实验四：内存监视

一、 实验目的

了解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。

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,VirtualAlloc, GetPerformanceInfo, GlobalMemoryStatusEx ...

三、实验环境

硬件配置 : 联想 IdeaPadY480 笔记本。内存 4G，硬盘 1T。

操作系统 :Windows 7

四、 程序设计与实现

打印出内存的相关信息，调用的函数有 GetSystemInfo,VirtualQueryEx,VirtualAlloc, GetPerformanceInfo, GlobalMemoryStatusExt 等。

五、实验代码、结果和分析

实验代码 ：

```
// 实验四 .cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
```

```
// MemoryWatcher.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"

#include<iostream>
#include<cstdio>
#include<windows.h>
#include<tchar.h>
#include<psapi.h>
#include<tlhelp32.h>
#include<shlwapi.h>
#include<iomanip>
#include"conio.h"
#pragma comment(lib, "psapi.lib")
#pragma comment(lib, "Shlwapi.lib")
#pragma warning(disable: 4996)
using namespace std;

#define WIDTH 10
#define DIV (1024*1024)

//WIN API 得到当前 console 的(x,y)
void console_gotoxy(int x, int y)
{
    // 得到当前 console的句柄
    HANDLE hc = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD cursor = { x, y };
    //设置新的 cursor位置
    SetConsoleCursorPosition(hc, cursor);
}
//WIN API 设置当前 console 的(x, y)
void console_getxy(int& x, int& y)
{
    // 得到当前 console的句柄
    HANDLE hc = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
    // 屏幕缓冲区信息
    CONSOLE_SCREEN_BUFFER_INFO  csbi;
    //得到相应缓冲区信息
    GetConsoleScreenBufferInfo(hc, &csbi);
    x = csbi.dwCursorPosition.X;
    y = csbi.dwCursorPosition.Y;
}
HANDLE  GetProcessHandle(int ProcessID)
{
    return OpenProcess(PROCESS_ALL_ACCESS, FALSE, ProcessID);
}

//显示保护标记，该标记表示允许应用程序对内存进行访问的类型
inline bool TestSet(DWORD dwTarget, DWORD dwMask)
{
    return ((dwTarget &dwMask) == dwMask);
}

#define SHOWMASK (dwTarget, type) \
if (TestSet(dwTarget, PAGE_##type))\
{cout<< ", " <<# type;}

void ShowProtection(DWORD dwTarget)
{
    //定义的页面保护方式
    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，显示单个进程虚拟地址空间布局
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
    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,                  //进程句柄
            pBlock,                                     //开始位置
            &mbi,                                       //缓冲区
            sizeof(mbi)) ==  sizeof(mbi))           //长度的确认，如果失败则
返回0
        {
            //块的结尾指针
            LPCVOID  pEnd = (PBYTE )pBlock + mbi.RegionSize;
            TCHAR  szSize[MAX_PATH  ];
            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
                << (wcslen(szSize) == 7 ?" (" : " (") << szSize
                << ") ";

            //显示块的状态

            switch (mbi.State) {
            caseMEM_COMMIT  :
                printf(" Committed"); break;
            caseMEM_FREE :
                printf(" Free"); break;
            caseMEM_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) {
            caseMEM_IMAGE  :
                printf(", Image"); break;
            caseMEM_MAPPED  :
                printf(", Mapped"); break;
            caseMEM_PRIVATE  :
                printf(", Private"); break;
            }

            //检测可执行的映像

            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)                //实际使用的缓冲区长度
            {
```

