第 20 章 Windows中的线程同步

本章不再重复介绍第 18 章关于“临界区”和“同步”的理论，而主要介绍 Windows 中的线程同步方法，这些技术对应于 Linux 平台下的互斥量和信号量等线程同步技术。

20.1 同步方法的分类及 CRITICAL_SECTION 同步

Windows中存在多种同步技术，它们的基本概念大同小异，相互间也有一定联系，所以不难掌握。

用户模式（User mode）和内核模式（Kernal mode）

Windows操作系统的运行方式（程序运行方式）是“双模式操作”（Dual-mode Operation），这意味着Windows在运行过程中存在如下2种模式。

- 用户模式：运行应用程序的基本模式，禁止访问物理设备，而且会限制访问的内存区域。
- 内核模式：操作系统运行时的模式，不仅不会限制访问的内存区域，而且访问的硬件设备也不会受限。

内核是操作系统的核心模块，可以简单定义为如下形式。

- 用户模式：应用程序的运行模式。
- 内核模式：操作系统的运行模式。

实际上，在应用程序运行过程中，Windows操作系统不会一直停留在用户模式，而是在用户模式和内核模式之间切换。例如，各位可以在Windows中创建线程。虽然创建线程的请求是由应用程序的函数调用完成，但实际创建线程的是操作系统。因此，创建线程的过程中无法避免向内核模式的转换。

定义这2种模式主要是为了提高安全性。应用程序的运行时错误会破坏操作系统及各种资源。

特别是C/C++可以进行指针运算，很容易发生这类问题。例如，因为错误的指针运算覆盖了操作系统中存有重要数据的内存区域，这很可能引起操作系统崩溃。但实际上各位从未经历过这类事件，因为用户模式会保护与操作系统有关的内存区域。因此，即使遇到错误的指针运算也仅停止应用程序的运行，而不会影响操作系统。总之，像线程这种伴随着内核对象创建的资源创建过程中，都要默认经历如下模式转换过程：

用户模式→内核模式→用户模式

从用户模式切换到内核模式是为了创建资源，从内核模式再次切换到用户模式是为了执行应用程序的剩余部分。不仅是资源的创建，与内核对象有关的所有事务都在内核模式下进行。模式切换对系统而言其实也是一种负担，频繁的模式切换会影响性能。

用户模式同步

用户模式同步是用户模式下进行的同步，即无需操作系统的帮助而在应用程序级别进行的同步。用户模式同步的最大优点是——速度快。无需切换到内核模式，仅考虑这一点也比经历内核模式切换的其他方法要快。而且使用方法相对简单，因此，适当运用用户模式同步并无坏处。但因为这种同步方法不会借助操作系统的力量，其功能上存在一定局限性。稍后将介绍属于用户模式同步的、基于“CRITICAL_SECTION”的同步方法。

内核模式同步

前面已介绍过用户模式同步，即使不另作说明，相信各位也能大概说出内核模式同步的特性及优缺点。下面给出内核模式同步的优点。

- 比用户模式同步提供的功能更多。
- 可以指定超时，防止产生死锁。

因为都是通过操作系统的帮助完成同步的，所以提供更多功能。特别是在内核模式同步中，可以跨越进程进行线程同步。与此同时，由于无法避免用户模式和内核模式之间的切换，所以性能上会受到一定影响。

大家此时很可能想到：“因为是基于内核对象的操作，所以可以进行不同进程之间的同步！”因为内核对象并不属于某一进程，而是操作系统拥有并管理的。

知识补给站　死锁

我在第18章中把临界区比喻为洗手间。那么请考虑如下情况。有人进入洗手间后把门锁上，一会儿又来人开始在门外等待。本来里面的人应该先解锁再离开，但现在不知何种原因，里面的人通过小窗户爬出了洗手间。当然谁都不得而知，那在外面等的人该

怎么办呢？本来应该询问里面的情况，并决定是否去其他洗手间，但此人一直在等待，直到洗手间被强行关闭（程序强行退出）。

虽然多少有些荒诞，但这种情况描述的就是死锁。发生死锁时，等待进入临界区的阻塞状态的线程无法退出。死锁发生的原因有很多，以第18章的Mutex为例，调用pthread_mutex_lock函数进入临界区的线程如果不调用pthread_mutex_unlock函数，将发生死锁。这其实是极为简单的情形，大部分情况下发生死锁的原因都很难确认。

基于CRITICAL_SECTION的同步

基于CRITICAL_SECTION的同步中将创建并运用“CRITICAL_SECTION对象”，但这并非内核对象。与其他同步对象相同，它是进入临界区的一把“钥匙”（Key）。因此，为了进入临界区，需要得到CRITICAL_SECTION对象这把“钥匙”。相反，离开时应上交CRITICAL_SECTION对象（以下简称CS）。下面介绍CS对象的初始化及销毁相关函数。

```
#include <windows.h>

void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection　Init...函数中传入需要初始化的CRITICAL_SECTION对象的地址值，反之，Del...函数中传入需要解除的CRITICAL_SECTION对象的地址值。

上述函数的参数类型LPCRITICAL_SECTION是CRITICAL_SECTION指针类型。另外DeleteCriticalSection并不是销毁CRITICAL_SECTION对象的函数。该函数的作用是销毁CRITICAL_SECTION对象使用过的（CRITICAL_SECTION对象相关的）资源。接下来介绍获取（拥有）及释放CS对象的函数，可以简单理解为获取和释放“钥匙”的函数。

```
#include <windows.h>

void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection　获取（拥有）和释放的CRITICAL_SECTION对象的地址值。

与Linux部分中介绍过的互斥量类似，相信大部分人仅靠这些函数介绍也能写出示例程序（我的个人经验）。下面利用CS对象将第19章的示例thread3_win.c改为同步程序。

❖ SyncCS_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>
4.
5.  #define NUM_THREAD  50
6.  unsigned WINAPI threadInc(void * arg);
7.  unsigned WINAPI threadDes(void * arg);
8.
9.  long long num=0;
10. CRITICAL_SECTION cs;
11.
12. int main(int argc, char *argv[])
13. {
14.     HANDLE tHandles[NUM_THREAD];
15.     int i;
16.
17.     InitializeCriticalSection(&cs);
18.     for(i=0; i<NUM_THREAD; i++)
19.     {
20.         if(i%2)
21.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadInc, NULL, 0, NULL);
22.         else
23.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadDes, NULL, 0, NULL);
24.     }
25.
26.     WaitForMultipleObjects(NUM_THREAD, tHandles, TRUE, INFINITE);
27.     DeleteCriticalSection(&cs);
28.     printf("result: %lld \n", num);
29.     return 0;
30. }
31.
32. unsigned WINAPI threadInc(void * arg)
33. {
34.     int i;
35.     EnterCriticalSection(&cs);
36.     for(i=0; i<50000000; i++)
37.         num+=1;
38.     LeaveCriticalSection(&cs);
39.     return 0;
40. }
41. unsigned WINAPI threadDes(void * arg)
42. {
43.     int i;
44.     EnterCriticalSection(&cs);
45.     for(i=0; i<50000000; i++)
46.         num-=1;
47.     LeaveCriticalSection(&cs);
48.     return 0;
49. }
```

代码说明

- 第17、27行：CS对象的初始化和解除代码。
- 第35~38、44~47行：第33~38行与第44~47行之间构成1个临界区，防止同时访问。为了尽快得到结果，将整个循环当做临界区。

❖ 运行结果：SyncCS_win.c

```
result: 0
```

程序中将整个循环纳入临界区，主要是为了减少运行时间。如果只将访问num的语句纳入临界区，那将不知何时才能得到运行结果（如果时间充裕可以一试，但运行时间会长得让人怀疑是否发生了死锁），因为这将导致大量获取和释放CS对象。另外，上述示例仅仅是为了学习同步机制而编写的，没有任何现实意义（如此编写程序的情况本身并不现实）。

20.2 内核模式的同步方法

典型的内核模式同步方法有基于事件（Event）、信号量、互斥量等内核对象的同步，下面从互斥量开始逐一介绍。

基于互斥量（Mutual Exclusion）对象的同步

基于互斥量对象的同步方法与基于CS对象的同步方法类似，因此，互斥量对象同样可以理解为“钥匙”。首先介绍创建互斥量对象的函数。

```
#include <windows.h>

HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName);
```

➔ 成功时返回创建的互斥量对象句柄，失败时返回 NULL。

- lpMutexAttributes　传递安全相关的配置信息，使用默认安全设置时可以传递NULL。
- bInitialOwner　如果为TRUE，则创建出的互斥量对象属于调用该函数的线程，同时进入non-signaled状态；如果为FALSE，则创建出的互斥量对象不属于任何线程，此时状态为signaled。
- lpName　用于命名互斥量对象。传入NULL时创建无名的互斥量对象。

从上述参数说明中可以看到，如果互斥量对象不属于任何拥有者，则将进入signaled状态。利用该特点进行同步。另外，互斥量属于内核对象，所以通过如下函数销毁。

```
#include <windows.h>

BOOL CloseHandle(HANDLE hObject);
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

- hObject　要销毁的内核对象的句柄。

上述函数是销毁内核对象的函数，所以同样可以销毁即将介绍的信号量及事件。下面介绍获取和释放互斥量的函数，但我认为只需介绍释放的函数，因为获取是通过各位熟悉的WaitForSingleObject函数完成的。

```
#include <windows.h>

BOOL ReleaseMutex(HANDLE hMutex);
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

- hMutex　需要释放（解除拥有）的互斥量对象句柄。

接下来分析获取和释放互斥量的过程。互斥量被某一线程获取时（拥有时）为signaled状态，释放时（未拥有时）进入signaled状态。因此，可以使用WaitForSingleObject函数验证互斥量是否已分配。该函数的调用结果有如下2种。

- 调用后进入阻塞状态：互斥量对象已被其他线程获取，现处于non-signaled状态。
- 调用后直接返回：其他线程未占用互斥量对象，现处于signaled状态。

互斥量在WaitForSingleObject函数返回时自动进入non-signaled状态，因为它是第19章介绍过的“auto-reset”模式的内核对象。结果，WaitForSingleObject函数成为申请互斥量时调用的函数。因此，基于互斥量的临界区保护代码如下。

