**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2019 年 12 月 10 日

**实验名称：操作系统实验5 物理存储器与进程逻辑地址空间管理（2分）**

**实验目的：**以一个教学型操作系统EOS为例，深入理解物理存储器以及进程逻辑地址空间的管理方法；能对核心源代码进行分析；训练分析问题、解决问题以及自主学习能力，逐步达到能独立对小型操作系统的功能进行分析、设计和实现。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

通过查看EOS物理存储器的使用情况，练习物理内存的分配与回收，分析相关源代码；通过查看进程逻辑地址空间的使用情况，练习虚拟内存的分配与回收，分析相关源代码，完成在应用进程中分配虚拟页和释放虚拟页的功能。

**实验步骤：**

**1）EOS物理内存分配和回收的练习以及源代码分析**

（练习物理内存的分配和回收；分析相关源代码，阐述EOS中物理存储器的管理方法，包括数据结构和算法等；简要说明在本部分实验过程中完成的主要工作）

* 1. **EOS物理内存分配和回收的简述**

EOS 使用了分页内存管理方式，所以对物理存储器也是以页为单位来进行管理的。EOS 中使用页框号数据库（PFN Database）来管理所有物理页。PFN Database 其实是一个数组，数组中的元素是由一个结构体来定义的，数组的长度和物理存储器所包含的物理页数量是一致的。数组中的第 N 项描述了页框号为N 的物理页的状态，并且该项还指向了具有相同状态的另一个物理页的页框号，以构成链表。

用于描述数组中元素的结构体在文件 mm/i386/mi386.h 中定义如下：

typedef struct \_MMPFN

{

ULONG Unused : 9; // 未用

ULONG PageState : 3; // 物理页的状态

ULONG Next : 20; // 下一个物理页的页框号

}MMPFN, \*PMMPFN;

页的状态被定义为枚举类型，在文件 mm/mi.h 中定义如下：

typedef enum \_PAGE\_STATE {

ZEROED\_PAGE, // 零页

FREE\_PAGE, // 自由页

BUSY\_PAGE, // 占用页

} PAGE\_STATE;

* 1. **分配物理页**

1. 分配未经零初始化的自由页

相关函数：MiAllocateAnyPages(

IN ULONG\_PTR NumberOfPages,

OUT PULONG\_PTR PfnArray

)

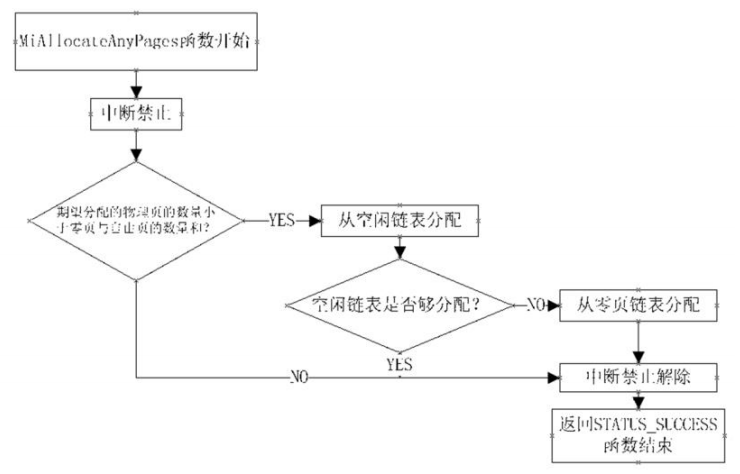
功能描述：分配物理页。首先从空闲页链表中分配，如果空闲页链表不足则再从零页链表分配。

参数：NumberOfPages -- 期望分配的物理页的数量。

PfnArray -- 指针，指向用于输出物理页框号的缓冲区。

返回值：如果成功则返回STATUS\_SUCCESS，否则返回STATUS\_NO\_MEMORY。

函数流程图：



源代码：

STATUS

MiAllocateAnyPages(

IN ULONG\_PTR NumberOfPages,

OUT PULONG\_PTR PfnArray

)

