Windows操作系统

C/C++ 程序实验

首都师范大学 信息工程学院

系统分析与管理实验室

## 实验七 Windows 读者写者问题

### 一、背景知识

  在Windows环境下，创建一个控制台进程，此进程包含n个线程。用这n个线程来表示n个读者或写者。每个线程按相应测试数据文件（后面有介绍）的要求进行读写操作。用信号量机制分别实现读者优先和写者优先的读者—写者问题。

读者—写者问题的读写操作限制（包括读者优先和写者优先）：

1)写—写互斥，即不能有两个写者同时进行写操作。

2)读—写互斥，即不能同时有一个线程在读，而另一个线程在写。

3)读—读允许，即可以有一个或多个读者在读。

读者优先的附加限制：如果一个读者申请进行读操作时已有另一个读者正在进行读操作，则该读者可直接开始读操作。

写者优先的附加限制：如果一个读者申请进行读操作时已有另一写者在等待访问共享资源，则该读者必须等到没有写者处于等待状态后才能开始读操作。

运行结果显示要求：要求在每个线程创建、发出读写操作申请、开始读写操作和结束读写操作时分别显示一行提示信息，以确定所有处理都遵守相应的读写操作限制。

### 二、实验目的

测试数据文件包括n行测试数据，分别描述创建的n个线程是读者还是写者，以及读写操作的开始时间和持续时间。每行测试数据包括四个字段，各个字段间用空格分隔。第一字段为一个正整数，表示线程序号。第二字段表示相应线程角色，R表示读者，W表示写者。第三字段为一个正数，表示读写操作的开始时间：线程创建后，延迟相应时间（单位为秒）后发出对共享资源的读写申请。第四字段为一个正数，表示读写操作的持续时间。当线程读写申请成功后，开始对共享资源的读写操作，该操作持续相应时间后结束，并释放共享资源。

### 三、工具/准备工作

在开始本实验之前，请回顾教科书的相关内容。

需要做以下准备：

1) 一台运行Windows 操作系统的计算机。

2) 计算机中需安装Microsoft Visual Studio Code。

### 四、实验内容与步骤

可以将所有读者和所有写者分别存于一个读者等待队列和一个写者等待队列中，每当读允许时，就从读者队列中释放一个或多个读者线程进行读操作；每当写允许时，就从写者队列中释放一个写者进行写操作。

#### 1. 读者写者问题

**步骤1**：登录进入Windows 。

**步骤2**：在“开始”菜单中单击“程序”-“Microsoft Visual Studio Code”。

**步骤3：**新建项目名为“7-1”，并且新建项“7-1.cpp”。

**清单7-1 读者写者问题**

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

#include <fstream>

#include <io.h>

#include <string.h>

#include <stdio.h>

using namespace std;

#define READER 'R' //读者

#define WRITER 'W' //写者

#define INTE\_PER\_SEC 1000 //每秒钟中断数目

#define MAX\_THREAD\_NUM 64 //最大线程数目

#define MAX\_FILE\_NUM 32 //最大数据文件数目

#define MAX\_STR\_LEN 32 //字符串长度

int readcount = 0; //读者数目

int writecount = 0; //写者数目

CRITICAL\_SECTION RP\_Write; //临界区

CRITICAL\_SECTION cs\_Write;

CRITICAL\_SECTION cs\_Read;

struct ThreadInfo

{

int serial; //线程序号

char entity; //线程类别

double delay; //线程延迟时间

double persist; //线程读写文件持续时间

};

////////////////////////////////////////////////////////

//读者优先--读者线程

//p：读者线程信息

void RP\_ReaderThread(void \*p)

{

//互斥变量

HANDLE h\_Mutex;

h\_Mutex = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "mutex\_for\_readcount");

