

第14章 虚拟内存

上一章介绍了系统如何管理虚拟内存，每个进程如何获得它自己的私有地址空间，进程的地址空间是个什么样子等内容。这一章不再介绍抽象的概念，而要具体介绍几个 Windows 函数，这些函数能够提供关于系统内存管理以及进程中的虚拟地址空间等信息。

14.1 系统信息

许多操作系统的值是根据主机而定的，比如页面的大小，分配粒度的大小等。这些值决不应该用硬编码的形式放入你的源代码。相反，你始终都应该在进程初始化的时候检索这些值，并在你的源代码中使用检索到的值。GetSystemInfo函数将用于检索与主机相关的值：

```
VOID GetSystemInfo(LPSYSTEM_INFO psinf);
```

必须传递SYSTEM_INFO结构的地址给这个函数。这个函数将初始化所有的结构成员然后返回。下面是SYSTEM_INFO数据结构的样子。

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;    // Obsolete, do not use
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

当系统引导时，它要确定这些成员的值是什么。对于任何既定的系统来说，这些值总是相同的，因此决不需要为任何既定的进程多次调用该函数。由于有了 GetSystemInfo函数，因此应用程序能够在运行的时候查询这些值。在该结构的所有成员中，只有 4 个成员与内存有关。表14-1对这4个成员作了描述。

表14-1 与内存有关的成员函数

成 员 名	描 述
dwPageSize	用于显示CPU的页面大小。在x86 CPU上，这个值是4096字节。在Alpha CPU上，这个值是8192字节。在IA-64上，这个值是8192字节
lpMinimumApplicationAddress	用于给出每个进程的可用地址空间的最小内存地址。在 Windows 98上，这个值是4 194 304，或0x00400000，因为每个进程的地址空间中下面的4MB是不能使用的。在 Windows 2000上，这个值是65 536或0x00010000，因为每个进程的地址空间中开头的64KB总是空闲的

(续)

成 员 名	描 述
IpMaximumApplicationAddress	用于给出每个进程的可用地址空间的最大内存地址。在 Windows 98 上，这个地址是 2 147 483 647 或 0x7FFFFFFF，因为共享内存映射文件区域和共享操作系统代码包含在上面的 2 GB 分区中。在 Windows 2000 上，这个地址是内核方式内存开始的地址，它不足 64KB
dwAllocationGranularity	显示保留的地址空间区域的分配粒度。截止到撰写本书时，在所有 Windows 平台上，这个值都是 65 536

该结构的其他成员与内存管理毫无关系，为了完整起见，下面也对它们进行了介绍（见表 14-2）。

表14-2 与内存无关的成员函数

成 员 名	描 述
dwOemId	已作废，不引用
WRederved	保留供将来使用，不引用
dwNumberOfProcessors	用于指明计算机中的 CPU 数目
dwActiveProcessorMask	一个位屏蔽，用于指明哪个 CPU 是活动的（允许运行线程）
dwProcessorType	只用于 Windows 98，不用于 Windows 2000，用于指明处理器的类型，如 Intel 386、486 或 Pentium
wProcessorArchitecture	只用于 Windows 2000，不用于 Windows 98，用于指明处理的结构，如 Intel、Alpha、Intel 64 位或 Alpha 64 位
wProcessorLevel	只用于 Windows 2000，不用于 Windows 98，用于进一步细分处理器的结构，如用于设定 Intel Pentium Pro 或 Pentium II
wProcessorRevision	只用于 Windows 2000，不用于 Windows 98，用于进一步细分处理器的级别

系统信息示例应用程序

清单 14-1 显示的 SysInfo 应用程序（“ 14 SysInfo.exe ”）是一个简单的调用 GetSystemInfo 函

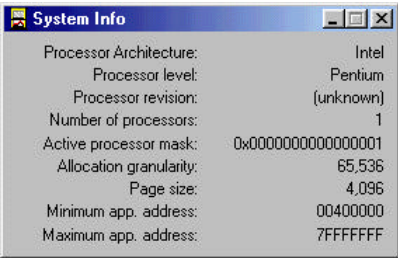


图14-1 在x86 CPU上运行Windows 98 时返回的结果

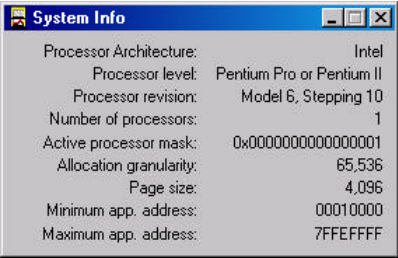


图14-2 在x86 CPU上运行32位Windows 2000 时返回的结果

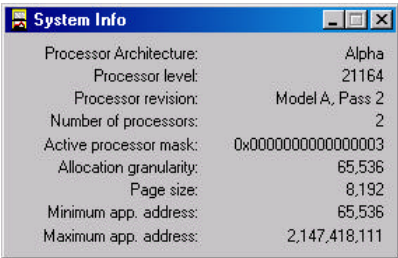


图14-3 在Alpha CPU上运行32位Windows 2000 时返回的结果

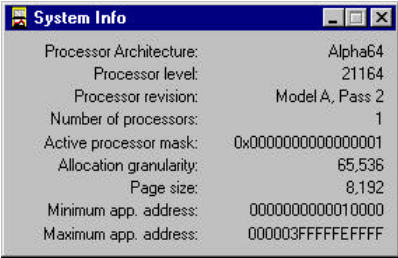


图14-4 在Alpha CPU上运行64位Windows时 返回的结果

数的程序，它显示SYSTEM_INFO结构中返回的信息。该应用程序的源代码和资源文件均位于本书所附光盘上的14-SysInfo目录下。图14-1、图14-2、图14-3和图14-4所示的两个对话框显示了在若干不同平台上运行的SysInfo应用程序返回的结果。

清单14-1 SysInfo应用程序



SysInfo.cpp

```

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

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

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

// Set to TRUE if the app is running on Windows 9x.
BOOL g_fWin9xIsHost = FALSE;

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

// This function accepts a number and converts it to a
// string, inserting commas where appropriate.
PTSTR BigNumToString(LONG lNum, PTSTR szBuf) {

    TCHAR szNum[100];
    wsprintf(szNum, TEXT("%d"), lNum);
    NUMBERFMT nf;
    nf.NumDigits = 0;
    nf.LeadingZero = FALSE;
    nf.Grouping = 3;
    nf.lpDecimalSep = TEXT(".");
    nf.lpThousandSep = TEXT(",");
    nf.NegativeOrder = 0;
    GetNumberFormat(LOCALE_USER_DEFAULT, 0, szNum, &nf, szBuf, 100);
    return(szBuf);
}

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

void ShowCPUInfo(HWND hwnd, WORD wProcessorArchitecture, WORD wProcessorLevel,

