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（超算）

学生姓名: 黄玟瑜

学生学号: 19335074

实验地点: 中山大学广州校区东校园

实验成绩:

报告时间: 2021 年 3 月 27 日

1. 实验要求

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. 实验内容

Assignment 1

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，可以参考《于渊：一个操作系统的实现 2》P183-184。

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 lab2 的 debug 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。

3. 实验过程

Assignment 1

1.1

新建一个文件 bootloader.asm，然后将 lab2 的 mbr.asm 中输出 Hello World 部份的代码，放入 bootloader.asm，加入后的·bootloader.asm 如下所示。

```
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

output_helloworld_tag:
    mov al, 'H'
    mov [gs:2 * 80], ax

    mov al, 'e'
    mov [gs:2 * 81], ax

    mov al, 'l'
    mov [gs:2 * 82], ax

    mov al, 'l'
    mov [gs:2 * 83], ax

    mov al, 'o'
    mov [gs:2 * 84], ax

    mov al, ' '
```

```
    mov [gs:2 * 85], ax

    mov al, 'W'
    mov [gs:2 * 86], ax

    mov al, 'o'
    mov [gs:2 * 87], ax

    mov al, 'r'
    mov [gs:2 * 88], ax

    mov al, 'l'
    mov [gs:2 * 89], ax

    mov al, 'd'
    mov [gs:2 * 90], ax


jmp $ ; 死循环

bootloader_tag db 'run bootloader'
bootloader_tag_end:
```

然后我们在 mbr.asm 处放入使用 LBA 模式读取硬盘的代码，然后在 MBR 中加载 bootloader 到地址 0x7e00。使用如下指令

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

将 mbr.asm 和 bootloader.asm 转化为二进制文件，再使用

```
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

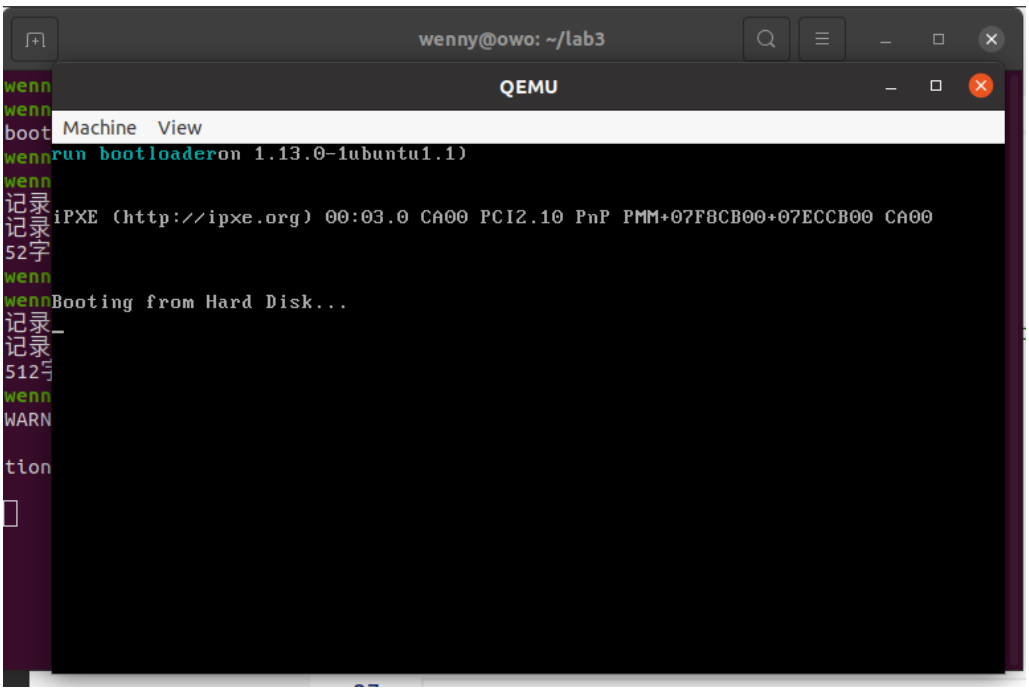
将 mbr.bin 写入编号为 0 的扇区，使用