DWORD wait\_for\_mutex; //等待互斥变量所有权

DWORD m\_delay; //延迟时间

DWORD m\_persist; //读文件持续时间

int m\_serial; //线程序号

//从参数中获得信息

m\_serial = ((ThreadInfo\*)(p))->serial;

m\_delay = (DWORD)(((ThreadInfo\*)(p))->delay\*INTE\_PER\_SEC);

m\_persist = (DWORD)(((ThreadInfo\*)(p))->persist \* INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Reader thread %d sents the reading require.\n", m\_serial);

//等待互斥信号， 保证对readcount的访问、修改互斥

wait\_for\_mutex = WaitForSingleObject(h\_Mutex, -1);

//读者数量增加

readcount++;

if (readcount == 1)

{

//第一个读者，等待资源

EnterCriticalSection(&RP\_Write);

}

ReleaseMutex(h\_Mutex); //释放互斥信号

//读文件

printf("Reader thread %d, begins to read file. \n", m\_serial);

Sleep(m\_persist);

//退出线程

printf("Reader thread %d finished reading file.\n", m\_serial);

//等待互斥信号， 保证对readcount的访问、修改互斥

wait\_for\_mutex = WaitForSingleObject(h\_Mutex, -1);

//读者数量减少

readcount--;

if (readcount == 0)

{

//如果所有读者读完，唤醒写者

LeaveCriticalSection(&RP\_Write);

}

ReleaseMutex(h\_Mutex); //释放互斥信号

}

////////////////////////////////////////////////////////////////////////////

//读者优先--写者线程

//p：写者线程信息

void RP\_WriterThread(void \*p)

{

DWORD m\_delay; //延迟时间

DWORD m\_persist; //写文件持续时间

int m\_serial; //线程序号

//从参数中获得信号

m\_serial = ((ThreadInfo\*)(p))->serial;

m\_delay = (DWORD)(((ThreadInfo\*)(p))->delay \* INTE\_PER\_SEC);

m\_persist = (DWORD)(((ThreadInfo\*)(p))->persist \* INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Writer thread %d, sents the writing require.\n", m\_serial);

//等待资源

EnterCriticalSection(&RP\_Write);

//写文件

printf("Writer thread %d begins to write to the file.\n", m\_serial);

Sleep(m\_persist);

//退出线程

printf("Writer thread %d finishing writing to the file.\n", m\_serial);

//释放资源

LeaveCriticalSection(&RP\_Write);

}

//////////////////////////////////////////////////////////////////

//读者优先处理函数

//file：文件名

void ReaderPriority(char\* file)

{

DWORD n\_thread = 0; //线程数量

DWORD thread\_ID; //线程ID

DWORD wait\_for\_all; //等待所有线程结束

//互斥对象

HANDLE h\_Mutex;

h\_Mutex = CreateMutex(NULL, FALSE, "mutex\_for\_readcount");

//线程对象的数组

HANDLE h\_Thread[MAX\_THREAD\_NUM];

ThreadInfo thread\_info[MAX\_THREAD\_NUM];

readcount = 0; //初始化readcount

InitializeCriticalSection(&RP\_Write); //初始化临界区

ifstream inFile;

inFile.open(file); //打开文件

printf("Reader Priority:\n\n");

while (inFile)

{

//读入第一个读者、写者的信息

inFile >> thread\_info[n\_thread].serial;

inFile >> thread\_info[n\_thread].entity;

inFile >> thread\_info[n\_thread].delay;

inFile >> thread\_info[n\_thread++].persist;

inFile.get();

}

for (int i = 0; i < (int)(n\_thread); i++)

{

if (thread\_info[i].entity == READER || thread\_info[i].entity == 'r')

{

//创建读者线程

h\_Thread[i] = CreateThread(NULL, 0,

(LPTHREAD\_START\_ROUTINE)(RP\_ReaderThread),

&thread\_info[i], 0, &thread\_ID);

}

else

{

//创建写者信息

h\_Thread[i] = CreateThread(NULL, 0,

(LPTHREAD\_START\_ROUTINE)(RP\_WriterThread),

&thread\_info[i], 0, &thread\_ID);

