



本科生实验报告

实验课程: 操作系统

实验名称: 从实模式到保护模式

专业名称: 信息与计算科学

学生姓名: 张文沁

学生学号: 20337268

实验地点:

实验成绩:

报告时间: 2022.3.28

1. 实验要求

- DDL: 2021年3月30号 23:59
- 提交的内容: 将3个assignment的代码和实验报告放到压缩包中, 命名为“lab3-姓名-学号”, 并交到课程网站上[\[http://course.dds-sysu.tech/course/3/homework\]](http://course.dds-sysu.tech/course/3/homework)
- 材料的Example的代码放置在 `src` 目录下。

1. 实验不限语言，C/C++/Rust都可以。
2. 实验不限平台，Windows、Linux和MacOS等都可以。
3. 实验不限CPU，ARM/Intel/Risc-V都可以。

2. 实验过程

Assignment1:

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。

1.2

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。

$$CS = 0, HS = 0, SS = 1, PS = 63, PH = 255$$

CHS到LBA

$$LBA = (C-CS) \times PH \times PS + (H-HS) \times PS + (S-SS)$$

LBA到CHS

$$C = LBA / (PH \times PS) + C$$

$$H = (LBA / PS) \% PH + HS$$

$$S = LBA \% PS + SS$$

Assignment2:

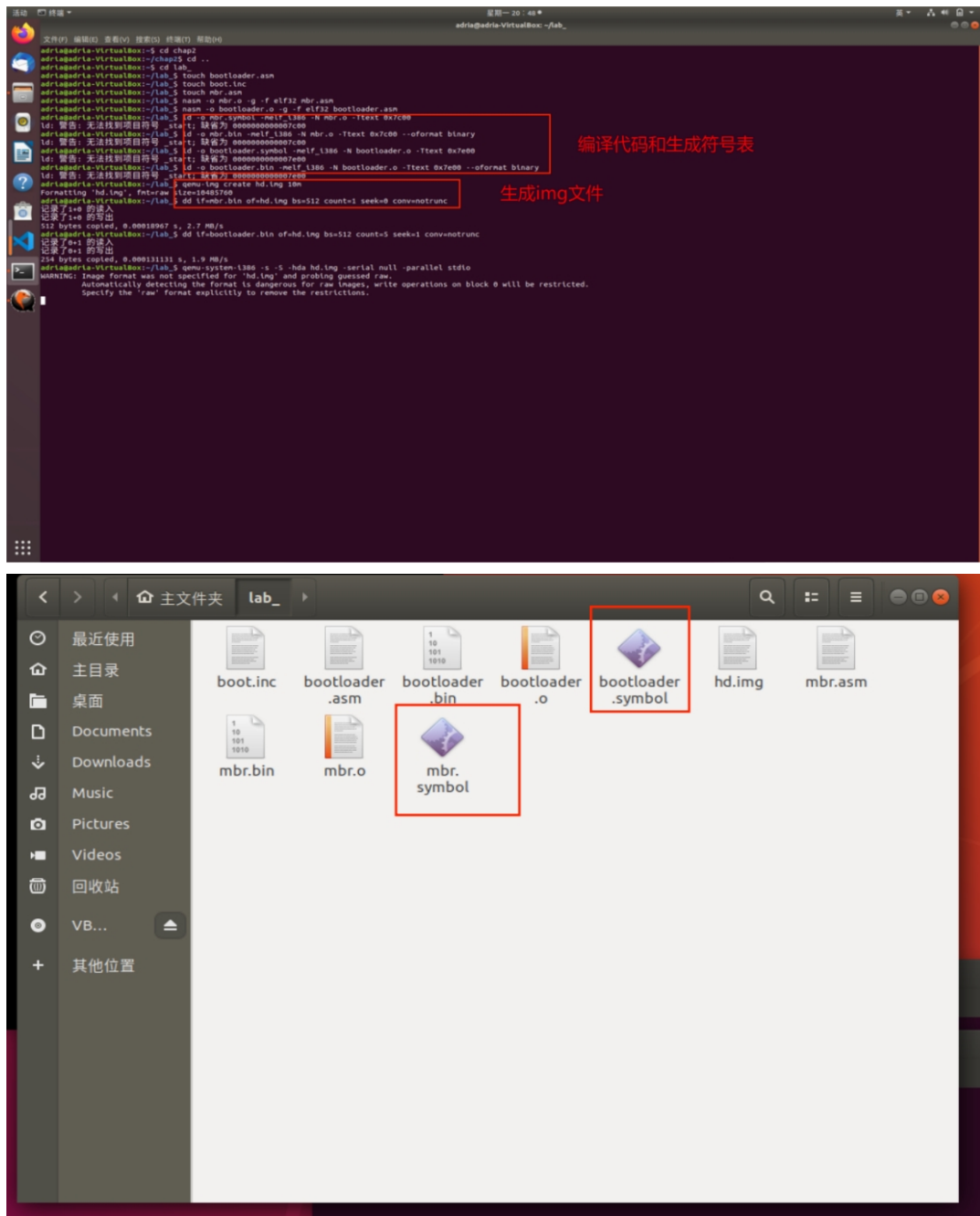
复现Example 2，使用gdb或其他debug工具在进入保护模式的4个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这4个步骤，最后附上结果截图。gdb的使用可以参考appendix的“debug with gdb and qemu”部份。

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。

5. 调试

注：因为图片直接插入Typora会导致失真，于是另外附上pdf文档（Assignment2.pdf）解释本任务

1. 生成符号表和img文件



2. 设置断点查看寄存器表

第一个断点(0x7c00)处的寄存器表

```
adria@adria-VirtualBox: ~/lab_
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) info registers
eax          0xaa55      43605
ecx          0x0         0
edx          0x80        128
ebx          0x0         0
esp          0x6f04      0x6f04
ebp          0x0         0x0
esi          0x0         0
edi          0x0         0
eip          0x7c00      0x7c00
eflags       0x202       [ IF ]
cs           0x0         0
ss           0x0         0
ds           0x0         0
es           0x0         0
fs           0x0         0
gs           0x0         0
(gdb)
```

