****

**实 验 报 告**

**实验课程： 操作系统课程实践**

**学生姓名： 马星**

**学 号： 5418122020**

**专业班级： 计算机科学与技术(卓越)221班**

**2024年11月1日**

** 南昌大学实验报告**

**---（2）编程模拟进程间的同步和互斥**

学生姓名： 马星 学 号： 541812020 专业班级：计算机科学与技术(卓越)221班

实验类型：□ 验证 □ 综合 ■ 设计 □ 创新 实验日期： 实验成绩：

# 一、实验目的

通过实验加强对进程同步和互斥的理解，学会实验信号量解决资源共享问题，掌握信号量的定义、赋初始值及Wait/Signal操作，并掌握进程（线程）的创建和调用方法。

# 二、实验内容

多个进程并发运行的简单的进程管理模拟，根据用户的需求来创建n个进程(n为即满足用户需求，又不超出系统可容的最大进程数),各进程互斥地访问使用临界资源 S1、S2。具体可以编写程序实现生产-消费者问题或读者-写者问题。

# 三、实验要求

1. 需写出设计说明；
2. 设计实现代码及说明
3. 运行结果

# 四、主要实验步骤

## (1) 生产者-消费者问题模拟

### 1、设计说明：

有多个生产者和多个消费者共享一个数据队列

生产者: 负责生成数据(整型随机数), 并放入共享队列中

消费者: 从队列中取出数据进行处理

使用信号量 empty 和 full 来控制对共享队列的访问，确保生产者不会在队列满时继续生产，消费者不会在队列空时继续消费。

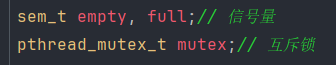
使用互斥锁 mutex 来保护对共享资源的访问，防止多个线程同时修改共享队列导致数据不一致。

