2023 年春季超级计算原理与实践 Pthread 编程作业

曾慧蕾 21307358

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：



请使用pthread中的semaphore计算π²/6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。

C源码：

#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 = 10000;

   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;

    long double my\_sum = 0.0;

    long long i;

    long long group\_num=n/thread\_count;

    long long group\_first\_i=group\_num\*my\_rank;

    long long group\_last\_i=group\_first\_i+group\_num;

    for(i=group\_first\_i;i<group\_last\_i;i++){

      if(i==0) continue;

      my\_sum+=1.0/(i\*i);

    }

    sem\_wait(&sem);

    sum+=my\_sum;

    sem\_post(&sem);

    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");

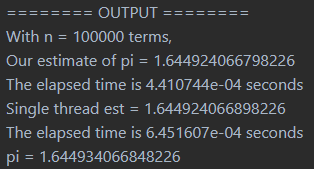
                      fprintf(stderr, "   n should be evenly divisible by the number of threads\n");

                      exit(0);

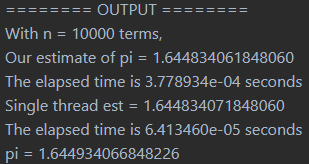
}  /\* Usage \*/

运行结果：

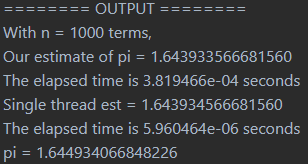
N=100000时：

加速比6.45160/4.4107≈1.462

N=10000时：

加速比64.134/3.7789≈0.1697

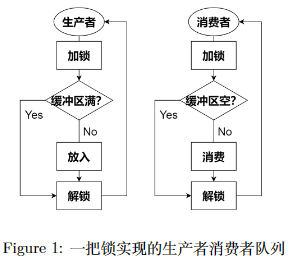
N=1000时：

加速比596.04/3.8195≈0.01561

可见两者的加速比会随着n的增大而增加，即，在运行程序相同的情况下，n越大，并行提高的效率越明显，效果越好。

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:



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。

C源码

#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 my\_lock1 = PTHREAD\_MUTEX\_INITIALIZER;//互斥量

//pthread\_mutex\_t my\_lock2 = PTHREAD\_MUTEX\_INITIALIZER;//互斥量

void \*produce(void \*args)

{

  // int err;

   while(times--){

     pthread\_mutex\_lock(&my\_lock1);

     if(capacity<CAPACITY){ //缓冲区未满 可以继续装

       capacity+=1;

       printf("加入一个产品，当前产品数为：%d\n",capacity);

     }

     else{

       printf("操作失败，缓冲区已满。当前产品数为：%d\n",capacity);

     }

     //err=pthread\_mutex\_unlock(&my\_lock2);

     pthread\_mutex\_unlock(&my\_lock1);

   }

  return NULL;

}

void \* consume(void \*args)

{

  int times=NUMS;

  while(times--){

    //pthread\_mutex\_lock(&my\_lock2);

    pthread\_mutex\_lock(&my\_lock1);

    if(capacity>0){ //缓冲区未满 可以继续装

      capacity-=1;

      printf("消费一个产品，当前产品数为：%d\n",capacity);

    }

    else{

      printf("操作失败，缓冲区已空。当前产品数为：%d\n",capacity);

    }

    pthread\_mutex\_unlock(&my\_lock1);

  }

  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));