```

```
WORD wProcessorRevision) {

TCHAR szCPUArch[64] = TEXT("(unknown)");
TCHAR szCPULevel[64] = TEXT("(unknown)");
TCHAR szCPURev[64] = TEXT("(unknown)");

switch (wProcessorArchitecture) {
case PROCESSOR_ARCHITECTURE_INTEL:
    lstrcpy(szCPUArch, TEXT("Intel"));
    switch (wProcessorLevel) {
case 3: case 4:
        wsprintf(szCPULevel, TEXT("80%c86"), wProcessorLevel + '0');
        if (!g_fWin9xIsHost)
            wsprintf(szCPURev, TEXT("%c%d"),
                HIBYTE(wProcessorRevision) + TEXT('A'),
                LOBYTE(wProcessorRevision));
        break;

case 5:
        wsprintf(szCPULevel, TEXT("Pentium"));
        if (!g_fWin9xIsHost)
            wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
                HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
        break;
case 6:
        wsprintf(szCPULevel, TEXT("Pentium Pro or Pentium II"));
        if (!g_fWin9xIsHost)
            wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
                HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
        break;
    }
    break;

case PROCESSOR_ARCHITECTURE_ALPHA:
    lstrcpy(szCPUArch, TEXT("Alpha"));
    wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
        HIBYTE(wProcessorRevision) + TEXT('A'),
        LOBYTE(wProcessorRevision));
    break;

case PROCESSOR_ARCHITECTURE_IA64:
    lstrcpy(szCPUArch, TEXT("IA-64"));
    wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
        HIBYTE(wProcessorRevision) + TEXT('A'),
        LOBYTE(wProcessorRevision));
    break;

case PROCESSOR_ARCHITECTURE_ALPHA64:
    lstrcpy(szCPUArch, TEXT("Alpha64"));
    wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
        HIBYTE(wProcessorRevision) + TEXT('A'),
        LOBYTE(wProcessorRevision));
}
```

```

        break;

        case PROCESSOR_ARCHITECTURE_UNKNOWN:
        default:
            wsprintf(szCPUArch, TEXT("Unknown"));
            break;
    }
    SetDlgItemText(hwnd, IDC_PROCCARCH, szCPUArch);
    SetDlgItemText(hwnd, IDC_PROCLEVEL, szCPULevel);
    SetDlgItemText(hwnd, IDC_PROCREV, szCPURev);
}

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

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

    chSETDLGICONS(hwnd, IDI_SYSINFO);

    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    if (g_fWin9xIsHost) {
        sinf.wProcessorLevel = (WORD) (sinf.dwProcessorType / 100);
    }

    ShowCPUInfo(hwnd, sinf.wProcessorArchitecture,
        sinf.wProcessorLevel, sinf.wProcessorRevision);

    TCHAR szBuf[50];
    SetDlgItemText(hwnd, IDC_PAGESIZE,
        BigNumToString(sinf.dwPageSize, szBuf));

    _stprintf(szBuf, TEXT("%p"), sinf.lpMinimumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MINAPPADDR, szBuf);

    _stprintf(szBuf, TEXT("%p"), sinf.lpMaximumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MAXAPPADDR, szBuf);

    _stprintf(szBuf, TEXT("0x%016I64X"), (__int64) sinf.dwActiveProcessorMask);
    SetDlgItemText(hwnd, IDC_ACTIVEPROCMASK, szBuf);

    SetDlgItemText(hwnd, IDC_NUMOFPROCS,
        BigNumToString(sinf.dwNumberOfProcessors, szBuf));

    SetDlgItemText(hwnd, IDC_ALLOCGRAN,
        BigNumToString(sinf.dwAllocationGranularity, szBuf));

    return(TRUE);
}

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

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

    switch (id) {
        case IDCANCEL:

```

```

        EndDialog(hwnd, id);
        break;
    }
}

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

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

    switch (uMsg) {
        chHANDLE_DLGMSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hDlg, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

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

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

    OSVERSIONINFO vi = { sizeof(vi) };
    GetVersionEx(&vi);
    g_fWin9xIsHost = (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SYSINFO), NULL, Dlg_Proc);
    return(0);
}

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

```

SysInfo.rc

```

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

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

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

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

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

```

```

#endif // _WIN32

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

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

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

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

#endif // APSTUDIO_INVOKED

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

IDD_SYSINFO_DIALOG DISCARDABLE 18, 18, 186, 97
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "System Info"
FONT 8, "MS Sans Serif"
BEGIN
    RTEXT        "Processor Architecture:", IDC_STATIC, 4, 4, 88, 8,
                SS_NOPREFIX
    RTEXT        "ID_PROCCARCH", IDC_PROCCARCH, 96, 4, 84, 8, SS_NOPREFIX
    RTEXT        "Processor level:", IDC_STATIC, 4, 14, 88, 8, SS_NOPREFIX
    RTEXT        "ID_PROCCLEVEL", IDC_PROCCLEVEL, 96, 14, 84, 8, SS_NOPREFIX
    RTEXT        "Processor revision:", IDC_STATIC, 4, 24, 88, 8, SS_NOPREFIX
    RTEXT        "ID_PROCCREV", IDC_PROCCREV, 96, 24, 84, 8, SS_NOPREFIX
    RTEXT        "Number of processors:", IDC_STATIC, 4, 34, 88, 8, SS_NOPREFIX
    RTEXT        "ID_NUMOFPROCS", IDC_NUMOFPROCS, 96, 34, 84, 8, SS_NOPREFIX
    RTEXT        "Active processor mask:", IDC_STATIC, 4, 44, 88, 8,
                SS_NOPREFIX
    RTEXT        "ID_ACTIVEPROCMAK", IDC_ACTIVEPROCMAK, 96, 44, 84, 8,
                SS_NOPREFIX
    RTEXT        "Allocation granularity:", IDC_STATIC, 4, 54, 88, 8,
                SS_NOPREFIX
    RTEXT        "ID_ALLOCGRAN", IDC_ALLOCGRAN, 96, 54, 84, 8, SS_NOPREFIX
    RTEXT        "Page size:", IDC_STATIC, 4, 64, 88, 8, SS_NOPREFIX

```

```

RTEXT          "ID_PAGESIZE", IDC_PAGESIZE, 96, 64, 84, 8, SS_NOPREFIX
RTEXT          "Minimum app. address:", IDC_STATIC, 4, 74, 88, 8, SS_NOPREFIX
RTEXT          "ID_MINAPPADDR", IDC_MINAPPADDR, 96, 74, 84, 8, SS_NOPREFIX
RTEXT          "Maximum app. address:", IDC_STATIC, 4, 84, 88, 8, SS_NOPREFIX
RTEXT          "ID_MAXAPPADDR", IDC_MAXAPPADDR, 96, 84, 84, 8, SS_NOPREFIX
END

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

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SYSINFO          ICON          DISCARDABLE          "SysInfo.Ico"

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

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_SYSINFO, DIALOG
    BEGIN
        RIGHTMARGIN, 170
        BOTTOMMARGIN, 77
    END
END
#endif          // APSTUDIO_INVOKED

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

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

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