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

            printf("\n");
            //移动块指针以获得下一个块
            pBlock = pEnd;
        }
    }
}
//关于当前系统的信息
void ShowProcessAddress()
{
    int lineX = 0, lineY = 0;
    int flag = 0;
    SYSTEM_INFO sys_info;    //系统信息结构
    ZeroMemory(&sys_info, sizeof(sys_info));// 初始化
    while (!kbhit())
    {
        //使用win api控制缓冲区刷新输出
        if (flag == 0)
        {
            console_getxy(lineX, lineY);
            flag++;
        }
        else
        {
            console_gotoxy(lineX, lineY);
        }
        //获得系统信息
        GetSystemInfo(&sys_info);
        printf("虚拟内存分页大小 : %d KB\n", sys_info.dwPageSize / 1024);
        printf("处理器总数 : %d\n", sys_info.dwNumberOfProcessors);
```

```
        printf("处理器架构 : %d\n", sys_info.dwProcessorType);
        printf("虚拟内存粒度 : %d KB\n",
sys_info.dwAllocationGranularity / 1024);
        printf("体系结构相关的处理器等级  : %d\n",
sys_info.wProcessorLevel);
        printf("体系结构相关的处理器修订  : %x\n",
sys_info.wProcessorRevision);
        printf("应用最小地址 : 0x%0.8x\n",
sys_info.lpMinimumApplicationAddress);
        printf("应用最大地址 : 0x%0.8x\n",
sys_info.lpMaximumApplicationAddress);
        printf("应用可用虚拟内存大小  : %0.2f GB\n",
((DWORD )sys_info.lpMaximumApplicationAddress
            - (DWORD )sys_info.lpMinimumApplicationAddress) /
(1024.0*1024.0*1024.0));
        Sleep(1000);
    }

}
void ShowMemory()
{
    int lineX = 0, lineY = 0;
    int flag = 0;
    MEMORYSTATUS  total;
    total.dwLength = sizeof(total);
    while (!kbhit())
    {
        //使用 win api 控制缓冲区刷新输出
        if (flag == 0)
        {
            console_getxy(lineX, lineY);
            flag++;
        }
        else
        {
            console_gotoxy(lineX, lineY);
```

```
        }
        //得到当前物理内存和虚拟内存
        GlobalMemoryStatus(&total);
        cout << "加载的内存 : " << total.dwMemoryLoad <<  "%\n";
        cout << "总的物理内存 : " << total.dwTotalPhys / DIV  << "MB\n" ;
        cout << "可用物理内存 : " << total.dwAvailPhys /  DIV  << "MB\n" ;

        cout << "总的虚拟内存 : " << (total.dwTotalVirtual /   DIV ) <<
"MB\n" ;
        cout << "可用虚拟内存 : " << (total.dwAvailVirtual /   DIV ) <<
"MB\n" ;
        cout << "总的页的大小 : " << total.dwTotalPageFile / DIV  <<
"MB\n" ;
        cout << "可用页大小 : " << total.dwAvailPageFile /  DIV  <<
"MB\n" ;

        Sleep(1000);
    }
}
void ShowPerformance()
{
    int lineX = 0, lineY = 0;
    int flag = 0;
    PERFORMANCE_INFORMATION    perfor_info;
    perfor_info.cb =  sizeof(perfor_info);
    while (!kbhit())
    {
        //使用 win api 控制缓冲区刷新输出
        if (flag == 0)
        {
            console_getxy(lineX, lineY);
            flag++;
        }
        else
        {
```

