1 实验概述

学习 X86 汇编, 并完成以下任务:

- 编写 MBR, 使其加载在操作系统中显示”Hello World” 与自己的学号
- 学习实模式下的中断, 利用中断实现光标的位置获取和光标移动、使用实模式下的中断来输出学号、利用键盘中断实现输入键盘输入并回显
- 使用 X86 汇编语言, 分别实现分支逻辑、循环逻辑、过程调用
- 利用汇编制作字符回旋程序, 实现在 qemu 显示屏上以不同颜色和字符内容进行绕圈

2 实验过程

2.1 实验任务一

2.1.1 复现 example1

计算机在加电启动时, 会找到首扇区并将首扇区的 512 字节加载到内存 0x7c00 处执行, 因此在程序中需要通过汇编伪指令指定代码中的代码标号和数据标号从内存地址的 0x7c00 开始:

```
org 0x7c00
[bits 16]     ;按16位编码格式编译代码
```

计算机 BIOS 识别该扇区是有效引导扇区的标志是扇区的最后两个字节:0x55 和 0xaa, 因此需要在程序末尾将该扇区填满, 并显式指定最后两个字节:

```
times 510 - ($ - $$) db 0   ;计算从当前位置到510字节处还需要填充多少字节,并填0
db 0x55, 0xaa    ;最后两个字节填0x55  0x55
```

qemu 显示屏的坐标与显存地址的关系为:

$$\text{显存起始地址} = 0xB8000 + 2 * (80x + y)$$

在 16 位实模式下, 物理地址由段地址寄存器 gs 和偏移地址表示:

$$\text{物理地址} = gs << 4 + \text{偏移地址}$$

因此给段地址寄存器 gs 赋初始值:

```
    mov ax, 0xb800  ;gs=0xB8000>>4=0xb800
    mov gs, ax
```

在输出字符时, 要注意 mov 指令的操作数大小要配对,char 为 8 位, 赋给 al, 而在显存中每个显示字符用两个字节表示, 因此要用 ax:

```
    mov al, 'H'
    mov [gs:2 * 0], ax
```

完成汇编代码后用以下指令启动 qemu 模拟计算机启动:

```
nasm -f bin mbr.asm -o mbr.bin
qemu-img create hd.img 10m
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
qemu-system-i386 -hda hd.img -serial null -parallel stdio
```

启动后便能看到输出 Hello World。

2.1.2 显示自己学号

使用不同的前景色和背景色, 从 (16,10) 开始输出我的学号 23336326
计算坐标时, 运用此公式:

$$显存起始地址 = 0xB8000 + 2 * (80x + y)$$

前景色和背景色通过寄存器 ah 来设定, 高四位决定背景色, 低四位决定前景色。具体细节如下表:

R	G	B	背景色	前景色	
			K=0 时不闪烁,K=1 时闪烁	I=0	I=1
0	0	0	黑	黑	灰
0	0	1	蓝	蓝	浅蓝
0	1	0	绿	绿	浅绿
0	1	1	青	青	浅青
1	0	0	红	红	浅红
1	0	1	品（洋）红	品 (洋) 红	浅品 (洋) 红
1	1	0	棕	棕	黄
1	1	1	白	白	亮白

我采用白底红字, 因此颜色编码为 0x74, 主要代码如下:

```
mov ah, 0x74 ; red
mov al, '2'
mov [gs:2580], ax
mov al, '3'
mov [gs:2582], ax
mov al, '3'
mov [gs:2584], ax
mov al, '3'
mov [gs:2586], ax
mov al, '6'
mov [gs:2588], ax
mov al, '3'
mov [gs:2590], ax
mov al, '2'
mov [gs:2592], ax
mov al, '6'
mov [gs:2594], ax
```

2.2 实验任务二

本实验任务与 BIOS 中断调用相关,BIOS 在 X86 的第 17 个中断向量中建立了一个中断处理程序提供实模式下的视频服务。调用这个功能时, 在寄存器 AH 赋予子功能号, 在其它寄存器赋予其它所需的参数, 用指令 int 10H 调用即可。

2.2.1 利用中断实现光标的位置获取和光标移动

获取光标位置:
使用子功能 AH=03H, 光标位置的行与列返回在寄存器 dh 和 dl 中:

```
mov ah,03H
mov bh,00H
int 10H
```

调整光标位置:
使用子功能 AH=02H, 通过寄存器 dh 和 dl 设置光标的行与列:

```
    add dl,10    ;调整光标的位置为列坐标+10
    mov ah,02H
    mov bh,00H
    int 10H
```

即可看到光标位置的变化

2.2.2 使用实模式的中断输出学号

利用实模式的中断输出我的学号:23336326, 利用子功能 AH=13H, 各寄存器的功能如下:

- al 设定写模式, 如字符串只包含字符或者是同时包含字符和属性
- bh 设定页码
- bl 设定颜色, 颜色是的定义同上
- cx 设定字符串长度
- dh 和 dl 设定字符串起始位置的行与列
- bp 设定字符串的偏移地址

