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	2022.10.3
实验名称	分页机制	实验周次	第三周
姓名	学号	专业	班级
李心杨	2020302181022	信息安全	2
王宇骥	2020302181008	信息安全	2
林锟扬	2020302181032	信息安全	2
郑炳捷	2020302181024	信息安全	2

1 实验目的及实验内容

1.1 实验目的

掌握分页机制。

1.2 实验内容

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。(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？)
3. 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)？)
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。(画出流程图)
6. 基础题：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) 完善分页管理功能，补充 alloc_pages, free_pages 两个函数功能，试试你能一次分配的最大空间有多大，如果超出了有什么办法解决呢?
7. 进阶题（选做)：设计一个内存管理器。提示，均按照页为最小单位进行分配、对于空闲空间管理可采用位图法或其他方法进行管理，分配策略不限。

2 实验环境及实验步骤

2.1 实验环境

- Ubuntu 16.04.1;
- VMWare Workstation 16 player;
- bochs 2.7。

2.2 实验步骤

1. 调试代码 pmtest6.asm，设置恰当的断点查看循环，总结分页机制的基本方法与思路。
2. 运行代码 pmtest7.asm，熟悉得到内存布局信息的方法。
3. 调试 pmtest8.asm，体会分页，掌握内存地址映射关系的切换。
4. 编写函数，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实验过程。
5. 选用伙伴策略，设计一个内存管理器。

3 实验过程分析

3.1 调试 pmtest6.asm

3.1.1 阅读代码

先观察 pmtest6.asm。代码大致分为 gdt 段、数据段、全局堆栈段、进入保护模式前的 16 位代码段、32 位代码段、退出保护模式后的 16 位代码段。代码运行的流程图大致如图1所示。

很多部分是与 pmtest2.asm 是相同的，这里不再赘述。下面阐释 SetupPaging 关键代码细节。

阅读下文中的代码段。第 6-9 行是让段寄存器 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。设置 eax 值为 PageTblBase | PG_P | PG_USU | PG_RWW。查阅资料，stosd 需要寄存器 edi 配合使用。每执行一次 stosd，就将 eax 中的内容复制到 [edi] 中。注意 edi 能实现自动增减（由 DF 标志位决定是自增还是自减，前面的代码已经执行了 cld 指令设置 DF=0，此时 edi 为自增状态）。在代码的 12-15 行，向存放在连续的物理地址中的 PDE 写入 eax 值，每循环 1 次 eax 增加 4096（这表示所有页表在内存中是连续的），共循环 1024 次（对应代码的第 8 行）。

代码的 17-27 行是在初始化所有页表中的 PTE。共 1024 个页表，每个页表有 1024 个 PTE，所以共需要循环 1024^2 次。因为在前面我们令页表在内存中连续存储，所以这

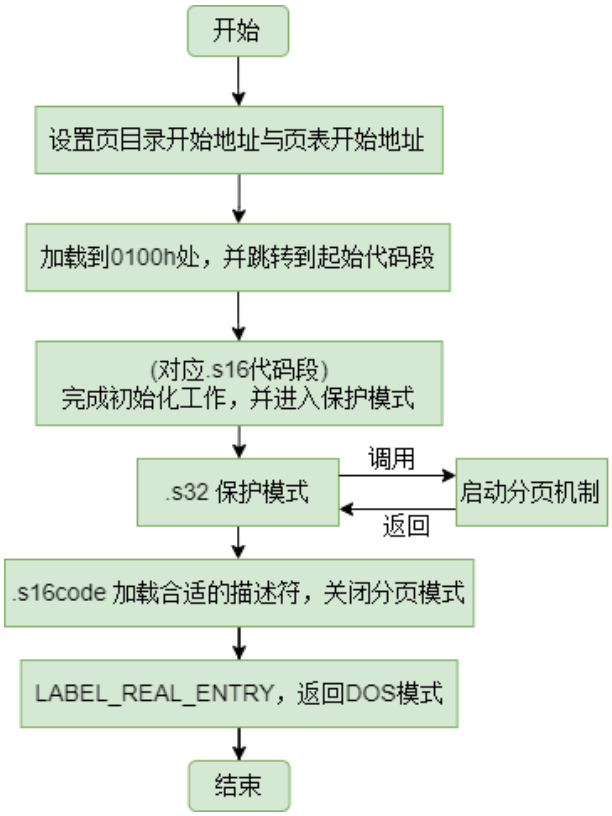

图 1: pmtest6 流程图

里直接使用 edi 进行自增即可，无需跳转。循环前，令 es:edi 指向第 0 个页表的第 0 个 PTE 的地址，eax 赋值为 PG_P | PG_USU | PG_RWW。这说明第 0 个页表的第 0 个 PTE 指示的页首地址为 0，所以在这段程序中，线性地址被映射到同样的物理地址中。

完成 PDE、PTE 初始化后，修改 CR3 的值，并且将 CR0 最高位置为 1。此时分页机制就被启动了。

```
1     ; 启动分页机制
2  SetupPaging:
3     ; 为简化处理，所有线性地址对应相等的物理地址.
4
5     ; 首先初始化页目录
6     mov ax, SelectorPageDir ; 此段首地址为 PageDirBase
7     mov es, ax
8     mov ecx,pmtest6 1024          ; 共 1K 个表项
9     xor edi, edi
10    xor eax, eax
11    mov eax, PageTblBase | PG_P  | PG_USU | PG_RWW
12 .1:
13    stosd
14    add eax, 4096         ; 为了简化，所有页表在内存中是连续的.
15    loop     .1
16
17    ; 再初始化所有页表 (1K 个，4M 内存空间)
18    mov ax, SelectorPageTbl ; 此段首地址为 PageTblBase
19    mov es, ax
20    mov ecx, 1024 * 1024     ; 共 1M 个页表项，也即有 1M 个页
```

```
21      xor edi, edi
22      xor eax, eax
23      mov eax, PG_P  | PG_USU | PG_RWW
24  .2:
25      stosd
26      add eax, 4096         ; 每一页指向 4K 的空间
27      loop    .2
28
29      mov eax, PageDirBase
30      mov cr3, eax
31      mov eax, cr0
32      or  eax, 80000000h
33      mov cr0, eax
34      jmp short .3
35  .3:
36      nop
37
38      ret
39  ; 分页机制启动完毕
```

