操作系统课程设计实验报告

实验名称：内存监视

姓名/学号：曾煜瑾/1120172765

一、 实验目的

熟悉 Windows 存储器管理中提供的各类机制和实现的请求调页技术。

通过实验，了解 Windows 内存结构和虚拟内存管理，学习如何在应用程序中管理内存。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；能够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, GetPerformanceInfo, GlobalMemoryStatusEx …

三、 实验环境

	名称	版本
Windows 操作系统	Windows 10	企业版
Windows IDE	VSCode	1.40.1.0

四、 程序设计与实现

设计思路：

有 5 个模式选择，分别为：实时显示进程相关信息、实时显示整个系统相关信息，实时显示内存的使用情况、查询所有进程控制信息、查询单个进程控制信息，当选择某个模式时，则调用对应的函数，完成相应功能。

数据结构介绍：

1. MEMORYSTATUSEX：表示内存状态，包含了物理内存和虚拟内存的现有信息

声明如下：

```
typedef struct _MEMORYSTATUS
 {
```

DWORD dwLength;//结构体大小

DWORD dwMemoryLoad;//物理内存使用的百分比（从 0 到 100）

DWORDLONG dwTotalPhys;//实际物理内存的大小（单位：字节）

DWORDLONG dwAvailPhys;//现在可用的物理内存大小（单位：字节）

DWORDLONG dwTotalPageFile;//系统或当前进程已经提交的内存限制中的较小者（单位：字节）

DWORDLONG dwAvailPageFile;//当前进程可以提交的最大内存量（单位：字节）

DWORDLONG dwTotalVirtual;//调用进程的虚拟地址空间的用户模式部分的大小（单位：字节）

DWORDLONG dwAvailVirtual;//当前调用进程的虚拟地址空间的用户模式中未保留和未提交的内存量（单位：字节）

DWORDLONG dwAvailExtendedVirtual;//保留值，始终为 0

} MEMORYSTATUS, *LPMEMORYSTATUSEX;

头文件：sysinfoapi.h（包括在 Windows.h 中）

2. SYSTEM_INFO：保存当前计算机系统的信息，包括处理器的体系结构和类型，系统中处理器的数量，页面大小和其他信息

声明如下：

```
typedef struct _SYSTEM_INFO
{
  union {
    DWORD dwOemId;
    struct{
      WORD  wProcessorArchitecture;
      WORD  wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD       dwPageSize;//页面大小和页面保护和提交的粒度，是 VirtualAlloc 函数使用的页面大小
  LPVOID      lpMinimumApplicationAddress;//指向应用程序和动态链接库（DLL）
```

可访问的最低内存指针

LPVOID lpMaximumApplicationAddress;//指向应用程序和动态链接库（DLL）可访问的最高内存指针

DWORD_PTR dwActiveProcessorMask;//配置到系统中的处理器集掩码

DWORD dwNumberOfProcessors;//当前组中的逻辑处理器数，若要检索，则需要调用 GetLogicalProcessorInformation 函数

DWORD dwProcessorType;

DWORD dwAllocationGranularity;//可以分配虚拟内存的其实地址的粒度

WORD wProcessorLevel;

WORD wProcessorRevision;

} SYSTEM_INFO, *LPSYSTEM_INFO;

头文件：sysinfoapi.h（包含于 Windows.h 中）

3. 保存性能信息

声明如下：

typedef struct _PERFORMANCE_INFORMATION

{

DWORD cb;//结构体大小（单位：字节）

SIZE_T CommitTotal;//系统当前提交的页数，提交页面（使用 VirtualAlloc 和 MEM_COMMIT）会立即更新该值；但是在访问页面之前，物理内存不会被填充

SIZE_T CommitLimit;

SIZE_T CommitPeak;//自上次系统重新引导以来同时处于已提交状态的最大页数

SIZE_T PhysicalTotal;//实际物理内存，以页为单位

SIZE_T PhysicalAvailable;//当前可用的物理内存量，以页为单位。这部分内存是可以立即重用无需写入磁盘的内存，是备用，空闲和零列表大小的总和

SIZE_T SystemCache;//系统缓存内存量，以页为单位。该值为备用列表大小加上系统工作集

SIZE_T KernelTotal;//分页和非分页内核池中当前内存的总和，以页为单位

SIZE_T KernelPaged;当前在分页内核池的内存，以页为单位

SIZE_T KernelNonpaged;//当前在非分页内核池中的内存，以页为单位

```
  SIZE_T    PageSize;//页面大小，以字节为单位
  DWORD    HandleCount;//当前打开手柄的数量
  DWORD    ProcessCount;//当前进程数
  DWORD    ThreadCount;//当前线程数
}PERFORMANCE_INFORMATION,*PPERFORMANCE_INFORMATION,
```

