**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题目: 基于条件变量的同步互斥**

**学院 计算机学院（国家示范性软件学院）**

**班级 2021211301**

**学号 2021213586**

**姓名 郭栩源**

**2024年 4月**

## 目 录

第一章 实验概述 1

1.1 实验内容 1

1.1.1探索基于条件变量（管程）的同步互斥方式 1

1.1.2使用信号量方式和条件变量（管程方式）解决经典同步互斥问题 1

第二章 实验步骤 2

2.1 生产者-消费者问题的信号量解法 2

2.1.1实验代码 2

2.1.2实验结果 4

2.2 生产者-消费者问题的条件变量（管程）解法 4

2.2.1实验代码 4

2.2.2实验结果 8

2.3 读-写者问题的信号量解决方法 8

2.3.1实验代码 8

2.3.1实验结果 10

2.4 读-写者问题的条件变量（管程）解决方法 10

2.4.1实验代码 10

2.4.2实验结果 12

第三章 总结 12

# **第一章 实验概述**

##### 1.1 实验内容

# **1.1.1 探索基于条件变量（管程）的同步互斥方式**

**管程**是一种同步构造，用于实现多线程程序中的线程间通信与协调。它封装了共享数据的访问，并且只允许一个线程在任何时刻进入管程，从而提供了一种简单有效的方法来避免竞态条件。管程内部通常包含：

1. 一组私有变量：由管程管理的数据，仅在管程内部访问。
2. 一组过程：也就是可以操作这些私有变量的方法或函数。
3. 一个互斥锁（Mutex）：确保任何时刻只有一个线程可以执行任何一个过程。
4. 条件变量：用于线程之间的同步，比如当等待某些条件成立或通知条件已经成立时。

**条件变量**是一种在多线程编程中用于阻塞和唤醒线程的同步机制。它需要与互斥锁（mutex）一起使用，使得线程可以在某些条件尚未满足时阻塞，并在条件满足时被唤醒。条件变量的主要操作包括：

1. 等待（Wait）：线程调用等待操作时，它必须已经获得了互斥锁。调用等待操作会释放互斥锁，并使线程阻塞，直到其他线程在相同条件变量上调用通知操作。
2. 通知单个线程（Signal）：唤醒在该条件变量上等待的一个线程。如果没有线程在等待，此操作无效果。
3. 通知所有线程（Broadcast）：唤醒在该条件变量上等待的所有线程。

**管程**使用**条件变量**来管理线程间的等待和通知机制。当在管程内的操作不能继续执行因为某个条件尚未成立时，线程可以在一个条件变量上等待。当其他线程改变了条件并使其成立时，它可以通知一个或多个在该条件变量上等待的线程。

# **1.1.2 使用信号量方式和条件变量（管程方式）解决经典同步互斥问题**

* **生产者消费者问题**

生产者消费者问题涉及两类进程或线程：生产者和消费者。生产者负责生成数据，消费者负责使用这些数据。二者之间需要一个共享的缓冲区或队列来存储生产的数据。关键的同步需求是：

1. **缓冲区的容量有限**：生产者在缓冲区满时必须等待，直到消费者消费了数据，释放了空间。
2. **空缓冲区**：消费者在缓冲区空时必须等待，直到生产者生产了新的数据。

* **读写者问题**

读写者问题涉及一组读者和写者，它们需要共享访问某种数据资源。关键点在于：

1. 多个读者同时：允许多个读者同时访问数据，因为读操作不会改变数据。
2. 单个写者独占：当写者访问数据时，必须独占访问权，不允许其他读者或写者同时访问。
3. 公平性问题：必须合理安排读者和写者的访问权限，避免写者饥饿（长时间得不到访问机会）或读者饥饿。

