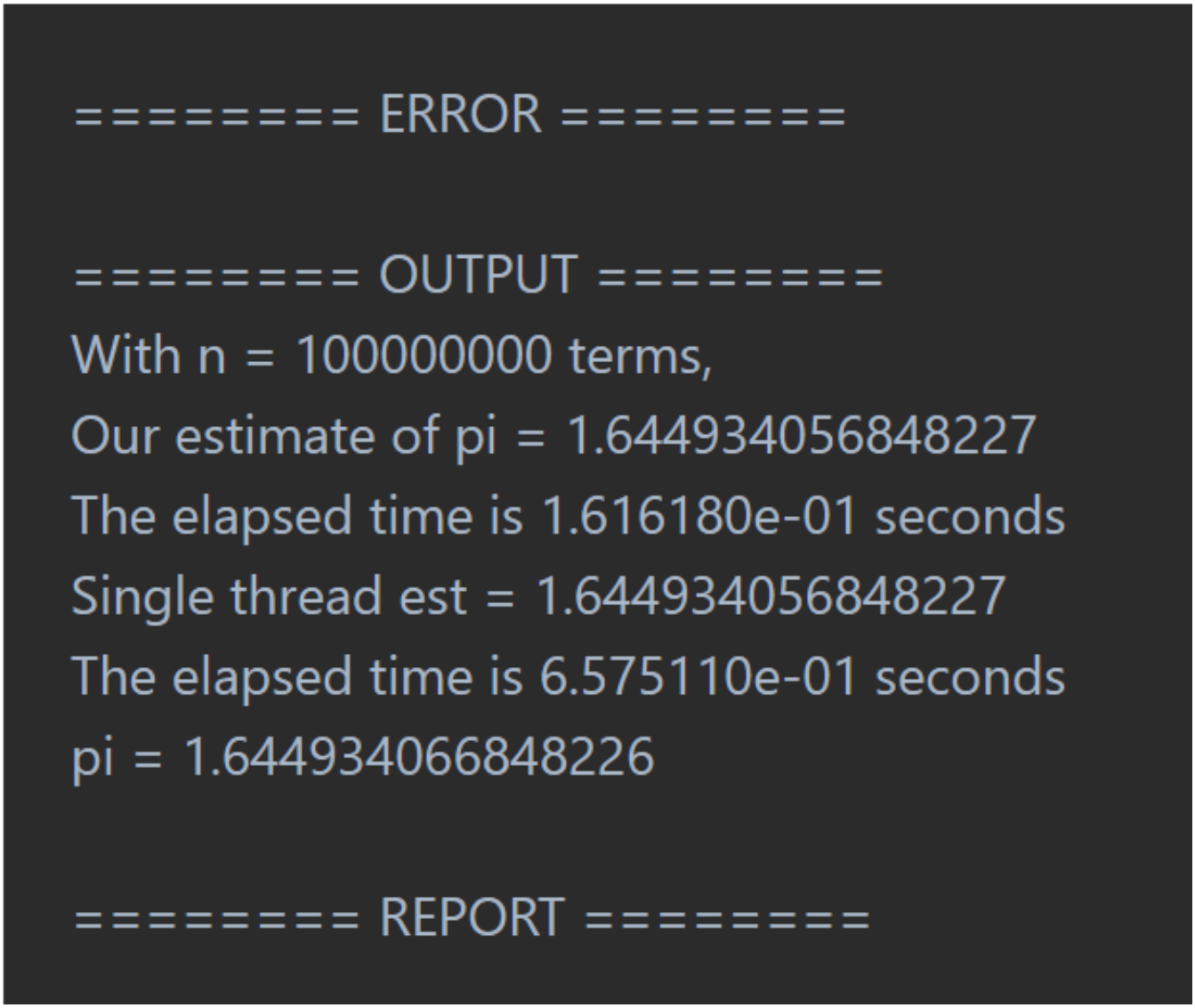
n = 10000,thread_count = 4



n = 1000000,thread_count = 4



n = 100000000,thread_count = 4



实验结果分析

加速比=Ts/Tp

thread_count	n	并行计算时间	串行计算时间	加速比
4	10000	3.299713e-04s	6.413460e-05s	19.44%
4	1000000	1.900911e-03s	6.484985e-03s	341.16%
4	100000000	1.616180e-01s	6.575110e-01s	406.83%

