1 实验概述

在本次实验中，掌握读写硬盘，以及从实模式进入保护模式的方法，并初识保护模式，具体任务如下:

- 实验任务一: 复现 example 中用 LBA 读写硬盘的方式，并用 CHS 读取硬盘
- 实验任务二: 复现进入保护模式的 example2, 并学会用 gdb 进行调试
- 实验任务三: 在保护模式下实现字符旋转程序

2 实验过程

2.1 实验任务一

2.1.1 使用 **LBA** 读取硬盘

计算机启动后会自动加载 MBR(第 0 个扇区) 到内存中运行, 但 MBR 只局限于 512MB，因此我们需要在 MBR 中实现读取扇区的程序, 将一段程序从外存加载进内存中。即 bootloader, 存放在第 1-5 块扇区中。
读取硬盘时, 需要通过端口来实现:
端口 0x1f3-0x1f6 设置起始的逻辑扇区号

```
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
```

端口 0x1f2 设置读取扇区的数量, 我们采取一个一个扇区读的方式，因此将 0x1f2 设置为 1

```
mov dx, 0x1f2
mov al, 1
out dx, al    ; 读取1个扇区
```

端口 0x1f7 中设置读命令

```
mov dx, 0x1f7     ; 0x1f7
mov al, 0x20      ;读命令
out dx,al
```

请求硬盘读时, 硬盘中可能正在进行其它读写操作, 需要等待完成后才能进行硬盘读, 而硬盘的状态可通过 0x1f7 端口来访问

```
.waits:   ;等待其它操作完成
in al, dx         ; dx = 0x1f7
```

```
and al,0x88
cmp al,0x08
jnz .waits
```

最后进行硬盘读取, 从 0x1f0 端口进行读取, 每次读取两字节:

```
mov cx, 256
mov dx, 0x1f0    ;硬盘接口的数据端口
.readw:
in ax, dx
mov [bx], ax
add bx, 2
loop .readw
```

将上面的单个扇区读取封装为函数, 依次读入五个扇区，最后原跳转至第一个扇区开始处，就可以执行 bootloader 中的命令了

```
mov ax, 1
mov cx, 0
mov bx, 0x7e00
load_bootloader:  ;依次读取五个硬盘
    call asm_read_hard_disk
    inc ax
    cmp ax, 5
    jle load_bootloader
jmp 0x0000:0x7e00
```

在 bootloader 中, 实现在 qemu 显示屏中显示的程序, 随后将它打包写入硬盘起始编号为 1 的扇区，共有 5 个扇区。而 MBR 写入 0 号扇区。

```
nasm -f bin bootloader.asm -o bootloader.bin
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
```

最后启动 qemu 就可以看到结果了。

2.1.2 CHS 方式读取硬盘

CHS 模式是直接使用磁头柱面扇区来对硬盘进行寻址，其中 C、H、S 分别代表柱面、磁头、扇区。而 LBA 模式是直接通过逻辑扇区号来进行寻址。因此，两者可以相互转化，其中特别注意 LBA 的编号从 0 开始，CHS 模式的扇区号从 1 开始。

设 NS 为每磁道扇区数，NH 为磁头数，C、H、S 分别代表柱面号、磁头号、扇区号，LBA 为逻辑扇区号。磁盘寻址按照柱面 → 磁头 → 扇区进行寻址，因此得到 CHS 到 LBA 的转化关系:

$$LBA = NH * NS * C + NS * H + S - 1$$

因此也可以由 LBA 逻辑扇区号反解出 C、H、S:

$$C = (LBA \div NS) \div NH \tag{1}$$

$$H = (LBA \div NS) \mod NH \tag{2}$$

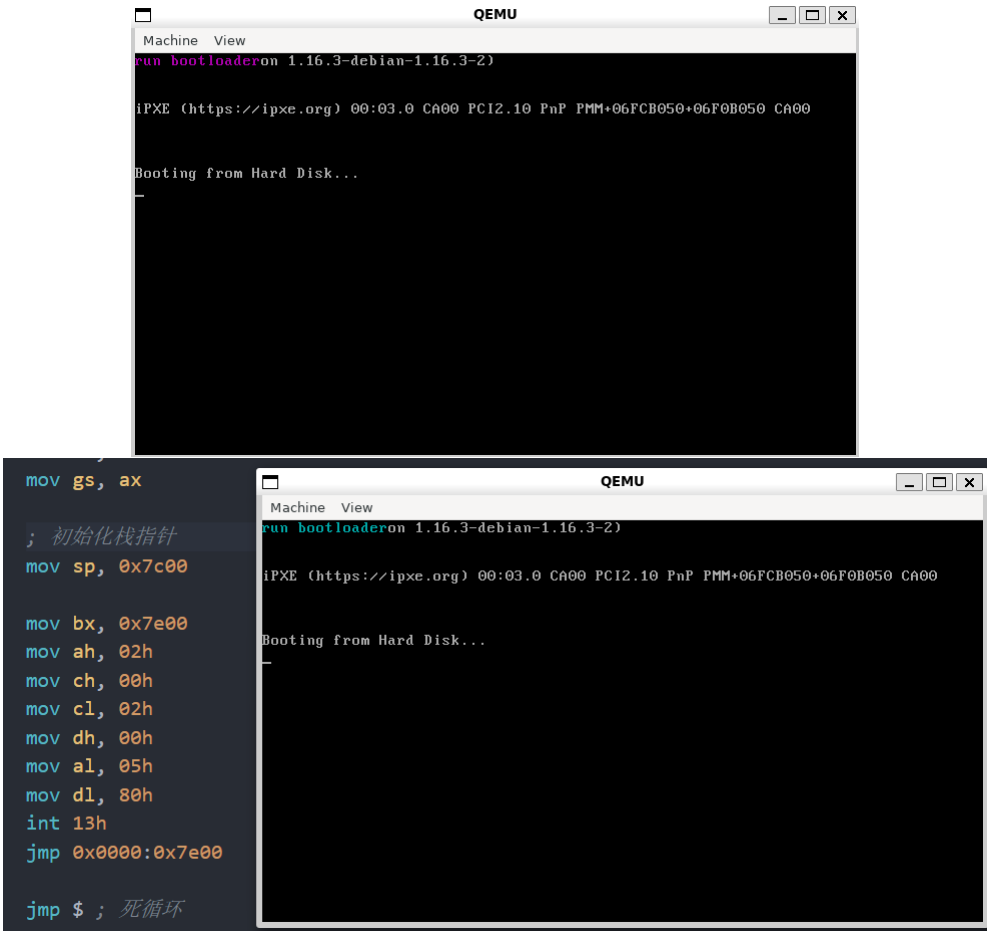
$$S = (LBA \mod NS) + 1 \tag{3}$$

在实现上，应用 BIOS 的 int13h 中断，也称为直接磁盘服务。
其中运用子功能 02H 实现读扇区。在本实验中，需要读取 LBA 编号为 1-5 的扇区，对应 CHS 的起始值为 (C:0 H:0 S:2)
代码如下:

```
mov bx, 0x7e00              ; bootloader的加载地址
mov ah, 02h                 ;功能：读扇区
mov ch, 00h                 ;柱面
mov cl, 02h                 ;扇区   (LBA从0开始,CHS的扇区从1开始)
mov dh, 00h                 ;磁头
mov al, 05h                 ;扇区数
mov dl, 80h                 ;驱动器:80h-0FFh是硬盘
int 13h
jmp 0x0000:0x7e00           ;  跳转到bootloader
```

2.1.3 实验结果展示

下图分别为两种方式读取硬盘的结果，由于在 qemu 上显示并无区别，故用颜色区分:



2.2 实验任务二

2.2.1 进入保护模式

进入保护模式，需要完成以下操作:

- 在 mbr 中实现硬盘的读取，从而跳转至 bootloader 中的程序
- 在 bootloader 中，首先准备 GDT，并将其起始地址和大小保存至寄存器 GDTR 中
- 打开第 21 根地址线
- 开启 cr0 的保护模式标志位
- 远跳转至保护模式

mbr 的实现与实验任务一相同，这里重点展示 bootloader.asm 的实现。
首先逐个准备数据段、栈段、视频段、代码段 GDT:
在此，我们主要分析各 GDT 的以下信息:

- 粒度，以字节为单位还是以 4KB 为单位，从而决定了寻址范围
- S 位，S 位是描述符类型。S=0 表示该段是系统段，S=1 表示该段位代码段或数据段
- 段基地址: 段的起始地址
- 段界限: 段的偏移地址范围

其中 GDT_START_ADDRESS 为 GDT 的起始位置，在内存中处于逻辑硬盘 5 之后 GDT 必须以空描述符开始:

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

接着是数据段 GDT: 粒度为 4KB,S=1, 段基地址为 0x00000000，段界限为 0xfffff

```
;小端编址
mov dword [GDT_START_ADDRESS+0x08],0x0000ffff
mov dword [GDT_START_ADDRESS+0x0c],0x00cf9200
```

接着是堆栈段 GDT: 粒度为 1B,S=1, 基地址为 0x00000000, 段界限为 0x00000

```
    ;小端编址
mov dword [GDT_START_ADDRESS+0x10],0x00000000
mov dword [GDT_START_ADDRESS+0x14],0x00409600
```

接着是视频段 GDT: 粒度为 1B，基地址为 0x000b8000, 段界限为 0x07fff

```
;小端编址
mov dword [GDT_START_ADDRESS+0x18],0x80007fff
mov dword [GDT_START_ADDRESS+0x1c],0x0040920b
```

接着是代码段 GDT: 粒度为 4KB，基地址为 0x00000000, 段界限为 0xfffff

```
;小端编址
mov dword [GDT_START_ADDRESS+0x20],0x0000ffff
mov dword [GDT_START_ADDRESS+0x24],0x00cf9800
```

由于我们创建了五个描述符，因此 GDT 界限:

$$8 * 5 - 1 = 39$$

因此可以初始化寄存器 GDTR

```
pgdt dw 0
dd GDT_START_ADDRESS
;初始化描述符表寄存器GDTR
mov word [pgdt], 39
lgdt [pgdt]
```

接着分别打开第 21 根地址线和和设置 PE 位, 其中保护模式下中断机制尚未建立，应禁止中断

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

cli                                 ;关中断
mov eax,cr0
or eax,1
mov cr0,eax                         ;设置PE位
```

最后远跳转进入保护模式，从而可以开始编写保护模式下的代码

```
jmp dword CODE_SELECTOR:protect_mode_begin

[bits 32]
protect_mode_begin:
```

在保护模式下，寻址方式表示为选择子: 偏移地址，其中选择子的前 12 位为段描述符索引，因此可得各段的段选择子:

数据段:0x8
堆栈段:0x10
视频段:0x18
代码段:0x20

接着把各段选择子存入对应的段寄存器:

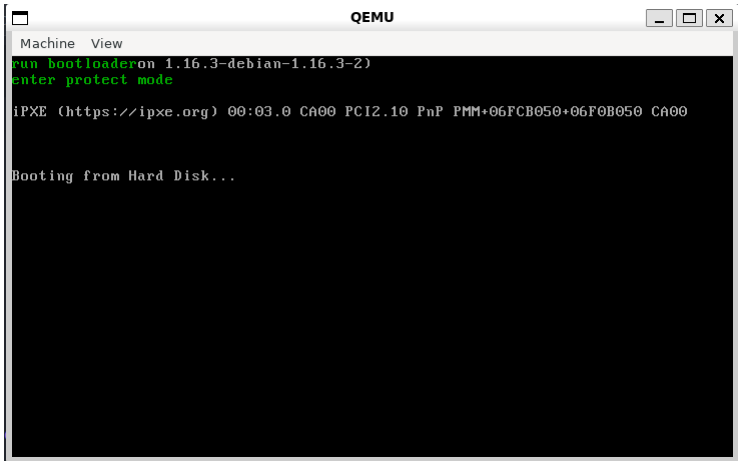
```
mov eax, DATA_SELECTOR                      ;加载数据段(0..4GB)选择子
mov ds, eax
mov es, eax
mov eax, STACK_SELECTOR
mov ss, eax
mov eax, VIDEO_SELECTOR
mov gs, eax
```

在保护模式下，寻址为选择子: 偏移地址, 以下为在保护模式下在 qemu 显示'enter protect mode' 的代码

