# 题目：实验2-1 基于条件变量的同步互斥

# 实验目的

参照相关资料，采用条件变量（管程），实现进程、线程的同步互斥。

# 实验内容

通过基于条件变量的生产者-消费者问题来理解条件变量在同步互斥中的应用。

# 实验设计原理/步骤

## 3.1阅读相关资料，理解条件变量的定义

在C++11中，线程使用条件变量（condition\_variable）实现相互间的同步操作。条件变量利用线程间共享全局变量（即条件变量）进行同步，主要包括两个动作：

（1）一个线程等待“条件变量的条件成立”，当该条件不满足时，被阻塞、挂起，即进入等待状态；

（2）其它线程使“定义在条件变量上的条件成立”，给出信号，通知唤醒等待的线程。

条件变量与互斥锁/互斥量mutex、用户提供的判定条件相互配合，一起组合使用，可以实现生产者-消费者等复杂的同步互斥问题。条件变量可以原子地使得线程在唤醒时检查用户定义的判定条件，如果条件不满足就释放互斥锁，并入阻塞。

线程修改条件变量的动作为：

* + （1）获得一个定义在条件变量std::condition\_variable上的互斥锁std::mutex；
  + （2）当持有互斥锁后，修改条件变量；
  + （3）对条件变量执行notify\_one或notify\_all。说明：当执行notify动作时，不必持有锁

