China-pub.com
下载

第一部分　程序员必读

第1章　对程序错误的处理

在开始介绍Microsoft Windows 的特性之前，必须首先了解Windows的各个函数是如何进行错误处理的。

当调用一个Windows函数时，它首先要检验传递给它的的各个参数的有效性，然后再设法执行任务。如果传递了一个无效参数，或者由于某种原因无法执行这项操作，那么操作系统就会返回一个值，指明该函数在某种程度上运行失败了。表 1-1列出了大多数Windows函数使用的返回值的数据类型。

表1-1　Windows函数常用的返回值类型

数 据 类 型	表示失败的值
VOID	该函数的运行不可能失败。Windows函数的返回值类型很少是VOID
BOOL	如果函数运行失败，那么返回值是0，否则返回的是非0值。最好对返回值进行测试，以确定它是0还是非0。不要测试返回值是否为TRUE
HANDLE	如果函数运行失败，则返回值通常是NULL，否则返回值为HANDLE，用于标识你可以操作的一个对象。注意，有些函数会返回一个句柄值INVALID_ HANDLE_VALUE，它被定义为-1。函数的Platform SDK文档将会清楚地说明该函数运行失败时返回的是NULL还是INVALID_HANDLE_VALID
PVOID	如果函数运行失败，则返回值是NULL，否则返回PVOID，以标识数据块的内存地址
LONG/DWORD	这是个难以处理的值。返回数量的函数通常返回 LONG或DWORD。如果由于某种原因，函数无法对想要进行计数的对象进行计数，那么该函数通常返回 0或-1（根据函数而定）。如果调用的函数返回了 LONG/DWORD，那么请认真阅读Platform SDK文档，以确保能正确检查潜在的错误

一个Windows函数返回的错误代码对了解该函数为什么会运行失败常常很有用。 Microsoft公司编译了一个所有可能的错误代码的列表，并且为每个错误代码分配了一个 32位的号码。

从系统内部来讲，当一个Windows函数检测到一个错误时，它会使用一个称为线程本地存储器(thread-local storage)的机制，将相应的错误代码号码与调用的线程关联起来(线程本地存储器将在第21章中介绍)。这将使线程能够互相独立地运行，而不会影响各自的错误代码。当函数返回时，它的返回值就能指明一个错误已经发生。若要确定这是个什么错误，请调用GetLastError函数：

```
DWORD GetLastError();
```

该函数只返回线程的32位错误代码。

当你拥有32位错误代码的号码时，必须将该号码转换成更有用的某种对象。 WinError.h头文件包含了Microsoft公司定义的错误代码的列表。下面显示了该列表的某些内容，使你能够看到它的大概样子：

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
```

China-pub.com 下载

```
//
//  The operation completed successfully.
//
#define ERROR_SUCCESS                    0L

#define NO_ERROR 0L                             // dderror
#define SEC_E_OK                        ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
//  Incorrect function.
//
#define ERROR_INVALID_FUNCTION           1L     // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND             2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND             3L

//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
//  The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES        4L

//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
//  Access is denied.
//
#define ERROR_ACCESS_DENIED              5L
```

如你所见，每个错误都有3种表示法：一个消息ID（这是你可以在源代码中使用的一个宏，以便与GetLastError的返回值进行比较），消息文本（对错误的英文描述）和一个号码（应该避

China-pub.com
下载

免使用这个号码，可使用消息ID）。请记住，这里只显示了WinError.h头文件中的很少一部分内容，整个文件的长度超过21000行。

当Windows函数运行失败时，应该立即调用GetLastError函数。如果调用另一个Windows函数，它的值很可能被改写。

注意 GetLastError能返回线程产生的最后一个错误。如果该线程调用的Windows函数运行成功，那么最后一个错误代码就不被改写，并且不指明运行成功。有少数Windows函数并不遵循这一规则，它会更改最后的错误代码；但是Platform SDK文档通常指明，当函数运行成功时，该函数会更改最后的错误代码。

Windows 98 许多Windows 98的函数实际上是用Microsoft公司的16位Windows 3.1产品产生的16位代码来实现的。这种比较老的代码并不通过GetLastError之类的函数来报告错误，而且Microsoft公司并没有在Windows 98中修改16位代码，以支持这种错误处理方式。对于我们来说，这意味着Windows 98中的许多Win32函数在运行失败时不能设置最后的错误代码。该函数将返回一个值，指明运行失败，这样你就能够发现该函数确实已经运行失败，但是你无法确定运行失败的原因。

有些Windows函数之所以能够成功运行，其中有许多原因。例如，创建指明的事件内核对象之所以能够取得成功，是因为你实际上创建了该对象，或者因为已经存在带有相同名字的事件内核对象。你应搞清楚成功的原因。为了将该信息返回，Microsoft公司选择使用最后错误代码机制。这样，当某些函数运行成功时，就能够通过调用GetLadtError函数来确定其他的一些信息。对于具有这种行为特性的函数来说，Platform SDK文档清楚地说明了GetLastError函数可以这样使用。请参见该文档，找出CreateEvent函数的例子。

进行调试的时候，监控线程的最后错误代码是非常有用的。在Microsoft Visual studio 6.0中，Microsoft的调试程序支持一个非常有用的特性，即可以配置Watch窗口，以便始终都能显示线程的最后错误代码的号码和该错误的英文描述。通过选定 Watch窗口中的一行，并键入“@err,hr”，就能够做到这一点。观察图1-1，你会看到已经调用了CreateFile函数。该函数返回INVALID_HANDLE_VALUE（-1）的HANDLE，表示它未能打开指定的文件。但是Watch窗口向我们显示最后错误代码（即如果调用 GetLastError函数，该函数返回的错误代码）是0x00000002。该Watch窗口又进一步指明错误代码2是指“系统不能找到指定的文件。”你会发现它与WinError.h头文件中的错误代码2所指的字符串是相同的。

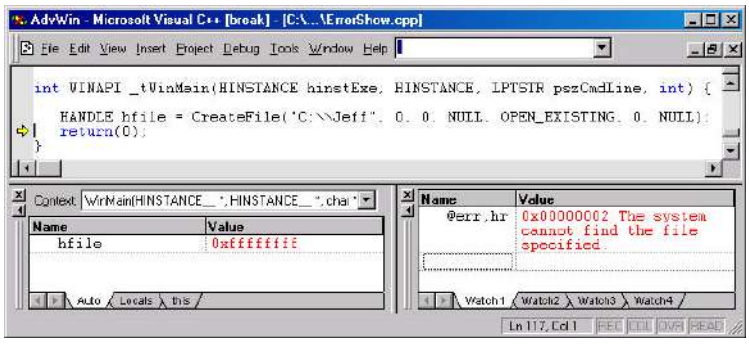


图1-1 在Visual Studio 6.0的Watch窗口中键入“@err,hr”，就可以查看当前线程的最后错误代码

China-pub.com 下载

Visual studio还配有一个小的实用程序，称为Error Lookup。可以使用Error Lookup将错误代码的号码转换成相应文本描述（见图1-2）。

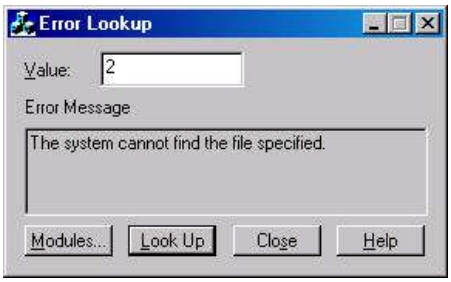


图1-2 Error Lookup窗口

如果在编写的应用程序中发现一个错误，可能想要向用户显示该错误的文本描述。Windows提供了一个函数，可以将错误代码转换成它的文本描述。该函数称为FormatMessage，显示如下：

```
DWORD FormatMessage(
   DWORD dwFlags,
   LPCVOID pSource,
   DWORD dwMessageId,
   DWORD dwLanguageId,
   PTSTR pszBuffer,
   DWORD nSize,
   va_list *Arguments);
```

FormatMessage函数的功能实际上是非常丰富的，在创建向用户显示的字符串信息时，它是首选函数。该函数之所以有这样大的作用，原因之一是它很容易用多种语言进行操作。该函数能够检测出用户首选的语言（在Regional Settings Control Panel小应用程序中设定），并返回相应的文本。当然，首先必须自己转换字符串，然后将已转换的消息表资源嵌入你的.exe文件或DLL模块中，然后该函数会选定正确的嵌入对象。ErrorShow示例应用程序（本章后面将加以介绍）展示了如何调用该函数，以便将Microsoft公司定义的错误代码转换成它的文本描述。

有些人常常问我，Microsoft公司是否建立了一个主控列表，以显示每个Windows函数可能返回的所有错误代码。可惜，回答是没有这样的列表，而且Microsoft公司将永远不会建立这样的一个列表。因为在创建系统的新版本时，建立和维护该列表实在太困难了。

建立这样一个列表存在的问题是，你可以调用一个Windows函数，但是该函数能够在内部调用另一个函数，而这另一个函数又可以调用另一个函数，如此类推。由于各种不同的原因，这些函数中的任何一个函数都可能运行失败。有时，当一个函数运行失败时，较高级的函数对它进行恢复，并且仍然可以执行你想执行的操作。为了创建该主控列表，Microsoft公司必须跟踪每个函数的运行路径，并建立所有可能的错误代码的列表。这项工作很困难。而且，当创建系统的新版本时，这些函数的运行路径还会改变。

1.1 定义自己的错误代码

前面已经说明Windows函数是如何向函数的调用者指明发生的错误，你也能够将该机制用于自己的函数。比如说，你编写了一个希望其他人调用的函数，你的函数可能因为这样或那样的原因而运行失败，你必须向函数的调用者说明它已经运行失败。

若要指明函数运行失败，只需要设定线程的最后的错误代码，然后让你的函数返回FALSE、INVALID_HANDLE_VALUE、NULL或者返回任何合适的信息。若要设定线程的最后错误代码，只需调用下面的代码：

请将你认为合适的任何32位号码传递给该函数。尝试使用WinError.h中已经存在的代码，

```
VOID SetLastError(DWORD dwErrCode);
```

China-pub.com
下载

只要该代码能够正确地指明想要报告的错误即可。如果你认为 WinError.h中的任何代码都不能正确地反映该错误的性质，那么可以创建你自己的代码。错误代码是个 32位的数字，划分成表 1-2 所示的各个域。

表1-2 错误代码的域

位	31~30	29	28	27~16	15~0
内容 含义	严重性 0=成功 1=供参考 2=警告 3=错误	Microsoft/客户 0=Microsoft公司定义的代码 1=客户定义的代码	保留 必须是0	设备代码 由Microsoft 公司定义	异常代码 由Microsoft/ 客户定义

这些域将在第24章中详细讲述。现在，需要知道的重要域是第 29位。Microsoft公司规定，他们建立的所有错误代码的这个信息位均使用 0。如果创建自己的错误代码，必须使 29位为1。这样，就可以确保你的错误代码与Microsoft公司目前或者将来定义的错误代码不会发生冲突。

1.2 ErrorShow示例应用程序

ErrorShow应用程序“ 01 ErrorShow.exe ”(在清单1-1中列出)展示了如何获取错误代码的文本描述的方法。该应用程序的源代码和资源文件位于本书所附光盘上的 01-ErrorShow目录下。一般来说，该应用程序用于显示调试程序的 Watch窗口和 Error Lookup程序是如何运行的。当启动该程序时，就会出现如图1-3所示的窗口。



图1-3 Error Show窗口

可以将任何错误代码键入该编辑控件。当单击 Look up按钮时，在底部的滚动窗口中就会显示该错误的文本描述。该应用程序唯一令人感兴趣的特性是如何调用 FormatMessage函数。下面是使用该函数的方法：

```
// Get the error code
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL;   // Buffer that gets the error message string

// Get the error code's textual description
BOOL fOk = FormatMessage(
   FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
   NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
   (LPTSTR) &hlocal, 0, NULL);
.
.
.
if (hlocal != NULL) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
}
```

第一个代码行用于从编辑控件中检索错误代码的号码。然后，建立一个内存块的句柄并将它初始化为NULL。FormatMessage函数在内部对内存块进行分配，并将它的句柄返回给我们。

China-pub.com 下载

当调用FormatMessage函数时，传递了FORMAT_MESSAGE_FROM_SYSTEM标志。该标志告诉FormatMessage函数，我们想要系统定义的错误代码的字符串。还传递了 FORMAT_MESSAGE_ALLOCATE_BUFFER标志，告诉该函数为错误代码的文本描述分配足够大的内存块。该内存块的句柄将在hlocal变量中返回。第三个参数指明想要查找的错误代码的号码，第四个参数指明想要文本描述使用什么语言。

如果FormatMessage函数运行成功，那么错误代码的文本描述就位于内存块中，将它拷贝到对话框底部的滚动窗口中。如果FormatMessage函数运行失败，设法查看NetMsg.dll模块中的消息代码，以了解该错误是否与网络有关。使用 NetMsg.dll模块的句柄，再次调用FormatMessage函数。你会看到，每个DLL（或.exe）都有它自己的一组错误代码，可以使用Message Compiler（MC.exe）将这组错误代码添加给该模块，并将一个资源添加给该模块。这就是Visual Studio的Error Lookup工具允许你用Modules对话框进行的操作。以下是清单 1-1 ErrorShow示例应用程序。

清单1-1 ErrorShow示例应用程序

ErrorShow.cpp

```
/******************************************************************************
Module:  ErrorShow.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


#define ESM_POKECODEANDLOOKUP    (WM_USER + 100)
const TCHAR g_szAppName[] = TEXT("Error Show");


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_ERRORSHOW);

   // Don't accept error codes more than 5 digits long
   Edit_LimitText(GetDlgItem(hwnd, IDC_ERRORCODE), 5);

   // Look up the command-line passed error number
   SendMessage(hwnd, ESM_POKECODEANDLOOKUP, lParam, 0);
   return(TRUE);
}
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
switch (id) {

case IDCANCEL:
   EndDialog(hwnd, id);
   break;

case IDC_ALWAYSONTOP:
   SetWindowPos(hwnd, IsDlgButtonChecked(hwnd, IDC_ALWAYSONTOP)
      ? HWND_TOPMOST : HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
   break;

case IDC_ERRORCODE:
   EnableWindow(GetDlgItem(hwnd, IDOK), Edit_GetTextLength(hwndCtl) > 0);
   break;

case IDOK:
   // Get the error code
   DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

   HLOCAL hlocal = NULL;   // Buffer that gets the error message string

   // Get the error code's textual description
   BOOL fOk = FormatMessage(
      FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
      NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
      (PTSTR) &hlocal, 0, NULL);

   if (!fOk) {
      // Is it a network-related error?
      HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
         DONT_RESOLVE_DLL_REFERENCES);

      if (hDll != NULL) {
         FormatMessage(
            FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_FROM_SYSTEM,
            hDll, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
            (PTSTR) &hlocal, 0, NULL);
         FreeLibrary(hDll);
      }
   }

   if (hlocal != NULL) {
      SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
      LocalFree(hlocal);
         } else {
            SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
         }
         break;
      }
   }

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);

   case ESM_POKECODEANDLOOKUP:
      SetDlgItemInt(hwnd, IDC_ERRORCODE, (UINT) wParam, FALSE);
      FORWARD_WM_COMMAND(hwnd, IDOK, GetDlgItem(hwnd, IDOK), BN_CLICKED,
         PostMessage);
      SetForegroundWindow(hwnd);
      break;
   }

   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   HWND hwnd = FindWindow(TEXT("#32770"), TEXT("Error Show"));
   if (IsWindow(hwnd)) {
      // An instance is already running, activate it and send it the new #
      SendMessage(hwnd, ESM_POKECODEANDLOOKUP, _ttoi(pszCmdLine), 0);
   } else {
      DialogBoxParam(hinstExe, MAKEINTRESOURCE(IDD_ERRORSHOW),
         NULL, Dlg_Proc, _ttoi(pszCmdLine));
   }
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ERRORSHOW DIALOGEX 0, 0, 182, 42
STYLE DS_SETFOREGROUND | DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
CAPTION "Error Show"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Error:",IDC_STATIC,4,4,19,8
    EDITTEXT        IDC_ERRORCODE,24,2,24,14,ES_AUTOHSCROLL | ES_NUMBER
    DEFPUSHBUTTON   "Look up",IDOK,56,2,36,14
    CONTROL         "&On top",IDC_ALWAYSONTOP,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,104,4,38,10
    EDITTEXT        IDC_ERRORTEXT,4,20,176,20,ES_MULTILINE | ES_AUTOVSCROLL |
                    ES_READONLY | NOT WS_BORDER | WS_VSCROLL,
                    WS_EX_CLIENTEDGE
END


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_ERRORSHOW, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 175
        TOPMARGIN, 7
        BOTTOMMARGIN, 35
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
```

China-pub.com 下载

```
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_ERRORSHOW           ICON    DISCARDABLE     "ErrorShow.ico"
#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第2章 Unicode

随着Microsoft公司的Windows操作系统在全世界日益广泛的流行，对于软件开发人员来说，将目标瞄准国际上的各个不同市场，已经成为一个越来越重要的问题。美国的软件版本比国际版本提前6个月推向市场，这曾经是个司空见惯的现象。但是，由于各国对 Windows操作系统提供了越来越多的支持，因此就更加容易为国际市场生产各种应用软件，从而缩短了软件的美国版本与国际版本推出的时间间隔。

Windows操作系统始终不遗地提供各种支持，以帮助软件开发人员进行应用程序的本地化工作。应用软件可以从各种不同的函数中获得特定国家的信息，并可观察控制面板的设置，以确定用户的首选项。Windows甚至支持不同的字体，以适应应用的需要。

之所以将这一章放在本书的开头，是因为考虑到 Unicode是开发任何应用程序时要采用的基本步骤。本书的每一章中几乎都要讲到关于 Unicode的问题，而且书中给出的所有示例应用程序都是“用Unicode实现的”。如果你为Microsoft Windows 2000或Microsoft Windows CE开发应用程序，你应该使用Unicode进行开发。如果你为Microsoft Windows 98开发应用程序，你必须对某些问题作出决定。本章也要讲述Windows 98的有关问题。

2.1 字符集

软件的本地化要解决的真正问题，实际上就是如何来处理不同的字符集。多年来，许多人一直将文本串作为一系列单字节字符来进行编码，并在结尾处放上一个零。对于我们来说，这已经成了习惯。当调用strlen函数时，它在以0结尾的单字节字符数组中返回字符的数目。

问题是，有些文字和书写规则（比如日文中的汉字就是个典型的例子）的字符集中的符号太多了，因此单字节（它提供的符号最多不能超过 256个）是根本不敷使用的。为此出现了双字节字符集（DBCS），以支持这些文字和书写规则。

2.1.1 单字节与双字节字符集

在双字节字符集中，字符串中的每个字符可以包含一个字节或包含两个字节。例如，日文中的汉字，如果第一个字符在 0x81与0x9F之间，或者在0xE0与0xFC之间，那么就必须观察下一个字节，才能确定字符串中的这个完整的字符。使用双字节字符集，对于程序员来说简直是个很大的难题，因为有些字符只有一个字节宽，而有些字符则是两个字节宽。

如果只是调用 strlen函数，那么你无法真正了解字符串中究竟有多少字符，它只能告诉你到达结尾的0之前有多少个字节。 ANSI的C运行期库中没有配备相应的函数，使你能够对双字节字符集进行操作。但是， Microsoft Visual C++的运行期库却包含许多函数，如 _mbslen,它可以用来操作多字节（既包括单字节也包括双字节）字符串。

为了帮助你对DBCS字符串进行操作，Windows提供了下面的一组帮助函数(见表2-1)。

前两个函数CharNext 和Char Prev 允许前向或逆向遍历DBCS 字符串，方法是每次一个字符。第三个函数 IsDBCSLeadByte, 在字节返回到一个两字字节符的第一个字节时将返回TRUE。

China-pub.com 下载

表2-1 对DBCS字符串进行操作的帮助函数

函 数	描 述
PTSTR CharNext（PCTSTR pszCurrentChar）；	返回字符串中的下一个字符的地址
PTSTR CharPrev（PCTSTR pszStart,PCTSTR pszCurrentChar)；	返回字符串中的上一个字符的地址
BOOL IsDBCSLeadByteTRUE(BYTE bTestChar)；	如果该字节是DBCS字符的第一个字节，则返回

尽管这些函数使得我们对 DBCS的操作更容易，但还需要，一个更好的方法让我们来看看Unicode。

2.1.2 Unicode：宽字节字符集

Unicode是Apple和Xerox公司于1988年建立的一个技术标准。1991年，成立了一个集团机构负责Unicode的开发和推广应用。该集团由Apple、Compaq、HP、IBM、Microsoft、Oracle、Silicon Graphics, Inc.、Sybase、Unisys和Xerox等公司组成（若要了解该集团的全部成员，请通过网址www.Unicode.org查找）。该集团负责维护Unicode标准。Unicode的完整描述可以参阅AddisonWesley出版的《Unicode Standard》一书（该书可以通过网址www.Unicode.org订购）。

Unicode提供了一种简单而又一致的表示字符串的方法。Unicode字符串中的所有字符都是16位的（两个字节）。它没有专门的字节来指明下一个字节是属于同一个字符的组成部分，还是一个新字符。这意味着你只需要对指针进行递增或递减，就可以遍历字符串中的各个字符，不再需要调用CharNext、CharPrev和IsDBCSLeadByte之类的函数。

由于Unicode用一个16位的值来表示每个字符，因此总共可以得到 65 000个字符，这样，它就能够对世界各国的书面文字中的所有字符进行编码，远远超过了单字节字符集的 256个字符的数目。

目前，已经为阿拉伯文、中文拼音、西里尔字母（俄文）、希腊文、西伯莱文、日文、韩文和拉丁文（英文）字母定义了 Unicode代码点㊀。这些字符集中还包含了大量的标点符号、数学符号、技术符号、箭头、装饰标志、区分标志和其他许多字符。如果将所有这些字母和符号加在一起，总计约达35000个不同的代码点，这样，总计65 000多个代码点中，大约还有一半可供将来扩充时使用。

这65 536个字符可以分成不同的区域。表 2-2 显示了这样的区域的一部分以及分配给这些区域的字符。

表2-2 区域字符

16位代码	字 符	16 位 代 码	字 符
0000-007F	ASCII	0300-036F	通用区分标志
0080-00FF	拉丁文1字符	0400-04FF	西里尔字母
0100-017F	欧洲拉丁文	0530-058F	亚美尼亚文
0180-01FF	扩充拉丁文	0590-05FF	西伯莱文
0250-02AF	标准拼音	0600-06FF	阿拉伯文
02B0-02FF	修改型字母	0900-097F	梵文

目前尚未分配的代码点大约还有29 000个，不过它们是保留供将来使用的。另外，大约有6000个代码点是保留供个人使用的。

㊀ 代码点是字符集中符号的位置。

China-pub.com
下载

2.2 为什么使用Unicode

当开发应用程序时，当然应该考虑利用Unicode的优点。即使现在你不打算对应用程序进行本地化，开发时将Unicode放在心上，肯定可以简化将来的代码转换工作。此外，Unicode还具备下列功能：

- 可以很容易地在不同语言之间进行数据交换。
- 使你能够分配支持所有语言的单个二进制.exe文件或DLL文件。
- 提高应用程序的运行效率（本章后面还要详细介绍）。

2.3 Windows 2000与Unicode

Windows 2000是使用Unicode从头进行开发的，用于创建窗口、显示文本、进行字符串操作等的所有核心函数都需要Unicode字符串。如果调用任何一个Windows函数并给它传递一个ANSI字符串，那么系统首先要将字符串转换成Unicode，然后将Unicode字符串传递给操作系统。如果希望函数返回ANSI字符串，系统就会首先将Unicode字符串转换成ANSI字符串，然后将结果返回给你的应用程序。所有这些转换操作都是在你看不见的情况下发生的。当然，进行这些字符串的转换需要占用系统的时间和内存。

例如，如果调用CreateWindowEx函数，并传递类名字和窗口标题文本的非Unicode字符串，那么CreateWindowEx必须分配内存块（在你的进程的默认堆中），将非Unicode字符串转换成Unicode字符串，并将结果存储在分配到的内存块中，然后调用Unicode版本的CreateWindowEx函数。

对于用字符串填入缓存的函数来说，系统必须首先将Unicode字符串转换成非Unicode字符串，然后你的应用程序才能处理该字符串。由于系统必须执行所有这些转换操作，因此你的应用程序需要更多的内存，并且运行的速度比较慢。通过从头开始用Unicode来开发应用程序，就能够使你的应用程序更加有效地运行。

2.4 Windows 98与Unicode

Windows 98不是一种全新的操作系统。它继承了16位Windows操作系统的特性，它不是用来处理Unicode的。如果要增加对Unicode的支持，其工作量非常大，因此在该产品的特性列表中没有包括这个支持项目。由于这个原因，Windows 98像它的前任产品一样，几乎都是使用ANSI字符串来进行所有的内部操作的。

仍然可以编写用于处理Unicode字符和字符串的Windows应用程序，不过，使用Windows函数要难得多。例如，如果想要调用CreateWindowEx函数并将ANSI字符串传递给它，这个调用的速度非常快，不需要从你进程的默认堆栈中分配缓存，也不需要进行字符串转换。但是，如果想要调用CreateWindowEx函数并将Unicode字符串传递给它，就必须明确分配缓存，并调用函数，以便执行从Unicode到ANSI字符串的转换操作。然后可以调用CreateWindowEx，传递ANSI字符串。当CreateWindowEx函数返回时，就能释放临时缓存。这比使用Windows 2000上的Unicode要麻烦得多。本章的后面要介绍如何在Windows 98下进行这些转换。

虽然大多数Unicode函数在Windows 98中不起任何作用，但是仍有少数Unicode函数确实非常有用。这些函数是：

- EnumResourceLanguagesW
- EnumResourceNamesW
- GetTextExtentPoint32W
- GetTextExtentPointW

China-pub.com 下载

- EnumResourceTypesW
- ExtTextOutW
- FindResourceW
- FindResourceExW
- GetCharWidthW
- GetCommandLineW
- LstrlenW
- MessageBoxExW
- MessageBoxW
- TextOutW
- WideCharToMultiByte
- MultiByteToWideChar

可惜的是，这些函数中有许多函数在Windows 98中会出现各种各样的错误。有些函数无法使用某些字体，有些函数会破坏内存堆栈，有些函数会使打印机驱动程序崩溃，等等。如果要使用这些函数，必须对它们进行大量的测试。即使这样，可能仍然无法解决问题。因此必须向用户说明这些情况。

2.5 Windows CE与Unicode

Windows CE操作系统是为小型设备开发的，这些设备的内存很小，并且不带磁盘存储器。你可能认为，由于Microsoft公司的主要目标是建立一种尽可能小的操作系统，因此它会使用ANSI作为自己的字符集。但是Microsoft公司并非鼠目寸光，他们懂得，Windows CE的设备要在世界各地销售，他们希望降低软件开发成本，这样就能更加容易地开发应用程序。为此，Windows CE本身就是使用Unicode的一种操作系统。

但是，为了使Windows CE尽量做得小一些，Microsoft公司决定完全不支持ANSI Windows函数。因此，如果要为Windows CE开发应用程序，必须懂得Unicode，并且在整个应用程序中使用Unicode。

2.6 需要注意的问题

下面让我们进一步明确一下“Microsoft公司对Unicode支持的情况”：

- Windows 2000既支持Unicode，也支持ANSI，因此可以为任意一种开发应用程序。
- Windows 98只支持ANSI，只能为ANSI开发应用程序。
- Windows CE只支持Unicode，只能为Unicode开发应用程序。

虽然Microsoft公司试图让软件开发人员能够非常容易地开发在这3种平台上运行的软件，但是Unicode与ANSI之间的差异使得事情变得困难起来，并且这种差异通常是我遇到的最大的问题之一。请不要误解，Microsoft公司坚定地支持Unicode，并且我也坚决鼓励你使用它。不过你应该懂得，你可能遇到一些问题，需要一定的时间来解决这些问题。建议你尽可能使用Unicode。如果运行Windows 98，那么只有在必要时才需转换到ANSI。

不过，还有另一个小问题你应该了解，那就是COM。

2.7 对COM的简单说明

当Microsoft公司将COM从16位Windows转换成Win32时，公司作出了一个决定，即需要字符串的所有COM接口方法都只能接受Unicode字符串。这是个了不起的决定，因为COM通常用于使不同的组件能够互相进行通信，而Unicode则是传递字符串的最佳手段。

如果你为Windows 2000或Windows CE开发应用程序，并且也使用COM，那么你将会如虎添翼。在你的整个源代码中使用Unicode，将使与操作系统进行通信和与COM对象进行通信的操作变成一件轻而易举的事情。

如果你为Windows 98开发应用程序，并且也使用COM，那么将会遇到一些问题。COM要

China-pub.com
下载

求使用Unicode字符串，而操作系统的大多数函数要求使用 ANSI字符串。那是多么难办的事情啊！我曾经从事过若干个项目的开发，在这些项目中，我编写了许多代码，仅仅是为了来回进行字符串的转换。

2.8 如何编写Unicode源代码

Microsoft公司为Unicode设计了Windows API，这样，可以尽量减少对你的代码的影响。实际上，你可以编写单个源代码文件，以便使用或者不使用 Unicode来对它进行编译。只需要定义两个宏（UNICODE和_UNICODE），就可以修改然后重新编译该源文件。

2.8.1 C运行期库对Unicode的支持

为了利用Unicode字符串，定义了一些数据类型。标准的 C头文件String.h已经作了修改，以便定义一个名字为wchar_t的数据类型，它是一个Unicode字符的数据类型：

```
typedef unsigned short wchar_t;
```

例如，如果想要创建一个缓存，用于存放最多为 99个字符的Unicode字符串和一个结尾为零的字符，可以使用下面这个语句：

```
wchar_t szBuffer[100];
```

该语句创建了一个由 100个16位值组成的数组。当然，标准的 C运行期字符串函数，如strcpy、strchr和strcat等，只能对ANSI字符串进行操作，不能正确地处理Unicode字符串。因此，ANSI C也拥有一组补充函数。清单2-1显示了一些标准的ANSI C字符串函数，后面是它们的等价Unicode函数。

```
char * strcat(char *, const char *);
wchar_t * wcscat(wchar_t *, const wchar_t *);
```

清单2-1 标准的ANSI C字符串函数和它们的等价Unicode函数

```
char * strchr(const char *, int);
wchar_t * wcschr(const wchar_t *, wchar_t);

int strcmp(const char *, const char *);
int wcscmp(const wchar_t *, const wchar_t *);

char * strcpy(char *, const char *);
wchar_t * wcscpy(wchar_t *, const wchar_t *);

size_t strlen(const char *);
size_t wcslen(const wchar_t *);
```

请注意，所有的Unicode函数均以wcs开头，wcs是宽字符串的英文缩写。若要调用Unicode函数，只需用前缀wcs来取代 ANSI字符串函数的前缀str即可。

注意 大多数软件开发人员可能已经不记得这样一个非常重要的问题了，那就是Microsoft公司提供的C运行期库与ANSI的标准C运行期库是一致的。ANSI C规定，C运行期库支持Unicode字符和字符串。这意味着始终都可以调用C运行期函数，以便对Unicode字符和字符串进行操作，即使是在Windows 98上运行，也可以调用这些函数。换句话说，wcscat、wcslen和wcstok等函数都能够在Windows 98上很好地运行，这些都是必须关心的操作系统函数。

China-pub.com 下载

对于包含了对str函数或wcs函数进行显式调用的代码来说，无法非常容易地同时为ANSI和Unicode对这些代码进行编译。本章前面说过，可以创建同时为ANSI和Unicode进行编译的单个源代码文件。若要建立双重功能，必须包含TChar.h文件，而不是包含String.h文件。

TChar.h文件的唯一作用是帮助创建ANSI/Unicode通用源代码文件。它包含你应该用在源代码中的一组宏，而不应该直接调用str函数或者wcs函数。如果在编译源代码文件时定义了_UNICODE，这些宏就会引用wcs这组函数。如果没有定义_UNICODE，那么这些宏将引用str这组宏。

例如，在TChar.h中有一个宏称为_tcscpy。如果在包含该头文件时没有定义_UNICODE,那么_tcscpy就会扩展为ANSI的strcpy函数。但是如果定义了_UNICODE, _tcscpy将扩展为Unicode的wcscpy函数。拥有字符串参数的所有C运行期函数都在TChar.h文件中定义了一个通用宏。如果使用通用宏，而不是ANSI/Unicode的特定函数名，就能够顺利地创建可以为ANSI或Unicode进行编译的源代码。

但是，除了使用这些宏之外，还有一些操作是必须进行的。TChar.h文件包含了另外一些宏。

若要定义一个ANSI/Unicode通用的字符串数组，请使用下面的TCHAR数据类型。如果定义了_UNICODE，TCHAR将声明为下面的形式：

```
typedef wchar_t TCHAR;
```

如果没有定义_UNICODE，则TCHAR将声明为下面的形式：

```
typedef char TCHAR;
```

使用该数据类型，可以像下面这样分配一个字符串：

```
TCHAR szString[100];
```

也可以创建对字符串的指针：

```
TCHAR *szError = "Error";
```

不过上面这行代码存在一个问题。按照默认设置，Microsoft公司的C++编译器能够编译所有的字符串，就像它们是ANSI字符串，而不是Unicode字符串。因此，如果没有定义_UNICODE，该编译器将能正确地编译这一行代码。但是，如果定义了_UNICODE，就会产生一个错误。若要生成一个Unicode字符串而不是ANSI字符串，必须将该代码行改写为下面的样子：

```
TCHAR *szError = L"Error";
```

字符串（literal string）前面的大写字母L，用于告诉编译器该字符串应该作为Unicode字符串来编译。当编译器将字符串置于程序的数据部分中时，它在每个字符之间分散插入零字节。这种变更带来的问题是，现在只有当定义了_UNICODE时，程序才能成功地进行编译。我们需要另一个宏，以便有选择地在字符串的前面加上大写字母L。这项工作由_TEXT宏来完成，_TEXT宏也在TChar.h文件中做了定义。如果定义了_UNICODE，那么_TEXT定义为下面的形式：

```
#define _TEXT(x) L ## x
```

如果没有定义_UNICODE，_TEXT将定义为

```
#define _TEXT(x) x
```

使用该宏，可以改写上面这行代码，这样，无论是否定义了_UNICODE宏，它都能够正确地进行编译。如下所示：

```
TCHAR *szError = _TEXT("Error");
```

China-pub.com
下载

_TEXT宏也可以用于字符串。例如，若要检查一个字符串的第一个字符是否是大写字母 J，只需编写下面的代码即可：

```
if (szError[0] == _TEXT('J')) {
   // First character is a 'J'
   .
   .
   .
} else {
   // First character is not a 'J'
   .
   .
   .
}
```

2.8.2 Windows定义的Unicode数据类型

Windows头文件定义了表2-3列出的数据类型。

表2-3 Uincode 数据类型

数据类型	说 明
WCHAR	Unicode字符
PWSTR	指向Unicode字符串的指针
PCWSTR	指向一个恒定的Unicode字符串的指针

这些数据类型是指Unicode字符和字符串。Windows头文件也定义了ANSI/Unicode通用数据类型PTSTR和PCTSTR。这些数据类型既可以指ANSI字符串，也可以指Unicode字符串，这取决于当编译程序模块时是否定义了UNICODE宏。

请注意，这里的UNICODE宏没有前置的下划线。_UNICODE宏用于C运行期头文件，而UNICODE宏则用于Windows头文件。当编译源代码模块时，通常必须同时定义这两个宏。

2.8.3 Windows中的Unicode函数和ANSI函数

前面已经讲过，有两个函数称为CreateWindowEx，一个CreateWindowEx接受Unicode字符串，另一个CreateWindowEx接受ANSI字符串。情况确实如此，不过，这两个函数的原型实际上是下面的样子：

```
HWND WINAPI CreateWindowExW(
   DWORD dwExStyle,
   PCWSTR pClassName,
   PCWSTR pWindowName,
   DWORD dwStyle,
   int X,
   int Y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU hMenu,
   HINSTANCE hInstance,
   PVOID pParam);

HWND WINAPI CreateWindowExA(
   DWORD dwExStyle,
   PCSTR pClassName,
```

China-pub.com 下载

```
    PCSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

CreateWindowExW是接受Unicode字符串的函数版本。函数名结尾处的大写字母W是英文wide(宽)的缩写。每个Unicode字符的长度是16位，因此，它们常常称为宽字符。CreateWindowExA的结尾处的大写字母A表示该函数可以接受ANSI字符串。

但是，在我们的代码中，通常只包含了对 CreateWindowEx的调用，而不是直接调用CreateWindowExW或者CreateWindowExA。在WinUser.h文件中，CreateWindowEx实际上是定义为下面这种形式的一个宏：

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif // !UNICODE
```

当编译源代码模块时， UNICODE是否已经作了定义，将决定你调用的是哪个CreateWindowEx版本。当转用一个16位的Windows应用程序时，你在编译期间可能没有定义UNICODE。对CreateWindowEx函数的任何调用都会将该宏扩展为对CreateWindowExA的调用，即对CreateWindowEx的ANSI版本的调用。由于16位Windows只提供了CreateWindowsEx的ANSI版本，因此可以比较容易地转用它的应用程序。

在Windows 2000下，Microsoft的CreateWindowExA源代码只不过是一个形实替换程序层或翻译层，用于分配内存，以便将 ANSI字符串转换成Unicode字符串。该代码然后调用 CreateWindowExW，并传递转换后的字符串。当CreateWindowExW返回时，CreateWindowExA便释放它的内存缓存，并将窗口句柄返回给你。

如果要创建其他软件开发人员将要使用的动态链接库（ DLL），请考虑使用下面的方法。在DLL中提供两个输出函数。一个是ANSI版本，另一个是Unicode版本。在ANSI版本中，只需要分配内存，执行必要的字符串转换，并调用该函数的 Unicode版本（本章后面部分介绍这个进程）。

在Windows 98下，Microsoft的CreateWindowExA源代码是执行操作的函数。Windows 98提供了接受Unicode参数的所有Windows函数的进入点，但是这些函数并不将Unicode字符串转换成ANSI字符串，它们只返回运行失败的消息。调用 GetLastError将返回ERROR_CALL_NOT_IMPLEMENTED。这些函数中只有ANSI版本的函数才能正确地运行。如果编译的代码调用了任何宽字符函数，应用程序将无法在Windows 98下运行。

Windows API中的某些函数，比如WinExec和OpenFile等，只是为了实现与16位Windows程序的向后兼容而存在，因此，应该避免使用。应该使用对CreateProcess和CreateFile函数的调用来取代对WinExec和OpenFile函数的调用。从系统内部来讲，老的函数完全可以调用新的函数。老的函数存在的一个大问题是，它们不接受 Unicode字符串。当调用这些函数时，必须传递ANSI字符串。另一方面，所有新的和未过时的函数在 Windows 2000中都同时拥有ANSI和

China-pub.com
下载

Unicode两个版本。

2.8.4 Windows字符串函数

Windows还提供了一组范围很广的字符串操作函数。这些函数与C运行期字符串函数（如strcpy和wcscpy）很相似。但是该操作系统函数是操作系统的一个组成部分，操作系统的许多组件都使用这些函数，而不使用C运行期库。建议最好使用操作系统函数，而不要使用C运行期字符串函数。这将有助于稍稍提高你的应用程序的运行性能，因为操作系统字符串函数常常被大型应用程序比如操作系统的外壳进程Explorer.exe所使用。由于这些函数使用得很多，因此，在你的应用程序运行时，它们可能已经被装入RAM。

若要使用这些函数，系统必须运行Windows 2000或Windows 98。如果安装了Internet Explorer 4.0或更新的版本，也可以在较早的Windows版本中获得这些函数。

在经典的操作系统函数样式中，操作系统字符串函数名既包含大写字母，也包含小写字母，它的形式类似这个样子：StrCat、StrChr、StrCmp和StrCpy等。若要使用这些函数，必须加上ShlWApi.h头文件。另外，如前所述，这些字符串函数既有ANSI版本，也有Unicode版本，例如StrCatA和StrCatW。由于这些函数属于操作系统函数，因此，当创建应用程序时，如果定义了UNICODE（不带前置下划线），那么它们的符号将扩展为宽字符版本。

2.9 成为符合ANSI和Unicode的应用程序

即使你不打算立即使用Unicode，最好也应该着手将你的应用程序转换成符合Unicode的应用程序。下面是应该遵循的一些基本原则：

- 将文本串视为字符数组，而不是chars数组或字节数组。
- 将通用数据类型（如TCHAR和PTSTR）用于文本字符和字符串。
- 将显式数据类型（如BYTE和PBYTE）用于字节、字节指针和数据缓存。
- 将TEXT宏用于原义字符和字符串。
- 执行全局性替换（例如用PTSTR替换PSTR）。
- 修改字符串运算问题。例如函数通常希望你在字符中传递一个缓存的大小，而不是字节。这意味着你不应该传递sizeof(szBuffer),而应该传递（sizeof(szBuffer)/sizeof(TCHAR)。另外，如果需要为字符串分配一个内存块，并且拥有该字符串中的字符数目，那么请记住要按字节来分配内存。这就是说，应该调用malloc(nCharacters *sizeof(TCHAR)),而不是调用malloc(nCharacters)。在上面所说的所有原则中，这是最难记住的一条原则，如果操作错误，编译器将不发出任何警告。

当我为本书的第一版编写示例程序时，我编写的原始程序只能编译为ANSI程序。后来，当我开始撰写本章的内容时，我想我应该鼓励使用Unicode，并且打算创建一些示例程序，以便展示你可以非常容易地编写既可以用Unicode也可以用ANSI来编译的程序。这时我发现最好的办法是将本书的所有示例程序进行转换，使它们都能够用Unicode和ANSI进行编译。

我用了大约4个小时将所有程序进行了转换。考虑到我以前从来没有这方面的转换经验，这个速度是相当不错了。

2.9.1 Windows字符串函数

Windows也提供了一组用于对Unicode字符串进行操作的函数，表2-4对它们进行了描述。

China-pub.com 下载

表2-4 对Unicode字符串进行操作的函数

函 数	描 述
lstrcat	将一个字符串置于另一个字符串的结尾处
lstrcmp	对两个字符串进行区分大小写的比较
lstrcmpi	对两个字符串进行不区分大小写的比较
lstrcpy	将一个字符串拷贝到内存中的另一个位置
lstrlen	返回字符串的长度（按字符数来计量）

这些函数是作为宏来实现的，这些宏既可以调用函数的Unicode版本，也可以调用函数的ANSI版本，这要根据编译源代码模块时是否已经定义了UNICODE而定。例如，如果没有定义UNICODE，lstrcat函数将扩展为lstrcatA。如果定义了UNICODE，lstrcat将扩展为lstrcatW。

有两个字符串函数，即lstrcmp和lstrcmpi，它们的行为特性与等价的C运行期函数是不同的。C运行期函数strcmp、strcmpi、wcscmp和wcscmpi只是对字符串中的代码点的值进行比较，这就是说，这些函数将忽略实际字符的含义，只是将第一个字符串中的每个字符的数值与第二个字符串中的字符的数值进行比较。而Windows函数lstrcmp和lstrcmpi是作为对Windows函数CompareString的调用来实现的。

```
int CompareString(
   LCID lcid,
   DWORD fdwStyle,
   PCWSTR pString1,
   int cch1,
   PCTSTR pString2,
   int cch2);
```

该函数对两个Unicode字符串进行比较。CompareString的第一个参数用于设定语言ID（LCID），这是个32位值，用于标识一种特定的语言。CompareString使用这个LCID来比较这两个字符串，方法是对照一种特定的语言来查看它们的字符的含义。这种操作方法比C运行期函数简单地进行数值比较更有意义。

当lstrcmp函数系列中的任何一个函数调用CompareString时，该函数便将调用Windows的GetThreadString函数的结果作为第一个参数来传递：

```
LCID GetThreadLocale();
```

每次创建一个线程时，它就被赋予一种语言。函数将返回该线程的当前语言设置。

CompareString的第二个参数用于标识一些标志，这些标志用来修改该函数比较两个字符串时所用的方法。表2-5显示了可以使用的标志。

表2-5 Compare String 的标志及含义

标 志	含 义
NORM_IGNORECASE	忽略字母的大小写
NORM_IGNOREKANATYPE	不区分平假名与片假名字符
NORM_IGNORENONSPACE	忽略无间隔字符
NORM_IGNORESYMBOLS	忽略符号
NORM_IGNOREWIDTH	不区分单字节字符与作为双字节字符的同一个字符
SORT_STRINGSORT	将标点符号作为普通符号来处理

当lstrcmp调用CompareString时，它传递0作为fdwStyle的参数。但是，当lstrcmpi调用CompareString时，它就传递NORM_IGNORECASE。CompareString的其余4个参数用于设定两

China-pub.com
下载

个字符串和它们各自的长度。如果为cch1参数传递-1，那么该函数将认为pString1字符串是以0结尾，并计算该字符串的长度。对于pString2字符串来说，参数cch2的作用也是一样。

其他C运行期函数没有为Unicode字符串的操作提供很好的支持。例如， tolower和toupper函数无法正确地转换带有重音符号的字符。为了弥补 C运行期库中的这些不足，必须调用下面这些Windows函数，以便转换Unicode字符串的大小写字母。这些函数也可以正确地用于 ANSI字符串。

头两个函数：

```
PTSTR CharLower(PTSTR pszString);
```

和

```
PTSTR CharUpper(PTSTR pszString);
```

既可以转换单个字符，也可以转换以 0结尾的整个字符串。若要转换整个字符串，只需要传递字符串的地址即可。若要转换单个字符，必须像下面这样传递各个字符：

```
TCHAR cLowerCaseChar = CharLower((PTSTR) szString[0]);
```

将单个字符转换成一个PTSTR，便可调用该函数，将一个值传递给它，在这个值中，较低的16位包含了该字符，较高的16位包含0。当该函数看到较高位是0时，该函数就知道你想要转换单个字符，而不是整个字符串。返回的值是个32位值，较低的16位中是已经转换的字符。

下面两个函数与前面两个函数很相似，差别在于它们用于转换缓存中包含的字符（该缓存不必以0结尾）：

```
DWORD CharLowerBuff(
   PTSTR pszString,
   DWORD cchString);
DWORD CharUpperBuff(
   PTSTR pszString,
   DWORD cchString);
```

其他的C运行期函数，如isalpha、islower和isupper，返回一个值，指明某个字符是字母字符、小写字母还是大写字母。 Windows API提供了一些函数，也能返回这些信息，但是Windows函数也要考虑用户在控制面板中指定的语言：

```
BOOL IsCharAlpha(TCHAR ch);
BOOL IsCharAlphaNumeric(TCHAR ch);
BOOL IsCharLower(TCHAR ch);
BOOL IsCharUpper(TCHAR ch);
```

printf函数家族是要介绍的最后一组C运行期函数。如果在定义了 _UNICODE的情况下编译你的源代码模块，那么printf函数家族便希望所有字符和字符串参数代表Unicode字符和字符串。但是，如果在没有定义 _UNICODE的情况下编译你的源代码模块， printf函数家族便希望传递给它的所有字符和字符串都是ANSI字符和字符串。

Microsoft公司已经给C运行期的printf函数家族增加了一些特殊的域类型。其中有些域类型尚未被ANSI C采用。新类型使你能够很容易地对ANSI和Unicode字符和字符串进行混合和匹配。操作系统的wsprintf函数也得到了增强。下面是一些例子（请注意大写 S和小写s的使用）：

```
char  szA[100];    // An ANSI string buffer
WCHAR szW[100];    // A Unicode string buffer

// Normal sprintf: all strings are ANSI
sprintf(szA,   "%s",   "ANSI Str");
```

China-pub.com 下载

```
// Converts Unicode string to ANSI
sprintf(szA,   "%S",  L"Unicode Str");

// Normal swprintf: all strings are Unicode
swprintf(szW, L"%s",  L"Unicode Str");

// Converts ANSI string to Unicode
swprintf(szW, L"%S",   "ANSI Str");
```

2.9.2 资源

当资源编译器对你的所有资源进行编译时，输出文件是资源的二进制文件。资源（字符串表、对话框模板和菜单等）中的字符串值总是写作Unicode字符串。在Windows 98和Windows 2000下，如果应用程序没有定义UNICODE宏，那么系统就会进行内部转换。

例如，如果在编译源代码模块时没有定义UNICODE，调用LoadString实际上就是调用LoadStringA函数。这时LoadStringA就从你的资源中读取字符串，并将该字符串转换成ANSI字符串。ANSI形式的字符串将从该函数返回给你的应用程序。

2.9.3 确定文本是ANSI文本还是Unicode文本

到现在为止，Unicode文本文件仍然非常少。实际上，Microsoft公司自己的大多数产品并没有配备任何Unicode文本文件。但是预计将来这种情况是会改变的（尽管这需要一个很长的过程）。当然，Windows 2000的Notepad(记事本)应用程序允许你既能打开Unicode文件，也能打开ANSI文件，并且可以创建这些文件。图2-1显示了Notepad的Save As（文件另存为）对话框。请注意可以用不同的方法来保存文本文件。



图2-1 Windows 2000 Notepad的File Save As对话框

对于许多用来打开文本文件和处理这些文件的应用程序（如编译器）来说，打开一个文件后，应用程序就能方便地确定该文本文件是包含ANSI字符还是Unicode字符。IsTextUnicode函数能够帮助进行这种区分：

```
DWORD IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

文本文件存在的问题是，它们的内容没有严格和明确的规则，因此很难确定该文件是包含

China-pub.com
下载

ANSI字符还是Unicode字符。IsTextUnicode使用一系列统计方法和定性方法，以便猜测缓存的内容。由于这不是一种确切的科学方法，因此IsTextUnicode有可能返回不正确的结果。

第一个参数pvBuffer用于标识要测试的缓存的地址。该数据是个无效指针，因为你不知道你拥有的是ANSI字符数组还是Unicode字符数组。

第二个参数cb用于设定pvBuffer指向的字节数。同样，由于你不知道缓存中放的是什么，因此cb是个字节数，而不是字符数。请注意，不必设定缓存的整个长度。当然，IsTextUnicode能够测试的字节越多，得到的结果越准确。

第三个参数pResult是个整数的地址，必须在调用IsTextUnicode之前对它进行初始化。对该整数进行初始化后，就可以指明你要IsTextUnicode执行哪些测试。也可以为该参数传递NULL，在这种情况下，IsTextUnicode将执行它能够进行的所有测试（详细说明请参见Platform SDK文档）。

如果IsTextUnicode认为缓存包含Unicode文本，便返回TRUE，否则返回FALSE。确实是这样，尽管Microsoft将该函数的原型规定为返回DWORD，但是它实际上返回一个布尔值。如果在pResult参数指向的整数中必须进行特定的测试，该函数就会在返回之前设定整数中的信息位，以反映每个测试的结果。

Windows98 在Windows 98下，IsTextUnicode函数没有有用的实现代码，它只是返回FALSE。调用GetLastError函数将返回ERROR_CALL_NOT_IMPLEMENTD。

第17章中的Flie Rev示例应用程序演示了IsTextUnicode函数的使用。

2.9.4 在Unicode与ANSI之间转换字符串

Windows函数MultiByteToWideChar用于将多字节字符串转换成宽字符串。下面显示了MultiByteToWideChar函数。

```
int MultiByteToWideChar(
   UINT uCodePage,
   DWORD dwFlags,
   PCSTR pMultiByteStr,
   int cchMultiByte,
   PWSTR pWideCharStr,
   int cchWideChar);
```

uCodePage参数用于标识一个与多字节字符串相关的代码页号。dwFlags参数用于设定另一个控件，它可以用重音符号之类的区分标记来影响字符。这些标志通常并不使用，在dwFlags参数中传递0。pMultiByteStr参数用于设定要转换的字符串，cchMultiByte参数用于指明该字符串的长度（按字节计算）。如果为cchMultiByte参数传递-1，那么该函数用于确定源字符串的长度。

转换后产生的Unicode版本字符串将被写入内存中的缓存，其地址由pWideCharStr参数指定。必须在cchWideChar参数中设定该缓存的最大值（以字符为计量单位）。如果调用MultiByteToWideChar，给cchWideChar参数传递0，那么该参数将不执行字符串的转换，而是返回为使转换取得成功所需要的缓存的值。一般来说，可以通过下列步骤将多字节字符串转换成Unicode等价字符串：

1) 调用MultiByteToWideChar函数，为pWideCharStr参数传递NULL，为cchWideChar参数传递0。

2) 分配足够的内存块，用于存放转换后的Unicode字符串。该内存块的大小由前面对

China-pub.com 下载

MultByteToWideChar的调用返回。

3) 再次调用MultiByteToWideChar，这次将缓存的地址作为pWideCharStr参数来传递，并传递第一次调用MultiByteToWideChar时返回的缓存大小，作为cchWidechar参数。

4. 使用转换后的字符串。

5) 释放Unicode字符串占用的内存块。

函数WideCharToMultiByte将宽字符串转换成等价的多字节字符串，如下所示：

```
int WideCharToMultiByte(
   UINT uCodePage,
   DWORD dwFlags,
   PCWSTR pWideCharStr,
   int cchWideChar,
   PSTR pMultiByteStr,
   int cchMultiByte,
   PCSTR pDefaultChar,
   PBOOL pfUsedDefaultChar);
```

该函数与MultiBiteToWideChar函数相似。同样，uCodePage参数用于标识与新转换的字符串相关的代码页。dwFlags则设定用于转换的其他控件。这些标志能够作用于带有区分符号的字符和系统不能转换的字符。通常不需要为字符串的转换而拥有这种程度的控制手段，你将为dwFlags参数传递0。

pWideCharStr参数用于设定要转换的字符串的内存地址，cchWideChar参数用于指明该字符串的长度（用字符数来计量）。如果你为cchWideChar参数传递-1，那么该函数用于确定源字符串的长度。

转换产生的多字节版本的字符串被写入由pMultiByteStr参数指明的缓存。必须在cchMultiByte参数中设定该缓存的最大值（用字节来计量）。如果传递0作为WideCharToMultiByte函数的cchMultiByte参数，那么该函数将返回目标缓存需要的大小值。通常可以使用将多字节字符串转换成宽字节字符串时介绍的一系列类似的事件，将宽字节字符串转换成多字节字符串。

你会发现，WideCharToMultiByte函数接受的参数比MultiByteToWideChar函数要多2个，即pDefaultChar和pfUsedDefaultChar。只有当WideCharToMultiByte函数遇到一个宽字节字符，而该字符在uCodePage参数标识的代码页中并没有它的表示法时，WideCharToMultiByte函数才使用这两个参数。如果宽字节字符不能被转换，该函数便使用 pDefaultChar参数指向的字符。如果该参数是NULL（这是大多数情况下的参数值），那么该函数使用系统的默认字符。该默认字符通常是个问号。这对于文件名来说是危险的，因为问号是个通配符。

pfUsedDefaultChar参数指向一个布尔变量，如果宽字符串中至少有一个字符不能转换成等价多字节字符，那么函数就将该变量置为 TRUE。如果所有字符均被成功地转换，那么该函数就将该变量置为FALSE。当函数返回以便检查宽字节字符串是否被成功地转换后，可以测试该变量。同样，通常为该测试传递NULL。

关于如何使用这些函数的详细说明，请参见Platform SDK文档。

如果使用这两个函数，就可以很容易创建这些函数的Unicode版本和ANSI版本。例如，你可能有一个动态链接库，它包含一个函数，能够转换字符串中的所有字符。可以像下面这样编写该函数的Unicode版本：

```
BOOL StringReverseW(PWSTR pWideCharStr) {

   // Get a pointer to the last character in the string.
   PWSTR pEndOfStr = pWideCharStr + wcslen(pWideCharStr) - 1;
```

China-pub.com
下载


```
   wchar_t cCharT;
   // Repeat until we reach the center character in the string.
   while (pWideCharStr < pEndOfStr) {
      // Save a character in a temporary variable.
      cCharT = *pWideCharStr;

      // Put the last character in the first character.
      *pWideCharStr = *pEndOfStr;

      // Put the temporary character in the last character.
      *pEndOfStr = cCharT;

      // Move in one character from the left.
      pWideCharStr++;

      // Move in one character from the right.
      pEndOfStr--;
   }

   // The string is reversed; return success.
   return(TRUE);
}
```

你可以编写该函数的ANSI版本以便该函数根本不执行转换字符串的实际操作。你也可以编写该函数的ANSI版本，以便该函数它将ANSI字符串转换成Unicode字符串，将Unicode字符串传递给StringReverseW函数，然后将转换后的字符串重新转换成ANSI字符串。该函数类似下面的样子：

```
BOOL StringReverseA(PSTR pMultiByteStr) {
   PWSTR pWideCharStr;
   int nLenOfWideCharStr;
   BOOL fOk = FALSE;

   // Calculate the number of characters needed to hold
   // the wide-character version of the string.
   nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
      pMultiByteStr, -1, NULL, 0);

   // Allocate memory from the process's default heap to
   // accommodate the size of the wide-character string.
   // Don't forget that MultiByteToWideChar returns the
   // number of characters, not the number of bytes, so
   // you must multiply by the size of a wide character.
   pWideCharStr = HeapAlloc(GetProcessHeap(), 0,
      nLenOfWideCharStr * sizeof(WCHAR));

   if (pWideCharStr == NULL)
      return(fOk);

   // Convert the multibyte string to a wide-character string.
   MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, -1,
      pWideCharStr, nLenOfWideCharStr);

   // Call the wide-character version of this
   // function to do the actual work.
   fOk = StringReverseW(pWideCharStr);
```

China-pub.com 下载

```
   if (fOk) {
      // Convert the wide-character string back
      // to a multibyte string.
      WideCharToMultiByte(CP_ACP, 0, pWideCharStr, -1,
         pMultiByteStr, strlen(pMultiByteStr), NULL, NULL);
   }

   // Free the memory containing the wide-character string.
   HeapFree(GetProcessHeap(), 0, pWideCharStr);

   return(fOk);
}
```

最后，在用动态链接库分配的头文件中，可以像下面这样建立这两个函数的原型：

```
BOOL StringReverseW(PWSTR pWideCharStr);
BOOL StringReverseA(PSTR pMultiByteStr);

#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

China-pub.com
下载

第3章 内核对象

在介绍Windows API的时候，首先要讲述内核对象以及它们的句柄。本章将要介绍一些比较抽象的概念，在此并不讨论某个特定内核对象的特性，相反只是介绍适用于所有内核对象的特性。

首先介绍一个比较具体的问题，准确地理解内核对象对于想要成为一名Windows软件开发能手的人来说是至关重要的。内核对象可以供系统和应用程序使用来管理各种各样的资源，比如进程、线程和文件等。本章讲述的概念也会出现在本书的其他各章之中。但是，在你开始使用实际的函数来操作内核对象之前，是无法深刻理解本章讲述的部分内容的。因此当阅读本书的其他章节时，可能需要经常回过来参考本章的内容。

3.1 什么是内核对象

作为一个Windows软件开发人员，你经常需要创建、打开和操作各种内核对象。系统要创建和操作若干类型的内核对象，比如存取符号对象、事件对象、文件对象、文件映射对象、I/O完成端口对象、作业对象、信箱对象、互斥对象、管道对象、进程对象、信标对象、线程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，CreateFileMapping函数可使系统能够创建一个文件映射对象。每个内核对象只是内核分配的一个内存块，并且只能由该内核访问。该内存块是一种数据结构，它的成员负责维护该对象的各种信息。有些数据成员（如安全性描述符、使用计数等）在所有对象类型中是相同的，但大多数数据成员属于特定的对象类型。例如，进程对象有一个进程ID、一个基本优先级和一个退出代码，而文件对象则拥有一个字节位移、一个共享模式和一个打开模式。

由于内核对象的数据结构只能被内核访问，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。Microsoft规定了这个限制条件，目的是为了确保内核对象结构保持状态的一致。这个限制也使Microsoft能够在不破坏任何应用程序的情况下在这些结构中添加、删除和修改数据成员。

如果我们不能直接改变这些数据结构，那么我们的应用程序如何才能操作这些内核对象呢？解决办法是，Windows提供了一组函数，以便用定义得很好的方法来对这些结构进行操作。这些内核对象始终都可以通过这些函数进行访问。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。该句柄可以被视为一个不透明值，你的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，这样，系统就能知道你想操作哪个内核对象。本章后面还要详细讲述这些句柄的特性。

为了使操作系统变得更加健壮，这些句柄值是与进程密切相关的。因此，如果将该句柄值传递给另一个进程中的一个线程（使用某种形式的进程间的通信）那么这另一个进程使用你的进程的句柄值所作的调用就会失败。在3.3节“跨越进程边界共享内核对象”中，将要介绍3种机制，使多个进程能够成功地共享单个内核对象。

3.1.1 内核对象的使用计数

内核对象由内核所拥有，而不是由进程所拥有。换句话说，如果你的进程调用了一个创建

China-pub.com 下载

内核对象的函数，然后你的进程终止运行，那么内核对象不一定被撤消。在大多数情况下，对象将被撤消，但是如果另一个进程正在使用你的进程创建的内核对象，那么该内核知道，在另一个进程停止使用该对象前不要撤消该对象，必须记住的是，内核对象的存在时间可以比创建该对象的进程长。

内核知道有多少进程正在使用某个内核对象，因为每个对象包含一个使用计数。使用计数是所有内核对象类型常用的数据成员之一。当一个对象刚刚创建时，它的使用计数被置为 1。然后，当另一个进程访问一个现有的内核对象时，使用计数就递增 1。当进程终止运行时，内核就自动确定该进程仍然打开的所有内核对象的使用计数。如果内核对象的使用计数降为 0，内核就撤消该对象。这样可以确保在没有进程引用该对象时系统中不保留任何内核对象。

3.1.2 安全性

内核对象能够得到安全描述符的保护。安全描述符用于描述谁创建了该对象，谁能够访问或使用该对象，谁无权访问该对象。安全描述符通常在编写服务器应用程序时使用，如果你编写客户机端的应用程序，那么可以忽略内核对象的这个特性。

> Windows 98 根据原来的设计，Windows 98并不用作服务器端的操作系统。为此，Microsoft公司没有在Windows 98中配备安全特性。不过，如果你现在为 Windows 98 设计软件，在实现你的应用程序时仍然应该了解有关的安全问题，并且使用相应的访问信息，以确保它能在Windows 2000上正确地运行

用于创建内核对象的函数几乎都有一个指向 SECURITY_ATTRIBUTES结构的指针作为其参数，下面显示了CreateFileMapping函数的指针：

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);
```

大多数应用程序只是为该参数传递 NULL，这样就可以创建带有默认安全性的内核对象。默认安全性意味着对象的管理小组的任何成员和对象的创建者都拥有对该对象的全部访问权，而其他所有人均无权访问该对象。但是，可以指定一个 SECURITY_ATTRIBUTES结构，对它进行初始化，并为该参数传递该结构的地址。SECURITY_ATTRIBUTES结构类似下面的样子：

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

尽管该结构称为SECURITY_ATTRIBUTES，但是它包含的与安全性有关的成员实际上只有一个，即lpSecurityDescriptor。如果你想要限制人们对你创建的内核对象的访问，必须创建一个安全性描述符，然后像下面这样对SECURITY_ATTRIBUTES结构进行初始化：

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);          // Used for versioning
sa.lpSecurityDescriptor = pSD;    // Address of an initialized SD
sa.bInheritHandle = FALSE;        // Discussed later
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
```

China-pub.com
下载


```
                                                    , dsa,
   PAGE_READWRITE, 0, 1024, "MyFileMapping");
.
.
.
```

由于bInheritHandle这个成员与安全性毫无关系，因此准备推迟到本章后面部分继承性一节中再介绍bInheritHandle这个成员。

当你想要获得对相应的一个内核对象的访问权（而不是创建一个新对象）时，必须设定要对该对象执行什么操作。例如，如果想要访问一个现有的文件映射内核对象，以便读取它的数据，那么应该调用下面这个OpenfileMapping函数：

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
   "MyFileMapping");
```

通过将FILE_MAP_READ作为第一个参数传递给OpenFileMapping，指明打算在获得对该文件映象的访问权后读取该文件，OpenFileMapping函数在返回一个有效的句柄值之前，首先执行一次安全检查。如果（已登录用户）被允许访问现有的文件映射内核对象，OpenFileMapping就返回一个有效的句柄。但是，如果被拒绝访问该对象，OpenFileMapping将返回NULL，而调用GetLastError函数则返回5（ERROR_ACCESS_DENIED），同样，大多数应用程序并不使用该安全性，因此将不进一步讨论这个问题。

Windows 98 虽然许多应用程序不需要考虑安全性问题，但是Windows的许多函数要求传递必要的安全访问信息。为Windows 98设计的若干应用程序在Windows 2000上无法正确地运行，因为在实现这些应用程序时没有对安全问题给于足够的考虑。

例如，假设一个应用程序在开始运行时要从注册表的子关键字中读取一些数据。为了正确地进行这项操作，你的代码应该调用 RegOpenKeyEx，传递KEY_QUERY_VALUE，以便获得必要的访问权。

但是，许多应用程序原先是为Windows 98开发的，当时没有考虑到运行Windows 2000的需要。由于Windows 98没有解决注册表的安全问题，因此软件开发人员常常要调用RegOpenKeyEx函数，传递KEY_All_ACCESS，作为必要的访问权。开发人员这样做的原因是，它是一种比较简单的解决方案，意味着开发人员不必考虑究竟需要什么访问权。问题是注册表的子关键字可以被用户读取，但是不能写入。

因此，当该应用程序现在放在Windows 2000上运行时，用KEY_ALL_ACCESS调用RegOpenKeyEx就会失败，而且，没有相应的错误检查方法，应用程序的运行就会产生不可预料的结果。

如果开发人员想到安全问题，把KEY_ALL_ACCESS改为KEY_QUERY_VALUE，则该产品可适用于两种操作系统平台。

开发人员的最大错误之一就是忽略安全访问标志。使用正确的标志会使最初为Windows 98 设计的应用程序更易于向Windows 2000 转换。

除了内核对象外，你的应用程序也可以使用其他类型的对象，如菜单、窗口、鼠标光标、刷子和字体等。这些对象属于用户对象或图形设备接口（GDI）对象，而不是内核对象。当初次着手为Windows编程时，如果想要将用户对象或GDI对象与内核对象区分开来，你一定会感到不知所措。比如，图标究竟是用户对象还是内核对象呢？若要确定一个对象是否属于内核对象，最容易的方法是观察创建该对象所用的函数。创建内核对象的所有函数几乎都有一个参数，你可以用来设定安全属性的信息，这与前面讲到的CreateFileMapping函数是相同的。

用于创建用户对象或GDI对象的函数都没有PSECURITY_ATTRIBUTES参数。例如，让我们来看一看下面这个CreateIcon函数：

China-pub.com 下载

```
HICON CreateIcon(
   HINSTANCE hinst,
   int nWidth,
   int nHeight,
   BYTE cPlanes,
   BYTE cBitsPixel,
   CONST BYTE *pbANDbits,
   CONST BYTE *pbXORbits);
```

3.2 进程的内核对象句柄表

当一个进程被初始化时,系统要为它分配一个句柄表。该句柄表只用于内核对象不用于用户对象或GDI对象。句柄表的详细结构和管理方法并没有具体的资料说明。通常我并不介绍操作系统中没有文档资料的那些部分。不过，在这种情况下，我会进行例外处理，因为，作为一个称职的Windows程序员，必须懂得如何管理进程的句柄表。由于这些信息没有文档资料，因此不能保证所有的详细信息都正确无误，同时，在Windows 2000、Windows 98和Windows CE中，它们的实现方法是不同的。为此，请认真阅读下面介绍的内容以加深理解，在此不学习系统是如何进行操作的。

表3-1显示了进程的句柄表的样子。可以看到，它只是个数据结构的数组。每个结构都包含一个指向内核对象的指针、一个访问屏蔽和一些标志。

表3-1 进程的句柄结构

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
…	…	…	…

3.2.1 创建内核对象

当进程初次被初始化时，它的句柄表是空的。然后，当进程中的线程调用创建内核对象的函数时，比如CreateFileMapping，内核就为该对象分配一个内存块，并对它初始化。这时，内核对进程的句柄表进行扫描，找出一个空项。由于表3-1中的句柄表是空的，内核便找到索引1位置上的结构并对它进行初始化。该指针成员将被设置为内核对象的数据结构的内存地址，访问屏蔽设置为全部访问权，同时，各个标志也作了设置（关于标志，将在本章后面部分的继承性一节中介绍）。

下面列出了用于创建内核对象的一些函数（但这决不是个完整的列表）：

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD dwCreationFlags,
   PDWORD pdwThreadId);

HANDLE CreateFile(
   PCTSTR pszFileName,
   DWORD dwDesiredAccess,
```

China-pub.com
下载


```
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

用于创建内核对象的所有函数均返回与进程相关的句柄，这些句柄可以被在相同进程中运行的任何或所有线程成功地加以使用。该句柄值实际上是放入进程的句柄表中的索引，它用于标识内核对象的信息存放的位置。因此当调试一个应用程序并且观察内核对象句柄的实际值时，会看到一些较小的值，如1，2等。请记住，句柄的含义并没有记入文档资料，并且可能随时变更。实际上在Windows 2000中，返回的值用于标识放入进程的句柄表的该对象的字节数，而不是索引号本身。

每当调用一个将内核对象句柄接受为参数的函数时，就要传递由一个 Create*&函数返回的值。从内部来说，该函数要查看进程的句柄表，以获取要生成的内核对象的地址，然后按定义得很好的方式来生成该对象的数据结构。

如果传递了一个无效索引（句柄），该函数便返回失败，而 GetLastError则返回 6（ERROR_INVALID_HANDLE）。由于句柄值实际上是放入进程句柄表的索引，因此这些句柄是与进程相关的，并且不能由其他进程成功地使用。

如果调用一个函数以便创建内核对象，但是调用失败了，那么返回的句柄值通常是 0（NULL）。发生这种情况是因为系统的内存非常短缺，或者遇到了安全方面的问题。不过有少数函数在运行失败时返回的句柄值是-1（INVALID_HANDLE_VALUE）。例如，如果CreateFile未能打开指定的文件，那么它将返回INVALID_HANDLE_VALUE，而不是返回NULL。当查看创建内核对象的函数返回值时，必须格外小心。特别要注意的是，只有当调用CreateFile函数时，才能将该值与INVALID_HANDLE_VALUE进行比较。下面的代码是不正确的：

```
HANDLE hMutex = CreateMutex(…);
if (hMutex == INVALID_HANDLE_VALUE) {
   // We will never execute this code because
   // CreateMutex returns NULL if it fails.
}
```

同样，下面的代码也不正确：

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
   // We will never execute this code because CreateFile
   // returns INVALID_HANDLE_VALUE(-1) if it fails.
}
```

China-pub.com 下载

3.2.2 关闭内核对象

无论怎样创建内核对象，都要向系统指明将通过调用CloseHandle来结束对该对象的操作：

```
BOOL CloseHandle(HANDLE hobj);
```

该函数首先检查调用进程的句柄表，以确保传递给它的索引（句柄）用于标识一个进程实际上无权访问的对象。如果该索引是有效的，那么系统就可以获得内核对象的数据结构的地址，并可确定该结构中的使用计数的数据成员。如果使用计数是0，该内核便从内存中撤消该内核对象。

如果将一个无效句柄传递给 CloseHandle，将会出现两种情况之一。如果进程运行正常，CloseHandle返回FALSE，而GetLastError则返回ERROR_INVALID_HANDLE。如果进程正在排除错误，系统将通知调试程序，以便能排除它的错误。

在CloseHandle返回之前，它会清除进程的句柄表中的项目，该句柄现在对你的进程已经无效，不应该试图使用它。无论内核对象是否已经撤消，都会发生清除操作。当调用CloseHandle函数之后，将不再拥有对内核对象的访问权，不过，如果该对象的使用计数没有递减为0，那么该对象尚未被撤消。这没有问题，它只是意味着一个或多个其他进程正在使用该对象。当其他进程停止使用该对象时（通过调用 CloseHandle），该对象将被撤消。

假如忘记调用CloseHandle函数，那么会不会出现内存泄漏呢？答案是可能的，但是也不一定。在进程运行时，进程有可能泄漏资源（如内核对象）。但是，当进程终止运行时，操作系统能够确保该进程使用的任何资源或全部资源均被释放，这是有保证的。对于内核对象来说，系统将执行下列操作：当进程终止运行时，系统会自动扫描进程的句柄表。如果该表拥有任何无效项目（即在终止进程运行前没有关闭的对象），系统将关闭这些对象句柄。如果这些对象中的任何对象的使用计数降为0，那么内核便撤消该对象。

因此，应用程序在运行时有可能泄漏内核对象，但是当进程终止运行时，系统将能确保所有内容均被正确地清除。另外，这个情况适用于所有对象、资源和内存块，也就是说，当进程终止运行时，系统将保证进程不会留下任何对象。

3.3 跨越进程边界共享内核对象

许多情况下，在不同进程中运行的线程需要共享内核对象。下面是为何需要共享的原因：

- 文件映射对象使你能够在同一台机器上运行的两个进程之间共享数据块。
- 邮箱和指定的管道使得应用程序能够在连网的不同机器上运行的进程之间发送数据块。
- 互斥对象、信标和事件使得不同进程中的线程能够同步它们的连续运行，这与一个应用程序在完成某项任务时需要将情况通知另一个应用程序的情况相同。

由于内核对象句柄与进程相关，因此这些任务的执行情况是不同的。不过， Microsoft公司有若干很好的理由将句柄设计成与进程相关的句柄。最重要的理由是要实现它的健壮性。如果内核对象句柄是系统范围的值，那么一个进程就能很容易获得另一个进程使用的对象的句柄，从而对该进程造成很大的破坏。另一个理由是安全性。内核对象是受安全性保护的，进程在试图操作一个对象之前，首先必须申请获得操作该对象的许可权。对象的创建人只需要拒绝向用户赋予许可权，就能防止未经授权的用户接触该对象。

在下面的各节中，将要介绍允许进程共享内核对象的3个不同的机制。

3.3.1 对象句柄的继承性

只有当进程具有父子关系时，才能使用对象句柄的继承性。在这种情况下，父进程可以使

China-pub.com
下载

用一个或多个内核对象句柄，并且该父进程可以决定生成一个子进程，为子进程赋予对父进程的内核对象的访问权。若要使这种类型的继承性能够实现，父进程必须执行若干个操作步骤。

首先，当父进程创建内核对象时，必须向系统指明，它希望对象的句柄是个可继承的句柄。请记住，虽然内核对象句柄具有继承性，但是内核对象本身不具备继承性。

若要创建能继承的句柄，父进程必须指定一个 SECURITY_ATTRIBUTES结构并对它进行初始化，然后将该结构的地址传递给特定的 Create函数。下面的代码用于创建一个互斥对象，并将一个可继承的句柄返回给它：

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;   // Make the returned handle inheritable.

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);
:
:
```

该代码对一个SECURITY_ATTRIBUTES结构进行初始化，指明该对象应该使用默认安全性（在Windows 98中该安全性被忽略）来创建，并且返回的句柄应该是可继承的。

Windows 98 尽管Windows 98不拥有完整的对安全性的支持，但是它却支持继承性，因此，Windows 98能够正确地使用bInheritHandle成员的值。

现在介绍存放在进程句柄表项目中的标志。每个句柄表项目都有一个标志位，用来指明该句柄是否具有继承性。当创建一个内核对象时，如果传递NULL作为PSECURITY_ATTRIBUTES的参数，那么返回的句柄是不能继承的，并且该标志位是 0。如果将bInheritHandle成员置为TRUE，那么该标志位将被置为1。

表3-2显示了一个进程的句柄表。

表3-2 包含两个有效项目的进程句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(无)	(无)
3	0xF0000010	0x????????	0x00000001

表3-2表示该进程拥有对两个内核对象（句柄1和3）的访问权。句柄1是不可继承的，而句柄3是可继承的。

使用对象句柄继承性时要执行的下一个步骤是让父进程生成子进程。这要使用 CreateProcess函数来完成：

```
BOOL CreateProcess(
   PCTSTR pszApplicationName,
   PTSTR pszCommandLine,
   PSECURITY_ATTRIBUTES psaProcess,
   PSECURITY_ATTRIBUTES pszThread,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   PVOID pvEnvironment,
   PCTSTR pszCurrentDirectory,
   LPSTARTUPINFO pStartupInfo,
   PPROCESS_INFORMATION pProcessInformation);
```

China-pub.com 下载

下一章将详细介绍这个函数的用法，不过现在我想要让你注意bInheritHandle这个参数。一般来说，当生成一个进程时，将为该参数传递FALSE。该值告诉系统，不希望子进程继承父进程的句柄表中的可继承句柄。

但是，如果为该参数传递TRUE，那么子进程就可以继承父进程的可继承句柄值。当传递TRUE时，操作系统就创建该新子进程，但是不允许子进程立即开始执行它的代码。当然，系统为子进程创建一个新的和空的句柄表，就像它为任何新进程创建句柄表那样。不过，由于将TRUE传递给了CreateProcess的bInheritHandles参数，因此系统要进行另一项操作，即它要遍历父进程的句柄表，对于它找到的包含有效的可继承句柄的每个项目，系统会将该项目准确地拷贝到子进程的句柄表中。该项目拷贝到子进程的句柄表中的位置将与父进程的句柄表中的位置完全相同。这个情况非常重要，因为它意味着在父进程与子进程中，标识内核对象所用的句柄值是相同的。

除了拷贝句柄表项目外，系统还要递增内核对象的使用计数，因为现在两个进程都使用该对象。如果要撤消内核对象，那么父进程和子进程必须调用该对象上的CloseHandle函数，也可以终止进程的运行。子进程不必首先终止运行，但是父进程也不必首先终止运行。实际上，CreateProcess函数返回后，父进程可以立即关闭对象的句柄，而不影响子进程对该对象进行操作的能力。

表3-3显示了子进程被允许运行前该进程的句柄表。可以看到，项目1和项目2尚未初始化，因此是个无效句柄，子进程是无法使用的。但是，项目3确实标识了一个内核对象。实际上，它标识的内核对象的地址是0xF0000010，这与父进程的句柄表中的对象地址相同。访问屏蔽与父进程中的屏蔽相同，两者的标志也相同。这意味着如果该子进程要生成它自己的子进程（即父进程的孙进程），该孙进程也将继承与该内核对象句柄相同的句柄值、同样的访问权和相同的标志，同时，对象的使用计数再次被递增。

表3-3 继承父进程的可继承句柄后的子进程句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	（无）	（无）
2	0x00000000	（无）	（无）
3	0xF0000010	0x????????	0x00000001

应该知道，对象句柄的继承性只有在生成子进程的时候才能使用。如果父进程准备创建带有可继承句柄的新内核对象，那么已经在运行的子进程将无法继承这些新句柄。

对象句柄的继承性有一个非常奇怪的特征，那就是当使用它时，子进程不知道它已经继承了任何句柄。只有在另一个进程生成子进程时记录了这样一个情况，即它希望被赋予对内核对象的访问权时，才能使用内核对象句柄的继承权。通常，父应用程序和子应用程序都是由同一个公司编写的，但是，如果另一个公司记录了子应用程序期望的对象，那么该公司也能够编写子应用程序。

子进程为了确定它期望的内核对象的句柄值，最常用的方法是将句柄值作为一个命令行参数传递给子进程，该子进程的初始化代码对命令行进行分析（通常通过调用sscanf函数来进行分析），并取出句柄值。一旦子进程拥有该句柄值，它就具有对该对象的无限访问权。请注意，句柄继承权起作用的唯一原因是，父进程和子进程中的共享内核对象的句柄值是相同的，这就是为什么父进程能够将句柄值作为命令行参数来传递的原因。

China-pub.com
下载

当然，可以使用其他形式的进程间通信，将已继承的内核对象句柄值从父进程传送给子进程。方法之一是让父进程等待子进程完成初始化（使用第9章介绍的WaitForInpuIdle函数），然后，父进程可以将一条消息发送或展示在子进程中的一个线程创建的窗口中。

另一个方法是让父进程将一个环境变量添加给它的环境程序块。该变量的名字是子进程知道要查找的某种信息，而变量的值则是内核对象要继承的值。这样，当父进程生成子进程时，子进程就继承父进程的环境变量，并且能够非常容易地调用 GetEnvironmentVariable函数，以获取被继承对象的句柄值。如果子进程要生成另一个子进程，那么使用这种方法是极好的，因为环境变量可以被再次继承。

3.3.2 改变句柄的标志

有时会遇到这样一种情况，父进程创建一个内核对象，以便检索可继承的句柄，然后生成两个子进程。父进程只想要一个子进程来继承内核对象的句柄。换句话说，有时可能想要控制哪个子进程来继承内核对象的句柄。若要改变内核对象句柄的继承标志，可以调用SetHandleInformation函数：

```
BOOL SetHandleInformation(
   HANDLE hObject,
   DWORD dwMask,
   DWORD dwFlags);
```

可以看到，该函数拥有3个参数。第一个参数hObject用于标识一个有效的句柄。第二个参数dwMask告诉该函数想要改变哪个或那几个标志。目前有两个标志与每个句柄相关联：

```
#define HANDLE_FLAG_INHERIT    0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002
```

如果想同时改变该对象的两个标志，可以逐位用 OR将这些标志连接起来。 SetHandle Information函数的第三个参数是dwFlags，用于指明想将该标志设置成什么值。例如，若要打开一个内核对象句柄的继承标志，请创建下面的代码：

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

若要关闭该标志，请创建下面的代码：

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, 0);
```

HANDLE_FLAG_PROTECT_FROM_CLOSE标志用于告诉系统，该句柄不应该被关闭：

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
   HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hobj);   // Exception is raised
```

如果一个线程试图关闭一个受保护的句柄， CloseHandle就会产生一个异常条件。很少想要将句柄保护起来，使他人无法将它关闭。但是如果一个进程生成了子进程，而子进程又生成了孙进程，那么该标志可能有用。父进程可能希望孙进程继承赋予子进程的对象句柄。不过，子进程有可能在生成孙进程之前关闭该句柄。如果出现这种情况，父进程就无法与孙进程进行通信，因为孙进程没有继承该内核对象。通过将句柄标明为“受保护不能关闭”，那么孙进程就能继承该对象。

但是这种处理方法有一个问题。子进程可以调用下面的代码来关闭 HANDLE_FLAG_PROTECT_FROM_CLOSE标志，然后关闭句柄。

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hobj);
```

China-pub.com 下载

父进程可能打赌说，子进程将不执行该代码。当然，父进程也可能打赌说，子进程将生成孙进程。因此这种打赌没有太大的风险。

为了完整地说明问题，也要讲一下GetHandleInformation函数的情况：

```
BOOL GetHandleInformation(
   HANDLE hObj,
   PDWORD pdwFlags);
```

该函数返回pdwFlags指向的DWORD中特定句柄的当前标志的设置值。若要了解句柄是否是可继承的，请使用下面的代码：

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

3.3.3 命名对象

共享跨越进程边界的内核对象的第二种方法是给对象命名。许多（虽然不是全部）内核对象都是可以命名的。例如，下面的所有函数都可以创建命名的内核对象：

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL bInitialOwner,
   PCTSTR pszName);

HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   BOOL bInitialState,
   PCTSTR pszName);

HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);

HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   PCTSTR pszName);

HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);

HANDLE CreateJobObject(
   PSECURITY_ATTRIBUTES psa,
   PCTSTR pszName);
```

所有这些函数都有一个共同的最后参数pszName。当为该参数传递NULL时，就向系统指明了想创建一个未命名的（匿名）内核对象。当创建一个未命名的对象时，可以通过使用继承

China-pub.com
下载

性（如上一节介绍的那样）或DuplicateHandle(下一节将要介绍)共享跨越进程的对象。若要按名字共享对象，必须为对象赋予一个名字。

如果没有为pszName参数传递MULL，应该传递一个以0结尾的字符串名字的地址。该名字的长度最多可以达到MAX_PATH（定义为260）个字符。但是，Microsoft没有提供为内核对象赋予名字的指导原则。例如，如果试图创建一个称为“ JeffObj ”的对象，那么不能保证系统中不存在一个名字为“ JeffObj ”的对象。更为糟糕的是，所有这些对象都共享单个名空间。由于这个原因，对下面这个CreateSemaphore函数的调用将总是返回NULL：

```
HANDLE hMutex = CreateMutex(NULL, FALSE, "JeffObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "JeffObj");
DWORD dwErrorCode = GetLastError();
```

如果在执行上面的代码后观察 dwErrorcode的值，会看到返回的代码是 6（ERROR_INVALID_HANDLE)。这个错误代码没有很强的表意性，但是你又能够做什么呢？

既然已知道如何给对象命名，那么让我们来看一看如何用这种方法来共享对象。比如说，Process A启动运行，并调用下面的函数：

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, "JeffMutex");
```

调用该函数能创建一个新的互斥内核对象，为它赋予名字“ JeffMutex ”。请注意，Process A的句柄hMutexProcessA不是一个可继承的句柄，并且当你只是命名对象时，它不必是个可继承的句柄。

过些时候，某个进程会生成Process B。Process B不一定是Process A的子进程。它可能是Explorer或其他任何应用程序生成的进程。 Process B不必是Process A的子进程这一事实正是使用命名对象而不是继承性的优越性。当Process B启动运行时，它执行下面的代码：

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, "JeffMutex");
```

当Process B调用CreateMutex时，系统首先要查看是否已经存在一个名字为“ JeffMutex ”的内核对象。由于确实存在一个带有该名字的对象，因此内核要检查对象的类型。由于试图创建一个互斥对象，而名字为“ JeffMutex ”的对象也是个互斥对象，因此系统会执行一次安全检查，以确定调用者是否拥有对该对象的完整的访问权。如果拥有这种访问权，系统就在Process B的句柄表中找出一个空项目，并对该项目进行初始化，使该项目指向现有的内核对象。如果该对象类型不匹配，或者调用者被拒绝访问，那么CreateMutex将运行失败（返回NULL)。

当Process B对CreateMutex的调用取得成功时，它并不实际创建一个互斥对象。相反，Process B只是被赋予一个与进程相关的句柄值，用于标识内核中现有的互斥对象。当然，由于Process B的句柄表中的一个新项目要引用该对象，互斥对象的使用计数就会递增。在Process A和Process B同时关闭它们的对象句柄之前，该对象是不会被撤消的。请注意，这两个进程中的句柄值很可能是不同的值。这是可以的。 Process A将使用它的句柄值，而Process B则使用它自己的句柄值来操作一个互斥内核对象。

注意 当你的多个内核对象拥有相同的名字时，有一个非常重要的细节必须知道。当Process B调用CreateMutex时，它将安全属性信息和第二个参数传递给该函数。如果已经存在带有指定名字的对象，那么这些参数将被忽略。应用程序能够确定它是否确实创建了一个新内核对象，而不是打开了一个现有的对象。方法是在调用 Create*函数后立即调用GetLastError：

```
if (GetLastError() == ERROR_ALREADY_EXISTS) {
   // Opened a handle to an existing object.
```

China-pub.com 下载

```
   // sa.lpSecurityDescriptor and the second parameter
   // (FALSE) are ignored.
} else {
   // Created a brand new object.
   // sa.lpSecurityDescriptor and the second parameter
   // (FALSE) are used to construct the object.
}
```

按名字共享对象的另一种方法是，进程不调用Create*函数，而是调用下面显示的Open*函数中的某一个：

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenEvent(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenJobObject(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

注意，所有这些函数都拥有相同的原型。最后一个参数pszName用于指明内核对象的名字。不能为该参数传递NULL，必须传递以0结尾的地址。这些函数要搜索内核对象的单个名空间，以便找出匹配的空间。如果不存在带有指定名字的内核对象，该函数返回NULL，GetLastError返回2（ERROR_FILE_NOT_FOUND）。但是，如果存在带有指定名字的内核对象，并且它是相同类型的对象，那么系统就要查看是否允许执行所需的访问（通过dwDesiredAccess参数进行访问）。如果拥有该访问权，调用进程的句柄表就被更新，对象的使用计数被递增。如果为bInheritHandle参数传递TRUE，那么返回的句柄将是可继承的。

调用Create*函数与调用Open*函数之间的主要差别是，如果对象并不存在，那么Create*函数将创建该对象，而Open*函数则运行失败。

如前所述，Microsoft没有提供创建唯一对象名的指导原则。换句话说，如果用户试图运行来自不同公司的两个程序，而每个程序都试图创建一个称为“MyObject”的对象，那么这就是个问题。为了保证对象的唯一性，建议创建一个GUID，并将GUID的字符串表达式用作对象名。命名对象常常用来防止运行一个应用程序的多个实例。若要做到这一点，只需要调用main或

China-pub.com
下载

WinMain函数中Create*函数，以便创建一个命名对象（创建的是什么对象则是无所谓的）。当Create*函数返回时，调用GetLastError函数。如果GetLastError函数返回ERROR_ALREADY_EXISTS，那么你的应用程序的另一个实例正在运行，新实例可以退出。下面是说明这种情况的部分代码：

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, PSTR pszCmdLine,
   int nCmdShow) {
   HANDLE h = CreateMutex(NULL, FALSE,
      "{FA531CC1-0497-11d3-A180-00105A276C3E}");
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // There is already an instance of this application running.
      return(0);
   }

   // This is the first instance of this application running.
   :
   :

   // Before exiting, close the object.
   CloseHandle(h);
   return(0);
}
```

3.3.4 终端服务器的名字空间

注意，终端服务器能够稍稍改变上面所说的情况。终端服务器拥有内核对象的多个名字空间。如果存在一个可供内核对象使用的全局名字空间，就意味着它可以供所有的客户程序会话访问。该名字空间主要供服务程序使用。此外，每个客户程序会话都有它自己的名字空间。它能防止运行相同应用程序的两个或多个会话之间出现互相干扰的情况，也就是说一个会话无法访问另一个会话的对象，尽管该对象拥有相同的名字。在没有终端服务器的机器上，服务程序和应用程序拥有上面所说的相同的内核对象名字空间，而在拥有终端服务器的机器上，却不是这样。

服务程序的名字空间对象总是放在全局名字空间中。按照默认设置，在终端服务器中，应用程序的命名内核对象将放入会话的名字空间中。但是，如果像下面这样将“Global\”置于对象名的前面，就可以使命名对象进入全局名字空间：

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Global\\MyName");
```

也可以显式说明想让内核对象进入会话的名字空间，方法是将“Local\”置于对象名的前面：

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Local\\MyName");
```

Microsoft将Global和Local视为保留关键字，除非要强制使用特定的名字空间，否则不应该使用这两个关键字。Microsoft还将Session视为保留关键字，虽然目前它没有任何意义。请注意，所有这些保留关键字是区分大小写字母的。如果主机不运行终端服务器，这些关键字将被忽略。

3.3.5 复制对象句柄

共享跨越进程边界的内核对象的最后一个方法是使用DuplicateHandle函数：

```
BOOL DuplicateHandle(
   HANDLE hSourceProcessHandle,
```

China-pub.com 下载

```
HANDLE hSourceHandle,
HANDLE hTargetProcessHandle,
PHANDLE phTargetHandle,
DWORD dwDesiredAccess,
BOOL bInheritHandle,
DWORD dwOptions);
```

简单说来，该函数取出一个进程的句柄表中的项目，并将该项目拷贝到另一个进程的句柄表中。DuplicateHandle函数配有若干个参数，但是实际上它是非常简单的。DuplicateHandle函数最普通的用法要涉及系统中运行的3个不同进程。

当调用DuplicateHandle函数时，第一和第三个参数hSourceProcessHandle和hTargetProcessHandle是内核对象句柄。这些句柄本身必须与调用DuplicateHandle函数的进程相关。此外，这两个参数必须标识进程的内核对象。如果将句柄传递给任何其他类型的内核对象，那么该函数运行就会失败。第4章将详细介绍进程的内核对象，而现在只需要知道，每当系统中启动一个新进程时都会创建一个进程内核对象。

第二个参数hSourceHandle是任何类型的内核对象的句柄。但是该句柄值与调用DuplicateHandle的进程并无关系。相反，该句柄必须与hSourceProcessHandle句柄标识的进程相关。第四个参数phTargetHandle是HANDLE变量的地址，它将接收获取源进程句柄信息拷贝的项目索引。返回的句柄值与hTargetProcessHandle标识的进程相关。

DuplicateHandle的最后3个参数用于指明该目标进程的内核对象句柄表项目中使用的访问屏蔽值和继承性标志。dwOptions参数可以是0（零），也可以是下面两个标志的任何组合：DUPLICATE_SAME_ACCESS和DUPLICATE_CLOSE_SOURCE。

如果设定了DUPLICATE_SAME_ACCESS标志，则告诉DuplicateHandle函数，你希望目标进程的句柄拥有与源进程句柄相同的访问屏蔽。使用该标志将使DuplicateHandle忽略它的dwDesiredAccess参数。

如果设定了DUPLICATE_CLOSE_SOURCE标志，则可以关闭源进程中的句柄。该标志使得一个进程能够很容易地将内核对象传递给另一个进程。当使用该标志时，内核对象的使用计数不会受到影响。

下面用一个例子来说明DuplicateHandle函数是如何运行的。在这个例子中，Process S是目前可以访问某个内核对象的源进程，Process T是将要获取对该内核对象的访问权的目标进程。Process C是执行对DuplicateHandle调用的催化进程。

Process C的句柄表（表3-4）包含两个句柄值，即1和2。句柄值1用于标识Process S的进程内核对象，句柄值2则用于标识Process T的进程内核对象。

表3-4 Process C的句柄表

索　引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标　志（标志位的DWORD）
1	0xF0000000 (Process S的内核对象)	0x????????	0x00000000
2	0xF0000010 (Process T的内核对象)	0x????????	0x00000000

表3-5是Process S的句柄表，它包含句柄值为2的单个项目。该句柄可以标识任何类型的内核对象，就是说它不必是进程的内核对象。

China-pub.com
下载

表3-5 Process S的句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	(无)	(无)
2	0xF0000020 (任何内核对象)	0x????????	0x00000000

表3-6显示了Process C调用DuplicateHandle函数之前Process T的句柄表包含的项目。如你所见，Process T的句柄表只包含句柄值为2的单个项目，句柄项目1目前未用。

表3-6 调用DuplicateHandle函数之前Process T的句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0x00000000	(无)	(无)
2	0xF0000030 (任何内核对象)	0x????????	0x00000000

如果Process C现在使用下面的代码来调用DuplicateHandle，那么只有Process T的句柄表改变更，如表3-7所示。

表3-7 调用DuplicateHandle函数之后Process T的句柄表

索 引	内核对象内存块的指针	访问屏蔽（标志位的DWORD）	标 志（标志位的DWORD）
1	0xF00000020	0x????????	0x00000001
2	0xF00000030 (任何内核对象)	0x????????	0x00000000

Process S的句柄表中的第二项已经被拷贝到 Process T的句柄表中的第一项。 DuplicateHandle也已经将值1填入Process C的hObj变量中。值1是Process T的句柄表中的索引，新项目将被放入该索引。

由于DUPLICATE_SAME_ACCESS标志被传递给了DuplicateHandle，因此Process T的句柄表中该句柄的访问屏蔽与 Process S的句柄表项目中的访问屏蔽是相同的。另外，传递DUPLICATE_SAME_ACCESS标志将使DuplicateHandle忽略它的DesiredAccess参数。最后请注意，继承位标志已经被打开，因为给DuplicateHandle的bInheritHandle参数传递的是TRUE。

显然，你永远不会像在这个例子中所做的那样，调用传递硬编码数字值的 DuplicateHandle函数。这里使用硬编码数字，只是为了展示函数是如何运行的。在实际应用程序中，变量可能拥有各种不同的句柄值，可以传递该变量，作为函数的参数。

与继承性一样，DuplicateHandle函数存在的奇怪现象之一是，目标进程没有得到关于新内核对象现在可以访问它的通知。因此， Process C必须以某种方式来通知Process T,它现在拥有对内核对象的访问权，并且必须使用某种形式的进程间通信方式，以便将 hObj中的句柄值传递给Process T。显然，使用命令行参数或者改变Process T的环境变量是不行的，因为该进程已经启动运行。因此必须使用窗口消息或某种别的IPC机制。

上面是DuplicateHandle的最普通的用法。如你所见，它是个非常灵活的函数。不过，它很少在涉及3个不同进程的情况下被使用（因为 Process C不可能知道对象的句柄值正在被 Process

China-pub.com 下载

S使用)。通常，当只涉及两个进程时，才调用DuplicateHandle函数。比如一个进程拥有对另一个进程想要访问的对象的访问权，或者一个进程想要将内核对象的访问权赋予另一个进程。例如，Process S拥有对一个内核对象的访问权，并且想要让Process T能够访问该对象。若要做到这一点，可以像下面这样调用DuplicateHandle:

```
// All of the following code is executed by Process S.

// Create a mutex object accessible by Process S.
HANDLE hObjProcessS = CreateMutex(NULL, FALSE, NULL);

// Open a handle to Process T's kernel object.
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
   dwProcessIdT);

HANDLE hObjProcessT;   // An uninitialized handle relative to Process T.

// Give Process T access to our mutex object.
DuplicateHandle(GetCurrentProcess(), hObjProcessS, hProcessT,
   &hObjProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// Use some IPC mechanism to get the handle
// value in hObjProcessS into Process T.
.
.
.

// We no longer need to communicate with Process T.
CloseHandle(hProcessT);
.
.
.

// When Process S no longer needs to use the mutex, it should close it.
CloseHandle(hObjProcessS);
```

在这个例子中，对GetCurrentProcess的调用将返回一个伪句柄，该句柄总是用来标识调用端的进程Process S。一旦DuplicateHandle返回，hObjProcessT就是与Process T相关的句柄，它所标识的对象与引用Process S中的代码时hObjProcessS的句柄标识的对象相同。Process S决不应该执行下面的代码：

```
// Process S should never attempt to close the
// duplicated handle.
CloseHandle(hObjProcessT);
```

如果Process S要执行该代码，那么对代码的调用可能失败，也可能不会失败。如果Process S恰好拥有对内核对象的访问权，其句柄值与hObjProcessT的值相同，那么调用就会成功。该代码的调用将会关闭某个对象，这样Process S就不再拥有对它的访问权，这当然会导致应用程序产生不希望有的行为特性。

下面是使用DuplicateHandle函数的另一种方法。假设一个进程拥有对一个文件映射对象的读和写访问权。在某个位置上，一个函数被调用，它通过读取文件映射对象来访问它。为了使应用程序更加健壮，可以使用DuplicateHandle为现有的对象创建一个新句柄，并确保这个新句柄拥有对该对象的只读访问权。然后将只读句柄传递给该函数，这样，该函数中的代码就永远不会偶然对该文件映射对象执行写入操作。下面这个代码说明了这个例子：

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
   LPSTR szCmdLine, int nCmdShow) {
```

China-pub.com
下载


```
   // Create a file-mapping object; the handle has read/write access.
   HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
      NULL, PAGE_READWRITE, 0, 10240, NULL);

   // Create another handle to the file-mapping object;
   // the handle has read-only access.
   HANDLE hFileMapRO;
   DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
      &hFileMapRO, FILE_MAP_READ, FALSE, 0);

   // Call the function that should only read from the file mapping.
   ReadFromTheFileMapping(hFileMapRO);

   // Close the read-only file-mapping object.
   CloseHandle(hFileMapRO);

   // We can still read/write the file-mapping object using hFileMapRW.
   :
   :

   // When the main code doesn't access the file mapping anymore,
   // close it.
   CloseHandle(hFileMapRW);
}
```

China-pub.com
下载

第二部分　编程的具体方法

第4章　进　　程

本章介绍系统如何管理所有正在运行的应用程序。首先讲述什么是进程，以及系统如何创建进程内核对象，以便管理每个进程。然后将说明如何使用相关的内核对象来对进程进行操作。接着，要介绍进程的各种不同的属性，以及查询和修改这些属性所用的若干个函数。还要讲述创建或生成系统中的辅助进程所用的函数。当然，如果不深入说明如何来结束进程的运行，那么这样的介绍肯定是不完整的。现在就来介绍进程的有关内容。

进程通常被定义为一个正在运行的程序的实例，它由两个部分组成：

- 一个是操作系统用来管理进程的内核对象。内核对象也是系统用来存放关于进程的统计信息的地方。
- 另一个是地址空间，它包含所有可执行模块或 DLL模块的代码和数据。它还包含动态内存分配的空间。如线程堆栈和堆分配空间。

进程是不活泼的。若要使进程完成某项操作，它必须拥有一个在它的环境中运行的线程，该线程负责执行包含在进程的地址空间中的代码。实际上，单个进程可能包含若干个线程，所有这些线程都“同时”执行进程地址空间中的代码。为此，每个线程都有它自己的一组CPU寄存器和它自己的堆栈。每个进程至少拥有一个线程，来执行进程的地址空间中的代码。如果没有线程来执行进程的地址空间中的代码，那么进程就没有存在的理由了，系统就将自动撤消该进程和它的地址空间。

若要使所有这些线程都能运行，操作系统就要为每个线程安排一定的 CPU时间。它通过以一种循环方式为线程提供时间片（称为量程），造成一种假象，仿佛所有线程都是同时运行的一样。图4-1显示了在单个CPU的计算机上是如何实现这种运行方式的。如果计算机拥有多个 CPU，那么操作系统就要使用复杂得多的算法来实现 CPU上线程负载的平衡。

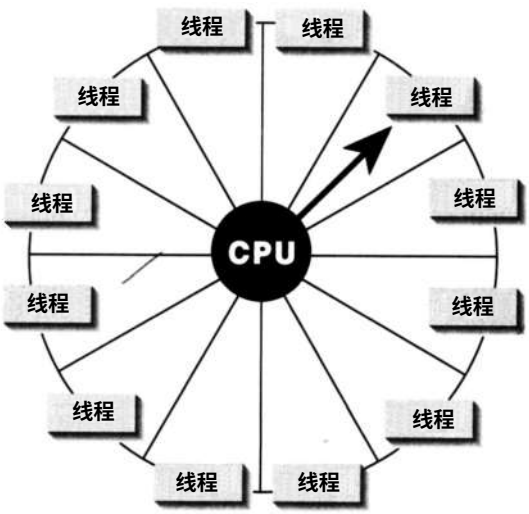


图4-1　操作系统在单个CPU计算机上用循环方式为各个线程提供时间片

当创建一个进程时，系统会自动创建它的第一个线程，称为主线程。然后，该线程可以创建其他的线程，而这些线程又能创建更多的线程。

Windows 2000　Micorsoft Windows 2000能够在拥有多个CPU的计算机上运行。例如，我用来撰写本书的计算机就包含两个处理器。Windows 2000可以在每个CPU上运行不

China-pub.com 下载

同的线程，这样，多个线程就真的在同时运行了。Windows 2000的内核能够在这种类型的系统上进行所有线程的管理和调度。不必在代码中进行任何特定的设置就能利用多处理器提供的各种优点。

Windows 98　Windows 98只能在单处理器计算机上运行。即使计算机配有多个处理器，Windows每次只能安排一个线程运行，而其他的处理器则处于空闲状态。

4.1 编写第一个Windows应用程序

Windows支持两种类型的应用程序。一种是基于图形用户界面（GUI）的应用程序，另一种是基于控制台用户界面（CUI）的应用程序。基于GUI的应用程序有一个图形前端程序。它能创建窗口，拥有菜单，可以通过对话框与用户打交道，并可使用所有的标准“Windows”组件。Windows配备的所有应用程序附件（如Notepad、Calculator和WordPad），几乎都是基于GUI的应用程序。基于控制台的应用程序属于文本操作的应用程序。它们通常不能用于创建窗口或处理消息，并且它们不需要图形用户界面。虽然基于CUI的应用程序包含在屏幕上的窗口中，但是窗口只包含文本。命令外壳程序CMD.EXE（用于Windows 2000）和COMMAND.COM (用于Windows 98)都是典型的基于CUI的应用程序。

这两种类型的应用程序之间的界限是非常模糊的。可以创建用于显示对话框的CUI应用程序。例如，命令外壳程序可能拥有一个特殊的命令，使它能够显示一个图形对话框，在这个对话框中，可以选定你要执行的命令，而不必记住该外壳程序支持的各个不同的命令。也可以创建一个基于GUI的应用程序，它能将文本字符串输出到一个控制台窗口。我常常创建用于建立控制台窗口的GUI应用程序，在这个窗口中，我可以查看应用程序执行时的调试信息。当然你也可以在应用程序中使用图形用户界面，而不是老式的字符界面，因为字符界面使用起来不太方便。

当使用Microsoft Visual C++来创建应用程序时，这种集成式环境安装了许多不同的链接程序开关，这样，链接程序就可以将相应的子系统嵌入产生的可执行程序。用于CUI应用程序的链接程序开关是/SUBSYSTEM:CONDOLE，而用于GUI应用程序的链接程序开关是SUBSYSTEM:WINDOWS。当用户运行一个应用程序时，操作系统的加载程序就会查看可执行图形程序的标题，并抓取该子系统的值。如果该值指明一个CUI应用程序，那么加载程序就会自动保证为该应用程序创建文本控制台窗口。

如果该值指明这是个GUI应用程序，那么加载程序不创建控制台窗口，而只是加载应用程序。一旦应用程序启动运行，操作系统就不再考虑应用程序拥有什么类型的用户界面。

Windows应用程序必须拥有一个在应用程序启动运行时调用的进入点函数。可以使用的进入点函数有4个：

```
int WINAPI WinMain(
   HINSTANCE hinstExe,
   HINSTANCE,
   PSTR pszCmdLine,
   int nCmdShow);

int WINAPI wWinMain(
   HINSTANCE hinstExe,
   HINSTANCE,
   PWSTR pszCmdLine,
```

China-pub.com
下载


```
    int nCmdShow);

int __cdecl main(
    int argc,
    char *argv[],
    char *envp[]);

int __cdecl wmain
    int argc,
    wchar_t *argv[],
    wchar_t *envp[]);
```

操作系统实际上并不调用你编写的进入点函数。它调用的是C/C++运行期启动函数。该函数负责对C/C++运行期库进行初始化，这样，就可以调用malloc和free之类的函数。它还能够确保已经声明的任何全局对象和静态C++对象能够在代码执行以前正确地创建。下面说明源代码中可以实现哪个进入点以及何时使用该进入点(见表4-1)。

表4-1 应用程序的进入点

应用程序类型	进 入 点	嵌入可执行文件的启动函数
需要ANSI字符和字符串的GUI应用程序	WinMain	WinMainCRTStartup
需要Unicode字符和字符串的GUI应用程序	wWinMain	wWinMainCRTStartup
需要ANSI字符和字符串的CUI应用程序	main	mainCRTStartup
需要Unicode字符和字符串的CUI应用程序	wmain	wmainCRTStartup

链接程序负责在它连接可执行文件时选择相应的C/C++运行期启动函数。如果设定了/SUBSYSTEM:WINDOWS链接程序开关，那么该链接程序期望找到一个WinMain或wWinmain函数。如果这两个函数都不存在，链接程序便返回一个“未转换的外部符号”的错误消息。否则，它可以分别选择WinMainCRTStartup函数或wWinMainCRTStartup函数。

同样，如果设定了/SUBSYSTEM:CONSOLE链接程序开关，那么该链接程序便期望找到main或wmain函数，并且可以分别选择mainCRTStartup函数或wmainCRTStartup函数。同样，如果main或wmain都不存在，那么链接程序返回一条“未转换外部符号”的消息。

但是，人们很少知道这样一个情况，即可以从应用程序中全部删除/SUBSYSTEM链接程序开关。当这样做的时候，链接程序能够自动确定应用程序应该连接到哪个子系统。当进行链接时，链接程序要查看代码中存在4个函数（WinMain、wWinMain、main或wmain）中的哪一个。然后确定可执行程序应该是哪一个子系统，并且确定可执行程序中应该嵌入哪个C/C++启动函数。

Windows/Visual C++编程新手常犯的错误之一是，当创建新的应用程序时，不小心选择了错误的应用程序类型。例如，编程员可能创建一个新的Win32应用程序项目，但是创建了一个进入点函数main。当创建应用程序时，编程员会看到一个链接程序错误消息，因为win32应用程序项目设置了/SUBSYSTEM:WINDOWS链接程序开关，但是不存在WinMain或wWinMain函数。这时，编程员可以有4个选择：

- 将main函数改为WinMain。通常这不是最佳的选择，因为编程员可能想要创建一个控制台应用程序。
- 用Visual C++创建一个新的Win32 控制台应用程序，并将现有的源代码添加给新应用程

China-pub.com 下载

序项目。这个选项冗长而乏味，因为它好像是从头开始创建应用程序，而且必须删除原始的应用程序文件。

- 单击Project Settings对话框的Link选项卡，将/SUBSYSTEM:WINDOWS开关改为/SUBSYSTEM:CONSOLE。这是解决问题的一种比较容易的方法，很少有人知道他们只需要进行这项操作就行了。
- 单击Project Settings对话框的Link选项卡，然后全部删除/SUBSYSTEM:WINDOWS开关。这是我喜欢选择的方法，因为它提供了最大的灵活性。现在，连接程序将根据源代码中实现的函数进行正确的操作。当用Visual C++的Developer Studio创建新Win32应用程序或Win32控制台应用程序项目时，我不知道为什么这没有成为默认设置。

所有的C/C++运行期启动函数的作用基本上都是相同的。它们的差别在于，它们究竟是处理ANSI字符串还是Unicode字符串，以及它们在对C运行期库进行初始化后它们调用哪个进入点函数。Visual C++配有C运行期库的源代码。可以在CR t0.c文件中找到这4个启动函数的代码。现在将启动函数的功能归纳如下：

- 检索指向新进程的完整命令行的指针。
- 检索指向新进程的环境变量的指针。
- 对C/C++运行期的全局变量进行初始化。如果包含了StdLib.h文件，代码就能访问这些变量。表4-1列出了这些变量。
- 对C运行期内存单元分配函数（malloc和calloc）和其他低层输入/输出例程使用的内存栈进行初始化。
- 为所有全局和静态C++类对象调用构造函数。

当所有这些初始化操作完成后，C/C++启动函数就调用应用程序的进入点函数。如果编写了一个wWinMain函数，它将以下面的形式被调用：

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = wWinMain(GetModuleHandle(NULL), NULL, pszCommandLineUnicode,
   (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
      ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

如果编写了一个WinMain函数，它将以下面的形式被调用：

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = WinMainGetModuleHandle(NULL), NULL, pszCommandLineAnsi,
   (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
      ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

如果编写了一个wmain函数，它将以下面的形式被调用：

```
int nMainRetVal = wmain(__argc, __wargv, _wenviron);
```

如果编写了一个main函数，它将以下面的形式被调用：

```
int nMainRetVal = main(__argc, __argv, _environ);
```

当进入点函数返回时，启动函数便调用C运行期的exit函数，将返回值（nMainRetVal）传递给它。Exit函数负责下面的操作：

- 调用由_onexit函数的调用而注册的任何函数。
- 为所有全局的和静态的C++类对象调用析构函数。
- 调用操作系统的ExitProcess函数，将nMainRetVal传递给它。这使得该操作系统能够撤消进程并设置它的exit代码。

China-pub.com
下载

表4-2显示了程序能够使用的C/C++运行期全局变量。

表4-2 程序能够使用的C/C++运行期全局变量

变 量 名	类 型	说 明
_osver	unsigned int	操作系统的测试版本。例如，Windows 2000 Beta 3是测试版本2031。因此_osver的值是2031
_winmajor	unsigned int	采用十六进制表示法的Windows主要版本。对于Windows 2000来说，它的值是5
_winminor	unsigned int	采用十六进制表示法的Windows次要版本。对于Windows 2000来说，它的值是0
_winver	unsigned int	(_winmajor<<8)+_winminor在命令行上传递的参数号
--argc	unsigned int	
--argv	char**	带有指向ANSI/Unicode字符串的指针的__argc大小的数组
--wargv	wchar_t**	每个数组项均指向一个命令行参数
_environ	char**	指向ANSI/Unicode字符串的指针的数组。每个数组项指向一个环境字符串
_wenviron	wchar_t**	
_pgmptr	char*	正在运行的程序的ANSI/Unicode全路径和名字
_wpgmptr	wchar_t*	

4.1.1 进程的实例句柄

加载到进程地址空间的每个可执行文件或DLL文件均被赋予一个独一无二的实例句柄。可执行文件的实例作为(w)WinMain的第一个参数hinstExe来传递。对于加载资源的函数调用来说，通常都需要该句柄的值。例如，若要从可执行文件的映象来加载图标资源，需要调用下面这个函数：

```
HICON LoadIcon(
   HINSTANCE hinst,
   PCTSTR pszIcon);
```

LoadIcon的第一个参数用于指明哪个文件（可执行文件或DLL文件）包含你想加载的资源。许多应用程序在全局变量中保存(w)WinMain的hinstExe参数，这样，它就很容易被所有可执行文件的代码访问。

Platform SDK文档中说，有些函数需要HMODULE类型的一个参数。它的例子是下面所示的GetModuleFileName函数：

```
DWORD GetModuleFileName(
   HMODULE hinstModule,
   PTSTR pszPath,
   DWORD cchPath);
```

注意 实际情况说明，HMODULE与HINSTANCE是完全相同的对象。如果函数的文档指明需要一个HMODULE，那么可以传递一个HINSTANCE，反过来，如果需要一个HINSTANCE，也可以传递一个HMODULE。之所以存在两个数据类型，原因是在16位Windows中，HMODULE和HINSTANCE用于标识不同的东西。

(w)WinMain的hinstExe参数的实际值是系统将可执行文件的映象加载到进程的地址空间时使用的基本地址空间。例如，如果系统打开了可执行文件并且将它的内容加载到地址0x00400000中，那么(w)WinMain的hinstExe参数的值就是0x00400000。

可执行文件的映像加载到的基地址是由链接程序决定的。不同的链接程序可以使用不同的

China-pub.com 下载

默认基地址。Visual C++链接程序使用的默认基地址是0x00400000，因为这是在运行Windows 98时可执行文件的映象可以加载到的最低地址。可以改变应用程序加载到的基地址，方法是使用Microsoft的链接程序中的/BASE:address链接程序开关。

如果你想在Windows上加载的可执行文件的基地址小于0x00400000，那么Windows 98加载程序必须将可执行文件重新加载到另一个地址。这会增加加载应用程序所需的时间，不过，这样一来，至少该应用程序能够运行。如果开发的应用程序将要同时在 Windows 98和Windows 2000上运行，应该确保应用程序的基地址是0x00400000或者大于这个地址。

下面的GetModuleHandle函数返回可执行文件或DLL文件加载到进程的地址空间时所用的句柄/基地址：

```
HMODULE GetModuleHandle(PCTSTR pszModule);
```

当调用该函数时，你传递一个以0结尾的字符串，用于设定加载到调用进程的地址空间的可执行文件或 DLL文件的名字。如果系统找到了指定的可执行文件或 DLL文件名，GetModuleHandle便返回该可执行文件或DLL文件映象加载到的基地址。如果系统没有找到该文件，则返回 NULL。也可以调用 GetModuleHandle，为pszModule参数传递 NULL，GetModuleHandle返回调用的可执行文件的基地址。这正是C运行期启动代码调用 (w)WinMain函数时该代码执行的操作。

请记住GetModuleHandle函数的两个重要特性。首先，它只查看调用进程的地址空间。如果调用进程不使用常用的对话框函数，那么调用GetModuleHandle并为它传递“ComDlg32”后，就会返回 NULL，尽管ComDlg32.dll可能加载到了其他进程的地址空间。第二，调用GetModuleHandle并传递NULL值，就会返回进程的地址空间中可执行文件的基地址。因此，即使通过包含在DLL中的代码来调用（NULL），返回的值也是可执行文件的基地址，而不是DLL文件的基地址。

4.1.2 进程的前一个实例句柄

如前所述，C/C++运行期启动代码总是将NULL传递给(w)WinMain的hinstExePrev参数。该参数用在16位Windows中，并且保留了(w)WinMain的一个参数，目的仅仅是为了能够容易地转用16位Windows应用程序。决不应该在代码中引用该参数。由于这个原因，我总是像下面这样编写(w)WinMain函数：

```
int WINAPI WinMain(
   HINSTANCE hinstExe,
   HINSTANCE,
   PSTR pszCmdLine,
   int nCmdShow);
```

由于没有为第二个参数提供参数名，因此编译器不会发出“没有引用参数”的警告。

4.1.3 进程的命令行

当一个新进程创建时，它要传递一个命令行。该命令行几乎永远不会是空的，至少用于创建新进程的可执行文件的名字是命令行上的第一个标记。但是在后面介绍 CreateProcess函数时我们将会看到，进程能够接收由单个字符组成的命令行，即字符串结尾处的零。当 C运行期的启动代码开始运行的时候，它要检索进程的命令行，跳过可执行文件的名字，并将指向命令行其余部分的指针传递给WinMain的pszCmdLine参数。

值得注意的是，pszCmdLine参数总是指向一个ANSI字符串。但是，如果将WinMain改为

China-pub.com
下载

wWinMain，就能够访问进程的Unicode版本命令行。

应用程序可以按照它选择的方法来分析和转换命令行字符串。实际上可以写入pszCmdLine参数指向的内存缓存，但是在任何情况下都不应该写到缓存的外面去。我总是将它视为只读缓存。如果我想修改命令行，首先我要将命令行拷贝到应用程序的本地缓存中，然后再修改本地缓存。

也可以获得一个指向进程的完整命令行的指针，方法是调用GetCommandLine函数：

```
PTSTR GetCommandLine();
```

该函数返回一个指向包含完整命令行的缓存的指针，该命令行包括执行文件的完整路径名。

许多应用程序常常拥有转换成它的各个标记的命令行。使用全局性 __argc（或__wargv）变量，应用程序就能访问命令行的各个组成部分。下面这个函数 CommandLineToArgvW将Unicode字符串分割成它的各个标记：

```
PWSTR CommandLineToArgvW(
   PWSTR pszCmdLine,
   int* pNumArgs);
```

正如该函数名的结尾处的W所暗示的那样，该函数只存在于Unicode版本中（W是英文单词‘Wide’的缩写）。第一个参数pszCmdLine指向一个命令行字符串。这通常是较早时调用GetCommandLineW而返回的值。PNumArgs参数是个整数地址，该整数被设置为命令行中的参数的数目。CommandLineToArgvW将地址返回给一个Unicode字符串指针的数组。

CommaneLineToArgvW负责在内部分配内存。大多数应用程序不释放该内存，它们在进程运行终止时依靠操作系统来释放内存。这是完全可行的。但是如果想要自己来释放内存，正确的方法是像下面这样调用HeapFree函数：

```
int nNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), &nNumArgs);

// Use the arguments…
if (*ppArgv[1] == L'x') {
   :
   :

}
// Free the memory block
HeapFree(GetProcessHeap(), 0, ppArgv);
```

4.1.4 进程的环境变量

每个进程都有一个与它相关的环境块。环境块是进程的地址空间中分配的一个内存块。每个环境块都包含一组字符串，其形式如下：

```
VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0
:
:

VarNameX=VarValueX\0
\0
```

每个字符串的第一部分是环境变量的名字，后跟一个等号，等号后面是要赋予变量的值。

China-pub.com 下载

环境块中的所有字符串都必须按环境变量名的字母顺序进行排序。

由于等号用于将变量名与变量的值分开，因此等号不能是变量名的一部分。另外，变量中的空格是有意义的。例如，如果声明下面两个变量，然后将XYZ的值与ABC的值进行比较，那么系统将报告称，这两个变量是不同的，因为紧靠着等号的前面或后面的任何空格均作为比较时的条件被考虑在内。

```
XYZ= Windows  (Notice the space after the equal sign.)
ABC=Windows
```

例如，如果将下面两个字符串添加给环境块，后面带有空格的环境变量 XYZ包含Home，而没有空格的环境变量XYZ则包含Work。

```
XYZ =Home  (Notice the space before the equal sign.)
XYZ=Work
```

最后，必须将一个0字符置于所有环境变量的结尾处，以表示环境块的结束。

Windows98 若要为 Windows 98创建一组初始环境变量，必须修改系统的AutoExec.bat文件，将一系列SET行放入该文件。每个SET行都必须采用下面的形式：

```
SET VarName=VarValue
```

当重新引导系统时， AutoExec.bat文件的内容被分析，设置的任何环境变量均可供在Windows 98会话期间启动的任何进程使用。

Windows 2000 当用户登录到Windows 2000中时，系统创建一个外壳进程并将一组环境字符串与它相关联。通过查看注册表中的两个关键字，系统可以获得一组初始环境字符串。

第一个关键字包含一个适用于系统的所有环境变量的列表：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
   Session Manager\Environment
```

第二个关键字包含适用于当前登录的用户的所有环境变量的列表：

```
HKEY_CURRENT_USER\Environment
```

用户可以对这些项目进行增加、删除或修改，方法是选定控制面板的System小应用程序，单击Advanced选项卡，再单击Environment Variables按钮，打开图4-2所示的对话框：

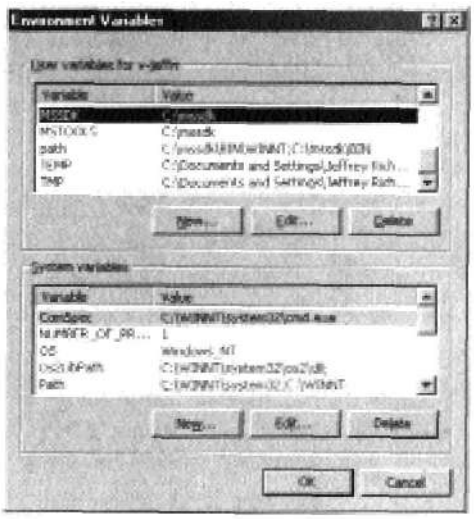


图4-2 使用Environment Variables 对话框修改变量

China-pub.com
下载

只有拥有管理员权限的用户才能修改系统变量列表中的变量。

应用程序也可以使用各种注册表函数来修改这些注册表项目。但是，若要使这些修改在所有应用程序中生效，用户必须退出系统，然后再次登录。有些应用程序，如Explorer、Task Manager和Control Panel等，在它们的主窗口收到WM_SETTINGCHANGE消息时，用新注册表项目来更新它们的环境块。例如，如果要更新注册表项目，并且想让有关的应用程序更新它们的环境块，可以调用下面的代码：

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE,
   0, (LPARAM) TEXT("Environment"));
```

通常，子进程可以继承一组与父进程相同的环境变量。但是，父进程能够控制子进程继承什么环境变量，后面介绍CreateProcess函数时就会看到这个情况。所谓继承，指的是子进程获得它自己的父进程的环境块拷贝，子进程与父进程并不共享相同的环境块。这意味着子进程能够添加、删除或修改它的环境块中的变量，而这个变化在父进程的环境块中却得不到反映。

应用程序通常使用环境变量来使用户能够调整它的行为特性。用户创建一个环境变量并对它进行初始化。然后，当用户启动应用程序运行时，该应用程序要查看环境块，找出该变量。如果找到了变量，它就分析变量的值，调整自己的行为特性。

环境变量存在的问题是，用户难以设置或理解这些变量。用户必须正确地拼写变量的名字，而且必须知道变量值期望的准确句法。另一方面，大多数图形应用程序允许用户使用对话框来调整应用程序的行为特性。这种方法对用户来说更加友好。

如果仍然想要使用环境变量，那么有几个函数可供应用程序调用。使用GetEnvironment Variable函数，就能够确定某个环境变量是否存在以及它的值：

```
DWORD GetEnvironmentVariable(
   PCTSTR pszName,
   PTSTR pszValue,
   DWORD cchValue);
```

当调用GetEnvironmentVariable时，pszName指向需要的变量名，pszValue指向用于存放变量值的缓存，cchValue用于指明缓存的大小（用字符数来表示）。该函数可以返回拷贝到缓存的字符数，如果在环境中找不到该变量名，也可以返回0。

许多字符串包含了里面可取代的字符串。例如，我在注册表中的某个地方找到了下面的字符串：

```
%USERPROFILE%\My Documents
```

百分数符号之间的部分表示一个可取代的字符串。在这个例子中，环境变量的值USERPROFILE应该被放入该字符串中。在我的计算机中，我的USERPROFILE环境变量的值是：

```
C:\Documents and Settings\Administrator
```

因此，在执行字符串替换后，产生的字符串就成为：

```
C:\Documents and Settings\Administrator\My Documents
```

由于这种类型的字符串替换是很常用的，因此Windows提供了ExpandEnvironmentStrings函数：

```
DWORD ExpandEnvironmentStrings(
   PCSTR pszSrc,
   PSTR pszDst,
   DWORD nSize);
```

当调用该函数时，pszSrc参数是包含可替换的环境变量字符串的这个字符串的地址。pszDst

China-pub.com 下载

参数是接收已展开字符串的缓存的地址，nSize参数是该缓存的最大值（用字符数来表示）。

最后，可以使用SetEnvironmentVariable函数来添加变量、删除变量或者修改变量的值：

```
BOOL SetEnvironmentVariable(
   PCTSTR pszName,
   PCTSTR pszValue);
```

该函数用于将pszName参数标识的变量设置为pszValue参数标识的值。如果带有指定名字的变量已经存在，SetEnvironmentVariable就修改该值。如果指定的变量不存在，便添加该变量，如果pszValue是NULL，便从环境块中删除该变量。

应该始终使用这些函数来操作进程的环境块。前面讲过，环境块中的字符串必须按变量名的字母顺序来存放，这样，SetEnvironmentVariable就会很容易地找到它们。SetEnvironmentVariable函数具有足够的智能，使环境变量保持有序排列。

4.1.5 进程的亲缘性

一般来说，进程中的线程可以在主计算机中的任何一个CPU上执行。但是一个进程的线程可能被强制在可用CPU的子集上运行。这称为进程的亲缘性，将在第7章详细介绍。子进程继承了父进程的亲缘性。

4.1.6 进程的错误模式

与每个进程相关联的是一组标志，用于告诉系统，进程对严重的错误应该如何作出反映，这包括磁盘介质故障、未处理的异常情况、文件查找失败和数据没有对齐等。进程可以告诉系统如何处理每一种错误。方法是调用SetErrorMode函数：

```
UINT SetErrorMode(UINT fuErrorMode);
```

fuErrorMode参数是表4-3的任何标志按位用OR连接在一起的组合。

表4-3 fuError Mode 参数的标志

标 志	说 明
SEM_FAILCRITICALERRORS	系统不显示关键错误句柄消息框，并将错误返回给调用进程
SEM_NOGOFAULTERRORBOX	系统不显示一般保护故障消息框。本标志只应该由采用异常情况处理程序来处理一般保护（GP）故障的调试应用程序来设定
SEM_NOOPENFILEERRORBOX	当系统找不到文件时，它不显示消息框。
SEM_NOALIGNMENTFAULTEXCEPT	系统自动排除内存没有对齐的故障，并使应用程序看不到这些故障。本标志对x86处理器不起作用

默认情况下，子进程继承父进程的错误模式标志。换句话说，如果一个进程的SEM_NOGPFAULTERRORBOX标志已经打开，并且生成了一个子进程，该子进程也拥有这个打开的标志。但是，子进程并没有得到这一情况的通知，它可能尚未编写以便处理GP故障的错误。如果GP故障发生在子进程的某个线程中，该子进程就会终止运行，而不通知用户。父进程可以防止子进程继承它的错误模式，方法是在调用CreateProcess时设定CREATE_DEFAULT_ERROR_MODE标志（本章后面部分的内容将要介绍CreateProcess函数）。

4.1.7 进程的当前驱动器和目录

当不提供全路径名时，Windows的各个函数就会在当前驱动器的当前目录中查找文件和目

China-pub.com
下载

录。例如，如果进程中的一个线程调用CreateFile来打开一个文件（不设定全路径名），那么系统就会在当前驱动器和目录中查找该文件。

系统将在内部保持对进程的当前驱动器和目录的跟踪。由于该信息是按每个进程来维护的，因此改变当前驱动器或目录的进程中的线程，就可以为该进程中的所有线程改变这些信息。

通过调用下面两个函数，线程能够获得和设置它的进程的当前驱动器和目录：

```
DWORD GetCurrentDirectory(
   DWORD cchCurDir,
   PTSTR pszCurDir);
BOOL SetCurrentDirectory(PCTSTR pszCurDir);
```

4.1.8 进程的当前目录

系统将对进程的当前驱动器和目录保持跟踪，但是它不跟踪每个驱动器的当前目录。不过，有些操作系统支持对多个驱动器的当前目录的处理。这种支持是通过进程的环境字符串来提供的。例如，进程能够拥有下面所示的两个环境变量：

```
=C:=C:\Utility\Bin
=D:=D:\Program Files
```

这些变量表示驱动器C的进程的当前目录是\Utility\Bin，并且指明驱动器D的进程的当前目录是\Program Files。

如果调用一个函数，传递一个驱动器全限定名，以表示一个驱动器不是当前驱动器，那么系统就会查看进程的环境块，找出与指定驱动器名相关的变量。如果该驱动器的变量存在，系统将该变量的值用作当前驱动器。如果该变量不存在，系统将假设指定驱动器的当前目录是它的根目录。

例如，如果进程的当前目录是C:\Utility|Bin，并且你调用CreateFile来打开D:ReadMe.Txt，那么系统查看环境变量=D。因为=D变量存在，因此系统试图从D:\Program Files目录打开该ReadMe.Txt文件。如果=D变量不存在，系统将试图从驱动器D的根目录来打开ReadMe.Txt。Windows的文件函数决不会添加或修改驱动器名的环境变量，它们只是读取这些变量。

注意 可以使用C运行期函数_chdir，而不是使用Windows的SetCurrentDirectory函数来变更当前目录。_chdir函数从内部调用SetCurrentDirectory，但是_chdir也能够添加或修改该环境变量，这样，不同驱动器的当前目录就可以保留。

如果父进程创建了一个它想传递给子进程的环境块，子进程的环境块不会自动继承父进程的当前目录。相反，子进程的当前目录将默认为每个驱动器的根目录。如果想要让子进程继承父进程的当前目录，该父进程必须创建这些驱动器名的环境变量。并在生成子进程前将它们添加给环境块。通过调用GetFullPathName，父进程可以获得它的当前目录：

```
DWORD GetFullPathName(
   PCTSTR pszFile,
   DWORD cchPath,
   PTSTR pszPath,
   PTSTR *ppszFilePart);
```

例如，若要获得驱动器C的当前目录，可以像下面这样调用GetFullPathName：

```
TCHAR szCurDir[MAX_PATH];
DWORD GetFullPathName(TEXT("C:"), MAX_PATH, szCurDir, NULL);
```

记住，进程的环境变量必须始终按字母顺序来排序。因此驱动器名的环境变量通常必须置

China-pub.com 下载

于环境块的开始处。

4.1.9 系统版本

应用程序常常需要确定用户运行的是哪个Windows版本。例如，通过调用安全性函数，应用程序就能利用它的安全特性。但是这些函数只有在Windows 2000上才能得到全面的实现。

Windows API拥有下面的GetVersion函数：

```
DWORD GetVersion();
```

该函数已经有相当长的历史了。最初它是为16位Windows设计的。它的作用很简单，在高位字中返回MS-DOS版本号，在低位字中返回Windows版本号。对于每个字来说，高位字节代表主要版本号，低位字节代表次要版本号。

但是，编写该代码的程序员犯了一个小小的错误，函数的编码结果使得Windows的版本号颠倒了，即主要版本号位于低位字节，而次要版本号位于高位字节。由于许多程序员已经开始使用该函数，Microsoft不得不保持函数的原样，并修改了文档，以说明这个错误。

由于围绕着GetVersion函数存在着各种混乱，因此Microsoft增加了一个新函数GetVersionEx:

```
BOOL GetVersionEx(POSVERSIONINFO pVersionInformation);
```

该函数要求在应用程序中指定一个OSVERSIONINFOEX结构，并将该结构的地址传递给GetVersionEx。OSVERSIONINFOEX的结构如下所示：

```
typedef struct {
   DWORD dwOSVersionInfoSize;
   DWORD dwMajorVersion;
   DWORD dwMinorVersion;
   DWORD dwBuildNumber;
   DWORD dwPlatformId;
   TCHAR szCSDVersion[128];
   WORD  wServicePackMajor;
   WORD  wServicePackMinor;
   WORD  wSuiteMask;
   BYTE  wProductType;
   BYTE  wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX;
```

OSVERSIONINFOEX结构在Windows 2000中是个新结构。Windows的其他版本使用较老的OSVERSIONINFO结构,它没有服务程序包、程序组屏蔽、产品类型和保留成员。

注意，对于系统的版本号中的每个成分来说，该结构拥有不同的成员。这样做的目的是，程序员不必提取低位字、高位字、低位字节和高位字节，因此应用程序能够更加容易地对它们期望的版本号与主机系统的版本号进行比较。表4-4描述了OSVERSIONINFOEX结构的成员。

表4-4 OSVERSIONINFOEX结构的成员

成 员	描 述
dwOSVersionInfoSize	在调用GetVersionEx函数之前,必须置为sizeof(OSVERSIONINFO)或sizeof(OSVERSIONINFOEX)。
dwMajorVersion	主系统的主要版本号
dwMinorVersion	主系统的次要版本号
dwBuildNumber	当前系统的构建号

China-pub.com
下载

(续)

成 员	描 述
dw Platform Id	用于标识当前系统支持的平台。它可以是 VER_PLATFORM_WIN32（Win32），VER_PLATFORM_WIN32_WINDOWS（Windows 95/Windows 98），VER_PLATFORM_WIN32_NT（Windows NT/Windows 2000）或VER_PLATFORM_WIN32_CEHH（Windows CE）
szCSDVersion	本域包含了附加文本，用于提供关于已经安装的操作系统的详细信息
wServicePackMajor	最新安装的服务程序包的主要版本号
wServicePackMinor	最新安装的服务程序包的次要版本号
wSuiteMask	用于标识系统上存在哪个程序组（ VER_SUITE_SMALLBUSINESS, VER_SUITE_ENTERPRISE,VER_SUITE_BACKOFFICE,VER_SUITE_COMMUNICATIONS,VER_SUITE_TERMINAL,VER_SUITE_ SMALLBUSINESS_RESTRICTED, VER_SUITE_EMBEDDEDNT和VER_SUITE_ DATACENTER）
wProductType	用于标识安装了下面的哪个操作系统： VER_NT_WORKSTATION, VER_NT_SERVER或VER_NT_DOMAIN_CONTROLLER
wReserved	留作将来使用

为了使操作更加容易，Windows 2000提供了一个新的函数，即VerifyVersionInfo，用于对主机系统的版本与你的应用程序需要的版本进行比较：

```
BOOL VerifyVersionInfo(
   POSVERSIONINFOEX pVersionInformation,
   DWORD dwTypeMask,
   DWORDLONG dwlConditionMask);
```

若要使用该函数，必须指定一个OSVERSIONINFOEX结构，将它的dwOSVersionInfoSize成员初始化为该结构的大小，然后对该结构中的其他成员（这些成员对你的应用程序来说很重要）进行初始化。当调用VerifyVersionInfo时，dwTypeMask参数用于指明该结构的哪些成员已经进行了初始化。 dwTypeMask参数是用 OR连接在一起的下列标志中的任何一个标志：VER_MINORVERSION，VER_MAJORVERSION，VER_BUILDNUMBER，VER_PLATFORMID，VER_ SERVICEPACKMINOR， VER_SERVICEPACKMAJOR， VER_SUITENAME，VER_PRODUCT_ TYPE。最后一个参数dwlConditionMask是个64位值，用于控制该函数如何将系统的版本信息与需要的信息进行比较。

dwlConditionMask描述了如何使用一组复杂的位组合进行的比较。若要创建需要的位组合，可以使用VER_SET_CONDITION宏：

```
VER_SET_CONDITION(
   DWORDLONG dwlConditionMask,
   ULONG dwTypeBitMask,
   ULONG dwConditionMask)
```

第一个参数dwlConditionMask用于标识一个变量，该变量的位是要操作的那些位。请注意，不必传递该变量的地址，因为 VER_SET_CONDITION是个宏，不是一个函数。dwTypeBitMask参数用于指明想要比较的OSVERSIONINFOEX结构中的单个成员。若要比较多个成员，必须多次调用 VER_SET_CONDITION宏，每个成员都要调用一次。传递给VerifyVersionInfo的dwTypeMask参数（VER_MINORVERSION，VER_BUILDNUMBER等）的标志与用于VER_SET_CONDITION的dwTypeBitMask参数的标志是相同的。

VER_SET_CONDITION的最后一个参数dwConditionMask用于指明想如何进行比较。它可以是下列值之一：VER_EQUAL，VER_GREATER，VER_GREATER_EQUAL，VER_LESS或VER_LESS_EQUAL。请注意，当比较VER_PRODUCT_TYPE信息时，可以使用这些值。例如，

China-pub.com 下载

VER_NT_WORKSTATION小于VER_NT_SERVER。但是对于VER_SUITENAME信息来说，不能使用这些测试值。相反，必须使用 VER_AND（所有程序组都必须安装）或 VER_OR（至少必须安装程序组产品中的一个产品）。

当建立一组条件后，可以调用 VerifyVersionInfo函数，如果调用成功（如果主机系统符合应用程序的所有要求），则返回非零值。如果 VerifyVersionInfo返回0，那么主机系统不符合要求，或者表示对该函数的调用不正确。通过调用 GetLastError函数，就能确定该函数为什么返回0。如果GetLastError返回ERROR_OLD_WIN_VERSION，那么对该函数的调用是正确的，但是系统没有满足要求。

下面是如何测试主机系统是否正是Windows 2000的一个例子：

```
// Prepare the OSVERSIONINFOEX structure to indicate Windows 2000.
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionInfoSize = sizeof(osver);
osver.dwMajorVersion = 5;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATFORM_WIN32_NT;

// Prepare the condition mask.
DWORDLONG dwlConditionMask = 0;// You MUST initialize this to 0.
VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

// Perform the version test.
if (VerifyVersionInfo(&osver, VER_MAJORVERSION  | VER_MINORVERSION | VER_PLATFORMID,
   dwlConditionMask)) {
   // The host system is Windows 2000 exactly.
} else {
   // The host system is NOT Windows 2000.
}
```

4.2 CreateProcess函数

可以用CreateProcess函数创建一个进程：

```
BOOL CreateProcess(
   PCTSTR pszApplicationName,
   PTSTR pszCommandLine,
   PSECURITY_ATTRIBUTES psaProcess,
   PSECURITY_ATTRIBUTES psaThread,
   BOOL bInheritHandles,
   DWORD fdwCreate,
   PVOID pvEnvironment,
   PCTSTR pszCurDir,
   PSTARTUPINFO psiStartInfo,
   PPROCESS_INFORMATION ppiProcInfo);
```

当一个线程调用CreateProcess时，系统就会创建一个进程内核对象，其初始使用计数是 1。该进程内核对象不是进程本身，而是操作系统管理进程时使用的一个较小的数据结构。可以将进程内核对象视为由进程的统计信息组成的一个较小的数据结构。然后，系统为新进程创建一个虚拟地址空间，并将可执行文件或任何必要的 DLL文件的代码和数据加载到该进程的地址空间中。

China-pub.com
下载

然后，系统为新进程的主线程创建一个线程内核对象（其使用计数为 1）。与进程内核对象一样，线程内核对象也是操作系统用来管理线程的小型数据结构。通过执行 C/C++运行期启动代码，该主线程便开始运行，它最终调用 WinMain、wWinMain、main或wmain函数。如果系统成功地创建了新进程和主线程，CreateProcess便返回TRUE。

注意 在进程被完全初始化之前，CreateProcess返回TRUE。这意味着操作系统加载程序尚未试图找出所有需要的 DLL。如果一个DLL无法找到，或者未能正确地初始化，那么该进程就终止运行。由于 CreateProcess返回TRUE，因此父进程不知道出现的任何初始化问题。

这就是总的概述。下面各节将分别介绍CreateProcess的各个参数。

4.2.1 pszApplicationName和pszCommandLine

pszApplicationName和pszCommandLine参数分别用于设定新进程将要使用的可执行文件的名字和传递给新进程的命令行字符串。下面首先让我们谈一谈 pszCommandLine参数。

注意 请注意，pszCommandLine参数的原型是PTSTR。这意味着CreateProcess期望你将传递一个非常量字符串的地址。从内部来讲，CreateProcess实际上并不修改你传递给它的命令行字符串。不过，在 CreateProcess返回之前，它将该字符串恢复为它的原始形式。

这个问题很重要，因为如果命令行字符串不包含在文件映象的只读部分中，就会出现违规访问的问题。例如，下面的代码就会导致违规访问的问题，因为 Visual C++将“ NOTEPAD ”字符串放入了只读内存：

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

当CreateProcess试图修改该字符串时，就会发生违规访问（较早的 Visual C++版本将该字符串放入读/写内存，因此调用CreateProcess不会导致违规访问的问题）。

解决这个问题的最好办法是在调用 CreateProcess之前像下面这样将常量字符串拷贝到临时缓存中：

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPAD");
CreateProcess(NULL, szCommandLine, NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

也可以考虑使用Visual C++的/Gf和/GF编译器开关，这些开关用于控制重复字符串的删除和确定这些字符串是否被放入只读内存部分（另外请注意，/ZI开关允许使用Visual Studio的Edit &Continue调试特性，它包含了/GF开关的功能）。能做的最好工作是使用/GF编译器开关和临时缓存。Microsoft能做的最好事情是安装好Create-Process，使它能够制作一个该字符串的临时拷贝，这样我们就不必进行这项操作。也许将来的Windows版本能够做到这一点。

另外，如果调用Windows 2000上的CreateProcess的ANSI版本，就不会违规访问，因为系统已经制作了一个命令行字符串的临时拷贝（详细信息请见第 2章）。

China-pub.com 下载

可以使用pszCommandLine参数设定一个完整的命令行，以便CreateProcess用来创建新进程。当CreateProcess分析pszCommandLine字符串时，它将查看字符串中的第一个标记，并假设该标记是想运行的可执行文件的名字。如果可执行文件的文件名没有扩展名，便假设它的扩展名为.exe。CreateProcess也按下面的顺序搜索该可执行文件：

1) 包含调用进程的.exe文件的目录。
2) 调用进程的当前目录。
3) Windows的系统目录。
4) Windows目录。
5) PATH环境变量中列出的目录。

当然，如果文件名包含全路径，系统将使用全路径来查看可执行文件，并且不再搜索这些目录。如果系统找到了可执行文件，那么它就创建一个新进程，并将可执行文件的代码和数据映射到新进程的地址空间中。然后系统将调用C/C++运行期启动例程。正如前面我们讲过的那样，C/C++运行期启动例程要查看进程的命令行，并将地址作为(w)WinMain的pszCmdLine参数传递给可执行文件的名字后面的第一个参数。

这一切都是在pszApplicationName参数是NULL（99%以上的时候都应该属于这种情况）时发生的。如果不传递NULL，可以将地址传递给pszApplicationName参数中包含想运行的可执行文件的名字的字符串。请注意，必须设定文件的扩展名，系统将不会自动假设文件名有一个.exe扩展名。CreateProcess假设该文件位于当前目录中，除非文件名前面有一个路径。如果在当前目录中找不到该文件，CreateProcess将不会在任何其他目录中查找该文件，它运行失败了。

但是，即使在pszApplicationName参数中设定了文件名，CreateProcess也会将pszCommandLine参数的内容作为它的命令行传递给新进程。例如，可以像下面这样调用CreateProcess:

```
// Make sure that the path is in a read/write section of memory.
TCHAR szPath[] = TEXT("WORDPAD README.TXT");

// Spawn the new process.
CreateProcess(TEXT("C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE"),szPath,...);
```

系统启动Notepad应用程序，但是Notepad的命令行是WORDPAD README.TXT。这种变异情况当然有些奇怪，不过这正是CreateProcess运行的样子。这个由pszApplicationName参数提供的能力实际上被添加给了CreateProcess，以支持Windows 2000的POSIX子系统。

4.2.2 psaProcess、psaThread和binheritHandles

若要创建一个新进程，系统必须创建一个进程内核对象和一个线程内核对象（用于进程的主线程），由于这些都是内核对象，因此父进程可以得到机会将安全属性与这两个对象关联起来。可以使用psaProcess和psaThread参数分别设定进程对象和线程对象需要的安全性。可以为这些参数传递NULL，在这种情况下，系统为这些对象赋予默认安全性描述符。也可以指定两个SECURITY_ATTRIBUTES结构，并对它们进行初始化，以便创建自己的安全性权限，并将它们赋予进程对象和线程对象。

将SECURITY_ATTRIBUTES结构用于psaProcess和psaThread参数的另一个原因是，父进程将来生成的任何子进程都可以继承这两个对象句柄中的任何一个（第3章已经介绍了内核对象句柄的继承性的有关理论）。

清单4-1显示了一个说明内核对象继承性的简单程序。假设Process A创建了Process B，方

China-pub.com
下载

法是调用CreateProcess，为psaProcess参数传递一个SECURITY_ATTRIBUTES结构的地址，在这个结构中，bInheritHandles成员被置为TRUE。在同样这个函数调用中，psaThread参数指向另一个SECURITY_ATTRIBUTES结构，在这个结构中，bInheritHandles成员被置为FALSE。

当系统创建Process B时，它同时指定一个进程内核对象和一个线程内核对象，并且将句柄返回给ppiProcInfo参数（很快将介绍该参数）指向的结构中的Process A。这时，使用这些句柄，Process A就能够对新创建的进程对象和线程对象进行操作。

现在，假设Process A第二次调用CreateProcess函数，以便创建Process C。Process A可以决定是否为Process C赋予对Process A能够访问的某些内核对象进行操作的能力。BInheritHandles参数可以用于这个目的。如果bInheritHandles被置为TRUE，系统就使Process C继承Process A中的任何可继承句柄。在这种情况下， Process B的进程对象的句柄是可继承的。无论CreateProcess的bInheritHandles参数的值是什么，Process B的主线程对象的句柄均不能继承。同样，如果Process A调用CreateProcess，为bInheritHandles传递FALSE，那么Process C将不能继承Process A目前使用的任何句柄。

清单4-1 内核对象句柄继承性的一个示例

Inherit.c

```
/******************************************************************
Module name: Inherit.c
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************/


#include <Windows.h>


int WINAPI WinMain (HINSTANCE hinstExe, HINSTANCE,
   PSTR pszCmdLine, int nCmdShow) {

   // Prepare a STARTUPINFO structure for spawning processes.
   STARTUPINFO si = { sizeof(si) };
   SECURITY_ATTRIBUTES saProcess, saThread;
   PROCESS_INFORMATION piProcessB, piProcessC;
   TCHAR szPath[MAX_PATH];

   // Prepare to spawn Process B from Process A.
   // The handle identifying the new process
   // object should be inheritable.
   saProcess.nLength = sizeof(saProcess);
   saProcess.lpSecurityDescriptor = NULL;
   saProcess.bInheritHandle = TRUE;

   // The handle identifying the new thread
   // object should NOT be inheritable.
   saThread.nLength = sizeof(saThread);
   saThread.lpSecurityDescriptor = NULL;
   saThread.bInheritHandle = FALSE;

   // Spawn Process B.
   lstrcpy(szPath, TEXT("ProcessB"));
   CreateProcess(NULL, szPath, &saProcess, &saThread,
```

China-pub.com 下载

```
      FALSE, 0, NULL, NULL, &si, &piProcessB);

   // The pi structure contains two handles
   // relative to Process A:
   // hProcess, which identifies Process B's process
   // object and is inheritable; and hThread, which identifies
   // Process B's primary thread object and is NOT inheritable.
   // Prepare to spawn Process C from Process A.
   // Since NULL is passed for the psaProcess and psaThread
   // parameters, the handles to Process C's process and
   // primary thread objects default to "noninheritable."

   // If Process A were to spawn another process, this new
   // process would NOT inherit handles to Process C's process
   // and thread objects.

   // Because TRUE is passed for the bInheritHandles parameter,
   // Process C will inherit the handle that identifies Process
   // B's process object but will not inherit a handle to
   // Process B's primary thread object.
   lstrcpy(szPath, TEXT("ProcessC"));
   CreateProcess(NULL, szPath, NULL, NULL,
      TRUE, 0, NULL, NULL, &si, &piProcessC);

   return(0);
}
```

4.2.3 fdwCreate

fdwCreate参数用于标识标志，以便用于规定如何来创建新进程。如果将标志逐位用 OR操作符组合起来的话，就可以设定多个标志。

- EBUG_PROCESS标志用于告诉系统，父进程想要调试子进程和子进程将来生成的任何进程。本标志还告诉系统，当任何子进程（被调试进程）中发生某些事件时，将情况通知父进程（这时是调试程序）。
- DEBUG_ONLY_THIS_PROCESS标志与DEBUG_PROCESS标志相类似，差别在于，调试程序只被告知紧靠父进程的子进程中发生的特定事件。如果子进程生成了别的进程，那么将不通知调试程序在这些别的进程中发生的事件。
- CREATE_SUSPENDED标志可导致新进程被创建，但是，它的主线程则被挂起。这使得父进程能够修改子进程的地址空间中的内存，改变子进程的主线程的优先级，或者在进程有机会执行任何代码之前将进程添加给一个作业。一旦父进程修改了子进程，父进程将允许子进程通过调用ResumeThread函数来执行代码（第7章将作详细介绍）。
- DETACHED_PROCESS标志用于阻止基于CUI的进程对它的父进程的控制台窗口的访问，并告诉系统将它的输出发送到新的控制台窗口。如果基于 CUI的进程是由另一个基于CUI的进程创建的，那么按照默认设置，新进程将使用父进程的控制台窗口（当通过命令外壳程序来运行C编译器时，新控制台窗口并不创建，它的输出将被附加在现有控制台窗口的底部）。通过设定本标志，新进程将把它的输出发送到一个新控制台窗口。
- CREATE_NEW_CONSOLE标志负责告诉系统，为新进程创建一个新控制台窗口。如果同时设定CREATE_NEW_CONSOLE和DETACHED_PROCESS标志，就会产生一个错误。

China-pub.com
下载

- CREATE_NO_WINDOW标志用于告诉系统不要为应用程序创建任何控制台窗口。可以使用本标志运行一个没有用户界面的控制台应用程序。
- CREATE_NEW_PROCESS_GROUP标志用于修改用户在按下Ctrl+C或Ctrl+Break键时得到通知的进程列表。如果在用户按下其中的一个组合键时，你拥有若干个正在运行的CUI进程，那么系统将通知进程组中的所有进程说，用户想要终止当前的操作。当创建一个新的CUI进程时，如果设定本标志，可以创建一个新进程组。如果该进程组中的一个进程处于活动状态时用户按下Ctrl+C或Ctrl_Break键，那么系统只通知用户需要这个进程组中的进程。
- CREATE_DEFAULT_ERROR_MODE标志用于告诉系统，新进程不应该继承父进程使用的错误模式（参见本章前面部分中介绍的SetErrorMode函数）。
- CREATE_SEPARATE_WOW_VDM标志只能当你在Windows 2000上运行16位Windows应用程序时使用。它告诉系统创建一个单独的DOS虚拟机（VDM），并且在该VDM中运行16位Windows应用程序。按照默认设置，所有16位Windows应用程序都在单个共享的VDM中运行。在单独的VDM 中运行应用程序的优点是，如果应用程序崩溃，它只会使单个VDM停止工作，而在别的VDM中运行的其他程序仍然可以继续正常运行。另外，在单独的VDM中运行的16位Windows应用程序有它单独的输入队列。这意味着如果一个应用程序临时挂起，在各个VDM中的其他应用程序仍然可以继续接收输入信息。运行多个VDM的缺点是，每个VDM都要消耗大量的物理存储器。Windows 98在单个VDM中运行所有的16位Windows应用程序，不能改变这种情况。
- CREATE_SHARED_WOW_VDM标志只能当你在Windows 2000上运行16位Windows应用程序时使用。按照默认设置，除非设定了CREATE_SEPARATE_WOW_VDM标志，否则所有16位Windows应用程序都必须在单个 VDM中运行。但是，通过在注册表中将HKEY_LOCAL_MACHINE\system\CurrentControlSet\Control\WOW下的DefaultSeparateVDM设置为“yes”，就可以改变该默认行为特性。这时，CREATE_SHARED_WOW_VDM标志就在系统的共享VDM中运行16位Windows应用程序。
- CREATE_UNICODE_ENVIRONMENT标志用于告诉系统，子进程的环境块应该包含Unicode字符。按照默认设置，进程的环境块包含的是ANSI字符串。
- CREATE_FORCEDOS标志用于强制系统运行嵌入16位OS/2应用程序的MOS-DOS应用程序。
- CREATE_BREAKAWAY_FROM_JOB标志用于使作业中的进程生成一个与作业相关联的新进程（详细信息见第5章）。

fdwCreate参数也可以用来设定优先级类。不过用不着这样做，并且对于大多数应用程序来说不应该这样做，因为系统会为新进程赋予一个默认优先级。表4-5显示了各种可能的优先级类别。

表4-5 优先级类别

优先级类别	标志的标识符
空闲	IDLE_PRIORITY_CLASS
低于正常	BELOW_NORMAL_PRIORITY_CLASS
正常	NORMAL_PRIORITY_CLASS
高于正常	ABOVE_NORMAL_PRIORITY_CLASS
高	HIGH_PRIORITY_CLASS
实时	REALTIME_PRIORITY_CLASS

China-pub.com 下载

这些优先级类将会影响进程中包含的线程如何相对于其他进程的线程来进行调度。详细说明请见第7章。

注意 BELOW_NORMAL_PRIORITY_CLASS和ABOVE_NORMAL_PRIORITY_CLASS这两个优先级类在Windows 2000中是新类，Windows NT 4(或更早的版本)、Windows 95或Windows 98均不支持这两个类。

4.2.4 pvEnvironment

pvEnvironment参数用于指向包含新进程将要使用的环境字符串的内存块。在大多数情况下，为该参数传递NULL，使子进程能够继承它的父进程正在使用的一组环境字符串。也可以使用GetEnvironmentStrings函数：

```
PVOID GetEnvironmentStrings();
```

该函数用于获得调用进程正在使用的环境字符串数据块的地址。可以使用该函数返回的地址，作为CreateProcess的pvEnvironment参数。如果为pvEnvironment参数传递NULL，那么这正是CreateProcess函数所做的操作。当不再需要该内存块时，应该调用 FreeEnvironmentStrings函数将内存块释放：

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

4.2.5 pszCurDir

pszCurDir参数允许父进程设置子进程的当前驱动器和目录。如果本参数是 NULL，则新进程的工作目录将与生成新进程的应用程序的目录相同。如果本参数不是 NULL，那么pszCurDir必须指向包含需要的工作驱动器和工作目录的以 0结尾的字符串。注意，必须设定路径中的驱动器名。

4.2.6 psiStartInfo

psiStartInfo参数用于指向一个STARTUPINFO结构：

```
typedef struct _STARTUPINFO {
   DWORD cb;
   PSTR lpReserved;
   PSTR lpDesktop;
   PSTR lpTitle;
   DWORD dwX;
   DWORD dwY;
   DWORD dwXSize;
   DWORD dwYSize;
   DWORD dwXCountChars;
   DWORD dwYCountChars;
   DWORD dwFillAttribute;
   DWORD dwFlags;
   WORD wShowWindow;
   WORD cbReserved2;
   PBYTE lpReserved2;
   HANDLE hStdInput;
   HANDLE hStdOutput;
   HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

China-pub.com
下载

当Windows创建新进程时，它将使用该结构的有关成员。大多数应用程序将要求生成的应用程序仅仅使用默认值。至少应该将该结构中的所有成员初始化为零，然后将 cb成员设置为该结构的大小：

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

如果未能将该结构的内容初始化为零，那么该结构的成员将包含调用线程的堆栈上的任何无用信息。将该无用信息传递给 CreateProcess，将意味着有时会创建新进程，有时则不能创建新进程，完全取决于该无用信息。有一点很重要，那就是将该结构的未用成员设置为零，这样，CreateProcess就能连贯一致地运行。不这样做是开发人员最常见的错误。

这时，如果想要对该结构的某些成员进行初始化，只需要在调用 CreateProcess之前进行这项操作即可。我们将依次介绍每个成员。有些成员只有在子应用程序创建一个重叠窗口时才有意义，而另一些成员则只有在子应用程序执行基于 CUI的输入和输出时才有意义。表4-6描述了每个成员的作用。

表4-6 STARTUPINFO结构的成员

成 员	窗口，控制台还是两者兼有	作 用
cb	两者兼有	包含STARTUPINFO结构中的字节数。如果Microsoft将来扩展该结构，它可用作版本控制手段。应用程序必须将 cb初始化为sizeof(STARTUPINFO)
lpReserved	两者兼有	保留。必须初始化为 NULL
lpDesktop	两者兼有	用于标识启动应用程序所在的桌面的名字。如果该桌面存在，新进程便与指定的桌面相关联。如果桌面不存在，便创建一个带有默认属性的桌面，并使用为新进程指定的名字。如果lpDesktop是NULL（这是最常见的情况），那么该进程将与当前桌面相关联
lpTitle	控制台	用于设定控制台窗口的名称。如果 lpTitle是NULL，则可执行文件的名字将用作窗口名
dwX dwY	两者兼有	用于设定应用程序窗口在屏幕上应该放置的位置的 x和y坐标（以像素为单位）。只有当子进程用CW_USEDEFAULT作为CreateWindow的x参数来创建它的第一个重叠窗口时，才使用这两个坐标。若是创建控制台窗口的应用程序，这些成员用于指明控制台窗口的左上角
dwXSize dwYsize	两者兼有	用于设定应用程序窗口的宽度和长度（以像素为单位）只有当子进程将CW_USEDEFAULT用作CreateWindow的nWidth参数来创建它的第一个重叠窗口时，才使用这些值。若是创建控制台窗口的应用程序，这些成员将用于指明控制台窗口的宽度
dwXCountChars dwYCountChars	控制台	用于设定子应用程序的控制台窗口的宽度和高度（以字符为单位）
dwFillAttribute	控制台	用于设定子应用程序的控制台窗口使用的文本和背景颜色
dwFlags	两者兼有	请参见下一段和表4-7的说明
wShowWindow	窗口	用于设定如果子应用程序初次调用的 ShowWindow将SW_SHOWDEFAULT作为nCmdShow参数传递时，该应用程序的第一个重叠窗口应该如何出现。本成员可以是通常用于Show Window函数的任何一个SW_*标识符

China-pub.com 下载

(续)

成 员	窗口，控制台还是两者兼有	作 用
cbReserved2	两者兼有	保留。必须被初始化为0
lpReserved2	两者兼有	保留。必须被初始化为NULL
hStdInput hStdOutput hStdError	控制台	用于设定供控制台输入和输出用的缓存的句柄。按照默认设置，hStdInput用于标识键盘缓存，hStdOutput和hStdError用于标识控制台窗口的缓存

现在介绍dwFlags的成员。该成员包含一组标志，用于修改如何来创建子进程。大多数标志只是告诉CreateProcess，STARTUPINFO结构的其他成员是否包含有用的信息，或者某些成员是否应该忽略。表4-7标出可以使用的标志及其含义。

表4-7 使用标志及含义

标 志	含 义
STARTF_USESIZE	使用dwXSize和dwYSize成员
STARTF_USESHOWWINDOW	使用wShowWindow成员
STARTF_USEPOSITION	使用dwX和dwY成员
STARTF_USECOUNTCHARS	使用dwXCountChars和dwYCount Chars成员
STARTF_USEFILLATTRIBUTE	使用dwFillAttribute成员
STARTF_USESTDHANDLES	使用hStdInput、hStdOutput和hStdError成员
STARTF_RUN_FULLSCREEN	强制在x86计算机上运行的控制台应用程序以全屏幕方式启动运行

另外还有两个标志，即 STARTF_FORCEONFEEDBACK和STARTF_＋FORCEOFFF-EEDBACK，当启动一个新进程时，它们可以用来控制鼠标的光标。由于 Windows支持真正的多任务抢占式运行方式，因此可以启动一个应用程序，然后在进程初始化时，使用另一个程序。为了向用户提供直观的反馈信息， CreateProcess能够临时将系统的箭头光标改为一个新光标，即沙漏箭头光标：



该光标表示可以等待出现某种情况，也可以继续使用系统。当启动另一个进程时，CreateProcess函数使你能够更好地控制光标。当设定 STARTF_FORCEOFFFEEDBACK标志时，CreateProcess并不将光标改为沙漏。

STARTF_FORCEONFEEDBACK可使CreateProcess能够监控新进程的初始化，并可根据结果来改变光标。当使用该标志来调用 CreateProcess时，光标改为沙漏。过2s后，如果新进程没有调用GUI，CreateProcess 将光标恢复为箭头。

如果该进程在2s内调用了GUI，CreateProcess将等待该应用程序显示一个窗口。这必须在进程调用GUI后5s内发生。如果没有显示窗口， CreateProcess就会恢复原来的光标。如果显示了一个窗口， CreateProcess将使沙漏光标继续保留 5s。如果某个时候该应用程序调用了GetMessage函数，指明它完成了初始化，那么CreateProcess就会立即恢复原来的光标，并且停止监控新进程。

在结束这一节内容的介绍之前，我想讲一讲STARTUPINFO的wShowWindow成员。你将该成员初始化为传递给(w)WinMain的最后一个参数nCmdShow的值。该成员显示你想要传递给新进程的(w)WinMain函数的最后一个参数nCmdShow的值。它是可以传递给ShowWindow函数的标识符之一。通常，nCmdShow的值既可以是SW_SHOWNORMAL，也可以是SW_ SHOWMINNOACTIVE。

China-pub.com
下载

但是，它有时可以是SW_SHOWDEFAULT。

当在Explorer中启动一个应用程序时，该应用程序的(w)WinMain函数被调用，而SW_SHOWNORMAL则作为nCmdShow参数来传递。如果为该应用程序创建了一个快捷方式，可以使用快捷方式的属性页来告诉系统，应用程序的窗口最初应该如何显示。图4-3显示了运行Notepad的快捷方式的属性页。注意，使用Run选项的组合框，就能够设定如何显示Notepad的窗口。

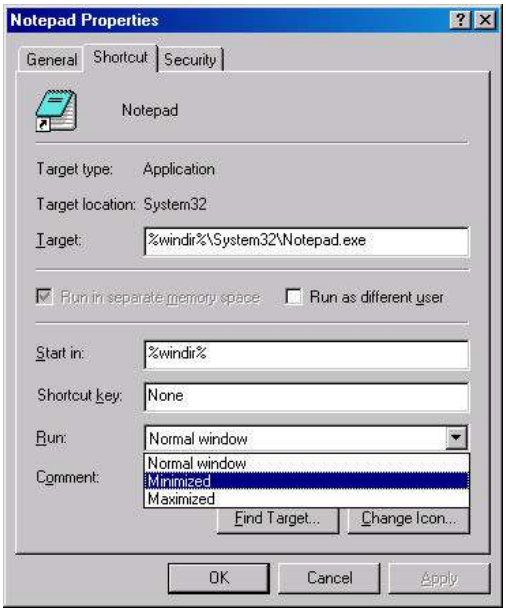


图4-3 运行Notepad的快捷方式的属性页

当使用Explorer来启动该快捷方式时，Explorer会正确地准备STARTUPINFO结构并调用CreateProcess。这时Notepad开始运行，并且为nCmdShow参数将SW_SHOWMINNOACTIVE传递给它的(w)WinMain函数。

运用这样的方法，用户能够很容易地启动一个应用程序，其主窗口可以用正常状态、最小或最大状态进行显示。

最后，应用程序可以调用下面的函数，以便获取由父进程初始化的STARTUPINFO结构的拷贝。子进程可以查看该结构，并根据该结构的成员的值来改变它的行为特性。

```
VOID GetStartupInfo(LPSTARTUPINFO pStartupInfo);
```

注意 虽然Windows文档没有明确地说明，但是在调用GetStartInfo函数之前，必须像下面这样对该结构的cb成员进行初始化：

```
STARTUPINFO si = { sizeof(si) };
GetStartupInfo(&si);
:
```

4.2.7 ppiProcInfo

ppiProcInfo参数用于指向你必须指定的PROCESS_INFORMATION结构。CreateProcess在返回之前要对该结构的成员进行初始化。该结构的形式如下面所示：

```
typedef struct _PROCESS_INFORMATION {
   HANDLE hProcess;
   HANDLE hThread;
   DWORD  dwProcessId;
   DWORD  dwThreadId;
} PROCESS_INFORMATION;
```

如前所述，创建新进程可使系统建立一个进程内核对象和一个线程内核对象。在创建进程的时候，系统为每个对象赋予一个初始使用计数值1。然后，在createProcess返回之前，该函数打开进程对象和线程对象，并将每个对象的与进程相关的句柄放入PROCESS_INFORMATION结构的hProcess和hThread成员中。当CreateProcess在内部打开这些对象时，每个对象的使用计数就变为2。

China-pub.com 下载

这意味着在系统能够释放进程对象前，该进程必须终止运行（将使用计数递减为 1），并且父进程必须调用 CloseHandle（再将使用计数递减 1，使之变为 0）。同样，若要释放线程对象，该线程必须终止运行，父进程必须关闭线程对象的句柄（关于释放线程对象的详细说明，请参见本章后面“子进程”一节的内容）。

注意 必须关闭子进程和它的主线程的句柄，以避免在应用程序运行时泄漏资源。当然，当进程终止运行时，系统会自动消除这些泄漏现象，但是，当进程不再需要访问子进程和它的线程时，编写得较好的软件能够显式关闭这些句柄（通过调用 CloseHandle函数来关闭）。不能关闭这些句柄是开发人员最常犯的错误之一。

由于某些原因，许多开发人员认为，关闭进程或线程的句柄，会促使系统撤消该进程或线程。实际情况并非如此。关闭句柄只是告诉系统，你对进程或线程的统计数据不感兴趣。进程或线程将继续运行，直到它自己终止运行。

当进程内核对象创建后，系统赋予该对象一个独一无二的标识号，系统中的其他任何进程内核对象都不能使用这个相同的 ID号。线程内核对象的情况也一样。当一个线程内核对象创建时，该对象被赋予一个独一无二的、系统范围的 ID号。进程 ID和线程 ID共享相同的号码池。这意味着进程和线程不可能拥有相同的 ID。另外，对象决不会被赋予 0作为其 ID。在 CreateProcess返回之前，它要用这些 ID填入PROCESS_INFORMATION结构的dwProcessId和 dwThreadId成员中。ID使你能够非常容易地识别系统中的进程和线程。一些实用工具（如 Task Manager）对ID使用得最多，而高效率的应用程序则使用得很少。由于这个原因，大多数应用程序完全忽略ID。

如果应用程序使用 ID来跟踪进程和线程，必须懂得系统会立即复用进程 ID和线程ID。例如，当一个进程被创建时，系统为它指定一个进程对象，并为它赋予 ID值122。如果创建了一个新进程对象，系统不会将相同的ID赋予给它。但是，如果第一个进程对象被释放，系统就可以将122赋予创建的下一个进程对象。记住这一点后，就能避免编写引用不正确的进程对象或线程对象的代码。获取进程 ID是很容易的，保存该 ID也不难，但是，接下来你应该知道，该 ID标识的进程已被释放，新进程被创建并被赋予相同的 ID。当使用已经保存的进程 ID时，最终操作的是新进程，而不是原先获得 ID的进程。

有时，运行的应用程序想要确定它的父进程。首先应该知道只有在生成子进程时，才存在进程之间的父子关系。在子进程开始执行代码前，Windows不再考虑存在什么父子关系。较早的Windows版本没有提供让进程查询其父进程的函数。现在，ToolHelp函数通过 PROCESSENTRY32结构使得这种查询成为可能。在这个结构中有一个 th32ParentProcessID成员，根据文档的说明，它能返回进程的父进程的 ID。

系统无法记住每个进程的父进程的ID，但是，由于ID是被立即重复使用的，因此，等到获得父进程的ID时，该ID可能标识了系统中一个完全不同的进程。父进程可能已经终止运行。如果应用程序想要与它的“创建者”进行通信，最好不要使用 ID。应该定义一个持久性更好的机制，比如内核对象和窗口句柄等。

若要确保进程ID或线程ID不被重复使用，唯一的方法是保证进程或线程的内核对象不会被撤消。如果刚刚创建了一个新进程或线程，只要不关闭这些对象的句柄，就能够保证进程对象不被撤消。一旦应用程序结束使用该 ID，那么调用CloseHandle就可以释放内核对象，要记住，这时使用或依赖进程 ID，对来说将不再安全。如果使用的是子进程，将无法保证父进程或父线程的有效性，除非父进程复制了它自己的进程对象或线程对象的句柄，并让子进程继承这些句柄。

China-pub.com
下载

4.3 终止进程的运行

若要终止进程的运行，可以使用下面四种方法：

- 主线程的进入点函数返回（最好使用这个方法）。
- 进程中的一个线程调用ExitProcess函数（应该避免使用这种方法）。
- 另一个进程中的线程调用TerminateProcess函数（应该避免使用这种方法）。
- 进程中的所有线程自行终止运行（这种情况几乎从未发生）。

这一节将介绍所有这四种方法，并且说明进程结束时将会发生什么情况。

4.3.1 主线程的进入点函数返回

始终都应该这样来设计应用程序，即只有当主线程的进入点函数返回时，它的进程才终止运行。这是保证所有线程资源能够得到正确清除的唯一办法。

让主线程的进入点函数返回，可以确保下列操作的实现：

- 该线程创建的任何C++对象将能使用它们的析构函数正确地撤消。
- 操作系统将能正确地释放该线程的堆栈使用的内存。
- 系统将进程的退出代码（在进程的内核对象中维护）设置为进入点函数的返回值。
- 系统将进程内核对象的返回值递减1。

4.3.2 ExitProcess函数

当进程中的一个线程调用ExitProcess函数时，进程便终止运行：

```
VOID ExitProcess(UINT fuExitCode)
```

该函数用于终止进程的运行，并将进程的退出代码设置为 fuExitCode。ExitProcess函数并不返回任何值，因为进程已经终止运行。如果在调用 ExitProcess之后又增加了什么代码，那么该代码将永远不会运行。

当主线程的进入点函数（ WinMain、wWinMain、main或wmain）返回时，它将返回给C/C++运行期启动代码，它能正确地清除该进程使用的所有的C运行期资源。当C运行期资源被释放之后，C运行期启动代码就显式调用 ExitProcess，并将进入点函数返回的值传递给它。这解释了为什么只需要主线程的进入点函数返回，就能够终止整个进程的运行。请注意，进程中运行的任何其他线程都随着进程而一道终止运行。

Windows Platform SDK文档声明，进程要等到所有线程终止运行之后才终止运行。就操作系统而言，这种说法是对的。但是， C/C++运行期对应用程序采用了不同的规则，通过调用ExitProcess，使得C/C++运行期启动代码能够确保主线程从它的进入点函数返回时，进程便终止运行，而不管进程中是否还有其他线程在运行。不过，如果在进入点函数中调用 ExitThread，而不是调用ExtiProcess或者仅仅是返回，那么应用程序的主线程将停止运行，但是，如果进程中至少有一个线程还在运行，该进程将不会终止运行。

注意，调用ExitProcess或ExitThread可使进程或线程在函数中就终止运行。就操作系统而言，这很好，进程或线程的所有操作系统资源都将被全部清除。但是， C/C++应用程序应该避免调用这些函数，因为C/C++运行期也许无法正确地清除。请看下面的代码：

```
#include <windows.h>
#include <stdio.h>
class CSomeObj {
public:
```

China-pub.com 下载

```
   CSomeObj()  { printf("Constructor\r\n"); }
   ~CSomeObj() { printf("Destructor\r\n"); }
};

CSomeObj g_GlobalObj;

void main () {
   CSomeObj LocalObj;
   ExitProcess(0);      // This shouldn't be here

   // At the end of this function, the compiler automatically added
   // the code necessary to call LocalObj's destructor.
   // ExitProcess prevents it from executing.
}
```

当上面的代码运行时，将会看到：

```
Constructor
Constructor
```

它创建了两个对象，一个是全局对象，另一个是局部对象。不过决不会看到 Destructor这个单词出现， C++对象没有被正确地撤消，因为 ExitProcess函数强制进程在现场终止运行，C/C++运行期没有机会进行清除。

如前所述，决不应该显式调用 ExitProcess函数。如果在上面的代码中删除了对 ExitProcess的调用，那么运行该程序产生的结果如下：

```
Constructor
Constructor
Destructor
Destructor
```

只要让主线程的进入点函数返回， C/C++运行期就能够执行它的清除操作，并且正确地撤消任何或所有的C++对象。顺便讲一下，这个说明不仅仅适用于 C++对象。C++运行期能够代表进程执行许多操作，最好允许运行期正确地将它清除。

注意 显式调用ExitProcess和ExitThread是导致应用程序不能正确地将自己清除的常见原因。在调用ExitThread时，进程将继续运行，但是可能会泄漏内存或其他资源。

4.3.3 TerminateProcess函数

调用TerminateProcess函数也能够终止进程的运行：

```
BOOL TerminateProcess(
   HANDLE hProcess,
   UINT fuExitCode);
```

该函数与ExitProcess有一个很大的差别，那就是任何线程都可以调用 TerminateProcess来终止另一个进程或它自己的进程的运行。 hProcess参数用于标识要终止运行的进程的句柄。当进程终止运行时，它的退出代码将成为你作为 fuExitCode参数来传递的值。

只有当无法用另一种方法来迫使进程退出时，才应该使用 TerminateProcess。终止运行的进程绝对得不到关于它将终止运行的任何通知，因为应用程序无法正确地清除，并且不能避免自己被撤消（除非通过正常的安全机制）。例如，进程无法将内存中它拥有的任何信息迅速送往磁盘。

China-pub.com
下载

虽然进程确实没有机会执行自己的清除操作，但是操作系统可以在进程之后进行全面的清除，使得所有操作系统资源都不会保留下来。这意味着进程使用的所有内存均被释放，所有打开的文件全部关闭，所有内核对象的使用计数均被递减，同时所有的用户对象和 GDI对象均被撤消。

一旦进程终止运行（无论采用何种方法），系统将确保该进程不会将它的任何部分遗留下来。绝对没有办法知道该进程是否曾经运行过。进程一旦终止运行，它绝对不会留下任何蛛丝马迹。希望这是很清楚的。

注意 TerminateProcess函数是个异步运行的函数，也就是说，它会告诉系统，你想要进程终止运行，但是当函数返回时，你无法保证该进程已经终止运行。因此，如果想要确切地了解进程是否已经终止运行，必须调用WaitForSingleObject函数（第9章介绍）或者类似的函数，并传递进程的句柄。

进程中的线程何时全部终止运行

如果进程中的所有线程全部终止运行（因为它们调用了 ExitThread函数，或者因为它们已经用TerminateProcess函数终止运行），操作系统就认为没有理由继续保留进程的地址空间。这很好，因为在地址空间中没有任何线程执行任何代码。当系统发现没有任何线程仍在运行时，它就终止进程的运行。出现这种情况时，进程的退出代码被设置为与终止运行的最后一个线程相同的退出代码。

4.3.4 进程终止运行时出现的情况

当进程终止运行时，下列操作将启动运行：

1) 进程中剩余的所有线程全部终止运行。

2) 进程指定的所有用户对象和GDI对象均被释放，所有内核对象均被关闭（如果没有其他进程打开它们的句柄，那么这些内核对象将被撤消。但是，如果其他进程打开了它们的句柄，内核对象将不会撤消）。

3) 进程的退出代码将从STILL_ACTIVE改为传递给ExitProcess或TerminateProcess的代码。

4) 进程内核对象的状态变成收到通知的状态（关于传送通知的详细说明，参见第 9章）。系统中的其他线程可以挂起，直到进程终止运行。

5) 进程内核对象的使用计数递减1。

注意，进程的内核对象的寿命至少可以达到进程本身那么长，但是进程内核对象的寿命可能大大超过它的进程寿命。当进程终止运行时，系统能够自动确定它的内核对象的使用计数。如果使用计数降为0，那么没有其他进程拥有该对象打开的句柄，当进程被撤消时，对象也被撤消。

不过，如果系统中的另一个进程拥有正在被撤消的进程的内核对象的打开句柄，那么该进程内核对象的使用计数不会降为 0。当父进程忘记关闭子进程的句柄时，往往就会发生这样的情况。这是个特性，而不是错误。记住，进程内核对象维护关于进程的统计信息。即使进程已经终止运行，该信息也是有用的。例如，你可能想要知道进程需要多少 CPU时间，或者，你想通过调用GetExitCodeProcess来获得目前已经撤消的进程的退出代码：

```
BOOL GetExitCodeProcess(
   HANDLE hProcess,
   PDWORD pdwExitCode);
```

该函数查看进程的内核对象（由hProcess参数来标识），取出内核对象的数据结构中用于标

China-pub.com 下载

识进程的退出代码的成员。该退出代码的值在pdwExitCode参数指向的DWORD中返回。

可以随时调用该函数。如果调用GetExitCodeProcess函数时进程尚未终止运行，那么该函数就用STILL_ACTIVE标识符（定义为0x103）填入DWORD。如果进程已经终止运行，便返回数据的退出代码值。

也许你会认为，你可以编写代码，通过定期调用GetExitCodeProcess函数并且检查退出代码来确定进程是否已经终止运行。大多数情况下，这是可行的，但是效率不高。下一段将介绍用什么正确的方法来确定进程何时终止运行。

再一次提醒你，应该通过调用CloseHandle函数，告诉系统你对进程的统计数据已经不再感兴趣。如果进程已经终止运行，CloseHandle将递减内核对象的使用计数，并将它释放。

4.4 子进程

当你设计应用程序时，可能会遇到这样的情况，即想要另一个代码块来执行操作。通过调用函数或子例程，你可以一直象这样分配工作。当调用一个函数时，在函数返回之前，代码将无法继续进行操作。大多数情况下，需要实施这种单任务同步。让另一个代码块来执行操作的另一种方法是在进程中创建一个新线程，并让它帮助进行操作。这样，当其他线程在执行需要的操作时，代码就能继续进行它的处理。这种方法很有用，不过，当线程需要查看新线程的结果时，它会产生同步问题。

另一个解决办法是生成一个新进程，即子进程，以便帮助你进行操作。比如说，需要进行的操作非常复杂。若要处理该操作，只需要在同一个进程中创建一个新线程。你编写一些代码，对它进行测试，但是得到一些不正确的结果。也许你的算法存在错误，也可能间接引用的对象不正确，并且不小心改写了地址空间中的某些重要内容。进行操作处理时，如果要保护地址空间，方法之一是让一个新进程来执行这项操作。然后，在继续进行工作之前，可以等待新进程终止运行，或者可以在新进程工作时，继续进行工作。

不过，新进程可能需要对地址空间中包含的数据进行操作。这时最好让进程在它自己的地址空间中运行，并且只让它访问父进程地址空间中的相关数据，这样就能保护与手头正在执行的任务无关的全部数据。Windows提供了若干种方法，以便在不同的进程中间传送数据，比如动态数据交换（DDE）、OLE、管道和邮箱等。共享数据最方便的方法之一是，使用内存映射文件（关于内存映射文件的详细说明请参见第17章）。

如果想创建新进程，让它进行一些操作，并且等待结果，可以使用类似下面的代码：

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

// Spawn the child process.
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

   // Close the thread handle as soon as it is no longer needed!
   CloseHandle(pi.hThread);

   // Suspend our execution until the child has terminated.
   WaitForSingleObject(pi.hProcess, INFINITE);

   // The child process terminated; get its exit code.
   GetExitCodeProcess(pi.hProcess, &dwExitCode);
```

China-pub.com
下载


```
    // Close the process handle as soon as it is no longer needed.
    CloseHandle(pi.hProcess);
}
```

在上面的代码段中，你创建了一个新进程，如果创建成功，可以调用 WaitForSingleObject 函数：

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeout);
```

第9章将全面介绍 WaitForSingleObject函数。现在，必须知道的情况是，它会一直等到hObject参数标识的对象得到通知的时候。当进程对象终止运行时，它们才会得到通知。因此对WaitForSingleObject的调用会将父进程的线程挂起，直到子进程终止运行。当WaitForSingleObject返回时，通过调用GetExitCodeProcess函数，就可以获得子进程的退出代码。

在上面的代码段中调用CloseHandle函数，可使系统为线程和进程对象的使用计数递减为0，从而使对象的内存得以释放。

你会发现，在这个代码段中，在CreateProcess返回后，立即关闭了子进程的主线程内核对象的句柄。这并不会导致子进程的主线程终止运行，它只是递减子进程的主线程对象的使用计数。这种做法的优点是，假设子进程的主线程生成了另一个线程，然后主线程终止运行，这时，如果父进程不拥有子进程的主线程对象的句柄，那么系统就可以从内存中释放子进程的主线程对象。但是，如果父进程拥有子进程的线程对象的句柄，那么在父进程关闭句柄前，系统将不能释放该对象。

运行独立的子进程

大多数情况下，应用程序将另一个进程作为独立的进程来启动。这意味着进程创建和开始运行后，父进程并不需要与新进程进行通信，也不需要在完成它的工作后父进程才能继续运行。这就是Explorer的运行方式。当Explorer为用户创建一个新进程后，它并不关心该进程是否继续运行，也不在乎用户是否终止它的运行。

若要放弃与子进程的所有联系，Explorer必须通过调用CloseHandle来关闭它与新进程及它的主线程之间的句柄。下面的代码示例显示了如何创建新进程以及如何让它以独立方式来运行：

```
PROCESS_INFORMATION pi;

// Spawn the child process.
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

   // Allow the system to destroy the process & thread kernel
   // objects as soon as the child process terminates.
   CloseHandle(pi.hThread);
   CloseHandle(pi.hProcess);
}
```

4.5 枚举系统中运行的进程

许多软件开发人员都试图为 Windows编写需要枚举正在运行的一组进程的工具或实用程序。Windows API原先没有用于枚举正在运行的进程的函数。不过， Windows NT一直在不断

China-pub.com 下载

更新称为Performance Data的数据库。该数据库包含大量的信息，并且可以通过注册表函数来访问（比如以HKEY_PERFORMANCE_DATA为根关键字的RegQueryValueEx函数）。由于下列原因，很少有Windows程序员知道性能数据库的情况：

- 它没有自己特定的函数，它只是使用现有的注册表函数。
- Windows 95和Windows 98没有配备该数据库。
- 该数据库中的信息布局比较复杂，许多软件开发人员都不愿使用它。这妨碍了人们通过言传口说来传播它的存在。

为了使该数据库的使用变得更加容易，Microsoft开发了一组Performance Data Helper函数（包含在PDH.dll文件中）。若要了解它的详细信息，请查看Platform SDK文档中的Performance Data Helper的内容。

如前所述，Windows 95和Windows 98没有配备该数据库。不过它们有自己的一组函数，可以用于枚举关于它们的进程和信息。这些函数均在ToolHelp API中。详细信息请参见Platform SDK文档中的Process32First和Process32Next函数。

更加有趣的是，Microsoft的Windows NT开发小组因为不喜欢ToolHelp函数，所以没有将这些函数添加给Windows NT。相反，他们开发了自己的Process Status函数，用于枚举进程（这些函数包含在PSAPI.dll文件中）。关于这些函数的详细说明，请参见Platform SDK文档中的EnumProcesses函数。

Microsoft似乎使得工具和实用程序开发人员的日子很不好过，不过我高兴地告诉他们，Microsoft已经将ToolHelp函数添加给Windows 2000。最后，开发人员终于有了一种方法，可以为Windows 95、Windows 98和Windows 2000编写具有公用源代码的工具和实用程序。

进程信息示例应用程序

ProcessInfo应用程序“04 ProcessInfo.exe”（本章结尾处的清单4-2列出了该文件）显示了如何使用ToolHelp函数来开发非常有用的实用程序。用于应用程序的源代码和资源文件均放在本书所附光盘上04-ProcessInfo目录中。当启动该程序时，便会出现图4-4所示的窗口。

ProcessInfo首先枚举目前正在运行的一组进程，并在顶部的组合框中列出每个进程的名字和ID。然后，第一个进程被选定，并在较大的编辑控件中显示关于该进程的信息。可以看到，与该进程的ID一道显示的还有它的父进程的ID，进程的优先级类，以及该进程环境中当前正在运行的线程数目。这些信息中的大多数不在本章介绍的范围之内，将在本章后面的内容中加以说明。

当查看这个进程列表时，可以使用VMMap菜单项（当查看模块信息时，该菜单项禁用）。如果选定VMMap菜单项，可使VMMap示例应用程序（参见第14章）启动运行。该应用程序将在指定进程的地址空间中运行。

模块信息部分显示了映射到进程的地址空间中的模块的列表（可执行文件和DLL文件）。固定模块是指进程初始化时隐含加载的模块。如果是显式加载的DLL模块，则显示DLL的使用计数。第二个域显示映射模块的地址。如果模块不是在它的首选基地址上映射的，那么首选基地址显示在括号中。第三个域显示模块的大小（用字节数表示）。最后显示的是模块的全路径名。线程信息部分显示了该进程中当前运行的一组线程。每个线程ID和优先级均被显示。

除了进程信息外，可以选择Modules!菜单项。这将使得ProcessInfo能够枚举当前通过系统加载的模块，并将每个模块的名字放入顶部的组合框。然后ProcessInfo可以选定第一个模块，并显示关于它的信息，如图4-5所示。

China-pub.com
下载

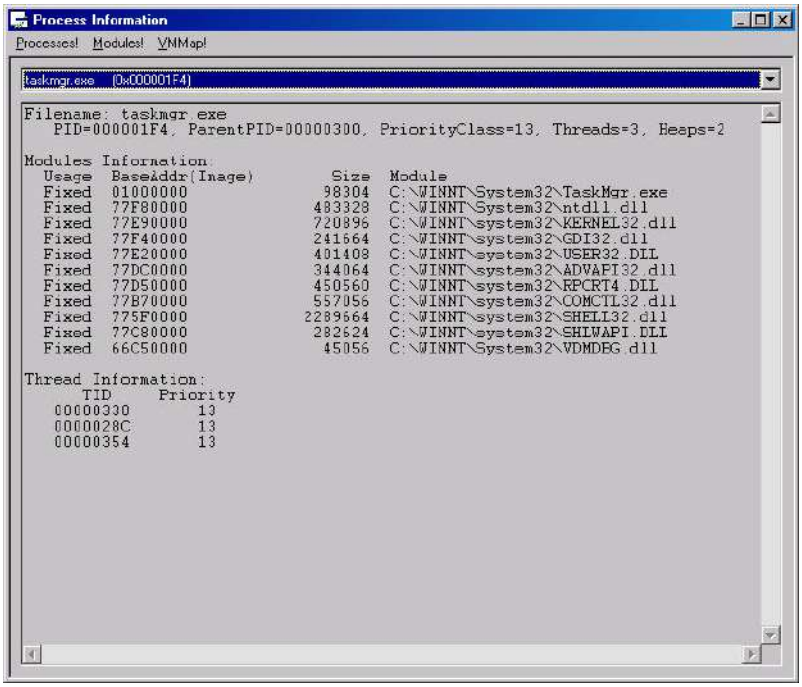


图4-4 运行中的ProcessInfo

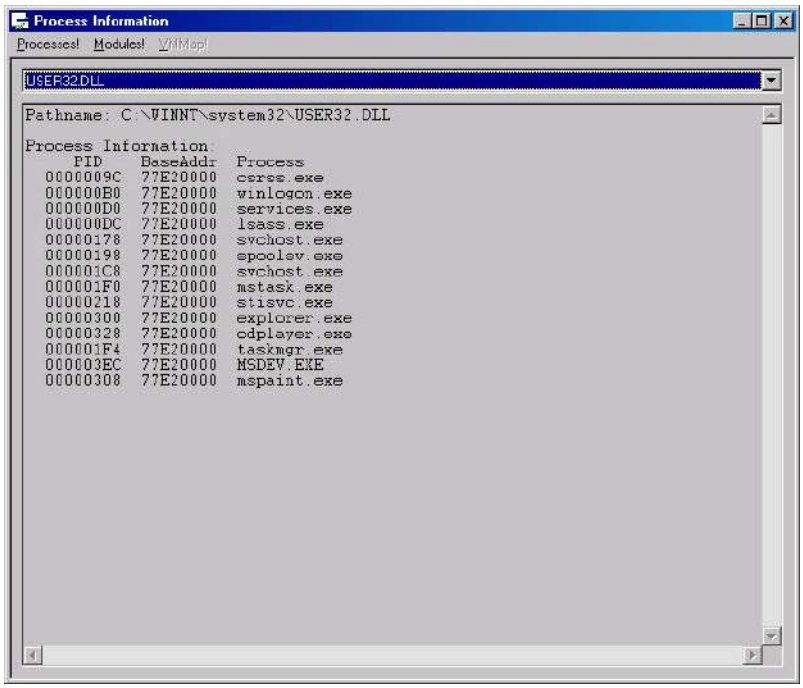


图4-5 ProcessInfo显示User32.dll加载到它们的地址空间的所有进程

当以这种方法使用ProcessInfo实用程序时，能够方便地确定哪些进程正在使用某个模块。如你所见，模块的全路径名显示在顶部。然后，进程信息部分显示包含该模块的进程列表。除

China-pub.com 下载

了每个进程的ID和名字外，还显示每个进程中模块加载到的地址。

ProcessInfo应用程序显示的所有信息基本上都是通过调用 ToolHelp的各个函数而产生的。为了使ToolHelp函数的使用更加容易，我创建了一个C++类（包含在ToolHelp.h文件中）。这个C++类封装了一个ToolHelp快照，使得调用其他ToolHelp函数稍稍容易一些。

ProcessInfo.cpp中的GetModulePreferredBaseAddr函数是个特别有意思的函数：

```
PVOID GetModulePreferredBaseAddr(
   DWORD dwProcessId,
   PVOID pvModuleRemote);
```

该函数接受一个进程ID和该进程中的一个模块的地址。然后它查看该进程的地址空间，找出该模块，并读取模块的标题信息，以确定该模块首选的基地址，一个模块始终应该加载到它的首选基地址中，否则，使用该模块的应用程序将需要更多的内存，并且在初始化时会对性能产生影响。由于这是个非常可怕的情况，因此我增加了这个函数并且显示何时模块没有加载到它的首选基地址中。第20章将要进一步介绍首选基地址和这次/内存性能。

清单4-2 ProcessInfo应用程序

ProcessInfo.cpp

```
/******************************************************************************
Module:  ProcessInfo.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tlhelp32.h>
#include <tchar.h>
#include <stdarg.h>
#include <stdio.h>
#include "Toolhelp.h"
#include "Resource.h"
//////////////////////////////////////////////////////////////////////////////


// Adds a string to an edit control
void AddText(HWND hwnd, PCTSTR pszFormat, ...) {

   va_list argList;
   va_start(argList, pszFormat);

   TCHAR sz[20 * 1024];
   Edit_GetText(hwnd, sz, chDIMOF(sz));
   _vstprintf(_tcschr(sz, 0), pszFormat, argList);
   Edit_SetText(hwnd, sz);
   va_end(argList);
}


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
VOID Dlg_PopulateProcessList(HWND hwnd) {

   HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
   SetWindowRedraw(hwndList, FALSE);
   ComboBox_ResetContent(hwndList);

   CToolhelp thProcesses(TH32CS_SNAPPROCESS);
   PROCESSENTRY32 pe = { sizeof(pe) };
   BOOL fOk = thProcesses.ProcessFirst(&pe);
   for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {
      TCHAR sz[1024];

     // Place the process name (without its path) & ID in the list
     PCTSTR pszExeFile = _tcsrchr(pe.szExeFile, TEXT('\\'));
     if (pszExeFile == NULL) pszExeFile = pe.szExeFile;
     else pszExeFile++; // Skip over the slash
     wsprintf(sz, TEXT("%s     (0x%08X)"), pszExeFile, pe.th32ProcessID);
     int n = ComboBox_AddString(hwndList, sz);

      // Associate the process ID with the added item
      ComboBox_SetItemData(hwndList, n, pe.th32ProcessID);
   }
   ComboBox_SetCurSel(hwndList, 0);  // Select the first entry

   // Simulate the user selecting this first item so that the
   // results pane shows something interesting
   FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
      hwndList, CBN_SELCHANGE, SendMessage);

   SetWindowRedraw(hwndList, TRUE);
   InvalidateRect(hwndList, NULL, FALSE);
}

//////////////////////////////////////////////////////////////////////////////

VOID Dlg_PopulateModuleList(HWND hwnd) {

   HWND hwndModuleHelp = GetDlgItem(hwnd, IDC_MODULEHELP);
   ListBox_ResetContent(hwndModuleHelp);

   CToolhelp thProcesses(TH32CS_SNAPPROCESS);
   PROCESSENTRY32 pe = { sizeof(pe) };
   BOOL fOk = thProcesses.ProcessFirst(&pe);
   for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {

      CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
      MODULEENTRY32 me = { sizeof(me) };
      BOOL fOk = thModules.ModuleFirst(&me);
      for (; fOk; fOk = thModules.ModuleNext(&me)) {
        int n = ListBox_FindStringExact(hwndModuleHelp, -1, me.szExePath);
         if (n == LB_ERR) {
          // This module hasn't been added before
            ListBox_AddString(hwndModuleHelp, me.szExePath);
         }
      }
```

China-pub.com 下载

```
   }


   HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
   SetWindowRedraw(hwndList, FALSE);
   ComboBox_ResetContent(hwndList);
   int nNumModules = ListBox_GetCount(hwndModuleHelp);
   for (int i = 0; i < nNumModules; i++) {
      TCHAR sz[1024];
      ListBox_GetText(hwndModuleHelp, i, sz);
      // Place module name (without its path) in the list
      int nIndex = ComboBox_AddString(hwndList, _tcsrchr(sz, TEXT('\\')) + 1);
      // Associate the index of the full path with the added item
      ComboBox_SetItemData(hwndList, nIndex, i);
   }

   ComboBox_SetCurSel(hwndList, 0);  // Select the first entry

   // Simulate the user selecting this first item so that the
   // results pane shows something interesting
   FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
      hwndList, CBN_SELCHANGE, SendMessage);

   SetWindowRedraw(hwndList, TRUE);
   InvalidateRect(hwndList, NULL, FALSE);
}


//////////////////////////////////////////////////////////////////////////////


PVOID GetModulePreferredBaseAddr(DWORD dwProcessId, PVOID pvModuleRemote) {

   PVOID pvModulePreferredBaseAddr = NULL;
   IMAGE_DOS_HEADER idh;
   IMAGE_NT_HEADERS inth;

   // Read the remote module's DOS header
   Toolhelp32ReadProcessMemory(dwProcessId,
      pvModuleRemote, &idh, sizeof(idh), NULL);

   // Verify the DOS image header
   if (idh.e_magic == IMAGE_DOS_SIGNATURE) {
      // Read the remote module's NT header
      Toolhelp32ReadProcessMemory(dwProcessId,
         (PBYTE) pvModuleRemote + idh.e_lfanew, &inth, sizeof(inth), NULL);

      // Verify the NT image header
      if (inth.Signature == IMAGE_NT_SIGNATURE) {
         // This is valid NT header, get the image's preferred base address
         pvModulePreferredBaseAddr = (PVOID) inth.OptionalHeader.ImageBase;
      }
   }
   return(pvModulePreferredBaseAddr);
}
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////

VOID ShowProcessInfo(HWND hwnd, DWORD dwProcessID) {

   SetWindowText(hwnd, TEXT(""));   // Clear the output box

   CToolhelp th(TH32CS_SNAPALL, dwProcessID);

   // Show Process details
   PROCESSENTRY32 pe = { sizeof(pe) };
   BOOL fOk = th.ProcessFirst(&pe);
   for (; fOk; fOk = th.ProcessNext(&pe)) {
      if (pe.th32ProcessID == dwProcessID) {
         AddText(hwnd, TEXT("Filename: %s\r\n"), pe.szExeFile);
         AddText(hwnd, TEXT("   PID=%08X, ParentPID=%08X, ")
            TEXT("PriorityClass=%d, Threads=%d, Heaps=%d\r\n"),
            pe.th32ProcessID, pe.th32ParentProcessID,
            pe.pcPriClassBase, pe.cntThreads,
            th.HowManyHeaps());
         break;   // No need to continue looping
      }
   }

   // Show Modules in the Process
   // Number of characters to display an address
   const int cchAddress = sizeof(PVOID) * 2;
   AddText(hwnd, TEXT("\r\nModules Information:\r\n")
      TEXT("  Usage  %-*s(%-*s)  %8s  Module\r\n"),
      cchAddress, TEXT("BaseAddr"),
      cchAddress, TEXT("ImagAddr"), TEXT("Size"));

   MODULEENTRY32 me = { sizeof(me) };
   fOk = th.ModuleFirst(&me);
   for (; fOk; fOk = th.ModuleNext(&me)) {
      if (me.ProccntUsage == 65535) {
         // Module was implicitly loaded and cannot be unloaded
         AddText(hwnd, TEXT("  Fixed"));
      } else {
         AddText(hwnd, TEXT("  %5d"), me.ProccntUsage);
      }
      PVOID pvPreferredBaseAddr =
         GetModulePreferredBaseAddr(pe.th32ProcessID, me.modBaseAddr);
      if (me.modBaseAddr == pvPreferredBaseAddr) {
         AddText(hwnd, TEXT("  %p %*s   %8u  %s\r\n"),
            me.modBaseAddr, cchAddress, TEXT(""),
            me.modBaseSize, me.szExePath);
      } else {
         AddText(hwnd, TEXT("  %p(%p)  %8u  %s\r\n"),
            me.modBaseAddr, pvPreferredBaseAddr, me.modBaseSize, me.szExePath);
      }
   }


   // Show threads in the process
```

China-pub.com 下载

```
   AddText(hwnd, TEXT("\r\nThread Information:\r\n")
      TEXT("      TID     Priority\r\n"));
   THREADENTRY32 te = { sizeof(te) };
   fOk = th.ThreadFirst(&te);
   for (; fOk; fOk = th.ThreadNext(&te)) {
      if (te.th32OwnerProcessID == dwProcessID) {
         int nPriority = te.tpBasePri + te.tpDeltaPri;
         if ((te.tpBasePri < 16) && (nPriority > 15)) nPriority = 15;
         if ((te.tpBasePri > 15) && (nPriority > 31)) nPriority = 31;
         if ((te.tpBasePri < 16) && (nPriority <  1)) nPriority =  1;
         if ((te.tpBasePri > 15) && (nPriority < 16)) nPriority = 16;
         AddText(hwnd, TEXT("   %08X       %2d\r\n"),
            te.th32ThreadID, nPriority);
      }
   }
}

//////////////////////////////////////////////////////////////////////////////


VOID ShowModuleInfo(HWND hwnd, LPCTSTR pszModulePath) {

   SetWindowText(hwnd, TEXT(""));   // Clear the output box

   CToolhelp thProcesses(TH32CS_SNAPPROCESS);
   PROCESSENTRY32 pe = { sizeof(pe) };
   BOOL fOk = thProcesses.ProcessFirst(&pe);
   AddText(hwnd, TEXT("Pathname: %s\r\n\r\n"), pszModulePath);
   AddText(hwnd, TEXT("Process Information:\r\n"));
   AddText(hwnd, TEXT("     PID    BaseAddr  Process\r\n"));
   for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {
       CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
       MODULEENTRY32 me = { sizeof(me) };
       BOOL fOk = thModules.ModuleFirst(&me);
       for (; fOk; fOk = thModules.ModuleNext(&me)) {
          if (_tcscmp(me.szExePath, pszModulePath) == 0) {
             AddText(hwnd, TEXT("  %08X  %p  %s\r\n"),
                pe.th32ProcessID, me.modBaseAddr, pe.szExeFile);
          }
       }
    }
}

//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_PROCESSINFO);

   // Hide the module-helper listbox.
   ShowWindow(GetDlgItem(hwnd, IDC_MODULEHELP), SW_HIDE);

   // Have the results window use a fixed-pitch font
```

China-pub.com
下载


```
   SetWindowFont(GetDlgItem(hwnd, IDC_RESULTS),
      GetStockFont(ANSI_FIXED_FONT), FALSE);

   // By default, show the running processes
   Dlg_PopulateProcessList(hwnd);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

   RECT rc;
   int n = LOWORD(GetDialogBaseUnits());

   HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
   GetClientRect(hwndCtl, &rc);
   SetWindowPos(hwndCtl, NULL, n, n, cx - n - n, rc.bottom, SWP_NOZORDER);
   hwndCtl = GetDlgItem(hwnd, IDC_RESULTS);
   SetWindowPos(hwndCtl, NULL, n, n + rc.bottom + n,
      cx - n - n, cy - (n + rc.bottom + n) - n, SWP_NOZORDER);

   return(0);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   static BOOL s_fProcesses = TRUE;

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case ID_PROCESSES:
         s_fProcesses = TRUE;
         EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_ENABLED);
         DrawMenuBar(hwnd);
         Dlg_PopulateProcessList(hwnd);
         break;

      case ID_MODULES:
         EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_GRAYED);
         DrawMenuBar(hwnd);
         s_fProcesses = FALSE;
         Dlg_PopulateModuleList(hwnd);
         break;

      case IDC_PROCESSMODULELIST:
```

China-pub.com 下载

```
         if (codeNotify == CBN_SELCHANGE) {
            DWORD dw = ComboBox_GetCurSel(hwndCtl);
            if (s_fProcesses) {
               dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw); // Process ID
               ShowProcessInfo(GetDlgItem(hwnd, IDC_RESULTS), dw);
            } else {
               // Index in helper listbox of full path
               dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw);
               TCHAR szModulePath[1024];
               ListBox_GetText(GetDlgItem(hwnd, IDC_MODULEHELP),
               dw, szModulePath);
               ShowModuleInfo(GetDlgItem(hwnd, IDC_RESULTS), szModulePath);
            }
         }
         break;

      case ID_VMMAP:
         STARTUPINFO si = { sizeof(si) };
         PROCESS_INFORMATION pi;
         TCHAR szCmdLine[1024];
         HWND hwndCB = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
         DWORD dwProcessId = (DWORD)
            ComboBox_GetItemData(hwndCB, ComboBox_GetCurSel(hwndCB));
         wsprintf(szCmdLine, TEXT("\"14 VMMap\" %d"), dwProcessId);
         BOOL fOk = CreateProcess(NULL, szCmdLine, NULL, NULL,
            FALSE, 0, NULL, NULL, &si, &pi);
         if (fOk) {
            CloseHandle(pi.hProcess);
            CloseHandle(pi.hThread);
         } else {
            chMB("Failed to execute VMMAP.EXE.");
         }
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_SIZE,       Dlg_OnSize);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   CToolhelp::EnableDebugPrivilege(TRUE);
```

China-pub.com
下载


```
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_PROCESSINFO), NULL, Dlg_Proc);
    CToolhelp::EnableDebugPrivilege(FALSE);
    return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

ProcessInfo.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_PROCESSINFO DIALOGEX 0, 0, 400, 317
STYLE DS_3DLOOK | DS_NOFAILCREATE | DS_CENTER | WS_MINIMIZEBOX |
    WS_MAXIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME
EXSTYLE WS_EX_NOPARENTNOTIFY | WS_EX_CLIENTEDGE
CAPTION "Process Information"
MENU IDR_PROCESSINFO
FONT 8, "MS Sans Serif"
BEGIN
    COMBOBOX        IDC_PROCESSMODULELIST,4,4,392,156,CBS_DROPDOWNLIST |
                    CBS_AUTOHSCROLL | CBS_SORT | WS_VSCROLL | WS_TABSTOP
    LISTBOX         IDC_MODULEHELP,0,0,48,40,NOT LBS_NOTIFY | LBS_SORT |
                    LBS_NOINTEGRALHEIGHT | NOT WS_VISIBLE | NOT WS_BORDER |
                    WS_TABSTOP
    EDITTEXT        IDC_RESULTS,4,24,392,284,ES_MULTILINE | ES_AUTOVSCROLL |
                    ES_AUTOHSCROLL | ES_READONLY | WS_VSCROLL | WS_HSCROLL
END


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_PROCESSINFO, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 393
        TOPMARGIN, 7
        BOTTOMMARGIN, 310
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDR_PROCESSINFO MENU DISCARDABLE
BEGIN
    MENUITEM "&Processes!",                 ID_PROCESSES
    MENUITEM "&Modules!",                   ID_MODULES
    MENUITEM "&VMMap!",                     ID_VMMAP
END
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_PROCESSINFO         ICON    DISCARDABLE     "ProcessInfo.ico"
#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Toolhelp.h

```
/*****************************************************************************
Module:  Toolhelp.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tlhelp32.h>
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


class CToolhelp {
private:
   HANDLE m_hSnapshot;

public:
   CToolhelp(DWORD dwFlags = 0, DWORD dwProcessID = 0);
   ~CToolhelp();

   BOOL CreateSnapshot(DWORD dwFlags, DWORD dwProcessID = 0);

   BOOL ProcessFirst(PPROCESSENTRY32 ppe) const;
   BOOL ProcessNext(PPROCESSENTRY32 ppe) const;
   BOOL ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const;

   BOOL ModuleFirst(PMODULEENTRY32 pme) const;
   BOOL ModuleNext(PMODULEENTRY32 pme) const;
```

China-pub.com 下载

```
   BOOL ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const;
   BOOL ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const;
   BOOL ThreadFirst(PTHREADENTRY32 pte) const;
   BOOL ThreadNext(PTHREADENTRY32 pte) const;

   BOOL HeapListFirst(PHEAPLIST32 phl) const;
   BOOL HeapListNext(PHEAPLIST32 phl) const;
   int  HowManyHeaps() const;

   // Note: The heap block functions do not reference a snapshot and
   // just walk the process's heap from the beginning each time. Infinite
   // loops can occur if the target process changes its heap while the
   // functions below are enumerating the blocks in the heap.
   BOOL HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
      UINT_PTR dwHeapID) const;
   BOOL HeapNext(PHEAPENTRY32 phe) const;
   int  HowManyBlocksInHeap(DWORD dwProcessID, DWORD dwHeapId) const;
   BOOL IsAHeap(HANDLE hProcess, PVOID pvBlock, PDWORD pdwFlags) const;

public:
   static BOOL EnableDebugPrivilege(BOOL fEnable = TRUE);
   static BOOL ReadProcessMemory(DWORD dwProcessID, LPCVOID pvBaseAddress,
      PVOID pvBuffer, DWORD cbRead, PDWORD pdwNumberOfBytesRead = NULL);
};


//////////////////////////////////////////////////////////////////////////////


inline CToolhelp::CToolhelp(DWORD dwFlags, DWORD dwProcessID) {

   m_hSnapshot = INVALID_HANDLE_VALUE;
   CreateSnapshot(dwFlags, dwProcessID);
}


//////////////////////////////////////////////////////////////////////////////


inline CToolhelp::~CToolhelp() {

   if (m_hSnapshot != INVALID_HANDLE_VALUE)
      CloseHandle(m_hSnapshot);
}
//////////////////////////////////////////////////////////////////////////////


inline CToolhelp::CreateSnapshot(DWORD dwFlags, DWORD dwProcessID) {

   if (m_hSnapshot != INVALID_HANDLE_VALUE)
      CloseHandle(m_hSnapshot);

   if (dwFlags == 0) {
      m_hSnapshot = INVALID_HANDLE_VALUE;
   } else {
```

China-pub.com
下载


```
      m_hSnapshot = CreateToolhelp32Snapshot(dwFlags, dwProcessID);
   }
   return(m_hSnapshot != INVALID_HANDLE_VALUE);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CToolhelp::EnableDebugPrivilege(BOOL fEnable) {

   // Enabling the debug privilege allows the application to see
   // information about service applications
   BOOL fOk = FALSE;    // Assume function fails
   HANDLE hToken;

   // Try to open this process's access token
   if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
      &hToken)) {

      // Attempt to modify the "Debug" privilege
      TOKEN_PRIVILEGES tp;
      tp.PrivilegeCount = 1;
      LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tp.Privileges[0].Luid);
      tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
      AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
      fOk = (GetLastError() == ERROR_SUCCESS);
      CloseHandle(hToken);
   }
   return(fOk);
}
//////////////////////////////////////////////////////////////////////////////

inline BOOL CToolhelp::ReadProcessMemory(DWORD dwProcessID,
   LPCVOID pvBaseAddress, PVOID pvBuffer, DWORD cbRead,
   PDWORD pdwNumberOfBytesRead) {

   return(Toolhelp32ReadProcessMemory(dwProcessID, pvBaseAddress, pvBuffer,
      cbRead, pdwNumberOfBytesRead));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CToolhelp::ProcessFirst(PPROCESSENTRY32 ppe) const {

   BOOL fOk = Process32First(m_hSnapshot, ppe);
   if (fOk && (ppe->th32ProcessID == 0))
      fOk = ProcessNext(ppe); // Remove the "[System Process]" (PID = 0)
   return(fOk);
}

inline BOOL CToolhelp::ProcessNext(PPROCESSENTRY32 ppe) const {
```

China-pub.com 下载

```
   BOOL fOk = Process32Next(m_hSnapshot, ppe);
   if (fOk && (ppe->th32ProcessID == 0))
      fOk = ProcessNext(ppe); // Remove the "[System Process]" (PID = 0)
   return(fOk);
}


inline BOOL CToolhelp::ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe)
   const {

   BOOL fFound = FALSE;
   for (BOOL fOk = ProcessFirst(ppe); fOk; fOk = ProcessNext(ppe)) {
      fFound = (ppe->th32ProcessID == dwProcessId);
      if (fFound) break;
   }
   return(fFound);
}
//////////////////////////////////////////////////////////////////////////////

inline BOOL CToolhelp::ModuleFirst(PMODULEENTRY32 pme) const {

   return(Module32First(m_hSnapshot, pme));
}

inline BOOL CToolhelp::ModuleNext(PMODULEENTRY32 pme) const {

   return(Module32Next(m_hSnapshot, pme));
}

inline BOOL CToolhelp::ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const {

   BOOL fFound = FALSE;
   for (BOOL fOk = ModuleFirst(pme); fOk; fOk = ModuleNext(pme)) {
      fFound = (pme->modBaseAddr == pvBaseAddr);
      if (fFound) break;
   }
   return(fFound);
}

inline BOOL CToolhelp::ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const {
   BOOL fFound = FALSE;
   for (BOOL fOk = ModuleFirst(pme); fOk; fOk = ModuleNext(pme)) {
      fFound = (lstrcmpi(pme->szModule,  pszModName) == 0) ||
               (lstrcmpi(pme->szExePath, pszModName) == 0);
      if (fFound) break;
   }
   return(fFound);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CToolhelp::ThreadFirst(PTHREADENTRY32 pte) const {

   return(Thread32First(m_hSnapshot, pte));
}
```

China-pub.com
下载


```
inline BOOL CToolhelp::ThreadNext(PTHREADENTRY32 pte) const {

   return(Thread32Next(m_hSnapshot, pte));
}
///////////////////////////////////////////////////////////////////////////////

inline int CToolhelp::HowManyHeaps() const {

   int nHowManyHeaps = 0;
   HEAPLIST32 hl = { sizeof(hl) };
   for (BOOL fOk = HeapListFirst(&hl); fOk; fOk = HeapListNext(&hl))
      nHowManyHeaps++;
   return(nHowManyHeaps);
}

inline int CToolhelp::HowManyBlocksInHeap(DWORD dwProcessID,
   DWORD dwHeapID) const {

   int nHowManyBlocksInHeap = 0;
   HEAPENTRY32 he = { sizeof(he) };
   BOOL fOk = HeapFirst(&he, dwProcessID, dwHeapID);
   for (; fOk; fOk = HeapNext(&he))
      nHowManyBlocksInHeap++;
   return(nHowManyBlocksInHeap);
}

inline BOOL CToolhelp::HeapListFirst(PHEAPLIST32 phl) const {

   return(Heap32ListFirst(m_hSnapshot, phl));
}

inline BOOL CToolhelp::HeapListNext(PHEAPLIST32 phl) const {

   return(Heap32ListNext(m_hSnapshot, phl));
}

inline BOOL CToolhelp::HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
   UINT_PTR dwHeapID) const {

   return(Heap32First(phe, dwProcessID, dwHeapID));
}

inline BOOL CToolhelp::HeapNext(PHEAPENTRY32 phe) const {

   return(Heap32Next(phe));
}
inline BOOL CToolhelp::IsAHeap(HANDLE hProcess, PVOID pvBlock,
   PDWORD pdwFlags) const {

   HEAPLIST32 hl = { sizeof(hl) };
   for (BOOL fOkHL = HeapListFirst(&hl); fOkHL; fOkHL = HeapListNext(&hl)) {
      HEAPENTRY32 he = { sizeof(he) };
      BOOL fOkHE = HeapFirst(&he, hl.th32ProcessID, hl.th32HeapID);
      for (; fOkHE; fOkHE = HeapNext(&he)) {
```

China-pub.com 下载

```
            MEMORY_BASIC_INFORMATION mbi;
            VirtualQueryEx(hProcess, (PVOID) he.dwAddress, &mbi, sizeof(mbi));
            if (chINRANGE(mbi.AllocationBase, pvBlock,
               (PBYTE) mbi.AllocationBase + mbi.RegionSize)) {

               *pdwFlags = hl.dwFlags;
               return(TRUE);
            }
         }
      }
      return(FALSE);
}


//////////////////////////////// End of File //////////////////////////////////
```

China-pub.com
下载

第5章 作 业

通常，必须将一组进程当作单个实体来处理。例如，当让 Microsoft Developer Studio为你创建一个应用程序项目时，它会生成 Cl.exe，Cl.exe则必须生成其他的进程（比如编译器的各个函数传递）。如果用户想要永远停止该应用程序的创建，那么 Developer Studio必须能够终止 Cl.exe和它的所有子进程的运行。在 Windows中解决这个简单（和常见的）的问题是极其困难的，因为Windows并不维护进程之间的父/子关系。即使父进程已经终止运行，子进程仍然会继续运行。

当设计一个服务器时，也必须将一组进程作为单个进程组来处理。例如，客户机可能要求服务器执行一个应用程序（这可以生成它自己的子应用程序），并给客户机返回其结果。由于可能有许多客户机与该服务器相连接，如果服务器能够限制客户机的要求，即用什么手段来防止任何一个客户机垄断它的所有资源，那么这是非常有用的。这些限制包括：可以分配给客户机请求的最大CPU时间，最小和最大的工作区的大小，防止客户机的应用程序关闭计算机，以及安全性限制等。

Microsoft Windoss 2000提供了一个新的作业内核对象，使你能够将进程组合在一起，并且创建一个“沙框”，以便限制进程能够进行的操作。最好将作业对象视为一个进程的容器。但是，创建包含单个进程的作业是有用的，因为这样一来，就可以对该进程加上通常情况下不能加的限制。

我的StartRestrictedProcess函数（见清单5-1）将一个进程放入一个作业，以限制该进程进行某些操作的能力。

Windows 98 Windows 98不支持作业的操作。

清单5-1 StartRestrictedProcess函数

```
void StartRestrictedProcess() {
   // Create a job kernel object.
   HANDLE hjob = CreateJobObject(NULL, NULL);

   // Place some restrictions on processes in the job.

