# 第 2 章 Windows API 多线程编程模型

### 章概述

多核程序设计流程的第二部分编程模型与实现要求选定编程语言，编写满足功能需求的代码。这部分具体的工作包括：编码、调试、运行。常用的编程模型分为显式模型和隐式模型两大类。本章将介绍显式编程模型——Windows API。

### 章重点与难点

重点：Windows API 的调用、参数的传递。

难点：数据竞争的解决方法。

东软电子出版社

## 导引项目：生产者-消费者问题

### 【项目描述】

生产者把生产的一系列的产品放入有限的存储空间内，每一个产品占据一个存储空间，消费者从这有限的存储空间内获取产品进行消费使用。

### 【算法设计】

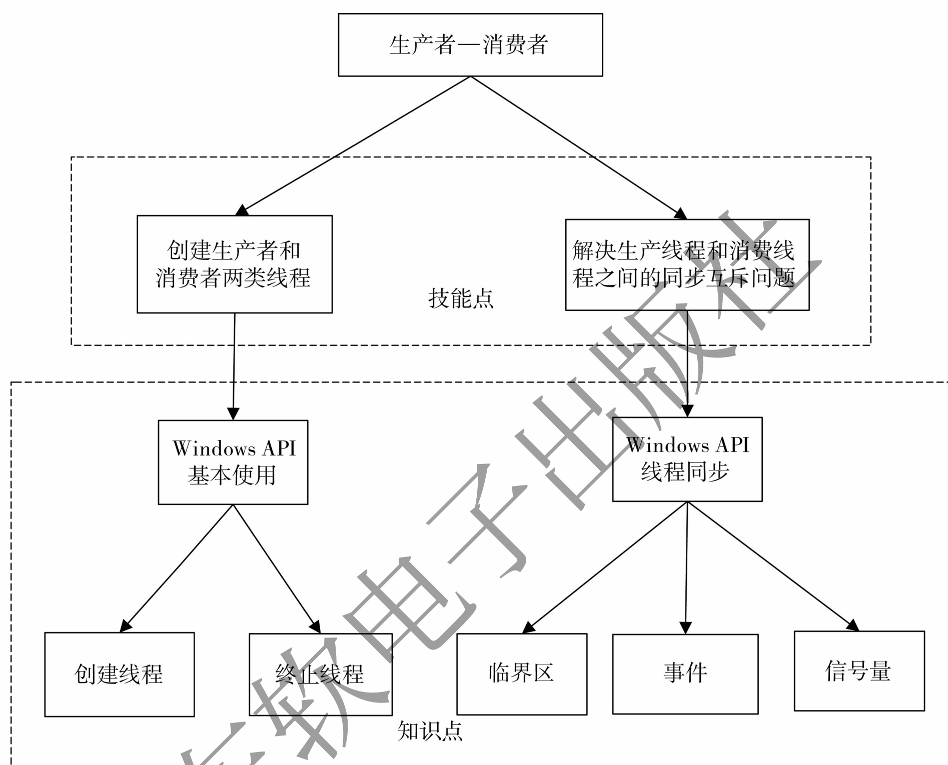
分析问题描述发现当生产者和消费者足够多并且有限的存储空间较大时，这一问题会变得相当复杂；当生产产品和消费产品的消耗时间较长时，这一问题的执行时间也会比较长。因此在模拟该问题时为了达到良好的性能和执行效率可以考虑采取多线程编程的方法来解决。为了便于思考，可以将问题简化为只有一个生产者和一个消费者。由于这一问题中有明显的生产者和消费者两种不同的角色完成不同的功能，因此可以采取任务分解的方法，由多个线程分别扮演着两种角色，完成不同的功能。

在本项目中需要注意的是生产者和消费者之间的同步和互斥关系。首先生产者能够生产产品的条件是需要有空闲的存储空间，而消费者能够消费的条件是需要有已经放入存储空间的产品，这两者之间是一种同步的关系。同时生产者和消费者又不可以同时对同一存储空间进行

操作，因此两者之间又具有互斥关系。

**【选择编程模型】**

本项目选择 Windows API 编程模型实现，首先利用 Windows API 创建生产者和消费者两类线程并行执行各自的功能，然后在线程并行执行时，利用临界区、信号量、事件等同步机制解决线程之间的互斥和同步问题。本项目所需的技能点和知识结构如图 2-1 所示。



东软电子出版社

**图 2-1　导引项目：生产者和消费者技能点与知识点结构图**

下面将按章节详细介绍各个知识点，并在章末给出该项目的实现代码。

# 2.1　Windows 线程库介绍

Windows API 是 Microsoft 公司 Windows 操作系统的内核与上层应用程序的接口，是开发 Windows 应用程序的手段之一。应用程序通过对相应接口的调用来获得系统的相关能力。直接利用 Win32 API 编写代码虽然需要开发人员对操作系统有一定的了解，但是程序的代码量小，运行效率较高。

Windows NT 操作系统的诞生使 Windows 应用程序开发人员可以利用 Windows 线程库开发出多线程程序，而处理器技术的飞速发展也为多线程程序的开发提供了更加良好的平台。因此 Windows 线程库也成为在 Windows 平台上开发多线程程序的方法之一。在 Windows 线

程库中提供了大量的与多线程程序开发相关的 API,供程序开发人员用以管理线程,有效地解决线程间同步的问题,开发出高性能的并行程序。

# 2.2 Windows API 的基本使用

## 2.2.1 内核对象

在介绍 Windows 线程库之前,首先需要简单地了解一下内核对象以及句柄的概念。内核对象是由操作系统内核分配的,只能由内核访问的一个内存块,用来供系统和应用程序使用和管理各种系统资源。在编写 Windows 开发程序时,经常需要创建、打开和操作各种内核对象,包括线程对象、事件对象、文件对象、文件映射对象、作业对象、互斥量对象、管道对象、进程对象、信标对象和等待计时器对象等。这些对象都是通过调用函数来创建的。例如:CreateThread() 函数可以使系统创建一个线程对象。不同的内核对象拥有不同的数据结构,它的成员负责维护该对象的各种信息。

由于内核对象的数据结构只能被内核程序访问,因此应用程序无法在内存中找到这些数据结构并直接改变它们的内容。Windows 规定了这种限制条件,目的是为了确保内核对象的结构保持状态的一致性。为了方便对内核对象的应用,Windows 提供了一组函数对这些结构进行操作。当调用一个用于创建内核对象的函数时,该函数就返回一个用于标识该对象的句柄。该句柄是一个值,进程中的任何线程都可以使用这个值,将这个句柄传递给 Windows 的各个函数,让系统明确当前操作的内核对象。每个进程被初始化时,系统为它分配一个句柄表,用于保存该进程使用的内核对象的信息,而句柄值则是相应内核对象在句柄表中的索引值,因此句柄值是进程相关的,相同的句柄值在不同的进程中可能标识不同的内核对象,也就是说,句柄是相对于进程的,这使得对内核对象的使用更加安全,操作系统更加健壮。

内核对象由内核拥有,而不是由进程拥有,因此进程是可以共享内核对象的,一个进程终止执行,它使用的内核对象并不一定会被撤销。内核知道有多少进程正在使用某个内核对象,因为每个内核对象都包含一个使用计数,每当有一个进程使用该内核对象时,其使用计数就加 1,而使用该内核对象的进程中止或结束时,其使用计数就减 1,当使用计数为 0 时,操作系统才撤销该内核对象。

## 2.2.2 线程管理

### 1. 线程的创建

在 Windows API 中,创建线程的 API 是 CreateThread(),语法格式如下:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, //进行高级设置
```

东北电子出版社

```
DWORD dwStackSize，  //线程堆栈大小
LPTHREAD_START_ROUTINE lpStartAddress，  //函数指针，指向实际运行的代码
LPVOID lpParameter，  //参数指针
DWORD dwCreationFlags，//设置标志
LPDWORD lpThreadId)；  //线程 ID
```

在程序中想创建线程时，调用该函数即可。当 CreateThread()被调用时，系统会创建一个线程内核对象，返回一个线程句柄，通过该句柄可以对线程做其他操作。

CreateThread()的第一个参数，lpThreadAttributes，是一个指向 LPSECURITY_ATTRIBUTES 数据结构的指针，用于对线程句柄在系统中的使用方法进行高级控制，制定了线程的安全属性，默认值为 NULL。

第二个参数，dwStackSize，用于设定线程堆栈的大小，以字节为单位。一般情况下设置为 0，由系统按默认值设置堆栈的大小。

第三个参数，lpStartAddress，定义了一个指向子线程实际运行代码的函数指针，不能为 NULL。该函数原型如下：

DWORD WINAPI 函数名(LPVOID p)；

从线程函数的定义可以看出，线程在执行时可以以 void 指针作为参数传参，线程执行结束后可以返回一个状态信息，这些都为线程与外界提供了通信机制。

第四个参数，lpParameter，用于父线程向子线程传递参数，它与线程函数形参的类型一致。该参数可以是一个数字值，也可以是指向包含其他信息的一个数据结构的指针。没有参数时值为 NULL。

第五个参数，dwCreationFlags，用于控制创建线程的状态，有两种可选值。如果该值是 0，那么线程创建后立即进行调度执行。如果该值是 CREATE_SUSPENDED，则系统产生该线程后，该线程处于挂起状态，直到通过调用 ResumeThread(HANDLE hThread)函数唤醒线程。

第六个参数，lpThreadId，它必须是 DWORD 的一个有效地址，CreateThread 使用这个地址来存放系统分配给新线程的 ID。线程 ID 是系统赋予线程在系统范围内唯一的标识符。

在这里应该强调的是，使用 ID 来跟踪线程是存在风险的，因为虽然系统不会为其他线程赋予与被跟踪线程相同的 ID，但是当被跟踪线程终止后，系统就可以将这个 ID 赋予给其他线程了，这将导致预料之外的错误。如果不需要使用线程 ID，可以将 lpThreadId 设置为 NULL。

**2. 线程的终止**

在 Windows API 中常用的终止线程执行的方法有三种。

(1)直接返回。

通过线程函数的简单返回，可以保证线程资源被正确地清除。这是一种值得推荐的线程终止的方法。

(2) ExitThread()函数。

线程在执行结束前可以通过显式调用 ExitThread()函数进行线程的终止。函数原型：

(3)VOID ExitThread(DWORD dwExitCode)。

该函数将终止线程的运行，并且会使操作系统清除该线程使用的所有操作系统资源。但是，C++资源(如 C++类对象)将不被撤销。由于这个原因，最好从线程函数返回，而不是通过

东软电子出版社

调用 ExitThread 来返回。

(4) CloseHandle()函数。

当需要从某线程外部终止一线程时,可以通过调用 CloseHandle()函数来实现。函数原型:

```
BOOL CloseHandle(HANDLE hObject)
```

在终止线程时,只需要将被终止线程的句柄作为参数,调用 CloseHandle()函数即可。

**3. 线程的挂起与恢复**

进程中的每一个线程在其数据结构中保存一个挂起计数,该挂起计数被内核监视,用于控制线程的状态转换。当挂起计数为 0 时,表示线程准备被执行;当挂起计数大于 0 时,表示线程处于挂起状态。对于挂起计数的访问可以通过调用 SuspendThread()和 ResumeThread()函数来实现。

挂起线程的函数原型:DWORD SuspendThread(HANDLE hThread)。

恢复线程的函数原型:DWORD ResumeThread(HANDLE hThread)。

调用 SuspendThread()挂起指定线程,如果挂起执行成功,则线程执行被终止,挂起计数的值增加 1。调用 ResumeThread()后,挂起计数的值减 1。如果挂起计数的值为 0 则不会再减。

**4. 线程的等待**

在并行程序中,线程间的执行经常需要保持一定的执行顺序。一个线程需要等待其他线程执行完毕或者内核对象发出一个特定的信号后才可以继续执行。在这里,等待线程常采用将自己阻塞而等待其他线程执行完或内核对象成为激发态的方法来保证线程间的执行顺序。Windows API 中提供了一组使线程阻塞其自身执行等待其他线程执行完或内核对象为已激活状态的方法 WaitForSingleObject()和 WaitForMultipleObjects()。当等待的内核对象为线程时,表示线程等待。当只需要等待一个内核对象时,可以调用 WaitForSingleObject()函数。函数原型:

