第5章 Visual C++中的多线程编程

本章主要介绍 Visual C++中的多线程编程技术以及多线程技术在 WinSock 编程中的应用。主要内容包括进程和线程的相关概念、Visual C++中的多线程技术、TCP 服务器端程序的多线程编程、线程间的通信、线程的同步与互斥等。

5.1 进程和线程的概念

进程是现代操作系统理论的核心概念之一。支持多任务并发执行的操作系统需要为多个并发执行的程序合理地分配内存、外设、CPU 时间等资源，为了便于描述和实现系统中各程序运行过程的独立性、并发性、动态性以及它们相互之间因资源共享而引起的相互制约性，操作系统引入了进程(Process)的概念。

进程是指具有一定独立功能的程序在某个数据集合上的一次运行活动，是系统进行资源分配和调度运行的一个独立单位，每个进程都有自己的独立的内存地址空间。

进程是程序在计算机上的一次执行活动，当你启动了一个程序，你就启动了一个进程，退出一个程序也就结束了一个进程。但需要明确，程序并不等价于进程，程序只是一组指令的有序集合，是一个静态实体，而进程是程序在某个数据集上的执行，是一个动态实体。

程序只有被装入内存后才能运行，程序一旦进入到内存就成为进程了，因此，进程的创建过程也就是程序由外存储器被加载到内存的过程。

进程在其存在过程中，由于多个进程的并发执行，受到 CPU、外部设备等资源的制约，使得它们的状态不断发生变化。进程的基本状态有以下三种。

(1) 就绪状态：进程获得了除 CPU 之外的一切所需资源，一旦获得 CPU 即可运行。

(2) 运行状态：进程获得了 CPU 等一切所需资源，正在 CPU 上运行。

(3) 阻塞状态：正在 CPU 上运行的进程，由于某种原因不再具备运行的条件而暂时停止运行，比如需要等待 I/O 操作完成、等待其他进程发来消息等。

当就绪进程的数目多于 CPU 的数目时，需要按一定的算法动态地将 CPU 分配给就绪进程队列中的某一个使之运行，这就是所谓的进程调度。当分配给某个进程的运行时间(时间片)用完了时进程就会由运行状态回到就绪状态；运行中的进程如果需要执行 I/O 操作，比如从键盘输入数据，就会进入到阻塞状态等待 I/O 操作完成，I/O 操作完成后，就会转入

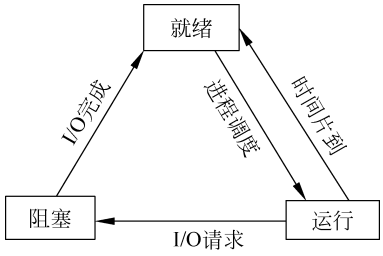

图 5.1　进程的三种状态及相互转换

就绪状态等待下一次调度，如图 5.1 所示。

进程因创建而产生，因调度而运行，因等待资源或事件而被处于等待状态，因完成任务而被撤销，它反映了一个程序在一定的数据集上运行的全部动态过程。

线程是为了在进程内部实现并发性而引入的概念。进程内部的并发性是指在同一个进程内部可以同时进行多项工作，而线程就是完成其中某一项工作的单一指令序列。一般情况下，同一进程中的多个线程各自完成不同的工作，比如一个线程负责通过网络收发数据，另一个线程完成所需的计算工作，第三个线程来执行文件输入输出，当其中一个由于某种原因阻塞后，比如通过网络收发数据的线程等待对方发送数据，另外的线程仍然能执行而不会被阻塞。

一个进程内的所有线程则是在同一进程的地址空间运行的。各线程自己并不独自拥有系统资源，它与同属一个进程的其他线程共享进程的地址空间等全部资源，解决同一进程的各线程之间如何共享内存、如何通信等问题是多线程编程中的难点。

由于线程之间的相互制约，以及程序功能的需求，线程在运行中也会呈现出间断性，因此一个线程在其生命期内有两种存在状态——运行状态和阻塞(也称挂起)状态。有很多原因可导致线程在这两种状态之间进行切换。图 5.2 给出了线程的状态及相互转换的原因。

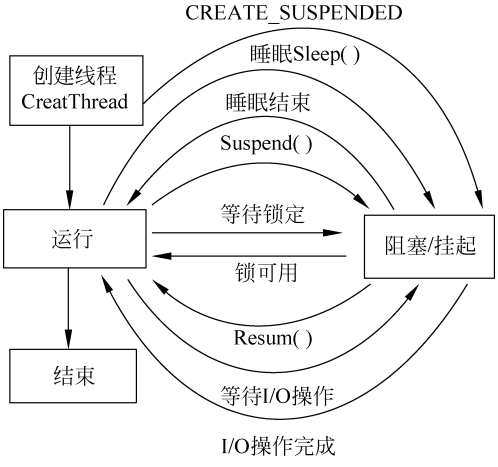

图 5.2　线程的状态及其转换

线程仅简单地借用了进程切换的概念，它把进程间的切换转变成了同一个进程内的几个函数间的切换。同一个进程中函数间的切换相对于进程切换来说所需的开销要小得多，它只需要保存少数几个寄存器、一个堆栈指针以及程序计数器等少量内容。在进程内创建、终止线程比操作系统创建、终止进程要快。由于一个进程中的所有线程都在该进程的地址空间中，共同使用地址空间中的全局变量和系统资源，所以线程间的通信非常方便。

有多个线程的程序称为多线程程序。Windows 系统支持多线程程序，允许程序中存在多个线程。事实上，任何一个 Windows 中的进程都至少有一个线程，即主线程，它不是由用户主动创建的，而是由系统自动创建的，其他线程都是用户根据需要在主线程或其他线程中创建的，称为子孙线程。进程创建完成后就已启动了该进程的主线程。在 Visual C++程序中，主线程的启动点是以函数形式(即 main 或 WinMain 函数)提供给 Windows 系统的。主线程终止了，进程也将随之终止，而不管其他线程是否执行完毕。

5.2　Visual C++中的多线程编程

多线程给应用开发带来了许多好处，但并非任何情况下都要使用多线程，一定要根据应用程序的具体情况来综合考虑。一般来说，在以下情况下可以考虑使用多线程。

(1) 应用程序中的各任务相对独立;

(2) 某些任务耗时较多;

(3) 各任务需要有不同的优先级。

在 Visual C++程序设计中,有多种方法在程序中实现多线程:使用 Win32 SDK 函数、使用 C/C++运行库函数、使用 MFC 类库。由于使用 MFC 类库实现多线程的技术涉及的概念较多,较为复杂,篇幅所限本书暂不介绍。

5.2.1 使用 Win32 SDK 函数实现多线程

1. 创建线程

在程序中创建一个线程需要以下两个步骤。

1) 编写线程函数

所有线程必须从一个指定的函数开始执行,该函数就是所谓的线程函数。线程函数必须具有类似下面所示的函数原型。

```
DWORD _stdcall ThreadFunc( LPVOID lpvThreadParm);
```

关键字_stdcall 是函数调用规范的一种。函数调用规范主要规定了被调函数的参数传递顺序、调用堆栈是由调用函数还是被调用函数清理等。_stdcall 规定参数从右向左压入堆栈,由被调函数清理堆栈。另外一种常见的函数调用约定是_cdecl,它是 C/C++的默认调用规范,_cdecl 规定参数按从右至左的顺序压参数入栈,由调用者负责清理堆栈。

ThreadFunc 是线程函数的名字,可以由编程者任意指定,但必须符合 Visual C++标识符的命名规范。该函数仅有一个 LPVOID 型的参数,LPVOID 的类型定义如下。

```
typedef void * LPVOID;
```

它既可以是一个 DWORD 型的整数,也可以是一个指向一个缓冲区的 void 类型指针。函数返回一个 DWORD 型的值。

一般来说,C++的类成员函数不能作为线程函数。这是因为在类中定义的成员函数,编译器会给其加上 this 指针。但如果需要线程函数像类的成员函数那样能访问类的所有成员,可采用两种方法:第一种方法是将该成员函数声明为 static 类型,但 static 成员函数只能访问 static 成员,不能访问类中的非静态成员,解决此问题的一种途径是可以在调用类静态成员函数(线程函数)时将 this 指针作为参数传入,并在该线程函数中用强制类型转换将 this 转换成指向该类的指针,通过该指针访问非静态成员。第二种是不定义类成员函数为线程函数,而将线程函数定义为类的友元函数,这样线程函数也可以有类成员函数同等的权限。

