



中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程: 操作系统原理实验

任课教师: 刘宁

实验题目: 从实模式到保护模式

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024 年 3 月 25 日

Section 1 实验概述

在上一节中，我们已经学习了 x86 汇编的相关内容、初步了解计算机的启动过程并在 16 位的实模式环境下进行编程。在本节中，将会学习到如何从 16 位的实模式跳转到 32 位的保护模式，然后在平坦模式下运行 32 位程序。同时，同学们将学习到如何使用 I/O 端口和硬件交互，为后面保护模式编程打下基础。

Section 2 预备知识与实验环境

- 预备知识：x86 汇编语言程序设计、IA-32 处理器体系结构，LBA 方式读写硬盘和 CHS 方式读写硬盘的相关知识。
- 实验环境：
 - 虚拟机版本/处理器型号：
11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2
 - 代码编辑环境：VS Code
 - 代码编译工具：g++
 - 重要三方库信息：Linux 内核版本号：linux-5.10.210
Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：
Busybox_1_33_0

Section 3 实验任务

- 实验任务 1：
课后思考题第 9 题，复现“加载 bootloader”这一节，说说你是怎么做的并提供结果截图，也可以参考 Ucore、Xv6 等系统源码，实现自己的 LBA 方式的磁盘访问。
- 实验任务 2：
在实验任务一的基础上完成课后思考题第 10 题。
- 实验任务 3：
完成课后思考题 11 并利用 gdb 调试解释和分析
- 实验任务 4：
进入保护模式后，编写汇编代码按要求改变前景和背景色显示学号和姓名缩写。

Section 4 实验步骤与实验结果

----- 实验任务 1 -----

- 任务要求：课后思考题第 9 题，复现“加载 bootloader”这一节。
- 思路分析：阅读指导书相关内容，学习如何将输出部分放在 bootloader 中，然后再 MBR 中加载 bootloader 到内存。

MBR 和 bootloader 在内存的位置如下：

name	start	length	end
MBR	0x7c00	0x200(512B)	0x7e00
bootloader	0x7e00	0xa00(512B * 5)	0x8800

- 实验步骤：

1. 我们先新建一个文件 bootloader.asm，然后将 lab2 的 mbr.asm 中输出 Hello World 部份的代码，放入 bootloader.asm。

Bootloader 代码如下：

```
org 0x7e00
[bits 16]
mov ax, 0xb800
mov gs, ax
mov ah, 0x03 ;青色
mov ecx, bootloader_tag_end - bootloader_tag
xor ebx, ebx
mov esi, bootloader_tag
output_bootloader_tag:
    mov al, [esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    loop output_bootloader_tag
jmp $ ; 死循环
bootloader_tag db 'run bootLoader'
bootloader_tag_end:
```

2. 然后我们在 mbr.asm 处放入使用 LBA 模式读取硬盘的代码，然后在 MBR 中加载 bootloader 到地址 0x7e00。

MBR.asm 代码如下：

```
org 0x7c00
[bits 16]
xor ax, ax ; eax = 0
; 初始化段寄存器，段地址全部设为 0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax
```

```

; 初始化栈指针
mov sp, 0x7c00
mov ax, 1          ; 逻辑扇区号第0~15 位
mov cx, 0          ; 逻辑扇区号第16~31 位
mov bx, 0x7e00     ; bootLoader 的加载地址
load_bootloader:
    call asm_read_hard_disk ; 读取硬盘
    inc ax
    cmp ax, 5
    jle load_bootloader
    jmp 0x0000:0x7e00      ; 跳转到bootLoader
    jmp $ ; 死循环
asm_read_hard_disk:
; 从硬盘读取一个逻辑扇区
; 参数列表
; ax=逻辑扇区号 0~15 位
; cx=逻辑扇区号 16~28 位
; ds:bx=读取出的数据放入地址
; 返回值
; bx=bx+512
    mov dx, 0x1f3
    out dx, al      ; LBA 地址 7~0
    inc dx          ; 0x1f4
    mov al, ah
    out dx, al      ; LBA 地址 15~8
    mov ax, cx
    inc dx          ; 0x1f5
    out dx, al      ; LBA 地址 23~16
    inc dx          ; 0x1f6
    mov al, ah
    and al, 0x0f
    or al, 0xe0     ; LBA 地址 27~24
    out dx, al
    mov dx, 0x1f2
    mov al, 1
    out dx, al      ; 读取 1 个扇区
    mov dx, 0x1f7   ; 0x1f7
    mov al, 0x20     ; 读命令
    out dx, al
    ; 等待处理其他操作
.waits:
    in al, dx        ; dx = 0x1f7
    and al, 0x88
    cmp al, 0x08
    jnz .waits
    ; 读取 512 字节到地址 ds:bx
    mov cx, 256     ; 每次读取一个字, 2 个字节, 因此读取 256 次即可
    mov dx, 0x1f0
.readw:
    in ax, dx
    mov [bx], ax
    add bx, 2
    loop .readw

```

```
ret
times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

3. 然后我们编译 bootloader.asm，写入硬盘起始编号为 1 的扇区，共有 5 个扇区。

在终端使用以下命令：

