操作系统 -- 实验五实验报告

实验五：内存管理

一、实验题目

（1）观察和调整 Windows XP/7 的内存性能。

（2）了解和检测进程的虚拟内存空间。

二、实验目的

> （1） 通过对 Windows xp/7“任务管理器”、“计算机管理”、“我的电脑”属性、“系统信息”、“系 统监视器”等程序的应用，学习如何察看和调整 Windows 的内存性能，加深对操作系统内存管理、虚拟存储管理等理论知识的理解。 （2） 了解 Windows xp/7 的内存结构和虚拟内存的管理，理解进程的虚拟内存空间和物理内存的 映射关系。

三、总体设计

1. Windows 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。
2. 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

   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

   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。

程序主要包括：

- main() 函数：控制程序整体执行流程，显示虚拟内存的基本信息，遍历当前进程的虚拟内存。
- ShowVirtualMemory()
- WalkVM()
- ShowProtection()
- TestSet()

四、详细设计

实验代码：

```
// 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,
                pBlock,
                &mbi,
                sizeof(mbi))==sizeof(mbi) )
// 相关的进程
// 开始位置
// 缓冲区
// 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 << "-"
```

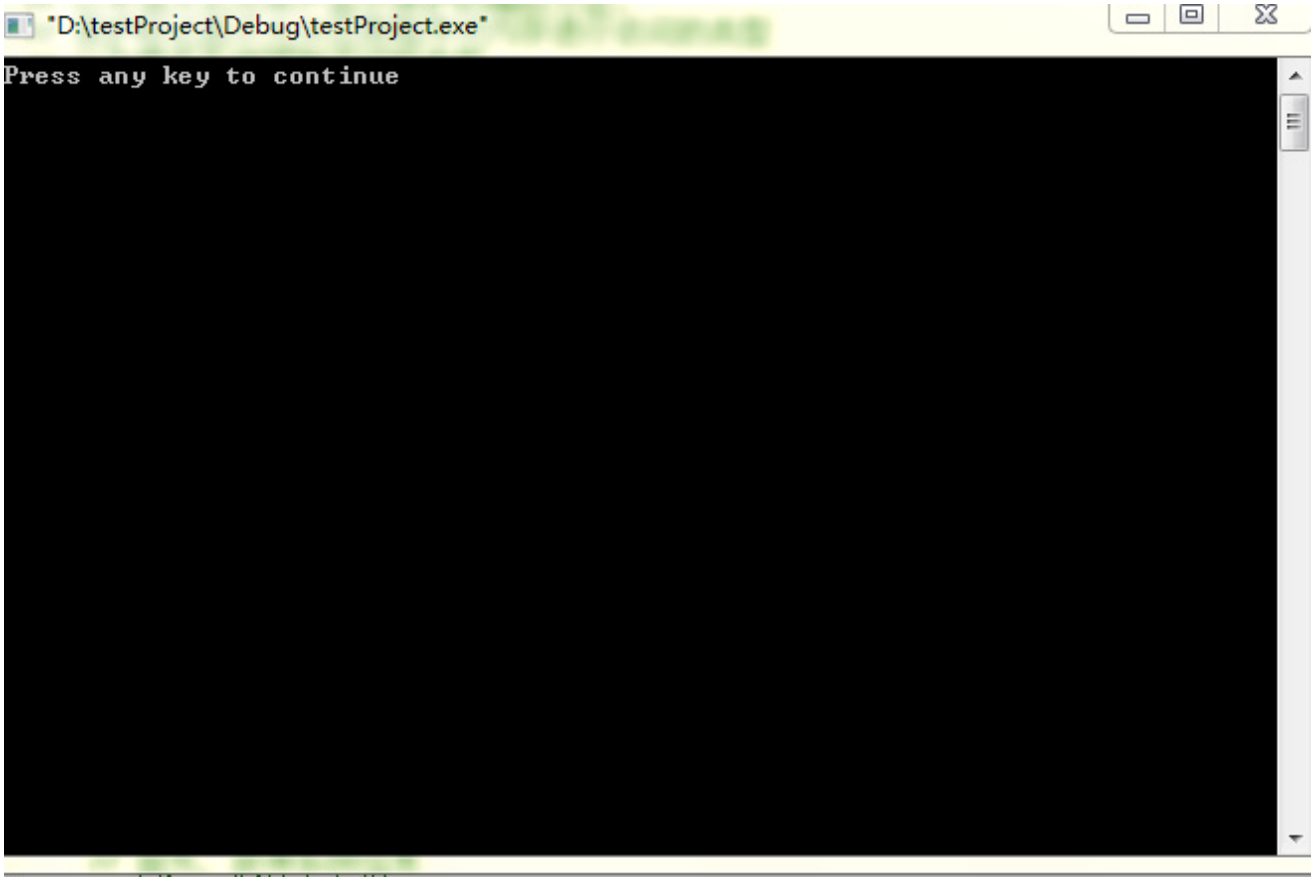
```
                << std :: hex << std :: setw(8) << (DWORD) pEnd << (::
strlen(szSize)==7? " (" : " (") << szSize << ") " ;
// 显示块的状态
                switch(mbi.State)
                {
                case MEM_COMMIT :
                    std :: cout << "Committed" ;
                    break;
                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,
                            szFilename,
                            MAX_PATH)>0)
// 实际虚拟内存的模块句柄
//完全指定的文件名称
//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;
            }

        }
```