```
DWORD WaitForSingleObject(
            HANDLE hHandle, //被等待对象的句柄
            DWORD dwMilliseconds); //调用线程的有效等待时间
```

其中 hHandle 表示被等待的内核对象的句柄。dwMilliseconds 表示等待的有效时间,单位毫秒,如果在有效时间内被等待的内核对象没有被激活则调用线程不再等待,将 dwMilliseconds 设置为 INFINITE 表示一直等待至内核对象被激活为止。当 WaitForSingleObject()函数用于等待线程时,调用该函数的线程被阻塞挂起。函数中第一个参数表示被等待的线程的句柄,第二个参数表示等待的有效时间,在有效时间内被等待的线程执行完毕,则等待线程被激活继续执行代码,如果在有效时间内被等待的线程没有返回,则等待线程不再等待而是继续执行代码。

WaitForSingleObject()函数的返回值能够指明等待线程由阻塞变为激活状态的原因。如果被等待的内核对象变为已激活状态,那么返回值是 WAIT_OBJECT_0。如果设置的等待时间超时,则返回值是 WAIT_TIMEOUT。如果函数执行过程中发生错误返回值是 WAIT_FAILED。

如果线程需要同时等待多个内核对象时,使用 WaitForMultipleObjects()函数原型如下:

```
DWORD WaitForMultipleObjects(
          DWORD nCount,
          CONST HANDLE * lpHandles,
```

东软电子出版社

```
BOOL fWaitAll,
DWORD dwMilliseconds);
```

其中 nCount 表示等待的内核对象的句柄数；lpHandles 指向所等待的句柄对象数组；fWaitAll 参数是一个布尔值，如果为 TRUE，表示只有当指定的所有内核对象都变为已激活状态函数才返回，如果为 FALSE，表示只要指定的内核对象中有一个内核对象变为已激活状态函数即可返回；dwMilliseconds 参数的作用与它在 WaitForSingleObject()中的作用完全相同。当该函数用于线程对象时，表示等待多个线程，等待线程被阻塞挂起。nCount 表示等待的线程数，* lpHandles 指向被等待的多个线程的句柄，fWaitAll 为 TRUE 表示只有当所有被等待线程都执行完毕返回后等待线程才被激活，fWaitAll 为 FALSE 表示只要有一个被等待线程执行完毕返回调用线程就被激活，dwMilliseconds 表示线程等待的有效时间。

WaitForMultipleObjects()函数的返回值与 WaitForSingleObject()大致相同，只是当 fWaitAll 参数为 FALSE 时，应用程序可能希望知道是哪个内核对象变为已激活状态，这时的返回值为 WAIT_OBJECT_0 与 WAIT_OBJECT_0＋dwCount－1 之间的一个值。可以通过对返回值的判断来判定返回的是哪一个内核对象。

## 2.2.3　简单的例子

下面通过一个例子详细描述在 Microsoft Visual Studio 2010 环境下编写和运行多线程程序方法。

首先双击打开 Microsoft Visual Studio 2010 软件，通过选择菜单“文件”→“新建”→“项目”，在项目模板中选择“Win32”下的“Win32 控制台应用程序”，项目名称“Helloworld”，如图 2-2 所示。单击“确定”后，其他选项保持默认选项不变。

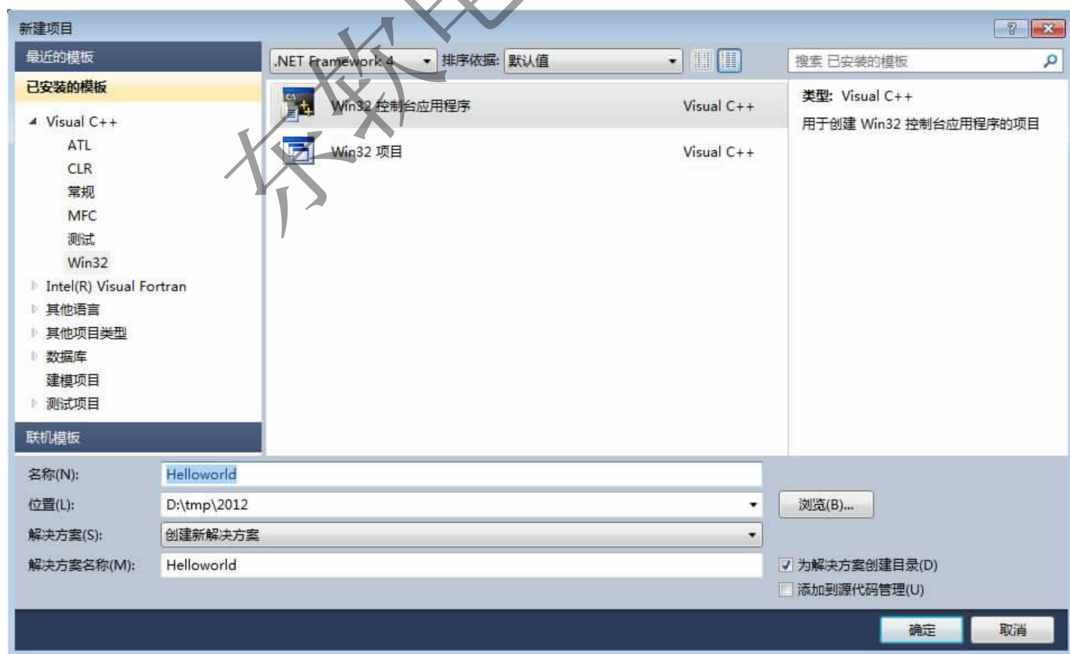



图 2-2　在 Microsoft Visual Studio 2010 下创建 Windows API 应用程序

在自动生成的文件 Helloworld.cpp 中输入代码 2-1。

东软电子出版社

**代码 2-1**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  const int numThreads=4;
4  DWORD WINAPI threadFunc(LPVOID arg) {
5      int myNum= *((int *)arg);
6      printf("Hello Thread! Thread number is %d\n", myNum);
7      return 0;
8  }
9  int _tmain(int argc, _TCHAR * argv[]){
10     HANDLE hThread[numThreads];
11   int tNum[numThreads];
12    for(int i=0; i < numThreads; i++)
13     {
14         tNum[i]=i;
15         hThread[i]=CreateThread(NULL, 0, threadFunc, &tNum[i], 0, NULL);
16     }
17    WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);
18    return 0;
19  }
```

在运行该项目前，需要单击“项目”→“属性”菜单，按照图 2-3～图 2-6 配置项目属性。配置完成后，按【F5】编译程序，按【Ctrl+F5】查看执行结束。

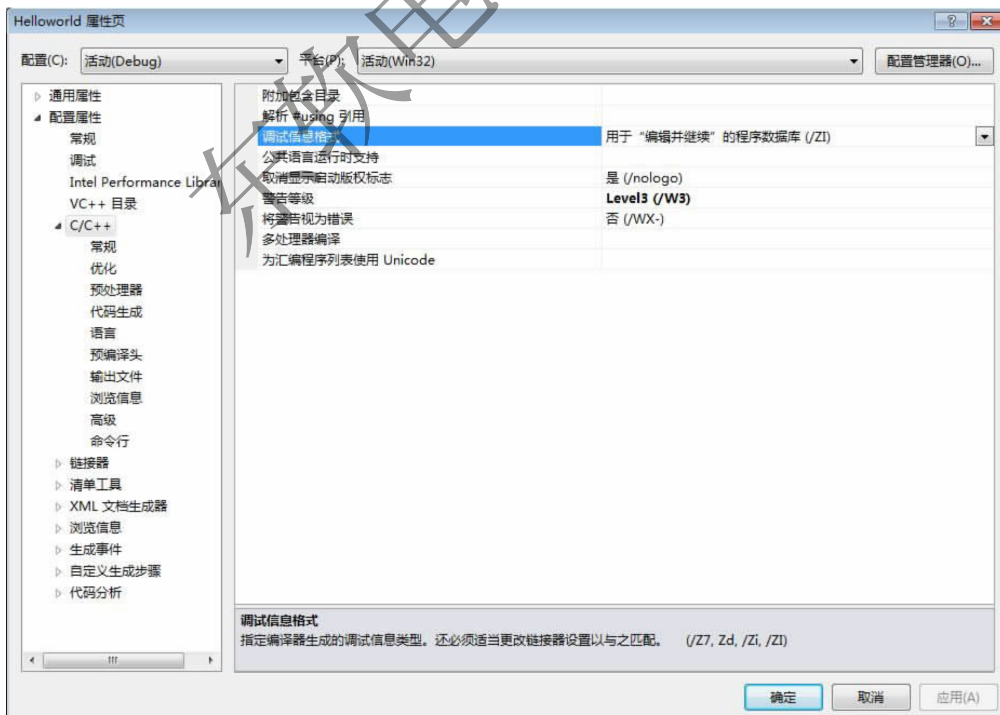



图 2-3 项目属性设置：“C/C++”→“常规”→“调试信息格式”

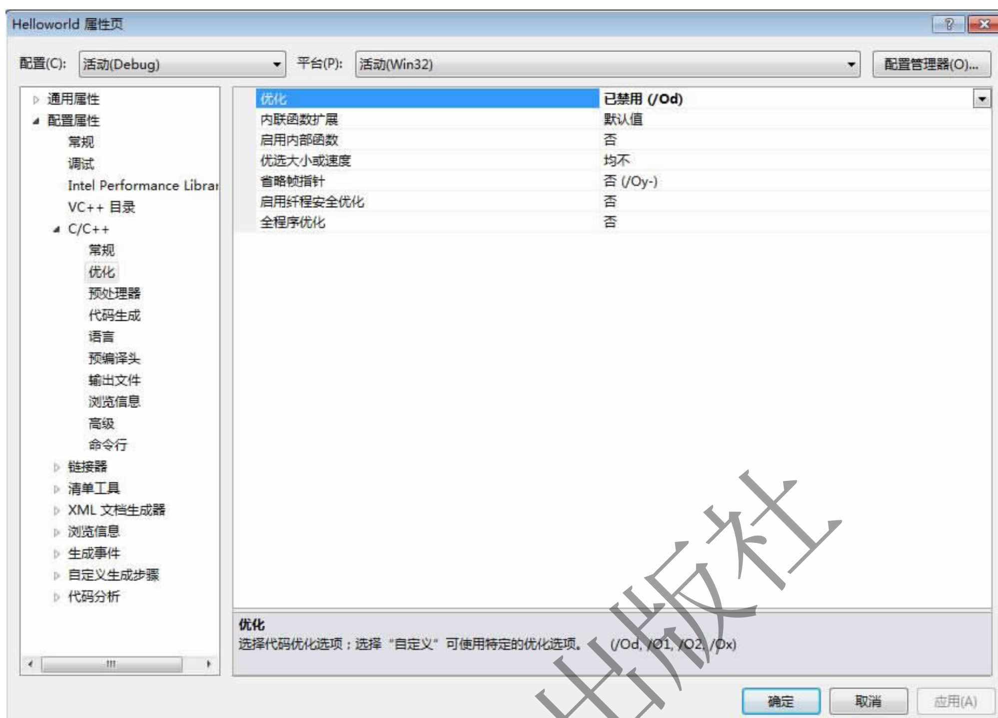



图 2-4 项目属性设置："C/++"→"优化"→"优化"

