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

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：数据删除

学生学号：数据删除

实验成绩：

报告时间：2025 年 3 月 20 日

1. 实验要求

Assignment 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。
2. 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。

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

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。

2. 实验过程

Assignment 1

1.1

使用下发的 Example 代码，为了方便运行，编写如下所示的 Makefile 文件：

```
HD_IMG := ../../../../hd/hd.img
BUILD_DIR := ../../../../build

run:
    @qemu-system-i386 -hda $(HD_IMG) -serial null -parallel stdio
build:
    @nasm -f bin mbr.asm -o $(BUILD_DIR)/mbr.bin
    @nasm -f bin bootloader.asm -o $(BUILD_DIR)/bootloader.bin
    @dd if=$(BUILD_DIR)/mbr.bin of=$(HD_IMG) bs=512 count=1 seek=0
conv=notrunc
    @dd if=$(BUILD_DIR)/bootloader.bin of=$(HD_IMG) bs=512 count=5
seek=1 conv=notrunc
clean:
    @rm $(BUILD_DIR)/*.bin
```

在代码目录运行：

```
make build
make run
```

qemu 会运行并输出预期结果。

1.2

计算可得 CHS 访问磁盘 bootloader 的磁头号为 0，柱面号为 0，扇区号为 2~6，

因为 bootloader 没有跨柱面或磁头，因此可以一次性读 5 个扇区。

为了方便调试，将 Makefile 文件修改如下：

```
HD_IMG := ../../../../hd/hd.img
BUILD_DIR := ../../../../build
MBR_O := $(BUILD_DIR)/mbr.o
BL_O := $(BUILD_DIR)/bootloader.o
```

```
MBR_SYMBOL := $(BUILD_DIR)/mbr.symbol
BL_SYMBOL := $(BUILD_DIR)/bootloader.symbol
MBR_BIN := $(BUILD_DIR)/mbr.bin
BL_BIN := $(BUILD_DIR)/bootloader.bin

run:
    @qemu-system-i386 -hda $(HD_IMG) -serial null -parallel stdio
debug:
    @gnome-terminal -e "qemu-system-i386 -s -S -hda $(HD_IMG) -
serial null -parallel stdio"
build:
    @nasm -g -f elf32 mbr.asm -o $(MBR_O)
    @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 -g -f elf32 bootloader.asm -o $(BL_O)
    @ld -o $(BL_SYMBOL) -melf_i386 -N $(BL_O) -Ttext 0x7e00
    @ld -o $(BL_BIN) -melf_i386 -N $(BL_O) -Ttext 0x7e00 --oformat
binary

    @dd if=$(MBR_BIN) of=$(HD_IMG) bs=512 count=1 seek=0
conv=notrunc
    @dd if=$(BL_BIN) of=$(HD_IMG) bs=512 count=5 seek=1
conv=notrunc
clean:
    @rm $(BUILD_DIR)/*.bin $(BUILD_DIR)/*.o $(BUILD_DIR)/*.symbol
```

我使用 vscode+ssh 远程调试，因此编写以下 launch.json 文件：

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug",
      "type": "gdb",
      "request": "attach",
      "target": ":1234",
      "cwd": "${fileDirname}",
      "remote": true,
      "valuesFormatting": "parseText",
      "autorun": [
        "set output-radix 16",
        "set disassembly-flavor intel",
        "add-symbol-file /path/to/mbr.symbol 0x7c00",
```

