中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验三

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：杨培凯

学生学号：23336279

实验地点：实验楼 B203

报告时间：2025 年 4 月 7 日

Section1 实验概述

- **实验任务一**：使用MBR加载bootloader到内存中，并运行bootloader的代码
- **实验任务二**：在bootloader中进入保护模式，并结合gdb调试来分析进入保护模式的四个重要步骤
- **实验任务三**：在保护模式中运行字符回旋程序

Section 2 实验步骤与实验结果

实验任务一

- **任务要求**
  - 在MBR中加载bootloader到内存中，并跳转到bootloader起始地址开始执行，在qemu显示屏输出assignment1.1 : run bootloader
  - 将LBA28读取硬盘的方式换成CHS读取,并使用int 13h中断把bootloader读入到内存中，并同样在qemu显示屏输出assignment1.2 : run bootloader
- **思路分析**
  - **任务1.1**
    - 在MBR中使用循环来调用五次asm_read_hard_disk函数把占据了五个扇区的bootlaoder读入到起始地址为0x7e00对应的空间中，每次读入一个扇区，然后跳转到bootloader在内存中的起始地址
    - 其中，asm_read_hard_disk函数采用的是LBA28硬盘读取方式，先把读命令写入到地址0x1f3~0x1f7对应的空间中，然后等待硬盘空闲，通过循环读入一个扇区的512个字节到内存中
    - 然后，从0x7e00开始运行bootloader,qemu显示屏上输出assignment1.1 : run bootloader
  - **任务1.2**
    - 相较于任务1.1，任务1.2主要是把硬盘读取方式变成CHS模式，其余操作同上。
    - **LBA28->CHS**：C、H、S分别代表柱面号、磁头号、扇区号， 对应的空间大小关系是：柱面 > 磁头 > 扇区。因此当扇区数量等于每磁头扇区数时，这些扇区就能组成一个磁头；当磁头数量等于每柱面磁头数时，这些磁头就能组成一个柱面。而LBA28硬盘读取方式恰好给出的就是逻辑扇区号(通俗理解就是把柱面和磁头拆成扇区大小，按顺序排成线性队列，然后某个扇区在队列中的序列号就是逻辑扇区号)，因此我们通过把扇区号进行除法和取余就能得到CHS模式下对应的柱面号、磁头号、扇区号。
    - CHS硬盘读取模式需要使用int 13h中断，而且能够一次性读入指定数量的扇区，不用把读命令写入指定地址，也不需要多次循环，代码编写简单。

```
; 使用 int 13h 磁盘读取中断，来一次性把逻辑扇区 1 ～ 4 放入到0x0000 :
0x7e00
mov ah, 02h             ; 功能号
mov al, 4               ; 一次性读入的扇区数
mov ch, 0               ; 柱面号
```

```
mov cl, 2               ; 扇区号
mov dh, 0               ; 磁头号
mov dl, 80h             ; 硬盘号
int 13h                 ; 中断
```

