《操作系统》课第 08 次实验报告

学院:	软件学院
姓名:	郁万祥
学号:	2013852
邮箱:	yuwanxiang0114@163.com
时间:	2022.11.4

0. 开篇感言

一个小问题：在使用 C++完成代码编写时候，会出现代码逻辑没有问题，但是输出结果存在小问题，经过查阅资料，C++中的 cout 输出并不是线程安全的，会出现内容交替出现的状况，而 printf 是线程安全的，所以输出改成 printf 就会避免这个问题。

1. 实验题目

生产者消费者问题

2. 实验目标

1) 严格按时序输出每个生产者、消费者的行为，其中包括生产产品 k、消费产品 k、进入临界区、存入产品、取出产品、离开临界区；
2) 需要考虑边界（某生产者生产第 K 个产品后所有生产者结束；某消费者消费第 K 个产品后所有消费者结束）
3) 需要考虑随机函数，生产者生产时需要一个随机时间；消费者消费时也需要一个随机时间；

4) 编号：无论生产者还是消费者都需要有编号；产品同样也需要编号；缓冲区的各个产品项也需要有编号；

5) 输出形式可以采用标准输出、图形动态显示及同时文字记录输出等方式，无论是生产者还是消费者，其主要输出内容如下：

a) 进入临界区前，输出某某编号（生产者/消费者）线程准备进入临界区

b) 进入临界区后，输出某某编号（生产者/消费者）线程已进入临界区

c) 离开临界区后，输出某某编号（生产者/消费者）线程已离开临界区

d) 生产者生产一个产品时，需要输出产品信息；

e) 生产者将产品放入缓冲区时，需要输出相关信息；

f) 消费者将产品从缓冲区取出时，需要输出相关信息；

