

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	2024.9.23
实验名称	保护模式工作机理	实验周次	第二周
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

Contents

1 实验目的及实验内容	4
1.1 实验知识要求:	4
1.2 实验内容:	4
2 实验环境及实验步骤	4
2.1 实验环境	4
2.2 认真阅读章节资料, 掌握什么是保护模式, 弄清关键数据结构: GDT、descriptor、selector、GDTR, 及其之间关系, 阅读 pm.inc 文件中数据结构以及含义, 写出对宏 Descriptor 的分析	5
2.2.1 各种数据结构	5
2.3 关系	6
2.4 对宏 Descriptor 的分析	7
2.5 调试代码, /a/ 掌握从实模式到保护模式的基本方法, 画出代码流程图, 特别注意跳转问题, 如果把跳转直接改成 jmp offset, 而不用 selector:offset 形式, 会是什么结果, 反汇编比较一下区别	7
2.5.1 实模式到保护模式	7
2.5.2 原理阐述	10
2.5.3 代码流程图	12
2.5.4 更改为 jmp offset	12
2.6 调试代码, /b/, 掌握 GDT 的构造与切换, 从保护模式切换回实模式方法	13
2.7 调试代码, /c/, 掌握 LDT 切换	15
2.8 调试代码, /d/ 掌握一致代码段、非一致代码段、数据段的权限访问规则, 掌握 CPL、DPL、RPL 之间关系, 以及段间切换的基本方法	16
2.8.1 权限访问规则	17
2.8.2 CPL、DPL、RPL 之间关系	17
2.8.3 段间切换的基本方法	17
2.9 调试代码, /e/ 掌握利用调用门进行特权级变换的转移的基本方法	18
2.9.1 特权级变换的转移的基本方法	19
3 实验过程分析	19
3.1 问题 1: GDT、Descriptor、Selector、GDTR 结构, 及其含义是什么? 他们的关联关系如何? pm.inc 所定义的宏怎么使用?	19
3.1.1 DT、Descriptor、Selector、GDTR 结构, 及其含义	19
3.1.2 关系	20
3.1.3 pm.inc 所定义的宏	21
3.2 问题 2: 从实模式到保护模式, 关键步骤有哪些? 为什么要关中断? 为什么要打开 A20 地址线? 从保护模式切换回实模式, 又需要哪些步骤?	22
3.2.1 实模式到保护模式的关键步骤:	22
3.2.2 为什么要进行关中断:	22
3.2.3 为什么要打开 A20 地址线:	22
3.2.4 保护模式切换回实模式的步骤:	22
3.3 问题 3: 解释不同权限代码的切换原理, call, jmp, retf 使用场景如何, 能够互换吗?	23
3.3.1 call 指令	23
3.3.2 jmp 指令	23
3.3.3 retf 指令	23
3.3.4 互换	23

3.4 动手改 1：自定义添加 1 个 GDT 代码段、1 个 LDT 代码段，GDT 段内要对一个内存数据结构写入一段字符串，然后 LDT 段内代码段功能为读取并打印该 GDT 的内容	23
3.5 动手改 2：自定义 2 个 GDT 代码段 A、B，分属于不同特权级，功能自定义，要求实现 A->B 的跳转，以及 B->A 的跳转。	24
3.6 实验问题与故障分析：	26
4 实验结果总结	28
5 各人实验贡献与体会（每人各自撰写）	29
6 教师评语	29

1 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；必要的原理分析)

1.1 实验知识要求：

1. x86 CPU 的基本模式：实模式、保护模式
2. 环保护及其拓展
3. 段的特权类型
4. 不同特权级别段之间的代码转移
5. 不同特权级代码之间切换，上下文如何恢复？
6. 如何实现高特权级低特权级

1.2 实验内容：

(一)

1. 认真阅读章节资料，掌握什么是保护模式，弄清关键数据结构：GDT、descriptor、selector、GDTR，及其之间关系，阅读 pm.inc 文件中数据结构以及含义，写出对宏 Descriptor 的分析
2. 调试代码，/a/ 掌握从实模式到保护模式的基本方法，画出代码流程图，如果代码/a/中，第 71 行有 dword 前缀和没有前缀，编译出来的代码有区别么，为什么，请调试截图。
3. 调试代码，/b/，掌握 GDT 的构造，体会保护模式下地址空间的变化、从保护模式切换回实模式方法
4. 调试代码，/c/，掌握 LDT 的构造
5. 调试代码，/d/掌握一致代码段、非一致代码段、数据段的权限访问规则，掌握 CPL、DPL、RPL 之间关系，以及段间切换的基本方法
6. 调试代码，/e/掌握利用调用门进行特权级变换的转移的基本方法

(二)

1. GDT、Descriptor、Selector、GDTR 结构，及其含义是什么？他们的关联关系如何？pm.inc 所定义的宏怎么使用？
2. 从实模式到保护模式，关键步骤有哪些？为什么要关中断？为什么要打开 A20 地址线？从保护模式切换回实模式，又需要哪些步骤？
3. 阐述不同权限代码的切换方法，call, jmp, retf 使用场景如何，能够互换吗？
4. 课后动手改：
 - a) 自定义添加 1 个 GDT 代码段、1 个 LDT 代码段，GDT 段内要对一个内存数据结构写入一段字符串，然后 LDT 段内代码段功能为读取并打印该 GDT 的内容；
 - b) 自定义 2 个 GDT 代码段 A、B，分属于不同特权级，功能自定义，要求实现 A->B 的跳转，以及 B->A 的跳转。

2 实验环境及实验步骤

2.1 实验环境

WSL2 Ubuntu20.04

Bochs 2.7

2.2 认真阅读章节资料，掌握什么是保护模式，弄清关键数据结构：GDT、descriptor、selector、GDTR，及其之间关系，阅读 pm.inc 文件中数据结构以及含义，写出对宏 Descriptor 的分析

2.2.1 各种数据结构

- 保护模式：

保护模式是 x86 处理器的工作模式之一，它与实模式（Real Mode）相比，提供了更强的内存管理功能和更复杂的权限控制机制。在保护模式下，CPU 可以使用硬件提供的内存保护、多任务处理、虚拟内存等特性，从而提升系统的稳定性和安全性。关键特性包括：

- 段式内存管理
- 权限控制
- 分页

- GDT：

GDT 是保护模式下用于管理内存的结构，包含多个段描述符（Descriptor），在保护模式下提供了更灵活的内存管理机制。

- descriptor：

在保护模式下，通过 GDT 定义多个段，每个段有自己的基地址和限长，保护模式提供了更灵活的内存管理机制。主要字段

- 基地址：短的起始地址
- 段界限：段的大小
- 段的类型和权限：例如这是数据段、代码段，是否可写、可执行，以及其特权级别（Ring 0 到 Ring 3）。

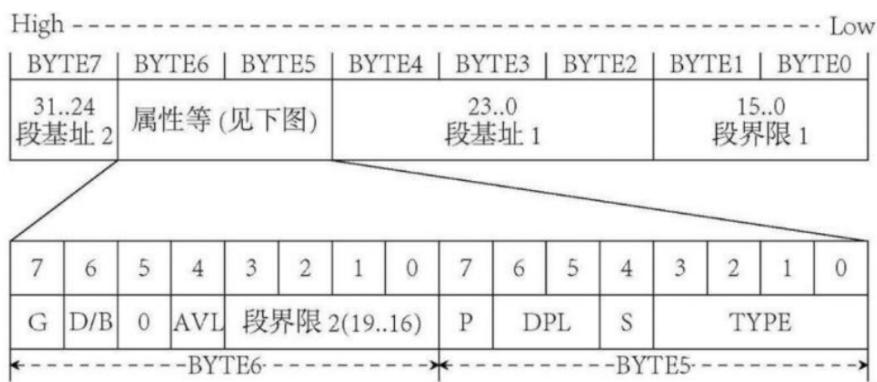


Figure 1: 段描述符的结构示意图

- Selector：

选择子，是一个 16 位的值，用于在 GDT 或 LDT 中索引一个描述符，选择子有三部分：

- 索引：GDT 中的某个描述符的编号
- TI 位：指示选择子指向的是 GDT（值为 0）还是 LDT（值为 1）

- RPL: 请求的特权级别, 用于权限检查

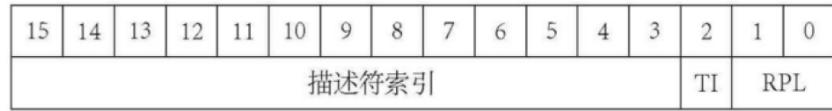


Figure 2: 选择子的结构示意图

- GDTR:
全局描述符表寄存器, 存储 GDT 的基地址和限长

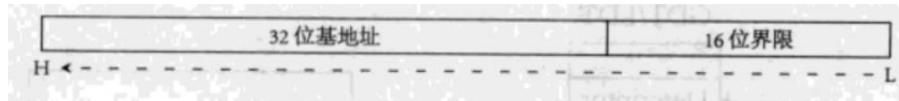


Figure 3: GDTR 的结构示意图

2.3 关系

保护模式启用后, CPU 可以访问 **GDT** 来进行段式内存管理。**GDT** 存放所有段描述符的表, 描述了内存段的各种信息 (基地址、界限、权限等)。**Descriptor** 存储在 GDT 中的每个条目, 描述某个段的具体属性。**Selector** 存储在段寄存器中的值, 用于索引 GDT 或 LDT 中的描述符。**GDTR** 一个特殊寄存器, 指向 GDT 在内存中的位置。