```

14.2 虚拟内存的状态

Windows函数GlobalMemoryStatus可用于检索关于当前内存状态的动态信息：

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS pmst);
```

我认为这个函数的名字起的并不好，因为 GlobalMemoryStatus表示这个函数多少与 16 位

Windows中的全局堆栈有些相关。我认为 GlobalMemoryStatus应该改为 VirtualMemoryStatus比较好些。

当调用GlobalMemoryStatus时，必须传递一个MEMORYSTATUS结构的地址。下面显示了MEMORYSTATUS的数据结构。

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

在调用GlobalMemoryStatus之前，必须将dwLength成员初始化为用字节表示的结构的大小，即一个MEMORYSTATUS结构的大小。这个初始化操作使得 Microsoft能够将成员添加给将来的Windows版本中的这个结构，而不会破坏现有的应用程序。当调用GlobalMemoryStatus时，它将对该结构的其余成员进行初始化并返回。下一节中的VMStat示例应用程序将要描述各个成员及其含义。

如果希望应用程序在内存大于 4GB 的计算机上运行，或者合计交换文件的大小大于 4GB，那么可以使用新的 GlobalMemoryStatusEx函数：

```
BOOL GlobalMemoryStatusEx(LPMEMORYSTATUSEX pmst);
```

必须给该函数传递新的MEMORYSTATUSEX结构的地址：

```
typedef struct _MEMORYSTATUSEX {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvailPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONG ullAvailVirtual;
    DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

这个结构与原先的MEMORYSTATUS结构基本相同，差别在于新结构的所有成员的大小都是64位宽，因此它的值可以大于4GB。最后一个成员是ullAvailExtendedVirtual，用于指明在调用进程的虚拟地址空间的极大内存（VLM）部分中未保留内存的大小。该VLM部分只适用于某些配置中的某些CPU结构。

虚拟内存状态示例应用程序

清单14-2列出的VMStat应用程序显示了一个简单的对话框，用于列出调用 GlobalMemoryStatus函数的结果。对话框中的信息每秒钟更新一次，因此，你可以在对系统中的其他进程进行操作时，使应用程序继续运行。该应用程序的源代码和资源文件均位于本书所附光盘上的14-VMStat目录下。图14-5显示了在内存为128 MB的Intel Pentium II计算机上的Windows 2000下运行该程序的结果。

dwMemoryLoad成员（图14-5中显示为Memory Load）给出了内存管理系统的大致繁忙程

度。该数字的范围是0至100。计算这个值时使用的具体算法在Windows 98与Windows 2000上是不同的。根据将来的操作系统版本的情况,该算法还会有所变化。实际上,该成员变量报告的值是没有什么用处的。

dwTotalPhys成员(图14-5中显示为TotalPhys)用于指明存在的物理存储器(EAM)的总字节数。在128 MB的Pentium II计算机上,这个值是133 677 056,它只比128 MB少540 672个字节。GlobalMemoryStatus之所以不报告全部的128MB,原因是在引导进程中,系统将一些内存保留为非页面内存池。这些内存甚至不能被内核使用。dwAvailPhys成员(图14-5中显示为AvailPyhs)用于指明可供分配的物理存储器的总字节数。

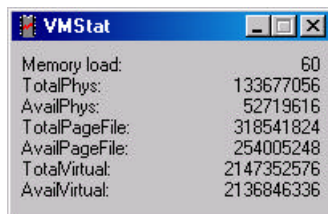
dwTotalPageFile成员(图14-5中显示为TotalPageFile)用于指明你的硬盘上的页文件中包含的最大字节数。虽然VMStat报告的页文件当前是318 574 592字节,但是系统可以根据需要对页交换文件进行扩大和压缩。dwAvailPageFile成员(图14-5中显示为AvailPageFile)用于指明页文件中有233 046 016字节尚未提交给任何进程,因此,如果一个进程决定提交任何私有内存的话,目前就可以使用这些字节。

dwTotalVirtual成员(图14-5中显示为TotalVitual)用于指明每个进程的地址空间中私有的总字节数。它的值是2 147 352 576,比准确的2 GB少128 KB。从0x00000000至0x0000FFFF以及从0x7FFF0000至0x7FFFFFFF的两个分区是不能访问的地址空间,这正好等于128 KB这个差额。如果在Windows 98下运行VMStat,你将看到dwTotalVirtual返回的值是2 143 289 344,这比准确的2 GB只少4 MB。之所以差4 MB,原因是系统决不允许应用程序访问从0x00000000至0x003FFFFFFF之间的这个4 MB分区。

最后一个成员dwAvailVirtual(图14-5中显示为AvailVirtual)是该结构中专门用于调用GlobalMemoryStatus的进程的唯一成员,所有其他成员在系统中的使用情况都是一样的,而不管是哪个进程调用GlobalMemoryStatus。若要计算这个值,GlobalMemoryStatus将调用进程的地址空间中的所有空闲区域相加。dwAvailVirtual的值2 136 846 336表示可供VMStat随意使用的空闲地址空间的数量。如果将dwTotalVirtual的值减去dwAvailVirtual的值,你会看到VMStat在它的虚拟地址空间中保留了10 506 240个字节。

没有一个成员能够指明进程当前使用的物理存储器的数量。

清单14-2 VMStat应用程序



Memory load:	60
TotalPhys:	133677056
AvailPhys:	52719616
TotalPageFile:	318541824
AvailPageFile:	254005248
TotalVirtual:	2147352576
AvailVirtual:	2136846336

图14-5 VMStat 运行结果显示



VMStat.cpp

```

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

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

```

```

////////////////////////////////////////////////////////////////
// The update timer's ID
#define IDT_UPDATE 1

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

    chSETDLGICONS(hwnd, IDI_VMSTAT);

    // Set a timer so that the information updates periodically.
    SetTimer(hwnd, IDT_UPDATE, 1 * 1000, NULL);

    // Force a timer message for the initial update.
    FORWARD_WM_TIMER(hwnd, IDT_UPDATE, SendMessage);
    return(TRUE);
}

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

void Dlg_OnTimer(HWND hwnd, UINT id) {

    // Initialize the structure length before calling GlobalMemoryStatus.
    MEMORYSTATUS ms = { sizeof(ms) };
    GlobalMemoryStatus(&ms);

    TCHAR szData[512] = { 0 };
    _stprintf(szData, TEXT("%d\n%d\n%I64d\n%I64d\n%I64d\n%I64d\n%I64d"),
        ms.dwMemoryLoad, ms.dwTotalPhys,
        (__int64) ms.dwAvailPhys, (__int64) ms.dwTotalPageFile,
        (__int64) ms.dwAvailPageFile, (__int64) ms.dwTotalVirtual,
        (__int64) ms.dwAvailVirtual);
    SetDlgItemText(hwnd, IDC_DATA, szData);
}

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

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

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

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

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

    switch (uMsg) {
        case chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        case chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
        case chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);
    }
}

```

```

    }
    return(FALSE);
}

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

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

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMSTAT), NULL,Dlg_Proc);
    return(0);
}

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

```

VMStat.rc

```

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

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

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

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

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

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

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

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