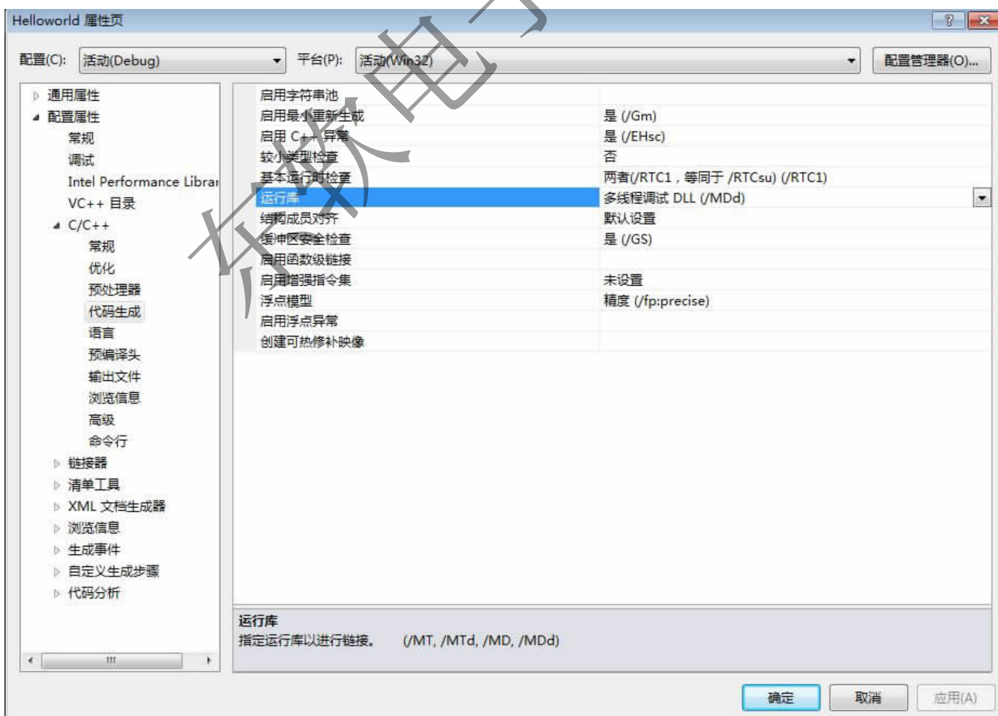



图 2-5 项目属性设置："C/++"→"代码生成"→"运行库"

东软电子出版社

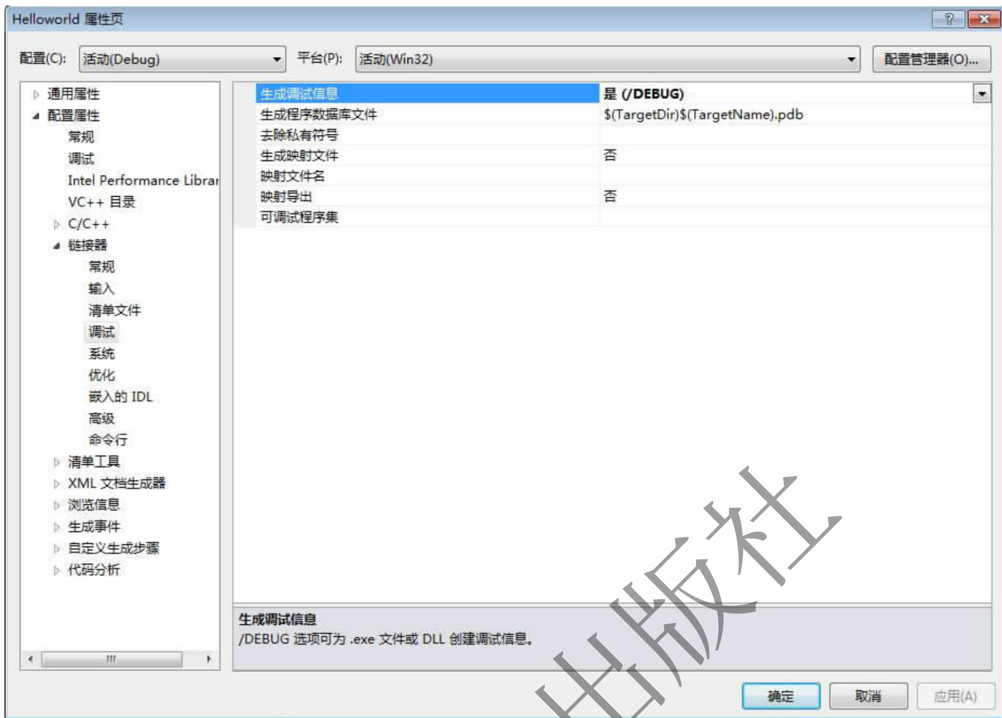



图 2-6　项目属性设置："链接器"→"生成调试信息"

这是一个简单的利用 Windows 多线程 API 创建线程的例子。在调用 Windows API 时，需要引入 windows. h 头文件。在主程序中用循环的方式调用 CreateThread()函数生成四个子线程分别去执行 helloFunc()函数，并将线程的句柄保存于 hThread 数组中(代码第 15 行)。四个线程各执行一次 helloFunc()函数，一种可能的输出结果为：

```
Thread number 0
Thread number 2
Thread number 1
Thread number 3
```

为了保证主线程在四个子线程都执行完成后再退出，在主程序中添加了第 17 行代码来等待所有子线程的返回，确保了程序执行的正确性。该例子还说明了给线程传递参数的方法。例子中子线程执行函数 threadFunc()时需要输出指针参数 arg 所指向的内容，而 arg 的值是由生成子线程的函数 CreateThread()的第四个参数传递。由于主线程为每一个子线程传递了不同的参数，所以通过子线程的执行结果可以判断出四个线程的执行顺序。

东软电子出版社

## 2. 3　Windows API 的线程同步

在多线程程序中，由于线程执行的并行性和操作系统对线程调度的随机性，使得程序开发人员不能预知线程的执行顺序。这虽然可以提高系统资源的利用效率，使程序能够达到更好的

执行性能，但也使程序的开发过程面临了一些随之而来的问题。可以预见，当所有的线程在互相之间不需要进行通信的情况下，程序不但可以很好的顺利进行，并且会达到最好的执行性能。但是，线程很少能够在所有的执行时间内都独立地进行操作。通常线程间会产生通信，引起数据竞争。常见的数据竞争有以下两种情况。

（1）当有多个线程访问共享资源，而共享资源具有唯一性。

（2）当某线程的执行需要其他线程的通知。

第一种情况统称为线程互斥，第二种情况统称为线程同步。在 Windows API 中应用于线程间通信，解决数据冲突的机制主要包括临界区、事件和信号量等。

## 2.3.1 临界区

临界区是一个特殊的代码段，在这段代码中有对共享资源的访问，线程首先需要取得对该临界区的访问权才可以进入临界区执行代码。对临界区访问权的获取保证了每次只有一个线程进入临界区执行，当一个线程进入临界区执行代码后，其他试图获取临界区访问权的线程会被系统阻塞挂起等待，直到临界区中的线程执行完毕释放访问权。这样保证了临界区操作的原子性以及对共享资源的互斥访问。

在 Windows API 中，关于临界区操作的函数有四个，如表 2-1 所示。在使用这些函数之前，必须定义一个临界区对象，它是一个类型为 CRITICAL_SECTION 的实例，通常为全局变量，这样方便多个线程对其引用。声明方法：

```
CRITICAL_SECTION cs;
```

**表 2-1　Windows API 临界区操作的函数**

| 函数原型 | 功能描述 | 举例 |
|---|---|---|
| void WINAPI InitializeCriticalSection( LPCRITICAL_SECTION lpCriticalSection)； | 对临界区对象的初始化。只有当临界区对象被初始化之后，才可以进一步被使用 | IntializeCriticalSection (&cs) |
| void WINAPI EnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection)； | 当临界区中没有线程时，调用该函数的线程将获得访问权进入临界区执行，否则将被阻塞，直到临界区被释放 | EnterCriticalSection (&cs) |
| void WINAPI LeaveCriticalSection( LPCRITICAL_SECTION lpCriticalSection)； | 释放临界区的函数，此时处于阻塞等待的一个线程将拥有临界区的访问权，继续执行 | LeaveCriticalSection (&cs) |
| void WINAPI DeleteCriticalSection( LPCRITICAL_SECTION lpCriticalSection)； | 当临界区不再被程序所需要时，可以通过调用该函数将其删除，该函数释放所有被分配来维护此临界区对象的系统资源 | DeleteCriticalSection (&cs) |

东软电子出版社

在使用临界区时，可以根据需要，在程序中定义多个临界区对象，比如 cs1 和 cs2 等，用来对不同的代码段进行互斥访问，这样做的好处是可以增加程序的并行度。总之，在设计多线程程序时，应该为每一个需要互斥的共享资源定义一个临界区变量。