```
nasm -f bin bootloader.asm -o bootloader.bin
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
```

4. mbr.asm 也要重新编译和写入硬盘起始编号为 0 的扇区。

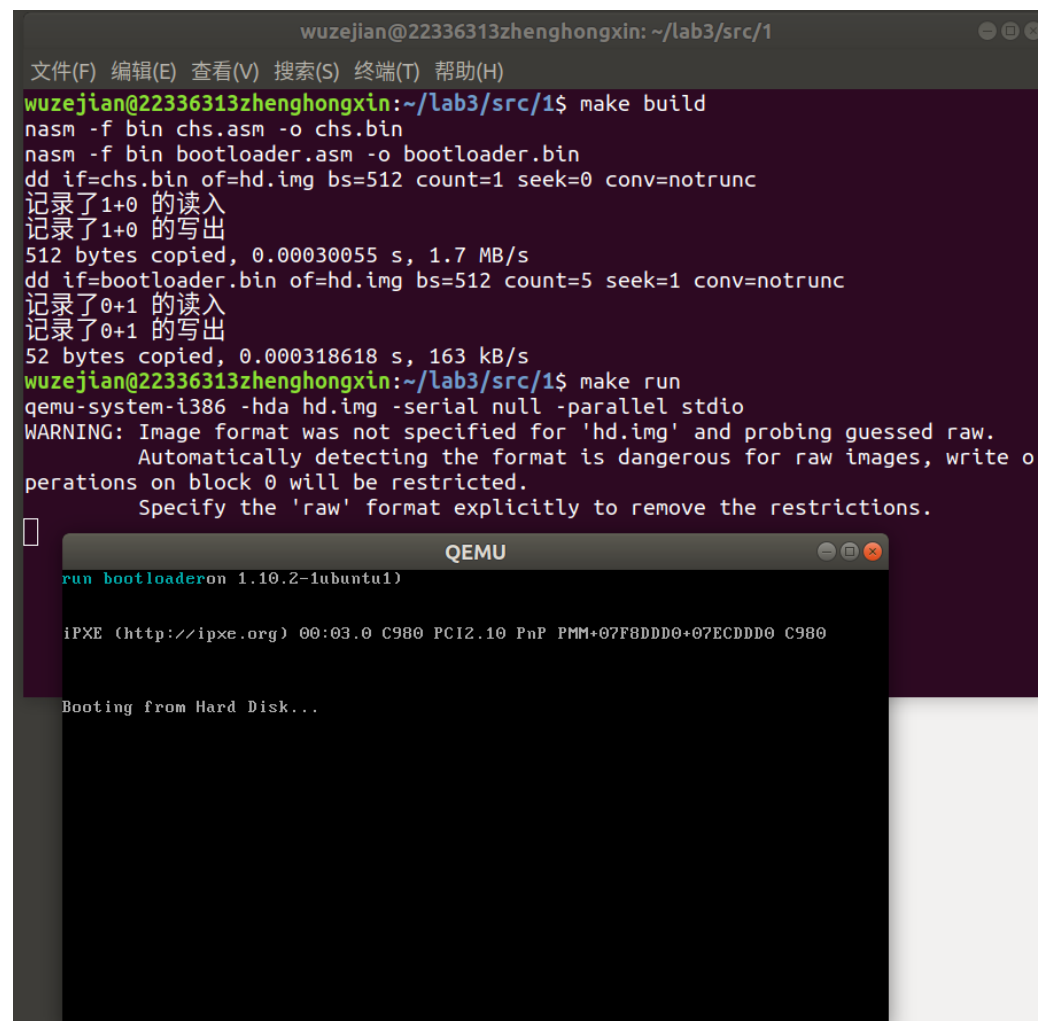
在终端使用以下命令：

```
nasm -f bin mbr.asm -o mbr.bin
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

5. 最后启动 qemu 运行即可。

利用 makefile 可以使用 make build 进行编译，make run 进行运行。

● 实验结果展示：



The screenshot shows a terminal window with the following commands and output:

```
wuzejian@22336313zhenghongxin: ~/lab3/src/1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
wuzejian@22336313zhenghongxin:~/lab3/src/1$ make build
nasm -f bin chs.asm -o chs.bin
nasm -f bin bootloader.asm -o bootloader.bin
dd if=chs.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.00030055 s, 1.7 MB/s
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
52 bytes copied, 0.000318618 s, 163 kB/s
wuzejian@22336313zhenghongxin:~/lab3/src/1$ make run
qemu-system-i386 -hda hd.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

Below the terminal window, a QEMU window is open, displaying the following text:

```
run bootloader on 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DD0+07ECDD0 C980