代码如下:
数据段设字符串:

```
    msg db '23336326',0
```

代码段:

```
    mov si,msg ;获取字符串首地址
    mov ah,0x13
    mov al,0x01
    mov bh,0x00
```

```
    mov bl,0x74
    mov dh,5
    mov dl,10
    mov bp,si
    mov cx,8
    int 0x10
```

即可看到学号显示

2.2.3 实现键盘的输入并回显

运用中断 int 16h 获取键盘的单个字符输入到寄存器 al, 通过中断 int 10h 的子功能 AH=0EH 进行回显, 实现如下:

```
start:
    mov ah,00H
    int 16h    ;获取输入到键盘

    mov ah,0eH
    mov bh,00H
    int 10h    ;在qemu进行回显

    cmp al,27 ;若输入是esc(ascll码为27)就退出
    jne start
```

但是在实验时发现:
当输入退格键时, 光标只是进行了简单的移动, 被” 退格” 掉的字符并没有消失;
当输入回车键时, 光标只是移到了当前行的开头, 接下来输入的字符会覆盖掉当前的字符, 与设想的功能不一致。
因此在代码中完善了 Backspace 和 Enter 键的功能, 为了代码简洁, 将字符回显包装为 print 函数进行代码复用, 具体功能的实现在注释中给出:

```
start:
    mov ah,00H
    int 16h
    cmp al,0dH  ;处理回车键(ascll码为0dh)
    je handle_enter
    cmp al,08H  ;处理空格键(ascll码为08h)
    je handle_backspace
    call print
    cmp al,27   ;若输入是esc(ascll码为27)就退出
    jne start
    jmp exit
handle_enter:   ;对回车的处理:输出回车键+换行键
    mov al,0dH
    call print
    mov al,0aH
    call print
    jmp start
handle_backspace: ;对空格的处理:输出退格键+空格键+退格键,实现“退格”的效果
```

```
    mov ah,03H
    mov bh,00H
    int 10h
    cmp dl,00H    ;特判:在行首时退格无效
    je start
    mov al,08H
    call print
    mov al,' '
    call print
    mov al,08H
    call print
    jmp start

print:
    mov ah,0eH
    mov bh,00H
    int 10h
    ret
exit:
```

2.3 实验任务三

使用 32 位寄存器, 用 x86 汇编实现以下指定的逻辑功能:
分支逻辑:

```
if a1 >= 40 then
    if_flag = (a1 + 3) / 5
else if a1 >= 18 then
    if_flag = 80 - (a1 * 2)
else
    if_flag = a1 << 5
end
```

循环逻辑:

```
while a2 < 25 then
    call my_random          // my_random将产生一个随机数放到eax中返回
    while_flag[a2 * 2] = eax
    ++a2
end
```

函数的实现:

```
your_function:
    for i = 0; string[i] != '\0'; ++i then
        pushad
        push (string[i] + 9) to stack
        call print_a_char
        pop stack
```

```
            popad
        end
        return
    end
```

2.3.1 分支逻辑

在本段代码中, 需要重点注意除法的处理: idiv 有符号除法指令中, 被除数为 64 位:(edx:eax) 组成, 因此在除法前需要将 eax 的符号位扩展至 edx, 组成 64 位被除数。代码的实现如下:

```
your_if:
; put your implementation here
    mov eax,[a1]

    cmp eax,40
    jl branch1

    add eax,3
    cdq   ;扩展eax的符号位至edx
    mov ecx,5
    idiv ecx
    mov [if_flag],eax

    jmp done
branch1:
    cmp eax,18
    jl branch2

    mov ecx,eax
    shl ecx,1
    mov eax,80
    sub eax,ecx
    mov [if_flag],eax

    jmp done
branch2:
    shl eax,5
    mov [if_flag],eax
done:
```

2.3.2 循环逻辑

这里需要特别注意 while_flag 的使用。
观察 test.cpp 中 while_flag 的定义:

```
    char *while_flag, *random_buffer;
```

while_flag 是一个指向字符数组起始位置的指针, 则在 nasm 汇编中, 对于变量 while_flag :

[while_flag] 是 while_flag 的值, 即字符数组的起始地址; [[while_flag]] 才是字符数组的第一个字符。
理清楚这一点之后, 代码就能顺利运行:

```
your_while:
; put your implementation here
    mov eax,[a2]
    mov esi,[while_flag]
while:
    cmp eax,25
    jge endwhile
    mov [a2],eax

    call my_random
    mov ecx,[a2]
    shl ecx,1
    mov [esi+ecx],al  ;相当于[[while_flag]+a2*2]

    mov eax,[a2]
    inc eax
    jmp while
endwhile:
    mov [a2],eax
```