寻址方式: 保护模式下先通过 **GDTR** 找到 GDT, 然后通过 **descriptor** 找到对应段的地址, 然后加上段内偏移 **offset**, 得到地址。

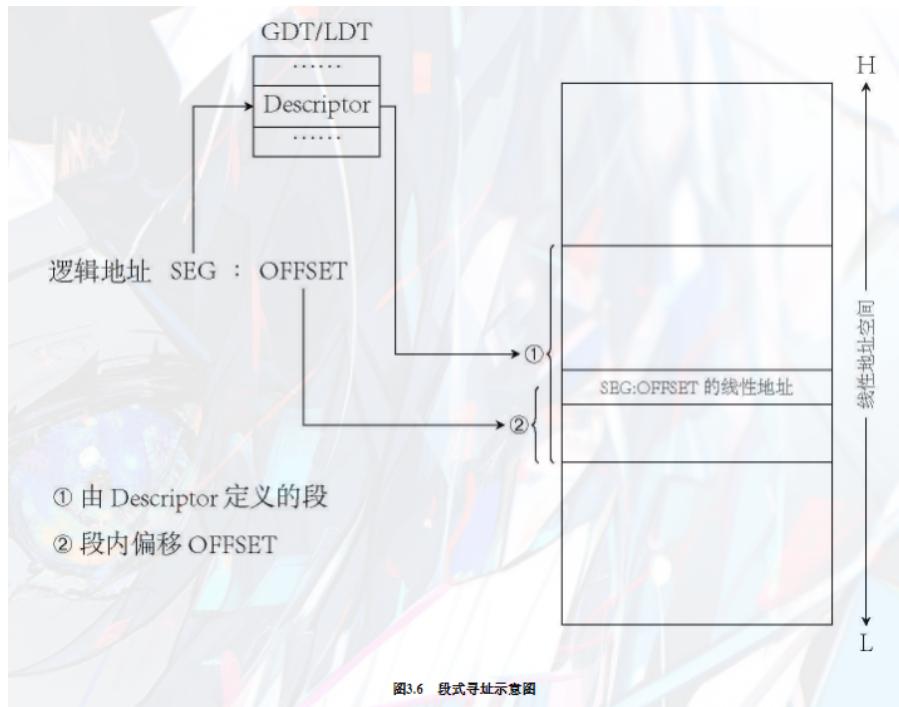


Figure 4: 段式寻址示意图

2.4 对宏 Descriptor 的分析

```
1 %macro Descriptor 3 ; 宏
2     dw %2 & OFFFFh           ; 段界限 1
3     ; 意思是宏的第二个参数传入，并且把高 16 位截掉，表示段界限
4     ; 将段界限的低 16 位存储在描述符的低两字节中。
5     dw %1 & OFFFFh           ; 段基址 1
6     ; 将段基址的低 16 位存储在描述符的后两字节中。
7     db (%1 >> 16) & OFFh    ; 段基址 2
8     ; 将段基址的中间 8 位（第 17 到 24 位）存储在描述符中。
9     dw ((%2 >> 8) & OF00h) | (%3 & OFFFFh) ; 属性 1 + 段界限 2 + 属性 2
10    ; 这一步将段界限的高 4 位和描述符属性存储在描述符的合适位置
11    db (%1 >> 24) & OFFh    ; 段基址 3
12    ; 将段基址的最高 8 位存储在描述符的最后一个字节中。
13 %endmacro ; 共 8 字节
```

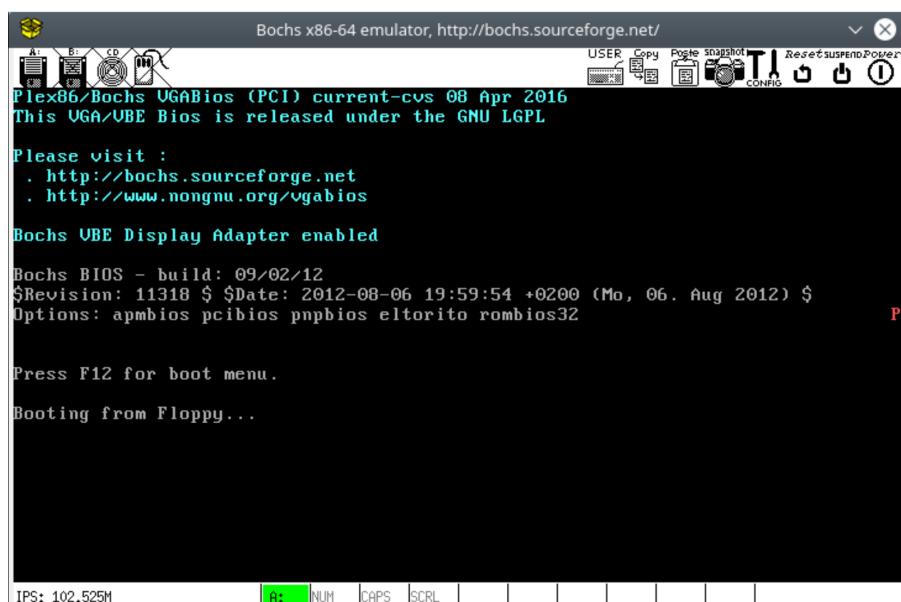
宏通过使用位运算来拆分和组合段基址和段界限，并添加段属性，使得段描述符能够存储足够的信息来描述内存段。这些字节最终会被放置在全局描述符表（GDT）或局部描述符表（LDT）中，用于内存管理和保护模式的地址转换。

2.5 调试代码，/a/ 掌握从实模式到保护模式的基本方法，画出代码流程图，特别注意跳转问题，如果把跳转直接改成 jmp offset，而不用 selector:offset 形式，会是什么结果，反汇编比较一下区别

2.5.1 实模式到保护模式

通过引导扇区进入保护模式 编译 pmtest1.asm，并将生成的二进制写入软盘映像，最后运行 Bochs

```
1 nasm pmtest1.asm -o pmtest1.bin
2 dd if=pmtest1.bin of=a.img bs=512 count=1 conv=notrunc
3 bochs -f bochsrc
```



可以看到在屏幕中部右侧出现了一个红色的大写字母“P”，然后程序再也不动了

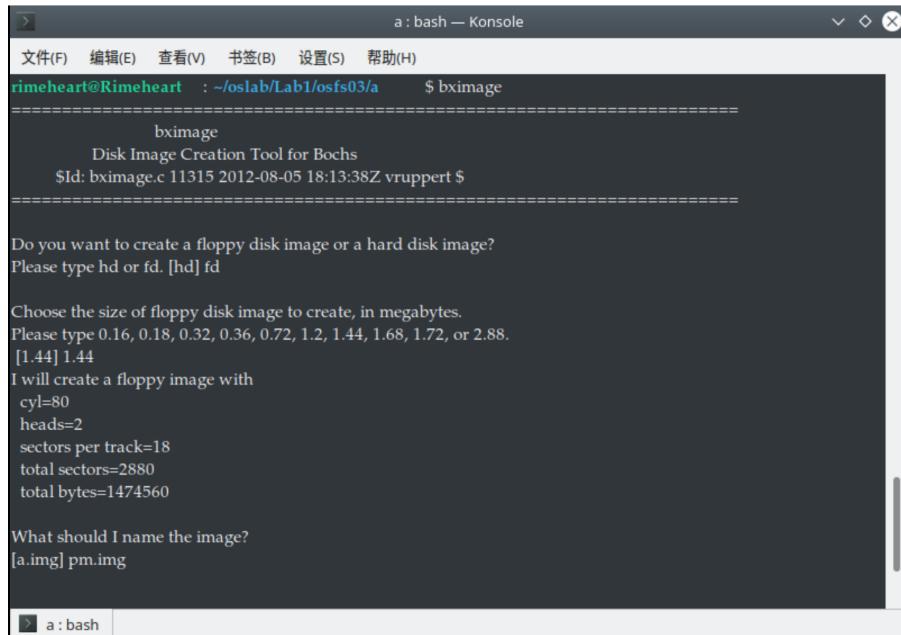
通过 DOS 进入保护模式 因为引导扇区的空间有限，最多容纳 512 字节，所以接下来尝试把程序编译成 COM 文件然后让 DOS 来执行它

COM 文件格式概述：

- 单段结构：COM 文件是单一段（代码段、数据段和堆栈段共用）的简单二进制文件，没有文件头，加载到内存后直接从偏移地址 **0x0100** 开始执行。
- 最大大小 64KB：由于它使用的是实模式地址，COM 文件最大只能是 64KB。

步骤：

- 先下载 freedos，解压并将其中的 a.img 复制到代码所在的工作目录，改名为 freedos.img。
下载链接：[freedos](#)
- 用 bximage 生成软盘映像，pm.img



- 修改 bochsrc，添加下面三行

```
1 floppya: 1_44=freedos.img, status=inserted
2 floppyb: 1_44=pm.img, status=inserted
3 boot: a
```

相当于有两个软盘，软盘 a 作为 bootloader，引导操作系统启动，软盘 b 里面放置我们自己的对操作系统的操作（相当于操作系统内核）

The screenshot shows the Kate text editor with the file 'bochsrc' open. The code in the editor is as follows:

```

megs: 32

# filename of ROM images
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/vgabios/vgabios.bin

# what disk images will be used
floppya: 1_44=a.img, status=inserted

# choose the boot disk.
boot: floppy

# where do we send log messages?
log: bochsout.txt

# disable the mouse
mouse: enabled=0

# enable key mapping, using US layout as default.
keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map

floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=pm.img, status=inserted
boot: a

```

At the bottom of the editor window, it says '行 29, 列 8' (Line 29, Column 8), '插入 软制表符: 4' (Insert Soft Tab: 4), 'UTF-8', 'Normal', and there are search and replace buttons.

