

武汉大学国家网络安全学院教学实验报告

课程名称	操作系统设计与实践	实验日期	2024.9.25
实验名称	分页机制	实验周次	第三周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

✧ 一、实验目的及内容

实验目的

- 掌握内存分页机制的工作原理。
- 学习通过代码实现分页机制的基本步骤和方法。
- 了解分页机制下页目录表 (PDE) 和页表项 (PTE) 的计算方式。
- 熟悉如何获取系统的内存布局以及如何进行地址映射的切换。

实验内容

- 阅读实验资料，掌握分页机制的基本概念。

- 2 调试代码，分析页表和页目录表的初始化过程。
- 3 理解分页机制中的 PDE 和 PTE 的计算方法，并动手绘制映射图。
- 4 学习获取当前系统的内存布局，掌握地址映射和切换的方法。
- 5 实现分配页 (alloc_pages) 和释放页 (free_pages) 的函数，并测试最大分配空间。

原理分析

- 分页机制：分页机制是 x86 内存管理机制的第二部分。它在分段机制的基础上完成了线性地址到物理地址的转换。分段机制把逻辑地址转换成线性地址，而分页机制则把线性地址转换成物理地址。分页机制会把线性地址空间划分成页面，每个页面即为一块内存，在 80386 中页的大小是固定的 4096 字节。分页机制使得线性地址中任意一个页都能映射到物理地址中的任何一个页，从而使内存管理变得更加灵活。

分页机制的打开方法是：首先初始化页目录表、页表、CR3 的内容，使各部分指向正确的位置，然后修改 CR0 的 PG 位，使分页机制打开。
- 页目录表及页表：在两级页表的机制下，第一级叫做页目录，大小为 4KB，存储在一个物理页中，其每个表项 PDE 4 字节长，共有 1024 个表项。页目录每个表项对应第二级的一个页表，第二级的每一个页表也有 1024 个表项，每一个表项 PTE 对应一个物理页。通常，由寄存器 cr3 指定的页目录表的起始地址。在这种机制下，一个线性地址将分为三部分，进行转换时先是从由寄存器 cr3 指定的页目录中根据线性地址的高 10 位（页表号）得到页表地址，然后在页表中根据线性地址的第 12 到 21 位（页号）得到物理页首地址，将这个首地址加上线性地址低 12 位（页内偏移）便得到了物理地址。
- int 15h 及 ARDS: int 15h 通常用于取得当前物理内存信息（功能号 ax 为 0E820h）。该方法可以得到内存的大小以及对不同内存段的一些描述，这些描述将被保存在一个缓冲区中。每次调用 int 15h 将会返回一个地址范围描述符 ARDS，当 ebx 的值为 0 并且 CF 没有进位表示此时返回的是最后一个地址范围描述符。

地址范围描述符 ARDS (Address Range Descriptor Struct)，其结构如下：

偏移	名称	意义
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	长度（字节）的低 32 位
12	LengthHigh	长度（字节）的高 32 位
16	Type	这个地址范围的地址类型

✧ 二、实验环境与步骤

实验环境

- 虚拟化软件：VMware Workstation Pro，用于搭建和运行实验环境。
- 操作系统：Kubuntu 18.04.5 32 位，作为实验的虚拟机操作系统，进行调试和实验操作。

实验步骤概述

- 1 调试代码，设置断点，分析关键代码循环中页表和页目录表的初始化过程，掌握分页机制的实现。
- 2 掌握 PDE、PTE 的计算：分析代码中的地址映射关系，绘制映射图并回答为什么 PDE 初始化时需要添加基地址，而 PTE 不需要。
- 3 获取内存布局：通过 `int 15h` 中断读取物理内存信息，打印出当前系统的内存布局。
- 4 地址映射切换：研究分页机制如何通过页表和页目录表进行内存地址映射的切换。
- 5 编写内存管理函数：完成 `alloc_pages` 和 `free_pages` 函数，实现页分配与释放，测试最大分配空间。

✧ 三、实验过程分析

调试代码，掌握分页机制基本方法与思路

涉及基本分页机制的代码为 `pmtest6.asm`，其关键代码为下面的循环：

```
SetupPaging:
    ; 为简化处理，所有线性地址对应相等的物理地址。

    ; 首先初始化页目录
    mov     ax, SelectorPageDir      ; 此段首地址为 PageDirBase
    mov     es, ax
    mov     ecx, 1024                ; 共 1K 个表项
    xor     edi, edi
    xor     eax, eax
    mov     eax, PageTblBase | PG_P | PG_USU | PG_RWW                ;
.1:
    stosd                                ; 将eax的内容复制到edi的内存空间，复制四个字节，并将edi加4个字节
    add     eax, 4096                  ; 为了简化，所有页表在内存中是连续的。
    loop    .1

    ; 再初始化所有页表（1K 个，4M 内存空间）
    mov     ax, SelectorPageTbl      ; 此段首地址为 PageTblBase
    mov     es, ax
    mov     ecx, 1024 * 1024         ; 共 1M 个页表项，也即有 1M 个页
    xor     edi, edi
    xor     eax, eax
    mov     eax, PG_P | PG_USU | PG_RWW
.2:
    stosd                                ; 每一页指向 4K 的空间
    add     eax, 4096
    loop    .2

    mov     eax, PageDirBase
    mov     cr3, eax
    mov     eax, cr0
    or      eax, 80000000h
    mov     cr0, eax
    jmp     short .3

.3:
    nop                                ;

    ret
; 分页机制启动完毕 -----
```

这段代码的功能是开启分页机制。开启分页机制的方法为：

1) 填充 PDE、PTE:

让页目录表、页表分别作为一段，段寄存器 `es` 对应页目录表段，再将 `edi` 设置为 0，此时 `es: edi` 就指向了目录表的开始。

第 0 个 PDE 的初始值 `eax` 为 `PageTblBase | PG_P | PG_USU | PG_RWW`。即存在的可读可写的用户级别页表，页表基址为 `PageTblBase`。遍历 1024 个 PDE，每次初始值叠加 4096。初始添加 `PageTblBase` 以及叠加 4096 的原因将在下题详述。

遍历初始化 PDE 的过程采用 `stosd` 指令完成。`stosd` 将 `eax` 的内容复制到 `edi` 的内存空间，复制四个字节，并将 `edi` 加 4 个字节。设置循环.1 执行 `stosd` 1024 次即可初始化 PDE。

初始化 PTE 的过程类似，`eax` 为 `PG_P | PG_USU | PG_RWW`，每次叠加 4096，循环 1024 * 1024 次，对应代码中的.2 循环。

2) 设置 `cr3`：`cr3` 存放页目录表的基址，将 `PageDirBase` 放入 `cr3` 即可。

3) 打开 `cr0` 的 PG 位。使用 `or` 指令将 `cr0` 寄存器的最高位置 1，开启分页机制。

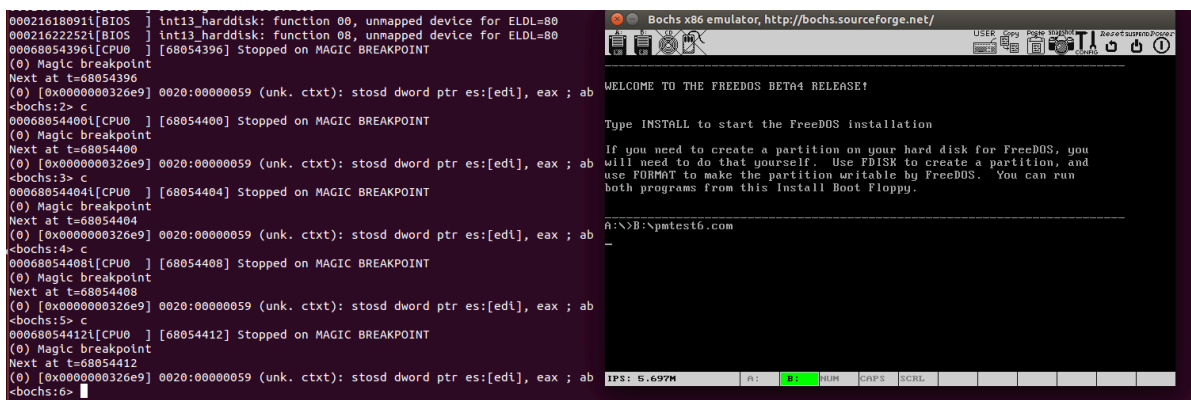
代码调试:

- 调试第一个循环。将.s1 修改如下:

```
.1:
    xchg bx, bx
    stosd                ; 将eax的内容复制到edi的内存空间, 复制四个字节, 并将
    edi加4个字节
    add eax, 4096        ; 为了简化, 所有页表在内存中是连续的.
    loop .1
```

为了在 bochs 中调试 .com 文件, 需要修改 bochsrc 文件, 在文件末尾添加” magic_break: enabled = 1”, 然后在需要添加调试的语句前面增加 xchg bx bx, 则执行时候 bochs 就会停下来。

可以看到, 在 DOS 状态下运行 pmtest6.com, 程序停在了调试点:



The screenshot shows the Bochs x86 emulator running the pmtest6.com program. The left pane displays the CPU state and memory dump, showing the program has stopped at a magic breakpoint. The right pane shows the FreeDOS installation screen, which is currently displaying the welcome message and instructions for creating a partition.