```
mov ecx, protect_mode_tag_end - protect_mode_tag
mov ebx, 80 * 2
mov esi, protect_mode_tag
mov ah, 0x02
output_protect_mode_tag:
    mov al, [esi]
    mov word[gs:ebx], ax
    add ebx, 2
    inc esi
    loop output_protect_mode_tag
```

2.2.2 实验结果展示

如图为 example2 的结果，上面那句'run bootloader' 是在实模式下执行显示，下面的'enter protect mode' 则是处于保护模式下



2.2.3 Debug

使用 gdb 调试工具在进入保护模式的 4 个重要步骤上设置断点, 以验证进入保护模式:

- 一、查看各 GDT 的内容

```
22 ;创建描述符，这是一个数据段，对应0~4GB的线性地址空间
23 mov dword [GDT_START_ADDRESS+0x08],0x0000ffff    ; 基地址为0，段界限为0xFFFFF
24 mov dword [GDT_START_ADDRESS+0x0c],0x00cf9200    ; 粒度为4KB，存储器段描述符
25
26 ;建立保护模式下的堆栈段描述符
27 mov dword [GDT_START_ADDRESS+0x10],0x00000000    ; 基地址为0x00000000，界限0x0
28 mov dword [GDT_START_ADDRESS+0x14],0x00409600    ; 粒度为1个字节
29
30 ;建立保护模式下的显存描述符
31 mov dword [GDT_START_ADDRESS+0x18],0x80007fff    ; 基地址为0x000B8000，界限0x07FFF
32 mov dword [GDT_START_ADDRESS+0x1c],0x0040920b    ; 粒度为字节
33
34 ;创建保护模式下平坦模式代码段描述符
35 mov dword [GDT_START_ADDRESS+0x20],0x0000ffff    ; 基地址为0，段界限为0xFFFFF
36 mov dword [GDT_START_ADDRESS+0x24],0x00cf9800    ; 粒度为4kb，代码段描述符
37
```

```
remote Thread 1.1 (src) In: protect_mode_begin
Breakpoint 3 at 0x7eb6: file bootloader.asm, line 61.
(gdb) c
Continuing.

Breakpoint 3, protect_mode_begin () at bootloader.asm:61
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000      0x00cf92000000ffff
0x8810: 0x0040960000000000      0x0040920b80007fff
0x8820: 0x00cf98000000ffff
```

如图, 在保护模式开始处设置断点, 查看上面设置的五个段描述符的值，均与设想相同。可以看到段描述符的高位存在高地址，因此可以判断 **Linux-0.11** 的存储方式，是小端存储

- 二、查看第 20 根地址线是否被打开

