

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

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。

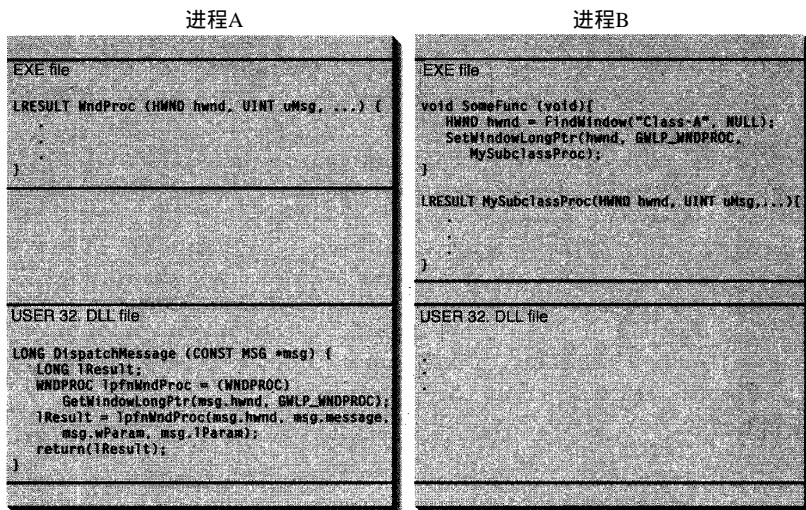


图22-1 进程B中的线程试图为进程A中的线程创建的窗口建立子类

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

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。

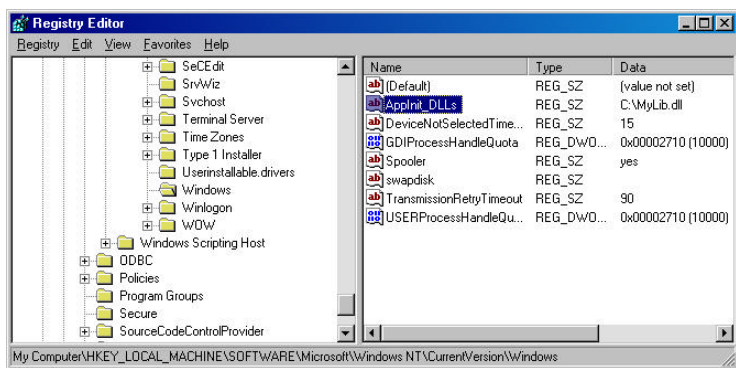


图 22-2 注册表窗口

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

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。

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:

$$\text{GetMsgProc B} = \text{hinstDll B} + (\text{GetMsgProc A} - \text{hinstDll A})$$

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(HH00K 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UnhookWindowsHookEx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

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。

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 的实现是很容易的，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(
```

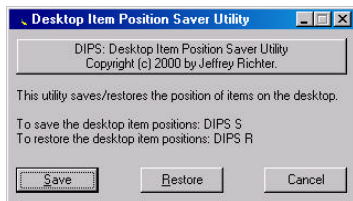


图 22-3 桌面项目位置保存器实用工具窗口

```
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DIPS.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。

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) {

```



```

        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.

```

```

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, (LPARAM) 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

```

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

```
// 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.
```

```

        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.
    }
}

```


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

```

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) {
        case WM_CLOSE, DLGMSG(hwnd, WM_CLOSE, Dlg_OnClose);

        case WM_APP:
    }
}

```

DIPSLib.h

DIPSLib.rc

[illegible]

[illegible]

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);
```

```
#ifndef 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);
```

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);
```

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LoadLibraryW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。

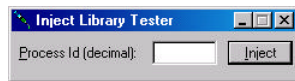


图22-4 Inject Library Tester对话框

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。

清单22-2 InjLib示例应用程序



InjLib.cpp

```
#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"
```

```
#ifndef UNICODE
#define InjectLib InjectLibW
#define EjectLib EjectLibW
#else
#define InjectLib InjectLibA
#define EjectLib EjectLibA
#endif // !UNICODE
```

```

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);
    }
}

```

```

    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
    wprintfW(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");
    }
}

```

```

    if (pfnInreadRtn == 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) {

```



```

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 //

```

//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

```

```

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

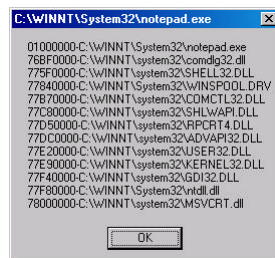


图22-5 查找结果对话框

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

```
/* **** */
```

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

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

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ExitProcess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，

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程序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内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

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.
}
```

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

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。

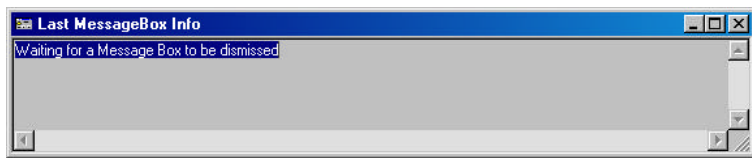


图22-6 运行LastMsgBoxInfo时出现的对话框

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。

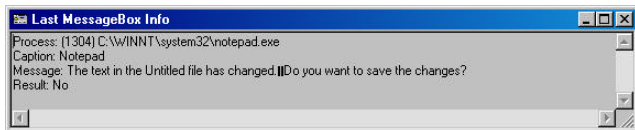
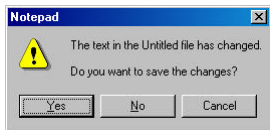


图22-7 运行Notepad时显示的消息框

图22-8 关闭Notepad时显示的LastMsgBoxInfo对话框

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  
*/
```

```
#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) {
```



```

//
// 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

```

```
#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.
```

```

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);
}

```



```

    // 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);
}

```

```

/////////////////////////////////////////////////////////////////
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
*****/

```

*****/

```
#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;
```

```

    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)

```

```

        ? (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

```

```

for (; pThunk->ul.Function; pThunk++) {

    // Get the address of the function address
    PROC* ppfn = (PROC*) &pThunk->ul.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);

```



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

void WINAPI 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);
}
```

```
// 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
};
```

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