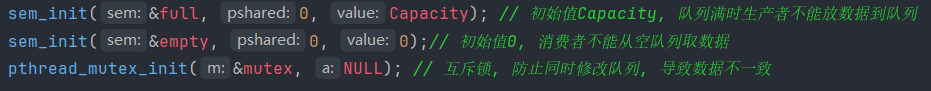
### 2、设计实现代码及说明：

#### ①说明:



临界资源是一个固定大小的队列 queue，使用循环数组表示, 队列的最大容量为Capacity



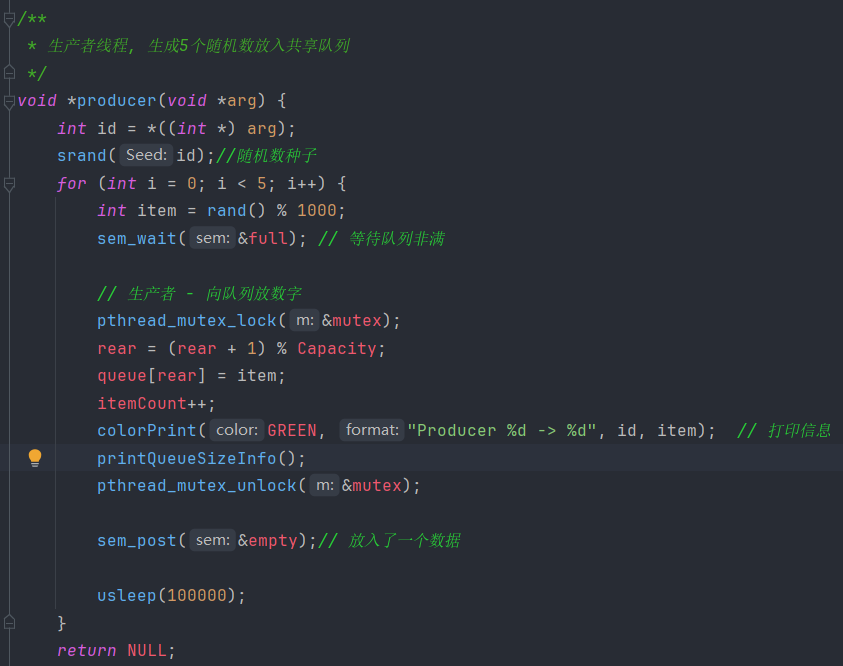


定义empty和full两个信号量

full: 初始值Capacity, 当full减为0时, 表示队列满, 此时生产者无法进入临界区, 需要等待消费者从队列中取数据

empty: 初始值0, 表示队列空, 此时消费者无法进入临界区, 需要等待生产者放入数据

互斥锁: 在临界区前后加锁, 保证访问临界资源是原子的





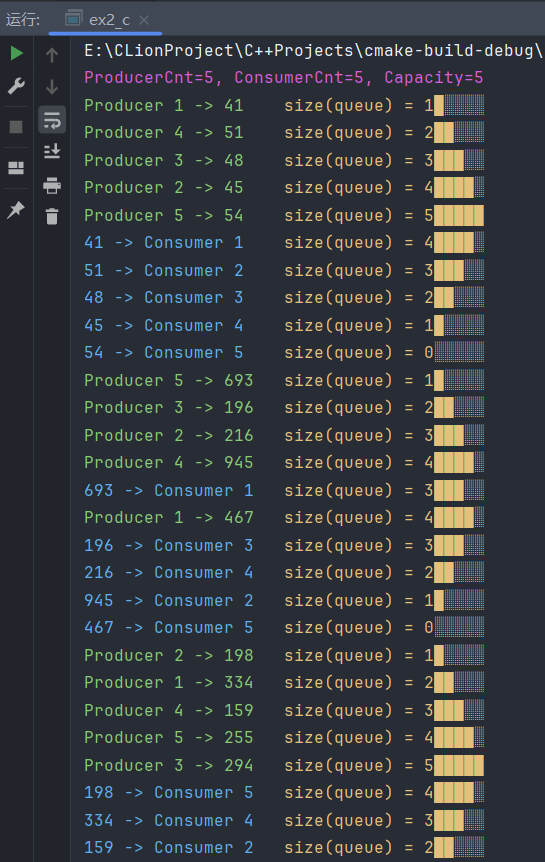
创建生产者和消费者线程，每个线程执行相应的函数 producer 和 consumer。

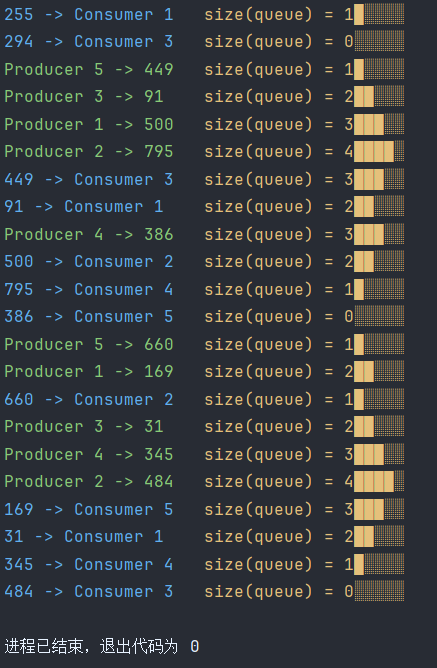
在生产者函数中，生成一个随机数作为数据项，等待 full信号量，然后获取互斥锁并将数据项放入队列中，最后释放互斥锁并增加 empty 信号量。

在消费者函数中，等待 empty信号量，然后获取互斥锁从队列中取出数据项，最后释放互斥锁并增加 full 信号量。

主函数中等待所有生产者和消费者线程完成工作后，销毁信号量和互斥锁。

#### ②运行结果：





多次运行, 改变生产者、消费者数量以及队列容量, 均未产生队列的正负溢出

#### ③完整代码:

*#include* <pthread.h>  
*#include* <semaphore.h>  
*#include* <stdlib.h>  
*#include* <unistd.h>  
*#include* "ColorPrint.h"  
  
*#define* Capacity 5  
*#define* ProducerCnt 5  
*#define* ConsumerCnt 5  
  
*int* queue[Capacity], front = 0, rear = -1, itemCount = 0;*// 队列*sem\_t empty, full;*// 信号量*pthread\_mutex\_t mutex;*// 互斥锁  
  
void* printQueueSizeInfo() {  
 *if* (0 <= itemCount && itemCount <= Capacity) {  
 colorPrint(YELLOW, "\tsize(queue) = %d", itemCount);  
 *for* (*int* i = 0; i < itemCount; i++) colorPrint(YELLOW, "█");  
 *for* (*int* i = 0; i < Capacity - itemCount; i++) colorPrint(YELLOW, "░");  
 printf("\n");  
 } *else* {*// 队列正溢出或负溢出* colorPrint(RED, "\tsize(queue) = %d\n", itemCount);  
 }  
}  
  
