# 操作系统\_生产者消费者设计方案

2152831陈峥海

## 1.问题描述

* 有界缓冲区有20个存储单元，存取数据为1-20这20个整型数；
* 当生产者和消费者对有界缓冲区进行操作后,即时显示有界缓冲区的全部内容、当前位置、生产者/消费者进程的标识符；
* 生产者和消费者各有两个以上；

## 2. 运行环境

**编译环境**: Visual Studio 2022

**编译器**: mingw64(Windows)

**语言:** c++

## 3.问题解析

这是一个多线程问题，缓存区的大小是固定的。生产者的作用是生产数据并放入缓冲区，消费者的作用是消耗缓存区中的数据。对于这些线程来说，缓存区是一个共享的数据空间。而问题的关键在于：当缓存区满了，生产者不能进行生产；当缓存区为空，消费者不能进行消耗。

### 3.1缓存区设计

缓存区，我使用队列这一数据结构来存储。原因是队列是先进先出的逻辑，能使每一部分空间都拥有“平均的”被使用的概率。缓存区是被共享的，当有多个线程进行访问时，我们需要保证每一次访问都是唯一的，即一次只能有一个线程访问缓存区。如果一次有多个线程访问缓存区，那么很可能使得缓存区一次突然增加或消耗多个数据，导致溢出或异常消耗。怎么实现呢？就需要使用互斥锁mutex来保证访问的唯一性。

生产者生产的商品push到队列之中，而消费者消费之后，队列进行pop操作

### 3.2多线程控制

当某个生产者线程获得了锁，却发现缓存区已经满了，那么它就应该将缓存区的使用权移交出来，进入等待队列，直到被唤醒。实现此过程，需要使用条件变量（对于condition\_variable，其作用是用来同步线程，它能使当前线程进入wait状态，即释放使用的互斥锁，直到被notify来重新获取互斥锁的使用权）。逻辑为若此时缓存区已满，则用condProducer.wait(lockerProducer)释放缓存区资源，使得别的线程获得互斥锁（若是生产者线程获得，则其也进入wait状态；若是消费者线程，则可以进行消费了）。消费者线程同理。

当任意生产者生产完毕后，就唤醒notify一个消费者的条件变量（可能并没有在等待队列的消费者线程，但这样能保证如果有的话，一定会被唤醒）。消费者线程同理。

具体实现代码如下

condition\_variable condProducer;   
condition\_variable condConsumer;   
void ProducerActor()   
{   
 unique\_lock<mutex> lockerProducer(m);//创建锁，保证缓存区的访问是互斥的   
 while (Q.size() >= MAX\_CACHE\_LENGTH) {   
 //当前缓存区已经满了，那么该进程需要被暂停   
 cout << "当前缓冲区已满，等待" << endl;   
 condProducer.wait(lockerProducer);//先释放互斥锁，等待被唤醒   
 }   
 //生产商品   
 ID = ID % 20 + 1;   
 Product temp = { ID, "\*\*\*" };   
 Q.push(temp);   
 condConsumer.notify\_one();   
}   
void ConsumerActor()   
{   
 unique\_lock<mutex> lockerConsumer(m);//创建锁，保证缓存区的访问是互斥的   
 while (Q.empty()) {   
 cout << "当前缓存区是空的，等待" << endl;   
 //当前缓存区是空的了，那么该进程需要被暂停   
 condConsumer.wait(lockerConsumer);//先释放互斥锁，等待被唤醒   
 }   
 //消耗商品   
 Product temp = Q.front();   
 Q.pop();   
 condProducer.notify\_one();   
}

### 3.3 互斥锁的使用

metux用来对缓存区进行访问控制，有上锁和解锁两个步骤，这里使用unique\_lock进行所有权的转移，它能自动管理mutex的上锁和解锁。

mutex m;// 缓存区的访问是互斥操作，即一个只能由一个进程访问   
unique\_lock<mutex> lockerConsumer(m);//创建锁，保证缓存区的访问是互斥的