   // First, set some basic restrictions.
   JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };

   // The process always runs in the idle priority class.
   jobli.PriorityClass = IDLE_PRIORITY_CLASS;

   // The job cannot use more than 1 second of CPU time.
   jobli.PerJobUserTimeLimit.QuadPart = 10000000; // 1 sec in 100-ns intervals

   // These are the only 2 restrictions I want placed on the job (process).
   jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
      | JOB_OBJECT_LIMIT_JOB_TIME;
   SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
      sizeof(jobli));
```

China-pub.com 下载

```
   // Second, set some UI restrictions.
   JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
   jobuir.UIRestrictionsClass  = JOB_OBJECT_UILIMIT_NONE;      // A fancy zero

   // The process can't log off the system.
   jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

   // The process can't access USER objects (such as other windows)
   // in the system.
   jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_HANDLES;

   SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
      sizeof(jobuir));

   // Spawn the process that is to be in the job.
   // Note: You must first spawn the process and then place the process in
   //       the job. This means that the process's thread must be initially
   //       suspended so that it can't execute any code outside of the job's
   //       restrictions.
   STARTUPINFO si = { sizeof(si) };
   PROCESS_INFORMATION pi;
   CreateProcess(NULL, "CMD", NULL, NULL, FALSE,
      CREATE_SUSPENDED, NULL, NULL, &si, &pi);
   // Place the process in the job.
   // Note: If this process spawns any children, the children are
   //       automatically part of the same job.
   AssignProcessToJobObject(hjob, pi.hProcess);

   // Now we can allow the child process's thread to execute code.
   ResumeThread(pi.hThread);
   CloseHandle(pi.hThread);

   // Wait for the process to terminate or
   // for all the job's allotted CPU time to be used.
   HANDLE h[2];
   h[0] = pi.hProcess;
   h[1] = hjob;
   DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
   switch (dw - WAIT_OBJECT_0) {
      case 0:
         // The process has terminated...
         break;
      case 1:
         // All of the job's allotted CPU time was used...
         break;
   }

   // Clean up properly.
   CloseHandle(pi.hProcess);
   CloseHandle(hjob);
}
```

现在，解释一下StartRestrictedProcess函数是如何工作的。首先，调用下面的代码，创建一个新作业内核对象：

China-pub.com
下载


```
HANDLE CreateJobObject(
   PSECURITY_ATTRIBUTES psa,
   PCTSTR pszName);
```

与所有的内核对象一样，它的第一个参数将安全信息与新作业对象关联起来，并且告诉系统，是否想要使返回的句柄成为可继承的句柄。最后一个参数用于给作业对象命名，使它可以供另一个进程通过下面所示的OpenJobObject函数进行访问。

```
HANDLE OpenJobObject(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

与平常一样，如果知道你将不再需要访问代码中的作业对象，那么就必须通过调用CloseHandle来关闭它的句柄。可以在我的StartRestrictedProcess函数的结尾处看到这个代码的情况。应该知道，关闭作业对象并不会迫使作业中的所有进程终止运行。该作业对象实际上做上了删除标记，只有当作业中的所有进程全部终止运行之后，该作业对象才被自动撤消。

注意，关闭作业的句柄后，尽管该作业仍然存在，但是该作业将无法被所有进程访问。请看下面的代码：

```
// Create a named job object.
HANDLE hjob = CreateJobObject(NULL, TEXT("Jeff"));

// Put our own process in the job.
AssignProcessToJobObject(hjob, GetCurrentProcess());

// Closing the job does not kill our process or the job.
// But the name ("Jeff") is immediately disassociated with the job.
CloseHandle(hjob);

// Try to open the existing job.
hjob = OpenJobObject(JOB_OBJECT_ALL_ACCESS, FALSE, TEXT("Jeff"));
// OpenJobObject fails and returns NULL here because the name ("Jeff")
// was disassociated from the job when CloseHandle was called.
// There is no way to get a handle to this job now.
```

5.1 对作业进程的限制

进程创建后，通常需要设置一个沙框（设置一些限制），以便限制作业中的进程能够进行的操作。可以给一个作业加上若干不同类型的限制：

- 基本限制和扩展基本限制，用于防止作业中的进程垄断系统的资源。
- 基本的UI限制，用于防止作业中的进程改变用户界面。
- 安全性限制，用于防止作业中的进程访问保密资源（文件、注册表子关键字等）。

通过调用下面的代码，可以给作业加上各种限制：

```
BOOL SetInformationJobObject(
   HANDLE hJob,
   JOBOBJECTINFOCLASS JobObjectInformationClass,
   PVOID pJobObjectInformation,
   DWORD cbJobObjectInformationLength);
```

第一个参数用于标识要限制的作业。第二个参数是个枚举类型，用于指明要使用的限制类型。第三个参数是包含限制设置值的数据结构的地址，第四个参数用于指明该结构的大小（用于确定版本）。表5-1列出了如何来设置各种限制条件。

China-pub.com 下载

表5-1 设置限制条件

限制类型	第二个参数的值	第三个参数的结构
基本限制	JobObjectBasicLimitInformation	JOBOBJECT_BASIC_LIMIT_INFORMATION
扩展基本限制	JobObjectExtendedLimitInformation	JOBOBJECT_EXTENDED_LIMIT_INFORMATION
基本UI限制	JobObjectBasicUIRestrictions	JOBOBJECT_BASIC_UI_RESTRICTIONS
安全性限制	JobObjectSecurityLimitInformation	JOBOBJECT_SECURITY_LIMIT_INFORMATION

在StartRestrictedProcess函数中，我只对作业设置了一些最基本的限制。指定了一个JOB_OBJECT_BASIC_LIMIT_INFORMATION结构，对它进行了初始化，然后调用SetInformationJobObject函数。JOB_OBJECT_BASIC_LIMIT_INFORMATION结构类似下面的样子：

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD         LimitFlags;
    DWORD         MinimumWorkingSetSize;
    DWORD         MaximumWorkingSetSize;
    DWORD         ActiveProcessLimit;
    DWORD_PTR     Affinity;
    DWORD         PriorityClass;
    DWORD         SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

表5-2简单地描述了它的各个成员的情况。

表5-2 JOB_OBJECT_BASIC_LIMIT_INFORMATION结构的成员

成员	描述	说明
PerProcessUserTimeLimit	设定分配给每个进程的用户方式的最大时间（以100ns为间隔时间）	任何进程占用的时间如果超过了分配给它的时间，系统将自动终止它的运行。若要设置这个限制条件，请在LimitFlags成员中设定JOB_OBJECT_LIMIT_PROCESS_TIME
PerJobUserTimeLimit	设定该作业中可以使用多少用户方式的时间(以100ns为间隔时间)	按照默认设置，当达到该时间限制时，系统将自动终止所有进程的运行。可以在作业运行时定期改变这个值。若要设置该限制条件，请在LimitFlags成员中设定JOB_OBJECT_ LIMIT_JOB_TIME
LimitFlags	指明哪些限制适用于该作业	详细说明参见本表下面的一段
MinimumWorkingSetSize/MaximumWorking SetSize	设定每个进程（不是作业中的所有进程）的最小和最大工作区的大小	通常，进程的工作区可能扩大而超过它的最小值。设置MaximumWorkingSetSize后，就可以实施硬限制。一旦进程的工作区达到该限制值，进程就会对此作出页标记。各个进程对SetProcessWorkingSetSize的调用将被忽略，除非该进程只是试图清空它的工作区。若要设置该限制，请在LimitFlags成员中设定JOB_OBJECT_LIMIT_WORKINGSET标志
ActiveProcessLimit	设定作业中可以同时运行的进	超过这个限制的任何尝试都会导致新进程被迫终止

China-pub.com
下载

（续）

成 员	描 述	说 明
	程的最大数量	运行，并产生一个“定额不足”的错误。若要设置这个限制，请在 LimitFlags成员中设定 JOB_OBJECT_LIMIT_ACTIVE_PROCESS
Affinity	设定能够运行的进程的CPU子集	单个进程甚至能够进一步对此作出限制。若要设置这个限制，请在 LimitFlags成员中设定 JOB_OBJECT_LIMIT_AFFINITY
PriorityClass	设定所有进程使用的优先级	如果进程调用SetPriorityClass函数，即使该函数调用失败，它也能成功地返回。如果进程调用GetPriorityClass函数，该函数将返回进程已经设置的优先级类，尽管这可能不是进程的实际优先级类。此外，SetThreadPriority无法将线程的优先级提高到正常的优先级之上，不过它可以用于降低线程的优先级。若要设置这个限制，请在 LimitFlags成员中设定JOB_OBJECT_LIMIT_PRIORITY_CLASS
SchedulingClass	设定分配给作业中的线程的相对时段差	它的值可以在0到9之间（包括0和9），默认值是5。详细说明参见本表后面的文字。若要设置这个限制，请在 LimitFlags成员中设定 JOB_OBJECT_LIMIT_SCHEDULING_CLASS

关于这个结构的某些问题在Platform SDK文档中并没有说清楚，因此在这里作一些说明。你在LimitFlags成员中设置了一些信息，来指明想用于作业的限制条件。我设置了JOB_OBJECT_LIMIT_PRIORITY_CLASS和JOB_OBJECT_LIMIT_JOB_TIME这两个标志。这意味着它们是我用于该作业的唯一的两个限制条件。我没有对CPU的亲缘关系、工作区的大小、每个进程占用的CPU时间等作出限制。

当作业运行时，它会维护一些统计信息，比如作业中的进程已经使用了多少CPU时间。每次使用JOB_OBJECT_LIMIT_JOB_TIME标志来设置基本限制时，作业就会减去已经终止运行的进程的CPU时间的统计信息。这显示当前活动的进程使用了多少CPU时间。如果想改变作业运行所在的CPU的亲缘关系，但是没有重置CPU时间的统计信息，那将要如何处理呢？为了处理这种情况，必须使用JOB_OBJECT_LIMIT_AFFINITY 标志来设置新的基本限制条件，并且必须退出JOB_OBJECT_LIMIT_JOB_TIME标志的设置。这样一来，就告诉作业，不再想要使用CPU的时间限制。这不是你想要的。

你想要的是改变CPU亲缘关系的限制，保留现有的CPU时间限制。你只是不想减去已终止运行的进程的 CPU时间的统计信息。为了解决这个问题，可以使用一个特殊标志，即JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME。这个标志与JOB_OBJECT_LIMIT_JOB_TIME标志是互斥的。JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME标志表示你想改变限制条件，而不减去已经终止运行的进程的CPU时间的统计信息。

现在介绍一下JOBOBJECT_BASIC_LIMIT_INFORMATION结构的SchedulingClass成员。假如你有两个正在运行的作业，你将两个作业的优先级类都设置为 NORMAL_PRIORITY_CLASS。但是你还想让一个作业中的进程获得比另一个进程多的 CPU时间。可以使用SchedulingClass成员来改变拥有相同优先级的作业的相对调度关系。可以设置一个0至9之间的值（包括0和9），5是默认值。在Windows 2000上，如果这个设置值比较大，那么系统就会给某个作业的进程中的线程提供较长的CPU时间量。如果设置的值比较小，就减少该线程的CPU时间量。

China-pub.com 下载

例如，我有两个拥有正常优先级类的作业。每个作业包含一个进程，每个进程只有一个（拥有正常优先级的）线程。在正常环境下，这两个线程将按循环方式进行调度，每个线程获得相同的CPU时间量。但是，如果将第一个作业的 SchedulingClass成员设置为3，那么，当该作业中的线程被安排CPU时间时，它得到的时间量将比第二个作业中的线程少。

如果使用SchedulingClass成员，应该避免使用大数字即较大的时间量，因为较大的时间量会降低系统中的其他作业、进程和线程的总体响应能力。另外，我只是介绍了 Windows 2000中的情况。Microsoft计划在将来的Windows版本中对线程调度程序进行更重要的修改，因为它认为操作系统应该为作业、进程和线程提供更宽松的线程调度环境。

需要特别注意的最后一个限制是 JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION限制标志。这个限制可使系统为与作业相关的每个进程关闭“未处理的异常情况”对话框。系统通过调用SetErrorMode函数，将作业中的每个进程的SEM_NOGPFAULTERRORBOX标志传递给它。作业中产生未处理的异常情况的进程会立即终止运行，不显示任何用户界面。对于服务程序和其他面向批处理的作业来说，这是个非常有用的限制标志。如果没有这个标志，作业中的进程就会产生一个异常情况，并且永远不会终止运行，从而浪费了系统资源。

除了基本限制外，还可以使用JOBOBJECT_EXTENDED_LIMIT_INFORMATION结构对作业设置扩展限制：

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
   JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
   IO_COUNTERS IoInfo;
   SIZE_T ProcessMemoryLimit;
   SIZE_T JobMemoryLimit;
   SIZE_T PeakProcessMemoryUsed;
   SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

如你所见，该结构包含一个JOBOBJECT_BASIC_LIMIT_INFORMATION结构，它构成了基本限制的一个超集。这个结构有一点儿特殊，因为它包含的成员与设置作业的限制毫无关系。首先，IoInfo成员保留不用，无论如何不能访问它。本章后面将要介绍如何查询I/O计数器信息。此外，PackProcessMemoryUsed和PackJobMemoryUsed成员是只读成员，分别告诉你作业中的任何一个进程和所有进程需要使用的已确认的内存最大值。

另外两个成员ProcessMemoryLimit和JobMemoryLimit分别用于限制作业中的任何一个进程和所有进程使用的已确认的内存量。若要设置这些限制值，可以在 LimitFlags成员中分别设定JOB_OBJECT_LIMIT_JOB_MEMORY和JOB_OBJECT_LIMIT_PROCESS_MEMORY两个标志。

现在看一下可以对作业设置的另一些限制。下面是JOBOBJECT_BASIC_UI_RESTRICTIONS结构的样子：

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
   DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

这个结构只有一个数据成员，即 UIRestrictionsClass，它用于存放表5-3中简单描述的一组位标志。

最后一个标志JOB_OBJECT_UILIMIT_HANDLES是特别有趣的。这个限制意味着作业中没有一个进程能够访问该作业外部的进程创建的 USER对象。因此，如果试图在作业内部运行Microsoft Spy++，那么除了Spy++自己创建的窗口外，你看不到任何别的窗口。图 5-1显示的

China-pub.com
下载

Spy++中打开了两个MDI子窗口。注意，Threads 1的窗口包含一个系统中的线程列表。这些线程中只有一个线程，即000006AC SPYXX似乎创建了一些窗口。这是因为我是在它自己的作业中运行Spy++的，并且限制了它对 UI句柄的使用。在同一个窗口中，可以看到 MSDEV和EXPLORER两个线程，但是看来它们尚未创建任何窗口。可以保证，这些线程肯定创建了窗口，但是Spy++无法访问它们。在对话框的右边，可以看到 Windows 3窗口，在这个窗口中，Spy++显示了桌面上存在的所有窗口的层次结构。注意，它只有一个项目，即 00000000。Spy++必须将它作为占位符放在这里。

表5-3 用于作业对象的基本用户界面限制的位标志

标 志	描 述
JOB_OBJECT_UILIMIT_EXITWINDOWS	用于防止进程通过ExitWindowsEx函数退出、关闭、重新引导或关闭系统电源
JOB_OBJECT_UILIMIT_READCLIPBOARD	防止进程读取剪贴板的内容
JOB_OBJECT_UILIMIT_WRITECLIPBOARD	防止进程删除剪贴板的内容
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	防止进程通过SystemParametersInfor函数来改变系统参数
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	防止进程通过ChangeDisplaySettings函数来改变显示设置
JOB_OBJECT_UILIMIT_GLOBALATOMS	为作业赋予它自己的基本结构表，使作业中的进程只能访问该作业的表
JOB_OBJECT_UILIMIT_DESKTOP	防止进程使用CreateDesktop或SwitchDesktop函数创建或转换桌面
JOB_OBJECT_UILIMIT_HANDLES	防止作业中的进程使用同一作业外部的进程创建的USER对象（如HWND）

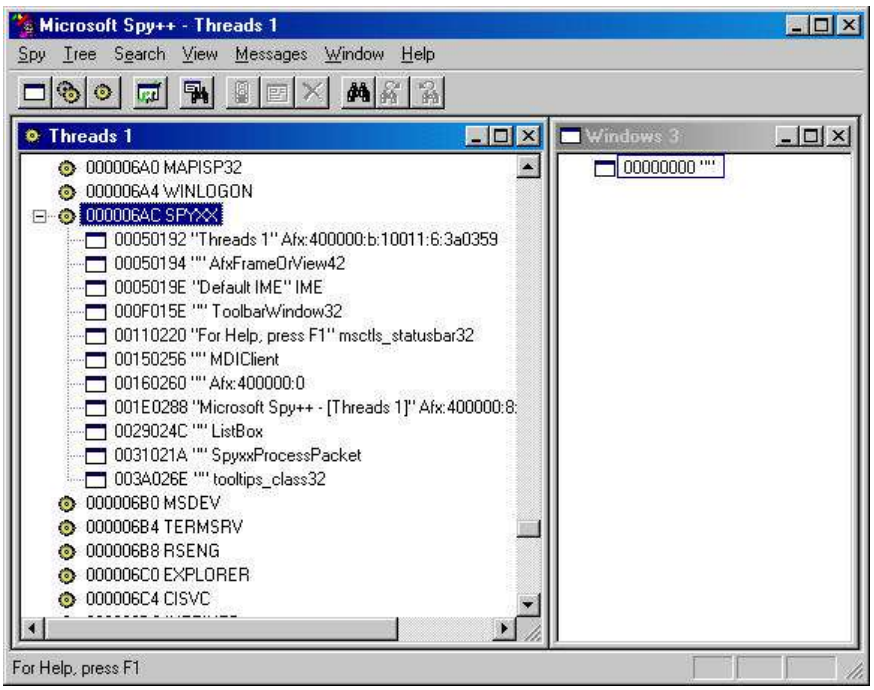


图5-1 在作业中运行的Microsoft Spy++可以限制对UI句柄的访问

注意，这个UI限制是单向的。这就是说，作业外部的进程能够看到作业内部的进程创建的USDR对象。例如，如果我在一个作业中运行 Notepad，并在作业的外部运行Spy++，那么，如

China-pub.com 下载

果Notepad所在的作业设定了JOB_OBJECT_UILIMIT_HANDLES标志，Spy++将能够看到Notepad的窗口。同样，如果Spy++在它自己的作业中，那么它也可以看到Notepad的窗口，只要它设定了JOB_OBJECT_UILIMIT_HANDLES标志。

如果想为作业进程的操作创建真正的沙框，那么限制UI句柄是可怕的。但是，如果作为作业组成部分的一个进程要与作业外部的进程进行通信，就可以使用这种限制。

实现这个目的有一个简便的方法，那就是使用窗口消息，但是，如果作业的进程不能访问UI句柄，那么作业中的进程就无法将窗口消息发送或显示在作业外部的进程创建的窗口中。不过，可以使用下面这个新函数来解决这个问题：

```
BOOL UserHandleGrantAccess(
   HANDLE hUserObj,
   HANDLE hJob,
   BOOL fGrant);
```

hUserObj参数用于指明一个USER对象，可以为作业中的进程赋予或者撤消对该对象的访问权。它几乎总是一个窗口句柄，但是它可以是另一个USER对象，比如桌面、挂钩、图标或菜单。最后两个参数hjob和fGrant用于指明你赋予或撤消对哪个作业的访问权。注意，如果从hjob标识的作业中的一个进程来调用该函数，该函数的运行就会失败——这可以防止作业中的进程总是为它自己赋予访问一个对象的权限。

对作业施加的最后一种限制类型与安全性相关（注意，一旦使用这种限制，就无法取消安全性限制）。JOBOBJECT_SECURITY_LIMIT_INFORMATION的结构类似下面的形式：

```
typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
   DWORD SecurityLimitFlags;
   HANDLE JobToken;
   PTOKEN_GROUPS SidsToDisable;
   PTOKEN_PRIVILEGES PrivilegesToDelete;
   PTOKEN_GROUPS RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION, *PJOBOBJECT_SECURITY_LIMIT_INFORMATION;
```

表5-4简单地描述了它的各个成员。

表5-4 JOBOBJECT_SECURITY_LIMIT_INFORMATIDN 的成员

成 员	描 述
SecurityLimitFlags	指明是否不允许管理员访问、不允许无限制的标记访问、强制使用特定的访问标记，或者停用某些安全性标识符和优先权
JobToken	作业中的所有进程使用的访问标记
SidsToDisable	指明为访问检查停用哪些SID
PrivilegesToDelete	指明要从访问标记中删除哪些优先权
RestrictedSids	指明应该添加给访问标记的一组仅为拒绝(deny only)的SID

当然，一旦给作业设置了限制条件，就可以查询这些限制。通过调用下面的代码，就可以进行这一操作

```
BOOL QueryInformationJobObject(
   HANDLE hJob,
   JOBOBJECTINFOCLASS JobObjectInformationClass,
   PVOID pvJobObjectInformation,
   DWORD cbJobObjectInformationLength,
   PDWORD pdwReturnLength);
```

你为该函数传递作业的句柄（就像你对SetInformationJobObject操作时那样），这些句柄包

China-pub.com
下载

括用于指明你想要的限制信息的枚举类型，函数要进行初始化的数据结构的地址，以及包含该结构的数据块的长度。最后一个参数是 pdwReturnLength，用于指向该函数填写的 DWORD，它告诉你有多少字节放入了缓存。如果你愿意的话，可以（并且通常）为该参数传递 NULL。

> **注意** 作业中的进程可以调用QueryInformationJobObject，以便通过为作业的句柄参数传递NULL，获取关于该进程所属的作业的信息。由于它使进程能够看到已经对它实施了哪些限制，所以这个函数非常有用。但是，如果为作业句柄参数传递 NULL，那么SetInformationJobObject函数运行就会失败，因为这将允许进程删除对它实施的限制。

5.2 将进程放入作业

上面介绍的是设置和查询限制方面的信息。现在回到 StartRestrictedProcess这个函数的操作上来。当对作业实施一些限制之后，通过调用 CreateProcess，生成了一个进程，我想将它放入作业。但是，注意，当调用 CreateProcess时，我使用了 CREATE_SUSPENDED标志。这样，创建了一个新进程，但是不允许它执行任何代码。由于 Start-ReatrictedProcess函数是从不属于作业组成部分的进程来执行的，因此子进程也不属于作业的组成部分。如果准备立即允许子进程开始执行代码，那么它将跑出我的沙框，并且能够成功地执行我想限制它做的工作。因此，当创建子进程之后，在我允许它开始运行之前，我必须显式地将该进程放入我新创建的作业，方法是调用下面的代码：

```
BOOL AssignProcessToJobObject(
   HANDLE hJob,
   HANDLE hProcess);
```

该函数告诉系统，将该进程（由 hProcess标识）视为现有作业（由 hJob标识）的一部分。注意，该函数只允许将尚未被赋予任何作业的进程赋予一个作业。一旦进程成为一个作业的组成部分，它就不能转到另一个作业，并且不能是无作业的进程。另外，当作为作业的一部分的进程生成另一个进程的时候，新进程将自动成为父作业的组成部分。不过可以用下面的方法改变它的行为特性：

- 打开JOBOBJECT_BASIC_LIMIT_INFORMATION的LimitFlags成员中的JOB_OBJECT_BREAKAWAY_OK标志，告诉系统，新生成的进程可以在作业外部运行。若要做到这一点，必须用新的CREATE_BREAKAWAY_FROM_JOB标志来调用CreateProcess。如果用CREATE_BREAKAWAY_FROM_JOB标志调用CreateProcess函数，但是该作业并没有打开CREATE_BREAKAWAY_FROM_JOB这个标志，那么CreateProcess函数运行就会失败。如果新生成的进程也能控制作业，那么这个机制是有用的。
- 打开JOBOBJECT_BASIC_LIMIT_INFORMATION的LimitFlags成员中的JOB_OBJECT_SILENT_BREAKAWAY_OK标志。该标志也告诉系统，新生成的进程不应该是作业的组成部分。但是没有必要将任何其他标志传递给 CreateProcess。实际上，该标志将使新进程不能成为作业的组成部分。该标志可以用于原先对作业对象一无所知的进程。

至于StartRestrictedProcess函数，当调用AssignProcessToJobObject后，新进程就成为受限制的作业的组成部分。然后调用ResumeThread，这样，进程的线程就可以在作业的限制下执行代码。这时，也可以关闭线程的句柄，因为不再需要它了。

5.3 终止作业中所有进程的运行

当然，想对作业进行的最经常的操作是撤消作业中的所有进程。本章开头讲过， Developer

China-pub.com 下载

Studio没有配备任何便于使用的方法，来停止进程中的某个操作，因为它不知道哪个进程是由第一个进程生成的（这非常复杂。我在Microsoft Systems Journal期刊1998年9月号上Win32 问与答栏中介绍了Developer Studio是如何做到这一点的）。我认为，Developer Studio的将来版本将会改用作业来进行操作，因为代码的编写要容易得多，可以用它做更多的工作。

若要撤消作业中的进程，只需要调用下面的代码：

```
BOOL TerminateJobObject(
   HANDLE hJob,
   UINT uExitCode);
```

这类似为作业中的每个进程调用 TerminateProcess函数，将它们的所有退出代码设置为uExitCode。

5.4 查询作业统计信息

前面已经介绍了如何使用QueryInformationJobObject函数来获取对作业的当前限制信息。也可以使用它来获取关于作业的统计信息。例如，若要获取基本的统计信息，可以调用QueryInformationJobObject，为第二个参数传递JobObjectBasicAccountingInformation,并传递JOBOBJECT_BASIC_ACCOUNTING_INFORMATION结构的地址：

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
   LARGE_INTEGER TotalUserTime;
   LARGE_INTEGER TotalKernelTime;
   LARGE_INTEGER ThisPeriodTotalUserTime;
   LARGE_INTEGER ThisPeriodTotalKernelTime;
   DWORD TotalPageFaultCount;
   DWORD TotalProcesses;
   DWORD ActiveProcesses;
   DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
   *PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

表5-5简要描述了它的各个成员。

表5-5 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION的成员

成 员	描 述
TotalUserTime	设定作业中的进程已经使用多少用户方式CPU时间
TotalKernelTime	设定作业中的进程已经使用多少内核方式CPU时间
ThisPeriodTotalUserTime	与TotalUserTime的作用相同，差别是，当调用SetInformation-JobObject以便改变基本限制信息并且不使用 JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME限制标记时，本值复置为0
ThisPeriodTotalKernelTime	与ThisPeriodTotalUserTime相同，差别是，本值显示的是内核方式时间
TotalPageFaultCount	设定作业中的进程已经产生的页面故障数量
TotalProcesses	设定曾经成为作业组成部分的进程总数
ActiveProcesses	设定当前作为作业的组成部分的进程的数量
TotalTerminatedProcesses	设定由于超过分配给它们的CPU时间限制而被撤消的进程的数量

除了查询这些基本统计信息外，可以进行一次函数调用，以同时查询基本统计信息和 I/O统计信息。为此，必须为第二个参数传递 JobObjectBasicAndIoAccountingInformation,并传递JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION结构的地址：

```
typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
   JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
```

China-pub.com
下载


```
   IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

如你所见，这个结构只返回一个JOBOBJECT_BASIC_ACCOUNTING_INFORMATION结构和IO_COUNTERS结构：

```
typedef struct _IO_COUNTERS {
   ULONGLONG ReadOperationCount;
   ULONGLONG WriteOperationCount;
   ULONGLONG OtherOperationCount;
   ULONGLONG ReadTransferCount;
   ULONGLONG WriteTransferCount;
   ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

该结构告诉你作业中的进程已经执行的读、写和非读/写操作的数量（以及在这些操作期间传送的字节数）。另外，可以使用下面这个新的GetProcessIoCounters函数，以便获取不是这些作业中的进程的这些信息：

```
BOOL GetProcessIoCounters(
   HANDLE hProcess,
   PIO_COUNTERS pIoCounters);
```

也可以随时调用QueryInformationJobObject函数，以便获取当前在作业中运行的进程的一组进程ID。若要进行这项操作，首先必须确定你想在作业中看到多少进程，然后必须分配足够的内存块，来放置这些进程ID的数组，并指定JOBOBJECT_BASIC_PROCESS_ID_LIST结构的大小：

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
   DWORD NumberOfAssignedProcesses;
   DWORD NumberOfProcessIdsInList;
   DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

因此，若要获得当前作业中的一组进程ID，必须执行类似下面的代码：

```
void EnumProcessIdsInJob(HANDLE hjob) {

   // I assume that there will never be more
   // than 10 processes in this job.
   #define MAX_PROCESS_IDS      10

   // Calculate the number of bytes needed for structure & process IDs.
   DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
      (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

   // Allocate the block of memory.
   PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil = _alloca(cb);

   // Tell the function the maximum number of processes
   // that we allocated space for.
   pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;

   // Request the current set of process IDs.
   QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
      pjobpil, cb, &cb);

   // Enumerate the process IDs.
```

China-pub.com 下载

```
   for (int x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {
      // Use pjobpil->ProcessIdList[x]...
   }

   // Since _alloca was used to allocate the memory,
   // we don't need to free it here.
}
```

这就是你使用这些函数的所有实现方法，不过操作系统实际上保存了更多的关于作业的信息。它是使用性能计数器来进行这项操作的。可以使用 Performance Data Helper函数库（PDH.dll）中的函数来检索这些信息。也可以使用 Microsoft Management Console(MMC)的Performance Monitor Snap-In来查看作业信息。图5-2显示了系统中的作业对象可以使用的一些计数器。图5-3显示了可以使用的一些作业对象的明细计数器。也可以看到 Jeff的作业里有4个进程，即calc、cmd、notepad和workpad。

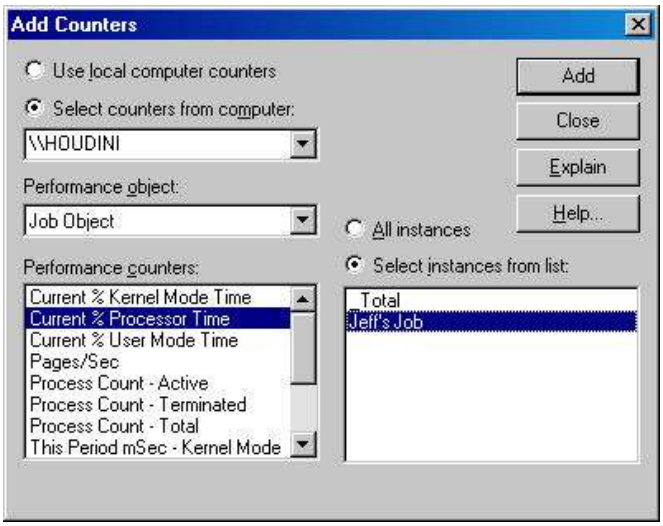


图5-2 MMC的性能监控器：作业对象计数器

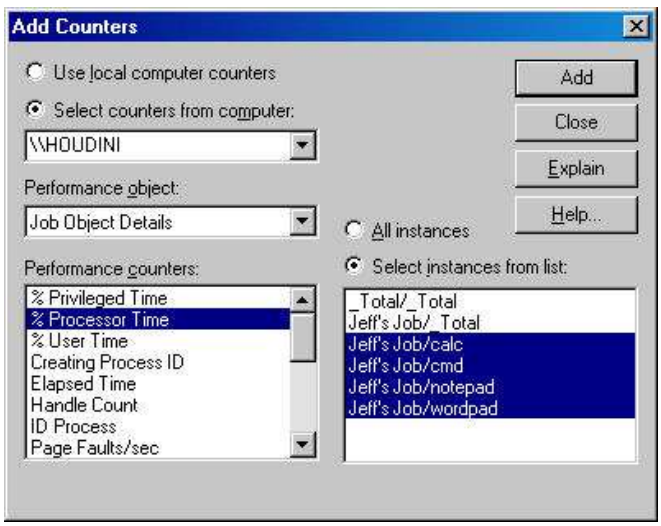


图5-3 MMC的性能监控器：作业对象明细计数器

China-pub.com
下载

注意，当调用CreateJobObject函数时，只能为已经赋予名字的作业获取性能计数器信息。由于这个原因，即使不打算按名字来共享跨越进程的作业对象，也应该创建带有名字的这些对象。

5.5 作业通知信息

现在，已经知道了关于作业对象的基本知识，剩下要介绍的内容是关于通知的问题。例如，是否想知道作业中的所有进程何时终止运行或者分配的全部CPU时间是否已经到期呢？也许想知道作业中何时生成新进程或者作业中的进程何时终止运行。如果不关心这些通知信息（而且许多应用程序也不关心这些信息），作业的操作非常容易。如果关心这些事件，那么还有一些工作要做。

如果关心的是分配的所有CPU时间是否已经到期，那么可以非常容易地得到这个通知信息。当作业中的进程尚未用完分配的CPU时间时，作业对象就得不到通知。一旦分配的所有CPU时间已经用完，Windows就强制撤消作业中的所有进程，并将情况通知作业对象。通过调用WaitForSingleObject(或类似的函数)，可以很容易跟踪这个事件。有时，可以在晚些时候调用SetInformationJobObject函数，使作业对象恢复未通知状态，并为作业赋予更多的CPU时间。

当开始对作业进行操作时，我觉得当作业中没有任何进程运行时，应该将这个事件通知作业对象。毕竟当进程和线程停止运行时，进程和线程对象就会得到通知。因此，当作业停止运行时它也应该得到通知。这样，就能够很容易确定作业何时结束运行。但是，Microsoft选择在分配的CPU时间到期时才向作业发出通知，因为这显示了一个错误条件。由于许多作业启动时有一个父进程始终处于工作状态，直到它的所有子进程运行结束，因此只需要在父进程的句柄上等待，就可以了解整个作业何时运行结束。StartRestrictedProcess函数用于显示分配给作业的CPU时间何时到期，或者作业中的进程何时终止运行。

前面介绍了如何获得某些简单的通知信息，但是尚未说明如何获得更高级的通知信息，如进程创建/终止运行等。如果想要得到这些通知信息，必须将更多的基础结构放入应用程序。特别是，必须创建一个I/O完成端口内核对象，并将作业对象或多个作业对象与完成端口关联起来。然后，必须让一个或多个线程在完成端口上等待作业通知的到来，这样它们才能得到处理。

一旦创建了I/O完成端口，通过调用SetInformationJobObject函数，就可以将作业与该端口关联起来，如下面的代码所示：

```
JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp;
joacp.CompletionKey  = 1;   // Any value to uniquely identify this job
joacp.CompletionPort = hIOCP;   // Handle of completion port that
                                // receives notifications
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,
   &joacp,  sizeof(joacp));
```

当上面的代码运行时，系统将监视该作业的运行，当事件发生时，它将事件送往I/O完成端口（顺便说一下，可以调用QueryInformatiomJobObject函数来检索完成关键字和完成端口句柄。但是，这样做的机会很少）。线程通过调用GetQueuedCompletionStatus函数来监控I/O完成端口：

```
BOOL GetQueuedCompletionStatus(
   HANDLE hIOCP,
   PDWORD pNumBytesTransferred,
```

China-pub.com 下载

```
    PULONG_PTR pCompletionKey,
    POVERLAPPED *pOverlapped,
    DWORD dwMilliseconds);
```

当该函数返回一个作业事件通知时，*pCompletionKey包含了调用SetInformationJobObject时设置的完成关键字值，用于将作业与完成端口关联起来。它使你能够知道哪个作业存在一个事件。*pNumBytesTransferred中的值用于指明发生了哪个事件。根据事件（见表5-6），*pOverlapped 中的值将指明一个进程ID。

表5-6 系统可以发送给作业的相关完成端口的作业事件通知

事 件	描 述
JOB_OBJECT_MSG_ACTIVE PROCESS_ZERO	当作业中没有进程运行时发送
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	当超过分配给进程的 CPU时间时发送。进程终止运行，并赋予进程的ID
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	当试图超过作业中运行的进程数量时发送
JOB_OBJECT_MSG_PROCESS MEMORY_LIMIT	当进程试图占用超过限额的内存时发送。给出进程的 ID
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	当进程试图占用的内存超过作业的内存限制时发送。给出进程的ID
JOB_OBJECT_MSG_NEW_PROCESS	当一个进程添加给作业时发送。给出进程的ID
JOB_OBJECT_MSG_EXIT_PROCESS	当进程终止运行时发送。给出进程的ID
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	当进程由于未处理的异常事件而终止运行时发送。给出进程的ID
JOB_OBJECT_MSG_END_OF_JOB_TIME	当超过分配给作业的CPU时间时发送。这些进程没有终止运行。可以允许它们继续运行，设置一个新的时间限制，或者自己调用TerminateJobObject函数

最后要说明的一点是，按照默认设置，作业对象是这样配置的：当分配给作业的 CPU时间已经到期时，作业的所有进程均自动停止运行，而 JOB_OBJECT_MSG_END_OF_JOB_TIME通知尚未发送。如果想要防止作业对象撤消进程而只是通知你时间已经超过，必须执行下面这样的代码：

```
// Create a JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure
// and initialize its only member.
JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
joeojti.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;

// Tell the job object what we want it to do when the job time is
// exceeded.
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,
   &joeojti, sizeof(joeojti));
```

为作业设定结束时间而使用的另一个值是JOB_OBJECT_TERMINATE_AT_END_OF_JOB，这是作业创建时的默认值。

5.6 JobLab示例应用程序

JobLab应用程序“ 05JobLab.exe ”（在本章末尾处清单 5-2中列出）使你能够很容易地对作业进行实验性操作。该应用程序的源代码和资源文件放在本书所附光盘上 05-JobLab目录中。当启动该程序时，出现图5-4所示的窗口。

当进程被初始化时，它创建一个作业对象。我创建的这个作业对象的名字是JobLab，这样，就可以使用MMC的Performance Monitor Snap-In来观察和监控它的性能。该应用程序还创建了

China-pub.com
下载

一个I/O完成端口，并将作业对象与它相关联。这样就可以对来自作业的通知进行监控，并可显示在窗口底部的列表框中。

开始时，该作业不包含进程，也没有各种限制条件。顶部的各个域用于设定对作业的基本限制和扩展限制条件。要做的工作是用有效值填写这些域，然后点击 Apply Limits按钮。如果将一个域置空，那么该限制条件就不起作用。除了基本限制和扩展限制条件外，还可以打开和关闭各种UI限制。注意，Preserve Job Time When Applying Limits（当运用各个限制时保留作业时间）复选框并不用于设置限制条件。它只是让你在查询基本统计信息时可以改变作业的限制条件，而不重置ThisPeriodTotalUserTime和ThisPeriod-TotalKernelTime成员。当运用单个作业的时间限制时，该复选框不起作用。

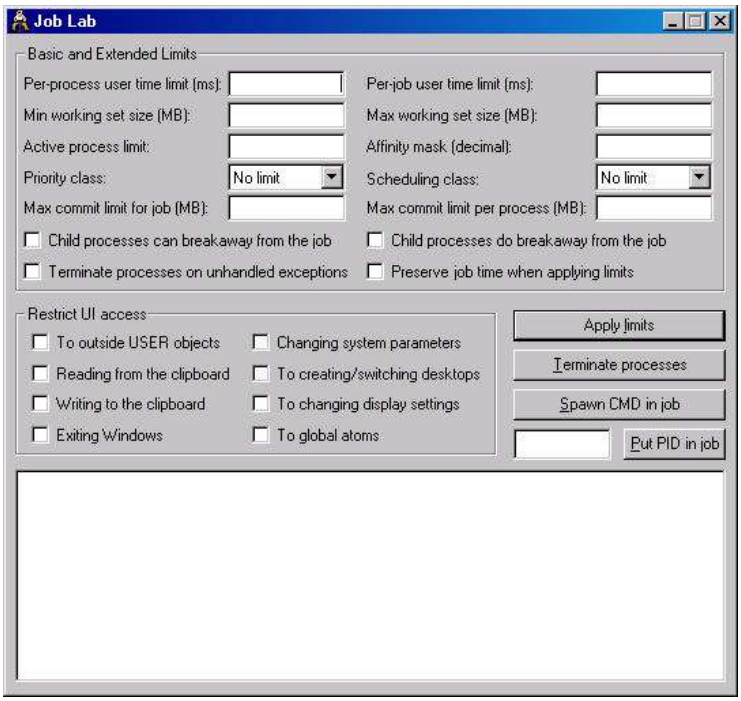


图5-4 JobLab示例应用程序

其余的按钮供你用其他方式对作业进行操作。 Terminate Processes按钮用于撤消作业中的所有进程。Spawn CMD In Job按钮用于生成与作业相关的命令外壳进程。从该命令外壳程序中，可以生成更多的子进程，并且可以看到它们如何作为作业的组成部分来运行。我发现这对试验操作是非常有用的。最后一个按钮是 Put PID In Job，它用于将现有的无作业进程与作业相关联。

窗口底部的列表框显示了更新的关于作业的状态信息。每隔 10s，该窗口显示一次基本统计信息和I/O统计信息，以及进程/作业的内存峰值使用量。同时也显示作业中当前的每个进程的ID。

最后要说明的是，如果修改了源代码，并且创建一个没有名字的作业内核对象，那么可以运行该应用程序的多个拷贝，以便在同一台机器上创建两个或多个作业对象，并且进行更多的试验。

就源代码而言，没有什么特殊的东西需要介绍，因为源代码已经做了非常完善的说明。不过，我创建了一个Job.h文件，它定义了一个Cjob C++类，用于封装操作系统的作业对象。这

China-pub.com 下载

使得操作起来更加容易，因为不必到处传递作业的句柄。这个类还减少了平常调用 QueryInformationJobObject和 SetInformatiomJobObject函数时需要进行的转换工作量。

清单5-2 JobLab示例应用程序



JobLab.cpp

```
/*++
Module:  JobLab.cpp
Notice:  Copyright (c) 2000 Jeffrey Richter
--*/


#include "..\CmnHdr.h"
#include <windowsx.h>
#include <process.h>     // For _beginthreadex
#include <tlhelp32.h>
#include <stdio.h>
#include "Resource.h"
#include "Ch8.h"


///////////////////////////////////////////////////////////////////////////


CJob g_job;              // Job object

HWND g_hwnd;             // Handle to dialog box (accessible by all threads)

HANDLE g_hIOCP;          // I/O completion port that receives job notifications
HANDLE g_hThreadIOCP;    // I/O completion port thread

// I/O completion keys for the completion port
#define COMPKEY_TERMINATE  ((ULONG_PTR) 0)
#define COMPKEY_STATUS     ((ULONG_PTR) 1)
#define COMPKEY_NEWJOB     ((ULONG_PTR) 2)


///////////////////////////////////////////////////////////////////////////


DWORD WINAPI JobNotify(PVOID) {
   TCHAR sz[2000];
   BOOL fDone = FALSE;

   while (!fDone) {
      DWORD dwBytesXferred;
      ULONG_PTR CompKey;
      LPOVERLAPPED po;
      GetQueuedCompletionStatus(g_hIOCP,
         &dwBytesXferred, &CompKey, &po, INFINITE);

      // The app is shutting down, exit this thread
      fDone = (CompKey == COMPKEY_TERMINATE);

      HWND hwndLB = FindWindow(NULL, TEXT("Job Lab"));
```

China-pub.com
下载


```
hwndLB = GetDlgItem(hwndLB, IDC_STATUS);


if (CompKey == COMPKEY_JOBOBJECT) {
   lstrcpy(sz, TEXT("--> Notification: "));
   LPTSTR psz = sz + lstrlen(sz);
   switch (dwBytesXferred) {
   case JOB_OBJECT_MSG_END_OF_JOB_TIME:
      wsprintf(psz, TEXT("Job time limit reached"));
      break;

   case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
      wsprintf(psz, TEXT("Job process (Id=%d) time limit reached"), po);
      break;

   case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
      wsprintf(psz, TEXT("Too many active processes in job"));
      break;

   case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
      wsprintf(psz, TEXT("Job contains no active processes"));
      break;

   case JOB_OBJECT_MSG_NEW_PROCESS:
      wsprintf(psz, TEXT("New process (Id=%d) in Job"), po);
      break;

   case JOB_OBJECT_MSG_EXIT_PROCESS:
      wsprintf(psz, TEXT("Process (Id=%d) terminated"), po);
      break;

   case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
      wsprintf(psz, TEXT("Process (Id=%d) terminated abnormally"), po);
      break;

   case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
      wsprintf(psz, TEXT("Process (Id=%d) exceeded memory limit"), po);
      break;

   case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
      wsprintf(psz,
         TEXT("Process (Id=%d) exceeded job memory limit"), po);
      break;

   default:
      wsprintf(psz, TEXT("Unknown notification: %d"), dwBytesXferred);
      break;
   }
   ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
   CompKey = 1;   // Force a status update when a notification arrives
}


if (CompKey == COMPKEY_STATUS) {

   static int s_nStatusCount = 0;
   _stprintf(sz, TEXT("--> Status Update (%u)"), s_nStatusCount++);
```

China-pub.com 下载

```
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

            // Show the basic accounting information
            JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION jobai;
            g_job.QueryBasicAccountingInfo(&jobai);

            _stprintf(sz, TEXT("Total Time: User=%I64u, Kernel=%I64u        ")
               TEXT("Period Time: User=%I64u, Kernel=%I64u"),
               jobai.BasicInfo.TotalUserTime.QuadPart,
               jobai.BasicInfo.TotalKernelTime.QuadPart,
               jobai.BasicInfo.ThisPeriodTotalUserTime.QuadPart,
               jobai.BasicInfo.ThisPeriodTotalKernelTime.QuadPart);
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

            _stprintf(sz, TEXT("Page Faults=%u, Total Processes=%u, ")
               TEXT("Active Processes=%u, Terminated Processes=%u"),
               jobai.BasicInfo.TotalPageFaultCount,
               jobai.BasicInfo.TotalProcesses,
               jobai.BasicInfo.ActiveProcesses,
               jobai.BasicInfo.TotalTerminatedProcesses);
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

            // Show the I/O accounting information
            _stprintf(sz, TEXT("Reads=%I64u (%I64u bytes), ")
               TEXT("Write=%I64u (%I64u bytes), Other=%I64u (%I64u bytes)"),
               jobai.IoInfo.ReadOperationCount,  jobai.IoInfo.ReadTransferCount,
               jobai.IoInfo.WriteOperationCount, jobai.IoInfo.WriteTransferCount,
               jobai.IoInfo.OtherOperationCount, jobai.IoInfo.OtherTransferCount);
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

            // Show the peak per-process and job memory usage
            JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli;
            g_job.QueryExtendedLimitInfo(&joeli);
            _stprintf(sz, TEXT("Peak memory used: Process=%I64u, Job=%I64u"),
               (__int64) joeli.PeakProcessMemoryUsed,
               (__int64) joeli.PeakJobMemoryUsed);
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

            // Show the set of Process IDs
            DWORD dwNumProcesses = 50, dwProcessIdList[50];
            g_job.QueryBasicProcessIdList(dwNumProcesses,
               dwProcessIdList, &dwNumProcesses);
            _stprintf(sz, TEXT("PIDs: %s"),
               (dwNumProcesses == 0) ? TEXT("(none)") : TEXT(""));
            for (DWORD x = 0; x < dwNumProcesses; x++) {
               _stprintf(_tcschr(sz, 0), TEXT("%d "), dwProcessIdList[x]);
            }
            ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
         }
      }
      return(0);
   }

   //////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_JOBLAB);

   // Save our window handle so that the completion port thread can access it
   g_hwnd = hwnd;
   HWND hwndPriorityClass = GetDlgItem(hwnd, IDC_PRIORITYCLASS);
   ComboBox_AddString(hwndPriorityClass, TEXT("No limit"));
   ComboBox_AddString(hwndPriorityClass, TEXT("Idle"));
   ComboBox_AddString(hwndPriorityClass, TEXT("Below normal"));
   ComboBox_AddString(hwndPriorityClass, TEXT("Normal"));
   ComboBox_AddString(hwndPriorityClass, TEXT("Above normal"));
   ComboBox_AddString(hwndPriorityClass, TEXT("High"));
   ComboBox_AddString(hwndPriorityClass, TEXT("Realtime"));
   ComboBox_SetCurSel(hwndPriorityClass, 0); // Default to "No Limit"

   HWND hwndSchedulingClass = GetDlgItem(hwnd, IDC_SCHEDULINGCLASS);
   ComboBox_AddString(hwndSchedulingClass, TEXT("No limit"));
   for (int n = 0; n <= 9; n++) {
      TCHAR szSchedulingClass[2] = { (TCHAR) (TEXT('0') + n), 0 };
      ComboBox_AddString(hwndSchedulingClass, szSchedulingClass);
   }
   ComboBox_SetCurSel(hwndSchedulingClass, 0); // Default to "No Limit"
   SetTimer(hwnd, 1, 10000, NULL);             // 10 second accounting update
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_ApplyLimits(HWND hwnd) {
   const int nNanosecondsPerSecond = 100000000;
   const int nMillisecondsPerSecond = 1000;
   const int nNanosecondsPerMillisecond =
      nNanosecondsPerSecond / nMillisecondsPerSecond;
   BOOL f;
   __int64 q;
   SIZE_T s;
   DWORD d;

   // Set Basic and Extended Limits
   JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli = { 0 };
   joeli.BasicLimitInformation.LimitFlags = 0;

   q = GetDlgItemInt(hwnd, IDC_PERPROCESSUSERTIMELIMIT, &f, FALSE);
   if (f) {
      joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PROCESS_TIME;
      joeli.BasicLimitInformation.PerProcessUserTimeLimit.QuadPart =
         q * nNanosecondsPerMillisecond / 100;
   }
   q = GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
   if (f) {
      joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_TIME;
      joeli.BasicLimitInformation.PerJobUserTimeLimit.QuadPart =
         q * nNanosecondsPerMillisecond / 100;
   }
```

China-pub.com 下载

```
s = GetDlgItemInt(hwnd, IDC_MINWORKINGSETSIZE, &f, FALSE);
if (f) {
   joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_WORKINGSET;
   joeli.BasicLimitInformation.MinimumWorkingSetSize = s * 1024 * 1024;
   s = GetDlgItemInt(hwnd, IDC_MAXWORKINGSETSIZE, &f, FALSE);
   if (f) {
      joeli.BasicLimitInformation.MaximumWorkingSetSize = s * 1024 * 1024;
   } else {
      joeli.BasicLimitInformation.LimitFlags &=~JOB_OBJECT_LIMIT_WORKINGSET;
      chMB("Both minimum and maximum working set sizes must be set.\n"
         "The working set limits will NOT be in effect.");
   }
}

d = GetDlgItemInt(hwnd, IDC_ACTIVEPROCESSLIMIT, &f, FALSE);
if (f) {
   joeli.BasicLimitInformation.LimitFlags |=
      JOB_OBJECT_LIMIT_ACTIVE_PROCESS;
   joeli.BasicLimitInformation.ActiveProcessLimit = d;
}

s = GetDlgItemInt(hwnd, IDC_AFFINITYMASK, &f, FALSE);
if (f) {
   joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_AFFINITY;
   joeli.BasicLimitInformation.Affinity = s;
}

joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PRIORITY_CLASS;
switch (ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_PRIORITYCLASS))) {
   case 0:
      joeli.BasicLimitInformation.LimitFlags &=
         ~JOB_OBJECT_LIMIT_PRIORITY_CLASS;
      break;

   case 1:
      joeli.BasicLimitInformation.PriorityClass =
         IDLE_PRIORITY_CLASS;
      break;
   case 2:
      joeli.BasicLimitInformation.PriorityClass =
         BELOW_NORMAL_PRIORITY_CLASS;
      break;

   case 3:
      joeli.BasicLimitInformation.PriorityClass =
         NORMAL_PRIORITY_CLASS;
      break;

   case 4:
      joeli.BasicLimitInformation.PriorityClass =
         ABOVE_NORMAL_PRIORITY_CLASS;
      break;

   case 5:
      joeli.BasicLimitInformation.PriorityClass =
         HIGH_PRIORITY_CLASS;
      break;
```

China-pub.com
下载


```
      case 6:
         joeli.BasicLimitInformation.PriorityClass =
            REALTIME_PRIORITY_CLASS;
         break;
   }

   int nSchedulingClass =
      ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_SCHEDULINGCLASS));
   if (nSchedulingClass > 0) {
      joeli.BasicLimitInformation.LimitFlags |=
         JOB_OBJECT_LIMIT_SCHEDULING_CLASS;
      joeli.BasicLimitInformation.SchedulingClass = nSchedulingClass - 1;
   }

   s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERJOB, &f, FALSE);
   if (f) {
      joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_MEMORY;
      joeli.JobMemoryLimit = s * 1024 * 1024;
   }

   s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERPROCESS, &f, FALSE);
   if (f) {
      joeli.BasicLimitInformation.LimitFlags |=
         JOB_OBJECT_LIMIT_PROCESS_MEMORY;
      joeli.ProcessMemoryLimit = s * 1024 * 1024;
   }

   if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESCANBREAKAWAYFROMJOB))
      joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_BREAKAWAY_OK;

   if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESDOBREAKAWAYFROMJOB))
      joeli.BasicLimitInformation.LimitFlags |=
      JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

   if (IsDlgButtonChecked(hwnd, IDC_TERMINATEPROCESSONEXCEPTIONS))
      joeli.BasicLimitInformation.LimitFlags |=
      JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION;

   f = g_job.SetExtendedLimitInfo(&joeli,
      ((joeli.BasicLimitInformation.LimitFlags & JOB_OBJECT_LIMIT_JOB_TIME)
         != 0) ? FALSE :
         IsDlgButtonChecked(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS));
   chASSERT(f);

   // Set UI Restrictions
   DWORD jobuir =  JOB_OBJECT_UILIMIT_NONE;  // A fancy zero (0)
   if (IsDlgButtonChecked(hwnd, IDC_RESTRICTACCESSTOOUTSIDEUSEROBJECTS))
      jobuir |=  JOB_OBJECT_UILIMIT_HANDLES;

   if (IsDlgButtonChecked(hwnd, IDC_RESTRICTREADINGCLIPBOARD))
      jobuir |=  JOB_OBJECT_UILIMIT_READCLIPBOARD;

   if (IsDlgButtonChecked(hwnd, IDC_RESTRICTWRITINGCLIPBOARD))
      jobuir |=  JOB_OBJECT_UILIMIT_WRITECLIPBOARD;

   if (IsDlgButtonChecked(hwnd, IDC_RESTRICTEXITWINDOW))
      jobuir |=  JOB_OBJECT_UILIMIT_EXITWINDOWS;
```

China-pub.com 下载

```
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTCHANGINGSYSTEMPARAMETERS))
   jobuir |=  JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDESKTOPS))
   jobuir |= JOB_OBJECT_UILIMIT_DESKTOP;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDISPLAYSETTINGS))
   jobuir |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTGLOBALATOMS))
   jobuir |=  JOB_OBJECT_UILIMIT_GLOBALATOMS;

chVERIFY(g_job.SetBasicUIRestrictions(jobuir));
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         // User is terminating our app, kill the job too.
         KillTimer(hwnd, 1);
         g_job.Terminate(0);
         EndDialog(hwnd, id);
         break;

      case IDC_PERJOBUSERTIMELIMIT:
         {
         // The job time must be reset if setting a job time limit
         BOOL f;
         GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
         EnableWindow(
            GetDlgItem(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS), !f);
         }
         break;

      case IDC_APPLYLIMITS:
         Dlg_ApplyLimits(hwnd);
         PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
         break;

      case IDC_TERMINATE:
         g_job.Terminate(0);
         PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
         break;

      case IDC_SPAWNCMDINJOB:
         {
         // Spawn a command shell and place it in the job
         STARTUPINFO si = { sizeof(si) };
         PROCESS_INFORMATION pi;
         TCHAR sz[] = TEXT("CMD");
         CreateProcess(NULL, sz, NULL, NULL,
            FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
         g_job.AssignProcess(pi.hProcess);
         ResumeThread(pi.hThread);
         CloseHandle(pi.hProcess);
```

China-pub.com
下载


```
      CloseHandle(pi.hThread);
      }
      PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
      break;
      case IDC_ASSIGNPROCESSTOJOB:
         {
         DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
         HANDLE hProcess = OpenProcess(
            PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALSE, dwProcessId);
         if (hProcess != NULL) {
            chVERIFY(g_job.AssignProcess(hProcess));
            CloseHandle(hProcess);
         } else chMB("Could not assign process to job.");
         }
         PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

void WINAPI Dlg_OnTimer(HWND hwnd, UINT id) {

   PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_TIMER,      Dlg_OnTimer);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

   // Create the completion port that receives job notifications
   g_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

   // Create a thread that waits on the completion port
   g_hThreadIOCP = chBEGINTHREADEX(NULL, 0, JobNotify, NULL, 0, NULL);
   // Create the job object
   g_job.Create(NULL, TEXT("JobLab"));
   g_job.SetEndOfJobInfo(JOB_OBJECT_POST_AT_END_OF_JOB);
   g_job.AssociateCompletionPort(g_hIOCP, COMPKEY_JOBOBJECT);

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_JOBLAB), NULL, Dlg_Proc);

   // Post a special key that tells the completion port thread to terminate
```

China-pub.com 下载

```
   PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_TERMINATE, NULL);

   // Wait for the completion port thread to terminate
   WaitForSingleObject(g_hThreadIOCP, INFINITE);

   // Clean up everything properly
   CloseHandle(g_hIOCP);
   CloseHandle(g_hThreadIOCP);

   // NOTE: The job is closed when the g_job's destructor is called.
   return(0);
}

//////////////////////////////// End Of File /////////////////////////////////
```

Job.h

```
/******************************************************************************
Module:  Job.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once


//////////////////////////////////////////////////////////////////////////////


#include <malloc.h>  // for _alloca


//////////////////////////////////////////////////////////////////////////////
class CJob {
public:
   CJob(HANDLE hJob = NULL);
   ~CJob();

   operator HANDLE() const { return(m_hJob); }

   // Functions to create/open a job object
   BOOL Create(LPSECURITY_ATTRIBUTES psa = NULL, LPCTSTR pszName = NULL);
   BOOL Open(LPCTSTR pszName, DWORD dwDesiredAccess,
      BOOL fInheritHandle = FALSE);

   // Functions that manipulate a job object
   BOOL AssignProcess(HANDLE hProcess);
   BOOL Terminate(UINT uExitCode = 0);

   // Functions that set limits/restrictions on the job
   BOOL SetExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli,
      BOOL fPreserveJobTime = FALSE);
   BOOL SetBasicUIRestrictions(DWORD fdwLimits);
   BOOL GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant = TRUE);
   BOOL SetSecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);
```

China-pub.com
下载


```
   // Functions that query job limits/restrictions
   BOOL QueryExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli);
   BOOL QueryBasicUIRestrictions(PDWORD pfdwRestrictions);
   BOOL QuerySecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

   // Functions that query job status information
   BOOL QueryBasicAccountingInfo(
      PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai);
   BOOL QueryBasicProcessIdList(DWORD dwMaxProcesses,
      PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned = NULL);

   // Functions that set/query job event notifications
   BOOL AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey);
   BOOL QueryAssociatedCompletionPort(
      PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp);
   BOOL SetEndOfJobInfo(
      DWORD fdwEndOfJobInfo = JOB_OBJECT_TERMINATE_AT_END_OF_JOB);
   BOOL QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo);

private:
   HANDLE m_hJob;
};
//////////////////////////////////////////////////////////////////////////////


inline CJob::CJob(HANDLE hJob) {

   m_hJob = hJob;
}


//////////////////////////////////////////////////////////////////////////////


inline CJob::~CJob() {

   if (m_hJob != NULL)
      CloseHandle(m_hJob);
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::Create(PSECURITY_ATTRIBUTES psa, PCTSTR pszName) {

   m_hJob = CreateJobObject(psa, pszName);
   return(m_hJob != NULL);
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::Open(
```

China-pub.com 下载

```
   PCTSTR pszName, DWORD dwDesiredAccess, BOOL fInheritHandle) {

   m_hJob = OpenJobObject(dwDesiredAccess, fInheritHandle, pszName);
   return(m_hJob != NULL);
}


//////////////////////////////////////////////////////////////////////////////
inline BOOL CJob::AssignProcess(HANDLE hProcess) {

   return(AssignProcessToJobObject(m_hJob, hProcess));
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey) {

   JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp = { (PVOID) CompKey, hIOCP };
   return(SetInformationJobObject(m_hJob,
      JobObjectAssociateCompletionPortInformation, &joacp, sizeof(joacp)));
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::SetExtendedLimitInfo(
   PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli, BOOL fPreserveJobTime) {

   if (fPreserveJobTime)
      pjoeli->BasicLimitInformation.LimitFlags |=
         JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME;

   // If we are to preserve the job's time information,
   // the JOB_OBJECT_LIMIT_JOB_TIME flag must not be on
   const DWORD fdwFlagTest =
      (JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME | JOB_OBJECT_LIMIT_JOB_TIME);

   if ((pjoeli->BasicLimitInformation.LimitFlags & fdwFlagTest)
      == fdwFlagTest) {
      // These flags are mutually exclusive but both are on, error
      DebugBreak();
   }

   return(SetInformationJobObject(m_hJob,
      JobObjectExtendedLimitInformation, pjoeli, sizeof(*pjoeli)));
}


//////////////////////////////////////////////////////////////////////////////
inline BOOL CJob::SetBasicUIRestrictions(DWORD fdwLimits) {

   JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir = { fdwLimits };
   return(SetInformationJobObject(m_hJob,
```

China-pub.com
下载


```
      JobObjectBasicUIRestrictions, &jobuir, sizeof(jobuir)));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::SetEndOfJobInfo(DWORD fdwEndOfJobInfo) {

   JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti = { fdwEndOfJobInfo };
   joeojti.EndOfJobTimeAction = fdwEndOfJobInfo;
   return(SetInformationJobObject(m_hJob,
      JobObjectEndOfJobTimeInformation, &joeojti, sizeof(joeojti)));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::SetSecurityLimitInfo(
   PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

   return(SetInformationJobObject(m_hJob,
      JobObjectSecurityLimitInformation, pjosli, sizeof(*pjosli)));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryAssociatedCompletionPort(
   PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp) {

   return(QueryInformationJobObject(m_hJob,
      JobObjectAssociateCompletionPortInformation, pjoacp, sizeof(*pjoacp),
      NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryBasicAccountingInfo(
   PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai) {

   return(QueryInformationJobObject(m_hJob,
      JobObjectBasicAndIoAccountingInformation, pjobai, sizeof(*pjobai),
      NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryExtendedLimitInfo(
   PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli) {

   return(QueryInformationJobObject(m_hJob, JobObjectExtendedLimitInformation,
```

China-pub.com 下载

```
      pjoeli, sizeof(*pjoeli), NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryBasicProcessIdList(DWORD dwMaxProcesses,
   PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned) {

   // Calculate the # of bytes necessary
   DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
      (sizeof(DWORD) * (dwMaxProcesses - 1));

   // Allocate those bytes from the stack
   PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil =
      (PJOBOBJECT_BASIC_PROCESS_ID_LIST) _alloca(cb);

   // Were those bytes allocated OK? If so, keep going
   BOOL fOk = (pjobpil != NULL);

   if (fOk) {
      pjobpil->NumberOfProcessIdsInList = dwMaxProcesses;
      fOk = ::QueryInformationJobObject(m_hJob, JobObjectBasicProcessIdList,
         pjobpil, cb, NULL);

      if (fOk) {
         // We got the information, return it to the caller
         if (pdwProcessesReturned != NULL)
            *pdwProcessesReturned = pjobpil->NumberOfProcessIdsInList;
         CopyMemory(pdwProcessIdList, pjobpil->ProcessIdList,
            sizeof(DWORD) * pjobpil->NumberOfProcessIdsInList);
      }
   }
   return(fOk);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryBasicUIRestrictions(PDWORD pfdwRestrictions) {

   JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
   BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
      &jobuir, sizeof(jobuir), NULL);
   if (fOk)
      *pfdwRestrictions = jobuir.UIRestrictionsClass;
   return(fOk);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CJob::QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo) {

   JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
```

China-pub.com
下载


```
   BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
      &joeojti, sizeof(joeojti), NULL);
   if (fOk)
      *pfdwEndOfJobTimeInfo = joeojti.EndOfJobTimeAction;
   return(fOk);
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::QuerySecurityLimitInfo(
   PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

   return(QueryInformationJobObject(m_hJob, JobObjectSecurityLimitInformation,
      pjosli, sizeof(*pjosli), NULL));
}
//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::Terminate(UINT uExitCode) {

   return(TerminateJobObject(m_hJob, uExitCode));
}


//////////////////////////////////////////////////////////////////////////////


inline BOOL CJob::GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant) {

   return(UserHandleGrantAccess(hUserObj, m_hJob, fGrant));
}


//////////////////////////////// End of File /////////////////////////////////
```

JobLab.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources
```

China-pub.com 下载

```
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MEMRESET            ICON    DISCARDABLE     "MemReset.ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第6章　线程的基础知识

理解线程是非常关键的，因为每个进程至少需要一个线程。本章将更加详细地介绍线程的知识。尤其是要讲述进程与线程之间存在多大的差别，它们各自具有什么作用。还要介绍系统如何使用线程内核对象来管理线程。与进程内核对象一样，线程内核对象也拥有属性，我们将要观察许多用于查询和修改这些属性的函数。此外还要介绍可以在进程中创建和生成更多的线程时所用的函数。

第4章介绍了进程是由两个部分构成的，一个是进程内核对象，另一个是地址空间。同样，线程也是由两个部分组成的：

- 一个是线程的内核对象，操作系统用它来对线程实施管理。内核对象也是系统用来存放线程统计信息的地方。
- 另一个是线程堆栈，它用于维护线程在执行代码时需要的所有函数参数和局部变量（第16章将进一步介绍系统如何管理线程堆栈）。

第4章中讲过，进程是不活泼的。进程从来不执行任何东西，它只是线程的容器。线程总是在某个进程环境中创建的，而且它的整个寿命期都在该进程中。这意味着线程在它的进程地址空间中执行代码，并且在进程的地址空间中对数据进行操作。因此，如果在单进程环境中，你有两个或多个线程正在运行，那么这两个线程将共享单个地址空间。这些线程能够执行相同的代码，对相同的数据进行操作。这些线程还能共享内核对象句柄，因为句柄表依赖于每个进程而不是每个线程存在。

如你所见，进程使用的系统资源比线程多得多，原因是它需要更多的地址空间。为进程创建一个虚拟地址空间需要许多系统资源。系统中要保留大量的记录，这要占用大量的内存。另外，由于.exe和.dll文件要加载到一个地址空间，因此也需要文件资源。而线程使用的系统资源要少得多。实际上，线程只有一个内核对象和一个堆栈，保留的记录很少，因此需要很少的内存。

由于线程需要的开销比进程少，因此始终都应该设法用增加线程来解决编程问题，而要避免创建新的进程。但是，这个建议并不是一成不变的。许多程序设计用多个进程来实现会更好些。应该懂得权衡利弊，经验会指导你的编程实践。

在详细介绍线程之前，首先花一点时间讲一讲如何正确地在应用程序结构中使用线程。

6.1　何时创建线程

线程用于描述进程中的运行路径。每当进程被初始化时，系统就要创建一个主线程。该线程与C/C++运行期库的启动代码一道开始运行，启动代码则调用进入点函数（ main、wmain、WinMain或wWinMain），并且继续运行直到进入点函数返回并且 C/C++运行期库的启动代码调用ExitProcess为止。对于许多应用程序来说，这个主线程是应用程序需要的唯一线程。不过，进程能够创建更多的线程来帮助执行它们的操作。

每个计算机都拥有一个功能非常强大的资源，即 CPU。让CPU闲置起来是绝对没有道理的（如果忽略节省电能问题的话）。为了使CPU处于繁忙状态之中，可以让它执行各种不同的工作。下面是一些例子：

China-pub.com 下载

- 可以打开Microsoft Windows 2000配备的内容索引服务程序。它能够创建一个低优先级的线程，以便定期打开你的磁盘驱动器上的文件内容并给内容做索引。若要找到一个文件，可以打开Search Result（搜索结果）窗口（方法是单击 Start按钮，从Search菜单中选定For Files Or Folders），再将你的搜索条件输入Containing Text域。这时就可以搜索到索引，相关的文件就会立即显示出来。内容索引服务程序大大改进了性能，因为每次搜索不必打开、扫描和关闭磁盘驱动器上的每个文件。
- 可以使用Windows 2000配备的磁盘碎片整理软件。通常情况下，这种类型的实用程序拥有许多管理选项，一般用户可能不懂，比如该实用程序应该相隔多长时间运行一次，何时运行。使用低优先级线程，可以在后台运行该实用程序，并且在系统空闲时对驱动器进行碎片整理。
- 可以很容易地设想将来版本的编译器，每当暂停键入时，它就可以自动编译你的源代码文件。输出窗口可以向你（几乎）实时显示警告和出错信息。当键入变量和函数名时出现错误时，就能立即发现。在某种程度上讲， Microsoft Visual Studio已经实现了这个功能，使用Workspace的ClassView窗格，就能够看到这些信息。
- 电子表格应用程序能够在后台执行各种计算。
- 字处理程序能够执行重新分页、拼写和语法检查及在后台进行打印。
- 文件可以在后台拷贝到其他介质中。
- Web浏览器在后台与它们的服务器进行通信。因此，在来自当前 Web站点的结果输入之前，用户可以缩放浏览器的窗口或者转到另一个Web站点。

这些例子中，有一个重要问题应该注意，那就是多线程能够简化应用程序的用户界面。如果每当停止键入时，编译器建立了你的应用程序，那么就没有必要提供 Build菜单选项。文字处理应用程序不需要Check Spelling(拼写检查)和Check Grammar（语法检查）菜单选项。

在Web浏览器的例子中，注意，将不同的线程用于 I/O（网络、文件或其他），应用程序的用户界面就能够始终保持工作状态。比如有一个应用程序负责给数据库记录进行排序、打印文档或拷贝文件。如果将独立的线程用于处理这个与 I/O相关的任务，用户就可以在进程中继续使用应用程序界面来取消操作。

设计一个拥有多线程的应用程序，就会扩大该应用程序的功能。我们在下一章中可以看到，每个线程被分配了一个CPU。因此，如果你的计算机拥有两个CPU，你的应用程序中有两个线程，那么两个CPU都将处于繁忙状态。实际上，你是让两个任务在执行一个任务的时间内完成操作。

每个进程至少拥有一个线程。因此，如果你在应用程序中不执行任何特殊的操作，在多进程操作系统上运行，就能够得到许多好处。例如，可以建立一个应用程序，并同时使用文字处理程序（我常常这样做）。如果计算机拥有两个CPU，那么该应用程序就可以在一个处理器上执行，而另一个处理器则负责处理文档。另外，如果编译器出现一个错误，导致它的线程进入一个无限循环，仍然可以使用其他的进程（16位Windows和MS-DOS应用程序则不行）。

6.2 何时不能创建线程

至今为止，一直在讨论多线程应用程序的优点。虽然多线程应用程序的优点很多，但是它也存在某些不足之处。有些开发人员认为，解决问题的方法是将它分割成多个线程。这种想法是完全错误的。

线程确实是非常有用的，但是，当使用线程时，在解决原有的问题时可能产生新的问题。

China-pub.com
下载

例如，你开发了一个文字处理应用程序，并且想要让打印函数作为它自己的线程来运行。这听起来是个很好的主意，因为用户可以在打印文档时立即回头着手编辑文档。但是，这意味着文档中的数据可能在文档打印时变更。也许最好是不要让打印操作在它自己的线程中发生，不过这种“方案”看起来有点儿极端。如果你让用户编辑另一个文档，但是锁定正在打印的文档，使得打印结束前该文档不能修改，那将会怎样呢？这里还有第三种思路，将文档拷贝到一个临时文件，然后打印该临时文件的内容，并让用户修改原始文档。当包含该文档的临时文件结束打印时，删除临时文件。

如你所见，线程能够解决某些问题，但是却又会产生新的问题。在开发应用程序的用户界面时，很可能出现对线程的另一种误用。几乎在所有的应用程序中，所有用户界面的组件（窗口）应该共享同一个线程。单个线程应该创建窗口的所有子窗口。有时在不同的线程上创建不同的窗口是有用的，不过这种情况确实非常少见。

通常情况下，一个应用程序拥有一个用户界面线程，用于创建所有窗口，并且有一个GetMessage循环。进程中的所有其他线程都是工作线程，它们与计算机或I/O相关联，但是这些线程从不创建窗口。另外，一个用户界面线程通常拥有比工作线程更高的优先级，因此用户界面负责向用户作出响应。

虽然单个进程拥有多个用户界面线程的情况并不多见，但是这种情况有着某种有效的用途。Windows Explorer为每个文件夹窗口创建了一个独立的线程。它使你能够将文件从一个文件夹拷贝到另一个文件夹，并且仍然可以查看你的系统上的其他文件夹。另外，如果Explorer中存在一个错误，那么负责处理文件夹的线程可能崩溃，但是仍然能够对其他文件夹进行操作，至少在执行的操作导致其他文件夹也崩溃之前，仍然可以对它们进行操作（关于线程和用户界面的详细说明，参见第26和27章）。

上述内容的实质是应该慎重地使用多线程。不要想用就用。仅仅使用赋予进程的主线程，就能够编写出许多非常有用的和功能强大的应用程序。

6.3 编写第一个线程函数

每个线程必须拥有一个进入点函数，线程从这个进入点开始运行。前面已经介绍了主线程的进入点函数：即main、wmain、WinMain或wWinMain。如果想要在你的进程中创建一个辅助线程，它必定也是个进入点函数，类似下面的样子：

```
DWORD WINAPI ThreadFunc(PVOID pvParam){
   DWORD dwResult = 0;.
   :
   :
   return(dwResult);
}
```

你的线程函数可以执行你想要它做的任何任务。最终，线程函数到达它的结尾处并且返回。这时，线程终止运行，该堆栈的内存被释放，同时，线程的内核对象的使用计数被递减。如果使用计数降为0，线程的内核对象就被撤消。与进程内核对象的情况相同，线程内核对象的寿命至少可以达到它们相关联的线程那样长，不过，该对象的寿命可以远远超过线程本身的寿命。

下面对线程函数的几个问题作一说明：

- 主线程的进入点函数的名字必须是main、wmain、WinMain或wWinMain，与这些函数不同的是，线程函数可以使用任何名字。实际上，如果在应用程序中拥有多个线程函数，必须为它们赋予不同的名字，否则编译器/链接程序会认为你为单个函数创建了多个实现函数。

China-pub.com 下载

- 由于给你的主线程的进入点函数传递了字符串参数，因此可以使用 ANSI/Unicode版本的进入点函数：main/wmain和WinMain/wWinMain。可以给线程函数传递单个参数，参数的含义由你而不是由操作系统来定义。因此，不必担心 ANSI/Unicode问题。
- 线程函数必须返回一个值，它将成为该线程的退出代码。这与 C/C++运行期库关于让主线程的退出代码作为进程的退出代码的原则是相似的。
- 线程函数（实际上是你的所有函数）应该尽可能使用函数参数和局部变量。当使用静态变量和全局变量时，多个线程可以同时访问这些变量，这可能破坏变量的内容。然而，参数和局部变量是在线程堆栈中创建的，因此它们不太可能被另一个线程破坏。

既然懂得了实现线程函数的方法，下面讲述如何让操作系统来创建能够执行线程函数的线程。

6.4 CreateThread函数

前面已经讲述了调用CreateProcess函数时如何创建进程的主线程。如果想要创建一个或多个辅助函数，只需要让一个已经在运行的线程来调用 CreateThread：

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   DWORD cbStack,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD fdwCreate,
   PDWORD pdwThreadID);
```

当CreateThread被调用时，系统创建一个线程内核对象。该线程内核对象不是线程本身，而是操作系统用来管理线程的较小的数据结构。可以将线程内核对象视为由关于线程的统计信息组成的一个小型数据结构。这与进程和进程内核对象之间的关系是相同的。

系统从进程的地址空间中分配内存，供线程的堆栈使用。新线程运行的进程环境与创建线程的环境相同。因此，新线程可以访问进程的内核对象的所有句柄、进程中的所有内存和在这个相同的进程中的所有其他线程的堆栈。这使得单个进程中的多个线程确实能够非常容易地互相通信。

注意 CreateThread函数是用来创建线程的 Windows函数。不过，如果你正在编写C/C++代码，决不应该调用 CreateThread。相反，应该使用 Visual C++运行期库函数_beginthreadex。如果不使用Microsoft的Visual C++编译器，你的编译器供应商有它自己的CreateThred替代函数。不管这个替代函数是什么，你都必须使用。本章后面将要介绍_beginthreadex能够做什么，它的重要性何在。

这就是Create Thread函数的概述，下面各节将要具体介绍CreateThread的每个参数。

6.4.1 psa

psa参数是指向SECURITY_ATTRIBUTES结构的指针。如果想要该线程内核对象的默认安全属性，可以（并且通常能够）传递 NULL。如果希望所有的子进程能够继承该线程对象的句柄，必须设定一个SECURITY_ATTRIBUTES结构，它的bInheritHandle成员被初始化为TRUE。详细信息参见第3章。

6.4.2 cbStack

cbStack参数用于设定线程可以将多少地址空间用于它自己的堆栈。每个线程拥有它自己的堆栈。当CreateProcess启动一个进程时，它就在内部调用 CreateThread来对进程的主线程进

China-pub.com
下载

行初始化。对于cbStack参数来说，CreateProcess使用存放在可执行文件中的一个值。可以使用链接程序的/STACK开关来控制这个值：

```
/STACK:[reserve] [,commit]
```

reserve参数用于设定系统应该为线程堆栈保留的地址空间量。默认值是 1 MB。Commit参数用于设定开始时应该承诺用于堆栈保留区的物理存储器的容量。默认值是 1页。当线程中的代码执行时，可能需要多个页面的存储器。当线程溢出它的堆栈时，就生成一个异常条件（关于线程堆栈和堆栈溢出的异常条件的详细说明，参见第 16章，关于一般异常条件的处理的详细说明，参见第23章）。系统抓取该异常条件，并且将另一页（或者你为 commit参数设定的任何值）用于保留空间，这使得线程的堆栈能够根据需要动态地扩大。

当调用CreateThread时，如果传递的值不是0，就能使该函数将所有的存储器保留并分配给线程的堆栈。由于所有的存储器预先作了分配，因此可以确保线程拥有指定容量的可用堆栈存储器。保留空间的容量既可以是/STACK链接程序设定的容量，也可以是CbStack的值，谁大就用谁。分配的存储器容量应该与传递的 cbStack值相一致。如果将 0传递给 CbStack参数，CreateThread就保留一个区域，并且将链接程序嵌入 .exe文件的/STACK链接程序开关信息指明的存储器容量分配给线程堆栈。

保留空间的容量用于为堆栈设置一个上限，这样就可以抓住代码中的循环递归错误。例如，你编写一个递归自调用函数，该函数也包含导致循环递归的一个错误。每次函数调用自己的时候，堆栈上就创建一个新的堆栈框。如果系统不设定堆栈的最大值，该递归函数就永远不会停止对自己的调用。进程的所有地址空间将被分配，大量的物理存储器将被分配给该堆栈。通过设置一个堆栈限制值，就可以防止应用程序用完大量的物理存储器，同时，也可以更快地知道何时程序中出现了错误（第16章中的Summation示例应用程序显示了如何跟踪和处理应用程序中的堆栈溢出）。

6.4.3 pfnStartAddr和pvParam

pfnStartAddr参数用于指明想要新线程执行的线程函数的地址。线程函数的 pvParam参数与原先传递给 CreateThread的pvParam参数是相同的。CreateThread使用该参数不做别的事情，只是在线程启动执行时将该参数传递给线程函数。该参数提供了一个将初始化值传递给线程函数的手段。该初始化数据既可以是数字值，也可以是指向包含其他信息的一个数据结构的指针。

创建多个线程，使这些线程拥有与起始点相同的函数地址，这是完全合乎逻辑的并且是非常有用的。例如，可以实现一个Web服务器，以便创建一个新线程来处理每个客户机的请求。每个线程都知道它正在处理哪个客户机的请求，因为当创建线程时，你传递了一个不同的pzParam值。

记住，Windows是个抢占式多线程系统，这意味着新线程和调用 CreateThread的线程可以同时执行。由于线程可以同时运行，就会出现一些问题。请看下面的代码：

```
DWORD WINAPI FirstThread(PVOID pvParam) {
   // Initialize a stack-based variable
   int x = 0;
   DWORD dwThreadID;

   // Create a new thread.
   HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID) &x,
      0, &dwThreadId);

   // We don't reference the new thread anymore,
   // so close our handle to it.
   CloseHandle(hThread);
```

China-pub.com 下载

```
   // Our thread is done.
   // BUG: our stack will be destroyed, but
   //      SecondThread might try to access it.
   return(0);
}


DWORD WINAPI SecondThread(PVOID pvParam) {
   // Do some lengthy processing here.
   .
   .
   .
   // Attempt to access the variable on FirstThread's stack.
   // NOTE: This may cause an access violation - it depends on timing!
   * ((int *) pvParam) = 5;
   .
   .
   .
   return(0);
}
```

在上面这个代码中，FirstThread可以在SecondThread将5分配给FirstThread的x之前结束它的操作。如果出现这种情况，SecondThread将不知道FirstThread已经不再存在，并且仍然试图修改现在已经是个无效地址的内容。这会导致SecondThread产生一次访问违规，因为FirstThread的堆栈已经在FirstThread终止运行时被撤消。解决这个问题的方法之一是将 x声明为一个静态变量，这样，编译器就为应用程序的数据部分中的x创建一个存储区，而不是在堆栈上创建存储区。

但是这使得函数成为不可重新进入的函数。换句话说，无法创建两个执行相同函数的线程，因为两个线程将共享该静态变量。解决这个问题（和它的更复杂的变形）的另一种方法是使用正确的线程同步技术（第8、9章和10章介绍）。

6.4.4 fdwCreate

fdwCreate参数可以设定用于控制创建线程的其他标志。它可以是两个值中的一个。如果该值是0，那么线程创建后可以立即进行调度。如果该值是 CREATE_SUSPENDED，系统可以完整地创建线程并对它进行初始化，但是要暂停该线程的运行，这样它就无法进行调度。

CREATE_SUSPENDED标志使得应用程序能够在它有机会执行任何代码之前修改线程的某些属性。由于这种必要性很少，因此该标志并不常用。第 5章介绍的JobLab应用程序说明了该标志的正确方法。

6.4.5 pdwThreadID

CreateThread的最后一个参数是pdwThreadID，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(进程和线程的ID已经在第4章中作了介绍)。

注意 在Windows 2000（和Windows NT 4）下，可以（并且通常是这样做的）为该参数传递NULL。它告诉函数，你对线程的 ID不感兴趣，但是线程已经创建了。在Windows 95和Windows 98下，为该参数传递NULL会导致函数运行失败，因为函数试图将ID写入地址NULL（这是不合法的）。因此线程不能创建。

当然，操作系统之间的不一致现象会给编程人员带来一些问题。例如，在Windows 2000下（即使为pdwThreadID参数传递了NULL，它也创建了该线程）编写和测试了一

China-pub.com
下载

个应用程序，当后来在Windows 98上运行该应用程序时，CreateThread将不创建新的线程。必须始终在你声称支持的所有操作系统（和所有版本）上充分测试应用程序。

6.5 终止线程的运行

若要终止线程的运行，可以使用下面的方法：

- 线程函数返回（最好使用这种方法）。
- 通过调用ExitThread函数，线程将自行撤消（最好不要使用这种方法）。
- 同一个进程或另一个进程中的线程调用TerminateThread函数（应该避免使用这种方法）。
- 包含线程的进程终止运行（应该避免使用这种方法）。

下面将介绍终止线程运行的方法，并且说明线程终止运行时会出现什么情况。

6.5.1 线程函数返回

始终都应该将线程设计成这样的形式，即当想要线程终止运行时，它们就能够返回。这是确保所有线程资源被正确地清除的唯一办法。

如果线程能够返回，就可以确保下列事项的实现：

- 在线程函数中创建的所有C++对象均将通过它们的撤消函数正确地撤消。
- 操作系统将正确地释放线程堆栈使用的内存。
- 系统将线程的退出代码（在线程的内核对象中维护）设置为线程函数的返回值。
- 系统将递减线程内核对象的使用计数。

6.5.2 ExitThread函数

可以让线程调用ExitThread函数，以便强制线程终止运行：

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

该函数将终止线程的运行，并导致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）将不被撤消。由于这个原因，最好从线程函数返回，而不是通过调用ExitThread来返回（详细说明参见第4章）。

当然，可以使用ExitThread的dwExitThread参数告诉系统将线程的退出代码设置为什么。ExitThread函数并不返回任何值，因为线程已经终止运行，不能执行更多的代码。

注意 终止线程运行的最佳方法是让它的线程函数返回。但是，如果使用本节介绍的方法，应该知道ExitThread函数是Windows用来撤消线程的函数。如果编写C/C++代码，那么决不应该调用ExitThread。应该使用Visual C++运行期库函数_endthreadex。如果不使用Microsoft的Visual C++编译器，你的编译器供应商有它自己的ExitThread的替代函数。不管这个替代函数是什么，都必须使用。本章后面将说明_endthreadex的作用和它的重要性。

6.5.3 TerminateThread函数

调用TerminateThread函数也能够终止线程的运行：

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

China-pub.com 下载

与ExitThread不同，ExitThread总是撤消调用的线程，而TerminateThread能够撤消任何线程。hThread参数用于标识被终止运行的线程的句柄。当线程终止运行时，它的退出代码成为你作为dwExitCode参数传递的值。同时，线程的内核对象的使用计数也被递减。

注意 TerminateThread函数是异步运行的函数，也就是说，它告诉系统你想要线程终止运行，但是，当函数返回时，不能保证线程被撤消。如果需要确切地知道该线程已经终止运行，必须调用WaitForSingleObject(第9章介绍)或者类似的函数，传递线程的句柄。

设计良好的应用程序从来不使用这个函数，因为被终止运行的线程收不到它被撤消的通知。线程不能正确地清除，并且不能防止自己被撤消。

注意 当使用返回或调用ExitThread的方法撤消线程时，该线程的内存堆栈也被撤消。但是，如果使用TerminateThread，那么在拥有线程的进程终止运行之前，系统不撤消该线程的堆栈。Microsoft故意用这种方法来实现TerminateThread。如果其他仍然正在执行的线程要引用强制撤消的线程堆栈上的值，那么其他的线程就会出现访问违规的问题。如果将已经撤消的线程的堆栈留在内存中，那么其他线程就可以继续很好地运行。

此外，当线程终止运行时，DLL通常接收通知。如果使用Terminate Thread强迫线程终止，DLL就不接收通知，这能阻止适当的清除（详细信息参见第20章）。

6.5.4 在进程终止运行时撤消线程

第4章介绍的ExitProcess和TerminateProcess函数也可以用来终止线程的运行。差别在于这些线程将会使终止运行的进程中的所有线程全部终止运行。另外，由于整个进程已经被关闭，进程使用的所有资源肯定已被清除。这当然包括所有线程的堆栈。这两个函数会导致进程中的剩余线程被强制撤消，就像从每个剩余的线程调用TerminateThread一样。显然，这意味着正确的应用程序清除没有发生，即C++对象撤消函数没有被调用，数据没有转至磁盘等等。

6.5.5 线程终止运行时发生的操作

当线程终止运行时，会发生下列操作：

- 线程拥有的所有用户对象均被释放。在Windows中，大多数对象是由包含创建这些对象的线程的进程拥有的。但是一个线程拥有两个用户对象，即窗口和挂钩。当线程终止运行时，系统会自动撤消任何窗口，并且卸载线程创建的或安装的任何挂钩。其他对象只有在拥有线程的进程终止运行时才被撤消。
- 线程的退出代码从STILL_ACTIVE改为传递给ExitThread或TerminateThread的代码。
- 线程内核对象的状态变为已通知。
- 如果线程是进程中最后一个活动线程，系统也将进程视为已经终止运行。
- 线程内核对象的使用计数递减1。

当一个线程终止运行时，在与它相关联的线程内核对象的所有未结束的引用关闭之前，该内核对象不会自动被释放。

一旦线程不再运行，系统中就没有别的线程能够处理该线程的句柄。然而别的线程可以调用GetExitcodeThread来检查由hThread标识的线程是否已经终止运行。如果它已经终止运行，则确定它的退出代码：

China-pub.com
下载


```
BOOL GetExitCodeThread(
   HANDLE hThread,
   PDWORD pdwExitCode);
```

退出代码的值在pdwExitCode指向的DWORD中返回。如果调用GetExitCodeThread时线程尚未终止运行，该函数就用STILL_ACTIVE标识符（定义为0x103）填入DWORD。如果该函数运行成功，便返回TRUE（第9章将详细地介绍如何使用线程的句柄来确定何时线程终止运行）。

6.6 线程的一些性质

到现在为止，讲述了如何实现线程函数和如何让系统创建线程以便执行该函数。本节将要介绍系统如何使这些操作获得成功。

图6-1显示了系统在创建线程和对线程进行初始化时必须做些什么工作。让我们仔细看一看这个图，以便确切地了解发生的具体情况。调用CreateThread可使系统创建一个线程内核对象。该对象的初始使用计数是2（在线程停止运行和从CreateThread返回的句柄关闭之前，线程内核对象不会被撤消）。线程的内核对象的其他属性也被初始化，暂停计数被设置为1，退出代码始终为STILL_ACTIVE（0x103），该对象设置为未通知状态。

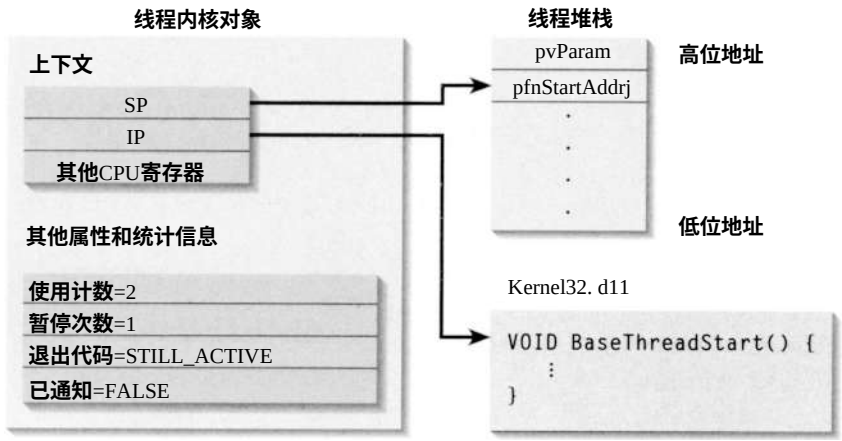


图6-1 线程的创建和初始化的示意图

一旦内核对象创建完成，系统就分配用于线程的堆栈的内存。该内存是从进程的地址空间分配而来的，因为线程并不拥有它自己的地址空间。然后系统将两个值写入新线程的堆栈的上端（线程堆栈总是从内存的高地址向低地址建立）。写入堆栈的第一个值是传递给CreateThread的pvParam参数的值。紧靠它的下面是传递给CreateThread的pfnStartAddr参数的值。

每个线程都有它自己的一组CPU寄存器，称为线程的上下文。该上下文反映了线程上次运行时该线程的CPU寄存器的状态。线程的这组CPU寄存器保存在一个CONTEXT结构（在WinNT.h头文件中作了定义）中。CONTEXT结构本身则包含在线程的内核对象中。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器。记住，线程总是在进程的上下文中运行的。因此，这些地址都用于标识拥有线程的进程地址空间中的内存。当线程的内核对象被初始化时，CONTEXT结构的堆栈指针寄存器被设置为线程堆栈上用来放置pfnStartAddr的地址。指令指针寄存器置为称为BaseThreadStart的未文档化（和未输出）的函数的地址中。该函数包含在Kernel32.dll模块中（这也是实现CreateThread函数的地方）。图6-1显示了它

China-pub.com 下载

的全部情况。

下面是BaseThreadStart函数执行的基本操作：

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
   __try {
      ExitThread((pfnStartAddr)(pvParam));
   }
   __except(UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here.
}
```

当线程完全初始化后，系统就要查看 CREATE_SUSPENDED标志是否已经传递给CreateThread。如果该标志没有传递，系统便将线程的暂停计数递减为0，该线程可以调度到一个进程中。然后系统用上次保存在线程上下文中的值加载到实际的CPU寄存器中。这时线程就可以执行代码，并对它的进程的地址空间中的数据进行操作。

由于新线程的指令指针被置为BaseThreadStart，因此该函数实际上是线程开始执行的地方。BaseThreadStart的原型会使你认为该函数接收了两个参数，但是这表示该函数是由另一个函数来调用的，而实际情况并非如此。新线程只是在此处产生并且开始执行。 BaseThreadStart认为它是由另一个函数调用的，因为它可以访问两个函数。但是，之所以可以访问这些参数，是因为操作系统将值显式写入了线程的堆栈（这就是参数通常传递给函数的方法）。注意，有些CPU结构使用CPU寄存器而不是堆栈来传递参数。对于这些结构来说，系统将在允许线程执行BaseThreadStart函数之前对相应的寄存器正确地进行初始化。

当新线程执行BaseThreadStart函数时，将会出现下列情况：

- 在线程函数中建立一个结构化异常处理（SEH）帧，这样，在线程执行时产生的任何异常情况都会得到系统的某种默认处理（关于结构化异常处理的详细说明参见第23、24和25章）。
- 系统调用线程函数，并将你传递给CreateThread函数的pvParam参数传递给它。
- 当线程函数返回时， BaseThreadStart调用ExitThread，并将线程函数的返回值传递给它。该线程内核对象的使用计数被递减，线程停止执行。
- 如果线程产生一个没有处理的异常条件，由BaseThreadStart函数建立的SEH帧将负责处理该异常条件。通常情况下，这意味着向用户显示一个消息框，并且在用户撤消该消息框时，BzsethreadStart调用ExitThread，以终止整个进程的运行，而不只是终止线程的运行。

注意，在BaseThreadStart函数中，线程要么调用ExitThread，要么调用ExitProcess。这意味着线程不能退出该函数，它总是在函数中被撤消。这就是 BaseThreadStart的原型规定返回VOID，而它从来不返回的原因。

另外，由于使用BaseThreadStart，线程函数可以在它完成处理后返回。当 BaseThreadStart调用线程函数时，它会把返回地址推进堆栈，这样，线程函数就能知道在何处返回。但是，BaseThreadStart不允许返回。如果它不强制撤消线程，而只是试图返回，那么几乎可以肯定会引发访问违规，因为线程堆栈上不存在返回地址，并且 BaseThreadStart将试图返回到某个随机内存位置。

当进程的主线程被初始化时，它的指令指针被设置为另一个未文档化的函数，称为BaseProcessStart。该函数几乎与BaseThreadStart相同，形式类似下面的样子：

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
   __try {
```

China-pub.com
下载


```
        ExitThread((pfnStartAddr)());
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // NOTE: We never get here.
}
```

这两个函数之间的唯一差别是，BaseProcessStart没有引用pvParam参数。当BaseProcessStart开始执行时，它调用C/C++运行期库的启动代码，该启动代码先要初始化main、wmain、WinMain或wWinMain函数，然后调用这些函数。当进入点函数返回时，C/C++运行期库的启动代码就调用ExitProcess。因此，对于C/C++应用程序来说，主线程从不返回BaseProcessStart函数。

6.7 C/C++运行期库的考虑

Visual C++配有6个C/C++运行期库。表6-1对它们进行了描述。

表6-1 C/C++运行期库

库 名	描 述
LibC.lib	用于单线程应用程序的静态链接库（当创建新应用程序时，它是默认库）
LibCD.lib	用于单线程应用程序的静态链接库的调试版
LibCMt.lib	用于多线程应用程序的静态链接库的发行版
LibCMtD.lib	用于多线程应用程序的静态链接库的调试版
MSVCRt.lib	用于动态链接MSVCRt.dll库的发行版的输入库
MSVCRtD.lib	用于动态链接MSVCRtD.dll的调试版的输入库。该库同时支持单线程应用程序和多线程应用程序

当实现任何类型的编程项目时，必须知道将哪个库与你的项目相链接。可以使用图6-2所示的Project Settings对话框来选定一个库。在C/C++选项卡上，在Code Generation（生成的代码）类别中，从Use run-time library（使用运行期库）组合框中选定6个选项中的一个。

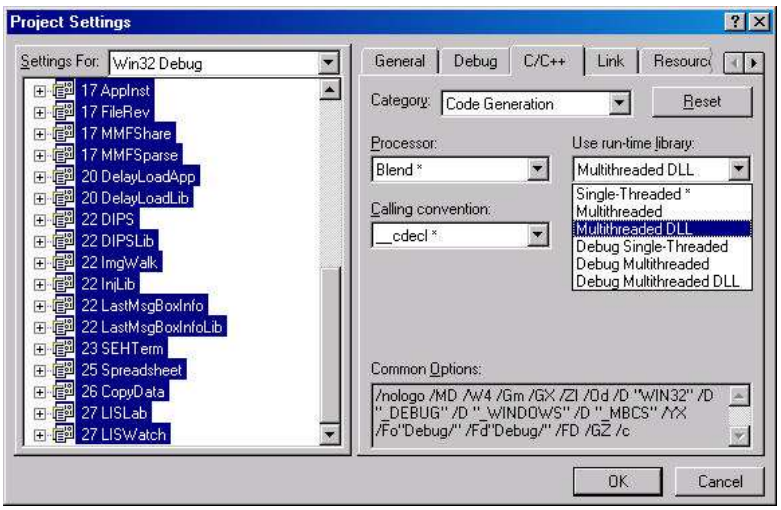


图6-2 Project Settings 对话框

China-pub.com 下载

应该考虑的第一件事情是，“为什么必须将一个库用于单线程应用程序，而将另一个库用于多线程应用程序？”，原因是，标准C运行期库是1970年问世的，它远远早于线程在任何应用程序上的应用。运行期库的发明者没有考虑到将C运行期库用于多线程应用程序的问题。

考虑一下标准C运行期的全局变量errno。有些函数在发生错误时设置该变量。假设拥有下面这个代码段：

```
BOOL fFailure = (system("NOTEPAD.EXE README.TXT") == -1);

if (fFailure) {
   switch (errno) {
   case E2BIG:   // Argument list or environment too big
      break;
   case ENOENT:  // Command interpreter cannot be found
      break;
   case ENOEXEC: // Command interpreter has bad format
      break;
   case ENOMEM:  // Insufficient memory to run command
      break;
   }
}
```

现在，假设在调用system函数之后和调用if语句之前，执行上面代码的线程中断运行，同时假设，该线程中断运行是为了让同一进程中的第二个线程开始执行，而这个新线程将执行另一个负责设置全局变量errno的C运行期函数。当CPU在晚些时候重新分配给第一个线程时，errno的值将不再能够反映调用上面代码中的system函数时的错误代码。为了解决这个问题，每个线程都需要它自己的errno变量。此外，必须有一种机制，使得线程能够引用它自己的errno变量，但是又不触及另一个线程的errno变量。

这是标准C/C++运行期库原先并不是设计用于多线程应用程序的唯一一个例子。在多线程环境中存在问题的C/C++运行期库变量和函数包括errno、_doserrno、strtok、_wcstok、strerror、_strerror、tmpnam、tmpfile、asctime、_wasctime、gmtime、_ecvt和_fcvt等。

若要使多线程C和C++程序能够正确地运行，必须创建一个数据结构，并将它与使用C/C++运行期库函数的每个线程关联起来。当你调用C/C++运行期库时，这些函数必须知道查看调用线程的数据块，这样就不会对别的线程产生不良影响。

那么系统是否知道在创建新线程时分配该数据块呢？回答是它不知道。系统根本不知道你得到的应用程序是用C/C++编写的，也不知道你调用函数的线程本身是不安全的。问题在于你必须正确地进行所有的操作。若要创建一个新线程，绝对不要调用操作系统的CreateThread函数，必须调用C/C++运行期库函数_beginthreadex：

```
unsigned long _beginthreadex(
   void *security,
   unsigned stack_size,
   unsigned (*start_address)(void *),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr);
```

_beginthreadex函数的参数列表与CreateThread函数的参数列表是相同的，但是参数名和类型并不完全相同。这是因为Microsoft的C/C++运行期库的开发小组认为，C/C++运行期函数不应该对Windows数据类型有任何依赖。_beginthreadex函数也像CreateThread那样，返回新创建

China-pub.com
下载

的线程的句柄。因此，如果调用源代码中的CreateThread，就很容易用对_beginthreadex的调用全局取代所有这些调用。不过，由于数据类型并不完全相同，所以必须进行某种转换，使编译器运行得顺利些。为了使操作更加容易，我在源代码中创建了一个宏chBEGINTHREADEX：

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
   pvParam, fdwCreate, pdwThreadID)                 \
      ((HANDLE) _beginthreadex(                     \
         (void *) (psa),                            \
         (unsigned) (cbStack),                      \
         (PTHREAD_START) (pfnStartAddr),            \
         (void *) (pvParam),                        \
         (unsigned) (fdwCreate),                    \
         (unsigned *) (pdwThreadID)))
```

注意，_beginthreadex函数只存在于C/C++运行期库的多线程版本中。如果链接到单线程运行期库，就会得到一个链接程序报告的“未转换的外部符号”错误消息。当然，从设计上讲，这个错误的原因是单线程库在多线程应用程序中不能正确地运行。另外需要注意，当创建一个新项目时，Visual Studio默认选定单线程库。这并不是最安全的默认设置，对于多线程应用程序来说，必须显式转换到多线程的C/C++运行期库。

由于Microsoft为C/C++运行期库提供了源代码，因此很容易准确地确定CreateThread究竟无法执行哪些_beginthreadex能执行的操作。实际上，我搜索了Visual Studio的光盘，发现_beginthreadex的源代码在Threadex.c中。代换重新打印它的源代码，这里提供了它的伪代码版本，并且列出它的一些令人感兴趣的要点：

```
unsigned long __cdecl _beginthreadex (
   void *psa,
   unsigned cbStack,
   unsigned (__stdcall * pfnStartAddr) (void *),
   void * pvParam,
   unsigned fdwCreate,
   unsigned *pdwThreadID) {

   _ptiddata ptd;          // Pointer to thread's data block
   unsigned long thdl;     // Thread's handle

   // Allocate data block for the new thread.
   if ((ptd = _calloc_crt(1, sizeof(struct tiddata))) == NULL)
      goto error_return;

   // Initialize the data block.
   initptd(ptd);

   // Save the desired thread function and the parameter
   // we want it to get in the data block.
   ptd->_initaddr = (void *) pfnStartAddr;
   ptd->_initarg = pvParam;

   // Create the new thread.
   thdl = (unsigned long) CreateThread(psa, cbStack,
      _threadstartex, (PVOID) ptd, fdwCreate, pdwThreadID);
   if (thdl == NULL) {
      // Thread couldn't be created, cleanup and return failure.
```

China-pub.com 下载

```
        goto error_return;
    }

    // Create created OK, return the handle.
    return(thdl);

error_return:
    // Error: data block or thread couldn't be created.

    _free_crt(ptd);
    return((unsigned long)0L);
}
```

下面是关于_beginthreadex的一些要点：

- 每个线程均获得由C/C++运行期库的堆栈分配的自己的tiddata内存结构。（tiddata结构位于Mtdll.h文件中的Visual C++源代码中）。我在清单6-1中重建了它的结构。
- 传递给_beginthreadex的线程函数的地址保存在tiddata内存块中。传递给该函数的参数也保存在该数据块中。
- _beginthreadex确实从内部调用CreateThread，因为这是操作系统了解如何创建新线程的唯一方法。
- 当调用CreatetThread时，它被告知通过调用_threadstartex而不是pfnStartAddr来启动执行新线程。还有，传递给线程函数的参数是tiddata结构而不是pvParam的地址。
- 如果一切顺利，就会像CreateThread那样返回线程句柄。如果任何操作失败了，便返回NULL。

清单6-1 C/C++运行期库的线程局部tiddata结构

```
struct _tiddata {
    unsigned long   _tid;       /* thread ID */

    unsigned long   _thandle;   /* thread handle */

    int     _terrno;            /* errno value */
    unsigned long   _tdoserrno; /* _doserrno value */
    unsigned int    _fpds;      /* Floating Point data segment */
    unsigned long   _holdrand;  /* rand() seed value */
    char *      _token;         /* ptr to strtok() token */
#ifdef _WIN32
    wchar_t *   _wtoken;        /* ptr to wcstok() token */
#endif  /* _WIN32 */
    unsigned char * _mtoken;    /* ptr to _mbstok() token */

    /* following pointers get malloc'd at runtime */
    char *      _errmsg;        /* ptr to strerror()/_strerror() buff */
    char *      _namebuf0;      /* ptr to tmpnam() buffer */
#ifdef _WIN32
    wchar_t *   _wnamebuf0;     /* ptr to _wtmpnam() buffer */
#endif  /* _WIN32 */
    char *      _namebuf1;      /* ptr to tmpfile() buffer */
#ifdef _WIN32
    wchar_t *   _wnamebuf1;     /* ptr to _wtmpfile() buffer */
#endif  /* _WIN32 */
    char *      _asctimebuf;    /* ptr to asctime() buffer */
```

China-pub.com
下载


```
#ifdef _WIN32
    wchar_t *   _wasctimebuf;   /* ptr to _wasctime() buffer */
#endif  /* _WIN32 */
    void *      _gmtimebuf;     /* ptr to gmtime() structure */
    char *      _cvtbuf;        /* ptr to ecvt()/fcvt buffer */

    /* following fields are needed by _beginthread code */
    void *      _initaddr;      /* initial user thread address */
    void *      _initarg;       /* initial user thread argument */

    /* following three fields are needed to support signal handling and
    * runtime errors */
    void *      _pxcptacttab;   /* ptr to exception-action table */
    void *      _tpxcptinfoptrs; /* ptr to exception info pointers */
    int         _tfpecode;      /* float point exception code */

    /* following field is needed by NLG routines */
    unsigned long   _NLG_dwCode;

    /*
    * Per-Thread data needed by C++ Exception Handling
    */
    void *      _terminate;     /* terminate() routine */
    void *      _unexpected;    /* unexpected() routine */
    void *      _translator;    /* S.E. translator */
    void *      _curexception;  /* current exception */
    void *      _curcontext;    /* current exception context */
#if defined (_M_MRX000)
    void *      _pFrameInfoChain;
    void *      _pUnwindContext;
    void *      _pExitContext;
    int         _MipsPtdDelta;
    int         _MipsPtdEpsilon;
#elif defined (_M_PPC)
    void *      _pExitContext;
    void *      _pUnwindContext;
    void *      _pFrameInfoChain;
    int         _FrameInfo[6];
#endif  /* defined (_M_PPC) */
};

typedef struct _tiddata * _ptiddata;
```

既然为新线程指定了tiddata结构，并且对该结构进行了初始化，那么必须了解该结构与线程之间是如何关联起来的。让我们观察一下 _threadstartex函数（它也位于C/C++运行期库的Threadex.c文件中）。这里是该函数的伪代码版本：

```
static unsigned long WINAPI threadstartex (void* ptd) {
   // Note: ptd is the address of this thread's tiddata block.

   // Associate the tiddata block with this thread.
   TlsSetValue(__tlsindex, ptd);

   // Save this thread ID in the tiddata block.
   ((_ptiddata) ptd)->_tid = GetCurrentThreadId();
```

China-pub.com 下载

```
    // Initialize floating-point support (code not shown).

    // Wrap desired thread function in SEH frame to
    // handle run-time errors and signal support.
    __try {
       // Call desired thread function, passing it the desired parameter.
       // Pass thread's exit code value to _endthreadex.
       _endthreadex(
          ( (unsigned (WINAPI *)(void *))(((_ptiddata)ptd)->_initaddr) )
             ( ((_ptiddata)ptd)->_initarg ) ) ;
    }
    __except(_XcptFilter(GetExceptionCode(), GetExceptionInformation())){
       // The C run-time's exception handler deals with run-time errors
       // and signal support; we should never get it here.
       _exit(GetExceptionCode());
    }

    // We never get here; the thread dies in this function.
    return(0L);
}
```

下面是关于_threadstartex的一些重点：

- 新线程开始从 BasethreadStart函数（在 kernel32.dll文件中）执行，然后转移到 _threadstartex。
- 到达该新线程的tiddata块的地址作为其唯一参数被传递给 _threadstartex。
- TlsSetValue是个操作系统函数，负责将一个值与调用线程联系起来。这称为线程本地存储器（TLS），将在第21章介绍。_threadstartex函数将tiddata块与线程联系起来。
- 一个SEH帧被放置在需要的线程函数周围。这个帧负责处理与运行期库相关的许多事情——例如，运行期错误（比如放过了没有抓住的 C++异常条件）和 C/C++运行期库的signal函数。这是特别重要的。如果用 CreateThread函数来创建线程，然后调用 C/C++运行期库的signal函数，那么该函数就不能正确地运行。
- 调用必要的线程函数，传递必要的参数。记住，函数和参数的地址由 _beginthreadex保存在tiddata块中。
- 必要的线程函数返回值被认为是线程的退出代码。注意， _threadstartex并不只是返回到 BaseThreadStart。如果它准备这样做，那么线程就终止运行，它的退出代码将被正确地设置，但是线程的 tiddata内存块不会被撤消。这将导致应用程序中出现一个漏洞。若要防止这个漏洞，可以调用另一个C/C++运行期库函数_endthreadex,并传递退出代码。

需要介绍的最后一个函数是 _endthreadex（位于C运行期库的Threadex.c文件中）。下面是该函数的伪代码版本：

```
void __cdecl _endthreadex (unsigned retcode) {
   _ptiddata ptd;          // Pointer to thread's data block

   // Clean up floating-point support (code not shown).

   // Get the address of this thread's tiddata block.
   ptd = _getptd();

   // Free the tiddata block.
   _freeptd(ptd);
```

China-pub.com
下载


```
    // Terminate the thread.
    ExitThread(retcode);
}
```

下面是关于_endthreadex的一些要点：

- C运行期库的_getptd函数内部调用操作系统的TlsGetValue函数，该函数负责检索调用线程的tiddata内存块的地址。
- 然后该数据块被释放，而操作系统的ExitThread函数被调用，以便真正撤消该线程。当然，退出代码要正确地设置和传递。

本章前面说过，始终都应该设法避免使用ExitThread函数。这一点完全正确，我并不想收回我已经说过的话。ExitThread函数将撤消调用函数，并且不允许它从当前执行的函数返回。由于该函数不能返回，所以创建的任何C++对象都不会被撤消。避免调用ExitThread的另一个原因是，它会使得线程的tiddata内存块无法释放，这样，应用程序将会始终占用内存（直到整个进程终止运行为止）。

Microsoft的Visual C++开发小组认识到编程人员喜欢调用ExitThread，因此他们实现了他们的愿望，并且不会让应用程序始终占用内存。如果真的想要强制撤消线程，可以让它调用_endthreadex（而不是调用ExitThread）以便释放线程的tiddata块，然后退出。不过建议不要调用_endthreadex函数。

现在应该懂得为什么C/C++运行期库的函数需要为它创建的每个线程设置单独的数据块，同时，也应该了解如何通过调用_beginthreadex来分配数据块，再对它进行初始化，将该数据块与你创建的线程联系起来。你还应该懂得_endthreadex函数是如何在线程终止运行时释放数据块的。

一旦数据块被初始化并且与线程联系起来，线程调用的任何需要单线程实例数据的C/C++运行期库函数都能很容易地（通过TlsGetValue）检索调用线程的数据块地址，并对线程的数据进行操作。这对于函数来说很好，但是你可能想知道它对errno之类的全局变量效果如何。Errno定义在标准的C头文件中，类似下面的形式：

```
#if defined(_MT) || defined(_DLL)
extern int * __cdecl _errno(void);
#define errno (*_errno())
#else /* ndef _MT && ndef _DLL */
extern int errno;
#endif /* _MT || _DLL */
```

如果创建一个多线程应用程序，必须在编译器的命令行上设定/MT（指多线程应用程序）或/MD（指多线程DLL）开关。这将使编译器能够定义_MT标识符。然后，每当引用errno时，实际上是调用内部的C/C++运行期库函数_errno。该函数返回调用线程的相关数据块中的errno数据成员的地址。你将会发现，errno宏被定义为获取该地址的内容的宏。这个定义是必要的，因为可以编写类似下面形式的代码：

```
int *p = &errno;
if (*p == ENOMEM) {
    .
    .
    .
}
```

如果内部函数_errno只返回errno的值，那么上面的代码将不进行编译。

China-pub.com 下载

多线程版本的C/C++运行期库还给某些函数设置了同步的基本要素。例如，如果两个线程同时调用malloc，那么内存堆栈就可能遭到破坏。多线程版本的C/C++运行期库能够防止两个线程同时从堆栈中分配内存。为此，它要让第二个线程等待，直到第一个线程从malloc返回。然后第二个线程才被允许进入（关于线程同步的问题将在第8、9章和10章详细介绍）。

显然，所有这些附加操作都会影响多线程版本的C/C++运行期库的性能。这就是为什么Microsoft公司除了多线程版本外，还提供单线程版本的静态链接的C/C++运行期库的原因。

C/C++运行期库的动态连接版本编写成为一种通用版本。这样它就可以被使用C/C++运行期库函数的所有正在运行的应用程序和DLL共享。由于这个原因，运行期库只存在于多线程版本中。由于DLL中提供了C/C++运行期库，因此应用程序（.exe文件）和DLL不需要包含C/C++运行期库函数的代码，结果它们的规模就比较小。另外，如果Microsoft排除了C/C++运行期库DLL中的错误，应用程序中的错误也会自动得到解决。

正如希望的那样，C/C++运行期库的启动代码为应用程序的主线程分配了数据块，并且对数据块进行了初始化，这样，主线程就能安全地调用C/C++运行期函数中的任何函数。当主线程从它的进入点函数返回时，C/C++运行期库就会释放相关的数据块。此外，启动代码设置了相应的结构化异常处理代码，以便主线程能够成功地调用C/C++运行期库的signal函数。

6.7.1 Oops——错误地调用了CreateThread

也许你想知道，如果调用CreateThread，而不是调用C/C++运行期库的_beginthreadex来创建新线程，将会发生什么情况。当一个线程调用要求tiddata结构的C/C++运行期库函数时，将会发生下面的一些情况（大多数C/C++运行期库函数都是线程安全函数，不需要该结构）。首先，C/C++运行期库函数试图(通过调用TlsGetValue)获取线程的数据块的地址。如果返回NULL作为tiddata块的地址，调用线程就不拥有与该地址相关的tiddata块。这时，C/C++运行期库函数就在现场为调用线程分配一个tiddata块，并对它进行初始化。然后该tiddata块（通过TlsSetValue）与线程相关联。此时，只要线程在运行，该tiddata将与线程待在一起。这时，C/C++运行期库函数就可以使用线程的tiddata块，而且将来被调用的所有C/C++运行期函数也能使用tiddata块。

当然，这看来有些奇怪，因为线程运行时几乎没有任何障碍。不过，实际上还是存在一些问题。首先，如果线程使用C/C++运行期库的signal函数，那么整个进程就会终止运行，因为结构化异常处理帧尚未准备好。第二，如果不是调用_endthreadex来终止线程的运行，那么数据块就不会被撤消，内存泄漏就会出现（那么谁还为使用CreateThread函数创建的线程来调用_endthreadex呢？）。

注意 如果程序模块链接到多线程DLL版本的C/C++运行期库，那么当线程终止运行并释放tiddata块（如果已经分配了tiddata块的话）时，该运行期库会收到一个DLL_THREAD_DETACH通知。尽管这可以防止tiddata块的泄漏，但是强烈建议使用_bdginthreadex而不是使用Createthread来创建线程。

6.7.2 不应该调用的C/C++运行期库函数

C/C++运行期库也包含另外两个函数：

```
unsigned long _beginthread(
   void (__cdecl *start_address)(void *),
   unsigned stack_size,
```

China-pub.com
下载


```
    void *arglist);
```

和

```
void _endthread(void);
```

创建这两个函数的目的是用来执行_beginthreadex和_endthreadex函数的功能。但是，如你所见，_beginthread函数的参数比较少，因此比特性全面的_beginthreadex函数受到更大的限制。例如，如果使用_beginthread，就无法创建带有安全属性的新线程，无法创建暂停的线程，也无法获得线程的ID值。_endthread函数的情况与之类似。它不带参数，这意味着线程的退出代码必须硬编码为0。

endthread函数还存在另一个很难注意到的大问题。在_endthread调用ExitThread之前，它调用CloseHandle，传递新线程的句柄。若要了解为什么这是个大问题，请看下面的代码：

```
DWORD dwExitCode;
HANDLE hThread = _beginthread(...);
GetExitCodeThread(hThread, &dwExitCode);
CloseHandle(hThread);
```

新创建的线程可以在第一个线程调用GetExitCodeThread之前运行、返回和终止。如果出现这种情况，hThread中的值将无效，因为_endthread已经关闭了新线程的句柄。不用说，由于相同的原因，对CloseHandle的调用也将失败。

新的_endthreadex函数并不关闭线程的句柄，因此，如果用调用 beginthreadex来取代调用_beginthread，那么上面的代码段将能正确运行。记住，当线程函数返回时， _beginthrteadex调用_endthreadex，而_beginthread则调用_endthread。

6.8 对自己的ID概念应该有所了解

当线程运行时，它们常常想要调用Windows函数来改变它们的运行环境。例如，线程可能想要改变它的优先级或它的进程的优先级（优先级将在第7章中介绍）。由于线程常常要改变它的（或它的进程的）环境，因此Windows提供了一些函数，使线程能够很容易引用它的进程内核对象，或者引用它自己的线程内核对象：

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

上面这两个函数都能返回调用线程的进程的伪句柄或线程内核对象的伪句柄。这些函数并不在创建进程的句柄表中创建新句柄。还有，调用这些函数对进程或线程内核对象的使用计数没有任何影响。如果调用CloseHandle，将伪句柄作为参数来传递，那么CloseHandle就会忽略该函数的调用并返回FALSE。

当调用一个需要进程句柄或线程句柄的Windows函数时，可以传递一个伪句柄，使该函数执行它对调用进程或线程的操作。例如，通过调用下面的GetProcessTimes函数，线程可以查询它的进程的时间使用情况：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
   &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

同样，通过调用GetThreadTimes函数，线程可以查询它自己的线程时间：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(GetCurrentThread(),
   &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

少数Windows函数允许用进程或线程在系统范围内独一无二的 ID来标识某个进程或线程。

China-pub.com 下载

下面这两个函数使得线程能够查询它的进程的唯一ID或它自己的唯一ID：

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

这两个函数通常不像能够返回伪句柄的函数那样有用，但是有的时候用起来还是很方便的。

将伪句柄转换为实句柄

有时可能需要获得线程的实句柄而不是它的伪句柄。所谓“实句柄”，我是指用来明确标识一个独一无二的线程的句柄。请看下面的代码：

```
DWORD WINAPI ParentThread(PVOID pvParam) {
   HANDLE hThreadParent = GetCurrentThread();
   CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
   // Function continues...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
   HANDLE hThreadParent = (HANDLE) pvParam;
   FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
   GetThreadTimes(hThreadParent,
      &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
   // Function continues...
}
```

你能发现这个代码段存在的问题吗？这个代码的目的是让父线程给子线程传递一个线程句柄，以标识父线程。但是，父线程传递了一个伪句柄，而不是一个实句柄。当子线程开始运行时，它将一个伪句柄传递给GetThreadTime函数，使子线程得到它自己的CPU时间，而不是父线程的CPU时间。出现这种情况的原因是线程的伪句柄是当前线程的句柄，也就是说，它是调用函数的线程的句柄。

为了修改这个代码，必须将伪句柄变成实句柄。DuplicateHandle函数能够执行这一转换：

```
BOOL DuplicateHandle(
   HANDLE hSourceProcess,
   HANDLE hSource,
   HANDLE hTargetProcess,
   PHANDLE phTarget,
   DWORD fdwAccess,
   BOOL bInheritHandle,
   DWORD fdwOptions);
```

通常可以使用这个函数，用与另一个进程相关的内核对象来创建一个与进程相关的新句柄。然而，可以用一种特殊的方法来使用这个函数，以便修改上面介绍的代码段。正确的代码段应该是下面的样子：

```
DWORD WINAPI ParentThread(PVOID pvParam) {
   HANDLE hThreadParent;

   DuplicateHandle(
      GetCurrentProcess(),     // Handle of process that thread
                               // pseudo-handle is relative to
      GetCurrentThread(),      // Parent thread's pseudo-handle
      GetCurrentProcess(),     // Handle of process that the new, real,
                               // thread handle is relative to
      &hThreadParent,          // Will receive the new, real, handle
```

China-pub.com
下载


```
                                    // identifying the parent thread
      0,                            // Ignored due to DUPLICATE_SAME_ACCESS
      FALSE,                        // New thread handle is not inheritable
      DUPLICATE_SAME_ACCESS);       // New thread handle has same
                                    // access as pseudo-handle

   CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
   // Function continues...
}
DWORD WINAPI ChildThread(PVOID pvParam) {
   HANDLE hThreadParent = (HANDLE) pvParam;
   FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
   GetThreadTimes(hThreadParent,
      &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
   CloseHandle(hThreadParent);
   // Function continues...
}
```

当父线程运行时，它就将标识父线程所用的伪句柄转换成明确标识父线程所用的新的实句柄。同时它将这个实句柄传递给CreateThread。当子线程启动运行时，它的pvParam参数包含了线程的实句柄。对传递该句柄的函数的任何调用都将影响父线程而不是子线程。

由于DuplicateHandle会递增特定对象的使用计数，因此当完成对复制对象句柄的使用时，应该将目标句柄传递给CloseHandle，从而递减对象的使用计数，这一点很重要。上面的代码段已经显示出这一点。在调用GetThreadTimes之后，紧接着子线程调用CloseHandle，以便递减父线程对象的使用计数。在这个代码段中，我假设子线程不使用该句柄来调用任何其他函数。如果其他函数被调用，以便传递父线程的句柄，那么在子线程不再需要该句柄之前，不应该调用CloseHandle。

还要指出，DuplicateHandle可以用来将进程的伪句柄转换成进程的实句柄，如下面的代码所示：

```
HANDLE hProcess;
DuplicateHandle(
   GetCurrentProcess(),     // Handle of process that the process
                            // pseudo-handle is relative to
   GetCurrentProcess(),     // Process's pseudo-handle
   GetCurrentProcess(),     // Handle of process that the new, real,
                            // process handle is relative to
   &hProcess,               // Will receive the new, real
                            // handle identifying the process
   0,                       // Ignored because of DUPLICATE_SAME_ACCESS
   FALSE,                   // New thread handle is not inheritable
   DUPLICATE_SAME_ACCESS); // New process handle has same
                            // access as pseudo-handle
```

China-pub.com
下载

第7章　线程的调度、优先级和亲缘性

抢占式操作系统必须使用某种算法来确定哪些线程应该在何时调度和运行多长时间。本章将要介绍Microsoft Windows 98和Windows 2000使用的一些算法。

上一章介绍了每个线程是如何拥有一个上下文结构的，这个结构维护在线程的内核对象中。这个上下文结构反映了线程上次运行时该线程的CPU寄存器的状态。每隔20ms左右，Windows要查看当前存在的所有线程内核对象。在这些对象中，只有某些对象被视为可以调度的对象。Windows选择可调度的线程内核对象中的一个，将它加载到CPU的寄存器中，它的值是上次保存在线程的环境中的值。这项操作称为上下文转换。Windows实际上保存了一个记录，它说明每个线程获得了多少个运行机会。使用Microsoft Spy++这个工具，就可以了解这个情况。图7-1显示了一个线程的属性。注意，该线程已经被调度了37 379次。

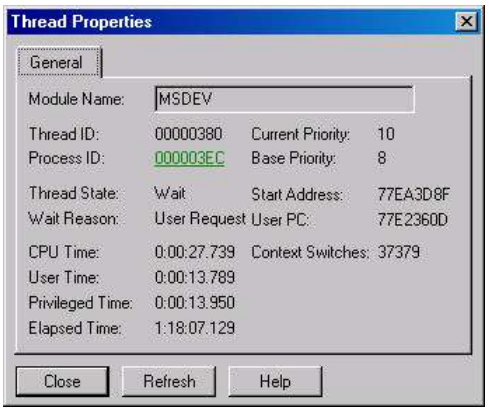


图7-1　线程的属性

目前，线程正在执行代码，并对它的进程的地址空间中的数据进行操作。再过20ms左右，Windows就将CPU的寄存器重新保存到线程的上下文中。线程不再运行。系统再次查看其余的可调度线程内核对象，选定另一个线程的内核对象，将该线程的上下文加载到CPU的寄存器中，然后继续运行。当系统引导时，便开始加载线程的上下文，让线程运行，保存上下文和重复这些操作，直到系统关闭。

总之，这就是系统对线程进行调度的过程。这很简单，是不是？Windows被称为抢占式多线程操作系统，因为一个线程可以随时停止运行，随后另一个线程可进行调度。如你所见，可以对它进行一定程度的控制，但是不能太多。记住，无法保证线程总是能够运行，也不能保证线程能够得到整个进程，无法保证其他线程不被允许运行等等。

> **注意**　程序员常常问我，如何才能保证线程在某个事件的某个时间段内开始运行，比如，如何才能确保某个线程在数据从串行端口传送过来的1ms内开始运行呢？我的回答是，办不到。实时操作系统才能作出这样的承诺，但是Windows不是实时操作系统。实时操作系统必须清楚地知道它是在什么硬件上运行，这样它才能知道它的硬盘控制器和键盘等的等待时间。Microsoft对Windows规定的目标是，使它能够在各种不同的硬件上运行，即能够在不同的CPU、不同的驱动器和不同的网络上运行。简而言之，Windows没有设计成为一种实时操作系统。

尽管应强调这样一个概念，即系统只调度可以调度的线程，但是实际情况是，系统中的大多数线程是不可调度的线程。例如，有些线程对象的暂停计数大于1。这意味着该线程已经暂停运行，不应该给它安排任何CPU时间。通过调用使用CREATE_SUSPENDED标志的CreateProcess或CreateThread函数，可以创建一个暂停的线程。（本章后面还要介绍Suspend

China-pub.com
下载

Thread和ResumeThread函数。）

除了暂停的线程外，其他许多线程也是不可调度的线程，因为它们正在等待某些事情的发生。例如，如果运行Notepad，但是并不键入任何数据，那么Notepad的线程就没有什么事情要做。系统不给无事可做的线程分配CPU时间。当移动Notepad的窗口时，或者Notepad的窗口需要刷新它的内容，或者将数据键入Notepad，系统就会自动使Notepad的线程成为可调度的线程。这并不意味着Notepad的线程立即获得了CPU时间。它只是表示Notepad的线程有事情可做，系统将设法在某个时间（不久的将来）对它进行调度。

7.1 暂停和恢复线程的运行

在线程内核对象的内部有一个值，用于指明线程的暂停计数。当调用 CreateProcess或CreateThread函数时，就创建了线程的内核对象，并且它的暂停计数被初始化为1。这可以防止线程被调度到CPU中。当然，这是很有用的，因为线程的初始化需要时间，你不希望在系统做好充分的准备之前就开始执行线程。

当线程完全初始化好了之后， CreateProcess或CreateThread要查看是否已经传递了CREATE_ SUSPENDED标志。如果已经传递了这个标志，那么这些函数就返回，同时新线程处于暂停状态。如果尚未传递该标志，那么该函数将线程的暂停计数递减为 0。当线程的暂停计数是0的时候，除非线程正在等待其他某种事情的发生，否则该线程就处于可调度状态。

在暂停状态中创建一个线程，就能够在线程有机会执行任何代码之前改变线程的运行环境（如优先级）。一旦改变了线程的环境，必须使线程成为可调度线程。要进行这项操作，可以调用ResumeThread，将调用CreateThread函数时返回的线程句柄传递给它（或者是将传递给CreateProcess的ppiProcInfo参数指向的线程句柄传递给它）：

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

如果ResumeThread函数运行成功，它将返回线程的前一个暂停计数，否则返回0xFFFFFFFF。

单个线程可以暂停若干次。如果一个线程暂停了3次，它必须恢复3次，然后它才可以被分配给一个CPU。当创建线程时，除了使用 CREATE_SUSPENDED外，也可以调用 SuspendThread函数来暂停线程的运行：

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

任何线程都可以调用该函数来暂停另一个线程的运行（只要拥有线程的句柄）。不用说，线程可以自行暂停运行，但是不能自行恢复运行。与 ResumeThread一样，SuspendThread返回的是线程的前一个暂停计数。线程暂停的最多次数可以是 MAXIMUM_SUSPEND_COUNT次（在WinNT.h中定义为127）。注意，SuspendThread与内核方式的执行是异步进行的，但是在线程恢复运行之前，不会发生用户方式的执行。

在实际环境中，调用SuspendThread时必须小心，因为不知道暂停线程运行时它在进行什么操作。如果线程试图从堆栈中分配内存，那么该线程将在该堆栈上设置一个锁。当其他线程试图访问该堆栈时，这些线程的访问就被停止，直到第一个线程恢复运行。只有确切知道目标线程是什么（或者目标线程正在做什么），并且采取强有力的措施来避免因暂停线程的运行而带来的问题或死锁状态，SuspendThread才是安全的（死锁和其他线程同步问题将在第8、9和10章介绍）。

China-pub.com 下载

7.2 暂停和恢复进程的运行

对于Windows来说，不存在暂停或恢复进程的概念，因为进程从来不会被安排获得 CPU时间。但是，曾经有人无数次问我如何暂停进程中的所有线程的运行。 Windows确实允许一个进程暂停另一个进程中的所有线程的运行，但是从事暂停操作的进程必须是个调试程序。特别是，进程必须调用WaitForDebugEvent和ContinueDebugEvent之类的函数。

由于竞争的原因，Windows没有提供其他方法来暂停进程中所有线程的运行。例如，虽然许多线程已经暂停，但是仍然可以创建新线程。从某种意义上说，系统必须在这个时段内暂停所有新线程的运行。Microsoft已经将这项功能纳入了系统的调试机制。

虽然无法创建绝对完美的SuspendProcess函数，但是可以创建一个该函数的实现代码，它能够在许多条件下出色地运行。下面是我的SuspendProcess函数的实现代码：

```
VOID SuspendProcess(DWORD dwProcessID, BOOL fSuspend) {

   // Get the list of threads in the system.
   HANDLE hSnapshot = CreateToolhelp32Snapshot(
      TH32CS_SNAPTHREAD, dwProcessID);

   if (hSnapshot != INVALID_HANDLE_VALUE) {

      // Walk the list of threads.
      THREADENTRY32 te = { sizeof(te) };
      BOOL fOk = Thread32First(hSnapshot, &te);
      for (; fOk; fOk = Thread32Next(hSnapshot, &te)) {

         // Is this thread in the desired process?
         if (te.th32OwnerProcessID == dwProcessID) {

            // Attempt to convert the thread ID into a handle.
            HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,
               FALSE, te.th32ThreadID);

            if (hThread != NULL) {

               // Suspend or resume the thread.
               if (fSuspend)
                  SuspendThread(hThread);
               else
                  ResumeThread(hThread);
            }
            CloseHandle(hThread);
         }
      }
      CloseHandle(hSnapshot);
   }
}
```

我的SuspendProcess函数使用ToolHelp函数来枚举系统中的线程列表。当我找到作为指定进程的组成部分的线程时，我调用OpenThread：

```
HANDLE OpenThread(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   DWORD dwThreadID);
```

China-pub.com
下载

这个新Windows 2000函数负责找出带有匹配的线程ID的线程内核对象，对内核对象的使用计数进行递增，然后返回对象的句柄。运用这个句柄，我调用SuspendThread(或ResumeThread)。由于OpenThread在Windows 2000中是个新函数，因此我的SuspendProcess函数在Windows 95或Windows 98上无法运行，在Windows NT 4.0或更早的版本上也无法运行。

也许你懂得为什么SuspendProcess不能总是运行，原因是当枚举线程组时，新线程可以被创建和撤消。因此，当我调用CreateToolhelp32Snapshot后，一个新线程可能会出现在目标进程中，我的函数将无法暂停这个新线程。过了一些时候，当调用SuspendProcess函数来恢复线程的运行时，它将恢复它从未暂停的一个线程的运行。更糟糕的是，当枚举线程ID时，一个现有的线程可能被撤消，一个新线程可能被创建，这两个线程可能拥有相同的ID。这将会导致该函数暂停任意些个（也许在目标进程之外的一个进程中的）线程的运行。

当然，这些情况不太可能出现。如果非常了解目标进程是如何运行的，那么这些问题也许根本不是问题。我提供这个函数供酌情使用。

7.3 睡眠方式

线程也能告诉系统，它不想在某个时间段内被调度。这是通过调用Sleep函数来实现的：

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

该函数可使线程暂停自己的运行，直到dwMilliseconds过去为止。关于Sleep函数，有下面几个重要问题值得注意：

- 调用Sleep，可使线程自愿放弃它剩余的时间片。
- 系统将在大约的指定毫秒数内使线程不可调度。不错，如果告诉系统，想睡眠100ms，那么可以睡眠大约这么长时间，但是也可能睡眠数秒钟或者数分钟。记住，Windows不是个实时操作系统。虽然线程可能在规定的时间被唤醒，但是它能否做到，取决于系统中还有什么操作正在进行。
- 可以调用Sleep，并且为dwMilliseconds参数传递INFINITE。这将告诉系统永远不要调度该线程。这不是一件值得去做的事情。最好是让线程退出，并还原它的堆栈和内核对象。
- 可以将0传递给Sleep。这将告诉系统，调用线程将释放剩余的时间片，并迫使系统调度另一个线程。但是，系统可以对刚刚调用Sleep的线程重新调度。如果不存在多个拥有相同优先级的可调度线程，就会出现这种情况。

7.4 转换到另一个线程

系统提供了一个称为SwitchToThread的函数，使得另一个可调度线程（如果存在能够运行）：

```
BOOL SwitchToThread();
```

当调用这个函数的时候，系统要查看是否存在一个迫切需要CPU时间的线程。如果没有线程迫切需要CPU时间，SwitchToThread就会立即返回。如果存在一个迫切需要CPU时间的线程，SwitchToThread就对该线程进行调度（该线程的优先级可能低于调用SwitchToThread的线程）。这个迫切需要CPU时间的线程可以运行一个时间段，然后系统调度程序照常运行。

该函数允许一个需要资源的线程强制另一个优先级较低、而目前却拥有该资源的线程放弃该资源。如果调用SwitchToThread函数时没有其他线程能够运行，那么该函数返回FALSE，否则返回一个非0值。

China-pub.com 下载

调用SwitchToThread函数与调用Sleep是相似的，并且传递给它一个0ms的超时。差别是SwitchToThread允许优先级较低的线程运行。即使低优先级线程迫切需要CPU时间，Sleep也能够立即对调用线程重新进行调度。

Windows 98 Windows 98没有配备该函数的非常有用的实现代码。

7.5 线程的运行时间

有时想要计算线程执行某个任务需要多长的时间。许多人采取的办法是编写类似下面的代码：

```
// Get the current time (start time).
DWORD dwStartTime = GetTickCount();

// Perform complex algorithm here.

// Subtract start time from current time to get duration.
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

这个代码做了一个简单的假设：即它不会被中断。但是，在抢占式操作系统中，永远无法知道线程何时被赋予CPU时间。当取消线程的CPU时间时，就更难计算线程执行不同任务时所用的时间。我们需要一个函数，以便返回线程得到的CPU时间的数量。幸运的是，Windows提供了一个称为GetThreadTimes的函数，它能返回这些信息：

```
BOOL GetThreadTimes(
   HANDLE hThread,
   PFILETIME pftCreationTime,
   PFILETIME pftExitTime,
   PFILETIME pftKernelTime,
   PFILETIME pftUserTime);
```

GetThreadTimes函数返回4个不同的时间值，这些值如表7-1所示。

表7-1 GetThreadTimes 函数的返回时间值

时 间 值	含 义
创建时间	用英国格林威治时间1601年1月1日午夜后100ns的时间间隔表示的英国绝对值，用于指明线程创建的时间
退出时间	用英国格林威治时间1601年1月1日午夜后100ns的时间间隔表示的英国绝对值，用于指明线程退出的时间。如果线程仍然在运行，退出时间则未定义
内核时间	一个相对值，用于指明线程执行操作系统代码已经经过了多少个100ns的CPU时间
用户时间	一个相对值，用于指明线程执行应用程序代码已经经过了多少个100ns的CPU时间

使用这个函数，可以通过使用下面的代码确定执行复杂的算法时需要的时间量：

```
__int64 FileTimeToQuadWord (PFILETIME pft) {
   return(Int64ShllMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}


void PerformLongOperation () {
```

China-pub.com
下载


```
   FILETIME ftKernelTimeStart, ftKernelTimeEnd;
   FILETIME ftUserTimeStart,   ftUserTimeEnd;
   FILETIME ftDummy;
   __int64 qwKernelTimeElapsed, qwUserTimeElapsed,
      qwTotalTimeElapsed;

   // Get starting times.
   GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
      &ftKernelTimeStart, &ftUserTimeStart);


   // Perform complex algorithm here.


   // Get ending times.
   GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
      &ftKernelTimeEnd, &ftUserTimeEnd);

   // Get the elapsed kernel and user times by converting the start
   // and end times from FILETIMEs to quad words, and then subtract
   // the start times from the end times.
   qwKernelTimeElapsed = FileTimeToQuadWord(&ftKernelTimeEnd) -
      FileTimeToQuadWord(&ftKernelTimeStart);

   qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd) -
      FileTimeToQuadWord(&ftUserTimeStart);

   // Get total time duration by adding the kernel and user times.
   qwTotalTimeElapsed = qwKernelTimeElapsed + qwUserTimeElapsed;

   //-The-total-elapsed-time-is-in-qwTotalTimeElapsed.
}
```

注意，GetProcessTimes是个类似GetThreadTimes的函数，适用于进程中的所有线程：

```
BOOL GetProcessTimes(
   HANDLE hProcess,
   PFILETIME pftCreationTime,
   PFILETIME pftExitTime,
   PFILETIME pftKernelTime,
   PFILETIME pftUserTime);
```

GetProcessTimes返回的时间适用于某个进程中的所有线程（甚至是已经终止运行的线程）。例如，返回的内核时间是所有进程的线程在内核代码中经过的全部时间的总和。

Windows 98 遗憾的是，GetThreadTimes和GetProcessTimes这两个函数在Windows 98中不起作用。在Windows 98中，没有一个可靠的机制可供应用程序来确定线程或进程已经使用了多少CPU时间。

对于高分辨率的配置文件来说，GetThreadTimes并不完美。Windows确实提供了一些高分辨率性能函数：

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);

BOOL QueryPerformanceCounter(LARGE_INTEGER* pliCount);
```

虽然这些函数认为，正在执行的线程并没有得到抢占的机会，但是高分辨率的配置文件是

China-pub.com 下载

为短期存在的代码块设置的。为了使这些函数运行起来更加容易一些，我创建了下面这个 C++ 类：

```
class CStopwatch {
public:
   CStopwatch() { QueryPerformanceFrequency(&m_liPerfFreq); Start();

   void Start() { QueryPerformanceCounter(&m_liPerfStart); }

   __int64 Now() const {   //
 Returns # of milliseconds since Start was called
      LARGE_INTEGER liPerfNow;
      QueryPerformanceCounter(&liPerfNow);
      return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)
         / m_liPerfFreq.QuadPart);
   }

private:
   LARGE_INTEGER m_liPerfFreq;   // Counts per second
   LARGE_INTEGER m_liPerfStart;  // Starting count
};
```

使用这个类如下：

```
// Create a stopwatch timer (which defaults to the current time).
CStopwatch stopwatch;

// Execute the code I want to profile here.

// Get how much time has elapsed up to now.
 __int64 qwElapsedTime = stopwatch.Now();

// qwElapsedTime indicates how long the profiled code
// executed in milliseconds.
```

7.6 运用结构环境

现在应该懂得环境结构在线程调度中所起的重要作用了。环境结构使得系统能够记住线程的状态，这样，当下次线程拥有可以运行的CPU时，它就能够找到它上次中断运行的地方。

知道这样低层的数据结构也会完整地记录在 Platform SDK文档中确实使人吃惊。不过如果查看该文档中的CONTEXT结构，会看到下面这段文字：

“CONTEXT结构包含了特定处理器的寄存器数据。系统使用 CONTEXT结构执行各种内部操作。目前，已经存在为 Intel、MIPS、Alpha和PowerPC处理器定义的CONTEXT结构。若要了解这些结构的定义，参见头文件WinNT.h”。

该文档并没有说明该结构的成员，也没有描述这些成员是谁，因为这些成员要取决于Windows 2000在哪个CPU上运行。实际上，在Windows定义的所有数据结构中，CONTEXT结构是特定于CPU的唯一数据结构。

那么CONTEXT结构中究竟存在哪些东西呢？它包含了主机 CPU上的每个寄存器的数据结构。在x86计算机上，数据成员是Eax、Ebx、Ecx、Edx等等。如果是Alpha处理器，那么数据成员包括IntV0、IntT0、IntT1、IntS0、IntRa和IntZero等等。下面这个代码段显示了 x86 CPU的完整的CONTEXT结构：

China-pub.com
下载


```
typedef struct _CONTEXT {

    //
    // The flags values within this flag control the contents of
    // a CONTEXT record.
    //
    // If the context record is used as an input parameter, then
    // for each portion of the context record controlled by a flag
    // whose value is set, it is assumed that that portion of the
    // context record contains valid context. If the context record
    // is being used to modify a threads context, then only that
    // portion of the threads context will be modified.
    //
    // If the context record is used as an IN OUT parameter to capture
    // the context of a thread, then only those portions of the thread's
    // context corresponding to set flags will be returned.
    //
    // The context record is never used as an OUT only parameter.
    //

    DWORD ContextFlags;

    //
    // This section is specified/returned if CONTEXT_DEBUG_REGISTERS is
    // set in ContextFlags.  Note that CONTEXT_DEBUG_REGISTERS is NOT
    // included in CONTEXT_FULL.
    //

    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;

    //
    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_FLOATING_POINT.
    //

    FLOATING_SAVE_AREA FloatSave;

    //
    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_SEGMENTS.
    //

    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;

    //
    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_INTEGER.
```

China-pub.com 下载

```
    //

    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;
    //
    // This section is specified/returned if the
    // ContextFlags word contians the flag CONTEXT_CONTROL.
    //

    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;              // MUST BE SANITIZED
    DWORD   EFlags;             // MUST BE SANITIZED
    DWORD   Esp;
    DWORD   SegSs;

    //
    // This section is specified/returned if the ContextFlags word
    // contains the flag CONTEXT_EXTENDED_REGISTERS.
    // The format and contexts are processor specific
    //

    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];

} CONTEXT;
```

CONTEXT结构可以分成若干个部分。CONTEXT_CONTROL包含CPU的控制寄存器，比如指令指针、堆栈指针、标志和函数返回地址（与x86处理器不同，Alpya CPU在调用函数时，将该函数的返回地址放入一个寄存器中）。CONTEXT_INTEGER用于标识CPU的整数寄存器。CONTEXT_FLOATING_POINT用于标识CPU的浮点寄存器。CONTEXT_SEGMENTS用于标识CPU的段寄存器（仅为x86处理器）。CONTEXT_DEBUG_ REGISTER用于标识CPU的调试寄存器（仅为x86处理器）。CONTEXT_EXTENDED_ REGISTERS用于标识CPU的扩展寄存器（仅为x86处理器）。

Windows实际上允许查看线程内核对象的内部情况，以便抓取它当前的一组 CPU寄存器。若要进行这项操作，只需要调用GetThreadContext函数：

```
BOOL GetThreadContext(
   HANDLE hThread,
   PCONTEXT pContext);
```

若要调用该函数，只需指定一个CONTEXT结构，对某些标志（该结构的ContextFlags成员）进行初始化，指明想要收回哪些寄存器，并将该结构的地址传递给 GetThreadContext。然后该函数将数据填入你要求的成员。

在调用GetThreadContext函数之前，应该调用SuspendThread，否则，线程可能被调度，而且线程的环境可能与你收回的不同。一个线程实际上有两个环境。一个是用户方式，一个是内核方式。GetThreadContext只能返回线程的用户方式环境。如果调用 SuspendThread来停止线程的运行，但是该线程目前正在用内核方式运行，那么，即使 SuspendThread实际上尚未暂停该

China-pub.com
下载

线程的运行，它的用户方式仍然处于稳定状态。线程在恢复用户方式之前，它无法执行更多的用户方式代码，因此可以放心地将线程视为处于暂停状态， GetThreadContext函数将能正常运行。

CONTEXT结构的ContextFlags成员并不与任何CPU寄存器相对应。无论是何种CPU结构，该成员存在于所有CONTEXT结构定义中。ContextFlags成员用于向GetThreadContext函数指明你想检索哪些寄存器。例如，如果想要获得线程的控制寄存器，可以编写下面的代码：

```
// Create a CONTEXT structure.
CONTEXT Context;

// Tell the system that we are interested in only the
// control registers.
Context.ContextFlags = CONTEXT_CONTROL;

// Tell the system to get the registers associated with a thread.
GetThreadContext(hThread, &Context);

// The control register members in the CONTEXT structure
// reflect the thread's control registers. The other members
// are undefined.
```

注意，在调用GetThreadContext之前，首先必须对CONTEXT结构中的ContextFlags成员进行初始化。如果想要获得线程的控制寄存器和整数寄存器，应该像下面这样对 ContexFlags进行初始化：

```
// Tell the system that we are interested
// in the control and integer registers.
Context.ContextFlags = CONTEXT_CONTROL | CONTEXT_INTEGER;
```

下面是一些标识符，使用这些标识符可以获得线程的所有重要的寄存器（即 Microsoft视为最常用的那些寄存器）：

```
// Tell the system we are interested in the important registers.
Context.ContextFlags = CONTEXT_FULL;
```

在WinNT.h文件中，定义了CONTEXT_FULL，请看表7-2。

表7-2 CONTEXT-FULL的意义

CPU类型	CONTEXT_FULL的定义
X86	CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS
Alpha	CONTEXT_CONTROL | CONTEXT_FLOATING_POINT | CONTEXT_INTEGER

当GetThreadContext返回时，能够很容易地查看线程的任何寄存器的值，但是要记住，这意味着必须编写与CPU相关的代码。表7-3根据CPU类型列出了CONTEXT结构的指令指针和堆栈指针。

表7-3 CONTEXT 结构的指令指针和堆栈指针

CPU类型	指令指针	堆栈指针
X86	CONTEXT.Eip	CONTEXT.Esp
Alpha	CONTEXT.Fir	CONTEXT.IntSp

Windows为编程人员提供了多么强大的功能啊！如果你认为它确实不错，那么你一定会喜

China-pub.com 下载

欢它的，因为Windows使你能够修改CONTEXT结构中的成员，然后通过调用SetThreadContext将新寄存器值放回线程的内核对象中：

```
BOOL SetThreadContext(
   HANDLE hThread,
   CONST CONTEXT *pContext);
```

同样，修改其环境的线程应该首先暂停，否则其结果将无法预测。

在调用SetThreadContext之前，必须再次对CONTEXT的ContextFlags成员进行初始化，如下面的代码所示：

```
CONTEXT Context;

// Stop the thread from running.
SuspendThread(hThread);

// Get the thread's context registers.
Context.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(hThread, &Context);
// Make the instruction pointer point to the address of your choice.
// Here I've arbitrarily set the address instruction pointer to
// 0x00010000.
#if defined(_ALPHA_)
Context.Fir = 0x00010000;
#elif defined(_X86_)
Context.Eip = 0x00010000;
#else
#error Module contains CPU-specific code; modify and recompile.
#endif


// Set the thread's registers to reflect the changed values.
// It's not really necessary to reset the ControlFlags member
// because it was set earlier.
Context.ControlFlags = CONTEXT_CONTROL;
SetThreadContext(hThread, &Context);

// Resuming the thread will cause it to begin execution
// at address 0x00010000.
ResumeThread(hThread);
```

这有可能导致远程线程中的访问违规，向用户显示未处理的异常消息框，同时，远程进程终止运行。你将成功地终止另一个进程的运行，而你的进程则可以继续很好地运行。

GetThreadContext和SetThreadContext函数使你能够对线程进行许多方面的控制，但是在使用它们时应该小心。实际上，几乎没有应用程序调用这些函数。增加这些函数是为了增强调试程序和其他工具的功能。任何应用程序都可以调用它们。

第24章将详细地介绍CONTEXT结构。

7.7 线程的优先级

本章开头讲述了CPU是如何只使线程运行20ms，然后调度程序将另一个可调度的线程分配给CPU的。如果所有线程具有相同的优先级，那么就会发生这种情况，但是，在现实环境中，线程被赋予许多不同的优先级，这会影响到调度程序将哪个线程取出来作为下一个要运

China-pub.com
下载

行的线程。

每个线程都会被赋予一个从0（最低）到31（最高）的优先级号码。当系统确定将哪个线程分配给CPU时，它首先观察优先级为31的线程，并以循环方式对它们进行调度。如果优先级为31的线程可以调度，那么就将该线程赋予一个CPU。在该线程的时间片结束时，系统要查看是否还有另一个优先级为31的线程可以运行，如果有，它将允许该线程被赋予一个CPU。

只要优先级为31的线程是可调度的，系统就绝对不会将优先级为0到30的线程分配给CPU。这种情况称为渴求调度（starvation）。当高优先级线程使用大量的CPU时间，从而使得低优先级线程无法运行时，便会出现渴求情况。在多处理器计算机上出现渴求情况的可能性要少得多，因为在这样的计算机上，优先级为31和优先级为30的线程能够同时运行。系统总是设法使CPU保持繁忙状态，只有当没有线程可以调度的时候，CPU才处于空闲状态。

人们可能认为，在这样的系统中，低优先级线程永远得不到机会运行。不过正像前面指出的那样，在任何一个时段内，系统中的大多数线程是不能调度的。例如，如果进程的主线程调用GetMessage函数，而系统发现没有线程可以供它使用，那么系统就暂停进程的线程运行，释放该线程的剩余时间片，并且立即将CPU分配给另一个等待运行的线程。

如果没有为GetMessage函数显示可供检索的消息，那么进程的线程将保持暂停状态，并且决不会被分配给CPU。但是，当消息被置于线程的队列中时，系统就知道该线程不应该再处于暂停状态。此时，如果没有更高优先级的线程需要运行，系统就将该线程分配给一个CPU。

现在考虑另一个问题。高优先级线程将抢在低优先级线程之前运行，不管低优先级线程正在运行什么。例如，如果一个优先级为5的线程正在运行，系统发现一个高优先级的线程准备要运行，那么系统就会立即暂停低优先级线程的运行（即使它处于它的时间片中），并且将CPU分配给高优先级线程，使它获得一个完整的时间片。

还有，当系统引导时，它会创建一个特殊的线程，称为0页线程。该线程被赋予优先级0，它是整个系统中唯一的一个在优先级0上运行的线程。当系统中没有任何线程需要执行操作时，0页线程负责将系统中的所有空闲RAM页面置0。

7.8 对优先级的抽象说明

当Microsoft的开发人员设计线程调度程序时，他们发现该调度程序无法在所有时间适应所有人的需要。他们还发现，计算机的“作用”是不断变化的。当Windows NT问世时，对象链接和嵌入（OLE）应用程序还刚刚开始编写。现在，OLE应用程序已经司空见惯。游戏软件已经相当流行。当然，在Windows NT的早期，并没有更多地考虑Internet的问题。

调度算法对用户运行的应用程序类型有着相当大的影响。从一开始，Microsoft的开发人员就认识到，随着系统的用途的变化，他们必须不断修改调度算法。但是，软件开发人员需要在今天编写软件，而Microsoft则要保证软件能够在将来的系统版本上运行。那么Microsoft如何改变系统工作的方式并仍然保证软件能够运行呢？下面是解决这个问题的一些办法：

- Microsoft没有将调度程序的行为特性完全固定下来。
- Microsoft没有让应用程序充分利用调度程序的特性。
- Microsoft声称调度程序的算法是变化的，在编写代码时应有所准备。

Windows API展示了系统的调度程序上的一个抽象层，这样就永远不会直接与调度程序进行通信。相反，要调用Windows函数，以便根据运行的系统版本“转换”参数。本章将介绍这个抽象层。

当设计一个应用程序时，你应该考虑到还有什么别的应用程序会与你的应用程序一道运行。

China-pub.com 下载

然后，应该根据你的应用程序中的线程应该具备何种响应性，选择一个优先级类。这听起来有些费解，不过情况确实如此。Microsoft不想作出任何将来可能影响你的代码运行的承诺。

Windows支持6个优先级类：即空闲、低于正常、正常、高于正常、高和实时。当然，正常优先级是最常用的优先级类，99%的应用程序均使用这个优先级类。表7-4描述了这些优先级类。

表7-4 Windows支持的优先级类

优先级类	描述
实时	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。该进程中的线程还会抢先于操作系统组件之前运行。使用本优先级类时必须极端小心
高	进程中的线程必须立即对事件作出响应，以便执行关键时间的任务。Task Manager（任务管理器）在这个类上运行，以便用户可以撤消脱离控制的进程
高于正常	进程中的线程在正常优先级与高优先级之间运行（这是Windows 2000中的新优先级类）
正常	进程中的线程没有特殊的调度需求
低于正常	进程中的线程在正常优先级与空闲优先级之间运行（这是Windows 2000中的新优先级类）
空闲	进程中的线程在系统空闲时运行。该进程通常由屏幕保护程序或后台实用程序和搜集统计数据的软件使用

当系统什么也不做的时候，将空闲优先级类用于应用程序的运行是最恰当不过的。没有用交互方式使用的计算机有可能仍然很繁忙（比如作为文件服务器），不应该与屏幕保护程序争用CPU时间。定期更新系统的某些状态的统计信息跟踪应用程序不应该干扰关键任务的运行。

只有当绝对必要的时候，才可以使用高优先级类。你会惊奇地发现，Windows Explorer是在高优先级上运行的。大多数时间Explorer的线程是暂停的，等待用户按下操作键或者点击鼠标按钮时被唤醒。当Explorer的线程处于暂停状态时，系统不将它的线程分配给CPU。因为这将使低优先级线程得以运行。但是一旦用户按下一个操作键或组合键，如Ctrl+Esc，系统就会唤醒Explorer的线程（当用户按下Ctrl+Esc组合键时，也会出现Start菜单）。如果低优先级线程正在运行，系统会立即抢在这些线程的前面，让Explorer的线程优先运行。

Microsoft就是按这种方法设计Explorer的，因为用户希望无论系统中正在运行什么，外壳程序都具有极强的响应能力。实际上，即使低优先级线程在无限循环中暂停运行，也能显示Explorer的窗口。由于Explorer的线程拥有较高的优先级，因此执行无限循环的线程被抢占，Explorer让用户终止挂起进程的运行。Explorer的运行特性非常出色，大部分时间它的线程无事可做，不必占用CPU时间。如果情况不是如此，那么整个系统的运行速度就会慢得多，许多应用程序就不会作出响应。

应该尽可能避免使用实时优先级类。实际上Windows NT 3.1的早期测试版并没有向应用程序展示这个优先级类，尽管该操作系统支持这个类。实时优先级是很高的优先级，它可能干扰操作系统任务的运行，因为大多数操作系统线程均以较低的优先级来运行。因此实时线程可能阻止必要的磁盘I/O信息和网络信息的产生。此外，键盘和鼠标输入将无法及时得到处理，用户可能以为系统已经暂停运行。大体来说，必须有足够的理由才能使用实时优先级，比如需要以很短的等待时间来响应硬件事件，或者执行某些不能中断的短期任务。

注意 除非用户拥有“提高调度优先级”的权限，否则进程不能用实时优先级类来运

China-pub.com
下载

行。凡是被指定为管理员或特权用户的用户，均默认拥有该权限。

当然，大多数进程都属于正常优先级类。低于正常和高于正常的优先级类是Windows 2000中的新增优先级。Microsoft增加这些优先级类的原因是，有若干家公司抱怨现有的优先级类无法提供足够的灵活性。

一旦选定了优先级类之后，就不必考虑你的应用程序与其他应用程序之间的关系，只需要集中考虑你的应用程序中的各个线程。Windows支持7个相对的线程优先级：即空闲、最低、低于正常、正常、高于正常、最高和关键时间优先级。这些优先级是相对于进程的优先级类而言的。大多数线程都使用正常线程优先级。表7-5描述了这些相对的线程优先级。

表7-5 相对的线程优先级

相对的线程优先级	描 述
关键时间	对于实时优先级类来说，线程在优先级31上运行，对于其他优先级类来说，线程在优先级15上运行
最高	线程在高于正常优先级的上两级上运行
高于正常	线程在正常优先级的上一级上运行
正常	线程在进程的优先级类上正常运行
低于正常	线程在低于正常优先级的下一级上运行
最低	线程在低于正常优先级的下两级上运行
空闲	对于实时优先级类来说，线程在优先级16上运行对于其他优先级类来说，线程在优先级1上运行

概括起来说，进程是优先级类的一个组成部分，你为进程中的线程赋予相对线程优先级。这里没有讲到0到31的优先级的任何情况。应用程序开发人员从来不必具体设置优先级。相反，系统负责将进程的优先级类和线程的相对优先级映射到一个优先级上。正是这种映射方式，Microsoft不想拘泥不变。实际上这种映射方式是随着系统的版本的升级而变化的。

表7-6显示了这种映射方式是如何用于Windows 2000的，注意，Windows NT的早期版本和某些Windows 95和Windows 98版本采用了不同的映射方式。未来的Windows版本中的映射方式也会变化。

例如，正常进程中的正常线程被赋予的优先级是8。由于大多数进程属于正常优先级类，而大多数线程属于正常线程优先级，因此系统中的大多数线程的优先级是8。

如果高优先级进程中有一个正常线程，该线程的优先级将是13。如果将进程的优先级类改为8，那么线程的优先级就变为4。如果改变了进程的优先级类，线程的相对优先级不变，但是它的优先级的等级却发生了变化。

表7-6 进程优先级类和线程相对优先级的映射

相对线程优先级	空闲	低于正常	正常	高于正常	高	实时
关键时间	15	15	5	15	15	31
最高	6	8	10	12	15	26
高于正常	5	7	9	11	14	25
正常	4	6	8	10	13	24
低于正常	3	5	7	9	12	23
最低	2	4	6	8	11	22
空闲	1	1	1	1	1	16

China-pub.com 下载

注意，表7-6并没有显示优先级的等级为0的线程。这是因为0优先级保留供零页线程使用，系统不允许任何其他线程拥有0优先级。另外，下列优先级等级是无法使用的：17、18、19、20、21、27、28、29和30。如果编写一个以内核方式运行的设备驱动程序，可以获得这些优先级等级，而用户方式的应用程序则不能。另外还要注意，实时优先级类中的线程不能低于优先级等级16。同样，非实时优先级类中的线程的等级不能高于15。

注意 有些人常常搞不清进程优先级类的概念。他们认为这可能意味着进程是可以调度的。但是进程是根本不能调度的，只有线程才能被调度。进程优先级类是个抽象概念，Microsoft提出这个概念的目的，是为了帮助你将它与调度程序的内部运行情况区分开来。它没有其他目的。

注意 一般来说，大多数时候高优先级的线程不应该处于可调度状态。当线程要进行某种操作时，它能迅速获得CPU时间。这时线程应该尽可能少地执行CPU指令，并返回睡眠状态，等待再次变成可调度状态。相反，低优先级的线程可以保持可调度状态，执行大量的CPU指令来进行它的操作。如果按照这些原则来办，整个操作系统就能正确地对用户作出响应。

7.9 程序的优先级

进程是如何被赋予优先级类的呢？当调用CreateProcess时，可以在fdwCreate参数中传递需要的优先级类。表7-7显示了优先级类的标识符。

表7-7 优先级类的标识类

优先级类	标识符
实时	REALTIME_PRIORITY_CLASS
高	HIGH_PRIORITY_CLASS
高于正常	ABOVE_NORMAL_PRIORITY_CLASS
正常	NORMAL_PRIORITY_CLASS
低于正常	BELOW_NORMAL_PRIORITY_CLASS
空闲	IDLE_PRIORITY_CLASS

创建子进程的进程负责选择子进程运行的优先级类，这看起来有点奇怪。让我们以Explorer为例来说明这个问题。当使用Explorer来运行一个应用程序时，新进程按正常优先级运行。Explorer不知道进程在做什么，也不知道隔多长时间它的线程需要进行调度。但是，一旦子进程运行，它就能够通过调用SetPriorityClass来改变它自己的优先级类：

```
BOOL SetPriorityClass(
   HANDLE hProcess,
   DWORD fdwPriority);
```

该函数将hProcess标识的优先级类改为fdwPriority参数中设定的值。fdwPriority参数可以是表7-7显示的标识符之一。由于该函数带有一个进程句柄，因此，只要拥有该进程的句柄和足够的访问权，就能够改变系统中运行的任何进程的优先级类。

一般来说，进程将试图改变它自己的优先级类。下面是如何使一个进程将它自己的优先级类设置为空闲的例子：

```
BOOL SetPriorityClass(
   GetCurrentProcess(),
   IDLE_PRIORITY_CLASS);
```

China-pub.com
下载

下面是用来检索进程的优先级类的补充函数：

```
DWORD GetPriorityClass(HANDLE hProcess);
```

正如你所期望的那样，该函数将返回表7-7中列出的标识符之一。

当使用命令外壳启动一个程序时，该程序的起始优先级是正常优先级。但是，如果使用Start命令来启动该程序，可以使用一个开关来设定应用程序的起始优先级。例如，在命令外壳输入下面的命令可使系统启动Calculator，并在开始时按空闲优先级来运行它：

```
C:\>START /LOW CALC.EXE
```

Start命令还能识别/BELOWNORMAL、/NORMAL、/ABOVENORMAL、/HIGH和/REALTIME等开关，以便按它们各自的优先级启动执行一个应用程序。当然，一旦应用程序启动运行，它就可以调用SetPriorityClass函数，将它自己的优先级改为它选择的任何优先级。

Windows 98 Windows 98的Start命令并不支持这些开关中的任何一个。Windows 98命令外壳启动的进程总是使用正常优先级类来运行。

Windows 2000的Task Manager使得用户可以改变进程的优先级类。图7-2显示了Task Manager的Processes选项卡，它显示了当前运行的所有进程。Base Pri列显示了每个进程的优先级类。可以改变进程的优先级类，方法是选定一个进程，然后从上下文菜单的Set Priority（设置优先级）子菜单中选择一个选项。

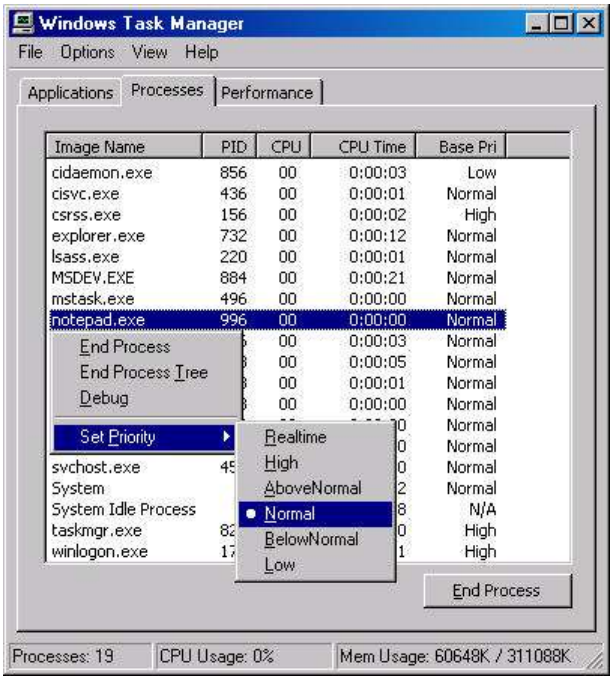


图7-2 Windows Task Manager对话框

当一个线程刚刚创建时，它的相对线程优先级总是设置为正常优先级。我总感到有些奇怪，CreateThread没有为调用者提供一个设置新线程的相对优先级的方法。若要设置和获得线程的相对优先级，必须调用下面的这些函数：

```
BOOL SetThreadPriority(
   HANDLE hThread,
   int nPriority);
```

China-pub.com 下载

当然，hThread参数用于标识想要改变优先级的单个线程， nPriority参数是表7-8列出的7个标识符之一。

表7-8 线程相对优先级的标识符常量

相对线程优先级	标识符常量
关键时间	THREAD_PRIORITY_TIME_CRITICAL
最高	THREAD_PRIORITY_HIGHEST
高于正常	THREAD_PRIORITY_ABOVE_NORMAL
正常	THREAD_PRIORITY_NORMAL
低于正常	THREAD_PRIORITY_BELOW_NORMAL
最低	THREAD_PRIORITY_LOWEST
空闲	THREAD_PRIORITY_IDLE

下面是检索线程的相对优先级的补充函数：

```
int GetThreadPriority(HANDLE hThread);
```

该函数返回表7-8列出的标识符之一。

若要创建一个带有相对优先级为空闲的线程，可以执行类似下面的代码：

```
DWORD dwThreadID;
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,
   CREATE_SUSPENDED, &dwThreadID);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```

注意，CreateThread函数创建的新函数带有的相对优先级总是正常优先级。若要使线程以空闲优先级来运行，应该将CREATE_SUSPENDED标志传递给CreateThread函数，这可以防止线程执行任何代码。然后可以调用 SetThreadPriority，将线程的优先级改为相对空闲优先级。这时可以调用ResumeThread，使得线程成为可调度的线程。你不知道线程何时能够获得 CPU时间，但是调度程序会考虑这样一个情况，即该线程拥有一个空闲优先级。最后，可以关闭新线程的句柄，一旦线程终止运行，内核对象就能被撤消。

注意 Windows没有提供返回线程的优先级的函数。这是故意进行的。记住，Microsoft保留了随时修改调度算法的权利。你不会设计需要调度算法专门知识的应用程序。如果坚持使用进程优先级类和相对线程优先级，你的应用程序不仅现在能够顺利地运行，而且在系统的将来版本上也能很好地运行。

7.9.1 动态提高线程的优先级等级

通过将线程的相对优先级与线程的进程优先级类综合起来考虑，系统就可以确定线程的优先级等级。有时这称为线程的基本优先级等级。系统常常要提高线程的优先级等级，以便对窗口消息或读取磁盘等I/O事件作出响应。

例如，在高优先级类进程中的一个正常优先级等级的线程的基本优先级等级是 13。如果用户按下一个操作键，系统就会将一个 WM_KEYDOWN消息放入线程的队列中。由于一个消息已经出现在线程的队列中，因此该线程就是可调度的线程。此外，键盘设备驱动程序也能够告诉系统暂时提高线程的优先级等级。该线程的优先级等级可能提高 2级，其当前优先级等级改为15。

China-pub.com
下载

系统在优先级为15时为一个时间片对该线程进行调度。一旦该时间片结束，系统便将线程的优先级递减1，使下一个时间片的线程优先级降为14。该线程的第三个时间片按优先级等级13来执行。如果线程要求执行更多的时间片，均按它的基本优先级等级13来执行。

注意，线程的当前优先级等级决不会低于线程的基本优先级等级。此外，导致线程成为可调度线程的设备驱动程序可以决定优先级等级提高的数量。Microsoft并没有规定各个设备驱动程序可以给线程的优先级提高多少个等级。这样就使得Microsoft可以不断地调整线程优先级提高的动态等级，以确定最佳的总体响应性能。

系统只能为基本优先级等级在1至15之间的线程提高其优先级等级。实际上这是因为这个范围称为动态优先级范围。此外，系统决不会将线程的优先级等级提高到实时范围（高于15）。由于实时范围中的线程能够执行大多数操作系统的函数，因此给等级的提高规定一个范围，就可以防止应用程序干扰操作系统的运行。另外，系统决不会动态提高实时范围内的线程优先级等级。

有些编程人员抱怨说，系统动态提高线程优先级等级的功能对他们的线程性能会产生一种不良的影响，为此Microsoft增加了下面两个函数，这样就能够使系统的动态提高线程优先级等级的功能不起作用：

```
BOOL SetProcessPriorityBoost(
   HANDLE hProcess,
   BOOL DisablePriorityBoost);
BOOL SetThreadPriorityBoost(
   HANDLE hThread,
   BOOL DisablePriorityBoost);
```

SetProcessPriorityBoost负责告诉系统激活或停用进行中的所有线程的优先级提高功能，而SetThreadPriorityBoost则让你激活或停用各个线程的优先级提高功能。这两个函数具有许多相似的共性，可以用来确定是激活还是停用优先级提高功能：