```
        "add-symbol-file /path/to/bootloader.symbol 0x7e00",
      ],
      "preLaunchTask": "make build & debug here"
    }
  ]
}
```

cwd 修改为${fileDirname}而不是${workspaceRoot}，因为每个 assignment 的代码并不是在根目录存放，而经过多次尝试，得到结论：调试时的工作目录必须和代码目录一致（也可能是执行 make build）时的目录一致。

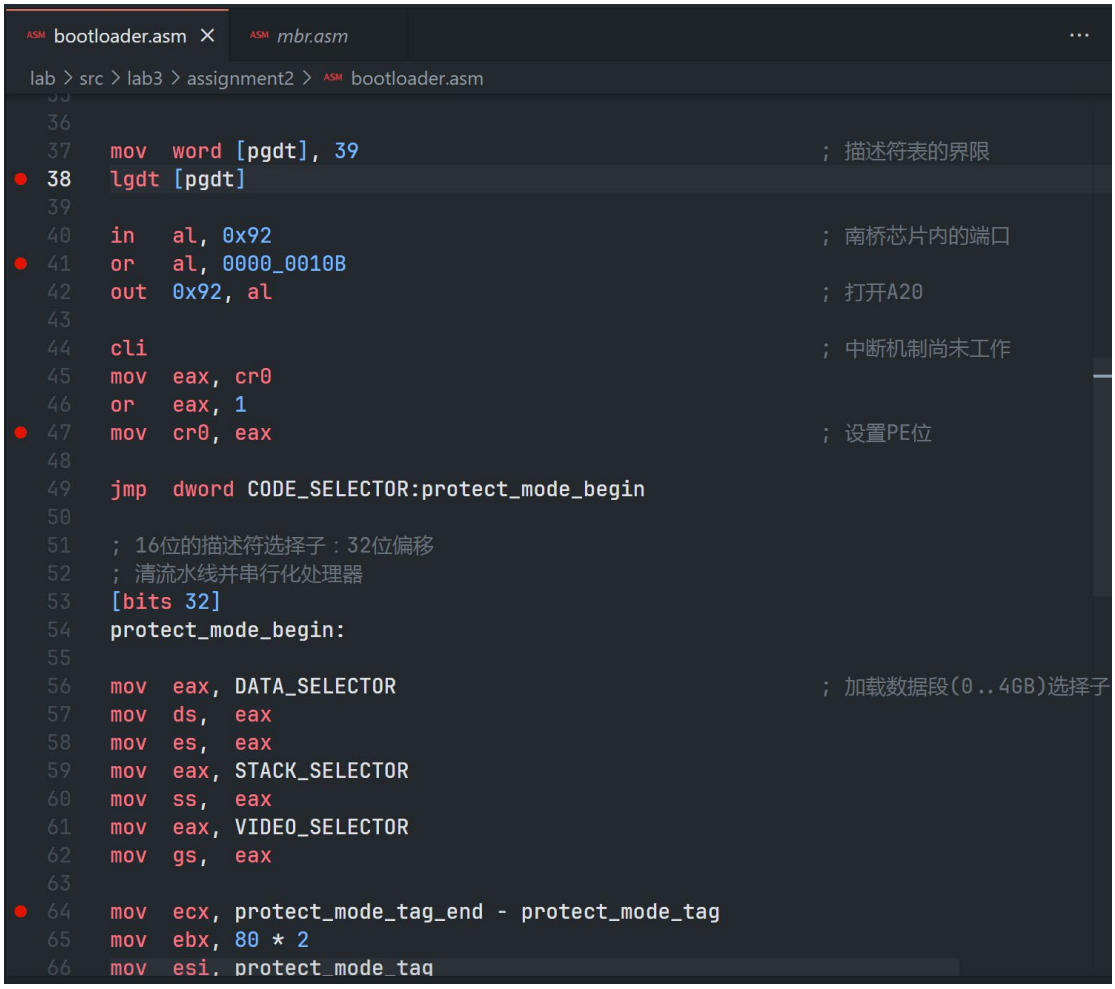
对应的 preLaunchTask 如下所示：

```
{
  "label": "make build & debug here",
  "type": "shell",
  "command": "cd ${fileDirname} && make build && make debug",
  "problemMatcher": []
}
```

然后就可以按 f5 一键编译并启动调试。

Assignment 2

使用下发的 Example 代码，在 vscode 在进入保护模式的 4 个重要步骤上设置断点，如图：



开启调试，程序成功在停止在第一个断点。

然而 lgdt 的运行结果不知道怎么看，所以略过。

尝试打开 A20，发现在本机 qemu 下该地址线是默认打开的。

找了下发现个相关的文章：

OS boot 的时候为什么要 enable A20？ - 北极的回答 - 知乎

https://www.zhihu.com/question/29375534/answer/44144613

打开 cr0 保护模式标记位，从监视窗口可以发现 cr0 多出一个 PF 标志位，表示保护模式标志位已置位。

远跳转进入保护模式，加载段选择子。从监视窗口可以看到段寄存器的内容变成了指定的段选择子，程序正常工作。

Assignment 3

修改 Lab 2 Assignment 4 最大的挑战在于，32 位保护模式无法调用 BIOS 中断来完成输出/延时等功能，因此需要自己实现。

考虑将功能封装成函数。引用上次实验报告中的实现思路：

> 程序要实现的功能分为两部分：渲染反弹的数字、渲染标题。从示例视频看标题不会被反弹的数字覆盖，所以为了方便起见，每次都完整渲染一次标题。
>
> 渲染反弹的数字在反弹逻辑设计上如果直接分四个状态做会比较困难。不妨拆分成水平方向速度和垂直方向速度，独立地检测水平和垂直是否即将要出界，如果是，则将对应方向的速度取负数即可。
>
> 剩下的部分比较简单：要渲染变化的数字，直接用字符串“1357924680”加上一个循环的下标即可；渲染变化的颜色直接用一个 byte，每次加一即可。

稍加分析可知，需要拆解出的函数有：清屏、延时、输出标题、在指定位置输出字符、更新下一次要输出字符的位置和方向向量。因为延时无法用中断，因此采用循环计数实现；除此之外将寄存器改为 32 位即可。

3. 关键代码

Assignment 1

1.2