Booting from Hard Disk...
```

-----实验任务 2-----

- 任务要求：在实验任务一的基础上完成课后思考题第 10 题。

课后思考题 10 如下：

在"加载 bootloader"一节中，我们使用了 LBA28 的方式来读取硬盘。此时，我们只要给出逻辑扇区号即可，但需要手动去读取 I/O 端口。然而，BIOS 提供了实模式下读取硬盘的中断，其不需要关心具体的 I/O 端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可，又被称为 CHS 模式。

- 思路分析：依照指导书和参考资料了解 LBA 方式和 CHS 方式读取硬盘的方式的区别和二者如何相互转换。

参考的两个网址如下：

- LBA 向 CHS 模式的转换：[C/H/S 与 LBA 的转换关系 c/h/s 与 lba 地址的对应关系-CSDN 博客](#)
- int 13h 中断：[INT13 中断详解_int13 中断功能-CSDN 博客](#)
- 由于第一篇参考资料需要付费解锁，故重新寻找了一篇资料以学习 LBA 向 CHS 的转换：[LBA和CHS转换\(转\) - 姜大伟 - 博客园](#)
- 实验步骤：

1.利用以下公式计算柱面号：

$$\text{柱面号(Cylinder)} = \frac{\text{逻辑扇区号}}{SPT * HPC}$$

2.利用以下公式计算磁头号：

$$\text{磁头号(Head)} = \frac{\text{逻辑扇区号}}{SPT} \% HPC$$

3.利用以下公式计算扇区号：

$$\text{扇区号(Sector)} = \text{逻辑扇区号} \% SPT + 1$$

注意上述公式中 SPT 为每磁道扇区数，HPC 为每柱面磁头数，这些关键参数在课后思考题中已经给出：

参数	数值
驱动器号 (DL寄存器)	80h
每磁道扇区数	63
每柱面磁头数 (每柱面总的磁道数)	18

4.配合计算得出的柱面号，磁头号和扇区号和 int13H 中断,修改 mbr.asm 为 CHS.asm 如下所示：

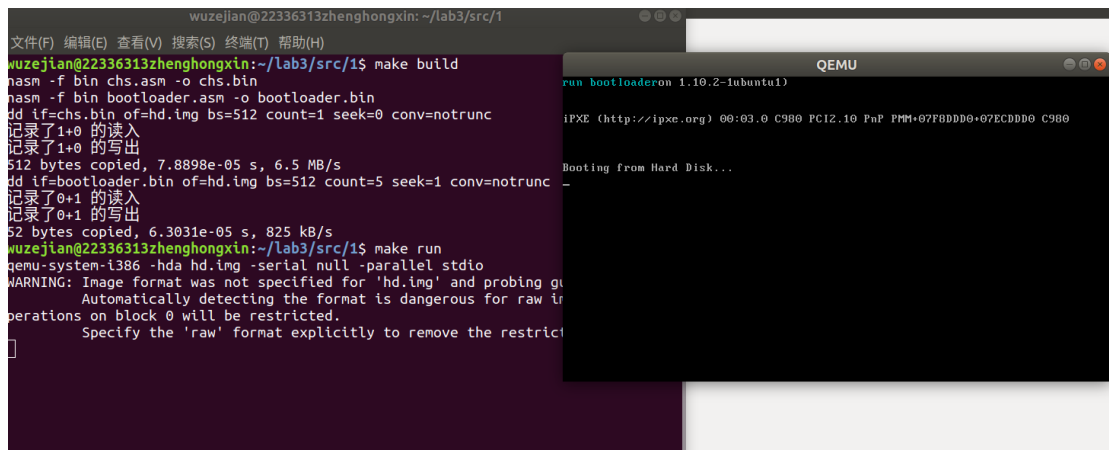
```
org 0x7c00
[bits 16]
xor ax, ax ; eax = 0
; 初始化段寄存器，段地址全部设为 0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax
; 初始化栈指针
mov sp, 0x7c00
mov bx, 0x7e00 ; 目标内存地址
mov ah, 0x02 ; 功能号：读取扇区
mov al, 5 ; 读取扇区数
mov ch, 0 ; 柱面号
mov cl, 2 ; 扇区号
mov dh, 0 ; 磁头号
int 0x13 ; 调用 BIOS 中断
jmp 0x0000:0x7e00 ; 跳转到 bootloader
jmp $ ; 死循环
times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

5.修改 makefile 代码中的 mbr.asm 为 chs.asm 如下：

```
build:
    nasm -f bin chs.asm -o chs.bin
    nasm -f bin bootloader.asm -o bootloader.bin
    dd if=chs.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
    dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1
conv=notrunc
run:
    qemu-system-i386 -hda hd.img -serial null -parallel stdio
clean:
    rm -fr *.bin
```

● 实验结果展示：

通过 make build 进行编译，make run 进行运行，得到结果如下图所示：



----- 实验任务 3 -----

- 任务要求：完成课后思考题 11 并利用 gdb 调试解释和分析

课后思考题 11 如下：

复现“进入保护模式”一节，使用 gdb 或其他 debug 工具在进入保护模式的 4 个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这 4 个步骤，最后附上结果截图。gdb 的使用可以参考 appendix 的“debug with gdb and qemu”部份。

- 思路分析：参考指导书“进入保护模式”这一节，了解进入保护模式的 4 个步骤：

- 1.准备 GDT，用 lgdt 指令加载 GDTR 信息。
- 2.打开第 21 根地址线。
- 3.开启 cr0 的保护模式标志位。
- 4.远跳转，进入保护模式。

