Windows操作系统

C/C++ 程序实验

首都师范大学 信息工程学院

系统分析与管理实验室

## 实验六 Windows 虚拟内存

### 一、背景知识

在Windows 环境下，4GB的虚拟地址空间被划分成两个部分：低端2GB提供给进程使用，高端2GB提供给系统使用。这意味着用户的应用程序代码，包括DLL以及进程使用的各种数据等，都装在用户进程地址空间内 (低端2GB) 。用户过程的虚拟地址空间也被分成三部分：

1) 虚拟内存的已调配区 (committed) ：具有备用的物理内存，根据该区域设定的访问权限，用户可以进行写、读或在其中执行程序等操作。

2) 虚拟内存的保留区 (reserved) ：没有备用的物理内存，但有一定的访问权限。

3) 虚拟内存的自由区 (free) ：不限定其用途，有相应的PAGE\_NOACCESS权限。

与虚拟内存区相关的访问权限告知系统进程可在内存中进行何种类型的操作。例如，用户不能在只有PAGE\_READONLY权限的区域上进行写操作或执行程序；也不能在只有PAGE\_EXECUTE权限的区域里进行读、写操作。而具有PAGE\_ NOACCESS权限的特殊区域，则意味着不允许进程对其地址进行任何操作。

在进程装入之前，整个虚拟内存的地址空间都被设置为只有PAGE\_NOACCESS权限的自由区域。当系统装入进程代码和数据后，才将内存地址的空间标记为已调配区或保留区，并将诸如EXECUTE、READWRITE和READONLY的权限与这些区域相关联。

清单6-1还显示了如何理解Virtual QueryEX() API填充的MEMORY\_BASIC\_ INFORMATION结构，如表6-l所示。此数据描述了进程虚拟内存空间中一组虚拟内存页面的当前状态。其中State项表明这些区域是否为自由区、已调配区或保留区；Protect项则包含了Windows系统为这些区域添加了何种访问保护；Type项则表明这些区域是可执行图像、内存映射文件还是简单的私有内存。VirtualQueryEX() API能让用户在指定的进程中，对虚拟内存地址的大小和属性进行检测。

Windows还提供了一整套能使用户精确控制应用程序的虚拟地址空间的虚拟内存API。一些用于虚拟内存操作及检测的API见表6-1所示。

**表6-1 MEMORY\_BASIC\_INFORMATION结构的成员**

|  |  |
| --- | --- |
| **成员名称** | **目的** |
| PVOID BaseAddress | 虚拟内存区域开始处的指针 |
| PVOID AllocationBase | 如果这个特定的区域为子分配区的话，则为虚拟内存外面区域的指针；否则此值与BaseAddress相同 |
| DWORD AllocationProtect | 虚拟内存最初分配区域的保护属性。其可能值包括：  PAGE\_NOACCESS，PAGE\_READONLY，PAGE\_READWRITE和PAGE \_EXECUTE\_READ |
| DWORD RegionSize | 虚拟内存区域的字节数 |
| DWORD State | 区域的当前分配状态。其可能值为MEM\_COMMIT，MEM\_FREE和MEM\_RESERVE |
| DWORD Protect | 虚拟内存当前区域的保护属性。可能值与AllocationProtect成员的相同 |
| DWORD Type | 虚拟内存区域中出现的页面类型。可能值为MEM\_IMAGE， MEM\_MAPPED和MEM\_PRIVATE |

**表6-2 虚拟内存的API**

|  |  |
| --- | --- |
| **API名称** | **描 述** |
| VirtualQueryEx() | 通过填充MEMORY\_BASIC\_INFORMATION结构检测进程内虚拟内存的区域 |
| VirtualAlloc() | 保留或调配进程的部分虚拟内存，设置分配和保护标志 |
| VirtualFree() | 释放或收回应用程序使用的部分虚拟地址 |
| VirtualProtect() | 改变虚拟内存区域保护规范 |
| VirtualLock() | 防止系统将虚拟内存区域通过系统交换到页面文件中 |
| VirtualUnlock() | 释放虚拟内存的锁定区域，必要时，允许系统将其交换到页面文件中 |