```
BOOL GetProcessPriorityBoost(
   HANDLE hProcess,
   PBOOL pDisablePriorityBoost);
BOOL GetThreadPriorityBoost(
   HANDLE hThread,
   PBOOL pDisablePriorityBoost);
```

对于这两个函数中的每个函数，可以传递想要查询的进程或线程的句柄，以及由函数设置的BOOL的地址。

Windows 98 Windows 98没有提供这4个函数的有用的实现代码。它们全部返回FALSE，后来对GetLastError的调用将返回ERROR_CALL_NOT_IMPLEMENTED。

另一种情况也会导致系统动态地提高线程的优先级等级。比如有一个优先级为4的线程准备运行但是却不能运行，因为一个优先级为8的线程正连续被调度。在这种情况下，优先级为4的线程就非常渴望得到CPU时间。当系统发现一个线程在大约3至4s内一直渴望得到CPU时间，它就将这个渴望得到CPU时间的线程的优先级动态提高到15，并让该线程运行两倍于它的时间量。当到了两倍时间量的时候，该线程的优先级立即返回到它的基本优先级。

7.9.2 为前台进程调整调度程序

当用户对进程的窗口进行操作时，该进程就称为前台进程，所有其他进程则称为后台进程。当然，用户希望他正在使用的进程比后台进程具有更强的响应性。为了提高前台进程的响应性，

China-pub.com 下载

Windows能够为前台进程中的线程调整其调度算法。对于Windows 2000来说，系统可以为前台进程的线程提供比通常多的CPU时间量。这种调整只能在前台进程属于正常优先级类的进程时才能进行。如果它属于其他任何优先级类，就无法进行任何调整。

Windows 2000实际上允许用户对这种调整进行相应的配置。在System Properties（系统属性）对话框的Advanced选项卡上，用户可以单击Performance Options(性能选项)按钮，打开图7-3所示的对话框。

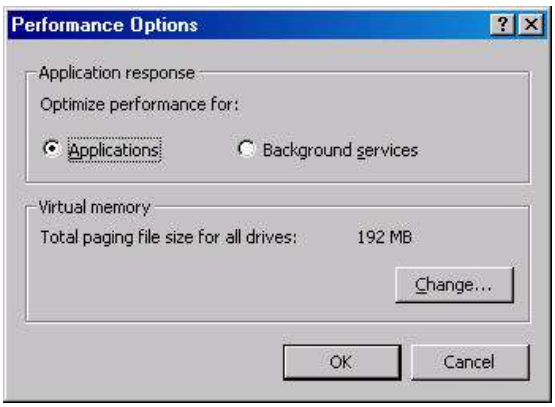


图7-3 Performance Options 对话框

如果用户选择优化应用程序的性能，系统就执行配置的调整。如果用户选择优化后台服务程序的性能，系统就不进行调整。当安装Windows 2000的专业版时，Applications就会被默认选定。对于Windows 2000的所有其他版本，则默认选定Background Services，因为计算机将主要由非交互式用户使用。

当进程移到前台时，Windows 98也会对正常优先级类的进程中的线程调度算法进行调整。当一个优先级为正常的进程移到前台时，系统便将最低、低于正常、正常、高于正常和最高等优先级的线程的优先级提高1，优先级为空闲和关键时间的线程的优先级则不予提高。因此，在正常优先级类的进程中运行的、其相对优先级为正常的线程，它的优先级等级是9而不是8。当进程返回后台时，进程中的线程便自动返回它们定义好的基本优先级等级。

Windows 98 Windows 98没有提供允许用户配置这种调整手段的任何用户界面，因为Windows 98不是作为专用服务器来运行的。

将进程改为前台进程的原因是，使它们能够对用户的输入更快地作出响应。如果不改为前台进程，那么在后台的正常打印进程与在后台接收用户输入的正常进程就会平等地争用CPU时间。用户会发现文本无法在前台应用程序中顺利地显示。但是，由于系统改变了前台进程的线程优先级，前台进程的线程就能对用户的输入更好地作出响应。

7.9.3 Scheduling Lab示例应用程序

使用Scheduling Lab应用程序“07SchedLab.exe”(见后面的清单7-1)，可以对进程优先级类和相对线程优先级进行操作试验，以了解它们对系统的总体性能产生的影响。该应用程序的源代码和源文件位于本书所附光盘上的07-SchedLab目录中。当启动该程序时，就会出现图7-4所示的窗口。

开始时，主线程总是处于繁忙状态，因此CPU的使用量立即跳到100%。该主线程连续递增一个数字，并将它添加给右边的列表框。这个数字并没有任何意义，它只是显示线程正在忙

China-pub.com
下载

于进行什么操作。若要了解线程调度对系统会产生什么实际影响，建议至少要同时运行该示例应用程序的两个实例，看一看改变一个实例的优先级会对另一个实例带来的影响。也可以运行Task Manager，以便监控所有实例的CPU使用量。

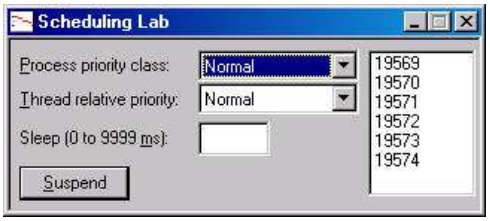


图7-4 进程优先级类和相对线程优先级的试验窗口

当进行这些测试时，CPU的使用量开始时将上升为100%，该应用程序的所有实例将获得大约相等的CPU时间（Task Manager应该显示应用程序的所有实例大致相同的CPU使用量百分比）。

如果将一个实例的优先级类改为高于正常或高优先级类，那么应该看到它得到了大部分的CPU使用量。而其他实例中的数字滚动则没有规律。但是其他实例的数字不会完全停止滚动，因为系统将为渴求CPU时间的线程自动执行优先级的动态提高。不管怎样，可以随意调整优先级类和相对线程优先级，以了解它们对其他实例的影响。我有目的地对Scheduling Lab应用程序进行了编码，这样就无法将进程改为实时优先级类，这可以防止操作系统线程的不正常的运行。如果想要试用实时优先级，必须自己修改源代码。

可以使用Sleep域，使主线程在0到9999之间的任意毫秒内无法调度。请试用这项功能，并观察传递仅为1ms的睡眠值时可以重新获得多少CPU时间。在我的300MHz Pentium II笔记本电脑上，我赢得了99%的CPU时间。

单击Suspend（暂停）按钮，可使主线程产生一个子线程。这个子线程能够暂停主线程的运行，并显示图7-5所示的消息框。



图7-5 消息框

当这个消息框显示时，主线程将完全暂停运行，并且不使用任何CPU时间。子线程也不使用任何CPU时间，因为它是在等待用户执行某种操作。当消息框显示时，可以将它移到应用程序的主窗口，然后将它移开，这样就能够看到主窗口。由于主线程已经暂停运行，因此主窗口将无法接收任何窗口消息（包括WM_PAINT）。这证明该线程已经暂停运行。当关闭该消息框时，主线程就恢复运行，CPU使用量回到100%。

若要再进行一次试验，请打开前一节介绍的Performance Options对话框，将Application改为Background Services，或者将Background Services改为Application。然后打开SchedLab程序的多个实例，将它们全部设置为正常优先级类，并激活其中的一个，使之成为一个前台进程。这时就能够看到性能的设置对前台/后台进程产生的影响。

清单7-1 SchedLab示例应用程序

SchedLab.cpp

```
/******************************************************************************
Module:  SchedLab.cpp
```

China-pub.com 下载

```
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.H"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>       // For _beginthreadex
#include "Resource.H"

//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI ThreadFunc(PVOID pvParam) {
   HANDLE hThreadPrimary = (HANDLE) pvParam;
   SuspendThread(hThreadPrimary);
   chMB(
      "The Primary thread is suspended.\n"
      "It no longer responds to input and produces no output.\n"
      "Press OK to resume the primary thread & exit this secondary thread.\n");
   ResumeThread(hThreadPrimary);
   CloseHandle(hThreadPrimary);

   // To avoid deadlock, call EnableWindow after ResumeThread.
   EnableWindow(
      GetDlgItem(FindWindow(NULL, TEXT("Scheduling Lab")), IDC_SUSPEND),
      TRUE);
   return(0);
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SCHEDLAB);
   // Initialize process priority classes
   HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSPRIORITYCLASS);

   int n = ComboBox_AddString(hwndCtl, TEXT("High"));
   ComboBox_SetItemData(hwndCtl, n, HIGH_PRIORITY_CLASS);

   // Save our current priority class
   DWORD dwpc = GetPriorityClass(GetCurrentProcess());

   if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

      // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

      // Restore our original priority class
      SetPriorityClass(GetCurrentProcess(), dwpc);

      // Add the Above Normal priority class
```

China-pub.com
下载


```
            n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
            ComboBox_SetItemData(hwndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);

            dwpc = 0;  // Remember that this system supports below normal
         }

         int nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
         ComboBox_SetItemData(hwndCtl, n, NORMAL_PRIORITY_CLASS);

         if (dwpc == 0) {

            // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

            // Add the Below Normal priority class
            n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
            ComboBox_SetItemData(hwndCtl, n, BELOW_NORMAL_PRIORITY_CLASS);
         }

         n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
         ComboBox_SetItemData(hwndCtl, n, IDLE_PRIORITY_CLASS);

         ComboBox_SetCurSel(hwndCtl, nNormal);

         // Initialize thread relative priorities
         hwndCtl = GetDlgItem(hwnd, IDC_THREADRELATIVEPRIORITY);

         n = ComboBox_AddString(hwndCtl, TEXT("Time critical"));
         ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_TIME_CRITICAL);
   n = ComboBox_AddString(hwndCtl, TEXT("Highest"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_HIGHEST);

   n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_ABOVE_NORMAL);

   nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_NORMAL);

   n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_BELOW_NORMAL);

   n = ComboBox_AddString(hwndCtl, TEXT("Lowest"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_LOWEST);

   n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
   ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_IDLE);

   ComboBox_SetCurSel(hwndCtl, nNormal);

   Edit_LimitText(GetDlgItem(hwnd, IDC_SLEEPTIME), 4);    // Maximum of 9999

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
void Dlg_OnCommand (HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         PostQuitMessage(0);
         break;

      case IDC_PROCESSPRIORITYCLASS:
         if (codeNotify == CBN_SELCHANGE) {
            SetPriorityClass(GetCurrentProcess(), (DWORD)
               ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
         }
         break;
      case IDC_THREADRELATIVEPRIORITY:
         if (codeNotify == CBN_SELCHANGE) {
            SetThreadPriority(GetCurrentThread(), (DWORD)
               ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
         }
         break;

      case IDC_SUSPEND:
         // To avoid deadlock, call EnableWindow before creating
         // the thread which calls SuspendThread.
         EnableWindow(hwndCtl, FALSE);

         HANDLE hThreadPrimary;
         DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
            GetCurrentProcess(), &hThreadPrimary,
            THREAD_SUSPEND_RESUME, FALSE, DUPLICATE_SAME_ACCESS);
         DWORD dwThreadID;
         CloseHandle(chBEGINTHREADEX(NULL, 0, ThreadFunc,
            hThreadPrimary, 0, &dwThreadID));
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain (HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {
```

China-pub.com
下载


```
   HWND hwnd =
      CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_SCHEDLAB), NULL, Dlg_Proc);
   BOOL fQuit = FALSE;

   while (!fQuit) {
      MSG msg;
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

         // IsDialogMessage allows keyboard navigation to work properly.
         if (!IsDialogMessage(hwnd, &msg)) {

            if (msg.message == WM_QUIT) {
               fQuit = TRUE;  // For WM_QUIT, terminate the loop.
            } else {
               // Not a WM_QUIT message. Translate it and dispatch it.
               TranslateMessage(&msg);
               DispatchMessage(&msg);
            }
         }  // if (!IsDialogMessage())
      } else {

         // Add a number to the listbox
         static int s_n = -1;
         TCHAR sz[20];
         wsprintf(sz, TEXT("%u"), ++s_n);
         HWND hwndWork = GetDlgItem(hwnd, IDC_WORK);
         ListBox_SetCurSel(hwndWork, ListBox_AddString(hwndWork, sz));

         // Remove some strings if there are too many entries
         while (ListBox_GetCount(hwndWork) > 100)
            ListBox_DeleteString(hwndWork, 0);

         // How long should the thread sleep
         int nSleep = GetDlgItemInt(hwnd, IDC_SLEEPTIME, NULL, FALSE);
         if (chINRANGE(1, nSleep, 9999))
            Sleep(nSleep);
      }
   }
   DestroyWindow(hwnd);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

SchedLab.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
```

China-pub.com 下载

```
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SCHEDLAB DIALOGEX 0, 0, 209, 70
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
EXSTYLE WS_EX_NOPARENTNOTIFY | WS_EX_CLIENTEDGE
CAPTION "Scheduling Lab"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "&Process priority class:",IDC_STATIC,4,6,68,8
    COMBOBOX        IDC_PROCESSPRIORITYCLASS,84,4,72,80,CBS_DROPDOWNLIST |
                    WS_TABSTOP
    LTEXT           "&Thread relative priority:",IDC_STATIC,4,20,72,8
    COMBOBOX        IDC_THREADRELATIVEPRIORITY,84,18,72,76,CBS_DROPDOWNLIST |
                    WS_TABSTOP
    LTEXT           "Sleep (0 to 9999 &ms):",IDC_STATIC,4,36,68,8
    EDITTEXT        IDC_SLEEPTIME,84,34,32,14,ES_NUMBER
    PUSHBUTTON      "&Suspend",IDC_SUSPEND,4,52,49,14
    LISTBOX         IDC_WORK,160,4,48,60,NOT LBS_NOTIFY |
                    LBS_NOINTEGRALHEIGHT | LBS_NOSEL | WS_TABSTOP
END


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_SCHEDLAB, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 202
        TOPMARGIN, 7
        BOTTOMMARGIN, 63
    END
```

China-pub.com
下载


```
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SCHEDLAB            ICON    DISCARDABLE     "SchedLab.ico"
#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

7.10 亲缘性

按照默认设置，当系统将线程分配给处理器时，Windows 2000使用软亲缘性来进行操作。这意味着如果所有其他因素相同的话，它将设法在它上次运行的那个处理器上运行线程。让线程留在单个处理器上，有助于重复使用仍然在处理器的内存高速缓存中的数据。

China-pub.com 下载

有一种新的计算机结构，称为NUMA（非统一内存访问），在该结构中，计算机包含若干块插件板，每个插件板上有4个CPU和它自己的内存区。图7-6显示了一台配有3块插件板的计算机，总共有12个CPU，这样，任何一个线程都可以在12个CPU中的任何一个上运行。

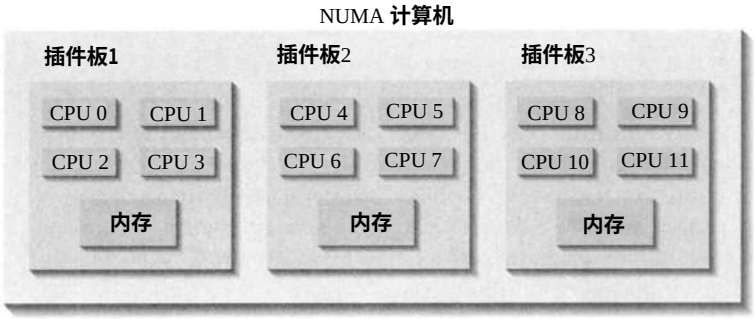


图7-6 NUMA 计算机结构示意图

当CPU访问的内存是它自己的插件板上的内存时，NUMA系统运行的性能最好。如果CPU需要访问位于另一个插件板上的内存时，性能就会大大降低。在这样的环境中，就需要来自一个进程中的线程在CPU 0至3上运行，让另一个进程中的线程在CPU 4至7上运行，依次类推。为了适应这种计算机结构的需要，Windows 2000允许设置进程和线程的亲缘性。换句话说，可以控制哪个CPU能够运行某些线程。这称为硬亲缘性。

计算机在引导时，系统要确定机器中有多少个CPU可供使用。通过调用GetSystemInfo函数（第14章介绍），应用程序就能查询机器中的CPU数量。按照默认设置，任何线程都可以调度到这些CPU中的任何一个上去运行。为了限制在可用CPU的子集上运行的单个进程中的线程数量，可以调用SetProcessAffinityMask：

```
BOOL SetProcessAffinityMask(
   HANDLE hProcess,
   DWORD_PTR dwProcessAffinityMask);
```

第一个参数hProcess用于指明要影响的是哪个进程。第二个参数dwProcessAffinityMask是个位屏蔽，用于指明线程可以在哪些CPU上运行。例如，传递0x00000005表示该进程中的线程可以在CPU 0和CPU 2上运行，但是不能在CPU 1和CPU3至31上运行。

注意，子进程可以继承进程的亲缘性。因此，如果一个进程的亲缘性屏蔽是0x00000005，那么它的子进程中的任何线程都拥有相同的位屏蔽，并共享相同的CPU。此外，可以使用作业内核对象将一组进程限制在要求的一组CPU上运行。

当然，还有一个函数也能够返回进程的亲缘性位屏蔽，它就是GetProcessAffinityMask，如下面的代码所示：

```
BOOL GetProcessAffinityMask(
   HANDLE hProcess,
   PDWORD_PTR pdwProcessAffinityMask,
   PDWORD_PTR pdwSystemAffinityMask);
```

这里也可以传递想要亲缘性屏蔽的进程句柄，该函数填入pdwProcessAffinityMask指向的变量。该函数还能返回系统的亲缘性屏蔽（在pdwSystemAffinityMask指向的变量中）。系统的亲缘性屏蔽用于指明系统的哪个CPU能够处理线程。进程的亲缘性屏蔽始终是一个系统的亲缘性屏蔽的正确子集。

China-pub.com
下载

Windows 98 无论计算机中实际拥有多少个CPU，Windows 98只使用一个CPU。因此，GetProcessAffinityMask总是用1填入两个变量中。

到现在为止，已经介绍了如何将进程的多个线程限制到一组CPU上去运行。有时可能想要将进程中的一个线程限制到一组CPU上去运行。例如，可能有一个包含4个线程的进程，它们在拥有4个CPU的计算机上运行。如果这些线程中的一个线程正在执行非常重要的操作，而你想增加某个CPU始终可供它使用的可能性，为此你对其他3个线程进行了限制，使它们不能在CPU 0上运行，而只能在CPU 1、2和3上运行。

通过调用SetThreadAffinityMask，就能为各个线程设置亲缘性屏蔽：

```
DWORD_PTR SetThreadAffinityMask(
   HANDLE hThread,
   DWORD_PTR dwThreadAffinityMask);
```

该函数中的hThread参数用于指明要限制哪个线程，dwThreadAffinityMask用于指明该线程能够在哪个CPU上运行。dwThreadAffinityMask必须是进程的亲缘性屏蔽的相应子集。返回值是线程的前一个亲缘性屏蔽。因此，若要将3个线程限制到CPU 1、2和3上去运行，可以这样操作：

```
// Thread 0 can only run on CPU 0.
SetThreadAffinityMask(hThread0, 0x00000001);

// Threads 1, 2, 3 run on CPUs 1, 2, 3.
SetThreadAffinityMask(hThread1, 0x0000000E);
SetThreadAffinityMask(hThread2, 0x0000000E);
SetThreadAffinityMask(hThread3, 0x0000000E);
```

Windows 98 由于计算机中无论配有多少个CPU，Windows 98只使用一个CPU，因此dwThreadAffinityMask参数必须始终是1。

当一个x86系统引导时，系统要执行相应的代码，以便测定主机上的哪些CPU遇到了著名的Pentium浮点错误。系统必须为每个CPU测试其浮点错误，方法是将线程的亲缘性设置为第一个CPU，执行潜在的故障分割操作，并将结果与已知的正确答案进行比较。然后对下一个CPU进行上述同样的操作，如此等等。

注意 在大多数环境中，改变线程的亲缘性就会影响调度程序有效地在各个CPU之间移植线程的能力，而这种能力可以最有效地使用CPU时间。表7-9显示了一个例子。

表7-9 线程的亲缘性示例

线 程	优 先 级	亲缘性屏蔽	结 果
A	4	0x00000001	CPU 0
B	8	0x00000003	CPU 1
C	6	0x00000002	不能运行

当线程A被唤醒时，调度程序发现该线程可以在CPU 0上运行，因此它被分配给CPU 0。然后线程B被唤醒，调度程序发现该线程可以被分配给CPU 0或1，但是，由于CPU 0正在使用之中，因此调度程序将线程B分配给了CPU 1。至此，一切进行得都很顺利。

这时线程C被唤醒，调度程序发现它只能在CPU 1上运行。但是CPU 1正在被线程

China-pub.com 下载

B使用着，它是个优先级为8的线程。由于线程C的优先级为6，因此它不能抢在线程B的前面运行。线程C可以抢在线程A的前面运行，因为线程A的优先级是4，但是调度程序不会使它抢在线程A的前面运行，因为线程C不能在CPU 0上运行。

这显示出为线程设置硬亲缘性将会对调度程序的优先级设置方案产生什么样的影响。

有时强制将一个线程分配给特定的 CPU的做法是不妥当的。例如，有 3个线程都只能在CPU 0上运行，而CPU 1、2和3则闲着无事可做。在这种情况下，如果告诉系统想让一个线程在某个CPU上运行，但是允许该线程在可能的情况下移到另一个 CPU上去运行，那么这种办法会更好些。

若要为线程设置一个理想的CPU，可以调用SetThreadIdealProcessor:

```
DWORD SetThreadIdealProcessor(
   HANDLE hThread,
   DWORD dwIdealProcessor);
```

hThread用于指明要为哪个线程设置首选的 CPU。与我们已经介绍的其他函数不同，dwIdealProcessor函数不是个位屏蔽函数，它是个从0到31的整数，用于指明供线程使用的首选CPU。可以传递一个MAXIMUM_PROCESSORS的值（在WinNT.h中定义为32），用于指明不存在理想的CPU。如果没有为该线程设置理想的 CPU，那么该函数返回前一个理想的 CPU或MAXIMUM_PROCESSORS。

也可以在一个可执行文件的头上设置处理器亲缘性。奇怪的是，似乎不存在它的链接程序开关，不过可以使用类似下面的代码：

```
// Load the EXE into memory.
PLOADED_IMAGE pLoadedImage = ImageLoad(szExeName, NULL);

// Get the current load configuration information for the EXE.
IMAGE_LOAD_CONFIG_DIRECTORY ilcd;
GetImageConfigInformation(pLoadedImage, &ilcd);

// Change the processor affinity mask.
ilcd.ProcessAffinityMask = 0x00000003; // I desire CPUs 0 and 1

// Save the new load configuration information.
SetImageConfigInformation(pLoadedImage, &ilcd);

// Unload the EXE from memory.
ImageUnload(pLoadedImage);
```

这里不想详细说明所有这些函数的情况，如果有兴趣的话，可以在 Platform SDK文档中查看这些函数的具体情况。另外，可以使用称为 ImageCfg.exe的实用程序，以便改变可执行程序模块的头上的某些标志。当运行ImageCfg.exe时，它会显示下面的使用情况：

```
usage: IMAGECFG [switches] image-names...
               [-?] display this message
               [-a Process Affinity mask value in hex]
               [-b BuildNumber]
               [-c Win32 GetVersionEx Service Pack return value in hex]
               [-d decommit thresholds]
[-g bitsToClear bitsToSet]
[-h 1|0 (Enable/Disable Terminal Server Compatible bit)
```

China-pub.com
下载


```
[-k StackReserve[,StackCommit]
[-l enable large (>2GB) addresses
[-m maximum allocation size]
[-n bind no longer allowed on this image
[-o default critical section timeout
[-p process heap flags]
[-q only print config info if changed
[-r run with restricted working set]
[-s path to symbol files]
[-t VirtualAlloc threshold]
[-u Marks image as uniprocessor only]
[-v MajorVersion.MinorVersion]
[-w Win32 GetVersion return value in hex]
[-x Mark image as Net - Run From Swapfile
[-y Mark image as Removable - Run From Swapfile
```

若要修改应用程序的可允许的亲缘性屏蔽，可以执行ImageCfg.exe来设定-a开关。当然，该实用程序所做的工作只是调用上面这个代码段中显示的各个函数。还要注意的是-u，它负责告诉系统，可执行程序只能在单个CPU系统上运行。

最后，Windows 2000的Task Manager允许用户改变进程的CPU亲缘性，方法是选定一个进程，显示它的上下文菜单。如果在多处理器计算机上运行，会看到一个Set Affinity菜单项（该菜单项在单处理器计算机中没有）。当选择该菜单项时，会看到图7-7所示的对话框，在这个对话框中，可以选定进程中的线程能够在上面运行的CPU。

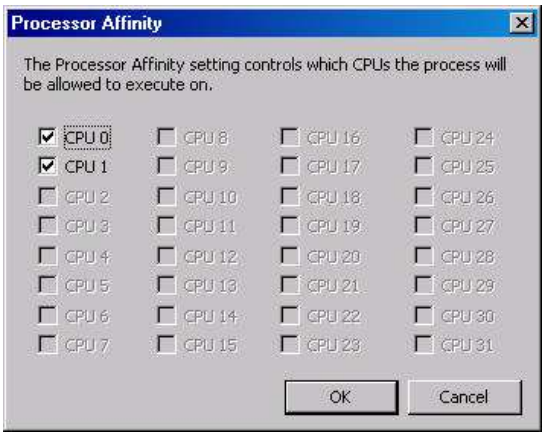


图7-7 CPU亲缘性对话框

Windows 2000 当Windows 2000在x86计算机上引导时，可以限制系统能够使用的CPU的数量。在引导过程中，系统要查看称为Boot.ini的文件，该文件位于引导驱动器的根目录中。下面是我的双处理器计算机上的Boot.ini文件：

```
[boot loader]
timeout=2
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
   /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
   /fastdetect /NumProcs=1
```

这个Boot.ini文件是Windows 2000安装时产生的，不过我使用Notepad加上了最后一行代码。这行代码告诉系统，在系统引导时，我只应该使用机器中的一个处理器。 /NumProcs=1这个开关是用来实现这一点的关键。我常常发现它对调试非常有用。

注意，由于只考虑到打印方面的需要，因此上面的程序清单中的各个选项都是在单独的一行上列出的。Boot.ini文件要求各个选项和到达根分区的ARC路径必须出现在一行上。

China-pub.com
下载

第8章　用户方式中线程的同步

当所有的线程在互相之间不需要进行通信的情况下就能够顺利地运行时，Microsoft Windows的运行性能最好。但是，线程很少能够在所有的时间都独立地进行操作。通常情况下，要生成一些线程来处理某个任务。当这个任务完成时，另一个线程必须了解这个情况。

系统中的所有线程都必须拥有对各种系统资源的访问权，这些资源包括内存堆栈，串口，文件，窗口和许多其他资源。如果一个线程需要独占对资源的访问权，那么其他线程就无法完成它们的工作。反过来说，也不能让任何一个线程在任何时间都能访问所有的资源。如果在一个线程从内存块中读取数据时，另一个线程却想要将数据写入同一个内存块，那么这就像你在读一本书时另一个人却在修改书中的内容一样。这样，书中的内容就会被搞得乱七八糟，结果什么也看不清楚。

线程需要在下面两种情况下互相进行通信：

- 当有多个线程访问共享资源而不使资源被破坏时。
- 当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

线程的同步包括许多方面的内容，下面几章将分别对它们进行介绍。值得高兴的是，Windows提供了许多方法，可以非常容易地实现线程的同步。但是，要想随时了解一连串的线程想要做什么，那是非常困难的。我们的头脑的工作不是异步的，我们希望以一种有序的方式来思考许多事情，每次前进一步。不过多线程环境不是这样运行的。

我是在大约1992年的时候开始从事多线程的编程工作的。最初，我犯过许多编程错误，在我出版的书籍和杂志文章中实际上都存在着与线程同步相关的错误。现在我的编程工作熟练了许多，但是并未做到完美无缺。希望本书中的全部内容不存在任何错误（尽管现在我知道我可以做得更好些）。要搞好线程的同步，唯一的办法是通过实践。下面几章将要介绍系统是如何运行的，并展示如何实现线程的正确同步，不过应该面对这样一个问题：即取得经验的同时，难免要犯错误。

8.1　原子访问：互锁的函数家族

线程同步问题在很大程度上与原子访问有关，所谓原子访问，是指线程在访问资源时能够确保所有其他线程都不在同一时间内访问相同的资源。让我们来看一看下面这个简单例子：

```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
   g_x++;
   return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
   g_x++;
   return(0);
}
```

China-pub.com
下载

在这个代码中，声明了一个全局变量g_x，并将它初始化为0。现在，假设创建两个线程，一个线程执行ThreadFunc1，另一个线程执行ThreadFunc2。这两个函数中的代码是相同的，它们都将1添加给全局变量g_x。因此，当两个线程都停止运行时，你可能希望在g_x中看到2这个值。但是你真的看到了吗？回答是，也许看到了。根据代码的编写方法，你无法说明g_x中最终包含了什么东西。下面我们来说明为什么会出现这种情况。假设编译器生成了下面这行代码，以便将g_x递增1：

```
MOV EAX, [g_x]        ; Move the value in g_x into a register.
INC EAX               ; Increment the value in the register.
MOV [g_x], EAX        ; Store the new value back in g_x.
```

两个线程不可能在完全相同的时间内执行这个代码。因此，如果一个线程在另一个线程的后面执行这个代码，那么下面就是实际的执行情况：

```
MOV EAX, [g_x]        ; Thread 1: Move 0 into a register.
INC EAX               ; Thread 1: Increment the register to 1.
MOV [g_x], EAX        ; Thread 1: Store 1 back in g_x.

MOV EAX, [g_x]        ; Thread 2: Move 1 into a register.
INC EAX               ; Thread 2: Increment the register to 2.
MOV [g_x], EAX        ; Thread 2: Store 2 back in g_x.
```

当两个线程都将g_x的值递增之后，g_x中的值就变成了2。这很好，并且正是我们希望的：即取出零（0），两次将它递增1，得出的值为2。太好了。不过不要急，Windows是个抢占式多线程环境。一个线程可以随时中断运行，而另一个线程则可以随时继续执行。这样，上面的代码就无法完全按编写的那样来运行。它可能按下面的形式运行：

```
MOV EAX, [g_x]        ; Thread 1: Move 0 into a register.
INC EAX               ; Thread 1: Increment the register to 1.

MOV EAX, [g_x]        ; Thread 2: Move 0 into a register.
INC EAX               ; Thread 2: Increment the register to 1.
MOV [g_x], EAX        ; Thread 2: Store 1 back in g_x.

MOV [g_x], EAX        ; Thread 1: Store 1 back in g_x.
```

如果代码按这种形式来运行，g_x中的最后值就不是2，而是你预期的1。这使人感到非常担心，因为你对调度程序的控制能力非常小。实际上，如果有100个线程在执行相同的线程函数，当它们全部退出之后，g_x中的值可能仍然是1。显然，软件开发人员无法在这种环境中工作。我们希望在所有情况下两次递增0产生的结果都是2。另外，不要忘记，编译器生成代码的方法，哪个CPU在执行这个代码，以及主计算机中安装了多少个CPU等因素，决定了产生的结果可能是不同的。这就是该环境的运行情况，我们对此无能为力。但是，Windows确实提供了一些函数，如果正确地使用这些函数，就能确保产生应用程序的代码得到的结果。

为了解决上面的问题，需要某种比较简单的方法。我们需要一种手段来保证值的递增能够以原子操作方式来进行，也就是不中断地进行。互锁的函数家族提供了我们需要的解决方案。互锁的函数尽管用处很大，而且很容易理解，却有些让人望而生畏，大多数软件开发人员用得很少。所有的函数都能以原子操作方式对一个值进行操作。让我们看一看下面这个InterlockedExchangeAdd函数：

```
LONG InterlockedExchangeAdd(
   PLONG plAddend,
```

China-pub.com 下载

```
LONG lIncrement);
```

这是个最简单的函数了。只需调用这个函数，传递一个长变量地址，并指明将这个值递增多少即可。但是这个函数能够保证值的递增以原子操作方式来完成。因此可以将上面的代码重新编写为下面的形式：

```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
   InterlockedExchangeAdd(&g_x, 1);
   return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
   InterlockedExchangeAdd(&g_x, 1);
   return(0);
}
```

通过这个小小的修改，g_x就能以原子操作方式来递增，因此可以确保g_x中的值最后是2。这样是不是感到好一些？注意，所有线程都应该设法通过调用这些函数来修改共享的长变量，任何线程都不应该通过调用简单的C语句来修改共享的变量：

```
// The long variable shared by many threads
LONG g_x;
.
.
.
// Incorrect way to increment the long
g_x++;
.
.
.
// Correct way to increment the long
InterlockedExchangeAdd(&g_x, 1);
```

互锁函数是如何运行的呢？答案取决于运行的是何种CPU平台。对于x86家族的CPU来说，互锁函数会对总线发出一个硬件信号，防止另一个CPU访问同一个内存地址。在Alpha平台上，互锁函数能够执行下列操作：

1) 打开CPU中的一个特殊的位标志，并注明被访问的内存地址。

2) 将内存的值读入一个寄存器。

3) 修改该寄存器。

4) 如果CPU中的特殊位标志是关闭的，则转入第二步。否则，特殊位标志仍然是打开的，寄存器的值重新存入内存。

你也许会问，执行第4步时CPU中的特殊位标志是如何关闭的呢？答案是：如果系统中的另一个CPU试图修改同一个内存地址，那么它就能够关闭CPU的特殊位标志，从而导致互锁函数返回第二步。

不必清楚地了解互锁函数是如何工作的。重要的是要知道，无论编译器怎样生成代码，无论计算机中安装了多少个CPU，它们都能保证以原子操作方式来修改一个值。还必须保证传递给这些函数的变量地址正确地对齐，否则这些函数就会运行失败（第13章将介绍数据对齐问题）。

对于互锁函数，需要了解的另一个重要问题是，它们运行的速度极快。调用一个互锁函数

China-pub.com
下载

通常会导致执行几个CPU周期（通常小于50），并且不会从用户方式转换为内核方式（通常这需要执行1000个CPU周期）。

当然，可以使用InterlockedExchangeAdd减去一个值——只要为第二个参数传递一个负值。InterlockedExchangeAdd将返回在*plAddend中的原始值。

下面是另外两个互锁函数：

```
LONG InterlockedExchange(
   PLONG plTarget,
   LONG lValue);

PVOID InterlockedExchangePointer(
   PVOID* ppvTarget,
   PVOID pvValue);
```

InterlockedExchange和InterlockedExchangePointer能够以原子操作方式用第二个参数中传递的值来取代第一个参数中传递的当前值。如果是32位应用程序，两个函数都能用另一个32位值取代一个32位值。但是，如果是个64位应用程序，那么InterlockedExchange能够取代一个32位值，而InterlockedExchangePointer则取代64位值。两个函数都返回原始值。当实现一个循环锁时，InterlockedExchange是非常有用的：

```
// Global variable indicating whether a shared resource is in use or not
BOOL g_fResourceInUse = FALSE;
.
.
.
void Func1() {
   // Wait to access the resource.
   while (InterlockedExchange (&g_fResourceInUse, TRUE) == TRUE)
      Sleep(0);

   // Access the resource.
   .
   .
   .

   // We no longer need to access the resource.
   InterlockedExchange(&g_fResourceInUse, FALSE);
}
```

while循环是循环运行的，它将g_fResourceInUse中的值改为TRUE，并查看它的前一个值，以了解它是否是TRUE。如果这个值原先是FALSE，那么该资源并没有在使用，而是调用线程将它设置为在用状态并退出该循环。如果前一个值是TRUE，那么资源正在被另一个线程使用，while循环将继续循环运行。

如果另一个线程要执行类似的代码，它将在while循环中运行，直到g_fResourceInUse重新改为FALSE。调用函数结尾处的InterlockedExchange，可显示应该如何将g_fResourceInUse重新设置为FALSE。

当使用这个方法时必须格外小心，因为循环锁会浪费CPU时间。CPU必须不断地比较两个值，直到一个值由于另一个线程而“奇妙地”改变为止。另外，该代码假定使用循环锁的所有线程都以相同的优先级等级运行。也可以把执行循环锁的线程的优先级提高功能禁用（通过调用SetProcessPriorityBoost或setThreadPriorityBoost函数来实现之）。

此外，应该保证将循环锁变量和循环锁保护的数据维护在不同的高速缓存行中（本章后面部分介绍）。如果循环锁变量与数据共享相同的高速缓存行，那么使用该资源的CPU将与试图访问该资源的任何CPU争用高速缓存行。

China-pub.com 下载

应该避免在单个CPU计算机上使用循环锁。如果一个线程正在循环运行，它就会浪费前一个CPU时间，这将防止另一个线程修改该值。我在上面的 while循环中使用了Sleep,从而在某种程度上解决了浪费CPU时间的问题。如果使用 Sleep，你可能想睡眠一个随机时间量；每次请求访问该资源均被拒绝时，你可能想进一步延长睡眠时间。这可以防止线程浪费 CPU时间。根据情况，最好是全部删除对Sleep的调用。或者使用对SwitchToThread（Windows 98中没有这个函数）的调用来取代它。勇于试验和不断纠正错误，是学习的最好方法。

循环锁假定，受保护的资源总是被访问较短的时间。这使它能够更加有效地循环运行，然后转为内核方式并进入等待状态。许多编程人员循环运行一定的次数（比如 400次），如果对资源的访问仍然被拒绝，那么该线程就转为内核方式，在这种方式下，它要等待（不消耗 CPU时间），直到该资源变为可供使用为止。这就是关键部分实现的方法。

循环锁在多处理器计算机上非常有用，因为当一个线程循环运行的时候，另一个线程可以在另一个CPU上运行。但是，即使在这种情况下，也必须小心。不应该让线程循环运行太长的时间，也不能浪费更多的CPU时间。本章后面将进一步介绍循环锁。第10章将介绍如何使用循环锁。

下面是最后两个互锁函数：

```
PVOID InterlockedCompareExchange(
   PLONG plDestination,
   LONG lExchange,
   LONG lComparand);

PVOID InterlockedCompareExchangePointer(
   PVOID* ppvDestination,
   PVOID pvExchange,
   PVOID pvComparand);
```

这两个函数负责执行一个原子测试和设置操作。如果是 32位应用程序，那么两个函数都在32位值上运行，但是，如果是64位应用程序，InterlockedCompareExchange函数在32位值上运行，而InterlockedCompareExchangePointer函数则在64位值上运行。在伪代码中，它的运行情况如下面所示：

```
LONG InterlockedCompareExchange(PLONG plDestination,
   LONG lExchange, LONG lComparand) {

   LONG lRet = *plDestination;   // Original value

   if (*plDestination == lComparand)
      *plDestination = lExchange;
   return(lRet);
}
```

该函数对当前值（plDestination参数指向的值）与lComparand参数中传递的值进行比较。如果两个值相同，那么 *plDestination改为lExchange参数的值。如果 *plDestination中的值与lExchange的值不匹配，*plDestination保持不变。该函数返回*plDestination中的原始值。记住，所有这些操作都是作为一个原子执行单位来进行的。

没有任何互锁函数仅仅负责对值进行读取操作（而不改变这个值），因为这样的函数根本是不需要的。如果线程只是试图读取值的内容，而这个值始终都由互锁函数来修改，那么被读取的值总是一个很好的值。虽然你不知道你读取的是原始值还是更新值，但是你知道它是这两个值中的一个。对于大多数应用程序来说，这一点很重要。此外，当要对共享内存区域（比如

China-pub.com
下载

内存映象文件）中的值的访问进行同步时，互锁函数也可以供多进程中的线程使用（第 9章中包含了几个示例应用程序，以显示如何正确地使用互锁函数）。

虽然Windows还提供了另外几个互锁函数，但是上面介绍的这些函数能够实现其他函数能做的一切功能，甚至更多。下面是两个其他的函数：

```
LONG InterlockedIncrement(PLONG plAddend);

LONG InterlockedDecrement(PLONG plAddend);
```

InterlockedExchangeAdd函数能够取代这些较老的函数。新函数能够递增或递减任何值，老的函数只能加1或减1。

8.2 高速缓存行

如果想创建一个能够在多处理器计算机上运行的高性能应用程序，必须懂得 CPU的高速缓存行。当一个CPU从内存读取一个字节时，它不只是取出一个字节，它要取出足够的字节来填入高速缓存行。高速缓存行由 32或64个字节组成（视CPU而定），并且始终在第32个字节或第64个字节的边界上对齐。高速缓存行的作用是为了提高 CPU运行的性能。通常情况下，应用程序只能对一组相邻的字节进行处理。如果这些字节在高速缓存中，那么 CPU就不必访问内存总线，而访问内存总线需要多得多的时间。

但是，在多处理器环境中，高速缓存行使得内存的更新更加困难，下面这个例子就说明了这一点：

1) CPU1读取一个字节，使该字节和它的相邻字节被读入CPU1的高速缓存行。

2) CPU2读取同一个字节，使得第一步中的相同的各个字节读入CPU2的高速缓存行。

3) CPU1修改内存中的该字节，使得该字节被写入CPU1的高速缓存行。但是该信息尚未写入RAM。

4) CPU2再次读取同一个字节。由于该字节已经放入CPU2的高速缓存行，因此它不必访问内存。但是CPU2将看不到内存中该字节的新值。

这种情况会造成严重的后果。当然，芯片设计者非常清楚这个问题，并且设计它们的芯片来处理这个问题。尤其是，当一个CPU修改高速缓存行中的字节时，计算机中的其他CPU会被告知这个情况，它们的高速缓存行将变为无效。因此，在上面的情况下， CPU2的高速缓存在CPU1修改字节的值时变为无效。在第 4步中，CPU1必须将它的高速缓存内容迅速转入内存，CPU2必须再次访问内存，重新将数据填入它的高速缓存行。如你所见，高速缓存行能够帮助提高运行的速度，但是它们也可能是多处理器计算机上的一个不利因素。

这一切意味着你应该将高速缓存行存储块中的和高速缓存行边界上的应用程序数据组合在一起。这样做的目的是确保不同的CPU能够访问至少由高速缓存行边界分开的不同的内存地址。还有，应该将只读数据（或不常读的数据）与读写数据分开。同时，应该将同一时间访问的数据组合在一起。

下面是设计得很差的数据结构的例子：

```
struct CUSTINFO {
   DWORD     dwCustomerID;     // Mostly read-only
   int       nBalanceDue;      // Read-write
   char      szName[100];      // Mostly read-only
   FILETIME ftLastOrderDate;   // Read-write
};
```

下面是该结构的改进版本：

China-pub.com 下载

```
// Determine the cache line size for the host CPU.
#ifdef _X86_
#define CACHE_ALIGN  32
#endif
#ifdef _ALPHA_
#define CACHE_ALIGN  64
#endif
#ifdef _IA64_
#define CACHE_ALIGN  ??
#endif

#define CACHE_PAD(Name, BytesSoFar) \
   BYTE Name[CACHE_ALIGN - ((BytesSoFar) % CACHE_ALIGN)]

struct CUSTINFO {
   DWORD     dwCustomerID;     // Mostly read-only
   char      szName[100];      // Mostly read-only

   // Force the following members to be in a different cache line.
   CACHE_PAD(bPad1, sizeof(DWORD) + 100);

   int       nBalanceDue;      // Read-write
   FILETIME ftLastOrderDate;   // Read-write

   // Force the following structure to be in a different cache line.
   CACHE_PAD(bPad2, sizeof(int) + sizeof(FILETIME));
};
```

上面定义的CACHE_ALIGN宏是不错的，但是并不很好。问题是必须手工将每个成员变量的字节值输入该宏。如果增加、移动或删除数据成员，也必须更新对 CACHE_PAD宏的调用。将来，Microsoft的C/C++编译器将支持一种新句法，该句法可以更容易地调整数据成员。它的形式类似__declspec(align(32))。

注意 最好是始终都让单个线程来访问数据（函数参数和局部变量是确保做到这一点的最好方法），或者始终让单个CPU访问这些数据（使用线程亲缘性）。如果采取其中的一种方法，就能够完全避免高速缓存行的各种问题。

8.3 高级线程同步

当必须以原子操作方式来修改单个值时，互锁函数家族是相当有用的。你肯定应该先试试它们。但是大多数实际工作中的编程问题要解决的是比单个32位或64位值复杂得多的数据结构。为了以原子操作方式使用更加复杂的数据结构，必须将互锁函数放在一边，使用 Windows提供的其他某些特性。

前面强调了不应该在单处理器计算机上使用循环锁，甚至在多处理器计算机上，也应该小心地使用它们。原因是CPU时间非常宝贵，决不应该浪费。因此需要一种机制，使线程在等待访问共享资源时不浪费CPU时间。

当线程想要访问共享资源，或者得到关于某个“特殊事件”的通知时，该线程必须调用一个操作系统函数，给它传递一些参数，以指明该线程正在等待什么。如果操作系统发现资源可供使用，或者该特殊事件已经发生，那么函数就返回，同时该线程保持可调度状态（该线程可以不必立即执行，它处于可调度状态，可以使用前一章介绍的原则将它分配给一个 CPU）。

China-pub.com
下载

如果资源不能使用，或者特殊事件还没有发生，那么系统便使该线程处于等待状态，使该线程无法调度。这可以防止线程浪费CPU时间。当线程处于等待状态时，系统作为一个代理，代表你的线程来执行操作。系统能够记住你的线程需要什么，当资源可供使用的时候，便自动使该线程退出等待状态，该线程的运行将与特殊事件实现同步。

从实际情况来看，大多数线程几乎总是处于等待状态。当系统发现所有线程有若干分钟均处于等待状态时，系统的强大的管理功能就会发挥作用。

要避免使用的一种方法

如果没有同步对象，并且操作系统不能发现各种特殊事件，那么线程就不得不使用下面要介绍的一种方法使自己与特殊事件保持同步。不过，由于操作系统具有支持线程同步的内置特性，因此决不应该使用这种方法。

运用这种方法时，一个线程能够自己与另一个线程中的任务的完成实现同步，方法是不断查询多个线程共享或可以访问的变量的状态。下面的代码段说明了这个情况：

```
volatile BOOL g_fFinishedCalculation = FALSE;

int WINAPI WinMain(...) {
   CreateThread(..., RecalcFunc, ...);
   .
   .
   .
   // Wait for the recalculation to complete.
   while (!g_fFinishedCalculation)
        ;
   .
   .
   .
}

DWORD WINAPI RecalcFunc(PVOID pvParam) {
   // Perform the recalculation.
   .
   .
   .
   g_fFinishedCalculation = TRUE;
   return(0);
}
```

如你所见，当主线程（执行WinMain）必须使自己与RecalcFunc函数的完成运行实现同步时，它并没有使自己进入睡眠状态。由于主线程没有进入睡眠状态，因此操作系统继续为它调度CPU时间，这就要占用其他线程的宝贵时间周期。

前面代码段中使用的查询方法存在的另一个问题是，BOOL变量g_f FinishedCalculation从来没有被设置为TRUE。当主线程的优先级高于执行RecalcFunc函数的线程时，就会发生这种情况。在这种情况下，系统决不会将任何时间片分配给RecalcFunc线程。如果执行WinMain函数的线程被置于睡眠状态，而不是进行查询，那么这就不是已调度的时间。系统可以将时间调度给低优先级的线程，如RecalcFunc线程，使它们得以运行。

应该说，有时查询迟早都可以进行，毕竟是循环锁执行的操作。不过有些方法进行这项操作是恰当的，而有些方法是不恰当的。一般来说，应该调用一些函数，使线程进入睡眠状态，直到线程需要的资源可供使用为止。下一节将介绍一种正确的方法。

首先，在前面介绍的代码段的开头，你会发现它使用了volatile一词。为了使这个代码段更加接近工作状态，必须有一个volatile类型的限定词。它告诉编译器，变量可以被应用程序本身以外的某个东西进行修改，这些东西包括操作系统，硬件或同时执行的线程等。尤其是，

China-pub.com 下载

volatile限定词会告诉编译器，不要对该变量进行任何优化，并且总是重新加载来自该变量的内存单元的值。比如，编译器为前面的代码段中的while语句生成了下面的伪代码：

```
MOV   Reg0, [g_fFinishedCalculation]  ; Copy the value into a register
Label: TEST  Reg0, 0                  ; Is the value 0?
JMP   Reg0 == 0, Label                ; The register is 0, try again
...                                   ; The register is not 0 (end of loop)
```

如果不使布尔变量具备易变性，编译器就能像上面所示的那样优化你的C代码。为了实现这样的优化，编译器只需将BOOL变量的值装入一个CPU寄存器一次。然后，它对该CPU寄存器反复进行测试。这样得出的性能当然要比不断地重复读取内存地址中的值并对它进行重复测试要好，因此，优化编译器能够编写上面所示的那种代码。但是，如果编译器进行这样的操作，线程就会进入一个无限循环，永远无法唤醒。另外，使一个结构具备易变性，可以确保它的所有成员都具有易变性，当它们被引用时，总是可以从内存中读取它们。

你也许会问，循环变量g_fResourceInUse是否应该声明为volatile变量。答案是不必，因为我们将该变量的地址传递给各个不同的互锁函数，而不是传递变量值本身。当将一个变量地址传递给一个函数时，该函数必须从内存读取该值。优化程序不会对它产生任何影响。

8.4 关键代码段

关键代码段是指一个小代码段，在代码能够执行前，它必须独占对某些共享资源的访问权。这是让若干行代码能够“以原子操作方式”来使用资源的一种方法。所谓原子操作方式，是指该代码知道没有别的线程要访问该资源。当然，系统仍然能够抑制你的线程的运行，而抢先安排其他线程的运行。不过，在线程退出关键代码段之前，系统将不给想要访问相同资源的其他任何线程进行调度。

下面是个有问题的代码，它显示了不使用关键代码段会发生什么情况：

```
const int MAX_TIMES = 1000;
int   g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];

DWORD WINAPI FirstThread(PVOID pvParam) {

   while (g_nIndex < MAX_TIMES) {
      g_dwTimes[g_nIndex] = GetTickCount();
      g_nIndex++;
   }
   return(0);
}


DWORD WINAPI SecondThread(PVOID pvParam) {

   while (g_nIndex < MAX_TIMES) {
      g_nIndex++;
      g_dwTimes[g_nIndex - 1] = GetTickCount();
   }
   return(0);
}
```

如果分开来看，这两个线程函数将会产生相同的结果，不过每个函数的编码略有不同。如果FirstThread函数自行运行，它将用递增的值填入g_dwTimes数组。如果SecondThread函数也

China-pub.com
下载

是自行运行，那么情况也一样。在理想的情况下，我们希望两个线程能够同时运行，并且仍然使g_dwTimes数组能够产生递增的值。但是，上面的代码存在一个问题，那就是 g_dwTimes不会被正确地填入数据，因为两个线程函数要同时访问相同的全局变量。

下面是如何出现这种情况的一个例子。比如说，我们刚刚在只有一个 CPU的系统上启动执行两个线程。操作系统首先启动运行 SecondThread（这种情况很可能出现），当SecondThread将g_nIndex递增为1之后，系统就停止该线程的运行，而让 FirstThread运行。这时FirstThread将g_dwTimes[1]设置为系统时间，然后系统停止该线程的运行，将　CPU时间重新赋予SecondThread线程。然后SecondThread将g_dwTimes[1 －1]设置为新的系统时间。由于这个操作发生在较晚的时间，因此新系统时间的值大于放入 FirstThread数组中的时间值，另外要注意，g_dwTimes的索引1填在索引0的前面。数组中的数据被破坏了。

应该说明的是，这个例子的设计带有一定的故意性，因为要设计一个实际工作中的例子而不使用好几页的源代码是很难的。不过，通过这个例子，能够看到这个问题在实际工作中有些什么表现。考虑一下管理一个链接对象列表的情况。如果对该链接列表的访问没有取得同步，那么一个线程可以将一个项目添加给这个列表，而另一个线程则试图搜索该列表中的一个项目。如果两个线程同时给这个列表添加项目，那么这种情况会变得更加复杂。通过运用关键代码段，就能够确保在各个线程之间协调对数据结构的访问。

既然已经了解了存在的所有问题，那么下面让我们用关键代码段来修正这个代码：

```
const int MAX_TIMES = 1000;
int   g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {

   while (g_nIndex < MAX_TIMES) {
      EnterCriticalSection(&g_cs);
      g_dwTimes[g_nIndex] = GetTickCount();
      g_nIndex++;
      LeaveCriticalSection(&g_cs);
   }
   return(0);
}


DWORD WINAPI SecondThread(PVOID pvParam) {

   while (g_nIndex < MAX_TIMES) {
      EnterCriticalSection(&g_cs);
      g_nIndex++;
      g_dwTimes[g_nIndex - 1] = GetTickCount();
      LeaveCriticalSection(&g_cs);
   }
   return(0);
}
```

这里指定了一个CRITICAL_SECTION数据结构g_cs，然后在对EnterCriticalSection和LeaveCriticalSection函数调用中封装了要接触共享资源（在这个例子中为g_nIndex和g_dwTimes）的任何代码。注意，在对EnterCriticalSection和LeaveCriticalSection的所有调用中，我传递了g_cs的地址。

China-pub.com 下载

有一个关键问题必须记住。当拥有一项可供多个线程访问的资源时，应该创建一个CRITICAL_SECTION结构。由于我是在飞行旅途上编写这个代码的，让我描绘下面这个模拟情况。CRITICAL_SECTION就像飞机上的厕所，抽水马桶是你要保护的数据。由于厕所很小，每次只能一个人（线程）进入厕所（关键代码段）使用抽水马桶（受保护的资源）。

如果有多个资源总是被一道使用，可以将它们全部放在一个厕所里，也就是说可以创建一个CRITICAL_SECTION结构来保护所有的资源。

如果有多个不是一道使用的资源，比如线程1和线程2访问一个资源，而线程1和线程3访问另一个资源，那么应该为每个资源创建一个独立的厕所，即CRITICAL_SECTION结构。

现在，无论在何处拥有需要访问资源的代码，都必须调用EnterCriticalSection函数，为它传递用于标识该资源的CRITICAL_SECTION结构的地址。这就是说，当一个线程需要访问一个资源时，它首先必须检查厕所门上的“有人”标志。CRITICAL_SECTION结构用于标识线程想要进入哪个厕所，而EnterCriticalSection函数则是线程用来检查“有人”标志的函数。

如果EnterCriticalSection函数发现厕所中没有任何别的线程（门上的标志显示“无人”），那么调用线程就可以使用该资源。如果EnterCriticalSection发现厕所中有另一个线程正在使用，那么调用函数必须在厕所门的外面等待，直到厕所中的另一个线程离开厕所。

当一个线程不再执行需要访问资源的代码时，它应该调用LeaveCriticalSection函数。这样，它就告诉系统，它准备离开包含该资源的厕所。如果忘记调用LeaveCriticalSection，系统将认为该线程仍然在厕所中，因此不允许其他正在等待的线程进入厕所。这就像离开了厕所但没有换上“无人”的标志。

> **注意** 最难记住的一件事情是，编写的需要使用共享资源的任何代码都必须封装在EnterCriticalSection和LeaveCriticalSection函数中。如果忘记将代码封装在一个位置，共享资源就可能遭到破坏。例如，如果我删除了FristThread线程对EnterCriticalSection和LeaveCriticalSection的调用，g_nIndex和g_dwTimes变量就会遭到破坏。即使SecondThread线程仍然正确地调用EnterCriticalSection和LeaveCriticalSection，也会出现这种情况。
>
> 忘记调用EnterCriticalSection和LeaveCriticalSection函数就像是不请求允许进入厕所。线程只是想努力挤入厕所并对资源进行操作。可以想象，只要有一个线程表现出这种相当粗暴的行为，资源就会遭到破坏。

当无法用互锁函数来解决同步问题时，你应该试用关键代码段。关键代码段的优点在于它们的使用非常容易，它们在内部使用互锁函数，这样它们就能够迅速运行。关键代码的主要缺点是无法用它们对多个进程中的各个线程进行同步。不过在第19章中，我将要创建我自己的同步对象，称为Optex。这个对象将显示操作系统如何来实现关键代码段，它也能用于多个进程中的各个线程。

8.4.1 关键代码段准确的描述

现在你已经从理论上对关键代码段有了一定的了解。已经知道为什么它们非常有用，以及它们是如何实现“以原子操作方式”对共享资源进行访问的。下面让我们更加深入地看一看关键代码段是如何运行的。首先介绍一下CRITICAL_SECTION数据结构。如果想查看一下Platform SDK文档中关于该结构的说明，也许你会感到无从下手。那么问题究竟何在呢？

并不是CRITICAL_SECTION结构没有完整的文档，而是Microsoft认为没有必要了解该结

China-pub.com
下载

构的全部情况，这是对的。对于我们来说，这个结构是透明的，该结构有文档可查，但是该结构中的成员变量没有文档。当然，由于这只是个数据结构，可以在 Windows头文件中查找这些信息，可以看到这些数据成员（CRITICAL_SECTION在WinNT.h中定义为RTL_CRITICAL_SECTION；RTL_CRITICAL_SECTION结构在WinBase.h中作了定义）。但是决不应该编写引用这些成员的代码。

若要使用CRITICAL_SECTION结构，可以调用一个Windows函数，给它传递该结构的地址。该函数知道如何对该结构的成员进行操作，并保证该结构的状态始终一致。因此下面让我们将注意力转到这些函数上去。

通常情况下，CRITICAL_SECTION结构可以作为全局变量来分配，这样，进程中的所有线程就能够很容易地按照变量名来引用该结构。但是， CRITICAL_SECTION结构也可以作为局部变量来分配，或者从堆栈动态地进行分配。它只有两个要求，第一个要求是，需要访问该资源的所有线程都必须知道负责保护资源的 CRITICAL_SECTION结构的地址，你可以使用你喜欢的任何机制来获得这些线程的这个地址；第二个要求是， CRITICAL_SECTION结构中的成员应该在任何线程试图访问被保护的资源之前初始化。该结构通过调用下面的函数来进行初始化：

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

该函数用于对（pcs指向的）CRITICAL_SECTION结构的各个成员进行初始化。由于该函数只是设置了某些成员变量。因此它的运行不会失败，并且它的原型采用了 VOID的返回值。该函数必须在任何线程调用 EnterCriticalSection函数之前被调用。Platform SDK的文档清楚地说明，如果一个线程试图进入一个未初始化的 CRTICAL_SECTION，那么结果将是很难预计的。

当知道进程的线程不再试图访问共享资源时，应该通过调用下面的函数来清除该CRITICAL_SECTION结构：

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

DeleteCriticalSection函数用于对该结构中的成员变量进行删除。当然，如果有任何线程仍然使用关键代码段，那么不应该删除该代码段。同样， Platform SDK文档清楚地说明如果删除了关键代码段，其结果就无法知道。当编写要使用共享资源的代码时，必须在该代码的前面放置对下面的函数的调用：

```
VOID EnterCriticalSection(PCRITICAL_SECTION pcs);
```

EnterCriticalSection函数负责查看该结构中的成员变量。这些变量用于指明当前是哪个变量正在访问该资源。EnterCriticalSection负责进行下列测试：

- 如果没有线程访问该资源，EnterCriticalSection便更新成员变量，以指明调用线程已被赋予访问权并立即返回，使该线程能够继续运行（访问该资源）。
- 如果成员变量指明，调用线程已经被赋予对资源的访问权，那么 EnterCriticalSection便更新这些变量，以指明调用线程多少次被赋予访问权并立即返回，使该线程能够继续运行。这种情况很少出现，并且只有当线程在一行中两次调用 EnterCriticalSection而不影响对LeaveCriticalSection的调用时，才会出现这种情况。
- 如果成员变量指明，一个线程（除了调用线程之外）已被赋予对资源的访问权，那么EnerCriticalSection将调用线程置于等待状态。这种情况是极好的，因为等待的线程不会浪费任何CPU时间。系统能够记住该线程想要访问该资源并且自动更新CRITICAL_SECTION的成员变量，一旦目前访问该资源的线程调用 LeaveCriticalSection

China-pub.com 下载

函数，该线程就处于可调度状态。

从内部来讲，EnterCriticalSection函数并不十分复杂。它只是执行一些简单的测试。为什么这个函数是如此有用呢？因为它能够以原子操作方式来执行所有的测试。如果在多处理器计算机上有两个线程在完全相同的时间同时调用 EnterCriticalSection函数，该函数仍然能够正确地起作用，一个线程被赋予对资源的访问权，而另一个线程则进入等待状态。

如果EnterCriticalSection将一个线程置于等待状态，那么该线程在很长时间内就不能再次被调度。实际上，在编写得不好的应用程序中，该线程永远不会再次被赋予 CPU时间。如果出现这种情况，该线程就称为渴求CPU时间的线程。

Windows 2000 在实际操作中，等待关键代码段的线程绝对不会渴求 CPU时间。对EnterCriticalSectiom的调用最终将会超时，导致产生一个异常条件。这时可以将一个调试程序附加给应用程序，以确定究竟出了什么问题。超时的时间量是由下面的注册表子关键字中包含的CriticalSectionTimeout数据值来决定的。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager
```

这个值以秒为单位，默认为2 592 000s，即大约30天。不要将这个值设置得太小（比如小于3s），否则就会对系统中正常等待关键代码段超过 3s的线程和其他应用程序产生不利的影响。

可以使用下面这个函数来代替EnterCriticalSection：

```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

TryEnterCriticalSection函数决不允许调用线程进入等待状态。相反，它的返回值能够指明调用线程是否能够获得对资源的访问权。因此，如果 TryEnterCriticalSection发现该资源已经被另一个线程访问，它就返回FALSE。在其他所有情况下，它均返回TRUE。

运用这个函数，线程能够迅速查看它是否可以访问某个共享资源，如果不能访问，那么它可以继续执行某些其他操作，而不必进行等待。如果 TryEnterCriticalSection函数确实返回了TRUE，那么CRITICAL_SECTION的成员变量已经更新，以便反映出该线程正在访问该资源。因此，对返回TRUE的TryEnterCriticalSection函数的每次调用都必须与对LeaveCriticalSection函数的调用相匹配。

Windows 98 Windows 98没有可以使用的TryEnterCriticalSection函数的实现代码。调用该函数总是返回FALSE。

在接触共享资源的代码结尾处，必须调用下面这个函数：

```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

LeaveCriticalSection要查看该结构中的成员变量。该函数每次计数时要递减1，以指明调用线程多少次被赋予对共享资源的访问权。如果该计数大于0，那么LeaveCriticalSection不做其他任何操作，只是返回而已。

如果该计数变为0，它就要查看在调用EnterCriticalSection中是否有别的线程正在等待。如果至少有一个线程正在等待，它就更新成员变量，并使等待线程中的一个线程（“公正地”选定）再次处于可调度状态。如果没有线程正在等待，LeaveCriticalSection函数就更新成员变量，以指明没有线程正在访问该资源。

与EnterCriticalSection函数一样，LeaveCriticalSection函数也能以原子操作方式执行所有这些测试和更新。不过，LeaveCriticalSection从来不使线程进入等待状态，它总是立即返回。

China-pub.com
下载

8.4.2 关键代码段与循环锁

当线程试图进入另一个线程拥有的关键代码段时，调用线程就立即被置于等待状态。这意味着该线程必须从用户方式转入内核方式（大约1000个CPU周期）。这种转换是要付出很大代价的。在多处理器计算机上，当前拥有资源的线程可以在不同的处理器上运行，并且能够很快放弃对资源的控制。实际上拥有资源的线程可以在另一个线程完成转入内核方式之前释放资源。如果出现这种情况，就会浪费许多CPU时间。

为了提高关键代码段的运行性能，Microsoft将循环锁纳入了这些代码段。因此，当EnterCriticalSection函数被调用时，它就使用循环锁进行循环，以便设法多次取得该资源。只有当为了取得该资源的每次试图都失败时，该线程才转入内核方式，以便进入等待状态。

若要将循环锁用于关键代码段，应该调用下面的函数，以便对关键代码段进行初始化：

```
BOOL InitializeCriticalSectionAndSpinCount(
   PCRITICAL_SECTION pcs,
   DWORD dwSpinCount);
```

与InitializeCriticalSection中的情况一样，InitializeCriticalSectionAndSpinCount的第一个参数是关键代码段结构的地址。但是在第二个参数dwSpinCount中，传递的是在使线程等待之前它试图获得资源时想要循环锁循环迭代的次数。这个值可以是0至0x00FFFFFF之间的任何数字。如果在单处理器计算机上运行时调用该函数，dwSpinCount参数将被忽略，它的计数始终被置为0。这是对的，因为在单处理器计算机上设置循环次数是毫无用处的，如果另一个线程正在循环运行，那么拥有资源的线程就不能放弃它。

通过调用下面的函数，就能改变关键代码段的循环次数：

```
DWORD SetCriticalSectionSpinCount(
   PCRITICAL_SECTION pcs,
   DWORD dwSpinCount);
```

同样，如果主计算机只有一个处理器，那么dwSpinCount的值将被忽略。我认为，始终都应该将循环锁用于关键代码段，因为这样做有百利而无一害。难就难在确定为dwSpinCount参数传递什么值。为了实现最佳的性能，只需要调整这些数字，直到对性能结果满意为止。作为一个指导原则，保护对进程的堆栈进行访问的关键代码段使用的循环次数是4000次。

第10章将要介绍如何实现关键代码段。这种实现将包括循环锁。

8.4.3 关键代码段与错误处理

InitializeCriticalSection函数的运行可能失败（尽管可能性很小）。Microsoft在最初设计该函数时并没有真正想到这个问题，正因为这个原因，该函数的原型才设计为返回VOID。该函数的运行可能失败，因为它分配了一个内存块以便系统得到一些内部调试信息。如果该内存的分配失败，就会出现一个STATUS_NO_MEMORY异常情况。可以使用结构化异常处理（第23、24和25章介绍）来跟踪代码中的这种异常情况。

使用更新的InitializeCriticalSectionAndSpinCount函数，就能够更加容易地跟踪这个问题。该函数也为调试信息分配了内存块，如果内存无法分配，那么它就返回FALSE。

当使用关键代码段时还会出现另一个问题。从内部来说，如果两个或多个线程同时争用关键代码段，那么关键代码段将使用一个事件内核对象（第10章介绍Coptex C++类时，我将要说明如何使用该内核对象）。由于争用的情况很少发生，因此，在初次需要之前，系统将不创建事件内核对象。这可以节省大量的系统资源，因为大多数关键代码段从来不被争用。

China-pub.com 下载

在内存不足的情况下，关键代码段可能被争用，同时系统可能无法创建必要的事件内核对象。这时EnterCriticalSection函数将会产生一个EXCEPTION_INVALID_HANDLE异常。大多数编程人员忽略了这个潜在的错误，在他们的代码中没有专门的处理方法，因为这个错误非常少见。但是，如果想对这种情况有所准备，可以有两种选择。

可以使用结构化异常处理方法来跟踪错误。当错误发生时，既可以不访问关键代码段保护的资源，也可以等待某些内存变成可用状态，然后再次调用 EnterCriticalSection函数。

另一种选择是使用InitializeCriticalSectionAndSpinCount函数创建关键代码段，确保设置了dwSpinCount参数的高信息位。当该函数发现高信息位已经设置时，它就创建该事件内核对象，并在初始化时将它与关键代码段关联起来。如果事件无法创建，该函数返回 FALSE。可以更加妥善地处理代码中的这个事件。如果事件创建成功，你知道 EnterCriticalSection将始终都能运行，并且决不会产生异常情况（如果总是预先分配事件内核对象，就会浪费系统资源。只有当你的代码不能容许EnterCriticalSection运行失败，或者你有把握会出现争用现象，或者你预计进程将在内存非常短缺的环境中运行时，你才能预先分配事件内核对象）。

8.4.4 非常有用的提示和技巧

当使用关键代码段时，有些很好的方法可以使用，而有些方法则应该避免。下面是在使用关键代码段时对你有所帮助的一些提示和技巧。这些技巧也适用于内核对象的同步（下一章介绍）。

1. 每个共享资源使用一个CRITICAL_SECTION变量

如果应用程序中拥有若干个互不相干的数据结构，应该为每个数据结构创建一个CRITICAL_SECTION变量。这比只有单个CRITICAL_SECTION结构来保护对所有共享资源的访问要好，请观察下面这个代码段：

```
int   g_nNums[100];       // A shared resource
TCHAR g_cChars[100];      // Another shared resource
CRITICAL_SECTION g_cs;    // Guards both resources


DWORD WINAPI ThreadFunc(PVOID pvParam) {

   EnterCriticalSection(&g_cs);

   for (int x = 0; x < 100; x++) {
      g_nNums[x]  = 0;
      g_cChars[x] = TEXT('X');
   }

   LeaveCriticalSection(&g_cs);
   return(0);

}
```

这个代码使用单个关键代码段，以便在 g_nNums数组和g_cChars数组初始化时对它们同时实施保护。但是，这两个数组之间毫无关系。当这个循环运行时，没有一个线程能够访问任何一个数组。如果ThreadFunc函数按下面的形式来实现，那么两个数组将分别被初始化：

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
```

China-pub.com
下载


```
   EnterCriticalSection(&g_cs);

   for (int x = 0; x < 100; x++)
      g_nNums[x] = 0;

   for (x = 0; x < 100; x++)
      g_cChars[x] = TEXT('X');

   LeaveCriticalSection(&g_cs);
   return(0);
}
```

从理论上讲，当 g_nNums数组初始化后，另一个只需要访问 g_nNums数组而不需要访问 g_cChars数组的线程就可以开始执行，同时 ThreadFunc可以继续对g_cChars数组进行初始化。但是实际上这是不可能的，因为有一个关键代码段保护着这两个数据结构。为了解决这个问题，可以创建下面两个关键代码段：

```
int g_nNum[100];              // A shared resource
CRITICAL_SECTION g_csNums;    // Guards g_nNums

TCHAR g_cChars[100];          // Another shared resource
CRITICAL_SECTION g_csChars;   // Guards g_cChars

DWORD WINAPI ThreadFunc(PVOID pvParam) {

   EnterCriticalSection(&g_csNums);

   for (int x = 0; x < 100; x++)
      g_nNums[x] = 0;

   LeaveCriticalSection(&g_csNums);

   EnterCriticalSection(&g_csChars);

   for (x = 0; x < 100; x++)
      g_cChars[x] = TEXT('X');

   LeaveCriticalSection(&g_ csChars);
   return(0);
}
```

运用这个实现代码，一旦 ThreadFunc完成对g_nNums数组的初始化，另一个线程就可以开始使用g_nNums数组。也可以考虑让一个线程对 g_nNums数组进行初始化，而另一个线程函数对g_nChars数组进行初始化。

2. 同时访问多个资源

有时需要同时访问两个资源。如果这是 ThreadFunc的要求，可以用下面的代码来实现：

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {

   EnterCriticalSection(&g_csNums);
   EnterCriticalSection(&g_csChars);

   // This loop requires simultaneous access to both resources.
   for (int x = 0; x < 100; x++)
      g_nNums[x] = g_cChars[x];

   LeaveCriticalSection(&g_csChars);
```

China-pub.com 下载

```
   LeaveCriticalSection(&g_csNums);
   return(0);
}
```

假定下面这个函数的进程中的另一个线程也要求访问这两个数组：

```
DWORD WINAPI OtherThreadFunc(PVOID pvParam) {

   EnterCriticalSection(&g_csChars);
   EnterCriticalSection(&g_csNums);

   for (int x = 0; x < 100; x++)
      g_nNums[x] = g_cChars[x];

   LeaveCriticalSection(&g_csNums);
   LeaveCriticalSection(&g_csChars);
   return(0);
}
```

在上面这个函数中我只是切换了对EnterCriticalSection和LeaveCriticalSection函数的调用顺序。但是，由于这两个函数是按上面这种方式编写的，因此可能产生一个死锁状态。假定ThreadFunc开始执行，并且获得了g_csNums关键代码段的所有权，那么执行OtherThreadFunc函数的线程就被赋予一定的CPU时间，并可获得g_csChars关键代码段的所有权。这时就出现了一个死锁状态。当ThreadFunc或OtherThreadFunc中的任何一个函数试图继续执行时，这两个函数都无法取得对它需要的另一个关键代码段的所有权。

为了解决这个问题，必须始终按照完全相同的顺序请求对资源的访问。注意，当调用LeaveCriticalSection函数时，按照什么顺序访问资源是没有关系的，因为该函数决不会使线程进入等待状态。

3. 不要长时间运行关键代码段

当一个关键代码段长时间运行时，其他线程就会进入等待状态，这会降低应用程序的运行性能。下面这个方法可以用来最大限度地减少关键代码段运行所花费的时间。这个代码能够防止其他线程在WM_SOMEMSG消息发送到一个窗口之前改变g_s的值：

```
SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {
   EnterCriticalSection(&g_cs);

   // Send a message to a window.
   SendMessage(hwndSomeWnd, WM_SOMEMSG, &g_s, 0);

   LeaveCriticalSection(&g_cs);
   return(0);
}
```

无法确定窗口过程处理WM_SOMEMSG消息时需要花费多长时间，它可能是几个毫秒，也可能需要几年时间。在这个时间内，其他线程都不能访问g_s结构。这个代码最好编写成下面的形式：

```
SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;
```

China-pub.com
下载


```
DWORD WINAPI SomeThread(PVOID pvParam) {

   EnterCriticalSection(&g_cs);
   SOMESTRUCT sTemp = g_s;
   LeaveCriticalSection(&g_cs);

   // Send a message to a window.
   SendMessage(hwndSomeWnd, WM_SOMEMSG, &sTemp, 0);
   return(0);
}
```

这个代码将该值保存在临时变量 sTemp中。也许你能够猜到CPU需要多长时间来执行这行代码 —— 只需要几个CPU周期。当该临时变量保存后，LeaveCriticalSection函数就立即被调用，因为这个全局结构不再需要保护。上面的第二个实现代码比第一个要好得多，因为其他线程只是在几个CPU周期内被停止使用g_s结构，而不是无限制地停止使用该结构。当然，这个方法的前提是该结构的“瞬态图”应当做到非常好才行，以方便于窗口过程读取。此外，窗口过程不需要改变该结构中的成员。

China-pub.com
下载

第9章　线程与内核对象的同步

上一章介绍了如何使用允许线程保留在用户方式中的机制来实现线程同步的方法。用户方式同步的优点是它的同步速度非常快。如果强调线程的运行速度，那么首先应该确定用户方式的线程同步机制是否适合需要。

虽然用户方式的线程同步机制具有速度快的优点，但是它也有其局限性。对于许多应用程序来说，这种机制是不适用的。例如，互锁函数家族只能在单值上运行，根本无法使线程进入等待状态。可以使用关键代码段使线程进入等待状态，但是只能用这些代码段对单个进程中的线程实施同步。还有，使用关键代码段时，很容易陷入死锁状态，因为在等待进入关键代码段时无法设定超时值。

本章将要介绍如何使用内核对象来实现线程的同步。你将会看到，内核对象机制的适应性远远优于用户方式机制。实际上，内核对象机制的唯一不足之处是它的速度比较慢。当调用本章中提到的任何新函数时，调用线程必须从用户方式转为内核方式。这个转换需要很大的代价：往返一次需要占用x86平台上的大约1000个CPU周期，当然，这还不包括执行内核方式代码，即实现线程调用的函数的代码所需的时间。

本书介绍了若干种内核对象，包括进程，线程和作业。可以将所有这些内核对象用于同步目的。对于线程同步来说，这些内核对象中的每种对象都可以说是处于已通知或未通知的状态之中。这种状态的切换是由Microsoft为每个对象建立的一套规则来决定的。例如，进程内核对象总是在未通知状态中创建的。当进程终止运行时，操作系统自动使该进程的内核对象处于已通知状态。一旦进程内核对象得到通知，它将永远保持这种状态，它的状态永远不会改为未通知状态。

当进程正在运行的时候，进程内核对象处于未通知状态，当进程终止运行的时候，它就变为已通知状态。进程内核对象中是个布尔值，当对象创建时，该值被初始化为FALSE（未通知状态）。当进程终止运行时，操作系统自动将对应的对象布尔值改为TRUE，表示该对象已经得到通知。

如果编写的代码是用于检查进程是否仍在运行，那么只需要调用一个函数，让操作系统去检查进程对象的布尔值，这非常简单。你也可能想要告诉系统使线程进入等待状态，然后当布尔值从FALSE改为TRUE时自动唤醒该线程。这样，你可以编写一个代码，在这个代码中，需要等待子进程终止运行的父进程中的线程只需要使自己进入睡眠状态，直到标识子进程的内核对象变为已通知状态即可。你将会看到，Microsoft的Windows提供了一些能够非常容易地完成这些操作的函数。

刚才讲了Microsoft为进程内核对象定义了一些规则。实际上，线程内核对象也遵循同样的规则。即线程内核对象总是在未通知状态中创建。当线程终止运行时，操作系统会自动将线程对象的状态改为已通知状态。因此，可以将相同的方法用于应用程序，以确定线程是否不再运行。与进程内核对象一样，线程内核对象也可以处于已通知状态或未通知状态。

下面的内核对象可以处于已通知状态或未通知状态：

- 进程
- 线程
- 作业
- 文件修改通知
- 事件
- 可等待定时器

China-pub.com
下载

- 文件
- 信标
- 控制台输入
- 互斥对象

线程可以使自己进入等待状态，直到一个对象变为已通知状态。注意，用于控制每个对象的已通知/未通知状态的规则要根据对象的类型而定。前面已经提到进程和线程对象的规则及作业的规则。

本章将要介绍允许线程等待某个内核对象变为已通知状态所用的函数。然后我们将要讲述Windows提供的专门用来帮助实现线程同步的各种内核对象、如事件、等待计数器，信标和互斥对象。

当我最初开始学习这项内容时，我设想内核对象包含了一面旗帜（在空中飘扬的旗帜，不是耷拉下来的旗帜），这对我很有帮助。当内核对象得到通知时，旗帜升起来；当对象未得到通知时，旗帜就降下来（见图9-1）。

当线程等待的对象处于未通知状态（旗帜降下）中时，这些线程不可调度。但是一旦对象变为已通知状态（旗帜升起），线程看到该标志变为可调度状态，并且很快恢复运行（见图9-2）。

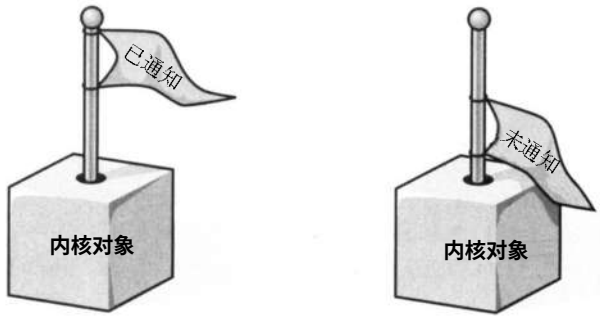


图9-1 内核对象中的旗帜状态

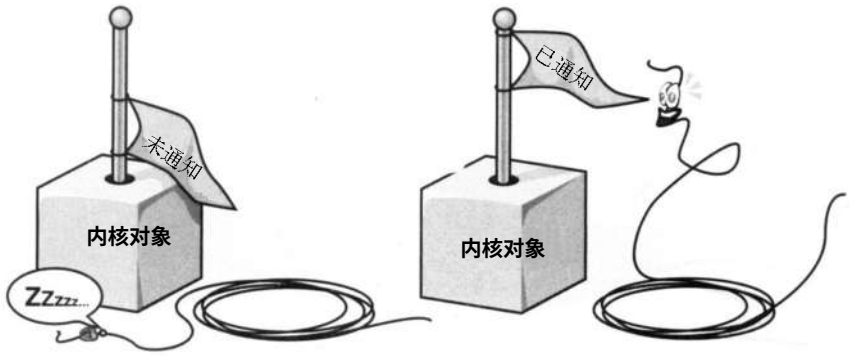


图9-2 内核对象中旗帜状态与线程可调度性示意图

9.1 等待函数

等待函数可使线程自愿进入等待状态，直到一个特定的内核对象变为已通知状态为止。这些等待函数中最常用的是WaitForSingleObject:

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

China-pub.com 下载

当线程调用该函数时，第一个参数 hObject标识一个能够支持被通知/未通知的内核对象（前面列出的任何一种对象都适用）。第二个参数dwMilliseconds允许该线程指明，为了等待该对象变为已通知状态，它将等待多长时间。

调用下面这个函数将告诉系统，调用函数准备等待到hProcess句柄标识的进程终止运行为止：

```
WaitForSingleObject(hProcess, INFINITE);
```

第二个参数告诉系统，调用线程愿意永远等待下去（无限时间量），直到该进程终止运行。

通常情况下，INFINITE是作为第二个参数传递给WaitForSingleObject的，不过也可以传递任何一个值（以毫秒计算）。顺便说一下，INFINITE已经定义为0xFFFFFFFF（或-1）。当然，传递INFINITE有些危险。如果对象永远不变为已通知状态，那么调用线程永远不会被唤醒，它将永远处于死锁状态，不过，它不会浪费宝贵的CPU时间。

下面是如何用一个超时值而不是INFINITE来调用WaitForSingleObject的例子：

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {

   case WAIT_OBJECT_0:
      // The process terminated.
      break;

   case WAIT_TIMEOUT:
      // The process did not terminate within 5000 milliseconds.
      break;

   case WAIT_FAILED:
      // Bad call to function (invalid handle?)
      break;
}
```

上面这个代码告诉系统，在特定的进程终止运行之前，或者在 5000ms时间结束之前，调用线程不应该变为可调度状态。因此，如果进程终止运行，那么这个函数调用将在不到5000ms的时间内返回，如果进程尚未终止运行，那么它在大约 5000ms时间内返回。注意，不能为dwMillisecond传递0。如果传递了0，WaitForSingleObject函数将总是立即返回。

WaitForSingleObject的返回值能够指明调用线程为什么再次变为可调度状态。如果线程等待的对象变为已通知状态，那么返回值是 WAIT_OBJECT_0。如果设置的超时已经到期，则返回值是WAIT_TIMEOUT。如果将一个错误的值（如一个无效句柄）传递给 WaitForSingleObject，那么返回值将是WAIT_FAILED（若要了解详细信息，可调用GetLastError）。

下面这个函数WaitForMultipleObjects与WaitForSingleObject函数很相似，区别在于它允许调用线程同时查看若干个内核对象的已通知状态：

```
DWORD WaitForMultipleObjects(
   DWORD dwCount,
   CONST HANDLE* phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds);
```

dwCount参数用于指明想要让函数查看的内核对象的数量。这个值必须在 1与MAXIMUM_WAIT_OBJECTS（在Windows头文件中定义为64）之间。phObjects参数是指向内核对象句柄的数组的指针。

可以以两种不同的方式来使用WaitForMultipleObjects函数。一种方式是让线程进入等待状

China-pub.com
下载

态，直到指定内核对象中的任何一个变为已通知状态。另一种方式是让线程进入等待状态，直到所有指定的内核对象都变为已通知状态。fWaitAll参数告诉该函数，你想要让它使用何种方式。如果为该参数传递TRUE，那么在所有对象变为已通知状态之前，该函数将不允许调用线程运行。

dwMilliseconds参数的作用与它在WaitForSingleObject中的作用完全相同。如果在等待的时候规定的时间到了，那么该函数无论如何都会返回。同样，通常为该参数传递INFINITE，但是在编写代码时应该小心，以避免出现死锁情况。

WaitForMultipleObjects函数的返回值告诉调用线程，为什么它会被重新调度。可能的返回值是WAIT_FAILED和WAIT_TIMEOUT，这两个值的作用是很清楚的。如果为fWaitAll参数传递TRUE，同时所有对象均变为已通知状态，那么返回值是WAIT_OBJECT_0。如果为fWaitAll传递FALSE，那么一旦任何一个对象变为已通知状态，该函数便返回。在这种情况下，你可能想要知道哪个对象变为已通知状态。返回值是WAIT_OBJECT_0与（WAIT_OBJECT_0+dwCount-1）之间的一个值。换句话说，如果返回值不是WAIT_TIMEOUT，也不是WAIT_FAILED，那么应该从返回值中减去WAIT_OBJECT_0。产生的数字是作为第二个参数传递给WaitForMultipleObjects的句柄数组中的索引。该索引说明哪个对象变为已通知状态。下面是说明这一情况的一些示例代码：

```
HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
   case WAIT_FAILED:
      // Bad call to function (invalid handle?)
      break;

   case WAIT_TIMEOUT:
      // None of the objects became signaled within 5000 milliseconds.
      break;

   case WAIT_OBJECT_0 + 0:
      // The process identified by h[0] (hProcess1) terminated.
      break;

   case WAIT_OBJECT_0 + 1:
      // The process identified by h[1] (hProcess2) terminated.
      break;

   case WAIT_OBJECT_0 + 2:
      // The process identified by h[2] (hProcess3) terminated.
      break;
}
```

如果为fWaitAll参数传递FALSE，WaitForMultipleObjects就从索引0开始向上对句柄数组进行扫描，同时已通知的第一个对象终止等待状态。这可能产生一些你不希望有的结果。例如，通过将3个进程句柄传递给该函数，你的线程就会等待3个子进程终止运行。如果数组中索引为0的进程终止运行，WaitForMultipleObjects就会返回。这时该线程就可以做它需要的任何事情，然后循环反复，等待另一个进程终止运行。如果该线程传递相同的3个句柄，该函数立即再次

China-pub.com 下载

返回WAIT_OBJECT_0。除非删除已经收到通知的句柄，否则代码就无法正确地运行。

9.2 成功等待的副作用

对于有些内核对象来说，成功地调用WaitForSingleObject和WaitForMultipleObjects，实际上会改变对象的状态。成功地调用是指函数发现对象已经得到通知并且返回一个相对于WAIT_OBJECT_0的值。如果函数返回WAIT_TIMEOUT或WAIT_FAILED，那么调用就没有成功。如果函数调用没有成功，对象的状态就不可能改变。

当一个对象的状态改变时，我称之为成功等待的副作用。例如，有一个线程正在等待自动清除事件对象（本章后面将要介绍）。当事件对象变为已通知状态时，函数就会发现这个情况，并将WAIT_OBJECT_0返回给调用线程。但是就在函数返回之前，该事件将被置为未通知状态，这就是成功等待的副作用。

这个副作用将用于自动清除内核对象，因为它是Microsoft为这种类型的对象定义的规则之一。其他对象拥有不同的副作用，而有些对象则根本没有任何副作用。进程和线程内核对象就根本没有任何副作用，也就是说，在这些对象之一上进行等待决不会改变对象的状态。由于本章要介绍各种不同的内核对象，因此我们将要详细说明它们的成功等待的副作用。

究竟是什么原因使得WaitForMultipleObjects函数如此有用呢，因为它能够以原子操作方式来执行它的所有操作。当一个线程调用WaitForMultipleObjects函数时，该函数能够测试所有对象的通知状态，并且能够将所有必要的副作用作为一项操作来执行。

让我们观察一个例子。两个线程以完全相同的方式来调用WaitForMultipleObjects：

```
HANDLE h[2];
h[0] = hAutoResetEvent1;   // Initially nonsignaled
h[1] = hAutoResetEvent2;   // Initially nonsignaled
WaitForMultipleObjects(2, h, TRUE, INFINITE);
```

当WaitForMultipleObjects函数被调用时，两个事件都处于未通知状态，这就迫使两个线程都进入等待状态。然后hAutoResetEvent1对象变为已通知状态。两个线程都发现，该事件已经变为已通知状态，但是它们都无法被唤醒，因为hAutoResetEvent2仍然处于未通知状态。由于两个线程都没有等待成功，因此没有对hAutoResetEvent1对象产生任何副作用。

接着，hAutoResetEvent2变为已通知状态。这时，两个线程中的一个发现，两个对象都变为已通知状态。等待取得了成功，两个事件对象均被置为未通知状态，该线程变为可调度的线程。但是另一个线程的情况如何呢？它将继续等待，直到它发现两个事件对象都处于已通知状态。尽管它原先发现hAutoResetEvent1处于已通知状态，但是现在它将该对象视为未通知状态。

前面讲过，有一个重要问题必须注意，即WaitForMultipleObjects是以原子操作方式运行的。当它检查内核对象的状态时，其他任何线程都无法背着对象改变它的状态。这可以防止出现死锁情况。试想，如果一个线程看到hAutoResetEvent1已经得到通知并将事件重置为未通知状态，然后，另一个线程发现hAutoResetEvent2已经得到通知并将该事件重置为未通知状态，那么这两个线程均将被冻结：一个线程将等待另一个线程已经得到的对象，另一个线程将等待该线程已经得到的对象。WaitForMultipleObjects能够确保这种情况永远不会发生。

这会产生一个非常有趣的问题，即如果多个线程等待单个内核对象，那么当该对象变成已通知状态时，系统究竟决定唤醒哪个线程呢？Microsoft对这个问题的正式回答是：“算法是公平的。”Microsoft不想使用系统使用的内部算法。它只是说该算法是公平的，这意味着如果多

China-pub.com
下载

个线程正在等待，那么每当对象变为已通知状态时，每个线程都应该得到它自己的被唤醒的机会。

这意味着线程的优先级不起任何作用，即高优先级线程不一定得到该对象。这还意味着等待时间最长的线程不一定得到该对象。同时得到对象的线程有可能反复循环，并且再次得到该对象。但是，这对于其他线程来说是不公平的，因此该算法将设法防止这种情况的出现。但是这不一定做得到。

在实际操作中，Microsoft使用的算法是常用的“先进先出”的方案。等待了最长时间的线程将得到该对象。但是系统中将会执行一些操作，以便改变这个行为特性，使它不太容易预测。这就是为什么Microsoft没有明确说明该算法如何起作用的原因。操作之一是让线程暂停运行。如果一个线程等待一个对象，然后该线程暂停运行，那么系统就会忘记该线程正在等待该对象。这是一个特性，因为没有理由为一个暂停运行的线程进行调度。当后来该线程恢复运行时，系统将认为该线程刚刚开始等待该对象。

当调试一个进程时，只要到达一个断点，该进程中的所有线程均暂停运行。因此，调试一个进程会使“先进先出”的算法很难预测其结果，因为线程常常暂停运行，然后再恢复运行。

9.3 事件内核对象

在所有的内核对象中，事件内核对象是个最基本的对象。它们包含一个使用计数（与所有内核对象一样），一个用于指明该事件是个自动重置的事件还是一个人工重置的事件的布尔值，另一个用于指明该事件处于已通知状态还是未通知状态的布尔值。

事件能够通知一个操作已经完成。有两种不同类型的事件对象。一种是人工重置的事件，另一种是自动重置的事件。当人工重置的事件得到通知时，等待该事件的所有线程均变为可调度线程。当一个自动重置的事件得到通知时，等待该事件的线程中只有一个线程变为可调度线程。

当一个线程执行初始化操作，然后通知另一个线程执行剩余的操作时，事件使用得最多。事件初始化为未通知状态，然后，当该线程完成它的初始化操作后，它就将事件设置为已通知状态。这时，一直在等待该事件的另一个线程发现该事件已经得到通知，因此它就变成可调度线程。这第二个线程知道第一个线程已经完成了它的操作。

下面是CreateEvent函数，用于创建事件内核对象：

```
HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL fManualReset,
   BOOL fInitialState,
   PCTSTR pszName);
```

第3章已经介绍了内核对象的操作技巧，比如，如何设置它们的安全性，如何进行使用计数，如何继承它们的句柄，如何按名字共享对象等。由于现在你对所有这些对象都已经熟悉了，所以不再介绍该函数的第一个和最后一个参数。

FMannualReset参数是个布尔值，它能够告诉系统是创建一个人工重置的事件（TRUE）还是创建一个自动重置的事件（FALSE）。fInitialState参数用于指明该事件是要初始化为已通知状态（TRUE）还是未通知状态（FALSE）。当系统创建事件对象后，createEvent就将与进程相关的句柄返回给事件对象。其他进程中的线程可以获得对该对象的访问权，方法是使用在pszName参数中传递的相同值，使用继承性，使用 DuplicateHandle函数等来调用CreateEvent，或者调用OpenEvent,在pszName参数中设定一个与调用CreateEvent时设定的名字相匹配的名字：

China-pub.com 下载

```
HANDLE OpenEvent(
   DWORD fdwAccess,
   BOOL fInherit,
   PCTSTR pszName);
```

与所有情况中一样，当不再需要事件内核对象时，应该调用CloseHandle函数。

一旦事件已经创建，就可以直接控制它的状态。当调用SetEvent时，可以将事件改为已通知状态：

```
BOOL SetEvent(HANDLE hEvent);
```

当调用ResetEvent函数时，可以将该事件改为未通知状态：

```
BOOL ResetEvent(HANDLE hEvent);
```

就是这么容易。

Microsoft为自动重置的事件定义了应该成功等待的副作用规则，即当线程成功地等待到该对象时，自动重置的事件就会自动重置到未通知状态。这就是自动重置的事件如何获得它们的名字的方法。通常没有必要为自动重置的事件调用ResetEvent函数，因为系统会自动对事件进行重置。但是，Microsoft没有为人工重置的事件定义成功等待的副作用。

让我们观察一个简单的例子，以便说明如何使用事件内核对象对线程进行同步。下面就是这个代码：

```
// Create a global handle to a manual-reset, nonsignaled event.
HANDLE g_hEvent;

int WINAPI WinMain(...) {

   // Create the manual-reset, nonsignaled event.
   g_hEvent = CreateEve\nt(NULL, TRUE, FALSE, NULL);

   // Spawn 3 new threads.
   HANDLE hThread[3];
   DWORD dwThreadID;
   hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);
   hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);
   hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadID);


   OpenFileAndReadContentsIntoMemory(...);

   // Allow all 3 threads to access the memory.
   SetEvent(g_hEvent);
   ...
}

DWORD WINAPI WordCount(PVOID pvParam) {
   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}

DWORD WINAPI SpellCheck (PVOID pvParam) {
```

China-pub.com
下载


```
   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}
```

当这个进程启动时，它创建一个人工重置的未通知状态的事件，并且将句柄保存在一个全局变量中。这使得该进程中的其他线程能够非常容易地访问同一个事件对象。现在 3个线程已经产生。这些线程要等待文件的内容读入内存，然后每个线程都要访问它的数据。一个线程进行单词计数，另一个线程运行拼写检查器，第三个线程运行语法检查器。这 3个线程函数的代码的开始部分都相同，每个函数都调用 WaitForSingleObject，这将使线程暂停运行，直到文件的内容由主线程读入内存为止。

一旦主线程将数据准备好，它就调用 SetEvent，给事件发出通知信号。这时，系统就使所有这3个辅助线程进入可调度状态，它们都获得了 CPU时间，并且可以访问内存块。注意，这3个线程都以只读方式访问内存。这就是所有 3个线程能够同时运行的唯一原因。还要注意，如何计算机上配有多个 CPU，那么所有3个线程都能够真正地同时运行，从而可以在很短的时间内完成大量的操作。

如果你使用自动重置的事件而不是人工重置的事件，那么应用程序的行为特性就有很大的差别。当主线程调用 SetEvent之后，系统只允许一个辅助线程变成可调度状态。同样，也无法保证系统将使哪个线程变为可调度状态。其余两个辅助线程将继续等待。

已经变为可调度状态的线程拥有对内存块的独占访问权。让我们重新编写线程的函数，使得每个函数在返回前调用 SetEvent函数（就像WinMain函数所做的那样）。这些线程函数现在变成下面的形式：

```
DWORD WINAPI WordCount(PVOID pvParam) {

   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   SetEvent(g_hEvent);
   return(0);
}

DWORD WINAPI SpellCheck (PVOID pvParam) {

   // Wait until the file's data is in memory.
```

China-pub.com 下载

```
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   SetEvent(g_hEvent);
   return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   SetEvent(g_hEvent);
   return(0);
}
```

当线程完成它对数据的专门传递时，它就调用 SetEvent函数，该函数允许系统使得两个正在等待的线程中的一个成为可调度线程。同样，我们不知道系统将选择哪个线程作为可调度线程，但是该线程将进行它自己的对内存块的专门传递。当该线程完成操作时，它也将调用SetEvent函数，使第三个即最后一个线程进行它自己的对内存块的传递。注意，当使用自动重置事件时，如果每个辅助线程均以读 /写方式访问内存块，那么就不会产生任何问题，这些线程将不再被要求将数据视为只读数据。这个例子清楚地展示出使用人工重置事件与自动重置事件之间的差别。

为了完整起见，下面再介绍一个可以用于事件的函数：

```
BOOL PulseEvent(HANDLE hEvent);
```

PulseEvent函数使得事件变为已通知状态，然后立即又变为未通知状态，这就像在调用SetEvent后又立即调用ResetEvent函数一样。如果在人工重置的事件上调用 PulseEvent函数，那么在发出该事件时，等待该事件的任何一个线程或所有线程将变为可调度线程。如果在自动重置事件上调用PulseEvent函数，那么只有一个等待该事件的线程变为可调度线程。如果在发出事件时没有任何线程在等待该事件，那么将不起任何作用。

PulseEvent函数并不非常有用。实际上我在自己的应用程序中从未使用它，因为根本不知道什么线程将会看到事件的发出并变成可调度线程。由于在调用 PulseEvent时无法知道任何线程的状态，因此该函数并不那么有用。我相信在有些情况下，虽然 PulseEvent函数可以方便地供你使用，但是你根本想不起要去使用它。关于 PulseEvent函数的比较详细的说明，请参见本章后面对SingleObjectAndWait函数的介绍。

Handshake示例应用程序

清单9-1中列出了Handshake（“09 Handshake.exe ”）应用程序，它展示了自动重置事件的使用情况。该应用程序的源代码文件和资源文件均在本书所附光盘上的 09-Handshake目录下。当运行Handshake应用程序时，就会出现图9-3所示的对话框。

Handshake应用程序接受一个请求字符串，再将该字符串中的所有字符反转，然后将结果放入Result域。Handshake应用程序的出色之处在于它完成这个重要任务时所用的方法不同一般。

Handshake能够解决常见的编程问题。现在有一个客户机和一个服务器，它们之间需要互

China-pub.com
下载

相进行通信。开始时，服务器无事可做，因此它进入等待状态。当客户机准备将一个请求提交给服务器时，它将该请求放入一个共享内存缓冲区中，然后发出一个事件通知，这样，服务器线程就会知道查看数据缓冲区并处理客户机的请求。当服务器线程忙于处理该请求的时候，客户机的线程必须进入等待状态，直到服务器准备好请求的结果为止。因此客户机进入等待状态，直到服务器发出另一个事件通知，指明结果已经准备好，可供客户机进行处理。当客户机再次被唤醒的时候，它就知道结果已经放入共享数据缓冲区中，并且可以将结果显示给用户。

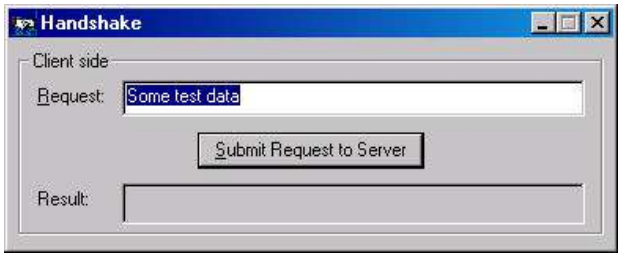


图9-3 Handshake 对话框

当该应用程序启动运行时，它立即创建两个未通知的自动重置的事件对象。一个事件是 g_hevtRequestSubmitted，用于指明何时为服务器准备一个请求。该事件由服务器线程等待，并由客户机线程发出通知。第二个事件是g_hevtResultReturned，用来指明何时为客户机准备好结果。客户机线程等待该事件，而服务器线程则负责发出该事件的通知。

当各个事件创建后，服务器线程就产生并且执行 ServerThread函数。该函数立即让服务器等待客户机的请求。与此同时，主线程（它也是客户机线程）调用 DialogBox函数，该函数负责显示应用程序的用户界面。可以将一些文字输入Request域，然后，当点击Submit Request To Server(将请求提交给服务器)时，请求字符串将被放入由客户机和服务器线程共享的一个缓冲区，并发出g_hevtRequestSubmitted事件的通知。然后客户机线程通过等待g_hevtResultReturened事件来等待服务器的结果。

服务器醒来，将共享内存缓冲区中的字符串反转，然后发出 g_hevtResultReturned事件的通知。服务器的线程循环运行，以便等待客户机的另一个请求。注意，该应用程序决不会调用ResetEvent函数，因为没有必要。自动重置的事件在等待成功后会自动恢复未通知状态。与此同时，客户机线程发现g_hevtResultReturned事件已经变为已通知状态。它醒来，并将字符串从共享内存缓冲区拷贝到用户界面的Result域。

也许这个应用程序剩下的唯一的一个值得注意的特性是它如何关闭。若要关闭该应用程序，只需要关闭它的对话框。这会导致调用的 _tWinMain中的DialogBox函数返回。这时，主线程将一个特殊字符串拷贝到共享缓冲区，并唤醒服务器的线程，以便处理该特殊请求。主线程等待服务器线程确认请求已经收到，并等待服务器线程终止运行。当服务器线程发现该特殊的客户机请求字符串时，它就退出循环，而该线程则终止运行。

我选择让主线程等待服务器线程终止运行，方法是调用WaitForMultipleObjects函数，这样，就可以看到该函数是如何使用的。实际上，也可以调用 WaitForSingleObject函数，传递服务器线程的句柄，一切将以完全相同的方式来运行。

一旦主线程知道服务器线程已经停止运行后，我将 3次调用CloseHandle函数，以便正确地撤消应用程序正在使用的所有内核对象。当然，系统能够自动执行这项操作，但是如果我自己进行操作，我的感觉会更好些。我喜欢能够随时控制我的代码。

China-pub.com 下载

清单9-1 Handshake示例应用程序



Handshake.cpp

```
/******************************************************************************
Module:  Handshake.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>       // For beginthreadex
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


// This event is signaled when the client has a request for the server
HANDLE g_hevtRequestSubmitted;

// This event is signaled when the server has a result for the client
HANDLE g_hevtResultReturned;

// The buffer shared between the client and server threads
TCHAR  g_szSharedRequestAndResultBuffer[1024];
// The special value sent from the client that causes the
// server thread to terminate cleanly.
TCHAR  g_szServerShutdown[] = TEXT("Server Shutdown");


//////////////////////////////////////////////////////////////////////////////


// This is the code executed by the server thread
DWORD WINAPI ServerThread(PVOID pvParam) {

   // Assume that the server thread is to run forever
   BOOL fShutdown = FALSE;

   while (!fShutdown) {

      // Wait for the client to submit a request
      WaitForSingleObject(g_hevtRequestSubmitted, INFINITE);

      // Check to see if the client wants the server to terminate
      fShutdown =
         (lstrcmpi(g_szSharedRequestAndResultBuffer, g_szServerShutdown) == 0);

      if (!fShutdown) {
         // Process the client's request (reverse the string)
         _tcsrev(g_szSharedRequestAndResultBuffer);
      }
```

China-pub.com
下载


```
      // Let the client process the request's result
      SetEvent(g_hevtResultReturned);
   }

   // The client wants us to shutdown, exit
   return(0);
}

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_HANDSHAKE);

   // Initialize the edit control with some test data request
   Edit_SetText(GetDlgItem(hwnd, IDC_REQUEST), TEXT("Some test data"));

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {

      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_SUBMIT:  // Submit a request to the server thread

         // Copy the request string into the shared data buffer
         Edit_GetText(GetDlgItem(hwnd, IDC_REQUEST),
            g_szSharedRequestAndResultBuffer,
            chDIMOF(g_szSharedRequestAndResultBuffer));

         // Let the server thread know that a request is ready in the buffer
         SetEvent(g_hevtRequestSubmitted);

         // Wait for the server to process the request and give us the result
         WaitForSingleObject(g_hevtResultReturned, INFINITE);

         // Let the user know the result
         Edit_SetText(GetDlgItem(hwnd, IDC_RESULT),
            g_szSharedRequestAndResultBuffer);

         break;
   }
}

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Create & initialize the 2 nonsignaled, auto-reset events
   g_hevtRequestSubmitted = CreateEvent(NULL, FALSE, FALSE, NULL);
   g_hevtResultReturned   = CreateEvent(NULL, FALSE, FALSE, NULL);

   // Spawn the server thread
   DWORD dwThreadID;
   HANDLE hThreadServer = chBEGINTHREADEX(NULL, 0, ServerThread, NULL,
      0, &dwThreadID);

   // Execute the client thread's user-interface
   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_HANDSHAKE), NULL, Dlg_Proc);

   // The client's UI is closing, have the server thread shutdown
   lstrcpy(g_szSharedRequestAndResultBuffer, g_szServerShutdown);
   SetEvent(g_hevtRequestSubmitted);

   // Wait for the server thread to acknowledge the shutdown AND
   // wait for the server thread to fully terminate
   HANDLE h[2];
   h[0] = g_hevtResultReturned;
   h[1] = hThreadServer;
   WaitForMultipleObjects(2, h, TRUE, INFINITE);

   // Properly clean up everything
   CloseHandle(hThreadServer);
   CloseHandle(g_hevtRequestSubmitted);
   CloseHandle(g_hevtResultReturned);

   // The client thread terminates with the whole process
   return(0);
}
//////////////////////////////// End of File //////////////////////////////////
```

Handshake.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
```

China-pub.com
下载


```
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_HANDSHAKE DIALOG DISCARDABLE  0, 0, 256, 81
STYLE DS_CENTER | WS_MINIMIZEBOX | WS_CAPTION | WS_SYSMENU
CAPTION "Handshake"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "Client side",IDC_STATIC,4,4,248,72
    LTEXT           "&Request:",IDC_STATIC,12,18,30,8
    EDITTEXT        IDC_REQUEST,48,16,196,14,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "&Submit Request to Server",IDC_SUBMIT,80,36,96,14
    LTEXT           "Result:",IDC_STATIC,12,58,23,8
    EDITTEXT        IDC_RESULT,48,56,196,16,ES_AUTOHSCROLL | ES_READONLY
END
/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_HANDSHAKE, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 249
        TOPMARGIN, 7
        BOTTOMMARGIN, 74
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
```

China-pub.com 下载

```
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_HANDSHAKE           ICON    DISCARDABLE     "Handshake.ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

9.4 等待定时器内核对象

等待定时器是在某个时间或按规定的间隔时间发出自己的信号通知的内核对象。它们通常用来在某个时间执行某个操作。

若要创建等待定时器，只需要调用CreateWaitableTimer函数：

```
HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL fManualReset,
   PCTSTR pszName);
```

psa和pszName这两个参数在第3章中做过介绍。当然，进程可以获得它自己的与进程相关的现有等待定时器的句柄，方法是调用OpenWaitableTimer函数：

```
HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

China-pub.com
下载

与事件的情况一样，fManualReset参数用于指明人工重置的定时器或自动重置的定时器。当发出人工重置的定时器信号通知时，等待该定时器的所有线程均变为可调度线程。当发出自动重置的定时器信号通知时，只有一个等待的线程变为可调度线程。

等待定时器对象总是在未通知状态中创建。必须调用 SetWaitableTimer函数来告诉定时器你想在何时让它成为已通知状态：

```
BOOL SetWaitableTimer(
   HANDLE hTimer,
   const LARGE_INTEGER *pDueTime,
   LONG lPeriod,
   PTIMERAPCROUTINE pfnCompletionRoutine,
   PVOID pvArgToCompletionRoutine,
   BOOL fResume);
```

这个函数带有若干个参数，使用时很容易搞混。显然， hTimer参数用于指明你要设置的定时器。pDueTime和lPeriod两个参数是一道使用的。PDueTimer参数用于指明定时器何时应该第一次报时，而lPeriod参数则用于指明此后定时器应该间隔多长时间报时一次。下面的代码用于将定时器的第一次报时的时间设置在2002年1月1日的下午1点钟，然后每隔6小时报时一次：

```
// Declare our local variables.
HANDLE hTimer;
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
LARGE_INTEGER liUTC;

// Create an auto-reset timer.
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// First signaling is at January 1, 2002, at 1:00 P.M. (local time).
st.wYear         = 2002; // Year
st.wMonth        = 1;    // January
st.wDayOfWeek    = 0;    // Ignored
st.wDay          = 1;    // The first of the month
st.wHour         = 13;   // 1PM
st.wMinute       = 0;    // 0 minutes into the hour
st.wSecond       = 0;    // 0 seconds into the minute
st.wMilliseconds = 0;    // 0 milliseconds into the second

SystemTimeToFileTime(&st, &ftLocal);
// Convert local time to UTC time.
LocalFileTimeToFileTime(&ftLocal, &ftUTC);
// Convert FILETIME to LARGE_INTEGER because of different alignment.
liUTC.LowPart  = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;

// Set the timer.
SetWaitableTimer(hTimer, &liUTC, 6 * 60 * 60 * 1000,
   NULL, NULL, FALSE);
.
.
.
```

代码首先对SYSTEMTIME结构进行初始化，该结构用于指明定时器何时第一次报时（发出信号通知）。我将该时间设置为本地时间，即计算机所在时区的正确时间。 SetWaitableTimer的

China-pub.com 下载

第二个参数的原型是个常量LARGE_INTEGER *，因此它不能直接接受SYSTEMTIME结构。但是，FILETIME结构和LARGE_INTEGER结构拥有相同的二进制格式，都包含两个 32位的值。因此，我们可以将SYSTEMTIME结构转换成FILETIME结构。再一个问题是，SetWaitableTimer希望传递给它的时间始终都采用世界协调时（UTC）的时间。调用LocalFileTimeToFileTime函数，就可以很容易地进行时间的转换。

由于FILETIME和LARGE_INTEGER结构具有相同的二进制格式，因此可以像下面这样将FILETIME结构的地址直接传递给SetWaitableTimer：

```
// Set the timer.
SetWaitableTimer(hTimer, (PLARGE_INTEGER) &ftUTC,
   6 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

实际上，这是我最初的做法。但是这是个大错误。虽然 FILETIME和LARGE_INTEGER结构采用相同的二进制格式，但是这两个结构的调整要求不同。所有 FILETIME结构的地址必须从一个32位的边界开始，而所有 LARGE_INTEGER结构的地址则必须从 64位的边界开始。调用SetWaitableTimer函数和给它传递一个 FILETIME结构时是否能够正确地运行，取决于FILETIME结构是否恰好位于64位的边界上。但是，编译器能够确保LARGE_INTEGER结构总是从64位的边界开始，因此要进行的正确操作（也就是所有时间都能保证起作用的操作）是将FILETIME的成员拷贝到LARGE_INTEGER的成员中，然后将LARGE_INTEGER的地址传递给SetWaitableTimer。

注意 x86处理器能够悄悄地处理未对齐的数据引用。因此当应用程序在 x86 CPU上运行时，将FILETIME的地址传递给SetWaitablrTimer总是可行的。但是，其他处理器，如Alpha处理器，则无法像x86处理器那样悄悄地处理未对齐的数据引用。实际上，大多数其他处理器都会产生一个 EXCEPTION_DATATYPE_MISALIGNMENT异常，它会导致进程终止运行。当你将 x86计算机上运行的代码移植到其他处理器时，产生问题的最大原因是出现了对齐错误。如果现在注意对齐方面的问题，就能够在以后省去几个月的代码移植工作。关于对齐问题的详细说明，参见第 13章。

现在，若要使定时器在 2002年1月1日下午1点之后每隔6h进行一次报时，我们应该将注意力转向lPeriod参数。该参数用于指明定时器在初次报时后每隔多长时间（以毫秒为单位）进行一次报时。如果是每隔 6h进行一次报时，那么我传递 21 600 000（6h×每小时60min×每分钟60s×每秒1000ms）。另外，如果给它传递了以前的一个绝对时间，比如 1975年1月1日下午1点，那么SetWaitableTimer的运行就不会失败。

如果不设置定时器应该第一次报时的绝对时间，也可以让定时器在一个相对于调用SetWaitableTimer的时间进行报时。只需要在 pDueTime参数中传递一个负值。传递的值必须是以100ns为间隔。由于我们通常并不以 100ns的间隔来思考问题，因此我们要说明一下 100ns的具体概念：1s=1000ms=1000000μs=1000000000ns 。

下面的代码用于将定时器设置为在调用 SetWaitableTimer函数后5s第一次报时：

```
// Declare our local variables.
HANDLE hTimer;
LARGE_INTEGER li;

// Create an auto-reset timer.
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// Set the timer to go off 5 seconds after calling SetWaitableTimer.
```

China-pub.com
下载


```
// Timer unit is 100-nanoseconds.
const int nTimerUnitsPerSecond = 10000000;

// Negate the time so that SetWaitableTimer knows we
// want relative time instead of absolute time.
li.QuadPart = -(5 * nTimerUnitsPerSecond);
// Set the timer.
SetWaitableTimer(hTimer, &li, 6 * 60 * 60 * 1000,
   NULL, NULL, FALSE);

:
```

通常情况下，你可能想要一个一次报时的定时器，它只是发出一次报时信号，此后再也不发出报时信号。若要做到这一点，只需要为lPeriod参数传递0即可。然后可以调用CloseHandle函数，关闭定时器，或者再次调用SetWaitableTimer函数，重新设置时间，为它规定一个需要遵循的新条件。

SetWaitableTimer的最后一个参数是fResume，它可以用于支持暂停和恢复的计算机。通常可以为该参数传递FALSE，就像我在上面这个代码段中设置的那样。但是，如果你编写了一个会议安排类型的应用程序，在这个应用程序在中，你想设置一个为用户提醒会议时间安排的定时器，那么应该传递TRUE。当定时器报时的时候，它将使计算机摆脱暂停方式（如果它处于暂停状态的话），并唤醒等待定时器报时的线程。然后该应用程序运行一个波形文件，并显示一个消息框，告诉用户即将举行的会议。如果为fResume参数传递FALSE，定时器对象就变为已通知状态，但是它唤醒的线程必须等到计算机恢复运行（通常由用户将它唤醒）之后才能得到CPU时间。

除了上面介绍的定时器函数外，最后还有一个CancelWaitableTimer函数：

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

这个简单的函数用于取出定时器的句柄并将它撤消，这样，除非接着调用SetWaitableTimer函数以便重新设置定时器，否则定时器决不会进行报时。如果想要改变定时器的报时条件，不必在调用SetWaitableTimer函数之前调用CancelWaitableTimer函数。每次调用SetWaitableTimer函数，都会在设置新的报时条件之前撤消定时器原来的报时条件。

9.4.1 让等待定时器给APC项排队

到现在为止，你已经学会了如何创建定时器和如何设置定时器。你还知道如何通过将定时器的句柄传递给WaitForSingleObject或WaitForMultipleObjects函数，以便等待定时器报时。Microsoft还允许定时器给在定时器得到通知信号时调用SetWaitableTimer函数的线程的异步过程调用（APC）进行排队。

一般来说，当调用SetWaitableTimer函数时，你将同时为pfnCompletionRoutine和pvArgCompletionRoutine参数传递NULL。当SetWaitableTime函数看到这些参数的NULL时，它就知道，当规定的时间到来时，就向定时器发出通知信号。但是，如果到了规定的时间，你愿意让定时器给一个APC排队，那么你必须传递定时器APC例程的地址，而这个例程是你必须实现的。该函数应该类似下面的形式：

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
   DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

   // Do whatever you want here.
}
```

China-pub.com 下载

我已经将该函数命名为TimerAPCRoutine，不过可以根据需要给它赋予任何一个名字。该函数可以在定时器报时的时候由调用SetWaitableTimer函数的同一个线程来调用，但是只有在调用线程处于待命状态下才能调用。换句话说，该线程必须正在 SleepEx,WaitForSngleObjectEx，WaitForMultipleObjectsEx，MsgWaitForMultipleObjectsEx或SingleObject-AndWait等函数的调用中等待。如果该线程不在这些函数中的某个函数中等待，系统将不给定时器 APC例程排队。这可以防止线程的APC队列中塞满定时器APC通知，这会浪费系统中的大量内存。

当定时器报时的时候，如果你的线程处于待命的等待状态中，系统就使你的线程调用回调例程。回调例程的第一个参数的值与你传递给SetWaitableTimer函数的pvArgToCompletionRoutine参数的值是相同的。你可以将某些上下文信息（通常是你定义的某个结构的指针）传递给TimerAPCRoutine。剩余的两个参数dwTimerLowValue和dwTimerHighValue用于指明定时器何时报时。下面的代码使用了该信息，并将它显示给用户：

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
   DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

   FILETIME ftUTC, ftLocal;
   SYSTEMTIME st;
   TCHAR szBuf[256];

   // Put the time in a FILETIME structure.
   ftUTC.dwLowDateTime = dwTimerLowValue;
   ftUTC.dwHighDateTime = dwTimerHighValue;

   // Convert the UTC time to the user's local time.
   FileTimeToLocalFileTime(&ftUTC, &ftLocal);

   // Convert the FILETIME to the SYSTEMTIME structure
   // required by GetDateFormat and GetTimeFormat.
   FileTimeToSystemTime(&ftLocal, &st);

   // Construct a string with the
   // date/time that the timer went off.
   GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE,
      &st, NULL, szBuf, sizeof(szBuf) / sizeof(TCHAR));
   _tcscat(szBuf, __TEXT(" "));
   GetTimeFormat(LOCALE_USER_DEFAULT, 0,
      &st, NULL, _tcschr(szBuf, 0),
      sizeof(szBuf) / sizeof(TCHAR) - _tcslen(szBuf));

   // Show the time to the user.
   MessageBox(NULL, szBuf, "Timer went off at...", MB_OK);
}
```

只有当所有的APC项都已经处理之后，待命的函数才会返回。因此，必须确保定时器再次变为已通知状态之前，TimerAPCRoutine函数完成它的运行，这样，APC项的排队速度就不会比它被处理的速度快。

下面的代码显示了使用定时器和APC项的正确方法：

```
void SomeFunc() {
   // Create a timer. (It doesn't matter whether it's manual-reset
   // or auto-reset.)
   HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);

   // Set timer to go off in 5 seconds.
```

China-pub.com
下载


```
   LARGE_INTEGER li = { 0 };
   SetWaitableTimer(hTimer, &li, 5000, TimerAPCRoutine, NULL, FALSE

   // Wait in an alertable state for the timer to go off.
   SleepEx(INFINITE, TRUE);

   CloseHandle(hTimer);
}
```

最后要说明的是，线程不应该等待定时器的句柄，也不应该以待命的方式等待定时器。请看下面的代码：

```
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(hTimer, ..., TimerAPCRoutine,...);
WaitForSingleObjectEx(hTimer, INFINITE, TRUE);
```

不应该编写上面的代码，因为调用WaitForSingleObjectEx函数实际上是两次等待该定时器，一次是以待命方式等待，一次是等待内核对象句柄。当定时器变为已通知状态时，等待就成功了，线程被唤醒，这将使该线程摆脱待命状态，而APC例程则没有被调用。前面讲过，通常没有理由使用带有等待定时器的APC例程，因为你始终都可以等待定时器变为已通知状态，然后做你想要做的事情。

9.4.2 定时器的松散特性

定时器常常用于通信协议中。例如，如果客户机向服务器发出一个请求，而服务器没有在规定的时间内作出响应，那么客户机就会认为无法使用服务器。目前，客户机通常要同时与许多服务器进行通信。如果你为每个请求创建一个定时器内核对象，那么系统的运行性能就会受到影响。可以设想，对于大多数应用程序来说，可以创建单个定时器对象，并根据需要修改定时器报时的时间。

定时器报时时间的管理方法和定时器时间的重新设定是非常麻烦的，只有很少的应用程序采用这种方法。但是在新的线程共享函数（第 11 章中介绍）中有一个新函数，称为CreateTimerQueueTimer，它能够为你处理所有的操作。如果你发现自己创建和管理了若干个定时器对象，那么应该观察一下这个函数，以减少应用程序的开销。

虽然定时器能够给APC项进行排队是很好的，但是目前编写的大多数应用程序并不使用APC，它们使用I/O完成端口机制。过去，我自己的线程池中（由一个 I/O完成端口负责管理）有一个线程，它按照特定的定时器间隔醒来。但是，等待定时器没有提供这个方法。为了做到这一点，我创建了一个线程，它的唯一工作是设置而后等待一个等待定时器。当定时器变为已通知状态时，线程就调用 PostQueuedCompletionStatus函数，将一个事件强加给线程池中的一个线程。

最后要说明的是，凡是称职的 Windows编程员都会立即将等待定时器与用户定时器（用SetTimer函数进行设置）进行比较。它们之间的最大差别是，用户定时器需要在应用程序中设置许多附加的用户界面结构，这使定时器变得资源更加密集。另外，等待定时器属于内核对象，这意味着它们可以供多个线程共享，并且是安全的。

用户定时器能够生成WM_TIMER消息，这些消息将返回给调用 SetTimer（用于回调定时器）的线程和创建窗口（用于基于窗口的定时器）的线程。因此，当用户定时器报时的时候，只有一个线程得到通知。另一方面，多个线程可以在等待定时器上进行等待，如果定时器是个人工重置的定时器，则可以调度若干个线程。

China-pub.com 下载

如果要执行与用户界面相关的事件，以便对定时器作出响应，那么使用用户定时器来组织代码结构可能更加容易些，因为使用等待定时器时，线程必须既要等待各种消息，又要等待内核对象（如果要改变代码的结构，可以使用 MsgWaitForMultipleObjects函数）。最后，运用等待定时器，当到了规定时间的时候，更有可能得到通知。正如第 27章介绍的那样，WM_TIMER消息始终属于最低优先级的消息，当线程的队列中没有其他消息时，才检索该消息。等待定时器的处理方法与其他内核对象没有什么差别，如果定时器发出报时信息，而你的线程正在等待之中，那么你的线程就会醒来。

9.5 信标内核对象

信标内核对象用于对资源进行计数。它们与所有内核对象一样，包含一个使用数量，但是它们也包含另外两个带符号的32位值，一个是最大资源数量，一个是当前资源数量。最大资源数量用于标识信标能够控制的资源的最大数量，而当前资源数量则用于标识当前可以使用的资源的数量。

为了正确地说明这个问题，让我们来看一看应用程序是如何使用信标的。比如说，我正在开发一个服务器进程，在这个进程中，我已经分配了一个能够用来存放客户机请求的缓冲区。我对缓冲区的大小进行了硬编码，这样它每次最多能够存放 5个客户机请求。如果5个请求尚未处理完毕时，一个新客户机试图与服务器进行联系，那么这个新客户机的请求就会被拒绝，并出现一个错误，指明服务器现在很忙，客户机应该过些时候重新进行联系。当我的服务器进程初始化时，它创建一个线程池，里面包含 5个线程，每个线程都准备在客户机请求到来时对它进行处理。

开始时，没有客户机提出任何请求，因此我的服务器不允许线程池中的任何线程成为可调度线程。但是，如果3个客户机请求同时到来，那么线程池中应该有 3个线程处于可调度状态。使用信标，就能够很好地处理对资源的监控和对线程的调度，最大资源数量设置为 5，因为这是我进行硬编码的缓冲区的大小。当前资源数量最初设置为 0，因为没有客户机提出任何请求。当客户机的请求被接受时，当前资源数量就递增，当客户机的请求被提交给服务器的线程池时，当前资源数量就递减。

信标的使用规则如下：

- 如果当前资源的数量大于0，则发出信标信号。
- 如果当前资源数量是0，则不发出信标信号。
- 系统决不允许当前资源的数量为负值。
- 当前资源数量决不能大于最大资源数量。

当使用信标时，不要将信标对象的使用数量与它的当前资源数量混为一谈。

下面的函数用于创建信标内核对象：

```
HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTE psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);
```

psa和pszName两个参数在第3章中作过介绍。当然，通过调用 OpenSemaphore函数，另一个进程可以获得它自己的进程与现有信标相关的句柄：

```
HANDLE OpenSemaphore(
   DWORD fdwAccess,
```

China-pub.com
下载


```
BOOL bInheritHandle,
PCTSTR pszName);
```

lMaximumCount参数用于告诉系统，应用程序处理的最大资源数量是多少。由于这是个带符号的32位值，因此最多可以拥有2 147 483 647个资源。lInitialCount参数用于指明开始时（当前）这些资源中有多少可供使用。当我的服务器进程初始化时，没有任何客户机请求，因此我调用下面这个CreateSemaphore函数：

```
HANDLE hsem = CreateSemaphore(NULL, 0, 5, NULL);
```

该函数创建了一个信标，其最大资源数量为5，但是开始时可以使用的资源为0（由于偶然的原因，该内核对象的使用数量是1，因为我刚刚创建了这个内核对象，请不要与计数器搞混）。由于当前资源数量被初始化为0，因此不发出信标信号。等待信标的所有线程均进入等待状态。

通过调用等待函数，传递负责保护资源的信标的句柄，线程就能够获得对该资源的访问权。从内部来说，该等待函数要检查信标的当前资源数量，如果它的值大于0（信标已经发出信号），那么计数器递减1，调用线程保持可调度状态。信标的出色之处在于它们能够以原子操作方式来执行测试和设置操作，这就是说，当向信标申请一个资源时，操作系统就要检查是否有这个资源可供使用，同时将可用资源的数量递减，而不让另一个线程加以干扰。只有当资源数量递减后，系统才允许另一个线程申请对资源的访问权。

如果该等待函数确定信标的当前资源数量是0（信标没有发出通知信号），那么系统就调用函数进入等待状态。当另一个线程将对信标的当前资源数量进行递增时，系统会记住该等待线程（或多个线程），并允许它变为可调度状态（相应地递减它的当前资源数量）。

通过调用ReleaseSemaphore函数，线程就能够对信标的当前资源数量进行递增：

```
BOOL ReleaseSemaphore(
   HANDLE hsem,
   LONG lReleaseCount,
   PLONG plPreviousCount);
```

该函数只是将lReleaseCount中的值添加给信标的当前资源数量。通常情况下，为lReleaseCount参数传递1，但是，不一定非要传递这个值。我常常传递2或更大的值。该函数也能够在它的*plPreviousCount中返回当前资源数量的原始值。实际上几乎没有应用程序关心这个值，因此可以传递NULL，将它忽略。

有时，有必要知道信标的当前资源数量而不修改这个数量，但是没有一个函数可以用来查询信标的当前资源数量的值。起先我认为调用ReleaseSemaphore并为lReleaseCount参数传递0，也许会在*plPreviousCount中返回资源的实际数量。但是这样做是不行的，ReleaseSemaphore用0填入这个长变量。接着，我试图传递一个非常大的数字，作为第二个参数，希望它不会影响当前资源数量，因为它将取代最大值。同样，ReleaseSemaphore用0填入*plPrevious。可惜，如果不对它进行修改，就没有办法得到信标的当前资源数量。

9.6 互斥对象内核对象

互斥对象（mutex）内核对象能够确保线程拥有对单个资源的互斥访问权。实际上互斥对象是因此而得名的。互斥对象包含一个使用数量，一个线程ID和一个递归计数器。互斥对象的行为特性与关键代码段相同，但是互斥对象属于内核对象，而关键代码段则属于用户方式对象。这意味着互斥对象的运行速度比关键代码段要慢。但是这也意味着不同进程中的多个线程能够访问单个互斥对象，并且这意味着线程在等待访问资源时可以设定一个超时值。

ID用于标识系统中的哪个线程当前拥有互斥对象，递归计数器用于指明该线程拥有互斥对

China-pub.com 下载

象的次数。互斥对象有许多用途，属于最常用的内核对象之一。通常来说，它们用于保护由多个线程访问的内存块。如果多个线程要同时访问内存块，内存块中的数据就可能遭到破坏。互斥对象能够保证访问内存块的任何线程拥有对该内存块的独占访问权，这样就能够保证数据的完整性。

互斥对象的使用规则如下：

- 如果线程ID是0（这是个无效ID），互斥对象不被任何线程所拥有，并且发出该互斥对象的通知信号。
- 如果ID是个非0数字，那么一个线程就拥有互斥对象，并且不发出该互斥对象的通知信号。
- 与所有其他内核对象不同， 互斥对象在操作系统中拥有特殊的代码，允许它们违反正常的规则（后面将要介绍这个异常情况）。

若要使用互斥对象，必须有一个进程首先调用CreateMutex，以便创建互斥对象：

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL fInitialOwner,
   PCTSTR pszName);
```

psa和pszName参数在第3章中做过介绍。当然，通过调用 OpenMutex，另一个进程可以获得它自己进程与现有互斥对象相关的句柄：

```
HANDLE OpenMutex(
   DWORD fdwAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

fInitialOwner参数用于控制互斥对象的初始状态。如果传递 FALSE（这是通常情况下传递的值），那么互斥对象的ID和递归计数器均被设置为0。这意味着该互斥对象没有被任何线程所拥有，因此要发出它的通知信号。

如果为fInitialOwner参数传递TRUE，那么该对象的线程ID被设置为调用线程的ID，递归计数器被设置为1。由于ID是个非0数字，因此该互斥对象开始时不发出通知信号。

通过调用一个等待函数，并传递负责保护资源的互斥对象的句柄，线程就能够获得对共享资源的访问权。在内部，等待函数要检查线程的 ID，以了解它是否是0（互斥对象发出通知信号）。如果线程ID是0，那么该线程ID被设置为调用线程的ID，递归计数器被设置为 1，同时，调用线程保持可调度状态。

如果等待函数发现ID不是0（不发出互斥对象的通知信号），那么调用线程便进入等待状态。系统将记住这个情况，并且在互斥对象的 ID重新设置为0时，将线程ID设置为等待线程的ID，将递归计数器设置为 1，并且允许等待线程再次成为可调度线程。与所有情况一样，对互斥内核对象进行的检查和修改都是以原子操作方式进行的。

对于互斥对象来说，正常的内核对象的已通知和未通知规则存在一个特殊的异常情况。比如说，一个线程试图等待一个未通知的互斥对象。在这种情况下，该线程通常被置于等待状态。然而，系统要查看试图获取互斥对象的线程的ID是否与互斥对象中记录的线程ID相同。如果两个线程ID相同，即使互斥对象处于未通知状态，系统也允许该线程保持可调度状态。我们不认为该“异常”行为特性适用于系统中的任何地方的其他内核对象。每当线程成功地等待互斥对象时，该对象的递归计数器就递增。若要使递归计数器的值大于 1，唯一的方法是线程多次等待相同的互斥对象，以便利用这个异常规则。

China-pub.com
下载

一旦线程成功地等待到一个互斥对象，该线程就知道它已经拥有对受保护资源的独占访问权。试图访问该资源的任何其他线程（通过等待相同的互斥对象）均被置于等待状态中。当目前拥有对资源的访问权的线程不再需要它的访问权时，它必须调用 ReleaseMutex函数来释放该互斥对象：

```
BOOL ReleaseMutex(HANDLE hMutex);
```

该函数将对象的递归计数器递减 1。如果线程多次成功地等待一个互斥对象，在互斥对象的递归计数器变成 0之前，该线程必须以同样的次数调用 ReleaseMutex函数。当递归计数器到达0时，该线程ID也被置为0，同时该对象变为已通知状态。

当该对象变为已通知状态时，系统要查看是否有任何线程正在等待互斥对象。如果有，系统将“按公平原则”选定等待线程中的一个，为它赋予互斥对象的所有权。当然，这意味着线程ID被设置为选定的线程的ID，并且递归计数器被置为 1。如果没有其他线程正在等待互斥对象，那么该互斥对象将保持已通知状态，这样，等待互斥对象的下一个线程就立即可以得到互斥对象。

9.6.1 释放问题

互斥对象不同于所有其他内核对象，因为互斥对象有一个“线程所有权”的概念。本章介绍的其他内核对象中，没有一种对象能够记住哪个线程成功地等待到该对象，只有互斥对象能够对此保持跟踪。互斥对象的线程所有权概念是互斥对象为什么会拥有特殊异常规则的原因，这个异常规则使得线程能够获取该互斥对象，尽管它没有发出通知。

这个异常规则不仅适用于试图获取互斥对象的线程，而且适用于试图释放互斥对象的线程。当一个线程调用ReleaseMutex函数时，该函数要查看调用线程的ID是否与互斥对象中的线程ID相匹配。如果两个ID相匹配，递归计数器就会像前面介绍的那样递减。如果两个线程的 ID不匹配，那么ReleaseMutex函数将不进行任何操作，而是将 FALSE（表示失败）返回给调用者。此时调用GetLastError，将返回ERROR_NOT_OWNER（试图释放不是调用者拥有的互斥对象）。

因此，如果在释放互斥对象之前，拥有互斥对象的线程终止运行（使用 ExitThread、TerminateThread、ExitProcess或TerminateProcess函数），那么互斥对象和正在等待互斥对象的其他线程将会发生什么情况呢？答案是，系统将把该互斥对象视为已经被放弃——拥有互斥对象的线程决不会释放它，因为该线程已经终止运行。

由于系统保持对所有互斥对象和线程内核对象的跟踪，因此它能准确的知道互斥对象何时被放弃。当一个互斥对象被放弃时，系统将自动把互斥对象的 ID复置为0，并将它的递归计数器复置为 0。然后，系统要查看目前是否有任何线程正在等待该互斥对象。如果有，系统将“公平地”选定一个等待线程，将 ID设置为选定的线程的ID，并将递归计数器设置为 1，同时，选定的线程变为可调度线程。

这与前面的情况相同，差别在于等待函数并不将通常的 WAIT_OBJECT_0值返回给线程。相反，等待函数返回的是特殊的 WAIT_ABANDONED值。这个特殊的返回值（它只适用于互斥对象）用于指明线程正在等待的互斥对象是由另一个线程拥有的，而这另一个线程已经在它完成对共享资源的使用前终止运行。这不是可以进入的最佳情况。新调度的线程不知道目前资源处于何种状态，也许该资源已经完全被破坏了。在这种情况下必须自己决定应用程序应该怎么办。

在实际运行环境中，大多数应用程序从不明确检查 WAIT_ABANDONED返回值，因为线程很少是刚刚终止运行（上面介绍的情况提供了另一个例子，说明为什么决不应该调用

China-pub.com 下载

TerminateThread函数）。

9.6.2 互斥对象与关键代码段的比较

就等待线程的调度而言，互斥对象与关键代码段之间有着相同的特性。但是它们在其他属性方面却各不相同。表9-1对它们进行了各方面的比较。

表9-1 互斥对象与关键代码段的比较

特 性	互斥对象	关键代码段
运行速度	慢	快
是否能够跨进程边界来使用	是	否
声明	HANDLE hmtx；	CRITICAL_SECTION cs；
初始化	hmtx=CreateMutex（NULL，FALSE，NULL）；	InitializeCriticalSection(&es)；
清除	CloseHandle（hmtx）；	DeleteCriticalSection（&cs）；
无限等待	WaitForSingleObject（hmtx,INFINITE）；	EnterCriticalSection（&cs）；
0等待	WaitForSingleObject（hmtx,0）；	TryEnterCriticalSection（&cs）；
任意等待	WaitForSingleObject（hmtx,dwMilliseconds）；	不能
释放	ReleaseMutex（hmtx）；	LeaveCriticalSection（&cs）；
是否能够等待其他内核对象	是（使用WaitForMultipleObjects或类似的函数）	否

9.6.3 Queue示例应用程序

后面的清单9-2列出的Queue（“09 Queue.exe）应用程序使用一个互斥对象和一个信标来控制数据元素的队列。该应用程序的源代码和资源文件位于本书所附光盘中的 09-Queue目录下。当运行Queue应用程序时，就会出现图9-4对话框。

当Queue初始化时，它创建4个客户机线程和两个服务器线程。每个客户机线程均会睡眠一定的时间周期，然后将一个请求元素附加给队列。当每个元素排队时， Client Threads（客户机线程）列表框就被更新。列表框的每一项表示哪个客户机线程附加了这个项，以及它是个什么项。例如，列表框中的第一项表示客户机线程 0附加了它的第一个请求。然后客户机线程 1至3附加它们的第一个请求，接着客户机线程0又附加它的第二个请求，如此类推。

服务器线程在队列中出现第一个元素之前一直无事可做。当第一个元素出现时，一个服务器线程醒来，对该请求进程处理。 Server Threads（服务器线程）列表框显示了服务器线程的状态。它的第一项显示服务器线程 0正在处理来自客户机线程 0的一个请求。正在处理的请求是客户机线程的第一个请求。第二项显示服务器线程 1正在处理客户机线程 1的第一个请求。

在这个例子中，服务器线程无法以足够快的速度来处理客户程序的请求，队列中的请求达到了最大的容量。我对队列数据结构进行了初始化，使它一次能够存放的元素不能超过 10个，这将导致队列很快被填满。另外，客户机线程有 4个，而服务器线程只有两个。我们看到，当客户机线程3试图将它的第5个请求附加给队列时，队列已经填满了。

China-pub.com
下载

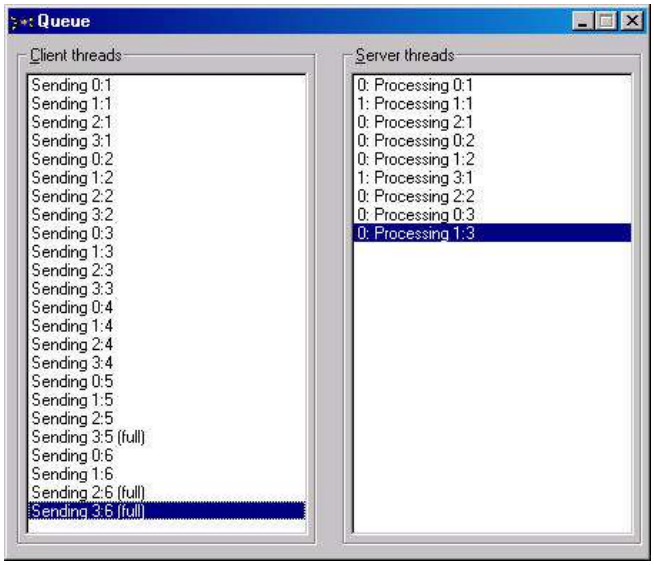


图9-4 Queue 对话框

好了，这就是你看到的情况，更有意思的是它的运行情况。这个队列是由一个 C++类CQueue进行管理和控制的：

```
class CQueue {
public:
   struct ELEMENT {
      int m_nThreadNum, m_nRequestNum;
      // Other element data should go here.
   };
   typedef ELEMENT* PELEMENT;

private:
   PELEMENT m_pElements;         // Array of elements to be processed
   int      m_nMaxElements;      // # of elements in the array
   HANDLE   m_h[2];              // Mutex & semaphore handles
   HANDLE   &m_hmtxQ;            // Reference to m_h[0]
   HANDLE   &m_hsemNumElements; // Reference to m_h[1]

public:
   CQueue(int nMaxElements);
   ~CQueue();

   BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
   BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};
```

这个类中的公有ELEMENT结构用于定义队列数据元素是个什么样子。它的实际内容并不重要。对于这个示例应用程序，客户程序将它们的线程号和它们的请求号放入这个元素，这样，当服务器处理检索出来的元素时，就能在服务器的列表框中显示这些信息。实际运行的应用程序通常不需要这些信息。

至于私有成员，我们有m_pElements，它指向一个固定大小的ELEMENT结构的数组。这是需要保护使之不受多个客户机/服务器线程影响的数据。M_nMaxElements成员用于表示创建

China-pub.com 下载

Cqueue对象时该数组被初始化为多大的规模。下一个成员 m_h是由两个内核对象句柄组成的数组。为了正确地保护队列的数据元素，需要两个内核对象，一个是互斥对象，另一个是信标对象。在CQueue构造函数中，这两个对象被创建，它们的句柄放入该数组。

下面很快就会看到，该代码有时调用 WaitForMultipleObjects函数，传递句柄数组的地址。也会看到，有时该代码只需要引用这些内核对象句柄中的一个句柄。为了使代码更容易阅读和维护，我还声明了两个句柄参考成员，即 m_hmtxQ和m_hsemNumElements。当CQueue构造函数运行时，它就将这些句柄参考成员分别初始化为 m_h[0]和m_h[1]。

现在应该能够毫无困难地理解 CQueue的构造函数与析构函数的方法了，因此让我们将注意力转向Append方法。这个方法试图将一个 ELEMENT附加给队列。但是，线程首先必须确保它拥有对该队列的独占访问权。 Append方法通过调用WaitForSingleObject函数，传递m_hmtxQ互斥对象的句柄，实现其独占访问权。如果返回 WAIT_OBJECT_0,那么该线程就拥有对该队列的独占访问权。

接着，Append方法必须调用ReleaseSemaphore函数，传递释放数量 1，以便使队列中的元素数量递增。如果ReleaseSemaphore函数调用成功，队列中的元素没有放满，那么新元素就可以附加给队列。幸好ReleaseSemaphore函数返回了lPreviousCount变量中的队列元素的前一个数量。这确切地告诉你新元素应该放入哪个数组索引中。当该元素被拷贝到队列的数组中后，该函数就返回。一旦该元素完全附加给队列， Append便调用Release互斥对象，这样其他线程就能够访问该队列。Append函数的剩余部分与故障情况和错误处理有关。

现在让我们来看一看服务器线程如何调用 Remove方法，以便从队列中取出元素的。首先，线程必须确保它拥有对队列的独占访问权，同时队列中至少必须拥有一个元素。当然，如果队列中没有任何元素，那么服务器线程就没有理由被唤醒。因此， Remove方法首先要调用WairForMultipleObjects，并且同时传递互斥对象和信标的句柄。只有当这两个对象都得到通知时，服务器线程才被唤醒。

如果返回WAIT_OBJECT_0，该线程将拥有对队列的独占访问权，并且队列中至少必须有一个元素。这时，代码就取出数组中索引号为 0的元素，然后将数组中的剩余元素下移一位。这不是使用队列的最有效的方法，因为用这种方法进行内存的拷贝要占用大量的资源，但是我们在这里是想展示线程同步的情况，所以就使用了这种方法。最后，要调用 ReleaseMutex函数，这样其他线程就能安全地访问该队列。

注意，信标对象能够随时跟踪某个时间队列中存在多少个元素。你能够看到这个数字在递增。当一个新元素被附加给队列时， Append方法就调用ReleaseSemaphore。但是，当一个元素从队列中删除时，你无法立即看到这个数字递减。递减是由Remove方法调用WaitForMultipleObjects函数来进行的。记住，成功地等待信标的副作用是它的数量递减 1。这对我们来说是很方便的。

现在已经懂得CQueue类是如何运行的，源代码的其余部分很容易理解。

清单9-2 CQueue示例应用程序

Queue.cpp

```
/******************************************************************************
Module:  Queue.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
```

China-pub.com
下载


```
#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>       // For _beginthreadex
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

class CQueue {
public:
   struct ELEMENT {
      int m_nThreadNum, m_nRequestNum;
      // Other element data should go here
   };
   typedef ELEMENT* PELEMENT;

private:
   PELEMENT m_pElements;         // Array of elements to be processed
   int      m_nMaxElements;      // Maximum # of elements in the array
   HANDLE   m_h[2];              // Mutex & semaphore handles
   HANDLE   &m_hmtxQ;            // Reference to m_h[0]
   HANDLE   &m_hsemNumElements; // Reference to m_h[1]

public:
   CQueue(int nMaxElements);
   ~CQueue();

   BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
   BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};


//////////////////////////////////////////////////////////////////////////////


CQueue::CQueue(int nMaxElements)
   : m_hmtxQ(m_h[0]), m_hsemNumElements(m_h[1]) {

   m_pElements = (PELEMENT)
      HeapAlloc(GetProcessHeap(), 0, sizeof(ELEMENT) * nMaxElements);
   m_nMaxElements = nMaxElements;
   m_hmtxQ = CreateMutex(NULL, FALSE, NULL);
   m_hsemNumElements = CreateSemaphore(NULL, 0, nMaxElements, NULL);
}


//////////////////////////////////////////////////////////////////////////////


CQueue::~CQueue() {

   CloseHandle(m_hsemNumElements);
   CloseHandle(m_hmtxQ);
   HeapFree(GetProcessHeap(), 0, m_pElements);
}


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
BOOL CQueue::Append(PELEMENT pElement, DWORD dwTimeout) {

   BOOL fOk = FALSE;
   DWORD dw = WaitForSingleObject(m_hmtxQ, dwTimeout);

   if (dw == WAIT_OBJECT_0) {
      // This thread has exclusive access to the queue

      // Increment the number of elements in the queue
      LONG lPrevCount;
      fOk = ReleaseSemaphore(m_hsemNumElements, 1, &lPrevCount);
      if (fOk) {
         // The queue is not full; append the new element
         m_pElements[lPrevCount] = *pElement;
      } else {

         // The queue is full; set the error code and return failure
         SetLastError(ERROR_DATABASE_FULL);
      }

      // Allow other threads to access the queue
      ReleaseMutex(m_hmtxQ);

   } else {
      // Timeout, set error code and return failure
      SetLastError(ERROR_TIMEOUT);
   }

   return(fOk);   // Call GetLastError for more info
}

//////////////////////////////////////////////////////////////////////////////

BOOL CQueue::Remove(PELEMENT pElement, DWORD dwTimeout) {

   // Wait for exclusive access to queue and for queue to have element.
   BOOL fOk = (WaitForMultipleObjects(chDIMOF(m_h), m_h, TRUE, dwTimeout)
      == WAIT_OBJECT_0);

   if (fOk) {
      // The queue has an element; pull it from the queue
      *pElement = m_pElements[0];

      // Shift the remaining elements down
      MoveMemory(&m_pElements[0], &m_pElements[1],
      sizeof(ELEMENT) * (m_nMaxElements - 1));

   // Allow other threads to access the queue
   ReleaseMutex(m_hmtxQ);

} else {
   // Timeout, set error code and return failure
   SetLastError(ERROR_TIMEOUT);
}

return(fOk);   // Call GetLastError for more info
```

China-pub.com
下载


```
   }

   //////////////////////////////////////////////////////////////////////////////

   CQueue g_q(10);                        // The shared queue
   volatile BOOL g_fShutdown = FALSE;     // Signals client/server threads to die
   HWND g_hwnd;                           // How client/server threads give status

   // Handles to all client/server threads & number of client/server threads
   HANDLE g_hThreads[MAXIMUM_WAIT_OBJECTS];
   int    g_nNumThreads = 0;

   //////////////////////////////////////////////////////////////////////////////

   DWORD WINAPI ClientThread(PVOID pvParam) {

      int nThreadNum = PtrToUlong(pvParam);
      HWND hwndLB = GetDlgItem(g_hwnd, IDC_CLIENTS);

      for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

         TCHAR sz[1024];
         CQueue::ELEMENT e = { nThreadNum, nRequestNum };

         // Try to put an element on the queue
         if (g_q.Append(&e, 200)) {

            // Indicate which thread sent it and which request
            wsprintf(sz, TEXT("Sending %d:%d"), nThreadNum, nRequestNum);
         } else {
            // Couldn't put an element on the queue
            wsprintf(sz, TEXT("Sending %d:%d (%s)"), nThreadNum, nRequestNum,
               (GetLastError() == ERROR_TIMEOUT)
                  ? TEXT("timeout") : TEXT("full"));
         }

         // Show result of appending element
         ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
         Sleep(2500);   // Wait before appending another element
      }

      return(0);
   }

   //////////////////////////////////////////////////////////////////////////////

   DWORD WINAPI ServerThread(PVOID pvParam) {

      int nThreadNum = PtrToUlong(pvParam);
      HWND hwndLB = GetDlgItem(g_hwnd, IDC_SERVERS);
```

China-pub.com 下载

```
while (!g_fShutdown) {

   TCHAR sz[1024];
   CQueue::ELEMENT e;

   // Try to get an element from the queue
   if (g_q.Remove(&e, 5000)) {

      // Indicate which thread is processing it, which thread
      // sent it, and which request we're processing
      wsprintf(sz, TEXT("%d: Processing %d:%d"),
         nThreadNum, e.m_nThreadNum, e.m_nRequestNum);

      // The server takes some time to process the request
      Sleep(2000 * e.m_nThreadNum);

   } else {
      // Couldn't get an element from the queue
      wsprintf(sz, TEXT("%d: (timeout)"), nThreadNum);
   }

   // Show result of processing element
   ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

   }

   return(0);
}

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_QUEUE);

   g_hwnd = hwnd; // Used by client/server threads to show status

   DWORD dwThreadID;

   // Create the client threads
   for (int x = 0; x < 4; x++)
      g_hThreads[g_nNumThreads++] =
         chBEGINTHREADEX(NULL, 0, ClientThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

   // Create the server threads
   for (x = 0; x < 2; x++)
      g_hThreads[g_nNumThreads++] =
         chBEGINTHREADEX(NULL, 0, ServerThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_QUEUE), NULL, Dlg_Proc);
   InterlockedExchangePointer((PVOID*) &g_fShutdown, (PVOID) TRUE);

   // Wait for all the threads to terminate & then cleanup
   WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);
   while (g_nNumThreads--)
      CloseHandle(g_hThreads[g_nNumThreads]);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Queue.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_QUEUE DIALOG DISCARDABLE  38, 36, 298, 225
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Queue"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "&Client threads",IDC_STATIC,4,4,140,216
    LISTBOX         IDC_CLIENTS,8,16,132,200,NOT LBS_NOTIFY |
                    LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
    GROUPBOX        "&Server threads",IDC_STATIC,156,4,140,216
    LISTBOX         IDC_SERVERS,160,16,132,200,NOT LBS_NOTIFY |
                    LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
END

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_QUEUE               ICON    DISCARDABLE     "Queue.Ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

China-pub.com
下载


```
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_QUEUE, DIALOG
    BEGIN
        RIGHTMARGIN, 244
        BOTTOMMARGIN, 130
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

9.7 线程同步对象速查表

表9-2所示的速查表综合列出了各种内核对象与线程同步之间的相互关系。

表9-2 内核对象与线程同步之间的相互关系

对 象	何时处于未通知状态	何时处于已通知状态	成功等待的副作用
进程	当进程仍然活动时	当进程终止运行时（ExitProcess，TerminateProcess）	无
线程	当线程仍然活动时	当线程终止运行时（ExitThread，TerminateThread）	无
作业	当作业的时间尚未结束时	当作业的时间已经结束时	无
文件	当I/O请求正在处理时	当I/O请求处理完毕时	无
控制台输入	不存在任何输入	当存在输入时	无
文件修改通知	没有任何文件被修改	当文件系统发现修改时	重置通知
自动重置事件	ResetEvent,Pulse-Event或等待成功	当调用SetEvent/PulseEvent时	重置事件
人工重置事件	ResetEvent或PulseEvent	当调用SetEvent/PulseEvent时	无

China-pub.com 下载

(续)

对 象	何时处于未通知状态	何时处于已通知状态	成功等待的副作用
自动重置等待定时器	CancelWaitableTimer或等待成功	当时间到时(SetWaitableTimer)	重置定时器
人工重置等待定时器	CancelWaitableTimer	当时间到时(SetWaitableTimer)	无
信标	等待成功	当数量>0时(ReleaseSemaphore)	数量递减1
互斥对象	等待成功	当未被线程拥有时(Release互斥对象)	将所有权赋予线程
关键代码段(用户方式)	等待成功((Try) EnterCriticalSection)	当未被线程拥有时(LeaveCriticalSection)	将所有权赋予线程

互锁(用户方式)函数决不会导致线程变为非调度状态,它们会改变一个值并立即返回。

9.8 其他的线程同步函数

WaitForSingleObject和WaitForMultipleObjects是进行线程同步时使用得最多的函数。但是,Windows还提供了另外几个稍有不同的函数。如果理解了WaitForSingleObject和WaitForMultipleObjects函数,那么要理解其他函数如何运行,就不会遇到什么困难。本节简单地介绍一些这样的函数。

9.8.1 异步设备I/O

异步设备I/O使得线程能够启动一个读操作或写操作,但是不必等待读操作或写操作完成。例如,如果线程需要将一个大文件装入内存,那么该线程可以告诉系统将文件装入内存。然后,当系统加载该文件时,该线程可以忙于执行其他任务,如创建窗口、对内部数据结构进行初始化等等。当初始化操作完成时,该线程可以终止自己的运行,等待系统通知它文件已经读取。

设备对象是可以同步的内核对象,这意味着可以调用WaitForSingleObject函数,传递文件、套接字和通信端口的句柄。当系统执行异步I/O时,设备对象处于未通知状态。一旦操作完成,系统就将对象的状态改为已通知状态,这样,该线程就知道操作已经完成。此时,该线程就可以继续运行。

9.8.2 WaitForInputIdle

线程也可以调用WaitForInputIdle来终止自己的运行:

```
DWORD WaitForInputIdle(
  HANDLE hProcess,
  DWORD dwMilliseconds);
```

该函数将一直处于等待状态,直到hProcess标识的进程在创建应用程序的第一个窗口的线程中已经没有尚未处理的输入为止。这个函数可以用于父进程。父进程产生子进程,以便执行某些操作。当父进程的线程调用CreateProcess时,该父进程的线程将在子进程初始化时继续运行。父进程的线程可能需要获得子进程创建的窗口的句柄。如果父进程的线程想要知道子进程何时完成初始化,唯一的办法是等待,直到子进程不再处理任何输入为止。因此,当调用CreateProcess后,父进程的线程就调用WaitForInputIdle。

当需要将击键输入纳入应用程序时,也可以调用WaitForInputIdle。比如说,可以将下面的

China-pub.com
下载

消息显示在应用程序的主窗口：

WM_KEYDOWN	使用虚拟键VK_MENU
WM_KEYDOWN	使用虚拟键VK_F
WM_KEYUP	使用虚拟键VK_E
WM_KEYUP	使用虚拟键VK_MENU
WM_KEYDOWN	使用虚拟键VK_O
WM_KEYUP	使用虚拟键VK_O

这个序列将Alt+F,O发送给应用程序，对于大多数使用英语的应用程序来说，它从应用程序的文件菜单中选择 Open命令。该命令打开一个对话框，但是，在对话框出现以前，Windows必须加载来自文件的对话框摸板，遍历摸板中的所有控件，并为每个摸板调用 CreateWindow。这可能需要花费一定的时间。因此，显示 WM_KEY*消息的应用程序可以调用WaitForInputIdle，WaitForlnputIdle将导致应用程序处于等待状态，直到对话框创建完成并准备接受用户的输入。这时，该应用程序可以将其他的击键输入纳入对话框及其控件，使它能够继续执行它需要的操作。

编写16位Windows应用程序的编程人员常常要面对这个问题。应用程序想要将消息显示在窗口中，但是它并不确切知道窗口何时创建完成、作好接受消息的准备。 WaitForInputIdle函数解决了这个问题。

9.8.3 MsgWaitForMultipleObjects(Ex)

线程可以调用MsgWaitForMultipleObjects或MsgWaitForMultipleObjectsEx函数，让线程等待它自己的消息：

```
DWORD MsgWaitForMultipleObjects(
   DWORD dwCount,
   PHANDLE phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds,
   DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD dwCount,
   PHANDLE phObjects,
   DWORD dwMilliseconds,
   DWORD dwWakeMask,
   DWORD dwFlags);
```

这些函数与WaitForMultipleObjects函数十分相似。差别在于它们允许线程在内核对象变成已通知状态或窗口消息需要调度到调用线程创建的窗口中时被调度。

创建窗口和执行与用户界面相关的任务的线程，应该调用 MsgWaitForMultipleObjectsEx函数，而不应该调用WaitForMultipleObjects函数，因为后面这个函数将使线程的用户界面无法对用户作出响应。该函数将在第27章中详细介绍。

9.8.4 WaitForDebugEvent

Windows将非常出色的调试支持特性内置于操作系统之中。当调试程序启动运行时，它将自己附加给一个被调试程序。该调试程序只需闲置着，等待操作系统将与被调试程序相关的调试事件通知它。调试程序通过调用WaitForDebugEvent函数来等待这些事件的发生：

China-pub.com 下载

```
BOOL WaitForDebugEvent(
   PDEBUG_EVENT pde,
   DWORD dwMilliseconds);
```

当调试程序调用该函数时，调试程序的线程终止运行，系统将调试事件已经发生的情况通知调试程序，方法是允许调用的WaitForDebugEvent函数返回。pde参数指向的结构在唤醒调试程序的线程之前由系统填入信息。该结构包含了关于刚刚发生的调试事件的信息。

9.8.5 SingleObjectAndWait

SingleObjectAndWait函数用于在单个原子方式的操作中发出关于内核对象的通知并等待另一个内核对象：

```
DWORD SignalObjectAndWait(
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD dwMilliseconds,
   BOOL fAlertable);
```

当调用该函数时，hObjectToSignal参数必须标识一个互斥对象、信标对象或事件。任何其他类型的对象将导致该函数返回WAIT_FAILED，并使GetLastError函数返回ERROR_INVALID_HANDLE。在内部，该函数将观察对象的类型，并分别运行ReleaseMutex、ReleaseSemaphore(其数量为1)或ResetEvent中的相应参数。

hObjectToWaitOn参数用于标识下列任何一个内核对象：互斥对象、信标、事件、定时器、进程、线程、作业、控制台输入和修改通知。与平常一样， dwMilliseconds参数指明该函数为了等待该对象变为已通知状态，应该等待多长时间，而fAlertable标志则指明线程等待时该线程是否应该能够处理任何已经排队的异步过程调用。

该函数返回下列几个值中的一个：WAIT_OBJECT_0、WAIT_TIMEOUT、WAIT_FAILED、WAIT_ABANDONED（本章前面已经介绍）或WAIT_IO_COMPLETION。

该函数是对Windows的令人欢迎的一个补充，原因有二。首先，因为常常需要通知一个对象，等待另一个对象，用一个函数来执行两个操作可以节省许多处理时间。每次调用一个函数，使线程从用户方式代码变成内核方式代码，大约需要运行1000个CPU周期。例如，运行下面的代码至少需要2000个CPU周期：

```
ReleaseMutex(hMutex);
WaitForSingleObject(hEvent, INFINITE);
```

在高性能服务器应用程序中，SignalObjectAndWait函数能够节省大量的处理时间。

第二，如果没有SignalObjectAndWait函数，一个线程将无法知道另一个线程何时处于等待状态。对于PluseEvent之类的函数来说，知道这个情况是很有用的。本章前面讲过，PulseEvent函数能够通知一个事件，并且立即对它进行重置。如果目前没有任何线程等待该事件，那么就没有事件会抓住这个情况。曾经有人编写过类似下面的代码：

```
// Perform some work.
.
.
.
SetEvent(hEventWorkerThreadDone);
WaitForSingleObject(hEventMoreWorkToBeDone, INFINITE);
// Do more work.
.
.
.
```

China-pub.com
下载

一个工作线程负责运行一些代码，然后调用 SetEvent，以指明这项工作已经完成。另一个线程负责执行下面的代码：

```
WaitForSingleObject(hEventWorkerThreadDone);
PulseEvent(hEventMoreWorkToBeDone);
```

这个工作线程的代码段设计得很差，因为它无法可靠地运行。当工作线程调用 SetEvent之后，另一个线程可能立即醒来，并调用 PulseEvent。该工作线程不得不停止运行，没有机会从它对 SetEvent的调用中返回，更不要说调用 WaitForSingleObject函数了。结果， hEvent MoreWorkToBeDone事件的通知就完全被工作线程错过了。

如果像下面所示的那样重新编写工作线程的代码，以便调用 SingleObjectAndWait函数，那么该代码就能够可靠地运行，因为通知和等待都能够以原子操作方式来进行：

```
// Perform some work.
.
.
.
SignalObjectAndWait(hEventWorkerThreadDone,
   hEventMoreWorkToBeDone, INFINITE, FALSE);
// Do more work.
.
.
.
```

当非工作线程醒来时，它能够百分之百地确定工作线程正在等待 hEventMoreWorkToBe Done事件，因此能够确保看到产生该事件。

Windows 98 Windows 98 没有这个函数的可以使用的实现代码。

China-pub.com
下载

第10章　线程同步工具包

多年来，我对线程的同步问题进行了许多开发工作，编写了一些C++类和组件。本章将介绍这些内容。希望这些代码有用，能够使你节省许多编程时间。

本章首先要介绍如何实现关键代码段和将各种特性添加给它的方法。尤其是，要学习如何在多个进程中使用关键代码段。然后要学习如何将数据类型包装在C++类中，使对象成为对线程安全的对象。使用这些类，将展示一种其行为特性与信标相反的对象。

接着要介绍如何解决一个常见的编程问题，即当多个线程读取一种资源但是只有一个线程写入资源时如何进行编程。Windows没有预先内置能够容易地实现这种类型的同步的特性，因此我编写了一个C++类以便实现这个特性。

最后要介绍如何实现WaitForMultipleExpressions函数，该函数可以用来创建复杂的表达式，以便指明应该何时唤醒线程（它的作用很像WaitForMultipleObjects函数，该函数可以用来等待任何单个对象变成已通知状态，或者使所有对象处于已通知状态）。

10.1　实现关键代码段：Optex

关键代码段始终对我有着巨大的吸引力。但是，如果它们只是用户方式对象，为什么不能自己来实现它们呢？为什么需要操作系统的支持特性才能使关键代码段运行呢？另外，如果编写自己的关键代码段，可能需要将各种特性添加给它，并用某种方法来增强它的性能。至少想要让它跟踪目前究竟哪个线程拥有该资源。如果有一个关键代码段能够实现这些操作，就能帮助解决代码中的死锁问题；可以使用一个调试程序来发现哪个线程没有释放该资源。

在进一步的说明之前，让我们来看一看究竟如何来实现关键代码段。我反复说，关键代码段属于用户方式对象。实际上，这种说法并不是百分之百的正确。如果一个线程试图进入另一个线程拥有的关键代码段，那么该线程就会被置于等待状态。如果要使它进入等待状态，唯一的办法是从用户方式转入内核方式。用户方式线程通过循环运行，就能够停止执行有用的操作，但是这不是个有效的等待方式，因此应该避免使用它。

关键代码段必须包含某个内核对象，以便使线程进入有效的等待状态。关键代码段的运行速度很快，因为只有当争用该关键代码段的时候，才使用该内核对象。只要线程能够立即获得对资源的访问权，并且使用该资源，然后释放该资源，而不与其他线程争用该资源，那么就不使用该内核对象，而且该线程决不会退出用户方式。在大多数应用程序中，两个线程很少会同时争用关键代码段。

Optex.h和Optex.cpp文件（见后面清单10-1）说明了关键代码段的实现方法。这里称关键代码段是一个Optex（这是optimized互斥对象（优化互斥对象）的缩略词），并将它作为一个C++类来实现。一旦理解了这个代码，就会懂得关键代码段的运行速度为什么比互斥对象内核对象快。

由于实现了关键代码段，因此可以将有用的特性添加给它。例如，COptex类使得不同进程中的线程能够实现同步。这是个令人叫绝的附加特性，这样就得到了一个高速运行的机制，使得不同进程中的线程之间能够互相进行通信。

若要使用optex，只需要声明一个COptex对象。该对象有3个构造函数：

```
COptex::(DWORD dwSpinCount = 4000);
```

China-pub.com
下载


```
COptex::(PCSTR  pszName, DWORD dwSpinCount = 4000);
COptex::(PCWSTR pszName, DWORD dwSpinCount = 4000);
```

第一个构造函数用于创建只能用来对单个进程中的各个线程进行同步的COptex对象。这种类型的optex占用的开销比跨进程的optex要少得多。另外两个构造函数可以用来创建在多个进程中的线程之间实现同步的optex。对于pszName参数，必须传递一个ANSI或Unicode字符串，该字符串用于对每个共享的optex进行标识。若要使两个或多个进程共享一个optex，那么两个进程必须建立一个COptex对象的实例，并且传递相同的字符串名字。

如果线程要进入和退出COptex对象，请调用Enter和leave方法：

```
void COptex::Enter();
void COptex::Leave();
```

我甚至列入了关键代码段的TryEnterCriticalSection和SetCriticalSectionSpinCount函数的等价方法：

```
BOOL COptex::TryEnter();
void COptex::SetSpinCount(DWORD dwSpinCount);
```

如果需要知道optex是属于单进程optex还是跨进程optex，可以调用下面所示的最后一种方法（很少需要调用这个方法，但是内部方法函数有时要调用它）。

```
BOOL COptex::IsSingleProcessOptex() const;
```

这些就是在使用optex时需要知道的所有（公用）函数。现在我准备介绍optex是如何运行的。一般来说，optex（和关键代码段）包含了许多成员变量。这些变量反映了optex的状态。在Optex.h文件中，大多数成员变量采用SHAREDINFO结构，少数成员变量属于类本身的成员。表10-1描述了每个成员的作用。

表10-1 成员变量描述

成 员	描 述
m_lLockCount	指明线程试图进入optex的次数。如果没有线程进入optex，那么这个值是0
m_dwThreadId	指明拥有optex的线程的唯一ID。如果没有线程拥有optex，那么这个值是0
m_lRecurseCount	指明线程拥有optex的次数。如果optex没有被线程所拥有，则这个值是0
m_hevt	这是个事件内核对象的句柄，只有当一个线程试图在另一个线程拥有optex时进入该optex，才使用这个句柄。内核对象句柄是与进程相关的句柄，这就是该成员为什么不使用SHAREDINFO结构的原因
m_dwSpinCount	指明试图进入optex的线程在等待事件内核对象之前应该尝试进入的次数。在单处理器计算机上，这个值总是0
m_hfm	这是文件映象内核对象的句柄，当多进程共享一个optex时，便使用这个句柄。内核对象句柄属于与进程相关的句柄，这就是为什么该成员不是SHAREDINFO结构的原因。对于单进程optex来说，这个值总是NULL
m_psi	这是指向潜在的共享optex数据成员的指针。内存地址是与进程相关的地址，这就是为什么该成员不使用SHAREDINFO结构的原因。对于单进程optex来说，它指向一个堆栈分配的内存块。对于多进程optex来说，它指向一个内存映射文件

该源代码已经作了充分的说明，因此要理解optex如何来运行，不会遇到什么困难。需要特别指出的是，optex之所以能够获得较快的运行速度，原因是它大量使用了互锁函数家族。这使得该代码始终能够以用户方式来运行，并且避免了方式转换操作。

Optex示例应用程序

清单10-1列出的Optex（“10 Optex.exe”）应用程序用于测试COptex类，以便确保它能够正

China-pub.com 下载

确地运行。该应用程序的源代码和源文件位于本书所附光盘上的 10-Optex目录下。我总是在调试程序中运行该应用程序，因此能够密切注意所有的成员函数和变量。

当运行该应用程序时，它首先要检测它是不是运行该应用程序的第一个实例。我的做法是创建一个带名字的事件内核对象。在该应用程序的任何地方，实际上并不使用事件对象，创建这个对象只是为了观察GetLastError是否返回ERROR_ALREADY_EXISTS。如果它返回了这个值，那么我就知道它是运行的该应用程序的第二个实例。下面说明为什么要运行该应用程序的两个实例。

如果这是第一个实例，那么我创建一个单进程COptex对象，并且调用FirstFunc函数。该函数对该optex对象执行一系列的操作。这时创建第二个线程，它也对同一个 optex对象进行操作。这时，对该optex进行操作的两个线程都在同一个进程中。可以观察源代码，以便了解我执行的是什么测试。我设法说明所有可能出现的情况，这样， Coptex类中的所有代码都能够有机会运行。

测试了单进程optex后，我又测试了跨进程optex。在_tWinMain中，当首次调用的FirstFunc返回时，我创建了另一个 COptex optex对象，但是这一次我给该 optex赋予了一个字符串名字CrossOptexTest。只需创建一个带名字的optex，就可以使它成为一个跨进程optex。接着，我第二次调用FirstFunc函数，给它传递了跨进程optex的地址。这时FirstFunc基本上执行与以前一样的代码。但是这次它不是产生第二个线程，而是产生了一个子进程。

该子进程只是同一个应用程序的另一个实例。但是，当它启动运行时，它创建了一个事件内核对象，并且发现该事件对象已经存在。这就是第二个应用程序实例如何知道它是第二个实例并且执行与第一个实例不同的代码的方法。第二个实例首先做的事情是调用 DebugBreak函数：

```
VOID DebugBreak();
```

这个使用方便的函数能够强制一个调试程序运行，并且将它自己与该进程连接起来。这样就可以方便地调试应用程序的这两个实例。然后第二个实例创建一个跨进程 optex，传递相同的字符串名字。由于字符串名字是相同的，因此两个进程共享该 optex。顺便要说明的是，两个以上的进程可以共享同一个optex。

接着，应用程序的第二个实例调用 SecondFunc函数，为它传递跨进程optex的地址。这时，进行同一组测试，但是对optex进行操作的两个线程不在同一进程中。

清单10-1 optex示例应用程序



Optex.cpp

```
/******************************************************************************
Module:  Optex.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include "Optex.h"


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
// 0=multi-CPU, 1=single-CPU, -1=not set yet
BOOL COptex::sm_fUniprocessorHost = -1;


//////////////////////////////////////////////////////////////////////////////


PSTR COptex::ConstructObjectName(PSTR pszResult,
   PCSTR pszPrefix, BOOL fUnicode, PVOID pszName) {

   pszResult[0] = 0;
   if (pszName == NULL)
      return(NULL);

   wsprintfA(pszResult, fUnicode ? "%s%S" : "%s%s", pszPrefix, pszName);
   return(pszResult);
}


//////////////////////////////////////////////////////////////////////////////
void COptex::CommonConstructor(DWORD dwSpinCount,
   BOOL fUnicode, PVOID pszName) {

   if (sm_fUniprocessorHost == -1) {
      // This is the 1st object constructed, get the number of CPUs
      SYSTEM_INFO sinf;
      GetSystemInfo(&sinf);
      sm_fUniprocessorHost = (sinf.dwNumberOfProcessors == 1);
   }

   m_hevt = m_hfm = NULL;
   m_psi  = NULL;

   if (pszName == NULL) {  // Creating a single-process optex

      m_hevt = CreateEventA(NULL, FALSE, FALSE, NULL);
      chASSERT(m_hevt != NULL);

      m_psi = new SHAREDINFO;
      chASSERT(m_psi != NULL);
      ZeroMemory(m_psi, sizeof(*m_psi));

   } else {                // Creating a cross-process optex

      // Always use ANSI so that this works on Win9x and Windows 2000
      char szResult[100];
      ConstructObjectName(szResult, "Optex_Event_", fUnicode, pszName);
      m_hevt = CreateEventA(NULL, FALSE, FALSE, szResult);
      chASSERT(m_hevt != NULL);

      ConstructObjectName(szResult, "Optex_MMF_", fUnicode, pszName);
      m_hfm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
         PAGE_READWRITE, 0, sizeof(*m_psi), szResult);
      chASSERT(m_hfm != NULL);
```

China-pub.com 下载

```
   m_psi = (PSHAREDINFO) MapViewOfFile(m_hfm,
      FILE_MAP_WRITE, 0, 0, 0);
   chASSERT(m_psi != NULL);

   // Note: SHAREDINFO's m_lLockCount, m_dwThreadId, and m_lRecurseCount
   // members need to be initialized to 0. Fortunately, a new pagefile
   // MMF sets all of its data to 0 when created. This saves us from
   // some thread synchronization work.
}

   SetSpinCount(dwSpinCount);
}


//////////////////////////////////////////////////////////////////////////////


COptex::~COptex() {

#ifdef _DEBUG
   if (IsSingleProcessOptex() && (m_psi->m_dwThreadId != 0)) {
      // A single-process optex shouldn't be destroyed if any thread owns it
      DebugBreak();
   }

   if (!IsSingleProcessOptex() &&
      (m_psi->m_dwThreadId == GetCurrentThreadId())) {

      // A cross-process optex shouldn't be destroyed if our thread owns it
      DebugBreak();
   }
#endif

   CloseHandle(m_hevt);

   if (IsSingleProcessOptex()) {
      delete m_psi;
   } else {
      UnmapViewOfFile(m_psi);
      CloseHandle(m_hfm);
   }
}


//////////////////////////////////////////////////////////////////////////////


void COptex::SetSpinCount(DWORD dwSpinCount) {

   // No spinning on single CPU machines
   if (!sm_fUniprocessorHost)
      InterlockedExchangePointer((PVOID*) &m_psi->m_dwSpinCount,
        (PVOID) (DWORD_PTR) dwSpinCount);
}
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////


void COptex::Enter() {

   // Spin, trying to get the optex
   if (TryEnter())
      return;  // We got it, return

   // We couldn't get the optex, wait for it.
   DWORD dwThreadId = GetCurrentThreadId();

   if (InterlockedIncrement(&m_psi->m_lLockCount) == 1) {

      // Optex is unowned, let this thread own it once
      m_psi->m_dwThreadId = dwThreadId;
      m_psi->m_lRecurseCount = 1;

   } else {

      if (m_psi->m_dwThreadId == dwThreadId) {

         // If optex is owned by this thread, own it again
         m_psi->m_lRecurseCount++;

      } else {

         // Optex is owned by another thread, wait for it
         WaitForSingleObject(m_hevt, INFINITE);

         // Optex is unowned, let this thread own it once
         m_psi->m_dwThreadId = dwThreadId;
         m_psi->m_lRecurseCount = 1;
      }
   }
}


//////////////////////////////////////////////////////////////////////////////


BOOL COptex::TryEnter() {

   DWORD dwThreadId = GetCurrentThreadId();
BOOL fThisThreadOwnsTheOptex = FALSE;     // Assume a thread owns the optex
DWORD dwSpinCount = m_psi->m_dwSpinCount; // How many times to spin

do {
   // If lock count = 0, optex is unowned, we can own it
   fThisThreadOwnsTheOptex = (0 ==
      InterlockedCompareExchange(&m_psi->m_lLockCount, 1, 0));

   if (fThisThreadOwnsTheOptex) {

      // Optex is unowned, let this thread own it once
```

China-pub.com 下载

```
         m_psi->m_dwThreadId = dwThreadId;
         m_psi->m_lRecurseCount = 1;

      } else {

         if (m_psi->m_dwThreadId == dwThreadId) {

            // If optex is owned by this thread, own it again
            InterlockedIncrement(&m_psi->m_lLockCount);
            m_psi->m_lRecurseCount++;
            fThisThreadOwnsTheOptex = TRUE;
         }
      }

   } while (!fThisThreadOwnsTheOptex && (dwSpinCount-- > 0));

   // Return whether or not this thread owns the optex
   return(fThisThreadOwnsTheOptex);
}


//////////////////////////////////////////////////////////////////////////////


void COptex::Leave() {

#ifdef _DEBUG
   // Only the owning thread can leave the optex
   if (m_psi->m_dwThreadId != GetCurrentThreadId())
      DebugBreak();
#endif

   // Reduce this thread's ownership of the optex
   if (--m_psi->m_lRecurseCount > 0) {
      // We still own the optex
      InterlockedDecrement(&m_psi->m_lLockCount);

   } else {

      // We don't own the optex anymore
      m_psi->m_dwThreadId = 0;

      if (InterlockedDecrement(&m_psi->m_lLockCount) > 0) {

         // Other threads are waiting, the auto-reset event wakes one of them
         SetEvent(m_hevt);
      }
   }
}


//////////////////////////////// End of File //////////////////////////////////
```

Optex.h

China-pub.com
下载


```
/******************************************************************************
Module name: Optex.h
Written by:  Jeffrey Richter
******************************************************************************/


#pragma once


////////////////////////////////////////////////////////////////////////////////


class COptex {
public:
   COptex(DWORD dwSpinCount = 4000);
   COptex(PCSTR  pszName, DWORD dwSpinCount = 4000);
   COptex(PCWSTR pszName, DWORD dwSpinCount = 4000);
   ~COptex();

   void SetSpinCount(DWORD dwSpinCount);
   void Enter();
   BOOL TryEnter();
   void Leave();
   BOOL IsSingleProcessOptex() const;

private:
   typedef struct {
      DWORD m_dwSpinCount;
      long  m_lLockCount;
      DWORD m_dwThreadId;
      long  m_lRecurseCount;
   } SHAREDINFO, *PSHAREDINFO;

   HANDLE      m_hevt;
   HANDLE      m_hfm;
   PSHAREDINFO m_psi;

private:
   static BOOL sm_fUniprocessorHost;

private:
   void CommonConstructor(DWORD dwSpinCount, BOOL fUnicode, PVOID pszName);
   PSTR ConstructObjectName(PSTR pszResult,
      PCSTR pszPrefix, BOOL fUnicode, PVOID pszName);
};


////////////////////////////////////////////////////////////////////////////////

inline COptex::COptex(DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, FALSE, NULL);
}

////////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
inline COptex::COptex(PCSTR pszName, DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, FALSE, (PVOID) pszName);
}


//////////////////////////////////////////////////////////////////////////////
inline COptex::COptex(PCWSTR pszName, DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, TRUE, (PVOID) pszName);
}


//////////////////////////////////////////////////////////////////////////////


inline COptex::IsSingleProcessOptex() const {

   return(m_hfm == NULL);
}


/////////////////////////////////// End of File //////////////////////////////////
```

Optex.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
```

China-pub.com
下载


```
// remains consistent on all systems.
IDI_OPTEX               ICON    DISCARDABLE     "Optex.ICO"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

OptexTest.cpp

```
/******************************************************************************
Module name: OptexTest.cpp
Written by:  Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <process.h>
#include "Optex.h"
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

DWORD WINAPI SecondFunc(PVOID pvParam) {

   COptex& optex = * (COptex*) pvParam;

   // The primary thread should own the optex here, this should fail
   chVERIFY(optex.TryEnter() == FALSE);

   // Wait for the primary thread to give up the optex
   optex.Enter();

   optex.Enter();  // Test recursive ownership
   chMB("Secondary: Entered the optex\n(Dismiss me 2nd)");

   // Leave the optex but we still own it once
   optex.Leave();
   chMB("Secondary: The primary thread should not display a box yet");
   optex.Leave();  // The primary thread should be able to run now

   return(0);
}

//////////////////////////////////////////////////////////////////////////////
VOID FirstFunc(BOOL fLocal, COptex& optex) {

   optex.Enter();  // Gain ownership of the optex

   // Since this thread owns the optex, we should be able to get it again
   chVERIFY(optex.TryEnter());

   HANDLE hOtherThread = NULL;
   if (fLocal) {
      // Spawn a secondary thread for testing purposes (pass it the optex)

      DWORD dwThreadId;
      hOtherThread = chBEGINTHREADEX(NULL, 0,
         SecondFunc, (PVOID) &optex, 0, &dwThreadId);

   } else {
      // Spawn a secondary process for testing purposes
      STARTUPINFO si = { sizeof(si) };
      PROCESS_INFORMATION pi;
      TCHAR szPath[MAX_PATH];
      GetModuleFileName(NULL, szPath, chDIMOF(szPath));
      CreateProcess(NULL, szPath, NULL, NULL,
         FALSE, 0, NULL, NULL, &si, &pi);
      hOtherThread = pi.hProcess;
      CloseHandle(pi.hThread);
   }

   // Wait for the second thread to own the optex
   chMB("Primary: Hit OK to give optex to secondary");
```

China-pub.com
下载


```
   // Let the second thread gain ownership of the optex
   optex.Leave();
   optex.Leave();

   // Wait for the second thread to own the optex
   chMB("Primary: Hit OK to wait for the optex\n(Dismiss me 1st)");

   optex.Enter();  // Try to gain ownership back

   WaitForSingleObject(hOtherThread, INFINITE);
   CloseHandle(hOtherThread);
   optex.Leave();
}

//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // This event is just used to determine which instance this is.
   HANDLE hevt = CreateEvent(NULL, FALSE, FALSE, TEXT("OptexTest"));

   if (GetLastError() != ERROR_ALREADY_EXISTS) {

      // This is the first instance of this test application

      // First, let's test the single-process optex
      COptex optexSingle;   // Create a single-process optex
      FirstFunc(TRUE, optexSingle);

      // Now, let's test the cross-process optex
      COptex optexCross("CrossOptexTest");   // Create a cross-process optex
      FirstFunc(FALSE, optexCross);

   } else {

      // This is the second instance of this test application
      DebugBreak();  // Force debugger connection for tracing

      // Test the cross-process optex
      COptex optexCross("CrossOptexTest");   // Create a cross-process optex
      SecondFunc((PVOID) &optexCross);
   }

   CloseHandle(hevt);
   return(0);
}


//////////////////////////////// End of File ////////////////////////////////
```

10.2 创建线程安全的数据类型和反信标

有一天我正在编写一些代码，这时需要一个内核对象，这个对象的行为特性必须与信标对

China-pub.com 下载

象的行为特性相反。当它的当前资源数量变为0时，便通知该对象，当它的当前资源数量大于0时，就不向该对象发出通知。

我发现这种类型的对象有许多用途。例如，有一个线程，当将某个操作运行100次时，需要将该线程唤醒。为了实现这个要求，需要一个能够将它初始化为100的内核对象。当该内核对象的数量大于0时，该对象不应该被通知。每当执行某个操作时，必须递减该内核对象中的数量。当该数量降到0时，该对象应该得到通知，这样，其他线程就能够醒来，以便处理某些操作。这是个常见的问题，我不知道Windows为什么没有提供这个内置特性。

实际上，Microsoft只要让一个信标的当前资源数量变为负值，就能很容易解决这个问题。可以将该信标的数量初始化为-99，然后在执行每个操作后调用ReleaseSemaphore函数。当该信标的数量到达1时，该对象就得到通知，并且其他线程能够醒来执行它的操作。可惜Microsoft不允许信标的数量变为负值，在可以预见的将来他们不会修改这个代码。

本节将介绍一组C++模板类，它们具有反信标的行为特性和一整套其他特性。这些类的代码都在Interlocked.h文件中（参见后面的清单10-2）。

当着手解决这个问题的时候，我认识到这个解决方案的核心是需要一个进行变量操作时线程安全的方法。我想设计一个巧妙的解决方案，以便能够非常容易地编写引用变量的代码。显然，要使资源成为线程安全的资源，最容易的办法是用关键代码段来保护它。使用C++类，就能够非常容易地为数据对象赋予线程安全的特性。要做的工作只不过是创建一个C++类，它包含想要保护的变量和一个CRITICAL_SECTION数据结构。然后，在该构造函数中，调用InitializeCriticalSection，在析构函数中，调用DeleteCriticalSection。对于所有其他成员变量，可以调用EnterCriticalSection，然后对变量进行操作，调用LeaveCriticalSection。如果用这种方法实现一个C++类，就可以很容易编写能用线程安全方式访问数据结构的代码。这是本节介绍的所有C++类的基本原则（当然可以使用前一节中讲述的optex，而不使用关键代码段）。

第一个类是个资源保护类，称为CResGuard。它包含一个CRITICAL_SECTION数据成员和一个LONG数据成员。LONG数据成员用于跟踪线程进入关键代码段的次数。这个信息可以用于调试。CResGuard对象的构造函数和析构函数分别调用InitializeCriticalSection和DeleteCriticalSection。由于只有单个线程能够创建对象，因此C++对象的构造函数和析构函数不必是线程安全的函数。IsGuarded成员函数只是返回是否为该对象至少调用了一次EnterCriticalSection。如前所述，这是用于调试目的的。将CRITICAL_SECTION放入一个C++对象，可以确保关键代码段被正确地初始化和删除。

CResGuard类也提供了一个嵌套的公用C++类CGuard。CGuard对象包含了一个对CResGuard对象的引用，并且只提供一个构造函数和析构函数。构造函数调用CResGuard的Guard成员函数，而该成员函数则调用EnterCriticalSection。CGuard的析构函数调用CResGuard的Unguard成员函数，而该成员函数则调用LeaveCriticalSection。有这样的方法来建立这些类，就可以更加容易地操作CRITICAL_SECTION。下面是使用这些类的一个小代码段：

```
struct SomeDataStruct {
   ...
} g_SomeSharedData;

// Create a CResGuard object that protects g_SomeSharedData.
// Note: The constructor initializes the critical section and
//       the destructor deletes the critical section.
CResGuard g_rgSomeSharedData;
```

China-pub.com
下载


```
void AFunction() {

   // This function touches the shared data.

   // Protect the resource from being accessed from multiple threads.
   CResGuard::CGuard gDummy(g_rgSomeSharedData); //Enter critical section
   // Touch the g_SomeSharedData resource.
   ...

}  // Note: LeaveCriticalSection is called when gDummy goes out of scope.
```

下一个C++类CInterlockedType包含了创建线程安全的数据对象时必要的全部元素。我将CInterlockedType类做成了一个模板类，这样，它就可以用于使任何数据类型成为线程安全的数据类型。例如，可以使用这个类建立一个线程安全的整数，一个线程安全的字符串 或一个线程安全的数据结构。

CInterlockedType对象的每个实例都包含两个数据成员。第一个数据成员是你想要使之成为线程安全的模板数据类型的实例。该数据成员是私有的，只能使用 CInterlockedType的成员函数进行操作。第二个数据成员是CResGuard对象的一个实例，它负责保护对数据成员的访问。CResGuard对象是个受保护的数据成员，因此，由CInterlockedType派生的类能够很容易地保护它的数据。

总是应该将 CInterlockedType类用作一个基类，以便派生一个新类。前面说过，CInterlockedType类提供了创建线程安全的对象时需要的所有元素，但是该派生类负责使用这些元素，以便用线程安全的方式来正确地访问数据值。

CInterlockedType类只提供了4个公有函数，一个是不对数据值进行初始化的构造函数，一个是对数据值进行初始化的构造函数，一个是什么也不做的虚拟析构函数，还有一个是类型转换操作符函数。类型转换操作符函数只是负责保证对数据值的线程安全访问，方法是保护资源并返回对象的当前值（当局部变量x超出规定的范围时，资源将自动失去保护）。类型转换操作符函数使得你能够更加容易地观察线程安全方式中的这个类包含的数据对象的值。

CInterlockedType类还提供了3个派生类将要调用的非虚拟保护的函数。两个 GetVal函数返回数据对象的当前值。在文件的调试代码中，这两个函数首先要查看数据对象是否得到了保护。如果该对象没有被保护，GetVal可以为该对象返回一个值，然后在原始调用线程查看该值之前，允许另一个线程修改该对象的值。

我假设调用线程获得了对象的值，因此它能够以某种方法来修改该值。根据这个假设，GetVal函数要求调用线程拥有对数据值的受保护的访问权。如果 GetVal函数确定该数据受到了保护，那么就返回数据的当前值。这两个 GetVal函数是相同的，不同之处在于其中一个是在对象的固定版本上运行的。这两个版本允许编写可以对常量数据类型和非常量数据类型进行操作的代码而不需要编译器生成警告信息。

第三个非虚拟受保护成员函数是 SetVal。当一个派生类的成员函数想要修改数据值时，该派生类的函数应该保护对该数据的访问权，然后调用 SetVal函数。与GetVal函数一样，SetVal函数首先要进行调试检查，以确保派生类的代码不会忘记对数据值访问权的保护。然后， SetVal要查看数据值是否真的正在被修改。如果正在被修改， SetVal就将老的值保存起来，将对象改为它的新值，然后调用一个虚拟的、受保护的成员函数 OnValChanged，并将老的和新的数据值传递给它。CInterlockedType类实现了一个OnValChanged成员函数，但是该函数并不做任何工作。可以使用OnValChanged成员函数将一些强大的功能添加给该派生类，就像下面我们介

China-pub.com 下载

绍CWhenZero类时的情况那样。

到现在为止，已经介绍了许多抽象类和它们的概念。下面来说明如何使用所有这些结构。首先介绍CInterlockedScalar类，这是由CInterlockedType派生的一个模板类。可以使用这个类创建线程安全的标量数据类型，如字节、字符、 16位整数、32位整数、64位整数和浮点值等。由于CInterlockedScalar类是CInterlockedType类派生而来的，因此它并不拥有它自己的数据类型。CInterlockedScalar的构造函数只是调用 CInterlockedType的构造函数，为该标量传递一个初始值。由于CInterlockedScalar类总是使用数字值，因此将默认构造函数的参数设置为 0，这样，对象总是在已知状态中创建。 CInterlockedScalar的构造函数根本就不进行任何操作。

CInterlockedScalar的其余成员函数全部用于修改标量值。可以用于对标量值进行的每个操作都有一个成员函数。为了使该CInterlockedScalar类能够用线程安全的方式对它的数据对象进行操作，所有这些成员函数都在操作前对数据值进行了保护。这些成员函数很简单，因此不对它们进行详细的说明。可以观察其代码，以便了解它们能够做些什么。不过要介绍一下如何使用这些类。下面的代码声明了线程安全的BYTE数据类型，并可以对它进行操作：

```
CInterlockedScalar<BYTE> b = 5;   // A thread-safe BYTE
BYTE b2 = 10;                     // A non-thread-safe BYTE
b2 = b++;                         // b2=5, b=6
b *= 4;                           // b=24
b2 = b;                           // b2=24, b=24
b += b;                           // b=48
b %= 2;                           // b=0
```

对线程安全的标量值进行操作就像对非线程安全的标量值的操作一样简单。实际上，由于C++的操作符被重载，它们的代码是相同的。运用迄今为止介绍的 C++类，可以很容易地将非线程安全的变量转换成线程安全的变量，只需要对源代码做很少的修改即可。

当我开始设计所有这些类的时候，我的脑子里就已经有了一个特定的目的：想要创建一个对象，它的行为特性与信标的行为特性正好相反。提供这个行为特性的C++类是CWhenZero类。CWhenZero类是由CInterlockedScalar类派生而来的。当标量值是0时，便通知CWhenZero对象。当该数据值不是0时，便不通知CWhenZero对象。这与信标的行为特性正好相反。

如你所知，C++对象不能被通知，只有内核对象才能被通知，并且可以用于线程的同步。因此，CWhenZero对象必须包含某些别的数据成员，这些成员是事件内核对象的句柄。一个CWhenZero对象包含两个数据成员，一个是 m_hevtZero，这是当数据值是0时得到通知的事件内核对象的句柄，另一个成员是m_hevtNotZero，这是当数据值不是0时得到通知的事件内核对象的句柄。

CWhenZero的构造函数接受数据对象的初始值，并且也让你设定这两个事件内核对象是人工重置的对象（默认值）还是自动重置的对象。然后该构造函数调用 CreateEvent，创建两个事件内核对象，并且根据数据的初始值是 0还是非0，将它们设置为已通知状态或未通知状态。CWhenZero的析构函数仅仅负责关闭两个事件的句柄。由于 CWhenZero的类公开继承了CInterlockedScalar类，因此CWhenZero对象的用户可以使用重载操作符函数的所有成员函数。

还记得OnValChanged能够保护CInterlockedType类中声明的成员函数吗？CWhenZero类重载了该虚拟函数。该函数负责根据数据对象的值，使事件内核对象保持已通知状态或者未通知状态，每当数据值变更时，便调用 OnValChanged函数。CWhenZero对该函数的实现代码查看新的值是否是0。如果是0，它就设置m_hevtZero事件，并且重置m_hevtNotZero事件。如果新值不是0，OnValChanged就不反转。

China-pub.com
下载

现在，当想要使线程等待数据值变为0时，只需要编写下面的代码：

```
CWhenZero<BYTE> b = 0;              // A thread-safe BYTE

// Returns immediately because b is 0
WaitForSingleObject(b, INFINITE);

b = 5;

//Returns only if another thread sets b to 0
WaitForSingleObject(b, INFINITE);
```

可以像前面那样编写对WaitForSingleObject的调用代码，因为CWhenZero类也包含一个类型转换操作符成员函数，它能够将CWhenZero对象转换成一个内核对象句柄。换句话说，如果将一个CWhenZero C++对象传递给期望一个HANDLE对象的函数，该类型转换操作符函数就会被调用，它的返回值被传递给该函数。 CWhenZero的HANDLE类型转换操作符函数返回了m_hevtNotZero事件内核对象的句柄。

CWhenZero类中的m_hevtNotZero事件句柄使你能够编写等待数据值不是0的代码。不幸的是，我已经拥有一个HANDLE类型转换操作符函数，因此不能拥有另一个返回 m_hevtNotZero句柄的函数。为了获得该句柄，我添加了 GetNotZeroHandle成员函数。使用该函数，可以编写下面的代码：

```
CWhenZero<BYTE> b = 5;      // A thread-safe BYTE

// Returns immediately because b is not 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);

b = 0;

// Returns only if another thread sets b to not 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);
```

InterlockedType示例应用程序

清单10-2所示的InterlockedType（“10 InterlockedType.exe”）应用程序用于测试我刚刚介绍的C++类。该应用程序的源代码和资源文件在本书所附光盘上的10- InterlockedType目录下。我总是在调试程序中运行该应用程序，因此能够清楚地观察所有的类成员函数和变量的变化。

该代码显示一种常见的编程环境，这个环境是，一个线程产生若干个工作线程，并对内存块进行初始化。然后主线程唤醒工作线程，这样它们就能够开始处理该内存块。这时，主线程必须终止自己的运行，直到所有工作线程运行完成。然后主线程用新数据重新对内存块进行初始化，并唤醒工作线程，以便重新启动整个进程。

通过观察这个代码，可以看到，要用便于阅读和维护的 C++代码来解决这个常见的编程问题是多么烦琐。如你所见，CWhenZero类为我们提供的行为特性大大超过了信标的相反行为特性。现在我们有一个线程安全的数字，当它的值是 0 时，它就被通知。虽然你可以递增或递减信标的值，但是你也可以用CWhenZero对象对该值进行加、减、乘、除、求模，并且将它设置为任何值，甚至运行位操作。CWhenZero对象的功能实际上比信标内核对象的功能大得多。

了解这些C++模板类的概念是非常有用的。例如，可以创建一个由 CInterlockedType类派生的CInterlockedString类。可以使用CInterlockedString类以线程安全的方式对字符串进行操作。然后可以用CInterlockedString类派生一个CWhenCertainString类，当字符串变成某个值或某几

China-pub.com 下载

个值时，它就能通知一个事件内核对象。这种可能性是无限的。

清单10-2 InterlockedType示例应用程序



IntLockTest.cpp

```
/******************************************************************************
Module:  IntLockTest.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include "Interlocked.h"


//////////////////////////////////////////////////////////////////////////////


// Set to TRUE when worker threads should terminate cleanly.
volatile BOOL g_fQuit = FALSE;


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI WorkerThread(PVOID pvParam) {

   CWhenZero<BYTE>& bVal = * (CWhenZero<BYTE> *) pvParam;

   // Should worker thread terminate?
   while (!g_fQuit) {

      // Wait for something to do
      WaitForSingleObject(bVal.GetNotZeroHandle(), INFINITE);

      // If we should quit, quit
      if (g_fQuit)
         continue;

      // Do something
      chMB("Worker thread: We have something to do");

      bVal--;     // We're done

      // Wait for all worker threads to stop
      WaitForSingleObject(bVal, INFINITE);
   }

   chMB("Worker thread: terminating");
   return(0);
}
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Initialize to indicate that NO worker threads have anything to do
   CWhenZero<BYTE> bVal = 0;

   // Create the worker threads
   const int nMaxThreads = 2;
   HANDLE hThreads[nMaxThreads];
   for (int nThread = 0; nThread < nMaxThreads; nThread++) {
      DWORD dwThreadId;
      hThreads[nThread] = CreateThread(NULL, 0,
         WorkerThread, (PVOID) &bVal, 0, &dwThreadId);
   }

   int n;
   do {
      // Do more work or stop running?
      n = MessageBox(NULL,
         TEXT("Yes: Give worker threads something to do\nNo: Quit"),
         TEXT("Primary thread"), MB_YESNO);

      // Tell worker threads that we're quitting
      if (n == IDNO)
      InterlockedExchangePointer((PVOID*) &g_fQuit, (PVOID) TRUE);

   bVal = nMaxThreads;  // Wake the worker threads

   if (n == IDYES) {

      // There is work to do, wait for the worker threads to finish
      WaitForSingleObject(bVal, INFINITE);
   }

} while (n == IDYES);

// There is no more work to do, the process wants to die.
// Wait for the worker threads to terminate
WaitForMultipleObjects(nMaxThreads, hThreads, TRUE, INFINITE);

// Close the worker thread handles.
for (nThread = 0; nThread < nMaxThreads; nThread++)
   CloseHandle(hThreads[nThread]);

// Tell the user that the process is dying
chMB("Primary thread: terminating");

return(0);
}

//////////////////////////////// End of File //////////////////////////////////
```

China-pub.com 下载

Interlocked.h

```
/******************************************************************************
Module:  Interlocked.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once

////////////////////////////////////////////////////////////////////////////////

// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
class CResGuard {
public:
   CResGuard()  { m_lGrdCnt = 0; InitializeCriticalSection(&m_cs); }
   ~CResGuard() { DeleteCriticalSection(&m_cs); }

   // IsGuarded is used for debugging
   BOOL IsGuarded() const { return(m_lGrdCnt > 0); }

public:
   class CGuard {
   public:
      CGuard(CResGuard& rg) : m_rg(rg) { m_rg.Guard(); };
      ~CGuard() { m_rg.Unguard(); }

   private:
      CResGuard& m_rg;
   };

private:
   void Guard()   { EnterCriticalSection(&m_cs); m_lGrdCnt++; }
   void Unguard() { m_lGrdCnt--; LeaveCriticalSection(&m_cs); }

   // Guard/Unguard can only be accessed by the nested CGuard class.
   friend class CResGuard::CGuard;

private:
   CRITICAL_SECTION m_cs;
   long m_lGrdCnt;   // # of EnterCriticalSection calls
};


////////////////////////////////////////////////////////////////////////////////


// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CInterlockedType {

public:     // Public member functions
```