2) 创建一个线程

进程的主线程是操作系统在创建进程时自动生成的,但如果要让一个线程创建一个新的线程,则必须调用线程创建函数。Win32 SDK 提供的线程创建函数是 CreateThread()。

函数原型

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

```
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
    );
```

函数参数

- lpThreadAttributes：指向一个 LPSECURITY_ATTRIBUTES 结构的指针，该结构决定了线程的安全属性，一般置为 NULL。
- dwStackSize：指定线程的堆栈深度，一般设置为 0。
- lpStartAddress：线程起始地址，通常为线程函数名。

LPTHREAD_START_ROUTINE 类型定义：

```
typedef unsigned long(_stdcall * LPTHREAD_START_ROUTINE)(void * lpParameter);
```

- lpParameter：线程函数的参数。
- dwCreationFlags：控制线程创建的附加标志。该参数为 0，则线程在被创建后立即开始执行；如果该参数为 CREATE_SUSPENDED，则创建线程后该线程处于挂起状态，直至函数 ResumeThread 被调用。
- lpThreadId：该参数返回所创建线程的 ID。

返回值

该函数在其调用进程的进程空间里创建一个新的线程，并返回已创建线程的句柄，如果创建成功，则返回线程的句柄，否则返回 NULL。

注意，使用同一个线程函数可以创建多个各自独立工作的线程。

例 5.1 一个简单的线程函数定义及线程创建的例子。

```
#include "windows.h"
#include "iostream"
using namespace std;
//定义线程函数,这里的形参 p 是为了满足线程函数的格式要求,函数内并没使用
DWORD _stdcall ThreadFun1( LPVOID p)
{
    for(int i = 1;i < 100;i++)
    {
        Sleep(1000);                          //阻塞 1000ms
        cout << i << ",This is Thread 1\n";
    }
    return 0;
}
HANDLE hThread1;                              //线程句柄
DWORD ThreadID1;                              //线程 ID
int main()
{
    //创建线程,线程函数的参数在函数内部并没用到,所以 CreateThread 的第四个参数为 NULL
    hThread1 = CreateThread(NULL,0,ThreadFun1, NULL, 0, &ThreadID1);
    for(int j = 1;j < 10;j++)
```

```
    {
        Sleep(1000);
        cout << j <<",This is MainThread!\n";
    }
    return 0;
}
```

程序中的 Sleep()函数是一个 Windows API 函数,其功能是使线程阻塞,直到指定时间过完。使用本函数需要包含头文件“windows.h”。

函数原型

```
VOID Sleep(DWORD dwMilliseconds);
```

函数参数

dwMilliseconds:指定线程阻塞的时间长度,时间的单位是毫秒(ms)。如果参数取值为 0,执行该函数也将使线程阻塞转而执行其他同优先级的线程,如果不存在其他同优先级的线程,线程将立刻恢复执行。如果取值为常量 INFINITE,则线程将被无限期阻塞。

2. 线程函数的参数传递

由 CreateThread 函数原型可以看出,创建线程时可以给线程传递一个 void 指针类型的参数,该参数为 CreateThread()函数的第四个参数。

当需要将一个整型数据作为线程函数的参数传递给线程时,可将该整型数据强制转换为 LPVOID 类型,作为其实参传递给线程函数。

当需要向线程传递一个字符串时,则创建线程时的实参传递既可以使用字符数组,也可以使用 CString 类。使用字符数组时,实参可直接使用字符数组名或指向字符数组的 char * 指针;使用 CString 类时,可将指向 CString 对象的指针强制转换为 LPVOID 类型。

如果需要向线程传送多个数值时,由于线程函数的参数只有一个,所以需要先将它们封装在一个结构体变量中,然后将该变量的指针作为参数传给线程函数。

例 5.2 将整数、字符数组和 CString 对象作为参数传递给线程。

注意,下面的程序运行时应将“项目属性”中的“字符集”设为多字节字符集,否则用 cout 不能正确输出 CString 对象的值。

```
#include "iostream"
#include "windows.h"
#include "atlstr.h"
using namespace std;
DWORD _stdcall ThreadF0(LPVOID lpParam)
{
    int a = (int)lpParam;                          //将传入的参数值强制转换为整数
    Sleep(100);
    cout << "I am Thread0,the number main thread given me is" << a << endl;
    return 0;
}
DWORD _stdcall ThreadF1(LPVOID lpParam)
{
    char * p = (char * )lpParam;                   //获取传入的字符数组指针
```

```
    Sleep(500);
    cout << "This is Thread1,the string main thread given me is: " << p << endl;
    return 0;
}
DWORD _stdcall ThreadF2(LPVOID lpParam)
{
    CString * p = (CString * )lpParam;          //获取传入的 CString 对象的指针
    Sleep(1000);
    cout <<"This is Thread2,the string main thread given me is:"<< * p << endl;
    return 0;
}
int main()
{
    HANDLE hThrd0,hThrd1, hThrd2;               //定义线程句柄变量
    DWORD ThrdID0,ThrdID1, ThrdID2;
    int a = 888;                                //传递给线程 0 的整数
    char s1[] = "ABCDEFGH";                     //传递给线程 1 的字符数组
    CString s2("abcdef");                       //传递给线程 2 的字符串对象
    cout << "This is MainThread!"<< s2 << endl;
    hThrd0 = CreateThread(NULL, 0, ThreadF0, (void * )a, 0, &ThrdID0);
                                                //将 int 型数据强制转换为(void * )作为参数
    hThrd1 = CreateThread(NULL, 0, ThreadF1, (void * )s1, 0, &ThrdID1);
                                                //将字符数组地址强制转换为(void * )作为参数
    hThrd2 = CreateThread(NULL, 0, ThreadF2, (void * )&s2, 0, &ThrdID2);
                                                //将 CString 对象地址强制转换为(void * )作为参数
    Sleep(2000);
    cout << "MainThread Exit!\n";
}
```

3. 线程的挂起与恢复

在创建线程时,如果 CreateThread()函数的第五个参数,用于控制线程创建的附加标志设置为 CREATE_SUSPENDED,则线程创建后将处于挂起状态。另外,线程自身也可以通过调用 SuspendThread()函数使自身进入到挂起状态。

函数原型

```
DWORD SuspendThread(HANDLE hThread);
```

函数参数

hThread:要挂起的线程的句柄,该句柄是创建线程时由 CreateThread()函数返回的。该函数用于挂起指定的线程,对于没有被挂起的线程,程序员可以调用 SuspendThread()函数强行挂起它,如果函数执行成功,则线程的执行被终止。

返回值

函数调用成功,返回函数此前被挂起的一个计数,调用失败返回(DWORD)-1。需要注意,一个已被挂起的线程也可以再次调用本函数挂起,每调用一次,其挂起计数就增加 1,正运行的线程挂起计数为 0。

一创建就进入到挂起状态或调用 SuspendThread 进入到挂起状态的线程,可以被其他

线程通过调用 ResumeThread()函数恢复运行。ResumeThread()函数的功能是使处于挂起状态的进程恢复运行。

函数原型

```
DWORD ResumeThread(HANDLE hThread);
```

函数参数

hThread：将要恢复运行的线程的句柄。

返回值

执行成功函数将返回线程的挂起计数，否则返回(DWORD) －1。

线程可以自行调用 SuspendThread()进入到挂起状态，但是不能自行恢复运行，必须由其他线程通过调用 ResumeThread()函数恢复运行。一个线程可以被挂起多次，如果一个线程被挂起 n 次，则该线程也必须被恢复 n 次才可能得以执行，这里的 n 就是线程的挂起计数。

