操作系统

实验报告

实验名称：线程的创建和并发执行

院系：计算机学院·网络空间安全学院

班级：

学号：

姓名：

指导老师：匡林爱

日期：2020年10月13日

1. **实验目的和要求：**

本实验要求编写一个程序，完成多个线程的创建，使得所创建的多线程在处理机上并发执行。通过本实验，学习在Win32程序中利用操作系统提供的API创建线程，并通过所创建线程的运行情况来获得关于多线程并发的感性认识，以及加深对临界资源(本实验中临界资源是屏幕)互斥访问的理解。

1. **实验内容：**

程序启动时由系统系统创建一个进程来执行程序，此时进程只包含一个主线程，该主线程执行程序的main代码。

主线程运行时首先等待用户键入要创建的线程数(赋予变量iThread)，以及每个线程运行的时间(秒数，赋予变量wRunTime)，接着调用API函数GetSystemTime( )获得当前的系统时间(格林威治时间)，计算出新线程的生命结束点时间(均转换成北京时间)；然后循环调用函数CreateThread( )来创建iThread个子线程，这些子线程执行同一段代码(threadwork( ) 函数)；最后，主线程根据系统时间判断这些子线程的生命周期是否结束，当到达子线程结束时刻，则关runFlag(全局变量)。

各子线程：在每个子线程执行的这段代码(函数threadwork( )的代码)中，首先给出本线程开始运行的提示信息(不同线程用线程号区别，线程号是主线程创建子线程时传递给子线程的，程序中传送的是循环控制变量i的值作为子线程号的)，然后循环判断runFlag是否被主线程关闭，未关则显示“Thread x 第 n 次被调度运行”的提示(此处x代表子线程编号，n是某个自然数)，睡眠一段随机时间。如此不断循环；当runFlag关时，子线程结束循环，显示“几号线程已结束”字样，然后调用return语句结束本线程。

**相关Win32 Api函数介绍：**

1．CreateThread( ) // 创建一个新线程

2．GetSystemTime( ) // 获得系统时间

3．InitializeCriticalSection( ) // 初始化临界区对象

4．EnterCriticalSection( ) // 等待指定临界区对象的所有权

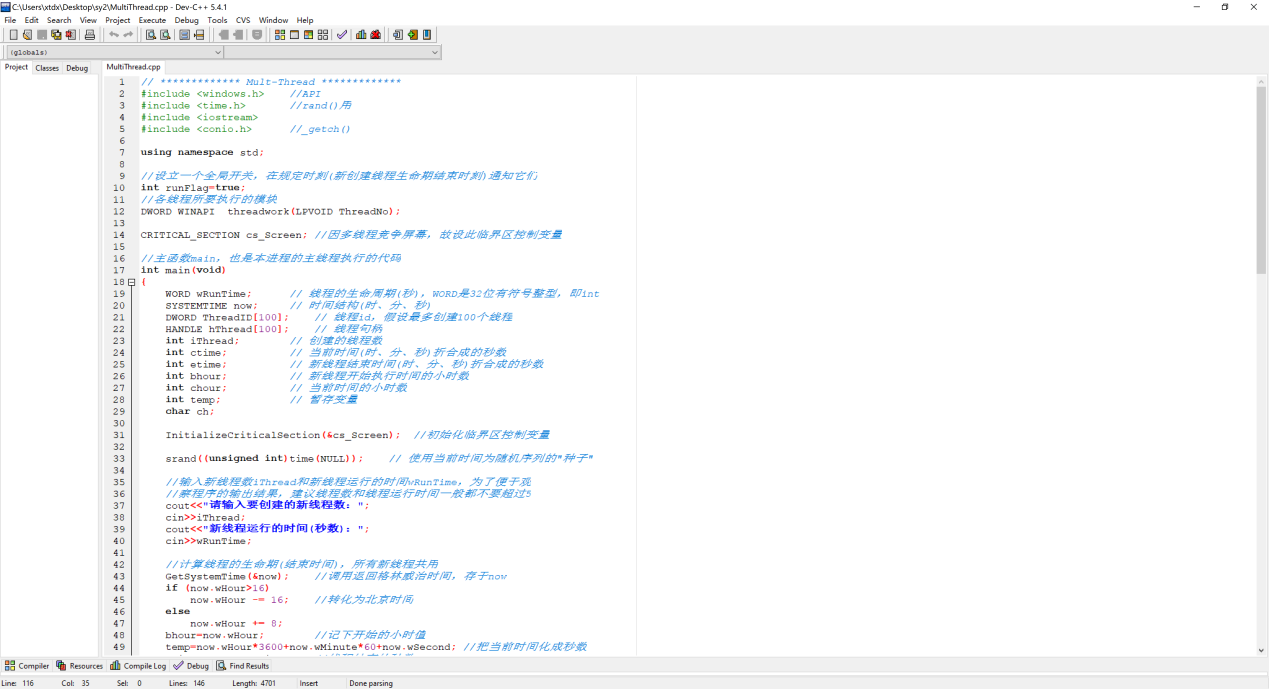
5．LeaveCriticalSection( ) // 释放指定临界区对象的所有权