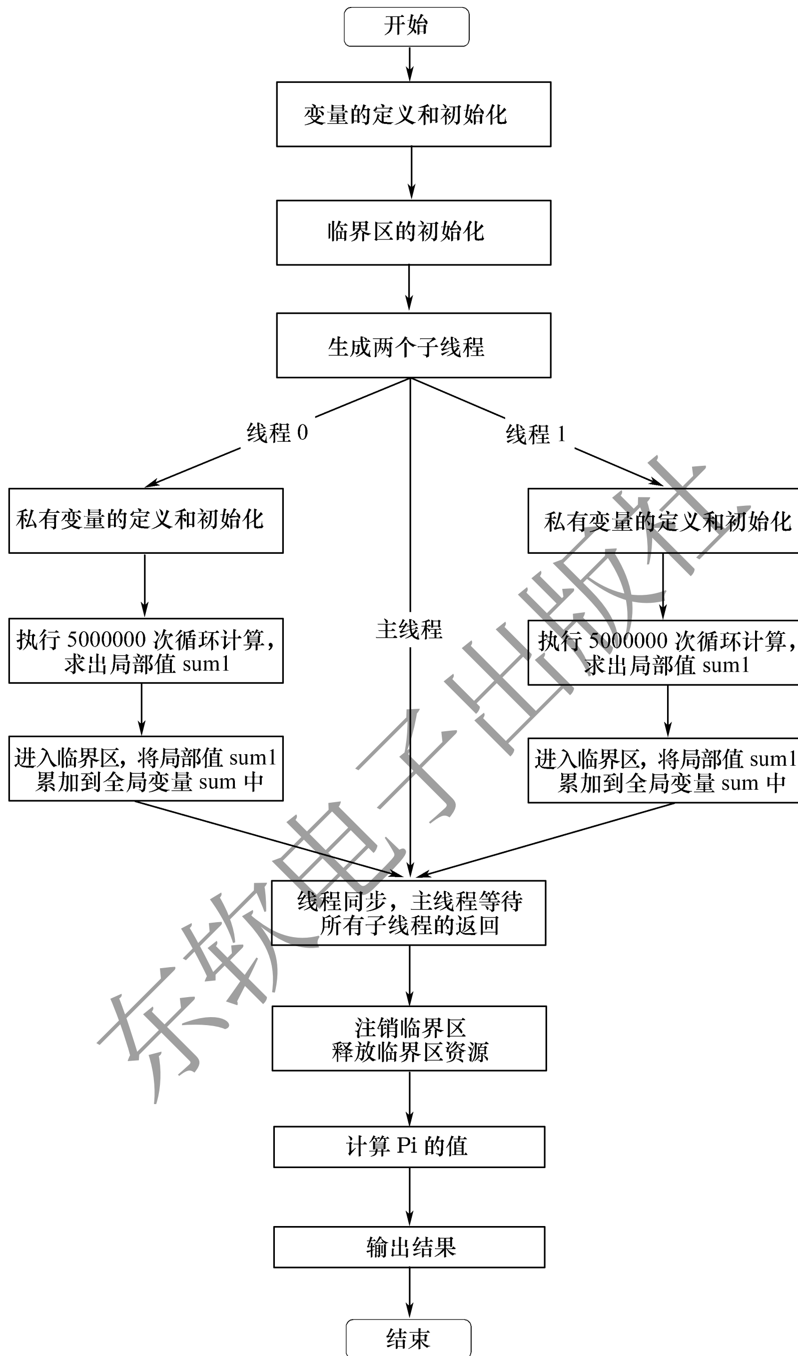
代码 2-2 以串行的方法实现了对圆周率Π值的计算。

**代码 2-2**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  #include <time.h>
4  static long num_steps=10000000;
5  double step, pi;
6  int _tmain(int argc, _TCHAR* argv[])
7  {
8    clock_t start, stop;
9    start=clock();
10   int i;
11   double x, sum=0.0;
12   step=1.0/(double) num_steps;
13   for(i=0; i< num_steps; i++){
14       x=(i+0.5)*step;
15       sum=sum+4.0/(1.0+x*x);
16   }
17   pi=step * sum;
18   stop=clock();
19   printf("Pi= %12.12f\n",pi);
20   printf("The time of calculation was %f seconds\n",((double)(stop-start)/1000.0));
21   return 0;
22 }
```

东软电子出版社

代码的实现采用积分的方法，通过 10000000 次迭代完成。执行结果 Pi=3.141592653590，计算时间 0.140000 秒。如果将代码 2-2 并行化，首先需要分析出程序中的并行性。因为程序的大部分计算由 for 循环迭代完成，所以并行时采取将循环划分成若干部分，每部分由一个线程执行，这是一种数据分解的方法，在分解过程中要注意迭代中数据的相关性。确定并行性后，设计并行算法。设计算法时，注意对共享资源的保护，并行程序算法流程图如图 2-7 所示，采用双线程并行。实现代码如代码 2-3 所示。




东软电子出版社

图 2-7　计算 Π 值的并行程序算法流程图

**代码 2-3**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  #include <time.h>
```

```
4  static long num_steps=10000000;
5  const int gNumThreads=2;
6  double step=0.0;
7  double pi=0.0,sum=0.0;
8  CRITICAL_SECTION gCS;   //定义全局的临界区变量gCS。
9  DWORD WINAPI threadFunction(LPVOID pArg)
10  {
11  int myNum= *((int *)pArg); //获取参数的值。
12  double sum1=0.0, x;
13  for(int i=myNum; i<num_steps; i+=gNumThreads)  //循环计算局部结果sum1。
14  {
15         x=(i+0.5)*step;
16         sum1=sum1+ 4.0/(1.0+x*x);
17  }
18  EnterCriticalSection(&gCS);  //进入临界区。
19    sum+=sum1;             //将sum1的值累加到sum中。
20  LeaveCriticalSection(&gCS);  //离开临界区
21  return 0;
22  }
23  int _tmain(int argc, _TCHAR* argv[])
24{
25  clock_t start, stop;
26   start=clock();
27  HANDLE threadHandles[gNumThreads]; //定义HANDLE类型的数组,用于存放线程的句柄。
28  int tNum[gNumThreads];
29  InitializeCriticalSection(&gCS); //临界区初始化。
30  step=1.0/(double) num_steps;
31  for(int i=0; i<gNumThreads;++i)  //循环创建两个子线程。
32  {
33    tNum[i]=i;
34    threadHandles[i]=CreateThread(NULL, 0, threadFunction,&tNum[i],0, NULL);    //子线程的
    创建,创建的子线程的同时传递tNum[i]参数,子线程执行threadFunction()函数。
35  }
36  WaitForMultipleObjects(gNumThreads, threadHandles, TRUE, INFINITE); //线程同步,主线程阻塞
    等待子线程。
37  DeleteCriticalSection(&gCS);  //注销临界区。
38  pi=step * sum;   //计算pi的值。
39  stop=clock();
40  printf("Pi= %12.12f\n",pi);  //输出结果。
41  printf("The time of calculation  was %f seconds\n",((double)(stop-start)/1000.0));
42  return 0;
43  }
```

东软电子出版社

执行结果 Pi=3.141592653590，计算时间 0.078000 秒，加速比 1.795。本例中实现了两个线程的协同工作，分别将计算的局部结果 sum1 累加到全局变量 sum 中。由于多线程对 sum 变量的修改会引起数据竞争，因此在这里采用了临界区方法，将对 sum 的更新操作定义为临界区操作（第 18～20 行代码）。在多线程程序中，对于在子线程执行的代码中出现的被更新的全局变量要多加考虑，因为具有这种特点的变量很有可能会引起数据冲突，如果处理不好会使程序产生功能性错误。

在本例中还应该注意临界区的使用方法：一个临界区定义及四个接口函数的调用。通常情况下，将临界区对象定义为全局变量，方便多个线程的使用。在创建子线程前调用 InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)函数对临界区进行初始化，然后调用 EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)函数获取临界区的访问权，访问结束后调用 LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)函数释放临界区的访问权，在退出程序前需要调用 DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)函数注销临界区，释放临界区资源。

### 2.3.2　事件

在所有内核对象中，事件内核对象是个比较简单的对象。它包括一个实用计数，一个用于指明该事件是自动重置事件还是人工重置事件的布尔值，还有一个用于指明该事件是否处于已激发状态的布尔值，它主要用于标识一个操作是否已经完成。

事件用于协调线程间的执行顺序来保证对共享资源操作的完整性。例如，当线程 A 在执行过程中需要线程 B 的计算结果时，需要协调 A，B 线程的执行顺序，保证 B 线程执行完成后，A 线程再获取 B 的结果进行计算。这时就可以使用事件机制完成这种线程间的同步，首先定义一个未激发的事件，A 线程在执行时以阻塞的方式等待事件被激发，而 B 线程在计算结束后将事件由未激发态设置为激发态，从而恢复 A 线程，使之继续执行。

在 Windows API 中，与事件相关的函数如表 2-2 所示。

东软电子出版社

**表 2-2　　Windows API 与事件相关的函数**

| 函数原型 | 功能描述 | 举例 |
|---|---|---|
| HANDLE CreateEvent(<br>LPSECURITY_ATTRIBUTES<br>lpEventAttributes，<br>BOOL bManualReset，<br>BOOL bInitialState，<br>LPCSTR lpName)； | 用于创建事件内核对象<br>lpEventAttributes 是设置事件安全性的参数<br>bManualReset 参数是个布尔值，指明创建一个人工重置的事件(TRUE)还是一个自动重置事件(FALSE)<br>bInitialState 参数用于指明该事件的初始化状态，激发状(TRUE)，未激发状(FALSE)<br>lpName 参数定义事件的名称 | HANDLE<br>hObj=CreateEvent(<br>NULL，FALSE，<br>FALSE，NULL)； |

| 函数原型 | 功能描述 | 举例 |
|---|---|---|
| DWORD WaitForSingleObject(<br>HANDLE hHandle,<br>DWORD dwMilliseconds);<br>DWORD WaitForMultipleObjects<br>(DWORD nCount,<br>CONST HANDLE *<br>lpHandles,<br>BOOL fWaitAll,<br>DWORD dwMilliseconds); | 用于等待事件被激发,等待线程处于阻塞状态<br>参数的具体含义上面已介绍过,不再累述 | WaitForSingleObject(<br>hObj, INFINITE); |
| BOOL SetEvent(HANDLE event); | 该函数仅用于人工重置时间,调用该函数可以将未激发的事件设置为激发态 | SetEvent(hObj); |
| BOOL ResetEvent(HANDLE event); | 调用该函数可以将该事件设置为未激发态 | ResetEvent(hObj); |

事件在创建时可以通过第二个参数来指定是人工重置事件还是自动重置事件。人工重置就是指通过 SetEvent()和 ResetEvent()来设置事件的激发状态,当人工重置的事件得到激发时,等待该事件的所有线程均被恢复,变为可调度线程。而自动重置的含义则是,当事件被 SetEvent()函数设置为已激发状态时,在等待该事件的诸多线程中,只有一个线程被恢复,之后系统自动将该事件置为未激发状态,在这个过程中,哪个线程被恢复是不可预知的。

事件的主要用途是标志事件的发生,并以此协调线程的执行顺序。下面是一个简单的应用事件的例子。

**代码 2-4**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  HANDLE hObj[2];
4  BOOL bigFind()
5    {
6    printf("this is a large-scale computation.\n");
7    return true;
8    }
9  DWORD WINAPI threadFunc(LPVOID arg)
10  {
11    BOOL bFound=bigFind();
12    if(bFound)
13     {
14    SetEvent(hObj[0]); //设置事件被激发。
15    }
16    return 0;
17   }
18   int _tmain(int argc, _TCHAR* argv[]){
19   hObj[0]=CreateEvent(NULL, FALSE, FALSE, NULL);  //创建一个未激发的自动重置事件。
```

东软电子出版社

```
20  hObj[1]=CreateThread(NULL,0,threadFunc,NULL,0,NULL); //创建子线程,执行 threadFunc()函数。
21  DWORD waitRet = WaitForMultipleObjects(2, hObj, FALSE, INFINITE);  //阻塞等待事件和线程
    返回。
22  switch(waitRet) {
23    case WAIT_OBJECT_0:     //事件返回。
24            printf("found it! \n");
25            WaitForSingleObject(hObj[1], INFINITE);  //阻塞等待线程。
26    case WAIT_OBJECT_0+1:       //线程返回。
27            printf("thread done\n");
28            break;
29    default:
30            printf("wait error: ret %u\n", waitRet);
31            break;
32  }
33  return 0;
34  }
```

在代码 2-4 中的主线程分别创建了一个子线程和一个未激发的自动重置事件两个内核对象。事件的句柄保存于 hOb 数组的第 0 个元素中,而子线程的句柄保存于 hOb 数组的第 1 个元素中。子线程执行 threadFunc()函数,主线程在代码第 21 行调用WaitForMultipleObjects()函数阻塞等待两个内核对象的激发,并且通过该函数的返回值来判断被激发的内核对象是线程还是事件。返回值是 WAIT_OBJECT_0 表示是事件被激发,返回值是 WAIT_OBJECT_0+1 表示线程已执行完成。子线程的执行完成了事件由未激发态到激发态的设置(第 14 行代码)。一种可能的执行结果:

```
this is a large-scale computation.
found it!
thread done
```

**代码 2-5**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  #define NUMTHREADS 4
4  HANDLE *threadHandles;
5  HANDLEeventHandles;
6  int sum;
7  DWORD WINAPI threadProc(LPVOID par)
8  {
9   int threadId= *((int *)par);
10  WaitForSingleObject(eventHandles,INFINITE); //等待事件被激发。
11  printf("The thread is %d,sum= %d.\n",threadId,threadId+sum);
12  return 0;
13  }
14  DWORD WINAPI masterThreadProc(LPVOID par)
```

东软电子出版社

```
15 {
16 sum=100;
17 SetEvent(eventHandles); //将事件设置为激发态。
18 return 0;
19 }
20 int _tmain(int argc, _TCHAR * argv[])
21 {
22 threadHandles=new HANDLE[NUMTHREADS+1];
23 int tNum[NUMTHREADS];
24 eventHandles=CreateEvent(NULL, TRUE, FALSE, NULL);//创建未激发的人工重置事件。
25 for(int i=0; i<NUMTHREADS;i++)   //循环创建四个子线程,并行 threadProc()代码。
26 {
27   tNum[i]=i;
28   threadHandles[i]=CreateThread(NULL, 0, threadProc, &tNum[i], 0, NULL);
29 }
30 threadHandles[NUMTHREADS]=CreateThread(NULL, 0, masterThreadProc, NULL, 0, NULL); //创建第
   五个线程,执行 masterThreadProc()函数。
31 WaitForMultipleObjects(NUMTHREADS+1,threadHandles,TRUE,INFINITE);  //主线程等待五个子线
   程的返回。
32 printf("The master  thread is  over.\n");
33 return 0;
34 }
```

代码 2-5 的一种可能的执行结果是:

```
The thread is 0,sum=100.
The thread is 1,sum=101.
The thread is 2,sum=102.
The thread is 3,sum=103.
The master thread is over.
```

在这个例子中,主线程共创建五个子线程和一个未激发的人工重置事件 eventHandles(第 24 行)。前四个子线程的执行需要第五个子线程为其准备好 sum 变量的值,为了保证程序执行的正确性,需要利用事件机制控制线程的执行顺序。例子中,前四个子线程执行threadProc()函数,第五个子线程执行 masterThreadProc()函数。前四个子线程等待 eventHandles 事件被激发,从而获取到 sum 变量的值,而第五个线程为 sum 变量赋值后激发 eventHandles 事件。由于 eventHandles 事件是一个人工重置事件,所以事件被激发后,所有等待该事件的线程都被恢复开始执行各自的代码。