```
            console_gotoxy(lineX, lineY);
        }
        GetPerformanceInfo(&perfor_info,  sizeof(perfor_info));
        cout << "分页大小：" << perfor_info.PageSize / 1024 <<"KB"  <<
endl;
        cout << "系统提交的页面总数   : " << perfor_info.CommitTotal <<   "
Pages"<< endl;
        cout << "系统提交的页面限制   : " << perfor_info.CommitLimit <<
" Pages"<< endl;
        cout << "系统提交的页面峰值   : " << perfor_info.CommitPeak <<  "
Pages"<< endl;
        cout << "按页分配的物理内存总数   : " << perfor_info.PhysicalTotal
<< " Pages"<< endl;
        cout << "按页分配的物理内存可用量   : " <<
perfor_info.PhysicalAvailable <<  " Pages" << endl;
        cout << "系统物理内存占用   : " << (perfor_info.PhysicalTotal -
perfor_info.PhysicalAvailable)*(perfor_info.PageSize / 1024)*1.0 /   DIV  <<
"GB"  << endl;
        cout << "系统物理内存可用   : " <<
perfor_info.PhysicalAvailable*(perfor_info.PageSize / 1024)*1.0 /   DIV  <<
"GB"  << endl;
        cout << "系统物理内存总数   : " <<
perfor_info.PhysicalTotal*(perfor_info.PageSize / 1024)*1.0 /   DIV  <<
"GB"  << endl;
        cout << "系统缓存总量：     " << perfor_info.PhysicalAvailable <<  "
Pages"<< endl;
        cout << "系统内核内存占据页面总数：      " <<
perfor_info.KernelTotal <<  " Pages"<< endl;
        cout << "系统内核内存占据分页页面数：      " <<
perfor_info.KernelNonpaged << " Pages"<< endl;
        cout << "系统内核内存占据不分页页面数：      " <<
```

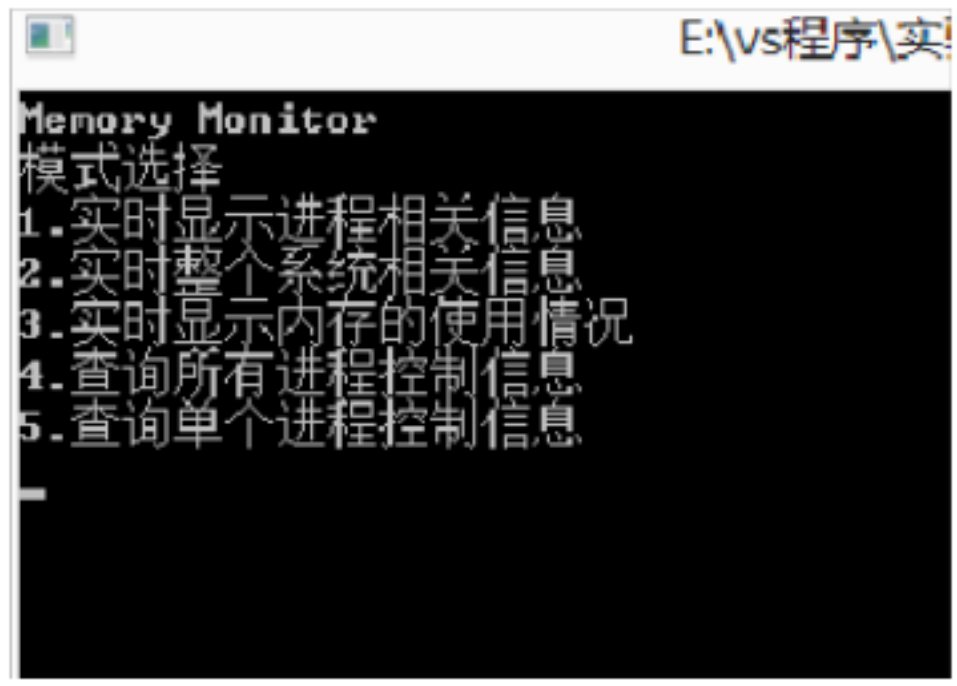
```
perfor_info.KernelPaged << " Pages" << endl;
        cout << "系统句柄总量：     " << perfor_info.HandleCount <<  "
Pages"<< endl;
        cout << "系统进程总量：     " << perfor_info.ProcessCount << "
Pages"<< endl;
        cout << "系统线程总量：     " << perfor_info.ThreadCount <<  "
Pages"<< endl;
            Sleep(1000);
    }


}
//如果 pid  为-1，获取所有进程
void  ShowAllProcess(int pid)
{

    PROCESSENTRY32 pe32;    //存储进程信息

    pe32.dwSize = sizeof(pe32); //在使用这个结构前，先设置它的大小

    PROCESS_MEMORY_COUNTERS  ppsmemCounter;//struct,存储进
程内存的使用信息，便于用函数    GetProcessMemoryInfo获取进程的相关
信息

    ppsmemCounter.cb =sizeof(ppsmemCounter); //初始化大小

    HANDLE   hProcessSnap;
    hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
0); //快照句柄