}

}

//等待所有线程结束

wait\_for\_all = WaitForMultipleObjects(n\_thread, h\_Thread, TRUE, -1);

printf("All reader and writer have finished, operation.\n");

}

/////////////////////////////////////////////////////////////////////////////////

//写者优先——读者线程

//p：读者线程信息

void WP\_ReaderThread(void \*p)

{

//互斥变量

HANDLE h\_Mutex1;

h\_Mutex1 = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "mutex1");

HANDLE h\_Mutex2;

h\_Mutex2 = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "mutex2");

DWORD wait\_for\_mutex1; //等待互斥变量所有权

DWORD wait\_for\_mutex2;

DWORD m\_delay; //延迟时间

DWORD m\_persist; //读文件持续时间

int m\_serial; //线程序号

//从参数中获得信息

m\_serial = ((ThreadInfo\*)(p))->serial;

m\_delay = (DWORD)(((ThreadInfo\*)(p))->delay \* INTE\_PER\_SEC);

m\_persist = (DWORD)(((ThreadInfo\*)(p))->persist \* INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Reader thread %d sents the reading require.\n", m\_serial);

wait\_for\_mutex1 = WaitForSingleObject(h\_Mutex1, -1);

//进入读者临界区

EnterCriticalSection(&cs\_Read);

//阻塞互斥对象mutex2,保证对readcount的访问、修改互斥

wait\_for\_mutex2 = WaitForSingleObject(h\_Mutex2, -1);

//修改读者数目

readcount++;

if (readcount == 1)

{

//如果是第一个读者，等待写者写完

EnterCriticalSection(&cs\_Write);

}

ReleaseMutex(h\_Mutex2); //释放互斥信号mutex2

//让其他读者进入临界区

LeaveCriticalSection(&cs\_Read);

ReleaseMutex(h\_Mutex1);

//读文件

printf("Reader thread %d begine to read file .\n", m\_serial);

Sleep(m\_persist);

//退出线程

printf("Reader thread %d finished reading file.|n", m\_serial);

//阻塞互斥对象mutex2，对readcount的访问、修改互斥

wait\_for\_mutex2 = WaitForSingleObject(h\_Mutex2, -1);

readcount--;

if (readcount == 0)

{

//最后一个读者,唤醒写者

LeaveCriticalSection(&cs\_Write);

}

ReleaseMutex(h\_Mutex2); //释放互斥信号

}

////////////////////////////////////////////////////////////////////////////////

//写者优先--写者线程

//p：写者线程信息

void WP\_WriterThread(void \*p)

{

DWORD m\_delay; //延迟时间

DWORD m\_persist; //写文件持续时间

int m\_serial; //线程序号

DWORD wait\_for\_mutex3;

//互斥对象

HANDLE h\_Mutex3;

h\_Mutex3 = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "mutex3");

//从参数中获得信息

m\_serial = ((ThreadInfo\*)(p))->serial;

m\_delay = (DWORD)(((ThreadInfo\*)(p))->delay \* INTE\_PER\_SEC);

m\_persist = (DWORD)(((ThreadInfo\*)(p))->persist \* INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Writer thread %d sents the writing require.\n", m\_serial);

//阻塞互斥对象mutex3，保证对writecount的访问、修改互斥

wait\_for\_mutex3 = WaitForSingleObject(h\_Mutex3, -1);

writecount++; //修改写者数目

if (writecount == 1)

{

//第一个写者，等待读者读完

EnterCriticalSection(&cs\_Read);

}

ReleaseMutex(h\_Mutex3);

//进入写者临界区

EnterCriticalSection(&cs\_Write);

//写文件

printf("Writer thread %d begines to write to the file.\n", m\_serial);

Sleep(m\_persist);

//退出线程

printf("Writer thread %d finishing writing to the file.\n", m\_serial);

//离开临界区

LeaveCriticalSection(&cs\_Write);

