LAB2 实验报告

实验课程: 操作系统原理实验

任课教师: 刘宁

实验题目: 实模式和保护模式下的 OS 启动

专业名称: 计算机科学与技术

学生姓名: 马福泉

学生学号: 23336179

实验地点: 实验中心 B202

实验时间: 2025.3.12

Section 1 实验概述

在第二章中，需要掌握 x86 汇编、计算机的启动过程、IA-32 处理器架构和字符显存原理。并且根据所学的知识，自己编写程序，然后让计算机在启动后加载运行，以此增进对计算机启动过程的理解，为后面编写操作系统加载程序奠定基础。同时，需要掌握如何使用 gdb 来调试程序的基本方法。

Section 2 预备知识与实验环境

- 预备知识：汇编语言基础，计算机体系结构，操作系统基础，调试工具的使用（如 gdb）等。
- 实验环境：
  - 虚拟机版本/处理器型号：VMware-Ubuntu22.04.5/i386（32 位）
  - 代码编辑环境： 编辑器：VSCode
    插件：C/C++插件，汇编插件
  - 代码编译工具： 编译器：gcc
    工具链：binutils、make、qemu、nasm 等
    调试工具：gdb
  - 重要三方库信息：GNU binutils：工具集（如 as、ld）
    gdb：调试工具
    QEMU、Bochs：模拟器用于测试 等

Section 3 实验任务

- 实验任务 1：掌握 IA-32 处理器的架构以及汇编代码等相关知识
- 实验任务 2：操作系统的启动 Hello World

Section 4 实验步骤与实验结果

---------- 实验任务----------

- 任务要求：操作系统的启动 Hello World
- 思路分析：编写并加载一个简单的操作系统引导扇区（MBR），该引导扇区将被加载到计算机的内存中并在屏幕上输出“Hello World”

- 实验步骤：

1. 学习字符显示的原理。
2. 编写 MBR 代码：

使用汇编语言编写 MBR 代码，该代码将负责初始化显示，并在屏幕上输出“Hello World”。

MBR 代码：见文件 mbr.asm

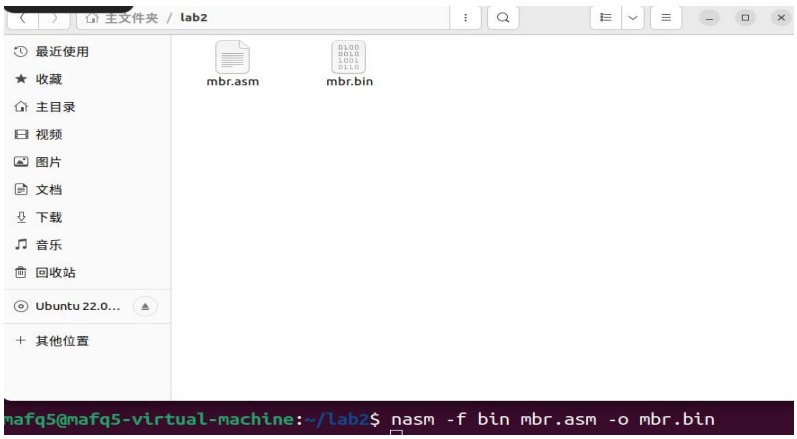
```
org 0x7c00
[bits 16]
xor ax, ax          ; eax = 0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

; 初始化栈指针
mov sp, 0x7c00
mov ax, 0xb800
mov gs, ax

; 显示 Hello World
mov ah, 0x01        ; 蓝色
mov al, 'H'
mov [gs:2 * 0], ax

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

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

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

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

mov al, ' '
mov [gs:2 * 5], ax

mov al, 'W'
```

```
    mov [gs:2 * 6], ax

    mov al, 'o'
    mov [gs:2 * 7], ax

    mov al, 'r'
    mov [gs:2 * 8], ax

    mov al, 'l'
    mov [gs:2 * 9], ax

    mov al, 'd'
    mov [gs:2 * 10], ax

    jmp $                   ; 死循环

    times 510 - ($ - $$) db 0
    db 0x55, 0xaa           ; MBR 标志
```

3. 编译 MBR：

使用 nasm 汇编器编译 MBR 代码为二进制文件,如图所示：

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



4. 创建虚拟硬盘并写入 MBR：

使用 qemu-img 创建一个虚拟硬盘：

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

使用 dd 命令将 MBR 写入虚拟硬盘的首扇区：

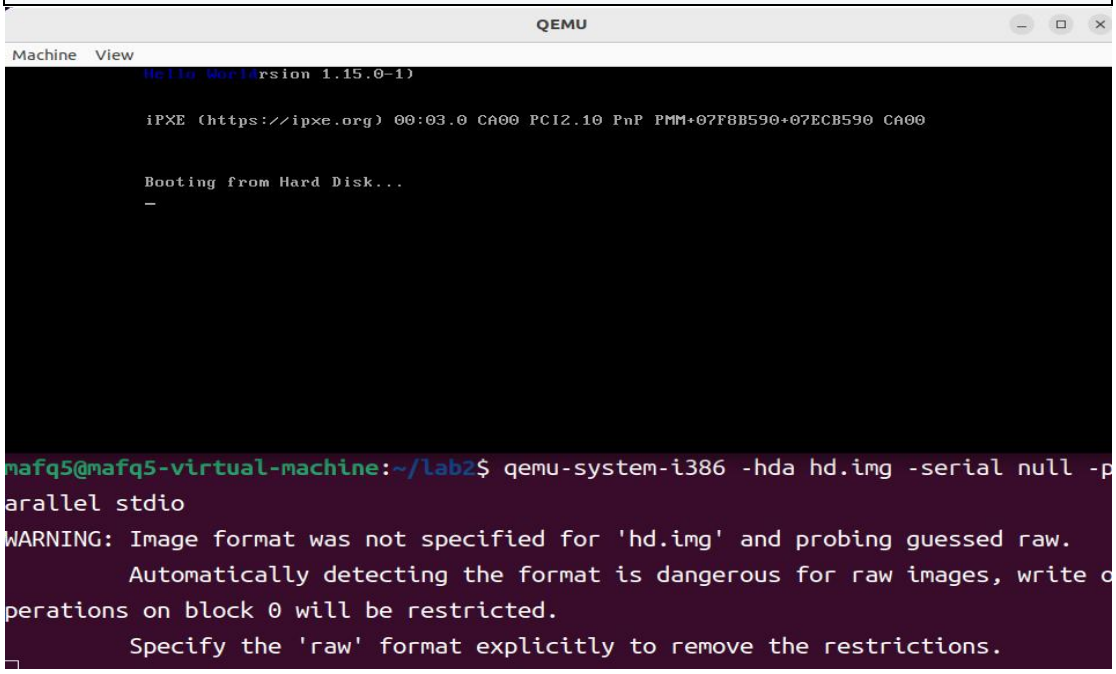
```
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