# **第二章 实验步骤**

##### 2.1 生产者-消费者问题的信号量解法

# **2.1.1 实验代码**

本次实验使用POSIX线程库提供的信号量功能来完成信号量相关代码的实现。

POSIX（Portable Operating System Interface for Unix）是一个旨在促进应用程序与各种UNIX系统之间兼容性的标准集合。它由 IEEE（电气和电子工程师协会）定义为一系列标准，这些标准指定了编程接口、命令行和实用程序的一致接口，以确保不同的UNIX系统之间的软件兼容性。

POSIX库中使用sem\_t类型来创建一个信号量，使用sem\_wait和sem\_post来实现信号量的wait和signal操作。

注意本实验代码仅在可在**POSIX 兼容系统**中使用。

以下是生产者-消费者问题的信号量解法的详细实验代码：

#include<iostream>

#include<thread>

#include<sys/types.h>

#include<unistd.h>

#include<semaphore.h>

#define NUM 30

//生产者，消费者线程个数

const int PRODUCER\_THREAD\_NUM = 3;

const int CONSUMER\_THREAD\_NUM = 3;

// 使用数组模拟缓冲区

int cnt,head,tail,queue[NUM];

// 生产者、消费者信号量

sem\_t psem, csem;

void producer\_thread(int thread\_id) {

while (true) {

usleep(500000);

//对缓冲区队列加锁

sem\_wait(&psem);

//添加数据

queue[tail]=++cnt;

tail=(tail+1)%NUM;

std::cout << "producer\_thread: " << thread\_id << " producer data: " << cnt;

std::cout << " queue size: " << (tail-head+NUM) % NUM << std::endl;

//释放锁

sem\_post(&csem);

}

}

void consumer\_thread(int thread\_id){

while (true) {

usleep(550000);

//对缓冲区队列加锁

sem\_wait(&csem);

//取数据

int data=queue[head];

head=(head+1)%NUM;

std::cout << "\tconsumer\_thread: " << thread\_id << " consumer data: ";

std::cout << data << " deque size: " << (tail-head+NUM) % NUM << std::endl;

//释放锁

sem\_post(&psem);

}

}

int main() {

sem\_init(&psem, 0, NUM);

sem\_init(&csem, 0, 0);

std::thread arrProducerThread[PRODUCER\_THREAD\_NUM];

std::thread arrConsumerThread[CONSUMER\_THREAD\_NUM];

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

arrProducerThread[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++) {

arrProducerThread[i].join();