第二个断点(0x7e00)处的寄存器表

```
remote Thread 1 In:
(gdb) add-symbol-file mbr.symbol 0x7c00
add symbol table from file "mbr.symbol" at
      .text_addr = 0x7c00
(y or n) y
Reading symbols from mbr.symbol...done.
(gdb) b *0x7e00
Breakpoint 2 at 0x7e00
(gdb) info registers
eax          0xaa55      43605
ecx          0x0         0
edx          0x80        128
ebx          0x0         0
esp          0x6f04      0x6f04
ebp          0x0         0x0
esi          0x0         0
edi          0x0         0
eip          0x7c00      0x7c00
eflags       0x202       [ IF ]
cs           0x0         0
ss           0x0         0
ds           0x0         0
es           0x0         0
fs           0x0         0
gs           0x0         0
(gdb)
```

第二个断点处

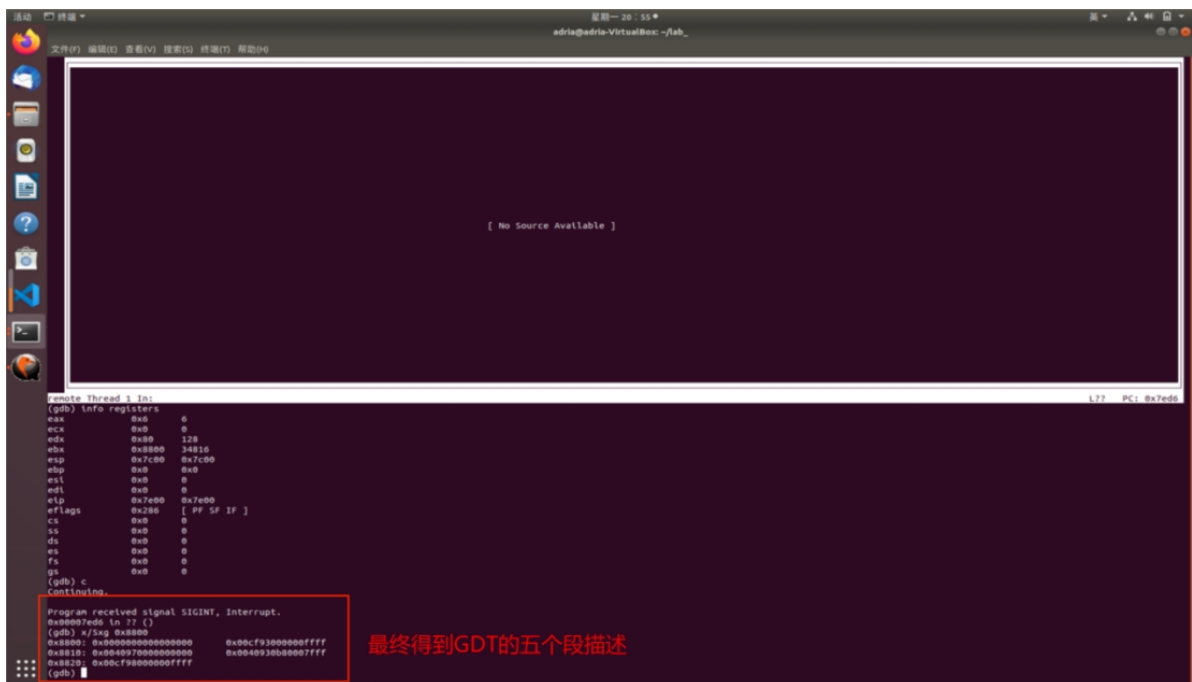
第三个断点 (protect_mode_begin) 处的寄存器表

```
remote Thread 1 In:
gs          0x0      0
(gdb) b protect_mode_begin
Function "protect_mode_begin" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 3 (protect_mode_begin) pending.
(gdb) c
Continuing.

Breakpoint 2, 0x00007e00 in ?? ()
(gdb) info registers
eax          0x6      6
ecx          0x0      0
edx          0x80     128
ebx          0x8800   34816
esp          0x7c00   0x7c00
ebp          0x0      0
esi          0x0      0
edi          0x0      0
eip          0x7e00   0x7e00
eflags      0x286    [ PF SF IF ]
cs           0x0      0
ss           0x0      0
ds           0x0      0
es           0x0      0
fs           0x0      0
gs           0x0      0
(gdb)
```

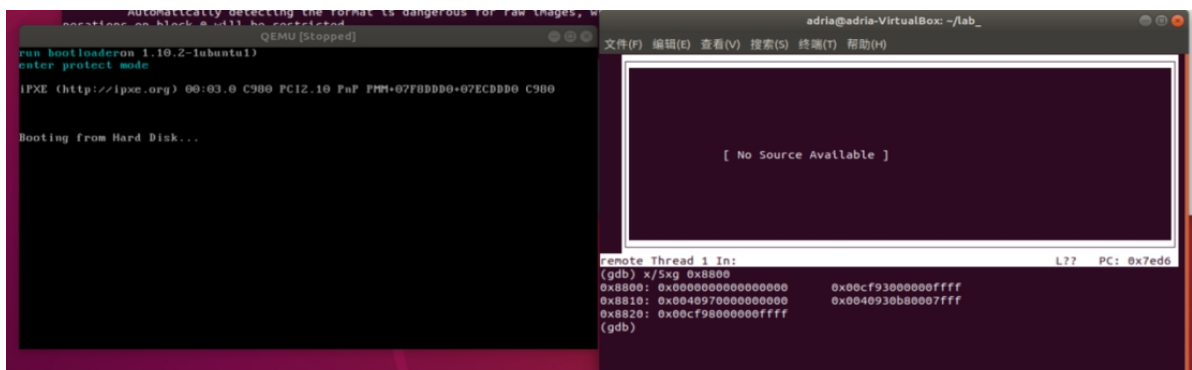
3. 查看GDT五个段描述是否和初始设置内容一致

最终GDT的五个段描述



```
remote Thread 1 In:
(gdb) info registers
eax          0x6      6
ecx          0x0      0
edx          0x80     128
ebx          0x8800   34816
esp          0x7c00   0x7c00
ebp          0x0      0
esi          0x0      0
edi          0x0      0
eip          0x7e00   0x7e00
eflags      0x286    [ PF SF IF ]
cs           0x0      0
ss           0x0      0
ds           0x0      0
es           0x0      0
fs           0x0      0
gs           0x0      0
(gdb) c
Continuing.
Program received signal SIGINT, Interrupt.
(gdb) x/5x 0x8800
0x8800: 0x0000000000000000 0x00cf930000000000
0x8810: 0x0040970000000000 0x0040930b00007fff
0x8820: 0x00cf980000000000
(gdb)
```