```
WaitForSingleObject(hMutex, INFINITE);
// 临界区的开始
// . . . . . .
// 临界区的结束
ReleaseMutex(hMutex);
```

WaitForSingleObject函数使互斥量进入non-signaled状态，限制访问临界区，所以相当于临界区的门禁系统。相反，ReleaseMutex函数使互斥量重新进入signal状态，所以相当于临界区的出口。下面将之前介绍过的SyncCS_win.c示例改为互斥量对象的实现方式。更改后的程序与SyncCS_win.c没有太大区别，故省略相关说明。

❖ SyncMutex_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>
4.
5.  #define NUM_THREAD  50
6.  unsigned WINAPI threadInc(void * arg);
7.  unsigned WINAPI threadDes(void * arg);
8.
9.  long long num=0;
10. HANDLE hMutex;
11.
12. int main(int argc, char *argv[])
13. {
14.     HANDLE tHandles[NUM_THREAD];
15.     int i;
16.
17.     hMutex=CreateMutex(NULL, FALSE, NULL);
18.     for(i=0; i<NUM_THREAD; i++)
19.     {
20.         if(i%2)
21.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadInc, NULL, 0, NULL);
22.         else
23.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadDes, NULL, 0, NULL);
24.     }
25.
26.     WaitForMultipleObjects(NUM_THREAD, tHandles, TRUE, INFINITE);
27.     CloseHandle(hMutex);
28.     printf("result: %lld \n", num);
29.     return 0;
30. }
31.
32. unsigned WINAPI threadInc(void * arg)
33. {
34.     int i;
35.     WaitForSingleObject(hMutex, INFINITE);
36.     for(i=0; i<50000000; i++)
37.         num+=1;
38.     ReleaseMutex(hMutex);
39.     return 0;
40. }
41. unsigned WINAPI threadDes(void * arg)
42. {
43.     int i;
44.     WaitForSingleObject(hMutex, INFINITE);
45.     for(i=0; i<50000000; i++)
46.         num-=1;
47.     ReleaseMutex(hMutex);
48.     return 0;
49. }
```

❖ 运行结果：SyncMutex_win.c

```
result: 0
```

基于信号量对象的同步

Windows中基于信号量对象的同步也与Linux下的信号量类似，二者都是利用名为“信号量值”（Semaphore Value）的整数值完成同步的，而且该值都不能小于0。当然，Windows的信号量值注册于内核对象。

下面介绍创建信号量对象的函数，当然，其销毁同样是利用CloseHandle函数进行的。

```
#include <windows.h>

HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, LONG lInitialCount,
        LONG lMaximumCount, LPCTSTR lpName);
```

➔ 成功时返回创建的信号量对象的句柄，失败时返回 NULL。

- lpSemaphoreAttributes　安全配置信息，采用默认安全设置时传递NULL。
- lInitialCount　指定信号量的初始值，应大于0小于lMaximumCount。
- lMaximumCount　信号量的最大值。该值为1时，信号量变为只能表示0和1的二进制信号量。
- lpName　用于命名信号量对象。传递NULL时创建无名的信号量对象。

可以利用“信号量值为0时进入non-signaled状态，大于0时进入signaled状态”的特性进行同步。向lInitialCount参数传递0时，创建non-signaled状态的信号量对象。而向lMaximumCount传入3时，信号量最大值为3，因此可以实现3个线程同时访问临界区时的同步。下面介绍释放信号量对象的函数。

```
#include <windows.h>

BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG
lpPreviousCount);
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

- hSemaphore　传递需要释放的信号量对象。
- lReleaseCount　释放意味着信号量值的增加，通过该参数可以指定增加的值。超过最大值则不增加，返回FALSE。
- lpPreviousCount　用于保存修改之前值的变量地址，不需要时可传递NULL。

信号量对象的值大于0时成为signaled状态，为0时成为non-signaled状态。因此，调用WaitForSingleObject函数时，信号量大于0的情况下才会返回。返回的同时将信号量值减1，同时进入non-signaled状态（当然，仅限于信号量减1后等于0的情况）。可以通过如下程序结构保护临界区。

```
WaitForSingleObject(hSemaphore, INFINITE);
// 临界区的开始
// . . . . . .
// 临界区的结束
ReleaseSemaphore(hSemaphore, 1, NULL);
```

下面给出信号量对象相关示例，该示例只是第18章semaphore.c的Windows移植版。关于程序流说明请参考之前的内容，本示例中主要补充说明调用同步函数的部分。

❖ SyncSema_win.c

```
#include <stdio.h>
#include <windows.h>
#include <process.h>
unsigned WINAPI Read(void * arg);
unsigned WINAPI Accu(void * arg);

static HANDLE semOne;
static HANDLE semTwo;
static int num;

int main(int argc, char *argv[])
{
    HANDLE hThread1, hThread2;
    semOne=CreateSemaphore(NULL, 0, 1, NULL);
    semTwo=CreateSemaphore(NULL, 1, 1, NULL);

    hThread1=(HANDLE)_beginthreadex(NULL, 0, Read, NULL, 0, NULL);
    hThread2=(HANDLE)_beginthreadex(NULL, 0, Accu, NULL, 0, NULL);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    CloseHandle(semOne);
    CloseHandle(semTwo);
    return 0;
}

unsigned WINAPI Read(void * arg)
{
    int i;
    for(i=0; i<5; i++)
    {
```

```
33.          fputs("Input num: ", stdout);
34.          WaitForSingleObject(semTwo, INFINITE);
35.          scanf("%d", &num);
36.          ReleaseSemaphore(semOne, 1, NULL);
37.      }
38.      return 0;
39. }
40. unsigned WINAPI Accu(void * arg)
41. {
42.      int sum=0, i;
43.      for(i=0; i<5; i++)
44.      {
45.          WaitForSingleObject(semOne, INFINITE);
46.          sum+=num;
47.          ReleaseSemaphore(semTwo, 1, NULL);
48.      }
49.      printf("Result: %d \n", sum);
50.      return 0;
51. }
```

代码说明

- 第14、15行：创建2个信号量对象。第14行将信号量值设置为0进入non-signaled状态，第15行将信号量值设置为1进入signaled状态。第三个参数均为1，因此，2个信号量都是值为0或1的二进制信号量。
- 第34~36行、45~47行：本示例的特点是必须在循环内部构建临界区。但无论何种示例，通常都要尽量缩小临界区的范围以提高程序性能。

❖ 运行结果：SyncSema_win.c

```
Input num: 1
Input num: 2
Input num: 3
Input num: 4
Input num: 5
Result: 15
```

基于事件对象的同步

事件同步对象与前2种同步方法相比有很大不同，区别就在于，该方式下创建对象时，可以在自动以non-signaled状态运行的auto-reset模式和与之相反的manual-reset模式中任选其一。而事件对象的主要特点是可以创建manual-reset模式的对象，我也将对此进行重点讲解。首先介绍用于创建事件对象的函数。

```
#include <windows.h>

HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, BOOL bManualReset,
    BOOL bInitialState, LPCTSTR lpName);
```

➔ 成功时返回创建的事件对象句柄，失败时返回 NULL。

- lpEventAttributes　安全配置相关参数，采用默认安全配置时传入NULL。
- bManualReset　传入TRUE时创建manual-reset模式的事件对象，传入FALSE时创建auto-reset模式的事件对象。
- bInitialState　传入TRUE时创建signaled状态的事件对象，传入FALSE时创建non-signaled状态的事件对象。
- lpName　用于命名事件对象。传递NULL时创建无名的事件对象。

相信各位也发现了，上述函数中需要重点关注的是第二个参数。传入TRUE时创建manual-reset模式的事件对象，此时即使WaitForSingleObject函数返回也不会回到non-signaled状态。因此，在这种情况下，需要通过如下2个函数明确更改对象状态。

```
#include <windows.h>

BOOL ResetEvent(HANDLE hEvent);  //to the non-signaled
BOOL SetEvent(HANDLE hEvent);    //to the signaled
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

传递事件对象句柄并希望改为non-signaled状态时，应调用ResetEvent函数。如果希望改为signaled状态，则可以调用SetEvent函数。通过如下示例介绍事件对象的具体使用方法，该示例中的2个线程将同时等待输入字符串。

❖ SyncEvent_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>
4.  #define STR_LEN    100
5.
6.  unsigned WINAPI NumberOfA(void *arg);
7.  unsigned WINAPI NumberOfOthers(void *arg);
8.
9.  static char str[STR_LEN];
10. static HANDLE hEvent;
11.
12. int main(int argc, char *argv[])
```

```
13. {
14.     HANDLE hThread1, hThread2;
15.     hEvent=CreateEvent(NULL, TRUE, FALSE, NULL);
16.     hThread1=(HANDLE)_beginthreadex(NULL, 0, NumberOfA, NULL, 0, NULL);
17.     hThread2=(HANDLE)_beginthreadex(NULL, 0, NumberOfOthers, NULL, 0, NULL);
18.
19.     fputs("Input string: ", stdout);
20.     fgets(str, STR_LEN, stdin);
21.     SetEvent(hEvent);
22.
23.     WaitForSingleObject(hThread1, INFINITE);
24.     WaitForSingleObject(hThread2, INFINITE);
25.     ResetEvent(hEvent);
26.     CloseHandle(hEvent);
27.     return 0;
28. }
29.
30. unsigned WINAPI NumberOfA(void *arg)
31. {
32.     int i, cnt=0;
33.     WaitForSingleObject(hEvent, INFINITE);
34.     for(i=0; str[i]!=0; i++)
35.     {
36.         if(str[i]=='A')
37.             cnt++;
38.     }
39.     printf("Num of A: %d \n", cnt);
40.     return 0;
41. }
42. unsigned WINAPI NumberOfOthers(void *arg)
43. {
44.     int i, cnt=0;
45.     WaitForSingleObject(hEvent, INFINITE);
46.     for(i=0; str[i]!=0; i++)
47.     {
48.         if(str[i]!='A')
49.             cnt++;
50.     }
51.     printf("Num of others: %d \n", cnt-1);
52.     return 0;
53. }
```

代码说明

- 第15行：以non-signaled状态创建manual-reset模式的事件对象。
- 第16、17行：创建以NumberOfA和NumberOfOther函数为main函数的线程。这2个线程通过调用第20行的函数进入等待输入的状态。同时，第33、45行调用WaitForSingleObject函数。
- 第21行：读入字符串后将事件对象改为signaled状态。第33、45行中正在等待的2个线程将摆脱等待状态，开始执行。2个线程之所以能同时摆脱等待状态是因为事件对象仍处于signaled状态。
- 第25行：虽然本示例中没有太大的必要，但还是把事件对象的状态改为non-signaled。如果不进行明确更改，对象将继续停留在signaled状态。

❖ 运行结果：SyncEvent_win.c