4. 终止线程

一般情况下，线程函数执行完毕正常返回后线程也就结束了。但是，应用程序的其他线程可以调用 TerminateThread()函数来强行终止某一未结束的线程。

函数原型

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

函数参数

- hThread：将被终结的线程的句柄。
- dwExitCode：用于指定线程的退出码。线程退出码可用 GetExitCodeThread()获得。

返回值

函数执行成功则返回 TRUE，执行失败返回 FALSE。

GetExitCodeThread()函数获取一个已终止线程的退出代码。

函数原型

```
BOOL WINAPI GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

函数参数

- hThread：想获取退出代码的一个线程的句柄。
- lpExitCode：指向一个用于装载线程退出代码的长整数变量。如线程尚未中断，则设为常数 STILL_ACTIVE。

返回值

如果执行成功，返回 TRUE，退出码被 lpExitCode 指向内存记录；否则返回 FALSE，可通过 GetLastError()获知错误原因。如果线程尚未结束，lpExitCode 带回来的将是 STILL_ALIVE。

使用 TerminateThread()终止某个线程的执行是不安全的，可能会引起系统不稳定；虽然该函数立即终止线程的执行，但并不释放线程所占用的资源。除了 TerminateThread()可以

终结一个线程外，Win32 API 提供的 ExitThread()函数也可以用来终止一个线程，只不过该函数用于线程终结自身，即调用该函数的线程将被结束。

函数原型

```
VOID ExitThread(DWORD dwExitCode);
```

函数参数

dwExitCode：用来设置线程的退出码。线程的退出码可调用 GetExitCodeThread()函数获得。

例 5.3 使用 CreateThread 创建两个线程，在这两个线程中 Sleep 一段时间，主线程通过 GetExitCodeThread()来判断两个线程是否结束运行。程序代码如下。

```
#include "iostream"
#include <windows.h>
using namespace std;
//线程函数
DWORD _stdcall ThreadFunc(LPVOID n)
{
    int m = (DWORD)n;
    Sleep(10 * (5 - m));
    return m * 10;
}
int main()
{
    HANDLE hThread1, hThread2;
    DWORD exitCode1 = 0, exitCode2 = 0;
    DWORD ThreadId1, ThreadId2;
    hThread1 = CreateThread(NULL, 0, ThreadFunc, (LPVOID)1, 0, &ThreadId1);
    if (hThread1) cout << "Thread 1 launched\n";
    hThread2 = CreateThread(NULL, 0, ThreadFunc, (LPVOID)2,
        CREATE_SUSPENDED, &ThreadId2);          //线程创建后进入挂起状态
    if (hThread2)
    {
        ResumeThread(hThread2);                 //使线程进入运行状态
        cout << "Thread 2 launched\n";
    }
    //检测两个线程是否结束,并输出结束码
    for (;;)
    {
        GetExitCodeThread(hThread1, &exitCode1);
        GetExitCodeThread(hThread2, &exitCode2);
        if (exitCode1 == STILL_ACTIVE)
            cout << "Thread 1 is still running!" << endl;
        else
            cout << "线程 1 的退出码为:" << exitCode1 << endl;
        if (exitCode2 == STILL_ACTIVE)
            cout << "Thread 2 is still running!" << endl;
        else
            cout << "线程 2 的退出码为:" << exitCode2 << endl;
```

```
            if (exitCode1 != STILL_ACTIVE && exitCode2 != STILL_ACTIVE)
                break;
        }
        return EXIT_SUCCESS;
    }
```

5.2.2 C++运行库中的多线程函数

标准C运行时库是1970年问世的,当时还没有多线程的概念。因此,C运行时库早期的设计者们不可能考虑到让其支持多线程应用程序。

Visual C++提供了两种版本的C运行时库:一个版本供单线程应用程序调用,另一个版本供多线程应用程序调用。多线程运行时库与单线程运行时库有以下两个重大差别。

(1) 类似errno的全局变量,每个线程单独设置一个,这样从每个线程中可以获取正确的错误信息。

(2) 多线程库中的数据结构以同步机制加以保护。这样可以避免访问时候的冲突。

说明:为防止和正常的返回值混淆,C/C++语言的系统调用一般不直接返回错误码,而是将错误码存入一个名为errno的全局变量。errno变量和各种错误码的定义均位于<errno.h>文件中。如果一个系统调用或者库函数调用失败,可以通过读取errno的值来确定问题所在,推测程序出错的原因。

Visual C++提供的多线程运行时库又分为静态链接库和动态链接库两类,而每一类运行时库又可再分为debug版和release版,因此Visual C++共提供了6个运行时库,参见表5.1。在这6个运行库时中,后4个运行时库均支持多线程。使用C++运行时库中的多线程函数编写多线程程序的方法与使用Windows API函数基本相同,下面仅简单给出C++创建线程和结束线程的库函数,这些库函数所在库文件为process.h。

表5.1 C++的6个运行时库

C++运行时库	库 文 件
Single thread(static link)	libc.lib
Debug single thread(static link)	Libcd.lib
MultiThread(static link)	libcmt.lib
Debug multiThread(static link)	libcmtd.lib
MultiThread(dynamic link)	msvert.lib
Debug multiThread(dynamic link)	msvertd.lib

下面给出线程创建函数_beginthread()和_beginthreadex()的函数原型。

函数原型

```
unsigned long _beginthread(
   void( _cdecl * start_address )( void * ),
   unsigned stack_size,
   void * arglist
);
unsigned long _beginthreadex(
  void * security,
```

```
    unsigned stack_size,
    unsigned ( __stdcall * start_address )( void * ),
    void * arglist,
    unsigned initflag,
    unsigned * thrdaddr
);
```

函数参数

- start_address：新线程的起始地址，指向新线程调用的函数的起始地址。
- stack_size：新线程的堆栈大小，可以为 0。
- arglist：传递给线程的参数列表，无参数时应指定为 NULL。
- security：一个指向 SECURITY_ATTRIBUTES 结构的指针，该结构用于决定函数返回的句柄能否被子线程继承，如果设为 NULL 则不能被继承。
- initflag：控制线程创建的附加标志。该参数为 0，则线程在被创建后立即开始执行；如果该参数为 CREATE_SUSPENDED，则创建线程后该线程处于挂起状态，使用 ResumeThread 可使线程运行。
- thrdaddr：指向 32 位的无符号整数的指针，用于存放线程的 ID。

返回值

成功则返回新创建线程的句柄。如果失败_beginthread 将返回－1。

需要注意，_beginthreadex()函数要求的线程函数的原型与 CreatThread()函数的相同，而_beginthread()函数的线程函数的原型则与 CreatThread()函数的不同，必须具有类似下面所示的函数原型。

```
void ThreadFunc( void * lpvThreadParm);
```

在线程内部停止_beginthread()或_beginthreadex()创建的线程，可分别使用_endthread()函数和_endthreadex()函数，这两个函数的格式如下。

