操作系统课程设计实验报告

实验四 内存监视器模拟实现

指导教师：陆慧梅老师

班　　级：07111507

学　　号：1120151880

姓　　名：廖汉龙

邮　　箱： liaohanlong@outlook.com

2018 年 3 月 27 日

实验在线链接：

https://github.com/HanlongLiao/Course/tree/master/OS

目录

一、实验目的

存储器管理是操作系统的 4 个基本功能之一。在操作系统发展的过程中，存储器管理也是经历了众多变化，从最早的直接寻址，到后来的段式，页式，段页式，再到现代操作系统中广泛采用的虚拟存储器管理。 Windows 为用户提供了功能强大的实存管理 API 和虚存管理 API，可以让用户实时地查看实存和虚存的使用情况。

使用这些 API 来编写内存管理程序，是掌握操作系统存储器管理功能的重要一环，不仅可以让我们了解 Windows 的内存结构和虚拟内存的管理，同时，也加深了对操作系统原理中，相关知识的理解。

完成本实验，做到掌握在 Windows 系统中设计内存监视器

(1). 熟悉 Windows 环境下查看内存信息的系统函数。

学习 Windows 系统下查看内存信息的有关系统函数，掌握函数作用，理解函数参数代表的具体内容。

(2). 掌握查看内存状态的方法。

学会查看系统内存大小，内存占用率，系统地址空间的布局等信息，以及某个进程的虚拟地址空间使用情况和工作集信息。

二、实验内容

在 Windows 系统下设计一个内存监视器，要求：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；实时显示实验二进程控制(ParentProcess.exe)的虚拟地址空间布局和工作集信息 。

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

三、实验环境

操作系统	**Windows 10 家庭中文版**
CPU	Core i5 4200
RAM	8G
编译环境	Visual Studio Code, gcc 4.9.2

四、实验步骤与内容

4.1 实验原理

4.1.1 地址空间管理机制

在 32 位 Windows 系统中，每个进程独占一个专属的地址空间。每个地址空间上，允许每个用户进程占有 4G 的虚存空间。低 2GB 为进程的私有地址空间，高 2GB 为进程公用的操作系统空间。Windows 管理进程私有地址空间采用两种描述方式：

(1) 虚拟地址描述符（VAD， Virtual Address Descriptor）

(2) 区域对象（Section Object）

区域对象在实验三时候有过使用经验，所以不再讨论，本节重点讨论用于在虚拟地址空间中分配内存的虚拟地址描述符（VAD）。 Windows 对于存储器管理器采用请求页式调度算法，进程页表的构建一直推迟到访问页时才建立，这是一种懒惰的方式，可以最大限度地减轻系统负担。当一个线程要求分配一块连续虚存时，存储器管理器并不立即为其构造页表，而是为它建立一个 VAD 结构，记录该地址空间的相关信息。进程的页表依据 VAD 来建立。 VAD 结构包括被分配的地址域、该域是共享的还是私有的、该域的存取保护以及是否可继承等信息。

存储管理器通过维护一组 VAD 结构，记录每个进程地址空间的状态。当申请使用地址空间中的一段地址时，操作系统就会将该区域的状态由空闲（free）变为保留。一个进程的一组 VAD 结构构成一棵自平衡二叉树，以便快速查找。

4.1.2 windows 分页机制

Windows 使用基于分页机制的虚拟内存。对于 32 位的 Win32API，页的大小是 4K 字节。页表和页目录表是分页机制中最重要的两个概念。物理内存分页：一个物理页的大小为 4K 字节，第 0 个物理页从物理地址 0x00000000 处开始。由于页的大小为 4KB，就是 0x1000 字节，所以第 1 页从物理地址 0x00001000 处开始。第 2 页从物理地址 0x00002000 处开始。页的大小是 4KB，所以只需要 32bit 的地址中高 20bit 来寻址物理页。

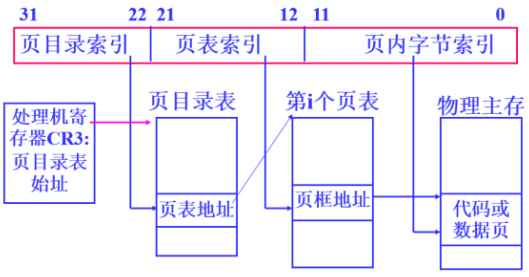


图 1 windows 分页机制

页表：一个页表的大小为 4K 字节，放在一个物理页中。由 1024 个 4 字节的页表项组成。页表项的大小为 4 个字节(32bit)，所以一个页表中有 1024 个页表项。页表中的每一项的内容（每项 4 个字节,32bit）高 20bit 用来放一个物理页的物理地址，低 12bit 放着一些标志。页目录：一个页目录大小为 4K 字节，放在一个物理页中。由 1024 个 4 字节的页目录项组成。页目录项的大小为 4 个字节(32bit)，所以一个页目录中有 1024 个页目录项。页目录中的每一项的内容高 20bit 用来放一个页表（页表放在一个物理页中）的物理地址，低 12bit 放着一些标志。（由于 2 的 20 次方乘以 2 的 12 次方（页大小）就等于了进程的 4G 地址空间）。

4.2 本实验基本 API 介绍

- **GetSystemInfo() 获得当前系统的一些特征信息**

```
void WINAPI GetSystemInfo(
    LPSYSTEM_INFO lpSystemInfo    //为指向SYSTEM_INFO 结构的指针，该结构由此函数填充
);
```