//阻塞互斥对象mutex3，保证对writercount的访问、修改互斥

wait\_for\_mutex3 = WaitForSingleObject(h\_Mutex3, -1);

writecount--;

if (writecount == 0)

{

//写者写完，读者可以读

LeaveCriticalSection(&cs\_Read);

}

ReleaseMutex(h\_Mutex3);

}

//////////////////////////////////////////////////////////////////

//写者优先处理函数

//file：文件名

void WriterPriority(char\* file)

{

DWORD n\_thread = 0; //线程数目

DWORD thread\_ID; //线程ID

DWORD wait\_for\_all; //等待所有线程结束

//互斥对象

HANDLE h\_Mutex1;

h\_Mutex1 = CreateMutex(NULL, FALSE, "mutex1");

HANDLE h\_Mutex2;

h\_Mutex2 = CreateMutex(NULL, FALSE, "mutex2");

HANDLE h\_Mutex3;

h\_Mutex3 = CreateMutex(NULL, FALSE, "mutex3");

//线程对象

HANDLE h\_Thread[MAX\_THREAD\_NUM];

ThreadInfo thread\_info[MAX\_THREAD\_NUM];

readcount = 0; //初始化readcount

writecount = 0; //初始化writecount

InitializeCriticalSection(&cs\_Write); //初始化临界区

InitializeCriticalSection(&cs\_Read);

ifstream inFile;

inFile.open(file); //打开文件

printf("Writer Priority:\n\n");

while (inFile)

{

//读入每一个读者、写者的信息

inFile >> thread\_info[n\_thread].serial;

inFile >> thread\_info[n\_thread].entity;

inFile >> thread\_info[n\_thread].delay;

inFile >> thread\_info[n\_thread++].persist;

inFile.get();

}

for (int i = 0; i < (int)(n\_thread); i++)

{

if (thread\_info[i].entity == READER || thread\_info[i].entity == 'r')

{

//创建读者线程

h\_Thread[i] = CreateThread(NULL, 0,

(LPTHREAD\_START\_ROUTINE)(WP\_ReaderThread),

&thread\_info[i], 0, &thread\_ID);

}

else

{

//创建写者线程

h\_Thread[i] = CreateThread(NULL, 0,

(LPTHREAD\_START\_ROUTINE)(WP\_WriterThread),

&thread\_info[i], 0, &thread\_ID);

}

}

//等待所有线程结束

wait\_for\_all = WaitForMultipleObjects(n\_thread, h\_Thread, TRUE, -1);

printf("All reader and writer have finished, operation.\n");

}

////////////////////////////////////////////////////////////////////////////

//主函数

int main(int argc, char\* argv[])

{

char ch;

while (true)

{

//打印提示信息

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf(" 1:Reader Priority\n");

printf(" 2:Writer Priority\n");

printf(" 3:Exit to Windows\n");

printf("Enter your choice(1,2 or 3):");

//如果输入信息不正确，继续输入

do

{

ch = (char)\_getch();

} while (ch != '1' && ch != '2' && ch != '3');

system("cls");

//选择3，返回

if (ch == '3')

return 0;

//选择1，读者优先

else if (ch == '1')

ReaderPriority("thread.dat");

//选择2，写者优先

else

WriterPriority("thread.dat");

//结束

printf("\nPress Any Key To Continue:");

\_getch();

system("cls");

}

return 0;

