实验四 内存监视器

徐恒达 1120151811

目录

1 实验目的

熟悉 Windows 的存储器管理提供的各种机制，了解 Windows 的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。

2 实验内容

设计一个 Windows 下的内存监视器，能实现如下功能：

- 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等。
- 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。

3 实验环境

3.1 硬件配置

处理器：Intel(R) Core(TM) i3-4020Y CPU @ 1.5GHz
内存：4.0GB DDR3

3.2 操作系统

Windows 10 Pro Version 1709 64bit

4 实验过程

4.1 实现思路

程序具体实现如下功能：

- 打印输出处理器信息，包括处理器架构、逻辑核数量、处理器掩码，当前的进程数、线程数、句柄数等。
- 打印输出内存信息，包括物理内存使用率、内存页大小、物理内存使用情况、页文件使用情况、虚拟内存使用情况等。
- 打印输出系统中正在运行的进程列表，列出每个进程的进程名，进程 ID，父进程 ID，线程数，优先级信息。

- 输出指定进程的虚拟内存使用情况，包括每一块空间的地址、大小、状态、保护情况、类型等。
- 输出指定进程所包含的模块信息，包括模块名称，其值地址，占用地址空间大小，存储路径等。

主体结构的伪代码如下所示。

Algorithm 1 Monitor

Require: 命令行参数 cmd

```
if cmd = "p" then
    打印处理器信息
else if cmd = "m" then
    打印内存信息
else if cmd = "l" then
    打印当前进程列表
else if cmd = "v" then
    获取进程号 pid
    打印进程 pid 的虚拟内存使用情况
else if cmd = "d" then
    获取进程号 pid
    打印进程 pid 的模块列表
else
    输出错误信息
end if
```

4.2 主函数

主函数整体上是程序伪代码的实现，从命令行读取用户输入的命令选项，根据不同的参数执行不同的功能。这里根据 Windows 命令行程序的惯例，用“/”表示命令选项的开始。程序如下所示。

```
int main(int argc, char *argv[])
{
    if (strcmp(argv[1], "/p") == 0){
        PrintProcessorInfo();
    }
    else if (strcmp(argv[1], "/m") == 0) {
        PrintMemoryInfo();
```

```
    }
    else if (strcmp(argv[1], "/l") == 0) {
        PrintProcessList();
    }
    else if (strcmp(argv[1], "/v") == 0) {
        int pid = strtol(argv[2], NULL, 10);
        if (pid > 0) PrintVirtulMemory(pid);
        else printf("Invalid pid '%s'\n", argv[2]);
    }
    else if (strcmp(argv[1], "/d") == 0) {
        int pid = strtol(argv[2], NULL, 10);
        if (pid > 0) ListProcessModules(pid);
        else printf("Invalid pid '%s'\n", argv[2]);
    }
    else {
        printf("monitor: undefined option\n\n");
    }
    return 0;
}
```

4.3 处理器信息

获取处理器信息主要使用两个 Windows 系统函数，GetSystemInfo() 和 GetPerformanceInfo()，分别介绍如下。

函数 GetSystemInfo() 的声明如下，传入一个指向SYSTEM_INFO结构体的指针，函数将系统信息写在指针指向的结构体中。

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

结构体SYSTEM_INFO的定义如下。

```
typedef struct _SYSTEM_INFO {
  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;
```

其中 wProcessorArchitecture 字段存储了处理器架构信息,若其值为PROCESSOR_ARCHITECTURE_INTEL，表示 Intel x86 架构，PROCESSOR_ARCHITECTURE_AMD64表示 AMD 或 Intel 的 x64 架构，PROCESSOR_ARCHITECTURE_ARM表示 ARM 架构。dwNumberOfProcessors 字段表示处理器中逻辑核的数量，dwActiveProcessorMask表示处理器掩码。其它一些字段的含义在下文涉及到时予以说明。

函数 GetPerformanceInfo() 的声明如下。

```
BOOL WINAPI GetPerformanceInfo(
  _Out_ PPERFORMANCE_INFORMATION pPerformanceInformation,
  _In_ DWORD                cb
);
```

函数接受一个指向PERFORMANCE_INFORMATION结构体的指针，将信息写到指针所指向的结构体中。cb字段传入结构体的大小。

结构体PERFORMANCE_INFORMATION的定义如下。

```
typedef struct _PERFORMANCE_INFORMATION {
  DWORD cb;
  SIZE_T CommitTotal;
  SIZE_T CommitLimit;
  SIZE_T CommitPeak;
  SIZE_T PhysicalTotal;
  SIZE_T PhysicalAvailable;
  SIZE_T SystemCache;
  SIZE_T KernelTotal;
  SIZE_T KernelPaged;
  SIZE_T KernelNonpaged;
  SIZE_T PageSize;
  DWORD HandleCount;
```

```
  DWORD ProcessCount;
  DWORD ThreadCount;
} PERFORMANCE_INFORMATION;
```

