|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统设计与实践 | | | 成 绩 |  | 教师签名 |  |
| 实验名称 | 分页机制 | | | 实验序号 | 3 | 实验日期 |  |
| 姓 名 | 张子航 | 学 号 | 2021302181026 | | | | 组长 |
| 姓 名 | 辜汝曦 | 学 号 | 2021302141194 | | | | 组员 |
| 姓 名 | 杨馨悦 | 学 号 | 2021302181212 | | | | 组员 |
| 姓 名 | 赵敏 | 学 号 | 2021302181215 | | | | 组员 |

**《操作系统设计与实践》实验报告**

# 实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容等）

## 实验目的

掌握内存分页机制

## 参考资料

* 对应参考阅读章节：第三章3.3节
* 对应参考代码为
  + 基本分页机制 /f/pmtest6.asm
  + 读取内存，并合理填充 /g/pmtest7.asm
  + 体会分页 /h/pmtest8.asm

## 实验要求

* 认真阅读章节资料，掌握什么是分页机制
* 调试代码，掌握分页机制基本方法与思路

代码3.22中，212行---237行，设置断点调试这几个循环，分析究竟在这里做了什么？（结合PPT第7页）

* 掌握PDE，PTE的计算方法

动手画一画这个映射图

为什么代码3.22里面，PDE初始化添加了一个PageTblBase(Line 212)，而PTE初始化时候没有类似的基地址呢（Line224）？

* 掌握熟悉如何获取当前系统内存布局的方法
* 内存地址映射关系的切换，并画出流程图
* 依据实验的代码

（1）自定义一个函数，给定一个虚拟地址，能够返回该地址从虚拟地址到物理地址的计算过程，如果该地址不存在，则返回一个错误提示。

（2）（选做）完善分页管理功能，补充alloc\_pages, free\_pages两个函数功能，实现内存的分配与回收。前者传入参数为分配的页数，后者传入的参数为释放内存的线性地址和页数。

# 实验环境及实验步骤

（列出本次实验所使用的软件、工具；简要概括实验步骤）

## 实验环境

虚拟机工具：VMWare Workstation 16

虚拟机版本：Ubuntu 14.04.6(内存4GB，硬盘40GB，双核处理器)

开发与调试工具：bochs 2.6.8

## 实验步骤

1.认真阅读章节资料，掌握什么是分页机制

2.调试代码，掌握分页机制基本方法与思路

3.掌握PDE，PTE的计算方法

4.熟悉如何获取当前系统内存布局的方法

5.掌握内存地址映射关系的切换，并画出流程图

6.依据实验的代码实现相应功能

# 实验过程分析

（详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。）

## 本次实验内容

### 认真阅读章节资料，掌握什么是分页机制

分页机制就像一个函数，物理地址=f（线性地址），用两级页表，每个页表4kb有1024项，每项32bit，第一级每项指向页表，第二级每项指向物理页

### 调试代码，掌握分页机制基本方法与思路

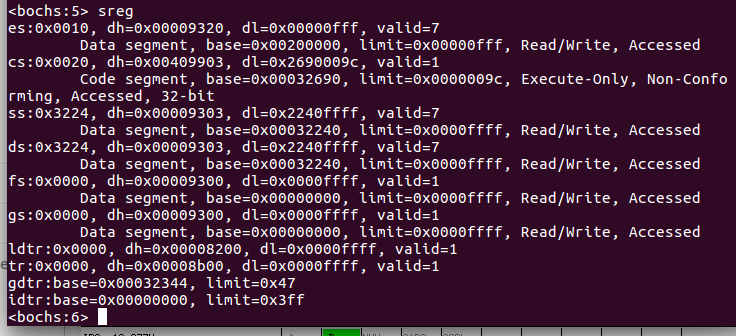
**stosd:**将eax的内容（4字节）存储到es:edi指向的内存单元中，同时edi的值根据方向标志的值(DF==0)增加或者减少（DF==1）相应的还有stosb（1字节）,stosw（2字节）

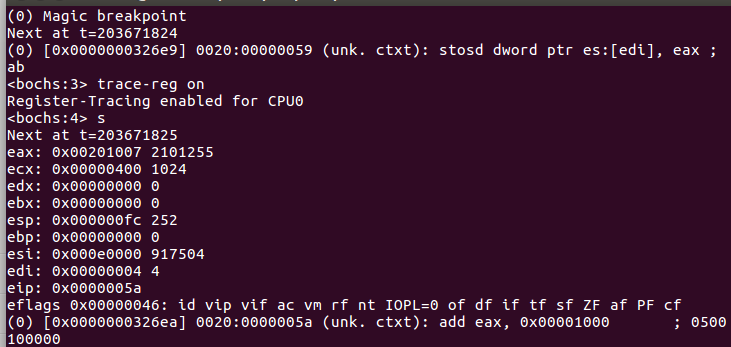
寻址方式：如果是实模式，就 es \* 16 + edi 如果是保护模式，则 es是段选择符，edi是段内偏移量。 根据该选择符es 去 GDT或者LDT中查找相应的 段描述符，再根据得到的段描述符找到对应段的物理基地址， 然后加上edi，算出最终的物理地址。

**.1循环，即在页目录表填充页表项的地址，循环1024次**

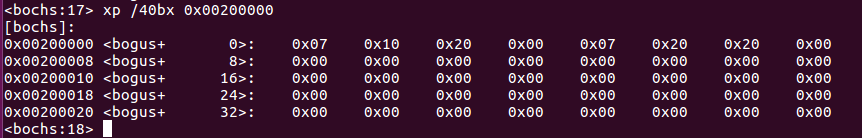
**.2循环，即在页表中填充页的物理地址，循环1024\*1024（注：为了简化所有页表项和物理地址都是连续的）**

