

# 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 $\pi$ 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  $\pi$  的关系：

$$\frac{\pi^2}{6} = \sum_{i=1}^{\infty} \frac{1}{i^2}$$

请使用 pthread 中的 semaphore 计算  $\pi^2/6$  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:

```

===== OUTPUT =====
With n = 100000 terms,
Our estimate of pi = 1.644924066798226
The elapsed time is 4.410744e-04 seconds
Single thread est = 1.644924066898226
The elapsed time is 6.451607e-04 seconds
pi = 1.644934066848226

```

加速比  $6.45160/4.4107 \approx 1.462$

N=10000 时:

```
===== OUTPUT =====  
With n = 10000 terms,  
Our estimate of pi = 1.644834061848060  
The elapsed time is 3.778934e-04 seconds  
Single thread est = 1.644834071848060  
The elapsed time is 6.413460e-05 seconds  
pi = 1.644934066848226
```

加速比  $64.134/3.7789 \approx 0.1697$

N=1000 时:

```
===== OUTPUT =====  
With n = 1000 terms,  
Our estimate of pi = 1.643933566681560  
The elapsed time is 3.819466e-04 seconds  
Single thread est = 1.643934566681560  
The elapsed time is 5.960464e-06 seconds  
pi = 1.644934066848226
```

加速比  $596.04/3.8195 \approx 0.01561$

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:

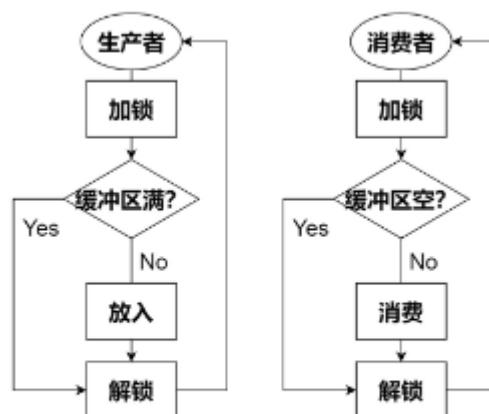


Figure 1: 一把锁实现的生产者消费者队列

- 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。

### 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{
```

```

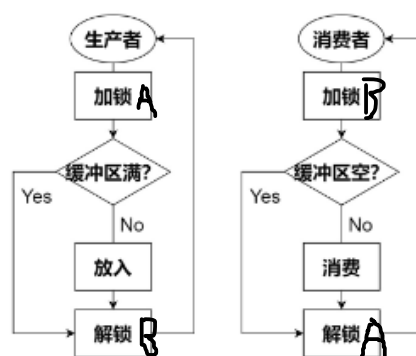
        printf("操作失败, 缓冲区已空。当前产品数为: %d\n", capacity);
    }
    pthread_mutex_unlock(&my_lock1);
}
return NULL;
}
int main(int argc, char** argv) {
    int err;
    pthread_t produce_tid, consume_tid;
    void *ret;
    err = pthread_create(&produce_tid, NULL, produce, NULL); //创建线程
    if (err != 0) {
        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("生产者线程分解失败:%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:

[illegible]

2、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



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:





```

#include "threadpool.h"

void Job_running(threadpool* pool){
    while(pool->flag){
        pthread_mutex_lock(&pool->mutexpool);
        while(pool->poolhead==NULL && pool->flag==1){
            pthread_cond_wait(&pool->notempty,&pool->mutexpool);
        }
        if(pool->flag==0){
            pthread_mutex_unlock(&pool->mutexpool);
            pthread_exit(0);
        }
        //threadjob *t = (threadjob*)malloc(sizeof(threadjob));
        threadjob *t=pool->poolhead;
        pool->poolhead=pool->poolhead->next;
        pool->jobnum--;
        pthread_mutex_unlock(&pool->mutexpool);
        t->data.pf(t->data.arg);
        free(t);
        t=NULL;
    }
    pthread_exit(0);
}

void *worker(void *arg){
    threadpool *pool = (threadpool *)arg;
    Job_running(pool);
    return NULL;
}

threadpool* Pool_init(int Maxthread){
    threadpool* pool;
    pool = (threadpool*)malloc(sizeof(threadpool));

    if(pool==NULL){
        printf("malloc error\n");
        return NULL;
    }

    pool->flag = 1;
    pool->Maxthread=Maxthread;
    pool->jobnum=0;
    pool->poolhead=NULL;
    pool->tail=NULL;
    pool->threads=NULL;
    pthread_mutex_init(&pool->mutexpool,NULL);
    pthread_cond_init(&pool->notempty,NULL);

    for(int i=0;i<Maxthread;i++){
        pthread_t tid;
        pthread_create(&tid,NULL,worker,pool);
    }
    return pool;
}

