



# 本科生实验报告

实验课程：\_\_\_\_操作系统原理实验\_\_\_\_

实验名称：\_\_\_\_实验入门\_\_\_\_

专业名称：\_\_\_\_计算机科学与技术\_\_\_\_

学生姓名：\_\_\_\_熊彦钧\_\_\_\_

学生学号：\_\_\_\_23336266\_\_\_\_

实验地点：\_\_\_\_实验楼 B203\_\_\_\_

实验成绩：\_\_\_\_

报告时间：\_\_\_\_2023 年 3 月 13 ‘’ 日\_\_\_\_

## Section 1 实验概述

- 1. 学习 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；
- 2. 根据所学的知识，尝试自己编写程序，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；
- 3. 学习如何使用 gdb 来调试程序的基本方法。

## Section 2 实验步骤与实验结果

### ----- 实验任务 1（对应 assignment1.1） -----

- 任务要求： 复现 example 1（操作系统启动并输出 ‘hello world’）
- 实验步骤：

1.了解基础知识：详情请见实验指导，比较重要的是了解字符的前景色和背景色设置以及各个矩阵点坐标如何换算为显存起始位置

$$\text{显存起始位置} = 0xB8000 + 2 \cdot (80 \cdot x + y)$$

2.编写 MBR：用 VScode 新建.asm 文件并编写 MBR。经实践检验，`mov`  
`ah, 0x03` 才是青色字符，因此对代码作出修改。部分代码截图如下（完整代码请见附件）：

```
mov ah, 0x03 ;青色
mov al, 'H'
mov [gs:2 * 0], ax

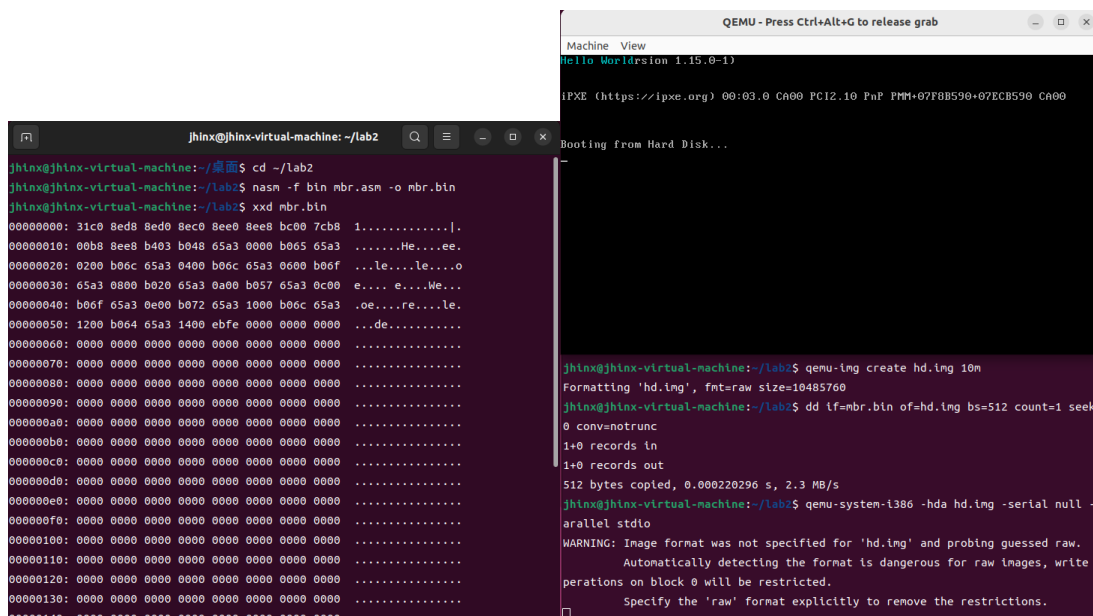
mov al, 'e'
mov [gs:2 * 1], ax
```

编写好 MBR 后，即可进行下一步骤。

3.启动程序：所需的命令行代码如下：

```
nasm -f bin mbr.asm -o mbr.bin（转为二进制文件）
xxd mbr.bin（查看文件内容）
qemu-img create hd.img 10m（创建大小为 10m 的磁盘）
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc（写入磁盘）
qemu-system-i386 -hda hd.img -serial null -parallel stdio
```

输入命令行后，系统会自动打开 qemu 显示屏并且输出 ‘hello world’，输出截图如下：



经观察，程序成功在 qemu 中输出了青色的‘hello world’，实验任务完成。

## ----- 实验任务 2（对应 assignment1.2） -----

- 任务要求：修改 example 1 的代码，使得 MBR 被加载到 0x7C00 后在(16,10)处开始输出你的学号。学号的前景色和背景色须和教程的不同。
- 实验步骤：实验步骤与实验任务 1 一致

1.编写 MBR：在编写之前，我们需要计算（16,10）在 qemu 中的位置，由于 qemu 显示屏的大小为 80\*25，因此（16,10）应该对应  $16*80+10=1290$ 。

查看实验指导的表格后，我们可以知道红色背景，绿色前景对应的是 0x42。 .asm 文件的部分代码截图如下：

```

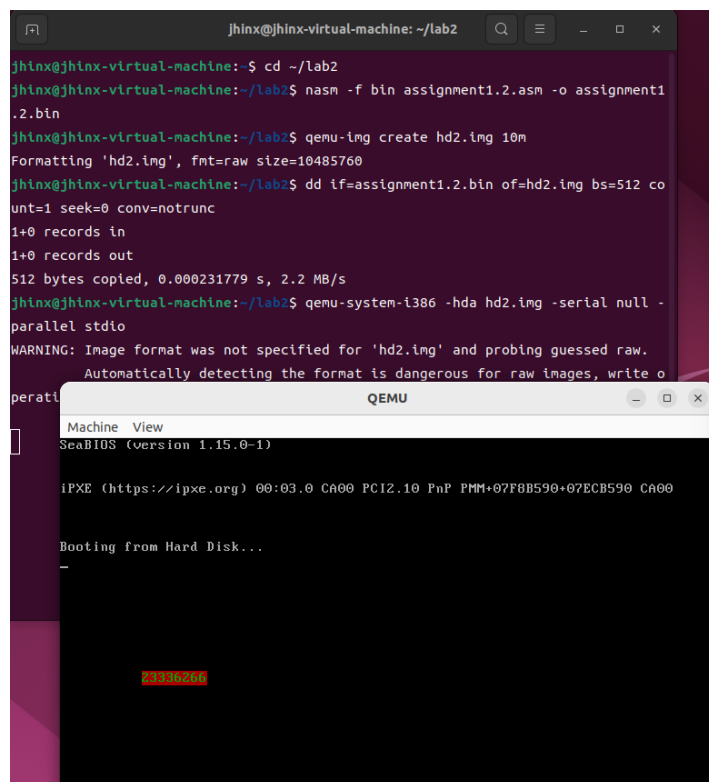
17  mov ah, 0x42 ; 红色背景, 绿色前景
18  mov al, '2'
19  mov [gs:2 * 1290], ax
20  |
21  mov al, '3'
22  mov [gs:2 * 1291], ax
23
24  mov al, '3'
25  mov [gs:2 * 1292], ax
26
27  mov al, '3'
28  mov [gs:2 * 1293], ax
29
30  mov al, '6'
31  mov [gs:2 * 1294], ax
32
33  mov al, '2'
34  mov [gs:2 * 1295], ax
35