其中 ProcessCount 字段存储了当前系统中正在运行的进程数量，ThreadCount 字段存储了系统中正在运行的系统级线程的数量，HandleCount 字段存储了系统中的句柄数量。其他字段的含义在下文涉及到时再作阐述。

定义 PrintProcessorInfo() 函数用来显示系统信息，调用这两个系统函数并将其中的内容打印到标准输出。函数的主体框架如下所示，具体实现中有一些控制输出格式的部分，这里没有给出，完整代码见附录。

```
void PrintProcessorInfo() {
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    printf("Processor architecture: %d\n", si.
        wProcessorArchitecture);
    printf("Logical processors: %d\n", si.dwNumberOfProcessors)
        ;
    printf("Processors mask: 0x%08x\n", si.
        dwActiveProcessorMask);
    printf("Processor level: %d\n", si.wProcessorLevel);
    printf("Processor revision: %d\n", si.wProcessorRevision);

    PERFORMANCE_INFORMATION pi;
    GetPerformanceInfo(&pi, sizeof(pi));
    printf("Total processes: %d\n", pi.ProcessCount);
    printf("Total threads: %d\n", pi.ThreadCount);
    printf("Total handles: %d\n", pi.HandleCount);
}
```

4.4 内存信息

使用 GlobalMemoryStatusEx()、GetSystemInfo() 和 GetPerformanceInfo() 这三个函数来获取内存信息，其中函数 GetPerformanceInfo() 的声明如下。

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

函数接受一个指向 MEMORYSTATUSEX 结构体的指针，将信息写在指针指向的区域中。结构体 MEMORYSTATUSEX 的定义如下。

```
typedef struct _MEMORYSTATUSEX {
  DWORD    dwLength;
  DWORD    dwMemoryLoad;
  DWORDLONG ullTotalPhys;
  DWORDLONG ullAvailPhys;
  DWORDLONG ullTotalPageFile;
  DWORDLONG ullAvailPageFile;
  DWORDLONG ullTotalVirtual;
  DWORDLONG ullAvailVirtual;
  DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX;
```

程序中主要使用了 dwMemoryLoad 字段，获得当前的物理内存使用率。

函数GetSystemInfo()和GetPerformanceInfo()在上文中已经介绍过，下面对SYSTEM_INFO和PERFORMANCE_INFORMATION这两个结构体中的字段加以说明。

```
typedef struct _SYSTEM_INFO {
  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;
```

- lpMinimumApplicationAddress：程序可访问的最小内存地址 - lpMaximumApplicationAddress：程序可访问的最大内存地址 - dwAllocationGranularity：虚拟内存的分配粒度 - ullTotalPhys：物理内存总大小 - ullAvailPhys：可用物理内存大小 - ullTotalPageFile：页文件总大小 - ullAvailPageFile：可用页文件大小 - ullTotalVirtual：虚拟内存总大小 -

ullAvailVirtual：可用虚拟内存大小

```
typedef struct _PERFORMANCE_INFORMATION {
  DWORD cb;
  SIZE_T CommitTotal;
  SIZE_T CommitLimit;
  SIZE_T CommitPeak;
  SIZE_T PhysicalTotal;
  SIZE_T PhysicalAvailable;
  SIZE_T SystemCache;
  SIZE_T KernelTotal;
  SIZE_T KernelPaged;
  SIZE_T KernelNonpaged;
  SIZE_T PageSize;
  DWORD HandleCount;
  DWORD ProcessCount;
  DWORD ThreadCount;
} PERFORMANCE_INFORMATION;
```

- CommitTotal：提交页面总大小 - SystemCache：系统 Cache 大小 - KernelTotal：操作系统内核大小 - KernelPaged：操作系统内核占用页文件大小 - KernelNonpaged：操作系统内核在物理内存大小