为了方便观察, 让循环执行五次, 查看页目录表对应内存 (0x00200000), 如下:

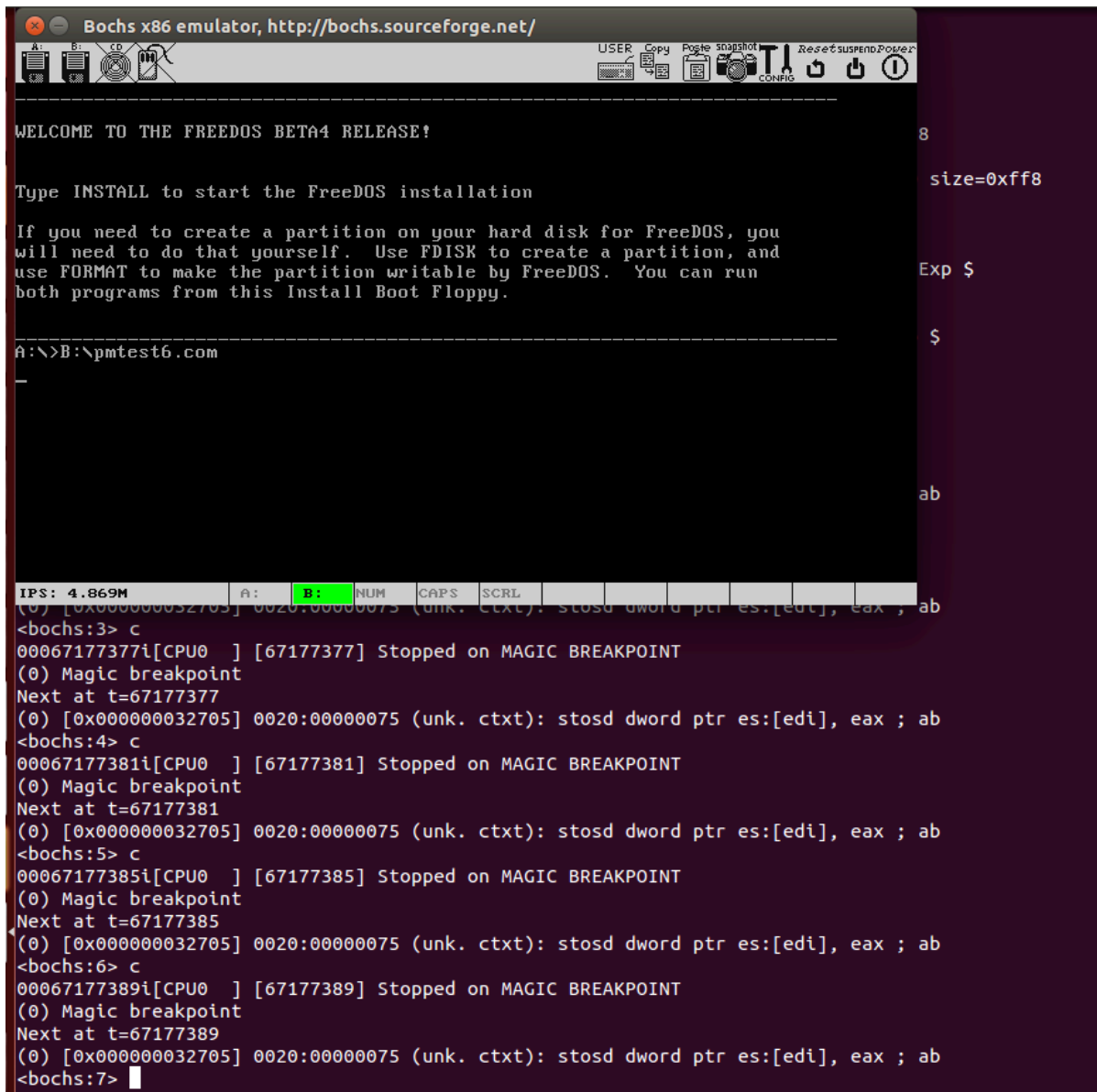
```
<bochs:7> x /64xb 0x00200000
[bochs]:
0x00200000 <bogus+ 0>: 0x07 0x10 0x20 0x00 0x07 0x20 0x20 0x00
0x00200008 <bogus+ 8>: 0x07 0x30 0x20 0x00 0x07 0x40 0x20 0x00
0x00200010 <bogus+ 16>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00200018 <bogus+ 24>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00200020 <bogus+ 32>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00200028 <bogus+ 40>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00200030 <bogus+ 48>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00200038 <bogus+ 56>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
<bochs:8> █
```

前五个 PDE 分别为: 0x00201007, 0x00202007, 0x00203007, 0x00204007, 0x00204007, 符合 $(PageTblBase + n * 4096) | PG_P | PG_USU | PG_RWW$ 的形式, PDE 正确初始化。经过 1024 次循环, 页目录表中所有的 PDE 都能够被赋为合适的值。

- 调试第二个循环。将.s2 修改如下:

```
.2:
    xchg bx, bx
    stosd
    add eax, 4096        ; 每一页指向 4K 的空间
    loop .2
```

可以看到，在 DOS 状态下运行 pmtest6.com，程序停在了调试点：



```
WELCOME TO THE FREEDOS BETA4 RELEASE!

Type INSTALL to start the FreeDOS installation

If you need to create a partition on your hard disk for FreeDOS, you
will need to do that yourself. Use FDISK to create a partition, and
use FORMAT to make the partition writable by FreeDOS. You can run
both programs from this Install Boot Floppy.

A:\>B:\pmtest6.com

IPS: 4.869M  A:  B:  NUM  CAPS  SCRL
(0) [0x000000032705] 0020:00000075 (unk. ctxt): stosd dword ptr es:[edi], eax ; ab
<bochs:3> c
00067177377i[CPU0 ] [67177377] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=67177377
(0) [0x000000032705] 0020:00000075 (unk. ctxt): stosd dword ptr es:[edi], eax ; ab
<bochs:4> c
00067177381i[CPU0 ] [67177381] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=67177381
(0) [0x000000032705] 0020:00000075 (unk. ctxt): stosd dword ptr es:[edi], eax ; ab
<bochs:5> c
00067177385i[CPU0 ] [67177385] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=67177385
(0) [0x000000032705] 0020:00000075 (unk. ctxt): stosd dword ptr es:[edi], eax ; ab
<bochs:6> c
00067177389i[CPU0 ] [67177389] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=67177389
(0) [0x000000032705] 0020:00000075 (unk. ctxt): stosd dword ptr es:[edi], eax ; ab
<bochs:7> 
```

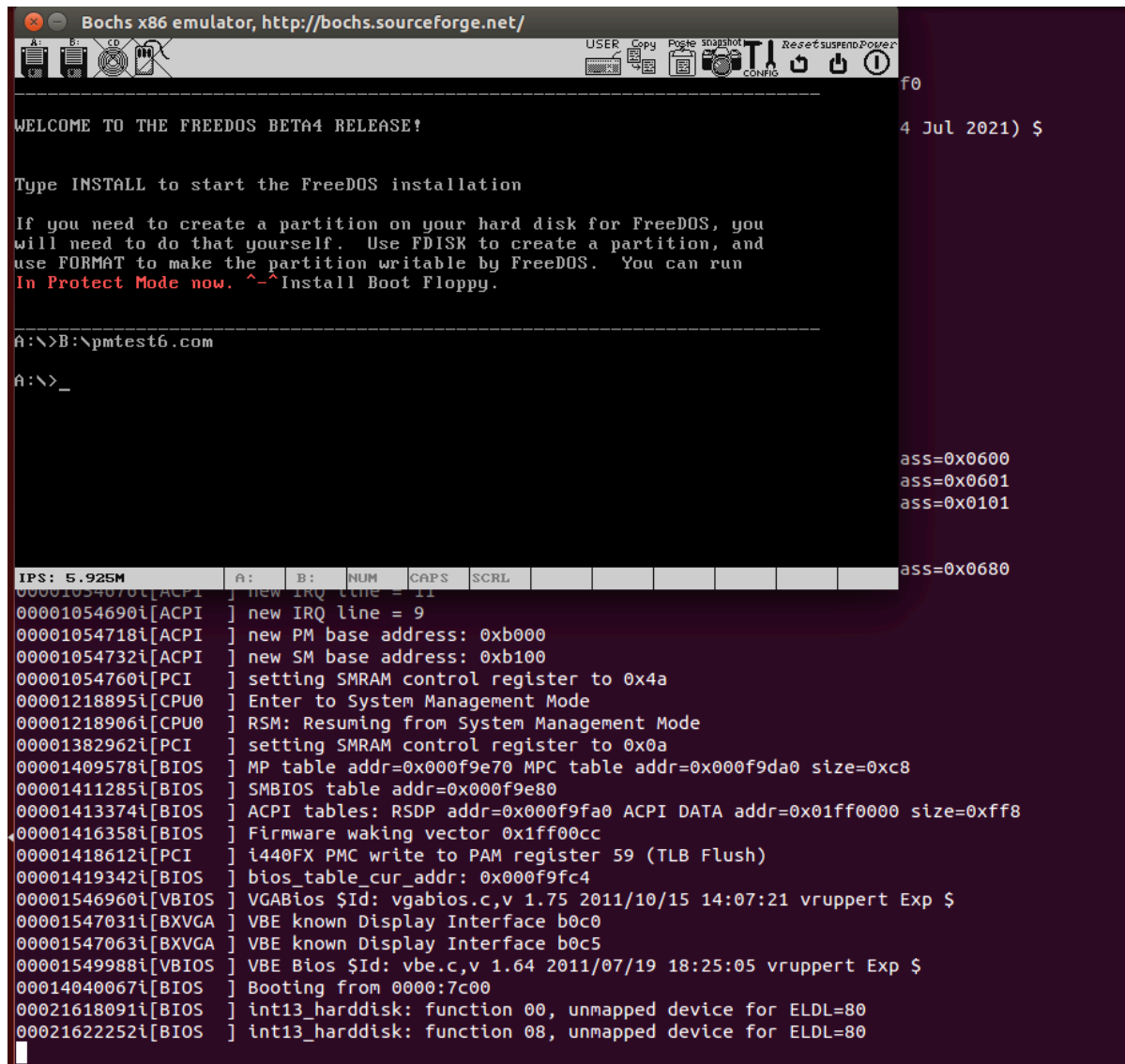
为了方便观察，让循环执行五次，查看页表段对应内存（基址 0x00201000），如下：

```
<bochs:7> x /64xb 0x00201000
[bochs]:
0x00201000 <bogus+ 0>: 0x07 0x00 0x00 0x00 0x07 0x10 0x00 0x00
0x00201008 <bogus+ 8>: 0x07 0x20 0x00 0x00 0x07 0x30 0x00 0x00
0x00201010 <bogus+ 16>: 0x07 0x40 0x00 0x00 0x00 0x00 0x00 0x00
0x00201018 <bogus+ 24>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00201020 <bogus+ 32>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00201028 <bogus+ 40>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00201030 <bogus+ 48>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00201038 <bogus+ 56>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
<bochs:8> 
```

前五个 PDE 分别为：0x00000007, 0x00001007, 0x00002007, 0x00003007, 0x00004007，符合 $(n * 4096) | PG_P | PG_USU | PG_RWW$ 的形式，PTE 正确初始化。经过 1024 * 1024 次循环，所有页表中所有的 PTE 都能够被赋为合适的值。

