实验3 内存监视

班级： 07112002 学号： 1120200822 姓名： 郑子帆

# 一、实验目的

独立设计并实现一个内存监视器，以加深对内存管理的理解。

# 二、实验内容

1. 在Windows系统下设计实现一个内存监视器。
2. 使用该内存监视器：  
   能够实时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。
3. 可使用的相关系统调用包括：GetSystemInfo, VirtualQueryEx, VirtualAlloc, GetPerformanceInfo, GlobalMemoryStatusEx等。

# 三、实验步骤

## 3.1 实验环境

1. 硬件配置：Intel(R)Core(TM)i7-10750H CPU @ 2.60GHz，6个内核

2. 操作系统：Microsoft Windows 10家庭中文版

3. 代码编辑器：Microsoft Visual Studio Code 2020

4. 编译器：MinGW64 g++

## 3.2 实验思路

编写程序MemoryMonitor，通过调用GetSystemInfo, VirtualQueryEx, GetPerformanceInfo, GlobalMemoryStatusEx等API以获得当前计算机系统信息（这部分实验没有要求）和内存使用情况（包括系统地址空间的布局、物理内存的使用情况等等）以及实现某个进程的虚拟地址空间布局和工作集信息等。

## 3.3 数据结构介绍

### 3.3.1 计算机系统信息结构体

具体定义如下：

typedef struct \_SYSTEM\_INFO {

  union {

    DWORD dwOemId;

    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, \*LPSYSTEM\_INFO;

其中部分成员介绍如下：

wProcessorArchitecture：表示了该安装操作系统中的处理器的架构。

取值有以下几种类型：

PROCESSOR\_ARCHITECTURE\_AMD64：用于x64架构

PROCESSOR\_ARCHITECTURE\_ARM：用于ARM架构

PROCESSOR\_ARCHITECTURE\_ARM64：用于ARM64(x86-64)架构

PROCESSOR\_ARCHITECTURE\_IA64：基于Intel安腾架构

PROCESSOR\_ARCHITECTURE\_INTEL：基于x86架构

PROCESSOR\_ARCHITECTURE\_UNKNOWN：未知的架构

dwPageSize：页面大小和页面保护和提交的粒度；

lpMinimumApplictionAddress：指向应用程序和动态链接库（DLL）可访问的最低内存指针；

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

dwActiveProcessorMask：系统配备的处理器集掩码。其中位0代表CPU0，位1代表CPU1，以此类推；

dwNumberOfProcessors：当前组中的逻辑处理器数；

dwAllocationGranularity：能够保留地址空间的最小单位；

### 3.3.2 内存状态结构体

具体定义如下：

typedef struct \_MEMORYSTATUSEX {

    DWORD     dwLength;

    DWORD     dwMemoryLoad;

    DWORDLONG ullTotalPhys;

    DWORDLONG ullAvailPhys;

    DWORDLONG ullTotalPageFile;

    DWORDLONG ullAvailPageFile;

    DWORDLONG ullTotalVirtual;

    DWORDLONG ullAvailVirtual;

    DWORDLONG ullAvailExtendedVirtual;

} MEMORYSTATUSEX, \*LPMEMORYSTATUSEX;

其中部分成员介绍如下：

dwLength：这个结构体的大小；

dwMemoryLoad：物理内存大约使用的百分比；

ullTotalPhys：物理内存的大小；

ullAvailPhys：当前可用物理内存的大小；

ullTotalPageFile：页文件最大值；

ullAvailPageFile：可用页文件大小；

ullTotalVirtual：主调进程在用户模式下虚拟地址空间部分的大小；

ullAvailVirtual：主调进程在用户模式下的虚拟地址空间部分中未保留和未提交的内存大小。

ullAvailExtendedVirtual：保留。这个值总是0。

### 3.3.3 性能信息结构体

具体定义如下：

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, \*PPERFORMANCE\_INFORMATION, PERFORMACE\_INFORMATION, \*PPERFORMACE\_INFORMATION;