}

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

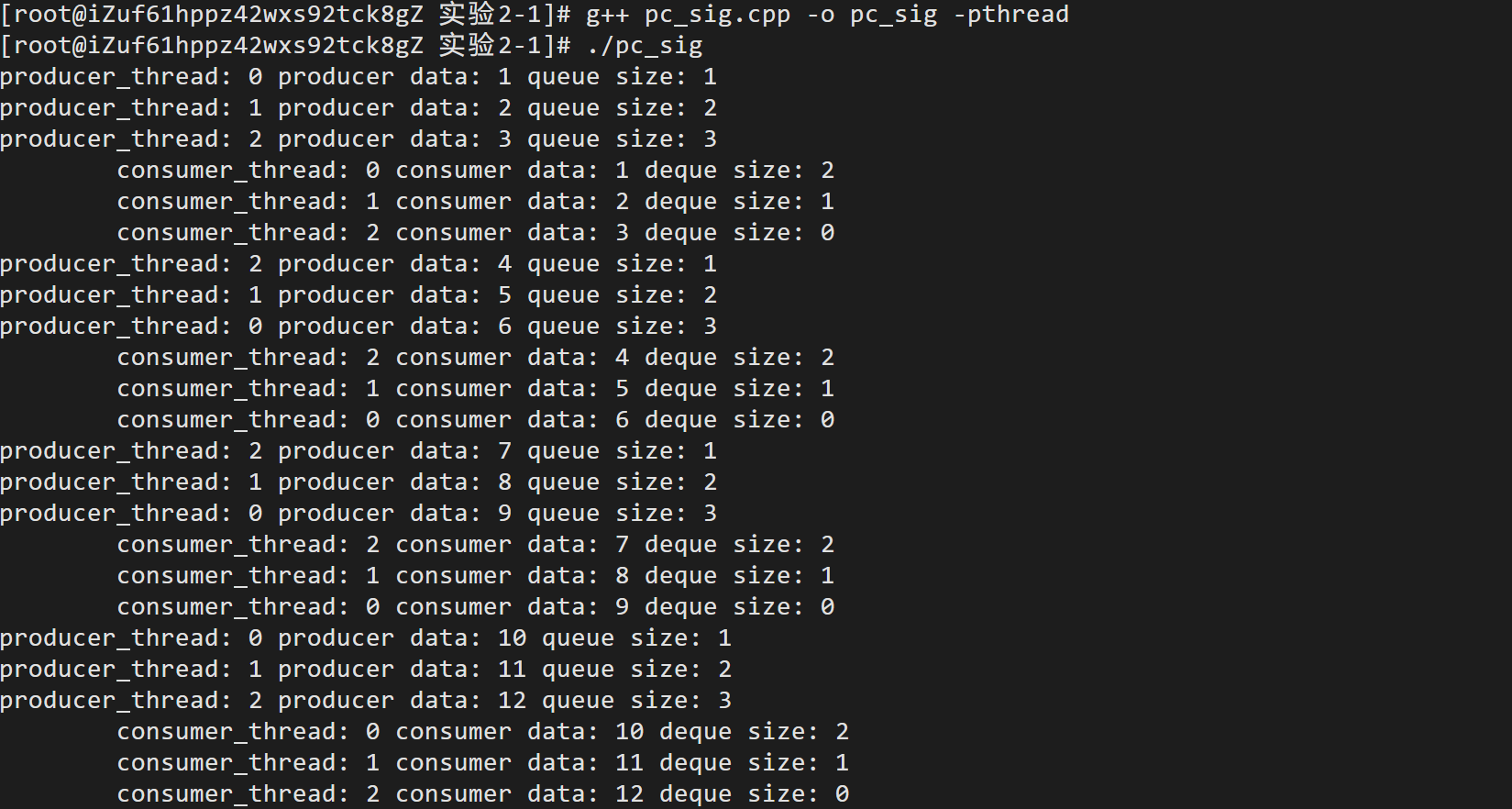
arrConsumerThread[i].join();

}

return 0;

}

# **2.1.2 实验结果**



可以看到，该程序成功实现了生产者的0、1、2号线程和消费者的0、1、2号线程互斥访问缓冲区；同时，对缓冲区的读写操作也没有问题，即，消费者从缓冲区中读取的信息都是正确的。

##### 2.2 生产者-消费者问题的条件变量（管程）解法

# **2.2.1 实验代码**

本实验使用C++11标准中提供的mutex和conditional\_variable来完成条件变量（管程）的代码实现

使用std::mutex类型来创建一个互斥量，使用std::condition\_variable来创建一个条件变量。

* std::mutex类型的成员函数：
* lock()

调用lock()的线程将锁住互斥锁，线程调用该函数会发生3 种情况：

（1）如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用unlock去释放锁。

（2）如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程被阻塞。

（3）如果当前互斥量被当前调用线程锁住，将产生死锁。

* unlock()

解锁，释放对互斥量的所有权，即释放对共享资源的控制权。

* try\_lock()

尝试锁住互斥量，如果互斥量已经被其它线程占有，当前线程**不会**被阻塞。线程调用该函数会发生3 种情况：

（1）如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用unlock去释放锁。

（2）如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程返回false，并不会被阻塞。

（3）如果当前互斥量被当前调用线程锁住，将产生死锁。

* lock\_guard和unique\_lock：

std::unique\_lock 和 std::lock\_guard 都是用于管理互斥锁（mutex）的 RAII（资源获取即初始化）工具，但它们提供了不同的功能和灵活性：

