



本科生实验报告

实验课程：_____操作系统原理实验_____

实验名称：_____实验 3 从实模式到保护模式_____

专业名称：_____计算机科学与技术(超级计算方向) _____

学生姓名：_____林天皓_____

学生学号：_____18324034_____

实验地点：_____

实验成绩：_____

报告时间：_____2021. 3. 30_____

Assignment 1 硬盘读取

1.1 复现 Example 1，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。

24bitLBA 通过端口读取硬盘：

首先需要理解 LBA 访问硬盘的方式：

LBA 省去了 CHS 中稍微复杂的磁头，柱面，扇区的形式直接统一为一个参数。同时分为 24bit 的 LBA 访问方式和 48bit 的 LBA 访问方式。我们这里使用到的是 24bit 的 LBA 访问方式。下面介绍 24bitLBA 的访问方式

写 0x1f1	0
写 0x1f2	要读的扇区数
写 0x1f3	LBA 参数的 0~7 位
写 0x1f4	LBA 参数的 8~15 位
写 0x1f5	LBA 参数的 16~23 位
写 0x1f6	7~5 位,111,第 4 位 0 表示主盘,1 表示从盘,3~0 位,LBA 参数的 24~27 位
写 0x1f7	0x20 为读, 0x30 为写
读 0x1f7	第 4 位为 0 表示读写完成，否则要一直循环等待
读 0x1f0	每次读取 1 个 word,反复循环，直到读完所有数据

表 1-24bitLBA 访问外存方式

在助教给出的示例代码中,通过将扇区号和硬盘设置写入端口 0x1f3~0x1f6,读取的扇区数量写入端口 0x1f2，向 0x1F7 端口写入 0x20,硬盘收到请求后就会准备好数据，我们就可以从 0x1F0 中循环的读取数据，对应代码如下。使用了 5

个循环。

```
;第一部分：写入读取过程=====写入读取硬盘的地址=====
mov al,1
mov dx, 0x1f3
out dx, al
inc dx
mov al, ah ;al=0
out dx, al
mov ax, cx
inc dx
out dx, al ; LBA 地址 23~16
inc dx
mov al, ah
and al, 0x0f
or al, 0xe0 ; LBA 地址 27~24,这里包含的设置读取方式为 LBA 和主硬盘
out dx, al
mov dx, 0x1f2
mov al, 1
;=====写入读取扇区数量=====
out dx, al ; 读取 1 个扇区
mov dx, 0x1f7 ; 0x1f7
mov al, 0x20 ;读命令
out dx,al
```

接下来使用等待函数等待硬盘准备数据。然而这里我们使用的是模拟器的虚拟硬盘，所以如果你的代码没有读错扇区的话应该不会在该等待函数中循环。对应代码如下

```
;====并不会循环等待的等待函数，除非你写错扇区号====
.waits:
    in al, dx ; dx = 0x1f7
    and al,0x88
    cmp al,0x08
    jnz .waits
```

最后每次循环读取一个扇区，读取一个扇区的过程中对应了 256 次读取双字，对应的代码如下。

```
; =====循环 256 次读取硬盘=====
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

```

直接按照助教的代码，运行 make run 运行即可输出 run bootloader。

```

→ os git:(main) X cd lab3/assignment1
→ assignment1 git:(main) X make run
ld: warning: cannot find entry symbol _start; defaulting
c00
QEMU
run bootloader on rel-1.13.0-0-gf21b5a4aeb02-prebuilt.qemu.org)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0
Booting from Hard Disk...
_

```

图 1-复现 example 1

为了更加直观明显的看到读取硬盘的过程，我们可以通过 gdb 查看内存中的值展现。

```

65
66      ; 读取512字节到地址ds:bx
67      mov cx, 256      ; 每次读取一个字，2个字节，因此读取256次即可
68      mov dx, 0x1f0
69      .readw:
> 70      in ax, dx
71      mov [bx], ax
72      add bx, 2
73      loop .readw
74
75      ret
76
77      times 510 - ($ - $$) db 0
78      db 0x55, 0xaa^?
79
80
81
82
83
remote Thread 1.1 In: asm_read_hard_disk.readw
2: $cx = 0
1: $ax = 88
2: $cx = 0
1: $ax = 8
2: $cx = 0
1: $ax = 8
2: $cx = 0
1: $ax = 8
2: $cx = 0
1: $ax = 8
2: $cx = 256
1: $ax = 8
2: $cx = 256
(gdb) x/8a 0x7e00
0x7e00: 0x0    0x0    0x0    0x0
0x7e10: 0x0    0x0    0x0    0x0
(gdb)

```

图 2-未开始载入内存

代码运行到此处，内存中 0x7e00 附近的地址值全为 0，还未将硬盘中的数据读取到内存中，下面我们执行若干次 ni

```
66      ; 读取512字节到地址ds:bx
67      mov cx, 256      ; 每次读取一个字，2个字节，因此读取256次即可
68      mov dx, 0x1f0
69      .readw:
70      in ax, dx
71      mov [bx], ax
> 72      add bx, 2
73      loop .readw
74
75      ret
76
77      times 510 - ($ - $$) db 0
78      db 0x55, 0xaa^?
79
80
81
82
83
```

```
remote Thread 1.1 In:
2: $cx = 251
1: $ax = 0
2: $cx = 251
1: $ax = 0
2: $cx = 250
(gdb) x/8a 0x7e00
0x7e00: 0x8eb800b8      0x6603b4e8      0xeb9      0x0
0x7e10: 0x0      0x0      0x0      0x0
(gdb) ni
1: $ax = 26112
2: $cx = 250
1: $ax = 26112
2: $cx = 250
(gdb) x/8a 0x7e00
0x7e00: 0x8eb800b8      0x6603b4e8      0xeb9      0x6600
0x7e10: 0x0      0x0      0x0      0x0
(gdb) |
```