```
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
```

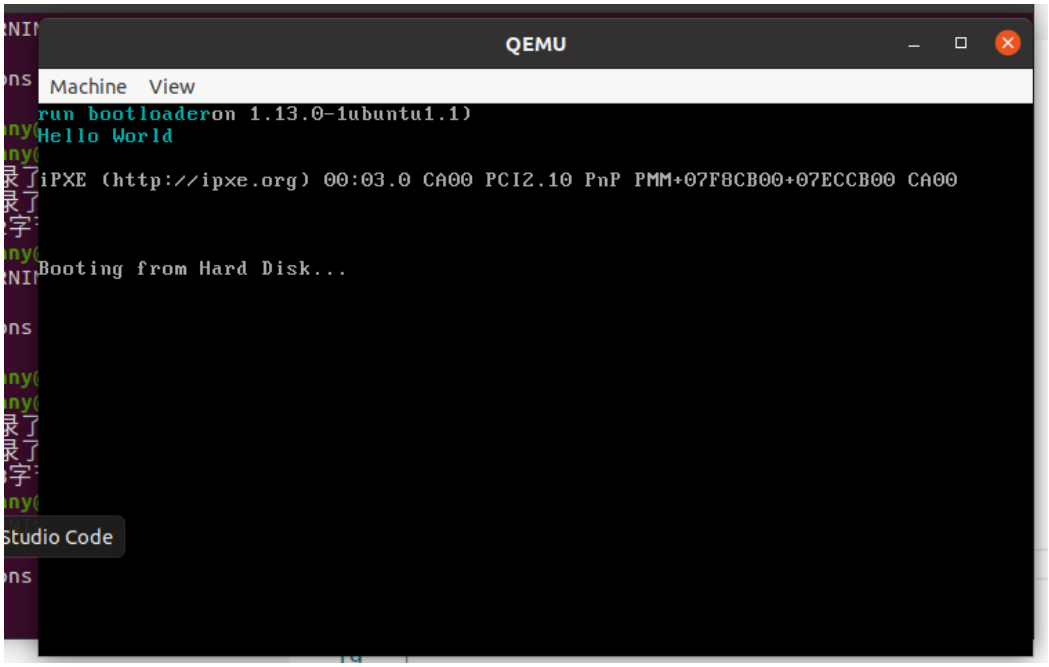
将 bootloader.bin 写入编号为 1~5 的扇区，启用 qemu

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

运行结果如下：



未加入 Hello world



加入 Hello world 后

1.2 使用 CHS 读取硬盘

使用到 BIOS 中断 INT 13h

中断号	寄存器	作用
13H	AH = 0x02 AL = 扇区数 CH = 柱面/磁道号(0 为起始号) CL = 起始扇区号(1 为起始号) DH = 磁头/盘面号，对软盘是 0 或 1 DL = 驱动器号: 软盘:0=驱动器 A，1=驱动器 B，... 硬盘:80h=驱动器 1，81h=驱动器 2，... ES:BX = 数据区中 I/O 缓冲区的地址(除检验操作外)	从磁盘读入数据到 ES:BX 指向的缓冲区中

逻辑扇区号向 CHS 的转换公式如下：

$$\frac{\text{逻辑扇区号}}{\text{单个磁道的扇区数}(18)} \rightarrow \begin{matrix}\text{商}Q \\ \text{余数}R\end{matrix}$$

$$\begin{aligned}\text{柱面号}C &= Q >> 1 \\ \text{磁头号}H &= Q\&1 \\ \text{起始扇区号}S &= R + 1\end{aligned}$$

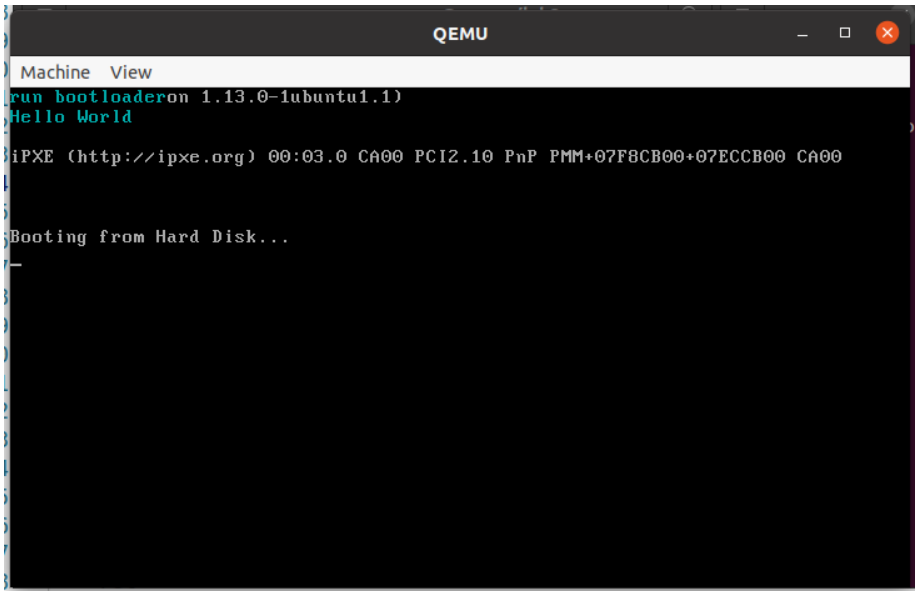
bootloader 的逻辑扇区号为 1~5，经计算，C=0，H=0，S=2~6，因此 CH=0，·CL=2，DH=0，DL=80H (驱动器 1)，EX=0x0000，BX=0x7e00，AH=02H，AL=1，编写 mbr.asm 如下所示：