**可以看到edi=0x00000004=4,eax=0x00201007=2101255,**输入sreg看到es=00x0010,物理地址为：0x00200000

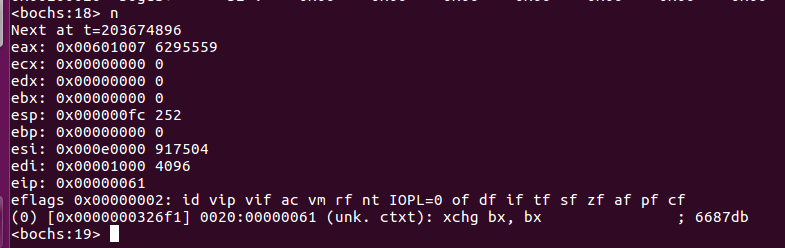




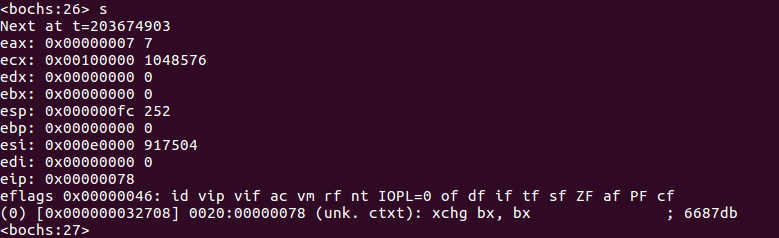
查看物理地址，发现a已经写入

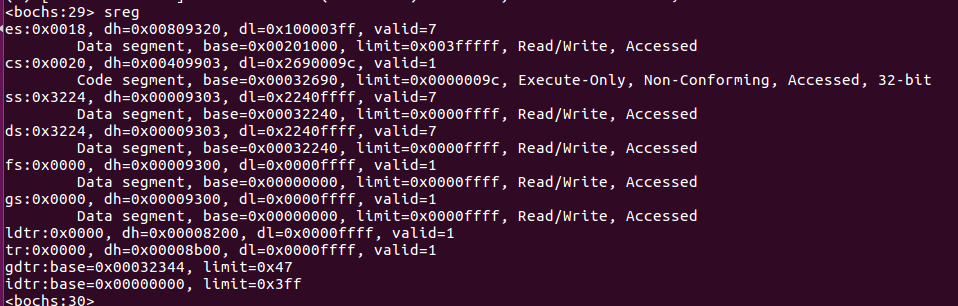


在.1循环结束后，可以看到eax已经变为0x00601007h,说明写入了0x00400000即4\*1024=4096Byte,edi变为0x00001000h

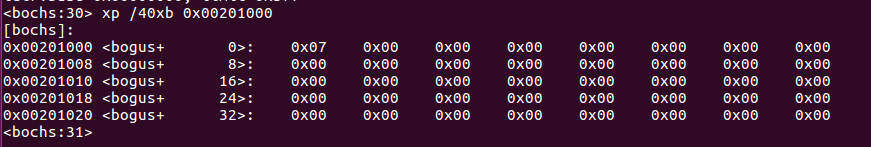


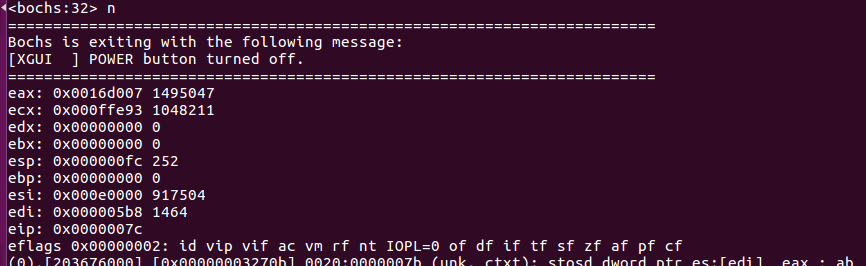
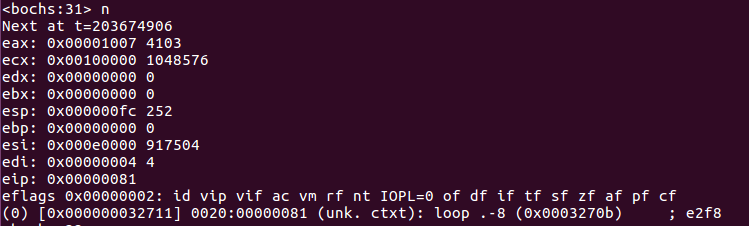
在.2循环开始前eax=0x00000007h,ecx=0x0010000h即1024\*1024，eip=0x00000078





查看寄存器es，物理地址为0x00201000,继续执行代码，可以看到a已经被写入页表项中





总体来说：.1函数在物理地址0x00200000-0x00201000中写入1024个大小为4byte的页表项，.2函数在0x00201000-0x00601000写入物理地址

### 掌握PDE，PTE的计算方法

页目录表地址：CR3高20位

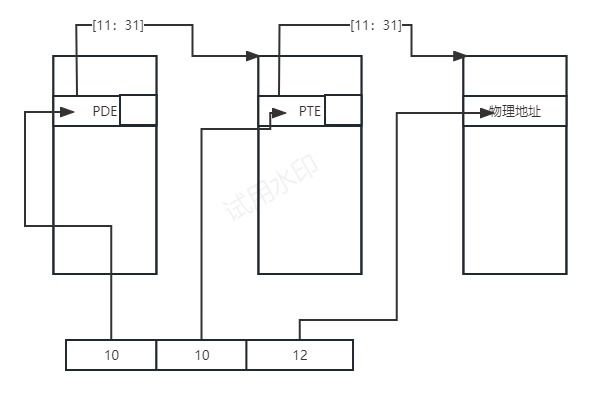
PDE地址：页目录表地址+线性地址高10位（左移2位）

页表基址：PDE高20位

PTE地址：页表地址+线性地址中间10位（左移2位）

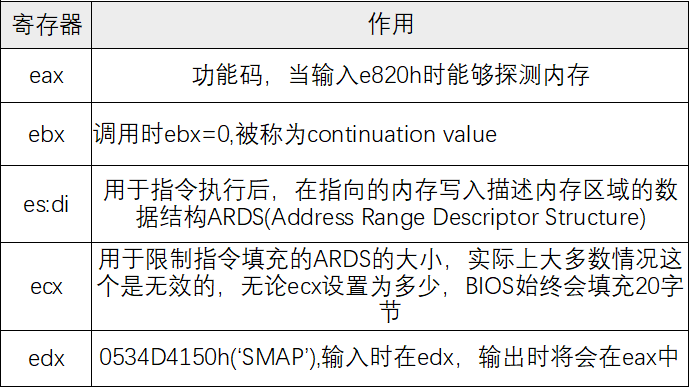
物理页基址：PTE高20位

pte初始化时不用基地址是因为最大寻址就为4GB=2^32,若不从0开始,则高位的页表项对应的物理地址会缺失，每个页表有1024条页表项，共有1024个页表，每个页表项寻址2^12bit。映射图的结构如下：



