操作系统课程设计实验报告

实验名称：　　内存监视

姓名/学号：　　郭艺璇/1120161744

一、 实验目的

熟悉 Windows 系统存储器管理提供的各种机制和实现的请求调页和群集技术。了解当前内存的使用情况，包括系统地址空间的布局，物理内存的使用情况，能实时显示某个进程的虚拟地址空间布局和工作集信息等。

Windows 提供给应用程序内存方式具有统一的简明和保护性的特定。另外，用户不需要知道操作系统如何分配内存，只需要知道应用程序如何分配内存即可。

通过实验，了解 Windows 内存结构和虚拟内存的管理，学习如何在应用程序中管理内存，体会 Windows 程序使用内存的简单性。

二、 实验内容

设计一个内存监视器，能实时地显示当前系统中内存的使用情况，包括系统地址空间的布局，物理内存的使用情况；能实时显示某个进程的虚拟地址空间布局和工作集信息等。

相关的系统调用：GetSystemInfo, VirtualQueryEx, GetPerformanceInfo, GlobalMemoryStatusEx

三、 实验环境

（1）实验使用的操作系统：

（2）实验使用的 IDE：

四、 实验步骤设计与实现

在 Windows 下运行的每个应用程序都认为能独占 4GB 的地址空间。其中，低 2GB 为进程所有的地址空间，用来存放用于程序和动态链接库，高 2GB 为所有进程共享区，也即操作系统占用区。事实上，很少有进程占优 2GB 的存储空间。Windows 把每个进程的虚拟内存（virtual memory）映射为物理地址内存空间。

物理内存就是计算机配装的 RAM，系统可以管理所有的物理内存。Windows 通过分配 RAM，页面文件或两者中的空间，可准确知道应用程序所示需要的内存。下面介绍本次实验所调用的函数。

本实验使用主要函数有：

（1） 获得当前系统的一些特征信息 GetSystemInfo()函数

函数格式：void GetSystemInfo（LPSYSTEM_INFO lpSystemInfo）;

参数：lpSystemInfo 为指向 SYSTEM_INFO 结构的指针，该结构有此函数填充

返回值：无

（2） 将数字转化成字符串 StrFormatByteSize()函数

函数格式：

```
LPTSTR StrFormatByteSize(LONGLONG qdw,
                         LPTSTR pszBuf,
                         UINT uiBufSize);
```

参数：longlong qdw 是将要转变的数字的值，pszBuf 是指向保存将数字转化为字符串的缓冲区指针，uiBufSize 为缓冲区的容量。

返回值：如果函数调用成功，返回一个字符串的地址指针。

（3） 检查进程虚拟内存的当前信息 VirtualQuerEx()函数

函数格式:

```
DWORD VirtualQueryEx(HANDLE hProcess,
                     LPCVOID lpAddress,
                     PMEMORY_BASIC_INFORMATION lpBuffer,
                     SIZE_T dwLength);
```

参数:hProcess 为进程句柄,lpAddress 为指向要查询的页基址指针，lpBuffer 为指向包含 MEMORY_BASIC_INFORMATION 结构的缓冲区指针用以接收要查询的内存信息，dwLength 为 MEMORY_BASIC_INFORMATION 结构的大小

返回值：如果函数调用成功，返回写入结构 lpBuffer 的字节数

（4） 去掉完成路径名的路径部分 PathStripPath()函数

函数格式：VOID PathStripPath(LPTSTR pszPath);

参数：pszPath 为完整路径名。

返回值：无

（5） 在调用进程的虚拟地址空间保留或提交一部分页 VirtualAlloc()

函数格式：LPVOID VirtualAlloc(LPVOID lpAddress,
SIZE_T dwSize,

DWORD flAllocationType,
DWORD flProtect);

参数：lpAddressshi 为待分配区域的起始地址，dwSize 为要分配或保留的区域的大小，flAllocationType 为定义分配区域的类型属性，FlProtect 为指定分配区域保护属性

返回值：如果函数调用成功，返回值为所分配页面的基地址，否则 NULL。

（6） 获得当前系统的存储器使用情况 GetPerformanceInfo()

函数格式：

BOOL WINAPI GetPerformanceInfo(
PPERFORMANCE_INFORMATION pPerformanceInformation,
DWORD cb);

参数：pPerformanceInformation 为指向 PERFORMANCE_INFORMATION 结构体的指针，cb 是 PERFORMANCE_INFORMATION 结构体的大小