```
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 ; es=0x0000
    mov fs, ax
```

```
    mov gs, ax
; 初始化栈指针
    ; mov sp, 0x7c00
    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
asm_read_hard_disk:
; 从硬盘读取一个扇区
; 入口参数：AH＝02H AL＝扇区数
; CH＝柱面 CL＝扇区
; DH＝磁头 DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘
; ES:BX＝缓冲区的地址
; 出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号 01H
中的说明
; 返回值
; bx=bx+512
    push ax

    mov cx, 0x0002
    mov dx, 0x0080
    mov ax, 0x0201
    int 0x13
    jc read_err

    pop ax
    ret

read_err:
    mov ah, 0x0d
    mov al, 'E'
    mov [gs:2 * 0], ax

    mov al, 'r'
    mov [gs:2 * 1], ax

    mov al, 'r'
    mov [gs:2 * 2], ax

    mov al, 'o'
    mov [gs:2 * 3], ax
```

```
    mov al, 'r'
    mov [gs:2 * 4], ax
    jmp $
times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

运行结果如下：



结果符合预期。

Assignment 2

首先创建 bootloader.asm，内容如如下所示：

D: > oslab > lab3 > src > example-2 > bootloader.asm

```
%include "boot.inc"
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

;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

```

在最后加上如下语句，填充文件到 512*5 字节大小。

```
times 512*5 - ($ - $$) db 0
```

创建 mbr.asm，如下所示：

D: > oslab > lab3 > src > example-2 > mbr.asm

```
%include "boot.inc"
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, 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:
```