g) 消费者消费一个产品时，需要输出产品信息；

6) *不能出现竞态

7) **不能出现忙等待

3. 原理方法

1、简介相互制约（互斥）：因为进程在并发执行的时候共享临界资源而形成的相互制约的关系，需要对临界资源互斥地访问。

2、直接制约关系（同步）：多个进程之间为完成同一任务而相互合作而形成的制约关系。

3、临界资源：同一时刻只允许一个进程可以访问的资源称之为临界资源，诸进程之间应采取互斥方式，实现对临界资源的共享。

4、临界区：进程中访问临界资源的那段代码。显然若能保证诸进程互斥的进入自己的临界区，便可实现进程间对临界资源的互斥访问。

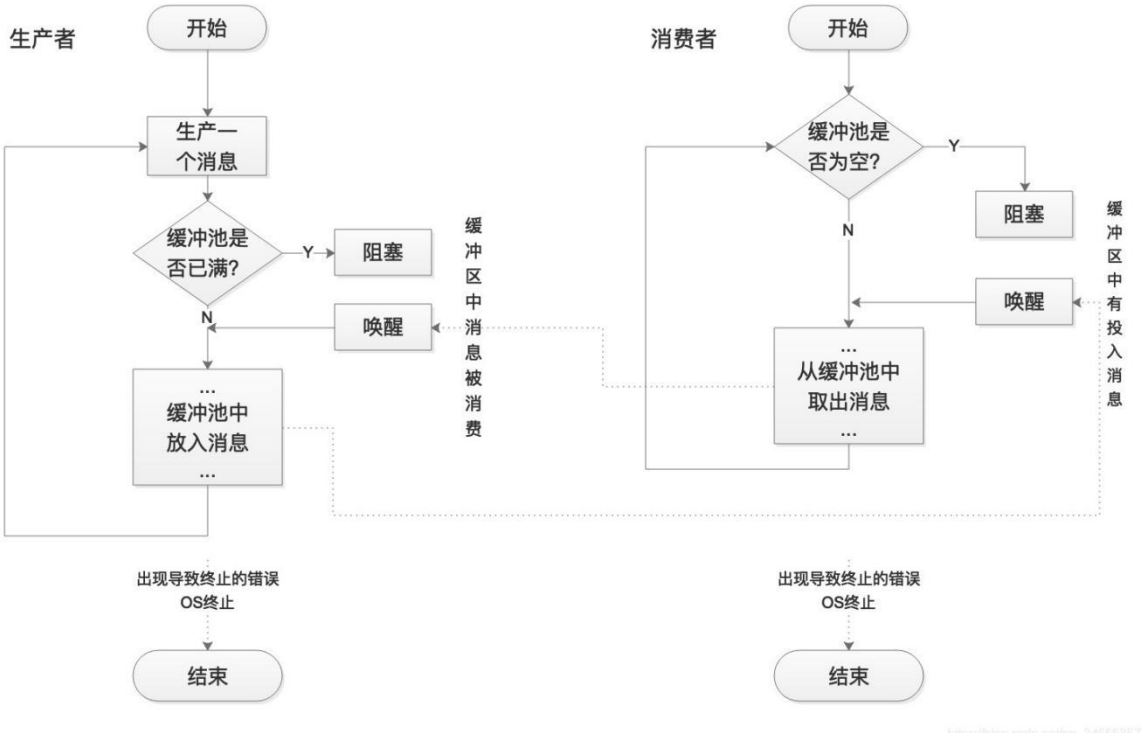
5、竞态：一个线程读取共享变量并以该共享变量为基础进行计算的期间另外的

一个线程更新了该共享变量的值而导致的干扰（读取脏数据）或冲突（丢失更新）的结果。

6、忙等待：处于忙等状态的进程是一直占有着处理机去不断的判断临界区是否可以进入，在此期间，进程一直在运行（忙死了都，但是都是白忙），这也就是“忙等”状态。

4. 具体步骤

1、分析问题



2、信号量设置：

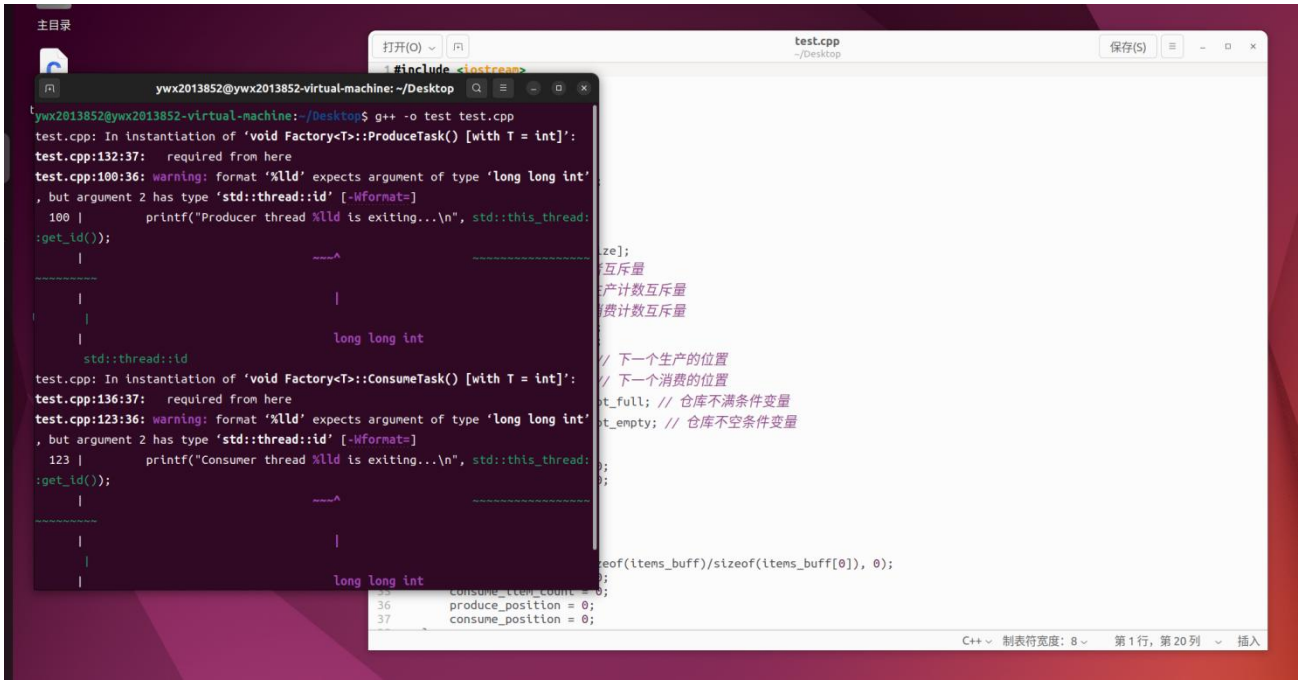
由于生产和消费行为都会修改数据，因此两者操作必须互斥，需引入生产消费互斥锁。当我们要生产（或消费）一定数量的产品时，需要计数判断是否已经完成工作；多个生产者进行生产时，都会对生产的计数变量进行修改，因此需引入生产计数互斥锁和消费计数互斥锁，保证同时只有一个生产（或消费）进程对计数变量进行修改。