    HANDLE   hProcess;//进程句柄
    if (hProcessSnap ==INVALID_HANDLE_VALUE     ) {
        printf( "创建进程快照失败   .\n");
        exit(0);
    }
```

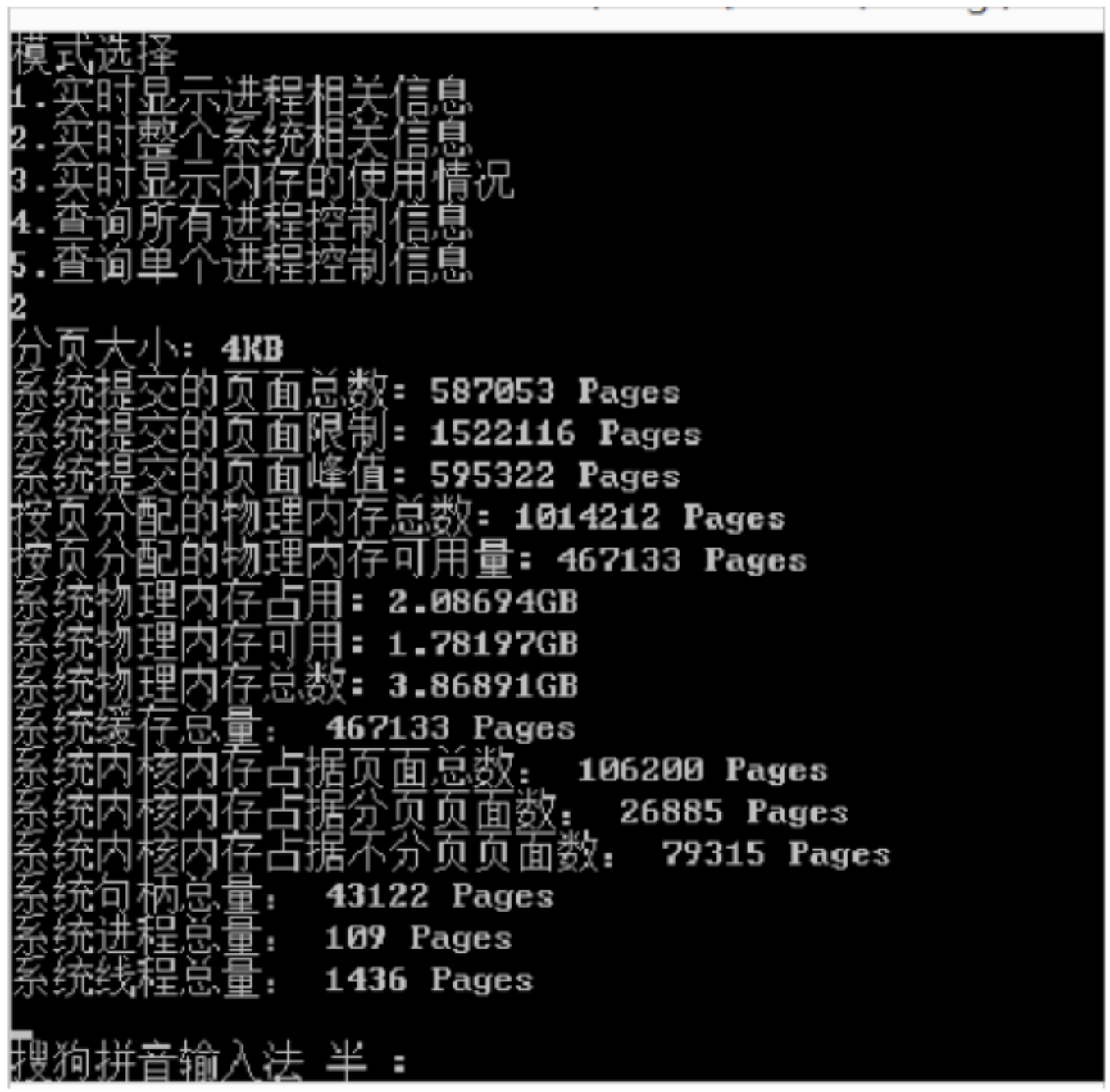
```
    //遍历进程快照，轮流显示每个进程的信息
    BOOL  bMore = Process32First(hProcessSnap, &pe32);  //获取系统快照第一个进程的信息，结果返回到    pe32结构里
    printf("进程的工作集信息  :\n");
    while  (bMore) {
        if  (pid != -1)
        {
            if  (pid == pe32.th32ProcessID)
            {
                wcout << "进程名称 :" << pe32.szExeFile << endl;//进程信息（存储于  pe32中）
                cout << "进程 ID:"  << pe32.th32ProcessID << endl;
                cout << "线程数 :" << pe32.cntThreads << endl;