定义 PrintMemoryInfo() 函数来显示系统内存信息，调用这三个系统函数并将其中的内容打印到标准输出。函数的主体框架如下所示，具体实现中有一些控制输出格式的部分，这里没有给出，完整代码见附录。

```
void PrintMemoryInfo()
{
    MEMORYSTATUSEX ms;
    ms.dwLength = sizeof(ms);
    GlobalMemoryStatusEx(&ms);
    printf("Physical memory in use: %d%%\n", ms.dwMemoryLoad);

    SYSTEM_INFO si;
    GetSystemInfo(&si);
    printf("Page size: %d KB\n", si.dwPageSize / KILO);
    printf("Lowest memory address: 0x%08x\n", si.
        lpMinimumApplicationAddress);
```

```
    printf("Highest memory address: 0x%08x\n", si.
        lpMaximumApplicationAddress);
    printf("Granularity: %d KB\n", si.dwAllocationGranularity /
         KILO);

    printf("Total physical memory: %.3lf MB\n", // 物理内存
          (double)ms.ullTotalPhys / MEGA);
    printf("Available physical memory: %.3lf MB\n",
           (double)ms.ullAvailPhys / MEGA);
    printf("Total page file: %.3lf MB\n",    // 页文件
           (double)ms.ullTotalPageFile / MEGA);
    printf("Available page file: %.3lf MB\n",
           (double)ms.ullAvailPageFile / MEGA);
    printf("Total virtual memory: %.3lf TB\n", // 虚拟内存
           (double)ms.ullTotalVirtual / TERA);
    printf("Available virtual memory: %.3lf TB\n",
           (double)ms.ullAvailVirtual / TERA);

    PERFORMANCE_INFORMATION pi;
    GetPerformanceInfo(&pi, sizeof(pi));
    printf("Committed pages: %d Page\n", pi.CommitTotal);
    printf("System cache memory: %d Page\n", pi.SystemCache);
    printf("Total kernel pages: %d Page\n", pi.KernelTotal);
    printf("Paged kernel pages: %d Page\n", pi.KernelPaged);
    printf("Nonpaged kernel pages: %d Page\n", pi.
        KernelNonpaged);
}
```

4.5 进程列表

使用 CreateToolhelp32Snapshot() 函数获取当前系统中所有进程的一个快照，然后使用 Process32First() 和 Process32Next() 函数遍历快照，从而获取每个进程的信息。三个函数的介绍如下。

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  _In_ DWORD dwFlags,
  _In_ DWORD th32ProcessID
```

```
);
```

向 dwFlags 参数传入TH32CS_SNAPPROCESS标志表示获取系统进程快照,此时 th32ProcessID 参数忽略。函数返回一个 Snapshot 对象的句柄，通过这个对象来遍历所有进程。

函数 Process32First() 的声明如下。

```
BOOL WINAPI Process32First(
  _In_   HANDLE          hSnapshot,
  _Inout_ LPPROCESSENTRY32 lppe
);
```

函数接受一个 Snapshot 句柄和一个指向 PROCESSENTRY32 结构体的指针，将快照中第一个进程的信息写到结构体中。结构体 PROCESSENTRY32 的定义如下。

```
typedef struct tagPROCESSENTRY32 {
  DWORD    dwSize;
  DWORD    cntUsage;
  DWORD    th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD    th32ModuleID;
  DWORD    cntThreads;
  DWORD    th32ParentProcessID;
  LONG     pcPriClassBase;
  DWORD    dwFlags;
  TCHAR    szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

- szExeFile：可执行文件名 - th32ProcessID：进程 ID - th32ParentProcessID：父进程 ID - cntThreads：线程数 - pcPriClassBase：基本优先级

之后不断使用 Process32Next() 函数读取下一个进程信息，知道函数返回 FALSE 表示遍历结束。

```
BOOL WINAPI Process32Next(
  _In_ HANDLE          hSnapshot,
  _Out_ LPPROCESSENTRY32 lppe
);
```

定义 PrintProcessList() 函数打印进程列表，函数整体结构如下，少量控制输出格式的代码没有给出，完整代码请见附件。

```
int PrintProcessList()
{
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(
        TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    BOOL bSucceed = Process32First(hProcessSnap, &pe32);
    while (bSucceed) {
        printf("%s ", pe32.szExeFile);
        printf("%d ", pe32.th32ProcessID);
        printf("%d ", pe32.th32ParentProcessID);
        printf("%d ", pe32.cntThreads);
        printf("%d\n", pe32.pcPriClassBase);
        bSucceed = Process32Next(hProcessSnap, &pe32);
    }
    CloseHandle(hProcessSnap);
    return 0;
}
```

4.6 进程虚拟地址空间

要遍历指定进程的整个虚拟地址空间，首先使用 OpenProcess() 函数打开进程，获取进程句柄，还要再次调用 GetSystemInfo() 函数，获取程序可访问的最小地址和最大地址，然后就是从最小地址开始，依次查询空间使用情况，直到遍历完整个地址空间。

使用 VirtualQueryEx() 函数查询程序的虚拟地址空间使用情况，函数声明如下。

```
SIZE_T WINAPI VirtualQueryEx(
  _In_     HANDLE                    hProcess,
  _In_opt_ LPCVOID                   lpAddress,
  _Out_    PMEMORY_BASIC_INFORMATION lpBuffer,
  _In_     SIZE_T                    dwLength
);
```