由表格数据可以看出，随着n的增大加速比也增大，在n=10000时并行计算时间慢于串行计算时间，n=100000000时并行计算时间是串行计算时间的1/4

2.实验二 ex2_producer

代码实现

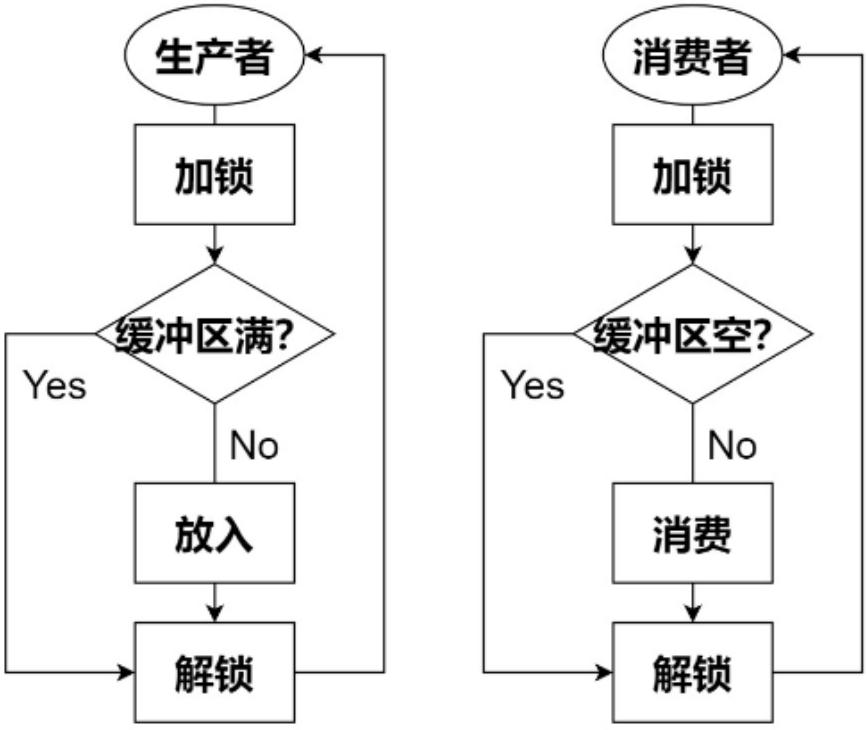
```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <pthread.h>
#define NUMS 100 //表示生产，消费的次数
#define CAPACITY 5 //定义缓冲区最大值
int capacity = 0; //当前缓冲区的产品个数
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;//互斥量

void *produce(void *args)//生产者函数
{
     /*****************************************************************/
    int produce_times=3;//固定生产的次数上限
    while(produce_times>0){//当未达到固定生产的次数上限，继续生产、
    if(capacity>=CAPACITY)continue;//防止对锁的竞争造成的饥饿问题
    pthread_mutex_lock(&mylock);//上锁
    if(capacity<CAPACITY)produce_times--;//若未达到缓冲区最大值，则可以生产
    while(capacity<CAPACITY){//生产直到不能再生产
        printf("Producer:capacity=%d\n",++capacity);//输出缓冲区产品个数
    }
    printf("Producer:The buffer is full!\n");
    pthread_mutex_unlock(&mylock);//解锁
    }
    /
    return NULL;


   /*****************************************************************/
}
void * consume(void *args)//消费者函数
{
     /*****************************************************************/
    int consume_times=3;//固定消费的次数上限
    while(consume_times>0){//当未达到固定消费的次数上限，继续消费
    if(capacity<=0)continue;//防止对锁的竞争造成的无法生产问题
    pthread_mutex_lock(&mylock);//上锁
    if(capacity>0)consume_times--;//若缓冲区非空，则可以消费
    while(capacity>0){//消费直到缓冲区空
        printf("Consumer:capacity=%d\n",--capacity);//输出缓冲区产品个数
    }
    printf("Consumer:The buffer is empty!\n");
    pthread_mutex_unlock(&mylock);//解锁
    }
    return NULL;



   /*****************************************************************/
}
int main(int argc, char** argv) {
    int err;
    pthread_t produce_tid, consume_tid;
    void *ret;
    err = pthread_create(&produce_tid, NULL, produce, NULL);//创建线程
    if (err != 0) {
```

```
            printf("线程创建失败:%s\n", strerror(err));
            exit(-1);
        }
        err = pthread_create(&consume_tid, NULL, consume, NULL);
        if (err != 0)  {
            printf("线程创建失败:%s\n", strerror(err));
            exit(-1);
        }
        err = pthread_join(produce_tid, &ret);//主线程等到子线程退出
        if (err != 0) {
            printf("生产着线程分解失败:%s\n", strerror(err));
            exit(-1);
        }
        err = pthread_join(consume_tid, &ret);
        if (err != 0) {
            printf("消费者线程分解失败:%s\n", strerror(err));
            exit(-1);
        }

        return (EXIT_SUCCESS);
    }
```