```
void _endthread( void );
void _endthreadex( unsigned retval );
```

参数 retval 为线程的退出代码。

5.3 用多线程实现 TCP 并发服务器

通常服务器可同时为多客户提供服务，即可同时与多个客户机保持通信。前面已介绍的服务器端程序的编写方法并不支持这一功能。本节介绍使用多线程编程技术实现这一功能的方法。

采用多线程技术同时为多个客户提供服务的 TCP 服务器程序流程如图 5.3 所示，主线程创建套接字并启动监听，然后调用 accept() 函数接收客户请求，当有客户请求到达后，就创建一个新的子线程，由该子线程负责与客户程序完成通信，而主线程则再次调用 accept()函数等待新的客户连接请求到达，新客户请求到达后，则再次创建一个新的线程为新客户提供服务，如此循环往复。

与客户进行通信所使用的套接字是由 accept()函数创建并返回的。每次成功调用

accept()函数接收一个客户连接请求后,accept()函数都会创建一个新的与客户端已连接好的套接字并返回该套接字的标识符。创建子线程时,主线程将该套接字的标识符作为线程函数的参数传递给子线程,子线程通过主线程传来的线程标识符使用该套接字与客户通信。当连续有多个客户请求到达时,服务器进程内就会有多个子线程同时与多个不同客户通信。尽管这些线程是使用同一个线程函数创建的,但它们分别使用不同的套接字与不同的客户进行通信。

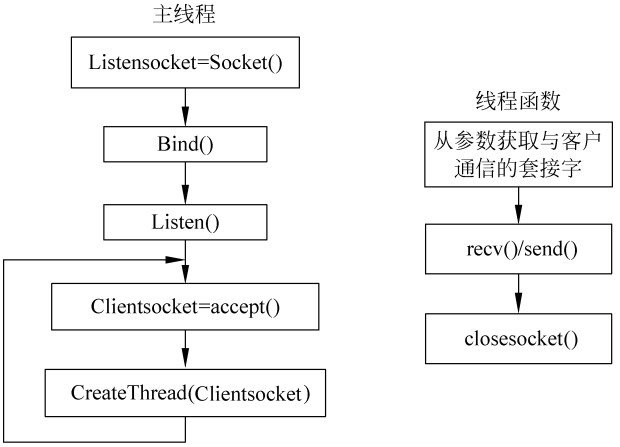

图 5.3 使用多线程的 TCP 服务器程序流程

例 5.4 使用多线程技术实现一个可同时为多个客户提供服务的回送服务器。所谓回送服务器,就是指这样一个服务器程序,它只将收到的客户发来的信息原样再发回去。

本例流程如图 5.4 所示,主程序要完成的工作首先是加载 WinSock 库、创建监听套接字、给监听套接字绑定地址、开始监听等准备工作,然后就进入循环,不断调用 accept()函数检测是否有客户请求到达,一旦有客户请求到达则创建一个新的线程。新线程回送任何从客户端发来的内容,如果客户端关闭连接,则线程关闭套接字后结束。

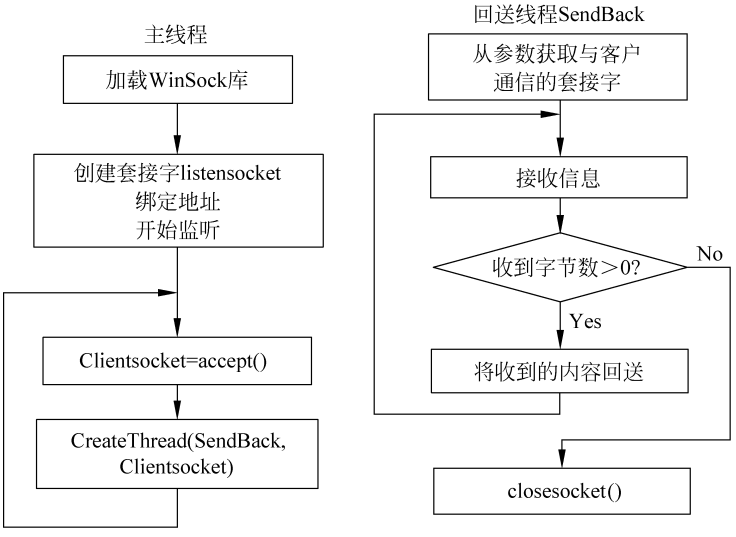

图 5.4 例 5.4 的主线程和内容回送线程的流程图

程序代码如下,其中创建线程的函数使用了 C++运行时库函数_beginthread(),需要注意,该函数对线程函数的格式要求是与 CreateThread()函数不同的。

```
#include "iostream"
#include "process.h"        //使用 C++运行时库中的函数创建多线程
#include "winsock2.h"
#include "WS2tcpip.h"       //使用 inet_ntop()函数进行地址格式转换要求包含该头文件
#define PORT 65432          //定义服务器的监听端口号
#pragma comment(lib,"ws2_32.lib")
using namespace std;
void SendBack(void * par);  //声明符合_beginthread()函数要求的线程函数
int main()
{
    SOCKET sock_server, newsock;
    struct sockaddr_in addr, client_addr;
    unsigned hThread;
    int addr_len = sizeof(struct sockaddr_in);
    /*** 初始化 winsock DLL ***/
    WSADATA wsaData;
    if (WSAStartup( MAKEWORD(2, 2), &wsaData) != 0)
    {
        cout << "加载 winsock.dll 失败!\n";
        return 0;
    }
    /*** 创建套接字 ***/
    if ((sock_server = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET) //建立一
个 socket
    {
        cout << "创建套接字失败!\n";
        WSACleanup();
        return 0;
    }
        /*** 绑定 IP 端口 ***/
    memset((void *)&addr, 0, addr_len);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);    //使用本机的所有 IP 地址
    if (bind(sock_server, (LPSOCKADDR)&addr, sizeof(addr)) != 0)
    {
        cout << "绑定地址失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    /*** 开始监听 ***/
    if (listen(sock_server, 5) != 0)
    {
        cout << "listen 函数调用失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    else
        cout << "listenning.....\n";
```

```
    /*** 接收并处理客户连接 ***/
    char client_ip[20];
    in_addr a;
    while (1)
    {
        newsock = accept(sock_server, (LPSOCKADDR)&client_addr, &addr_len);
        if (newsock != INVALID_SOCKET)
        {
            a = client_addr.sin_addr;
            cout << "cnnect from " << inet_ntop(AF_INET,&a, client_ip,20) << endl;
            hThread = _beginthread(SendBack, 0, (LPVOID)newsock);    //启动线程
        }
        else
            break;
    }
    closesocket(sock_server);
    WSACleanup();
    return 0;
}
/****************** 回送客户信息的线程函数 ********************************/
void SendBack(void * par)
{
    char buffer[1000];
    SOCKET sock = (SOCKET)par;
    int size = recv(sock, buffer, sizeof(buffer), 0);    //接收客户发来的消息
    while (size > 0)
    {
        if(send(sock, (char * )buffer, size, 0)< = 0)
            break;
        size = recv(sock, buffer, sizeof(buffer), 0);
    }
    closesocket(sock);     //关闭 socket
    return;
}
```

用于测试该服务器的客户端代码比较简单,读者可自己编写。该书配套的教学资源里面提供了完整的客户端程序代码供读者参考。

5.4 线程的同步与互斥

5.4.1 线程的同步

线程同步是指线程之间所具有的一种制约关系,一个线程的执行依赖另外一个或多个线程的消息,当它没有得到这些线程的消息时应等待,直到消息到达时才被唤醒。同步可以理解为这样一种情况:若干线程各自对自己的数据进行处理,然后在某个点必须汇总一下数据,否则不能进行下一步的工作。也可以理解为这样一种情况:若干线程等待某个事件发生,当等待的事件发生时,便一起开始执行。

在 Windows 系统中,通常使用事件对象实现同步。事件对象是最简单的同步对象,用于一个线程在某种情况发生时唤醒另外一个线程。

事件对象是 Windows 系统内核维护的一种数据结构，由于是内核维护的，因此只能被内核访问，应用程序无法在内存中直接找到并改变其内容。

事件对象有两种工作状态：一种是“有信号”(signaled)状态；另一种是“无信号”(nonsignaled)状态。当与事件对象关联的事件发生时，事件对象会从“无信号”状态变成“有信号”状态。

事件对象有两种工作模式：人工重设(manual reset)模式和自动重设(auto reset)模式。在人工重设模式下，程序完成对事件的处理后需要调用相关函数将其重新设置为“无信号”状态；在自动重设模式下则会自动返回“无信号”状态。

为了便于程序使用事件对象，Windows 系统提供了一组 Windows API 函数对事件对象进行操作，例如，创建事件对象可用 CreatEvent()函数，事件状态可用 SetEvent()函数或 ResetEvent()来设置，事件的状态可以被 WaitForSingleObject()函数或 WaitForMultipleObject()函数等“事件通知等待”函数捕捉。

MFC 用类 CEvent 对事件对象进行了包装，因此，在 MFC 编程中，事件对象可由 CEvent 类的对象来表示。在创建 CEvent 类的对象时，默认创建的是人工重设模式事件。常用的 CEvent 类的各成员函数的原型和参数说明如下。

1. 构造函数

函数原型

```
CEvent(
    BOOL bInitiallyOwn = FALSE,
    BOOL bManualReset = FALSE,
    LPCTSTR lpszName = NULL,
    LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

函数参数

- bInitiallyOwn：指定事件对象初始化状态，TRUE 为有信号，FALSE 为无信号。
- bManualReset：指定事件对象的类型。如果为 TRUE，则指定事件对象是一种人工事件；如果设置为 FALSE，则事件对象是一个自动事件。
- lpszName：CEvent 对象的名称。如果为 NULL，该名称将为空。
- lpsaAttribute：指向一个 LPSECURITY_ATTRIBUTES 结构的指针。

