



本科生实验报告

实验课程：____操作系统原理实验____

实验名称：____从实模式到保护模式____

专业名称：____计算机科学与技术____

学生姓名：____熊彦钧____

学生学号：____23336266____

实验地点：____实验楼 B203____

实验成绩：____

报告时间：____2023 年 3 月 23 日____

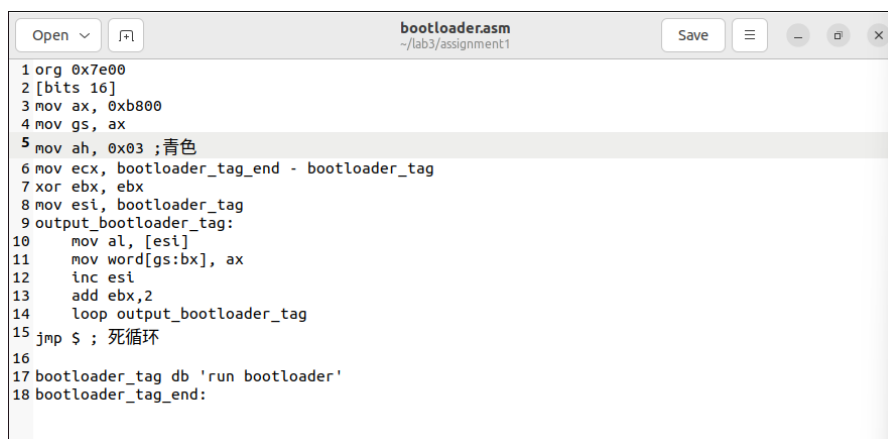
Section 1 实验概述

- 实验任务 1：① 复现 Example 1，说说你是怎么做的并提供结果截图；
②将 LBA28 读取硬盘的方式换成 CHS 读取，同时给出逻辑扇区号向 CHS 的转换公式。最后说说是怎么做的并提供结果截图。
- 实验任务 2：复现 Example 2，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。
- 实验任务 3：改造“Lab2-Assignment 4”为 32 位代码，即在加载到保护模式后执行自定义的汇编程序。

Section 2 实验步骤与实验结果

----- 实验任务 1（对应 assignment1.1） -----

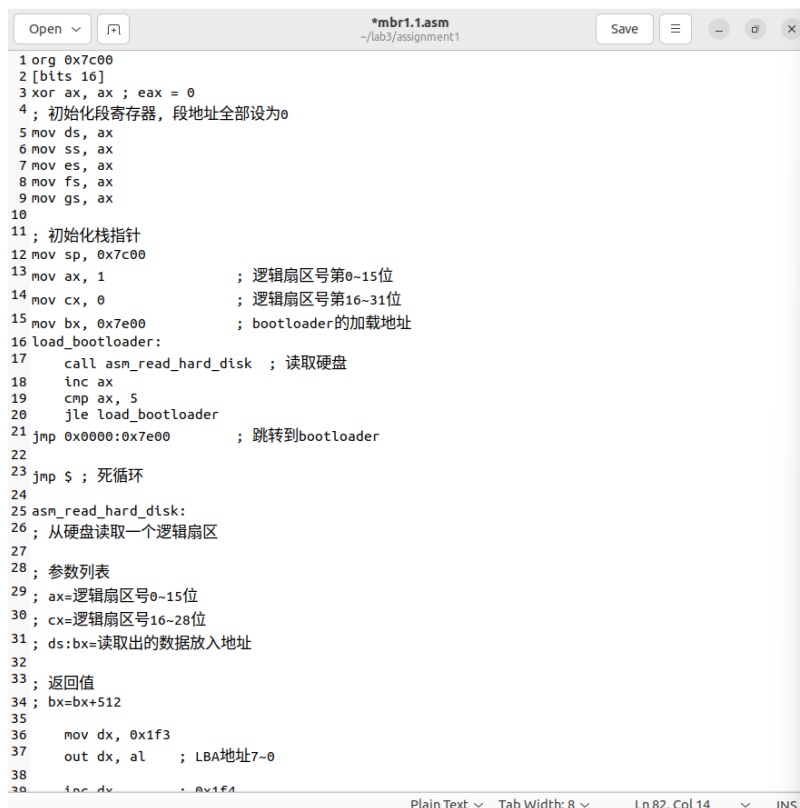
- 任务要求： 复现 example1:加载 bootloader。在本节中，我们将 lab2 中输出 Hello World 部份的代码放入到 bootloader 中，然后在 MBR 中加载 bootloader 到内存，并跳转到 bootloader 的起始地址执行。
- 实验步骤：①复制 bootloader.asm：这一文件的作用是在 qemu 显示屏上输出字符串”run bootloader”。和使用中断不同的是，我们可以通过“减法”来给 ecx 赋值，告诉系统输出字节的大小。这一部分的代码截图如下：



```
1 org 0x7e00
2 [bits 16]
3 mov ax, 0xb800
4 mov gs, ax
5 mov ah, 0x03 ;青色
6 mov ecx, bootloader_tag_end - bootloader_tag
7 xor ebx, ebx
8 mov esi, bootloader_tag
9 output_bootloader_tag:
10     mov al, [esi]
11     mov word[gs:bx], ax
12     inc esi
13     add ebx, 2
14     loop output_bootloader_tag
15 jmp $ ; 死循环
16
17 bootloader_tag db 'run bootloader'
18 bootloader_tag_end:
```

②复制 mbr.asm：和实模式的 mbr 不同的是，我们需要不少的代码来加载 bootloader。分析指导书的代码，我们可以看出来，我们需要用 in 指令和 out 指令，来传入 LBA 的地址，并且由于需要放在磁盘的 1~5 块上，因此我们的外循环需要执行五次，而由于我们使用的是 LBA28，因此我们需要分多