其中，SYSTEM_INFO 结构定义如下：

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD  dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    };
  };
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO;
```

该字段中，有四个字段与内存有关

dwPageSize 为内存页的大小，当计算机 CPU 为 x86 时，该值就为 4096

lpMinimumApplicationAddress 为每个进程可用地址空间的最小内存地址

lpMaximumApplicationAddress 为每个进程可用的私有地址空间最大的内存地址

dwAllocationGranularity 为能够保留地址空间区域的最小单位，win32 默认为 64kb

- **GlobalMemoryStatus() 检索当前系统使用的物理与虚拟内存的信息**

```
VOID GlobalMemoryStatus
    (
        LPMEMORYSTATUS lpBuffer    //指向MEMORYSTATUS的指针，该结构由该函数填充
    );
```

其中 MEMORYSTATUS 结构体的定义如下：

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;             // MEMORYSTATUS数据结构的大小 ，以字节为单位
    DWORD dwMemoryLoad;         // 介于0和100之间的数字，指定正在使用的物理内存的近似百分
                                // 比（0表示不使用内存，100表示使用全部内存）
    DWORD dwTotalPhys;          // 实际物理内存的大小，以字节为单位
    DWORD dwAvailPhys;          // 当前可用的物理内存量，以字节为单位
    DWORD dwTotalPageFile;      // 物理内存加上页面文件的大小，减去一小部分开销
    DWORD dwAvailPageFile;      // 当前进程可以提交的最大内存量，以字节为单位
    DWORD dwTotalVirtual;       // 调用进程的虚拟地址空间的用户模式部分的大小，以字节为单位
    DWORD dwAvailVirtual;       // 当前在调用进程的虚拟地址空间的用户模式部分中的未保留和未
                                // 提交的内存量，以字节为单位
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

此接口不返回值

- **CreateToolhelp32Snapshot() 获取指定进程的快照，以及这些进程使用的堆，模块和线程**

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,      //指定要包含在快照中的系统部分
    DWORD th32ProcessID //要包含在快照中的进程的进程标识符。该参数可以为零以指示当前进程
);
```

如果函数成功，它将返回一个打开的句柄到指定的快照。

如果该函数失败，则返回 INVALID_HANDLE_VALUE。要获得扩展的错误信息，请调用 GetLastError。可能的错误代码包括 ERROR_BAD_LENGTH。这个接口和其他几个接口一起使用，并且，这里边还会有一个结构 PROCESSENTRY32

使用 PROCESSENTRY32 来存储进程的全部信息，其定义如下：

```
typedef struct tagPROCESSENTRY32{
    DWORD dwSize;                   //结构的大小
    DWORD cntUsage;                 //此进程的引用数
    DWORD th32ProcessID;            //PID
    ULONG_PTR th32DefaultHeapID;    //进程默认堆，默认为0
    DWORD th32ModuleID;             //进程模块ID
    DWORD cntThreads;               //此进程开启的进程数
    DWORD th32ParentProcessID;      //父进程的PID
    LONG pcPriClassBase;            //线程优先权
    DWORD dwtemps;                  //此成员不再被使用，默认为0
    TCHAR szExeFile[MAX_PATH];      //进程全名（字符数组）
} PROCESSENTRY32, *PPROCESSENTRY32;
```

在使用了 CreateToolhelp32Snapshot()获得了快照的句柄之后，使用 Process32First(), Process32Next()等结构取得快照中的第一个，下一个进程的信息，赋值给当前定义的 PROCESSENTRY32 结构体

- OpenProcess() 打开一个现有的本地进程对象

```
HANDLE WINAPI OpenProcess(
    DWORD dwDesiredAccess,   //对过程对象的访问,根据进程的安全描述符检查此访问权限,该
                             //参数可以是一个或多个过程访问权限
                             //如果调用者启用了SeDebugPrivilege权限，则不管安全描述符
                             //的内容如何，都会授予请求的访问权限
    BOOL  bInheritHandle,    //如果此值为TRUE，则由此进程创建的进程将继承该句柄。否则，
                             //进程不会继承此句柄
    DWORD dwProcessId        //要打开的本地进程的标识符
);
```

如果函数成功，返回值是指定进程的打开句柄。

如果函数失败，返回值为 NULL。要获得扩展的错误信息，调用 GetLastError。

- ZeroMemory() 用 0 填充一块内存

```
void ZeroMemory(
    PVOID  Destination,   //指向内存块的起始地址以填充0的指针。
    SIZE_T Length         //用0填充的内存块大小（以字节为单位）
);
```

这个接口没有返回值

- VirtualQueryEx() 检查进程虚拟内存的当前信息

```
SIZE_T WINAPI VirtualQueryEx(
    HANDLE                    hProcess,  //为进程句柄
    LPCVOID                   lpAddress, //为指向要查询的页基地址指针
    PMEMORY_BASIC_INFORMATION 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
    SIZE_T                    dwLength   //为MEMORY_BASIC_INFORMATION 结构大小
);
```