mutex mtx; // 生产者消费者互斥量　保证只有生产者（或消费者）进入临界区

mutex produce_mutex; // 生产计数互斥量　保证只有一个生产者

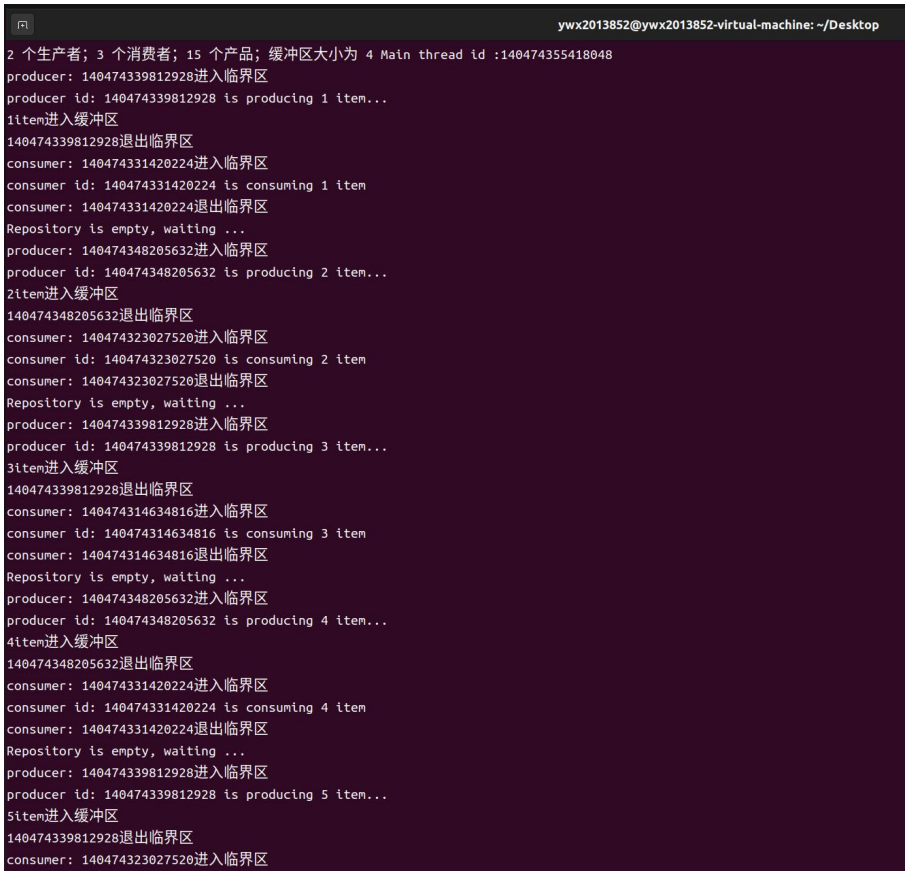
mutex consume_mutex; // 消费计数互斥量　　保证只有一个消费者