```

2.启动程序：运行的代码需要将 1.1 的代码中 mbr.asm 改为 assignment1.2.asm（本人以此来命名文件），把 mbr.bin 改为 assignment1.2.bin，把 hd.img 改为 hd2.img。如果不修改，将会因为文件夹中已存在相应的文件而新建失败，或者会

产生覆盖，导致 assignment1.1 的文件丢失。由于代码比较简单，因此不在此处展示。

代码运行的截图如下：



The screenshot shows a terminal window with the following commands and output:

```
jhinx@jhinx-virtual-machine: ~/lab2
jhinx@jhinx-virtual-machine:~$ cd ~/lab2
jhinx@jhinx-virtual-machine:~/lab2$ nasm -f bin assignment1.2.asm -o assignment1.2.bin
jhinx@jhinx-virtual-machine:~/lab2$ qemu-img create hd2.img 10m
Formatting 'hd2.img', fmt=raw size=10485760
jhinx@jhinx-virtual-machine:~/lab2$ dd if=assignment1.2.bin of=hd2.img bs=512 count=1 seek=0 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000231779 s, 2.2 MB/s
jhinx@jhinx-virtual-machine:~/lab2$ qemu-system-i386 -hda hd2.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd2.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations may be unsafe.
See https://qemu.org/wiki/faq/raw for more details.

```

Below the terminal window, the QEMU window is visible, showing the SeaBIOS (version 1.15.0-1) boot screen. The screen displays the IPXE (https://ipxe.org) boot loader and the message "Booting from Hard Disk...". At the bottom of the screen, the number "23336266" is displayed in red on a green background.

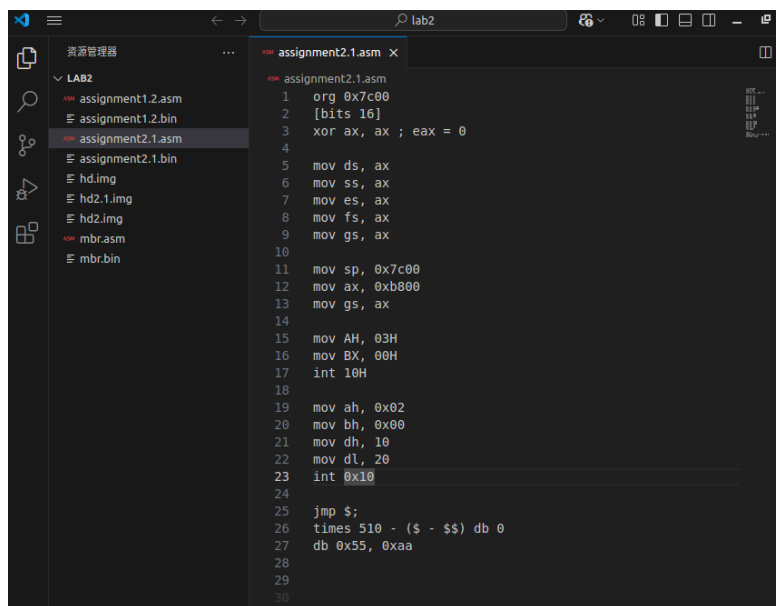
经观察，成功在 qemu 显示屏的（16,10）处开始输出红色背景，绿色前景的‘23336266’，实验任务完成。

### ----- 实验任务 3 （对应 assignment2.1） -----

- 任务要求：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。说说你是怎么做的，并将结果截图。

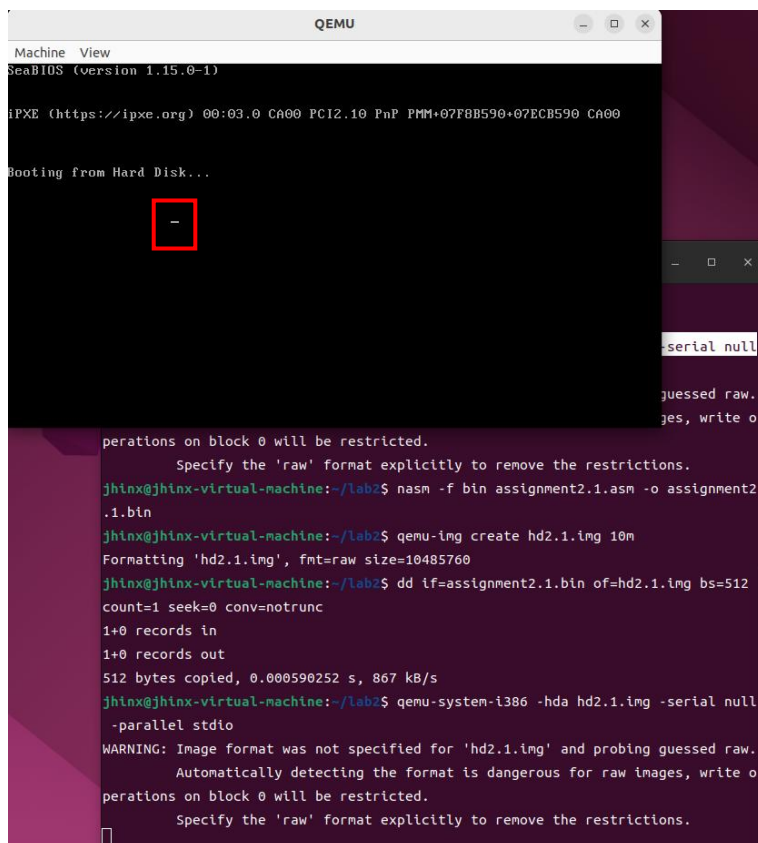
- 实验步骤：本实验的流程和实验任务 2 一致，区别只是.asm 文件的具体代码