如果调用函数成功，则返回写入结构 lpBuffer 的字节数

需要说明一下进程虚拟内存空间的基本信息结构 MEMORY_BASIC_INFORMATION

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;          //为也对其方式分配时，分配包含基地址的最小页号
  PVOID  AllocationBase;       //为应用程序的实际起始地址，显然它包含在BaseAddress 代表的页中
  DWORD  AllocationProtect;    //为该区域的访问方式
  SIZE_T RegionSize;           //为该虚拟区域的大小
  DWORD  State;                //为该区域的状态是空闲，预留，还是提交
  DWORD  Protect;              //该区域设置的访问方式，可能取值为AllocationProtect的标志之一
                               //为其中的几个标志的组合。
  DWORD  Type;                 //为该区域的页面类型。这些相邻的页面拥有相同的保护属性，状态和类型
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

- GetMuduleFilenName() 检验可执行的映像

```
DWORD WINAPI GetModuleFileName(
    HMODULE hModule,          //实际虚拟内存的模块句柄
    LPTSTR lpFilename,        //完全指定的文件名称
    DWORD nSize               //实际使用的缓冲区长度
);
```

如果函数成功，则返回值是复制到缓冲区的字符串的长度（以字符为单位），不包括终止空字符。如果缓冲区太小，无法容纳模块名，字符串被截断为 n 大小字符，包括终止空字符，该函数返回 n 大小和功能设置的最后一个错误 ERROR_INSUFFICIENT_BUFFER。

4.3 实验步骤

4.3.1 实验程序结构

这个实验的主要需要实现的功能是调用 Windows 系统中的接口，所以操作过程较为简单，下图是程序中需要实现的功能和每个部分使用的主要的接口函数的说明。

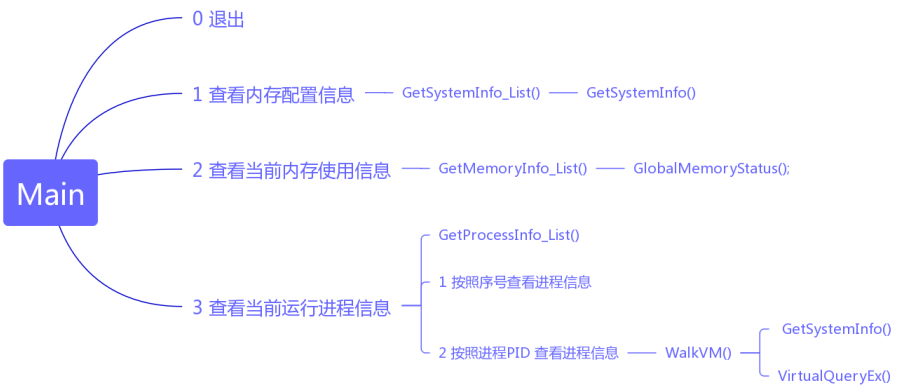


图 2 程序结构

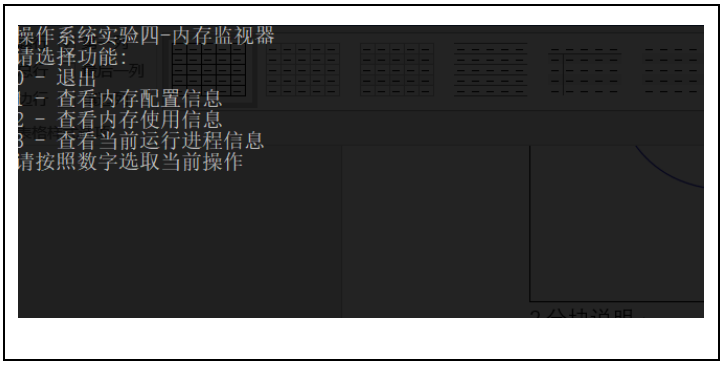
如果用户输入了 1 功能，则调用系统内存配置模块，该模块负责调用 GetSystemInfo() API，获取并转化输出系统划分的页框大小， 地址空间的起始逻辑地址， 地址空间的最高逻辑地址，可用空间，以及 CPU 数量。

如果用户选择了 2 功能，则调用内存使用情况模块。该模块的核心是调用 GlobalMemoryStatus() ,转化并输出物理内存使用率，总物理内存，可用物理内存 ，交换文件的逻辑尺寸，空闲交换文件，空闲虚拟内存。在 3 号功能下，首先用 CreateToolhelp32Snapshot( )获取当前进程的快照，其返回结果是一个链式结构，遍历这个链表，就可以得到当前活动的进程。这时，再次监听用户输入，如果输入一个进程的 pid，则调用 VirtualQueryEx() API 来遍历整个地址空间，地址空间是一个线性结构，访问完整个空间之后就可以列举出地址空间的使用情况。