### 掌握熟悉如何获取当前系统内存布局的方法

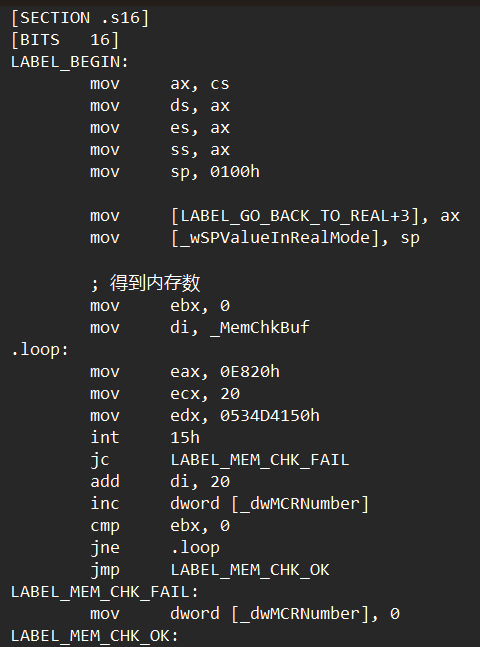
1. 调用中断15h获取内存信息；循环读取ARDS结构（地址范围描述符结构），在调用中断之前将 eax 赋值为 0E820h，将 ebx 赋值为 0，将 edx 赋值为 0534D4150h(’SMAP’)



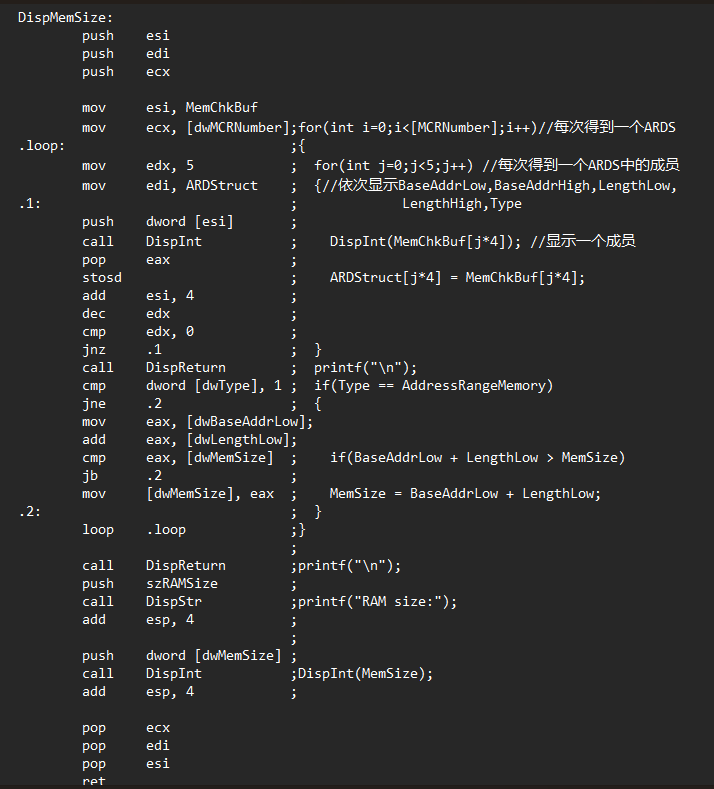
1. 定义一块256字节的缓冲区，将每次得到的内存信息连续写入这块缓冲区，形成一个结构体数组。然后在保护模式下把它们读出来，显示在屏幕上，以此得到内存的容量。

关键代码如下：

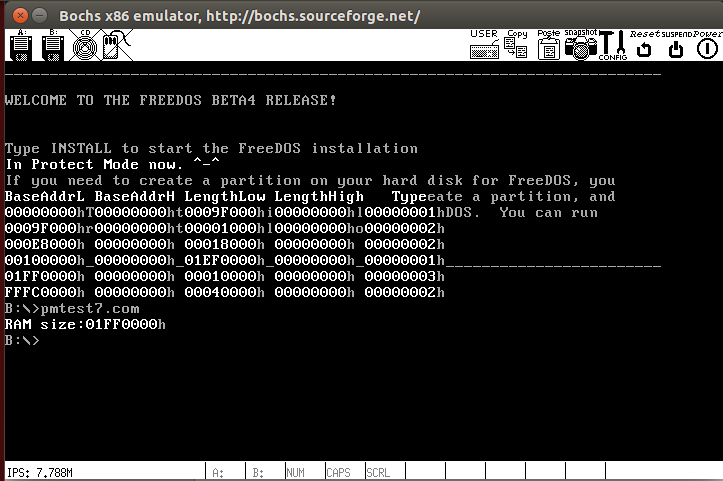
首先下列代码段定义了一块 256 字节的缓冲区，最多可以存放12个20字节大小的结构体（现在还不知道它到底够不够用，这个大小仅仅是凭猜测设定）。将每次得到的内存信息连续写入这块缓冲区，形成一个结构体数组。这个过程中，eax, ecx, edx 的值都不会变，ebx 的值先不管，es:di 指向 \_MemeChkBuf 的开始处，每次 di += 20 ，\_dwMCRNumber += 1 。最后当 CF 被置位或者 ebx 为零，则结束循环。



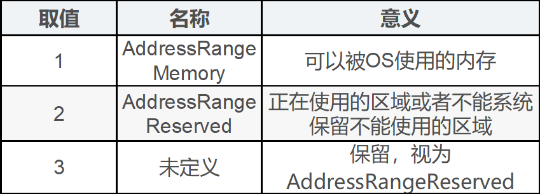
然后在保护模式下的32位代码，用双重循环把它们读出来，每次读取一个 ARDStruct ，打印其中每个成员的各项；然后根据当前结构的类型，得到可以被操作系统使用的内存的上限，结果存入 dwMemSize 中，最后用 DispInt, DispStr （和 DispAL, DispReturn 一起放入 lib.inc 中）显示在屏幕上，凭借它们得到内存的容量：



1. 运行程序可得结果：

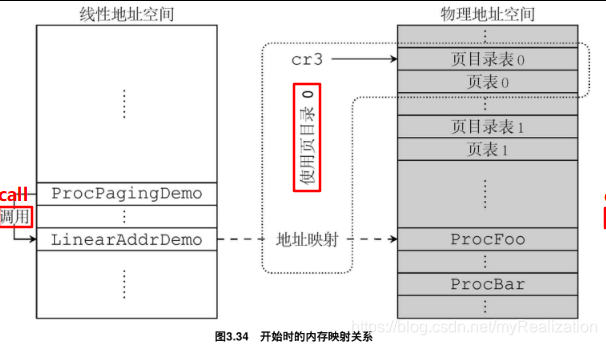
：

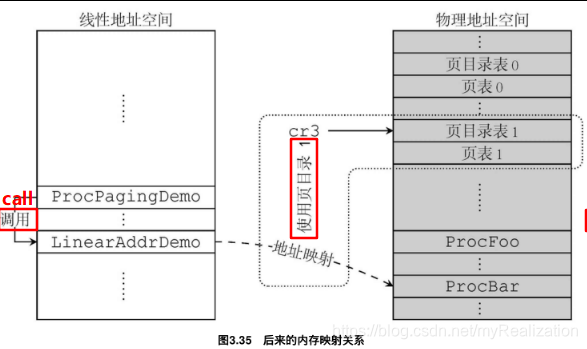
1. 分析结果：根据上下图可知，操作系统有两段可以被os使用的内存，两段被保留，一段未定义。同时可得到操作系统能使用的最大内存地址为 01FFFFFFh （从零开始） ，所以机器有32MB的内存。

