

# 操作系统大作业实验报告

无 53  
2015011025  
秦嘉昊

## 实验 1：进程间同步/互斥问题

### 一. 实验目的

1. 通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和 P、V 操作的原理
2. 对 Windows 或 Linux 涉及的几种互斥、同步机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与互斥、同步有关的函数。

### 二. 实验题目

选择⑤银行柜员服务问题

银行有  $n$  个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个柜员空闲下来，就叫下一个号。

编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

实现要求：

1. 某个号码只能由一名顾客取得；
2. 不能有多于一个柜员叫同一个号；
3. 有顾客的时候，柜员才叫号；
4. 无柜员空闲的时候，顾客需要等待；
5. 无顾客的时候，柜员需要等待。

实验环境：

Windows10

VS2015

### 三. 设计思路&程序结构

题目要求顾客取号与柜台叫号通过互斥对象完成，故建立两个信号量分别作为取号与叫号的信号量。

此外，题目要求顾客与柜台间进行同步操作，为每个顾客分配一个信号量用于完成顾客与柜台间的同步。具体实现可参考项目代码，其中进行了详细的注释

题目整体完成过程如下：

首先从输入文件读取各个顾客的信息

而后创建时钟函数用于记录程序开始的时间（程序中时钟周期记录在 T 中）

接下来依次创建柜台实例并为每个柜台开启一个线程，该线程进行忙等待直至有顾客等待（waitlist 队列长度不为 0），则叫号并开始服务。

为每个柜台开启叫号线程后顾客开始进入，判定顾客进入的方法同样为忙等待，判定当前时间每个顾客是否应当进入，若到达其进入时间则为其开启一个取号线程。

当所有顾客服务结束后输出最终结果。

题目整体完成结构见 main.cpp 中的 `int main()` 及 `void start()`

模拟时钟计时操作如下：

定义了一个全局变量 time 用于记录当前时刻，clock 线程不断循环睡眠 Tms，每两次睡眠之间 time 自增一次，即完成了用 time 记录当前时刻的功能

模拟时钟过程见 main.cpp 中的 `void clock()`

模拟顾客取号操作过程如下：

定义了全局变量 M\_GET\_NUM 为用户取号信号量，全局变量 NUM 为当前号码。

此外定义了顾客等待队列 waitlist 用于记录当前在排队的顾客及其顺序。M\_CUSTOM 为顾客柜台同步信号量，用于保护服务过程中可能用到的资源，在用户实例初始化时进行 P 操作。当用户取号时，先对该信号量进行一次 P 操作，而后记录当前号码 NUM 为该顾客取得的号码，当前号码加一。将该顾客加入到排队的队列 waitlist 中。顾客取号操作完毕，对 M\_GET\_NUM 进行 V 操作

顾客取号过程见 main.cpp 中的 `void Get_num(customer *this_customer)`

模拟柜台叫号操作过程如下：

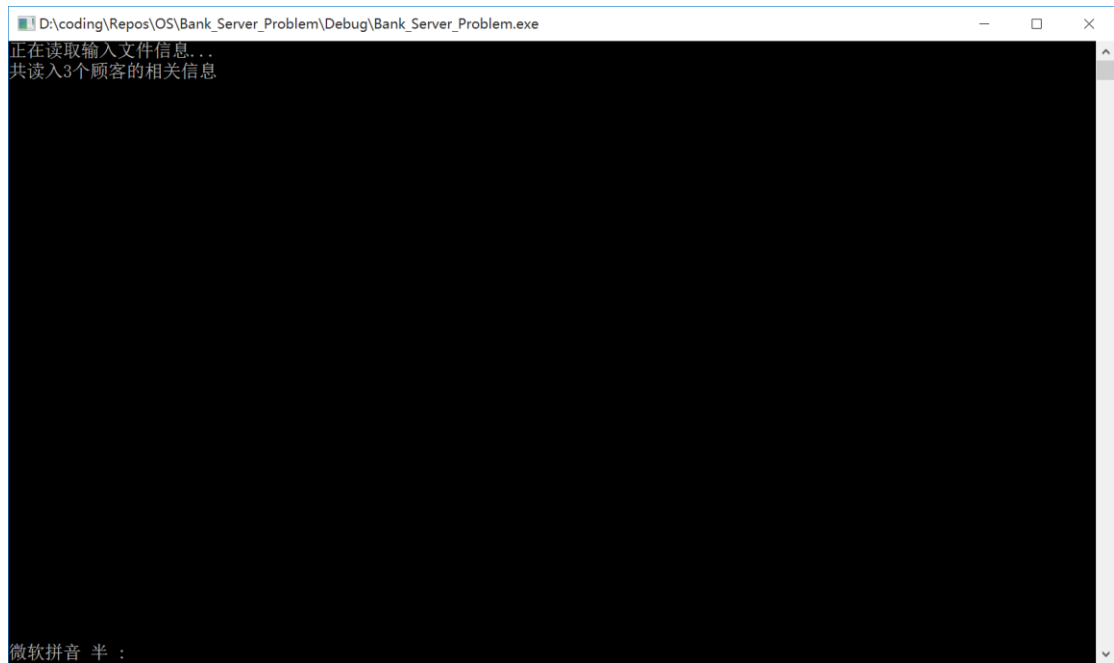
定义了全局变量 M\_CALL\_NUM 为柜台叫号信号量，全局变量 NUM\_DONE 为当前已完成服务的顾客数。当柜台叫号时，先对 M\_CALL\_NUM 进行 P 操作，而后判断当前等待队列 waitlist 是否为空，若不为空则直接对 M\_CALL\_NUM 进行 V 操作，继续进行忙等待，否则对当前等待队列 waitlist 中的队首取出，从队列中移除，叫号过程结束，对 M\_CALL\_NUM 进行 V 操作，而后对刚刚取出的其进行服务。开始服务时被服务的顾客记录当前时刻为开始服务时刻，当前柜台为服务柜台。服务过程通过 Sleep 函数模拟，服务结束后顾客记录当前时间为结束服务时间，并对 M\_CUSTOM 进行 V 操作，即服务完成顾客用于服务的相应资源可继续用于其他用途。柜台在服务完成后继续开始叫号。

