《操作系统》课程实验三

实验报告

存储管理问题

<Windows 虚拟内存分配问题>

班级：无 47

姓名：刘 前[1]

学号：2014011216

日期：2016 年 12 月 10 日

操作平台：Windows 8.1

编程语言：C++

[1] 清华大学电子工程系(E-mail: liuqian14@mails.tsinghua.edu.cn)

实验三 存储管理问题

实验题目

本实验共有 3 个实验题目，任选其中之一。不同实验的难度不同，基准分也不同，请同学根据自己的情况加以选择。

实验题目	基准分
① 文件字节倒放问题	90
② Windows 虚拟内存分配问题	95
③ AVL 树→红黑树问题	100 + 鼓励

实验报告内容要求

1. 写出设计思路和程序结构，并对主要代码进行分析；
2. 实际程序运行情况；
3. 对提出的问题进行解答；
4. 体会或者是遇到的问题。

注：本人在实验三中选择的是 **Windows 虚拟内存分配问题**。

Windows 虚拟内存分配问题

一、 问题描述及实验要求

1. 问题描述

使用 Win32 API 函数，编写一个包含两个线程的进程，一个线程用于模拟内存分配活动，一个线程用于跟踪第一个线程的内存行为。

要求模拟的操作包括保留一个区域、提交一个区域、回收一个区域、释放一个区域，以及锁定与解锁一个区域。跟踪线程要输出每次内存分配操作后的内存状态。

通过对内存分配活动的模拟和跟踪的编程实现，从不同侧面对 Windows 对用户进程的虚拟内存空间的管理、分配方法，对 Windows 分配虚拟内存、改变内存状态，以及对物理内存和页面文件状态查询的 API 函数的功能、参数限制、使用规则进一步有所了解。同时了解跟踪程序的编写方法。

2. 实验环境

操作系统平台为 Windows，编程语言不限。

3. 函数参考

VirtualAlloc 用来保留、提交虚拟内存区域；
VirtualLock 和 VirtualUnlock 用于锁定、解锁内存区域；
VirtualFree 用于回收、释放内存区域；
GetSystemInfo 用于返回关于当前系统的信息；
VirtualQuery 用于查询进程地址空间中内存地址的信息。

4. 实现提示

1. 模拟内存分配活动的线程 Allocator 与内存跟踪线程 Tracker 可以用一对信号量 allo 和 trac 来实现同步；

2. Allocator 线程先等待 Tracker 输出结束(allo 信号量被释放)，才能够进行内存分配操作，并且每进行一项内存操作就释放 trac 信号量，通知 Tracker 进行输出；

3. 跟踪线程 Tracker 在得到信号量 trac 之后开始输出内存状态，然后释放 allo 信号量，通知 Allocator 线程进行下一次内存分配操作。

5. 思考题

1) 分析不同参数的设置对内存分配的结果影响。

2) 如果调换分配、回收、内存复位、加锁、解锁、提交、回收的次序，会有什么变化，并分析原因。

3) 在 Linux 操作系统上没有提供系统调用来实现“以页为单位的虚拟内存分配方式”，如果用户希望实现这样的分配方式，可以怎样做？解释你的思路。

二、 设计思路

解决该问题的核心思路是创建一个包含两个线程的进程，一个线程(记作 Allocator)用于模拟内存的分配活动，另一个线程(记作 Tracker)用于跟踪第一个线程的内存行为。因而，首先需要设计这两个线程实现相应的功能。

对于 Allocator 线程，需要使用相应的 Win32 API 函数模拟内存的分配活动。问题中要求模拟 6 个针对内存的操作，分别为：保存、提交、回收、释放、锁定及解锁。其中保存和提交需要由 VirtualAlloc 函数实现，回收和释放由 VirtualFree 函数实现，锁定由 VirtualLock 函数实现，函数具体的使用方法可以查询 MSDN 得到，在本实验报告的附录中也对各函数的参数和使用方法进行了说明。

对于 Tracker 线程，要跟踪 Allocator 线程并输出 Allocator 线程每次内存分配操作后的内存状态。对内存状态的检测由 VirtualQuery 函数实现，能够获取相关内存区域的所有基本信息，再进行输出操作，即可实现对内存状态的检测。

设计得到两个线程之后，关键问题是实现两个线程的同步以及 Tracker 线程对 Allocator 线程的跟踪。这可以由一对信号量实现(记作 allo 和 trac)。主要实现方法为：Allocator 线程每进行一项内存操作，释放 trac 信号量，Tracker 线程接收到信号量 trac 后获取当前的内存状态信息，并输出内存状态；输出结束后，allo 信号量被释放，通知 Allocator 线程进行下一次内存分配操作，Allocator 线程得到信号量 allo 之后，执行下一次内存分配操作。