（地址范围描述符结构）（Type的取值及其意义）

### 内存地址映射关系的切换，并画出流程图

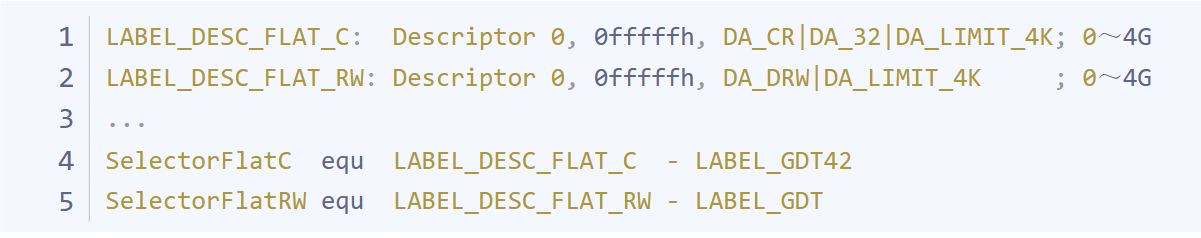
1. 内存地址切换目的





1. 内存地址切换代码解析

首先，我们用到了另一套页目录表和页表，原来的页目录表和页表段已经不够用了。这里，我们用一个段 FLAT 存放两套页目录表和页表。之所以用两个描述符描述这个段，是因为我们不仅要读写这段内存，还要执行其中的代码，所以属性要求不一样。两个段的段基址都是零，长度都是4GB。



然后修改启动分页的代码：

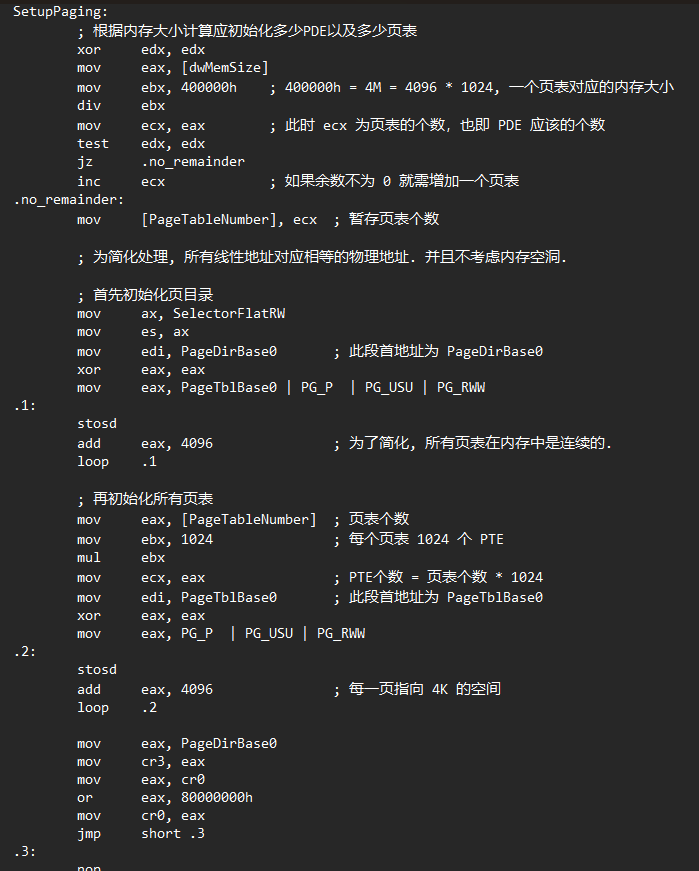
\* 我们要存储页表的个数，因为有多个页目录和页表；



\* es = SelectorFlatRW ，只需要将地址赋给 edi ，es:edi 就会指向对应的物理地址；

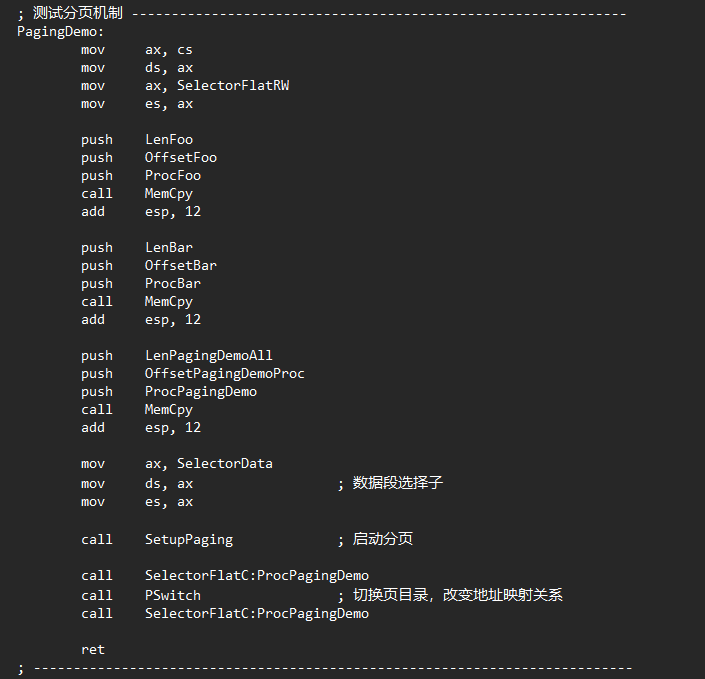
\* 然后，用 stosd 将eax 的内容持续复制4个字节给 es:edi ，并增加 edi 4个字节指向下一个表项；

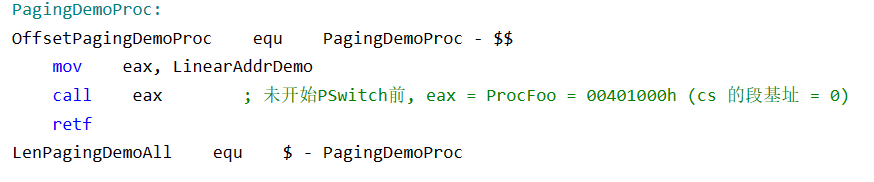
(需要注意的就是这里的页目录表的地址PageDirBase0，页表的地址是PageTblBase0，强调这点的原因在于之后PSwitch 这个函数中则是 PageDirBase1 和 PageTblBase1。也就是说实际上数据中有两个页面管理的数据结构(页目录表和页表合起来相当于一个管理页面的数据结构)