图 3-数据被载入内部

可见 0x7e00 内存地址中的值已经载入内存，下面我们对检查该值是否正确

```
≡ .gdbinit  ASM mbr.asm  ≡ bootloader.bin.hexdump ×  M makefile
1  Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2  00000000: B8 00 B8 8E E8 B4 03 66 B9 0E 00 00 00 66 31 DB 8.8.h
3  00000010: 66 BE 26 00 00 00 67 8A 06 65 89 07 66 46 66 83 f>&..
4  00000020: C3 02 E2 F2 EB FE 72 75 6E 20 62 6F 6F 74 6C 6F C.brk
5  00000030: 61 64 65 72 ader
6
```

图 4-验证数据正确性

对照 bootloader.bin 的 hexdump 查看，经过验证，我们写入的值是正确的。。

修改：多快好省，读 5 个扇区

接下来我们尝试修改该代码，例如将助教的循环 5 次代码改为不循环，但是
一次循环我们直接读取 5 个扇区，进行 1280 次双字读取。修改的代码如下：

```
;1.修改 ax 比较的值
load_bootloader:
    call asm_read_hard_disk ; 读取硬盘
    inc ax
    cmp ax, 1 ;从 5 修改为 1
    jle load_bootloader
jmp 0x0000:0x7e00 ; 跳转到 bootloader
```

```
;2.修改循环中读取双字的数量
    mov cx, 256*5 ; 改为读取 256*5 次
    mov dx, 0x1f0
.readw:
    in ax, dx
    mov [bx], ax
    add bx, 2
    loop .readw
```