```
; 读取磁盘
mov   ah, 0x02            ; 功能号
mov   al, 5               ; 扇区数
mov   ch, 0               ; 柱面
```

```
mov   cl, 2               ; 扇区
mov   dh, 0               ; 磁头
mov   dl, 0x80            ; 驱动器
mov   bx, 0x7e00          ; 缓冲区地址
int   0x13

jc    load_error          ; 检测 CF 是否不为 0
jmp   0x0000:0x7e00       ; 跳转到 bootloader

load_error:
jmp   $                   ; 死循环
```

Assignment 2

```
%include"boot.inc"
[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

mov  dword [GDT_START_ADDRESS+0x00], 0x00
mov  dword [GDT_START_ADDRESS+0x04], 0x00

; 创建描述符，这是一个数据段，对应 0~4GB 的线性地址空间
mov  dword [GDT_START_ADDRESS+0x08], 0x0000ffff          ; 基地址为
0，段界限为 0xFFFFF
mov  dword [GDT_START_ADDRESS+0x0c], 0x00cf9200          ; 粒度为
4KB，存储器段描述符

                                                        ; 建立保护模式下
的堆栈段描述符
mov  dword [GDT_START_ADDRESS+0x10], 0x00000000          ; 基地址为
0x00000000，界限 0x0
mov  dword [GDT_START_ADDRESS+0x14], 0x00409600          ; 粒度为 1
个字节
```

```
                                                    ; 建立保护模式下
的显存描述符
mov  dword [GDT_START_ADDRESS+0x18], 0x80007fff          ; 基地址为
0x000B8000，界限 0x07FFF
mov  dword [GDT_START_ADDRESS+0x1c], 0x0040920b          ; 粒度为字
节

                                                    ; 创建保护模式下
平坦模式代码段描述符
mov  dword [GDT_START_ADDRESS+0x20], 0x0000ffff          ; 基地址为
0，段界限为 0xFFFFF
mov  dword [GDT_START_ADDRESS+0x24], 0x00cf9800          ; 粒度为
4kb，代码段描述符



mov  word [pgdt], 39                                ; 描述符表的界
限
lgdt [pgdt]

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

cli                                                 ; 中断机制尚未
工作
mov  eax, cr0
or   eax, 1
mov  cr0, eax                                       ; 设置 PE 位

jmp  dword CODE_SELECTOR:protect_mode_begin

; 16 位的描述符选择子：32 位偏移
; 清流水线并串行化处理器
[bits 32]
protect_mode_begin:

mov  eax, DATA_SELECTOR                             ; 加载数据段
(0..4GB)选择子
mov  ds,  eax
mov  es,  eax
mov  eax, STACK_SELECTOR
mov  ss,  eax
```

```
mov  eax, VIDEO_SELECTOR
mov  gs,  eax

mov  ecx, protect_mode_tag_end - protect_mode_tag
mov  ebx, 80 * 2
mov  esi, protect_mode_tag
mov  ah,  0x3
output_protect_mode_tag:
    mov  al,          [esi]
    mov  word[gs:ebx], ax
    add  ebx,         2
    inc  esi
    loop output_protect_mode_tag

jmp $                                                   ; 死循环


bootloader_tag db 'run bootloader'
bootloader_tag_end:

pgdt          dw 0
              dd GDT_START_ADDRESS

protect_mode_tag db 'enter protect mode'
protect_mode_tag_end:
```

Assignment 3

```
call clear_screen
; 用清屏函数

while:
    ; 输出第一个字符
    movzx eax, byte [pos_1]
    movzx ebx, byte [pos_1 + 1]
    movzx ecx, byte [string_index_1]
    movzx edx, byte [color_1]
    call display_char

    ; 更新颜色
    inc byte [color_1]

    ; 更新字符索引
    inc byte [string_index_1]
```

```
    movzx eax, byte [string_index_1]
    mov bl, byte [string + eax]
    cmp bl, 0
    jne reset_index_end_1
    mov byte [string_index_1], 0
    reset_index_end_1:

    ; 更新第一个字符的位置
    call update_position_1

    ; 输出第二个字符
    movzx eax, byte [pos_2]
    movzx ebx, byte [pos_2 + 1]
    movzx ecx, byte [string_index_2]
    movzx edx, byte [color_2]
    call display_char

    ; 更新颜色
    inc byte [color_2]

    ; 更新字符索引
    inc byte [string_index_2]
    movzx eax, byte [string_index_2]
    mov bl, byte [string + eax]
    cmp bl, 0
    jne reset_index_end_2
    mov byte [string_index_2], 0
    reset_index_end_2:

    ; 更新第二个字符的位置
    call update_position_2

    ; 显示标题
    call display_title

    ; 延时
    mov ecx, 0x500000
    delay_loop:
        loop delay_loop

    jmp while

clear_screen:
    push eax
```

```
    push ecx
    push edi

    mov eax, 0x0720
    mov ecx, 80*25
    mov edi, 0

    clear_loop:
        mov word [gs:edi], ax
        add edi, 2
        loop clear_loop

    pop edi
    pop ecx
    pop eax
    ret


display_char:
    ; eax: 行号
    ; ebx: 列号
    ; ecx: 字符索引
    ; edx: 颜色
    push eax
    push ebx
    push ecx
    push edx
    push edi

    ; 计算显存偏移
    imul eax, 80
    add eax, ebx
    shl eax, 1
    mov edi, eax

    ; 获取字符
    mov al, byte [string + ecx]
    mov ah, dl

    ; 写入显存
    mov word [gs:edi], ax

    pop edi
    pop edx
    pop ecx
```

```
    pop ebx
    pop eax
    ret


update_position_1:
    push eax
    push ebx

    ; 计算新位置
    mov al, byte [pos_1]
    mov bl, byte [pos_1 + 1]
    add al, byte [velocity_1]
    add bl, byte [velocity_1 + 1]

    ; 检查行边界
    cmp al, 0
    jl reverse_velocity_x_1
    cmp al, 24
    jg reverse_velocity_x_1
    jmp reverse_velocity_x_end_1

    reverse_velocity_x_1:
        neg byte [velocity_1]
        mov al, byte [pos_1]
        add al, byte [velocity_1]
    reverse_velocity_x_end_1:

    ; 检查列边界
    cmp bl, 0
    jl reverse_velocity_y_1
    cmp bl, 79
    jg reverse_velocity_y_1
    jmp reverse_velocity_y_end_1

    reverse_velocity_y_1:
        neg byte [velocity_1 + 1]
        mov bl, byte [pos_1 + 1]
        add bl, byte [velocity_1 + 1]
    reverse_velocity_y_end_1:

    ; 保存新位置
    mov byte [pos_1], al
    mov byte [pos_1 + 1], bl
```

```
    pop ebx
    pop eax
    ret


update_position_2:
    push eax
    push ebx

    ; 计算新位置
    mov al, byte [pos_2]
    mov bl, byte [pos_2 + 1]
    add al, byte [velocity_2]
    add bl, byte [velocity_2 + 1]

    ; 检查行边界
    cmp al, 0
    jl reverse_velocity_x_2
    cmp al, 24
    jg reverse_velocity_x_2
    jmp reverse_velocity_x_end_2

    reverse_velocity_x_2:
        neg byte [velocity_2]
        mov al, byte [pos_2]
        add al, byte [velocity_2]
    reverse_velocity_x_end_2:

    ; 检查列边界
    cmp bl, 0
    jl reverse_velocity_y_2
    cmp bl, 79
    jg reverse_velocity_y_2
    jmp reverse_velocity_y_end_2

    reverse_velocity_y_2:
        neg byte [velocity_2 + 1]
        mov bl, byte [pos_2 + 1]
        add bl, byte [velocity_2 + 1]
    reverse_velocity_y_end_2:

    ; 保存新位置
    mov byte [pos_2], al
    mov byte [pos_2 + 1], bl
```

```
    pop ebx
    pop eax
    ret

display_title:
    push eax
    push esi
    push edi

    mov eax, 0
    mov esi, 0
    mov edi, 64

    display_title_loop:
        mov cl, byte [title + esi]
        cmp cl, 0
        je display_title_end

        ; 写入字符到显存
        mov al, cl
        mov ah, 2
        mov word [gs:edi], ax

        inc esi
        add edi, 2
        jmp display_title_loop

    display_title_end:

    pop edi
    pop esi
    pop eax
    ret

jmp $

title db 'test 00000000',0
string db '1357924680',0

string_index_1 dw 0
color_1 db 0
pos_1 db 2,0
velocity_1 db 1,1
```

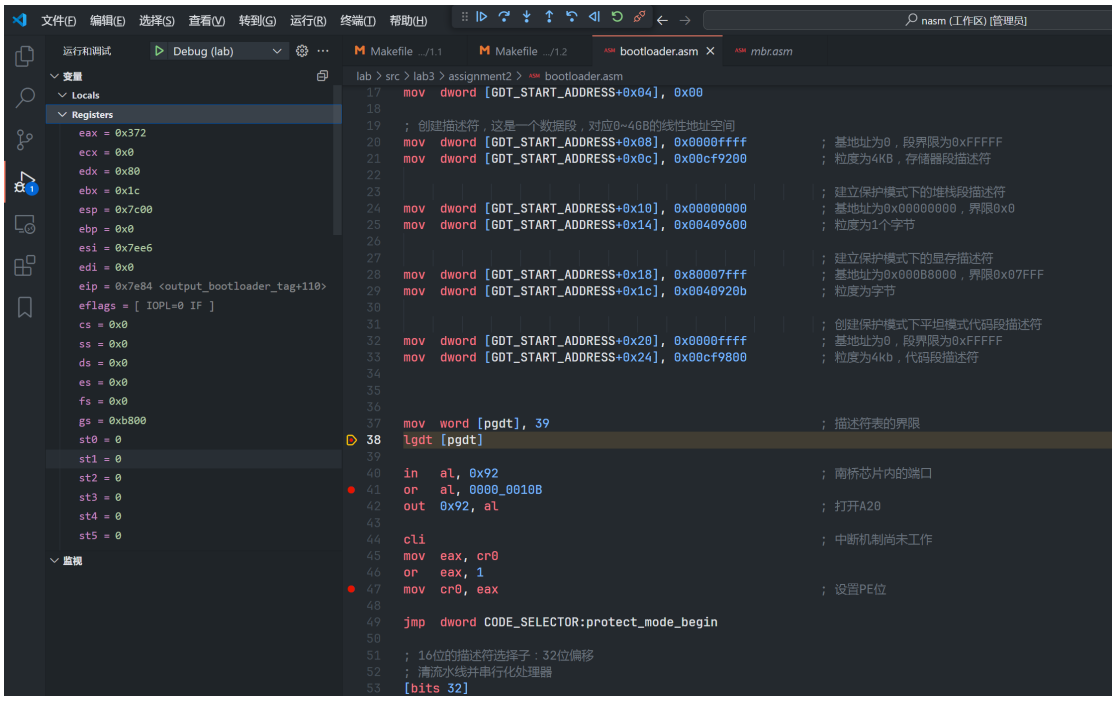
```
string_index_2 dw 0
color_2 db 0x21
pos_2 db 3,79
velocity_2 db -1,-1
```

4. 实验结果

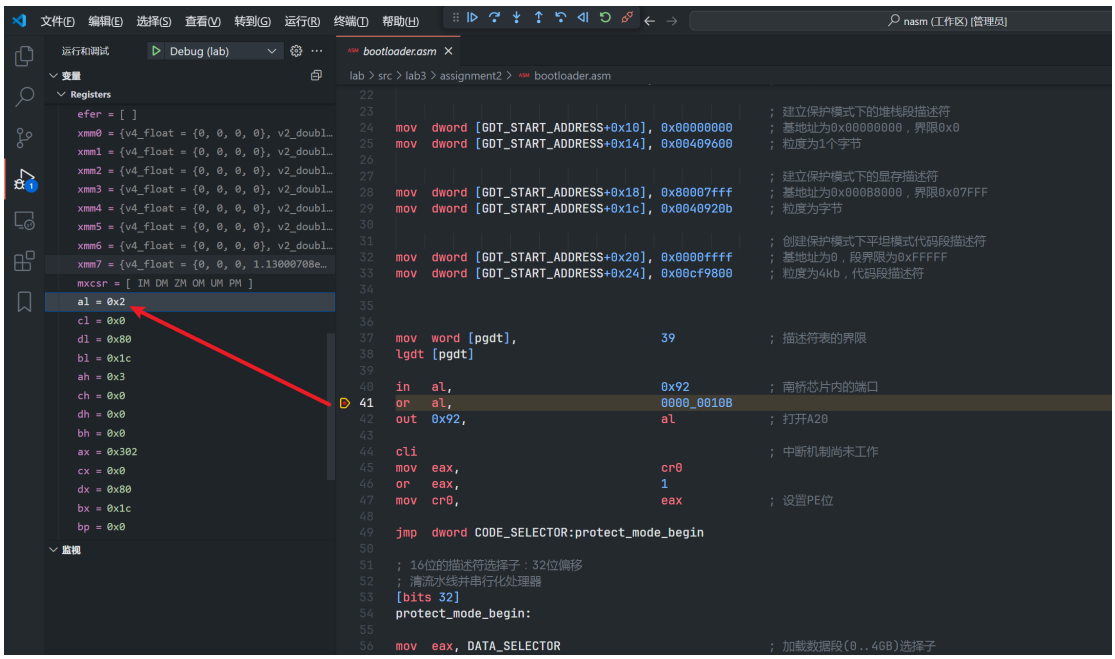
Assignment 1

Assignment 2

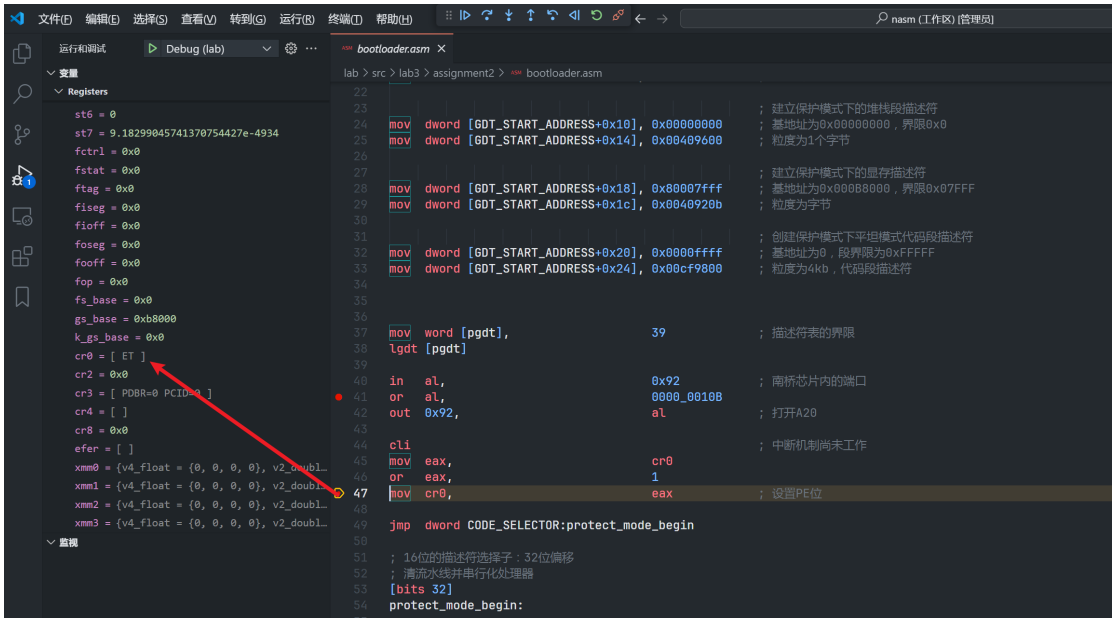
加载 gdt:

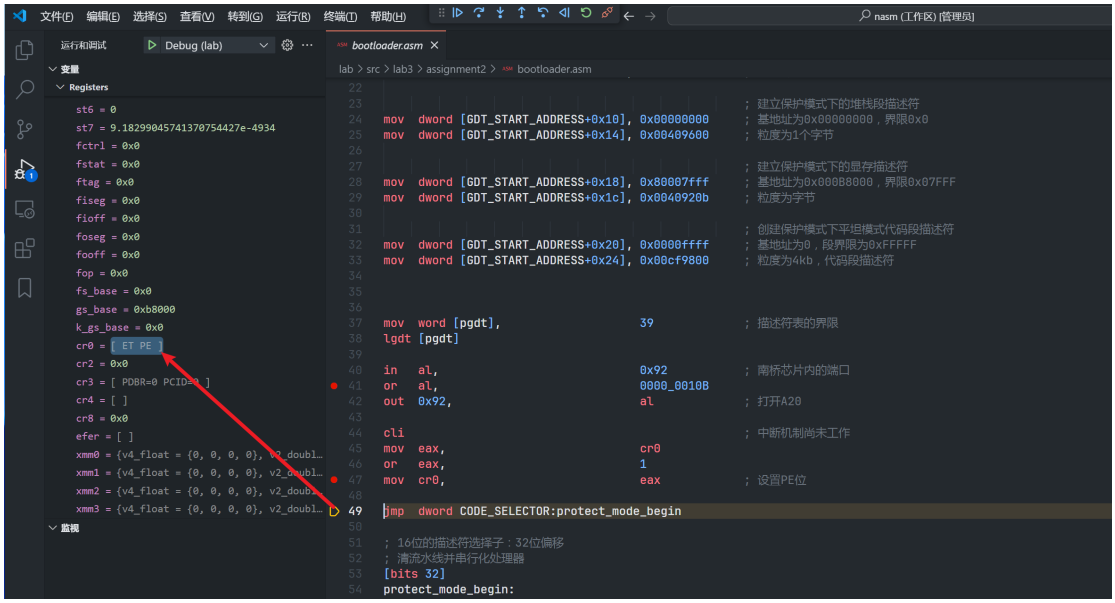


尝试打开 A20：