```
      40 ;初始化描述符表寄存器GDTR
      41 mov word [pgdt], 39       ;描述符表的界限
      42 lgdt [pgdt]
      43
      44 in al,0x92                          ;南桥芯片内的端口
      45 or al,0000_0010B
      46 out 0x92,al                        ;打开A20
      47
 >    48 cli                                ;中断机制尚未工作
      49 mov eax,cr0
      50 or eax,1
      51 mov cr0,eax                        ;设置PE位
      52
      53 ;以下进入保护模式
remote Thread 1.1 (src) In:
(gdb) info registers al
al             0x2                 2
(gdb) set $al = $inb(0x92)
Invalid data type for function to be called.
(gdb) n
(gdb) set $al = $inb(0x92)
Invalid data type for function to be called.
(gdb) info registers al
al             0x2                 2
```

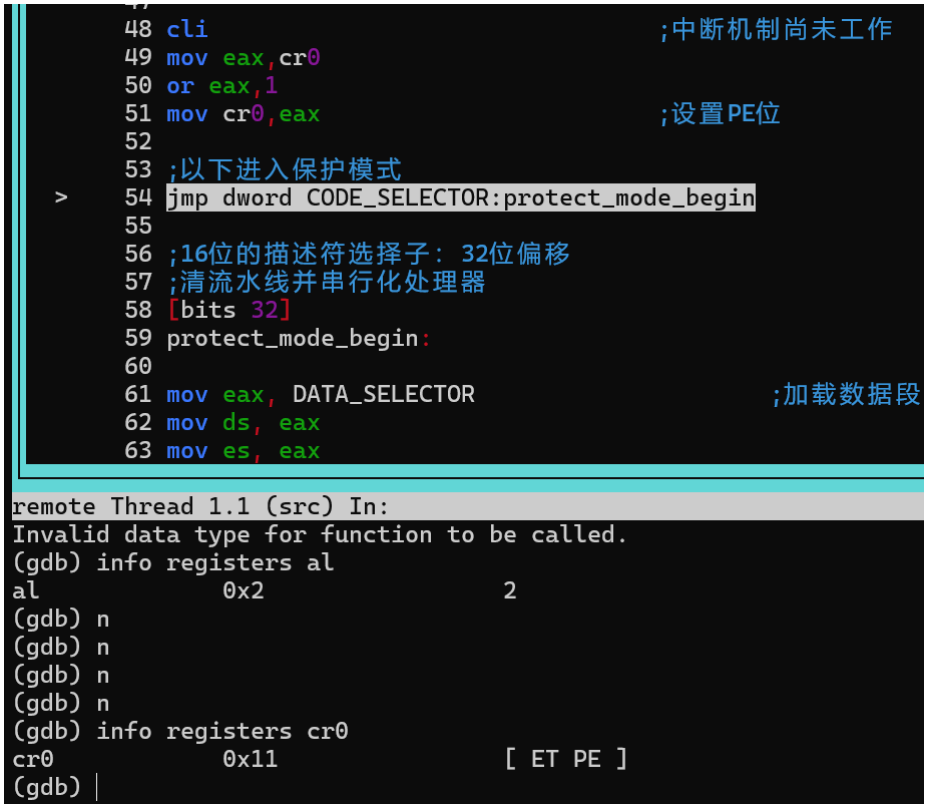
如图，寄存器 al 已被设置为 0x2，并写入 0x92 端口，因此第 20 根地址线已被打开

- 三、查看 PE 位是否被设置

```
      40 ;初始化描述符表寄存器GDTR
      41 mov word [pgdt], 39       ;描述符表的界限
      42 lgdt [pgdt]
      43
      44 in al,0x92                          ;南桥芯片内的端口
      45 or al,0000_0010B
      46 out 0x92,al                        ;打开A20
      47
 >    48 cli                                ;中断机制尚未工作
      49 mov eax,cr0
      50 or eax,1
      51 mov cr0,eax                        ;设置PE位
      52
      53 ;以下进入保护模式
remote Thread 1.1 (src) In:
(gdb) info registers al
al             0x2                 2
(gdb) set $al = $inb(0x92)
Invalid data type for function to be called.
(gdb) n
(gdb) set $al = $inb(0x92)
Invalid data type for function to be called.
(gdb) info registers al
al             0x2                 2
```

如图，PE 位已被设置

- 四、远跳转进入保护模式



2.3 实验任务三

在本实验中, 改造 Lab2 中的字符旋转程序为 32 位代码，在保护模式下运行。
由于保护模式与实模式有诸多不同，完成本实验有两大困难:

- 困难一: 在实模式中用 BIOS 中断来实现字符展示的，在保护模式下行不通:
  - 1. 实模式下的中断向量表（IVT）和保护模式下的中断描述符表（IDT）不同。在实模式里，中断服务程序的地址是直接存储于中断向量表中的；而在保护模式下，中断服务程序的地址是通过中断描述符表中的描述符来间接指定的。要是直接沿用实模式的中断调用方法，系统就无法找到正确的中断服务程序。
  - 2. 保护模式具备特权级的概念，也就是有 0 - 3 四个特权级。一般来说，操作系统内核运行在特权级 0，而用户程序运行在特权级 3。若中断服务程序的描述符权限设置有误，或者中断调用时的权限不匹配，就会触发保护异常。
  - 3. 在保护模式下，段寄存器的值代表的是段选择子，而非实模式下的段基址。要是中断服务程序里未正确设置段寄存器，就可能访问到错误的内存区域。
- 困难二: 在实模式下直接输出字符到 qemu 显存中的代码，直接照搬过来也行不通:
  因为段寄存器的值代表的是段选择子，不是实模式下的段基地址，因此需要重新设置段寄存器

解决方案: 相比于设置中断描述符表的繁琐步骤，我选择直接将字符输出到 qemu 显存中，此方法最需要注意的是要设置好段寄存器的值为段选择子。
首先, 设置段寄存器的值:

```
[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
```

清屏操作:

```
mov ecx,80*25
mov edi,0
mov ah,0x07
mov al,' '
clear:
    mov [gs:edi],ax
    add edi,2
    loop clear
```

初始化存储当前输出行、列、颜色、字符，以及方向选择的寄存器:

```
xor esi, esi         ; col
xor ebp, ebp         ; row
xor ebx, ebx         ; direction
mov al, '1'          ;
mov dl, 0x01         ; color
```

延时，不能采用中断的方式，换成 10w 次循环以达到效果:

```
    push eax
    push ebx
    mov ecx, 2000
delay_outer:
    mov eax, 5000
delay_inner:
    dec eax
    jnz delay_inner
    loop delay_outer
    pop ebx
    pop eax
```

选择运动方向:

```
cmp ebx, 0
je right
cmp ebx, 1
je down
cmp ebx, 2
je left
cmp ebx, 3
je up
```

接着是字符上下左右运动的逻辑:

```
    right:
    inc esi
    cmp esi, 80
    jb update_char
```

```
    mov esi, 79
    inc ebp
    mov ebx, 1
    jmp update_char

down:
    inc ebp
    cmp ebp, 25
    jb update_char
    mov ebp, 24
    dec esi
    mov ebx, 2
    jmp update_char

left:
    dec esi
    cmp esi, 0
    jge update_char
    mov esi, 0
    dec ebp
    mov ebx, 3
    jmp update_char

up:
    dec ebp
    cmp ebp, 0
    jge update_char
    mov ebp, 0
    inc esi
    mov ebx, 0
```

接着是每一步后更新字符和颜色的逻辑:

```
    update_char:
    cmp al, '9'
    je reset_char
    inc al
    jmp update_color
reset_char:
    mov al, '0'

update_color:
    inc dl
    cmp dl, 255
    jb main
    mov dl, 1
    jmp main
```

2.3.1 实验结果展示

3 实验总结与心得体会

3.1

在将 mbr 与 bootloader 写入磁盘前，必须先创建足够大小的磁盘:

```
qemu-img create hd.img 10m
```

否则在做 assignment2、3 时就因磁盘太小得不到希望的结果

3.2

经过本次实验，对实模式和保护模式下的寻址，中断有了充分的认识、区分。在编写代码时需要注意区别这两种模式的特点，从而以合适的方式实现自己的程序。

3.3

在程序运行遇到问题时，要擅于利用 gdb 工具来调试。我在 assignment3 时一度遇到了字符无法显示的问题，经过调试，发现问题出在了寻址方式上，错误的运用了实模式下的寻址导致地址错误，从而得以及时修正。

4 参考资料

https://blog.csdn.net/G_Spider/article/details/6906184
https://blog.csdn.net/brainkick/article/details/7583727