结构体MEMORY_BASIC_INFORMATION的定义如下。

```
typedef struct _MEMORY_BASIC_INFORMATION {
```

```
  PVOID BaseAddress;
  PVOID AllocationBase;
  DWORD AllocationProtect;
  SIZE_T RegionSize;
  DWORD State;
  DWORD Protect;
  DWORD Type;
} MEMORY_BASIC_INFORMATION;
```

- BaseAddress：查询区域的基地址 - RegionSize：区域大小 - State：地址空间的状态，有三种可能的取值，MEM_COMMIT表示提交状态，MEM_FREE表示空闲状态，MEM_RESERVE表示保留状态。- Protect：地址空间的保护情况，不同的标记标志不同的访问权限设置，如PAGE_EXECUTE表示可执行，PAGE_READONLY表示只读，PAGE_READWRITE表示可读写等。- Type：内存页的使用类型，有三种可能的取值，MEM_IMAGE表示对应到可执行映像，MEM_MAPPED表示对应到映射文件，MEM_PRIVATE表示空间私有。

定义 PrintVirtulMemory() 函数打印进程的地址空间使用情况，其中略去了部分控制打印格式的代码，完整内容请见附件。

```
void PrintVirtulMemory(int pid)
{
    HANDLE hProcess = OpenProcess(
           PROCESS_ALL_ACCESS | PROCESS_VM_READ,
           FALSE, (DWORD)pid);
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    MEMORY_BASIC_INFORMATION mbi;
    ZeroMemory(&mbi, sizeof(mbi));
    int count = 0;
    LPCVOID pBlock = si.lpMinimumApplicationAddress;
    while (pBlock < si.lpMaximumApplicationAddress){
        VirtualQueryEx(hProcess, pBlock, &mbi, sizeof(mbi));

        printf("%4d ", ++count);
        LPCVOID pEnd = pBlock + mbi.RegionSize;
        printf("%08x-%08x ", pBlock, pEnd);
        printf("%10u ", mbi.RegionSize);
        printf("%6u ", mbi.RegionSize / si.dwPageSize);
```

```
        char state[10];
        GetState(mbi.State, state);
        printf("%-9s ", state);

        char protect[10] = "";
        if (mbi.State == MEM_COMMIT)
            GetProtect(mbi.Protect, protect);
        printf("%-7s ", protect);

        char type[10];
        GetType(mbi.Type, type);
        printf("%-7s \n", type);

        pBlock = pEnd;
    }
}
```

4.7 进程模块

显示指定进程模块信息的过程于显示进程列表的过程类似,首先使用 CreateToolhelp32Snapshot() 函数获取指定进程的模块信息快照，为第一个参数传入TH32CS_SNAPMODULE标志，第二个参数给出进程 ID，函数就返回一个包含进程所有模块信息快照 Snapshot 的句柄，类似地，使用 Module32First() 和 Module32Next() 函数遍历整个快照，读取每个模块的信息，每个模块的信息放在一个 MODULEENTRY32 结构体中。具体代码如下所示。

```
int ListProcessModules(int pid)
{
    HANDLE hModuleSnap = CreateToolhelp32Snapshot(
            TH32CS_SNAPMODULE, (DWORD)pid);
    MODULEENTRY32 me32;
    me32.dwSize = sizeof(MODULEENTRY32);
    BOOL bSucceed = Module32First(hModuleSnap, &me32);
    while (bSucceed) {
        printf("%-20s ", me32.szModule);
        printf("%08x ", me32.modBaseAddr);
        printf("%8d ", me32.modBaseSize);
```

```
        printf("%s\n", me32.szExePath);
        bSucceed = Module32Next(hModuleSnap, &me32);
    }
    CloseHandle(hModuleSnap);
    return 0;
}
```

5 实验结果

5.1 编译

源程序文件命名为 monitor.c，编译的 makefile 文件内容如下所示。其中需要链接 psapi 库。

```
monitor: monitor.c
    gcc monitor.c -lpsapi -o monitor
```

5.2 处理器信息

使用“/p”选项打印处理器信息。

```
> monitor /p
Processor Information
---------------------------------------
Processor architecture : 9
Logical processors     : 4
Processors mask        : 0x0000000f
Processor level        : 6
Processor revision     : 17665

Total processes        : 138
Total threads          : 1571
Total handles          : 51706
```

5.3 内存信息

使用“/m”选项打印内存信息。

```
> monitor /m
Memory Infomation
---------------------------------------
Physical memory in use : 48%

Page size              : 4 KB
Lowest memory address  : 0x00010000
Highest memory address : 0xfffeffff
Granularity            : 64 KB

Total physical memory  : 4001.078 MB
Available physical memory : 2041.578 MB

Total page file        : 5345.078 MB
Available page file    : 2788.961 MB

Total virtual memory   : 128.000 TB
Available virtual memory : 128.000 TB

Committed pages        : 654366 Page
System cache memory    : 347536 Page
Total kernel pages     : 80754 Page
Paged kernel pages     : 57935 Page
Nonpaged kernel pages  : 22819 Page
```