4.3.2 分块程序说明

（1）主函数的主要部分。

为了实现如图所示的选择功能，需要在主函数中进行简单的条件语句的设计




```
int temp = 4;
    cout << "操作系统实验四-内存监视器";
    while (1) {
        cout << endl << "请选择功能: " << endl
            << "0 - 退出" << endl
            << "1 - 查看内存配置信息" << endl
            << "2 - 查看内存使用信息" << endl
            << "3 - 查看当前运行进程信息" << endl
            << "请按照数字选取当前操作" << endl;
        cin >> temp;
        if (temp == 0) {
            return 0;
        }
        else if (temp == 1) {
            GetSystemInfo_List();   //获取实时内存配置信息
        }
        else if (temp == 2) {
            GetMemoryInfo_List();
        }
        else if (temp == 3) {
            GetProcessInfo_List();
            int f, i, PID;
            cout << "查看某一进程的虚拟内存信息，按序号查询输入1，按PID查询输入2" << endl;
            cin >> f;
```

```
            if (f == 1) {
                cout << "请输入序号" << endl;
                cin >> i;
                if (i > ProcessTotalNum || i < 0) {
                    printf("无当前序号对应的进程");
                    continue;
                }
                else {
                    PID = ProcessInfoList[i];
                    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, PID);
                    if (hProcess == 0x00000000) {
                        printf("发生错误！ 返回的句柄为空，可能是因为无权限访问！");
                        continue;
                    }
                    WalkVM(hProcess);
                    continue;
                }
            }
            else if (f == 2) {
                cout << "请输入进程PID" << endl;
                cin >> PID;
                HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, PID);  //获取PID值
                WalkVM(hProcess);
                continue;
            }
            else {
                printf("请输入正确的指令！");
                continue;
            }
        }
    }
    return 0;

}
```

（2）功能 1 查看内存配置信息

在本次实验中， 我选择了输出页框大小， 起始逻辑地址， 最大逻辑地址， 可用空间和 CPU 数量几个指标。起始逻辑地址和最大逻辑地址的形式为一串 16 进制数字，单位为字节，用两个地址相减即可得到可用空间大小。但它的单位是字节， 为了以用户习惯的方式将其输出，Windows 提供了 StrFormatByteSizeA()函数，用以将字节换为 MB 或 GB 为单位的字符串。

这部分代码主要是要实现查看内存的信息，定义了函数 GetSystemInfo_List()， 在函数中使用的 API 为 GetSystemInfo(),核心调用的代码如下：

```
SYSTEM_INFO si;

    ZeroMemory(&si, sizeof(si));
    GetSystemInfo(&si);
```

执行了上述的过程之后，再将 si 中的各项打印。

（3） 功能 2 查看内存使用信息

由于我使用的是 MinGW 编译器，所以其调用的是 WIN32 API，在 Windows 32 子系统中，总的地址不能超过 4GB，所以可以显示出来的内存就是 2GB，但是实际安装的内存容量可能要比 2GB 要多，交换文件尺寸为 4GB，与实际的 C 盘下的 pagefile 大小不同。为了获得以 MB 为单位的结果，将返回值中的各个字段除以 1024 的平方即可。

这部分代码中要实现查看当存的使用信息，定义了函数 GetMemory_List()， 在函数中使用的 API 为 GlobalMemoryStatu(),核心调用代码如下：

```
MEMORYSTATUS stat;
long int DIV = 1024 * 1024;
long int DIV2 = 1;
GlobalMemoryStatus(&stat);
```

将 stat 中的各项打印。

（4）功能 3 查看当前运行进程的信息

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。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 2GB）。地址空间被划分为一个个地址段，每一个地址段使用一个虚拟地址描述符（VAD）来描述， VAD 记录了段起始，终止，状态和访问权限等信息。为了提高检索地址空间的速度，一个进程的一组 VAD 结构构成一棵自平衡二叉树，组织的方法是，以地址段的起始地址为权值，从小到大排序。平衡二叉树可以根据其权值构造一个线性表，这样一来，对虚拟地址空间的访问就变得非常方便。虚拟地址描述符的组织方式如图 3 所示。

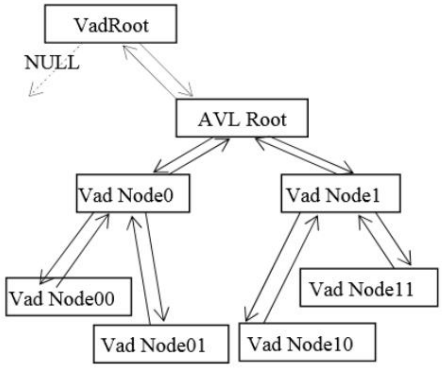


图 3 VAD 的组织方式

每一段地址空间都有其特定的状态和页面保护方式，这些方式都记录在其 VAD 中。用户进程的虚拟地址空间可能的状态有自由（free）， 已分配（committed）或保留的（reserved）三种，其可能的访问权限有 EXECUTE， READWRITE， READONLY 或 PAGE_NOCCESS 的权限。当申请使用地址空间中的一段地址时，操作系统就会将该区域的状态由空闲（free）变为保留。VirtualQueryEx( )函数可以根据进程句柄和给出的地址，获取当前段的具体信息，其最为重要的功能不仅仅是获取段的状态和页面保护方式，其还可以获取当前段的长度，有了段的长度信息之后，就可以迭代访问整个地址空间，只需要判断当前地址是否超过上界即可。每次迭代完之后，下一段的起点就是当前段的终点。