同时还要同步更新 a2 的值, 一方面寄存器 eax 会被 my_random 函数破坏, 另一方面 a2 的值不更新则会过不了 test.cpp 的测试。

2.3.3 函数的实现

在这里首先需要理解 pushad 和 popad 的意义:
pushad 是入栈所有通用寄存器的状态,popad 同理, 用于在调用 print_a_char 前保存寄存器。
同时, 在调用完函数后需要记得弹栈, 否则再后序 popad 时会出现参数传递错误。

```
your_function:
; put your implementation here
    mov esi,[your_string]
    xor ecx,ecx
for:
    mov al,[esi+ecx]
    test al,al
    jz end_for
    pushad

    add al,9
    movzx eax,al
    push eax
    call print_a_char
    add esp,4   ;要记得手动弹栈！！！
```

```
    popad
    inc ecx
    jmp for
end_for:
    ret
```

2.4 实验任务四

用汇编代码实现字符回旋程序, 使其在 qemu 显示屏上面以不同颜色、字符内容进行顺时针绕圈。

2.4.1 字符在最外侧绕圈

由于代码中涉及的寄存器过多, 以下为主要寄存器在本代码中的功能:

- si: 当前光标的列
- di: 当前光标的行
- bx: 当前光标的运行方向:0,1,2,3 分别代表右下左上
- al: 当前输出的字符
- 数据段 color: 当前输出的颜色, 范围为 1-256

首先使用 int 10h 中断中的 AH=06H 进行清屏操作, 并把光标移至屏幕的 (0,0) 开始处:

```
;clear screen
mov ah,06h
mov al,0
mov bh,07h
mov ch,0
mov cl,0
mov dh,24
mov dl,79
int 10h

;reset clip
mov ah,02h
mov bh,0
mov dh,0
mov dl,0
int 10h
```

接着是代码的主要段落, 分为移动光标、打印字符、延时效果、判断方向、上下左右操作的实现、更新下一个字符、更新下一种颜色:

```
    ;重置寄存器
    mov si,0    ;col
    mov di,0    ;row
    mov bx,0    ;direction
    mov al,'1'

    print:
        push bx
        ;bx存着方向状态,下面使用了bx寄存器,因此要先备份
        mov ah,02h
        mov bh,0
        ;这里是将行列di和si分别赋值给dh和dl,但是di和si是16位,dh和dl是8位,不能直接mov
        ;需要借助寄存器ax和al进行赋值
        push ax
        ;还要注意,al中存着当前的输出字符,在利用ax时要先保存ax的初始值
        mov ax,di
        mov dh,al
        mov ax,si
        mov dl,al
        pop ax   ;恢复ax的值
        int 10h ;调整光标位置

        mov ah,09h
        mov bh,0
        mov bl,[color]
        mov cx,1
        int 10h ;输出字符

        pop bx   ;恢复bx的值
        ;delay time,采用10w次循环进行延时效果
        mov cx,2000
    outer_loop:
        mov dx,5000
    inner_loop:
        dec dx
        jnz inner_loop
        loop outer_loop
        ;判断方向
        cmp bx,0
        je right

        cmp bx,1
        je down

        cmp bx,2
        je left
```

```
47
48          cmp bx,3
49          je up
50
51          cmp bx,4
52          je done
53
54      right:
55          inc si
56          cmp si,80
57          jb updatechar
58
59          mov si,79
60          mov bx,1    ;到边界,改变方向
61          inc di
62          jmp updatechar
63
64      down:
65          inc di
66          cmp di,25
67          jb updatechar
68
69          mov di,24
70          mov bx,2
71          dec si
72          jmp updatechar
73
74      left:
75          dec si
76          cmp si,0
77          jge updatechar
78
79          mov si,0
80          mov bx,3
81          dec di
82          jmp updatechar
83
84      up:
85          dec di
86          cmp di,0
87          jge updatechar
88
89          mov di,0
90          mov bx,0
91          inc si
92          jmp updatechar
```

```
93
94      updatechar:
95          cmp al,'9'
96          je resetchar
97
98          inc al
99          jmp updatecolor
100     resetchar:
101         mov al,'0'
102
103     updatecolor:
104         push ax  ;记得备份
105         mov al,[color]
106         inc al
107         cmp al,256
108         jne setcolor
109         mov al,1
110     setcolor:
111         mov [color],al
112         pop ax
113
114         jmp print
115
116     done:
```

编写此代码时, 需要注意:

- mov 指令的两个操作数必须等长
- 若正在使用的寄存器将被破坏, 需要进行压栈备份
- 在使用 BIOS 中断时, 需要注意当前寄存器的值会不会被覆盖, 如 int 10h, 在设定 bh 和 bl 时将 bx 的值也覆盖了, 这时需要压栈备份 bx。

2.4.2 字符不断向内绕圈

在 4.1 的基础上, 加上四个角落的限制, 即可实现向内绕圈。由于寄存器数量不足, 因此采用数据段的变量进行四个角落的标记。

```
1       mincol db 0
2       maxcol db 79
3       minrow db 0
4       maxrow db 24
```