次写入端口。随后请求硬盘读并进行等待。这一部分的代码截图如下，以表示已经放在了本地文件中：

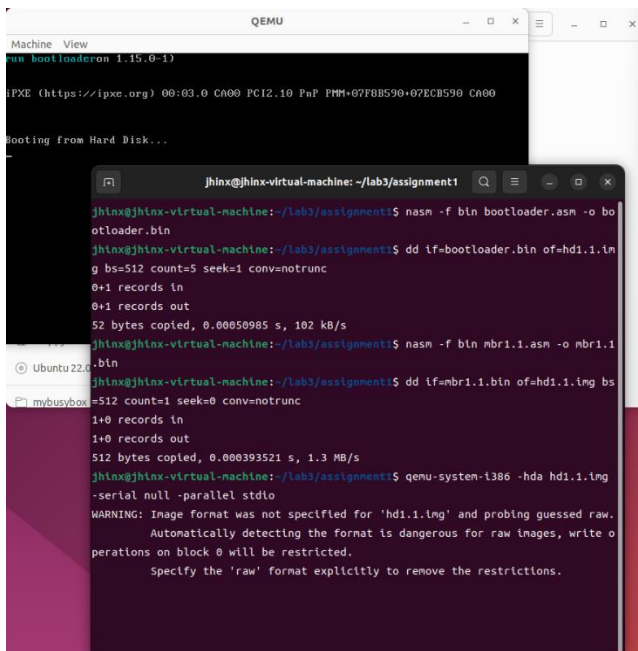


```
1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4 ; 初始化段寄存器，段地址全部设为0
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 ; 初始化栈指针
12 mov sp, 0x7c00
13 mov ax, 1 ; 逻辑扇区号第0~15位
14 mov cx, 0 ; 逻辑扇区号第16~31位
15 mov bx, 0x7e00 ; bootloder的加载地址
16 load_bootloader:
17     call asm_read_hard_disk ; 读取硬盘
18     inc ax
19     cmp ax, 5
20     jle load_bootloader
21     jmp 0x0000:0x7e00 ; 跳转到bootloder
22
23 jmp $ ; 死循环
24
25 asm_read_hard_disk:
26 ; 从硬盘读取一个逻辑扇区
27
28 ; 参数列表
29 ; ax=逻辑扇区号0~15位
30 ; cx=逻辑扇区号16~28位
31 ; ds:bx=读取出的数据放入地址
32
33 ; 返回值
34 ; bx=bx+512
35
36 mov dx, 0x1f3
37 out dx, al ; LBA地址7~0
38
39 inc dx ; 0x1f4
```

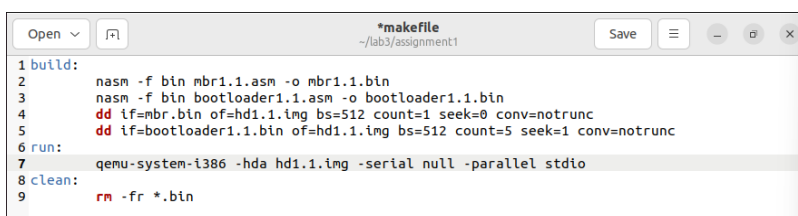
③编译并启动 qemu：同为启动 qemu，因此命令行代码和 lab2 的代码类似，唯一的区别是需要将两个.asm 文件整合到同一个磁盘中：

```
nasm -f bin bootloader.asm -o bootloader.bin
qemu-img create hd.img 10m
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
nasm -f bin mbr.asm -o mbr.bin
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
qemu-system-i386 -had hd.img -serial null -parallel stdio
```

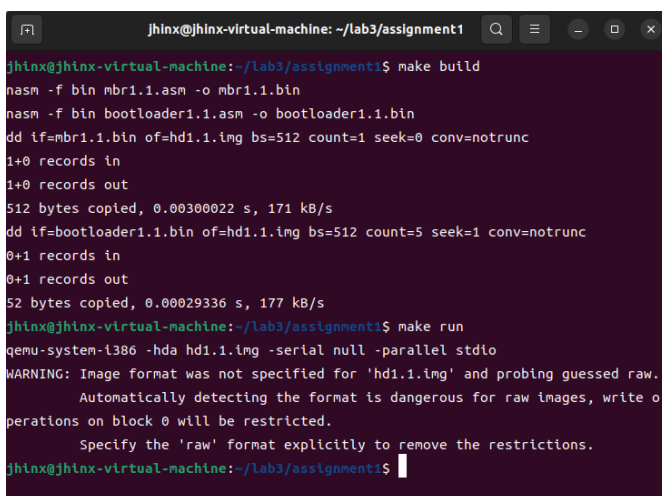
启动后截图如下：



在执行完这一步后，我才发现实验指导书提供了使用 `makefile` 文件运行的方法，经过实践后发现，使用 `makefile` 整合命令，可以在需要多次启动时节省大量的时间，由于本人在本地创建的相关文件命名有些许差异（如 `bootloader.asm` 命名成了 `bootloader1.1.asm`，以区分开不同实验的文件），因此也需要同步修改 `makefile` 中的命令，截图如下：



使用 `makefile` 运行的截图如下：



经过检验，实验结果相同，实验任务完成。

----- 实验任务 2（对应 assignment1.2） -----

- 任务要求： 将 LBA28 读取硬盘的方式换成 CHS 读取，同时给出逻辑扇区号向 CHS 的转换公式。
- 实验步骤： ①理解 CHS 模式的逻辑：首先我们要知道，CHS 是怎么分配硬盘的容量的，通过互联网搜索，我们了解到：一个硬盘的容量=柱面数（或磁道数）×磁头数×扇区数×每个扇区的大小（通常是 512 字节），并且得到了 LBA 向 CHS 转化的公式：

扇区号 $S=(\text{逻辑扇区号}L\%63(\text{每磁道的扇区数SPT}))+1$

磁头号 $H=(L/63)\%18$ （每柱面的磁头数）

柱面号 $C=(L/63)/18$

同时，我们还了解到用 CHS 读取时，各个寄存器应该存储的数据如下：

```
; 参数列表
; ax=逻辑扇区号低16位
; cx=逻辑扇区号高16位
; ds:bx=读取出的数据放入地址
; al=扇区数量
; ch=柱面号的低8位
; cl=扇区号(低6位)|柱面号高2位(cl的高2位)
; dh=磁头号
; dl=磁盘编号(0x80是第一个硬盘)
; bx=目标内存地址
; L=逻辑扇区号(LBA)
```