该程序运行时，内存结构如下图所示。

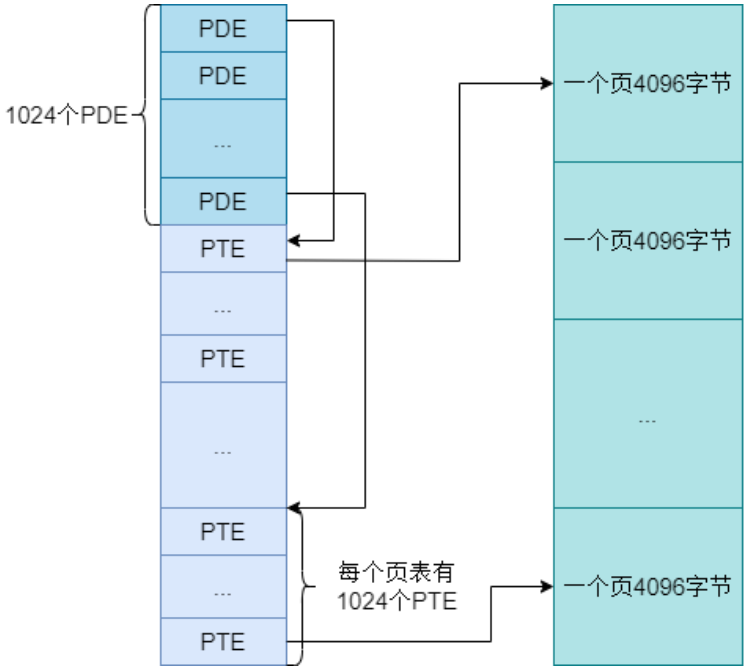

图 2: pmtest6 内存展示

3.1.2 调试代码

- 调试第一个循环。将.s1 修改如下：

```
.1:
        xchg bx,bx      ;调试代码
        stosd
        add     eax, 4096               ; 为了简化，所有页表在内存中是连续的.
        loop    .1
```

图 3: debug 第一个循环

编译 pmtest6.asm 并装载。在 bochsrc 中加入 magic_break: enabled=1 一行，开启调试模式。可以看到在 DOS 状态下运行 pmtest6.com，程序停在了调试点，等待调试。

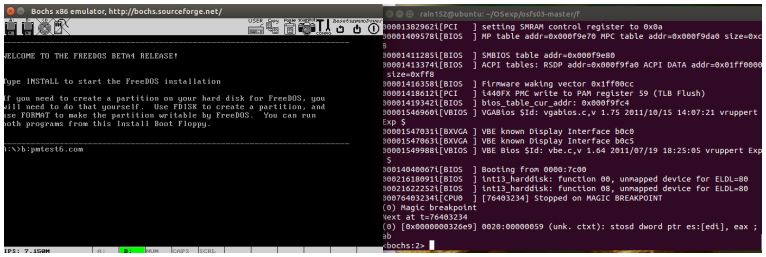

图 4: 调试模式

执行 4 层循环，查看 0x00200000 地址处的值。

```
<bochs:15> xp /32bx 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
<bochs:16>
```

可以看到 4 个 PDE 的内容分别是 0x00201007, 0x00202007, 0x00203007, 0x00204007。这和我们在阅读代码中的理解是一致的。经过这 1024 层循环，所有的 PDE 都能够被赋为合适的值。此处不再继续调试。

- 调试第二个循环。将.s2 修改如下：

```
.2:
        xchg bx,bx ;调试代码
        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
```

图 5: debug 第二个循环

继续在 DOS 中进行调试。停在调试点后，我们先查看 PDE 的赋值情况。可以看到 PDE 完成了赋值。

```
<bochs:6> xp /32bx 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>:    0x07    0x50    0x20    0x00    0x07    0x60    0x20    0x00
0x00200018 <bogus+      24>:    0x07    0x70    0x20    0x00    0x07    0x80    0x20    0x00
<bochs:7>
```

图 6: PDE 完成赋值

执行 4 层循环后，查看 0x201000 地址处的值（阅读代码时我们知道 PTE 从 0x201000 处开始存放）。

```
<bochs:8> xp /32bx 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
<bochs:9>
```

可以看到前 4 个 PTE 的值分别为 0x00000007, 0x00001007, 0x00002007, 0x00002007。对应的页基址 0，1，2，3。PTE 赋值也与我们的理解相符。经过 1024*1024 层循环后，所有的 PTE 完成赋值，进而实现将线性地址映射到对应的物理地址。

• 运行程序。取消断点，直接运行 pmtest6.com。运行结果如图7所说。打印的结果与 pmtest2 是相同的。说明程序正确运行。

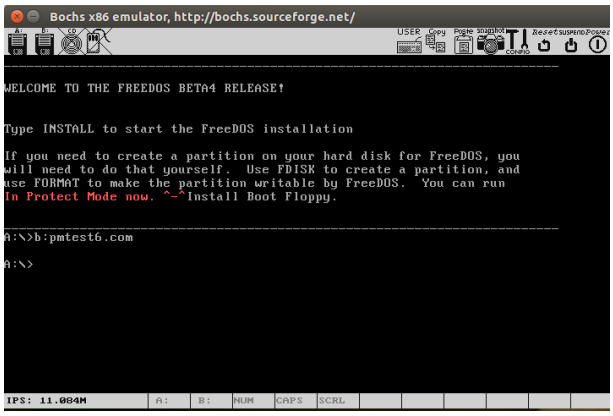

图 7: pmtest6 运行结果

3.2 运行 pmtest7.asm

3.2.1 阅读代码

pmtest7.asm 代码在 pmtest6.asm 的基础上增加了关于内存计算的一些操作。其代码流程图如图8。

• 计算内存。相关代码如下。这段代码主要利用了 int 15h 来得到内存信息。更具体地说，通过该调用能够将内存信息即地址范围描述符结构 ARDS 写入缓冲区中，供后续输出函数使用。关于 ARDS 结构以及 int 15h 的细节这里不过多涉及。执行这段代码后，ARDS 信息保存在 MemChkBuf 起始的缓冲区中，_dwMCRNumber 保存循环次数（也就是 ARDS 的总个数）。

```
; 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
LABEL_MEM_CHK_FAIL:
    mov dword [_dwMCRNumber], 0
LABEL_MEM_CHK_OK:
```

- 打印内存信息。该部分代码如下。每次循环读取一个 ARDS，从中抽取出对应的信息，打印在屏幕上。其中调用了 DispInt 和 DispStr 函数，它们封装在 lib.inc 中。循环的次数在计算内存时已经保存在 dwMCRNumber 指向的地址中。

```
DispMemSize:
    push    esi
    push    edi
    push    ecx

    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,
.1:                     ;              LengthHigh,Type
    push    dword [esi] ;
    call    DispInt     ;    DispInt(MemChkBuf[j*4]); //显示一个成员
    pop eax             ;
    stosd               ;    ARDStruct[j*4] = MemChkBuf[j*4];
    add esi, 4          ;
    dec edx             ;
    cmp edx, 0          ;
    jnz .1              ;  }
    call    DispReturn  ;  printf("\n");
    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:                     ;  }
    loop    .loop       ;}
                        ;
    call    DispReturn  ;printf("\n");
    push    szRAMSize   ;
```

```
33      call    DispStr         ;printf("RAM size:");
34      add esp, 4              ;
35                      ;
36      push    dword [dwMemSize]  ;
37      call    DispInt         ;DispInt(MemSize);
38      add esp, 4              ;
39
40      pop ecx
41      pop edi
42      pop esi
43      ret
```