柜台叫号过程见 main.cpp 中的 `void Call_num(server *this_server)`

模拟顾客与柜同步操作过程在前两过程中已有描述不再单独列出

#### 四. 程序运行情况

程序运行界面如下：



取柜台数  $N=3$ ，对于输入信息

1	1	1	10
2	2	5	2
3	3	6	3

得到输出结果每次均不同，截图几张为例：

第一次：

1	1	1	11	2
2	5	5	7	1
3	6	6	9	0

第二次：

1	1	1	11	2
2	5	5	7	0
3	6	6	9	1

第三次：

1	1	1	11	1
2	5	5	7	2
3	6	6	9	0

经验证其答案均正确。每次服务相应顾客的柜台有所差异是因为线程工作时最开始各柜台均为闲置状态，进入一个顾客不一定会被哪个柜台取走，因而各次输出不同恰恰是实验成功的证明。

## 五. 思考题解答

1. 柜员人数和顾客人数对结果分别有什么影响？

当柜台数量增加时由于同时叫号的线程较多，每个线程抽到顾客的概率也就下降，更加会导致每次输出结果均有所不同，同时每个顾客的平均等待时间也会相应地下降。

当顾客数量增加，就好像现实生活中柜台叫号后并不能立即开始服务，结束后也并未立即叫号一样，由于每个柜台实际叫号，服务时间中有损耗（如并非叫号之后立即开始服务）当顾客数量较多，损耗累计后可能使得结果与理想情况下计算的结果有差异

2. 实现互斥的方法有哪些?各自有什么特点?效率如何?

实现互斥的方法有临界区、互斥量、信号量三种。期中临界区只能用在单个进程中，互斥量可用在多个进程使用互斥资源时，信号量可用在多个进程使用限制数量的资源时。其效率为临界区优于互斥量优于信号量，但适用范围信号量大于互斥量大于临界区。

## 六. 体会&遇到的问题

通过本次实验我实际使用了 Windows 提供的相应 api，第一次编写了多线程程序。在程序编写的过程中我查阅了 MSDN 上的相关资料，也在 CSDN 上看到了很多关于信号量的使用以及多线程管理的实例，逐渐理解了相关 api 接口的应用。在程序一开始编写时我对 P 操作、V 操作应当何时进行还不甚了解，在 debug 的过程中我的 NUM 变量由于 V 操作的位置有问题而导致多个顾客领到了相同的号码，导致后续程序出现了问题，对其的修正让我切实地体会到了 PV 操作在实际编程中对于资源保护的具体作用。

同时，相比于单线程编程，多线程程序在 debug 时异常困难，加上资源量，互斥量等不像常规变量那样在 debug 时刻直接通过断点调试观察取值，本次实验让我的程序调试能力也有了较大的提升。

## 实验 2：高级进程间通信问题

### 一. 实验目的

1. 通过对进程间高级通信问题的编程实现，加深理解进程间高级通信的原理；
2. 对 Windows 或 Linux 涉及的几种高级进程间通信机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与高级进程间通信有关的函数。

### 二. 实验题目

选择②快速排序问题

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

- (1) 首先产生包含 1,000,000 个随机数（数据类型可选整型或者浮点型）的数据文件；
- (2) 每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于 1000 以后不再分割（控制产生的进程在 20 个左右）；
- (3) 线程（或进程）之间的通信可以选择下述机制之一进行：
  - 管道（无名管道或命名管道）
  - 消息队列
  - 共享内存
- (4) 通过适当的函数调用创建上述 IPC 对象，通过调用适当的函数调用实现数据的读出与写入；
- (5) 需要考虑线程（或进程）间的同步；
- (6) 线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

实验环境：

Windows10

VS2015

### 三. 设计思路&程序结构

本次实验使用共享内存实现线程间通信，考虑到快速排序过程中每次排序均对不同片段进行，因而可将待排序片段全体放入共享内存中，令多个线程同时完成排序。**具体实现可参考项目代码，其中进行了详细的注释**