功能 4 首先打印出当前快照中的所有进程的基本信息，用户可以输入序号或者进程的 PID 查询选中的进程的按照存储块给出的详细信息。

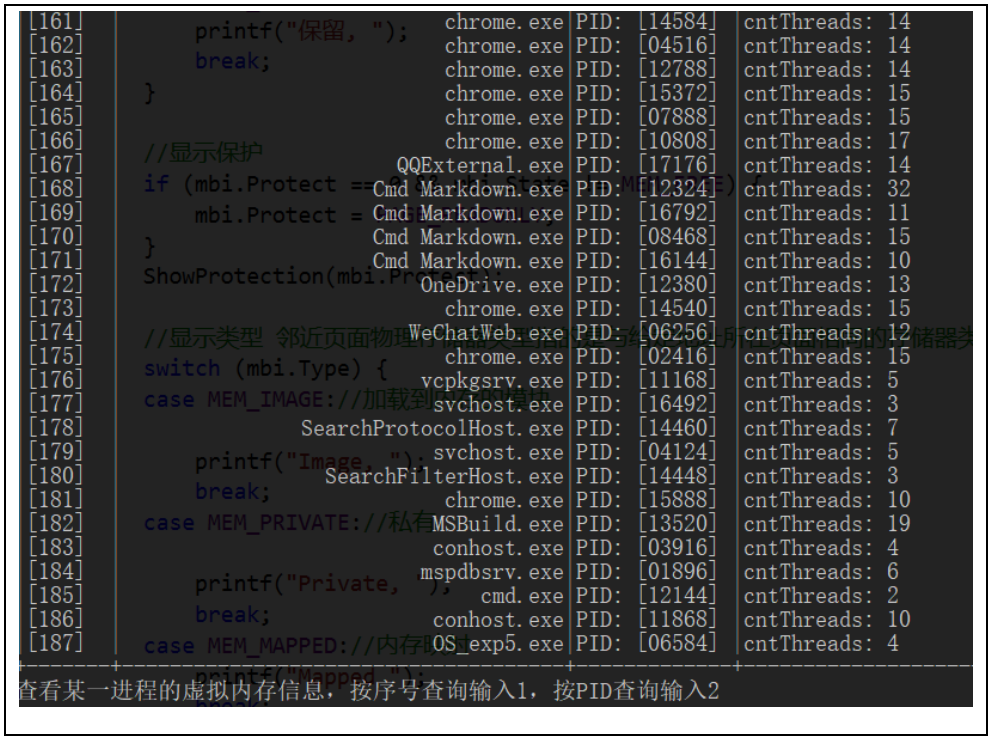


图 4 快表信息

功能 3 的实现过程的流程如下图所示，主要针对每一个进程打印三部分信息，块的状态，如已经提交或者空闲状态；块的权限，可读，可写等；以及邻近页面物理存储器类型指的是与给定地址所在页面相同的存储器类型，如 Image，Private 以及 Mapped 类型等。

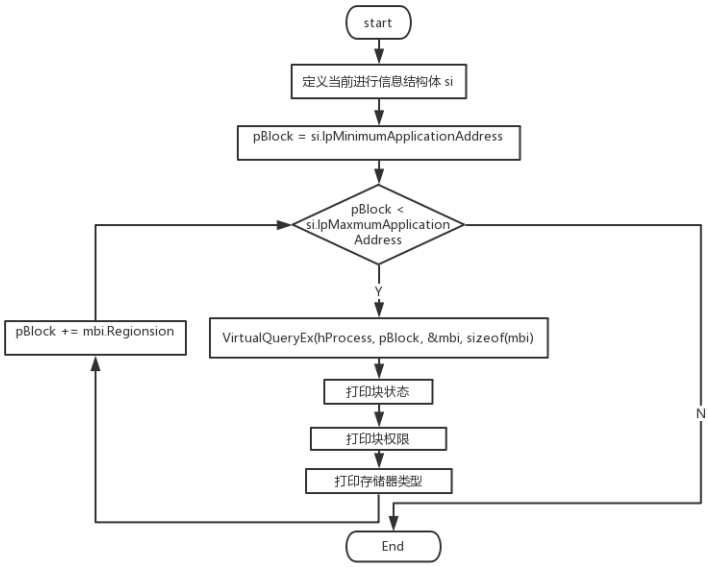


图 5 功能 3 流程图

- 打印块的状态主要代码如下：

```
//显示块的状态
switch (mbi.State) {
case MEM_COMMIT:
    printf("已被提交, ");
    break;
case MEM_FREE:
    printf("空闲, ");
    break;
case MEM_RESERVE:
    printf("保留, ");
    break;
}
```

- 打印块权限自定义了函数 ShowProtection()，定义如下：

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

//显示当前块页面保护方式
void ShowProtection(DWORD dwTarget) {
    //定义的页面保护方式
    SHOWMASK(dwTarget, READONLY);                    //只读
    SHOWMASK(dwTarget, GUARD);                       //保护
    SHOWMASK(dwTarget, NOCACHE);                     //无缓存
    SHOWMASK(dwTarget, READWRITE);                  //读写
    SHOWMASK(dwTarget, WRITECOPY);                  //写时复制
    SHOWMASK(dwTarget, EXECUTE);                     //
    SHOWMASK(dwTarget, EXECUTE_READ);               //
    SHOWMASK(dwTarget, EXECUTE_READWRITE);  //
    SHOWMASK(dwTarget, EXECUTE_WRITECOPY);  //
    SHOWMASK(dwTarget, NOACCESS);                    //未访问
}
```

- 打印 VAD 状态的主要代码如下

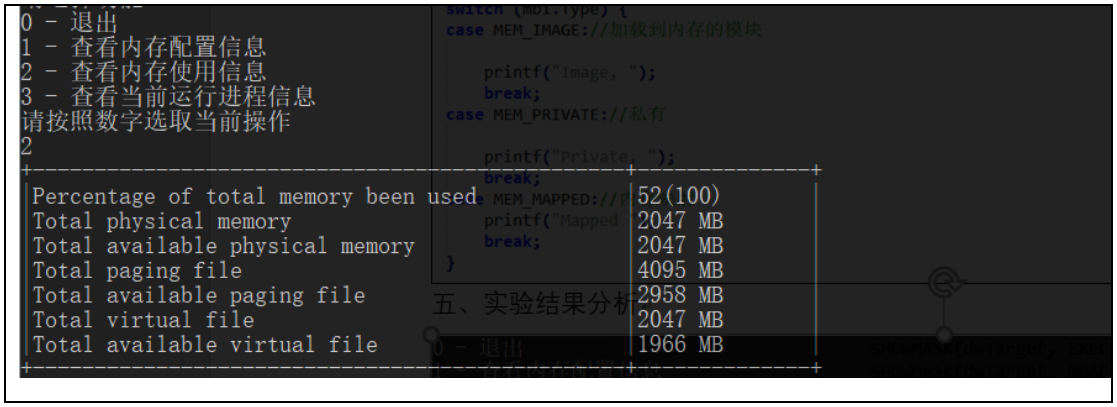
```
//显示类型 邻近页面物理存储器类型指的是与给定地址所在页面相同的存储器类型
switch (mbi.Type) {
case MEM_IMAGE://加载到内存的模块