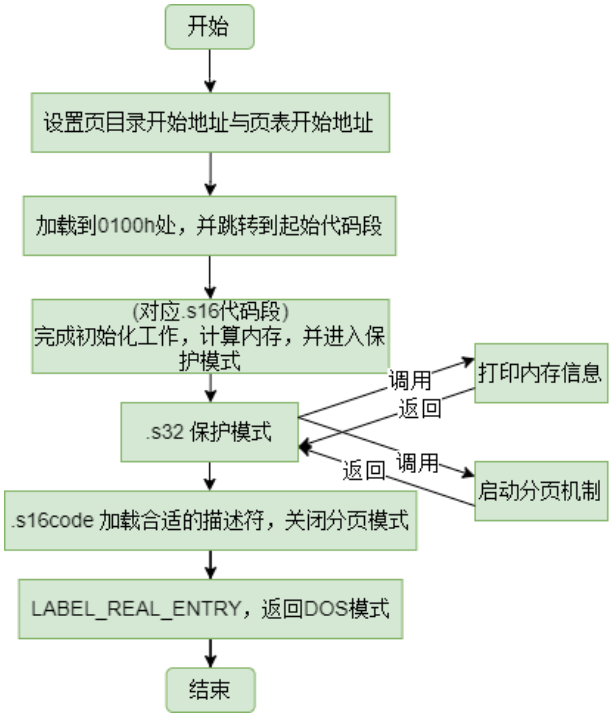

图 8: pmtest7 流程图

3.2.2 运行程序

对 pmtest7.asm 编译、装载。在 DOS 中运行该程序，运行结果如下图所示。

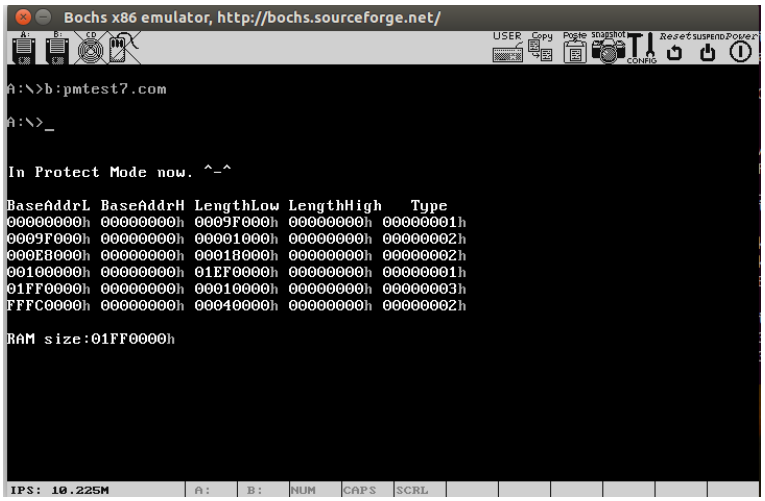

图 9: pmtest7 运行结果

给出这六段内存的解释。

表 1: 内存情况

内存段	属性	是否可被 OS 使用
00000000h∼0009EFFFh	AddressRangeEntry	可
0009F000h∼000E7FFFh	AddressRangeReserved	不可
000E8000h∼000FFFFFh	AddressRangeReserved	不可
00100000h∼01FEFFFFh	AddressRangeEntry	可
01FF0000h∼FFFBFFFFh	未定义	不可
FFFC0000h∼FFFFFFFFh	AddressRangeReserved	不可

由此可得，操作系统可以使用的最大的内存地址为 O1FEFFFFh，机器最大内存约为 32MB。此时我们发现，32MB 的内存只需要 32KB 的页表即可实现访问。这意味着在初始化 PDE 和 PTE 时，可以控制它们有合适的数量，这样可以有更多的空间来存储其他信息。

3.2.3 pmtest7 关于 PDE、PTE 分配改进

基于前面的分析，在分配 PDT 空间时，可以精确计算出最合适的数量。pmtest7 的 SetupPaging 部分的改进如下。[dwMemSize] 存放机器总内存，所以执行 3-5 行代码后，eax 中存放页表的个数（为了严谨，后面几行代码检测余数，若不为 0 则需增加一个页表）。后面 PTE 的个数也需要进行相应修改。如此改进后，页表所占的空间小得多，程序所需的内存空间也会相应减小。

```
1 SetupPaging:
2     ; 根据内存大小计算应初始化多少PDE以及多少页表
3     xor edx, edx
```

```
    mov eax, [dwMemSize]
    mov ebx, 400000h    ; 400000h = 4M = 4096 * 1024, 一个页表对应的内存大小
    div ebx
    mov ecx, eax    ; 此时 ecx 为页表的个数, 也即 PDE 应该的个数
    test    edx, edx
    jz  .no_remainder
    inc ecx     ; 如果余数不为 0 就需增加一个页表
.no_remainder:
    push ecx        ; 暂存页表个数
```

3.3 运行 pmtest8.asm

3.3.1 阅读代码

在前面的代码中已经实现了根据可利用内存合理分页，但是并没有利用到分页机制的优势，于是在 pmtest8 中通过改变地址映射关系来让在两次执行同一个线性地址的模块时产生不同的结果。其代码流程如图10。