**题目整体完成过程如下：**

首先产生包含 1,000,000 个随机整数的数据文件。

创建共享内存并将输入信息读入到共享内存中。

接下来的部分尝试使用了两种方法，接下来将依次叙述：

第一种方法：用一个资源量来表示当前的线程数，同时用一个队列来记录未进行过排序的片段的首位位置。首先将待排序片段全体加入队列中。之后不断忙等待判断是否有未排序的片段而后根据线程资源量判断是否可以加入线程，可以加入则加入一个线程而后开始排序。排序过程将判断取出的未排序片段长度而后直接进行排 (<1000) 或进行一次划分后将两侧的片段加入未排序片段队列中。当排序结束后输出排序结果。但实际运行过程中发现此种方法将大量时间浪费在了开辟线程上，以致最后实际运行时间甚至不如单线程进行排序，且此方法难以实现在各个线程结束后对线程的关闭。

第二种方法：相比于第一种方法，第二种方法排序过程类似，但开辟线程的方法为在最开始开辟 20 个线程，而后每个线程均进行忙等待，待排序片段队列不为空则从中取出一个片段（类似第一问中的银行柜台问题），通过该方法大大减少了开辟线程所浪费的时间，当排序结束后依次关闭所有线程并输出结果。

**题目整体完成结构见 main.cpp(方法一)及 main.cpp(方法二)中的 `int main()` 及 `void Start()`。接下来主要阐述方法二的具体实现，方法一具体实现与之类似**

**每个线程中进行如下：**

定义了队列 `unsorts` 用于记录当前尚未排序片段的首位位置，同时定义了一个信息量 `M_UNSORTS` 用于判断是否可以对待排序队列 `unsorts` 进行操作

首先在创建线程时读取共享内存。

而后开始忙等待，不断判断当前待排序队列是否为空，若其不为空则对 `M_UNSORTS` 进行 P 操作，读取 `unsorts` 的队首，将读取出的片段与指向共享内存的指针传递给排序函数

**线程执行过程见 main.cpp 中的 `void Thread()`**

**排序函数操作过程如下：**

定义了 `NUM_SORTED` 用于记录当已排序过的位置的数量，同时定义了一个信息量 `M_SORTED` 用于判断是否可以对 `NUM_SORTED` 进行操作

排序函数首先读取待排序片段的首位位置，而后对 `unsorts` 队列进行出栈操作，对 `M_UNSORTS` 进行 V 操作。接下来首先判断当前片段长度，若小于 1000 则直接对其进行快速排序，而后对 `M_SORTED` 进行 P 操作，更新 `NUM_SORTED`，对 `M_SORTED` 进行 V 操作；否则进行一次划分后将划分位置两侧的片段加入待排序队列，同样对 `M_SORTED` 及 `NUM_SORTED` 进行操作。

**顾客取号过程见 main.cpp 中的 `void Sort(unsort * target, int * pBuffer)`**

#### 四. 程序运行情况

程序运行界面如下：

方法一结果：



方法二结果：



直接使用快速排序方法耗时大约 1200ms 左右，可以看出使用方法一并不能对程序耗时产生提高，甚至降低了排序效率，这是由于每次开辟线程耗时较多而单个线程操作又较为简单耗时较低，因而结果十分糟糕。而使用方法二则将排序速度缩减了将近一倍，因而多线程确实起到了提高效率的效果。



## 五. 思考题解答

1. 你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由。

本次实验通过多个进程同时对同一序列进行操作，因而使用共享内存可以完成多个线程间的通信，而管道和消息队列则主要是单线程与单线程之间进行通信，用于此试验需要每个线程都与主线程进行通信才能实现，实现较麻烦且效率较低

2. 你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

可以实现，每个线程都与主线程进行通信，要进行排队前先与主线程进行通信获取当前序列信息，而后在排序结束后与主线程通信传回排序结果。

## 六. 体会&遇到的问题

通过本次实验加深了我对多线程通信的具体实现流程的理解，也让我在优化算法的过程中对多线程编程的效率问题有所体会。在具体实现共享内存的过程中，我查阅了 MSDN 上的相关资料，更加了解了 Windows 提供线程间通信 API 接口的使用方式，对共享内存的原理有了更深刻的了解。同时由于最开始使用方法一最终发现并未对运行时间产生优化，在算法优化的过程中我考量了多线程编程中各因素对结果的影响最终了解了设计算法中的漏洞并进行优化取得了良好的成果。

## 实验 3：存储管理问题

### 一. 实验目的

1. 通过对存储管理问题的编程实现，加深理解存储管理的原理；
2. 熟悉 Windows 或 Linux 中定义的与存储管理有关的函数。

### 二. 实验题目

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。

实验环境：

Windows10

VS2015

### 三. 设计思路&程序结构

本次实验要求开辟两线程实现模拟内存分配以及对相应线程的跟踪，使用两个信号量 M\_ALLOCATE 和 M\_TRACK 分别用于标记是否可以进行内存分配以及跟踪。