头文件：psapi.h

4. PROCESSENTRY32：描述在拍摄快照时驻留在系统地址空间中的进程列表中的条目

声明如下：

```
typedef struct tagPROCESSENTRY32
 {
  DWORD        dwSize;//结构体大小，以字节为单位。在调用 Process32First 函数之前要设置为 sizeof(PROCESSENTRY32)，若没有设置，则函数调用会是失败
  DWORD        cntUsage;
  DWORD        th32ProcessID;//进程标识符 PID
  ULONG_PTR    th32DefaultHeapID;
  DWORD        th32ModuleID;
  DWORD        cntThreads;//进程启动的线程数
  DWORD        th32ParentProcessID;//创建此进程的进程标识符，即父进程的 PID
  LONG         pcPriClassBase;//此进程创建的线程的基本优先级
  DWORD        dwFlags;
  CHAR         szExeFile[MAX_PATH];//进程的可执行文件
} PROCESSENTRY32;
```

头文件：tlhelp32.h

5. PROCESS_MEMORY_COUNTERS：保存进程的内存统计信息

声明如下：

```
typedef struct _PROCESS_MEMORY_COUNTERS
 {
  DWORD    cb;//结构体大小
```

DWORD PageFaultCount;//页面错误的数量

SIZE_T PeakWorkingSetSize;//峰值工作集大小（单位：字节）

SIZE_T WorkingSetSize;//当前工作集大小（单位：字节）

SIZE_T QuotaPeakPagedPoolUsage;//峰值分页池使用情况（单位：字节）

SIZE_T QuotaPagedPoolUsage;//当前页面缓冲池使用情况（单位：字节）

SIZE_T QuotaPeakNonPagedPoolUsage;//非页面缓冲池使用率的峰值（单位：字节）

SIZE_T QuotaNonPagedPoolUsage;//当前非分页池使用情况（单位：字节）

SIZE_T PagefileUsage;//此进程的 Commit Charge 值（单位：字节），Commit Charge 是内存管理器为正在运行的进程提交的内存总量

SIZE_T PeakPagefileUsage;//此进程的生命周期内提交的峰值（单位：字节）

} PROCESS_MEMORY_COUNTERS;

头文件：psapi.h

6. MEMORY_BASIC_INFORMATION：保存有关进程虚拟地址空间的一系列页面的信息，提供给 VirtualQuery 和 VirtualQueryEx 使用

声明如下：

typedef struct _MEMORY_BASIC_INFORMATION

{

PVOID BaseAddress;//指向页面区域的基址指针

PVOID AllocationBase;//指向 VirtualAlloc 函数分配的一系列页面的基址指针

DWORD AllocationProtect;//最初分配区域时的内存保护选项

SIZE_T RegionSize;//从基址开始的区域大小，其中所有页面都具有相同属性（单位：字节）

DWORD State;//页面的状态

DWORD Protect;//该区域中页面的访问保护

DWORD Type;//页面的类型

} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;

头文件：winnt.h（包含于 Windows.h 中）

API 介绍：

1. GlobalMemoryStatusEx 函数：检索有关系统物理和虚拟内存当前使用情况的信息

声明如下：

```
BOOL GlobalMemoryStatusEx(
  LPMEMORYSTATUSEX  lpBuffer
);
```

说明：

lpSystemInfo：指向接受信息的 SYSTEM_INFO 结构体指针

返回值：若函数成功，返回值非零；若函数失败，返回值为 0

头文件：sysinfoapi.h（包括在 Windows.h 中）

DLL：KERNEL32.DLL

2. GetPerformanceInfo 函数：获取性能信息，填充 PERFORMANCE_INFORMATION 结构中的性能值

声明如下：

```
BOOL GetPerformanceInfo(
  PPERFORMANCE_INFORMATION     pPerformanceInformation,
  DWORD   cb
);
```