实验原理

本程序是一个消费者、生产者问题(生产者函数),具体流程如图:



实验重点:

(1)问题:所有生产者和消费者都是异步方式运行的，但它们必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经装满产品且尚未被取走的缓冲区中投放产品

解决方法：

利用锁，每次对缓冲区操作前上锁、结束操作解锁，能避免生产者和消费者同时对互斥区进行操作造成冲突问题

(2)问题：参考代码只使用了一把锁来实现生产者消费者队列，但是这存在一个问题，极端情况下，生产者每次都加锁成功，那缓冲区会满，产品无法放入缓冲区。消费者会饥饿，因为他一直无法获得锁,产生饥饿问题

解决方法：

产生饥饿问题的原因是对生产者和消费者对锁也有竞争问题，参考代码规定只使用了一把锁，我们可以通过对capacity即缓冲区产品的数目的条件判断规定生产者和消费者对锁的使用:

若capacity>=CAPACITY,则说明缓冲区已满，不需要生产者上锁，所以使生产者忙等，直到capacity < CAPACITY，满足生产者生产条件可以上锁

```
void *produce(void *args)//生产者函数
{
     /*********************************************************************/
    int produce_times=3;//固定生产的次数上限
    while(produce_times>0){//当未达到固定生产的次数上限，继续生产、
    if(capacity>=CAPACITY)continue;//防止对锁的竞争造成的饥饿问题
    pthread_mutex_lock(&mylock);//上锁
    if(capacity<CAPACITY)produce_times--;//若未达到缓冲区最大值，则可以生产
    while(capacity<CAPACITY){//生产直到不能再生产
        printf("Producer:capacity=%d\n",++capacity);//输出缓冲区产品个数
    }
    printf("Producer:The buffer is full!\n");
    pthread_mutex_unlock(&mylock);//解锁
    }

    return NULL;


   /*********************************************************************/
}
```

若capacity<=0,则说明缓冲区空，不需要消费者上锁，所以使消费者忙等，直到capacity > 0，满足消费者消费条件可以上锁

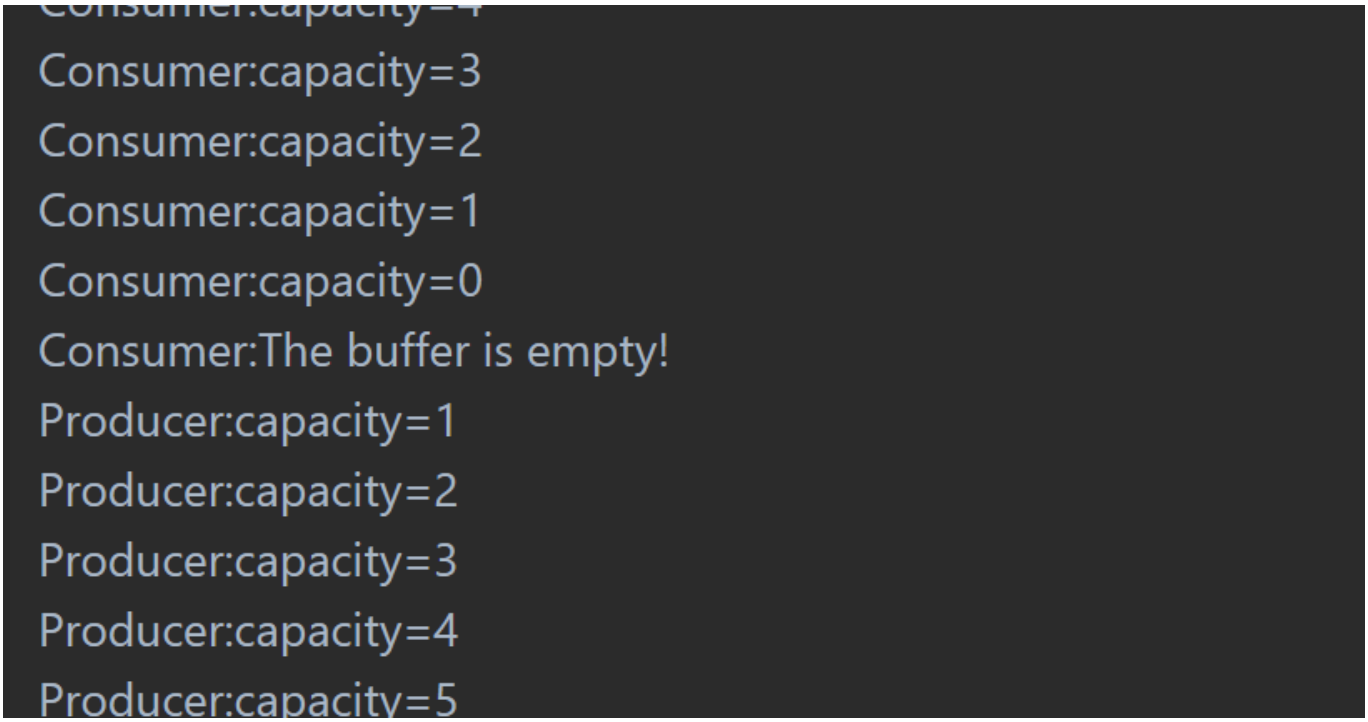
```
void * consume(void *args)//消费者函数
{
     /*********************************************************************/
    int consume_times=3;//固定消费的次数上限
    while(consume_times>0){//当未达到固定消费的次数上限，继续消费
    if(capacity<=0)continue;//防止对锁的竞争造成的无法生产问题
```