具体实现可参考项目代码，其中进行了详细的注释

题目整体完成过程如下：

定义了 M\_ALLOCATE 与 M\_TRACK 作为信号量判断当前能否进行 allocate 或 track，两信号量最开始皆置为 0 即不可分配或跟踪

首先开辟一个共享内存用于存储虚拟内存的指针

接下来分别开启分配进程 Allocator 和跟踪进程 Tracker

题目整体完成结构见 main.cpp 中的 `int main()`

分配线程进行如下：

首先打开共享内存，之后若执行内存分配则将相应地址存入共享内存中。

只要用户输入的指令不为 quit 则不断循环。而后对 M\_ALLOCATE 进行 P 操作，判断是否可以分配。若输入指令为 help 则提示目前可使用的指令，并直接对 M\_ALLOCATE 进行 V 操作，即可继续进行输入。若为对虚拟内存操作指令则执行相应操作并输出操作名称，而后对 M\_TRACK 进行 V 操作，即已进行完 allocate，可进行 track 输出相应信息。若既不为 help 或 quit，也不为操作指令则提示用户输入了错误的指令并直接对 M\_ALLOCATE 进行 V 操作，即可继续进行输入

分配线程执行过程见 main.cpp 中的 `void Allocator()`

跟踪线程进行如下：

首先打开共享内存，读取虚拟内存地址

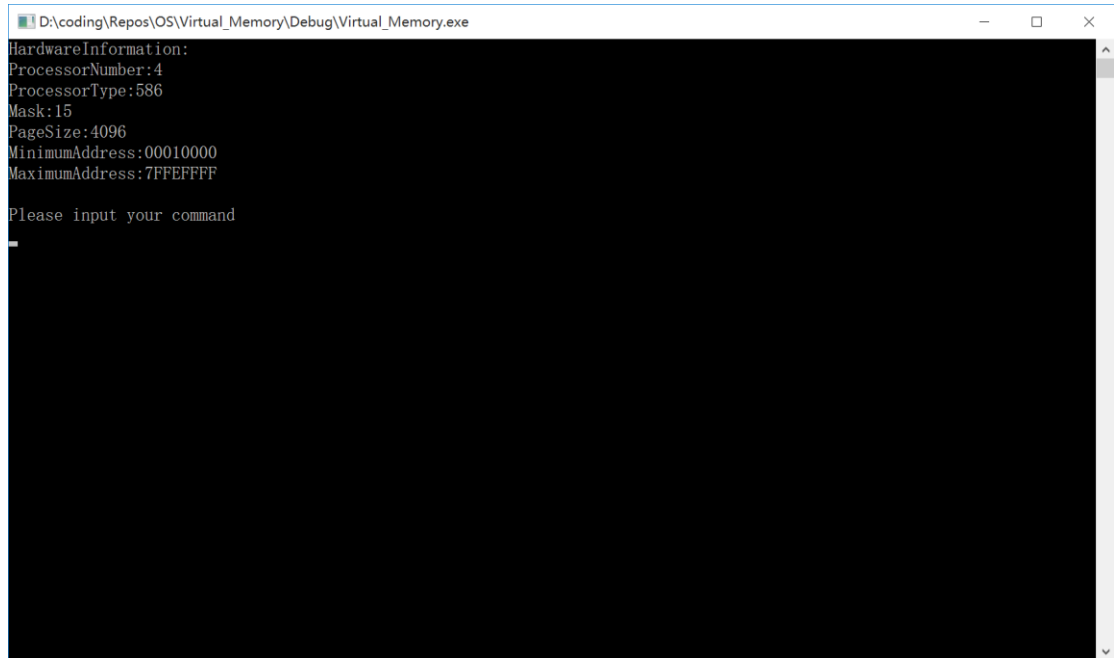
首先读取当前系统信息并输出，而后对 M\_ALLOCATE 进行 V 操作，提示分配进程当前可以进行 allocate。之后对 M\_TRACK 进行 P 操作，待可以进行跟踪后提取当前虚拟内存信息，调用 MEMORY\_BASIC\_INFORMATION2mem\_info 函数将其转换为可通过 cout 输出的形式，而后进行输出。输出结束后对 M\_ALLOCATE 进行 V 操作，提示分配进程可继续进行 allocate

分配线程执行过程见 main.cpp 中的 `void Tracker()`，转换函数见 `mem_info MEMORY_BASIC_INFORMATION2mem_info(MEMORY_BASIC_INFORMATION &lpBuffer)`