②根据公式修改 mbr 文件：我们需要修改 mbr.asm 中从实模式进入保护模式这部分的代码（即 asm_read_hard_disk 函数）：

```
mov si, 63 ;每磁道有63个扇区
mov di, 18 ;每柱面有18个磁头

xor dx, dx ;将dx清零
div si
inc dx
mov cl, dl ;将扇区号存储到cl

xor dx, dx
div di
mov dh, dl ;将余数存储到dh,表示磁头号

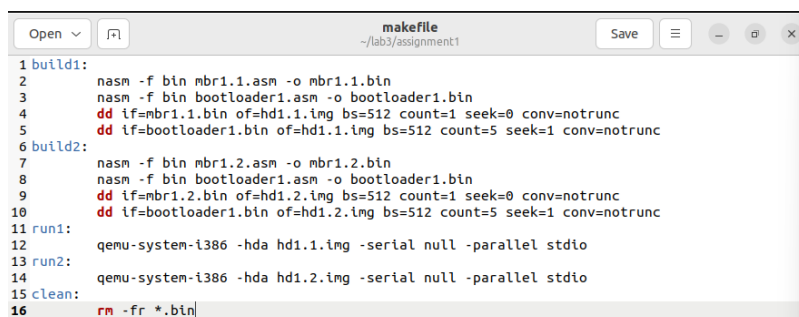
mov ch, al
shr ax, 8 ;将ax右移8位,获取柱面号的高2位
and al, 0x03 ;保留高2位
or cl, al ;将柱面号的高2位与扇区号合并

mov dl, 0x80 ;0x80为第一个硬盘
mov ah, 0x02 ;功能号为0x02(读扇区)
mov al, 1 ;设置读取的扇区数为1
int 0x13

add bx, 512 ;更新 bx 地址

ret
```

③修改 makefile 文件: 为了让 assignment1.2 和 assignment1.1 共同放在同一个文件夹中, 并且二者的文件不会互相覆盖, 本人在 assignment1.1 的基础上进行了修改, 修改如下:

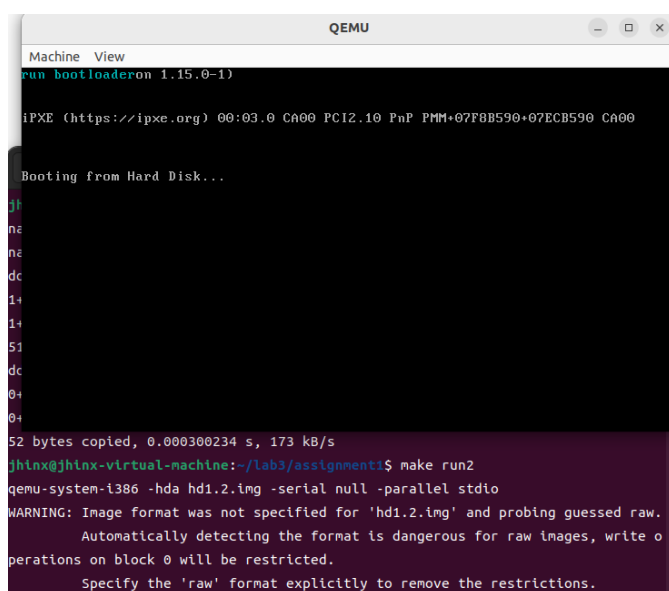


```
1 build1:
2     nasm -f bin mbr1.1.asm -o mbr1.1.bin
3     nasm -f bin bootloader1.asm -o bootloader1.bin
4     dd if=mbr1.1.bin of=hd1.1.img bs=512 count=1 seek=0 conv=notrunc
5     dd if=bootloader1.bin of=hd1.1.img bs=512 count=5 seek=1 conv=notrunc
6 build2:
7     nasm -f bin mbr1.2.asm -o mbr1.2.bin
8     nasm -f bin bootloader1.asm -o bootloader1.bin
9     dd if=mbr1.2.bin of=hd1.2.img bs=512 count=1 seek=0 conv=notrunc
10    dd if=bootloader1.bin of=hd1.2.img bs=512 count=5 seek=1 conv=notrunc
11 run1:
12    qemu-system-i386 -hda hd1.1.img -serial null -parallel stdio
13 run2:
14    qemu-system-i386 -hda hd1.2.img -serial null -parallel stdio
15 clean:
16    rm -fr *.bin
```

此时, 1 对应的是 assignment1.1, 2 对应的是 assignment1.2。例如: 运行 1.2 的代码应该是:

```
make build2
make run2
```