```
mafq5@mafq5-virtual-machine:~/lab2$ qemu-img create hd.img 10m
Formatting 'hd.img', fmt=raw size=10485760
mafq5@mafq5-virtual-machine:~/lab2$ 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字节已复制，0.000858354 s，596 kB/s
```

5. 启动 QEMU 模拟器：

使用 qemu-system-i386 启动 QEMU 模拟器，并加载虚拟硬盘：

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



6. 调试：gdb 常用 debug 指令

调试示例:

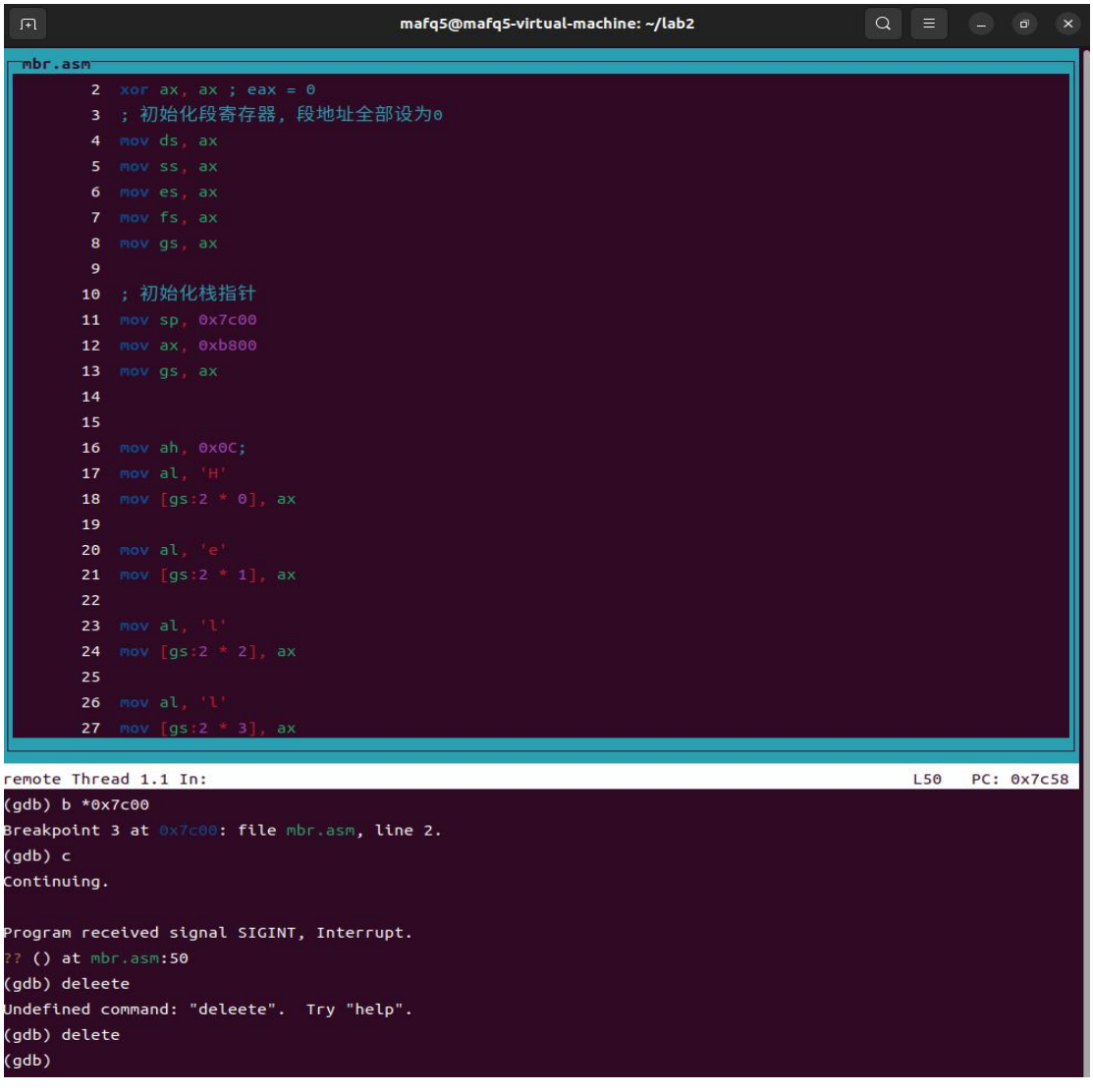
7.

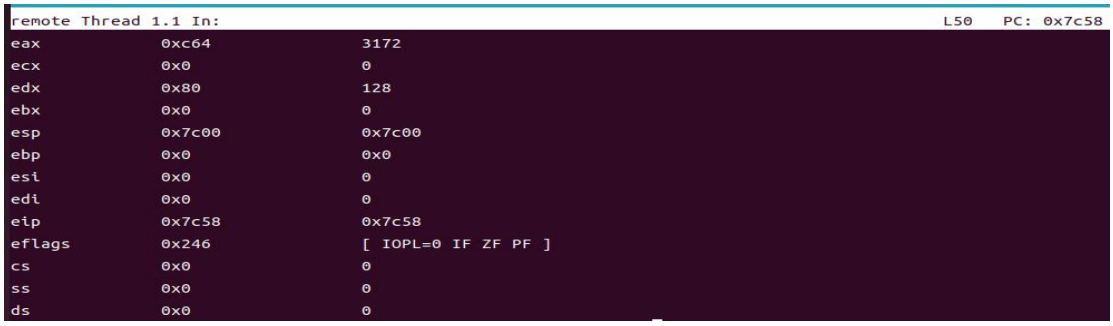
```
nasm -o mbr.o -g -f elf32 mbr.asm   # 编译汇编文件，生成符号表
ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00   # 链接目标文件并生成
符号表
qemu-system-i386 -hda hd.img -s -S -parallel stdio -serial null   # 启动
QEMU，等待调试
```

在另一终端打开 gdb:

```
gdb   # 启动 GDB 调试器
target remote :1234   # 连接到 QEMU 提供的调试端口
add-symbol-file mbr.symbol 0x7c00   # 加载符号表
layout src   # 显示源代码
b *0x7c00   # 设置断点，程序会在此停住
delete(num)  #删除断点
```

```
c   # 继续执行，直到断点处
info registers   # 查看寄存器状态
x/10i $pc   # 显示当前程序计数器位置的 10 条汇编指令
p *0x7c00   # 查看内存地址 0x7C00 的值
set disassembly-flavor intel   # 设置反汇编语法为 Intel 风格
```





```
remote Thread 1.1 In:                                                    L50    PC: 0x7c58
=> 0x7c58:      jmp    0x7c58
   0x7c5a:      add    %al,(%eax)
   0x7c5c:      add    %al,(%eax)
   0x7c5e:      add    %al,(%eax)
   0x7c60:      add    %al,(%eax)
   0x7c62:      add    %al,(%eax)
   0x7c64:      add    %al,(%eax)
   0x7c66:      add    %al,(%eax)
   0x7c68:      add    %al,(%eax)
   0x7c6a:      add    %al,(%eax)
(gdb) p *0x7c00
$1 = -661733327
(gdb) set disassembly-flavor intel
```

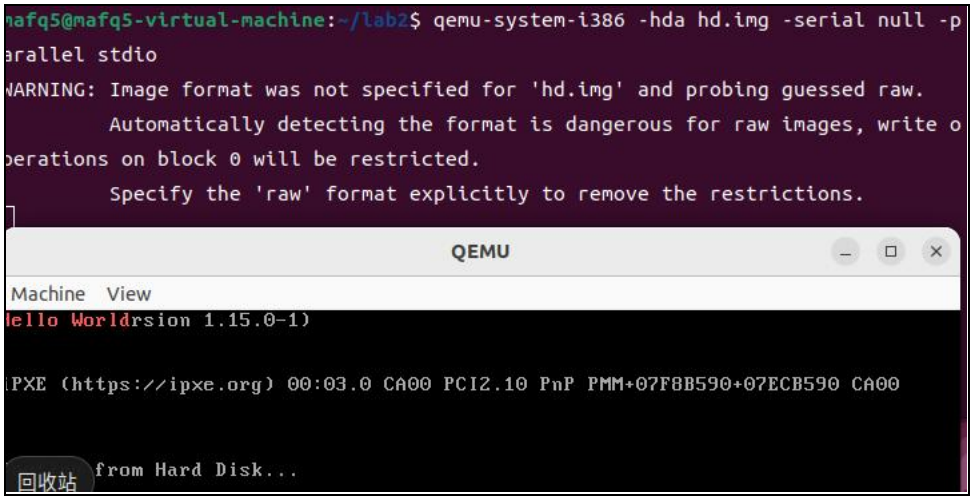
- 实验结果展示：

```
mov ah, 0x01  ; 0x0 = 黑色背景, 0x1 = 蓝色前景
```

蓝色不明显，我们改为亮红色，代码如下：

```
mov ah, 0x0C  ; 0x0 = 黑色背景, 0xC = 亮红色前景
```

得到结果如图：



Section 5 实验总结与课后习题

1. 在 C 程序中，通常会有一个 main() 函数作为程序的入口点。但在汇编语言中，尤其是在操作系统的引导程序（MBR）中，没有传统的 main 函数。汇编语言中，程序的入口点通常由 _start 符号标识，_start 是程序的入口地址。本实验没有在代码中显式定义 _start，所以链接器无法找到它，只能默认为 0x7C00。