```

```

END

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

#endif    // APSTUDIO_INVOKED

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

IDD_VMSTAT DIALOG DISCARDABLE  60, 60, 132, 66
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "VMStat"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT          "Memory load:\nTotalPhys:\nAvailPhys:\nTotalPageFile:\n\
                    AvailPageFile:\nTotalVirtual:\nAvailVirtual:",
                    IDC_STATIC,4,4,51,56
    RTEXT          "Memory load:\nTotalPhys:\nAvailPhys:\nTotalPageFile:\n\
                    AvailPageFile:\nTotalVirtual:\nAvailVirtual:",
                    IDC_DATA,60,4,68,56
END

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

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMSTAT          ICON    DISCARDABLE     "VMStat.Ico"

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

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_VMSTAT, DIALOG
    BEGIN
        RIGHTMARGIN, 108
        BOTTOMMARGIN, 65
    END
END
#endif    // APSTUDIO_INVOKED

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

```

```

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

//
#endif // not APSTUDIO_INVOKED

```

14.3 确定地址空间的状态

Windows提供了一个函数，可以用来查询地址空间中内存地址的某些信息（如大小，存储器类型和保护属性等）。实际上本章后面显示的VMMMap示例应用程序就使用这个函数来生成第13章所附的虚拟内存表交换信息。这个函数称为VirtualQuery：

```

DWORD VirtualQuery(
    LPCVOID pvAddress,
    PMEMORY_BASIC_INFORMATION pmbi,
    DWORD dwLength);

```

Windows还提供了另一个函数，它使一个进程能够查询另一个进程的内存信息：

```

DWORD VirtualQueryEx(
    HANDLE hProcess,
    LPCVOID pvAddress,
    PMEMORY_BASIC_INFORMATION pmbi,
    DWORD dwLength);

```

这两个函数基本相同，差别在于使用VirtualQueryEx时，可以传递你想要查询的地址空间信息的进程的句柄。调试程序和其他实用程序使用这个函数最多，几乎所有的应用程序都只需要调用VirtualQuery函数。当调用VirtualQuery（Ex）函数时，pvAddress参数必须包含你想要查询其信息的虚拟内存地址。Pmbi参数是你必须分配的MEMORY_BASIC_INFORMATION结构的地址。该结构在WinNT.h文件中定义为下面的形式：

```

typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;

```

最后一个参数是dwLength，用于设定MEMORY_BASIC_INFORMATION结构的大小。VirtualQuery（Ex）函数返回拷贝到缓存中的字节的数量。

根据在pvAddress参数中传递的地址，VirtualQuery（Ex）函数将关于共享相同状态、保护属性和类型的相邻页面的范围信息填入MEMORY_BASIC_INFORMATION结构中。表14-3描述了该结构的成员。

表14-3 MEMORY_BASIC_INFORMATION结构的成员函数

成 员 名	描 述
BaseAddress	与pvAddress参数的值相同，但是四舍五入为页面的边界值
AllocationBase	用于指明包含在pvAddress参数中设定的地址区域的基地址
AllocationProtect	用于指明一个地址空间区域被初次保留时赋予该区域的保护属性
RegionSize	用于指明从基地址开始的所有页面的大小（以字节为计量单位） 这些页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型
State	用于指明所有相邻页面的状态（MEM_FREE、MEM_RESERVE或MEM_COMMIT）。这些页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型 如果它的状态是空闲，那么AllocationBase、AllocationProtect、Protect和Type等成员均未定义 如果状态是MEM_RESERVE，则Protect成员未定义
Protect	用于指明所有相邻页面的保护属性（PAGE_*）。这些页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型
Type	用于指明支持所有相邻页面的物理存储器的类型（MEM_IMAGE、MEM_MAPPED或MEM_PRIVATE）。这些相邻页面与含有用pvAddress参数设定的地址的页面拥有相同的保护属性、状态和类型。如果是Windows 98，那么这个成员将总是MEM_PRIVATE

14.3.1 VMQuery函数

当我刚刚开始学习Windows的内存结构的时候，我将VirtualQuery函数用作我的学习指导。实际上，如果观察本书的第1版(原英文版第1版)，你会发现VMMMap示例应用程序比本书下一节中介绍的程序简单得多。在老版本中，我使用了一个简单的循环，它反复调用VirtualQuery函数，而每次调用时，我只是创建一个简单的代码行，它包含了MEMORY_BASIC_INFORMATION结构的各个成员。我研究了它的交换方式，并且在参考SDK文档（当时这个文档很不完善）时将内存管理结构组合在一起。从此我学到了许多知识。虽然VirtualQuery和MEMORY_BASIC_INFORMATION结构使你能够深入了解许多关于内存的情况，但是现在它们无法为你提供使你真正了解全部情况的足够信息。

问题是MEMORY_BASIC_INFORMATION结构无法返回系统已经存放在内部的所有信息。如果你有一个内存地址，想获得关于该地址的某些简单的信息，那么使用VirtualQuery函数的效果是相当不错的。如果只是想知道是否给一个地址提交了物理存储器，或者是否可以向一个内存地址读取或写入信息，那么VirtualQuery函数的作用也很好。但是，如果想知道已保留的地址空间区域的合计大小，或者想要知道一个区域中的地址空间块的数量，或者想知道一个区域是否包含线程堆栈，那么仅仅一次调用VirtualQuery将无法为你提供你想知道的信息。

为了获得完整的内存信息，我创建了一个函数，即VMQuery：

```

BOOL VMQuery(
    HANDLE hProcess,
    PVOID pvAddress,
    PVMQUERY pVMQ);

```

该函数与VirtualQueryEx有些类似，它拥有一个进程句柄（在hProcess中），一个内存地址（在pvAddress中）和一个指向将被填充的结构的指针（由pVMQ设定）。该结构是个VMQUERY结构，下面是结构的定义：

```

typedef struct {
    // Region information

```



```

PVOID pvRgnBaseAddress;
DWORD dwRgnProtection; // PAGE_*
SIZE_T RgnSize;
DWORD dwRgnStorage;      // MEM_*: Free, Image, Mapped, Private
DWORD dwRgnBlocks;
DWORD dwRgnGuardBlks;    // If > 0, region contains thread stack
BOOL fRgnIsAStack;       // TRUE if region contains thread stack

// Block information
PVOID pvBlkBaseAddress;
DWORD dwBlkProtection; // PAGE_*
SIZE_T BlkSize;
DWORD dwBlkStorage;      //
MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;

```

只要简单地看一眼就会发现，我的 VMQUERY 结构比 Windows 的 MEMORY_BASIC_INFORMATION 结构包含了多得多的信息。我的结构分成两个不同的部分：一个是区域信息，另一个是块信息。区域信息部分用于描述关于区域的信息，块信息部分用于描述关于包含用 pvAddress 参数设定的地址块的信息。表 14-4 对所有的成员进行了说明。