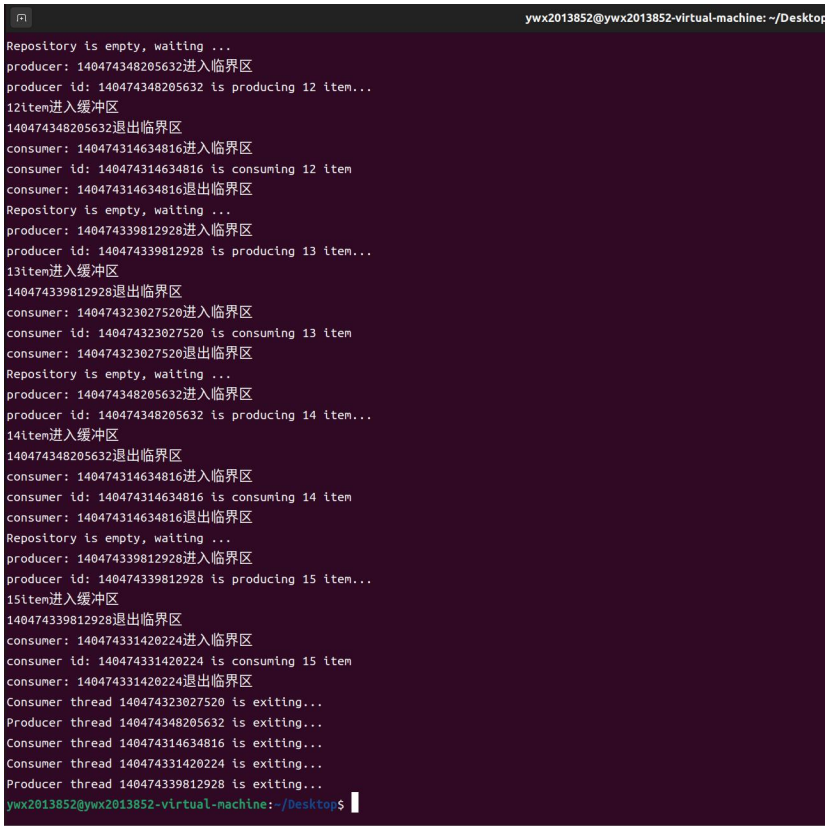
3、编写程序并编译



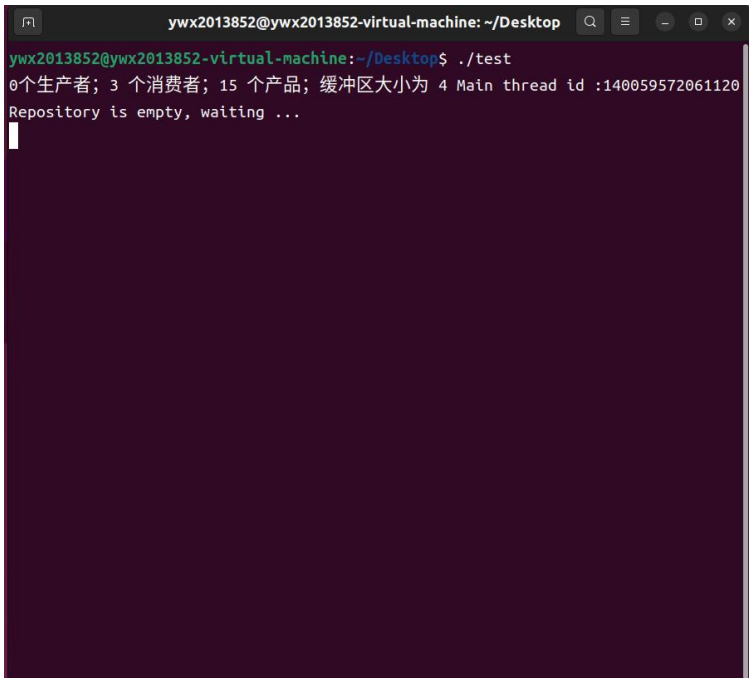
4、运行测试

a) 2 个生产者；3 个消费者；15 个产品；缓冲区大小为 4

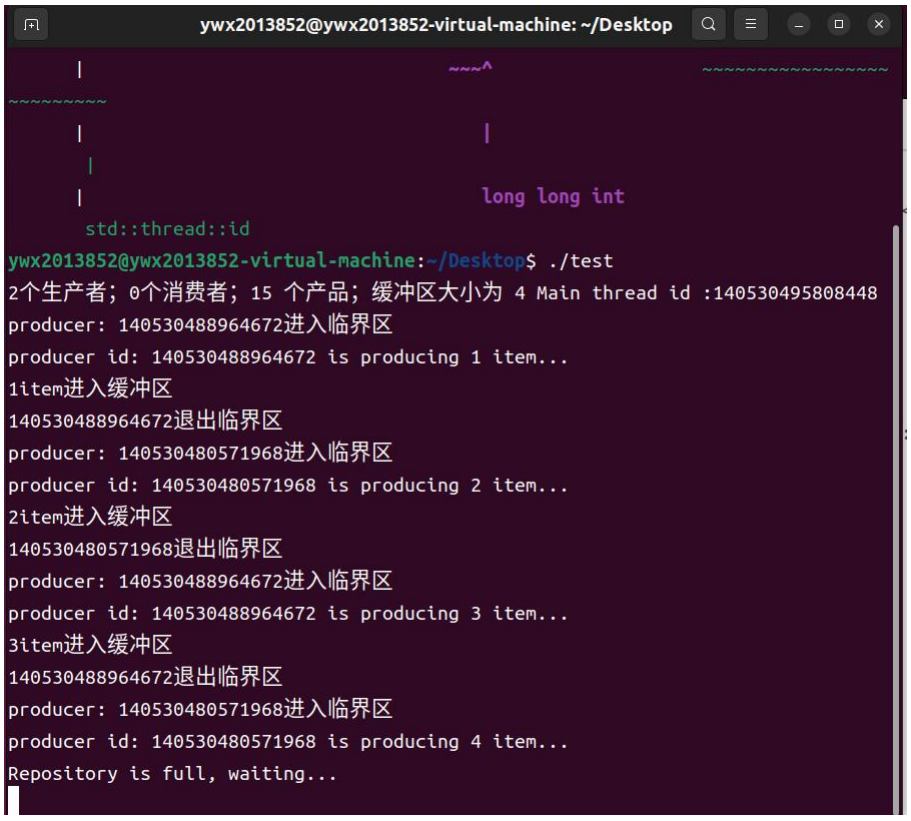




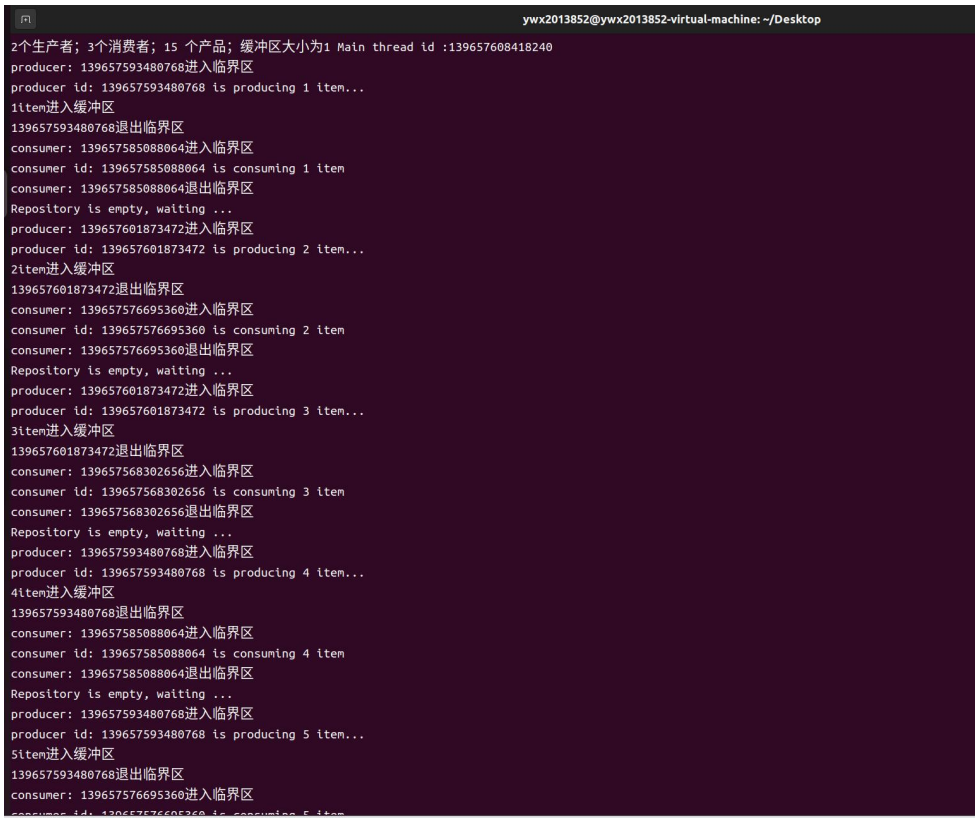
b) 0 个生产者；3 个消费者；15 个产品；缓冲区大小为 4

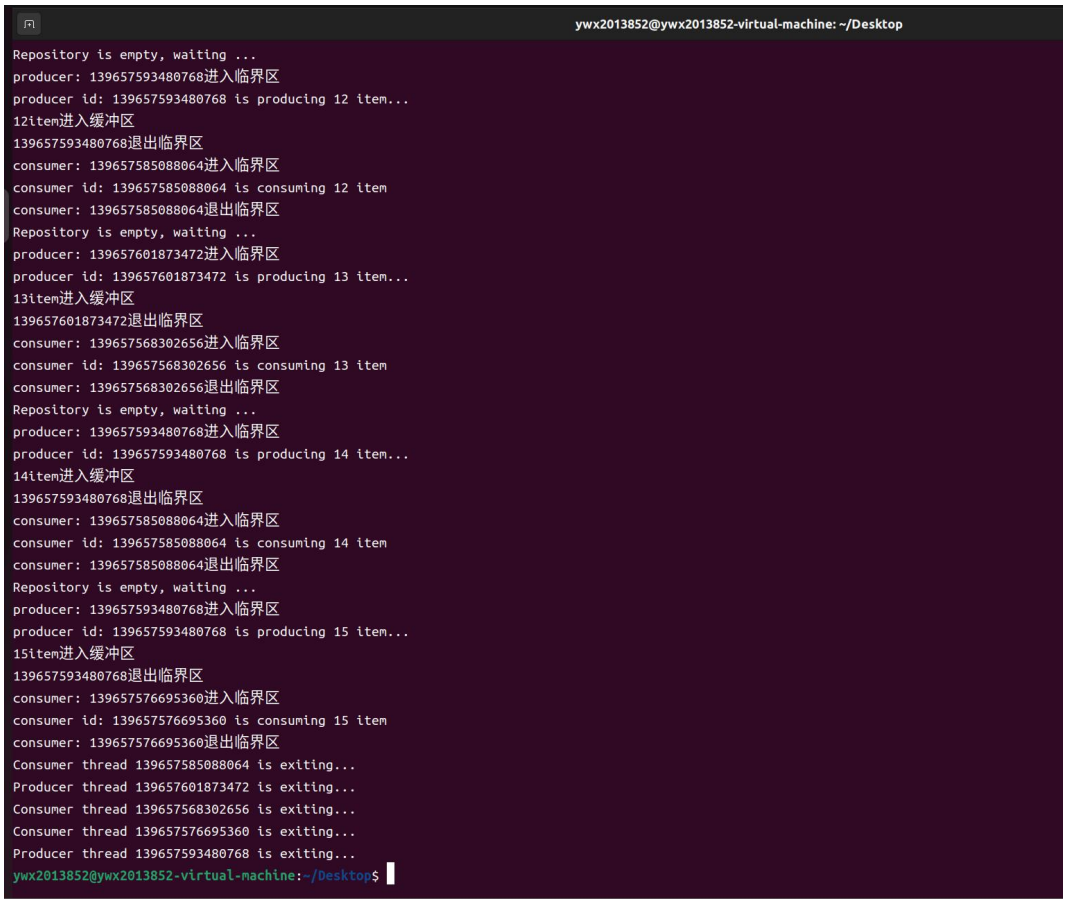


c) 2 个生产者；0 个消费者；15 个产品；缓冲区大小为 4