- 启动 Bochs，在 FreeDOS 使用 **format b:** 命令格式化B: 盘

The screenshot shows a terminal window in the Bochs x86-64 emulator. The window title is 'Bochs x86-64 emulator, http://bochs.sourceforge.net/'. The terminal output is as follows:

```

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:>format b:

Reading boot sector...
Cylinder: 0 Head: 0

Drive appears unformatted, UNFORMAT information not saved.

Creating file system...
Cylinder: 0 Head: 0

Format operation complete.

A:>

```

At the bottom of the terminal window, it says 'IPS: 27.052M' and has a status bar with keys A:, B:, NUM, CAPS, SCRL.

- 将代码第 8 行中的 07c00h 改为 0100h 并重新编译

The screenshot shows the Kate text editor with the file 'pmtest1.asm' open. The code in the editor is as follows:

```

; -----
; pmtest1.asm
; 编译方法: nasm pmtest1.asm -o pmtest1.bin
; -----
%include "pm.inc" ; 常量, 宏, 以及一些说明
org 0100h

```

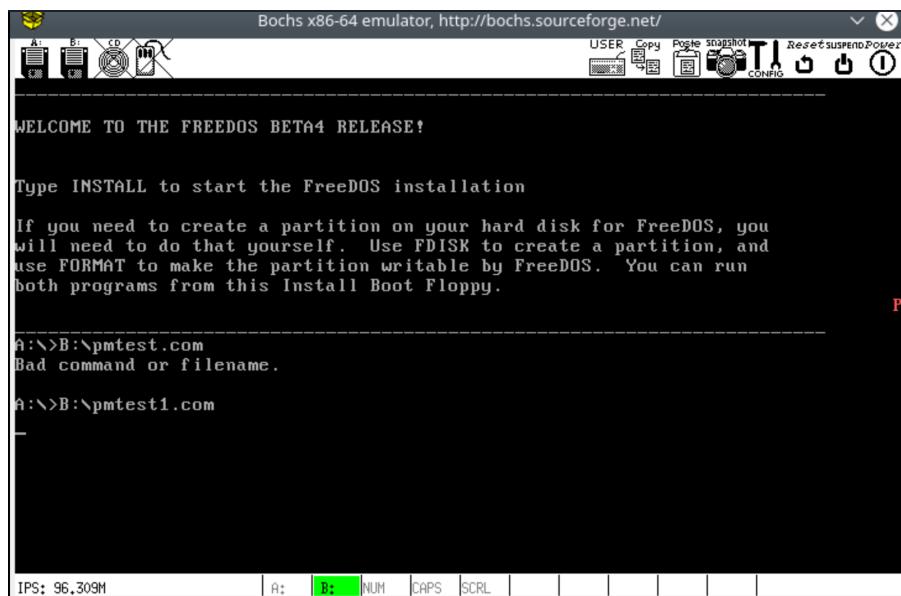
- 把代码编译成.com 文件

```
1 nasm pmtest1.asm -o pmtest1.com
```

- 把 pmtest1.com 复制到虚拟软盘 pm.img 上

```
1 sudo mkdir /mnt/floppy
2 sudo mount -o loop pm.img /mnt/floppy
3 sudo cp pmtest1.com /mnt/floppy
4 sudo umount /mnt/floppy
```

- 运行 Bochs 后，在 freedos 中使用 **pmtest1.com** 即可执行



2.5.2 原理阐述

- 准备 GDT 描述符

```
1 [SECTION .gdt]
2 ;                                     段基址 ,          段界限      , 属性
3 LABEL_GDT:     Descriptor      0,           0, 0          ; 空
   ; 描述符
4 LABEL_DESC_CODE32: Descriptor      0, SegCode32Len - 1, DA_C + DA_32
   ; 非一致代码段
5 LABEL_DESC_VIDEO:  Descriptor 0B8000h,           0ffffh, DA_DRW
   ; 显存首地址
6 ; GDT 结束
```

- 定义了 GDT 起始位置。
- 定义了 32 位代码段描述符。
- 定义了任务的显存首地址。

- 用 lgdt 加载 gdtr

- 将 GDT 的基地址写入 GdtPtr 中以准备加载 GDTR 寄存器

```
1 xor eax, eax
2 mov ax, ds
3 shl eax, 4
4 add eax, LABEL_GDT
5 mov dword [GdtPtr + 2], eax
```

- 加载 GDTR

```
1 lgdt [GdtPtr]
```

- 打开 A20

```
1 in al, 92h
2 or al, 00000010b
3 out 92h, al
```

通过 IO 寻址方式，使用 in 和 out 指令打开地址线 A20，使得 CPU 能够访问 1MB 以上的内存地址

- 置寄存器 cr0 的 PE 位

```
1 mov eax, cr0
2 or eax, 1      ;PE位为第0位
3 mov cr0, eax
```