程序运行后 qemu 显示的内容如下:



```
Machine View
run bootloader on 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...

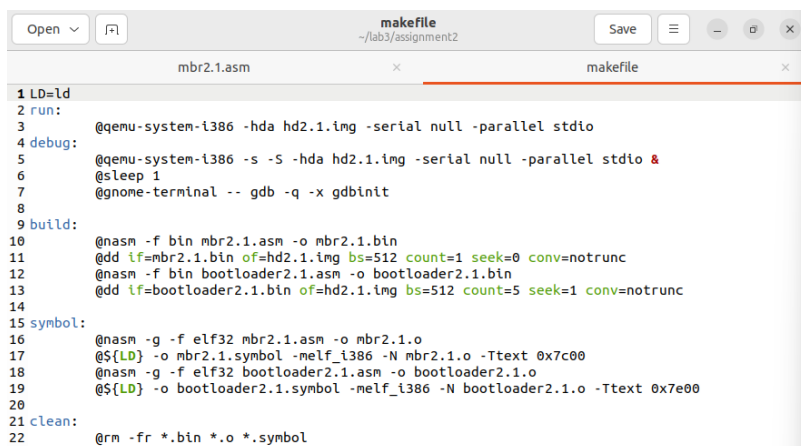
52 bytes copied, 0.000300234 s, 173 kB/s
jhlhx@jhlhx-virtual-machine:~/lab3/assignment1$ make run2
qemu-system-i386 -hda hd1.2.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd1.2.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

可以看到, qemu 同样显示了“run bootloader”, 实验任务完成。

----- 实验任务 3 (对应 assignment2) -----

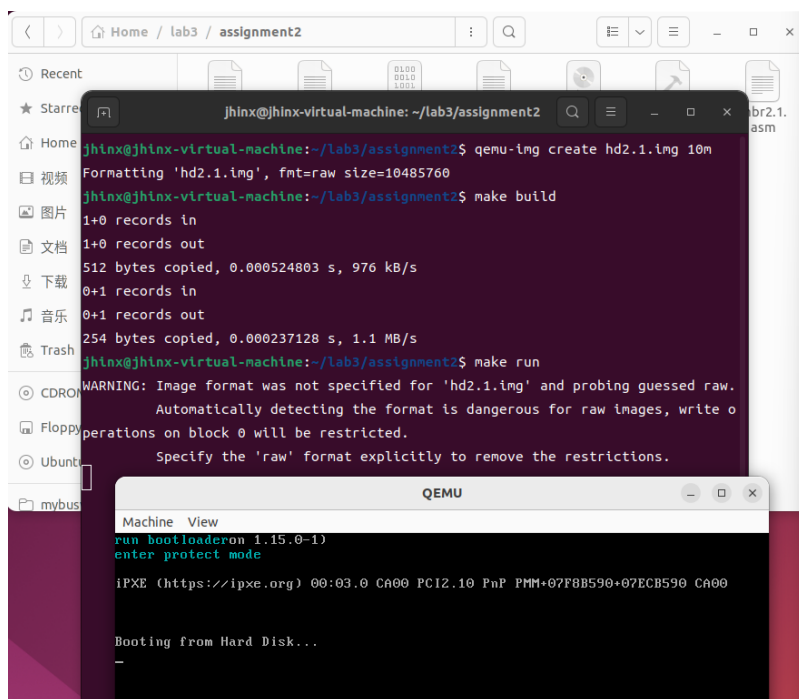
- 任务要求: 在 bootloader 中进入保护模式, 并在进入保护模式后在显示屏上输出“protect mode”。使用 gdb 或其他 debug 工具在进入保护模式的 4 个重要步骤上设置断点, 并结合代码、寄存器的内容等来分析这 4 个步骤。
- 实验步骤: ①复制代码并运行: 首先我们需要复制代码到本地, 随后, 像

assignment1 一样，因为我在本地的文件名和实验指导的不一样，因此需要修改 makefile 文件，修改截图如下：



```
1 LD=ld
2 run:
3     @qemu-system-i386 -hda hd2.1.img -serial null -parallel stdio
4 debug:
5     @qemu-system-i386 -s -S -hda hd2.1.img -serial null -parallel stdio &
6     @sleep 1
7     @gnome-terminal -- gdb -q -x gdbinit
8
9 build:
10    @nasm -f bin mbr2.1.asm -o mbr2.1.bin
11    @dd if=mbr2.1.bin of=hd2.1.img bs=512 count=1 seek=0 conv=notrunc
12    @nasm -f bin bootloader2.1.asm -o bootloader2.1.bin
13    @dd if=bootloader2.1.bin of=hd2.1.img bs=512 count=5 seek=1 conv=notrunc
14
15 symbol:
16    @nasm -g -f elf32 mbr2.1.asm -o mbr2.1.o
17    @${LD} -o mbr2.1.symbol -melf_i386 -N mbr2.1.o -Ttext 0x7c00
18    @nasm -g -f elf32 bootloader2.1.asm -o bootloader2.1.o
19    @${LD} -o bootloader2.1.symbol -melf_i386 -N bootloader2.1.o -Ttext 0x7e00
20
21 clean:
22    @rm -fr *.bin *.o *.symbol
```

修改后使用 makefile 运行程序，运行结果如下：



可以看到 qemu 显示屏成功输出“enter protect mode”，进入保护模式成功。

②使用 gdb 进行 debug：在前两个实验中，我们通过 gdb 远程连接 qemu 进行调试。在这一实验中依然沿袭这一思路，唯一的区别就是，在这一实验中我们创建了 makefile 文件，可以把一系列所需的命令整合到里面。此时，只需要输入 `make debug` 即可启动 gdb 进行调试。

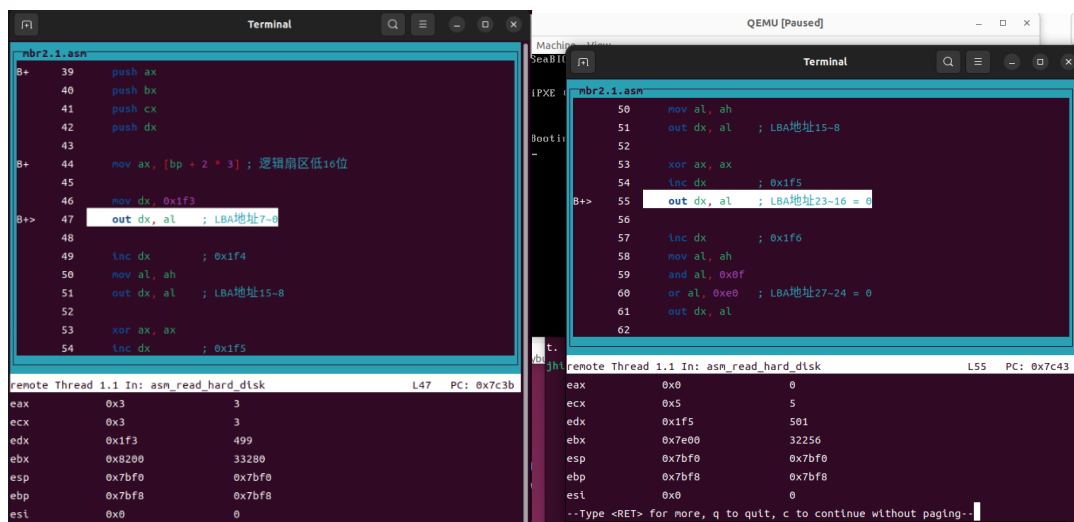
启动后，我们还需要输入以下指令来设置第一个断点，并且打开可视化界面

```
b *0x7c00
c
```

layout src

打开可视化界面后，我们就可以通过对相应的指令设置断点来查看寄存器以及其他信息。部分 debug 信息如下：

(1) 在分别输入 LBA28 时，可以查看 edx 寄存器的值，验证是否输入到了正确的地方：



