进程同步与互斥

1. 实验目的

掌握进程同步和互斥原理，理解生产者-消费者模型；

学习Windows2000/xp中的多线程并发执行机制；

学习使用Windows SDK解决读者－写者问题。

1. 实验内容

参考给出的伪码，编程解决读者－写者问题的程序。

要求：

有两组并发进程：读者和写者，共享一个文件F，要求：

(1)允许多个读者可同时对文件执行读操作；

(2)只允许一个写者往文件中写信息；

(3)任一写者在完成写操作之前不允许其他读者或写者工作；

(4)写者执行写操作前，应需已有的写者和读者全部退出。

(5)要求仿真程序产生3个读者进程，两个写者进程，读写者都周期性地产生读写要求，读写操作要持续一定时间。

1. 实验过程

这里设置了读者优先

实验数据：

1 W 1 5

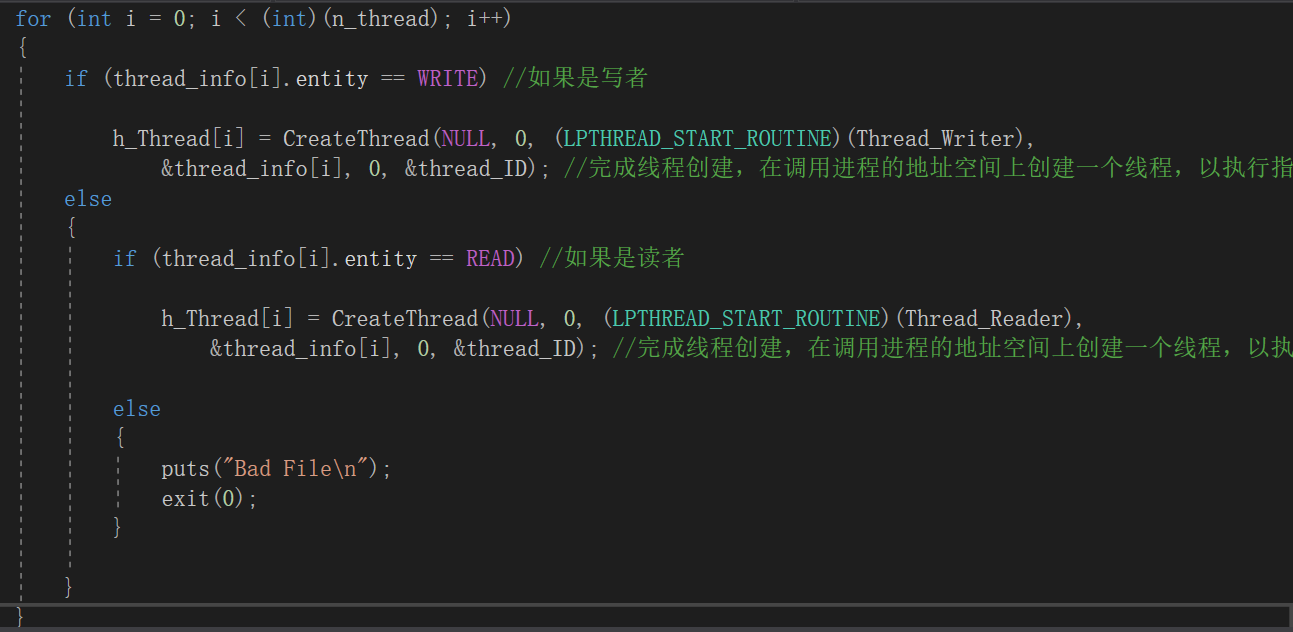