根据以上思路，即可实现对内存分配的模拟和对内存状态的跟踪。

三、 程序结构

1. 基本数据结构和函数

两个线程，分别为 Allocator 和 Tracker，在程序中表现为两个函数；两个信号量分别为 allo 和 trac。在定义信号量时，参数中需要设定信号量的最大资源数，因为本次实验中共执行 6 次内存分配操作，每次内存分配操作都需要释放 allo 和 trac 信号量，所以两个信号量的最大资源数全部设定为 6。

2. 获取系统信息

因为模拟内存分配活动时，需要使用系统内存的页大小信息，所以首先使用 GetSystemInfo 函数获取当前系统信息。

函数说明：

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

GetSystemInfo 函数只有一个参数，lpSystemInfo 相当于函数的返回结果，是一个 SYSTEM_INFO 变量的指针，函数运行结束后，lpSystemInfo 获取到当前系统的信息。

SYSTEM_INFO 结构体内存储的系统信息主要包括：

参数类型	形参	解释说明
DWORD	dwOemId	为兼容性保留的废弃成员
DWORD	dwPageSize	页大小
LPVOID	lpMinimumApplicationAddress	指向最低的应用程序地址
LPVOIDDWOR	lpMaximumApplicationAddress	指向最高的应用程序地址
D_PTR	dwActiveProcessorMask	课理解为系统中的处理器
DWORD	dwNumberOfProcessors	逻辑处理器的数量
DWORD	dwProcessorType	处理器类型
DWORD	dwAllocationGranularity	内存分配起始地址
WORD	wProcessorLevel	处理器等级
•••	•••	•••

3. 输出内存状态信息

每次内存分配操作后需要在控制台窗口输出内存的状态，内存的状态可以由 VirtualQuery 函数查询，查询后的结果存储到一个 MEM_BASIC_INFORMATION 结构体中。但是该结构体的数据成员都不是字符串型，因而不能直接输出，需要先将这些信息转换为字符串的形式，再进行输出。

转换方法比较简单，只需要使用 switch 函数根据数据成员取值的不同，转换为相同语义的字符串即可。

4. 创建线程

两个线程实现对内存分配的模拟和对内存操作的跟踪。具体实现方式和使用函数请见代码分析部分

四、 代码分析

1. 全局变量：

```
HANDLE trac = CreateSemaphore(NULL,0,OperationNum,NULL);
HANDLE allo = CreateSemaphore(NULL,0,OperationNum,NULL);
```

信号量 trac，完成一次内存分配操作后被释放，Tracker 线程输出此时的内存信息；信号量 allo，Tracker 线程输出完内存信息后被释放，Allocator 进行下一次内存分配操作。

2. Allocator 线程

等待 Tracker 线程给出释放信号量 allo，之后即可进行内存操作，使用相应的 Win32 API 函数实现对内存区域的保留、提交、锁定、解锁、回收和释放等操作。

以“保留一个区域”为例，使用 VirtualAlloc 函数，并且第三个参数为 MEM_RESERVE，函数返回值为保留的内存区域的基址。内存分配完毕之后，释放 trac，提示 Tracker 获取并输出当前的内存信息。

```
if(WaitForSingleObject(allo,INFINITE) == WAIT_OBJECT_0)
{
    Mem_Addr =
VirtualAlloc(NULL,PAGE_SIZE,MEM_RESERVE,PAGE_READWRITE);
    if(Mem_Addr == NULL)
        GetLastError();
    else
    {
        cout<<"Memory State After RESERVE:"<<endl<<endl;
        ReleaseSemaphore(trac,1,NULL);
    }
}
```