可以看出，.1 循环是对 PDE 进行初始化，.2 循环是对 PTE 进行初始化。

最后，对 pmtest6.asm 进行编译、装载、运行，运行结果如下图所示。



```
Bochs x86 emulator, http://bochs.sourceforge.net/

=====
WELCOME TO THE FREEDOS BETA4 RELEASE!

Type INSTALL to start the FreeDOS installation

If you need to create a partition on your hard disk for FreeDOS, you
will need to do that yourself. Use FDISK to create a partition, and
use FORMAT to make the partition writable by FreeDOS. You can run
In Protect Mode now. ^^Install Boot Floppy.

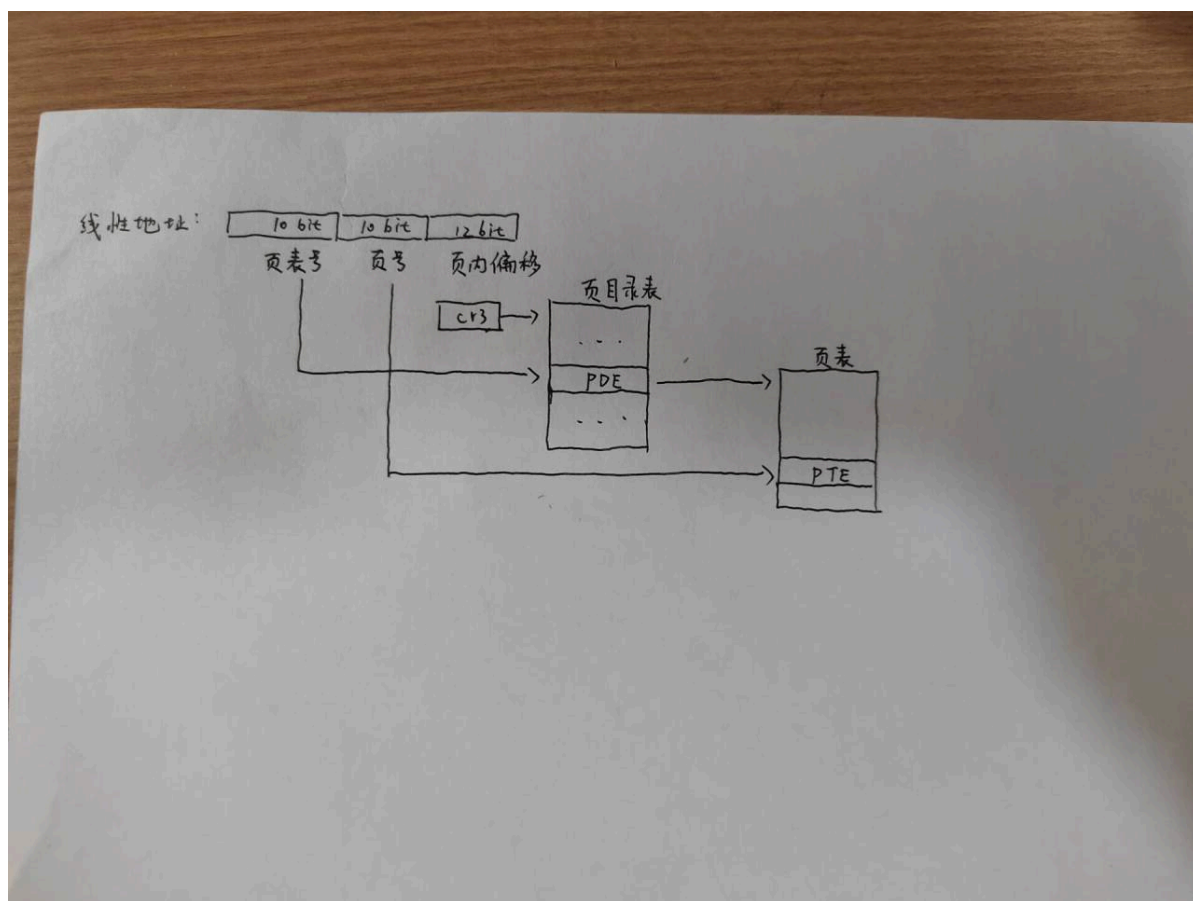
=====
A:\>B:\pmtest6.com
A:\>_

=====
ass=0x0600
ass=0x0601
ass=0x0101
ass=0x0680

IPS: 5.925M
A: B: NUM CAPS SCRL
00001054070i[ACPI] ] new IRQ line = 11
00001054690i[ACPI] ] new IRQ line = 9
00001054718i[ACPI] ] new PM base address: 0xb000
00001054732i[ACPI] ] new SM base address: 0xb100
00001054760i[PCI] ] setting SMRAM control register to 0x4a
00001218895i[CPU0] ] Enter to System Management Mode
00001218906i[CPU0] ] RSM: Resuming from System Management Mode
00001382962i[PCI] ] setting SMRAM control register to 0x0a
00001409578i[BIOS] ] MP table addr=0x000f9e70 MPC table addr=0x000f9da0 size=0xc8
00001411285i[BIOS] ] SMBIOS table addr=0x000f9e80
00001413374i[BIOS] ] ACPI tables: RSDP addr=0x000f9fa0 ACPI DATA addr=0x01ff0000 size=0xff8
00001416358i[BIOS] ] Firmware waking vector 0x1ff00cc
00001418612i[PCI] ] i440FX PMC write to PAM register 59 (TLB Flush)
00001419342i[BIOS] ] bios_table_cur_addr: 0x000f9fc4
00001546960i[VBIOS] ] VGABios $Id: vgabios.c,v 1.75 2011/10/15 14:07:21 vruppert Exp $
00001547031i[BXVGA] ] VBE known Display Interface b0c0
00001547063i[BXVGA] ] VBE known Display Interface b0c5
00001549988i[VBIOS] ] VBE Bios $Id: vbe.c,v 1.64 2011/07/19 18:25:05 vruppert Exp $
00014040067i[BIOS] ] Booting from 0000:7c00
00021618091i[BIOS] ] int13_harddisk: function 00, unmapped device for ELDL=80
00021622252i[BIOS] ] int13_harddisk: function 08, unmapped device for ELDL=80
```


掌握 PDE、PTE 的计算方法

PDE、PTE 的映射图如下：



- 为什么 PDE 初始化添加了一个 PageTblBase，而 PTE 初始化时候没有类似的基地址？

在本实验中，采取的映射关系为线性地址 = 物理地址，因此 0 号页表的 0 号页对应物理地址以 0 开始的页，0 号页表的 1 号页对应物理地址以 4096 开始的页（页内偏移 12 位，开始地址即 2 的 12 次方），以此类推。故第 0 个 PTE 存储的地址是 0，第 1 个 PTE 存储的地址是 4096，逐个叠加 4096。初始化时不可以添加类似 PageBase 的基地址，否则映射关系将不为线性地址 = 物理地址，而是物理地址 = 线性地址 + PageBase(也可以说，添加的基地址为 0)。

而 PDE 实际上并不直接反映线性地址和物理地址的映射关系，而是告诉系统页表的位置，页表从哪里开始存并不影响映射关系，只需不改变页表内容即可。因此，PDE 在初始化时需要添加 PageTblBase 的基地址。

同时，也可以看出第 0 个 PDE 存储 PageTblBase 指向第一个页表，第 1 个 PDE 存储 PageTblBase + 4096（一个页表就是一个页，一页 = 4096 字节），指向第 1 个页表，以此类推。

获取系统内存布局

涉及获取当前系统内存布局的代码为 `pmtest7.asm`，获取当前系统内存布局的步骤为：

1) 获取内存信息：代码主要利用了 `int 15h` 来得到内存信息。该中断调用能够将地址范围描述符结构 ARDS 写入缓冲区中，供后续使用。

`int 15h` 的参数为：

- `eax` 由 `ax` 的值决定功能号，获取内存信息需要将 `ax` 赋值为 `0E820h`。
- `ebx` 放置着后续值，第一次调用时 `ebx` 必须为 0。
- `es: di` 指向一个地址范围描述符结构 ARDS，BIOS 将会填充此结构。
- `ecx` `es: di` 所指向的地址范围描述符结构的大小（以字节为单位）。无论 `es: di` 所指向的结构如何，设置 BIOS 最多将会填充 `ecx` 个字节。通常情况下无论 `ecx` 为多大，BIOS 只填充 20 字节（有些 BIOS 忽略 `ecx` 的值总是填充 20 字节）。
- `edx` 值为 `0534D4150h` ('SMAP')。BIOS 将会使用此标志对调用者将要请求的系统映像信息进行校验，这些信息会被 BIOS 放置到 `es: di` 所指向的结构中。

结果存放于下列寄存器中：

- `CF` `CF = 0` 表示没有错误否则存在错误。
- `eax` `0534D4150h` ('SMAP')。
- `es: di` 返回的地址范围描述符结构指针和输入值相同。
- `ecx` BIOS 填充在地址范围描述符中的字节数量，被 BIOS 所返回的最小值是 20 字节。
- `ebx` 这里放置着为等到下一个地址描述符所需要的后续值。这个值的实际形式依赖于具体的 BIOS 的实现，调用者不必关心它的具体形式，只需在下次迭代时将其原封不动地放置到 `ebx` 中就可以通过它获取下一个地址范围描述符。如果它的值为 0 并且 `CF` 没有进位表示它是最后一个地址范围描述符。

准备好参数放入寄存器中，设置一个 256 字节长的缓冲区 MemChkBuf，将返回的 ARDS 信息保存在 MemChkBuf 中，根据 ebx 是否为 0 判断循环是否结束，_dwMCRNumber 保存循环次数（ARDS 的总个数），该部分关键代码如下：

```

; 得到内存数
mov     ebx, 0
mov     di, _MemChkBuf
.loop:
mov     eax, 0E820h
mov     ecx, 20
mov     edx, 0534D4150h
int     15h
jc      LABEL_MEM_CHK_FAIL
add     di, 20
inc     dword [_dwMCRNumber]
cmp     ebx, 0
jne     .loop
jmp     LABEL_MEM_CHK_OK

```

2) 打印输出内存布局：

每次循环读取一个 ARDS，分别读取出结构体的每个成员（BaseAddrLow, BaseAddrHigh, LengthLow, LengthHigh, Type），打印在屏幕上。其中调用了 DispInt 和 DispStr 函数，它们封装在 lib.inc 中。循

环的次数在 1) 中已经保存在 dwMCRNumber 指向的地址中。该部分代码如下：

```

mov     esi, MemChkBuf
mov     ecx, [dwMCRNumber];for(int i=0;i<[MCRNumber];i++)//每次得到一个ARDS
.loop:
mov     edx, 5 ; for(int j=0;j<5;j++) //每次得到一个ARDS中的成员
mov     edi, ARDStruct ; { //依次显示BaseAddrLow,BaseAddrHigh,LengthLow,
; LengthHigh,Type
.1:
push    dword [esi] ;
call    DispInt ; DispInt(MemChkBuf[j*4]); //显示一个成员
pop     eax ;
stosd   ; ARDStruct[j*4] = MemChkBuf[j*4];将eax的内容复制到edi的内存空间，复制四个字节，并将edi加4个字节
add     esi, 4 ;
dec     edx ;
cmp     edx, 0 ;
jnz     .1 ; }
call    DispReturn ; printf("\n");

