1 欧拉公式

```
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex
With n = 100000000 terms,
   Our estimate of pi = 1.644934056839816
The elapsed time is 1.546249e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 5.649080e-01 seconds
                  pi = 1.644934066848226
```

可以看到，求和的计算结果与直接计算得到的结果很相近，程序运行成功。

```
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 4 100000000
With n = 100000000 terms,
   Our estimate of pi = 1.644934056839816
The elapsed time is 1.542702e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 5.635731e-01 seconds
                  pi = 1.644934066848226
speedup = 3.653156759229495
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 4 500000000
With n = 500000000 terms,
   Our estimate of pi = 1.644934063834575
The elapsed time is 7.153399e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 2.815213e+00 seconds
                  pi = 1.644934066848226
speedup = 3.935489924018940
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 4 1000000000
With n = 1000000000 terms,
   Our estimate of pi = 1.644934060834575
The elapsed time is 1.418147e+00 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 5.630418e+00 seconds
                  pi = 1.644934066848226
speedup = 3.970264312922075
```

这里使用了 4 个线程，并增加了输出加速比的语句。可以看到随着 n 的增加，加速比越来越接近 4。

```
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 8 1000000000
With n = 1000000000 terms,
   Our estimate of pi = 1.644934064834575
The elapsed time is 7.059641e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 5.633517e+00 seconds
                  pi = 1.644934066848226
speedup = 7.979891780827470
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 8 500000000
With n = 500000000 terms,
   Our estimate of pi = 1.644934064638939
The elapsed time is 3.725979e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 2.817048e+00 seconds
                  pi = 1.644934066848226
speedup = 7.560557439296028
(base) lin@LindeMacBook-Pro 2 % ./ex1_pi_mutex 8 100000000
With n = 100000000 terms,
   Our estimate of pi = 1.644934056847037
The elapsed time is 1.027138e-01 seconds
   Single thread est  = 1.644934057834575
The elapsed time is 5.665350e-01 seconds
                  pi = 1.644934066848226
speedup = 5.515664569082177
```

这里使用了 8 个线程，也可以看到，n 越大，加速比越接近 8。

实验结果说明了，随着 n 值的增加，加数比趋于线性加速比。

2 生产者消费者问题

```
(base) lin@LindeMacBook-Pro 2 % ./ex2_producer
生产者生产第1号产品，当前缓存区产品个数为1
生产者生产第2号产品，当前缓存区产品个数为2
生产者生产第3号产品，当前缓存区产品个数为3
生产者生产第4号产品，当前缓存区产品个数为4
生产者生产第5号产品，当前缓存区产品个数为5
消费者消费第1号产品，当前缓存区产品个数为4
消费者消费第2号产品，当前缓存区产品个数为3
消费者消费第3号产品，当前缓存区产品个数为2
消费者消费第4号产品，当前缓存区产品个数为1
消费者消费第5号产品，当前缓存区产品个数为0
生产者生产第6号产品，当前缓存区产品个数为1
生产者生产第7号产品，当前缓存区产品个数为2
生产者生产第8号产品，当前缓存区产品个数为3
生产者生产第9号产品，当前缓存区产品个数为4
生产者生产第10号产品，当前缓存区产品个数为5
消费者消费第6号产品，当前缓存区产品个数为4
消费者消费第7号产品，当前缓存区产品个数为3
消费者消费第8号产品，当前缓存区产品个数为2
消费者消费第9号产品，当前缓存区产品个数为1
消费者消费第10号产品，当前缓存区产品个数为0
生产者生产第11号产品，当前缓存区产品个数为1
生产者生产第12号产品，当前缓存区产品个数为2
生产者生产第13号产品，当前缓存区产品个数为3
生产者生产第14号产品，当前缓存区产品个数为4
生产者生产第15号产品，当前缓存区产品个数为5
消费者消费第11号产品，当前缓存区产品个数为4
消费者消费第12号产品，当前缓存区产品个数为3
消费者消费第13号产品，当前缓存区产品个数为2
消费者消费第90号产品，当前缓存区产品个数为0
生产者生产第91号产品，当前缓存区产品个数为1
生产者生产第92号产品，当前缓存区产品个数为2
生产者生产第93号产品，当前缓存区产品个数为3
生产者生产第94号产品，当前缓存区产品个数为4
生产者生产第95号产品，当前缓存区产品个数为5
消费者消费第91号产品，当前缓存区产品个数为4
消费者消费第92号产品，当前缓存区产品个数为3
消费者消费第93号产品，当前缓存区产品个数为2
消费者消费第94号产品，当前缓存区产品个数为1
消费者消费第95号产品，当前缓存区产品个数为0
生产者生产第96号产品，当前缓存区产品个数为1
生产者生产第97号产品，当前缓存区产品个数为2
生产者生产第98号产品，当前缓存区产品个数为3
生产者生产第99号产品，当前缓存区产品个数为4
生产者生产第100号产品，当前缓存区产品个数为5
消费者消费第96号产品，当前缓存区产品个数为4
消费者消费第97号产品，当前缓存区产品个数为3
消费者消费第98号产品，当前缓存区产品个数为2
消费者消费第99号产品，当前缓存区产品个数为1
消费者消费第100号产品，当前缓存区产品个数为0
```