```
mafq5@mafq5-virtual-machine:~/lab2$ ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext
0x7c00
ld: 警告: 无法找到项目符号 _start; 缺省为 0000000000007c00
```

2. MBR（Master Boot Record）被加载到内存地址 0x7C00 而不是 0x0000 的原因，主要与计算机的启动过程和内存管理有关：

（1）计算机的启动过程：当计算机加电后，CPU 会执行固化在 ROM 中的代码（BIOS），这段代码负责硬件初始化和引导过程。BIOS 执行的第一个

任务是从硬盘的第一个扇区读取数据，这个扇区包含了 MBR（通常是 512 字节），并将其加载到内存。

（2）在传统的 x86 体系结构中，BIOS 将 MBR 加载到内存的地址 0x7C00。这是因为：BIOS 通常是从 0xFFFF0 开始执行的，地址 0xFFFF0 是 CPU 复位时的默认执行地址。BIOS 执行完初始化后，会将控制权转交给 MBR，而将 MBR 加载到 0x7C00，这个地址紧跟在 BIOS 的代码区域后面。

3. 注意

（1） 在操作系统实验中，我们常常会看到一些约定俗成的规则。这些规则往往是无法通过逻辑推导出来。例如 IA-32 处理器启动先进入的是 16 位实模式，然后由实模式跳转到 32 位保护模式，而 arm 处理器一启动便可进入 32 或 64 位的寻址模式。但是，arm 处理器于 90 年代发布，晚于 IA-32 处理器 20 年。

（2） 特别注意，通用寄存器有如下特殊用法或者别名，本教程称之为约定俗成的规则。

eax 在乘法和除法指令中被自动使用，通常称之为扩展累加寄存器。

在 loop 指令中，ecx 默认为循环计数器。

esp 用于堆栈寻址。因此，我们绝对不可以随意使用 esp 来存放我们的数据。esp 被称为栈指针寄存器。

esi 被称为源指针。

edi 被称为目的指针寄存器。

ebp 通常在函数用来引用函数参数和局部变量。ebp 被称之为帧指针寄存器

（3） 我们指令中如果没有显式指定段地址，那么我们的代码中给出的地址是偏移地址。此时，CPU 在计算线性地址时用到了默认的段寄存器，规则如下：

访问数据段，使用段寄存器 ds。

访问代码段，使用段寄存器 cs。

访问栈段，使用段寄存器 ss。

当然，我们也可以使用“段地址:偏移地址”的形式指定段地址，此时 CPU 不使用默认段寄存器的段地址，而是使用指令给出的段地址。例如，我们可以

指定段地址为 es 段寄存器的内容。mov ax, [es:tag]

4. 课后习题

（1） 请你谈谈对多层语言模型的理解，即为什么需要有机器语言、汇编语言和高级语言三层？

机器语言是计算机能够直接理解和执行的二进制代码。它与硬件的工作紧密结合，效率高，但可读性差。

汇编语言是机器语言的符号化表示，使用助记符（如 MOV、ADD）表示指令，可以让程序员更加容易地理解和编写程序，但仍需与机器语言对应。

高级语言（如 C、Java）具有更高的抽象层次，使用自然语言或数学符号表达程序逻辑，开发效率高，但需要通过编译器转换为机器语言。

（2） 请你描述下 IA-32 处理器的种类和用法，例如 eax 又可以分为哪几个寄存器来访问？esp 的用途是什么？

EAX 是一个 32 位通用寄存器，可以分为几个部分进行访问： AX（低 16 位）AH（高 8 位）AL（低 8 位）

ESP 寄存器： ESP 是堆栈指针寄存器，指示当前栈顶的位置。

（3） 请查阅相关资料，说说 eflags 的各个位有什么含义？

CF：进位标志（Carry Flag）表示运算是否产生进位或借位。

ZF：零标志（Zero Flag）表示运算结果是否为零。

SF：符号标志（Sign Flag）表示结果的符号（负或正）。

OF：溢出标志（Overflow Flag）表示有符号运算是否发生溢出。

PF：奇偶标志（Parity Flag）表示运算结果的最低字节中 1 的个数的奇偶性。

AF：辅助进位标志（Auxiliary Carry Flag），用于 BCD 运算。

（4） 什么是线性地址？实模式的寻址模式是什么？地址空间大小如何？

线性地址：是指计算机系统中的虚拟地址或物理地址，在段选择符与偏移量相加后产生的地址。

实模式寻址：在实模式下，CPU 只能访问 1MB 的内存，使用段寄存器和偏移量进行寻址。

地址空间大小：在实模式下，地址空间为 1MB（20 位地址总线）。

（5） nasm 汇编中的内存寻址方式有哪些？语法是什么？请分别描述。

直接寻址：使用具体的内存地址，如 MOV AX, [1234h]。

寄存器寻址：操作数在寄存器中，如 MOV AX, BX。

间接寻址：通过寄存器间接引用内存，如 MOV AX, [BX]。

基址加变址寻址：通过基址和变址寄存器计算内存地址，如 MOV AX, [BX+SI]。

（6） 在什么情况下会使用默认寄存器 cs，ds，ss？如何避免 CPU 在计算线性地址时使用默认寄存器？

CS：代码段寄存器，指示当前代码段的位置，通常在执行代码时自动使用。

DS：数据段寄存器，默认用于数据段的访问。

SS：堆栈段寄存器，默认用于堆栈操作。

为了避免默认寄存器的使用：可以使用显式的段寄存器，如 MOV AX, [DS:BX]，指定具体的段寄存器来避免使用默认值。

（7） 我们已经知道，push ax 等价于下面的语句。

```
; 下面语句等价于 push ax
sub sp, 2          ; 从高地址向低地址增长，16 位实模式
mov [sp], ax
```

请问为什么不是等价于下面的语句？

```
; 下面语句等价于 push ax
mov [sp], ax
sub sp, 2      ; 从高地址向低地址增长, 16 位实模式
```

为什么不是等价于下面的语句？

```
; 下面语句等价于 push ax
sub sp, 2      ; 从高地址向低地址增长, 16 位实模式
mov sp, ax
```

第一个代码是正确的等价操作，先调整栈指针 (sub sp, 2)，再将数据存入栈中 (mov [sp], ax)。

第二个代码错误，先存数据再调整栈指针，导致数据存储在错误的地址。

第三个代码错误，mov sp, ax 会改变栈指针的值，而不是向栈中压入数据。

（8） jmp 指令和 call 指令有什么不同之处？

jmp 指令：无条件跳转到指定的地址，不保存返回地址，程序会跳转到目标地址继续执行。

call 指令：跳转到指定的子程序，并将当前指令的地址（返回地址）压入栈中，允许执行完子程序后通过 ret 指令返回到该地址。

（9） 请写出下列伪代码对应的汇编语句。其中，ax，bx，cx 是寄存器。