```
Input string: ABCDABC
Num of A: 2
Num of others: 5
```

上述简单示例演示的是2个线程同时退出等待状态的情景。在这种情况下，以manual-reset模式创建事件对象应该是更好的选择。

20.3 Windows 平台下实现多线程服务器端

第18章讲完线程的创建和同步方法后，最终实现了多线程聊天服务器端和客户端。按照这种顺序，本章最后也将在Windows平台下实现聊天服务器端和客户端。首先给出聊天服务器端的源代码。该程序是第18章chat_serv.c的Windows移植版，故省略其说明。

❖ chat_serv_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <windows.h>
5.  #include <process.h>
6.
7.  #define BUF_SIZE 100
8.  #define MAX_CLNT 256
9.
10. unsigned WINAPI HandleClnt(void * arg);
11. void SendMsg(char * msg, int len);
12. void ErrorHandling(char * msg);
13.
14. int clntCnt=0;
15. SOCKET clntSocks[MAX_CLNT];
16. HANDLE hMutex;
17.
18. int main(int argc, char *argv[])
19. {
20.     WSADATA wsaData;
21.     SOCKET hServSock, hClntSock;
22.     SOCKADDR_IN servAdr, clntAdr;
23.     int clntAdrSz;
24.     HANDLE hThread;
25.     if(argc!=2) {
26.         printf("Usage : %s <port>\n", argv[0]);
27.         exit(1);
28.     }
29.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
30.         ErrorHandling("WSAStartup() error!");
31.
32.     hMutex=CreateMutex(NULL, FALSE, NULL);
33.     hServSock=socket(PF_INET, SOCK_STREAM, 0);
```

```
34.
35.     memset(&servAdr, 0, sizeof(servAdr));
36.     servAdr.sin_family=AF_INET;
37.     servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
38.     servAdr.sin_port=htons(atoi(argv[1]));
39.
40.     if(bind(hServSock, (SOCKADDR*) &servAdr, sizeof(servAdr))==SOCKET_ERROR)
41.         ErrorHandling("bind() error");
42.     if(listen(hServSock, 5)==SOCKET_ERROR)
43.         ErrorHandling("listen() error");
44.
45.     while(1)
46.     {
47.         clntAdrSz=sizeof(clntAdr);
48.         hClntSock=accept(hServSock, (SOCKADDR*)&clntAdr,&clntAdrSz);
49.
50.         WaitForSingleObject(hMutex, INFINITE);
51.         clntSocks[clntCnt++]=hClntSock;
52.         ReleaseMutex(hMutex);
53.
54.         hThread=
55.             (HANDLE)_beginthreadex(NULL, 0, HandleClnt, (void*)&hClntSock, 0, NULL);
56.         printf("Connected client IP: %s \n", inet_ntoa(clntAdr.sin_addr));
57.     }
58.     closesocket(hServSock);
59.     WSACleanup();
60.     return 0;
61. }
62.
63. unsigned WINAPI HandleClnt(void * arg)
64. {
65.     SOCKET hClntSock=*((SOCKET*)arg);
66.     int strLen=0, i;
67.     char msg[BUF_SIZE];
68.
69.     while((strLen=recv(hClntSock, msg, sizeof(msg), 0))!=0)
70.         SendMsg(msg, strLen);
71.
72.     WaitForSingleObject(hMutex, INFINITE);
73.     for(i=0; i<clntCnt; i++)   // remove disconnected client
74.     {
75.         if(hClntSock==clntSocks[i])
76.         {
77.             while(i++<clntCnt-1)
78.                 clntSocks[i]=clntSocks[i+1];
79.             break;
80.         }
81.     }
82.     clntCnt--;
83.     ReleaseMutex(hMutex);
84.     closesocket(hClntSock);
85.     return 0;
86. }
87. void SendMsg(char * msg, int len)    // send to all
```

```
88. {
89.     int i;
90.     WaitForSingleObject(hMutex, INFINITE);
91.     for(i=0; i<clntCnt; i++)
92.         send(clntSocks[i], msg, len, 0);
93.     ReleaseMutex(hMutex);
94. }
95. void ErrorHandling(char * msg)
96. {
97.     fputs(msg, stderr);
98.     fputc('\n', stderr);
99.     exit(1);
100.}
```

下面介绍聊天客户端。该示例是第18章的chat_clnt.c的Windows移植版，故同样省略其说明。

❖ chat_clnt_win.c

```
1.  #include <"头文件声明与chat_serv_win.c一致，故省略。">
2.  #define BUF_SIZE 100
3.  #define NAME_SIZE 20
4.
5.  unsigned WINAPI SendMsg(void * arg);
6.  unsigned WINAPI RecvMsg(void * arg);
7.  void ErrorHandling(char * msg);
8.
9.  char name[NAME_SIZE]="[DEFAULT]";
10. char msg[BUF_SIZE];
11.
12. int main(int argc, char *argv[])
13. {
14.     WSADATA wsaData;
15.     SOCKET hSock;
16.     SOCKADDR_IN servAdr;
17.     HANDLE hSndThread, hRcvThread;
18.     if(argc!=4) {
19.         printf("Usage : %s <IP> <port> <name>\n", argv[0]);
20.         exit(1);
21.     }
22.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
23.         ErrorHandling("WSAStartup() error!");
24.
25.     sprintf(name, "[%s]", argv[3]);
26.     hSock=socket(PF_INET, SOCK_STREAM, 0);
27.
28.     memset(&servAdr, 0, sizeof(servAdr));
29.     servAdr.sin_family=AF_INET;
30.     servAdr.sin_addr.s_addr=inet_addr(argv[1]);
31.     servAdr.sin_port=htons(atoi(argv[2]));
32.
33.     if(connect(hSock, (SOCKADDR*)&servAdr, sizeof(servAdr))==SOCKET_ERROR)
34.         ErrorHandling("connect() error");
```

```
35.
36.     hSndThread=
37.         (HANDLE)_beginthreadex(NULL, 0, SendMsg, (void*)&hSock, 0, NULL);
38.     hRcvThread=
39.         (HANDLE)_beginthreadex(NULL, 0, RecvMsg, (void*)&hSock, 0, NULL);
40.
41.     WaitForSingleObject(hSndThread, INFINITE);
42.     WaitForSingleObject(hRcvThread, INFINITE);
43.     closesocket(hSock);
44.     WSACleanup();
45.     return 0;
46. }
47.
48. unsigned WINAPI SendMsg(void * arg)      // send thread main
49. {
50.     SOCKET hSock=*((SOCKET*)arg);
51.     char nameMsg[NAME_SIZE+BUF_SIZE];
52.     while(1)
53.     {
54.         fgets(msg, BUF_SIZE, stdin);
55.         if(!strcmp(msg,"q\n")||!strcmp(msg,"Q\n"))
56.         {
57.             closesocket(hSock);
58.             exit(0);
59.         }
60.         sprintf(nameMsg,"%s %s", name, msg);
61.         send(hSock, nameMsg, strlen(nameMsg), 0);
62.     }
63.     return 0;
64. }
65.
66. unsigned WINAPI RecvMsg(void * arg)      // read thread main
67. {
68.     int hSock=*((SOCKET*)arg);
69.     char nameMsg[NAME_SIZE+BUF_SIZE];
70.     int strLen;
71.     while(1)
72.     {
73.         strLen=recv(hSock, nameMsg, NAME_SIZE+BUF_SIZE-1, 0);
74.         if(strLen==-1)
75.             return -1;
76.         nameMsg[strLen]=0;
77.         fputs(nameMsg, stdout);
78.     }
79.     return 0;
80. }
81.
82. void ErrorHandling(char *msg)
83. {
84.     //与示例chat_serv_clnt.c的ErrorHandling一致。
85. }
```

运行结果同样与chat_serv.c、chat_clnt.c的运行结果相同，故省略。之前已省略了不少内容，

我对此感到很抱歉。这是为了减少多余或重复内容而做出的决定，希望各位理解。

20.4 习题

(1) 关于Windows操作系统的用户模式和内核模式的说法中正确的是?

a. 用户模式是应用程序运行的基本模式，虽然访问的内存空间没有限制，但无法访问物理设备。

b. 应用程序运行过程中绝对不会进入内核模式。应用程序只在用户模式中运行。

c. Windows为了有效使用内存空间，分别定义了用户模式和内核模式。

d. 应用程序运行过程中也有可能切换到内核模式。只是切换到内核模式后，进程将一直保持该状态。

(2) 判断下列关于用户模式同步和内核模式同步描述的正误。

- ❑ 用户模式的同步中不会切换到内核模式。即非操作系统级别的同步。( )
- ❑ 内核模式的同步是由操作系统提供的功能，比用户模式同步提供更多功能。( )
- ❑ 需要在用户模式和内核模式之间切换，这是内核模式同步的缺点。( )
- ❑ 除特殊情况外，原则上应使用内核模式同步。用户模式同步是操作系统提供内核模式同步机制前使用的同步方法。( )

(3) 本章示例SyncSema_win.c的Read函数中，退出临界区需要较长时间，请给出解决方案并实现。

(4) 请将本章SyncEvent_win.c示例改为基于信号量的同步方式，并得出相同运行结果。

done!