然后配合 gdb 进行设置断点和查看寄存器的值，进行验证分析。

- 实验步骤：

- 1.首先对 mbr.asm 和 bootloader.asm 编译生成符号表：

在终端输入下列命令：

```
nasm -o mbr.o -g -f elf32 mbr.asm
ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00
ld -o mbr.bin -melf_i386 -N mbr.o -Ttext 0x7c00 --oformat binary
nasm -o bootloader.o -g -f elf32 bootloader.asm
ld -o bootloader.symbol -melf_i386 -N bootloader.o -Ttext 0x7e00
```



```
ld -o bootloader.bin -melf_i386 -N bootloader.o -Ttext 0x7e00 --oformat binary
```

注意：这一步编译之前需要删除 mbr.asm 和 bootloader.asm 的 org 语句，因为我们会在链接的过程中指定他们代码和数据的起始地址，其效果和 org 指令完全相同。

2. 然后使用 qemu 加载 hd.img 运行：

```
qemu-system-i386 -s -S -hda hd.img -serial null -parallel stdio
```

3. 在另一个终端开启 gdb：

```
gdb
```

4. 在 gdb 的命令窗口中连接 qemu 和设置断点（断点在第一条指令的位置，键入 c 运行至断点处）

```
target remote:1234
b *0x7c00
c
```

5. 打开可以显示源代码的窗口：

```
layout src
```

6. 由于没有加载符号表，所以看不见我们的源代码，为了后续方便 debug，加载符号表：

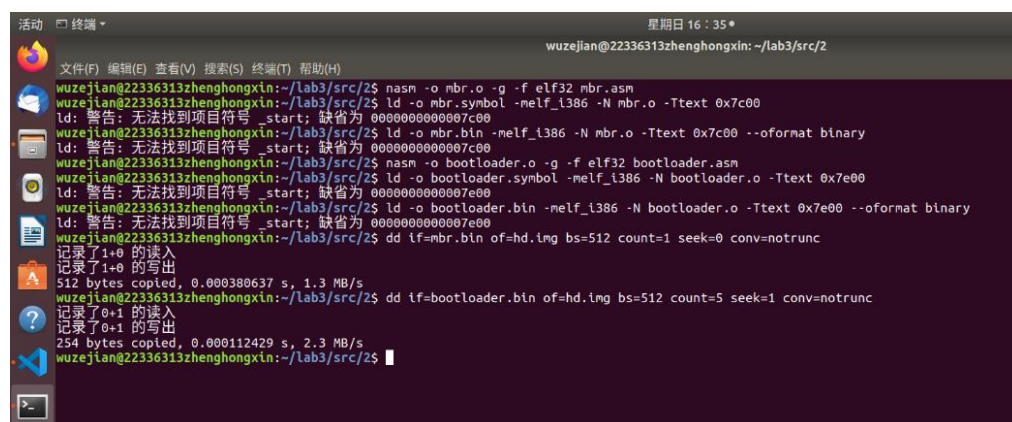
```
add-symbol-file mbr.symbol 0x7c00
add-symbol-file bootloader.symbol 0x7e00
```