字符不能无休止的向内绕圈, 因此需要进行边界检查:

```
1       push ax   ;需要进行压栈备份
2       mov al,[mincol]
```

```
    cmp al,[maxcol]
    jg done

    mov al,[minrow]
    cmp al,[maxrow]
    jg done
    pop ax
```

而字符串上下左右绕圈的主要代码段如下:

```
right:
    inc si
    mov cx,0
    mov cl,[maxcol]
    cmp si,cx
    jbe updatechar

    mov cx,0
    mov cl,[maxcol]
    mov si,cx
    inc byte [minrow]
    mov bx,1
    inc di
    jmp updatechar

down:
    inc di
    mov cx,0
    mov cl,[maxrow]
    cmp di,cx
    jbe updatechar

    mov cx,0
    mov cl,[maxrow]
    mov di,cx
    dec byte [maxcol]
    mov bx,2
    dec si
    jmp updatechar

left:
    dec si
    mov cx,0
    mov cl,[mincol]
    cmp si,cx
    jge updatechar

```

```
    mov cx,0
    mov cl,[mincol]
    mov si,cx
    dec byte [maxrow]
    mov bx,3
    dec di
    jmp updatechar

up:
    dec di
    mov cx,0
    mov cl,[minrow]
    cmp di,cx
    jge updatechar

    mov cx,0
    mov cl,[minrow]
    mov di,cx
    mov bx,0
    inc byte [mincol]
    inc si
    ;mov bx,4
    jmp updatechar
```

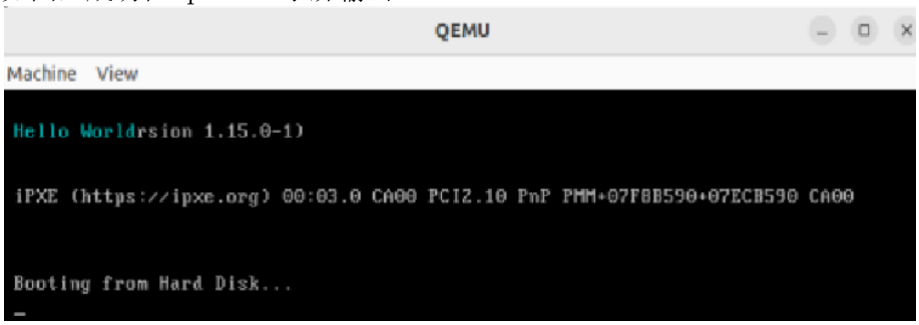
需要格外注意的是: 变量大小为 8 位, 而 di 和 si 寄存器是 16 位, 在边界检查时不能直接比较, 需要借助 16 位寄存器 cx 进行比较。