{

BOOL IntState;

ULONG\_PTR Pfn;

ULONG\_PTR i;

IntState = KeEnableInterrupts(FALSE);

if (NumberOfPages <= MiZeroedPageCount + MiFreePageCount) {

// 先从空闲链表中分配。

for (i = 0; i < NumberOfPages && MiFreePageCount > 0; i++) {

Pfn = MiFreePageListHead;

MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;

MiFreePageCount--;

MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;

PfnArray[i] = Pfn;

}

// 如果空闲链表不足则继续从零页链表分配。

for (; i < NumberOfPages; i++) {

Pfn = MiZeroedPageListHead;

MiZeroedPageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;

MiZeroedPageCount--;

MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;

PfnArray[i] = Pfn;

}

}

KeEnableInterrupts(IntState);

return STATUS\_SUCCESS;

}

1. 分配零页

相关函数：MiAllocateZeroedPages(

IN ULONG\_PTR NumberOfPages,

OUT PULONG\_PTR PfnArray

)

功能描述：首先从零页链表中分配，如果零页链表不足则再从空闲页。链表分配（会先

清零）。

参数：NumberOfPages -- 期望分配的物理页的数量。

PfnArray -- 指针，指向用于输出物理页框号的缓冲区。

返回值：如果成功则返回STATUS\_SUCCESS，否则返回STATUS\_NO\_MEMORY。

源代码：

STATUS

MiAllocateZeroedPages(

IN ULONG\_PTR NumberOfPages,

OUT PULONG\_PTR PfnArray

)

{

BOOL IntState;

ULONG\_PTR Pfn;

PVOID ZeroBuffer;

ULONG\_PTR i;

IntState = KeEnableInterrupts(FALSE);

if (NumberOfPages <= MiZeroedPageCount + MiFreePageCount) {

// 先从零页链表分配。

for (i = 0; i < NumberOfPages && MiZeroedPageCount > 0; i++) {

Pfn = MiZeroedPageListHead;

MiZeroedPageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;

MiZeroedPageCount--;

MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;

PfnArray[i] = Pfn;

}

// 如果零页链表不足，则继续从空闲页链表分配。

for (; i < NumberOfPages; i++) {

Pfn = MiFreePageListHead;

MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;

MiFreePageCount--;

MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;

// 将物理页映射到系统PTE区域进行清零。

ZeroBuffer = MiMapPageToSystemPte(Pfn);

memset(ZeroBuffer, 0, PAGE\_SIZE);

MiFreeSystemPte(ZeroBuffer);

PfnArray[i] = Pfn;

}

}

KeEnableInterrupts(IntState);

return STATUS\_SUCCESS;

}

1. 调试MiAllocateAnyPages函数的代码
2. 初始未完成MiAllocateAnyPages函数时，“物理内存”窗口查看当前物理内存的额使用情况



1. 本次分配的物理页的数量是1，分配的物理页的页框号为0x409；
2. 由MiAllocateAnyPages代码实现可知，在分配物理页时首先从空闲页链表中分配，如果空闲页链表不足则再从零页链表分配；
3. 减少空闲页数量

MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;

MiFreePageCount--;

1. 将刚刚分配的物理页由空闲状态修改为忙状态

MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;

1. 当MiAllocateAnyPages函数运行完毕后，刷新“物理内存”窗口。可以看到页框号为0x409的物理页相对于未执行该函数时的物理内存被分配出去，分配的物理页为1页，符合我们的函数执行结果。且空闲页数量由7134减少为7133，已使用页的数量由1042增加为1043，符合目标。



* 1. **回收物理页**

相关函数：MiFreePages(

IN ULONG\_PTR NumberOfPages,

IN PULONG\_PTR PfnArray

)