第 21 章

异步通知I/O模型

21

从本章开始将分 3 个章节介绍 Windows 提供的扩展 I/O 模型。第 21 章~第 23 章之间的联系非常紧密，不能理解本章内容就无法掌握其他章节，希望各位认真学习这几章。

21.1 理解异步通知 I/O 模型

各位应该还记得之前介绍过的select函数，它是实现并发服务器端的方法之一。本章内容可以理解为select模型的改进方式。

理解同步和异步

首先解释“异步”（Asynchronous）的含义。异步主要指“不一致”，它在数据I/O中非常有用。之前的Windows示例中主要通过send & recv函数进行同步I/O。调用send函数时，完成数据传输后才能从函数返回（确切地说，只有把数据完全传输到输出缓冲后才能返回）；而调用recv函数时，只有读到期望大小的数据后才能返回。因此，相当于同步方式的I/O处理。

“究竟哪些部分是同步的？”

各位或许有这种疑问，但我想反问大家：“哪些部分进行了同步处理？”同步的关键是函数的调用及返回时刻，以及数据传输的开始和完成时刻。

“调用send函数的瞬间开始传输数据，send函数执行完（返回）的时刻完成数据传输。”

“调用recv函数的瞬间开始接收数据，recv函数执行完（返回）的时刻完成数据接收。”

可以通过图21-1解释上述两句话的含义（上述语句和图中的“完成传输”都是指数据完全传输到输出缓冲）。

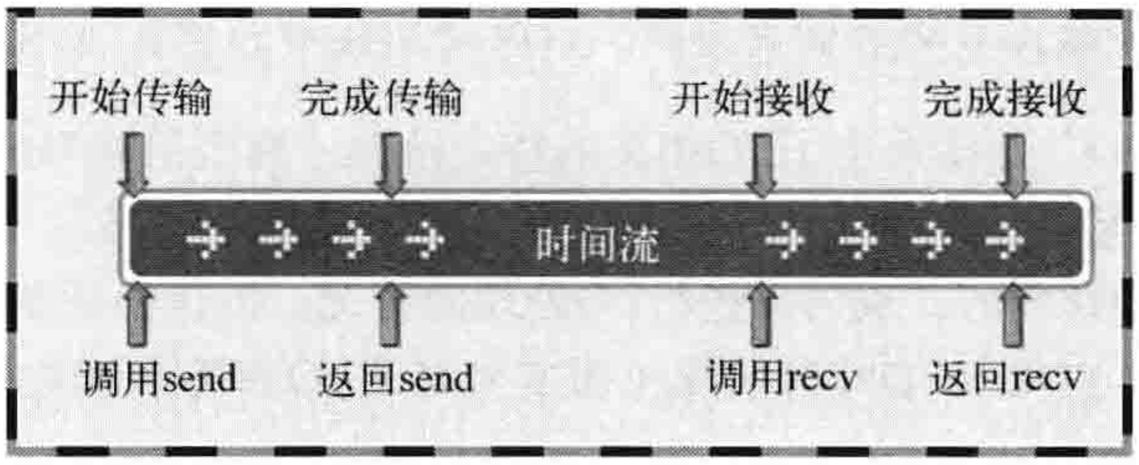


图21-1 调用同步的I/O函数

相信各位能够通过上述图文理解同步的关键所在。那异步I/O的含义又是什么呢？图21-2给出解释，希望大家与图21-1进行对比。

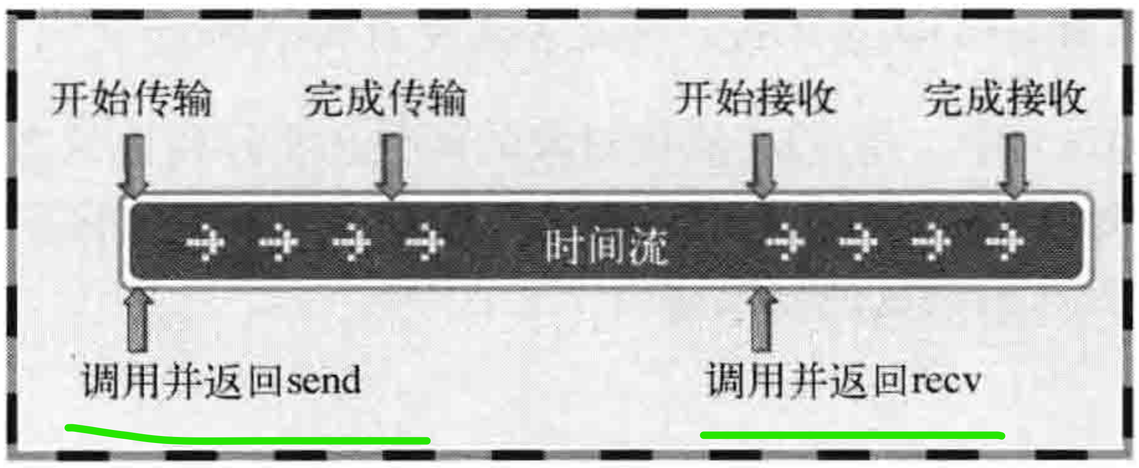


图21-2 调用异步I/O函数

从图21-2中可以看到，异步I/O是指I/O函数的返回时刻与数据收发的完成时刻不一致。如此看来，我们接触过异步I/O。如果记不清这些内容，可以回顾第17章epoll的异步I/O部分。

同步 I/O 的缺点及异步方式的解决方案

异步I/O就是为了克服同步I/O的缺点而设计的模型。同步I/O有哪些缺点？异步方式又是如何解决的呢？其实，第17章的最后部分“条件触发和边缘触发孰优孰劣”中给出过答案。各位可能因为忘记这些内容而感到沮丧，考虑到这一点，我将以不同的、更简单的方式解释。从图21-1中很容易找到同步I/O的缺点：“进行I/O的过程中函数无法返回，所以不能执行其他任务！”而图21-2中，无论数据是否完成交换都返回函数，这就意味着可以执行其他任务。所以说“异步方式能够比同步方式更有效地使用CPU”。

理解异步通知 I/O 模型

之前分析了同步和异步方式的I/O函数，确切地说，分析了同步和异步方式下I/O函数返回时间点的差异。下面我希望扩展讨论的对象（同步和异步并不局限于I/O）。

本章题目为“异步通知I/O模型”，意为“通知I/O”是以异步方式工作的。首先了解一下“通知I/O”的含义：

“通知输入缓冲收到数据并需要读取，以及输出缓冲为空故可以发送数据。”

顾名思义，“通知I/O”是指发生了I/O相关的特定情况。典型的通知I/O模型是select方式。还记得select监视的3种情况吗？其中具代表性的就是“收到数据的情况”。select函数就是从返回调用的函数时通知需要I/O处理的，或可以进行I/O处理的情况。但这种通知是以同步方式进行的，原因在于，需要I/O或可以进行I/O的时间点（简言之就是I/O相关事件发生的时间点）与select函数的返回时间点一致。

相信各位已理解通知I/O模型的含义。与“select函数只在需要或可以进行I/O的情况下返回”不同，异步通知I/O模型中函数的返回与I/O状态无关。本章的WSAEventSelect函数就是select函数的异步版本。

“既然函数的返回与I/O状态无关，那是否需要监视I/O状态变化？”

当然需要！异步通知I/O中，指定I/O监视对象的函数和实际验证状态变化的函数是相互分离的。因此，指定监视对象后可以离开执行其他任务，最后再回来验证状态变化。以上就是通知I/O的所有理论，下面通过具体函数实现该模型。

提　示

select 函数中也可以设置超时时间

设置超时时间可以在未发生 I/O 状态变化的情况下防止函数阻塞，所以可编写类似异步方式的代码。但之后为了验证 I/O 的状态变化需要再次集中句柄（文件描述符），以便再次调用 select 函数。换言之，select 函数默认为同步方式的通知 I/O 模型，只是为了弥补缺点才定义为可设置超时的方式。

21.2　理解和实现异步通知 I/O 模型

异步通知I/O模型的实现方法有2种：一种是使用本书介绍的WSAEventSelect函数，另一种是使用WSAAsyncSelect函数。使用WSAAsyncSelect函数时需要指定Windows句柄以获取发生的事件（UI相关内容），因此本书不会涉及，但大家要知道有这个函数。

WSAEventSelect 函数和通知

如前所述，告知I/O状态变化的操作就是“通知”。I/O的状态变化可以分为不同情况。

- 套接字的状态变化：套接字的I/O状态变化。
- 发生套接字相关事件：发生套接字I/O相关事件。

这2种情况都意味着发生了需要或可以进行I/O的事件，我将根据上下文适当混用这些概念。

首先介绍WSAEventSelect函数，该函数用于指定某一套接字为事件监视对象。

```
#include <winsock2.h>

int WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents);
```

➔ 成功时返回0，失败时返回SOCKET_ERROR。

- s　监视对象的套接字句柄。
- hEventObject　传递事件对象句柄以验证事件发生与否。
- lNetworkEvents　希望监视的事件类型信息。

传入参数s的套接字内只要发生lNetworkEvents中指定的事件之一，WSAEventSelect函数就将hEventObject句柄所指内核对象改为signaled状态。因此，该函数又称“连接事件对象和套接字的函数”。

另外一个重要的事实是，无论事件发生与否，WSAEventSelect函数调用后都会直接返回，所以可以执行其他任务。也就是说，该函数以异步通知方式工作。下面介绍作为该函数第三个参数的事件类型信息，可以通过位或运算同时指定多个信息。

- FD_READ：是否存在需要接收的数据？
- FD_WRITE：能否以非阻塞方式传输数据？
- FD_OOB：是否收到带外数据？
- FD_ACCEPT：是否有新的连接请求？
- FD_CLOSE：是否有断开连接的请求？

以上就是WSAEventSelect函数的调用方法。各位或许有如下疑问（很好的问题）：

“啊？select函数可以针对多个套接字对象调用，但WSAEventSelect函数只能针对1个套接字对象调用！”

的确，仅从概念上看，WSAEventSelect函数的功能偏弱。但使用该函数时，没必要针对多个套接字进行调用。从select函数返回时，为了验证事件的发生需要再次针对所有句柄（文件描述符）调用函数，但通过调用WSAEventSelect函数传递的套接字信息已注册到操作系统，所以无需再次调用。这反而是WSAEventSelect函数比select函数的优势所在。

提　示

epoll 和 WSAEventSelect

如上所述，无需针对已注册的套接字再次调用 WSAEventSelect 函数，这种特性在 Linux 的 epoll 中首次介绍过。回顾这部分内容可以更详细地了解这种方式带来的好处。

