实验二

题目2 同步互斥--生产者-消费者

实验目的

实现一个使用条件变量进行同步的 生产者，消费者系统。

实验内容

实现一个使用条件变量进行同步的 生产者，消费者系统。

实验设计原理/步骤

生产者

生产者线程开启一个工作循环，首先对缓冲区队列加锁为lk，此处是采用一个全局变量锁来初始化一个lock，使用的是unique_lock，通过这个对象，可以在析构函数中，自动解锁unique_lock。

```
g_cv.wait(lk, []()
        { return g_data_deque.size() <= MAX_NUM; });
```

然后使用 g_cv 这个条件变量，进行wait。此处这个wait的是全局变量g_data_deque中的size，实现的功能是当缓冲区队列未满的，添加数据。
这里这个wait会导致当条件不满足的时候，线程继续沉睡。在代码中，是调用了一个返回bool值的lambda表达式，若返回false则阻塞，返回true则取消阻塞，同时，用代码中的写法，实际上是在一个循环中进行等待，防止虚假唤醒的问题。

这里返回true的情况是缓冲区小于最大值，相等于缓冲区未满，此时就会指针下移，然后将新的数据加入队列。

输出当生产者的情况后，再使用`g_cv.notify_all()`唤醒其他线程。

消费者

消费者线程开启一个工作循环，首先对缓冲区队列加锁为lk，此处是采用一个全局变量锁来初始化一个lock，使用的是unique_lock，通过这个对象，可以在析构函数中，自动解锁unique_lock。

```
g_cv.wait(lk, []
        { return !g_data_deque.empty(); });
```

然后使用 g_cv 这个条件变量，进行wait。此处这个wait的是全局变量g_data_deque中的empty，实现的功能是当缓冲区队列不为空时，取出数据。
这里这个wait会导致当条件不满足的时候，线程继续沉睡。在代码中，是调用了一个返回bool值的lambda表达式，若返回false则阻塞，返回true则取消阻塞，同时，用代码中的写法，实际上是在一个循环中进行等待，防止虚假唤醒的问题。

这里返回true的情况是非empty，也就是队列不为空，此时会取出数据，然后输出内容，最后唤醒其他线程。

main函数

在main函数中，我们首先创造了生产者，然后创造了消费者，最后使用

```
    for (int i = 0; i < PRODUCER_THREAD_NUM; i++)
    {
        arrRroducerThread[i].join();
    }

    for (int i = 0; i < CONSUMER_THREAD_NUM; i++)
    {
        arrConsumerThread[i].join();
    }
```

使主线程等待生产者消费者线程的结束，由于生产者和消费者都是while的无限循环，所以永不会结束，我们的主线程也就永不会结束。

实验结果及分析--条件变量和信号量在实现生产者消费者上的对比

当缓冲区没有数据的时候，消费者竞争缓冲区是没有意义的，但是只采用信号量的情况下，很难避免消费者对缓冲区的竞争，但是采用了条件变量后，如果缓冲区是空的，消费者们都会进入睡眠状态，从而避免了对缓冲的竞争，减少了CPU空转，提高了效率。

同时信号量本身只能做到+1 -1，在复杂的场景下相对的较难使用，但是经过仔细的设计后，能够使用信号量在实现条件变量。

程序代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
std::mutex g_cvMutex;          //二元互斥信号量
std::condition_variable g_cv; //条件变量

//缓冲区队列/buffer
std::deque<int> g_data_deque;
//缓冲区buffer最大数目
const int MAX_NUM = 30;
```

```
//缓冲区指针
int g_next_index = 0;

//生产者，消费者线程个数
const int PRODUCER_THREAD_NUM = 3;
const int CONSUMER_THREAD_NUM = 3;

void producer_thread(int thread_id)
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(500)); //线程延时、睡
眠
        //对缓冲区队列加锁
        std::unique_lock<std::mutex> lk(g_cvMutex);
        //当缓冲区队列未满时，继续添加数据
        g_cv.wait(lk, []()
              { return g_data_deque.size() <= MAX_NUM; }); //队列未满
        g_next_index++;                          //指针下移
        g_data_deque.push_back(g_next_index);              //数据加入队列
        std::cout << "producer_thread: " << thread_id << " producer data: " <<
g_next_index;
        std::cout << " queue size: " << g_data_deque.size() << std::endl;
        //唤醒其他线程
        g_cv.notify_all();
        //自动释放锁
    }
}

void consumer_thread(int thread_id)
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(550));
        //对缓冲区队列加锁
        std::unique_lock<std::mutex> lk(g_cvMutex);
        //检测条件是否达成：队列不为空，有数据
        g_cv.wait(lk, []
              { return !g_data_deque.empty(); });
        //互斥操作，从队列中取数据
        int data = g_data_deque.front();
        g_data_deque.pop_front();
        std::cout << "\tconsumer_thread: " << thread_id << " consumer data: ";
        std::cout << data << " deque size: " << g_data_deque.size() <<
std::endl;
        //唤醒其他线程
        g_cv.notify_all();
        //自动释放锁
    }
}

int main()
{
    std::thread arrRroducerThread[PRODUCER_THREAD_NUM];
    std::thread arrConsumerThread[CONSUMER_THREAD_NUM];

    for (int i = 0; i < PRODUCER_THREAD_NUM; i++)
    {
```

```
        arrRroducerThread[i] = std::thread(producer_thread, i);
    }

    for (int i = 0; i < CONSUMER_THREAD_NUM; i++)
    {
        arrConsumerThread[i] = std::thread(consumer_thread, i);
    }

    for (int i = 0; i < PRODUCER_THREAD_NUM; i++)
    {
        arrRroducerThread[i].join();
    }

    for (int i = 0; i < CONSUMER_THREAD_NUM; i++)
    {
        arrConsumerThread[i].join();
    }

    return 0;
}
```