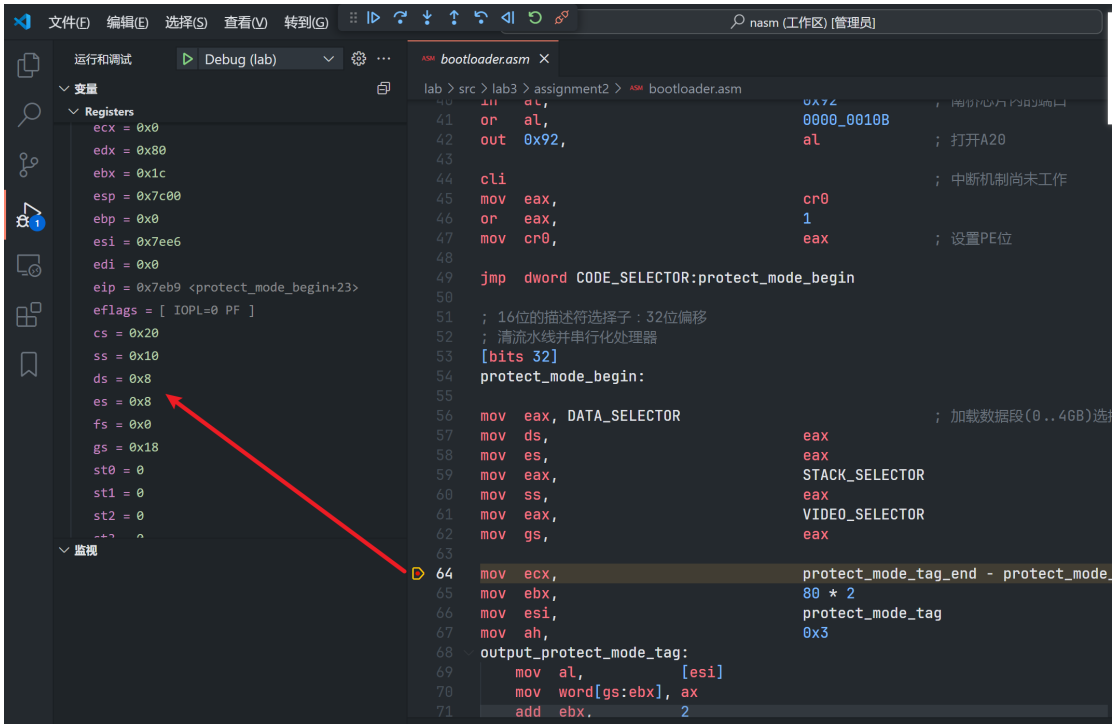


打开 cr0 保护模式标记位：

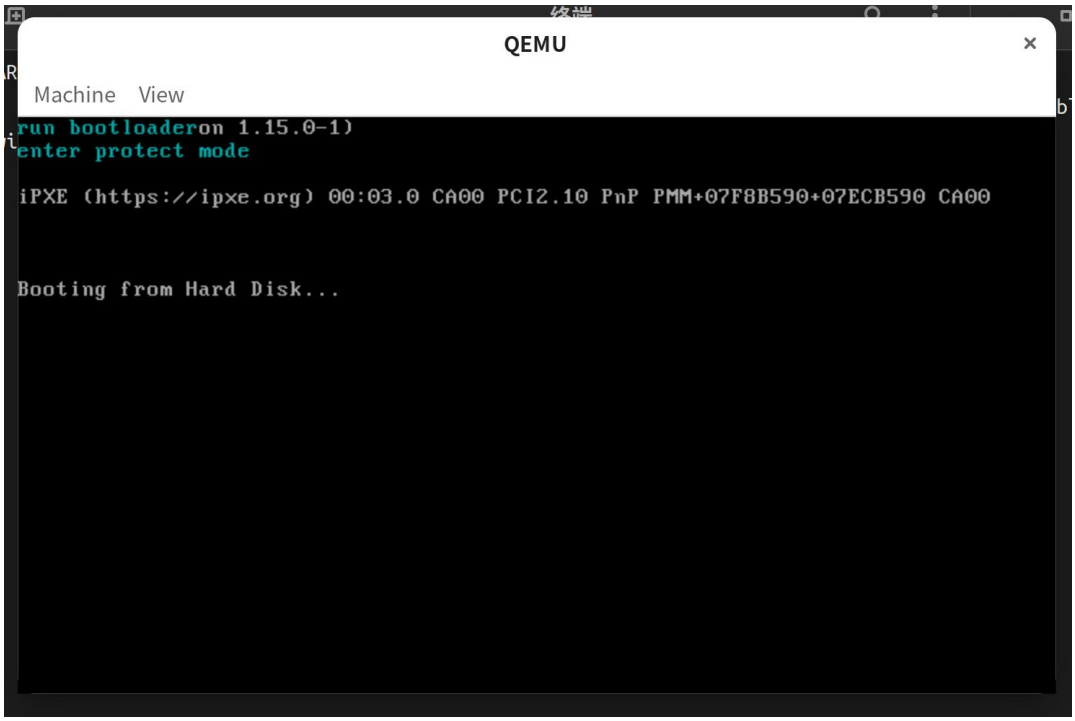




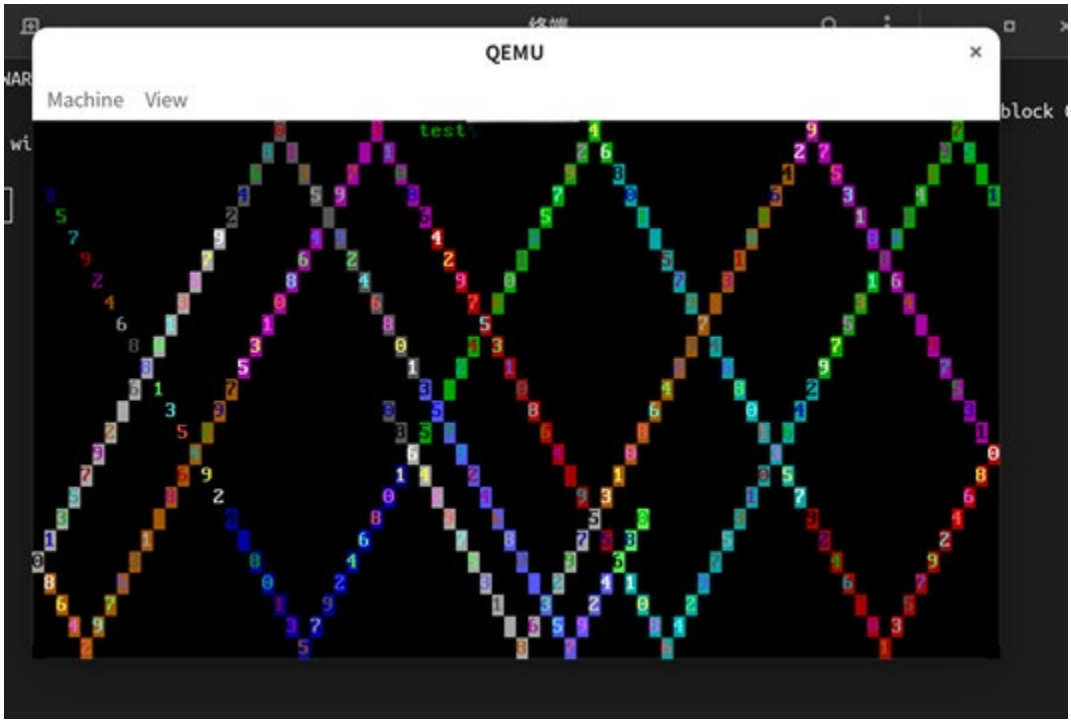
远跳转进入保护模式，加载段选择子：


文件(F) 编辑(E) 选择(S) 查看(V) 转到(G)
nasm (工作区) [管理员]
运行和调试
Debug (lab)
bootloader.asm
lab > src > lab3 > assignment2 > bootloader.asm
变量
Registers
ecx = 0x0
edx = 0x80
ebx = 0x1c
esp = 0x7c00
ebp = 0x0
esi = 0x7ee6
edi = 0x0
eip = 0x7eb9 <protect_mode_begin+23>
eflags = [IOPL=0 PF]
cs = 0x20
ss = 0x10
ds = 0x8
es = 0x8
fs = 0x0