*/\*\*  
 \* 生产者线程, 生成5个随机数放入共享队列  
 \*/  
void* \*producer(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 srand(id);*//随机数种子  
 for* (*int* i = 0; i < 5; i++) {  
 *int* item = rand() % 1000;  
 sem\_wait(&full); *// 等待队列非满  
  
 // 生产者 - 向队列放数字* pthread\_mutex\_lock(&mutex);  
 rear = (rear + 1) % Capacity;  
 queue[rear] = item;  
 itemCount++;  
 colorPrint(GREEN, "Producer %d -> %d", id, item); *// 打印信息* printQueueSizeInfo();  
 pthread\_mutex\_unlock(&mutex);  
  
 sem\_post(&empty);*// 放入了一个数据* usleep(100000);  
 }  
 *return* NULL;  
}  
  
*/\*\*  
 \* 消费者线程, 从共享队列取出5个数据  
 \*/  
void* \*consumer(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 *for* (*int* i = 0; i < 5; i++) {  
 sem\_wait(&empty);*// 等待队列非空  
  
 // 消费者 - 从队列取数字* pthread\_mutex\_lock(&mutex);  
 *int* item = queue[front];  
 front = (front + 1) % Capacity;  
 itemCount--;  
 *// 打印信息* colorPrint(BLUE, "%d -> Consumer %d", item, id);  
 printQueueSizeInfo();  
  
 pthread\_mutex\_unlock(&mutex);  
  
 sem\_post(&full);*// 取走了一个数据* usleep(100000);  
 }  
 *return* NULL;  
}  
  
*int* main() {  
 sem\_init(&full, 0, Capacity); *// 初始值Capacity, 队列满时生产者不能放数据到队列* sem\_init(&empty, 0, 0);*// 初始值0, 消费者不能从空队列取数据* pthread\_mutex\_init(&mutex, NULL); *// 互斥锁, 防止同时修改队列, 导致数据不一致* colorPrint(PURPLE, "ProducerCnt=%d, ConsumerCnt=%d, Capacity=%d\n", ProducerCnt, ConsumerCnt, Capacity);  
 *// 开启线程* pthread\_t producers[ProducerCnt], consumers[ConsumerCnt];  
 *int* pIds[ProducerCnt], cIds[ConsumerCnt];  
  
 *for* (*int* i = 0; i < ProducerCnt; i++) {  
 pIds[i] = i + 1;  
 pthread\_create(&producers[i], NULL, producer, &pIds[i]);  
 }  
  
 *for* (*int* i = 0; i < ConsumerCnt; i++) {  
 cIds[i] = i + 1;  
 pthread\_create(&consumers[i], NULL, consumer, &cIds[i]);  
 }  
  
 *for* (*int* i = 0; i < ProducerCnt; i++) pthread\_join(producers[i], NULL);  
 *for* (*int* i = 0; i < ConsumerCnt; i++) pthread\_join(consumers[i], NULL);  
  
 sem\_destroy(&empty);  
 sem\_destroy(&full);  
 pthread\_mutex\_destroy(&mutex);  
  
 *return* 0;  
}

## (2) 读者-写者问题

### 1、问题说明:

有多个读者和多个写者共享一个数据

写者: 修改数据, 同一时间只能有一个写者写数据

读者: 读取数据, 多个读者可以在同一时间读取数据

读者和写者互斥，写者和写者互斥，读者和读者不互斥。

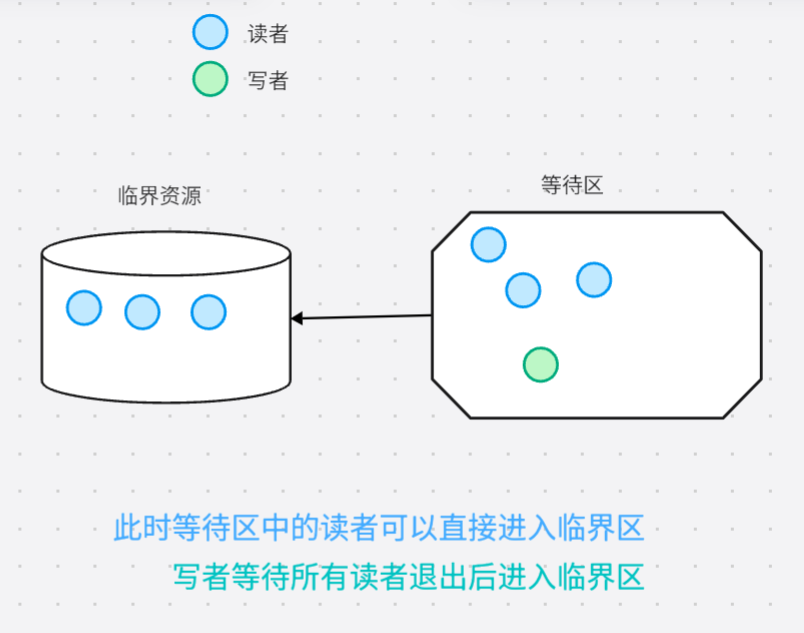
### 2、算法1:

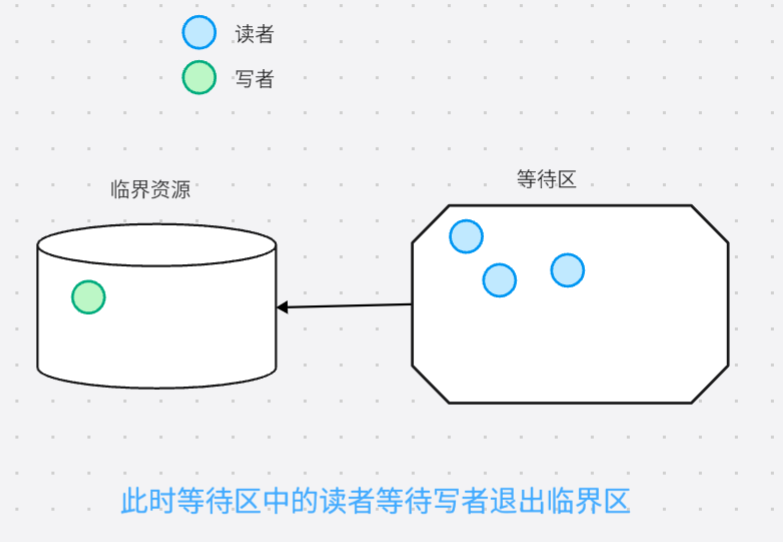
#### 设计:

写者: 如果当前有读者, 则等待所有读者退出后进行写

读者: 如果当前有写者, 则等待当前写者退出后进行读, 多个读者可以同时进行读(使用int记录个数), 当最后一个读者结束时通知写者进行写

#### 图示说明:

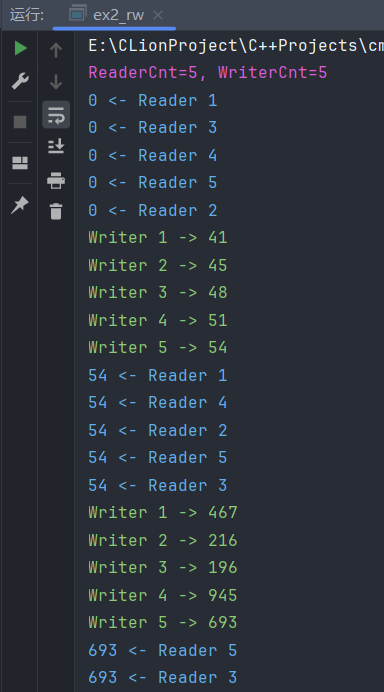




#### 代码(主要部分):

*void* \*writer(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 srand(id);*//随机数种子  
 for* (*int* i = 0; i < 5; i++) {  
 sem\_wait(&wMutex); *// 等待所有读者完成  
 // 写入数据* sharedData = rand() % 1000;  
 colorPrint(GREEN, "Writer %d -> %d\n", id, sharedData);  
 usleep(100000);  
  
 sem\_post(&wMutex);  
  
 usleep(100000);  
 }  
 *return* NULL;  
}  
  
*void* \*reader(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 *for* (*int* i = 0; i < 5; i++) {  
 sem\_wait(&rMutex);  
 *if* (readerCount == 0) sem\_wait(&wMutex);*// 如果是第一个读者, 等待当前写者完成* readerCount++;  
 sem\_post(&rMutex);  
  