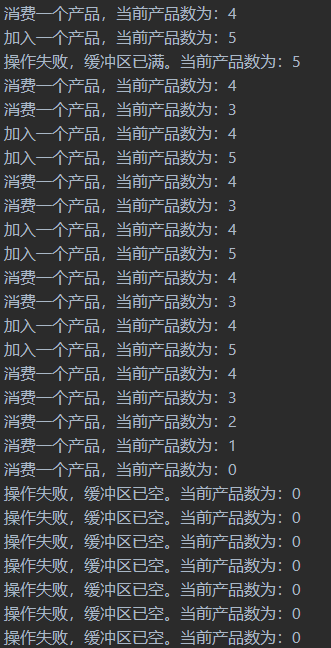
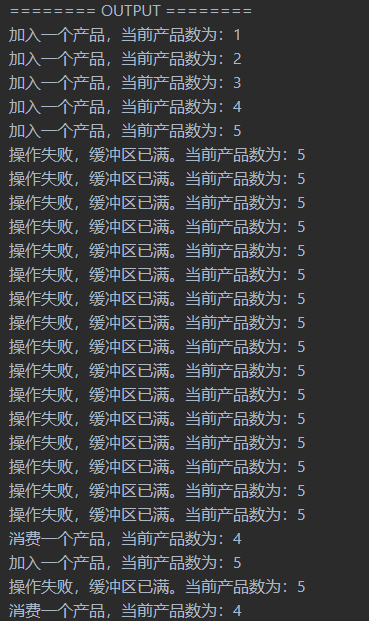
        exit(-1);

    }

    return (EXIT\_SUCCESS);

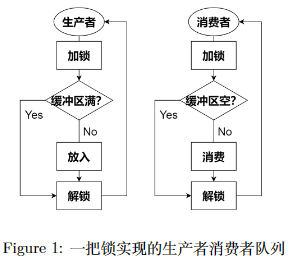
}

运行结果：



可见在一把锁的情况下，生产者与消费者会经常进行无效操作。下面来探讨如何解决饥饿问题。

1. 思路：采用互斥量来完成，设置两个锁，当其中一个执行其操作时，便给自己上锁，并给另外一个解锁





代码主要改的有：

全局变量：

pthread\_mutex\_t my\_lock1 = PTHREAD\_MUTEX\_INITIALIZER;//互斥量

pthread\_mutex\_t my\_lock2 = PTHREAD\_MUTEX\_INITIALIZER;//互斥量

生产者与消费者函数：

void \*produce(void \*args){

   int times=NUMS;

  int err;

   while(times--){

     pthread\_mutex\_lock(&my\_lock1);

     if(capacity<CAPACITY){ //缓冲区未满 可以继续装

       ……

     err=pthread\_mutex\_unlock(&my\_lock2);

   }

  return NULL;

}

void \* consume(void \*args)

{

  int times=NUMS;

  while(times--){

    pthread\_mutex\_lock(&my\_lock2);

    ……

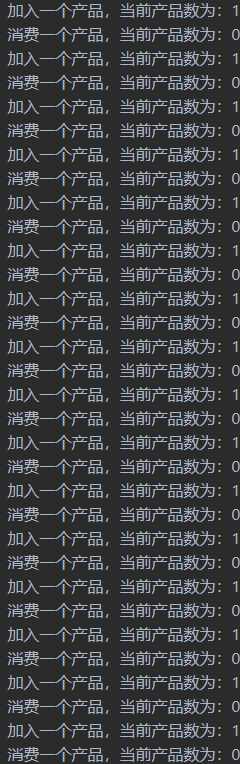
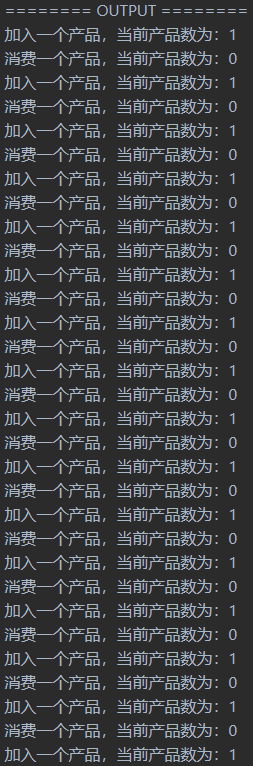
    pthread\_mutex\_unlock(&my\_lock2);

  }

  return NULL;

}

运行效果如下：



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

*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。

补充的代码详细可见文件内，这里仅展示threadpool.c中补充的代码。





运行结果如下所示：