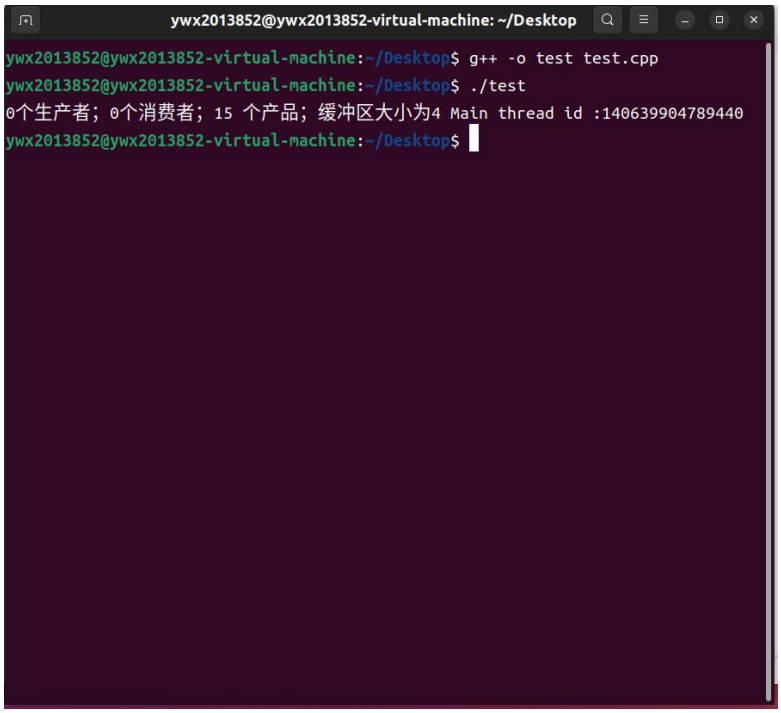


d) 2 个生产者；3 个消费者；15 个产品；缓冲区大小为 1

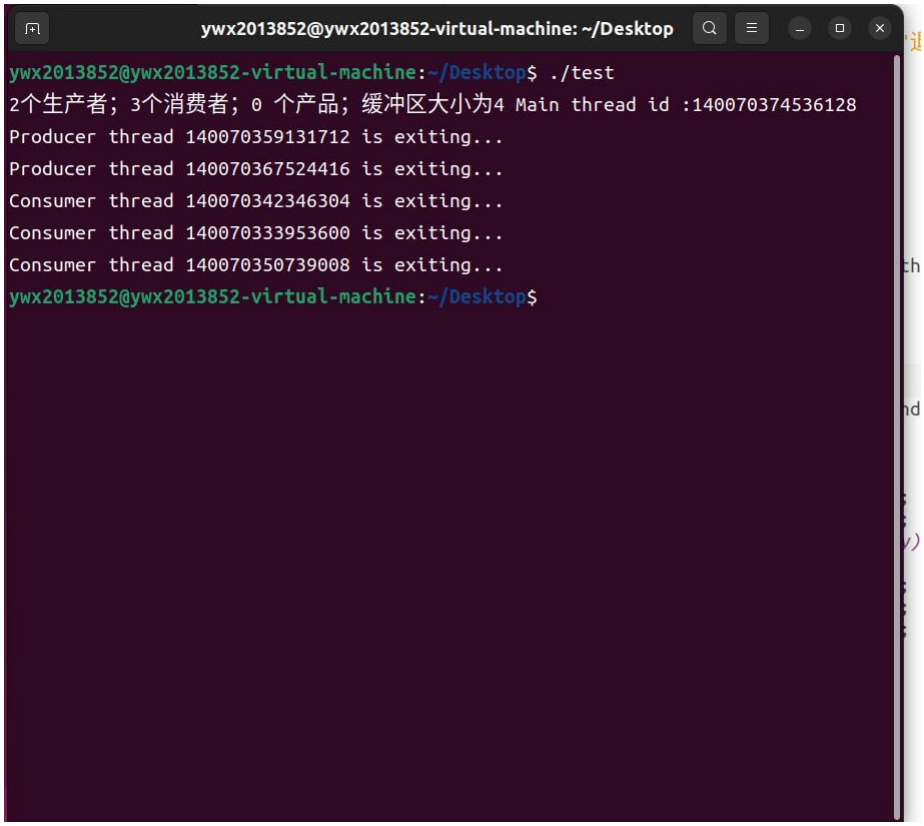




e) 0 个生产者；0 个消费者；15 个产品；缓冲区大小为 4



f) 2 个生产者；3 个消费者；0 个产品；缓冲区大小为 4



5. 总结心得

其实多线程问题，大部分问题是很好理解的，甚至对于编程语言，他们已经有了一套成熟的方法，然而，作为一种高效方式，作为开发者，特别需要额外注意的就是线程安全，防止竞争已经忙等待，这通过信号量的设置，通过互斥的访问临界区，会使得无论是逻辑上还是代码上都比较简洁，就像是一个人进去关上了门，别人就无法进入，屋内的人出来才会打开门，同时对于内部的信号量又是如此，这样就可以保证线程的互斥访问。

6. 参考资料

源码：

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <unistd.h>
#include <thread>
using namespace std;

const int kProduceItems = 0;
const int kRepositorySize = 4;

template<class T>
class Repository {
public:
    T items_buff[kRepositorySize];
```

```
    mutex mtx; //  生产者消费者互斥量
    mutex produce_mutex; //  生产计数互斥量
    mutex consume_mutex; //  消费计数互斥量
    size_t produce_item_count;
    size_t consume_item_count;
    size_t produce_position; //  下一个生产的位置
    size_t consume_position; //  下一个消费的位置
    condition_variable repo_not_full; //  仓库不满条件变量
    condition_variable repo_not_empty; //  仓库不空条件变量