从前面关于WSAEventSelect函数的说明中可以看出，需要补充如下内容。

- WSAEventSelect函数的第二个参数中用到的事件对象的创建方法。
- 调用WSAEventSelect函数后发生事件的验证方法。
- 验证事件发生后事件类型的查看方法。

上述过程中只要插入WSAEventSelect函数的调用就与服务器端的实现过程完全一致，下面分别讲解。

manual-reset 模式事件对象的其他创建方法

我们之前利用CreateEvent函数创建了事件对象。CreateEvent函数在创建事件对象时，可以在auto-reset模式和manual-reset模式中任选其一。但我们只需要manual-reset模式non-signaled状态的事件对象，所以利用如下函数创建较为方便。

```
#include <winsock2.h>

WSAEVENT WSACreateEvent(void);
```

➔ 成功时返回事件对象句柄，失败时返回 WSA_INVALID_EVENT。

上述声明中返回类型WSAEVENT的定义如下：

```
#define WSAEVENT HANDLE
```

实际上就是我们熟悉的内核对象句柄，这一点需要注意。另外，为了销毁通过上述函数创建的事件对象，系统提供了如下函数。

```
#include <winsock2.h>

BOOL WSACloseEvent(WSAEVENT hEvent);
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

验证是否发生事件

既然介绍了WSACreateEvent函数，那调用WSAEventSelect函数应该不成问题。接下来就要考虑调用WSAEventSelect函数后的处理。为了验证是否发生事件，需要查看事件对象。完成该任务的函数如下，除了多1个参数外，其余部分与WaitForMultipleObjects函数完全相同。

```
#include <winsock2.h>

DWORD WSAWaitForMultipleEvents(
    DWORD cEvents, const WSAEVENT * lphEvents, BOOL fWaitAll, DWORD dwTimeout,
BOOL fAlertable);
```

➔ 成功时返回发生事件的对象信息，失败时返回 WSA_INVALID_EVENT。

- cEvents　需要验证是否转为signaled状态的事件对象的个数。
- lphEvents　存有事件对象句柄的数组地址值。
- fWaitAll　传递TRUE时，所有事件对象在signaled状态时返回；传递FALSE时，只要其中1个变为signaled状态就返回。
- dwTimeout　以1/1000秒为单位指定超时，传递WSA_INFINITE时，直到变为signaled状态时才会返回。
- fAlertable　传递TRUE时进入alertable wait（可警告等待）状态（第22章）。
- 返回值　返回值减去常量WSA_WAIT_EVENT_0时，可以得到转变为signaled状态的事件对象句柄对应的索引，可以通过该索引在第二个参数指定的数组中查找句柄。如果有多个事件对象变为signaled状态，则会得到其中较小的值。发生超时将返回WAIT_TIMEOUT。

由于发生套接字事件，事件对象转为signaled状态后该函数才返回，所以它非常有利于确认事件发生与否。但由于最多可传递64个事件对象，如果需要监视更多句柄，就只能创建线程或扩展保存句柄的数组，并多次调用上述函数。

提　示

最大句柄数

可通过以宏的方式声明的 WSA_MAXIMUM_WAIT_EVENTS 常量得知 WSAWaitForMultipleEvents 函数可以同时监视的最大事件对象数。该常量值为 64，所以最大句柄数为 64 个。但可以更改这一限制，日后发布新版本的操作系统时可能更改该常量。

对于WSAWaitForMultipleEvents函数，各位可能产生如下疑问：

“WSAWaitForMultipleEvents函数如何得到转为signaled状态的所有事件对象句柄的信息？”

答案是：只通过1次函数调用无法得到转为signaled状态的所有事件对象句柄的信息。通过该函数可以得到转为signaled状态的事件对象中的第一个（按数组中的保存顺序）索引值。但可以利用“事件对象为manual-reset模式”的特点，通过如下方式获得所有signaled状态的事件对象。

```
int posInfo, startIdx, i;
. . . . . .
```

```
posInfo = WSAWaitForMultipleEvents(numOfSock, hEventArray, FALSE, WSA_INFINITE,
FALSE);
startIdx = posInfo - WSA_WAIT_EVENT_0;
. . . . . .
for(i = startIdx; i < numOfSock; i++)
{
    int sigEventIdx = WSAWaitForMultipleEvents(1, &hEventArray[i], TRUE, 0, FALSE);
    . . . . . .
}
```

注意观察上述代码中的循环。循环中从第一个事件对象到最后一个事件对象逐一依序验证是否转为signaled状态（超时信息为0，所以调用函数后立即返回）。之所以能做到这一点，完全是因为事件对象为manual-reset模式，这也解释了为何在异步通知I/O模型中事件对象必须为manual-reset模式。

区分事件类型

既然已经通过WSAWaitForMultipleEvents函数得到了转为signaled状态的事件对象，最后就要确定相应对象进入signaled状态的原因。为完成该任务，我们引入如下函数。调用此函数时，不仅需要signaled状态的事件对象句柄，还需要与之连接的（由WSAEventSelect函数调用引发的）发生事件的套接字句柄。

```
#include <winsock2.h>

int WSAEnumNetworkEvents(
    SOCKET s, WSAEVENT hEventObject, LPWSANETWORKEVENTS lpNetworkEvents);
```

➔ 成功时返回0，失败时返回SOCKET_ERROR。

- s　发生事件的套接字句柄。
- hEventObject　与套接字相连的（由WSAEventSelect函数调用引发的）signaled状态的事件对象句柄。
- lpNetworkEvents　保存发生的事件类型信息和错误信息的WSANETWORKEVENTS结构体变量地址值。

上述函数将manual-reset模式的事件对象改为non-signaled状态，所以得到发生的事件类型后，不必单独调用ResetEvent函数。下面介绍与上述函数有关的WSANETWORKEVENTS结构体。

```
typedef struct _WSANETWORKEVENTS
{
    long lNetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
```

```
} WSANETWORKEVENTS, * LPWSANETWORKEVENTS;
```

上述结构体的lNetworkEvents成员将保存发生的事件信息。与WSAEventSelect函数的第三个参数相同，需要接收数据时，该成员为FD_READ；有连接请求时，该成员为FD_ACCEPT。因此，可通过如下方式查看发生的事件类型。

```
WSANETWORKEVENTS netEvents;
. . . . .
WSAEnumNetworkEvents(hSock, hEvent, &netEvents);
if(netEvents.lNetworkEvents & FD_ACCEPT)
{
    //FD_ACCEPT 事件的处理
}
if(netEvents.lNetworkEvents & FD_READ)
{
    //FD_READ 事件的处理
}
if(netEvents.lNetworkEvents & FD_CLOSE)
{
    //FD_CLOSE 事件的处理
}
```

另外，错误信息将保存到声明为成员的iErrorCode数组（发生错误的原因可能很多，因此用数组声明）。验证方法如下。

- 如果发生FD_READ相关错误，则在iErrorCode[FD_READ_BIT]中保存除0以外的其他值。
- 如果发生FD_WRITE相关错误，则在iErrorCode[FD_WRITE_BIT]中保存除0以外的其他值。

可通过如下描述理解上述内容。

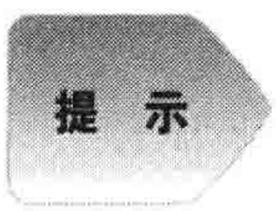
“如果发生FD_XXX相关错误，则在iErrorCode[FD_XXX_BIT]中保存除0以外的其他值”

因此可以用如下方式检查错误。

```
WSANETWORKEVENTS netEvents;
. . . . .
WSAEnumNetworkEvents(hSock, hEvent, &netEvents);
. . . . .
if(netEvents.iErrorCode[FD_READ_BIT] != 0)
{
```

```
    //发生 FD_READ 事件相关错误
}
```

以上就是异步通知I/O模型的全部内容，下面利用这些知识编写示例。



本来就是较难理解的部分

理解并应用异步通知 I/O 模型是很难的，也许各位在阅读过程中没少叹气。函数调用关系也较为复杂，而且需要理解更多内容。但这是每个人都需要经历的过程，希望大家放松心态，经过一段时间后就能完全掌握。

利用异步通知 I/O 模型实现回声服务器端

下面要介绍的回声服务器端代码相对偏长，所以将分为几个部分逐个介绍。

❖ AsynNotiEchoServ_win.c One

```
#include <stdio.h>
#include <string.h>
#include <winsock2.h>

#define BUF_SIZE 100

void CompressSockets(SOCKET hSockArr[], int idx, int total);
void CompressEvents(WSAEVENT hEventArr[], int idx, int total);
void ErrorHandling(char *msg);

int main(int argc, char *argv[])
{
    WSADATA wsaData;
    SOCKET hServSock, hClntSock;
    SOCKADDR_IN servAdr, clntAdr;

    SOCKET hSockArr[WSA_MAXIMUM_WAIT_EVENTS];
    WSAEVENT hEventArr[WSA_MAXIMUM_WAIT_EVENTS];
    WSAEVENT newEvent;
    WSANETWORKEVENTS netEvents;

    int numOfClntSock=0;
    int strLen, i;
    int posInfo, startIdx;
    int clntAdrLen;
    char msg[BUF_SIZE];

    if(argc!=2) {
```

```
        printf("Usage: %s <port>\n", argv[0]);
        exit(1);
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
        ErrorHandling("WSAStartup() error!");
```

首先是一些通用的声明和初始化部分，这不需要特别说明，只不过是把稍后用到的变量集中到了一起。下面介绍后续代码。

❖ AsynNotiEchoServ_win.c Two

```
hServSock=socket(PF_INET, SOCK_STREAM, 0);
memset(&servAdr, 0, sizeof(servAdr));
servAdr.sin_family=AF_INET;
servAdr.sin_addr.s_addr=htonl(INADDR_ANY);
servAdr.sin_port=htons(atoi(argv[1]));

if(bind(hServSock, (SOCKADDR*) &servAdr, sizeof(servAdr))==SOCKET_ERROR)
    ErrorHandling("bind() error");

if(listen(hServSock, 5)==SOCKET_ERROR)
    ErrorHandling("listen() error");

newEvent=WSACreateEvent();
if(WSAEventSelect(hServSock, newEvent, FD_ACCEPT)==SOCKET_ERROR)
    ErrorHandling("WSAEventSelect() error");

hSockArr[numOfClntSock]=hServSock;
hEventArr[numOfClntSock]=newEvent;
numOfClntSock++;
```

上述代码创建了用于接收客户端连接请求的服务器端套接字（监听套接字）。为了完成监听任务，针对FD_ACCEPT事件调用了WSAEventSelect函数。此处需要注意如下2条语句。