最终qemu和gdb显示



```
run bootloaderon 1.10.2-1ubuntu1
enter protect mode
IPXE (http://ipxe.org) 00:03:0 C900 PC12.10 PaP FPM-07F0BDD0-07ECDD0 C900
Booting from Hard Disk...

remote Thread 1 In:
(gdb) x/5x 0x8800
0x8800: 0x0000000000000000 0x00cf930000000000
0x8810: 0x0040970000000000 0x0040930b00007fff
0x8820: 0x00cf980000000000
(gdb)
```

Assignment3:

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。

Assignment3 相比Assignment2，主要区别在于

1. 替换输出字符的部分
2. 16位寄存器改为32位
3. 中断需要进行改变

3. 关键代码

Assignment1:

1.1

```
bootloader.asm
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 my bootloader'
bootloader_tag_end:
```

```
mbr.asm
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

```

1.2

```

;RATSB00T
;TAB=4

;定义常量
DISC_ADDR      EQU 0x8000      ;磁盘第一个扇区开始，加载到内存缓冲的地址
SECTOR_NUM     EQU 18          ;读取扇区数
CYLINDER_NUM   EQU 10          ;读取柱面数

        ORG 0x7c00      ;指明程序的偏移的基地址

        JMP      Entry

        DB      0x90          ;nop,0x02
        DB      "RATSB00T"    ;（8字节）启动区的名称可以是任意的字符串
        DW      512          ;每个扇区（sector）的大小（必须为512 字节）
        DB      8            ;簇（cluster）的大小（每个簇为8个扇区）
        DW      584          ;保留扇区数,包括启动扇区
        DB      2            ;FAT的个数（必须为2）
        DW      0            ;最大根目录条目个数
        DW      0            ;总扇区数（如果是0，就使用偏移0x20处的4字节值）
        DB      0x00f8       ;磁盘介质描述
        DW      0            ;（FAT16）每个文件分配表的扇区
        DW      63           ;每个磁道扇区数
        dw      255          ;磁头数
        dd      63           ;隐藏扇区
        dd      3902913      ;磁盘大小，总共扇区数（如果超过65535，参见偏移
0x13）
        dd      3804          ;每个文件分配表的扇区，3804个扇区

        dw      0            ;Flagss
        dw      0            ;版本号
        dd      2            ;根目录起始簇

        dw      1            ;FSInfo扇区
        dw      6            ;启动扇区备份
        times 12 db 0        ;保留未使用

        DW      0            ;操作系统自引导代码
        db      0x80         ;BIOS设备代号
        db      0            ;未使用
        db      0x29         ;标记
        DD      0xffffffff   ;序列号
        DB      "HELLO-OS  " ;（11字节）磁盘名称，卷标。字符串长度固定
        DB      "FAT32  "    ;（8字节）FAT文件系统类型。 0x52

        times 12 db 0

;程序核心内容

```


Entry:

```
MOV AX,0 ;初始化寄存器
MOV SS,AX
MOV SP,0x7c00
MOV DS,AX
```

```
MOV DI,StartMessage ;将Message1段的地址放入SI
CALL DisplayStr ;调用函数
MOV DI,BootMessage ;将Message1段的地址放入SI
ADD DH,1
CALL DisplayStr ;调用函数
```

;读取磁盘初始化

```
MOV AX,DISC_ADDR/0x10 ;设置磁盘读取的缓冲区基本地址为ES=0x820。
[ES:BX]=ES*0x10+BX
```

```
MOV ES,AX ;BIOS中断参数: ES:BX=缓冲区的地址
```

```
MOV CH,0 ;设置柱面为0
MOV DH,0 ;设置磁头为0
MOV CL,1 ;设置扇区为2
```

ReadSectorLoop:

```
CALL ReadDisk0; ;读取一个扇区
```

;准备下一个扇区

ReadNextSector:

```
MOV AX,ES
ADD AX,0x0020
MOV ES,AX ;内存单元基址后移0x20(512字节)。[ES+0x20:]
ADD CL,1 ;读取扇区数递增+1
CMP CL,SECTOR_NUM ;判断是否读取到18扇区
JBE ReadSectorLoop ;上面cmp判断(<=)结果为true则跳转到DisplayError
```

;读取另一面磁头。循环读取柱面

```
MOV CL,1 ;设置柱面为0
ADD DH,1 ;设置磁头递增+1:读取下一个磁头
CMP DH,2 ;判断磁头是否读取完毕
JB ReadSectorLoop ;上面cmp判断(<)结果为true则跳转到DisplayError
```

```
MOV DH,0 ;设置磁头为0
ADD CH,1 ;设置柱面递增+1;读取下一柱面
CMP CH,CYLINDER_NUM ;判断是否已经读取10个柱面
JB ReadSectorLoop ;上面cmp判断(<)结果为true则跳转到DisplayError
```

;LoadSuccess:

```
MOV DI,Succmsg
MOV DH,3
CALL DisplayStr
```

;加载执行boot文件:

```
;MOV [0x0ff0],CH ;将总共读取的柱面数存储在内存单元中
;JMP 0xc200 ;跳转执行在内存单元0xc200的代码
```

