

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。

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 Microsoft 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

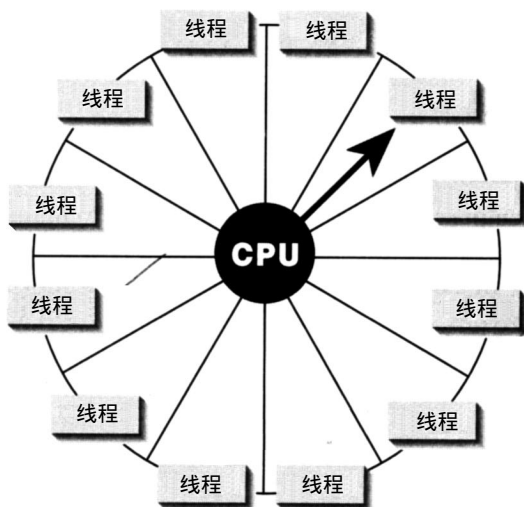


图4-1 操作系统在单个CPU计算机上用循环方式为各个线程提供时间片

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:CONSOLE，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,
```

```
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，程序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，程序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，程序员可以有 4 个选择：

- 将 main 函数改为 WinMain。通常这不是最佳的选择，因为程序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

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 = WinMain(GetModuleHandle(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。

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
_wenviron	wchar_t**	一个环境字符串
_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像来加载图标资源，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像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

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 并为其传递 "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

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。

CommandLineToArgvW 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。

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。

Windows 98 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：

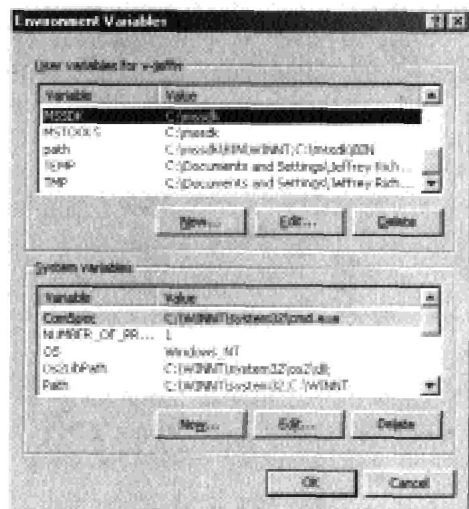


图4-2 使用Environment Variables 对话框修改变量

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 GetEnvironmentVariable 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

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_NOGFAULTERRORBOX	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_NOGPFALTERRORBOX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

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

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

(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,

VER_NT_WORKSTATION小于VER_NT_SERVER。但是对于VER_SUITE_NAME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。

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）。

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`。这种异常情况当然有些奇怪，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