China-pub.com
下载


```
   // Note: Constructors & destructors are always thread-safe
   CInterlockedType() { }
   CInterlockedType(const TYPE& TVal) { m_TVal = TVal; }
   virtual ~CInterlockedType()  { }

   // Cast operator to make writing code that uses
   // thread-safe data type easier
   operator TYPE() const {
      CResGuard::CGuard x(m_rg);
      return(GetVal());
   }

protected:  // Protected function to be called by derived class
   TYPE& GetVal() {
      chASSERT(m_rg.IsGuarded());
      return(m_TVal);
   }

   const TYPE& GetVal() const {
      assert(m_rg.IsGuarded());
      return(m_TVal);
   }

   TYPE SetVal(const TYPE& TNewVal) {
      chASSERT(m_rg.IsGuarded());
      TYPE& TVal = GetVal();
      if (TVal != TNewVal) {
         TYPE TPrevVal = TVal;
         TVal = TNewVal;
         OnValChanged(TNewVal, TPrevVal);
      }
      return(TVal);
   }

protected:  // Overridable functions
   virtual void OnValChanged(
      const TYPE& TNewVal, const TYPE& TPrevVal) const {
      // Nothing to do here
   }

protected:
   // Protected guard for use by derived class functions
      mutable CResGuard m_rg;

private:    // Private data members
   TYPE m_TVal;
};

//////////////////////////////////////////////////////////////////////////////

// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CInterlockedScalar : protected CInterlockedType<TYPE> {

public:
```

China-pub.com 下载

```
    CInterlockedScalar(TYPE TVal = 0) : CInterlockedType<TYPE>(TVal) {
    }

    ~CInterlockedScalar() { /* Nothing to do */ }

    // C++ does not allow operator cast to be inherited.
    operator TYPE() const {
       return(CInterlockedType<TYPE>::operator TYPE());
    }

    TYPE operator=(TYPE TVal) {
       CResGuard::CGuard x(m_rg);
       return(SetVal(TVal));
    }

    TYPE operator++(int) {    // Postfix increment operator
       CResGuard::CGuard x(m_rg);
       TYPE TPrevVal = GetVal();
       SetVal((TYPE) (TPrevVal + 1));
       return(TPrevVal);      // Return value BEFORE increment
    }

    TYPE operator--(int) {    // Postfix decrement operator.
       CResGuard::CGuard x(m_rg);
       TYPE TPrevVal = GetVal();
       SetVal((TYPE) (TPrevVal - 1));
       return(TPrevVal);      // Return value BEFORE decrement
    }

    TYPE operator += (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() +  op)); }
    TYPE operator++()
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() +   1)); }
    TYPE operator -= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() -  op)); }
    TYPE operator--()
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() -   1)); }
    TYPE operator *= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() *  op)); }
    TYPE operator /= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() /  op)); }
    TYPE operator %= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() %  op)); }
    TYPE operator ^= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() ^  op)); }
    TYPE operator &= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() &  op)); }
    TYPE operator |= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() |  op)); }
    TYPE operator <<=(TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() << op)); }
    TYPE operator >>=(TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() >> op)); }
};
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////


// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CWhenZero : public CInterlockedScalar<TYPE> {
public:
   CWhenZero(TYPE TVal = 0, BOOL fManualReset = TRUE)
      : CInterlockedScalar<TYPE>(TVal) {

      // The event should be signaled if TVal is 0
      m_hevtZero = CreateEvent(NULL, fManualReset, (TVal == 0), NULL);

      // The event should be signaled if TVal is NOT 0
      m_hevtNotZero = CreateEvent(NULL, fManualReset, (TVal != 0), NULL);
   }

   ~CWhenZero() {
      CloseHandle(m_hevtZero);
      CloseHandle(m_hevtNotZero);
   }
   // C++ does not allow operator= to be inherited.
   TYPE operator=(TYPE x) {
      return(CInterlockedScalar<TYPE>::operator=(x));
   }

   // Return handle to event signaled when value is zero
   operator HANDLE() const { return(m_hevtZero); }

   // Return handle to event signaled when value is not zero
   HANDLE GetNotZeroHandle() const { return(m_hevtNotZero); }

   // C++ does not allow operator cast to be inherited.
   operator TYPE() const {
      return(CInterlockedScalar<TYPE>::operator TYPE());
   }

protected:
   void OnValChanged(const TYPE& TNewVal, const TYPE& TPrevVal) const {
      // For best performance, avoid jumping to
      // kernel mode if we don't have to
      if ((TNewVal == 0) && (TPrevVal != 0)) {
         SetEvent(m_hevtZero);
         ResetEvent(m_hevtNotZero);
      }
      if ((TNewVal != 0) && (TPrevVal == 0)) {
         ResetEvent(m_hevtZero);
         SetEvent(m_hevtNotZero);
      }
   }

private:
   HANDLE m_hevtZero;      // Signaled when data value is 0
```

China-pub.com 下载

```
    HANDLE m_hevtNotZero;   // Signaled when data value is not 0
};

//////////////////////////////////// End of File /////////////////////////////////////
```

InterlockedType.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_INTERLOCKEDTYPE     ICON    DISCARDABLE     "InterLockedType.ICO"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END
```

China-pub.com
下载


```
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

10.3 单个写入程序/多个阅读程序的保护

许多应用程序存在一个基本的同步问题，这个问题称为单个写入程序 /多个阅读程序环境。该问题涉及到试图访问共享资源的任意数量的线程。这些线程中，有些线程（写入程序）需要修改数据的内容，而有些线程（阅读程序）则需要读取数据。由于下面 4个原则，它们之间的同步是必要的：

1) 当一个线程正在写入数据时，其他任何线程不能写入数据。

2) 当一个线程正在写入数据时，其他任何线程不能读取数据。

3) 当一个线程正在读取数据时，其他任何线程不能写入数据。

4) 当一个线程正在读取数据时，其他线程也能够读取数据。

让我们观察一下数据库应用程序环境中的这个问题。比如说，有 5个最终用户，他们都要访问同一个数据库。两个员工将一些记录输入该数据库，3个员工则从该数据库中检索记录。

在这个环境中，必须使用原则 1，因为我们不能让员工 1和员工2同时更新同一个记录。如果两个员工都试图修改同一个记录，那么员工 1的修改和员工2的修改就会在同一时间内进行，该记录中的信息就会被破坏。

原则2用于在某个员工更新数据库中的记录时禁止另一个员工访问该记录。如果不能防止这种情况的发生，那么员工 4就能够在员工1修改一个记录时读取该记录的内容。当员工 4的计算机显示该记录时，该记录将包含某些老的信息和某些更新过的信息，这当然是不行的。原则3可以用来解决同样的问题。无论谁首先拥有对数据库记录的访问权，即无论是试图写入的员工还是试图读取的员工，原则2和原则3用词上的差别都可以防止出现上面所说的情况。

原则4用于解决性能上的问题。如果没有员工试图修改数据库中的记录，那么数据库中的内容就不会变更，因此凡是只想从数据库中检索记录的员工都应该被允许进行检索。它的另一个前提是阅读者多于写入者。

好了，你已经掌握了问题的要领。现在的问题是，如何来解决这个问题。

China-pub.com 下载

注意 这里介绍的代码是新编写的。以前我提出的对这个问题的解决办法受到了一些人的批评，原因有二。首先，我以前编写的代码运行速度太慢，因为它们的设计初衷是要在许多环境中都能运行。例如，它们使用了较多的内核对象，因此不同进程中的线程能够同步它们对数据库的访问。当然，我的实现代码仍然可以在单进程环境中运行，但是，大量使用内核对象后，当所有线程都在单进程中运行时，就会增加很大的开销。必须承认，单进程的情况是更加常见的一种情况。

第二个批评是说，我的实现代码有可能完全把写入线程锁在外面。从前面介绍的几个原则来看，如果有许多阅读线程访问数据库，那么写入线程就永远无法访问该资源。

这里展示的实现代码解决了上面所说的两个问题。它尽可能避免使用内核对象，而是使用关键代码段，以便实现大多数同步问题。

为了简化操作，我将解决方案封装在一个C++类中，它称为CSWNRG，是“单个写入程序/多个阅读程序的保护”的英文缩写。SWMRG.h和SWMRG.cpp文件（参见后面的清单10-3）显示了实现代码。

CSWMRG的使用是再容易不过的了。只需要创建CSWMRG C++类的一个对象，然后按照应用程序的指令调用相应的成员函数。C++类中只有下面的3个方法（不包括构造函数和析构函数）：

```
VOID CSWMRG::WaitToRead();  // Call this to gain shared read access.
VOID CSWMRG::WaitToWrite(); // Call this to gain exclusive write access.
VOID CSWMRG::Done();        // Call this when done accessing the resource.
```

在执行读取共享资源的代码之前，调用第一个方法WaitToRead。在执行读取或写入共享资源的代码之前，调用第二个方法WaitToWrite。当代码不再访问共享资源时，调用第三个方法Done。这不是很简单吗？

一般来说，CSWMRG对象包含许多成员变量，这些变量反映了线程访问共享资源时的状态。表10-2描述了每个成员变量的作用，并且综合说明了它的整个工作情况。详细信息参见源代码。

表10-2 CSWMRG对象中成员变量的作用

成 员	描 述
m_cs	用于保护所有的其他成员变量，这样，对它们的操作就能够以原子操作方式来完成
m_nActive	用于反映共享资源的当前状态。如果该值是0，那么没有线程在访问资源。如果该值大于0，这个值用于表示当前读取该资源的线程的数量。如果这个数量是负值，那么写入程序正在将数据写入该资源。唯一有效的负值是-1
m_nWaitingReaders	表示想要访问资源的阅读线程的数量。该值被初始化为0，当m_nActive是-1时，每当线程调用一次WaitToRead，该值就递增1
m_nWaitingWriters	表示想要访问资源的写入线程的数量。该值被初始化为0，当m_nActive大于0时，每当线程调用一次WaitToWrite，该值就递增1
m_hsemWriters	当线程调用WaitToWrite，但是由于m_nActive大于0而被拒绝访问时，所有写入线程均等待该信标。当一个线程正在等待时，新阅读线程将被拒绝访问该资源。这可以防止阅读线程垄断该资源。当最后一个拥有资源访问权的阅读线程调用Done时，该信标就被释放，其数量是1，从而唤醒一个正在等待的写入线程
m_hsemReaders	当许多线程调用WaitToRead，但是由于m_nActive是-1而被拒绝访问时，所有阅读线程均等待该信标。当最后一个正在等待的阅读线程调用Done时，该信标被释放，其数量是m_nWaitingReaders，从而唤醒所有正在等待的阅读线程

China-pub.com
下载

SWMRG示例应用程序

清单10-3列出的SWMRG应用程序（“10 SWMRG.exe”）用于测试CSWMRG C++类。该应用程序的源代码和资源文件均存放在本书所附光盘上的10 SWMRG目录下。我总是在调试程序中运行该应用程序，这样，就能清楚地观察所有的类成员函数和变量的变化。

当运行该应用程序时，主线程就会产生若干个线程，它们全部运行相同的线程函数。然后，主线程通过调用 WaitForMultipleObjects函数，等待所有这些线程终止运行，当所有的线程终止运行后，它们的句柄被关闭，该进程退出。

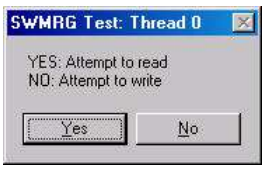


图10-1 辅助线程显示的消息

每个辅助线程均显示一条类似图10-1所示的消息。

如果想要该线程模拟阅读该资源，单击 Yes按钮。如果想要该线程模拟写入该资源，单击 No。这些操作只是使线程分别调用 SWMRG对象的WaitToRead或WaitToWrite函数。

当调用两个函数中的一个之后，该线程就显示另一个消息框，类似图 10-2或图10-3所示的消息框。

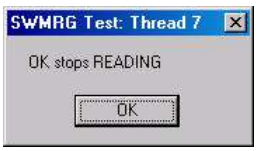


图10-2 调用SWMRG对象的WaitToRead后显示的消息框



图10-3 调用SWMRG对象的WaitToWrite后显示的消息框

该消息框将暂停线程的运行，暂停的时间相当于拥有资源访问权的线程对该资源进行操作所用的时间。

当然，如果一个线程当前正在读取资源，而你又命令另一个线程写入该资源，那么写入线程的消息框就不会出现，因为该线程正在它对 WaitToWrite的调用中等待。同样，如果命令一个线程读取资源，而另一个线程的消息框已经显示，那么想要读取资源的线程就会在对WaitToRead的调用中暂停运行，它的消息框将不会出现，直到任何一个或所有写入线程完成它们对资源的模拟访问为止。

当单击OK按钮，以退出任何一个消息框时，拥有对资源访问权的线程就调用 Done。而SWMRG对象便终止任何正在等待的线程的运行。

清单10-3 SWMRG示例应用程序



SWMRG.cpp

```
/******************************************************************************
Module:  SWMRG.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include "SWMRG.h"
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

CSWMRG::CSWMRG() {

   // Initially no readers want access, no writers want access, and
   // no threads are accessing the resource
   m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
   m_hsemReaders = CreateSemaphore(NULL, 0, MAXLONG, NULL);
   m_hsemWriters = CreateSemaphore(NULL, 0, MAXLONG, NULL);
   InitializeCriticalSection(&m_cs);
}


//////////////////////////////////////////////////////////////////////////////


CSWMRG::~CSWMRG() {

#ifdef _DEBUG
   // A SWMRG shouldn't be destroyed if any threads are using the resource
   if (m_nActive != 0)
      DebugBreak();
#endif

   m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
   DeleteCriticalSection(&m_cs);
   CloseHandle(m_hsemReaders);
   CloseHandle(m_hsemWriters);
}


//////////////////////////////////////////////////////////////////////////////


VOID CSWMRG::WaitToRead() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   // Are there writers waiting or is a writer writing?
   BOOL fResourceWritePending = (m_nWaitingWriters || (m_nActive < 0));

   if (fResourceWritePending) {

      // This reader must wait, increment the count of waiting readers
      m_nWaitingReaders++;
  } else {

     // This reader can read, increment the count of active readers
     m_nActive++;
  }

  // Allow other threads to attempt reading/writing
  LeaveCriticalSection(&m_cs);

  if (fResourceWritePending) {
```

China-pub.com
下载


```
      // This thread must wait
      WaitForSingleObject(m_hsemReaders, INFINITE);
   }
}

//////////////////////////////////////////////////////////////////////////////

VOID CSWMRG::WaitToWrite() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   // Are there any threads accessing the resource?
   BOOL fResourceOwned = (m_nActive != 0);

   if (fResourceOwned) {

      // This writer must wait, increment the count of waiting writers
      m_nWaitingWriters++;
   } else {

      // This writer can write, decrement the count of active writers
      m_nActive = -1;
   }

   // Allow other threads to attempt reading/writing
   LeaveCriticalSection(&m_cs);

   if (fResourceOwned) {

      // This thread must wait
      WaitForSingleObject(m_hsemWriters, INFINITE);
   }
}

//////////////////////////////////////////////////////////////////////////////

VOID CSWMRG::Done() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   if (m_nActive > 0) {

      // Readers have control so a reader must be done
      m_nActive--;
   } else {

      // Writers have control so a writer must be done
      m_nActive++;
   }

   HANDLE hsem = NULL;  // Assume no threads are waiting
   LONG lCount = 1;     // Assume only 1 waiter wakes; always true for writers

   if (m_nActive == 0) {
```

China-pub.com 下载

```
      // No thread has access, who should wake up?
      // NOTE: It is possible that readers could never get access
      //       if there are always writers wanting to write

      if (m_nWaitingWriters > 0) {

         // Writers are waiting and they take priority over readers
         m_nActive = -1;          // A writer will get access
         m_nWaitingWriters--;     // One less writer will be waiting
         hsem = m_hsemWriters;    // Writers wait on this semaphore
         // NOTE: The semaphore will release only 1 writer thread

      } else if (m_nWaitingReaders > 0) {

         // Readers are waiting and no writers are waiting
         m_nActive = m_nWaitingReaders;   // All readers will get access
         m_nWaitingReaders = 0;           // No readers will be waiting
         hsem = m_hsemReaders;            // Readers wait on this semaphore
         lCount = m_nActive;              // Semaphore releases all readers
      } else {

         // There are no threads waiting at all; no semaphore gets released
      }
   }
   // Allow other threads to attempt reading/writing
   LeaveCriticalSection(&m_cs);

   if (hsem != NULL) {

      // Some threads are to be released
      ReleaseSemaphore(hsem, lCount, NULL);
   }
}

//////////////////////////////// End of File //////////////////////////////////
```

SWMRG.h

```
/******************************************************************************
Module:  SWMRG.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once

//////////////////////////////////////////////////////////////////////////////

class CSWMRG {
public:
   CSWMRG();                  // Constructor
   ~CSWMRG();                 // Destructor

   VOID WaitToRead();         // Call this to gain shared read access
   VOID WaitToWrite();        // Call this to gain exclusive write access
   VOID Done();               // Call this when done accessing the resource
```

China-pub.com
下载


```
private:
   CRITICAL_SECTION m_cs;    // Permits exclusive access to other members
   HANDLE m_hsemReaders;     // Readers wait on this if a writer has access
   HANDLE m_hsemWriters;     // Writers wait on this if a reader has access
   int    m_nWaitingReaders; // Number of readers waiting for access
   int    m_nWaitingWriters; // Number of writers waiting for access
   int    m_nActive;         // Number of threads currently with access
                             //   (0=no threads, >0=# of readers, -1=1 writer)
};

////////////////////////////////// End of File /////////////////////////////////////
```

SWMRG.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SWMRG               ICON    DISCARDABLE     "SWMRG.ico"

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

China-pub.com 下载

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

SWMRGTest.cpp

```
/******************************************************************************
Module:  SWMRGTest.Cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <process.h>       // for _beginthreadex
#include "SWMRG.h"
/////////////////////////////////////////////////////////////////////////////


// Global Single-Writer/Multiple-Reader Guard synchronization object
CSWMRG g_swmrg;


/////////////////////////////////////////////////////////////////////////////


DWORD WINAPI Thread(PVOID pvParam) {

   TCHAR sz[50];
   wsprintf(sz, TEXT("SWMRG Test: Thread %d"), PtrToShort(pvParam));
   int n = MessageBox(NULL,
```

China-pub.com
下载


```
      TEXT("YES: Attempt to read\nNO: Attempt to write"), sz, MB_YESNO);

   // Attempt to read or write
   if (n == IDYES)
      g_swmrg.WaitToRead();
   else
      g_swmrg.WaitToWrite();

   MessageBox(NULL,
      (n == IDYES) ? TEXT("OK stops READING") : TEXT("OK stops WRITING"),
      sz, MB_OK);

   // Stop reading/writing
   g_swmrg.Done();
   return(0);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Spawn a bunch of threads that will attempt to read/write
   HANDLE hThreads[MAXIMUM_WAIT_OBJECTS];
   for (int nThreads = 0; nThreads < 8; nThreads++) {
      DWORD dwThreadId;
      hThreads[nThreads] =
         chBEGINTHREADEX(NULL, 0, Thread, (PVOID) (DWORD_PTR) nThreads,
            0, &dwThreadId);
   }
   // Wait for all the threads to exit
   WaitForMultipleObjects(nThreads, hThreads, TRUE, INFINITE);
   while (nThreads--)
      CloseHandle(hThreads[nThreads]);

   return(0);
}

//////////////////////////////// End of File ////////////////////////////////
```

10.4 实现一个WaitForMultipleExpressions函数

不久以前，我正在编写一个应用程序，我必须解决复杂的线程同步问题。WaitForMultipleObjects函数虽然能够让线程等待单个对象或多个对象，但是它不能满足我的需要。我需要的是一个能够表达更丰富的等待环境的函数。我有3个内核对象，即一个进程，一个信标和一个事件。我需要一种方法，使我的线程能够等待进程和信标都得到通知，或者进程和事件都得到通知为止。

通过创造性地运用现有的一些Windows函数，我创建了所需要的函数WaitForMultipleExpressions。我建立了下面这个函数原型：

China-pub.com 下载

```
DWORD WINAPI WaitForMultipleExpressions(
   DWORD nExpObjects,
   CONST HANDLE* phExpObjects,
   DWORD dwMilliseconds);
```

若要调用该函数，首先必须指定一个HANDLE数组，并对该数组的所有项目进行初始化。nExpObjects参数用于指明phExpObjects参数指向的数组中的项目数量。该数组包含多组内核对象句柄，每组句柄之间用一个NULL句柄项分开，WaitForMultipleExpressions将单组句柄中的对象视为用AND组合起来的对象组，而各个句柄组则是用OR组合起来的句柄组。因此，调用WaitForMultipleExpressions函数就可以暂停调用线程的运行，直到单组句柄中的所有对象均已同时得到通知为止。

下面是个例子。假设使用表10-3中的4个内核对象。

表10-3 内核对象的句柄值

对 象	句 柄 值
线程	0x1111
信标	0x2222
事件	0x3333
进程	0x4444

像下面这样对句柄数组进行初始化，就可以命令WaitForMultipleObjects函数暂停调用线程的运行，直到线程与信标得到通知，或者信标与事件与进程得到通知，或者线程与进程得到通知。如表10-4所示。

表10-4 句柄数组

索 引	句 柄 值	组
0	0x1111（线程）	0
1	0x2222（信标）	
2	0x0000（OR）	
3	0x2222（信标）	1
4	0x3333（事件）	
5	0x4444（进程）	
6	0x0000（OR）	
7	0x1111（线程）	2
8	0x4444（进程）	

也许你还记得，不能调用WaitForMultipleObjects来传递超过64个（MAXIMUM_WAIT_OBJECTS）项目的句柄数组。运用WaitForMultipleExpressions，该句柄数组的项目可以大大超过64。然而你不得拥有64个以上的表达式，并且每个表达式可以包含63个以上的句柄。另外，如果将一个互斥对象的句柄传递给它，那么WaitForMultipleExpressions将不能正确运行。

表10-5显示了WaitForMultipleExpressions可能的返回值。如果一个表达式真的实现了，那么WaitForMultipleExpressions将返回基于WAIT_OBJECT_0的该表达式的索引。使用该例子，如果线程和进程对象变为已通知状态，WaitForMultipleExpressions就返回索引WAIT_OBJECT_0+2。

China-pub.com
下载

表10-5 WaitForMultipleExpressions 的返回值

返 回 值	描 述
WAIT_OBJECT_0至（WAIT_OBJECT_0+表达式－1的号码）	用于指明哪个表达式被选定了
WAIT_TIMEOUT	在指定的时间内没有选定表达式
WAIT_FAILED	产生一个错误。若要了解详细信息，调用 GetLastError。如果产生的错误是ERROR_TOO_MANY_SECRETS，那么意味着设定的表达式超过了 64个。如果产生的错误是ERROR_SECRET_TOO_LONG，则意味着至少有一个表达式设定的对象超过了63个。也可能返回别的错误代码（为了我自己的目的，我不得不使用这两个错误代码）

WaitForMultipleExpressions示例应用程序

清单10-4列出的WaitForMultipleExpressions应用程序（“10 WaitForMultExp.exe”）用于测试WaitForMultipleExpressions函数。该应用程序的源代码和资源文件均存放在本书所附光盘上10-WaitForMultExp目录下。当运行该应用程序时，就会出现图10-4所示的对话框。

如果不改变任何设置，并且单击Wait For Multiple Expressions按钮，就会出现图10-5所示对话框。

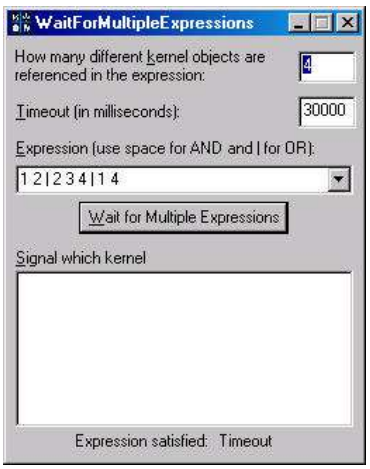


图10-4 WaitForMultiple Expressions 对话框

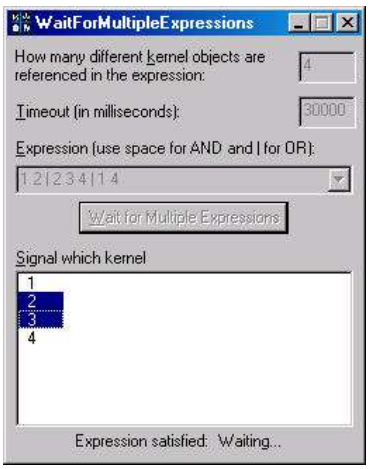


图10-5 单击WaitForMultipleExpressions按钮后出现的对话框

在内部，该应用程序创建了4个事件内核对象，开始时它们都处于未通知状态，同时，它为每个内核对象将一个项目放入这个多列、多节列表框中。然后，该应用程序对表达式的各个域进行分析，并创建句柄数组。我选择了与前面例子相吻合的4个内核对象和一个表达式。

由于我设定的超时为30000ms，因此有30s时间可以用来对列表框中的项目进行打开和关闭的切换，以便选定和取消有关的事件对象。如果选定了一个项目，则调用 SetEvent，给对象发送通知，如果取消一个项目，则调用 ResetEvent，使该事件取消已通知状态。当切换了足够的项目以便满足表达式中的某一个后， WaitForMultipleExpressions函数就返回，并在对话框的底部显示哪个表达式的条件得到了满足。如果在 30s内没有表达式的条件得到满足，则出现“Timeout”（超时）字样。

China-pub.com 下载

下面要介绍如何来实现WaitForMultipleExpressions函数。这是个不容易实现的函数，当使用这个函数的时候，必须注意某些开销问题。如你所知，Windows提供了WaitForMultipleObjects函数，它使得线程可以等待单个AND表达式：

```
DWORD WaitForMultipleObjects(
   DWORD dwObjects,
   CONST HANDLE* phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds);
```

为了扩展该函数，使之包含使用OR的表达式，必须生成多个线程：每个OR表达式需要一个线程。这些线程中的每一个都使用WaitForMultipleObjectsEx来等待AND表达式（我使用WaitForMultipleObjectsEx，而不是使用更常用的WaitForMultipleObjects函数，其原因将在下面说明）。当其中的一个表达式被选定时，生成的线程中就有一个被唤醒并终止运行。

调用WaitForMultipleExpressions函数的线程（它与产生所有OR线程的线程相同）必须等待，直到其中的一个OR表达式得以实现。它是通过调用WaitForMultipleObjectsEx来实现的。生成的线程（OR表达式）的数量被传递给dwObjects参数，phObjects参数则指向一个数组，该数组包含生成的线程句柄的列表。对于fWaitAll参数，则传递FALSE，这样，一旦任何一个表达式得以实现，主线程就立即醒来。最后，传递给WaitForMultipleExpressions的dwTimeout值被传递给WaitForMultipleObjectsEx。

如果在规定的时间内没有任何表达式得以实现，WaitForMultipleObjectsEx就返回WAIT_TIMEOUT，而WaitForMultipleExpressions也返回WAIT_TIMEOUT。如果有一个表达式得以实现，那么WaitForMultipleExpressions就返回一个索引值，指明哪个线程已经终止运行。由于每个线程都是一个独立的表达式，因此该索引值也指明哪个表达式已经得以实现，并且WaitForMultipleExpressions返回了相同的索引值。

这就是WaitForMultipleExpressions函数如何运行的基本情况。但是还有3个较小的问题需要具体加以说明。首先，我们不希望多个OR线程在调用WaitForMultipleExpressions的时候同时醒来，因为成功地等待某个内核对象会导致该对象改变其状态。例如，等待一个信标会导致它的数量递减1。WaitForMultipleExpressions只等待一个表达式得以实现，因此必须防止对象多次改变它的状态。

对这个问题的解决方案实际上很简单。在生成OR线程之前，我创建了一个我自己的信标对象，其初始数量是1。然后，OR线程对WaitForMultipleObjectsEx的每次调用都包含该信标的句柄和表达式中的其他句柄。这说明每组句柄可以设定的句柄不得超过63个。为了使一个OR线程醒来，它等待的所有对象，包括我的私有信标对象，都必须得到通知。由于我给信标设定的初始数量是1，因此唤醒的OR线程不能超过1个，同时其他对象也不会不小心改变其状态。

第二个需要说明的具体问题是如何强制正在等待的线程停止等待，以便正确地撤消。添加信标可以确保醒来的线程不会超过1个，但是一旦我知道哪个表达式等待实现，我就必须强制剩余的线程醒来，这样，它们就能够明确地终止运行，释放它们的内存堆栈。应该始终避免调用TerminateThread函数，因此需要另一个机制。在思考了一会儿后，我想到，当一个项目进入异步过程调用（APC）队列时，如果等待线程处于待命状态，那么它们就会被强制唤醒。

我的WaitForMultipleExpressions实现代码使用QueueUserAPC来强制等待线程醒来。当主线程调用的WaitForMultipleObjects返回时，我将一个APC项放入每个还在等待的OR线程的队列：

China-pub.com
下载


```
// Break all the waiting expression threads out of their
// wait state so that they can terminate cleanly.
for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

   if ((WAIT_TIMEOUT == dwWaitRet) ||
      (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {

      QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
   }
}
```

回调函数WFME_ExpressionAPC之所以采用这种形式，原因是我实际上并没有什么事情要做，只是想要让线程停止等待。

```
// This is the APC callback routine function.
VOID WINAPI WFME_ExpressionAPC(DWORD dwData) {

   // This function intentionally left blank
}
```

第三个需要说明的具体问题是如何正确处理超时。如果线程在等待时没有任何表达式等待实现，那么主线程调用的WaitForMultipleObjects就返回一个WAIT_TIMEOUT值。如果出现这种情况，我就想要防止任何表达式得以实现，这可能导致对象改变它们的状态。下面这个代码能够做到这一点：

```
// Wait for an expression to come TRUE or for a timeout.
dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
   FALSE, dwMilliseconds);

if (WAIT_TIMEOUT == dwWaitRet) {
// We timed out; check if any expressions were satisfied by
// checking the state of the hsemOnlyOne semaphore.
dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

if (WAIT_TIMEOUT == dwWaitRet) {

   // If the semaphore was not signaled, some thread expression
   // was satisfied; we need to determine which expression.
   dwWaitRet = WaitForMultipleObjects(dwExpNum,
      ahThreads, FALSE, INFINITE);

} else {

   // No expression was satisfied and WaitForSingleObject just gave
   // us the semaphore, so we know that no expression can ever be
   // satisfied now -- waiting for an expression has timed out.
   dwWaitRet = WAIT_TIMEOUT;
}
}
```

我用等待信标的办法来防止其他表达式的实现。这将使信标的数量递减为 0，同时所有的OR线程都不会醒来。但是，在主线程调用WaitForMultipleObjects和它调用WaitForSingleObject之后的某个位置上，一个表达式可能得以实现。这就是为什么要检查调用 WaitForSingleObject的返回值的原因。如果它返回WAIT_OBJECT_0，那么主线程得到了信标对象，并且没有一个

China-pub.com 下载

表达式得以实现。但是，如果返回WAIT_TIMEOUT，那么在主线程得到信标前，一个表达式就会得以实现。若要确定哪个表达式得到了实现，主线程要再次调用超时为INFINITE的WaitForMultipleObjects，这是可行的，因为我知道一个OR线程得到了信标，并且将要终止运行。这时，必须强制OR线程醒来，这样它们才能彻底退出。调用QueueUserAPC的循环能够进行这项操作。

由于WaitForMultipleExpressions是通过使用不同的线程来等待每组用AND连接起来的对象而在内部实现的，因此可以非常容易地了解为什么不能使用互斥对象。与其他内核对象不同，互斥对象可以被线程所拥有。因此，如果AND线程之一获得对互斥对象的所有权，那么当线程终止运行时，它就会放弃该互斥对象。如果Microsoft给Windows增加了一个函数，使得一个线程能够将互斥对象的所有权转交给另一个线程，那么WaitForMultipleExpressions函数就能够很容易地调整以便相应地支持互斥对象。在出现这个WaitForMultipleExpressions函数之前，一直没有支持互斥对象的出色方法。

清单10-4 WaitForMultipleExpressions示例应用程序



WaitForMultExp.cpp

```
/******************************************************************************
Module:  WaitForMultExp.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <malloc.h>
#include <process.h>
#include "WaitForMultExp.h"


//////////////////////////////////////////////////////////////////////////////


// Internal data structure representing a single expression.
// Used to tell OR-threads what objects to wait on.
typedef struct {
   PHANDLE m_phExpObjects;   // Points to set of handles
   DWORD   m_nExpObjects;    // Number of handles
} EXPRESSION, *PEXPRESSION;


//////////////////////////////////////////////////////////////////////////////


// The OR-thread function
DWORD WINAPI WFME_ThreadExpression(PVOID pvParam) {

   // This thread function just waits for an expression to come true.
   // The thread waits in an alertable state so that it can be forced
```

China-pub.com
下载


```
   // to stop waiting by queuing an entry to its APC queue.
   PEXPRESSION pExpression = (PEXPRESSION) pvParam;
   return(WaitForMultipleObjectsEx(
      pExpression->m_nExpObjects, pExpression->m_phExpObjects,
      TRUE, INFINITE, TRUE));
}


//////////////////////////////////////////////////////////////////////////////
// This is the APC callback routine function
VOID WINAPI WFME_ExpressionAPC(ULONG_PTR dwData) {

   // This function intentionally left blank
}


//////////////////////////////////////////////////////////////////////////////


// Function to wait on multiple Boolean expressions
DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
   CONST HANDLE* phExpObjects, DWORD dwMilliseconds) {

   // Allocate a temporary array because we modify the passed array and
   // we need to add a handle at the end for the hsemOnlyOne semaphore.
   PHANDLE phExpObjectsTemp = (PHANDLE)
      _alloca(sizeof(HANDLE) * (nExpObjects + 1));
   CopyMemory(phExpObjectsTemp, phExpObjects, sizeof(HANDLE) * nExpObjects);
   phExpObjectsTemp[nExpObjects] = NULL;  // Put sentinel at end

   // Semaphore to guarantee that only one expression gets satisfied
   HANDLE hsemOnlyOne = CreateSemaphore(NULL, 1, 1, NULL);

   // Expression information: 1 per possible thread
   EXPRESSION Expression[MAXIMUM_WAIT_OBJECTS];

   DWORD dwExpNum  = 0;    // Current expression number
   DWORD dwNumExps = 0;    // Total number of expressions

   DWORD dwObjBegin = 0;   // First index of a set
   DWORD dwObjCur   = 0;   // Current index of object in a set

   DWORD dwThreadId, dwWaitRet = 0;

   // Array of thread handles for threads: 1 per expression
   HANDLE ahThreads[MAXIMUM_WAIT_OBJECTS];

   // Parse the callers handle list by initializing a structure for
   // each expression and adding hsemOnlyOne to each expression.
   while ((dwWaitRet != WAIT_FAILED) && (dwObjCur <= nExpObjects)) {

      // While no errors, and object handles are in the caller's list...
   // Find next expression (OR-expressions are separated by NULL handles)
   while (phExpObjectsTemp[dwObjCur] != NULL)
      dwObjCur++;
```

China-pub.com 下载

```
      // Initialize Expression structure which an OR-thread waits on
      phExpObjectsTemp[dwObjCur] = hsemOnlyOne;
      Expression[dwNumExps].m_phExpObjects =  &phExpObjectsTemp[dwObjBegin];
      Expression[dwNumExps].m_nExpObjects  =  dwObjCur - dwObjBegin + 1;

      if (Expression[dwNumExps].m_nExpObjects > MAXIMUM_WAIT_OBJECTS) {
         // Error: Too many handles in single expression
         dwWaitRet = WAIT_FAILED;
         SetLastError(ERROR_SECRET_TOO_LONG);
      }

      // Advance to the next expression
      dwObjBegin = ++dwObjCur;
      if (++dwNumExps == MAXIMUM_WAIT_OBJECTS) {
         // Error: Too many expressions
         dwWaitRet = WAIT_FAILED;
         SetLastError(ERROR_TOO_MANY_SECRETS);
      }
   }

   if (dwWaitRet != WAIT_FAILED) {

      // No errors occurred while parsing the handle list

      // Spawn thread to wait on each expression
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

         ahThreads[dwExpNum] = chBEGINTHREADEX(NULL,
            1, // We only require a small stack
            WFME_ThreadExpression, &Expression[dwExpNum],
            0, &dwThreadId);
      }

      // Wait for an expression to come TRUE or for a timeout
      dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
         FALSE, dwMilliseconds);

      if (WAIT_TIMEOUT == dwWaitRet) {

         // We timed-out, check if any expressions were satisfied by
         // checking the state of the hsemOnlyOne semaphore.
         dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

         if (WAIT_TIMEOUT == dwWaitRet) {

            // If the semaphore was not signaled, some thread expressions
            // was satisfied; we need to determine which expression.
            dwWaitRet = WaitForMultipleObjects(dwExpNum,
               ahThreads, FALSE, INFINITE);

         } else {

            // No expression was satisfied and WaitForSingleObject just gave
            // us the semaphore so we know that no expression can ever be
            // satisfied now -- waiting for an expression has timed-out.
```

China-pub.com
下载


```
            dwWaitRet = WAIT_TIMEOUT;
         }
      }

      // Break all the waiting expression threads out of their
      // wait state so that they can terminate cleanly.
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

         if ((WAIT_TIMEOUT == dwWaitRet) ||
             (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {

            QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
         }
      }

#ifdef _DEBUG
      // In debug builds, wait for all of expression threads to terminate
      // to make sure that we are forcing the threads to wake up.
      // In non-debug builds, we'll assume that this works and
      // not keep this thread waiting any longer.
      WaitForMultipleObjects(dwExpNum, ahThreads, TRUE, INFINITE);
#endif

      // Close our handles to all the expression threads
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {
         CloseHandle(ahThreads[dwExpNum]);
      }
   }  // error occurred while parsing
      CloseHandle(hsemOnlyOne);
      return(dwWaitRet);
   }


   ////////////////////////////////// End of File //////////////////////////////////////
```

WaitForMultExp.h

```
   /******************************************************************************
   Module:  WaitForMultExp.h
   Notices: Copyright (c) 2000 Jeffrey Richter
   ******************************************************************************/


   #pragma once


   ////////////////////////////////////////////////////////////////////////////////


   DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
      CONST HANDLE* phExpObjects, DWORD dwMilliseconds);


   ////////////////////////////////// End of File //////////////////////////////////////
```

China-pub.com 下载

WfMETest.cpp

```
/******************************************************************************
Module: WfMETest.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>
#include "resource.h"
#include "WaitForMultExp.h"
//////////////////////////////////////////////////////////////////////////////


// g_ahObjs contains the list of event kernel object
// handles referenced in the Boolean expression.
#define MAX_KERNEL_OBJS     1000
HANDLE g_ahObjs[MAX_KERNEL_OBJS];


// ahExpObjs contains all the expressions. A single expression
// consists of a contiguous set of kernel object handles that
// is TRUE when all the objects are signaled at the same time.
// A NULL handle is used to separate OR expressions.

// A handle value may NOT appear multiple times within an AND
// expression but the same handle value may appear in
// different OR expressions.

// An expression can have a maximum of 64 sets with no more
// than 63 handles/set plus a NULL handle to separate each set
#define MAX_EXPRESSION_SIZE   ((64 * 63) + 63)


// m_nExpObjects is the number of entries used in the ahExpObjects array.
typedef struct {
   HWND     m_hwnd;                              // Where to send results
   DWORD    m_dwMilliseconds;                    // How long before timeout
   DWORD    m_nExpObjects;                       // # of entries in object list
   HANDLE   m_ahExpObjs[MAX_EXPRESSION_SIZE];    // List of objs
} AWFME, *PAWFME;
AWFME g_awfme;


// This message is posted to the UI thread when an expression
// comes true or when we timeout while waiting for an
// expression to come TRUE.
#define WM_WAITEND      (WM_USER + 101)


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
   BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
   chSETDLGICONS(hwnd, IDI_WFMETEXT);

   // Initialize the controls in the dialog box
   SetDlgItemInt(hwnd,  IDC_NUMOBJS, 4, FALSE);
   SetDlgItemInt(hwnd,  IDC_TIMEOUT, 30000, FALSE);
   SetDlgItemText(hwnd, IDC_EXPRESSION,
      _T("1 2 | 2 3 4 | 1 4"));

   // Set the multicolumn listbox's column size
   ListBox_SetColumnWidth(GetDlgItem(hwnd, IDC_OBJLIST),
      LOWORD(GetDialogBaseUnits()) * 4);

   return(TRUE);  // Accept default focus window.
}


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI AsyncWaitForMultipleExpressions(PVOID pvParam) {

   PAWFME pawfme = (PAWFME) pvParam;

   DWORD dw = WaitForMultipleExpressions(pawfme->m_nExpObjects,
      pawfme->m_ahExpObjs, pawfme->m_dwMilliseconds);
   PostMessage(pawfme->m_hwnd, WM_WAITEND, dw, 0);
   return(0);
}


//////////////////////////////////////////////////////////////////////////////


LRESULT Dlg_OnWaitEnd(HWND hwnd, WPARAM wParam, LPARAM lParam) {

   // Close all the event kernel object handles
   for (int n = 0; g_ahObjs[n] != NULL; n++)
      CloseHandle(g_ahObjs[n]);

   // Tell the user the result of running the test
   if (wParam == WAIT_TIMEOUT)
      SetDlgItemText(hwnd, IDC_RESULT, __TEXT("Timeout"));
   else
      SetDlgItemInt(hwnd, IDC_RESULT, (DWORD) wParam - WAIT_OBJECT_0, FALSE);
   // Allow the user to change values and run the test again
   EnableWindow(GetDlgItem(hwnd, IDC_NUMOBJS),    TRUE);
   EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT),    TRUE);
   EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), TRUE);
   EnableWindow(GetDlgItem(hwnd, IDOK),           TRUE);
   SetFocus(GetDlgItem(hwnd, IDC_EXPRESSION));

   return(0);
}
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   // Obtain the user's settings from the dialog box controls.
   TCHAR szExpression[100];
   ComboBox_GetText(GetDlgItem(hwnd, IDC_EXPRESSION), szExpression,
      sizeof(szExpression) / sizeof(szExpression[0]));

   int nObjects = GetDlgItemInt(hwnd, IDC_NUMOBJS, NULL, FALSE);

   switch (id) {
   case IDCANCEL:
      EndDialog(hwnd, id);
      break;

   case IDC_OBJLIST:
      switch (codeNotify) {
      case LBN_SELCHANGE:
         // An item changed state, reset all items and set the selected ones.
         for (int n = 0; n < nObjects; n++)
            ResetEvent(g_ahObjs[n]);

         for (n = 0; n < nObjects; n++) {
            if (ListBox_GetSel(GetDlgItem(hwnd, IDC_OBJLIST), n))
               SetEvent(g_ahObjs[n]);
         }
         break;
      }
      break;
     case IDOK:
        // Prevent the user from changing values while the test is running
        SetFocus(GetDlgItem(hwnd, IDC_OBJLIST));
        EnableWindow(GetDlgItem(hwnd, IDC_NUMOBJS),    FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT),    FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDOK),           FALSE);

        // Notify the user that the test is running
        SetDlgItemText(hwnd, IDC_RESULT, TEXT("Waiting..."));

        // Create all of the desired kernel objects
        ZeroMemory(g_ahObjs, sizeof(g_ahObjs));
        g_awfme.m_nExpObjects = 0;
        ZeroMemory(g_awfme.m_ahExpObjs, sizeof(g_awfme.m_ahExpObjs));
        g_awfme.m_hwnd = hwnd;
        g_awfme.m_dwMilliseconds = GetDlgItemInt(hwnd, IDC_TIMEOUT, NULL, FALSE);

        ListBox_ResetContent(GetDlgItem(hwnd, IDC_OBJLIST));
        for (int n = 0; n < nObjects; n++) {
           TCHAR szBuf[20];
           g_ahObjs[n] = CreateEvent(NULL, FALSE, FALSE, NULL);

           wsprintf(szBuf, TEXT("  %d"), n + 1);
```

China-pub.com
下载


```
            ListBox_AddString(GetDlgItem(hwnd, IDC_OBJLIST),
               &szBuf[lstrlen(szBuf) - 3]);
         }

         PTSTR p = _tcstok(szExpression, TEXT(" "));
         while (p != NULL) {
            g_awfme.m_ahExpObjs[g_awfme.m_nExpObjects++] =
               (*p == TEXT('|')) ? NULL : g_ahObjs[_ttoi(p) - 1];
            p = _tcstok(NULL, TEXT(" "));
         }

         DWORD dwThreadId;
         CloseHandle(chBEGINTHREADEX(NULL, 0,
            AsyncWaitForMultipleExpressions, &g_awfme,
            0, &dwThreadId));
         break;
      }
   }

   ///////////////////////////////////////////////////////////////////////////
   INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

      switch (uMsg) {
         chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
         chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);

         case WM_WAITEND:
            return(Dlg_OnWaitEnd(hwnd, wParam, lParam));
      }

      return(FALSE);
   }


   ///////////////////////////////////////////////////////////////////////////


   int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

      DialogBox(hinstExe, MAKEINTRESOURCE(IDD_TESTW4ME), NULL, Dlg_Proc);
      return(0);
   }


   ////////////////////////////////// End of File /////////////////////////////////////
```

WfMETest.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
```

China-pub.com 下载

```
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_WFMETEST DIALOGEX 0, 0, 168, 185
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "WaitForMultipleExpressions"
FONT 8, "MS Sans Serif", 0, 0, 0x1
BEGIN
    LTEXT           "How many different &kernel objects are referenced in the expression:"
                    IDC_STATIC,3,4,121,17
    EDITTEXT        IDC_NUMOBJS,138,6,27,14,ES_AUTOHSCROLL
    LTEXT           "&Timeout (in milliseconds):",IDC_STATIC,4,28,83,8
    EDITTEXT        IDC_TIMEOUT,138,26,27,14,ES_AUTOHSCROLL
    LTEXT           "&Expression (use space for AND and | for OR):",
                    IDC_STATIC,4,44,143,8
    COMBOBOX        IDC_EXPRESSION,4,56,160,76,CBS_DROPDOWN | WS_VSCROLL |
                    WS_TABSTOP
    DEFPUSHBUTTON   "&Wait for Multiple Expressions",IDOK,34,72,99,14
    LTEXT           "&Signal which kernel objects:",IDC_STATIC,4,92,83,8
    LISTBOX         IDC_OBJLIST,4,102,160,68,LBS_MULTIPLESEL |
                    LBS_NOINTEGRALHEIGHT | LBS_MULTICOLUMN | WS_VSCROLL |
                    WS_HSCROLL | WS_TABSTOP
    LTEXT           "Expression satisfied:",IDC_STATIC,32,172,63,8
    LTEXT           "Timeout",IDC_RESULT,100,172,36,8
END

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

China-pub.com
下载


```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_WFMETEXT            ICON    DISCARDABLE     "WaitForMultExp.ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第11章　线程池的使用

第8章讲述了如何使用让线程保持用户方式的机制来实现线程同步的方法。用户方式的同步机制的出色之处在于它的同步速度很快。如果关心线程的运行速度，那么应该了解一下用户方式的同步机制是否适用。

到目前为止，已经知道创建多线程应用程序是非常困难的。需要会面临两个大问题。一个是要对线程的创建和撤消进行管理，另一个是要对线程对资源的访问实施同步。为了对资源访问实施同步，Windows提供了许多基本要素来帮助进行操作，如事件、信标、互斥对象和关键代码段等。这些基本要素的使用都非常方便。为了使操作变得更加方便，唯一的方法是让系统能够自动保护共享资源。不幸的是，在Windows提供一种让人满意的保护方法之前，我们已经有了一种这样的方法。

在如何对线程的创建和撤消进行管理的问题上，人人都有自己的好主意。近年来，我自己创建了若干不同的线程池实现代码，每个实现代码都进行了很好的调整，以便适应特定环境的需要。Microsoft公司的Windows 2000提供了一些新的线程池函数，使得线程的创建、撤消和基本管理变得更加容易。这个新的通用线程池并不完全适合每一种环境，但是它常常可以适合你的需要，并且能够节省大量的程序开发时间。

新的线程池函数使你能够执行下列操作：

- 异步调用函数。
- 按照规定的时间间隔调用函数。
- 当单个内核对象变为已通知状态时调用函数。
- 当异步I/O请求完成时调用函数。

为了完成这些操作，线程池由4个独立的部分组成。表11-1显示了这些组件并描述了控制其行为特性的规则。

表11-1　线程池的组件及其行为特性

	组件			
	定时器	等待	I/O	非I/O
线程的初始数值	总是1	0	0	0
当创建一个线程时	当调用第一个线程池函数时	每63个注册对象有一个线程	系统使用试探法，但是这里有一些因素会影响线程的创建： • 自从添加线程后已经过去一定的时间（以秒计算） • 使用WT_EXECUTELONGFUNCTION标志 • 已经排队的工作项目的数量超过了某个阈值	
当线程被撤消时	当进程终止运行时	当已经注册的等待对象数量是0时	当线程没有未处理的I/O请求并且已经空闲了一个阈值周期(约1ms)时	当线程空闲了一个阈值周期（约1ms)时
线程如何等待	待命状态	WaitForMultipleObjects	待命状态	GetQueued-Completion-Status

China-pub.com
下载

(续)

	组件			
	定时器	等待	I/O	非I/O
是什么唤醒了线程	等待定时器通知排队的用户APC	内核对象变为已通知状态	排队的用户APC和已完成的I/O请求	展示已完成的状态和I/O请示（完成端口最多允许数量为2*的CPU线程同时运行的数量）

当进程初始化时，它并不产生与这些组件相关联的任何开销。但是，一旦新线程池函数之一被调用时，就为进程创建某些组件，并且其中有些组件将被保留，直到进程终止运行为止。如你所见，使用线程池所产生的开销并不小。相当多的线程和内部数据结构变成了你的进程的一个组成部分。因此必须认真考虑线程池能够为你做什么和不能做什么，不要盲目地使用这些函数。

好了，上述说明已经足够了。下面让我们来看一看这些函数能够做些什么。

11.1　方案1：异步调用函数

假设有一个服务器进程，该进程有一个主线程，正在等待客户机的请求。当主线程收到该请求时，它就产生一个专门的线程，以便处理该请求。这使得应用程序的主线程循环运行，并等待另一个客户机的请求。这个方案是客户机/服务器应用程序的典型实现方法。虽然它的实现方法非常明确，但是也可以使用新线程池函数来实现它。

当服务器进程的主线程收到客户机的请求时，它可以调用下面这个函数：

```
BOOL QueueUserWorkItem(
   PTHREAD_START_ROUTINE pfnCallback,
   PVOID pvContext,
   ULONG dwFlags);
```

该函数将一个“工作项目”排队放入线程池中的一个线程中并且立即返回。所谓工作项目是指一个（用pfnCallback参数标识的）函数，它被调用并传递单个参数pvContext。最后，线程池中的某个线程将处理该工作项目，导致函数被调用。所编的回调函数必须采用下面的原型：

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

尽管必须使这个函数的原型返回DWORD，但是它的返回值实际上被忽略了。

注意，你自己从来不调用CreateThread。系统会自动为你的进程创建一个线程池，线程池中的一个线程将调用你的函数。另外，当该线程处理完客户机的请求之后，该线程并不立即被撤消。它要返回线程池，这样它就可以准备处理已经排队的任何其他工作项目。你的应用程序的运行效率可能会变得更高，因为不必为每个客户机请求创建和撤消线程。另外，由于线程与完成端口相关联，因此可以同时运行的线程数量限制为CPU数量的两倍。这就减少了线程的上下文转移的开销。

该函数的内部运行情况是，QueueUserWorkItem检查非I/O组件中的线程数量，然后根据负荷量（已排队的工作项目的数量）将另一个线程添加给该组件。接着QueueUserWorkItem执行对PostQueuedCompletionStatus的等价调用，将工作项目的信息传递给I/O完成端口。最后，在完成端口上等待的线程取出信息（通过调用GetQueuedCompletionStatus），并调用函数。当函

China-pub.com 下载

数返回时，该线程再次调用GetQueuedCompletionStatus，以便等待另一个工作项目。

线程池希望经常处理异步I/O请求，即每当线程将一个I/O请求排队放入设备驱动程序时，便要处理异步I/O请求。当设备驱动程序执行该I/O时，请求排队的线程并没有中断运行，而是继续执行其他指令。异步I/O是创建高性能可伸缩的应用程序的秘诀，因为它允许单个线程处理来自不同客户机的请求。该线程不必顺序处理这些请求，也不必在等待I/O请求运行结束时中断运行。

但是，Windows对异步I/O请求规定了一个限制，即如果线程将一个异步I/O请求发送给设备驱动程序，然后终止运行，那么该I/O请求就会丢失，并且在I/O请求运行结束时，没有线程得到这个通知。在设计良好的线程池中，线程的数量可以根据客户机的需要而增减。因此，如果线程发出一个异步I/O请求，然后因为线程池缩小而终止运行，那么该I/O请求也会被撤消。因为这种情况实际上并不是你想要的，所以你需要一个解决方案。

如果你想要给发出异步I/O请求的工作项目排队，不能将该工作项目插入线程池的非I/O组件中。必须将该工作项目放入线程池的I/O组件中进行排队。该I/O组件由一组线程组成，如果这组线程还有尚未处理的I/O请求，那么它们决不能终止运行。因此你只能将它们用来运行发出异步I/O请求的代码。

若要为I/O组件的工作项目进行排队，仍然必须调用QueueUserWorkItem函数，但是可以为dwFlags参数传递WT_EXECUTEINIOTHREAD。通常只需传递WT_EXECUTEDEFAULT（定义为0），这使得工作项目可以放入非I/O组件的线程中。

Windows提供的函数（如RegNotifyChangeKeyValue）能够异步执行与非I/O相关的任务。这些函数也要求调用线程不能终止运行。如果想使用永久线程池的线程来调用这些函数中的一个，可以使用WT_EXECUTEINPERSISTENTTHREAD标志，它使定时器组件的线程能够执行已排队的工作项目回调函数。由于定时器组件的线程决不会终止运行，因此可以确保最终发生异步操作。应该保证回调函数不会中断，并且保证它能迅速执行，这样，定时器组件的线程就不会受到不利的影响。

设计良好的线程池也必须设法保证线程始终都能处理各个请求。如果线程池包含4个线程，并且有100个工作项目已经排队，每次只能处理4个工作项目。如果一个工作项目只需要几个毫秒来运行，那么这是不成问题的。但是，如果工作项目需要运行长得多的时间，那么将无法及时处理这些请求。

当然，系统无法很好地预料工作项目函数将要进行什么操作，但是，如果知道工作项目需要花费很长的时间来运行，那么可以调用QueueUserWorkItem函数，为它传递WT_EXECUTELONGFUNCTION标志。该标志能够帮助线程池决定是否要将新线程添加给线程池。如果线程池中的所有线程都处于繁忙状态，它就会强制线程池创建一个新线程。因此，如果同时对10 000个工作项目进行了排队（使用WT_EXECUTELONGFUNCTION标志），那么这10 000个线程就被添加给该线程池。如果不想创建10 000个线程，必须分开调用QueueUserWorkItem函数，这样某些工作项目就有机会完成运行。

线程池不能对线程池中的线程数量规定一个上限，否则就会发生渴求或死锁现象。假如有10 000个排队的工作项目，当第10 001个项目通知一个事件时，这些工作项目将全部中断运行。如果你已经设置的最大数量为10 000个线程，第10 001个工作项目没有被执行，那么所有的10 000个线程将永远被中断运行。

当使用线程池函数时，应该查找潜在的死锁条件。当然，如果工作项目函数在关键代码段、信标和互斥对象上中断运行，那么必须十分小心，因为这更有可能产生死锁现象。始终都应该

China-pub.com
下载

了解哪个组件（I/O、非I/O、等待或定时器等）的线程正在运行你的代码。另外，如果工作项目函数位于可能被动态卸载的DLL中，也要小心。调用已卸载的DLL中的函数的线程将会产生违规访问。若要确保不卸载带有已经排队的工作项目的DLL，必须对已排队工作项目进行引用计数，在调用QueueUserWorkItem函数之前递增计数器的值，当工作项目函数完成运行时则递减该计数器的值。只有当引用计数降为0时，才能安全地卸载DLL。

11.2 方案2：按规定的时间间隔调用函数

有时应用程序需要在某些时间执行操作任务。Windows提供了一个等待定时器内核对象，因此可以方便地获得基于时间的通知。许多程序员为应用程序执行的每个基于时间的操作任务创建了一个等待定时器对象，但是这是不必要的，会浪费系统资源。相反，可以创建一个等待定时器，将它设置为下一个预定运行的时间，然后为下一个时间重置定时器，如此类推。然而，要编写这样的代码非常困难，不过可以让新线程池函数对此进行管理。

若要调度在某个时间运行的工作项目，首先要调用下面的函数，创建一个定时器队列：

```
HANDLE CreateTimerQueue();
```

定时器队列对一组定时器进行组织安排。例如，有一个可执行文件控制着若干个服务程序。每个服务程序需要触发定时器，以帮助保持它的状态，比如客户机何时不再作出响应，何时收集和更新某些统计信息等。让每个服务程序占用一个等待定时器和专用线程，这是不经济的。相反，每个服务程序可以拥有它自己的定时器队列（这是个轻便的资源），并且共享定时器组件的线程和等待定时器对象。当一个服务程序终止运行时，它只需要删除它的定时器队列即可，因为这会删除该队列创建的所有定时器。

一旦拥有一个定时器队列，就可以在该队列中创建下面的定时器：

```
BOOL CreateTimerQueueTimer(
   PHANDLE phNewTimer,
   HANDLE hTimerQueue,
   WAITORTIMERCALLBACK pfnCallback,
   PVOID pvContext,
   DWORD dwDueTime,
   DWORD dwPeriod,
   ULONG dwFlags);
```

对于第二个参数，可以传递想要在其中创建定时器的定时器队列的句柄。如果只是创建少数几个定时器，只需要为hTimerQueue参数传递NULL，并且完全避免调用CreateTimerQueue函数。传递NULL，会告诉该函数使用默认的定时器队列，并且简化了你的代码。pfnCallback和pvContext参数用于指明应该调用什么函数以及到了规定的时间应该将什么传递给该函数。dwDueTime参数用于指明应该经过多少毫秒才能第一次调用该函数（如果这个值是0，那么只要可能，就调用该函数，使得CreateTimerQueueTimer函数类似QueueUserWorkItem）。dwPeriod参数用于指明应该经过多少毫秒才能在将来调用该函数。如果为dwPeriod传递0，那么就使它成为一个单步定时器，使工作项目只能进行一次排队。新定时器的句柄通过函数的phNewTimer参数返回。

工作回调函数必须采用下面的原型：

```
VOID WINAPI WaitOrTimerCallback(
   PVOID pvContext,
   BOOL fTimerOrWaitFired);
```

China-pub.com 下载

当该函数被调用时，fTimerOrWaitFired参数总是TRUE，表示该定时器已经触发。

下面介绍CreateTimerQueueTimer的dwFlags参数。该参数负责告诉函数，当到了规定的时间时，如何给工作项目进行排队。如果想要让非 I/O组件的线程来处理工作项目，可以使用WT_EXECUTEDEFAULT。如果想要在某个时间发出一个异步 I/O请求，可以使用WT_EXECUTEINIOTHREAD。如果想要让一个决不会终止运行的线程来处理该工作项目，可以使用WT_EXECUTEPERSISTENTTHREAD。如果认为工作项目需要很长的时间来运行，可以使用WT_EXECUTELONGFUNCTION。

也可以使用另一个标志，即WT_EXECUTEINTIMERTHREAD，下面将介绍它。在表11-1中，能够看到线程池有一个定时器组件。该组件能够创建单个定时器内核对象，并且能够管理它的到期时间。该组件总是由单个线程组成。当调用 CreateTimerQueueTimer函数时，可以使定时器组件的线程醒来，将你的定时器添加给一个定时器队列，并重置等待定时器内核对象。然后该定时器组件的线程便进入待命睡眠状态，等待该等待定时器将一个 APC放入它的队列。当等待定时器将该APC放入队列后，线程就醒来，更新定时器队列，重置等待定时器，然后决定对现在应该运行的工作项目执行什么操作。

接着，该线程要检查下面这些标志：WT_EXECUTEDEFAULT、WT_EXECUTEINIOTHREAD、WT_EXECUTEINPERSISTENTTHREAD、WT_EXECUTELONGFUNCTION和WT_EXECUTEINTIMERTHREAD。不过现在可以清楚地看到WT_EXECUTEDINTIMERTHREAD标志执行的是什么操作：它使定时器组件的线程能够执行该工作项目。虽然这使工作项目的运行效率更高，但是这非常危险。如果工作项目函数长时间中断运行，那么等待定时器的线程就无法执行任何其他操作。虽然等待定时器可能仍然将 APC项目排队放入该线程，但是在当前运行的函数返回之前，这些工作项目不会得到处理。如果打算使用定时器线程来执行代码，那么该代码应该迅速执行，不应该中断。

WT_EXECUTEINIOTHREAD、WT_EXECUTEINPERSISTENTTHREAD和WT_EXECUTEINTIMERTHREAD等标志是互斥的。如果不传递这些标志中的任何一个（或者使用WT_EXECUTEDEFAULT标志），那么工作项目就排队放入I/O组件的线程中。另外，如果设定了WT_EXECUTEINTIMERTHREAD标志，那么WT_EXECUTELONGFUNCTION将被忽略。

当不再想要触发定时器时，必须通过调用下面的函数将它删除：

```
BOOL DeleteTimerQueueTimer(
   HANDLE hTimerQueue,
   HANDLE hTimer,
   HANDLE hCompletionEvent);
```

即使对于已经触发的单步定时器，也必须调用该函数。 hTimerQueue参数指明定时器位于哪个队列中。hTimer参数指明要删除的定时器，句柄通过较早时调用CreateTimerQueueTimer来返回。

最后一个参数hCompletionEvent告诉你，由于该定时器，什么时候将不再存在没有处理的已排队的工作项目。如果为该参数传递 INVALID_HANDLE_VALUE，那么在该定时器的所有已排队工作项目完成运行之前， DeleteTimerQueueTimer函数不会返回。请想一想这将意味着什么。如果在定时器处理自己的工作项目期间对定时器进行一次中断删除，就会造成一个死锁条件。虽然你正在等待工作项目完成处理操作，但是你在等待它完成操作时却中断了它的处理。只有当线程不是处理定时器的工作项目的线程时，该线程才能进行对定时器的中断删除。

另外，如果你正在使用定时器组件的线程，不应该试图对任何定时器进行中断删除，否则

China-pub.com
下载

就会产生死锁。如果试图删除一个定时器，就会将一个 APC通知放入该定时器组件的线程队列中。如果该线程正在等待一个定时器被删除，而它不能删除该定时器，那么就会发生死锁。

如果不为hCompletionEvent参数传递INVALID_HANDLE_VALUE，可以传递NULL。这将告诉该函数，你想尽快删除定时器。在这种情况下， DeleteTimerQueueTimer将立即返回，但是你不知道该定时器的所有工作项目何时完成处理。最后，你可以传递一个事件内核对象的句柄作为hCompletionEvent的参数。当这样操作时， DeleteTimerQueueTimer将立即返回，同时，当定时器的所有已经排队的工作项目完成运行之后，定时器组件的线程将设置该事件。在调用DeleteTimerQueueTimer之前，千万不要给该事件发送通知，否则你的代码将认为排队的工作项目已经完成运行，但是实际上它们并没有完成。

一旦创建了一个定时器，可以调用下面这个函数来改变它的到期时间和到期周期：

```
BOOL ChangeTimerQueueTimer(
   HANDLE hTimerQueue,
   HANDLE hTimer,
   ULONG dwDueTime,
   ULONG dwPeriod);
```

这里传递了定时器队列的句柄和想要修改的现有定时器的句柄。可以修改定时器的dwDueTime和dwPeriod。注意，试图修改已经触发的单步定时器是不起作用的。另外，你可以随意调用该函数，而不必担心死锁。

当不再需要一组定时器时，可以调用下面这个函数，删除定时器队列：

```
BOOL DeleteTimerQueueEx(
   HANDLE hTimerQueue,
   HANDLE hCompletionEvent);
```

该函数取出一个现有的定时器队列的句柄，并删除它里面的所有定时器，这样就不必为删除每个定时器而显式调用DeleteTimerQueueTimer。hCompletionEvent参数在这里的语义与它在DeleteTimerQueueTimer函数中的语义是相同的。这意味着它存在同样的死锁可能性，因此必须小心。

在开始介绍另一个方案之前，让我们说明两个其他的项目。首先，线程池的定时器组件创建等待定时器，这样，它就可以给 APC项目排队，而不是给对象发送通知。这意味着操作系统能够连续给APC项目排队，并且定时器事件从来不会丢失。因此，设置一个定期定时器能够保证每个间隔时间都能为你的工作项目排队。如果创建一个定期定时器，每隔 10s触发一次，那么每隔10s就调用你的回调函数。必须注意这在使用多线程时也会发生必须对工作项目函数的各个部分实施同步。

如果不喜欢这种行为特性，而希望你的工作项目在每个项目执行之后的 10s进行排队，那么应该在工作项目函数的结尾处创建单步定时器。或者可以创建一个带有高超时值的单个定时器，并在工作项目函数的结尾处调用ChangeTimerQueueTimer.

TimedMsgBox示例应用程序

清单11-1列出的TimedMsgBox应用程序（“11 TimedMsgBox.exe ”）显示了如何使用线程池的定时器函数来实现一个用户在规定时间内不作出响应时能自动关闭的消息框。该应用程序的源代码和资源文件位于本书所附光盘上的11-TimedMsgBox目录下。

当启动该程序时，它将全局变量g_nSecLeft设置为10。这表示用户必须在规定时间内对消息框作出响应的秒数。然后调用 CreateTimerQueueTimer函数，指令线程池每秒钟调用一次

China-pub.com 下载

MsgBoxTimeout函数。一旦一切都已初始化，便调用MessageBox，并向用户显示图11-1所示的消息框。

在等待用户作出响应的时候，线程池中的一个线程便调用MsgBoxTimeout函数。该函数寻找消息框的窗口句柄，对全局变量 g_nSecLeft进行递减，并更新消息框中的字符串。当MsgBoxTimeout第一次被调用后，消息框就类似下面的样子(见图11-2)。



图11-1 调用 Message Box 时出现的消息框



图11-2 调用MsgBox Timeout 后出现的消息框

当MsgBoxTimeout第10次被调用时，g_nSecLeft变量变为0，同时MsgBoxTimeout调用EndDialog函数来撤消该消息框。主线程调用的MessageBox返回，DeleteTimerQueueTimer被调用，以告诉线程池停止调用MsgBoxTimeout函数，这时出现图11-3所示的消息框，告诉用户他没有在分配给他的时间内对图11-1所示的消息框作出响应。

如果用户没有在时间到期之前作出响应，便出现图11-4所示的消息框。



图11-3 对图11-1所示的消息框不作出响应时出现的消息框



图11-4 在超时前不作出响应时出现的消息框

清单11-1 TimedMsgBox示例应用程序



TimedMsgBox.cpp

```
/******************************************************************************
Module:  TimedMsgBox.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


// The caption of our message box
TCHAR g_szCaption[] = TEXT("Timed Message Box");


// How many seconds we'll display the message box
```

China-pub.com
下载


```
int g_nSecLeft = 0;


// This is STATIC window control ID for a message box
#define ID_MSGBOX_STATIC_TEXT    0x0000ffff


//////////////////////////////////////////////////////////////////////////////


VOID WINAPI MsgBoxTimeout(PVOID pvContext, BOOLEAN fTimeout) {

   // NOTE: Due to a thread race condition, it is possible (but very unlikely)
   // that the message box will not be created when we get here.
   HWND hwnd = FindWindow(NULL, g_szCaption);

   if (hwnd != NULL) {
      // The window does exist; update the time remaining.
      TCHAR sz[100];
      wsprintf(sz, TEXT("You have %d seconds to respond"), g_nSecLeft--);
      SetDlgItemText(hwnd, ID_MSGBOX_STATIC_TEXT, sz);

      if (g_nSecLeft == 0) {
         // The time is up; force the message box to exit.
         EndDialog(hwnd, IDOK);
      }
   } else {

      // The window does not exist yet; do nothing this time.
      // We'll try again in another second.
   }
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   chWindows9xNotAllowed();

   // How many seconds we'll give the user to respond
   g_nSecLeft = 10;

   // Create a multishot 1-second timer that begins firing after 1 second.
   HANDLE hTimerQTimer;
   CreateTimerQueueTimer(&hTimerQTimer, NULL, MsgBoxTimeout, NULL,
      1000, 1000, 0);

   // Display the message box.
   MessageBox(NULL, TEXT("You have 10 seconds to respond"),
      g_szCaption, MB_OK);

   // Cancel the timer & delete the timer queue
   DeleteTimerQueueTimer(NULL, hTimerQTimer, NULL);
```

China-pub.com 下载

```
   // Let us know if the user responded or if we timed out.
   MessageBox(NULL,
      (g_nSecLeft == 0) ? TEXT("Timeout") : TEXT("User responded"),
         TEXT("Result"), MB_OK);

   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

TimedMsgBox.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_TIMEDMSGBOX         ICON    DISCARDABLE     "TimedMsgBox.ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

China-pub.com
下载


```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

11.3　方案3：当单个内核对象变为已通知状态时调用函数

Microsoft发现，许多应用程序产生的线程只是为了等待内核对象变为已通知状态。一旦对象得到通知，该线程就将某种通知移植到另一个线程，然后返回，等待该对象再次被通知。有些编程人员甚至编写了代码，在这种代码中，若干个线程各自等待一个对象。这对系统资源是个很大的浪费。当然，与创建进程相比，创建线程需要的的开销要小得多，但是线程是需要资源的。每个线程有一个堆栈，并且需要大量的CPU指令来创建和撤消线程。始终都应该尽量减少它使用的资源。

如果想在内核对象得到通知时注册一个要执行的工作项目，可以使用另一个新的线程池函数：

```
BOOL RegisterWaitForSingleObject(
   PHANDLE phNewWaitObject,
   HANDLE hObject,
   WAITORTIMERCALLBACK pfnCallback,
   PVOID pvContext,
   ULONG dwMilliseconds,
   ULONG dwFlags);
```

该函数负责将参数传送给线程池的等待组件。你告诉该组件，当内核对象（用 hObject进行标识）得到通知时，你想要对工作项目进行排队。也可以传递一个超时值，这样，即使内核对象没有变为已通知状态，也可以在规定的某个时间内对工作项目进行排队。超时值　0和

China-pub.com 下载

INFINITE是合法的。一般来说，该函数的运行情况与 WaitForSingleObject函数（第9章已经介绍）相似。当注册了一个等待组件后，该函数返回一个句柄（通过 phNewWaitObject参数）以标识该等待组件。

在内部，等待组件使用WaitForMultipleObjects函数来等待已经注册的对象，并且要受到该函数已经存在的任何限制的约束。限制之一是它不能多次等待单个句柄。因此，如果想要多次注册单个对象，必须调用DuplicateHandle函数，并对原始句柄和复制的句柄分开进行注册。当然，WaitForMultipleObjects能够等待已通知的对象中的任何一个，而不是所有的对象。如果熟悉WaitForMultipleObjects函数，那么一定知道它一次最多能够等待 64（MAXIMUM_WAIT_OBJECTS）个对象。如果用RegisterWaitForSingleObject函数注册的对象超过64个，那么将会出现什么情况呢？这时等待组件就会添加另一个也调用 WaitForMultipleObjects函数的线程。实际上，每隔63个对象后，就要将另一个线程添加给该组件，因为这些线程也必须等待负责控制超时的等待定时器对象。

当工作项目准备执行时，它被默认排队放入非 I/O组件的线程中。这些线程之一最终将会醒来，并且调用你的函数，该函数的原型必须是下面的形式：

```
VOID WINAPI WaitOrTimerCallbackFunc(
   PVOID pvContext,
   BOOLEAN fTimerOrWaitFired);
```

如果等待超时了，fTimerOrWaitFired参数的值是TRUE。如果等待时对象变为已通知状态，则该参数是FALSE。

对于RegisterWaitForSingleObject函数的dwFlags参数，可以传递 WT_EXECUTEINWAITTHREAD，它使等待组件的线程之一运行工作项目函数本身。它的运行速率更高，因为工作项目不必排队放入I/O组件中。但是这样做有一定的危险性，因为正在执行工作项目的等待组件函数的线程无法等待其他对象得到通知。只有当工作项目函数运行得很快时，才应该使用该标志。

如果工作项目将要发出异步I/O请求，或者使用从不终止运行的线程来执行某些操作，那么也可以传递WT_EXECUTEINIOTHREAD或者WT_EXECUTEINPERSISTENTTHREAD。也可以使用WT_EXECUTELONGFUNCTION标志来告诉线程池，你的函数可能要花费较长的时间来运行，而且它应该考虑将一个新线程添加给线程池。只有当工作项目正在被移植到非 I/O组件或I/O组件中时，才能使用该标志，如果使用等待组件的线程，不应该运行长函数。

应该了解的最后一个标志是WT_EXECUTEONLYONCE。假如你注册一个等待进程内核对象的组件，一旦该进程对象变为已通知状态，它就停留在这个状态中。这会导致等待组件连续地给工作项目排队。对于进程对象来说，可能不需要这个行为特性。如果使用 WT_EXECUTEONLYONCE标志，就可以防止出现这种情况，该标志将告诉等待组件在工作项目执行了一次后就停止等待该对象。

现在，如果正在等待一个自动重置的事件内核对象。一旦该对象变为已通知状态，该对象就重置为它的未通知状态，并且它的工作项目将被放入队列。这时，该对象仍然处于注册状态，同时，等待组件再次等待该对象被通知，或者等待超时（它已经重置）结束。当不再想让该等待组件等待你的注册对象时，必须取消它的注册状态。即使是使用 WT_EXECUTEONLYONCE标志注册的并且已经拥有队列的工作项目的等待组件，情况也是如此。调用下面这个函数，可以取消等待组件的注册状态：

```
BOOL UnregisterWaitEx(
```

China-pub.com
下载


```
    HANDLE hWaitHandle,
    HANDLE hCompletionEvent);
```

第一个参数指明一个注册的等待（由RegisterWaitForSingleObject返回），第二个参数指明当已注册的、正在等待的所有已排队的工作项目已经执行时，你希望如何通知你。与DeleteTimerQueueTimer函数一样，可以传递 NULL（如果不要通知的话），或者传递INVALID_HANDLE_VALUE（中断对函数的调用，直到所有排队的工作项目都已执行），也可以传递一个事件对象的句柄（当排队的工作项目已经执行时，它就会得到通知）。对于无中断的函数调用，如果没有排队的工作项目，那么UnregisterWaitEx返回TRUE，否则它返回FALSE，而GetLastError返回STATUS_PENDING。

同样，当你将INVALID_HANDLE_VALUE传递给UnregisterWaitEx时，必须小心避免死锁状态。在试图取消等待组件的注册状态，从而导致工作项目运行时，该工作项目函数不应该中断自己的运行。这好像是说：暂停我的运行，直到我完成运行为止一样——这会导致死锁。然而，如果等待组件的线程运行一个工作项目，而该工作项目取消了导致工作项目运行的等待组件的注册状态，UnregisterWaitEx是可以用来避免死锁的。还有一点需要说明，在取消等待组件的注册状态之前，不要关闭内核对象的句柄。这会使句柄无效，同时，等待组件的线程会在内部调用WaitForMultipleObjects函数，传递一个无效句柄。WaitForMultipleObjects的运行总是会立即失败，整个等待组件将无法正常工作。

最后，不应该调用PulseEvent函数来通知注册的事件对象。如果这样做了，等待组件的线程就可能忙于执行某些别的操作，从而错过了事件的触发。这不应该是个新问题了。PulseEvent几乎能够避免所有的线程结构产生这个问题。

11.4　方案4：当异步I/O请求完成运行时调用函数

最后一个方案是个常用的方案，即服务器应用程序发出某些异步 I/O请求，当这些请求完成时，需要让一个线程池准备好来处理已完成的 I/O请求。这个结构是I/O完成端口原先设计时所针对的一种结构。如果要管理自己的线程池，就要创建一个 I/O完成端口，并创建一个等待该端口的线程池。还需要打开多个I/O设备，将它们的句柄与完成端口关联起来。当异步 I/O请求完成时，设备驱动程序就将“工作项目”排队列入该完成端口。

这是一种非常出色的结构，它使少数线程能够有效地处理若干个工作项目，同时它又是一种很特殊的结构，因为线程池函数内置了这个结构，使你可以节省大量的设计和精力。若要利用这个结构，只需要打开设备，将它与线程池的非 I/O组件关联起来。记住，I/O组件的线程全部在一个I/O组件端口上等待。若要将一个设备与该组件关联起来，可以调用下面的函数：

```
BOOL BindIoCompletionCallback(
   HANDLE hDevice,
   POVERLAPPED_COMPLETION_ROUTINE pfnCallback,
   ULONG dwFlags);
```

该函数在内部调用 CreateIoCompletionPort，传递hDevice和内部完成端口的句柄。调用BindIoCompletionCallback也可以保证至少有一个线程始终在非 I/O组件中。与该设备相关联的完成关键字是重叠完成例程的地址。这样，当该设备的 I/O运行完成时，非I/O组件就知道要调用哪个函数，以便它能够处理已完成的I/O请求。该完成例程必须采用下面的原型：

```
VOID WINAPI OverlappedCompletionRoutine(
   DWORD dwErrorCode,
   DWORD dwNumberOfBytesTransferred,
```

China-pub.com 下载

```
POVERLAPPED pOverlapped);
```

你将会注意到没有将一个 OVERLAPPED结构传递给 BindIoCompletionCallback。OVERLAPPED结构被传递给ReadFile和WriteFile之类的函数。系统在内部始终保持对这个带有待处理I/O请求的重叠结构进行跟踪。当该请求完成时，系统将该结构的地址放入完成端口，从而使它能够被传递给你的OverlappedCompletionRoutine函数。另外，由于该完成例程的地址是完成的关键，因此，如果要将更多的上下文信息放入 OverlappedCompletionRoutine函数，应该使用将上下文信息放入OVERLAPPED结构的结尾处的传统方法。

还应该知道，关闭设备会导致它的所有待处理的 I/O请求立即完成，并产生一个错误代码。要作好准备，在你的回调函数中处理这种情况。如果关闭设备后你想确保没有运行任何回调函数，那么必须引用应用程序中的计数特性。换句话说，每次发出一个 I/O请求时，必须使计数器的计数递增，每次完成一个I/O请求，则递减计数器的计数。

目前没有特殊的标志可以传递给 BindIoCompletionCallback函数的dwFlags参数，因此必须传递0。相信你能够传递的标志是 WT_EXECUTEINIOTHREAD。如果一个I/O请求已经完成，它将被排队放入一个非I/O组件线程。在OverlappedCompletionRoutine函数中，可以发出另一个异步I/O请求。但是记住，如果发出I/O请求的线程终止运行，该I/O请求也会被撤消。另外，非I/O组件中的线程是根据工作量来创建或撤消的。如果工作量很小，该组件中的线程就会终止运行，其 I/O请求仍然处于未处理状态。如果 BindIoCompletionCallback函数支持WT_EXECUTEINIOTHREAD标志，那么在完成端口上等待的线程就会醒来，并将结果移植到一个I/O组件的线程中。由于在I/O请求处于未处理状态下时这些线程决不会终止运行，因此可以发出I/O请求而不必担心它们被撤消。

虽然WT_EXECUTEINIOTHREAD标志的作用不错，但是可以很容易模仿刚才介绍的行为特性。在 OverlappedCompletionRoutine函数中，只需要调用 QueueUserWorkItem，传递WT_EXECUTEINIOTHREAD标志和想要的任何数据（至少是重叠结构）。这就是线程池函数能够为你执行的全部功能。

China-pub.com
下载

第12章 纤 程

Microsoft公司给Windows添加了一种纤程，以便能够非常容易地将现有的 UNIX服务器应用程序移植到Windows中。UNIX服务器应用程序属于单线程应用程序（由 Windows定义），但是它能够为多个客户程序提供服务。换句话说， UNIX应用程序的开发人员已经创建了他们自己的线程结构库，他们能够使用这种线程结构库来仿真纯线程。该线程包能够创建多个堆栈，保存某些CPU寄存器，并且在它们之间进行切换，以便为客户机请求提供服务。

显然，若要取得最佳的性能，这些 UNIX应用程序必须重新设计，仿真的线程库应该用Windows提供的纯线程来替代。然而，这种重新设计需要花费数月甚至更长的时间才能完成，因此许多公司首先将它们现有的 UNIX代码移植到Windows中，这样就能够将某些应用软件推向Windows市场。

当你将UNIX代码移植到Windows中时，一些问题就会因此而产生。尤其是 Windows管理线程的内存栈的方法要比简单地分配内存复杂得多。 Windows内存栈开始时的物理存储器的容量比较小，然后根据需要逐步扩大。这个过程在第 16章“ 线程的堆栈 ”中详细介绍。由于结构化异常处理机制的原因，代码的移植就更加复杂了。

为了能够更快和更正确地将它们的代码移植到 Windows中，Microsoft公司在操作系统中添加了纤程。本章将要介绍纤程的概念、负责操作纤程的函数以及如何利用纤程的特性。要记住，如果有设计得更好的使用Windows自身线程的应用程序，那么应该避免使用纤程。

12.1 纤程的操作

首先要注意的一个问题是，实现线程的是Windows内核。操作系统清楚地知道线程的情况，并且根据Microsoft定义的算法对线程进行调度。纤程是以用户方式代码来实现的，内核并不知道纤程，并且它们是根据用户定义的算法来调度的。由于你定义了纤程的调度算法，因此，就内核而言，纤程采用非抢占式调度方式。

需要了解的下一个问题是，单线程可以包含一个或多个纤程。就内核而言，线程是抢占调度的，是正在执行的代码。然而，线程每次执行一个纤程的代码 —— 你决定究竟执行哪个纤程（随着我们讲解的深入，这些概念将会越来越清楚）。

当使用纤程时，你必须执行的第一步操作是将现有的线程转换成一个纤程。可以通过调用ConvertThreadToFiber函数来执行这项操作：

```
PVOID ConvertThreadToFiber(PVOID pvParam);
```

该函数为纤程的执行环境分配相应的内存（约为 200字节）。该执行环境由下列元素组成：

- 一个用户定义的值，它被初始化为传递给 ConvertThreadToFiber的pvParam参数的值。
- 结构化异常处理链的头。
- 纤程内存栈的最高和最低地址（当将线程转换成纤程时，这也是线程的内存栈）。
- CPU寄存器，包括堆栈指针、指令指针和其他。

当对纤程的执行环境进行分配和初始化后，就可以将执行环境的地址与线程关联起来。该线程被转换成一个纤程，而纤程则在该线程上运行。 ConvertThreadToFiber函数实际上返回纤程的执行环境的内存地址。虽然必须在晚些时候使用该地址，但是决不应该自己对该执行环境

China-pub.com 下载

数据进行读写操作，因为必要时纤程函数会为你对该结构的内容进行操作。现在，如果你的纤程（线程）返回或调用ExitThread函数，那么纤程和线程都会终止运行。

除非打算创建更多的纤程以便在同一个线程上运行，否则没有理由将线程转换成纤程。若要创建另一个纤程，该线程（当前正在运行纤程的线程）可以调用 CreateFiber函数：

```
PVOID CreateFiber(
   DWORD dwStackSize,
   PFIBER_START_ROUTINE pfnStartAddress,
   PVOID pvParam);
```

CreateFiber首先设法创建一个新内存栈，它的大小由 dwStackSize参数来指明。通常传递的参数是0，按照默认设置，它创建一个内存栈，其大小可以扩展为 1MB，不过开始时有两个存储器页面用于该内存栈。如果设定一个非0值，那么就用设定的大小来保存和使用内存栈。

接着，CreateFiber函数分配一个新的纤程执行环境结构，并对它进行初始化。该用户定义的值被设置为传递给 CreateFiber的pvParam参数的值，新内存栈的最高和最低地址被保存，同时，纤程函数的内存地址（作为pfnStartAddress参数来传递）也被保存。

PfnStartAddress参数用于设定必须实现的纤程例程的地址，它必须采用下面的原型：

```
VOID WINAPI FiberFunc(PVOID pvParam);
```

当纤程被初次调度时，该函数就开始运行，并且将原先传递给 CreateFiber的pvParam的值传递给它。可以在这个纤程函数中执行想执行的任何操作。但是该函数的原型规定返回值是VOID，这并不是因为返回值没有任何意义，而是因为该函数根本不应该返回。如果纤程确实返回了，那么线程和该线程创建的所有纤程将立即被撤消。

与ConvertThreadToFiber函数一样，CreateFiber函数也返回纤程运行环境的内存地址。但是，与ConvertThreadToFiber不同的是，这个新纤程并不执行，因为当前运行的纤程仍然在执行。在单个线程上，每次只能运行一个纤程。若要使新纤程能够运行，可以调用 Switch To Fiber函数：

```
VOID SwitchToFiber(PVOID pvFiberExecutionContext);
```

Switch To Fiber函数只有一个参数，即 pvFiberExecutionContext，它是上次调用ConvertThreadToFiber或CreateFiber函数时返回的纤程的执行环境的内存地址。该内存地址告诉该函数要对哪个纤程进行调度。SwitchToFiber函数在内部执行下列操作步骤：

1) 它负责将某些当前的CPU寄存器保存在当前运行的纤程执行环境中，包括指令指针寄存器和堆栈指针寄存器。

2) 它将上一次保存在即将运行的纤程的执行环境中的寄存器装入 CPU寄存器。这些寄存器包括堆栈指针寄存器。这样，当线程继续执行时，就可以使用该纤程的内存栈。

3) 它将纤程的执行环境与线程关联起来，线程运行特定的纤程。

4) 它将线程的指令指针设置为已保存的指令指针。线程（纤程）从该纤程上次执行的地方开始继续执行。

SwitchToFiber函数是纤程获得CPU时间的唯一途径。由于你的代码必须在相应的时间显式调用SwitchToFiber函数，因此你对纤程的调度可以实施全面的控制。记住，纤程的调度与线程调度毫不相干。纤程运行所依赖的线程始终都可以由操作系统终止其运行。当线程被调度时，当前选定的纤程开始运行，而其他纤程则不能运行，除非显式调用 SwitchToFiber函数。若要撤消纤程，可以调用DeleteFiber函数：

```
VOID DeleteFiber(PVOID pvFiberExecutionContext);
```

China-pub.com
下载

该函数用于删除pvFiberExecutionContext参数指明的纤程，当然这是纤程的执行环境的地址。该函数能够释放纤程栈使用的内存，然后撤消纤程的执行环境。但是，如果传递了当前与线程相关联的纤程地址，那么该函数就在内部调用 ExitThread函数，该线程及其创建的所有纤程全部被撤消。

DeleteFiber函数通常由一个纤程调用，以便删除另一个纤程。已经删除的纤程的内存栈将被撤消，纤程的执行环境被释放。注意，纤程与线程之间的差别在于，线程通常通过调用 ExitThread函数将自己撤消。实际上，用一个线程调用 TerminateThread函数来终止另一个线程的运行，是一种不好的方法。如果你确实调用了 TerminateThread函数，系统并不撤消已经终止运行的线程的内存栈。可以利用纤程的这种能力来删除另一个纤程，后面介绍示例应用程序时将说明这是如何实现的。

为了使操作更加方便，还可以使用另外两个纤程函数。一个线程每次可以执行一个纤程，操作系统始终都知道当前哪个纤程与该线程相关联。如果想要获得当前运行的纤程的执行环境的地址，可以调用GetCurrentFiber函数：

```
PVOID GetCurrentFiber();
```

另一个使用非常方便的函数是GetFiberData：

```
PVOID GetFiberData();
```

前面讲过，每个纤程的执行环境包含一个用户定义的值。这个值使用作为 ConvertThread ToFiber或CreateFiber的pvParam参数而传递的值进行初始化。该值也可以作为纤程函数的参数来传递。GetFiberData只是查看当前执行的纤程的执行环境，并返回保存的值。

无论GetCurrentFiber还是GetFiberData，运行速度都很快，并且通常是作为内蕴函数(infrinsic funcfion)来实现的，这意味着编译器能够为这些函数生成内联代码。

12.2 Counter示例应用程序

后面清单12-1中的Counter应用程序（“12 Counter.exe ”）使用纤程来实现后台处理。当运行该应用程序时，便出现图 12-1所示的对话框（建议通过运行该应用程序来了解阅读下面的内容时将会出现什么情况并观察它的行为特性）。

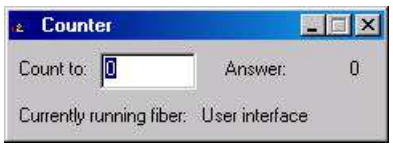


图12-1 Counter 对话框

可以将该应用程序视为包含两个单元格的超小型电子表格。第一个单元格是个可写入的单元格，它是作为编辑控件（标注为 Count To）来实现的。第二个单元格是个只读单元格，它是作为一个静态控件（标注为 Answer）来实现的。当改变编辑控件中的数字时，Answer单元格就会自动重新计算。对于这个简单的应用程序来说，重新计算是由计数器来进行的，它的起始数字是 0，然后慢慢递增，直到 Answer单元格中的值与输入的数字相同为止。为了演示的需要，对话框底部的静态控件中的数字不断更新，以显示当前执行的是哪个纤程。该纤程既可以是用户界面纤程，也可以是重新计算的纤程。

为了测试该应用程序的运行情况，可以将 5键入编辑控件。Currently Running Fiber（当前运行的纤程）域改为 Recalculation(重新计算)，Answer域中的数字慢慢地从 0递增为5。当计数完成时，Currently Running Fiber域重新改为User Interface（用户界面），线程转入睡眠状态。这时，在编辑框中，在5后面键入0（使数字变为50），然后观察计数从0 开始，逐步递增为50。不过这次在Answer域中的数字递增的同时，你移动屏幕上的窗口。你将会发现，重新计算的纤程暂停运行，而用户界面纤程重新被调度，使应用程序的用户界面保持对用户的响应状态。当

China-pub.com 下载

你停止移动窗口时，重新计算的纤程又被重新调度，而 Answer域则从上次暂停的地方继续开始计数。

最后一个测试项是，当重新计算纤程正在计数时，改变编辑框中的数字。同样，你会看到用户界面对你的输入作出响应，但是你也会看到当你停止键入时，重新计算纤程开始从头计数。这就是你在功能完善的电子表格应用程序中需要的那种行为特性。

记住，这个应用程序中没有使用任何关键代码段或其他线程同步对象，一切都是使用由两个纤程组成的单线程来完成的。

下面让我们来说明这个应用程序是如何实现的。当进程的主线程通过执行 _tWinMain（在程序清单的结尾处）开始运行时，ConvertThreadToFiber函数被调用，以便将线程转换成纤程，并且允许我们在以后创建另一个纤程。然后，创建一个无模式对话框，它是应用程序的主对话框，接着，一个状态变量被初始化，指明后台处理的状态（ BPS）。该状态变量是全局变量g_FiberInfo中包含的bps成员。如表12-1所示，共有3个状态。

表12-1 全局变量g_FiberInfo 中DPS成员的状态

状 态	描 述
BPS_DONE	重新计算即将完成，用户没有修改需要重新计算的任何东西
BPS_STARTOVER	用户修改了某些东西，因此需要从头开始重新计算
BPS_CONTINUE	重新计算已经开始，但是尚未完成。另外，用户没有修改需要从头开始重新计算的任何东西

后台处理的状态变量是在线程的消息循环中进行观察的，该消息循环比普通的消息循环更加复杂。下面是消息循环的作用：

- 如果存在一个窗口消息（用户界面处于活动状态）。那么它就负责处理该消息。保持用户界面的响应特性的优先级总是高于重新计算的优先级。
- 如果用户界面无事可做，它可以查看是否需要进行任何重新计算操作（后台处理状态是BPS_STARTOVER或BPS_CONTINUE）。
- 如果没有任何重新计算操作需要执行（ BPS_DONE），它就通过调用WaitMessage函数暂停线程的运行；只有用户界面事件能够导致需要进行的重新计算。

如果用户界面纤程无事可做，同时，用户刚刚修改了编辑框中的值，那么需要从头开始重新计算（BPS_STARTOVER）。首先必须了解，我们可能已经有一个重新计算纤程正在运行。如果是这种情况，必须删除该纤程，并且创建一个从头开始计数的新纤程。用户界面纤程调用DeleteFiber函数来撤消现有的重新计算纤程。这正是纤程（比线程更加）便于使用的一个表现。删除重新计算的纤程是完全没有问题的，该纤程的内存栈和执行环境将被完全彻底地释放。如果使用线程而不是纤程，那么用户界面线程将不会完全彻底地删除重新计算的线程，必须使用某种形式的线程之间的通信手段，并且等待重新计算的线程自行终止运行。一旦知道不再存在重新计算的纤程，就可以创建新的重新计算纤程，并将后台处理状态设置为 BPS_CONTINUE。

当用户界面空闲而重新计算的纤程有事可做时，可以通过调用 SwitchToFiber函数为它调度所需的时间。SwitchToFiber函数要等到重新计算的纤程再次调用 SwitchToFiber函数并传递用户界面纤程的执行环境的地址时才会返回。

FiberFunc函数包含了重新计算纤程执行的代码。该纤程函数将得到全局结构 g_FiberInfo的地址，因此它知道对话框窗口的句柄、用户界面纤程的执行环境的地址，以及当前后台处理的状态。该结构的地址不需要传递，因为它是位于一个全局变量之中，但是我想要展示如何将参数传递给纤程函数。此外，传递该地址可以较少地依赖代码，这总是一种好的做法。

China-pub.com
下载

纤程函数首先更新对话框中的状态控件，以便指明重新计算纤程正在运行。然后它要获取编辑框中的数字，并进入一个循环，从0开始计数，直到计算到编辑框中的这个数字。每当递增到接近这个数字的时候，便调用GetQueueStatus函数，以了解线程的消息队列中是否显示有任何消息（单个线程上运行的所有纤程均共享线程的消息队列）。当显示一条消息时，用户界面线程就有事可做了。由于我们希望它拥有高于重新计算的优先级，因此立即调用SwitchToFiber函数，使用户界面纤程能够处理该消息。当消息处理完毕后，用户界面纤程便重新调度该重新计算的纤程（如前面介绍的那样），同时，后台处理继续进行。

当没有消息需要处理时，重新计算纤程便更新对话框中的 Answer域的数字，然后睡眠200ms。在实际的代码中，应该删除对Sleep的调用，我在这里加上了对Sleep的调用，是为了强调展示进行重新计算所需要的时间。

当重新计算纤程完成Answer域数字的计算时，后台处理状态变量被设置为 BPS_DONE，同时，对SwitchToFiber函数的调用将对用户界面纤程进行重新调度。这时，如果用户界面纤程无事可做，它将调用WaitMessage，暂停线程的运行，使得CPU时间不会浪费。

清单12-1 Counter示例应用程序

Counter.cpp

```
/******************************************************************************
Module:  Counter.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


// The possible state of the background processing
typedef enum {
   BPS_STARTOVER,  // Start the background processing from the beginning.
   BPS_CONTINUE,   // Continue the background processing.
   BPS_DONE        // There is no background processing to do.
} BKGNDPROCSTATE;


typedef struct {
   PVOID pFiberUI;      // User interface fiber execution context
   HWND  hwnd;          // Handle of main UI window
   BKGNDPROCSTATE bps;  // State of background processing
} FIBERINFO, *PFIBERINFO;


// A global that contains application state information. This
// global is accessed directly by the UI fiber and indirectly
```

China-pub.com 下载

```
   // by the background processing fiber.
   FIBERINFO g_FiberInfo;


   ////////////////////////////////////////////////////////////////////////////


   void WINAPI FiberFunc(PVOID pvParam) {
   PFIBERINFO pFiberInfo = (PFIBERINFO) pvParam;

   // Update the window showing which fiber is executing.
   SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));

   // Get the current count in the EDIT control.
   int nCount = GetDlgItemInt(pFiberInfo->hwnd, IDC_COUNT, NULL, FALSE);

   // Count from 0 to nCount, updating the STATIC control.
   for (int x = 0; x <= nCount; x++) {

      // UI events have higher priority than counting.
      // If there are any UI events, handle them ASAP.
      if (HIWORD(GetQueueStatus(QS_ALLEVENTS)) != 0) {

         // The UI fiber has something to do; temporarily
         // pause counting and handle the UI events.
         SwitchToFiber(pFiberInfo->pFiberUI);

         // The UI has no more events; continue counting.
         SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));
      }

      // Update the STATIC control with the most recent count.
      SetDlgItemInt(pFiberInfo->hwnd, IDC_ANSWER, x, FALSE);

      // Sleep for a while to exaggerate the effect; remove
      // the call to Sleep in production code.
      Sleep(200);
   }

   // Indicate that counting is complete.
   pFiberInfo->bps = BPS_DONE;

   // Reschedule the UI thread. When the UI thread is running
   // and has no events to process, the thread is put to sleep.
   // NOTE: If we just allow the fiber function to return,
   // the thread and the UI fiber die -- we don't want this!
   SwitchToFiber(pFiberInfo->pFiberUI);
}


////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_COUNTER);
```

China-pub.com
下载


```
   SetDlgItemInt(hwnd, IDC_COUNT, 0, FALSE);
   return(TRUE);
}

///////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         PostQuitMessage(0);
         break;

      case IDC_COUNT:
         if (codeNotify == EN_CHANGE) {

            // When the user changes the count, start the
            // background processing over from the beginning.
            g_FiberInfo.bps = BPS_STARTOVER;
         }
         break;
   }
}

///////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

///////////////////////////////////////////////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Counter fiber execution context
   PVOID pFiberCounter = NULL;

   // Convert this thread to a fiber.
   g_FiberInfo.pFiberUI = ConvertThreadToFiber(NULL);

   // Create the application's UI window.
   g_FiberInfo.hwnd = CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_COUNTER),
      NULL, Dlg_Proc);

   // Update the window showing which fiber is executing.
   SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER, TEXT("User interface"));
```

China-pub.com 下载

```
// Initially, there is no background processing to be done.
g_FiberInfo.bps = BPS_DONE;

// While the UI window still exists...
BOOL fQuit = FALSE;
while (!fQuit) {

   // UI messages are higher priority than background processing.
   MSG msg;
   if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

      // If a message exists in the queue, process it.
      fQuit = (msg.message == WM_QUIT);
      if (!IsDialogMessage(g_FiberInfo.hwnd, &msg)) {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }

   } else {

      // No UI msgs exist; check the state of the background processing.
      switch (g_FiberInfo.bps) {
         case BPS_DONE:
            // No background processing to do; wait for a UI event.
            WaitMessage();
            break;

         case BPS_STARTOVER:
            // User changed the count; restart the background processing.
            if (pFiberCounter != NULL) {
               // A recalculation fiber exists; delete it so that
               // background processing starts over from the beginning.
               DeleteFiber(pFiberCounter);
               pFiberCounter = NULL;
            }

            // Create a new recalc fiber that starts from the beginning.
            pFiberCounter = CreateFiber(0, FiberFunc, &g_FiberInfo);

            // The background processing started; it should continue.
            g_FiberInfo.bps = BPS_CONTINUE;

            // Fall through to BPS_CONTINUE case...

         case BPS_CONTINUE:
            // Allow the background processing to execute...
            SwitchToFiber(pFiberCounter);

            // The background processing has been paused
            // (because a UI message showed up) or has been
            // stopped (because the counting has completed).

            // Update the window showing which fiber is executing.
            SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER,
               TEXT("User interface"));
```

China-pub.com
下载


```
            if (g_FiberInfo.bps == BPS_DONE) {
               // The background processing ran to completion. Delete the
               // fiber so that processing will restart next time.
               DeleteFiber(pFiberCounter);
               pFiberCounter = NULL;
            }
            break;
         }  // switch on background processing state

      }  // No UI messages exist
   }  // while the window still exists
   DestroyWindow(g_FiberInfo.hwnd);

   return(0);  // End the application.
}

//////////////////////////////// End of File //////////////////////////////////
```

Counter.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_COUNTER DIALOG DISCARDABLE  0, 0, 156, 37
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
CAPTION "Counter"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Count to:",IDC_STATIC,4,6,34,8
    EDITTEXT        IDC_COUNT,38,4,40,14,ES_AUTOHSCROLL | ES_NUMBER
```

China-pub.com 下载

```
    LTEXT           "Answer:",IDC_STATIC,90,6,25,8
    RTEXT           "0",IDC_ANSWER,122,6,23,8
    LTEXT           "Currently running fiber:",IDC_STATIC,4,24,75,8
    LTEXT           "Fiber",IDC_FIBER,80,24,72,8
END
/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_COUNTER, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 149
        TOPMARGIN, 7
        BOTTOMMARGIN, 30
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
```

China-pub.com
下载


```
// remains consistent on all systems.
IDI_COUNTER             ICON    DISCARDABLE     "Counter.ico"
#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第三部分　内 存 管 理

第13章　Windows的内存结构

操作系统使用的内存结构是理解操作系统如何运行的最重要的关键。当开始对一个新的操作系统进行操作时，你会想到一系列的问题。比如，“如何在两个应用程序之间共享数据呢？”“系统将要查找的信息存放在什么地方呢？”“如何使我的程序能够更加有效地运行呢？”等等。

很好地理解系统如何管理内存，可以帮助你更快和更准确地地回答这些问题。本章将要介绍Microsoft公司的Windows操作系统使用的内存结构。

13.1　进程的虚拟地址空间

每个进程都被赋予它自己的虚拟地址空间。对于32位进程来说，这个地址空间是4GB，因为32位指针可以拥有从0x00000000至0xFFFFFFFF之间的任何一个值。这使得一个指针能够拥有4 294 967 296个值中的一个值，它覆盖了一个进程的4GB虚拟空间的范围。对于64位进程来说，这个地址空间是16EB（10^{18}字节），因为64位指针可以拥有从0x0000000000000000至0xFFFFFFFFFFFFFFFF之间的任何值。这使得一个指针可以拥有18 446 744 073 709 551 616个值中的一个值，它覆盖了一个进程的16EB虚拟空间的范围。这是相当大的一个范围。

由于每个进程可以接收它自己的私有的地址空间，因此当进程中的一个线程正在运行时，该线程可以访问只属于它的进程的内存。属于所有其他进程的内存则隐藏着，并且不能被正在运行的线程访问。

> **注意**　在Windows 2000中，属于操作系统本身的内存也是隐藏的，正在运行的线程无法访问。这意味着线程常常不能访问操作系统的数据。Windows 98中，属于操作系统的内存是不隐藏的，正在运行的线程可以访问。因此，正在运行的线程常常可以访问操作系统的数据，也可以破坏操作系统（从而有可能导致操作系统崩溃）。在Windows 98中，一个进程的线程不可能访问属于另一个进程的内存。

前面说过，每个进程有它自己的私有地址空间。进程A可能有一个存放在它的地址空间中的数据结构，地址是0x12345678，而进程B则有一个完全不同的数据结构存放在它的地址空间中，地址是0x12345678。当进程A中运行的线程访问地址为0x12345678的内存时，这些线程访问的是进程A的数据结构。当进程B中运行的线程访问地址为0x12345678的内存时，这些线程访问的是进程B的数据结构。进程A中运行的线程不能访问进程B的地址空间中的数据结构。反之亦然。

当你因为拥有如此大的地址空间可以用于应用程序而兴高采烈之前，记住，这是个虚拟地址空间，不是物理地址空间。该地址空间只是内存地址的一个范围。在你能够成功地访问数据而不会出现违规访问之前，必须赋予物理存储器，或者将物理存储器映射到各个部分的地址空间。本章后面将要具体介绍这是如何操作的。

China-pub.com 下载

13.2 虚拟地址空间如何分区

每个进程的虚拟地址空间都要划分成各个分区。地址空间的分区是根据操作系统的基本实现方法来进行的。不同的Windows内核，其分区也略有不同。表13-1显示了每种平台是如何对进程的地址空间进行分区的。

表13-1 进程的地址空间如何分区

分 区	32位Windows 2000（x86和Alpha处理器）	32位Windows 2000(x86 w/3 GB用户方式)	64位Windows 2000（Alpha和IA-64处理器）	Windows 98
NULL指针分配的分区	0x00000000 0x0000FFFF	0x00000000 0x0000FFFF	0x00000000 00000000 0x00000000 0000FFFF	0X00000000 0x00000FFF
DOS/16位Windows应用程序兼容分区	无	无	无	0x000001000 0x003FFFFF
用户方式	0x00010000 0x7FFEFFFF	0x00010000 0xBFFEFFFFF	0x00000000 00010000 0x000003FF FFFEFFFF	0x00400000 0x7FFFFFFF
64-KB禁止进入	0x7FFF0000 0x7FFFFFFF	0xBFFF0000 0xBFFFFFFF	0x000003FFFFFF0000 0x000003FFFFFFFFFF	无
共享内存映射文件（MMF）内核方式	无 0x800000000 0xFFFFFFFF	无 0xC0000000 0xFFFFFFFF	无 0x00000400 00000000 0xFFFFFFFFF FFFFFFF	0x80000000 0xBFFFFFFF 0xC0000000 0xFFFFFFFF

如你所见，32位Windows 2000的内核与64位Windows 2000的内核拥有大体相同的分区，差别在于分区的大小和位置有所不同。另一方面，可以看到Windows 98下的分区有着很大的不同。下面让我们看一下系统是如何使用每一个分区的。

注意 Microsoft公司正在积极开发64位Windows 2000。但是当我撰写本书时，该系统仍在开发之中。应该使用本书中关于64位Windows 2000的信息，将它们用于你的当前项目的设计和实现中。不过应该知道，等到64位Windows 2000上市时，本章中介绍的一些详细信息很可能已经发生了变化。至于IA-64（64位Intel结构）的内存管理，分区和系统页面大小的特定虚拟地址范围也有可能变更。

13.2.1 NULL指针分配的分区——适用于Windows 2000和Windows 98

进程地址空间的这个分区的设置是为了帮助程序员掌握NULL指针的分配情况。如果你的进程中的线程试图读取该分区的地址空间的数据，或者将数据写入该分区的地址空间，那么CPU就会引发一个访问违规。保护这个分区是极其有用的，它可以帮助你发现NULL指针的分配情况。

C/C++程序中常常不进行严格的错误检查。例如，下面这个代码就没有进行任何错误检查：

```
int* pnSomeInteger = (int*) malloc(sizeof(int));
*pnSomeInteger = 5;
```

China-pub.com
下载

如果malloc不能找到足够的内存来满足需要，它就返回 NULL。但是，该代码并不检查这种可能性，它认为地址的分配已经取得成功，并且开始访问 0x00000000地址的内存。由于这个分区的地址空间是禁止进入的，因此就会发生内存访问违规现象，同时该进程将终止运行。这个特性有助于编程员发现应用程序中的错误。

13.2.2 MS-DOS/16位Windows应用程序兼容分区 —— 仅适用于Windows 98

进程地址空间的这个4MB分区是Windows 98需要的，目的是维护MS-DOS应用程序与16位应用程序之间的兼容性。不应该试图从 32位应用程序来读取该分区的数据，或者将数据写入该分区。在理想的情况下，如果进程中的线程访问该内存， CPU应该产生一个访问违规，但是由于技术上的原因，Microsoft无法保护这个4MB的地址空间。

在Windows 2000中，16位MS-DOS与16位Windows应用程序是在它们自己的地址空间中运行的，32位应用程序不会对它们产生任何影响。

13.2.3 用户方式分区 —— 适用于Windows 2000和Windows 98

这个分区是进程的私有（非共享）地址空间所在的地方。一个进程不能读取、写入、或者以任何方式访问驻留在该分区中的另一个进程的数据。对于所有应用程序来说，该分区是维护进程的大部分数据的地方。由于每个进程可以得到它自己的私有的、非共享分区，以便存放它的数据，因此，应用程序不太可能被其他应用程序所破坏，这使得整个系统更加健壮。

Windows 2000 在Windows 2000中，所有的.exe和DLL模块均加载这个分区。每个进程可以将这些DLL加载到该分区的不同地址中（不过这种可能性很小）。系统还可以在这个分区中映射该进程可以访问的所有内存映射文件。

Windows 98 在Windows 98中，主要的Win32系统DLL（Kernel32.dll，AdvAPI32.dll，User32.dll和GDI32.dll）均加载共享内存映射文件分区中。 .exe和所有其他DLL模块则加载到这个用户方式分区中。所有进程的共享 DLL均位于相同的虚拟地址中，但是其他DLL可以将这些DLL加载到用户方式分区的不同地址中（不过这种可能性不大）。另外，在Windows 98中，用户方式分区中决不会出现内存映射文件。

当我最初观察32位进程的地址空间的时候，我惊奇地发现可以使用的地址空间还不到我的进程的全部地址空间的一半。难道内核方式分区真的需要上面的一半地址空间吗？实际上回答是肯定的。系统需要这个地址空间，供内核代码、设备驱动程序代码、设备 I/O高速缓存、非页面内存池的分配和进程页面表等使用。实际上 Microsoft将内核压缩到这个2GB空间之中。在64位Windows 2000中，内核终于得到了它真正需要的空间。

1. 在x86 的Windows2000中获得3GB用户方式分区

多年来，编程人员一直强烈要求扩大用户方式的地址空间。为了满足这个需要， Microsoft允许x86的Windows 2000 Advanced Server版本和Windows 2000 Data Center版本将用户方式分区扩大为3GB。若要使所有进程都能够使用 3GB用户方式分区和 1GB内核方式分区，必须将/3GB开关附加到系统的 BOOT.INI文件的有关项目中。表 13-1中的“ 32位Windows 2000（x86 w/3GB用户方式）”这一列显示了使用3GB开关时它的地址空间是个什么样子。

在Microsoft添加/3GB开关之前，应用程序无法看到设置了高位的内存指针。一些有创意的编程员自己将这个高位用作一个标志，这个标志只对他们的应用程序具有意义。这时，当应用程序访问内存地址时，运行的代码将在内存地址被使用之前清除该指针的高位。可以想象，

China-pub.com 下载

当应用程序在3GB的用户方式环境中运行时，该应用程序转眼之间就会运行失败。

Microsoft不得不提出一个解决方案，以便使该应用程序能够在3GB环境中运行。当系统准备运行一个应用程序时，它要查看该应用程序是否与/LARGEADDRESSAWARE链接程序开关相链接。如果是链接的，那么应用程序就声称它并没有对内存地址执行什么特殊的操作，并且完全准备充分利用3GB 用户方式地址空间。另一方面，如果该应用程序没有与/LARGEADDRESSAWARE开关相链接，那么操作系统将保留 0x80000000至0xBFFFFFFF之间的1GB区域。这可以防止在已经设置了高位的内存地址上进行内存分配。

注意，内核已经被紧紧地压缩到了一个 2GB的分区中。当使用 3GB的开关时，内核勉强地被放入一个1 GB的分区中。使用/3GB的开关，可以减少系统能够创建的线程、堆栈和其他资源的数量。此外，系统最多只能使用 16GB的RAM，而通常情况下最多可以使用 64GB的RAM，因为内核方式中没有足够的虚拟地址空间可以用来管理更多的 RAM。

注意 当操作系统创建进程的地址空间时，需要检查一个可执行的 LARGEADDRESSAWARE标志。对于DLL，系统则忽略该标志。在编写 DLL时，必须使之能够在 3 GB用户方式分区中正确地运行，否则它们的行为特性是无法确定的。

2. 在64位Windows 2000中获得2 GB用户方式分区

Microsoft发现许多编程人员需要尽可能迅速而方便地将现有的 32位应用程序移植到64位环境中去。但是，在许多源代码中，指针被视为 32位值。如果简单地重新编写应用程序，就会造成指针被截断的错误和不正确的内存访问。

然而，如果系统能够确保不对 0x000000007FFFFFFF以上的内存地址进行分配，那么应用程序就能很好地运行。当较高的33位是0时，将64位地址截断为32位地址，不会产生任何问题。通过在地址空间范围内运行应用程序，而这个地址空间范围将进程的可用地址空间限制为最低的GB，那么系统就能够确保这一点。

默认情况下，当启动一个 64位应用程序时，系统将保留从 0x000000080000000开始的所有用户地址空间。这可以确保在底部的 2GB 64位地址空间中进行所有的内存分配。这就是地址空间的范围。对于大多数应用程序来说，这个地址空间足够了。若要使 64位应用程序能够访问它的全部4 TB(terabyte)用户方式分区，该应用程序必须使用 /LARGEADDRESSAWARE 链接开关来创建。

注 意 当操作系统创建进程的 64位地址空间时，要检查一个可执行文件的 LARGEADDRESSAWARE标志。如果是DLL，那么系统将忽略该标志。编写DLL时，必须使之能够在整个4 TB用户方式分区中正确地运行，否则它们的行为特性将无法确定。

13.2.4 64 KB禁止进入的分区 —— 仅适用于Windows 2000

这个位于用户方式分区上面的64 KB分区是禁止进入的，访问该分区中的内存的任何企图均将导致访问违规。Microsoft之所以保留该分区，是因为这样做将使得Microsoft能够更加容易地实现操作系统。当将内存块的地址和它的长度传递给Windows函数时，该函数将在执行它的操作之前使内存块生效。可以很容易创建类似下面这个代码（在32位Windows 2000系统上运行）：

```
BYTE  bBuf[70000];
DWORD dwNumBytesWritten;
WriteProcessMemory(GetCurrentProcess(), (PVOID) 0x7FFEEE90, bBuf,
   sizeof(bBuf), &dwNumBytesWritten);
```

China-pub.com
下载

对于WriteProcessMemory这样的函数来说，写入的内存区是由内核方式代码来使之生效的，该代码能够成功地访问内核方式分区中的内存（32位系统上0x80000000以上的地址）。如果在0x80000000地址上存在内存，上面的函数调用就能成功地将数据写入只应该由内核方式代码访问的内存。为了防止出现这种情况，并使这个内存区迅速生效，Microsoft选择的办法是使该分区始终保持禁止进入的状态。只要试图读取或写入该分区中的内存，就一定会导致访问违规。

13.2.5 共享的MMF分区——仅适用于Windows 98

这个1GB分区是系统用来存放所有32位进程共享数据的地方。例如，系统的动态链接库Kernel32.dll、AdvAPI32.dll、User32.dll和GDI32.dll等，全部存放在这个地址空间分区中，因此，所有32位进程都能很容易同时访问它们。系统还为每个进程将DLL加载相同的内存地址。此外，系统将所有内存映射文件映射到这个分区中。内存映射文件将在第17章中详细介绍。

13.2.6 内核方式分区——适用于Windows 2000和Windows 98

这个分区是存放操作系统代码的地方。用于线程调度、内存管理、文件系统支持、网络支持和所有设备驱动程序的代码全部在这个分区加载。驻留在这个分区中的一切均可被所有进程共享。在Windows 2000中，这些组件是完全受到保护的。如果你试图访问该分区中的内存地址，你的线程将会产生访问违规，导致系统向用户显示一个消息框，并关闭你的应用程序。关于访问违规和如何处理这些违规的详细说明，请参见第23、24和25章的内容。

Windows 2000　在64位Windows 2000中，4 TB用户方式分区看上去与16, 777, 212 TB的内核方式分区非常不成比例。并不是内核方式分区需要使用该虚拟地址空间的全部空间，它只是说明64位地址空间是非常大的，而该地址空间的大部分是不用的。系统允许应用程序使用4 TB分区，并且允许内核使用它需要的东西，而内核方式分区的大部分是不用的。幸好系统并不需要任何内部数据结构来维护内核方式分区的不用部分。

Windows 98　不幸的是，在Windows 98中该分区中的数据是不受保护的。任何应用程序都可以从该分区读取数据，也可以写入数据，因此有可能破坏操作系统。

13.3 地址空间中的区域

当进程被创建并被赋予它的地址空间时，该可用地址空间的主体是空闲的，即未分配的。若要使用该地址空间的各个部分，必须通过调用VirtualAlloc函数（第15章介绍）来分配它里边的各个区域。对一个地址空间的区域进行分配的操作称为保留(reserving)。

每当你保留地址空间的一个区域时，系统要确保该区域从一个分配粒度的边界开始。对于不同的CPU平台来说，分配粒度是各不相同的。但是，截止到撰写本书时，所有的CPU平台（x86、32位Alpha、64位Alpha和IA-64）都使用64KB这个相同的分配粒度。

当你保留地址空间的一个区域时，系统还要确保该区域的大小是系统的页面大小的倍数。页面是系统在管理内存时使用的一个内存单位。与分配粒度一样，不同的CPU，其页面大小也是不同的。x86使用的页面大小是4 KB，而Alpha（当既能运行32位Windows 2000也能运行64位Windows 2000时）使用的页面大小则是8 KB。在撰写本书时，Microsoft预计IA-64也使用8 KB的页面。但是，如果测试显示使用更大的页面能够提高系统的总体性能，那么Microsoft可

China-pub.com 下载

以切换到更大的页面（16KB或更大）。

注意 有时系统能够代表你的进程来保留地址空间的区域。例如，系统可以分配一个地址空间区域，以便存放进程环境块（FEB）。FEB是由系统创建、操作和撤消的一个小型数据结构。当创建一个进程时，系统就为FEB分配一个地址空间区域。

系统也需要创建一个线程环境块（TEB），以便管理进程中当前存在的所有线程。用于这些TEB的区域将根据进程中的线程被创建和撤消等情况而保留和释放。

虽然系统规定，要求保留的地址空间区域均从分配粒度边界（目前所有平台上均为64KB）开始，但是系统本身并不受这个规定的限制。为你的进程的PEB和TEB保留的地址空间区域很可能不是从64 KB这个边界开始的。不过这些保留区域仍然必须是CPU的页面大小的倍数。

如果想保留一个10 KB的地址空间区域，系统将自动对你的请求进行四舍五入，使保留的地址空间区域的大小是页面大小的倍数。这意味着，在x86平台上，系统将保留一个12KB的区域，在Alpha平台上，系统将保留一个16KB的区域。

当你的程序算法不再需要访问已经保留的地址空间区域时，该区域应该被释放。这个过程称为释放地址空间的区域，它是通过调用VirtualFree函数来完成的。

13.4 提交地址空间区域中的物理存储器

若要使用已保留的地址空间区域，必须分配物理存储器，然后将该物理存储器映射到已保留的地址空间区域。这个过程称为提交物理存储器。物理存储器总是以页面的形式来提交的。若要将物理存储器提交给一个已保留的地址空间区域，也要调用VirtualAlloc函数。

当将物理存储器提交给地址空间区域时，不必将物理存储器提交给整个区域。例如，可以保留一个64KB的区域，然后将物理存储器提交给该区域中的第二和第四个页面。图13-1显示了进程的地址空间是个什么样子。注意，根据运行的CPU平台的不同，地址空间是各有差别的。左边的地址空间显示了x86计算机（它的页面大小是4 KB）上的情况，而右边的地址空间则显示了Alpha计算机（它的页面大小是8 KB）上发生的情况。

当你的程序算法不再需要访问保留的地址空间区域中已提交的物理存储器时，该物理存储器应该被释放。这个过程称为回收物理存储器，它是通过VirtualFree函数来完成的。



图13-1 不同的CPU使用的示例进程地址空间

13.5 物理存储器与页文件

在较老的操作系统中，物理存储器被视为计算机拥有的RAM的容量。换句话说，如果计

China-pub.com
下载

算机拥有16MB的RAM，那么加载和运行的应用程序最多可以使用16MB的RAM。今天的操作系统能够使得磁盘空间看上去就像内存一样。磁盘上的文件通常称为页文件，它包含了可供所有进程使用的虚拟内存。

当然，若要使虚拟内存能够运行，需要得到CPU本身的大量帮助。当一个线程试图访问一个字节的内存时，CPU必须知道这个字节是在RAM中还是在磁盘上。

从应用程序的角度来看，页文件透明地增加了应用程序能够使用的RAM（即内存）的数量。如果计算机拥有64MB的RAM，同时在硬盘上有一个100 MB的页文件，那么运行的应用程序就认为计算机总共拥有164MB的RAM。

当然，实际上并不拥有164MB的RAM。相反，操作系统与CPU相协调，共同将RAM的各个部分保存到页文件中，当运行的应用程序需要时，再将页文件的各个部分重新加载到RAM。由于页文件增加了应用程序可以使用的RAM的容量，因此页文件的使用是视情况而定的。如果没有页文件，那么系统就认为只有较少的RAM可供应用程序使用。但是，我们鼓励用户使用页文件，这样他们就能够运行更多的应用程序，并且这些应用程序能够对更大的数据集进行操作。最好将物理存储器视为存储在磁盘驱动器（通常是硬盘驱动器）上的页文件中的数据。这样，当一个应用程序通过调用VirtualAlloc函数，将物理存储器提交给地址空间的一个区域时，地址空间实际上是从硬盘上的一个文件中进行分配的。系统的页文件的大小是确定有多少物理存储器可供应用程序使用时应该考虑的最重要的因素，RAM的容量则影响非常小。

现在，当你的进程中的一个线程试图访问进程的地址空间中的一个数据块时，将会发生两种情况之一，参见图13-2中的流程图。

在第一种情况中，线程试图访问的数据是在RAM中。在这种情况下，CPU将数据的虚拟内存地址映射到内存的物理地址中，然后执行需要的访问。

在第二种情况中，线程试图访问的数据不在RAM中，而是存放在页文件中的某个地方。这时，试图访问就称为页面失效，CPU将把试图进行的访问通知操作系统。这时操作系统就寻找RAM中的一个内存空页。如果找不到空页，系统必须释放一个空页。如果一个页面尚未被修改，系统就可以释放该页面。但是，如果系统需要释放一个已经修改的页面，那么它必须首先将该页面从RAM拷贝到页交换文件中，然后系统进入该页文件，找出需要访问的数据块，并将数据加载到空闲的内存页面。然后，操作系统更新它的用于指明数据的虚拟内存地址现在已经映射到RAM中的相应的物理存储器地址中的表。这时CPU重新运行生成初始页面失效的指令，但是这次CPU能够将虚拟内存地址映射到一个物理RAM地址，并访问该数据块。

系统需要将内存页面拷贝到页文件并反过来将页文件拷贝到内存页面的次数越多，你的硬盘倒腾的次数就越多，系统运行得越慢（倒腾意味着操作系统要花费更多的时间将页面从内存中转出转进，而不是将时间用于程序的运行）。因此，通过给你的计算机增加更多的RAM，就可以减少运行应用程序所需的倒腾次数，这就必然可以大大提高系统的运行速度。所以必须遵循一条基本原则，那就是要让你的计算机运行得更块，增加更多的RAM。实际上，在大多数情况下，若要提高系统的运行性能，增加RAM比提高CPU的速度所产生的效果更好。

不在页文件中维护的物理存储器

当阅读了上一节后，你必定会认为，如果同时运行许多文件的话，页文件就可能变得非常大，而且你会认为，每当你运行一个程序时，系统必须为进程的代码和数据保留地址空间的一些区域，将物理存储器提交给这些区域，然后将代码和数据从硬盘上的程序文件拷贝到页文件中已提交的物理存储器中。

China-pub.com 下载

实际上系统并不进行上面所说的这些操作。如果它进行这些操作的话，就要花费很长的时间来加载程序并启动它运行。相反，当启动一个应用程序的时候，系统将打开该应用程序的.exe文件，确定该应用程序的代码和数据的大小。然后系统要保留一个地址空间的区域，并指明与该区域相关联的物理存储器是在.exe文件本身中。即系统并不是从页文件中分配地址空间，而是将.exe文件的实际内容即映像用作程序的保留地址空间区域。当然，这使应用程序的加载非常迅速，并使页文件能够保持得非常小。

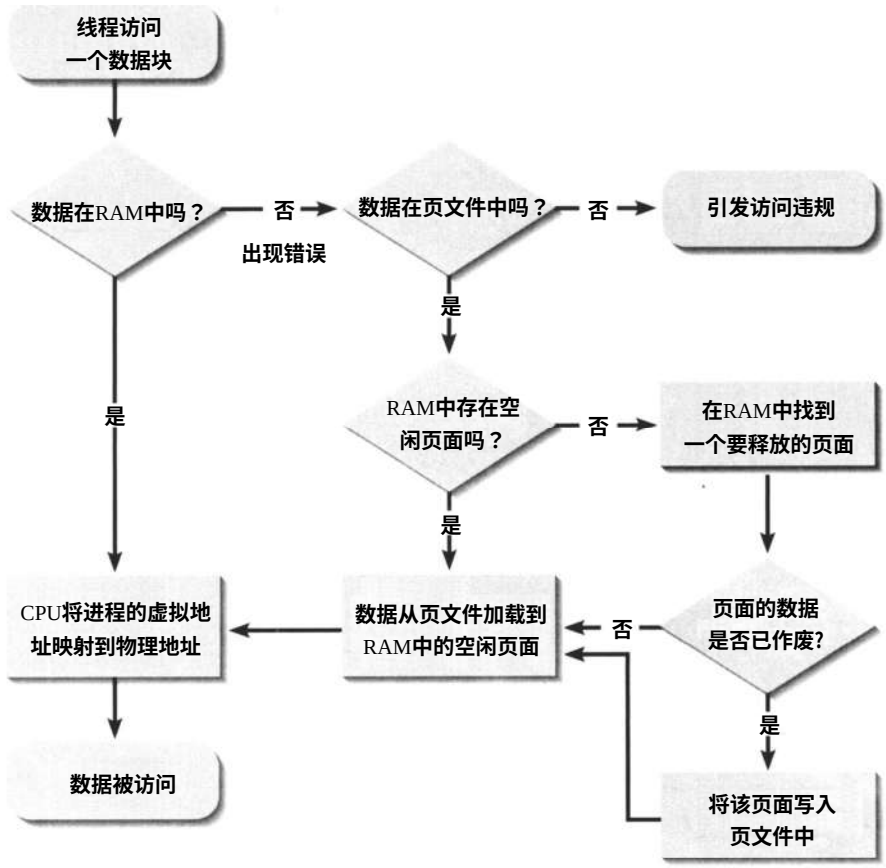


图13-2 将虚拟地址转换成物理存储器地址的流程图

当硬盘上的一个程序的文件映像（这是个.exe文件或DLL文件）用作地址空间的区域的物理存储器时，它称为内存映射文件。当一个.exe文件或DLL文件被加载时，系统将自动保留一个地址空间的区域，并将该文件映像映射到该区域中。但是，系统也提供了一组函数，使你能够将数据文件映射到一个地址空间的区域中。关于内存映射文件的详细说明，将在第17章中介绍。

Windows 2000 Windows 2000能够使用多个页文件。如果多个页文件存在于不同的物理硬盘驱动器上，系统的运行将能得快得多，因为它能够将数据同时写入多个驱动器。打开System Properties Control Panel（系统属性控制面板）小程序，再选择Advanced选项卡，单击Performance Options（性能选项）按钮，就能够添加或删除页文件。图13-3显示了该对话框的形式。

China-pub.com
下载

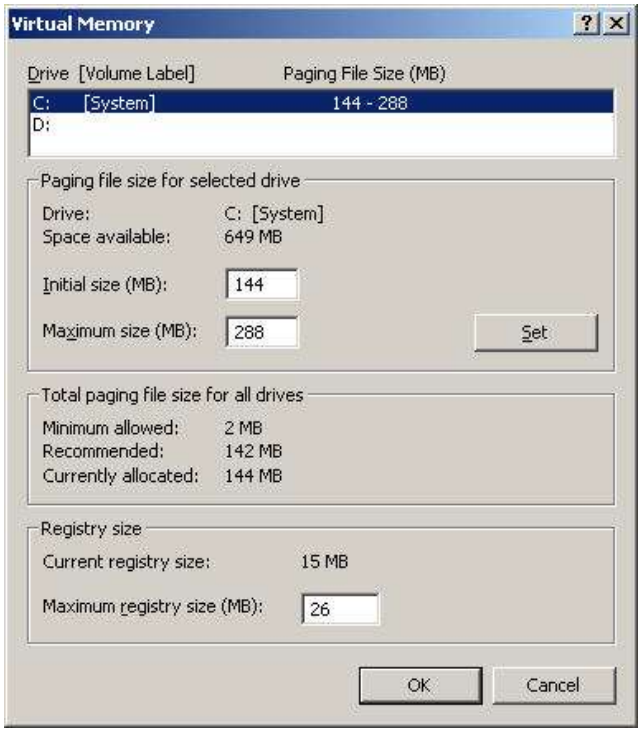


图13-3 Virtual Memory 对话框

注意 当.exe或DLL文件从软盘加载时，Windows 98和Windows 2000都能将整个文件从软盘拷贝到系统的RAM中。此外，系统将从页文件中分配足够的内存，以便存放该文件的映像。如果系统选择对当前包含该文件的一部分映像的RAM页面进行裁剪，那么该内存属于只能写入的内存。如果系统RAM上的负载比较小，那么文件始终都可以直接从RAM来运行。

Microsoft不得不通过软盘来运行的映射文件，这样，安装应用程序才能正确运行。安装程序常常从一个软盘开始，然后用户将软盘从驱动器中取出来，再插入另一个软盘。如果系统需要回到第一个软盘，以便加载.exe或DLL文件的某些代码，当然该代码已经不再在软盘驱动器中了。然而，由于系统将文件拷贝到RAM（并且受页文件的支持），要访问安装程序是不会有任何问题的。

系统并不将RAM映射文件拷贝在其他可换式介质上，如光盘或网络驱动器，除非映射文件是用/SWAPRUN：CD或/SWAPRUN：NET开关链接的。注意，Windows 98不支持/SWAPRUN映像标志。

13.6 保护属性

已经分配的物理存储器的各个页面可以被赋予不同的保护属性。表13-2显示了这些保护属性。

x86和Alpha CPU不支持“执行”保护属性，不过操作系统软件却支持这个属性。这些CPU将读访问视为执行访问。这意味着如果将PAGE_EXECUTE保护属性赋予内存，那么该内存也将拥有读优先权。当然，不应该依赖这个行为特性，因为在其他CPU上的Windows实现代码很

China-pub.com 下载

表13-2 页面的保护属性

保护属性	描述
PAGE_NOACCESS	如果试图在该页面上读取、写入或执行代码，就会引发访问违规
PAGE_READONLY	如果试图在该页面上写入或执行代码，就会引发访问违规
PAGE_READWRITE	如果试图在该页面上执行代码，就会引发访问违规
PAGE_EXECUTE	如果试图在该页面上对内存进行读取或写入操作，就会引发访问违规
PAGE_EXECUTE_READ	如果试图在该页面上对内存进行写入操作，就会引发访问违规
PAGE_EXECUTE_READWRITE	对于该页面不管执行什么操作，都不会引发访问违规
PAGE_WRITECOPY	如果试图在该页面上执行代码，就会引发访问违规。如果试图在该页面上写入内存，就会导致系统将它自己的私有页面（受页文件的支持）拷贝赋予该进程
PAGE_EXECUTE_WRITECOPY	对于该地址空间的区域，不管执行什么操作，都不会引发访问违规。如果试图在该页面上的内存中进行写入操作，就会将它自己的私有页面（受页文件的支持）拷贝赋予该进程

可能将“执行”保护视为“仅为执行”保护。

Windows 98 Windows 98只支持PAGE_NOACCESS、PAGE_READONLY和PAGE_READWRITE等保护属性。

13.6.1 Copy-On-Write 访问

表13-2列出的保护属性都是非常容易理解的，不过最后两个属性需要作一些说明。一个是PAGE_WRITECOPY，另一个是PAGE_EXECUTE_WRITECOPY。这两个属性的作用是为了节省RAM的使用量和页文件的空间。Windows支持一种机制，使得两个或多个进程能够共享单个内存块。因此，如果10个Notepad实例正在运行，那么所有实例可以共享应用程序的代码和数据页面。让所有实例共享同样的内存页面将能够大大提高系统的性能，但是这要求所有实例都将该内存视为只读或只执行的内存。如果一个实例中的线程将数据写入内存修改它，那么其他实例看到的这个内存也将被修改，从而造成一片混乱。

为了防止出现这种混乱，操作系统给共享内存块赋予了 Copy-On-Write保护属性。当一个.exe或DLL模块被映射到一个内存地址时，系统将计算有多少页面是可以写入的（通常包含代码的页面标为 PAGE_EXECUTE_READ，而包含数据的页面则标为 PAGE_READWRITE）。然后，系统从页文件中分配内存，以适应这些可写入的页面的需要。除非该模块的可写入页面是实际的写入模块，否则这些页文件内存是不使用的。

当一个进程中的线程试图将数据写入一个共享内存块时，系统就会进行干预，并执行下列操作步骤：

1) 系统查找RAM中的一个空闲内存页面。注意，当该模块初次被映射到进程的地址空间时，该空闲页面将被页文件中已分配的页面之一所映射。当该模块初次被映射时，由于系统要分配所有可能需要的页文件，因此这一步不可能运行失败。

2) 系统将试图被修改的页面内容拷贝到第一步中找到的页面。该空闲页面将被赋予PAGE_READWRITE或PAGE_EXECUTE_READWRITE保护属性。原始页面的保护属性和数据不发生任何变化。

3) 然后系统更新进程的页面表，使得被访问的虚拟地址被转换成新的 RAM页面。

当系统执行了这3个操作步骤之后，该进程就可以访问它自己的内存页面的私有实例。第

China-pub.com
下载

17章还要详细地介绍共享内存和Copy-On-Write保护属性。

此外，当使用 VirtualAlloc函数来保留地址空间或者提交物理存储器时，不应该传递PAGE_WRITECOPY或PAGE_EXECUTE_WRITECOPY。如果传递的话，将会导致VirtualAlloc调用的失败。对GetLastError的调用将返回ERROR_INVALID_PARAMETER。当操作系统映射.exe或DLL文件映像时，这两个属性将被操作系统使用。

Windows 98 Windows 98不支持Copy-On-Write保护。当Windows 98发现需要Copy_On_Write保护时，它就立即进行数据的拷贝，而不是等待试图对内存进行写入操作。

13.6.2 特殊的访问保护属性的标志

除了上面介绍的保护属性外，还有 3个保护属性标志，即 PAGE_NOCACHE，PAGE_WRITECOMBINE和PAGE_GUARD。可以用OR逐位将它们连接，以便将这 3个标志用于任何一个保护属性（PAGE_NOCACHE除外）。

第一个保护属性标志PAGE_NOCACHE用于停用已提交页面的高速缓存。一般情况下最好不要使用该标志，因为它主要是供需要处理内存缓冲区的硬件设备驱动程序的开发人员使用的。

第二个保护属性PAGE_WRITECOMBINE也是供设备驱动程序开发人员使用的。它允许把单个设备的多次写入合并在一起，以便提高运行性能。

最后一个保护属性标志PAGE_GUARD可以在页面上写入一个字节时使应用程序收到一个通知（通过一个异常条件）。该标志有一些非常巧妙的用法。 Windows 2000在创建线程堆栈时使用该标志。关于该标志的详细说明，参见第16章。

Windows 98 Windows 98将忽略PAGE_NOCACHE、PAGE_WRITECOMBINE和PAGE_GUARD这3个保护属性标志。

13.7 综合使用所有的元素

本节要将地址空间、分区、区域、内存块和页面等元素综合起来加以使用。为了更好地说明问题，我们首先来看一看虚拟内存表，它可以显示单个进程中的所有地址空间的区域。该进程正好是第14章中介绍的VMMap示例应用程序。为了全面了解进程的地址空间，首先要介绍一下当在32位x86计算机上的Windows 2000下运行VMMap时，地址空间是个什么样子。表13-3显示了一个示例地址空间表。后面将要介绍 Windows 2000与Windows 98的地址空间之间的差别。

表13-3中的地址空间表显示了进程的地址空间中的各个不同区域。每行显示一个区域，每行包含6列。

第一列，即最左边的一列显示了区域的基地址。你会发现我们是从地址为 0x00000000的区域开始观察进程的地址空间的，并在可用地址空间的最后一个区域结束，该区域的起始地址是0x7FFE0000。所有区域都是相邻的。你也会注意到，非空闲区域的所有基地址几乎都是从64KB的倍数上开始的。这是由系统采用的保留地址空间的分配粒度所决定的。不是从分配粒度边界开始的区域，表示该区域是由操作系统代码代表你的进程来分配的。

China-pub.com 下载

表13-3 显示了32位x86计算机上运行的Windows 2000下的地址空间区域的示例地址空间表

基地址	类型	大小	块	保护属性	描述
00000000	空闲	65 536			
00010000	私有	4096	1	-RW-	
00011000	空闲	61 440			
00020000	私有	4096	1	-RW-	
00021000	空闲	61 440			
00030000	私有	1 048 576	3	-RW-	线程堆栈
00130000	私有	1 048 576	2	-RW-	
00230000	映射	65 536	2	-RW-	
00240000	映射	90 112	1	-RW-	\Device\HarddiskVolume1\WINNT\system32\unicode.nls
00256000	空闲	40 960			
00260000	映射	208 896	1	-R--	\Device\HarddiskVolume1\WINNT\system32\locale.nls
00293000	空闲	53248			
002A0000	映射	266 240	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sortkey.nls
002E1000	空闲	61 400			
002F0000	映射	16 384	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sorttbls.nls
002F4000	空闲	49 152			
00300000	映射	819 200	4	ER--	
003C8000	空闲	229 376			
00400000	映像	106 496	5	ERWC	C:\CD\x86\Debug\14\VMMap.exe
0041A000	空闲	24 576			
00420000	映像	274 432	1	-R--	
00463000	空闲	53 248			
00470000	映像	3145 728	2	ER--	
00770000	私有	4096	1	-RW-	
00771000	空闲	61 440			
00780000	私有	4096	1	-RW-	
00781000	空闲	61 440			
00790000	私有	65 536	2	-RW-	
007A0000	映射	8192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A2000	空闲	1 763 893 248			
699D0000	映像	45 056	4	ERWC	C：\WINNT\System32\PSAPI.dll
699DB000	空闲	238 505 984			
77D50000	映像	450 560	4	ERWC	C：\WINNT\System32\RPCRT4.dll
77DBE000	空闲	8192			
77DC0000	映像	344 064	5	ERWC	C：\WINNT\System32\ADVAPI32.dll
77E14000	空闲	49152			
77E20000	映像	401 408	4	ERWC	C：\WINNT\System32\USER32.dll
77E82000	空闲	57 344			
77E90000	映像	720 896	5	ERWC	C：\WINNT\System32\KERNEL32.dll
77F40000	映像	241 664	4	ERWC	C：\WINNT\System32\GDI32.dll
77F7B000	空闲	20 480			

China-pub.com
下载

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
77F80000	映像	483 328	5	ERWC	C：\WINNT\System32\ntdl1.dll
77FF6000	空闲	40 960			
78000000	映像	290 816	6	ERWC	C：\WINNT\System32\MSVCRT.dll
78047000	空闲	124 424 192			
7F6F0000	映射	1 048 576	2	ER--	
7F7F0000	空闲	8 126 464			
7FFB0000	映射	147 456	1	-R--	
7FFD4000	空闲	40 960			
7FFDE000	私有	4096	1	ERW-	
7FFDF000	私有	4096	1	ERW-	
7FFE0000	私有	65 536	2	-R--	

第二列显示了区域的类型。区域类型共有 4个值，即空闲，私有，映像或映射。表 13-4对它们进行了介绍。

表13-4 区域类型说明

类 型	说 明
空闲	该区域的虚拟地址不受任何内存的支持。该地址空间没有被保留。应用程序既可以将一个区域保留在显示的基地址上，也可以保留在空闲区域中的任何位置上
私有	该区域的虚拟地址将受系统的页文件的支持。
映像	该区域的虚拟地址原先受内存映射的映像文件（如 .exe或DLL文件）的支持，但也许不再受映像文件的支持。例如，当写入模块映像中的全局变量时，“写入时拷贝”的机制将由页文件来支持特定的页面，而不是受原始映像文件的支持
映射	该区域的虚拟地址原先是受内存映射的数据文件的支持，但也许不再受数据文件的支持。例如，数据文件可以使用“写入时拷贝”的保护属性来映射。对文件的任何写入操作都将导致页文件而不是原始数据支持特定的页面

我的VMMap应用程序计算这一列的方法可能产生错误的结果。当地址空间区域不空闲时，VNNap示例应用程序就要猜测剩余的 3个值中哪一个可以使用。没有一个函数可以调用，以便确定该区域的准确用途。这一列的值的方法是，对区域中的所有内存块进行扫描，然后进行合乎逻辑的推测。可以参考第14章中的代码，以便更好地了解计算方法。

第三列显示了为该区域保留的字节数量。例如，系统将 User.DLL的映像映射到内存地址 0x77E20000。当系统为该映像保留地址空间时，它必须保留 401 408个字节。第三列中的数字总是CPU的页面大小的倍数（x86CPU为4096字节）。

第四列显示了保留区域中的块的数量。所谓块是指一组相邻的页面，它们拥有相同的保护属性，并且都是受相同类型的物理存储器支持的。下一节将要详细介绍这个问题。对于空闲区域来说，这个值始终都是 0，因为在空闲区域中不能提交任何内存（在第四列中空闲区域不显示任何信息）。对于非空闲区域来说，这个值可以是 1到区域大小/页面大小的最大数字之间的任何值。例如，从内存地址 0x77E20000开始的区域，它的区域大小是 401 408个字节。由于该进程是在x86 CPU上运行的（x86 CPU的页面大小是4096个字节），因此提交的各种不同的块的最大数量是98（401 408/4096）。表中显示该区域中的块的数量是 4。

第五列显示了区域的保护属性。各个字母所代表的含义是： E=执行，R=读取，W=写入，C=写入时拷贝。如果一个区域没有显示任何保护属性，那么该区域就没有访问保护。空闲区域没有显示任何保护属性，因为未保留区域不拥有与其相关联的保护属性。这里决不会出现

China-pub.com 下载

GUARD保护属性标志或NO-CACHE保护属性标志。只有当这些标志与物理存储器相关联，而不是与保留的地址空间相关联时，它们才具有意义。给一个区域赋予保护属性的目的只是为了提高效率，并且总是会被赋予物理存储器的保护属性所取代。

第六列即最后一列显示了区域中的信息的文字描述。如果是空闲区域，那么这一列总是空的。如果是私有区域，它通常也是空的，因为 VMMap没有办法知道应用程序为什么要保留这个私有地址空间区域。但是，VMMap能够识别包含线程堆栈的私有区域。VMMap通常能够发现线程堆栈，因为它们通常拥有一个具有 GUARD保护属性的物理存储器块。不过，当线程堆栈满了的时候，它就不再拥有一个保护属性为GUARD的物理存储器块，同时VMMap将无法发现它。

对于映像区域来说，VMMap将显示映射到该区域中的文件的全路径名。使用 ToolHelp函数，VMMap就可以获得该信息。在 Windows 2000中，通过调用 GetMappedFileName函数（Windows 98中没有这个函数），VMMap就能够显示受数据文件支持的区域。

13.7.1 区域的内部情况

我们还可以将区域划分得比表 13-3显示的情况更细一些。表 13-5显示的地址空间与表 13-3所示的地址空间相同，但是它也显示了每个区域中包含的内存块。

表13-5 显示了32位x86计算机上的Windows 2000下的内存区域和块的示例地址空间表

基地址	类型	大小	块	保护属性	描述
00000000	空闲	65 536			
00010000	私有	4096	1	-RW-	
00010000	私有	4096		-RW- ---	
00011000	空闲	61 440			
00020000	私有	4096	1	-RW-	
00020000	私有	4096		-RW- ---	
00021000	空闲	61 440			
00030000	私有	1 048 576	3	-RW-	线程堆栈
00030000	保留	905 216		-RW- ---	
0010D000	私有	4096		-RW- G--	
0010E000	私有	139 264		-RW- ---	
00130000	私有	1048576	2	-RW-	
00130000	私有	36 864		-RW- ---	
00139000	保留	1 011 712		-RW- ---	
00230000	映射	65 536	2	-RW-	
00230000	映射	4096		-RW- ---	
00231000	保留	61 440		-RW- ---	
00240000	映射	90 112	1	-R--	\Device\HarddiskVolumel\WINNT\system32\unicode.nls
00240000	映射	90 112		-R-- ---	
00256000	空闲	40 960			
00260000	映射	208 896	1	-R--	\Device\HarddiskVolumel\WINNT\system32\locale.nls
00260000	映射	208 896		-R-- ---	
00293000	空闲	53248			

China-pub.com
下载

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
002A0000	映射	266240	1	-R--	\Device\HarddiskVolumel\WINNT\ system32\sortkey.nls
002A0000	映射	266240		-R-- ---	
002E1000	空闲	61 440			
002F0000	映射	16 384	1	-R--	\Device\HarddiskVolumel\WINNT system32\sorttbls.nls
002F0000	映射	16 384		-R-- ---	
002F4000	空闲	49 152			
00300000	映射	819 200	4	ER--	
00300000	映射	16 384		ER-- ---	
00304000	保留	770 048		ER-- ---	
003C0000	映射	8192		ER-- ---	
003C2000	保留	24 576		ER-- ---	
003C8000	空闲	229 376			
00400000	映像	106 496	5	ERWC	C:\CD\x86\Debug\14 VMMap.exe
00400000	映像	4096		-R-- ---	
00401000	映像	81 920		ER-- ---	
00415000	映像	4096		-R-- ---	
00416000	映像	8192		-RW- ---	
00418000	映像	8192		-R-- ---	
0041A000	空闲	24 576			
00420000	映射	274 432	1	-R--	
00420000	映射	274 432		-R-- ---	
00463000	空闲	53 248			
00470000	映射	3 145 728	2	ER--	
00470000	映射	274 432		ER-- ---	
004B3000	保留	2 871 296		ER-- ---	
00770000	私有	4096	1	-RW-	
00770000	私有	4096		-RW- ---	
00771000	空闲	61 440			
00780000	私有	4096	1	-RW-	
00780000	私有	4096		-RW- ---	
00781000	空闲	61 440			
00790000	私有	65 536	2	-RW-	
00790000	私有	20 480		-RW- ---	
00795000	保留	45 056		-RW- ---	
007A0000	映射	8192	1	-R--	\Device\HarddiskVolumel\WINNT\ system32\ctype.nls
007A0000	映射	8192		-R-- ---	
007A2000	空闲	57 344			
007B0000	私有	524 288	2	-RW-	
007B0000	私有	4096		-RW- ---	
007B1000	保留	520 192		-RW- ---	
00830000	空闲	1 763 311 616			
699D0000	映像	45 056	4	ERWC	C:\WINNT\System32\PSAPI.dll
699D0000	映像	4096		-R-- ---	
699D1000	映像	16 384		ER-- ---	

China-pub.com 下载

(续)

基地址	类型	大小	块	保护属性	描述
699D5000	映像	16 384		-RWC ---	
699D9000	映像	8192		-R-- ---	
699DB000	空闲	238 505 984			
77D50000	映像	450 560	4	ERWC	C:\WINNT\system32\RPCRT4.DLL
77D50000	映像	4096		-R-- ---	
77D51000	映像	421 888		ER-- ---	
77DB8000	映像	4096		-RW- ---	
77DB9000	映像	20 480		-R-- ---	
77DBE000	空闲	8192			
77DC0000	映像	344 064	5	ERWC	C:\WINNT\system32\ ADVAPI32.dll
77DC0000	映像	4096		-R-- ---	
77DC1000	映像	307 200		ER-- ---	
77E0C000	映像	4096		-RW- ---	
77E0D000	映像	4096		-RWC ---	
77E0E000	映像	24 576		-R-- ---	
77E14000	空闲	49 152			
77E20000	映像	401 408	4	ERWC	C:\WINNT\system32\USER32.dll
77E20000	映像	4096		-R-- ---	
77E21000	映像	348 160		ER-- ---	
77E76000	映像	4096		-RW- ---	
77E77000	映像	45 056		-R-- ---	
77E82000	空闲	57 344			
77E90000	映像	720 896	5	ERWC	C:\WINNT\system32\KERNEL32.dll
77E90000	映像	4096		-R-- ---	
77E91000	映像	368 640		ER-- ---	
77EEB000	映像	8192		-RW- ---	
77EED000	映像	4096		-RWC ---	
77EEE000	映像	335 872		-R-- ---	
77F40000	映像	241 664	4	ERWC	C:\WINNT\system32\GDI32.DLL
77F40000	映像	4096		-R-- ---	
77F41000	映像	221 184		ER-- ---	
77F77000	映像	4096		-RW- ---	
77F78000	映像	12 288		--R-- ---	
77F7B000	空闲	20 480			
77F80000	映像	483 328	5	ERWC	C:\WINT\system32\ntdll.dll
77F80000	映像	4096		-R-- ---	
77F81000	映像	299 008		ER-- ---	
77FCA000	映像	8192		-RW- ---	
77FCC000	映像	4096		-RWC ---	
77FCD000	映像	167 936		-R-- ---	
77FF6000	空闲	40 960			
78000000	映像	290 816	6	ERWC	C:\WINNT\system32\MSVCRT.dll
78000000	映像	4096		-R-- ---	
78001000	映像	208 896		ER-- ---	
78034000	映像	32 768		-R-- ---	
7803C000	映像	12 288		-RW- ---	
7803F000	映像	16 384		RWC- ---	

China-pub.com
下载

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
78043000	映像	16 384		-R-- ---	
78047000	空闲	124 424 192			
7F6F0000	映射	1 048 576	2	ER--	
7F6F0000	映射	28 672		ER-- ---	
7F6F7000	保留	1 019 904		ER-- ---	
7F7F0000	空闲	8 126 464			
7FFB0000	映射	147 456	1	-R--	
7FFB0000	映射	147 456		-R-- ---	
7FFD4000	空闲	40 960			
7FFDE000	私有	4096	1	ERW-	
7FFDE000	私有	4096		ERW- ---	
7FFDF000	私有	4096	1	ERW-	
7FFDF000	私有	4096		ERW- ---	
7FFE0000	私有	65 536	2	-R--	
7FFE0000	私有	4096		-R-- ---	
7FFE1000	保留	61 440		-R-- ---	

当然，空闲区域根本不会扩展，因为它们里面没有已经提交的内存页面。每个内存块的行显示4列，下面介绍它们的具体情况。

第一列显示一组页面的地址，这些页面具有相同的状态和保护属性。例如，具有只读保护属性的内存的单个页面（4096字节）被提交的地址是0x77E20000。在地址0x77E21000上，有一个85页（348 160字节）的已提交内存块，它具有执行和读保护特性。如果这两个内存块具有相同的保护属性，那么它们就被组合起来，在内存表中显示为一个86个页面（352 256字节）的项目。

第二列显示的是何种类型的物理存储器支持保留区域中的内存块。这一列中可以出现 5个值中的一个。这 5个值是空闲，私有，映射，映像和保留。如果这个值是私有、映射或映像，则表示内存块是分别受页文件、数据文件或加载的 .exe或DLL文件中的物理存储器支持的。如果这个值是空闲或保留，那么该内存块根本没有任何物理存储器的支持。

大多数情况下，相同类型的物理存储器支持单个区域中的所有提交的内存块。但是单个区域中不同的已提交内存块可以受不同类型的物理存储器的支持。例如，内存映射的文件映像可以受 .exe或DLL文件的支持。如果要在该区域中写入拥有 PAGE_WRITECOPY或PAGE_EXECUTE_WRITECOPY保护属性的单个页面，那么系统就会使你的进程成为一个由页文件而不是文件映像支持的私有页面拷贝。这个新页面拥有的属性将与没有 copy_on_write保护属性的原始页面相同。

第三列显示了地址空间块的大小。一个区域中的所有地址空间块都是相邻的，块与块之间没有任何空隙。

第四列显示保留区域中的块的数量。

第五列显示块的保护属性和保护属性标志。块的保护属性优先于包含该块的区域的保护属性。块使用的保护属性与区域的保护属性相同，但是，与区域不关联的保护属性标志 PAGE_GUARD、PAGE_NOCACHE和PAGE_WRITECOMBINE可以与块相关联。

13.7.2 与Windows 98地址空间的差别

表13-6显示了在Windows 98下运行相同的VMMAP程序时的地址空间表。为了节省篇幅，

China-pub.com 下载

我们没有显示0x80018000至0x85620000之间的虚拟地址。

表13-6 显示Windows 98下地址空间区域内块的示例地址空间表

基地址	类型	大小	块	保护属性	描述
00000000	空闲	4 194 304			
00400000	私有	131 072	6	----	C：\CD\X86\DEBUG\14 VMMAP.EXE
00400000	私有	8192		-R-- ---	
00402000	私有	8192		-RW- ---	
00404000	私有	73 728		-R-- ---	
00416000	私有	8192		-RW- ---	
00418000	私有	8192		-R-- ---	
0041A000	保留	24 576		---- ---	
00420000	私有	1 114 112	4	----	
00420000	私有	20 480		-RW- ---	
00425000	保留	1 028 096		---- ---	
00520000	私有	4096		-RW- ---	
00521000	保留	61440		---- ---	
00530000	私有	65 536	2	-RW-	
00530000	私有	4096		-RW- ---	
00531000	保留	61 440		-RW- ---	
00540000	私有	1 179 648	6	----	线程堆栈
00540000	保留	942 080		---- ---	
00626000	私有	4096		-RW- ---	
00627000	保留	24576		---- ---	
0062D000	私有	4096		---- ---	
0062E000	私有	139 264		-RW- ---	
00650000	保留	65 536		---- ---	
00660000	私有	1 114 112	4	----	
00660000	私有	20 480		-RW- ---	
00665000	保留	1 028 096		---- ---	
00760000	私有	4096		-RW- ---	
00761000	保留	61 440		---- ---	
00770000	私有	1 048 576	2	-RW-	
00770000	私有	32 768		-RW- ---	
00778000	保留	1 015 808		-RW- ---	
00870000	空闲	2 004 418 560			
7800000	私有	262114	3	----	C:\WINDOWS\SYSTEM\MSVCRT.DLL
78000000	私有	188 416		-R-- ---	
7802E000	私有	57 344		-RW- ---	
7803C000	私有	16 384		-R-- ---	
78040000	空闲	133 955 584			
80000000	私有	4096	1	----	
80000000	保留	4096		---- ---	
80001000	私有	4096	1	----	
80001000	私有	4096		-RW- ---	
80002000	私有	4096	1	----	
80002000	私有	4096		-RW- ---	
80003000	私有	4096	1	----	
80003000	私有	4096		-RW- ---	
80004000	私有	65 536	2	----	

China-pub.com
下载

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
80004000	私有	32 768		-RW- ---	
8000C000	保留	32 768		---- ---	
80014000	私有	4096	1	----	
80014000	私有	4096		-RW- ---	
80015000	私有	4096	1	----	
80015000	私有	4096		-RW- ---	
80016000	私有	4096	1	----	
80016000	私有	4096		-RW- ---	
80017000	私有	4096	1	----	
80017000	私有	4096		-RW- ----	
85620000	空闲	9 773 056			
85F72000	私有	151 552	1	----	
85F72000	私有	151 552		-R-- ---	
85F97000	私有	327 680	1	----	
85F97000	私有	327 680		-R-- ---	
85FE7000	空闲	22 052 864			
874EF000	私有	4194304	1	----	
874EF000	保留	4 194 304		---- ---	
878EF000	空闲	679 219 200			
B00B0000	私有	880 640	3	----	
B00B0000	私有	233 472		-R-- ---	
B00E9000	私有	20 480		-RW- ---	
B00EE000	私有	626 288		-R-- ---	
B0187000	空闲	177 311 744			
BAAA0000	私有	315 392	7	----	
BAAA0000	私有	4096		-R-- ---	
BAAA1000	私有	4096		-RW- ---	
BAAA2000	私有	241 664		-R-- ---	
BAADD000	私有	4096		-RW- ---	
BAADE000	私有	4096		-R-- ---	
BAADF000	私有	32 768		-RW- ---	
BAAE7000	私有	24 576		-R-- ---	
BAAED000	空闲	86 978 560			
BFDE0000	私有	20 480	1	----	
BFDE0000	私有	20 480		-R-- ---	
BFDE5000	空闲	45 056			
BFDF0000	私有	65 536	3	----	
BFDF0000	私有	40 960		-R-- ---	
BFDFA000	私有	4096		-RW- ---	
BFDFB000	私有	20 480		-R-- ---	
BFE00000	空闲	131 072			
BFE20000	私有	16 384	3	----	
BFE20000	私有	8192		-R-- ---	
BFE22000	私有	4096		-RW- ---	
BFE23000	私有	4096		-R-- ---	
BFE24000	空闲	245 760			
BFE60000	私有	24 576	3	----	

China-pub.com 下载

(续)

基地址	类型	大小	块	保护属性	描述
BFE60000	私有	8192		-R-- ---	
BFE62000	私有	4096		-RW- ---	
BFE63000	私有	12 288		-R-- ---	
BFE66000	空闲	40 960			
BFE70000	私有	24 576	3	----	
BFE70000	私有	8192		-R-- ---	
BFE72000	私有	4096		-RW- ---	
BFE73000	私有	12 288		-R-- ---	
BFE76000	空闲	40 960			
BFE80000	私有	65 536	3	----	C:\WINDOWS\SYSTEM\ADVAPI32.DLL
BFE80000	私有	49 152		-R-- ---	
BFE8C000	私有	4096		-RW- ---	
BFE8D000	私有	12 288		-R-- ---	
BFE90000	私有	573 440	3	----	
BFE90000	私有	425 984		-R-- ---	
BFEF8000	私有	4096		-RW- ---	
BFEF9000	私有	143 360		-R-- ---	
BFF1C000	空闲	16 384			
BFF20000	私有	155 648	5	----	C：\WINDOWS\SYSTEM\GDI32.DLL
BFF20000	私有	126 976		-R-- ---	
BFF3F000	私有	8192		-RW- ---	
BFF41000	私有	4096		-R-- ---	
BFF42000	私有	4096		-RW- ---	
BFF43000	私有	12 288		-R-- ---	
BFF46000	空闲	40 960			
BFF50000	私有	69 632	3	----	C:\WINDOWS\SYSTEM\USER32.DLL
BFF50000	私有	53 248		-R-- ---	
BFF5D000	私有	4096		-RW- ---	
BFF5E000	私有	12 288		-R-- ---	
BFF61000	空闲	61 440			
BFF70000	私有	585 728	5	----	C:\WINDOWS\SYSTEM\KERNEL32.DLL
BFF70000	私有	352 256		-R-- ---	
BFFC6000	保留	12 288		---- ---	
BFFC9000	私有	16 384		-RW- ---	
BFFCD000	私有	90 112		-R-- ---	
BFFE3000	保留	114 688		---- ---	
BFFFF000	空闲	4096			

两个地址空间表的最大不同是在Windows 98下缺少了某些的信息。例如，每个区域和块能反映出地址空间的区域是空闲、保留还是私有的。你决不会看到映射或者映像之类的字样，因为Windows 98没有提供更多的信息来指明支持该区域的物理存储器的是个内存映射文件还是包含在.exe或DLL中的文件映像。

你会发现大多数地址空间区域的大小是分配粒度（64KB）的倍数。如果包含在地址空间区域中的块的大小不是分配粒度的倍数，那么在地址空间区域的结尾处常常有一个保留的地址空间块。这个地址空间块的大小必须使得地址空间区域能够符合分配粒度边界（64KB）倍数的要求。例如，从地址0x00530000开始的地址空间区域包含两个地址块，一个是4 KB的已提

China-pub.com
下载

交内存块，另一个是占用60 KB内存地址范围的已保留的地址块。

最后，保护标志从来不反映执行或copy-on-write访问权，因为Windows 98不支持这些标志。它也不支持 3个保护属性标志，即 PAGE_NOCACHE、PAGE_WRITECOMBINE和PAGE_GUARD。由于不支持PAGE_GUARD标志，因此VMMap使用更加复杂的技术来确定是否已经为线程的堆栈保留了地址空间区域。

你将注意到，与Windows 2000不同，在Windows 98中，0x80000000至0xBFFFFFFF之间的地址空间区域是可以查看的。这个分区包含了所有 32位应用程序共享的地址空间。如你所见，有4个系统DLL被加载了这个地址空间区域，可以供所有进程使用。

13.8 数据对齐的重要性

本节不再讨论进程的虚拟地址空间问题，而是要介绍数据对齐的重要性。数据对齐并不是操作系统的内存结构的一部分，而是CPU结构的一部分。

当CPU访问正确对齐的数据时，它的运行效率最高。当数据大小的数据模数的内存地址是0时，数据是对齐的。例如，WORD值应该总是从被2除尽的地址开始，而DWORD值应该总是从被4除尽的地址开始，如此等等。当 CPU试图读取的数据值没有正确对齐时， CPU可以执行两种操作之一。即它可以产生一个异常条件，也可以执行多次对齐的内存访问，以便读取完整的未对齐数据值。

下面是访问未对齐数据的某个代码：

```
VOID SomeFunc(PVOID pvDataBuffer) {

   // The first byte in the buffer is some byte of information
   char c = * (PBYTE) pvDataBuffer;

   // Increment past the first byte in the buffer
   pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

   // Bytes 2-5 contain a double-word value
   DWORD dw = * (DWORD *) pvDataBuffer;

   // The line above raises a data misalignment exception on the Alpha
   .
   .
   .
}
```

显然，如果CPU执行多次内存访问，应用程序的运行速度就会放慢。在最好的情况下，系统访问未对齐的数据所需要的时间将是访问对齐数据的时间的两倍，不过在有些情况下，访问时间可能更长。为了使应用程序获得最佳的运行性能，编写的代码必须使数据正确地对齐。

下面让我们更加深入地说明x86 CPU是如何进行数据对齐的。 X86 CPU的EFLAGS寄存器中包含一个特殊的位标志，称为 AC（对齐检查的英文缩写）标志。按照默认设置，当 CPU首次加电时，该标志被设置为0。当该标志是0时，CPU能够自动执行它应该执行的操作，以便成功地访问未对齐的数据值。然而，如果该标志被设置为 1，每当系统试图访问未对齐的数据时，CPU就会发出一个INT 17H中断。x86的Windows 2000和Windows 98版本从来不改变这个CPU标志位。因此，当应用程序在 x86处理器上运行时，你根本看不到应用程序中出现数据未对齐的异常条件。

现在让我们来看一看 Alpha CPU的情况。Alpha CPU不能自动处理对未对齐数据的访问。

China-pub.com 下载

当未对齐的数据访问发生时，CPU就会将这一情况通知操作系统。这时，Windows 2000将会确定它是否应该引发一个数据未对齐异常条件。它也可以执行一些辅助指令，对问题默默地加以纠正，并让你的代码继续运行。按照默认设置，当在Alpha计算机上安装Windows 2000时，操作系统会对未对齐数据的访问默默地进行纠正。然而，可以改变这个行为特性。当引导Windows 2000时，系统就会在注册表中查找的这个关键字：

```
HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Session Manager
```

在这个关键字中，可能存在一个值，称为EnableAlignmentFaultExceptions。如果这个值不存在（这是通常的情况），Windows 2000会默默地处理对未对齐数据的访问。如果存在这个值，系统就能获取它的相关数据值。如果数据值是0，系统会默默地进行访问的处理。如果数据值是1，系统将不执行默默的处理，而是引发一个未对齐异常条件。几乎从来都不需要修改该注册表值的数据值，因为如果修改有些应用程序能够引发数据未对齐的异常条件并终止运行。

为了更加容易地修改该注册表项。Alpha处理器上运行的Microsoft Visual C++版本包含了一个小型实用程序AXPAlign.exe。AXPAlign的用法如下面所示：

```
Alpha AXP alignment fault exception control

Usage: axpalign [option]

Options:
   /enable   to enable alignment fault exceptions.
   /disable  to disable alignment fault exceptions.
   /show     to display the current alignment exception setting.

Enable alignment fault exceptions:
   In this mode any aligned access to unaligned data will result in a
   data misalignment exception and no automatic operating system fixups
   will occur. The application may be terminated or a debugger can be
   used to locate the source of alignment faults in your code.

   This setting applies to all running processes and thus care should
   be taken since older applications may get exceptions and terminate.

   Note that SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT) can be used to
   suppress alignment exceptions even in this mode.

Disable alignment fault exceptions:
   This is the default mode with Windows NT for Alpha AXP, versions 3.1
   and 3.5. In this mode the operating system will fixup any misaligned
   data accesses should they occur and applications or debuggers will
   not see them. This may lead to performance degradation if it occurs
   at a high rate. Perfmon or wperf may be used to monitor the rate.
```

该实用程序只是修改注册表值的状态，或者显示值的当前状态。当用该实用程序修改数据值后，必须重新引导操作系统，使所做的修改生效。

如果不使用AXPAlign实用程序，仍然可以让系统为进程中的所有线程默默地纠正对未对齐数据的访问，方法是让进程的线程调用SetErrorMode函数：

```
UINT SetErrorMode(UINT fuErrorMode);
```

就我们的讨论来说，需要说明的标志是SEM_NOALIGNMENTFAULTEXCEPT标志。当该标志设定后，系统会将自动纠正对未对齐数据的访问。当该标志重新设置时，系统将不纠正对未对齐数据的访问，而是引发数据未对齐异常条件。注意，修改该标志将会影响拥有调用该函

China-pub.com
下载

数的线程的进程中包含的所有线程。换句话说，改变该标志不会影响其他进程中的任何线程。还要注意，进程的错误方式标志是由所有的子进程继承的。因此，在调用 CreateProcess函数之前，必须临时重置该标志（不过通常不必这样做）。

当然，无论在哪个 CPU平台上运行，都可以调用 SetErrorMode函数，传递 SEM_NOALIGNMENTFAULTEXCEPT标志。但是，结果并不总是相同。如果是 x86系统，该标志总是打开的，并且不能被关闭。如果是 Alpha系统，那么只有当 EnableAlignmentFault Exceptions注册表值被设置为 1时，才能关闭该标志。

可以使用 Windows 2000的 MMC Performance Monitor来查看每秒钟系统执行多少次数据对齐的调整修改。图 13-4显示了在将该计数器添加给图表之前，Add Counter（添加计数器）对话框是个什么样子。

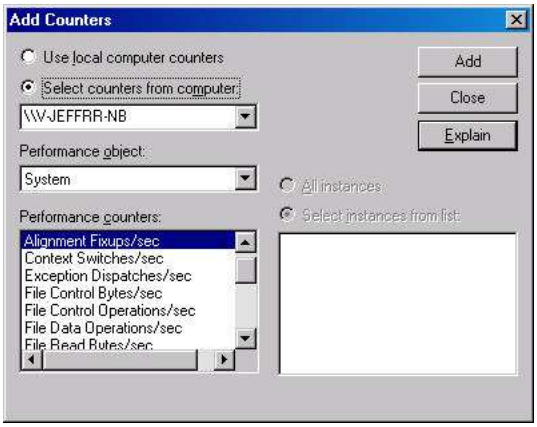


图13-4 Add Counters 对话框

该计数器显示的是每秒钟 CPU通知操作系统的未对齐数据访问的次数。如果监视 x86计算机上的这个计数器，你会看到它总是报告每秒钟为 0次数据对齐的调整。这是因为 x86 CPU本身正在进行调整，因此没有通知操作系统。由于是 x86 CPU而不是操作系统来进行这种调整，因此访问 x86计算机上的未对齐数据对性能产生的影响并不像需要用软件（Windows 2000操作系统代码）来进行数据对齐调整的 CPU那样大。

可以看到，只需要调用 SetErrorMode函数便足以使你的应用程序能够正确运行。但是这个解决方案肯定不是效率最高的方案。实际上，Digital 出版社出版的《Alpha Architecture Reference Manual》手册上讲到，系统默默纠正未对齐数据访问的仿真代码运行时所花费的时间相当于普通情况下的 100倍。这是个相当大的开销。不过有一种更加有效的解决方案可以解决你的问题。

Microsoft用于 Alpha CPU的 C/C++编译器支持一个特殊的关键字，称为 __unaligned。可以像使用 const或 volatile修改符那样使用 __unaligned修改符，差别在于 __unaligned修改符只有在用于指针变量时才起作用。当通过未对齐指针来访问数据时，编译器就会生成一个代码，该代码假设数据没有正确对齐，因此添加一些访问数据时必须使用的辅助 CPU指令。下面显示的代码是前面已经讲过的代码的修改版。这个新版本利用了关键字 __unaligned。

```
VOID SomeFunc(PVOID pvDataBuffer) {

   // The first byte in the buffer is some byte of information
   char c = * (PBYTE) pvDataBuffer;
```

China-pub.com 下载

```
   // Increment past the first byte in the buffer
   pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

   // Bytes 2-5 contain a double-word value
   DWORD dw = * (__unaligned DWORD *) pvDataBuffer;

   // The line above causes the compiler to generate additional
   // instructions so that several aligned data accesses are performed
   // to read the DWORD.
   // Note that a data misalignment exception is not raised.