7. 查看 GDT 中 5 个段描述符的内容：

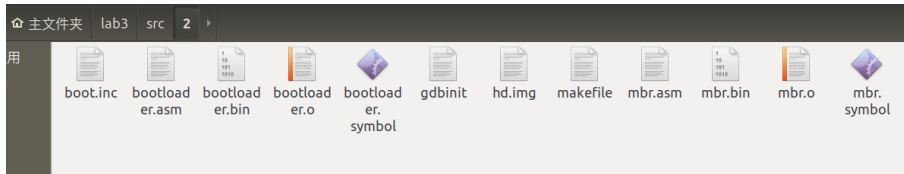
```
x/5xg 0x8800
```

● 实验结果展示：

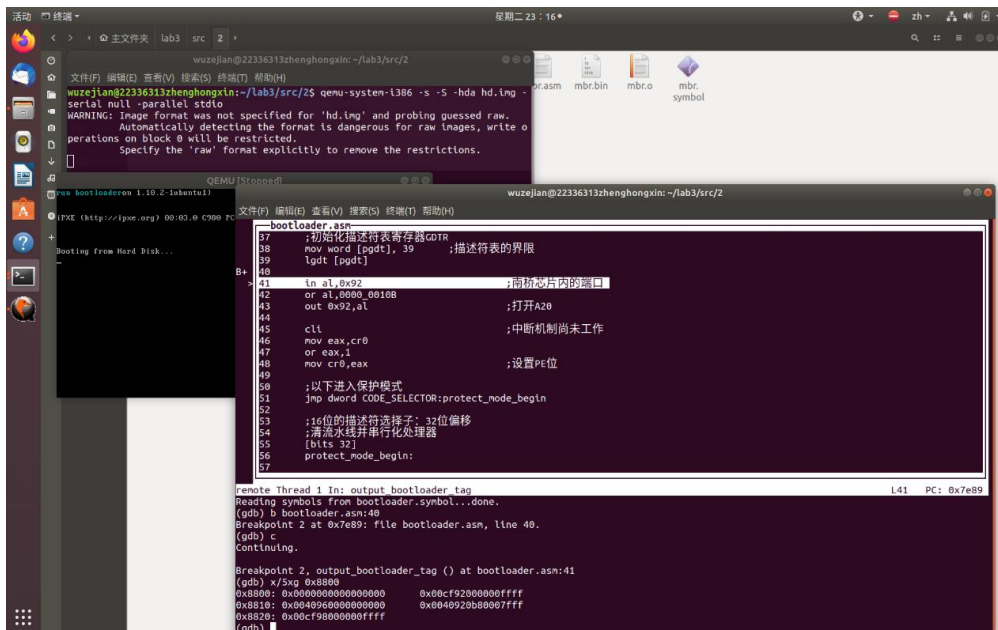
生成符号表：



```
wuzejian@22336313zhenghongxin: ~/lab3/src/2
wuzejian@22336313zhenghongxin:~/lab3/src/2$ nasm -o mbr.o -g -f elf32 mbr.asm
wuzejian@22336313zhenghongxin:~/lab3/src/2$ ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00
ld: 警告: 无法找到项目符号 _start; 缺省为 00000000000007c00
wuzejian@22336313zhenghongxin:~/lab3/src/2$ ld -o mbr.bin -melf_i386 -N mbr.o -Ttext 0x7c00 --oformat binary
ld: 警告: 无法找到项目符号 _start; 缺省为 00000000000007c00
wuzejian@22336313zhenghongxin:~/lab3/src/2$ nasm -o bootloader.o -g -f elf32 bootloader.asm
ld: 警告: 无法找到项目符号 _start; 缺省为 00000000000007e00
wuzejian@22336313zhenghongxin:~/lab3/src/2$ ld -o bootloader.symbol -melf_i386 -N bootloader.o -Ttext 0x7e00
ld: 警告: 无法找到项目符号 _start; 缺省为 00000000000007e00
wuzejian@22336313zhenghongxin:~/lab3/src/2$ dd if=bootloader.bin -melf_i386 -N bootloader.o -Ttext 0x7e00 --oformat binary
dd: 警告: 无法找到项目符号 _start; 缺省为 00000000000007e00
wuzejian@22336313zhenghongxin:~/lab3/src/2$ dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
dd: 警告: 无法找到项目符号 _start; 缺省为 00000000000007e00
wuzejian@22336313zhenghongxin:~/lab3/src/2$
```