```
hSockArr[numOfClntSock] = hServSock;
hEventArr[numOfClntSock] = newEvent;
```

这段代码把通过WSAEventSelect函数连接的套接字和事件对象的句柄分别存入hSockArr和hEventArr数组，但应该维持它们之间的关系。也就是说，应该可以通过hSockArr[idx]找到连接到套接字的事件对象，反之，也可以通过hEventArr[idx]找到连接到事件对象的套接字。因此，该示例将套接字和事件对象句柄保存到数组时统一了保存位置。也就有了下列公式（程序中应该遵守该公式）。

- 与hSockArr[n]中的套接字相连的事件对象应保存到hEventArr[n]。
- 与hEventArr[n]中的事件对象相连的套接字应保存到hSockArr[n]。

下面介绍后续的while循环部分，之前学习的大部分知识都集中于此。特别是验证转为signaled

状态的事件对象句柄的方法、查看发生事件类型的方法、检查错误的方法等，希望各位在此基础上分析下列代码。

❖ AsynNotiEchoServ_win.c Three

```
while(1)
{
    posInfo=WSAWaitForMultipleEvents(
        numOfClntSock, hEventArr, FALSE, WSA_INFINITE, FALSE);
    startIdx=posInfo-WSA_WAIT_EVENT_0;

    for(i=startIdx; i<numOfClntSock; i++)
    {
        int sigEventIdx=
            WSAWaitForMultipleEvents(1, &hEventArr[i], TRUE, 0, FALSE);
        if((sigEventIdx==WSA_WAIT_FAILED || sigEventIdx==WSA_WAIT_TIMEOUT))
        {
            continue;
        }
        else
        {
            sigEventIdx=i;
            WSAEnumNetworkEvents(
                hSockArr[sigEventIdx], hEventArr[sigEventIdx], &netEvents);
            if(netEvents.lNetworkEvents & FD_ACCEPT) //请求连接时
            {
                if(netEvents.iErrorCode[FD_ACCEPT_BIT]!=0)
                {
                    puts("Accept Error");
                    break;
                }
                clntAdrLen=sizeof(clntAdr);
                hClntSock=accept(
                    hSockArr[sigEventIdx], (SOCKADDR*)&clntAdr, &clntAdrLen);
                newEvent=WSACreateEvent();
                WSAEventSelect(hClntSock, newEvent, FD_READ|FD_CLOSE);

                hEventArr[numOfClntSock]=newEvent;
                hSockArr[numOfClntSock]=hClntSock;
                numOfClntSock++;
                puts("connected new client...");
            }

            if(netEvents.lNetworkEvents & FD_READ) // 接收数据时
            {
                if(netEvents.iErrorCode[FD_READ_BIT]!=0)
                {
                    puts("Read Error");
                    break;
                }
                strLen=recv(hSockArr[sigEventIdx], msg, sizeof(msg), 0);
                send(hSockArr[sigEventIdx], msg, strLen, 0);
```

```
            }

            if(netEvents.lNetworkEvents & FD_CLOSE) // 断开连接时
            {
                if(netEvents.iErrorCode[FD_CLOSE_BIT]!=0)
                {
                    puts("Close Error");
                    break;
                }
                WSACloseEvent(hEventArr[sigEventIdx]);
                closesocket(hSockArr[sigEventIdx]);

                numOfClntSock--;
                CompressSockets(hSockArr, sigEventIdx, numOfClntSock);
                CompressEvents(hEventArr, sigEventIdx, numOfClntSock);
            }
        }
    }
    WSACleanup();
    return 0;
} // end of main function
```

最后给出上述代码中调用的2个函数CompressSockets和CompressEvents。

❖ AsynNotiEchoServ_win.c Four

```
void CompressSockets(SOCKET hSockArr[], int idx, int total)
{
    int i;
    for(i=idx; i<total; i++)
        hSockArr[i]=hSockArr[i+1];
}

void CompressEvents(WSAEVENT hEventArr[], int idx, int total)
{
    int i;
    for(i=idx; i<total; i++)
        hEventArr[i]=hEventArr[i+1];
}

void ErrorHandling(char *msg)
{
    fputs(msg, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

断开连接并从数组中删除套接字及与之相连的事件对象时调用上述2个函数（以Compress…开头），它们主要用于填充数组空间，只有同时调用才能维持套接字和事件对象之间的关系。

既然分析了所有代码，本应给出运行结果，但因其与之前的回声服务器端/客户端并无差异，故省略。另外，上述示例可以与任意回声客户端配合运行，各位可以选择Windows平台下的回声客户端作为配套程序。

21.3　习题

(1) 结合send & recv函数解释同步和异步方式的I/O。并请说明同步I/O的缺点，以及怎样通过异步I/O进行解决。

(2) 异步I/O并不是所有情况下的最佳选择。它具有哪些缺点？何种情况下同步I/O更优？可以参考异步I/O相关源代码，亦可结合线程进行说明。

(3) 判断下列关于select模型描述的正误。

- ❑ select模型通过函数的返回值通知I/O相关事件，故可视为通知I/O模型。(　　)
- ❑ select模型中I/O相关事件的发生时间点和函数返回的时间点一致，故不属于异步模型。(　　)
- ❑ WSAEventSelect函数可视为select方式的异步模型，因为该函数的I/O相关事件的通知方式为异步方式。(　　)

(4) 请从源代码的角度说明select函数和WSAEventSelect函数在使用上的差异。

(5) 第17章的epoll可以在条件触发和边缘触发这2种方式下工作。哪种方式更适合异步I/O模型？为什么？请概括说明。

(6) Linux中的epoll同样属于异步I/O模型。请说明原因。

(7) 如何获取WSAWaitForMultipleEvents函数可以监视的最大句柄数？请编写代码读取该值。

(8) 为何异步通知I/O模型中的事件对象必须是manual-reset模式？

(9) 请在本章的通知I/O模型的基础上编写聊天服务器端。要求该服务器端能够结合第20章的聊天客户端chat_clnt_win.c运行。

done!

第22章

重叠I/O模型

本章的重叠I/O与异步有很大关系，若各位尚未完全理解异步的含义，请尽快复习，并掌握不同环境下区分同步和异步的能力。

22.1 理解重叠 I/O 模型

第21章异步处理的并非I/O，而是“通知”。本章讲解的才是以异步方式处理I/O的方法。只有理解了二者的区别和各自的优势，才能更轻松地学习第23章的IOCP。

重叠 I/O

其实各位对于重叠I/O并不陌生。大家已经掌握了异步I/O。我通过图21-2说明过异步I/O模型，实际上，这种异步I/O就相当于重叠I/O。下面我将讲解重叠I/O，各位可自行判断二者是否相似。图22-1给出重叠I/O的原理。

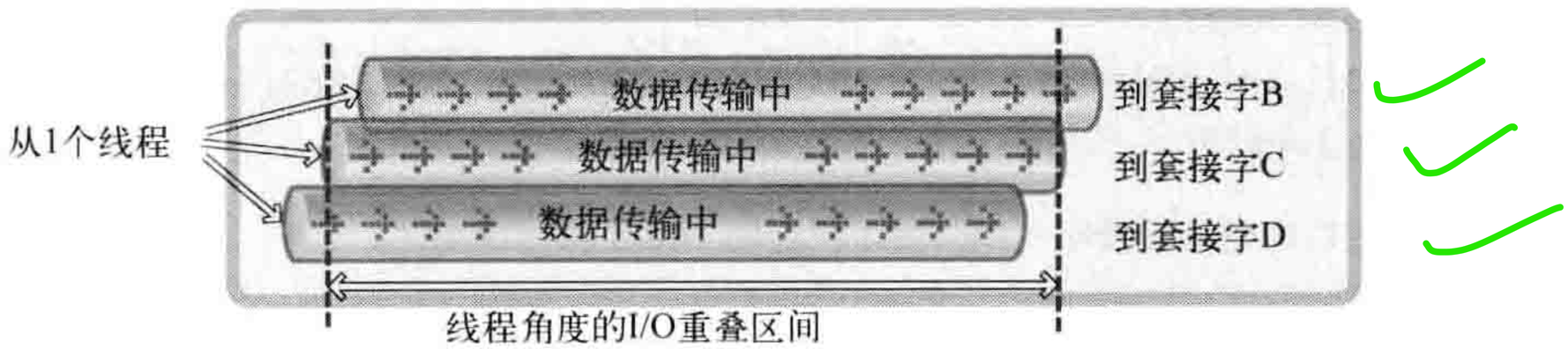


图22-1 重叠I/O模型

如图22-1所示，同一线程内部向多个目标传输（或从多个目标接收）数据引起的I/O重叠现象称为“重叠I/O”。为了完成这项任务，调用的I/O函数应立即返回，只有这样才能发送后续数据。从结果上看，利用上述模型收发数据时，最重要的前提条件就是异步I/O。而且，为了完成异步I/O，调用的I/O函数应以非阻塞模式工作。

接下来的判断交给各位。异步I/O和重叠I/O之间是否存在差异众说纷纭，关键是要理解二者

关系。异步方式进行I/O处理时，即使不采用本章介绍的方式，也可以通过其他方法构造如图22-1所示的I/O处理方式。因此，我认为不用明确区分。

本章讨论的重叠 I/O 的重点不在于 I/O

前面对异步I/O和重叠I/O进行了比较，这些内容看似是本章的全部理论说明，但其实还未进入重叠I/O的正题。因为Windows中重叠I/O的重点并非I/O本身，而是如何确认I/O完成时的状态。不管是输入还是输出，只要是非阻塞模式的，就要另外确认执行结果。关于这种确认方法我们还一无所知。确认执行结果前需要经过特殊的处理过程，这就是本章要讲述的内容。Windows中的重叠I/O不仅包含图22-1所示的I/O（这是基础），还包含确认IO完成状态的方法。