\* 新建一个函数 PagingDemo ，将分页的内容放在里面

就是拷贝三份代码分别到ProcFoo, ProcBar, ProcPagingDemo 处(这四个都是物理内存,并且后面因为段基址是0(FLAT\_C 段基址)于是很容易访问到了物理地址)。然后开启分页模式(其实几乎没什么影响 因为仍然和分段一样 线性地址 = 物理地址)。然后调用 被拷贝的函数 ProcPagingDemo ,ProcPagingDemo 函数调用 ProcFoo函数（此时LinearAddrDemo指向的是ProcFoo）,显示字符 "Foo"然后两次返回(ret)。





\* 切换页表后 : call PSwitch

PSwitch函数初始化页目录和页表后，依旧有线性地址 = 物理地址 。

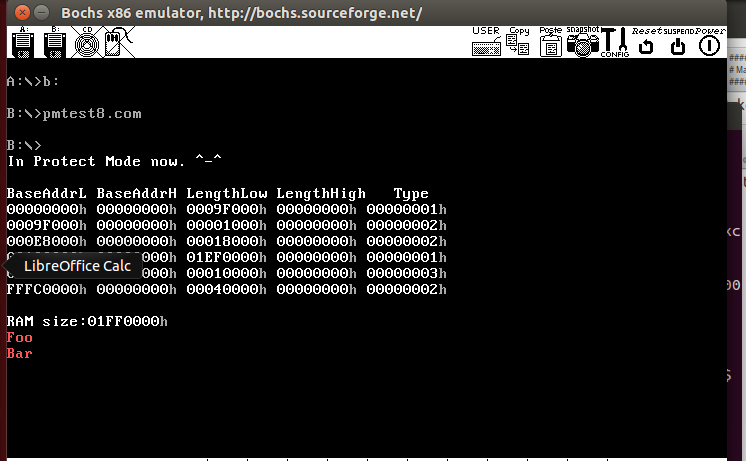
但是后续LinearAddrDemo对应的页表项的地址PaocFoo(00401000h) 换成了 ProcBar(00501000h) 的地址。

于是便调用了 ProcBar 段的代码,而这段的代码是第二次调用MemCpy时候复制到物理地址 0:0x501000h 的。也就是显示一个字符串 "Bar", 然后返回到PagingDemo的最后一句 ret,再次返回。

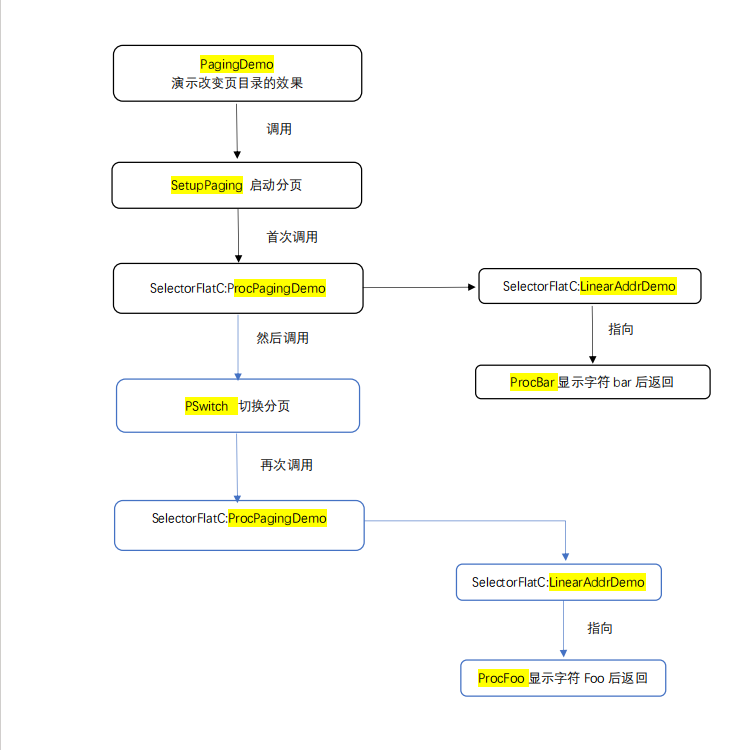


但是其实其他地方(除了这个页之外)的线性地址 = 物理地址依然成立。也是上面这段代码很小,一定是小于 4k(一页的大小)，于是只需要修改一个页表项即可。

1. 程序运行结果



1. 地址映射切换流程图



### 依据实验代码完成虚拟地址到物理地址的函数

我们将转换函数命名为L2P（意为Linear to Physical），它接受一个虚拟地址的输入，输出转换后的物理地址；然后我们编写一个测试函数TestL2P，它可以在屏幕上输出虚拟地址和与之对应的物理地址。

对于测试分页机制的主函数而言，需要在切换页目录之前和之后分别调用一次TestL2P来测试。下面我们定义一个样例输入：

**LinearAddrExp equ 00401000h**

它将存放于eax寄存器中，并通过eax输入到TestL2P中。相关代码为：

**call SetupPaging ; 启动分页**

**mov eax, LinearAddrExp**

**call TestL2P ; 切换前**

**call SelectorFlatC:ProcPagingDemo**

**call PSwitch ; 切换页目录**

**call SelectorFlatC:ProcPagingDemo**

**mov eax, LinearAddrDemo**

**call TestL2P ; 切换后**

下面来看TestL2P函数，我们需要在线性地址和物理地址之前加上描述的字符串：

**\_linear: db "LinearAddress:", 0**

**\_physical: db "PhysicalAddress:", 0**

最后，我们来看TestL2P的具体实现，该函数调用了lib.inc用于打印操作的函数：

**TestL2P:**

**push ds**

**push edx**

**mov dx, SelectorData**

**mov ds, dx**

**push eax**

**call DispReturn**

**pop eax**

**push eax**

**push Linear**

**call DispStr ;开始打印虚拟地址，首先打印描述语句**

**add esp, 4**

**pop eax**

**push eax**

**call DispInt ;然后打印具体虚拟地址信息**

**pop eax**

**call L2P**

**push eax**

**call DispReturn**

**pop eax**

**push eax**

**push Physical**

**call DispStr ;开始打印物理地址，首先打印描述语句**

**add esp, 4**

**pop eax**

**push ebx**

**call DispInt ;然后打印具体物理地址信息**

**pop ebx**

**pop edx**

**pop ds**

**ret**

最后，我们关注L2P的具体实现，它从eax寄存器中读取线性地址，将计算得出的物理地址返回到ebx中。如果该地址不存在，就转到not\_exist分支中，将返回值全部置1以给出错误提示。下面是具体的地址对应关系：