```
mbr2.1.asm
39  push ax
40  push bx
41  push cx
42  push dx
43
44  mov ax, [bp + 2 * 3] ; 逻辑扇区低16位
45
46  mov dx, 0x1f3
47  out dx, al ; LBA地址7-8
48
49  inc dx ; 0x1f4
50  mov al, ah
51  out dx, al ; LBA地址15-8
52
53  xor ax, ax
54  inc dx ; 0x1f5

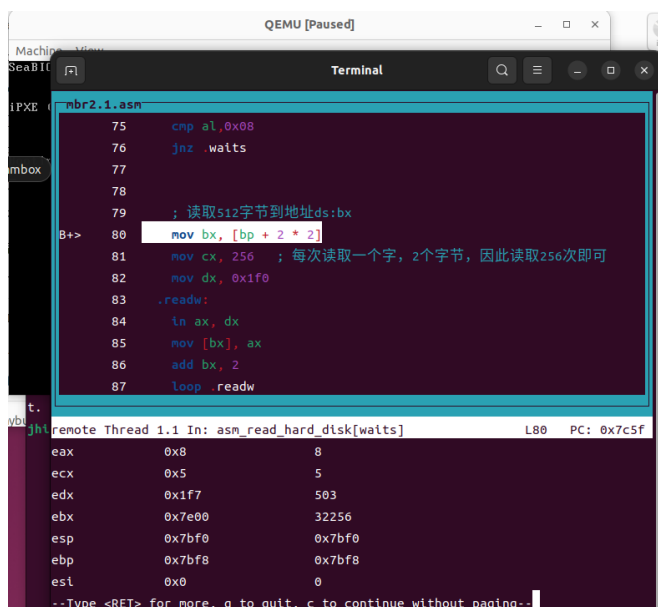
remote Thread 1.1 In: asm_read_hard_disk L47 PC: 0x7c3b
eax 0x3 3
ecx 0x3 3
edx 0x1f3 499
ebx 0x8200 33280
esp 0x7bf0 0x7bf0
ebp 0x7bf8 0x7bf8
esi 0x0 0

QEMU [Paused]
mbr2.1.asm
50  mov al, ah
51  out dx, al ; LBA地址15-8
52
53  xor ax, ax
54  inc dx ; 0x1f5
55  out dx, al ; LBA地址23-16 = 0
56
57  inc dx ; 0x1f6
58  mov al, ah
59  and al, 0x0f
60  or al, 0xe0 ; LBA地址27-24 = 0
61  out dx, al
62

remote Thread 1.1 In: asm_read_hard_disk L55 PC: 0x7c43
eax 0x0 0
ecx 0x5 5
edx 0x1f5 501
ebx 0x7e00 32256
esp 0x7bf0 0x7bf0
ebp 0x7bf8 0x7bf8
esi 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--
```

可以看到，edx 随着指令的修改而被修改，传参正确。

(2) 在跳转到保护模式之前，需要读取 512 字节到 bx 里面，因此我们可以查看数据是否读入到了正确的地址中：



```
mbr2.1.asm
75  cmp al, 0x08
76  jnz waits
77
78
79  ; 读取512字节到地址ds:bx
80  mov bx, [bp + 2 * 2]
81  mov cx, 256 ; 每次读取一个字, 2个字节, 因此读取256次即可
82  mov dx, 0x1f0
83  .readw:
84  in ax, dx
85  mov [bx], ax
86  add bx, 2
87  loop .readw

remote Thread 1.1 In: asm_read_hard_disk[waits] L80 PC: 0x7c5f
eax 0x8 8
ecx 0x5 5
edx 0x1f7 503
ebx 0x7e00 32256
esp 0x7bf0 0x7bf0
ebp 0x7bf8 0x7bf8
esi 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--
```

可以看到，ebx 此时的内容是 0x7e00，读入地址正确。

(3) 接下来我们将分别检验保护模式的四大步骤：准备 GDT，用 lgdt 指令加载 GDTR 信息；打开第 21 根地址线；开启 cr0 的保护模式标志位；远跳

转，进入保护模式。

④用 lgdt 加载 GDTR 信息前后的寄存器数据如下：

```
bootloder2.1.asm
32
33 ;创建保护模式下平坦模式代码段描述符
34 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限
35 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段
36
37 ;初始化描述符表寄存器GDTR
38 mov word [pgdt], 39 ;描述符表的界限
B++ 39 lgdt [pgdt]
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010B
43 out 0x92,al ;打开A20
44
45 cll ;中断机制尚未工作
46 mov eax,cr0

remote Thread 1.1 In: output_bootloader_tag L38 PC: 0x7e7e
eax 0x372 882
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--

bootloder2.1.asm
34 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限
35 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段
36
37 ;初始化描述符表寄存器GDTR
38 mov word [pgdt], 39 ;描述符表的界限
B++ 39 lgdt [pgdt]
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010B
43 out 0x92,al ;打开A20
44
45 cll ;中断机制尚未工作
46 mov eax,cr0

remote Thread 1.1 In: output_bootloader_tag L39 PC: 0x7e84
eax 0x372 882
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--
```

各个寄存器的值并非发生变化。

⑤打开第 21 根地址线：这一步骤一共有三条指令，其中前两天指令执行后的寄存器数据分别如下：

```
in al,0x92
or al,0000_0010B
```

```
bootloder2.1.asm
34 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0, 段界限
35 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb, 代码段
36
37 ;初始化描述符表寄存器GDTR
38 mov word [pgdt], 39 ;描述符表的界限
39 lgdt [pgdt]
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010B
B++ 43 out 0x92,al ;打开A20
44
45 cll ;中断机制尚未工作
46 mov eax,cr0

remote Thread 1.1 In: output_bootloader_tag L42 PC: 0x7e8b
eax 0x302 770
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--

bootloder2.1.asm
36
37 ;初始化描述符表寄存器GDTR
38 mov word [pgdt], 39 ;描述符表的界限
39 lgdt [pgdt]
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010B
B++ 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 ;设置PE位
48 mov cr0, eax ;设置PE位

remote Thread 1.1 In: output_bootloader_tag L43 PC: 0x7e8d
eax 0x302 770
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--
```

可以看到，eax 的值不变，依然为 0x302。

⑥开启 cr0 的保护模式标志位：我们还是一样查看会改变寄存器的两条指令：