2. 状态设置函数

```
BOOL CEvent::SetEvent();
```

将 CEvent 类对象的状态设置为有信号状态。如果事件是人工事件，则 CEvent 类对象保持为有信号状态，直到调用成员函数 ResetEvent()将其重新设置为无信号状态时为止。如果 CEvent 类对象为自动事件，则在 SetEvent()将事件设置为有信号状态后，CEvent 类对象由系统自动重置为无信号状态。

如果该函数执行成功，则返回非零值，否则返回零。

3. 状态恢复函数

```
BOOL CEventa::ResetEvent();
```

该函数将事件的状态设置为无信号状态,并保持该状态直至 SetEvent()被调用时为止。由于自动事件是由系统自动重置,故自动事件不需要调用该函数。如果该函数执行成功,返回非零值,否则返回零。

程序中一般通过调用 WaitForSingleObject()函数或 WaitForMultipleObject()来监视事件状态。WaitForSingleObject()函数通常用来监视一个事件对象,该函数被调用运行时将阻塞,直到其参数指定事件对象变为有信号状态时才返回; WaitForMultipleObjects()可用来监视多个事件对象。对于自动信号,WaitForSingleObject()函数或 WaitForMultipleObjects()函数返回时系统将自动将其设置为无信号状态。

WaitForSingleObject()函数原型

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

函数参数

- hHandle: 对象句柄。可以指定各种不同的对象,如 Event、Mutex、Process、Semaphore 等,**这些对象的句柄可由其继承自其父类的成员变量 m_hObject 获取**。注意,当等待仍在挂起状态时,句柄指向的对象被关闭,那么函数行为是未定义的。
- dwMilliseconds: 指定最长等待时间,单位为 ms(毫秒),如果指定一个非零值,函数处于等待状态直到 hHandle 指定的对象被触发(比如,CEvent 对象变为有信号状态),或者消耗完该指定时间。如果 dwMilliseconds 为 0,函数不会进入一个等待状态,它总是立即返回。如果 dwMilliseconds 为 INFINITE,那么只有对象被触发信号后,函数才会返回。

函数功能

WaitForSingleObject()函数用来检测 hHandle 事件的信号状态,在某一线程中调用该函数时,线程暂时挂起,如果在挂起的 dwMilliseconds 毫秒内,线程所等待的对象变为有信号状态,则该函数立即返回; 如果超时时间已经到达 dwMilliseconds 毫秒,但 hHandle 所指向的对象还没有变成有信号状态,函数照样返回。参数 dwMilliseconds 有两个具有特殊意义的值: 0 和 INFINITE。若为 0,则该函数立即返回; 若为 INFINITE,则线程一直被挂起,直到 hHandle 所指向的对象变为有信号状态时为止。

返回值

返回值指示出引发函数返回的事件,有可能返回的值如下所列。

WAIT_ABANDONED(0x00000080): 当 hHandle 为 mutex 时,如果拥有 mutex 的线程在结束时没有释放核心对象会引发此返回值。

WAIT_OBJECT_0(0x00000000): 对象已被激活。

WAIT_TIMEOUT(0x00000102): 等待超时。

WAIT_FAILED(0xFFFFFFFF): 出现错误,可通过 GetLastError()得到错误代码。

WaitForMultipleObject()函数原型

```
DWORD WaitForMultipleObjects(
```

```
    DWORD nCount ,
    CONST HANDLE * lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds
);
```

函数参数

- nCount：函数监测的对象的数量，取值必须介于 1 与 MAXIMUM_WAIT_OBJECTS(在 Windows 头文件中定义为 64)之间。
- lpHandles：指向对象句柄数组的指针。
- bWaitAll：指定函数的使用方式。该函数有两种不同的使用方式：一种是让线程进入等待状态，直到指定对象中的任意一个被触发(CEvent 对象变为有信号状态)；另一种方式是让线程进入等待状态，直到所有指定的对象都被触发。如果该参数为 TRUE，则为后一种方式；如果为 FALSE，则是前一种方式。
- dwMilliseconds：与 WaitForSingleObject()中的同名参数作用完全相同。

返回值

返回值指出引发函数返回的事件。如果 bWaitAll 为 TRUE，同时所有对象均变为已触发状态，则返回值是 WAIT_OBJECT_0；如果 bWaitAll 为 FALSE，则一旦某一对象变为触发状态，该函数将返回 WAIT_OBJECT_0 与 WAIT_OBJECT_0 + dwCount－1 之间的一个值，在这种情况下，要想知道哪个对象变为已触发状态，只要将返回值减去 WAIT_OBJECT_0 后得到的值作为 WaitForMultipleObjects()函数的第二个参数指定的句柄数组的下标便可找到该对象。返回 WAIT_FAILED 或 WAIT_TIMEOUT 的意义与 WaitForSingleObject()完全相同。

使用事件控制线程同步的步骤如下。

第一步，创建全局 Event 对象；

第二步，在先运行的线程的适当位置通过调用 CEvent::SetEvent()设置相应的事件为有信号状态；

第三步，在后运行的线程中调用 WaitForSingleObject()函数或 WaitForMultipleObjects()函数等待事件对象状态由“无信号”变为“有信号”。

例 5.5 线程同步的例子。设计一个服务器端程序，用于统计多种不同商品在两个商场一天的销售量，要求两个商场在晚九点使用相同客户程序输入各商品的销售量并上传到服务器，服务器程序在两个商场都上传完成后计算各商品的总销售量并输出。

为简化程序设计并且还要模仿出实际的情况，假设商品种类为 10，只上传每个商品的一项数据，即销售量，每输入完成一个商品的数据便立刻上传，且输入和上传顺序严格按照事先约定。

本程序使用多线程技术实现与两个客户的并发通信。主线程与子线程共享的数据包括通信所需的套接字、事件对象和用户传来的数据，这些数据对不同子线程而言是完全不同的，因此保存这些数据的变量对每个子线程来说也应不同。为此可定义三个全局数组来存放这三种数据，每个线程对应于数组的一个不同元素，这样，主线程在创建子线程时，只需将相应数组元素的下标传给子线程就可以了。

由于只有当两个商场的数据均上传完成后主线程才能进行数据合并，因此子线程在完

成数据接收后需要通知主线程,为此,线程函数在接收数据完成后需要将本线程对应的事件对象设置为“有消息”状态,主程序在启动接收数据的线程后需要调用 WaitForMultipleObjects() 等待两个子线程均设置了事件对象为“有消息”状态后才能继续运行。服务器程序的详细流程如图 5.5 所示。