* 页目录表地址：CR3高20位
* PDE地址：页目录表地址+线性地址高10位（左移2位）
* 页表基址：PDE高20位
* PTE地址：页表地址+线性地址中间10位（左移2位）
* 物理页基址：PTE高20位

实现代码如下：

**Linear2Physical:**

**push ds**

**mov bx, SelectorFlatRW**

**mov ds,bx**

**mov ebx,cr3 ;从cr3中获得页目录物理地址**

**and ebx,0xfffff000**

**;push eax**

**;call DispInt**

**;add esp,4**

**mov ecx, eax**

**shr eax,22**

**and eax,0x3ff**

**shl eax,2 ;寻找对应的页目录项，下一句会从页目录项中得到页表物理地址**

**mov ebx,[eax+ebx]**

**test ebx,0x00000001**

**jz .not\_exist**

**and ebx, 0xfffff000**

**mov eax, ecx**

**shr eax,12**

**and eax,0x3ff**

**shl eax,2 ;寻找对应的页表项，下一句会从页表项中得到页物理地址高20位**

**mov ebx,[eax+ebx]**

**test ebx,0x00000001**

**jz .not\_exist**

**and ebx,0xfffff000**

**and ecx,0xfff**

**add ebx,ecx ;页物理地址高20位和低12位进行拼接，得到完整的物理地址**

**mov eax,ebx**

**jmp .exit**

**.not\_exist:**

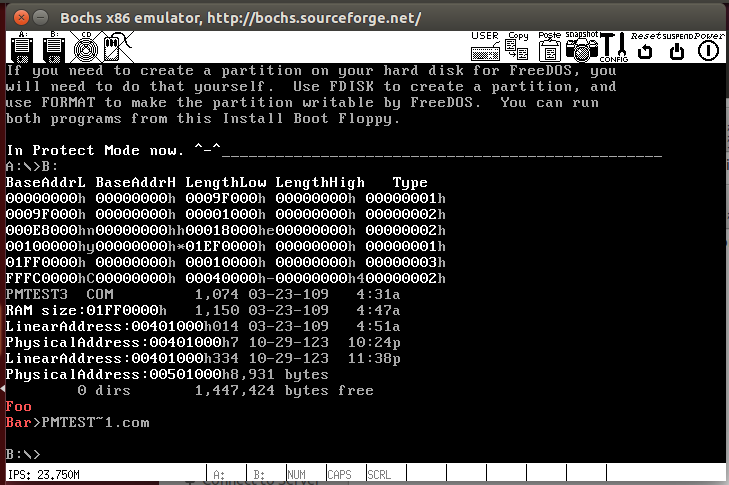
**mov ebx,0xffffffff**

**.exit:**

**pop ds**

**ret**

下面是最终程序的输入结果：



在切换页目录之后，Linear Address没有改变（00401000h - 00401000h），但Physical Address却改变了(00401000h - 00501000h)，证明该函数编写成功。

### 依据实验代码完成分页管理的alloc和free函数

首先，我们定义页面分配的函数为AllocPages，它接受的输入为分配的页数，存放在eax寄存器中，该函数会分配指定页数的页面；然后是对应页面释放的函数FreePages，它接受的输入为存放在eax寄存器中的虚拟地址、以及存放在ebx容器中的页数，函数会将该页面释放，也就是将对应的PDE、PTE全部标记为不可用。

我们编写测试函数TestAllocFree，该函数在主函数开启分页管理之后执行。

TestAllocFree的代码如下：

**TestAllocFree:**

**xchg bx,bx ;该断点测试alloc之前的地址映射关系**

**mov eax,4**

**call AllocPages**

**xchg bx,bx ;该断点测试alloc之后的地址映射关系**

**mov eax,ebx**

**mov ebx,4**

**call FreePages**

**xchg bx,bx ;该断点测试free之后的地址映射关系**

**ret**

对于AllocPages来说，我们需要找到空闲页的地址，才可以进行分页的操作。

首先，我们定义位图BitMap，以及线性地址的初始分配地址：

**\_BitMap: times 32 db 0xff**

**times 32 db 0x0**

**BitMapLen: equ $ - \_BitMap**

**\_AvaLinearAddress dd 0x8000\_0000**

之后，我们通过位图来寻找物理内存中的空闲页并通过eax寄存器返回物理地址，找不到直接使用hlt命令停机；找到就将对应物理地址存放到eax寄存器中：

**Alloc4kPage:**

**push ds**

**push es**

**xor eax, eax**

**mov ax, SelectorFlatRW**

**mov es, ax**

**mov ax, SelectorData**

**mov ds, ax**

**.search:**

**bts [BitMap], eax ;在bitmap中寻找0比特**

**jnc .find**

**inc eax**

**cmp eax, BitMapLen\*8**

**jl .search**

**hlt ;找不到就直接halt**

**.find:**

**shl eax, 12 ;寻找到就将该比特的物理地址存放到eax寄存器中**

**pop es**

**pop ds**

**ret**

之后，我们开始分配页面，下面是AllocPages函数。AllocPages首先获取线性地址和CR3的值，由此获得PDE和PTE，如果PDE对应的页表项不存在，则创建对应的页表；否则直接修改对应页表项，将其替换为Alloc4kPage函数返回的可用物理地址：

**AllocPages:**

**push ds**

**push es**

**mov bx, SelectorFlatRW**

**mov ds, bx**

**mov bx, SelectorData**

**mov es, bx**

**mov ecx, eax ; ECX中存放页数，即之后的循环迭代次数**

**mov ebx, 4096**

**mul ebx**

**mov ebx, [es:AvaLinearAddress] ; EBX中存放原有的线性地址**

**add [es:AvaLinearAddress],eax ; 更新线性地址的值**

**push ebx**

**mov eax, ebx**

**mov ebx, cr3**

**and ebx, 0xfffff000 ;EBX存放CR3的高20位，也就是页目录表地址**

**and eax, 0xffc00000 ;线性地址高10位**

**shr eax, 20**

**add ebx, eax ;合成PDE地址**

**mov edx, ebx**

**mov ebx, [ebx] ;得到对应的表项，存放在ebx中**

**test ebx, 0x0000\_0001 ;判断PDE表项是否存在**

**jnz .pde\_exist**

**mov ebx, cr3 ;不存在就创建一个页表**

**mov ebx, [ebx]**

**and ebx, 0xfffff000**

**shl eax, 10**

**add ebx, eax**

**or ebx, 0x0000\_0007**

**mov [edx], ebx**

**.pde\_exist:**

**mov eax,[esp]**

**and ebx, 0xfffff000**

**and eax, 0x003ff000**

**shr eax, 10**

**add ebx, eax**

**.change\_pte:**

**call Alloc4kPage ;修改对应页表项，分配地址在eax寄存器中**

**or eax, 0x00000007 ;标记为可用**

**mov [ebx], eax**

**add ebx, 4**

**loop .change\_pte ;在这里进行循环，直到分配页面数满足要求**

**pop ebx**

**pop es**

**pop ds**

**ret**

然后是释放页面的函数FreePages。该函数直接将分页机制对应的表项设置为不可用。首先，根据线性地址和CR3寄存器中的值找到PDE和PTE，根据页数进行迭代，将它们的值设置为不可用；之后，BitMap中对应位置的值也需要置0：