```

3) 计算内存大小并显示：

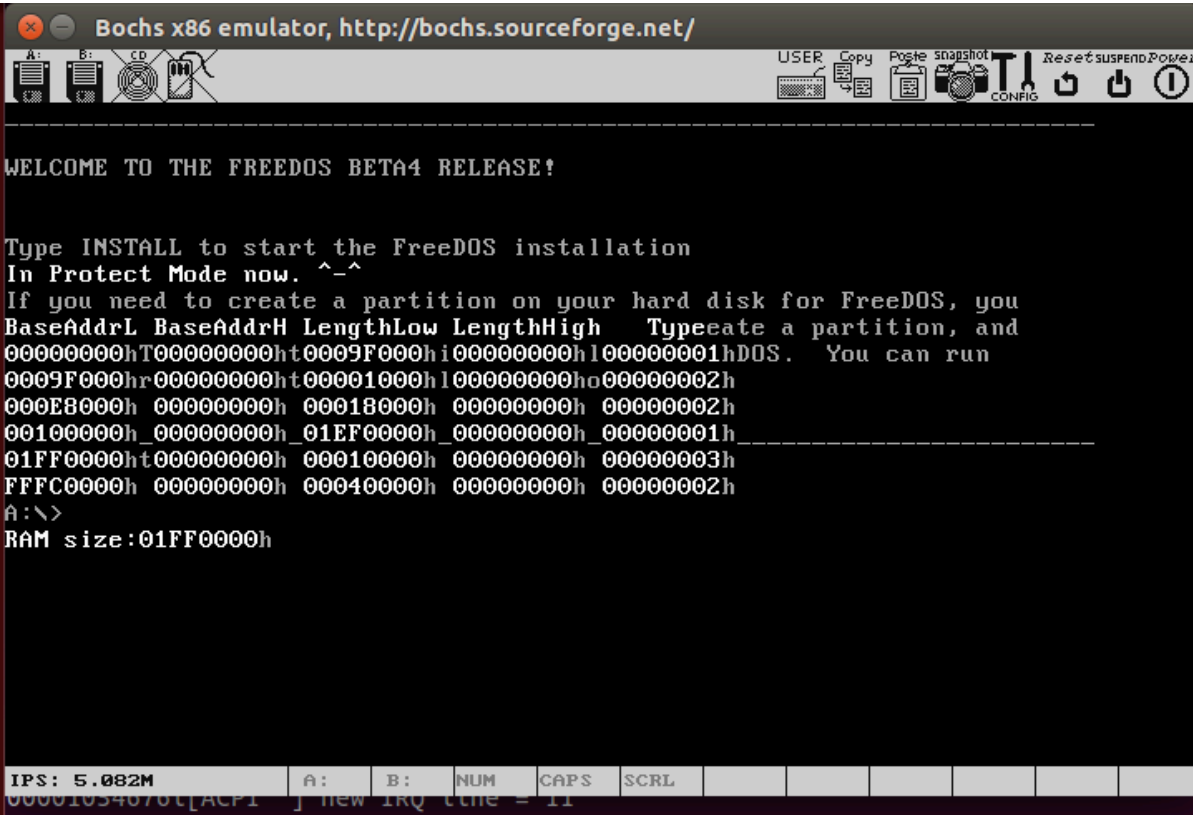
设置变量 dwMemSize（实模式下对应_dwMemSize）存储 RAM 大小。在循环读取 ARDS 时进行判断：若 BaseAddrLow + LengthLow > dwMemSize，则更新 dwMemSize。注意只有当 ARDS 的 Type 取值为 AddressRangeMemory（可被 os 使用的 RAM）才进行此判断。最后调用 DispInt 对 dwMemSize 进行输出显示。相关代码为：

```

cmp     dword [dwType], 1 ; if(Type == AddressRangeMemory)
jne     .2 ; {
mov     eax, [dwBaseAddrLow];
add     eax, [dwLengthLow];
cmp     eax, [dwMemSize] ; if(BaseAddrLow + LengthLow > MemSize)
jb     .2 ;
mov     [dwMemSize], eax ; MemSize = BaseAddrLow + LengthLow;
.2:

```

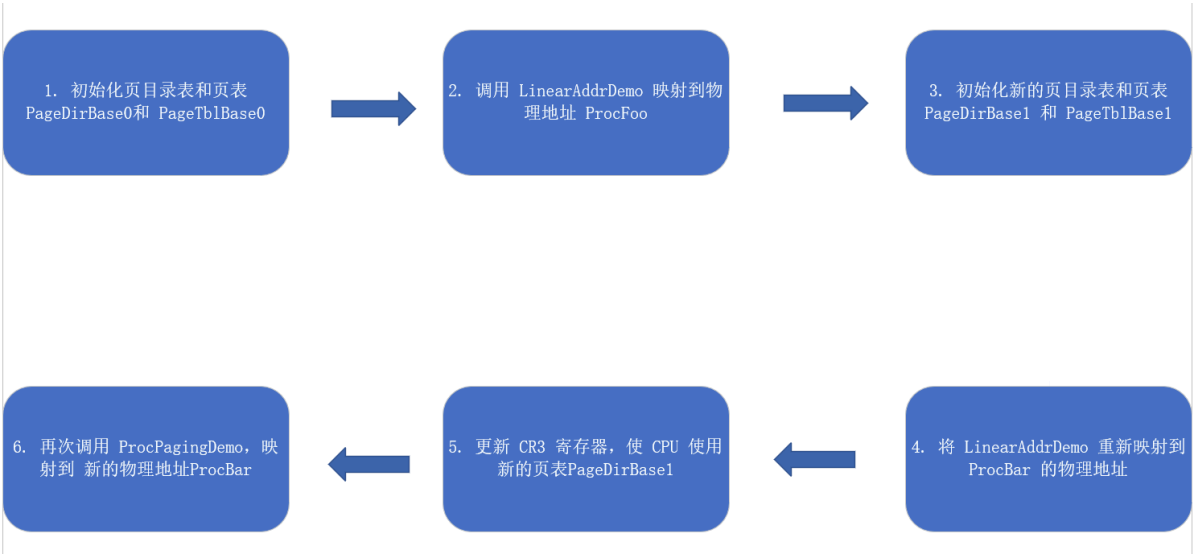
对 pmtest7.asm 进行编译、装载、运行，运行结果如下图所示。



可以看到操作系统所能使用的最大内存地址为 01FFFFFFh，此机器共拥有 32MB 的内存。

掌握内存地址映射关系的切换

pmtest8 展示了如何通过两个不同的页目录表和页表来实现分页机制的切换，通过改变地址映射关系，让两次执行同一个线性地址的模块时产生不同的结果。流程图如下：



- 1 初始化时，新增了一个变量页表数量 `PageTableNumber`，从而方便进行两个页表的初始化。

```
SetupPaging:
    mov [PageTableNumber], ecx ; 暂存页表个数
    ; 首先初始化页目录
    ; .....
    ; 再初始化所有页表
    mov eax, [PageTableNumber] ; 页表个数
    ; .....
```

- 2 一开始通过 `call SelectorFlatC:ProcPagingDemo` 调用 `ProcPagingDemo`，使用第一个页目录 `PageDirBase0` 将虚拟地址 `LinearAddrDemo`（`0x00401000`）映射到物理地址空间中的 `ProcFoo` 处，打印出红色的字符串 `F00`

```
PagingDemoProc:
OffsetPagingDemoProc    equ PagingDemoProc - $$
    mov eax, LinearAddrDemo ; 将线性地址 0x00401000 加载到 eax
    call    eax             ; 调用映射到物理地址的 ProcFoo 代码
    retf                   ; 远返回
LenPagingDemoAll        equ $ - PagingDemoProc
```

- 3 之后 `PSwitch` 函数进行页目录的切换：

- 重新初始化新的页目录 `PageDirBase1` 和页表 `PageTblBase1`

```
    ; 初始化页目录
    mov ax, SelectorFlatRW
    mov es, ax
    mov edi, PageDirBase1 ; 使用新的页目录 PageDirBase1
    xor eax, eax
    mov eax, PageTblBase1 | PG_P | PG_USU | PG_RWW
    mov ecx, [PageTableNumber]

.1:
    stosd
    add eax, 4096 ; 页表是连续的，设置映射
    loop .1

    ; 初始化所有页表
    mov eax, [PageTableNumber] ; 页表个数
    mov ebx, 1024 ; 每个页表有 1024 个 PTE
    mul ebx
    mov ecx, eax ; PTE 个数 = 页表个数 * 1024
    mov edi, PageTblBase1 ; 页表基地址为 PageTblBase1
    xor eax, eax
    mov eax, PG_P | PG_USU | PG_RWW

.2:
```

```

stosd
add eax, 4096      ; 每一页指向 4K 的空间
loop    .2

```

- 更新 `LinearAddrDemo` 的映射到 `ProcBar`：计算 `LinearAddrDemo` 页表项在页表中的偏移，加上页表 `PageDirBase1` 基址，最后将 `ProcBar` 的物理地址写入计算出的页表项地址中

```

; 更新 LinearAddrDemo 的映射到 ProcBar
mov eax, LinearAddrDemo
shr eax, 22      ; 取页目录索引
mov ebx, 4096    ; 计算偏移
mul ebx          ; 将 eax (页目录索引) 乘以 4096
mov ecx, eax
mov eax, LinearAddrDemo
shr eax, 12      ; 获取页表索引
and eax, 03FFh   ; 取低 10 位作为页表索引
mov ebx, 4
mul ebx          ; 计算页表项在页表中的偏移
add eax, ecx     ; 将页目录表的偏移和页表项的偏移相加，计算出在页表中的具体地址
add eax, PageTblBase1 ; 加上页表基址 PageTblBase1, 得到页表项的实际物理地址
mov dword [es:eax], ProcBar | PG_P | PG_USU | PG_RWW ; 将 LinearAddrDemo 映射到 ProcBar