其中部分成员介绍如下：

cb：本结构体的大小（单位：byte）；

CommitTotal：当前系统中已被提交的页数；

CommitLimit：在没有扩大页文件的情况下，当前系统中可被允许提交的页数的最大值；

CommitPeak：自从上次系统重启后，可同时处于提交状态的页的最大数量；

PhysicalTotal：真正的物理内存大小（单位：页）；

PhysicalAvailable：当前可用的物理内存页的总数；

SystemCache：系统缓冲内存页的数量；

KernelTotal：当前分页及非分页的内核池的总数（单位：页）；

KernelPaged：当前分页的内核池的页数；

KernelNonpaged：当前未分页的内核池的页数；

PageSize：页的大小（单位：byte）；

HandleCount：当前打开的句柄数量；

ProcessCount：当前的进程数量；

ThreadCount：当前的线程数量。

3.3.4 虚拟地址空间信息结构体

typedef struct \_MEMORY\_BASIC\_INFORMATION {

    PVOID  BaseAddress;

    PVOID  AllocationBase;

    DWORD  AllocationProtect;

    SIZE\_T RegionSize;

    DWORD  State;

    DWORD  Protect;

    DWORD  Type;

} MEMORY\_BASIC\_INFORMATION, \*PMEMORY\_BASIC\_INFORMATION;

其中部分成员介绍如下：

BaseAddress：分配包含基址的最小页号;

AllocationBase：应用程序的实际起始地址;

AllocationProtect：该区域初始设置的访问方式;

RegionSize：虚存区的大小;

State：该区域的状态是空闲、预留还是提交（MEM\_FREE、MEM\_RESERVE或 MEM\_COMMIT）;

Protect：该区域设置的访问方式。这个值是AllocationProtect中的一项;

Type：该区域的页面类型，即可执行映像（MEM\_IMAGE）、内存映像文件（MEM\_MAPPING）或私有内存区（MEM\_PRIVATE）。

## 3.4 系统调用API介绍

### 3.4.1 GetSystemInfo

Windows使用GetSystemInfo函数获取当前系统的信息。语法如下：

void GetSystemInfo(

  LPSYSTEM\_INFO lpSytstemInfo

);

参数介绍：

lpSystemInfo：指向接受信息的SYSTEM\_INFO结构体指针。

### 3.4.2 GlobalMemoryStatusEx

Windows使用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。语法如下：

BOOL GlobalMemoryStatusEx(

  LPMEMORYSTATUSEX lpBuffer

);

其中，lpBuffer为指向MEMORYSTATUSEX结构的指针，该结构接受有关当前内存可用性的信息。

### 3.4.3 GetPerformanceInfo

Windows使用GetPerformanceInfo函数获取性能信息。语法如下：

BOOL GetPerformanceInfo(

    PPERFORMANCE\_INFORMATION pPerformanceInformation,

    DWORD cb

);

其中，pPerformanceInformation为指向PERFORMANCE\_INFORMATION结构的指针。

### 3.4.4 EnumProcesses

Windows使用EnumProcesses函数获取进程列表信息。语法如下：

BOOL EnumProcesses(

  DWORD   \*lpidProcess,

  DWORD   cb,

  LPDWORD lpcbNeeded

);

其中，lpidProcess是一个指向进程标识符数组的指针，lpcbNeeded是lpidProcess数组返回的字节数。

### 3.4.5 OpenProcess

Windows使用OpenProcess函数打开一个已存在的局部进程。语法如下：

HANDLE OpenProcess(

    DWORD dwDesiredAccess,

    BOOL  bInheritHandle,

    DWORD dwProcessId

);

其中，dwProcessId是将要被打开的局部进程的标识符。

### 3.4.6 EnumProcessModules

Windows使用EnumProcessModules函数获取当前进程中每一个模块的句柄。语法如下：

BOOL EnumProcessModules(

    HANDLE  hProcess,

    HMODULE \*lphModule,

    DWORD   cb,

    LPDWORD lpcbNeeded

);

其中，hProcess是一个指向该进程的句柄。

### 3.4.7 GetModuleBaseName

Windows使用GetModuleBaseName函数获取模块名称。语法如下：

DWORD GetModuleBaseName(

    HANDLE  hProcess,

    HMODULE hModule,

    LPSTR   lpBaseName,

    DWORD   nSize

);