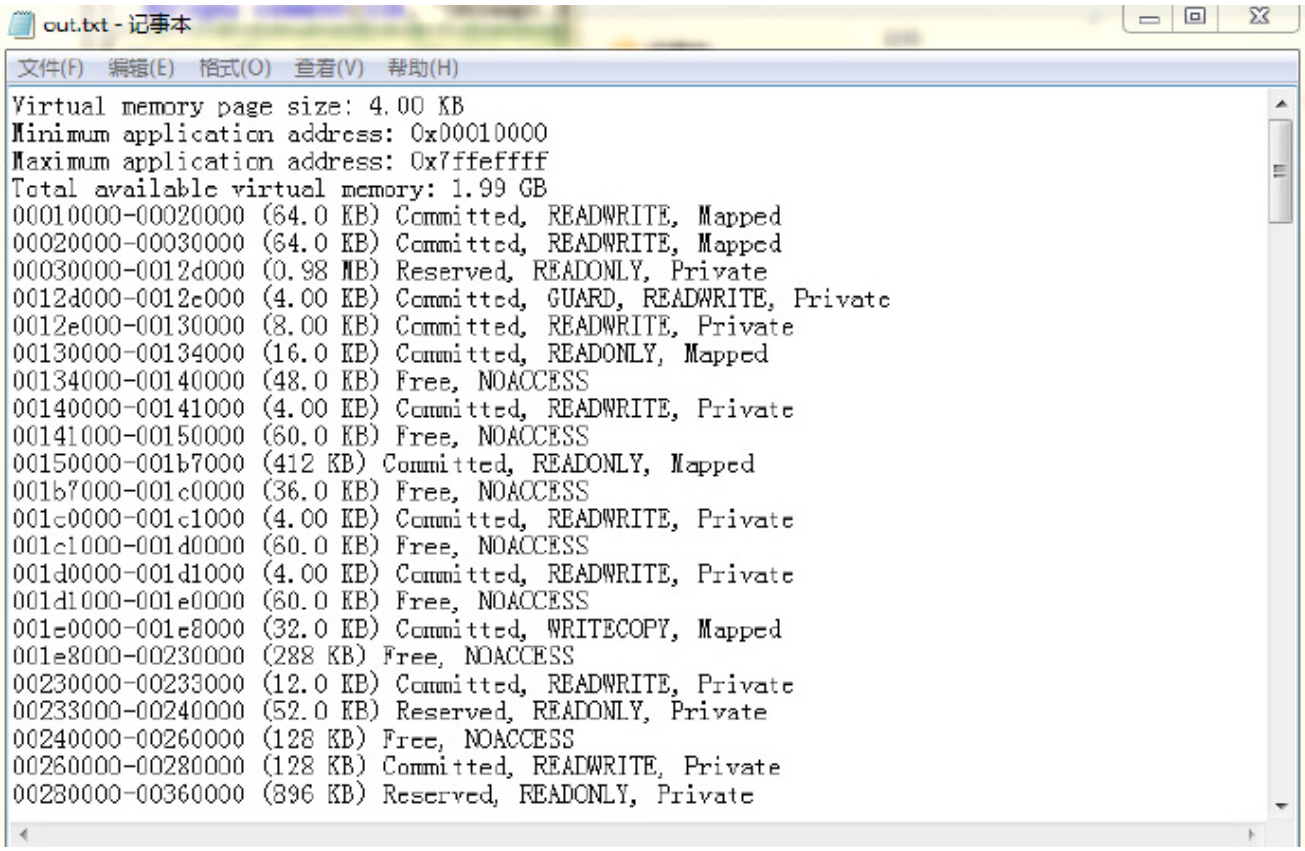
```
    }
    void ShowVirtualMemory()
    {
// 首先，让我们获得系统信息
        SYSTEM_INFO si;
        :: ZeroMemory(&si, sizeof(si) ) ;
        :: GetSystemInfo(&si) ;
// 使用外壳辅助程序对一些尺寸进行格式化
        TCHAR szPageSize[MAX_PATH];
        ::StrFormatByteSize(si.dwPageSize, szPageSize, MAX_PATH) ;
        25
        DWORD dwMemSize = (DWORD)si.lpMaximumApplicationAddress - (DWORD)
si.lpMinimumApplicationAddress;
        TCHAR szMemSize [MAX_PATH] ;
        :: StrFormatByteSize(dwMemSize, szMemSize, MAX_PATH) ; // 将内存信息显示出来
        std :: cout << "Virtual memory page size: " << szPageSize << std :: endl;
        std :: cout.fill ('0') ;
        std :: cout << "Minimum application address: 0x"
                    << std :: hex << std :: setw(8)
                    << (DWORD) si.lpMinimumApplicationAddress
                    << std :: endl;
        std :: cout << "Maximum application address: 0x"
                    << std :: hex << std :: setw(8)
                    << (DWORD) si.lpMaximumApplicationAddress
                    << std :: endl;
        std :: cout << "Total available virtual memory: "
                    << szMemSize << std :: endl ;
    }
    void main()
    {
//显示虚拟内存的基本信息
        ShowVirtualMemory();
// 遍历当前进程的虚拟内存
        ::WalkVM(::GetCurrentProcess());
    }
```