```

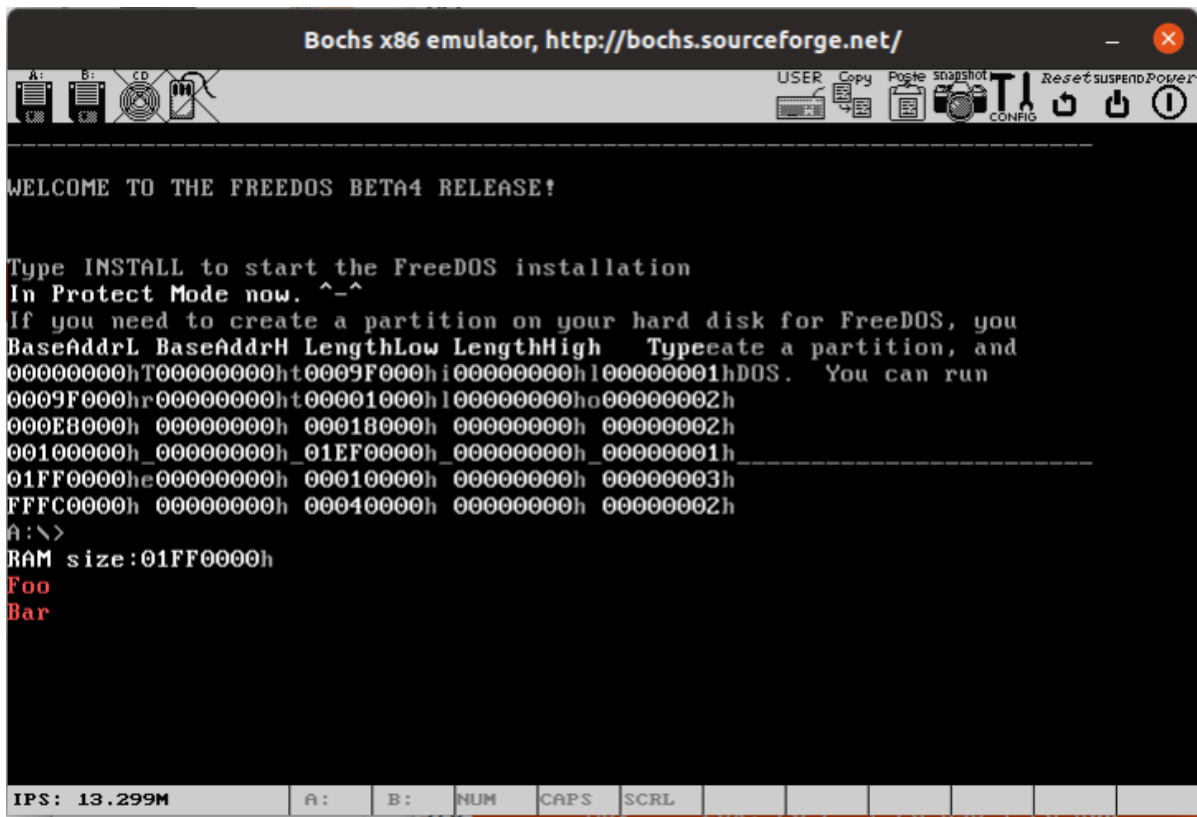
- 更新 `CR3` 切换页表，指向新的页目录 `PageDirBase1`，使得第二次调用 `ProcPagingDemo` 映射到物理地址 `ProcBar`，打印出红色的 `Bar`。

```

; 切换到新的页目录
mov eax, PageDirBase1
mov cr3, eax      ; 更新 CR3, 使其指向新的页目录

```

如图所示，实验结果符合预期，结果中打印出了红色的 `Foo` 和 `Bar`。分页机制使得应用程序不再直接使用物理地址，线性地址与物理地址之间的映射由操作系统负责，在一定程度上实现了内存保护。



基础题

任务一：模拟从虚拟地址到物理地址的计算过程

下面实现了 `GetPhysicalAddress` 函数，将一个虚拟地址转换为物理地址。通过页表映射机制，计算出虚拟地址对应的物理地址。

输入：

- `EAX`：包含要转换的虚拟地址。

输出：

- 成功时，`EAX` 返回对应的物理地址。
- 如果页目录项或页表项无效，`EAX` 返回 -1。

实现步骤：

- 1 设置段选择子：选择 Flat 段覆盖整个 32 位地址空间，并将该段选择子加载到 ES 寄存器中

- 2 计算页目录索引：从虚拟地址中取出高 10 位，加上 CR3（页目录基址寄存器），计算并读取出页目录项的位置。检查页目录项的有效性（P 位）。如果无效，则跳转到页错误处理流程，返回 -1

```
mov ebx, eax      ; 保存虚拟地址
shr eax, 22       ; 获取页目录索引 (高 10 位)
mov ecx, cr3      ; 获取页目录基址
shl eax, 2        ; 页目录项大小为 4 字节
add ecx, eax      ; 计算页目录项地址
mov eax, [es:ecx] ; 读取页目录项

test eax, 1       ; 检查页目录项是否存在 (P 位)
jz .PageFaultReturn ; 如果不存在, 跳转到页错误处理
and eax, 0FFFFFF0h ; 保留页表基址, 清除低 12 位
```

- 3 计算页表索引：同理从虚拟地址中取出中间 10 位，加上页表索引的偏移，读取出页表项的位置并检测

```
mov edx, ebx      ; 恢复原始虚拟地址
shr edx, 12       ; 获取页表索引 (中间 10 位)
and edx, 03FFh    ; 提取页表索引
shl edx, 2        ; 页表项大小为 4 字节
add eax, edx      ; 计算页表项地址
mov eax, [es:eax] ; 读取页表项

test eax, 1       ; 检查页表项是否存在 (P 位)
jz .PageFaultReturn ; 如果不存在, 跳转到页错误处理
and eax, 0FFFFFF0h ; 保留物理页基址, 清除低 12 位
```

- 4 计算物理地址：使用虚拟地址的低 12 位作为页内偏移，将其加到物理页基址，得到最终的物理地址。

```
and ebx, 0FFFh    ; 取出页内偏移 (低 12 位)
add eax, ebx      ; 加上页内偏移, 计算物理地址
```

完整代码如下，选取了两个示例虚拟地址，分别存在/不存在，并打印出字符串：

```
; =====
; 函数：获取虚拟地址的物理地址
; 输入：eax = 虚拟地址
; 输出：返回物理地址到 eax, 若地址不存在, 返回 -1
; =====
GetPhysicalAddress:
    ; 设置段选择子, 确保访问正确的内存段
    push ebx
    push ecx
```

```
push edx
```

```
mov bx, SelectorFlatRW ; 选择Flat段选择子, 覆盖整个32位地址空间  
mov es, bx ; 将选择子加载到es寄存器中
```

```
; 获取页目录索引
```

```
mov ebx, eax  
shr eax, 22  
mov ecx, cr3  
shl eax, 2  
add ecx, eax  
mov eax, [es:ecx]
```

```
test eax, 1  
jz .PageFaultReturn  
and eax, 0FFFFFF000h
```

```
; 获取页表索引
```

```
mov edx, ebx  
shr edx, 12  
and edx, 03FFh  
shl edx, 2  
add eax, edx  
mov eax, [es:eax]
```

```
test eax, 1  
jz .PageFaultReturn  
and eax, 0FFFFFF000h  
and ebx, 0FFFh  
add eax, ebx
```

```
pop edx  
pop ecx  
pop ebx  
ret
```

```
.PageFaultReturn:
```

```
mov eax, -1  
pop edx  
pop ecx  
pop ebx  
ret
```

```
; 到此停止
```

```
jmp SelectorCode16:0
```

测试代码:

```
LinearAddrDemo    equ 00401000h ; 示例虚拟地址
ErrorAddrDemo     equ 01401000h ; 示例虚拟地址
LABEL_DESC_FLAT_RW: Descriptor 0,          0ffffffh, DA_DRW|DA_LIMIT_4K
                  ; 0~4G
SelectorFlatRW     equ LABEL_DESC_FLAT_RW - LABEL_GDT
SuccessMessage:    db "Address Translation Successful!. ^-^", 0
PageFaultMessage: db "Page Fault!", 0
OffsetPageFaultMessage    equ PageFaultMessage - $$
OffsetSuccessMessage     equ SuccessMessage - $$

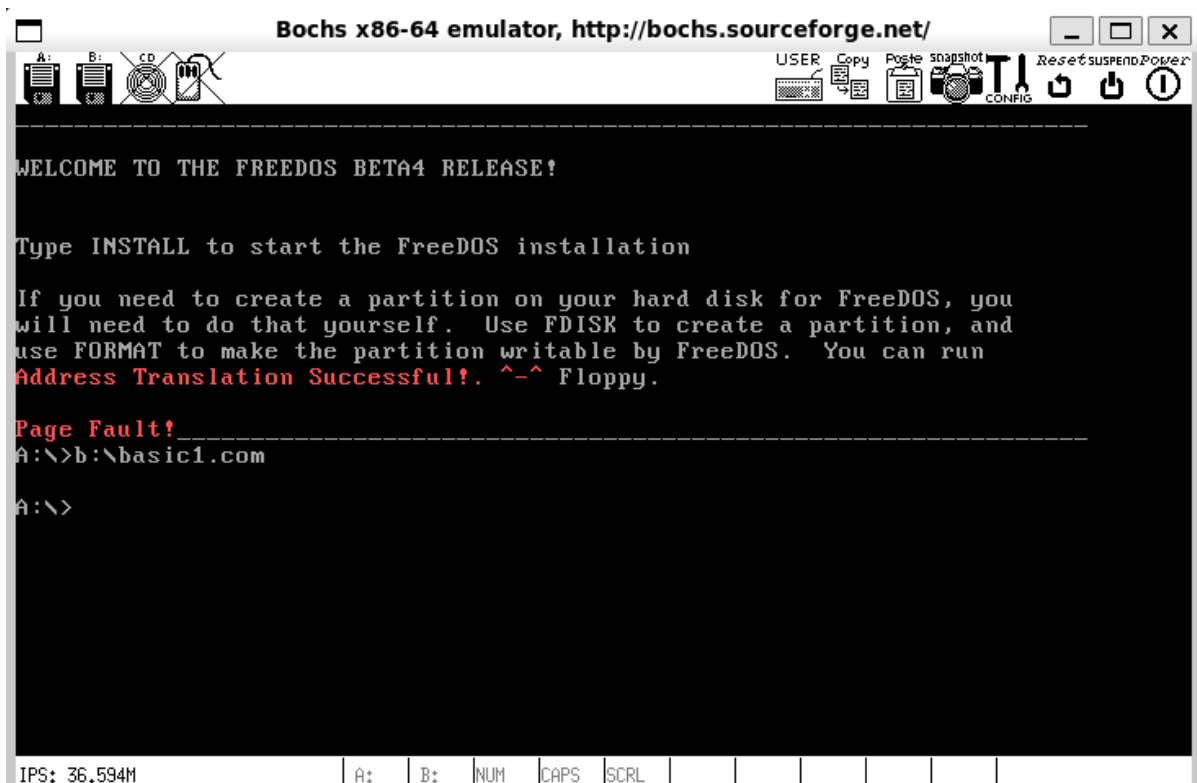
LABEL_SEG_CODE32:
    call    SetupPaging
    mov ax, SelectorData
    mov ds, ax          ; 数据段选择子
    mov ax, SelectorVideo
    mov gs, ax          ; 视频段选择子
    mov ax, SelectorStack
    mov ss, ax          ; 堆栈段选择子
    mov esp, TopOfStack

    mov eax, LinearAddrDemo
    call GetPhysicalAddress
    cmp eax, -1
    je .PageFault

    mov edi, (80 * 10 + 0) * 2      ; 第 10 行, 0 列
    mov esi, OffsetSuccessMessage   ; 设置显示的字符串
    call PrintString                ; 调用 PrintString 函数

    ; 测试错误地址, 模拟页面错误
    mov eax, ErrorAddrDemo
    call GetPhysicalAddress
    cmp eax, -1
    je .PageFault
    jmp SelectorCode16:0            ; 程序到此结束, 跳回实模式

.PageFault:
    ; 显示页错误信息在第 12 行显示
    mov esi, OffsetPageFaultMessage ; 设置要显示的页错误字符串
    add edi, 160
    call PrintString                ; 调用 PrintString 函数
    jmp SelectorCode16:0            ; 程序到此结束, 跳回实模式
```