* + std::lock\_guard

1. 简单且轻量级：std::lock\_guard 提供了一种简单的方式来保护作用域内的代码，确保在作用域开始时自动获取互斥锁，并在作用域结束时自动释放互斥锁。它是不可复制和不可移动的。
2. 自动锁管理：它只负责基本的锁定和解锁操作，不提供对锁的额外控制，如条件性锁定、延迟锁定或尝试锁定等功能。
   * std::unique\_lock
3. 更灵活：相比 std::lock\_guard，std::unique\_lock 提供了更多的灵活性。它允许你延迟锁定互斥锁、尝试锁定互斥锁、提前解锁互斥锁，以及传递所有权等。
4. 可移动但不可复制：std::unique\_lock 对象是可移动的，这意味着你可以将锁的所有权从一个函数传递到另一个函数，而 std::lock\_guard 则不允许这样做。
5. 条件变量兼容：std::unique\_lock 被设计用于与条件变量一起使用。在使用 std::condition\_variable 的 wait 函数时，std::unique\_lock 能够在等待时自动释放锁，并在条件变量触发后重新获取锁。

* std::condition\_variable类型的成员函数

1. 等待函数wait：

(1) wait(unique\_lock <mutex>＆lck)

当前线程的执行会被阻塞，直到收到notify为止。

(2) wait(unique\_lock <mutex>＆lck，Predicate pred)

当前线程仅在pred=false时阻塞；如果pred=true时，不阻塞。wait()执行包括三个步骤：释放互斥锁、等待在条件变量上、再次获取互斥锁。

2. 通知函数：

（1）notify\_one()：没有参数，也没有返回值。

解除阻塞当前正在等待此条件的线程之一。如果没有线程在等待，则函数不执行任何操作。如果超过一个，不会指定具体哪一线程。

（2）notify\_all():

这一点与 **notify\_one** 函数不同，notify\_all()函数同时唤醒所有因等待特定条件变量而阻塞的线程。若有多个进程被同时唤醒，则线程若要继续执行，还需重新获得之前释放的互斥锁。如果这个锁在那一刻仍然被其他线程持有，被唤醒的线程会自动进入锁的等待队列中。

以下是生产者-消费者问题的条件变量（管程）解法的详细实验代码：

#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];

std::thread arrConsumerThread[CONSUMER\_THREAD\_NUM];

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

arrProducerThread[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++) {

arrProducerThread[i].join();