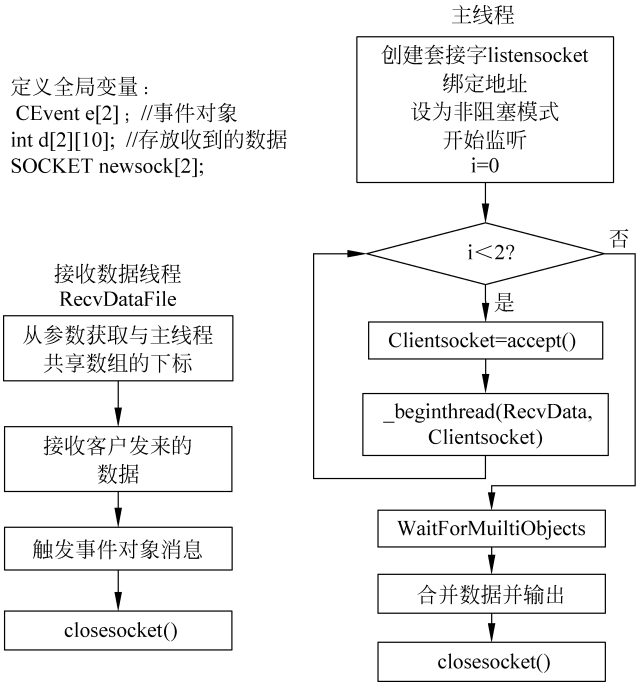

图 5.5 例 5.5 用于接收数据的子线程和主线程流程

服务器程序的完整代码如下。

```
#include "iostream"
#include "afxmt.h"                    //使用事件对象须包含此文件
#include "process.h"                  //使用 C++运行时库中的函数创建多线程
#include "WS2tcpip.h"                 //使用 inet_ntop()函数进行地址格式转换要求包含该头文件
#define PORT 65432
using namespace std;
int volatile d[2][10];                //volatile 是告诉编译器不要对数组 d 进行编译优化
CEvent e[2];
SOCKET newsock[2];
void RecvData(LPVOID par)
{
    int n = (int)par;
    for (int i = 0; i < 10; i++)  //接收客户发来的数据
        if (recv(newsock[n], (char * )d[n][i], sizeof(int), 0) < 0)
        {
            cout << "接收信息失败!错误代码:" << WSAGetLastError() << endl;
            break;
        }
    e[n].SetEvent();                  //设置事件为有信号状态
```

```
    closesocket(newsock[n]);      //关闭套接字
    return;
}
int main(int argc, char * argv[])
{
    SOCKET sock_server;
    struct sockaddr_in addr, client_addr;
    int addr_len = sizeof(struct sockaddr_in);
    /*** 初始化 winsock DLL ***/
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        cout << "加载 winsock.dll 失败!\n";
        return 0;
    }
    /*** 创建套接字 ***/
    if ((sock_server = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        cout << "创建套接字失败!\n";
        WSACleanup();
        return 0;
    }
    /*** 绑定 IP 端口 ***/
    memset((void *)&addr, 0, addr_len);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);    //监听本机的所有 IP 地址
    if (bind(sock_server, (LPSOCKADDR)&addr, sizeof(addr)) != 0)
    {
        cout << "绑定地址失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    /*** 开始监听 ***/
    if (listen(sock_server, 5) != 0)
    {
        cout << "listen 函数调用失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    else
        cout << "listenning.....\n";
    /*** 接收并处理客户连接 ***/
    char cip[20];
    in_addr a;
    for (int i = 0; i < 2; i++)
    {
        newsock[i] = accept(sock_server,(LPSOCKADDR)&client_addr,&addr_len);
        if(newsock[i] != INVALID_SOCKET)
```

```
            {
                a = client_addr.sin_addr;
                cout <<"cnnect from"<< inet_ntop(AF_INET,&a,cip,20)<< endl;
                _beginthread(RecvData, 0, (LPVOID)i);     //启动线程
            }
            else
            {
                cout << "连接失败!错误码为: " << WSAGetLastError() << endl;
                break;
            }
        }
        closesocket(sock_server);
        WSACleanup();
        HANDLE hEventhandls[2];
        hEventhandls[0] = e[0].m_hObject;     //将事件对象句柄存入句柄数组
        hEventhandls[1] = e[1].m_hObject;
        WaitForMultipleObjects(2,hEventhandls,true,INFINITE);     //等待事件消息
        for (int x = 0;x < 10;x++)
            cout << d[0][x]<<" + "<< d[1][x]<<" = "<< d[0][x] + d[1][x]<< endl;
        return 0;
    }
```

客户端程序比较简单,请读者自己完成。

注意:运行例 5.5 及本章以后的其他例题时,都需要修改项目"属性页"中的"配置属性"→"常规"→"MFC 的使用"为"在静态库中使用 MFC"。

5.4.2 线程间的互斥

在多线程应用程序中,如果一个线程完全独立,与其他线程没有数据存取等资源访问上的冲突,则可按照通常单线程的方法进行编程。但是,情况常常并不是这样,多个线程经常要同时访问一些共享资源。当两个或多个线程同时访问某个共享资源时,可能会引发一些不符合需求的、不可预知的结果。例如,一个线程可能正在更新某个共享的数据结构中的内容,而另一个线程则正在从同一数据结构中读取内容,这时就无法得知读取数据的线程读到的是更新前的数据还是更新后的数据。

为了防止这种现象发生,必须要求当一个线程访问共享数据区时其他线程不能访问,也就是所谓的互斥访问。线程互斥是指,当有若干个线程都要使用某一共享资源时,任何时刻最多只允许一个线程去使用,其他要使用该资源的线程必须等待,直到占用资源者释放该资源。实现互斥访问的方法有多种,比较典型的有使用临界区对象(CriticalSection)、使用互斥对象(Mutex)和使用信号量。下面以使用互斥对象为例,了解实现线程互斥的具体方法。

互斥对象 Mutex 很适合用来协调多个线程对共享资源的互斥访问。互斥对象不仅可以在同一应用程序的线程间实现互斥,还可以在不同的进程间实现互斥。

互斥对应一个 CMutex 类的对象,只有拥有互斥对象的线程才具有访问共享资源的权限,由于与共享资源对应的互斥对象只有一个,因此就决定了任何情况下此共享资源都不会同时被多个线程访问。占有互斥对象的线程在完成对共享资源的操作后应释放互斥对象,

以便其他线程访问共享资源。

使用互斥对象时必须首先为共享数据定义一个全局互斥对象。CMutex类的构造函数原型如下。

```
CMutex(
    BOOL bInitiallyOwn = FALSE,
    LPCTSTR lpszName = NULL,
    LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

函数参数

- bInitiallyOwn：用来指定互斥体对象初始状态是锁定(TRUE)还是非锁定(FALSE)。
- lpszName：用来指定互斥对象的名称。
- lpsaAttribute：为一个指向LPSECURITY_ATTRIBUTES结构的指针。创建的对象在用于线程互斥时一般为NULL。

定义互斥对象后，线程在访问共享资源时就可以调用互斥对象的Lock()成员函数获得互斥体对象的拥有权，从而阻止其他线程对共享资源的访问，访问完后，则调用UnLock()成员函数释放对互斥对象的拥有权，从而允许其他线程访问共享资源。

如果只处理单个互斥，除了直接使用互斥对象的Lock()和UnLock()函数锁定或解锁互斥对象外，还可以使用CSingleLock对象来管理互斥对象。CSingleLock对象的Lock()函数可以占有互斥，Unlock()则可释放互斥。CSingleLock类的构造函数如下。

```
CSingleLock( CSyncObject * pObject, BOOL bInitialLock = FALSE );
```

函数参数

- pObject：指向要被访问的同步对象(在这里是需要被管理的互斥对象)，不能是NULL。
- bInitialLock：指示是否要在最初尝试访问所提供的对象。默认值为FALSE。

如果线程中需要同时处理多个互斥对象，则必须创建一个CMultiLock对象来对多个互斥对象进行管理。CMultiLock类的构造函数如下。

```
CMultiLock(CSyncObject * ppObjects[],DWORD dwCount,BOOL bInitialLock = FALSE);
```

函数参数

- ppObjects：保存要处理的多个互斥对象的指针的数组，不能为NULL。
- dwCount：ppObjects的元素个数，必须大于0。
- bInitialLock：指定所提供的对象初始状态。

与CSingleLock对象一样，使用成员函数Lock()锁定所有管理的互斥对象，使用Unlock()释放所有管理的互斥对象。成员函数Lock()的原型如下。