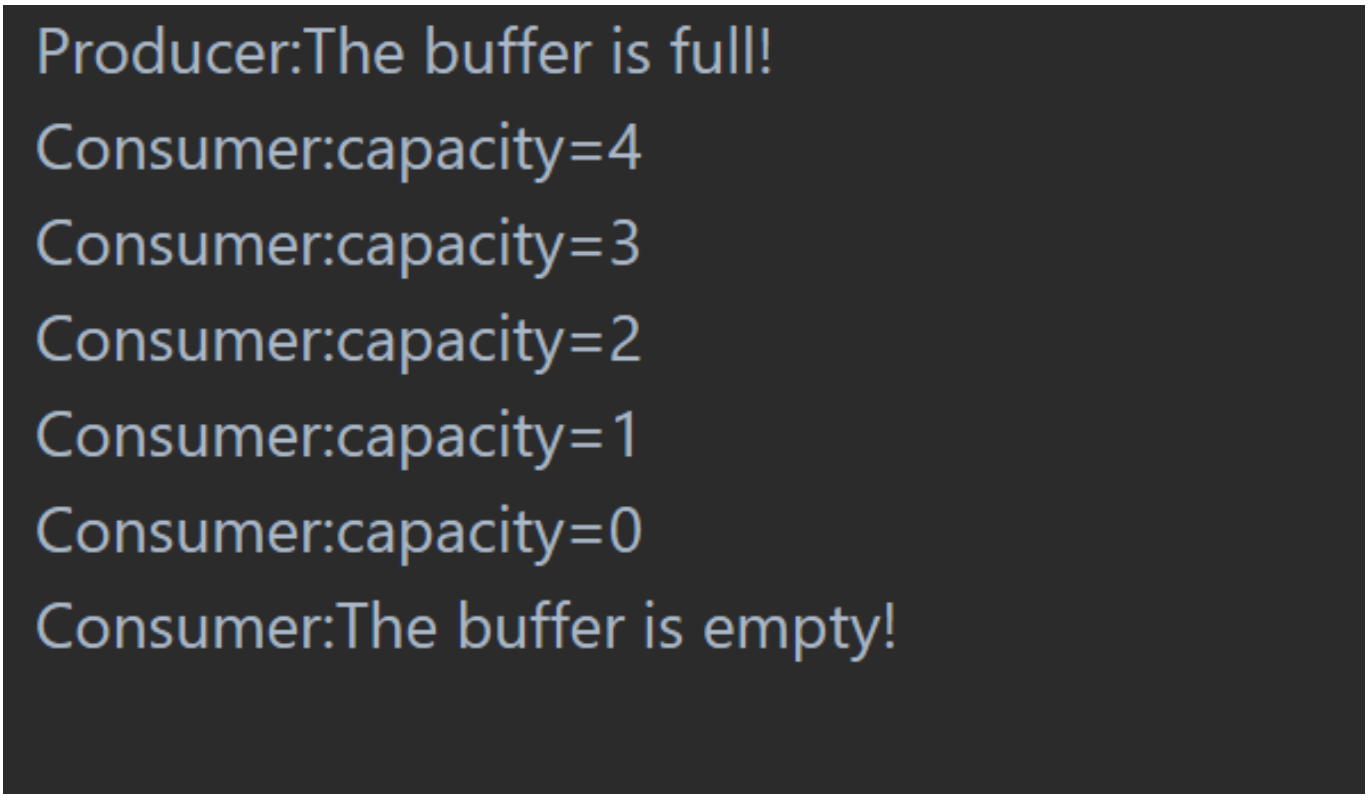
```
    pthread_mutex_lock(&mylock);//上锁
    if(capacity>0)consume_times--;//若缓冲区非空，则可以消费
    while(capacity>0){//消费直到缓冲区空
        printf("Consumer:capacity=%d\n",--capacity);//输出缓冲区产品个数
    }
    printf("Consumer:The buffer is empty!\n");
    pthread_mutex_unlock(&mylock);//解锁
    }
    return NULL;

   /*****************************************************************/
}
```