}

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

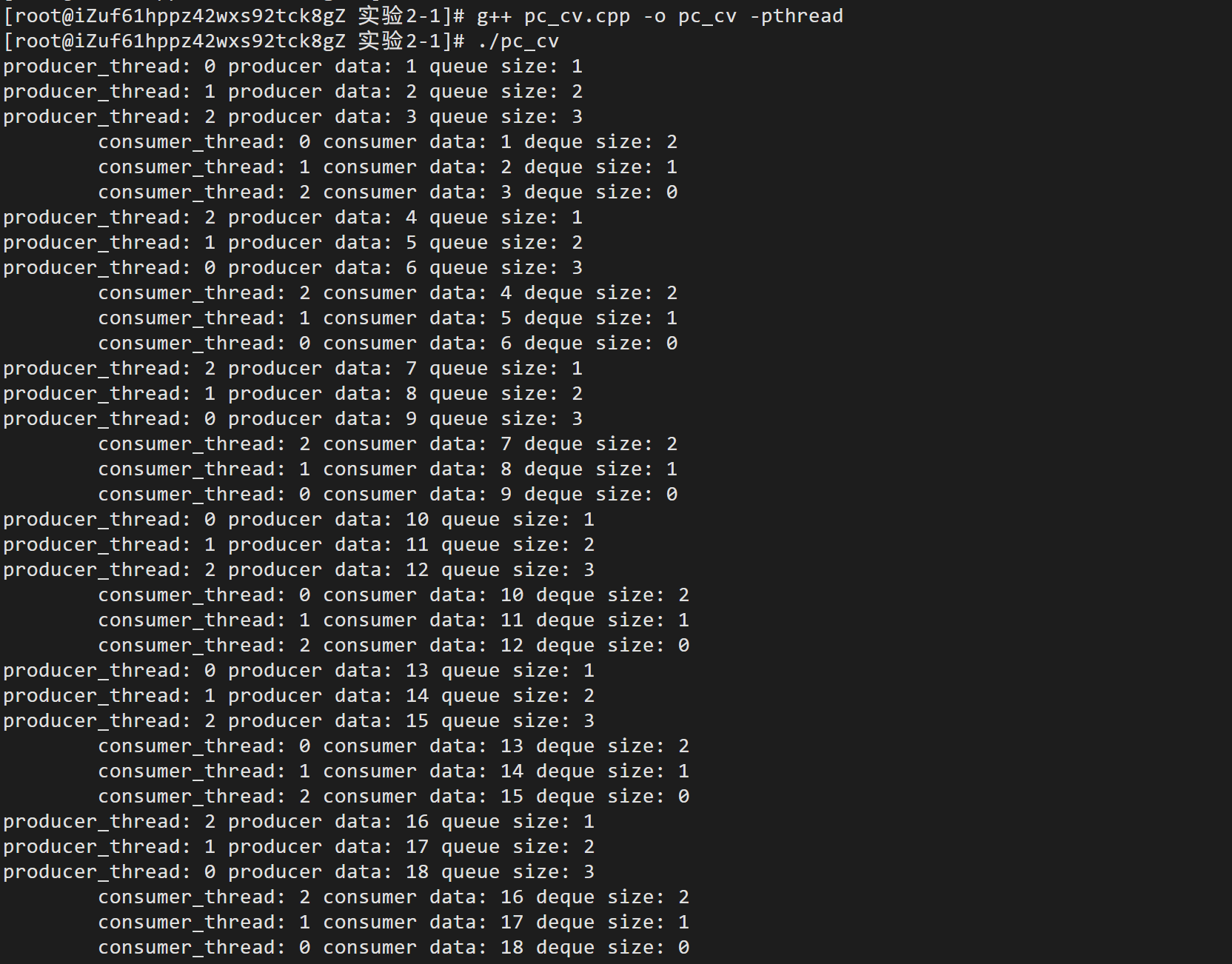
arrConsumerThread[i].join();

}

return 0;

}

# **2.2.2 实验结果**



可以看到，使用管程解决生产者-消费者问题的结果与使用信号量的方法相似，两者都能实现生产者与消费者对于有界缓冲区的互斥访问，并且对缓冲区的读写操作也都是正确的。

##### 2.3 读-写者问题的信号量解决方法

# **2.3.1 实验代码**

#include<iostream>

#include<thread>

#include<sys/types.h>

#include<unistd.h>

#include<semaphore.h>

#define NUM 30

//生产者，消费者线程个数

const int WRITER\_THREAD\_NUM = 3;

const int READER\_THREAD\_NUM = 3;

int rc;

sem\_t rc\_mtx, mtx;

void WRITER\_thread(int id) {

while (true) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

sem\_wait(&mtx);

std::cout << "Writer " << id << " is writing\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(500)); // Simulate writing

std::cout << "Writer " << id << " has finished writing\n";

sem\_post(&mtx);

}

}

void READER\_thread(int id){

while (true) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

sem\_wait(&rc\_mtx);

rc++;

if(rc==1){

sem\_wait(&mtx);

}

sem\_post(&rc\_mtx);

std::cout << "Reader " << id << " is reading\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(500)); // Simulate reading

std::cout << "Reader " << id << " has finished reading\n";

sem\_wait(&rc\_mtx);

rc--;

if(rc==0){

sem\_post(&mtx);

}

sem\_post(&rc\_mtx);