**1.编写 MBR：**（1）光标位置的获取，需要用到功能号 AH=03H，而光标的移动，也就是改变光标的位置，则需用到功能号 AH=02H。文件代码截图如下：（这一代码将光标移动到了坐标（10,20）处，下面还会有另一个代码作为对照）



```
1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 mov sp, 0x7c00
12 mov ax, 0xb800
13 mov gs, ax
14
15 mov AH, 03H
16 mov BX, 00H
17 int 10H
18
19 mov ah, 0x02
20 mov bh, 0x00
21 mov dh, 10
22 mov dl, 20
23 int 0x10
24
25 jmp $;
26 times 510 - ($ - $$) db 0
27 db 0x55, 0xaa
28
29
30
```

2.运行程序：采用和上面实验任务同样的方法，这一代码运行后 qemu 显示屏的结果如下：



```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

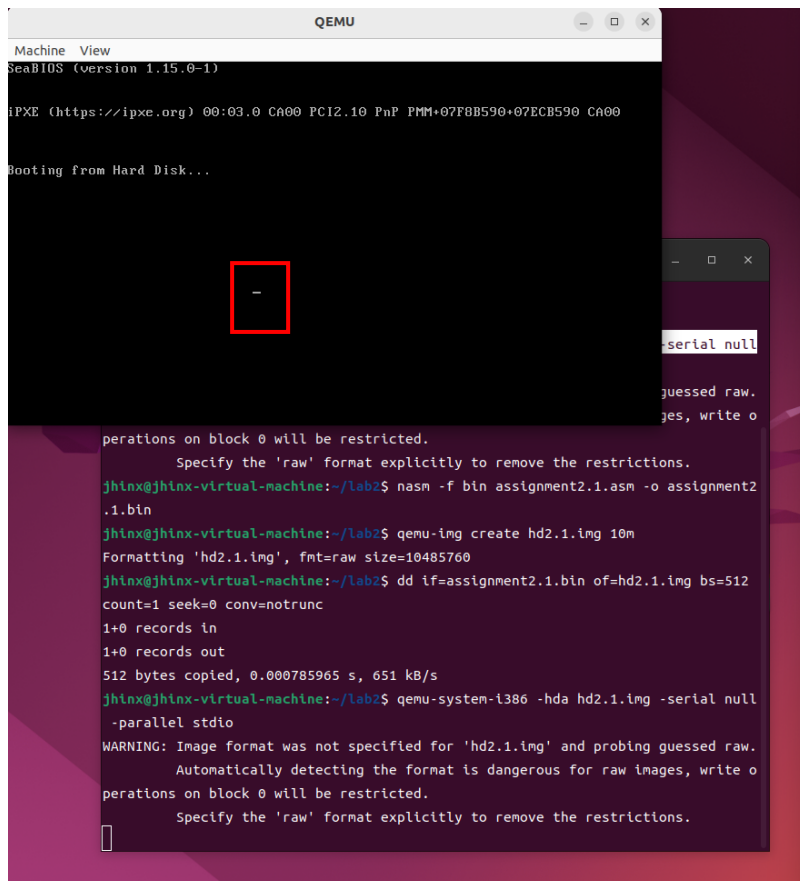
Booting from Hard Disk...
-

serial null

guessed raw.
ges, write o

perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
jhinx@jhinx-virtual-machine:~/lab2$ nasm -f bin assignment2.1.asm -o assignment2.1.bin
jhinx@jhinx-virtual-machine:~/lab2$ qemu-img create hd2.1.img 10m
Formatting 'hd2.1.img', fmt=raw size=10485760
jhinx@jhinx-virtual-machine:~/lab2$ dd if=assignment2.1.bin of=hd2.1.img bs=512 count=1 seek=0 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000590252 s, 867 kB/s
jhinx@jhinx-virtual-machine:~/lab2$ qemu-system-i386 -hda hd2.1.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd2.1.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

如图，图中红色边框处即为被移动位置后的光标。  
作为对照，我还做了另一组实验，把光标移动的位置修改为了（15,30），运行后 qemu 显示屏输出结果如下：



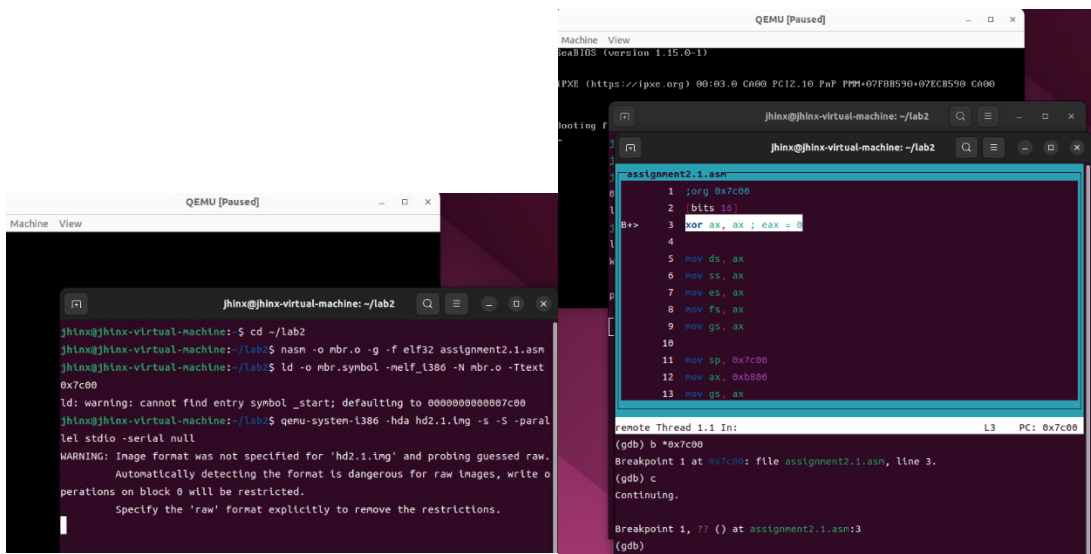
如图，图中红色方框即为移动后的光标，与（10,20）时的坐标位置有明显差异。

但是，此时仍然不能说明实验任务完成，因为 **qemu** 显示屏只能显示光标移动后的位置，并未得知获取光标位置这一任务的结果，而且也没有光标坐标的详细信息，因此，我们还需要进行以下步骤，以得到量化的结果。

**3.利用 gdb 进行 debug:** 在 debug 之前，需要使用命令行进行前置工作，所需的命令行代码如下：