运行结果

为避免不断生产和消费造成死循环，我们设置生产和消费的上限次数=3

```
======== OUTPUT ========
Producer:capacity=1
Producer:capacity=2
Producer:capacity=3
Producer:capacity=4
Producer:capacity=5
Producer:The buffer is full!
Consumer:capacity=4
Consumer:capacity=3
Consumer:capacity=2
Consumer:capacity=1
Consumer:capacity=0
Consumer:The buffer is empty!
Producer:capacity=1
Producer:capacity=2
Producer:capacity=3
Producer:capacity=4
Producer:capacity=5
Producer:The buffer is full!
Consumer:capacity=4
```





3.实验三 thread_pool

代码实现(这里展示thread_pool.c中代码，其他代码未改动,每个函数细节在注释中)

(1)每个线程执行的函数，若线程池未回收则一直等待任务执行

```
void Job_running(threadpool* pool)// 线程池中的每个线程执行此函数
{
    while(pool->flag==1){//pool未回收则一直循环执行任务
    while(pool->jobnum==0);//无任务时忙等
    Jobnode job = Pop(pool);  // pop函数里面会进行互斥和等待非空
```

```
    if(job.pf!=NULL){
    job.pf(job.arg);  // 执行任务函数
    }
    }
    pthread_exit(0);
}
```

(2)线程池的初始化操作

```
threadpool* Pool_init(int Maxthread)//构造函数
{

    threadpool* pool;
    pool = (threadpool*)malloc(sizeof(threadpool));

    pool->flag = 1;//flag=0时表示被回收

    /*****************************************************************/
    pool->threads=(pthread_t*)malloc(sizeof(pthread_t)*Maxthread);
        //PLEASE ADD YOURs CODES
    if(pool->threads==NULL){//分配空间失败
        printf("malloc threads fail\n");
        return NULL;
    }
    //初始化
    pool->poolhead=(threadjob*)malloc(sizeof(threadjob));
    pool->Maxthread=Maxthread;
    for(int i=0;i<pool->Maxthread;i++)pthread_create(&pool->threads[i],NULL,
(void*)Job_running,pool);//从线程池取出线程执行任务函数
    pool->poolhead->next=NULL;
    pool->poolhead->data.pf=NULL;
    pool->jobnum=0;

    sem_init(&pool->sem, 0, 1);//每次只能一个线程加入线程池操作
    /*****************************************************************/

    return pool;
}
```

(3)将任务节点添加到线程池中

```
// 主线程添加任务到线程池中
int Add_job(threadpool* pool , function_t pf , void* arg)
{
    /*****************************************************************/
    sem_wait(&pool->sem);//上锁
    //添加任务
    threadjob *head =pool->poolhead;
    while(head->next!=NULL){
```

```
        head=head->next;
    }
    head->data.pf=pf;
    head->data.arg=arg;
    pool->jobnum++;
    threadjob*new_node=(threadjob*)malloc(sizeof(threadjob));
    head->next=new_node;
    new_node->next=NULL;
    sem_post(&pool->sem);//解锁

    //PLEASE ADD YOURs CODES
    return 1;
    /*****************************************************************/
}
```

(4)push相应的任务节点

```
int Push(threadpool* pool , Jobnode data )//push相应的任务节点
{
    /*****************************************************************/
    sem_wait(&pool->sem);
    //添加任务
    threadjob *head =pool->poolhead;
    while(head->next!=NULL){
        head=head->next;
    }
    head->data.pf=data.pf;
    head->data.arg=data.arg;
    pool->jobnum++;
    threadjob*new_node=(threadjob*)malloc(sizeof(threadjob));
    head->next=new_node;
    new_node->next=NULL;
    sem_post(&pool->sem);

        //PLEASE ADD YOURs CODES
    return 1;
    /*****************************************************************/
}
```

(5)删除线程池

```
int Delete_pool(threadpool* pool)//删除线程池
{

    pool->flag = 0;
    /*****************************************************************/

    free(pool->threads);
    free(pool);
```

```
    return 0;
        //PLEASE ADD YOURs CODES
    /*****************************************************************/

}
```