                hProcess = GetProcessHandle(pe32.th32ProcessID);
                GetProcessMemoryInfo(hProcess, &ppsmemCounter, sizeof(ppsmemCounter));//进程内存使用信息  （存储于 ppsmemCounter中）
                cout << "已提交 :" << ppsmemCounter.PagefileUsage / 1024 << " KB"  << endl;
                cout << "工作集 :" << ppsmemCounter.WorkingSetSize / 1024 << " KB"  << endl;
                cout << "工作集峰值  :" << ppsmemCounter.PeakWorkingSetSize / 1024 <<" KB"  << endl;
            }
            bMore = Process32Next(hProcessSnap, &pe32);//获取系统快照下一个进程信息
        }
        else
        {
            wcout << "进程名称 :" << pe32.szExeFile << endl;//进程信息
```

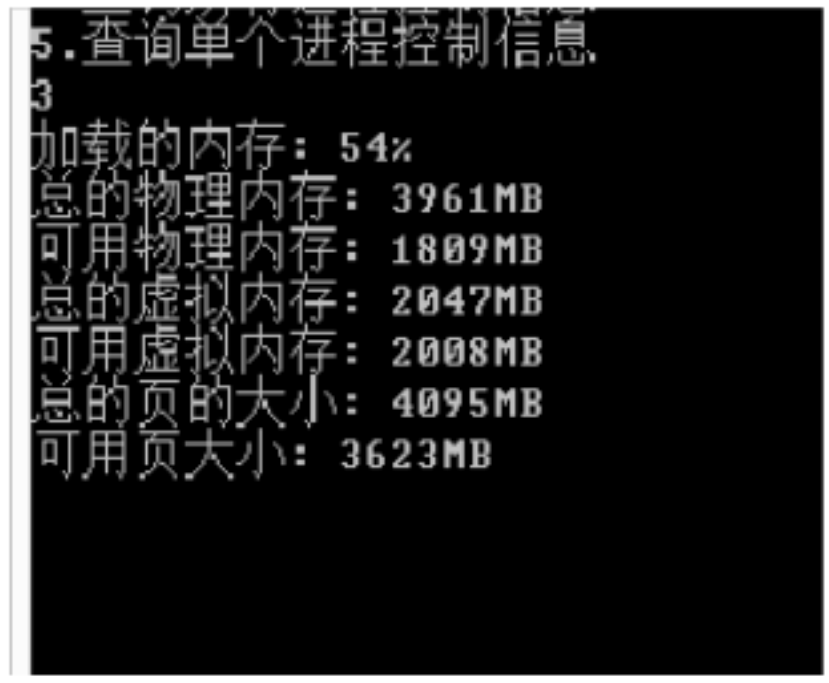
```
（存储于 pe32中）