其中，lpBaseName是指向接收模块基本名的缓冲区的指针。

# 四、实验结果及分析

## 4.1 代码分解

此处仅展示现时主要功能的代码，完整代码详见MemoryMonitor.cpp文件。

### 4.1.1 ShowSystemInfo函数

最终实现的代码如下：

1. void ShowSystemInfo(){
2. printStart("ShowSystemInfo");
3. SYSTEM\_INFO lpSystemInfo;
4. ZeroMemory(&lpSystemInfo, sizeof(lpSystemInfo)); *// 用0来填充一块内存区域，得到计算机系统信息*
5. GetSystemInfo(&lpSystemInfo);
6. printf("Processor architecture:\t\t\t\t"); *// 系统架构*
7. switch (lpSystemInfo.wProcessorArchitecture){
8. case 9:
9. printf("x64 (AMD or Intel)\n");
10. break;
11. case 5:
12. printf("ARM\n");
13. break;
14. case 12:
15. printf("x86-64\n");
16. break;
17. case 6:
18. printf("Intel Itanium-based\n");
19. break;
20. case 0:
21. printf("x86\n");
22. break;
23. default:
24. printf("Unknown architecture.\n");
25. break;
26. }
27. printf("Page size:\t\t\t\t\t"); *// 页面大小*
28. printf("%lu Bytes\n", lpSystemInfo.dwPageSize);
29. printf("Lowest memory address:\t\t\t\t"); *// 最低地址*
30. printf("0x%016llx\n", (unsigned long long)lpSystemInfo.lpMinimumApplicationAddress); *// 16位16进制地址*
31. printf("Highest memory address:\t\t\t\t"); *// 最高地址*
32. printf("0x%016llx\n", (unsigned long long)lpSystemInfo.lpMaximumApplicationAddress);
33. printf("Active processpor mask:\t"); *// 处理器集掩码*
34. printIn32bits(lpSystemInfo.dwActiveProcessorMask);
35. printf("Number of processors:\t\t\t\t"); *// 逻辑处理器数量*
36. printf("%lu\n", lpSystemInfo.dwNumberOfProcessors);
37. printf("Allocation Granularity:\t\t\t\t");
38. printf("%lu Bytes\n", lpSystemInfo.dwAllocationGranularity);
39. printEnd("ShowSystemInfo");
40. }

主要功能为：

输出系统架构（x64/arm/x86-64/Intel Itanium-based/x86）

输出页面大小；

输出内存最低、最高地址；

输出活动处理器集掩码；

输出逻辑处理器数量；

输出分配粒度。

### 4.1.2 ShowMemoryInfo函数

最终实现的代码如下：

1. void ShowMemoryInfo(){     *// 内存信息*
2. printStart("ShowMemoryInfo");
3. MEMORYSTATUSEX lpMemoryInfo;
4. lpMemoryInfo.dwLength = sizeof(lpMemoryInfo);
5. GlobalMemoryStatusEx(&lpMemoryInfo);
6. printf("Percentage of memory in use:\t\t\t%20ld%%\n", lpMemoryInfo.dwMemoryLoad);   *// 内存使用百分比*
7. printf("Total physical memory:\t\t\t\t%18lld KB\n", lpMemoryInfo.ullTotalPhys / DIV);  *// 物理内存大小*
8. printf("Free physical memory:\t\t\t\t%18lld KB\n", lpMemoryInfo.ullAvailPhys / DIV);  *// 可用物理内存大小*
9. printf("Total paging file:\t\t\t\t%18lld KB\n", lpMemoryInfo.ullTotalPageFile / DIV);  *// 页文件大小*
10. printf("Free paging file:\t\t\t\t%18lld KB\n", lpMemoryInfo.ullAvailPageFile / DIV);  *// 可用页文件大小*
11. printf("Total virtual memory:\t\t\t\t%18lld KB\n", lpMemoryInfo.ullTotalVirtual / DIV); *// 虚拟内存*
12. printf("Free virtual memory:\t\t\t\t%18lld KB\n\n", lpMemoryInfo.ullAvailVirtual / DIV);*// 虚拟内存可用大小*
13. PERFORMANCE\_INFORMATION lpPerformanceInformation;
14. lpPerformanceInformation.cb = sizeof(lpPerformanceInformation);
15. GetPerformanceInfo(&lpPerformanceInformation, sizeof(lpPerformanceInformation));
16. printf("Total physical pages:\t\t\t\t%18llu\n", lpPerformanceInformation.PhysicalTotal);  *// 物理页数量*
17. printf("Total committed pages:\t\t\t\t%18llu\n", lpPerformanceInformation.CommitTotal);  *// 已提交页数量*
18. printf("System cache:\t\t\t\t\t%18llu\n", lpPerformanceInformation.SystemCache);    *// 系统缓存*
19. printf("Total handles:\t\t\t\t\t%18lu\n", lpPerformanceInformation.HandleCount);    *// 句柄数量*
20. printf("Total processes:\t\t\t\t%18lu\n", lpPerformanceInformation.ProcessCount);    *// 进程数量*
21. printf("Total threads:\t\t\t\t\t%18lu\n", lpPerformanceInformation.ThreadCount);    *// 线程数量*
22. printEnd("ShowMemoryInfo");
23. }