```
DWORD Lock(
    DWORD dwTimeOut = INFINITE,
    BOOL bWaitForAll = TRUE,
    DWORD dwWakeMask = 0
);
```

函数参数

- dwTimeOut：指定等待能成功锁定所管理对象的最长容忍时间，默认值为 INFINITE，表示锁定不成功函数将永远阻塞。
- bWaitForAll：指定是否所有对象都锁定成功才返回。取值为 FALSE 时，只要所管理对象中有一个成功锁定就成功返回。
- dwWakeMask：指定其他返回条件。

返回值

函数失败返回－1，等待超时返回 WAIT_TIMEOUT。

例 5.6 线程互斥的例子。使用多线程技术编写一个并发服务器程序，该程序将一个给定的文件传输给多个客户。

根据使用多线程实现的并发 TCP 服务器程序流程，主程序在打开指定文件后，首先完成创建监听套接字，给监听套接字绑定地址，并使监听套接字进入监听状态，然后就进入循环不断接受客户连接请求。在循环体中，不断调用 accept()函数检测是否有客户请求到达，一旦有客户请求到达则创建一个新的线程给客户发送文件，新的线程发送文件完成后将自行退出。文件发送具体过程请参看第 4 章的相关内容，这里不再赘述。

这里遇到的一个新问题是，当两个或两个以上的客户同时请求下载文件时，服务器会同时启动多个线程读取同一个已打开的文件，由于文件对象的“文件读取位置指针”的唯一性，多个线程交替读取同一文件必然导致各线程都难以读取全部内容，因此这个已打开的要被发送的文件是互斥资源。这里使用互斥对象实现线程的互斥，因此程序开始需要定义一个全局互斥对象，线程函数内部需要在访问文件内容的代码前将互斥对象加锁，访问完后解锁。程序的流程图如图 5.6 所示。

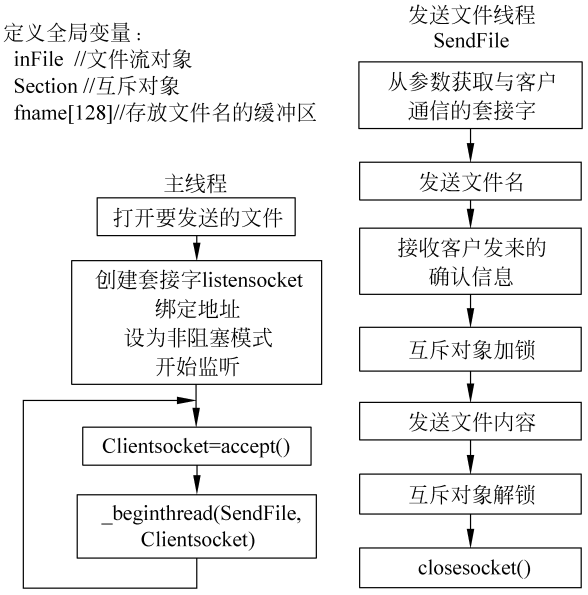

图 5.6 例 5.6 的主线程和发送文件内容线程的流程图

需要注意,在实际应用中,由于操作系统允许同一个文件以只读方式多次打开,并且不存在互斥问题,所以,在使用多线程技术将一个文件并发传输给多个客户时,不需要在主线程中先把文件打开,正常的做法应该是在每个子线程内分别以只读方式打开文件,传输完成后再各自关闭。这里这么做只是为了说明和演示线程互斥的相关概念和方法。

服务器端程序的完整代码如下。

```
#include "iostream"
#include "afxmt.h"                    //使用互斥对象须包含此文件
#include "process.h"                  //使用 C++运行时库中的函数创建多线程
#include "winsock2.h"
#include "fstream"
#define PORT 65432                    //定义服务器的监听端口号
#pragma comment(lib,"ws2_32.lib")
using namespace std;
/**** 主程序和线程函数共用的全局变量定义 *******/
char fname[128] = { 0 };              //发送给客户端的无路径信息的文件名
ifstream inFile;                      //定义文件输入流
CMutex Section;                       //创建互斥对象
void SendFile(void * par);            //发送文件的线程函数声明
/****** 主函数 *********/
int main(){
    /*** 定义相关的变量 ***/
    char filename[128];               //存放从键盘输入的含有信息的文件名
    int sock_server;
    struct sockaddr_in addr, client_addr;
    int addr_len = sizeof(struct sockaddr_in);
    cout << "请输入要发送的文件路径及名称(例如 d:\a.txt)\n";
    cin >> filename;
    /*** 以二进制读方式打开要分发的文件 ***/
    inFile.open(filename, ios::in | ios::binary);  //打开文件
    if (!inFile.is_open()){
        cout << "Cannot open " << filename << endl;
        return 0;                     //文件打开失败则退出
    }
    /*** 截取发送给客户端的文件名 ***/
    int len = strlen(filename);
    int i = len;
    while (filename[i] != '\\' && i >= 0) i--;
    if (i < 0)i = 0;else i++;
    int m = 0;
    while (filename[m + i] != '\0'){
        fname[m] = filename[m + i];
        m++;
    }
    /*** 初始化 winsock DLL ***/
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0){
        cout << "加载 winsock.dll 失败!\n";
        return 0;
    }
    /*** 创建套接字 ***/
    if ((sock_server = socket(AF_INET, SOCK_STREAM, 0)) < 0) //建立一个 socket
    {
```

```
        cout << "创建套接字失败!\n";
        WSACleanup();
        return 0;
    }
    /*** 绑定 IP 端口 ***/
    memset((void *)&addr, 0, addr_len);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);  //使用本机的所有 IP 地址
    if (bind(sock_server, (LPSOCKADDR)&addr, sizeof(addr)) != 0){
        cout << "绑定地址失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    /*** 开始监听 ***/
    if (listen(sock_server, 5) != 0){
        cout << "listen 函数调用失败!\n";
        closesocket(sock_server);
        WSACleanup();
        return 0;
    }
    else
        cout << "listenning.....\n";
    /*** 接收并处理客户连接 ***/
    SOCKET newsock;
    while(true){
        newsock = accept(sock_server, (LPSOCKADDR)&client_addr, &addr_len);
        if (newsock != INVALID_SOCKET){
            cout << "一个客户连接成功! " << endl;
            _beginthread(SendFile, 0, (LPVOID)newsock);    //启动文件发送线程
        }
        else
            break;
    }
    inFile.close();
    closesocket(sock_server);
    WSACleanup();
    return 0;
}
/****************** 文件传输线程函数 ********************************/
void SendFile(void *par){
    char buffer[1000];
    SOCKET sock = (SOCKET)par;
    send(sock, (char*)fname, strlen(fname) + 1, 0);        //发送文件名
    int size = recv(sock, buffer, sizeof(buffer), 0);      //接收"OK"消息
    if (strcmp(buffer, "OK") != 0){
        cout << "客户端出错!\n";
        closesocket(sock); //关闭 socket
        return;
    }
    /**** 传输文件内容 ****/
    Section.Lock(); //获取互斥对象
    inFile.seekg(0, ios::beg);
```