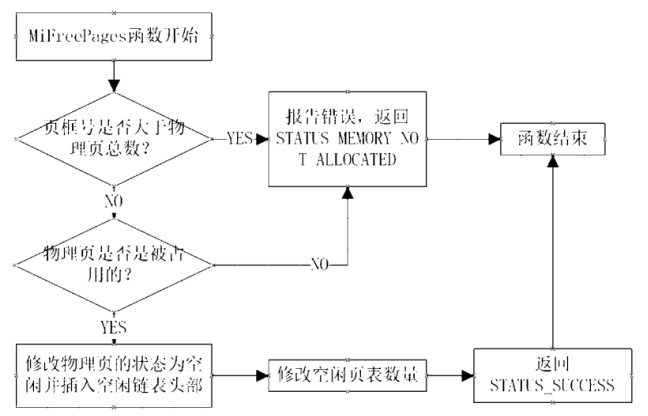
功能描述：释放物理页面。

参数：NumberOfPages -- 释放的页面数量。

PfnArray -- 指向页框号缓冲区的指针。

返回值：如果成功则返回STATUS\_SUCCESS。

函数流程图：



源代码：

STATUS

MiFreePages(

IN ULONG\_PTR NumberOfPages,

IN PULONG\_PTR PfnArray

)

{

ULONG\_PTR Pfn;

ULONG\_PTR i;

// 检查所有待回收的物理页，确保其为有效的已使用物理页。

for (i = 0; i < NumberOfPages; i++) {

Pfn = PfnArray[i];

if (Pfn >= MiTotalPageFrameCount || MiGetPfnDatabaseEntry(Pfn)->PageState != BUSY\_PAGE) {

ASSERT(FALSE);

return STATUS\_MEMORY\_NOT\_ALLOCATED;

}

}

// 修改这些物理页的状态为空闲，并将它们插入空闲页链表头部。

for (i = 0; i < NumberOfPages; i++) {

Pfn = PfnArray[i];

MiGetPfnDatabaseEntry(Pfn)->PageState = FREE\_PAGE;

MiGetPfnDatabaseEntry(Pfn)->Next = MiFreePageListHead;

MiFreePageListHead = Pfn;

}

MiFreePageCount += NumberOfPages;

return STATUS\_SUCCESS;

}

调试MiFreePages函数的代码：

* 1. 由函数参数可知，本次释放的物理页的数量为1（NumberOfPages=0x1），释放的物理页的页框号为0x409（PfnArray=0x409），即之前分配的物理页。
  2. 释放的物理页被放入了空闲页链表中，在for循环中设计实现，修改待回收的物理页状态为空闲，并将它们插入空闲页链表头部。

for (i = 0; i < NumberOfPages; i++) {

Pfn = PfnArray[i];

MiGetPfnDatabaseEntry(Pfn)->PageState = FREE\_PAGE;

MiGetPfnDatabaseEntry(Pfn)->Next = MiFreePageListHead;

MiFreePageListHead = Pfn;

}

* 1. 当MiFreePages函数运行完毕后，“物理内存”窗口如下图：



分析“物理内存”窗口可知成功回收页框号为0x409的一页物理页。

**2）EOS进程逻辑地址空间分配和回收的练习以及源代码分析**

（练习虚拟内存的分配和回收；分析相关源代码，阐述EOS中进程逻辑地址空间的管理方法，包括数据结构和算法等；给出在应用进程中分配虚拟页和释放虚拟页的实现方法简要描述、源代码、测试及结果等；简要说明在本部分实验过程中完成的主要工作）

**2.1 EOS虚拟内存的分配和回收简述**

物理内存是有限的，而且在 32 位地址线的情况下最大只能使用 4GB 物理内存，所以系统不可能为每个进程的 4G 虚拟地址空间全都映射物理页。只有在进程申请的时候，才为进程分配一定数量的物理页，并在进程指定的虚拟地址区域或由系统选择的一个可用虚拟地址区域进行映射。一般来说，进程虚拟地址空间对物理页的映射是稀疏的。为了记录进程虚拟地址空间中的哪些地址区域已经被使用，EOS 使用一个虚拟地址描述符（VAD，Virtual Address Descriptor）记录一段被使用的地址范围，并将所有 VAD 按照地址增序组成链表。