                cout << "进程 ID:"  << pe32.th32ProcessID << endl;

                cout << "线程数 :" << pe32.cntThreads << endl;

                hProcess = GetProcessHandle(pe32.th32ProcessID);
                GetProcessMemoryInfo(hProcess, &ppsmemCounter,
sizeof(ppsmemCounter));//进程内存使用信息 （存储于 ppsmemCounter中）

                cout << "已提交 :" << ppsmemCounter.PagefileUsage / 1024 <<
" KB"  << endl;

                cout << "工作集 :" << ppsmemCounter.WorkingSetSize / 1024
<< " KB"  << endl;

                cout << "工作集峰值  :" <<
ppsmemCounter.PeakWorkingSetSize / 1024 <<" KB"  << endl;

                bMore =  Process32Next(hProcessSnap, &pe32);//获取系统快照
下一个进程信息

            }


        }

        CloseHandle(hProcessSnap);// 关闭快照

}
void  QuerySingleProcess()
{
        int lineX = 0, lineY = 0;
        int flag = 0;
        HANDLE   hProcessSnap =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0); //快照句柄

        HANDLE   hProcess;//进程句柄

        if (hProcessSnap ==INVALID_HANDLE_VALUE     ) {

                printf( "CreateToolhelp32Snapshot 调用失败 .\n");

                exit(0);

        }
```

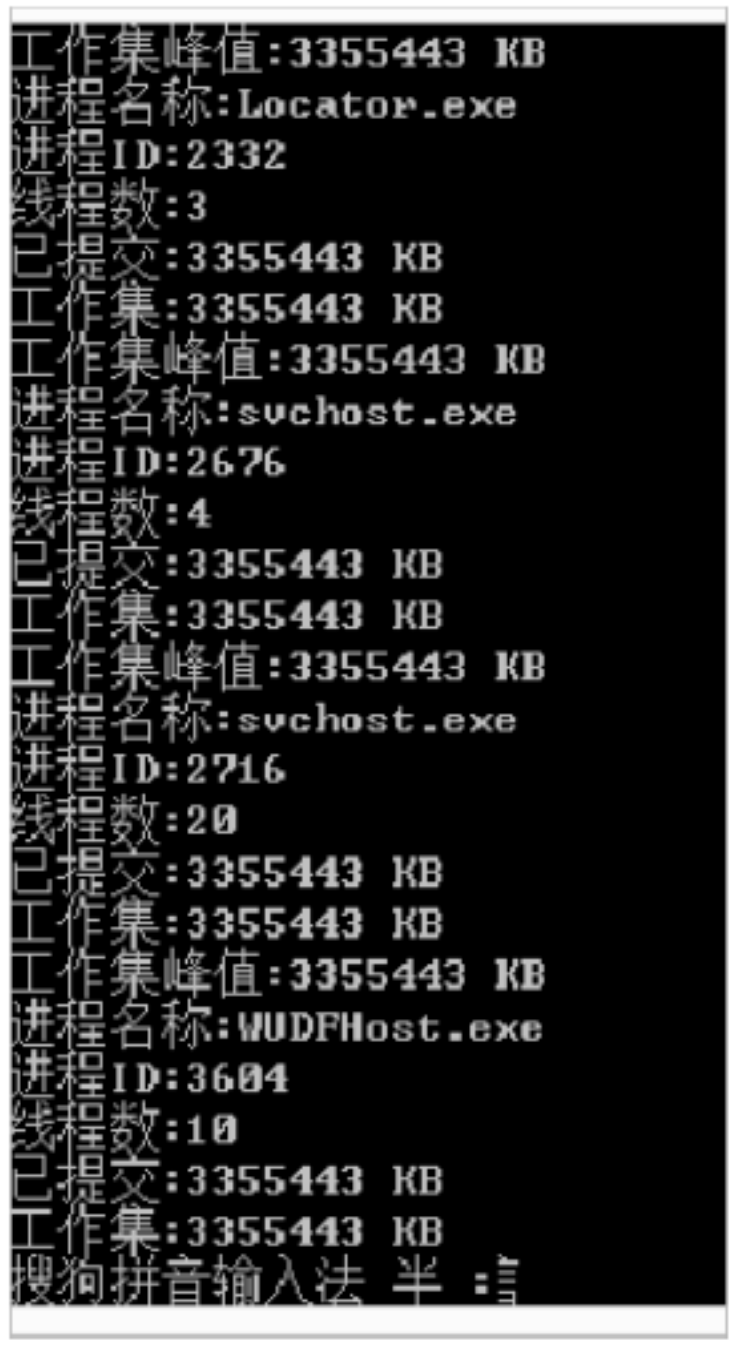
```
    cout << "输入进程 ID，查询进程的内存分布空间：    " << endl;
    int PID = 0;
    cin >> PID;
    hProcess = GetProcessHandle(PID);
    ShowAllProcess(PID);
    WalkVM(hProcess);
    Sleep(1000);