在相同目录下创建 Makefile 和 gdbinit，Makefile 如下：

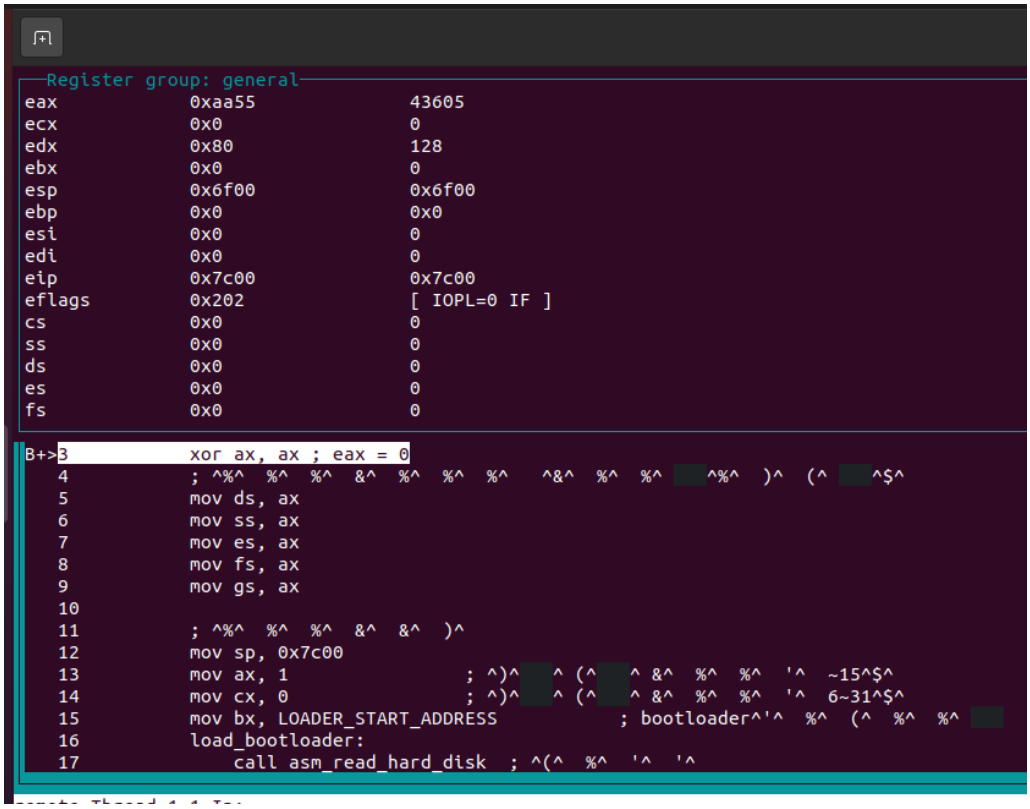
OS > Makefile

```
run:
    @qemu-system-i386 -hda hd.img -serial null -parallel stdio
debug:
    @qemu-system-i386 -s -S -hda hd.img -serial null -parallel stdio &
    @sleep 1
    @gnome-terminal -e "gdb -q -x gdbinit"
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 bootloader.o
    @ld -o bootloader.symbol -melf_i386 -N bootloader.o -Ttext 0x7e00
    @ld -o bootloader.bin -melf_i386 -N bootloader.o -Ttext 0x7e00 --oformat binary
    @dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
    @dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
clean:
    @rm -fr *.bin *.o