 *// 读取数据* colorPrint(BLUE, "%d <- Reader %d\n", sharedData, id);  
 usleep(100000);  
  
 sem\_wait(&rMutex);  
 readerCount--;  
 *if* (readerCount == 0) sem\_post(&wMutex); *// 如果是最后一个读者, 通知写者* sem\_post(&rMutex);  
  
 usleep(100000);  
 }  
 *return* NULL;  
}

#### 运行结果:



这种算法弊端非常明显, 写者只有等待所有读者全部完成后才能开始陆续进行写, 在读者数量非常大时, 对写者很不公平

### 3、算法2:

#### 设计:

为了解决算法1中对写者不公平的情况, 我在算法1的基础上增加了一个信号量mutex

mutex的作用类似于一个等待队列, 所有的线程先进入等待队列中

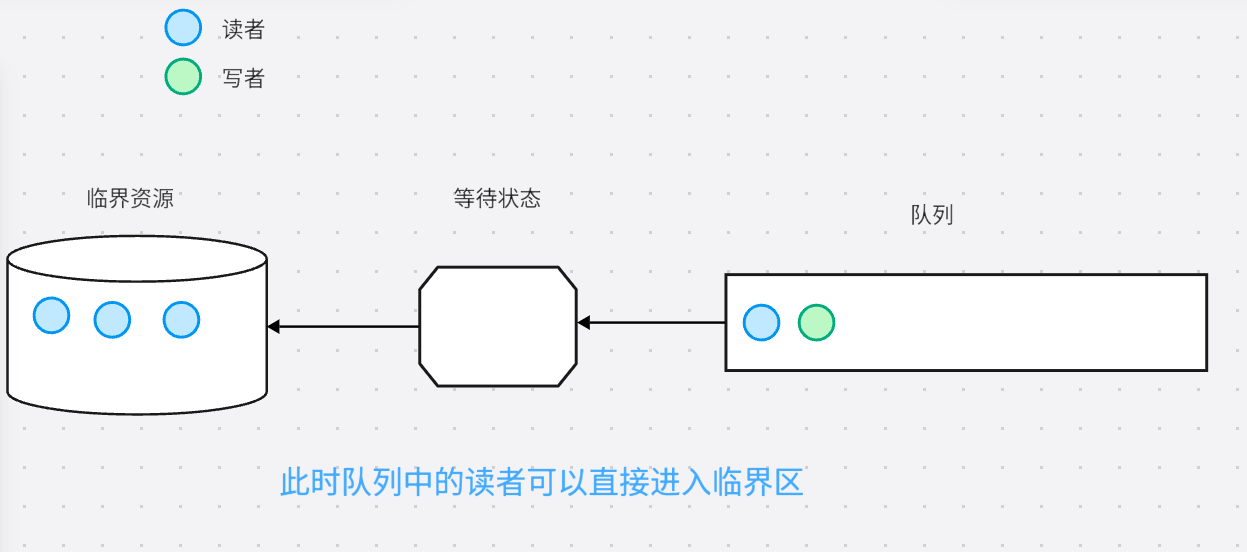
对于写者: 如果队列前方有读线程, 则等待; 如果队列前方没有读线程, 则进入”等待写”状态, 等到正在读的所有读者退出后进行写

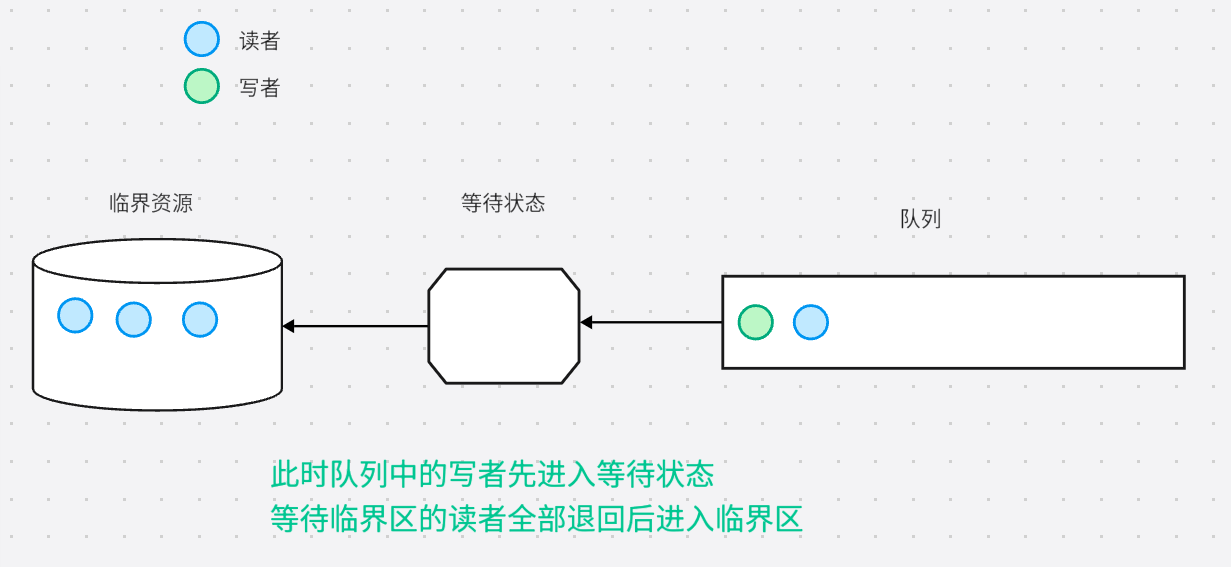
对于读者: 如果队列前方有写线程, 则等待; 如果队列前方没有写线程, 则进入”等待读”状态, 等到正在写的写者退出后进行读; 当所有读者退出后通知”等待写”的写者