具体示例计算过程：

对于 `LinearAddrDemo = 0x00401000`，转换过程如下：

- ① 二进制表示： `0000 0000 0100 0000 0001 0000 0000 0000`
- ② 页目录索引(PDE)：取高 10 位： `0000 0000 01`，即 `1`，页目录表项地址：
`0X00200000 (CR3) + 1 * 4 = 0X00202004`
- ③ 读取页目录表项 (PDE)：指向页表基址为 `0x00202007`，即 `0X00202000 | 7`，有效位为 `1`

```
<bochs:54> xp /4bx 0x00200004
[bochs]:
0x0000000000200004 <bogus+      0>:  0x07  0x20  0x20  0x00
```

- ④ 页目录表项 (PTE)：同理，取中间 10 位 `1`， `0X00202000 + 1 * 4 = 0x00202004`

- ⑤ 读取页表项 (PTE)：指向物理页 `0x00401007`，成功实现映射

```
<bochs:55> xp /4bx 0x00202004
[bochs]:
0x0000000000202004 <bogus+      0>:  0x07  0x10  0x40  0x00
```

对于 `ErrorAddrDemo = 0x01401000` 经过计算 PDE 为 `0x00200014`，该地址处值为 0

```
<bochs:52> xp /32bx 0x00200000
[bochs]:
0x000000000200000 <bogus+ 0>: 0x27 0x10 0x20 0x00 0x07 0x20 0x20 0x00
0x000000000200008 <bogus+ 8>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x000000000200010 <bogus+ 16>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x000000000200018 <bogus+ 24>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

任务二：补充 alloc_pages, free_pages 两个函数功能

本节通过 **bitmap** 数据结构实现 **alloc_pages** 和 **free_pages** 函数，从而能够方便进行内存的动态分配与释放。

alloc_pages:

- 输入: 请求的页数（存储在 **eax** 中）
- 输出: 成功返回页表基址，失败时返回 0。
- 功能: 用于分配指定数量的物理页，并设置相应的页表条目。如果请求的页数超出了可用的空闲页数，函数会返回失败标志。分配成功后，会记录分配的页号，并将页表中的页条目标记为有效。

```
; =====
; 函数: alloc_pages
; 功能: 为某个程序分配多个物理页
; 输入: eax = 请求的页数
; 输出: 分配成功返回页表基地址, 失败时 eax = 0
; =====
```

在遍历位图时，我们使用 **FoundPages** 数组存储每次查找到的空闲物理页号，将 **edi** 作为页号保存指针，方便成功时遍历数组保存的页号，将空闲页一同分配给程序，或者失败时不予分配，恢复页号保存指针为数组基址。

核心流程：

- 1 遍历位图（**BitmapBase**），使用 **bt** 指令判断位图中第 **ecx** 位是否被占用。当找到空闲页，物理页号会存入 **FoundPages** 数组。

```
alloc_pages:
    push esi
    push ebx
    push ecx
    push edi                ; 保存 edi 寄存器, 用于存储找到的页的索引

    mov cx, SelectorFlatRW ; 选择扁平段选择子, 覆盖整个32位地址空间
    mov es, cx              ; 将选择子加载到 es 寄存器中
```

```

    mov ecx, 0                ; 页表索引
    mov ebx, BitmapBase      ; 位图基址
    mov esi, 0                ; 已分配的物理页计数
    lea edi, [es:FoundPages] ; 将找到的页号保存到 FoundPages 数组中

find_free_pages:
    cmp ecx, MapLen           ; 检查是否到达位图末尾
    jz alloc_fail             ; 没找到足够的空闲页

    bt [es:ebx], ecx          ; 测试位图中第 ecx 位
    jc next_page              ; 如果该页被占用, 跳转到 next_page 页

    ; 找到一个空闲页, 记录页号
    mov [es:edi], ecx         ; 保存找到的页号到 FoundPages 数组中
    add edi, 4                ; 下一个页号位置
    add esi, 1                ; 更新已分配页计数
    cmp esi, eax              ; 检查是否分配了请求的页数 (eax = 请求的
    ; 页数)
    je alloc_success          ; 如果分配了足够的页数, 跳到成功

next_page:
    inc ecx                   ; 继续检查下一个页
    jmp find_free_pages       ; 回到循环开始

```

- 2 如果成功找到所需的页数, 则跳转到 `alloc_success` 函数, 打印出 `Successfully alloc ^-^` 字符串。遍历数组, 通过 `bts` 指令将这些页号对应的位标记为分配。

```

alloc_success:
    ; 成功分配了请求的页, 标记它们为已分配
    lea edi, [es:FoundPages] ; 恢复页号保存指针
    mov esi, eax              ; 请求的页数
    mov edx, PageTblBase     ; 页表基址

    mov esi, OffsetSucessMessage ; 源数据偏移
    mov edi, (80 * 10 + 0) * 2 ; 目的数据偏移。屏幕第 10 行, 第 0
    ; 列。
    call PrintString

set_allocated_pages:

    mov ecx, [es:edi]         ; 读取找到的页号
    bts dword [es:ebx], ecx   ; 将页号对应的位标记为分配

```

- 3 然后通过左移 12 位生成物理地址, 记录这些页的物理地址, 并设置有效位 P 位, 作为页表的 PTE 写入。


```

    shl ecx, 12                ; 每个页表条目指向物理页地址，左移12位生
成物理地址
    or ecx, 1                  ; 设置页表条目最低有效位（P位，表示该页有
效）
    mov [es:edx], ecx          ; 写入页表条目

    add edx, 4                 ; 递增页表基址，指向下一个页表条目
    add edi, 4                 ; 下一个页号
    dec esi                    ; 更新已分配页数
    jnz set_allocated_pages    ; 如果还没有完成，继续

```

- ④ 如果未能找到足够的空闲页，将进入 `alloc_fail` 函数，通过 `lea` 指令恢复页号保存指针，并打印出 `Page Fault T_T` 字符串。

```

alloc_fail:
    ; 分配失败，将之前分配的页释放
    mov esi, 0                ; 清空已分配计数
    lea edi, [es:FoundPages]  ; 恢复页号保存指针
    jmp alloc_done            ; 没有已分配页，直接退出

    mov esi, OffsetPageFaultMessage ; 源数据偏移
    mov edi, (80 * 12 + 0) * 2 ; 目的数据偏移。屏幕第 10 行，第 0
列。
    call PrintString

```

free_pages:

- 输入: 要释放的页数和页表的起始地址。
- 输出: 无（直接修改位图和页表条目）。
- 用于释放指定的页，清除页表条目并将位图中的对应位清零，以便这些页能够再次被分配使用。

```

; =====
; 函数: free_pages
; 功能: 释放程序使用的物理页
; 输入:
;     eax: 页表的起始地址。
;     ebx: 要释放的页数。
; 输出: 无
; =====

```

核心流程:

- ① 读取页表中的条目，逐个释放指定的页数。
- ② 将页表条目(PTE)清零，并在位图中清除相应的位。

- 更新剩余的页数，直到所有页都被释放。

```
mov edi, ebx          ; 需要释放的页数保存到 edi
mov ebx, BitmapBase   ; 位图基址

; 开始释放页
release_loop:

mov esi, [es:eax]      ; 读取页表条目 (PTE)
and dword [es:eax], 0   ; 清除页表条目中的有效位 (P位)

shr esi, 12            ; 提取页号 (物理地址的高20位)
btr dword [es:ebx], esi ; 清除位图中的对应位

; 处理下一个页
add eax, 4             ; 下一个页表条目 (PTE 占 4 字节)
dec edi                ; 更新剩余的页数
jnz release_loop       ; 如果还有页需要释放, 继续循环
```

下面是经过两个函数成功分配和释放后的结果

实验结果:

- 初始化时页表为空，并设置 `bitmap` 的 0、2、4、7 位为 1，表示已占用。