}

}

int main() {

sem\_init(&mtx, 0, 1);

sem\_init(&rc\_mtx, 0, 1);

std::thread arrWRITERThread[WRITER\_THREAD\_NUM];

std::thread arrREADERThread[READER\_THREAD\_NUM];

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

arrWRITERThread[i] = std::thread(WRITER\_thread, i);

}

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

arrREADERThread[i] = std::thread(READER\_thread, i);

}

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

arrWRITERThread[i].join();

}

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

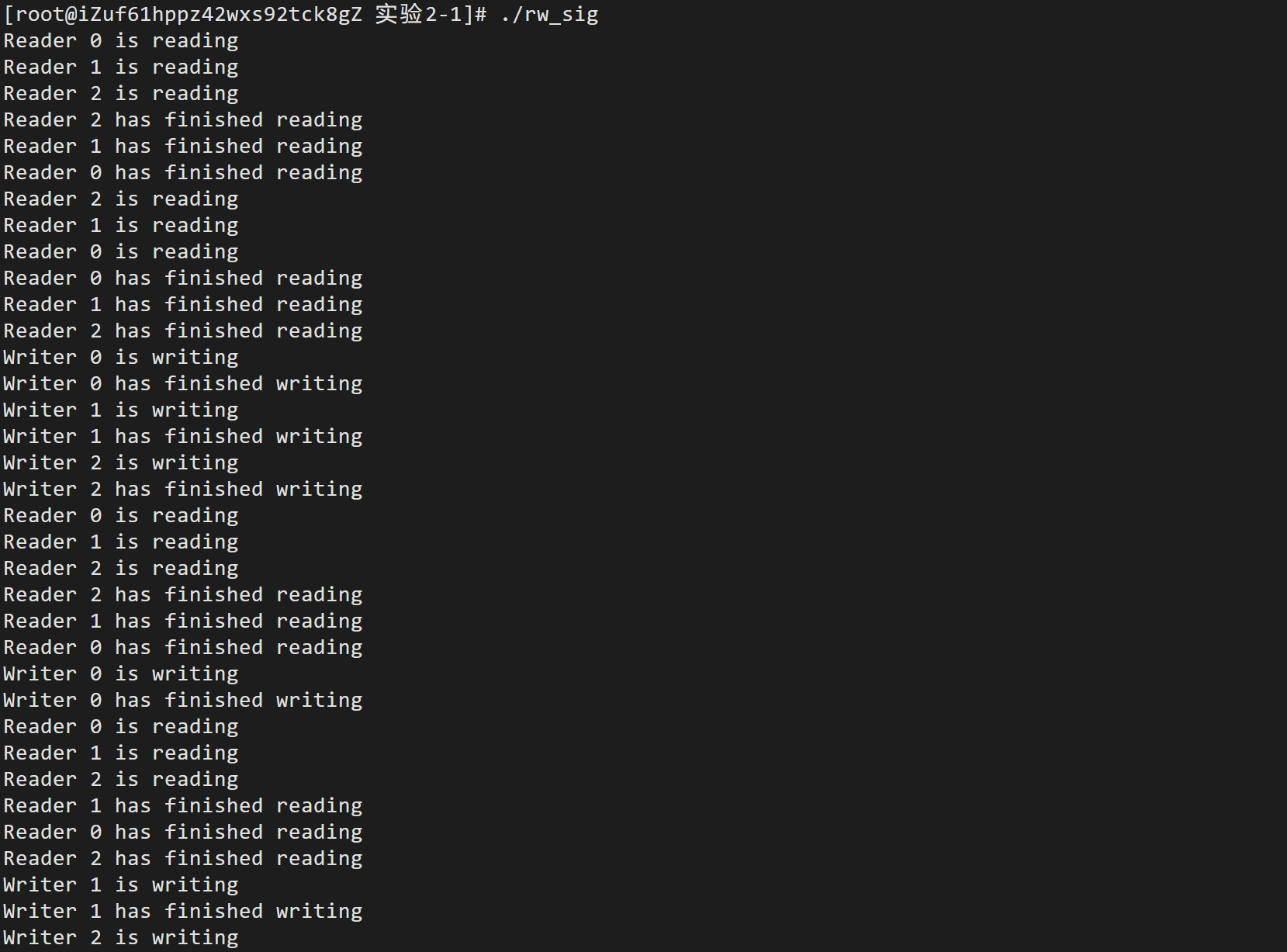
arrREADERThread[i].join();