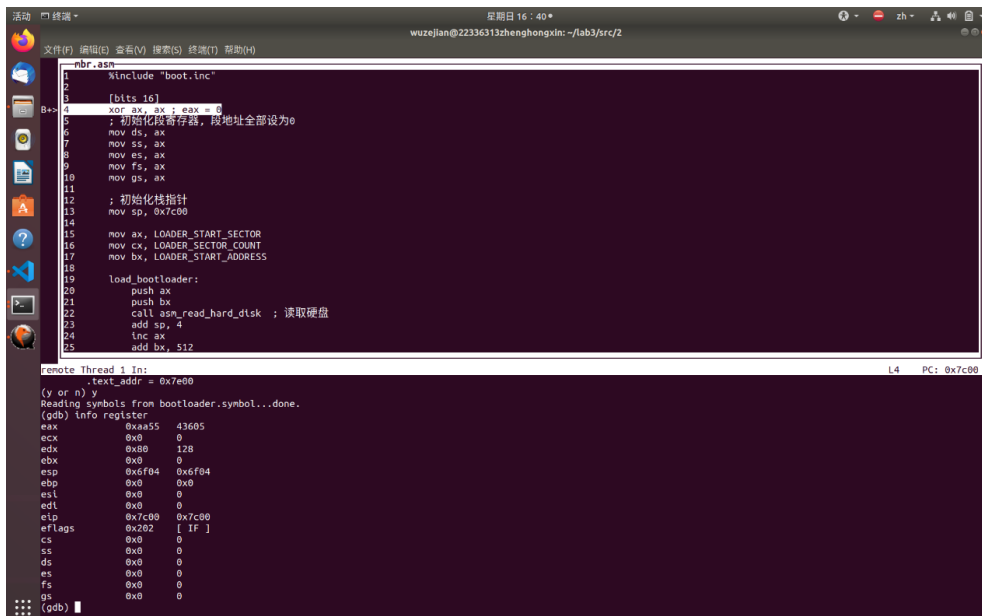


在 GDT 处查看 5 个段描述符的值：



可以看到段描述符已经变成我们设置的值。

完整的 4 个关键步骤调试的断点及寄存器信息如下：



```
活动 终端 - 星期日 16:42 *
wuzejian@22336313zhenghongxin: ~/lab3/src/2

bootloader.asm
28 ;建立保护模式下的显存描述符
29 mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x00080000, 界限0x07fff
30 mov dword [GDT_START_ADDRESS+0x1c],0x0040920b ; 粒度为字节
31
32 ;创建保护模式下平坦模式代码段描述符
33 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限为0xffff
34 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段描述符
35
36 ;初始化描述符表寄存器GDTR
37 mov word [pgdt], 39 ;描述符表的界限
38 lgdt [pgdt]
39
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010b ;打开A20
43 out 0x92,al
44
45 cll ;中断机制尚未工作
46 mov eax,cr0
47 or eax,1
48 mov cr0,eax ;设置PE位
49
50 ;以下进入保护模式
51 jmp dword CODE_SELECTOR:protect_mode_begin
52

Remote Thread 1 In: output bootloader_tag
Focus set to end of window.
(gdb) b bootloader.asm:40
Breakpoint 3 at 0x7e89: file bootloader.asm, line 40.
(gdb) info register
eax 0x372 882
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
edi 0x0 0
eip 0x7e84 0x7e84 <output_bootloader_tag+110>
eflags 0x202 [ IF ]
cs 0x0 0
ss 0x0 0
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0xb800 47104
(gdb)
```

```
活动 终端 - 星期日 16:43 *
wuzejian@22336313zhenghongxin: ~/lab3/src/2

bootloader.asm
28 ;建立保护模式下的显存描述符
29 mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x00080000, 界限0x07fff
30 mov dword [GDT_START_ADDRESS+0x1c],0x0040920b ; 粒度为字节
31
32 ;创建保护模式下平坦模式代码段描述符
33 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限为0xffff
34 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段描述符
35
36 ;初始化描述符表寄存器GDTR
37 mov word [pgdt], 39 ;描述符表的界限
38 lgdt [pgdt]
39
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010b ;打开A20
43 out 0x92,al ;打开A20
44
45 cll ;中断机制尚未工作
46 mov eax,cr0
47 or eax,1
48 mov cr0,eax ;设置PE位
49
50 ;以下进入保护模式
51 jmp dword CODE_SELECTOR:protect_mode_begin
52

Remote Thread 1 In: output bootloader_tag
Continuing.
Breakpoint 4, output_bootloader_tag () at bootloader.asm:43
(gdb) info register
eax 0x302 770
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
edi 0x0 0
eip 0x7e8d 0x7e8d <output_bootloader_tag+119>
eflags 0x202 [ IF ]
cs 0x0 0
ss 0x0 0
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0xb800 47104
(gdb)
```

可以看到在打开第 21 根地址线后，al 寄存器的值已经被修改。

```
活动 终端 - 星期日 16:47 *
wuzejian@22336313zhenghongxin: ~/lab3/src/2

bootloader.asm
28 ;建立保护模式下的显存描述符
29 mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x00080000, 界限0x07fff
30 mov dword [GDT_START_ADDRESS+0x1c],0x0040920b ; 粒度为字节
31
32 ;创建保护模式下平坦模式代码段描述符
33 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限为0xffff
34 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段描述符
35
36 ;初始化描述符表寄存器GDTR
37 mov word [pgdt], 39 ;描述符表的界限
38 lgdt [pgdt]
39
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010b ;打开A20
43 out 0x92,al
44
45 cll ;中断机制尚未工作
46 mov eax,cr0
47 or eax,1
48 mov cr0,eax ;设置PE位
49
50 ;以下进入保护模式
51 jmp dword CODE_SELECTOR:protect_mode_begin
52

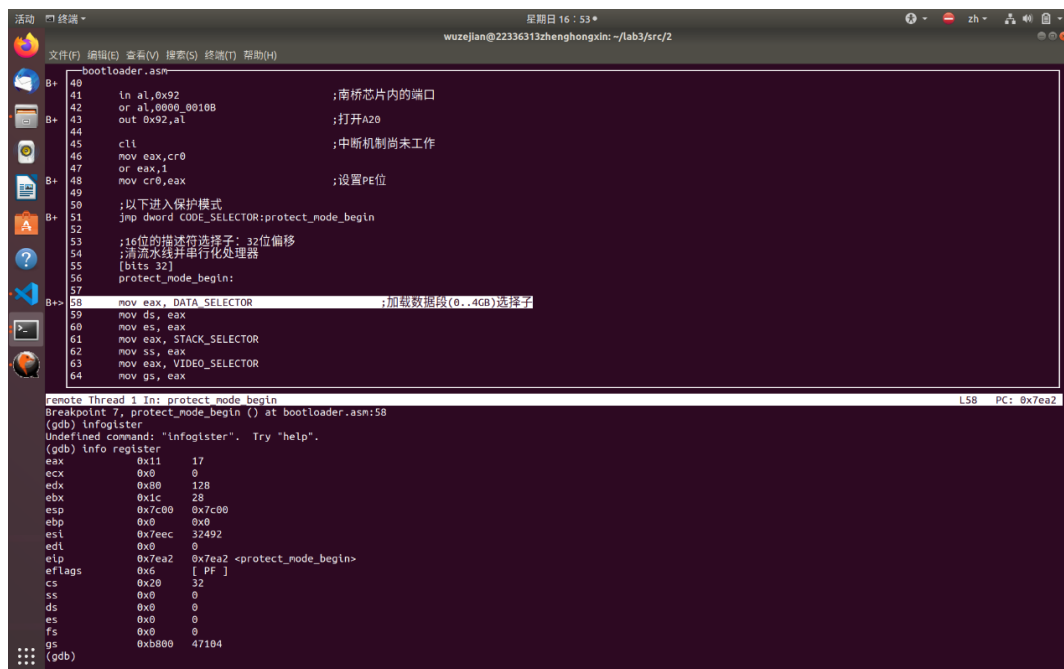
Remote Thread 1 In: output bootloader_tag
Continuing.
Breakpoint 5, output_bootloader_tag () at bootloader.asm:48
(gdb) info register
eax 0x11 17
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
edi 0x0 0
eip 0x7e97 0x7e97 <output_bootloader_tag+129>
eflags 0x6 [ PF ]
cs 0x0 0
ss 0x0 0
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0xb800 47104
(gdb)
```