提供虚拟内存分配功能的是VirtualAlloc() API。该API支持用户向系统要求新的虚拟内存或改变已分配内存的当前状态。用户若想通过VirtualAlloc() 函数使用虚拟内存，可以采用两种方式通知系统：

1) 简单地将内存内容保存在地址空间内；

2) 请求系统返回带有物理存储区 (RAM的空间或换页文件) 的部分地址空间。

用户可以用flAllocation Type参数 (commit和reserve) 来定义这些方式，用户可以通知Windows按只读、读写、不可读写、执行或特殊方式来处理新的虚拟内存。

与VirtualAlloc() 函数对应的是VirtualFree() 函数，其作用是释放虚拟内存中的已调配页或保留页。用户可利用dwFree Type参数将已调配页修改成保留页属性。

VirtualProtect() 是VirtualAlloc() 的一个辅助函数，利用它可以改变虚拟内存区的保护规范。

### 二、实验目的

1) 通过实验了解Windows 内存的使用，学习如何在应用程序中管理内存，体会Windows应用程序内存的简单性和自我防护能力。

2) 学习检查虚拟内存空间或对其进行操作；

3) 了解Windows 的内存结构和虚拟内存的管理，进而了解进程堆和Windows为使用内存而提供的一些扩展功能。

### 三、工具/准备工作

在开始本实验之前，请回顾教科书的相关内容。

您需要做以下准备：

1) 一台运行Windows 操作系统的计算机。

2) 计算机中需安装Visual Studio Code。

### 四、实验内容与步骤

#### 1. 虚拟内存的检测

清单6-1所示程序使用VirtualQueryEX() 函数来检查虚拟内存空间。

**步骤1**：登录进入Windows 。

**步骤2**：在“开始”菜单中单击“程序-Microsoft Visual Studio Code”。

**步骤3：**新建项目名为“6-1”，并且新建项“6-1.cpp”。

**清单6-1 检测进程的虚拟地址空间**

// 工程vmwalker

# include <windows.h>

# include <iostream>

# include <shlwapi.h>

# include <iomanip>

# pragma comment(lib, "Shlwapi.lib")

  // 以可读方式对用户显示保护的辅助方法。

// 保护标记表示允许应用程序对内存进行访问的类型

// 以及操作系统强制访问的类型

inline bool TestSet(DWORD dwTarget, DWORD dwMask)

{

return ((dwTarget & dwMask) == dwMask) ;

}

# define SHOWMASK(dwTarget, type) \

if (TestSet(dwTarget, PAGE\_##type) ) \

{std :: cout << ", " << #type; }

void ShowProtection(DWORD dwTarget)

{

SHOWMASK(dwTarget, READONLY) ;

SHOWMASK(dwTarget, GUARD) ;

SHOWMASK(dwTarget, NOCACHE) ;

SHOWMASK(dwTarget, READWRITE) ;

SHOWMASK(dwTarget, WRITECOPY) ;

SHOWMASK(dwTarget, EXECUTE) ;

SHOWMASK(dwTarget, EXECUTE\_READ) ;

SHOWMASK(dwTarget, EXECUTE\_READWRITE) ;

SHOWMASK(dwTarget, EXECUTE\_WRITECOPY) ;

SHOWMASK(dwTarget, NOACCESS) ;

}

  // 遍历整个虚拟内存并对用户显示其属性的工作程序的方法

void WalkVM(HANDLE hProcess)

{

// 首先，获得系统信息

SYSTEM\_INFO si;

:: ZeroMemory(&si, sizeof(si) ) ;

:: GetSystemInfo(&si) ;

  // 分配要存放信息的缓冲区

MEMORY\_BASIC\_INFORMATION mbi;

:: ZeroMemory(&mbi, sizeof(mbi) ) ;

  // 循环整个应用程序地址空间

LPCVOID pBlock = (LPVOID) si.lpMinimumApplicationAddress;

while (pBlock < si.lpMaximumApplicationAddress)

{

// 获得下一个虚拟内存块的信息

if (:: VirtualQueryEx(

hProcess, // 相关的进程

pBlock, // 开始位置

&mbi, // 缓冲区

sizeof(mbi))==sizeof(mbi) ) // 大小的确认

{

// 计算块的结尾及其大小

LPCVOID pEnd = (PBYTE) pBlock + mbi.RegionSize;