- **实验步骤**
  - 编写mbr代码并编译放入0号扇区

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

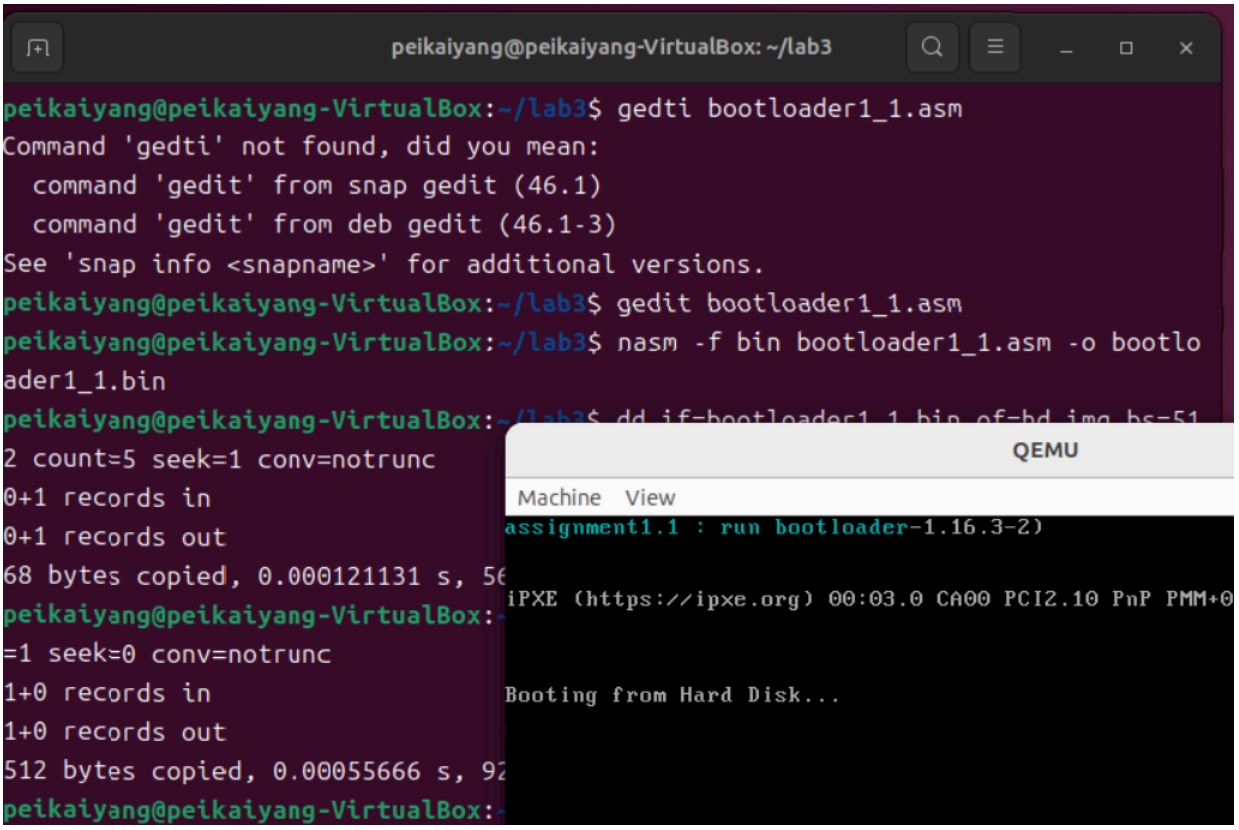
  - 编写bootloader代码编译并放入1号扇区

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

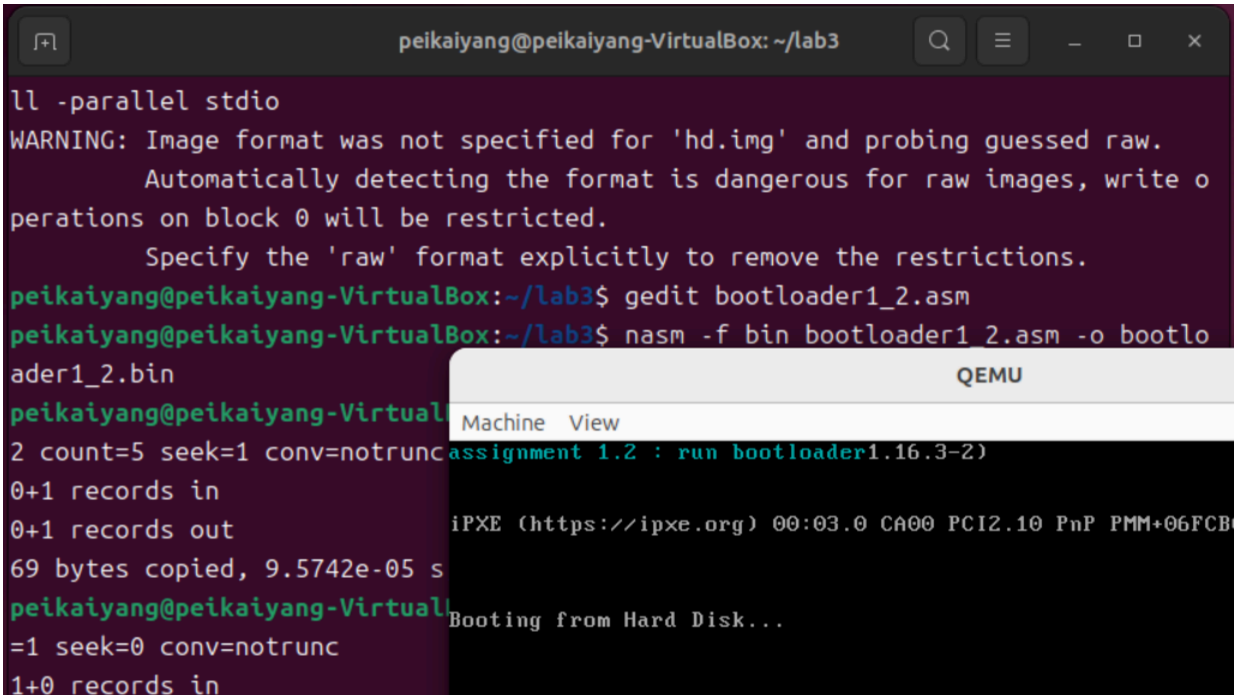
  - 运行qemu

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

- **实验结果**
  - 任务1.1



- 任务1.2



实验任务二

- **任务要求**
  - 使用gdb调试来分析在进入保护模式过程中 准备GDT、打开第二十一条地址线、开启cr0保护模式标志位、远跳转进入保护模式四个步骤寄存器内容的变化情况
- **思路分析**
  - **准备GDT**

        - GDT的地址为0x8800，只需要查看 GDTR初始化 前后的GDT中的内容的变化情况即可
    - **打开第二十一条地址线**
        - 由于in al, 0x92，因此第二十一条地址线是否被打开，只需查看or al, 0000_0010B命令前后的eax内容的变化情况即可
    - **开启cr0保护模式标志位**
        - 由于mov eax, cr0，因此PE位是否被设置为1，只需要查看or eax, 1命令前后的eax内容的变化情况即可(但实际上，在使用info registers命令时，也可以直接查看cr0的值，因此直接查看cr0值的变化是更直接的做法)
    - **远跳转进入保护模式**
        - 在bootloader中是通过jmp dword CODE_SELECTOR:protect_mode_begin来进入保护模式的，因此，我们只需查看在这行代码前后的段寄存器cs、ds、es、fs、gs以及栈指针eip内容的变化情况即可
- **实验步骤**
    - 在assignment文件夹中使用make build编译相关文件，然后输入make debug做好debug环境准备
    - 在gdb中设置断点break *0x7e00，并输入continue运行到该断点
    - 然后，不断输入stepi，逐行运行bootloader的汇编代码
    - 在准备GDT、打开第二十一条地址线、开启cr0保护模式标志位、远跳转进入保护模式四个步骤运行前后，使用info registers来查看相应寄存器的内容
    - 最后，使用coninue运行剩下代码，在qemu显示屏输出enter protect mode
- **实验结果**
    - **准备GDT**
        - GDTR初始化前

```
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000      0x0000000000000000
0x8810: 0x0000000000000000      0x0000000000000000
0x8820: 0x0000000000000000