- 执行长跳转，进入保护模式

```
1 jmp dword SelectorCode32:0
```

这条指令通过加载 SelectorCode32 到 cs 寄存器，跳转到保护模式下的 32 位代码段。

2.5.3 代码流程图

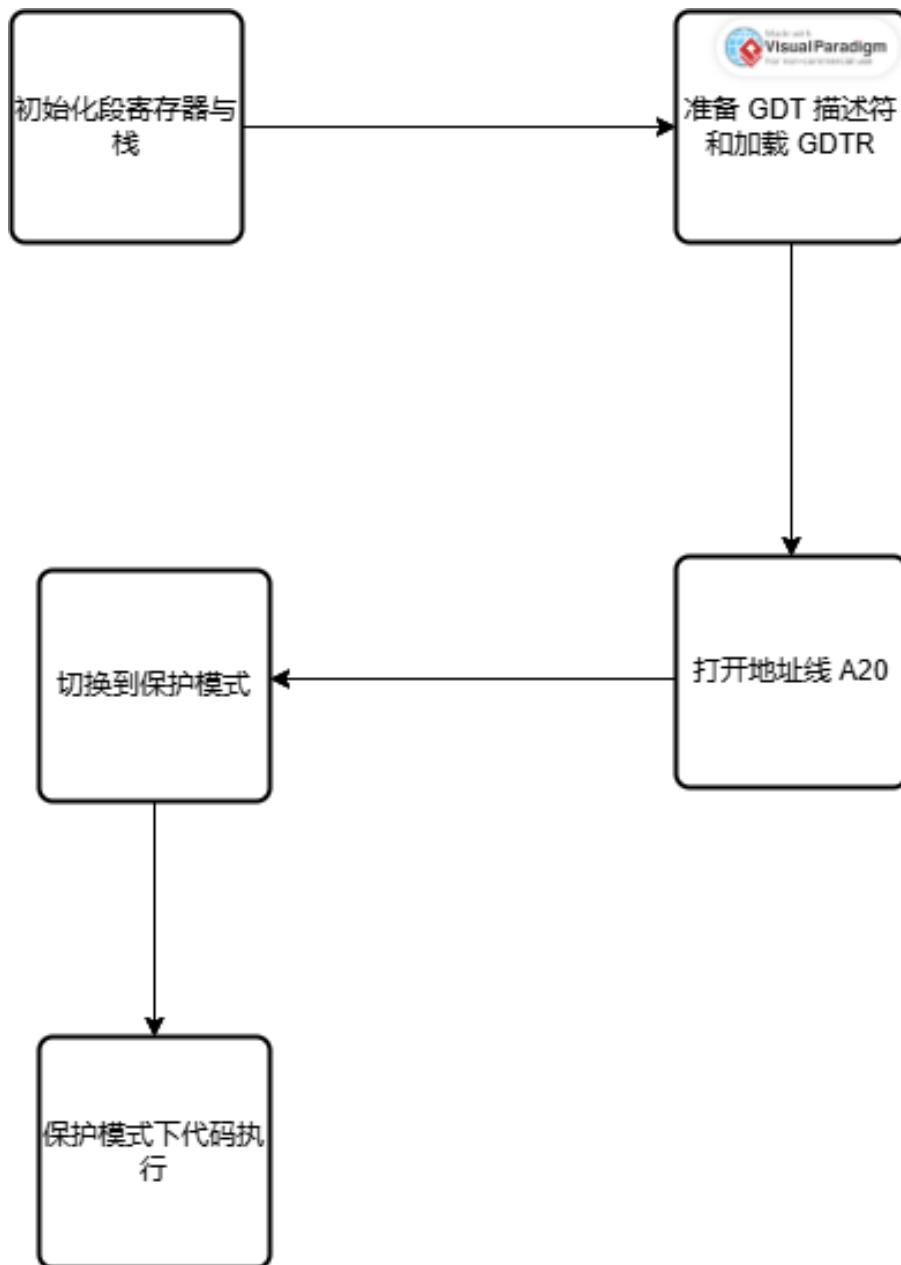


Figure 5: 代码流程图

2.5.4 更改为 jmp offset

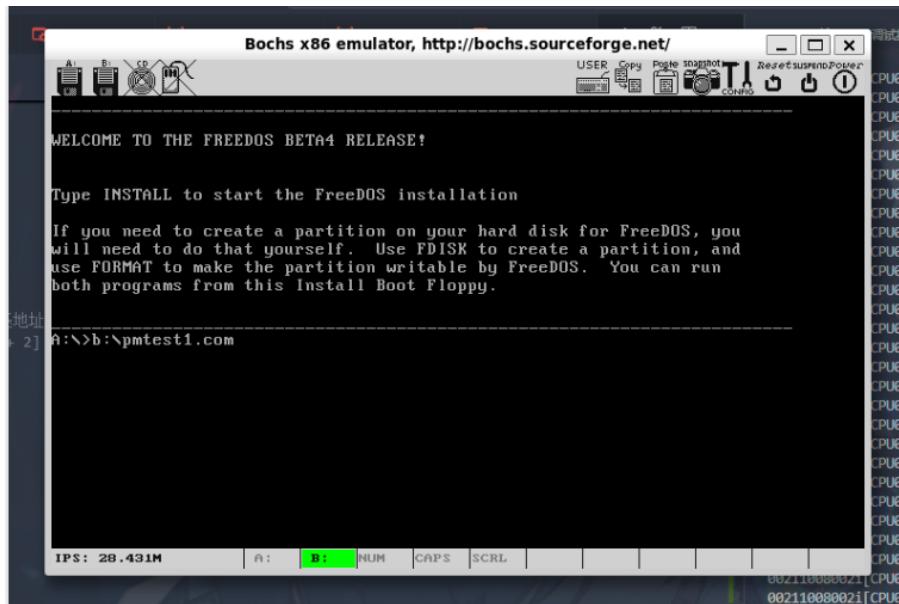
```
xchg bx, bx
xchg bx, bx
xchg bx, bx
xchg bx, bx
; 真正进入保护模式
; jmp dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs,
; ; 并跳转到 Code32Selector:0 处
jmp 0
; END of [SECTION .s16]
```

其中

```
1 xchg bx,bx
2 xchg bx,bx
3 xchg bx,bx
4 xchg bx,bx
5 jmp 0
```

前四行代码仅用于调试定位，后面为修改跳转指令如果把跳转直接改成 `jmp offset`，而不用 `selector:offset` 形式：

- 在 `jmp dword SelectorCode32:0` 处，使用的是 `selector:offset` 形式。这表示跳转时不仅指定了目标偏移量 0，还指定了一个段选择子 `SelectorCode32`。这保证了 CPU 在保护模式下正确加载段寄存器 CS（代码段），因为在保护模式下，段寄存器的内容不再是实模式中的段基址，而是通过 GDT 中的描述符来定义的。
- 如果改成 `jmp offset`，比如 `jmp 0`，那么问题就会出现在 CPU 在进入保护模式后仍然使用当前的 CS 段寄存器，而不是更新为保护模式下的段选择子。这样的话，代码可能会在错误的内存位置执行，导致未定义的行为，甚至崩溃。因为在保护模式下，段寄存器不再代表线性地址，而是指向 GDT 中的某个段描述符。

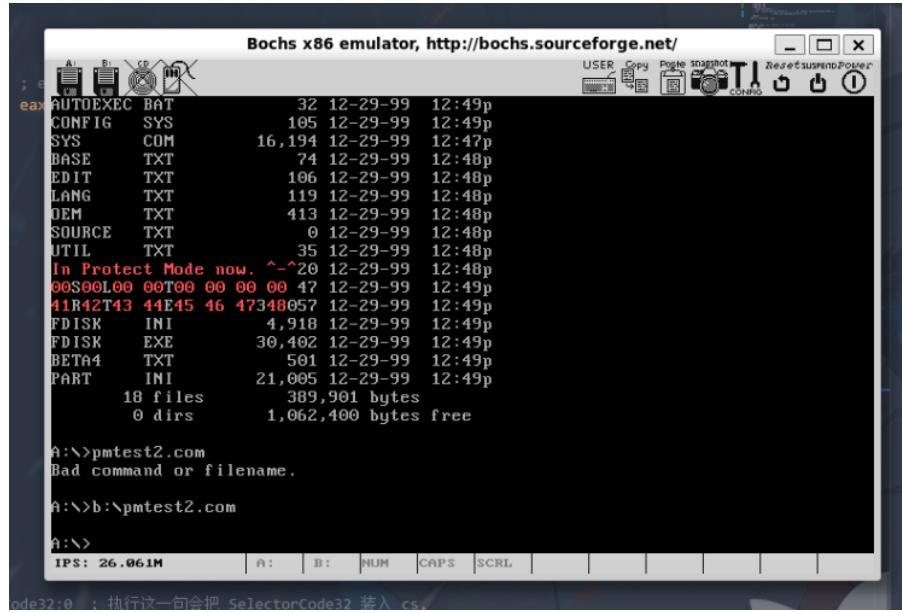


使用 `pmmmod1.com` 运行，最终界面卡住，实验结果如下：

- 中断错误：CPU 执行到 `int 0x20`，表明程序执行异常，进入了未预期的中断。
- 无限循环或崩溃：程序回到 BIOS 的入口点 (`f000:ffff0`)，引导过程出现了问题，无法正确加载操作系统或引导代码

2.6 调试代码，/b/，掌握 GDT 的构造与切换，从保护模式切换回实模式方法

启动 `bochs` 可以看到如下的结果：



可以看到，程序打印出两行数字，第一行全部是零，说明内存地址 5MB 处都是 00，而下一行变成了 41 42 43...，即十六进制的 A、B、C ...H，说明写操作成功。同时，程序执行结束后不再像上一个程序一样进入死循环，而是重新出现 DOS 提示符 A:，表明我们重新回到了实模式下的 DOS。

```

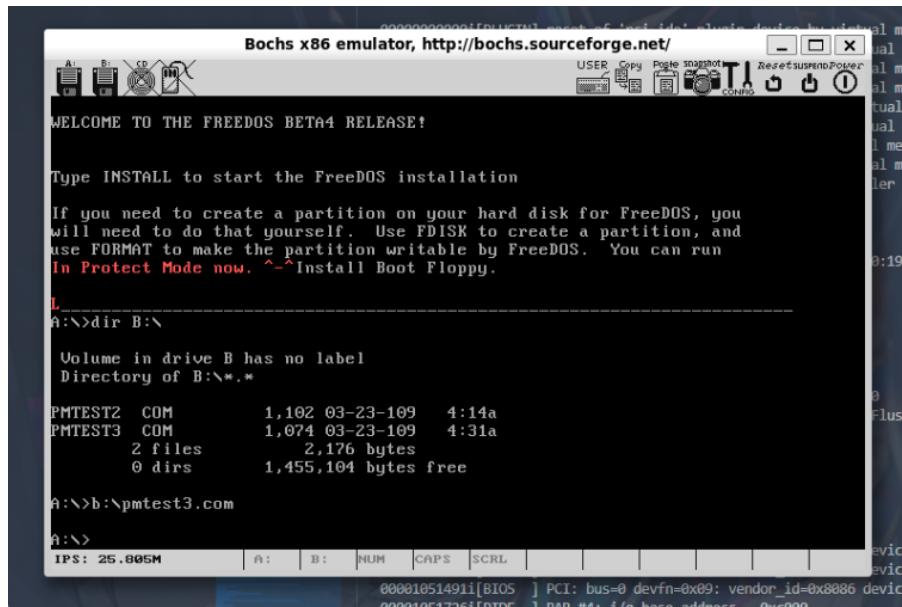
1 LABEL_REAL_ENTRY:
2     mov ax, cs
3     mov ds, ax
4     mov es, ax
5     mov ss, ax
6
7     mov sp, [SPValueInRealMode]
8
9     in al, 92h
10    and al, 11111101b
11    out 92h, al
12
13    sti
14    mov ax, 4c00h
15    int 21h

```

通过这段代码实现了从保护模式切换回实模式的过程，包括如下步骤：

- 重置段寄存器：将 ds、es、ss 重新指向当前代码段，以确保数据和栈的操作可以正确访问。
- 恢复栈指针：将实模式下的栈指针还原，以确保切换后栈上的数据不会丢失。
- 关闭 A20 地址线：通过对端口 92h 的操作，关闭 A20 地址线。这是为了保证实模式的寻址范围（1MB）。
- 开中断：使用 sti 指令打开中断，确保在实模式下可以处理外部中断。
- 返回 DOS：使用 int 21h，ax = 4c00h 表示程序正常结束并返回 DOS。

2.7 调试代码, /c/, 掌握 LDT 切换



- 打印字符串”In Protect Mode now.”:

```
1 mov ah, 0Ch ; 0000: 黑底 1100: 红字
2 xor esi, esi
3 xor edi, edi
4 mov esi, OffsetPMMessag ; 源数据偏移
5 mov edi, (80 * 10 + 0) * 2 ; 目的数据偏移。屏幕第 10 行，第 0 列。
6 cld
```

- 将字符串的偏移量 OffsetPMMessag 加载到 ESI。
- EDI 设置为屏幕第 10 行，第 0 列的偏移，用于显示字符。
- 使用 lodsb 逐字节读取字符串，显示字符直到遇到字符串结束符 (0)。

- 加载 LDT 并跳转到局部代码段:

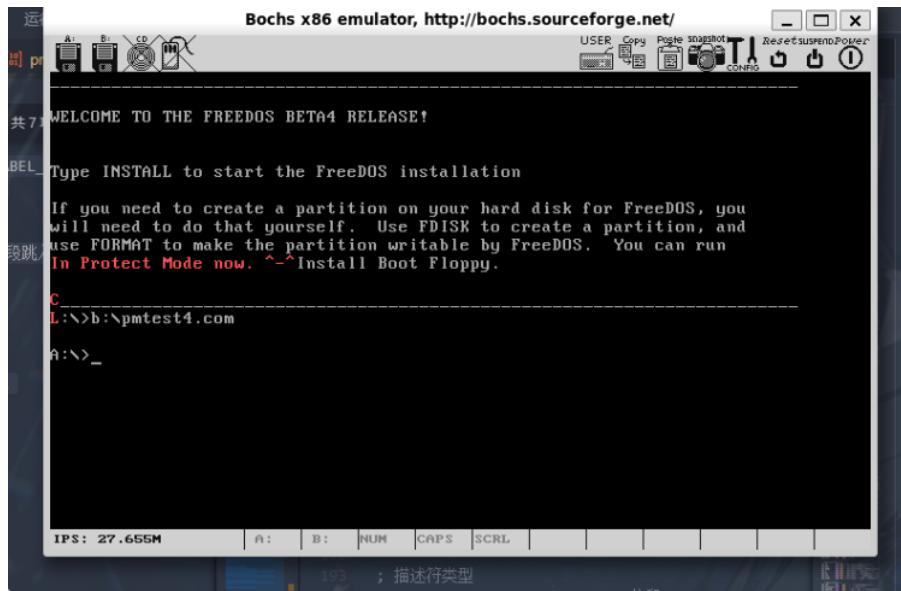
```
1 mov ax, SelectorLDT
2 lldt ax
3 jmp SelectorLDTCodeA:0 ; 跳入局部任务
```

- 局部描述符表中的代码段 ([SECTION .la])

```
1 mov ax, SelectorVideo
2 mov gs, ax ; 视频段选择子(目的)
3
4 mov edi, (80 * 12 + 0) * 2 ; 屏幕第 12 行，第 0 列。
5 mov ah, 0Ch ; 0000: 黑底 1100: 红字
6 mov al, 'L'
7 mov [gs:edi], ax
```

因此最终实现了在 [SECTION .S32] 中打印完 “In Protect Mode now.” 字符串后，出现一个红色的字符 L 。的结果

2.8 调试代码, /d/掌握一致代码段、非一致代码段、数据段的权限访问规则, 掌握 CPL、DPL、RPL 之间关系, 以及段间切换的基本方法



打印结果: “In Protect Mode now.” 字符串和红色的字符 L、C

- 实模式进入保护模式打印 “In Protect Mode now.”
- 调用门目标代码段 [SECTION .sdest], 打印字符 C:

```
1 mov ax, SelectorVideo
2 mov gs, ax
3 mov edi, (80 * 12 + 0) * 2
4 mov ah, 0Ch
5 mov al, 'C'
6 mov [gs:edi], ax
7 retf
```

- 设置显存段选择子，并将字符 C 显示在屏幕第 12 行，颜色为红色。随后通过 retf 返回调用门的调用点。
- 加载 LDT 并跳转到 LDT 代码段:

```
1 mov ax, SelectorLDT
2 lldt ax
3 jmp SelectorLDTCodeA:0
```

- LDT 代码段执行 ([SECTION .la]), 打印字符 L:

```
1 mov ax, SelectorVideo
2 mov gs, ax
3 mov edi, (80 * 13 + 0) * 2
4 mov ah, 0Ch
5 mov al, 'L'
6 mov [gs:edi], ax
7 jmp SelectorCode16:0
```

设置显存段选择子，并将字符 L 显示在屏幕第 13 行，颜色为红色。随后通过 jmp 跳转到 16 位代码段，准备返回实模式。

2.8.1 权限访问规则

代码段分为一致代码段和非一致代码段

- 一致代码段：

- 允许低特权级别 (CPL=CPL) 的代码跳转 (使用 call 或 jmp 指令) 到高特权级别的
一致代码段。
- 当从低 CPL 的代码段跳转到一致代码段时，CPU 会将 CPL 设置为目标段的
DPL=DPL (描述符特权级)，这可以使特权级别降低，但不提高。
- 一致代码段通常用于实现库函数或系统调用，以便用户级别代码可以访问内核级
别的功能。

- 非一致代码段：

- 允许低特权级别的代码跳转 (使用 call 或 jmp 指令) 到高特权级别的非一致代码
段。即 CPL-A >= DPL-B，此处 RPL 并不检查。
- 跳转到非一致代码段时，CPL 不会改变，仍然保持为低特权级别。
- 非一致代码段通常用于系统内核，以允许高特权级别的代码访问内核功能。

数据段主要用来存储数据，特权级控制了数据段的访问权限：

- 数据段的访问受 DPL 和访问代码的 CPL 控制
- 一般来说 CPL <= DPL 时允许访问数据段，否则不允许访问

2.8.2 CPL、DPL、RPL 之间关系

- CPL：当前正在执行的代码所在段的特权级，它反映了 CPU 当前的权限级别。CPL 存
储在 CS 寄存器的低两位中。
- DPL：段描述符中的特权级，它决定了该段的特权级别。DPL 存在于段描述符中。
- RPL：请求特权级别，是选择子中的两位字段，表示对段的访问请求级别。

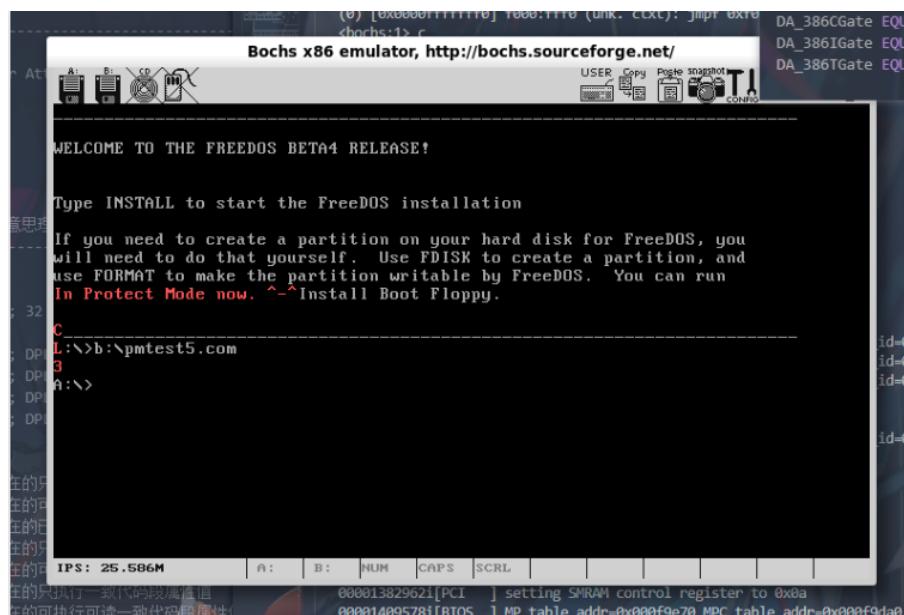
访问权限的判断主要通过三者的组合来进行，其中：段访问规则： $\max(CPL, RPL) \leq DPL$ ；
只有在请求者的权限 (CPL/DPL) 不高于段的 DPL 时，访问才被允许

2.8.3 段间切换的基本方法

- 准备好目标端描述符：在要进行切换的特权级别下，需要准备好目标段的描述符，包括
选择子 (Selector) 和段基地址。这通常在全局描述符表 (GDT) 或局部描述符表 (LDT)
中。
- 加载目标选择子：使用 mov 指令将目标选择子加载到相应的段寄存器中。不同的段寄
存器对应不同的段，例如 CS (代码段)、DS (数据段)、ES (附加数据段)、SS (堆栈
段) 等。

- 特权级别切换的方法取决于从哪个特权级别切换到哪个特权级别。在从用户模式 (CPL3) 切换到内核模式 (CPL0) 的情况下，通常需要执行以下步骤：
 - 关中断（使用 cli 指令）：这是为了防止在特权级别切换过程中发生中断，从而确保切换的原子性
 - 打开 A20 地址线：在某些系统中，需要通过 I/O 端口操作来打开 A20 地址线，以允许访问整个物理内存
 - 使用 jmp 指令：通过 jmp 指令，将控制权转移到新的代码段，以启动新特权级别的执行
- 完成特权级别切换：在新特权级别下，CPU 会从目标段开始执行指令。此时，CPL（当前特权级别）将与目标段的 DPL（描述符特权级别）相匹配

2.9 调试代码，/e/掌握利用调用门进行特权级变换的基本方法



打印字符 “In Protect Mode now.” 字符串和红色的字符 L、C 和 3
在前面的代码的基础上，/e/pmtest5.asm 代码中：

- 使用调用门进行特权级变换

```

1 push SelectorStack3
2 push TopOfStack3
3 push SelectorCodeRing3
4 push 0
5 retf ; Ring 0 -> Ring 3, 历史性转移！将打印数字 '3'。

```

在代码的关键部分，通过调用门从 Ring 3 特权级（用户态）进入 Ring 0 特权级（内核态）

- Ring3 中的操作：

```

1 mov ax, SelectorVideo
2 mov gs, ax ; 视频段选择子(目的)
3
4 mov edi, (80 * 14 + 0) * 2 ; 屏幕第 14 行, 第 0 列。
5 mov ah, 0Ch ; 0000: 黑底 1100: 红字
6 mov al, '3'
7 mov [gs:edi], ax
8
9 call SelectorCallGateTest:0 ; 测试调用门 (有特权级变换), 将打印字母
   'C'。
10 jmp $