#### 四. 程序运行情况

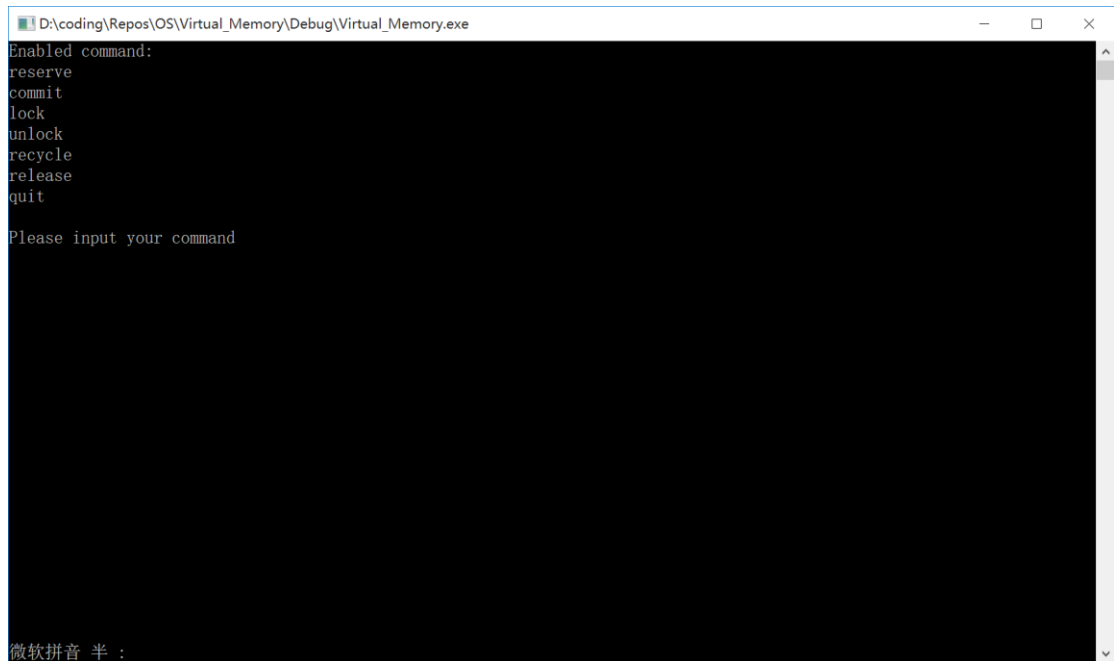
程序运行界面如下：

输出系统信息：



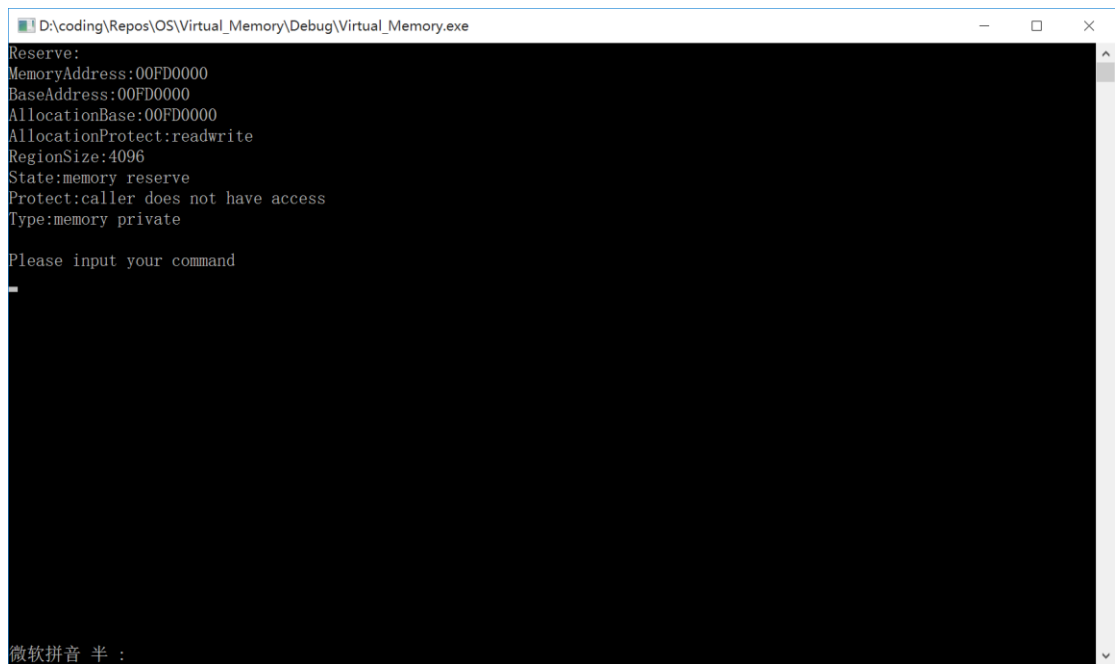
A screenshot of a Windows command window titled "D:\coding\Repos\OS\Virtual\_Memory\Debug\Virtual\_Memory.exe". The window displays the following text: "HardwareInformation:", "ProcessorNumber:4", "ProcessorType:586", "Mask:15", "PageSize:4096", "MinimumAddress:00010000", "MaximumAddress:7FFEFFFF", and "Please input your command". A cursor is visible on the line "Please input your command".

输入 help 结果：



A screenshot of a Windows command window titled "D:\coding\Repos\OS\Virtual\_Memory\Debug\Virtual\_Memory.exe". The window displays the following text: "Enabled command:", "reserve", "commit", "lock", "unlock", "recycle", "release", "quit", and "Please input your command". A cursor is visible on the line "Please input your command". At the bottom left of the window, there is a small text label "微软拼音 半：".