6．Sleep( ) // 当前线程休眠(挂起suspend)一段时间

7．ExitThread ( ) // 强行退出(主动结束)线程 ——本程序没有使用此函数

8．GetExitCodeThread ( ) // 获得当初调用ExitThread()时设置的退出码

1. **实验步骤：**
2. **编写“MultiThread.cpp”：**

****

1. **编译运行**
2. **实验源程序**

**MultiThread.cpp：**

// \*\*\*\*\*\*\*\*\*\*\*\*\* Mult-Thread \*\*\*\*\*\*\*\*\*\*\*\*\*

#include <windows.h> //API

#include <time.h> //rand()用

#include <iostream>

#include <conio.h> //\_getch()

using namespace std;

//设立一个全局开关，在规定时刻(新创建线程生命期结束时刻)通知它们

int runFlag=true;

//各线程所要执行的模块

DWORD WINAPI threadwork(LPVOID ThreadNo);

CRITICAL\_SECTION cs\_Screen; //因多线程竞争屏幕，故设此临界区控制变量

//主函数main，也是本进程的主线程执行的代码

int main(void)

{

WORD wRunTime; // 线程的生命周期(秒)，WORD是32位有符号整型，即int

SYSTEMTIME now; // 时间结构(时、分、秒)

DWORD ThreadID[100]; // 线程id，假设最多创建100个线程

HANDLE hThread[100]; // 线程句柄

int iThread; // 创建的线程数

int ctime; // 当前时间(时、分、秒)折合成的秒数

int etime; // 新线程结束时间(时、分、秒)折合成的秒数

int bhour; // 新线程开始执行时间的小时数

int chour; // 当前时间的小时数

int temp; // 暂存变量

char ch;

InitializeCriticalSection(&cs\_Screen); //初始化临界区控制变量

srand((unsigned int)time(NULL)); // 使用当前时间为随机序列的"种子"

//输入新线程数iThread和新线程运行的时间wRunTime，为了便于观

//察程序的输出结果，建议线程数和线程运行时间一般都不要超过5

cout<<"请输入要创建的新线程数：";

cin>>iThread;

cout<<"新线程运行的时间(秒数)：";

cin>>wRunTime;

//计算线程的生命期(结束时间)，所有新线程共用

GetSystemTime(&now); //调用返回格林威治时间，存于now

if (now.wHour>16)

now.wHour -= 16; //转化为北京时间

else

now.wHour += 8;

bhour=now.wHour; //记下开始的小时值

temp=now.wHour\*3600+now.wMinute\*60+now.wSecond; //把当前时间化成秒数

etime=temp+wRunTime; //线程结束的秒数

//循环创建新线程，数量有参数指定

cout.fill('0');

cout.width(2); cout<<"新线程开始时间："<<now.wHour<<':';

cout.width(2); cout<<now.wMinute<<':';

cout.width(2); cout<<now.wSecond<<"(主线程输出信息)"<<endl;

for (int i=0; i<iThread; i++) {

/\* 线程函数的参数类型必须是LPVOID(即void \*)，故用

强制类型转换运算符(LPVOID)将整型i转换成LPVOID \*/

hThread[i]=CreateThread(NULL,0,threadwork, (LPVOID)i,0,&ThreadID[i]);