    printf("Image, ");
    break;
case MEM_PRIVATE://私有

    printf("Private, ");
    break;
case MEM_MAPPED://内存映射
    printf("Mapped ");
    break;
}
```

五、实验结果分析

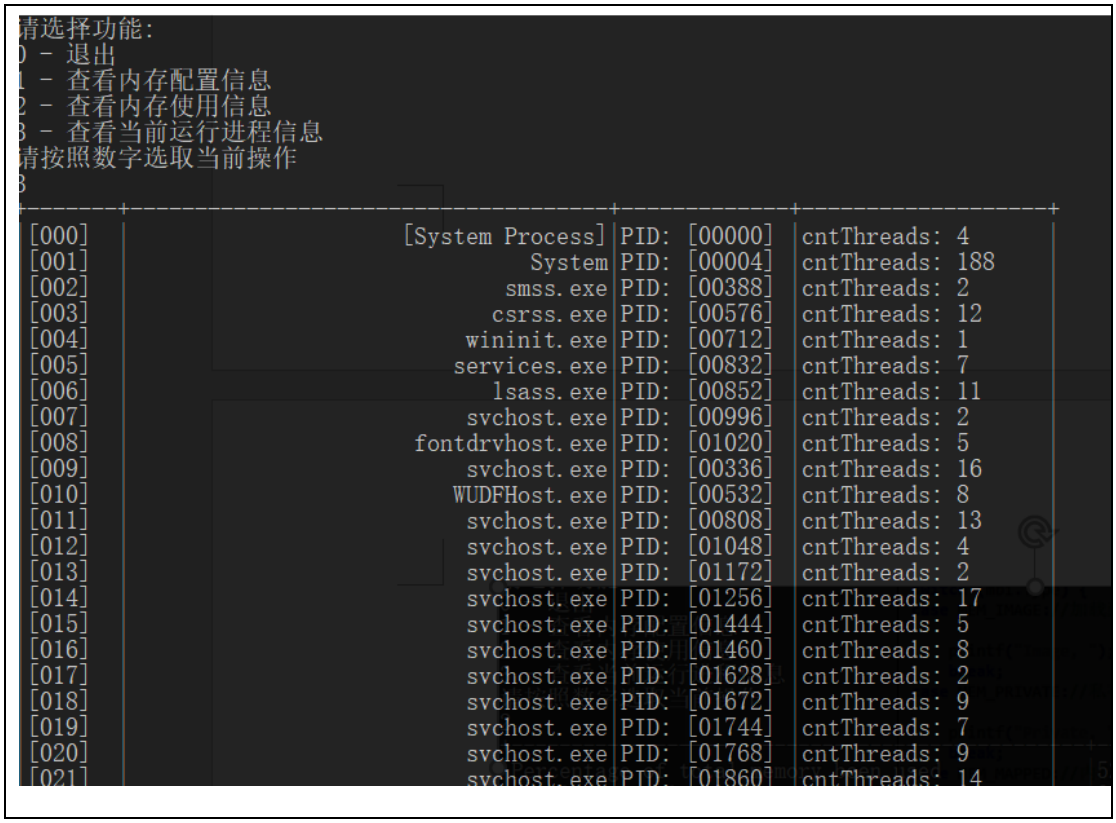
5.1 实验结果

（1）输入 1 查看内存配置信息

（2）输入 2 查看内存使用信息

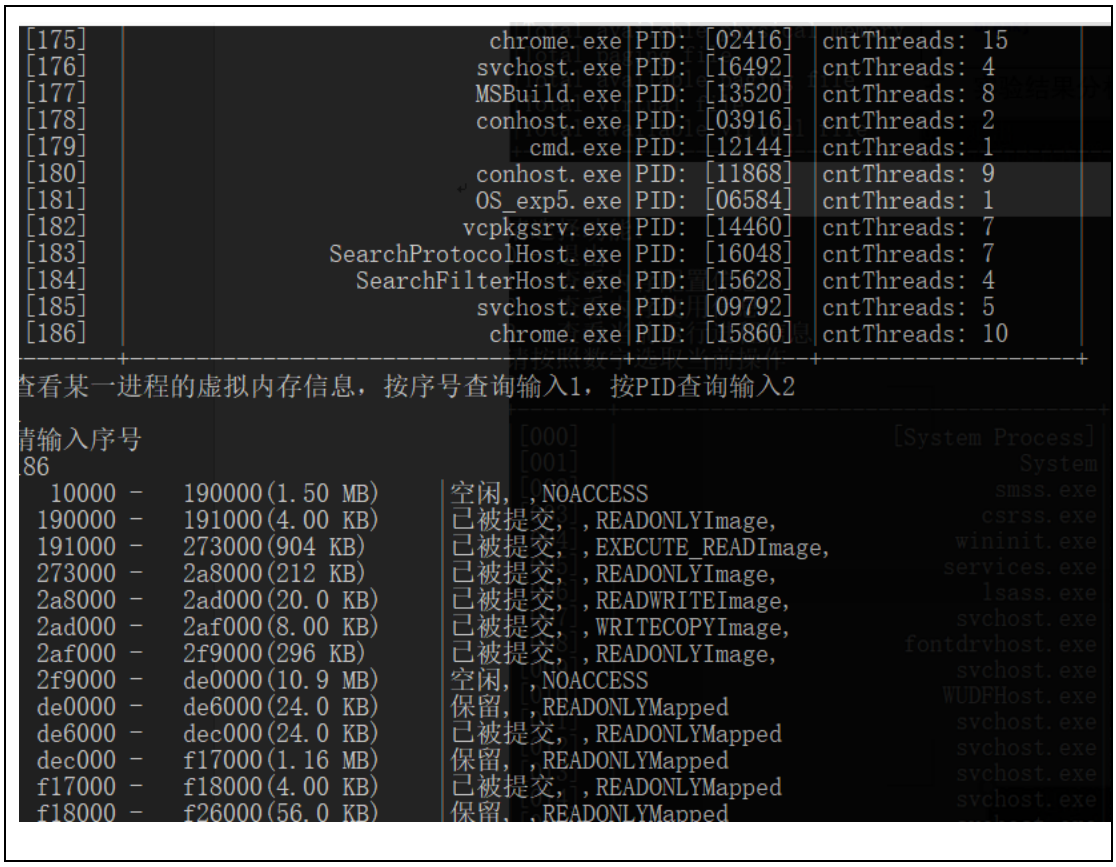