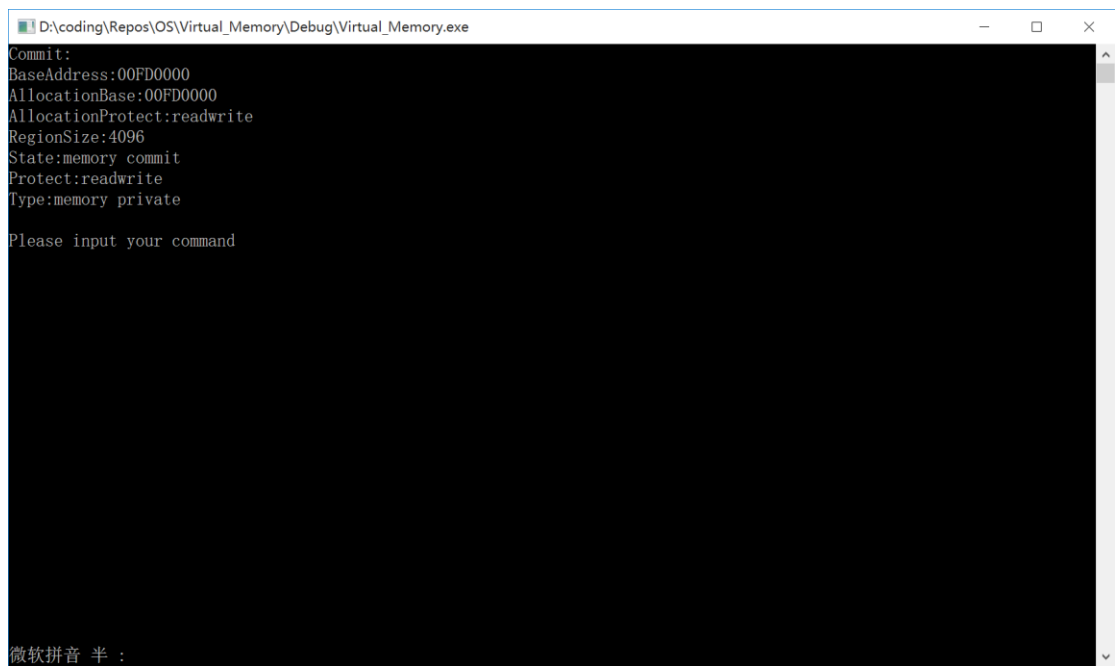
输入 reserve 结果：



```
Reserve:
MemoryAddress:00FD0000
BaseAddress:00FD0000
AllocationBase:00FD0000
AllocationProtect:readwrite
RegionSize:4096
State:memory reserve
Protect:caller does not have access
Type:memory private

Please input your command
_
```

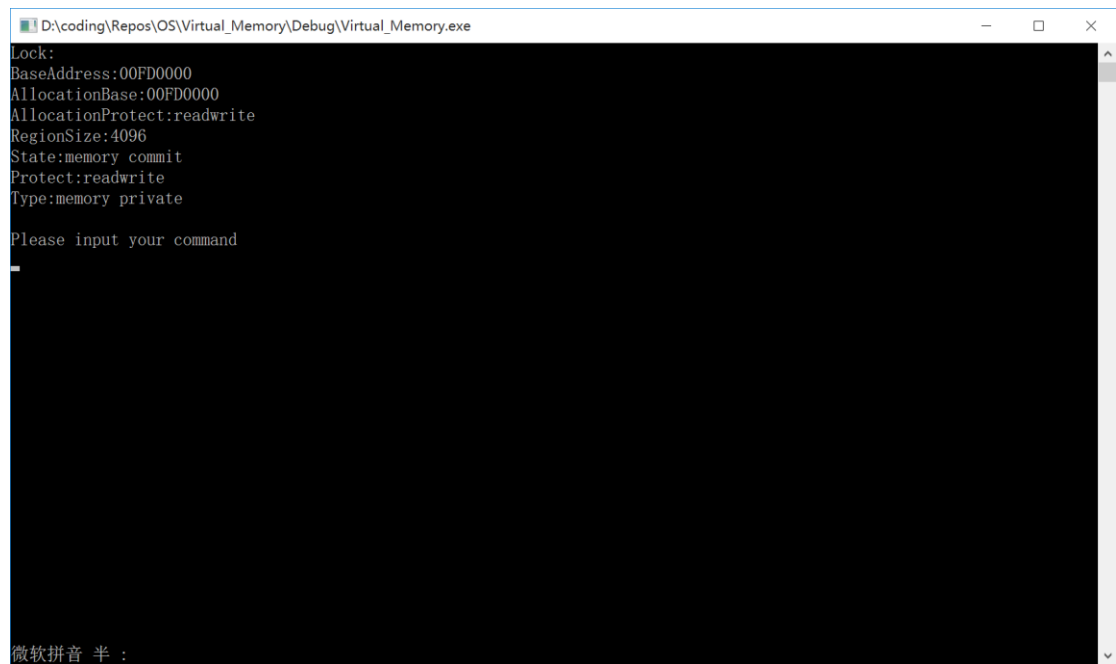
输入 commit 结果：



```
Commit:
BaseAddress:00FD0000
AllocationBase:00FD0000
AllocationProtect:readwrite
RegionSize:4096
State:memory commit
Protect:readwrite
Type:memory private

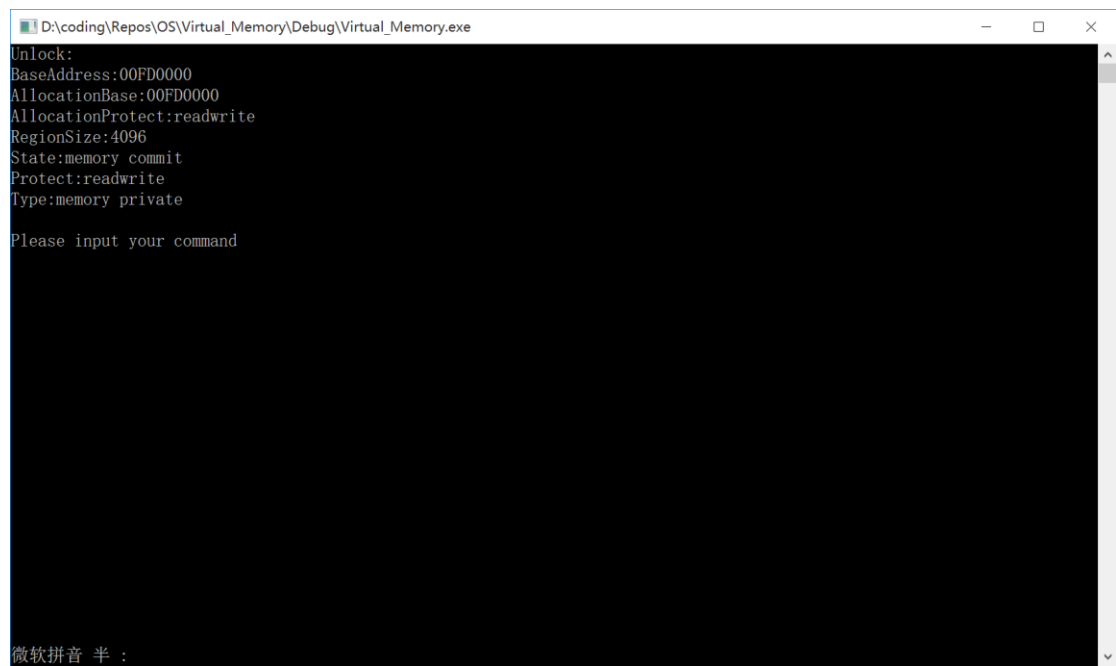
Please input your command
_
```