5.4 进程列表

使用“/l”选项打印进程列表。从输出中可以看到，Windows 资源管理器进程 explorer 的进程 ID 号是 5624。

```
> monitor /l
Name                 PID    PPID   Threads  Priority
-------------------- ----- ----- -------  --------
[System Process]       0      0        4         0
System                 4      0      139         8
smss.exe             368      4        2        11
```

```
csrss.exe              552    476      10       13
wininit.exe            656    476       1       13
csrss.exe              676    648      15       13
services.exe           728    656       8        9
lsass.exe              744    656       9        9
svchost.exe            852    728       2        8
fontdrvhost.exe        860    656       5        8
svchost.exe            912    728      26        8
WUDFHost.exe           940    728       8        8
winlogon.exe           444    648       4       13
fontdrvhost.exe       1044    444       5        8
dwm.exe               1152    444      12       13
svchost.exe           1192    728       4        8
explorer.exe          5624   5600      89        8
...
miktex-texworks.exe   3356   5624      19        8
svchost.exe           9516    728       5        8
smartscreen.exe       5840    912      10        8
cmd.exe               2916   5624       1        8
conhost.exe            256   2916      11        8
monitor.exe           2604   2916       3        8
```

5.5 进程虚拟地址空间

使用“/v”选项输出指定进程虚拟地址空间的使用情况，例如要查看 Windows 资源管理器进程 explorer 的虚拟地址空间，进程列表中得知进程 ID 号是 5624，则输入命令`monitor /v 5624`。

对输出的每一列说明如下：

No.： 虚拟空间块编号。从输出中可以看出 explorer 进程的地址空间被分为了 3410 块。

Address： 每一块的起始地址和结束地址，以 16 进制表示。

Size/B： 每一块的大小，以字节位单位。

Pages： 每一块所占的内存页数。

State： 每一块地址空间的状体，分为空闲（Free）、提交（Committed）、保留（Reversed）状态；

Protect： 保护状态。第一位标志 R 表示可读，表示不可读，相应的，W 表示可写，C 表示写时复制，X 表示可执行，G 表示 Guard 状态

Type： 类型。Mapped 表示映射到文件对象，Image 表示对应可执行映像，Private 表示空间私有，不与其他进程共享。

```
> monitor /v 5624
No.   Address                      Size/B    Pages   State     Protect  Type
----  ------------------------- ---------- ------ --------- ------- -------
   1  000000010000-000000140000  1245184    304   Free
   2  000000140000-000000150000    65536     16   Committed RW--     Mapped
   3  000000150000-000000153000    12288      3   Committed R---     Mapped
   4  000000153000-000000160000    53248     13   Free
   5  000000160000-000000179000   102400     25   Committed R---     Mapped
   6  000000179000-000000180000    28672      7   Free
   7  000000180000-0000001ef000   454656    111   Reversed           Private
   8  0000001ef000-0000001f2000    12288      3   Committed RW-G     Private
   9  0000001f2000-000000200000    57344     14   Committed RW--     Private
  10  000000200000-000000201000     4096      1   Reversed           Private

  ......

3404  7ffcadd13000-7ffcadd59000   286720     70   Committed R---     Image
3405  7ffcadd59000-7ffcadd5a000     4096      1   Committed RW--     Image
3406  7ffcadd5a000-7ffcadd5c000     8192      2   Committed -C--     Image
3407  7ffcadd5c000-7ffcadd61000    20480      5   Committed RW--     Image
3408  7ffcadd61000-7ffcadde0000   520192    127   Committed R---     Image
3409  7ffcadde0000-7ffffffe0000 1377828864 336384 Free
3410  7ffffffe0000-7fffffff0000    65536     16   Reversed           Private
```