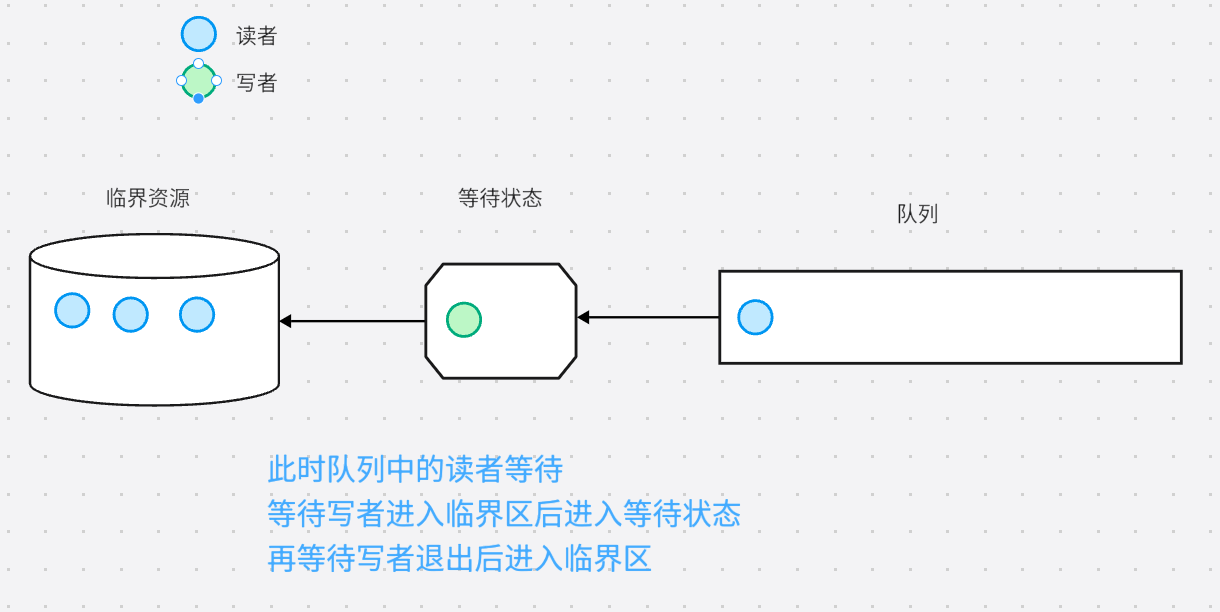
#### 图示说明:

队列->等待状态: 对应mutex信号量

等待状态->临界区: 对应wMutex信号量



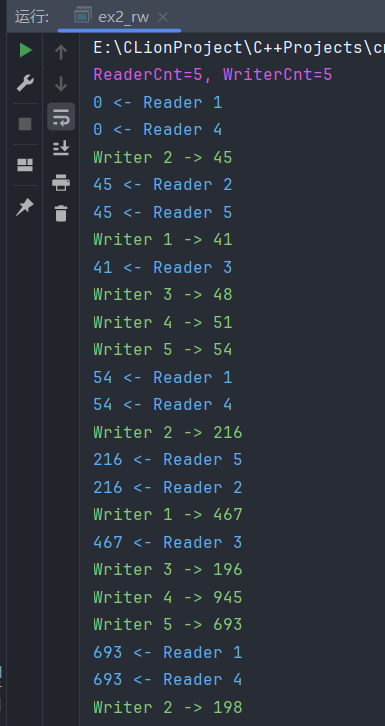




#### 代码实现:

*void* \*writer(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 srand(id);*//随机数种子  
 for* (*int* i = 0; i < 5; i++) {  
 sem\_wait(&mutex); *// 等待队列*  
 sem\_wait(&wMutex); *// 等待所有读者完成  
 // 写入数据* sharedData = rand() % 1000;  
 colorPrint(GREEN, "Writer %d -> %d\n", id, sharedData);  
 usleep(100000);  
  
 sem\_post(&wMutex);  
 sem\_post(&mutex);  
 usleep(100000);  
 }  
 *return* NULL;  
}  
  
*void* \*reader(*void* \*arg) {  
 *int* id = \*((*int* \*) arg);  
 *for* (*int* i = 0; i < 5; i++) {  
 sem\_wait(&mutex); *// 等待队列*  
 sem\_wait(&rMutex);  
 *if* (readerCount == 0) sem\_wait(&wMutex);*// 如果是第一个读者, 等待当前写者完成* readerCount++;  
 sem\_post(&rMutex);  
 sem\_post(&mutex);  
  
 *// 读取数据* colorPrint(BLUE, "%d <- Reader %d\n", sharedData, id);  
 usleep(100000);  
  
 sem\_wait(&rMutex);  
 readerCount--;  
 *if* (readerCount == 0) sem\_post(&wMutex); *// 如果是最后一个读者, 通知写者* sem\_post(&rMutex);  
  