说明：

pPerformanceInformation：指向 PERFORMANCE_INFORMATION 结构的指针

cb：PERFORMANCE_INFORMATION 结构的大小，（单位：字节）

返回值：若函数成功，返回 TRUE；若函数失败，返回 TALSE

头文件：psapi.h

3. GetSystemInfo 函数：获取当前系统的信息

声明如下：

```
void GetSystemInfo(
  LPSYSTEM_INFO   lpSystemInfo
);
```

说明：

lpSystemInfo：指向接受信息的 SYSTEM_INFO 结构体指针

头文件：sysinfoapi.h（包括在 Windows.h 中）

4. CreateToolhelp32Snapshot 函数：获取指定进程的快照，以及这些进程使用的堆、模块和线程

声明如下：

```
HANDLE CreateToolhelp32Snapshot(
  DWORD     dwFlags,
  DWORD    th32ProcessID
);
```

说明：

hSnapshot：从先前调用的 CreateToolhelp32Snapshot 函数返回的快照句柄

ppe：指向 PROCESSENTRY32 结构的指针

返回值：若进程列表第一个条目已经复制到缓冲区，则返回 TRUE，否则返回 FALSE

头文件：tlhelp32.h

5. Process32Next 函数：获取有关系统快照中记录的下一个进程的信息

声明如下：

```
BOOL Process32Next(
  HANDLE    hSnapshot,
  LPPROCESSENTRY32     lppe
);
```

说明：

hSnapshot：从先前调用的 CreateToolhelp32Snapshot 函数返回的快照句柄

lppe：指向 PROCESSENTRY32 结构的指针

头文件：tlhelp32.h

6. OpenProcess 函数：打开现有的本地进程对象

声明如下：

```
HANDLE OpenProcess(
  DWORD    dwDesiredAccess,
```

```
  BOOL    bInheritHandle,
  DWORD   dwProcessId
);
```

说明：

dwDesiredAccess：对进程对象的访问权限

binheritHandle：若此值为 TRUE，则该进程创建的进程将继承该句柄。否则，子进程不会继承该句柄

dwProcessId：要打开的本地进程的标识符，在实验中取 PROCESSENTRY32 结构中的 th32ProcessID

返回值：若函数成功，则返回指定进程的打开句柄；若函数失败，则返回 NULL

头文件：processthreadspai.h（包含于 Windows.h 中）

7. GetProcessMemoryInfo 函数：获取指定进程的内存使用情况的信息

声明如下：

```
BOOL GetProcessMemoryInfo(
  HANDLE    Process,
  PPROCESS_MEMORY_COUNTERS     ppsmemCounters,
  DWORD    cb
);
```

说明：

Process：进程的句柄

psemCounters：

指向 PROCESS_MEMORY_COUNTERS 或 PROCESS_MEMORY_COUNTERS_EX 结构的指针

cb：ppsemCounters 结构大小

返回值：若函数成功，则返回值非零；若函数失败，则返回值为零

头文件：psapi.h

代码分析：

1. 主函数

```
int main()
{
    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;
        cout << "6.退出" << endl;
        cin >> mode;
        switch (mode)
        {
            case 1: ShowProcessAddress(); break;
            case 2: ShowPerformance(); break;
            case 3: ShowMemory(); break;
            case 4: ShowAllProcess(-1); break;
            case 5: QuerySingleProcess(); break;
            case 6: return 0;
            default: cout << "输入格式不正确，请重新输出数字" << endl;
        }
    }
    return 0;
}
```

说明：通过输入模式对应的数字调用不同的函数，并且有异常处理。

2. 实时显示进程相关信息

调用的函数为 ShowProcessAddress()

```
//关于当前系统的信息
void ShowProcessAddress()
{
    SYSTEM_INFO sys_info;  //系统信息结构
    ZeroMemory(&sys_info, sizeof(sys_info));//初始化
    while (!kbhit())
    {
        //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));
        cout << endl;
        Sleep(1000);
    }

}
```

3. 实时显示系统相关信息

调用的函数为 ShowPerformance()

```
void ShowPerformance()
{
    PERFORMANCE_INFORMATION perfor_info;
    perfor_info.cb = sizeof(perfor_info);
    while (!kbhit())
    {
        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;
        cout << endl;
        Sleep(1000);
    }
}
```

4. 实时显示内存的使用情况

调用的函数为 ShowMemory()

```
void ShowMemory()
{
    MEMORYSTATUS total;
    total.dwLength = sizeof(total);
    while (!kbhit())
    {
        //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";
        cout << endl;
        Sleep(1000);
    }
}
```