(6)pop任务节点

```
Jobnode Pop(threadpool* pool)//pop相应的任务节点
{
    /*****************************************************************/
    sem_wait(&pool->sem);//上锁
    if(pool->jobnum==0){//任务为0，做特殊处理，线程不需要执行任务
    sem_post(&pool->sem);
    return pool->poolhead->data;//data为NULL
    }
    pool->jobnum--;
    threadjob*head=pool->poolhead;
    pool->poolhead=head->next;
    head->next=NULL;
    sem_post(&pool->sem);//解锁
    return head->data;
        //PLEASE ADD YOURs CODES

    /*****************************************************************/
}
```

运行结果

```
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ gcc threadpool1.c
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ ./a.out
this is the task1 running in <101938>, the answer = 1037037832
this is the task2 running in <101939>, the answer = 1800079522
this is the task1 running in <101938>, the answer = 964052537
this is the task1 running in <101940>, the answer = 66661969
this is the task2 running in <101947>, the answer = 824997573
this is the task2 running in <101942>, the answer = 756257002
this is the task1 running in <101940>, the answer = 1996567275
this is the task2 running in <101938>, the answer = 1567893058
this is the task2 running in <101938>, the answer = 868043668
this is the task1 running in <101938>, the answer = 1081605874
this is the task2 running in <101938>, the answer = 873924261
this is the task1 running in <101938>, the answer = 2114379532
this is the task1 running in <101940>, the answer = 1288012791
this is the task1 running in <101942>, the answer = 1201757409
this is the task1 running in <101939>, the answer = 194722833
this is the task2 running in <101938>, the answer = 1856605521
this is the task2 running in <101938>, the answer = 1587120109
this is the task1 running in <101938>, the answer = 710177488
this is the task2 running in <101945>, the answer = 102977869
this is the task2 running in <101938>, the answer = 1701126374
this is the task2 running in <101938>, the answer = 911251867
this is the task2 running in <101942>, the answer = 1178593342
this is the task2 running in <101942>, the answer = 852621555
this is the task2 running in <101947>, the answer = 730789044
this is the task2 running in <101947>, the answer = 1662853175
this is the task1 running in <101940>, the answer = 551147307
this is the task1 running in <101945>, the answer = 285596674
this is the task1 running in <101942>, the answer = 442442027
this is the task1 running in <101938>, the answer = 1180915022
this is the task1 running in <101939>, the answer = 1101654828
this is the task1 running in <101941>, the answer = 249650222
this is the task1 running in <101945>, the answer = 2021576573
this is the task2 running in <101940>, the answer = 737797685
this is the task1 running in <101939>, the answer = 169748269
this is the task2 running in <101944>, the answer = 1839545043
this is the task2 running in <101945>, the answer = 366262189
this is the task1 running in <101944>, the answer = 1711774745
this is the task2 running in <101939>, the answer = 365825149
this is the task1 running in <101944>, the answer = 603241395
```