可以看到 `eax` 的值已经被修改成对应的值



```
bootloader.asm
40      in al,0x92                ;南桥芯片内的端口
41      or al,0000_0010B         ;打开A20
42      out 0x92,al
43      cll                      ;中断机制尚未工作
44      mov eax,cr0
45      or eax,1
46      mov cr0,eax             ;设置PE位
47
48      ;以下进入保护模式
49      jmp dword CODE_SELECTOR:protect_mode_begin
50
51      ;16位的描述符选择子: 32位偏移
52      ;清除流水线并串行化处理器
53      [bits 32]
54      protect_mode_begin:
55
56      mov eax, DATA_SELECTOR   ;加载数据段(0..4GB)选择子
57      mov ds, eax
58      mov es, eax
59      mov eax, STACK_SELECTOR
60      mov ss, eax
61      mov eax, VIDEO_SELECTOR
62      mov gs, eax
63
64      remote Thread 1 In: output_bootloader_tag
Continuing.
Breakpoint 6, output_bootloader_tag () at bootloader.asm:51
(gdb) info register
eax             0x11      17
ecx             0x0       0
edx             0xb0      128
ebx             0x1c      28
esp             0x7c00    0x7c00
ebp             0x0        0
esi             0x7eec    32492
edi             0x0        0
eip             0x7e9a    0x7e9a <output_bootloader_tag+132>
eflags          0x6       [ PF ]
cs              0x0        0
ss              0x0        0
ds              0x0        0
es              0x0        0
fs              0x0        0
gs              0xb800    47104
(gdb)
```

可以看到正确到达原跳转指令的地址



```
bootloader.asm
40      in al,0x92                ;南桥芯片内的端口
41      or al,0000_0010B         ;打开A20
42      out 0x92,al
43      cll                      ;中断机制尚未工作
44      mov eax,cr0
45      or eax,1
46      mov cr0,eax             ;设置PE位
47
48      ;以下进入保护模式
49      jmp dword CODE_SELECTOR:protect_mode_begin
50
51      ;16位的描述符选择子: 32位偏移
52      ;清除流水线并串行化处理器
53      [bits 32]
54      protect_mode_begin:
55
56      mov eax, DATA_SELECTOR   ;加载数据段(0..4GB)选择子
57      mov ds, eax
58      mov es, eax
59      mov eax, STACK_SELECTOR
60      mov ss, eax
61      mov eax, VIDEO_SELECTOR
62      mov gs, eax
63
64      remote Thread 1 In: protect_mode_begin
Breakpoint 7, protect_mode_begin () at bootloader.asm:58
(gdb) info register
Undefined command: "info register". Try "help".
(gdb) info register
eax             0x11      17
ecx             0x0       0
edx             0xb0      128
ebx             0x1c      28
esp             0x7c00    0x7c00
ebp             0x0        0
esi             0x7eec    32492
edi             0x0        0
eip             0x7ea2    0x7ea2 <protect_mode_begin>
eflags          0x6       [ PF ]
cs              0x20      32
ss              0x0        0
ds              0x0        0
es              0x0        0
fs              0x0        0
gs              0xb800    47104
(gdb)
```

The screenshot shows a debugger window with two panes. The top pane displays assembly code from a file named `bootloader.asm`. The code includes instructions for setting up a loop to output a tag, with a comment in line 77: `jmp $; 死循环`. The bottom pane shows a GDB register dump after a SIGINT signal. The `eax` register contains `0x355`, and the `edi` register contains `0x7ed6`, which is noted as `<output_protect_mode_tag+12>`. Other registers like `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `ebp`, `edi`, `eflags`, `cs`, `ss`, `ds`, `es`, `fs`, and `gs` are also listed with their values.

通过 `info register` 命令可以看到，段寄存器的内容变成了段选择子。

----- 实验任务 4 -----

- 任务要求：

在进入保护模式后，按照如下要求，编写并执行一个自己定义的 32 位汇编程序。使用两种不同的自定义颜色和一个自定义的起始位置(x,y)，使得 `bootloader` 加载后，在显示屏坐标(x,y)处开始输出自己的学号+姓名拼音首字母缩写，要求相邻字符前景色和背景色必须是相互对调的。