   .
   .
   .
}
```

当我对Alpha计算机上运行的下面这行代码进行编译时，生成了7个CPU指令：

```
DWORD dw = * (__unaligned DWORD *) pvDataBuffer;
```

然而，如果我从这行代码中删除__unaligned关键字并进行编译，那么只生成3个CPU指令。可以看到，在Alpha CPU上使用__unaligned关键字，生成的CPU指令要多两倍以上。编译器添加的指令比CPU跟踪未对齐数据的访问并让操作系统来纠正未对齐数据的效率要高得多。实际上，如果监控Alignment Fixup/sec计数器，你将发现通过未对齐指针进行的访问对图表上显示的数据没有什么影响。

最后要说明的是，x86 CPU上运行的Visual C/C++编译器不支持__unaligned关键字。我想Microsoft公司也许认为这没有必要，因为CPU本身进行未对齐数据的纠正速度很快。但是，这也意味着x86编译器在遇到__unaligned关键字时就会产生错误。因此，如果打算为应用程序创建单个基本源代码，就必须使用UNALIGNED宏，而不是__unaligned关键字。在WinNT.h文件中，UNALIGNED宏定义为下面的形式：

```
#if defined(_M_MRX000) || defined(_M_ALPHA) || defined(_M_IA64)
#define UNALIGNED __unaligned
#if defined(_WIN64)
#define UNALIGNED64 __unaligned
#else
#define UNALIGNED64
#endif
#else
#define UNALIGNED
#define UNALIGNED64
#endif
```

China-pub.com
下载

第14章 虚拟内存

上一章介绍了系统如何管理虚拟内存，每个进程如何获得它自己的私有地址空间，进程的地址空间是个什么样子等内容。这一章不再介绍抽象的概念，而要具体介绍几个Windows函数，这些函数能够提供关于系统内存管理以及进程中的虚拟地址空间等信息。

14.1 系统信息

许多操作系统的值是根据主机而定的，比如页面的大小，分配粒度的大小等。这些值决不应该用硬编码的形式放入你的源代码。相反，你始终都应该在进程初始化的时候检索这些值，并在你的源代码中使用检索到的值。GetSystemInfo函数将用于检索与主机相关的值：

```
VOID GetSystemInfo(LPSYSTEM_INFO psinf);
```

必须传递SYSTEM_INFO结构的地址给这个函数。这个函数将初始化所有的结构成员然后返回。下面是SYSTEM_INFO数据结构的样子。

```
typedef struct _SYSTEM_INFO {
   union {
      DWORD dwOemId;   // Obsolete, do not use
      struct {
         WORD wProcessorArchitecture;
         WORD wReserved;
      };
   };
   DWORD     dwPageSize;
   LPVOID    lpMinimumApplicationAddress;
   LPVOID    lpMaximumApplicationAddress;
   DWORD_PTR dwActiveProcessorMask;
   DWORD     dwNumberOfProcessors;
   DWORD     dwProcessorType;
   DWORD     dwAllocationGranularity;
   WORD      wProcessorLevel;
   WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

当系统引导时，它要确定这些成员的值是什么。对于任何既定的系统来说，这些值总是相同的，因此决不需要为任何既定的进程多次调用该函数。由于有了 GetSystemInfo函数，因此应用程序能够在运行的时候查询这些值。在该结构的所有成员中，只有 4个成员与内存有关。表14-1对这4个成员作了描述。

表14-1 与内存有关的成员函数

成 员 名	描 述
dwPageSize	用于显示CPU的页面大小。在x86 CPU上，这个值是4096字节。在Alpha CPU上，这个值是8192字节。在IA-64上，这个值是8192字节
lpMinimumApplicationAddress	用于给出每个进程的可用地址空间的最小内存地址。在Windows 98上，这个值是4 194 304，或0x00400000，因为每个进程的地址空间中下面的4MB是不能使用的。在Windows 2000上，这个值是65 536或0x00010000，因为每个进程的地址空间中开头的64KB总是空闲的

China-pub.com 下载

(续)

成 员 名	描 述
lpMaximumApplicationAddress	用于给出每个进程的可用地址空间的最大内存地址。在Windows 98上，这个地址是2 147 483 647或0x7FFFFFFF，因为共享内存映射文件区域和共享操作系统代码包含在上面的2 GB分区中。在Windows 2000上，这个地址是内核方式内存开始的地址，它不足64KB
dwAllocationGranularity	显示保留的地址空间区域的分配粒度。截止到撰写本书时，在所有Windows平台上，这个值都是65 536

该结构的其他成员与内存管理毫无关系，为了完整起见，下面也对它们进行了介绍（见表14-2）。

表14-2 与内存无关的成员函数

成 员 名	描 述
dwOemId	已作废，不引用
WRederved	保留供将来使用，不引用
dwNumberOfProcessors	用于指明计算机中的CPU数目
dwActiveProcessorMask	一个位屏蔽，用于指明哪个CPU是活动的（允许运行线程）
dwProcessorType	只用于Windows 98，不用于Windows 2000，用于指明处理器的类型，如Intel 386、486或Pentium
wProcessorArchitecture	只用于Windows 2000，不用于Windows 98，用于指明处理的结构，如Intel、Alpha、Intel 64位或Alpha 64位
wProcessorLevel	只用于Windows 2000，不用于Windows 98，用于进一步细分处理器的结构，如用于设定Intel Pentium Pro或Pentium II
wProcessorRevision	只用于Windows 2000，不用于Windows 98，用于进一步细分处理器的级别

系统信息示例应用程序

清单14-1显示的SysInfo应用程序（“14 SysInfo.exe”）是一个简单的调用GetSystemInfo函

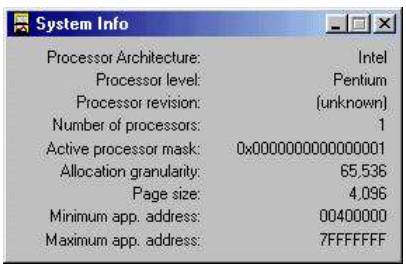


图14-1 在x86 CPU上运行Windows 98时返回的结果

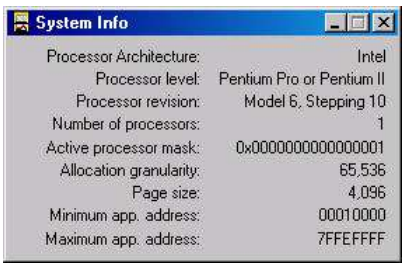


图14-2 在x86 CPU上运行32位Windows 2000时返回的结果

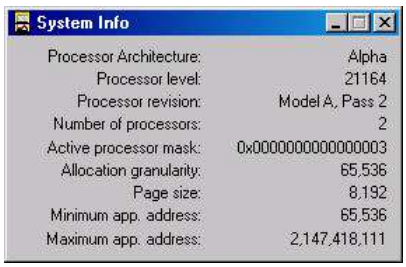


图14-3 在Alpha CPU上运行32位Windows 2000时返回的结果

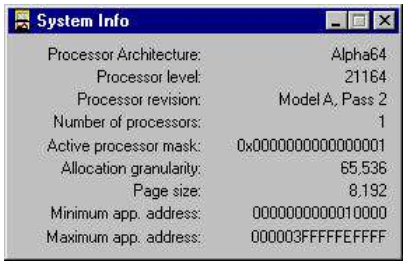


图14-4 在Alpha CPU上运行64位Windows时返回的结果

China-pub.com
下载

数的程序，它显示SYSTEM_INFO结构中返回的信息。该应用程序的源代码和资源文件均位于本书所附光盘上的14-SysInfo目录下。图14-1、图14-2、图14-3和图14-4所示的两个对话框显示了在若干不同平台上运行的SysInfo应用程序返回的结果。

清单14-1 SysInfo应用程序



SysInfo.cpp

```
/******************************************************************************
Module:  SysInfo.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////


// Set to TRUE if the app is running on Windows 9x.
BOOL g_fWin9xIsHost = FALSE;

//////////////////////////////////////////////////////////////////////////////


// This function accepts a number and converts it to a
// string, inserting commas where appropriate.
PTSTR BigNumToString(LONG lNum, PTSTR szBuf) {

   TCHAR szNum[100];
   wsprintf(szNum, TEXT("%d"), lNum);
   NUMBERFMT nf;
   nf.NumDigits = 0;
   nf.LeadingZero = FALSE;
   nf.Grouping = 3;
   nf.lpDecimalSep = TEXT(".");
   nf.lpThousandSep = TEXT(",");
   nf.NegativeOrder = 0;
   GetNumberFormat(LOCALE_USER_DEFAULT, 0, szNum, &nf, szBuf, 100);
   return(szBuf);
}


//////////////////////////////////////////////////////////////////////////////


void ShowCPUInfo(HWND hwnd, WORD wProcessorArchitecture, WORD wProcessorLevel,
```

China-pub.com 下载

```
WORD wProcessorRevision) {

TCHAR szCPUArch[64]  = TEXT("(unknown)");
TCHAR szCPULevel[64] = TEXT("(unknown)");
TCHAR szCPURev[64]   = TEXT("(unknown)");

switch (wProcessorArchitecture) {
   case PROCESSOR_ARCHITECTURE_INTEL:
      lstrcpy(szCPUArch, TEXT("Intel"));
      switch (wProcessorLevel) {
      case 3: case 4:
         wsprintf(szCPULevel, TEXT("80%c86"), wProcessorLevel + '0');
         if (!g_fWin9xIsHost)
            wsprintf(szCPURev, TEXT("%c%d"),
            HIBYTE(wProcessorRevision) + TEXT('A'),
            LOBYTE(wProcessorRevision));
         break;

      case 5:
         wsprintf(szCPULevel, TEXT("Pentium"));
         if (!g_fWin9xIsHost)
            wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
               HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
         break;
         case 6:
            wsprintf(szCPULevel, TEXT("Pentium Pro or Pentium II"));
            if (!g_fWin9xIsHost)
               wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
                  HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
            break;
         }
         break;

      case PROCESSOR_ARCHITECTURE_ALPHA:
         lstrcpy(szCPUArch, TEXT("Alpha"));
         wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
         wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'),
            LOBYTE(wProcessorRevision));
         break;

      case PROCESSOR_ARCHITECTURE_IA64:
         lstrcpy(szCPUArch, TEXT("IA-64"));
         wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
         wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'),
            LOBYTE(wProcessorRevision));
         break;

      case PROCESSOR_ARCHITECTURE_ALPHA64:
         lstrcpy(szCPUArch, TEXT("Alpha64"));
         wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
         wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'),
            LOBYTE(wProcessorRevision));
```

China-pub.com
下载


```
            break;

         case PROCESSOR_ARCHITECTURE_UNKNOWN:
         default:
            wsprintf(szCPUArch, TEXT("Unknown"));
            break;
      }
      SetDlgItemText(hwnd, IDC_PROCARCH,  szCPUArch);
      SetDlgItemText(hwnd, IDC_PROCLEVEL, szCPULevel);
      SetDlgItemText(hwnd, IDC_PROCREV,   szCPURev);
   }
//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SYSINFO);

   SYSTEM_INFO sinf;
   GetSystemInfo(&sinf);

   if (g_fWin9xIsHost) {
      sinf.wProcessorLevel = (WORD) (sinf.dwProcessorType / 100);
   }

   ShowCPUInfo(hwnd, sinf.wProcessorArchitecture,
      sinf.wProcessorLevel, sinf.wProcessorRevision);

   TCHAR szBuf[50];
   SetDlgItemText(hwnd, IDC_PAGESIZE,
      BigNumToString(sinf.dwPageSize, szBuf));

   _stprintf(szBuf, TEXT("%p"), sinf.lpMinimumApplicationAddress);
   SetDlgItemText(hwnd, IDC_MINAPPADDR, szBuf);

   _stprintf(szBuf, TEXT("%p"), sinf.lpMaximumApplicationAddress);
   SetDlgItemText(hwnd, IDC_MAXAPPADDR, szBuf);

   _stprintf(szBuf, TEXT("0x%016I64X"), (__int64) sinf.dwActiveProcessorMask);
   SetDlgItemText(hwnd, IDC_ACTIVEPROCMASK, szBuf);

   SetDlgItemText(hwnd, IDC_NUMOFPROCS,
      BigNumToString(sinf.dwNumberOfProcessors, szBuf));

   SetDlgItemText(hwnd, IDC_ALLOCGRAN,
      BigNumToString(sinf.dwAllocationGranularity, szBuf));

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
```

China-pub.com 下载

```
         EndDialog(hwnd, id);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hDlg, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   OSVERSIONINFO vi = { sizeof(vi) };
   GetVersionEx(&vi);
   g_fWin9xIsHost = (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS);

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SYSINFO), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

SysInfo.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
```

China-pub.com
下载


```
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//

// Dialog
//

IDD_SYSINFO DIALOG DISCARDABLE  18, 18, 186, 97
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "System Info"
FONT 8, "MS Sans Serif"
BEGIN
   RTEXT           "Processor Architecture:",IDC_STATIC,4,4,88,8,
                    SS_NOPREFIX
   RTEXT           "ID_PROCARCH",IDC_PROCARCH,96,4,84,8,SS_NOPREFIX
   RTEXT           "Processor level:",IDC_STATIC,4,14,88,8,SS_NOPREFIX
   RTEXT           "ID_PROCLEVEL",IDC_PROCLEVEL,96,14,84,8,SS_NOPREFIX
   RTEXT           "Processor revision:",IDC_STATIC,4,24,88,8,SS_NOPREFIX
   RTEXT           "ID_PROCREV",IDC_PROCREV,96,24,84,8,SS_NOPREFIX
   RTEXT           "Number of processors:",IDC_STATIC,4,34,88,8,SS_NOPREFIX
   RTEXT           "ID_NUMOFPROCS",IDC_NUMOFPROCS,96,34,84,8,SS_NOPREFIX
   RTEXT           "Active processor mask:",IDC_STATIC,4,44,88,8,
                    SS_NOPREFIX
   RTEXT           "ID_ACTIVEPROCMASK",IDC_ACTIVEPROCMASK,96,44,84,8,
                    SS_NOPREFIX
   RTEXT           "Allocation granularity:",IDC_STATIC,4,54,88,8,
                    SS_NOPREFIX
   RTEXT           "ID_ALLOCGRAN",IDC_ALLOCGRAN,96,54,84,8,SS_NOPREFIX
   RTEXT           "Page size:",IDC_STATIC,4,64,88,8,SS_NOPREFIX
```

China-pub.com 下载

```
    RTEXT           "ID_PAGESIZE",IDC_PAGESIZE,96,64,84,8,SS_NOPREFIX
    RTEXT           "Minimum app. address:",IDC_STATIC,4,74,88,8,SS_NOPREFIX
    RTEXT           "ID_MINAPPADDR",IDC_MINAPPADDR,96,74,84,8,SS_NOPREFIX
    RTEXT           "Maximum app. address:",IDC_STATIC,4,84,88,8,SS_NOPREFIX
    RTEXT           "ID_MAXAPPADDR",IDC_MAXAPPADDR,96,84,84,8,SS_NOPREFIX
END


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SYSINFO             ICON    DISCARDABLE     "SysInfo.Ico"

/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_SYSINFO, DIALOG
    BEGIN
        RIGHTMARGIN, 170
        BOTTOMMARGIN, 77
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

14.2 虚拟内存的状态

Windows函数GlobalMemoryStatus可用于检索关于当前内存状态的动态信息：

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS pmst);
```

我认为这个函数的名字起的并不好，因为 GlobalMemoryStatus表示这个函数多少与 16位

China-pub.com
下载

Windows中的全局堆栈有些相关。我认为GlobalMemoryStatus应该改为VirtualMemoryStatus比较好些。

当调用GlobalMemoryStatus时，必须传递一个MEMORYSTATUS结构的地址。下面显示了MOMORYSTATUS的数据结构。

```
typedef struct _MEMORYSTATUS {
   DWORD dwLength;
   DWORD dwMemoryLoad;
   SIZE_T dwTotalPhys;
   SIZE_T dwAvailPhys;
   SIZE_T dwTotalPageFile;
   SIZE_T dwAvailPageFile;
   SIZE_T dwTotalVirtual;
   SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

在调用GlobalMemoryStatus之前，必须将dwLength成员初始化为用字节表示的结构的大小，即一个MEMORYSTATUS结构的大小。这个初始化操作使得Microsoft能够将成员添加给将来的Windows版本中的这个结构，而不会破坏现有的应用程序。当调用GlobalMemoryStatus时，它将对该结构的其余成员进行初始化并返回。下一节中的VMStat示例应用程序将要描述各个成员及其含义。

如果希望应用程序在内存大于4GB的计算机上运行，或者合计交换文件的大小大于4GB，那么可以使用新的GlobalMemoryStatusEx函数：

```
BOOL GlobalMemoryStatusEx(LPMEMORYSTATUSEX pmst);
```

必须给该函数传递新的MEMORYSTATUSEX结构的地址：

```
typedef struct _MEMORYSTATUSEX {
   DWORD dwLength;
   DWORD dwMemoryLoad;
   DWORDLONG ullTotalPhys;
   DWORDLONG ullAvailPhys;
   DWORDLONG ullTotalPageFile;
   DWORDLONG ullAvailPageFile;
   DWORDLONG ullTotalVirtual;
   DWORDLONG ullAvailVirtual;
   DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

这个结构与原先的MEMORYSTATUS结构基本相同，差别在于新结构的所有成员的大小都是64位宽，因此它的值可以大于4GB。最后一个成员是ullAvailExtendedVirtual，用于指明在调用进程的虚拟地址空间的极大内存（VLM）部分中未保留内存的大小。该VLM部分只适用于某些配置中的某些CPU结构。

虚拟内存状态示例应用程序

清单14-2列出的VMStat应用程序显示了一个简单的对话框，用于列出调用GlobalMemoryStatus函数的结果。对话框中的信息每秒钟更新一次，因此，你可以在对系统中的其他进程进行操作时，使应用程序继续运行。该应用程序的源代码和资源文件均位于本书所附光盘上的14-VMStat目录下。图14-5显示了在内存为128 MB的Intel Pentium II计算机上的Windows 2000下运行该程序的结果。

dwMemoryLoad成员（图14-5中显示为Memory Load）给出了内存管理系统的大致繁忙程

China-pub.com 下载

度。该数字的范围是0至100。计算这个值时使用的具体算法在Windows 98与Windows 2000上是不同的。根据将来的操作系统版本的情况，该算法还会有所变化。实际上，该成员变量报告的值是没有什么用处的。



图14-5 VMStat 运行结果显示

dwTotalPhys成员（图14-5中显示为TotalPhys）用于指明存在的物理存储器（EAM）的总字节数。在128 MB的Pentium II计算机上，这个值是133 677 056，它只比128 MB少540 672个字节。GlobalMemoryStatus之所以不报告全部的128MB，原因是在引导进程中，系统将一些内存保留为非页面内存池。这些内存甚至不能被内核使用。dwAvailPhys成员（图14-5中显示为AvailPyhs）用于指明可供分配的物理存储器的总字节数。

dwTotalPageFile成员（图14-5中显示为TotalPageFile）用于指明你的硬盘上的页文件中包含的最大字节数。虽然VMStat报告的页文件当前是318 574 592字节，但是系统可以根据需要对页交换文件进行扩大和压缩。dwAvailPageFile成员（图14-5中显示为AvailPageFile）用于指明页文件中有233 046 016字节尚未提交给任何进程，因此，如果一个进程决定提交任何私有内存的话，目前就可以使用这些字节。

dwTotalVirtual成员（图14-5中显示为TotalVitual）用于指明每个进程的地址空间中私有的总字节数。它的值是2 147 352 576，比准确的2 GB少128 KB。从0x00000000至0x0000FFFF以及从0x7FFF0000至0x7FFFFFFF的两个分区是不能访问的地址空间，这正好等于128 KB这个差额。如果在Windows 98下运行VMStat，你将看到dwTotalVirtual返回的值是2 143 289 344，这比准确的2 GB只少4 MB。之所以差4 MB，原因是系统决不允许应用程序访问从0x00000000至0x003FFFFF之间的这个4 MB分区。

最后一个成员dwAvailVirtual（图14-5中显示为AvailVirtual）是该结构中专门用于调用GlobalMemoryStatus的进程的唯一成员，所有其他成员在系统中的使用情况都是一样的，而不管是哪个进程调用GlobalMemoryStatus。若要计算这个值，GlobalMemoryStatus将调用进程的地址空间中的所有空闲区域相加。dwAvailVirtual的值2 136 846 336表示可供VMStat随意使用的空闲地址空间的数量。如果将dwTotalVirtual的值减去dwAvailVirtual的值，你会看到VMStat在它的虚拟地址空间中保留了10 506 240个字节。

没有一个成员能够指明进程当前使用的物理存储器的数量。

清单14-2 VMStat应用程序

VMStat.cpp

```
/******************************************************************************
Module:  VMStat.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////

// The update timer's ID
#define IDT_UPDATE   1

//////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_VMSTAT);

   // Set a timer so that the information updates periodically.
   SetTimer(hwnd, IDT_UPDATE, 1 * 1000, NULL);

   // Force a timer message for the initial update.
   FORWARD_WM_TIMER(hwnd, IDT_UPDATE, SendMessage);
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnTimer(HWND hwnd, UINT id) {

   // Initialize the structure length before calling GlobalMemoryStatus.
   MEMORYSTATUS ms = { sizeof(ms) };
   GlobalMemoryStatus(&ms);

   TCHAR szData[512] = { 0 };
   _stprintf(szData, TEXT("%d\n%d\n%I64d\n%I64d\n%I64d\n%I64d\n%I64d"),
      ms.dwMemoryLoad, ms.dwTotalPhys,
      (__int64) ms.dwAvailPhys,     (__int64) ms.dwTotalPageFile,
      (__int64) ms.dwAvailPageFile, (__int64) ms.dwTotalVirtual,
      (__int64) ms.dwAvailVirtual);
   SetDlgItemText(hwnd, IDC_DATA, szData);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         KillTimer(hwnd, IDT_UPDATE);
         EndDialog(hwnd, id);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_TIMER,      Dlg_OnTimer);
```

China-pub.com 下载

```
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMSTAT), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

VMStat.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
```

China-pub.com
下载


```
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VMSTAT DIALOG DISCARDABLE  60, 60, 132, 66
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "VMStat"
FONT 8, "MS Sans Serif"
BEGIN
   LTEXT           "Memory load:\nTotalPhys:\nAvailPhys:\nTotalPageFile:\n\
                   AvailPageFile:\nTotalVirtual:\nAvailVirtual:",
                   IDC_STATIC,4,4,51,56
   RTEXT           "Memory load:\nTotalPhys:\nAvailPhys:\nTotalPageFile:\n\
                   AvailPageFile:\nTotalVirtual:\nAvailVirtual:",
                   IDC_DATA,60,4,68,56
END
/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMSTAT              ICON    DISCARDABLE     "VMStat.Ico"

/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
   IDD_VMSTAT, DIALOG
   BEGIN
      RIGHTMARGIN, 108
      BOTTOMMARGIN, 65
   END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

14.3 确定地址空间的状态

Windows提供了一个函数，可以用来查询地址空间中内存地址的某些信息（如大小，存储器类型和保护属性等）。实际上本章后面显示的VMMap示例应用程序就使用这个函数来生成第13章所附的虚拟内存表交换信息。这个函数称为VirtualQuery：

```
DWORD VirtualQuery(
   LPCVOID pvAddress,
   PMEMORY_BASIC_INFORMATION pmbi,
   DWORD dwLength);
```

Windows还提供了另一个函数，它使一个进程能够查询另一个进程的内存信息：

```
DWORD VirtualQueryEx(
   HANDLE hProcess,
   LPCVOID pvAddress,
   PMEMORY_BASIC_INFORMATION pmbi,
   DWORD dwLength);
```

这两个函数基本相同，差别在于使用VirtualQueryEx时，可以传递你想要查询的地址空间信息的进程的句柄。调试程序和其他实用程序使用这个函数最多，几乎所有的应用程序都只需要调用VirtualQuery函数。当调用VirtualQuery（Ex）函数时，pvAddress参数必须包含你想要查询其信息的虚拟内存地址。Pmbi参数是你必须分配的MEMORY_BASIC_INFORMATION结构的地址。该结构在WinNT.h文件中定义为下面的形式：

```
typedef struct _MEMORY_BASIC_INFORMATION {
   PVOID BaseAddress;
   PVOID AllocationBase;
   DWORD AllocationProtect;
   SIZE_T RegionSize;
   DWORD State;
   DWORD Protect;
   DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

最后一个参数是dwLength，用于设定MEMORY_BASIC_INFORMATION结构的大小。VirtualQuery（Ex）函数返回拷贝到缓存中的字节的数量。

根据在pvAddress参数中传递的地址，VirtualQuery（Ex）函数将关于共享相同状态、保护属性和类型的相邻页面的范围信息填入MEMORY_BASIC_INFORMATION结构中。表14-3描述了该结构的成员。

China-pub.com
下载

表14-3 MEMORY_BASIC_INFORMATION结构的成员函数

成 员 名	描 述
BaseAddress	与pvAddress参数的值相同，但是四舍五入为页面的边界值
AllocationBase	用于指明包含在pvAddress参数中设定的地址区域的基地址
AllocationProtect	用于指明一个地址空间区域被初次保留时赋予该区域的保护属性
RegionSize	用于指明从基地址开始的所有页面的大小（以字节为计量单位） 这些页面与含有用pvSddress参数设定的地址的页面拥有相同的保护属性、状态和类型
State	用于指明所有相邻页面的状态（MEM_FREE、MEM_RESERVE或MEM_COMMIT）。这些页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型 如果它的状态是空闲，那么AllocationBase、AllocationProtect、Protect和Type等成员均未定义 如果状态是MEM_RESERVE，则Protect成员未定义
Protect	用于指明所有相邻页面的保护属性（PAGE_*）。这些页面与含有用pvAddress参数设定的地址的页面拥有相同的保属性、状态和类型
Type	用于指明支持所有相邻页面的物理存储器的类型（MEM_IMAGE，MEM_MAPPED或MEM_PRIVATE）。这些相邻页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型。如果是Windows 98，那么这个成员将总是MEM_PRIVATE

14.3.1 VMQuery函数

当我刚刚开始学习Windows的内存结构的时候，我将VirtualQuery函数用作我的学习指导。实际上，如果观察本书的第1版(原英文版第1版)，你会发现VMMap示例应用程序比本书下一节中介绍的程序简单得多。在老版本中，我使用了一个简单的循环，它反复调用VirtualQuery函数，而每次调用时，我只是创建一个简单的代码行，它包含了MEMORY_BASE_INFORMATION结构的各个成员。我研究了它的交换方式，并且在参考SDK文档（当时这个文档很不完善）时将内存管理结构组合在一起。从此我学到了许多知识。虽然VirtualQuery和MEMORY_BASIC_ INFORMATION结构使你能够深入了解许多关于内存的情况，但是现在它们无法为你提供使你真正了解全部情况的足够信息。

问题是MEMORY_BASIC_INFORMATION结构无法返回系统已经存放在内部的所有信息。如果你有一个内存地址，想获得关于该地址的某些简单的信息，那么使用VirtualQuery函数的效果是相当不错的。如果只是想知道是否给一个地址提交了物理存储器，或者是否可以向一个内存地址读取或写入信息，那么VirtualQuery函数的作用也很好。但是，如果想知道已保留的地址空间区域的合计大小，或者想要知道一个区域中的地址空间块的数量，或者想知道一个区域是否包含线程堆栈，那么仅仅一次调用VirtualQuery将无法为你提供你想知道的信息。

为了获得完整的内存信息，我创建了一个函数，即VMQuery：

```
BOOL VMQuery(
   HANDLE hProcess,
   PVOID pvAddress,
   PVMQUERY pVMQ);
```

该函数与VirtualQueryEx有些类似，它拥有一个进程句柄（在hProcess中），一个内存地址（在pvAddress中）和一个指向将被填充的结构的指针（由pVMQ设定）。该结构是个VMQUERY结构，下面是结构的定义：

```
typedef struct {
   // Region information
```

China-pub.com 下载

```
   PVOID pvRgnBaseAddress;
   DWORD dwRgnProtection;  // PAGE_*
   SIZE_T RgnSize;
   DWORD dwRgnStorage;     // MEM_*: Free, Image, Mapped, Private
   DWORD dwRgnBlocks;
   DWORD dwRgnGuardBlks;   // If > 0, region contains thread stack
   BOOL fRgnIsAStack;      // TRUE if region contains thread stack

   // Block information
   PVOID pvBlkBaseAddress;
   DWORD dwBlkProtection;  // PAGE_*
   SIZE_T BlkSize;
   DWORD dwBlkStorage;     //
 MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;
```

只要简单地看一眼就会发现，我的 VMQUERY结构比Windows的MEMORY_BASIC_INFORMATION结构包含了多得多的信息。我的结构分成两个不同的部分：一个是区域信息，另一个是块信息。区域信息部分用于描述关于区域的信息，块信息部分用于描述关于包含用pvAddress参数设定的地址块的信息。表14-4对所有的成员进行了说明。

表14-4 VMQUERY 函数的成员变量

成 员 名	描 述
pvRgnBaseAddress	指明虚拟地址空间区域的基地址，该区域包含了用pvAddress参数设定的地址
dwRgnProtection	指明地址空间区域刚刚被保留时赋予该区域的保护属性（PAGE_*）
RgnSize	指明已经保留的地址空间区域的大小（以字节为计量单位）
dwRgnStorage	指明区域中的地址空间块主体使用的物理存储器的类型。该值可以是下列几个值之一：MEM_FREE、MEM_IMAGE、MEM_MAPPED或MEM_PRIVATE。Windows 98不区分不同的内存类型，因此在 Windows 98下本成员将总是MEM_FREE或MEM_PRIVATE
dwRgnBlocks	指明地址空间区域中包含的地址块的数量
dwRgnGuardBlks	指明PAGE_GUARD保护属性标志已经打开的地址块的数量。该值通常既可以是0，也可以是1。如果是1，则表示该区域已经被保留，以包含一个线程堆栈。在Windows 98下，这个成员的值总是0
fRgnIsAStack	指明一个区域是否包含线程堆栈。该值是通过“优化推测”来确定，因为不可能有百分之百的把握知道一个区域是否包含线程堆栈
pvBlkBaseAddress	指明包含用pvAddress参数设定的地址块的基地址
dwBlkProtection	指明包含用pvAddress参数设定的地址块的保护属性
BlkSize	指明包含用pvAddress参数设定的地址块的大小（以字节为计量单位）
dwBlkStorage	指明包含用pvAddress参数设定的地址块的内容。这个值可以是下列值之一：MEM_FREE、MEM_RESERVE、MEM_IMAGE、MEM_MAPPED或MEM_PRIVATE。在Windows 98下，这个值决不能是 MEM_IMAGE或MEM_MAPPED

毫无疑问，VMQuery函数必须执行相当数量的处理操作，包括多次调用 VirtualQueryEx，以便获取所有的信息，这意味着它的运行速度要大大低于 VirtualQueryEx函数。由于这个原因，在决定调用这两个函数中的哪一个时，应该三思而后行。如果不需要通过 VMQuery获得的额外信息，那么应该调用VirtualQuery或VirtualQueryEx函数。

清单14-3列出的VMQuery.cpp文件显示了我如何获得和管理设置 VMQUERY结构的成员时需要的全部信息。 VMQuery.cpp和VMQuery.h两个文件均在本书所附光盘上的 14-VMMap目录

China-pub.com
下载

下。清单14-3中的代码注释说明了处理数据的方法。

清单14-3 VMQuery的程序清单

VMQuery.cpp

```
Module:  VMQuery.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include "VMQuery.h"

//////////////////////////////////////////////////////////////////////////////

// Helper structure
typedef struct {
   SIZE_T RgnSize;
   DWORD  dwRgnStorage;     // MEM_*: Free, Image, Mapped, Private
   DWORD  dwRgnBlocks;
   DWORD  dwRgnGuardBlks;   // If > 0, region contains thread stack
   BOOL   fRgnIsAStack;     // TRUE if region contains thread stack
} VMQUERY_HELP;


// This global, static variable holds the allocation granularity value for
// this CPU platform. Initialized the first time VMQuery is called.
static DWORD gs_dwAllocGran = 0;


//////////////////////////////////////////////////////////////////////////////
// Iterates through a region's blocks and returns findings in VMQUERY_HELP
static BOOL VMQueryHelp(HANDLE hProcess, LPCVOID pvAddress,
   VMQUERY_HELP *pVMQHelp) {

   // Each element contains a page protection
   // (i.e.: 0=reserved, PAGE_NOACCESS, PAGE_READWRITE, etc.)
   DWORD dwProtectBlock[4] = { 0 };

   ZeroMemory(pVMQHelp, sizeof(*pVMQHelp));

   // Get address of region containing passed memory address.
   MEMORY_BASIC_INFORMATION mbi;
   BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
      == sizeof(mbi));

   if (!fOk)
      return(fOk);   // Bad memory address, return failure

   // Walk starting at the region's base address (which never changes)
   PVOID pvRgnBaseAddress = mbi.AllocationBase;

   // Walk starting at the first block in the region (changes in the loop)
   PVOID pvAddressBlk = pvRgnBaseAddress;
```

China-pub.com 下载

```
// Save the memory type of the physical storage block.
pVMQHelp->dwRgnStorage = mbi.Type;

for (;;) {
   // Get info about the current block.
   fOk = (VirtualQueryEx(hProcess, pvAddressBlk, &mbi, sizeof(mbi))
      == sizeof(mbi));
   if (!fOk)
      break;   // Couldn't get the information, end loop.

   // Is this block in the same region?
   if (mbi.AllocationBase != pvRgnBaseAddress)
      break;   // Found a block in the next region; end loop.

   // We have a block contained in the region.

   // The following if statement is for detecting stacks in Windows 98.
   // A Windows 98 stack region's last 4 blocks look like this:
   // reserved block, no access block, read-write block, reserved block
   if (pVMQHelp->dwRgnBlocks < 4) {
      // 0th through 3rd block, remember the block's protection
      dwProtectBlock[pVMQHelp->dwRgnBlocks] =
         (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
   } else {
      // We've seen 4 blocks in this region.
      // Shift the protection values down in the array.
      MoveMemory(&dwProtectBlock[0], &dwProtectBlock[1],
         sizeof(dwProtectBlock) - sizeof(DWORD));

      // Add the new protection value to the end of the array.
      dwProtectBlock[3] = (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
   }

   pVMQHelp->dwRgnBlocks++;              // Add another block to the region
   pVMQHelp->RgnSize += mbi.RegionSize; // Add block's size to region size

   // If block has PAGE_GUARD attribute, add 1 to this counter
   if ((mbi.Protect & PAGE_GUARD) == PAGE_GUARD)
      pVMQHelp->dwRgnGuardBlks++;

   // Take a best guess as to the type of physical storage committed to the
   // block. This is a guess because some blocks can convert from MEM_IMAGE
   // to MEM_PRIVATE or from MEM_MAPPED to MEM_PRIVATE; MEM_PRIVATE can
   // always be overridden by MEM_IMAGE or MEM_MAPPED.
   if (pVMQHelp->dwRgnStorage == MEM_PRIVATE)
      pVMQHelp->dwRgnStorage = mbi.Type;

   // Get the address of the next block.
   pvAddressBlk = (PVOID) ((PBYTE) pvAddressBlk + mbi.RegionSize);
}

// After examining the region, check to see whether it is a thread stack
// Windows 2000: Assume stack if region has at least 1 PAGE_GUARD block
// Windows 9x:   Assume stack if region has at least 4 blocks with
//               3rd block from end: reserved
```

China-pub.com
下载


```
   //                2nd block from end: PAGE_NOACCESS
   //                1st block from end: PAGE_READWRITE
   //                block at end: another reserved block.
   pVMQHelp->fRgnIsAStack =
      (pVMQHelp->dwRgnGuardBlks > 0)          ||
      ((pVMQHelp->dwRgnBlocks >= 4)           &&
      (dwProtectBlock[0] == 0)                &&
      (dwProtectBlock[1] == PAGE_NOACCESS)  &&
      (dwProtectBlock[2] == PAGE_READWRITE) &&
      (dwProtectBlock[3] == 0));

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ) {

   if (gs_dwAllocGran == 0) {
      // Set allocation granularity if this is the first call
      SYSTEM_INFO sinf;
      GetSystemInfo(&sinf);
      gs_dwAllocGran = sinf.dwAllocationGranularity;
   }

   ZeroMemory(pVMQ, sizeof(*pVMQ));

   // Get the MEMORY_BASIC_INFORMATION for the passed address.
   MEMORY_BASIC_INFORMATION mbi;
   BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
      == sizeof(mbi));

   if (!fOk)
      return(fOk);   // Bad memory address, return failure

   // The MEMORY_BASIC_INFORMATION structure contains valid information.
   // Time to start setting the members of our own VMQUERY structure.

   // First, fill in the block members. We'll fill the region members later.
   switch (mbi.State) {
      case MEM_FREE:       // Free block (not reserved)
         pVMQ->pvBlkBaseAddress = NULL;
         pVMQ->BlkSize = 0;
         pVMQ->dwBlkProtection = 0;
         pVMQ->dwBlkStorage = MEM_FREE;
         break;

      case MEM_RESERVE:    // Reserved block without committed storage in it.
         pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
         pVMQ->BlkSize = mbi.RegionSize;
      // For an uncommitted block, mbi.Protect is invalid. So we will
      // show that the reserved block inherits the protection attribute
      // of the region in which it is contained.
      pVMQ->dwBlkProtection = mbi.AllocationProtect;
```

China-pub.com 下载

```
      pVMQ->dwBlkStorage = MEM_RESERVE;
      break;

   case MEM_COMMIT:     // Reserved block with committed storage in it.
      pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
      pVMQ->BlkSize = mbi.RegionSize;
      pVMQ->dwBlkProtection = mbi.Protect;
      pVMQ->dwBlkStorage = mbi.Type;
      break;

   default:
      DebugBreak();
      break;
}

// Now fill in the region data members.
VMQUERY_HELP VMQHelp;
switch (mbi.State) {
   case MEM_FREE:       // Free block (not reserved)
      pVMQ->pvRgnBaseAddress = mbi.BaseAddress;
      pVMQ->dwRgnProtection  = mbi.AllocationProtect;
      pVMQ->RgnSize          = mbi.RegionSize;
      pVMQ->dwRgnStorage     = MEM_FREE;
      pVMQ->dwRgnBlocks      = 0;
      pVMQ->dwRgnGuardBlks   = 0;
      pVMQ->fRgnIsAStack     = FALSE;
      break;

   case MEM_RESERVE:    // Reserved block without committed storage in it.
      pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
      pVMQ->dwRgnProtection  = mbi.AllocationProtect;

      // Iterate through all blocks to get complete region information.
      VMQueryHelp(hProcess, pvAddress, &VMQHelp);

      pVMQ->RgnSize          = VMQHelp.RgnSize;
      pVMQ->dwRgnStorage     = VMQHelp.dwRgnStorage;
      pVMQ->dwRgnBlocks      = VMQHelp.dwRgnBlocks;
      pVMQ->dwRgnGuardBlks   = VMQHelp.dwRgnGuardBlks;
      pVMQ->fRgnIsAStack     = VMQHelp.fRgnIsAStack;
      break;
     case MEM_COMMIT:     // Reserved block with committed storage in it.
        pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
        pVMQ->dwRgnProtection  = mbi.AllocationProtect;

        // Iterate through all blocks to get complete region information.
        VMQueryHelp(hProcess, pvAddress, &VMQHelp);

        pVMQ->RgnSize          = VMQHelp.RgnSize;
        pVMQ->dwRgnStorage     = VMQHelp.dwRgnStorage;
        pVMQ->dwRgnBlocks      = VMQHelp.dwRgnBlocks;
        pVMQ->dwRgnGuardBlks   = VMQHelp.dwRgnGuardBlks;
        pVMQ->fRgnIsAStack     = VMQHelp.fRgnIsAStack;
        break;

     default:
        DebugBreak();
```

China-pub.com
下载


```
            break;
      }

      return(fOk);
}

//////////////////////////////// End of File //////////////////////////////////
```

VMQuery.h

```
/******************************************************************************
Module:  VMQuery.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

typedef struct {
   // Region information
   PVOID  pvRgnBaseAddress;
   DWORD  dwRgnProtection;  // PAGE_*
   SIZE_T RgnSize;
   DWORD  dwRgnStorage;     // MEM_*: Free, Image, Mapped, Private
   DWORD  dwRgnBlocks;
   DWORD  dwRgnGuardBlks;   // If > 0, region contains thread stack
   BOOL   fRgnIsAStack;     // TRUE if region contains thread stack
   // Block information
   PVOID  pvBlkBaseAddress;
   DWORD  dwBlkProtection;  // PAGE_*
   SIZE_T BlkSize;
   DWORD  dwBlkStorage;     // MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;

//////////////////////////////////////////////////////////////////////////////

BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ);

//////////////////////////////// End of File //////////////////////////////////
```

14.3.2 虚拟内存表示例应用程序

清单14-4列出的VMMap应用程序（14 VMMap.exe）列出了一个进程的地址空间，并且显示了各个地址空间区域和区域中的地址块。该应用程序的源代码和资源文件均位于本书所附光盘上的14-VMMap目录下。当启动该应用程序时，就会出现图14-6所示窗口。

我使用该应用程序的列表框的内容来生成第13章中的表13-3、表13-5和表13-6中显示的虚拟内存表交换信息。该列表框中的每一项都显示了调用VMQuery函数所获得的信息的结果。Refresh函数中的主要循环类似下面的样子：

```
BOOL fOk = TRUE;
PVOID pvAddress = NULL;

.
.
.
```

China-pub.com 下载

```
while (fOk) {

   VMQUERY vmq;
   fOk = VMQuery(hProcess, pvAddress, &vmq);

   if (fOk) {
      // Construct the line to be displayed, and add it to the list box.
      TCHAR szLine[1024];
      ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
      ListBox_AddString(hwndLB, szLine);

      if (fExpandRegions) {
         for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
            dwBlock++) {

            ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
            ListBox_AddString(hwndLB, szLine);

            // Get the address of the next region to test.
            pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
            if (dwBlock < vmq.dwRgnBlocks - 1) {
               // Don't query the memory info after the last block.
               fOk = VMQuery(hProcess, pvAddress, &vmq);
            }
         }
      }

      // Get the address of the next region to test.
      pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
   }
}
```

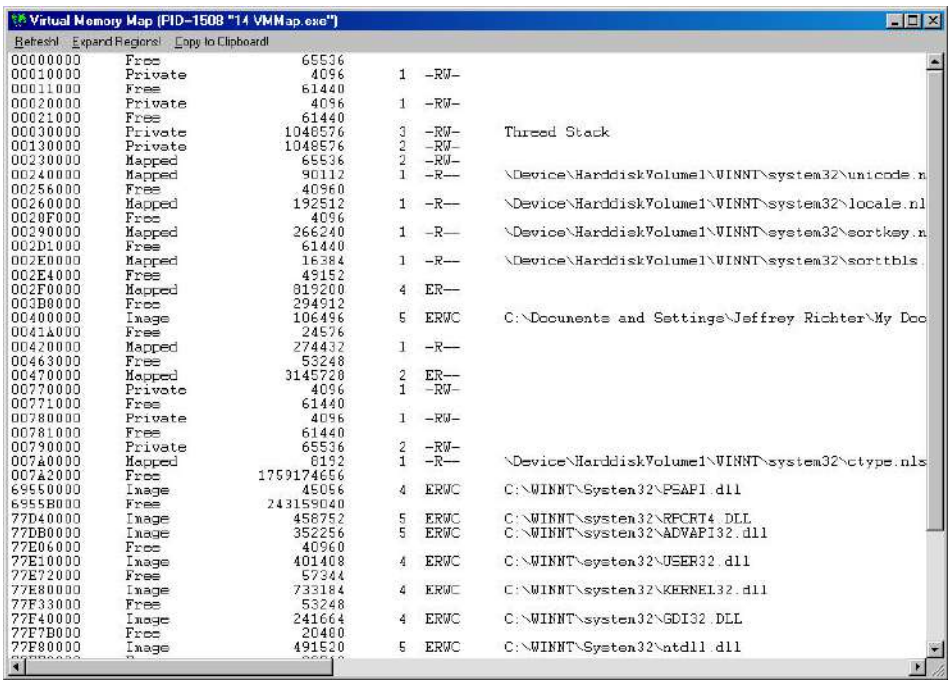


图14-6 启动VMMap时出现的窗口

China-pub.com
下载

该循环从虚拟地址NULL开始运行，当VMQuery函数返回FALSE时结束，表示它不再能够通过该进程的地址空间。该循环的每个重复操作，均有一个对 ConstructRgnInfoLine函数的调用，该函数将关于区域的信息填入一个字符缓存，然后这些信息被附加给列表框。

在这个主循环中，有一个嵌套的循环，它重复通过该地址空间区域中的每个地址块，该循环的每次重复操作均调用ConstructRgnInfoLine，以便用关于该区域的块的信息填入字符缓存，然后这些信息被附加给列表框。使用VMQuery函数，可以比较容易地通过进程的地址空间。

清单14-4 VMMap应用程序的程序清单



VMMap.cpp

```
Module:  VMMap.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <psapi.h>
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>         // For sprintf
#include "..\04-ProcessInfo\Toolhelp.h"
#include "Resource.h"
#include "VMQuery.h"


//////////////////////////////////////////////////////////////////////////////


DWORD g_dwProcessId = 0;  // Which process to walk?
BOOL  g_fExpandRegions = FALSE;
CToolhelp g_toolhelp;


// GetMappedFileName is only on Windows 2000 in PSAPI.DLL
// If this function exists on the host system, we'll use it
typedef DWORD (WINAPI* PFNGETMAPPEDFILENAME)(HANDLE, PVOID, PTSTR, DWORD);
static PFNGETMAPPEDFILENAME g_pfnGetMappedFileName = NULL;


//////////////////////////////////////////////////////////////////////////////
// I use this function to obtain the dump figures in the book.
void CopyControlToClipboard(HWND hwnd) {
   TCHAR szClipData[128 * 1024] = { 0 };

   int nCount = ListBox_GetCount(hwnd);
   for (int nNum = 0; nNum < nCount; nNum++) {
      TCHAR szLine[1000];
      ListBox_GetText(hwnd, nNum, szLine);
      _tcscat(szClipData, szLine);
      _tcscat(szClipData, TEXT("\r\n"));
   }
```

China-pub.com 下载

```
   OpenClipboard(NULL);
   EmptyClipboard();

   // Clipboard accepts only data that is in a block allocated
   // with GlobalAlloc using the GMEM_MOVEABLE and GMEM_DDESHARE flags.
   HGLOBAL hClipData = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
      sizeof(TCHAR) * (_tcslen(szClipData) + 1));
   PTSTR pClipData = (PTSTR) GlobalLock(hClipData);

   _tcscpy(pClipData, szClipData);

#ifdef UNICODE
   BOOL fOk = (SetClipboardData(CF_UNICODETEXT, hClipData) == hClipData);
#else
   BOOL fOk = (SetClipboardData(CF_TEXT, hClipData) == hClipData);
#endif
   CloseClipboard();

   if (!fOk) {
      GlobalFree(hClipData);
      chMB("Error putting text on the clipboard");
   }
}

//////////////////////////////////////////////////////////////////////////////


PCTSTR GetMemStorageText(DWORD dwStorage) {

   PCTSTR p = TEXT("Unknown");
   switch (dwStorage) {
   case MEM_FREE:    p = TEXT("Free   "); break;
   case MEM_RESERVE: p = TEXT("Reserve"); break;
   case MEM_IMAGE:   p = TEXT("Image  "); break;
   case MEM_MAPPED:  p = TEXT("Mapped "); break;
   case MEM_PRIVATE: p = TEXT("Private"); break;
   }
   return(p);
}

//////////////////////////////////////////////////////////////////////////////


PTSTR GetProtectText(DWORD dwProtect, PTSTR szBuf, BOOL fShowFlags) {

   PCTSTR p = TEXT("Unknown");
   switch (dwProtect & ~(PAGE_GUARD | PAGE_NOCACHE | PAGE_WRITECOMBINE)) {
   case PAGE_READONLY:          p = TEXT("-R--"); break;
   case PAGE_READWRITE:         p = TEXT("-RW-"); break;
   case PAGE_WRITECOPY:         p = TEXT("-RWC"); break;
   case PAGE_EXECUTE:           p = TEXT("E---"); break;
   case PAGE_EXECUTE_READ:      p = TEXT("ER--"); break;
   case PAGE_EXECUTE_READWRITE: p = TEXT("ERW-"); break;
```

China-pub.com
下载


```
      case PAGE_EXECUTE_WRITECOPY: p = TEXT("ERWC"); break;
      case PAGE_NOACCESS:          p = TEXT("----"); break;
      }
      _tcscpy(szBuf, p);
      if (fShowFlags) {
         _tcscat(szBuf, TEXT(" "));
         _tcscat(szBuf, (dwProtect & PAGE_GUARD)        ? TEXT("G") : TEXT("-"));
         _tcscat(szBuf, (dwProtect & PAGE_NOCACHE)      ? TEXT("N") : TEXT("-"));
         _tcscat(szBuf, (dwProtect & PAGE_WRITECOMBINE) ? TEXT("W") : TEXT("-"));
      }
      return(szBuf);
   }

   //////////////////////////////////////////////////////////////////////////////


   void ConstructRgnInfoLine(HANDLE hProcess, PVMQUERY pVMQ,
      PTSTR szLine, int nMaxLen) {
      _stprintf(szLine, TEXT("%p     %s  %16u  "),
         pVMQ->pvRgnBaseAddress,
         GetMemStorageText(pVMQ->dwRgnStorage),
         pVMQ->RgnSize);

      if (pVMQ->dwRgnStorage != MEM_FREE) {
         wsprintf(_tcschr(szLine, 0), TEXT("%5u  "), pVMQ->dwRgnBlocks);
         GetProtectText(pVMQ->dwRgnProtection, _tcschr(szLine, 0), FALSE);
      }

      _tcscat(szLine, TEXT("     "));

      // Try to obtain the module pathname for this region.
      int nLen = _tcslen(szLine);
      if (pVMQ->pvRgnBaseAddress != NULL) {
         MODULEENTRY32 me = { sizeof(me) };

         if (g_toolhelp.ModuleFind(pVMQ->pvRgnBaseAddress, &me)) {
            lstrcat(&szLine[nLen], me.szExePath);
         } else {
            // This is not a module; see if it's a memory-mapped file
            if (g_pfnGetMappedFileName != NULL) {
               DWORD d = g_pfnGetMappedFileName(hProcess,
                  pVMQ->pvRgnBaseAddress, szLine + nLen, nMaxLen - nLen);
               if (d == 0) {
                  // NOTE: GetMappedFileName modifies the string when it fails
                  szLine[nLen] = 0;
               }
            }
         }
      }

      if (pVMQ->fRgnIsAStack) {
         _tcscat(szLine, TEXT("Thread Stack"));
      }
   }
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

void ConstructBlkInfoLine(PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {

   _stprintf(szLine, TEXT("   %p  %s  %16u         "),
      pVMQ->pvBlkBaseAddress,
      GetMemStorageText(pVMQ->dwBlkStorage),
      pVMQ->BlkSize);

   if (pVMQ->dwBlkStorage != MEM_FREE) {
      GetProtectText(pVMQ->dwBlkProtection, _tcschr(szLine, 0), TRUE);
   }
}


//////////////////////////////////////////////////////////////////////////////


void Refresh(HWND hwndLB, DWORD dwProcessId, BOOL fExpandRegions) {

   // Delete contents of list box & add a horizontal scroll bar
   ListBox_ResetContent(hwndLB);
   ListBox_SetHorizontalExtent(hwndLB, 300 * LOWORD(GetDialogBaseUnits()));

   // Is the process still running?
   HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION,
      FALSE, dwProcessId);

   if (hProcess == NULL) {
      ListBox_AddString(hwndLB, TEXT(""));   // Blank line, looks better
      ListBox_AddString(hwndLB,
         TEXT("    The process ID identifies a process that is not running"));
      return;
   }

   // Grab a new snapshot of the process
   g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, dwProcessId);

   // Walk the virtual address space, adding entries to the list box.
   BOOL fOk = TRUE;
   PVOID pvAddress = NULL;

   SetWindowRedraw(hwndLB, FALSE);
   while (fOk) {

      VMQUERY vmq;
      fOk = VMQuery(hProcess, pvAddress, &vmq);
      if (fOk) {
         // Construct the line to be displayed, and add it to the list box.
         TCHAR szLine[1024];
         ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
         ListBox_AddString(hwndLB, szLine);

         if (fExpandRegions) {
```

China-pub.com
下载


```
            for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
               dwBlock++) {

               ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
               ListBox_AddString(hwndLB, szLine);

               // Get the address of the next region to test.
               pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
               if (dwBlock < vmq.dwRgnBlocks - 1) {
                  // Don't query the memory info after the last block.
                  fOk = VMQuery(hProcess, pvAddress, &vmq);
               }
            }
         }

         // Get the address of the next region to test.
         pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
      }
   }
   SetWindowRedraw(hwndLB, TRUE);
   CloseHandle(hProcess);
}

//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_VMMAP);

   // Show which process we're walking in the window's caption
   TCHAR szCaption[MAX_PATH * 2];
   GetWindowText(hwnd, szCaption, chDIMOF(szCaption));
   g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, g_dwProcessId);
   PROCESSENTRY32 pe = { sizeof(pe) };
   wsprintf(&szCaption[lstrlen(szCaption)], TEXT(" (PID=%u \"%s\")"),
      g_dwProcessId, g_toolhelp.ProcessFind(g_dwProcessId, &pe)
         ? pe.szExeFile : TEXT("unknown"));
   SetWindowText(hwnd, szCaption);

   // VMMap has so much info to show, let's maximize it by default
   ShowWindow(hwnd, SW_MAXIMIZE);

   // Force the list box to refresh itself
   Refresh(GetDlgItem(hwnd, IDC_LISTBOX), g_dwProcessId, g_fExpandRegions);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

   // The list box always fills the whole client area
```

China-pub.com 下载

```
   SetWindowPos(GetDlgItem(hwnd, IDC_LISTBOX), NULL, 0, 0, cx, cy,
      SWP_NOZORDER);
}

//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case ID_REFRESH:
         Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
            g_dwProcessId, g_fExpandRegions);
         break;

      case ID_EXPANDREGIONS:
         g_fExpandRegions = g_fExpandRegions ? FALSE: TRUE;
         Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
            g_dwProcessId, g_fExpandRegions);
         break;
         case ID_COPYTOCLIPBOARD:
            CopyControlToClipboard(GetDlgItem(hwnd, IDC_LISTBOX));
            break;
      }
   }

   //////////////////////////////////////////////////////////////////////////////


   INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

      switch (uMsg) {
         chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
         chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
         chHANDLE_DLGMSG(hwnd, WM_SIZE,       Dlg_OnSize);
      }
      return(FALSE);
   }


   //////////////////////////////////////////////////////////////////////////////


   int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

      CToolhelp::EnableDebugPrivilege();

      // Try to load PSAPI.DLL and get the address of GetMappedFileName
      HMODULE hmodPSAPI = LoadLibrary(TEXT("PSAPI"));
      if (hmodPSAPI != NULL) {
   #ifdef UNICODE
```

China-pub.com
下载


```
      g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
         GetProcAddress(hmodPSAPI, "GetMappedFileNameW");
#else
      g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
         GetProcAddress(hmodPSAPI, "GetMappedFileNameA");
#endif
   }

   g_dwProcessId = _ttoi(pszCmdLine);
   if (g_dwProcessId == 0) {
      g_dwProcessId = GetCurrentProcessId();
   }

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMMAP), NULL, Dlg_Proc);
   if (hmodPSAPI != NULL)
      FreeLibrary(hmodPSAPI); // Free PSAPI.DLL if we loaded it

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

VMMap.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
```

China-pub.com 下载

```
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VMMAP DIALOG DISCARDABLE  10, 18, 250, 250
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU | WS_THICKFRAME
CAPTION "Virtual Memory Map"
MENU IDR_VMMAP
FONT 8, "Courier"
BEGIN
   LISTBOX         IDC_LISTBOX,0,0,248,248,LBS_NOINTEGRALHEIGHT | NOT
                   WS_BORDER | WS_VSCROLL | WS_HSCROLL | WS_GROUP |
                   WS_TABSTOP
END


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMMAP               ICON    DISCARDABLE     "VMMap.Ico"

/////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDR_VMMAP MENU DISCARDABLE
BEGIN
   MENUITEM "&Refresh!",                   ID_REFRESH
   MENUITEM "&Expand Regions!",            ID_EXPANDREGIONS
```

China-pub.com
下载


```
    MENUITEM "&Copy to Clipboard!",         ID_COPYTOCLIPBOARD
END

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第15章　在应用程序中使用虚拟内存

Windows提供了3种进行内存管理的方法，它们是：

- 虚拟内存，最适合用来管理大型对象或结构数组。
- 内存映射文件，最适合用来管理大型数据流（通常来自文件）以及在单个计算机上运行的多个进程之间共享数据。
- 内存堆栈，最适合用来管理大量的小对象。

本章将要介绍第一种方法，即虚拟内存。内存映射文件和堆栈分别在第 17章和第18章介绍。

用于管理虚拟内存的函数可以用来直接保留一个地址空间区域，将物理存储器（来自页文件）提交给该区域，并且可以设置你自己的保护属性。

15.1　在地址空间中保留一个区域

通过调用VirtualAlloc函数，可以在进程的地址空间中保留一个区域：

```
PVOID VirtualAlloc(
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD fdwAllocationType,
   DWORD fdwProtect);
```

第一个参数pvAddress包含一个内存地址，用于设定想让系统将地址空间保留在什么地方。在大多数情况下，你为该参数传递MULL。它告诉VirtualAlloc，保存着一个空闲地址区域的记录的系统应该将区域保留在它认为合适的任何地方。系统可以从进程的地址空间的任何位置来保留一个区域，因为不能保证系统可以从地址空间的底部向上或者从上面向底部来分配各个区域。可以使用MEM_TOP_DOWN标志来说明该分配方式。这个标志将在本章的后面加以介绍。

对大多数程序员来说，能够选择一个特定的内存地址，并在该地址保留一个区域，这是个非同寻常的想法。当你在过去分配内存时，操作系统只是寻找一个其大小足以满足需要的内存块，并分配该内存块，然后返回它的地址。但是，由于每个进程有它自己的地址空间，因此可以设定一个基本内存地址，在这个地址上让操作系统保留地址空间区域。

例如，你想将一个从50 MB开始的区域保留在进程的地址空间中。这时，可以传递52 428 800（50×1024×1024）作为pvAddress参数。如果该内存地址有一个足够大的空闲区域满足你的要求，那么系统就保留这个区域并返回。如果在特定的地址上不存在空闲区域，或者如果空闲区域不够大，那么系统就不能满足你的要求，VirtualAlloc函数返回NULL。注意，为pvAddress参数传递的任何地址必须始终位于进程的用户方式分区中，否则对 VirtualAlloc函数的调用就会失败，导致它返回NULL。

第13章讲过，地址空间区域总是按照分配粒度的边界来保留的(迄今为止在所有的Windows环境下均是64KB)。因此，如果试图在进程地址空间中保留一个从19 668 992(300×65 536 + 8192)这个地址开始的区域，系统就会将这个地址圆整为 64KB的倍数，然后保留从19 660 800（300×65 536）这个地址开始的区域。

如果VirtualAlloc函数能够满足你的要求，那么它就返回一个值，指明保留区域的基地址。

China-pub.com
下载

如果传递一个特定的地址作为 VirtualAlloc的pvAddress参数，那么该返回值与传递给VirtualAlloc的值相同，并被圆整为（如果需要的话）64KB边界值。

VirtualAlloc函数的第二个参数是dwSize，用于设定想保留的区域的大小（以字节为计量单位）。由于系统保留的区域始终必须是 CPU页面大小的倍数，因此，如果试图保留一个跨越62KB的区域，结果就会在使用 4 KB、8 KB或16 KB页面的计算机上产生一个跨越 64KB的区域。

VirtualAlloc函数的第三个参数是fdwAllocationType，它能够告诉系统你想保留一个区域还是提交物理存储器（这样的区分是必要的，因为 VirtualAlloc函数也可以用来提交物理存储器）。若要保留一个地址空间区域，必须传递 MEM_RESERVE标识符作为FdwAllocationType参数的值。

如果保留的区域预计在很长时间内不会被释放，那么可以在尽可能高的内存地址上保留该区域。这样，该区域就不会从进程地址空间的中间位置上进行保留。因为在这个位置上它可能导致区域分成碎片。如果想让系统在最高内存地址上保留一个区域，必须为 pvAddress参数和fdwAllocationType参数传递 NULL，还必须逐位使用 OR将MEM_TOP_DOWN标志和MEM_RESERVE标志连接起来。

注意　在Windows 98下，MEM_TOP_DOWN标志将被忽略。

最后一个参数是fdwProtect，用于指明应该赋予该地址空间区域的保护属性。与该区域相关联的保护属性对映射到该区域的已提交内存没有影响。无论赋予区域的保护属性是什么，如果没有提交任何物理存储器，那么访问该范围中的内存地址的任何企图都将导致该线程引发一个访问违规。

当保留一个区域时，应该为该区域赋予一个已提交内存最常用的保护属性。例如，如果打算提交的物理存储器的保护属性是 PAGE_READWRITE（这是最常用的保护属性），那么应该用PAGE_READWRITE保护属性来保留该区域。当区域的保护属性与已提交内存的保护属性相匹配时，系统保存的内部记录的运行效率最高。

可以使用下列保护属性中的任何一个： PAGE_NOACCESS、PAGE_READWRITE、PAGE_READONLY、PAGE_EXECUTE、PAGE_EXECUTE_READ或PAGE_EXECUTE_READWRITE。但是，既不能设定 PAGE_WRITECOPY属性，也不能设定 PAGE_EXECUTE_WRITECOPY属性。如果设定了这些属性，VirtualAlloc函数将不保留该区域，并且返回NULL。另外，当保留地址空间区域时，不能使用保护属性标志 PAGE_GUARD，PAGE_NOCACHE或PAGE_WRITECOMBINE，这些标志只能用于已提交的内存。

注意　Windows 98只支持PAGE_NOACCESS、PAGE_READONLY和PAGE_READWRITE保护属性。如果试图保留使用 PAGE_EXECUTE或PAGE_EXECUTE_READ两个保护属性的区域，将会产生一个带有 PAGE_READONLY保护属性的区域。同样，如果保留一个使用 PAGE_EXECUTE_READWRITE保护属性的区域，就会产生一个带有PAGE_READWRITE保护属性的区域。

15.2　在保留区域中的提交存储器

当保留一个区域后，必须将物理存储器提交给该区域，然后才能访问该区域中包含的内存地址。系统从它的页文件中将已提交的物理存储器分配给一个区域。物理存储器总是按页面边界和页面大小的块来提交的。

China-pub.com 下载

若要提交物理存储器，必须再次调用VirtualAlloc函数。不过这次为fdwAllocationType参数传递的是MEM_COMMIT标志，而不是MEM_RESERVE标志。传递的页面保护属性通常与调用VirtualAlloc来保留区域时使用的保护属性相同（大多数情况下是PAGE_READWRITE），不过也可以设定一个不同的保护属性。

在已保留的区域中，你必须告诉VirtualAlloc函数，你想将物理存储器提交到何处，以及要提交多少物理存储器。为了做到这一点，可以在pvAddress参数中设定你需要的内存地址，并在dwSize参数中设定物理存储器的数量（以字节为计量单位）。注意，不必立即将物理存储器提交给整个区域。

下面让我们来看一个如何提交物理存储器。比如说，你的应用程序是在x86 CPU上运行的，该应用程序保留了一个从地址5 242 880开始的512 KB的区域。你想让应用程序将物理存储器提交给已保留区域的6 KB部分，从2 KB的地方开始，直到已保留区域的地址空间。为此，可以调用带有MEM_COMMIT标志的VirtualAlloc函数，如下所示：

```
VirtualAlloc((PVOID) (5242880 + (2 * 1024)), 6 * 1024,
   MEM_COMMIT, PAGE_READWRITE);
```

在这个例子中，系统必须提交8 KB的物理存储器，地址范围从5 242 880到5 251 071 (5 242 880 + 8 KB -1字节)。这两个提交的页面都拥有PAGE_READWRITE保护属性。保护属性只以整个页面为单位来赋予。同一个内存页面的不同部分不能使用不同的保护属性。然而，区域中的一个页面可以使用一种保护属性（比如PAGE_READWRITE），而同一个区域中的另一个页面可以使用不同的保护属性（比如PAGE_READONLY）。

15.3 同时进行区域的保留和内存的提交

有时你可能想要在保留区域的同时，将物理存储器提交给它。只需要一次调用VirtualAlloc函数就能进行这样的操作，如下所示：

```
PVOID pvMem = VirtualAlloc(NULL, 99 * 1024,
   MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

这个函数调用请求保留一个99 KB的区域，并且将99 KB的物理存储器提交给它。当系统处理这个函数调用时，它首先要搜索你的进程的地址空间，找出未保留的地址空间中一个地址连续的区域，它必须足够大，能够存放100 KB（在4 KB页面的计算机上）或104 KB（在8 KB页面的计算机上）。

系统之所以要搜索地址空间，原因是已将pvAddress参数设定为NULL。如果为pvAddress设定了内存地址，系统就要查看在该内存地址上是否存在足够大的未保留地址空间。如果系统找不到足够大的未保留地址空间，VirtualAlloc将返回NULL，

如果能够保留一个合适的区域，系统就将物理存储器提交给整个区域。无论是该区域还是提交的内存，都将被赋予PAGE_READWRITE保护属性。

最后需要说明的是，VirtualAlloc将返回保留区域和提交区域的虚拟地址，然后该虚拟地址被保存在pvMem变量中。如果系统无法找到足够大的地址空间，或者不能提交该物理存储器，VirtualAlloc将返回NULL。

当用这种方式来保留一个区域和提交物理存储器时，将特定的地址作为pvAddress参数传递给VirtualAlloc当然是可能的。否则就必须用OR将MEM_TOP_DOWN标志与fdwAllocationType参数连接起来，并为pvAddress参数传递NULL，让系统在进程的地址空间的顶部选定一个适当的区域。

China-pub.com
下载

15.4　何时提交物理存储器

假设想实现一个电子表格应用程序，这个电子表格为 200行 x 256列。对于每一个单元格，都需要一个CELLDATA结构来描述单元格的内容。若要处理这种二维单元格矩阵，最容易的方法是在应用程序中声明下面的变量：

```
CELLDATA CellData[200][256];
```

如果CELLDATA结构的大小是128字节，那么这个二维矩阵将需要6 553 600（200 x 256 x 128）个字节的物理存储器。对于电子表格来说，如果直接用页文件来分配物理存储器，那么这是个不小的数目了，尤其是考虑到大多数用户只是将信息放入少数的单元格中，而大部分单元格却空闲不用，因此显得有些浪费。内存的利用率非常低。

传统上，电子表格一直是用其他数据结构技术来实现的，比如链接表等。使用链接表，只需要为电子表格中实际包含数据的单元格创建 CELLDATA结构。由于电子表格中的大多数单元格都是不用的，因此这种方法可以节省大量的内存。但是这种方法使得你很难获得单元格的内容。如果想知道第 5行第 10列的单元格的内容，必须遍历链接表，才能找到需要的单元格，因此使用链接表方法比明确声明的矩阵方法速度要慢。

虚拟内存为我们提供了一种兼顾预先声明二维矩阵和实现链接表的两全其美的方法。运用虚拟内存，既可以使用已声明的矩阵技术进行快速而方便的访问，又可以利用链接表技术大大节省内存的使用量。

如果想利用虚拟内存技术的优点，你的程序必须按照下列步骤来编写：

1) 保留一个足够大的地址空间区域，用来存放 CELLDATA结构的整个数组。保留一个根本不使用任何物理存储器的区域。

2) 当用户将数据输入一个单元格时，找出 CELLDATA结构应该进入的保留区域中的内存地址。当然，这时尚未有任何物理存储器被映射到该地址，因此，访问该地址的内存的任何企图都会引发访问违规。

3) 就CELLDATA结构来说，只将足够的物理存储器提交给第二步中找到的内存地址（你可以告诉系统将物理存储器提交给保留区域的特定部分，这个区域既可以包含映射到物理存储器的各个部分，也可以包含没有映射到物理存储器的各个部分）。

4) 设置新的CELLDATA结构的成员。

现在物理存储器已经映射到相应的位置，你的程序能够访问内存，而不会引发访问违规。这个虚拟内存技术非常出色，因为只有在用户将数据输入电子表格的单元格时，才会提交物理存储器。由于电子表格中的大多数单元格是空的，因此大部分保留区域没有提交给它的物理存储器。

虚拟内存技术存在的一个问题是，必须确定物理存储器在何时提交。如果用户将数据输入一个单元格，然后只是编辑或修改该数据，那么就没有必要提交物理存储器，因为该单元格的CELLDATA结构的内存在数据初次输入时就已经提交了。

另外，系统总是按页面的分配粒度来提交物理存储器的。因此，当试图为单个 CELLDATA结构提交物理存储器时（像上面的第二步那样），系统实际上提交的是内存的一个完整的页面。这并不像它听起来那样十分浪费：为单个CELLDATA结构提交物理存储器的结果是，也要为附近的其他CELLDATA结构提交内存。如果这时用户将数据输入邻近的单元格（这是经常出现的情况），就不需要提交更多的物理存储器。

有4种方法可以用来确定是否要将物理存储器提交给区域的一个部分：

China-pub.com 下载

- 始终设法进行物理存储器的提交。每次调用VirtualAlloc函数的时候，不要查看物理存储器是否已经映射到地址空间区域的一个部分，而是让你的程序设法进行内存的提交。系统首先查看内存是否已经被提交，如果已经提交，那么就不要提交更多的物理存储器。这种方法最容易操作，但是它的缺点是每次改变CELLDATA结构时要多进行一次函数的调用，这会使程序运行得比较慢。
- （使用VirtualQuery函数）确定物理存储器是否已经提交给包含CELLDATA结构的地址空间。如果已经提交了，那么就不要进行任何别的操作。如果尚未提交，则可以调用VirtualAlloc函数以便提交内存。这种方法实际上比第一种方法差，它既会增加代码的长度，又会降低程序运行的速度（因为增加了对VirtualAlloc函数的调用）。
- 保留一个关于哪些页面已经提交和哪些页面尚未提交的记录。这样做可以使你的应用程序运行得更快，因为不必调用VirtualAlloc函数，你的代码能够比系统更快地确定内存是否已经被提交。它的缺点是，必须不断跟踪页面提交的信息，这可能非常简单，也可能非常困难，要根据你的情况而定。
- 使用结构化异常处理（SEH）方法，这是最好的方法。SEH是一个操作系统特性，它使系统能够在发生某种情况时将此情况通知你的应用程序。实际上可以创建一个带有异常处理程序的应用程序，然后，每当试图访问未提交的内存时，系统就将这个问题通知应用程序。然后你的应用程序便进行内存的提交，并告诉系统重新运行导致异常条件的指令。这时对内存的访问就能成功地进行了，程序将继续运行，仿佛从未发生过问题一样。这种方法是优点最多的方法，因为需要做的工作最少（也就是说要你编写的代码比较少），同时，你的程序可以全速运行。关于SEH的全面介绍，请参见第23、24和25章。第25章中的电子表格示例应用程序说明了如何按照上面介绍的方法来使用虚拟内存。

15.5 回收虚拟内存和释放地址空间区域

若要回收映射到一个区域的物理存储器，或者释放这个地址空间区域，可调用VirtualFree函数：

```
BOOL VirtualFree(
   LPVOID pvAddress,
   SIZE_T dwSize,
   DWORD fdwFreeType);
```

首先让我们观察一下调用VirtualFree函数来释放一个已保留区域的简单例子。当你的进程不再访问区域中的物理存储器时，就可以释放整个保留的区域和所有提交给该区域的物理存储器，方法是一次调用VirtualFree函数。

就这个函数的调用来说，pvAddress参数必须是该区域的基地址。此地址与该区域被保留时VirtualAlloc函数返回的地址相同。系统知道在特定内存地址上的该区域的大小，因此可以为dwSize参数传递0。实际上，必须为dwSize参数传递0，否则对VirtualFree的调用就会失败。对于第三个参数fdwFreeType，必须传递MEM_RELEASE，以告诉系统将所有映射的物理存储器提交给该区域并释放该区域。当释放一个区域时，必须释放该区域保留的所有地址空间。例如不能保留一个128 KB的区域，然后决定只释放它的64 KB。必须释放所有的128 KB。

当想要从一个区域回收某些物理存储器，但是却不释放该区域时，也可以调用VirtualFree函数，若要回收某些物理存储器，必须在VirtualFree函数的pvAddress参数中传递用于标识要回收的第一个页面的内存地址，还必须在dwSize参数中设定要释放的字节数，并在fdwFreeType

China-pub.com
下载

参数中传递MEM_DECOMMIT标志。

与提交物理存储器的情况一样，回收时也必须按照页面的分配粒度来进行。这就是说，设定页面中间的一个内存地址就可以回收整个页面。当然，如果 pvAddress + dwSize的值位于一个页面的中间，那么包含该地址的整个页面将被回收。因此位于 pvAddress 至pvAddress + dwSize范围内的所有页面均被回收。

如果dwSize是0，pvSddress是已分配区域的基地址，那么 VirtualFree将回收全部范围内的已分配页面。当物理存储器的页面已经回收之后，已释放的物理存储器就可以供系统中的所有其他进程使用，如果试图访问未回收的内存，将会造成访问违规。

15.5.1　何时回收物理存储器

在实践中，知道何时回收内存是非常困难的。让我们再以电子表格为例。如果你的应用程序是在x86计算机上运行，每个内存页面是4 KB ，它可以存放32个（4096/128）CELLDATA结构。如果用户删除了单元格 CellData[0][1]的内容，那么只要单元格 CellData[0][0]至CellData[0][31]也不被使用，就可以回收它的内存页面。那么怎么能够知道这个情况呢？可以用下面3种方法来解决这个问题。

- 毫无疑问，最容易的方法是设计一个 CELLDATA结构，它的大小只有一个页面。这时，由于始终都是每个页面使用一个结构，因此当不再需要该结构中的数据时，就可以回收该页面的物理存储器。即使你的数据结构是 x86 CPU上的8 KB或12 KB页面的倍数（通常这是非常大的数据结构），回收内存仍然是非常容易的。当然，如果要使用这种方法，必须定义你的数据结构，使之符合你针对的 CPU的页面大小而不是我们通常编写程序所用的结构。
- 更为实用的方法是保留一个正在使用的结构的记录。为了节省内存，可以使用一个位图。这样，如果有一个100个结构的数组，你也可以维护一个 100位的数组。开始时，所有的位均设置为0，表示这些结构都没有使用。当使用这些结构时，可以将对应的位设置为 1。然后，每当不需要某个结构，并将它的位重新改为 0时，你可以检查属于同一个内存页面的相邻结构的位。如果没有相邻的结构正在使用，就可以回收该页面。
- 最后一个方法是实现一个无用单元收集函数。这个方案依赖于这样一种情况，即当物理存储器初次提交时，系统将一个页面中的所有字节设置为 0。若要使用该方案，首先必须在你的结构中设置一个BOOL（也许称为fInUse）。然后，每次你将一个结构放入已提交的内存中，必须确保该fInUse被置于TRUE。

当你的应用程序运行时，必须定期调用无用单元收集函数。该函数应该遍历所有潜在的数据结构。对于每个数据结构，该函数首先要确定是否已经为该结构提交内存。如果已经提交，该函数将检查fInUse成员，以确定它是否是0。如果该值是0，则表示该结构没有被使用。如果该值是TRUE，则表示该结构正在使用。当无用单元函数检查了属于既定页面的所有结构后，如果所有结构都没有被使用，它将调用VirtualFree函数，回收该内存。

当一个结构不再被视为“在用”（In Use）后，就可以立即调用无用单元收集函数，不过这项操作需要的时间比你想像的要长，因为该函数要循环通过所有可能的结构。实现该函数的一个出色方法是让它作为低优先级线程的一部分来运行。这样，就不必占用执行主应用程序的线程的时间。每当主应用程序运行空闲时，或者主应用程序的线程执行文件的 I/O操作时，系统就可以给无用单元收集函数安排运行时间。

在上面列出的所有方法中，前面的两种方法是我个人喜欢使用的方法。不过，如果你的结

China-pub.com 下载

构比较小（小于一个页面），那么建议你使用最后一种方法。

15.5.2 虚拟内存分配的示例应用程序

清单15-1中列出的VMAlloc应用程序（“15 VMAlloc.exe”）显示了如何使用虚拟内存技术来处理一个结构数组。该应用程序的源代码和资源文件均位于本书所附光盘上的15-VMAlloc目录下。当启动该应用程序时，将出现图15-1所示的窗口。

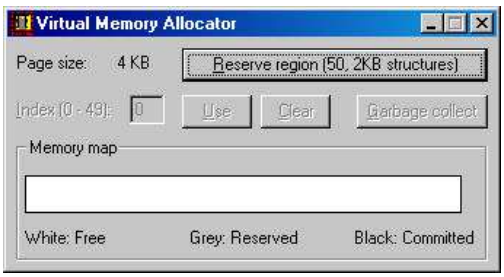


图15-1 运时VMAlloc 程序时出现的窗口

开始时，没有为该结构数组保留任何地址空间的区域，准备为它保留的所有地址空间都是空闲的，如内存表所示。当你点击Reserve Region（50,2 KB结构）按钮时，VMAlloc便调用VirtualAlloc函数，保留该区域，同时内存表被更新，以反映该区域已经被保留。当VirtualAlloc函数保留该区域后，其余按钮变均为活动按钮。

现在可以将一个索引键入编辑控件，以便选定一个索引，然后单击Use按钮。它的作用是将物理存储器提交给用于放置数组元素的内存地址。当一个内存页面被提交时，内存表被刷新，以反映整个数组的保留区域的状态。因此，如果该区域被保留后，你用Use按钮将数组元素7和46标记为“在用”，那么该窗口就显示出图15-2所示的样子（当在4 KB计算机上运行该程序时）。

单击Clear按钮，清除带有“在用”标记的任何元素。但是这样做并不回收映射到数组元素的物理存储器，因为每个页面都包含多个结构的空间，清除一个元素并不意味着其他元素也被清除。如果内存被回收，那么其他结构中的数据就会丢失。由于单击Clear并不影响区域的物理存储器，因此当数组元素被清除时，内存表不会被更新。

但是，当一个结构被清除时，它的fInUse成员将被设置为FALSE。这样的设置是必要的，因为这使得无用单元收集函数能够运行通过所有的结构，并回收不再使用的内存。如果你现在尚未想到这一点，那么Garbage Collect按钮会告诉VMAlloc执行它的无用单元收集例程。为了简化操作，我没有在各个线程上实现无用单元收集例程。

若要展示无用单元收集函数，清除索引46上的数组元素。注意，内存表并没有改变。现在单击Garbage Collect按钮。该程序回收包含元素46的内存页面，同时内存表被更新，以反映这个变化，如图15-3所示。注意，GarbageCollect函数可以很容易地用于你自己的应用程序。我将它用于对任意大小的数据结构数组的操作，这些结构不必完全等于一个页面的大小。唯一的要求是，结构的第一个成员的值必须是BOOL，这表示该结构是否处于在用状态。

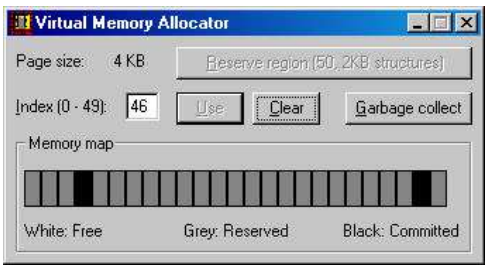


图15-2 标记数组元素后显示的窗口

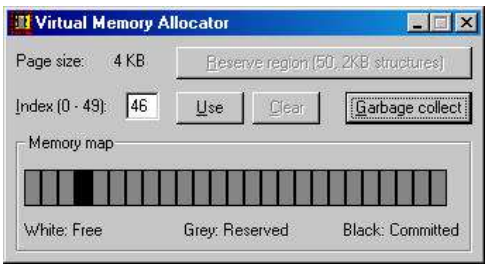


图15-3 内存表更新后显示的窗口

China-pub.com
下载

最后要说明的是，尽管没有直观的图形为你提供必要的信息，但是，当窗口关闭时，所有已提交的内存均被回收，保留的区域被释放。

该程序还包含另一个元素尚未加以说明。该程序必须在 3个位置上确定区域的地址空间中的内存状态：

- 当改变索引后，该程序必须激活 Use按钮，并停用 Clear按钮，或者激活 Use按钮，停用 Clear按钮。
- 在无用单元收集函数中，在实际查看是否已经设置 fInUse标志之前，该程序必须查看内存是否已经提交。
- 当更新内存表时，该程序必须知道哪个页面是空闲的，哪个页面已经被保留，哪个页面已经提交。

VMAlloc通过调用 VirtualQuery函数来执行所有这些测试。

清单15-1　VMAlloc示例应用程序



VMAlloc.cpp

```
/******************************************************************************
Module:  VMAlloc.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include " ..\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////


// The number of bytes in a page on this host machine.
UINT g_uPageSize = 0;

// A dummy data structure used for the array.
typedef struct {
   BOOL fInUse;
   BYTE bOtherData[2048 - sizeof(BOOL)];
} SOMEDATA, *PSOMEDATA;

// The number of structures in the array
#define MAX_SOMEDATA    (50)

// Pointer to an array of data structures
PSOMEDATA g_pSomeData = NULL;

// The rectangular area in the window occupied by the memory map
RECT g_rcMemMap;

//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
```

China-pub.com 下载

```
   chSETDLGICONS(hwnd, IDI_VMALLOC);

   // Initialize the dialog box by disabling all the nonsetup controls.
   EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT),      FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_INDEX),          FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_USE),            FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_CLEAR),          FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), FALSE);

   // Get the coordinates of the memory map display.
   GetWindowRect(GetDlgItem(hwnd, IDC_MEMMAP), &g_rcMemMap);
   MapWindowPoints(NULL, hwnd, (LPPOINT) &g_rcMemMap, 2);

   // Destroy the window that identifies the location of the memory map
   DestroyWindow(GetDlgItem(hwnd, IDC_MEMMAP));

   // Put the page size in the dialog box just for the user's information.
   TCHAR szBuf[10];
   wsprintf(szBuf, TEXT("(%d KB)"), g_uPageSize / 1024);
   SetDlgItemText(hwnd, IDC_PAGESIZE, szBuf);

   // Initialize the edit control.
   SetDlgItemInt(hwnd, IDC_INDEX, 0, FALSE);


   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////


void Dlg_OnDestroy(HWND hwnd) {

   if (g_pSomeData != NULL)
      VirtualFree(g_pSomeData, 0, MEM_RELEASE);
}


//////////////////////////////////////////////////////////////////////////////


VOID GarbageCollect(PVOID pvBase, DWORD dwNum, DWORD dwStructSize) {

   static DWORD s_uPageSize = 0;

   if (s_uPageSize == 0) {
      // Get the page size used on this CPU.
      SYSTEM_INFO si;
      GetSystemInfo(&si);
      s_uPageSize = si.dwPageSize;

   }


   UINT uMaxPages = dwNum * dwStructSize / g_uPageSize;
```

China-pub.com
下载


```
   for (UINT uPage = 0; uPage < uMaxPages; uPage++) {
      BOOL fAnyAllocsInThisPage = FALSE;
      UINT uIndex     = uPage  * g_uPageSize / dwStructSize;
      UINT uIndexLast = uIndex + g_uPageSize / dwStructSize;

      for (; uIndex < uIndexLast; uIndex++) {
         MEMORY_BASIC_INFORMATION mbi;
         VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
         fAnyAllocsInThisPage = ((mbi.State == MEM_COMMIT) &&
            * (PBOOL) ((PBYTE) pvBase + dwStructSize * uIndex));

         // Stop checking this page, we know we can't decommit it.
         if (fAnyAllocsInThisPage) break;
      }
      if (!fAnyAllocsInThisPage) {
         // No allocated structures in this page; decommit it.
         VirtualFree(&g_pSomeData[uIndexLast - 1], dwStructSize, MEM_DECOMMIT);
      }
   }
}

//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   UINT uIndex = 0;

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_RESERVE:
         // Reserve enough address space to hold the array of structures.
         g_pSomeData = (PSOMEDATA) VirtualAlloc(NULL,
            MAX_SOMEDATA * sizeof(SOMEDATA), MEM_RESERVE, PAGE_READWRITE);

         // Disable the Reserve button and enable all the other controls.
         EnableWindow(GetDlgItem(hwnd, IDC_RESERVE),        FALSE);
         EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT),      TRUE);
         EnableWindow(GetDlgItem(hwnd, IDC_INDEX),          TRUE);
         EnableWindow(GetDlgItem(hwnd, IDC_USE),            TRUE);
         EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), TRUE);

         // Force the index edit control to have the focus.
         SetFocus(GetDlgItem(hwnd, IDC_INDEX));

         // Force the memory map to update
         InvalidateRect(hwnd, &g_rcMemMap, FALSE);
         break;

      case IDC_INDEX:
         if (codeNotify != EN_CHANGE)
            break;
```

China-pub.com 下载

```
      uIndex = GetDlgItemInt(hwnd, id, NULL, FALSE);
      if ((g_pSomeData != NULL) && chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
         MEMORY_BASIC_INFORMATION mbi;
         VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
         BOOL fOk = (mbi.State == MEM_COMMIT);
         if (fOk)
            fOk = g_pSomeData[uIndex].fInUse;

         EnableWindow(GetDlgItem(hwnd, IDC_USE),   !fOk);
         EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), fOk);

      } else {
         EnableWindow(GetDlgItem(hwnd, IDC_USE),   FALSE);
         EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
      }
      break;

   case IDC_USE:
      uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
      if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {

         // NOTE: New pages are always zeroed by the system
         VirtualAlloc(&g_pSomeData[uIndex], sizeof(SOMEDATA),
            MEM_COMMIT, PAGE_READWRITE);

         g_pSomeData[uIndex].fInUse = TRUE;

         EnableWindow(GetDlgItem(hwnd, IDC_USE),   FALSE);
         EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), TRUE);

         // Force the Clear button control to have the focus.
         SetFocus(GetDlgItem(hwnd, IDC_CLEAR));

         // Force the memory map to update
         InvalidateRect(hwnd, &g_rcMemMap, FALSE);
      }
      break;

   case IDC_CLEAR:
      uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
      if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
         g_pSomeData[uIndex].fInUse = FALSE;
         EnableWindow(GetDlgItem(hwnd, IDC_USE),   TRUE);
         EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);

         // Force the Use button control to have the focus.
         SetFocus(GetDlgItem(hwnd, IDC_USE));
      }
      break;

   case IDC_GARBAGECOLLECT:
      GarbageCollect(g_pSomeData, MAX_SOMEDATA, sizeof(SOMEDATA));

      // Force the memory map to update
      InvalidateRect(hwnd, &g_rcMemMap, FALSE);
```

China-pub.com
下载


```
            break;
      }
   }

//////////////////////////////////////////////////////////////////////////////


void Dlg_OnPaint(HWND hwnd) {     // Update the memory map

   PAINTSTRUCT ps;
   BeginPaint(hwnd, &ps);

   UINT uMaxPages = MAX_SOMEDATA * sizeof(SOMEDATA) / g_uPageSize;
   UINT uMemMapWidth = g_rcMemMap.right - g_rcMemMap.left;

   if (g_pSomeData == NULL) {

      // The memory has yet to be reserved.
      Rectangle(ps.hdc, g_rcMemMap.left, g_rcMemMap.top,
         g_rcMemMap.right - uMemMapWidth % uMaxPages, g_rcMemMap.bottom);

   } else {

      // Walk the virtual address space, painting the memory map
      for (UINT uPage = 0; uPage < uMaxPages; uPage++) {

         UINT uIndex = uPage * g_uPageSize / sizeof(SOMEDATA);
         UINT uIndexLast = uIndex + g_uPageSize / sizeof(SOMEDATA);
         for (; uIndex < uIndexLast; uIndex++) {

            MEMORY_BASIC_INFORMATION mbi;
            VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
            int nBrush = 0;
            switch (mbi.State) {
               case MEM_FREE:    nBrush = WHITE_BRUSH; break;
               case MEM_RESERVE: nBrush = GRAY_BRUSH;  break;
               case MEM_COMMIT:  nBrush = BLACK_BRUSH; break;
            }

            SelectObject(ps.hdc, GetStockObject(nBrush));
            Rectangle(ps.hdc,
               g_rcMemMap.left + uMemMapWidth / uMaxPages * uPage,
               g_rcMemMap.top,
               g_rcMemMap.left + uMemMapWidth / uMaxPages * (uPage + 1),
               g_rcMemMap.bottom);

         }
      }
   }

   EndPaint(hwnd, &ps);
}

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_PAINT,      Dlg_OnPaint);
      chHANDLE_DLGMSG(hwnd, WM_DESTROY,    Dlg_OnDestroy);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

   // Get the page size used on this CPU.
   SYSTEM_INFO si;
   GetSystemInfo(&si);
   g_uPageSize = si.dwPageSize;

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMALLOC), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

VMAlloc.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS


//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources


#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
```

China-pub.com
下载


```
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END


2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END


3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VMALLOC DIALOG DISCARDABLE  15, 24, 224, 97
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Virtual Memory Allocator"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Page size:",IDC_STATIC,4,6,34,8
    CONTROL         "16 KB",IDC_PAGESIZE,"Static",SS_LEFTNOWORDWRAP |
                    SS_NOPREFIX | WS_GROUP,50,6,20,8
    DEFPUSHBUTTON   "&Reserve region (50, 2KB structures)",IDC_RESERVE,80,4,
                    140,14,WS_GROUP
    LTEXT           "&Index (0 - 49):",IDC_INDEXTEXT,4,26,45,8
    EDITTEXT        IDC_INDEX,56,24,16,12
    PUSHBUTTON      "&Use",IDC_USE,80,24,32,14
    PUSHBUTTON      "&Clear",IDC_CLEAR,116,24,32,14
    PUSHBUTTON      "&Garbage collect",IDC_GARBAGECOLLECT,160,24,60,14
    GROUPBOX        "Memory map",IDC_STATIC,4,42,216,52
    CONTROL         "",IDC_MEMMAP,"Static",SS_BLACKRECT,8,58,208,16
    LTEXT           "White: Free",IDC_STATIC,8,80,39,8
    CTEXT           "Grey: Reserved",IDC_STATIC,82,80,52,8
    RTEXT           "Black: Committed",IDC_STATIC,155,80,58,8
END
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMALLOC             ICON    DISCARDABLE     "VMAlloc.Ico"
#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

15.6 改变保护属性

虽然实践中很少这样做，但是可以改变已经提交的物理存储器的一个或多个页面的保护属性。例如，你编写了一个用于管理链接表的代码，将它的节点存放在一个保留区域中。可以设计一些函数，以便处理该链接表，这样，它们就可以在每个函数开始运行时将已提交内存的保护属性改为 PAGE_READWRITE，然后在每个函数终止运行时将保护属性重新改为 PAGE_NOACCESS。

通过这样的设置，就能够使链接表数据不受隐藏在程序中的其他错误的影响。如果进程中的任何其他代码存在一个迷失指针，试图访问你的链接表数据，那么就会引发访问违规。当试图寻找应用程序中难以发现的错误时，利用保护属性是极其有用的。

若要改变内存页面的保护属性，可以调用 VirtualProtect函数：

```
BOOL VirtualProtect(
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD flNewProtect,
   PDWORD pflOldProtect);
```

这里的pvAddress参数指向内存的基地址（它必须位于进程的用户方式分区中），dwSize参数用于指明你想要改变保护属性的字节数，而 flNewProtect参数则代表PAGE_*保护属性标志中的任何一个标志，但PAGE_WRITECOPY和PAGE_EXECUTE_WRITECOPY这两个标志除外。

最后一个参数pflOldProtect是DWORD的地址，VirtualProtect将用原先与pvAddress位置上的字节相关的保护属性填入该 DWORD。尽管许多应用程序并不需要该信息，但是必须为该参数传递一个有效地址，否则该函数的运行将会失败。

当然，保护属性是与内存的整个页面相关联的，而不是赋予内存的各个字节的。因此，如果要使用下面的代码来调用 4 KB 页面的计算机上的 VirtualProtect函数，其结果是把PAGE_NOACCESS保护属性赋予内存的两个页面：

China-pub.com
下载


```
VirtualProtect(pvRgnBase + (3 * 1024), 2 * 1024,
   PAGE_NOACCESS, &flOldProtect);
```

Windows 98　Windows 98只支持PAGE_READONLY和PAGE_READWRITE两个保护属性。如果试图将页面的保护属性改为PAGE_EXECUTE或PAGE_EXECUTE_READ，该页面可得到PAGE_READONLY保护属性。同样，如果试图将页面的保护属性改为PAGE_EXECUTE_READWRITE，那么该页面将得到PAGE_READWRITE保护属性。

VirtualProtect函数不能用于改变跨越不同保留区域的页面的保护属性。如果拥有相邻的保留区域并想改变这些区域中的一些页面的保护属性，那么必须多次调用VirtualProtect函数。

15.7　清除物理存储器的内容

Windows 98　Windows 98不支持物理存储器内容的清除。

当你修改物理存储器的各个页面的内容时，系统将尽量设法将修改的内容保存在RAM中。但是，当应用程序运行时，从.exe文件、DLL文件和/或页文件加载页面就可能需要占用系统的RAM。由于系统要查找RAM的页面，以满足当前加载页面的需求，因此系统必须将RAM的已修改页面转到系统的页文件中。

Windows 2000提供了一个特性，使得应用程序能够提高它的性能，这个特性就是对物理存储器内容进行清除。清除存储器意味着你告诉系统，内存的一个或多个页面上的数据并没有被修改。如果系统正在搜索RAM的一个页面，并且选择一个已修改的页面，系统必须将RAM的这个页面写入页文件。这个操作的速度很慢，而且会影响系统的运行性能。对于大多数应用程序来说，可以让系统将你修改了的页面保留在系统的页文件中。

然而，有些应用程序使用内存的时间很短，然后就不再要求保留该内存的内容。为了提高性能，应用程序可以告诉系统不要将内存的某些页面保存在系统的页文件中。这是应用程序告诉系统数据页面尚未修改的一个基本方法。因此，如果系统选择将RAM的页面用于别的目的，那么该页面的内容就不必保存在页文件中，从而可以提高应用程序的运行性能。若要清除内存的内容，应用程序可以调用VirtualAlloc函数，在第三个参数中传递MEM_RESET标志。

如果在调用VirtualAlloc函数时引用的页面位于页文件中，系统将删除这些页面。下次应用程序访问内存时，便使用最初被初始化为0的RAM页面。如果清除了当前RAM中的页面内容，那么它们将被标上未修改的标记，这样它们将永远不会被写入页文件。注意，虽然RAM页面的内容没有被置0，但是不应该继续从内存的该页面读取数据。如果系统不需要RAM的这个页面，它将包含其原始内容。但是如果系统需要RAM的这个页面，系统就可以提取该页面。然后当你试图访问该页面的内容时，系统将给你一个已经删除内容的新页面。由于你无法控制这个行为特性，因此，当清除页面的内容后，你必须假定该页面的内容是无用信息。

当清除内存的内容时，有两件事情必须记住。首先，当调用VirtualAlloc函数时，基地址通常圆整为一个页面边界的值，而字节数则圆整为一个页面的整数。当清除页面的内容时，用这种办法圆整基地址和字节数是非常危险的，因此，当传递MEM_RESET标志时，VirtualAlloc将按反方向对这些值进行圆整。例如，有下面这个代码：

```
PINT pnData = (PINT) VirtualAlloc(NULL, 1024,
   MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
pn[0] = 100;
pn[1] = 200;
VirtualAlloc((PVOID) pnData, sizeof(int), MEM_RESET, PAGE_RE
```

China-pub.com 下载

该代码提交了一个内存页面，然后指明前4个字节（sizeof(int)）不再需要，可以被清除。但是，与所有内存操作一样，一切操作都必须在页面边界上并按页面增量来进行。结果，对上面这个内存清除函数的调用会失败（ VirtualAlloc函数返回NULL）。为什么呢？因为当将MEM_RESET传递给VirtualAlloc时，传递给函数的基地址被圆整为页面边界的值，字节数圆整为一个页面的整数。这样做是为了确保重要的数据不会丢失。在上面的例子中，字节数圆整后得出的是0，而清除0字节是不合法的。

要记住的第二件事情是， MEM_RESET标志始终必须自己单独使用，不能用OR将它与任何其他标志连接起来使用。下面这个函数调用总是失败的，它返回的是NULL。

```
PVOID pvMem = VirtualAlloc(NULL, 1024,
   MEM_RESERVE | MEM_COMMIT | MEM_RESET, PAGE_READWRITE);
```

将MEM_RESET标志与任何其他标志连接起来确实没有任何意义。

最后请注意，带有MEM_RESET标志的VirtualAlloc函数要求传递一个有效的页面保护属性，即使该函数不使用这个值，也必须传递该值。

MemReset示例应用程序

清单15-2列出的MemReset应用程序（“15 MemReset.exe ”）显示了MEM_RESET标志是如何运行的。该应用程序的源代码和资源文件位于本书所附光盘上的15-MemReset目录下。

MemReset.cpp代码执行的第一项操作是保留和提交一个物理存储器的区域。由于传递给VirtualAlloc函数的页面大小是1024，因此系统将自动把这个值圆整为系统的页面大小。这时，使用lstrcpy函数将一个字符串拷贝到该缓存，导致页面的内容被修改。如果系统后来决定它需要我们的数据页面占用的RAM页面，那么系统将首先将我们页面中的数据写入系统的页文件。当我们的应用程序后来试图访问该数据时，系统将自动把该页面从页文件重新加载到 RAM的另一个页面，这样我们就能够成功地访问该数据。

当该字符串被写入内存页面后，该代码便向用户显示一个消息框，询问是否需要在晚些时候访问这些数据。如果用户选定No按钮，那么该代码就迫使操作系统认为该页面中的数据没有通过调用VirtualAlloc函数并传递MEM_RESET标志而被修改。

为了说明内存的内容已经被清除，我们必须对系统的 RAM提出大量的使用需求。若要进行这项操作，可以分3步来进行：

1) 调用GlobalMemoryStatus函数，获取计算机中RAM的总容量。

2) 调用VirtualAlloc函数，提交该数量的内存。这项操作的运行速度非常快，因为在进程试图访问页面之前，系统实际上并不为该内存分配RAM。

3) 调用ZeroMemory函数，使新提交的页面可以被访问。这将给系统的RAM带来沉重的负担，导致当前正在RAM中的某些页面被写入页文件。

如果用户指明该数据将在以后被访问，那么该数据将不被清除，并且在以后访问该数据时将数据转入RAM。但是，如果用户指明以后将不再访问该数据，那么数据将被清除，并且系统不把数据写入页文件，这样就可以提高应用程序的运行性能。

当ZeroMemory函数返回时，代码将对数据页面的内容与原先写入页面的字符串进行比较。如果数据没有被清除，那么可以保证它们的内容是相同的。如果数据页面已经被清除，那么它们的内容可能相同，也可能不同。在MemReset程序中，它们的内容决不可能相同，因为RAM中的所有页面均被强制写入页文件中。但是，如果这个伪区域小于计算机中的 RAM总容量，那么原始内容有可能仍然在RAM中。如前所述，操作时务必小心。

China-pub.com
下载

清单15-2　MemReset示例应用程序



MemReset.cpp

```
/******************************************************************************
Module:  MemReset.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

   chWindows9xNotAllowed();

   TCHAR szAppName[]  = TEXT("MEM_RESET tester");
   TCHAR szTestData[] = TEXT("Some text data");

   // Commit a page of storage and modify its contents.
   LPTSTR pszData = (LPTSTR) VirtualAlloc(NULL, 1024,
      MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
   lstrcpy(pszData, szTestData);

   if (MessageBox(NULL, TEXT("Do you want to access this data later?"),
      szAppName, MB_YESNO) == IDNO) {

      // We want this page of storage to remain in our process but the
      // contents aren't important to us anymore.
      // Tell the system that the data is not modified.

      // Note: Because MEM_RESET destroys data, VirtualAlloc rounds
      // the base address and size parameters to their safest range.
      // Here is an example:
      //    VirtualAlloc(pvData, 5000, MEM_RESET, PAGE_READWRITE)
      // resets 0 pages on CPUs where the page size is greater than 4 KB
      // and resets 1 page on CPUs with a 4-KB page. So that our call to
      // VirtualAlloc to reset memory below always succeeds, VirtualQuery
      // is called first to get the exact region size.
      MEMORY_BASIC_INFORMATION mbi;
      VirtualQuery(pszData, &mbi, sizeof(mbi));
      VirtualAlloc(pszData, mbi.RegionSize, MEM_RESET, PAGE_READWRITE);
   }

   // Commit as much storage as there is physical RAM.
   MEMORYSTATUS mst;
   GlobalMemoryStatus(&mst);
```

China-pub.com 下载

```
   PVOID pvDummy = VirtualAlloc(NULL, mst.dwTotalPhys,
      MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

   // Touch all the pages in the dummy region so that any
   // modified pages in RAM are written to the paging file.
   ZeroMemory(pvDummy, mst.dwTotalPhys);

   // Compare our data page with what we originally wrote there.
   if (lstrcmp(pszData, szTestData) == 0) {

      // The data in the page matches what we originally put there.
      // ZeroMemory forced our page to be written to the paging file.
      MessageBox(NULL, TEXT("Modified data page was saved."),
         szAppName, MB_OK);
   } else {

      // The data in the page does NOT match what we originally put there
      // ZeroMemory didn't cause our page to be written to the paging file
      MessageBox(NULL, TEXT("Modified data page was NOT saved."),
         szAppName, MB_OK);
   }
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

15.8 地址窗口扩展——适用于Windows 2000

随着时间的推移，应用程序需要的内存越来越多。对于服务器应用程序来说，情况更是如此。由于越来越多的客户机对服务器提出访问请求，服务器的运行性能就会降低。为了提高运行性能，服务器应用程序必须在RAM中保存更多的数据，并且缩小磁盘的页面。其他类别的应用程序，比如数据库、工程设计和科学应用程序，也需要具备处理大块内存的能力。对于所有这些应用程序来说，32位地址空间是不够使用的。

为了满足这些应用程序的需要，Windows 2000提供了一个新特性。称为地址窗口扩展(AWE)。Microsoft创建AWE是出于下面两个目的：

- 允许应用程序对从来不在操作系统与磁盘之间交换的RAM进行分配。
- 允许应用程序访问的RAM大于进程的地址空间。

AWE基本上为应用程序提供了一种分配一个或多个RAM块的手段。当分配RAM块时，在进程的地址空间中是看不见这些RAM块的。后来，应用程序（使用VirtualAlloc函数）保留一个地址空间区域，这个区域就成为地址窗口。这时应用程序调用一个函数，每次将一个RAM块赋予该地址窗口。将一个RAM块赋予地址窗口的速度是非常快的（通常只需要几个毫秒）。

显然，通过单个地址窗口，每次只能访问一个RAM块。这使得你的代码很难实现，因为，当需要时，必须在你的代码中显式调用函数，才能将不同的RAM块赋予地址窗口。

下面的代码显示了如何使用AWE的方法：

```
// First, reserve a 1MB region for the address window
ULONG_PTR ulRAMBytes = 1024 * 1024
PVOID pvWindow = VirtualAlloc(NULL, ulRAMBytes,
```

China-pub.com
下载


```
   MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);

// Get the number of bytes in a page for this CPU platform
SYSTEM_INFO sinf;
GetSystemInfo(&sinf);

// Calculate the required number of RAM pages for the
// desired number of bytes
ULONG_PTR ulRAMPages = ulRAMBytes / sinf.dwPageSize

// Allocate array for RAM page's page frame numbers
ULONG_PTR aRAMPages[ulRAMPages];
// Allocate the pages of RAM (requires Lock Pages in Memory user right)
AllocateUserPhysicalPages(
   GetCurrentProcess(), // Allocate the storage for our process
   &ulRAMPages,      // Input: # of RAM pages, Output: # pages allocated
   aRAMPages);       // Output: Opaque array indicating pages allocated

// Assign the RAM pages to our window
MapUserPhysicalPages(pvWindow,   // The address of the address window
   ulRAMPages,                   // Number of entries in array
   aRAMPages);                   // Array of RAM pages

// Access the RAM pages via the pvWindow virtual address
:
:

// Free the block of RAM pages
FreeUserPhysicalPages(
   GetCurrentProcess(), // Free the RAM allocated for our process
   &ulRAMPages,         // Input: # of RAM pages, Output: # pages freed
   aRAMPages);          // Input: Array indicating the RAM pages to free

// Destroy the address window
VirtualFree(pvWindow, 0, MEM_RELEASE);
```

如你所见，AWE的使用非常简单。现在我们要说明该代码的一些有意思的情况。

对VirtualAlloc函数的调用保留了一个 1 MB的地址窗口。通常该地址窗口要大得多。你必须选定一个适合于应用程序需要的 RAM块大小的窗口大小。当然，你创建的最大窗口取决于你的地址空间中可用的最大相邻空闲地址块。 MEM_RESERVE标志用于指明我正在保留一个地址区域。 MEM_PHYSICAL标志用于指明这个区域最终将受 RAM物理存储器的支持。 AWE的局限性是，映射到地址窗口的所有内存必须是可读的和可写入的，因此 PAGE_READWRITE是可以传递VirtualAlloc函数的唯一有效的保护属性。此外，不能使用 VirtualProtect函数来修改这个保护属性。

RAM物理存储器的分配是非常简单的，只需要调用 AllocateUserPhysicalPages：

```
BOOL AllocateUserPhysicalPages(
   HANDLE hProcess,
   PULONG_PTR pulRAMPages,
   PULONG_PTR aRAMPages);
```

该函数负责分配pulRAMPages参数指明的值设定的 RAM页面的数量，并且将这些页面赋予hProcess参数标识的进程。

China-pub.com 下载

每个RAM页面由操作系统赋予一个页框号。当系统选择供分配用的 RAM页面时，它就将每个RAM页面的页框号填入aRAMPages参数指向的数组。页框号本身对应用程序没有任何用处，不应该查看该数组的内容，并且肯定不应该修改该数组中的任何一个值。注意，你不知道哪些RAM页面已经被分配给该内存块，也不应该去关注这个情况。当地址窗口显示 RAM块中的页面时，它们显示为一个相邻的内存块。这使得 RAM非常便于使用，并且使你可以不必了解系统内部的运行情况。

该函数返回时，pulRAMPages参数中的值用于指明该函数成功地分配的页面数量。这个数量通常与传递给函数的值是相同的，但是它也可能是个较小的值。

只有拥有页面的进程才能使用已经分配的 RAM页面，AWE不允许RAM页面被映射到另一个进程的地址空间。因此不能在进程之间共享RAM块。

注意 当然，物理RAM是一种非常宝贵的资源，并且应用程序只能分配尚未指定用途的RAM。应该非常节省地使用 AWE，否则你的进程和其他进程将会过分地在内存与磁盘之间进行页面的交换，从而严重影响系统的运行性能。此外，如果可用 RAM的数量比较少，也会对系统创建新进程、线程和其他资源的能力产生不利的影响。应用程序可以使用GlobalMemoryStatusEx函数来监控物理存储器的使用情况。

为了保护RAM的分配，AllocateUserPhysicalPages函数要求调用者拥有Lock Pages in Memory（锁定内存中的页面）的用户权限，并且已经激活该权限，否则该函数的运行将会失败。按照默认设置，该权限不被赋予任何用户或用户组。该权限被赋予 Local System（本地系统）帐户，它通常用于服务程序。如果想要运行一个调用AllocateUserPhysicalPages函数的交互式应用程序，那么管理员必须在你登录和运行应用程序之前为你赋予该权限。

Windows 2000 在Windows 2000中，可以执行下列步骤，打开Lock Pages in Memory用户权限：

1) 单击Start按钮，选定Run菜单项，打开Computer Management MMC控制台。在Run框中，键入“ Compmgmt.msc/a ”，再单击OK按钮。

2) 如果在左边的窗格中没有显示 Local Computer Policy（本地计算机政策）项，那么在控制台菜单中选定 Add/Remove Snap-ins（添加/删除咬接项（snap-in））。在Standalone选项卡上，从Snap-ins Added To（咬接项添加到）组合框中选定 Computer Management(local)。现在单击Add按钮，显示Add Standalone Snap-in（添加独立咬接项）对话框。从Available Standalone Snap-ins（可用独立咬接项）中选定Group Policy（组政策）。并单击Add按钮。在Select Group Policy Object（选定组政策对象）对话框中，保留默认值，并单击Finish按钮。单击Add Standalone Snap-in对话框上的Close按钮，再单击Add/Remove Snap-in对话框上的OK按钮。这时，在Computer Management控制台的左窗格中就可以看到Local Computer Policy项。

3) 在控制台的左窗格中，双击下列项目，将它们展开：Local Computer Policy(本地计算机政策)、Computer Configuration(计算机配置)、Windows Settings（窗口设置）、Security Settings（安全性设置）和Local Policy（本地政策）。然后选定User Rights Assignment（用户权限赋值）项。

4) 在右窗格中，选定Lock Pages in Memory属性。

5) 从Action（操作）菜单中选定Security，显示Lock Pages in Memory对话框，单

China-pub.com
下载

击Add按钮。使用 Select Users or Group对话框，添加你想为其赋予 Lock Pages in Memory用户权限的用户和/或用户组。单击OK按钮，退出每个对话框。

当用户登录时，他将被赋予相应的用户权限。如果你只是将 Lock Pages in Memory权限赋予你自己，那么必须在该权限生效前退出并重新登录。

现在我们已经创建了地址窗口并且分配了一个 RAM块，可以通过调用 MapUserPhysical Pages函数将该RAM块赋予该地址窗口：

```
BOOL MapUserPhysicalPages(
   PVOID pvAddressWindow,
   ULONG_PTR ulRAMPages,
   PULONG_PTR aRAMPages);
```

第一个参数 pvAddressWindow用于指明地址窗口的虚拟地址，第二和第三个参数 ulRAMPages和aRAMPages用于指明该地址窗口中可以看到多少个 RAM页面以及哪些页面可以看到。如果窗口小于你试图映射的页面数量，那么函数运行就会失败。 Microsoft设置这个函数的主要目的是使它运行起来非常快。一般来说， MapUserPhysicalPages函数能够在几个微秒内映射该RAM块。

注意　也可以调用MapUserPhysicalPages函数来取消对当前RAM块的分配，方法是为 aRAMPages参数传递NULL。下面是它的一个例子：

```
// First, reserve a 1MB region for the address window
ULONG_PTR ulRAMBytes = 1024 * 1024
```

一旦RAM块被分配给地址窗口，只需要引用相对于地址窗口的基地址（在我的示例代码中是pvWindow）的虚拟地址，就可以很容易地访问该 RAM内存。

当不再需要RAM块时，应该调用FreeUserPhysicalPages函数将它释放：

```
BOOL FreeUserPhysicalPages(
   HANDLE hProcess,
   PULONG_PTR pulRAMPages,
   PULONG_PTR aRAMPages);
```

第一个参数hProcess用于指明哪个进程拥有你试图释放的 RAM页面。第二和第三个参数用于指明你要释放多少个页面和这些页面的页框号。如果该 RAM块目前已经被映射到该地址窗口，那么它将被取消映射并被释放。

最后，为了彻底清除页面，我仅仅调用了 VirtualFree函数，传递窗口的虚拟基地址，为区域大小传递0，再传递MEM_RELEASE，将地址窗口释放掉。

我的简单的示例代码创建了单个地址窗口和单个 RAM块。这使得我的应用程序能够访问没有与磁盘进行数据交换的 RAM。但是，应用程序也能够创建若干个地址窗口，并且可以分配若干个RAM块。虽然这些RAM块可以分配给任何一个地址窗口，但是系统不允许单个 RAM块同时出现在两个地址窗口中。

64位Windows 2000全面支持AWE。对使用AWE的32位应用程序进行移植是非常容易和简单的。不过对于 64位应用程序来说， AWE的用处比较小，因为进程的地址空间太大了。但是AWE仍然是有用的，因为它使得应用程序能够分配不与磁盘进行数据交换的物理 RAM。

AWE示例应用程序

清单15-3列出的AWE应用程序（“ 15-AWE.exe ”）显示了如何创建多个地址窗口和如何将

China-pub.com 下载

不同的内存块分配给这些窗口。该应用程序的源代码和资源文件位于本书所附光盘上的 15-AWE目录下。当启动该程序时，它在内部创建两个地址窗口区域，并且分配两个 RAM块。

首先，第一个RAM块用字符串“ Text in Storage 0 ”填入，第二个RAM块用字符串“ Text in Storage 1 ”填入。然后，第一个RAM块被赋予第一个地址窗口，第二个 RAM块被赋予第二个地址窗口。该应用程序的窗口反映了这个情况（见图 15-4）。



图15-4 AWE示例应用程序的窗口

使用该窗口，可以进行一些试验。首先，可以使用每个地址窗口的组合框将 RAM块赋予各个地址窗口。该组合框也提供了一个 No Storage选项，用于从地址窗口中撤消对任何内存的映射。其次，编辑窗口中的文本将会更新地址窗口中当前选定的 RAM块。

如果试图将一个RAM块同时赋予两个地址窗口，就会出现图 15-5所示的消息，因为 AWE不支持这种操作。



图15-5 消息显示

该示例应用程序的源代码是非常清楚的。为了使 AWE的操作更加容易些，我创建了 3个C++类，它们包含在AddrWindows.h文件中。第一个类是CSystemInfo，它是GetSystemInfo函数中的一个非常简单的包装类，另外两个类均用于创建 CSystemInfo类的实例。

第二个C++类是CAddrWindow，它用于封装一个地址窗口。 Create方法基本上用于保留一个地址窗口，Destroy方法则用于撤消一个地址窗口，UnmapStorage方法用于取消当前赋予地址窗口的任何RAM块的映像，而PVOID类型转换操作符方法只是用于返回地址窗口的虚拟地址。

第三个C++类是CAddrWindowStorage，它用于封装一个可以赋予 CAddrWindow对象的RAM块。Allocate方法用于激活Lock Pages in Memory用户权限，并设法分配RAM块，然后停用该用户权限。 Free方法用于释放RAM块。HowManyPagesAllocated方法返回已经成功地分配的页面数量。 MapStorage和UnmapStorage方法则用于与CAddrWindow对象之间进行RAM块的映射和取消映射。

使用这些C++类，可以使示例应用程序的实现变得容易得多。该示例应用程序创建了两个CAddrWindow对象和两个CAddrWindowStorage对象。代码的其余部分只是用于在正确的时间为正确的对象调用正确的方法而已。

清单15-3 AWE示例应用程序

AWE.cpp

```
/******************************************************************************
```

China-pub.com
下载


```
/******************************************************************************
Module:  AWE.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include "..\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>
#include "AddrWindow.h"
#include "Resource.h"
//////////////////////////////////////////////////////////////////////////////


CAddrWindow g_aw[2];              // 2 memory address windows
CAddrWindowStorage g_aws[2];      // 2 storage blocks
const ULONG_PTR g_nChars = 1024; // 1024 character buffers


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_AWE);

   // Create the 2 memory address windows
   chVERIFY(g_aw[0].Create(g_nChars * sizeof(TCHAR)));
   chVERIFY(g_aw[1].Create(g_nChars * sizeof(TCHAR)));

   // Create the 2 storage blocks
   if (!g_aws[0].Allocate(g_nChars * sizeof(TCHAR))) {
      chFAIL("Failed to allocate RAM.\nMost likely reason: "
         "you are not granted the Lock Pages in Memory user right.");
   }
   chVERIFY(g_aws[1].Allocate(g_nChars * sizeof(TCHAR)));

   // Put some default text in the 1st storage block
   g_aws[0].MapStorage(g_aw[0]);
   lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 0"));

   // Put some default text in the 2nd storage block
   g_aws[1].MapStorage(g_aw[0]);
   lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 1"));

   // Populate the dialog box controls
   for (int n = 0; n <= 1; n++) {
      // Set the combo box for each address window
      int id = ((n == 0) ? IDC_WINDOW0STORAGE : IDC_WINDOW1STORAGE);
      HWND hwndCB = GetDlgItem(hwnd, id);
      ComboBox_AddString(hwndCB, TEXT("No storage"));
      ComboBox_AddString(hwndCB, TEXT("Storage 0"));
      ComboBox_AddString(hwndCB, TEXT("Storage 1"));
      // Window 0 shows Storage 0, Window 1 shows Storage 1
      ComboBox_SetCurSel(hwndCB, n + 1);
      FORWARD_WM_COMMAND(hwnd, id, hwndCB, CBN_SELCHANGE, SendMessage);
```

China-pub.com 下载

```
      Edit_LimitText(GetDlgItem(hwnd,
         (n == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT), g_nChars);
   }

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {

   case IDCANCEL:
      EndDialog(hwnd, id);
      break;

   case IDC_WINDOW0STORAGE:
   case IDC_WINDOW1STORAGE:
      if (codeNotify == CBN_SELCHANGE) {

         // Show different storage in address window
         int nWindow  = ((id == IDC_WINDOW0STORAGE) ? 0 : 1);
         int nStorage = ComboBox_GetCurSel(hwndCtl) - 1;

         if (nStorage == -1) {   // Show no storage in this window
            chVERIFY(g_aw[nWindow].UnmapStorage());
         } else {
            if (!g_aws[nStorage].MapStorage(g_aw[nWindow])) {
               // Couldn't map storage in window
               chVERIFY(g_aw[nWindow].UnmapStorage());
               ComboBox_SetCurSel(hwndCtl, 0);  // Force "No storage"
               chMB("This storage can be mapped only once.");
            }
         }

         // Update the address window's text display
         HWND hwndText = GetDlgItem(hwnd,
            ((nWindow == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT));
         MEMORY_BASIC_INFORMATION mbi;
         VirtualQuery(g_aw[nWindow], &mbi, sizeof(mbi));
         // Note: mbi.State == MEM_RESERVE if no storage is in address window
         EnableWindow(hwndText, (mbi.State == MEM_COMMIT));
         Edit_SetText(hwndText, IsWindowEnabled(hwndText)
            ? (PCSTR) (PVOID) g_aw[nWindow] : TEXT("(No storage)"));
      }
      break;

   case IDC_WINDOW0TEXT:
   case IDC_WINDOW1TEXT:
      if (codeNotify == EN_CHANGE) {
         // Update the storage in the address window
         int nWindow = ((id == IDC_WINDOW0TEXT) ? 0 : 1);
         Edit_GetText(hwndCtl, (PSTR) (PVOID) g_aw[nWindow], g_nChars);
      }
```

China-pub.com
下载


```
         break;
      }
   }

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   chWindows2000Required();

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_AWE), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

AWE.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
//
// Dialog
//

IDD_AWE DIALOG DISCARDABLE  0, 0, 288, 45
STYLE DS_SETFOREGROUND | DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
CAPTION "Address Windowing Extensions"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Window 0:",IDC_STATIC,4,6,35,8
    COMBOBOX        IDC_WINDOW0STORAGE,44,4,80,58,CBS_DROPDOWNLIST |
                    WS_TABSTOP
    EDITTEXT        IDC_WINDOW0TEXT,132,4,152,14,ES_AUTOHSCROLL
    LTEXT           "Window 1:",IDC_STATIC,4,28,35,8
    COMBOBOX        IDC_WINDOW1STORAGE,44,25,80,58,CBS_DROPDOWNLIST |
                    WS_TABSTOP
    EDITTEXT        IDC_WINDOW1TEXT,132,25,152,14,ES_AUTOHSCROLL
END
////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_AWE, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 281
        TOPMARGIN, 7
        BOTTOMMARGIN, 38
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
```

China-pub.com
下载


```
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_AWE                 ICON    DISCARDABLE     "AWE.ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

AddrWindow.h

```
/******************************************************************************
Module:  AddrWindow.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once


////////////////////////////////////////////////////////////////////////////


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>


////////////////////////////////////////////////////////////////////////////
 class CSystemInfo : public SYSTEM_INFO {
 public:
    CSystemInfo() { GetSystemInfo(this); }
 };


 ////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
class CAddrWindow {
public:
   CAddrWindow()  { m_pvWindow = NULL; }
   ~CAddrWindow() { Destroy(); }

   BOOL Create(SIZE_T dwBytes, PVOID pvPreferredWindowBase = NULL) {
      // Reserve address window region to view physical storage
      m_pvWindow = VirtualAlloc(pvPreferredWindowBase, dwBytes,
         MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
      return(m_pvWindow != NULL);
   }

   BOOL Destroy() {
      BOOL fOk = TRUE;
      if (m_pvWindow != NULL) {
         // Destroy address window region
         fOk = VirtualFree(m_pvWindow, 0, MEM_RELEASE);
         m_pvWindow = NULL;
      }
      return(fOk);
   }

   BOOL UnmapStorage() {
      // Unmap all storage from address window region
      MEMORY_BASIC_INFORMATION mbi;
      VirtualQuery(m_pvWindow, &mbi, sizeof(mbi));
      return(MapUserPhysicalPages(m_pvWindow,
         mbi.RegionSize / sm_sinf.dwPageSize, NULL));
   }

   // Returns virtual address of address window
   operator PVOID() { return(m_pvWindow); }

   private:
      PVOID m_pvWindow;    // Virtual address of address window region
      static CSystemInfo sm_sinf;
   };

   ////////////////////////////////////////////////////////////////////////////

   CSystemInfo CAddrWindow::sm_sinf;

   ////////////////////////////////////////////////////////////////////////////

   class CAddrWindowStorage {
   public:
      CAddrWindowStorage()  { m_ulPages = 0; m_pulUserPfnArray = NULL; }
      ~CAddrWindowStorage() { Free(); }

      BOOL Allocate(ULONG_PTR ulBytes) {
         // Allocate storage intended for an address window
```

China-pub.com
下载


```
         Free();  // Cleanup this object's existing address window

         // Calculate number of pages from number of bytes
         m_ulPages = (ulBytes + sm_sinf.dwPageSize) / sm_sinf.dwPageSize;

         // Allocate array of page frame numbers
         m_pulUserPfnArray = (PULONG_PTR)
            HeapAlloc(GetProcessHeap(), 0, m_ulPages * sizeof(ULONG_PTR));

         BOOL fOk = (m_pulUserPfnArray != NULL);
         if (fOk) {
            // The "Lock Pages in Memory" privilege must be enabled
            EnablePrivilege(SE_LOCK_MEMORY_NAME, TRUE);
            fOk = AllocateUserPhysicalPages(GetCurrentProcess(),
               &m_ulPages, m_pulUserPfnArray);
            EnablePrivilege(SE_LOCK_MEMORY_NAME, FALSE);
         }
         return(fOk);
      }

      BOOL Free() {
         BOOL fOk = TRUE;
         if (m_pulUserPfnArray != NULL) {
            fOk = FreeUserPhysicalPages(GetCurrentProcess(),
               &m_ulPages, m_pulUserPfnArray);
            if (fOk) {
               // Free the array of page frame numbers
               HeapFree(GetProcessHeap(), 0, m_pulUserPfnArray);
               m_ulPages = 0;
               m_pulUserPfnArray = NULL;
            }
         }
         return(fOk);
      }

      ULONG_PTR HowManyPagesAllocated() { return(m_ulPages); }

      BOOL MapStorage(CAddrWindow& aw) {
         return(MapUserPhysicalPages(aw,
            HowManyPagesAllocated(), m_pulUserPfnArray));
      }

      BOOL UnmapStorage(CAddrWindow& aw) {
         return(MapUserPhysicalPages(aw,
            HowManyPagesAllocated(), NULL));
      }

   private:
      static BOOL EnablePrivilege(PCTSTR pszPrivName, BOOL fEnable = TRUE) {

         BOOL fOk = FALSE;    // Assume function fails
         HANDLE hToken;

         // Try to open this process's access token
         if (OpenProcessToken(GetCurrentProcess(),
```

China-pub.com 下载

```
         TOKEN_ADJUST_PRIVILEGES, &hToken)) {

         // Attempt to modify the "Lock pages in Memory" user right
         TOKEN_PRIVILEGES tp = { 1 };
         LookupPrivilegeValue(NULL, pszPrivName, &tp.Privileges[0].Luid);
         tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
         AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);

         fOk = (GetLastError() == ERROR_SUCCESS);
         CloseHandle(hToken);
      }
      return(fOk);
   }

private:
   ULONG_PTR  m_ulPages;         // Number of storage pages
   PULONG_PTR m_pulUserPfnArray; // Page frame number array

private:
   static CSystemInfo sm_sinf;
};


//////////////////////////////////////////////////////////////////////////////


CSystemInfo CAddrWindowStorage::sm_sinf;


//////////////////////////////// End of File //////////////////////////////////
```

China-pub.com
下载

第16章　线程的堆栈

有时系统会在你自己进程的地址空间中保留一些区域。第3章讲过，对于进程和线程环境块来说，就会出现这种情况。另外，系统也可以在你自己进程的地址空间中为线程的堆栈保留一些区域。

每当创建一个线程时，系统就会为线程的堆栈（每个线程有它自己的堆栈）保留一个堆栈空间区域，并将一些物理存储器提交给这个已保留的区域。按照默认设置，系统保留1 MB的地址空间并提交两个页面的内存。但是，这些默认值是可以修改的，方法是在你链接应用程序时设定Microsoft的链接程序的/STACK选项：

```
/STACK:reserve[,commit]
```

当创建一个线程的堆栈时，系统将会保留一个链接程序的/STACK开关指明的地址空间区域。但是，当调用CreateThread或_beginthreadex函数时，可以重载原先提交的内存数量。这两个函数都有一个参数，可以用来重载原先提交给堆栈的地址空间的内存数量。如果设定这个参数为0，那么系统将使用/STACK开关指明的已提交的堆栈大小值。后面将假定我们使用默认的堆栈大小值，即1 MB的保留区域，每次提交一个页面的内存。

图16-1显示了在页面大小为4 KB的计算机上的一个堆栈区域的样子（保留的起始地址是0x08000000）。该堆栈区域和提交给它的所有物理存储器均拥有页面保护属性PAGE_READWRITE。

内存地址	页面状态
0x080FF000	堆栈顶部：已提示的页面
0x080FD000	带有保护属性标志的已提交页面
0x080FD000	保留页面
0x08003000	保留页面
0x08002000	保留页面
0x08001000	保留页面
0x08000000	堆栈底部：保留页面

图16-1　线程的堆栈区域刚刚创建时的样子

China-pub.com 下载

当保留了这个区域后，系统将物理存储器提交给区域的顶部的两个页面。在允许线程启动运行之前，系统将线程的堆栈指针寄存器设置为指向堆栈区域的最高页面的结尾处（一个非常接近0x08100000的地址）。这个页面就是线程开始使用它的堆栈的位置。从顶部向下的第二个页面称为保护页面。当线程调用更多的函数来扩展它的调用树状结构时，线程将需要更多的堆栈空间。

每当线程试图访问保护页面中的存储器时，系统就会得到关于这个情况的通知。作为响应，系统将提交紧靠保护页面下面的另一个存储器页面。然后，系统从当前保护页面中删除保护页面的保护标志，并将它赋予新提交的存储器页面。这种方法使得堆栈存储器只有在线程需要时才会增加。最终，如果线程的调用树继续扩展，堆栈区域就会变成图16-2所示的样子。

如图16-2所示，假定线程的调用树非常深，堆栈指针（CPU寄存器指向堆栈内存地址）0x08003004。这时，当线程调用另一个函数时，系统必须提交更多的物理存储器。但是，当系统将物理存储器提交给0x08001000地址上的页面时，系统执行的操作与它给堆栈的其他内存区域提交物理存储器时的操作并不完全一样。图16-3显示了堆栈的保留内存区域的样子。

如你预计的那样，从地址0x08002000开始的页面的保护属性已经被删除，物理存储器被提交给从0x08001000地址开始的页面。它们的差别是，系统并不将保护属性应用于新的物理存储器页面（0x08001000）。这意味着该堆栈已保留的地址空间区域包含了它能够包含的全部物理存储器。最底下的页面总是被保留的，从来不会被提交。下面将要说明它的原因。

当系统将物理存储器提交给0x08001000地址上的页面时，它必须再执行一个操作，即它要引发一个EXCEPTION_STACK_OVERFLOW异常处理（在WinNT.h文件中定义为0xC00000FD）。通过使用结构化异常处理（SEH），你的程序将能得到关于这个异常处理条件的通知，并且能够实现适度恢复。关于SEH的详细说明，请参见第23、24和25章的内容。本章结尾处的Summation示例应用程序将展示如何对堆栈溢出进行适度恢复。

内存地址	页面状态
0x080FF000	堆栈顶部：已提交的页面
0x080FD000	已提交的页面
0x080FD000	已提交的页面
0x08003000	已提交的页面
0x08002000	带有保护属性标志的已提交页面
0x08001000	保留页面
0x08000000	堆栈底部：保留页面

图16-2 几乎完整的线程堆栈区域

China-pub.com
下载



图16-3　完整的线程堆栈区域

如果在出现堆栈溢出异常条件之后，线程继续使用该堆栈，那么在 0x08001000地址上的页面中的全部内存均将被使用，同时，该线程将试图访问从 0x08000000开始的页面中的内存。当该线程试图访问这个保留的（未提交的）内存时，系统就会引发一个访问违规异常条件。如果在线程试图访问该堆栈时引发了这个访问违规异常条件，线程就会陷入很大的麻烦之中。这时，系统就会接管控制权，并终止进程的运行 —— 不仅终止线程的运行，而切终止整个进程的运行。系统甚至不向用户显示一个消息框，整个进程都消失了！

下面要说明为什么堆栈区域的最后一个页面始终被保留着。这样做的目的是为了防止不小心改写进程使用的其他数据。可以看到，在 0x07FF000这个地址上（0x08000000下面的一个页面），另一个地址空间区域已经提交了物理存储器。如果 0x08000000地址上的页面包含物理存储器，系统将无法抓住线程访问已保留堆栈区域的尝试。如果堆栈深入到已保留堆栈区域的下面，那么线程中的代码就会改写进程的地址空间中的其他数据，这是个非常难以抓住的错误。

16.1　Windows 98下的线程堆栈

在Windows 98下，堆栈的行为特性与 Windows 2000下的堆栈非常相似。但是它们之间存在某些重大的差别。

图16-4显示了Windows 98下1 MB的堆栈的各个区域的样子（从 0x00530000地址上开始保留）。

首先请注意，尽管我们想要创建的堆栈大小最大只有 1 MB，但是堆栈区域的大小实际上是1 MB加128 KB 。在Windows 98中，每当为一个堆栈保留一个区域时，系统保留的区域实际上比要求的尺寸要大128 KB。该堆栈位于该区域的中间，堆栈的前面有一个 64 KB的块，堆栈的后面是另一个64 KB的块。

China-pub.com 下载

内存地址	大小	页面状态
0x00640000	16页面 (65 536字节)	堆栈顶部：保留供堆栈下溢时使用
0x0063F000	1页面 (4096字节)	带有PAGE_READWRITE保护属性的已提交页面，堆栈在用
0x0063E000	1页面 (4096字节)	仿真PAGE_GUARD标志的PAGE_NO_ACCESS页面
0x00638000	6页面 (24 576字节)	保留供堆栈溢出时使用的页面
0x00637000	1页 (4096字节)	带有PAGE_READ_ WRITE保护属性的已提交页面，用于与16位组件相兼容
0x006540000	247页面 (1 011 712字节)	保留页面，供堆栈扩展时使用
0x00530000	16页面 (65 536字节)	堆栈底部：保留供堆栈溢出时使用

图16-4 Windows 98下线程的堆栈区域刚刚创建时的样子

堆栈开始处的64 KB用于抓取堆栈的溢出条件，而堆栈后面的 64 KB则用于抓取堆栈的下溢条件。若要了解为什么需要检测堆栈下溢条件，请看下面这个代码段：

```
int WINAPI WinMain (HINSTANCE hinstExe, HINSTANCE,
   PSTR pszCmdLine, int nCmdShow) {

   char szBuf[100];
   szBuf[10000] = 0; // Stack underflow

   return(0);
}
```

当该函数的赋值语句执行时，便尝试访问线程堆栈结尾处之外的内存。当然，编译器和链接程序不会抓住上面代码中的错误，但是，如果应用程序是在Windows 98下运行，那么当该语句执行时，就会引发访问违规。这是Windows 98的一个出色特性，而Windows 2000是没有的。在Windows 2000中，可以在紧跟线程堆栈的后面建立另一个区域。如果出现这种情况，并且你试图访问你的堆栈外面的内存，那么你将会破坏与进程的另一个部分相关的内存，而系统将不会发现这个情况。

需要指出的第二个重要差别是，没有一个页面具有 PAGE_GUARD保护属性标志。由于Windows 98不支持这个标志，所以它使用一个不同的方法来扩展线程的堆栈。 Windows 98将紧靠堆栈下面的已提交页面标记为PAGE_NOACCESS保护属性（图16-4中的地址0x0063E000）。然后，当线程接触读/写页面下面的页面时，将会发生访问违规。系统抓住这个访问违规，将不能访问的页面改为读写页面，并提交前一个保护页面下面的一个新保护页面。

第三个应该注意的差别是图16-4中的0x00637000地址上的单个PAGE_READWRITE内存页面。这个页面是为了实现与 16位Windows相兼容而存在的。虽然 Microsoft从未将它纳入文档，但是开发人员发现16位应用程序的堆栈段（SS）开始处的16个字节包含了关于16位应用程序的堆栈、本地堆栈和本地原子表的信息。由于在 Windows 98上运行的Win32应用程序常常调用16位DLL组件，有些16位组件认为这些信息可以在堆栈段的开始处得到，因此 Microsoft不得不在Windows 98中仿真这些字节的设置。当32位代码转换为16位代码时，Windows 98 将把一个16

China-pub.com
下载

位CPU选择器映射到32位堆栈，并且将堆栈段寄存器设置为指向0x00637000地址上的页面。这时该16位代码就可以访问堆栈段的开始处的16个字节，并且可以继续运行而不会出任何问题。

现在，当Windows 98扩大它的线程堆栈时，它将继续扩大0x0063F000地址上的内存块。它也会不断地将保护页面下移，直到1 MB的堆栈内存被提交为止。然后保护页面消失，就像在Windows 2000下运行的情况一样。系统还继续为了16位Windows组件的兼容性而将页面下移，最后该页面将进入堆栈区域开始处的64 KB的内存块中。因此，Windows 98中一个完全提交的堆栈将类似图16-5所示的样子。

内存地址	大小	页面状态
0x00640000	16页面 (65 536字节)	堆栈顶部：保留供堆栈下溢时使用
0x00540000	256页面 (1MB)	带有PAGE_READWRITE保护属性的已提交页面，堆栈在用
0x00539000	7页面 (28 672字节)	供堆栈溢出时使用的保留页面
0x00538000	1页面 (4096字节)	带有PAGE_READWRITE保护属性的已提交页面，用于与16位组件相兼容
0x00538000	8页面 (32 768字节)	堆栈底部：保留供堆栈溢出时使用

图16-5 Windows 98下的一个完整的线程堆栈区域

16.2 C/C++运行期库的堆栈检查函数

C/C++运行期库包含一个堆栈检查函数。当编译源代码时，编译器将在必要时自动生成对该函数的调用。堆栈检查函数的作用是确保页面被适当地提交给线程的堆栈。下面让我们来看一个例子。

这是一个小型函数，它需要相当多的内存用于它的局部变量：

```
void SomeFunction () {
   int nValues[4000];

   // Do some processing with the array.
   nValues[0] = 0;  // Some assignment
}
```

该函数至少需要16 000个字节（4000 x sizeof(int),每个整数是4个字节）的堆栈空间，以便放置整数数组。通常情况下，编译器生成的用于分配该堆栈空间的代码只是将CPU的堆栈指针递减16 000个字节。但是，在程序试图访问内存地址之前，系统并不将物理存储器分配给堆栈区域的这个较低区域。

在使用4 KB或8 KB页面的系统上，这个局限性可能导致一个问题出现。如果初次访问堆栈是在低于保护页面的一个地址上进行的（如上面这个代码中的赋值行所示），那么线程将访问已经保留的内存并且引发访问违规。为了确保能够成功地编写上面所示的函数，编译器将插入对C运行期库的堆栈检查函数的调用。

当编译程序时，编译器知道你针对的CPU系统的页面大小。x86编译器知道页面大小是4 KB，Alpha编译器知道页面大小是8 KB。当编译器遇到程序中的每个函数时，它能确定该函数

China-pub.com 下载

需要的堆栈空间的数量。如果该函数需要的堆栈空间大于目标系统的页面大小，编译器将自动插入对堆栈检查函数的调用。

下面这个伪代码显示了堆栈检查函数执行什么操作。之所以称它是伪代码，是因为这个函数通常是由编译器供应商用汇编语言来实现的：

```
// The C run-time library knows the page size for the target system.
#ifdef _M_ALPHA
#define PAGESIZE   (8 * 1024)   // 8-KB page
#else
#define PAGESIZE   (4 * 1024)   // 4-KB page
#endif

void StackCheck(int nBytesNeededFromStack) {
   // Get the stack pointer position.
   // At this point, the stack pointer has NOT been decremented
   // to account for the function's local variables.
   PBYTE pbStackPtr = (CPU's stack pointer);
  while (nBytesNeededFromStack >= PAGESIZE) {
      // Move down a page on the stack--should be a guard page.
      pbStackPtr -= PAGESIZE;

      // Access a byte on the guard page--forces new page to be
      // committed and guard page to move down a page.
      pbStackPtr[0] = 0;

      // Reduce the number of bytes needed from the stack.
      nBytesNeededFromStack -= PAGESIZE;
   }

   // Before returning, the StackCheck function sets the CPU's
   // stack pointer to the address below the function's
   // local variables.
}
```

Microsoft 的Visual C++确实提供了一个编译器开关，使你能够控制一个页面大小的阈值，这个阈值可供编译器用来确定何时添加对 StackCheck函数的自动调用。只有当确切地知道究竟在进行什么操作并且有着特殊需要时，才能使用这个编译器开关。对于绝大多数应用程序和DLL来说，都不应该使用这个开关。

16.3 Summation示例应用程序

本章后面清单16-1中的Summation（“16 Summation.exe”）示例应用程序展示了如何使用异常过滤器和异常处理程序以便对堆栈溢出进行适度恢复的方法。该应用程序的源代码和资源文件均位于本书所附光盘上的16-Summation目录下。若要全面了解该应用程序是如何运行的，可以参见关于SEH的有关章节。

Summation应用程序用于计算从 0到x的全部数字的总和，其中 x是用户输入的一个数字。当然，进行这项操作的最简单的方法是创建一个称为 Sum的函数，它只是进行下面的计算：

```
Sum = (x * (x + 1)) / 2;
```

然而对于这个例子来说，我将 Sum编写为一个递归函数，这样，如果输入较大的数字，它将使用大量的堆栈空间。

当程序启动运行时，它将显示图16-6所示的对话框。

China-pub.com
下载

在这个对话框中，你将一个数字输入编辑控件，然后单击 Calculate按钮。这使程序创建一个新线程，该线程的唯一作用是将 0到x的全部数字进行相加。当这个新线程运行时，程序的主线程通过调用 WaitForSingleObject函数并传递新线程的句柄，等待线程运行的结果。当新线程运行终止时，系统就唤醒主线程。主线程取出合计的总数，方法是调用 GetExitCodeThread函数来获得新线程的退出代码。最后，最重要的一点是，主线程要关闭新线程的句柄，这样，系统就能完全撤消线程对象，并且使应用程序不会出现资源的泄漏。

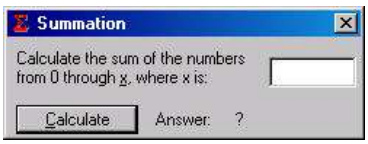


图16-6　Summation 对话框

这时，主线程要查看合计线程的退出代码。退出代码 UNIT_MAX指明出现了一个错误，即合计线程在计算数字的总数时产生了堆栈溢出，为此，主线程显示一个消息框，以说明这个情况。如果退出代码不是 UNIX_MAX，那么合计线程将成功地结束其运行，而退出代码则是合计。在这种情况下，主线程只是将合计的结果放入对话框。

下面让我们转入合计线程的介绍。该线程的线程函数称为 SumThreadFunc。当主线程创建该线程时，它将应该合计的各个整数的数量作为唯一的参数来传递，这个参数就是 pvParam。然后，函数将uSum变量初始化为UNIX_MAX，这意味着该函数认为它将不会成功地完成运行。接着，SumThreadFunc建立SEH，这样，它就能够抓住线程运行时出现的任何异常条件。然后调用递归函数Sum来计算总数。

如果成功地计算出总数，SumThreadFunc函数返回uSum变量的值，这是线程的退出代码。但是，如果在sum函数运行时引发了一个异常条件，系统将立即对 SEH过滤器表达式进行计算。换句话说，系统将调用 FilterFunc函数，并为它传递用于标识引发的异常条件的代码。如果是堆栈溢出异常，那么该代码是 EXCEPTION_STACK_OVERFLOW。如果想要观察程序适度处理堆栈溢出的异常条件，那么请告诉程序计算前面的 44 000个数字的总数。

我的FilterFunc函数非常简单。它查看是否出现了堆栈溢出异常条件。如果没有出现，它返回EXCEPTION_CONTINUE_SEARCH。否则，该过滤器返回 EXCEPTION_EXECUTE_HANDLER。它向系统指明过滤器预计到了这个异常条件，同时， Except块中包含的代码应该执行。对于这个示例应用程序来说，异常处理程序没有什么特殊的操作需要执行，而是让线程恰当地退出并返回代码UNIT_MAX（这是uSumNum中的值）。父线程将会看到这个特殊的返回值，并向用户显示一条警告消息。

要说明的最后一点是，为什么要在 Sum函数自己的线程中运行 Sum函数，而不是在主线程中建立一个SEH块，并从try块中调用Sum函数。创建这个独立线程的理由有三：

首先，每次创建一个线程时，它会得到它自己的 1 MB堆栈区域。如果我从主线程中调用 Sum函数，那么有些堆栈空间就已经在使用了，因此Sum函数将无法使用完整的1 MB堆栈空间。然而，我的示例应用程序是个简单的程序，也许它不需要使用那么多完整的堆栈空间，不过其他程序可能要复杂得多。我能够非常容易地想像到这样一种情况，即 Sum函数能够成功地计算出从1到1000的所有整数的总和。然后，当 Sum在以后被再次调用时，堆栈可能变得更深，从而在Sum试图只计算从0到750之间的整数的总和时，导致堆栈溢出的发生。因此，为了使 Sum函数的运行具备更好的一致性，我设法使它拥有一个尚未被其他代码使用过的完整的堆栈。

使用独立线程的第二个原因是，关于堆栈溢出的异常条件，线程只能得到一次通知。如果我调用主线程中的Sum函数，并且发生了堆栈溢出，那么就可以跟踪和恰当地处理该异常条件。但是，这时已经向堆栈的所有已保留地址空间提交了物理存储器，并且没有更多的带有已打开的保护属性标志的页面。如果用户执行另一个总数的计算，Sum函数就会使堆栈溢出，但是却不会引发堆栈溢

China-pub.com 下载

出异常条件。相反，一个访问违规异常将会发生，而这时恰当地处理这个异常条件就太晚了。

使用独立的线程的最后一个原因是，该堆栈的物理存储器可以释放。请看下面这个例子。用户要求Sum函数计算从0到30 000之间的整数的总和。这需要将相当数量的物理存储器提交给堆栈区域。然后，用户要进行若干个合计操作，其中最高的数字是 5000。在这种情况下，大量的内存被提交给堆栈区域，但是却不再被使用。该物理存储器是从页文件那里分配来的。不应该使该物理存储器保持提交状态，最好是释放该内存，重新将它交给系统和其他进程。通过使SumThreadFunc的线程终止运行，系统将自动收回已经提交给堆栈区域的物理存储器。

清单16-1 Summation示例应用程序

Summation.cpp

```
/******************************************************************************
Module:  Summation.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <limits.h>
#include <process.h>       // For _beginthreadex
#include <tchar.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

// An example of calling Sum for uNum = 0 through 9
// uNum: 0 1 2 3  4  5  6  7  8  9 ...
// Sum:  0 1 3 6 10 15 21 28 36 45 ...
UINT Sum(UINT uNum) {

   // Call Sum recursively.
   return((uNum == 0) ? 0 : (uNum + Sum(uNum - 1)));
}

//////////////////////////////////////////////////////////////////////////////

LONG WINAPI FilterFunc(DWORD dwExceptionCode) {

   return((dwExceptionCode == STATUS_STACK_OVERFLOW)
      ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}

//////////////////////////////////////////////////////////////////////////////
// The separate thread that is responsible for calculating the sum.
// I use a separate thread for the following reasons:
//    1. A separate thread gets its own 1 MB of stack space.
//    2. A thread can be notified of a stack overflow only once.
//    3. The stack's storage is freed when the thread exits.
DWORD WINAPI SumThreadFunc(PVOID pvParam) {

   // The parameter pvParam, contains the number of integers to sum.
   UINT uSumNum = PtrToUlong(pvParam);
```

China-pub.com
下载


```
   // uSum contains the summation of the numbers from 0 through uSumNum.
   // If the sum cannot be calculated, a sum of UINT_MAX is returned.
   UINT uSum = UINT_MAX;

   __try {
      // To catch the stack overflow exception, we must
      // execute the Sum function while inside an SEH block.
      uSum = Sum(uSumNum);
   }
   __except (FilterFunc(GetExceptionCode())) {
      // If we get in here, it's because we have trapped a stack overflow.
      // We can now do whatever is necessary to gracefully continue execution.
      // This sample application has nothing to do, so no code is placed
      // in this exception handler block.
   }

   // The thread's exit code is the sum of the first uSumNum
   // numbers, or UINT_MAX if a stack overflow occurred.
   return(uSum);
}

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SUMMATION);

   // Don't accept integers more than 9 digits long.
   Edit_LimitText(GetDlgItem(hwnd, IDC_SUMNUM), 9);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_CALC:
         // Get the number of integers the user wants to sum.
         UINT uSum = GetDlgItemInt(hwnd, IDC_SUMNUM, NULL, FALSE);
        // Create a thread (with its own stack) that is
        // responsible for performing the summation.
        DWORD dwThreadId;
        HANDLE hThread = chBEGINTHREADEX(NULL, 0,
           SumThreadFunc, (PVOID) (UINT_PTR) uSum, 0, &dwThreadId);

        // Wait for the thread to terminate.
        WaitForSingleObject(hThread, INFINITE);

        // The thread's exit code is the resulting summation.
        GetExitCodeThread(hThread, (PDWORD) &uSum);
```

China-pub.com 下载

```
            // Allow the system to destroy the thread kernel object
            CloseHandle(hThread);

            // Update the dialog box to show the result.
            if (uSum == UINT_MAX) {
               // If result is UINT_MAX, a stack overflow occurred.
               SetDlgItemText(hwnd, IDC_ANSWER, TEXT("Error"));
               chMB("The number is too big, please enter a smaller number");
            } else {
               // The sum was calculated successfully;
               SetDlgItemInt(hwnd, IDC_ANSWER, uSum, FALSE);
            }
            break;
      }
   }

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SUMMATION), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

Summation.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
```

China-pub.com
下载


```
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SUMMATION           ICON    DISCARDABLE     "Summation.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SUMMATION DIALOG DISCARDABLE  18, 18, 162, 41
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Summation"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Calculate the sum of the numbers from 0 through &x, where x is: ",
                    IDC_STATIC,4,4,112,20
    EDITTEXT        IDC_SUMNUM,120,8,40,13,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "&Calculate",IDC_CALC,4,28,56,12
    LTEXT           "Answer:",IDC_STATIC,68,30,30,8
    LTEXT           "?",IDC_ANSWER,104,30,56,8
END

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
```

China-pub.com 下载

```
    "#include ""windows.h""\r\n"
    "#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

第17章　内存映射文件

对文件进行操作几乎是所有应用程序都必须进行的，并且这常常是人们争论的一个问题。应用程序究竟是应该打开文件，读取文件并关闭文件，还是打开文件，然后使用一种缓冲算法，从文件的各个不同部分进行读取和写入呢？Microsoft提供了一种两全其美的方法，那就是内存映射文件。

与虚拟内存一样，内存映射文件可以用来保留一个地址空间的区域，并将物理存储器提交给该区域。它们之间的差别是，物理存储器来自一个已经位于磁盘上的文件，而不是系统的页文件。一旦该文件被映射，就可以访问它，就像整个文件已经加载内存一样。

内存映射文件可以用于3个不同的目的：

- 系统使用内存映射文件，以便加载和执行.exe和DLL文件。这可以大大节省页文件空间和应用程序启动运行所需的时间。
- 可以使用内存映射文件来访问磁盘上的数据文件。这使你可以不必对文件执行I/O操作，并且可以不必对文件内容进行缓存。
- 可以使用内存映射文件，使同一台计算机上运行的多个进程能够相互之间共享数据。Windows确实提供了其他一些方法，以便在进程之间进行数据通信，但是这些方法都是使用内存映射文件来实现的，这使得内存映射文件成为单个计算机上的多个进程互相进行通信的最有效的方法。

本章将要介绍内存映射文件的各种使用方法。

17.1　内存映射的可执行文件和DLL文件

当线程调用CreateProcess时，系统将执行下列操作步骤：

1) 系统找出在调用CreateProcess时设定的.exe文件。如果找不到这个.exe文件，进程将无法创建，CreateProcess将返回FALSE。

2) 系统创建一个新进程内核对象。

3) 系统为这个新进程创建一个私有地址空间。

4) 系统保留一个足够大的地址空间区域，用于存放该.exe文件。该区域需要的位置在.exe文件本身中设定。按照默认设置，.exe文件的基地址是0x00400000（这个地址可能不同于在64位Windows 2000上运行的64位应用程序的地址），但是，可以在创建应用程序的.exe文件时重载这个地址，方法是在链接应用程序时使用链接程序的/BASE选项。

5) 系统注意到支持已保留区域的物理存储器是在磁盘上的.exe文件中，而不是在系统的页文件中。

当.exe文件被映射到进程的地址空间中之后，系统将访问.exe文件的一个部分，该部分列出了包含.exe文件中的代码要调用的函数的DLL文件。然后，系统为每个DLL文件调用LoadLibrary函数，如果任何一个DLL需要更多的DLL，那么系统将调用LoadLibrary函数，以便加载这些DLL。每当调用LoadLibrary来加载一个DLL时，系统将执行下列操作步骤，它们均类似上面的第4和第5个步骤：

1) 系统保留一个足够大的地址空间区域，用于存放该DLL文件。该区域需要的位置在DLL文件本身中设定。按照默认设置，Microsoft的Visual C++建立的DLL文件基地址是

China-pub.com 下载

0x10000000（这个地址可能不同于在64位Windows 2000上运行的64位DLL的地址）但是，你可以在创建DLL文件时重载这个地址，方法是使用链接程序的/BASE选项。Windows提供的所有标准系统DLL都拥有不同的基地址，这样，如果加载到单个地址空间，它们就不会重叠。

2) 如果系统无法在该DLL的首选基地址上保留一个区域，其原因可能是该区域已经被另一个DLL或.exe占用，也可能是因为该区域不够大，此时系统将设法寻找另一个地址空间的区域来保留该DLL。如果一个DLL无法加载到它的首选基地址，这将是非常不利的，原因有二。首先，如果系统没有再定位信息，它就无法加载该DLL（可以在DLL创建时，使用链接程序的/FIXED开关，从DLL中删除再定位信息，这能够使DLL变得比较小，但是这也意味着该DLL必须加载到它的首选地址中，否则它就根本无法加载）。第二，系统必须在DLL中执行某些再定位操作。在Windows 98中，系统可以在页面被转入RAM时执行再定位操作。在Windows 2000中，这些再定位操作需要由系统的页文件提供更多的存储器，它们也增加了加载DLL所需要的时间量。

3) 系统会注意到支持已保留区域的物理存储器位于磁盘上的DLL文件中，而不是在系统的页文件中。如果由DLL无法加载到它的首选基地址，Windows 2000必须执行再定位操作，那么系统也将注意到DLL的某些物理存储器已经被映射到页文件中。

如果由于某个原因系统无法映射.exe和所有必要的DLL文件，那么系统就会向用户显示一个消息框，并且释放进程的地址空间和进程对象。CreateProcess函数将向调用者返回FALSE，调用者可以调用GetLastError函数，以便更好地了解为什么无法创建该进程。

当所有的.exe和DLL文件都被映射到进程的地址空间之后，系统就可以开始执行.exe文件的启动代码。当.exe文件被映射后，系统将负责所有的分页、缓冲和高速缓存的处理。例如，如果.exe文件中的代码使它跳到一个尚未加载到内存的指令地址，那么就会出现一个错误。系统能够发现这个错误，并且自动将这页代码从该文件的映像加载到一个RAM页面。然后，系统将这个RAM页面映射到进程的地址空间中的相应位置，并且让线程继续运行，就像这页代码已经加载了一样。当然，这一切是应用程序看不见的。当进程中的线程每次试图访问尚未加载到RAM的代码或数据时，该进程就会重复执行。

17.1.1 可执行文件或DLL的多个实例不能共享静态数据

当为正在运行的应用程序创建新进程时，系统将打开用于标识可执行文件映像的文件映射对象的另一个内存映射视图，并创建一个新进程对象和（为主线程创建）一个新线程对象。系统还要将新的进程ID和线程ID赋予这些对象。通过使用内存映射文件，同一个应用程序的多个正在运行的实例就能够共享RAM中的相同代码和数据。

这里有一个小问题需要注意。进程使用的是一个平面地址空间。当编译和链接你的程序时，所有的代码和数据都被合并在一起，组成一个很大的结构。数据与代码被分开，但仅限于跟在.exe文件中的代码后面的数据而已[⊖]。图17-1简单说明了应用程序的代码和数据究竟是如何加载到虚拟内存中，然后又被映射到应用程序的地址空间中的。

作为一个例子，假设应用程序的第二个实例正在运行。系统只是将包含文件的代码和数据的虚拟内存页面映射到第二个应用程序的地址空间，如图17-2所示。

如果应用程序的一个实例改变了驻留在数据页面中的某些全局变量，那么该应用程序的所有实例的内存内容都将改变。这种类型的改变可能带来灾难性的后果，因此是决不允许的。

⊖ 实际上，文件的内容被分割为不同的节。代码放在一个节中，全局变量放在另一个节中。各个节按照页面边界来对齐。通过调用Get SystemInfo函数，应用程序可以确定正在使用的页面的大小。在.exe或DLL文件中，代码节通常位于数据数据节的前面。

China-pub.com
下载

系统运用内存管理系统的copy-on-write（写入时拷贝）特性来防止进行这种改变。每当应用程序尝试将数据写入它的内存映射文件时，系统就会抓住这种尝试，为包含应用程序尝试写入数据的内存页面分配一个新内存块，再拷贝该页面的内容，并允许该应用程序将数据写入这个新分配的内存块。结果，同一个应用程序的所有其他实例的运行都不会受到影响。图 17-3显示了当应用程序的第一个实例尝试改变数据页面 2时出现的情况。



图17-1 应用程序的代码和数据加载及映射示意图



图17-2 应用程序与虚拟内存地址空间之间的关系示意图



图17-3 应用程序的第一个实例尝试改变数据页面 2时的情况

系统分配一个新的虚拟内存页面，并且将数据页面2的内容拷贝到新页面中。第一个实例的地址空间发生了变更，这样，新数据页面就被映射到与原始地址页面相同位置上的地址空间中。这时系统就可以让进程修改全局变量，而不必担心改变同一个应用程序的另一个实例的数据。

当应用程序被调试时，将会发生类似的事件。比如说，你正在运行一个应用程序的多个实例，并且只想调试其中的一个实例。你访问调试程序，在一行源代码中设置一个断点。调试程序修改了你的代码，将你的一个汇编语言指令改为能使调试程序自行激活的指令。因此你再次

China-pub.com 下载

遇到了同样的问题。当调试程序修改代码时，它将导致应用程序的所有实例在修改后的汇编语言指令运行时激活该调试程序。为了解决这个问题，系统再次使用 copy-on-write内存。当系统发现调试程序试图修改代码时，它就分配一个新内存块，将包含该指令的页面拷贝到新的内存页面中，并且允许调试程序修改页面拷贝中的代码。

Windows 98 当一个进程被加载时，系统要查看文件映像的所有页面。系统立即为通常用copy-on-write属性保护的那些页面提交页文件中的存储器。这些页面只是被提交而已，它们并不被访问。当文件映像中的页面被访问时，系统就加载相应的页面。如果该页面从来没有被修改，它就可以从内存中删除，并在必要时重新加载。但是，如果文件的页面被修改了，系统就将修改过的页面转到页文件中以前被提交的页面之一。

Windows 2000与Windows 98之间的行为特性的唯一差别，是在你加载一个模块的两个拷贝并且可写入的数据尚未被修改的时候显示出来的。在这种情况下，在Windows 2000下运行的进程能够共享数据，而在Windows 98下，每个进程都可以得到它自己的数据拷贝。如果只加载模块的一个拷贝，或者可写入的数据已经被修改（这是通常的情况），那么Windows 2000与Windows 98的行为特性是完全相同的。

17.1.2 在可执行文件或DLL的多个实例之间共享静态数据

全局数据和静态数据不能被同一个 .exe或DLL文件的多个映像共享，这是个安全的默认设置。但是，在某些情况下，让一个 .exe文件的多个映像共享一个变量的实例是非常有用和方便的。例如，Windows没有提供任何简便的方法来确定用户是否在运行应用程序的多个实例。但是，如果能够让所有实例共享单个全局变量，那么这个全局变量就能够反映正在运行的实例的数量。当用户启动应用程序的一个实例时，新实例的线程能够简单地查看全局变量的值（它已经被另一个实例更新）；如果这个数量大于 1，那么第二个实例就能够通知用户，该应用程序只有一个实例可以运行，而第二个实例将终止运行。

本节将介绍一种方法，它允许你共享 .exe或DLL文件的所有实例的变量。不过在介绍这个方法之前，首先让我们介绍一些背景知识。

每个.exe或DLL文件的映像都由许多节组成。按照规定，每个标准节的名字均以圆点开头。例如，当编译你的程序时，编译器会将所有代码放入一个名叫 .text的节中。该编译器还将所有未经初始化的数据放入一个 .bss节，而已经初始化的所有数据则放入 .data节中。

每一节都拥有与其相关的一组属性，这些属性如表 17-1所示。

表17-1 .exe或DLL文件各节的属性

属 性	含 义
READ	该节中的字节可以读取
WRITE	该节中的字节可以写入
EXECUTE	该节中的字节可以执行
SHARED	该节中的字节可以被多个实例共享（本属性能够有效地关闭 copy-on-write机制）

使用Microsoft的Visual Studio的DumpBin实用程序（带有 /Headers开关），可以查看 .exe或DLL映射文件中各个节的列表。下面选录的代码是在一个可执行文件上运行 DumpBin程序而生成的：

```
SECTION HEADER #1
   .text name
```

China-pub.com
下载


```
   11A70 virtual size
    1000 virtual address
   12000 size of raw data
    1000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

SECTION HEADER #2
  .rdata name
     1F6 virtual size
   13000 virtual address
    1000 size of raw data
   13000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only
 SECTION HEADER #3
    .data name
      560 virtual size
   14000 virtual address
    1000 size of raw data
   14000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

 SECTION HEADER #4
  .idata name
     58D virtual size
   15000 virtual address
    1000 size of raw data
   15000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #5
  .didat name
```

China-pub.com 下载

```
     7A2 virtual size
   16000 virtual address
    1000 size of raw data
   16000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write
SECTION HEADER #6
  .reloc name
     26D virtual size
   17000 virtual address
    1000 size of raw data
   17000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

  Summary
        1000 .data
        1000 .didat
        1000 .idata
        1000 .rdata
        1000 .reloc
       12000 .text
```

表17-2显示了比较常见的一些节的名字，并且说明了每一节的作用。

除了编译器和链接程序创建的标准节外，也可以在使用下面的命令进行编译时创建自己的节：

表17-2 常见的节名及作用

节 名	作 用
.bss	未经初始化的数据
.CRT	C运行期只读数据
.data	已经初始化的数据
.debug	调试信息
.didata	延迟输入文件名表
.edata	输出文件名表
.idata	输入文件名表
.rdata	运行期只读数据
.reloc	重定位表信息
.rsrc	资源
.text	.exe或DLL文件的代码
.tls	线程的本地存储器
.xdata	异常处理表

China-pub.com
下载


```
#pragma data_seg("sectionname")
```

我可以创建一个称为“ Shared ”的节，它包含单个LONG值，如下所示：

```
#pragma data_seg("Shared")
LONG g_lInstanceCount = 0;
#pragma data_seg()
```

当编译器对这个代码进行编译时，它创建一个新节，称为 Shared，并将它在编译指示后面看到的所有已经初始化（initialized）的数据变量放入这个新节中。在上面这个例子中，变量放入Shared节中。该变量后面的#pragma dataseg()一行告诉编译器停止将已经初始化的变量放入Shared节，并且开始将它们放回到默认数据节中。需要记住的是，编译器只将已经初始化的变量放入新节中。例如，如果我从前面的代码段中删除初始化变量（如下面的代码所示），那么编译器将把该变量放入Shared节以外的节中。

```
#pragma data_seg("Shared")
LONG g_lInstanceCount;
#pragma data_seg()
```

Microsoft 的Visual C++编译器提供了一个Allocate说明符，使你可以将未经初始化的数据放入你希望的任何节中。请看下面的代码：

```
// Create Shared section & have compiler place initialized data in it.
#pragma data_seg("Shared")

// Initialized, in Shared section
int a = 0;

// Uninitialized, not in Shared section
int b;

// Have compiler stop placing initialized data in Shared section.
#pragma data_seg()

// Initialized, in Shared section
__declspec(allocate("Shared")) int c = 0;
// Uninitialized, in Shared section
__declspec(allocate("Shared")) int d;

// Initialized, not in Shared section
int e = 0;

// Uninitialized, not in Shared section
int f;
```

上面的注释清楚地指明了指定的变量将被放入哪一节。若要使 Allocate声明的规则正确地起作用，那么首先必须创建节。如果删除前面这个代码中的第一行 #pragma data_seg，上面的代码将不进行编译。

之所以将变量放入它们自己的节中，最常见的原因也许是要在 .exe或DLL文件的多个映像之间共享这些变量。按照默认设置，.exe或DLL文件的每个映像都有它自己的一组变量。然而，可以将你想在该模块的所有映像之间共享的任何变量组合到它自己的节中去。当给变量分组时，系统并不为.exe或DLL文件的每个映像创建新实例。

仅仅告诉编译器将某些变量放入它们自己的节中，是不足以实现对这些变量的共享的。还

China-pub.com 下载

必须告诉链接程序，某个节中的变量是需要加以共享的。若要进行这项操作，可以使用链接程序的命令行上的/SECTION开关：

```
/SECTION:name,attributes
```

在冒号的后面，放入你想要改变其属性的节的名字。在我们的例子中，我们想要改变Shared节的属性。因此应该创建下面的链接程序开关：

```
/SECTION:Shared,RWS
```

在逗号的后面，我们设定了需要的属性。用 R代表READ，W代表WEITE，E代表EXECUTE，S代表SHARED。上面的开关用于指明位于Shared节中的数据是可以读取、写入和共享的数据。如果想要改变多个节的属性，必须多次设定 /SECTION开关，也就是为你要改变属性的每个节设定一个/SECTION开关。

也可以使用下面的句法将链接程序开关嵌入你的源代码中：

```
#pragma comment(linker, "/SECTION:Shared,RWS")
```

这一行代码告诉编译器将上面的字符串嵌入名字为“ .drectve ”的节。当链接程序将所有的.obj模块组合在一起时，链接程序就要查看每个 .obj模块的“ .drectve ”节，并且规定所有的字符串均作为命令行参数传递给该链接程序。我一直使用这种方法，因为它非常方便。如果将源代码文件移植到一个新项目中，不必记住在Visual C++的Project Settings（项目设置）对话框中设置链接程序开关。

虽然可以创建共享节，但是，由于两个原因， Microsoft并不鼓励你使用共享节。第一，用这种方法共享内存有可能破坏系统的安全。第二，共享变量意味着一个应用程序中的错误可能影响另一个应用程序的运行，因为它没有办法防止某个应用程序将数据随机写入一个数据块。

假设你编写了两个应用程序，每个应用程序都要求用户输入一个口令。然而你又决定给应用程序添加一些特性，使用户操作起来更加方便些：如果在第二个应用程序启动运行时，用户正在运行其中的一个应用程序，那么第二个应用程序就可以查看共享内存的内容，以便获得用户的口令。这样，如果程序中的某一个已经被使用，那么用户就不必重新输入他的口令。

这听起来没有什么问题。毕竟没有别的应用程序而只有你自己的应用程序加载了 DLL，并且知道到什么地方去查找包含在共享节中的口令。但是，黑客正在窥视着你的行动，如果他们想要得到你的口令，只需要编写一段很短的程序，加载到你的公司的 DLL文件中，然后监控共享内存块。当用户输入口令时，黑客的程序就能知道该用户的口令。

黑客精心编制的程序也可能试图反复猜测用户的口令并将它们写入共享内存。一旦该程序猜测到正确的口令，它就能够将各种命令发送给两个应用程序中的一个。如果有一种办法只为某些应用程序赋予访问权，以便加载一个特定的 DLL，那么这个问题也许是可以解决的。但是目前还不行，因为任何程序都能够调用LoadLibrary函数来显式加载DLL。

17.1.3 AppInst示例应用程序

清单17-1列出的AppInst示例应用程序（“ 17 AppInst.exe ”)显示了应用程序如何能够知道每次有多少个应用程序的实例正在运行。该应用程序的源代码和资源文件位于本书所附光盘上的17-AppInst目录下。当运行AppInst程序时，就会出现它的对话框（见图17-4），指明该应用程序的一个实例正在运行。



图17-4 运行AppInst 时出现的对话框

如果运行该应用程序的第二个实例，那么第一和第二个实例的对话框都会发生变化，以反映目前两个实例都在运行（见图 17-5，图17-6)。

China-pub.com
下载



图17-5 运行 AppInst 的第二个实例时，第一个实例对话框的变化



图17-6 运行 AppInst 的第二个实例时，第二个实例对话框的变化

可以根据你的喜好，运行和撤消任意多个实例，实例的数量始终都能正确地反映在仍然保留的实例中。

在靠近AppInst.cpp应用程序的顶部，可以看到下面的代码行：

```
// Tell the compiler to put this initialized variable in its own
// section so it is shared by all instances of this application.
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// Tell the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/Section:Shared,RWS")
```

这些代码行用于创建一个称为Shared的节，该节拥有读取、写入和共享保护属性。在这个节中，有一个变量是g_lApplicationInstances。该应用程序的所有实例均可以共享该变量。注意，该变量是个易失性变量，因此优化程序对我们不起多大的作用。

当每个实例的_tWinMain函数执行时，g_lApplicationInstances变量就递增1。在_tWinMain退出之前，该变量将递减1。我使用InterlockedExchangeAdd来改变这个变量，因为多个线程将要访问该共享资源。

当每个实例的对话框出现时，Dlg_OnInitDialog函数就被调用。该函数将一个注册窗口消息广播发送到所有的高层窗口（该消息的ID包含在g_aMsgAppInstCountUpdate变量中）：

```
PostMessage(HWND_BROADCAST, g_aMsgAppInstCountUpdate, 0, 0);
```

系统中的所有窗口将忽略这个注册窗口消息，但 AppInst的各个窗口例外。当我们的各个窗口中的一个接收到该消息时，Dlg_Proc中的代码将更新该对话框中的实例数量，以反映当前的实例数量（该数量在g_lApplicationInstances共享变量中进行维护）。

清单17-1 AppInst示例应用程序



AppInst.cpp

```
/******************************************************************************
Module:  AppInst.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
// The system-wide unique window message
UINT g_uMsgAppInstCountUpdate = INVALID_ATOM;

//////////////////////////////////////////////////////////////////////////////

// Tell the compiler to put this initialized variable in its own Shared
// section so it is shared by all instances of this application.
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// Tell the linker to make the Shared section readable, writable, and shared.
#pragma comment(linker, "/Section:Shared,RWS")

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_APPINST);

   // Force the static control to be initialized correctly.
   PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
   return(TRUE);
}
//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   if (uMsg == g_uMsgAppInstCountUpdate) {
      SetDlgItemInt(hwnd, IDC_COUNT, g_lApplicationInstances, FALSE);
   }

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

   // Get the numeric value of the systemwide window message used to notify
   // all top-level windows when the module's usage count has changed.
   g_uMsgAppInstCountUpdate =
      RegisterWindowMessage(TEXT("MsgAppInstCountUpdate"));

   // There is another instance of this application running.
   InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, 1);

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_APPINST), NULL, Dlg_Proc);
   // This instance of the application is terminating.
   InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, -1);

   // Have all other instances update their display.
   PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);

   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

AppInst.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_APPINST DIALOG DISCARDABLE  0, 0, 140, 21
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Application Instances"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Number of instances running:",IDC_STATIC,12,4,93,8,
                    SS_NOPREFIX
    RTEXT           "#",IDC_COUNT,112,4,16,12,SS_NOPREFIX
END


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPINST             ICON    DISCARDABLE     "AppInst.Ico"
/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
```

China-pub.com
下载


```
    IDD_APPINST, DIALOG
    BEGIN
        RIGHTMARGIN, 76
        BOTTOMMARGIN, 20
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

17.2 内存映射数据文件

操作系统使得内存能够将一个数据文件映射到进程的地址空间中。因此，对大量的数据进行操作是非常方便的。

为了理解用这种方法来使用内存映射文件的功能，让我们看一看如何用 4种方法来实现一个程序，以便将文件中的所有字节的顺序进行倒序。

17.2.1 方法1：一个文件，一个缓存

第一种方法也是理论上最简单的方法，它需要分配足够大的内存块来存放整个文件。该文件被打开，它的内容被读入内存块，然后该文件被关闭。文件内容进入内存后，我们就可以对所有字节的顺序进行倒序，方法是将第一个字节倒腾为最后一个字节，第二个字节倒腾为倒数第二个字节，依次类推。这个倒腾操作将一直进行下去直到文件的中间位置。当所有的字节都已经倒腾之后，就可以重新打开该文件，并用内存块的内容来改写它的内容。

这种方法实现起来非常容易，但是它有两个缺点。首先，必须分配一个与文件大小相同的内存块。如果文件比较小，那么这没有什么问题。但是如果文件非常大，比如说有 2GB大，那该怎么办呢？一个32位的系统不允许应用程序提交那么大的物理内存块。因此大文件需要使用不同的方法。

第二，如果进程在运行过程的中间被中断，也就是说当倒序后的字节被重新写入该文件时进程被中断，那么文件的内容就会遭到破坏。防止出现这种情况的最简单的方法是在对它的内容进行倒序之前先制作一个原始文件的拷贝。如果整个进程运行成功，那么可以删除该文件的拷贝。这种方法需要更多的磁盘空间。

17.2.2 方法2：两个文件，一个缓存

在第二种方法中，你打开现有的文件，并且在磁盘上创建一个长度为 0的新文件。然后分

China-pub.com 下载

配一个比较小的内部缓存，比如说8 KB。你找到离原始文件结尾还有8 KB的位置，将这最后的8 KB读入缓存，将字节倒序，再将缓存中的内容写入新创建的文件。这个寻找、读入、倒序和写入的操作过程要反复进行，直到到达原始文件的开头。如果文件的长度不是 8 KB的倍数，那么必须进行某些特殊的处理。当原始文件完全处理完毕之后，将原始文件和新文件关闭，并删除原始文件。

这种方法实现起来比第一种方法要复杂一些。它对内存的使用效率要高得多，因为它只需要分配一个8 KB的缓存块，但是它存在两个大问题。首先，它的处理速度比第一种方法要慢，原因是在每个循环操作过程中，在执行读入操作之前，必须对原始文件进行寻找操作。第二，这种方法可能要使用大量的硬盘空间。如果原始文件是 400 MB，那么随着进程的不断运行，新文件就会增大为400 MB。在原始文件被删除之前，两个文件总共需要占用800 MB的磁盘空间。这比应该需要的空间大400 MB。由于存在这个缺点，因此引来了下一个方法。

17.2.3 方法3：一个文件，两个缓存

如果使用这个方法，那么我们假设程序初始化时分配了两个独立的8 KB缓存。程序将文件的第一个8 KB读入一个缓存，再将文件的第二个8 KB 读入另一个缓存。然后进程将两个缓存的内容进行倒序，并将第一个缓存的内容写回文件的结尾处，将第二个缓存的内容写回同一个文件的开始处。每个迭代操作不断进行（以8 KB为单位，从文件的开始和结尾处移动文件块）。如果文件的长度不是16 KB的倍数，并且有两个8 KB的文件块相重叠，那么就需要进行一些特殊的处理。这种特殊处理比上一种方法中的特殊处理更加复杂，不过这难不倒经验丰富的编程员。

与前面的两种方法相比，这种方法在节省硬盘空间方面有它的优点。由于所有内容都是从同一个文件读取并写入同一个文件，因此不需要增加额外的磁盘空间，至于内存的使用，这种方法也不错，它只需要使用16 KB的内存。当然，这种方法也许是最难实现的方法。与第一种方法一样，如果进程被中断，本方法会导致数据文件被破坏。

下面让我们来看一看如何使用内存映射文件来完成这个过程。

17.2.4 方法4：一个文件，零缓存

当使用内存映射文件对文件内容进行倒序时，你打开该文件，然后告诉系统将虚拟地址空间的一个区域进行倒序。你告诉系统将文件的第一个字节映射到该保留区域的第一个字节。然后可以访问该虚拟内存的区域，就像它包含了这个文件一样。实际上，如果在文件的结尾处有一个单个0字节，那么只需要调用C运行期函数_strrev，就可以对文件中的数据进行倒序操作。

这种方法的最大优点是，系统能够为你管理所有的文件缓存操作。不必分配任何内存，或者将文件数据加载到内存，也不必将数据重新写入该文件，或者释放任何内存块。但是，内存映射文件仍然可能出现因为电源故障之类的进程中断而造成数据被破坏的问题。

17.3 使用内存映射文件

若要使用内存映射文件，必须执行下列操作步骤：

1) 创建或打开一个文件内核对象，该对象用于标识磁盘上你想用作内存映射文件的文件。
2) 创建一个文件映射内核对象，告诉系统该文件的大小和你打算如何访问该文件。
3) 让系统将文件映射对象的全部或一部分映射到你的进程地址空间中。

当完成对内存映射文件的使用时，必须执行下面这些步骤将它清除：

China-pub.com
下载

1) 告诉系统从你的进程的地址空间中撤消文件映射内核对象的映像。
2) 关闭文件映射内核对象。
3) 关闭文件内核对象。
下面将详细介绍这些操作步骤。

17.3.1 步骤1：创建或打开文件内核对象

若要创建或打开一个文件内核对象，总是要调用CreateFile函数：

```
HANDLE CreateFile(
   PCSTR pszFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwCreationDisposition,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile);
```

CreateFile函数拥有好几个参数。这里只重点介绍前3个参数，即pszFileName，dwDesiredAccess和dwShareMode。

你可能会猜到，第一个参数pszFileName用于指明要创建或打开的文件的名字（包括一个选项路径）。第二个参数dwDesiredAccess用于设定如何访问该文件的内容。可以设定表17-3所列的4个值中的一个。

表17-3 dwDesiredAccess的值

值	含 义
0	不能读取或写入文件的内容。当只想获得文件的属性时，请设定0
GENERIC_READ	可以从文件中读取数据
GENERIC_WRITE	可以将数据写入文件
GENERIC_READ |GENERIC_WRITE	可以从文件中读取数据，也可以将数据写入文件

当创建或打开一个文件，将它作为一个内存映射文件来使用时，请选定最有意义的一个或多个访问标志，以说明你打算如何访问文件的数据。对内存映射文件来说，必须打开用于只读访问或读写访问的文件，因此，可以分别设定 GENERIC_READ或GENERIC_READ | GENERIC_WRITE。

第三个参数dwShareMode告诉系统你想如何共享该文件。可以为dwShareMode设定表17-4所列的4个值之一。

表17-4 dwShareMode 的值

值	含 义
0	打开文件的任何尝试均将失败
FILE_SHARE_READ	使用GENERIC_WRITE打开文件的其他尝试将会失败
FILE_SHARE_WRITE	使用GENERIC_READ打开文件的其他尝试将会失败
FILE_SHARE_READ FILE_SHARE_WRITE|	打开文件的其他尝试将会取得成功

如果CreateFile函数成功地创建或打开指定的文件，便返回一个文件内核对象的句柄，否则返回INVALID_HANDLE_VALUE。

China-pub.com 下载

注意 能够返回句柄的大多数Windows函数如果运行失败，那么就会返回NULL。但是，CreateFile函数将返回INVALID_HANDLE_VALUE，它定义为（(HANDLE) -1)。

17.3.2 步骤2：创建一个文件映射内核对象

调用CreateFile函数，就可以将文件映像的物理存储器的位置告诉操作系统。你传递的路径名用于指明支持文件映像的物理存储器在磁盘（或网络或光盘）上的确切位置。这时，必须告诉系统，文件映射对象需要多少物理存储器。若要进行这项操作，可以调用 CreateFileMapping函数：

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD fdwProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);
```

第一个参数hFile用于标识你想要映射到进程地址空间中的文件句柄。该句柄由前面调用的CreateFile函数返回。psa参数是指向文件映射内核对象的SECURITY_ATTRIBUTES结构的指针，通常传递的值是NULL（它提供默认的安全特性，返回的句柄是不能继承的）。

本章开头讲过，创建内存映射文件就像保留一个地址空间区域然后将物理存储器提交给该区域一样。因为内存映射文件的物理存储器来自磁盘上的一个文件，而不是来自从系统的页文件中分配的空间。当创建一个文件映射对象时，系统并不为它保留地址空间区域，也不将文件的存储器映射到该区域（下一节将介绍如何进行这项操作）。但是，当系统将存储器映射到进程的地址空间中去时，系统必须知道应该将什么保护属性赋予物理存储器的页面。CreateFileMapping函数的fdwProtect参数使你能够设定这些保护属性。大多数情况下，可以设定表17-5中列出的3个保护属性之一。

表17-5 使用fdwProtect 参数设定的部分保护属性

保护属性	含义
PAGE_READONLY	当文件映射对象被映射时，可以读取文件的数据。必须已经将GENERIC_READ传递给CreateFile函数
PAGE_READWRITE	当文件映射对象被映射时，可以读取和写入文件的数据。必须已经将GENERIC_READ | GENERIC_WRITE传递给CreateFile
PAGE_WRITECOPY	当文件映射对象被映射时，可以读取和写入文件的数据。如果写入数据，会导致页面的私有拷贝得以创建。必须已经将GENERIC_READ或GENERIC_WRITE传递给CreateFile

Windows 98 在Windows 98下，可以将PAGE_WRITECOPY标志传递给CreateFileMapping，这将告诉系统从页文件中提交存储器。该页文件存储器是为数据文件的数据拷贝保留的，只有修改过的页面才被写入页文件。你对该文件的数据所作的任何修改都不会重新填入原始数据文件。其最终结果是，PAGE_WRITECOPY标志的作用在Windows 2000和Windows 98上是相同的。

除了上面的页面保护属性外，还有 4个节保护属性，你可以用 OR将它们连接起来放入CreateFileMapping函数的fdwProtect参数中。节只是用于内存映射的另一个术语。

China-pub.com
下载

节的第一个保护属性是SEC_NOCACHE，它告诉系统，没有将文件的任何内存映射页面放入高速缓存。因此，当将数据写入该文件时，系统将更加经常地更新磁盘上的文件数据。这个标志与PAGE_NOCACHE保护属性标志一样，是供设备驱动程序开发人员使用的，应用程序通常不使用。

Windows 98 Windows 98将忽略SEC_NOCACHE标志。

节的第二个保护属性是SEC_IMAGE，它告诉系统，你映射的文件是个可移植的可执行(PE)文件映像。当系统将该文件映射到你的进程的地址空间中时，系统要查看文件的内容，以确定将哪些保护属性赋予文件映像的各个页面。例如， PE文件的代码节(.text)通常用PAGE_EXECUTE_READ属性进行映射，而PE文件的数据节(.data)则通常用PAGE_READWRITE属性进行映射。如果设定的属性是SEC_IMAGE，则告诉系统进行文件映像的映射，并设置相应的页面保护属性。

Windows 98 Windows 98将忽略SEC_IMAGE标志。

最后两个保护属性是SEC_RESERVE和SEC_COMMIT，它们是两个互斥属性，当使用内存映射数据文件时，它们不能使用。这两个标志将在本章后面介绍。当创建内存映射数据文件时，不应该设定这些标志中的任何一个标志。CreateFileMapping将忽略这些标志。

CreateFileMapping的另外两个参数是dwMaximumSizeHigh和dwMaximumSizeLow，它们是两个最重要的参数。CreateFileMapping函数的主要作用是保证文件映射对象能够得到足够的物理存储器。这两个参数将告诉系统该文件的最大字节数。它需要两个32位的值，因为Windows支持的文件大小可以用64位的值来表示。dwMaximumSizeHigh参数用于设定较高的32位，而dwMaximumSizeLow参数则用于设定较低的32位值。对于4 GB或小于4 GB的文件来说，dwMaximumSizeHigh的值将始终是0。

使用64位的值，意味着Windows能够处理最大为16EB（10^{18}字节）的文件。如果想要创建一个文件映射对象，使它能够反映文件当前的大小，那么可以为上面两个参数传递0。如果只打算读取该文件或者访问文件而不改变它的大小，那么为这两个参数传递0。如果打算将数据附加给该文件，可以选择最大的文件大小，以便为你留出一些富裕的空间。如果当前磁盘上的文件包含0字节，那么可以给CreateFileMapping函数的dwMaximumSizeHigh和dwMaximumSizeLow传递两个0。这样做就可以告诉系统，你要的文件映射对象里面的存储器为0字节。这是个错误，CreateFileMapping将返回NULL。

如果你对我们讲述的内容一直非常关注，你一定认为这里存在严重的问题。Windows支持最大为16EB的文件和文件映射对象，这当然很好，但是，怎样将这样大的文件映射到32位进程的地址空间（32位地址空间是4GB文件的上限）中去呢？下一节介绍解决这个问题的办法。当然，64位进程拥有16 EB的地址空间，因此可以进行更大的文件的映射操作，但是，如果文件是个超大规模的文件，仍然会遇到类似的问题。

若要真正理解CreateFile和CreateFileMapping两个函数是如何运行的，建议你做一个下面的实验。建立下面的代码，对它进行编译，然后在一个调试程序中运行它。当你一步步执行每个语句时，你会跳到一个命令解释程序，并执行C：\目录上的“dir”命令。当执行调试程序中的每个语句时，请注意目录中出现的变化。

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE,
   PTSTR pszCmdLine, int nCmdShow) {

   // Before executing the line below, C:\ does not have
```

China-pub.com 下载

```
   // a file called "MMFTest.Dat."
   HANDLE hfile = CreateFile("C:\\MMFTest.dat",
      GENERIC_READ | GENERIC_WRITE,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL, NULL);

   // Before executing the line below, the MMFTest.Dat
   // file does exist but has a file size of 0 bytes.
   HANDLE hfilemap = CreateFileMapping(hfile, NULL, PAGE_READWRITE,
      0, 100, NULL);

   // After executing the line above, the MMFTest.Dat
   // file has a size of 100 bytes.

   // Cleanup
   CloseHandle(hfilemap);
   CloseHandle(hfile);

   // When the process terminates, MMFTest.Dat remains
   // on the disk with a size of 100 bytes.
   return(0);
}
```

如果调用CreateFileMapping函数，传递PAGE_READWRITE标志，那么系统将设法确保磁盘上的相关数据文件的大小至少与dwMaximumSizeHigh和dwMaximumSizeLow参数中设定的大小相同。如果该文件小于设定的大小，CreateFileMapping函数将扩展该文件的大小，使磁盘上的文件变大。这种扩展是必要的，这样，当以后将该文件作为内存映射文件使用时，物理存储器就已经存在了。如果正在用PAGE_READONLY或PAGE_WRITECOPY标志创建该文件映射对象，那么CreateFileMapping特定的文件大小不得大于磁盘文件的物理大小。这是因为你无法将任何数据附加给该文件。

CreateFileMapping函数的最后一个参数是pszName。它是个以0结尾的字符串，用于给该文件映射对象赋予一个名字。该名字用于与其他进程共享文件映射对象（本章后面展示了它的一个例子。第3章详细介绍了内核对象的共享操作）。内存映射数据文件通常并不需要被共享，因此这个参数通常是NULL。

系统创建文件映射对象，并将用于标识该对象的句柄返回该调用线程。如果系统无法创建文件映射对象，便返回一个NULL句柄值。记住，当CreateFile运行失败时，它将返回INVALID_HANDLE_VALUE（定义为-1），当CreateFileMapping运行失败时，它返回NULL。请不要混淆这些错误值。

17.3.3 步骤3：将文件数据映射到进程的地址空间

当创建了一个文件映射对象后，仍然必须让系统为文件的数据保留一个地址空间区域，并将文件的数据作为映射到该区域的物理存储器进行提交。可以通过调用MapViewOfFile函数来进行这项操作：

```
PVOID MapViewOfFile(
   HANDLE hFileMappingObject,
   DWORD dwDesiredAccess,
   DWORD dwFileOffsetHigh,
   DWORD dwFileOffsetLow,
   SIZE_T dwNumberOfBytesToMap);
```

China-pub.com
下载

参数hFileMappingObject用于标识文件映射对象的句柄，该句柄是前面调用CreateFile Mapping或OpenFileMapping（本章后面介绍）函数返回的。参数dwDesiredAccess用于标识如何访问该数据。不错，必须再次设定如何访问文件的数据。可以设定表17-6所列的4个值中的一个。

表17-6 值及其含义

值	含 义
FILE_MAP_WRITE	可以读取和写入文件数据。CreateFileMapping函数必须通过传递PAGE_READWRITE标志来调用
FILE_MAP_READ	可以读取文件数据。CreateFileMapping函数可以通过传递下列任何一个保护属性来调用：PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY
FILE_MAP_ALL_ACCESS	与FILE_MAP_WRITE相同
FILE_MAP_COPY	可以读取和写入文件数据。如果写入文件数据，可以创建一个页面的私有拷贝。在Windows 2000中，CreateFileMapping函数可以用PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY等保护属性中的任何一个来调用。在Windows 98中，CreateFileMapping必须用PAGE_WRITECOPY来调用

Windows要求所有这些保护属性一次又一次地重复设置，这当然有些奇怪和烦人。我认为这样做可以使应用程序更多地对数据保护属性进行控制。

剩下的3个参数与保留地址空间区域及将物理存储器映射到该区域有关。当你将一个文件映射到你的进程的地址空间中时，你不必一次性地映射整个文件。相反，可以只将文件的一小部分映射到地址空间。被映射到进程的地址空间的这部分文件称为一个视图，这可以说明MapViewOfFile是如何而得名的。

当将一个文件视图映射到进程的地址空间中时，必须规定两件事情。首先，必须告诉系统，数据文件中的哪个字节应该作为视图中的第一个字节来映射。你可以使用dwFileOffsetHigh和dwFileOffsetLow参数来进行这项操作。由于Windows支持的文件最大可达16EB，因此必须用一个64位的值来设定这个字节的位移值。这个64位值中，较高的32位传递给参数dwFileOffsetHigh，较低的32位传递给参数dwFileOffsetLow。注意，文件中的这个位移值必须是系统的分配粒度的倍数（迄今为止，Windows的所有实现代码的分配粒度均为64 KB）。第14章介绍了如何获取某个系统的分配粒度。

第二，必须告诉系统,数据文件有多少字节要映射到地址空间。这与设定要保留多大的地址空间区域的情况是相同的。可以使用dwNumberOfBytesToMap参数来设定这个值。如果设定的值是0，那么系统将设法把从文件中的指定位移开始到整个文件的结尾的视图映射到地址空间。

Windows 98 在Windows 98中，如果MapViewOfFile无法找到足够大的区域来存放整个文件映射对象，那么无论需要的视图是多大，MapViewOfFile均将返回NULL。

Windows 2000 在Windows 2000中，MapViewOfFile只需要为必要的视图找到足够大的一个区域，而不管整个文件映射对象是多大。

如果在调用MapViewOfFile函数时设定了FILE_MAP_COPY标志，系统就会从系统的页文件中提交物理存储器。提交的地址空间数量由dwNumberOfBytesToMap参数决定。只要你不进行其他操作，只是从文件的映像视图中读取数据，那么系统将决不会使用页文件中的这些提交的页面。但是，如果进程中的任何线程将数据写入文件的映像视图中的任何内存地址，那么系统将从页文件中抓取已提交页面中的一个页面，将原始数据页面拷贝到该页交换文件中，然后

China-pub.com 下载

将该拷贝的页面映射到你的进程的地址空间。从这时起，你的进程中的线程就要访问数据的本地拷贝，不能读取或修改原始数据。

当系统制作原始页面的拷贝时，系统将把页面的保护属性从 PAGE_WRITECOPY改为PAGE_READWRITE。下面这个代码段就说明了这个情况：

```
// Open the file that we want to map.
HANDLE hFile = CreateFile(pszFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
   OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// Create a file-mapping object for the file.
HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_WRITECOPY,
   0, 0, NULL);

// Map a copy-on-write view of the file; the system will commit
// enough physical storage from the paging file to accommodate
// the entire file. All pages in the view will initially have
// PAGE_WRITECOPY access.
PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_COPY,
   0, 0, 0);

// Read a byte from the mapped view.
BYTE bSomeByte = pbFile[0];
// When reading, the system does not touch the committed pages in
// the paging file. The page keeps its PAGE_WRITECOPY attribute.

// Write a byte to the mapped view.
pbFile[0] = 0;
// When writing for the first time, the system grabs a committed
// page from the paging file, copies the original contents of the
// page at the accessed memory address, and maps the new page
// (the copy) into the process's address space. The new page has
// an attribute of PAGE_READWRITE.

// Write another byte to the mapped view.
pbFile[1] = 0;
// Because this byte is now in a PAGE_READWRITE page, the system
// simply writes the byte to the page (backed by the paging file).

// When finished using the file's mapped view, unmap it.
// UnmapViewOfFile is discussed in the next section.
UnmapViewOfFile(pbFile);
// The system decommits the physical storage from the paging file.
// Any writes to the pages are lost.

// Clean up after ourselves.
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Windows 98 前面讲过，Windows 98必须预先为内存映射文件提交页文件中的存储器。然而，它只有在必要时才将修改后的页面写入页文件。

17.3.4 步骤4：从进程的地址空间中撤消文件数据的映像

当不再需要保留映射到你的进程地址空间区域中的文件数据时，可以通过调用下面的函数

China-pub.com
下载

将它释放：

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

该函数的唯一的参数 pvBaseAddress用于设定返回区域的基地址。该值必须与调用MapViewOfFile函数返回的值相同。必须记住要调用 UnmapViewOfFile函数。如果没有调用这个函数，那么在你的进程终止运行前，保留的区域就不会被释放。每当你调用 MapViewOfFile时，系统总是在你的进程地址空间中保留一个新区域，而以前保留的所有区域将不被释放。

为了提高速度，系统将文件的数据页面进行高速缓存，并且在对文件的映射视图进行操作时不立即更新文件的磁盘映像。如果需要确保你的更新被写入磁盘，可以强制系统将修改过的数据的一部分或全部重新写入磁盘映像中，方法是调用 FlushViewOfFile函数：

```
BOOL FlushViewOfFile(
   PVOID pvAddress,
   SIZE_T dwNumberOfBytesToFlush);
```

第一个参数是包含在内存映射文件中的视图的一个字节的地址。该函数将你在这里传递的地址圆整为一个页面边界值。第二个参数用于指明你想要刷新的字节数。系统将把这个数字向上圆整，使得字节总数是页面的整数。如果你调用 FlushViewOfFile函数并且不修改任何数据，那么该函数只是返回，而不将任何信息写入磁盘。

对于存储器是在网络上的内存映射文件来说， FlushViewOfFile能够保证文件的数据已经从工作站写入存储器。但是FlushViewOfFile不能保证正在共享文件的服务器已经将数据写入远程磁盘，因为服务器也许对文件的数据进行了高速缓存。若要保证服务器写入文件的数据，每当你为文件创建一个文件映射对象并且映射该文件映射对象的视图时，应该将 FILE_FLAG_WRITE_THROUGH标志传递给CreateFile函数。如果你使用该标志打开该文件，那么只有当文件的全部数据已经存放在服务器的磁盘驱动器中的时候， FlushViewOfFile函数才返回。

记住UnmapViewOfFile函数的一个特殊的特性。如果原先使用 FILE_MAP_COPY标志来映射视图，那么你对文件的数据所作的任何修改，实际上是对存放在系统的页文件中的文件数据的拷贝所作的修改。在这种情况下，如果调用 UnmapViewOfFile函数，该函数在磁盘文件上就没有什么可以更新，而只会释放页文件中的页面，从而导致数据丢失。

如果想保留修改后的数据，必须采用别的措施。例如，你可以用同一个文件创建另一个文件映射对象（使用PAGE_READWRITE），然后使用FILE_MAP_WRITE标志将这个新文件映射对象映射到进程的地址空间。之后，你可以扫描第一个视图，寻找带有 PAGE_READWRITE保护属性的页面。每当你找到一个带有该属性的页面时，可以查看它的内容，并且确定是否将修改了的数据写入该文件。如果不想用新数据更新该文件，那么继续对视图中的剩余页面进行扫描，直到视图的结尾。但是，如果你确实想要保存修改了的数据页面，那么只需要调用MoveMemory函数，将数据页面从第一个视图拷贝到第二个视图。由于第二个视图是用PAGE_READWRITE保护属性映射的，因此 MoveMemory函数将更新磁盘上的实际文件内容。可以使用这种方法来确定文件的变更并保存你的文件的数据。

> Windows 98 Windows 98不支持copy-on-write（写入时拷贝）保护属性，因此，当扫描内存映射文件的第一个视图时，无法测试用 PAGE_READWRITE标志做上标记的页面。你必须设计一种方法来确定第一个视图中的哪些页面已经做了修改。

17.3.5 步骤5和步骤6：关闭文件映射对象和文件对象

不用说，你总是要关闭你打开了的内核对象。如果忘记关闭，在你的进程继续运行时会出

China-pub.com 下载

现资源泄漏的问题。当然，当你的进程终止运行时，系统会自动关闭你的进程已经打开但是忘记关闭的任何对象。但是如果你的进程暂时没有终止运行，你将会积累许多资源句柄。因此你始终都应该编写清楚而又“正确的”代码，以便关闭你已经打开的任何对象。若要关闭文件映射对象和文件对象，只需要两次调用CloseHandle函数，每个句柄调用一次：

让我们更加仔细地观察一下这个进程。下面的伪代码显示了一个内存映射文件的例子：

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);

// Use the memory-mapped file.

UnmapViewOfFile(pvFile);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

上面的代码显示了对内存映射文件进行操作所用的“预期”方法。但是，它没有显示，当你调用MapViewOfFile时系统对文件对象和文件映射对象的使用计数的递增情况。这个副作用是很大的，因为它意味着我们可以将上面的代码段重新编写成下面的样子：

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
CloseHandle(hFile);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);
CloseHandle(hFileMapping);

// Use the memory-mapped file.

UnmapViewOfFile(pvFile);
```

当对内存映射文件进行操作时，通常要打开文件，创建文件映射对象，然后使用文件映射对象将文件的数据视图映射到进程的地址空间。由于系统递增了文件对象和文件映射对象的内部使用计数，因此可以在你的代码开始运行时关闭这些对象，以消除资源泄漏的可能性。

如果用同一个文件来创建更多的文件映射对象，或者映射同一个文件映射对象的多个视图，那么就不能较早地调用CloseHandle函数——以后你可能还需要使用它们的句柄，以便分别对CreateFileMapping和MapViewOfFile函数进行更多的调用。

17.3.6 文件倒序示例应用程序

清单17-2列出的FileRev应用程序（“17 FileRev.exe”）显示了如何使用内存映射对象来对ANSI或Unicode文本文件的内容进行倒序。该应用程序的源代码和资源文件位于本书所附光盘上的17-FileRev目录下。当启动该程序时，会出现图17-7所示的窗口。

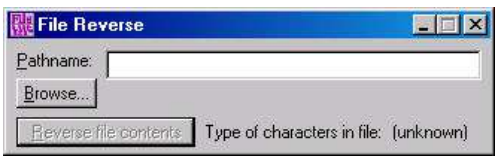


图17-7 运行FileRev 时出现的窗口

FileRev应用程序首先允许选定一个文件，然后，当单击 Reverse File Contents（对文件内容进行倒序）按钮时，该函数就会将文件中使用的字符进行倒序。该程序只能对文本文件进行正确的倒序，对二进制文件不能正确地进行倒序操作。 FileRev能够确定文本文件是ANSI文件还是Unicode文件，方法是调用IsTextUnicode函数（第2章中介绍）。

Windows 98 在Windows 98中，IsTextUnicode函数没有可以使用的实现代码，它只是返

China-pub.com
下载

回FALSE。如果调用GetLastError函数，则返回ERROR_CALL_NOT_ IMPLEMENTED。这意味着FileRev示例应用程序总是认为，当它在 Windows 98下运行时，它操作的是ANSI文本文件。

当单击Reverse File Contents按钮时，FileRev便制作指定文件的一个拷贝，称为FileRev.dat。它制作该拷贝的目的是，原始文件不会因为内容被倒序而变得无法使用。接着， FileRev调用FileReverse函数，该函数负责对文件进行倒序操作。 FileReverse则调用CreateFile函数，打开FileRev.dat,以便进行读取和写入。

前面说过，对文件内容进行倒序的最容易的方法是调用C运行期函数_strrev。与所有C字符串一样，字符串的最后一个字符必须是个 0结束符。由于文本文件不以 0为结束符，因此FileRev必须给文件附加一个0。若要进行这样的附加操作，首先要调用GetFileSize函数：

```
dwFileSize = GetFileSize(hFile, NULL);
```

现在已经得到了文件的长度，你能够通过调用 CreateFileMapping函数创建文件映射对象。创建的文件映射对象的长度是dwFileSize加一个宽字符的大小（对于0字符来说）。当文件映射对象创建后，该对象的视图就被映射到 FileRev的地址空间。变量pvFile包含了MapViewOfFile函数的返回值，并指向文本文件的第一个字节。

下一步是在文件的结尾处写一个0，并对字符串进行倒序：

```
PSTR pchANSI = (PSTR) pvFile;
pchANSI[dwFileSize / sizeof(CHAR)] = 0;
```

在文本文件中，每一行的结尾都是一个回车符（‘ \r ’）后随一个换行符（‘ \n ’）。但是，当调用_strrev对文件进行倒序时，这些字符也会被倒序。如果将已经倒序的文本文件加载到文本编辑器，那么出现的每一对“ \n\r ”字符都必须重新改为它的原始顺序。这个倒序操作是由下面的循环代码进行的：

```
while (pchANSI != NULL) {
   // We have found an occurrence....
   *pchANSI++ = '\r';   // Change '\n' to '\r'.
   *pchANSI++ = '\n';   // Change '\r' to '\n'.
   pchANSI = strchr(pchANSI, '\n'); // Find the next occurrence.
}
```

当你观察这样一个简单的代码时，可能很容易忘记你实际上是在对磁盘驱动器上的文件内容进行操作（这显示出内存映射文件的功能是多么大）。

在文件被倒序后， FileRev便进行清除操作，撤消文件映射对象的视图映象，关闭所有的内核对象句柄。此外，FileRev必须删除附加给文件结尾处的0字符（记住_strrev并不对结尾的0字符进行倒序）。如果没有删除 0字符，那么倒序的文件将会多出一个字符，如果再次调用FileRev函数，将无法使文件还原成它的原始样子。若要删除文件结尾处的 0字符，必须后退一步，使用文件管理函数，而不是通过内存映射对文件进行操作。

如果要强制已经倒序的文件在某个位置上结束，就需要将文件指针定位在指定的位置（原始文件的结尾处）并调用SetEndOfFile函数：

```
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
```

注意 SetEndOfFile函数必须在撤消视图的映象并且关闭文件映射对象之后调用，否则，该函数将返回FALSE，GetLastError则返回ERROR_USER_MAPPED_FILE。这个错误表示不能在与文件映射对象相关联的文件上执行文件末尾的操作。

China-pub.com 下载

FileRev函数做的最后一件事情是产生一个Notepad实例，这样，就可以查看已经倒序的文件。图17-8显示了在FileRev.cpp文件上运行FileRev函数时产生的结果。

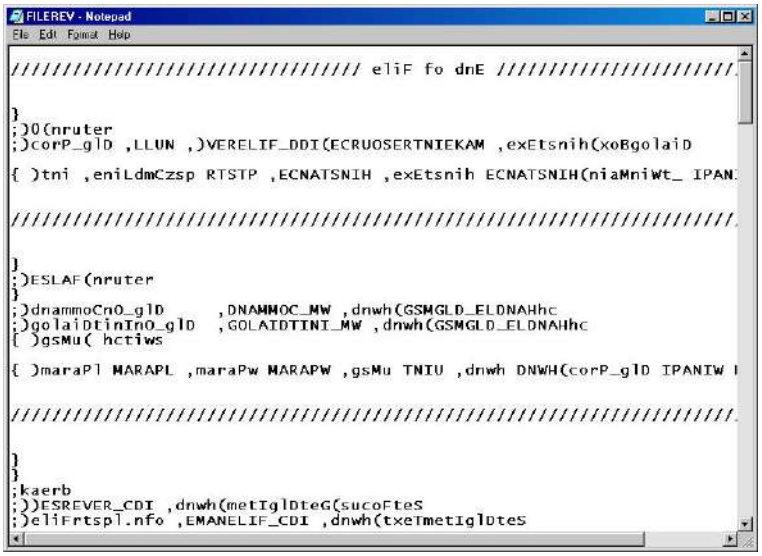


图17-8 FileRev 函数运行结果

清单17-2 FileRev示例应用程序

FileRev.cpp

```
/******************************************************************************
Module:  FileRev.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <commdlg.h>
#include <string.h>        // For _strrev
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


#define FILENAME  TEXT("FILEREV.DAT")


//////////////////////////////////////////////////////////////////////////////
BOOL FileReverse(PCTSTR pszPathname, PBOOL pfIsTextUnicode) {

   *pfIsTextUnicode = FALSE;  // Assume text is Unicode.
```

China-pub.com
下载


```
// Open the file for reading and writing.
HANDLE hFile = CreateFile(pszPathname, GENERIC_WRITE | GENERIC_READ, 0,
   NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

if (hFile == INVALID_HANDLE_VALUE) {
   chMB("File could not be opened.");
   return(FALSE);
}

// Get the size of the file (I assume the whole file can be mapped).
DWORD dwFileSize = GetFileSize(hFile, NULL);

// Create the file-mapping object. The file-mapping object is 1 character
// bigger than the file size so that a zero character can be placed at the
// end of the file to terminate the string (file). Because I don't yet know
// if the file contains ANSI or Unicode characters, I assume worst case
// and add the size of a WCHAR instead of CHAR.
HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
   0, dwFileSize + sizeof(WCHAR), NULL);

if (hFileMap == NULL) {
   chMB("File map could not be opened.");
   CloseHandle(hFile);
   return(FALSE);
}

// Get the address where the first byte of the file is mapped into memory.
PVOID pvFile = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

if (pvFile == NULL) {
   chMB("Could not map view of file.");
   CloseHandle(hFileMap);
   CloseHandle(hFile);
   return(FALSE);
}

// Does the buffer contain ANSI or Unicode?
int iUnicodeTestFlags = -1;   // Try all tests.
*pfIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);
if (!*pfIsTextUnicode) {
   // For all the file manipulations below, we explicitly use ANSI
   // functions because we are processing an ANSI file.

   // Put a zero character at the very end of the file.
   PSTR pchANSI = (PSTR) pvFile;
   pchANSI[dwFileSize / sizeof(CHAR)] = 0;

   // Reverse the contents of the file.
   _strrev(pchANSI);

   // Convert all "\n\r" combinations back to "\r\n" to
   // preserve the normal end-of-line sequence.
   pchANSI = strchr(pchANSI, '\n'); // Find first '\n'.

   while (pchANSI != NULL) {
      // We have found an occurrence....
```

China-pub.com 下载

```
            *pchANSI++ = '\r';   // Change '\n' to '\r'.
            *pchANSI++ = '\n';   // Change '\r' to '\n'.
            pchANSI = strchr(pchANSI, '\n'); // Find the next occurrence.
         }

      } else {
         // For all the file manipulations below, we explicitly use Unicode
         // functions because we are processing a Unicode file.