```

- Ring0 中的操作 mov ax, SelectorVideo mov gs, ax ; 视频段选择子 (目的)
mov edi, (80 * 12 + 0) * 2 ; 屏幕第 12 行, 第 0 列。 mov ah, 0Ch ; 0000: 黑底 1100: 红字 mov al, 'C' mov [gs:edi], ax

2.9.1 特权级变换的转移的基本方法

调用门本质上是一个添加了属性的特殊入口地址：代码 A → 调用门 G → 代码 B；通过 Call Gate + Call，实现了从低特权级 → 高特权级的访问

- 创建调用门：在全局描述符表 (GDT) 或局部描述符表 (LDT) 中创建一个调用门描述符。这个描述符定义了目标代码段的选择子 (Selector)、目标代码段的偏移地址、目标代码段的特权级别 (DPL, Descriptor Privilege Level) 以及其他相关属性。
- 调用调用门：使用 call 指令来调用创建的调用门。例如，call SelectorCallGate:0 0，其中 SelectorCallGate 是调用门的选择子。call 指令会触发特权级别的转移，将控制权传递到目标代码段。
- 执行目标代码段：一旦调用门被触发，CPU 会开始执行目标代码段。目标代码段的 DPL (Descriptor Privilege Level) 可以与当前特权级别 (CPL, Current Privilege Level) 不同。
- 返回：在目标代码段执行完成后，通常使用 retf 指令返回到调用门所在的特权级别。返回时，CPU 将恢复原来的特权级别和状态。

3 实验过程分析

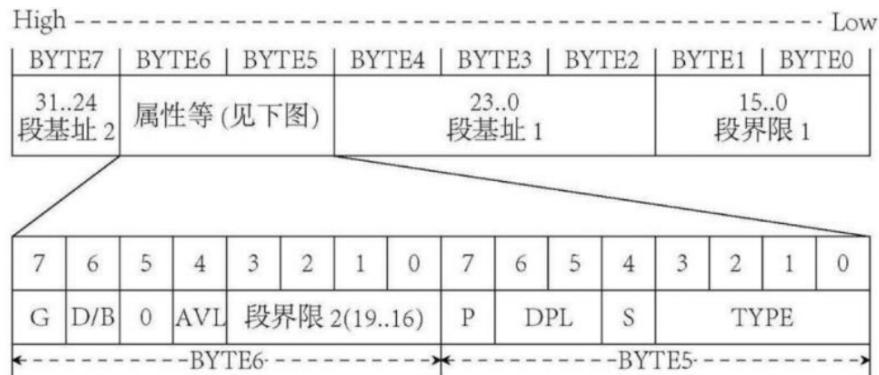
3.1 问题 1：GDT、Descriptor、Selector、GDTR 结构，及其含义是什么？他们的关联关系如何？pm.inc 所定义的宏怎么使用？

3.1.1 DT、Descriptor、Selector、GDTR 结构，及其含义

GDT： GDT 是保护模式下用于管理内存的结构，包含多个段描述符 (Descriptor)，在保护模式下提供了更灵活的内存管理机制。

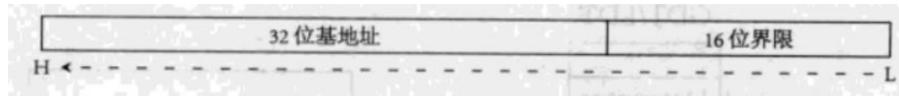
descriptor : 在保护模式下, 通过 GDT 定义多个段, 每个段有自己的基地址和限长, 保护模式提供了更灵活的内存管理机制。主要字段

- 基地址: 短的起始地址
- 段界限: 段的大小
- 段的类型和权限: 例如这是数据段、代码段, 是否可写、可执行, 以及其特权级别 (Ring 0 到 Ring 3)。



Selector: 选择子, 是一个 16 位的值, 用于在 GDT 或 LDT 中索引一个描述符, 选择子有三部分:

- 索引: GDT 中的某个描述符的编号
- TI 位: 指示选择子指向的是 GDT (值为 0) 还是 LDT (值为 1)
- RPL: 请求的特权级别, 用于权限检查



GDTR: 全局描述符表寄存器, 存储 GDT 的基地址和限长

3.1.2 关系

保护模式启用后, CPU 可以访问 GDT 来进行段式内存管理。GDT 存放所有段描述符的表, 描述了内存段的各种信息 (基地址、界限、权限等)。

Descriptor 存储在 GDT 中的每个条目, 描述某个段的具体属性。

Selector 存储在段寄存器中的值, 用于索引 GDT 或 LDT 中的描述符。GDTR 是一个特殊寄存器, 指向 GDT 在内存中的位置。

3.1.3 pm.inc 所定义的宏

pm.inc 中定义的宏有 Descriptor 和 Gate 两种。

```
1 ; 描述符
2 ; usage: Descriptor Base, Limit, Attr
3 ;     Base: dd
4 ;     Limit: dd (low 20 bits available)
5 ;     Attr: dw (lower 4 bits of higher byte are always 0)
6
7 %macro Descriptor 3
8     dw %2 & OFFFFh          ; 段界限1
9     dw %1 & OFFFFh          ; 段基址1
10    db (%1 >> 16) & OFFh   ; 段基址2
11    dw ((%2 >> 8) & OF00h) | (%3 & OF0FFh); 属性1 + 段界限2 + 属性2
12    db (%1 >> 24) & OFFh   ; 段基址3
13 %endmacro ; 共 8 字节
14 ;
15 ; 门
16 ; usage: Gate Selector, Offset, DCount, Attr
17 ;     Selector: dw
18 ;     Offset: dd
19 ;     DCount: db
20 ;     Attr: db
21
22 %macro Gate 4
23     dw (%2 & OFFFFh)        ; 偏移1
24     dw %1                   ; 选择子
25     dw (%3 & 1Fh) | ((%4 << 8) & OF00h) ; 属性
26     dw ((%2 >> 16) & OFFFFh)           ; 偏移2
27 %endmacro ; 共 8 字节
;
```

对于 Descriptor

```
1 %macro Descriptor 3 ;宏
2     dw %2 & OFFFFh          ; 段界限1 意思是宏的第二个参数传入，并
3     ; 将高16位截掉，表示段界限
4     ; 将段界限的低16位存储在描述符的低两字节中。
5     dw %1 & OFFFFh          ; 段基址1
6     ; 将段基址的低16位存储在描述符的后两字节中。
7     db (%1 >> 16) & OFFh   ; 段基址2
8     ; 将段基址的中间8位（第17到24位）存储在描述符中。
9     dw ((%2 >> 8) & OF00h) | (%3 & OF0FFh); 属性1 + 段界限2 + 属性2
10    ; 这一步将段界限的高4位和描述符属性存储在描述符的合适位置
11    db (%1 >> 24) & OFFh   ; 段基址3
12    ; 将段基址的最高8位存储在描述符的最后一个字节中。
13 %endmacro ; 共 8 字节
```

通过使用位运算来拆分和组合段基址和段界限，并添加段属性，使得段描述符能够存储足够的信息来描述内存段。这些字节最终会被放置在全局描述符表（GDT）或局部描述符表（LDT）中，用于内存管理和保护模式的地址转换。

对于 Gate

```
1 %macro Gate 4
2     dw  (%2 & OFFFFh)          ; 偏移 1
3     dw  %1                      ; 选择子
4     dw  (%3 & 1Fh) | ((%4 << 8) & OFF00h) ; 属性
5     dw  ((%2 >> 16) & OFFFFh)       ; 偏移 2
6 %endmacro ; 共 8 字节
```

通过使用位运算来拆分和组合了目标地址的偏移、段选择子、门参数数量以及门的属性，以便能够完整地构造一个门描述符。这些字节最终将被存储在中断描述符表 (IDT) 或全局描述符表 (GDT) 中，用于处理中断、异常和跨权限级别的函数调用，实现对硬件中断和系统调用的有效管理。

3.2 问题 2：从实模式到保护模式，关键步骤有哪些？为什么要关中断？为什么要打开 A20 地址线？从保护模式切换回实模式，又需要哪些步骤？

3.2.1 实模式到保护模式的关键步骤：

- 初始化 GDT 描述符；
- 加载 gdtr；
- 打开 A20 地址线；
- 设置寄存器 CR0 的 PE 位为 1，使之运行于保护模式；
- 执行跳转指令，让系统进入保护模式

3.2.2 为什么要进行关中断：

关中断是为了确保在切换到保护模式时，不会发生中断处理程序的执行。这是因为在实模式下，中断处理程序的地址是通过中断向量表中的中断向量来确定的，而在保护模式下，中断处理程序的地址是通过中断描述符表 (IDT) 中的中断门来确定的。如果在切换过程中发生中断，处理器可能会尝试执行错误的中断处理程序，导致系统异常。

3.2.3 为什么要打开 A20 地址线：

A20 地址线是用于扩展地址总线的一条线路。在实模式下，8086 处理器只能访问 1MB 的物理内存空间，而在保护模式下，处理器可以访问更大的内存空间。打开 A20 地址线可以使处理器能够访问超过 1MB 的内存，以充分利用保护模式下的内存管理功能。