VAD 结构体和 VAD 链表结构体在文件 mm/mi.h 中定义如下：

typedef struct \_MMVAD{

ULONG\_PTR StartingVpn; // 被使用区域的开始虚页框号

ULONG\_PTR EndVpn; // 被使用区域的结束虚页框号

LIST\_ENTRY VadListEntry; // 链表项，用于将描述同一地址空间的所有VAD串成链表

}MMVAD, \*PMMVAD;

typedef struct \_MMVAD\_LIST{

ULONG\_PTR StartingVpn; // 记录的进程地址空间的开始虚页号

ULONG\_PTR EndVpn; // 记录的进程地址空间的结束虚页号

LIST\_ENTRY VadListHead; // VAD链表头

}MMVAD\_LIST, \*PMMVAD\_LIST;

**2.2 虚拟内存的分配**

相关函数：MmAllocateVirtualMemory

功能描述：在当前进程地址空间或系统地址空间中分配虚拟内存。

参数：

BaseAddress -- 作为输入时，输入期望保留或者提交的地址区域的起始地址；作为输出时，输出实际保留者提交的地址区域的起始地址。如果输入非 NULL，则输出值为输入值向下对齐到页面边界，否则由系统决定具体位置（仍然对齐到页面边界）。

RegionSize -- 作为输入时，输入期望保留或者提交的内存区域的大小；作为输出时，输出实际保留或者提交的内存区域的大小，为页面大小的整数倍。

AllocationType -- 分配类型，可以取值：

MEM\_RESERVE，仅在进程地址空间中保留一段虚拟地址区域，以备使用；

MEM\_COMMIT，仅为已保留的区域提交内存（为虚拟地址映射物理内存）。

以上两个值可以同时指定（MEM\_RESERVE | MEM\_COMMIT），在保留地址区域的同时为整个保留区域提交物理内存。

SystemVirtual -- 是否在系统地址空间分配虚拟内存。

返回值：如果成功则返回 STATUS\_SUCCESS，否则表示失败。

源代码：

STATUS

MmAllocateVirtualMemory(

IN OUT PVOID \*BaseAddress,

IN OUT PSIZE\_T RegionSize,

IN ULONG AllocationType,

IN BOOL SystemVirtual

)

{

STATUS Status;

BOOL IntState;

PMMPAS Pas;

PMMVAD Vad;

ULONG\_PTR StartingVpn;

ULONG\_PTR EndVpn;

ASSERT(BaseAddress != NULL && RegionSize != NULL);

// 确保参数有效（地址范围没有溢出）。

if (0 == \*RegionSize || \*BaseAddress + \*RegionSize - 1 < \*BaseAddress) {

return STATUS\_INVALID\_PARAMETER;

}

// 只能在MEM\_RESERVE、MEM\_COMMIT中选择，至少选择一个，可组合选择。

if ((AllocationType & (MEM\_RESERVE | MEM\_COMMIT)) == 0 ||

(AllocationType & ~(MEM\_RESERVE | MEM\_COMMIT)) != 0) {

return STATUS\_INVALID\_PARAMETER;

}

IntState = KeEnableInterrupts(FALSE);

if (SystemVirtual) {

Pas = &MiSystemPas;

} else {

Pas = MiCurrentPas;

}

do {

if ((AllocationType & MEM\_RESERVE) != 0) {

// 在地址空间中保留一段地址区域，保留的单位为页。

Status = MiReserveAddressRegion( &Pas->VadList,

\*BaseAddress,

\*RegionSize,

&Vad );

if (!EOS\_SUCCESS(Status)) {

break;

}

// 记录实际保留的起始、结束虚页框号。

StartingVpn = Vad->StartingVpn;

