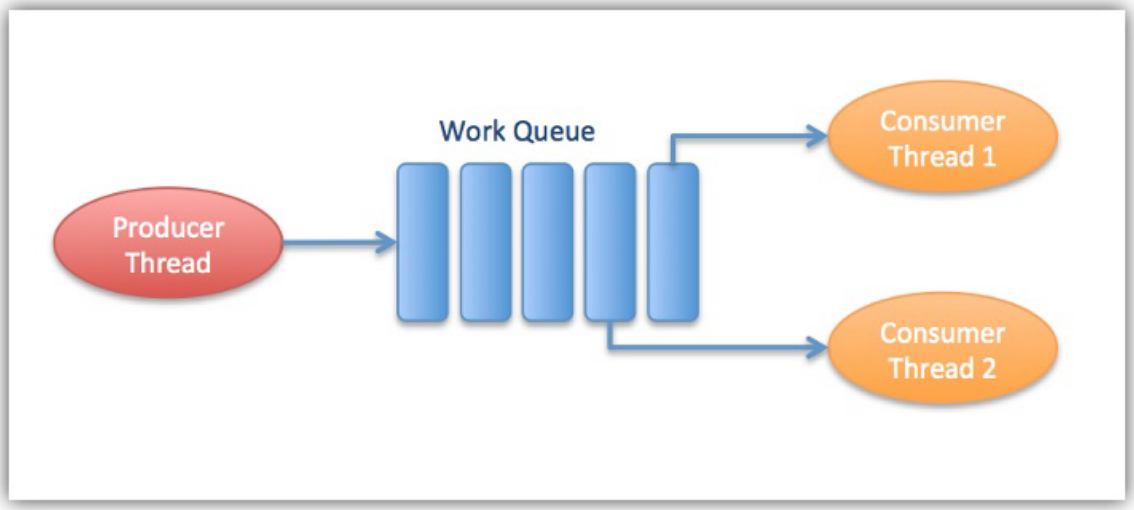
作业内容：

生产者消费者问题（也被称作有限缓冲问题） Producer–consumer problem (also known as the Bounded-buffer problem) 题目简述： 针对典型的生产者和消费者问题，使用OpenMP编程，实现生产者生成随机数，由消费者求和并打印的操作。请给出编程过程中遇到的问题分析，及其解决方案，并最终给出实验结果，同时说明结果的正确性。

详细说明： 这是一个典型的多进程/线程同步问题。可以形象地想象为一群生产者在生产产品，并将这些产品提供给消费者去消费。为了使生产者进程与消费者进程能够并发进行，在两者之间设置一个具有n个缓冲区的缓冲池，生产者进程将产品放入一个缓冲区中；消费者可以从一个缓冲区取走产品去消费。尽管所有的生产者进程和消费者进程是以异步方式运行，但它们必须保持同步：当一个缓冲区为空时不允许消费者去取走产品，当一个缓冲区满时也不允许生产者去存入产品。



https://images-cdn.shimo.im/Nz1wZVVyGucOB6Vm/image.png 先考虑较为简单的单生产者和单消费者问题，即仅有2个进程，分别为“生产者”和“消费者”，它们共享一个共有的、大小有限的数组，用队列的方式存取，来作为数据缓冲区。生产者的功能是重复生成数据，并将其放入缓冲区。而同时消费者的功能是使用数据，即将某一个数据从缓冲区中取出。核心问题是确保生产者不会在缓冲区满时放入数据，而消费者不会在缓冲区空时取出数据。一个简单的未解决该问题的参考代码在附件中，可适当改写或自行实现该过程，然后请选择OpenMP中合适的指令解决该问题。

接着考虑复杂一些的多生产者多消费者问题，即多个生产者向缓冲区中存入数据，而多个消费者从缓冲区中取出数据，缓冲区仍然采用队列读取的形式。核心问题是确保多个生产者之间、多个消费者之间、生产者消费者之间是互斥的。请将单生产者和单消费者问题代码扩展成该问题，并选择OpenMP中合适的指令解决该问题。

实现思路：

单生产者单消费者问题是多生产者与多消费者问题的子问题，多生产者多消费者问题比单生产者单消费者问题增加了多个生产者之间、多个消费者之间的约束。而实际上，单生产者单消费者也可以加上这样的约束，尽管这对原问题并没有什么影响，而且也不会影响程序实现的性能。因此，不妨直接考虑多生产者多消费者问题，然后在程序分配的线程数为2时，可分出单生产者单消费者，这样也就解决了单生产者单消费者问题。