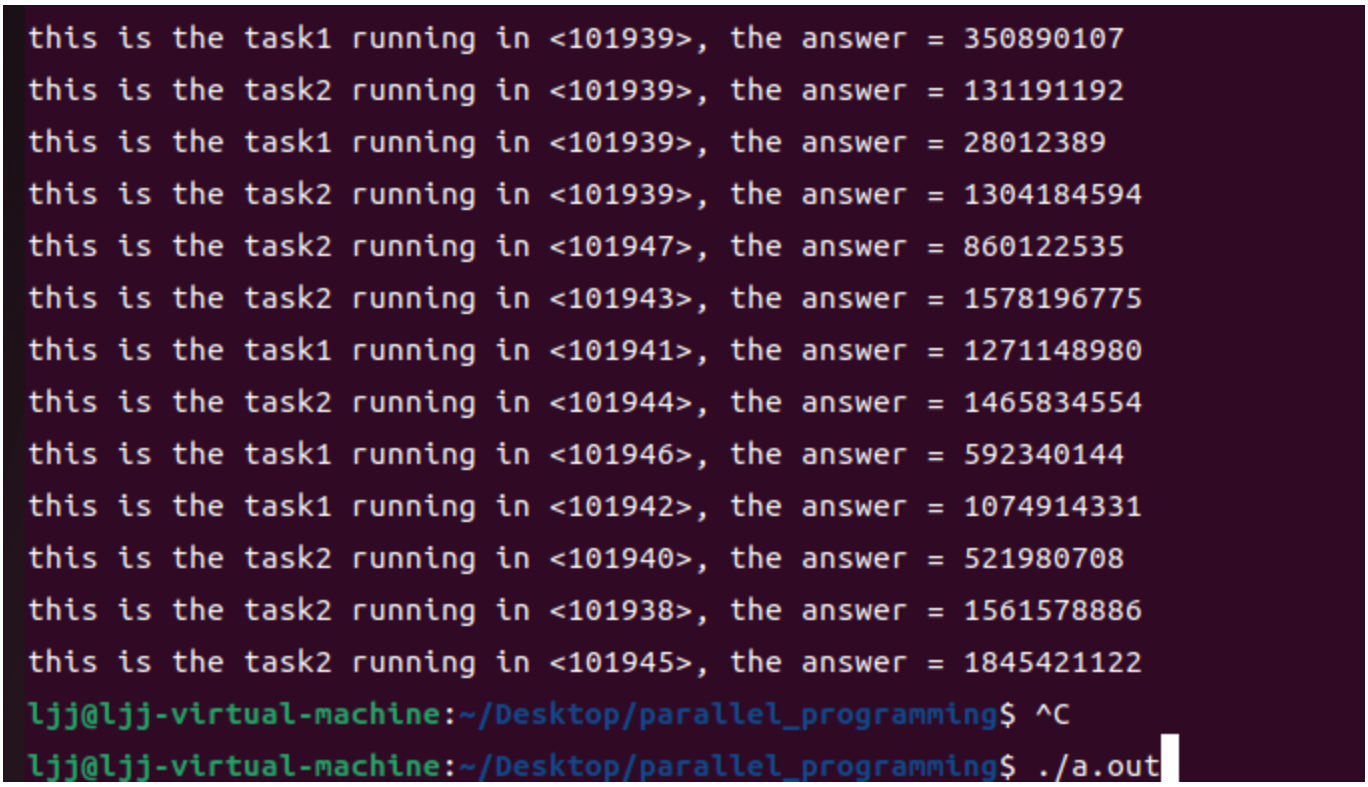
```
this is the task2 running in <101938>, the answer = 607842180
this is the task2 running in <101938>, the answer = 597534006
this is the task1 running in <101947>, the answer = 815252092
this is the task2 running in <101944>, the answer = 678774582
this is the task2 running in <101947>, the answer = 1914310254
this is the task2 running in <101942>, the answer = 495409754
this is the task1 running in <101939>, the answer = 1889527641
this is the task2 running in <101945>, the answer = 1825029515
this is the task2 running in <101939>, the answer = 708556020
```

```
this is the task1 running in <101943>, the answer = 1455331863
this is the task2 running in <101939>, the answer = 931218219
this is the task1 running in <101943>, the answer = 484274092
this is the task1 running in <101946>, the answer = 1775332371
this is the task1 running in <101940>, the answer = 2107371918
this is the task1 running in <101940>, the answer = 2100446749
this is the task2 running in <101940>, the answer = 285075094
this is the task1 running in <101940>, the answer = 1862390125
this is the task2 running in <101940>, the answer = 553242546
this is the task1 running in <101941>, the answer = 1766334495
this is the task1 running in <101943>, the answer = 934776364
this is the task2 running in <101941>, the answer = 653648863
this is the task2 running in <101946>, the answer = 654565815
this is the task2 running in <101939>, the answer = 1233265352
this is the task1 running in <101938>, the answer = 628312578
this is the task1 running in <101944>, the answer = 433174220
this is the task1 running in <101939>, the answer = 1136027938
this is the task1 running in <101942>, the answer = 546806024
this is the task1 running in <101944>, the answer = 971146780
this is the task1 running in <101940>, the answer = 225814792
this is the task1 running in <101946>, the answer = 198855265
this is the task2 running in <101938>, the answer = 797533636
this is the task2 running in <101938>, the answer = 800502739
this is the task1 running in <101938>, the answer = 2007546367
this is the task1 running in <101945>, the answer = 1027622393
this is the task1 running in <101945>, the answer = 774222568
this is the task2 running in <101939>, the answer = 1318636202
this is the task1 running in <101939>, the answer = 807885850
this is the task2 running in <101939>, the answer = 868330245
this is the task1 running in <101939>, the answer = 549314334
this is the task2 running in <101939>, the answer = 365112749
this is the task1 running in <101939>, the answer = 117531716
this is the task2 running in <101939>, the answer = 1764716560
this is the task1 running in <101939>, the answer = 1263552449
this is the task2 running in <101939>, the answer = 871220640
this is the task1 running in <101939>, the answer = 2052055620
this is the task2 running in <101939>, the answer = 699820164
```

```
this is the task1 running in <101939>, the answer = 1963244845
this is the task2 running in <101939>, the answer = 314025064
this is the task1 running in <101939>, the answer = 1322521682
this is the task2 running in <101939>, the answer = 20054941
this is the task1 running in <101939>, the answer = 1431698855
this is the task2 running in <101939>, the answer = 1782396017
this is the task1 running in <101939>, the answer = 1510771438
this is the task2 running in <101939>, the answer = 185024334
this is the task1 running in <101939>, the answer = 1407060632
this is the task2 running in <101939>, the answer = 1702979719
this is the task1 running in <101939>, the answer = 1338384499
this is the task2 running in <101939>, the answer = 32726094
this is the task1 running in <101939>, the answer = 1230586590
this is the task2 running in <101939>, the answer = 1846227242
this is the task1 running in <101939>, the answer = 1439044977
this is the task2 running in <101939>, the answer = 1008087618
this is the task1 running in <101939>, the answer = 796619639