         // Put a zero character at the very end of the file.
         PWSTR pchUnicode = (PWSTR) pvFile;
         pchUnicode[dwFileSize / sizeof(WCHAR)] = 0;

         if ((iUnicodeTestFlags & IS_TEXT_UNICODE_SIGNATURE) != 0) {
            // If the first character is the Unicode BOM (byte-order-mark),
            // 0xFEFF, keep this character at the beginning of the file.
            pchUnicode++;
         }

         // Reverse the contents of the file.
         _wcsrev(pchUnicode);

         // Convert all "\n\r" combinations back to "\r\n" to
         // preserve the normal end-of-line sequence.
         pchUnicode = wcschr(pchUnicode, L'\n'); // Find first '\n'.

         while (pchUnicode != NULL) {
            // We have found an occurrence....
            *pchUnicode++ = L'\r';   // Change '\n' to '\r'.
            *pchUnicode++ = L'\n';   // Change '\r' to '\n'.
            pchUnicode = wcschr(pchUnicode, L'\n'); // Find the next occurrence.
         }
      }

      // Clean up everything before exiting.
      UnmapViewOfFile(pvFile);
      CloseHandle(hFileMap);

      // Remove trailing zero character added earlier.
      SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
      SetEndOfFile(hFile);
      CloseHandle(hFile);

      return(TRUE);
   }

   ////////////////////////////////////////////////////////////////////////////


   BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

      chSETDLGICONS(hwnd, IDI_FILEREV);

      // Initialize the dialog box by disabling the Reverse button.
      EnableWindow(GetDlgItem(hwnd, IDC_REVERSE), FALSE);
      return(TRUE);
   }
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   TCHAR szPathname[MAX_PATH];

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_FILENAME:
         EnableWindow(GetDlgItem(hwnd, IDC_REVERSE),
            Edit_GetTextLength(hwndCtl) > 0);
         break;
         case IDC_REVERSE:
            GetDlgItemText(hwnd, IDC_FILENAME, szPathname, chDIMOF(szPathname));

            // Make a copy of input file so that we don't destroy it.
            if (!CopyFile(szPathname, FILENAME, FALSE)) {
               chMB("New file could not be created.");
               break;
            }

            BOOL fIsTextUnicode;
            if (FileReverse(FILENAME, &fIsTextUnicode)) {
               SetDlgItemText(hwnd, IDC_TEXTTYPE,
                  fIsTextUnicode ? TEXT("Unicode") : TEXT("ANSI"));

               // Spawn Notepad to see the fruits of our labors.
               STARTUPINFO si = { sizeof(si) };
               PROCESS_INFORMATION pi;
               TCHAR sz[] = TEXT("Notepad ") FILENAME;
               if (CreateProcess(NULL, sz,
                  NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {

                  CloseHandle(pi.hThread);
                  CloseHandle(pi.hProcess);
               }
            }
            break;

         case IDC_FILESELECT:
            OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
            ofn.hwndOwner = hwnd;
            ofn.lpstrFile = szPathname;
            ofn.lpstrFile[0] = 0;
            ofn.nMaxFile = chDIMOF(szPathname);
            ofn.lpstrTitle = TEXT("Select file for reversing");
            ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
            GetOpenFileName(&ofn);
            SetDlgItemText(hwnd, IDC_FILENAME, ofn.lpstrFile);
            SetFocus(GetDlgItem(hwnd, IDC_REVERSE));
            break;
   }
```

China-pub.com 下载

```
}

//////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG,  Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,     Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_FILEREV), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

FileRev.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
```

China-pub.com
下载


```
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_FILEREV             ICON    DISCARDABLE     "FileRev.ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//
IDD_FILEREV DIALOG DISCARDABLE  15, 24, 216, 46
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "File Reverse"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "&Pathname:",IDC_STATIC,4,4,35,8
    EDITTEXT        IDC_FILENAME,44,4,168,12,ES_AUTOHSCROLL
    PUSHBUTTON      "&Browse...",IDC_FILESELECT,4,16,36,12,WS_GROUP
    DEFPUSHBUTTON   "&Reverse file contents",IDC_REVERSE,4,32,80,12
    LTEXT           "Type of characters in file:",IDC_STATIC,88,34,80,8
    LTEXT           "(unknown)",IDC_TEXTTYPE,172,34,34,8
END


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//
```

China-pub.com 下载

```
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_FILEREV, DIALOG
    BEGIN
        RIGHTMARGIN, 192
        BOTTOMMARGIN, 42
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

17.4 使用内存映射文件来处理大文件

上一节讲过我要告诉你如何将一个 16 EB的文件映射到一个较小的地址空间中。当然，你是无法做到这一点的。你必须映射一个只包含一小部分文件数据的文件视图。首先映射一个文件的开头的视图。当完成对文件的第一个视图的访问时，可以取消它的映像，然后映射一个从文件中的一个更深的位移开始的新视图。必须重复这一操作，直到访问了整个文件。这使得大型内存映射文件的处理不太方便，但是，幸好大多数文件都比较小，因此不会出现这个问题。

让我们看一个例子，它使用一个8 GB的文件和一个32位的地址空间。下面是一个例程，它使用若干个步骤来计算一个二进制数据文件中的所有 0字节的数目：

```
__int64 Count0s(void) {

   // Views must always start on a multiple
   // of the allocation granularity
   SYSTEM_INFO sinf;
   GetSystemInfo(&sinf);

   // Open the data file.
   HANDLE hFile = CreateFile("C:\\HugeFile.Big", GENERIC_READ,
      FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);

   // Create the file-mapping object.
   HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
      PAGE_READONLY, 0, 0, NULL);
```

China-pub.com
下载


```
   DWORD dwFileSizeHigh;
   __int64 qwFileSize = GetFileSize(hFile, &dwFileSizeHigh);
   qwFileSize += (((__int64) dwFileSizeHigh) << 32);

   // We no longer need access to the file object's handle.
   CloseHandle(hFile);

   __int64 qwFileOffset = 0, qwNumOfOs = 0;

   while (qwFileSize > 0) {

      // Determine the number of bytes to be mapped in this view
      DWORD dwBytesInBlock = sinf.dwAllocationGranularity;
      if (qwFileSize < sinf.dwAllocationGranularity)
         dwBytesInBlock = (DWORD) qwFileSize;
       PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_READ,
          (DWORD) (qwFileOffset >> 32),          // Starting byte
          (DWORD) (qwFileOffset & 0xFFFFFFFF),   // in file
          dwBytesInBlock);                       // # of bytes to map

       // Count the number of Js in this block.
       for (DWORD dwByte = 0; dwByte < dwBytesInBlock; dwByte++) {
          if (pbFile[dwByte] == 0)
          qwNumOfOs++;
       }

       // Unmap the view; we don't want multiple views
       // in our address space.
       UnmapViewOfFile(pbFile);

       // Skip to the next set of bytes in the file.
       qwFileOffset += dwBytesInBlock;
       qwFileSize -= dwBytesInBlock;
    }

    CloseHandle(hFileMapping);
    return(qwNumOfOs);
}
```

这个算法用于映射64 KB（分配粒度的大小）或更小的视图。另外，要记住，MapViewOfFile函数要求文件的位移是分配粒度大小的倍数。当每个视图被映射到地址空间时，对0的扫描不断进行。当每个64 KB的文件块已经映射和扫描完毕时，就要通过关闭文件映射对象来对每个文件块进行整理。

17.5 内存映射文件与数据视图的相关性

系统允许你映射一个文件的相同数据的多个视图。例如，你可以将文件开头的10 KB映射到一个视图，然后将同一个文件的头4 KB映射到另一个视图。只要你是映射相同的文件映射对象，系统就会确保映射的视图数据的相关性。例如，如果你的应用程序改变了一个视图中的文件内容，那么所有其他视图均被更新以反映这个变化。这是因为尽管页面多次被映射到进程的虚拟地址空间，但是系统只将数据放在单个RAM页面上。如果多个进程映射单个数据文件的视图，那么数据仍然是相关的，因为在数据文件中，每个RAM页面只有一个实例——正是

China-pub.com 下载

这个RAM页面被映射到多个进程的地址空间。

注意 Windows允许创建若干个由单个数据文件支持的文件映射对象。Windows不能保证这些不同的文件映射对象的视图具有相关性。它只能保证单个文件映射对象的多个视图具有相关性。

然而，当对文件进行操作时，没有理由使另一个应用程序无法调用CreateFile函数以打开由另一个进程映射的同一个文件。这个新进程可以使用ReadFile和WriteFile函数来读取该文件的数据和将数据写入该文件。当然，每当一个进程调用这些函数时，它必须从内存缓冲区读取文件数据或者将文件数据写入内存缓冲区。该内存缓冲区必须是进程自己创建的一个缓冲区，而不是映射文件使用的内存缓冲区。当两个应用程序打开同一个文件时，问题就可能产生：一个进程可以调用ReadFile函数来读取文件的一个部分，并修改它的数据，然后使用WriteFile函数将数据重新写入文件，而第二个进程的文件映射对象却不知道第一个进程执行的这些操作。由于这个原因，当你为将被内存映射的文件调用CreateFile函数时，最好将dwShareMode参数的值设置为0。这样就可以告诉系统，你想要单独访问这个文件，而其他进程都不能打开它。

只读文件不存在相关性问题，因此它们可以作为很好的内存映射文件。内存映射文件决不应该用于共享网络上的可写入文件，因为系统无法保证数据视图的相关性。如果某个人的计算机更新了文件的内容，其他内存中含有原始数据的计算机将不知道它的信息已经被修改。

17.6 设定内存映射文件的基地址

正如你可以使用VirtualAlloc函数来确定对地址空间进行倒序所用的初始地址一样，你也可以使用MapViewOfFileEx函数而不是使用MapViewOfFile函数来确定一个文件被映射到某个特定的地址。请看下面的代码：

```
PVOID MapViewOfFileEx(
   HANDLE hFileMappingObject,
   DWORD dwDesiredAccess,
   DWORD dwFileOffsetHigh,
   DWORD dwFileOffsetLow,
   SIZE_T dwNumberOfBytesToMap,
   PVOID pvBaseAddress);
```

该函数的所有参数和返回值均与MapViewOfFile函数相同，唯一的差别是最后一个参数pvBaseAddress有所不同。在这个参数中，你为要映射的文件设定一个目标地址。与VirtualAlloc一样，你设定的目标地址应该是分配粒度边界（64 KB）的倍数，否则MapViewOfFileEx将返回NULL，表示出现了错误。

在Windows 2000下，如果设定的地址不是分配粒度的倍数，就会导致函数运行失败，同时GetLastError将返回1132（ERROR_MAPPED_ALIGNMENT）。在Windows 98中，该地址将圆整为分配粒度边界值。

如果系统无法将文件映射到该位置上（通常由于文件太大并且与另一个保留的地址空间相重叠），那么该函数的运行就会失败并且返回NULL。MapViewOfFileEx并不设法寻找另一个地址空间来放置该文件。当然，你可以设定NULL作为pvBaseAddress参数的值，这时，MapViewOfFileEx函数的运行特性与MapViewOfFile函数完全相同。

当你使用内存映射文件与其他进程共享数据时，你可以使用MapViewOfFileEx函数。例如，当两个或多个应用程序需要共享包含指向其他数据结构的一组数据结构时，可能需要在某个特定地址上的内存映射文件。链接表是个极好的例子。在链接表中，每个节点或元素均包含列表

China-pub.com
下载

中的另一个元素的内存地址。若要遍历该列表，必须知道第一个元素的地址，然后参考包含下一个元素地址的元素成员。当使用内存映射文件时，这可能成为一个问题。

如果一个进程建立了内存映射文件中的链接表，然后与另一个进程共享该文件，那么另一个进程就可能将文件映射到它的地址空间中的一个完全不同的位置上。当第二个进程视图遍历该链接表时，它查看链接表的第一个元素，检索下一个元素的内存地址，然后设法引用下一个元素。然而，第一个节点中的下一个元素的地址并不是第二个进程需要查找的地址。

可以用两种办法来解决这个问题。首先，当第二个进程将包含链接表的内存映射文件映射到它自己的地址空间中去时，它只要调用 MapViewOfFileEx函数而不是调用MapViewOfFile。当然，这种方法要求第二个进程必须知道第一个进程原先在建立链接表时将文件映射到了什么地方。当两个应用程序打算互相进行交互操作时（这是非常可能的），这就不会出现任何问题，因为地址可以通过硬编码放入两个应用程序，或者一个进程可以通知另一个进程使用另一种进程间通信的方式，比如将消息发送到窗口。

第二个方法是创建链接表的进程将下一个节点所在的地址中的位移存放在每个节点中。这要求应用程序将该位移添加给内存映射文件的基地址，以便访问每个节点。这种方法并不高明，因为它的运行速度可能比较慢，它会使程序变得更大（因为编译器要生成附加代码来执行所有的计算操作），而且它很容易出错。但是，它仍然是个可行的方法， Microsoft的编译器为使用__based关键字的基本指针提供了辅助程序。

Windows 98 当调用MapViewOfFileEx时，必须设定0x80000000与0xBFFFFFFF之间的一个地址，否则MapViewOfFileEx将返回ULL。

Windows 20000 当调用MapViewOfFileEx时，必须设定在你的进程的用户方式分区中的一个地址，否则MapViewOfFileEx将返回NULL。

17.7 实现内存映射文件的具体方法

Windows 98和Windows 2000实现内存映射文件的方法是不同的。必须知道这些差别，因为它们会影响你编写代码的方法，也会影响其他应用程序对你的数据进行不利的操作。

在Windows 98下，视图总是映射到0x80000000至0xBFFFFFFF范围内的地址空间分区中。因此，对MapViewOfFile函数的成功调用都会返回这个范围内的一个地址。你也许还记得，所有进程都共享该分区中的数据。这意味着如果进程映射了文件映射对象的视图，那么该文件映射对象的数据实际上就可以被所有进程访问，而不管它们是否已经映射了该文件映射对象的视图。如果另一个进程调用使用同一个文件映射对象的MapViewOfFile函数，Windows 98便将返回给第一个进程的同一个内存地址返回给第二个进程。这两个进程访问相同的数据，并且它们的视图具有相关性。

在Windows 98中，一个进程可以调用MapViewOfFile函数 ，并且可以使用某种进程间的通信方式将返回的内存地址传递给另一个进程的线程。一旦该线程收到这个内存地址，该线程就可以成功地访问文件映射对象的同一个视图。但是，不应该这样做，原因有二。

- 你的应用程序将无法在Windows 2000下运行，其原因将在下面说明。
- 如果第一个进程调用UnmapViewOfFile函数，地址空间区域将恢复为空闲状态，这意味着第二个进程的线程如果尝试访问视图曾经位于其中的内存，会引发一次访问违规。

如果第二个进程访问内存映射对象的视图，那么第二个进程中的线程应该调用 MapViewOfFile函数。当第二个进程这样做的时候，系统将对内存映射视图的使用计数进行递增。因此，如果第一个进程调用UnmapViewOfFile函数，那么在第二个进程也调用UnmapViewOfFile之前，

China-pub.com 下载

系统将不会释放视图占用的地址空间区域。

当第二个进程调用MapViewOfFile函数时，返回的地址将与第一个进程返回的地址相同。这样，第一个进程就没有必要使用进程间的通信方式将内存地址传送给第二个进程。

Windows 2000实现内存映射文件的方法要比Windows 98好，因为Windows 2000要求在进程的地址空间中的文件数据可供访问之前，该进程必须调用MapViewOfFile函数。如果一个进程调用MapViewOfFile函数，系统将为调用进程的地址空间中的视图进行地址空间区域的倒序操作，这样，其他进程都将无法看到该视图。如果另一个进程想要访问同一个文件映射对象中的数据，那么第二个进程中的线程就必须调用MapViewOfFile，同时，系统将为第二个进程的地址空间中的视图进行地址空间区域的倒序操作。

值得注意的是，第一个进程调用MapViewOfFile函数后返回的内存地址，很可能不同于第二个进程调用MapViewOfFile函数后返回的内存地址。即使这两个进程映射了相同文件映射对象的视图，它们返回的地址也可能不同。在Windows 98下，MapViewOfFile函数返回的内存地址是相同的，但是，如果想让你的应用程序在Windows 2000下运行，那么绝对不应该指望它们也返回相同的地址。

让我们再来观察另一个实现代码的差别。下面是一个小程序，它映射了单个文件映射对象的两个视图：

```
#include <Windows.h>

int WINAPI WinMain (HINSTANCE hinstExe, HINSTANCE,
   PTSTR pszCmdLine, int nCmdShow) {

   // Open an existing file-it must be bigger than 64 KB.
   HANDLE hFile = CreateFile(pszCmdLine, GENERIC_READ | GENERIC_WRITE,
      0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   // Create a file-mapping object backed by the data file.
   HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
      PAGE_READWRITE, 0, 0, NULL);

   // Map a view of the whole file into our address space.
   PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping,
      FILE_MAP_WRITE, 0, 0, 0);

   // Map a view of the file (starting 64 KB in) into our address space
   PBYTE pbFile2 = (PBYTE) MapViewOfFile(hFileMapping,
      FILE_MAP_WRITE, 0, 65536, 0);

   if ((pbFile + 65536) == pbFile2) {
      // If the addresses overlap, there is one address
      // space region for both views: this must be Windows 98.
      MessageBox(NULL, "We are running under Windows 98", NULL, MB_OK);
   } else {
      // If the addresses do not overlap, each view has its own
      // address space region: this must be Windows 2000.
      MessageBox(NULL, "We are running under Windows 2000", NULL, MB_OK);
   }

   UnmapViewOfFile(pbFile2);
   UnmapViewOfFile(pbFile);
   CloseHandle(hFileMapping);
   CloseHandle(hFile);

   return(0);
}
```

China-pub.com
下载

在Windows 98中，当文件映射对象的视图被映射时，系统将为整个文件映射对象保留足够的地址空间。即使调用MapViewOfFile函数时它的参数指明你想要系统只映射文件映射对象的一小部分，系统也会为它保留足够的地址空间。这意味着即使你规定只映射文件映射对象的一个64 KB的部分，也不能将一个1 GB的文件映射对象映射到一个视图中。

每当进程调用MapViewOfFile时，该函数将返回一个为整个文件映射对象保留的地址空间区域中的地址。因此，在上面的代码段中，第一次调用MapViewOfFile函数时返回包含整个映射文件的区域的基地址，第二次调用MapViewOfFile函数时返回离同一个地址空间区域64 KB位置上的地址。

Windows 2000的实现代码在这里同样存在很大的差别。在上面的代码段中，两次调用MapViewOfFile函数将导致Windows 2000保留两个不同的地址空间区域。第一个区域的大小是文件映射对象的大小，第二个区域的大小是文件映射对象的大小减去 64 KB。尽管存在两个不同的区域，但是它们的数据能够保证其相关性，因为两个视图都是从相同的文件映射对象映射而来的。在Windows 98下，各个视图具有相关性，因为它们位于同一个内存中。

17.8 使用内存映射文件在进程之间共享数据

Windows总是出色地提供各种机制，使应用程序能够迅速而方便地共享数据和信息。这些机制包括RPC、COM、OLE、DDE、窗口消息（尤其是WM_COPYDATA）、剪贴板、邮箱、管道和套接字等。在Windows中，在单个计算机上共享数据的最低层机制是内存映射文件。不错，如果互相进行通信的所有进程都在同一台计算机上的话，上面提到的所有机制均使用内存映射文件从事它们的烦琐工作。如果要求达到较高的性能和较小的开销，内存映射文件是举手可得的最佳机制。

数据共享方法是通过让两个或多个进程映射同一个文件映射对象的视图来实现的，这意味着它们将共享物理存储器的同一个页面。因此，当一个进程将数据写入一个共享文件映射对象的视图时，其他进程可以立即看到它们视图中的数据变更情况。注意，如果多个进程共享单个文件映射对象，那么所有进程必须使用相同的名字来表示该文件映射对象。

让我们观察一个例子，启动一个应用程序。当一个应用程序启动时，系统调用 CreateFile函数，打开磁盘上的.exe文件。然后系统调用CreateFileMapping函数，创建一个文件映射对象。最后，系统代表新创建的进程调用 MapViewOfFileEx函数（它带有SEC_IMAGE标志），这样，.exe文件就可以映射到进程的地址空间。这里调用的是 MapViewOfFileEx，而不是MapViewOfFile，这样，文件的映像将被映射到存放在.exe文件映像中的基地址中。系统创建该进程的主线程，将该映射视图的可执行代码的第一个字节的地址放入线程的指令指针，然后CPU启动该代码的运行。

如果用户运行同一个应用程序的第二个实例，系统就认为规定的 .exe文件已经存在一个文件映射对象，因此不会创建新的文件对象或者文件映射对象。相反，系统将第二次映射该文件的一个视图，这次是在新创建的进程的地址空间环境中映射的。系统所做的工作是将相同的文件同时映射到两个地址空间。显然，这是对内存的更有效的使用，因为两个进程将共享包含正在执行的这部分代码的物理存储器的同一个页面。

与所有内核对象一样，可以使用3种方法与多个进程共享对象，这3种方法是句柄继承性、句柄命名和句柄复制。关于这3种方法的详细说明，参见第3章的内容。

17.9 页文件支持的内存映射文件

到现在为止，已经介绍了映射驻留在磁盘驱动器上的文件视图的方法。许多应用程序在运

China-pub.com 下载

行时都要创建一些数据，并且需要将数据传送给其他进程，或者与其他进程共享。如果应用程序必须在磁盘驱动器上创建数据文件，并且将数据存储在磁盘上以便对它进行共享，那么这将是非常不方便的。

Microsoft公司认识到了这一点，并且增加了一些功能，以便创建由系统的页文件支持的内存映射文件，而不是由专用硬盘文件支持的内存映射文件。这个方法与创建内存映射磁盘文件所用的方法几乎相同，不同之处是它更加方便。一方面，它不必调用 CreateFile函数，因为你不是要创建或打开一个指定的文件，你只需要像通常那样调用 CreateFileMapping函数，并且传递INVALID_HANDLE_VALUE作为hFile参数。这将告诉系统，你不是创建其物理存储器驻留在磁盘上的文件中的文件映射对象，相反，你想让系统从它的页文件中提交物理存储器。分配的存储器的数量由CreateFileMapping函数的dwMaximumSizeHigh和dwMaximumSizeLow两个参数来决定。

当创建了文件映射对象并且将它的一个视图映射到进程的地址空间之后，就可以像使用任何内存区域那样使用它。如果你想要与其他进程共享该数据，可调用 CreateFileMapping函数，并传递一个以0结尾的字符串作为 pszName参数。然后，想要访问该存储器的其他进程就可以调用CreateFileMapping或OpenFileMapping函数，并传递相同的名字。

当进程不再想要访问文件映射对象时，该进程应该调用 CloseHandle函数。当所有句柄均被关闭后，系统将从系统的页文件中收回已经提交的存储器。

注意 下面是一个非常有意思的问题，它使单纯的程序员大吃一惊。你能猜到下面这个代码段错在哪里吗？

```
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(hFile, ...);
if (hMap == NULL)
   return(GetLastError());
.
.
.
```

如果上面这个对 CreateFile函数的调用失败，它将返回 INVALID_HANDLE_VALUE。但是，编写这个代码的程序员没有测试一下，看文件是否已经创建成功。当CreateFileMapping函数被调用时，在 hFile参数中传递了 INVALID_HANDLE_VALUE，这使得系统创建一个使用来自页文件而不是来自指定的磁盘文件存储器的文件映像。使用内存映射文件的任何辅助代码都能够正确地运行。但是，当文件映射对象被撤消时，写入文件映射存储器（页文件）的全部数据将被系统撤消。这时，程序员就坐在那里绞尽脑汁，不知道问题究竟出在哪里。因此，必须始终检查 CreateFile函数的返回值，以确定是否出现了错误，因为CreateFile运行失败的原因太多了。

共享内存映射文件的示例应用程序

清单17-3列出的MMFShare应用程序（“17 MMFShare.exe”）显示了如何使用内存映射文件在两个或多个独立的进程间传送数据。该应用程序的源代码和资源文件位于本书所附光盘上的17-MMFShare目录下。

至少需要执行MMFShare程序的两个实例。每个实例创建它自己的对话框，如图 17-9所示。

若要将数据从MMFShare的一个实例传送到另一个实例，请将要传送的数据键入Data编辑框。然后单击Create Mapping Of Data（创建数据映像）按钮。当进行这项操作时，MMFShare调用CreateFileMapping函数，创建一个由系统的页文件支持的4 KB内存映射文件对象，并将该对象

China-pub.com
下载

命名为MMFSharedData。如果MMFShare发现已经存在一个带有这个名字的文件映射对象，它就显示一个消息框，告诉你它不能创建该对象。如果MMFShare成功地创建了该对象，那么它将进一步将文件的视图映射到进程的地址空间，并将数据从编辑控件拷贝到内存映射文件中。



图17-9 运行MMFShare 时出现的对话框

当数据被拷贝后，MMFShare就撤消文件的视图，使Create Mapping Of Data按钮不起作用，并激活Close Mapping Of Data（关闭数据映像）按钮。这时，命名为MMFSharedData的内存映射文件仍然位于系统中的某个位置。没有任何进程映射了包含在文件中的数据视图。

如果这时转入MMFShare的另一个实例，并且单击该实例的 Open Mapping And Get Data（打开映像并获取数据）按钮，那么 MMFShare将设法通过调用OpenFileMapping函数，寻找一个称为MMFSharedData的文件映射对象。如果无法找到带有该名字的对象， MMFShare就会显示另一个消息框，将这个情况通知你。如果 MMFShare找到了这个对象，它将把该对象的视图映射到它的进程的地址空间，将数据从内存映射文件拷贝到对话框的编辑控件中，然后撤消它的映像，关闭文件映射对象。好极了，你已经成功地将数据从一个进程传送到另一个进程。

对话框中的Close Mapping Of Data（关闭数据映像）按钮用于关闭文件映射对象，它能够释放页文件中的存储器。如果不存在任何文件映射对象，那么 MMFShare的其他实例将无法打开文件映像并从中取出数据。另外，如果一个实例已经创建了一个内存映射文件，那么其他实例均不得创建内存映射文件并改写文件中包含的数据。

清单17-3 MMFShare示例应用程序

MMFShare.cpp

```
/******************************************************************************
Module:  MMFShare.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_MMFSHARE);

   // Initialize the edit control with some test data.
   Edit_SetText(GetDlgItem(hwnd, IDC_DATA), TEXT("Some test data"));

   // Disable the Close button because the file can't
```

China-pub.com 下载

```
   // be closed if it was never created or opened.
   Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), FALSE);
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   // Handle of the open memory-mapped file
   static HANDLE s_hFileMap = NULL;
   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_CREATEFILE:
         if (codeNotify != BN_CLICKED)
            break;

         // Create a paging file-backed MMF to contain the edit control text.
         // The MMF is 4 KB at most and is named MMFSharedData.
         s_hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
            PAGE_READWRITE, 0, 4 * 1024, TEXT("MMFSharedData"));

         if (s_hFileMap != NULL) {

            if (GetLastError() == ERROR_ALREADY_EXISTS) {
               chMB("Mapping already exists - not created.");
               CloseHandle(s_hFileMap);

            } else {

               // File mapping created successfully.

               // Map a view of the file into the address space.
               PVOID pView = MapViewOfFile(s_hFileMap,
                  FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

               if (pView != NULL) {
                  // Put edit text into the MMF.
                  Edit_GetText(GetDlgItem(hwnd, IDC_DATA),
                     (LPTSTR) pView, 4 * 1024);

                  // Protect the MMF storage by unmapping it.
                  UnmapViewOfFile(pView);

                  // The user can't create another file right now.
                  Button_Enable(hwndCtl, FALSE);

                  // The user closed the file.
                  Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), TRUE);

               } else {
                  chMB("Can't map view of file.");
```

China-pub.com
下载


```
                }
             }

          } else {
             chMB("Can't create file mapping.");
          }
          break;

       case IDC_CLOSEFILE:
          if (codeNotify != BN_CLICKED)
             break;

          if (CloseHandle(s_hFileMap)) {
             // User closed the file; fix up the buttons.
             Button_Enable(GetDlgItem(hwnd, IDC_CREATEFILE), TRUE);
             Button_Enable(hwndCtl, FALSE);
          }
          break;

       case IDC_OPENFILE:
          if (codeNotify != BN_CLICKED)
             break;

          // See if a memory-mapped file named MMFSharedData already exists.
          HANDLE hFileMapT = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
             FALSE, TEXT("MMFSharedData"));

          if (hFileMapT != NULL) {
             // The MMF does exist; map it into the process's address space.
             PVOID pView = MapViewOfFile(hFileMapT,
                FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

             if (pView != NULL) {

                // Put the contents of the MMF into the edit control.
                Edit_SetText(GetDlgItem(hwnd, IDC_DATA), (LPTSTR) pView);
                UnmapViewOfFile(pView);
             } else {
                chMB("Can't map view.");
             }

             CloseHandle(hFileMapT);

          } else {
             chMB("Can't open mapping.");
          }
          break;
    }
}

//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
```

China-pub.com 下载

```
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSHARE), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

MMFShare.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_MMFSHARE DIALOG DISCARDABLE  38, 36, 186, 61
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Memory-Mapped File Sharing"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON      "&Create mapping of Data",IDC_CREATEFILE,4,4,84,14,
                    WS_GROUP
    PUSHBUTTON      "&Close mapping of Data",IDC_CLOSEFILE,96,4,84,14
```

China-pub.com
下载


```
    LTEXT           "&Data:",IDC_STATIC,4,26,18,8
    EDITTEXT        IDC_DATA,28,24,153,12
    PUSHBUTTON      "&Open mapping and get Data",IDC_OPENFILE,40,44,104,14,
                    WS_GROUP
END


//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MMFSHARE            ICON    DISCARDABLE     "MMFShare.Ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com 下载

17.10 稀疏提交的内存映射文件

在迄今为止介绍的所有内存映射文件中，我们发现系统要求为内存映射文件提交的所有存储器必须是在磁盘上的数据文件中或者是在页文件中。这意味着我们不能根据我们的喜好来有效地使用存储器。让我们回到第15章中介绍电子表格的内容上来，比如说，你想要与另一个进程共享整个电子表格。如果我们使用内存映射文件，那么必须为整个电子表格提交物理存储器：

```
CELLDATA CellData[200][256];
```

如果CELLDATA结构的大小是128字节，那么这个数组需要6 553 600（200 x 256 x 128）字节的物理存储器。第15章讲过，如果用页文件为电子表格分配物理存储器，那么这是个不小的数目了，尤其是考虑到大多数用户只是将信息放入少数的单元格中，而大多数单元格却空闲不用时，这就显得有些浪费。

显然，我们宁愿将电子表格作为一个文件映射对象来共享，而不必预先提交所有的物理存储器。CreateFileMapping函数为这种操作提供了一种方法，即可以在fdwProtect参数中设定SEC_RESERVE或SEC_COMMIT标志。

只有当创建由系统的页文件支持的文件映射对象时，这些标志才有意义。SEC_COMMIT标志能使CreateFileMapping从系统的页文件中提交存储器。如果两个标志都不设定，其结果也一样。

当调用CreateFileMapping函数并传递SEC_RESERVE标志时，系统并不从它的页文件中提交物理存储器，它只是返回文件映射对象的一个句柄。这时可以调用MapViewOfFile或MapViewOfFileEx函数，创建该文件映射对象的视图。MapViewOfFile和MapViewOfFileEx将保留一个地址空间区域，并且不提交支持该区域的任何物理存储器。对保留区域中的内存地址进行访问的任何尝试均将导致线程引发访问违规。

现在我们得到的是一个保留的地址空间区域和用于标识该区域的文件映射对象的句柄。其他进程可以使用相同的文件映射对象来映射同一个地址空间区域的视图。物理存储器仍然没有被提交给该区域。如果其他进程中的线程试图访问它们区域中的视图的内存地址，这些线程将会引发访问违规。

下面是令人感兴趣的一些事情。若要将物理存储器提交给共享区域，线程需要做的操作只是调用VirtualAlloc函数：

```
PVOID VirtualAlloc(
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD fdwAllocationType,
   DWORD fdwProtect);
```

第15章已经介绍了这个函数。调用VirtualAlloc函数将物理存储器提交给内存映射视图区域，就像是调用VirtualAlloc函数将存储器提交给开始时通过调用带有MEM_RESERVE标志的VirtualAlloc函数而保留的区域一样。而且，就像你可以提交稀疏地存在于用VirtualAlloc保留的区域中的存储器一样，你也可以提交稀疏地存在于用MapViewOfFile或MapViewOfFileEx保留的区域中的存储器。但是，当你将存储器提交给用MapViewOfFile或MapViewOfFileEx保留的区域时，已经映射了相同文件映射对象视图的所有进程这时就能够成功地访问已经提交的页面。

使用SEC_RESERVE标志和VirtualAlloc函数，就能够成功地与其他进程共享电子表格应用

China-pub.com
下载

程序的CellData数组，并且能够非常有效地使用物理存储器。

Windows 98 通常情况下，当给VirtualAlloc函数传递的内存地址位于0x00400000至0x7FFFFFFF以外时，VirtualAlloc的运行就会失败。但是，当将物理存储器提交给使用SEC_RESERVE标志创建的内存映射文件时，必须调用VirtualAlloc函数，传递一个位于0x80000000至0xBFFFFFFF之间的内存地址。Windows 98知道你正在把存储器提交给一个保留的内存映射文件，并且让这个函数调用取得成功。

注意 在Windows 2000下，无法使用VirtualFree函数从使用SEC_RESERVE标志保留的内存映射文件中释放存储器。但是，Windows 98允许在这种情况下调用VirtualFree函数来释放存储器。

NT文件系统（NTFS 5）提供了对稀疏文件的支持。这是个非常出色的新特性。使用这个新的稀疏文件特性，能够很容易地创建和使用稀疏内存映射文件，在这些稀疏内存映射文件中，存储器包含在通常的磁盘文件中，而不是在系统的页文件中。

下面是如何使用稀疏文件特性的一个例子。比如，你想要创建一个MMF文件，以便存放记录的音频数据。当用户说话时，你想要将数字音频数据写入内存缓冲区，并且让该缓冲区得到磁盘上的一个文件的支持。稀疏MMF当然是在你的代码中实现这个要求的最容易和最有效的方法。问题是你不知道用户在单击Stop（停止）按钮之前讲了多长时间。你可能需要一个足够大的文件来存放5分钟或5小时的数据，这两个时间长度的差别太大了。但是，当使用稀疏MMF时，数据文件的大小确实无关紧要。

稀疏内存映射文件示例应用程序

清单17-4列出的MMFSparse应用程序（“17 MMFSparse.exe”）显示了如何创建一个由NTFS 5支持的内存映射文件。该应用程序的源代码和资源文件位于本书所附光盘上的17-MMFSparse目录下。当启动该程序时，便会出现图17-10所示的窗口。

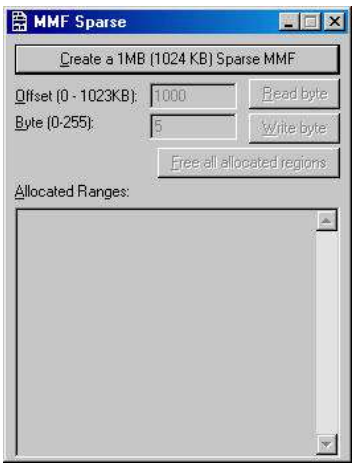


图17-10 MMF Sparse窗口

当单击Create a 1MB（1024 KB）Sparse MMF（创建一个1MB（1024 KB）稀疏MMF）按钮时，该程序将设法创建一个称为“C：\MMFSparse”的稀疏文件。如果你的C驱动器不是个NTFS 5卷，那么它的运行将会失败，并且该进程将终止运行。如果你的NTFS 5卷在另一个驱动器名上，你必须修改源代码，并且将它重建，以了解应用程序运行的情况。

一旦稀疏文件创建完成，它就被映射到进程的地址空间中。底部的Allocated Ranges（分配的范围）编辑框显示了文件的哪些部分实际上是由磁盘存储器支持的。开始时，该文件中没有任何存储器在里面，而编辑控件中则包含了“No allocated ranges in the file”(文件中没有分配范围)这一消息文本。

若要读取一个字节，只需将一个数字输入Offset（位移）编辑框中，并单击Read Byte（读取字节）按钮。输入的数字与1024（1 KB）相乘，在该位置上的字节被读取并放入Byte编辑框中。如果从没有支持存储器的任何部分中读取字节，将始终只能读取一个0字节。如果从拥有支持存储器的文件部分读取字节，将能够读取那里的任何字节。

若要写入一个字节，请将一个数字输入Offset编辑框，并且将一个字节值（0至255）输入

China-pub.com 下载

Byte编辑框。然后，当单击Write Byte（写入字节）按钮时，位移数字就与1024相乘，同时，该位置上的字节被修改以反映指定的字节值。这个写入操作可使系统为该部分文件提交支持的存储器。当读取或写入操作执行完成后， Allocated Ranges编辑框总是会得到更新，以便向你显示文件的哪些部分实际上得到存储器的支持。图17-11显示了在1 024 000（1000 x 1024）这个位移上仅仅写入一个字节后对话框是个什么样子。

注意图17-11中只存在一个分配范围，它从文件中的逻辑位移983 040字节开始，并且支持存储器的65 536个字节已经被分配。也可以使用Explorer来找出文件C：\MMFSparse并显示它的属性页，如图17-12所示。

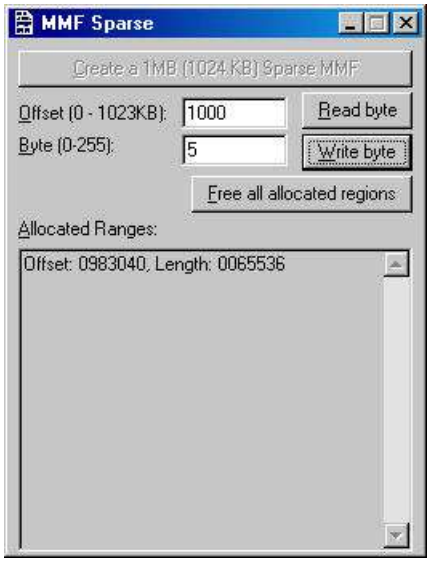


图17-11 写入一个字节后的MMF Sparse对话框

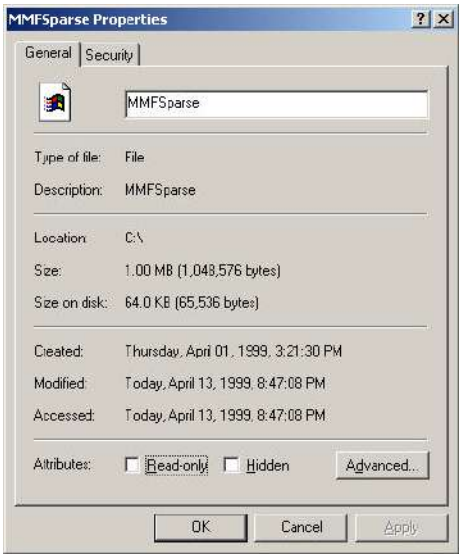


图17-12 MMF Sparse Properties 对话框

注意，该属性页显示了文件的大小是1 MB（这是文件的虚拟大小），但是该文件实际上只占用64 KB磁盘空间。

最后一个按钮是Free All Allocated Regions（释放所有分配的区域），该程序可以用它来释放用于文件的所有存储器。这个特性能够释放磁盘空间，使文件中的所有字节均显示为0。

下面让我们来介绍一下该程序是如何运行的。为了简便起见，我创建了一个CSparseStream的C++类（在SparseStream.h文件中实现）。这个类封装了可以用稀疏文件或数据流执行的任务。然后，在MMFSparse.cpp文件中，我创建了另一个C++类CMMFSparse，它是由CSparseStream派生而来的。因此，CMMFSparse对象将拥有CSparseStream的所有特性，并且要加上将稀疏数据流用作内存映射文件时特定的几个特性。该进程拥有CMMFSparse对象的单个全局实例，称为g_mmf。应用程序在它的整个代码中都要引用这个全局变量，以便对稀疏内存映射文件进行操作。

当用户单击Create a 1 MB（1024 KB）Sparse MMF（创建一个1 MB（1024 KB）的稀疏MMF）按钮时，CreateFile函数被调用，以便在NTFS 5磁盘分区上创建一个新文件。这是个普通的常规文件。但是，这时我使用g_mmf对象并且调用它的Initialize方法，传递该文件的句柄和文件的最大长度（1 MB）。在系统内部，Initialize方法调用CreateFileMapping函数，按照指定的大小创建文件映射内核对象，然后调用MapViewOfFile函数，使得该稀疏文件出现在进程的地址空间中。

China-pub.com
下载

当Initialize方法返回时，Dlg_ShowAllocatedRanges函数被调用。该函数在内部调用各个Windows函数，以便枚举已经为之分配了存储器的稀疏文件的逻辑范围。每个分配范围的起始位移和长度显示在对话框底部的编辑控件中。当g_mmf对象首次被初始化时，已经为磁盘上的文件分配的物理存储器实际上是0，编辑控件将反映出这个情况。

这时，用户可以设法从稀疏内存映射文件中读取数据，或者将数据写入该文件。如果试图写入数据，用户的位移和字节值可以从它们各自的编辑控件中获得，并将数据写入g_mmf对象中的内存地址。如果将数据写入g_mmf，文件系统就会将存储器分配给文件的这个逻辑分区，不过分配情况对应用程序来说是透明的。

如果用户试图从g_mmf对象中读取一个字节，那么该读取操作将设法在已经分配了存储器的文件中读取一个字节，否则该字节也许会标识一个尚未分配存储器的字节。如果该字节尚未被分配存储器，那么读取该字节就会返回0。同样，这对于应用程序来说是透明的。如果存在供被读取的字节使用的存储器，当然就返回它的实际值。

该应用程序说明的最后一个问题是如何清除文件，使它的所有已分配的存储器范围被释放，文件不再需要磁盘存储器。若要释放所有的已分配范围，用户可以单击Free All Allocated Ranges按钮。Windows无法为内存映射文件释放所有已分配的范围，因此应用程序要做的第一件事情是调用g_mmf对象的ForceClose方法。ForceClose方法在内部调用UnmapViewOfFile函数，然后调用CloseHandle，传递文件映射内核对象的句柄。

接着DecommitPortionOfStream函数被调用，为文件中的0至1 MB的逻辑字节释放所有的存储器。最后，再次为g_mmf对象调用Initialize方法，将内存映射文件重新初始化到进程的地址空间中。为了证明文件已经释放它的全部分配范围，需调用Dlg_ShowAllocatedRanges函数，它在编辑控件中显示“No allocated ranges in the file”（文件中没有已分配范围）字符串。

最后需要说明的是，如果在实际的应用程序中使用稀疏内存映射文件，当关闭文件时，可能必须截断文件实是的逻辑长度。虽然将包含0字节的稀疏文件的结尾截去不会对磁盘空间产生任何影响，这确实是很好的事情——Explorer和命令外壳的DIR命令能够向用户报告比较准确的文件大小。若要为文件设置文件标记的结尾，可以在调用ForceClose方法后，调用SetFilePointer和SetEndOfFile函数。

清单17-4 MMFSparse示例应用程序



MMFSparse.cpp

```
/******************************************************************************
Module:  MMFSparse.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <WindowsX.h>
#include <WinIoCtl.h>
#include "SparseStream.h"
#include "Resource.h"
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

// This class makes it easy to work with memory-mapped sparse files.
class CMMFSparse : public CSparseStream {
private:
   HANDLE m_hfilemap;      // File-mapping object
   PVOID  m_pvFile;        // Address to start of mapped file

public:
   // Creates a Sparse MMF and maps it in the process's address space.
   CMMFSparse(HANDLE hstream = NULL, SIZE_T dwStreamSizeMax = 0);

   // Closes a Sparse MMF.
   virtual ~CMMFSparse() { ForceClose(); }

   // Creates a sparse MMF and maps it in the process's address space.
   BOOL Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax);

   // MMF to BYTE cast operator returns address of first byte
   // in the memory-mapped sparse file.
   operator PBYTE() const { return((PBYTE) m_pvFile); }

   // Allows you to explicitly close the MMF without having
   // to wait for the destructor to be called.
   VOID ForceClose();
};


//////////////////////////////////////////////////////////////////////////////


CMMFSparse::CMMFSparse(HANDLE hstream, SIZE_T dwStreamSizeMax) {

   Initialize(hstream, dwStreamSizeMax);
}

//////////////////////////////////////////////////////////////////////////////


BOOL CMMFSparse::Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax) {

   if (m_hfilemap != NULL)
      ForceClose();

   // Initialize to NULL in case something goes wrong.
   m_hfilemap = m_pvFile = NULL;
   BOOL fOk = TRUE;  // Assume success.

   if (hstream != NULL) {
      if (dwStreamSizeMax == 0) {
         DebugBreak();  // Illegal stream size
      }

      CSparseStream::Initialize(hstream);
      fOk = MakeSparse();  // Make the stream sparse.
      if (fOk) {
```

China-pub.com
下载


```
         // Create a file-mapping object
         m_hfilemap = ::CreateFileMapping(hstream, NULL, PAGE_READWRITE,
            (DWORD) (dwStreamSizeMax >> 32i64), (DWORD) dwStreamSizeMax, NULL);

         if (m_hfilemap != NULL) {
            // Map the stream into the process's address space.
            m_pvFile = ::MapViewOfFile(m_hfilemap,
               FILE_MAP_WRITE | FILE_MAP_READ, 0, 0, 0);
         } else {
            // Failed to map the file; clean up
            CSparseStream::Initialize(NULL);
            ForceClose();
            fOk = FALSE;
         }
      }
   }
   return(fOk);
}

//////////////////////////////////////////////////////////////////////////////


VOID CMMFSparse::ForceClose() {

   // Clean up everything that was done successfully
   if (m_pvFile != NULL) {
      ::UnmapViewOfFile(m_pvFile);
      m_pvFile = NULL;
   }
   if (m_hfilemap != NULL) {
      ::CloseHandle(m_hfilemap);
      m_hfilemap = NULL;
   }
}
//////////////////////////////////////////////////////////////////////////////


#define STREAMSIZE      (1 * 1024 * 1024)    // 1 MB (1024 KB)
TCHAR szPathname[] = TEXT("C:\\MMFSparse.");
HANDLE g_hstream = INVALID_HANDLE_VALUE;
CMMFSparse g_mmf;


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_MMFSPARSE);

   // Initialize the dialog box controls.
   EnableWindow(GetDlgItem(hwnd, IDC_OFFSET), FALSE);
   Edit_LimitText(GetDlgItem(hwnd, IDC_OFFSET), 4);
   SetDlgItemInt(hwnd, IDC_OFFSET, 1000, FALSE);

   EnableWindow(GetDlgItem(hwnd, IDC_BYTE), FALSE);
```

China-pub.com 下载

```
   Edit_LimitText(GetDlgItem(hwnd, IDC_BYTE), 3);
   SetDlgItemInt(hwnd, IDC_BYTE, 5, FALSE);

   EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_READBYTE),  FALSE);
   EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), FALSE);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////


void Dlg_ShowAllocatedRanges(HWND hwnd) {

   // Fill in the Allocated Ranges edit control
   DWORD dwNumEntries;
   FILE_ALLOCATED_RANGE_BUFFER* pfarb =
      g_mmf.QueryAllocatedRanges(&dwNumEntries);

   if (dwNumEntries == 0) {
      SetDlgItemText(hwnd, IDC_FILESTATUS,
         TEXT("No allocated ranges in the file"));
   } else {
      TCHAR sz[4096] = { 0 };
      for (DWORD dwEntry = 0; dwEntry < dwNumEntries; dwEntry++) {
         wsprintf(_tcschr(sz, 0), TEXT("Offset: %7.7u, Length: %7.7u\r\n"),
            pfarb[dwEntry].FileOffset.LowPart, pfarb[dwEntry].Length.LowPart);
      }
      SetDlgItemText(hwnd, IDC_FILESTATUS, sz);
   }
   g_mmf.FreeAllocatedRanges(pfarb);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         if (g_hstream != INVALID_HANDLE_VALUE)
            CloseHandle(g_hstream);
         EndDialog(hwnd, id);
         break;

      case IDC_CREATEMMF:
         // Create the file
         g_hstream = CreateFile(szPathname, GENERIC_READ | GENERIC_WRITE,
            0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
         if (g_hstream == INVALID_HANDLE_VALUE) {
            chFAIL("Failed to create file.");
         }

         // Create a 1 MB (1024 KB) MMF using the file
```

China-pub.com
下载


```
            if (!g_mmf.Initialize(g_hstream, STREAMSIZE)) {
               chFAIL("Failed to initialize Sparse MMF.");
            }
            Dlg_ShowAllocatedRanges(hwnd);

            // Enable/disable the other controls.
            EnableWindow(GetDlgItem(hwnd, IDC_CREATEMMF), FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_OFFSET),    TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_BYTE),      TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_READBYTE),  TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), TRUE);
            // Force the Offset edit control to have the focus.
            SetFocus(GetDlgItem(hwnd, IDC_OFFSET));
            break;

         case IDC_WRITEBYTE:
            {
            BOOL fTranslated;
            DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
            if (fTranslated) {
               g_mmf[dwOffset * 1024] = (BYTE)
                  GetDlgItemInt(hwnd, IDC_BYTE, NULL, FALSE);
               Dlg_ShowAllocatedRanges(hwnd);
            }
            }
            break;

         case IDC_READBYTE:
            {
            BOOL fTranslated;
            DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
            if (fTranslated) {
               SetDlgItemInt(hwnd, IDC_BYTE, g_mmf[dwOffset * 1024], FALSE);
               Dlg_ShowAllocatedRanges(hwnd);
            }
            }
            break;

         case IDC_FREEALLOCATEDREGIONS:
            // Normally the destructor causes the file-mapping to close.
            // But, in this case, we wish to force it so that we can reset
            // a portion of the file back to all zeroes.
            g_mmf.ForceClose();

            // We call ForceClose above because attempting to zero a portion of
            // the file while it is mapped causes DeviceIoControl to fail with
            // error ERROR_USER_MAPPED_FILE ("The requested operation cannot
            // be performed on a file with a user-mapped section open.")
            g_mmf.DecommitPortionOfStream(0, STREAMSIZE);
            g_mmf.Initialize(g_hstream, STREAMSIZE);
            Dlg_ShowAllocatedRanges(hwnd);
            break;
      }
   }
}
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   chWindows2000Required();

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSPARSE), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

MMFSparse.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
```

China-pub.com
下载


```
// Dialog
//

IDD_MMFSHARE DIALOG DISCARDABLE  38, 36, 186, 61
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Memory-Mapped File Sharing"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON      "&Create mapping of Data",IDC_CREATEFILE,4,4,84,14,
                    WS_GROUP
    PUSHBUTTON      "&Close mapping of Data",IDC_CLOSEFILE,96,4,84,14
    LTEXT           "&Data:",IDC_STATIC,4,24,18,8
    EDITTEXT        IDC_DATA,28,24,153,12
    PUSHBUTTON      "&Open mapping and get Data",IDC_OPENFILE,40,44,104,14,
                    WS_GROUP
END

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MMFSHARE            ICON    DISCARDABLE     "MMFShare.Ico"

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

SparseStream.h

```
/******************************************************************************
Module:  SparseStream.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#include "..\CmnHdr.h"     /* See Appendix A. */
#include <WinIoCtl.h>


//////////////////////////////////////////////////////////////////////////////


#pragma once


//////////////////////////////////////////////////////////////////////////////


class CSparseStream {
public:
   static BOOL DoesFileSystemSupportSparseStreams(PCTSTR pszVolume);
   static BOOL DoesFileContainAnySparseStreams(PCTSTR pszPathname);

public:
   CSparseStream(HANDLE hstream = INVALID_HANDLE_VALUE) {
      Initialize(hstream);
   }

   virtual ~CSparseStream() { }

   void Initialize(HANDLE hstream = INVALID_HANDLE_VALUE) {
      m_hstream = hstream;
   }

public:
   operator HANDLE() const { return(m_hstream); }

public:
   BOOL IsStreamSparse() const;
   BOOL MakeSparse();
   BOOL DecommitPortionOfStream(
      __int64 qwFileOffsetStart, __int64 qwFileOffsetEnd);

   FILE_ALLOCATED_RANGE_BUFFER* QueryAllocatedRanges(PDWORD pdwNumEntries);
```

China-pub.com
下载


```
   BOOL FreeAllocatedRanges(FILE_ALLOCATED_RANGE_BUFFER* pfarb);

private:
   HANDLE m_hstream;
private:
   static BOOL AreFlagsSet(DWORD fdwFlagBits, DWORD fFlagsToCheck) {
      return((fdwFlagBits & fFlagsToCheck) == fFlagsToCheck);
   }
};

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileSystemSupportSparseStreams(
   PCTSTR pszVolume) {

   DWORD dwFileSystemFlags = 0;
   BOOL fOk = GetVolumeInformation(pszVolume, NULL, 0, NULL, NULL,
      &dwFileSystemFlags, NULL, 0);
   fOk = fOk && AreFlagsSet(dwFileSystemFlags, FILE_SUPPORTS_SPARSE_FILES);
   return(fOk);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::IsStreamSparse() const {

   BY_HANDLE_FILE_INFORMATION bhfi;
   GetFileInformationByHandle(m_hstream, &bhfi);
   return(AreFlagsSet(bhfi.dwFileAttributes, FILE_ATTRIBUTE_SPARSE_FILE));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::MakeSparse() {

   DWORD dw;
   return(DeviceIoControl(m_hstream, FSCTL_SET_SPARSE,
      NULL, 0, NULL, 0, &dw, NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DecommitPortionOfStream(
   __int64 qwOffsetStart, __int64 qwOffsetEnd) {

   // NOTE: This function does not work if this file is memory-mapped.
   DWORD dw;
   FILE_ZERO_DATA_INFORMATION fzdi;
   fzdi.FileOffset.QuadPart = qwOffsetStart;
   fzdi.BeyondFinalZero.QuadPart = qwOffsetEnd + 1;
   return(DeviceIoControl(m_hstream, FSCTL_SET_ZERO_DATA, (LPVOID) &fzdi,
      sizeof(fzdi), NULL, 0, &dw, NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileContainAnySparseStreams(
```

China-pub.com 下载

```
   PCTSTR pszPathname) {

   DWORD dw = GetFileAttributes(pszPathname);
   return((dw == 0xffffffff)
      ? FALSE : AreFlagsSet(dw, FILE_ATTRIBUTE_SPARSE_FILE));
}

//////////////////////////////////////////////////////////////////////////////

inline FILE_ALLOCATED_RANGE_BUFFER* CSparseStream::QueryAllocatedRanges(
   PDWORD pdwNumEntries) {

   FILE_ALLOCATED_RANGE_BUFFER farb;
   farb.FileOffset.QuadPart = 0;
   farb.Length.LowPart =
      GetFileSize(m_hstream, (PDWORD) &farb.Length.HighPart);

   // There is no way to determine the correct memory block size prior to
   // attempting to collect this data, so I just picked 100 * sizeof(*pfarb)
   DWORD cb = 100 * sizeof(farb);
   FILE_ALLOCATED_RANGE_BUFFER* pfarb = (FILE_ALLOCATED_RANGE_BUFFER*)
      HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cb);

   DeviceIoControl(m_hstream, FSCTL_QUERY_ALLOCATED_RANGES,
      &farb, sizeof(farb), pfarb, cb, &cb, NULL);
   *pdwNumEntries = cb / sizeof(*pfarb);
   return(pfarb);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::FreeAllocatedRanges(
   FILE_ALLOCATED_RANGE_BUFFER* pfarb) {

   // Free the queue entry's allocated memory
   return(HeapFree(GetProcessHeap(), 0, pfarb));
}

//////////////////////////////// End Of File ////////////////////////////////
```

China-pub.com
下载

第18章 堆 栈

对内存进行操作的第三个机制是使用堆栈。堆栈可以用来分配许多较小的数据块。例如，若要对链接表和链接树进行管理，最好的方法是使用堆栈，而不是第15章介绍的虚拟内存操作方法或第17章介绍的内存映射文件操作方法。堆栈的优点是，可以不考虑分配粒度和页面边界之类的问题，集中精力处理手头的任务。堆栈的缺点是，分配和释放内存块的速度比其他机制要慢，并且无法直接控制物理存储器的提交和回收。

从内部来讲，堆栈是保留的地址空间的一个区域。开始时，保留区域中的大多数页面没有被提交物理存储器。当从堆栈中进行越来越多的内存分配时，堆栈管理器将把更多的物理存储器提交给堆栈。物理存储器总是从系统的页文件中分配的，当释放堆栈中的内存块时，堆栈管理器将收回这些物理存储器。

Microsoft并没有以文档的形式来规定堆栈释放和收回存储器时应该遵循的具体规则，Windows 98与Windows 2000的规则是不同的。可以这样说，Windows 98更加注重内存的使用，因此只要可能，它就收回堆栈。Windows 2000更加注重速度，因此它往往较长时间占用物理存储器，只有在一段时间后页面不再使用时，才将它返回给页文件。Microsoft常常进行适应性测试并运行各种不同的条件，以确定在大部分时间内最适合的规则。随着使用这些规则的应用程序和硬件的变更，这些规则也会有所变化。如果了解这些规则对你的应用程序非常关键，那么请不要使用堆栈。相反，可以使用虚拟内存函数（即VirtualAlloc和VirtualFree），这样，就能够控制这些规则。

18.1 进程的默认堆栈

当进程初始化时，系统在进程的地址空间中创建一个堆栈。该堆栈称为进程的默认堆栈。按照默认设置，该堆栈的地址空间区域的大小是1 MB。但是，系统可以扩大进程的默认堆栈，使它大于其默认值。当创建应用程序时，可以使用/HEAP链接开关，改变堆栈的1MB默认区域大小。由于DLL没有与其相关的堆栈，所以当链接DLL时，不应该使用/HEAP链接开关。/HEAP链接开关的句法如下：

```
/HEAP:reserve[,commit]
```

许多Windows函数要求进程使用其默认堆栈。例如，Windows 2000的核心函数均使用Unicode字符和字符串执行它们的全部操作。如果调用Windows函数的ANSI版本，那么该ANSI版本必须将ANSI字符串转换成Unicode字符串，然后调用同一个函数的Unicode版本。为了进行字符串的转换，ANSI函数必须分配一个内存块，以便放置Unicode版本的字符串。该内存块是从你的进程的默认堆栈中分配的。Windows的其他许多函数需要使用一些临时内存块，这些内存块是从进程的默认堆栈中分配的。另外，老的16位Windows函数LocalAlloc和GlobalAlloc也是从进程的默认堆栈中进行它们的内存分配的。

由于进程的默认堆栈可供许多Windows函数使用，你的应用程序有许多线程同时调用各种Windows函数，因此对默认堆栈的访问是顺序进行的。换句话说，系统必须保证在规定的时间内，每次只有一个线程能够分配和释放默认堆栈中的内存块。如果两个线程试图同时分配默认堆栈中的内存块，那么只有一个线程能够分配内存块，另一个线程必须等待第一个线程的内存

China-pub.com 下载

块分配之后，才能分配它的内存块。一旦第一个线程的内存块分配完，堆栈函数将允许第二个线程分配内存块。这种顺序访问方法对速度有一定的影响。如果你的应用程序只有一个线程，并且你想要以最快的速度访问堆栈，那么应该创建你自己的独立的堆栈，不要使用进程的默认堆栈。不幸的是，你无法告诉Windows函数不要使用默认堆栈，因此，它们对堆栈的访问总是顺序进行的。

单个进程可以同时拥有若干个堆栈。这些堆栈可以在进程的寿命期中创建和撤消。但是，默认堆栈是在进程开始执行之前创建的，并且在进程终止运行时自动被撤消。不能撤消进程的默认堆栈。每个堆栈均用它自己的堆栈句柄来标识，用于分配和释放堆栈中的内存块的所有堆栈函数都需要这个堆栈句柄作为其参数。

可以通过调用GetProcessHeap函数获取你的进程默认堆栈的句柄：

```
HANDLE GetProcessHeap();
```

18.2 为什么要创建辅助堆栈

除了进程的默认堆栈外，可以在进程的地址空间中创建一些辅助堆栈。由于下列原因，你可能想要在自己的应用程序中创建一些辅助堆栈：

- 保护组件。
- 更加有效地进行内存管理。
- 进行本地访问。
- 减少线程同步的开销。
- 迅速释放。

下面让我们来详细说明每个原因。

18.2.1 保护组件

假如你的应用程序需要保护两个组件，一个是节点结构的链接表，一个是BRANCH结构的二进制树。你有两个源代码文件，一个是LnkLst.cpp，它包含负责处理NODE链接表的各个函数，另一个文件是BinTree.cpp，它包含负责处理分支的二进制树的各个函数。

如果节点和分支一道存储在单个堆栈中，那么这个组合堆栈将类似图18-1所示的样子。

现在假设链接表代码中有一个错误，它使节点 1后面的8个字节不小心被改写了，从而导致分支3中的数据被破坏。当BinTree.cpp文件中的代码后来试图遍历二进制树时，它将无法进行这项操作，因为它的内存已经被破坏。当然，这使你认为二进制树代码中存在一个错误，而实际上错误是在链接表代码中。由于不同类型的对象混合放在单个堆栈中，因此跟踪和确定错误将变得非常困难。

通过创建两个独立的堆栈，一个堆栈用于存放节点，另一个堆栈用于存放分支，就能够确定你的问题。你的链接表代码中的一个小错误不会破坏你的二进制树的完整性。反过来，二进制树中的小错误也不会影响链接表代码中的数据完整性。但是，你的代码中的错误仍然可能导致对堆栈进行杂乱的内存写操作，不过出现这种情况的可能性很小。



图18-1 将节点和分支存放在一起的单个堆栈

China-pub.com
下载

18.2.2 更有效的内存管理

通过在堆栈中分配同样大小的对象，就可以更加有效地管理堆栈。例如，假设每个节点结构需要24字节，每个分支结构需要32字节。所有这些对象均从单个堆栈中分配。图18-2显示了单个堆栈中已经分配的若干个节点和分支对象占满了这个堆栈。如果节点 2和节点4被释放，堆栈中的内存将变成许多碎片。这时，如果试图分配分支结构，那么尽管分支只需要32个字节，而实际上可以使用的有48个字节，但是分配仍将失败。

如果每个堆栈只包含大小相同的对象，那么释放一个对象后，另一个对象就可以恰好放入被释放的对象空间中。



图18-2 变成碎片的单个堆栈包含若干个节点和分支对象

18.2.3 进行本地访问

每当系统必须在RAM与系统的页文件之间进行RAM页面的交换时，系统的运行性能就会受到很大的影响。如果经常访问局限于一个小范围地址的内存，那么系统就不太可能需要在 RAM与磁盘之间进行页面的交换。

所以，在设计应用程序的时候，如果有些数据将被同时访问，那么最好把它们分配在互相靠近的位置上。让我们回到链接表和二进制树的例子上来，遍历链接表与遍历二进制树之间并无什么关系。如果将所有的节点放在一起（放在一个堆栈中），就可以使这些节点位于相邻的页面上。实际上，若干个节点很可能恰好放入单个物理内存页面上。遍历链接表将不需要 CPU为了访问每个节点而引用若干不同的内存页面。

如果将节点和分支分配在单个页面上，那么节点就不一定会互相靠在一起。在最坏的情况下，每个内存页面上可能只有一个节点，而其余的每个页面则由分支占用。在这种情况下，遍历链接表将可能导致每个节点的页面出错，从而使进程运行得极慢。

18.2.4 减少线程同步的开销

正如下面就要介绍的那样，按照默认设置，堆栈是顺序运行的，这样，如果多个线程试图同时访问堆栈，就不会使数据受到破坏。但是，堆栈函数必须执行额外的代码，以保证堆栈对线程的安全性。如果要进行大量的堆栈分配操作，那么执行这些额外的代码会增加很大的负担，从而降低你的应用程序的运行性能。当你创建一个新堆栈时，可以告诉系统，只有一个线程将访问该堆栈，因此额外的代码将不执行。但是要注意，现在你要负责保证堆栈对线程的安全性。系统将不对此负责。

18.2.5 迅速释放堆栈

最后要说明的是，将专用堆栈用于某些数据结构后，就可以释放整个堆栈，而不必显式释放堆栈中的每个内存块。例如，当Windows Explorer遍历硬盘驱动器的目录层次结构时，它必须在内存中建立一个树状结构。如果你告诉Windows Explorer刷新它的显示器，它只需要撤消包含这个树状结构的堆栈并且重新运行即可（当然，假定它将专用堆栈用于存放目录树信息）。对于许多应用程序来说，这是非常方便的，并且它们也能更快地运行。

China-pub.com 下载

18.3 如何创建辅助堆栈

你可以在进程中创建辅助堆栈，方法是让线程调用HeapCreate函数：

```
HANDLE HeapCreate(
   DWORD fdwOptions,
   SIZE_T dwInitialSize,
   SIZE_T dwMaximumSize);
```

第一个参数fdwOptions用于修改如何在堆栈上执行各种操作。你可以设定0、HEAP_NO_SERIALIZE、HEAP_GENERATE_EXCEPTIONS或者是这两个标志的组合。

按照默认设置，堆栈将顺序访问它自己，这样，多个线程就能够分配和释放堆栈中的内存块而不至于破坏堆栈。当试图从堆栈分配一个内存块时，HeapAlloc函数（下面将要介绍）必须执行下列操作：

1) 遍历分配的和释放的内存块的链接表。

2) 寻找一个空闲内存块的地址。

3) 通过将空闲内存块标记为“已分配”分配新内存块。

4) 将新内存块添加给内存块链接表。

下面这个例子说明为什么应该避免使用HEAP_NO_SERIALIZE标志。假定有两个线程试图同时从同一个堆栈中分配内存块。线程1执行上面的第一步和第二步，获得了空闲内存块的地址。但是，在该线程可以执行第三步之前，它的运行被线程2抢占，线程2得到一个机会来执行上面的第一步和第二步。由于线程1尚未执行第三步，因此线程2发现了同一个空闲内存块的地址。

由于这两个线程都发现了堆栈中它们认为是空闲的内存块，因此线程1更新了链接表，给新内存块做上了“已分配”的标记。然后线程2也更新了链接表，给同一个内存块做上了“已分配”标记。到现在为止，两个线程都没有发现问题，但是两个线程得到的是完全相同的内存块的地址。

这种类型的错误是很难跟踪的，因为它不会立即表现出来。相反，这个错误会在后台等待着，直到很不适合的时候才显示出来。可能出现的问题是：

- 内存块的链接表已经被破坏。在试图分配或释放内存块之前，这个问题不会被发现。
- 两个线程共享同一个内存块。线程1和线程2会将信息写入同一个内存块。当线程1查看该内存块的内容时，它将无法识别线程2提供的数据。
- 一个线程可能继续使用该内存块并且将它释放，导致另一个线程改写未分配的内存。这将破坏该堆栈。

解决这个问题的办法是让单个线程独占对堆栈和它的链接表的访问权，直到该线程执行了对堆栈的全部必要的操作。如果不使用HEAP_NO_SERIALIZE标志，就能够达到这个目的。只有当你的进程具备下面的一个或多个条件时，才能安全地使用HEAP_NO_SERIALIZE标志：

- 你的进程只使用一个线程。
- 你的进程使用多个线程，但是只有单个线程访问该堆栈。
- 你的进程使用多个线程，但是它设法使用其他形式的互斥机制，如关键代码段、互斥对象和信标（第8、9章中介绍），以便设法自己访问堆栈。

如果对是否可以使用HEAP_NO_SERIALIZE标志没有把握，那么请不要使用它。如果不使用该标志，每当调用堆栈函数时，线程的运行速度会受到一定的影响，但是不会破坏你的堆栈及其数据。

China-pub.com
下载

另一个标志HEAP_GENERATE_EXCEPTIONS，会在分配或重新分配堆栈中的内存块的尝试失败时，导致系统引发一个异常条件。所谓异常条件，只不过是系统使用的另一种方法，以便将已经出现错误的情况通知你的应用程序。有时在设计应用程序时让它查看异常条件比查看返回值要更加容易些。异常条件将在第23、24和25章中介绍。

HeapCreate的第二个参数dwInitialSize用于指明最初提交给堆栈的字节数。如果必要的话，HeapCreate函数会将这个值圆整为CPU页面大小的倍数。最后一个参数dwMaximumSize用于指明堆栈能够扩展到的最大值（即系统能够为堆栈保留的地址空间的最大数量）。如果dwMaximumSize大于0，那么你创建的堆栈将具有最大值。如果尝试分配的内存块会导致堆栈超过其最大值，那么这种尝试就会失败。

如果dwMaximumSize的值是0，那么可以创建一个能够扩展的堆栈，它没有内在的限制。从堆栈中分配内存块只需要使堆栈不断扩展，直到物理存储器用完为止。如果堆栈创建成功，HeapCreate函数返回一个句柄以标识新堆栈。该句柄可以被其他堆栈函数使用。

18.3.1 从堆栈中分配内存块

若要从堆栈中分配内存块，只需要调用HeapAlloc函数：

```
PVOID HeapAlloc(
   HANDLE hHeap,
   DWORD fdwFlags,
   SIZE_T dwBytes);
```

第一个参数hHeap用于标识分配的内存块来自的堆栈的句柄。dwBytes参数用于设定从堆栈中分配的内存块的字节数。参数fdwFlags用于设定影响分配的各个标志。目前支持的标志只有3个，即HEAP_ZERO_MEMORY、HEAP_GENERATE_EXCEPTIONS和HEAP_NO_SERIALIZE。

HEAP_ZERO_MEMORY标志的作用应该是非常清楚的。该标志使得HeapAlloc在返回前用0来填写内存块的内容。第二个标志HEAP_GENERATE_EXCEPTIONS用于在堆栈中没有足够的内存来满足需求时使HeapAlloc函数引发一个软件异常条件。当用HeapCreate函数创建堆栈时，可以设定HEAP_GENERATE_EXCEPTIONS标志，它告诉堆栈，当不能分配内存块时，就应该引发一个异常条件。如果在调用HeapCreate函数时设定了这个标志，那么当调用HeapAlloc函数时，就不需要设定该标志。另外，你可能想要不使用该标志来创建堆栈。在这种情况下，为HeapAlloc函数设定该标志只会影响对HeapAlloc函数的一次调用，并不是每次调用都会受到影响。

如果HeapAlloc运行失败，引发一个异常条件，那么这个异常条件将是表18-1中的两个异常条件之一。

表18-1 异常条件

标 志	含 义
STATUS_NO_MEMORY	由于内存不够，分配内存块的尝试失败
STATUS_ACCESS_VIOLATION	由于堆栈被破坏，或者函数的参数不正确，分配内存块的尝试失败

如果内存块已经成功地分配，HeapAlloc返回内存块的地址。如果内存不能分配并且没有设定HEAP_GENERATE_EXCEPTIONS标志，那么HeapAlloc函数返回NULL。

最后一个标志HEAP_NO_SERIALIZE可以用来强制对HeapAlloc函数的调用与访问同一个堆栈的其他线程不按照顺序进行。在使用这个标志时应该格外小心，因为如果其他线程在同一

China-pub.com 下载

时间使用该堆栈，那么堆栈就会被破坏。当从你的进程的默认堆栈中分配内存块时，决不要使用这个标志，因为数据可能被破坏，你的进程中的其他线程可能在同一时间访问默认堆栈。

Windows 98 如果调用HeapAlloc函数并且要求分配大于256 MB的内存块，Windows 98就将它看成是一个错误，函数的调用将失败。注意，在这种情况下，该函数总是返回NULL，并且不会引发异常条件，即使你在创建堆栈或者试图分配内存块时使用HEAP_GENERATE_ EXCEPTIONS标志，也不会引发异常条件。

注意 当你分配较大的内存块（大约1 MB或者更大）时，最好使用VirtualAlloc函数，应该避免使用堆栈函数。

18.3.2 改变内存块的大小

常常需要改变内存块的大小。有些应用程序开始时分配的内存块比较大，然后，当所有数据放入内存块后，再缩小内存块的大小。有些应用程序开始时分配的内存块比较小，后来需要将更多的数据拷贝到内存块中去时，再设法扩大它的大小。如果要改变内存块的大小，可以调用HeapReAlloc函数：

```
PVOID HeapReAlloc(
   HANDLE hHeap,
   DWORD fdwFlags,
   PVOID pvMem,
   SIZE_T dwBytes);
```

与其他情况一样，hHeap参数用于指明包含你要改变其大小的内存块的堆栈。 fdwFlags参数用于设定改变内存块大小时HeapReAlloc函数应该使用的标志。可以使用的标志只有下面4个，即HEAP_GENERATE_EXCEPTIONS、HEAP_NO_SERIALIZE、HEAP_ZERO_MEMORY和HEAP_REALLOC_IN_PLACE_ONLY。

前面两个标志在用于HeapAlloc时，其作用相同。HEAP_ZERO_MEMORY标志只有在你扩大内存块时才使用。在这种情况下，内存块中增加的字节将被置 0。如果内存块已经被缩小，那么该标志不起作用。

HEAP_REALLOC_IN_PLACE_ONLY标志告诉HeapReAlloc函数，它不能移动堆栈中的内存块。如果内存块在增大，HeapReAlloc函数可能试图移动内存块。如果HeapReAlloc能够扩大内存块而不移动它，那么它将会这样做并且返回内存块的原始地址。另外，如果 HeapReAlloc必须移动内存块的内容，则返回新的较大内存块的地址。如果内存块被缩小， HeapReAlloc将返回内存块的原始地址。如果内存块是链接表或二进制树的组成部分，那么可以设定HEAP_REALLOC_IN_PLACE_ONLY标志。在这种情况下，链接表或二进制树中的其他节点可能拥有该节点的指针，改变堆栈中的节点位置会破坏链接表的完整性。

其余的两个参数pvMem和dwBytes用于设定你要改变其大小的内存块的地址和内存块的新的大小（以字节为计量单位）。HeapReAlloc既可以返回新的改变了大小的内存块的地址，也可以在内存块不能改变大小时返回NULL。

18.3.3 了解内存块的大小

当内存块分配后，可以调用HeapSize函数来检索内存块的实际大小：

```
SIZE_T HeapSize(
   HANDLE hHeap,
```

China-pub.com
下载


```
    DWORD fdwFlags,
    LPCVOID pvMem);
```

参数hHeap用于标识堆栈，参数pvMem用于指明内存块的地址。参数fdwFlags既可以是0，也可以是HEAP_NO_SERIALIZE。

18.3.4 释放内存块

当不再需要内存块时，可以调用HeapFree函数将它释放：

```
BOOL HeapFree(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem);
```

HeapFree函数用于释放内存块，如果它运行成功，便返回TRUE。参数fdwFlags既可以是0，也可以是HEAP_NO_SERIALIZE。调用这个函数可使堆栈管理器收回某些物理存储器，但是这没有保证。

18.3.5 撤消堆栈

如果应用程序不再需要它创建的堆栈，可以通过调用HeapDestroy函数将它撤消：

```
BOOL HeapDestroy(HANDLE hHeap);
```

调用HeapDestroy函数可以释放堆栈中包含的所有内存块，也可以将堆栈占用的物理存储器和保留的地址空间区域重新返回给系统。如果该函数运行成功， HeapDestroy返回TRUE。如果在进程终止运行之前没有显式撤消堆栈，那么系统将为你将它撤消。但是，只有当进程终止运行时，堆栈才能被撤消。如果线程创建了一个堆栈，当线程终止运行时，该堆栈将不会被撤消。

在进程完全终止运行之前，系统不允许进程的默认堆栈被撤消。如果将进程的默认堆栈的句柄传递给HeapDestroy函数，系统将忽略对该函数的调用。

18.3.6 用C++程序来使用堆栈

使用堆栈的最好方法之一是将堆栈纳入现有的 C++程序。在C++中，调用new操作符，而不是调用通常的C运行期例程malloc，就可以执行类对象的分配操作。然后，当我们不再需要这个类对象时，调用delete操作符，而不是调用通常的C运行期例程free将它释放。例如，我们有一个称为CSomeClass的类，我们想要分配这个类的一个实例，那么可以使用类似下面的句法：

```
CSomeClass* pSomeClass = new CSomeClass;
```

当C++编译器查看这一行代码时，它首先查看 CSomeClass类是否包含new操作符的成员函数。如果包含，那么编译器就生成调用该函数的代码。如果编译器没有找到重载 new操作符的函数，那么编译器将生成调用标准C++的new操作符函数的代码。

当完成对已分配对象的使用后，可以通过调用 delete操作符将它撤消：

```
delete pSomeClass;
```

通过为我们的C++类重载new和delete操作符，就能够很容易地利用堆栈函数。为此，让我们在头文件中将我们的CSomeClass类定义为如下的形式：

```
class CSomeClass {
private:
```

China-pub.com 下载

```
   static HANDLE s_hHeap;
   static UINT s_uNumAllocsInHeap;

   // Other private data and member functions
   :
   :
public:
   void* operator new (size_t size);
   void operator delete (void* p);
   // Other public data and member functions
   :
   :
};
```

在这个代码段中，我声明了两个成员变量，即s_hHeap和s_uNumAllocsInHeap作为静态变量。由于它们是静态变量，因此C++将使CSomeClass的所有实例共享相同的变量，也就是说，C++将不为已经创建的该类的每个实例分配独立的s_hHeap和s_uNumAllocsInHeap变量。这个情况对我们来说是非常重要的，因为我们的CSomeClass类的所有实例都在相同的堆栈中分配。

变量s_hHeap将包含分配CSomeClass对象时所在堆栈的句柄。s_uNumAllocsInHeap变量只是一个计数器，用于计算堆栈中已经分配了多少个CSomeClass对象。每次在堆栈中分配一个新的CSomeClass对象时，s_uNumAllocsInHeap的数字就递减。当s_uNumAllocsInHeap的数字到达0时，堆栈就不再需要并被释放。用于对堆栈进行操作的代码应该包括在类似下面的.cpp文件中：

```
HANDLE CSomeClass::s_hHeap = NULL;
UINT CSomeClass::s_uNumAllocsInHeap = 0;

void* CSomeClass::operator new (size_t size) {
   if (s_hHeap == NULL) {
      // Heap does not exist; create it.
      s_hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);

      if (s_hHeap == NULL)
         return(NULL);
   }
   // The heap exists for CSomeClass objects.
   void* p = HeapAlloc(s_hHeap, 0, size);

   if (p != NULL) {
      // Memory was allocated successfully; increment
      // the count of CSomeClass objects in the heap.
      s_uNumAllocsInHeap++;
   }

   // Return the address of the allocated CSomeClass object.
   return(p);
}
```

注意，我首先在代码的开始处定义了两个静态数字变量，即s_hHeap和s_uNumAllocsInHeap，并且分别将它们初始化为NULL和0。

C++的new操作符接受一个参数，即Size。该参数用于指明存放CSomeClass对象所需要的字节数。new操作符函数的第一个任务是创建一个堆栈，如果这样的堆栈尚未创建的话。这只

China-pub.com
下载

需要查看s_hHeap变量来了解它的值是否是NULL。如果是NULL，那么就调用HeapCreate函数，创建一个新堆栈，同时将HeapCreate返回的句柄保存在s_hHeap中，这样，下次调用new操作符时，就不会创建另一个堆栈，而是使用我们刚刚创建的堆栈。

当调用上面的HeapCreate函数时，我使用了HEAP_NO_SERIALIZE标志，因为示例代码的剩余部分不具备对多线程安全的特性。调用 HeapCreate函数时使用的另外两个参数分别用于指明堆栈的初始大小和它的最大值。这里我为这两个值都选择了 0。第一个0表示该堆栈没有初始大小的值。第二个0表示该堆栈可以根据需要进行扩展。根据你的需要，可以改变这两个值中的任何一个，也可以同时改变两个值。

你可能认为将new操作符函数的size参数作为第二个参数传递给 HeapCreate函数是值得的。如果这样的话，你可以对堆栈进行初始化，使它大得足以包含该类的一个实例。然后，当HeapAlloc第一次被调用时，它将以更快的速度运行，因为堆栈不必改变它的大小以便存放类的实例。但是，事情并不总是按照你的想像来进行的。由于堆栈中的每个已分配内存块都需要与之相关的开销，因此调用 HeapAlloc时仍然必须改变堆栈的大小，这样它才能变得足够大，以便放置一个类的实例和它相关的开销。

一旦堆栈创建完成，就可以使用 HeapAlloc函数从该堆栈中分配新的 CSomeClass对象。第一个参数是堆栈的句柄，第二个参数是 CSomeClass对象的大小。HeapAlloc返回分配的内存块的地址。

当这个分配操作成功地执行时，我对 s_uNumAllocsInHeap变量进行了递增，这样就可以知道堆栈中已分配了一个内存块。New操作符函数做的最后一项工作是返回新分配的 CSomeClass对象的地址。

这就是创建新 CSomeClass对象的整个过程。下面要介绍的是，当应用程序不再需要CSomeClass时，如何将它撤消。这是delete操作符函数的责任，它的代码如下：

```
void CSomeClass::operator delete (void* p) {
   if (HeapFree(s_hHeap, 0, p)) {
      // Object was deleted successfully.
      s_uNumAllocsInHeap--;
   }

   if (s_uNumAllocsInHeap == 0) {
      // If there are no more objects in the heap,
      // destroy the heap.
      if (HeapDestroy(s_hHeap)) {
         // Set the heap handle to NULL so that the new operator
         // will know to create a new heap if a new CSomeClass
         // object is created.
         s_hHeap = NULL;
      }
   }
}
```

delete操作符函数只接受一个参数，即被删除的对象的地址。该函数进行的第一项操作是调用HeapFree，将堆栈的句柄和被释放的对象的地址传递给它。如果该对象被成功地释放了，s_uNumAllocsInHeap的值就被递减1，表示堆栈中又少了一个CSomeClass对象。接着该函数要检查s_uNumAllocsInHeap的值是否是0。如果是0，那么该函数就调用HeapDestroy，将堆栈的句柄传递给它。如果堆栈被成功地撤消了， s_hHeap将被设置为NULL。这一点非常重要，因为我们的程序可能在将来的某个时候分配另一个 CSomeClass对象。当它进行这项操作时， new

China-pub.com 下载

操作符将被调用，同时该操作符将查看s_hHeap变量，以确定它是应该使用现有的堆栈还是创建一个新堆栈。

这个例子显示了一种使用多个堆栈的简便方法。这个例子很容易建立，并且可以纳入若干个类中。不过应该对继承性问题有所考虑。如果你用CsomeClass类作为一个基类，派生一个新类，那么这个新类就可以继承CSomeClass的new和delete操作符。这个新类也可以继承CSomeClass的堆栈，这意味着当new操作符用于派生类时，该派生类对象的内存将从CSomeClass使用的同一个堆栈中分配。根据具体情况，你也许希望这样，也许不希望这样。如果对象的大小差别很大，建立的堆栈环境可能使你的堆栈变得支离破碎。正如本章前面部分中的“组件保护”和“更加有效地进行内存管理”两节所说的那样，你可能更加难以跟踪代码中的错误。

如果想将一个独立的堆栈用于各个派生类，只需要重复我在CSomeClass类中所进行的操作。也就是说，加上另一组s_hHeap和s_uNumAllocsInHeap变量，为new和delete操作符拷贝该代码。当进行编译时，编译器将发现你已经为该派生类重载了new和delete操作符，并将调用这些函数，而不是调用基类中的那些函数。

不为每个类创建新堆栈的唯一优点是，不必为每个堆栈分配开销和内存。但是，与这些堆栈相关的开销和内存并不很大，并且与带来的好处相比，这样做也许是值得的。我们采取的折中方案是，当你的应用程序已经经过很好的测试并且将要推向市场时，让每个类使用它自己的堆栈，让派生类共享基类的堆栈。不过堆栈碎片问题仍然可能存在。

18.4 其他堆栈函数

除了上面介绍的堆栈函数外，Windows还提供了若干个别的函数。下面对它们作一个简单的介绍。

ToolHelp的各个函数（第4章后面部分讲过）可以用来枚举进程的各个堆栈和这些堆栈中分配的内存块。关于这些函数的详细说明，请参见Platform SDK文档中的下列函数：Heap32First、Heap32Next、Heap32ListFirst和Heap32ListNext。ToolHelp函数的优点在于，在Windows 98和Windows 2000中都能够使用它们。

本节中介绍的其他堆栈函数只存在于Windows 2000中。

由于进程的地址空间中可以存在多个堆栈，因此可以使用GetProcessHeaps函数来获取现有堆栈的句柄：

```
DWORD GetProcessHeaps(
   DWORD dwNumHeaps,
   PHANDLE pHeaps);
```

若要调用GetProcessHeaps函数，必须首先分配一个HANDLE数组，然后调用下面的函数：

```
HANDLE hHeaps[25];
DWORD dwHeaps = GetProcessHeaps(25, hHeaps);
if (dwHeaps > 25) {
   // More heaps are in this process than we expected.
} else {
   // hHeaps[0] through hHeap[dwHeaps - 1]
   // identify the existing heaps.
}
```

注意，当该函数返回时，你的进程的默认堆栈的句柄也包含在堆栈句柄的数组中。

China-pub.com
下载

HeapValidate函数用于验证堆栈的完整性：

```
BOOL HeapValidate(
   HANDLE hHeap,
   DWORD fdwFlags,
   LPCVOID pvMem);
```

调用该函数时，通常要传递一个堆栈句柄，一个值为0的标志（唯一的另一个合法标志是HEAP_NO_SERIALIZE），并且为pvMem传递NULL。然后，该函数将遍历堆栈中的内存块以确保所有内存块都完好无损。为了使该函数运行得更快，可以为参数pvMem传递一个特定的内存块的地址。这样做可使该函数只检查单个内存块的有效性。

若要合并地址中的空闲内存块并收回不包含已经分配的地址内存块的存储器页面，可以调用下面的函数：

```
UINT HeapCompact(
   HANDLE hHeap,
   DWORD fdwFlags);
```

通常情况下，可以为参数fdwFlags传递0，但是也可以传递HEAP_NO_SERIALIZE。

下面两个函数HeapLock和HeapUnlock是结合在一起使用的：

```
BOOL HeapLock(HANDLE hHeap);
BOOL HeapUnlock(HANDLE hHeap);
```

这些函数是用于线程同步的。当调用HeapLock函数时，调用线程将成为特定堆栈的所有者。如果其他任何线程调用堆栈函数（设定相同的堆栈句柄），系统将暂停调用线程的运行，并且在堆栈被HeapUnlock函数解锁之前不允许它醒来。

HeapAlloc、HeapSize和HeapFree等函数在内部调用HeapLock和HeapUnlock函数来确保对堆栈的访问能够顺序进行。自己调用HeapLock或HeapUnlock这种情况是不常见的。

最后一个堆栈函数是HeapWalk：

```
BOOL HeapWalk(
   HANDLE hHeap,
   PPROCESS_HEAP_ENTRY pHeapEntry);
```

该函数只用于调试目的。它使你能够遍历堆栈的内容。可以多次调用该函数。每次调用该函数时，将传递必须分配和初始化的PROCESS_HEAP_ENTRY结构的地址：

```
typedef struct _PROCESS_HEAP_ENTRY {
   PVOID lpData;
   DWORD cbData;
   BYTE cbOverhead;
   BYTE iRegionIndex;
   WORD wFlags;
   union {
      struct {
         HANDLE hMem;
         DWORD dwReserved[ 3 ];
      } Block;
      struct {
         DWORD dwCommittedSize;
         DWORD dwUnCommittedSize;
         LPVOID lpFirstBlock;
         LPVOID lpLastBlock;
```

China-pub.com 下载

```
        } Region;
    };
} PROCESS_HEAP_ENTRY, *LPPROCESS_HEAP_ENTRY, *PPROCESS_HEAP_ENTRY;
```

当开始枚举堆栈中的内存块时，必须将成员 lpData设置为NULL。这将告诉HeapWalk对该结构中的成员进行初始化。当成功地调用 HeapWalk后，可以查看该结构的成员。若要进入堆栈的下一个内存块，只需要再次调用 HeapWalk，传递相同的堆栈句柄和在上次调用该函数时传递的PROCESS_HEAP_ENTRY结构的地址。当HeapWalk返回FALSE时，堆栈中就没有更多的内存块了。关于该结构中的成员的说明，请参见 Platform SDK文档。

在循环调用HeapWalk的时候，必须使用HeapLock和HeapUnlock函数，这样，当遍历堆栈时，其他线程将无法分配和释放堆栈中的内存块。

China-pub.com
下载

第四部分　动态链接库

第19章　DLL基础

自从Microsoft公司推出第一个版本的Windows操作系统以来，动态链接库（DLL）一直是这个操作系统的基础。 Windows API中的所有函数都包含在 DLL中。3个最重要的DLL是Kernel32.dll，它包含用于管理内存、进程和线程的各个函数； User32.dll，它包含用于执行用户界面任务（如窗口的创建和消息的传送）的各个函数； GDI32.dll，它包含用于画图和显示文本的各个函数。

Windows还配有若干别的 DLL，它们提供了用于执行一些特殊任务的函数。例如，AdvAPI32.dll包含用于实现对象安全性、注册表操作和事件记录的函数； ComDlg32.dll包含常用对话框（如File Open和File Save）；ComCtl32.DLL则支持所有的常用窗口控件。

本章将要介绍如何为应用程序创建DLL。下面是为什么要使用DLL的一些原因：

- 它们扩展了应用程序的特性。由于 DLL能够动态地装入进程的地址空间，因此应用程序能够在运行时确定需要执行什么操作，然后装入相应的代码，以便根据需要执行这些操作。例如，当一家公司开发了一种产品，想要让其他公司改进或增强该产品的功能时，那么就可以使用DLL。
- 它们可以用许多种编程语言来编写。可以选择手头拥有的最好的语言来编写 DLL。也许你的应用程序的用户界面使用Microsoft Visual Basic编写得最好，但是用C++来处理它的商用逻辑更好。系统允许Visual Basic程序加载C++ DLL、Cobol DLL和Fortran DLL等。
- 它们简化了软件项目的管理。如果在软件开发过程中不同的工作小组在不同的模块上工作，那么这个项目管理起来比较容易。但是，应用程序在销售时附带的文件应该尽量少一些。我知道有一家公司销售的产品附带了 100个DLL——每个程序员最多有 5个DLL。这样，应用程序的初始化时间将会长得吓人，因为系统必须打开 100个磁盘文件之后，程序才能执行它的操作。
- 它们有助于节省内存。如果两个或多个应用程序使用同一个 DLL，那么该DLL的页面只要放入RAM一次，所有的应用程序都可以共享它的各个页面。 C/C++运行期库就是个极好的例子。许多应用程序都使用这个库。如果所有的应用程序都链接到这个静态库，那么sprintf、strcpy和malloc等函数的代码就要多次存在于内存中。但是，如果所有这些应用程序链接到DLL C/C++运行期库，那么这些函数的代码就只需要放入内存一次，这意味着内存的使用将更加有效。
- 它们有助于资源的共享。 DLL可以包含对话框模板、字符串、图标和位图等资源。多个应用程序能够使用DLL来共享这些资源。
- 它们有助于应用程序的本地化。应用程序常常使用 DLL对自己进行本地化。例如，只包含代码而不包含用户界面组件的应用程序可以加载包含本地化用户界面组件的 DLL。
- 它们有助于解决平台差异。不同版本的 Widnows配有不同的函数。开发人员常常想要调用新的函数（如果它们存在于主机的Windows版本上的话）。但是，如果你的源代码包含

China-pub.com 下载

了对一个新函数的调用，而你的应用程序将要在不能提供该函数的 Windows版本上运行，那么操作系统的加载程序将拒绝运行你的进程。即使你实际上从不调用该函数，情况也是这样。如果将这些新函数保存在 DLL中，那么应用程序就能够将它们加载到 Windows的老版本上。当然，你仍然可以成功地调用该函数。

- 它们可以用于一些特殊的目的。 Windows使得某些特性只能为 DLL所用。例如，只有当 DLL中包含某个挂钩通知函数的时候，才能安装某些挂钩（使用 SetWindowsHookEx和 SetWinEventHook来进行安装）。可以通过创建必须在 DLL中生存的 COM对象来扩展 Windows Explorer的外壳程序。对于可以由 Web浏览器加载的、用于创建内容丰富的 Web页的ActiveX控件来说,情况也是一样.

19.1 DLL与进程的地址空间

创建DLL常常比创建应用程序更容易，因为 DLL往往包含一组应用程序可以使用的自主函数。在DLL中通常没有用来处理消息循环或创建窗口的支持代码。 DLL只是一组源代码模块，每个模块包含了应用程序（可执行文件）或另一个 DLL将要调用的一组函数。当所有源代码文件编译后，它们就像应用程序的可执行文件那样被链接程序所链接。但是，对于一个 DLL来说，你必须设定该连链程序的 /DLL开关。这个开关使得链接程序能够向产生的 DLL文件映像发出稍有不同的信息，这样，操作系统加载程序就能将该文件映像视为一个 DLL而不是应用程序。

在应用程序（或另一个DLL）能够调用DLL中的函数之前，DLL文件映像必须被映射到调用进程的地址空间中。若要进行这项操作，可以使用两种方法中的一种，即加载时的隐含链接或运行期的显式链接。隐含链接将在本章的后面部分介绍，显式链接将在第 20章中介绍。

一旦DLL的文件映像被映射到调用进程的地址空间中， DLL的函数就可以供进程中运行的所有线程使用。实际上， DLL几乎将失去它作为 DLL的全部特征。对于进程中的线程来说，DLL的代码和数据看上去就像恰巧是在进程的地址空间中的额外代码和数据一样。当一个线程调用DLL函数时，该DLL函数要查看线程的堆栈，以便检索它传递的参数，并将线程的堆栈用于它需要的任何局部变量。此外， DLL中函数的代码创建的任何对象均由调用线程所拥有，而DLL本身从来不拥有任何东西。

例如，如果VirtualAlloc函数被DLL中的一个函数调用，那么将从调用线程的进程地址空间中保留一个地址空间的区域，该地址空间区域将始终处于保留状态，因为系统并不跟踪 DLL中的函数保留该区域的情况。保留区域由进程所拥有，只有在线程调用 VirtualFree函数或者进程终止运行时才被释放。

如你所知，可执行文件的全局变量和静态变量不能被同一个可执行文件的多个运行实例共享。Windows 98能够确保这一点，方法是在可执行文件被映射到进程的地址空间时为可执行文件的全局变量和静态变量分配相应的存储器。 Windows 2000确保这一点的方法是使用第13章介绍的写入时拷贝（copy-on-write）机制。DLL中的全局变量和静态变量的处理方法是完全相同的。当一个进程将DLL的映像文件映射到它的地址空间中去时，系统将同时创建全局数据变量和静态数据变量的实例。

注意 必须注意的是，单个地址空间是由一个可执行模块和若干个 DLL模块组成的。这些模块中，有些可以链接到静态版本的 C/C++运行期库，有些可以链接到一个 DLL版本的C/C++运行期库，而有些模块（如果不是用 C/C++编写的话）则根本不需要C/C++运行期库。许多开发人员经常会犯一个常见的错误，因为他们忘记了若干个C/C++运行期库可以存在于单个地址空间中。请看下面的代码：

China-pub.com
下载


```
VOID EXEFunc() {
   PVOID pv = DLLFunc();
   // Access the storage pointed to by pv...
   // Assumes that pv is in EXE's C/C++ run-time heap
   free(pv);
}

PVOID DLLFunc() {
   // Allocate block from DLL's C/C++ run-time heap
   return(malloc(100));
}
```

那么你是怎么看待这个问题的呢？上面这个代码能够正确运行吗？DLL函数分配的内存块是由EXE的函数释放的吗？答案是可能的。上面显示的代码并没有为你提供足够的信息。如果EXE和DLL都链接到DLL的C/C++运行期库，那么上面的代码将能够很好地运行。但是，如果两个模块中的一个或者两个都链接到静态C/C++运行期库，那么对free函数的调用就会失败。我经常看到编程人员编写这样的代码，结果都失败了。

有一个很方便的方法可以解决这个问题。当一个模块提供一个用于分配内存块的函数时，该模块也必须提供释放内存的函数。让我们将上面的代码改写成下面的样子：

```
VOID EXEFunc() {
   PVOID pv = DLLFunc();
   // Access the storage pointed to by pv...
   // Makes no assumptions about C/C++ run-time heap
   DLLFreeFunc(pv);
}

PVOID DLLFunc() {
   // Allocate block from DLL's C/C++ run-time heap
   PVOID pv = malloc(100);
   return(pv);
}

BOOL DLLFreeFunc(PVOID pv) {
   // Free block from DLL's C/C++ run-time heap
   return(free(pv));
}
```

这个代码是正确的，它始终都能正确地运行。当你编写一个模块时，不要忘记其他模块中的函数也许没有使用C/C++来编写，因此可能无法使用malloc和free函数进行内存的分配。应该注意不要在代码中使用这些假设条件。另外，在内部调用 malloc和free函数时，这个原则对于C++的new和delete操作符也是适用的。

19.2 DLL的总体运行情况

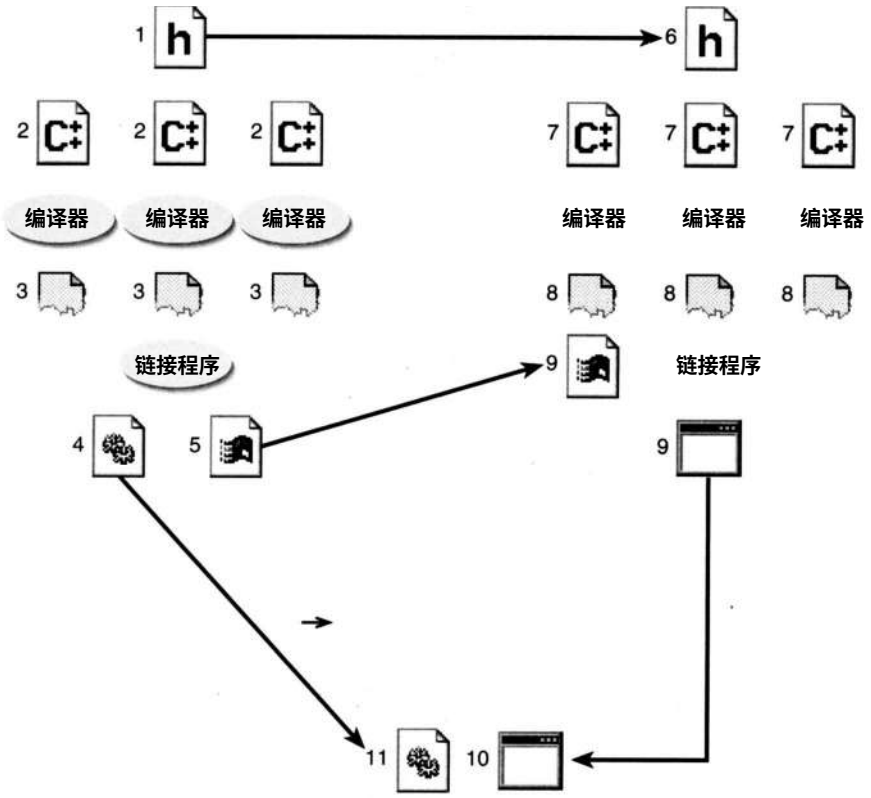
为了全面理解DLL是如何运行的以及你和系统如何使用 DLL，让我们首先观察一下DLL的整个运行情况。图19-1综合说明了它的所有组件一道配合运行的情况。

现在要重点介绍可执行模块和 DL模块之间是如何隐含地互相链接的。隐含链接是最常用的链接类型。Windows也支持显式链接（第20章介绍这个问题）。

在图19-1中你可以看到，当一个模块（比如一个可执行文件）使用DLL中的函数或变量时，将有若干个文件和组件参与发挥作用。为了简单起见，我将“可执行模块”称为来自 DLL的输

China-pub.com 下载

入函数和变量，将“DLL模块”称为用于可执行模块的输出函数和变量。但是要记住，DLL模块能够（并且确实常常）输入包含在其他DLL模块中的函数和变量。



创造DLL：
1) 建立带有输出原型/结构/符号的头文件。
2) 建立实现输出函数/变量的C/C++源文件。
3) 编译器为每个C/C++源文件生成 .obj模块。
4) 链接程序将生成DLL的 .obj模块链接起来。
5) 如果至少输出一个函数/变量，那么链接程序也生成lib 文件。
创造EXE：
6) 建立带有输入原型/结构/符号的头文件。
7) 建立引用输入函数/变量的C/C++源文件。
8) 编译器为每个C/C++源文件生成 .obj源文件。
9) 链接程序将各个 .obj模块链接起来，产生一个 .exe文件(它包含了所需要DLL模块的名字和输入符号的列表)。
运行应用程序：
10) 加载程序为 .exe 创建地址空间。
11) 加载程序将需要的DLL加载到地址空间中进程的主线程开始执行；应用程序启动运行。

图19-1 应用程序如何创建和隐含链接DLL的示意图

若要创建一个从DLL模块输入函数和变量的可执行模块，必须首先创建一个 DLL模块。然后就可以创建可执行模块。

若要创建DLL模块，必须执行下列操作步骤：

1) 首先必须创建一个头文件，它包含你想要从DLL输出的函数原型、结构和符号。DLL的所有源代码模块均包含该头文件，以帮助创建 DLL。后面将会看到，当创建需要使用DLL中包含的函数和变量的可执行模块（或多个模块）时，也需要这个头文件。

2) 要创建一个C/C++源代码模块（或多个模块），用于实现你想要在DLL模块中实现的函数和变量。由于这些源代码模块在创建可执行模块时是不必要的，因此创建 DLL的公司能够保

China-pub.com
下载

护公司的秘密。

3) 创建DLL模块，将使编译器对每个源代码模块进行处理，产生一个 .obj模块（每个源代码模块有一个.obj模块）。

4) 当所有的.obj模块创建完成后，链接程序将所有 .obj模块的内容组合在一起，产生一个DLL映象文件。该映像文件（即模块）包含了用于 DLL的所有二进制代码和全局 /静态数据变量。为了执行这个可执行模块，该文件是必不可少的。

5) 如果链接程序发现DLL的源代码模块至少输出了一个函数或变量，那么链接程序也生成一个.lib文件。这个.lib文件很小，因为它不包含任何函数或变量。它只是列出所有已输出函数和变量的符号名。为了创建可执行模块，该文件是必不可少的。

一旦创建了DLL模块，就可以创建可执行模块。其创建步骤是：

6) 在引用函数、变量、数据、结构或符号的所有源代码模块中，必须包含 DLL开发人员创建的头文件。

7) 要创建一个C/C++源代码模块（或多个模块），用于实现你想要在可执行模块中实现的函数和变量。当然该代码可以引用DLL头文件中定义的函数和变量。

8) 创建可执行模块，将使编译器对每个源代码模块进行处理，生成一个 .obj模块（每个源代码模块有一个.obj模块）。

9) 当所有.obj模块创建完成后，链接程序便将所有的.obj模块的内容组合起来，生成一个可执行的映像文件。该映像文件（或模块）包含了可执行文件的所有二进制代码和全局/静态变量。该可执行模块还包含一个输入节，列出可执行文件需要的所有 DLL模块名（关于各个节的详细说明，参见第17章）。此外，对于列出的每个 DLL名字，该节指明了可执行模块的二进制代码引用了哪些函数和变量符号。下面你会看到操作系统的加载程序将对该输入节进行分析。

一旦DLL和可执行模块创建完成，一个进程就可以执行。当试图运行可执行模块时，操作系统的加载程序将执行下面的操作步骤：

10) 加载程序为新进程创建一个虚拟地址空间。可执行模块被映射到新进程的地址空间。加载程序对可执行模块的输入节进行分析。对于该节中列出的每个 DLL名字，加载程序要找出用户系统上的DLL模块，再将该DLL映射到进程的地址空间。注意，由于 DLL模块可以从另一个DLL模块输入函数和变量，因此 DLL模块可以拥有它自己的输入节。若要对进程进行全面的初始化，加载程序要分析每个模块的输入节，并将所有需要的DLL模块映射到进程的地址空间。如你所见，对进程进行初始化是很费时间的。

一旦可执行模块和所有 DLL模块被映射到进程的地址空间中，进程的主线程就可以启动运行，同时应用程序也可以启动运行。下面各节将更加详细地介绍这个进程的运行情况。

19.3 创建DLL模块

当创建DLL 时，要创建一组可执行模块（或其他DLL）可以调用的函数。DLL可以将变量、函数或C/C++类输出到其他模块。在实际工作环境中，应该避免输出变量，因为这会删除你的代码中的一个抽象层，使它更加难以维护你的 DLL代码。此外，只有当使用同一个供应商提供的编译器对输入C++类的模块进行编译时，才能输出 C++类。由于这个原因，也应该避免输出C++类，除非知道可执行模块的开发人员使用的工具与 DLL模块开发人员使用的工具相同。

当创建DLL模块时，首先应该建立一个头文件，该文件包含了你想要输出的变量（类型和名字）和函数（原型和名字）。头文件还必须定义用于输出函数和变量的任何符号和数据结构。你的DLL的所有源代码模块都应该包含这个头文件。另外，必须分配该头文件，以便它能够包

China-pub.com 下载

含在可能输入这些函数或变量的任何源代码中。拥有单个头文件，供DLL创建程序和可执行模块的创建程序使用，就可以大大简化维护工作。

下面的代码说明了应该如何对单个头文件进行编码，以便同时包含可执行文件和DLL的源代码文件：

```
/*************************************************************************
Module:  MyLib.h
*************************************************************************/


#ifdef MYLIBAPI

// MYLIBAPI should be defined in all of the DLL's source
// code modules before this header file is included.

// All functions/variables are being exported.

#else

// This header file is included by an EXE source code module.
// Indicate that all functions/variables are being imported.
#define MYLIBAPI extern "C" __declspec(dllimport)

#endif

////////////////////////////////////////////////////////////////////////////
// Define any data structures and symbols here.

////////////////////////////////////////////////////////////////////////////

// Define exported variables here. (NOTE: Avoid exporting variables.)
MYLIBAPI int g_nResult;

////////////////////////////////////////////////////////////////////////////

// Define exported function prototypes here.
MYLIBAPI int Add(int nLeft, int nRight);

////////////////////////////// End of File /////////////////////////////////
```

在你的每个DLL源代码文件中，应该包含下面的头文件：

```
/*************************************************************************
Module:  MyLibFile1.cpp
*************************************************************************/


// Include the standard Windows and C-Runtime header files here.
#include <windows.h>


// This DLL source code file exports functions and variables.
#define MYLIBAPI extern "C" __declspec(dllexport)

// Include the exported data structures, symbols, functions, and variables.
#include "MyLib.h"
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////

// Place the code for this DLL source code file here.
int g_nResult;


int Add(int nLeft, int nRight) {
   g_nResult = nLeft + nRight;
   return(g_nResult);
}

//////////////////////////////// End of File /////////////////////////////////
```

当上面的DLL源代码文件被编译时，在MyLib.h头文件的前面使用__declspec(dllexport)对MYLIBAPI进行定义。当编译器看到负责修改变量、函数或C++类的__declspec(dllexport)时,它就知道该变量、函数或C++类是从产生的DLL模块输出的。注意，MYLIBAPI标志被置于头文件中要输出的变量的定义之前和要输出的函数之前。

另外，在源代码文件（MyLibFile1.cpp0）中，MYLIBAPI 标志并不出现在输出的变量和函数之前。MYLIBAPI标志在这里是不必要的，因为编译器在分析头文件时能够记住要输出哪些变量或函数。

你会发现，MYLIBAPI标志包含了extern“C”修改符。只有当你编写C++代码而不是直接编写C代码时，才能使用这个修改符。通常来说， C++编译器可能会改变函数和变量的名字，从而导致严重的链接程序问题。例如，假设你用 C++编写一个DLL，并直接用C编写一个可执行模块，当你创建DLL时，函数名被改变，但是，当你创建可执行模块时，函数名没有改变。当链接程序试图链接可执行模块时，它就会抱怨说，可执行模块引用的符号不存在。如果使用extern“C”，就可以告诉编译器不要改变变量名或函数名，这样，变量和函数就可以供使用 C、C++或任何其他编程语言编写的可执行模块来访问。

现在你已经知道DLL源代码文件是如何使用这个头文件的。但是，可执行模块的源代码文件情况又是如何呢？可执行模块的源代码文件不应该在这个头文件的前面定义 MYLIBAPI。由于MYLIBAPI没有定义，因此头文件将MYLIBAPI定义为__declspec(dllimport)。编译器看到可执行模块的源代码文件从DLL模块输入变量和函数。

如果观察Microsoft的标准Windows头文件，如WinBase.h，你将会发现Microsoft使用的方法基本上与上面介绍的方法相同。

19.3.1 输出的真正含义是什么

上一节介绍的一个真正有意思的东西是 __declspec(dllexport)修改符。当Microsoft的C/C++编译器看到变量、函数原型或C++类之前的这个修改符的时候，它就将某些附加信息嵌入产生的.obj文件中。当链接DLL的所有.obj文件时，链接程序将对这些信息进行分析。

当DLL被链接时，链接程序要查找关于输出变量、函数或 C++类的信息，并自动生成一个.lib文件。该.lib文件包含一个DLL输出的符号列表。当然，如果要链接引用该 DLL的输出符号的任何可执行模块，该.lib文件是必不可少的。除了创建.lib文件外，链接程序还要将一个输出符号表嵌入产生的DLL文件。这个输出节包含一个输出变量、函数和类符号的列表（按字母顺序排列）。该链接程序还将能够指明在何处找到每个符号的相对虚拟地址（ RVA）放入DLL模块。

使用Microsoft的Visual Studio的DumpBin.exe实用程序（带有 -exports开关），你能够看到

China-pub.com 下载

DLL的输出节是个什么样子。下面是 Kernel32.dll的输出节的一个代码段（我已经删除了DUMPBIN的某些输出，这样就不会占用本书的太多篇幅）。

```
C:\WINNT\SYSTEM32>DUMPBIN -exports Kernel32.DLL

Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file kernel32.dll

File Type: DLL

  Section contains the following exports for KERNEL32.dll

           0 characteristics
    36DB3213 time date stamp Mon Mar 01 16:34:27 1999
        0.00 version
           1 ordinal base
         829 number of functions
         829 number of names

    ordinal hint RVA      name

          1    0 0001A3C6 AddAtomA
          2    1 0001A367 AddAtomW
          3    2 0003F7C4 AddConsoleAliasA
          4    3 0003F78D AddConsoleAliasW
          5    4 0004085C AllocConsole
          6    5 0002C91D AllocateUserPhysicalPages
          7    6 00005953 AreFileApisANSI
          8    7 0003F1A0 AssignProcessToJobObject
          9    8 00021372 BackupRead
         10    9 000215CE BackupSeek
         11    A 00021F21 BackupWrite
      .  .
      .  .
      .  .
        828  33B 00003200 lstrlenA
        829  33C 000040D5 lstrlenW

  Summary

       3000 .data
       4000 .reloc
      4D000 .rsrc
      59000 .text
```

如你所见，这些符号是按字母顺序排列的，RVA这一列下面的数字用于指明在DLL文件映像中的什么位置能够找到输出符号的位移量。序号列可以与16位Windows源代码向后兼容，并且它不应该用于现在的应用程序中。hint（提示码）列可供系统用来改进代码的运行性能，在此并不重要。

注意 许多开发人员常常通过为函数赋予一个序号值来输出DLL函数。对于那些来自16位Windows环境的函数来说，情况尤其是如此。但是，Microsoft并没有公布系统DLL的序号值。当你的可执行模块或DLL模块链接到任何一个Windows函数时，

China-pub.com
下载

Microsoft要求你使用符号的名字进行链接。如果你按照序号进行链接，那么你的应用程序有可能无法在其他Windows平台或将来的Windows平台上运行。

实际上，我就遇到过这样的情况。我曾经发布了一个示例应用程序，它使用Microsoft System Journal中的序号。我的应用程序在Windows NT 3.1上运行得很好，但是当Windows NT 3.5推出时，我的应用程序就无法正确地运行。为了解决这个问题，我不得不用函数名代替序号。现在该应用程序既能够在Windows NT 3.1上运行，而且能够在所有更新的版本上运行。

我问过Microsoft公司，为什么它不使用序号，我得到的回答是："我们认为可移植的可执行文件格式不仅具有序号的优点（查找迅速），而且提供了按名字输入的灵活性。我们可以随时增加函数。在带有多个实现代码的大型程序项目中，序号很难管理。"

你可以将序号用于你创建的任何DLL，并且按照序号将你的可执行文件链接到这些DLL。Microsoft保证，即使在将来的操作系统版本中，这个方法也是可行的。但是我在我的工作中总是避免使用序号，并且从现在起只按名字进行链接。

19.3.2 创建用于非Visual C++工具的DLL

如果使用Microsoft Visual C++来创建DLL和将要链接到该DLL的可执行模块，可以跳过本节内容的学习。但是，如果使用Visual C++创建DLL，而这个DLL要链接到使用任何供应商的工具创建的可执行模块，那么必须做一些额外的工作。

前面讲过当进行C和C++混合编程时使用extern "C"修改符的问题。也讲过C++类的问题以及为什么因为名字改变的缘故你必须使用同一个编译器供应商的工具的问题。当你直接将C语言编程用于多个工具供应商时将会出现另一个问题。这个问题是，即使你根本不使用C++，Microsoft的C编译器也会损害C函数。当你的函数使用__stdcall(WINAPI)调用规则时会出现这种问题。这种调用规则是最流行的一种类型。当使用__stdcall将C函数输出时，Microsoft的编译器就会改变函数的名字，设置一个前导下划线，再加上一个@符号的前缀，后随一个数字，表示作为参数传递给函数的字节数。例如，下面的函数是作为DLL的输出节中的_MyFunc@8输出的：

```
__declspec(dllexport) LONG __stdcall MyFunc(int a, int b);
```

如果用另一个供应商的工具创建了一个可执行模块，它将设法链接到一个名叫MyFunc的函数，该函数在Microsoft编译器已有的DLL中并不存在，因此链接将失败。

若要使用与其他编译器供应商的工具链接的Microsoft的工具创建一个可执行模块，必须告诉Microsoft的编译器输出没有经过改变的函数名。可以用两种方法来进行这项操作。第一种方法是为编程项目建立一个.def文件，并在该.def文件中加上类似下面的EXPORTS节：

```
EXPORTS
   MyFunc
```

当Microsoft的链接程序分析这个.def文件时，它发现_MyFunc@8和MyFunc均被输出。由于这两个函数名是互相匹配的（除了截断的尾部外），因此链接程序使用MyFunc的.def文件名来输出该函数，而根本不使用_MyFunc@8的名字来输出函数。

现在你可能认为，如果使用Microsoft的工具创建一个可执行模块，并且设法将它链接到包含未截断名字的DLL，那么链接程序的运行将会失败，因为它将试图链接到称为_MyFunc@8的函数。当然，你会高兴地了解到Microsoft的链接程序进行了正确的操作，将可执行模块链接到名字为MyFunc的函数。

China-pub.com 下载

如果想避免使用.def文件，可以使用第二种方法输出未截断的函数版本。在DLL的源代码模块中，可以添加下面这行代码：

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

这行代码使得编译器发出一个链接程序指令，告诉链接程序，一个名叫MyFunc的函数将被输出，其进入点与称为_MyFunc@8的函数的进入点相同。第二种方法没有第一种方法容易，因为你必须自己截断函数名，以便创建该代码行。另外，当使用第二种方法时，DLL实际上输出用于标识单个函数的两个符号，即MyFunc和_MyFunc@8，而第一种方法只输出符号MyFunc。第二种方法并没有给你带来更多的好处，它只是使你可以避免使用.def的文件而已。

19.4 创建可执行模块

下面的代码段显示了一个可执行的源代码文件，它输入了DLL的输出符号，并且在代码中引用了这些符号。

```
/******************************************************************************
Module:  MyExeFile1.cpp
******************************************************************************/

// Include the standard Windows and C-Runtime header files here.
#include <windows.h>


// Include the exported data structures, symbols, functions, and variables.
#include "MyLib\MyLib.h"

////////////////////////////////////////////////////////////////////////////////

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

   int nLeft = 10, nRight = 25;

   TCHAR sz[100];
   wsprintf(sz, TEXT("%d + %d = %d"), nLeft, nRight, Add(nLeft, nRight));
   MessageBox(NULL, sz, TEXT("Calculation"), MB_OK);

   wsprintf(sz, TEXT("The result from the last Add is: %d"), g_nResult);
   MessageBox(NULL, sz, TEXT("Last Result"), MB_OK);
   return(0);
}

//////////////////////////////// End of File ///////////////////////////////////
```

当创建可执行源代码文件时，必须加上DLL的头文件。如果没有头文件，输入的符号将不会被定义，而且编译器将会发出许多警告和错误消息。

可执行源代码文件不应该定义DLL的头文件前面的MYLIBAPI。当上面显示的这个可执行源代码文件被编译时，MYLIBAPI由MyLib.h头文件使用__declspec(dllimport)进行定义。当编译器看到修改变量、函数或C++类的__declspec(dllimport)时，它知道这个符号是从某个DLL模块输入的。它不知道是从哪个DLL模块输入的，并且它也不关心这个问题。编译器只想确保你用正确的方法访问这些输入的符号。现在你在源代码中可以引用输入的符号，一切都将能够正常工作。

China-pub.com
下载

接着，链接程序必须将所有 .obj模块组合起来，创建产生的可执行模块。该链接程序必须确定哪些DLL包含代码引用的所有输入符号的 DLL。因此你必须将DLL的.lib文件传递给链接程序。如前所述，.lib文件只包含DLL模块输出的符号列表。链接程序只想知道是否存在引用的符号和哪个DLL模块包含该符号。如果连接程序转换了所有外部符号的引用，那么可执行模块就因此而产生了。

输入的真正含义是什么

上一节介绍了修改符 --declspec(dllimport)。当输入一个符号时，不必使用关键字 --declspec(dllimport)，只要使用标准的C关键字 extern即可。但是，如果编译器预先知道你引用的符号将从一个DLL的.lib文件输入，那么编译器就能够生成运行效率稍高的代码。因此建议你尽量将--declspec(dllimport)关键字用于输入函数和数据符号。当你调用标准 Windows函数中的任何一个时，Microsoft将为你进行这项设置。

当链接程序进行输入符号的转换时，它就将一个称为输入节的特殊的节嵌入产生的可执行模块。输入节列出了该模块需要的DLL模块以及由每个DLL模块引用的符号。

使用Visual Studio的DumpBin.exe实用程序（带有-imports开关），能够看到模块的输入节的样子。下面是 Calc.exe文件的输入节的一个代码段（同样，我删除了 DUMPBIN的某些输出，这样它就不会占用太多的篇幅）。

```
C:\WINNT\SYSTEM32>DUMPBIN -imports Calc.EXE

Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file calc.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    SHELL32.dll
               10010F4 Import Address Table
               1012820 Import Name Table
              FFFFFFFF time date stamp
              FFFFFFFF Index of first forwarder reference

      77C42983    7A  ShellAboutW

    MSVCRT.dll
               1001094 Import Address Table
               10127C0 Import Name Table
              FFFFFFFF time date stamp
              FFFFFFFF Index of first forwarder reference

      78010040   295  memmove
      78018124    42  _EH_prolog
      78014C34   2D1  toupper
      78010F6E   2DD  wcschr
      78010668   2E3  wcslen
  :
```

China-pub.com 下载

```
ADVAPI32.dll
            1001000 Import Address Table
            101272C Import Name Table
           FFFFFFFF time date stamp
           FFFFFFFF Index of first forwarder reference

  779858F4   19A  RegQueryValueExA
  77985196   190  RegOpenKeyExA
  77984BA1   178  RegCloseKey

KERNEL32.dll
            100101C Import Address Table
            1012748 Import Name Table
            FFFFFFFF time date stamp
            FFFFFFFF Index of first forwarder reference

   77ED4134   336  lstrcpyW
   77ED33E8   1E5  LocalAlloc
   77EDEF36    DB  GetCommandLineW
   77ED1610   15E  GetProfileIntW
   77ED4BA4   1EC  LocalReAlloc

  ⋮

Header contains the following bound import information:
  Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
  Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
  Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
  Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999

Summary

      2000 .data
      3000 .rsrc
     13000 .text
```

如你所见，这一节为Calc.exe需要的每个DLL设置了一个项目，这些DLL是Shell32.dll、MSVCRt.dll、AdvAPI32.dll、Kernel32.dll、GDI32.dll和User32.dll。在每个DLL的模块名下面，有一个Calc.exe从该特定模块输入的符号列表。例如，Calc模块调用包含在Kernel32.dll中的下列函数：lstrcpyW、LocalAlloc、GetCommandLineW和GetProfileIntW等。

紧靠符号名左边的数字是符号的提示（hint）值，它与讨论无关。每个符号行最左边的数字用于指明该符号在进程的地址空间中所在的内存地址。该内存地址只有在可执行模块相链接时才出现。在DumpBin的输出的结尾处，可以看到更多的链接信息。

19.5 运行可执行模块

当一个可执行文件被启动时，操作系统加载程序将为该进程创建虚拟地址空间。然后，加载程序将可执行模块映射到进程的地址空间中。加载程序查看可执行模块的输入节，并设法找出任何需要的DLL，并将它们映射到进程的地址空间中。

China-pub.com
下载

由于该输入节只包含一个DLL名而没有它的路径名。因此加载程序必须搜索用户的磁盘驱动器，找出DLL。下面是加载程序的搜索顺序：

1) 包含可执行映像文件的目录。
2) 进程的当前目录。
3) Windows系统目录。
4) Windows目录。
5) PATH环境变量中列出的各个目录。

应该知道其他的东西也会影响加载程序对一个 DLL的搜索（详细说明参见第 20章）。当DLL模块映射到进程的地址空间中时，加载程序要检查每个 DLL的输入节。如果存在输入节（通常它确实是存在的），那么加载程序便继续将其他必要的 DLL模块映射到进程的地址空间中。加载程序将保持对DLL模块的跟踪，使模块的加载和映射只进行一次（尽管多个模块需要该模块）。

如果加载程序无法找到需要的DLL模块，用户会看到图19-2、图19-3所示的消息框中的一个：如果是Windows 2000，那么将出现图19-2所示的消息框，如果是Windows 98，则出现图19-3所示的消息框。



图19-2 Windows 2000下加载程序搜索DLL时出现的消息框

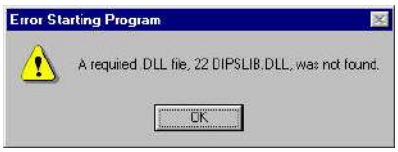


图19-3 Windows 98下加载程序搜索DLL时的消息框

当所有的DLL模块都找到并且映射到进程的地址空间中之后，加载程序就会确定对输入的符号的全部引用。为此，它要再次查看每个模块的输入节。对于列出的每个符号，加载程序都要查看指定的DLL的输出节，以确定该符号是否存在。如果该符号不存在（这种情况很少），那么加载程序就显示图 19-4、图19-5所示的消息框之一：如果是 Windows 2000，那么出现图19-4所示的消息框，如果是Windows 98，则出现图19-5所示的消息框。



图19-4 Windows 2000下加载程序查看DLL的输出节时出现的消息框

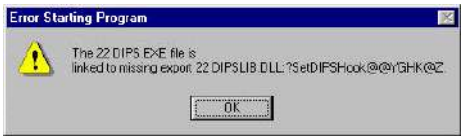


图19-5 Windows 98下加载程序查看DLL时出现的消息框

如果Windows 2000版本的消息框指明漏掉的是哪个函数，而不是显示用户难以识别的错误代码0xC000007B，那么这将是非常好的。也许下一个Windows版本能够做到这一点。

如果这个符号不存在，那么加载程序将要检索该符号的 RVA，并添加DLL模块被加载到的虚拟地址空间（符号在进程的地址空间中的位置）。然后它将该虚拟地址保存在可执行模块的输入节中。这时，当代码引用一个输入符号时，它将查看调用模块的输入节，并且捕获输入符号的地址，这样它就能够成功地访问输入变量、函数或 C++类的成员函数。好了，动态链接完成，进程的主线程开始执行，应用程序终于也开始运行了！

China-pub.com 下载

当然，这需要加载程序花费相当多的时间来加载这些 DLL模块，并用所有使用输入符号的正确地址来调整每个模块的输入节。由于所有这些工作都是在进程初始化的时候进行的，因此应用程序运行期的性能不会降低。不过，对于许多应用程序来说，初始化的速度太慢是不行的。为了缩短应用程序的加载时间，应该调整你的可执行模块和 DLL模块的位置并且将它们连接起来。真可惜很少有开发人员知道如何进行这项操作，因为这些技术是非常重要的。如果每个公司都能够使用这些技术，系统将能运行的更好。实际上，我认为操作系统销售时应该配有一个能够自动执行这些操作的实用程序。下一章将要介绍对模块调整位置和进行连接的方法。

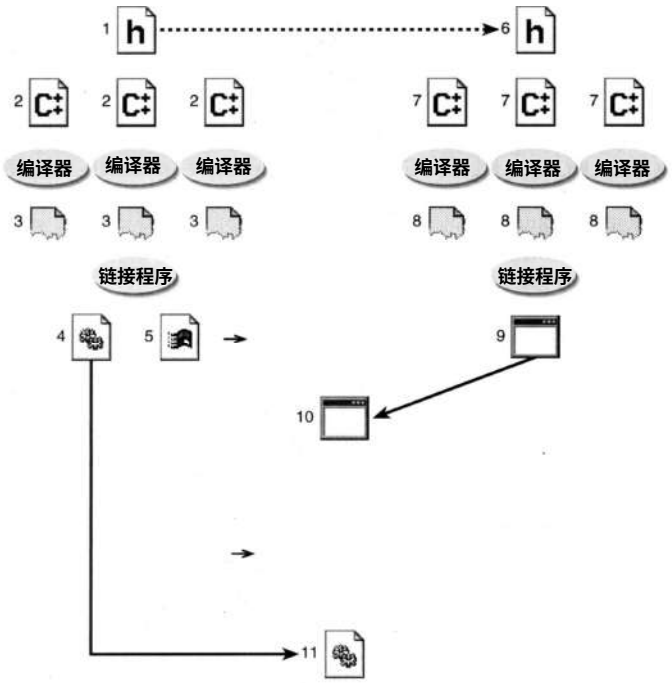
China-pub.com
下载

第20章　DLL的高级操作技术

上一章介绍了DLL链接的基本方法，并且重点说明了隐含链接的技术，这是DLL链接的最常用的形式。虽然对于大多数应用程序来说，只要了解上一章介绍的知识就足够了，但是还可以使用DLL进行更多的工作。本章将要介绍与DLL相关的各种操作方法。大多数应用程序不一定需要这些方法，但是它们是非常有用的，所以应该对它们有所了解。

20.1　DLL模块的显式加载和符号链接

如果线程需要调用DLL模块中的函数，那么DLL的文件映像必须映射到调用线程的进程地



创造DLL：
1) 建立带有输出原型/结构/符号的头文件。
2) 建立实现输出函数/变量的C/C++源文件。
3) 编译器为每个C/C++源文件生成.obj模块。
4) 链接程序将生成DLL的.obj模块链接起来。
5) 如果至少输出一个函数/变量，那么链接程序也生成.lib文件。
创造EXE：
6) 建立带有输入原型/结构/符号的头文件(视情况而定)。
7) 建立不引用输入函数/变量的C/C++源文件。
8) 编译器为每个C/C++源文件生成.obj源文件。
9) 链接程序将各个.obj模块链接起来，生成.exe文件。
注: DLL的lib文件是不需要的，因为并不直接引用输出符号。.exe文件不包含输入表。
运行应用程序：
10) 加载程序为.exe创建模块地址空进程的主线程开始执行；应用程序启动运行。
显式加载DLL：
11) 一个线程调用LoadLibrary (Ex)函数，将DLL加载到进程的地址空间这时线程可以调用GetProcAddress以便间接引用DLL的输出符号。

图20-1　应用程序创建和显式链接DLL的示意图

China-pub.com 下载

址空间中。可以用两种方法进行这项操作。第一种方法是让应用程序的源代码只引用 DLL中包含的符号。这样，当应用程序启动运行时，加载程序就能够隐含加载（和链接）需要的 DLL。

第二种方法是在应用程序运行时让应用程序显式加载需要的 DLL并且显式链接到需要的输出符号。换句话说，当应用程序运行时，它里面的线程能够决定它是否要调用 DLL中的函数。该线程可以将DLL显式加载到进程的地址空间，获得DLL中包含的函数的虚拟内存地址，然后使用该内存地址调用该函数。这种方法的优点是一切操作都是在应用程序运行时进行的。

图20-1显示了一个应用程序是如何显式地加载DLL并且链接到它里面的符号的。

20.1.1 显式加载DLL模块

无论何时，进程中的线程都可以决定将一个 DLL映射到进程的地址空间，方法是调用下面两个函数中的一个：

```
HINSTANCE LoadLibrary(PCTSTR pszDLLPathName);

HINSTANCE LoadLibraryEx(
   PCTSTR pszDLLPathName,
   HANDLE hFile,
   DWORD dwFlags);
```

这两个函数均用于找出用户系统上的文件映像（使用上一章中介绍的搜索算法），并设法将DLL的文件映像映射到调用进程的地址空间中。两个函数返回的 HINSTANCE值用于标识文件映像映射到的虚拟内存地址。如果 DLL不能被映射到进程的地址空间，则返回 NULL。若要了解关于错误的详细信息，可以调用 GetLastError.

你会注意到，LoadLibraryEx函数配有两个辅助参数，即hFile和dwFlags。参数hFile保留供将来使用，现在必须是 NULL。对于参数 dwFlags，必须将它设置为 0，或者设置为 DONT_RESOLVE_DLL_REFERENCES、LOAD_LIBRARY_AS_DATAFILE和LOAD_WITH_ALTERED_SEARCH_PATH等标志的一个组合。

1. DON T_RESOLVE_DLL_REFERENCES

DON T_RESOLVE_DLL_REFERENCES标志用于告诉系统将DLL映射到调用进程的地址空间中。通常情况下，当DLL被映射到进程的地址空间中时，系统要调用 DLL中的一个特殊函数，即 DllMain（本章后面介绍）。该函数用于对 DLL进行初始化。 DON T_RESOLVE_DLL_REFERENCES标志使系统不必调用 DllMain函数就能映射文件映像。

此外，DLL能够输入另一个DLL中包含的函数。当系统将一个 DLL映射到进程的地址空间中时，它也要查看该 DLL是否需要其他的 DLL，并且自动加载这些 DLL。当 DON T_RESOLVE_DLL_REFERENCES标志被设定时，系统并不自动将其他的 DLL加载到进程的地址空间中。

2. LOAD_LIBRARY_AS_DATAFILE

LOAD_LIBRARY_AS_DATAFILE标志与DON T_RESOLVE_DLL_REFERENCES标志相类似，因为系统只是将DLL映射到进程的地址空间中，就像它是数据文件一样。系统并不花费额外的时间来准备执行文件中的任何代码。例如，当一个 DLL被映射到进程的地址空间中时，系统要查看DLL中的某些信息，以确定应该将哪些页面保护属性赋予文件的不同的节。如果设定了LOAD_LIBRARY_AS_DATAFILE标志，系统将以它要执行文件中的代码时的同样方式来设置页面保护属性。

由于下面几个原因，该标志是非常有用的。首先，如果有一个 DLL（它只包含资源，但不

China-pub.com
下载

包含函数），那么可以设定这个标志，使DLL的文件映像能够映射到进程的地址空间中。然后可以在调用加载资源的函数时，使用LoadLibraryEx函数返回的HINSTANCE值。通常情况下，加载一个.exe文件，就能够启动一个新进程，但是也可以使用LoadLibraryEx函数将.exe文件的映像映射到进程的地址空间中。借助映射的.exe文件的HINSTANCE值，就能够访问文件中的资源。由于.exe文件没有DllMain函数，因此，当调用LoadLibraryEx来加载一个.exe文件时，必须设定LOAD_LIBRARY_AS_DATAFILE标志。

3. LOAD_WITH_ALTERED_SEARCH_PATH

LOAD_WITH_ALTERED_SEARCH_PATH标志用于改变LoadLibraryEx用来查找特定的DLL文件时使用的搜索算法。通常情况下，LoadLibraryEx按照第19章讲述的顺序进行文件的搜索。但是，如果设定了LOAD_WITH_ALTERED_SEARCH_PATH标志，那么LoadLibraryEx函数就按照下面的顺序来搜索文件：

1) pszDLLPathName参数中设定的目录。
2) 进程的当前目录。
3) Windows的系统目录。
4) Windows目录。
5) PATH环境变量中列出的目录。

20.1.2 显式卸载DLL模块

当进程中的线程不再需要DLL中的引用符号时，可以从进程的地址空间中显式卸载DLL，方法是调用下面的函数：

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

必须传递HINSTANCE值，以便标识要卸载的DLL。该值是较早的时候调用LoadLibrary(Ex)而返回的值。

也可以通过调用下面的函数从进程的地址空间中卸载DLL：

```
VOID FreeLibraryAndExitThread(
   HINSTANCE hinstDll,
   DWORD dwExitCode);
```

该函数是在Kernel32.dll中实现的，如下所示：

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll, DWORD dwExitCode) {
   FreeLibrary(hinstDll);
   ExitThread(dwExitCode);
}
```

初看起来，这并不是个非常高明的代码，你可能不明白，为什么Microsoft要创建FreeLibraryAndExitThread这个函数。其原因与下面的情况有关：假定你要编写一个DLL，当它被初次映射到进程的地址空间中时，该DLL就创建一个线程。当该线程完成它的操作时，它通过调用FreeLibrary函数，从进程的地址空间中卸载该DLL，并且终止运行，然后立即调用ExitThread。

但是，如果线程分开调用FreeLibrary和ExitThread，就会出现一个严重的问题。这个问题是调用FreeLibrary会立即从进程的地址空间中卸载DLL。当调用的FreeLibrary返回时，包含对ExitThread调用的代码就不再可以使用，因此线程将无法执行任何代码。这将导致访问违规，同时整个进程终止运行。

但是，如果线程调用FreeLibraryAndExitThread，该函数调用FreeLibrary，使DLL立即被卸

China-pub.com 下载

载。下一个执行的指令是在Kernel32.dll中，而不是在刚刚被卸载的DLL中。这意味着该线程能够继续执行，并且可以调用ExitThread。ExitThread使该线程终止运行并且不返回。

一般来说，并没有很大的必要去调用FreeLibraryAndExitThread函数。我曾经使用过一次，因为我执行了一个非常特殊的任务。另外，我为Microsoft Windows 3.1编写了一个代码，它并没有提供这个函数。因此我高兴地看到Microsoft将这个函数增加到了较新的Windows版本中。

在实际环境中，LoadLibrary和LoadLibraryEx这两个函数用于对与特定的库相关的进程使用计数进行递增，FreeLibrary和FreeLibraryAndExitThread这两个函数则用于对库的每个进程的使用计数进行递减。例如，当第一次调用LoadLibrary函数来加载DLL时，系统将DLL的文件映像映射到调用进程的地址空间中，并将DLL的使用计数设置为1。如果同一个进程中的线程后来调用LoadLibrary来加载同一个DLL文件映像，系统并不第二次将DLL映像文件映射到进程的地址空间中，它只是将与该进程的DLL相关的使用计数递增1。

为了从进程的地址空间中卸载DLL文件映像，进程中的线程必须两次调用FreeLibrary函数。第一次调用只是将DLL的使用计数递减为1，第二次调用则将DLL的使用计数递减为0。当系统发现DLL的使用计数递减为0时，它就从进程的地址空间中卸载DLL的文件映像。试图调用DLL中的函数的任何线程都会产生访问违规，因为特定地址上的代码不再被映射到进程的地址空间中。

系统为每个进程维护了一个DLL的使用计数，也就是说，如果进程A中的一个线程调用下面的函数，然后进程B中的一个线程调用相同的函数，那么MyLib.dll将被映射到两个进程的地址空间中，这样，进程A和进程B的DLL使用计数都将是1。

```
HINSTANCE hinstDll = LoadLibrary("MyLib.dll");
```

如果进程B中的线程后来调用下面的函数，那么进程B的DLL使用计数将变成0，并且该DLL将从进程B的地址空间中卸载。但是，进程A的地址空间中的DLL映射不会受到影响，进程A的DLL使用计数仍然是1。

```
FreeLibrary(hinstDll);
```

如果调用GetModuleHandle函数，线程就能够确定DLL是否已经被映射到进程的地址空间中：

```
HINSTANCE GetModuleHandle(PCTSTR pszModuleName);
```

例如，只有当MyLib.dll尚未被映射到进程的地址空间中时，下面这个代码才能加载该文件：

```
HINSTANCE hinstDll = GetModuleHandle("MyLib"); // DLL extension assumed
if (hinstDll == NULL) {
   hinstDll = LoadLibrary("MyLib"); // DLL extension assumed
}
```

如果只有DLL的HINSTANCE值，那么可以调用GetModuleFileName函数，确定DLL（或.exe）的全路径名：

```
DWORD GetModuleFileName(
   HINSTANCE hinstModule,
   PTSTR pszPathName,
   DWORD cchPath);
```

第一个参数是DLL（或.exe）的HINSTANCE。第二个参数pszPathName是该函数将文件映像的全路径名放入的缓存的地址。第三参数cchPath用于设定缓存的大小（以字符为计量单位）。

20.1.3 显式链接到一个输出符号

一旦DLL模块被显式加载，线程就必须获取它要引用的符号的地址，方法是调用下面的函数：

China-pub.com
下载


```
FARPROC GetProcAddress(
   HINSTANCE hinstDll,
   PCSTR pszSymbolName);
```

参数hinstDll是调用LoadLibrary(Ex)或GetModuleHandle函数而返回的，它用于设定包含符号的DLL的句柄。参数pszSymbolName可以采用两种形式。第一种形式是以0结尾的字符串的地址，它包含了你想要其地址的符号的名字：

```
FARPROC pfn = GetProcAddress(hinstDll, "SomeFuncInDll");
```

注意，参数pszSymbolName的原型是PCSTR，而不是PCTSTR。这意味着GetProcAddress函数只接受ANSI字符串，决不能将Unicode字符串传递给该函数，因为编译器/链接程序总是将符号名作为ANSI字符串存储在DLL的输出节中。

参数pszSymbolName的第二种形式用于指明你想要其地址的符号的序号：

```
FARPROC pfn = GetProcAddress(hinstDll, MAKEINTRESOURCE(2));
```

这种用法假设你知道你需要的符号名被DLL创建程序赋予了序号值2。同样，我要再次强调，Microsoft非常反对使用序号，因此你不会经常看到GetProcAddress的这个用法。

这两种方法都能够提供包含在DLL中的必要符号的地址。如果DLL模块的输出节中不存在你需要的符号，GetProcAddress就返回NULL，表示运行失败。

应该知道，调用GetProcAddress的第一种方法比第二种方法要慢，因为系统必须进行字符串的比较，并且要搜索传递的符号名字符串。对于第二种方法来说，如果传递的序号尚未被分配给任何输出的函数，那么GetProcAddress就会返回一个非NULL值。这个返回值将会使你的应用程序错误地认为你已经拥有一个有效的地址，而实际上你并不拥有这样的地址。如果试图调用该地址，肯定会导致线程引发一个访问违规。我在早期从事Windows编程时，并不完全理解这个行为特性，因此多次出现这样的错误。所以一定要小心（这个行为特性是应该避免使用序号而使用符号名的另一个原因）。

20.2 DLL的进入点函数

一个DLL可以拥有单个进入点函数。系统在不同的时间调用这个进入点函数，这个问题将在下面加以介绍。这些调用可以用来提供一些信息，通常用于供DLL进行每个进程或线程的初始化和清除操作。如果你的DLL不需要这些通知信息，就不必在DLL源代码中实现这个函数。例如，如果你创建一个只包含资源的DLL，就不必实现该函数。如果确实需要在DLL中接受通知信息，可以实现类似下面的进入点函数：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   switch (fdwReason) {
      case DLL_PROCESS_ATTACH:
         // The DLL is being mapped into the process's address space.
         break;

      case DLL_THREAD_ATTACH:
         // A thread is being created.
         break;

      case DLL_THREAD_DETACH:
         // A thread is exiting cleanly.
         break;
```

China-pub.com 下载

```
      case DLL_PROCESS_DETACH:
         // The DLL is being unmapped from the process's address space.
         break;
   }
   return(TRUE);  // Used only for DLL_PROCESS_ATTACH
}
```

注意 函数名DllMain是区分大小写的。许多编程人员有时调用的函数是 DLLMain。这是一个非常容易犯的错误，因为DLL这个词常常使用大写来表示。如果调用的进入点函数不是DllMain，而是别的函数，你的代码将能够编译和链接，但是你的进入点函数永远不会被调用，你的DLL永远不会被初始化。

参数hinstDll包含了DLL的实例句柄。与(w)WinMain函数的hinstExe参数一样，这个值用于标识DLL的文件映像被映射到进程的地址空间中的虚拟内存地址。通常应将这个参数保存在一个全局变量中，这样就可以在调用加载资源的函数（如 DialogBox和LoadString）时使用它。最后一个参数是fImpLoad，如果DLL是隐含加载的，那么该参数将是个非 0值，如果DLL是显式加载的，那么它的值是0。

参数fdwReason用于指明系统为什么调用该函数。该参数可以使用 4个值中的一个。这4个值是：DLL_PROCESS_ATTACH、DLL_PROCESS_DETACH、DLL_THREAD_ATTACH或DLL_THREAD_DETACH。这些值将在下面介绍。

注意 必须记住，DLL使用DllMain函数来对它们进行初始化。当你的DllMain函数执行时，同一个地址空间中的其他DLL可能尚未执行它们的DllMain函数。这意味着它们尚未初始化，因此你应该避免调用从其他DLL中输入的函数。此外，你应该避免从DllMain内部调用LoadLibrary(Ex)和FreeLibrary函数，因为这些函数会形式一个依赖性循环。

Platform SDK文档说，你的DllMain函数只应该进行一些简单的初始化，比如设置本地存储器（第21章介绍），创建内核对象和打开文件等。你还必须避免调用 User、Shell、ODBC、COM、RPC和套接字函数（即调用这些函数的函数），因为它们的DLL也许尚未初始化 ，或者这些函数可能在内部调用LoadLibrary(Ex)函数，这同样会形成一个依赖性循环。

另外，如果创建全局性的或静态的C++对象，那么应该注意可能存在同样的问题，因为在你调用DllMain函数的同时，这些对象的构造函数和析构函数也会被调用。

20.2.1 DLL_PROCESS_ATTACH通知

当DLL被初次映射到进程的地址空间中时，系统将调用该 DLL的DllMain函数，给它传递参数fdwReason的值DLL_PROCESS_ATTACH。只有当DLL的文件映像初次被映射时，才会出现这种情况。如果线程在后来为已经映射到进程的地址空间中的 DLL调用LoadLibrary(Ex)函数，那么操作系统只是递增 DLL的使用计数，它并不再次用 DLL_PROCESS_ATTACH的值来调用DLL的DllMain函数。

当处理DLL_PROCESS_ATTACH时，DLL应该执行DLL中的函数要求的任何与进程相关的初始化。例如，DLL可能包含需要使用它们自己的堆栈（在进程的地址空间中创建）的函数。通过在处理DLL_PROCESS_ATTACH通知时调用HeapCreate函数，该DLL的DllMain函数就能够创建这个堆栈。已经创建的堆栈的句柄可以保存在 DLL函数有权访问的一个全局变量中。

当DllMain处理一个DLL_PROCESS_ATTACH通知时，DllMain的返回值能够指明DLL的初

China-pub.com
下载

始化是否已经取得成功。如果对HeapCreate的调用取得了成功，DllMain应该返回TRUE。如果堆栈不能创建，它应该返回 FALSE。如果fdwReason使用的是其他的值，即 DLL_PROCESS_DETACH、DLL_THREAD_ATTACH和DLL_THREAD_DETACH，那么系统将忽略DllMain返回的值。

当然，系统中的有些线程必须负责执行DllMain函数中的代码。当一个新线程创建时，系统将分配进程的地址空间，然后将.exe文件映像和所有需要的DLL文件映像映射到进程的地址空间中。然后它创建进程的主线程，并使用该线程调用每个 DLL的带有DLL_PROCESS_ATTACH 值的DllMain函数。当已经映射的所有DLL都对通知信息作出响应后，系统将使进程的主线程开始执行可执行模块的C/C++运行期启动代码，然后执行可执行模块的进入点函数(main、wmain、WinMain或wWinMain)。如果DLL的任何一个DllMain函数返回FALSE，指明初始化没有取得成功，系统便终止整个进程的运行，从它的地址空间中删除所有文件映像，给用户显示一个消息框，说明进程无法启动运行。 Windows 2000的这个消息框如 图20-2所示，再下面是Windows 98的消息框（见图20-3）。



图20-2 Windows 2000下显示的消息框



图20-3 Windows 98下显示的消息框

下面让我们来看一看DLL被显式加载时的情况。当进程中的一个线程调用LoadLibrary(Ex)时，系统会找出特定的 DLL，并将它映射到进程的地址空间中。然后，系统使用调用LoadLibrary(Ex)的线程，调用DLL的带有DLL_PROCESS_ATTACH 值的DllMain函数。当DLL的DllMain函数处理了通知消息后，系统便允许调用的LoadLibrary(Ex)函数返回，同时该线程像平常一样继续进行处理。如果DllMain函数返回FALSE，指明初始化没有取得成功，那么系统就自动从进程的地址空间中卸载DLL的文件映像，而对LoadLibrary(Ex)的调用则返回NULL。

20.2.2 DLL_PROCESS_DETACH通知

DLL从进程的地址空间中被卸载时，系统将调用DLL的DllMain函数，给它传递fdwReason的值DLL_PROCESS_DETACH。当DLL处理这个值时，它应该执行任何与进程相关的清除操作。例如，DLL可以调用HeapDestroy函数来撤消它在DLL_PROCESS_DETACH通知期间创建的堆栈。注意，如果DllMain函数接收到DLL_PROCESS_DETACH通知时返回FALSE，那么DllMain就不是用DLL_PROCESS_DETACH通知调用的。如果因为进程终止运行而使DLL被卸载，那么调用ExitProcess函数的线程将负责执行DllMain函数的代码。在正常情况下，这是应用程序的主线程。当你的进入点函数返回到C/C++运行期库的启动代码时，该启动代码将显式调用ExitProcess函数，终止进程的运行。

如果因为进程中的线程调用FreeLibrary或FreeLibraryAndExitThread函数而将DLL卸载，那么调用函数的线程将负责执行DllMain函数的代码。如果使用FreeLibrary，那么要等到DllMain函数完成对DLL_PROCESS_DETACH通知的执行后，该线程才从对FreeLibrary函数的调用中返回。

注意，DLL能够阻止进程终止运行。例如，当DllMain接收到DLL_PROCESS_DETACH通知时，它就会进入一个无限循环。只有当每个DLL都已完成对DLL_PROCESS_DETACH通知

China-pub.com 下载

的处理时，操作系统才会终止该进程的运行。

注意 如果因为系统中的某个线程调用了TerminateProcess而使进程终止运行，那么系统将不调用带有DLL_PROCESS_DETACH值的 DLL的DllMain函数。这意味着映射到进程的地址空间中的任何DLL都没有机会在进程终止运行之前执行任何清除操作。这可能导致数据的丢失。只有在迫不得已的情况下，才能使用 TerminateProcess函数。

图20-4显示了线程调用LoadLibrary时执行的操作步骤。图20-5显示了线程调用FreeLibrary函数时执行的操作步骤。

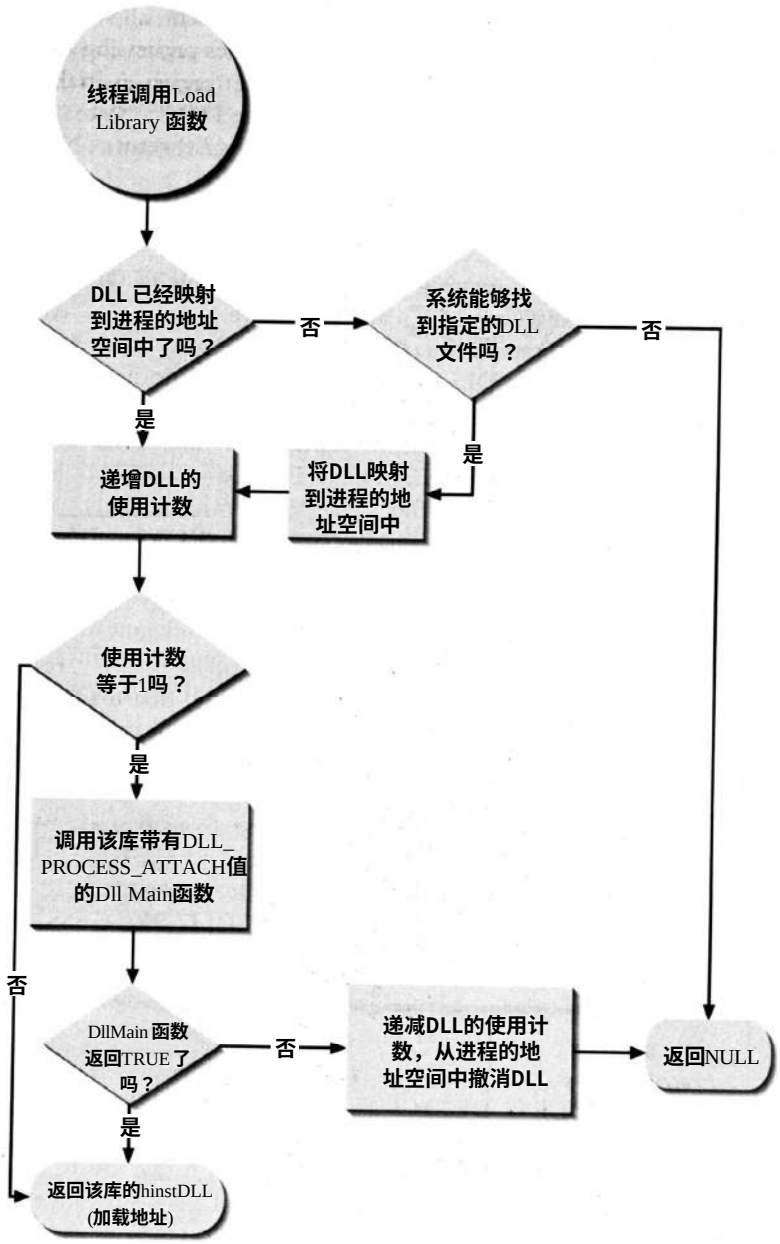


图20-4 线程调用LoadLibrary时系统执行的操作步骤

China-pub.com
下载

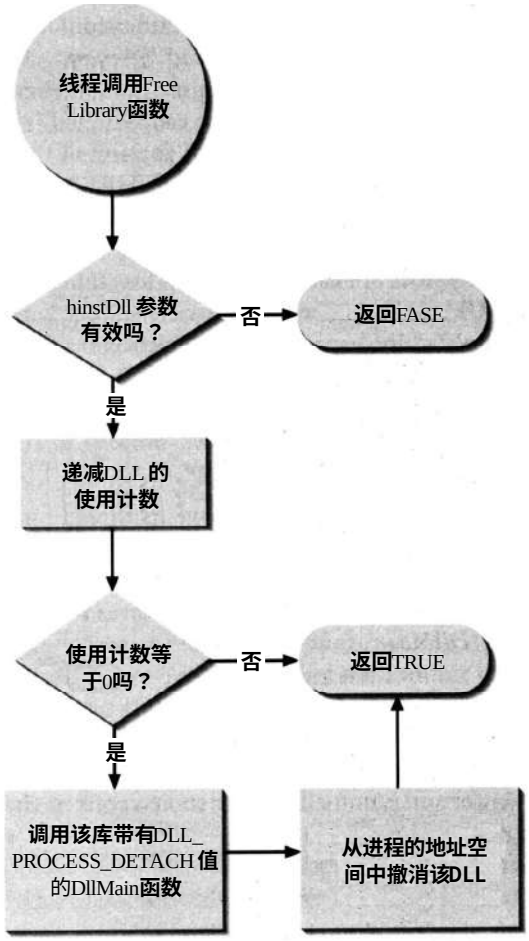


图20-5 线程调用FreeLibrary时系统执行的操作步骤

20.2.3 DLL_THREAD_ATTACH通知

当在一个进程中创建线程时，系统要查看当前映射到该进程的地址空间中的所有 DLL文件映像，并调用每个文件映像的带有 DLL_THREAD_ATTACH值的DllMain函数。这可以告诉所有的DLL执行每个线程的初始化操作。新创建的线程负责执行 DLL的所有DllMain函数中的代码。只有当所有的DLL都有机会处理该通知时，系统才允许新线程开始执行它的线程函数。

当一个新DLL被映射到进程的地址空间中时，如果该进程内已经有若干个线程正在运行，那么系统将不为现有的线程调用带有 DLL_THREAD_ATTACH值的DDL 的DllMain函数。只有当新线程创建时DLL被映射到进程的地址空间中，它才调用带有 DLL_THREAD_ATTACH值的DLL的DllMain函数。

另外要注意，系统并不为进程的主线程调用带有 DLL_THREAD_ATTACH值的任何DllMain函数。进程初次启动时映射到进程的地址空间中的任何 DLL均接收 DLL_PROCESS_ATTACH通知，而不是DLL_THREAD_ATTACH通知。

20.2.4 DLL_THREAD_DETACH通知

让线程终止运行的首选方法是使它的线程函数返回。这使得系统可以调用 ExitThread来撤

China-pub.com 下载

消该线程。ExitThread函数告诉系统，该线程想要终止运行，但是系统并不立即将它撤消。相反，它要取出这个即将被撤消的线程，并让它调用已经映射的DLL的所有带有DLL_THREAD_DETACH值的DllMain函数。这个通知告诉所有的DLL执行每个线程的清除操作。例如，DLL版本的C/C++运行期库能够释放它用于管理多线程应用程序的数据块。

注意，DLL能够防止线程终止运行。例如，当DllMain函数接收到DLL_THREAD_DETACH通知时，它就能够进入一个无限循环。只有当每个DLL已经完成对DLL_THREAD_DETACH通知的处理时，操作系统才会终止线程的运行。

> **注意** 如果因为系统中的线程调用TerminateThread函数而使该线程终止运行，那么系统将不调用带有DLL_THREAD_DETACH值的DLL的所有DllMain函数。这意味着映射到进程的地址空间中的任何一个DLL都没有机会在线程终止运行之前执行任何清除操作。这可能导致数据的丢失。与TerminateProcess一样，只有在迫不得已的时候，才可以使用TerminateThread函数。

如果当DLL被撤消时仍然有线程在运行，那么就不为任何线程调用带有DLL_THREAD_DETACH值的DllMain。可以在进行DLL_THREAD_DETACH的处理时查看这个情况，这样就能够执行必要的清除操作。

上述规则可能导致发生下面这种情况。当进程中的一个线程调用LoadLibrary来加载DLL时，系统就会调用带有DLL_PROCESS_ATTACH值的DLL的DllMain函数（注意，没有为该线程发送DLL_THREAD_ATTACH通知）。接着，负责加载DLL的线程退出，从而导致DLL的DllMain函数被再次调用，这次调用时带有DLL_THREAD_DETACH值。注意，DLL得到通知说，该线程将被撤消，尽管它从未收到DLL_THREAD_ATTACH的这个通知，这个通知告诉该库说线程已经附加。由于这个原因，当执行任何特定的线程清除操作时，必须非常小心。不过大多数程序在编写时就规定调用LoadLibrary的线程与调用FreeLibrary的线程是同一个线程。

20.2.5 顺序调用DllMain

系统是顺序调用DLL的DllMain函数的。为了理解这样做的意义，可以考虑下面这样一个环境。假设一个进程有两个线程，线程A和线程B。该进程还有一个DLL，称为SomeDLL.dll，它被映射到了它的地址空间中。两个线程都准备调用CreateThread函数，以便再创建两个线程，即线程C和线程D。

当线程A调用CreateThread来创建线程C时，系统调用带有DLL_THREAD_ATTACH值的SomeDLL.dll的DllMain函数。当线程C执行DllMain函数中的代码时，线程B调用CreateThread函数来创建线程D。这时系统必须再次调用带有DLL_THREAD_ATTACH值的DllMain函数，这次是让线程D执行代码。但是，系统是顺序调用DllMain函数的，因此系统会暂停线程D的运行，直到线程C完成对DllMain函数中的代码的处理并且返回为止。

当线程C完成DllMain的处理后，它就开始执行它的线程函数。这时系统唤醒线程D，让它处理DllMain中的代码。当它返回时，线程D开始处理它的线程函数。

通常情况下，根本不会考虑到DllMain的这个顺序操作特性。我曾经遇到过一个人，他的代码中有一个DllMain顺序操作带来的错误。他创建的代码类似下面的样子：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   HANDLE hThread;
   DWORD dwThreadId;
```

China-pub.com
下载


```
   switch (fdwReason) {
   case DLL_PROCESS_ATTACH:
      // The DLL is being mapped into the process's address space.

      // Create a thread to do some stuff.
      hThread = CreateThread(NULL, 0, SomeFunction, NULL,
         0, &dwThreadId);

      // Suspend our thread until the new thread terminates.
      WaitForSingleObject(hThread, INFINITE);

      // We no longer need access to the new thread.
      CloseHandle(hThread);
      break;

   case DLL_THREAD_ATTACH:
      // A thread is being created.
      break;

   case DLL_THREAD_DETACH:
      // A thread is exiting cleanly.
      break;

   case DLL_PROCESS_DETACH:
      // The DLL is being unmapped from the process's address space.
      break;
   }
   return(TRUE);
}
```

我们花了好几个小时才发现这个代码中存在的问题。你能够看出这个问题吗？当DllMain收到DLL_PROCESS_ATTACH通知时，一个新线程就创建了。系统必须用DLL_THREAD_ATTACH的值再次调用DllMain函数。但是，新线程被暂停运行，因为导致DLL_PROCESS_ATTACH被发送给DllMain函数的线程尚未完成处理操作。问题是调用WaitForSingleObject函数而产生的。这个函数使当前正在运行的线程暂停运行，直到新线程终止运行。但是新线程从未得到机会运行，更不要说终止运行，因为它处于暂停状态，等待当前线程退出DllMain函数。这里我们得到的是个死锁条件。两个线程将永远处于暂停状态。

当我刚刚开始考虑如何解决这个问题的时候，我发现了DisableThreadLibraryCalls函数：

```
BOOL DisableThreadLibraryCalls(HINSTANCE hinstDll);
```

DisableThreadLibraryCalls告诉系统说，你不想将DLL_THREAD_ATTACH和DLL_THREAD_DETACH通知发送给特定的DLL的DllMain函数。我认为这样做是有道理的，如果我们告诉系统不要将DLL通知发送给DLL，那么就不会发送死锁条件。但是当我测试解决方案时，我很快发现它解决不了问题。请看下面的代码：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   HANDLE hThread;
   DWORD dwThreadId;

   switch (fdwReason) {
   case DLL_PROCESS_ATTACH:
      // The DLL is being mapped into the process's address space.
```

China-pub.com 下载

```
            // Prevent the system from calling DllMain
            // when threads are created or destroyed.
            DisableThreadLibraryCalls(hinstDll);

            // Create a thread to do some stuff.
            hThread = CreateThread(NULL, 0, SomeFunction, NULL,
               0, &dwThreadId);

            // Suspend our thread until the new thread terminates.
            WaitForSingleObject(hThread, INFINITE);

            // We no longer need access to the new thread.
            CloseHandle(hThread);
            break;

         case DLL_THREAD_ATTACH:
            // A thread is being created.
            break;

         case DLL_THREAD_DETACH:
            // A thread is exiting cleanly.
            break;

         case DLL_PROCESS_DETACH:
            // The DLL is being unmapped from the process's address space.
            break;
         }
         return(TRUE);
      }
```

通过进一步的研究，我终于发现了问题。当进程被创建时，系统也创建一个互斥对象。每个进程都有它自己的互斥对象，也就是说多个进程并不共享互斥对象。当线程调用映射到进程的地址空间中的DLL的DllMain函数时，这个互斥对象负责对进程的所有线程实施同步。

当CreateThread函数被调用时，系统首先创建线程的内核对象和线程的堆栈。然后它在内部调用WaitForSingleObject函数，传递进程的互斥对象的句柄。一旦新线程拥有该互斥对象，系统就让新线程用DLL_THREAD_ATTACH的值调用每个DLL的DllMain函数。只有在这个时候，系统才调用ReleaseMutex，释放对进程的互斥对象的所有权。由于系统采用这种方式来运行，因此添加对DisableThreadLibraryCalls的调用，并不会防止线程被暂停运行。防止线程被暂停运行的唯一办法是重新设计这部分源代码，使得 WaitForSingleObject不会在任何DLL的DllMain函数中被调用。

20.2.6 DllMain与C/C++运行期库

在上面介绍的DllMain函数中，我假设你使用Microsoft的Visual C++编译器来创建你的DLL。当编写一个DLL时，你需要得到C/C++运行期库的某些初始帮助。例如，如果你创建的 DLL包含一个全局变量，而这个全局变量是个 C++类的实例。在你顺利地在 DllMain函数中使用这个全局变量之前，该变量必须调用它的构造函数。这是由 C/C++运行期库的DLL启动代码来完成的。

当你链接你的DLL时，链接程序将DLL的进入点函数嵌入产生的DLL文件映像。可以使用链接程序的/ENTRY开关来设定该函数的地址。按照默认设置，当使用 Microsoft的链接程序并

China-pub.com
下载

且设定/DLL开关时，链接程序假设进入点函数称为 _DllMainCRTStartup。该函数包含在C/C++运行期的库文件中，并且在你链接DLL时它被静态链接到你的DLL文件的映像中（即使你使用DLL版本的C/C++运行期库，该函数也是静态链接的）。

当你的DLL文件映像被映射到进程的地址空间中时，系统实际上是调用 _DllMainCRTStartup函数，而不是调用DllMain函数。_DllMainCRTStartup函数负责对C/C++运行期库进行初始化，并且确保在_DllMainCRTStartup收到DLL_PROCESS_ATTACH通知时创建任何全局或静态C++对象。当执行任何 C/C++运行期初始化时， _DllMainCRTStartup函数将调用你的DllMain函数。

当DLL收到DLL_PROCESS_DETACH通知时，系统再次调用_DllMainCRTStartup函数。这次该函数调用你的DllMain函数，当DllMain返回时，_DllMainCRTStartup就为DLL中的任何全局或静态C++对象调用析构函数。当_DllMainCRTStartup收到DLL_THREAD_ATTACH通知时，_DllMainCRTStartup函数并不执行任何特殊的处理操作。但是对于 DLL_THREAD_DETACH来说，C/C++运行期将释放线程的tiddata内存块（如果存在这样的内存块的话）。但是，通常情况下，这个 tiddata内存块是不应该存在的，因为编写正确的线程函数将返回到内部调用 _endthreadex的C/C++运行期的 _threadstartex函数（第6章已经介绍），它负责在线程试图调用ExitThread之前释放内存块。

然而，让我们看一看这样一种情况，即用 Pascal编写的应用程序调用 DLL中用C/C++编写的函数。在这种情况下， Pascal应用程序创建了一个线程，并且不使用 _beginthreadex。因此线程对C/C++运行期库的情况一无所知。这时线程调用 DLL中的一个函数，该函数又调用一个 C运行期函数。当你再次调用该函数时， C运行期函数为该线程创建一个 tiddata内存块，并且在创建过程中将它与线程关联起来。这意味着 Pascal应用程序能够创建成功地调用 C运行期函数的线程。当用Pascal编写的线程函数返回时， ExitThread被调用。C/C++运行期库的DLL收到DLL_THREAD_DETACH通知，并释放tiddata内存块，这样就不会出现任何内存泄漏。这确实是个非常出色的思路。

前面讲过，不必在DLL源代码中实现DllMain函数。如果你并不拥有自己的 DllMain函数，可以使用 C/C++运行期库的 DllMain函数的实现代码，它类似下面的形式（如果静态链接到C/C++运行期库的话）：

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   if (fdwReason == DLL_PROCESS_ATTACH)
      DisableThreadLibraryCalls(hinstDll);
   return(TRUE);
}
```

当链接程序链接DLL时，如果链接程序无法找到DLL的.obj文件中的DllMain函数，那么它就链接C/C++运行期库的 DllMain函数的实现代码。如果你没有提供自己的 DllMain函数，C/C++运行期库就正确地假设你不在乎 DLL_THREAD_ATTACH和DLL_THREAD_DETACH通知。为了提高创建和撤消线程的性能，则调用 DisableThreadLibraryCalls函数。

20.3 延迟加载DLL

Microsoft Visual C++ 6.0提供了一个出色的新特性，它能够使 DLL的操作变得更加容易。这个特性称为延迟加载 DLL。延迟加载的 DLL是个隐含链接的 DLL，它实际上要等到你的代码试图引用 DLL中包含的一个符号时才进行加载。延迟加载的 DLL在下列情况下是

China-pub.com 下载

非常有用的：

- 如果你的应用程序使用若干个DLL，那么它的初始化时间就比较长，因为加载程序要将所有需要的DLL映射到进程的地址空间中。解决这个问题的方法之一是在进程运行的时候分开加载各个DLL。延迟加载的DLL能够更容易地完成这样的加载。
- 如果调用代码中的一个新函数，然后试图在老版本的系统上运行你的应用程序，而该系统中没有该函数，那么加载程序就会报告一个错误，并且不允许该应用程序运行。你需要一种方法让你的应用程序运行，然后，如果（在运行时）发现该应用程序在老的系统上运行，那么你将不调用遗漏的函数。例如，一个应用程序在Windows 2000上运行时想要使用PSAPI函数，而在Windows 98上运行想要使用ToolHelp函数（比如Process32Next）。当该应用程序初始化时，它调用GetVersionEx函数来确定主操作系统，并正确地调用相应的其他函数。如果试图在Windows 98上运行该应用程序，就会导致加载程序显示一条错误消息，因为Windows 98上并不存在PSAPI.dll模块。同样，延迟加载的DLL能够使你非常容易地解决这个问题。

我花费了相当多的时间来检验Visual C++ 6.0中的延迟加载DLL特性，必须承认，Microsoft在实现这个特性方面做了非常出色的工作。它提供了许多特性，并且在Windows 98和Windows 2000上运行得都很好。

下面让我们从比较容易的操作开始介绍，也就是使延迟加载DLL能够运行。首先，你象平常那样创建一个DLL。也要象平常那样创建一个可执行模块，但是必须修改两个链接程序开关，并且重新链接可执行模块。下面是需要添加的两个链接程序开关：

```
/Lib:DelayImp.lib
/DelayLoad:MyDll.dll
```

Lib开关告诉链接程序将一个特殊的函数--delayLoadHelper嵌入你的可执行模块。第二个开关将下列事情告诉链接程序：

- 从可执行模块的输入节中删除MyDll.dll，这样，当进程被初始化时，操作系统的加载程序就不会显式加载DLL。
- 将新的Delay Import（延迟输入）节（称为.didata）嵌入可执行模块，以指明哪些函数正在从MyDll.dll输入。
- 通过转移到对--delayLoadHelper函数的调用，转换到对延迟加载函数的调用。

当应用程序运行时，对延迟加载函数的调用实际上是对--delayLoadHelper函数的调用。该函数引用特殊的Delay Import节，并且知道调用LoadLibrary之后再调用GetProcAddress。一旦获得延迟加载函数的地址，--delayLoadHelper就要安排好对该函数的调用，这样，将来的调用就会直接转向对延迟加载函数的调用。注意，当第一次调用同一个DLL中的其他函数时，必须对它们做好安排。另外，可以多次设定/delayLoad链接程序的开关，为想要延迟加载的每个DLL设定一次开关。

好了，整个操作过程就这么简单。但是还应该考虑另外两个问题。通常情况下，当操作系统的加载程序加载可执行模块时，它将设法加载必要的DLL。如果一个DLL无法加载，那么加载程序就会显示一条错误消息。如果是延迟加载的DLL，那么在进行初始化时将不检查是否存在DLL。如果调用延迟加载函数时无法找到该DLL，--delayLoadHelper函数就会引发一个软件异常条件。可以使用结构化异常处理（SEH）方法来跟踪该异常条件。如果不跟踪该异常条件，那么你的进程就会终止运行（SEH将在第23、24和25章中介绍）。

当--delayLoadHelper确实找到你的DLL，但是要调用的函数不在该DLL中时，将会出现另

China-pub.com
下载

一个问题。比如，如果加载程序找到一个老的 DLL版本，就会发生这种情况。在这种情况下，--delayLoadHelper也会引发一个软件异常条件，对这个软件异常条件的处理方法与上面相同。下一节介绍的示例应用程序显示了如何正确地编写SEH代码以便处理这些错误。

你会发现代码中有许多其他元素，这些元素与SEH和错误处理毫无关系。但是这些元素与你使用延迟加载的DLL时可以使用的辅助特性有关。下面将要介绍这些特性。如果你不使用更多的高级特性，可以删除这些额外的代码。

如你所见，Visual C++开发小组定义了两个软件异常条件代码，即 VcppException (ERROR_SEVERITY_ERROR、ERROR_MOD_NOT_FOUND)和VcppException（ERROR_SEVERITY_ERROR、ERROR_PROC_NOT_FOUND）。这些代码分别用于指明DLL模块没有找到和函数没有找到。我的异常过滤函数 DelayLoadDllExceptionFilter用于查找这两个异常代码。如果两个代码都没有找到，过滤函数将返回 EXCEPTION_CONTINUE_SEARCH，这与任何出色的过滤函数返回的值是一样的（对于你不知道如何处理的异常代码，请不要随意删除）。但是如果这两个代码中的一个已经找到，那么 --delayLoadHelper函数将提供一个指向包含某些辅助信息的DelayLoadInfo结构的指针。在Visual C++的DelayImp.h文件中，DelayLoadInfo结构定义为下面的形式：

```
typedef struct DelayLoadInfo {
   DWORD            cb;          // Size of structure
   PCImgDelayDescr  pidd;        // Raw data (everything is there)
   FARPROC *        ppfn;        // Points to address of function to load
   LPCSTR           szDll;       // Name of dll
   DelayLoadProc    dlp;         // Name or ordinal of procedure
   HMODULE          hmodCur;     // hInstance of loaded library
   FARPROC          pfnCur;      // Actual function that will be called
   DWORD            dwLastError;// Error received
} DelayLoadInfo, * PDelayLoadInfo;
```

这个数据结构是由 --delayLoadHelper函数来分配和初始化的。在该函数按步骤动态加载DLL并且获得被调用函数的地址的过程中，它将填写该结构的各个成员。在 SEH结构的内部，成员szDll指向你要加载的DLL的名字，想要查看的函数则在成员 dlp中。由于可以按序号或名字来查看各个函数，因此dlp成员类似下面的样子：

```
typedef struct DelayLoadProc {
   BOOL fImportByName;
   union {
      LPCSTR szProcName;
      DWORD  dwOrdinal;
   };
} DelayLoadProc;
```

如果DLL已经加载成功，但是它不包含必要的函数，也可以查看成员 hmodCur，以了解DLL被加载到的内存地址。也可以查看成员dwLastError，以了解是什么错误导致了异常条件的引发。不过对于异常过滤函数来说，这是不必要的，因为异常代码能够告诉你究竟发生了什么问题。成员 pfnCur包含了需要的函数的地址。在过滤函数中它总是置为 NULL，因为 --delayLoadHelper无法找到该函数的地址。

在其余的成员中，cb用于确定版本，pidd指向嵌入模块中包含延迟加载的DLL和函数的节，ppfn是函数找到时，函数的地址应该放入的地址。最后两个成员供 --delayLoadHelper函数内部使用。它们有着超高级的用途，现在还没有必要观察或者了解这两个成员。

China-pub.com 下载

到现在为止，已经讲述了如何使用延迟加载的DLL和正确解决错误条件的基本方法。但是Microsoft的延迟加载DLL的实现代码超出了迄今为止我已讲述的内容范围。比如，你的应用程序能够卸载延迟加载的DLL。假如你的应用程序需要一个特殊的DLL来打印一个文档，那么这个DLL就非常适合作为一个延迟加载的DLL，因为大部分时间它是不用的。不过，如果用户选择了Print命令，你就可以调用该DLL中的一个函数，然后它就能够自动进行DLL的加载。这确实很好，但是，当文档打印后，用户可能不会立即打印另一个文档，因此可以卸载这个 DLL，释放系统的资源。如果用户决定打印另一个文档，那么DLL就可以根据用户的要求再次加载。

若要卸载延迟加载的DLL，必须执行两项操作。首先，当创建可执行文件时，必须设定另一个链接程序开关（/delay:unload）。其次，必须修改源代码，并且在你想要卸载 DLL时调用--FUnloadDelayLoadedDLL函数：

```
BOOL __FUnloadDelayLoadedDLL(PCSTR szDll);
```

/Delay:unload链接程序开关告诉链接程序将另一个节放入文件中。该节包含了你清除已经调用的函数时需要的信息，这样它们就可以再次调用 --delayLoadHelper函数。当调用 --FUnloadDelayLoadedDll时，你将想要卸载的延迟加载的 DLL的名字传递给它。该函数进入文件中的未卸载节，并清除DLL的所有函数地址，然后--FUnloadDelayLoadedDll调用FreeLibrary，以便卸载该DLL。

下面要指出一些重要的问题。首先，千万不要自己调用 FreeLibrary来卸载DLL，否则函数的地址将不会被清除，这样，当下次试图调用 DLL中的函数时，就会导致访问违规。第二，当调用--FUnloadDelayLoadedDll时，传递的DLL名字不应该包含路径，名字中的字母必须与你将DLL名字传递给 /DelayLoad链接程序开关时使用的字母大小写相同，否则， --FUnload DelayLoadedDll的调用将会失败。第三，如果永远不打算卸载延迟加载的 DLL，那么请不要设定/Delay:unload链接程序开关，并且你的可执行文件的长度应该比较小。最后，如果你不从用/Delay:unload开关创建的模块中调用 --FUnloadDelayLoadedDll，那么什么也不会发生， --FUnloadDelayLoadedDll什么操作也不执行，它将返回FALSE。

延迟加载的DLL具备的另一个特性是，按照默认设置，调用的函数可以与一些内存地址相链接，在这些内存地址上，系统认为函数将位于一个进程的地址中（本章后面将介绍链接的问题）。由于创建可链接的延迟加载的 DLL节会使你的可执行文件变得比较大，因此链接程序也支持一个/Delay:nobind开关。因为人们通常都喜欢进行链接，因此大多数应用程序不应该使用这个链接开关。

延迟加载的DLL的最后一个特性是供高级用户使用的，它真正显示了 Microsoft的注意力之所在。当--delayLoadHelper函数执行时，它可以调用你提供的挂钩函数。这些函数将接收 --delayLoadHelper函数的进度通知和错误通知。此外，这些函数可以重载 DLL如何加载的方法以及如何获取函数的虚拟内存地址的方法。

若要获得通知或重载的行为特性，必须对你的源代码做两件事情。首先必须编写类似清单20-1所示的DliHook函数那样的挂钩函数。DliHook框架函数并不影响--delayLoadHelper函数的运行。若要改变它的行为特性，可启动 DliHook函数，然后根据需要对它进行修改。接着将函数的地址告诉--delayLoadHelper。

在DelayImp.lib静态链接库中，定义了两个全局变量，即 --pfnDliNotifyHook和--pfnDliFailureHook。这两个变量均属于pfnDliHook类型：

```
typedef FARPROC (WINAPI *PfnDliHook)(
   unsigned dliNotify,
   PDelayLoadInfo pdli);
```

China-pub.com
下载

如你所见，这是个数据类型的函数，与我的DliHook函数的原型相匹配。在DelayImp.lib文件中，两个变量被初始化为NULL，它告诉--delayLoadHelper不要调用任何挂钩函数。若要使你的函数被调用，必须将这两个函数中的一个设置为挂钩函数的地址。在我的代码中，我只是将下面两行代码添加到全局作用域：

```
PfnDliHook __pfnDliNotifyHook  = DliHook;
PfnDliHook __pfnDliFailureHook = DliHook;
```

如你所见，--delayLoadHelper实际上是与两个回调函数一道运行的。它调用一个函数以便报告通知，调用另一个函数来报告失败情况。由于这两个函数的原型是相同的，而第一个参数dliNotify告诉为什么调用这个函数，因此我总是通过创建单个函数并将两个变量设置为指向我的一个函数，使我的工作变得简单一些。

Visual C++ 6.0的延迟加载DLL的新特性非常出色，许多编程人员几年前就希望使用这个特性。可以想像许多应用程序（尤其是Microsoft的应用程序）都将充分利用这个特性。

DelayLoadApp示例应用程序

清单20-1中列出的DelayLoadApp应用程序（“20 DelayLoadApp.exe”）显示了在充分利用延迟加载DLL时应该做的所有工作。为了演示的需要，必须使用一个简单的的DLL，它的代码位于20-DelayLoadLib目录中。

由于该应用程序加载了“20 DelayLoadLib”模块，因此当运行该应用程序时，加载程序不必将该模块映射到进程的地址空间中。在该应用程序中，我定期调用IsModuleLoaded函数。该函数只是用来显示一个消息框，通知是否有一个模块加载到了进程的地址空间中。当该应用程序初次启动运行时，“20 DelayLoadLib”模块尚未加载，因此出现图20-6所示的消息框。

然后该应用程序调用从DLL输入的一个函数，这使得__delayLoadHelper函数能够自动加载该DLL。当该函数返回时，便出现图20-7所示的消息框。



图20-6 DelayLoadApp显示“20 DelayLoadLib”模块尚未加载



图20-7 DelayLoadApp显示“20 DelayLoadLib”模块已经加载

当这个消息框关闭时，DLL中的另一个函数被调用。由于该函数是在同一个DLL中，因此该DLL不必再次加载到该地址空间中去，不过该新函数的地址被转换并调用。

这时，FUnloadDelayLoadedDLL函数被调用，它负责卸载“20 DelayLoadedLib”。同样，如果调用IsModuleLoaded函数，就可以显示图20-6所示的消息框。最后，一个输入函数再次被调用，它由于重新加载“20 DelayLoadLib”模块，从而使最后一次对IsModuleLoaded的调用能够显示图20-7所示的消息框。

如果一切顺利，程序将按描述的那样工作。然而，如果在运行程序之前删除“20 Delay LoadLib”模块或者如果模块不包含导入的一个函数，就会引发异常。示例代码显示了如何从这种情况中恢复过来。

最后，该应用程序显示了如何正确地设置延迟加载的挂钩函数的方法。我的DliHook框架函数并没有执行什么新奇的功能。然而它能够捕获各个通知消息，显示收到这些通知时能够执行的操作。

China-pub.com 下载

清单20-1 DelayLoadApp示例应用程序



DelayLoadApp.cpp

```
/******************************************************************************
Module:  DelayLoadApp.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


#include <Delayimp.h>   // For error handling & advanced features
#include "..\20-DelayLoadLib\DelayLoadLib.h"    // My DLL function prototypes


//////////////////////////////////////////////////////////////////////////////
// Statically link __delayLoadHelper/__FUnloadDelayLoadedDLL
#pragma comment(lib, "Delayimp.lib")

// Tell the linker that my DLL should be delay loaded
// Note the 2 (\") because the filename has a space in it
#pragma comment(linker, "/DelayLoad:\"20 DelayLoadLib.dll\"")

// Tell the linker that I want to be able to unload my DLL
#pragma comment(linker, "/Delay:unload")

// Tell the linker to make the delay load DLL unbindable
// You usually want this, so I commented out this line
//#pragma comment(linker, "/Delay:nobind")


// The name of the Delay-Load module (only used by this sample app)
TCHAR g_szDelayLoadModuleName[] = TEXT("20 DelayLoadLib");


//////////////////////////////////////////////////////////////////////////////


// Forward function prototype
LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep);


//////////////////////////////////////////////////////////////////////////////


void IsModuleLoaded(PCTSTR pszModuleName) {
```

China-pub.com
下载


```
   HMODULE hmod = GetModuleHandle(pszModuleName);
   char sz[100];
#ifdef UNICODE
   wsprintfA(sz, "Module \"%S\" is %Sloaded.",
      pszModuleName, (hmod == NULL) ? L"not " : L"");
#else
   wsprintfA(sz, "Module \"%s\" is %sloaded.",
      pszModuleName, (hmod == NULL) ? "not " : "");
#endif
   chMB(sz);
}

///////////////////////////////////////////////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Wrap all calls to delay-load DLL functions inside SEH
   __try {
      int x = 0;

      // If you're in the debugger, try the new Debug.Modules menu item to
      // see that the DLL is not loaded prior to executing the line below
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib();  // Attempt to call delay-load function

      // Use Debug.Modules to see that the DLL is now loaded
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib2(); // Attempt to call delay-load function

      // Unload the delay-loaded DLL
      // NOTE: Name must exactly match /DelayLoad:(DllName)
      __FUnloadDelayLoadedDLL("20 DelayLoadLib.dll");

      // Use Debug.Modules to see that the DLL is now unloaded
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib();  // Attempt to call delay-load function

      // Use Debug.Modules to see that the DLL is loaded again
      IsModuleLoaded(g_szDelayLoadModuleName);
   }
   __except (DelayLoadDllExceptionFilter(GetExceptionInformation())) {
      // Nothing to do in here, thread continues to run normally
   }

   // More code can go here...

   return(0);
}


///////////////////////////////////////////////////////////////////////////////


LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep) {
```

China-pub.com 下载

```
   // Assume we recognize this exception
   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

   // If this is a Delay-load problem, ExceptionInformation[0] points
   // to a DelayLoadInfo structure that has detailed error info
   PDelayLoadInfo pdli =
      PDelayLoadInfo(pep->ExceptionRecord->ExceptionInformation[0]);

   // Create a buffer where we construct error messages
   char sz[500] = { 0 };

   switch (pep->ExceptionRecord->ExceptionCode) {
   case VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND):
      // The DLL module was not found at run time
      wsprintfA(sz, "Dll not found: %s", pdli->szDll);
      break;

   case VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND):
      // The DLL module was found, but it doesn't contain the function
      if (pdli->dlp.fImportByName) {
         wsprintfA(sz, "Function %s was not found in %s",
            pdli->dlp.szProcName, pdli->szDll);
      } else {
         wsprintfA(sz, "Function ordinal %d was not found in %s",
            pdli->dlp.dwOrdinal, pdli->szDll);
      }
      break;

   default:
      // We don't recognize this exception
      lDisposition = EXCEPTION_CONTINUE_SEARCH;
      break;
   }

   if (lDisposition == EXCEPTION_EXECUTE_HANDLER) {
      // We recognized this error and constructed a message, show it
      chMB(sz);
   }

   return(lDisposition);
}


///////////////////////////////////////////////////////////////////////////////
// Skeleton DliHook function that does nothing interesting
FARPROC WINAPI DliHook(unsigned dliNotify, PDelayLoadInfo pdli) {

   FARPROC fp = NULL;   // Default return value

   // NOTE: The members of the DelayLoadInfo structure pointed
   // to by pdli shows the results of progress made so far.

   switch (dliNotify) {
   case dliStartProcessing:
      // Called when __delayLoadHelper attempts to find a DLL/function
```

China-pub.com
下载


```
        // Return 0 to have normal behavior or nonzero to override
        // everything (you will still get dliNoteEndProcessing)
        break;

    case dliNotePreLoadLibrary:
        // Called just before LoadLibrary
        // Return NULL to have __delayLoadHelper call LoadLibary
        // or you can call LoadLibrary yourself and return the HMODULE
        fp = (FARPROC) (HMODULE) NULL;
        break;

    case dliFailLoadLib:
        // Called if LoadLibrary fails
        // Again, you can call LoadLibary yourself here and return an HMODULE
        // If you return NULL, __delayLoadHelper raises the
        // ERROR_MOD_NOT_FOUND exception
        fp = (FARPROC) (HMODULE) NULL;
        break;

    case dliNotePreGetProcAddress:
        // Called just before GetProcAddress
        // Return NULL to have __delayLoadHelper call GetProcAddress
        // or you can call GetProcAddress yourself and return the address
        fp = (FARPROC) NULL;
        break;

    case dliFailGetProc:
        // Called if GetProcAddress fails
        // You can call GetProcAddress yourself here and return an address
        // If you return NULL, __delayLoadHelper raises the
        // ERROR_PROC_NOT_FOUND exception
        fp = (FARPROC) NULL;
        break;
    case dliNoteEndProcessing:
        // A simple notification that __delayLoadHelper is done
        // You can examine the members of the DelayLoadInfo structure
        // pointed to by pdli and raise an exception if you desire
        break;
    }

    return(fp);
}


///////////////////////////////////////////////////////////////////////////


// Tell __delayLoadHelper to call my hook function
PfnDliHook __pfnDliNotifyHook  = DliHook;
PfnDliHook __pfnDliFailureHook = DliHook;


////////////////////////////////// End of File /////////////////////////////////////
```

DelayLoadLib.cpp

China-pub.com 下载

```
/******************************************************************************
Module:  DelayLoadLib.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


#define DELAYLOADLIBAPI extern "C" __declspec(dllexport)
#include "DelayLoadLib.h"


//////////////////////////////////////////////////////////////////////////////
int fnLib() {

   return(321);
}


//////////////////////////////////////////////////////////////////////////////


int fnLib2() {

   return(123);
}


//////////////////////////////// End of File /////////////////////////////////
```

DelayLoadLib.h

```
/******************************************************************************
Module:  DelayLoadLib.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#ifndef DELAYLOADLIBAPI
#define DELAYLOADLIBAPI extern "C" __declspec(dllimport)
#endif


//////////////////////////////////////////////////////////////////////////////


DELAYLOADLIBAPI int fnLib();
DELAYLOADLIBAPI int fnLib2();
```

China-pub.com
下载


```
//////////////////////////////// End of File ///////////////////////////////////
```

20.4 函数转发器

函数转发器是DLL的输出节中的一个项目，用于将对一个函数的调用转至另一个DLL中的另一个函数。例如，如果在Windows 2000的Kernel32.dll上运行Visual C++的DumpBin实用程序，那么将看到类似下面的一部分输出：

```
C:\winnt\system32>DumpBin -Exports Kernel32.dll
     (some output omitted)
360  167   HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)
361  168   HeapCompact  (000128D9)
362  169   HeapCreate  (000126EF)
363  16A   HeapCreateTagsW  (0001279E)
364  16B   HeapDestroy  (00012750)
365  16C   HeapExtend  (00012773)
366  16D   HeapFree (forwarded to NTDLL.RtlFreeHeap)
367  16E   HeapLock  (000128ED)
368  16F   HeapQueryTagW  (000127B8)
369  170   HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)
370  171   HeapSize (forwarded to NTDLL.RtlSizeHeap)
     (remainder of output omitted)
```

这个输出显示了4个转发函数。每当你的应用程序调用HeapAlloc、HeapFree、HeapReAlloc或HeapSize时，你的可执行模块就会自动与Kernel32.dll相链接。当激活你的可执行模块时，加载程序就加载Kernel32.dll并看到转发的函数实际上包含在NTDLL.dll中。然后它也加载NTDLL.dll。当你的可执行模块调用HeapAlloc时，它实际上调用的是NTDLL.dll中的RtlAllocateHeap函数。系统中的任何地方都不存在HeapAlloc函数。

如果调用下面的函数，GetProcAddress就会查看Kernel32的输出节，发现HeapAlloc是个转发函数，然后按递归方式调用GetProcAddress函数，查找NTDLL.dll的输出节中的RtlAllocateHeap。

```
GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");
```

也可以利用DLL模块中的函数转发器。最容易的方法是像下面这样使用一个pragma指令：

```
// Function forwarders to functions in DllWork
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
```

这个pragma告诉链接程序，被编译的DLL应该输出一个名叫SomeFunc的函数。但是SomeFunc函数的实现实际上位于另一个名叫SomeOtherFunc的函数中，该函数包含在称为DllWork.dll的模块中。必须为你想要转发的每个函数创建一个单独的pragma代码行。

20.5 已知的DLL

操作系统提供的某些DLL得到了特殊的处理。这些DLL称为已知的DLL。它们与其他DLL基本相同，但是操作系统总是在同一个目录中查找它们，以便对它们进行加载操作。在注册表中有下面的关键字：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
   Session Manager\KnownDLLs
```

在我的计算机上使用RegEdit.exe实用程序时显示的是如图20-8所示的对话框。

China-pub.com 下载

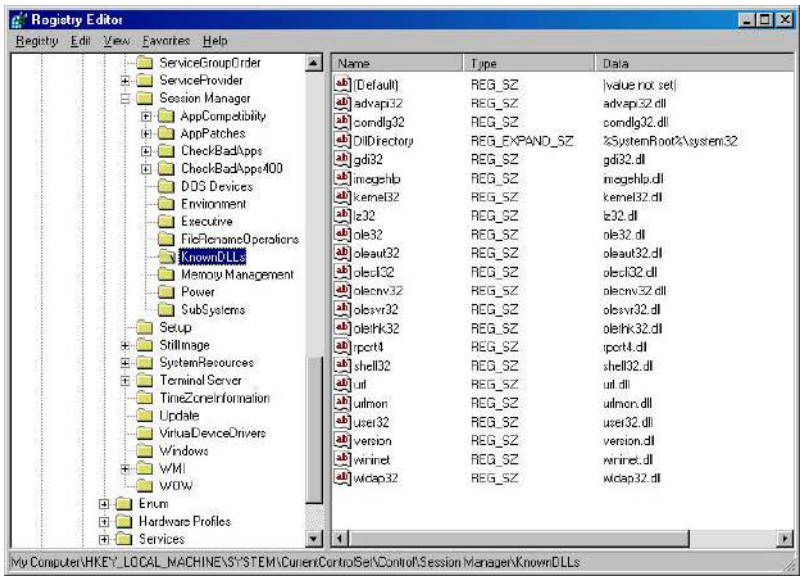


图20-8 使用 RegEdit .exe实用程序时显示的子关键字对话框

如你所见，这个关键字包含一组值的名字，这些名字是某些 DLL的名字。每个值名字都有一个数据值，该值恰好与带有.dll文件扩展名的值名字相同（不过，情况并非完全如此，我将在下面的例子中加以说明）。当LoadLibrary或LoadLibraryEx被调用时，这些函数首先查看是否传递了包含.dll扩展名的DLL名字。如果没有传递，那么它们将使用通常的搜索规则来搜索 DLL。

如果确实设定了 .dll扩展名，那么这些函数将删除扩展名，然后搜索注册表关键字 KnownDLL，以便确定它是否包含匹配的值名字。如果没有找到匹配的名字，便使用通常的搜索规则。但是，如果找到了匹配的值名字，系统将查找相关的值数据，并设法使用值数据来加载DLL。系统也开始在注册表中的 DllDirectory值数据指明的目录中搜索 DLL。按照默认设置，Windows 2000上的DllDirectory值的数据是%SystemRoot%\System32。

为了更好地说明情况，假设将下面的值添加给注册表关键字 KnownDLL：

```
Value name: SomeLib
Value data: SomeOtherLib.dll
```

当调用下面的函数时，系统将使用通常的搜索规则来查找该文件：

```
LoadLibrary("SomeLib");
```

但是，如果调用下面的函数，系统将会发现有一个匹配的值名字（记住，当系统检查注册表的值名字时，它将删除扩展名.dll）。

```
LoadLibrary("SomeLib.dll");
```

这时，系统设法加载称为 SomeOtherLib.dll的文件，而不是加载 SomeLib.dll。它首先在%SystemRoot%\System32目录中查找SomeOtherLib.dll。如果它在该目录中找到了文件，它就加载该文件。如果文件不在该目录中， LoadLibrary(Ex)运行失败并返回 NULL，同时，对 GetLastError的调用将返回2（ERROR_FILE_NOT_FOUND）。

20.6 DLL转移

Windows 98 Windows 98不支持DLL转移。

China-pub.com
下载

当Windows刚刚开发成功时，RAM和磁盘空间是非常宝贵的。因此Windows在设计时总是尽可能多地安排资源的共享，以节省宝贵的存储器资源。为了达到这个目的， Microsoft建议，多个应用程序共享的任何模块，如C/C++运行期库和Microsoft基础类（MFC）DLL等，应该放入Windows的系统目录中。这样，系统就能够方便地找到共享文件。

随着时间的推移，这变成一个非常严重的问题，因为安装程序会用旧文件或尚未完全实现向后兼容的新文件来改写该目录中的文件。这将使用户的其他应用程序无法正确地运行。今天，硬盘的容量已经非常大，并且很便宜， RAM的容量也相当富裕，价格也便宜了一些。因此，Microsoft改变了原先的开发策略，非常支持你将应用程序的所有文件放入它们自己的目录中，而不要去碰Windows的系统目录中的任何东西。这样，你的应用程序就不会损坏别的应用程序，别的应用程序也不会损坏你的应用程序。

为了给你提供相应的帮助，Microsoft给Windows 2000增加了一个DLL转移特性。这个特性能够强制操作系统的加载程序首先从你的应用程序目录中加载文件模块。只有当加载程序无法在应用程序目录中找到该文件时，它才搜索其他目录。

为了强制加载程序总是首先查找应用程序的目录，要做的工作就是在应用程序的目录中放入一个文件。该文件的内容可以忽略，但是该文件必须称为 AppName.local。

例如，如果有一个可执行文件的名字是 SuperApp.exe，那么转移文件必须称为SuperApp.exe.local。

在系统内部，LoadLibrary(Ex)已经被修改，以便查看是否存在该文件。如果应用程序的目录中存在该文件，该目录中的模块就已经被加载。如果应用程序的目录中不存在这个模块，LoadLibrary(Ex)将正常运行。

对于已经注册的COM对象来说，这个特性是非常有用的。它使应用程序能够将它的 COM对象DLL放入自己的目录，这样，注册了相同COM对象的其他应用程序就无法干扰你的操作。

20.7 改变模块的位置

每个可执行模块和DLL模块都有一个首选的基地址，用于标识模块应该映射到的进程地址空间中的理想内存地址。当创建一个可执行模块时，链接程序将该模块的首选基地址设置为0x00400000。如果是DLL模块，链接程序设置的首选基地址是0x10000000。使用Visual Studio的DumpBin实用程序（带有 /Headers开关），可以看到一个映像的首选基地址。下面是使用DumpBin来转储它自己的头文件信息的例子：

```
C:\>DUMPBIN /headers dumpbin.exe

Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file dumpbin.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (i386)
               3 number of sections
        3588004A time date stamp Wed Jun 17 10:43:38 1998
               0 file pointer to symbol table
```

China-pub.com 下载

```
              0 number of symbols
             E0 size of optional header
            10F characteristics
                  Relocations stripped
                  Executable
                  Line numbers stripped
                  Symbols stripped
                  32 bit word machine

OPTIONAL HEADER VALUES
             10B magic #
            6.00 linker version
            1000 size of code
            2000 size of initialized data
               0 size of uninitialized data
            1320 RVA of entry point
            1000 base of code
            2000 base of data
          400000 image base          <-- Module's preferred base address
            1000 section alignment
            1000 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            4000 size of image
            1000 size of headers
           127E2 checksum
               3 subsystem (Windows CUI)
               0 DLL characteristics
          100000 size of stack reserve
            1000 size of stack commit
          :
```

当这个可执行模块被调用时，操作系统加载程序为新进程创建一个虚拟地址。然后该加载程序将可执行模块映射到内存地址0x00400000，并将DLL模块映射到0x10000000。为什么这个首选基地址这么重要呢？让我们看一看下面的代码：

```
int g_x;

void Func() {
   g_x = 5;     // This is the important line.
}
```

当编译器处理Func函数时，该编译器和链接程序创建类似下面的机器代码：

```
MOV   [0x00414540], 5
```

换句话说，编译器和链接程序创建的机器代码实际上是在变量 g_x的地址0x00414540中硬编码的代码。该地址位于机器代码中，用于标识变量在进程的地址空间中的绝对位置。但是，当并且仅当可执行模块加载到它的首选基地址0x00400000中时，这个内存地址才是正确的。

如果在一个DLL模块中我们拥有与上面完全相同的代码，那将会如何呢？在这种情况下，编译器和链接程序将生成类似下面的机器代码：

```
MOV   [0x10014540], 5
```

同样，注意DLL的变量g_z的虚拟内存地址是在磁盘驱动器上的DLL文件映像中硬编码的代码。而且，如果该DLL确实是在它的首选基地址上加载的，那么这个内存地址是绝对正确的。

China-pub.com
下载

现在假设你设计的应用程序需要两个DLL。按照默认设置，链接程序将.exe模块的首选基地址设置为0x00400000，同时，链接程序将两个DLL模块的首选基地址均设置为0x10000000。如果想要运行.exe模块，那么加载程序便创建该虚拟地址空间，并将.exe模块映射到内存地址0x00400000中。然后加载程序将第一个DLL映射到内存地址0x10000000中。但是，当加载程序试图将第二个DLL映射到进程的地址空间中去时，它将无法把它映射到该模块的首选基地址中，必须改变该DLL模块的位置，将它放到别的什么地方。

改变可执行（或DLL）模块的位置是个非常可怕的过程，应该采取措施避免这样的操作。为什么要避免这样的操作呢？假设加载程序将第二个DLL的地址改到0x20000000。这时，将变量g_x的值改为5的代码应该是：

```
MOV   [0x20014540], 5
```

但是文件映像中的代码却类似下面的样子：

```
MOV   [0x10014540], 5
```

如果文件映像的代码被允许执行，那么第一个DLL模块中大约有4个字节的值将被值5改写。这是不能允许的。加载程序必须修改这个代码。当链接程序创建你的模块时，它将把一个移位节嵌入产生的文件中。这一节包含一个字节位移的列表。每个字节位移用于标识一个机器代码指令使用的内存地址。如果加载程序能够将一个模块映射到它的首选基地址中，那么系统将永远不会访问模块的移位节。这当然是我们所希望的——你永远不希望使用移位节。

另一方面，如果该模块不能映射到它的首选基地址中，加载程序便打开模块的移位节，并对所有项目重复执行该操作。对于找到的每个项目，加载程序将转至包含要修改机器代码指令的存储器页面。然后它抓取机器指令当前正在使用的内存地址，并将模块的首选基地址与模块实际映射到的地址之间的差与该地址相加。

因此，在上面的例子中，第二个DLL被映射到0x20000000，但是它的首选基地址是0x10000000。它产生的差是0x10000000，然后这个差与机器代码指令的地址相加，产生的结果如下：

```
MOV   [0x20014540], 5
```

现在，第二个DLL中的代码将能够正确地引用它的变量g_x。

当模块不能映射到它的首选基地址中去时，将会出现下面两个主要问题：

- 加载程序必须重复经过移位节，并且要修改模块的许多代码。这会影响到系统的运行速度，并且会增加应用程序的初始化时间。
- 当加载程序将数据写入模块的代码页面时，系统的写入时拷贝（copy-on-write）机制将强制这些页面被系统的页文件拷贝。

上面的第二个问题确实会产生非常不良的作用。它意味着模块的代码页面不再能够从该磁盘上的模块文件映像中删除和重新加载。相反，这些页面将在需要时从系统的页文件中来回倒腾。这也会影响系统的运行速度。但是，还有更为糟糕的事情。由于页文件拷贝了模块的所有代码页面，因此系统只有较少的存储器可供系统中的所有进程来运行。这就限制了用户的电子表格、文字处理文档、CAD图形和位图的大小。

另外，可以创建一个不包含移位节的可执行模块或DLL模块。当创建该模块时，你可以将/FIXED开关传递给链接程序。使用这个开关，能够使模块变得比较小一些，但是这意味着模块不能被改变位置。如果模块不能加载到它的首选基地址，那么它就根本无法加载。如果加载程序必须改变模块的位置，但是却不存在用于该模块的移位节，那么加载程序就会撤消整个进程，并且向用户显示一条“进程异常终止”的消息。

China-pub.com 下载

对于只包含资源的DLL来说，这是个问题。只包含资源的DLL中是没有代码的，因此，使用/FIXED开关来链接该DLL是很有意义的。但是，如果只有资源的DLL不能加载到它的首选基地址，那么该模块就根本不能加载。这很奇怪。为了解决这个问题，链接程序允许你用嵌入头文件中的信息来创建一个模块，以指明该模块不包含移位信息，因为不需要这样的信息。Windows 2000加载程序可以使用这个头文件信息，并且允许加载只包含资源的DLL，而不会降低系统的运行速度或者损害页文件的空间。

若要创建不需要进行任何移位的文件映像，请使用/SUBSYSTEM：WINDOWS，5.0开关，或者使用/SUBSYSTEM：CONSOLE，5.0开关，但是不要设定/FIXED开关。如果链接程序确定模块中没有什么东西需要进行移位设置，那么它就从模块中删除移位节，并且关闭头文件中的专用IMAGE_FILE_RELOCS_STRIPPED标志。当Windows 2000加载该模块时，它知道该模块可以移位（因为IMAGE_FILE_RELOCS_STRIPPED标志是关闭的），但是该模块没有移位（因为不存在移位节）。请注意，这是Windows 2000加载程序的一个新特性，它说明了/SUBSYSTEM开关的结尾处为什么需要带有5.0。

现在你已经知道首选基地址的重要性了。所以，如果你有多个模块需要加载到单个地址空间中，必须为每个模块设置不同的首选基地址。Microsoft Visual Studio的Project Settings（项目设置）对话框使得这项操作变得非常容易。你只需要选定Link（链接）选项卡，再选定Output（输出）类别。在Base Address（基地址）域中（该域默认为空），可以输入一个数字。在图20-9中，我将我的DLL模块的基地址设置为0x20000000。

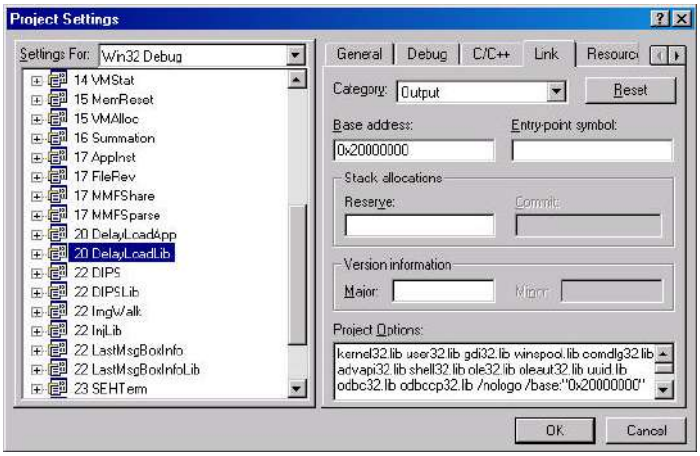


图20-9 Project Settings 对话框

另外，始终都应该从高位内存地址开始加载DLL，然后逐步向下加载到低位内存地址，以减少地址空间中出现的碎片。

注意 首选基地址必须始终从分配粒度边界开始。在迄今为止的所有平台上，系统的分配粒度是64 KB。将来这个分配粒度可能发生变化。第13章已经对分配粒度进行了详细的介绍。

好了，现在已经对所有的内容做了介绍。但是，如果将许多模块加载到单个地址空间中，情况将会如何呢？如果有一种非常容易的方法，可以为所有的模块设置很好的首选基地址，那就好了。幸运的是，这种方法确实有。

Visual Studio配有一个实用程序，称为Rebase.exe。如果运行不带任何命令行参数的Rebase

China-pub.com
下载

程序，会得到下面的使用信息：

```
usage:
REBASE [switches]
       [-R image-root [-G filename] [-O filename] [-N filename]]
       image-names...

       One of -b and -i switches are mandatory.

       [-a] Used with -x.  extract All debug info into .dbg file
       [-b InitialBase] specify initial base address
       [-c coffbase_filename] generate coffbase.txt
           -C includes filename extensions, -c does not
       [-d] top down rebase
       [-f] Strip relocs after rebasing the image

       [-i coffbase_filename] get base addresses from coffbase_filename
       [-l logFilePath] write image bases to log file.
       [-p] Used with -x.  Remove private debug info when extracting
       [-q] minimal output
       [-s] just sum image range
       [-u symbol_dir] Update debug info in .DBG along this path
       [-v] verbose output
       [-x symbol_dir] extract debug info into separate .DBG file first
       [-z] allow system file rebasing
       [-?] display this message

       [-R image_root] set image root for use by -G, -O, -N
       [-G filename] group images together in address space
       [-O filename] overlay images in address space
       [-N filename] leave images at their origional address
           -G, -O, -N, may occur multiple times.  File "filename"
           contains a list of files (relative to "image-root")
```

Platform SDK文档对Rebase实用程序进行了介绍，因此在这里就不再对它详细说明了。不过要补充的一点是，这个实用程序并没有什么奇特之处。从内部来说，它只是分别为每个指定的文件调用ReBaseImage函数：

```
BOOL ReBaseImage(
   PSTR CurrentImageName,      // Pathname of file to be rebased
   PSTR SymbolPath,            // Symbol file path so debug info
                               // is accurate
   BOOL fRebase,               // TRUE to actually do the work; FALSE
                               // to pretend
   BOOL fRebaseSysFileOk,      // FALSE to not rebase system images
   BOOL fGoingDown,            // TRUE to rebase the image below
                               // an address
   ULONG CheckImageSize,       // Maximum size that image can grow to
   ULONG* pOldImageSize,       // Receives original image size
   ULONG* pOldImageBase,       // Receives original image base address
   ULONG* pNewImageSize,       // Receives new image size
   ULONG* pNewImageBase,       // Receives new image base address
   ULONG TimeStamp);           // New timestamp for image
```

当你执行Rebase程序，为它传递一组映象文件名时，它将执行下列操作：

1) 它能够仿真创建一个进程的地址空间。

2) 它打开通常被加载到该地址空间中的所有模块。

China-pub.com 下载

3) 它仿真改变各个模块在仿真地址空间中的位置，这样，各个模块就不会重叠。

4) 对于已经移位的模块，它会分析该模块的移位节，并修改磁盘上的模块文件中的代码。

5) 它会更新每个移位模块的头文件，以反映新的首选基地址。

Rebase是个非常出色的工具，建议尽可能使用这个工具。应该在接近你的应用程序模块创建周期结束时运行这个实用程序，直到所有模块创建完成。另外，如果使用 Rebase实用程序，可以忽略 Project Settings对话框中的基地址的设置。链接程序将为 DLL提供一个基地址0x10000000，但是Rebase会修改这个地址。

顺便要指出的是，决不应该改变操作系统配备的任何模块的地址。 Microsoft在销售Windows操作系统之前，在操作系统提供的所有文件上运行了 Rebase程序，因此，如果将它们映射到单个地址空间中，所有的操作系统模块都不会重叠。