- 思路分析：通过修改 `bootloader.asm` 中的部分内容以完成实验。

- 实验步骤：

修改的部分如下所示：

```
output_protect_mode_tag:
    ; 检查 esi 中的值是否为偶数
    mov eax, edx ; 将 esi 指向的值移动到 eax 寄存器
    and eax, 1   ; 使用 AND 指令与 1 进行与操作，结果在 eax 的最低位
    jz even_number ; 如果结果为 0（偶数），跳转到 even_number 标签
    ; 如果是奇数，设置 ah 为 0x31
    mov ah, 0x17
    jmp print; 跳过下面的代码，直接结束检查
even_number:
    ; 如果是偶数，设置 ah 为 0x13
    mov ah, 0x71
print:
    mov al, [esi]
    inc edx
    mov word[gs:ebx], ax
    add ebx, 2
    inc esi
    loop output_protect_mode_tag
```

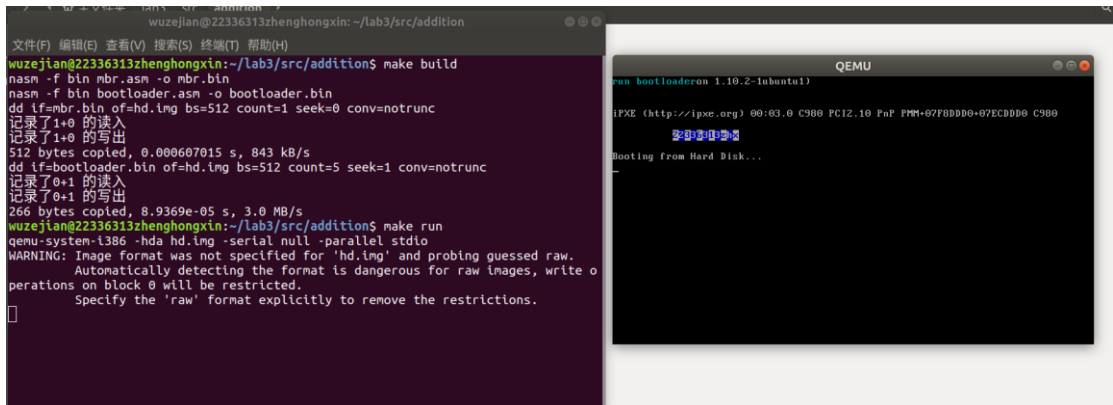
```

jmp $ ; 死循环
pgdt dw 0
      dd GDT_START_ADDRESS
bootloader_tag db 'run bootloader'
bootloader_tag_end:
protect_mode_tag db '22336313zhx'
protect_mode_tag_end:

```

● 实验结果展示:

通过执行前述代码, 可得下图结果:



Section 5 实验总结与心得体会

遇到的问题:

- 在完成选做任务时, 本来的思路是想寻找类似实模式的中断的方法来实现, 后续意识到这个思路存在较大问题, 进入保护模式后进行编程不可以使用中断进行光标位置的修改和在光标位置进行输出, 因为在保护模式下是 32 位, 而 BIOS 中断是 16 位程序, 在保护模式下无法实现。由于在保护模式完全无法使用 BIOS 的中断指令, 所以更换思路, 参考 Lab2 中字符显示原理, 采用计算显存地址的方式来设置需要进行输出的位置, 再通过一个变量的奇偶更换输出的前景色和背景色, 该变量初始化为 0, 每显示一个字符后自增 1。
- 在实验任务 2 中, 为了完成从 LBA 模式到 CHS 模式的转换, 本来的思路是在代码中完成柱面号, 磁头号 and 扇区号的换算, 但由于 8 位寄存器数量有限, 取余数操作较为麻烦等原因, 实现遇到很大困难, 后续直接笔算出柱面号, 磁头号 and 扇区号后直接调用中断 int13 完成实验。
- 在实验任务 3 中, gdb 窗口始终没有出现源代码, 故怀疑符号表生成有误, 但重复多次生成符号表操作后仍然无法载入符号表。后续检查发现原因, 在启动 qemu 后需要启动一个新终端来启动 gdb, 而此新终端必须

在符号表所在的文件夹中启动，否则加载符号表会失败。而如果不是在文件夹中打开终端，系统会默认终端是在主文件夹打开的。

总结:

本次实验中学习了从实模式到保护模式的转化，学习如何使用 I/O 端口和硬件交互。了解了读取硬盘的两种方法，一种是 LBA 模式，另一种是 CHS 模式，也学习了二者之间的联系区别和互相转化。最后在进入保护模式后编写了一个 32 位的汇编程序为保护模式编程打下了基础。

Section 6 对实验的改进建议和意见

在第一节 LBA 模式向 CHS 模式转化的参考资料中，第一篇参考资料需要付费解锁，故重新找了以下参考资料，以学习两种模式之间的相互转化。

[LBA和CHS转换\(转\) - 姜大伟 - 博客园](#)

Section 7 附录：参考资料清单

本节为可选章节，可以列出自己在实验过程中的一些重要参考书籍、博客网站等等，为将来的实验提供帮助。

指导书网址: <https://gitee.com/nelsoncheung/sysu-2023-spring-operating-system/tree/main/lab3>

Int13 中断参考: [INT13 中断详解_int13 中断功能-CSDN 博客](#)

Gdb 调试参考: https://gitee.com/nelsoncheung/sysu-2023-spring-operating-system/blob/main/appendix/debug_with_gdb_and_qemu