### 4.1.3 ShowProcessInfo函数

最终实现的代码如下：

1. void ShowProcessInfo(){    *// 进程信息*
2. printStart("ShowProcessInfo");
3. char process\_name[100] = {0};
4. printf("Please input the process name:\n");
5. fgets(process\_name, sizeof(process\_name), stdin);
6. int length = strlen(process\_name);
7. process\_name[length - 1] = '\0';
8. DWORD pid\_array[max\_process\_num], lpcbNeeded;
9. EnumProcesses(pid\_array, sizeof(pid\_array), &lpcbNeeded);
10. if (lpcbNeeded == sizeof(pid\_array)) {
11. printf("Larger array should be declared to save the pid array!\n");
12. exit(1);
13. }
14. int process\_num = lpcbNeeded / sizeof(DWORD);
15. printf("The number of process:%d\n", process\_num);
16. HANDLE required\_process[max\_process\_num];
17. int required\_num = 0;
18. for (int i = 0; i < process\_num; i++) {
19. HANDLE hProcess = OpenProcess( PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, pid\_array[i]);
20. if (hProcess == NULL) continue;
22. char now\_process\_name[100];
23. getProcessName(hProcess, now\_process\_name);
24. *//printf("%s\n", now\_process\_name);*
25. if (strcmp(process\_name, now\_process\_name) == 0) {
26. required\_process[required\_num ++] = hProcess;
27. }
28. }
29. printf("The number of process named '%s':%d\n\n", process\_name, required\_num);
30. for (int i = 0; i < required\_num; i++) {
31. ShowProcessVirualMemory(required\_process[i], i);
32. }
33. printEnd("ShowProcessInfo");
34. }

### 4.1.4 ShowProcessVirualMemory函数

1. void ShowProcessVirualMemory(HANDLE hProcess, int idx){ *// 虚拟地址信息*
2. printf("Process No. %d\n", idx);
3. printf("Address\t\t\t\t\tKbytes\t\tMode\tState\t\tType\n");
4. SYSTEM\_INFO lpSystemInfo;
5. ZeroMemory(&lpSystemInfo, sizeof(lpSystemInfo));
6. GetSystemInfo(&lpSystemInfo);
7. MEMORY\_BASIC\_INFORMATION lpMemoryInfo;
8. ZeroMemory(&lpMemoryInfo, sizeof(lpMemoryInfo));
9. unsigned long long pRegionStart = (unsigned long long)lpSystemInfo.lpMinimumApplicationAddress;
10. while (1){
11. if (VirtualQueryEx(hProcess, (LPCVOID)pRegionStart, &lpMemoryInfo, sizeof(lpMemoryInfo)) != sizeof(lpMemoryInfo)) break;
12. unsigned long long pRegionEnd = pRegionStart + lpMemoryInfo.RegionSize-1;
13. printf("%016llx-%016llx\t", pRegionStart, pRegionEnd);  *// 块长度*
14. if (lpMemoryInfo.RegionSize / DIV < 10000000) printf("%llu\t\t", lpMemoryInfo.RegionSize / DIV);
15. else printf("%llu\t", lpMemoryInfo.RegionSize / DIV); *// 大小*
16. switch (lpMemoryInfo.Protect & 0x00ff) { *// 保护*
17. case PAGE\_EXECUTE:
18. printf("--X");
19. break;
20. case PAGE\_EXECUTE\_READ:
21. printf("R-X");
22. break;
23. case PAGE\_EXECUTE\_READWRITE:
24. printf("RWX");
25. break;
26. case PAGE\_EXECUTE\_WRITECOPY:
27. printf("RCX");
28. break;
29. case PAGE\_NOACCESS:
30. printf("---");
31. break;
32. case PAGE\_READONLY:
33. printf("R--");
34. break;
35. case PAGE\_READWRITE:
36. printf("RW-");
37. break;
38. case PAGE\_WRITECOPY:
39. printf("-C-");
40. break;
41. default:
42. printf("---");
43. }
44. printf("\t");
45. switch (lpMemoryInfo.State) { *// 状态*
46. case MEM\_COMMIT:
47. printf("committed");
48. break;
49. case MEM\_FREE:
50. printf("free\t");
51. break;
52. case MEM\_RESERVE:
53. printf("reserved");
54. break;
55. default:
56. printf("?\t");
57. }
58. printf("\t");
59. switch (lpMemoryInfo.Type) { *// 类型*
60. case MEM\_IMAGE:
61. printf("image");
62. break;
63. case MEM\_MAPPED:
64. printf("mapped");
65. break;
66. case MEM\_PRIVATE:
67. printf("private");
68. break;
69. default:
70. printf("unknown");
71. }
72. printf("\n");
73. pRegionStart = pRegionEnd+1;
74. }
75. }