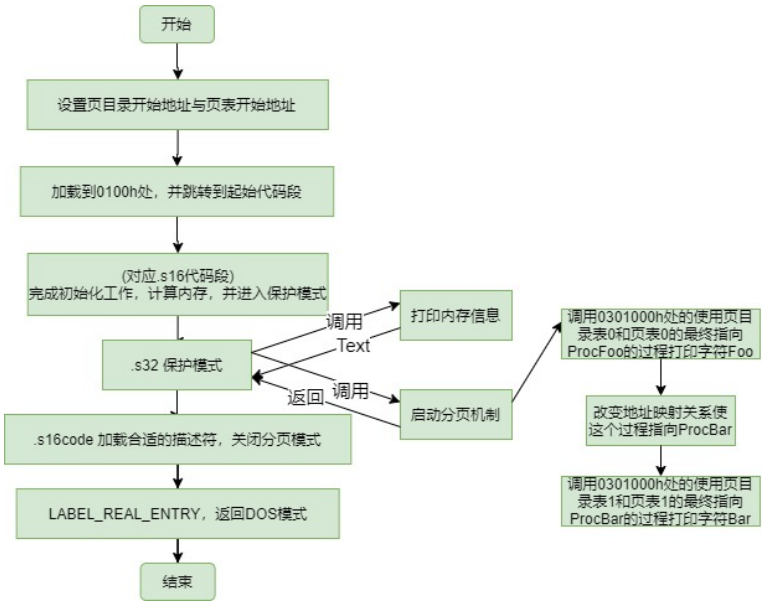

图 10: pmtest8 流程图

除了在 GDT 表中使用两个不同的描述符和选择子来指向同一个 Flat 段来表示可执行或者读写属性以外，其相对于 pmtest7 的改变如下：

• 分页机制设置：新增了一个变量页表数量 PageTableNumber，方便不同的页表的初始化。修改后的 SetupPaging 如下：

```
SetupPaging:
    ...
.no_remainder:
    mov [PageTableNumber], ecx  ; 暂存页表个数

    ; 为简化处理, 所有线性地址对应相等的物理地址. 并且不考虑内存空洞.
```

```
7
8       ; 首先初始化页目录
9       mov ax, SelectorFlatRW
10      mov es, ax
11      mov edi, PageDirBase0    ; 此段首地址为 PageDirBase0
12      xor eax, eax
13      mov eax, PageTblBase0 | PG_P  | PG_USU | PG_RWW
14  .1:
15      stosd
16      add eax, 4096           ; 为了简化, 所有页表在内存中是连续的.
17      loop    .1
18
19      ; 再初始化所有页表
20      mov eax, [PageTableNumber]  ; 页表个数
21      mov ebx, 1024           ; 每个页表 1024 个 PTE
22      mul ebx
23      mov ecx, eax            ; PTE个数 = 页表个数 * 1024
24      mov edi, PageTblBase0    ; 此段首地址为 PageTblBase0
25      xor eax, eax
26      mov eax, PG_P  | PG_USU | PG_RWW
27  .2:
28      stosd
29      add eax, 4096           ; 每一页指向 4K 的空间
30      loop    .2
31
32      mov eax, PageDirBase0
33      mov cr3, eax
34      ...
```

在初始化页目录和页表的过程中，ES 的值始终为 SelectorFlatRW，而为了简化处理，所有线性地址对应相等的物理地址，这样只需要将想取的物理地址赋值给 EDI 即可，在上述代码中的 stosd 等指令便利用了这一点。特别地，上面代码中的 PageDirBase0 和 PageTblBase0 是第 0 个页目录表和页表的基地址。

在流程图中提到的被两次调用的过程为 ProcPagingDemo，其调用指向打印字符的过程的 LinearAddrDemo。相比于 pmtest7，新建了函数 PagingDemo 来存放封装所有和分页有关的内容：

```
1   PagingDemo:
2       mov ax, cs
3       mov ds, ax
4       mov ax, SelectorFlatRW
5       mov es, ax
6
7       push    LenFoo
8       push    OffsetFoo
9       push    ProcFoo
10      call    MemCpy
11      add esp, 12
12
13      push    LenBar
14      push    OffsetBar
15      push    ProcBar
16      call    MemCpy
```

```
17      add esp, 12
18
19      push    LenPagingDemoAll
20      push    OffsetPagingDemoProc
21      push    ProcPagingDemo
22      call    MemCpy
23      add esp, 12
24
25      mov ax, SelectorData
26      mov ds, ax              ; 数据段选择子
27      mov es, ax
28
29      call    SetupPaging     ; 启动分页
30
31      call    SelectorFlatC:ProcPagingDemo
32      call    PSwitch         ; 切换页目录，改变地址映射关系
33      call    SelectorFlatC:ProcPagingDemo
34
35      ret
```

其中的 MemCpy 类似 C 语言的 memcpy 函数，作用是将用到的三个过程到指定的内存地址中。

• 改变地址映射关系：增加 PSwitch 函数来改变地址映射关系，这个函数与设置分页机制的 SetupPaging 类似，只是增加了改变线性地址 LinearAddrDemo 对应物理地址的语句。与 SetupPaging 中不同的是，页目录表和页表的基地址发生变化，变更为第 1 个：PageDirBase1 和 PageTblBase1（cr3 的值发生了变化），从而在这个函数被调用之后能够使线性地址指向的物理地址从 ProcFoo 变更为 ProcBar。

```
1   PSwitch:
2       ; 初始化页目录
3       mov ax, SelectorFlatRW
4       mov es, ax
5       mov edi, PageDirBase1   ; 此段首地址为 PageDirBase1
6       xor eax, eax
7       mov eax, PageTblBase1 | PG_P  | PG_USU | PG_RWW
8       mov ecx, [PageTableNumber]
9   .1:
10      stosd
11      add eax, 4096           ; 为了简化，所有页表在内存中是连续的.
12      loop    .1
13
14      ; 再初始化所有页表
15      mov eax, [PageTableNumber]  ; 页表个数
16      mov ebx, 1024           ; 每个页表 1024 个 PTE
17      mul ebx
18      mov ecx, eax            ; PTE个数 = 页表个数 * 1024
19      mov edi, PageTblBase1   ; 此段首地址为 PageTblBase1
20      xor eax, eax
21      mov eax, PG_P  | PG_USU | PG_RWW
22  .2:
23      stosd
24      add eax, 4096           ; 每一页指向 4K 的空间
25      loop    .2
```

```
26
27      ; 在此假设内存是大于 8M 的
28      mov eax, LinearAddrDemo
29      shr eax, 22
30      mov ebx, 4096
31      mul ebx
32      mov ecx, eax
33      mov eax, LinearAddrDemo
34      shr eax, 12
35      and eax, 03FFh  ; 1111111111b (10 bits)
36      mov ebx, 4
37      mul ebx
38      add eax, ecx
39      add eax, PageTblBase1
40      mov dword [es:eax], ProcBar | PG_P | PG_USU | PG_RWW
41
42      mov eax, PageDirBase1
43      mov cr3, eax
44      jmp short .3
45  .3:
46      nop
47
48      ret
```