5.6 进程模块

使用“/d”选项输出指定进程的模块信息。例如，若要查看 explorer 进程所加载的模块，其进程 ID 号是 5624，则输出命令monitor /d 5624，程序输出每个模块的文件名，起始地址，占用地址空间大小，存储路径信息。如下所示。

```
> monitor /d 5624
```

```
Module Name         Address   Size(KB)  Path
------------------- -------- -------- -------------------------
Explorer.EXE        f38c0000  3895296  C:\Windows\Explorer.EXE
ntdll.dll           adc00000  1966080  C:\Windows\SYSTEM32\ntdll.dll
KERNEL32.DLL        ada60000   712704  C:\Windows\System32\KERNEL32.DLL
KERNELBASE.dll      aabe0000  2514944  C:\Windows\System32\KERNELBASE.dll
msvcrt.dll          ad910000   643072  C:\Windows\System32\msvcrt.dll
combase.dll         ab5a0000  3178496  C:\Windows\System32\combase.dll
ucrtbase.dll        aa8c0000  1007616  C:\Windows\System32\ucrtbase.dll
RPCRT4.dll          ad6c0000  1175552  C:\Windows\System32\RPCRT4.dll

  ......

nlaapi.dll          a64b0000    98304  C:\Windows\System32\nlaapi.dll
shutdownux.dll      84170000   299008  C:\Windows\system32\shutdownux.dll
WINBRAND.dll        97a10000   110592  C:\Windows\system32\WINBRAND.dll
daxexec.dll         93530000   552960  C:\Windows\SYSTEM32\daxexec.dll
container.dll       93440000   233472  C:\Windows\System32\container.dll
wpnapps.dll         8c000000  1269760  C:\Windows\System32\wpnapps.dll
```

6 心得体会

进行了本实验后我切实了解了 Windows 中进程的地址空间，知道了如何利用操作系统的虚拟内存机制来增强程序对内存的管理的能力。

并且在实验过程中，通过调试各个 WINAPI 函数，逐渐了解了系统预定义的很多结构体以及函数其实是非常有利于程序员进行设计再开发的。在学习很多函数的头文件的时候，也发现了其实同一个功能，可能因为输入参数的不同数据类型，在库函数中就需要定义相类似的很多函数，这也是大大加强了库函数的适应能力，非常值得借鉴。

我还加深了对空闲、私有、映射、映像这四种状态的理解。在应用程序中，申请空间的过程称作保留 (预订)，可以用 VirtualAlloc，删除空间的过程为释放，可以用 VirtualFree。在程序里预订了地址空间以后，你还不可以存取数据，就和吃饭一样，因为你还没有付钱，没有真实的 RAM 和它关联。将此段区域标记为提交即可操作这块内存。默认情况下，区域状态是空闲，当 exe 或 dll 文件被映射进了进程空间后，区域状态变成映像；当一般数据文件被映射进了进程空间后，区域状态变成映射。

A 附录

A.1 monitor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <psapi.h>
#include <tlhelp32.h>

#define WIDTH 26
#define KILO 1024ll
#define MEGA (KILO*KILO)
#define GIGA (MEGA*KILO)
#define TERA (GIGA*KILO)

void PrintHeader(char *field[], int width[], int num)
{
    for (int i = 0; i < num; i++)
    {
        char fmt[10];
        sprintf(fmt, "%%-%ds ", width[i]);
        printf(fmt, field[i]);
    }
    printf("\n");
    for (int i = 0; i < num; i++)
    {
        for (int j = 0; j < width[i]; j++) printf("-");
        printf(" ");
    }
    printf("\n");
}

int ListProcessModules(int pid)
{
    HANDLE hModuleSnap = CreateToolhelp32Snapshot(
```

```
            TH32CS_SNAPMODULE, (DWORD)pid);
    if( hModuleSnap == INVALID_HANDLE_VALUE )
    {
        printf("Create module snapshot failed!\n");
        return 1;
    }

    char *field[] = {"Module Name", "Address", "Size(KB)", "
        Path"};
    int width[] = {20, 8, 8, 45};
    PrintHeader(field, width, 4);

    MODULEENTRY32 me32;
    me32.dwSize = sizeof( MODULEENTRY32 );
    BOOL bSucceed = Module32First(hModuleSnap, &me32);
    while (bSucceed)
    {
        printf("%-20s ", me32.szModule);
        printf("%08x ", me32.modBaseAddr);
        printf("%8d ", me32.modBaseSize);
        char path[50];
        strncpy(path, me32.szExePath, 45);
        printf("%s\n", path);
        bSucceed = Module32Next(hModuleSnap, &me32);
    }
    CloseHandle(hModuleSnap);
    return 0;
}

void GetState(DWORD mbiState, char state[])
{
    switch (mbiState)
    {
        case MEM_COMMIT: strcpy(state, "Committed"); break;
        case MEM_FREE: strcpy(state, "Free"); break;
        case MEM_RESERVE: strcpy(state, "Reversed"); break;
```