共享变量是原子性的，必须在mutex的保护下被修改，以便能够将对条件变量的修改改动正确notify发布到正在等待的线程。

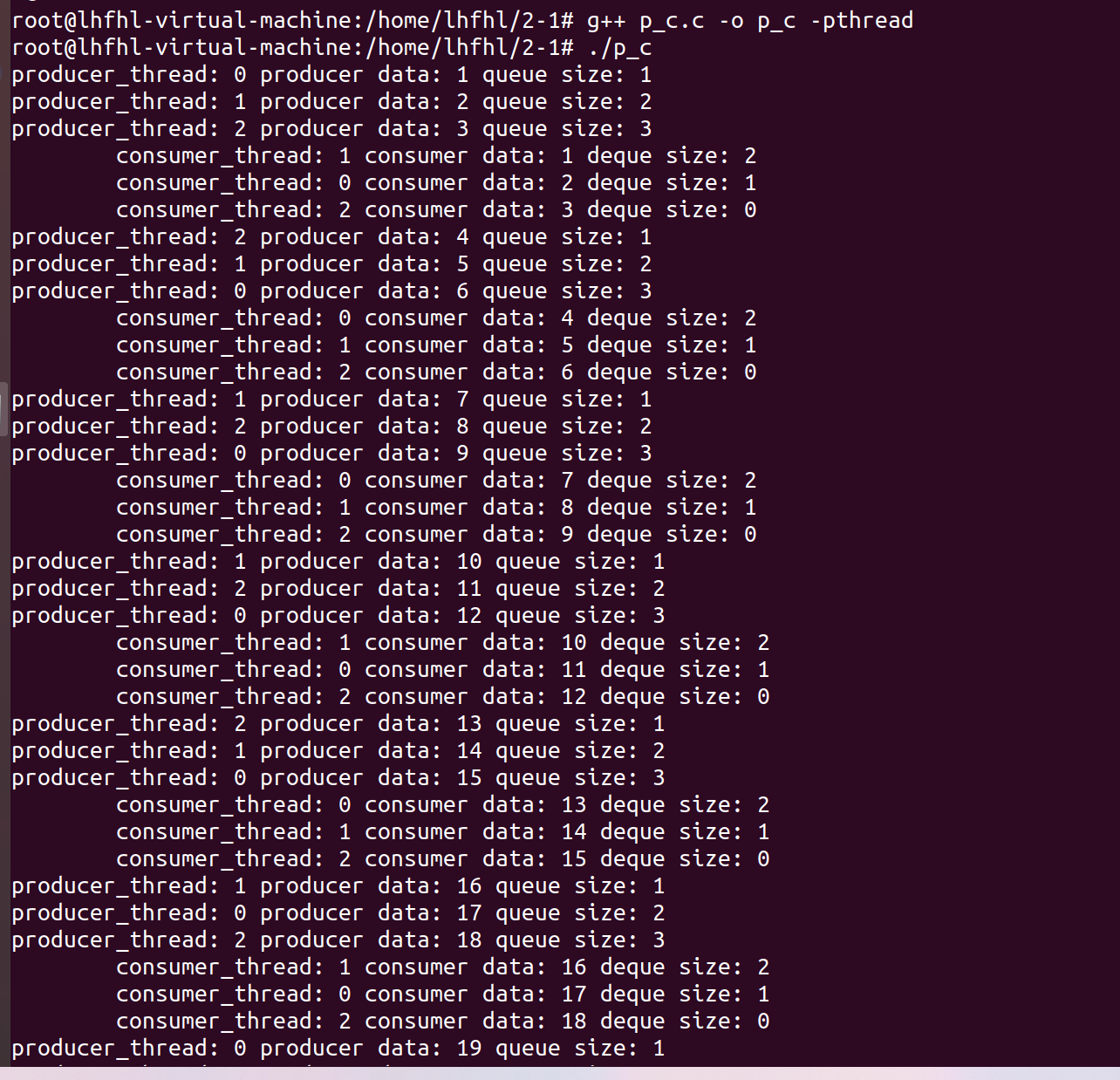
等待条件变量std::condition\_variable的线程必须：

* + （1）获取std::unique\_lock<std::mutex>，此处的互斥量mutex用于保护作为条件的共享变量；
  + （2）执行wait, wait\_for或者wait\_until. 这些等待动作原子性地释放mutex，并使得线程的执行暂停；
  + （3）当获得条件变量的通知notify，或者超时，或者一个虚假的唤醒，线程被唤醒，并且获得mutex. 然后线程检查条件是否成立，如果是虚假唤醒，就继续等待。

## 3.2编写代码如下，实现生产者-消费者问题



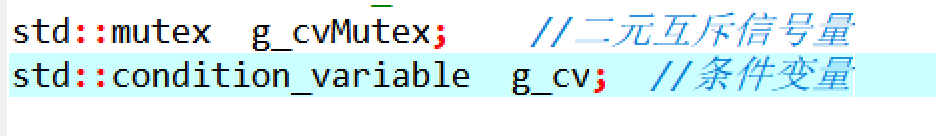
## 3.3编译执行，可以看到正确输出了



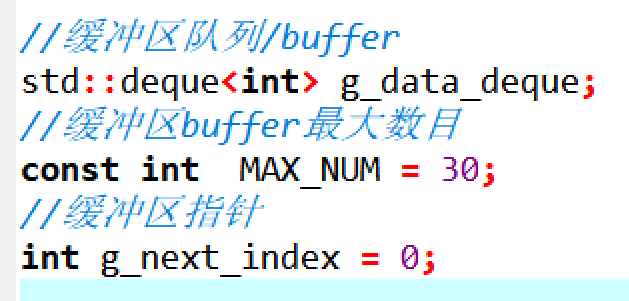
# 四、实验结果及分析

## 4.1代码分析

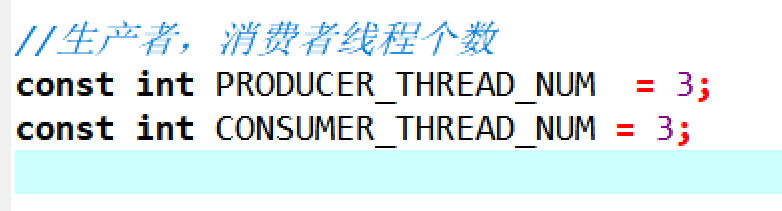
代码中首先定义了二元互斥信号量和条件变量如下：



然后定义了存放资源的缓冲区队列，以及缓冲区大小，还有缓冲区里存的数据的“指针”