```
cd ~/lab2/assignment2（进入程序所在的文件夹）
nasm -o assignment2.1.o -g -f elf32 assignment2.1.asm
ld -o mbr.symbol -melf_i386 -N assignment2.1.o -Ttext 0x7c00 （生成符号表）
cd ~/lab2/assignment2（在另一个 Terminal 进入 gdb）
gdb
target remote:1234（使用 gdb 连接 qemu）
add-symbol-file mbr.symbol 0x7c00（加载符号表）
layout src（打开显示窗口）
b *0x7c00（在此处设置断点）
c（执行到断点）
```

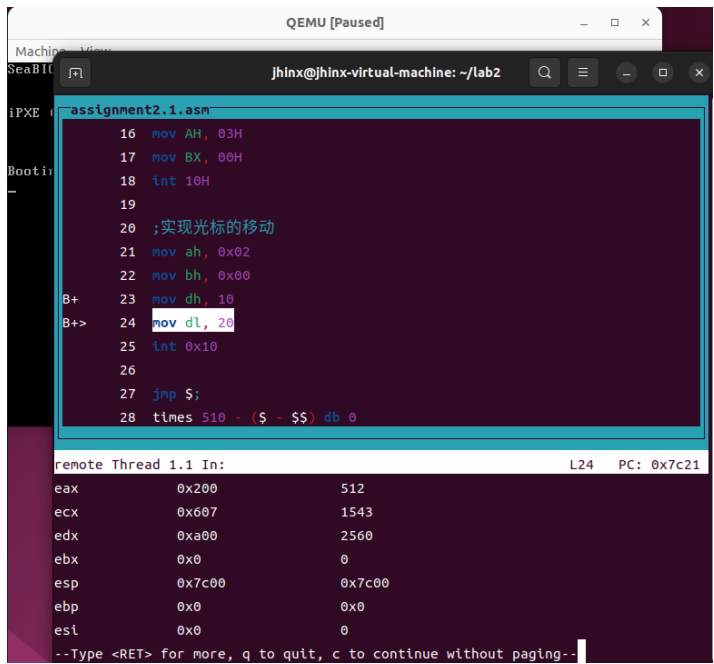
这一部分的运行截图如下：



随后，就可以使用 gdb 的命令代码进行调试了。在本次调试中，我们需要做的是对特点的行设置断点，并且观察寄存器的数值。

在调试的过程中，我们可以通过键盘的 ‘↑ ↓’ 键查看整个程序，然后利用代码 `break 行数` 来为特定的行设置断点，比每次都使用内存地址要方便很多。

设置断点后，我们可以使用代码 `info registers` 来查看断点处的寄存器数值。



如图可知，edx 此时的数值为 2560=2\*(15\*80+30)，因此前面设置光标位置成功。本实验任务完成。

## ----- 实验任务 4（对应 assignment2.2） -----

- 任务要求：请修改 1.2 的代码，使用实模式下的中断来输出你的学号。说说你是怎么做的，并将结果截图。

- 实验步骤：

1.编写 MBR：部分代码截图如下

```
;将光标移动到 (15,30) 处
mov ah, 0x02
mov bh, 0x00
mov dh, 15
mov dl, 30
int 0x10

;利用中断实现学号的输出
mov ah, 0x09
mov al, '2'
mov bh, 0x00
mov bl, 0x01
mov cx, 1
int 0x10

;移动光标到下一个位置 (列数加1)
inc dl
mov ah, 0x02
int 0x10

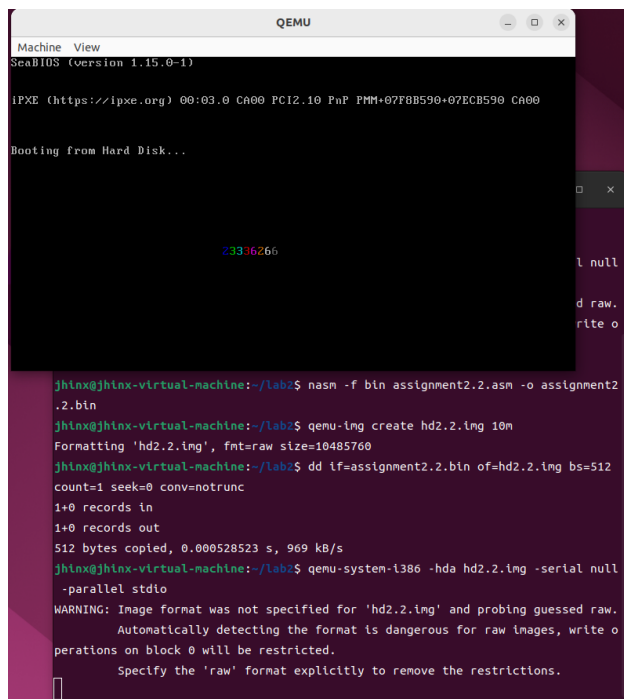
mov ah, 0x09
mov al, '3'
mov bh, 0x00
mov bl, 0x02
mov cx, 1
int 0x10

inc dl
mov ah, 0x02
int 0x10
```

代码说明如下：首先利用 AH=02H 将光标移动到（15,30）处，然后利用 AH=09H 输出信息，由于每次只能输出一个字符，因此输出后需要移动光标，以准备下一个字符的输出，我们利用的是 `inc dl` 使光标列数加一。随后不断重复实现这两个步骤即可把学号输出完。注意，为了防伪，本人在进行这一实验时，每次输出一个字符后都更改字符的颜色（即修改 bl 寄存器的值）。

2.运行程序：代码运行后在 qemu 的输出如下：





```
Machine View
Seabios (version 1.15.0-1)

IPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting From Hard Disk...