}

**步骤4：**将“thread.dat”文件复制到项目文件夹中。

1 R 3 5

2 W 4 5

3 R 5 2

4 R 6 5

5 W 5.1 3

**步骤5：**按“F5”开始调试，注意路径里不要含有中文。

**步骤6：**按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。



操作能否正常进行？如果不行，则可能的原因是什么？

操作能够正常运行，如果不行，可能是因为文件路径中含有中文，或者代码中含有中文字符。

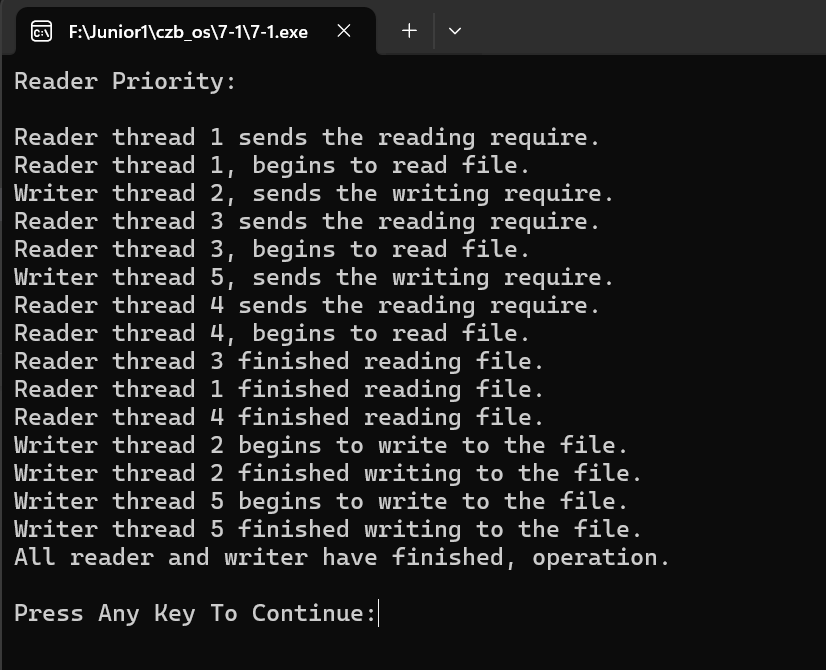
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

运行结果是：

文本

描述已自动生成

图片 1 对文件输入代码进行修改，改进开始会读入0号Write线程的问题



图片 2 读者优先运行结果

图形用户界面, 文本

描述已自动生成

图片 3 写者优先运行结果

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**步骤7：**分析程序里是如何实现读者/写者优先的，详细描述实现流程。

读者优先：

ReaderPriority函数先从thread.dat文件中读入读者写者的基本信息，再分别创建读者，写者线程。在读者线程函数RP\_ReaderThread中定义了控制对readcount值修改的互斥变量HANDLE h\_Mutex;在读者线程数量变化时通过wait\_for\_mutex = WaitForSingleObject(h\_Mutex, -1);和ReleaseMutex(h\_Mutex);对h\_Mutex进行down/up操作，保证读者线程互斥访问readcount。并且在第一个读者进入临界区时，通过EnterCriticalSection(&RP\_Write);对读者写者共用的读写临界区的互斥信号量RP\_Write进行down操作，在最后一个读者离开临界区时，通过LeaveCriticalSection(&RP\_Write);对读写临界区的互斥信号量 RP\_Write进行up操作。实现了只要有读者在访问临界资源，写者必须等待全部读者访问完毕才能进入临界区的读者优先功能。

其中，写者线程函数RP\_WriterThread就只有简单的对读写临界区的互斥信号量 RP\_Write进行down/up操作。但优先级较低。

写者优先：

WriterPriority函数先从thread.dat文件中读入读者写者的基本信息，再分别创建读者，写者线程。在读者线程函数WP\_ReaderThread中定义了HANDLE h\_Mutex1;和HANDLE h\_Mutex2;其中h\_Mutex1用于控制读者进入读者临界区，在对h\_Mutex1进行down操作之后，马上进行EnterCriticalSection(&cs\_Read);等待进入读者临界区，mutex2则保证对readcount的访问、修改互斥，如果是第一个读者，等待写者写完再对写者临界区的互斥信号量进行down操作EnterCriticalSection(&cs\_Write);直到最后一个读者结束才通过LeaveCriticalSection(&cs\_Write)唤醒写者。