```
if ax == 16 then
    bx = 17
```

```
else
    cx = bx
```

```
cmp ax, 16      ; 比较 ax 与 16
je  equal       ; 如果相等, 跳转到 equal 标签
mov cx, bx      ; 否则, 将 bx 的值赋给 cx
jmp end_if      ; 跳转到 end_if

equal:
mov bx, 17      ; 如果 ax == 16, bx = 17

end_if:
```

（10） 请写出 pushad、popad 对应的汇编语句。

pushad：将所有通用寄存器（EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP）的值压入栈中。

```
push eax
push ebx
push ecx
push edx
push esi
push edi
push ebp
push esp
```

popad：从栈中恢复所有通用寄存器的值。

```
pop esp
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
pop eax。
```

（11） 下面的代码是否有错？如有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
; $在 nasm 汇编中表示当前地址, 即 jmp $指令开始的地址
; 因此这条语句实际上是在做死循环, 程序到这里就不会往下执行了

my_function:
```

```
    push ax
    push bx

    sub ax, 10
    sub bx, 10
    ; 后进先出
    pop ax

    ret
```

错误分析：pop 顺序错误，ax 寄存器的值会被错误地修改

在 add ax, 10 后，程序会执行死循环 jmp $。因为 $ 表示当前位置，导致程序停在这条指令，无法继续执行。

最终结果：程序进入死循环。

（12） 下面的代码是否有错，有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
; $在 nasm 汇编中表示当前地址，即 jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

my_function:
    push bx
    push ax

    sub ax, 10
    sub bx, 10
    ; 后进先出
    pop bx
    pop ax

    ret
```

错误分析：pop 顺序错误，ax 寄存器的值会被错误地修改

在 add ax, 10 后，程序会执行死循环 jmp $。因为 $ 表示当前位置，导致程序停在这条指令，无法继续执行。

最终结果：程序进入死循环。

（13） 下面的代码是否有错，有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
```

```
; $在 nasm 汇编中表示当前地址，即 jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

my_function:
    push bx
    push ax

    sub ax, 10
    sub bx, 10
    ; 后进先出
    pop ax
    pop bx

    ret
```

错误分析：pop 顺序正确。

但是在 add ax, 10 后，程序会执行死循环 jmp $。因为 $ 表示当前位置，导致程序停在这条指令，无法继续执行。

最终结果：程序进入死循环。

（14） 代码如下：

初始化：设置段寄存器和堆栈指针，确保程序运行在正确的内存环境中。设置 gs 段寄存器指向显存地址 0xb800（彩色文本模式）。

主循环（proc）：从 counter 中获取当前数字，并将其转换为 ASCII 字符（通过加上 '0' 的 ASCII 码）。更新数字的颜色（通过改变 digit_color 的值）。跳转到 count_address 标签，计算字符在屏幕上的位置。

位置计算（count_address）：根据 x_direction_flag 和 y_direction_flag 的值，控制字符在水平和垂直方向上的移动。如果字符到达屏幕边界（max_x 或 max_y），改变移动方向（通过修改 x_direction_flag 和 y_direction_flag 的值）。

显示字符（proc1）：计算字符在显存中的实际地址。将字符及其颜色写入显存，从而在屏幕上显示字符。

代码：见文件 bouncing.asm

```
org 0x7c00
[bits 16]

section .data
    counter db 9
    digit_string db 48
```

```
    digit_color dw 00000010b
    cursor_x dw 2
    cursor_y dw 0
    max_x dw 80*2-2
    max_y dw 25*2-3
    x_direction_flag db 1
    y_direction_flag db 1
    short_delay dw 0x100

_start:
    xor ax, ax
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov sp, 0x7c00
    mov ax, 0xb800
    mov gs, ax

proc:
    mov al, [counter]
    add al, '0'                         ;用 ASCII
    mov byte [digit_string], al

    add word [digit_color], 00100010b ;换颜色

    jmp count_address

count_address:
    cmp byte [x_direction_flag], 1
    je add_value_x
    jmp sub_value_x
add_value_x:
    inc word [cursor_x]
    inc word [cursor_x]
    mov ax, [cursor_x]
    cmp ax, [max_x]
    jle y_count
    mov byte [x_direction_flag], 0
    jmp y_count
sub_value_x:
    dec word [cursor_x]
```

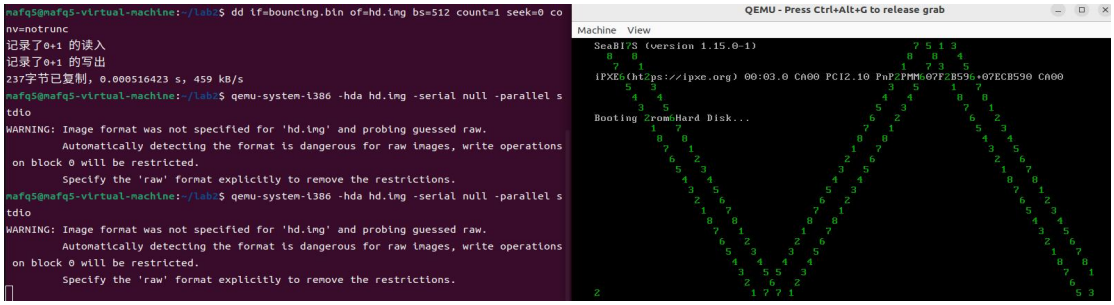
```
    dec word [cursor_x]
    cmp word [cursor_x], 2
    jge y_count
    mov byte [x_direction_flag], 1
    jmp y_count
y_count:
    cmp byte [y_direction_flag], 1
    je add_value_y
    jmp sub_value_y
add_value_y:
    inc word [cursor_y]
    inc word [cursor_y]
    mov ax, [cursor_y]
    cmp ax, [max_y]
    jle proc1
    mov byte [y_direction_flag], 0
    jmp proc1
sub_value_y:
    dec word [cursor_y]
    dec word [cursor_y]
    cmp word [cursor_y], 0
    jge proc1
    mov byte [y_direction_flag], 1
    jmp proc1

proc1:
    mov eax, 0
    mov ebx, 0
    mov ax, [cursor_y]
    mov bx, 80
    mul bx
    add ax, [cursor_x]
    mov bx, ax

    mov eax, 0
    mov ax, bx
    mov dh, [digit_color]
    mov dl, [digit_string]
    mov WORD [gs:eax], dx

    ; 时延
    mov cx, short_delay
delay_loop:
    nop
```