EndVpn = Vad->EndVpn;

} else {

// 查询MEM\_COMMIT操作地址区域是否为已保留地址区域，如不是则返回失败。

Status = MiFindReservedAddressRegion( &Pas->VadList,

\*BaseAddress,

\*RegionSize,

&Vad );

if (!EOS\_SUCCESS(Status)) {

break;

}

// MEM\_COMMIT操作的单位为页。

StartingVpn = MI\_VA\_TO\_VPN(\*BaseAddress);

EndVpn = MI\_VA\_TO\_VPN(\*BaseAddress + \*RegionSize - 1);

}

if ((AllocationType & MEM\_COMMIT) != 0) {

// 执行MEM\_COMMIT操作。

Status = MiCommitPages(StartingVpn, EndVpn);

if (!EOS\_SUCCESS(Status)) {

ASSERT(STATUS\_NO\_MEMORY == Status);

// 如果前面已执行MEM\_RESERVE则回滚MEM\_RESERVE。

if ((AllocationType & MEM\_RESERVE) != 0) {

MiFreeAddressRegion(&Pas->VadList, Vad);

}

break;

}

}

// 设置返回值。

\*BaseAddress = MI\_VPN\_TO\_VA(StartingVpn);

\*RegionSize = (EndVpn - StartingVpn + 1) << PAGE\_SHIFT;

Status = STATUS\_SUCCESS;

} while (0);

KeEnableInterrupts(IntState);

return Status;

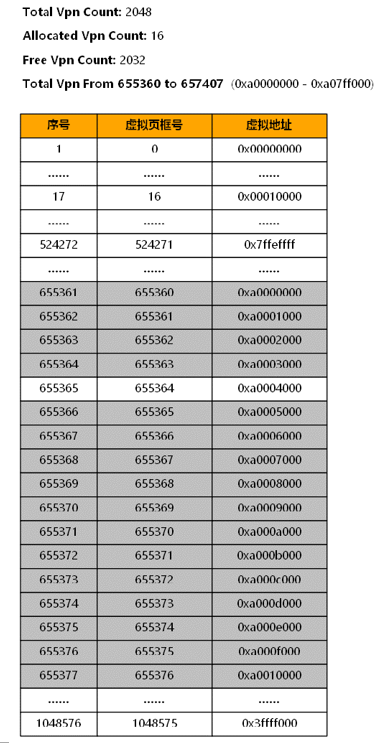
}

调试MmAllocateVirtualMemory函数的代码：

1. 初始未完成MmAllocateVirtualMemory函数时，“虚拟地址描述符”窗口如下图所示：



1. 在EOS控制台输入命令“vm 1”，vm开始执行后，会调用MmAllocateVirtualMemory函数，此时查看传入的参数BaseAddress即期望保留的地址区域的起始地址为0xa0003000，参数RegionSize即期望提交的地址区域的大小为1。第三个参数AllocationType可以设定分配虚拟页的同时为虚拟页映射实际的物理页。第四个参数SystemVirtual决定了分配的虚拟页是在系统地址空间的高2G分配的。
2. 在该函数中调用了MiReserveAddressRegion函数实现在地址空间中保留一段地址区域，MiReserveAddressRegion函数在vadlist.c第51行定义，阅读该函数实现代码，当传递的参数BaseAddress==NULL时，通过一个for循环从地址空间的起始端向后查找第一个满足申请大小的未保留区域，然后从系统内存池中分配一个虚拟地址描述符结构体，将虚拟地址描述符插入已保留区域链表中。
3. MiReserveAddressRegion函数返回MmAllocateVirtualMemory函数后，记录实际保留的起始、结束虚页框号。
4. 执行MEM\_COMMIT操作，设置返回值。此时查看“虚拟地址描述符”窗口显示如下图所示：