可以看到，生产者和消费者异步运行，互斥访问缓冲区

解决饥饿问题：
生产者加锁成功后，如果缓冲区已经满了，则应该下一次就让消费者访问缓冲区，这样能够避免消费者的饥饿问题。同理，消费者加锁成功后，如果缓冲区已经空了，则应该下一次就让生产者访问缓冲区，这样才能避免生产者的饥饿问题。
为了实现上面的功能，我们使用两把锁的结构进行改进。以生产者进程为例，生产者可以为生产者锁加锁，如果缓冲区满了，则为消费者锁解锁，这样消费者就可以消费缓冲区中的产品，不会出现消费者的饥饿问题。

```
(base) lin@LindeMacBook-Pro 2 % ./ex2_producer_2
生产者生产第1号产品，当前缓存区产品个数为1
消费者消费第1号产品，当前缓存区产品个数为0
生产者生产第2号产品，当前缓存区产品个数为1
消费者消费第2号产品，当前缓存区产品个数为0
生产者生产第3号产品，当前缓存区产品个数为1
消费者消费第3号产品，当前缓存区产品个数为0
生产者生产第4号产品，当前缓存区产品个数为1
消费者消费第4号产品，当前缓存区产品个数为0
生产者生产第5号产品，当前缓存区产品个数为1
消费者消费第5号产品，当前缓存区产品个数为0
生产者生产第6号产品，当前缓存区产品个数为1
消费者消费第6号产品，当前缓存区产品个数为0
生产者生产第7号产品，当前缓存区产品个数为1
消费者消费第7号产品，当前缓存区产品个数为0
生产者生产第8号产品，当前缓存区产品个数为1
消费者消费第8号产品，当前缓存区产品个数为0
生产者生产第9号产品，当前缓存区产品个数为1
消费者消费第9号产品，当前缓存区产品个数为0
生产者生产第10号产品，当前缓存区产品个数为1
消费者消费第10号产品，当前缓存区产品个数为0
生产者生产第93号产品，当前缓存区产品个数为1
消费者消费第93号产品，当前缓存区产品个数为0
生产者生产第94号产品，当前缓存区产品个数为1
消费者消费第94号产品，当前缓存区产品个数为0
生产者生产第95号产品，当前缓存区产品个数为1
消费者消费第95号产品，当前缓存区产品个数为0
生产者生产第96号产品，当前缓存区产品个数为1
消费者消费第96号产品，当前缓存区产品个数为0
生产者生产第97号产品，当前缓存区产品个数为1
消费者消费第97号产品，当前缓存区产品个数为0
生产者生产第98号产品，当前缓存区产品个数为1
消费者消费第98号产品，当前缓存区产品个数为0
生产者生产第99号产品，当前缓存区产品个数为1
消费者消费第99号产品，当前缓存区产品个数为0
生产者生产第100号产品，当前缓存区产品个数为1
消费者消费第100号产品，当前缓存区产品个数为0
```

可以看到，没改进前的程序，缓冲区经常为满或空，运行时间也更长。改进后的程序，生产者和消费者间隔访问缓冲区，运行时间也更短。

3 线程池

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/pd共享文件夹/Threadpool-2023-DCS244-homework4-main$ ./exce.o
this is the task1 running in <60438>, the answer = 269444819
this is the task2 running in <60439>, the answer = 294988721
this is the task2 running in <60442>, the answer = 1718153830
this is the task1 running in <60444>, the answer = 1401117866
this is the task1 running in <60438>, the answer = 679948369
this is the task2 running in <60440>, the answer = 1320465518
this is the task2 running in <60439>, the answer = 259001443
this is the task1 running in <60441>, the answer = 560797458
this is the task1 running in <60438>, the answer = 661843652
this is the task2 running in <60442>, the answer = 856451012
this is the task1 running in <60443>, the answer = 2012910316
this is the task1 running in <60442>, the answer = 1137032919
this is the task2 running in <60440>, the answer = 394903400
this is the task2 running in <60444>, the answer = 1246699793
this is the task1 running in <60442>, the answer = 1850173682
this is the task2 running in <60441>, the answer = 55033668
this is the task2 running in <60439>, the answer = 768607171
this is the task2 running in <60443>, the answer = 758592638
this is the task2 running in <60440>, the answer = 1870277850
```

……