表 14-4 VMQUERY 函数的成员变量

成 员 名	描 述
pvRgnBaseAddress	指明虚拟地址空间区域的基地址，该区域包含了用 pvAddress 参数设定的地址
dwRgnProtection	指明地址空间区域刚刚被保留时赋予该区域的保护属性（PAGE_*）
RgnSize	指明已经保留的地址空间区域的大小（以字节为计量单位）
dwRgnStorage	指明区域中的地址空间块主体使用的物理存储器的类型。该值可以是下列几个值之一：MEM_FREE、MEM_IMAGE、MEM_MAPPED 或 MEM_PRIVATE。Windows 98 不区分不同的内存类型，因此在 Windows 98 下本成员将总是 MEM_FREE 或 MEM_PRIVATE
dwRgnBlocks	指明地址空间区域中包含的地址块的数量
dwRgnGuardBlks	指明 PAGE_GUARD 保护属性标志已经打开的地址块的数量。该值通常既可以是 0，也可以是 1。如果是 1，则表示该区域已经被保留，以包含一个线程堆栈。在 Windows 98 下，这个成员的值总是 0
fRgnIsAStack	指明一个区域是否包含线程堆栈。该值是通过“优化推测”来确定，因为不可能有百分之百的把握知道一个区域是否包含线程堆栈
pvBlkBaseAddress	指明包含用 pvAddress 参数设定的地址块的基地址
dwBlkProtection	指明包含用 pvAddress 参数设定的地址块的保护属性
BlkSize	指明包含用 pvAddress 参数设定的地址块的大小（以字节为计量单位）
dwBlkStorage	指明包含用 pvAddress 参数设定的地址块的内容。这个值可以是下列值之一：MEM_FREE、MEM_RESERVE、MEM_IMAGE、MEM_MAPPED 或 MEM_PRIVATE。在 Windows 98 下，这个值决不能是 MEM_IMAGE 或 MEM_MAPPED

毫无疑问，VMQuery 函数必须执行相当数量的处理操作，包括多次调用 VirtualQueryEx，以便获取所有的信息，这意味着它的运行速度要大大低于 VirtualQueryEx 函数。由于这个原因，在决定调用这两个函数中的哪一个时，应该三思而后行。如果不需要通过 VMQuery 获得的额外信息，那么应该调用 VirtualQuery 或 VirtualQueryEx 函数。

清单 14-3 列出的 VMQuery.cpp 文件显示了我如何获得和管理设置 VMQUERY 结构的成员时需要的全部信息。VMQuery.cpp 和 VMQuery.h 两个文件均在本书所附光盘上的 14-VMMMap 目录

下。清单14-3中的代码注释说明了处理数据的方法。

清单14-3 VMQuery的程序清单

VMQuery.cpp

Module: VMQuery.cpp

Notices: Copyright (c) 2000 Jeffrey Richter

*****/

```
#include "..\CmnHdr.h" /* See Appendix A. */
```

```
#include <windowsx.h>
```

```
#include "VMQuery.h"
```

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

```
// Helper structure
```

```
typedef struct {
```

```
    SIZE_T RgnSize;
```

```
    DWORD dwRgnStorage; // MEM_*: Free, Image, Mapped, Private
```

```
    DWORD dwRgnBlocks;
```

```
    DWORD dwRgnGuardBlks; // If > 0, region contains thread stack
```

```
    BOOL fRgnIsAStack; // TRUE if region contains thread stack
```

```
} VMQUERY_HELP;
```

```
// This global, static variable holds the allocation granularity value for
```

```
// this CPU platform. Initialized the first time VMQuery is called.
```

```
static DWORD gs_dwAllocGran = 0;
```

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

```
// Iterates through a region's blocks and returns findings in VMQUERY_HELP
```

```
static BOOL VMQueryHelp(HANDLE hProcess, LPCVOID pvAddress,
```

```
    VMQUERY_HELP *pVMQHelp) {
```

```
    // Each element contains a page protection
```

```
    // (i.e.: 0=reserved, PAGE_NOACCESS, PAGE_READWRITE, etc.)
```

```
    DWORD dwProtectBlock[4] = { 0 };
```

```
    ZeroMemory(pVMQHelp, sizeof(*pVMQHelp));
```

```
    // Get address of region containing passed memory address.
```

```
    MEMORY_BASIC_INFORMATION mbi;
```

```
    BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
```

```
        == sizeof(mbi));
```

```
    if (!fOk)
```

```
        return(fOk); // Bad memory address, return failure
```

```
    // Walk starting at the region's base address (which never changes)
```

```
    PVOID pvRgnBaseAddress = mbi.AllocationBase;
```

```
    // Walk starting at the first block in the region (changes in the loop)
```

```
    PVOID pvAddressBlk = pvRgnBaseAddress;
```

```
// Save the memory type of the physical storage block.
pVMQHelp->dwRgnStorage = mbi.Type;

for (;;) {
    // Get info about the current block.
    fOk = (VirtualQueryEx(hProcess, pvAddressBlk, &mbi, sizeof(mbi))
        == sizeof(mbi));
    if (!fOk)
        break;    // Couldn't get the information, end loop.

    // Is this block in the same region?
    if (mbi.AllocationBase != pvRgnBaseAddress)
        break;    // Found a block in the next region; end loop.

    // We have a block contained in the region.

    // The following if statement is for detecting stacks in Windows 98.
    // A Windows 98 stack region's last 4 blocks look like this:
    // reserved block, no access block, read-write block, reserved block
    if (pVMQHelp->dwRgnBlocks < 4) {
        // 0th through 3rd block, remember the block's protection
        dwProtectBlock[pVMQHelp->dwRgnBlocks] =
            (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
    } else {
        // We've seen 4 blocks in this region.
        // Shift the protection values down in the array.
        MoveMemory(&dwProtectBlock[0], &dwProtectBlock[1],
            sizeof(dwProtectBlock) - sizeof(DWORD));

        // Add the new protection value to the end of the array.
        dwProtectBlock[3] = (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
    }

    pVMQHelp->dwRgnBlocks++;                // Add another block to the region
    pVMQHelp->RgnSize += mbi.RegionSize;    // Add block's size to region size

    // If block has PAGE_GUARD attribute, add 1 to this counter
    if ((mbi.Protect & PAGE_GUARD) == PAGE_GUARD)
        pVMQHelp->dwRgnGuardBlks++;

    // Take a best guess as to the type of physical storage committed to the
    // block. This is a guess because some blocks can convert from MEM_IMAGE
    // to MEM_PRIVATE or from MEM_MAPPED to MEM_PRIVATE; MEM_PRIVATE can
    // always be overridden by MEM_IMAGE or MEM_MAPPED.
    if (pVMQHelp->dwRgnStorage == MEM_PRIVATE)
        pVMQHelp->dwRgnStorage = mbi.Type;

    // Get the address of the next block.
    pvAddressBlk = (PVOID)((PBYTE)pvAddressBlk + mbi.RegionSize);
}

// After examining the region, check to see whether it is a thread stack
// Windows 2000: Assume stack if region has at least 1 PAGE_GUARD block
// Windows 9x: Assume stack if region has at least 4 blocks with
// 3rd block from end: reserved
```

```

//          2nd block from end: PAGE_NOACCESS
//          1st block from end: PAGE_READWRITE
//          block at end: another reserved block.
pVMQHelp->fRgnIsAStack =
    (pVMQHelp->dwRgnGuardBlks > 0)      ||
    ((pVMQHelp->dwRgnBlocks >= 4)        &&
    (dwProtectBlock[0] == 0)            &&
    (dwProtectBlock[1] == PAGE_NOACCESS) &&
    (dwProtectBlock[2] == PAGE_READWRITE) &&
    (dwProtectBlock[3] == 0));

return(TRUE);
}

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

BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ) {
    if (gs_dwAllocGran == 0) {
        // Set allocation granularity if this is the first call
        SYSTEM_INFO sinf;
        GetSystemInfo(&sinf);
        gs_dwAllocGran = sinf.dwAllocationGranularity;
    }

    ZeroMemory(pVMQ, sizeof(*pVMQ));

    // Get the MEMORY_BASIC_INFORMATION for the passed address.
    MEMORY_BASIC_INFORMATION mbi;
    BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
        == sizeof(mbi));

    if (!fOk)
        return(fOk); // Bad memory address, return failure

    // The MEMORY_BASIC_INFORMATION structure contains valid information.
    // Time to start setting the members of our own VMQUERY structure.

    // First, fill in the block members. We'll fill the region members later.
    switch (mbi.State) {
        case MEM_FREE: // Free block (not reserved)
            pVMQ->pvBlkBaseAddress = NULL;
            pVMQ->BlkSize = 0;
            pVMQ->dwBlkProtection = 0;
            pVMQ->dwBlkStorage = MEM_FREE;
            break;

        case MEM_RESERVE: // Reserved block without committed storage in it.
            pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
            pVMQ->BlkSize = mbi.RegionSize;
            // For an uncommitted block, mbi.Protect is invalid. So we will
            // show that the reserved block inherits the protection attribute
            // of the region in which it is contained.
            pVMQ->dwBlkProtection = mbi.AllocationProtect;