```

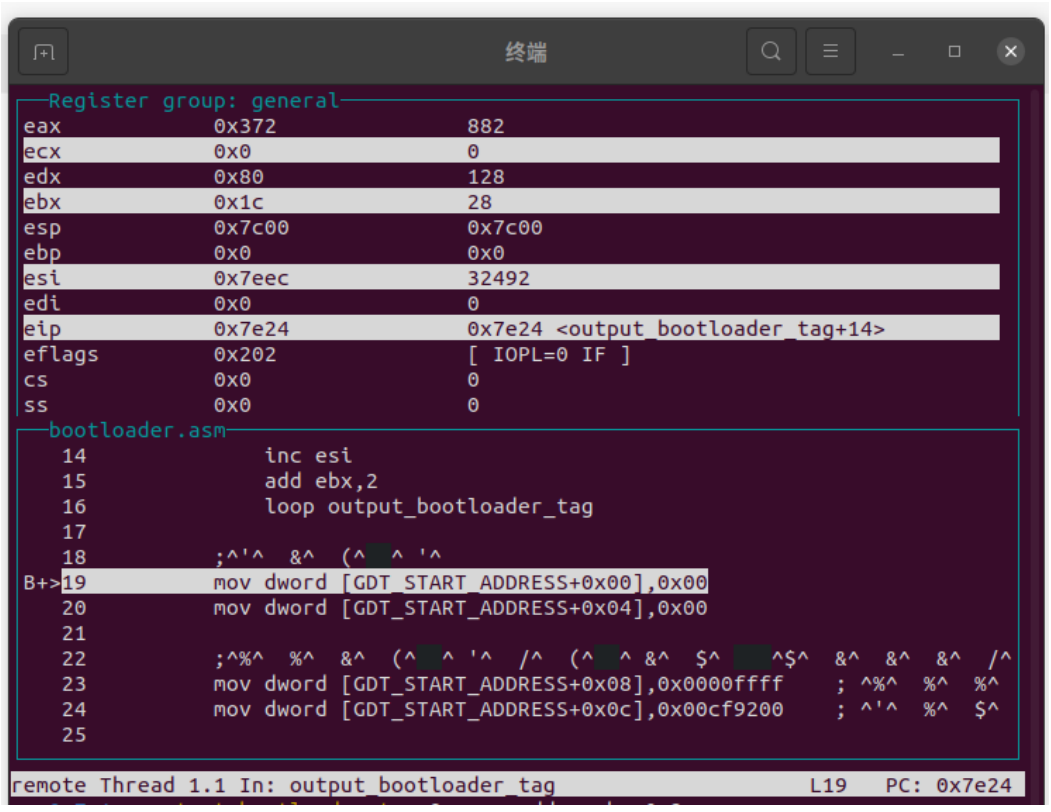
输入命令

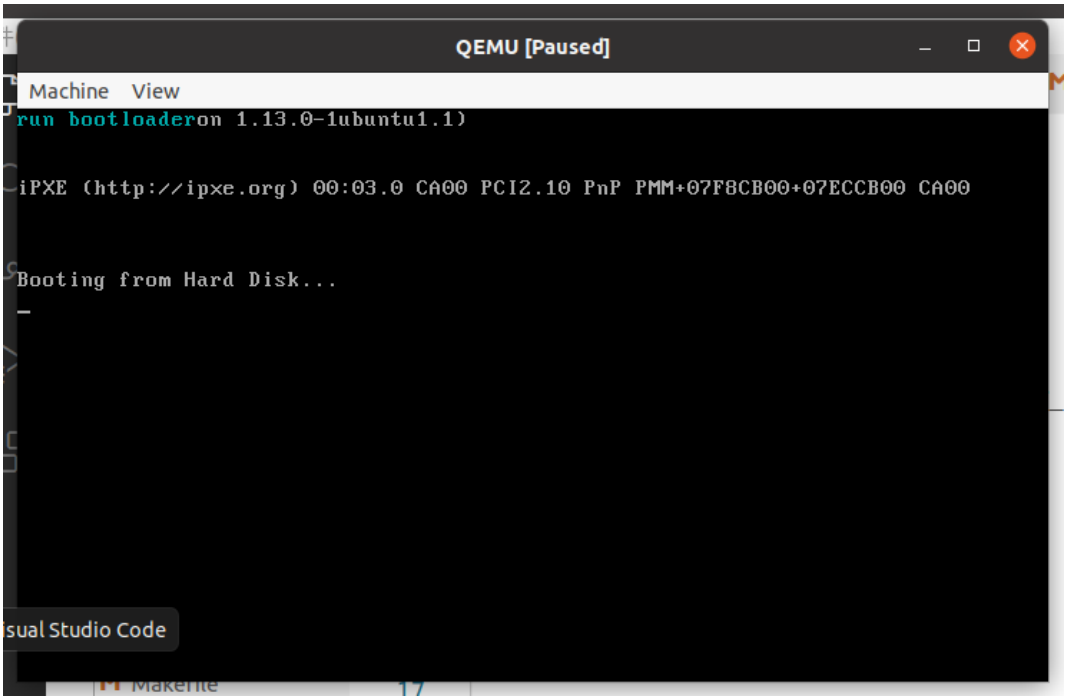
```
make build
make debug
```

在 0x7c00 设置断点运行后看到如下界面：



运行到 0x7e24 时，可以看到屏幕上输出了 run bootloader，说明 bootloader 以及顺利加载到相应地址并成功开始运行。

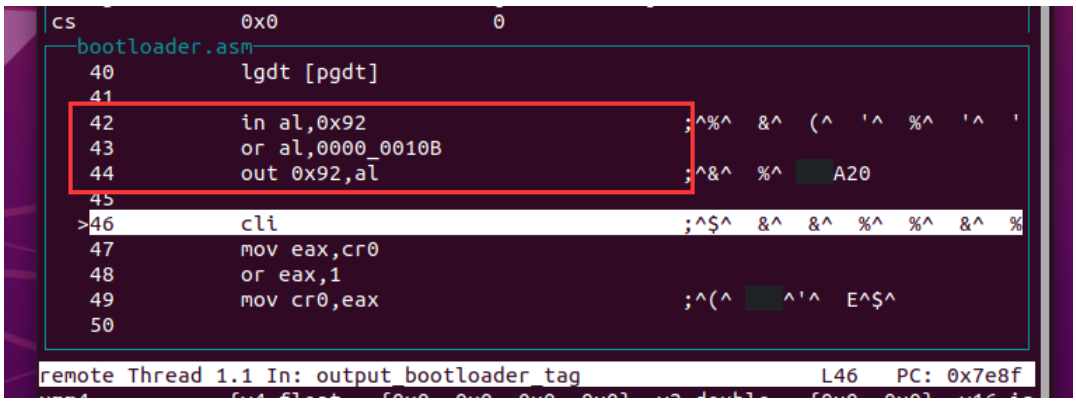




继续运行程序，程序依次完成 GDT 的准备、GDTR 信息的加载，加载 GDTR 信息完毕后查看 GDT 的 5 个段描述符信息，如下所示：

```
(gdb) si
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000      0x00cf92000000ffff
0x8810: 0x0040960000000000      0x0040920b80007fff
0x8820: 0x00cf98000000ffff
(gdb)
```

接下来打开 A20 地址线，打开的步骤如图：



查看 cr0 的信息，可以看到 cr0 处于关闭状态：

```
remote Thread 1.1 In: output_bootloader_tag
fs_base         0x0                 0
gs_base         0xb8000             753664
k_gs_base       0x0                 0
cr0             0x10                [ ET ]
cr2             0x0                 0
cr3             0x0                 [ PDBR=0 PCID=0 ]
cr4             0x0                 [ ]
cr8             0x0                 0
efer            0x0                 [ ]
xmm0            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_do
```

运行以下步骤，开启 cr0 的保护模式标志位：

```
cs              0x0                 0
 bootloader.asm
   45
   46           cli                                 ;^$^  &^  &^  %^  %^  &^  %
   47           mov eax,cr0
   48           or eax,1
   49           mov cr0,eax                         ;^(^      ^'^  E^$^
   50
   51           ;^$^  $^  (^   ^ %^  $^   ^ &^  &^  %^
  >52           jmp dword CODE_SELECTOR:protect_mode_begin
   53
   54           ;16^$^  '^  &^  (^   ^ '^  )^     ^ &^  %^  /^  2^$^  %^     ^ '^
   55           ;^&^  &^     ^&^  '^    ^%^  $^  (^  %^  %^  '^  %^
remote Thread 1.1 In: output_bootloader_tag                    L52   PC: 0x7e9a
```

再次查看 cr0 信息，可以看到 cr0 已经被打开：

```
remote Thread 1.1 In: output_bootloader_tag
gs_base         0xb8000             753664
k_gs_base       0x0                 0
cr0             0x11                [ ET PE ]
cr2             0x0                 0
cr3             0x0                 [ PDBR=0 PCID=0 ]
cr4             0x0                 [ ]
cr8             0x0                 0
efer            0x0                 [ ]
xmm0            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_doubl
```