```
this is the task2 running in <60442>, the answer = 1869727004
this is the task2 running in <60441>, the answer = 1011696468
this is the task2 running in <60439>, the answer = 562397126
this is the task2 running in <60445>, the answer = 1695908483
this is the task2 running in <60444>, the answer = 393084325
this is the task1 running in <60438>, the answer = 955221886
this is the task1 running in <60441>, the answer = 240138715
this is the task2 running in <60444>, the answer = 260909246
this is the task2 running in <60442>, the answer = 1383260489
this is the task1 running in <60445>, the answer = 335741703
this is the task1 running in <60440>, the answer = 914543426
this is the task1 running in <60438>, the answer = 2014070935
this is the task2 running in <60439>, the answer = 292458020
this is the task2 running in <60441>, the answer = 1416425948
this is the task1 running in <60443>, the answer = 527518602
this is the task1 running in <60446>, the answer = 1281422422
parallels@ubuntu-linux-22-04-desktop:/media/psf/pd共享文件夹/Threadpool-2023-DCS244-homework4-main$
```

可以看到任务被线程池中的线程分别执行。

以下是部分代码展示：

每个线程都不断从任务队列中取出任务，这里在线程池中增加了一个互斥锁以确保不同线程对任务队列和信号量的互斥访问。

```
threadpool *Pool_init(int Maxthread)
{

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

    pool->flag = 1;

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

    // PLEASE ADD YOURs CODES
    pool->poolhead = NULL;
    pool->jobnum = 0;
    pool->Maxthread = Maxthread;
    pool->threads = (pthread_t *)malloc(sizeof(pthread_t) * Maxthread);
    for (int i = 0; i < Maxthread; i++)
    {
        pthread_create(&(pool->threads[i]), NULL, Job_running, (void *)pool);
    }
    sem_init(&(pool->sem), 0, 0);
    pthread_mutex_init(&(pool->lock), NULL);
```

```
    /*********************************************************************/

    return pool;
}

void Job_running(threadpool *pool)
{
    while (pool->flag)
    {
        /*********************************************************************/

        // PLEASE ADD YOURs CODES

        pthread_mutex_lock(&pool->lock);
        sem_wait(&(pool->sem));
        pthread_mutex_unlock(&pool->lock);

        if (pool->flag <= 0)
        {
            break;
        }
        Jobnode job = Pop(pool);
        if (job.pf != NULL)
        {
            job.pf(job.arg);
        }

        /*********************************************************************/
    }
    pthread_exit(0);
}

int Add_job(threadpool *pool, function_t pf, void *arg)
{
    /*********************************************************************/

    // PLEASE ADD YOURs CODES

    Jobnode job;
    job.pf = pf;
    job.arg = arg;
    if (Push(pool, job) == 0)
    {
        sem_post(&(pool->sem));

        return 0;
    }
    return -1;

    /*********************************************************************/
}

int Push(threadpool *pool, Jobnode data)
{
    /*********************************************************************/

    // PLEASE ADD YOURs CODES
    pthread_mutex_lock(&pool->lock);

    threadjob *job = (threadjob *)malloc(sizeof(threadjob));
    if (job == NULL)
    {
        return -1;
    }
    job->data = data;
    job->next = NULL;

    threadjob *p = pool->poolhead;
    if (p == NULL)
```

```
    {
        pool->poolhead = job;
    }
    else
    {
        while (p->next != NULL)
        {
            p = p->next;
        }
        p->next = job;
    }
    pool->jobnum++;

    pthread_mutex_unlock(&pool->lock);
    return 0;
    /*********************************************************************/
}

Jobnode Pop(threadpool *pool)
{
    /*********************************************************************/

    // PLEASE ADD YOURs CODES
    pthread_mutex_lock(&pool->lock);

    Jobnode job;
    job.pf = NULL;
    job.arg = NULL;

    threadjob *p = pool->poolhead;
    if (p == NULL)
    {
        pthread_mutex_unlock(&pool->lock);
        return job;
    }
    else
    {
        pool->poolhead = p->next;
        job = p->data;
        free(p);
        pool->jobnum--;
    }

    pthread_mutex_unlock(&pool->lock);
    return job;
    /*********************************************************************/
}

int Delete_pool(threadpool *pool)
{

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

    // PLEASE ADD YOURs CODES
    for (int i = 0; i < pool->Maxthread; i++)
    {
        sem_post(&(pool->sem));
    }
    for (int i = 0; i < pool->Maxthread; i++)
    {
        pthread_join(pool->threads[i], NULL);
    }
    free(pool->threads);

    threadjob *p = pool->poolhead;
    while (p != NULL)
    {
        threadjob *q = p;
```

```
        p = p->next;
        free(q);
    }

    sem_destroy(&(pool->sem));

    free(pool);

    return 0;
    /*************************************************************************/
}
```