3 实验结果展示

3.1 实验任务一

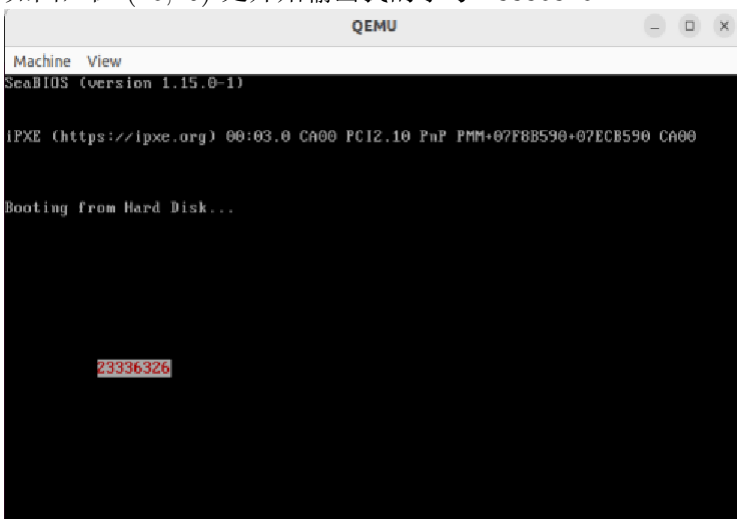
3.1.1 复现 example1

如图，成功在 qemu 显示屏输出 HelloWorld



3.1.2 显示自己的学号

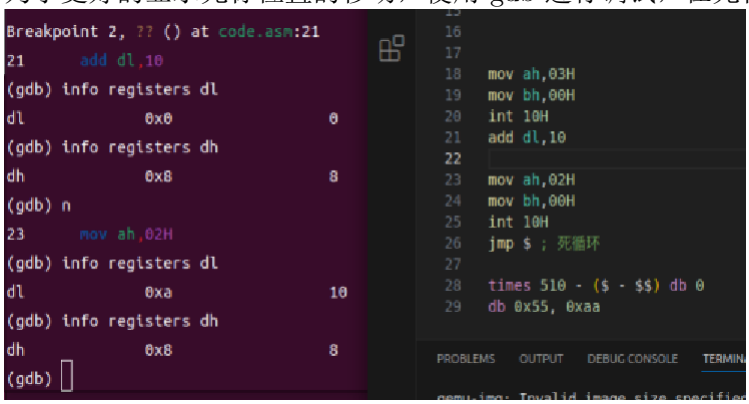
如图，在 (16,10) 处开始输出我的学号 23336326



3.2 实验任务二

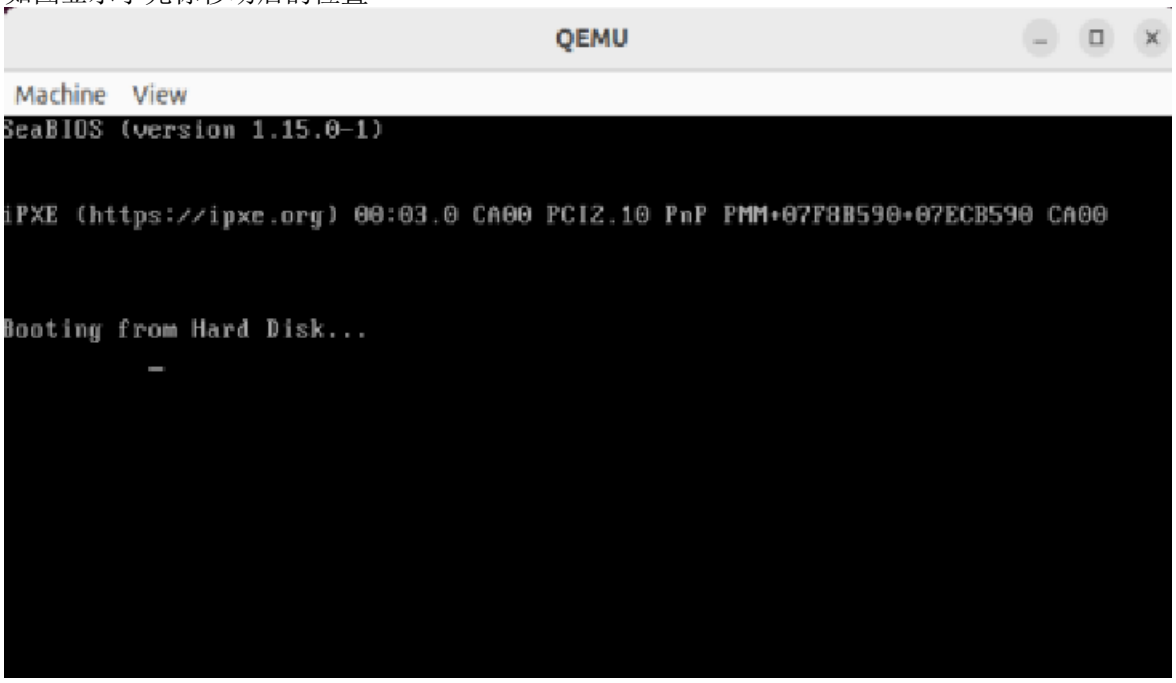
3.2.1 利用中断实现光标的位置获取和光标移动

为了更好的显示光标位置的移动，使用 gdb 进行调试，在光标移动前后查看寄存器 dl 和 dh 的值:

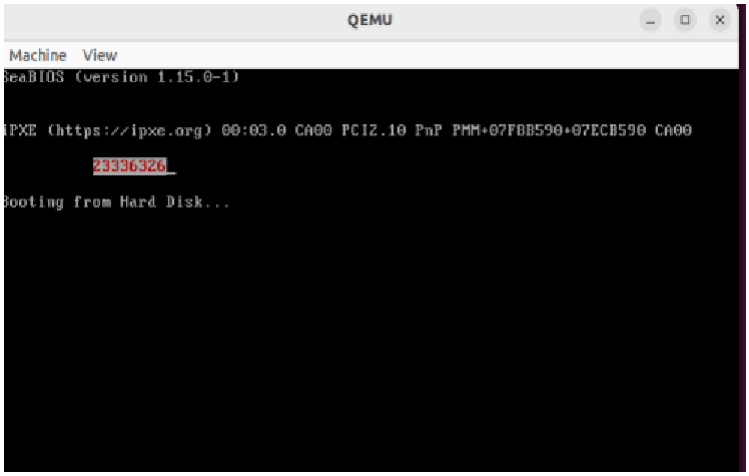


在移动前，光标的坐标为 (8,0)，移动后光标的坐标为 (8,1)