将该“虚拟地址描述符”窗口可以看到，序号为655364的虚拟页被分配出去。

**2.3 虚拟内存的回收**

相关函数：MmFreeVirtualMemory

功能描述：在当前进程地址空间或系统地址空间中释放虚拟内存。

参数：

BaseAddress -- 作输入时，输入期望释放的区域的起始地址，如果参数 FreeType 的值为

MEM\_RELEASE，则 BaseAddress 的值必须是用 MmAllocateVirtualMemory 保留区域时的返回值；作输出时，输出实际释放的地址区域的起始地址，由输入值向下对齐到页面边界计算得到。

RegionSize -- 作输入时如果参数 FreeType 的值为 MEM\_RELEASE 则必须为 0 否则输入期望 MEM\_DECOMMIT 的内存大小；作输出时，输出实际释放的区域大小。

FreeType -- 释放操作类型，其值只能是如下之一：

MEM\_RELEASE，释放整个已保留的区域，已提交内存同时被释放；

MEM\_DECOMMIT，对保留区域内部分已提交的区域进行反提交操作。

SystemVirtual -- 是否在系统地址空间释放虚拟内存。

返回值：如果成功则返回 STATUS\_SUCCESS。

源代码：

STATUS

MmFreeVirtualMemory(

IN OUT PVOID \*BaseAddress,

IN OUT PSIZE\_T RegionSize,

IN ULONG FreeType,

IN BOOL SystemVirtual

)

{

STATUS Status;

BOOL IntState;

PMMPAS Pas;

PMMVAD Vad;

ULONG\_PTR StartingVpn;

ULONG\_PTR EndVpn;

// \*BaseAddress必须是有效地址。

if (NULL == \*BaseAddress) {

return STATUS\_INVALID\_ADDRESS;

}

// 保证地址范围没有溢出（\*RegionSize可以为0）。

if (\*RegionSize > 0 && \*BaseAddress + \*RegionSize - 1 < \*BaseAddress) {

return STATUS\_INVALID\_PARAMETER;

}

// 只能在MEM\_DECOMMIT、MEM\_RELEASE中选择之一。

if ((FreeType & ~(MEM\_DECOMMIT | MEM\_RELEASE)) != 0 ||

((FreeType & (MEM\_DECOMMIT | MEM\_RELEASE)) == 0) ||

((FreeType & (MEM\_DECOMMIT | MEM\_RELEASE)) == (MEM\_DECOMMIT | MEM\_RELEASE))) {

return STATUS\_INVALID\_PARAMETER;

}

// 如果指定了MEM\_RELEASE则\*RegionSize必须为0。

if ((FreeType & MEM\_RELEASE) != 0 && \*RegionSize != 0) {

return STATUS\_INVALID\_PARAMETER;

}

IntState = KeEnableInterrupts(FALSE);

if (SystemVirtual) {

Pas = &MiSystemPas;

} else {

Pas = MiCurrentPas;

}

do {

// 查找已保留地址区域，如果目标区域非已保留区域则返回失败。

Status = MiFindReservedAddressRegion( &Pas->VadList,

\*BaseAddress,

\*RegionSize,

&Vad );

if (!EOS\_SUCCESS(Status)) {

break;

}

// 记录整个已保留区域的起止页号。

StartingVpn = Vad->StartingVpn;

EndVpn = Vad->EndVpn;

if ((FreeType & MEM\_RELEASE) != 0) {

// 执行MEM\_RELEASE操作时，\*BaseAddress必须是整个已保留区域的基址。

if (\*BaseAddress != MI\_VPN\_TO\_VA(StartingVpn)) {

Status = STATUS\_FREE\_VM\_NOT\_AT\_BASE;

break;

}

// 释放已保留地址区域。

MiFreeAddressRegion(&Pas->VadList, Vad);