### 4.1.5 主函数

1. int main(){
2. char command\_str[50] = {0};
3. int command\_len;
4. int quit = 0;
5. while (1) {
6. printf("Please input the command:\n");
7. if (fgets(command\_str, sizeof(command\_str), stdin) == NULL) break;
8. command\_len = strlen(command\_str);
9. if (command\_len != 2){
10. ShowError();
11. continue;
12. }
13. if(command\_str[0] == 's'){
14. ShowSystemInfo();
15. }
16. else if(command\_str[0] == 'm'){
17. ShowMemoryInfo();
18. }
19. else if(command\_str[0] == 'p'){
20. ShowProcessInfo();
21. }
22. else if(command\_str[0] == 'q'){
23. ShowQuit();
24. quit = 1;
25. }
26. else {
27. ShowError();
28. }
29. if (quit) break;
30. }
31. return 0;
32. }

其中：

‘s’命令：用于输出系统相关的信息。

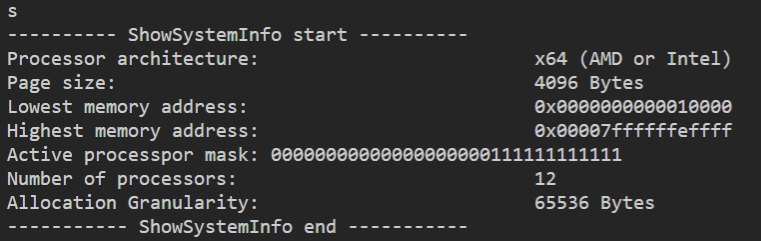
‘m’命令：用于输出内存相关的信息。

‘p’命令：用于输出特定名称进程的虚拟地址空间的布局信息等。

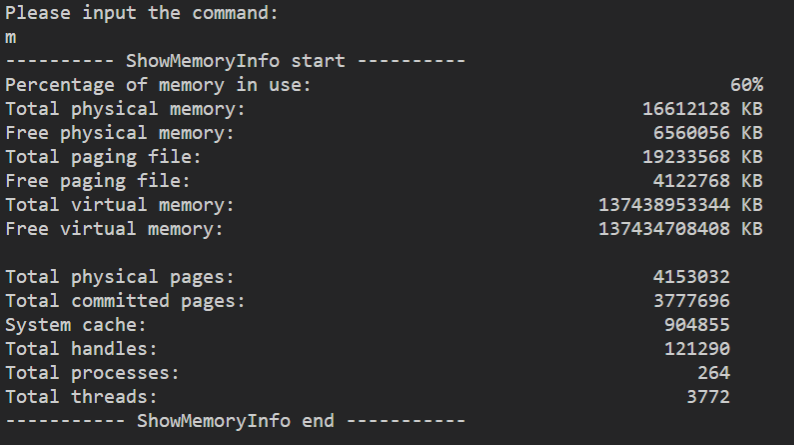
‘q’命令：退出程序。

## 4.2 运行结果截图

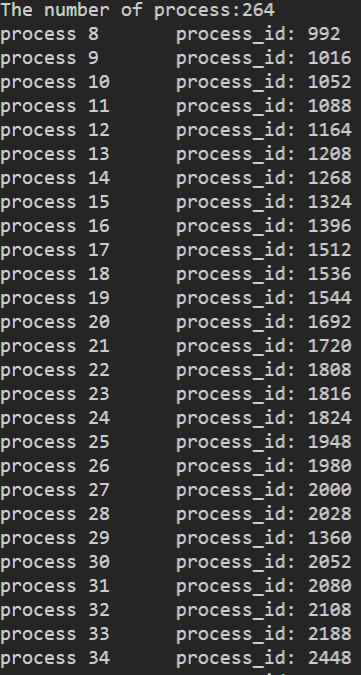