}

return 0;

}

# **2.3.2 实验结果**



可以看到，读-写者问题的信号量解决方法实现了读-写者问题读者并行访问资源、读-写互斥和写-写互斥的特点。

##### 2.4 读-写者问题的条件变量（管程）解决方法

# **2.4.1 实验代码**

在读写者问题的管程解决方法之中，为了发挥管程对问题进行进一步抽象的特点，我编写ReadWriteLock，提出了软件层面抽象的读写锁，在处理读-写者问题的时候，读者和写者只需要申请ReadWriteLock的读写锁即可。

#include <iostream>

#include <mutex>

#include <condition\_variable>

#include <thread>

#include <vector>

//生产者，消费者线程个数

const int READER\_THREAD\_NUM = 3;

const int WRITER\_THREAD\_NUM = 3;

class ReadWriteLock {

std::mutex mtx;

std::condition\_variable cv;

int active\_readers = 0;

int waiting\_writers = 0;

bool active\_writer = false;

public:

void readLock() {

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, [this]() {

return !active\_writer && waiting\_writers == 0;

});

active\_readers++;

}

void readUnlock() {

std::lock\_guard<std::mutex> guard(mtx);

active\_readers--;

if (active\_readers == 0) cv.notify\_all();

}

void writeLock() {

std::unique\_lock<std::mutex> lock(mtx);

waiting\_writers++;

cv.wait(lock, [this]() {

return !active\_writer && active\_readers == 0;

});

waiting\_writers--;

active\_writer = true;

}

void writeUnlock() {

std::lock\_guard<std::mutex> guard(mtx);

active\_writer = false;

cv.notify\_all();

}

};

ReadWriteLock rwLock;

void reader(int id) {

while(1) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

rwLock.readLock();

std::cout << "Reader " << id << " is reading\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(500)); // Simulate reading

std::cout << "Reader " << id << " has finished reading\n";

rwLock.readUnlock();

}

}

void writer(int id) {

while(1) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

rwLock.writeLock();

std::cout << "Writer " << id << " is writing\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(500)); // Simulate writing

std::cout << "Writer " << id << " has finished writing\n";

rwLock.writeUnlock();

}

}

int main() {

std::thread arrREADERThread[READER\_THREAD\_NUM];

std::thread arrWRITERThread[WRITER\_THREAD\_NUM];

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

arrREADERThread[i] = std::thread(reader, i);

}

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

arrWRITERThread[i] = std::thread(writer, i);

}

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

arrREADERThread[i].join();