} else {

// 当\*BaseAddress等于保留区域的起始地址且\*RegionSize为0时，可对整个

// 保留区域DECOMMIT，其它情况下\*RegionSize不能为0。

if(0 == \*RegionSize) {

if(\*BaseAddress != MI\_VPN\_TO\_VA(StartingVpn)) {

Status = STATUS\_FREE\_VM\_NOT\_AT\_BASE;

break;

}

} else {

// MEM\_DECOMMIT操作的单位为页。

StartingVpn = MI\_VA\_TO\_VPN(\*BaseAddress);

EndVpn = MI\_VA\_TO\_VPN(\*BaseAddress + \*RegionSize - 1);

}

}

// 执行MEM\_DECOMMIT操作。

Status = MiDecommitPages(StartingVpn, EndVpn);

ASSERT(EOS\_SUCCESS(Status));

// 设置返回值。

\*BaseAddress = MI\_VPN\_TO\_VA(StartingVpn);

\*RegionSize = (EndVpn - StartingVpn + 1) << PAGE\_SHIFT;

Status = STATUS\_SUCCESS;

} while (0);

KeEnableInterrupts(IntState);

return Status;

}

调试MmFreeVirtualMemory函数的代码：

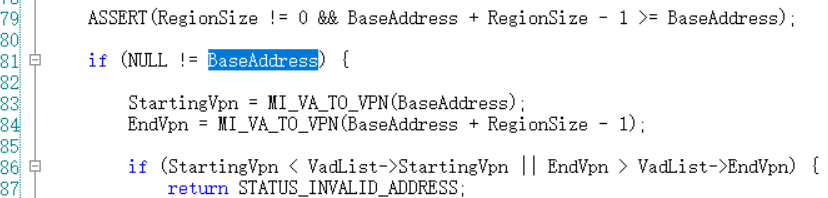
1. 查看传入的参数BaseAddress即期望保留的地址区域的起始地址为0xa0003000，可以知道是之前分配的虚拟页；
2. 在该函数中，调用了MiFindReservedAddressRegion函数，这个函数的作用是查找已保留地址区域，如果目标区域非已保留区域则返回失败。该函数的实现在vadlist.c中可以看到。调用MiDecommitPages函数用于释放映射在连续虚拟页框上的物理页框。
3. 待MmFreeVirtualMemory函数执行完毕后，在“虚拟地址描述符”窗口可以查看系统进程的虚拟地址描述符，可以发现释放的虚拟页就是之前分配的那一页。

**2.4 在应用进程中分配虚拟页和释放虚拟页**

* 实现方法简要描述

首先在 EOS 应用程序中调用VirtualAlloc函数（定义在api/eosapi.c文件的第48行），分配一个整型变量所需的空间，注意参数需要使用MEM\_RESERVE 标志和 MEM\_COMMIT 标志，并使用一个整型变量的指针指向这个空间，输出原始整型变量的值，修改整型变量的值后再输出，调用Sleep函数休眠10s，再调用 VirtualFree 函数释放之前分配的整形变量的空间。后进入死循环，这样应用程序就不会结束。

其中 VirtualAlloc 函数调用时第一个参数为 0 的原因： 在 MiReserveAddressRegion 函数中，判断保留虚拟地址区域时期望保留的地址区域的起始地址为 0，相关代码如下：



* 源代码

int main(int argc, char\* argv[])

{

// 在应用程序进程中分配虚拟页和释放虚拟页

INT \*p;

if(p=VirtualAlloc(0,sizeof(int),MEM\_RESERVE|MEM\_COMMIT)){

printf("Allocate %d bytes virtual memory at 0x%x.\n\n",sizeof(int),p);

printf("Virtual memory original value:0x%x\n",\*p);

\*p=0xFFFFFFFF;

printf("virtual memory new value:0x%x\n\n",\*p);

printf("Wait for 10 second\n\n");

Sleep(10000); //调用API函数Sleep，等待10秒钟

if(VirtualFree(p,0,MEM\_RELEASE))

printf("Release virtual memory success!\n\n");

else

printf("Release error\n");

printf("Endless loop!\n");

for(;;){

}

}

else{

printf("Error\n");

return -1;//若不能成功分配，打印出erroe，返回-1

}

printf("Hello world!\n");

return 0;