如图显示了光标移动后的位置

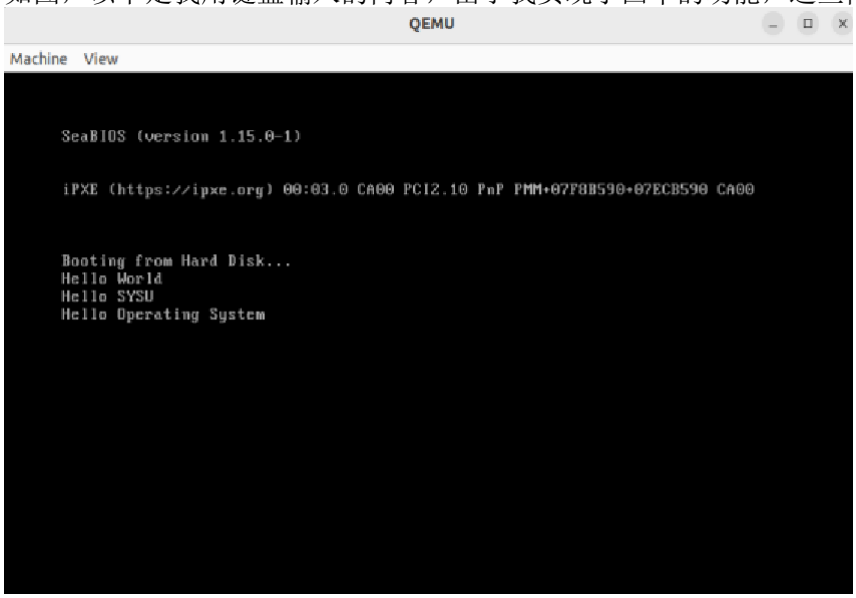


3.2.2 使用实模式下的中断输出学号



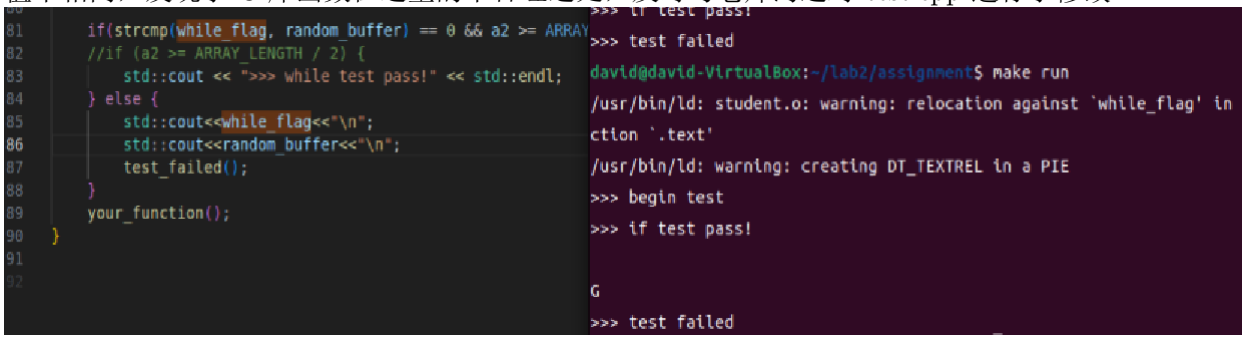
3.2.3 键盘中断实现键盘输入并回显

如图，以下是我用键盘输入的内容，由于我实现了回车的功能，这些内容得以换行展示:

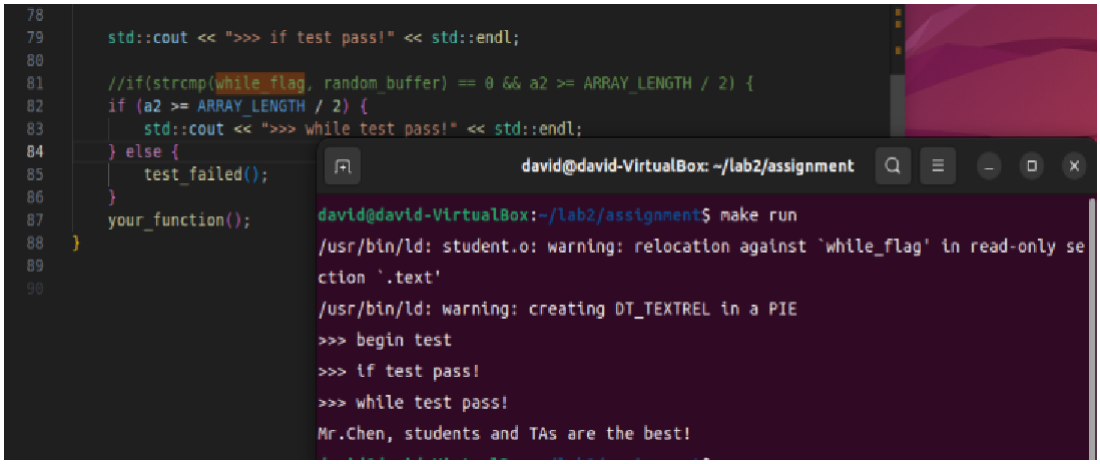


3.3 实验任务三

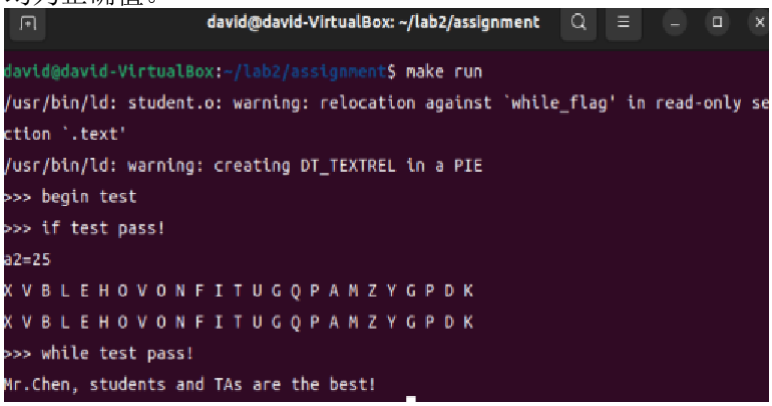
本实验遇到了许多困难，while test 多次尝试仍未通过。如下图: 在输出了 while_flag 和 random_buffer 后发现两者的值不相同，发现了 C 库函数在这里的不合理之处，及时与老师沟通对 test.cpp 进行了修改。



但修改后仍未通过 while test, 于是先单独检查了 a2，发现 a2 的值没问题，那么问题就出在了 while_flag 中。



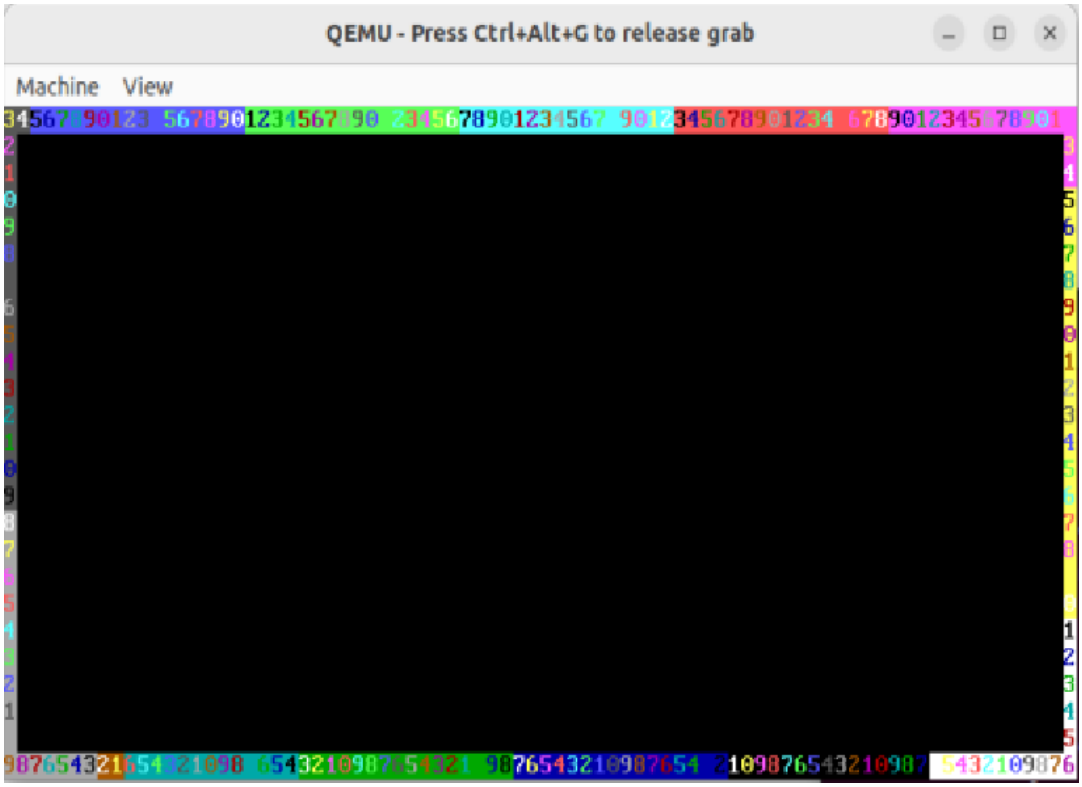
通过 debug，找出了最后的错误并修改，为了验证正确性，在 test.cpp 中手动输出了 a2、while_flag 和 random_buffer，均为正确值。