    Repository() {
        produce_item_count = 0;
        consume_item_count = 0;
        produce_position = 0;
        consume_position = 0;
    };

    void Init() {
        fill_n(items_buff, sizeof(items_buff)/sizeof(items_buff[0]), 0);
        produce_item_count = 0;
        consume_item_count = 0;
        produce_position = 0;
        consume_position = 0;
    }
};

template<class T>
class Factory {
private:
```

```
Repository<T> repo;


void ProduceItem(T item) {
    unique_lock<mutex> lock(repo.mtx);
    while ((repo.produce_position+1) % kRepositorySize == repo.consume_position) {
        cout << "Repository is full, waiting..." << endl;
        (repo.repo_not_full).wait(lock); // 阻塞时释放锁，被唤醒时获得锁
    }
    repo.items_buff[repo.produce_position++] = item;
    if (repo.produce_position == kRepositorySize)
        repo.produce_position = 0;
    (repo.repo_not_empty).notify_all(); // 唤醒所有因空阻塞的进程
    lock.unlock();
}


T ConsumeItem() {
    unique_lock<mutex> lock(repo.mtx);
    while (repo.consume_position == repo.produce_position) {
        cout << "Repository is empty, waiting ..." << endl;
        (repo.repo_not_empty).wait(lock);
    }
    T data = repo.items_buff[repo.consume_position++];
    if (repo.consume_position == kRepositorySize)
        repo.consume_position = 0;
    (repo.repo_not_full).notify_all();
    lock.unlock();
    return data;
}
```

```
public:
    void Reset() {
        repo.Init();
    }

    void ProduceTask() {
        bool ready_to_exit = false;
        while (true) {
            sleep(1);
            unique_lock<mutex> lock(repo.produce_mutex);

            if (repo.produce_item_count < kProduceItems) {
                ++(repo.produce_item_count);
                T item = repo.produce_item_count;\
                cout<< "producer: "<<this_thread::get_id()<<"进入临界区"<<endl;
                cout << "producer id: "<< this_thread::get_id() << " is producing "
                        << item << " item..." << endl;
                ProduceItem(item);
                cout<<item<<"item"<<"进入缓冲区"<<endl;
                cout<<this_thread::get_id()<<"退出临界区"<<endl;
            } else {
                ready_to_exit = true;
            }

            lock.unlock();
            // sleep(1);
            if (ready_to_exit)
                break;
        }
```

```
        printf("Producer thread %lld is exiting...\n", std::this_thread::get_id());
    }

    void ConsumeTask() {
        bool ready_to_exit =false;
        while (true) {
            sleep(1);
            unique_lock<mutex> lock(repo.consume_mutex);

            if (repo.consume_item_count < kProduceItems) {
                T item = ConsumeItem();
                cout<< "consumer: "<<this_thread::get_id()<<"进入临界区"<<endl;
                cout << "consumer id: " << this_thread::get_id() << " is consuming "
                        << item << " item" << endl;
                ++(repo.consume_item_count);
                cout<< "consumer: "<<this_thread::get_id()<<"退出临界区"<<endl;
            } else {
                ready_to_exit = true;
            }

            lock.unlock();
            // sleep(1);
            if (ready_to_exit)
                break;
        }
        printf("Consumer thread %lld is exiting...\n", std::this_thread::get_id());
    }
};
```

```
int main() {
    cout<<"2 个生产者；3 个消费者；0 个产品；缓冲区大小为 4 ";
    cout << "Main thread id :" << this_thread::get_id() << endl;

    Factory<int> myfactory;

    thread producer1(&Factory<int>::ProduceTask, &myfactory);
    thread producer2(&Factory<int>::ProduceTask, &myfactory);
    //thread producer3(&Factory<int>::ProduceTask, &myfactory);

    thread consumer1(&Factory<int>::ConsumeTask, &myfactory);
    thread consumer2(&Factory<int>::ConsumeTask, &myfactory);
    thread consumer3(&Factory<int>::ConsumeTask, &myfactory);

    producer1.join();
    producer2.join();
    //producer3.join();

    consumer1.join();
    consumer2.join();
    consumer3.join();

    return 0;
}
```