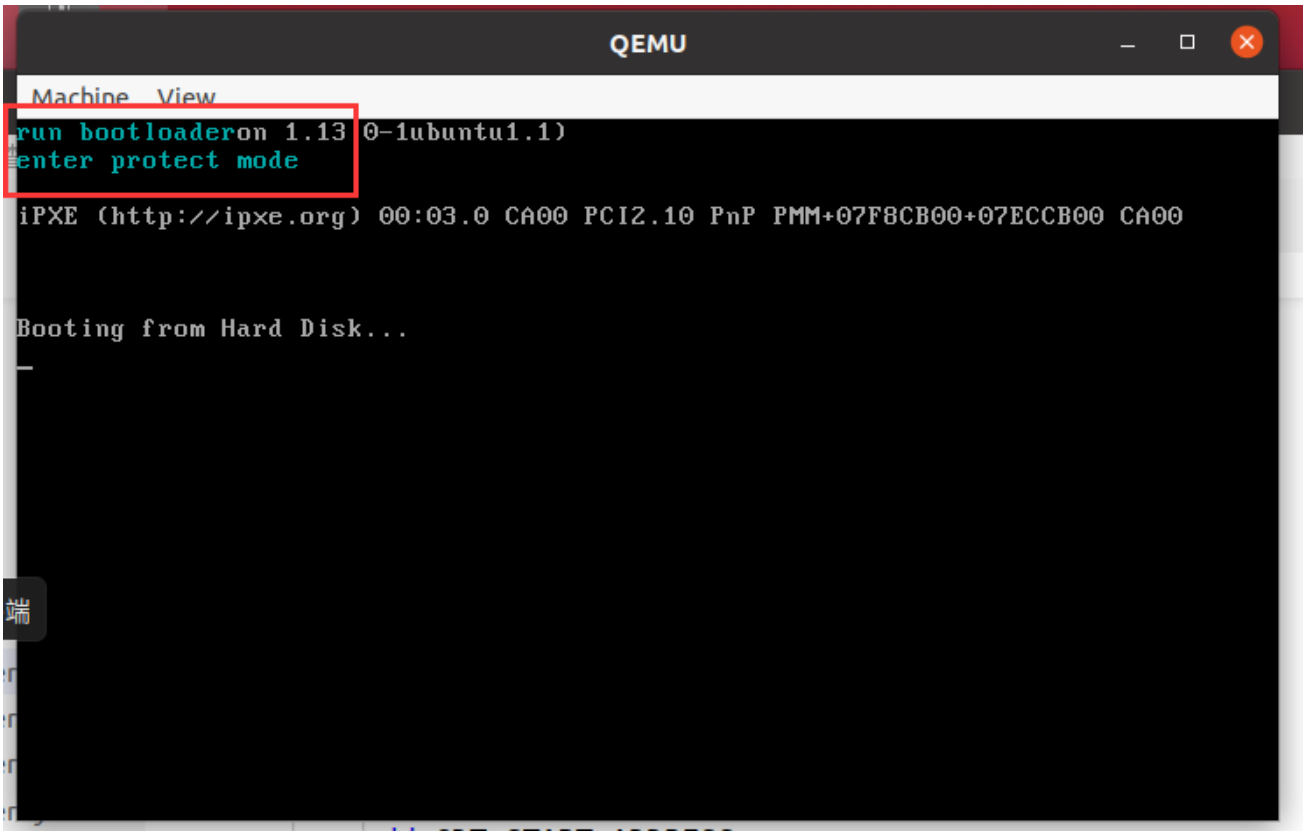
进行远跳转，指令如下：

```
jmp dword CODE_SELECTOR:protect_mode_begin
```

在保护模式开始处设置断点：

```
Breakpoint 6, protect_mode_begin () at bootloader.asm:59
(gdb)
```

跳转完毕后，界面上输出“enter protect mode”