在写者线程函数WP\_WriterThread中，定义了h\_Mutex3保证对writecount的访问、修改互斥，同样在第一个写者创建时，等待读者读完再对写者临界区的互斥信号量进行down操作EnterCriticalSection(&cs\_Read);直到最后一个写者结束写操作调用LeaveCriticalSection(&cs\_Read);函数离开读者临界区，读者才可以读。这样的互斥信号量控制实现了写者优先。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

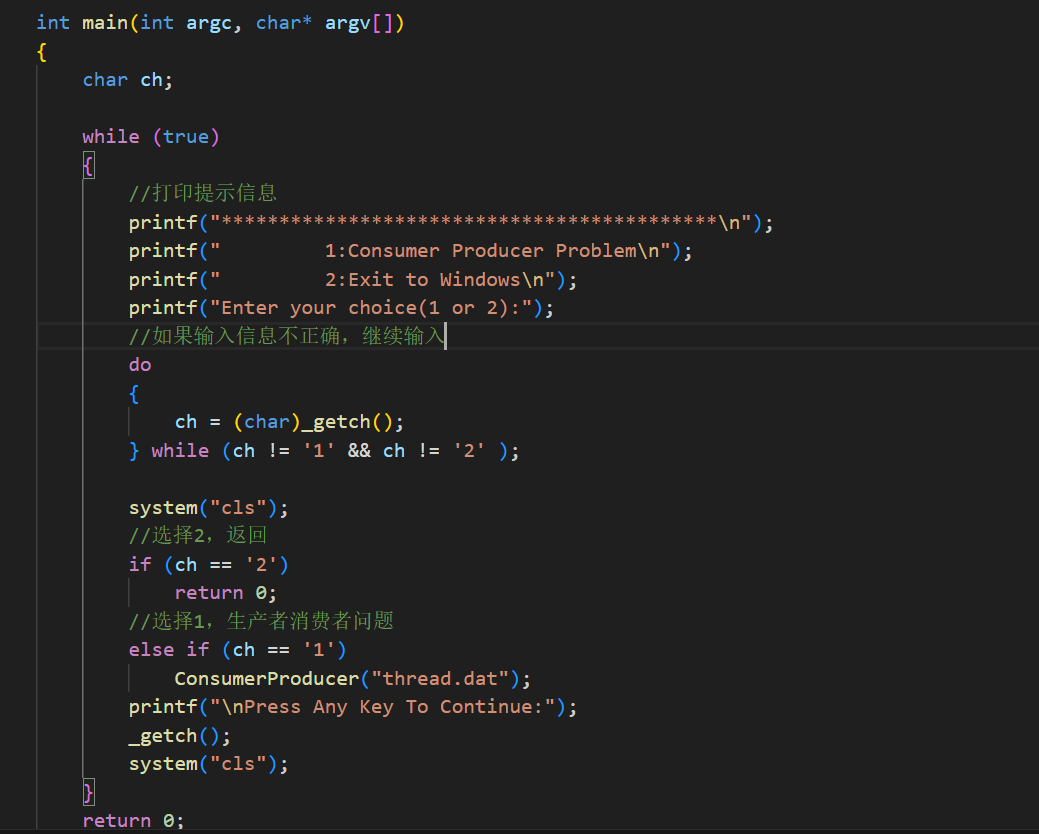
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**选作：**在熟悉清单7-1源代码的基础上，用P、V操作实现多个生产者—消费者问题。

测试数据文件包括n+1行测试数据，第一行说明几个缓冲区，其余n行分别描述创建的n个线程是生产者还是消费者，以及生产产品（或消费产品）的时间。每行测试数据包括几个字段，各字段间用空格分隔。第一字段为一个正整数，表示线程序号。第二字段表示相应线程角色，P表示生产者，C表示消费者。第三字段为一个正数，表示生产产品（或消费产品）的时间。消费者还可以有几个字段，分别表示此消费者消费哪些生产者（线程号）生产的产品。缓冲区需互斥访问。