```
mov ebx, BitmapBase   ; 加载位图基址
bts dword [es:ebx], 0  ; 设置第 0 位
bts dword [es:ebx], 2  ; 设置第 2 位
bts dword [es:ebx], 4  ; 设置第 4 位
bts dword [es:ebx], 7  ; 设置第 7 位
```

查看 `bitmap` 地址处的前八位为 10010101，页表地址前四个字节均为 0。

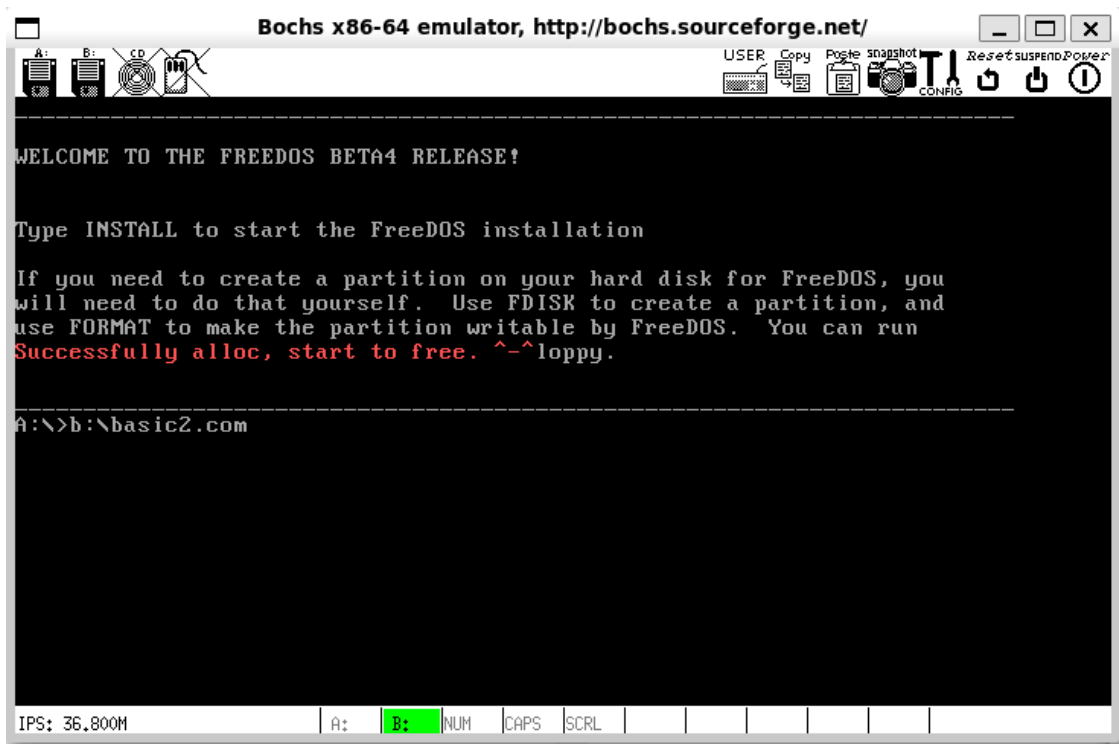
```
<bochs:8> xp /1bt 0x00200000
[bochs]:
0x0000000000200000 <bogus+ 0>: 10010101
<bochs:9> xp /4wx 0x00201000
[bochs]:
0x0000000000201000 <bogus+ 0>: 0x00000000 0x00000000 0x00000000 0x00000000
```

- 将 `eax` 设置为 4，请求 4 个页，调用 `alloc_pages` 函数，遍历位图时，找到 1、3、5、6 处为空闲页，分配这些页

```
mov eax, 4             ; 请求4个页
call alloc_pages
```

查看 `bitmap` 地址处的前八位为 11111111，页表地址前四个字节中间十位分别为 0x01、0x03、0x05、0x06，且标志位设为 1，符合预期，打印出字符串。

```
<bochs:5> xp /1bt 0x00200000
[bochs]:
0x0000000000200000 <bogus+ 0>: 11111111
<bochs:6> xp /4wx 0x00201000
[bochs]:
0x0000000000201000 <bogus+ 0>: 0x00001001 0x00003001 0x00005001 0x00006001
```



- 3 将 `ebx` 设置为 4，释放 4 个页，`eax` 为页表基址，调用 `free_pages` 函数，即刚才分配的页面全部释放

```
mov ebx, 4           ; 释放4个页
mov eax, PageTblBase ; 释放的页的起始地址
call free_pages
```

可见 `bitmap` 地址处的前八位为 10010101，页表地址前四个字节均为 0，回到最初情况。

```
(0) Magic breakpoint
Next at t=315637057
(0) [0x000000032c2d] 0018:000000000000007d (unk. ctxt): jmpf 0x0020:00000000 ; ea000000002000
<bochs:7> xp /1bt 0x00200000
[bochs]:
0x0000000000200000 <bogus+ 0>: 10010101
<bochs:8> xp /4wx 0x00201000
[bochs]:
0x0000000000201000 <bogus+ 0>: 0x00000000 0x00000000 0x00000000 0x00000000
```

一次分配的最大空间：

- `bitmap` 限制：位图的长度（`MapLen`）决定了能管理的总页数。 `MapLen` 为 256，说明最多可以管理 256 个物理页。

每个物理页的大小为 4KB，因此一次最大可以分配的内存空间为：

- 最大分配页数 = 位图的长度 × 每字节管理的页数 = `MapLen * 8`
- 每个物理页大小 = 4KB
- 最大内存空间 = 最大分配页数 × 4KB

那么系统可以管理的最大内存空间为：

$$256 \times 8 \times 4KB = 8192 \times 4KB = 32MB$$

这意味着在当前实验环境下，我们一次最多可以分配 **32MB** 的内存。

- 页表限制：

每个页表条目（PTE）对应一个物理页，并且页表条目的数量和结构同样限制了分配的内存空间。在 32 位系统中，每个页表最多能映射 1024 个页（每页 4KB），这意味着单个页表能映射的最大内存为：

$$1024 \times 4KB = 4MB$$

解决方案：使用多级页表结构，二级页表一次可以管理更多的页，映射的最大内存达到 4GB 级别。再增加位图的大小（MapLen），以达到最大内存管理。

进阶题：设计一个内存管理器

对于内存管理，下面分别从分块和分页进行讨论：

分块管理

操作系统可以采用空闲存储块链等数据结构，采取首次适应算法、最佳适应算法或者伙伴算法来管理空闲空间，具体情况如下

- ① **伙伴系统：**在伙伴系统中，可分配的内存空间是 2 的幂的大小的块（按题目要求，以页为最小单位进行分配，即最小为1024字节），伙伴是指一块被分为2个相同大小的块。
 - 当分配内存块时，不断以 **1/2** 形式分割空闲块找到满足请求大小的最小块。具体来说，将内存大小的一半和请求的大小进行比较，若请求的大小大于内存的一半，则分配整个块。否则，将该块拆分成两个相邻的块，如果请求的大小超过其中一个伙伴块的大小的一半，则将这个伙伴块分配给它，否则，该伙伴块再次被分成两半，如此往复，直至找到合适大小的块。
 - 当回收内存块时，只需检查伙伴是否空闲即可。找到这块内存的伙伴块，如果伙伴块没有被分配出去，则对这一对内存块进行合并，然后重复这个步骤，直到不能继续合并为止。

	0	128k	256k	512k	1024k
start	1024k				
A=70K	A	128	256	512	
B=35K	A	B 64	256	512	
C=80K	A	B 64	C 128	512	
A ends	128	B 64	C 128	512	
D=60K	128	B D	C 128	512	
B ends	128	64 D	C 128	512	
D ends	256		C 128	512	
C ends	512			512	
end	1024k				

2 首次适应算法：

- 首次适应算法从空闲分区表的第一个表目起查找该表，把最先能够满足要求的空闲区分配给作业。使用双向链表链接结构，在分配内存时，从链首开始顺序查找，直到找到一个大小能满足要求的空闲分区为止，然后再按照作业的大小，从该分区中划出一块内存空间分给请求者，余下的空闲分区仍停留在空闲链中。
- 当进程运行完毕释放内存，系统根据回收区的首址，从空闲区链表中找到相应的插入点
 - 回收区与插入点的前一个空闲分区F1相邻接，此时将两个分区合并
 - 回收区与插入点的后一个空闲分区F2相邻接，此时将两个分区合并
 - 回收区与插入点的前，后两个空闲分区相邻接，此时将三个分区合并
 - 回收区既不与F1相邻接，又不与F2相邻接，此时应为回收区单独建立一个新表项

分页管理

操作系统可以采用位示图的方式进行内存的分页管理，正如基础题实现 `alloc_pages` 和 `free_pages` 函数，从而能够方便进行内存的动态分配与释放。

- 开辟一块空间（往往是页的整数倍）来作为位示图。用二进制的1位来表示一个块的分配情况，1表示已分配，0表示未分配，每次分配内存时设置。分配策略可以算出要分配的页数，按照页为最小单位进行分配。每次从位示图的头部开始顺序查找，试探是否有足够的空闲页数用于分配，直至找到第一个符合空闲空间要求的起始页为止。

❖ 四、实验结果与总结

实验结果

- 1 掌握了分页机制的基本实现：通过调试代码，成功理解了页表和页目录表的初始化过程，以及如何通过分页机制进行内存地址映射。
- 2 获取了系统的内存布局：使用 `int 15h` 获取了系统的物理内存信息，并成功打印出内存段。
- 3 实现了内存分配与释放功能：通过编写 `alloc_pages` 和 `free_pages` 函数，成功实现了页分配与释放功能，并测试了最大分配空间。

思考题

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？

分页是指将地址空间分割成固定大小的页，并将这些页映射到物理内存中。而分段是将地址空间分割成不同大小的段，每个段是一个逻辑上完整的块，可以代表程序的不同部分，如代码段、数据段、堆栈段等。

- 区别： 分页主要是用于支持虚拟内存的实现，使得程序能够使用超过实际物理内存的空间。而分段用于更好地反映程序的结构，便于模块化和动态内存分配。

此外，分段主要针对的是逻辑地址转为线性地址，分页主要针对的是线性地址转为物理地址。分段依靠的是段表，而分页依靠的是页目录表和页表。

- 段页机制的搭配工作： 实验中，首先依靠 GDT 表和 LDT 表指定各段的起始位置、段长、段属性等，实现分段机制。寻址方式为： `段基址+段内偏移`，实现了逻辑地址向线性地址的转换。各段寄存器充当 GDT/LDT 表的索引，指向相应的段描述符以取得段基址。

接着，实验使用页目录表和页表来实现分页机制。线性地址被分为三部分，分别代表页表号、页号、内偏移，先找到页表，再找到对应物理页的基址，加上页内偏移，实现了线性地址向物理地址的转换。

综上，分页机制在分段机制的基础上实现，逻辑地址先通过分段机制转换为线性地址，再通过分页机制转换为物理地址，分段实现模块化，而分页用于实现虚拟内存，两者共同搭配工作。

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？

- PDE（Page Directory Entry）是指页目录表的表项，PTE（Page Table Entry）是指页表的表项。
- 初始化：例程将页目录表、页表分别当成一段，段基址用 `pageDirBase`、`pageTblBase` 记录，通过 `stosd` 指令来完成初始化。`stosd` 指令将 `eax` 的内容复制到 `esi: edi` 的内存空间，复制四个字节，并将 `edi` 加 4 个字节。以初始化 PDE 为例，页目录表共 1024 个表项，设置一个循环，循环执行 `stosd` 指令 1024 次，`esi: 0` 的初值即为 `pageDirBase: 0`。在本实验中，逻辑地址和线性地址是相等的关系，第 0 个 PDE 对应第 0 个页表，第 1 个 PDE 对应第 1 个页表，1 个页表占 1024×4 个字节，因此第 0 个 PDE 要填入的页表基址为 `pageTblBase`，第 1 个 PDE 要填入的页表基址为 `pageTblBase+4096`，设置页表属性为内存中存在的可读可写的用户级别页表，因此 `eax` 的初值应为 `PageTblBase | PG_P | PG_USU | PG_RWW`，每次循环加 4096。初始化 PTE 的方法同理，需要循环 1024×1024 次，`esi: 0` 的初值为 `pageTblBase: 0`，`eax` 的初值为 `PG_P | PG_USU | PG_RWW`，每次循环加 4096。最终得到初始化代码如下：