我给第4章中介绍的ProcessInfo.exe应用程序添加了一个特殊的性质。这个工具显示了一个列表，里面包括进程的地址空间中存在的所有模块。在 BaseAddr列的下面，会看到模块加载到的虚拟内存地址。在 BaseAddr列的右边，是ImagAddr列。通常这个列是空的，这表示模块加载到了它的首选基地址中。你希望看到所有模块都是这样。但是如果出现带有括号的另一个地址，就表示该模块没有加载到它的首选基地址中，这一列中的数字表示从模块的磁盘文件的头文件信息中读取的该模块的首选基地址。

下面是正在查看 Acrord32.exe进程的ProcessInfo.exe工具。注意，有些模块确实加载到了它们的首选基地址中，有些模块则没有。你还会发现所有模块的首选基地址都是0x10000000，表示它们是DLL模块，并且这些模块的创建者并没有考虑改变地址的问题（见图20-10）。

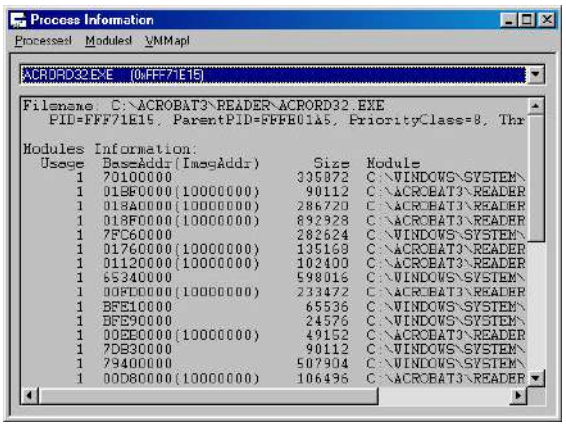


图20-10 Process Information 窗口

20.8 绑定模块

模块的移位是非常重要的，它能够大大提高整个系统的运行性能。但是，还可以使用许多别的办法来提高它的性能。比如说，可以改变应用程序的所有模块的位置。让我们回忆一下第19章中关于加载程序如何查看所有输入符号的地址的情况。加载程序将符号的虚拟地址写入可执行模块的输入节中。这样就可以参考输入的符号，以便到达正确的内存位置。

让我们进一步考虑一下这个问题。如果加载程序将输入符号的虚拟地址写入 .exe模块的输入节，那么拷贝输入节的这些页面将被写入虚拟地址。由于这些页面是写入时拷贝的页面，因此这些页面将被页文件拷贝。这样我们就遇到了一个与改变模块的位置相类似的问题，即映像文件的各个部分将与系统的页文件之间来回倒腾，而不是在必要时将它们删除或者从文件的磁盘映像中重新读取。另外，加载程序必须对（所有模块的）所有输入符号的地址进行转换，这是很费时间的。

可以将模块绑定起来，使应用程序能够更快的进行初始化，并且使用较少的存储器。绑定一个模块时，可以为该模块的输入节配备所有输入符号的虚拟地址。为了缩短初始化的时间和使用较少的存储器，当然必须在加载模块之前进行这项操作。

China-pub.com
下载

Visual Studio配有另一个实用程序，名字是Bind.exe，当运行这个不带任何命令行参数的程序时，它将输出下面的信息：

```
usage: BIND [switches] image-names...
            [-?] display this message
            [-c] no caching of import dlls
            [-o] disable new import descriptors
            [-p dll search path]
            [-s Symbol directory] update any associated .DBG file
            [-u] update the image
            [-v] verbose output
            [-x image name] exclude this image from binding
            [-y] allow binding on images located above 2G
```

Bind（绑定）实用程序在Platform SDK文档中作了描述，这里就不再详细介绍了。但是与Rebase一样，这个实用程序也不是不可思议的程序。从内部来说，它为每个指定的文件重复调用BindImageEx函数：

```
BOOL BindImageEx(
   DWORD dwFlags,        // Flags giving fine control over the function
   PSTR pszImageName,    // Pathname of file to be bound
   PSTR pszDllPath,      // Search path used for locating image files
   PSTR pszSymbolPath,   // Search path used to keep debug info accurate
   PIMAGEHLP_STATUS_ROUTINE StatusRoutine);  // Callback function
```

最后一个参数StatusRoutine是一个回调函数的地址，BindImageEx定期调用这个函数，这样就能够监控连接进程。下面是该函数的原型：

```
BOOL WINAPI StatusRoutine(
   IMAGEHLP_STATUS_REASON Reason, // Module/procedure not found, etc.
   PSTR pszImageName,        // Pathname of file being bound
   PSTR pszDllName,          // Pathname of DLL
   ULONG_PTR VA,             // Computed virtual address
   ULONG_PTR Parameter);     // Additional info depending on Reason
```

当执行Bind程序，传递给它一个映像文件名时，它将执行下列操作：

1) 打开指定映像文件的输入节。

2) 对于输入节中列出的每个DLL，它打开该DLL文件，查看它的头文件以确定它的首选基地址。

3) 查看DLL的输出节中的每个输入符号。

4) 取出符号的RVA，并将模块的首选基地址与它相加。将可能产生的输入符号的虚拟地址写入映像文件的输入节中。

5) 将某些辅助信息添加到映像文件的输入节中。这些信息包括映像文件绑定到的所有DLL模块的名字和这些模块的时戳。

在第19章中，我们使用DumpBin实用程序来查看Calc.exe的输入节的内容。该输出信息的底部显示了第5个步骤中添加的已链接输入信息。下面是输出信息的有关部分：

```
Header contains the following bound import information:
  Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
  Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
  Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
  Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
```

China-pub.com 下载

可以看到Calc.exe连接到了哪些模块，方括号中的数字表示 Microsoft是在何时创建每个DLL模块的。这个32位的时戳值在方括号的后面展开了，并以人们能够识别的字符串显示出来。

在执行整个进程期间，Bind程序做了两个重要的假设：

- 当进程初始化时，需要的DLL实际上加载到了它们的首选基地址中。可以使用前面介绍的Rebase实用程序来确保这一点。
- 自从绑定操作执行以来，DLL的输出节中引用的符号的位置一直没有改变。加载程序通过将每个DLL的时戳与上面第5个步骤中保存的时戳进行核对来核实这个情况。

当然，如果加载程序确定上面的两个假设中有一个是假的，那么 Bind就没有执行上面所说的有用的操作，加载程序必须通过人工来修改可执行模块的输入节，就像它通常所做的那样。但是，如果加载程序发现模块已经连接，需要的 DLL已经加载到它们的首选基地址中，而且时戳也匹配，那么它实际上已经无事可做。它不必改变任何模块的位置，也不必查看任何输入函数的虚拟地址。该应用程序只管启动运行就是了。

此外，它不需要来自系统的页文件的任何存储器。这太好了，我们简直拥有世界上最出色的系统。目前销售的商用应用程序许多都不具备相应的移位和绑定特性。

好了，现在你已经知道应该将应用程序配有的所有模块连接起来。但是应该在什么时候进行模块的连接呢？如果你在你的公司连接这些模块，可以将它们与你已经安装的系统 DLL绑定起来，而这些系统DLL并不一定是用户已经安装的。由于不知道用户运行的是 Windows 98还是Windows NT，或者是Windows 2000，也不知道这些操作系统是否已经安装了服务软件包，因此应该将绑定操作作为应用程序的安装操作的一部分来进行。

当然，如果用户能够对 Windows 98和Windows 2000进行双重引导，那么绑定的模块可能对这两个操作系统之一来说是不正确的。另外，如果用户在Windows 2000下安装你的应用程序，然后又升级到你的服务软件包，那么模块的绑定也是不正确的。在这些情况下，你和用户都可能无能为力。 Microsoft应该在销售操作系统时配备一个实用程序，使得操作系统升级后能够自动重新绑定每个模块。不过，现在还不存在这样的实用程序。

China-pub.com
下载

第21章　线程本地存储器

有时，将数据与对象的实例联系起来是很有帮助的。例如，窗口的附加字节可以使用SetWindowsWord和SetWindowLong函数将数据与特定的窗口联系起来。可以使用线程本地存储器将数据与执行的特定线程联系起来。例如，可以将线程的某个时间与线程联系起来。然后，当线程终止运行时，就能够确定线程的寿命。

C/C++运行期库要使用线程本地存储器（TLS）。由于运行期库是在多线程应用程序出现前的许多年设计的，因此运行期库中的大多数函数是用于单线程应用程序的。函数 strtok就是个很好的例子。应用程序初次调用 strtok时，该函数传递一个字符串的地址，并将字符串的地址保存在它自己的静态变量中。当你将来调用 strtok函数并传递NULL时，该函数就引用保存的字符串地址。

在多线程环境中，一个线程可以调用 strtok，然后，在它能够再次调用该函数之前，另一个线程也可以调用 Strtok。在这种情况下，第二个线程会在第一个线程不知道的情况下，让strtok用一个新地址来改写它的静态变量。第一个线程将来调用 strtok时将使用第二个线程的字符串，这就会导致各种各样难以发现和排除的错误。

为了解决这个问题，C/C++运行期库使用了TLS。每个线程均被赋予它自己的字符串指针，供strtok函数使用。需要予以同样对待的其他C/C++运行期库函数还有asctime和gmtime。

如果你的应用程序需要严重依赖全局变量或静态变量，那么TLS能够帮助解决它遇到的问题。但是编程人员往往尽可能减少对这些变量的使用，而更多地依赖自动（基于堆栈的）变量和通过函数的参数传递的数据。这样做是很好的，因为基于堆栈的变量总是与特定的线程相联系的。

标准的C运行期库一直是由许多不同的编译器供应商来实现和重新实现的。如果 C编译器不包含标准的C运行期库，那么就不值得去购买它。编程员多年来一直使用标准的 C运行期库，并且将会继续使用它，这意味着 strtok之类的函数的原型和行为特性必须与上面所说的标准 C运行期库完全一样。如果今天重新来设计C运行期库，那么它就必须支持多线程应用程序的环境，并且必须采取相应的措施来避免使用全局变量和静态变量。

在我的软件开发项目中，我总是尽可能避免使用全局变量和静态变量。如果你的应用程序使用全局变量和静态变量，那么建议你务必观察每个变量，并且了解一下它能否改变成基于堆栈的变量。如果打算将线程添加给应用程序，那么这样做可以节省大量时间，甚至单线程应用程序也能够从中得到许多好处。

在编写应用程序和DLL时，可以使用本章中介绍的两种TLS方法，即动态TLS和静态TLS。但是，当创建DLL时，这些TLS往往更加有用，因为DLL常常不知道它们链接到的应用程序的结构。不过，当编写应用程序时，你通常知道将要创建多少线程以及如何使用这些线程。然后就可以创造一些临时性的方法，或者最好是使用基于堆栈的方法（局部变量），将数据与创建的每个线程联系起来。不管怎样，应用程序开发人员也能从本章讲述的内容中得到一些启发。

21.1　动态TLS

若要使用动态TLS，应用程序可以调用一组 4个函数。这些函数实际上是 DLL用得最多的函数。图21-1显示了Windows用来管理TLS的内部数据结构。

China-pub.com 下载

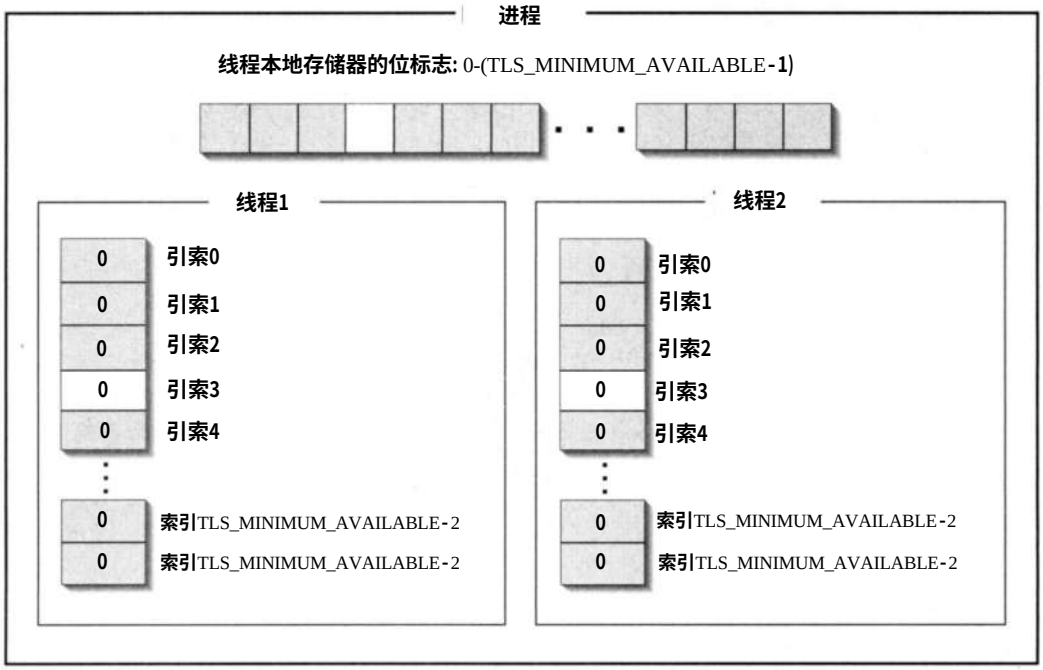


图21-1 用于管理TLS的内部数据结构

该图显示了系统中运行的线程正在使用的一组标志。每个标志均可设置为 FREE或者INUSE，表示TLS时隙(slot)是否正在使用。Microsoft保证至少TLS_MINIMUM_AVAILABLE位标志是可供使用的。另外，TLS_MINIMUM_AVAILABLE在WinNT.h中被定义为64。Windows 2000将这个标志数组扩展为允许有1000个以上的TLS时隙。对于任何一个应用程序来说，这个时隙数量足够了。

若要使用动态TLS，首先必须调用TlsAlloc函数：

```
DWORD TlsAlloc();
```

这个函数命令系统对进程中的位标志进行扫描，并找出一个 FREE标志。然后系统将该标志从FREE改为INUSE，并且TlsAlloc返回位数组中的标志的索引。DLL（或应用程序）通常将该索引保存在一个全局变量中。这是全局变量作为一个较好选择的情况之一，因为它的值是每个进程而不是每个线程使用的值。

如果TlsAlloc在该列表中找不到 FREE标志，它就返回 TLS_OUT_OF_INDEXES（在WinBase.h中定义为0xFFFFFFFF）。当TlsAlloc第一次被调用时，系统发现第一个标志是FREE，并将该标志改为INUSE，同时TlsAlloc返回0。TlsAlloc这样运行的概率是99%。下面介绍在另外的1%的概率下TlsAlloc是如何运行的。

当创建一个线程时，便分配一个 TLS_MINIMUM_AVAILABLEPVOID值的数组，并将它初始化为0，然后由系统将它与线程联系起来。如图21-1所示，每个线程均得到它自己的数组，数组中的每个PVOID可以存储任何值。

在能够将信息存储在线程的PVOID数组中之前，必须知道数组中的哪个索引可供使用，这就是前面调用TlsAlloc所要达到的目的。按照设计概念， TlsAlloc为你保留了一个索引。如果TlsAlloc返回索引3，那么就说明目前在进程中运行的每个线程中均为你保留了索引 3，而且在将来创建的线程中也保留了索引3。

China-pub.com
下载

若要将一个值放入线程的数组中，可以调用TlsSetValue函数：

```
BOOL TlsSetValue(
   DWORD dwTlsIndex,
   PVOID pvTlsValue);
```

该函数将一个PVOID值（用pvTlsValue参数标识）放入线程的数组中由dwTlsIndex参数标识的索引处。PvTlsValue的值与调用TlsSetValue的线程相联系。如果调用成功，便返回TRUE。

线程在调用TlsSetValue时，可以改变它自己的数组。但是它不能为另一个线程设置TLS值。我希望有另一个Tls函数能够用于使一个线程将数据存储到另一个线程的数组中，但是不存在这样一个函数。目前，将数据从一个线程传递到另一个线程的唯一方法是，将单个值传递给CreateThread或_beginthreadex，然后该函数将该值作为唯一的参数传递给线程的函数。

当调用TlsSetValue时，始终都应该传递较早的时候调用的 TlsAlloc函数返回的索引。Microsoft设计的这些函数能够尽快地运行，在运行时，将放弃错误检查。如果传递的索引是调用TlsAlloc时从未分配的，那么系统将设法把该值存储在线程的数组中，而不进行任何错误检查。

若要从线程的数组中检索一个值，可以调用TlsGetValue：

```
PVOID TlsGetValue(DWORD dwTlsIndex);
```

该函数返回的值将与索引 dwTlsIndex处的TLS时隙联系起来。与 TlsSetValue一样，TlsGetValue只查看属于调用线程的数组。还有， TlsGetValue并不执行任何测试，以确定传递的索引的有效性。

当在所有线程中不再需要保留TLS时隙的位置的时候，应该调用TlsFree：

```
BOOL TlsFree(DWORD dwTlsIndex);
```

该函数简单地告诉系统该时隙不再需要加以保留。由进程的位标志数组管理的INUSE标志再次被设置为FREE。如果线程在后来调用TlsAlloc函数，那么将来就分配该INUSE标志。如果TlsFree函数运行成功，该函数将返回TRUE。如果试图释放一个没有分配的时隙，将产生一个错误。

使用动态TLS

通常情况下，如果 DLL使用TLS，那么当它用 DLL_PROCESS_ATTACH标志调用它的DllMain函数时，它也调用TlsAlloc。当它用DLL_PROCESS_DETACH调用DllMain函数时，它就调用TlsFree。对TlsSetValue和TlsGetValue的调用很可能是在调用DLL中包含的函数时进行的。

将TLS添加给应用程序的方法之一是在需要它时进行添加。例如，你的DLL中可能有一个运行方式类似strtok的函数。第一次调用这个函数时，线程传递一个指向40字节的结构的指针。必须保存这个结构，这样，将来调用函数时就可以引用它。可以像下面这样对你的函数进行编码：

```
DWORD g_dwTlsIndex; // Assume that this is initialized
                    // with the result of a call to TlsAlloc.

.
.
.

void MyFunction(PSOMESTRUCT pSomeStruct) {
   if (pSomeStruct != NULL) {
      // The caller is priming this function.

      // See if we already allocated space to save the data.
      if (TlsGetValue(g_dwTlsIndex) == NULL) {
         // Space was never allocated. This is the first
         // time this function has ever been called by this thread.
         TlsSetValue(g_dwTlsIndex,
```

China-pub.com 下载

```
            HeapAlloc(GetProcessHeap(), 0, sizeof(*pSomeStruct));
      }

      // Memory already exists for the data;
      // save the newly passed values.
      memcpy(TlsGetValue(g_dwTlsIndex), pSomeStruct,
         sizeof(*pSomeStruct));

   } else {

      // The caller already primed the function. Now it
      // wants to do something with the saved data.

      // Get the address of the saved data.
      pSomeStruct = (PSOMESTRUCT) TlsGetValue(g_dwTlsIndex);

      // The saved data is pointed to by pSomeStruct; use it.

      .
      .
      .
   }
```

如果你的应用程序的线程从来不调用 MyFunction，那么也就从来不用为该线程分配内存块。

64个TLS位置看来超出了你的需要。但是请记住，应用程序可以动态地链接到若干个 DLL。第一个DLL可以分配10个TLS索引，第二个DLL可以分配5个TLS索引，依此类推。减少你需要的TLS索引始终是个好思路。减少所用 TLS索引的最好的办法，是采用前面的代码中的 MyFunction使用的那种方法。当然，可以在多个 TLS索引中保存全部40个字节，但是这样做不仅很浪费，而且使数据的操作很困难。相反，应该为数据分配一个内存块，并且像 MyFunction 那样，只将指针保存在单个TLS索引中。正如前面讲过的那样，Windows 2000允许设置1000多个TLS时隙。Microsoft增加了时隙的数量，因为许多编程人员对时隙的使用采取一种只顾自己不顾其他的态度，不给其他DLL分配时隙，从而导致它们运行失败。

前面介绍TlsAlloc函数时，只描述了该函数能够实现的 99%的功能。为了帮助了解剩下的1%的功能，让我们观察一下下面的代码段：

```
DWORD dwTlsIndex;
PVOID pvSomeValue;

.
.
.

dwTlsIndex = TlsAlloc();
TlsSetValue(dwTlsIndex, (PVOID) 12345);
TlsFree(dwTlsIndex);

// Assume that the dwTlsIndex value returned from
// this call to TlsAlloc is identical to the index
// returned by the earlier call to TlsAlloc.
dwTlsIndex = TlsAlloc();

pvSomeValue = TlsGetValue(dwTlsIndex);
```

在这个代码运行之后，你认为pvSomeValue将包含什么信息呢？包含12345？答案是包含0。TlsAlloc在返回之前，要遍历进程中的每个线程，在每个线程的数组中的新分配索引处放入 0。

China-pub.com
下载

这是很不错的，因为应用程序可能调用LoadLibrary来加载DLL，而DLL则调用TlsAlloc来分配索引，然后，线程调用FreeLibrary来删除DLL。DLL应该通过调用TlsFree来释放它的索引，但是谁知道DLL的代码将哪些值放入线程的数组中呢？接着，线程调用LoadLibrary，将另一个DLL加载到内存中。该DLL启动时也调用TlsAlloc，并获得与前面的DLL相同的索引。如果TlsAlloc没有为进程中的所有线程设置返回的索引，那么线程就可能看到一个老的值，而代码则无法正确地运行。

例如，这个新DLL可以查看对TlsGetValue函数的调用是否曾经为一个线程分配了内存，就像前面的代码段显示的那样。如果TlsAlloc没有删除每个线程的数组项目，那么第一个DLL的老数据仍然可以使用。如果线程调用MyFunction，那么MyFunction就会认为内存块已经分配，并且调用memcpy函数，将新数据拷贝到它所认为的内存块中。这可能造成灾难性的后果，不过，幸好TlsAlloc会对数组元素进行初始化，使这样的灾难永远不会发生。

21.2 静态TLS

与动态TLS一样，静态TLS也能够将数据与线程联系起来。但是，静态TLS在代码中使用起来要容易得多，因为不必调用任何函数就能够使用它。

比如说，你想要将起始时间与应用程序创建的每个线程联系起来。只需要将起始时间变量声明为下面的形式：

```
__declspec(thread) DWORD gt_dwStartTime = 0;
```

__declspec(thread)的前缀是Microsoft添加给Visual C++编译器的一个修改符。它告诉编译器，对应的变量应该放入可执行文件或DLL文件中它的自己的节中。__declspec(thread)后面的变量必须声明为函数中（或函数外）的一个全局变量或静态变量。不能声明一个类型为__declspec(thread)的局部变量。这不应该是个问题，因为局部变量总是与特定的线程相联系的。我将前缀gt_用于全局TLS变量，而将st_用于静态TLS变量。

当编译器对程序进行编译时，它将所有的TLS变量放入它们自己的节，这个节的名字是.tls。链接程序将来自所有对象模块的所有.tls节组合起来，形成结果的可执行文件或DLL文件中的一个大的.tls节。

为了使静态TLS能够运行，操作系统必须参与其操作。当你的应用程序加载到内存中时，系统要寻找你的可执行文件中的.tls节，并且动态地分配一个足够大的内存块，以便存放所有的静态TLS变量。你的应用程序中的代码每次引用其中的一个变量时，就要转换为已分配内存块中包含的一个内存位置。因此，编译器必须生成一些辅助代码来引用该静态TLS变量，这将使你的应用程序变得比较大而且运行的速度比较慢。在x86 CPU上，将为每次引用的静态TLS变量生成3个辅助机器指令。

如果在进程中创建了另一个线程，那么系统就要将它捕获并且自动分配另一个内存块，以便存放新线程的静态TLS变量。新线程只拥有对它自己的静态TLS变量的访问权，不能访问属于其他线程的TLS变量。

这就是静态TLS变量如何运行的基本情况。现在让我们来看一看DLL的情况。你的应用程序很可能要使用静态TLS变量，并且链接到也想使用静态TLS变量的一个DLL。当系统加载你的应用程序时，它首先要确定应用程序的.tls节的大小，并将这个值与你的应用程序链接的DLL中的任何.tls节的大小相加。当在你的进程中创建线程时，系统自动分配足够大的内存块来存放应用程序需要的所有TLS变量和所有隐含链接的DLL。

China-pub.com 下载

下面让我们来看一下当应用程序调用LoadLibrary，以便链接到也包含静态TLS变量的一个DLL时，将会发生什么情况。系统必须查看进程中已经存在的所有线程，并扩大它们的TLS内存块，以便适应新DLL对内存的需求。另外，如果调用FreeLibrary来释放包含静态TLS变量的DLL，那么与进程中的每个线程相关的的内存块应该被压缩。

对于操作系统来说，这样的管理任务太重了。虽然系统允许包含静态TLS变量的库在运行期进行显式加载，但是TLS数据没有进行相应的初始化。如果试图访问这些数据，就可能导致访问违规。这是使用静态TLS的唯一不足之处。当使用动态TLS时，不会出现这个问题。使用动态TLS的库可以在运行期进行加载，并且可以在运行期释放，根本不会产生任何问题。

China-pub.com
下载

第22章　插入DLL和挂接API

在Microsoft Windows中，每个进程都有它自己的私有地址空间。当使用指针来引用内存时，指针的值将引用你自己进程的地址空间中的一个内存地址。你的进程不能创建一个其引用属于另一个进程的内存指针。因此，如果你的进程存在一个错误，改写了一个随机地址上的内存，那么这个错误不会影响另一个进程使用的内存。

Windows 98　在Windows 98下运行的各个进程共享2 GB的地址空间，该地址空间从0x80000000至0xFFFFFFFF。只有内存映像文件和系统组件才能映射到这个区域。详细说明参见第13、14章和第17章的内容。

独立的地址空间对于编程人员和用户来说都是非常有利的。对于编程人员来说，系统更容易捕获随意的内存读取和写入操作。对于用户来说，操作系统将变得更加健壮，因为一个应用程序无法破坏另一个进程或操作系统的运行。当然，操作系统的这个健壮特性是要付出代价的，因为要编写能够与其他进程进行通信，或者能够对其他进程进行操作的应用程序将要困难得多。

有些情况下，必须打破进程的界限，访问另一个进程的地址空间，这些情况包括：

- 当你想要为另一个进程创建的窗口建立子类时。
- 当你需要调试帮助时（例如，当你需要确定另一个进程正在使用哪个DLL时）。
- 当你想要挂接其他进程时。

本章将介绍若干种方法，可以用来将DLL插入到另一个进程的地址空间中。一旦你的DLL进入另一个进程的地址空间，就可以对另一个进程为所欲为。这一定会使你非常害怕，因此，究竟应该怎样做，要三思而后行。

22.1　插入DLL：一个例子

假设你想为由另一个进程创建的窗口建立一个子类。你可能记得，建立子类就能够改变窗口的行为特性。若要建立子类，只需要调用SetWindowLongPtr函数，改变窗口的内存块中的窗口过程地址，指向一个新的（你自己的）WndProc。Platform SDK文档说，应用程序不能为另一个进程创建的窗口建立子类。这并不完全正确。为另一个进程的窗口建立子类的关键问题与进程地址空间的边界有关。

当调用下面所示的SetWindowsLongPtr函数，建立一个窗口的子类时，你告诉系统，发送到或者显示在hwnd设定的窗口中的所有消息都应该送往MySubclassProc，而不是送往窗口的正常窗口过程：

```
SetWindowLongPtr(hwnd, GWLP_WNDPROC, MySubclassProc);
```

换句话说，当系统需要将消息发送到指定窗口的WndProc时，要查看它的地址，然后直接调用WndProc。在本例中，系统发现MySubclassProc函数的地址与窗口相关联，因此就直接调用MySubclassProc函数。

为另一个进程创建的窗口建立子类时遇到的问题是，建立子类的过程位于另一个地址空间中。图22-1显示了一个简化了的图形，说明窗口过程是如何接受消息的。进程 A正在运行，并

China-pub.com 下载

且已经创建了一个窗口。文件User32.dll被映射到进程A的地址空间中。对User32.dll文件的映射是为了接收和发送在进程A中运行的任何线程创建的任何窗口中发送和显示的消息。当User32.dll的映像发现一个消息时，它首先要确定窗口的WndProc的地址，然后调用该地址，传递窗口的句柄、消息和wParam和lParam值。当WndProc处理该消息后，User32.dll便循环运行，并等待另一个窗口消息被处理。

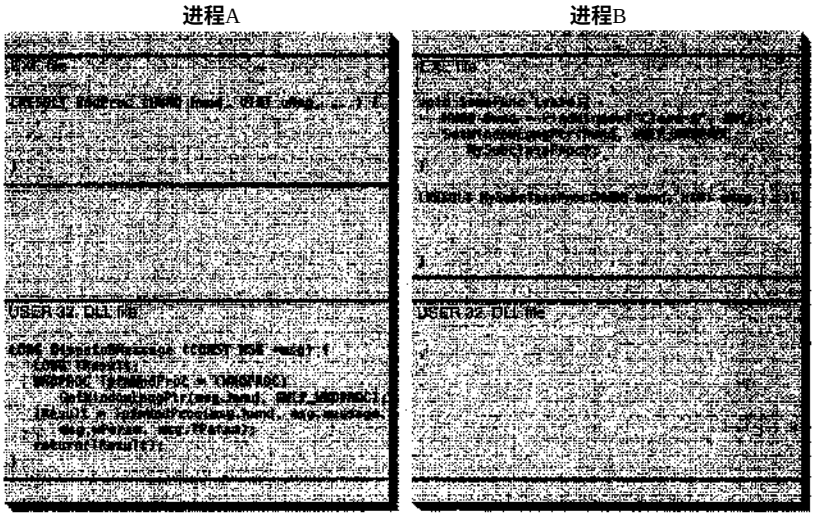


图22-1 进程B中的线程试图为进程A中的线程创建的窗口建立子类

现在假设你的进程是进程B，你想为进程A中的线程创建的窗口建立子类。你在进程B中的代码必须首先确定你想要建立子类的窗口的句柄。这个操作使用的方法很多。图22-1显示的例子只是调用FindWindow函数来获得需要的窗口。接着，进程B中的线程调用SetWindowLongPtr函数，试图改变窗口的WndProc的地址。请注意我说的“试图”二字。这个函数调用并不进行什么操作，它只是返回NULL。SetWindowLongPtr函数中的代码要查看是否有一个进程正在试图改变另一个进程创建的窗口的WndProc地址，然后将忽略这个函数的调用。

如果SetWindowLongPtr函数能够改变窗口的WndProc，那将出现什么情况呢？系统将把MySubclassProc的地址与特定的窗口关联起来。然后，当有一条消息被发送到这个窗口中时，进程A中的User32代码将检索该消息，获得MySubclassProc的地址，并试图调用这个地址。但是，这时可能遇到一个大问题。MySubclassProc将位于进程B的地址空间中，而进程A却是个活动进程。显然，如果User32想要调用该地址，它就要调用进程A的地址空间中的一个地址，这就可能造成内存访问的违规。

为了避免这个问题的产生，应该让系统知道MySubclassProc是在进程B的地址空间中，然后，在调用子类的过程之前，让系统执行一次上下文转换。Microsoft没有实现这个辅助函数功能，原因是：

- 应用程序很少需要为其他进程的线程创建的窗口建立子类。大多数应用程序只是为它们自己创建的窗口建立子类，Windows的内存结构并不阻止这种创建操作。
- 切换活动进程需要占用许多CPU时间。
- 进程B中的线程必须执行MySubclassProc中的代码。系统究竟应该使用哪个线程呢？是现有的线程，还是新线程呢？
- User32.dll怎样才能说明与窗口相关的地址是用于另一个进程中的过程，还是用于同一个

China-pub.com
下载

进程中的过程呢？

由于对这个问题的解决并没有什么万全之策，因此Microsoft决定不让SetWindowsLongPtr改变另一个进程创建的窗口过程。

不过仍然可以为另一个进程创建的窗口建立子类——只需要用另一种方法来进行这项操作。这并不是建立子类的问题，而是进程的地址空间边界的问题。如果能将你的子类过程的代码放入进程A的地址空间，就可以方便地调用SetWindowLongPtr函数，将进程A的地址传递给MySubclassProc函数。我将这个方法称为将DLL“插入”进程的地址空间。有若干种方法可以用来进行这项操作。下面将逐个介绍它们。

22.2 使用注册表来插入DLL

如果你曾经多少使用过Windows操作系统，你肯定熟悉注册表的情况。整个系统的配置都是在注册表中维护的，可以通过调整它的设置来改变系统的行为特性。将要介绍的项目是在下面的关键字中：

```
HKEY_LOCAL_MACHINE\Software\Microsoft
   \Windows NT\CurrentVersion\Windows\AppInit_DLLs
```

Windows 98 Windows 98将忽略注册表的这个关键字。在Windows 98下，无法使用该方法插入DLL。

图22-2显示了使用Registry Editor（注册表编辑器）时该关键字中的各个项目的形式。该关键字的值包含一个DLL文件名或者一组DLL文件名（用空格或逗号隔开）。由于空格用来将文件名隔开，因此必须避免使用包含空格的文件名。列出的第一个DLL文件名可以包含一个路径，但是包含路径的其他DLL均被忽略。由于这个原因，最好将你的DLL放入Windows的系统目录中，这样就不必设定路径。在窗口中，我将该值设置为单个DLL路径名C:\MyLib.dll。

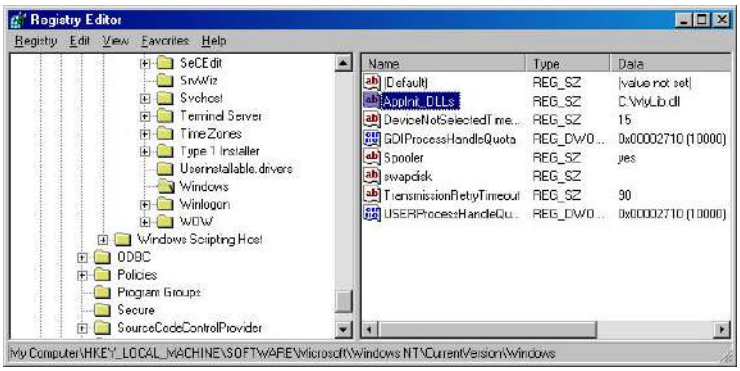


图22-2 注册表窗口

当重新启动计算机及Windows进行初始化时，系统将保存这个关键字的值。然后，当User32.dll库被映射到进程中时，它将接收到一个DLL_PROCESS_ATTACH通知。当这个通知被处理时，User32.dll便检索保存的这个关键字中的值，并且为字符串中指定的每个DLL调用LoadLibrary函数。当每个库被加载时，便调用与该库相关的DllMain函数，其fdwReason的值是DLL_PROCESS_ATTACH，这样，每个库就能够对自己进行初始化。由于插入的DLL在进程的寿命期中早早地就进行了加载，因此在调用函数时应该格外小心。调用kernel32.dll中的函数时应该不会出现什么问题，不过调用其他DLL中的函数时就可能产生一些问题。User32.dll并

China-pub.com 下载

不检查每个库是否已经加载成功，或者初始化是否取得成功。

在插入DLL时所用的所有方法中，这是最容易的一种方法。要做的工作只是将一个值添加到一个已经存在的注册表关键字中。不过这种方法也有它的某些不足：

- 由于系统在初始化时要读取这个关键字的值，因此在修改这个值后必须重新启动你的计算机——即使退出后再登录，也不行。当然，如果从这个关键字的值中删除DLL，那么在计算机重新启动之前，系统不会停止对库的映射操作。
- 你的DLL只会映射到使用User32.dll的进程中。所有基于GUI的应用程序均使用User32.dll，不过大多数基于CUI的应用程序并不使用它。因此，如果需要将DLL插入编译器或链接程序，这种方法将不起作用。
- 你的DLL将被映射到每个基于GUI的应用程序中，但是必须将你的库插入一个或几个进程中。你的DLL映射到的进程越多，“容器”进程崩溃的可能性就越大。毕竟在这些进程中运行的线程是在执行你的代码。如果你的代码进入一个无限循环，或者访问的内存不正确，就会影响代码运行时所在进程的行为特性和健壮性。因此，最好将你的库插入尽可能少的进程中。
- 你的DLL将被映射到每个基于GUI的应用程序中。这与上面的问题相类似。在理想的情况下，你的DLL只应该映射到需要的进程中，同时，它应该以尽可能少的时间映射到这些进程中。假设在用户调用你的应用程序时你想要建立WordPad的主窗口的子类。在用户调用你的应用程序之前，你的DLL不必映射到WordPad的地址空间中。如果用户后来决定终止你的应用程序的运行，那么你必须撤消WordPad的主窗口。在这种情况下，你的DLL将不再需要被插入WordPad的地址空间。最好是仅在必要时保持DLL的插入状态。

22.3 使用Windows挂钩来插入DLL

可以使用挂钩将DLL插入进程的地址空间。为了使挂钩能够像它们在16位Windows中那样工作，Microsoft不得不设计了一种方法，使得DLL能够插入另一个进程的地址空间中。

下面让我们来看一个例子。进程A（类似Microsoft Spy++的一个实用程序）安装了一个挂钩WN_GETMESSAGE，以便查看系统中的各个窗口处理的消息。该挂钩是通过调用下面的SetWindowsHookEx函数来安装的：

```
HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
   hinstDll, 0);
```

第一个参数WH_GETMESSAGE用于指明要安装的挂钩的类型。第二个参数GetMsgProc用于指明窗口准备处理一个消息时系统应该调用的函数的地址（在你的地址空间中）。第三个参数hinstDll用于指明包含GetMsgProc函数的DLL。在Windows中，DLL的hinstDll的值用于标识DLL被映射到的进程的地址空间中的虚拟内存地址。最后一个参数0用于指明要挂接的线程。对于一个线程来说，它可以调用SetWindowsHookEx函数，传递系统中的另一个线程的ID。通过为这个参数传递0，就告诉系统说，我们想要挂接系统中的所有GUI线程。

现在让我们来看一看将会发生什么情况：

1) 进程B中的一个线程准备将一条消息发送到一个窗口。

2) 系统查看该线程上是否已经安装了WH_GETMESSAGE挂钩。

3) 系统查看包含GetMsgProc函数的DLL是否被映射到进程B的地址空间中。

4) 如果该DLL尚未被映射，系统将强制该DLL映射到进程B的地址空间，并且将进程B中的DLL映像的自动跟踪计数递增1。

China-pub.com
下载

5) 当DLL的hinstDll用于进程B时，系统查看该函数，并检查该 DLL的hinstDll是否与它用于进程A时所处的位置相同。

如果两个hinstDll是在相同的位置上，那么 GetMsgProc函数的内存地址在两个进程的地址空间中的位置也是相同的。在这种情况下，系统只需要调用进程 A的地址空间中的GetMsgProc函数即可。

如果hinstDll的位置不同，那么系统必须确定进程B的地址空间中GetMsgProc函数的虚拟内存地址。这个地址可以使用下面的公式来确定：

```
GetMsgProc B = hinstDll B + (GetMsgProc A - hinstDll A)
```

将GetMsgProc A的地址减去hinstDll A的地址，就可以得到 GetMsgProc函数的地址位移（以字节为计量单位）。将这个位移与hinstDll B的地址相加，就得出GetMsgProc函数在用于进程B的地址空间中该DLL的映像时它的位置。

6) 系统将进程B中的DLL映像的自动跟踪计数递增1。

7) 系统调用进程B的地址空间中的GetMsgProc函数。

8) 当GetMsgProc函数返回时，系统将进程B中的DLL映像的自动跟踪计数递减1。

注意，当系统插入或者映射包含挂钩过滤器函数的 DLL时，整个DLL均被映射，而不只是挂钩过滤器函数被映射。这意味着 DLL中包含的任何一个函数或所有函数现在都存在，并且可以从进程B的环境下运行的线程中调用。

若要为另一个进程中的线程创建的窗口建立子类，首先可以在创建该窗口的挂钩上设置一个WH_GETMESSAGE挂钩，然后，当GetMsgProc函数被调用时，调用SetWindowLongPtr函数来建立窗口的子类。当然，子类的过程必须与 GetMsgProc函数位于同一个DLL中。

与插入DLL的注册表方法不同，这个方法允许你在另一个进程的地址空间中不再需要 DLL时删除该DLL的映像，方法是调用下面的函数：

```
BOOL UnhookWindowsHookEx(HHOOK hhook);
```

当一个线程调用UnhookWindowsHookEx函数时，系统将遍历它必须将 DLL插入到的各个进程的内部列表，并且对 DLL的自动跟踪计数进行递减。当自动跟踪计数递减为 0时，DLL就自动从进程的地址空间中被删除。应该记得，就在系统调用 GetMsgProc函数之前，它对DLL的自动跟踪计数进行了递增（见上面的第6个步骤）。这可以防止产生内存访问违规。如果该自动跟踪计数没有递增，那么当进程 B的线程试图执行GetMsgProc函数中的代码时，系统中运行的另一个线程就可以调用UnlookWindowsHookEx函数。

这一切意味着不能撤消该窗口的子类并且立即撤消该挂钩。该挂钩必须在该子类的寿命期内保持有效状态。

桌面项目位置保存器实用程序

清单22-2中列出的DIPS.exe应用程序使用窗口挂钩将一个DLL插入Explorer.exe的地址空间。该应用程序和DLL的源代码和资源文件均位于本书所附光盘的22-DIPS和22-DIPSlib目录下。

我基本上将我的计算机用于与商务有关的操作，我发现 1152 x 864的屏幕分辨率最适合我。但是在计算机上玩游戏时，大多数游戏设计时使用的分辨率是 640 x 480。因此，当我想要玩游戏时，我打开控制面板，使用 Display小应用程序，将分辨率改为 640 x 480。不玩游戏时，我又使用Display小应用程序将分辨率重新改为1152 x 864。

使用这种方法在运行过程中改变显示器的分辨率是非常麻烦的，但是它是 Windows的一个受欢迎的特性。不过我忽略了改变显示器分辨率时的一个问题，那就是桌面图标无法记住它原

China-pub.com 下载

来的位置。我的桌面上有若干个图标，可以立即访问各个应用程序，并可打开经常使用的文件。当改变显示器的分辨率时，桌面窗口便改变其大小，我的图标重新安排其位置，使我无法找到我要的东西。然后，当我将显示器的分辨率改为原来的样子时，我的所有图标又重新安排其位置，采用一种新的顺序。为了解决这个问题，我不得不用手工将桌面上的所有图标重新改为我喜欢的样子。真是烦死人了。

我非常讨厌用手工方式改变这些图标的位置，因此创建了桌面项目位置保存器实用程序DIPS。DIPS包含一个很小的可执行文件和一个很小的DLL。当运行这个可执行文件时，就会出现图22-3所示的消息框。



图22-3 桌面项目位置保存器实用工具窗口

这个消息框显示了该实用程序如何使用的情况。当你将S作为命令行参数传递给DIPS时，它就创建下面这个注册表子关键字，并且给桌面窗口上的每个项目添加一个值：

```
HKEY_CURRENT_USER\Software\Richter\Desktop Item Position Saver
```

每个项目都有一个与它一起保存的位置值。当改变屏幕分辨率以便玩游戏之前，运行DIPS S。当玩完游戏后，将屏幕的分辨率改为原来的状态，并且运行DIPS R。这使得DIPS打开注册表子关键字，对于桌面上与注册表中保存的项目相匹配的每个项目来说，当运行DIPS S时，项目的位置将被重新设置为原来的值。

最初你可能认为，DIPS的实现是非常容易的，毕竟你只需要获得桌面的ListView控件的窗口句柄，为它发送枚举各个项目的消息，获得它们的位置，然后将这些信息保存在注册表中就行了。但是，如果进行具体操作，就会发现事情并不那么简单。问题是大多数常用的控件窗口消息，比如LVM_GETITEM和LVM_GETITEMPOSITION，不能跨越进程的边界来运行。

原因是，LVM_GETITEM消息要求你为消息的LPARAM参数传递一个LV_ITEM数据结构的地址。由于这个内存地址只对发送消息的进程有意义，接收消息的进程无法保证能够使用它。因此，为了使DIPS能够按原定的要求来工作，必须将代码插入Explorer.exe，以便将LVM_GETITEM和LVM_GETITEMPOSITION消息成功地发送到桌面的ListView控件中。

注意 可以跨越进程的边界发送窗口消息，以便与内置控件（如按钮、编辑框、静态框、组合框和列表框等）进行交互操作，但是，对一些新的常用控件不能这样做。例如，可以将一个LB_GETTEXT消息发送给另一个进程中的线程创建的列表框控件，其中的LPARAM参数指向发送方进程中的一个字符串缓冲区。这是可行的，因为Microsoft专门查看LB_GETTEXT消息是否已经发送。如果已经发送，操作系统将在内部创建内存映射文件，并且跨越进程的边界来拷贝该字符串数据。

为什么Microsoft决定对内置控件这样做而不对新的常用控件这样做呢？答案是为了实现可移植性。在16位Windows中，所有应用程序都在单个地址空间中运行，一个应用程序可以将一个LB_GETTEXT消息发送给另一个应用程序创建的窗口。为了使这些16位应用程序能够非常容易地移植到Win32中，Microsoft采取了一些辅助措施来确保跨越进程的消息发送仍然能够进行。但是16位Windows中不存在新的常用控件，因此不存在移植问题，所以Microsoft没有为常用控件采取辅助的措施。

当运行DIPS.exe时，它首先得到桌面的ListView控件的窗口句柄：

```
// The Desktop ListView window is the
// grandchild of the ProgMan window.
hwndLV = GetFirstChild(
```

China-pub.com
下载


```
GetFirstChild(FindWindow(__TEXT("ProgMan"), NULL));
```

该代码首先寻找一个窗口，它的类是ProgMan。尽管Program Manager（程序管理器）应用程序正在运行，新外壳程序仍然要创建这个类的一个窗口，以便与较老版本的 Windows设计的应用程序实现向后兼容。该ProgMan窗口拥有单个子窗口，它的类是SHELLDLL_DefView。这个子窗口也拥有单个子窗口，它的类是SysListView32。该SysListView32窗口是桌面的ListView控件窗口（顺便说一下，我是使用Spy++获得所有这些信息的）。

一旦拥有ListView的窗口句柄，通过调用GetWindowThreadProcessId函数，就能够确定创建窗口的线程的 ID。将这个 ID传递给 SetDIPSHook函数（在 DIPSLib.cpp中实现）。SetDIPSHook负责在该线程上安装一个WH_GETMESSAGE挂钩，然后调用下面的函数，以强制Windows Explorer的线程醒来：

```
PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
```

由于已经在该线程上安装了一个 WH_GETMESSAGE挂钩，因此操作系统能够自动将DIPSLib.dll文件插入Explorer的地址空间，并且调用GetMsgProc函数。该函数首先查看它是否是初次被调用，如果是，那么它就创建一个隐藏的窗口，其标题是“ Richter DIPS。”请记住，Explorer的线程正在创建这个隐藏窗口。当它进行这项操作时， DIPS.exe线程从SetDIPSHook返回，然后调用下面的函数：

```
GetMessage(&msg, NULL, 0, 0);
```

这次函数调用将使线程进入睡眠状态，直到队列中显示一条消息为止。尽管 DIPS.exe并没有创建它自己的任何窗口，但是它仍然有一个消息队列，同时，只有调用 PostThreadMessage函数，才能将消息放入该队列。如果观察 DIPSLib.cpp的GetMsgProc函数中的代码，将会发现，在对CreateDialog的调用的后面，紧接着就是对 PostThreadMessage函数的调用，该函数将使SIPS.exe线程再次醒来。线程的ID保存在SetDISPHook函数的共享变量中。

注意，我将线程的消息队列用于线程的同步。这样做绝对没有什么错误，并且有时能够比使用各种内核对象（如互斥对象、信标和事件等）更容易实现线程的同步。 Windows拥有丰富的API，应该充分利用它们。

当DIPS可执行文件中的线程醒来时，它知道服务器对话框已经创建，并调用 FindWindow函数来获得窗口的句柄。这时可以使用窗口消息在客户机（ DIPS应用程序）与服务器（隐藏的对话框）之间进行通信。由于在 Windows Explorer的进程环境中运行的一个线程创建了这个对话框，因此在使用Windows Explorer时将会遇到一些限制。

若要让对话框保存或者还原桌面图标的位置，只需要发送一条消息：

```
// Tell the DIPS window which ListView window to manipulate
// and whether the items should be saved or restored.
SendMessage(hwndDIPS, WM_APP, (WPARAM) hwndLV, fSave);
```

我对该对话框的过程进行了编码，以便查找 WM_APP消息。当它收到该消息时，WPARAM参数就会指明被操作的ListView控件的句柄，而LPARAM参数则是个布尔值，用于指明当前项目的位置是否应该保存在注册表中，或者指明是否应该根据从注册表中读取的保存信息来改变项目的位置。

由于我使用SendMessage而不是Postmessage，因此该函数直到运行完成才返回。如果愿意的话，可以将消息添加给对话框的过程，使该程序能够进一步控制 Explorer的进程。当完成与对话框的通信时，并且（因此）想要终止服务器的运行时，我发送了一个 WM_CLOSE消息，告诉对话框将自己关闭。

China-pub.com 下载

最后，就在DIPS应用程序终止运行之前，它再次调用 SetDIPSHook函数，但是传递0作为线程的ID。0是个标记值，用于告诉函数撤消 WN_GETMESSAGE挂钩。当该挂钩被卸载时，操作系统自动从Explorer的地址空间中卸载DIPSLib.dll文件。对话框首先撤消，然后卸载挂钩，这一点很重要，否则对话框接收到的下一个消息将会导致 Explorer的线程引发一次访问违规。如果发生这种情况，操作系统就会终止 Explorer的运行。当使用DLL的插入操作时，必须非常小心。

清单22-1 DIPS实用程序



Dips.cpp

```
/******************************************************************************
Module:  DIPS.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-DIPSLib\DIPSLib.h"


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_DIPS);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDC_SAVE:
      case IDC_RESTORE:
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////
BOOL WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
```

China-pub.com
下载


```
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Convert command-line character to uppercase.
   CharUpperBuff(pszCmdLine, 1);
   TCHAR cWhatToDo = pszCmdLine[0];

   if ((cWhatToDo != TEXT('S')) && (cWhatToDo != TEXT('R'))) {

      // An invalid command-line argument; prompt the user.
      cWhatToDo = 0;
   }

   if (cWhatToDo == 0) {
      // No command-line argument was used to tell us what to
      // do; show usage dialog box and prompt the user.
      switch (DialogBox(hinstExe, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc)) {
         case IDC_SAVE:
            cWhatToDo = TEXT('S');
            break;

         case IDC_RESTORE:
            cWhatToDo = TEXT('R');
            break;
      }
   }

   if (cWhatToDo == 0) {
      // The user doesn't want to do anything.
      return(0);
   }
   // The Desktop ListView window is the grandchild of the ProgMan window.
   HWND hwndLV = GetFirstChild(GetFirstChild(
      FindWindow(TEXT("ProgMan"), NULL)));
   chASSERT(IsWindow(hwndLV));

   // Set hook that injects our DLL into the Explorer's address space. After
   // setting the hook, the DIPS hidden modeless dialog box is created. We
   // send messages to this window to tell it what we want it to do.
   chVERIFY(SetDIPSHook(GetWindowThreadProcessId(hwndLV, NULL)));

   // Wait for the DIPS server window to be created.
   MSG msg;
   GetMessage(&msg, NULL, 0, 0);

   // Find the handle of the hidden dialog box window.
```

China-pub.com 下载

```
   HWND hwndDIPS = FindWindow(NULL, TEXT("Richter DIPS"));

   // Make sure that the window was created.
   chASSERT(IsWindow(hwndDIPS));

   // Tell the DIPS window which ListView window to manipulate
   // and whether the items should be saved or restored.
   SendMessage(hwndDIPS, WM_APP, (WPARAM) hwndLV, (cWhatToDo == TEXT('S')));

   // Tell the DIPS window to destroy itself. Use SendMessage
   // instead of PostMessage so that we know the window is
   // destroyed before the hook is removed.
   SendMessage(hwndDIPS, WM_CLOSE, 0, 0);

   // Make sure that the window was destroyed.
   chASSERT(!IsWindow(hwndDIPS));

   // Unhook the DLL, removing the DIPS dialog box procedure
   // from the Explorer's address space.
   SetDIPSHook(0);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

DIPSLib.cpp

```
/******************************************************************************
Module:  DIPSLib.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <CommCtrl.h>

#define DIPSLIBAPI __declspec(dllexport)
#include "DIPSLib.h"
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


#ifdef _DEBUG
// This function forces the debugger to be invoked
void ForceDebugBreak() {
   __try { DebugBreak(); }
   __except(UnhandledExceptionFilter(GetExceptionInformation())) { }
}
#else
#define ForceDebugBreak()
#endif
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////


// Forward references
LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam);

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);


//////////////////////////////////////////////////////////////////////////////


// Instruct the compiler to put the g_hhook data variable in
// its own data section called Shared. We then instruct the
// linker that we want to share the data in this section
// with all instances of this application.
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
DWORD g_dwThreadIdDIPS = 0;
#pragma data_seg()

// Instruct the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/section:Shared,rws")


//////////////////////////////////////////////////////////////////////////////


// Nonshared variables
HINSTANCE g_hinstDll = NULL;


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   switch (fdwReason) {

      case DLL_PROCESS_ATTACH:
         // DLL is attaching to the address space of the current process.
         g_hinstDll = hinstDll;
         break;

      case DLL_THREAD_ATTACH:
         // A new thread is being created in the current process.
         break;

      case DLL_THREAD_DETACH:
         // A thread is exiting cleanly.
         break;

      case DLL_PROCESS_DETACH:
         // The calling process is detaching the DLL from its address space.
```

China-pub.com 下载

```
         break;
      }
      return(TRUE);
   }

   ////////////////////////////////////////////////////////////////////////////

   BOOL WINAPI SetDIPSHook(DWORD dwThreadId) {

      BOOL fOk = FALSE;

      if (dwThreadId != 0) {
         // Make sure that the hook is not already installed.
         chASSERT(g_hhook == NULL);

         // Save our thread ID in a shared variable so that our GetMsgProc
         // function can post a message back to the thread when the server
         // window has been created.
         g_dwThreadIdDIPS = GetCurrentThreadId();

         // Install the hook on the specified thread
         g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, g_hinstDll,
            dwThreadId);

         fOk = (g_hhook != NULL);
         if (fOk) {
            // The hook was installed successfully; force a benign message to
            // the thread's queue so that the hook function gets called.
            fOk = PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
         }
      } else {

         // Make sure that a hook has been installed.
         chASSERT(g_hhook != NULL);
         fOk = UnhookWindowsHookEx(g_hhook);
         g_hhook = NULL;
      }

      return(fOk);
   }

   ////////////////////////////////////////////////////////////////////////////

   LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam) {

      static BOOL fFirstTime = TRUE;
   if (fFirstTime) {
      // The DLL just got injected.
      fFirstTime = FALSE;

      // Uncomment the line below to invoke the debugger
      // on the process that just got the injected DLL.
      // ForceDebugBreak();

      // Create the DTIS Server window to handle the client request.
```

China-pub.com
下载


```
      CreateDialog(g_hinstDll, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc);

      // Tell the DIPS application that the server is up
      // and ready to handle requests.
      PostThreadMessage(g_dwThreadIdDIPS, WM_NULL, 0, 0);
   }

   return(CallNextHookEx(g_hhook, nCode, wParam, lParam));
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnClose(HWND hwnd) {

   DestroyWindow(hwnd);
}

//////////////////////////////////////////////////////////////////////////////

static const TCHAR g_szRegSubKey[] =
   TEXT("Software\\Richter\\Desktop Item Position Saver");

//////////////////////////////////////////////////////////////////////////////

void SaveListViewItemPositions(HWND hwndLV) {

   int nMaxItems = ListView_GetItemCount(hwndLV);

   // When saving new positions, delete the old position
   // information that is currently in the registry.
   LONG l = RegDeleteKey(HKEY_CURRENT_USER, g_szRegSubKey);

   // Create the registry key to hold the info
   HKEY hkey;
   l = RegCreateKeyEx(HKEY_CURRENT_USER, g_szRegSubKey, 0, NULL,
      REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hkey, NULL);
   chASSERT(l == ERROR_SUCCESS);

   for (int nItem = 0; nItem < nMaxItems; nItem++) {

      // Get the name and position of a ListView item.
      TCHAR szName[MAX_PATH];
      ListView_GetItemText(hwndLV, nItem, 0, szName, chDIMOF(szName));

      POINT pt;
      ListView_GetItemPosition(hwndLV, nItem, &pt);

      // Save the name and position in the registry.
      l = RegSetValueEx(hkey, szName, 0, REG_BINARY, (PBYTE) &pt, sizeof(pt));
      chASSERT(l == ERROR_SUCCESS);
   }
   RegCloseKey(hkey);
}

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
void RestoreListViewItemPositions(HWND hwndLV) {

   HKEY hkey;
   LONG l = RegOpenKeyEx(HKEY_CURRENT_USER, g_szRegSubKey,
      0, KEY_QUERY_VALUE, &hkey);
   if (l == ERROR_SUCCESS) {

      // If the ListView has AutoArrange on, temporarily turn it off.
      DWORD dwStyle = GetWindowStyle(hwndLV);
      if (dwStyle & LVS_AUTOARRANGE)
         SetWindowLong(hwndLV, GWL_STYLE, dwStyle & ~LVS_AUTOARRANGE);

      l = NO_ERROR;
      for (int nIndex = 0; l != ERROR_NO_MORE_ITEMS; nIndex++) {
         TCHAR szName[MAX_PATH];
         DWORD cbValueName = chDIMOF(szName);

         POINT pt;
         DWORD cbData = sizeof(pt), nItem;

         // Read a value name and position from the registry.
         DWORD dwType;
         l = RegEnumValue(hkey, nIndex, szName, &cbValueName,
            NULL, &dwType, (PBYTE) &pt, &cbData);

         if (l == ERROR_NO_MORE_ITEMS)
            continue;

         if ((dwType == REG_BINARY) && (cbData == sizeof(pt))) {
            // The value is something that we recognize; try to find
            // an item in the ListView control that matches the name.
            LV_FINDINFO lvfi;
            lvfi.flags = LVFI_STRING;
            lvfi.psz = szName;
            nItem = ListView_FindItem(hwndLV, -1, &lvfi);
            if (nItem != -1) {
               // We found a match; change the item's position.
               ListView_SetItemPosition(hwndLV, nItem, pt.x, pt.y);
            }
         }
      }
      // Turn AutoArrange back on if it was originally on.
      SetWindowLong(hwndLV, GWL_STYLE, dwStyle);
      RegCloseKey(hkey);
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_CLOSE, Dlg_OnClose);

      case WM_APP:
```

China-pub.com
下载


```
      // Uncomment the line below to invoke the debugger
      // on the process that just got the injected DLL.
      // ForceDebugBreak();

      if (lParam)
         SaveListViewItemPositions((HWND) wParam);
      else
         RestoreListViewItemPositions((HWND) wParam);
      break;
}
   return(FALSE);
}


//////////////////////////////// End of File /////////////////////////////////
```

DIPSLib.h

```
/******************************************************************************
Module:  DIPSLib.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#if !defined(DIPSLIBAPI)
#define DIPSLIBAPI __declspec(dllimport)
#endif


////////////////////////////////////////////////////////////////////////////////


// External function prototypes
DIPSLIBAPI BOOL WINAPI SetDIPSHook(DWORD dwThreadId);


//////////////////////////////// End of File /////////////////////////////////
```

DIPSLib.rc

```
//Microsoft Developer Studio generated resource script.
//
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
////////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_DIPS DIALOG DISCARDABLE  0, 0, 132, 13
STYLE WS_CAPTION
CAPTION "Richter DIPS"
FONT 8, "MS Sans Serif"
BEGIN
END

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

China-pub.com
下载

22.4 使用远程线程来插入DLL

插入DLL的第三种方法是使用远程线程。这种方法具有更大的灵活性。它要求你懂得若干个Windows特性、如进程、线程、线程同步、虚拟内存管理、 DLL和Unicode等（如果对这些特性不清楚，请参阅本书中的有关章节）。Windows的大多数函数允许进程只对自己进行操作。这是很好的一个特性，因为它能够防止一个进程破坏另一个进程的运行。但是，有些函数却允许一个进程对另一个进程进行操作。这些函数大部分最初是为调试程序和其他工具设计的。不过任何函数都可以调用这些函数。

这个DLL插入方法基本上要求目标进程中的线程调用 LoadLibrary函数来加载必要的DLL。由于除了自己进程中的线程外，我们无法方便地控制其他进程中的线程，因此这种解决方案要求我们在目标进程中创建一个新线程。由于是自己创建这个线程，因此我们能够控制它执行什么代码。幸好，Windows提供了一个称为CreateRemoteThread的函数，使我们能够非常容易地在另一个进程中创建线程：

```
HANDLE CreateRemoteThread(
   HANDLE hProcess,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD fdwCreate,
   PDWORD pdwThreadId);
```

CreateRemoteThread与CreateThread很相似，差别在于它增加了一个参数 hProcess。该参数指明拥有新创建线程的进程。参数 pfnStartAddr指明线程函数的内存地址。当然，该内存地址与远程进程是相关的。线程函数的代码不能位于你自己进程的地址空间中。

注意 在Windows 2000中，更常用的函数CreateThread是在内部以下面的形式来实现的：

```
HANDLE CreateThread(PSECURITY_ATTRIBUTES psa, DWORD dwStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam,
   DWORD fdwCreate, PDWORD pdwThreadID) {

   return(CreateRemoteThread(GetCurrentProcess(), psa, dwStackSize,
      pfnStartAddr, pvParam, fdwCreate, pdwThreadID));
}
```

Windows 98 在Windows 98中，CreateRemoteThread函数不存在有用的实现代码，它只是返回NULL。调用GetLastError函数将返回ERROR_CALL_NOT_IMPLEMENTED（CreateThread函数包含用于在调用进程中创建线程的完整的实现代码）。由于CreateRemoteThread没有实现，因此，在 Windows 98下，不能使用本方法来插入DLL。

好了，现在你已经知道如何在另一个进程中创建线程了，但是，如何才能让该线程加载我们的DLL呢？答案很简单，那就是需要该线程调用LoadLibrary函数：

```
HINSTANCE LoadLibrary(PCTSTR pszLibFile);
```

如果观察WinBase.h文件中的LoadLibrary函数，你将会发现下面的代码：

```
HINSTANCE WINAPI LoadLibraryA(LPCSTR  pszLibFileName);
HINSTANCE WINAPI LoadLibraryW(LPCWSTR pszLibFileName);
```

China-pub.com 下载

```
#ifdef UNICODE
#define LoadLibrary  LoadLibraryW
#else
#define LoadLibrary  LoadLibraryA
#endif // !UNICODE
```

实际上有两个LoadLibrary函数，即LoadLibraryA和LoadLibraryW。这两个函数之间存在的唯一差别是，传递给函数的参数类型不同。如果将库的文件名作为ANSI字符串来存储，那么必须调用LoadLibraryA（A是指ANSI）。如果将文件名作为Unicode字符串来存储，那么必须调用LoadLibraryW（W是指通配符）。不存在单个LoadLibrary的情况，只有LoadLibraryA和LoadLibraryW。对于大多数应用程序来说，LoadLibrary宏可以扩展为LoadLibraryA。

幸好LoadLibrary函数的原型与一个线程函数的原型是相同的。下面是一个线程函数的原型：

```
DWORD WINAPI ThreadFunc(PVOID pvParam);
```

这两个函数的原型并不完全相同，不过它们非常相似。两个函数都接受单个参数，并且都返回一个值。另外，两个函数都使用相同的调用规则。这是非常幸运的，因为我们要做的事情是创建一个新线程，并且使线程函数的地址成为LoadLibraryA或LoadLibraryW函数的地址。本质上，我们必须进行的操作是执行类似下面的一行代码：

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryA, "C:\\MyLib.dll", 0, NULL);
```

或者，如果喜欢Unicode，则执行下面这行代码：

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryW, L"C:\\MyLib.dll", 0, NULL);
```

当在远程进程中创建新线程时，该线程将立即调用LoadLibraryA（或LoadLibraryW）函数，并将DLL的路径名的地址传递给它。这是非常容易的。但是这里存在另外两个问题。

第一个问题是，不能像我在上面展示的那样，将LoadLibraryA或LoadLibraryW作为第四个参数传递给CreateRemoteThread。原因很简单。当你编译或者链接一个程序时，产生的二进制代码包含一个输入节（第19章中做了介绍）。这一节由一系列输入函数的形式替换程序（thunk）组成。所以，当你的代码调用一个函数如LoadLibraryA时，链接程序将生成一个对你模块的输入节中的形实替换程序的调用。接着，该形实替换程序便转移到实际的函数。

如果在对CreateRemoteThread的调用中使用一个对LoadLibraryA的直接引用，这将在你的模块的输入节中转换成LoadLibraryA的形实替换程序的地址。将形实替换程序的地址作为远程线程的起始地址来传递，会导致远程线程开始执行一些令人莫名其妙的东西。其结果很可能造成访问违规。若要强制直接调用LoadLibraryA函数，避开形实替换程序，必须通过调用GetProcAddress函数，获取LoadLibraryA的准确内存位置。

对CreateRemoteThread进行调用的前提是，Kernel32.dll已经被同时映射到本地和远程进程的地址空间中。每个应用程序都需要Kernel32.dll，根据我的经验，系统将Kernel32.dll映射到每个进程的同一个地址。因此，必须调用下面的CreateRemoteThread函数：

```
// Get the real address of LoadLibraryA in Kernel32.dll.
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   pfnThreadRtn, "C:\\MyLib.dll", 0, NULL);
```

China-pub.com
下载

或者，如果喜欢Unicode的话，调用下面的函数：

```
// Get the real address of LoadLibraryW in Kernel32.dll.
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

好了，这就解决了第一个问题。第二个问题与 DLL路径名字符串有关。字符串“ C:\\MyLib.dll ”是在调用进程的地址空间中。该字符串的地址已经被赋予新创建的远程线程，该线程将它传递给LoadLibraryA。但是，当LoadLibraryA取消对内存地址的引用时，DLL路径名字符串将不再存在，远程进程的线程就可能引发访问违规；向用户显示一个未处理的异常条件消息框，并且远程进程终止运行。记住，这是远程进程终止运行，不是你的进程终止运行。你可能成功地终止另一个进程的运行，而你的进程则继续正常地运行。

为了解决这个问题，必须将 DLL的路径名字符串放入远程进程的地址空间中。然后，当CreateRemoteThread函数被调用时，我们必须将我们放置该字符串的地址（相对于远程进程的地址）传递给它。同样，Windows提供了一个函数，即VirtualAllocEx，使得一个进程能够分配另一个进程的地址空间中的内存：

```
PVOID VirtualAllocEx(
   HANDLE hProcess,
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD flAllocationType,
   DWORD flProtect);
```

另一个函数则使我们能够释放该内存：

```
BOOL VirtualFreeEx(
   HANDLE hProcess,
   PVOID pvAddress,
   SIZE_T dwSize,
   DWORD dwFreeType);
```

这两个函数与它们的非Ex版本的函数（第15章已经做了介绍）是类似的。唯一的差别是这两个函数需要一个进程的句柄作为其第一个参数。这个句柄用于指明执行操作时所在的进程。

一旦为该字符串分配了内存，我们还需要一种方法将该字符串从我们的进程的地址空间拷贝到远程进程的地址空间中。Windows提供了一些函数，使得一个进程能够从另一个进程的地址空间中读取数据，并将数据写入另一个进程的地址空间。

```
BOOL ReadProcessMemory(
   HANDLE hProcess,
   PVOID pvAddressRemote,
   PVOID pvBufferLocal,
   DWORD dwSize,
   PDWORD pdwNumBytesRead);

BOOL WriteProcessMemory(
   HANDLE hProcess,
   PVOID pvAddressRemote,
   PVOID pvBufferLocal,
   DWORD dwSize,
   PDWORD pdwNumBytesWritten);
```

China-pub.com 下载

远程进程由hProcess参数来标识。参数pvAddressRemote用于指明远程进程中的地址，参数pvBufferLocal是本地进程中的内存地址，参数dwSize是需要传送的字节数，pdwNumBytesRead和pdwNumBytesWritten用于指明实际传送的字节数。当函数返回时，可以查看这两个参数的值。

既然已经知道了要进行操作，下面让我们将必须执行的操作步骤做一个归纳：

1) 使用VirtualAllocEx函数，分配远程进程的地址空间中的内存。

2) 使用WriteProcessMemory函数，将DLL的路径名拷贝到第一个步骤中已经分配的内存中。

3) 使用GetProcAddress函数，获取LoadLibraryA或LoadLibratyW函数的实地址（在Kernel32.dll中）。

4) 使用CreateRemoteThread函数，在远程进程中创建一个线程，它调用正确的LoadLibrary函数，为它传递第一个步骤中分配的内存的地址。

这时，DLL已经被插入远程进程的地址空间中，同时 DLL的DllMain函数接收到一个DLL_PROCESS_ATTACH通知，并且能够执行需要的代码。当DllMain函数返回时，远程线程从它对 LoadLibrary的调用返回到 BaseThreadStart函数（第 6章中已经介绍）。然后BaseThreadStart调用ExitThread，使远程线程终止运行。

现在远程进程拥有第一个步骤中分配的内存块，而DLL则仍然保留在它的地址空间中。若要将它删除，需要在远程线程退出后执行下面的步骤：

5) 使用VirtualFreeEx函数，释放第一个步骤中分配的内存。

6) 使用GetProcAddress函数，获得FreeLibrary函数的实地址（在Kernel32.dll中）。

7) 使用CreateRemoteThread函数，在远程进程中创建一个线程，它调用 FreeLibrary函数，传递远程DLL的HINSTANCE。

这就是它的基本操作步骤。这种插入 DLL的方法存在的唯一一个不足是 ,Windows 98并不支持这样的函数。只能在Windows 2000上使用这种方法。

22.4.1 Inject Library示例应用程序

清单22-2中列出的InjLib.exe应用程序使用CreateRemoteThread函数来插入DLL。该应用程序和DLL的源代码和资源文件位于本书所附光盘上的 22-InjLib和22-ImgWalk目录下。该程序使用图22-4所示的对话框来接收运行的进程ID。



图22-4 Inject Library Tester对话框

可以使用Windows 2000配有的Task Manager（任务管理器）获取进程的ID。使用这个ID，该程序将设法通过调用OpenProcess函数来打开正在运行的进程的句柄，申请相应的访问权：

```
hProcess = OpenProcess(
   PROCESS_CREATE_THREAD |   // For CreateRemoteThread
   PROCESS_VM_OPERATION  |   // For VirtualAllocEx/VirtualFreeEx
   PROCESS_VM_WRITE,         // For WriteProcessMemory
   FALSE, dwProcessId);
```

如果OpenProcess返回NULL，该应用程序就不是在允许它打开目标进程句柄的安全环境下运行。有些进程，如WinLogon、SvcHost和Csrss，是在本地系统帐户下运行的，这个帐户是已经登录的用户不能改变的。如果你有权并且激活调试安全优先级，那么就能够打开这些进程的句柄。第4章中的ProcessInfo示例应用程序展示了进行这项操作的方法。

China-pub.com
下载

如果OpenProcess函数运行成功，便使用要插入的DLL的全路径名对一个缓存进程初始化。然后InjectLib被调用，为它传递需要的远程进程的句柄和要插入的 DLL的路径名。最后，当InjectLib返回时，该程序显示一个消息框，指明DLL是否已经成功地加载到远程进程中，然后它关闭进程的句柄。这就是它的全部运行过程。

你可能在代码中发现，我专门查看了传递的进程ID是否是0。如果是0，我就调用GetCurrentProcessId函数，将进程的ID设置为InjectLib.exe自己的进程ID。这样，当InjectLib被调用时，DLL被插入到进程自己的地址空间中。这使得程序的调用比较容易。可以想象，当出现错误时，有时很难确定这些错误是在本地进程中还是在远程进程中。原先我用两个调试程序来调试我的代码，一个调试程序负责观察InjLib，另一个调试程序负责观察远程进程。结果表明这样做是很不方便的。后来我才明白，InjLib也能将DLL插入本身的程序中，也就是说插入与调用程序相同的地址空间中。这样调试代码就容易多了。

在源代码模块的顶部你会发现，InjectLib实际上是个符号，根据你编译源代码所用的方法，它可以展开为InjectLibA或InjectLibW。函数InjectLibW正是一切魔力之所在。程序的注释本身就说明了问题，当然也可以进一步补充说明。不过你将发现函数 InjectLibA比较短。它只是将ANSI DLL的路径名转换成对应的Unicode路径名，然后调用InjectLibW函数进行实际的操作。这种方法正是我在第2章中建议你使用的。它也意味着只需要使插入代码运行一次就行了。

清单22-2 InjLib示例应用程序



InjLib.cpp

```
/******************************************************************************
Module:  InjLib.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <stdio.h>
#include <tchar.h>
#include <malloc.h>        // For alloca
#include <TlHelp32.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


#ifdef UNICODE
#define InjectLib InjectLibW
#define EjectLib  EjectLibW
#else
#define InjectLib InjectLibA
#define EjectLib  EjectLibA
#endif   // !UNICODE


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
BOOL WINAPI InjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {

   BOOL fOk = FALSE; // Assume that the function fails
   HANDLE hProcess = NULL, hThread = NULL;
   PWSTR pszLibFileRemote = NULL;

   __try {
      // Get a handle for the target process.
      hProcess = OpenProcess(
         PROCESS_CREATE_THREAD     |   // For CreateRemoteThread
         PROCESS_VM_OPERATION      |   // For VirtualAllocEx/VirtualFreeEx
        PROCESS_VM_WRITE,             // For WriteProcessMemory
        FALSE, dwProcessId);
     if (hProcess == NULL) __leave;

     // Calculate the number of bytes needed for the DLL's pathname
     int cch = 1 + lstrlenW(pszLibFile);
     int cb  = cch * sizeof(WCHAR);

     // Allocate space in the remote process for the pathname
     pszLibFileRemote = (PWSTR)
        VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT, PAGE_READWRITE);
     if (pszLibFileRemote == NULL) __leave;

     // Copy the DLL's pathname to the remote process's address space
     if (!WriteProcessMemory(hProcess, pszLibFileRemote,
        (PVOID) pszLibFile, cb, NULL)) __leave;

     // Get the real address of LoadLibraryW in Kernel32.dll
     PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
        GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
     if (pfnThreadRtn == NULL) __leave;

     // Create a remote thread that calls LoadLibraryW(DLLPathname)
     hThread = CreateRemoteThread(hProcess, NULL, 0,
        pfnThreadRtn, pszLibFileRemote, 0, NULL);
     if (hThread == NULL) __leave;

     // Wait for the remote thread to terminate
     WaitForSingleObject(hThread, INFINITE);

     fOk = TRUE; // Everything executed successfully
   }
   __finally { // Now, we can clean everthing up

     // Free the remote memory that contained the DLL's pathname
     if (pszLibFileRemote != NULL)
        VirtualFreeEx(hProcess, pszLibFileRemote, 0, MEM_RELEASE);

     if (hThread  != NULL)
        CloseHandle(hThread);

     if (hProcess != NULL)
        CloseHandle(hProcess);
   }
```

China-pub.com
下载


```
   return(fOk);
}

//////////////////////////////////////////////////////////////////////////////

BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

   // Allocate a (stack) buffer for the Unicode version of the pathname
   PWSTR pszLibFileW = (PWSTR)
      _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

   // Convert the ANSI pathname to its Unicode equivalent
   wsprintfW(pszLibFileW, L"%S", pszLibFile);

   // Call the Unicode version of the function to actually do the work.
   return(InjectLibW(dwProcessId, pszLibFileW));
}

//////////////////////////////////////////////////////////////////////////////

BOOL WINAPI EjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {

   BOOL fOk = FALSE; // Assume that the function fails
   HANDLE hthSnapshot = NULL;
   HANDLE hProcess = NULL, hThread = NULL;

   __try {
      // Grab a new snapshot of the process
      hthSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwProcessId);
      if (hthSnapshot == NULL) __leave;

      // Get the HMODULE of the desired library
      MODULEENTRY32W me = { sizeof(me) };
      BOOL fFound = FALSE;
      BOOL fMoreMods = Module32FirstW(hthSnapshot, &me);
      for (; fMoreMods; fMoreMods = Module32NextW(hthSnapshot, &me)) {
         fFound = (lstrcmpiW(me.szModule,  pszLibFile) == 0) ||
                  (lstrcmpiW(me.szExePath, pszLibFile) == 0);
         if (fFound) break;
      }

      if (!fFound) __leave;

      // Get a handle for the target process.
      hProcess = OpenProcess(
         PROCESS_CREATE_THREAD
         | PROCESS_VM_OPERATION,  // For CreateRemoteThread
         FALSE, dwProcessId);
      if (hProcess == NULL) __leave;

      // Get the real address of LoadLibraryW in Kernel32.dll
      PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
         GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "FreeLibrary");
```

China-pub.com 下载

```
      if (pfnThreadRtn == NULL) __leave;

      // Create a remote thread that calls LoadLibraryW(DLLPathname)
      hThread = CreateRemoteThread(hProcess, NULL, 0,
         pfnThreadRtn, me.modBaseAddr, 0, NULL);
      if (hThread == NULL) __leave;

      // Wait for the remote thread to terminate
      WaitForSingleObject(hThread, INFINITE);

      fOk = TRUE; // Everything executed successfully
   }
   __finally { // Now we can clean everything up

      if (hthSnapshot != NULL)
         CloseHandle(hthSnapshot);

      if (hThread     != NULL)
         CloseHandle(hThread);

      if (hProcess    != NULL)
         CloseHandle(hProcess);
   }

   return(fOk);
}


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI EjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {
   // Allocate a (stack) buffer for the Unicode version of the pathname
   PWSTR pszLibFileW = (PWSTR)
      _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

   // Convert the ANSI pathname to its Unicode equivalent
   wsprintfW(pszLibFileW, L"%S", pszLibFile);

   // Call the Unicode version of the function to actually do the work.
   return(EjectLibW(dwProcessId, pszLibFileW));
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_INJLIB);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
```

China-pub.com
下载


```
switch (id) {
   case IDCANCEL:
      EndDialog(hwnd, id);
      break;

   case IDC_INJECT:
      DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
      if (dwProcessId == 0) {
         // A process ID of 0 causes everything to take place in the
         // local process; this makes things easier for debugging.
         dwProcessId = GetCurrentProcessId();
      }

      TCHAR szLibFile[MAX_PATH];
      GetModuleFileName(NULL, szLibFile, sizeof(szLibFile));
      _tcscpy(_tcsrchr(szLibFile, TEXT('\\')) + 1, TEXT("22 ImgWalk.DLL"));
         if (InjectLib(dwProcessId, szLibFile)) {
            chVERIFY(EjectLib(dwProcessId, szLibFile));
            chMB("DLL Injection/Ejection successful.");
         } else {
            chMB("DLL Injection/Ejection failed.");
         }
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   chWindows9xNotAllowed();
   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_INJLIB), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

InjLib.rc

China-pub.com 下载

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_INJLIB              ICON    DISCARDABLE     "InjLib.ico"

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
```

China-pub.com
下载


```
//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_INJLIB DIALOG DISCARDABLE  15, 24, 158, 24
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
CAPTION "Inject Library Tester"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "&Process Id (decimal):",-1,4,6,69,8
    EDITTEXT        IDC_PROCESSID,78,4,36,12,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "&Inject",IDC_INJECT,120,4,36,12,WS_GROUP
END


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_INJLIB, DIALOG
    BEGIN
        RIGHTMARGIN, 134
        BOTTOMMARGIN, 20
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

22.4.2 Image Walk DLL

清单22-3列出的ImgWalk.dll是个DLL，一旦它被插入进程的地址空间，就能够报告该进程正在使用的所有DLL（该DLL的源代码和资源文件均在本书所附光盘上的22-ImgWalk目录下）。例如，如果我首先运行Notepad，然后运行InjLib，为它传递Notepad的进程ID，InjLib将ImgWalk.dll插入Notepad的地址



图22-5 查找结果对话框

China-pub.com 下载

空间中。一旦进入该地址空间，ImgWalk便确定Notepad正在使用哪些文件映像（可执行文件和DLL），并且显示图22-5所示的消息框，它显示了查找的结果。

ImgWalk遍历进程的地址空间，查找已经映射的文件映像，反复调用VirtualQuery函数，填入一个MEMORY_BASIC_ INFORMATION结构中。运用循环的每个重复操作，ImgWalk找出一个文件路径名，并与一个字符串相连接。该字符串显示在消息框中。

```
char szBuf[MAX_PATH * 100] = { 0 };

PBYTE pb = NULL;
MEMORY_BASIC_INFORMATION mbi;
while (VirtualQuery(pb, &mbi, sizeof(mbi)) == sizeof(mbi)) {

   int nLen;
   char szModName[MAX_PATH];

   if (mbi.State == MEM_FREE)
      mbi.AllocationBase = mbi.BaseAddress;

   if ((mbi.AllocationBase == hinstDll) ||
       (mbi.AllocationBase != mbi.BaseAddress) ||
       (mbi.AllocationBase == NULL)) {

      // Do not add the module name to the list
      // if any of the following is true:
      // 1. This region contains this DLL.
      // 2. This block is NOT the beginning of a region.
      // 3. The address is NULL.
      nLen = 0;
   } else {
      nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
         szModName, chDIMOF(szModName));
   }

   if (nLen > 0) {
      wsprintfA(strchr(szBuf, 0), "\n%08X-%s",
         mbi.AllocationBase, szModName);
   }

   pb += mbi.RegionSize;
}
chMB(&szBuf[1]);
```

首先我查看区域的基地址是否与插入的DLL的基地址相匹配。如果匹配，则将nLen设置为0，这样，插入的库就不会出现在消息框中。如果不匹配，我将设法获取加载到该区域的基地址中的模块的文件名。如果nLen变量的值大于0，系统就知道该地址指明了一个已经加载的模块，同时，系统用该模块的全路径名填入szModName缓存。然后我将模块的HINSTANCE（基地址）和它的路径名与最终将显示在消息框中的szBuf字符串链接起来。当这个循环终止运行时，DLL显示了一个消息框，其内容是字符串。

清单22-3 ImgWalk.dll的源代码

ImgWalk.cpp

```
/*****************************************************************************
```

China-pub.com
下载


```
Module:  ImgWalk.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

   if (fdwReason == DLL_PROCESS_ATTACH) {
      char szBuf[MAX_PATH * 100] = { 0 };

      PBYTE pb = NULL;
      MEMORY_BASIC_INFORMATION mbi;
      while (VirtualQuery(pb, &mbi, sizeof(mbi)) == sizeof(mbi)) {

         int nLen;
         char szModName[MAX_PATH];

         if (mbi.State == MEM_FREE)
            mbi.AllocationBase = mbi.BaseAddress;

         if ((mbi.AllocationBase == hinstDll) ||
             (mbi.AllocationBase != mbi.BaseAddress) ||
             (mbi.AllocationBase == NULL)) {
            // Do not add the module name to the list
            // if any of the following is true:
            // 1. If this region contains this DLL
            // 2. If this block is NOT the beginning of a region
            // 3. If the address is NULL
            nLen = 0;
         } else {
            nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
               szModName, chDIMOF(szModName));
         }
         if (nLen > 0) {
            wsprintfA(strchr(szBuf, 0), "\n%p-%s",
               mbi.AllocationBase, szModName);
         }

         pb += mbi.RegionSize;
      }

      chMB(&szBuf[1]);
   }

   return(TRUE);
}


////////////////////////////////// End of File //////////////////////////////////
```

China-pub.com 下载

22.5 使用特洛伊DLL来插入DLL

插入DLL的另一种方法是取代你知道进程将要加载的DLL。例如，如果你知道一个进程将要加载Xyz.dll，就可以创建你自己的DLL，为它赋予相同的文件名。当然，你必须将原来的Xyz.dll改为别的什么名字。

在你的Xyz.dll中，输出的全部符号必须与原始的Xyz.dll输出的符号相同。使用函数转发器（第20章做了介绍），很容易做到这一点。虽然函数转发器使你能够非常容易地挂接某些函数，你应该避免使用这种方法，因为它不具备版本升级能力。例如，如果你取代了一个系统DLL，而Microsoft在将来增加了一些新函数，那么你的DLL将不具备它们的函数转发器。引用这些新函数的应用程序将无法加载和执行。

如果你只想在单个应用程序中使用这种方法，那么可以为你的DLL赋予一个独一无二的名字，并改变应用程序的.exe模块的输入节。更为重要的是，输入节只包含模块需要的DLL的名字。你可以仔细搜索文件中的这个输入节，并且将它改变，使加载程序加载你自己的DLL。这种方法相当不错，但是必须要非常熟悉.exe和DLL文件的格式。

22.6 将DLL作为调试程序来插入

调试程序能够对被调试的进程执行特殊的操作。当被调试进程加载时，在被调试进程的地址空间已经作好准备，但是被调试进程的主线程尚未执行任何代码之前，系统将自动将这个情况通知调试程序。这时，调试程序可以强制将某些代码插入被调试进程的地址空间中（比如使用WriteProcessMemory函数来插入），然后使被调试进程的主线程执行该代码。

这种方法要求你对被调试线程的CONTEXT结构进行操作，意味着必须编写特定CPU的代码。必须修改你的源代码，使之能够在不同的CPU平台上正确地运行。另外，必须对你想让被调试进程执行的机器语言指令进行硬编码。而且调试程序与它的被调试程序之间必须存在固定的关系。如果调试程序终止运行，Windows将自动撤消被调试进程。而你则无法阻止它。

22.7 用Windows 98上的内存映射文件插入代码

在Windows 98上插入你自己的代码是非常简单的。在Windows 98上运行的所有32位Windows应用程序均共享同样的最上面的2 GB地址空间。如果你分配这里面的某些存储器，那么该存储器在每个进程的地址空间中均可使用。若要分配2 GB以上的存储器，只要使用内存映射文件（第17章已经介绍）。可以创建一个内存映射文件，然后调用MapViewOfFile函数，使它显示出来。然后将数据填入你的地址空间区域（这是所有进程地址空间中的相同区域）。必须使用硬编码的机器语言来进行这项操作，其结果是这种解决方案很难移植到别的CPU平台。不过，如果进行这项操作，那么不必考虑不同的CPU平台，因为Windows 98只能在x86 CPU上运行。

这种方法的困难之处在于仍然必须让其他进程中的线程来执行内存映射文件中的代码。要做到这一点，需要某种方法来控制远程进程中的线程。CreateRemoteThread函数能够很好地执行这个任务，可惜Windows 98不支持该函数的运行，而我也无法提供相应的解决方案。

22.8 用CreateProcess插入代码

如果你的进程生成了你想插入代码的新进程，那么事情就会变得稍稍容易一些。原因之一是，你的进程（父进程）能够创建暂停运行的新进程。这就使你能够改变子进程的状态，而不

China-pub.com
下载

影响它的运行，因为它尚未开始运行。但是父进程也能得到子进程的主线程的句柄。使用该句柄，可以修改线程执行的代码。你可以解决上一节提到的问题，因为可以设置线程的指令指针，以便执行内存映射文件中的代码。

下面介绍一种方法，它使你的进程能够控制子进程的主线程执行什么代码：

1) 使你的进程生成暂停运行的子进程。

2) 从.exe模块的头文件中检索主线程的起始内存地址。

3) 将机器指令保存在该内存地址中。

4) 将某些硬编码的机器指令强制放入该地址中。这些指令应该调用 LoadLibrary函数来加载DLL。

5) 继续运行子进程的主线程，使该代码得以执行。

6) 将原始指令重新放入起始地址。

7) 让进程继续从起始地址开始执行，就像没有发生任何事情一样。

上面的步骤6和7要正确运行是很困难的，因为你必须修改当前正在执行的代码。不过这是可能的。

这种方法具有许多优点。首先，它在应用程序执行之前就能得到地址空间。第二，它既能在Windows 98上使用，也能在Windows 2000上使用。第三，由于你不是调试者，因此能够很容易使用插入的DLL来调试应用程序。最后，这种方法可以同时用于控制台和 GUI应用程序。

当然，这种方法也有某些不足。只有当你的代码是父进程时，才能插入 DLL。另外，这种方法当然不能跨越不同的CPU来运行，必须对不同的CPU平台进行相应的修改。

22.9 挂接API的一个示例

将DLL插入进程的地址空间是确定进程运行状况的一种很好的方法。但是，仅仅插入 DLL无法提供足够的信息，人们常常需要知道某个进程中的线程究竟是如何调用各个函数的，也可能需要修改Windows函数的功能。

例如，我知道一家公司生产的DLL是由一个数据库产品来加载的。该DLL的作用是增强和扩展数据库产品的功能。当数据库产品终止运行时，该 DLL就会收到一个 DLL_PROCESS_DETACH通知，并且只有在这时，它才执行它的所有清除代码。该DLL将调用其他DLL中的函数，以便关闭套接字连接、文件和其他资源，但是当它收到DLL_PROCESS_ DETACH通知时，进程的地址空间中的其他DLL已经收到它们的DLL_PROCESS_DETACH通知。因此，当该公司的DLL试图清除时，它调用的许多函数的运行将会失败，因为其他DLL已经撤消了初始化信息。

该公司聘请我去帮助他们解决这个问题，我建议挂接函数 ExitProcess。如你所知，调用ExitProcess将导致系统向该DLL发送DLL_PROCESS_DETACH通知。通过挂接ExitPrecess函数，我们就能确保当 ExitProcess函数被调用时，该公司的 DLL能够得到通知。这个通知将在任何DLL得到DLL_PROCESS_DETACH通知之前进来，因此进程中的所有 DLL仍然处于初始化状态中，并且能够正常运行。此时，该公司的DLL知道进程将要终止运行，并且能够成功地执行它的全部清除操作。然后，操作系统的 ExitProcess函数被调用，使所有 DLL收到它们的DLL_PROCESS_DETACH通知并进行清除操作。当该公司的 DLL收到这个通知时，它将不执行专门的清除操作，因为它已经做了它必须做的事情。

在这个例子中，插入DLL是可以随意进行的，因为数据库应用程序的设计已经允许进行这样的插入，并且它加载了公司的DLL。当该公司的DLL被加载时，它必须扫描所有已经加载的可执行模块和 DLL模块，以便找出对 ExitProcess的调用。当它发现对 ExitProcess的调用后，

China-pub.com 下载

DLL必须修改这些模块，这样，这些模块就能调用公司的DLL中的函数，而不是调用操作系统的ExitProcess函数（这个过程比想象的情况要简单的多）。一旦公司的ExitProcess替换函数（即通常所说的挂钩函数）执行它的清除代码，操作系统的ExitProcess函数（在Kernel32.dll文件中）就被调用。

这个例子显示了挂接API的一种典型用法。它用很少的代码解决了一个非常实际的问题。

22.9.1 通过改写代码来挂接API

API挂接并不是一个新技术，多年来编程人员一直在使用API挂接方法。如果要解决上面所说的问题，那么人们首先看到的“解决方案”是通过改写代码来进行挂接。下面是具体的操作方法：

1) 找到你想挂接的函数在内存中的地址（比如说Kernel32.dll中的ExitProcess）。

2) 将该函数的头几个字节保存在你自己的内存中。

3) 用一个JUMP CPU指令改写该函数的头几个字节，该指令会转移到你的替换函数的内存地址。当然，你的替换函数的标记必须与你挂接的函数的标记完全相同，即所有的参数必须一样，返回值必须一样，调用规则必须一样。

4) 现在，当一个线程调用已经挂接的函数时，JUMP指令实际上将转移到你的替换函数。这时，你就能够执行任何代码。

5) 取消函数的挂接状态，方法是取出（第二步）保存的字节，将它们放回挂接函数的开头。

6) 调用挂接的函数（它已不再被挂接），该函数将执行其通常的处理操作。

7) 当原始函数返回时，再次执行第二和第三步，这样你的替换函数就可以被调用。

这种方法在16位Windows编程员中使用得非常普遍，并且用得很好。今天，这种方法存在着若干非常严重的不足，因此建议尽量避免使用它。首先，它对CPU的依赖性很大，在x86、Alpha和其他的CPU上的JUMP指令是不同的，必须使用手工编码的机器指令才能使这种方法生效。第二，这种方法在抢占式多线程环境中根本不起作用。线程需要占用一定的时间来改写函数开头的代码。当代码被改写时，另一个线程可能试图调用该同一个函数。结果将是灾难性的。因此，只有当你知道在规定的时间只有一个线程试图调用某个函数时，才能使用这种方法。

> Windows 98 在Windows 98上，主要的Windows DLL（Kernel32、AdvAPI32、User32和GDI32）是这样受到保护的，即应用程序不能改写它们的代码页面。通过编写虚拟设备驱动程序（VxD）才能够获得这种保护。

22.9.2 通过操作模块的输入节来挂接API

另一种API挂接方法能够解决我前面讲到的两个问题。这种方法实现起来很容易，并且相当健壮。但是，要理解这种方法，必须懂得动态连接是如何工作的。尤其必须懂得模块的输入节中保护的是什么信息。第19章已经用了较多的篇幅介绍了输入节是如何生成的以及它包含的内容。当阅读下面的内容时，可以回头参考第19章的有关说明。

如你所知，模块的输入节包含一组该模块运行时需要的DLL。另外，它还包含该模块从每个DLL输入的符号的列表。当模块调用一个输入函数时，线程实际上要从模块的输入节中捕获需要的输入函数的地址，然后转移到该地址。

要挂接一个特定的函数，只需要改变模块的输入节中的地址，就这么简单。它不存在依赖

China-pub.com
下载

CPU的问题。同时，由于修改了函数的代码，因此不需要担心线程的同步问题。

下面这个函数就负责执行这个重要的操作。它接受一个模块的输入节，以便引用特定地址上的一个符号。如果存在这样的引用，那么它就改变该符号的地址。

```
void ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

   ULONG ulSize;
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
      ImageDirectoryEntryToData(hmodCaller, TRUE,
      IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

   if (pImportDesc == NULL)
      return;  // This module has no import section.

   // Find the import descriptor containing references
   // to callee's functions.
   for (; pImportDesc->Name; pImportDesc++) {
      PSTR pszModName = (PSTR)
         ((PBYTE) hmodCaller + pImportDesc->Name);
      if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
         break;
   }

   if (pImportDesc->Name == 0)
      // This module doesn't import any functions from this callee.
      return;

   // Get caller's import address table (IAT)
   // for the callee's functions.
   PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
      ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

   // Replace current function address with new function address.
   for (; pThunk->u1.Function; pThunk++) {

      // Get the address of the function address.
      PROC* ppfn = (PROC*) &pThunk->u1.Function;

      // Is this the function we're looking for?
      BOOL fFound = (*ppfn == pfnCurrent);

      // See the sample code for some tricky Windows 98
      // stuff that goes here.

      if (fFound) {
         // The addresses match; change the import section address.
         WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
         return;  // We did it; get out.
      }
   }

   // If we get to here, the function
   // is not in the caller's import section.
}
```

China-pub.com 下载

为了说明如何调用该函数，让我们首先介绍一种可能存在的环境。比如说，我们有一个模块称为DataBase.exe。该模块中的代码调用Kernel32.dll中包含的ExitProcess函数，但是我们想要调用我的DBExtend.dll模块中包含的MyExitProcess函数。为了完成这个操作，需要调用下面的ReplaceIATEntryInOneMod函数：

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"),
   "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("DataBase.exe");

void ReplaceIATEntryInOneMod(
   "Kernel32.dll", // Module containing the function (ANSI)
   pfnOrig,        // Address of function in callee
   MyExitProcess,  // Address of new function to be called
   hmodCaller);    // Handle of module that should call the new function
```

ReplaceIATEntryInOneMod函数要做的第一件事情是找出hmodCaller模块的输入节，方法是调用ImageDirectoryEntryToData函数，给它传递IMAGE_DIRECTORY_ENTRY_IMPORT。如果该函数返回NULL，DataBase.exe模块就没有输入节，并且不进行任何操作。

如果DataBase.exe有一个输入节，那么ImageDirectoryEntryToData就返回该输入节的地址，该地址是一个类型为PIMAGE_IMPORT_DESCRIPTOR的指针。现在我们必须查看该模块的输入节，找出包含我们想要修改的输入函数的DLL。在这个例子中，我们查找从“Kernel32.dll”输入的符号（这是传递给ReplaceIATEntryInOneMod函数的第一个参数）。for循环负责扫描DLL模块的名字。注意，模块的输入节中的所有字符串都是用ANSI（决不能用Unicode）编写。这就是为什么要显式调用lstrcmpiA而不是lstrcmpi宏的原因。

如果该循环终止运行，但是没有找到对“Kernel32.dll”中的任何符号的引用，那么该函数就返回，并且仍然无所事事。如果模块的输入节确实引用了“Kernel32.dll”中的符号，那么将得到包含输入符号信息的IMAGE_THUNK_DATA结构的数组的地址。然后，重复引用来自“Kernel32.dll”的所有输入符号，寻找与符号的当前地址相匹配的地址。在我们的例子中，我们寻找的是与ExitProcess函数的地址相匹配的地址。

如果没有与我们寻找的地址相匹配的地址，那么这个方法决不能输入需要的的符号，而ReplaceIATEntryInOneMod函数则返回。如果找到了一个匹配的地址，便调用WriteProcessMemory函数，以便改变替换函数的地址。使用WriteProcessMemory函数，而不是InterlockedExchangePointer函数是因为WriteProcessMemory能够改变字节，而不管这些字节拥有什么页面保护属性。例如，如果页面拥有PAGE_READONLY保护属性，那么InterlockedExchangePointer函数将会引发访问违规，而WriteProcessMemory函数则能够处理页面保护属性的所有变更，并且仍然能够正常运行。

从现在起，当任何线程执行DataBase.exe模块中调用ExitProcess的代码时，就能够很容易得到Kernel32.dll中的ExitProcess函数的实地址，并在我们想要进行通常的ExitProcess处理时调用它。

注意，ReplaceIATEntryInOneMod函数能够改变由单个模块中的代码进行的函数调用。但是，另一个DLL可能位于该地址空间中，而该DLL也可能调用ExitProcess。如果DataBase.exe之外的一个模块试图调用ExitProcess，那么在调用Kernel32.dll中的ExitProcess时，它的调用将会成功。

如果想要捕获从所有模块对ExitProcess进行的所有调用，必须为加载到进程的地址空间中

China-pub.com
下载

的每个模块进行一次对 ReplaceIATEntryInOneMod函数的调用。为此，我编写了另一个函数，称为ReplaceIATEntryInAllMods。该函数仅仅使用 ToolHelp函数来枚举加载到进程的地址空间中的所有模块，然后为每个模块调用 ReplaceIATEntryInOneMod，并为最后一个参数传递相应的模块句柄。

在少数几个地方可能发生一些问题。例如，如果在调用 ReplaceIATEntryInAllMods后，线程又调用LoadLibrary函数来加载新DLL，将会出现什么情况呢？在这种情况下，新加载的 DLL可能调用没有挂接的 ExitProcess函数。为了解决这个问题，必须挂接 LoadLibraryA、LoadLibraryW、LoadLibraryExA和LoadLibraryExW等函数，这样，就能够捕获这些函数的调用，并且为新加载的模块调用ReplaceIATEntryInOneMod。

最后一个问题与GetProcAddress有关。比如说有一个线程执行下面的代码：

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS) GetProcAddress(
   GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

这个代码让系统去获取Kernel32.dll中的ExitProcess函数的实地址，然后调用该地址。如果一个线程执行该代码，你的替换函数将不会被调用。为了解决这个问题，你也必须挂接GetProcAddress函数。如果它被调用，并且准备返回一个已经挂接的函数的地址，那么你必须返回替换函数的地址。

下一节中展示的示例应用程序显示了如何进行 API挂接，同时也解决了所有的LoadLibrary和GetProcAddress函数的问题。

22.9.3 LastMsgBoxInfo示例应用程序

清单22-4中列出的LastMsgBoxInfo应用程序（“22 LastMsgBoxInfo.exe ”）展示了API挂接的方法。它挂接了对User32.dll中包含的所有MessageBox函数的调用。若要挂接所有进程中的该函数，该应用程序使用 Windows挂接方法进行DLL的插入操作。该应用程序和 DLL的源代码和资源文件均位于本书所附光盘上的22- LastMsgBoxInfo和22- LastMsgBoxInfoLib目录下。

当运行该应用程序时，将出现图22-6所示的对话框。



图22-6 运行LastMsgBoxInfo时出现的对话框

这时，该应用程序进入等待状态。现在运行任何一个应用程序，使它显示一个消息框。为了测试的目的，我总是运行Notepad，输入一些文字，然后设法关闭Notepad，但是不保存输入的文字。这使得Notepad显示图22-7所示的消息框。

当关闭这个消息框时，LastMsgBoxInfo对话框将如图22-8所示。



图22-7 运行 Notepad 时显示的消息框



图22-8 关闭Notepad 时显示的LastMsgBoxInfo对话框

China-pub.com 下载

可以看到，LastMsgBoxInfo应用程序能够知道其他进程是如何调用MessageBox函数的。

显示和管理Last MessageBox Info对话框的代码是非常简单的。API挂接的设置正是全部工作的难点之所在。为了使API的挂接操作更加容易一些，我创建了一个CAPIHook C++类。这个类的定义是在APIHook.h文件中，类的实现是在APIHook.cpp文件中。这个类的使用是很方便的，因为它只有很少几个公有成员函数：一个构造函数，一个析构函数，还有一个返回原始函数的地址的函数。

若要挂接一个函数，只要像下面这样创建这个类的一个实例：

```
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
   (PROC) Hook_MessageBoxA, TRUE);

CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
   (PROC) Hook_MessageBoxW, TRUE);
```

注意，我必须挂接两个函数，即MessageBoxA和MessageBoxW。User32.dll包含了这两个函数。当MessageBoxA被调用时，我要使我的Hook_MessageBoxA被调用。当MessageBoxW被调用时，我要使我的Hook_MessageBoxW函数被调用。

我的CAPIHook类的构造函数只记住你决定要挂接的是什么API，并调用ReplaceIATEntryInAllMods，以便进行实际的挂接操作。

另一个公有成员函数是析构函数。当一个CAPIHook对象超出作用域时，析构函数就调用ReplaceIATEntryInAllMods，将符号的地址恢复成每个模块中它的原始地址，函数不再挂接。

第三个公有成员函数返回原始函数的地址。这个成员函数通常从替换函数内部进行调用，以便调用原始函数。下面是Hook_MessageBoxA函数中的代码：

```
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText,
   PCSTR pszCaption, UINT uType) {

   int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
      (hWnd, pszText, pszCaption, uType);
   SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);
   return(nResult);
}
```

这个代码引用全局g_MessageBoxA的CAPIHook对象。将这个对象转换成一个PROC数据类型将会导致成员函数返回User32.dll中的原始MessageBoxA函数的地址。

如果你使用这个C++类，那么这就是挂接和撤消挂接输入函数的全部方法。如果你观察CAPIHook.cpp文件结尾处的代码，将会发现C++类会自动建立CAPIHook对象的实例，以便捕获LoadLibraryA、LoadLibraryW、LoadLibraryExA和LoadLibraryExW。这样，CAPIHook类就能自动解决前面讲到的一些问题。

清单22-4 LastMsgBoxInfo示例应用程序



LastMsgBoxInfo.cpp

```
/******************************************************************************
Module:  LastMsgBoxInfo.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
```

China-pub.com
下载


```
#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-LastMsgBoxInfoLib\LastMsgBoxInfoLib.h"


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_LASTMSGBOXINFO);
   SetDlgItemText(hwnd, IDC_INFO,
      TEXT("Waiting for a Message Box to be dismissed"));
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

   SetWindowPos(GetDlgItem(hwnd, IDC_INFO), NULL,
      0, 0, cx, cy, SWP_NOZORDER);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) {

   // Some hooked process sent us some message box info, display it
   SetDlgItemTextA(hwnd, IDC_INFO, (PCSTR) pcds->lpData);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
```

China-pub.com 下载

```
   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_SIZE,       Dlg_OnSize);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_COPYDATA,   Dlg_OnCopyData);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DWORD dwThreadId = 0;
#ifdef _DEBUG
   HWND hwnd = FindWindow(NULL, TEXT("Untitled - Paint"));
   dwThreadId = GetWindowThreadProcessId(hwnd, NULL);
#endif
   LastMsgBoxInfo_HookAllApps(TRUE, dwThreadId);
   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LASTMSGBOXINFO), NULL, Dlg_Proc);
   LastMsgBoxInfo_HookAllApps(FALSE, 0);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

LastMsgBoxInfo.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
```

China-pub.com
下载


```
//
// Dialog
//

IDD_LASTMSGBOXINFO DIALOG DISCARDABLE  0, 0, 379, 55
STYLE DS_CENTER | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU | WS_THICKFRAME
 CAPTION "Last MessageBox Info"
 FONT 8, "MS Sans Serif"
 BEGIN
     EDITTEXT        IDC_INFO,0,0,376,52,ES_MULTILINE | ES_AUTOVSCROLL |
                     ES_AUTOHSCROLL | ES_READONLY | WS_VSCROLL | WS_HSCROLL
 END


 /////////////////////////////////////////////////////////////////////////////
 //
 // DESIGNINFO
 //

 #ifdef APSTUDIO_INVOKED
 GUIDELINES DESIGNINFO DISCARDABLE
 BEGIN
     IDD_LASTMSGBOXINFO, DIALOG
     BEGIN
         LEFTMARGIN, 7
         RIGHTMARGIN, 372
         TOPMARGIN, 7
         BOTTOMMARGIN, 48
     END
 END
 #endif    // APSTUDIO_INVOKED


 #ifdef APSTUDIO_INVOKED
 /////////////////////////////////////////////////////////////////////////////
 //
 // TEXTINCLUDE
 //

 1 TEXTINCLUDE DISCARDABLE
 BEGIN
     "resource.h\0"
 END

 2 TEXTINCLUDE DISCARDABLE
 BEGIN
     "#include ""afxres.h""\r\n"
     "\0"
 END
 3 TEXTINCLUDE DISCARDABLE
 BEGIN
     "\r\n"
     "\0"
 END
```

China-pub.com 下载

```
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_LASTMSGBOXINFO      ICON    DISCARDABLE     "LastMsgBoxInfo.ico"
#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

LastMsgBoxInfoLib.cpp

```
/******************************************************************************
Module:  LastMsgBoxInfoLib.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#define WINVER       0x0500
#include "..\CmnHdr.h"
#include <WindowsX.h>
#include <tchar.h>
#include <stdio.h>
#include "APIHook.h"

#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllexport)
#include "LastMsgBoxInfoLib.h"

//////////////////////////////////////////////////////////////////////////////

// Prototypes for the hooked functions
typedef int (WINAPI *PFNMESSAGEBOXA)(HWND hWnd, PCSTR pszText,
   PCSTR pszCaption, UINT uType);

typedef int (WINAPI *PFNMESSAGEBOXW)(HWND hWnd, PCWSTR pszText,
   PCWSTR pszCaption, UINT uType);

// We need to reference these variables before we create them.
```

China-pub.com
下载


```
extern CAPIHook g_MessageBoxA;
extern CAPIHook g_MessageBoxW;

////////////////////////////////////////////////////////////////////////////

// This function sends the MessageBox info to our main dialog box
void SendLastMsgBoxInfo(BOOL fUnicode,
   PVOID pvCaption, PVOID pvText, int nResult) {

   // Get the pathname of the process displaying the message box
   char szProcessPathname[MAX_PATH];
   GetModuleFileNameA(NULL, szProcessPathname, MAX_PATH);

   // Convert the return value into a human-readable string
   PCSTR pszResult = "(Unknown)";
   switch (nResult) {
      case IDOK:       pszResult = "Ok";        break;
      case IDCANCEL:   pszResult = "Cancel";    break;
      case IDABORT:    pszResult = "Abort";     break;
      case IDRETRY:    pszResult = "Retry";     break;
      case IDIGNORE:   pszResult = "Ignore";    break;
      case IDYES:      pszResult = "Yes";       break;
      case IDNO:       pszResult = "No";        break;
      case IDCLOSE:    pszResult = "Close";     break;
      case IDHELP:     pszResult = "Help";      break;
      case IDTRYAGAIN: pszResult = "Try Again"; break;
      case IDCONTINUE: pszResult = "Continue";  break;
   }

   // Construct the string to send to the main dialog box
   char sz[2048];
   wsprintfA(sz, fUnicode
      ? "Process: (%d) %s\r\nCaption: %S\r\nMessage: %S\r\nResult: %s"
      : "Process: (%d) %s\r\nCaption: %s\r\nMessage: %s\r\nResult: %s",
      GetCurrentProcessId(), szProcessPathname,
      pvCaption, pvText, pszResult);

   // Send the string to the main dialog box
   COPYDATASTRUCT cds = { 0, lstrlenA(sz) + 1, sz };
   FORWARD_WM_COPYDATA(FindWindow(NULL, TEXT("Last MessageBox Info")),
      NULL, &cds, SendMessage);
}

////////////////////////////////////////////////////////////////////////////

// This is the MessageBoxW replacement function
int WINAPI Hook_MessageBoxW(HWND hWnd, PCWSTR pszText, LPCWSTR pszCaption,
   UINT uType) {

   // Call the original MessageBoxW function
   int nResult = ((PFNMESSAGEBOXW)(PROC) g_MessageBoxW)
      (hWnd, pszText, pszCaption, uType);

   // Send the information to the main dialog box
   SendLastMsgBoxInfo(TRUE, (PVOID) pszCaption, (PVOID) pszText, nResult);
```

China-pub.com 下载

```
   // Return the result back to the caller
   return(nResult);
}

//////////////////////////////////////////////////////////////////////////////

// This is the MessageBoxA replacement function
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText, PCSTR pszCaption,
   UINT uType) {

   // Call the original MessageBoxA function
   int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
      (hWnd, pszText, pszCaption, uType);
   // Send the information to the main dialog box
   SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);

   // Return the result back to the caller
   return(nResult);
}

//////////////////////////////////////////////////////////////////////////////

// Hook the MessageBoxA and MessageBoxW functions
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
   (PROC) Hook_MessageBoxA, TRUE);

CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
   (PROC) Hook_MessageBoxW, TRUE);


// Since we do DLL injection with Windows' hooks, we need to save the hook
// handle in a shared memory block (Windows 2000 actually doesn't need this)
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
#pragma data_seg()
#pragma comment(linker, "/Section:Shared, rws")

//////////////////////////////////////////////////////////////////////////////

static LRESULT WINAPI GetMsgProc(int code, WPARAM wParam, LPARAM lParam) {

   // NOTE: On Windows 2000, the 1st parameter to CallNextHookEx can
   // be NULL.  On Windows 98, it must be the hook handle.
   return(CallNextHookEx(g_hhook, code, wParam, lParam));
}

//////////////////////////////////////////////////////////////////////////////

// Returns the HMODULE that contains the specified memory address
static HMODULE ModuleFromAddress(PVOID pv) {

   MEMORY_BASIC_INFORMATION mbi;
   return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
      ? (HMODULE) mbi.AllocationBase : NULL);
}
```

China-pub.com
下载


```
///////////////////////////////////////////////////////////////////////////////
BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL fInstall, DWORD dwThreadId) {

   BOOL fOk;

   if (fInstall) {

      chASSERT(g_hhook == NULL); // Illegal to install twice in a row

      // Install the Windows' hook
      g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
         ModuleFromAddress(LastMsgBoxInfo_HookAllApps), dwThreadId);

      fOk = (g_hhook != NULL);
   } else {

      chASSERT(g_hhook != NULL); // Can't uninstall if not installed
      fOk = UnhookWindowsHookEx(g_hhook);
      g_hhook = NULL;
   }

   return(fOk);
}

//////////////////////////////// End of File //////////////////////////////////
```

LastMsgBoxInfoLib.h

```
/******************************************************************************
Module:  LastMsgBoxInfoLib.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#ifndef LASTMSGBOXINFOLIBAPI
#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllimport)
#endif


///////////////////////////////////////////////////////////////////////////////


LASTMSGBOXINFOLIBAPI BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL fInstall,
   DWORD dwThreadId);

//////////////////////////////// End of File //////////////////////////////////
```

APIHook.cpp

```
/******************************************************************************
Module:  APIHook.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
```

China-pub.com 下载

```
******************************************************************************/


#include "..\CmnHdr.h"
#include <ImageHlp.h>
#pragma comment(lib, "ImageHlp")

#include "APIHook.h"
#include "..\04-ProcessInfo\Toolhelp.h"

//////////////////////////////////////////////////////////////////////////////

// When an application runs on Windows 98 under a debugger, the debugger
// makes the module's import section point to a stub that calls the desired
// function. To account for this, the code in this module must do some crazy
// stuff. These variables are needed to help with the crazy stuff.


// The highest private memory address (used for Windows 98 only)
PVOID CAPIHook::sm_pvMaxAppAddr = NULL;
const BYTE cPushOpCode = 0x68;   // The PUSH opcode on x86 platforms

//////////////////////////////////////////////////////////////////////////////

// The head of the linked-list of CAPIHook objects
CAPIHook* CAPIHook::sm_pHead = NULL;

//////////////////////////////////////////////////////////////////////////////

CAPIHook::CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,
   BOOL fExcludeAPIHookMod) {

   if (sm_pvMaxAppAddr == NULL) {
      // Functions with address above lpMaximumApplicationAddress require
      // special processing (Windows 98 only)
      SYSTEM_INFO si;
      GetSystemInfo(&si);
      sm_pvMaxAppAddr = si.lpMaximumApplicationAddress;
   }

   m_pNext  = sm_pHead;    // The next node was at the head
   sm_pHead = this;        // This node is now at the head

   // Save information about this hooked function
   m_pszCalleeModName   = pszCalleeModName;
   m_pszFuncName        = pszFuncName;
   m_pfnHook            = pfnHook;
   m_fExcludeAPIHookMod = fExcludeAPIHookMod;
   m_pfnOrig            = GetProcAddressRaw(
      GetModuleHandleA(pszCalleeModName), m_pszFuncName);
   chASSERT(m_pfnOrig != NULL);  // Function doesn't exist

   if (m_pfnOrig > sm_pvMaxAppAddr) {
      // The address is in a shared DLL; the address needs fixing up
      PBYTE pb = (PBYTE) m_pfnOrig;
```

China-pub.com
下载


```
      if (pb[0] == cPushOpCode) {
         // Skip over the PUSH op code and grab the real address
         PVOID pv = * (PVOID*) &pb[1];
         m_pfnOrig = (PROC) pv;
      }
   }

   // Hook this function in all currently loaded modules
   ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnOrig, m_pfnHook,
      m_fExcludeAPIHookMod);
}

////////////////////////////////////////////////////////////////////////////

CAPIHook::~CAPIHook() {

   // Unhook this function from all modules
   ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnHook, m_pfnOrig,
      m_fExcludeAPIHookMod);

   // Remove this object from the linked list
   CAPIHook* p = sm_pHead;
    if (p == this) {     // Removing the head node
       sm_pHead = p->m_pNext;
    } else {

       BOOL fFound = FALSE;

       // Walk list from head and fix pointers
       for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
          if (p->m_pNext == this) {
             // Make the node that points to us point to the our next node
             p->m_pNext = p->m_pNext->m_pNext;
             break;
          }
       }
       chASSERT(fFound);
    }
}

////////////////////////////////////////////////////////////////////////////

// NOTE: This function must NOT be inlined
FARPROC CAPIHook::GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName) {

   return(::GetProcAddress(hmod, pszProcName));
}

////////////////////////////////////////////////////////////////////////////

// Returns the HMODULE that contains the specified memory address
static HMODULE ModuleFromAddress(PVOID pv) {

   MEMORY_BASIC_INFORMATION mbi;
   return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
```

China-pub.com 下载

```
      ? (HMODULE) mbi.AllocationBase : NULL);
}

//////////////////////////////////////////////////////////////////////////////

void CAPIHook::ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, BOOL fExcludeAPIHookMod) {

   HMODULE hmodThisMod = fExcludeAPIHookMod
      ? ModuleFromAddress(ReplaceIATEntryInAllMods) : NULL;
   // Get the list of modules in this process
   CToolhelp th(TH32CS_SNAPMODULE, GetCurrentProcessId());

   MODULEENTRY32 me = { sizeof(me) };
   for (BOOL fOk = th.ModuleFirst(&me); fOk; fOk = th.ModuleNext(&me)) {

      // NOTE: We don't hook functions in our own module
      if (me.hModule != hmodThisMod) {

         // Hook this function in this module
         ReplaceIATEntryInOneMod(
            pszCalleeModName, pfnCurrent, pfnNew, me.hModule);
      }
   }
}

//////////////////////////////////////////////////////////////////////////////

void CAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

   // Get the address of the module's import section
   ULONG ulSize;
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
      ImageDirectoryEntryToData(hmodCaller, TRUE,
      IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

   if (pImportDesc == NULL)
      return;  // This module has no import section


   // Find the import descriptor containing references to callee's functions
   for (; pImportDesc->Name; pImportDesc++) {
      PSTR pszModName = (PSTR) ((PBYTE) hmodCaller + pImportDesc->Name);
      if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
         break;   // Found
   }

   if (pImportDesc->Name == 0)
      return;  // This module doesn't import any functions from this callee

   // Get caller's import address table (IAT) for the callee's functions
   PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
      ((PBYTE) hmodCaller + pImportDesc->FirstThunk);
  // Replace current function address with new function address
```

China-pub.com
下载


```
   for (; pThunk->u1.Function; pThunk++) {

      // Get the address of the function address
      PROC* ppfn = (PROC*) &pThunk->u1.Function;

      // Is this the function we're looking for?
      BOOL fFound = (*ppfn == pfnCurrent);

      if (!fFound && (*ppfn > sm_pvMaxAppAddr)) {

         // If this is not the function and the address is in a shared DLL,
         // then maybe we're running under a debugger on Windows 98. In this
         // case, this address points to an instruction that may have the
         // correct address.

         PBYTE pbInFunc = (PBYTE) *ppfn;
         if (pbInFunc[0] == cPushOpCode) {
            // We see the PUSH instruction, the real function address follows
            ppfn = (PROC*) &pbInFunc[1];

            // Is this the function we're looking for?
            fFound = (*ppfn == pfnCurrent);
         }
      }

      if (fFound) {
         // The addresses match, change the import section address
         WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
         return;  // We did it, get out
      }
   }

   // If we get to here, the function is not in the caller's import section
}

////////////////////////////////////////////////////////////////////////////

// Hook LoadLibrary functions and GetProcAddress so that hooked functions
// are handled correctly if these functions are called.

CAPIHook CAPIHook::sm_LoadLibraryA  ("Kernel32.dll", "LoadLibraryA",
   (PROC) CAPIHook::LoadLibraryA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryW  ("Kernel32.dll", "LoadLibraryW",
   (PROC) CAPIHook::LoadLibraryW, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExA("Kernel32.dll", "LoadLibraryExA",
   (PROC) CAPIHook::LoadLibraryExA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExW("Kernel32.dll", "LoadLibraryExW",
   (PROC) CAPIHook::LoadLibraryExW, TRUE);

CAPIHook CAPIHook::sm_GetProcAddress("Kernel32.dll", "GetProcAddress",
   (PROC) CAPIHook::GetProcAddress, TRUE);
```

China-pub.com 下载

```
//////////////////////////////////////////////////////////////////////////////

void CAPIHook::FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags) {

   // If a new module is loaded, hook the hooked functions
   if ((hmod != NULL) && ((dwFlags & LOAD_LIBRARY_AS_DATAFILE) == 0)) {

      for (CAPIHook* p = sm_pHead; p != NULL; p = p->m_pNext) {
         ReplaceIATEntryInOneMod(p->m_pszCalleeModName,
            p->m_pfnOrig, p->m_pfnHook, hmod);
      }
   }
}

//////////////////////////////////////////////////////////////////////////////

HMODULE WINAPI CAPIHook::LoadLibraryA(PCSTR pszModulePath) {

   HMODULE hmod = ::LoadLibraryA(pszModulePath);
   FixupNewlyLoadedModule(hmod, 0);
   return(hmod);
}

//////////////////////////////////////////////////////////////////////////////

HMODULE WINAPI CAPIHook::LoadLibraryW(PCWSTR pszModulePath) {

   HMODULE hmod = ::LoadLibraryW(pszModulePath);
   FixupNewlyLoadedModule(hmod, 0);
   return(hmod);
}
//////////////////////////////////////////////////////////////////////////////

HMODULE WINAPI CAPIHook::LoadLibraryExA(PCSTR pszModulePath,
   HANDLE hFile, DWORD dwFlags) {

   HMODULE hmod = ::LoadLibraryExA(pszModulePath, hFile, dwFlags);
   FixupNewlyLoadedModule(hmod, dwFlags);
   return(hmod);
}

//////////////////////////////////////////////////////////////////////////////

HMODULE WINAPI CAPIHook::LoadLibraryExW(PCWSTR pszModulePath,
   HANDLE hFile, DWORD dwFlags) {

   HMODULE hmod = ::LoadLibraryExW(pszModulePath, hFile, dwFlags);
   FixupNewlyLoadedModule(hmod, dwFlags);
   return(hmod);
}

//////////////////////////////////////////////////////////////////////////////

FARPROC WINAPI CAPIHook::GetProcAddress(HMODULE hmod, PCSTR pszProcName) {

   // Get the true address of the function
   FARPROC pfn = GetProcAddressRaw(hmod, pszProcName);
```

China-pub.com
下载


```
   // Is it one of the functions that we want hooked?
   CAPIHook* p = sm_pHead;
   for (; (pfn != NULL) && (p != NULL); p = p->m_pNext) {

      if (pfn == p->m_pfnOrig) {

         // The address to return matches an address we want to hook
         // Return the hook function address instead
         pfn = p->m_pfnHook;
         break;
      }
   }

   return(pfn);
}

//////////////////////////////// End of File //////////////////////////////////
```

APIHook.h

```
/*****************************************************************************
Module:  APIHook.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****************************************************************************/


#pragma once

//////////////////////////////////////////////////////////////////////////////

class CAPIHook {
public:
   // Hook a function in all modules
   CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,
      BOOL fExcludeAPIHookMod);

   // Unhook a function from all modules
   ~CAPIHook();

   // Returns the original address of the hooked function
   operator PROC() { return(m_pfnOrig); }

public:
   // Calls the real GetProcAddress
   static FARPROC WINAPI GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName);

private:
   static PVOID sm_pvMaxAppAddr; // Maximum private memory address
   static CAPIHook* sm_pHead;    // Address of first object
   CAPIHook* m_pNext;            // Address of next object

   PCSTR m_pszCalleeModName;     // Module containing the function (ANSI)
   PCSTR m_pszFuncName;          // Function name in callee (ANSI)
   PROC  m_pfnOrig;              // Original function address in callee
```

China-pub.com 下载

```
   PROC  m_pfnHook;              // Hook function address
   BOOL  m_fExcludeAPIHookMod;   // Hook module w/CAPIHook implementation?

private:
   // Replaces a symbol's address in a module's import section
   static void WINAPI ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
      PROC pfnOrig, PROC pfnHook, BOOL fExcludeAPIHookMod);
      // Replaces a symbol's address in all module's import sections
      static void WINAPI ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
         PROC pfnOrig, PROC pfnHook, HMODULE hmodCaller);

   private:
      // Used when a DLL is newly loaded after hooking a function
      static void    WINAPI FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags);

      // Used to trap when DLLs are newly loaded
      static HMODULE WINAPI LoadLibraryA(PCSTR  pszModulePath);
      static HMODULE WINAPI LoadLibraryW(PCWSTR pszModulePath);
      static HMODULE WINAPI LoadLibraryExA(PCSTR  pszModulePath,
         HANDLE hFile, DWORD dwFlags);
      static HMODULE WINAPI LoadLibraryExW(PCWSTR pszModulePath,
         HANDLE hFile, DWORD dwFlags);

      // Returns address of replacement function if hooked function is requested
      static FARPROC WINAPI GetProcAddress(HMODULE hmod, PCSTR pszProcName);

   private:
      // Instantiates hooks on these functions
      static CAPIHook sm_LoadLibraryA;
      static CAPIHook sm_LoadLibraryW;
      static CAPIHook sm_LoadLibraryExA;
      static CAPIHook sm_LoadLibraryExW;
      static CAPIHook sm_GetProcAddress;
   };

   ////////////////////////////////// End of File //////////////////////////////////
```

China-pub.com
下载

第五部分　结构化异常处理

第23章　结束处理程序

你可以闭上眼睛，想象有这样一个编程环境，在这个环境里，你编写的代码永远不会出错。总有足够的内存，没有人会传递给你一个无效的指针，你需要的文件总是存在。如果按照这种假想，编写程序不是很愉快的事情吗？那样的话，程序会非常容易编写、阅读和理解。我们不会再让每个程序中通篇的if语句和goto语句搞得昏头昏脑，只需从头到尾书写代码就是了。

如果说这种直接的编程环境只是一个美妙的梦想的话，那么结构化异常处理（SEH）就会给你一个现实的惊喜。使用SEH的好处就是当你编写程序时，只需要关注程序要完成的任务。如果在运行时发生什么错误，系统会发现并将发生的问题通知你。

利用SEH，你可以完全不用考虑代码里是不是有错误，这样就把主要的工作同错误处理分离开来。这样的分离，可以使你集中精力处理眼前的工作，而将可能发生的错误放在后面处理。

微软在Windows中引入SEH的主要动机是为了便于操作系统本身的开发。操作系统的开发人员使用SEH，使得系统更加强壮。我们也可以使用SEH，使我们的自己的程序更加强壮。

使用SEH所造成的负担主要由编译程序来承担，而不是由操作系统承担。当异常块（exception block）出现时，编译程序要生成特殊的代码。编译程序必须产生一些表（table）来支持处理SEH的数据结构。编译程序还必须提供回调（callback）函数，操作系统可以调用这些函数，保证异常块被处理。编译程序还要负责准备栈结构和其他内部信息，供操作系统使用和参考。在编译程序中增加SEH支持不是一件容易的事。不同的编译程序厂商会以不同的方式实现SEH，这一点并不让人感到奇怪。幸亏我们可以不必考虑编译程序的实现细节，而只使用编译程序的SEH功能。

由于各编译程序的实现上存在着差别，这样以特定的方式用特定的代码例子讨论SEH的优点就很困难。但大多数编译程序厂商都采用微软建议的文法。本书中的例子使用的文法和关键字可能与其他一些公司编译程序所使用的不同，但主要的SEH概念是一样的。本章使用Microsoft Visual C++编译程序的文法。

> **注意**　不要将结构化异常处理同C++的异常处理相混淆。C++异常处理是一种不同形式的异常处理，其形式是使用C++关键字catch和throw。微软的Visual C++也支持C++的异常处理，并且在内部实现上利用了已经引入到编译程序和Windows操作系统的结构化异常处理的功能。

SEH实际包含两个主要功能：结束处理（termination handling）和异常处理（exception handling）。本章讨论结束处理，下一章讨论异常处理。

一个结束处理程序能够确保去调用和执行一个代码块（结束处理程序，termination handler），而不管另外一段代码（保护体，guarded body）是如何退出的。结束处理程序的文法结构（使用微软的Visual C++编译程序）如下：

China-pub.com 下载

```
__try {
   // Guarded body
   :
}
__finally {
   // Termination handler
   :
}
```

--try和--finally关键字用来标出结束处理程序两段代码的轮廓。在上面的代码段中，操作系统和编译程序共同来确保结束处理程序中的 --finally代码块能够被执行，不管保护体（try块）是如何退出的。不论你在保护体中使用 return，还是goto，或者是longjump，结束处理程序（finally块）都将被调用。下面将通过几个例子来说明这一点。

23.1 通过例子理解结束处理程序

由于在使用SEH时，编译程序和操作系统直接参与了程序代码的执行，为了解释 SEH如何工作，最好的办法就是考察源代码例子，讨论例子中语句执行的次序。

因此，在下面几节给出不同的源代码片段，对每一个片段解释编译程序和操作系统如何改变代码的执行次序。

23.2 Funcenstein1

为了甄别使用结束处理程序的各种情况，我们来考察更具体的代码例子。

```
DWORD Funcenstein1() {
   DWORD dwTemp;

   // 1. Do any processing here.
   :
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // 4. Continue processing.
   return(dwTemp);
}
```

上面程序中加了标号的注释指出了代码执行的次序。在 Funcenstein1中，使用try-finally块并没有带来很多好处。代码要等待信标（semaphore），改变保护数据的内容，保存局部变量dwTemp的新值，释放信标，将新值返回给调用程序。

23.3 Funcenstein2

现在我们把这个程序稍微改动一下，看会发生什么事情。

```
DWORD Funcenstein2() {
```

China-pub.com
下载


```
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;

      // Return the new value.
      return(dwTemp);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // Continue processing--this code
   // will never execute in this version.
   dwTemp = 9;
   return(dwTemp);
}
```

在Funcenstein2中，try块的末尾增加了一个return语句。这个return语句告诉编译程序在这里要退出这个函数并返回dwTemp变量的内容，现在这个变量的值是5。但是，如果这个return语句被执行，该线程将不会释放信标，其他线程也就不能再获得对信标的控制。可以想象，这样的执行次序会产生很大的问题，那些等待信标的线程可能永远不会恢复执行。

通过使用结束处理程序，可以避免return语句的过早执行。当return语句试图退出try块时，编译程序要确保finally块中的代码首先被执行。要保证finally块中的代码在try块中的return语句退出之前执行。Funcenstein2中，将对ReleaseSemaphore的调用放在结束处理程序块中，保证信标总会被释放。这样就不会造成一个线程一直占有信标，否则将意味着所有其他等待信标的线程永远不会被分配CPU时间。

在finally块中的代码执行之后，函数实际上就返回。任何出现在finally块之下的代码将不再执行，因为函数已在try块中返回。所以这个函数的返回值是5，而不是9。

读者可能要问编译程序是如何保证在try块可以退出之前执行finally块的。当编译程序检查源代码时，它看到在try块中有return语句。这样，编译程序就生成代码将返回值（本例中是）保存在一个编译程序建立的临时变量中。编译程序然后再生成代码来执行finally块中包含的指令，这称为局部展开。更特殊的情况是，由于try块中存在过早退出的代码，从而产生局部展开，导致系统执行finally块中的内容。在finally块中的指令执行之后，编译程序临时变量的值被取出并从函数中返回。

可以看到，要完成这些事情，编译程序必须生成附加的代码，系统要执行额外的工作。在不同的CPU上，结束处理所需要的步骤也不同。例如，在Alpha处理器上，必须执行几百个甚至几千个CPU指令来捕捉try块中的过早返回并调用finally块。在编写代码时，就应该避免引起结束处理程序的try块中的过早退出，因为程序的性能会受到影响。本章后面，将讨论__leave关键字，它有助于避免编写引起局部展开的代码。

设计异常处理的目的是用来捕捉异常的——不常发生的语法规则的异常情况（在我们的例子中，就是过早返回）。如果情况是正常的，明确地检查这些情况，比起依赖操作系统和编译

China-pub.com 下载

程序的 SEH功能来捕捉常见的事情要更有效。

注意当控制流自然地离开try块并进入finally块（就像在Funcenstein1中）时，进入finally块的系统开销是最小的。在x86 CPU上使用微软的编译程序，当执行离开 try 块进入finally块时，只有一个机器指令被执行，读者可以在自己的程序中注意到这种系统开销。当编译程序要生成额外的代码，系统要执行额外的工作时（如同在Funcenstein2中），系统开销就很值得注意了。

23.4 Funcenstein3

现在我们对函数再做修改，看会出现什么情况：

```
DWORD Funcenstein3() {
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;

      // Try to jump over the finally block.
      goto ReturnValue;
   }

   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   dwTemp = 9;
   // 4. Continue processing.
   ReturnValue:
   return(dwTemp);
}
```

在Funcenstein3中，当编译程序看到try块中的goto语句，它首先生成一个局部展开来执行finally块中的内容。这一次，在finally块中的代码执行之后，在ReturnValue标号之后的代码将执行，因为在try块和finally块中都没有返回发生。这里的代码使函数返回 5。而且，由于中断了从try块到finally块的自然流程，可能要蒙受很大的性能损失（取决于运行程序的CPU）。

23.5 Funcfurter1

现在我们来考察另外的情况，这里可以真正显示结束处理的价值。看下面的函数：

```
DWORD Funcfurter1() {
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   .
```

China-pub.com
下载


```
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      dwTemp = Funcinator(g_dwProtectedData);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // 4. Continue processing.
   return(dwTemp);
}
```

现在假想一下，try块中的Funcinator函数调用包含一个错误，会引起一个无效内存访问。如果没有SEH，在这种情况下，将会给用户显示一个很常见的Application Error对话框。当用户忽略这个错误对话框，该进程就结束了。当这个进程结束（由于一个无效内存访问），信标仍将被占用并且永远不会被释放，这时候，任何等待信标的其他进程中的线程将不会被分配CPU时间。但若将对ReleaseSemaphore的调用放在finally块中，就可以保证信标获得释放，即使某些其他函数会引起内存访问错误。

如果结束处理程序足够强，能够捕捉由于无效内存访问而结束的进程，我们就可以相信它也能够捕捉setjump和longjump的结合，还有那些简单语句如break和continue。

23.6 突击测验：FuncaDoodleDoo

现在做一个测试，读者判断一下下面的函数返回什么值？

```
DWORD FuncaDoodleDoo() {
   DWORD dwTemp = 0;

   while (dwTemp < 10) {

      __try {
         if (dwTemp == 2)
            continue;

         if (dwTemp == 3)
            break;
      }
      __finally {
         dwTemp++;
      }

      dwTemp++;
   }

   dwTemp += 10;
   return(dwTemp);
}
```

我们一步一步地分析函数做了什么。首先dwTemp被设置成0。try块中的代码执行，但两个if语句的值都不为TRUE。执行自然移到finally块中的代码，在这里dwTemp增加到1。然后finally块之后的指令又增加dwTemp，使它的值成为2。

China-pub.com 下载

当循环继续，dwTemp为2，try块中的continue语句将被执行。如果没有结束处理程序在从try块中退出之前强制执行finally块，执行就立即跳回while测试，dwTemp不会被改变，将出现无限（死）循环。利用一个结束处理程序，系统知道continue语句要引起控制流过早退出try块，而将执行移到finally块。在finally块中，dwTemp被增加到3。但finally块之后的代码不执行，因为控制流又返回到continue，再到循环的开头。

现在我们处理循环的第三次重复。这一次，第一个 if语句的值是FALSE，但第二个语句的值是TRUE。系统又能够捕捉要跳出try块的企图，并先执行finally块中的代码。现在dwTemp增加到4。由于break语句被执行，循环之后程序部分的控制恢复。这样， finally块之后的循环中的代码没有执行。循环下面的代码对dwTemp增加10，这时dwTemp的值是14，这就是调用这个函数的结果。当然，实际上我们不会写出FuncaDoodleDoo这样的代码。这里将continue和break语句放在代码的中间，是为了说明结束处理程序的操作。

尽管结束处理程序可以捕捉try块过早退出的大多数情况，但当线程或进程被结束时，它不能引起finally块中的代码执行。当调用ExitThread或ExitProcess时，将立即结束线程或进程，而不会执行 finally块中的任何代码。另外，如果由于某个程序调用 TerminateThread或TerminateProcess，线程或进程将死掉，finally块中的代码也不执行。某些C运行期函数（例如abort）要调用ExitProcess，也使finally块中的代码不能执行。虽然没有办法阻止其他程序结束你的一个线程或进程，但你可以阻止你自己过早调用 ExitThread和ExitProcess。

23.7 Funcenstein4

我们再看一种异常处理的情况。

```
DWORD Funcenstein4() {
   DWORD dwTemp;
   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;

      // Return the new value.
      return(dwTemp);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
      return(103);
   }

   // Continue processing--this code will never execute.
   dwTemp = 9;
   return(dwTemp);
}
```

在Funcenstein4中，try块将要执行，并试图将dwTemp的值（5）返回给Funcenstein4的调用者。如同对Funcenstein2的讨论中提到的，试图从try块中过早的返回将导致产生代码，把返回

China-pub.com
下载

值置于由编译程序建立的临时变量中。然后，finally块中的代码被执行。在这里，与Funcenstein2不同的是在finally块中增加了一个return语句。Funcenstein4会向调用程序返回5还是103？这里的答案是103，因finally块中的return语句引起值103存储在值5所存储的临时变量中，覆盖了值5。当finally块完成执行，现在临时变量中的值（103）从Funcenstein4返回给调用程序。

我们已经看到结束处理程序在补救try块中的过早退出的执行方面很有效，但也看到结束处理程序由于要阻止try块的过早退出而产生了我们不希望有的结果。更好的办法是在结束处理程序的try块中避免任何会引起过早退出的语句。实际上，最好将return、continue、break、goto等语句从结束处理程序的try块和finally块中移出去，放在结束处理程序的外面。这样做会使编译程序产生较小的代码，因为不需要再捕捉try块中的过早退出，也使编译程序产生更快的代码（因为执行局部展开的指令也少）。另外，代码也更容易阅读和维护。

23.8 Funcarama1

我们已经谈过了结束处理程序的基本语法和语意。现在看一看如何用结束处理程序来简化一个更复杂的编程问题。先看一个完全没有利用结束处理程序的函数：

```
BOOL Funcarama1() {
   HANDLE hFile = INVALID_HANDLE_VALUE;
   PVOID pvBuf = NULL;
   DWORD dwNumBytesRead;
   BOOL fOk;

   hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
      FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
   if (hFile == INVALID_HANDLE_VALUE) {
      return(FALSE);
   }
   pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
   if (pvBuf == NULL) {
      CloseHandle(hFile);
      return(FALSE);
   }

   fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
   if (!fOk || (dwNumBytesRead == 0)) {
      VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
      CloseHandle(hFile);
      return(FALSE);
   }

   // Do some calculation on the data.
   .
   .
   .
   // Clean up all the resources.
   VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
   CloseHandle(hFile);
   return(TRUE);
}
```

Funcarama1中的各种错误检查使这个函数非常难以阅读，也使这个函数难以理解、维护和修改。

China-pub.com 下载

23.9 Funcarama2

当然，可以重新编写 Funcarama1，**使它更清晰一些，也更容易理解。**

```
BOOL Funcarama2() {
   HANDLE hFile = INVALID_HANDLE_VALUE;
   PVOID pvBuf = NULL;
   DWORD dwNumBytesRead;
   BOOL fOk, fSuccess = FALSE;

   hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
      FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

   if (hFile != INVALID_HANDLE_VALUE) {

      pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);

      if (pvBuf != NULL) {

         fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
         if (fOk && (dwNumBytesRead != 0)) {
            // Do some calculation on the data.
            :
            :
            fSuccess = TRUE;
         }

      }

      VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
   }

   CloseHandle(hFile);
   return(fSuccess);
}
```

Funcarama2**尽管比**Funcarama1**容易理解一些，但还是不好修改和维护。而且，当增加更多的条件语句时，这里的缩排格式就会走向极端，很快就到屏幕的最右边。**

23.10 Funcarama3

我们使用一个SEH**结束处理程序来重新编写**Funcarama1。

```
DWORD Funcarama3() {

   // IMPORTANT: Initialize all variables to assume failure.
   HANDLE hFile = INVALID_HANDLE_VALUE;
   PVOID pvBuf = NULL;

   __try {
      DWORD dwNumBytesRead;
      BOOL fOk;

      hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
         FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
      if (hFile == INVALID_HANDLE_VALUE) {
         return(FALSE);
      }
```

China-pub.com
下载


```
      pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
      if (pvBuf == NULL) {
         return(FALSE);
      }

      fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
      if (!fOk || (dwNumBytesRead != 1024)) {
         return(FALSE);
      }

      // Do some calculation on the data.
      .
      .
      .
   }

   __finally {
      // Clean up all the resources.
      if (pvBuf != NULL)
         VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
      if (hFile != INVALID_HANDLE_VALUE)
         CloseHandle(hFile);
   }
   // Continue processing.
   return(TRUE);
}
```

Funcarama3版的真正好处是函数的所有清理（cleanup）代码都局部化在一个地方且只在一个地方：finally块。如果需要在这个函数中再增加条件代码，只需在 finally块中简单地增加一个清理行，不需要回到每个可能失败的地方添加清理代码。

23.11 Funcarama4：最终的边界

Funcarama3版本的问题是系统开销。就像在 Funcenstein4中讨论的，应该尽可能避免在 try块中使用return语句。

为了帮助避免在try块中使用return语句，微软在其C/C++编译程序中增加了另一个关键字--leave。这里是Funcarma4版，它使用了--leave关键字：

```
DWORD Funcarama4() {

   // IMPORTANT: Initialize all variables to assume failure.
   HANDLE hFile = INVALID_HANDLE_VALUE;
   PVOID pvBuf = NULL;

   // Assume that the function will not execute successfully.
   BOOL fFunctionOk = FALSE;

   __try {
      DWORD dwNumBytesRead;
      BOOL fOk;
      hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
         FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
      if (hFile == INVALID_HANDLE_VALUE) {
         __leave;
      }

      pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
```

China-pub.com 下载

```
      if (pvBuf == NULL) {
         __leave;
      }

      fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
      if (!fOk || (dwNumBytesRead == 0)) {
         __leave;
      }

      // Do some calculation on the data.
      .
      .
      .
      // Indicate that the entire function executed successfully.
      fFunctionOk = TRUE;
   }
   __finally {
      // Clean up all the resources.
      if (pvBuf != NULL)
         VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
      if (hFile != INVALID_HANDLE_VALUE)
         CloseHandle(hFile);
   }
   // Continue processing.
   return(fFunctionOk);
}
```

在try块中使用--leave关键字会引起跳转到try块的结尾。可以认为是跳转到try块的右大括号。由于控制流自然地从try块中退出并进入finally块，所以不产生系统开销。当然，需要引入一个新的Boolean型变量fFunctionOk，用来指示函数是成功或失败。这是比较小的代价。

当按照这种方式利用结束处理程序来设计函数时，要记住在进入try块之前，要将所有资源句柄初始化为无效的值。然后，在finally块中，查看哪些资源被成功的分配，就可以知道哪些要释放。另外一种确定需要释放资源的办法是对成功分配的资源设置一个标志。然后，finally块中的代码可以检查标志的状态，来确定资源是否要释放。

23.12 关于finally块的说明

我们已经明确区分了强制执行finally块的两种情况：

- 从try块进入finally块的正常控制流。
- 局部展开：从try块的过早退出（goto、longjump、continue、break、return等）强制控制转移到finally块。

第三种情况，全局展开（global unwind），在发生的时候没有明显的标识，我们在本章前面Funcfurter1函数中已经见到。在Funcfurter1的try块中，有一个对Funcinator函数的调用。如果Funcinator函数引起一个内存访问违规（memory access violation），一个全局展开会使Funcfurter1的finally块执行。下一章将详细讨论全局展开。

由于以上三种情况中某一种的结果而导致finally块中的代码开始执行。为了确定是哪一种情况引起finally块执行，可以调用内部函数（或内蕴函数，intrinsic function）Abnormal Termination：

```
BOOL AbnormalTermination();
```

这个内部函数只在finally块中调用，返回一个Boolean值。指出与finally块相结合的try块是否过早退出。换句话说，如果控制流离开try块并自然进入finally块，AbnormalTermination将返

China-pub.com
下载

回FALSE。如果控制流非正常退出try块——通常由于goto、return、break或continue语句引起的局部展开，或由于内存访问违规或其他异常引起的全局展开——对AbnormalTermination的调用将返回TRUE。没有办法区别finally块的执行是由于全局展开还是由于局部展开。但这通常不会成为问题，因为可以避免编写执行局部展开的代码。

注意 内部函数是编译程序识别的一种特殊函数。编译程序为内部函数产生内联（inline）代码而不是生成调用函数的代码。例如，memcpy是一个内部函数（如果指定/Oi编译程序开关）。当编译程序看到一个对memcpy的调用，它直接将memcpy的代码插入调用memcpy的函数中，而不是生成一个对memcpy函数的调用。其作用是代码的长度增加了，但执行速度加快了。

23.13 Funcfurter2

这里是Funcfurter2，说明AbnormalTermination内部函数的使用。

```
DWORD Funcfurter2() {
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Request permission to access
      //    protected data, and then use it.
      WaitForSingleObject(g_hSem, INFINITE);

      dwTemp = Funcinator(g_dwProtectedData);
   }
   __finally {
      // 3. Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);

      if (!AbnormalTermination()) {
         // No errors occurred in the try block, and
         // control flowed naturally from try into finally.
         .
         .
         .
      } else {
         // Something caused an exception, and
         // because there is no code in the try block
         // that would cause a premature exit, we must
         // be executing in the finally block
         // because of a global unwind.

         // If there were a goto in the try block,
         // we wouldn't know how we got here.
         .
         .
         .
      }
   }

   // 4. Continue processing.
   return(dwTemp);
}
```

我们已经知道如何编写结束处理程序了，下一章读者会看到异常过滤程序和异常处理程序

China-pub.com 下载

更有用，更重要。在继续之前，回顾一下使用结束处理程序的理由：

- 简化错误处理，因所有的清理工作都在一个位置并且保证被执行。
- 提高程序的可读性。
- 使代码更容易维护。
- 如果使用得当，具有最小的系统开销。

23.14 SEH结束处理示例程序

SEHTerm程序，“23 SEHTerm.exe”（见清单23-1），说明了结束处理程序如何工作。这个程序的源代码和资源文件在本书附带的CD-ROM的23-SEHTerm目录下。

当运行这个程序时，主线程进入一个try块。在这个try块中显示图23-1所示的消息框。

这个消息框询问是否要程序存取一个无效的内存字节（许多程序的编写不这样考虑，只是不加询问地存取无效内存）。我们来看一看当选择Yes按钮时会发生什么。在这种情况下，线程试图向内存地址NULL写一个5。向地址NULL的写操作总要引起存取异常。当该线程引发一个存取异常时，系统显示图23-2所示的消息框（在Windows98中）。

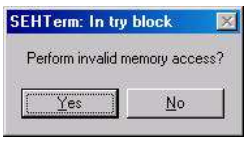


图23-1 运行SEHTerm 时显示的消息框

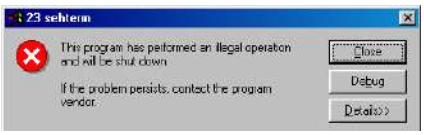


图23-2 在Windows 98中线程引发存取异常时显示的消息框

在Windows2000中，消息框如图23-3所示。

如果现在选择Close按钮（在windows98中）或OK按钮（在Windows 2000中），进程将结束。但在源代码中有一个finally块，所以在进程结束之前，finally块要先执行。这个finally块显示图23-4所示的消息框。

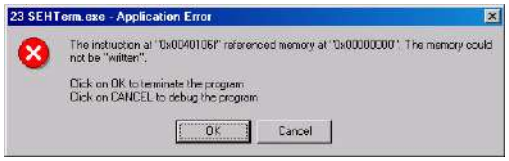


图23-3 在Windows 2000中，线程引发存取异常时显示的消息框



图23-4 SEH Term: In finally block 消息框

finally块执行是因为相应的try块非正常退出。当忽略这个消息框，进程就真的结束了。

我们再运行这个程序。这一次，我们选择No按钮，不去试图存取无效的内存。当点击No按钮，线程从try块中自然流出进入finally块。然后，finally块显示图23-5所示的消息框。

注意，这次消息框指出try块正常退出。当忽略这个消息框时，线程离开finally块，并显示图23-6所示的消息框。



图23-5 选择No按钮时出现的SEH Term: In finally block 消息框



图23-6 忽略图23-5所示的消息框时，显示的SEHTerm：After finally block 消息框

China-pub.com
下载

当忽略这个消息框时，因WinMain返回，进程自然结束时。注意当进程结束，由于有一个存取异常，你实际看不到最后一个消息框。

清单23-1 SEHTerm示例程序



SEHTerm.cpp

```
/******************************************************************************
Module:  SEHTerm.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/

#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   __try {
      int n = MessageBox(NULL, TEXT("Perform invalid memory access?"),
         TEXT("SEHTerm: In try block"), MB_YESNO);

      if (n == IDYES) {
         * (PBYTE) NULL = 5;  // This causes an access violation.
      }
   }
   __finally {
      PCTSTR psz = AbnormalTermination()
         ? TEXT("Abnormal termination") : TEXT("Normal termination");
      MessageBox(NULL, psz, TEXT("SEHTerm: In finally block"), MB_OK);
   }

   MessageBox(NULL, TEXT("Normal process termination"),
      TEXT("SEHTerm: After finally block"), MB_OK);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

China-pub.com 下载

China-pub.com
下载

第24章　异常处理程序和软件异常

异常是我们不希望有的事件。在编写程序的时候，程序员不会想去存取一个无效的内存地址或用0来除一个数值。不过，这样的错误还是常常会发生的。 CPU负责捕捉无效内存访问和用0除一个数值这种错误，并相应引发一个异常作为对这些错误的反应。 CPU引发的异常，就是所谓的硬件异常（hardware exception）。在本章的后面，我们还会看到操作系统和应用程序也可以引发相应的异常，称为软件异常（software exception）。

当出现一个硬件或软件异常时，操作系统向应用程序提供机会来考察是什么类型的异常被引发，并能够让应用程序自己来处理异常。下面就是异常处理程序的文法：

```
__try {
   // Guarded body
   .
   .
   .
}
__except (exception filter) {
   // Exception handler
   .
   .
   .
}
```

注意--except关键字。每当你建立一个try块，它必须跟随一个finally块或一个except块。一个try 块之后不能既有finally块又有except块。但可以在try-except块中嵌套try-finally块，反过来也可以。

24.1　通过例子理解异常过滤器和异常处理程序

与结束处理程序（前一章讨论过）不同，异常过滤器（ exception filter）和异常处理程序是通过操作系统直接执行的，编译程序在计算异常过滤器表达式和执行异常处理程序方面不做什么事。下面几节的内容举例说明try-except块的正常执行，解释操作系统如何以及为什么计算异常过滤器，并给出操作系统执行异常处理程序中代码的环境。

24.1.1　Funcmeister1

这里是一个try-exception块的更具体的例子。

```
DWORD Funcmeister1() {
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Perform some operation.
      dwTemp = 0;
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Handle an exception; this never executes.
```

China-pub.com
下载


```
   :
   }

   // 3. Continue processing.
   return(dwTemp);
}
```

在Funcmeister1的try块中，只是把一个0赋给dwTemp变量。这个操作决不会造成异常的引发，所以except块中的代码永远不会执行。注意这与 try-finally行为的不同。在dwTemp被设置成0之后，下一个要执行的指令是return语句。

尽管在结束处理程序的try块中使用return、goto、continue和break语句遭到强烈地反对，但在异常处理程序的try块中使用这些语句不会产生速度和代码规模方面的不良影响。这样的语句出现在与except块相结合的try块中不会引起局部展开的系统开销。

24.1.2 Funcmeister2

让我们修改这个函数，看会发生什么事情：

```
DWORD Funcmeister2() {
   DWORD dwTemp = 0;

   // 1. Do any processing here.
   :
   __try {
      // 2. Perform some operation(s).
      dwTemp = 5 / dwTemp;      // Generates an exception
      dwTemp += 10;             // Never executes
   }
   __except ( /* 3. Evaluate filter. */ EXCEPTION_EXECUTE_HANDLER) {
      // 4. Handle an exception.

      MessageBeep(0);
      :
    }

   // 5. Continue processing.
   return(dwTemp);
}
```

Funcmeister2中，try块中有一个指令试图以0来除5。CPU将捕捉这个事件，并引发一个硬件异常。当引发了这个异常时，系统将定位到except块的开头，并计算异常过滤器表达式的值，过滤器表达式的结果值只能是下面三个标识符之一，这些标识符定义在 Windows的Excpt.h文件中（见表24-1）。

表24-1 标识符及其定义

标 识 符	定 义 为
EXCEPTION_EXECUTE_HANDLER	1
EXCEPTION_CONTINUE_SEARCH	0
EXCEPTION_CONTINUE_EXECUTION	-1

China-pub.com 下载

下面几节将讨论这些标识符如何改变线程的执行。在阅读这些内容时可参阅图 24-1，该图概括了系统如何处理一个异常的情况。

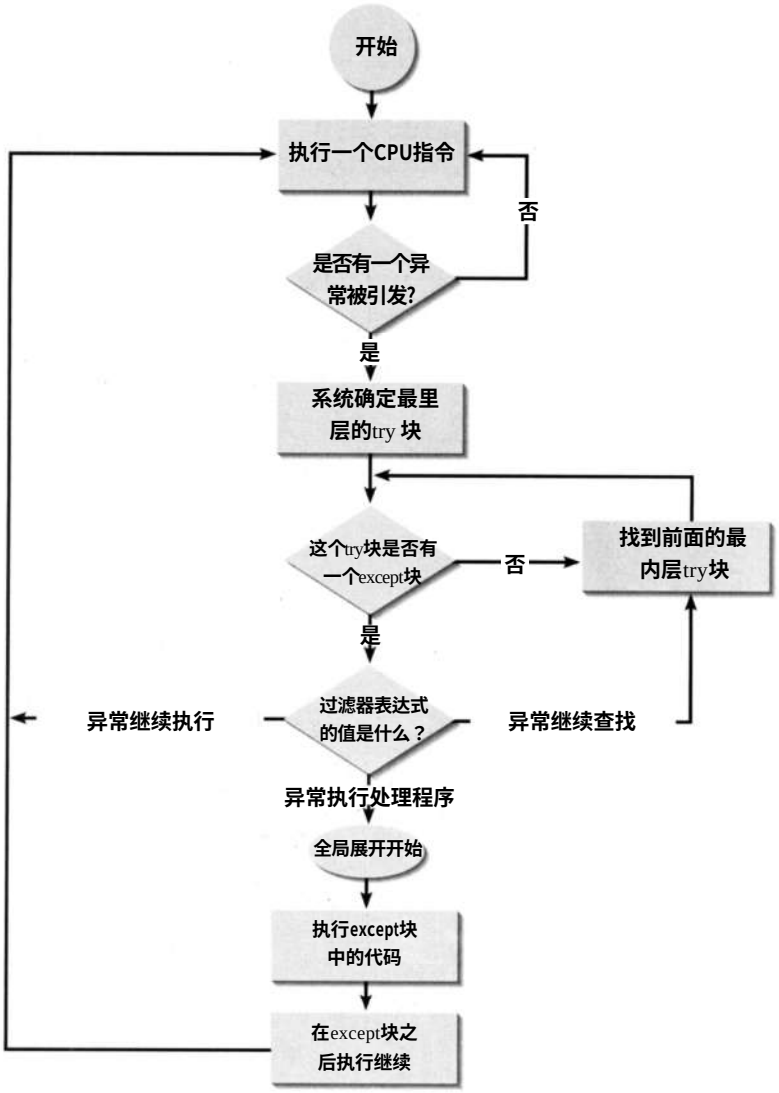


图24-1 系统如何处理一个异常

24.2 EXCEPTION_EXECUTE_HANDLER

在Funcmeister2中，异常过滤器表达式的值是EXCEPTION_EXECUTE_HANDLER。这个值的意思是要告诉系统："我认出了这个异常。即，我感觉这个异常可能在某个时候发生，我已编写了代码来处理这个问题，现在我想执行这个代码。"在这个时候，系统执行一个全局展开（本章后面将讨论），然后执行向except块中代码（异常处理程序代码）的跳转。在except块中代码执行完之后，系统考虑这个要被处理的异常并允许应用程序继续执行。这种机制使Windows应用程序可以抓住错误并处理错误，再使程序继续运行，不需要用户知道错误的发生。

但是，当except块执行后，代码将从何处恢复执行？稍加思索，我们就可以想到几种可能性。

China-pub.com
下载

第一种可能性是从产生异常的CPU指令之后恢复执行。在Funcmeister2中执行将从对dwTemp加10的指令开始恢复。这看起来像是合理的做法，但实际上，很多程序的编写方式使得当前面的指令出错时，后续的指令不能够继续成功地执行。

在Funcmeister2中，代码可以继续正常执行，但是，Funcmeister2已不是正常的情况。代码应该尽可能地结构化，这样，在产生异常的指令之后的CPU指令有望获得有效的返回值。例如，可能有一个指令分配内存，后面一系列指令要执行对该内存的操作。如果内存不能够被分配，则所有后续的指令都将失败，上面这个程序重复地产生异常。

这里是另外一个例子，说明为什么在一个失败的CPU指令之后，执行不能够继续。我们用下面的程序行来替代Funcmeister2中产生异常的C语句：

```
malloc(5 / dwTemp);
```

对上面的程序行，编译程序产生CPU指令来执行除法，将结果压入栈中，并调用malloc函数。如果除法失败，代码就不能继续执行。系统必须向栈中压东西，否则，栈就被破坏了。

所幸的是，微软没有让系统从产生异常的指令之后恢复指令的执行。这种决策使我们免于面对上面的问题。

第二种可能性是从产生异常的指令恢复执行。这是很有意思的可能性。如果在except块中有这样的语句会怎么样呢：

```
dwTemp = 2;
```

在except块中有了这个赋值语句，可以从产生异常的指令恢复执行。这一次，将用2来除5，执行将继续，不会产生其他的异常。可以做些修改，让系统重新执行产生异常的指令。你会发现这种方法将导致某些微妙的行为。我们将在“EXCEPTION_ CONTINUE_EXECUTION”一节中讨论这种技术。

第三种可能性是从except块之后的第一条指令开始恢复执行。这实际是当异常过滤器表达式的值为EXCEPTION_EXECUTE_HANDLER时所发生的事。在except块中的代码结束执行后，控制从except块之后的第一条指令恢复。

24.2.1 一些有用的例子

假如要实现一个完全强壮的应用程序，该程序需要每周7天，每天24小时运行。在今天的世界里，软件变得这么复杂，有那么多的变量和因子来影响程序的性能，笔者认为如果不用SEH，要实现完全强壮的应用程序简直是不可能的。我们先来看一个样板程序，即C的运行时函数strcpy：

```
char* strcpy(
   char* strDestination,
   const char* strSource);
```

这是一个相当简单的函数，它怎么会引起一个进程结束呢？如果调用者对这些参数中的某一个传递NULL（或任何无效的地址），strcpy就引起一个存取异常，并且导致整个进程结束。

使用SEH，就可以建立一个完全强壮的strcpy函数：

```
char* RobustStrCpy(char* strDestination, const char* strSource) {

   __try {
      strcpy(strDestination, strSource);
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Nothing to do here
   }
```

China-pub.com 下载

```
   return(strDestination);
}
```

这个函数所做的一切就是将对strcpy的调用置于一个结构化的异常处理框架中。如果strcpy执行成功，函数就返回。如果strcpy引起一个存取异常，异常过滤器返回EXCEPTION_EXECUTE_HANDLER，导致该线程执行异常处理程序代码。在这个函数中，处理程序代码什么也不做，RobustStrCpy只是返回到它的调用者，根本不会造成进程结束。

我们再看另外一个例子。这个函数返回一个字符串里的以空格分界的符号个数：

```
int RobustHowManyToken(const char* str) {

   int nHowManyTokens = -1;  // -1 indicates failure
   char* strTemp = NULL;     // Assume failure

   __try {

      // Allocate a temporary buffer
      strTemp = (char*) malloc(strlen(str) + 1);

      // Copy the original string to the temporary buffer
      strcpy(strTemp, str);

      // Get the first token
      char* pszToken = strtok(strTemp, " ");

      // Iterate through all the tokens
      for (; pszToken != NULL; pszToken = strtok(NULL, " "))
         nHowManyTokens++;

      nHowManyTokens++;       // Add 1 since we started at -1
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Nothing to do here
   }
   // Free the temporary buffer (guaranteed)
   free(strTemp);

   return(nHowManyTokens);
}
```

这个函数分配一个临时缓冲区并将一个字符串复制到里面。然后该函数用C运行时函数strtok来获取字符串的符号。临时缓冲区是必要的，因strtok要修改它所操作的串。感谢有了SEH，这个非常简单的函数就处理了所有的可能性。我们来看在几个不同的情况下函数是如何执行的。

首先，如果调用者向函数传递了NULL（或任何无效的内存地址），nHowManyTokens被初始化成-1。在try块中对strlen的调用会引起存取异常。异常过滤器获得控制并将控制转移给except块，except块什么也不做。在except块之后，调用free来释放临时内存块。但是，这个内存从未分配，所以结束调用free，向它传递NULL作为参数。ANSI C明确说明用NULL作为参数调用free是合法的。这时free什么也不做，这并不是错误。最后，函数返回-1，指出失败。注意进程并没有结束。

其次，调用者可能向函数传递了一个有效的地址，但对malloc的调用（在try块中）可能失败并返回NULL。这将导致对strcpy的调用引起一个存取异常。同样，异常过滤器被调用，except块执行（什么也不做），free被调用，传递给它NULL（什么也不做），返回-1，告诉调用程序该函数失败。注意进程也没有结束。

最后，假定调用者向函数传递了一个有效的地址，并且对malloc的调用也成功了。这种情况下，其余的代码也会成功地在nHowManyTokens变量中计算符号的数量。在try块的结尾，异

China-pub.com
下载

常过滤器不会被求值，except块中代码不会被执行，临时内存缓冲区将被释放，并向调用者返回nHowManyTokens。

使用SEH会感觉很好。RobustHowManyToken函数说明了如何在不使用try-finally的情况下保证释放资源。在异常处理程序之后的代码也都能保证被执行（假定函数没有从try块中返回——应避免的事情）。

我们再看一个特别有用的SEH例子。这里的函数重复一个内存块：

```
PBYTE RobustMemDup(PBYTE pbSrc, size_t cb) {

   PBYTE pbDup = NULL;        // Assume failure

   __try {

      // Allocate a buffer for the duplicate memory block
      pbDup = (PBYTE) malloc(cb);

      memcpy(pbDup, pbSrc, cb);
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      free(pbDup);
      pbDup = NULL;
   }

   return(pbDup);
}
```

这个函数分配一个内存缓冲区，并从源块向目的块复制字节。然后函数将复制的内存缓冲区的地址返回给调用程序（如果函数失败则返回NULL）。希望调用程序在不需要缓冲区时释放它。这是在except块中实际有代码的第一个例子。我们看一看这个函数在不同条件下是如何执行的。

- 如果调用程序对pbSrc参数传递了一个无效地址，或者如果对malloc的调用失败（返回NULL），memcpy将引起一个存取异常。该存取异常执行过滤器，将控制转移到except块。在except块内，内存缓冲区被释放，pbDup被设置成NULL以便调用程序能够知道函数失败。这里，注意ANSI C允许对free传递NULL。
- 如果调用程序给函数传递一个有效地址，并且如果对malloc的调用成功，则新分配内存块的地址返回给调用程序。

24.2.2 全局展开

当一个异常过滤器的值为EXCEPTION_EXECUTE_HANDLER时，系统必须执行一个全局展开(global unwind)。这个全局展开使所有那些在处理异常的y_except块之后开始执行但未完成的try-finally块恢复执行。图24-2是描述系统如何执行全局展开的流程图，在解释后面的例子时，请参阅这个图。

```
void FuncOStimpyl() {

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Call another function.
      FuncORen1();

      // Code here never executes.
   }

   __except ( /* 6. Evaluate filter. */ EXCEPTION_EXECUTE_HANDLER) {
      // 8. After the unwind, the exception handler executes.
```

China-pub.com 下载

```
      MessageBox(…);
   }

   // 9. Exception handled--continue execution.
   :
   :
}
void FuncORen1() {
   DWORD dwTemp = 0;

   // 3. Do any processing here.
   :
   :
   __try {
      // 4. Request permission to access protected data.
      WaitForSingleObject(g_hSem, INFINITE);

      // 5. Modify the data.
      //    An exception is generated here.
      g_dwProtectedData = 5 / dwTemp;
   }
   __finally {
      // 7. Global unwind occurs because filter evaluated
      //    to EXCEPTION_EXECUTE_HANDLER.

      // Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // Continue processing--never executes.
   :
   :
}
```

函数FuncOStimpy1和FuncOren1结合起来可以解释SEH最令人疑惑的方面。程序中注释的标号给出了执行的次序，我们现在开始做一些分析。

FuncOStimpy1开始执行，进入它的try块并调用FuncORen1。FuncORen1开始执行，进入它的try块并等待获得信标。当它得到信标，FuncORen1试图改变全局数据变量g_dwProtectedData。但由于除以0而产生一个异常。系统因此取得控制，开始搜索一个与 except块相配的try块。因为FuncORen1中的try与同一个finally块相配，所以系统再上溯寻找另外的try块。这里，系统在FuncOStimpy1中找到一个try块，并且发现这个try块与一个except块相配。

系统现在计算与FuncOStimpy1中except块相联的异常过滤器的值，并等待返回值。当系统看到返回值是EXCEPTION_EXECUTE_HANDLER的，系统就在FuncORen1的finally块中开始一个全局展开。注意这个展开是在系统执行 FuncOStimpy1的except块中的代码之前发生的。对于一个全局展开，系统回到所有未完成的 try块的结尾，查找与 finally块相配的try块。在这里，系统发现的finally块是FuncORen1中所包含的finally块。

当系统执行 FuncORen1的finally块中的代码时，就可以清楚地看到 SEH的作用了。FuncORen1释放信标，使另一个线程恢复执行。如果这个 finally块中不包含ReleaseSemaphore的调用，则信标不会被释放。

在finally块中包含的代码执行完之后，系统继续上溯，查找需要执行的未完成 finally块。在这个例子中已经没有这样的 finally块了。系统到达要处理异常的try-except块就停止上溯。这时，全局展开结束，系统可以执行except块中所包含的代码。

结构化异常处理就是这样工作的。 SEH比较难于理解，是因为在代码的执行当中与系统牵

China-pub.com
下载

扯太多。程序代码不再是从头到尾执行，系统使代码段按照它的规定次序执行。这种执行次序虽然复杂，但可以预料。按图24-1和图24-2的流程图去做，就可以有把握地使用SEH。

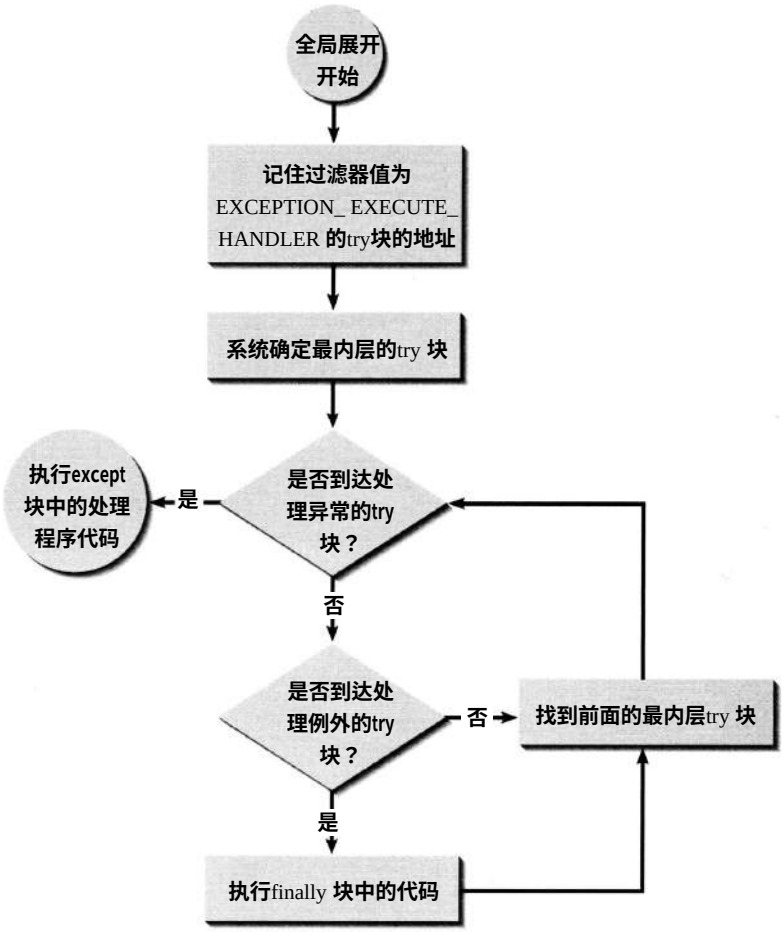


图24-2 系统如何执行一个全局展开

为了更好地理解这个执行次序，我们再从不同的角度来看发生的事情。当一个过滤器返回EXCEPTION_EXECUTE_HANDLER时，过滤器是在告诉系统，线程的指令指针应该指向except块中的代码。但这个指令指针在FuncORen1的try块里。回忆一下第23章，每当一个线程要从一个try-finally块离开时，必须保证执行finally块中的代码。在发生异常时，全局展开就是保证这条规则的机制。

24.2.3 暂停全局展开

通过在finally块里放入一个return语句，可以阻止系统去完成一个全局展开。请看下面的代码：

```
void FuncMonkey() {
   __try {
      FuncFish();
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      MessageBeep(0);
   }
   MessageBox(…);
```

China-pub.com 下载

```
}

void FuncFish() {
   FuncPheasant();
   MessageBox(…);
}

void FuncPheasant() {

   __try {
      strcpy(NULL, NULL);
   }

   __finally {
      return;
   }
}
```

在FuncPheasant的try块中，当调用strcpy函数时，会引发一个内存存取异常。当异常发生时，系统开始查看是否有一个过滤器可以处理这个异常。系统会发现在 FuncMonkey中的异常过滤器是处理这个异常的，并且系统开始一个全局展开。

全局展开启动，先执行FuncPheasant的finally块中的代码。这个代码块包含一个return语句。这个return语句使系统停止做展开，FuncPheasant将实际返回到FuncFish。然后FuncFish又返回到函数FuncMonkey。FuncMonkey中的代码继续执行，调用MessageBox。

注意FuncMonkey的异常块中的代码从不会执行对 MessageBeep的调用。FuncPheasant的finally块中的return语句使系统完全停止了展开，继续执行，就像什么也没有发生。

微软专门设计SEH按这种方式工作。程序员有可能希望使展开停止，让代码继续执行下去。这种方法为程序员提供了一种手段。原则上，应该小心避免在 finally块中安排return语句。

24.3 EXCEPTION_CONTINUE_EXECUTION

我们再仔细考察一下异常过滤器，看它是如何计算出定义在 Excpt.h中的三个异常标识符之一的。在“ Funcmeister2 ”一节中，为简单起见，在过滤器里直接硬编码了标识符EXCEPTION_EXECUTE_HANDLER，但实际上可以让过滤器调用一个函数确定应该返回哪一个标识符。这里是另外一个例子。

```
char g_szBuffer[100];

void FunclinRoosevelt1() {
   int x = 0;
   char *pchBuffer = NULL;

   __try {
      *pchBuffer = 'J';
      x = 5 / x;
   }
   __except (OilFilter1(&pchBuffer)) {
      MessageBox(NULL, "An exception occurred", NULL, MB_OK);
   }
   MessageBox(NULL, "Function completed", NULL, MB_OK);
}
```

China-pub.com
下载


```
LONG OilFilter1(char **ppchBuffer) {
   if (*ppchBuffer == NULL) {
      *ppchBuffer = g_szBuffer;
      return(EXCEPTION_CONTINUE_EXECUTION);
   }
   return(EXCEPTION_EXECUTE_HANDLER);
}
```

这里，首先遇到的问题是在我们试图向pchBuffer所指向的缓冲区中放入一个字母‘J’时发生的。因为这里没有初始化pchBuffer，使它指向全局缓冲区g_szBuffur。pchBuffer实际指向NULL。CPU将产生一个异常，并计算与异常发生的try块相关联的except块的异常过滤器。在except块中，对OilFilter函数传递了pchBuffer变量的地址。

当OilFilter获得控制时，它要查看*ppchBuffer是不是NULL，如果是，把它设置成指向全局缓冲区g_szBuffer。然后这个过滤器返回EXCEPTION_CONTINUE_EXECUTION。当系统看到过滤器的值是EXCEPTION_CONTINUE_EXECUTION时，系统跳回到产生异常的指令，试图再执行一次。这一次，指令将执行成功，字母‘J’将放在g_szBuffer的第一个字节。

随着代码继续执行，我们又在try块中碰到除以0的问题。系统又要计算过滤器的值。这一次，OilFilter看到*ppchBuffer不是NULL，就返回EXCEPTION_EXECUTE_HANDLER，这是告诉系统去执行except块中的代码。这会显示一个消息框，用文本串报告发生了异常。

如你所见，在异常过滤器中可以做很多的事情。当然过滤器必须返回三个异常标识符之一，但可以执行任何其他你想执行的任务。

使用带警告的EXCEPTION_CONTINUE_EXECUTION

我们已经知道，修改FunclinRoosevelt1函数中的问题可使系统继续执行，也可能不执行，这要取决于程序的目标CPU，取决于编译程序为C／C++语句生成的指令，取决于编译程序的选项设置。

一个编译程序对下面的C/C++语句可能生成两条机器指令：

```
*pchBuffer = 'J';
```

机器指令可能是这个样子：

```
MOV EAX, [pchBuffer]   // Move the address into a register
MOV [EAX], 'J'         // Move 'J' into the address
```

第二条指令产生异常。异常过滤器可以捕获这个异常，修改pchBuffer的值，并告诉系统重新执行第二条CPU指令。但问题是，寄存器的值可能不改变，不能反映装入到pchBuffer的新值，这样重新执行CPU指令又产生另一个异常。这就发生了死循环。

如果编译程序优化了代码，继续执行可能顺利；如果编译程序没有优化代码，继续执行就可能失败。这可能是个非常难修复的bug，需要检查源代码生成的汇编语言程序，确定程序出了什么错。这个例子的寓意就是在异常过滤器返回EXCEPTION_CONTINUE_EXECUTION时，要特别地小心。

有一种情况可保证EXCEPTION_CONTINUE_EXECUTION每次总能成功：当离散地向一个保留区域提交存储区时。在第15章讨论过如何保存一个大的地址空间，并向这个地址空间离散地提交存储区。VMAlloc示例程序说明了这个例子。编写VMAlloc程序的一种更好的办法是必要时使用SEH提交存储区，而不是每次调用Virtual Alloc函数。

在第16章，我们讨论了线程栈。特别是，我们讲解了系统如何为线程的栈保留一个1MB的地址空间范围，以及在线程需要内存区时，系统如何自动向栈提交新的内存区。为此，系统在

China-pub.com 下载

内部建立了一个SEH框架。当一个线程试图去存取并不存在的栈存储区时，就产生一个异常。系统的异常过滤器可以确定这个异常是源于试图存取栈的保留地址空间。异常过滤器调用VirtualAlloc向线程的栈提交更多的存储区，然后过滤器返回 EXCEPTION_CONTINUE_EXECUTION。这时，试图存取栈存储区的CPU指令可以成功执行，线程可以继续运行。

将虚拟内存技术同结构化异常处理结合起来，可以编写一些执行非常快，非常高效的程序。下一章的SpreadSheet示例程序将说明如何使用SEH有效地实现内存管理。这个代码执行得非常快。

24.4 EXCEPTION_CONTINUE_SEARCH

迄今为止我们看到的例子都很平常。通过增加一个函数调用，让我们来看看其他方面的问题：

```
char g_szBuffer[100];

void FunclinRoosevelt2() {
   char *pchBuffer = NULL;

   __try {
      FuncAtude2(pchBuffer);
   }
   __except (OilFilter2(&pchBuffer)) {
      MessageBox(…);
   }
}

void FuncAtude2(char *sz) {
   *sz = 0;
}

LONG OilFilter2 (char **ppchBuffer) {
   if (*ppchBuffer == NULL) {
      *ppchBuffer = g_szBuffer;
      return(EXCEPTION_CONTINUE_EXECUTION);
   }
   return(EXCEPTION_EXECUTE_HANDLER);
}
```

当FunclinRoosevelt2执行时，它调用FuncAtude2并传递参数NULL。当FuncAtude2执行时，引发了一个异常。同前面的例子一样，系统计算与最近执行的 try块相关联的异常过滤器的值。在这个例子中，FunclinRoosevelt2中的try块是最近执行的try块，所以系统调用OilFilter2函数来求异常过滤器的值——尽管这个异常是在FunclinAtude2函数中产生的。

现在我们让问题变得更复杂一点，在程序中再增加一个try_except块。

```
char g_szBuffer[100];

void FunclinRoosevelt3() {

   char *pchBuffer = NULL;

   __try {
      FuncAtude3(pchBuffer);
   }
```

China-pub.com
下载


```
    __except (OilFilter3(&pchBuffer)) {
       MessageBox(…);
    }
}
void FuncAtude3(char *sz) {
    __try {
       *sz = 0;
    }
    __except (EXCEPTION_CONTINUE_SEARCH) {
       // This never executes.
       .
       .
       .
    }
}

LONG OilFilter3(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
       *ppchBuffer = g_szBuffer;
       return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

现在，当FuncAtude 3试图向地址NULL里存放0时，会引发一个异常。但这时将执行FuncAtude3的异常过滤器。FuncAtude3的异常过滤器很简单，只是取值EXCEPTION_CONTINUE_SEARCH。这个标识符是告诉系统去查找前面与一个except块相匹配的try块，并调用这个try块的异常处理器。

因为FuncAtude3的过滤器的值为EXCEPTION_CONTINUE_SEARCH，系统将查找前面的try块（在FunclinRoosevelt3里），并计算其异常过滤器的值，这里异常过滤器是OilFilter3。OilFilter3看到pchBuffer是NULL，将pchBuffer设定为指向全局缓冲区，然后告诉系统恢复执行产生异常的指令。这将使FuncAtude3的try块中的代码执行，但不幸的是，FuncAtude3的局部变量sz没有变化，恢复执行失败的指令只是产生另一个异常。这样，又造成死循环。

前面说过，系统要查找最近执行的与except块相匹配的try块，并计算它的过滤器值。这就是说，系统在查找过程当中，将略过那些与finally块相匹配而不是与except块相匹配的try块。这样做的理由很明显：finally块没有异常过滤器，系统没有什么要计算的。如果前面例子中FuncAtude3包含一个finally块而不是except块，系统将在一开始就通过FunclinRoosevelt3的OilFilter3计算异常过滤器的值。

第25章提供有关EXCEPTION_CONTINUE_SEARCH的更多信息。

24.5 GetExceptionCode

一个异常过滤器在确定要返回什么值之前，必须分析具体情况。例如，异常处理程序可能知道发生了除以0引起的异常时该怎么做，但是不知道该如何处理一个内存存取异常。异常过滤器负责检查实际情况并返回适当的值。

下面的代码举例说明了一种方法，指出所发生异常的类别：

```
__try {
    x = 0;
    y = 4 / x;
}
```

China-pub.com 下载

```
__except ((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO) ?
   EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
   // Handle divide by zero exception.
}
```

内部函数GetExceptionCode返回一个值，该值指出所发生异常的种类：

```
DWORD GetExceptionCode();
```

下面列出所有预定义的异常和相应的含意，这些内容取自 Platform SDK文档。这些异常标识符可以在WinBase.h文件中找到。我们对这些异常做了分类。

1. 与内存有关的异常

- EXCEPTION_ACCESS_VIOLATION。线程试图对一个虚地址进行读或写，但没有做适当的存取。这是最常见的异常。
- EXCEPTION_DATATYPE_MISALIGNMENT。线程试图读或写不支持对齐（alignment）的硬件上的未对齐的数据。例如，16位数值必须对齐在2字节边界上，32位数值要对齐在4字节边界上。
- EXCEPTION_ARRAY_BOUNDS_EXCEEDED。线程试图存取一个越界的数组元素，相应的硬件支持边界检查。
- EXCEPTION_IN_PAGE_ERROR。由于文件系统或一个设备启动程序返回一个读错误，造成不能满足要求的页故障。
- EXCEPTION_GUARD_PAGE。一个线程试图存取一个带有PAGE_GUARD保护属性的内存页。该页是可存取的，并引起一个EXCEPTION_GUARD_PAGE异常。
- EXCEPTION_STACK_OVERFLOW。线程用完了分配给它的所有栈空间。
- EXCEPTION_ILLEGAL_INSTRUCTION。线程执行了一个无效的指令。这个异常由特定的CPU结构来定义；在不同的CPU上，执行一个无效指令可引起一个陷井错误。
- EXCEPTION_PRIV_INSTRUCTION。线程执行一个指令，其操作在当前机器模式中不允许。

2. 与异常相关的异常

- EXCEPTION_INVALID_DISPOSITION。一个异常过滤器返回一值，这个值不是EXCEPTION_EXECUTE_HANDLER、EXCEPTION_CONTINUE_SEARCH、EXCEPTION_CONTINUE_EXECUTION三者之一。
- EXCEPTION_NONCONTINUABLE_EXCEPTION。一个异常过滤器对一个不能继续的异常返回EXCEPTION_CONTINUE_EXECUTION。

3. 与调试有关的异常

- EXCEPTION_BREAKPOINT。遇到一个断点。
- EXCEPTION_SINGLE_STEP。一个跟踪陷井或其他单步指令机制告知一个指令已执行完毕。
- EXCEPTION_INVALID_HANDLE。向一个函数传递了一个无效句柄。

4. 与整数有关的异常

- EXCEPTION_INT_DIVIDE_BY_ZERO。线程试图用整数0来除一个整数
- EXCEPTION_INT_OVERFLOW。一个整数操作的结果超过了整数值规定的范围。

5. 与浮点数有关的异常

- EXCEPTION_FLT_DENORMAL_OPERAND。浮点操作中的一个操作数不正常。不正常的值是一个太小的值，不能表示标准的浮点值。
- EXCEPTION_FLT_DIVIDE_BY_ZERO。线程试图用浮点数0来除一个浮点。

China-pub.com
下载

- EXCEPTION_FLT_INEXACT_RESULT。浮点操作的结果不能精确表示成十进制小数。
- EXCEPTION_FLT_INVALID_OPERATION。表示任何没有在此列出的其他浮点数异常。
- EXCEPTION_FLT_OVERFLOW。浮点操作的结果超过了允许的值。
- EXCEPTION_FLT_STACK_CHECK。由于浮点操作造成栈溢出或下溢。
- EXCEPTION_FLT_UNDERFLOW。浮点操作的结果小于允许的值。

内部函数GetExceptionCode只能在一个过滤器中调用（--except之后的括号里），或在一个异常处理程序中被调用。下面的代码是合法的：

```
__try {
   y = 0;
   x = 4 / y;
}

__except (
   ((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ||
    (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)) ?
   EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {

   switch (GetExceptionCode()) {
      case EXCEPTION_ACCESS_VIOLATION:
         // Handle the access violation.
         .
         .
         .
         break;
      case EXCEPTION_INT_DIVIDE_BY_ZERO:
         // Handle the integer divide by 0.
         .
         .
         .
         break;
   }
}
```

但是，不能在一个异常过滤器函数里面调用 GetExceptionCode。编译程序会捕捉这样的错误。当编译下面的代码时，将产生编译错。

```
__try {
   y = 0;
   x = 4 / y;
}

__except (CoffeeFilter()) {

   // Handle the exception.
   .
   .
   .
}

LONG CoffeeFilter (void) {
   // Compilation error: illegal call to GetExceptionCode.
   return((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

可以按下面的形式改写代码：

```
__try {
```

China-pub.com 下载

```
   y = 0;
   x = 4 / y;
}

__except (CoffeeFilter(GetExceptionCode())) {

   // Handle the exception.
   :
   :
}

LONG CoffeeFilter (DWORD dwExceptionCode) {
   return((dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

异常代码遵循在WinError.h文件中定义的有关错误代码的规则。每个DWORD被划分如表24-2所示。

表24-2 一个错误代码的构成

位	31-30	29	28	27-16	15-0
内容	严重性系数	微软/客户	保留	设备代码	异常代码
意义	0=成功 1=信息 2=警告 3=错误	0=微软定义的代码 1=客户定义的代码	必须为0	微软定义(见表24-3)	微软／客户定义

目前，微软定义了下面一些设备代码(见表24-3)。

表24-3 设备代码及其值

设备代码	值	设备代码	值
FACILITY_NULL	0	FACILITY_CONTROL	10
FACILITY_RPC	1	FACILITY_CERT	11
FACILITY_DISPATCH	2	FACILITY_INTERNET	12
FACILITY_STORAGE	3	FACILITY_MEDIASERVER	13
FACILITY_ITF	4	FACILITY_MSMQ	14
FACILITY_WIN32	7	FACILITY_SETUPAPI	15
FACILITY_WINDOWS	8	FACILITY_SCARD	16
FACILITY_SECURITY	9	FACILITY_COMPLUS	17

我们将EXCEPTION_ACCESS_VIOLATION异常代码拆开来，看各位(bit)都是什么。在WinBase.h中找到EXCEPTION_ACCESS_VIOLATION，它的值为0xC0000005：

C 0 0 0 0 0 0 5 （十六进制）

1100 0000 0000 0000 0000 0000 0000 0101 （二进制）

第30位和第31位都是1，表示存取异常是一个错误（线程不能继续运行）。第29位是0，表示Microsoft已经定义了这个代码。第28位是0，留待后用。第16位至27位是0，代表FACILITY_NULL（存取异常可发生在系统中任何地方，不是使用特定设备才发生的异常）。第0位到第15位包含一个数5，表示微软将存取异常这种异常的代码定义成5。

24.6 GetExceptionInformation

当一个异常发生时，操作系统要向引起异常的线程的栈里压入三个结构，这三个结构是：

China-pub.com
下载

EXCEPTION_RECORD结构、CONTEXT结构和EXCEPTION_POINTERS结构。

EXCEPTION_RECORD结构包含有关已发生异常的独立于CPU的信息，CONTEXT结构包含已发生异常的依赖于CPU的信息。EXCEPTION_POINTERS结构只有两个数据成员，二者都是指针，分别指向被压入栈的EXCEPTION_RECORD和CONTEXT结构：

```
typedef struct _EXCEPTION_POINTERS {
   PEXCEPTION_RECORD ExceptionRecord;
   PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

为了取得这些信息并在你自己的程序中使用这些信息，需要调用 GetExceptionInformation 函数：

```
PEXCEPTION_POINTERS GetExceptionInformation();
```

这个内部函数返回一个指向EXCEPTION_POINTERS结构的指针。

关于GetExceptionInformation函数，要记住的最重要事情是它只能在异常过滤器中调用，因为仅仅在处理异常过滤器时， CONTEXT、EXCEPTION_RECORD和EXCEPTION_POINTERS才是有效的。一旦控制被转移到异常处理程序，栈中的数据就被删除。

如果需要在你的异常处理程序块里面存取这些异常信息（虽然很少有必要这样做），必须将EXCEPTION_POINTERS结构所指向的CONTEXT数据结构和／或EXCEPTION_RECORD数据结构保存在你所建立的一个或多个变量里。下面的代码说明了如何保存 EXCEPTION_RECORD和CONTEXT数据结构：

```
void FuncSkunk() {
   // Declare variables that we can use to save the exception
   // record and the context if an exception should occur.
   EXCEPTION_RECORD SavedExceptRec;
   CONTEXT SavedContext;
   :
   __try {
      :
   }

   __except (
      SavedExceptRec =
         *(GetExceptionInformation())->ExceptionRecord,
      SavedContext =
         *(GetExceptionInformation())->ContextRecord,
      EXCEPTION_EXECUTE_HANDLER) {

      // We can use the SavedExceptRec and SavedContext
      // variables inside the handler code block.
      switch (SavedExceptRec.ExceptionCode) {
         :
      }
   }
   :
}
```

China-pub.com 下载

注意在异常过滤器中C语言逗号（,）操作符的使用。许多程序员不习惯使用这个操作符。它告诉编译程序从左到右执行以逗号分隔的各表达式。当所有的表达式被求值之后，返回最后的（或最右的）表达式的结果。

在FuncSkunk中，左边的表达式将执行，将栈中的 EXCEPTION_RECORD结构保存在Saved ExceptRec局部变量里。这个表达式的结果是SavedExceptRec的值。这个结果被丢弃，再计算右边下一个表达式。第二个表达式将栈中的 CONTEXT结构保存在SavedContext局部变量里。第二个表达式的结果是 SavedContext，同样当计算第三个表达式时丢弃第二个表达式的结果。第三个表达式很简单，只是一个数值EXCEPTION_EXECUTE_HANDLER。这个最右边的表达式的结果就是整个由逗号分隔的表达式组的结果。

由于异常过滤器的值是 EXCEPTION_EXECUTE_HANDLER，except块中的代码要执行。这时，已被初始过的 SavedExceptRec和SavedContext变量可以在 except块中使用。要记住，SavedExceptRec和SavedContext变量要在try块之外说明，这一点很重要。

我们都可以猜到， EXCEPTION_POINTERS结构的 ExceptionRecord成员指向EXCEPTION_RECORD结构：

```
typedef struct _EXCEPTION_RECORD {
   DWORD ExceptionCode;
   DWORD ExceptionFlags;
   struct _EXCEPTION_RECORD *ExceptionRecord;
   PVOID ExceptionAddress;
   DWORD NumberParameters;
   ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

EXCEPTION_RECORD结构包含有关最近发生的异常的详细信息，这些信息独立于 CPU：

- ExceptionCode包含异常的代码。这同内部函数GetExceptionCode返回的信息是一样的。
- ExceptionFlags包含有关异常的标志。当前只有两个值，分别是 0（指出一个可以继续的异常）和EXCEPTION_NONCONTINUABLE（指出一个不可继续的异常）。在一个不可继续的异常之后，若要继续执行，会引发一个 EXCEPTION_NONCONTINUABLE_EXCEPTION异常。
- ExceptionRecord指向另一个未处理异常的EXCEPTION_RECORD结构。在处理一个异常的时候，有可能引发另外一个异常。例如，异常过滤器中的代码就可能用零来除一个数。当嵌套异常发生时，可将异常记录链接起来，以提供另外的信息。如果在处理一个异常过滤器的过程当中又产生一个异常，就发生了嵌套异常。如果没有未处理异常，这个成员就包含一个NULL。
- ExceptionAddress指出产生异常的CPU指令的地址。
- NumberParameters规定了与异常相联系的参数数量（0到15）。这是在ExceptionInformation数组中定义的元素数量。对几乎所有的异常来说，这个值都是零。
- ExceptionInformation规定一个附加参数的数组，用来描述异常。对大多数异常来说，数组元素是未定义的。

EXCEPTION_RECORD结构的最后两个成员，NumberParameters和ExceptionInformation向异常过滤器提供一些有关异常的附加信息。目前只有一种类型的异常提供附加信息，就是EXCEPTION_ACCESS_VIOLATION。所有其他可能的异常都将 NumberParameters设置成零。我们可以检验ExceptionInformation的数组成员来查看关于所产生异常的附加信息。

对于一个EXCEPTION_ACCESS_VIOLATION异常来说，ExceptionInformation[0]包含一个

China-pub.com
下载

标志，指出引发这个存取异常的操作的类型。如果这个值是 0，表示线程试图要读不可访问的数据。如果这个值是1，表示线程要写不可访问的数据。ExceptionInformation[1] 指出不可访问数据的地址。

通过使用这些成员，我们可以构造异常过滤器，提供大量有关程序的信息。例如，可以这样编写异常过滤器：

```
__try {
   :
}
__except (ExpFltr(GetExceptionInformation()->ExceptionRecord)) {
   :
}

LONG ExpFltr (PEXCEPTION_RECORD pER) {
   char szBuf[300], *p;
   DWORD dwExceptionCode = pER->ExceptionCode;

   sprintf(szBuf, "Code = %x, Address = %p",
      dwExceptionCode, pER->ExceptionAddress);
   // Find the end of the string.
   p = strchr(szBuf, 0);

   // I used a switch statement in case Microsoft adds
   // information for other exception codes in the future.
   switch (dwExceptionCode) {
      case EXCEPTION_ACCESS_VIOLATION:
         sprintf(p, "Attempt to %s data at address %p",
            pER->ExceptionInformation[0] ? "write" : "read",
            pER->ExceptionInformation[1]);
         break;

      default:
         break;
   }

   MessageBox(NULL, szBuf, "Exception", MB_OK | MB_ICONEXCLAMATION);

   return(EXCEPTION_CONTINUE_SEARCH);
}
```

EXCEPTION_POINTERS结构的ContextRecord成员指向一个CONTEXT结构（第7章讨论过）。这个结构是依赖于平台的，也就是说，对于不同的CPU平台，这个结构的内容也不一样。

本质上，对CPU上每一个可用的寄存器，这个结构相应地包含一个成员。当一个异常被引发时，可以通过检验这个结构的成员找到更多的信息。遗憾的是，为了得到这种可能的好处，要求程序员编写依赖于平台的代码，以确认程序所运行的机器，使用适当的 CONTEXT结构。最好的办法是在代码中安排一个 #ifdefs指令。Windows支持的不同CPU的CONTEXT结构定义在WinNT.h文件中。

24.7 软件异常

迄今为止，我们一直在讨论硬件异常，也就是 CPU捕获一个事件并引发一个异常。在代码

China-pub.com 下载

中也可以强制引发一个异常。这也是一个函数向它的调用者报告失败的一种方法。传统上，失败的函数要返回一些特殊的值来指出失败。函数的调用者应该检查这些特殊值并采取一种替代的动作。通常，这个调用者要清除所做的事情并将它自己的失败代码返回给它的调用者。这种错误代码的逐层传递会使源程序的代码变得非常难于编写和维护。

另外一种方法是让函数在失败时引发异常。用这种方法，代码更容易编写和维护，而且也执行得更好，因为通常不需要执行那些错误测试代码。实际上，仅当发生失败时也就是发生异常时才执行错误测试代码。