提　示

后文中的重叠 I/O

后面提到的“重叠 I/O”不仅包含图 22-1 中的 I/O 模型（再次强调，这是基础），还包含确认 I/O 完成状态的方法，同时也指 Windows 平台下的重叠 I/O 模型。

创建重叠 I/O 套接字

首先要创建适用于重叠I/O的套接字，可以通过如下函数完成。

```
#include <winsock2.h>

SOCKET WSASocket(
    int af, int type, int protocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g,
DWORD dwFlags);
```

➔ 成功时返回套接字句柄，失败时返回 INVALID_SOCKET。

af	协议族信息。
type	套接字数据传输方式。
protocol	2个套接字之间使用的协议信息。
lpProtocolInfo	包含创建的套接字信息的WSAPROTOCOL_INFO结构体变量地址值，不需要时传递NULL。
g	为扩展函数而预约的参数，可以使用0。
dwFlags	套接字属性信息。

各位对前3个参数比较熟悉，第四个和第五个参数与目前的工作无关，可以简单设置为NULL和0。可以向最后一个参数传递WSA_FLAG_OVERLAPPED，赋予创建出的套接字重叠I/O特性。总之，可以通过如下函数调用创建出可以进行重叠I/O的非阻塞模式的套接字。

```
WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
```

执行重叠 I/O 的 WSASend 函数

创建出具有重叠I/O属性的套接字后，接下来2个套接字（服务器端/客户端之间的）连接过程与一般的套接字连接过程相同，但I/O数据时使用的函数不同。先介绍重叠I/O中使用的数据输出函数。

```
#include <winsock2.h>

int WSASend(
    SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent, DWORD dwFlags, LPWSAOVERLAPPED
lpOverlapped,LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

s	套接字句柄，传递具有重叠I/O属性的套接字句柄时，以重叠I/O模型输出。
lpBuffers	WSABUF结构体变量数组的地址值，WSABUF中存有待传输数据。
dwBufferCount	第二个参数中数组的长度。
lpNumberOfBytesSent	用于保存实际发送字节数的变量地址值（稍后进行说明）。
dwFlags	用于更改数据传输特性，如传递MSG_OOB时发送OOB模式的数据。
lpOverlapped	WSAOVERLAPPED结构体变量的地址值，使用事件对象，用于确认完成数据传输。
lpCompletionRoutine	传入Completion Routine函数的入口地址值，可以通过该函数确认是否完成数据传输。

接下来介绍上述函数的第二个结构体参数类型，该结构体中存有待传输数据的地址和大小等信息。

```
typedf struct __WSABUF
{
    u_long len;     // 待传输数据的大小
    char FAR * buf; // 缓冲地址值
} WSABUF, * LPWSABUF;
```

下面给出上述函数的调用示例。利用上述函数传输数据时可以按如下方式编写代码。

```
WSAEVENT event;
```

```
WSAOVERLAPPED overlapped;
WSABUF dataBuf;
char buf[BUF_SIZE] = {"待传输的数据"};
int recvBytes = 0;
. . . . .
event = WSACreateEvent();
memset(&overlapped, 0, sizeof(overlapped)); // 所有位初始化为 0!
overlapped.hEvent = event;
dataBuf.len = sizeof(buf);
dataBuf.buf = buf;
WSASend(hSocket, &dataBuf, 1, &recvBytes, 0, &overlapped, NULL);
. . . . .
```

调用WSASend函数时将第三个参数设置为1，因为第二个参数中待传输数据的缓冲个数为1。另外，多余参数均设置为NULL或0，其中需要注意第六个和第七个参数（稍后将具体解释，现阶段只需留意即可）。第六个参数中的WSAOVERLAPPED结构体定义如下。

```
typedef struct _WSAOVERLAPPE
{
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, * LPWSAOVERLAPPED;
```

Internal、InternalHigh成员是进行重叠I/O时操作系统内部使用的成员，而Offset、OffsetHigh同样属于具有特殊用途的成员。所以各位实际只需要关注hEvent成员，稍后将介绍该成员的使用方法。

“为了进行重叠I/O，WSASend函数的lpOverlapped参数中应该传递有效的结构体变量地址值，而不是NULL。”

如果向lpOverlapped传递NULL，WSASend函数的第一个参数中的句柄所指的套接字将以阻塞模式工作。还需要了解以下这个事实，否则也会影响开发。

“利用WSASend函数同时向多个目标传输数据时，需要分别构建传入第六个参数的WSAOVERLAPPED结构体变量。”

这是因为，进行重叠I/O的过程中，操作系统将使用WSAOVERLAPPED结构体变量。

关于 WSASend 再补充一点

前面谈到，通过WSASend函数的lpNumberOfBytesSent参数可以获得实际传输的数据大小。各位关于这一点不感到困惑吗？

“既然WSASend函数在调用后立即返回，那如何得到传输的数据大小呢？”

实际上，WSASend函数调用过程中，函数返回时间点和数据传输完成时间点并非总不一致。如果输出缓冲是空的，且传输的数据并不大，那么函数调用后可以立即完成数据传输。此时，WSASend函数将返回0，而lpNumberOfBytesSent中将保存实际传输的数据大小的信息。反之，WSASend函数返回后仍需要传输数据时，将返回SOCKET_ERROR，并将WSA_IO_PENDING注册为错误代码，该代码可以通过WSAGetLastError函数（稍后再介绍）得到。这时应该通过如下函数获取实际传输的数据大小。

```
#include <winsock2.h>

BOOL WSAGetOverlappedResult(
  SOCKET s, LPWSAOVERLAPPED lpOverlapped, LPDWORD lpcbTransfer, BOOL fWait,
LPDWORD lpdwFlags);
```

➔ 成功时返回 TRUE，失败时返回 FALSE。

s	进行重叠I/O的套接字句柄。
lpOverlapped	进行重叠I/O时传递的WSAOVERLAPPED结构体变量的地址值。
lpcbTransfer	用于保存实际传输的字节数的变量地址值。
fWait	如果调用该函数时仍在进行I/O，fWait为TRUE时等待I/O完成，fWait为FALSE时将返回FALSE并跳出函数。
lpdwFlags	调用WSARecv函数时，用于获取附加信息（例如OOB消息）。如果不需要，可以传递NULL。

通过此函数不仅可以获取数据传输结果，还可以验证接收数据的状态。如果给出示例前进行过多理论说明会使人感到乏味，所以稍后将通过示例讲解此函数的使用方法。

进行重叠 I/O 的 WSARecv 函数

有了WSASend函数的基础，WSARecv函数将不难理解。因为它们大同小异，只是在功能上有接收和传输之分。

```
#include <winsock2.h>

int WSARecv(
    SOCKETS, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

➔ 成功时返回 0，失败时返回 SOCKET_ERROR。

s	赋予重叠I/O属性的套接字句柄。
lpBuffers	用于保存接收数据的WSABUF结构体数组地址值。
dwBufferCount	向第二个参数传递的数组的长度。
lpNumberOfBytesRecvd	保存接收的数据大小信息的变量地址值。
lpFlags	用于设置或读取传输特性信息。
lpOverlapped	WSAOVERLAPPED结构体变量地址值。
lpCompletionRoutine	Completion Routine函数地址值。

关于上述函数的使用方法将同样结合示例进行说明。

以上就是重叠I/O中的数据I/O方法，下一节将介绍I/O完成及如何确认结果。

知识补给站　Gather/Scatter I/O

Gather/Scatter I/O是指，将多个缓冲中的数据累积到一定程度后一次性传输（Gather输出），将接收的数据分批保存（Scatter输入）。第13章的writev & readv函数具有Gather/Scatter I/O功能，但Windows下并没有这些函数的定义。不过，可以通过重叠I/O中的WSASend和WSARecv函数获得类似功能。刚才讲了这2个函数，从它们的第二个和第三个参数中可以判断其具有Gather/Scatter I/O功能。

22.2　重叠 I/O 的 I/O 完成确认

重叠I/O中有2种方法确认I/O的完成并获取结果。

- 利用WSASend、WSARecv函数的第六个参数，基于事件对象。
- 利用WSASend、WSARecv函数的第七个参数，基于Completion Routine。

只有理解了这2种方法，才能算是掌握了重叠I/O（其实比22.1节更重要）。首先介绍利用第六个参数的方法。

使用事件对象

之前已经介绍了WSASend、WSARecv函数的第六个参数——WSAOVERLAPPED结构体，因此直接给出示例。希望各位通过该示例验证如下2点。

- 完成I/O时，WSAOVERLAPPED结构体变量引用的事件对象将变为signaled状态。
- 为了验证I/O的完成和完成结果，需要调用WSAGetOverlappedResult函数。

需要说明的是，该示例的目的在于整理之前的一系列知识点。因此，推荐各位在此基础上自行编写可以体现重叠I/O优点的示例。

❖ OverlappedSend_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.  void ErrorHandling(char *msg);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      WSADATA wsaData;
9.      SOCKET hSocket;
10.     SOCKADDR_IN sendAdr;
11.
12.     WSABUF dataBuf;
13.     char msg[]="Network is Computer!";
14.     int sendBytes=0;
15.
16.     WSAEVENT evObj;
17.     WSAOVERLAPPED overlapped;
18.
19.     if(argc!=3) {
20.         printf("Usage: %s <IP> <port>\n", argv[0]);
21.         exit(1);
22.     }
23.     if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
24.         ErrorHandling("WSAStartup() error!");
25.
26.     hSocket=WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
27.     memset(&sendAdr, 0, sizeof(sendAdr));
28.     sendAdr.sin_family=AF_INET;
29.     sendAdr.sin_addr.s_addr=inet_addr(argv[1]);
30.     sendAdr.sin_port=htons(atoi(argv[2]));
31.
32.     if(connect(hSocket, (SOCKADDR*)&sendAdr, sizeof(sendAdr))==SOCKET_ERROR)
33.         ErrorHandling("connect() error!");
34.
35.     evObj=WSACreateEvent();
36.     memset(&overlapped, 0, sizeof(overlapped));
37.     overlapped.hEvent=evObj;
38.     dataBuf.len=strlen(msg)+1;
39.     dataBuf.buf=msg;
```