Sleep(100); // 主线程让出时间，使新建的线程运行

}

// 在子线程工作时不断循环，判断这些线程生命是否到期

while (runFlag)

{

EnterCriticalSection(&cs\_Screen); //准备进入临界区

cout<<"\*\*\*\*\*\*\*\*\*\*\*\* 主线程正在运行 ………… "<<endl;

LeaveCriticalSection(&cs\_Screen); // 退出临界区

GetSystemTime(&now);

if (now.wHour>16)

now.wHour -= 16; //转化为北京时间

else

now.wHour += 8;

if (now.wHour<bhour) //处理时间过"00::00"的情况

chour=now.wHour+24;

else

chour=now.wHour;

ctime=chour\*3600+now.wMinute\*60+now.wSecond; //当前时间的秒数

if (ctime>=etime) //若已到新线程结束时间

{

runFlag = FALSE;

EnterCriticalSection(&cs\_Screen); //准备进入临界区

cout.width(2); cout<<"新线程结束时间："<<now.wHour<<':';

cout.width(2); cout<<now.wMinute<<':';

cout.width(2); cout<<now.wSecond<<"(主线程输出信息)"<<endl;

LeaveCriticalSection(&cs\_Screen); // 退出临界区

}

Sleep(1000); //主线程睡眠100ms

}

//在整个进程(主线程)结束前留出一段时间好让各子线程完成自己

//的工作(也便于我们从它们结束前的输出来观察线程同步问题)

Sleep(2000);

GetSystemTime(&now);

if (now.wHour>16)

now.wHour -= 16; //转化为北京时间

else

now.wHour += 8;

//由于子线程已经结束，屏幕显示不再需要互斥

cout.width(2); cout<<"主线程结束时间："<<now.wHour<<':';

cout.width(2); cout<<now.wMinute<<':';

cout.width(2); cout<<now.wSecond<<"(主线程输出信息)"<<endl;

cout.fill(' ');

ch=\_getch(); //为便于观察程序不在开发环境运行的输出结果

return 0;

}

//每个新线程执行的代码

DWORD WINAPI threadwork(LPVOID ThreadNO) //LPVOID即(void \*)

{

int napTime; //睡眠时间(毫秒)

int iThreadNO=(int)ThreadNO; // 线程编号

int count=0;

//提示创建的相应线程已经启动

EnterCriticalSection(&cs\_Screen); //准备进入临界区

cout.width(2);

cout<<"Thread "<<iThreadNO<<" 开始运行。"<<endl;

LeaveCriticalSection(&cs\_Screen); // 退出临界区

//以下线程用于使线程睡眠一段时间(让出CPU共其它线程动

//作，并保证主线程能更及时获得CPU以改变runFlag标志)

while (runFlag)

{

napTime=100+rand()%200; //子线程睡眠时间在100~300ms之间

count++;

EnterCriticalSection(&cs\_Screen); //准备进入临界区

cout<<"Thread "<<iThreadNO<<" 第";

cout.width(2);cout<<count<<" 次被调度运行 …… ";

cout<<"(====== 这是线程 "<<iThreadNO<<" 输出的信息 ======)"<<endl;