如果本例中第 24 行代码修改为:eventHandles=CreateEvent(NULL, FALSE, FALSE, NULL);那么此时定义的就是一个自动重置事件,这种事件被激活后只能恢复一个等待该事件的线程,而其他等待该事件的线程仍然处于阻塞状态,不能被调度执行。

### 2.3.3 信号量

信号量是一种内核对象,用于对共享资源进行互斥访问,它除了像其他内核对象一样拥有

东软电子出版社

使用计数外,还拥有两个带符号的 32 位整数值,一个是最大资源数量,用于标识信号量能够控制的资源的最大数量,另一个是当前资源数量,用于标识当前可以使用的资源数量。

信号量的原理是:如果当前资源数量大于 0,那么等待信号量的线程可以获得一个资源并继续执行,信号量的当前资源数将减 1;如果当前资源数为 0,那么等待信号量的线程将处于阻塞等待状态,直到有线程释放信号量。

可以看出,信号量与前面提到的临界区有一定的相似之处。信号量也用于对一段代码进行访问控制,不同之处在于同时执行信号量代码的线程的个数可以大于 1,这一点使得信号量更适用于对有限资源的控制。例如:8 个线程的线程组对 2 台打印机的访问,很显然每次只能处理两个线程的打印任务,而其他线程的打印请求只能被阻塞至有空闲的打印机为止。在这种情况下,应用信号量完成对资源的控制是比较好的。信号量经常被用于对以下情况的控制:

(1)对有限数据空间的访问控制;

(2)对一段给定代码的线程访问数量的控制;

(3)对有限资源的访问控制。

在 Windows API 中,与信号量相关的函数如表 2-3 所示。

**表 2-3 Windows API 信号量相关的函数**

| 函数原型 | 功能描述 | 举例 |
|---|---|---|
| HANDLE CreateSemaphore(<br>LPSECURITY_ATTRIBUTES lpEventAttributes,<br>LONG lSemInitial,<br>LONG lSemMax,<br>LPCSTR lpSemName); | 用于创建信号量内核对象<br>lSemInitial 是设置信号量安全性的参数<br>lSemInitial 用于标识信号量的初始资源数<br>lSemMax 用于标识信号量的最大资源数<br>lpSemName 定义信号量的名称<br>在这里要注意:<br>0 <=lSemInitial <=lSemMax. lSemMax > 0 | HANDLE<br>hSem=CreateSemaphore(<br>NULL,1,1, NULL); |
| DWORD WaitForSingleObject<br>(HANDLE hHandle,<br>DWORD dwMilliseconds); | 执行减 1 操作,如果信号量的值等于 0 时,线程将被阻塞;信号量的值大于 0 时,信号量的值将被减 1 并返回,线程进入信号量代码执行 | WaitForSingleObject(<br>hSem, INFINITE); |
| BOOL ReleaseSemaphore<br>(HANDLE hSemaphore,<br>LONG cReleaseCount,<br>LPLONG lpPreviousCount); | 调用该函数,使线程在退出信号量代码前能够对信号量的当前资源数量进行递增操作,从而释放信号量<br>该函数将 cReleaseCount 的值添加到当前资源数量中,通常情况下 cReleaseCount 的值是 1,当然也可以传递更大的值。PlPreviousCount 用来返回信号量原始的当前资源数量,通常几乎没有应用程序关心这个值,因此可以传递 NULL 将它忽略 | ReleaseSemaphore(hsem,1,<br>NULL); |

信号量的应用实例如代码 2-6 所示。

**代码 2-6**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  FILE * fd;
4  int TotalWords=0;
5  const int NUMTHREADS=4;
6  HANDLE hSem1, hSem2;
```

东软电子出版社

```
7  int GetNextLine(FILE * f, char * Line)  //按行读取文件的内容到 Line 数组中。
8  {
9   if(fgets(Line, 132, f)==NULL)
10    if(feof(f))
11        return EOF;
12    return 1;
13  }
14  int GetWordAndLetterCount(char * Line)  //分析 Line 所指向的字符串中单词的个数。
15  {
16  int Word_Count=0, Letter_Count=0;
17      for(int i=0;i<132;i++)
18      {
19      if((Line[i]! =' ')&&(Line[i]! =0)&&(Line[i]! ='\n')) Letter_Count++;
20      else {
21              if(Letter_Count! =0){
22                      Word_Count++;
23                      Letter_Count=0;
24              }
25              if(Line[i]==0) break;
26          }
27    }
28  return Word_Count;
29  }
30  DWORD WINAPI CountWords(LPVOID arg)
31  {
32    BOOL bDone=FALSE;
33    char inLine[132]; int lCount=0;
34    while(! bDone)
35      {
36      WaitForSingleObject(hSem1, INFINITE);  //信号量 hSem1 的减一操作。
37           bDone=(GetNextLine(fd, inLine)==EOF);  //调用 GetNextLine()函数读取 fd 指针指
           向的当前行的内容,互斥 fd 指针。
38      ReleaseSemaphore(hSem1, 1, NULL);   //信号量 hSem1 的加一操作。
39      if(! bDone){
40           lCount=GetWordAndLetterCount(inLine); //计算 inLine 指向字符串中单词的个数。
41           WaitForSingleObject(hSem2, INFINITE); //信号量 hSem2 的减一操作。
42             TotalWords+=lCount;             //将 lCount 的值累加到 TotalWords 变量中,互斥
             TotalWords 变量。
43           ReleaseSemaphore(hSem2, 1, NULL);    //信号量 hSem2 的加一操作。
44         }
45   }
46  return 0;
```

东软电子出版社

```
47 }
48 int _tmain(int argc, _TCHAR * argv[])
49 {
50   HANDLE hThread[NUMTHREADS];
51   hSem1=CreateSemaphore(NULL, 1, 1, NULL);  // 创建信号量 hSem1。
52   hSem2=CreateSemaphore(NULL, 1, 1, NULL);  // 创建信号量 hSem2。
53   fopen_s(&fd, "InFile1.txt", "r");          //以只读的方式打开文件 InFile1.txt。
54   for(int i=0; i < NUMTHREADS; i++)       //循环生成四个子线程,并行 CountWords 函数。
55       hThread[i]=CreateThread(NULL,0,CountWords,NULL,0,NULL);
56   WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);  //同步等待子线程
57   fclose(fd);
58   printf("Total Words= %8d\n\n", TotalWords);
59   return 0;
60 }
```

代码 2-6 用于计算文件 InFile1.txt 中单词的个数。代码中应用了两个信号量 hSem1 和 hSem2。hSem1 用于线程对文件指针 fd 的互斥,hSem2 用于对全局变量 TotalWords 的互斥。第 36 行代码和第 41 行代码是对信号量执行减 1 的操作,第 38 行代码和第 43 行代码是对信号量执行加 1 的操作,而第 56 行代码实现了对四个线程的同步等待。

东软电子出版社

# 2.4　线程池

在某些应用程序中,为执行某些任务,开发人员可能需要动态地分配一些线程,这些线程在数量上完全不受开发人员的控制,可能非常多,也可能仅需要几个,各种情况的差别可能很大。例如,一个 Web 服务器应用程序,有时会因为服务器空闲而不需要做任何处理工作,有时却需要在给定的时间内处理多达数千个请求。从软件角度处理这种情况的一个方法就是采用动态线程创建技术。当系统开始不断接收越来越多的工作时,开发人员需要不断创建新的线程来处理输入的请求。而当系统变得空闲时,由于没有足够多的工作需要完成,并且线程所占用的系统资源也非常宝贵,所以开发人员可能需要终止那些在系统中处于最大负载状况时所创建的一些线程。

动态线程创建技术中存在着一些问题。首先,线程的创建是一个开销很大的操作。当处于峰值流量时,Web 服务器在线程创建上所花费的时间可能比实际响应用户请求的时间还要多。为了解决这个问题,操作系统提供了线程池技术用于对一组线程进行管理。开发人员可以在应用程序启动时就创建一组线程,当有处理请求来到时,这些线程已经做好了准备,这样就可以解决线程创建所带来的开销问题了。线程池中的线程由系统管理,程序员不需要费力于线程管理,可以集中精力处理应用程序任务。

从 Windows 2000 起,Microsoft 开始提供线程池 API,该 API 可以极大地减少开发人员实现线程池所需完成的代码量。使用线程池最重要的函数是 QueueUserWorkItem()。

```
BOOL QueueUserWorkItem(
LPTHREAD_START_ROUTINE Function,
```

```
PVOID Context,
ULONG Flags);
```

Function 参数是一个函数指针，指向线程池中的线程必须要完成的工作。该函数必须具有以下形式：

```
DWORD WINAPI Function(LPVOID parameter);
```

读者可以发现，这个函数与创建线程时的线程函数形式相同。这个函数的返回值就是线程的退出代码，该退出代码可以通过调用 GetExitCodeThread()获得。Context 参数是一个 void 指针，与传递给线程函数的 pvParam 参数功能相同。Flags 参数将在后面进行介绍。

当第一次调用 QueueUserWorkItem()时，Windows 将创建一个线程池，其中的一个线程将执行 Function 函数，函数执行完成后，该线程返回线程池，等待新的任务。由于 Windows 依赖于该过程来完成线程池的功能，因此 Function 中不能有任何终止该线程的调用，如 ExitThread()函数。

当调用 QueueUserWorkItem()时，如果没有可用的线程，Windows 就可以通过创建额外的线程增加线程池中线程的数量。线程池中线程的数量是动态的，并且受 Windows 的控制，Windows 内部的调度算法决定处理当前线程工作负载的最佳方式。

如果知道所要处理的工作需要很长时间才能完成，可以在调用 QueueUserWorkItem()时，将参数 Flags 设置为 WT_EXECUTELONGFUNCTION。这时如果线程池中所有的线程都处于忙状态，那么 Windows 将自动创建新的线程。

Windows 线程池中的线程有两种类型：一种可以用来处理异步 I/O，另一种则不能。前者依赖于 I/O 完成端口。I/O 完成端口（I/O completion port）是一种 Windows 内核对象，它可以将线程和 I/O 端口绑定在特定的系统资源上。对带有完成端口的 I/O 进行处理是一个复杂的过程，在此不多介绍。调用 QueueUserWorkItem()时，需要标识哪些线程执行 I/O，哪些线程不执行 I/O。将 QueueUserWorkItem()中的 Flags 设置成 WT_EXECUTIONDEFAULT，就可以告知线程池该线程不执行异步 I/O，从而可以对其进行相应的管理。对于执行异步 I/O 的线程，则应该将其 Flags 设置为 WT_EXECUTEINIOTHREAD。

当使用多个线程对目标工作进行功能分解时，也可以考虑使用线程池 API 来减轻程序设计的负担，使 Windows 有机会协助进行线程的管理，从而使应用程序能够达到最佳的性能。

# 2.5　生产者-消费者问题的实现

在实现过程中采用信号量机制解决线程同步和互斥的代码，如代码 2-7 所示。

**代码 2-7**

```
1  ＃include "stdafx.h"
2  ＃include <windows.h>
3  ＃define AREASIZE 10
4  ＃define NUMTHREADS 2
5  HANDLE hFULL, hEMPTY;//同步信号量。
6  HANDLE Mutex;  //互斥信号量。
```

东软电子出版社