```
    //将文件读指针移动到文件头部,否则第二个客户收到的文件将是 0 字节
    while (!inFile.eof()){
        inFile.read(buffer, sizeof(buffer));
        size = inFile.gcount();                          //获取实际读取的字节数
        send(sock, (char*)buffer, size, 0);
    }
    Section.Unlock();                                    //释放互斥对象
    cout << "文件传输结束!\n";
    closesocket(sock);                                   //关闭 socket
    return ;
}
```

5.5 主监控线程和线程池

在网络通信中使用多线程主要有两种方式,即主监控线程和线程池。

在主监控线程方式中,程序使用一个主线程监控某特定端口,一旦在这个端口上发生连接请求,则主监控线程动态使用 CreateThread 派生出新的子线程处理该请求。主线程在派生子线程后不再对子线程加以控制和调度,而由子线程独自和客户方发生连接并处理异常。

使用这种方式的优点一是可以较快地实现原型设计,在用户数目较少、连接保持时间较长时表现较好;二是主线程不与子线程发生通信,在一定程度上减少了系统资源的消耗。

其缺点则是生成和终止子线程的开销比较大;对远端用户的控制较弱。这种多线程方式总的特点是"动态生成,静态调度"。

线程池是应用程序管理调度多个线程的一种方式,在使用线程池的程序中,程序的主线程在初始化时静态地生成一定数量的悬挂子线程,放置于线程池中,随后,主线程将对这些悬挂子线程进行动态调度。在网络通信程序中,使用线程池的服务器一旦收到客户发出连接请求,主线程将从线程池中查找一个悬挂的子线程。如果找到,主线程将该连接分配给这个被发现的子线程,子线程从主线程处接管该连接,并与用户通信,当连接结束时,该子线程将自动悬挂,并进入线程池等待再次被调度;如果已没有可用子线程,主线程将通知发起连接的客户。

使用这种方法进行设计的优点,一是主线程可以更好地对派生的子线程进行控制和调度;二是对远程用户的监控和管理能力较强。

虽然主线程对子线程的调度要消耗一定的资源,但是与主监控线程方式中派生和终止线程所要耗费的资源相比,要少很多。因此,使用该种方法设计和实现的系统在客户端连接和终止变更频繁时有上佳表现。

习题

1. 选择题

(1) 下面叙述正确的是(　　)。

A. 在同一进程中,一个线程函数只可以创建一个线程

B. 只有当进程中的所有线程都运行完毕,进程才会结束

C. 主线程是程序启动时由系统创建的,而子线程是由主线程或其他子线程创建的

D. 子线程在创建时,父线程会为其分配独立的地址空间

(2) 在 Windows 程序中,如果主线程终止了,(　　)。

A. 则进程也将随之终止,而不管其他线程是否执行完毕

B. 其他线程将继续执行完毕后终止进程

C. 若有其他线程还未执行完毕,则会造成资源浪费

D. 说明所有其他线程都已终止,因为主线程会等待所有子线程终止后才会终止

(3) 以下叙述正确的是(　　)。

A. 线程函数的格式可以是任意的

B. 一创建就进入到挂起状态的线程在 CPU 空闲时可自动进入运行状态

C. 使用同一个线程函数可以创建多个各自独立工作的线程

D. 一个被挂起的线程不能被 SuspendThread()函数重复挂起

(4) 用于创建线程的 Win32 SDK 函数是(　　)。

A. _beginthread()　　B. beginthread()

C. CreateThread()　　D. AfxBeginThread()

(5) 在下面的函数声明中,不能作为线程函数的是(　　)。

A. DWORD　f1(LPVOID　p);

B. int f2();

C. int f3(int x);

D. DWORD　f4(LPVOID p, int x);

(6) 关于线程函数的参数传递,下面叙述错误的是(　　)。

A. 实参为整型数据时可将其直接强制转换为 LPVOID 类型

B. 实参为字符数组时,可直接使用字符数组名或指向字符数组的 char * 类型的指针

C. 实参为 CString 对象时,可将指向 CString 对象的指针强制转换为 LPVOID

D. 实参为 double 类型时,可将其强制转换为 LPVOID 类型

(7) 使用多线程技术实现 TCP 并发服务器时,(　　)。

A. 主线程创建套接字并启动监听,子线程调用 accept()与客户建立连接并完成通信

B. 主线程调用 accept()与客户建立连接,并将 accept()返回的套接字传递给子线程

C. 主线程使用不同的线程函数,分别为每一个客户创建一个子线程进行通信

D. 传递给多个子线程的已连接套接字是同一个套接字

(8) 若线程函数原型为 DWORD　f();,则以下创建线程语句正确的是(　　)。

A. CreateThread(NULL,0, (LPTHREAD_START_ROUTINE) f, NULL, 0, &trdID);

B. CreateThread(NULL, 0, f, NULL , 0, &trdID);

C. CreateThread(NULL, 0, f(), NULL , 0, &trdID);

D. CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) f(), NULL, 0, NULL);

(9) 若线程函数原型为 DWORD _stdcall f();,则以下创建线程语句正确的是(　　)。

A. CreateThread(NULL,0, (LPTHREAD_START_ROUTINE) f, NULL, 0, &trdID);

B. CreateThread(NULL, 0, f, NULL , 0, &trdID);

C. CreateThread(NULL, 0, f(), NULL , 0, &trdID);

D. CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) f(), NULL, 0, NULL);

(10) 应用程序中的某一线程要强行终止另一未结束的其他线程可以调用(　　)函数。

A. ExitThread()　　B. TerminateThread()

C. SuspendThread()　　D. ResumeThread()

2. 填空题

(1) _____是指具有一定独立功能的程序在某个数据集合上的一次运行活动,是系统进行资源分配和调度运行的一个独立单位。

(2) 进程的基本状态有三种:_____、_____、_____。

(3) 线程从创建到结束之间可能存在的状态有两种,即_____状态和_____状态。

(4) Windows 系统支持多线程程序,任何一个 Windows 中的应用进程都至少有一个线程,即_____,其他线程都是由该线程直接或间接创建的。

(5) 要将一个整型数据作为参数传递给线程时,可将该整型数据强制转换为 LPVOID 类型作为 CreateThread()函数的_____传递给线程。

(6) 如果一个 TCP 服务器可同时保持与多个客户的连接,该服务器就被称为_____服务器,多线程技术是实现这种服务器的一种典型技术。

3. 简答题

(1) 实现线程间的通信有几种方法?

(2) 什么是事件对象?程序中要用事件对象实现线程间的通信需要哪几个步骤?

(3) 什么是线程的互斥?有哪些方法可实现线程间的互斥?

(4) 线程间的同步是指什么?实现同步的方法有哪些?

4. 编程题

(1) 使用多线程技术编写一个图形界面的点到点聊天程序,服务器端程序和客户端程序的界面分别如图 5.7 和图 5.8 所示。在客户端输入服务器的 IP 地址,单击“连接”按钮建立连接后服务器端和客户端便可进行聊天,聊天过程中双方均可在任意时刻发送信息给对方。

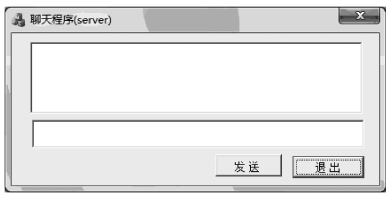

图 5.7　编程题(1)的服务器端程序界面

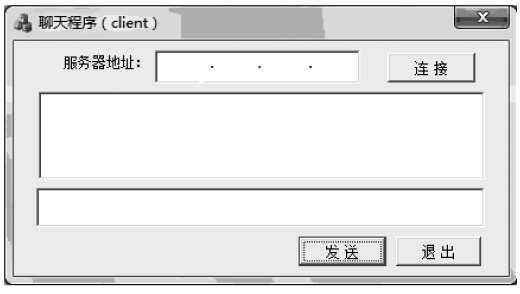

图 5.8　编程题(1)的客户端程序界面

提示:客户端程序和服务器端程序均使用单独的线程接收显示对方发来的信息。发送数据的代码写在“发送”按钮的命令处理函数中。接收数据的代码写在一个自己定义的线程

函数中,该线程函数不是对话框类的成员函数,但它是对话框类的实现文件中的一个普通函数。由于只与一个客户聊天,服务器端可将 accept()与 recv()放在同一线程中,该线程在窗口类的 OnInitDialog()函数中创建;客户端的数据接收线程,则应在"连接"按钮的处理函数中,调用 connect()后创建。

(2) 上一题中能否实现服务器端同时与多个客户聊天?应如何实现?允许在界面上添加控件。

(3) 编写一个程序,该程序用于收集多个客户端发来的姓名与电话两项信息。要求服务器端采用多线程技术实现多客户的并发连接;并且所有信息都以文本方式保存在同一个文件中,服务器在收到一个客户数据后打开文件,写入后立刻关闭该文件。

提示:当一个线程打开文件后另一个线程再试图来打开同一个文件会引起文件打开错误,需要用到线程间的互斥来解决这一问题。

实验 4 TCP 服务器端的多线程编程

一、实验目的

(1) 掌握多线程的概念及多线程编程的基本方法;

(2) 掌握 TCP 服务器端使用多线程技术同时与多个客户通信的编程方法。

二、实验设备及软件

已联网的运行 Windows 系统的计算机,Visual Studio 2017(已选择安装 MFC)。

三、实验内容

(1) 将实验 3 的实验内容 1 中的服务器程序用多线程编程技术改写,使之可同时与多个客户端通信。显示收到的内容时,按如下格式显示。

客户 IP 地址:学号 姓名 性别 考试成绩

收到姓名为 end 的信息时断开与客户端的连接。

(2) 将实验 3 的实验内容 2 中的服务器程序用多线程编程技术改写,使之可同时接收多个客户端上传文件。

四、实验步骤

(1) 编写实验内容(1)要求的服务器程序,先自己调试成功后再与至少其他两位同学配合,进一步测试自己编写的服务器程序。

(2) 编写实验内容(2)要求的服务器程序,先自己调试成功后再与至少其他两位同学配合,进一步测试自己编写的服务器程序。

五、思考题

如何将实验内容 1 中服务器端收到的不同客户端发来的信息保存在同一个文本文件中?需不需要考虑文件访问的互斥问题?