其他内存操作实现基本类似，只不过使用的函数和参数不同，详见代码文件 VirtualMemory.cpp 文件。

3. Tracker 线程

每进行一次内存分配操作，Tracker 线程都会紧接着对内存状态进行查询和输出，主要由 VirtualQuery 函数实现，获取信息之后主要的任务是将这些内存状态信息转换为相应的字符串，输出到控制台窗口。

```
if(WaitForSingleObject(trac,INFINITE) == WAIT_OBJECT_0)
{
    VirtualQuery(Mem_Addr,&meminfo,sizeof(meminfo));   //查询内存信息

    cout<<" Allocation Base:"<<meminfo.AllocationBase<<endl; //获取指向内存区域基址的指针
    cout<<" Base Address:"<<meminfo.BaseAddress<<endl; //内存区域的基址
    cout<<" Region Size:"<<meminfo.RegionSize<<endl; //内存区域的大小(字节数)

    switch(meminfo.AllocationProtect)   //将内存分配的保护转为字符串
    {
    case PAGE_EXECUTE:allopro = "EXECUTE";break;
    case PAGE_EXECUTE_READ:allopro = "EXECUTE_READ";break;
    case PAGE_EXECUTE_READWRITE:allopro = "EXECUTE_READWRITE";break;
    case PAGE_EXECUTE_WRITECOPY:allopro = "EXECUTE_WRITECOPY";break;
    case PAGE_GUARD:allopro = "GUARD";break;
    case PAGE_NOACCESS:allopro = "NOACCESS";break;
    case PAGE_NOCACHE:allopro = "NOCACHE";break;
    case PAGE_READONLY:allopro = "READONLY";break;
    case PAGE_READWRITE:allopro = "READWRITE";break;
    case PAGE_WRITECOPY:allopro = "WRITECOPY";break;
    case PAGE_WRITECOMBINE:allopro = "WRITECOMBINE";break;
    default: allopro = "Cannot Access!";
}
  cout<<" Allocation Protect:"<<allopro<<endl; //输出内存分配的保护机制

  switch(meminfo.Protect) //内存区域对页访问的保护
  {
  case PAGE_EXECUTE:protect = "EXECUTE";break;
  case PAGE_EXECUTE_READ:protect = "EXECUTE_READ";break;
  case PAGE_EXECUTE_READWRITE:protect = "EXECUTE_READWRITE";break;
  case PAGE_EXECUTE_WRITECOPY:protect = "EXECUTE_WRITECOPY";break;
  case PAGE_GUARD:protect = "GUARD";break;
  case PAGE_NOACCESS:protect = "NOACCESS";break;
  case PAGE_NOCACHE:protect = "NOCACHE";break;
  case PAGE_READONLY:protect = "READONLY";break;
  case PAGE_READWRITE:protect = "READWRITE";break;
  case PAGE_WRITECOPY:protect = "WRITECOPY";break;
  case PAGE_WRITECOMBINE:protect = "WRITECOMBINE";break;
```

```
    default: protect = "Cannot Access!";
}
    cout<<" Protect:"<<protect<<endl;   //输出内存区域对页访问的保护

    switch(meminfo.State)   //内存区域中页的状态
    {
    case MEM_COMMIT: state = "MEMORY COMMIT";break;
    case MEM_FREE: state = "MEMORY FREE";break;
    case MEM_RESERVE: state = "MEMORY RESERVE";break;
    default: state = "Cannot Access!";
    }
    cout<<" State:"<<state<<endl;   //输出当前内存区域中页的状态

    switch(meminfo.Type) //内存区域中页的类型
    {
    case MEM_IMAGE: type = "MEMORY IMAGE";break;
    case MEM_MAPPED:type = "MEMORY MAPPED";break;
    case MEM_PRIVATE:type = "MEMORY PRIVATE";break;
    default:type = "Cannot Access!";
    }
    cout<<" Type:"<<type<<endl<<endl<<endl; //输出当前内存区域中的页的类型
```

4. 创建并执行线程

等待所有信息输出完毕，程序即可结束。

```
HANDLE allohandle = CreateThread(NULL,0,LPTHREAD_START_ROUTIN
(Allocator),NULL,0,NULL); //创建 Allocator 线程
HANDLE trachandle = CreateThread(NULL,0,LPTHREAD_START_ROUTINE
(Tracker),NULL,0,NULL);     //创建 Tracker 线程
WaitForSingleObject(trachandle,INFINITE);
```

五、 程序运行结果

程序运行结果如下所示：

System Information:

OEM ID:0
Number of Processors:4
Page Size:4096
Processor Type:PROCESSOR_AMD_X8664
Processor Level:6
Active Processor Mask:15
Minimum Application Address:00010000
Maximum Application Address:7FFEFFFF

Memory State After RESERVE:

Allocation Base:007F0000
Base Address:007F0000
Region Size:4096
Allocation Protect:READWRITE
Protect:Cannot Access!
State:MEMORY RESERVE
Type:MEMORY PRIVATE

Memory State After COMMIT:

Allocation Base:00B40000
Base Address:00B40000
Region Size:4096
Allocation Protect:READWRITE
Protect:READWRITE
State:MEMORY COMMIT
Type:MEMORY PRIVATE

Memory State After LOCK:

Allocation Base:00B40000
Base Address:00B40000
Region Size:4096
Allocation Protect:READWRITE
Protect:READWRITE
State:MEMORY COMMIT

```
        Type:MEMORY PRIVATE


Memory State After UNLOCK:

        Allocation Base:00B40000
        Base Address:00B40000
        Region Size:4096
        Allocation Protect:READWRITE
        Protect:READWRITE
        State:MEMORY COMMIT
        Type:MEMORY PRIVATE


Memory State After DECOMMIT:

        Allocation Base:00B40000
        Base Address:00B40000
        Region Size:4096
        Allocation Protect:READWRITE
        Protect:Cannot Access!
        State:MEMORY RESERVE
        Type:MEMORY PRIVATE


Memory State After RELEASE:

        Allocation Base:00000000
        Base Address:00B40000
        Region Size:65536
        Allocation Protect:Cannot Access!
        Protect:NOACCESS
        State:MEMORY FREE
        Type:Cannot Access!
```

根据上面的运行结果，可以看出，Windows8.1 操作系统的页大小为 4KB，即 4096 字节。先进行保留操作，虚拟内存区域状态为 MEM_RESERVE；提交操作之后，状态变为 MEM_COMMIT，锁定、解锁、回收操作之后的状态均为 MEM_COMMIT，释放操作之后虚拟内存区域的状态变为 MEM_FREE，其他状态信息如结果所示。

六、 思考题

1) 分析不同参数的设置对内存分配的结果影响。

解：

1. VirtualAlloc

功能：用于保留、提交或者改变页的状态。

函数说明：

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

VirtualAlloc 的输入共 4 个参数，各参数意义及使用方法如下：

1) lpAddress：表示内存分配时的起始地址，若令该参数为 NULL，则系统会自动决定分配内存的区域。
2) dwSize：区域的大小(字节数)，本实验中取为 Windows 8.1 操作系统的页大小，可以使用 GetSystemInfo 函数获取。
3) flAllocationTpye：内存分配的类型，主要包括以下值：
   a) MEM_COMMIT；
   b) MEM_RESERVE；
   c) MEM_RESET；
   d) MEM_RESET_UNDO；
   e) MEM_PHYSICAL；等。
4) flProtect：对要分配的页执行的保护机制，取值常被称为内存保护常量(Memory Protection Constants)，主要包括：
   a) PAGE_EXECUTE；
   b) PAGE_EXECUTE_READ；
   c) PAGE_EXECUTE_READWRITE；
   d) PAGE_EXECUTE_WRITECOPY；
   e) PAGE_NOACESS；
   f) PAGE_READONLY；
   g) PAGE_READWRITE；
   h) PAGE_WRITECOPY；
   i) PAGE_GUARD；
   j) PAGE_NOCACHE；
   k) PAGE_WRITECOMBINE；

VirtualAlloc 函数的返回值：

内存分配成功时，函数返回值为所分配内存的基址；内存分配失败时，返回值为 NULL，可以使用 GetLastError 函数获取相关信息。

2. VirtualLock

功能：用于锁定一块内存区域。

函数说明：

```
BOOL WINAPI VirtualLock(
   _In_ LPVOID lpAddress,
   _In_ SIZE_T dwSize
);
```

VirtualLock 函数包括两个参数：

1) lpAddress：表示要锁定的内存区域的基址；
2) dwSize：表示要锁定的内存区域的大小。

返回值：

若锁存成功，返回值非零；若锁存失败，返回值为零，可以使用 GetLastError 函数获取失败的信息。

3. VirtualUnlock

功能：用于解锁之前锁定的内存区域。

函数说明：

```
BOOL WINAPI VirtualUnlock(
   _In_ LPVOID lpAddress,
   _In_ SIZE_T dwSize
);
```

VitualUnlock 函数与 VirtualLock 函数相对应，参数包括：

1) lpAddress：表示要解锁的内存区域的基址；
2) dwSize：表示要解锁的内存区域的大小。

返回值：

解锁成功，返回值非零；若解锁失败，返回值为零，可以使用 GetLastError 函数获取失败的信息。

4. VirtualFree

功能：用于回收和释放内存区域。
函数说明：

```
BOOL WINAPI VirtualFree(
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD    dwFreeType
);
```

VirtualFree 的参数：

1) lpAddress：
指向要被释放的内存区域的基址。
2) dwSize：
要被释放的内存区域的大小(字节数)。注意：当 dwFreeType 参数为 MEM_RELEASE 时，该参数必须设置为 0。
3) dwFreeType：
内存操作的类型，取值可以是：
   a) MEM_DECOMMIT：回收之前提交的内存区域，回收之后，内存区域的状态应当变为 Reserve(保留)状态。
   b) MEM_RELEASE：释放特定的内存区域，释放之后，内存区域的状态应当变为 Free 状态。