5. 查询所有进程控制信息

调用的函数为 ShowAllProcess()

```
//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
    //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)
            {
                cout << "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;
                cout << endl;
            }
            bMore = Process32Next(hProcessSnap, &pe32);//获取系统快照下一个进程信息
        }
        else
```

```
        if (pid != -1)
        {
            if (pid == pe32.th32ProcessID)
            {
                cout << "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;
                cout << endl;
            }
            bMore = Process32Next(hProcessSnap, &pe32);//获取系统快照下一个进程信息
        }
        else
        {
            cout << "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;
            cout << endl;
            bMore = Process32Next(hProcessSnap, &pe32);//获取系统快照下一个进程信息
        }
    }
    CloseHandle(hProcessSnap); //关闭快照
}
```

6. 查询单个进程控制信息

调用的函数为 QuerySingleProcess()

```
void QuerySingleProcess()
{
    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
}
```

其中 WalkVM 函数是显示单个进程虚拟地址空间布局，如下：

```
//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
                << "  " << szSize;
```

```
    //显示块的状态
    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;
            }
        }
        cout << endl;
    }
```

运行效果：

1. 编译源代码

```
C:\Users\lenovo\Desktop\windows>g++ MemoryMonitor.cpp -lpsapi -lShlwapi -o MemoryMonitor

C:\Users\lenovo\Desktop\windows>
```

2. 运行

```
C:\Users\lenovo\Desktop\windows>MemoryMonitor
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
6.退出
```

3. 选择实时显示进程相关信息

```
C:\Users\lenovo\Desktop\windows>MemoryMonitor
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
6.退出
1
虚拟内存分页大小: 4 KB
处理器总数: 4
处理器架构: 8664
虚拟内存粒度: 64 KB
体系结构相关的处理器等级: 6
体系结构相关的处理器修订: 8e09
应用最小地址: 0x00010000
应用最大地址: 0xfffeffff
应用可用虚拟内存大小: 32768.00 GB

虚拟内存分页大小: 4 KB
处理器总数: 4
处理器架构: 8664
虚拟内存粒度: 64 KB
体系结构相关的处理器等级: 6
体系结构相关的处理器修订: 8e09
应用最小地址: 0x00010000
应用最大地址: 0xfffeffff
应用可用虚拟内存大小: 32768.00 GB
```

4. 选择实时系统相关信息

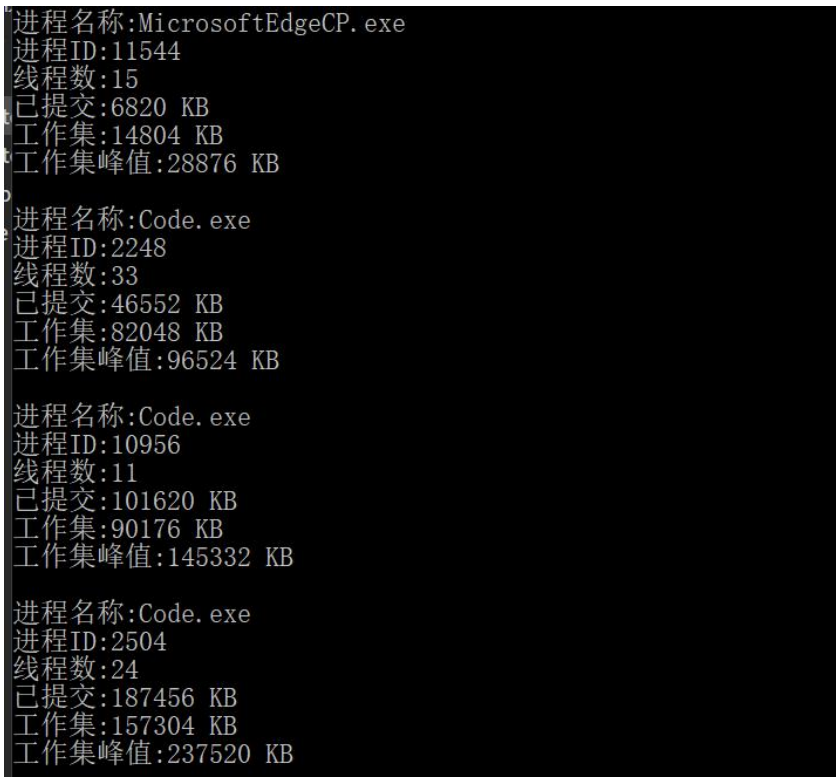
```
C:\Users\lenovo\Desktop\windows>MemoryMonitor
模式选择
1.实时显示进程相关信息
2.实时显示系统相关信息
3.实时显示内存的使用情况
4.查询所有进程控制信息
5.查询单个进程控制信息
6.退出
2
分页大小：4KB
系统提交的页面总数：1320492 Pages
系统提交的页面限制：2725088 Pages
系统提交的页面峰值：1392772 Pages
按页分配的物理内存总数：1021152 Pages
按页分配的物理内存可用量：169469 Pages
系统物理内存占用：3.24891GB
系统物理内存可用：0.646473GB
系统物理内存总数：3.89539GB
系统缓存总量： 169469 Pages
系统内核内存占据页面总数： 141560 Pages
系统内核内存占据分页页面数： 72588 Pages
系统内核内存占据不分页页面数： 68972 Pages
系统句柄总量： 81320 Pages
系统进程总量： 192 Pages
系统线程总量： 2302 Pages

分页大小：4KB
系统提交的页面总数：1321068 Pages
系统提交的页面限制：2725088 Pages
系统提交的页面峰值：1392772 Pages
按页分配的物理内存总数：1021152 Pages
按页分配的物理内存可用量：169304 Pages
系统物理内存占用：3.24954GB
系统物理内存可用：0.645844GB
系统物理内存总数：3.89539GB
系统缓存总量： 169304 Pages
系统内核内存占据页面总数： 141560 Pages
系统内核内存占据分页页面数： 72588 Pages
系统内核内存占据不分页页面数： 68972 Pages
系统句柄总量： 81321 Pages
```