为了简化模型，我将程序分配的线程数分成相等的两部分，作为生产者或消费者。当然，这样的假设在生产数量不能被线程数的一半整除时是有问题的，这可以通过宏来调整生产数量。

首先定义队列：

```
double *q;
q = (double *)malloc(MAX_QUEUE_SIZE*sizeof(double));
int front=0;
int back=0;
```

其中q是有限缓冲，MAX_QUEUE_SIZE是队列的最大长度，front指向队列头，back指向队列尾。

定义进队函数Enqueue和出队函数Dequeue

```
void Enqueue(double q[], double rand)
{
    q[back % MAX_QUEUE_SIZE]=rand;
    back++;
}

void Dequeue()
{
    front++;
}
```

这里向缓冲放入生产的随机数时要注意偏移量。由于back是只增不减的，它增加1个单位当且仅当生产并发送后，因此back的值实际就是目前生产的数量。由于缓冲一般比生产总数少，因此要将back对MAX_QUEUE_SIZE求余，这样就保证了数组不越界的情况。

出队只需要将队首指针front加1即可。

生产者的生产及发送的函数Send_msg：

```
int Send_msg(double q[])
{
    int flag = 0;
    //队满时不发送
    if(back - front == MAX_QUEUE_SIZE)
        flag = 0;
    //不满时发送
    else{
        flag = 1;
#   pragma omp critical
{
        //生产
        randy = (RAND_MULT * randy + RAND_ADD) % RAND_MOD;
//        printf("cri: %f\n", ((double) randy)/((double) RAND_MOD));
        //发送
        Enqueue(q,((double) randy)/((double) RAND_MOD));
}
    }
    return flag;
```

```
}
```

发送前首先要判断缓冲是否已满。这里front也是只增不减，增1当且仅当消费者消费。因此可以知道front的值代表着目前消费的数量。由上文可知，back的值代表着目前生产的数量，那么back - front就代表着缓冲队列大小。

如果back - front的值等于最大缓冲总量MAX_QUEUE_SIZE，那么说明缓冲已满，不能再生产了，返回0提示生产者生产失败。这也部分体现了生产者与消费者之间的互斥（不能无限生产）。

如果缓冲未满，那么可以生产并发送。这里要考虑到生产者之间互斥的原则，因此要在生产并发送的这两步设为临界区，这样就保证了生产者之间的互斥。（缓冲满时不用设为临界区，因为没有改变任何共享变量的值）

消费者的接收函数Try_receive

```
//从队列中收信息
double Try_receive(double q[])
{
    double rand = 0;
    int queue_size;
    queue_size = back - front;
    if(queue_size == 0)
        rand = -1;
    else{
        rand = q[front % MAX_QUEUE_SIZE];
        Dequeue();
    }
    return rand;
}
```

back - front == 0时说明缓冲为空，此时返回-1，提示消费者缓冲为空，不能获取生产产品。这也部分体现了生产者与消费者之间的互斥（不能无限消费）。

否则就获取当前指向的缓冲产品，然后出队。这里没有用临界区是因为整个函数都需要包含在临界区中（这会在main函数里实现），否则在queue_size时各个消费者就会获得相同的queue_size，然后取同一个缓冲的值，这违背了多消费者互斥的原则。

最后消费者要确认生产完成，消费完成的函数Done：

```
//如果队列大小为0时且生产随机数的线程的任务均完成了，消费结束
int Done(int done_sending, int thread_count)
{
    int queue_size = back - front;
    if(queue_size == 0 && done_sending == (thread_count / 2))
        return 1;
    else
        return 0;
}
```

如果队列大小为0时且生产随机数的线程的任务均完成了，消费结束。

主函数生产者部分：

```
int main(int argc, char* argv[])
{
    ...
    if(my_id < thread_count / 2){
        for(i = 0; i < (N * 2 / thread_count); i++){
            //发送失败，重发
            if(!Send_msg(q))
                i--;
        }
    #   pragma omp critical
        done_sending += 1;
    }
    ...
}
```

将前面一半的线程作为生产者，每个生产者生产`N * 2 / thread_count`个随机数。由于在`Send_msg`函数里已设了临界区，保证了生产者之间互斥，因此不用再写关于互斥的约束。如果生产发送失败，那么需要重新生产发送，将完成任务数i减去1即可。同时要记录完成生产任务的线程数done_sending来提示消费的结束。这要用到临界区来防止done_sending的值的不确定。

主函数消费者部分：

```
int main(int argc, char* argv[])
{
    ...
    else{
        while(!Done(done_sending, thread_count)){
        #  pragma omp critical
            r = Try_receive(q);
            if(r==-1){
                continue;
            }
            my_sum += r;
        }
    #   pragma omp critical
        sum += my_sum;
    }
    ...
}
```