3.3.2 运行程序

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。

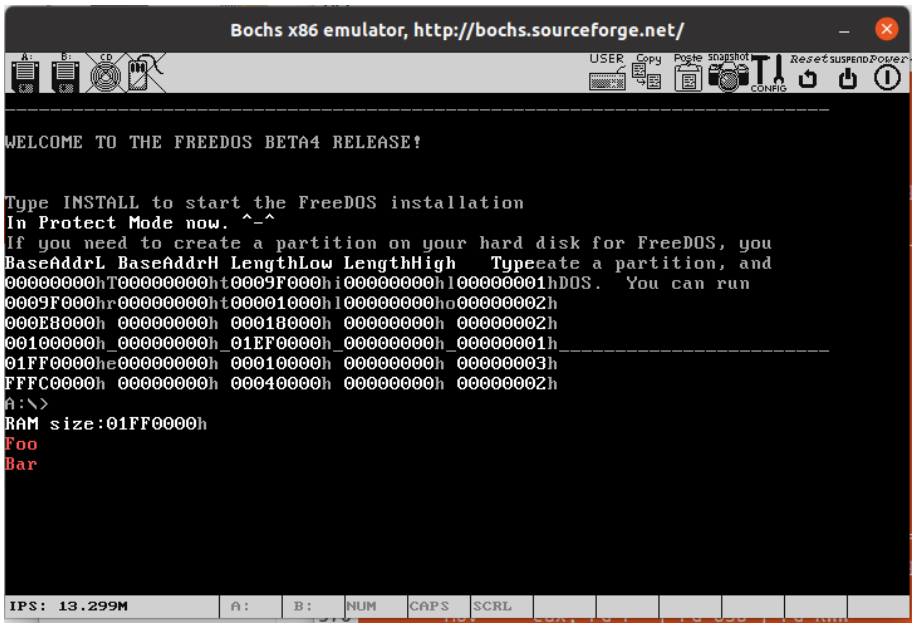

图 11: pmtest8 运行结果

3.4 基础题

3.4.1 任务一：模拟从虚拟地址到物理地址的计算过程

模拟 CPU 计算物理地址的过程，逐步找到 PDE、PTE，最终用 PTE 对应的物理页地址与页内偏移合并即可。

实现的两个函数功能如下：

1. TestL2P 传入 eax 线性地址显示线性地址转化为物理地址的过程。

2. GetL2P 返回 eax 对应的物理地址在 ebx 中，错误信息放在 ecx 中。

其中 1 用于调试，2 用于计算。完整的代码在 libm.inc 中，代码较长，不便粘贴。具体的计算过程在前面的流程描述中已经详细介绍过，此处不再赘述。此处仅描述实现中的部分细节：

为了在计算过程中显示进度，TestL2P 中使用了自己定义的显示函数，这些函数需要 DATA 段来支撑。本人在实现中使用了用堆栈实时保存段寄存器值的方法。以下面获取页表地址的代码为例：

```
mov ebx, cr3
shr eax, 22
and eax, 03FFh
shl eax, 2 ; 每个页目录项占 4 个字节
push ds
mov edx, SelectorFlatRW
mov ds, edx
mov ebx, [ebx + eax]
pop ds
push ebx
call DispInt
pop ebx
ShowStr strh32
```

虽然在本例中，为了方便，我们将线性地址基本设置成与物理地址一一对应。而且课本在查找页表和修改页表时也没有关闭。但我们知道，我们模拟时使用的页表、页目录中存的地址，都是页的物理地址而不是虚拟地址。在分页开启的情况下模拟地址转换的过程，在逻辑上有点“自相矛盾”。因此，在程序开头，出于谨慎，我们关闭了 cr0 的分页开关。并在后面进行了恢复。

```
...
mov eax, cr0
and eax, 7FFFFFFFh
mov cr0, eax
...
mov eax, cr0
or eax, 80000000h
mov cr0, eax
...
```

由于在编程时，我们还没有讨论关闭和开启分页后的 jmp 和 nop 的意义，因此我们的代码中除了分页时模仿随书源码加上了 jmp 以外，开关分页时都没有做这一步。这是一个小小的疏忽。

3.4.2 任务二：汇编实现 alloc_pages 和 free_pages

这两个函数的实现较为复杂。完整的代码和接口的定义在 manage_page.inc 中。此处我们用汇编实现的，只是个相对简单、理想化的过程，用于理解和模拟内存分配的过程。为了辅助实现这一逻辑，我们定义了 BitMap 用于标记内存中页的占用情况。由于我们还没有学习操作系统内核中如何对这一结构进行维护，因此，在代码中，我们对这一结构采取了简单的处理方式：

```
_BitMap:
  ; 观察可以发现，我们实际使用的内存都在 0x00100000 内
  ; 不妨直接假定 1M 内的内存都是被分配的
  ; 1M ~ 2M 的内存是可分配的
  times 32 db 0FFh ; 32 * 8 * 4K = 1M
  times 32 db 00h
BitMap equ _BitMap - $$
```

对于 BitMap 的读写，我们封装在了 bitmap.inc 的三个函数中：CheckVaild, SetVaild, ResetVaild。

此外，由于我们在分配页目录时，并没有完整的为 1024 个 PDE 分配页表地址。所以我们还需要一个函数 alloc_PDE 来初始化一个空的页目录项。由于前面对 BitMap 的简单处理，它的范围并不包含我们用来分配页表的地址。因此，我们在 DATA 段定义了额外的变量，来记录最后分配上的页表地址。通过这个变量来为新建的页目录赋值。

```
_LastPDE: dd 0
LastPDE equ _LastPDE - $$
```

alloc_pages 的流程如下：

1. 计算 eax 开始的连续 ecx 字节占用了多少个页。
2. 在 BitMap 中寻找连续的空闲页。如果没有找到，退出函数，将报错保存在 ebx。
3. 如果找到，就调用 link_page，将从那里开始的连续物理页与虚拟地址 eax 开始的连续虚拟页对应起来。

free_page 则较为简单，直接找到 eax 对应的连续页，调用 stop_page 将其页表项一一删除即可。

3.4.3 调试笔记

由于是第一次完整的重构代码，所以注意到了一些之前没有注意到的细节：

1. 从保护模式退回实模式的那个 16 位段，它对于 cs 的作用类似于 Normal 描述符对数据段的作用。它的界限值必须设置为 0ffffh。不然在跳回的偏移地址较高时会出错，导致 dos 重启。

2. 根据官方文档，在 nasm 汇编中，想要使用转义字符，需要使用反引号。

3. 从页目录获取页表基址后，要记得去掉属性位。

3.4.4 运行结果

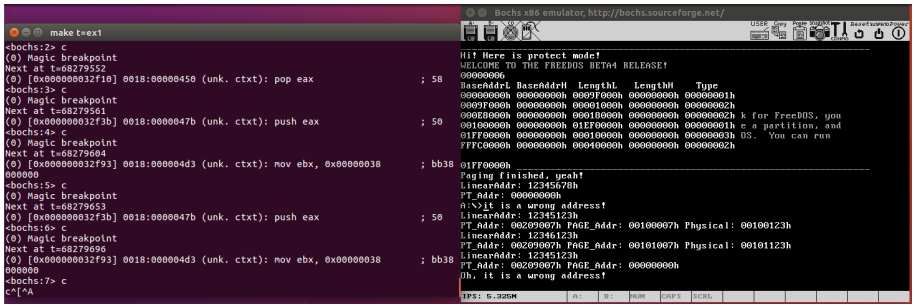

图 12: 成功运行的截图

除了内存显示的部分，截图下面的部分是下面代码的输出结果：

```
    mov eax, 012345678h
    call TestL2P
    mov eax, 012345000h
    mov ecx, 2000h
    call alloc_pages
    test ebx, ebx
    jz .d1
    ShowStr strDEBUG
.d1:
    mov eax, 012345123h
    call TestL2P
    mov eax, 012346123h
    call TestL2P
    mov eax, 012345000h
    mov ecx, 2000h
    call free_pages
    mov eax, 012345123h
    call TestL2P
```

第一次查询 0x12345678 时，由于页表和页目录都不存在，这个虚拟地址并没有对应值，程序提示找不到地址。第二、三次查询前，我们为 0x12345000 0x12346000 的虚拟地址分配了物理内存，因此再次查询时，顺利计算出了物理地址。第三次查询前，我们释放了这里的物理内存，因此又无法找到物理内存了。