GoLoader:

```
MOV [0x0ff0],CH ;将总共读取的柱面数存储在内存单元中
```

```

        JMP 0xc200                ;跳转执行在内存单元0xc200的代码:DISC_ADDR-
0x200+0x4200                      ;(启动扇区开始地址0x8000+软盘代码:boot文件开始
0x4200)

LoadError:
        MOV DI,Errormsg
        MOV DH,3
        CALL DisplayStr ;如果加载失败显示加载错误

;程序挂起
Fin:
        HLT                      ;让CPU挂起,等待指令。
        JMP Fin

; -----
; 读取一个扇区函数:ReadDisk0
; -----
; 参数:ES:BS 缓冲区地址,CH柱面,DH磁头,CL扇区,AL扇区数=1,DL驱动器=0x
; -----
ReadDisk0:

        MOV SI,0                ;初始化读取失败次数,用于循环计数

;为了防止读取错误,循环读取5次
;调用BIOS读取一个扇区
ReadFiveLoop:

        MOV AH,0x02             ;BIOS中断参数:读扇区
        MOV AL,1                ;BIOS中断参数:读取扇区数
        MOV BX,0
        MOV DL,0x00             ;BIOS中断参数:设置读取驱动器为软盘
        INT 0x13                ;调用BIOS中断操作磁盘:读取扇区
        JNC ReadEnd             ;条件跳转,操作成功进位标志=0。则跳转执行
ReadNextSector

        ADD SI,1                ;循环读取次数递增+1
        CMP SI,5                ;判断是否已经读取超过5次
        JAE LoadError           ;上面cmp判断(>=)结果为true则跳转到DisplayError

        MOV AH,0x00             ;BIOS中断参数:磁盘系统复位
        MOV DL,0x00             ;BIOS中断参数:设置读取驱动器为软盘
        INT 0x13                ;调用BIOS中断操作磁盘:磁盘系统复位
        JMP ReadFiveLoop

;扇区读取完成
ReadEnd:
        RET

; -----
; 显示字符串函数:DisplayStr
; -----
; 参数:SI:字符串开始地址,DH为第N行
; -----
DisplayStr:
        MOV CX,0                ;BIOS中断参数:显示字符串长度
        MOV BX,DI
        .1:;获取字符串长度

```

```

MOV AL,[BX]          ;读取1个字节。这里必须为AL
ADD BX,1             ;读取下个字节
CMP AL,0             ;是否以0结束
JE .2
ADD CX,1             ;计数器
JMP .1
.2:;显示字符串
MOV BX,DI
MOV BP,BX
MOV AX,DS
MOV ES,AX            ;BIOS中断参数: 计算[ES:BP]为显示字符串开始地址

MOV AH,0x13          ;BIOS中断参数: 显示文字串
MOV AL,0x01          ;BIOS中断参数: 文本输出方式(40x25 16色 文本)
MOV BH,0x0           ;BIOS中断参数: 指定分页为0
MOV BL,0x0c          ;BIOS中断参数: 指定白色字体07
MOV DL,0             ;列号为0
INT 0x10             ;调用BIOS中断操作显卡。输出字符
RET

;数据初始化
StartMessage: DB "hello,Adria's CHS call",0
BootMessage:  DB "booting.....",0
Errormsg:     DB "Fail",0
Succmsg:      DB "Success",0

FillSector:
RESB 510-($-$$)      ;处理当前行$至结束(1FE)的填充
DB 0x55, 0xaa

```

Assignment2:

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。
5. 调试

boot.inc

```

; 常量定义区
; _____Loader_____
; 加载器扇区数
LOADER_SECTOR_COUNT equ 5
; 加载器起始扇区
LOADER_START_SECTOR equ 1
; 加载器被加载地址
LOADER_START_ADDRESS equ 0x7e00
; _____GDT_____
; GDT起始位置
GDT_START_ADDRESS equ 0x8800
; _____selector_____
;平坦模式数据段选择子
DATA_SELECTOR equ 0x8
;平坦模式栈段选择子

```

```

STACK_SELECTOR equ 0x10
;平坦模式视频段选择子
VIDEO_SELECTOR equ 0x18
VIDEO_NUM equ 0x18
;平坦模式代码段选择子
CODE_SELECTOR equ 0x20

```

bootloader.asm

```

%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

;空描述符
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

;初始化描述符表寄存器GDTR
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

```

```

;以下进入保护模式
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

pgdt dw 0
    dd GDT_START_ADDRESS

bootloader_tag db 'run bootloader'
bootloader_tag_end:

protect_mode_tag db 'enter protect mode'
protect_mode_tag_end:

```

mbr.asm:

```

#include "boot.inc"
[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, LOADER_START_SECTOR
mov cx, LOADER_SECTOR_COUNT
mov bx, LOADER_START_ADDRESS

load_bootloader:
    push ax
    push bx
    call asm_read_hard_disk ; 读取硬盘
    add sp, 4
    inc ax
    add bx, 512
    loop load_bootloader

    jmp 0x0000:0x7e00 ; 跳转到bootloader

```

```

jmp $ ; 死循环

```

```

; asm_read_hard_disk(memory,block)
; 加载逻辑扇区号为block的扇区到内存地址memory

```

```

asm_read_hard_disk:
    push bp
    mov bp, sp

    push ax
    push bx
    push cx
    push dx

    mov ax, [bp + 2 * 3] ; 逻辑扇区低16位

    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

    xor ax, ax
    inc dx ; 0x1f5
    out dx, al ; LBA地址23~16 = 0

    inc dx ; 0x1f6
    mov al, ah
    and al, 0x0f
    or al, 0xe0 ; LBA地址27~24 = 0
    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 bx, [bp + 2 * 2]
    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

    pop dx
    pop cx
    pop bx
    pop ax
    pop bp

    ret

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

```

Assignment3:

实模式下的代码:

```

#include "boot.inc"
;org 0x7e00
[bits 16]
mov ax, 0xb800
mov gs, ax
xor ebx, ebx

;空描述符
mov dword [GDT_START_ADDRESS+0x00], 0x00 ;0x7e24
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
```

```
;初始化描述符表寄存器GDTR
```

```
mov word [pgdt], 39 ;描述符表的界限 0x7e7e
```

```
lgdt [pgdt] ;0x7e84
```

```
in al,0x92 ;南桥芯片内的端口 0x7e89
```

```
or al,0000_0010B ;0x7e8b
```

```
out 0x92,al ;打开A20 0x7e8d
```

```
cli ;中断机制尚未工作 0x7e8f
```

```
mov eax,cr0 ;0x7e90
```

```
or eax,1 ;0x7e93
```

```
mov cr0,eax ;设置PE位 0x7e97
```

```
;以下进入保护模式
```

```
jmp dword CODE_SELECTOR:protect_mode_begin ;0x7e9a
```

```
;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
```

```
;以下为字符弹射程序的代码
```

```
;把背景变成全黑
```

```
pushad
```

```
mov bx,0
```

```
mov cx,4000 ;4000=2*25*80
```

```
loop0:
```

```
    cmp bx,cx
```

```
    jz loop0end
```

```
    mov ah,0x00 ;黑色
```

```
    mov al,'0'
```

```
    mov [gs:bx],ax
```

```
    add bx,2
```

```
    jmp loop0
```

```
loop0end:
```

```
popad
```

```
;起始: 设置光标位置为(2,0)
```

```
    mov ebx,160
```

```
loop1: ;循环输出
```

```
    mov al,[num]
```

```
    mov ah,[color]
```

```
    add al,'0'
```

```
    mov [gs:ebx],ax
```

```
;输出完后判断下一步是否要改方向
```

```
pushad
```



```

judge_dh_0:
    mov ax,160
    cmp bx,ax
    jg judge_dh_24 ; 不在第0行
    mov al,1
    mov [down],al ;若行数为0则往下走
judge_dh_24:
    mov ax,3840 ;3840 = 24*160
    cmp bx,ax
    jl judge_dl_0 ; bx<24*160,不在第24行
    mov al,0
    mov [down],al ;若行数为24则往上走
judge_dl_0:
    mov ax,bx
    mov cl,160
    div cl ;bx/160,余数在ah中
    mov dx,0
    cmp ah,dh
    jne judge_dl_79 ; 不在第0列
    mov al,1
    mov [right],al ;若列为0则往右走
judge_dl_79:
    mov dx,158 ;158=2*79
    cmp ah,d1
    jne judge_end ; 不在第79列
    mov al,0
    mov [right],al ;若列为79则往左走
judge_end:
popad

```

;设置下一步的坐标

```

push ax
push cx
if_right:
    mov al,1
    mov cl,[right]
    cmp cl,al
    jne else1 ;right=0,跳去else1
    add bx,2
    jmp if_down
else1:
    sub bx,2
if_down:
    mov al,1
    mov cl,[down]
    cmp cl,al
    jne else2 ;down=0,跳去else2
    add bx,160
    jmp set_xy_end
else2:
    sub bx,160
set_xy_end:
pop cx
pop ax

```

;改下一步的数字和颜色

```

pushad
    mov al,[num]

```

```

    inc al
    mov bl,10
    cmp al,bl
    jne not_need_set_0 ;num!=10,不需要置零
    mov al,0
not_need_set_0:
    mov [num],al

    mov al,[color]
    inc al
    mov bl,255
    cmp al,bl
    jne not_need_set_1 ;num!=255,不需要置零
    mov al,0
not_need_set_1:
    mov [color],al
popad

;每显示一个数字延迟一段时间
pushad
    mov ecx,1000000
for_wait:
    mov eax,1
    and eax,1
    loop for_wait
popad

jmp loop1
myinfo db '                                zwq20337268 '
    infolen dw $-myinfo
    curcolor db 80h
    curcolor2 db 09h
    times 510-($-$$) db 0
    db 55h, 0AAh
;,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
jmp $ ; 死循环

pgdt dw 0
    dd GDT_START_ADDRESS

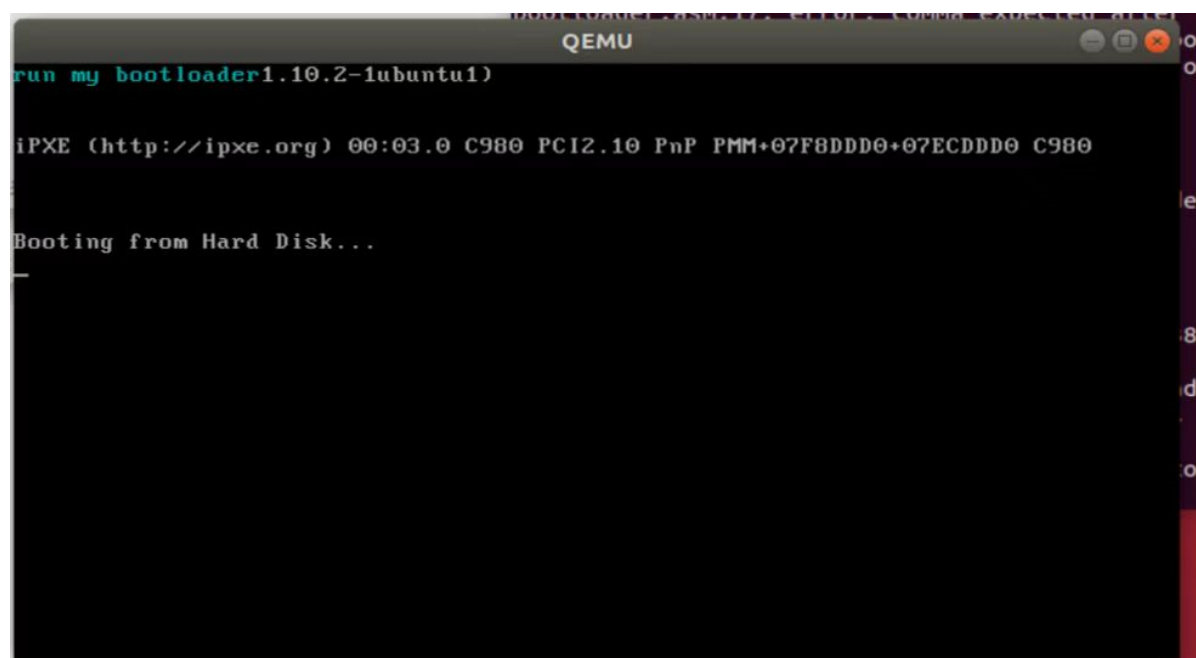
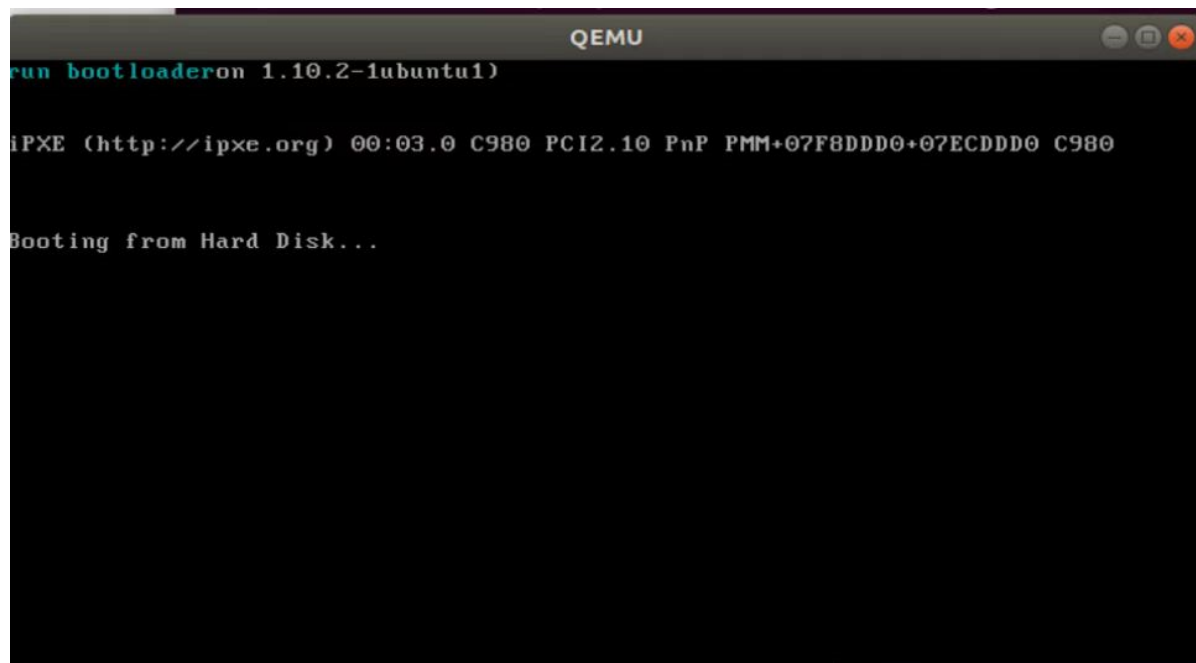
bootloader_tag db 'run bootloader'
bootloader_tag_end:
protect_mode_tag db 'enter protect mode'
protect_mode_tag_end:
num db 0 ;显示的数字和颜色
color db 8
right db 1 ;标识方向
down db 1