23336266

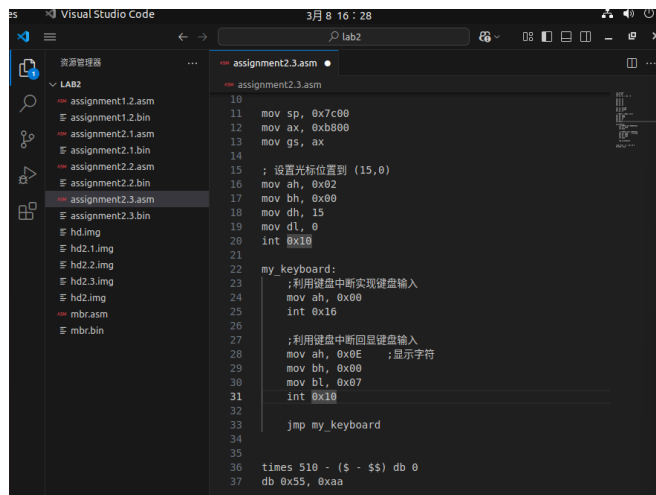
jhinx@jhinx-virtual-machine:~/lab2$ nasm -f bin assignment2.2.asm -o assignment2.2.bin
jhinx@jhinx-virtual-machine:~/lab2$ qemu-img create hd2.2.img 10m
Formatting 'hd2.2.img', fmt=raw size=10485760
jhinx@jhinx-virtual-machine:~/lab2$ dd if=assignment2.2.bin of=hd2.2.img bs=512 count=1 seek=0 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000528523 s, 969 kB/s
jhinx@jhinx-virtual-machine:~/lab2$ qemu-system-i386 -hda hd2.2.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd2.2.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

观察可知，成功在 qemu 的 (15,30) 处开始输出学号，并且每个字符的颜色都不一样。实验任务完成。

## 实验任务 5

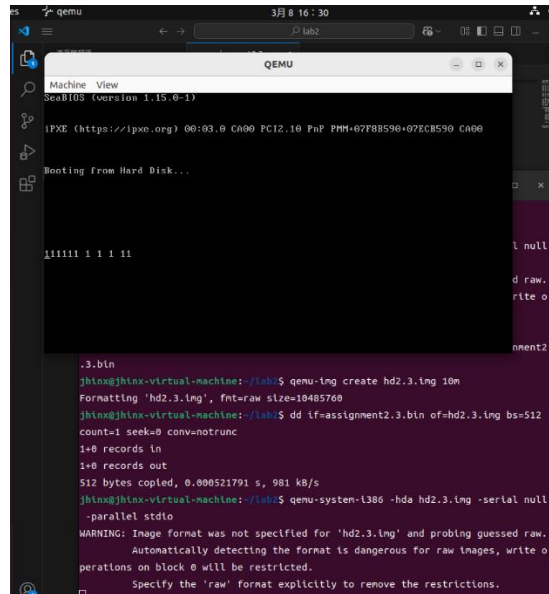
- 任务要求： 在 2.1 和 2.2 的知识的基础上，探索实模式的键盘中断，利用键盘中断实现键盘输入并回显。说说你是怎么做的，并将结果截图。
- 实验步骤：

1.编写 MBR：在编写 MBR 之前，我们需要先学习如何利用键盘中断实现输入并回显。查阅资料可知，利用 int 16H 中断，并把 ah 设置为 0x00，即可实现键盘输入，然后利用 int 10H 中断，并把 ah 设置成 0x0E，即可实现键盘回显。

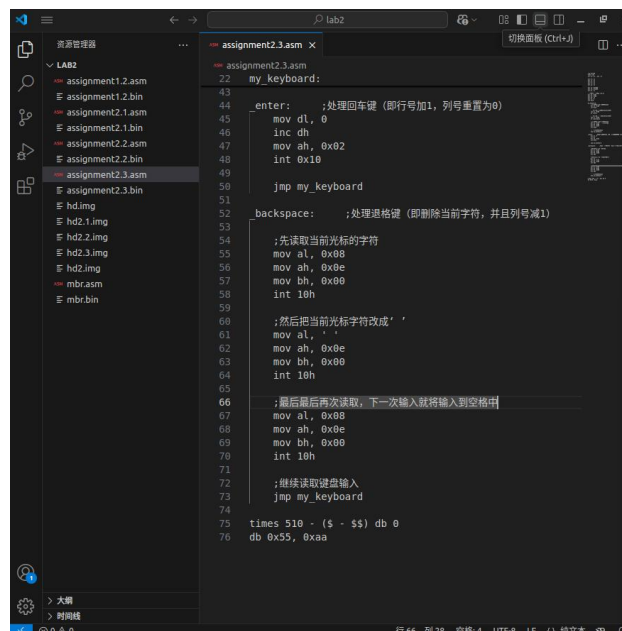


```
assignment2.3.asm
10
11 mov sp, 0x7c00
12 mov ax, 0xb800
13 mov gs, ax
14
15 ; 设置光标位置到 (15,0)
16 mov ah, 0x02
17 mov bh, 0x00
18 mov dh, 15
19 mov dl, 0
20 int 0x10
21
22 my_keyboard:
23 ;利用键盘中断实现键盘输入
24 mov ah, 0x00
25 int 0x16
26
27 ;利用键盘中断回显键盘输入
28 mov ah, 0x0E ;显示字符
29 mov bh, 0x00
30 mov bl, 0x07
31 int 0x10
32
33 jmp my_keyboard
34
35
36 times 510 - ($ - $$) db 0
37 db 0x55, 0xaa
```