this is the task2 running in <101939>, the answer = 927321209
this is the task1 running in <101939>, the answer = 780036460
this is the task2 running in <101939>, the answer = 2064807124
this is the task1 running in <101939>, the answer = 1992142358
this is the task2 running in <101939>, the answer = 939924828
this is the task1 running in <101939>, the answer = 1505128415
this is the task2 running in <101939>, the answer = 667845424
this is the task1 running in <101939>, the answer = 1486048806
this is the task2 running in <101939>, the answer = 1584061638
this is the task1 running in <101939>, the answer = 1834827101
this is the task2 running in <101939>, the answer = 1832218829
this is the task1 running in <101939>, the answer = 1057483884
this is the task2 running in <101939>, the answer = 348613726
this is the task1 running in <101939>, the answer = 19453680
this is the task2 running in <101939>, the answer = 2128680977
this is the task1 running in <101939>, the answer = 747926731
this is the task2 running in <101939>, the answer = 146143973
this is the task1 running in <101939>, the answer = 1797468591
this is the task2 running in <101939>, the answer = 562901676
this is the task1 running in <101939>, the answer = 551623260
this is the task2 running in <101939>, the answer = 552517043
```

```
this is the task1 running in <101939>, the answer = 892131908
this is the task2 running in <101939>, the answer = 1745093
this is the task1 running in <101939>, the answer = 608103342
this is the task2 running in <101939>, the answer = 1289627305
this is the task1 running in <101939>, the answer = 97403697
this is the task2 running in <101939>, the answer = 463417241
this is the task1 running in <101939>, the answer = 360106806
```

```
this is the task1 running in <101939>, the answer = 647394064
this is the task2 running in <101939>, the answer = 689996685
this is the task1 running in <101939>, the answer = 1855015278
this is the task2 running in <101939>, the answer = 659443394
this is the task1 running in <101939>, the answer = 259614276
this is the task2 running in <101939>, the answer = 896822261
this is the task1 running in <101939>, the answer = 1279978731
this is the task2 running in <101939>, the answer = 1888805487
this is the task1 running in <101939>, the answer = 1979281291
this is the task2 running in <101939>, the answer = 1664068152
this is the task1 running in <101939>, the answer = 513436249
this is the task2 running in <101939>, the answer = 1748083450
this is the task1 running in <101939>, the answer = 1021044621
this is the task2 running in <101939>, the answer = 972113674
this is the task1 running in <101939>, the answer = 1826557404
this is the task2 running in <101939>, the answer = 410892128
this is the task1 running in <101939>, the answer = 381818969
this is the task2 running in <101939>, the answer = 1415370784
this is the task1 running in <101939>, the answer = 906521447
this is the task2 running in <101939>, the answer = 1428098461
this is the task1 running in <101939>, the answer = 1244915966
this is the task2 running in <101939>, the answer = 1998076557
this is the task1 running in <101939>, the answer = 1369628513
this is the task2 running in <101939>, the answer = 1429147560
this is the task1 running in <101939>, the answer = 1002721347
this is the task2 running in <101939>, the answer = 587579464
this is the task1 running in <101939>, the answer = 821566133
this is the task2 running in <101939>, the answer = 1682562778
this is the task1 running in <101939>, the answer = 586089581
this is the task2 running in <101939>, the answer = 406052504
this is the task1 running in <101939>, the answer = 259890187
this is the task2 running in <101939>, the answer = 794197811
```



可以看到执行任务的线程的id号为101938到101948十个数字，说明执行任务的线程是不断地从线程池中的十个线程中取出来的，200个任务都是这十个不断执行操作的结果。

这样就能避免你创造太多个线程执行，每个线程都是执行一个时间很短的任务就结束了，频繁创建线程就会大大降低系统的效率

四、心得体会

(1) 通过编写实验一、二，我对避免互斥区的冲突问题有了更深的了解，同时学会在编程中使用锁和信号量对互斥区进行保护。

(2) 通过实验一，学会了使用多线程并行执行，用并行程序进行加速。

(3) 通过实验三，对线程池有了更深的了解，同时学会了编写简单的线程池重复使用线程。