```
mov eax, cr0
or eax, 1
```

```
bootloader2.1.asm
B+ 42 or al,0000_0010B
B+ 43 out 0x92,al ;打开A20
44
45 cli ;中断机制尚未工作
B+ 46 mov eax,cr0
B+ 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
53 ;16位的描述符选择子: 32位偏移
54 ;清流水线并串行化处理器

remote Thread 1.1 In: output_bootloader_tag L47 PC: 0x7e93
eax 0x10 16
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--

bootloader2.1.asm
B+ 42 or al,0000_0010B
B+ 43 out 0x92,al ;打开A20
44
45 cli ;中断机制尚未工作
B+ 46 mov eax,cr0
B+ 47 or eax,1
B+> 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
53 ;16位的描述符选择子: 32位偏移
54 ;清流水线并串行化处理器

remote Thread 1.1 In: output_bootloader_tag L48 PC: 0x7e97
eax 0x11 17
ecx 0x0 0
edx 0x80 128
ebx 0x1c 28
esp 0x7c00 0x7c00
ebp 0x0 0x0
esi 0x7eec 32492
--Type <RET> for more, q to quit, c to continue without paging--
```

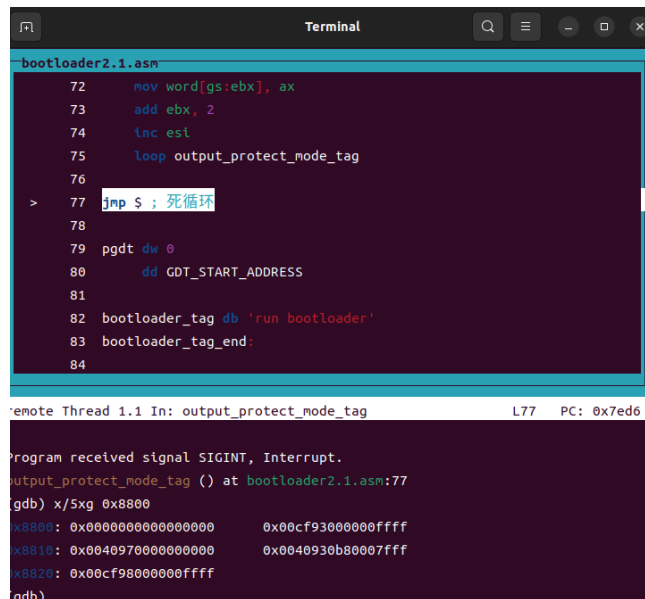
通过查看 `eax` 的值，我们可以看到 `eax` 先被赋值 16(`cr0`)，随后变成了 17，最后会把 17 赋值回给 `cr0`，作为标志位。说明代码运行正确。

④远跳转：远跳转后，系统会进入保护模式，这里我们选择保护模式下的其中一条指令设置断点，查看寄存器：

```
bootloader2.1.asm
61 mov eax, STACK_SELECTOR
62 mov ss, eax
63 mov eax, VIDEO_SELECTOR
64 mov gs, eax
65
66 mov ecx, protect_mode_tag_end - protect_mode_tag
67 mov ebx, 80 * 2
68 mov esi, protect_mode_tag
69 mov ah, 0x3
70 output_protect_mode_tag:
B+> 71 mov al, [esi]
72 mov word[gs:ebx], ax
73 add ebx, 2
74 inc esi

remote Thread 1.1 In: output_protect_mode_tag L71 PC: 0x7eca
0x7eca
eflags 0x0 [ IOPL=0 PF ]
cs 0x20 32
ss 0x10 16
ds 0x8 8
es 0x8 8
fs 0x0 0
gs 0x18 24
--Type <RET> for more, q to quit, c to continue without paging--
```

我们可以看到段寄存器的内容变成了段选择子，进入保护模式成功。最后，我们在死循环处中断程序运行，并查看 GDT 的 5 个段描述符的内容：



```
bootloader2.1.asm
72  mov word[gs:ebx], ax
73  add ebx, 2
74  inc esi
75  loop output_protect_mode_tag
76
> 77  jmp $ ; 死循环
78
79  pgdt dw 0
80      dd GDT_START_ADDRESS
81
82  bootloader_tag db 'run bootloader'
83  bootloader_tag_end:
84

Remote Thread 1.1 In: output_protect_mode_tag L77 PC: 0x7ed6

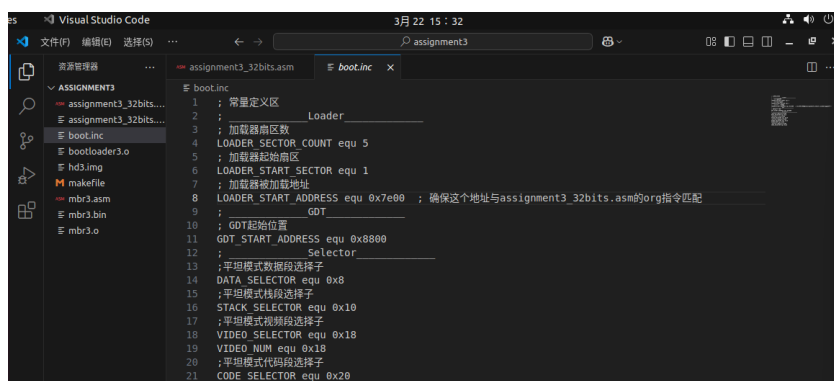
Program received signal SIGINT, Interrupt.
output_protect_mode_tag () at bootloader2.1.asm:77
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000 0x00cf93000000ffff
0x8810: 0x0040970000000000 0x0040930b80007fff
0x8820: 0x00cf98000000ffff
(gdb)
```

可以看到 GDT 的内容和我们的设置相吻合。综上所述，实验任务完成。

----- 实验任务 4 （对应 assignment3） -----

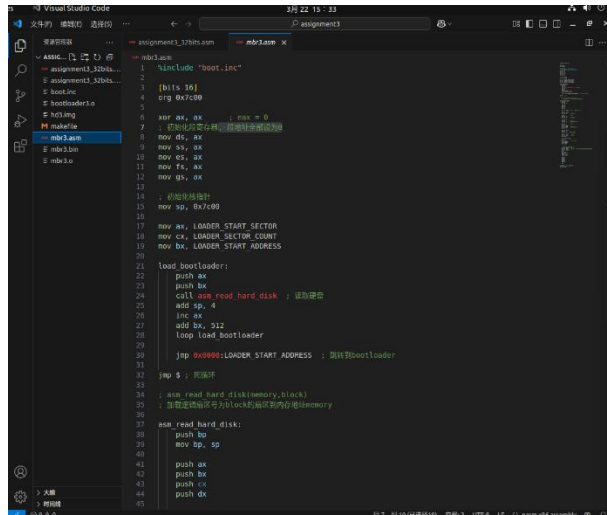
- 任务要求：改造“Lab2-Assignment 4”为 32 位代码，即在加载到保护模式后执行自定义的汇编程序。
- 实验步骤：①编写程序：首先，我们需要了解整个程序有多少个文件，并分别作出修改：

boot.inc:这是保护模式的常量定义区，无需修改：