2 W 2 5

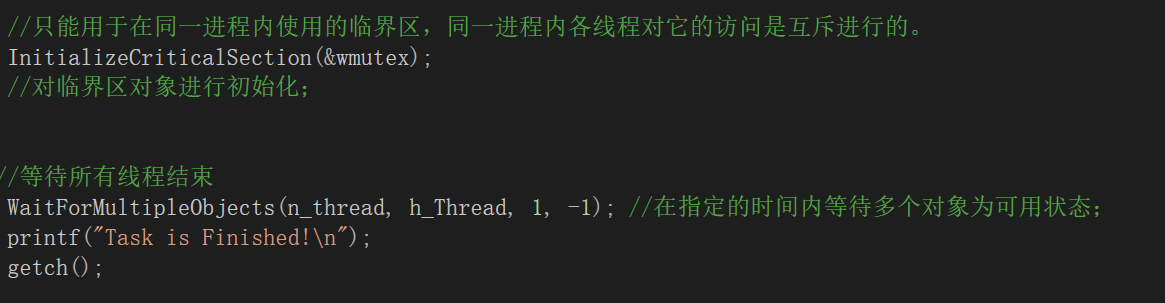
3 R 3 2

4 R 4 5

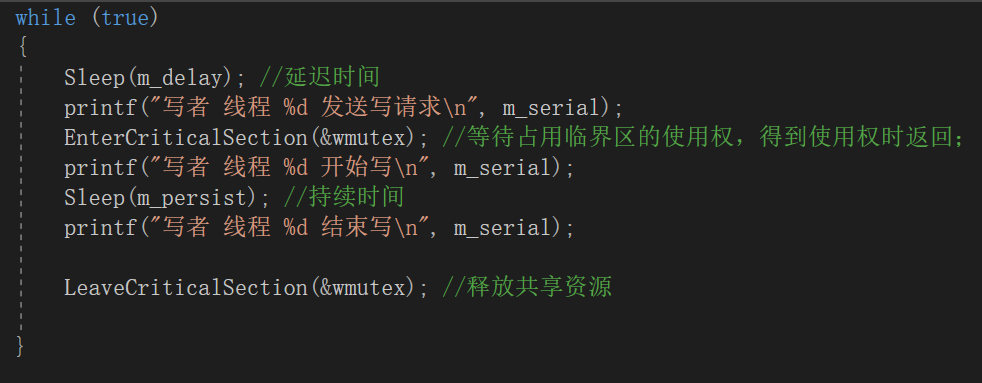
5 R 5 5

初始化：

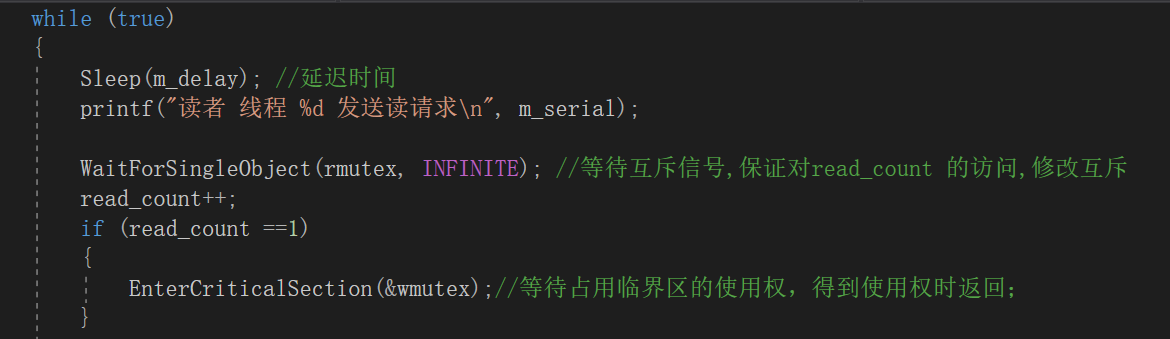


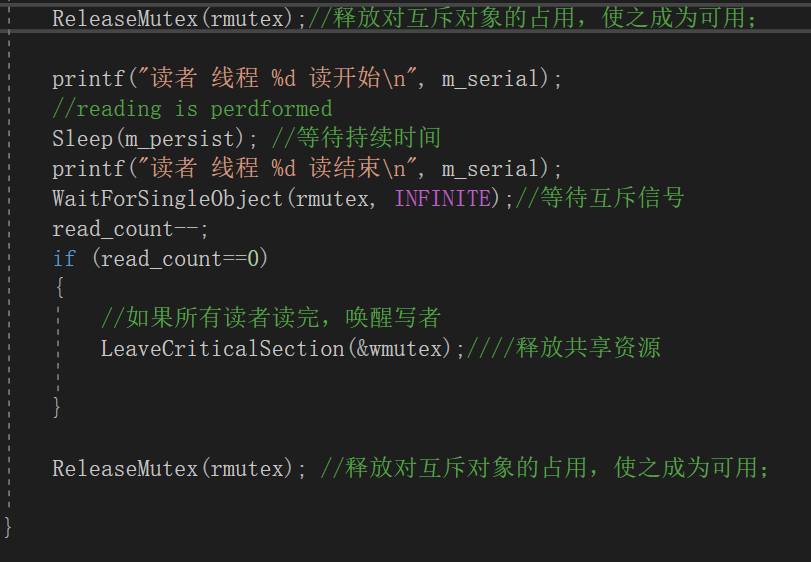


写线程：



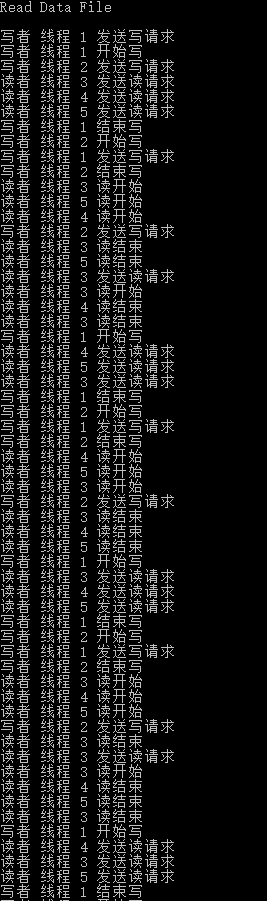
读线程：





1. 实验结果

由于程序持续一定时间，这里截取部分读写情况



1. 实验源代码

// ProducConsum.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include"pch.h"