```

- GDTR初始化后

```
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000      0x00cf93000000ffff
0x8810: 0x0040970000000000      0x0040930b80007fff
0x8820: 0x00cf98000000ffff
```

- **打开第二十一条地址线**
  - **打开前**

```
eax            0xb802           47106
```

  - **打开后**

```
eax            0xb802           47106
```

- **开启cr0保护模式标志位**
  - **开启前**

```
cr0            0x10             [ ET ]
eax            0x10             16
```

  - **开启后**

```
cr0            0x11             [ ET PE ]
eax            0x11             17
```

- **远跳转进入保护模式**
  - **跳转前**

```
cs             0x0             0
ss             0x0             0
ds             0x0             0
es             0x0             0
gs             0xb800          47104
eip            0x7eae          0x7eae
```

- **跳转后**

```
cs             0x20            32
ss             0x10            16
ds             0x8             8
es             0x8             8
gs             0x18            24
eip            0x7eb6          0x7eb6 <protect_mode_begin>
```

实验任务三

- **任务要求**
  - 改写`lab2 assignment4`的字符回旋程序，使其能在保护模式下运行
- **思路分析**
  - 字符回旋程序在实模式和保护模式中的最大区别就是，在始陌始终可以使用中断来控制光标位置以及字符输出，但是在保护模式中，不能使用中断命令，因此就需要**直接操作显存**
  - 按照`assignment2`的方法进入保护模式
  - 先清空显示屏(这里使用的方法是全屏输出空字符' ')

```
;清空屏幕
Clear_screen:pusha
mov ecx, 80
mov ebx, 25
imul ecx, ebx
mov ebx, 0
mov ah, 0x07
mov al, ' '
Clear_loop:    mov [gs:ebx], ax    ;直接操作显存
    add ebx, 2
```

```
    loop Clear_loop
popa
ret
```