make run 运行，我们可以得到一样的结果

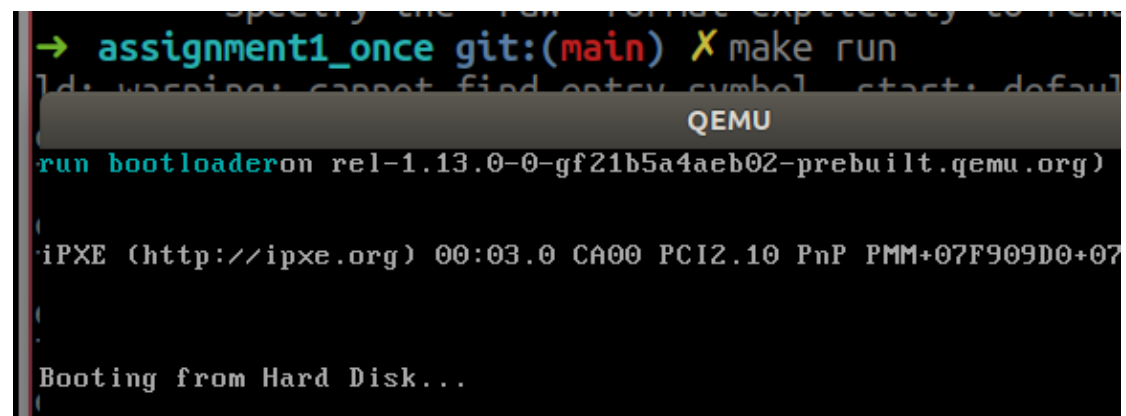


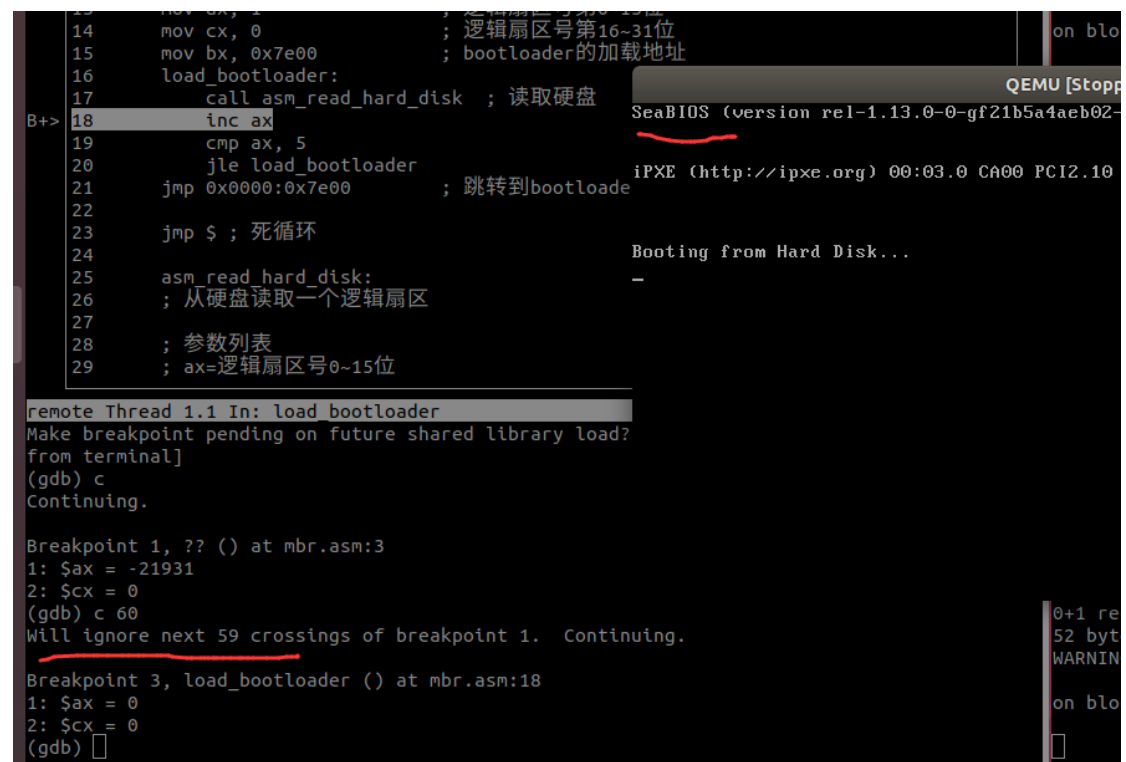
图 5-输出结果正确

仍未能理解的对代码的疑问

值得一提的是在尝试使用 gdb 理解这段代码的时候遇到了一些问题。在下列
代码中使用了 ax，进入函数之前也没有对 ax 压栈进行保护，本以为该段代码

应该和按照 cmp 循环 5 次的时候停止，但是我通过在 inc ax 这条指令处加了一个断点，使用 continue 进行运行，经过我的测试直到几十上百次 continue 运行到该断点时才会跳出函数。

```
load_bootloader:
    call asm_read_hard_disk
    inc ax ;设置断点
    cmp ax, 5
    jle load_bootloader
```



```
14     mov cx, 0                ; 逻辑扇区号第16~31位
15     mov bx, 0x7e00           ; bootloader的加载地址
16     load_bootloader:
17         call asm_read_hard_disk ; 读取硬盘
B+> 18     inc ax
19         cmp ax, 5
20         jle load_bootloader
21         jmp 0x0000:0x7e00      ; 跳转到bootloader
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

remote Thread 1.1 In: load bootloader
Make breakpoint pending on future shared library load?
from terminal]
(gdb) c
Continuing.

Breakpoint 1, ?? () at mbr.asm:3
1: $ax = -21931
2: $cx = 0
(gdb) c 60
Will ignore next 59 crossings of breakpoint 1. Continuing.

Breakpoint 3, load_bootloader () at mbr.asm:18
1: $ax = 0
2: $cx = 0
(gdb) □
```

图 6-经过 60 个断点仍无输出

执行 60 次 continue，还未进入 0x7e00 函数

```
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     ; bootloader的加载地址
16    load_bootloader:
17        call asm_read_hard_disk ; 读取硬盘
B+> 18    inc ax
19        cmp ax, 5
20        jle load_bootloader
21        jmp 0x0000:0x7e00      ; 跳转到bootloader
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

run bootloader on rel-1.13.0-0
iPXE (http://ipxe.org) 00:03.0
Booting from Hard Disk...

remote Thread 1.1 In: load_bootloader
2: $cx = 0
(gdb) c 10
Will ignore next 9 crossings of breakpoint 3. Continuing.

Breakpoint 3, load_bootloader () at mbr.asm:18
1: $ax = 0
2: $cx = 0
(gdb) c 30
Will ignore next 29 crossings of breakpoint 3. Continuing.

Breakpoint 3, load_bootloader () at mbr.asm:18
1: $ax = 0
2: $cx = 0
Will ignore next 29 crossings of breakpoint 3. Continuing.
```

图 7-总共执行 100 次 continue 输出正确

又执行了 40 次 continue，最终才执行输出 run bootloader，非预期的方式获得了正确的结果。

下面我尝试加入了 push eax 和 pop eax 保护 ax 寄存器

```
load_bootloader:
    push eax
    call asm_read_hard_disk ; 读取硬盘
    pop eax
    inc ax
    cmp ax, 5
    jle load_bootloader
```

修改之后运行，虽然确实只执行了 5 次循环，但是最终输出的值却不是预期的。

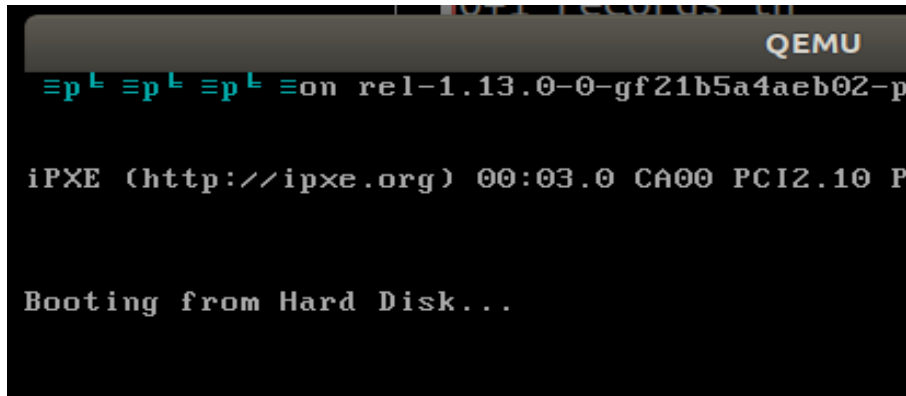


图 8-加入 pop 后的不正常输出

按道理来说，原来的 example 程序 call asmreadhard_disk 的时候会改变 ax 的值，也没有对 ax 的值进行压栈保护，循环也不是预期的 5 次，但是最终却能得到正确的值，这个问题直到现在也没有明白。