```

```

    pVMQ->dwBlkStorage = MEM_RESERVE;
    break;

case MEM_COMMIT:    // Reserved block with committed storage in it.
    pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
    pVMQ->BlkSize = mbi.RegionSize;
    pVMQ->dwBlkProtection = mbi.Protect;
    pVMQ->dwBlkStorage = mbi.Type;
    break;

default:
    DebugBreak();
    break;
}

// Now fill in the region data members.
VMQUERY_HELP VMQHelp;
switch (mbi.State) {
    case MEM_FREE:    // Free block (not reserved)
        pVMQ->pvRgnBaseAddress = mbi.BaseAddress;
        pVMQ->dwRgnProtection = mbi.AllocationProtect;
        pVMQ->RgnSize = mbi.RegionSize;
        pVMQ->dwRgnStorage = MEM_FREE;
        pVMQ->dwRgnBlocks = 0;
        pVMQ->dwRgnGuardBlks = 0;
        pVMQ->fRgnIsAStack = FALSE;
        break;

    case MEM_RESERVE: // Reserved block without committed storage in it.
        pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
        pVMQ->dwRgnProtection = mbi.AllocationProtect;

        // Iterate through all blocks to get complete region information.
        VMQueryHelp(hProcess, pvAddress, &VMQHelp);

        pVMQ->RgnSize = VMQHelp.RgnSize;
        pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
        pVMQ->dwRgnBlocks = VMQHelp.dwRgnBlocks;
        pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
        pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
        break;

    case MEM_COMMIT: // Reserved block with committed storage in it.
        pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
        pVMQ->dwRgnProtection = mbi.AllocationProtect;

        // Iterate through all blocks to get complete region information.
        VMQueryHelp(hProcess, pvAddress, &VMQHelp);

        pVMQ->RgnSize = VMQHelp.RgnSize;
        pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
        pVMQ->dwRgnBlocks = VMQHelp.dwRgnBlocks;
        pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
        pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
        break;

default:
    DebugBreak();
}

```

```

        break;
    }

    return(fOk);
}

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

```

VMQuery.h

```

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

typedef struct {
    // Region information
    PVOID pvRgnBaseAddress;
    DWORD dwRgnProtection; // PAGE_*
    SIZE_T RgnSize;
    DWORD dwRgnStorage; // MEM_*: Free, Image, Mapped, Private
    DWORD dwRgnBlocks;
    DWORD dwRgnGuardBlks; // If > 0, region contains thread stack
    BOOL fRgnIsAStack; // TRUE if region contains thread stack
    // Block information
    PVOID pvBlkBaseAddress;
    DWORD dwBlkProtection; // PAGE_*
    SIZE_T BlkSize;
    DWORD dwBlkStorage; // MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;

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

BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ);

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

```

14.3.2 虚拟内存表示例应用程序

清单14-4列出的VMMMap应用程序（14 VMMMap.exe）列出了一个进程的地址空间，并且显示了各个地址空间区域和区域中的地址块。该应用程序的源代码和资源文件均位于本书所附光盘上的14-VMMMap目录下。当启动该应用程序时，就会出现图14-6所示窗口。

我使用该应用程序的列表框的内容来生成第13章中的表13-3、表13-5和表13-6中显示的虚拟内存表交换信息。该列表框中的每一项都显示了调用 VMQuery函数所获得的信息的结果。Refresh函数中的主要循环类似下面的样子：

```

BOOL fOk = TRUE;
PVOID pvAddress = NULL;

:

```

```

while (fOk) {
    VMQUERY vmq;
    fOk = VMQuery(hProcess, pvAddress, &vmq);

    if (fOk) {
        // Construct the line to be displayed, and add it to the list box.
        TCHAR szLine[1024];
        ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
        ListBox_AddString(hwndLB, szLine);

        if (fExpandRegions) {
            for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
                dwBlock++) {

                ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
                ListBox_AddString(hwndLB, szLine);

                // Get the address of the next region to test.
                pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
                if (dwBlock < vmq.dwRgnBlocks - 1) {
                    // Don't query the memory info after the last block.
                    fOk = VMQuery(hProcess, pvAddress, &vmq);
                }
            }
        }

        // Get the address of the next region to test.
        pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
    }
}