输入 lock 结果：



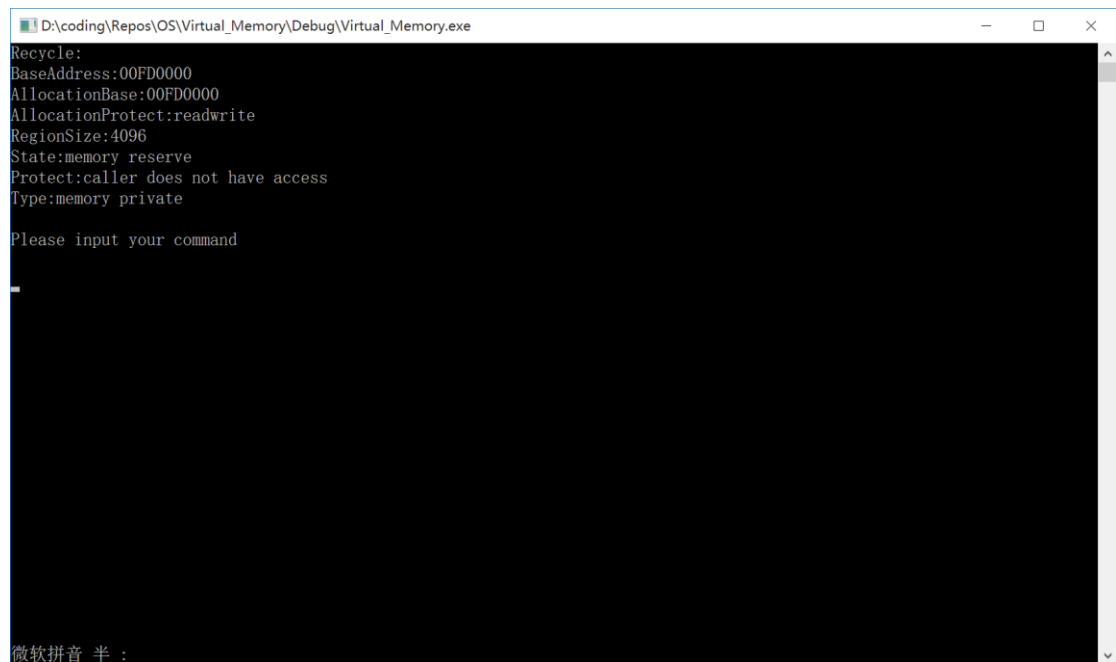
```
D:\coding\Repos\OS\Virtual_Memory\Debug\Virtual_Memory.exe
Lock:
BaseAddress:00FD0000
AllocationBase:00FD0000
AllocationProtect:readwrite
RegionSize:4096
State:memory commit
Protect:readwrite
Type:memory private
Please input your command
_
```

输入 unlock 结果:



```
D:\coding\Repos\OS\Virtual_Memory\Debug\Virtual_Memory.exe
Unlock:
BaseAddress:00FD0000
AllocationBase:00FD0000
AllocationProtect:readwrite
RegionSize:4096
State:memory commit
Protect:readwrite
Type:memory private
Please input your command
_
```