我终于懂了!!!

example 中这一段错误的代码能够输出正确的代码，完全是因为天大的巧合。

首先在调试这段代码中发现了两个疑点：

1.为什么 ax 始终为 0，但是经过很多次 continue 之后，突然能够成功跳出循环？

2.根据后面的 assignment1.2,就算我们读取了正确的值进入内存中，我们也不会得到正确的输出，这里为什么能正确输出？

这个巧合来自于两个部分

1.bx 在每次循环运行后恰好每次增加 512

2.cmp 判定 ax 的值的时候是根据 CPU 寄存器的标志位 ZF,SF 和 OF

两个巧合发生情况如下：

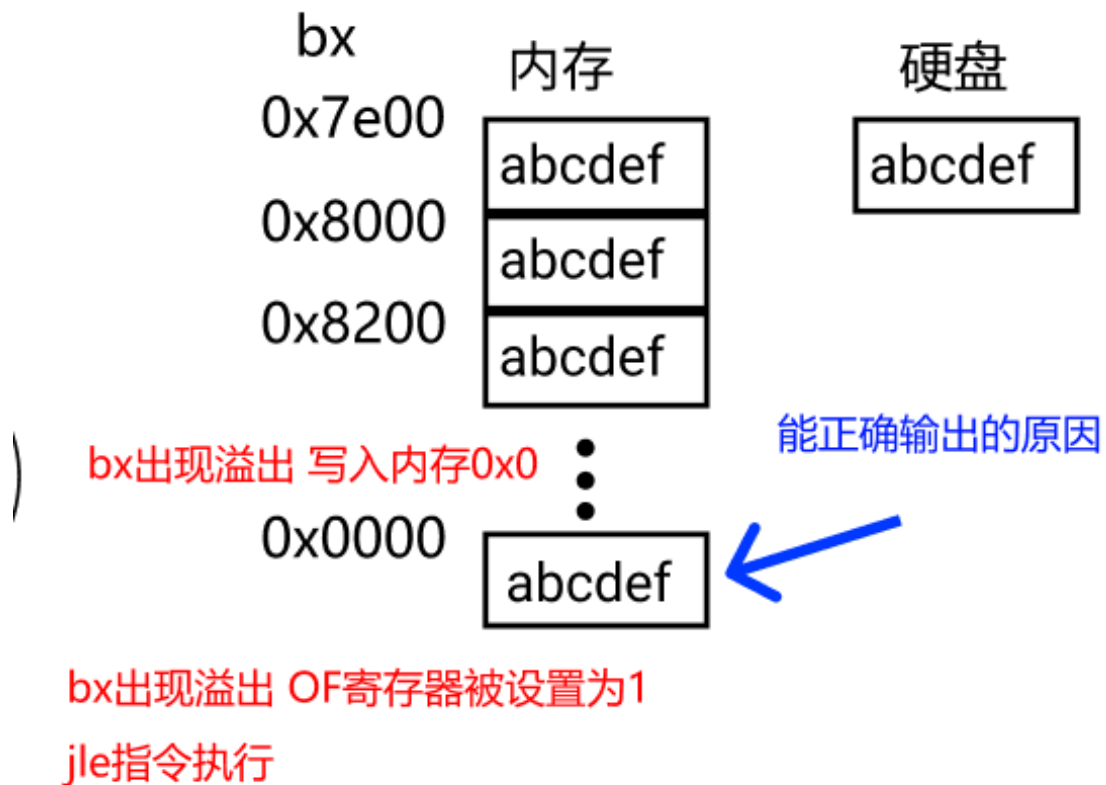


图 9-bx 写入内存的说明

CMP 结果	标志位
目的操作数 < 源操作数	SF ≠ OF
目的操作数 > 源操作数	SF=OF
目的操作数 = 源操作数	ZF=1

表 2-条件跳转指令如何判断是否跳转

首先 bx 一直循环，将硬盘中的数据读入内存中。接下来由于 bx 溢出，0x0000~0x100 的内存地址空间也被写入硬盘中的数据。同时，由于 bx 的溢出，CPU 产生 OF 标志位，使得 jle 指令被执行，跳转到了 0x7e00。在输出过程中，由于 0x26 内存空间确实是字符串的地址，所以字符串被成功输出。

1.2 实模式中断读取硬盘

在 Example1 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。现在，同学们需要将 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。最后说说你是怎么做的并提供结果截图，可以参考《于渊：一个操作系统的实现 2》P183-184。