LeaveCriticalSection(&cs\_Screen); // 退出临界区

Sleep(napTime); //睡眠一随机时间更能观察线程并发执行情况

}

EnterCriticalSection(&cs\_Screen); //准备进入临界区

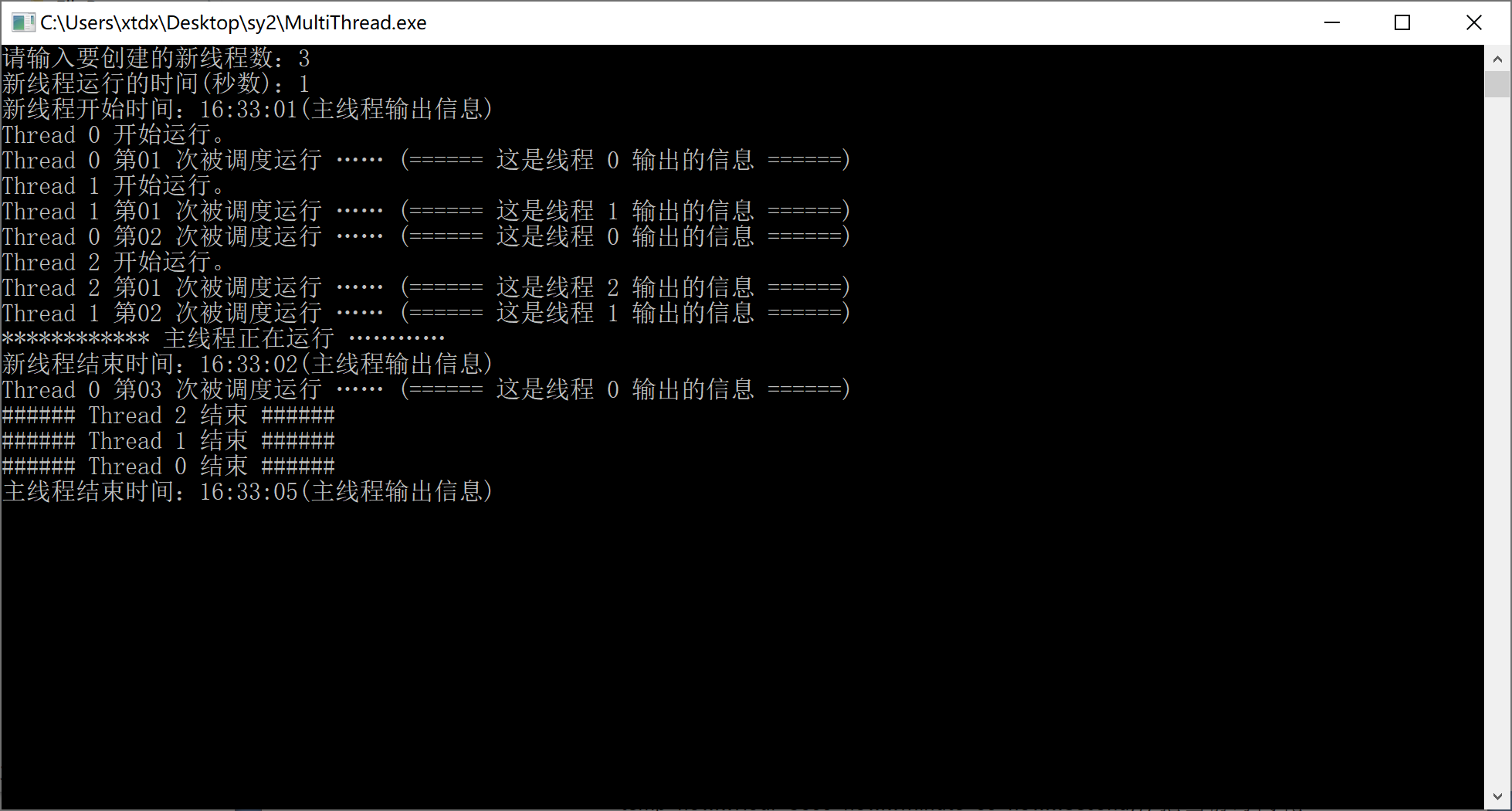
cout<<"###### Thread "<<iThreadNO<<" 结束 ######"<<endl;

LeaveCriticalSection(&cs\_Screen); // 退出临界区

return (DWORD)iThreadNO;

**}**

1. **实验结果**

****

1. **实验总结：**

通过这次线程的创建和并发执行实验，我学习到了在Win32程序中利用操作系统提供的API创建线程，并通过所创建线程的运行情况来获得关于多线程并发的感性认识，以及加深对临界资源互斥访问的理解。学会了他们的用法之后我觉得对我以后的代码生涯也大有益处！我在实验期间还详细查找了多线程相关资料，通过阅读这些资料，我也弄懂了许多之前没有弄懂的问题。这次成功的线程创建和并发执行实验也增强了我的自信心。总而言之，这次实验也和之前的实验一样，拓宽了我的眼界，使我更加熟悉了代码编写方面的操作，我现在已经有了一个比较纯熟的处理手段，我相信这对我以后的工作生活一定会起到很大的帮助！