```
7  int Area[AREASIZE];
8  int Pcur,Ccur;
9  DWORD WINAPI Producer(LPVOID arg) {
10  while(1) {
11    WaitForSingleObject(hEMPTY, INFINITE);//生产者判断是否有空闲的存储空间。
12    WaitForSingleObject(Mutex,INFINITE);     //互斥存储空间。
13    printf("Producer do %d  \n",Pcur);
14    Area[Pcur++]=1;//生产产品
15    if(Pcur>=AREASIZE-1){
16      ReleaseSemaphore(Mutex, 1, NULL);    //释放存储空间。
17      ReleaseSemaphore(hFULL, 1, NULL);   //增加产品数量。
18       break;
19     }
20     ReleaseSemaphore(Mutex, 1, NULL);        //释放存储空间。
21     ReleaseSemaphore(hFULL, 1, NULL);       //增加产品数量。
22  }
23  return 0;
24  }
25  DWORD WINAPI Consumer(LPVOID arg) {
26  while(1)
27  {
28     WaitForSingleObject(hFULL, INFINITE);      //消费者判断是否有产品。
29       WaitForSingleObject(Mutex,INFINITE);        //互斥存储空间。
30         printf("***Consumer do %d  \n",Ccur);
31         Area[Ccur++]=2;//消费产品。
32         if(Ccur>=AREASIZE-1) {
33           ReleaseSemaphore(Mutex, 1, NULL);      //释放存储空间。
34           ReleaseSemaphore(hEMPTY, 1, NULL);   //增加空闲存储空间的数量。
35           break;
36         }
37       ReleaseSemaphore(Mutex, 1, NULL);           //释放存储空间。
38       ReleaseSemaphore(hEMPTY, 1, NULL);        //增加空闲存储空间的数量。
39  }
40  return 0;
41  }
42  int _tmain(int argc, _TCHAR* argv[]){
43  int i;
44  HANDLE hThread[NUMTHREADS];
45  Pcur=0;Ccur=0;
46   for(i=0;i<AREASIZE;i++)//初始化共享存储空间。
47    Area[i]=0;
48   hEMPTY=CreateSemaphore(NULL, AREASIZE,AREASIZE , NULL);  // 初始化消费者资源信号量。
```

东软电子出版社

```
49   hFULL=CreateSemaphore(NULL, 0, AREASIZE, NULL);  // 初始化生产者资源信号量。
50   Mutex=CreateSemaphore(NULL, 1, 1, NULL);//初始化互斥信号量。
51   hThread[0]=CreateThread(NULL,0,Producer,NULL,0,NULL);//生产者。
52   hThread[1]=CreateThread(NULL,0,Consumer,NULL,0,NULL); //消费者。
53   WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);//栅栏同步。
54   return 0;
55  }
```

代码 2-7 中，设置了三个信号量 Mutex，hFULL 和 hEMPTY，设置的有限的存储空间是10。Mutex 用来避免生产者和消费者对同一存储空间的同时操作，是互斥信号量；hEMPTY 信号量用来保证生产者在具备空闲存储空间的前提下才生产产品；hFULL 信号量用来保证消费者在有产品的条件下才消费，他们是同步信号量。一种可能的执行结果如图 2-8 所示：

```
Producer do 0
Producer do 1
***Consumer do 0
Producer do 2
***Consumer do 1
Producer do 3
***Consumer do 2
Producer do 4
Producer do 5
Producer do 6
***Consumer do 3
Producer do 7
***Consumer do 4
Producer do 8
***Consumer do 5
***Consumer do 6
***Consumer do 7
***Consumer do 8
```

**图 2-8　代码 2-7 的执行结果**

如果采用事件和信号量相结合的方法实现，代码如 2-8 所示。

**代码 2-8**

```
1  #include "stdafx.h"
2  #include <windows.h>
3  #define AREASIZE 10
4  #define NUMTHREADS 2
5  HANDLE hFULL, hEMPTY;//同步信号量。
6  HANDLE Mutex;  //事件解决互斥。
7  int Area[AREASIZE];
8  int Pcur,Ccur;
9  DWORD WINAPI Producer(LPVOID arg) {
10  while(1) {
11     WaitForSingleObject(hEMPTY, INFINITE); //生产者判断是否有空闲的存储空间。
12     WaitForSingleObject(Mutex,INFINITE);      //互斥存储空间。
13       printf("Producer do %d  \n",Pcur);
14       Area[Pcur++]=1;//生产产品。
15       if(Pcur>=AREASIZE-1){
16         SetEvent(Mutex);                    //激发事件。
```

```
17         ReleaseSemaphore(hFULL, 1, NULL);   //增加产品数量。
18         break;
19     }
20     SetEvent(Mutex);                        //激发事件。
21     ReleaseSemaphore(hFULL, 1, NULL);       //增加产品数量。
22 }
23 return 0;
24 }
25 DWORD WINAPI Consumer(LPVOID arg) {
26 while(1)
27 {
28     WaitForSingleObject(hFULL, INFINITE);     //消费者判断是否有产品。
29     WaitForSingleObject(Mutex,INFINITE);      //互斥存储空间。
30       printf("***Consumer do %d  \n",Ccur);
31       Area[Ccur++]=2;//消费产品
32       if(Ccur>=AREASIZE-1) {
33          SetEvent(Mutex);                        //激发事件。
34          ReleaseSemaphore(hEMPTY, 1, NULL);   //增加空闲存储空间的数量。
35          break;
36       }
37       SetEvent(Mutex);                          //激发事件。
38       ReleaseSemaphore(hEMPTY, 1, NULL);       //增加空闲存储空间的数量。
39 }
40 return 0;
41 }
42 int _tmain(int argc, _TCHAR* argv[]){
43 int i;
44 HANDLE hThread[NUMTHREADS];
45 Pcur=0;Ccur=0;
46  for(i=0;i<AREASIZE;i++)//初始化共享存储空间。
47    Area[i]=0;
48  hEMPTY=CreateSemaphore(NULL, AREASIZE,AREASIZE , NULL);  // 初始化消费者资源信号量。
49  hFULL=CreateSemaphore(NULL, 0, AREASIZE, NULL);  // 初始化生产者资源信号量。
50  Mutex=CreateEvent(NULL, FALSE, TRUE, NULL);     //初始化激发态事件。
51  hThread[0]=CreateThread(NULL,0,Producer,NULL,0,NULL);     //生产者线程。
52  hThread[1]=CreateThread(NULL,0,Consumer,NULL,0,NULL);   //消费者线程。
53  WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);//栅栏同步。
54  return 0;
55 }
```

代码 2-8 中，设置了两个信号量 hFULL 和 hEMPTY，一个激发事件 Mutex。生产线程和消费线程并发时，先获得事件的线程将继续执行，后来的线程将等待事件被激发，直到获得事件的线程对临界区操作完成而再次激发事件，才会有一个等待线程可以进入临界区执行。

东软电子出版社

# 本章小结

本章系统地介绍了 Windows 线程库。详细阐述了利用 Windows 多线程 API 创建线程、管理线程以及实现线程同步的方法，并列举了大量的实例。

本章主要知识点包括：

· 内核对象的概念。

· Windows 多线程 API 的基本管理。

线程的创建：

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,DWORD dwCreationFlags,
    LPDWORD lpThreadId);
```

线程的终止：

```
VOID ExitThread(DWORD dwExitCode)
BOOL CloseHandle(HANDLE hObject)
```

线程的挂起与恢复：

```
DWORD SuspendThread(HANDLE hThread)
DWORD ResumeThread(HANDLE hThread)
```

线程间的等待：

```
DWORD WaitForSingleObject(
    HANDLE  hHandle,
    DWORD  dwMilliseconds);
DWORD WaitForMultipleObjects(
    DWORD nCount,
    CONST HANDLE * lpHandles,
    BOOL  fWaitAll,
    DWORD dwMilliseconds);
```

· 线程间的同步。

与临界区相关的 API：

```
void WINAPI InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

与事件相关的 API：

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL    bManualReset,
    BOOL    bInitialState,
```

东软电子出版社

```
    LPCSTR lpName);
DWORD WaitForSingleObject(
    HANDLE  hHandle,
    DWORD  dwMilliseconds);
DWORD WaitForMultipleObjects(
    DWORD nCount, CONST HANDLE * lpHandles,
    BOOL  fWaitAll, DWORD dwMilliseconds);
BOOL SetEvent(HANDLE event);
BOOL ResetEvent(HANDLE event);
```

与信号量相关的 API：

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    LONG      lSemInitial,
    LONG      lSemMax,
    LPCSTR   lpSemName);
DWORD WaitForSingleObject(
    HANDLE  hHandle,
    DWORD  dwMilliseconds);
DWORD BOOL  ReleaseSemaphore(
    HANDLE   hSemaphore,
    LONG       cReleaseCount,
    LPLONG   lpPreviousCount);
```

- 线程池的概念。

# 习题

(1)下面的代码是否可以实现代码 2-1 的功能？为什么？

```
#include "stdafx.h"
#include <windows.h>
const int numThreads=4;
DWORD WINAPI helloFunc(LPVOID arg) {
  int * p=(int * ) arg;
    int myNum= * p;
    printf("Hello Thread ! Thread number is %d\n", myNum);
}
int _tmain(int argc, _TCHAR * argv[])
{
   HANDLE hThread[numThreads];
    for(int i=0; i < numThreads; i++)
   hThread[i]=CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
   WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE)
}
```

东软电子出版社

(2)利用 Windows 线程库实现矩阵加法的并行程序开发。

(3)利用 Windows 线程库实现矩阵乘法的并行程序开发。

(4)根据下面的问题描述,编写多线程程序。

从一单精度浮点数串中找出乘积最大的连续的 4 个浮点数。例如在下列浮点数串[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 5.0, 3.0, 3.0, 2.0, 9.0, 1.0, 2.0]找到[4.0,5.0,6.0,5.0].

命令行:

> answer1 data1.dat

输入文件格式:

含有以二进制形式存放的浮点数的文件,浮点数数量大概在若干千万。

输出结果:

打印连续 4 个单元的首单元序号(以零为起始),以及该连续 4 个单元的乘积。

程序框架:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <alloc.h>
int main(int argc, char *argv[])
{
  float *farray, product;
  struct stat buf;
  int num,index;
/******************************/
  if(stat(argv[1],&buf))  //get the file size
  {
    printf("The file does not exist\n");
    exit(0);
  }
  num=buf.st_size/sizeof(float);
  farray=(float *)malloc(buf.st_size);
  FILE *fp=fopen(argv[1],"rb");
  fread(farray, sizeof(float), num, fp);
  fclose(fp);
/******************************/
  /*在这里处理数据*/
/******************************/
  /*输出结果*/
  printf("乘积=%f 首数的序号=%d\n", product, index);
  free(farray);
/******************************/
}
```

东软电子出版社

# 项目训练:Windows API 多线程编程

本项目分为四个模块:基础模块、临界区模块、事件模块、信号量模块。通过本项目训练逐步熟悉和掌握 Win32 API 多线程编程的语法结构、基本思路和方法。理解 API 之间的调用关系,参数的含义。

## 【学时】

8 学时

## 【目的】

掌握 Microsoft Visual Studio 2010 编写编译 Win32 API 多线程程序的方法。

掌握 Win32 API 编写多线程程序的语法。

掌握 Win32 API 编写多线程程序的思路。

掌握 Win32 多线程 API 的应用。

能解决简单的数据竞争。

## 【环境】

Windows XP 系统。

Microsoft Visual Studio 2010。

## 【相关知识】

· 内核对象的概念:由操作系统内核分配的,只能由内核访问的一个内存块,用来供系统和应用程序使用和管理的各种系统资源。

· Windows 多线程 API 的基本管理。

线程的创建:

```
HANDLE   CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE     lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId);
```

线程的终止:

```
VOID ExitThread(DWORD dwExitCode)
BOOL CloseHandle(HANDLE hObject)
```

东软电子出版社

线程的挂起与恢复：

```
DWORD SuspendThread(HANDLE hThread)
DWORD ResumeThread(HANDLE hThread)
```

线程间的等待：

```
DWORD WaitForSingleObject(
HANDLE   hHandle,
         DWORD   dwMilliseconds);
DWORD WaitForMultipleObjects(
         DWORD nCount,
         CONST HANDLE * lpHandles,
         BOOL   fWaitAll,
         DWORD dwMilliseconds);
```