2. 如果调换分配、回收、内存复位、加锁、解锁、提交、回收的次序，会有什么变化，并分析原因。

解：

在调换顺序时，为保证程序正常运行，必须注意以下条件，不然会出现错误。

a) 提交操作需要在保留操作之后。

如果提交操作在保留操作之前进行，那么提交时 VirtualAlloc 参数内的 lpAddress 参数没有给定一个保留区域的起始地址，因而无法执行提交操作，导致程序无法正常执行。

b) 释放操作需要在回收操作之后。

对内存区域的释放操作如果在回收之前，程序出现错误，无法正确执行。

c) 解锁操作需要在锁定操作之后。

解锁是针对锁定的区域进行的，如果没有使用 VirtualLock 函数锁定一个虚拟内存区域，VirtualUnlock 函数对未锁定区域的解锁会出错，但也不会对内存区域造成破坏。

3. 在 Linux 操作系统上没有提供系统调用来实现“以页为单位的虚拟内存分配方式”，如果用户希望实现这样的分配方式，可以怎样做？解释你的思路。

解：

Windows 系统下虚拟内存的机制主要依托于 Windows 对内存及磁盘空间的管理和系统调用。Linux 操作系统没有这样的系统调用，不过可以通过创建进程，对内存和磁盘空间进行管理，按照类似于 Windows 系统的机制，模拟出虚拟内存的页面、索引结构等必要元素，在进程中实现“以页为单位的”对虚拟内存的分配。

七、 实验总结及体会

本次实验中，解决问题的思路比较简单和清晰，关键问题主要集中在 Win32 API 函数的使用上，涉及了一系列有关虚拟内存操作的函数。在实验前，我首先从 MSDN 上查阅了涉及到的所有函数，对各函数的功能、参数限制、使用方法及注意事项进行了比较细致的阅读和总结。

实验中除了一些内存操作的函数，还涉及到两个结构体变量，之前没有接触过，分别是 SYSTEM_INFO 和 MEM_BASIC_INFORMATION。SYSTEM_INFO 是使用 GetSystemInfo 获得的系统信息，而 MEM_BAISC_INFORMATION 是使用 VirtualQuery 得到的内存的相关信息。

在了解了各函数和数据类型的基础之上，按照实验思路创建线程即可实现对虚拟内存的分配和跟踪。

这次实验最大的收获是基本了解了 Windows 对用户进程的虚拟内存空间的管理、分配方法，对 Windows 分配虚拟内存、改变内存状态和对物理内存和页面文件状态查询的相关函数有了基本的掌握，也加深了对操作系统页式存储管理的认识。

三个实验至此完全结束，本人选择的三次实验分别实现了进程间的通信、高级进程间通信问题以及 Windows 的虚拟内存分配问题，对操作系统的进程和线程机制有了基本的掌握，也对操作系统有了更加真实的认识，收获很大!

八、 附录

1. VirtualQuery
   功能：用于查询进程地址空间中内存地址的信息。
   函数说明：

```
SIZE_T WINAPI VirtualQuery(
  _In_opt_ LPCVOID                    lpAddress,
  _Out_    PMEMORY_BASIC_INFORMATION lpBuffer,
  _In_     SIZE_T                     dwLength
);
```

VirtualQuery 函数的参数：

1) lpAddress
   指向被查询的内存区域的基地址；
2) lpBuffer
   查询到的内存区域的基本信息结构体指针，相当于该函数的返回值，(MEMORY_BASIC_INFORMATION)；
3) dwLength
   lpBuffer 指针指向的参数大小(字节数)。

2. MEM_BASIC_INFORMATION

   MEM_BASIC_INFORMATION 结构体用于存储进程空间的内存信息。
结构体说明：

```
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;
```

1) BaseAddress：指向内存区域基址的指针。
2) AllocationBase：指向 VirtualAlloc 分配的内存区域基址的指针。
3) AllocationProtect：内存保护机制的选择，取值可以为内存保护常量(详见之前 VirtualAlloc 的 flProtect 参数)。
4) RegionSize：内存区域大小。
5) State：内存区域中页的状态，主要包括：

a) MEM_COMMIT；
b) MEM_FREE；
c) MEM_RESERVE；

6) Protect：内存区域中对页的访问保护，取值可以是 AllocationProtect 的取值集合中的一个。
7) Type：内存区域中页的类型，主要包括：
   a) MEM_IMAGE；
   b) MEM_MAPPED；
   c) MEM_PRIVATE。