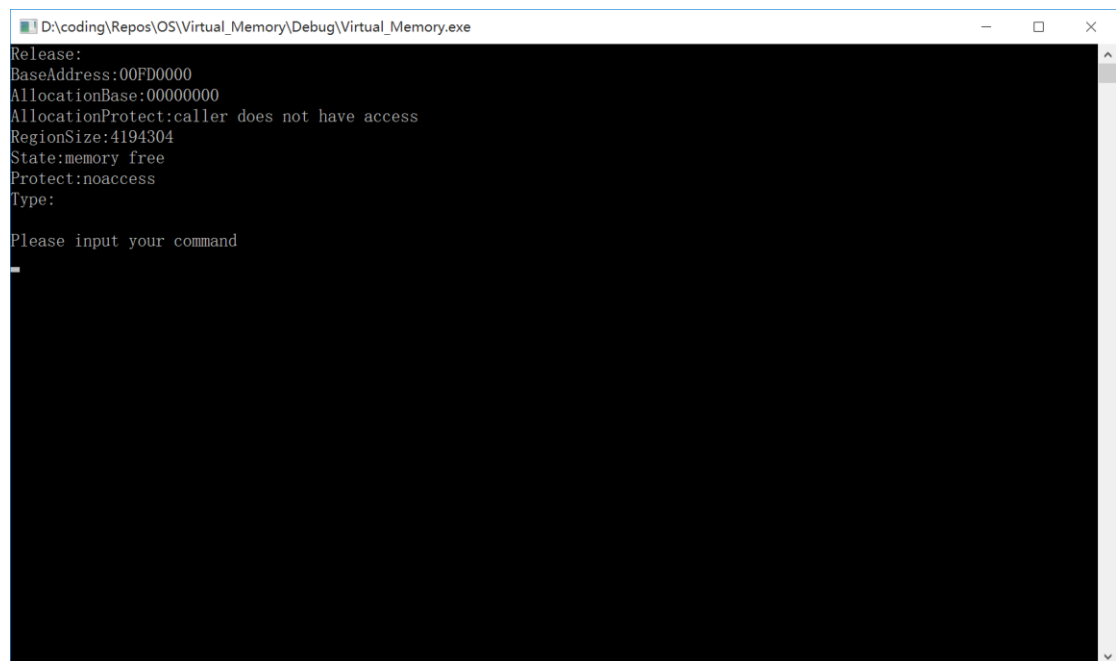
输入 recycle 结果：



```
D:\coding\Repos\OS\Virtual_Memory\Debug\Virtual_Memory.exe
Recycle:
BaseAddress:00FD0000
AllocationBase:00FD0000
AllocationProtect:readwrite
RegionSize:4096
State:memory reserve
Protect:caller does not have access
Type:memory private
Please input your command
_
```

微软拼音 半：

输入 release 结果：



```
D:\coding\Repos\OS\Virtual_Memory\Debug\Virtual_Memory.exe
Release:
BaseAddress:00FD0000
AllocationBase:00000000
AllocationProtect:caller does not have access
RegionSize:4194304
State:memory free
Protect:noaccess
Type:
Please input your command
_
```

输入 quit 后退出程序

可以看到程序运行情况与预期相符，实验成功

## 五. 思考题解答

1. 分析不同参数的设置对内存分配的结果影响。

查找 MSDN 可以看到

### Syntax



```
LPVOID VirtualAlloc(  
    LPVOID lpAddress,  
    DWORD dwSize,  
    DWORD flAllocationType,  
    DWORD flProtect  
);
```

下面对其四个参数解释分别为：

**lpAddress**：分配区域开始地址，输入 NULL 则由系统决定

**dwSize**：区域大小，lpAddress 为 NULL 则改值被扩至下一页边界

**flAllocationType**：区域分配类型，可为 MEM\_COMMIT、MEM\_RESERVE、MEM\_RESET、MEM\_TOP\_DOWN，分别表示提交、保留、复位以及撤销复位

**flProtect**：区域保护形式，同样有多种选择不在此一一列出

选取不同参数会对相应结果产生影响

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。

由于提交的页面必须为保留页面，若提交操作提前至保留操作之前则程序会出错。类似的，如果把释放操作提前至回收操作前面，则释放操作会出错，且可能减少可用物理内存总数；若把解锁操作提前至锁操作前，则解锁操作不会产生任何影响。此外，若分配操作放至其余操作前则前后不对同一区域进行操作，且之前可能无法进行操作（除了提交操作会对默认地址进行）



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。

考虑到页面虚拟内存的本质是磁盘空间，在 Linux 中可编写一个管理磁盘空间的进程来模拟页面的 size、索引结构，从而实现虚拟的内存分配的操作

## 六. 体会&遇到的问题

相比于前两个实验，本实验难度较低，大多数工作量在于相关函数使用方法及相关参数的搜索，通过本次实验让我更加熟悉了 MSDN 的使用，让我对 Windows 虚拟内存分配相关的 API 有了初步的认识，也对存储管理方面的相关知识有了具象化的，更深入的认识