返回值：如果函数调用成功，返回值为 TRUE，失败返回值为 FALSE。

（7） 获取系统当前物理内存和虚拟内存的使用情况函数

函数格式：

BOOL WINAPI GlobalMemoryStatusEx(LPMEMORYSTATUSEX lpBuffer);

参数：lpBuffer 为指向 MEMORYSTATUSEX 结构体的指针。

返回值：如果函数调用成功，返回非 0，失败返回 0

（8） 获取当前进程已加载模块的文件的完整路径 GetModuleFileName()

函数格式：

DWORD WINAPI GetModuleFileName(HMODULE hModule, //模块句柄
LPTSTR lpFilename, //存放文件路径名的字符缓冲区
DWORD nSize //缓冲区的大小);

参数：hModule 为模块句柄，lpFilename 存放文件路径名的字符缓冲区 ，nSize 为缓冲区的大小

返回值：如果函数调用成功，返回复制到 lpFilename 的实际字符数量，如果失败返回值为 0。

（9） 将进程虚拟空间指定范围的页面解锁 VirtualUnlock()函数

函数格式：BOOL VirtualUnlock(LPVOID lpAddress, SIZE_T dwSize);

参数：lpAddress 为指向解锁页面区域的基地址，dwSize 为解锁区域的长度（即字节数）。

返回值：如果函数调用成功，返回非 0。如果失败返回值为 0。

用到的结构体：

（1）SYSTEM_INFO 系统信息

```
typedef struct _SYSTEM_INFO {
union {
    DWORD dwOemId;          // Obsolete field...do not use
    struct {
```

```
        WORD wProcessorArchitecture;
        WORD wReserved;
      } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD dwPageSize;
  LPVOID lpMinimumApplicationAddress;
  LPVOID lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD dwNumberOfProcessors;
  DWORD dwProcessorType;
  DWORD dwAllocationGranularity;
  WORD wProcessorLevel;
  WORD wProcessorRevision;
} SYSTEM_INFO
```

该结构中，只有四个字段与内存有关。

（1） dwPageSize 为内存页的大小，当计算机 CPU 为 x86 时，该值为 4096.

（2） lpMininumApplicationAddress 为每个进程可用地址空间的最小内存地址，在 Windows NT 下，该值是 65536，因为每个进程地址空间中最低的 64KB 不可用。

（3） lpMaximumApplicationAddress 为每个进程可用的私有空间的最大内存地址，在 Windows NT 下，该值是 2147483647（即 0xFFFFFFF），之后的高 2GB 用来存放操作系统代码和内存映射文件。

（4） dwAllocationGranularity 为能够保留地址空间区域的最小单位，win32 默认是 64KB。

代码设计思路：

（1） 调用 GlobalMemoryStatusEx()函数获取系统内存信息，打印系统虚拟内存和物理内存的使用比例。

```
//获取系统内存信息
GlobalMemoryStatusEx(&statex);
```

（2） 调用 GetPerformanceInfo()函数获取系统物理存储器的信息，打印物理内存的详细信息。

```
//获取系统的存储器使用情况
GetPerformanceInfo(&pi, sizeof(pi));
```

（3） 从控制台输入进程名称，用 bMore 指针指向系统当前正在运行的进程集合中的第一个进程，调用 GetProcessMemoryInfo()函数获得进程名称，如果不是输入的进程名称。就将 bMore 指针先后移动，指向下一个进程信息。如果当前指向的进程名称是输入的进程名，就打印内存信息。

```
GetProcessMemoryInfo(hP, &pmc, sizeof(pmc))
```