Assignment 3

修改后的主要部分代码如下：

```
assigment3:
xor bx,bx
xor ecx,ecx
xor dx,dx
mov dx, 0x0200
mov bx, 0x0101
; dh 行 dl 列
; bh 行方向 bl 列方向 1=increase 0=decrease
bounce:
    mov ax,0

;cursor 1:position
    mov al, dh
    imul cx, ax, 80
    mov al, dl
    add cx, ax
    shl cx, 1
    mov esi, ecx
```

```
    call print_a_char

; ;cursor 2:position which is symatrical to the other
    mov al, dh
    sub al, 24
    neg al
    imul cx, ax, 80
    mov al, dl
    sub al, 79
    neg al
    add cx, ax
    shl cx, 1
    mov esi, ecx
    call print_a_char


;always display
    mov ah, 0x0d
    mov al, 'w'
    mov [gs:2 * 33], ax
    mov al, 'e'
    mov [gs:2 * 34], ax
    mov al, 'n'
    mov [gs:2 * 35], ax
    mov al, 'n'
    mov [gs:2 * 36], ax
    mov al, 'y'
    mov [gs:2 * 37], ax

    mov al, 0
    mov [gs:2 * 38], ax
    mov al, '1'
    mov [gs:2 * 39], ax
    mov al, '9'
    mov [gs:2 * 40], ax
    mov al, '3'
    mov [gs:2 * 41], ax
    mov al, '3'
    mov [gs:2 * 42], ax
    mov al, '5'
    mov [gs:2 * 43], ax
    mov al, '0'
    mov [gs:2 * 44], ax
    mov al, '7'
    mov [gs:2 * 45], ax
```

```
    mov al, '4'
    mov [gs:2 * 46], ax

;change position depending on the direction stored in bx
;if inc
    add dh, bh
    add dl, bl
;if dec
    mov ch, bh
    cmp ch, 0
    jne L1          ;if equal, decrease dh by 1, or else just skip it
    call dec_dh
L1:
    mov ch, bl
    cmp ch, 0
    jne L2
    call dec_dl     ;if equal, decrease dl by 1
L2:

    call getDir     ;update the direction

    call delay

    jmp bounce      ;loop

dec_dh:
    dec dh
    ret
dec_dl:
    dec dl
    ret
getDir:
;if the cursor meets the up and down side, flip dl
;if the cursor meets the left and right side, flip dh
    mov ch, dh
    cmp ch, 0
    jne L3
    call flip_bh
L3:
    mov ch, dh
    cmp ch, 24
    jne L4
    call flip_bh
L4:
```

```
    mov ch, dl
    cmp ch, 0
    jne L5
    call flip_bl
L5:
    mov ch, dl
    cmp ch, 79
    jne getDir_end
    call flip_bl
getDir_end:
    ret


flip_bl:
    xor bl,1
    ret
flip_bh:
    xor bh,1
    ret
print_a_char:
    mov ch, [count]
    call inc_count  ;increase to change the color
    mov cl, 'w'
    mov word [ gs : esi ], cx
    ret
delay:
    pushad
    mov ecx, 0x00070000
delay_s:

    loop delay_s
    popad

    ret
inc_count:
    mov al, [count]
    inc al
    mov [count], al
    ret

count db 0x03
```

程序设计的思想和实模式下大致相同，不同处在于不能使用实模式下的 BIOS 中断来实现延时，因此修改延时函数如下：

```
delay:
    pushad
    mov ecx, 0x00070000
delay_s:

    loop delay_s
    popad
    ret
```

使用 loop 指令不断的空循环，从而实现延时。

实验结果如下：

4. 总结

在这次实验中，我学会了用 LBA 和 CHS 两种方式读取硬盘，LBA 方式读取磁盘要注意逻辑扇区号各个位和 I/O 端口的对应关系、以及磁盘状态的判断，要等到磁盘不再繁忙时才能开始读。

与 LBA 方式相比，CHS 方式的代码量更少，也更为简洁，只需要计算出磁头号、

柱面号、扇区号以及扇区数，再选择驱动器，最后调用中断即可。

在 Assignment2 中遇到的问题主要是没有对 bootloader 的大小进行填充，没有最后那一行填充代码会导致只有头一个扇区可以被读取，后 4 个扇区的内容无法读取。一开始认为是读取方式的问题，反复尝试，浪费了很多时间，最后发现了问题所在，对 bootloader 进行填充后成功完成了实验。仔细阅读实验指导以及相关书籍，了解和基本掌握了从实模式到保护模式的方法，在我看来这更多的是一下硬性的规则，希望能在以后的一次一次实验中逐渐掌握于心。

最后一个实验遇到的问题是在保护模式下实模式的 BIOS 中断不能再调用，虽然体会了保护模式的好处（拥有更大的寻址空间等），但 BIOS 中断却不能调用了。最后通过查阅资料和思考，想到了空循环来实现延时。这是主要的也是唯一一个问题，实模式下的代码在保护模式基本是能运行的，其他地方没有太大的改动。

总而言之，此次实验使我受益匪浅。

参考：

https://blog.csdn.net/loomman/article/details/3052995