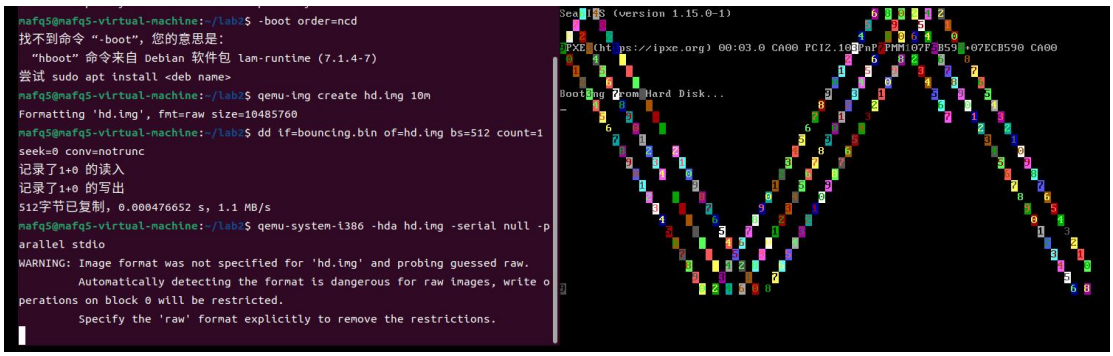
```
        mov bx, short_delay
    loop2:
        dec bx
        cmp bx, 0
        nop
        jne loop2
        nop
        loop delay_loop

        dec byte [counter]
        cmp byte [counter], 0
        je reset_counter
        jmp proc

    reset_counter:
        mov byte [counter], 9     ; Reset counter to 9
        jmp proc
```

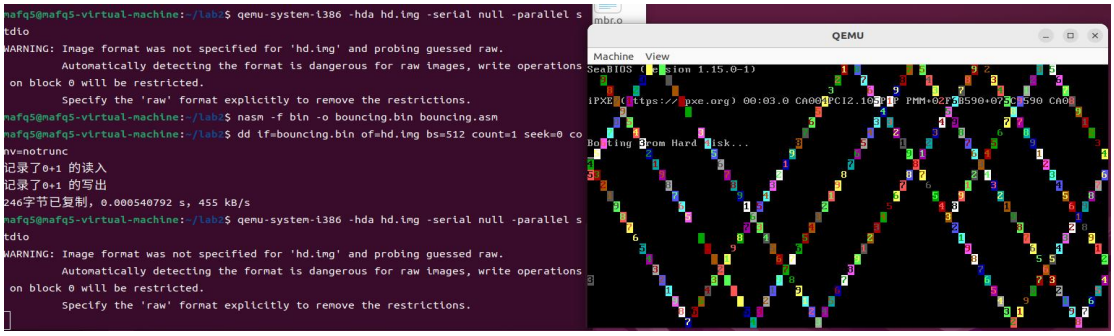
结果如下：

图一实现倒数，图二实现颜色变换，

图三将两者整合并且调整了边距（所以看着不大一样）







（16）实模式中断

（1） 请探索实模式下的光标中断，利用中断实现光标的位置获取和光标的移动，实现代码如下：

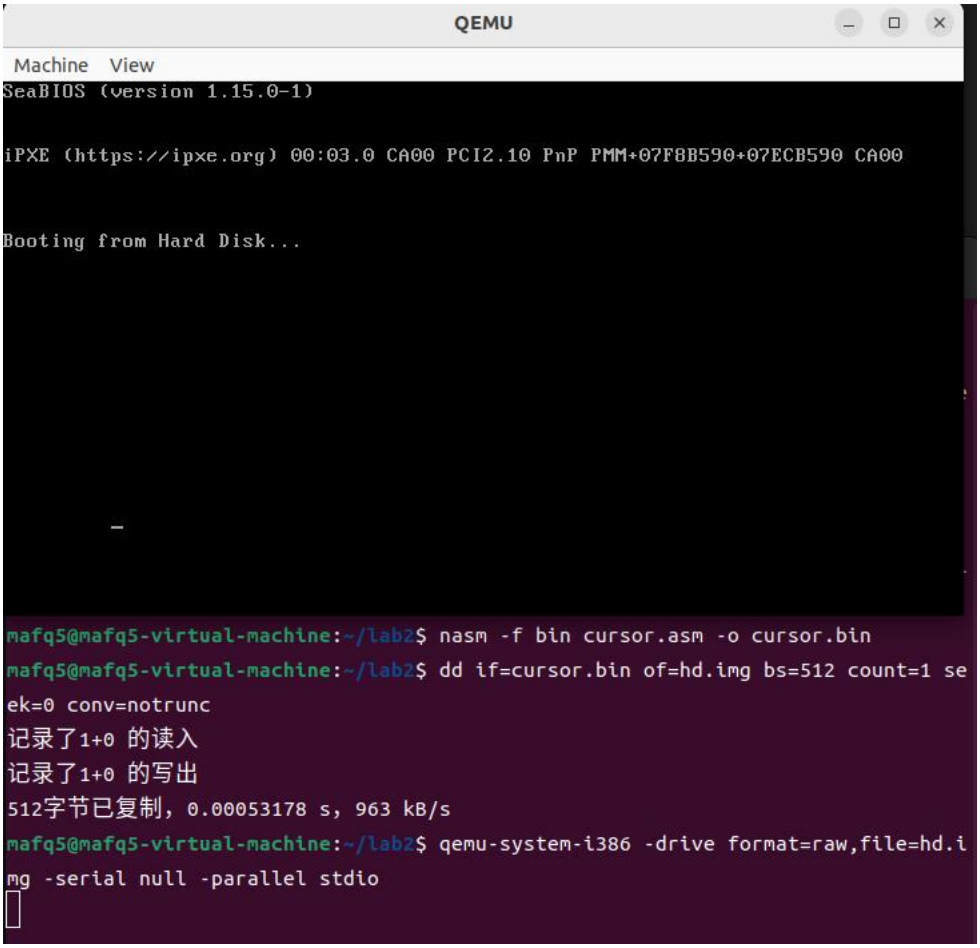
首先通过 xor ax, ax 和 mov 指令将所有段寄存器初始化为 0，。接着设置栈指针 sp 到 0x7c00。通过 int 10h 中断调用，使用 ah=02h 设置光标到第 20 行第 10 列，再用 ah=03h 获取光标位置，并将行 dh 和列 dl 存储到内存 0x5000 和 0x5001。最后通过 jmp $进入死循环。