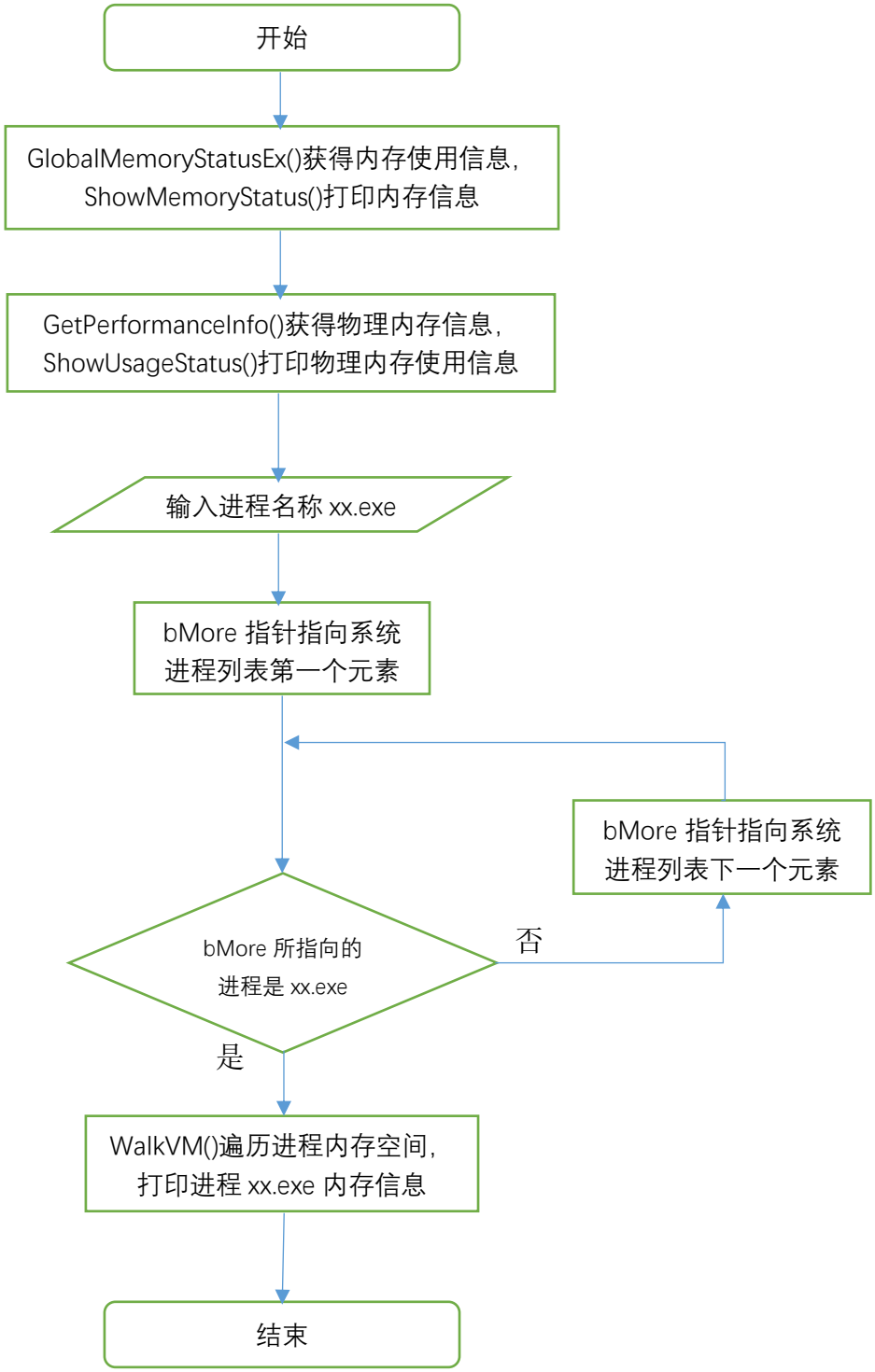
（4） 用 OpenProcess 返回该进程句柄，然后用自定义的 WalkVM()遍历该进程的虚拟内存。

```
//遍历整个虚拟内存，并显示各内存区属性的工作程序的方法
void WalkVM(HANDLE hProcess)
```