```
40.
41.     if(WSASend(hSocket, &dataBuf, 1, &sendBytes, 0, &overlapped, NULL)
42.         ==SOCKET_ERROR)
43.     {
44.         if(WSAGetLastError()==WSA_IO_PENDING)
45.         {
46.             puts("Background data send");
47.             WSAWaitForMultipleEvents(1, &evObj, TRUE, WSA_INFINITE, FALSE);
48.             WSAGetOverlappedResult(hSocket, &overlapped, &sendBytes, FALSE, NULL);
49.         }
50.         else
51.         {
52.             ErrorHandling("WSASend() error");
53.         }
54.     }
55.
56.     printf("Send data size: %d \n", sendBytes);
57.     WSACloseEvent(evObj);
58.     closesocket(hSocket);
59.     WSACleanup();
60.     return 0;
61. }
62.
63. void ErrorHandling(char *msg)
64. {
65.     fputs(msg, stderr);
66.     fputc('\n', stderr);
67.     exit(1);
68. }
```

代码说明

- 第35~39行：创建事件对象并初始化待传输数据的缓冲。
- 第41行：该语句中调用的WSASend函数若不返回SOCKET_ERROR，则说明数据传输完成，所以sendBytes中的值有意义。
- 第44行：第41行调用的WSASend函数返回SOCKET_ERROR，第44行的WSAGetLastError函数返回WSA_IO_PENDING时，意味着数据传输尚未完成，仍处于传输状态。此时，sendBytes中的数据没有意义。
- 第47、48行：完成数据传输时，通过第37行注册的事件对象进入signaled状态，故可通过第47行的函数调用等待数据传输完成。完成数据传输后，可以通过第48行的函数调用获取传输结果。

上述示例的第44行调用的WSAGetLastError函数的定义如下。调用套接字相关函数后，可以通过该函数获取错误信息。

```
#include <winsock2.h>

int WSAGetLastError(void);
```

➔ 返回错误代码（表示错误原因）。

上述示例中该函数的返回值为WSA_IO_PENDING，由此可以判断WSASend函数的调用结果并非发生了错误，而是尚未完成（Pending）的状态。下面介绍与上述示例配套使用的Receiver，该示例的结构与之前的Sender类似。

❖ OverlappedRecv_win.c

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>

#define BUF_SIZE 1024
void ErrorHandling(char *message);

int main(int argc, char* argv[])
{
    WSADATA wsaData;
    SOCKET hLisnSock, hRecvSock;
    SOCKADDR_IN lisnAdr, recvAdr;
    int recvAdrSz;

    WSABUF dataBuf;
    WSAEVENT evObj;
    WSAOVERLAPPED overlapped;

    char buf[BUF_SIZE];
    int recvBytes=0, flags=0;
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsaData)!=0)
        ErrorHandling("WSAStartup() error!");

    hLisnSock=WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
    memset(&lisnAdr, 0, sizeof(lisnAdr));
    lisnAdr.sin_family=AF_INET;
    lisnAdr.sin_addr.s_addr=htonl(INADDR_ANY);
    lisnAdr.sin_port=htons(atoi(argv[1]));

    if(bind(hLisnSock, (SOCKADDR*) &lisnAdr, sizeof(lisnAdr))==SOCKET_ERROR)
        ErrorHandling("bind() error");
    if(listen(hLisnSock, 5)==SOCKET_ERROR)
        ErrorHandling("listen() error");

    recvAdrSz=sizeof(recvAdr);
    hRecvSock=accept(hLisnSock, (SOCKADDR*)&recvAdr,&recvAdrSz);

    evObj=WSACreateEvent();
    memset(&overlapped, 0, sizeof(overlapped));
    overlapped.hEvent=evObj;
    dataBuf.len=BUF_SIZE;
    dataBuf.buf=buf;
```

```
47.
48.     if(WSARecv(hRecvSock, &dataBuf, 1, &recvBytes, &flags, &overlapped, NULL)
49.         ==SOCKET_ERROR)
50.     {
51.         if(WSAGetLastError()==WSA_IO_PENDING)
52.         {
53.             puts("Background data receive");
54.             WSAWaitForMultipleEvents(1, &evObj, TRUE, WSA_INFINITE, FALSE);
55.             WSAGetOverlappedResult(hRecvSock, &overlapped, &recvBytes, FALSE, NULL);
56.         }
57.         else
58.         {
59.             ErrorHandling("WSARecv() error");
60.         }
61.     }
62.
63.     printf("Received message: %s \n", buf);
64.     WSACloseEvent(evObj);
65.     closesocket(hRecvSock);
66.     closesocket(hLisnSock);
67.     WSACleanup();
68.     return 0;
69. }
70.
71. void ErrorHandling(char *message)
72. {
73.     //与示例Overlappedsend_win.c的ErrorHandling函数一致。
74. }
```

代码说明

- 第48行：若WSARecv函数没有返回SOCKET_ERROR，则说明已完成数据接收。此时，recvBytes中保存的值是有意义的。
- 第51行：WSARecv函数返回SOCKET_ERROR、WSAGetLastError函数返回WSA_IO_PENDING时，说明正在接收数据。
- 第54、55行：通过第54行的函数调用验证是否完成数据接收，通过第55行的函数调用获取接收数据的结果。

各位若分析过OverlappedSend_win.c示例，应该很容易理解上述Receiver代码。下面给出运行结果。如果在1台计算机上同时运行Sender和Receiver，即使交换足够多的数据量也很难观察到I/O的Pending（未完成I/O的）情况。因此，大家更应该理解上述示例演示的内容，而非运行结果。

❖ 运行结果：OverlappedSend_win.c

```
Send data size: 21
```

❖ 运行结果：OverlappedRecv_win.c

```
Received messsage: Network is Computer!
```

使用 Completion Routine 函数

前面的示例通过事件对象验证了I/O完成与否，下面介绍如何通过WSASend、WSARecv函数的最后一个参数中指定的Completion Routine（以下简称CR）函数验证I/O完成情况。“注册CR”具有如下含义：

“Pending的I/O完成时调用此函数！”

I/O完成时调用注册过的函数进行事后处理，这就是Completion Routine的运作方式。如果执行重要任务时突然调用Completion Routine，则有可能破坏程序的正常执行流。因此，操作系统通常会预先定义规则：

“只有请求I/O的线程处于alertable wait状态时才能调用Completion Routine函数！”

“alertable wait状态”是等待接收操作系统消息的线程状态。调用下列函数时进入alertable wait状态。

- WaitForSingleObjectEx
- WaitForMultipleObjectsEx
- WSAWaitForMultipleEvents
- SleepEx

第一、第二、第四个函数提供的功能与WaitForSingleObject、WaitForMultipleObjects、Sleep函数相同。上述函数只增加了1个参数，如果该参数为TRUE，则相应线程将进入alertable wait状态。另外，第21章介绍过以WSA为前缀的函数，该函数的最后一个参数设置为TRUE时，线程同样进入alertable wait状态。因此，启动I/O任务后，执行完紧急任务时可以调用上述任一函数验证I/O完成与否。此时操作系统知道线程进入alertable wait状态，如果有已完成的I/O，则调用相应Completion Routine函数。调用后，上述函数将全部返回WAIT_IO_COMPLETION，并开始执行接下来的程序。

以上就是Completion Routine函数相关的全部理论说明。下面将之前的OverlappedRecv_win.c改为Completion Routine方式。

CmplRoutinesRecv_win.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <winsock2.h>
4.
5.  #define BUF_SIZE 1024
6.  void CALLBACK CompRoutine(DWORD, DWORD, LPWSAOVERLAPPED, DWORD);
7.  void ErrorHandling(char *message);
8.
9.  WSABUF dataBuf;
```

```
10. char buf[BUF_SIZE];
11. int recvBytes=0;
12.
13. int main(int argc, char* argv[])
14. {
15.     WSADATA wsaData;
16.     SOCKET hLisnSock, hRecvSock;
17.     SOCKADDR_IN lisnAdr, recvAdr;
18.
19.     WSAOVERLAPPED overlapped;
20.     WSAEVENT evObj;
21.
22.     int idx, recvAdrSz, flags=0;
23.     if(argc!=2) {
24.         printf("Usage: %s <port>\n", argv[0]);
25.         exit(1);
26.     }
27.     if(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
28.         ErrorHandling("WSAStartup() error!");
29.
30.     hLisnSock=WSASocket(PF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
31.     memset(&lisnAdr, 0, sizeof(lisnAdr));
32.     lisnAdr.sin_family=AF_INET;
33.     lisnAdr.sin_addr.s_addr=htonl(INADDR_ANY);
34.     lisnAdr.sin_port=htons(atoi(argv[1]));
35.
36.     if(bind(hLisnSock, (SOCKADDR*) &lisnAdr, sizeof(lisnAdr))==SOCKET_ERROR)
37.         ErrorHandling("bind() error");
38.     if(listen(hLisnSock, 5)==SOCKET_ERROR)
39.         ErrorHandling("listen() error");
40.
41.     recvAdrSz=sizeof(recvAdr);
42.     hRecvSock=accept(hLisnSock, (SOCKADDR*)&recvAdr,&recvAdrSz);
43.     if(hRecvSock==INVALID_SOCKET)
44.         ErrorHandling("accept() error");
45.
46.     memset(&overlapped, 0, sizeof(overlapped));
47.     dataBuf.len=BUF_SIZE;
48.     dataBuf.buf=buf;
49.     evObj=WSACreateEvent();          // Dummy event object
50.
51.     if(WSARecv(hRecvSock, &dataBuf, 1, &recvBytes, &flags, &overlapped, CompRoutine)
52.         ==SOCKET_ERROR)
53.     {
54.         if(WSAGetLastError()==WSA_IO_PENDING)
55.             puts("Background data receive");
56.     }
57.
58.     idx=WSAWaitForMultipleEvents(1, &evObj, FALSE, WSA_INFINITE, TRUE);
59.     if(idx==WAIT_IO_COMPLETION)
60.         puts("Overlapped I/O Completed");
61.     else    // If error occurred!
62.         ErrorHandling("WSARecv() error");
63.
```