按照 wiki 上关于使用实模式下中断 int 13h 通过 CHS 方式读取硬盘的参数如下

AH	02h
AL	Sectors To Read Count
CH	Cylinder
CL	Sector
DH	Head
DL	Drive
ES:BX	Buffer Address Pointer

表 3-实模式 int 13h 读取硬盘

我们修改 mbr.asm 函数

```
load_bootloader:
    call asm_read_hard_disk ; 读取硬盘
jmp 0x0000:0x7e00      ; 跳转到 bootloader
jmp $ ; 死循环
asm_read_hard_disk:    ;根据上表填写的参数
    mov ah,2
    mov al,5 ;直接读取 5 个扇区
    mov ch,0
    mov cl,2
    mov dh,0
    mov dl,80h ;设置为读取第一个硬盘
    int 13h
    nop ;加一个 nop 方便设置断点
    ret
```

然后 make run 执行

```
→ assignment1_int13 git:(main) X
→ assignment1_int13 git:(main) X make run

QEMU
on rel-1.13.0-0-gf21b5a4aeb02-prebu
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP P
```

图 9-和 assignment1.1 遇到相同结果

又现 assignment1.1 中的错误

出现了和实验 1 最后中一样的错误，debug 查看。

```
19      jmp $ ; 死循环
20      asm_read_hard_disk:
21          mov ah,2
22          mov al,5
23          mov ch,0
24          mov cl,2
25          mov dh,0
26          mov dl,80h
27          int 13h
28      brk:
B+> 29      nop
30      ret
31      times 510 - ($ - $$) db 0
32      db 0x55, 0xaa^?

remote Thread 1.1 In: brk L29 PC: 0x7c30
edi      0x0      0
eip      0x7c30   0x7c30 <brk>
eflags   0x247    [ IOPL=0 IF ZF PF CF ]
cs       0x0      0
ss       0x0      0
ds       0x0      0
es       0x0      0
fs       0x0      0
gs       0x0      0
---Type <return> to continue, or q <return> to quit---
Quit
(gdb) x/16a 0x7e00
0x7e00: 0x8eb80b58      0x6603b4e8      0xeb9   0xdb316600
0x7e10: 0x26be66      0x8a670000      0x7896506   0x83664666
0x7e20 <output_bootloader_tag+10>: 0xf2e20c3      0x7572feeb      0x6f62206e      0x6f6c746f
0x7e30: 0x72656461      0x0      0x0      0x0
(gdb)
```

图 10- 查看内存中的值正常

可见进行中断后 bootloader 中存在硬盘的值已经全部被读取进入内存，读取硬盘

这个步骤是没有问题的。接下来我们查看 bootloader 中的指令

```
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: ^?
19
20
21
22
23
24
25
26
27
28
29
30
31
32

remote Thread 1.1 In: L6
0x7e28:      outsb  %ds:(%esi),(%dx)
(gdb) ni
(gdb) x/13i $pc
⇒ 0x7e07:      mov     $0xe,%cx
0x7e0b:      add     %al,(%eax)
0x7e0d:      xor     %bx,%bx
0x7e10:      mov     $0x26,%si ←
0x7e14:      add     %al,(%eax)
0x7e16 <output_bootloader_tag>:      mov     -0x769b,%al
0x7e1b <output_bootloader_tag+5>:      pop     %es
0x7e1c <output_bootloader_tag+6>:      inc     %si
0x7e1e <output_bootloader_tag+8>:      add     $0x2,%bx
0x7e22 <output_bootloader_tag+12>:      loop    0x7e16 <output_bootloader_tag>
0x7e24 <output_bootloader_tag+14>:      jmp     0x7e24 <output_bootloader_tag+14>
0x7e26 <bootloader_tag>:      jbe     0x7e9d
0x7e28:      outsb  %ds:(%esi),(%dx)
(gdb) |
```

图 11-载入的 bootloader_tag 不正常

在这一条指令中发现了问题，按照需求我们应该需要这条指令为 bootloader_tag 的地址 0x7e26，但是这里却是 26。

从编译理解并修正错误

考虑到我们编译的过程就不难理解这里的问题。

1.使用 nasm 编译的时候，编译器直接按照本文件中的符号的地址进行编码为

0x26, 没有考虑偏移量。

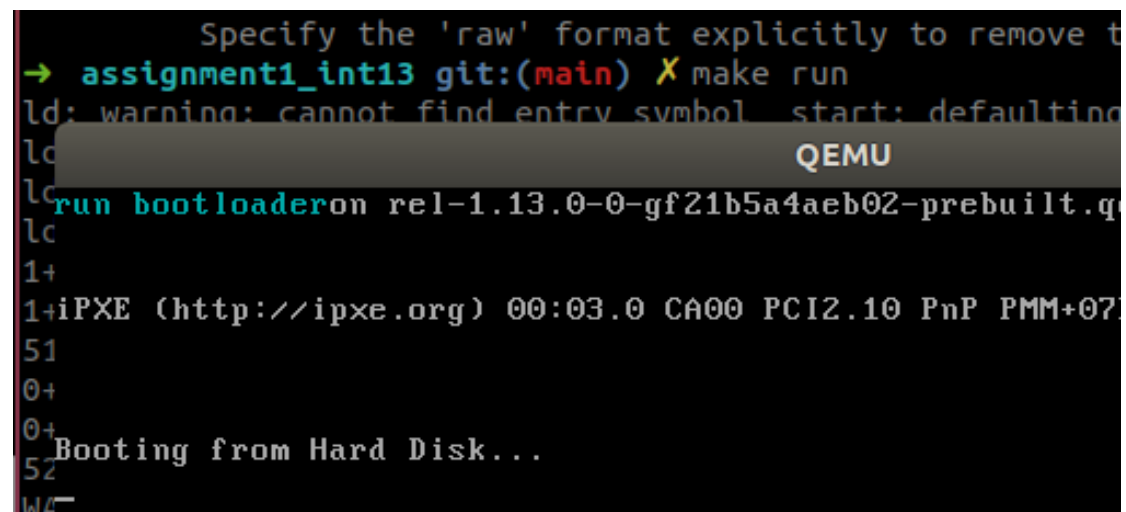
2.链接获取符号表的过程, 由于正确指定了-OText 0x7e00 , 获得的符号表正确