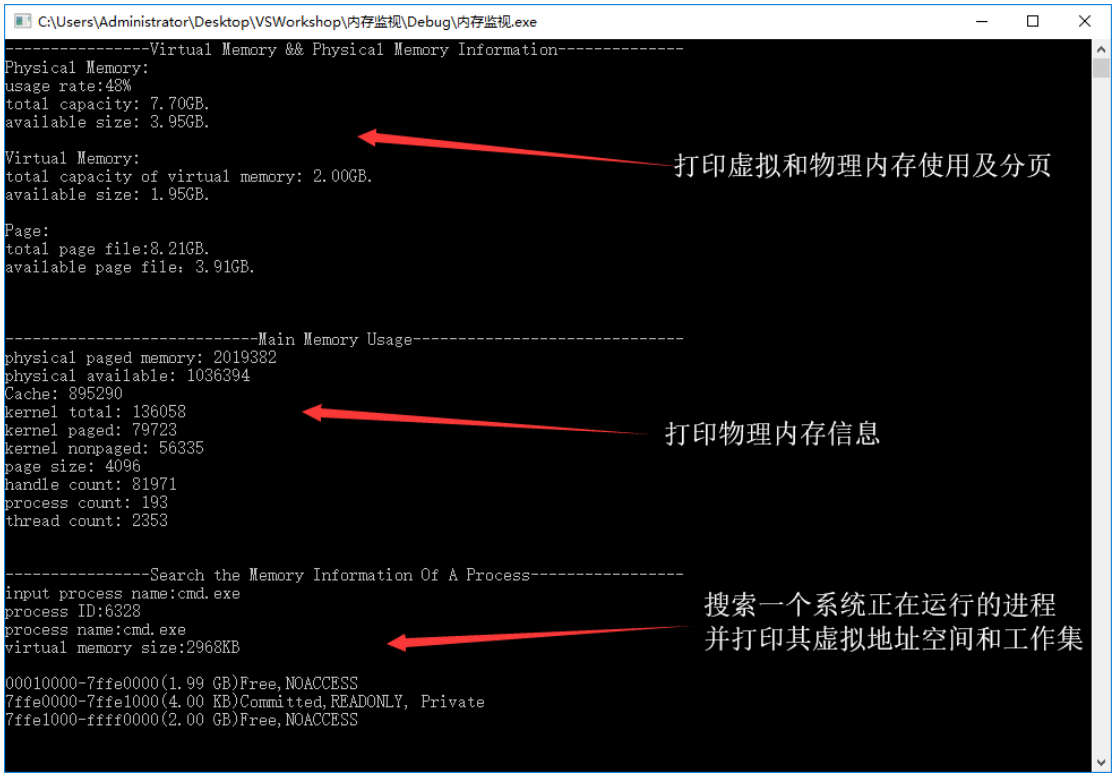
（5） 在 WalkVM()中调用 VirtualQueryEx()函数打印显示每一块的大小，类型，状态。

```
VirtualQueryEx(hProcess, pBlock, &mbi, sizeof(mbi))
```

五、 实验流程图



六、 实验结果



七、 实验收获与体会

这个实验让我们了解了几个新的 Windows 的 API，对 Windows 的内存分配有了新的更深刻的理解。

我通过实验认识到虚拟内存机制的优秀。原理上，一个 win32 进程可以使用的地址空间有 4GB。但在实际内存中，一个 win32 进程占用的存储空间远没有 4G，系统将进程需要用到的地址空间映射进物理内存。另外，一个页大小是 4KB，页面是作为一个整体被分配的。