## 4.实现

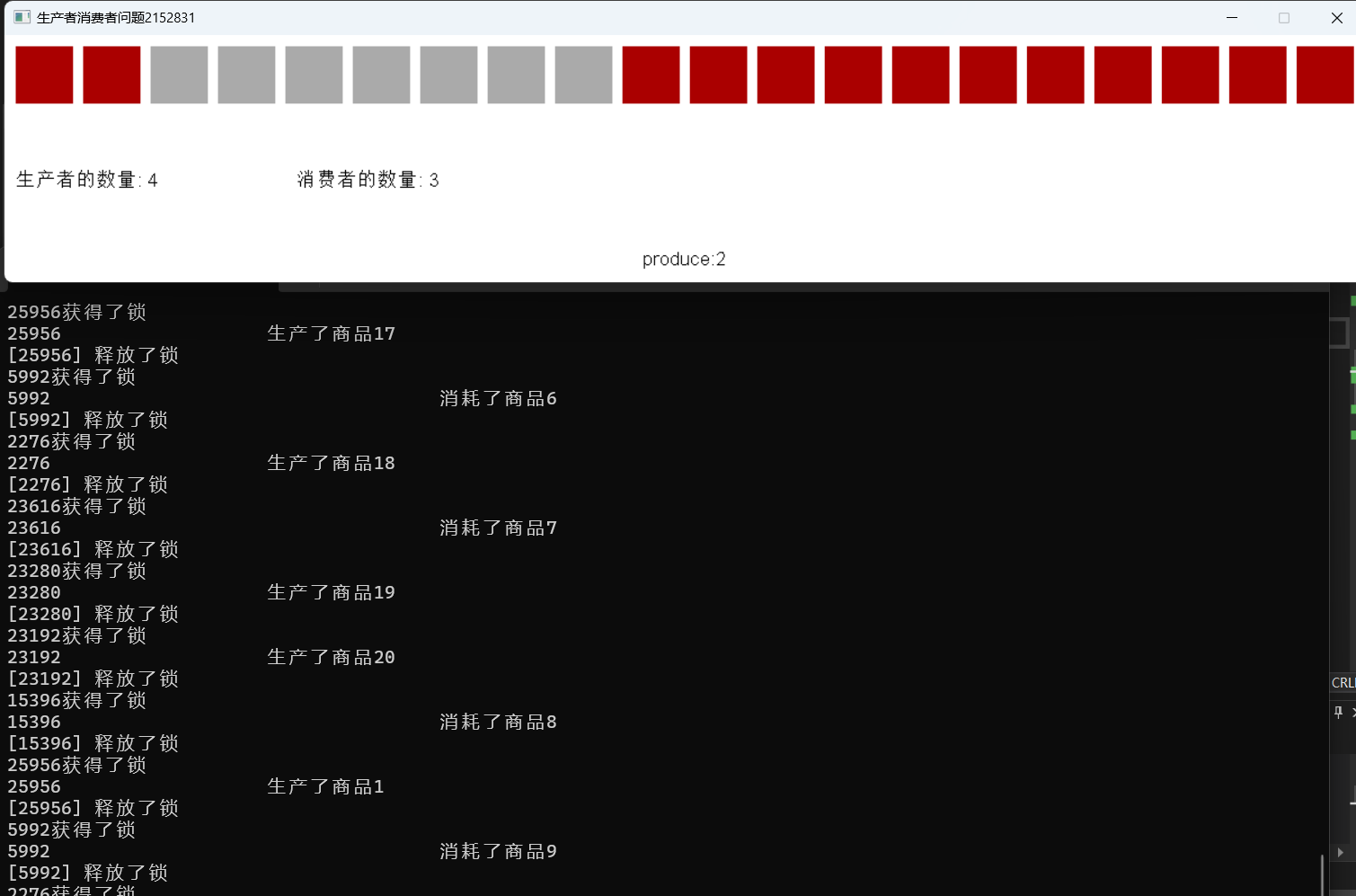
### 4.1界面与初始输入

首先进行生产者和消费者数量的输入，之后绘制20的方块代表缓存区的20个资源，当某个资源被使用则将其标红，反之则为灰色。这里使用easyX进行简单的图形绘制。

cout << "请输入生产者的数量：";   
cin >> producerNum;   
cout << "请输入消费者的数量：";   
cin >> consumerNum;   
draw();   
   
//1行20列的图像   
void draw()   
{   
 initgraph(WINDOW\_WIDTH, WINDOW\_HEIGHT);   
 setbkcolor(WHITE);   
 cleardevice();   
   
 // draw cells   
 for (int row = 1; row <= ROWS; row++)   
 {   
 for (int col = 1; col <= COLS; col++)   
 {   
 DrawCell(row, col, LIGHTGRAY);   
 }   
 }   
}

### 4.2输出与界面更改

为了使线程的进行更加直观，在命令行会输出各个线程和缓存区的情况，而生产者和消费者生产或消费之后，会调用相关的函数更改界面。



void SetCellRed(int row, int col)   
{   
 int left = (col - 1) \* (CELL\_SIZE + MARGIN) + MARGIN;   
 int top = (row - 1) \* (CELL\_SIZE + MARGIN) + MARGIN;   
 int right = left + CELL\_SIZE;   
 int bottom = top + CELL\_SIZE;   
 setfillcolor(RED);   
 solidrectangle(left, top, right, bottom);   
}   
   
void SetCellGray(int row, int col)   
{   
 int left = (col - 1) \* (CELL\_SIZE + MARGIN) + MARGIN;   
 int top = (row - 1) \* (CELL\_SIZE + MARGIN) + MARGIN;   
 int right = left + CELL\_SIZE;   
 int bottom = top + CELL\_SIZE;   
 setfillcolor(LIGHTGRAY);   
 solidrectangle(left, top, right, bottom);   
}

### 4.3线程的开启

在c++中，线程创建之后就会被启动，而若未进行设计，当前作用域结束或者说是主进程运行完后，这些线程会被关闭。为了避免这个问题，使用join语句，保证线程结束之前会一直阻塞主进程

void ProducerTask()   
{   
 while (1)   
 ProducerActor();   
}   
   
void ConsumerTask()   
{   
 while (1)   
 ConsumerActor();   
}   
void ThreadInit(int producerNum, int consumerNum)   
{   
 vector<thread> producerThread;   
 vector<thread> consumerThread;//消费者线程   
 for (int i = 0; i < producerNum; i++)   
 producerThread.push\_back(thread(ProducerTask));//创建生产者进程   
   
 for (int i = 0; i < consumerNum; i++)   
 consumerThread.push\_back(thread(ConsumerTask));//创建消费者进程   
   
 //使用join语句使得线程得以完整执行   
 for (int i = 0; i < producerNum; i++) {   
 if (producerThread[i].joinable())   
 producerThread[i].join();   
 }   
 for (int i = 0; i < consumerNum; i++) {   
 if (consumerThread[i].joinable())   
 consumerThread[i].join();   
 }   
}