五、实验结果与分析

程序运行截图：



程序新建文件内容截图：



实验结果分析：

> Windows Xp 是 32 位的操作系统，它使计算机 CPU 可以用 32 位地址对 32 位内存块进行操作。 内存中的每一个字节都可以用一个 32 位的指针来寻址。这样，最大的存储空间就是 232字节或 4000 兆字节 (4GB) 。这样，在 Windows 下运行的每一个应用程序最大可能占有 4GB 大小的空间。 然而， 实际上每个进程一般不会占有 4GB 内存。 Windows 在幕后将虚拟内存 (virtual memory, VM) 地址映射到了各进程的物理内存地址上。而所谓物理内存是指计算机的 RAM 和由 Windows 分配到用户驱动器根目录上的换页文件。物理内存完全由系统管理。 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) 。用户进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。

六、小结与心得体会

> （1） 分页过程 当 Windows 求助于硬盘以获得虚拟内存时，这个过程被称为分页 (paging) 。分页就是将信息 从主内存移动到磁盘进行临时存储的过程。 应用程序将物理内存和虚拟内存视为一个独立的实体， 甚至不知道 Windows 使用了两种内存方案，而认为系统拥有比实际内存更多的内存。例如，系统的 内存数量可能只有 256MB，但每一个应用程序仍然认为有 4GB 内存可供使用。 使用分页方案带来了很多好处，不过这是有代价的。当进程需要已经交换到硬盘上的代码或数 据时，系统要将数据送回物理内存，并在必要时将其他信息传输到硬盘上，而硬盘与物理内存在性 能上的差异极大。例如，硬盘的访问时间通常大约为 4-10 毫秒，而物理内存的访问时间为 60 us， 甚至更快。 （2） 内存共享 应用程序经常需要彼此通信和共享信息。为了提供这种能力， Windows 必须允许访问某些内存 空间而不危及它和其他应用程序的安全性和完整性。从性能的角度来看，共享内存的能力大大减少 了应用程序使用的内存数量。运行一个应用程序的多个副本时，每一个实例都可以使用相同的代码 和数据，这意味着不必维护所加载应用程序代码的单独副本并使用相同的内存资源。无论正在运行 多少个应用程序实例，充分支持应用程序代码所需求的内存数量都相对保持不变。 （3） 未分页合并内存与分页合并内存 Windows 决定了系统内存组件哪些可以以及哪些不可以交换到磁盘上。显然，不应该将某些代 码 (例如内核) 交换出主内存。因此， Windows 将系统使用的内存进一步划分为未分页合并内存和 分页合并内存。 分页合并内存是存储迟早需要的可分页代码或数据的内存部分。虽然可以将分页合并内存中的 任何系统进程交换到磁盘上，但是它临时存储在主内存的这一部分，以防系统立刻需要它。在将系 统进程交换到磁盘上之前， Windows 会交换其他进程。 未分页合并内存包含必须驻留在内存中的占用代码或数据。这种结构类似于早期的 MS-DOS 程序使用的结构，在 MS-DOS 中，相对较小的终止并驻留程序 (Terminate and Stay Resident， TSR) 在启动时加载到内存中。这些程序在系统重新启动或关闭之前一直驻留在内存的特定部分中。例如， 防病毒程序将加载为 TSR 程序，以预防可能的病毒袭击。 未分页合并内存中包含的进程保留在主内存中，并且不能交换到磁盘上。物理内存的这个部分 用于内核模式操作（例如，驱动程序）和必须保留在主内存中才能有效工作的其他进程。没有主内 存的这个部分，内核组件就将是可分页的，系统本身就有变得不稳定的危险。 分配到未分页内存池的主内存数量取决于服务器拥有的物理内存数量以及进程对系统上的内 存地空间的需求。不过， Windows XP 将未分页合并内存限制为 256MB (在 Windows NT 4 中的限制 为 128MB) 。根据系统中的物理内存数量，复杂的算法在启动时动态确定 Windows XP 系统上的未 分页合并内存的最大数量。 Windows XP 内部的这一自我调节机制可以根据当前的内存配置自动调 整大小。例如，如果增加或减少系统中的内存数量，那么 Windows Xp 将自动调整未分页合并内存 的大小，以反映这一更改。