3.我们实际执行的过程为从 0x26 获取字符, 发生了错误。

解决方案: mov esi, bootloader_tag 的时候加上一个 0x7e00

```
mov ecx, bootloader_tag_end - bootloader_tag
xor ebx, ebx
mov esi, bootloader_tag+0x7e00;这里需要+0x7e00
output_bootloader_tag:
    mov al, [esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    loop output_bootloader_tag
jmp $ ; 死循环
```

再次 make run



```
Specify the 'raw' format explicitly to remove t
→ assignment1_int13 git:(main) X make run
ld: warning: cannot find entry symbol start; defaulting
lc
lc QEMU
lc run bootloader on rel-1.13.0-0-gf21b5a4aeb02-prebuilt.q
lc
1+
1+iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07
51
0+
0+
52 Booting from Hard Disk...
W/
```

图 12-修改代码后正常输出

成功输出 run bootloader,符合预期。

综上, 在 assignment1 两个实验中, 完成了使用 LBA 通过端口的方式读取硬盘,
与使用 int 13h 中断读取硬盘。

Assignment 2 开启保护模式

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。gdb 的使用可以参考 appendix 的“debug with gdb and qemu”部份。

根据题目要求，

1.准备 GDT，用 lgdt 指令加载 GDTR 信息。

```
25      ;建立保护模式下的堆栈段描述符
26      mov dword [GDT_START_ADDRESS+0x10],0x00000000      ; 基地址为0x00000000，界限0x0
27      mov dword [GDT_START_ADDRESS+0x14],0x00409600      ; 粒度为1个字节
28
29      ;建立保护模式下的显存描述符
30      mov dword [GDT_START_ADDRESS+0x18],0x00007fff      ; 基地址为0x000B8000，界限0x07FFF
31      mov dword [GDT_START_ADDRESS+0x1c],0x0040920b      ; 粒度为字节
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，段界限为0xFFFFF
35      mov dword [GDT_START_ADDRESS+0x24],0x00cf9800      ; 粒度为4kb，代码段描述符
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
43      out 0x92,al      ;打开A20
44
45      cli      ;中断机制尚未工作
46      mov eax,cr0
47      or eax,1

remote Thread 1.1 In: creat_gdt L38 PC: 0x7e7
0x8810: 0x0
(gdb) x/10a 0x8800
0x8800: 0x0 0x0 0xffff 0xcf9200
0x8810: 0x0 0x409600 0x80007fff 0x40920b
0x8820: 0x0 0x0
(gdb) ni
1: $ax = 882
2: /x $bx = 0x1c
3: $cx = 0
1: $ax = 882
2: /x $bx = 0x1c
3: $cx = 0
(gdb) x/10a 0x8800
0x8800: 0x0 0x0 0xffff 0xcf9200
0x8810: 0x0 0x409600 0x80007fff 0x40920b
0x8820: 0xffff 0xcf9800
(gdb)
```

图 13-查看 gdtr 信息

2.打开第 21 根地址线。对应代码为

```
in al,0x92      ;南桥芯片内的端口
or al,0000_0010B
out 0x92,al      ;打开 A20
```

通过向南桥芯片通信打开第 21 根地址线

3. 开启 cr0 的保护模式标志位。

cr0 个寄存器各个位上储存如下信息

bit	label	description
0	pe	protected mode enable
1	mp	monitor co-processor
2	em	emulation
3	ts	task switched
4	et	extension type
5	ne	numeric error
16	wp	write protect
18	am	alignment mask
29	nw	not-write through
30	cd	cache disable
31	pg	paging

表 4-CR0 寄存器各位意义

根据上表，为了打开保护模式，我们需要将 cr0 寄存器第 0 位设置为 1，对应代码：

```
mov eax,cr0
or eax,1
mov cr0,eax ;设置 PE 位
```

print \$eax 查看寄存器的值


```

45      cli                                ; 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      cr0_ok:
50      ; 以下进入保护模式
b+ 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
55      [bits 32]
56      protect_mode_begin:
57
b+ 58      mov eax, DATA_SELECTOR            ; 加载数据段(0..4G
59      mov ds, eax
60      mov es, eax
61      mov eax, STACK_SELECTOR
62      mov ss, eax

remote Thread 1.1 In: creat_gdt
(gdb) print /t $eax
$7 = 1100000010