八、 实验代码

```
#include "stdafx.h"
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
#include <windows.h>
#include <psapi.h>
#include <tlhelp32.h>
#include <shlwapi.h>
#include <iomanip>
#include <conio.h>
#pragma comment(lib,"Shlwapi.lib")

using namespace std;

//judge the protection type of a process
inline bool TestSet(DWORD dwTarget, DWORD dwMask);

//display the page protection type
void ShowProtection(DWORD dwTarget);

//display the physical and virtual memory status
void ShowMemoryStatus(MEMORYSTATUSEX statex);

//display the usage status of system physical paged memmory
void ShowUsageStatus(PERFORMANCE_INFORMATION pi);

//Traverse entire virtual memory and displays the properties for each memory region of
the working program
void WalkVM(HANDLE hProcess);

//Get the position of cursor in console
void GetConsoleCursorPosition(int &x, int &y);

//move the cursor to (x,y)
void gotoxy(int x, int y);


int main(int argc, char* argv[])
{
    MEMORYSTATUSEX statex;
    statex.dwLength = sizeof(statex);
    //get the usage of memory
    GlobalMemoryStatusEx(&statex);
    ShowMemoryStatus(statex);

    PERFORMANCE_INFORMATION pi;
    pi.cb = sizeof(pi);
    //get the usage status of system physical paged memmory
```

```
    GetPerformanceInfo(&pi, sizeof(pi));
    ShowUsageStatus(pi);

    //Search the Memory Information Of A Process
    printf("----------------Search the Memory Information Of A Process----------------
\n");
    PROCESSENTRY32 pe;
    //pe.dwSize = sizeof(pe);
    HANDLE hProcessSnap = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    BOOL bMore = ::Process32First(hProcessSnap, &pe);

    printf("input process name:");
    char process_name[100];
    cin >> process_name;
    while (bMore)
    {
        HANDLE hP = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pe.th32ProcessID);
        PROCESS_MEMORY_COUNTERS pmc;
        ZeroMemory(&pmc, sizeof(pmc));

        //search for the process we want to watch
        if (!strcmp(process_name, pe.szExeFile))
        {
            if (GetProcessMemoryInfo(hP, &pmc, sizeof(pmc)) == TRUE)
            {
                cout << "process ID:";
                wcout << pe.th32ProcessID << endl;
                cout << "process name:";
                wcout << pe.szExeFile << endl;
                cout << "virtual memory size:" << (float)pmc.WorkingSetSize / 1024 <<
"KB" << endl;
                cout << endl;
                WalkVM(hP);
                break;
            }
        }
        else//if the process bMore pointing at is not the process we search, move to
the next one
        {
            bMore = ::Process32Next(hProcessSnap, &pe);
        }
    }
    printf("\n\n");
```

```
    //int ID;
    //cin >> ID;
    //HANDLE hP = OpenProcess(PROCESS_ALL_ACCESS, FALSE, ID);
    //WalkVM(hP);
    getchar();
    getchar();
    return 0;
}

//display the protection type of each process
inline bool TestSet(DWORD dwTarget, DWORD dwMask)
{
    return ((dwTarget &dwMask) == dwMask);
}



//display the page protection type
void ShowProtection(DWORD dwTarget)
{
    if (TestSet(dwTarget, PAGE_READONLY))
        cout << "," << "READONLY";
    if (TestSet(dwTarget, PAGE_GUARD))
        cout << "," << "GUARD";
    if (TestSet(dwTarget, PAGE_NOCACHE))
        cout << "," << "NOCACHE";
    if (TestSet(dwTarget, PAGE_READWRITE))
        cout << "," << "READWRITE";
    if (TestSet(dwTarget, PAGE_WRITECOPY))
        cout << "," << "WRITECOPY";
    if (TestSet(dwTarget, PAGE_EXECUTE))
        cout << "," << "EXECUTE";
    if (TestSet(dwTarget, PAGE_EXECUTE_READ))
        cout << "," << "EXECUTE_READ";
    if (TestSet(dwTarget, PAGE_EXECUTE_READWRITE))
        cout << "," << "EXECUTE_READWRITE";
    if (TestSet(dwTarget, PAGE_EXECUTE_WRITECOPY))
        cout << "," << "EXECUTE_WRITECOPY";
    if (TestSet(dwTarget, PAGE_NOACCESS))
        cout << "," << "NOACCESS";
}
```

```
//Traverse entire virtual memory and displays the properties for each memory region of
the working program
void WalkVM(HANDLE hProcess)
{
    SYSTEM_INFO si;  //系统信息结构
    ZeroMemory(&si, sizeof(si));  //初始化
    GetSystemInfo(&si);  //获得系统信息

    MEMORY_BASIC_INFORMATION mbi;  //进程虚拟内存空间的基本信息结构
    ZeroMemory(&mbi, sizeof(mbi)); //分配缓冲区，用于保存信息

    //timeslot for refreshing, after 5s break from while
    int timeslot = 10000;

    clock_t start, end;
    start = clock();

    //int x = 0, y = 0;
    //get the original start position of cursor
    //GetConsoleCursorPosition(x, y);
/*
    while (1)
    {
        end = clock();
        if (end - start > timeslot)
        {
            break;
        }*/

        LPCVOID pBlock = (LPVOID)si.lpMinimumApplicationAddress;
        //循环整个应用程序地址空间
        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))
            {