```
    ; 首先初始化页目录
    mov     ax, SelectorPageDir      ; 此段首地址为 PageDirBase
    mov     es, ax
    mov     ecx, 1024                ; 共 1K 个表项
    xor     edi, edi
    xor     eax, eax
    mov     eax, PageTblBase | PG_P | PG_USU | PG_RWW
.1:
    stosd                    ; 将eax的内容复制到edi的内存空间，复制四个字节，并将edi加4个字节
    add     eax, 4096          ; 为了简化，所有页表在内存中是连续的。
    loop    .1

    ; 再初始化所有页表（1K 个，4M 内存空间）
    mov     ax, SelectorPageTbl      ; 此段首地址为 PageTblBase
    mov     es, ax
    mov     ecx, 1024 * 1024         ; 共 1M 个页表项，也即有 1M 个页
    xor     edi, edi
    xor     eax, eax
    mov     eax, PG_P | PG_USU | PG_RWW
.2:
    stosd
    add     eax, 4096            ; 每一页指向 4K 的空间
    loop    .2

    mov     eax, PageDirBase
    mov     cr3, eax
    mov     eax, cr0
    or      eax, 80000000h
    mov     cr0, eax
    jmp     short .3
```

- CPU 访问 PDE、PTE，计算物理地址的方法：CPU 首先需要知道页目录表的基址，基址通常存在 cr3 寄存器中，cr3 寄存器的结构如下：



接着，线性地址被分为三部分，第一部分（高 10 位）作为页表号在页目录表中索引，访问到 PDE，进而得到页表地址，第二部分（第 12 到 21 位）作为页号在页表中索引，访问到 PTE，得到物理页首地址，将这个首地址加上线性地址低 12 位（页内偏移）便得到了物理地址。

开启分页机制之后，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 GDT 表中存放的是线性地址。GDT 表用于实现分段机制，分段机制的寻址方式为 $\text{线性地址} = \text{段基址} + \text{段内偏移}$ ，而段基址正存储在 GDT 表中，因此对应的是线性地址。
- 在 PDE、PTE 中存放的是物理地址。PDE、PTE 均用于实现分页机制，分页机制的寻址方式为 $\text{物理地址} = \text{页基址} + \text{页内偏移}$ ，而 PTE 存储的即为页基址（还有页属性），故为物理地址。同理，PDE 存储的是对应页表的基址，应该也为物理地址，才能正确找到页表。

为什么 PageTblBase 初始值为 2M+4K？能不能比这个值小？

因为实验中设置页目录表起始位置为 2M，而页目录表占 4K（1024 个表项），且页目录表与页表在内存中相邻，所以是 2M+4K。

为了让 PageTblBase 减小，可以：

- 减小页目录表项，但是同时也会减小寻址的大小。
- 前移页目录表的起始位置（小于 2M）。

怎么读取本机的实际物理内存信息？

可以通过 `int 15h` 指令，并设置功能号 `ax` 为 `0E820h` 来读取本机的实际物理内存信息。该方法可以得到内存的大小以及对不同内存段的一些描述，这些描述将被保存在一个缓冲区中。每次调用 `int 15h` 将会返回一个地址范围描述符 `ARDS`，可以通过设置一个循环的方式，多次调用 `int 15h` 中断，当 `ebx` 的值为 0 并且 `CF` 没有进位表示此时返回的是最后一个地址范围描述符，退出循环。每次得到的内存信息连续写入一块缓冲区形成一个结构体数组，最后进行处理。将 `ARDS` 结构体的内存打印出来即可，内存 `RAM` 的大小 = `max(ARDS.BaseAddrLow + ARDS.LengthLow)`。

如何进行地址映射与切换？

- 地址映射：地址映射指虚拟内存系统中将虚拟地址转换为物理地址。
 - 1 通过移位操作得到虚拟地址的页目录索引（`PDE index`）和页表索引（`PTE index`）。
 - 2 使用页目录索引从页目录表（`PDT`）中找到对应的页目录项（`PDE`）
 - 3 页目录项存储了页表的基址，结合页表索引，从相应的页表中找到页表项（`PTE`）。
 - 4 页表项存储了页帧地址，加上页内偏移，可以得到虚拟地址映射到的最终物理地址。
- 地址切换：地址切换是通过切换页目录表和页表来改变虚拟地址到物理地址的映射关系
 - 1 初始化新的页目录表（`PDT`）和页表（`PT`）
 - 2 通过更新页目录表中的条目（`PDE`）和页表中的条目（`PTE`），改变虚拟地址到物理地址的映射
 - 3 修改 `CR3` 寄存器，将其指向新的页目录表。

假设虚拟地址 `0x00403000`，并且页目录表基址存储在 `CR3` 中：

- 1 获取页目录基址：
`CR3` 寄存器的值为 `0x00002000`，指向当前页目录表。
- 2 页目录索引计算：
通过 `0x00403000 >> 22` 得到页目录索引 `1`（`0x1`）。

3 页目录项地址：

页目录项的地址为 $CR3 + (PDE\ Index * 4) = 0x00002000 + (0x1 * 4) = 0x00002004$ 。

4 加载页目录项（PDE）：

读取地址 $0x00002004$ 的内容，假设其值为 $0x00003000 | 1$ （其中 $0x00003000$ 是页表基址， 1 表示页存在）。

5 页表索引计算：

通过 $(0x00403000 \gg 12) \& 0x3FF$ 得到页表索引 3 （ $0x3$ ）。

6 页表项地址：

页表项的地址为 $页表基址 + (PTE\ Index * 4) = 0x00003000 + (0x3 * 4) = 0x0000300C$ 。

7 加载页表项（PTE）：

读取地址 $0x0000300C$ 的内容，假设其值为 $0x00004000 | 1$ （其中 $0x00004000$ 是物理页基址， 1 表示页存在）。

8 计算物理地址：

物理地址 = 物理页基址 + 页内偏移。虚拟地址的低 12 位为 $0x000$ ，因此物理地址为 $0x00004000 + 0x000 = 0x00004000$ 。

如何实现 alloc_pages, free_pages

内存分配使用位图法。每个物理页都由位图中的一位表示，0 表示空闲，1 表示占用。这种方式可以高效管理物理内存的分配和释放。

alloc_pages: 循环遍历位图中的每一位，检测该位是否为 0（表示该页空闲），是则记录当前页号。如果找到的页数等于请求的页数，标记页为已分配（**bts** 指令），并在页表中设置 PTE 完成虚拟地址对物理地址的映射。

free_pages: 遍历指定的页表项(PTE)，从页表中读取物理地址，将页表项清除（**and dword [es:eax], 0**），或者可以只清除有效位。再将物理地址右移 12 位提取物理页号，把对应位图中的位清除（**btr** 指令）。

实验改进意见

- 1 在本实验中，使用的分页机制是两级页表。我们可以继续思考如果面临更大的寻址空间要求，如何设计三级页表的分页机制。在操作系统理论课中，我们还讨论了反向页表的概念，我们可以尝试了解反向页表的具体实现，增进对分页机制的理解。
- 2 除了页表和页目录表之外，我们还可以探索 TLB 快表是如何与页表结合使用的，查询资料 TLB 快表在 x86 架构中的实现，更深入地理解 TLB 快表是如何让页表的查询更快的。

❖ 五、各人实验贡献与体会

1 程序：

- 承担任务：独立完成实验，负责实验的原理分析、实验步骤的前四问、思考题的前四问。
- 个人体会：本次实验共分为三部分：页表、页目录表的初始化和基本分页机制的认识、实际物理内存信息的读取、地址映射的切换。在操作系统理论课上，我们通过多级页表、虚拟内存等概念来认识分页机制，而本次实验通过代码实践的方式，让我了解了分页机制是如何在实际的操作系统中发挥作用的。同时，实验还加深了我对分页机制和分段机制配合工作的理解，为继续深入操作系统的实验打下了基础。

2 黄东威：

- 承担任务：独立完成实验，对于多份代码调试研究了分页机制的实现，与王浚杰一同探讨了内存管理的函数实现以及空闲页的分配与回收。
- 个人体会：本次实验起初有点困难，因为虽然在上学期学习过段页式，但是对线性地址转换成物理地址等等操作还是在很多细节上出现了问题。比如在代码编写中，对地址的转换不熟悉，忘记了需要右移 22 位后再左移 4 位，导致 debug 时间很长，所以需要在实验前进一步熟悉理论。但更多的是，在实验过程中的不断试错，也让我逐步清晰和掌握了对于 PDE 和 PTE、物理内存信息的读取以及内存的映射切换等相关知识。希望在后续实验中能进一步巩固汇编语言的知识，同时加深对操作系统的理解！

3 周业营：

- **承担任务：** 参照资料完成了大部分实验步骤，认真阅读了分页机制相关内容，学习 PDE、PTE 的计算方法，熟悉如何获取当前系统内存布局的方法，思考 PDE、PTE 初始化和计算物理地址等问题。
- **个人体会：** 通过阅读教材和调试代码，我对分页机制基本方法与思路有了更深刻的了解，特别是有 PDE、PTE 的分页机制的工作方式。对于获取系统内存布局可以利用中断 15h 获取内存信息的中断使用方法。这次实验与上学期学习的 OS 理论课联系较为紧密，深刻接触了分页模式下，利用线性地址得到物理地址的过程，对操作系统中的内存管理部分分页存储机制进行了巩固复习，同时也学习到了很多操作系统新知识。同时，我对使用不同的页表实现地址映射的偏移部分代码十分感兴趣，课下对这部分机制进行了进一步的学习和思考。

4 王浚杰：

- **承担任务：** 独立完成大部分实验，负责内存地址映射关系的切换、基础题以及与之相关的后两个思考题的实验报告。设计并实现了 `GetPhysicalAddress` 函数，从虚拟地址到物理地址的转换；设计并实现了内存管理中的 `alloc_pages` 和 `free_pages` 两个核心函数，使用位图法进行空闲页的分配与回收。深入分析映射过程关键两个数据结构页目录表项（PDE）和页表项（PTE），以及逐级映射虚拟地址到物理地址过程。
- **个人体会：** 通过本次实验，我深入理解了内存管理的机制和分页的基本工作原理，熟悉了页目录表、页表的结构。通过动手实践，我明白了如何通过计算地址索引、移位操作实现地址转换，以及如何通过页表和页目录表实现虚拟地址到物理地址的映射。页表切换操作使我认识到多级页表的优点，如它在管理大规模内存时的高效性。内存分配和释放功能的实现让我学会了如何利用位图管理内存，理解了操作系统中对内存的高效管理策略。