请描述你所做的工作：



图片 4 修改主函数

文本

描述已自动生成

图片 5 预先设置缓冲区，缓冲区槽个数为2，临界区，缓冲区队列，信号量Mutex/Empty/Full

文本

描述已自动生成

图片 6 生产者消费者问题函数（1）

文本

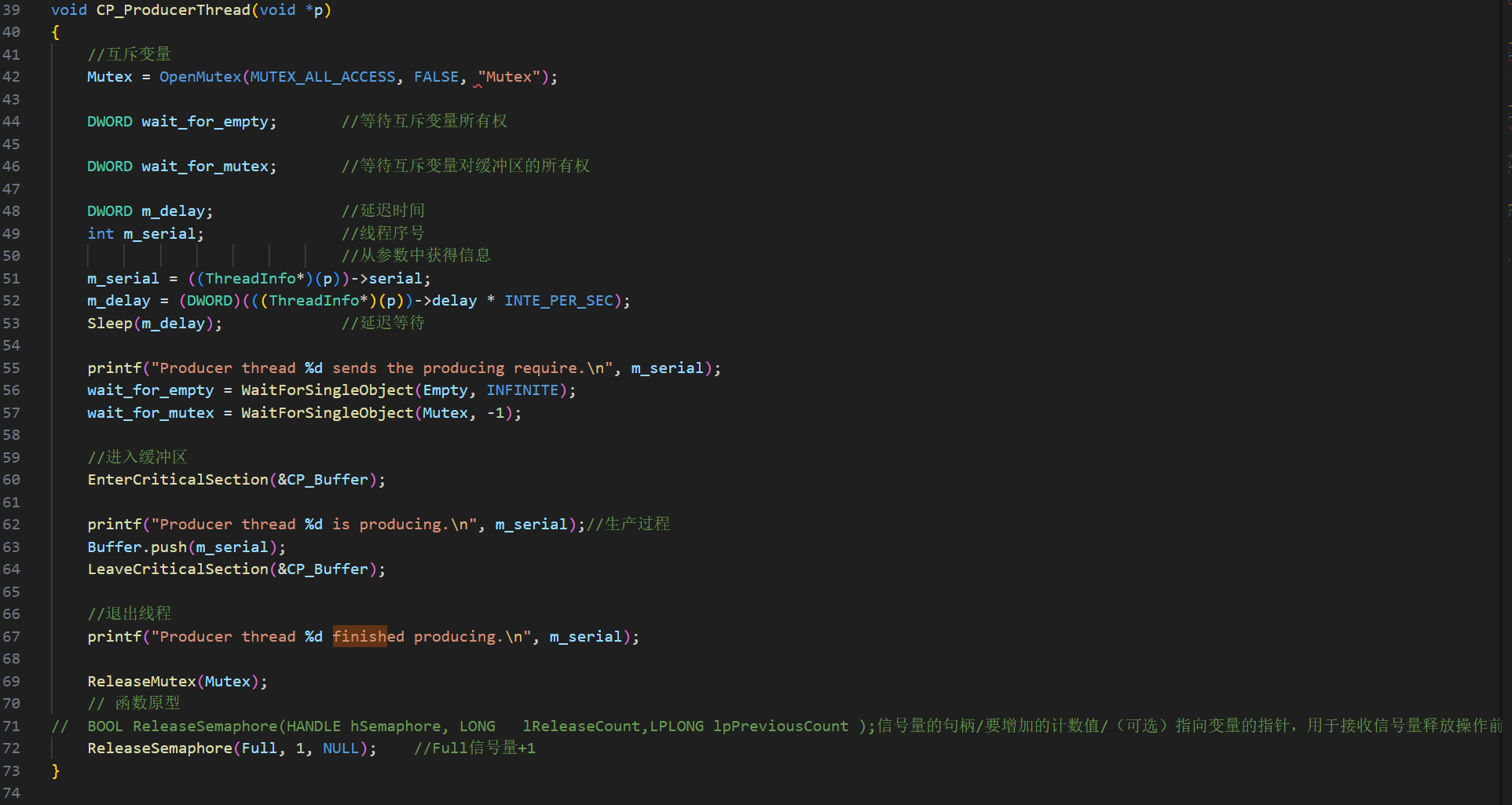
描述已自动生成

图片 7 生产者消费者问题函数（2），读入文件，创建线程

表格

中度可信度描述已自动生成

图片 8 thread.dat文件内容（这里只用了一个缓冲区）

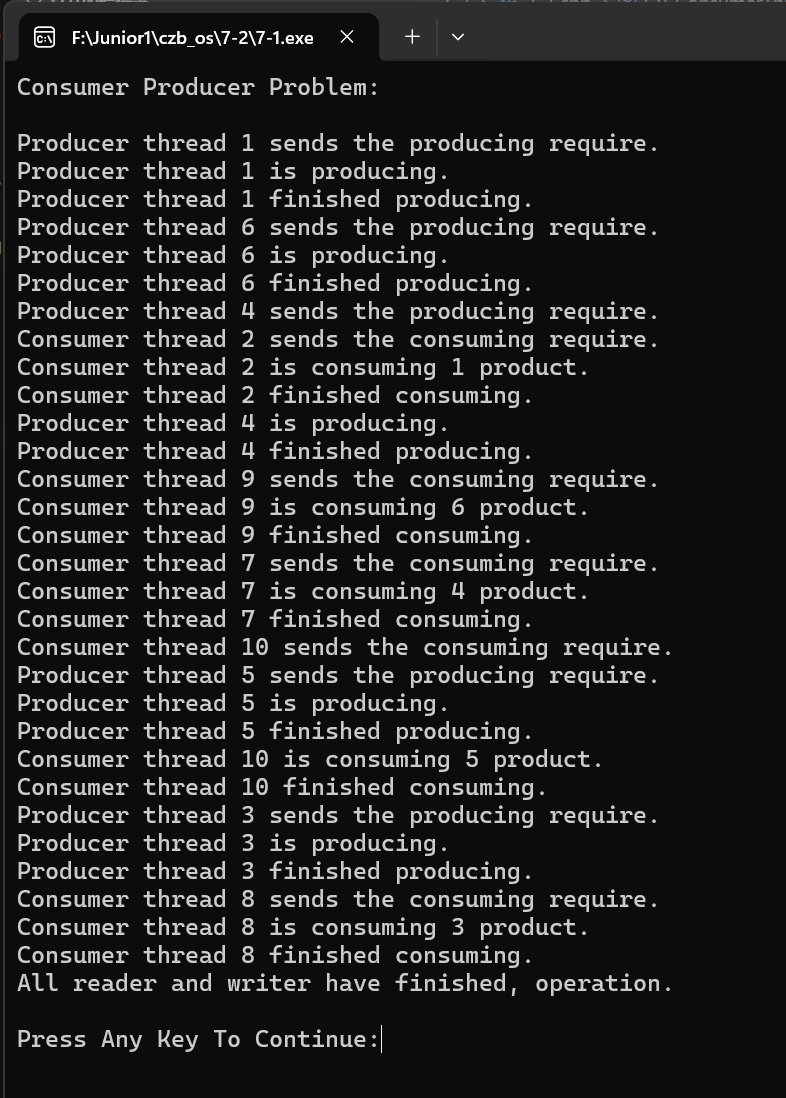


图片 9 生产者线程函数，先同步再互斥访问缓冲区临界资源，生产时将自己的线程序号压入Buffer队列，离开临界区时，给Full信号量up

文本

描述已自动生成

图片 10 消费者线程函数，先同步再互斥访问缓冲区临界资源，消费时输出消费的产品序号，并且将该产品推出队列，离开临界区时给Empty信号量up



图片 11 运行结果，由于缓冲区只有两个槽存放产品，所以生产者线程4在发出请求后被阻塞，等到消费者线程2消费掉产品1后才允许生产者4继续生产。同理，当缓冲区槽为空时，消费者线程10被阻塞，直到生产者线程5生产完之后才能消费。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_