代码：见文件 cursor.asm

```
    org 0x7c00
    [bits 16]

    ; 初始化
    xor ax, ax
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov sp, 0x7c00

    ; 设置光标位置到第 20 行，第 10 列
    mov ah, 02h
    mov bh, 00h
    mov dh, 14h
    mov dl, 09h
    int 10h

    ; 获取光标位置
    mov ah, 03h
    mov bx, 00h
    int 10h
```

```
; 将光标位置存储到内存中
mov [0x5000], dh
mov [0x5001], dl

jmp $ ; 死循环

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

根据代码，光标出现在第 20 行，第 10 列，如图所示：



（2） 修改 Hello World 代码，**使用实模式下的中断来输出学号：23336179**。代码如下：

初始化段寄存器：通过 xor ax, ax 清零 ax 寄存器，然后将 ax 的值分别赋给 ds 和 es，初始化数据段和附加段寄存器。

设置显示模式：通过 int 10h 中断调用，设置显示模式为文本模式。

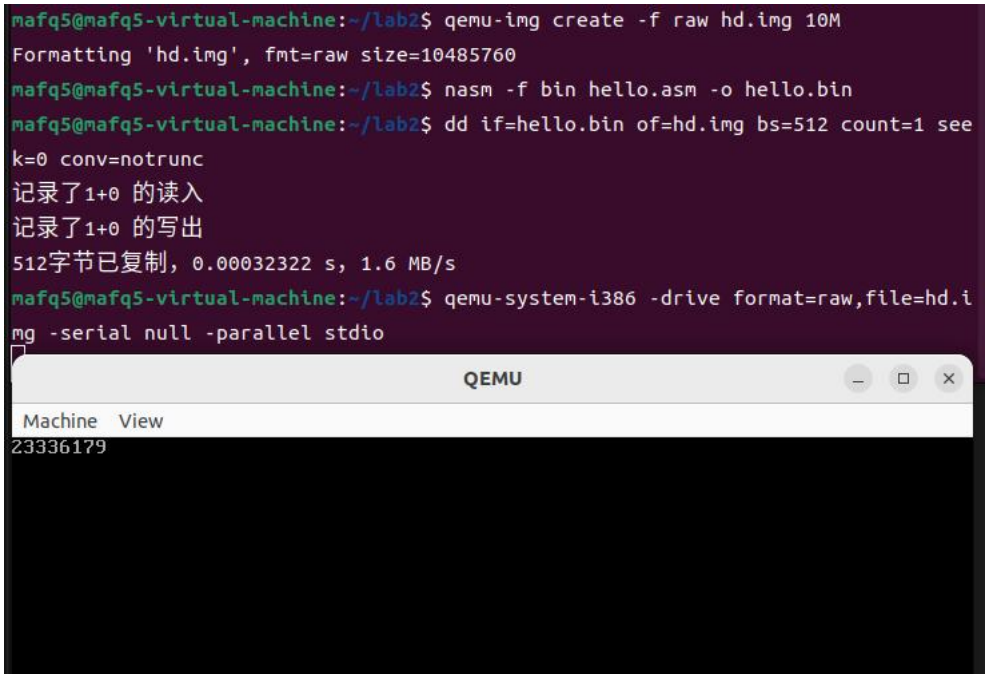
输出学号：使用 lodsb 指令逐字节加载字符串 student_id 中的字符到 al，

通过 int 10h（ah=0Eh）逐字符输出到屏幕，直到遇到字符串结尾的 0。

死循环：在输出完成后，程序跳转到自身地址，进入死循环。

代码：文件 hello.asm

```
org 0x7c00
[bits 16]