由于消费者的接收函数Try_receive函数没有互斥约束，因此对Try_receive函数进行临界区约束，来保证消费者之间的互斥。当接收到-1时说明接收失败，则继续接受。否则就消费，将接收的值与自己计算的总值my_sum相加。

在接收完所有的随机数后，将各个消费者的my_sum加起来，得到所有随机数的总值，这也要用到临界区防止sum值的不确定。

实验结果截图：

N = 10000 时：

原程序：

```
PS C:\Users\28562\Desktop\new> .\prod_cons.exe
 In 0.000000 seconds, The sum is 5030.674031
```

并行程序的运行参数即线程数为4时是2生产者2消费者：

```
PS C:\Users\28562\Desktop> .\prod_cons.exe 4
 In 0.248000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 4
 In 0.237000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 4
 In 0.230000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 4
 In 0.240000 seconds, The sum is 5030.674031
```

并行程序的运行参数线程数为2时是单生产者单消费者

```
PS C:\Users\28562\Desktop> .\prod_cons.exe 2
 In 0.221000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 2
 In 0.220000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 2
 In 0.236000 seconds, The sum is 5030.674031
PS C:\Users\28562\Desktop> .\prod_cons.exe 2
 In 0.228000 seconds, The sum is 5030.674031
```

并行程序的代码：

```
/*
**  PROGRAM: A simple serial producer/consumer program
**
**  One function generates (i.e. produces) an array of random values.
**  A second functions consumes that array and sums it.
**
**  HISTORY: Written by Tim Mattson, April 2007.
*/
#include <omp.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>

#define N        10000
#define MAX_QUEUE_SIZE 200
/* Some random number constants from numerical recipies */
#define SEED       2531
```

```
#define RAND_MULT  1366
#define RAND_ADD   150889
#define RAND_MOD   714025
int randy = SEED;

//队列
int front = 0;
int back = 0;

//入队
void Enqueue(double q[], double rand)
{
    q[back % MAX_QUEUE_SIZE]=rand;
    back++;
}

//出队
void Dequeue()
{
    front++;
}

//发消息至队列
int Send_msg(double q[])
{
    int flag = 0;
    //队满时不发送
    if(back - front == MAX_QUEUE_SIZE)
        flag = 0;
    //不满时发送
    else{
        flag = 1;
#   pragma omp critical
{
        randy = (RAND_MULT * randy + RAND_ADD) % RAND_MOD;
//        printf("cri: %f\n", ((double) randy)/((double) RAND_MOD));
        Enqueue(q,((double) randy)/((double) RAND_MOD));
}
    }
    return flag;
}

//从队列中收信息
double Try_receive(double q[])
{
    double rand = 0;
    int queue_size;
    queue_size = back - front;
    if(queue_size <= 0)
        rand = -1;
    else{
        rand = q[front % MAX_QUEUE_SIZE];
        Dequeue();
    }
    return rand;
}

//如果队列大小为0时且生产随机数的线程的任务均完成了，程序结束
```

```
int Done(int done_sending, int thread_count)
{
    int queue_size = back - front;
    if(queue_size == 0 && done_sending == (thread_count / 2))
        return 1;
    else
        return 0;
}


int main(int argc, char* argv[])
{
    double sum, runtime;
    //double q[MAX_QUEUE_SIZE];
    double *q;
    q = (double *)malloc(MAX_QUEUE_SIZE*sizeof(double));
    int thread_count;
    int done_sending = 0;
    thread_count = strtol(argv[1], NULL, 10);
    sum = 0;
    runtime = omp_get_wtime();
#   pragma omp parallel num_threads(thread_count)
{
    double r;
    int i;
    double my_sum = 0;
    int my_id = omp_get_thread_num();
    if(my_id < thread_count / 2){
        for(i = 0; i < (N * 2 / thread_count); i++){
            //发送失败，重发
            if(!Send_msg(q))
                i--;
/*            for(j = 0;j < back;j++){
                printf("%d ",j);
                printf("%f\n",q[j]);
            }*/
        }
    #   pragma omp critical
        done_sending += 1;
    }
    else{
        while(!Done(done_sending, thread_count)){
        #  pragma omp critical
            r = Try_receive(q);
            if(r==-1){
                continue;
            }
            my_sum += r;

        }
    #   pragma omp critical
        sum += my_sum;
    }
}
    runtime = omp_get_wtime() - runtime;

    printf(" In %f seconds, The sum is %f \n",runtime,sum);
    free(q);
```

```
    return 0;
}
```