```
boot.inc
1 ; 常量定义区
2 ; Loader
3 ; 加载器扇区数
4 LOADER_SECTOR_COUNT equ 5
5 ; 加载器起始扇区
6 LOADER_START_SECTOR equ 1
7 ; 加载器被加载地址
8 LOADER_START_ADDRESS equ 0x7e00 ; 确保这个地址与assignment3_32bits.asm的org指令匹配
9 ; GDT
10 ; GDT起始位置
11 GDT_START_ADDRESS equ 0x8800
12 ; Selector
13 ; 平坦模式数据段选择子
14 DATA_SELECTOR equ 0x08
15 ; 平坦模式栈段选择子
16 STACK_SELECTOR equ 0x10
17 ; 平坦模式视频段选择子
18 VIDEO_SELECTOR equ 0x18
19 VIDEO_NUM equ 0x18
20 ; 平坦模式代码段选择子
21 CODE_SELECTOR equ 0x20
```

mbr3.asm:这是用于实模式跳转到保护模式的程序，由于我们的汇编程序需要到保护模式中执行，因此这一部分也无需修改：



assignment3_32bits.asm:这是系统进入保护模式后的代码，这一部分需要在 lab2 的基础上进行大量修改，部分关键位置修改如下：

（1）显示学号：由于我们需要到保护模式下显示学号，不能像 lab2 一样通过中断单独输出每个字符，我们首先修改了 ecx 的传参，使用类似 assignment2 的方法进行传参，并且我们把这一部分的函数移动到了保护模式启动后的区域内，以保证信息输出在保护模式下的 qemu 中，代码如下：

```

68 ; 在保护模式下显示学号
69 call xian_shi_xue_hao
70
71 ; 初始化变量
72 mov dword [pos], 0 ; 初始位置(左上角)
73 mov dword [counter], 0 ; 字符计数器初始化为0
74 mov byte [current_char], 0 ; 数字序列索引
75 mov byte [front_color], 1 ; 前景色初始化
76 mov byte [background_color], 0x20 ; 背景色初始化
77 mov byte [color_counter], 0 ; 颜色计数器初始化
78
79 ; 跳转到主循环
80 jmp main
81
82 ; 在保护模式下显示学号
83 xian_shi_xue_hao:
84 pushad
85
86 mov edi, 1986 ; 屏幕中间位置
87 mov ecx, xian_shi_xue_hao_end - message ; 字符串长度
88 mov ebx, message
89 mov ah, 0xEC ; 红色前景，黄色背景
90
91 .display_loop:
92 mov al, [ebx] ; 获取字符
93 mov [gs:edi], ax ; 写入显存
94 add edi, 2 ; 移动到下一个字符位置
95 inc ebx ; 指向下一个字符
96 loop .display_loop
97
98 popad
99 ret

```

（2）延时函数：由于在保护模式下，数据的大小可以为 32 位，因此我们无需再使用双层循环来保证执行速度降低到一定程度，我们可以通过一层循环，但是增大 ecx 的值来延时，经过实验，ecx=0x50000 是一个比较恰当的速度。

```

; 延时函数(通过循环实现)
_delay:
    pushad
    mov ecx, 0x50000 ; 32位下调整为更大的计数
delay_loop:
    loop delay_loop
    popad
    ret

```

(3) 位置更新函数以及颜色修改函数：由于我们在 lab2 的时候，仅使用赋值命令、算数逻辑命令以及条件跳转命令来实现了这两个函数，并没有使用中断，因此在执行命令'cli'关中断进入保护模式后，这一部分代码依旧可以运行，无需修改。

(4) makefile：还是前面老生常谈的问题，我们需要修改 makefile 文件中的文件名：

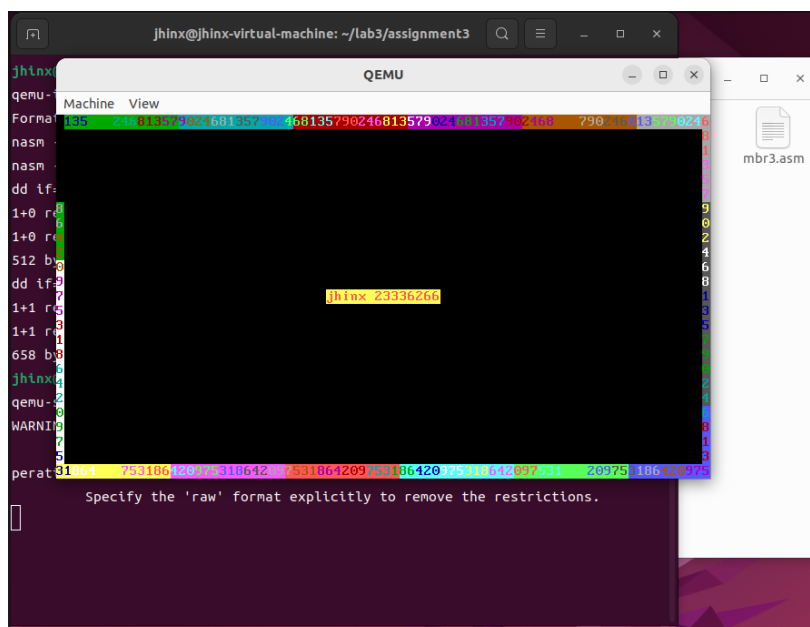
```