3.5 设计一个内存管理器

3.5.1 原理阐述

Buddy 算法是一种广泛使用的内存管理算法，其特点是实现简单、效率高，但容易产生碎片。在伙伴系统中，可分配的内存空间是 2 的幂的大小的单个块。当接受到第一个请求时，如果请求的大小大于初始请求的一半，则分配整个块。否则，将该块拆分成两个绑定且相邻的“伙伴”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”块直接分配给它，否则，其中一个“伙伴”块再次被分成两半。重复这个步骤，一直到找到满足请求大小的最小块，并分配出去。

当回收内存块时，找到这块内存的“伙伴”块，如果“伙伴”块没有被分配出去，则这一对内存块可以进行合并。合并之后，可以继续判断合并后的内存块的“伙伴”块是否可以被合并。重复这个步骤，直到不能继续合并为止。

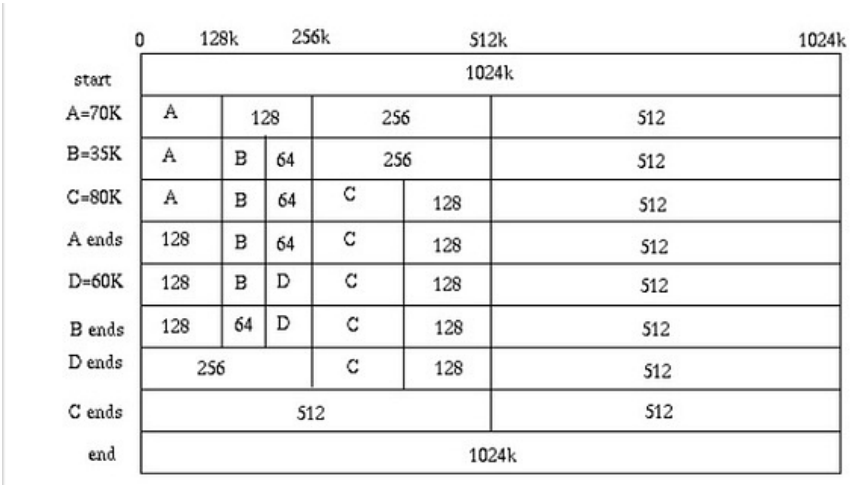

图 13: Buddy 算法原理

图13描述了伙伴算法分配和回收的过程。

3.5.2 设计概述

在3.4的基础上，我们选择采用伙伴（Buddy）算法进行物理内存的管理，实现了一个物理内存分配器。分配器采用了 C 语言实现，现阶段还无法整合到系统中，仅测试了接口的正确性。

我们使用一系列位图来管理物理内存空间，如图14所示。位图中的一个二进制位对应了一块内存的使用情况，0 表示这块内存未被分配。共有 10 个位图，每个位图所标明的内存粒度不同：位图的下标 i 表示大小为 2^i 字节的页框的使用情况，我们将这个下表值叫做 $order$。例如，假设页面大小为 4KiB，并且 free_area[2].maps 的第 0x30 位为 0，则表示物理内存上 0xc0000~0xc3fff 的位置是可用的。初始状态下，除了最大粒度的位图（$order = 9$）中的可用部分为 0，不可用部分（为外设、BIOS 等保留的地址）和其余位图均为 1。

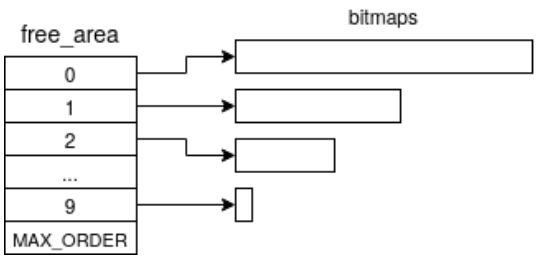

图 14: free_area 数组结构

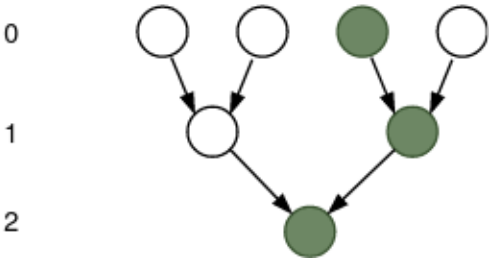

图 15: 二叉树角度看 bitmaps

所有的 bitmaps 还可以以二叉树的视角来看待，如图15所示，其中实心表示位图对应二进制位值为 1，空心表示值为 0。每次分配时，先在请求的 *order* 层进行寻找，如果找到为 0 的二进制位，则直接将其设为 1，表示这部分物理内存已经被分配出去，并返回地址；如果没有找到，则需要将更大块的内存分解，并将其中一块用于满足请求，而将另一块在位图对应位置上设为 0，表示可以被分配。

3.5.3 代码实现

完整代码参见GitHub。

基本数据结构和宏的声明如下：

```
typedef struct {
  unsigned int *map;
  int size; /* size of bitmap array */
} free_area_t;
free_area_t free_area[MAX_ORDER]; /* array of free space bitmap */
```

其中，map 所指向的空间需要额外进行分配。

如3.5.2中所述，外层的 while 循环从小块内存向大块内存搜索可用空间，corder 表示现在正在搜索的 order 大小。一旦搜索到可用空间，则一层层回到 *order* 处，用设置位图上值的方式将过程中的大内存块分解。分配页面时，使用 clz() 函数计算位图数组高位中前导 0 个数，并存储在 pos 中。通过位运算 pos << order << LPAGE_SIZE 即可得到位图某位对应的物理内存地址基址。

```
PM alloc_pages(unsigned int order) {
  unsigned long pos = 0; /* position of available bit in bitmap*/
  unsigned int corder = order; /* current order to search */
  PM paddr = NULL;
  while(true) {
    pos = 0;
    for(int i = 0; i < free_area[corder].size; i++) {
      if (free_area[corder].map[i] == 0) {
        pos += 32;
      } else {
        pos += clz(free_area[corder].map[i]);
      }
    }

```

```
    if (pos == free_area[corder].size * 32) {
      /* no available space in request order */
      /* find in higher order and split */
      ++corder;
      continue;
    }

    SET_BIT(free_area[corder].map, pos);
    for(--corder; corder > order; --corder) {
      pos <<= 1;
      CLEAR_BIT(free_area[corder].map, pos ^ 1); /* set buddy to free */
    }
    return (PM)(pos << LPAGE_SIZE << order);
  }
}
```

在回收时，需要查看当前回收的内存的“伙伴”是否为空闲状态，如果是，则将这两个内存块合并，并继续到上一层查看是否可以合并。通过 pos^1，即可得到“伙伴”块的位图位置。

```
void free_pages(PM paddr, unsigned int order) {
  unsigned int corder = order;
  unsigned long pos = (unsigned long) paddr >> LPAGE_SIZE >> order;
  while(corder < MAX_ORDER) {
    if (TEST_BIT(free_area[corder].map, pos ^ 1)) {
      CLEAR_BIT(free_area[corder].map, pos);
      /* buddy is occupied, stop */
      break;
    }
    /* buddy is free */
    SET_BIT(free_area[corder].map, pos);
    ++corder; pos >>= 1;
  }
}
```