```

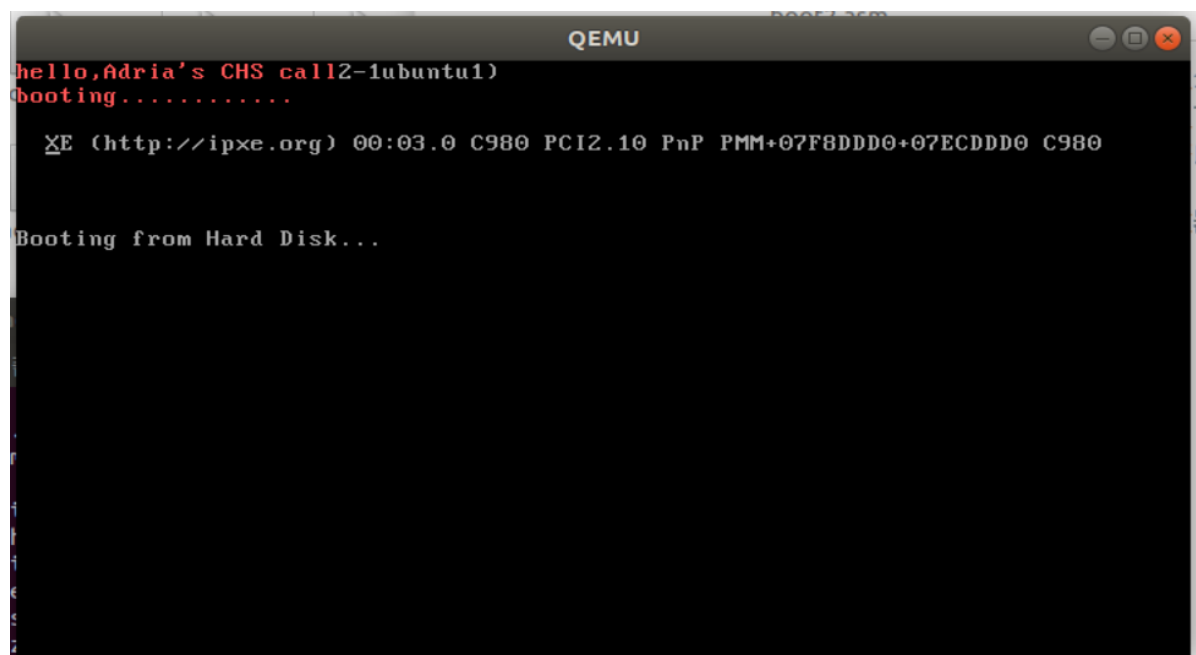
4. 实验结果

Assignment1:

1.1



1.2



Assignment2: GDT的五段描述符如下所示

;空描述符

```
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0xFFFF  
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

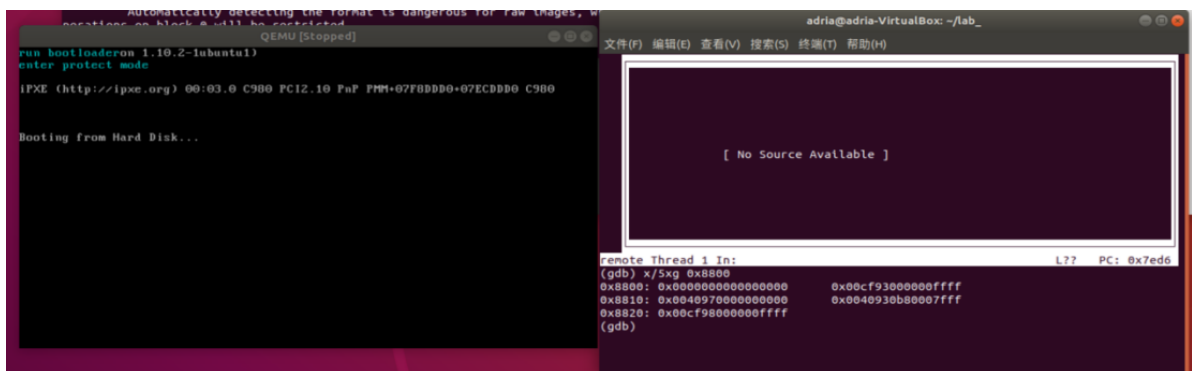
```
mov dword [GDT_START_ADDRESS+0x10],0x00000000 ; 基地址为0x00000000,界限0x0  
mov dword [GDT_START_ADDRESS+0x14],0x00409600 ; 粒度为1个字节
```

;建立保护模式下的显存描述符

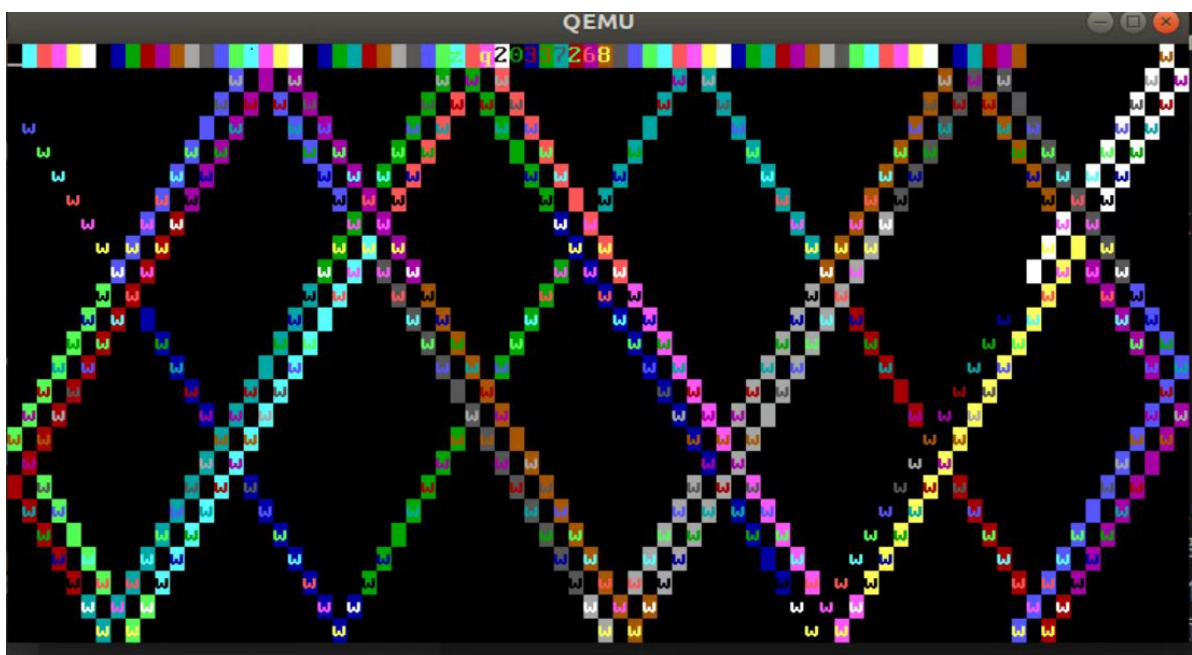
```
mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x000B8000,界限0x07FFF  
mov dword [GDT_START_ADDRESS+0x1c],0x0040920b ; 粒度为字节
```

;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0xFFFF  
mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb,代码段描述符
```



Assignment3:



5. 总结

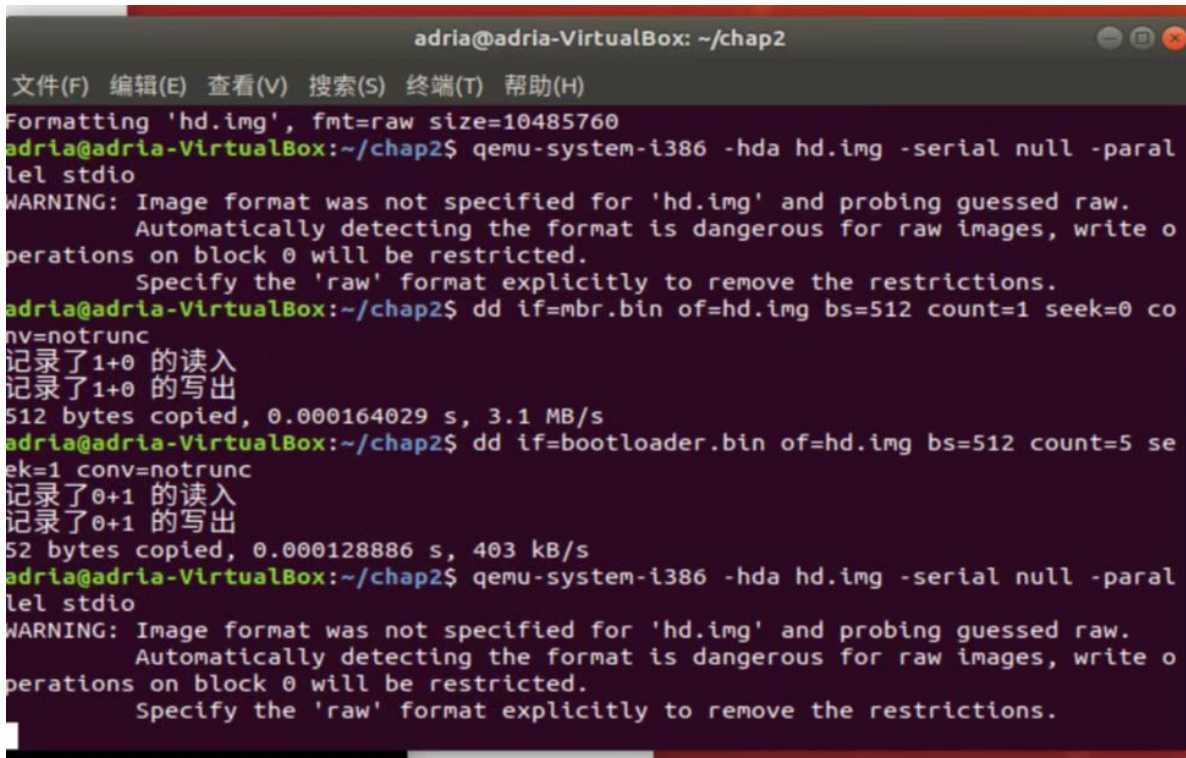
Assignment1:

1.1

1. 如果将字符串改为“Run Adria's bootloader” 会显示 "too long", 拓展读入字节即可

```
adria@adria-VirtualBox:~/chap2$ nasm -f bin mbr.asm -o mbr.bin
adria@adria-VirtualBox:~/chap2$ nasm -f bin bootloader.asm -o bootloader.bin
bootloader.asm:17: warning: unterminated string
bootloader.asm:17: warning: character constant too long
bootloader.asm:17: error: comma expected after operand 1
```

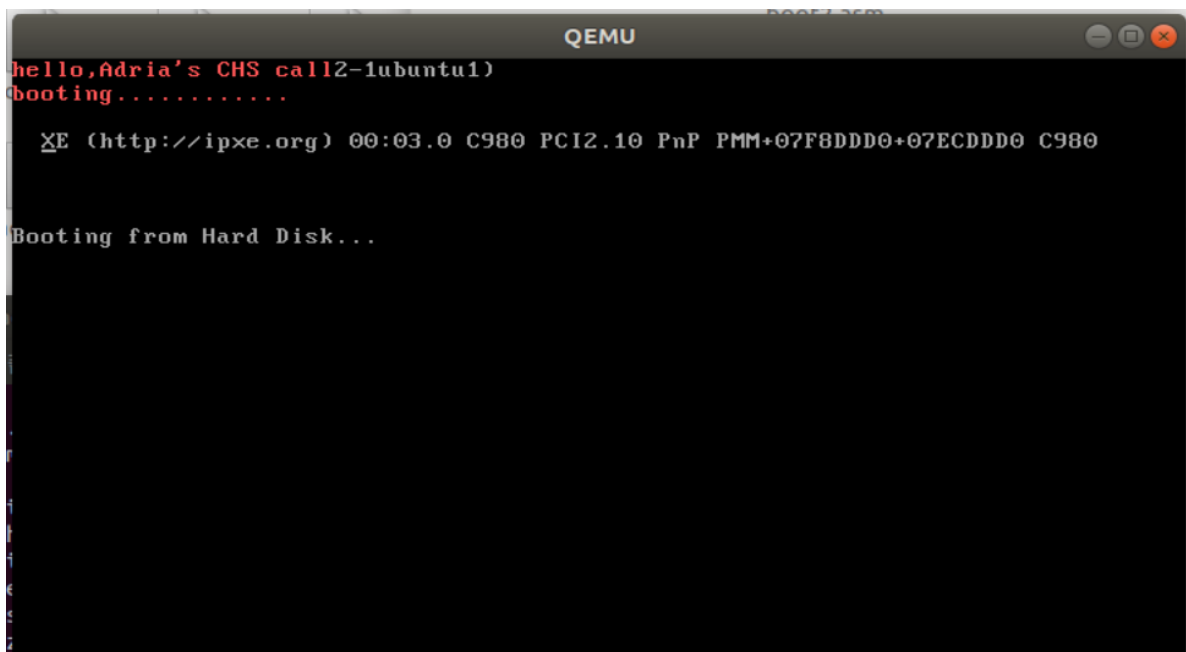
2. 在终端执行命令的时候没有注意先后顺序导致报错



```
adria@adria-VirtualBox: ~/chap2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Formatting 'hd.img', fmt=raw size=10485760
adria@adria-VirtualBox:~/chap2$ 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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
adria@adria-VirtualBox:~/chap2$ dd if=mbr.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, 0.000164029 s, 3.1 MB/s
adria@adria-VirtualBox:~/chap2$ 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
512 bytes copied, 0.000128886 s, 403 kB/s
adria@adria-VirtualBox:~/chap2$ 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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

1.2

1. 代码写完无法正常运行, 无输出: 重启之后正常运行, 猜测可能是运行一段时间后寄存器初始值不符合本代码要求



```
QEMU
hello,Adria's CHS call2-1ubuntu1)
booting.....

XE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

Booting from Hard Disk...
```

Assignment2:

1. 正确生成了符号表, 但是无法展示src, 如下所示 (No Source Available), 网络可供参考资料太少暂未找到解决方法

```
活动 终端
adria@adria-VirtualBox: ~/lab_

[ No Source Available ]

remote Thread 1 In:
(gdb) info registers
eax      0x0
ecx      0x0
edx      0x80  128
ebx      0x5800  23040
esp      0x7c00  31744
ebp      0x0
esi      0x0
edi      0x0
eip      0x7e00  32256
eflags   0x286  [ PF SF IF ]
cs       0x0
ss       0x0
ds       0x0
fs       0x0
gs       0x0
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
demonofeds to ?? (?)
(gdb) x/5gx 0x8000
0x8000: 0x0000000000000000  0x00cf93000000ffff
0x8010: 0x0040970000000000  0x004097000000ffff
0x8020: 0x00cf93000000ffff
(gdb)
```

最终得到GDT的五个段描述