gs = 0x18
st0 = 0
st1 = 0
st2 = 0
监视
41 or al, 0000_0010B
42 out 0x92, al ; 打开A20
44 cli ; 中断机制尚未工作
45 mov eax, cr0
46 or eax, 1
47 mov cr0, eax ; 设置PE位
49 jmp dword CODE_SELECTOR:protect_mode_begin
51 ; 16位的描述符选择子：32位偏移
52 ; 清流水线并串行化处理器
53 [bits 32]
54 protect_mode_begin:
56 mov eax, DATA_SELECTOR ; 加载数据段(0..4GB)选
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
64 mov ecx, protect_mode_tag_end - protect_mode_
65 mov ebx, 80 * 2
66 mov esi, protect_mode_tag
67 mov ah, 0x3
68 output_protect_mode_tag:
69 mov al, [esi]
70 mov word[gs:ebx], ax
71 add ebx, 2

运行结果：


QEMU
Machine View
run bootloaderon 1.15.0-1)
enter protect mode
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00
Booting from Hard Disk...

Assignment 3



5. 总结

这次实验学习了如何从 16 位实模式跳转到 32 位保护模式，也初步学会了如何与 IO 设备交互。实验过程中最大的挑战是如何配置远程调试，配置过程中一度出现配置与指导书几乎完全相同但是跑不通的情况，最后发现是磁盘文件使用的是上一节的大小（512 字节，仅能容纳 mbr 程序），除此之外还遇到了调试器工作目录没配置好导致无法找到源代码等问题，这些问题看似微小，却耗费了大量时间进行排查。希望在接下来的实验中能够对配置的细节进行细致的遵守。