}

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

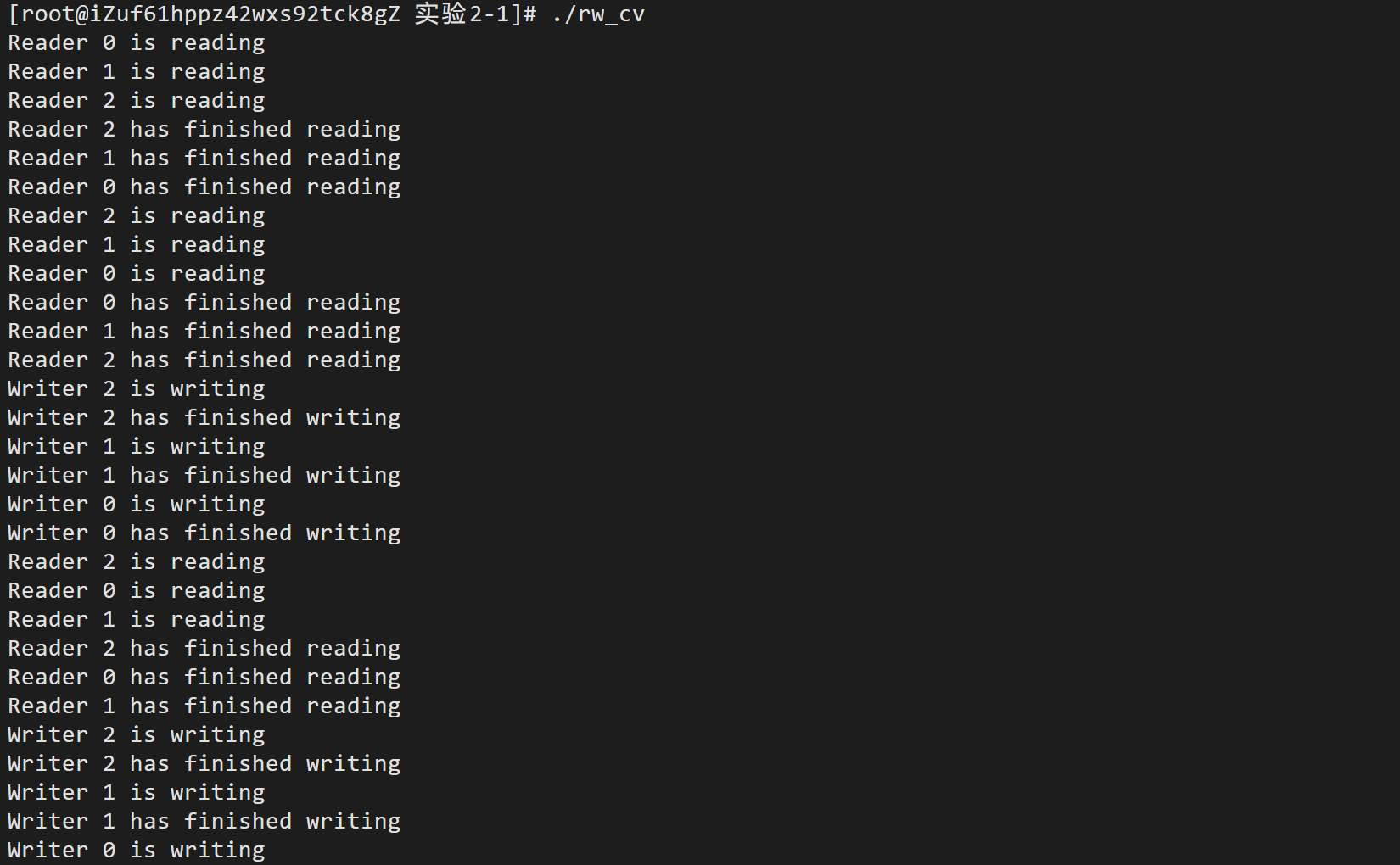
arrWRITERThread[i].join();

}

return 0;

}

# **2.3.2 实验结果**



可以看到，基于条件变量（管程）的读-写者问题解决方法实现了读-写者问题读者并行访问资源、读-写互斥和写-写互斥的特点，其运行结果与基于信号量的解决方法相似。

# **第三章 实验总结**

在本实验中，我分别使用信号量和管程解决了生产者-消费者问题和读-写者问题。

信号量和管程是用于解决多线程并发访问共享资源的两种不同的同步机制。总的来说，管程提供了一种更高级别、更抽象的同步机制，隐藏了同步细节，使得编写并发程序更容易。与此相比，信号量提供了更底层的原语，允许程序员对并发访问共享资源进行更细粒度的控制。

本次实验让我对操作系统同步与互斥有了更深的认识，同时，编写程序实现操作系统同步与互斥的经典模型也增加了我对POSIX库和Linux编程有了更多的了解。