                //计算块的结尾及其长度
                LPCVOID pEnd = (PBYTE)pBlock + mbi.RegionSize;
                TCHAR szSize[MAX_PATH];
                //将数字转换成字符串
                StrFormatByteSize(mbi.RegionSize, szSize, MAX_PATH);

                //显示块地址和长度
```

```
                cout.fill('0');
                cout << hex << setw(8) << (DWORD)pBlock << "-" << hex << setw(8) <<
(DWORD)pEnd << (strlen(szSize) == 7 ? "(" : "(") << szSize << ")";

                //display the status of each memory block
                switch (mbi.State)
                {
                case MEM_COMMIT:
                    printf("Committed");
                    break;
                case MEM_FREE:
                    printf("Free");
                    break;
                case MEM_RESERVE:
                    printf("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:
                    printf(", Image");
                    break;
                case MEM_MAPPED:
                    printf(", Mapped");
                    break;
                case MEM_PRIVATE:
                    printf(", Private");
                    break;
                }

                //检验可执行的映像
                //TCHAR szFilename[MAX_PATH];
                //if (GetModuleFileName((HMODULE)pBlock, szFilename, MAX_PATH) > 0)
                //实际使用的缓冲区长度
                //{
```

```
                //  //除去路径并显示
                //  PathStripPath(szFilename);
                //  printf(", Module:%s", szFilename);
                //}
                printf("\n");

                //移动块指针以获得下一个块
                pBlock = pEnd;

            }//if (VirtualQueryEx(hProcess, pBlock, &mbi, sizeof(mbi)) == sizeof(mbi))

        }//while (pBlock < si.lpMaximumApplicationAddress)

        //move the cursor back to where it used to be.
        //gotoxy(x, y);
        //system("cls");
    //}
}




//display the physical and virtual memory status
void ShowMemoryStatus(MEMORYSTATUSEX statex)
{
    printf("----------------Virtual Memory && Physical Memory Information--------------
\n");
    printf("Physical Memory:\n");
    printf("usage rate:%ld%%\n", statex.dwMemoryLoad);
    printf("total capacity: %.2fGB.\n", (float)statex.ullTotalPhys / 1024 / 1024 /
1024);
    printf("available size: %.2fGB.\n\n", (float)statex.ullAvailPhys / 1024 / 1024 /
1024);
    printf("Virtual Memory:\n");
    printf("total capacity of virtual memory: %.2fGB.\n", (float)statex.ullTotalVirtual
/ 1024 / 1024 / 1024);
    printf("available size: %.2fGB.\n\n", (float)statex.ullAvailVirtual / 1024 / 1024 /
1024);
    printf("Page:\n");
    printf("total page file:%.2fGB.\n", (float)statex.ullTotalPageFile / 1024 / 1024 /
1024);
    printf("available page file: %.2fGB.\n\n", (float)statex.ullAvailPageFile / 1024 /
1024 / 1024);
```

```
    //printf("保留字段的容量为：%.2fByte.\n", statex.ullAvailExtendedVirtual);
    printf("\n\n");
}

//display the usage status of system physical paged memmory
void ShowUsageStatus(PERFORMANCE_INFORMATION pi)
{
    printf("----------------------------Main Memory Usage-------------------------------
\n");
    cout << "physical paged memory: " << pi.PhysicalTotal << endl;
    cout << "physical available: " << pi.PhysicalAvailable << endl;
    cout << "Cache: " << pi.SystemCache << endl;
    //cout << "commit total: " << pi.CommitTotal << endl;
    //cout << "max number of commit page total: " << pi.CommitLimit << endl;
    //cout << "system committed page peak: " << pi.CommitPeak << endl;
    cout << "kernel total: " << pi.KernelTotal << endl;
    cout << "kernel paged: " << pi.KernelPaged << endl;
    cout << "kernel nonpaged: " << pi.KernelNonpaged << endl;
    cout << "page size: " << pi.PageSize << endl;
    cout << "handle count: " << pi.HandleCount << endl;
    cout << "process count: " << pi.ProcessCount << endl;
    cout << "thread count: " << pi.ThreadCount << endl;
    printf("\n\n");
}

//get the position of cursor in console
void GetConsoleCursorPosition(int &x, int &y)
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    //COORD coordScreen = { 0, 0 };
    //光标位置
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    if (GetConsoleScreenBufferInfo(hConsole, &csbi))
    {
        printf("光标坐标:(%d,%d)\n", csbi.dwCursorPosition.X,
```

```
csbi.dwCursorPosition.Y);
        x = csbi.dwCursorPosition.X;
        y = csbi.dwCursorPosition.Y;
    }
}
```

```
//move the cursor to (x,y)
void gotoxy(int x, int y)
{
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    HANDLE hConsoleOut;
    hConsoleOut = GetStdHandle(STD_OUTPUT_HANDLE);
    GetConsoleScreenBufferInfo(hConsoleOut, &csbiInfo);
    csbiInfo.dwCursorPosition.X = x;
    csbiInfo.dwCursorPosition.Y = y;
    SetConsoleCursorPosition(hConsoleOut, csbiInfo.dwCursorPosition);
}
```