; 初始化
xor ax, ax
mov ds, ax
mov es, ax

; 设置显示模式为文本模式
mov ah, 00h
mov al, 03h
int 10h

; 逐字符输出学号
mov si, student_id
print_loop:
    lodsb
    or al, al
    jz done

    mov ah, 0x0E
    int 10h
    jmp print_loop  ; 继续处理下一个字符

done:
    jmp $
student_id db "23336179", 0  ; 学号字符串以 0 结尾

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



（3） 探索实模式的键盘中断，利用键盘中断实现键盘输入并回显

初始化段寄存器：确保所有段寄存器都初始化为 0。

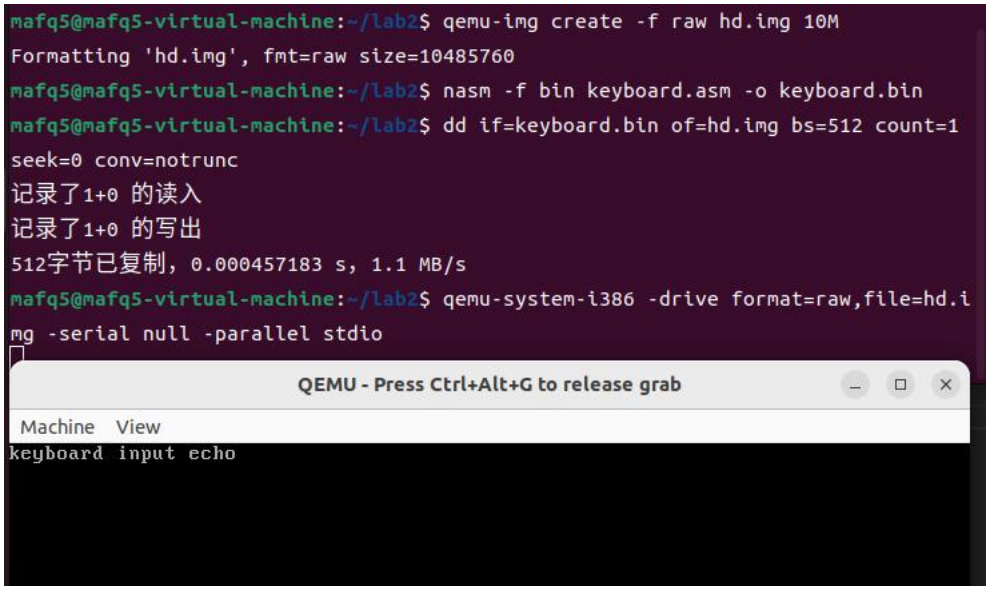
初始化栈指针：将栈指针 sp 设置为 0x7c00，指向引导扇区的起始位置。设置显示模式：通过 int 10h 中断调用，设置显示模式为文本模式（ah=00h，al=03h），这是标准的 80×25 字符模式。

主循环：使用 int 16h（ah=00h）等待键盘输入，该中断会返回按键的 ASCII 码到 al。使用 int 10h（ah=0Eh）将输入的字符回显到屏幕，并自动移动光标。然后跳回主循环的开始，继续等待下一个按键输入。

代码：文件 keyboard.asm

```
org 0x7c00
[bits 16]

; 初始化
xor ax, ax
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

mov sp, 0x7c00

mov ah, 00h
```

```
    mov al, 03h
    int 10h

    ; 主循环:等待键盘输入并回显
    main_loop:
        mov ah, 00h
        int 16h

        mov ah, 0Eh
        int 10h      ; 显示到屏幕, 并自动移动光标

        jmp main_loop

    jmp $ ; 死循环

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



（17）汇编代码的编写

三部分伪代码的汇编代码如下：

全部代码见 assignment 文件夹，其中 student.asm:

```
    section .data
    section .text
        global student_function
        global your_function

        extern a1
        extern a2
```

```
    extern my_random
    extern your_string
    extern if_flag
    extern while_flag
    extern print_a_char

student_function:
    pushad

    ; 分支逻辑实现
your_if:
    mov eax, [a1]
    cmp eax, 12
    jl if_case_1

    cmp eax, 24
    jl if_case_2

    ; if_flag = a1 << 4
    shl eax, 4
    mov [if_flag], eax
    jmp if_end

if_case_1:
    ; if_flag = a1 / 2 + 1
    mov eax, [a1]
    shr eax, 1          ; a1 / 2
    add eax, 1          ; +1
    mov [if_flag], eax
    jmp if_end

if_case_2:
    ; if_flag = (24 - a1) * a1
    mov eax, 24
    sub eax, [a1]
    mov ebx, [a1]
    mul ebx
    mov [if_flag], eax

if_end:
    ; 循环逻辑实现
your_while:
    ; while a2 >= 12
    mov eax, [a2]
```

```
        cmp eax, 12
        jl while_end

        call my_random

        mov edi, [while_flag]
        mov ebx, [a2]
        sub ebx, 12

        mov byte [edi + ebx], al
        dec dword [a2]
        jmp your_while

    while_end:

        popad
        Ret

    ; 函数实现:遍历字符数组 string
    your_function:
        pushad
        mov esi, [your_string]
    your_function_loop:
        mov al, byte [esi]
        test al, al
        jz your_function_end

        movzx eax, al
        push eax
        call print_a_char
        add esp, 4              ;

        inc esi
        jmp your_function_loop

    your_function_end:
        popad
        ret
```

Assignment 其他文件不变，成功运行截图如下：

```
mafq5@mafq5-virtual-machine:~/lab2/assignment$ make run
/usr/bin/ld: student.o: warning: relocation against `while_flag' in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
>>> begin test
>>> if test pass!
>>> while test pass!
Mr.Chen, students and TAs are the best!
```

Section 6 附录：参考资料清单

1. 指令以及代码 debug 除了参考实验指导，部分还参考了 ai 大模型。

   如 14 题代码修改扩展，16 题部分代码 debug.

2. 学习参考了文章（部分）

为什么 bios 将 mbr 装载到 0x7c00 地址 - jinzi - 博客园
0x7c00 的历史根源-电子工程专辑
键盘 I/O 中断调用（INT 16H）和常见的 int 17H、int 1A H-CSDN 博客
键盘扫描码表(keyboard scan code) 举头三尺你不懂 新浪博客