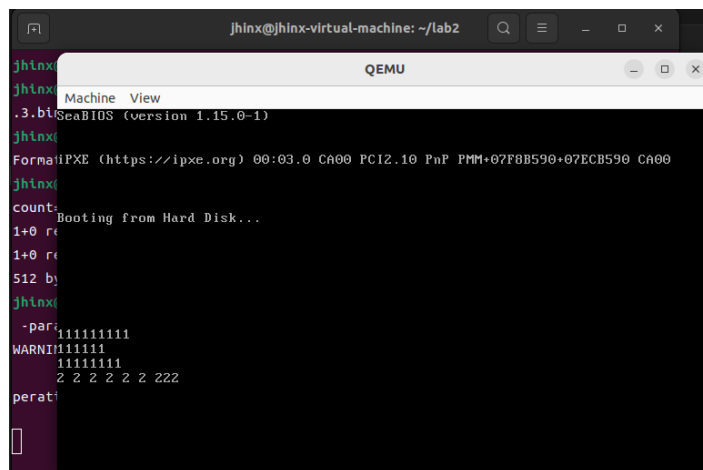
但是，这一代码运行的时候存在两个问题：1.回车键换行没有实现，回车后光标会回到当前行的最前面；2.退格键未实现，按一次退格不仅会清除当前光标的字符，还会额外再多退一行。如图：



因此，我们需要完善代码以实现回车和退格功能。其中回车功能相当于改变光标位置，将行数加一，并将列数清零；而退格功能相当于将光标的上一个位置的字符清除，并且将光标移动到上一个位置。了解实现方式后，我写出了以下代码段来实现回车和退格功能：



2.运行程序：用 qemu 模拟器运行程序后得到的结果如下：



经实际运行检验，回车和退格功能正常，并且键盘输入可以顺利回显，实验任务完成。

附加：除了回车和退格以外，本人还出于好奇，实现了 `tab` 键的功能（即输出四个空格），这一部分的代码如下：

```
_tab:
    mov ah, 0x03
    mov bx, 0x00
    int 0x10

    mov ah, 0x0e
    mov al, ' '
    int 0x10

    mov ah, 0x0e
    mov al, ' '
    int 0x10

    mov ah, 0x0e
    mov al, ' '
    int 0x10

    mov ah, 0x0e
    mov al, ' '
    int 0x10

    jmp my_keyboard
```

## -----实验任务 6（对应 assignment3）-----

### ● 任务要求：

- 1. 将实验指导书的伪代码转换成汇编代码，并放置在标号 `your_if` 之后；
- 2. 将实验指导书的伪代码转换成汇编代码，并放置在标号 `your_while` 之

后;

- 3. 编写函数 `your_function` 并调用之, 函数的内容是遍历字符数组 `string`;
- 4. 编写好代码之后, 在目录 `assignment` 下使用命令 `make run` 即可测试, 最后附上 `make run` 的截图, 并说说你是怎么做的。

● 实验步骤:

1.分支逻辑的实现: 这一部分的代码有以下几处关键的地方:

①在写第一个选择结构是, 汇编代码可以在当前函数写满足条件后的执行代码, 而无需再多设置一个函数来实现当前条件的满足情况, 如下面的代码所示:

```
your_if:
    mov eax, [a1]
    cmp eax, 40
    jl elseif_satisfy (如果不满足则跳转)
    add eax, 3 (从这条指令开始都是满足条件的执行过程)
    cdq
    mov ecx, 5
    idiv ecx
    mov [if_flag], eax
    jmp end_if (相当于伪代码的 'end')
```

②在进行除法时, 会遇到浮点数运算报错, 经查阅资料, 解决方法是在为除数赋值之前需要加上指令 `cdq` (如上面的代码所示), 作用是将 `EAX` 寄存器中的有符号整数扩展到 `EDX:EAX` 双寄存器中。

③由于 `your_if` 并不是函数, 因此在 `end_if` 部分不需要加上 `ret`, 如果加上了, 会导致整个程序提前结束, 最终无法输出正确结果。

2.循环逻辑的实现: 这一部分的代码有以下几处关键的地方:

①在汇编代码的逻辑里, 并不能像高级语言一样直接修改“数组” `while_flag` 中各个元素的值, 所以需要先用代码

```
mov esi, [while_flag]
```

将 `while_flag` 的首地址赋给 `esi` 寄存器, 然后再用 `esi` 进行赋值。

②同理, 在对 `while_flag[a2*2]` 进行赋值时, 需要用以下方法进行赋值, 而不能直接用立即数来访问数组中的元素

```
call my_random
mov ecx, [a2]
shl ecx, 1 (相当于 a2*2)
```

```
mov [esi+ecx], al
```

③由于 a2 是这一循环用来判断条件的变量，而在执行循环的过程中也会用到 a2，因此在执行过程中不能轻易改变 a2 的值，在这里我们使用 ecx 来创建一个 a2 的副本，再用左移操作来生成  $a2*2$ 。

④由于我们在判断循环是否满足时，使用的是 eax 和 25 进行比较，因此在每次循环执行结束后，我们都应该更新 eax 的数值，否则会导致 eax 一直是 a2 最开始的值，从而产生死循环。

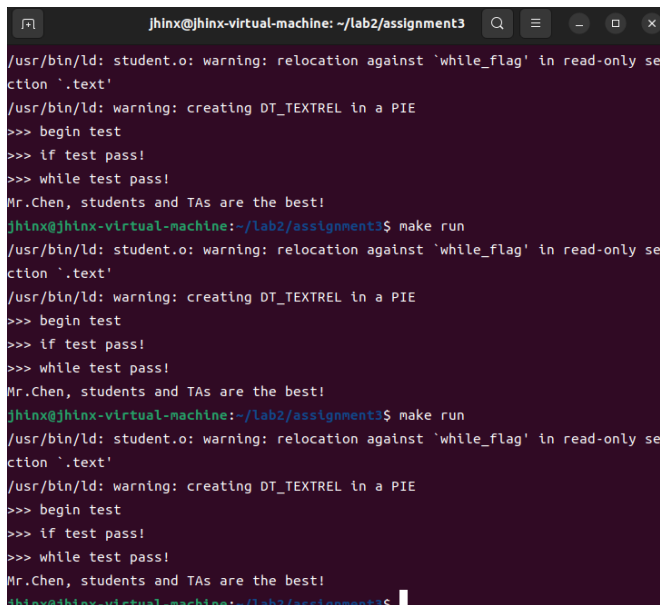
3.函数的实现：这一部分的代码有以下几处关键的地方：

①由于 string 是字符数组，因此我们不能使用 eax 来寄存数组中的元素，而应该改为 al，否则在运行时会报错。

②在 x86 中，原本伪代码的入栈和出栈通过如下方法实现：

```
Pushad (在函数调用之前，保存所有通用寄存器的值)
add al, 9 (对应 string[i]+9)
push eax (入栈)
call print_a_char
add esp, 4 (出栈。Esp: 栈寄存器)
popad (在函数返回之前，恢复所有通用寄存器的值)
```