```

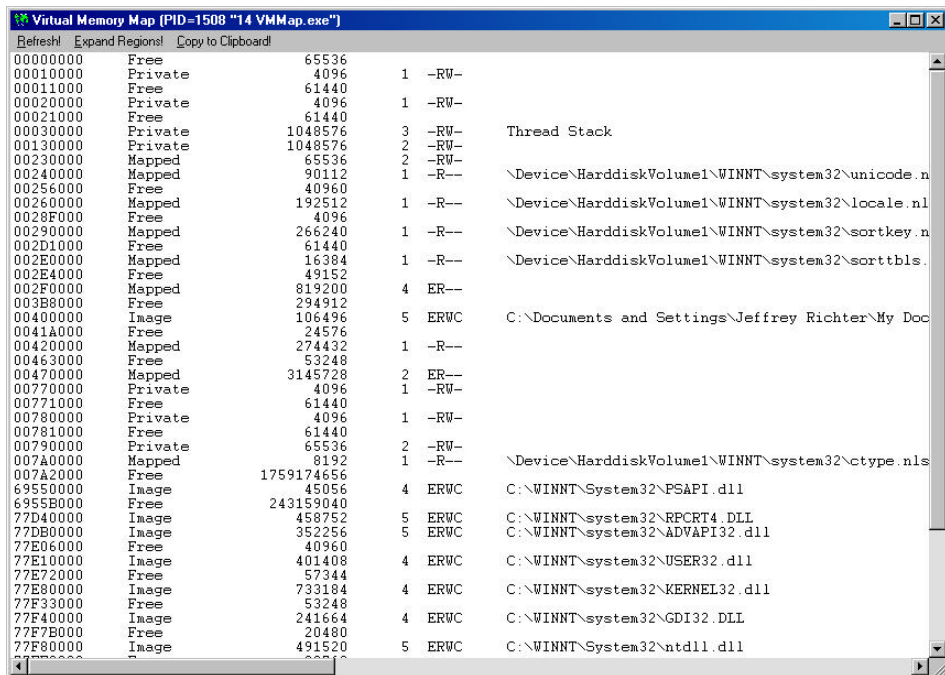


图14-6 启动VMMMap时出现的窗口

该循环从虚拟地址NULL开始运行，当VMQuery函数返回FALSE时结束，表示它不再能够通过该进程的地址空间。该循环的每个重复操作，均有一个对 ConstructRgnInfoLine函数的调用，该函数将关于区域的信息填入一个字符缓存，然后这些信息被附加给列表框。

在这个主循环中，有一个嵌套的循环，它重复通过该地址空间区域中的每个地址块，该循环的每次重复操作均调用 ConstructRgnInfoLine，以使用关于该区域的块的信息填入字符缓存，然后这些信息被附加给列表框。使用 VMQuery函数，可以比较容易地通过进程的地址空间。

清单14-4 VMMMap应用程序的程序清单



VMMMap.cpp

Module: VMMMap.cpp

Notices: Copyright (c) 2000 Jeffrey Richter

*****/

```
#include "..\CmnHdr.h"      /* See Appendix A. */
#include <psapi.h>
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>          // For sprintf
#include "..\04-ProcessInfo\Toolhelp.h"
#include "Resource.h"
#include "VMQuery.h"
```

//

```
DWORD g_dwProcessId = 0; // Which process to walk?
BOOL g_fExpandRegions = FALSE;
CToolhelp g_toolhelp;
```

```
// GetMappedFileName is only on Windows 2000 in PSAPI.DLL
// If this function exists on the host system, we'll use it
typedef DWORD (WINAPI* PFNGETMAPPEDFILENAME)(HANDLE, PVOID, PTSTR, DWORD);
static PFNGETMAPPEDFILENAME g_pfnGetMappedFileName = NULL;
```

//

```
// I use this function to obtain the dump figures in the book.
```

```
void CopyControlToClipboard(HWND hwnd) {
    TCHAR szClipData[128 * 1024] = { 0 };

    int nCount = ListBox_GetCount(hwnd);
    for (int nNum = 0; nNum < nCount; nNum++) {
        TCHAR szLine[1000];
        ListBox_GetText(hwnd, nNum, szLine);
        _tcscat(szClipData, szLine);
        _tcscat(szClipData, TEXT("\r\n"));
    }
}
```

```

OpenClipboard(NULL);
EmptyClipboard();

// Clipboard accepts only data that is in a block allocated
// with GlobalAlloc using the GMEM_MOVEABLE and GMEM_DDESHARE flags.
HGLOBAL hClipData = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
    sizeof(TCHAR) * (_tcslen(szClipData) + 1));
PTSTR pClipData = (PTSTR) GlobalLock(hClipData);

_tcscpy(pClipData, szClipData);

#ifdef UNICODE
    BOOL fOk = (SetClipboardData(CF_UNICODETEXT, hClipData) == hClipData);
#else
    BOOL fOk = (SetClipboardData(CF_TEXT, hClipData) == hClipData);
#endif
    CloseClipboard();

    if (!fOk) {
        GlobalFree(hClipData);
        chMB("Error putting text on the clipboard");
    }
}

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

PCTSTR GetMemStorageText(DWORD dwStorage) {

    PCTSTR p = TEXT("Unknown");
    switch (dwStorage) {
    case MEM_FREE:    p = TEXT("Free  "); break;
    case MEM_RESERVE: p = TEXT("Reserve"); break;
    case MEM_IMAGE:   p = TEXT("Image  "); break;
    case MEM_MAPPED:  p = TEXT("Mapped "); break;
    case MEM_PRIVATE: p = TEXT("Private"); break;
    }
    return(p);
}

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

PCTSTR GetProtectText(DWORD dwProtect, PTSTR szBuf, BOOL fShowFlags) {

    PCTSTR p = TEXT("Unknown");
    switch (dwProtect & ~(PAGE_GUARD | PAGE_NOCACHE | PAGE_WRITECOMBINE)) {
    case PAGE_READONLY:    p = TEXT("-R--"); break;
    case PAGE_READWRITE:   p = TEXT("-RW-"); break;
    case PAGE_WRITECOPY:   p = TEXT("-RWC"); break;
    case PAGE_EXECUTE:     p = TEXT("E---"); break;
    case PAGE_EXECUTE_READ: p = TEXT("ER--"); break;
    case PAGE_EXECUTE_READWRITE: p = TEXT("ERW-"); break;
    }
}

```



```

case PAGE_EXECUTE_WRITECOPY: p = TEXT("ERWC"); break;
case PAGE_NOACCESS:          p = TEXT("----"); break;
}
_tcscpy(szBuf, p);
if (fShowFlags) {
    _tcscat(szBuf, TEXT(" "));
    _tcscat(szBuf, (dwProtect & PAGE_GUARD)      ? TEXT("G") : TEXT("-"));
    _tcscat(szBuf, (dwProtect & PAGE_NOCACHE)     ? TEXT("N") : TEXT("-"));
    _tcscat(szBuf, (dwProtect & PAGE_WRITECOMBINE) ? TEXT("W") : TEXT("-"));
}
return(szBuf);
}

```

//

```

void ConstructRgnInfoLine(HANDLE hProcess, PVMQUERY pVMQ,
    PTSTR szLine, int nMaxLen) {
    _stprintf(szLine, TEXT("%p    %s %16u "),
        pVMQ->pvRgnBaseAddress,
        GetMemStorageText(pVMQ->dwRgnStorage),
        pVMQ->RgnSize);

    if (pVMQ->dwRgnStorage != MEM_FREE) {
        wsprintf(_tcschr(szLine, 0), TEXT("%5u "), pVMQ->dwRgnBlocks);
        GetProtectText(pVMQ->dwRgnProtection, _tcschr(szLine, 0), FALSE);
    }

    _tcscat(szLine, TEXT("    "));

    // Try to obtain the module pathname for this region.
    int nLen = _tcslen(szLine);
    if (pVMQ->pvRgnBaseAddress != NULL) {
        MODULEENTRY32 me = { sizeof(me) };

        if (g_toolhelp.ModuleFind(pVMQ->pvRgnBaseAddress, &me)) {
            lstrcat(&szLine[nLen], me.szExePath);
        } else {
            // This is not a module; see if it's a memory-mapped file
            if (g_pfnGetMappedFileName != NULL) {
                DWORD d = g_pfnGetMappedFileName(hProcess,
                    pVMQ->pvRgnBaseAddress, szLine + nLen, nMaxLen - nLen);
                if (d == 0) {
                    // NOTE: GetMappedFileName modifies the string when it fails
                    szLine[nLen] = 0;
                }
            }
        }
    }

    if (pVMQ->fRgnIsAStack) {
        _tcscat(szLine, TEXT("Thread Stack"));
    }
}