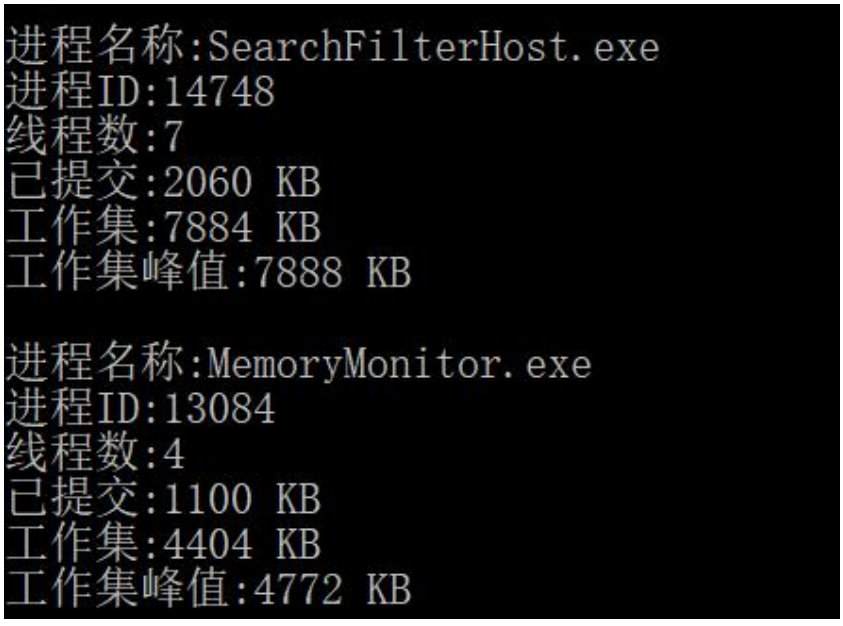
5. 实时显示内存的使用情况

```
模式选择
1.实时显示进程相关信息
2.实时显示系统相关信息
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
6.退出
3
加载的内存：83%
总的物理内存：3988MB
可用物理内存：662MB
总的虚拟内存：134217727MB
可用虚拟内存：134213543MB
总的页的大小：10644MB
可用页大小：5490MB

加载的内存：83%
总的物理内存：3988MB
可用物理内存：662MB
总的虚拟内存：134217727MB
可用虚拟内存：134213543MB
总的页的大小：10644MB
可用页大小：5490MB

加载的内存：83%
总的物理内存：3988MB
可用物理内存：662MB
总的虚拟内存：134217727MB
可用虚拟内存：134213543MB
总的页的大小：10644MB
可用页大小：5492MB
```