```

```

int Add_job(threadpool* pool , function_t pf , void* arg){
    if(pool->flag==0||pool->jobnum==pool->Maxthread){
        return -1;
    }
    threadjob* t=(threadjob*)malloc(sizeof(threadjob));
    t->data.arg=arg;
    t->data.pf=pf;
    t->next=NULL;
    pthread_mutex_lock(&pool->mutexpool);
    if(pool->poolhead==NULL){
        pool->poolhead=t;
        pool->tail=t;
    }
    else{
        pool->tail->next=t;
        pool->tail=t;
    }
    pool->jobnum++;
    pthread_cond_signal(&pool->notempty);
    pthread_mutex_unlock(&pool->mutexpool);
    return 1;
}

int Push(threadpool* pool , Jobnode data ){
    threadjob* t=(threadjob*)malloc(sizeof(threadjob));
    t->data=data;
    t->next=NULL;
    pthread_mutex_lock(&pool->mutexpool);
    if(pool->poolhead==NULL){
        pool->poolhead=t;
        pool->tail=t;
    }
    else{
        pool->tail->next=t;
        pool->tail=t;
    }
    pool->jobnum++;
    pthread_cond_signal(&pool->notempty);
    pthread_mutex_unlock(&pool->mutexpool);
    return 1;
}

Jobnode Pop(threadpool* pool){
    if(pool->flag==0||pool->jobnum==0){
        return pool->poolhead->data;
    }
    threadjob* t=pool->poolhead;
    pool->poolhead=pool->poolhead->next;
    free(t);
    t=NULL;
    return pool->poolhead->data;
}

```

```

int Delete_pool(threadpool* pool){
    pool->flag = 0;
    for(int i=0;i<pool->jobnum;i++){
        pthread_cond_signal(&pool->notempty);
    }
    if(pool->threads){
        free(pool->threads);
    }

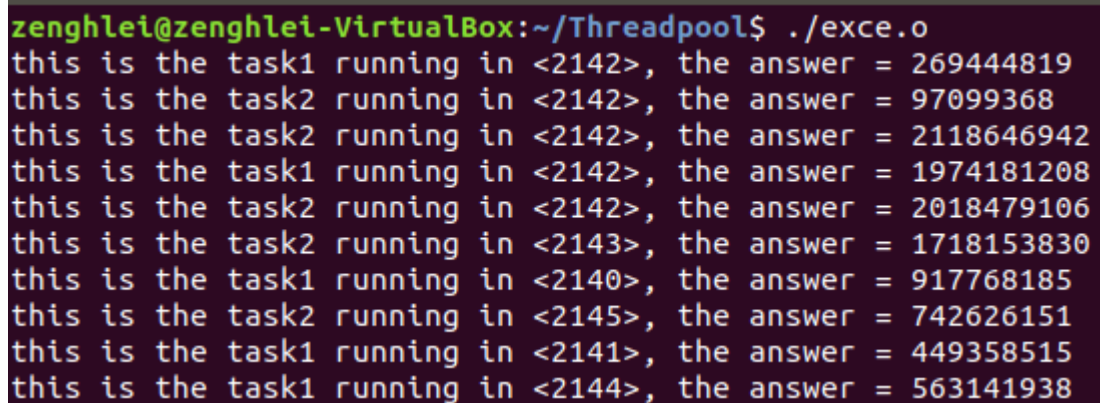
    pthread_mutex_destroy(&pool->mutexpool);
    pthread_cond_destroy(&pool->notempty);

    free(pool);
    pool=NULL;
    return 0;
}

```

---

运行结果如下所示：



```

zenghlei@zenghlei-VirtualBox:~/Threadpool$ ./exce.o
this is the task1 running in <2142>, the answer = 269444819
this is the task2 running in <2142>, the answer = 97099368
this is the task2 running in <2142>, the answer = 2118646942
this is the task1 running in <2142>, the answer = 1974181208
this is the task2 running in <2142>, the answer = 2018479106
this is the task2 running in <2143>, the answer = 1718153830
this is the task1 running in <2140>, the answer = 917768185
this is the task2 running in <2145>, the answer = 742626151
this is the task1 running in <2141>, the answer = 449358515
this is the task1 running in <2144>, the answer = 563141938

```