·线程间的同步。

与临界区相关的 API：

```
void WINAPI InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void WINAPI DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

与事件相关的 API：

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL     bManualReset,
    BOOL     bInitialState,
    LPCSTR lpName);
DWORD WaitForSingleObject(
    HANDLE   hHandle,
    DWORD   dwMilliseconds);
DWORD WaitForMultipleObjects(
    DWORD nCount,
    CONST HANDLE * lpHandles,
    BOOL   fWaitAll,
    DWORD dwMilliseconds);
BOOL SetEvent(HANDLE event);
BOOL ResetEvent(HANDLE event);
```

与信号量相关的 API：

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    LONG      lSemInitial,
    LONG      lSemMax,
    LPCSTR   lpSemName);
DWORD WaitForSingleObject(
```

东软电子出版社

```
    HANDLE  hHandle,
    DWORD  dwMilliseconds);
DWORD BOOL  ReleaseSemaphore(
    HANDLE   hSemaphore,
    LONG       cReleaseCount,
    LPLONG   lpPreviousCount);
```

**【项目内容】**

## 模块一:基础模块

本模块通过一个简单的 Win32 API 多线程程序,使初学者明确使用 Win32 API 编写多线程程序的结构,并且掌握 Microsoft Visual Studio 2010 的基本用法。

**实验步骤:**

(1)用 Microsoft Visual Studio 2010 创建控制台项目 HelloThreads 。

(2)创建 HelloThreads. cpp 文件,内容如下:

```
#include "stdafx.h"
#include <windows.h>
const int numThreads=4;
DWORD WINAPI helloFunc(LPVOID pArg)
{

    printf("Hello Thread \n");
    return 0;
}
int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE hThread[numThreads];
     int tNum[10];
    for(int i=0; i < numThreads; i++)
    {   tNum[i]=i;
    hThread[i]=
    CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
    }
WaitForMultipleObjects(numThreads, hThread, TRUE, INFINITE);
return 0;
}
```

(3)点击“项目”→“属性”菜单,按图 2-9～图 2-12 配置项目属性。

东软电子出版社

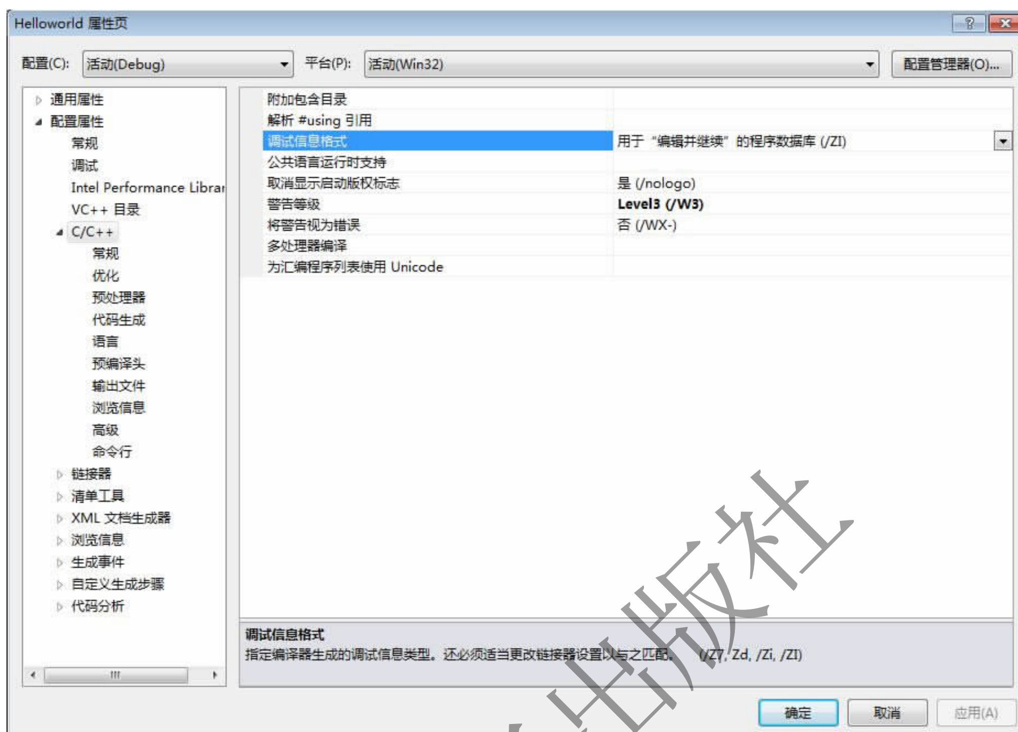



图 2-9　项目属性设置:"C/C++"→"常规"→"调试信息格式"

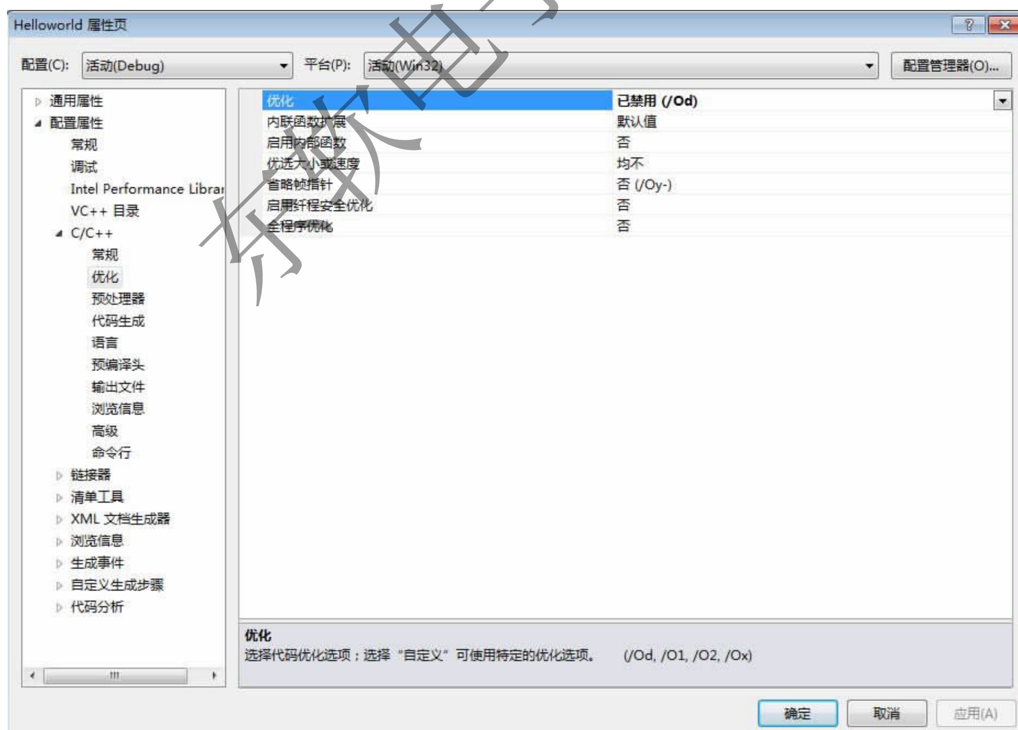



图 2-10　项目属性设置:"C/C++"→"优化"→"优化"

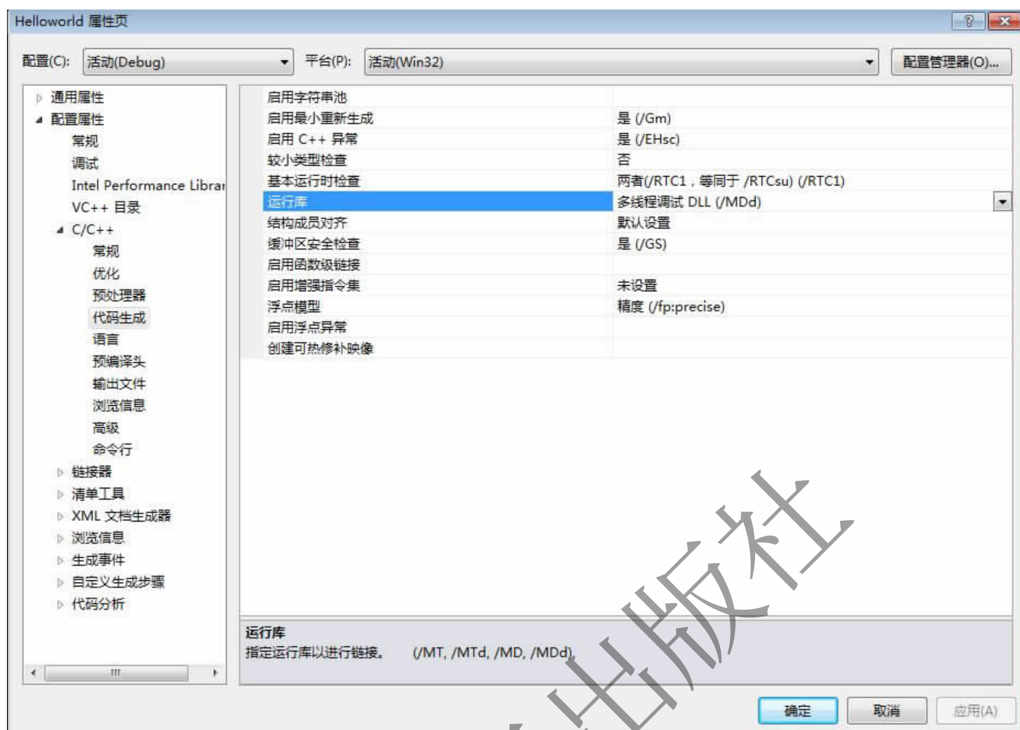



图 2-11 项目属性设置:“C/C++”→“代码生成”→“运行库”

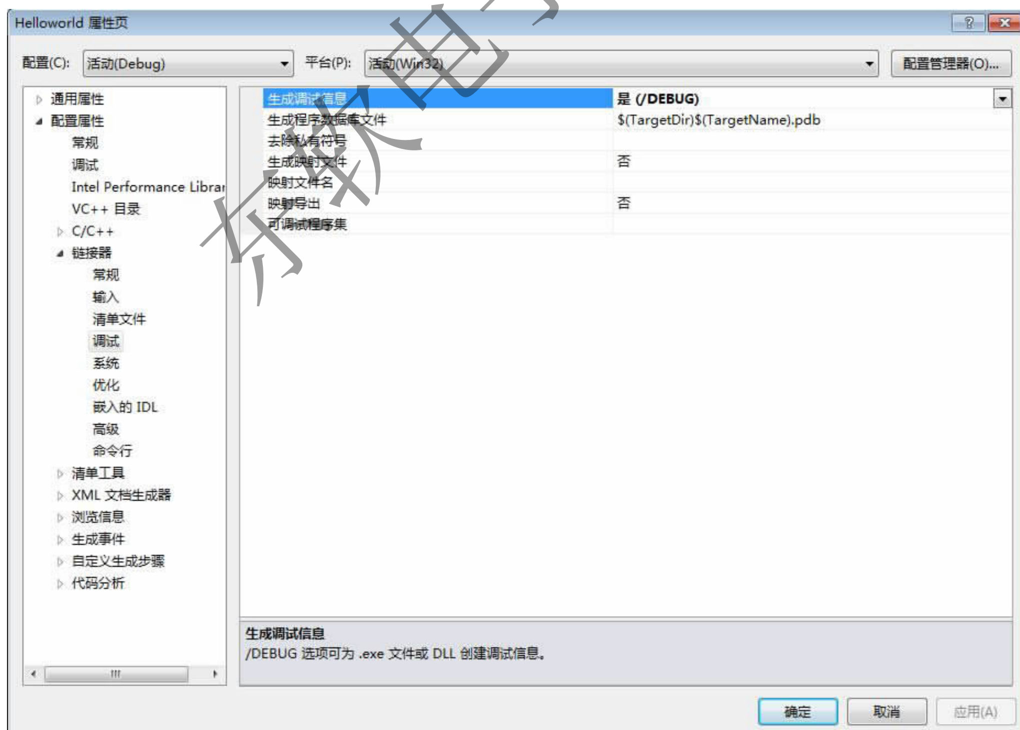



图 2-12 项目属性设置:“ 链接器”→“生成调试信息”

东软电子出版社

(4)编译执行，输出结果：________________________________________。

(5)修改代码,使之输出结果可以表示出各线程的输出顺序。

Hello Thread 0

Hello Thread 1

Hello Thread 2

Hello Thread 3

**简答与思考：**

(1)写出修改后的 HelloThreads 的代码。

(2)实验总结。

## 模块二:临界区模块

本模块将以数值积分的方法计算 Pi 的值,采用 Win32 API 来实现程序的并行化。

实验步骤：

(1)用 Microsoft Visual Studio 2010 创建控制台项目 WinPi 。

(2)创建 WinPi. cpp,内容如下：