#include<stdio.h>

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

#include <fstream>

#include <io.h>

#include <string.h>

#include <stdio.h>

#define INTE\_PER\_SEC 100 //延迟时间ms

#define MAX\_THREAD\_NUM 64 //最大线程数

#define SEM\_MAX\_FULL 64 //信号量的最大值

#define INFINITE 0xFFFFFF

#define WRITE 'W' //文件中的读者

#define READ 'R' //文件中的写者

using namespace std;

struct ThreadInfo //线程结构体

{

int serial; //标识

char entity; //实体

double delay; //延迟时间

double persist; //持续时间

};

int buff\_num; //临界区大小

int read\_count=0;

//CRITICAL\_SECTION sem\_mutex;

CRITICAL\_SECTION wmutex;//临界资源

HANDLE rmutex=CreateMutexA(NULL,FALSE,"rmutex"); //互斥对象

void ProducConsum(const char \*file); //

void Thread\_Writer(void \*p); //写者

void Thread\_Reader(void \*p); //读者

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

{

ProducConsum("wr\_data.txt");

return 0;

}

void ProducConsum(const char \*file) //初始化

{

DWORD n\_thread = 0;

DWORD thread\_ID;

HANDLE h\_Thread[MAX\_THREAD\_NUM]; //最大线程数

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

ifstream inFile;

inFile.open(file); //读取文件

puts("Read Data File \n");

while (inFile)

{

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

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

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

inFile >> thread\_info[n\_thread].persist;//持续时间

n\_thread++;

inFile.get();

}

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

{

if (thread\_info[i].entity == WRITE) //如果是写者

h\_Thread[i] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)(Thread\_Writer),

&thread\_info[i], 0, &thread\_ID); //完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。

else

{

if (thread\_info[i].entity == READ) //如果是读者

h\_Thread[i] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)(Thread\_Reader),

&thread\_info[i], 0, &thread\_ID); //完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。

else

{

puts("Bad File\n");

exit(0);

}

}

}

//只能用于在同一进程内使用的临界区，同一进程内各线程对它的访问是互斥进行的。

InitializeCriticalSection(&wmutex);

//对临界区对象进行初始化；

//等待所有线程结束

WaitForMultipleObjects(n\_thread, h\_Thread, 1, -1); //在指定的时间内等待多个对象为可用状态；

printf("Task is Finished!\n");

getch();

}

void Thread\_Writer(void \*p) //写线程

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

//读参数

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

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

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

while (true)

{

Sleep(m\_delay); //延迟时间

printf("写者 线程 %d 发送写请求\n", m\_serial);

EnterCriticalSection(&wmutex); //等待占用临界区的使用权，得到使用权时返回；

printf("写者 线程 %d 开始写\n", m\_serial);

Sleep(m\_persist); //持续时间

printf("写者 线程 %d 结束写\n", m\_serial);

LeaveCriticalSection(&wmutex); //释放共享资源

}

}

void Thread\_Reader(void \*p) //读者线程

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

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

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

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

while (true)

{

Sleep(m\_delay); //延迟时间

printf("读者 线程 %d 发送读请求\n", m\_serial);

WaitForSingleObject(rmutex, INFINITE); //等待互斥信号,保证对read\_count 的访问,修改互斥

read\_count++;

if (read\_count ==1)

{

EnterCriticalSection(&wmutex);//等待占用临界区的使用权，得到使用权时返回；

}

ReleaseMutex(rmutex);//释放对互斥对象的占用，使之成为可用；

printf("读者 线程 %d 读开始\n", m\_serial);

//reading is perdformed

Sleep(m\_persist); //等待持续时间

printf("读者 线程 %d 读结束\n", m\_serial);

WaitForSingleObject(rmutex, INFINITE);//等待互斥信号

read\_count--;

if (read\_count==0)

{

//如果所有读者读完，唤醒写者

LeaveCriticalSection(&wmutex);////释放共享资源

}

ReleaseMutex(rmutex); //释放对互斥对象的占用，使之成为可用；

}

}