4. 运行程序：在代码所在的文件夹中实验命令 make run 即可运行，运行后代码截图如下。



```
jhinx@jhinx-virtual-machine: ~/lab2/assignment3
/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!
jhinx@jhinx-virtual-machine: ~/lab2/assignment3$ 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!
jhinx@jhinx-virtual-machine: ~/lab2/assignment3$
```

为了检验代码的正确性，可以修改 test.cpp 中的 al，本人先后将 al 修改为 0, 1, 60 后重新运行程序，均得到正确的输出，实验任务完成。

## -----实验任务 7（对应 assignment4）-----

- 任务要求：字符回旋程序。请你用汇编代码实现一个字符回旋程序，使其能够在 qemu 显示屏上面，以不同的颜色、字符内容进行顺时针绕圈（见下图）。请注意，对于这个绕圈的字符，你还需要考虑他的运动速度，不要让其运动过快（可自行查阅其他资料，采用合适的中断来实现延时效果）
- 实验步骤：

### 1.编写 MBR：我们把整个程序分割成多个部分分别实现

①显示屏中间显示个人信息：可以依照 assignment2.2 的方式实现，但是发现篇幅过长。鉴于我们在 assignment3 时学习了循环逻辑的实现，因此在这里我们也尝试使用循环来输出个人信息。显而易见，想要用循环来输出，需要把个人信息存进一个字符数组中，并且每次循环输出一个字符，并且更新光标位置以进行下一次输出，这一部分的代码如下：

```
mov ax, 1986          (1986 为经过计算后的显示屏中间位置)
mov di, ax
mov si, message       (message 为字符数组)
mov cx, 14             (字符串长度为 7 (2*7=14))
mov ah, 0xEC           (红色前景，黄色背景)

xian_shi_xue_hao:
    mov al, [si]
    mov [gs:di], ax
    add di, 2          (更新光标位置)
    inc si             (遍历字符数组的下一个元素)
    loop xian_shi_xue_hao

.....
message db 'jhinx 23336266'
```

②字符的伪随机输出：观察得知，字符的输出是以“1357902468”为序列的伪随机输出，并且每输出两个字符更改一次前景色，每输出 16 个字符更改一次背景色。为了显示区别，在本人的代码中，每输出三个字符更改一次前景色，每输出 14 个字符更改一次背景色，并且增加了一个判断背景色是否为黑色的代码段，以避免输出黑色的背景色。为了实现这些循环，需要设置一个数组：字符输出数组；还需要设置以下变量：记录当前输出的是字符串序列的第几个元素的变量、记录当前前/背景色颜色的变量、记录当前

前/背景色已经输出了多少个字符的变量。变量的声明代码如下：

```
current_chars db '1','3','5','7','9','0','2','4','6','8'
current_char db 0           ;显示当前输出的是字符串序列的第几个元素
front_color db 1
background_color db 0x20
counter dw 0               ;显示这一背景色下已经输出了多少个字符了
color_counter db 0        ;显示这一前景色下已经输出了多少个字符了
```

这一部分的代码如下：

```
show_current_char:
    ;获取当前要显示的数字
    mov bx, current_chars
    movzx ax, byte [current_char]    ;(扩展成 16 位)
    add bx, ax
    mov al, [bx]                    ;从数组中获取数字
    ;设置颜色属性
    mov ah, [background_color]
    add ah, [front_color]
    ;显示字符
    mov bx, [pos]
    mov [gs:bx], ax
    ;更新 color_counter 并检查是否需要更改颜色(每 3 个字符更新一次)
    inc byte [color_counter]
    cmp byte [color_counter], 3
    jne skip_front_update
    ;设置了 16 种颜色，因此每到 16 的倍数需要重置颜色计数器
    mov byte [color_counter], 0
    inc byte [front_color]
    cmp byte [front_color], 16
    jne skip_frontcolor_cycle
    mov byte [front_color], 1
skip_frontcolor_cycle:
skip_front_update:
    ;字符序列循环输出
    inc byte [current_char]
    cmp byte [current_char], 10
    jne skip_string_cycle
    mov byte [current_char], 0
skip_string_cycle:
    ;更新计数器和背景色
    inc word [counter]
    mov ax, [counter]
    mov dx, 0
    mov bx, 14
```

```

div bx          (通过求余来查看是否已经输出够了 14 个字符)
cmp dx, 0       ;检查是否需要改变背景色
jne skip_background_change

add byte [background_color], 0x10 ;改变背景色
cmp byte [background_color], 0    ;检查是否为黑色 (黑色为 0)
jne skip_background_change
mov byte [background_color], 0x20 ;如果是黑色, 改为绿色
skip_background_change:
ret

```

③实现在最外圈顺时针输出：实际上就是不断更新光标的位置，但是光标的位置始终在最外圈。依据 **assignment1** 中学到的坐标逻辑，我们可以知道，光标向右变化每次列数+1（**ax+2**），向下变化每次行数+1（**ax+160**），向左变化每次列数-1（**ax-2**），向上变化每次行数-1（**ax-160**）。我们需要设置以下变量：

```

pos dw 0          ;显示当前位置
direction dw 0     ;显示当前方位

```

我们通过方位变量和立即数来比较以判断当前应该向哪个方向变化，在每次到角落时改变 **direction** 的值来实现转向。然后通过 **pos** 来显示当前光标在显示屏中的位置。部分代码如下：

```

update_position:
    mov ax, [pos]
    cmp word [direction], 0    ;向右
    je move_right
    cmp word [direction], 1    ;向下
    je move_down
    cmp word [direction], 2    ;向左
    je move_left
    cmp word [direction], 3    ;向上
    je move_up

move_right:
    add ax, 2
    cmp ax, 158                ;右边界
    jle change_position
    sub ax, 2                   ;退回到最后一个有效位置
    add ax, 160                 ;向下移动一行
    mov word [direction], 1
    jmp change_position

move_down:

```



```

    add ax, 160
    cmp ax, 4000      ;下边界
    jle change_position
    sub ax, 160       ;退回到最后一个有效位置
    sub ax, 2         ;向左移动一列
    mov word [direction], 2
    jmp change_position
..... (move left 和 move up 在此省略)
change_position:
    mov [pos], ax
    ret

```

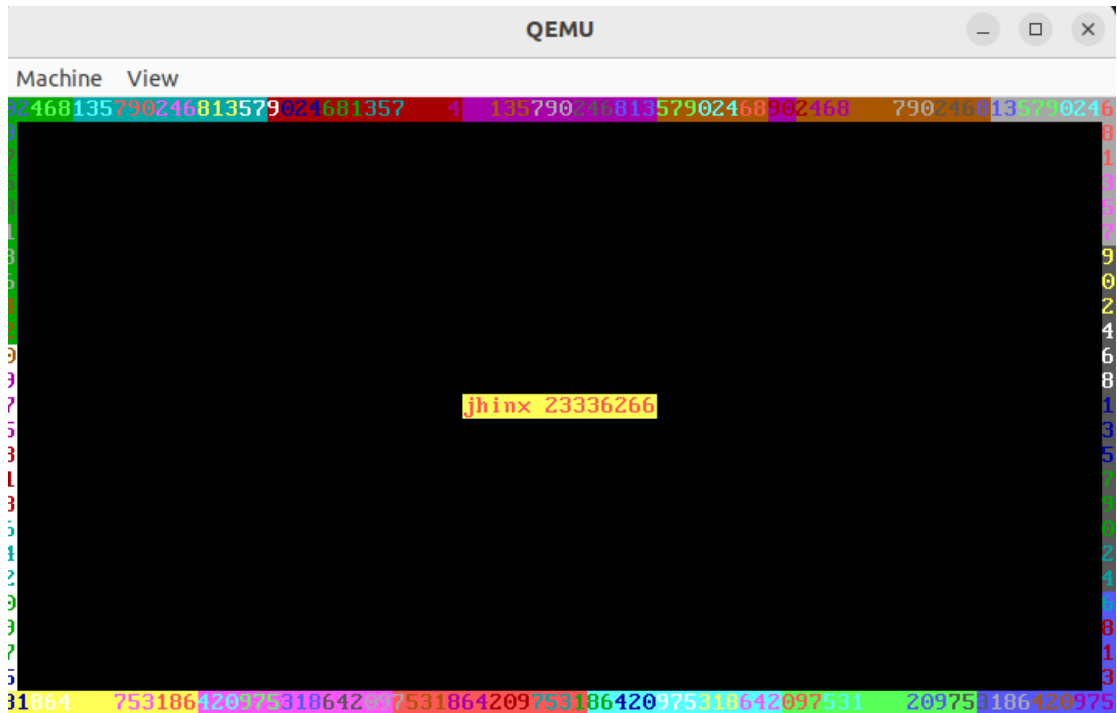
④实现延时函数：延时函数的作用是减缓输出频率，我们可以通过一个双重循环来让一个寄存器的值不断自增来实现。经过实践，外层循环次数为 10 的时候输出速度比较合适。延时函数的代码如下：

```

_delay:
    pusha
    mov cx, 10      (10 次外层循环)
delay_outer:
    push cx
    mov cx, 0xFFFF
delay_inner:
    loop delay_inner
    pop cx
    loop delay_outer
    popa
    ret

```

2. 运行程序：MBR 程序写好后，运行结果如下：



经观察，该程序实现了顺时针循环输出，实验任务完成。

## Section 5 实验总结与心得体会

该实验过程中，本人遇到了以下几个情况，有以下几个注意事项：

1.在实验 2 的 debug 中，本人发现就像实验指导书中说的，在进行 gdb debug 的时候，如果单步运行到 `int 10h` 这种中断指令时，确实会导致代码无法继续调试。此时可以使用 `break+行号` 设置断点，跳过中断指令，也可以使用键盘 `↓` 键移动跳过中断指令，然后直接输入 `break` 可以为高亮行设置断点，也可以跳过中断指令。

2.在进行 debug 调试的时候，有时候我们设置了断点后才发现，在断点前面有地方我们也想设置断点进行调试。但是，让程序向下运行是容易的，而让程序按反方向运行确实很难的。是否只能通过重启程序重新查看？其实有一个更简单一些的方法，我们可以先在所有想看的喊设置断点，随后输入命令 `info breakpoints` 可以查看断点表，如下图所示：

```
jhinx@jhinx-virtual-machine: ~/lab2
assignment2.1.asm
16  mov AH, 03H
17  mov BX, 00H
18  int 10H
19
20  ;实现光标的移动
21  mov ah, 0x02
22  mov bh, 0x00
23  mov dh, 10
24  mov dl, 20
25  int 0x10
26
27  jmp $;
28  times 510 - ($ - $$) db 0

remote Thread 1.1 In: L24 PC: 0x7c21
Num Type Disp Enb Address What
1 breakpoint keep y 0x00007c00 assignment2.1.asm:3
breakpoint already hit 1 time
2 breakpoint keep y 0x00007c1f assignment2.1.asm:23
breakpoint already hit 1 time
3 breakpoint keep y 0x00007c21 assignment2.1.asm:24
breakpoint already hit 1 time
(gdb) jump *0x00007c1f
```

在断点表中，我们可以看到每个断点的地址（address），然后我们可以通过输入指令 `jump *地址` 来跳转到我们想要去到的断点处，这样即可重新调试前面的代码。

3.在 assignment2.2 中，起初写退格实现代码时遇到了困难，许多次写的程序都不对，后面通过查阅资料，了解到功能号 AH=0EH 可以自动更新光标的位置，问题也就迎刃而解。在做完整个实验后，本人又混合使用传统中断给出了另一个实现代码：

```
_backspace:
    mov ah, 0x03
    mov bx, 0x00
    int 0x10

    dec dl
    mov ah, 0x02
    int 0x10

    mov ah, 0x0e
    mov al, ' '
    int 0x10

    mov ah, 0x03
    mov bx, 0x00
    int 0x10

    dec dl
```

```
mov ah, 0x02  
int 0x10  
jmp my_keyboard
```

## Section 6 附录：代码清单

所有的汇编代码详见附件。