 usleep(100000);  
 }  
 *return* NULL;  
}

#### 运行结果:

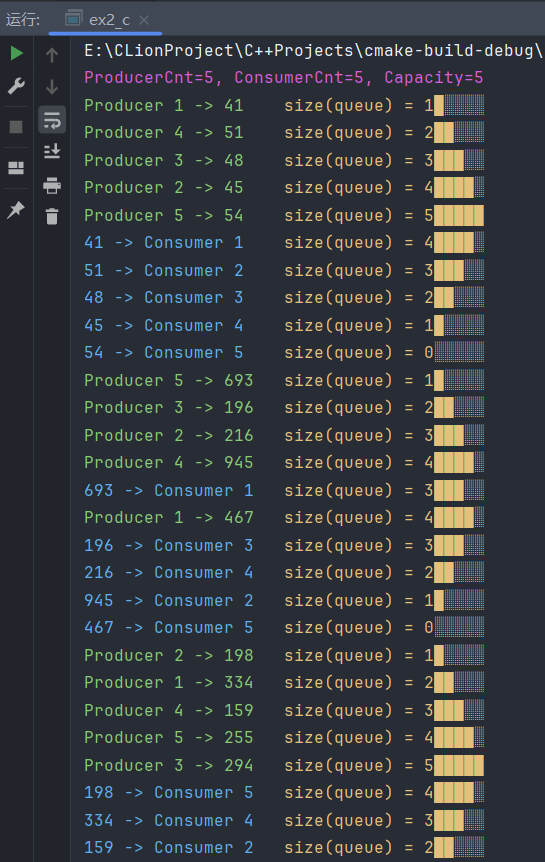


可以看到, 读写的频率比较均匀, 我认为这种算法更合理一些

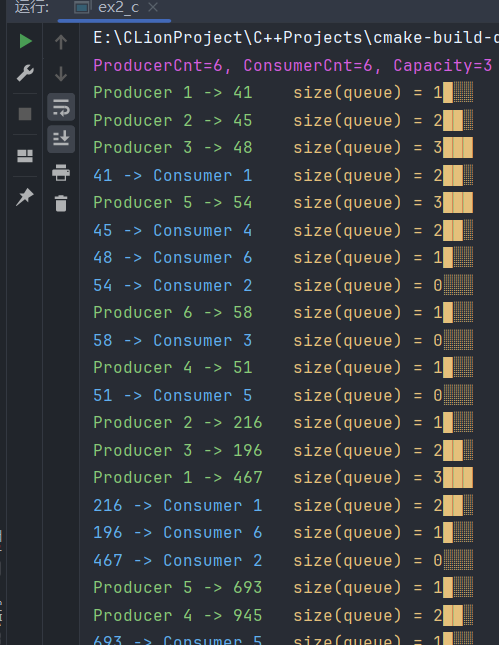
# 五、实验数据及处理结果

## 生产者-消费者问题运行结果:

下图为其中一组数据, 程序未产生死锁, 生产者和消费者成功地协同工作，没有发生数据丢失或重复的情况

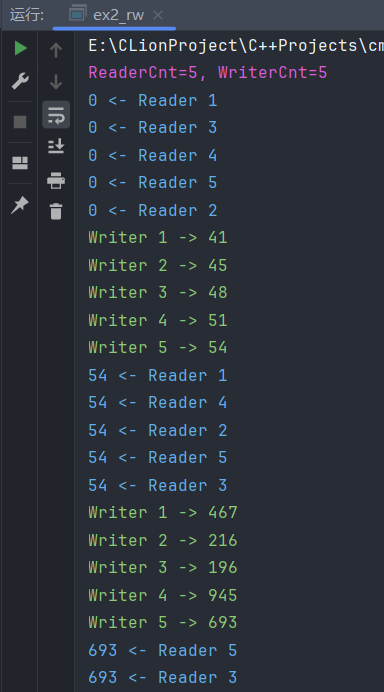


后面又进行了多次测试, 更改进程数量、资源数量, 均未出现死锁情况(如图)



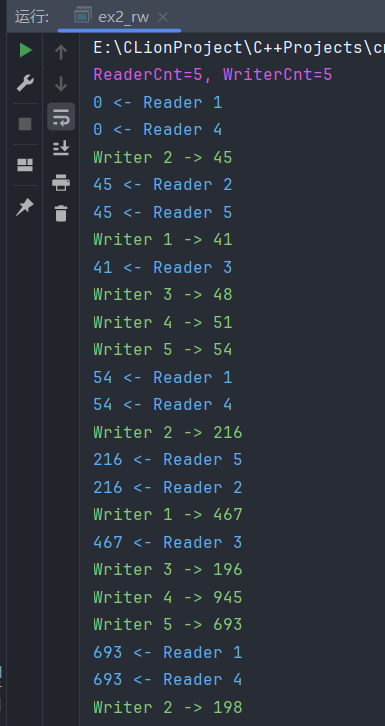
## 读者写者问题运行结果:

算法1结果如图:



多次运行均未出现死锁情况, 但此算法对写者不公平, 因此改进为算法2

算法2结果如图:



未出现死锁情况, 且读者写者获取临界资源公平

# 六、实验体会或对改进实验的建议

通过本次实验，我深刻理解了进程同步和互斥在操作系统中的重要性。在实际编程中，如何有效地管理多个进程对共享资源的访问是一个关键问题。通过模拟消费者-生产者问题和读者-写者问题，我学会了以下几方面的知识：

1. **进程间通信**：在消费者-生产者和读者-写者模型中，信号量是实现进程间通信的一种有效手段。通过使用信号量，可以确保多个进程之间能够正确地协调工作，避免竞争条件。
2. **资源管理**：在多进程环境下，如何正确管理和分配资源是一个复杂的问题。通过本次实验，我学会了如何使用信号量来控制对共享资源的访问，从而保证系统的稳定性和效率。
3. **死锁预防**：在设计多进程系统时，死锁是一个需要特别注意的问题。通过合理地安排信号量的获取和释放顺序，可以避免死锁的发生，提高系统的可靠性。
4. **并发控制**：并发控制是多进程系统中的核心问题之一。在本次实验中，我学会了如何使用信号量来实现进程的同步和互斥，从而有效地解决并发控制的问题。
5. **问题建模**：通过模拟实际问题的模型，如消费者-生产者和读者-写者问题，我学会了如何将复杂的现实问题抽象成具体的计算模型，并通过程序加以解决。
6. **代码调试**：在编写和调试多进程程序时，我遇到了许多挑战，但也学到了许多调试技巧。例如，如何使用工具和日志来跟踪进程的状态和行为，以及如何定位和解决死锁等问题。

总的来说，本次实验不仅让我掌握了进程同步与互斥的基本概念和方法，还提高了我的编程能力和解决问题的能力。通过实践，我更加深刻地认识到理论知识的重要性，并体会到了理论与实践相结合的乐趣和成就感。

# 七、参考资料

《计算机操作系统实验指导》