**FreePages:**

**push ds**

**push es**

**push ebx**

**push eax**

**mov bx, SelectorFlatRW**

**mov ds, bx**

**mov bx, SelectorData**

**mov es, bx**

**mov ebx, cr3**

**and ebx, 0xfffff000**

**and eax, 0xffc00000**

**shr eax, 20**

**add ebx, eax ;根据线性地址和CR3得到对应的PDE**

**mov edx, [ebx]**

**and edx, 0xfffffff8**

**mov [ebx], edx**

**mov ebx, [ebx]**

**mov eax, [esp]**

**add esp, 4**

**and ebx, 0xfffff000**

**and eax, 0x003ff000 ;根据线性地址的中间10位找到对应的PTE**

**shr eax,10**

**add ebx, eax**

**mov ecx, [esp] ;表示需要释放页的页数**

**add esp,4**

**.change\_pte:**

**mov eax, [ebx]**

**and eax, 0xfffffff8 ;标记为不可用**

**mov edx, eax**

**shr edx, 12**

**btr [BitMap], edx ;使用BTR指令将BitMap的相应位置置0**

**mov [ebx], eax**

**add ebx,32**

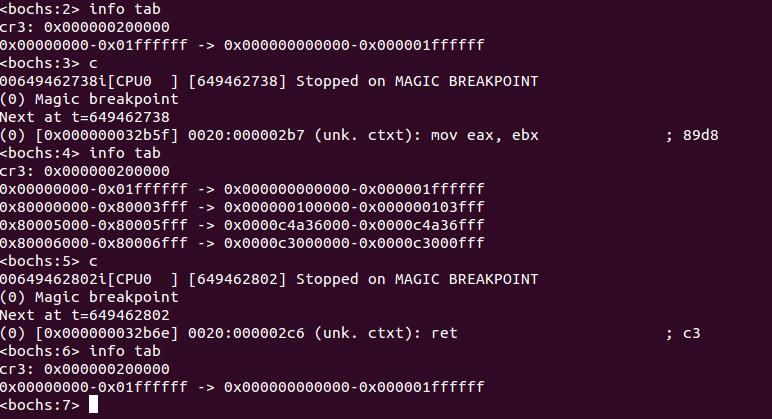
**loop .change\_pte**

**pop es**

**pop ds**

**ret**

下面是具体的实验结果：



在调用AllocPages之前，存在一条地址映射；调用AllocPages之后，存在四条地址映射；调用FreePages之后，又只剩下了一条地址映射。由此可见，内存管理的alloc函数与free函数编写成功。

## 本次实验要思考的问题

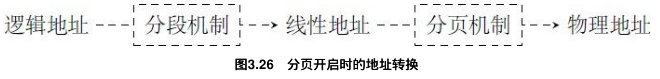
### 分页和分段有何区别？在本次实验中，段页机制是怎么搭配工作的？

**什么是"段"**：段是信息的逻辑单位，它含有一组其意义相对完整的信息，分段的目的是为了更好地满足用户的需要；

**什么是"页"**：页是一块内存，是信息的物理单位，分页是出于系统管理的需要，提高内存的利用率。页的大小在不同的机器上都不同，本次实验讨论页大小为 4KB 的情况；

**段页机制**：分段机制已经提供了很好的保护机制，而分页机制的主要目的在于实现虚拟存储器，线性地址中任何一个页都可以映射到物理地址的任何一个页，使内存管理变得更灵活。

在未打开分页机制时，线性地址等同于物理地址，即逻辑地址通过分段机制直接转换成物理地址；分页开启后，分段机制将逻辑地址转换为线性地址，线性地址通过分页机制转换为物理地址。

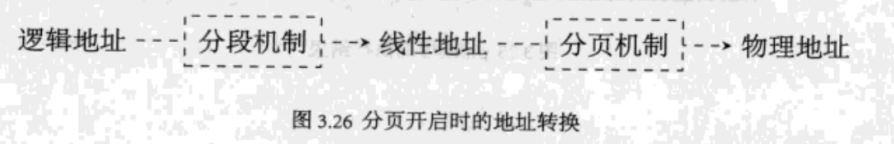


### PDE、PTE，是什么？例程中如何进行初始化？CPU是怎样访问到PDE、PTE，从而计算出物理地址的？

1. PDE和PTE是分页机制中的两个重要数据结构。其中PDE（Page Directory Entry）是页目录项，用于映射虚拟地址到页表。PTE（Page Table Entry）是页表项，用于映射虚拟地址到物理地址。
2. 例程中的初始化过程：
   1. 定义两个段，分别存放页目录表和页表，并分别分配大小为4KB和4MB物理内存空间
   2. 初始化PDE：es:edi指向页目录表的首地址，循环1024次给所有页目录表中的PDE赋值
   3. 初始化PTE：es:edi指向页表段的首地址，循环1024\*1024次给所有页目录表中的PDE赋值
   4. 最终将线性地址映射到相同的物理地址
3. CPU访问PDE和PTE的过程：
   1. CPU从寄存器CR3指定的页目录中，根据线性地址的高10位作为页目录项的索引，找到对应的页目录项，得到页表地址
   2. 在页表中根据线性地址的第12到21位得到物理页首地址
   3. 该首地址与线性地址第12位相加，得到最终的物理地址

### 开页机制之后，在GDT表中、在PDE、PTE中存的地址是物理地址、线性地址，还是逻辑地址，为什么？

在启用分页机制之后，在GDT表中、PDE和PTE中存储的是物理地址。这是因为分页开启时，分段机制将逻辑地址转换成线性地址，线性地址再通过分页机制转换成物理地址，即如下图所示：



### 为什么PageTblBase初始值为2M+4K？能不能比这个值小？

1. PageTblBase初始值为2MB+4KB是为了将页目录和页表放置在合适的内存位置上。
2. 页目录表位于地址2MB处，共有1024个表项，每个表项占用4字节，因此页目录表占用1024\*4B=4KB空间；由于页表紧跟在页目录表后面，所以页表地址的初始值是2MB+4KB，这样才能确保页目录和页表不会与其他重要的系统数据结构冲突，并且可以提供足够的地址空间来进行映射。
3. 将PageTblBase初始值设置为2MB+4KB后，共有1024\*1024个页表，页表占用4MB空间，能够映射到4GB的内存空间。如果PageTblBase的初始值比2MB+4KB小，则页目录表的表项会少于1024个，能够映射到的地址范围会缩小，可能会导致地址映射范围不足。