3.2.4 保护模式切换回实模式的步骤：

- 关中断：与从实模式到保护模式的切换一样，切换回实模式前需要关中断。
- 清除控制寄存器：将控制寄存器 CR0 的第 0 位清零，以禁用保护模式。
- 刷新段选择子：在切换回实模式后，需要刷新所有段选择子，以便它们指向实模式下的段描述符。
- 重新加载段寄存器：由于实模式下的段寄存器与保护模式下的段寄存器不同，需要重新加载段寄存器，以确保正确访问内存。

- 关闭 A20 地址线：如果在保护模式下打开了 A20 地址线，切换回实模式时可能需要关闭它，以限制对 1MB 内存空间的访问。

3.3 问题 3：解释不同权限代码的切换原理，call, jmp, retf 使用场景如何，能够互换吗？

3.3.1 call 指令

切换原理：call 指令可以调用一个过程或函数，并将控制权转移到目标代码段中的指定地址。当使用 call 指令时，处理器会将当前代码段的返回地址（即下一条指令的地址）压入堆栈，并跳转到目标代码段中的指定地址。可以调用并切换权限，支持权限级别的自动切换及返回。

使用场景：常用于实现函数调用、子程序调用和中断处理等场景。

3.3.2 jmp 指令

切换原理：jmp 指令可以无条件跳转到目标代码段中的指定地址。与 call 指令不同的是，jmp 指令不会将返回地址压入堆栈，因此无法直接实现权限级别的切换。但是，可以通过在目标代码段中设置适当的段选择子，通过 ‘jmp dword 选择子: 0’ 的方式，可以实现从一个权限级别的代码段跳转到另一个权限级别的代码段。可以实现代码的任意跳转，但没有返回机制。

使用场景：常用于实现循环、条件语句和跳转表等场景。

3.3.3 retf 指令

切换原理：retf 指令用于从过程或函数返回，并将控制权转移到调用者的代码段中的指定地址。retf 指令会从堆栈中弹出返回地址，并跳转到该地址。与 call 指令类似，retf 指令可以在不同的权限级别之间进行切换。可以返回调用方并恢复之前的权限级别。

使用场景：常用于实现函数返回、中断返回和任务切换等场景。

3.3.4 互换

- call 和 jmp 的互换
 - 调用子程序但不需要返回，可以使用 jmp 代替 call
 - 递归或循环调用，可以用 jmp 代替 call
- jmp 和 retf 的互换
 - 实现无条件跳转或退出且不需要返回到原调用位置，可以使用 jmp 而不需要 retf 来恢复返回地址。

3.4 动手改 1：自定义添加 1 个 GDT 代码段、1 个 LDT 代码段，GDT 段内要对一个内存数据结构写入一段字符串，然后 LDT 段内代码段功能为读取并打印该 GDT 的内容

该问题参考 pmtest2.asm 和 pmtest3.asm 文件进行修改，拟打印的字符串为团队成员的名字，具体步骤如下：

1. 修改 [SECTION .data1] 数据段中内容。

将 StrTest 中的内容修改为 StrTest: db “We are a team! Huang Dongwei, Wang Junjie, Cheng Xv, Zhou Yeying” , 0

```
StrTest:      db  "We are a team! Huang Dongwei, Wang Junjie, Cheng Xu, Zhou Yeying ", 0
```

2. 编写 GDT 代码段

在 pmtest3.asm 的 [SECTION .S32] 代码段基础上，call DispReturn 和 Load LDT 之间加上调用函数 call TestRead 和 call TestWrite

```
call      DispReturn  
  
call      TestRead  
call      TestWrite  
  
; Load LDT  
mov ax, SelectorLDT  
lldt    ax
```

3. 编写 LDT 代码段，用来读取 GDT 内容并打印

```
; 显示一个字符串  
mov ah, 0Ch           ; 0000: 黑底     1100: 红字  
xor esi, esi  
xor edi, edi  
mov esi, OffsetStrTest  
mov edi, (80 * 12 + 0) * 2 ; 屏幕第 12 行，第 0 列。  
cld  
.1:  
lodsb  
test al, al  
jz  .2  
mov [gs:edi], ax  
add edi, 2  
jmp  .1  
.2:  
call DispReturn  
call TestRead  
; 准备经由16位代码段跳回实模式  
jmp SelectorCode16:0
```

4. 编译运行，查看输出结果

3.5 动手改 2：自定义 2 个 GDT 代码段 A、B，分属于不同特权级，功能自定义，要求实现 A→B 的跳转，以及 B→A 的跳转。

该问题参考 pmtest5.asm 文件进行修改，代码段 A 输出结果字符串”A”，代码段 B 输出结果为字符串” B “，具体步骤如下：

1. 在 LABEL_CODE_RING3 代码段中（即打印 B 的部分），添加一段逻辑来跳回执行 A。可以通过设置一个返回跳转来实现。



```

1 ; CodeRing3
2 [SECTION .ring3]
3 ALIGN 32
4 [BITS 32]
5 LABEL_CODE_RING3:
6     mov ax, SelectorVideo
7     mov gs, ax           ; 视频段选择子(目的)
8
9     mov edi, (80 * 14 + 0) * 2 ; 屏幕第 14 行, 第 0 列。
10    mov ah, 0Ch            ; 0000: 黑底      1100: 红字
11    mov al, 'B'
12    mov [gs:edi], ax
13
14    ; 改为使用调用门回到 LABEL_CODE_A
15    call SelectorCallGateTest:0 ; 调用门到 LABEL_SEG_CODE_DEST
16    (现改为跳回 'A')
17
18    jmp $ ; 无限循环, 保持状态
19 SegCodeRing3Len equ $ - LABEL_CODE_RING3
20 ; END of [SECTION .ring3]

```

2. 修改调用门的目标，使 A 段的代码可以再次被执行。

```

1 ; 修改调用门目标段
2 [SECTION .sdest]; 调用门目标段
3 [BITS 32]
4
5 LABEL_SEG_CODE_DEST:
6     mov ax, SelectorVideo
7     mov gs, ax           ; 视频段选择子(目的)
8
9     mov edi, (80 * 13 + 0) * 2 ; 屏幕第 13 行, 第 0 列。
10    mov ah, 0Ch            ; 0000: 黑底      1100: 红字
11    mov al, 'A'
12    mov [gs:edi], ax
13
14    ; Load LDT

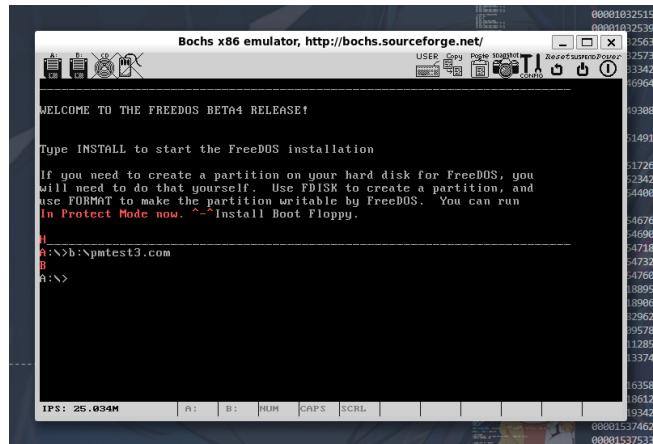
```

```

15     mov ax, SelectorLDT
16     lldt    ax
17
18     ; 调用门执行完，继续跳回到 `LABEL_CODE_RING3` 段
19     jmp SelectorLDTCodeA:0 ; 跳入局部任务，将打印字母 'A'。
20
21 SegCodeDestLen equ $ - LABEL_SEG_CODE_DEST
22 ; END of [SECTION .sdest]

```

3. 编译运行，输出结果。



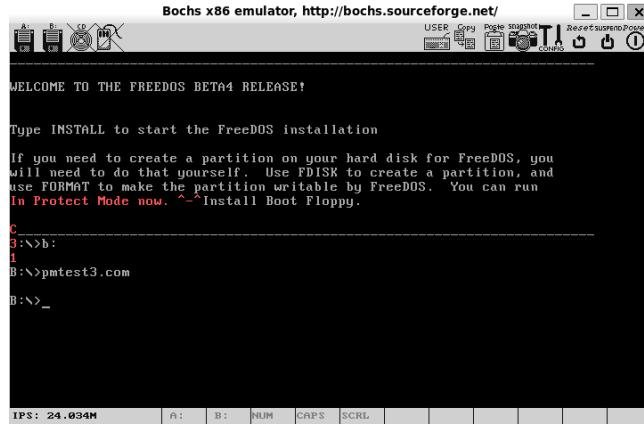
3.6 实验问题与故障分析：

问题一陈述：将 pmtest1.com 复制到虚拟软驱 pm.img 报错：mount point /mnt/floppy does not exist

```
mount: mount point /mnt/floppy does not exist
```

解决方案：原因在于在/mnt 目录下不存在 floppy 文件，只需要指令 mkdir 创建即可，但要进入管理员模式才能在/mnt 目录下建立文件夹；输入指令 sudo 进入超级管理员模式输入密码，即可完成创建

问题二陈述：在运行 bochs 时，不小心在终端通过 Ctrl+Z 将进程强行终止之后，bochs 窗口无法关闭



解决方案：使用指令 ps -e | grep bochs 查找 bochs 进程的 pid，再通过 kill -9 pid 来将进程杀死，通过这样的方法将 bochs 窗口关闭。

```
elephant@Zhouyeying ~/0/0/c/c/diy> ps -e | grep bochs
105322 pts/5    00:00:12 bochs
elephant@Zhouyeying ~/0/0/c/c/diy> kill -9 105322
fish: Job 1, 'bochs' terminated by signal SIGKILL (Forced quit)
```