```
#include "stdafx.h"
#include <windows.h>
#include <time.h>
static long num_steps=1000000000;
double step, pi;
int main()
{   clock_t start, stop;
   start=clock();
     int i;
   double x, sum=0.0;
   step=1.0/(double) num_steps;
   for(i=0; i< num_steps; i++){
       x=(i+0.5)*step;
       sum=sum+4.0/(1.0+x*x);
   }
   pi=step * sum;
   stop=clock();
   printf("Pi= %12.9f\n",pi);
   printf("The time of calculation was %f seconds\n",((double)(stop-start)/1000.0));
}
```

(3)编译执行,记录结果:Pi=________________________________

The time of calculationwas ____________________ seconds。

(4)将 WinPi. cpp 程序修改为 Windows Threads 并行程序。步骤如下：

①分析串行代码中的可并行的部分。

②定义线程执行的函数。

东软电子出版社

函数原型为 DWORD WINAPI 函数名(LPVOID p);

③提取可并行的代码,作为(2)中定义的函数的函数体。

④生成多个线程调用(2)中定义的函数。

⑤解决线程间的同步和互斥。

(5)采用临界区的方法进行必要的互斥。

(6)编译执行,记录结果:Pi=________________________

The time of calculationwas________________ seconds。

(7)加速比:____________ ,并行效率:____________ 。

**简答与思考:**

(1)如何进行并行化的?请写出并行化的思路与具体的代码。

(2)在本实验中,哪些变量是需要保护的?为什么?采取什么方法实现的?

(3)是否可以对该并行化方案进行优化?如何优化?

(4)实验总结。

## 模块三:事件模块

本模块利用麦凯特尔对数级数估算 ln(1+x),(−1 < x <=1)的值。线程被创建时是挂起的状态。这些线程被一个“master”线程唤醒。线程将分别执行自己的任务,"master"线程将等待所有的线程执行完毕后,汇总每一个线程的执行结果。

**实验步骤:**

(1)用 Microsoft Visual Studio 2010 创建控制台项目 ThreadEvent 。

(2)创建 ThreadEvent. cpp,内容如下:

```
#include "stdafx.h"
#include <conio.h>
#include <windows.h>
#include <math.h>
#include <time.h>
#define NUMTHREADS 4
#define SERIES_MEMBER_COUNT 100000
HANDLE *threadHandles, masterThreadHandle;
CRITICAL_SECTION countCS;
double *sums;
double x=1.0, res=0.0;
int threadCount=0;
double getMember(int n, double x)
{
    double numerator=1;
    for(int i=0; i<n; i++)
        numerator=numerator*x;
    if(n%2==0)
        return(-numerator / n);
```

东软电子出版社

```
    else
        return numerator/n;
}
DWORD WINAPI threadProc(LPVOID par)
{
        int threadIndex= *((int *)par);
        sums[threadIndex]=0;
        for(int i=threadIndex; i<SERIES_MEMBER_COUNT;i+=NUMTHREADS)
            sums[threadIndex]+=getMember(i+1, x);
        EnterCriticalSection(&countCS);
          threadCount++;
        LeaveCriticalSection(&countCS);
        delete par;
        return 0;
}
DWORD WINAPI masterThreadProc(LPVOID par)
{
    for(int i=0; i<NUMTHREADS; i++) ResumeThread(threadHandles[i]);  // Start computing threads
    while(threadCount != NUMTHREADS){}   // busy wait until all threads are done with computation of
    partial sums
    res=0;
    for(int i=0; i<NUMTHREADS; i++)
        res+=sums[i];
    return 0;
}
int _tmain(int argc, _TCHAR* argv[])

    {   clock_t start, stop;
    threadHandles=new HANDLE[NUMTHREADS+1];
    InitializeCriticalSection(&countCS);
    sums=new double[NUMTHREADS];
     start=clock();
    for(int i=0; i<NUMTHREADS;i++)
    {
        int * threadIdPtr=new int;
         *threadIdPtr=i;
        threadHandles[i]=CreateThread(NULL, 0, threadProc, threadIdPtr, CREATE_SUSPENDED, NULL);
    }
    threadHandles[NUMTHREADS]=CreateThread(NULL, 0, masterThreadProc, NULL, 0, NULL);
    printf("Count of ln(1+x) Mercator's series members is %d\n",SERIES_MEMBER_COUNT);
    printf("Argument value of x is %f\n",(double)x);
```

东软电子出版社

```
    WaitForMultipleObjects(NUMTHREADS+1,threadHandles,TRUE,INFINITE);
        stop=clock();
    for(int i=0; i<NUMTHREADS+1; i++) CloseHandle(threadHandles[i]);
    delete threadHandles;
    DeleteCriticalSection(&countCS);
    delete sums;

    printf("Result is %10.8f\n", res);
    printf("By function call ln(1+%f)=%10.8f\n",x, log(1+x));
    printf("The time of calculation  was %f seconds\n",((double)(stop-start)/1000.0));
    printf("Press any key … ");
    getch();
    return 0;
}
```

(3)编译执行,记录结果:Result is ________________

The time of calculation was ________________ seconds。

(4)阅读代码,回答下面问题。

①主线程共创建________________个子线程。

②各子线程调用的函数各是什么?

③主线程等待________________个子线程的执行结束。

(5)请分析出本代码中影响执行时间的原因,并改进。(提示:"master"线程在等待其他线程时,采用了全局变量计数的方式。请利用事件的知识改进)

(6)改进后,编译执行,记录结果:Result is ________________

The time of calculation was ________________ seconds。

(7)加速比:________________,并行效率:________________。

**简答与思考:**

(1)在 WINAPI threadProc(LPVOID par){}函数中为什么用临界区实现了线程对 threadCount 的互斥访问?为什么对于全局数据变量 sums 的访问没有互斥?

(2)简述源代码中存在的问题,详述提出的改进方案及相关代码。

(3)是否可以对该并行化方案进行优化?如何优化?

(4)实验总结。

## 模块四:信号量模块

本模块对一个串行程序并行化,以信号量的方式解决数据冲突。代码分析了指定文件中字符串的个数,含有偶数个字符的字符串的个数以及含有奇数个字符的字符串的个数。

**实验步骤:**

(1)用 Microsoft Visual Studio 2010 创建控制台项目 SemaphoreS。

(2)创建 SemaphoreS.cpp,内容如下:

```
#include "stdafx.h"
#include <windows.h>
```

东软电子出版社

```
FILE * fd;
int TotalEvenWords=0, TotalOddWords=0, TotalWords=0;
int GetNextLine(FILE * f, char * Line)
{
        if(fgets(Line, 132, f)==NULL) if(feof(f))return EOF; else return 1;
}
int GetWordAndLetterCount(char * Line)
{
    int Word_Count=0, Letter_Count=0;
    for(int i=0;i<132;i++)
    {
        if((Line[i]! =' ')&&(Line[i]! =0)&&(Line[i]! ='\n')) Letter_Count++;
        else {
          if(Letter_Count! =0){
            if(Letter_Count % 2) {
                TotalOddWords++;
                Word_Count++;
                Letter_Count=0;
            }
            else {
                TotalEvenWords++;
                Word_Count++;
                Letter_Count=0;
            }
          }
          if(Line[i]==0) break;
      }
    }
    return(Word_Count);     // encode two return values
}
DWORD WINAPI CountWords()
{
    BOOL bDone=FALSE;
    char inLine[132];
     while(! bDone)
    {
      bDone=(GetNextLine(fd, inLine)==EOF);
      if(! bDone){
           TotalWords+=GetWordAndLetterCount(inLine);
         }
   }
   return 0;
```

东软电子出版社

```
}
int main()
{
   fd=fopen("InFile1.txt", "r"); // Open file for read
   CountWords();
   fclose(fd);
     printf("Total Words= %8d\n\n", TotalWords);
     printf("Total Even Words= %7d\nTotal Odd Words= %7d\n", TotalEvenWords, TotalOddWords);
}
```

(3)这是串行代码实现，编译执行，记录结果。

Total Words：＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

Total Even Words：＿＿＿＿＿＿＿＿＿＿＿＿＿＿

Total Odd Words：＿＿＿＿＿＿＿＿＿＿＿＿＿＿

(4)用 Microsoft Visual Studio 2010 创建控制台项目 SemaphoreT。

(5)创建 SemaphoreT.cpp，这是一个试图并行化的版本，内容如下：

```
#include "stdafx.h"
#include <windows.h>
FILE *fd;
int TotalEvenWords=0, TotalOddWords=0, TotalWords=0;
const int NUMTHREADS=4;
int GetNextLine(FILE *f, char *Line)
{
     if(fgets(Line, 132, f)==NULL) if(feof(f))return EOF; else return 1;
}
int GetWordAndLetterCount(char *Line)
{
     int Word_Count=0, Letter_Count=0;
     for(int i=0;i<132;i++)
     {
          if((Line[i]!=' ')&&(Line[i]!=0)&&(Line[i]!='\n')) Letter_Count++;
          else {
               if(Letter_Count!=0){
                 if(Letter_Count%2){
                    TotalOddWords++;
                    Word_Count++;
                    Letter_Count=0;
                }
               else {
                    TotalEvenWords++;
                    Word_Count++;
                    Letter_Count=0;
               }
```

东软电子出版社

```
            }
              if(Line[i]==0) break;
          }
      }
      return(Word_Count);
}
DWORD WINAPI CountWords(LPVOID arg)
{
      BOOL bDone=FALSE;
      char inLine[132];
      while(! bDone)
      {
        bDone=(GetNextLine(fd, inLine)==EOF);
      if(! bDone){
              TotalWords+=GetWordAndLetterCount(inLine);
          }
      }
      return 0;
}
int main()
{
      HANDLE hThread[NUMTHREADS];
      fd=fopen("InFile1.txt", "r"); // Open file for read
      for(int i=0; i < NUMTHREADS; i++)
          hThread[i]=CreateThread(NULL,0,CountWords,NULL,0,NULL);
      WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);
      fclose(fd);
      printf("Total Words= %8d\n\n", TotalWords);
      printf("Total Even Words= %7d\nTotal Odd Words= %7d\n", TotalEvenWords, TotalOddWords);

}
```

(6)编译执行并行版本,多次运行,记录结果:

第 1 次执行结果:

Total Words: ______________________________

Total Even Words: ______________________________

Total Odd Words: ______________________________

第 2 次执行结果:

Total Words: ______________________________

Total Even Words: ______________________________

Total Odd Words: ______________________________

第 3 次执行结果:

东软电子出版社

Total Words：________________

Total Even Words：________________

Total Odd Words：________________

(7)观察串行和并行输出结果，分析输出结果不一致的原因。

(8)修改 SemaphoreT. cpp 代码中的错误，使之输出正确的结果，完成程序的并行化。

(9)在并行过程中请采用信号量的手段解决数据竞争问题。

(10)修正后项目的输出结果为：

Total Words：________________

Total Even Words：________________

Total Odd Words：________________

**简答与思考：**

(1)SemaphoreS 项目与 SemaphoreT 项目执行结果不一致的原因是什么？

(2)如何修改 SemaphoreT 项目源代码？写出修改思路和关键代码。

(3)是否还有更优的修改方案？如何修改？

(4)实验总结。

东软电子出版社