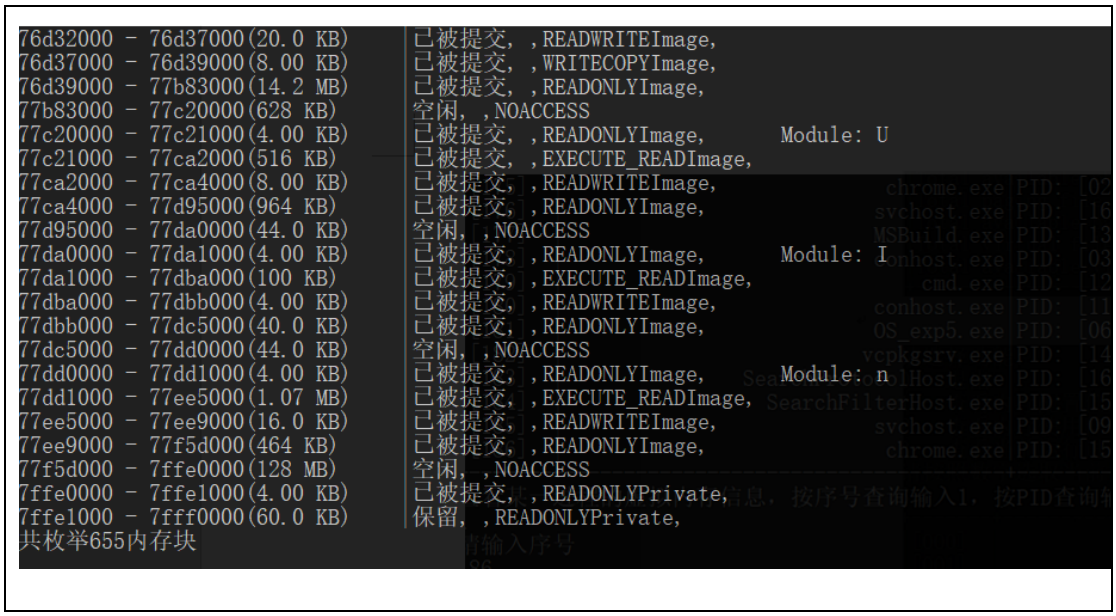
（3）输入 3 查看当前运行进程信息



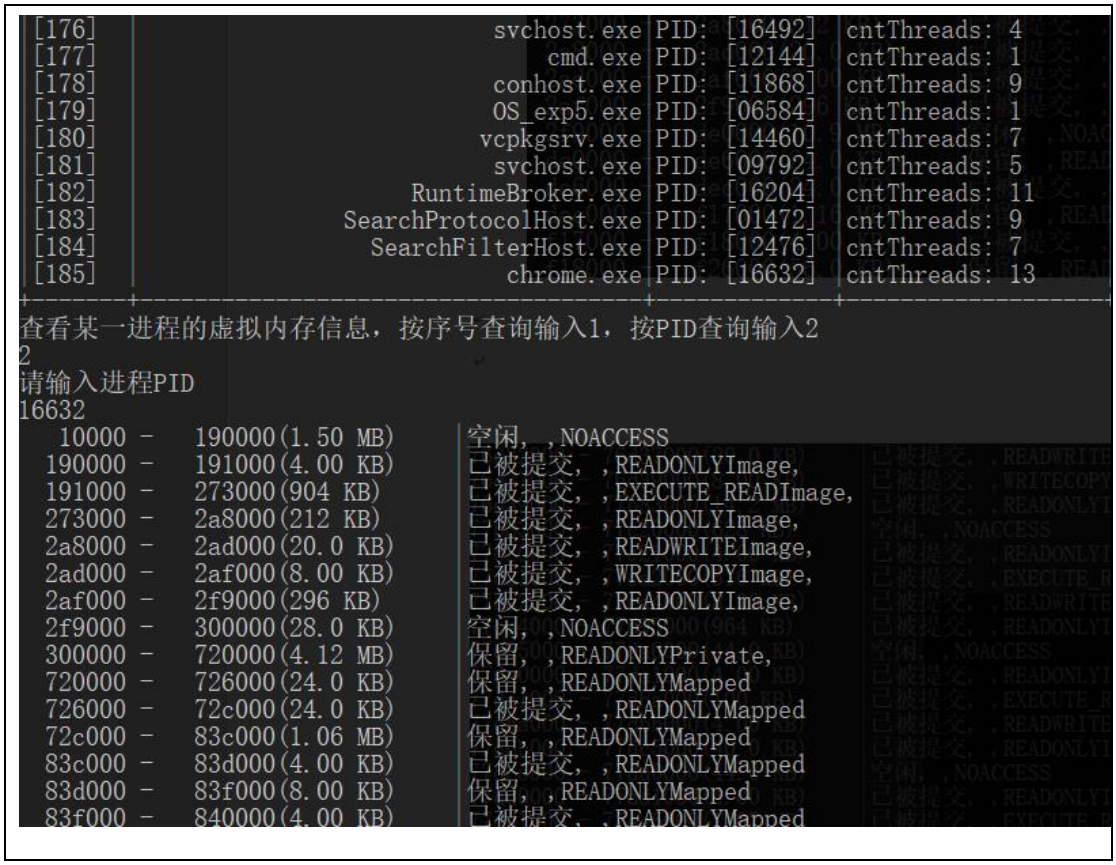
在 3 下有两种方式，可以查看指定进程的详细信息，输入 1 和输入 2

当输入 1 时的结果如下图所示



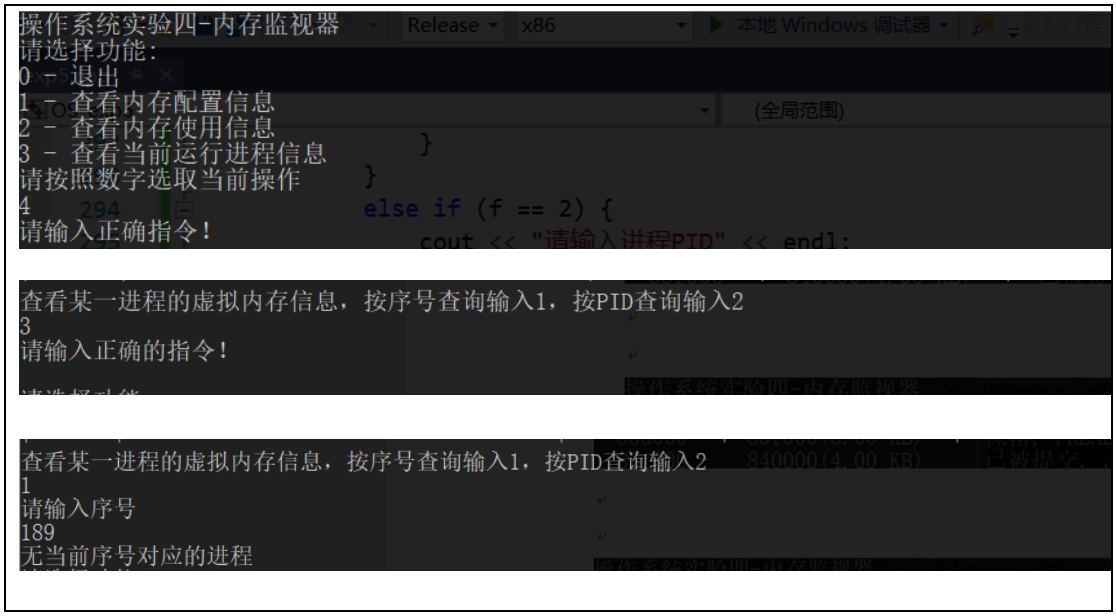


当输入 2 时，结果为



（4）程序提供容错机制

当前的指令有误或者当前需要查询的进程不存在的时候都会返回错误提示：



5.2 实验总结

本次实验是在 Windows 系统下去调用相应的 API，实现对内存，进程的进行的监视，并且在控制台输出

通过这次实现，我了解到了 GetSystemInfo, VirtualQueryEx,VirtualAlloc, GetPerformanceInfo, GlobalMemoryStatusEx ...

等一系列 windows 内部 API 的使用，并且初步了解了 Windows 内部对于内存与进程的监控机制，以及内存的分段分页存储的基础知识，实验过程中也出现了一些问题，比如

在打印进程详细信息的时候一开始返回了空句柄，后来查阅信息知道了是因为权限不足导致。