TCHAR szSize[MAX\_PATH];

:: StrFormatByteSize(mbi.RegionSize, szSize, MAX\_PATH) ;

  // 显示块地址和大小

std :: cout.fill ('0') ;

std :: cout

<< std :: hex << std :: setw(8) << (DWORD) pBlock << "-"

<< std :: hex << std :: setw(8) << (DWORD) pEnd

<< (:: strlen(szSize)==7? " (" : " (") << szSize << ") " ;

  // 显示块的状态

switch(mbi.State)

{

case MEM\_COMMIT:

std :: cout << "Committed" ;

break;

case MEM\_FREE:

std :: cout << "Free" ;

break;

case MEM\_RESERVE:

std :: cout << "Reserved" ;

break;

}

// 显示保护

if (mbi.Protect==0 && mbi.State!=MEM\_FREE)

{

mbi.Protect=PAGE\_READONLY;

}

ShowProtection(mbi.Protect);

  // 显示类型

switch (mbi.Type)

{

case MEM\_IMAGE:

std :: cout << ", Image" ;

break;

case MEM\_MAPPED:

std :: cout << ", Mapped" ;

break;

case MEM\_PRIVATE:

std :: cout << ", Private" ;

break;

}

  // 检验可执行的影像

TCHAR szFilename [MAX\_PATH] ;

if ( :: GetModuleFileName (

(HMODULE) pBlock, // 实际虚拟内存的模块句柄

szFilename, // 完全指定的文件名称

MAX\_PATH)>0) // 实际使用的缓冲区大小

{

// 除去路径并显示

:: PathStripPath(szFilename) ;

std :: cout << ", Module: " << szFilename;

}

  std :: cout << std :: endl;

// 移动块指针以获得下一下个块

pBlock = pEnd;

}

}

}

  void main()

{

// 遍历当前进程的虚拟内存

::WalkVM(::GetCurrentProcess());

system("pause");

}

清单6-1中显示一个WalkVM() 函数开始于某个进程可访问的最低端虚拟地址处，并在其中显示各块虚拟内存的特性。虚拟内存中的块由VirtualQueryEx() API定义成连续快或具有相同状态 (自由区、已调配区等等) 的内存，并分配以一组统一的保护标志 (只读、可执行等等) 。

**步骤4：**在代码宏定义里添加 #define \_CRT\_SECURE\_NO\_WARNINGS

**步骤5：**按“F5”开始调试，注意路径里不要含有中文。

**步骤6：**按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。



操作能否正常进行？如果不行，则可能的原因是什么？

操作能正常进行，如果不行可能是因为路径中包含中文或代码中包含中文字符。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. 分析运行结果 (如果运行不成功，则可能的原因是什么？) 。

文本

描述已自动生成

图片 1 运行结果如图

按committed、reserved、free等三种虚拟地址空间分别记录实验数据。其中“描述”是指对该组数据的简单描述，例如，对下列一组数据：

00010000 – 00012000 <8.00KB> Committed, READWRITE, Private

可描述为：具有READWRITE权限的已调配私有内存区。

将系统当前的自由区 (free) 虚拟地址空间填入表6-3中。

**表6-3 实验记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 址** | **大小** | **虚拟地址**  **空间类型** | **访问权限** | **描 述** |
| 0000000000010000-000000007ffe0000 | (1.99 GB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| 000000007ffe1000-000000007ffe6000 | (20.0 KB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| 000000007ffe7000-000000fd7de40000 | (3.96 GB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| 000000fd7df40000-000000fd7e000000 | (768 KB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| 000000fd7e400000-00000285e32d0000 | (1.57 GB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| 00000285e330f000-00000285e3310000 | (4.00 KB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |
| ： | ： | ： | ： | ： |
| 00007ffa2bb27000-00007fffffff0000 | (3.31 GB) | free | NOACCESS | 未分配内存空间的自由区，无法访问 |