    CloseHandle(hProcess)//关闭进程
}

int main()
{
    cout << "Memory Monitor"   << endl;
    while (1)
    {
        int mode = 0;

        cout << "模式选择 " << endl;

        cout << "1.实时显示进程相关信息   " << endl;

        cout << "2.实时整个系统相关信息   " << endl;

        cout << "3.实时显示内存的使用情况   " << endl;

        cout << "4.查询所有进程控制信息   " << endl;

        cout << "5.查询单个进程控制信息   " << endl;

        cin >> mode;
        switch (mode)
        {
        case1:ShowProcessAddress();break;
        case2:ShowPerformance(); break;
        case3:ShowMemory();  break;
        case4:ShowAllProcess(-1); break;
        case5:QuerySingleProcess();break;

        default:cout << "输入格式不正确，请重新输出数字    " << endl;

        }
    }
    return 0;
}
```

运行结果如下：


E:\vs程序\实
Memory Monitor
模式选择
1.实时显示进程相关信息
2.实时整个系统相关信息
3.实时显示内存的使用情况
4.查询所有进程控制信息
5.查询单个进程控制信息


模式选择
1.实时显示进程相关信息
2.实时整个系统相关信息
3.实时显示内存的使用情况
4.查询所有进程控制信息
5.查询单个进程控制信息
2
分页大小: 4KB
系统提交的页面总数: 587053 Pages
系统提交的页面限制: 1522116 Pages
系统提交的页面峰值: 595322 Pages
按页分配的物理内存总数: 1014212 Pages
按页分配的物理内存可用量: 467133 Pages
系统物理内存占用: 2.08694GB
系统物理内存可用: 1.78197GB
系统物理内存总数: 3.86891GB
系统缓存总量: 467133 Pages
系统内核内存占据页面总数: 106200 Pages
系统内核内存占据分页页面数: 26885 Pages
系统内核内存占据不分页页面数: 79315 Pages
系统句柄总量: 43122 Pages
系统进程总量: 109 Pages
系统线程总量: 1436 Pages
搜狗拼音输入法 半 :


5.查询单个进程控制信息
3
加载的内存: 54%
总的物理内存: 3961MB
可用物理内存: 1809MB
总的虚拟内存: 2047MB
可用虚拟内存: 2008MB
总的页的大小: 4095MB
可用页大小: 3623MB


工作集峰值:3355443 KB
进程名称:Locator.exe
进程ID:2332
线程数:3
已提交:3355443 KB
工作集:3355443 KB
工作集峰值:3355443 KB
进程名称:svchost.exe
进程ID:2676
线程数:4
已提交:3355443 KB
工作集:3355443 KB
工作集峰值:3355443 KB
进程名称:svchost.exe
进程ID:2716
线程数:20
已提交:3355443 KB
工作集:3355443 KB
工作集峰值:3355443 KB
进程名称:WUDFHost.exe
进程ID:3604
线程数:10
已提交:3355443 KB
工作集:3355443 KB
搜狗拼音输入法 半 :