```

图 14-执行前 CR0 寄存器最低位为 0

```

45      cli                                ; 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      cr0_ok:
50      ; 以下进入保护模式
b+ 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
55      [bits 32]
56      protect_mode_begin:
57
b+ 58      mov eax, DATA_SELECTOR            ; 加载数据段(
59      mov ds, eax
60      mov es, eax
61      mov eax, STACK_SELECTOR
62      mov ss, eax

remote Thread 1.1 In: creat_gdt
(gdb) print /t $eax
$19 = 10001

```

图 15-执行前 CR0 寄存器最低位为 1，打开保护模式

4.远跳转，进入保护模式。

```
B+> 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
55      [bits 32]
56      protect_mode_begin:
57
b+ 58      mov eax, DATA_SELECTOR          ;加载数据段
59      mov ds, eax
60      mov es, eax
61      mov eax, STACK_SELECTOR
62      mov ss, eax

remote Thread 1.1 In: cr0_ok

Breakpoint 6, cr0_ok () at bootloader.asm:51
1: $ax = 17
2: /x $bx = 0x1c
3: $cx = 0
(gdb) x/10i $pc
=> 0x7e9a <cr0_ok>:      jmp      0x0:0x7ea2
0x7ea0 <cr0_ok+6>:      and      BYTE PTR [eax],al
0x7ea2 <protect_mode_begin>: mov     eax,0x8
0x7ea7 <protect_mode_begin+5>:      mov     ds,eax
0x7ea9 <protect_mode_begin+7>:      mov     es,eax
0x7eab <protect_mode_begin+9>:      mov     eax,0x10
0x7eb0 <protect_mode_begin+14>:     mov     ss,eax
```

图 16-pc 跳转执行保护模式指令

由上图中可见的指令，pc 跳转到 0x7ea2 执行保护模式的第一条指令

```

B+> 58    mov eax, DATA_SELECTOR          ;加载数据段(0.
      59    mov ds, eax
      60    mov es, eax
      61    mov eax, STACK_SELECTOR
      62    mov ss, eax

remote Thread 1.1 In: protect_mode_begin
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      0x7ea2  0x7ea2 <protect_mode_begin>
eflags   0x6     [ IOPL=0 PF ]
cs       0x20    32
ss       0x0     0
ds       0x0     0
es       0x0     0
fs       0x0     0
gs       0xb800  47104

```

图 17-写入段寄存器前 cs,ss,ds,es 都为 0

```

> 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:
    71        mov al, [esi]
    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

remote Thread 1.1 In: protect_mode_begin
eax      0x18    24
ecx      0x0     0
edx      0x80    128
ebx      0x1c    28
esp      0x7c00  0x7c00
ebp      0x0     0x0
esi      0x7eec  32492
edi      0x0     0
eip      0x7eb9  0x7eb9 <protect_mode_begin+23>
eflags   0x6     [ IOPL=0 PF ]
cs       0x20    32
ss       0x10    16
ds       0x8     8
es       0x8     8
fs       0x0     0
gs       0x18    24

```

图 18-写入段寄存器后 cs,ss,ds,es 写入了段选择子

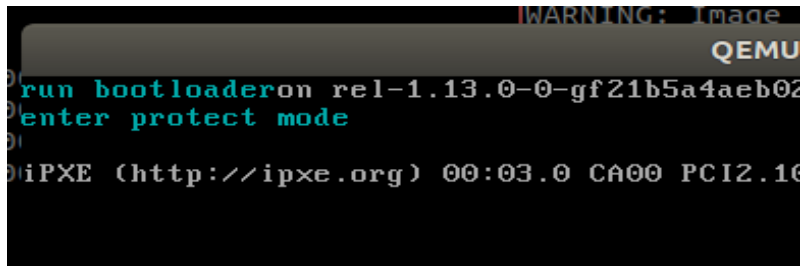


图 19-最终输出结果

最终输出 enter protect mode

综上，该实验中完成了从实模式到保护模式的转换。

Assignment 3

改造“Lab2-Assignment 4”为 32 位代码，即在保护模式后执行自定义

的汇编程序。

好在 32 位的保护模式仍然兼容 16 位程序，这里需要把原先定义的显存地址替换为 `eax`。把原来的程序放置在初始化段寄存器之后，直接执行即可。在该段代码之后填入之前已经实现的程序即可。

```
protect_mode_begin:
mov  eax, DATA_SELECTOR
mov  ds,  eax
mov  es,  eax
mov  eax, STACK_SELECTOR
mov  ss,  eax
mov  eax, VIDEO_SELECTOR
mov  gs,  eax
; 把自己之前实现的程序直接放在这个位置下面即可
```

make run 运行



图 20-将 lab2 assignment4 代码运行在保护模式下

综上，该实验完成了将原来实模式中运行的显示效果在保护模式中运行

实验感想

从本次实验中，assignment1 中学习了引导程序中硬盘读取的方法，分别是通过端口使用 24bitLBA 方式读写对应端口，和使用 int 13h 中断使用 CHS 寻址方式读取硬盘。assignment2 中，学习了有关 CPU 进入保护模式的流程，主要包括写入全局描述表，从 CR0 寄存器开始保护模式使能，载入段选择寄存器，长跳转到保护模式开始运行的流程。assignment3 中，学习了实模式下的代码与保护模式代码的兼容性。