问题三陈述：编译发现报错 no bootable device，查找资料找到的类似错误的解决方案无效。

```
00001054732i [ACPI] new SM base address: 0xb100
00001054760i [PCI] setting SDRAM control register to 0x4a
00001218895i [CPU0] Enter to System Management Mode
00001218906i [CPU0] RSM: Resuming from System Management Mode
00001382962i [PCI] setting SDRAM 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=0xffff
00001416358i [BIOS] Firmware waking vector 0xffff00cc
00001418612i [PCI] i440FX PMC write to PAM register 59 (TLB Flush)
00001419342i [BIOS] bios_table_cur_addr: 0x000f9fc4
00001537462i [BIOS] VGA Bios $Id: vgabios.c 288 2021-05-28 19:05:28Z vruppert $
00001537533i [VXGA] VBE known Display Interface b6c8
00001537565i [VXGA] VBE known Display Interface b6c5
00001540208i [VBIOS] VBE Bios $Id: vbe.c 292 2021-06-03 12:24:22Z vruppert $
00014065605p [BIOS] >>PANIC<< No bootable device.
=====
Bochs is exiting with the following message:
[BIOS] No bootable device.
=====
00014065605i [CPU0] CPU is in real mode (active)
00014065605i [CPU0] CS.mode = 16 bit
00014065605i [CPU0] SS.mode = 16 bit
00014065605i [CPU0] EFER = 0x00000000
00014065605i [CPU0] | EAX=0000040a EBX=0000cd24 ECX=00090004 EDX=00000402
00014065605i [CPU0] | ESP=0000ffa8 EBP=0000ffac ESI=000e0000 EDI=0000ffac
00014065605i [CPU0] | IOPL=0 id vip vif ac vm rf nt of df if tf sf ZF af Pf cf
00014065605i [CPU0] | SEG s1tr(index|til|rpl) base limit G D
00014065605i [CPU0] | CS:f000( 0004| 0) 000f0000 000fffff 0 0
00014065605i [CPU0] | DS:f000( 0005| 0) 000f0000 000fffff 0 0
00014065605i [CPU0] | SS:f000( 0005| 0) 00000000 000fffff 0 0
00014065605i [CPU0] | ES:07c0( 0005| 0) 00007c00 000fffff 0 0
00014065605i [CPU0] | FS:0000( 0005| 0) 00000000 000fffff 0 0
00014065605i [CPU0] | GS:0000( 0005| 0) 00000000 000fffff 0 0
00014065605i [CPU0] | EIP=0000054b( 0000054a)
00014065605i [CPU0] | CR0=0x60000018 CR2=0x00000000
00014065605i [CPU0] | CR3=0x00000000 CR4=0x00000000
(0).[14065605] [0x0000000f054a] f000:054a (unk. ctxt): out dx, al ; ee
00014065605i [CMOS] Last time is 1727632118 (Mon Sep 30 01:48:38 2024)
00014065605i [SIM] quit_sim called with exit code 1
make: *** [Makefile:17: run] Error 1
```

解决方案：镜像文件有缺损，无法使用，需要重新下载。

4 实验结果总结

(对实验结果进行分析，完成思考题目，并提出实验的改进意见)

根据实验步骤，我们总结了以下 OS 原理的相关知识点：

1. 保护模式与实模式的区别：保护模式下启用了段机制，可以实现内存保护、特权级保护等；而实模式下没有这些保护机制。
2. 关键数据结构：GDT(全局描述符表)、LDT(局部描述符表)、描述符、选择子等。描述符定义了段的属性；选择子用于定位描述符。
3. 段选择子是一个 16 位的寄存器，低 3 位中的 RPL 用于特权检查，TI 引用描述符表指示位， $TI=0$ 指示从全局描述符表 GDT 中读取描述符， $TI=1$ 指示从局部描述符表 LDT 中读取描述符。
4. 从实模式进入保护模式的步骤：关中断，打开 A20 地址线，加载 GDT 表，将 CR0 的 PE 位置 1。这些步骤确保进入保护模式时系统状态正确。
5. 在保护模式下，如果 A20 被打开，则可以访问的内存则是连续的；如果 A20 被关闭，则能访问的内存只能是偶数段。
6. 段的特权级保护：代码段的 CPL、数据段的 DPL 和选择子的 RPL 决定了段的访问权限。
7. 调用门可以在不同特权级之间转移，实现特权级的变换。
8. CR0 控制寄存器的 PE 位决定了 CPU 的模式： $PE=0$ 为实模式， $PE=1$ 为保护模式。
9. 段描述符中定义了段的基地址、限长、访问权限等信息。
10. 从保护模式切回实模式，需要清除 CR0 的 PE 位。
11. 加载不同的 LDT 可以实现不同的地址空间。
12. 实验中通过自定义的 GDT 和 LDT 演示了段机制的基本应用。

通过这个实验加深了对保护模式和段机制相关概念的理解，也对 OS 内存保护的原理有了直观的了解。实验中的代码对应并体现了理论知识中的关键要点。

5 各人实验贡献与体会（每人各自撰写）

1. 黄东威：负责解决实验目的及实验内容，独立完成大部分实验，进行了实验报告的编写
个人体会：

- 从实模式到保护模式的转变：我学会了如何从实模式切换到保护模式。这个过程是理解操作系统如何工作的基础，因为它涉及到对内存和权限的灵活管理。
- 特权级别的作用：通过实验，我更清楚地了解了特权级别的概念。它们是计算机系统中的关键要素，决定了哪些代码可以访问哪些资源，因此对于操作系统和软件开发非常重要。
- 调用门的奥秘：我学会了如何使用调用门来实现不同特权级别之间的通信。这是一项关键技能，实现了低特权级到高特权级的高效转换。
- 错误排除能力：当遇到问题时，我必须花费很多的时间去 debug，这方面能力的培养非常重要，对未来的工程开发有非常大的帮助。

复习与巩固：这次实验让我学习了很多关于操作系统的理论知识，也回顾了前面 bochs 相关的内容，锻炼了我通过查找资料和教材解决问题的能力，也提高了我和同学间互相交流协作的能力。

2. 周业营：调试运行实验代码，负责实验解决问题与动手改任务、实验问题与故障分析部分实验报告的撰写

个人体会：本次实验相比上次更加耗时，因为需要阅读更多的书本内容，初次接触 X86 汇编，还要看懂大篇幅的汇编代码。通过本次实验，我学习到了 GDT、LDT 和其描述符及选择子的作用和结构，对保护模式和段式存储机制有了更深入的了解。编写与调试添加的汇编代码，增强了我的动手能力，让我切实体会到了实模式和保护模式下汇编指令运行的流程。同时，在与小组成员一起讨论如何解决问题的过程中也增加了团队合作能力和问题分析解决能力。

3. 程序：负责实验 GDT、LDT 部分的代码调试，补充部分实验问题与故障分析。

个人体会：本次实验涉及的参考资料内容较多，且关系较复杂，需要通过大段代码来理解保护模式和实模式的切换，特权级之间的切换等等。通过本次实验，我理解了 descriptor、selector、GDT、LDT 等数据结构，以及操作系统中的环保护机制，不同段之间的访问规则。同时，通过调试代码的实践，进一步加深了对代码的理解并了解了各种报错异常的应对方法，为学习后面的知识打下基础。

4. 王浚杰：负责从实模式到保护模式互相切换的基本流程，通过反汇编和调试解决了跳转问题，验证了错误跳转可能导致的程序崩溃或中断异常。同时掌握了 GDT 的构造方法，成功实现了 GDT 的切换、段寄存器的恢复和 A20 地址线的关闭，确保程序在保护模式执行后能够正确返回到 DOS 实模式，顺利完成内存写操作并重启系统。

个人体会：通过此次任务，我深入理解了 x86 处理器从实模式到保护模式的切换机制，掌握了 GDT 的构造和加载过程，弄清关键数据结构：GDT、descriptor、selector、GDTR，及其之间关系。通过不断的汇编调试，我理解了保护模式下的段寄存器不再是线性地址，而是通过选择子指向 GDT 中的段描述符，对 x86 架构下的内存管理和模式切换有了更深入的认识。

6 教师评语

（实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评）

分数	评语
85-100	<ul style="list-style-type: none"> • 实验内容完整或者有超出课程实验大纲的内容； • 实验步骤详尽，能够体现完整的实验过程； • 实验结果正确且实验数据分析得当； • 实验心得与思考全面并且有自己的独立思考； • 实验报告文档规范、排版整齐。
75-84	<ul style="list-style-type: none"> • 实验内容较为完整； • 实验步骤较为详尽，能够体现实验过程； • 实验结果正确且实验数据分析较为得当； • 实验心得与思考全面； • 实验报告文档规范、排版较为整齐。
60-74	<ul style="list-style-type: none"> • 实验内容有缺失； • 实验步骤不够详尽，不能够体现完整的实验过程； • 实验结果部分正确； • 实验心得与思考无或者不够深入； • 实验报告文档规范性有待增强。
60 以下	<ul style="list-style-type: none"> • 实验内容严重缺失、实验态度不够端正； • 实验步骤不够详尽，不能够体现完整的实验过程； • 实验结果部分正确； • 实验心得与思考无或者不够深入； • 实验报告文档规范性有待增强。

姓名	学号	分数
黄东威	2022302181148	
王浚杰	2022302181143	
程序	2022302181131	
周业营	2022302181145	

教师评分（请填写好姓名、学号）

教师签名：

年 月 日