```

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

```
void ConstructBlkInfoLine(PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {
    _stprintf(szLine, TEXT("    %p %s %16u    "),
        pVMQ->pvBlkBaseAddress,
        GetMemStorageText(pVMQ->dwBlkStorage),
        pVMQ->BlkSize);

    if (pVMQ->dwBlkStorage != MEM_FREE) {
        GetProtectText(pVMQ->dwBlkProtection, _tcschr(szLine, 0), TRUE);
    }
}
```

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

```
void Refresh(HWND hwndLB, DWORD dwProcessId, BOOL fExpandRegions) {

    // Delete contents of list box & add a horizontal scroll bar
    ListBox_ResetContent(hwndLB);
    ListBox_SetHorizontalExtent(hwndLB, 300 * LOWORD(GetDialogBaseUnits()));

    // Is the process still running?
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION,
        FALSE, dwProcessId);

    if (hProcess == NULL) {
        ListBox_AddString(hwndLB, TEXT("")); // Blank line, looks better
        ListBox_AddString(hwndLB,
            TEXT("    The process ID identifies a process that is not running"));
        return;
    }

    // Grab a new snapshot of the process
    g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, dwProcessId);

    // Walk the virtual address space, adding entries to the list box.
    BOOL fOk = TRUE;
    PVOID pvAddress = NULL;

    SetWindowRedraw(hwndLB, FALSE);
    while (fOk) {

        VMQUERY vmq;
        fOk = VMQuery(hProcess, pvAddress, &vmq);
        if (fOk) {
            // Construct the line to be displayed, and add it to the list box.
            TCHAR szLine[1024];
            ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
            ListBox_AddString(hwndLB, szLine);

            if (fExpandRegions) {
```

```

    for (DWORD dwBlock = 0; f0k && (dwBlock < vmq.dwRgnBlocks);
        dwBlock++) {

        ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
        ListBox_AddString(hwndLB, szLine);

        // Get the address of the next region to test.
        pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
        if (dwBlock < vmq.dwRgnBlocks - 1) {
            // Don't query the memory info after the last block.
            f0k = VMQuery(hProcess, pvAddress, &vmq);
        }
    }

    // Get the address of the next region to test.
    pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
}

SetWindowRedraw(hwndLB, TRUE);
CloseHandle(hProcess);

```

//

```

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

    chSETDLGICONS(hwnd, IDI_VMMAP);

    // Show which process we're walking in the window's caption
    TCHAR szCaption[MAX_PATH * 2];
    GetWindowText(hwnd, szCaption, chDIMOF(szCaption));
    g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, g_dwProcessId);
    PROCESSENTRY32 pe = { sizeof(pe) };
    wsprintf(&szCaption[lstrlen(szCaption)], TEXT(" (PID=%u \\"%s\\")"),
        g_dwProcessId, g_toolhelp.ProcessFind(g_dwProcessId, &pe)
        ? pe.szExeFile : TEXT("unknown"));
    SetWindowText(hwnd, szCaption);

    // VMMAP has so much info to show, let's maximize it by default
    ShowWindow(hwnd, SW_MAXIMIZE);

    // Force the list box to refresh itself
    Refresh(GetDlgItem(hwnd, IDC_LISTBOX), g_dwProcessId, g_fExpandRegions);
    return(TRUE);
}

```

//

```

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

    // The list box always fills the whole client area

```

```

        SetWindowPos(GetDlgItem(hwnd, IDC_LISTBOX), NULL, 0, 0, cx, cy,
        SWP_NOZORDER);
    }

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

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

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

        case ID_REFRESH:
            Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
                g_dwProcessId, g_fExpandRegions);
            break;

        case ID_EXPANDREGIONS:
            g_fExpandRegions = g_fExpandRegions ? FALSE: TRUE;
            Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
                g_dwProcessId, g_fExpandRegions);
            break;

        case ID_COPYTOCLIPBOARD:
            CopyControlToClipboard(GetDlgItem(hwnd, IDC_LISTBOX));
            break;
    }
}

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

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

    switch (uMsg) {
        case WM_INITDIALOG: Dlg_OnInitDialog();
        case WM_COMMAND:    Dlg_OnCommand();
        case WM_SIZE:       Dlg_OnSize();
    }
    return(FALSE);
}

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

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

    CToolhelp::EnableDebugPrivilege();

    // Try to load PSAPI.DLL and get the address of GetMappedFileName
    HMODULE hmodPSAPI = LoadLibrary(TEXT("PSAPI"));
    if (hmodPSAPI != NULL) {
#ifdef UNICODE

```

```

        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
        GetProcAddress(hmodPSAPI, "GetMappedFileNameW");
    #else
        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
        GetProcAddress(hmodPSAPI, "GetMappedFileNameA");
    #endif
    }

    g_dwProcessId = _ttoi(pszCmdLine);
    if (g_dwProcessId == 0) {
        g_dwProcessId = GetCurrentProcessId();
    }

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMMAP), NULL, Dlg_Proc);
    if (hmodPSAPI != NULL)
        FreeLibrary(hmodPSAPI); // Free PSAPI.DLL if we loaded it

    return(0);
}

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

```

VMMMap.rc

```

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

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

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

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

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

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

1 TEXTINCLUDE DISCARDABLE

```

```

BEGIN
    "Resource.h\0"
END

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

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

#endif    // APSTUDIO_INVOKED

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

IDD_VMMAP_DIALOG DISCARDABLE 10, 18, 250, 250
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU | WS_THICKFRAME
CAPTION "Virtual Memory Map"
MENU IDR_VMMAP
FONT 8, "Courier"
BEGIN
    LISTBOX            IDC_LISTBOX,0,0,248,248,LBS_NOINTEGRALHEIGHT | NOT
                        WS_BORDER | WS_VSCROLL | WS_HSCROLL | WS_GROUP |
                        WS_TABSTOP
END

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

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMMAP            ICON    DISCARDABLE    "VMMMap.Ico"

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

IDR_VMMAP MENU DISCARDABLE
BEGIN
    MENUITEM "&Refresh!",                ID_REFRESH
    MENUITEM "&Expand Regions!",          ID_EXPANDREGIONS

```

```
MENUITEM "&Copy to Clipboard!",          ID_COPYTOCLIPBOARD
END

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

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

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