查询系统信息：



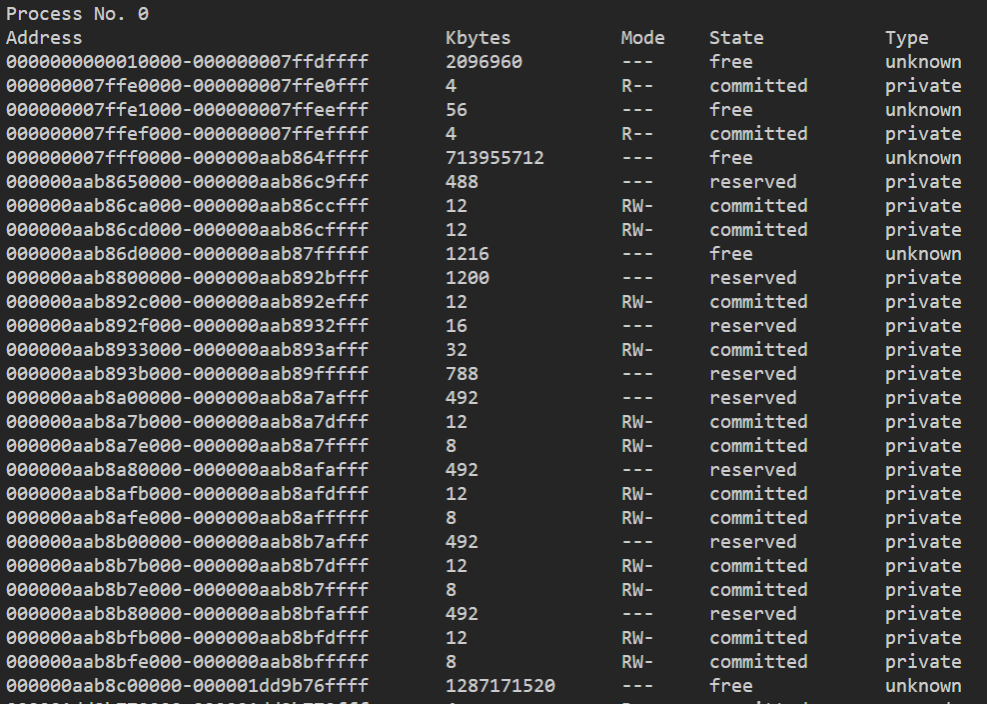
查询内存信息：



查询进程列表（部分截图）：



查询第10个进程（pid为1052）的虚拟地址空间布局（部分截图）：



# 五、实验收获与体会

## 5.1 实验中遇到的问题

1. 在编写完部分程序后，运行时如下类似报错：“undefined reference to XXX”，经过上网查阅资料后发现，从.cpp源代码到可执行的ELF文件过程中，要经过预处理、编译和链接三个阶段，此时预处理和编译已经通过了，但是链接出现了问题，故出现了报错。解决方案为：在编译的时候加入-lpsapi命令，进行链接，问题得以解决。

2. 在程序编写中，编译时遇到了“cannot convert type wchar\* to type char\* for argument xxx”，大意为wchar\*类型的字符指针无法转换为char\*类型的，尝试了强制转换也无法通过编译。通过上网查找资料，发现调用WideCharToMultiByte函数可以实现wchar\*和char\*类型的相互转化，故编写了wchar2char函数，解决了该问题。

3. 在尝试调用函数获得进程名称时，一直为乱码，考虑可能是编码方面的不兼容所导致的，该问题暂时还没有解决。

## 5.2 心得体会

通过这次实验，我学习了Windows操作系统中有关系统、内存、虚拟地址信息存储的数据结构以及相关的API使用方法，并通过上网查询资料进一步实现了一些API的调用以实现实验所要求的功能。在实验过程中，我也遇到了很多问题，其中大部分再后来得到了解决，还有一些待解决的问题我将在之后进一步学习思考并尝试解决。

附录：程序清单及说明

1120200822-郑子帆-07112002-实验3 内存监视.docx：实验报告

MemoryMonitor.cpp文件：内存监控实现代码

实验运行录屏.mp4：代码运行时的录屏文件