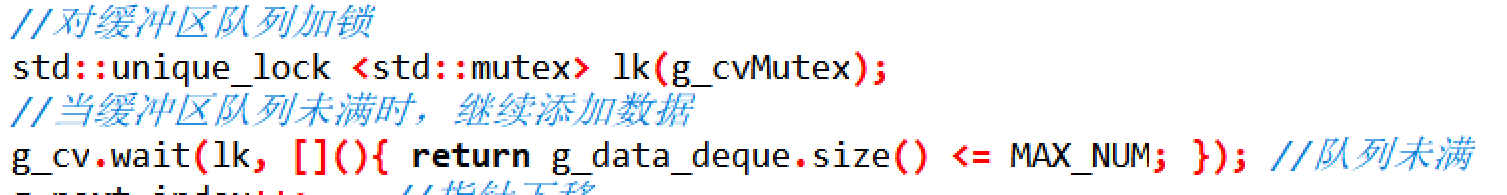
然后定义了生产者和消费者的线程个数：



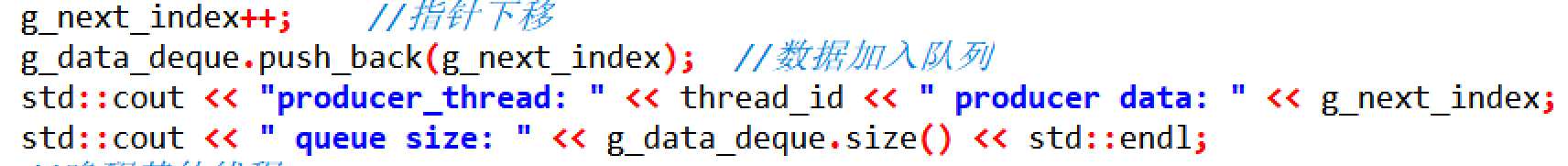
4.1.1分析生产者部分代码——void producer\_thread(int thread\_id)

内部是一个大while死循环，说明会一直反复执行。

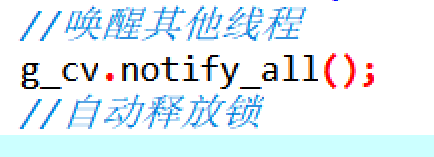
进入循环，会先获得锁，然后通过条件变量来判断是否要释放锁阻塞还是持有锁继续执行后续部分代码



若是持有锁继续执行后续部分代码，则就是按下述代码处理：



之后唤醒其他线程：



否则就是释放锁阻塞，等待唤醒

4.1.2分析消费者代码

类似生产者，略。

## 4.2整体分析

整个程序就是有三个生产者线程和三个消费者线程，然后通过join函数让主函数线程无法结束（因为生产者和消费者的线程都是死循环），

通过条件变量和互斥信号量来实现生产者和消费者的问题，最后的结果可以看到，每次生产者产生出的三个数据，都会被后续的消费者使用掉，循环往复。

## 4.3条件变量与信号量的区别

1. 使用条件变量可以一次唤醒所有等待者，而这个信号量没有的功能。
2. 条件变量只能结合互斥量做同步用；信号机制除了做同步，还能用于共享资源并发访问加锁等。

# 五、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 arrProducerThread[PRODUCER\_THREAD\_NUM]; //生产者线程数组

std::thread arrConsumerThread[CONSUMER\_THREAD\_NUM]; //消费者线程数组

for (int i = 0; i < PRODUCER\_THREAD\_NUM; i++)

{

arrProducerThread[i] = std::thread(producer\_thread, i); //依次调用生产者线程 0,1,2

}

for (int i = 0; i < CONSUMER\_THREAD\_NUM; i++)

{

arrConsumerThread[i] = std::thread(consumer\_thread, i);//依次调用消费者线程 0,1,2

}

for (int i = 0; i < PRODUCER\_THREAD\_NUM; i++)

{

arrProducerThread[i].join(); //等待生产者线程0,1,2结束

}

for (int i = 0; i < CONSUMER\_THREAD\_NUM; i++)

{

arrConsumerThread[i].join(); //等待消费者线程0,1,2结束

}

return 0;

}