6. 查询所有进程控制信息

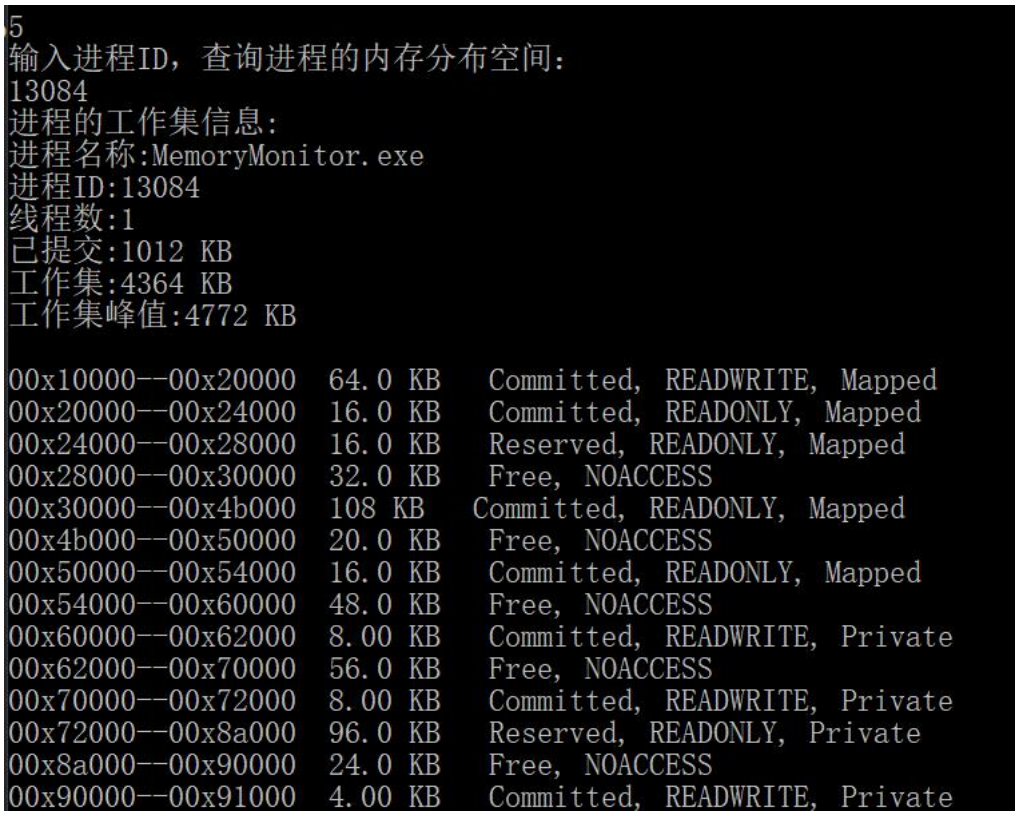
部分截图为：





7. 查看单个进程控制信息

查看 MemoryMonitor，进程 ID 为 13084



说明：打印出来的依次为：块地址、块长度、块状态、块属性、块的类型

对于块的状态，有如下几种：

COMMIT：表示已在内存中或磁盘页面文件中为其分配物理存储的已提交页面

FREE：表示调用进程无法访问且可以分配的空闲页面

RESERVE：表示保留进程的虚拟地址空间范围而不分配任何物理存储的保留页面

对于块的类型，有如下几种：

IMAGE：指示区域内的内存页面映射到映射文件到视图中

MAPPED：指示区域内的内存页面映射到节的仕途

PRIVATE：指示区域内页面是私有的

而块属性的定义如下：

```
#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);
}
```

五、实验收获与体会

这个实验总算做完了，总的来说这个实验相对比较简单，通过选择模式调用对应的函数实现对应的功能。在功能函数里，也只需打印出结构体内部对应的变量即可。

题目中要求实时显示当前系统中内存的使用的情况，对于实时显示，是通过kbhit()函数监控是否操控键盘，如果没有操控键盘则循环显示，直到按下键盘上的某个按键停止监控。

随着这个实验的实验报告即将写完，这个学期的 OS 设计课程也基本结束了。这门课，虽然花了很多很多时间，基本上每个星期都要花两整天时间做实验，写实验报告。但是收获很大，了解到 LInux 和 Windows 系统调用接口的区别，提高了自己的编码能力和阅读代码能力，另外之前烦躁的配环境提高了配环境能力，比如双系统装配 VScode、共享文件夹、VMware Tools、Linux 内核等。这些都挺有意思的。

今年是 20 世纪第二个十年的最后一天，啊，时间真是太快了。现在许个愿：希望接下来的 10 年能过得不错，逐步靠近自己的理想，为梦想奋斗，一定可以的。也祝老师新年快乐，身体健康，万事如意。

哈哈，就这样了，刚好 20 页整。 ——2019 年 12 月 31 日