- 然后，在显示屏中央输出个人信息ypk 23336279。由于不需要使用中断一个字符一个字符的输出，我们可以按照输出字符串的方式来输出

```
; 计算光标输出的初始位置，即显存的(11,34)位置处
mov ebx, 12
mov ecx, 80
imul ebx, ecx
add ebx, 35
mov ecx, 2
imul ebx, ecx

; 在指定光标位置开始输出'ypk 23336279'
mov ecx, private_information_end - private_information
mov esi, private_information
mov ah, 0x70
output_private_information:    mov al, [esi]
    mov word [gs:ebx], ax  ; 直接操作显存
    add ebx, 2
    inc esi
    loop output_private_information
```

```
private_information db 'ypk 23336279'
private_information_end:
```

- 字符回旋程序的主要思路还是和lab2 assignment4一致，只是把其中中断实现的部分换成了对显存的直接操作(用ebx来表示偏移地址，gs表示显存段选择子)

```
  Loop_above -> Loop_right
      ↑             ↓
  Loop_left  <- Loop_below
```

  - **自左向右**

    ebx从0开始，每次增加2，自左向右遍历显示屏上边缘

```
;在屏幕上边缘 自左向右 输出字符
Loop_above:     cmp ebx, 158
    je Loop_right
```

```
    ;输出 字符
    mov word [gs:ebx], ax ;直接操作显存

    ;修改 字符内容
    call Change_char

    ;把光标向右移动1位
    add ebx, 2

    ;控制背景色和前景色的改变
    inc ah

    ;控制字符输出速度
    call Slow_down

    jmp Loop_above
```

- **自上到下**

  ebx从158开始，每次增加160，自上向下遍历显示屏右边缘

```
;在屏幕右边缘  自上向下  输出字符
Loop_right:     cmp ebx, 3998
    je Loop_below

    ;输出 字符
    mov word [gs:ebx], ax ;直接操作显存

    ;修改 字符内容
    call Change_char

    ;把光标向下移动1位
    add ebx, 160

    ;控制背景色和前景色的改变
    inc ah

    ;控制字符输出速度
    call Slow_down

    jmp Loop_right
```

- **自右到左**

  ebx从3998开始，每次减小2，自右向左遍历显示屏下边缘

```
;在屏幕下边缘  自右向左  输出字符
Loop_below:     cmp ebx, 3840
    je Loop_left
```

```
    ;输出 字符
    mov word [gs:ebx], ax ;直接操作显存

    ;修改 字符内容
    call Change_char

    ;把光标向左移动1位
    sub ebx, 2

        ;控制背景色和前景色的改变
    inc ah

    ;控制字符输出速度
    call Slow_down

    jmp Loop_below
```