makefile
~/lab3/assignment3

1 build:
2     qemu-img create hd3.img 10m
3     nasm -f bin mbr3.asm -o mbr3.bin
4     nasm -f bin assignment3_32bits.asm -o assignment3_32bits.bin
5     dd if=mbr3.bin of=hd3.img bs=512 count=1 seek=0 conv=notrunc
6     dd if=assignment3_32bits.bin of=hd3.img bs=512 count=5 seek=1 conv=notrunc
7 run:
8     qemu-system-i386 -hda hd3.img -serial null -parallel stdio
9 clean:
10     rm -fr *.bin

```

程序运行时 qemu 截图如下：



可以看到，qemu 中成功显示个人信息以及顺时针的渐变条带，实验任务完成。（完整代码请详见附件）

Section 5 实验总结与心得体会

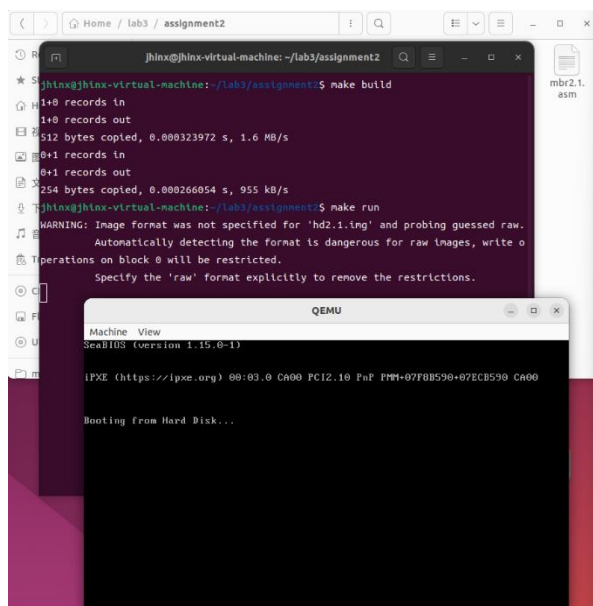
本人在实验过程中遇到以下收获/困惑：

(1) 在 `gdb` 调试的时候，以前都是在当前指令设置断点后查看寄存器，结果在本次实验发现，需要在当前指令的下一条指令设置断点，才能查看到当前指令的寄存器数值，详情请见 `assignment2` 的调试。

(2) 在进入保护模式的四个步骤，对于查看寄存器的值，特别是远跳转指令，本人实在是不清楚通过哪个寄存器的值可以检验跳转到了保护模式，也因此留下了疑问。

本人在实验过程中遇到了以下几个问题：

(1) 在第一次尝试 `assignment2` 的时候，由于我是在本地文件夹中手动新建 `.asm` 文件以及 `makefile`，并把内容拷贝过去，因此第一次尝试运行时出现了以下问题：`qemu` 虽然被正常打开，但是没有输出该输出的内容。



经过检查发现，是我忘记在本地创建加载的磁盘，使用命令 `'qemu-img create hd2.img 10m'` 后程序就能正常运行了。

(2) 在 `assignment2` 中，本人第一次输入 `make debug` 的时候，出现了如下报错：

```
jhinx@jhinx-virtual-machine: ~/lab3/assignment2
jhinx@jhinx-virtual-machine:~/lab3/assignment2$ make clean
jhinx@jhinx-virtual-machine:~/lab3/assignment2$ make build
ld: warning: cannot find entry symbol _start; defaulting to 0000000000007c00
ld: warning: cannot find entry symbol _start; defaulting to 0000000000007c00
bootloader2.1.asm:2: error: parser: instruction expected
make: *** [makefile:11: build] Error 1
jhinx@jhinx-virtual-machine:~/lab3/assignment2$
```

经过检查，是由于忘记注释掉.asm 文件的 `org`，注释掉后文件即可正常运行。

(3)在 debug 过程中，本人在调试完 mbr 文件后，尝试直接跳转到 bootloader 文件，但是输入 `b *0x7e00` 和 `c` 后，虽然成功跳转，但是输入 `layout src`，却没有显示出 bootloader 的源码：

```
Terminal
mbr2.1.asm
40  push bx
41  push cx
42  push dx
43
B+> 44  mov ax, [bp + 2 * 3] ; 逻辑扇区低16位
45
46  mov dx, 0x1f3
B+ 47  out dx, al ; LBA地址7~0
48
49  inc dx ; 0x1f4
50  mov al, ah
B+ 51  out dx, al ; LBA地址15~8
52
53  xor ax, ax
B+ 54  inc dx ; 0x1f5
B+ 55  out dx, al ; LBA地址23~16 = 0

remote Thread 1.1 In: asm_read_hard_disk L44 PC: 0x7c35
Breakpoint 11 at 0x7ea2: file bootloader2.1.asm, line 58.
(gdb) c
Continuing.
```

经过尝试，关闭 qemu 和 gdb，从头开始设置断点 `b *0x7e00`，即可成功显示。

```
QEMU [Paused]
Machine: VM
SeaBIOS (0.1.1)
iPXE (0.27.0)
Bootiv
-
vbusbv

Terminal
bootloader2.1.asm
1  %include "boot.inc"
2  ;org 0x7e00
3  [bits 16]
B+> 4  mov ax, 0xb800
5  mov gs, ax
6  mov ah, 0x03 ;青色
7  mov ecx, bootloader_tag_end - bootloader_tag
8  xor ebx, ebx
9  mov esi, bootloader_tag
10 output_bootloader_tag:
11  mov al, [esi]
12  mov word[gs:bx], ax
13  inc esi

remote Thread 1.1 In: L4 PC: 0x7e00
(gdb)
```

Section 6 附录：代码清单

代码请见附件。