```
        default: strcpy(state, "Unknown"); break;
    }
}

void GetType(DWORD mbiType, char type[])
{
    switch (mbiType)
    {
        case MEM_IMAGE: strcpy(type, "Image"); break;
        case MEM_PRIVATE: strcpy(type, "Private"); break;
        case MEM_MAPPED: strcpy(type, "Mapped"); break;
        default: strcpy(type, ""); break;
    }
}

void GetProtect(DWORD mbiProtect, char protect[])
{
    strcpy(protect, "----");
    WORD wProtect = (WORD)(mbiProtect & 0x00ff);
    switch (wProtect)
    {
        case PAGE_EXECUTE: strncpy(protect, "--X", 3); break;
        case PAGE_EXECUTE_READ: strncpy(protect, "R-X", 3);
            break;
        case PAGE_EXECUTE_READWRITE: strncpy(protect, "RWX", 3)
            ; break;
        case PAGE_EXECUTE_WRITECOPY: strncpy(protect, "RCX", 3)
            ; break;
        case PAGE_NOACCESS: strncpy(protect, "---", 3); break;
        case PAGE_READONLY: strncpy(protect, "R--", 3); break;
        case PAGE_READWRITE: strncpy(protect, "RW-", 3); break;
        case PAGE_WRITECOPY: strncpy(protect, "-C-", 3); break;
        default: strncpy(protect, "---", 3); break;
    }
    if (mbiProtect & PAGE_GUARD) protect[3] = 'G';
    if (mbiProtect & PAGE_NOCACHE) protect[3] = 'N';
```

```
    if (mbiProtect & PAGE_WRITECOMBINE) protect[3] = 'B';
}

void PrintVirtulMemory(int pid)
{
    HANDLE hProcess = OpenProcess(
           PROCESS_ALL_ACCESS | PROCESS_VM_READ,
           FALSE, (DWORD)pid);
    if (hProcess == NULL)
    {
        printf("Open process '%d' failed!\n", pid);
        exit(1);
    }

    char *field[] = {
        "No.", "Address", "Size/B", "Pages",
        "State", "Protect", "Type"};
    int width[] = {4, 17, 10, 6, 9, 7, 7};
    PrintHeader(field, width, 7);

    SYSTEM_INFO si;
    GetSystemInfo(&si);

    MEMORY_BASIC_INFORMATION mbi;
    ZeroMemory(&mbi, sizeof(mbi));

    int count = 0;
    LPCVOID pBlock = si.lpMinimumApplicationAddress;
    while (pBlock < si.lpMaximumApplicationAddress)
    {
        VirtualQueryEx(hProcess, pBlock, &mbi, sizeof(mbi));

        printf("%4d ", ++count);
        LPCVOID pEnd = pBlock + mbi.RegionSize;
        printf("%08x-%08x ", pBlock, pEnd);
        printf("%10u ", mbi.RegionSize);
```

```
            printf("%6u ", mbi.RegionSize / si.dwPageSize);

            char state[10];
            GetState(mbi.State, state);
            printf("%-9s ", state);

            char protect[10] = "";
            if (mbi.State == MEM_COMMIT)
                GetProtect(mbi.Protect, protect);
            printf("%-7s ", protect);

            char type[10];
            GetType(mbi.Type, type);
            printf("%-7s \n", type);

            pBlock = pEnd;
        }
    }

    void PrintMemoryInfo()
    {
        printf("Memory Infomation\n");
        printf("----------------------------------------\n");

        MEMORYSTATUSEX ms;
        ms.dwLength = sizeof(ms);
        GlobalMemoryStatusEx(&ms);

        printf("%-*s: %d%%\n", WIDTH, "Physical memory in use",
               ms.dwMemoryLoad);
        printf("\n");

        SYSTEM_INFO si;
        GetSystemInfo(&si);
        printf("%-*s: %d KB\n", WIDTH, "Page size",
               si.dwPageSize / KILO);
```

```
printf("%-*s: 0x%08x\n", WIDTH, "Lowest memory address",
       si.lpMinimumApplicationAddress);
printf("%-*s: 0x%08x\n", WIDTH, "Highest memory address",
       si.lpMaximumApplicationAddress);
printf("%-*s: %d KB\n", WIDTH, "Granularity",
       si.dwAllocationGranularity / KILO);
printf("\n");

printf("%-*s: %.3lf MB\n", WIDTH,"Total physical memory",
       (double)ms.ullTotalPhys / MEGA);
printf("%-*s: %.3lf MB\n", WIDTH, "Available physical
    memory",
       (double)ms.ullAvailPhys / MEGA);
printf("\n");
printf("%-*s: %.3lf MB\n", WIDTH, "Total page file",
       (double)ms.ullTotalPageFile / MEGA);
printf("%-*s: %.3lf MB\n", WIDTH, "Available page file",
       (double)ms.ullAvailPageFile / MEGA);
printf("\n");
printf("%-*s: %.3lf TB\n", WIDTH, "Total virtual memory",
       (double)ms.ullTotalVirtual / TERA);
printf("%-*s: %.3lf TB\n", WIDTH, "Available virtual memory
    ",
       (double)ms.ullAvailVirtual / TERA);
printf("\n");