- **自下到上**

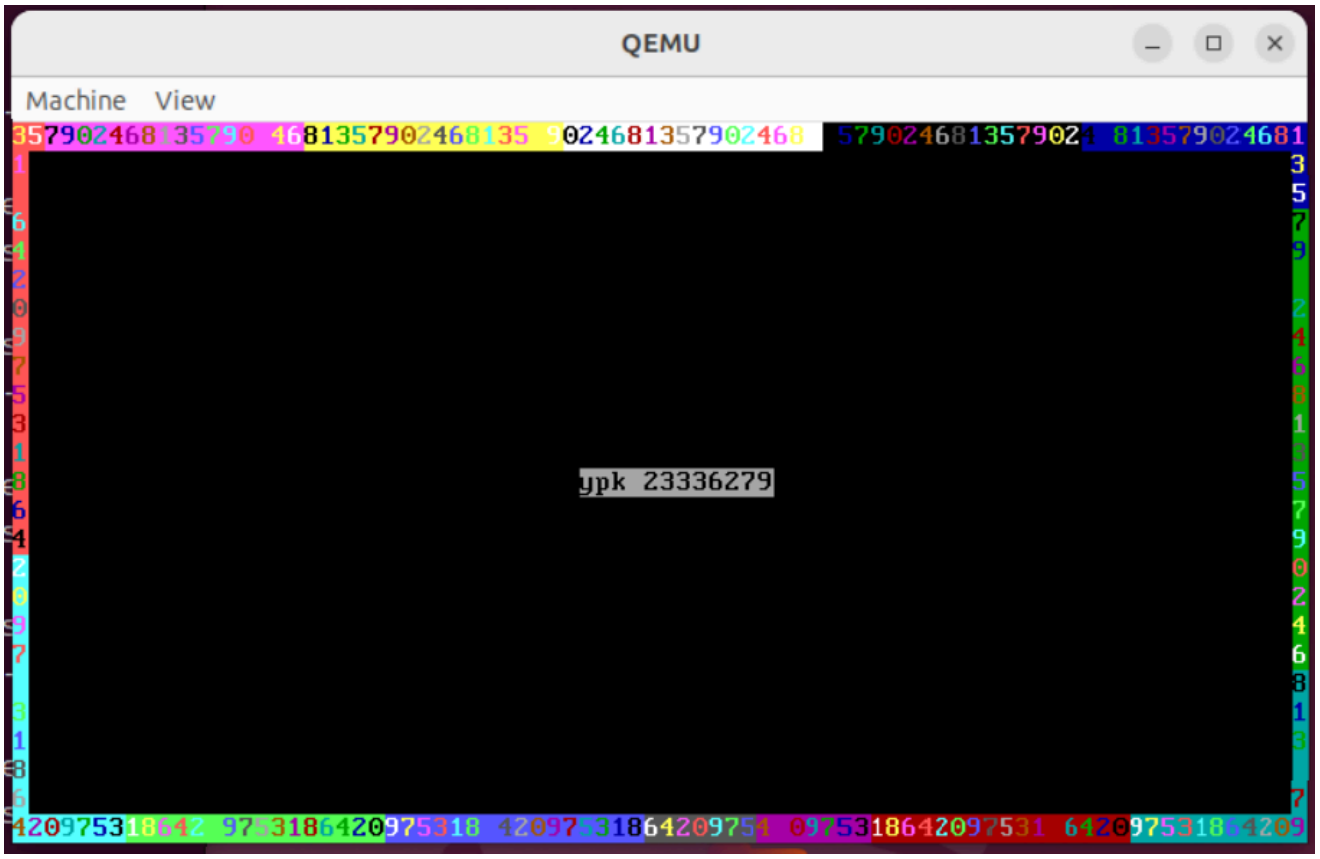
  ebx从3840开始，每次减小160，自下向上遍历显示屏左边缘

```
;在屏幕左边缘  自下向上 输出字符
Loop_left:      cmp ebx, 0
    je Loop_above

    ;输出 字符
    mov word [gs:ebx], ax ;直接操作显存

    ;修改 字符内容
    call Change_char

    ;把光标向上移动1位
    sub ebx, 160

    ;控制背景色和前景色的改变
    inc ah

    ;控制字符输出速度
    call Slow_down

    jmp Loop_left
```

- **实验步骤**
  - 编写mbr文件(这里直接使用assignment1.1的mbr文件)
  - 编写bootloader文件(把lab2 assignment4和lab3 assignment2的代码整合到一起，重写有关中断的部分即可)

- 编译并放入虚拟硬盘，启动qemu显示结果

- **实验结果**



Section 3 实验总结与心得体会

- 前面两个任务在实验文档上都有详细指导，没有很大的复现难度

- 在任务三中，其实bootloader.asm文件没有问题，问题恰恰是出在mbr.asm。我一开始使用的是lab3 assignment1.2的mbr文件，因为感觉使用CHS来读取硬盘的方式更简便。但是在打开qemu后，显示屏只是在快速闪烁，并没有输出程序中的内容。我以为是bootloader文件有问题，尝试改了很多地方，都没有任何改进。机缘巧合之下，我把mbr文件换成lab3 assignment1.1的mbr文件，然后就直接运行成功了。我觉得之所以是这样有些滑稽的解决方式，可能是因为lab3 assignment1.2的mbr文件还是有地方没考虑周全的，但是在lab3 assignment1.2的情况下，能够碰巧运行，就导致了在任务3的失败，所以lab3 assignment1.2的mbr文件，还需要去仔细检查，修改漏洞。

Section 4 参考资料清单

[1] 课程实验文档：https://gitee.com/apshuang/sysu-2025-spring-operating-system/tree/master/lab3

[2] deepseek提供了一些nasm的语法支持，以及assignment3的思路(直接操作显存)也主要来源于deepseek