4 实验结果总结

4.1 分页机制

分页存储的逻辑是：将用户程序（进程）的逻辑地址空间分成若干个大小相等的页（4KiB）并编号，同时将内存的物理地址也分成若干个块或页框（4KiB）并编号。再通过页表实现两者的一一映射关系，从而实现将进程的各个页离散地存储在内存的任一物理块中，有效提高了内存的利用率。

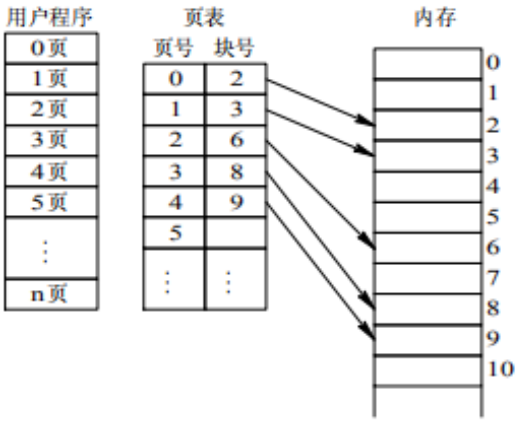

图 16: 页表

分页机制建立在分段机制的基础上。分页的开启与关闭由 CR0 寄存器的最高位 PG 标识。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。地址转换关系如图17所示。

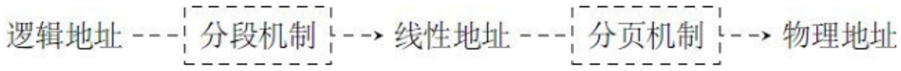

图 17: 地址关系

在线性地址转化成物理地址的过程中，分页机制使用页目录表和页表两级转换。其中页目录表的物理地址需要通过 CR3 寄存器也就是 PDBR(Page Table Base Register) 来确定。它的高 20 位将是页目录表首地址的高 20 位。

图 18: CR3 的结构

转换地址对应的示意图如图19所示。

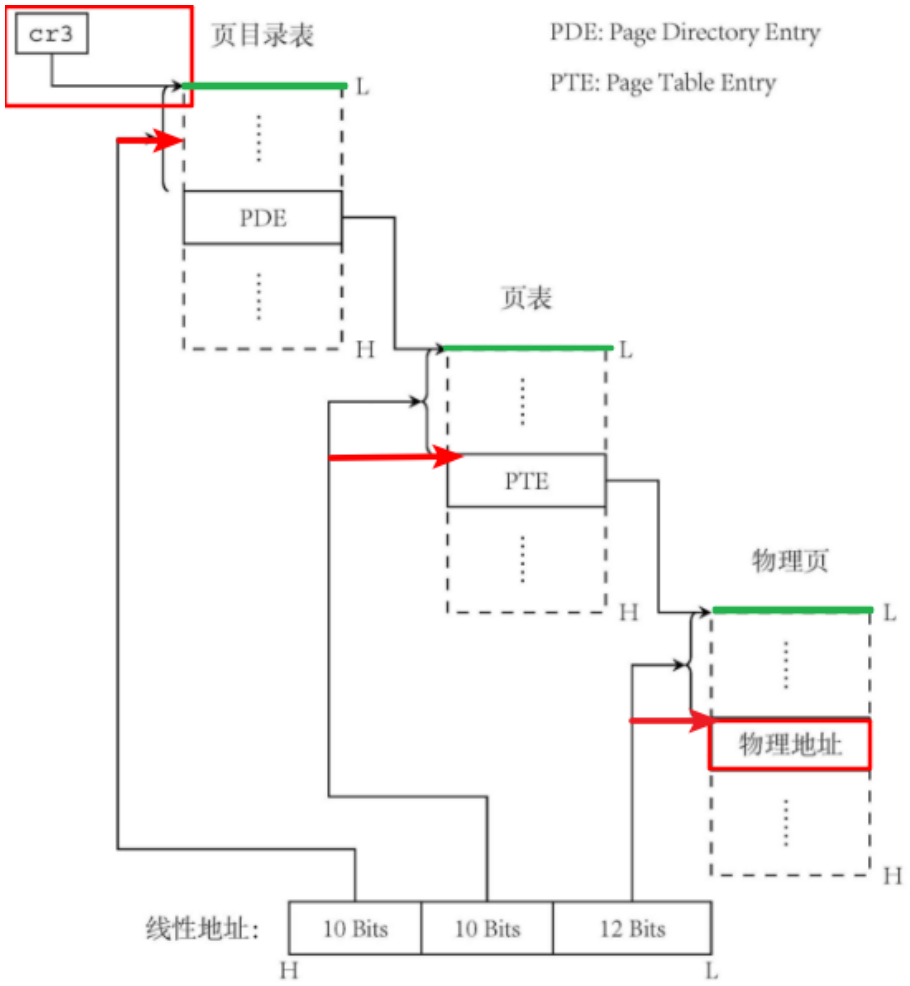

图 19: 分页机制

确定物理地址的步骤分为如下三步：

• 从 CR3 中取出进程的页目录地址，根据线性地址前十位，在页目录表中找到对应的索引项；

• 根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

• 将页的起始地址与线性地址中最后 12 位相加，得到最终的物理地址。

PDE 和 PTE 的结构如下：

31 12	11 8	7	6	5	4	3	2	1	0
页表基址	Avail	G	PS	0	A	PCD	PWT	U/S	R/W	P

图 20: PDE 结构

31 12	11 8	7	6	5	4	3	2	1	0
页基址	Avail	G	PAT	D	A	PCD	PWT	U/S	R/W	P

图 21: PTE 结构

关于不同位的含义，总结成下表：

表 2: 不同位的含义解释

名称	含义
P	存在位标志，表示当前条目所指向的页或页表是否在物理内存中
R/W	读写位标志，指定一个页或者一组页的读写权限。
U/S	用户/超级用户标志，指定一个页或者一组页的特权级
PWT	标志对单个页或者页表的 Write-back 或 Write-through 缓冲策略
PCD	用于控制对单个页或者页表的缓冲
A	Access，标志页或者页表是否被访问
D	Dirty，标志页或者页表是否被写入
PAT	Page Table Atttribute Index，实验中将该位设为 0 即可。
PS	Page Size，页大小。PS=0 时页大小为 4KiB，PDE 指向页表。
G	指示全局页。
AVL	保留字段。

至此，关于分页机制的基本知识已经较为全面地给出。

4.2 分页机制的基本方法和思路

pmtest6.asm 中给出了基本的分页机制方法和思路。大致总结如下：初始化 PDE → 初始化 PTE → 正确设置 CR3 的值，指向页目录表基地址 → 修改 CR0 最高位为 1，开启分页。

4.3 PDE 和 PTE 的计算方法

pmtest6.asm 内存映射图如图2所示。每个 PDE 前 20 位表示对应的页表的基地址。每个页表包含 1024 个 PTE。每个 PTE 指向一个页。在 pmtest6 中，1024 个页表占用 4M 的空间，它们一共可以指向 1024*1024 个页，对应 4GB 的空间。关于寻址的计算方法在19也已经提及。

Q：为什么代码 3.22 里面，PDE 初始化添加了一个 PageTblBase(Line212)，而 PTE 初始化时候没有类似的基地址呢（Line224)?