但令人遗憾的是，许多开发人员不习惯于在错误处理中使用异常。这有两方面的原因。第一个原因是多数开发人员不熟悉SEH。即使有一个程序员熟悉它，但其他程序员可能不熟悉它。如果一个程序员编写了一个引发异常的函数，但其他程序员并不编写SEH框架来捕获这个异常，那么进程就会被操作系统结束。

开发人员不使用SEH的第二个原因是它不能移植到其他操作系统。许多公司的产品要面向多种操作系统，因此希望有单一的源代码作为产品的基础，这是可以理解的。 SEH是专门针对Windows的技术。

本段讨论通过异常返回错误有关的内容。首先，让我们看一看 Windows Heap函数，例如HeapCreate、heapAlloc等。回顾第18章的内容，我们知道这些函数向开发人员提供一种选择。通常当某个堆（heap）函数失败，它会返回 NULL来指出失败。然而可以对这些堆函数传递HEAP_GENERATE_EXCEPTIONS标志。如果使用这个标志并且函数失败，函数不会返回NULL，而是由函数引发一个STATUS_NO_MEMORY软件异常，程序代码的其他部分可以用SEH框架来捕获这个异常。

如果想利用这个异常，可以编写你的try块，好像内存分配总能成功。如果内存分配失败，可以利用except块来处理这个异常，或通过匹配 try块与finally块，清除函数所做的事。这非常方便。

程序捕获软件异常采取的方法与捕获硬件异常完全相同。也就是说，前一章介绍的内容可以同样适用于软件异常。

本节重讨论如何让你自己的函数引发软件异常，作为指出失败的方法。实际上，可以用类似于微软实现堆函数的方法来实现你的函数：让函数的调用者传递一个标志，告诉函数如何指出失败。

引发一个软件异常很容易，只需要调用RaiseException函数：

```
VOID RaiseException(
   DWORD dwExceptionCode,
   DWORD dwExceptionFlags,
   DWORD nNumberOfArguments,
   CONST ULONG_PTR *pArguments);
```

第一个参数dwExceptionCode是标识所引发异常的值。 HeapAlloc函数对这个参数设定STATUS_NO_MEMORY。如果程序员要定义自己的异常标识符，应该遵循标准 Windows错误代码的格式，像WinError.h文件中定义的那样。参阅表24-1。

如果要建立你自己的异常代码，要填充DWORD的4个部分：

- 第31位和第30位包含严重性系数(severity)。
- 第29位是1（0表示微软建立的异常，如HeapAlloc的STATUS_NO_MEMORY）。
- 第28位是0。
- 第27位到16位是某个微软定义的设备代码。

China-pub.com
下载

• 第15到0位是一个任意值，用来标识引起异常的程序段。

RaiseException的第二个参数 dwExceptionFlags，必须是 0或EXCEPTION_NONCONTINUABLE。本质上，这个标志是用来规定异常过滤器返回 EXCEPTION_CONTINUE _EXECUTION来响应所引发的异常是否合法。如果没有向 RaiseException传递EXCEPTION_ NONCONTINUABLE参数值，则过滤器可以返回 EXCEPTION_CONTINUE_EXECUTION。正常情况下，这将导致线程重新执行引发软件异常的同一 CPU指令。但微软已做了一些动作，所以在调用RaiseException函数之后，执行会继续进行。

如果你向RaiseException传递了EXCEPTION_NONCONTINUABLE标志，你就是在告诉系统，你引发异常的类型是不能被继续执行的。这个标志在操作系统内部被用来传达致命（不可恢复）的错误信息。另外，当 HeapAlloc引发STATUS_NO_MEMORY软件异常时，它使用EXCEPTION_NONCONTINUABLE标志来告诉系统，这个异常不能被继续。意思就是没有办法强制分配内存并继续运行。

如果一个过滤器忽略EXCEPTION_NONCONTINUABLE并返回EXCEPTION_CONTINUE_EXECUTION，系统会引发新的异常：EXCEPTION_NONCONTINUABLE_EXCEPTION。

当程序在处理一个异常的时候，有可能又引发另一个异常。比如说，一个无效的内存存取有可能发生在一个 finally块、一个异常过滤器、或一个异常处理程序里。当发生这种情况时，系统压栈异常。回忆一下GetExceptionInformation函数。这个函数返回EXCEPTION_ POINTERS结构的地址。EXCEPTION_POINTERS的ExceptionRecord成员指向一个EXCEPTION_ RECORD结构，这个结构包含另一个 ExceptionRecord成员。这个成员是一个指向另外的 EXCEPTION_RECORD的指针，而这个结构包含有关以前引发异常的信息。

通常系统一次只处理一个异常，并且ExceptionRecord成员为NULL。然而如果处理一个异常的过程中又引发另一个异常，第一个EXCEPTION_RECORD结构包含有关最近引发异常的信息，并且这个EXCEPTION_RECORD结构的ExceptionRecord成员指向以前发生的异常的EXCEPTION_RECORD结构。如果增加的异常没有完全处理，可以继续搜索这个 EXCEPTION_RECORD结构的链表，来确定如何处理异常。

RaiseException的第三个参数nNumberOfArguments和第四个参数pArguments，用来传递有关所引发异常的附加信息。通常，不需要附加的参数，只需对 pArguments参数传递NULL，这种情况下， RaiseException函数忽略 nNumberOfArguments参数。如果需要传递附加参数，nNumberOfArguments参数必须规定由 pArguments参数所指向的ULONG_PTR数组中的元素数目。这个数目不能超过EXCEPTION_MAXIMUM_PARAMETERS，EXCEPTION_ MAXIMUM_PARAMETERS 在WinNT.h中定义成15。

在处理这个异常期间，可使异常过滤器参照 EXCEPTION_RECORD结构中的 NumberParameters和ExceptionInformation成员来检查nNumberOfArguments和pArguments参数中的信息。

你可能由于某种原因想在自己的程序中产生自己的软件异常。例如，你可能想向系统的事件日志发送通知消息。每当程序中的一个函数发现某种问题，你可以调用 RaiseException并让某些异常处理程序上溯调用树查看特定的异常，或者将异常写到日志里或弹出一个消息框。你还可能想建立软件异常来传达程序内部致使错误的信息。

China-pub.com
下载

第25章　未处理异常和C++异常

前一章讨论了当一个异常过滤器返回EXCEPTION_CONTINUE_SEARCH时会发生什么事情。返回EXCEPTION_CONTINUE_SEARCH 是告诉系统继续上溯调用树，去寻找另外的异常过滤器。但是当每个过滤器都返回EXCEPTION_CONTINUE_SEARCH时会出现什么情况呢？在这种情况下，就出现了所谓的“未处理异常”(Unhandled exception)。

在第6章里我们已经知道，每个线程开始执行，实际上是利用 Kernel32.dll中的一个函数来调用BaseProcessStart或BaseThreadStart。这两个函数实际是一样的，区别在于一个函数用于进程的主线程（Primary thread）：

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
   __try {
      ExitThread((pfnStartAddr)());
   }
   __except (UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here
}
```

另一个函数用于进程的所有辅助线程（Secondary thread）：

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
   __try {
      ExitThread((pfnStartAddr)(pvParam));
   }
   __except (UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here
}
```

注意这两个函数都包含一个SEH框架。每个函数都有一个try块，并从这个try块里调用主线程或辅助线程的进入点函数。所以，当线程引发一个异常，所有过滤器都返回 EXCEPTION_CONTINUE_SEARCH时，将会自动调用一个由系统提供的特殊过滤器函数： UnhandledExceptionFilter。

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

这个函数负责显示一个消息框，指出有一个进程的线程存在未处理的异常，并且能让用户结束或调试这个进程。在Windows 98中，这个消息框如图25-1所示。

在Windows 2000中，消息框如图25-2所示。

在Windows 2000下的消息框中，第一段文字指出发生了哪一种异常，并给出在进程地址空间中产生异常的指令的地址。在这里，消息框中指出发生了一个内存访问异常，系统报告了要访问的无效内存地址，指出由于试图读这个内存而引发了异常。 UnhandledExceptionFilter函数从这个异常生成的EXCEPTION_RECORD结构的ExceptionInformation成员中获取这些附加的信息。

China-pub.com
下载

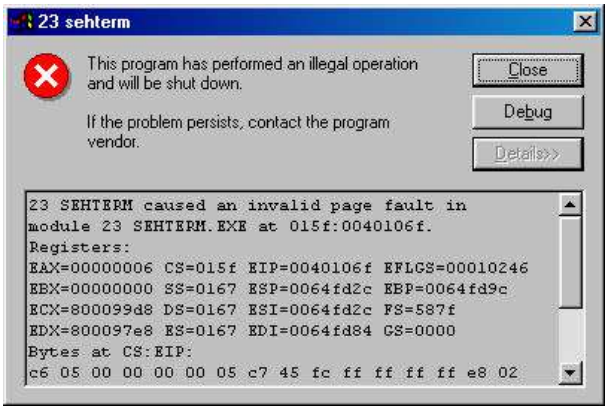


图25-1　Windows 98下显示进程的线程存在未处理的异常的消息框



图25-2　Windows 2000下的消息框

在消息框中的异常描述之后，又提供给用户两种选择。第一种选择是点击 OK按钮，这将导致UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。这又引起全局展开的发生，所有的finally块都要执行，然后BaseProcessStart或BaseThreadStart中的处理程序执行。这两个处理程序都叫ExitProcess，意思是退出进程，这就是程序结束的原因。注意进程的退出代码就是异常代码。还要注意是进程的线程废（ kill）了进程本身，而不是操作系统。这也意味着程序员可以控制这种行为并可以改变它。

第二种选择是点击 Cancel按钮。在这里程序员的梦想成真。当点击 Cancel按钮时，UnhandledExceptionFilter试图加载一个调试程序，并将这个调试程序挂接在进程上。通过将调试程序附在进程上，可以检查全局、局部和静态变量的状态，设置断点，检查调用树，重新启动进程，以及调试一个进程时可以做的任何事情。

这里真正的好处是，你可以在程序运行当中错误发生时就处理错误。在其他操作系统中，必须在调试程序调用程序时才能对其进行调试。在那些操作系统中，如果一个进程中发生了一个异常，必须结束这个进程，启动调试程序，调用程序，再使用调试程序。这样，你必须重新产生这个错误，才有可能去修正它。但谁能记住问题最初发生时的各种条件和各变量的值？按这种方式解决程序错误问题非常困难。将调试程序动态挂接在运行中的进程上，是 Windows最好的特性之一。

Windows 2000　本书着重讨论用户方式（user-mode）的程序开发。但对于运行在内核方式（kernel-mode）的线程引发的未处理异常会造成什么情况，读者也许会感兴趣。内核方式中的异常同用户方式中的异常是按同样方式处理的。如果一个低级虚拟内存函数产生一个异常，系统查找是否有内核方式异常过滤器准备处理这个异常。如果系统找不到一个异常过滤器来处理这个异常，则异常就是未处理的。对于内核方式的异

China-pub.com 下载

常，未处理异常是在操作系统中或（更可能）在设备驱动程序中，而不是在应用程序中。这样一个未处理异常表示一个严重的程序错误（bug）！

如果一个未处理异常发生在内核方式，让系统继续运行是不安全的。所以系统在这种情况下不去调用UnhandledExceptionFilter函数，而是显示所谓的蓝屏死机（Blue Screen of Death）。显示画屏切换到只包含文本的蓝屏视频方式，并且计算机被停机（balt）。显示的文本告诉是哪个设备驱动程序被加载，并且该模块中包含有引发未处理异常的代码。用户应该记下这些信息并送交微软或设备驱动程序的厂商，以便修复这个错误。因为计算机被挂起，要想再做其他事情就必须重新启动计算机，所有未保存的工作都丢失了。

25.1 即时调试

随时将调试程序连接到任何进程的能力称为即时调试（Just-in-time Debugging）。这里我们对它如何工作稍加说明：当程序员点击Cancel按钮，就是告诉UnhandledExceptionFilter函数对进程进行调试。

在内部，UnhandledExceptionFilter调用调试程序，这需要查看下面的注册表子关键字：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

在这个子关键字里，有一个名为Debugger的数值，在安装Visual Studio时被设置成下面的值：

```
"C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe"
   -p %ld -e %ld
```

Windows 98 在Windows 98中，这些值不是存放在注册表中，而是存放在Win.ini文件中。

这一行代码是告诉系统要将哪一个程序（这里是MSDev. exe）作为调试程序运行。当然也可以选择其他调试程序。UnhandledExceptionFilter还在这个命令行中向调试程序传递两个参数。第一个参数是被调试进程的ID。第二个参数规定一个可继承的手工复位事件，这个事件是由UnhandledExceptionFilter按无信号状态建立的。厂商必须实现他们的调试程序，这样才能认识指定进程ID和事件句柄的-p和-e选项。

在进程ID和事件句柄都合并到这个串中之后，UnhandledExceptionFilter通过调用CreateProcess来执行调试程序。这时，调试程序进程开始运行并检查它的命令行参数。如果存在-p选项，调试程序取得进程ID，并通过调用DebugActiveProcess将自身挂接在该进程上。

```
BOOL DebugActiveProcess(DWORD dwProcessID);
```

一旦调试程序完成自身的挂接，操作系统将被调试者（debuggee）的状态通报给调试程序。例如，系统将告诉调试程序，在被调试的进程中有多少线程？哪些DDL加载到被调试进程的地址空间中？调试程序需要花时间来积累这些数据，以准备调试进程。在这些准备工作进行的时候，UnhandledExceptionFilter中的线程必须等待。为此，这要调用WaitForSingleObject函数并传递已经建立的手工复位事件的句柄作为参数。这个事件是按无信号状态建立起来的，所以被调试进程的线程要立即被挂起以等待事件。

在调试程序完全初始化之后，它要再检查它的命令行，找-e选项。如果该选项存在，调试程序取得相应的事件句柄并调用SetEvent。调试程序可以直接使用事件的句柄值，因为事件句柄具有创建的可继承性，并且被调试进程对UnhandledExceptionFilter函数的调用也使调试程序

China-pub.com
下载

进程成为一个子进程。

设定这个事件将唤醒被调试进程的线程。被唤醒的线程将有关未处理异常的信息传递给调试程序。调试程序接收这些通知并加载相应的源代码文件，再将自身放在引发异常的指令位置上。

还有，不必在调试进程之前等待异常的出现。可以随时将一个调试程序连接在任何进程上，只需运行“ MSDEV -p PID ”，其中PID是要调试的进程的ID。实际上，利用 Windows 2000 Task Manager，做这些事很容易。当观察Process标记栏时，可以选择一个进程，点击鼠标右键，并选择Debug菜单选项。这将引起 Task Manager去查看前面讨论过的注册表子关键字，调用CreateProcess，并传递所选定的进程的ID作为参数。在这里， Task Manager为事件句柄传送0值。

25.2　关闭异常消息框

有时候，在异常发生时，你可能不想在屏幕上显示异常消息框。例如，你可能不想让这些消息框出现在你产品的发售版本中。如果出现了消息框，很容易导致最终用户意外地启动调试程序来调试你的程序。最终用户只需点击一下消息框中的 Cancel按钮，就进入了不熟悉的、令人恐惶的区域 —— 调试程序。可以使用几种不同的方法来防止这种消息框的出现。

25.2.1　强制进程终止运行

为防止UnhandledExceptionFilter显示异常消息框，可以调用下面的 SetErrorModel函数，并向它传递一个SEM_NOGPFAULTERRORBOX标识符：

```
UINT SetErrorMode(UINT fuErrorMode);
```

然后，当调用UnhandledExceptionFilter函数来处理异常时，看到已经设置了这个标志，就会立即返回EXCEPTION_EXECUTE_HANDLER。这将导致全局展开并执行BaseProcessStart或BaseThreadStart中的处理程序。该处理程序结束进程。

笔者并不喜欢这种方式。在这种方式下，用户完全没有得到警告，程序只是自行消失。

25.2.2　包装一个线程函数

使用另外一种办法也可以避免出现这个消息框，就是针对主线程进入点函数（ main、wmain、WinMain或wWinMain）的整个内容安排一个try-except块。保证异常过滤器的结果值总是EXCEPTION_EXECUTE_HANDLER，这样就保证异常得到处理，防止了系统再调用UnhandledExceptionFilter函数。

在你的异常处理程序中，你可以显示一个对话框，在上面显示一些有关异常的诊断信息。用户可以记录下这些信息，并通报给你公司的客户服务部门，以便能够找到程序的问题根源。你应该建立这个对话框，这样用户只能结束程序而不能调用调试程序。

这种方法的缺点是它只能捕捉进程的主线程中发生的异常。如果其他线程在运行，并且其中有一个线程发生了一个未处理异常，系统就要调用内部的 UnhandledExceptionFilter函数。为了改正这一点，需要在所有的辅助线程进入点函数中包含 try-except块。

25.2.3　包装所有的线程函数

Windows还提供另外一个函数， SetUnhandledExceptionFilter，利用它可以按SEH格式包装所有的线程函数：

China-pub.com 下载

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
   PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

在进程调用这些函数之后，进程的任何线程中若发生一个未处理的异常，就会导致调用程序自己的异常过滤器。需要将这个过滤器的地址作为参数传递给 SetUnhandledExceptionFilter。过滤器函数原型必须是下面的样子：

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

你可能会注意到这个函数同UnhandledExceptionFilter函数的形式是一样的。程序员可以在自己的异常过滤器中执行任何想做的处理，但要返回三个 EXCEPTION_*标识符中的一个。表25-1给出了当返回各标识符时所发生的事。

表25-1 返回标识符时的情况

标 识 符	出现的情况
EXCEPTION_EXECUTE_HANDLER	进程只是结束，因系统在其异常处理程序块中没有执行任何动作
EXCEPTION_CONTINUE_EXECUTION	从引起异常的指令处继续执行。可以参照PEXCEPTION_POINTERS参数修改异常信息
EXCEPTION_CONTINUE_SEARCH	执行正规的Windows UnhandledExceptionFilter函数

为了使UnhandledExceptionFilter函数再成为默认的过滤器，可以调用 SetUnhandled ExceptionFilter并传递 NULL给它。而且，每当设置一个新的未处理的异常过滤器时，SetUnhandled ExceptionFilter就返回以前安装的异常过滤器的地址。如果 Unhandled Exception Filter是当前所安装的过滤器，则这个返回的地址就是 NULL。如果你自己的过滤器要返回 EXCEPTION_CONTINUE_SEARCH，你就应该调用以前安装的过滤器，其地址通过 SetUnhandledExceptionFilter函数返回。

25.2.4 自动调用调试程序

现在再介绍关闭UnhandledExceptionFilter消息框的最后一种方法。在前面提到的同一个注册表子关键字里，还有另外一个数据值，名为 Auto。这个值用来规定UnhandledExceptionFilter是应该显示消息框，还是仅启动调试程序。如果 Auto设置成1，UnhandledExceptionFilter就不显示消息框向用户报告异常，而是立即调用调试程序。如果 Auto子关键设置成 0，UnhandledExceptionFilter就显示异常消息框，并按前面描述的那样操作。

25.3 程序员自己调用UnhandledExceptionFilter

UnhandledExceptionFilter函数是一个公开的、文档完备的 Windows函数，程序员可以直接在自己的代码中调用这个函数。这里是使用这个函数的一个例子：

```
void Funcadelic() {
   __try {
      :
      :
   }
   __except (ExpFltr(GetExceptionInformation())) {
      :
      :
   }
```

China-pub.com
下载


```
}

LONG ExpFltr(PEXCEPTION_POINTERS pEP) {
   DWORD dwExceptionCode = pEP->ExceptionRecord.ExceptionCode;

   if (dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
      // Do some work here....
      return(EXCEPTION_CONTINUE_EXECUTION);
   }

   return(UnhandledExceptionFilter(pEP));
}
```

在Funcadelic函数中，try块中的一个异常导致ExpFltr函数被调用。GetExceptionInformation的返回值作为参数传递给ExpFltr函数。在异常过滤器内，要确定异常代码并与EXCEPTION_ACCESS_VIOLATION相比较。如果发生一个存取违规，异常过滤器改正这个问题，并从过滤器返回EXCEPTION_CONTINUE_EXECUTION。这个返回值导致系统从最初引起异常的指令继续执行。

如果发生了其他异常，ExpFltr调用UnhandledExceptionFilter，将EXCEPTION_POINTERS结构的地址传递给它作为参数。UnhandledExceptionFilter显示消息框，可使程序员结束进程或开始调试进程。UnhandledExceptionFilter的返回值再由ExpFltr返回。

25.4　UnhandledExceptionFilter函数的一些细节

当笔者最初接触异常处理时，就认为如果能详细了解系统的UnhandledExceptionFilter函数所做的事情，就能得到许多有用的信息。为此，笔者仔细研究了这个函数。下面所列的步骤详细描述了UnhandledExceptionFilter函数的内部执行情况：

1) 如果发生一个存取违规并且是由于试图写内存（相对于读内存）引起的，系统要查看你是不是要修改一个.exe模块或DLL模块中的资源。默认时，资源是（而且应该是）只读的。试图修改资源会引起存取异常。然而16位Windows允许修改资源，从兼容性考虑，32位和64位Windows也应该允许修改资源。所以当想要修改资源的时候，UnhandledExceptionFilter调用VirtualProtect，将资源页上的保护改成PAGE_READWRITE，并返回EXCEPTION_CONTINUE_EXECUTION。

2) 如果你已经调用SetUnhandledExceptionFilter指定了你自己的过滤器，UnhandledExceptionFilter就调用你自己的过滤器函数。如果你自己的过滤器函数返回EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_EXECUTION，UnhandledExceptionFilter就将这个值返回给系统。如果你没有设置你自己的未处理异常过滤器，或者你的未处理异常过滤器返回EXCEPTION_CONTINUE_SEARCH，转到第3步继续处理。

> Windows 98　Windows 98有这样的错误（bug）：如果进程不在调试中，系统只调用程序自己的未处理异常过滤器函数。这样我就不可能调试本章后面介绍的spreadsheet示例程序。

3) 如果你的进程是在调试程序下运行的，就返回EXCEPTION_CONTINUE_SEARCH。你可能会对此感到不解，因系统已经为线程执行了最高层的try或except框架，再往高层已经没有其他的异常过滤器可搜索。当系统看到最高层过滤器返回EXCEPTION_CONTINUE_SERCH时，系统知道要同调试程序联系并告诉调试程序，被调试程序只是有一个未处理异常。作为回答，

China-pub.com 下载

调试程序显示一个消息框并允许你调试进程（注意，IsDebuggerPresent函数用来确定一个进程是否正在被调试）。

4) 如果进程中的一个线程以SEM_NOGPFAULTERRORBOX标志为参数调用SetErrorMode，UnhandledExceptionFilter就返回EXCEPTION_EXECUTE_HANDLER。

5) 如果进程在一个作业（job）里并且作业的限制信息设定了JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION标志，则UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。

Windows 98 Windows 98不支持作业，所以这一步略去。

6) UnhandledExceptionFilter查阅注册表并取出Auto值。如果这个值是1，跳到第7步。如果这个值是0，向用户显示一个消息框。这个消息框指出引发了什么异常。如果注册表子关键字也包含这个Debugger值，消息框有OK按钮和Cancel按钮。如果注册表子关键字没有Debugger值，消息框只包含OK按钮。如果用户点击OK按钮，UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。如果Cancel按钮可用并且用户按了这个按钮，转到第7步继续处理。

Windows 98 在Windows 98中，这些值不是保存在注册表里，而是保存在Win.ini文件里。

7) UnhandledExceptionFilter现在要产生调试程序。它首先调用CreateEvent建立一个无信号的、手工复位的事件。这个事件的句柄是可继承的。然后它从注册表中取出Debugger值，调用sprintf把它粘贴到进程ID（通过调用GetCurrentProcessID函数得到）和事件句柄里。STARTUPINFO的lpDesktop成员也设置成“Winsta0\\Default”，这样调试程序就出现在交互式桌面上。调用CreateProcess，其中fInheritHandles参数设置成TRUE，这个函数再调用调试程序进程并允许它继承事件对象的句柄。UnhandledExceptionFilter通过以事件句柄为参数调用WaitForSingleObjectEx，等待调试程序初始化。注意这里是用WaitForSingleObjectEx函数而不是WaitForSingleObject函数，所以线程是在可报警状态等待。这样就可以处理线程的任何排队的同步过程调用（APC）。

8) 当调试程序完成初始化时，就设置事件句柄，这将唤醒UnhandledExceptionFilter中的线程。现在进程就在调试程序之下运行，UnhandledExceptionFilter返回EXCEPTION_CONTINUE_SEARCH。注意这就是第3步所发生的事。

25.5 异常与调试程序

Microsoft Visual C++调试程序支持调试异常。当一个进程的线程引起一个异常，操作系统立即通知调试程序（如果挂接了一个调试程序）。这个通知被称作最初机会通知（first-chance notificontion）。正常情况下，调试程序要响应最初机会通知，告诉线程去搜索异常过滤器。如果所有的异常过滤器都返回EXCEPTION_CONTINUE_SEARCH，操作系统再通知调试程序，称为最后机会通知(last-chance notification)。使用这两种通知是为了使软件开发人员对调试异常有更多的控制能力。

使用调试程序的Exceptions对话框（见图25-3）来告诉调试程序如何响应最初机会异常通知。

我们可以看到，对话框由系统定义的全部异常所组成。对话框中显示了异常的32位代码，后面跟着文本描述和调试程序的动作。在上面的窗口中，选择了存取违规(Access Violation)异

China-pub.com
下载

常，将它的动作改变成 Stop always。这样，当被调试程序中的一个线程引起一个存取违规时，调试程序会接到其最初机会通知，并显示类似图 25-4所示的消息框。

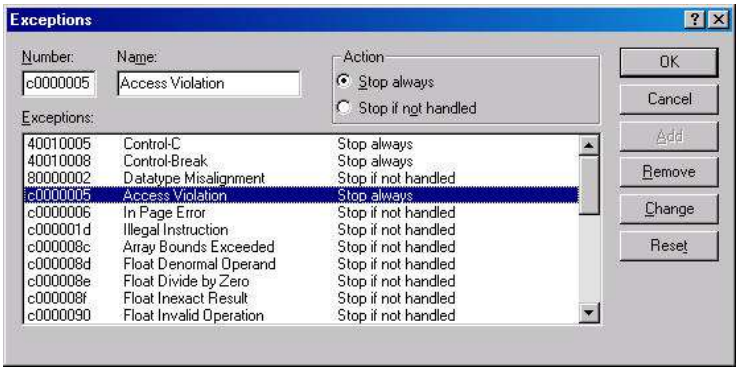


图25-3　Exceptions 对话框



图25-4　“ 最初机会通知 ”消息框

在这一步，线程还没有机会去搜索异常过滤器。现在我们可以在代码中设置断点，检查变量，或检查线程的调用栈。现在还没有异常过滤器执行，异常只是发生而已。如果我们现在使用调试程序单步执行代码，会出现图 25-5所示的提示消息框。



图25-5　提示消息框

点击Cancel按钮，返回到调试程序。点击NO按钮是告诉被调试的线程去重试失败的CPU指令。对大多数异常，重试失败的指令只会再一次引起异常，没有什么用处。但对于一个由函数 RaiseException引起的异常，重试是告诉线程继续执行，就好像异常从未发生。按这种方式继续执行，对于调试 C++程序特别有用：这就好像是一个从未执行过的C++ throw语句。C++异常处理在本章后面讨论。

点击 Yes按钮会使被调试程序的线程去搜索异常过滤器。如果找到一个返回EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_EXECUTION的异常过滤器，则一切都好，线程可以继续执行其代码。如果所有的过滤器都返回 EXCEPTION_C ONTINUE_SEARCH，调试程序收到一个最后机会通知，显示类似图 25-6所示的消息框。



图25-6　调试程序接收到“ 最后机会通知 ”时显示的消息框

China-pub.com 下载

在这一步，必须要调试这个程序或结束这个程序。

我们讨论了当调试程序的动作设置成 Stop Always时所发生的各种可能情况。但对大多数异常，Stop If Not Handled是默认的动作。如果被调试程序中的一个线程引起一个异常，调试程序会收到最初机会通知。如果调试程序动作设置成 Stop If Not Handled，调试程序只是在调试程序的Output窗口显示一个字符串表示它收到了这个通知（见图 25-7）。

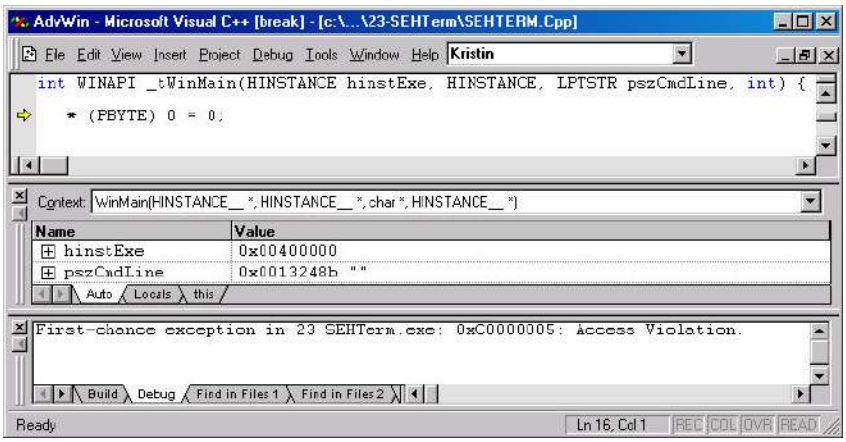


图25-7 调试程序的Output窗口

如果将对存取违规的动作设置成 Stop If Not Handled，调试程序会使线程去搜索异常过滤器。仅当异常没有被处理时，调试程序才会显示图 25-8所示的消息框。



图25-8 异常没有被处理时显示的消息框

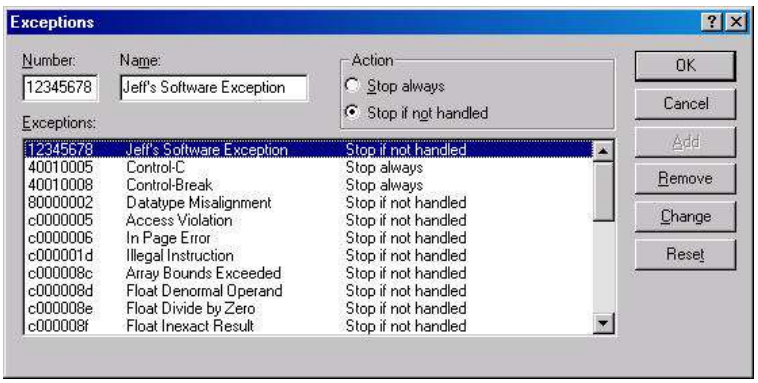


图25-9 Exceptions 窗口

注意 要记住的重要一点是最初机会通知并不指出程序中的问题或错误。实际上，这个通知只是在进程被调试时出现。调试程序只是报告引发了一个异常，但调试程序如

China-pub.com
下载

果不显示消息框，处理异常的过滤器和程序会继续运行。一个最后机会通知意味着程序中有问题或错误，必须要进行修改。

在结束这一节之前，需要再说一说调试程序的 Exception对话框。这个对话框完全支持程序员自己定义的任何软件异常。需要做的只是要输入你的软件异常代码编号，并保证唯一性，输入异常的字符串名，还有选择的动作。然后点击 Add按钮，将所定义的异常添加到异常表中。图25-9所示的窗口说明了如何使调试程序知道自己定义的软件异常。

Spreadsheet示例程序

本节后面清单25-1所列的Spreadsheet示例程序说明了如何使用结构化异常处理向一个预留的地址空间范围稀疏地提交存储区。程序的源代码和资源文件在本书所附光盘的“ 25-Spreadsheet ”目录下。当执行Spreadsheet程序时，会出现图25-10所示的对话框。

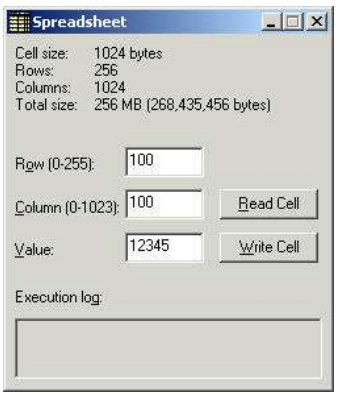


图25-10　Spreadsheet 对话框

在内部，程序为二维的空白表格保留一个范围。空白表格包含256行乘1024列，每个单元长度是1024字节。如果程序要提交内存覆盖整个空白表格，则需要 268 435 456字节，即256MB。为了保留宝贵的内存空间，程序保留一个 256MB的地址空间范围而不提交任何内存来覆盖这个范围。

假定用户试图在第100行第100列的单元中存放数值12345（见图25-10）。当用户点击Write Cell按钮时，程序代码试图对空白表格的这个位置写数据。当然，这个试图写操作要引起一个存取违规。但由于在程序中使用了 SEH，异常过滤器检测到这个试图写操作，在对话框的底部显示“ Violation: Attempting to Write ”消息，对该单元提存储区，并使CPU重新执行引起异常的指令。因存储区已经提交，数值被写入空白表格的单元（见图25-11）。

我们再做另一个实验。试图读第 5行第20列单元中的值。当你试图从这个单元读数据时又引起一个存取违规。对一个试图读操作，异常过滤器不提交存储区，但在对话框中显示“ Violation: Attempting to Read ”消息。程序通过删除对话框中 Value字段中的数值，很自然地从失败的读操作中恢复，见图25-12。

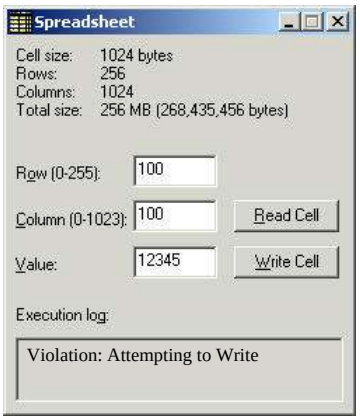


图25-11　Spreadsheet 对话框的下部显示信息

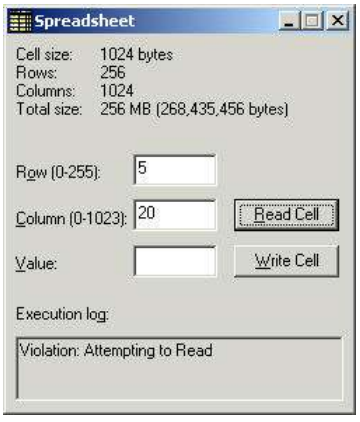


图25-12　删除Value 字段中的数值时的对话框

China-pub.com 下载

我们的第3个实验，是读第100行第100列单元的数值。因已经对这个单元提交了存储区，所以没有异常发生，也没有异常过滤器执行（改善了性能）。对话框如图25-13所示。

我们再看第4个实验：试图向第100行第101列单元写数值54321。当你试图这么做时，没有发生异常，这个单元在单元（100，100）相同的内存页上。我们可以看到对话框底部显示消息“No Violation Raised”（见图25-14）。

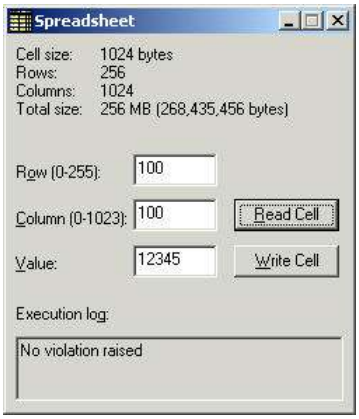


图25-13 第3个实验显示的对话框

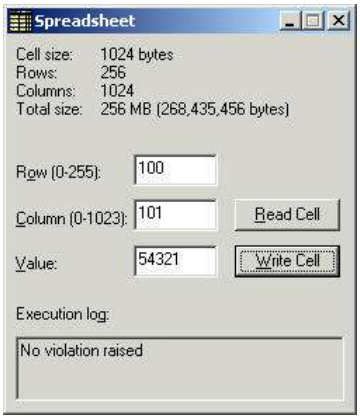


图25-14 第4个实验显示的对话框

笔者在自己的项目中倾向于使用虚拟内存和 SEH。后来决定建立一个模板化的 C++类，CVMArray，它封装了所有困难的工作。这个C++类的源代码在VMArray.h文件中（清单25-1所示的Spreadsheet示例程序的一部分）。可以按两种方式使用CVMArray类。第一，可以建立一个这个类的实例，将数组中元素的最大值传递给构造函数。这个类自动启动一个全进程范围内未处理异常过滤器，每当任何线程中的代码存取虚拟内存数组中的内存地址时，未处理异常过滤器调用VirtualAlloc为新元素提交存储区，并返回EXCEPTION_CONTINUE_EXECUTION。按这种方式使用CVMArray类，可以使你只用几行代码就实现稀疏存储区，并且也不必在源代码中到处使用SEH框架。这种方法的唯一不足是当由于某种原因存储区不能提交时，程序不能以自然的方式恢复。

使用CVMArray类的第二种方法是从中派生出自己的 C++类。使用派生类可以得到基类的优点，但同时也可以添加自己所需要的特性。例如，现在通过过载（ overload）虚拟函数OnAccessViolation更自然地解决了存储区不够的问题。 Spreadsheet示例程序展示了如何在CVMArray的派生类中添加这些特性的方法。

清单25-1 Spreadsheet示例程序

Spreadsheet.cpp

```
/******************************************************************************
Module:  Spreadsheet.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
```

China-pub.com
下载


```
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "VMArray.h"


//////////////////////////////////////////////////////////////////////////////


HWND g_hwnd;   // Global window handle used for SEH reporting

const int g_nNumRows = 256;
const int g_nNumCols = 1024;
// Declare the contents of a single cell in the spreadsheet
typedef struct {
   DWORD dwValue;
   BYTE  bDummy[1020];
} CELL, *PCELL;


// Declare the data type for an entire spreadsheet
typedef CELL SPREADSHEET[g_nNumRows][g_nNumCols];
typedef SPREADSHEET *PSPREADSHEET;


//////////////////////////////////////////////////////////////////////////////


// A spreadsheet is a 2-dimensional array of CELLs
class CVMSpreadsheet : public CVMArray<CELL> {
public:
   CVMSpreadsheet() : CVMArray<CELL>(g_nNumRows * g_nNumCols) {}

private:
   LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);
};


//////////////////////////////////////////////////////////////////////////////


LONG CVMSpreadsheet::OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
   PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

   TCHAR sz[200];
   wsprintf(sz, TEXT("Violation: Attempting to %s"),
      fAttemptedRead ? TEXT("Read") : TEXT("Write"));
   SetDlgItemText(g_hwnd, IDC_LOG, sz);

   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;
   if (!fAttemptedRead) {

      // Return whatever the base class says to do
      lDisposition = CVMArray<CELL>::OnAccessViolation(pvAddrTouched,
```

China-pub.com 下载

```
            fAttemptedRead, pep, fRetryUntilSuccessful);
      }
      return(lDisposition);
   }

   ////////////////////////////////////////////////////////////////////////////


   // This is the global CVMSpreadsheet object
   static CVMSpreadsheet g_ssObject;

   // Create a global pointer that points to the entire spreadsheet region
   SPREADSHEET& g_ss = * (PSPREADSHEET) (PCELL) g_ssObject;


   ////////////////////////////////////////////////////////////////////////////


   BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

      chSETDLGICONS(hwnd, IDI_SPREADSHEET);

      g_hwnd = hwnd; // Save for SEH reporting

      // Put default values in the dialog box controls
      Edit_LimitText(GetDlgItem(hwnd, IDC_ROW),    3);
      Edit_LimitText(GetDlgItem(hwnd, IDC_COLUMN), 4);
      Edit_LimitText(GetDlgItem(hwnd, IDC_VALUE),  7);
      SetDlgItemInt(hwnd, IDC_ROW,    100,   FALSE);
      SetDlgItemInt(hwnd, IDC_COLUMN, 100,   FALSE);
      SetDlgItemInt(hwnd, IDC_VALUE,  12345, FALSE);
      return(TRUE);
   }


   ////////////////////////////////////////////////////////////////////////////


   void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

      int nRow, nCol;

      switch (id) {
         case IDCANCEL:
            EndDialog(hwnd, id);
            break;
         case IDC_ROW:
            // User modified the row, update the UI
            nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
               chINRANGE(0, nRow, g_nNumRows - 1));
            EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
               chINRANGE(0, nRow, g_nNumRows - 1));
            break;
```

China-pub.com
下载


```
      case IDC_COLUMN:
         // User modified the column, update the UI
         nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
         EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
            chINRANGE(0, nCol, g_nNumCols - 1));
         EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
            chINRANGE(0, nCol, g_nNumCols - 1));
         break;

      case IDC_READCELL:
         // Try to read a value from the user's selected cell
         SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
         nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
         nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
         __try {
            SetDlgItemInt(hwnd, IDC_VALUE, g_ss[nRow][nCol].dwValue, FALSE);
         }
         __except (
            g_ssObject.ExceptionFilter(GetExceptionInformation(), FALSE)) {

            // The cell is not backed by storage, the cell contains nothing.
            SetDlgItemText(hwnd, IDC_VALUE, TEXT(""));
         }
         break;

      case IDC_WRITECELL:
         // Try to read a value from the user's selected cell
         SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
         nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
         nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);

         // If the cell is not backed by storage, an access violation is
         // raised causing storage to automatically be committed.
         g_ss[nRow][nCol].dwValue =
            GetDlgItemInt(hwnd, IDC_VALUE, NULL, FALSE);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SPREADSHEET), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

Spreadsheet.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
```

China-pub.com
下载


```
    "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SPREADSHEET DIALOG DISCARDABLE  18, 18, 164, 165
STYLE DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Spreadsheet"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Cell size:\nRows:\nColumns:\nTotal size:",IDC_STATIC,4,
                    4,36,36
    LTEXT           "1024 bytes\n256\n1024\n256 MB (268,435,456 bytes)",
                    IDC_STATIC,44,4,104,36
    LTEXT           "R&ow (0-255):",IDC_STATIC,4,56,42,8
    EDITTEXT        IDC_ROW,60,52,40,14,ES_AUTOHSCROLL | ES_NUMBER
    LTEXT           "&Column (0-1023):",IDC_STATIC,4,76,54,8
    EDITTEXT        IDC_COLUMN,60,72,40,14,ES_AUTOHSCROLL | ES_NUMBER
    PUSHBUTTON      "&Read Cell",IDC_READCELL,108,72,50,14
    LTEXT           "&Value:",IDC_STATIC,4,96,21,8
    EDITTEXT        IDC_VALUE,60,92,40,14,ES_AUTOHSCROLL | ES_NUMBER
    PUSHBUTTON      "&Write Cell",IDC_WRITECELL,108,92,50,14
    LTEXT           "Execution lo&g:",IDC_STATIC,4,118,48,8
    EDITTEXT        IDC_LOG,4,132,156,28,ES_MULTILINE | ES_AUTOHSCROLL |
                    ES_READONLY
END


////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SPREADSHEET         ICON    DISCARDABLE     "Spreadsheet.Ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
#endif    // not APSTUDIO_INVOKED
```

VMArray.h

```
/******************************************************************************
Module:  VMArray.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once


////////////////////////////////////////////////////////////////////////////////


// NOTE: This C++ class is not thread safe. You cannot have multiple threads
// creating and destroying objects of this class at the same time.

// However, once created, multiple threads can access different CVMArray
// objects simultaneously and you can have multiple threads accessing a single
// CVMArray object if you manually synchronize access to the object yourself.


////////////////////////////////////////////////////////////////////////////////


template <class TYPE>
class CVMArray {
public:
   // Reserves sparse array of elements
   CVMArray(DWORD dwReserveElements);

   // Frees sparse array of elements
   virtual ~CVMArray();

   // Allows accessing an element in the array
   operator TYPE*()             { return(m_pArray); }
   operator const TYPE*() const { return(m_pArray); }

   // Can be called for fine-tuned handling if commit fails
   LONG ExceptionFilter(PEXCEPTION_POINTERS pep,
      BOOL fRetryUntilSuccessful = FALSE);

protected:
   // Override this to fine-tune handling of access violation
   virtual LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);

private:
   static CVMArray* sm_pHead;    // Address of first object
   CVMArray* m_pNext;            // Address of next  object

   TYPE* m_pArray;               // Pointer to reserved region array
   DWORD m_cbReserve;            // Size of reserved region array (in bytes)
```

China-pub.com
下载


```
private:
   // Address of previous unhandled exception filter
   static PTOP_LEVEL_EXCEPTION_FILTER sm_pfnUnhandledExceptionFilterPrev;

   // Our global unhandled exception filter for instances of this class
   static LONG WINAPI UnhandledExceptionFilter(PEXCEPTION_POINTERS pep);
};


//////////////////////////////////////////////////////////////////////////////


// The head of the linked-list of objects
template <class TYPE>
CVMArray<TYPE>* CVMArray<TYPE>::sm_pHead = NULL;

// Address of previous unhandled exception filter
template <class TYPE>
PTOP_LEVEL_EXCEPTION_FILTER CVMArray<TYPE>::sm_pfnUnhandledExceptionFilterPrev;


//////////////////////////////////////////////////////////////////////////////


template <class TYPE>
CVMArray<TYPE>::CVMArray(DWORD dwReserveElements) {

   if (sm_pHead == NULL) {
      // Install our global unhandled exception filter when
      // creating the first instance of the class.
      sm_pfnUnhandledExceptionFilterPrev =
         SetUnhandledExceptionFilter(UnhandledExceptionFilter);
   }
    m_pNext = sm_pHead;  // The next node was at the head
    sm_pHead = this;     // This node is now at the head

    m_cbReserve = sizeof(TYPE) * dwReserveElements;

    // Reserve a region for the entire array
    m_pArray = (TYPE*) VirtualAlloc(NULL, m_cbReserve,
       MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
    chASSERT(m_pArray != NULL);
}


//////////////////////////////////////////////////////////////////////////////


template <class TYPE>
CVMArray<TYPE>::~CVMArray() {

   // Free the array's region (decommitting all storage within it)
   VirtualFree(m_pArray, 0, MEM_RELEASE);

   // Remove this object from the linked list
```

China-pub.com 下载

```
   CVMArray* p = sm_pHead;
   if (p == this) {     // Removing the head node
      sm_pHead = p->m_pNext;
   } else {

      BOOL fFound = FALSE;

      // Walk list from head and fix pointers
      for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
         if (p->m_pNext == this) {
            // Make the node that points to us point to the next node
            p->m_pNext = p->m_pNext->m_pNext;
            break;
         }
      }
      chASSERT(fFound);
   }
}


//////////////////////////////////////////////////////////////////////////////
// Default handling of access violations attempts to commit storage
template <class TYPE>
LONG CVMArray<TYPE>::OnAccessViolation(PVOID pvAddrTouched,
   BOOL fAttemptedRead, PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

   BOOL fCommittedStorage = FALSE;  // Assume committing storage fails

   do {
      // Attempt to commit storage
      fCommittedStorage = (NULL != VirtualAlloc(pvAddrTouched,
         sizeof(TYPE), MEM_COMMIT, PAGE_READWRITE));

      // If storage could not be committed and we're supposed to keep trying
      // until we succeed, prompt user to free storage
      if (!fCommittedStorage && fRetryUntilSuccessful) {
         MessageBox(NULL,
            TEXT("Please close some other applications and Press OK."),
            TEXT("Insufficient Memory Available"), MB_ICONWARNING | MB_OK);
      }
   } while (!fCommittedStorage && fRetryUntilSuccessful);

   // If storage committed, try again. If not, execute the handler
   return(fCommittedStorage
      ? EXCEPTION_CONTINUE_EXECUTION : EXCEPTION_EXECUTE_HANDLER);
}

//////////////////////////////////////////////////////////////////////////////

// The filter associated with a single CVMArray object
template <class TYPE>
LONG CVMArray<TYPE>::ExceptionFilter(PEXCEPTION_POINTERS pep,
   BOOL fRetryUntilSuccessful) {

   // Default to trying another filter (safest thing to do)
```

China-pub.com
下载


```
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // We only fix access violations
   if (pep->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
      return(lDisposition);

   // Get address of attempted access and get attempted read or write
   PVOID pvAddrTouched = (PVOID) pep->ExceptionRecord->ExceptionInformation[1];
   BOOL fAttemptedRead = (pep->ExceptionRecord->ExceptionInformation[0] == 0);
   // Is attempted access within this VMArray's reserved region?
   if ((m_pArray <= pvAddrTouched) &&
       (pvAddrTouched < ((PBYTE) m_pArray + m_cbReserve))) {

      // Access is in this array, try to fix the problem
      lDisposition = OnAccessViolation(pvAddrTouched, fAttemptedRead,
         pep, fRetryUntilSuccessful);
   }

   return(lDisposition);
}

//////////////////////////////////////////////////////////////////////////////

// The filter associated with all CVMArray objects
template <class TYPE>
LONG WINAPI CVMArray<TYPE>::UnhandledExceptionFilter(PEXCEPTION_POINTERS pep) {

   // Default to trying another filter (safest thing to do)
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // We only fix access violations
   if (pep->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

      // Walk all the nodes in the linked-list
      for (CVMArray* p = sm_pHead; p != NULL; p = p->m_pNext) {

         // Ask this node if it can fix the problem.
         // NOTE: The problem MUST be fixed or the process will be terminated!
         lDisposition = p->ExceptionFilter(pep, TRUE);

         // If we found the node and it fixed the problem, stop the loop
         if (lDisposition != EXCEPTION_CONTINUE_SEARCH)
            break;
      }
   }

   // If no node fixed the problem, try the previous exception filter
   if (lDisposition == EXCEPTION_CONTINUE_SEARCH)
      lDisposition = sm_pfnUnhandledExceptionFilterPrev(pep);

   return(lDisposition);
}

//////////////////////////////// End of File /////////////////////////////////
```

China-pub.com 下载

25.6 C++异常与结构性异常的对比

笔者常碰到开发人员询问，在开发程序时应该使用结构化异常处理，还是使用 C++异常处理？本节对此作出回答。

首先要提醒读者，SEH是可用于任何编程语言的操作系统设施，而异常处理只能用于编写C++代码。如果你在编写C++程序，你应该使用C++异常处理而不是结构化异常处理。理由是C++异常处理是语言的一部分，编译器知道 C++类对象是什么。也就是说编译器能够自动生成代码来调用C++对象析构函数，保证对象的清除。

但是也应该知道，Microsoft Visual C++编译器已经利用操作系统的结构化异常处理实现了C++异常处理。所以当你建立一个 C++ try块时，编译器就生成一个 SEH__try块。一个C++ catch测试变成一个SEH异常过滤器，并且catch中的代码变成SEH__except块中的代码。实际上，当你写一条C++ throw语句时，编译器就生成一个对 Windows的RaiseException函数的调用。用于throw语句的变量传递给 RaiseException作为附加的参数。

下面的代码段可以使上面的叙述更清楚一些。左边的函数使用 C++异常处理，右边的函数说明了C++编译器如何生成等价的结构化异常处理。

```
void ChunkyFunky() {
   try {
      // Try body
      .
      .
      .
      throw 5;

   }
   catch (int x) {

      // Catch body
      .
      .
      .
   }
   .
   .
   .
}
```

```
void ChunkyFunky() {
   __try {
      // Try body
      .
      .
      .
      RaiseException(Code=0xE06D7363,
         Flag=EXCEPTION_NONCONTINUABLE,
         Args=5);
   }
   __except ((ArgType == Integer) ?
      EXCEPTION_EXECUTE_HANDLER :
      EXCEPTION_CONTINUE_SEARCH) {
      // Catch body
      .
      .
      .
   }
   .
   .
   .
}
```

你可能注意到上面代码中一些有趣的细节。首先，函数 RaiseException的调用使用了异常代码0xE06D7363。这是由Visual C++的开发人员选择的软件异常代码，在引发C++异常时使用。事实上你可以验证这一点，方法是打开调试程序的 Exceptions对话框，滚动到异常列表的底部，见图25-15。

当引发了C++异常时，总要使用EXCEPTION_NONCONTINUEABLE标志。C++异常不能被重新执行。对于诊断C++异常的过滤器，如果返回EXCEPTION_CONTINUE_EXECUTION，那将是个错误。实际上，我们看一看上面程序段右边函数中的 __except过滤器，就会发现它只能计算成EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_SEARCH。

传递到RaiseException的其余参数是用来作为一种机制，用于实际引发指定的变量。被引发的变量信息是如何传递到 RaiseException的，这一点没有公开。但不难想像编译器的开发人员可以实现这一点。

China-pub.com
下载



图25-15　Exceptions 对话框

最后要指出的是__except过滤器。这个过滤器的用途是将throw变量的数据类型同用于C++ catch语句的变量类型相比较。如果数据类型相同，过滤器返回 EXCEPTION_EXECUTE_HANDLER，导致catch块（__except块）中的语句执行。如果数据类型不同，过滤器返回EXCEPTION_CONTNUE_SEARCH，导致catch过滤器上溯要计算的调用树。

注意　由于C++异常在内部是由结构性异常实现的，所以可以在一个程序中使用两种机制。例如，当引起存取违规时，我乐意使用虚拟内存来提交存储区。而 C++语言完全不支持这种类型的再恢复异常处理（resumptive exception handling）。我可以在我的代码的某些部分使用结构化处异常处理。在这些部分需要利用这种结构的优点，可使我自己的__except过滤器返回EXCEPTION_CONTINUE_EXECUTION。对于代码的其他部分，如果不要求这种再恢复异常处理，我还是使用C++异常处理。

用C++来捕获结构性异常

正常情况下，C++异常处理不能使程序从硬件异常中恢复，硬件违规就是存取违规或零作除数这种异常。但微软已经对其编译器增加了这种支持能力。例如，下面的代码可以防止进程不正常地结束：

```
void main() {
   try {
      * (PBYTE) 0 = 0;       // Access violation
   }
   catch (...) {
      // This code handles the access-violation exception
   }
   // The process is terminating normally
}
```

这很棒，它可以使程序从硬件异常中很自然地恢复。但是如果 catch的异常描述能够区分不同的异常代码，那就更棒了。例如，如果能够按下面的方式编写代码，就更好了。

```
void Functastic() {

   try {
      * (PBYTE) 0 = 0;       // Access violation
```

China-pub.com 下载

```
            int x = 0;
            x = 5 / x;              // Division by zero
         }
         catch (StructuredException) {
            switch (StructuredExceptionCode) {
               case EXCEPTION_ACCESS_VIOLATION:
                  // This code handles an access-violation exception
                  break;

               case EXCEPTION_INT_DIVIDE_BY_ZERO:
                  // This code handles a division-by-zero exception
                  break;

               default:
                  // We don't handle any other exceptions
                  throw;    // Maybe another catch is looking for this
                  break;    // Never executes
            }
         }
      }
```

使我们高兴的是，Visuad C++有一种机制可以实现这一点。你需要做的是建立你自己的C++类，在代码中用来标识结构性异常。这里是一个例子：

```
#include <eh.h>                 // For _set_se_translator
.
.
.
class CSE {
public:
   // Call this function for each thread.
   static void MapSEtoCE() { _set_se_translator(TranslateSEtoCE); }

   operator DWORD() { return(m_er.ExceptionCode); }

private:
   CSE(PEXCEPTION_POINTERS pep) {
      m_er      = *pep->ExceptionRecord;
      m_context = *pep->ContextRecord;
   }

   static void _cdecl TranslateSEtoCE(UINT dwEC,
      PEXCEPTION_POINTERS pep) {
      throw CSE(pep);
   }

private:
   EXCEPTION_RECORD m_er;       // CPU independent exception information
   CONTEXT          m_context; // CPU dependent exception information
};
```

在每个线程的进入点函数里，调用静态成员函数MapSEtoCE。这个函数调用C运行时函数_set_se_translator，并传递CSE类的TranslateSEtoCE函数的地址作为参数。通过调用_set_se_translator，告诉C++运行时系统，在结构性异常发生时调用TranslateSEtoCE函数。这个函数构造一个CSE类对象并初始化两个数据成员以包含有关异常的CPU独立和CPU依赖的信息。在构造了CSE对象之后，它就被引发，如同任何正常的变量可被引发一样。现在你的C++

China-pub.com
下载

代码可以通过捕获一个这类的变量来处理结构性异常。

下面是如何捕获这个C++对象的例子。

```
void Functastic() {

   CSE::MapSEtoCE(); // Must be called before any exceptions are raised

   try {
      * (PBYTE) 0 = 0;       // Access violation

      int x = 0;
      x = 5 / x;             // Division by zero
   }
   catch (CSE se) {
      switch (se) {    // Calls the operator DWORD() member function
         case EXCEPTION_ACCESS_VIOLATION:
            // This code handles an access-violation exception
            break;

         case EXCEPTION_INT_DIVIDE_BY_ZERO:
            // This code handles a division-by-zero exception
            break;

         default:
            // We don't handle any other exceptions
            throw;   // Maybe another catch is looking for this
            break;   // Never executes
      }
   }
}
```

China-pub.com
下载

第六部分 窗 口

第26章 窗口消息

本章介绍Microsoft Windows的消息系统是如何支持带有图形用户界面的应用程序的。在设计Windows 2000或Windows 98所使用的窗口系统时，微软有两个主要目标：

- 尽可能保持与过去16位Windows的兼容性，便于开发人员移植他们已有的16位Windows程序。
- 使窗口系统强壮，一个线程不会对系统中其他线程产生不利影响。

但是，这两个目标是直接相互冲突的。在16位Windows系统中，向窗口发送一个消息总是按同步方式执行的：发送程序要在接受消息的窗口完全处理完消息之后才能继续运行。这通常是一个所期望的特性。但是，如果接收消息的窗口花很长的时间来处理消息或者出现挂起，则发送程序就不能再执行。这意味着系统是不强壮的。

这种冲突给微软的设计人员带来了一定的困难。他们的解决方案是两个相互冲突目标之间的出色折衷方案。如果在阅读本章时记住这两个目标，你就会更多地理解微软为什么会做出这样的设计。

我们从一些基本原则开始讨论。Windows允许一个进程至多建立10 000个不同类型的用户对象（User object）：图符、光标、窗口类、菜单、加速键表等等。当一个线程调用一个函数来建立某个对象时，则该对象就归这个线程的进程所拥有。这样，当进程结束时，如果没有明确删除这个对象，则操作系统会自动删除这个对象。对窗口和挂钩（hook)这两种User对象，它们分别由建立窗口和安装挂钩的线程所拥有。如果一个线程建立一个窗口或安装一个挂钩，然后线程结束，操作系统会自动删除窗口或卸载挂钩。

这种线程拥有关系的概念对窗口有重要的意义：建立窗口的线程必须是为窗口处理所有消息的线程。为了使这个概念更加明确具体，可以想像一个线程建立了一个窗口，然后就结束了。在这种情况下，窗口不会收到一个WM_DESTROY或WM_NCDESTROY消息，因为线程已经结束，不可能被用来使窗口接收和处理这些消息。

这也意味着每个线程，如果它至少建立了一个窗口，都由系统对它分配一个消息队列。这个队列用于窗口消息的派送（dispatch)。为了使窗口接收这些消息，线程必须有它自己的消息循环。本章要考查每个线程的消息队列。特别是要看看消息是如何被放置在队列中的，以及线程如何从队列中取出消息并处理它们。

26.1 线程的消息队列

前面已经说过，Windows的一个主要目标是为程序的运行提供一个强壮的环境。为实现这个目标，要保证每个线程运行在一个环境中，在这个环境中每个线程都相信自己是唯一运行的线程。更确切地说，每个线程必须有完全不受其他线程影响的消息队列。而且，每个线程必须有一个模拟环境，使线程可以维持它自己的键盘焦点（keyboard focus)、窗口激活、鼠标捕获等概念。

China-pub.com 下载

当一个线程第一次被建立时，系统假定线程不会被用于任何与用户相关的任务。这样可以减少线程对系统资源的要求。但是，一旦这个线程调用一个与图形用户界面有关的函数（例如检查它的消息队列或建立一个窗口），系统就会为该线程分配一些另外的资源，以便它能够执行与用户界面有关的任务。特别是，系统分配一个THREADINFO结构，并将这个数据结构与线程联系起来。

这个THREADINFO结构包含一组成员变量，利用这组成员，线程可以认为它是在自己独占的环境中运行。THREADINFO是一个内部的、未公开的数据结构，用来指定线程的登记消息队列（posted-message queue）、发送消息队列（send-message queue）、应答消息队列（reply-message queue）、虚拟输入队列（virtualized-input queue）、唤醒标志（wake flag）、以及用来描述线程局部输入状态的若干变量。图26-1描述了THREADINFO结构和与之相联系的三个线程。

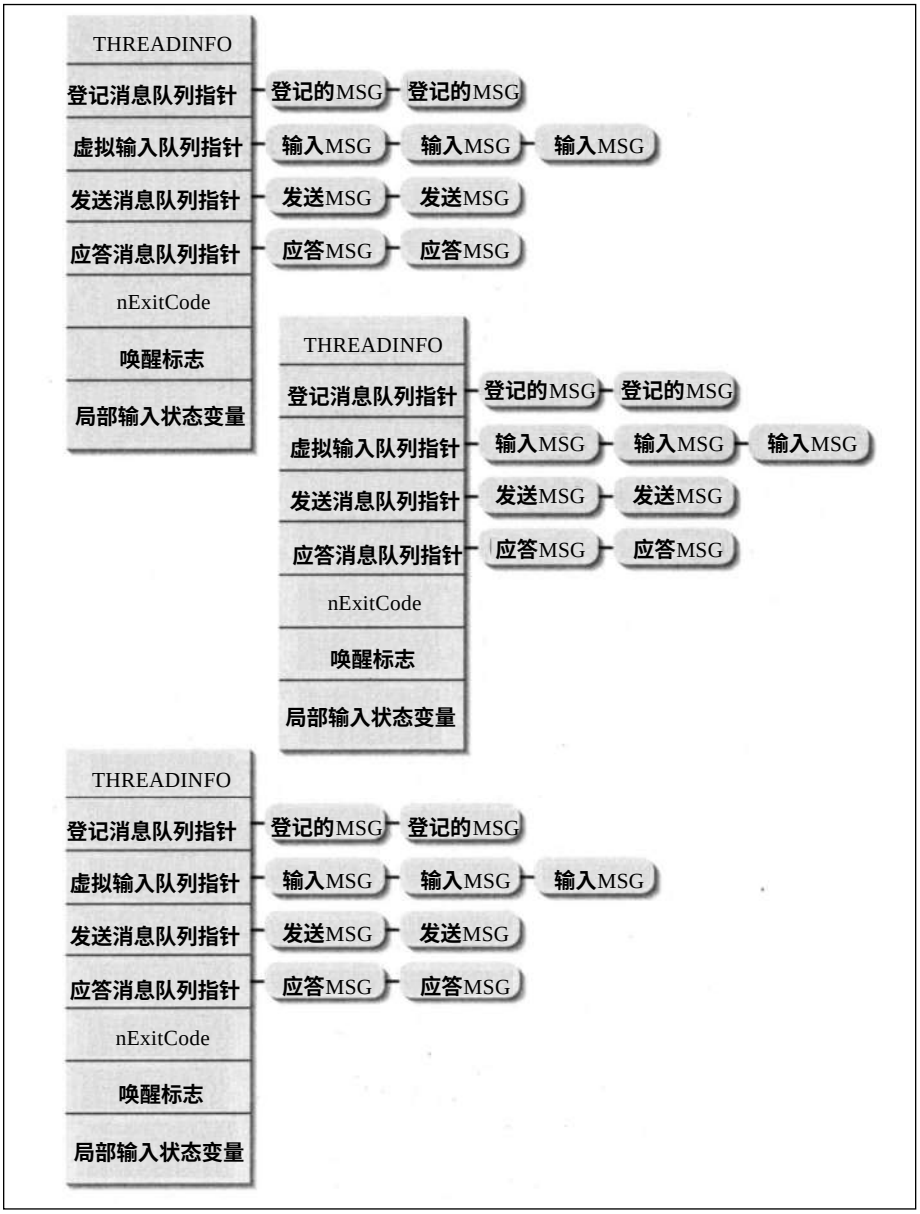


图26-1 三个线程及相应的THREADINFO结构

China-pub.com
下载

这个THREADINFO结构是窗口消息系统的基础，在阅读下面各节内容时，应该参考该图。

26.2 将消息发送到线程的消息队列中

当线程有了与之相联系的THREADINFO结构时，线程就有了自己的消息队列集合。如果一个进程建立了三个线程，并且所有这些线程都调用CreateWindow，则将有三个消息队列集合。消息被放置在线程的登记消息队列中，这要通过调用PostMessage函数来完成：

```
BOOL PostMessage(
   HWND hwnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam);
```

当一个线程调用这个函数时，系统要确定是哪一个线程建立了用 hwnd参数标识的窗口。然后系统分配一块内存，将这个消息参数存储在这块内存中，并将这块内存增加到相应线程的登记消息队列中。并且，这个函数还设置QS_POSTMESSAGE唤醒位（后面会简单讨论）。函数PostMessage在登记了消息之后立即返回，调用该函数的线程不知道登记的消息是否被指定窗口的窗口过程所处理。实际上，有可能这个指定的窗口永远不会收到登记的消息。如果建立这个特定窗口的线程在处理完它的消息队列中的所有消息之前就结束了，就会发生这种事。

还可以通过调用PostThreadMessage将消息放置在线程的登记消息队列中。

```
BOOL PostThreadMessage(
   DWORD dwThreadId,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam);
```

注意 可以通过调用GetWindowsThreadProcessId来确定是哪个线程建立了一个窗口。

```
DWORD GetWindowThreadProcessId(
   HWND hwnd,
   PDWORD pdwProcessId);
```

这个函数返回线程的ID，这个线程建立了hwnd参数所标识的窗口。线程ID在全系统范围内是唯一的。还可以通过对pdwProcessId参数传递一个DWORD地址来获取拥有该线程的进程ID，这个进程ID在全系统范围内也是唯一的。通常，我们不需要进程ID，只须对这个参数传递一个NULL。

PostThreadMessage函数所期望的线程由第一个参数dwThreadId所标记。当消息被设置到队列中时，MSG结构的hwnd成员将设置成NULL。当程序要在主消息循环中执行一些特殊处理时要调用这个函数。

要对线程编写主消息循环以便在GetMessage或PeekMessage取出一个消息时，主消息循环代码检查hwnd是否为NULL，并检查MSG结构的msg成员来执行特殊的处理。如果线程确定了该消息不被指派给一个窗口，则不调用DispatchMessage，消息循环继续取下一个消息。

像PostMessage函数一样，PostThreadMessage在向线程的队列登记了消息之后就立即返回。调用该函数的线程不知道消息是否被处理。

向线程的队列发送消息的函数还有PostQuitMessage：

```
VOID PostQuitMessage(int nExitCode);
```

为了终止线程的消息循环，可以调用这个函数。调用PostQuitMessage类似于调用：

China-pub.com 下载

```
PostThreadMessage(GetCurrentThreadId(), WM_QUIT, nExitCode, 0);
```

但是，PostQuitMessage并不实际登记一个消息到任何一个THREADINFO结构的队列。只是在内部，PostQuitMessage设定QS__QUIT唤醒标志（后面将要讨论），并设置THREADINFO结构的nExitCode成员。因为这些操作永远不会失败，所以PostQuitMessage的原型被定义成返回VOID。

26.3 向窗口发送消息

使用SendMessage函数可以将窗口消息直接发送给一个窗口过程：

```
LRESULT SendMessage(
   HWND hwnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam);
```

窗口过程将处理这个消息。只有当消息被处理之后， SendMessage才能返回到调用程序。由于具有这种同步特性，比之PostMessage或PostThreadMessage，SendMessage用得更频繁。调用这个函数的线程在下一行代码执行之前就知道窗口消息已经被完全处理。

SendMessage是如何工作的呢？如果调用SendMessage的线程向该线程所建立的一个窗口发送一个消息，SendMessage就很简单：它只是调用指定窗口的窗口过程，将其作为一个子例程。当窗口过程完成对消息的处理时，它向SendMessage返回一个值。SendMessage再将这个值返回给调用线程。

但是，当一个线程向其他线程所建立的窗口发送消息时， SendMessage的内部工作就复杂得多（即使两个线程在同一进程中也是如此）。Windows要求建立窗口的线程处理窗口的消息。所以当一个线程调用SendMessage向一个由其他进程所建立的窗口发送一个消息，也就是向其他的线程发送消息，发送线程不可能处理窗口消息，因为发送线程不是运行在接收进程的地址空间中，因此不能访问相应窗口过程的代码和数据。实际上，发送线程要挂起，而由另外的线程处理消息。所以为了向其他线程建立的窗口发送一个窗口消息，系统必须执行下面将讨论的动作。

首先，发送的消息要追加到接收线程的发送消息队列，同时还为这个线程设定QS_SENDMESSAGE标志（后面将讨论）。其次，如果接收线程已经在执行代码并且没有等待消息（如调用GetMessage、PeekMessage或WaitMessage），发送的消息不会被处理，系统不能中断线程来立即处理消息。当接收进程在等待消息时，系统首先检查QS_SENDMESSAGE唤醒标志是否被设定，如果是，系统扫描发送消息队列中消息的列表，并找到第一个发送的消息。有可能在这个队列中有几个发送的消息。例如，几个线程可以同时向一个窗口分别发送消息。当发生这样的事时，系统只是将这些消息追加到接收线程的发送消息队列中。

当接收线程等待消息时，系统从发送消息队列中取出第一个消息并调用适当的窗口过程来处理消息。如果在发送消息队列中再没有消息了，则QS_SENDMESSAGE唤醒标志被关闭。当接收线程处理消息的时候，调用SendMessage的线程被设置成空闲状态（idle），等待一个消息出现在它的应答消息队列中。在发送的消息处理之后，窗口过程的返回值被登记到发送线程的应答消息队列中。发送线程现在被唤醒，取出包含在应答消息队列中的返回值。这个返回值就是调用SendMessage的返回值。这时，发送线程继续正常执行。

当一个线程等待SendMessage返回时，它基本上是处于空闲状态。但它可以执行一个任务：如果系统中另外一个线程向一个窗口发送消息，这个窗口是由这个等待 SendMessage返回的线

China-pub.com
下载

程所建立的，则系统要立即处理发送的消息。在这种情况下，系统不必等待线程去调用GetMessage、Peek Message或WaitMessage。

由于Windows使用上述方法处理线程之间发送的消息，所以有可能造成线程挂起（hang）。例如，当处理发送消息的线程含有错误时，会导致进入死循环。那么对于调用SendMessage的线程会发生什么事呢？它会恢复执行吗？这是否意味着一个程序中的bug会导致另一个程序挂起？答案是确实有这种可能。

利用4个函数——SendMessageTimeout、SendMessageCallback、SendNotifyMessage和ReplyMessage，可以编写保护性代码防止出现这种情况。第一个函数是SendMessageTimeout：

```
LRESULT SendMessageTimeout(
   HWND hwnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam,
   UINT fuFlags,
   UINT uTimeout,
   PDWORD_PTR pdwResult);
```

利用SendMessageTimeout函数，可以规定等待其他线程答回你消息的时间最大值。前4个参数与传递给SendMessage的参数相同。对fuFlags参数，可以传递值SMTO_NORMAL(定义为0)、SMTO_ABORTIFHUNG、SMTO_BLOCK、SMTO_NOTIMEOUTIFNOTHUNG或这些标志的组合。

SMTO_ABORTIFHUNG标志是告诉SendMessageTimeout去查看接收消息的线程是否处于挂起状态，如果是，就立即返回。SMTO_NOTIMEOUTIFNOTHUNG标志使函数在接收消息的线程没有挂起时不考虑等待时间限定值。SMTO_BLOCK标志使调用线程在SendMessageTimeout返回之前，不再处理任何其他发送来的消息。SMTO_NORMAL标志在Winuser.h中定义成0，如果不想指定任何其他标志及组合，就使用这个标志。

前面说过，一个线程在等待发送的消息返回时可以被中断，以便处理另一个发送来的消息。使用SMTO_BLOCK标志阻止系统允许这种中断。仅当线程在等待处理发送的消息的时候（不能处理别的发送消息），才使用这个标志。使用SMTO_BLOCK可能会产生死锁情况，直到等待时间期满。例如，如果你的线程向另外一个线程发送一个消息，而这个线程又需要向你的线程发送消息。在这种情况下，两个线程都不能继续执行，并且都将永远挂起。

SendMessageTimeout函数中的uTimeout参数指定等待应答消息时间的毫秒数。如果这个函数执行成功，返回TRUE，消息的结果复制到一个缓冲区中，该缓冲区的地址由pdwResult参数指定。

顺便提一下，这个函数在WinUser.h头文件中的原型是不正确的。这个函数的原型应该被定义成返回一个BOOL型值，因为LRESULT实际是通过函数的一个参数返回的。这会引起一些问题，因为如果对函数传递一个无效的窗口句柄或者等待超时，SendMessageTimeout都会返回FALSE。要知道函数失败详细情况的唯一办法是调用GetLastError。如果函数是由于等待超时而失败，则GetLastError为0（ERROR_SUCCESS）。如果对参数传递了一个无效句柄，GetLastError为1400（ERROR_INVALID_WINDOW_HANDLE）。

如果调用SendMessageTimeout向调用线程所建立的窗口发送一个消息，系统只是调用这个窗口的窗口过程，并将返回值赋给pdwResult。因为所有的处理都必须发生在一个线程里，调用SendMessageTimeout函数之后出现的代码要等消息被处理完之后才能开始执行。

用来在线程间发送消息的第二个函数是SendMessageCallback：

China-pub.com 下载

```
BOOL SendMessageCallback(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    SENDASYNCPROC pfnResultCallBack,
    ULONG_PTR dwData);
```

同样，前4个参数同SendMessage中使用的一样。当一个线程调用SendMessageCallback时，该函数发送消息到接收线程的发送消息队列，并立即返回使发送线程可以继续执行。当接收线程完成对消息的处理时，一个消息被登记到发送线程的应答消息队列中。然后，系统通过调用一个函数将这个应答通知给发送线程，该函数是使用下面的原型编写的。

```
VOID CALLBACK ResultCallBack(
    HWND hwnd,
    UINT uMsg,
    ULONG_PTR dwData,
    LRESULT lResult);
```

必须将这个函数的地址传递给SendMessageCallback函数作为pfnResultCallback参数值。当调用这个函数时，要把完成消息处理的窗口的句柄传递到第一个参数，将消息值传递给第二个参数。第三个参数dwData，总是取传递到SendMessageCallback函数的dwData参数的值。系统只是取出对SendMessageCallback函数指定的参数值，再直接传递到 ResultCallback函数。ResultCallback函数的最后一个参数是处理消息的窗口过程返回的结果。

因为SendMessageCallback在执行线程间发送时会立即返回，所以在接收线程完成对消息的处理时不是立即调用这个回调函数。而是由接收线程先将一个消息登记到发送线程的应答消息队列。发送线程在下一次调用GetMessage、PeekMessage、WaitMessage或某个SendMessage＊函数时，消息从应答消息队列中取出，并执行ResultCallBack函数。

SendMessageCallback函数还有另外一个用处。Windows提供了一种广播消息的方法，用这种方法你可以向系统中所有现存的重叠（overlapped）窗口广播一个消息。这可以通过调用SendMessage函数，对参数hwnd传递HWND_BROADCAST（定义为－1）。使用这种方法广播的消息，其返回值我们并不感兴趣，因为 SendMessage函数只能返回一个LRESULT。但使用SendMessageCallback，就可以向每一个重叠窗口广播消息，并查看每一个返回结果。对每一个处理消息的窗口的返回结果都要调用ResultCallback函数。

如果调用SendMessageCallback向一个由调用线程所建立的窗口发送一个消息，系统立即调用窗口过程，并且在消息被处理之后，系统调用 ResultCallBack函数。在ResultCallBack函数返回之后，系统从调用SendMessageCallback的后面的代码行开始执行。

线程间发送消息的第三个函数是SendNotifyMessage：

```
BOOL SendNotifyMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

SendNotifyMessage将一个消息置于接收线程的发送消息队列中，并立即返回到调用线程。这一点与PostMessage函数一样，但SendNotifyMessage在两方面与PostMessage不同。

首先，SendNotifyMessage是向另外的线程所建立的窗口发送消息，发送的消息比起接收线程消息队列中存放的登记消息有更高的优先级。换句话说，由 SendNotifyMessage函数存放

China-pub.com
下载

在队列中的消息总是在PostMessage函数登记到队列中的消息之前取出。

其次，当向一个由调用进程建立的窗口发送消息时，SendNotifyMessage同SendMessage函数完全一样：SendNotifyMessage在消息被处理完之后才能返回。

我们已经知道，发送给窗口的大多数消息是用于通知的目的。也就是，发送消息是因为窗口需要知道某个状态已经发生变化，在程序能够继续执行之前，窗口要做某种处理。例如，WM_ACTIVATE、WM_DESTROY、WM_ENABLE、WM_SIZE、WM_SETFOCUS和WM_MOVE等都是系统发送给窗口的通知，而不是登记的消息。这些消息是系统对窗口的通知，因此系统不需要停止运行以等待窗口过程处理这些消息。与此相对应，如果系统向一个窗口发送一个WM_CREATE消息，则在窗口处理完这个消息之前，系统必须等待。如果返回值是－1，则不再建立窗口。

第四个用于线程发送消息的函数是ReplyMessage：

```
BOOL ReplyMessage(LRESULT lResult);
```

这个函数与前面讨论过的三个函数不同。线程使用SendMessageTimeout、SendMessageCallback和SendNotifyMessage发送消息，是为了保护自己以免被挂起。而线程调用ReplyMessage是为了接收窗口消息。当一个线程调用ReplyMessage时，它是要告诉系统，为了知道消息结果，它已经完成了足够的工作，结果应该包装起来并登记到发送线程的应答消息队列中。这将使发送线程醒来，获得结果，并继续执行。

调用ReplyMessage的线程在lResult参数中指出消息处理的结果。在调用ReplyMessage之后，发送消息的线程恢复执行，而处理消息的线程继续处理消息。两个线程都不会被挂起，都可以正常执行。当处理消息的线程从它的窗口过程返回时，它返回的任何值都被忽略。

这里的问题是，ReplyMessage必须在接收消息的窗口过程中调用，而不是由调用某个Send＊函数的线程调用。为了编写保护性代码，最好不要用前面讨论过的三个Send＊函数中的一个代替对SendMessage的调用，而是依靠窗口过程的实现者来调用ReplyMessage。

还应该知道，如果在处理一个由同一线程发送来的消息时调用ReplyMessage，则该函数什么也不做。实际上，这就是ReplyMessage的返回值所指出的。如果你在处理线程间的消息发送时调用了ReplyMessage，则它返回TRUE，如果你在处理线程内的消息发送时调用ReplyMessage，它返回FALSE。

有时候，你可能想知道究竟是在处理线程间的消息发送，还是在处理线程内的消息发送。为了搞清楚这一点，可以调用InSendMessage：

```
BOOL InSendMessage();
```

这个函数的名字不能够确切说明它究竟做什么事。初看时，你会以为，当线程在处理一个发送的消息时，该函数返回TRUE，而在处理一个登记的消息时，它返回FALSE。如果这样想你就错了。这个函数在线程处理线程间发送的消息时，返回TRUE，而在线程处理线程内发送的或登记的消息时，返回FALSE。InSendMessage和ReplyMessage的返回值是一样的。

还可以调用另外一个函数来确定窗口过程正在处理的消息类型：

```
DWORD InSendMessageEx(PVOID pvReserved);
```

当调用这个函数时，必须对pvReserved参数传递NULL。这个函数的返回值指出正在处理的消息的类型。如果返回值是ISMEX_NOSEND（定义为0），表示线程正在处理一个线程内发送的或登记的消息。如果返回值不是ISMEX_NOSEND，就是表26-1中描述的位标志的组合。

China-pub.com 下载

表26-1 位标志的组合

标 志	描 述
ISMEX_SEND	线程在处理一个线程间发送的消息，该消息是用SendMessage或Send Message Timeout函数发送的。如果没有设定ISMEX_REPLIED标志，发送线程被阻塞，等待应答
ISMEX_NOTIFY	线程在处理一个线程间发送的消息，该消息是用SendNotify Message函数发送的。发送线程不等待应答，也不会阻塞
ISMEX_CALLBACK	线程在处理线程间发送的消息，该消息是用 SendMessage Callback发送的。发送线程不等待应答，也不会被阻塞
ISMEX_REPLIED	线程在处理线程间发送的消息，并已经调用ReplyMessage。发送线程不会被阻塞

26.4 唤醒一个线程

当一个线程调用GetMessage或WaitMessage，但没有对这个线程或这个线程所建立窗口的消息时，系统可以挂起这个线程，这样就不再给它分配CPU时间。当有一个消息登记或发送到这个线程，系统要设置一个唤醒标志，指出现在要给这个线程分配 CPU时间，以便处理消息。正常情况下，如果用户不按键或移动鼠标，就没有消息发送给任何窗口。这意味着系统中大多数线程没有被分配给CPU时间。

26.4.1 队列状态标志

当一个线程正在运行时，它可以通过调用GetQueueStatus函数来查询队列的状态：

```
DWORD GetQueueStatus(UINT fuFlags);
```

参数fuFlags是一个标志或一组由OR连接起来的标志，可用来测试特定的唤醒位。表 26-2给出了各个标志取值及含义。

表26-2 标志取值及含义

标 志	队列中的消息
QS_KEY	WM_KEYUP、WM_KEYDOWN、WM_SYSKEYUP或 WM_SYSKEYDOWN
QS_MOUSEMOVE	WM_MOUSEMOVE
QS_MOUSEBUTTON	WM_?BUTTON*（其中？代表L、M或R、*代表DOWN、UP或DBLCLK）
QS_MOUSE	同QS_MOUSEMOVE|QS_MOUSEBUTTON
QS_INPUT	同QS_MOUSE|QS_KEY
QS_PAINT	WM_PAINT
QS_TIMER	WM_TIMER
QS_HOTKEY	WM_HOTKEY
QS_POSTMESSAGE	登记的消息（不同于硬件输入事件）。当队列在期望的消息过滤器范围内没有登记的消息时，这个标志要消除。除此之外，这个标志与QS_ALLPOSTMESSAGE相同
QS_ALLPOSTMESSAGE	登记的消息（不同于硬件输入事件）。当队列完全没有登记的消息时（在任何消息过滤器范围），该标志被清除。除此之外，该标志与QS_POSTMESSAGE相同
QS_ALLEVENTS	同QS_INPUT|QS_POSTMESSAGE|QS_TIMER|QS_PAINT|QS_HOTKEY
QS_QUIT	已调用PostQuitMessage。注意这个标志没有公开，所以在WinUser.h 文件中没有。它由系统在内部使用
QS_SENDMESSAGE	由另一个线程发送的消息
QS_ALLINPUT	同QS_ALLEVENTS|QS_SENDMESSAGE

China-pub.com
下载

当调用GetQueueStatus函数时，fuFlags将队列中要检查的消息的类型告诉GetQueueStatus。用OR连接的QS_*标识符的数量越少，调用执行的就越快。当GetQueueStatus返回时，线程的队列中当前消息的类型在返回值的高字（两字节）中。这个返回的标志的集合总是所想要的标志集的子集。例如，对下面的调用：

```
BOOL fPaintMsgWaiting = HIWORD(GetQueueStatus(QS_TIMER)) & QS_PAINT;
```

fPaintMsgWaiting的值总是FALSE，不论队列中是否有一个WM_PAINT消息在等待，因为GetQueueStatus的参数中没有将QS_PAINT指定为一个标志。

GetQueueStatus返回值的低字指出已经添加到队列中，并且在上一次对函数GetQueueStatus、GetMessage或PeekMessage调用以来还没有处理的消息的类型。

不是所有的唤醒标志都由系统平等对待。对于QS_MOUSEMOVE标志，只要队列中存在一个未处理的WM_MOUSEMOVE消息，这个标志就要被设置。当GetMessage或PeekMessage (利用PM_REMOVE)从队列中放入新的WM_MOUSEMOVE消息之前，这个标志被关闭。QS_KEY、QS_MOUSEBUTTON和QS_HOTKEY标志都根据相应的消息按与此相同的方式处理。

QS_PAINT标志的处理与此不同。如果线程建立的一个窗口有无效的区域，QS_PAINT标志被设置。当这个线程建立的所有窗口所占据的区域变成无效时（通常由于对ValidateRect、ValidateRegion或BeginPaint的调用而引起），QS_PAINT标志就被关闭。只有当线程建立的所有窗口都无效时，这个标志才关闭。调用GetMessage或PeekMessage对这个唤醒标志没有影响。

当线程的登记消息队列中至少有一个消息时，QS_POSTMESSAGE标志就被设置。这不包括线程的虚拟输入队列中的硬件事件消息。当线程的登记消息队列中的所有消息都已经处理，队列变空时，这个标志被复位。

每当一个定时器（由线程所建立）报时（go off），QS_TIMER标志就被设置。在Get Message或PeekMessage返回WM_TIMER事件之后，QS_TIMER标志被复位，直到定时器再次报时。

QS_SENDMESSAGE标志指出有一个消息在线程的发送消息队中。系统在内部使用这个标志，用来确认和处理线程之间发送的消息。对于一个线程向自身发送的消息，不设置这个标志。虽然可以使用QS_SENDMESSAGE标志，但很少需要这样做。笔者还从未见到一个程序使用这个标志。

还有一个未公开的队列状态标志QS_QUIT。当一个线程调用PostQuitMessage时，QS_QUIT标志就被设置。系统并不实际向线程的消息队列追加一个WM_QUIT消息。GetQueueStatus函数也不返回这个标志的状态。

26.4.2 从线程的队列中提取消息的算法

当一个线程调用GetMessage或PeekMessage时，系统必须检查线程的队列状态标志的情况，并确定应该处理哪个消息。图26-2和下面叙述的步骤说明了系统是如何确定线程应该处理的下一个消息的情况。

1) 如果QS_SENDMESSAGE标志被设置，系统向相应的窗口过程发送消息。GetMessage或PeekMessage函数在内部进行这种处理，并且在窗口过程处理完消息之后不返回到线程，这些函数要等待其他要处理的消息。

2) 如果消息在线程的登记消息队列中，函数GetMessage或PeekMessage填充传递给它们的MSG结构，然后函数返回。这时，线程的消息循环通常调用DispatchMessage，让相应的窗口过程来处理消息。

3) 如果QS_QUIT标志被设置。GetMessage或PeekMessage返回一个WM_QUIT消息（其中wParam参数是规定的退出代码）并复位QS_QUIT标志。

China-pub.com 下载

4) **如果消息在线程的虚拟输入队列，函数**GetMessage**或**PeekMessage**返回硬件输入消息。**

5) **如果**QS_PAINT**标志被设置，**GetMessage**或**PeekMessage**为相应的窗口返回一个**WM-PAINT**消息。**

6) **如果**QS_TIMER**标志被设置，**GetMessage**或**PeekMessage**返回一个**WM_TIMER**消息。**

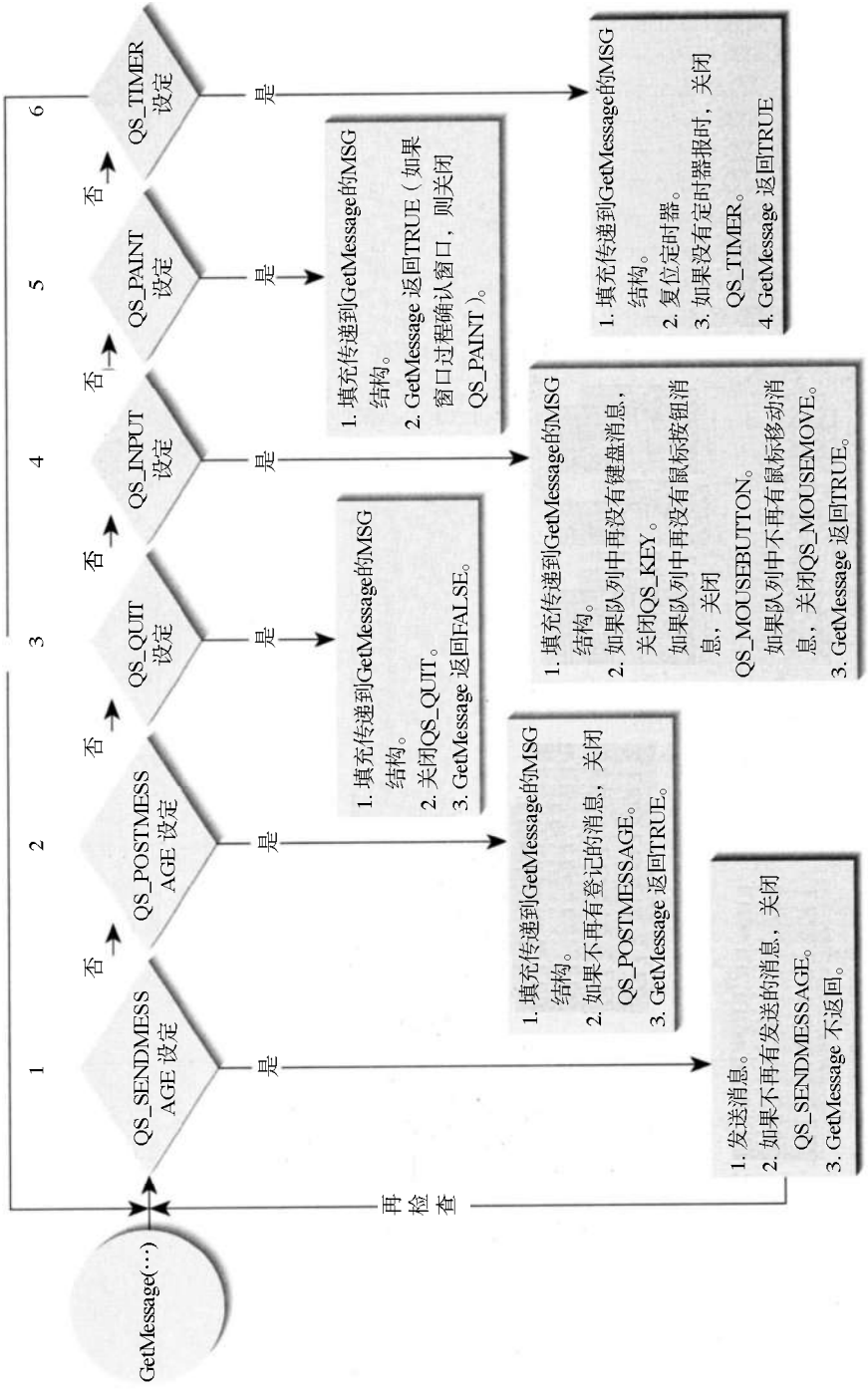


图26-2 **从线程队列中提取消息的算法**

China-pub.com
下载

尽管很难令人相信，但确有理由这样做。微软在设计这个算法时有一个大前提，就是应用程序应该是用户驱动的，用户通过建立硬件输入事件（键盘和鼠标操作）来驱动应用程序。在使用应用程序时，用户可能按一个鼠标按钮，引起一系列要发生的事件。应用程序通过向线程的消息队列中登记消息使每个个别的事件发生。

所以如果按鼠标按钮，处理WM_LBUTTONDOWN消息的窗口可能向不同的窗口投送三个消息。由于是硬件事件引发三个软件事件，所以系统要在读取用户的下一个硬件事件之前，先处理这些软件事件。这也说明了为什么登记消息队列要在虚拟输入队列之前检查。

这种事件序列的一个很好的例子是调用 TranslateMessage函数。这个函数检查是否有一个WM_KEYDOWN或一个WM_SYSKEYDOWN消息从输入队列中取出。如果有一个这样的消息被取出，系统检查虚键（virtual key）信息是否能转换成等价的字符。如果虚键信息能够转换，TranslateMessage调用PostMessage将一个WM_CHAR消息或一个WM_SYSCHAR消息放置在登记消息队列中。下次调用GetMessage时，系统首先检查登记消息队列中的内容，如果其中有消息存在，从队列中取出消息并将其返回。返回的消息将是 WM_CHAR消息或WM_SYSCHAR消息。再下一次调用GetMessage时，系统检查登记消息队列，发现队列已空。系统再检查输入队列，在其中找到WM_（SYS）KEYUP消息。GetMessage返回这个消息。

由于系统是按这种方式工作，下面的硬件事件序列 WM_KEYDOWN、WM_KEYUP生成下面的到窗口过程的消息序列（假定虚键信息可以转换成等价的字符）：

```
WM_KEYDOWN
WM_CHAR
WM_KEYUP
```

现在我们再回过头来讨论系统如何确定从 GetMessage或PeekMessage返回的消息。在系统检查了登记消息队列之后，但尚未检查虚拟输入队列时，它要检查 QS_QUIT标志。我们知道，当线程调用PostQuitMessage时设置QS_QUIT标志。调用PostQuitMessage类似于（但不相同）调用PostThreadMessage。PostThreadMessage将消息放置在消息队列的尾端，并使消息在检查输入队列之前被处理。为什么PostQuitMessage设置一个标志，而不是将WM_QUIT消息放入消息队列中？有两个理由。

第一，在低内存（low memory）情况下，登记一个消息有可能失败。如果一个程序想退出，它应该被允许退出，即使是在低内存的情况下。第二个理由是使用标志可使线程在线程的消息循环结束前完成对所有其他登记消息的处理。例如对下面的代码段， WM_USER消息将先于WM_QUIT消息从队列中取出，尽管WM_USER消息是在调用PostQuitMessage之后登记到队列中的。

```
case WM_CLOSE:
   PostQuitMessage(0);
   PostMessage(hwnd, WM_USER, 0, 0);
```

最后两个消息是WM_PAINT和WM_TIMER。因为画屏幕是一个慢过程，所以WM_PAINT消息的优先级低。如果每当窗口变得无效时就发送一个 WM_PAINT消息，系统运行就会太慢。在键盘输入之后放置 WM_PAINT消息，系统会运行得很快。例如，选择一个调用对话框的菜单项，从框中选定一个项，在对话框出现在屏幕上之前一直按 Enter。如果你的按键速度足够快，按键消息总是先于任何 WM_PAINT消息从队列中取出。当按 Enter接受对话框的选项，对话框窗口被清除，系统复位QS_PAINT标志。

最后一个消息WM_TIMER，比WM_PAINT的优先级还低。为理解这一点，想一想有一个程序用每个WM_TIMER消息来更新它的显示画屏。如果定时器消息来的太快，则显示画屏就

China-pub.com 下载

没有机会重画自己。在WM_TIMER消息之前先处理WM_PAINT消息，就可以避免这个问题，程序总能更新它的显示画屏。

注意 要记住GetMessage或PeekMessage函数只检查唤醒标志和调用线程的消息队列。这意味着一个线程不能从与其他线程挂接的队列中取得消息，包括同一进程内其他线程的消息。

26.4.3 利用内核对象或队列状态标志唤醒线程

GetMessage或PeekMessage函数导致一个线程睡眠，直到该线程需要处理一个与用户界面（UI）相关的任务。有时候，若能让线程被唤醒去处理一个与UI有关的任务或其他任务，就会带来许多方便。例如，一个线程可能启动一个长时间运行的操作，并可以让用户取消这个操作。这个线程需要知道何时操作结束（与UI无关的任务），或用户是否按了Cancel按钮（与UI相关的任务）来结束操作。

一个线程可以调用MsgWaitForMultipleObjects或MsgWaitForMultipleObjectsEx函数，使线程等待它自己的消息。

```
DWORD MsgWaitForMultipleObjects(
   DWORD nCount,
   PHANDLE phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds,
   DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD nCount,
   PHANDLE phObjects,
   DWORD dwMilliseconds,
   DWORD dwWakeMask,
   DWORD dwFlags);
```

这两个函数类似于WaitForMultipleObjects函数（在第9章讨论过）。不同之处是，当一个内核对象变成有信号状态（signaled）或当一个窗口消息需要派送到调用线程建立的窗口时，这两个函数用于线程调度。

在内部，系统只是向内核句柄的数组添加一个事件内核对象。dwWakeMask参数告诉系统何时让事件成为有信号状态。dwWakeMask参数的可能取值的合法范围与可传递到GetQueueStatus函数的参数值一样。

正常情况下，当WaitForMultipleObjects函数返回时，它返回变成有信号状态的对象的索引以满足调用（WAIT_OBJECT_O到WAIT_OBJECT_O+nCount-1）。增加dwWakeMask参数就如同向调用增加又一个句柄。如果由于唤醒掩码，MsgWaitForMultipleObjects(Ex)被满足，返回值将是WAIT_OBJECT_O＋nCount。

这里是一个例子，说明如何调用MsgWaitFor_MultipleObjects：

```
MsgWaitForMultipleObjects(0, NULL, TRUE, INFINITE, QS_INPUT);
```

这条语句的意思是没有传递任何同步对象的句柄，因为nCount和phObjects参数设定了O和NULL。这里让函数等待所有要变成有信号状态的对象，但只指定了一个要等待的对象，参数fWaitAll可以变成FALSE，而不会改变这个调用的作用。这里还告诉系统，程序将等待，不论等多长时间，直到有键盘消息或鼠标消息出现在调用线程的输入队列中。

China-pub.com
下载

当你要用MsgWaitForMultipleObjects函数做某些事的时候，就会发现这个函数缺少许多重要的特性。因此微软不得不又开发了 MsgWaitForMultipleObjectsEx函数。MsgWaitForMultipleObjectsEx是MsgWaitForMultipleObjects的一个超集（superset）。新的特性是通过dwFlags参数引进的。对这个参数，可以指定下面标志的任意组合（见表26-3）。

表26-3 dwFlags 参数的标志

标 志	描 述
MWMO_WAITALL	函数等待所有要变成有信号状态的内核对象及要出现在线程队列中的特定消息。如果没有这个标志，函数等待直到有一个内核对象变成signaled，或指定的消息出现在线程的队列中
MWMO_ALERTABLE	函数在一个可报警状态等待
MWMO_INPUTAVAILABLE	当任何指定的消息在线程的队列中时，函数醒来（本节后面详细解释）

如果不想要任何这些附加的特性，可对参数dwFlags传递零（0）。

下面是有关MsgWaitForMultipleObjects（Ex）的一些重要内容：

- 由于这个函数只是向内核句柄的数组增加一个内部事件内核对象， nCount参数的最大值是MAXIMUM_WAIT_OBJECT减1或63。
- 当对fWaitAll参数传递FALSE时，那么当一个内核对象是有信号的（signaled），或当指定的消息类型出现在线程的队列时，函数返回。
- 当对fWaitAll参数传递TRUE时，那么当所有内核对象成为有信号状态，并且指定的消息类型出现在线程的队列中时，函数返回。这种行为似乎使许多开发人员感到惊讶。开发人员希望有一种办法，当所有内核对象变成有信号的或者当指定的消息类型出现在线程的队列中时，可以唤醒线程。但没有函数能够这样。
- 当调用这两个函数时，实际是查看是否有指定类型的新消息被放入调用线程的队列。

注意，上述最后一条会使许多开发人员吃惊。这里有一个例子。假定一个线程的队列目前包含有两个按键消息。如果这个线程现在要调用 MsgWaitForMultipleObjects（Ex），其中dwWakeMask参数设置成QS_INPUT，线程将被唤醒，从队列中取出第一个按键消息，并处理这个消息。现在，如果这个线程要再调用 MsgWaitForMultipleObjects（Ex），线程将不会被唤醒，因为线程的队列中没有“新”的消息。

对开发人员来说，这已变成了一个主要问题，为此微软增加了 MWMO_INPUTAVAILABLE标志，这只用于MsgWaitForMultipleObjectsEx，而不用于MsgWaitForMultipleObjects。

这里是一个例子，讲述如何适当地编码一个使用MsgWaitForMultipleObjectsEx的消息循环：

```
BOOL  fQuit = FALSE;          // Should the loop terminate?

while (!fQuit) {

   // Wake when the kernel object is signaled OR
   // if we have to process a UI message.
   DWORD dwResult = MsgWaitForMultipleObjectsEx(1, &hEvent,
      INFINITE, QS_ALLEVENTS, MWMO_INPUTAVAILABLE);

   switch (dwResult) {
      case WAIT_OBJECT_0:     // The event became signaled.
         break;

      case WAIT_OBJECT_0 + 1:  // A message is in our queue.
```

China-pub.com 下载

```
            // Dispatch all of the messages.
            MSG msg;
            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

               if (msg.message == WM_QUIT) {
                  // A WM_QUIT message, exit the loop
                  fQuit = TRUE;
               } else {
                  // Translate and dispatch the message.
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
               }
            } // Our queue is empty.
            break;
         }
      }   // End of while loop
```

26.5 通过消息发送数据

本节将讨论系统如何利用窗口消息在进程之间传送数据。一些窗口消息在其 lParam参数中指出了一个内存块的地址。例如， WM_SETTEXT消息使用lParam参数作为指向一个以零结尾的字符串的指针，这个字符串为窗口规定了新的文本标题串。考虑下面的调用：

```
SendMessage(FindWindow(NULL, "Calculator"), WM_SETTEXT,
   0, (LPARAM) "A Test Caption");
```

这个调用看起来不会有害。它确定 Calculator程序窗口的窗口句柄，并试图将窗口的标题改成“ A Test Caption ”。但我们要仔细看一看究竟会发生什么。

新标题的字符串包含在调用进程的地址空间里。所以这个在调用进程空间的字符串的地址将传递给 lParam参数。当Calculator的窗口的窗口过程收到这个消息时，它要查看 lParam参数，并要读取这个以零结尾的字符串，使其成为新的标题。

但lParam中的地址指向调用进程的地址空间中的字符串，而不是alculator的地址空间。这会发生内存存取违规这种严重问题。但当你执行上面的代码时，你会看到执行是成功的，为什么会是这样？

答案是系统特别要检查 WM_SETTEXT消息，并用与处理其他消息不同的方法来处理这个消息。当调用 SendMessage时，函数中的代码要检查是否要发送一个 WM_SETTEXT消息。如果是，就将以零结尾的字符串从调用进程的地址空间放入到一个内存映像文件中，该内存映像文件可在进程间共享。然后再发送消息到其他进程的线程。当接收线程已准备好处理 WM_SETTEXT消息时，它在自己的地址空间中确定包含新的窗口文本标题的共享内存映像文件的位置，再将WM_SETTEXT消息派送到相应的窗口过程。在处理完消息之后，内存映像文件被删除。这样做看起来是不是太麻烦了一些。

幸而大多数消息不要求这种类型的处理。仅当这种消息是程序在进程间发送的消息，特别是消息的wParam或lParam参数表示一个指向数据结构的指针时，要做这样的处理。

我们再来看另外一个要求系统特殊处理的例子—— WM_GETTEXT消息。假定一个程序包含下面的代码：

```
char szBuf[200];
SendMessage(FindWindow(NULL, "Calculator"), WM_GETTEXT,
   sizeof(szBuf), (LPARAM) szBuf);
```

WM_GETTEXT消息请求Calculator的窗口过程用该窗口的标题填充 szBuf所指定的缓冲区。当一个进程向另一个进程的窗口发送这个消息时，系统实际上必须发送两个消息。首先，系统

China-pub.com
下载

要向那个窗口发送一个WM_GETTEXTLENGTH消息。窗口过程通过返回窗口标题的字符数来响应这个消息。系统利用这个数字来建立一个内存映像文件，用于在两个进程之间共享。

当内存映像文件被建立时，系统就发送消息来填充它。然后系统再转回到最初调用SendMessage的进程，从共享内存映像文件中将数据复制到 szBuf所指定的缓冲区中，然后从SendMessage调用返回。

对于系统已经知道的消息，发送消息时都可以按相应的方式来处理。如果你要建立自己的（WM_USER＋x）消息，并从一个进程向另一个进程的窗口发送，那又会怎么样？系统不知道你要用内存映像文件并在发送消息时改变指针。为此，微软建立了一个特殊的窗口消息，WM_COPYDATA以解决这个问题：

```
COPYDATASTRUCT cds;
SendMessage(hwndReceiver, WM_COPYDATA,
   (WPARAM) hwndSender, (LPARAM) &cds);
```

COPYDATASTRUCT是一个结构，定义在WinUser.h文件中，形式如下面的样子：

```
typedef struct tagCOPYDATASTRUCT {
   ULONG_PTR dwData;
   DWORD cbData;
   PVOID lpData;
} COPYDATASTRUCT;
```

当一个进程要向另一个进程的窗口发送一些数据时，必须先初始化 COPYDATASTRUCT结构。数据成员dwData是一个备用的数据项，可以存放任何值。例如，你有可能向另外的进程发送不同类型或不同类别的数据。可以用这个数据来指出要发送数据的内容。

cbData数据成员规定了向另外的进程发送的字节数， lpData数据成员指向要发送的第一个字节。lpData所指向的地址，当然在发送进程的地址空间中。

当SendMessage看到要发送一个WM_COPYDATA消息时，它建立一个内存映像文件，大小是cbData字节，并从发送进程的地址空间中向这个内存映像文件中复制数据。然后再向目的窗口发送消息。在接收消息的窗口过程处理这个消息时， lParam参数指向已在接收进程地址空间的一个COPYDATASTRUCT结构。这个结构的lpData成员指向接收进程地址空间中的共享内存映像文件的视图。

关于WM_COPYDATA消息，应该注意三个重要问题：

- 只能发送这个消息，不能登记这个消息。不能登记一个 WM_COPYDATA消息，因为在接收消息的窗口过程处理完消息之后，系统必须释放内存映像文件。如果登记这个消息，系统不知道这个消息何时被处理，所以也不能释放复制的内存块。
- 系统从另外的进程的地址空间中复制数据要花费一些时间。所以不应该让发送程序中运行的其他线程修改这个内存块，直到SendMessage调用返回。
- 利用WM_COPYDATA消息，可以实现16位和32位之间的通信。它也能实现32位与64位之间的通信。这是使新程序同旧程序交流的便捷方法。注意在 Windows 2000和Windows 98上完全支持WM_COPYDATA。但如果你依然在编写 16位Windows程序， Microsofe Visual C++ 1.52没有WM_COPYDATA消息的定义，也没有COPYDATASTRUCT结构的定义。需要手工添加这些代码：

```
// Manually include this in your 16-bit Windows source code.
#define WM_COPYDATA   0x004A

typedef VOID FAR* PVOID;
typedef struct tagCOPYDATASTRUCT {
   DWORD dwData;
```

China-pub.com 下载

```
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, FAR* PCOPYDATASTRUCT;
```

在解决进程间的通信问题方面，WM_COPYDATA消息是一个非常好的工具，可以节省程序员的许多时间。关于使用WM_COPYDATA消息的一个精采例子，见第22章的LastMsgBoxInfo示例程序。

CopyData示例程序

清单26-1所列的CopyData程序（“26 CopyData.exe）说明了如何使用WM_COPYDATA消息从一个程序向另一个程序发送一个数据块。该程序的源代码和资源文件在本书所附光盘的26-CopyData目录下。要看它如何工作，至少需要让两个CopyData的实例运行。每次启动一个CopyData时，它要显示如图26-3所示的对话框。



图26-3 CopyData Application 对话框

为了观看从一个程序到另一个程序的数据复制，首先改变Data1和Data2编辑控制框中的文本。然后点击某个Send Data* to Other Windows按钮，程序向所有运行的CopyData的实例发送数据。每个实例更新自己的编辑框中的内容来反应新数据。

下面描述CopyData如何工作。当一个用户点击图26-3中两个按钮中的某一个时，CopyData执行下面的动作。

1) 如果用户点击Send Data1 To Other Windows按钮，用0来初始化COPYDATASTRUCT的dwData成员，如果用户点击Send Data2 To Other Windows按钮，则用1来初始化dwData成员。

2) 从相应的文本框中求取文本串的长度（按字符数计），并加1（对应一个零结束符）。这个值乘以Sizeof(TCHAR)，从字符数转换成字节数。结果存入COPYDATASTRUCT的cbData成员中。

3) 调用_alloca分配一个内存块，大小足以容纳编辑框中的字符串加上零结束符。这个块的地址存放在COPYDATASTRUCT结构的lpData成员中。

4) 从编辑框向分配的内存块复制字符串。

这个时候，一切就绪，准备向其他窗口发送数据。为了确定要向哪个窗口发送WM_COPYDATA消息，CopyData调用FindWindowEx函数传递它自己的对话框标题，以便只有其他的CopyData程序实例才会被枚举。当找到每个实例的窗口时，发送WM_COPYDATA消息，每个实例更新它的编辑控制框。

清单26-1 CopyData示例程序

CopyData.cpp

```
/******************************************************************************
Module:  CopyData.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
```

China-pub.com
下载


```
#include <malloc.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

// WindowsX.h doesn't have a prototype for Cls_OnCopyData, so here it is.
/* BOOL Cls_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) */

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT cds) {

   Edit_SetText(GetDlgItem(hwnd, cds->dwData ? IDC_DATA2 : IDC_DATA1),
      (PTSTR) cds->lpData);

   return(TRUE);
}
//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_COPYDATA);

   // Initialize the edit controls with some test data.
   Edit_SetText(GetDlgItem(hwnd, IDC_DATA1), TEXT("Some test data"));
   Edit_SetText(GetDlgItem(hwnd, IDC_DATA2), TEXT("Some more test data"));
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_COPYDATA1:
      case IDC_COPYDATA2:
         if (codeNotify != BN_CLICKED)
            break;

         HWND hwndEdit = GetDlgItem(hwnd,
            (id == IDC_COPYDATA1) ? IDC_DATA1 : IDC_DATA2);

         // Prepare the COPYDATASTRUCT.
         COPYDATASTRUCT cds;
```

China-pub.com 下载

```
         // Indicate which data field we're sending (0=ID_DATA1, 1=ID_DATA2)
         cds.dwData = (DWORD) ((id == IDC_COPYDATA1) ? 0 : 1);

         // Get the length (in bytes) of the data block we're sending.
         cds.cbData = (Edit_GetTextLength(hwndEdit) + 1) * sizeof(TCHAR);

         // Allocate a block of memory to hold the string.
         cds.lpData = _alloca(cds.cbData);
         // Put the edit control's string in the data block.
         Edit_GetText(hwndEdit, (PTSTR) cds.lpData, cds.cbData);

         // Get the caption of our window.
         TCHAR szCaption[100];
         GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

         // Enumerate through all the top-level windows with the same caption.
         HWND hwndT = NULL;
         do {
            hwndT = FindWindowEx(NULL, hwndT, NULL, szCaption);
            if (hwndT != NULL) {
               FORWARD_WM_COPYDATA(hwndT, hwnd, &cds, SendMessage);
            }
         } while (hwndT != NULL);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_COPYDATA,   Dlg_OnCopyData);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_COPYDATA), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

CopyData.rc

```
//Microsoft Developer Studio generated resource script.
```

China-pub.com
下载


```
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_COPYDATA DIALOG DISCARDABLE  38, 36, 220, 42
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "CopyData Application"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Data&1:",IDC_STATIC,4,4,24,12
    EDITTEXT        IDC_DATA1,28,4,76,12
    PUSHBUTTON      "&Send Data1 to other windows",IDC_COPYDATA1,112,4,104,
                    14,WS_GROUP
    LTEXT           "Data&2:",IDC_STATIC,4,24,24,12
    EDITTEXT        IDC_DATA2,28,24,76,12
    PUSHBUTTON      "Send &Data2 to other windows",IDC_COPYDATA2,112,24,104,
                    14,WS_GROUP
END

////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_COPYDATA            ICON    DISCARDABLE     "CopyData.Ico"

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
```

China-pub.com 下载

```
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

26.6 Windows如何处理ANSI／Unicode字符和字符串

Windows 98 Windows 98只支持ANSI窗口类和ANSI窗口过程。

当你注册一个新的窗口类时，必须将负责为这个类处理消息的窗口过程的地址告诉系统。对某些消息（如WM_SETTEXT），消息的lParam参数指向一个字符串。在此之前，为了派送消息，使它被正确地处理，系统需要知道窗口过程要求该字符串是ANSI字符串还是Unicode字符串。

告诉系统一个窗口过程是要求ANSI字符串还是Unicode字符串，实际上取决于注册窗口类时所使用的函数。如果构造WNDCLASS结构，并调用RegisterClassA，系统就认为窗口过程要求所有的字符串和字符都属于ANSI。而用RegisterClassW注册窗口类，则系统就向窗口过程派送Unicode字符串和字符。宏RegisterClass对RegisterClassA和RegisterClassW都做了扩展，究竟代表哪一个要看在编译源模块时是否定义了UNICODE。

如果有了一个窗口句柄，就可以确定窗口过程所要求的字符和字符串类型。这可以通过调用下面的函数实现：

```
BOOL IsWindowUnicode(HWND hwnd);
```

China-pub.com
下载

如果这个窗口的窗口过程要求Unicode，这个函数返回TRUE，否则返回FALSE。

如果你建立一个ANSI串，并向一个窗口过程要求Unicode串的窗口发送WM_SETTEXT消息，则系统在发送消息之前，为你自动地转换字符串。很少需要调用IsWindow Unicode函数。

如果你对窗口过程派生子类，系统也会为你执行自动的转换。假定一个编辑控制框的窗口过程要求字符和字符串是Unicode。在你的程序的某处建立了一个编辑控制框，并建立窗口过程的子类，这可以调用

```
LONG_PTR SetWindowLongPtrA(
   HWND hwnd,
   int nIndex,
   LONG_PTR dwNewLong);
```

或

```
LONG_PTR SetWindowLongPtrW(
   HWND hwnd,
   int nIndex,
   LONG_PTR dwNewLong);
```

并将nIndex参数设置成GCLP_WNDPROC，dwNewLong参数设置成子类过程的地址。如果这个子类过程要求ANSI字符和字符串会出现什么情况？这可能引起严重的问题。系统决定怎样转换字符串和字符，要取决于究竟是用上面两个函数中的哪一个来建立子类。如果是调用SetWindowLongPtrA，就是告诉系统新的窗口过程（即子类过程）要接收ANSI字符和字符串。实际上，如果在调用SetWindowLongPtrA之后调用IsWindowUnicode函数，将返回FALSE，表示这个子类的编辑窗口过程不再要求Unicode字符和字符串。

但现在又有一个新的问题：如何能够保证原来的窗口过程得到正确的字符和字符串类型？系统需要有两条信息，才能正确地转换字符和字符串。第一条信息就是字符和字符串当前所具有的形式。这可以通过调用CallWindowProcA或CallWindowProcW来告诉系统：

```
LRESULT CallWindowProcA(
   WNDPROC wndprcPrev,
   HWND hwnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam);

LRESULT CallWindowProcW(
   WNDPROC wndprcPrev,
   HWND hwnd,
   UINT uMsg,
   WPARAM wParam,
   LPARAM lParam);
```

如果子类过程要把ANSI字符串传递给原来的窗口过程，子类过程必须调用CallWindow ProcA。如果子类过程要把Unicode字符串传递给原来的窗口过程，则子类过程必须调用CallWindowProcW。

系统需要的第二条信息是原来的窗口过程所要求的字符和字符串类型。系统从原来窗口过程的地址获取这个信息。当调用SetWindowLongPtrA或SetWindowLongPtrW函数时，系统要查看是否使用了一个ANSI子类过程派生了一个Unicode窗口过程，或用一个Unicode子类过程派生了一个ANSI窗口过程。如果没有改变所要求的字符串类型，则SetWindowLongPtr只返回原

China-pub.com 下载

先窗口过程的地址。如果改变了窗口过程要求的字符和字符串类型，SetWindowLongPtr不是返回原先窗口过程的实际地址，而是返回一个内部子系统数据结构的句柄。

这个数据结构包含原先窗口过程的地址及一个数值，用来指示窗口过程是要求ANSI还是要求Unicode字符串。当调用CallWindowProc时，系统要查看是传递了某个内部数据结构的地址，还是传递了一个窗口过程的地址。如果传递了一个窗口过程的地址，则调用原先的窗口过程，不需要执行字符和字符串转换。

如果传递了一个内部数据结构的句柄，则系统要将字符和字符串转换成适当的类型（ANSI或Unicode），然后调用原先的窗口过程。

China-pub.com
下载

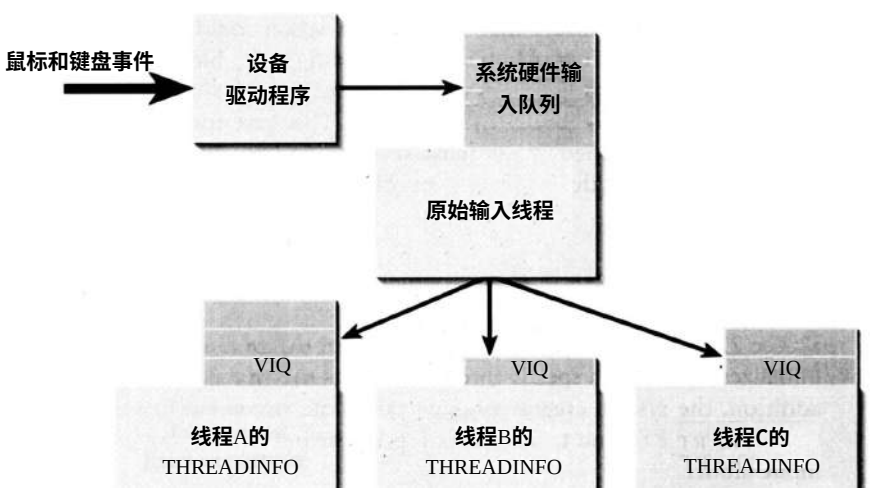
China-pub.com
下载

第27章　硬件输入模型和局部输入状态

本章将讨论系统的硬件输入模型。重点将考察按键和鼠标事件是如何进入系统并发送给适当的窗口过程的。微软设计输入模型的一个主要目标就是为了保证一个线程的动作不要对其他线程的动作产生不好的影响。这里是一个16位Windows中的例子：如果一个任务引起一个死循环，所有的任务都被挂起，不能再响应用户。用户只能重新启动计算机。这样就给一个单个的任务太多的控制。强壮的操作系统，例如Windows 2000和Windows 98，不会使一个挂起的线程妨碍系统中其他线程接收硬件输入。

27.1　原始输入线程

图27-1概括描述了系统的硬件输入模型。当系统初始化时，要建立一个特殊的线程，即原始输入线程（raw input thread，RIT）。此外，系统还要建立一个队列，称为系统硬件输入队列（System hardware input queue, SHIQ）。RIT和SHIQ构成系统硬件输入模型的核心。



China-pub.com 下载

件输入事件的响应。

那么RIT怎么才能知道要向哪一个线程的虚拟输入队列里增加硬件输入消息？对鼠标消息，RIT只是确定是哪一个窗口在鼠标光标之下。利用这个窗口， RIT调用GetWindowThreadProcessId来确定是哪个线程建立了这个窗口。返回的线程 ID指出哪一个线程应该得到这个鼠标消息。

对按键硬件事件的处理稍有不同。在任何给定的时刻，只有一个线程同 RIT“连接”。这个线程称为前景线程（foreground thread），因为它建立了正在与用户交互的窗口，并且这个线程的窗口相对于其他线程所建立的窗口来说处在画面中的前景。

当一个用户在系统上登录时， Windows Explorer进程让一个线程建立相应的任务栏（taskbar）和桌面。这个线程连接到RIT。如果你又要产生Calculator，那么就又有一个线程来建立一个窗口，并且这个线程变成连接到 RIT的线程。注意现在Windows Explorer的线程不再与RIT连接，因为在一个时刻只能有一个线程同RIT连接。当一个按键消息进入SHIQ时，RIT就被唤醒，将这个事件转换成适当的按键消息，并将消息放入与RIT连接的线程的虚拟输入队列。

不同的线程是如何连接到RIT的呢？我们已经说过，当产生一个进程时，这个进程的线程可以建立一个窗口。这个窗口处于前景，其建立窗口的线程同 RIT相连接。另外，RIT还要负责处理特殊的键组合，如Alt+Tab、Alt+Esc和Ctrl+Alt+Del等。因为RIT在内部处理这些键组合，就可以保证用户总能用键盘激活窗口。应用程序不能够拦截和废弃这些键组合。当用户按动了某个特殊的键组合时，RIT激活选定的窗口，并将窗口的线程连接到RIT。Windows也提供激活窗口的功能，使窗口的线程连接到RIT。这些功能在本章后面讨论。

从图27-1中可以看到如何保护线程，避免相互影响的。如果 RIT向窗口 B_1或窗口 B_2发送一个消息，消息到达线程 B的虚拟输入队列。在处理消息时，线程 B在与五个内核对象同步时可能会进入死循环或死锁。如果发生这种情况，线程仍然同 RIT连接在一起，并且可能有更多的消息要增加到线程的虚拟输入队列中。

这种情况下，用户会发现窗口 B1和B2都没有反应，可能想切换到窗口 A_1。为了做这种切换，用户按Alt+Tab。因为是RIT处理Alt+Tab按键组合，所以用户总能切换到另外的窗口，不会有什么问题。在选定窗口 A_1之后，线程 A就连接到RIT。这个时候，用户就可以对窗口 A_1进入输入，尽管线程及其窗口都没有响应。

27.2 局部输入状态

China-pub.com
下载

- 哪一个窗口有鼠标捕获。
- 鼠标光标的形状。
- 鼠标光标的可见性。

由于每个线程都有自己的输入状态变量，每个线程都有不同的焦点窗口、鼠标捕获窗口等概念。从一个线程的角度来看，或者它的某个窗口拥有键盘焦点，或者系统中没有窗口拥有键盘焦点；或者它的某个窗口拥有鼠标捕获，或者系统中没有窗口拥有鼠标捕获，等等。读者会想到，这种隔离应该有一些细节，对此我们将在后面讨论。

27.2.1 键盘输入与焦点

我们已经知道，RIT使用户的键盘输入流向一个线程的虚拟输入队列，而不是流向一个窗口。RIT将键盘事件放入线程的虚拟输入队列时不用涉及具体的窗口。当这个线程调用GetMessage时，键盘事件从队列中移出并分派给当前有输入焦点的窗口。（由该线程所建立）。图27-2说明了这个处理过程。

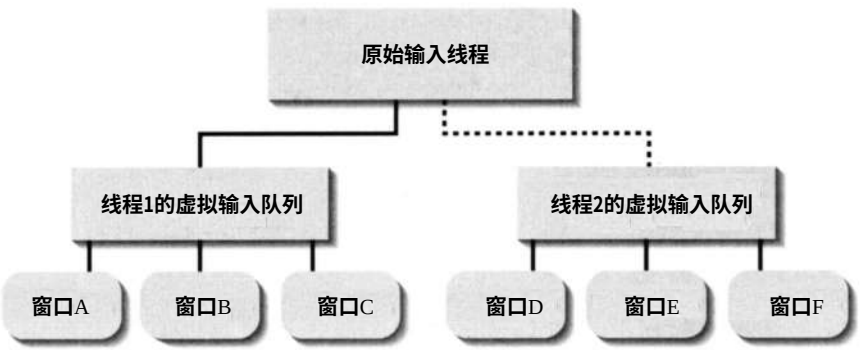


图27-2 RIT将用户的键盘输入导向一个线程的虚拟输入队列

线程1当前正在从RIT接收输入，用窗口A、窗口B或窗口C的句柄作参数调用SetFocus会引起焦点改变。失去焦点的窗口除去它的焦点矩形或隐藏它的插入符号，获得焦点的窗口画出焦点矩形或显示它的插入符号。

假定线程1仍然从RIT接收输入，并用窗口E的句柄作为参数调用SetFocus。这种情况下，

China-pub.com 下载

的线程不一样，那么，对于建立失去焦点窗口的线程，要更新它的局部输入状态变量，说明它没有窗口拥有焦点。这时调用GetFocus将返回NULL，这会使线程知道当前没有窗口拥有焦点。

函数SetActiveWindow激活系统中一个最高层（top-level）的窗口，并对这个窗口设定焦点：

```
HWND SetActiveWindow(HWND hwnd);
```

同SetFocus函数一样，如果调用线程没有创建作为函数参数的窗口，则这个函数什么也不做。

与SetActiveWindow相配合的函数是GetActiveWindow函数：

```
HWND GetActiveWindow();
```

这个函数的功能同GetFocus函数差不多，不同之处是它返回由调用线程的局部输入状态变量所指出的活动窗口的句柄。当活动窗口属于另外的线程时， GetActiveWindow返回NULL。

其他可以改变窗口的 Z序（Z-order）、活动状态和焦点状态的函数还包括 BringWindowTo Top和SetWindowPos：

```
BOOL BringWindowToTop(HWND hwnd);

BOOL SetWindowPos(
   HWND hwnd,
   HWND hwndInsertAfter,
   int x,
   int y,
   int cx,
   int cy,
   UINT fuFlags);
```

这两个函数功能相同（实际上， BringWindowToTop函数在内部调用 SetWindowPos，以HWND_TOP作为第二个参数）。如果调用这两个函数的线程没有连接到 RIT，则函数什么也不做。如果调用这些函数的线程同 RIT相连接，系统就会激活相应的窗口。注意即使调用线程不是建立这个窗口的线程，也同样有效。这意味着，这个窗口变成活动的，并且建立这个窗口的线程被连接到RIT。这也引起调用线程和新连接到RIT的线程的局部输入状态变量被更新。

有时候，一个线程想让它的窗口成为屏幕的前景。例如，有可能会利用 Microsoft Qutlook安排一个会议。在会议开始前的半小时， Outlook弹出一个对话框提醒用户会议将要开始。如果Qutlook的线程没有连接到RIT，这个对话框就会藏在其他窗口的后面，有可能看不见它。因

China-pub.com
下载

为了制止这种现象，微软对SetForegroundWindow函数增加了更多的智能。特别规定，仅当调用一个函数的线程已经连接到RIT或者当前与RIT相连接的线程在一定的时间内（这个时间量由SystemParametersInfo函数和SPI_SETFOREGROUND_LOCKTIMEOUT值来控制）没有收到任何输入，这个函数才有效。另外，如果有一个菜单是活动的，这个函数就失效。

如果不允许SetForegroundWindow将窗口移到前景，它会闪烁该窗口的标题栏和任务条上该窗口的按钮。用户看到任务条按钮闪烁，就知道该窗口想得到用户的注意。用户应该手工激活这个窗口，看一看要报告什么信息。还可以用SystemParametersInfo函数和SPI_SETFOREGROUND-FLASHCOUNT值来控制闪烁。

由于这些新的内容，系统又提供了另外一些函数。如果调用AllowSetForegroundWindow的线程能够成功调用SetForegroundWindow，第一个函数（见下面所列）可使指定进程的一个线程成功调用SetForegroundWindow。为了使任何进程都可以在你的线程的窗口上弹出一个窗口，指定ASFW_ANY(定义为－1)作为dwProcessId参数：

```
BOOL AllowSetForegroundWindow(DWORD dwProcessId);
```

此外，线程可以锁定SetForegroundWindow函数，使它总是失效的。方法是调用LockSetForegroundWindow。

```
BOOL LockSetForegroundWindow(UINT uLockCode);
```

对uLockCode参数可以指定LSFW_LOCK或者LSFW_UNLOCK。当一个菜单被激活时，系统在内部调用这个函数，这样一个试图跳到前景的窗口就不能关闭这个菜单。Windows Explorer在显示Start菜单时，需要明确地调用这些函数，因为Start菜单不是一个内置菜单。当用户按了Alt键或者将一个窗口拉到前景时，系统自动解锁SetForegroundWindow函数。这可以防止一个程序一直对SetForegroundWindow函数封锁。

关于键盘管理和局部输入状态，其他的内容是同步键状态数组。每个线程的局部输入状态变量都包含一个同步键状态数组，但所有的线程要共享一个同步键状态数组。这些数组反映了在任何给定时刻键盘所有键的状态。利用GetAsyncKeyState函数可以确定用户当前是否按下了键盘上的一个键：

```
SHORT GetAsyncKeyState(int nVirtKey);
```

参数nVirtKey指出要检查键的虚键代码。结果的高位指出该键当前是否被按下（是为1，否为0）。笔者在处理一个消息时，常用这个函数来检查用户是否释放了鼠标主按钮。为函数参数赋一个虚键值VK_LBUTTON，并等待返回值的高位成为0。注意，如果调用函数的线程不是

China-pub.com 下载

建立的窗口上，鼠标光标就可见了。

鼠标光标管理的另一个方面是使用ClipCursor函数将鼠标光标剪贴到一个矩形区域。

```
BOOL ClipCursor(CONST RECT *prc);
```

这个函数使鼠标被限制在一个由prc参数指定的矩形区域内。当一个程序调用ClipCursor函数时，系统该做些什么呢？允许剪贴鼠标光标可能会对其他线程产生不利影响，而不允许剪贴鼠标光标又会影响调用线程。微软实现了一种折衷的方案。当一个线程调用这个函数时，系统将鼠标光标剪贴到指定的矩形区域。但是，如果同步激活事件发生（当用户点击了其他程序的窗口，调用了SetForegroundWindow，或按了Ctrl+Esc组合键），系统停止剪贴鼠标光标的移动，允许鼠标光标在整个屏幕上自由移动。

现在我们再讨论鼠标捕获。当一个窗口“捕获”鼠标（通过调用 SetCapture）时，它要求所有的鼠标消息从RIT发到调用线程的虚拟输入队列，并且所有的鼠标消息从虚拟输入队列发到设置捕获的窗口。在程序调用 ReleaseCapture之前，要一直继续这种鼠标消息的捕获。

鼠标的捕获必须同系统的强壮性折衷，也只能是一种折衷。当一个程序调用 SetCapture时，RIT将所有鼠标消息放入线程的虚拟输入队列。 SetCapture还要为调用SetCapture的线程设置局部输入状态变量。

通常一个程序在用户按一个鼠标按钮时调用 SetCapture。但是即使鼠标按钮没有被按下，也没有理由说一个线程不能调用SetCapture。如果当一个鼠标按下时调用SetCapture，捕获在全系统范围内执行。但当系统检测出没有鼠标按钮按下时， RIT不再将鼠标消息只发往线程的虚拟输入队列，而是将鼠标消息发往与鼠标光标所在的窗口相联系的输入队列。这是不做鼠标捕获时的正常行为。

但是，最初调用 SetCapture的线程仍然认为鼠标捕获有效。因此，每当鼠标处于有捕获设置的线程所建立的窗口时，鼠标消息将发往这个线程的捕获窗口。换言之，当用户释放了所有的鼠标按钮时，鼠标捕获不再在全系统范围内执行，而是在一个线程的局部范围内执行。

此外，如果用户想激活一个其他线程所建立的窗口，系统自动向设置捕获的线程发送鼠标按钮按下和鼠标按钮放开的消息。然后系统更新线程的局部输入状态变量，指出该线程不再具有鼠标捕获。很明显，通过这种实现方式，微软希望鼠标点击和拖动是使用鼠标捕获的最常见理由。

China-pub.com
下载

27.3 将虚拟输入队列同局部输入状态挂接在一起

从上面的讨论我们可以看出这个输入模型是强壮的，因为每个线程都有自己的局部输入状态环境，并且在必要时每个线程可以连接到RIT或从RIT断开。有时候，我们可能想让两个或多个线程共享一组局部输入状态变量及一个虚拟输入队列。

可以利用AttachThreadInput函数来强制两个或多个线程共享同一个虚拟输入队列和一组局部输入状态变量：

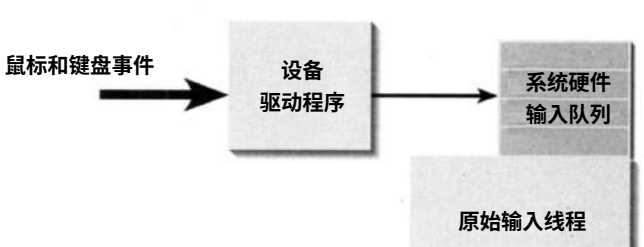
```
BOOL AttachThreadInput(
   DWORD idAttach,
   DWORD idAttachTo,
   BOOL fAttach);
```

函数的第一个参数idAttach，是一个线程的ID，该线程所包含的虚拟输入队列（以及局部输入状态变量）是你不想再使用的。第二个参数idAttachTo，是另一个线程的ID，这个线程所包含的虚拟输入队列（和局部输入状态变量）是想让两个线程共享的。第三个参数fAttach，当想让共享发生时，被设置为TRUE，当想把两个线程的虚拟输入队列和局部输入状态变量分开时，设定为FALSE。可以通过多次调用AttachThreadInput函数让多个线程共享同一个虚拟输入队列和局部输入状态变量。

我们再考虑前面的例子，假定线程A调用AttachThreadInput，传递线程A的ID作为第一个参数，线程B的ID作为第二个参数，TRUE作为最后一个参数：

```
SHORT GetKeyState(int nVirtKey);
```

如图27-3所示，现在每个发往窗口A1、窗口B1或窗口B2的硬件输入事件都将添加到线程B的虚拟输入队列中。线程A的虚拟输入队列将不再接收输入事件，除非再一次调用AttachThreadInput并传递FALSE作为最后一个参数，将两个线程的输入队列分开。



China-pub.com 下载

当将两个线程的输入都挂接在一起时，就使线程共享单一的虚拟输入队列和同一组局部输入状态变量。但线程仍然使用自己的登记消息队列、发送消息队列、应答消息队列和唤醒标志（见第26章的讨论）。

如果让所有的线程都共享一个输入队列，就会严重削弱系统的强壮性。如果某一个线程接收一个按键消息并且挂起，其他的线程就不能接收任何输入了。所以应该尽量避免使用AttachThreadInput函数。

在某些情况下，系统隐式地将两个线程挂接在一起。第一种情况是当一个线程安装一个日志记录挂钩（journal record hook）或日志播放挂钩（journal playback hook）的时候。当挂钩被卸载时，系统自动恢复所有线程，这样线程就可以使用挂钩安装前它们所使用的相同输入队列。

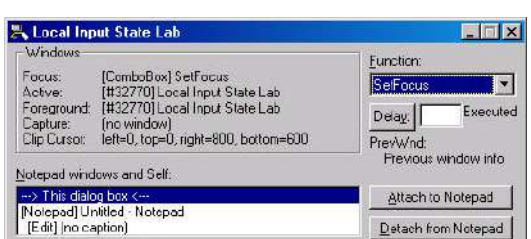
当一个线程安装一个日志记录挂钩时，它是让系统将用户输入的所有硬件事件都通知它。这个线程通常将这些信息保存或记录在一个文件上。因用户的输入必须按进入的次序来记录，所以系统中每个线程要共享一个虚拟输入队列，使所有的输入处理同步。

还有一些情况，系统会代替你隐式地调用 AttachThreadInput。假定你的程序建立了两个线程。第一个线程建立了一个对话框。在这个对话框建立之后，第二个线程调用 GreatWindow，使用WS_CHILD风格，并向这个子窗口的双亲传递对话框的句柄。系统用子窗口的线程调用AttachThreadInput，让子窗口的线程使用对话框线程所使用的输入队列。这样就使对话框的所有子窗口之间对输入强制同步。

27.3.1 LISLab 示例程序

LISLab程序（“27LISLab.exe”）清单列在清单27-1上。这是一个实验室，可以用它来实验局部输入状态。这个程序的源代码和资源文件在本书所附光盘的27-LISLab目录下。

为了用局部输入状态做实验，需要两个线程作为实验品。LISLab进程有一个线程，这里选择Notepad的线程作为另一个。如果当LISLab启动时Notepad没有在运行，LISLab将启动Notepad。在LISLab初始化之后，就会见到图27-4所示的对话框。



China-pub.com
下载


```
Module:  LISLab.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/
#define TIMER_DELAY (500)  // Half a second

UINT_PTR g_uTimerId = 1;
int      g_nEventId = 0;
DWORD    g_dwEventTime = 0;
HWND     g_hwndSubject = NULL;
HWND     g_hwndNotepad = NULL;


//////////////////////////////////////////////////////////////////////////////


void CalcWndText(HWND hwnd, PTSTR szBuf, int nLen) {

   TCHAR szClass[50], szCaption[50], szBufT[150];

   if (hwnd == (HWND) NULL) {
      _tcscpy(szBuf, TEXT("(no window)"));
      return;
   }


   if (!IsWindow(hwnd)) {
      _tcscpy(szBuf, TEXT("(invalid window)"));
      return;
   }

   GetClassName(hwnd, szClass, chDIMOF(szClass));
   GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

   wsprintf(szBufT, TEXT("[%s] %s"), (PTSTR) szClass,
      (*szCaption == 0) ? (PTSTR) TEXT("(no caption)") : (PTSTR) szCaption);
   _tcsncpy(szBuf, szBufT, nLen - 1);
   szBuf[nLen - 1] = 0; // Force zero-terminated string
}
```

China-pub.com 下载

```
void WalkWindowTreeRecurse(PWALKWINDOWTREEDATA pWWT) {

   if (!IsWindow(pWWT->hwndParent))
      return;

   pWWT->nLevel++;
   const int nIndexAmount = 2;

   for (pWWT->iBuf = 0; pWWT->iBuf < pWWT->nLevel * nIndexAmount; pWWT->iBuf++)
      pWWT->szBuf[pWWT->iBuf] = TEXT(' ');

   CalcWndText(pWWT->hwndParent, &pWWT->szBuf[pWWT->iBuf],
      chDIMOF(pWWT->szBuf) - pWWT->iBuf);
   pWWT->nIndex = ListBox_AddString(pWWT->hwndLB, pWWT->szBuf);
   ListBox_SetItemData(pWWT->hwndLB, pWWT->nIndex, pWWT->hwndParent);

   HWND hwndChild = GetFirstChild(pWWT->hwndParent);
   while (hwndChild != NULL) {
      pWWT->hwndParent = hwndChild;
      WalkWindowTreeRecurse(pWWT);
      hwndChild = GetNextSibling(hwndChild);
   }

   pWWT->nLevel--;
}


//////////////////////////////////////////////////////////////////////////////


void WalkWindowTree(HWND hwndLB, HWND hwndParent) {

   WALKWINDOWTREEDATA WWT;
   WWT.hwndLB = hwndLB;
   WWT.hwndParent = hwndParent;
   WWT.nLevel = -1;

   WalkWindowTreeRecurse(&WWT);
}
```

China-pub.com
下载


```
ComboBox_AddString(hwndT, TEXT("SetFocus"));
ComboBox_AddString(hwndT, TEXT("SetActiveWindow"));
ComboBox_AddString(hwndT, TEXT("SetForegroundWnd"));
ComboBox_AddString(hwndT, TEXT("BringWindowToTop"));
ComboBox_AddString(hwndT, TEXT("SetWindowPos-TOP"));
ComboBox_AddString(hwndT, TEXT("SetWindowPos-BTM"));
ComboBox_SetCurSel(hwndT, 0);

// Fill the windows list box with our window
hwndT = GetDlgItem(hwnd, IDC_WNDS);
ListBox_AddString(hwndT, TEXT("---> This dialog box <---"));

ListBox_SetItemData(hwndT, 0, hwnd);
ListBox_SetCurSel(hwndT, 0);

// Now add Notepad's windows
g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
if (g_hwndNotepad == NULL) {

    // Notepad isn't running; run it.
    STARTUPINFO si = { sizeof(si) };
      PROCESS_INFORMATION pi;
      TCHAR szCommandLine[] = TEXT("Notepad");
      if (CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE, 0,
         NULL, NULL, &si, &pi)) {

         // Wait for Notepad to create all its windows.
         WaitForInputIdle(pi.hProcess, INFINITE);
         CloseHandle(pi.hProcess);
         CloseHandle(pi.hThread);
         g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
      }
   }
   WalkWindowTree(hwndT, g_hwndNotepad);

   return(TRUE);
}
```

China-pub.com 下载

```
      g_dwEventTime = GetTickCount() + 1000 *
         GetDlgItemInt(hwnd, IDC_DELAY, NULL, FALSE);
      hwndT = GetDlgItem(hwnd, IDC_WNDS);
      g_hwndSubject = (HWND)
         ListBox_GetItemData(hwndT, ListBox_GetCurSel(hwndT));
      g_nEventId = ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_WNDFUNC));
      SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Pending"));
      break;
   case IDC_THREADATTACH:
      AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
         GetCurrentThreadId(), TRUE);
      break;

   case IDC_THREADDETACH:
      AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
         GetCurrentThreadId(), FALSE);
      break;

   case IDC_REMOVECLIPRECT:
      ClipCursor(NULL);
      break;

   case IDC_HIDECURSOR:
      ShowCursor(FALSE);
      break;

   case IDC_SHOWCURSOR:
      ShowCursor(TRUE);
      break;

   case IDC_INFINITELOOP:
      SetCursor(LoadCursor(NULL, IDC_NO));
      for (;;)
          ;
      break;

   case IDC_SETCLIPRECT:
      RECT rc;
      SetRect(&rc, 0, 0, GetSystemMetrics(SM_CXSCREEN) / 2
```

China-pub.com
下载


```
         ListBox_DeleteString(hwndLB, 0);
   } while (nIndex == LB_ERR);

   ListBox_SetCurSel(hwndLB, nIndex);
}

//////////////////////////////////////////////////////////////////////////////

int Dlg_OnRButtonDown(HWND hwnd, BOOL fDoubleClick,
   int x, int y, UINT keyFlags) {

   TCHAR szBuf[100];
   wsprintf(szBuf,
      TEXT("Capture=%-3s, Msg=RButtonDown, DblClk=%-3s, x=%5d, y=%5d"),
      (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"),
      fDoubleClick ? TEXT("Yes") : TEXT("No"), x, y);

   AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGS), szBuf);
   if (!fDoubleClick) SetCapture(hwnd);
   else ReleaseCapture();
   return(0);
}

//////////////////////////////////////////////////////////////////////////////

int Dlg_OnRButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

   TCHAR szBuf[100];
   wsprintf(szBuf, TEXT("Capture=%-3s, Msg=RButtonUp,   x=%5d, y=%5d"),
      (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

   AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGS), szBuf);
   return(0);
}
```

China-pub.com 下载

```
void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

   TCHAR szBuf[100];
   wsprintf(szBuf,
      TEXT("Capture=%-3s, Msg=LButtonUp,   x=%5d, y=%5d"),
      (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

   AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGS), szBuf);
}


//////////////////////////////////////////////////////////////////////////////

void Dlg_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags) {

   TCHAR szBuf[100];
   wsprintf(szBuf, TEXT("Capture=%-3s, Msg=MouseMove,  x=%5d, y=%5d"),
      (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

   AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGS), szBuf);
}


//////////////////////////////////////////////////////////////////////////////
void Dlg_OnTimer(HWND hwnd, UINT id) {

   TCHAR szBuf[100];

   CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
   SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);

   CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
   SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);

   CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
   SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);
```

China-pub.com
下载


```
            break;

         case 1:  // SetActiveWindow
            g_hwndSubject = SetActiveWindow(g_hwndSubject);
            break;

         case 2:  // SetForegroundWindow
            hwndT = GetForegroundWindow();
            SetForegroundWindow(g_hwndSubject);
            g_hwndSubject = hwndT;
            break;

         case 3:  // BringWindowToTop
            BringWindowToTop(g_hwndSubject);
            break;

         case 4:  // SetWindowPos w/HWND_TOP
            SetWindowPos(g_hwndSubject, HWND_TOP, 0, 0, 0, 0,
               SWP_NOMOVE | SWP_NOSIZE);
            g_hwndSubject = (HWND) 1;
            break;

         case 5:  // SetWindowPos w/HWND_BOTTOM
            SetWindowPos(g_hwndSubject, HWND_BOTTOM, 0, 0, 0, 0,
               SWP_NOMOVE | SWP_NOSIZE);
            g_hwndSubject = (HWND) 1;
            break;
      }

      if (g_hwndSubject == (HWND) 1) {
         SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), TEXT("Can't tell."));
      } else {
         CalcWndText(g_hwndSubject, szBuf, chDIMOF(szBuf));
         SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), szBuf);
      }

      g_hwndSubject = NULL; g_nEventId = 0; g_dwEventTime = 0;
      SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Executed"));
   }
```

China-pub.com 下载

```
      chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN,   Dlg_OnRButtonDown);
      chHANDLE_DLGMSG(hwnd, WM_RBUTTONDBLCLK, Dlg_OnRButtonDown);
      chHANDLE_DLGMSG(hwnd, WM_RBUTTONUP,     Dlg_OnRButtonUp);

      chHANDLE_DLGMSG(hwnd, WM_TIMER,         Dlg_OnTimer);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISLAB), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

LISLab.rc

```
//Microsoft Developer Studio generated-resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
```

China-pub.com
下载


```
// remains consistent on all systems.
IDI_LISLAB              ICON    DISCARDABLE     "LISLab.Ico"

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_LISLAB DIALOG DISCARDABLE  12, 38, 286, 178
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Local Input State Lab"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "Windows",IDC_STATIC,4,0,192,56
    LTEXT           "Focus:",IDC_STATIC,8,12,23,8
    LTEXT           "Focus window info",IDC_WNDFOCUS,52,12,140,8
    LTEXT           "Active:",IDC_STATIC,8,20,24,8
    LTEXT           "Active window info",IDC_WNDACTIVE,52,20,140,8
    LTEXT           "Foreground:",IDC_STATIC,8,28,40,8
    LTEXT           "Foreground window info",IDC_WNDFOREGROUND,52,28,140,8
    LTEXT           "Capture:",IDC_STATIC,8,36,29,8
    LTEXT           "Capture window info",IDC_WNDCAPTURE,52,36,140,8
    LTEXT           "Clip Cursor:",IDC_STATIC,8,44,39,8
    LTEXT           "Cursor clipping info",IDC_CLIPCURSOR,52,44,140,8
    LTEXT           "&Function:",IDC_STATIC,200,4,32,8
    COMBOBOX        IDC_WNDFUNC,200,14,82,54,CBS_DROPDOWNLIST | WS_VSCROLL |
                    WS_TABSTOP
    PUSHBUTTON      "Dela&y:",IDC_FUNCSTART,200,30,26,14
    EDITTEXT        IDC_DELAY,228,30,24,12,ES_AUTOHSCROLL
    LTEXT           "Executed",IDC_EVENTPENDING,252,30,32,10
    LTEXT           "PrevWnd:",IDC_STATIC,200,46,34,8
    LTEXT           "Previous window info",IDC_PREVWND,208,54,76,18
    LTEXT           "&Notepad windows and Self:",IDC_STATIC,4,62,90,8
    LISTBOX         IDC_WNDS,4,72,192,32,WS_VSCROLL | WS_TABSTOP
    PUSHBUTTON      "&Attach to Notepad",IDC_THREADATTACH,200,72,80,12
    PUSHBUTTON      "&Detach from Notepad",IDC_THREADDETACH,200,88,80,12
    LTEXT           "&Mouse messages received:",IDC_STATIC,4,102,89,8
    LISTBOX         IDC_MOUSEMSGS,4,112,192,32,WS_VSCROLL | WS_TABSTOP
    LTEXT           "Click right mouse button to set capture \nor Double click
```

China-pub.com 下载

```
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_LISLAB, DIALOG
    BEGIN
        RIGHTMARGIN, 283
        BOTTOMMARGIN, 170
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
```

China-pub.com
下载

有窗口拥有焦点并且没有窗口是活动的。

然后可以通过改变窗口的焦点来进行实验。首先在 Local Input State Lab对话框右上角的Function组合框内选择SetFocus。然后键入延迟时间（以秒计），即在调用SetFocus之前你想让LISLab等待的时间。对这个实验，你很可能会指定延迟为 0s。后面将简单介绍如何使用 Delay字段。

下一步选择一个窗口，作为调用SetFocus时的参数。用Local Input State Lab对话框左边的Notepad Windows And Self列表框选择一个窗口。对这个实验，选择列表框中的 [Notepad] Untitled-Notepad。现在已经为调用SetFocus做好准备。只需点击Delay按钮，观察Windows编组框会发生什么变化。什么也没发生。系统没有执行改变焦点的动作。

如果真想让SetFocus将焦点改变到Notepad，就点击Attach To Notepad按钮。点击这个按钮使LISLab调用下面的函数：

这个调用告诉LISLab的线程去使用Note- pad所使用的虚拟输入队列。另外， LISLab的线程也要与Notepad共享局部输入状态变量。

```
AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
   GetCurrentThreadId(), TRUE);
```

如果在点击了Attach To Notepad按钮之后，点击Notepad窗口，LISLab的对话框变成图27-5所示的样子。

现在注意，由于两个线程的输入队列是挂接在一起的， LISLab可以服从Notepad所做的窗口焦点改变。图 27-5所示的对话框显示Edit控制框当前具有焦点。如果我们显示Notepad中的File Open对话框，LISLab将继续更新它的显示屏内容，告诉我们哪一个Note-pad窗口具有焦点，哪个窗口是活动的等等。

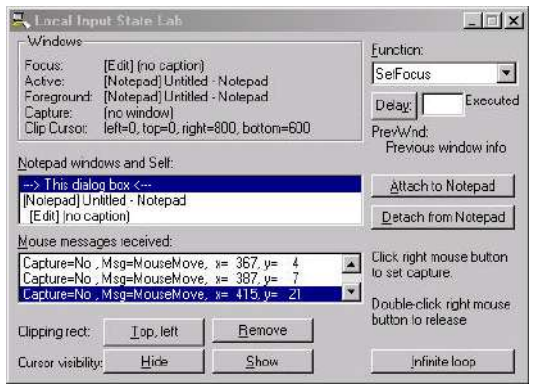


图27-5 点击Notepad窗口后的LISLab对话框

现在我们再回到 LISLab，点击 Delay按钮，让 SetFocus给Notepad焦点。这一次，对SetFocus的调用成功，因两个线程的输入队列是接在一起的。

读者可以继续实验，通过在 Function组合框中选择不同的函数，分别对 SetActiveWindow,

China-pub.com 下载

不过，RIT仍然同Notepad的线程相“连接”。

关于窗口和焦点还要说明一点：SetFocus函数和SetActiveWindow函数都返回原来拥有焦点或原来活动的窗口的句柄。有关这个窗口的信息显示在LISLab对话框的PrevWnd字段里。而且，LISLab在调用SetForegroundWindow之前，要先调用GetForegroundWindow来取得原来处于前景的窗口的句柄。这些信息也显示在PrevWnd字段。

现在我们对鼠标光标的内容进行实验。每当你在LISLab的对话框上移动鼠标（但没有在它的任何子窗口上移动），鼠标被显示成一个垂直箭头。当鼠标消息发送到这个对话框，消息要添加到Mouse Message Received列表框中。这样你就可以知道何时对话框在接收鼠标消息。如果你将鼠标移出对话框或移到某个子窗口之上，就会发现鼠标消息不再添加到 Mouse Message Received列表框中。

现在将鼠标移往对话框的右部，移在文本Click Right Mouse Button To Set Capture之上，然后点击并按住鼠标右键。这时， LISLab调用SetCapture并传递LISLab对话框的句柄作为参数。注意LISLab更新Windows编组框来反映它拥有鼠标捕获。

不要释放鼠标右键，在LISLab的子窗口上移动鼠标，并观察鼠标消息被添加到列表框里。注意，如果你将鼠标移出LISLab的对话框， LISLab会继续得知鼠标消息。不论你在屏幕上什么位置移动鼠标，鼠标光标都保持垂直箭头形状。

现在我们看一看系统的其他表现。释放鼠标右键，看会发生什么。在 LISLab对话框的上部所反映的捕获窗口继续显示 LISLab依然认为自己拥有鼠标捕获。如果你将鼠标移出 LISLab的对话框，鼠标光标就不再保持垂直箭头的形状，鼠标消息也不再向 Mouse Mssages Received列表框中添加，你会看到鼠标捕获依然有效，因为所有窗口都使用同一组局部输入状态变量。

当完成对鼠标捕获的实验时，可以使用下面两种办法将其关闭：

- 在Local Input State Lab对话框的任何地方双击鼠标右键，让LISLab安排ReleaseCapture的调用。
- 点击一个由LISLab的线程之外的线程所建立的窗口。这样系统会自动向 LISLab的对话框发送鼠标按钮弹起和鼠标按钮按下的消息。

不论使用哪种办法，要观察Windows编组框中的Capture字段是如何变化以反映没有窗口拥有鼠标捕获。

与鼠标有关的实验还有两个：其中之一涉及在一个矩形区域剪贴鼠标光标的移动，另一个涉及鼠标光标的可见性。当点击Top、Left按钮时，LISLab执行下面的代码：

China-pub.com
下载

点击Hide或Show Cursor按钮，使LISLab执行下面的代码：

```
ShowCursor(FALSE);
```

或

```
ShowCursor(TRUE);
```

当你隐藏了鼠标光标时，如果在LISLab的对话框上移动鼠标时，不会出现鼠标光标。但在这个对话框之外移动鼠标时，鼠标光标又会出现。使用 Show按钮来抵消Hide按钮的效果。注意隐藏光标的效果是要积累的，也就是，如果 5次点击Hide按钮，必须也5次点击Show按钮，才能使光标可见。

最后一个实验是使用Infinite Loop按钮。当点击这个按钮时，LISLab执行下面的代码：

```
SetCursor(LoadCursor(NULL, IDC_NO));
for (;;)
   ;
```

第一行代码将鼠标光标改变成一个缺口圆（slashed circle），第二行代码执行一个死循环。在点击Infinite Loop按钮之后，LISLab停止响应任何输入。如果在LISLab的对话框上移动鼠标，鼠标光标依然是缺口圆。如果将鼠标移出对话框，光标要改变，以反映它所处窗口的光标。可以用鼠标操作这些其他的窗口。

如果将鼠标移回到LISLab的对话框，系统看到LISLab没有响应，就自动将光标改回最近的形状——缺口圆。可以看到执行死循环的线程对用户是不方便的，但可以因此使用其他窗口。

注意，如果把一个窗口移到挂起的Local Input State Lab对话框，然后再把它移开，系统要发送一个WM_PAINT消息。但系统发现这个线程没有响应。系统为这个没有响应的程序重画窗口。当然，系统不能正确地重画这个窗口，因为系统不知道这个程序是干什么的。所以系统只是抹掉窗口的背景，并重画框架。

现在还有一个问题，如果屏幕上有一个窗口对我们做的任何事情（按键或击鼠标钮）都没有响应。我们如何清除这个窗口？在Windows 98中，必须按Ctrl+Alt+Del来显示图27-6的Close Program窗口。

在Windows 2000里，可以用鼠标右击Taskbar上的程序按钮，或显示图27-7的TaskManager窗口。

然后只要在窗口里选择我们想要结束的程序，在这里是 Local Input State Lab，再点击End

China-pub.com 下载

Task按钮。系统将试图用一种温和的方式（通过发送一个 WM_CLOSE消息）来结束LISLab，但发现该程序没有反应。在Windows 98里，这会引起系统显示图27-8的对话框。

在Windows 2000中会显示图27-9的对话框。



图27-8 Local Input State Lab 对话框

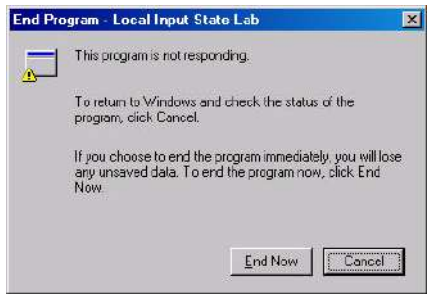


图27-9 End Program 对话框

选择End Task（在Windows 98里）或End Now（在Windows 2000里）使系统强制性地将LISLab从系统中清除。Cancel按钮是告诉系统你改变了主意，不再想结束这个程序。这里，我们选择End Task或End Now，从系统中清除LISLab。

这个实验的主要目的是为了说明系统的强壮性。一个程序不可能使操作系统处于这样一个状态——使其他程序不可用。还要注意在结束处理中，Windows 98和Windows 2000都可以自动释放线程所分配的资源，不会造成内存遗漏。

27.3.2 LISWatch示例程序

LISWatch程序（“27LISWatch.exe”）的源程序清单列在清单27-2中。这是一个有用的实用程序，用来监控活动窗口、焦点窗口和鼠标捕获窗口。在本书所附光盘的27-LISWatch目录下是这个程序的源代码和资源文件。

当运行LISWatch，会显示图27-10的对话框。



图27-10 运行LISWatch时显示的对话框

当这个对话框接收到一个WM_INITDIALOG消息时，它调用SetTimer来设置一个计时器，每秒钟激发两次。当收到WM_TIMER消息，对话框的内容就更新以反映哪个窗口是活动的、哪个窗口拥有焦点、哪个窗口捕获了鼠标。在这个对话框

China-pub.com
下载

以调用GetFocus、GetActiveWindow和GetCapture，所有这些函数都返回有效的窗口句柄。帮助函数CalcWndText构造一个字符串，包含每个窗口的类名和窗口标题。然后每个窗口的串在LISWatch的对话框中被更新。最后，Dlg_OnTimer在返回之前，再一次调用AttachThreadInput，但这次是将最后一个参数设定为FALSE，这样两个线程的局部输入状态就彼此断开。

前面解释了LISWatch的基本内容。然而，我们对LISWatch增加了其他一些特性，在这里解释一下。当启动LISWatch之后，它要监控系统中任何地方发生的窗口活动的变化。这就是对话框顶部的“全系统范围（System-wide）”的意思。LISWatch还可以让你只限于观察一个线程的局部输入状态的变化。利用这种特性，LISWatch可以向你报告一个线程的确切情况。

为了让LISWatch监控一个线程的局部输入状态，所要做的只是在LISWatch的窗口上按下鼠标左键，在另外一个线程建立的窗口上拖动鼠标光标，然后释放鼠标按钮。在释放鼠标按钮之后，LISWatch将全局变量g-dwThreadIdAttachTo设置成所选线程的ID。这个线程ID将替换LISWatch的对话框顶部的“System-wide”。当这个全局变量不是零，Dlg_OnTimer会略微改变它的行为。不是总将它的局部输入状态同前景线程的局部输入挂接在一起，而是将LISWatch本身同所选择的线程挂接在一起。用这种方式，LISWatch调用GetActiveWindow、GetFocus和GetCapture来反映所选择线程的局部输入状态的情况。

我们来做一个实验。运行Calculator，再用LISWatch来选择它的窗口。当激活Calculator的窗口时，LISWatch更新它的显示内容，见图27-11的对话框。

这里，Calculator的线程ID是0x000004ec。当前设定LISWatch来监控这一个线程的局部输入状态变化。如果点击Calculator的任何单选按钮或复选框，LISWatch都可以显示焦点的变化，因为所有这些窗口都是由线程0x000004ec建立的。

如果现在再激活一个由另外的程序（这里的例子是Notepad）建立的窗口，LISWatch的对话框是下面的样子（见图27-12）。



图27-11 激活Calculator窗口时的LISWatch对话框

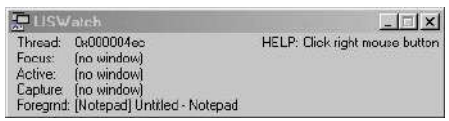


图27-12 由另一个程序激活窗口时显示的LISWatch对话框

China-pub.com 下载

```
#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <windowsx.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


#define TIMER_DELAY (500)        // Half a second

UINT_PTR g_uTimerId = 1;
DWORD g_dwThreadIdAttachTo = 0;  // 0=System-wide; Non-zero=specifc thread


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_LISWATCH);

   // Update our contents periodically
   g_uTimerId = SetTimer(hwnd, g_uTimerId, TIMER_DELAY, NULL);

   // Make our window on top of all others
   SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnRButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y,
   UINT keyFlags) {
   chMB("To monitor a specific thread, click the left mouse button in "
        "the main window and release it in the desired window.\n"
```

China-pub.com
下载


```
   SetCursor(LoadCursor(GetModuleHandle(NULL), MAKEINTRESOURCE(IDC_EYES)));
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

   if (GetCapture() == hwnd) {

      // If we had mouse capture set, get the ID of the thread that
      // created the window that is under the mouse cursor.
      POINT pt;
      pt.x = LOWORD(GetMessagePos());
      pt.y = HIWORD(GetMessagePos());
      ReleaseCapture();
      g_dwThreadIdAttachTo = GetWindowThreadProcessId(
         ChildWindowFromPointEx(GetDesktopWindow(), pt, CWP_SKIPINVISIBLE),
         NULL);

      if (g_dwThreadIdAttachTo == GetCurrentThreadId()) {
          // The mouse button is released on one of our windows;
          // monitor local-input state on a system-wide basis.
          g_dwThreadIdAttachTo = 0;

       } else {

          // The mouse button is released on a window that our thread didn't
          // create; monitor local input state for that thread only.
          AttachThreadInput(GetCurrentThreadId(), g_dwThreadIdAttachTo, TRUE);
       }
    }
 }

 //////////////////////////////////////////////////////////////////////////////
```

China-pub.com 下载

```
   _tcsncpy(szBuf, szBufT, nLen - 1);
   szBuf[nLen - 1] = 0; // Force zero-terminated string
}


//////////////////////////////////////////////////////////////////


void Dlg_OnTimer(HWND hwnd, UINT id) {
TCHAR szBuf[100] = TEXT("System-wide");
HWND hwndForeground = GetForegroundWindow();
DWORD dwThreadIdAttachTo = g_dwThreadIdAttachTo;

if (dwThreadIdAttachTo == 0) {

   // If monitoring local input state system-wide, attach our input
   // state to the thread that created the current foreground window.
   dwThreadIdAttachTo =
      GetWindowThreadProcessId(hwndForeground, NULL);
   AttachThreadInput(GetCurrentThreadId(), dwThreadIdAttachTo, TRUE);

} else {

   wsprintf(szBuf, TEXT("0x%08x"), dwThreadIdAttachTo);
}

SetWindowText(GetDlgItem(hwnd, IDC_THREADID), szBuf);

CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);

CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);

CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);

CalcWndText(hwndForeground, szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOREGRND), szBuf);
```

China-pub.com
下载


```
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG,  Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,     Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_TIMER,       Dlg_OnTimer);
      chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN, Dlg_OnRButtonDown);
      chHANDLE_DLGMSG(hwnd, WM_LBUTTONDOWN, Dlg_OnLButtonDown);
      chHANDLE_DLGMSG(hwnd, WM_LBUTTONUP,   Dlg_OnLButtonUp);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISWATCH), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

LISWatch.rc

```
//Microsoft Developer Studio generated resource script.
//
```

China-pub.com 下载

```
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_LISWATCH DIALOG DISCARDABLE  32768, 5, 240, 41
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "LISWatch"
FONT 8, "MS Sans Serif"
```

China-pub.com
下载


```
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_LISWATCH            ICON    DISCARDABLE     "LISWatch.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Cursor
//

IDC_EYES                CURSOR  DISCARDABLE     "Eyes.cur"

/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_LISWATCH, DIALOG
    BEGIN
        RIGHTMARGIN, 190
        BOTTOMMARGIN, 10
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
```

China-pub.com
下载

第七部分 附 录

附录A 建立环境

读者要想建立本书中的示例程序，必须要对编译程序和链接程序的开关选项进行设置。笔者试图将这些设置方面的细节从示例程序中隔离出来，把所有这些设置放在一个头文件里。这个头文件就是CmmHdr.h，它包含在所有示例程序的源代码文件中。

因为无法将所有的设置都放在这个头文件里，我们对每个示例程序的项目设置做了一些改变。对每个项目，我们显示Project Settings对话框，然后做下面所说的改变。

- 在General栏，设定Output Files目录，这样所有最终的.exe和.dll文件都在一个目录之下。
- 在C/C++栏，选择 Code Generation条目，并对 Use Run-Time Library字段选择 Multithreaded DLL。

这样就可以了。我只明确改变了两个设置，而接受了其他所有的默认设置。注意要对每个项目的Debug建立和Release建立都做上述两个改变。我可以在源代码中设定所有其他的编译程序和链接程序的设置，当你在你的项目中使用这里的任何源代码模块时，这些设置都将起作用。

A.1 CmmHdr.h头文件

所有的示例程序都要包含CmmHdr.h头文件，并且要在其他头文件之前包含。笔者编写的CmmHdr.h列在清单A-1里。这个文件给笔者带来不少便利。这个文件包含宏、链接程序指令、还有一些其他所有示例程序公用的内容。当我想做某些实验时，我只需修改并重建（rebuild）所有的示例程序。CmmHdr.h在所附光盘的根目录下。

这个附录的其余部分将分别讨论CmmHdr.h文件的每一节，解释每一节的基本原理，并描述在重建所有示例程序之前，如何及为什么要对这个文件进行修改。

A.1.1 Windows版本建立选项

因为有些示例程序调用了Microsoft Windows 2000中提供的新函数，本节定义_WIN32_WINNT符号如下：

```
#define _WIN32_WINNT 0x0500
```

这样做是因为新的Windows 2000函数在Windows头文件中被定义成下面这样的原型：

```
#if (_WIN32_WINNT >= 0x0500)
.
.
.
WINBASEAPI
BOOL
WINAPI
AssignProcessToJobObject(
    IN HANDLE hJob,
    IN HANDLE hProcess
    );
```

China-pub.com 下载

```
   :
#endif /* _WIN32_WINNT >= 0x0500 */
```

除非像我这样专门定义_WIN32_WINNT（在包含Windows.h之前），否则这些新函数的原型就没有被声明，当试图调用这些函数时，编译程序将产生错误。微软用 _WIN32_WINNT符号来保护这些函数，以使程序员开发的应用程序能够运行在 Windows 98及Windows NT的多个版本上。

A.1.2 Unicode建立选项

笔者编写的所有这些示例程序既可按ANSI来编译，也可按Unicode来编译。当针对x86CPU体系结构来编译这些程序时， ANSI为默认选择，这样程序可以在 Windows 98上执行。但对其他CPU体系结构建立程序就要用Unicode，这样程序可以占用较少的内存，并且执行得更快。

为了对x86体系结构建立Unicode版本，只需将定义UNICODE的那一行代码的注释符去掉，并重建程序。通过在CmmHdr.h定义UNICODE宏，可以很容易地控制如何建立示例程序。关于Unicode的详细内容，可参见第2章。

A.1.3 窗口定义和第4级警告

笔者在开发软件时，总是想保证代码的编译不受错误和警告的限制。我还喜欢在可能最高警告级上进行编译，这样编译程序可以替我做大多数工作，甚至为我检查很小的细节。对于Microsoft C/C++编译程序，这将意味着我要使用第4级警告来建立示例程序。

遗憾的是，微软的操作系统开发部在关于使用第4级警告做编译方面，与我没有共同的思想。其结果，当我使用第4级警告编译示例程序时，Windows头文件中的许多行引起编译器产生警告。幸好，这些警告并不表示代码中有问题。大多数情况是由于 C语言中非传统的用法所引起的，这些用法依赖编译程序的扩展，几乎所有与Windows兼容的编译程序厂商都实现了这些扩展。

本节我确保警告级设定为 3，而且CmmHdr.h包含标准的 Windows.h头文件。当包含了Windows.h时，在我编译其余代码时就设置第 4级警告。在第4级警告上，编译程序对那些我不认为有问题的内容发出“ 警告 ”，这样我通过使用#pragma warning指令显式地告诉编译程序忽略某些良性的警告错。

A.1.4 Pragma消息帮助宏

在我编写代码时，我喜欢让代码的某些部分能够立即运行起来，然后再完善它。为了提醒自己要特别注意某些代码，我习惯于加入下面这样一行代码：

```
#pragma message("Fix this later")
```

当编译程序对这一行进行编译时，它会输出一个字符串提醒我还需要再做一些工作。但这条消息不怎么有用。我决定寻找一种办法，让编译程序输出源代码文件的名字，以及 pragma出现的行号。这样，我不光知道要做一些工作，而且能够立刻确定在什么地方做。

为了达到这个目的，需要使用一系列宏来修饰pragma message指令。可以这样使用chMSG宏。

```
#pragma chMSG(Fix this later)
```

当编译程序编译上面这一行代码时，会产生这样一行内容：

使用Microsoft Visual Developer Studio，在输出窗口上双击这一行，将会自动定位到相应文件的确切位置上。

```
C:\CD\CmnHdr.h(82):Fix this later
```

China-pub.com
下载

还有一个方便之处，chMSG宏不要求对文本串使用引号。

A.1.5 chINRANGE和chDIMOF宏

我时常在编写程序时使用这两个方便有用的宏。第一个宏 chINRANGE，用来查看一个数值是否在另外两个数值之间。第二个宏 chDIMOF，只是返回一个数组中元素的数目。这个宏是用sizeof操作符先计算整个数组的字节数，然后再用这个数除以数组中一个数据项所占的字节数，从而得出结果。

A.1.6 chBEGINTHREADEX宏

本书中的所有多线程示例程序都使用了微软的 C／C++运行时函数库中的_beginthreadex函数，而不是操作系统的CreateThread函数。我使用这个函数是因为 _beginthreadex函数为新线程做好了准备，使新线程能够使用 C/C++运行时函数库中的函数，而且还因为它保证在线程返回时清除每个线程的C/C++运行时库信息（见第6章有关细节）。但遗憾的是_beginthreadex函数的原型是这样的。

tunately, the *_beginthreadex* function is protc

```
unsigned long __cdecl _beginthreadex(
   void *,
   unsigned,
   unsigned (__stdcall *)(void *),
   void *,
   unsigned,
   unsigned *);
```

尽管_beginthreadex函数用的参数值同CreateThread函数用的参数值是一样的，但二者的参数的数据类型都不相匹配。CreateThread函数的原型是这样的：

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(PVOID pvParam);

HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   DWORD cbStack,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD fdwCreate,
   PDWORD pdwThreadId);
```

微软在建立_beginthreadex函数的原型时没有使用Windows数据类型。这是因为微软的C/C++运行时库开发组不想对操作系统开发组有任何依赖。这使得_beginthreadex函数的使用更加困难。

微软定义 _beginthreadex函数原型的方式实际上存在着两个问题。首先，用于这个函数的一些数据类型同用于 CreateThread函数的原始类型不相匹配。例如 Windows数据类型 DWORD的定义是这样的：

```
typedef unsigned long DWORD;
```

这个数据类型用于 CreateThread函数的 cbStack参数以及 fdwCreate参数。问题是函数 _beginthreadex将这两个参数的原型定义为 unsigned，实际意思是 unsigned int。编译程序将 unsigned int看成是与unsigned long不同的东西，并且产生一个警告。_beginthreadex函数不属于标准的C/C++运行时函数库，只是作为调用 CreateThread函数的替代手段而存在，所以微软应该按下面的形式来定义_beginthreadex的原型，这样就不会产生警告了：

```
unsigned long __cdecl _beginthreadex(
```

China-pub.com 下载

```
    void *psa,
    unsigned long cbStack,
    unsigned (__stdcall *) (void *pvParam),
    void *pvParam,
    unsigned long fdwCreate,
    unsigned long *pdwThreadId);
```

第二个问题是第一个问题的一个小变种。_beginthreadex函数返回一个unsigned long型的值，代表新建立线程的句柄。程序中通常用HANDLE型数据变量来保存这个返回值：

```
HANDLE hThread = _beginthreadex(...);
```

上面这行代码又使编译程序产生另一个警告错。为了避免编译程序警告，必须改写这一行代码，引入一个转换（cast）：

```
HANDLE hThread = (HANDLE) _beginthreadex(...);
```

这又是一个不方便之处。为了方便起见，我在 CmnHdr.h中定义了一个 chBEGINTHREADEX宏，替我执行所有这些转换：

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
   pvParam, fdwCreate, pdwThreadId)                 \
      ((HANDLE)_beginthreadex(                      \
         (void *)        (psa),                     \
         (unsigned)      (cbStack),                 \
         (PTHREAD_START) (pfnStartAddr),            \
         (void *)        (pvParam),                 \
         (unsigned)      (fdwCreate),               \
         (unsigned *)    (pdwThreadId)))
```

A.1.7 对x86平台的调试断点改进

即使进程没有在一个调试程序下运行，有时候我也想在我的程序代码中强制一个断点。在Windows中要做这件事，可以让线程调用DebugBreak函数。这个函数在kernel32.dll中，可以使一个调试程序同进程挂接。当调试程序被挂接上时，指令指针就定位在引起断点的CPU指令上。这个指令包含在 kernel32.dll中的DebugBreak函数里，所以为了看到我的源代码，我必须在DebugBreak函数之外单步执行。

在x86体系结构上，通过执行“ int3 ”CPU指令来做一个断点。所以，在x86平台之上，我定义DebugBreak作为这个内联的汇编语言指令。当我的DebugBreak执行时，我不是在kernel32.dll中调用。断点发生在我的代码中，指令指针定位在下一个C/C++语句中。这样就方便多了。

A.1.8 建立软件异常代码

当处理软件异常时，必须建立你自己的32位异常代码。这些代码遵循特定的格式（见第24章的讨论）。为了更容易地建立这些代码，我使用MAKESOFTWAREEXCEPTION宏。

A.1.9 chMB宏

chMB宏只是显示一个消息框。消息框的标题是调用进程可执行代码的全路径名。

A.1.10 chASSERT和chVERIFY宏

在我开发这些示例程序时，为了查找潜在的问题，我在整个代码中多处使用chASSERT宏。这

China-pub.com
下载

个宏测试由x所标识的表达式是否为TRUE，如果不是，则显示一个消息框指出失败的文件、行和表达式。在程序的发行建立中，这个宏什么也不做。chVERIFY宏与chASSERT宏差不多，区别在于不论是调试建立（debug build）还是发行建立(release build)，chVERIFY都要对表达式进行测试。

A.1.11 chHANDLE_DLGMSG宏

当你通过对话框使用消息分流器时，不应该使用微软的 WindowsX.h头文件中的HANDLE_MSG宏，因为这个宏并不能返回TRUE或FALSE来指出消息是否由对话框的过程来处理。我定义的chHANDLE_DLGMSG宏会通知窗口消息的返回值，适当地处理返回值，以便在一个对话框过程中使用。

A.1.12 chSETDLGICONS宏

由于多数示例程序使用一个对话框作为主窗口，你必须手工改变对话框图标，以便让它正确地显示在Taskbar（任务条）、任务切换窗口和程序本身的标题上。当对话框接收到一个WM_INITDIALOG消息时，总要调用chSETDLGICONS宏，以正确设置图标。

A.1.13 OS版本检查内联函数

本书的大多数示例程序可运行在所有平台上，但也有一些程序要求一些 Windows 95和Windows 98所不支持的特性，有些程序要求一些只在Windows 2000中提供的特性。每个程序在初始化时要检查宿主系统的版本，如果要求更适用的操作系统时，就显示一个通知。

对那些不能在Windows 95和Windows 98上运行的程序，你会看到，在程序的_tWinMain函数中有一个对Windows9xNotAllowed函数的调用。对于要求Windows 2000的示例程序，你会看到在程序的_tWinMain函中有一个对chWindows2000Required函数的调用。

A.1.14 确认宿主系统是否支持Unicode

Windows 98不能像Windows 2000那样完全支持Unicode。实际上，调用Unicode函数的程序不能在Windows 98上运行。但遗憾的是，如果调用一个为Unicode编译的程序，Windows98不会给出任何通知信息。对本书中的程序，这意味着这些程序从开始到结束，都不会有它们想执行的提示信息。

这确实是一个难题。我需要有一种办法能够知道我的程序是对 Unicode建立的，但可能在Windows 98系统上运行。所以我建立了一个CUnicodeSupported C++类。这个类的构造函数只是检查宿主系统是不是对Unicode有良好的支持，如果不是，就显示一个消息框，并且进程结束。

读者会看到在CmnHdr.h中，我建立了这个类的一个全局的静态实例。当我的程序启动时，C/C++运行时库启动代码调用这个对象的构造函数。如果这个构造函数检测到操作系统完全支持Unicode，构造函数返回而程序继续执行。通过建立这个类的全局实例，我不需要在每个示例程序的源代码模块中再增加特殊的代码。对于非 Unicode的程序建立，不需要声明或实例化上述的C++类。让程序只管运行就是。

A.1.15 强制链接程序寻找(w)WinMain进入点函数

本书以前版本的一些读者，将书中我的源代码模块添加到他们自己的 VisualC++项目中，但在建立项目时出现链接错误。问题的原因是他们创建了 Win32 Console Application项目，导

China-pub.com 下载

致链接程序去寻找(w)main进入点函数。因为本书中所有示例程序都是GUI程序，所以我的代码都有一个_tWinMain进入点函数。这就是链接程序为什么要报错。

我的回答是，他们应该删除原来的项目，用Visual C++建立新的Win32 Application项目(注意在项目类型中不能出现“Console”一词)，再将我的源代码加进去。链接程序寻找一个(w)WinMain进入点函数，而这在我的代码中已提供，项目应该能够建立。

为了减少我收到的有关这个问题的电子邮件的数量，我在CmnHdr.h中加入了一个pragma，强制链接程序去寻找(w)WinMain进入点函数，即使是用Visual C++建立了一个Win32 Console Application项目。

在第4章，我详细说明了Visual C++项目类型的有关内容，链接程序如何选择进入点函数，及如何重载链接程序的默认动作等。下面的清单A-1是Cmn Hdr. h 头文件。

清单A-1 CmnHdr.h头文件

CmnHdr.h

```
/******************************************************************************
Module:  CmnHdr.h
Notices: Copyright (c) 2000 Jeffrey Richter
Purpose: Common header file containing handy macros and definitions
         used throughout all the applications in the book.
         See Appendix A.
******************************************************************************/


#pragma once   // Include this header file once per compilation unit


//////////////////////// Windows Version Build Option ////////////////////////


#define _WIN32_WINNT 0x0500
//#define WINVER       0x0500


////////////////////////////// Unicode Build Option //////////////////////////////


// If we are not compiling for an x86 CPU, we always compile using Unicode.
#ifndef _M_IX86
#define UNICODE
#endif

// To compile using Unicode on the x86 CPU, uncomment the line below.
//#define UNICODE

// When using Unicode Windows functions, use Unicode C-Runtime functions too.
#ifdef UNICODE
#define _UNICODE
#endif


/////////////////////////// Include Windows Definitions ///////////////////////////
#pragma warning(push, 3)
```

China-pub.com
下载


```
#include <Windows.h>
#pragma warning(pop)
#pragma warning(push, 4)


////////////// Verify that the proper header files are being used //////////////


#ifndef WT_EXECUTEINPERSISTENTTHREAD
#pragma message("You are not using the latest Platform SDK header/library ")
#pragma message("files. This may prevent the project from building correctly.")
#endif


////////////// Allow code to compile cleanly at warning level 4 //////////////


/* nonstandard extension 'single line comment' was used */
#pragma warning(disable:4001)

// unreferenced formal parameter
#pragma warning(disable:4100)

// Note: Creating precompiled header
#pragma warning(disable:4699)

// function not inlined
#pragma warning(disable:4710)

// unreferenced inline function has been removed
#pragma warning(disable:4514)

// assignment operator could not be generated
#pragma warning(disable:4512)


////////////////////////// Pragma message helper macro //////////////////////////


/*
When the compiler sees a line like this:
   #pragma chMSG(Fix this later)
it outputs a line like this:

  c:\CD\CmnHdr.h(82):Fix this later

You can easily jump directly to this line and examine the surrounding code.
*/

#define chSTR2(x)      #x
#define chSTR(x)   chSTR2(x)
#define chMSG(desc) message(__FILE__ "(" chSTR(__LINE__) "):" #desc)


////////////////////////////// chINRANGE Macro //////////////////////////////////
```

China-pub.com 下载

```
// This macro returns TRUE if a number is between two others.
#define chINRANGE(low, Num, High) (((low) <= (Num)) && ((Num) <= (High)))


////////////////////////////////// chDIMOF Macro ////////////////////////////////


// This macro evaluates to the number of elements in an array.
#define chDIMOF(Array) (sizeof(Array) / sizeof(Array[0]))


////////////////////////////// chBEGINTHREADEX Macro ////////////////////////////


// This macro function calls the C runtime's _beginthreadex function.
// The C runtime library doesn't want to have any reliance on Windows' data
// types such as HANDLE. This means that a Windows programmer needs to cast
// values when using _beginthreadex. Since this is terribly inconvenient,
// I created this macro to perform the casting.
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
   pvParam, fdwCreate, pdwThreadId)                 \
      ((HANDLE)_beginthreadex(                      \
         (void *)        (psa),                     \
         (unsigned)      (cbStack),                 \
         (PTHREAD_START) (pfnStartAddr),            \
         (void *)        (pvParam),                 \
         (unsigned)      (fdwCreate),               \
         (unsigned *)    (pdwThreadId)))


////////////////// DebugBreak Improvement for x86 platforms /////////////////


#ifdef _X86_
#define DebugBreak()    _asm { int 3 }
#endif


////////////////////////////// Software Exception Macro ////////////////////////


// Useful macro for creating your own software exception codes
#define MAKESOFTWAREEXCEPTION(Severity, Facility, Exception) \
   ((DWORD) ( \
   /* Severity code    */  (Severity       ) |     \
   /* MS(0) or Cust(1) */  (1         << 29) |     \
   /* Reserved(0)      */  (0         << 28) |     \
   /* Facility code    */  (Facility  << 16) |     \
   /* Exception code   */  (Exception <<  0)))


////////////////////////////// Quick MessageBox Macro //////////////////////////
```

China-pub.com
下载


```
inline void chMB(PCSTR s) {
   char szTMP[128];
   GetModuleFileNameA(NULL, szTMP, chDIMOF(szTMP));
   MessageBoxA(GetActiveWindow(), s, szTMP, MB_OK);
}


////////////////////////////// Assert/Verify Macros //////////////////////////////


inline void chFAIL(PSTR szMsg) {
   chMB(szMsg);
   DebugBreak();
}
// Put up an assertion failure message box.
inline void chASSERTFAIL(LPCSTR file, int line, PCSTR expr) {
   char sz[128];
   wsprintfA(sz, "File %s, line %d : %s", file, line, expr);
   chFAIL(sz);
}


// Put up a message box if an assertion fails in a debug build.
#ifdef _DEBUG
#define chASSERT(x) if (!(x)) chASSERTFAIL(__FILE__, __LINE__, #x)
#else
#define chASSERT(x)
#endif


// Assert in debug builds, but don't remove the code in retail builds.
#ifdef _DEBUG
#define chVERIFY(x) chASSERT(x)
#else
#define chVERIFY(x) (x)
#endif


////////,////////////////////// chHANDLE_DLGMSG Macro //////////////////////////////


// The normal HANDLE_MSG macro in WindowsX.h does not work properly for dialog
// boxes because DlgProc return a BOOL instead of an LRESULT (like
// WndProcs). This chHANDLE_DLGMSG macro corrects the problem:
#define chHANDLE_DLGMSG(hwnd, message, fn)                 \
   case (message): return (SetDlgMsgResult(hwnd, uMsg,     \
      HANDLE_##message((hwnd), (wParam), (lParam), (fn))))


////////////////////////// Dialog Box Icon Setting Macro //////////////////////////


// Sets the dialog box icons
inline void chSETDLGICONS(HWND hwnd, int idi) {
   SendMessage(hwnd, WM_SETICON, TRUE,  (LPARAM)
```

China-pub.com 下载

```
      LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
         MAKEINTRESOURCE(idi)));
   SendMessage(hwnd, WM_SETICON, FALSE, (LPARAM)
      LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
      MAKEINTRESOURCE(idi)));
}


////////////////////////////// OS Version Check Macros ////////////////////////////


inline void chWindows9xNotAllowed() {
   OSVERSIONINFO vi = { sizeof(vi) };
   GetVersionEx(&vi);
   if (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) {
      chMB("This application requires features not present in Windows 9x.");
      ExitProcess(0);
   }
}


inline void chWindows2000Required() {
   OSVERSIONINFO vi = { sizeof(vi) };
   GetVersionEx(&vi);
   if ((vi.dwPlatformId != VER_PLATFORM_WIN32_NT) && (vi.dwMajorVersion < 5)) {
      chMB("This application requires features present in Windows 2000.");
      ExitProcess(0);
   }
}


////////////////////////////// UNICODE Check Macro ////////////////////////////


// Since Windows 98 does not support Unicode, issue an error and terminate
// the process if this is a native Unicode build running on Windows 98.

// This is accomplished by creating a global C++ object. Its constructor is
// executed before WinMain.

#ifdef UNICODE

class CUnicodeSupported {
public:
   CUnicodeSupported() {
      if (GetWindowsDirectoryW(NULL, 0) <= 0) {
         chMB("This application requires an OS that supports Unicode.");
         ExitProcess(0);
      }
   }
};

// "static" stops the linker from complaining that multiple instances of the
// object exist when a single project contains multiple source files.
static CUnicodeSupported g_UnicodeSupported;
```

China-pub.com
下载


```
#endif

//////////////////////////// Force Windows subsystem ////////////////////////////

#pragma comment(linker, "/subsystem:Windows")

//////////////////////////////////// End of File ////////////////////////////////
```

China-pub.com
下载

附录B　消息分流器、子控件宏和API宏

每当我参加一些会议时，常问一些人是不是使用消息分流器，而回答通常是“ No ”。我再进一步深究这件事，发现很多人不知道消息分流器是干什么用的，甚至没有听说过它。在本书中，通过使用带有消息分流器的C/C++编写示例代码，我想向大家介绍这种不大为人所知但很有用的宏。

消息分流器定义在Microsoft Visual C++中提供的WindowsX.h文件里。通常在Windows.h文件之后紧接着包含这个文件。WindowsX..h文件就是一组#define指令，建立了一组供我们使用的宏。WindowsX.h的宏实际上分为三组：消息分流器、子控件宏和API宏。这些宏以下述的方式为我们提供帮助：

- 利用这些宏可以减少程序中要做的转换（casting）的数量，并可使所要求的转换是无错误的。使用C/C++的Windows编程中一个大的问题是所要求的转换数量。你很难看到一个不要求某种转换的Windows函数调用。但应该尽量避免使用转换，因为转换阻碍编译器发现代码中的潜在错误。一个转换是在告诉编译程序：“我知道我在这里传递了错误的转换，但就要这样做。我知道我在干什么。”当你做了许多转换时，就很容易出错。编译程序应该尽可能对此提供帮助。
- 使代码的可读性更好。
- 可简化16位Windows、32位Windows和64位Windows之间的代码移植工作。
- 易于理解（只是一些宏）
- 这些宏容易结合到已有的代码中。可以不管老的代码而立即在新的代码中使用这些宏。不必修改整个程序。
- 在C和C++代码中都可以使用这些宏，尽管当使用C++类时它们不是必需的。
- 如果需要某一个特性，而这些宏不直接支持这个特性，可以根据这个头文件中的宏，很容易地编写自己的宏。
- 不需要参照或记住费解的Windows构造。例如，许多Windows中的函数，要求一个long型参数，其中这个长参数的高字（high-word）的值代表一个东西，而其低字（low-word）又代表另一个东西。在调用这个函数之前，你必须用两个单独的值构造一个 long型值。通常利用WinDef.h中的MAKELONG宏来做这种事。我简直记不清有多少次把两个值的次序给弄反了，造成对函数传递了一个错误的值。而WindowsX.h中的宏可以帮我们的忙。

B.1　消息分流器

消息分流器（message cracker）使窗口过程的编写更加容易。通常，窗口过程是用一个大的switch语句实现的。在我的经验中，我见过有的窗口过程的switch语句包含5百多行代码。我们都知道按这种方式实现窗口过程是一种坏的习惯，但我们都这么做过。而利用消息分流器可将switch语句分成小的函数，每个窗口消息对应一个函数。这样使代码更容易管理。

有关窗口过程的另一个问题是每个消息都有wParam和lParam参数，并且根据消息的不同，这些参数的意思也不同。在某些情况下，如对WM_COMMAND消息，wParam包含两个不同的值。wParam参数的高字是通知码，而低字是控件的ID。或者是反过来？我总是忘了次序。如

China-pub.com
下载

果使用消息分流器，就不用记住或查阅这些内容。消息分流器之所以这样命名，是因为它们对任何给定的消息进行分流。为了处理WM_COMMAND消息，你只需编写这样一个函数：

```
void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
   UINT codeNotify) {

   switch (id) {

      case ID_SOMELISTBOX:
         if (codeNotify != LBN_SELCHANGE)
         break;

         // Do LBN_SELCHANGE processing.
         break;

      case ID_SOMEBUTTON:
         break;
      .
      .
      .
   }
}
```

这是多么容易！分流器查看消息的wParam和lParam参数，将参数分开，并调用你的函数。

为了使用消息分流器，必须对你的窗口过程的 switch语句做一些修改。看一看下面的窗口过程：

```
LRESULT WndProc (HWND hwnd, UINT uMsg,
   WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      HANDLE_MSG(hwnd, WM_COMMAND, Cls_OnCommand);
      HANDLE_MSG(hwnd, WM_PAINT,   Cls_OnPaint);
      HANDLE_MSG(hwnd, WM_DESTROY, Cls_OnDestroy);
      default:
         return(DefWindowProc(hwnd, uMsg, wParam, lParam));
   }
}
```

HANDLE_MSG宏在WindowsX.h中是这样定义的：

```
#define HANDLE_MSG(hwnd, message, fn) \
   case (message): \
      return HANDLE_##message((hwnd), (wParam), (lParam), (fn));
```

对于WM_COMMAND消息，预处理程序把这一行代码扩展成下面的代码：

```
case (WM_COMMAND):
   return HANDLE_WM_COMMAND((hwnd), (wParam), (lParam),
      (Cls_OnCommand));
```

定义在WindowsX.h 中的各HANDLE_WM_*宏是实际的消息分流器。它们分流 wParam参数和lParam参数，执行所有必要的转换，并调用适当的消息函数，如前面例举过的 Cls_OnCommand函数。HANDLE_WM_COMMAND宏的定义如下：

```
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
   ( (fn) ((hwnd), (int) (LOWORD(wParam)), (HWND)(lParam),
   (UINT) HIWORD(wParam)), 0L)
```

China-pub.com 下载

当预处理程序扩展这个宏时，其结果是用wParam和lParam参数的内容分流成各自的部分并经适当转换，来调用Cls_OnCommand函数。

在使用消息分流器来处理一个消息之前，应该打开WindowsX.h文件并搜索要处理的消息。例如，如果搜索WM_COMMAND，将会找到文件中包含下面代码行的部分：

```
/* void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
      UINT codeNotify); */
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
   ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)(lParam), \
   (UINT)HIWORD(wParam)), 0L)
#define FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, fn) \
   (void)(fn)((hwnd), WM_COMMAND, \
   MAKEWPARAM((UINT)(id),(UINT)(codeNotify)), \
   (LPARAM)(HWND)(hwndCtl))
```

第一行是注释行，展示要编写的函数原型。下一行是HANDLE_WM_*宏，我们已经讨论过。最后一行是消息转发器（forwarder）。假定在你处理WM_COMMAND消息时，你想调用默认的窗口过程，并让它为你做事。这个函数应该是这个样子：

```
void Cls_OnCommand (HWND hwnd, int id, HWND hwndCtl,
   UINT codeNotify) {

   // Do some normal processing.

   // Do default processing.
   FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify,
      DefWindowProc);
}
```

FORWARD_WM_*宏将分流开的消息参数重新构造成等价的wParam和lParam。然后这个宏再调用你提供的函数。在上面的例子中，宏调用DefWindowProc函数，但你可以简单地使用SendMessage或PostMessage。实际上，如果你想发送（或登记）一个消息到系统中的任何窗口，可以使用一个FORWARD_WM_*宏来帮助合并各个参数。

B.2 子控件宏

子控件宏（Child Control Macro）使发送消息到子控件变得更加容易。这些宏同FORWARO_WM_*宏很相似。每个宏的定义以一个控件类型开始（这个控件是要对它发送消息的控件），后面跟一个下横线和消息名。例如，向一个列表框发送一个 LB_GETCOUNT消息，就使用WindowsX.h中的这个宏：

```
#define ListBox_GetCount(hwndCtl) \
   ((int)(DWORD)SendMessage((hwndCtl), LB_GETCOUNT, 0, 0L))
```

关于这个宏我想说两件事。第一，它只用一个参数hwndCtl，这是列表框的窗口句柄。因为LB_GETCOUNT消息忽略了wParam和lParam参数，你不必再管这些参数。你可以看到，宏只传递了零。第二，当SendMessage返回时，结果被转换成int，所以你不必提供你自己的转换。

关于子控件宏，有一件事我不喜欢，就是这些宏要用控件窗口的句柄。许多时候，你要发送消息到一个控件，而这个控件是一个对话框的子控件。所以最终你总要调用GetDlgItem，产生这样的代码：

```
int n = ListBox_GetCount(GetDlgItem(hDlg, ID_LISTBOX));
```

China-pub.com
下载

比起使用SendDlgItemMessage，这个代码的运行虽然不慢，但你的程序会包含一些额外的代码。这是由于对GetDlgItem的额外调用。如果需要对同一控件发送几个消息，你可能想调用一次GetDlgItem，保存子窗口的句柄，然后调用你需要的所有的宏，见下面的代码：

```
HWND hwndCtl = GetDlgItem(hDlg, ID_LISTBOX);
int n = ListBox_GetCount(hwndCtl);
ListBox_AddString(hwndCtl, "Another string");
.
.
.
```

如果按这种方式设计你的代码，你的程序会运行得更快，因为这样就不会反复地调用GetDlgItem。如果你的对话框有许多控件并且你要寻找的控件在Z序的结尾，则GetDlgItem可能是个很慢的函数。

B.3 API宏

API宏可以简化某些常用的操作，如建立一种新字体，选择字体到设备环境，保存原来字体的句柄。代码的形式如下：

```
HFONT hfontOrig = (HFONT) SelectObject(hdc, (HGDIOBJ) hfontNew);
```

这个语句要求两个转换以得到没有编译警告错误的编译。在WindowsX.h中有一个宏，正是为了这个用途而设计：

```
#define SelectFont(hdc, hfont) \
   ((HFONT) SelectObject( (hdc), (HGDIOBJ) (HFONT) (hfont)))
```

如果你使用这个宏，你的程序中的代码行就变成：

```
HFONT hfontOrig = SelectFont(hdc, hfontNew);
```

这行代码更容易读，也不容易出错。

在WindowsX.h中还有其他一些API宏，有助于常用的Windows任务。建议读者了解并使用这些宏。