# 实验结果总结

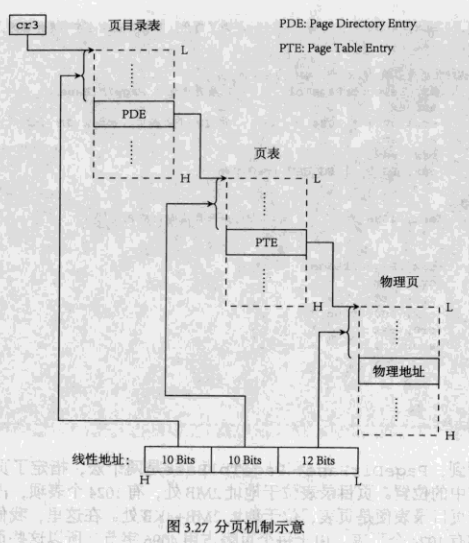
（对实验结果进行分析。并理论联系实际，思考并列出本实验对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。）

## 分页机制概述

如图采用的两级页表，第一级叫页目录，存储在一个物理页中，每个表项对应第二级的一个页表；第二级的每一个表项对应一个物理页。页目录的表项简称 PDE，页表的表项简称 PTE。

进行转换时，先从由寄存器 cr3 指定的页目录中根据线性地址的高 10 位得到页表地址，然后再页表中根据线性地址的第 12 位到 21 位得到物理页首地址，将这个首地址加上线性地址的低 12 位得到物理地址。

在本次实验程序里指定页目录表和页表在内存中的位置的宏定义过程中，需要注意：页目录表位于地址 2MB 处，有 1024 个表项，占用 4KB 空间，紧接着目录表便是页表，位于地址 2MB+4KB 处。为了代码的清晰渐变，程序定义了两个段，分别存放页目录表和页表，当cr0的最高位PG=1时，启动分页机制。



## PDE和PTE

PDE和PTE是分页机制中的两个重要数据结构。其中PDE（Page Directory Entry）是页目录项，用于映射虚拟地址到页表。PTE（Page Table Entry）是页表项，用于映射虚拟地址到物理地址。

处理器会将最近常用的页目录和页表项保存在一个叫做TLB的缓冲区中。只有在TLB中找不到被请求页的转换信息时，才会到内存中去寻找。这样就大大加快了访问页目录和页表的时间。

当页目录或页表项被更改时，操作系统应该马上使TLB中对应的条目无效，以便下次用到此条目时让它获得更新。而当cr3被加载时，所有TLB都会自动无效，除非页或页表条目的G位被设置。

## cr3

cr3 用于保存页目录表页面的物理地址，因此被称为 PDBR（Page-Directory Base Register）。由于目录是页对齐的，所以仅高 20位有效，用来表示4KB对齐的页表，低 12 位保留供更加高级的处理器使用。向 cr3 中装入一个新值时，低 12 位必须为 0；但从cr3 中取值时，低 12 位被忽略。cr3的结构如下图：



# 个人分工及心得体会

（每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。）

## 张子航

完成本次实验内容的第1，2，3题，并撰写实验报告中的对应部分，以及实验步骤和实验总结的部分。

本次实验，让我深刻的认识到了分页机制的逻辑思路，认识到了分页机制存在的意义，也了解到了PDE、PTE及其计算方法，对段页机制有了一个较深的印象。同时，在本次实验过程中，也让我锻炼了对代码的调试能力，以及对汇编代码的调试能力。学习通过设置断点等方式，通过观察寄存器信息、物理地址信息等方式来了解汇编代码的执行过程及其发生的变化所造成的影响。本次实验也提高了我对操作系统课程相应部分的知识理解，包括段页机制是怎么搭配工作的，PTE、PDE的作用，熟悉了如何获取当前系统内存布局的方法，也了解到了内存地址映射关系的切换。

## 辜汝曦

完成本次实验内容的第6题，并撰写实验报告中的对应部分（实验过程分析-本次实验内容的第6、7小节），和实验目的及实验内容部分。

本次实验应该算是比较难的一次实验，过程比较多，综合性比较强，而且由于汇编程序不便于调试，所以花了很多时间。具体来说，由于汇编没有变量名，因此需要时时刻刻记住每一个寄存器里面的值代表什么，这需要写汇编程序时思路连贯，清楚自己目前做了什么、还需要做什么；而且之前学习汇编语言时，对函数调用参数存取的push和pop不太熟悉，这次编写L2P、AllocPages和FreePages函数的过程帮我巩固了相关知识点，也进一步提升了我汇编程序的编写能力。同时，我对段页机制、PDE、PTE等实验相关内容进行了更进一步的学习，对操作系统理论课上的分页机制有了更加透彻的理解。

## 杨馨悦

完成本次实验要思考的问题的2-4题，并撰写实验报告中的对应部分（实验过程分析-本次实验要思考的问题的第2、3、4小节），撰写实验报告中实验结果总结的分页机制概述、PDE和PTE以及cr3部分。

通过本次实验，我掌握了分页机制的基本方法与思路，掌握了PDE和PTE的初始化过程及计算方法，熟悉了从逻辑地址到物理地址的映射过程，了解了cr3寄存器的结构。其中在学习分页机制的启动时，我对于页目录表和页表的基地址问题、页目录表和页表在内存当中的分布问题感到困惑，通过反复阅读教材并查阅资料，我逐渐理解了这段内容。另外，通过阅读并调试显示内存信息的代码段，我熟悉了调试命令xchg bx,bx指令的使用，同时对于如何获取内存信息及如何节约地使用内存也有了更深刻的了解，从而对分页机制的工作原理和作用有了更加清晰的认识。

## 赵敏

完成本次实验内容4、5题以及本次实验要思考的问题的第1题，并撰写实验报告中的对应部分（实验过程分析-本次实验内容的第4、5小节；实验过程分析-本次实验要思考的问题的第1节）。

通过本次实验，再次回顾了段式管理中的GDT的工作模式，结合新添加的分页机制内容，了解了段页机制的工作模式。在学习分页机制的过程中，详细探究了PDE和PTE的结构层次，仔细理解了其初始化过程部分代码，熟悉了从逻辑地址到物理地址的映射过程。为了合理使用内存，学习了利用中断int 15h获取内存布局的方法，并利用得到的内存信息计算所需的PDE数目。掌握了切换地址映射关系的方法，并分析切换流程，更深入地了解了分页机制的转换给地址映射关系带来的巨大改变。