A：PDE 初始化添加了一个 PageTblBase(Line 212) 是因为页表起始地址为 PageTblBase（0x201000h，即 2M+4K)。而在 PTE 初始化时，由于 pmtest6 中物理地址等于线性地址，PTE 的起始地址就是 0。所以可以省略基地址。

4.4 获取内存布局的方法

参照 pmtest7 的方法，可以调用中断 15h 来获取内存信息。调用该中断需要设置如下参数：

- eax：要想获取内存信息，需要将 ax 赋值为 0E820h。
- ebx：放置“后续值”，第一次调用需要设置 ebx 为 0。
- es:di：指向一个 ARDS。
- 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。
- edx：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。

中断调用之后，结果存放于下列寄存器之中。

- CF：CF=0 表示没有错误，否则存在错误。
- eax：0534D4150h（'SMAP'）。
- ecx：BIOS 填充在地址范围描述符中的字节数量。
- ebx：这里放置着为等到下一个地址描述符所需要的后续值。

ARDS 的结构为：

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	长度的低 32 位
12	LengthHigh	长度的高 32 位
16	Type	这个地址范围的地址类型

其中 Type 不同取值的含义见下表：

取值	名称	意义
1	AddressRangeMemory	这个内存段是可以被 OS 使用的 RAM
2	AddressRangeReserved	这个地址段正在被使用，或者被系统保留所以一定不能被 OS 使用
其他	未定义	保留，为未来使用

因为 int 15h 需要将内存段描述保存到缓冲区中，所以在调用该中断前，需要准备好缓冲区。我们可以将所有 ARDS 保留在同一片缓冲区中，但是要保证缓冲区大小充足。

5 思考题汇总

1. 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？

• 区别:

分页	分段
在分页中，进程的地址空间被划分为固定大小的页面	在分段中，进程的地址空间被划分为大小不同的段
操作系统负责分页	编译器负责分段
页大小由硬件决定	段大小由用户给出
速度比分段块	分段速度慢
分页会导致内部碎片	分段导致外部碎片
分页中，逻辑地址被划分为页号和页偏移	分段中，逻辑地址被划分为段号和段偏移
分页包含一个页表，页表包含每个页的基地址	分段包含段表，段表中包含段号和段偏移量
分页对于用户不可见	分段对于用户可见
在分页中，处理器需要页号和页偏移来计算实际物理地址	分段中，处理器使用段号和段偏移量计算地址

段页式如何搭配：分段和分页相结合，其地址结构由：段号、段内页号、页内地址三部分组成。在段页式系统中获得一条指令需要三次访问内存，第一次访问内存中的段表，第二次访问内存中的页表，第三次访问内存中的数据。

2. 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 是页目录表中的表项，PTE 是页表中的表项。初始化需要 loop 循环依次赋值，具体可以参考3.1.2节的细节阐述。

3. 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?

   GDT 表中是线性地址，PDE 和 PTE 中的地址是物理地址。GPT 是段式管理用到的，而分段就是将逻辑地址转换为线性地址。PDE 是定位 PTE 的，存放的是 PTE 的物理地址，PTE 是定位页的，存放的是页的物理地址。

4. 为什么 PageTblBase 初始值为 2M+4K？能不能比这个值小?

   不能。因为设置页目录表起始地址是 2M,且页目录表大小为 4K。所以 PageTblBase 初始值必须大于等于 2M+4K。否则 PTE 与 PDE 存放位置会有交叉，导致错误。

5. 怎么读取本机的实际物理内存信息?

   在3.2节中，我们使用的是 int 15h,ax=0xE820h 的中断。它可以得到 ARDS 的列表，这也是当下主流的查看内存的方法。事实上，在早期计算机上，还有许多其他

函数（“int 0x12”，“int 0x15，ax=0xE881”，“int 0x15，ax=0xE801”，“int 0x15，ah=0xC7”，“int 0x15，ah=0x8A ”，”int 0x15, ax=0xDA88”，”int 0x15, ah=0x88”）可以实现这一功能。不过，所有这些都在 1990 年代初被弃用了。

6. 如何进行地址映射与切换?

   参考3.3节内容，地址映射与切换的原理在其中已经清楚地阐述过了。

6 实验问题总结

1. 如果线性地址 = 物理地址不成立，那么在切换页表时需要哪些额外的操作?

   经过我们的初步讨论，有以下可行方案：可以关闭分页，直接操作物理内存地址；把 PTE 和 PDE 的地址设置为与线性地址一一对应；把 PTD 和 PDE 的地址存在特定虚拟地址上。

2. 在打开分页模式后，示例代码中使用了如下的短跳转。这个短跳转的意义是什么?如果不加是否能正确运行?

```
1     jmp short .3
2 .3:
3     nop
```

   针对这个问题，我们提出了以下猜想 • 出于内存对齐的原因。不过在讨论后，我们觉得内存对齐应该由汇编器负责，而不应该手动添加。• 通过短跳转刷新指令寄存器。短跳转也许可以将指令寄存器中的值从物理地址切换到虚拟地址。不过，我们在试验之后，发现去掉 jmp 后仍能运行。针对这个现象，有组员提出，能够运行或许是由于线性地址 = 物理地址，有待进一步验证。• 有可能是希望借助 jmp 刷新 CPU 指令缓存。因为在启用分页前后，缓存会从 Physically tagged 转换为 Virtually tagged。jmp 可能会刷新缓存，加速运行。有待进一步验证。

3. 代码中初始化页表时使用了 mul ebx，其中 ebx 中的值为 1024。这个命令可以改成 shl eax,10 吗?

   我们感觉可以。

7 各人实验贡献与体会

通过本次实验，对操作系统的段页式管理的原理和代码实现有更深刻的理解。在获取系统内存布局后，能够对 PDE、PTE 的数量有更加合理的设置，从而减小程序占用内存空间。通过对 pmtest8 的调试，对地址映射关系以及内存管理更加熟悉。在进阶题的思考过程中，我们对内存分配的几种方式又进行了回顾。

本次实验中，李心杨负责编写内存管理器；王宇骥调试 pmtest6、pmtest7 相关内容，回答相关思考题；郑炳捷调试 pmtest8 相关内容，回答相关思考题；林锟杨负责基础题的编程任务，完整实现了整个程序。

教师评语		
姓名	**学号**	**分数**
李心杨	2020302181022	
王宇骥	2020302181008	
林锟扬	2020302181032	
郑炳捷	2020302181024	
教师签名： 年 月 日		