|  |
| --- |
| **提示：**详细记录实验数据在实验活动中是必要的，但想想是否可以简化记录的办法？ |

将系统当前的已调配区 (committed) 虚拟地址空间填入表6-4中。

**表6-4 实验记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 址** | **大小** | **虚拟地址**  **空间类型** | **访问权限** | **描 述** |
| 000000007ffe0000-000000007ffe1000 | (4.00 KB) | committed | READONLY | 具有READONLY（只读）权限的已调配私有内存区。 |
| 000000007ffe6000-000000007ffe7000 | (4.00 KB) | committed | READONLY | 具有READONLY（只读）权限的已调配私有内存区。 |
| 000000fd7df39000-000000fd7df3c000 | (12.0 KB) | committed | GUARD, READWRITE | 具有保护（Guard）特性和READWRITE（读写）权限的已调配私有内存区。 |
| 000000fd7df3c000-000000fd7df40000 | (16.0 KB) | committed | READWRITE | 具有READWRITE（读写）权限的已调配私有内存区。 |
| 32808000-3280f000 | (28.0 KB) | committed | READWRITE | 具有READWRITE（读写）权限的已调配私有内存区。 |
| 000000fd7e2fb000-000000fd7e2fe000 | (12.0 KB) | committed | GUARD, READWRITE | 具有保护（Guard）特性和READWRITE权限的已调配私有内存区。 |
| ： | ： | ： | ： | ： |
| 00007ffa2ba9c000-00007ffa2bb27000 | (556 KB) | committed | READONLY | 包含映像文件的具有READONLY（只读）权限的内存区域。 |

将系统当前的保留区 (reserved) 虚拟地址空间填入表6-5中。

**表6-5 实验记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 址** | **大小** | **虚拟地址**  **空间类型** | **访问权限** | **描 述** |
| 000000fd7de40000-000000fd7df39000 | (996 KB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| 000000fd7e000000-000000fd7e0a1000 | (664 KB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| 000000fd7e0a8000-000000fd7e200000 | (1.34 MB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| 000000fd7e200000-000000fd7e2fb000 | (0.98 MB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| 000000fd7e300000-000000fd7e3fb000 | (0.98 MB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| 00000285e3463000-00000285e3540000 | (884 KB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |
| ： | ： | ： | ： | ： |
| 00007ff4b7b90000-00007ff5b7bb0000 | (128 KB) | reserved | READONLY | 预留的私有内存区域，设为只读权限。 |

2) 从上述输出结果，对照分析6-1程序，请描述程序运行的流程：

程序先通过::GetSystemInfo获取系统信息，其中包括进程可以应用的最大和最小地址范围，再初始化分配要存放信息的缓存区。将常量长指针pBlock初始化为进程的最小地址，并进入while循环，当pBlock没有超出进程的最大应用地址时，遍历虚拟内存。

每次进入while循环使用VirtualQueryEx获取内存块信息，并利用缓冲区mbi大小计算出末尾指针的地址，通过mbi.State获取内存状态（Commited/Free/Reserved）；再显示保护属性如果 mbi.Protect 为 0 并且状态不是 Free，则默认设置为 PAGE\_READONLY；通过mbi.Type显示内存类型（Image/Mapped/Private）；如果内存区域属于某个模块，输出可执行的影像的相关信息。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### 2. 虚拟内存操作

清单6-2的示例显示了如何分配一个大容量空间，将物理存储委托给其中的很小一部分 (千分之一) 并加以使用。

**步骤7**：新建项目名为“6-2”，并且新建项“6-2.cpp”。

**清单6-2 分配和使用大块内存**

// 工程largealloc

# include <windows.h>

# include <iostream>

  // 尝试以指定长度的零数字填充内存块的简便方法

void FillZero(LPVOID pBlock, DWORD dwSize)

{

\_try

{

BYTE\* arFill = (BYTE \*) pBlock;

for (DWORD dwFill = 0; dwFill < dwSize; ++dwFill)

{

arFill [dwFill] = 0;

}

std :: cout << “Memory zeroed.” << std :: endl;

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

std :: cout << “Could not zero memory. ” << std :: endl;

}

}

void main()

{

// 简单的常数

DWORD c\_dwGigabyte = 1 << 30;

DWORD c\_dwMegabyte = 1 << 20;

  // 使用内存分配来获得1GB块

{

LPVOID pBlock = :: malloc(c\_dwGigabyte) ;

:: FillZero (pBlock, c\_dwMegabyte) ;

:: free(pBlock) ;