PERFORMANCE_INFORMATION pi;
GetPerformanceInfo(&pi, sizeof(pi));
printf("%-*s: %d Page\n", WIDTH, "Committed pages", pi.
    CommitTotal);
printf("%-*s: %d Page\n", WIDTH, "System cache memory", pi.
    SystemCache);
printf("%-*s: %d Page\n", WIDTH, "Total kernel pages", pi.
    KernelTotal);
printf("%-*s: %d Page\n", WIDTH, "Paged kernel pages", pi.
    KernelPaged);
```

```
    printf("%-*s: %d Page\n", WIDTH, "Nonpaged kernel pages",
        pi.KernelNonpaged);
}

void PrintProcessorInfo()
{
    printf("Processor Information\n");
    printf("---------------------------------------\n");

    SYSTEM_INFO si;
    GetSystemInfo(&si);
    printf("%-*s: %d\n", WIDTH, "Processor architecture",
           si.wProcessorArchitecture);
    printf("%-*s: %d\n", WIDTH, "Logical processors",
           si.dwNumberOfProcessors);
    printf("%-*s: 0x%08x\n", WIDTH, "Processors mask",
           si.dwActiveProcessorMask);
    printf("%-*s: %d\n", WIDTH, "Processor level",
           si.wProcessorLevel);
    printf("%-*s: %d\n", WIDTH, "Processor revision",
           si.wProcessorRevision);
    printf("\n");

    PERFORMANCE_INFORMATION pi;
    GetPerformanceInfo(&pi, sizeof(pi));
    printf("%-*s: %d\n", WIDTH, "Total processes", pi.
        ProcessCount);
    printf("%-*s: %d\n", WIDTH, "Total threads", pi.ThreadCount
        );
    printf("%-*s: %d\n", WIDTH, "Total handles", pi.HandleCount
        );
}

int PrintProcessList()
{
```

```
    char *field[] = {"Name", "PID", "PPID", "Threads", "
        Priority"};
    int width[] = {21, 5, 5, 7, 8};
    PrintHeader(field, width, 5);

    HANDLE hProcessSnap = CreateToolhelp32Snapshot(
        TH32CS_SNAPPROCESS, 0);
    if(hProcessSnap == INVALID_HANDLE_VALUE)
    {
        printf("CreateToolhelp32Snapshot Failed!\n");
        return 1;
    }

    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    BOOL bSucceed = Process32First(hProcessSnap, &pe32);
    while (bSucceed)
    {
        char name[30];
        strncpy(name, pe32.szExeFile, 21);
        printf("%-21s ", name);
        printf("%5d ", pe32.th32ProcessID);
        printf("%5d ", pe32.th32ParentProcessID);
        printf("%7d ", pe32.cntThreads);
        printf("%8d\n", pe32.pcPriClassBase);
        bSucceed = Process32Next(hProcessSnap, &pe32);
    }
    CloseHandle(hProcessSnap);
    return 0;
}

void PrintUsage()
{
    printf("monitor [/p] [/m] [/l] [/v <pid>] [/d <pid>]\n\n");
    printf(" %-10s", "/p");
```

```
    printf("Show processer infomation.\n");
    printf(" %-10s", "/m");
    printf("Show memory infomation.\n");
    printf(" %-10s", "/l");
    printf("List the running processes.\n");
    printf(" %-10s", "/s <pid>");
    printf("List the virtual memory of process <pid>.\n");
    printf(" %-10s", "/d <pid>");
    printf("List the modules of process <pid>.\n");
    printf("\n");
}

int main(int argc, char *argv[])
{
    if (argc == 1)
    {
        printf("monitor: missing option\n\n");
        PrintUsage();
    }
    else if (strcmp(argv[1], "/?") == 0)
    {
        printf("Show system infomation.\n\n");
        PrintUsage();
    }
    else if (strcmp(argv[1], "/p") == 0)
    {
        PrintProcessorInfo();
    }
    else if (strcmp(argv[1], "/m") == 0)
    {
        PrintMemoryInfo();
    }
    else if (strcmp(argv[1], "/l") == 0)
    {
        PrintProcessList();
    }
```

```
    else if (strcmp(argv[1], "/v") == 0)
    {
        int pid = strtol(argv[2], NULL, 10);
        if (pid > 0) PrintVirtulMemory(pid);
        else printf("Invalid pid '%s'\n", argv[2]);
    }
    else if (strcmp(argv[1], "/d") == 0)
    {
        int pid = strtol(argv[2], NULL, 10);
        if (pid > 0) ListProcessModules(pid);
        else printf("Invalid pid '%s'\n", argv[2]);
    }
    else
    {
        printf("monitor: undefined option\n\n");
        PrintUsage();
    }
    return 0;
}
```

A.2 makefile

```
monitor: monitor.c
    gcc monitor.c -lpsapi -o monitor
```