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,

```

```

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。

- 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

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 and 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;
```

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初始化为零，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
<code>cb</code>	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 <code>cb</code> 初始化为 <code>sizeof(STARTUPINFO)</code>
<code>lpReserved</code>	两者兼有	保留。必须初始化为 NULL
<code>lpDesktop</code>	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 <code>lpDesktop</code> 是 NULL（这是最常见的情况），那么该进程将与当前桌面相关联
<code>lpTitle</code>	控制台	用于设定控制台窗口的名称。如果 <code>lpTitle</code> 是 NULL，则可执行文件的名称将用作窗口名
<code>dwX</code> <code>dwY</code>	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 <code>CW_USEDEFAULT</code> 作为 <code>CreateWindow</code> 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
<code>dwXSize</code> <code>dwYsize</code>	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 <code>CW_USEDEFAULT</code> 用作 <code>CreateWindow</code> 的 <code>nWidth</code> 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
<code>dwXCountChars</code> <code>dwYCountChars</code>	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）
<code>dwFillAttribute</code>	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
<code>dwFlags</code>	两者兼有	请参见下一段和表 4-7 的说明
<code>wShowWindow</code>	窗口	用于设定如果子应用程序初次调用的 <code>ShowWindow</code> 将 <code>SW_SHOWDEFAULT</code> 作为 <code>nCmdShow</code> 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 <code>Show Window</code> 函数的任何一个 <code>SW_*</code> 标识符

(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	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
hStdOutput		设置，hStdInput用于标识键盘缓存，hStdOutput和hStdError
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方式启动运行

另外还有两个标志，即 STARTF_FORCEONFEEDBACK和STARTF_ + FORCEOFFFEEDBACK，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。

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。

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 GetStartupInfo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。

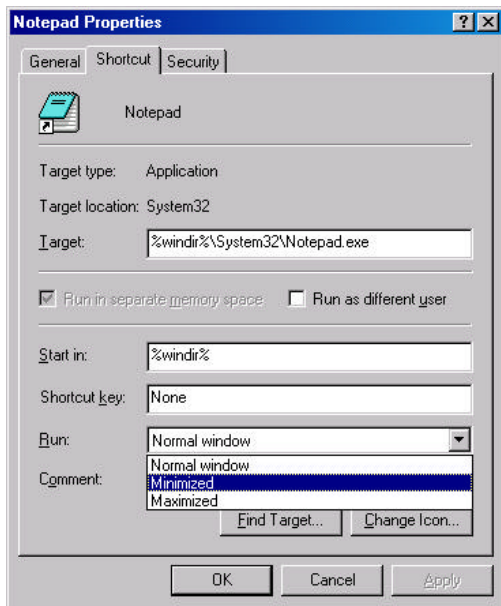


图4-3 运行Notepad的快捷方式的属性页

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。

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ExitProcess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:
```

```
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。

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，你希望通过调用 `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

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);
}
```



```
// 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

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 VMMMap 菜单项（当查看模块信息时，该菜单项禁用）。如果选定 VMMMap 菜单项，可使 VMMMap 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。如果模块不是在其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。

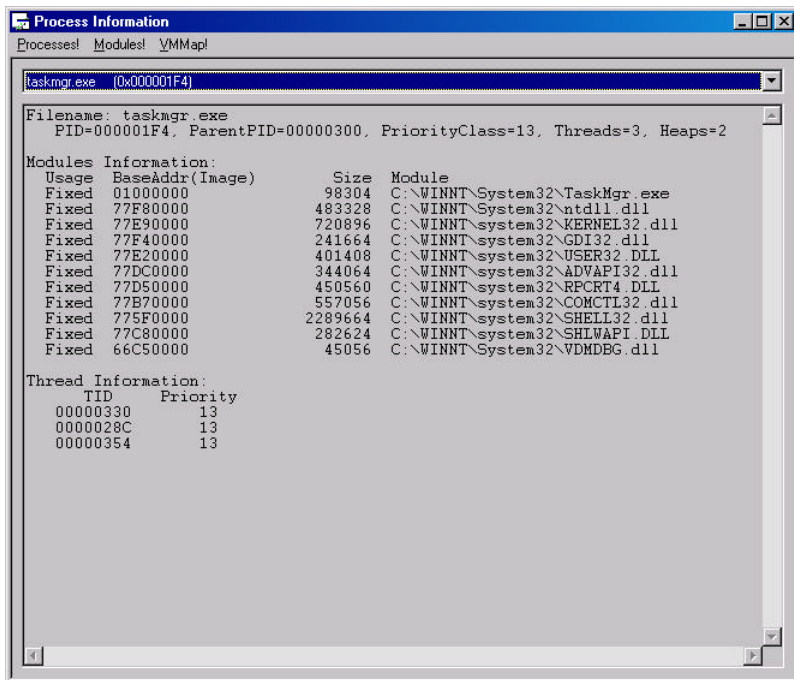


图4-4 运行中的ProcessInfo

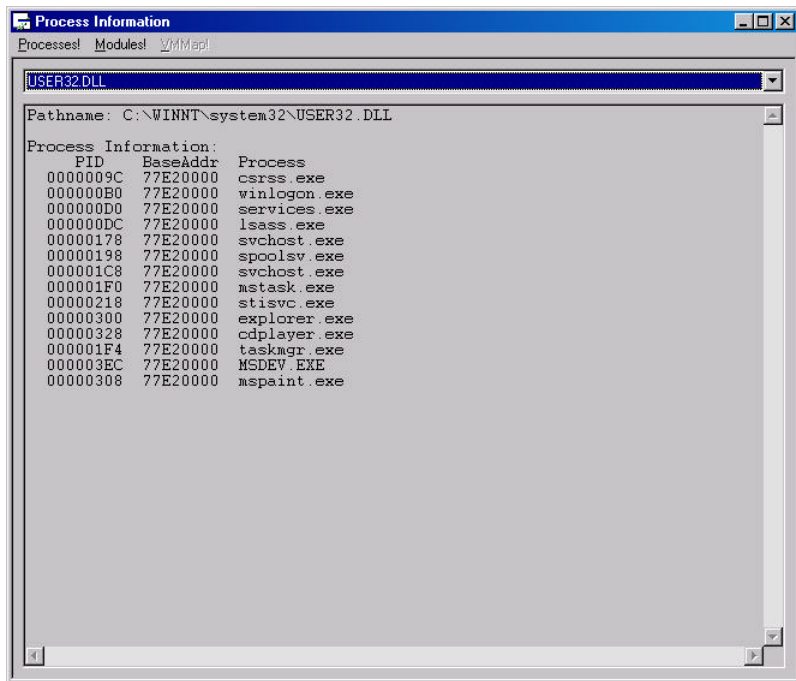


图4-5 ProcessInfo显示User32.dll加载到它们的地址空间的所有进程

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


```

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
        wprintf(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);
            }
        }
    }
}

```



```

}

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);
}

```

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

```

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

```

```

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

```

```

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:

```

```

    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 VMap\\\" %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);

```



```

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

```

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

```

//
// 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

```

```

////////////////////////////////////
//
// 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;

```

```

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 {

```

```

        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 {

```



```

    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));
}

```

```

inline BOOL CToolhelp::ThreadNext(PTHREADENTRY32 pte) const {
    return(Thread32Next(m_hSnapshot, pte));
}

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

inline int CToolhelp::HowManyHeaps() const {

    int nHowManyHeaps = 0;
    HEAPLIST32 h1 = { sizeof(h1) };
    for (BOOL fOk = HeapListFirst(&h1); fOk; fOk = HeapListNext(&h1))
        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 ph1) const {

    return(Heap32ListFirst(m_hSnapshot, ph1));
}

inline BOOL CToolhelp::HeapListNext(PHEAPLIST32 ph1) const {

    return(Heap32ListNext(m_hSnapshot, ph1));
}

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 h1 = { sizeof(h1) };
    for (BOOL fOkHL = HeapListFirst(&h1); fOkHL; fOkHL = HeapListNext(&h1)) {
        HEAPENTRY32 he = { sizeof(he) };
        BOOL fOkHE = HeapFirst(&he, h1.th32ProcessID, h1.th32HeapID);
        for (; fOkHE; fOkHE = HeapNext(&he)) {

```

```
MEMORY_BASIC_INFORMATION mbi;
VirtualQueryEx(hProcess, (PVOID) he.dwAddress, &mbi, sizeof(mbi));
if (chINRANGE(mbi.AllocationBase, pvBlock,
    (PBYTE) mbi.AllocationBase + mbi.RegionSize)) {

    *pdwFlags = h1.dwFlags;
    return(TRUE);
}
}
return(FALSE);
}
```

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