}

  // 使用虚拟分配以获得物理1GB块

{

LPVOID pBlock = :: VirtualAlloc(

NULL, // 不指定起始地址

c\_dwGigabyte, // 要求1GB

MEM\_COMMIT, // 调配物理存储

PAGE\_READWRITE) ; // 对此的读写操作

:: FillZero(pBlock, c\_dwMegabyte) ;

:: VirtualFree(pBlock, 0, MEM\_RELEASE) ;

}

  // 使用虚拟分配以获得虚拟1GB块

{

LPVOID pBlock = :: VirtualAlloc(

NULL, // 不指定起始地址

c\_dwGigabyte, // 要求1GB

MEM\_RESERVE, // 不调配物理存储

PAGE\_READWRITE) ; // 对此的读写操作

:: FillZero(pBlock, c\_dwMegabyte) ;

:: VirtualFree(pBlock, 0, MEM\_RELEASE) ;

}

  // 使用虚拟分配调配获得虚拟1GB块，再为其调配1MB物理存储

{

LPVOID pBlock = :: VirtualAlloc(

NULL, // 不指定起始地址

c\_dwGigabyte, // 要求1GB

MEM\_RESERVE, // 不调配物理存储

PAGE\_READWRITE) ; // 对此的读写操作

:: VirtualAlloc(

pBlock,

c\_dwMegabyte,

MEM\_COMMIT,

PAGE\_READWRITE) ;

:: FillZero(pBlock, c\_dwMegabyte) ;

:: VirtualFree(pBlock, 0, MEM\_RELEASE) ;

}

system("pause");

}

**步骤8：**按“F5”开始调试，注意路径里不要含有中文。

**步骤9：**按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。



运行结果 (如果运行不成功，则可能的原因是什么？) ：

1. 文件中含有中文双引号，导致运行不成功。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) 文件路径中含有中文，导致运行不成功。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

对照运行结果，分析程序6-2。为了给数据库保留1GB的段地址空间，清单6-2给出了内存分配的四种方法。图形用户界面, 文本

描述已自动生成

· **第一种技术**

即程序中说明为 使用内存分配来获得1GB块

文本

描述已自动生成

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_的程序段，该段程序试图利用标准C中的malloc() 函数，从已经已调配的小内存区获得内存。从运行结果看，这种技术成功了吗？这种技术成功了，分配的1GB内存块被成功以指定长度的零数字填充。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

· **第二种技术**

即程序中说明为  使用虚拟分配以获得物理1GB块文本

描述已自动生成

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_的程序段，该段程序试图通过VirtualAlloc() ，然后利用物理备用内存将整个块分配到虚拟内存空间的任何位置。这种技术只对拥有1GB以上的RAM且都有换页文件的计算机可行。从运行结果看，这种技术成功了吗？这种技术成功了，使用虚拟分配获得的1GB物理内存块被成功以指定长度的零数字填充。\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

· **第三种技术**

即程序中说明为// 使用虚拟分配以获得虚拟1GB块

文本

描述已自动生成\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_的程序段，该段程序利用VirtualAlloc() ，如果函数成功，则获得大块内存，但不将任何物理内存调配到此块中。从运行结果看，这种技术成功了吗？这种技术成功了，因为只分配了虚拟的1GB内存块而未给申请的内存空间调配物理存储，所以无法以指定长度的零数字填充内存块。\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

· **第四种技术**

即程序中说明为 使用虚拟分配调配获得虚拟1GB块，再为其调配1MB物理存储



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_的程序段，该段程序保留1GB的内存区，然后将物理内存调配给其中的很小一部分 (1MB) 。这就是清单6-2介绍的处理一个假想的数据库应用程序的方法：保留整个块，然后按要求在其一小部分内进行读操作，让系统将用过的区域换页到磁盘中。

利用VirtualLock() API，Windows可用来在自己的进程空间中控制虚拟内存的行为。这个函数与其成对的VirtualUnlock() 阻止或允许一块内存从物理RAM中换页和换页到页面文件中。这样就会通知系统有一段特定的内存区要求对用户作出强烈的响应，所以系统不应将其移出RAM。当然，如果要将整个虚拟内存空间锁定，系统就会停留于试图将系统中工作内存的每一小块换页到磁盘。