对于 assignment1 中的 example 1 的代码是我最难理解的一次，为什么错误的代码跑出了正确的结果，经过两个小时调试仍然没有结果只能放弃。直到做到 assignment1.2 中，发现出现了一样的行为才意识到这不是玄学 bug，而是一个有迹可循的预期错误，最终从 bx 寄存器溢出的角度合理解释了 example1 中错误的

代码为什么能得到正确的输出结果。通过这次的 debug 理解:代码的运行没有玄学,一定是有它背后运行的原理在其中。

对比起来,在 assignment2 和 assignment3 中,整个过程资料详实,实验过程非常顺利,做了一次就跑出结果来了。但是显然收获也更少,例如,如果 GDT 写入错误会有什么后果? CR0 寄存器写入错误会有什么现象? 段选择子写入错误对程序运行有什么影响? 在计算机这样一个实践为重的专业中,比起来学习什么样是正确的方式,或许知道错误的现象是什么,怎样解决或许更加重要。

同时在本次的练习中,对于 gdb 的掌握更加熟练了通过查看当前的指令和内存地址与寄存器,对代码的运行过程有了更多调试的方法。学习了使用 gdb -x gdbinit 直接初始化不用手动复制粘贴指令。熟悉了对于 makefile 的使用,例如可以通过 makefile 中伪目标和创建需求的方式在 run 之前自动 clean 和 build

```
.PHONY:clean build debug run
clean:
    @ touch 1.bin
    @ rm -fr *.bin *.o *.symbol
build:clean
    @ nasm -g -f elf32 mbr.asm -o mbr.o
    @ ld -o mbr.symbol -melf_i386 -r mbr.o
    @ ld -o mbr.bin -melf_i386 -N mbr.o
    @ nasm -g -f elf32 bootloader.asm -o bootloader.o
    @ ld -o bootloader.symbol -melf_i386 -r bootloader.o
    @ ld -o bootloader.bin -melf_i386 -r bootloader.o
    @ dd if=mbr.bin of=hd.img bs=512
    @ dd if=bootloader.bin of=hd.img bs=512
run:build
    @ qemu-system-i386 -hda hd.img
debug:build
    @ qemu-system-i386 -s -S -hda h
```

图 21-makefile 指定前置任务,`.phony`指定伪目标

ubuntu 中的 terminator 也十分好用,可以获得和 windows 下 windows terminal 类似的体验

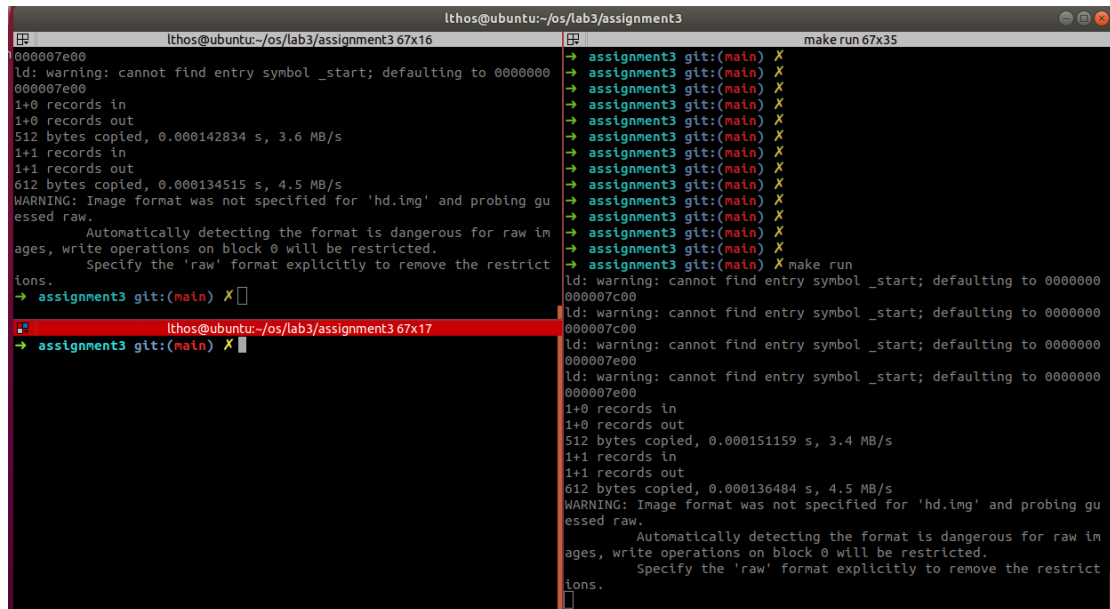


图 22-ubuntu 下 terminator, 支持分屏, 鼠标切换

参考资料

https://en.wikipedia.org/wiki/INT_13H

<https://en.wikipedia.org/wiki/Logicalblockaddressing>

<https://members.tripod.com/vitalyFilatov/ng/asm/asm024.3.html>

<https://stackoverflow.com/questions/55947714/how-do-i-fix-the-error-media-type-not-found-when-attempting-to-load-a-sector-w>

<https://www.win.tue.nl/~aeb/linux/kbd/A20.html>

<https://wiki.osdev.org/CPURegistersx86#CR0>

<https://en.wikipedia.org/wiki/GlobalDescriptorTable>