}

* 测试及结果



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS物理存储器管理与进程逻辑地址空间管理方法的特点、不足及改进意见；结合EOS对物理存储器与进程逻辑地址空间管理相关问题提出自己的思考；其他需要说明的问题）

1. EOS物理存储器管理与进程逻辑地址空间管理方法的特点

EOS使用了分页内存管理方式，使用页框号数据库（PFN Database）来管理所有物理页，离散分配的内存管理方式使得外部碎片减少，内存利用率大大增加，虚拟页式管理的方法也使内存空间“增大”。

2、思考题

1）在本实验 3.3 中，如果分配了物理页后，没有回收，会对EOS操作系统造成什么样的影响？目前 EOS 操作系统内核函数 MiAllocateAnyPages 能处理所有物理页被分配完毕的情况吗？例如在没有可分配的物理页的情况下调用该内核函数，是否会返回失败？如果内核函数 MiAllocateAnyPages 还不能处理这种极端情况，尝试修改代码解决这个问题。

答：如果分配了物理页后，没有回收，将会使可分配自由页和零页越来越少，最终导致内存溢出，系统无法运行。目前 EOS 操作系统内核函数 MiAllocateAnyPages 不能处理所有物理页被分配完毕的情况，在没有可分配的物理页的情况下，不会分配物理页，但仍然会返回成功，添加以下代码即可：

else {

return STATUS\_MEMORY\_NOT\_ALLOCATED； }

2）在本实验 3.3 中，在分配物理页时是调用的内核函数 MiAllocateAnyPages，该函数会优先分配空闲页，尝试修改代码，调用内核函数 MiAllocateZeroedPages 优先分配零页，并调试分配零页的情况。尝试从性能的角度分析内核函数 MiAllocateAnyPages 和MiAllocateZeroedPages。尝试从安全性的角度分析分配零页的必要性。

答：想要实现不优先分配空闲页，将MiAllocateAnyPages(1,PfnArray);修改为 MiAllocateZeroedPages(1,PfnArray);即可。系统启动时，所有空闲物理页都是未初始化的，此时零页链表为空，MiAllocateAnyPages函数可以直接从自由页链表分配，而 MiAllocateZeroedPages 函数会对从自由页链表中分配的每一页进行零初始化，确保所有分配页都是被零初始化的，再进行分配，因此MiAllocateZeroedPages 函数效率较低。但因为 MiAllocateZeroedPages 函数对自由页进行了初始化，减小了出错的可能性，从而安全性较高。

3）观察本实验 3.4 中使用“vm”命令输出的系统进程的虚拟地址描述符，可以看到在2号描述符和3号描述符之间有两个虚拟页的空隙，尝试结合虚拟页的分配和释放说明产生这个空隙的原因。

答：产生的空隙是由于有虚拟页被释放造成的，在EOS启动时有一个初始化线程在初始化完毕后就退出了，线程的堆栈所占用的虚拟页就被释放了。

4）在本实验 3.5 中，调用 MmAllocateVirtualMemory 函数分配虚拟页时只使用了MEM\_RESERVE 标志，没有使用 MEM\_COMMIT 标志，尝试说明这两个标志的区别。

答：使用 MEM\_RESERVE 标志分配虚拟页时，没有为其映射实际的物理页。使用 MEM\_COMMIT 表示分配虚拟页时，会为其映射实际的物理页。

3、总结

通过学习EOS中对物理存储器与进程逻辑地址空间的管理，调试EOS操作系统中物理存储空间和虚拟存储空间的分配和回收函数，对操作系统的内存管理机制和实现有了进一步的认识。同时，自己设计实现应用程序进程中分配虚拟页和释放虚拟页的程序，通过实践，对每个函数的含义及其参数的选择有了更深的理解。