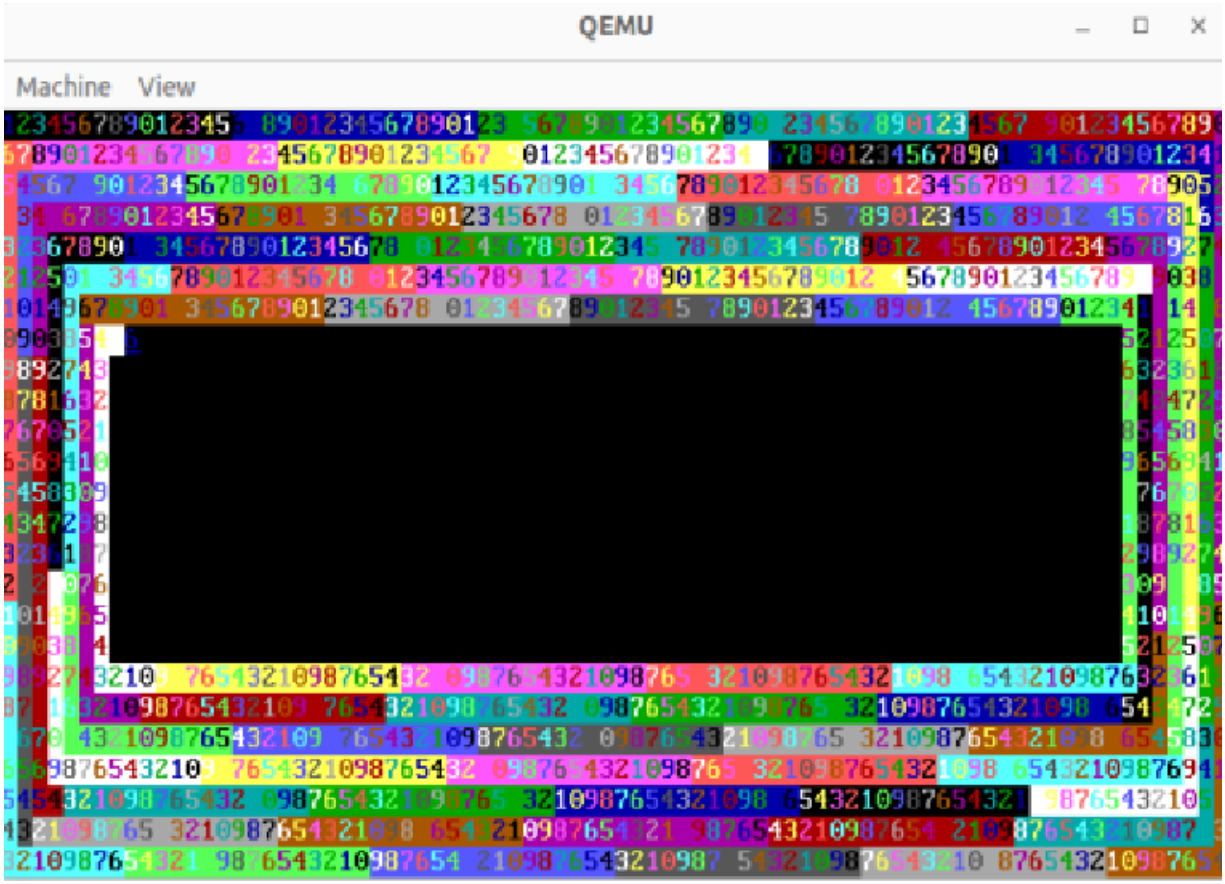
如图，your_function 函数输出的语句为:
”Mr.Chen, students and TAs are the best!”

3.4 实验任务四

3.4.1 字符在最外侧绕圈



3.4.2 字符不断向内绕圈



4 实验总结与心得体会

4.1

编写程序时往往不会一次就能运行成功，有时需要 qemu 配合 gdb 进行 debug，而 debug 的流程如下:
先把代码编译成二进制文件，再创建磁盘，最后把 MBR 写入磁盘首扇区。

```
nasm -f bin mbr.asm -o mbr.bin
qemu-img create hd.img 10m
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

接着生成符号表，启动 qemu，在另一个终端启动 gdb 进行调试:

```
nasm -o mbr.o -g -f elf32 mbr.asm
ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00
qemu-system-i386 -hda hd.img -s -S -parallel stdio -serial null
```

最后设置断点，就可以到指定位置单步执行程序，并查看寄存器的值了。
实验任务三与实验任务四的成功都归功于 gdb 的调试，才能发现微小的寄存器错误。

4.2

本次实验在 x86 汇编的基础下完成，编写 x86 汇编时有许多易错点，总结如下:

- 访问数据段变量的值需要用中括号，如 [a]

- mov 指令的两个操作数必须等长, 特别要区分一些 8 位，16 位的寄存器，以免在互相传递值的时候报错
- 在使用寄存器时需要检查其是否已经保存了一些重要的值，如果有的话需要先压栈保存再使用，以免原先的值被破坏
- 在使用 BIOS 中断时，有时也会隐式破坏了寄存器 ax、bx 等的值，需要细心考虑，压栈保存

5 参考资料清单

https://www.cnblogs.com/jiftle/p/8453106.html
https://zh.wikipedia.org/wiki/INT_10H