

lab2

1、实验要求

编写程序，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

学习使用gdb来调试程序的基本方法

2、实验过程+实验结果+关键代码

Assignment 1

1.1:example 1复现：操作系统启动并输出Hello World

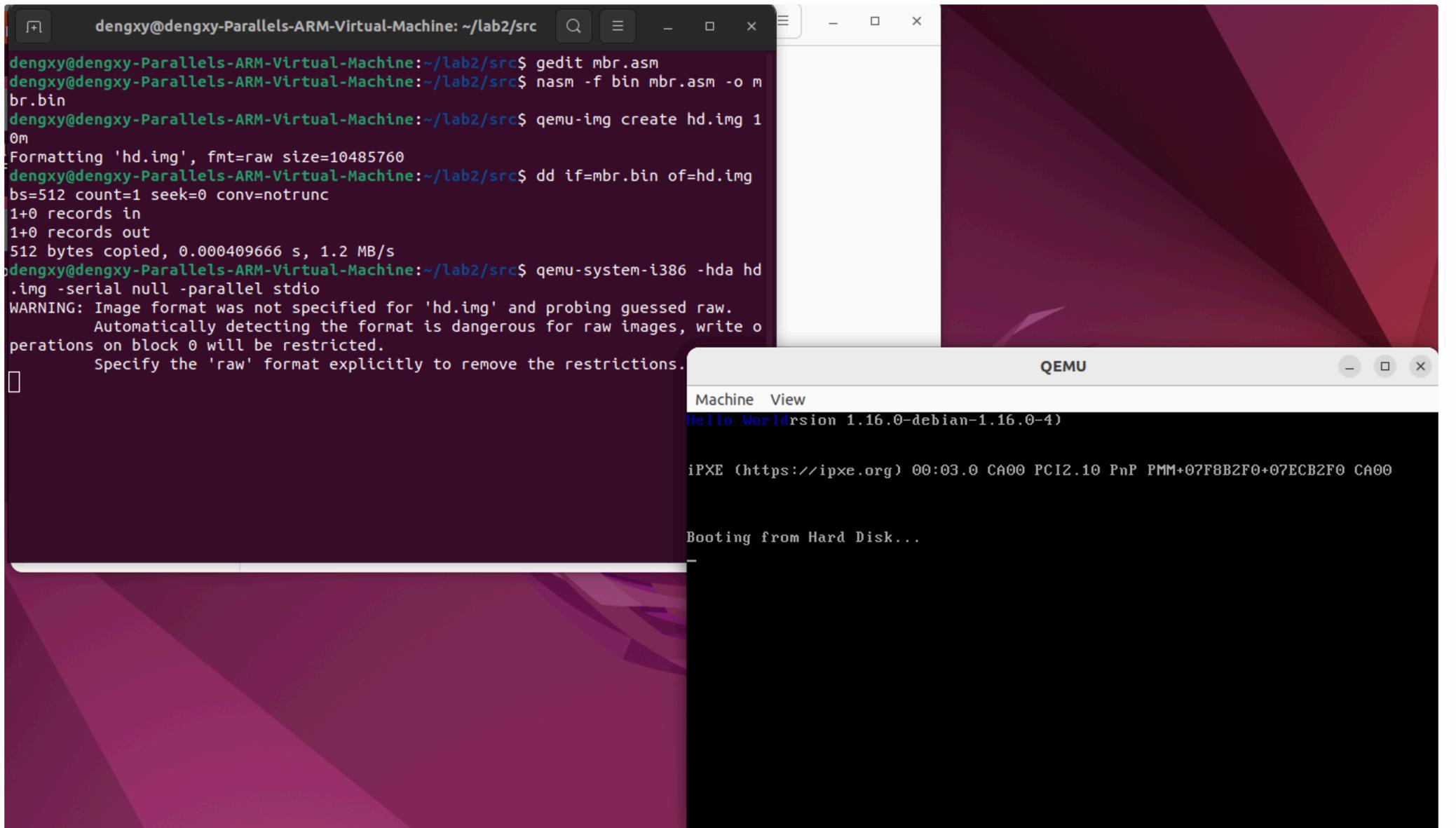
```
1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4 ; 初始化段寄存器，段地址全部设为0
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 ; 初始化栈指针
12 mov sp, 0x7c00
13 mov ax, 0xb800
14 mov gs, ax
15
16
17 mov ah, 0x01 ; 青色
18 mov al, 'H'
19 mov [gs:2 * 0], ax
20
21 mov al, 'e'
22 mov [gs:2 * 1], ax
23
24 mov al, 'l'
25 mov [gs:2 * 2], ax
26
27 mov al, 'l'
28 mov [gs:2 * 3], ax
29
30 mov al, 'o'
31 mov [gs:2 * 4], ax
32
33 mov al, ' '
```

```
34 mov [gs:2 * 5], ax
35
36 mov al, 'W'
37 mov [gs:2 * 6], ax
38
39 mov al, 'o'
40 mov [gs:2 * 7], ax
41
42 mov al, 'r'
43 mov [gs:2 * 8], ax
44
45 mov al, 'l'
46 mov [gs:2 * 9], ax
47
48 mov al, 'd'
49 mov [gs:2 * 10], ax
50
51 jmp $ ; 死循环
52
53 times 510 - ($ - $$) db 0
54 db 0x55, 0xaa
```

```
1 gedit mbr.asm #编辑mbr.asm文件
2 nasm -f bin mbr.asm -o mbr.bin
3 # -f参数指定的是输出的文件格式， -o指定的是输出的文件名。mbr.bin中保存的是机器可以识别的机器指令
4 qemu-img create hd.img 10m
5 # hd.img是生成的硬盘的文件名 10m虚拟磁盘大小
6 dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
7 # 将MBR写入hd.img的首扇区 使用dd命令
8 qemu-system-i386 -hda hd.img -serial null -parallel stdio
9 # 使用qemu模拟计算机启动
10 # -hda hd.img表示将文件hd.img作为第0号磁盘映像
11 # -serial dev表示重定向虚拟串口到空设备中
12 # -parallel stdio 表示重定向虚拟并口到主机标准输入输出设备中
```

dd 命令的相关参数

- `if` 表示输入文件。(`if=mbr.bin`)
- `of` 表示输出文件。(`of=hd.img`)
- `bs` 表示块大小，以字节表示。(`bs=512`)
- `count` 表示写入的块数目。(`count=1`)
- `seek` 表示越过输出文件中多少块之后再写入。(`seek=0`)
- `conv=notrunc` 表示不截断输出文件，如果不加上这个参数，那么硬盘在写入后多余部份会被截断。



在第一行成功输出蓝色 Hello World

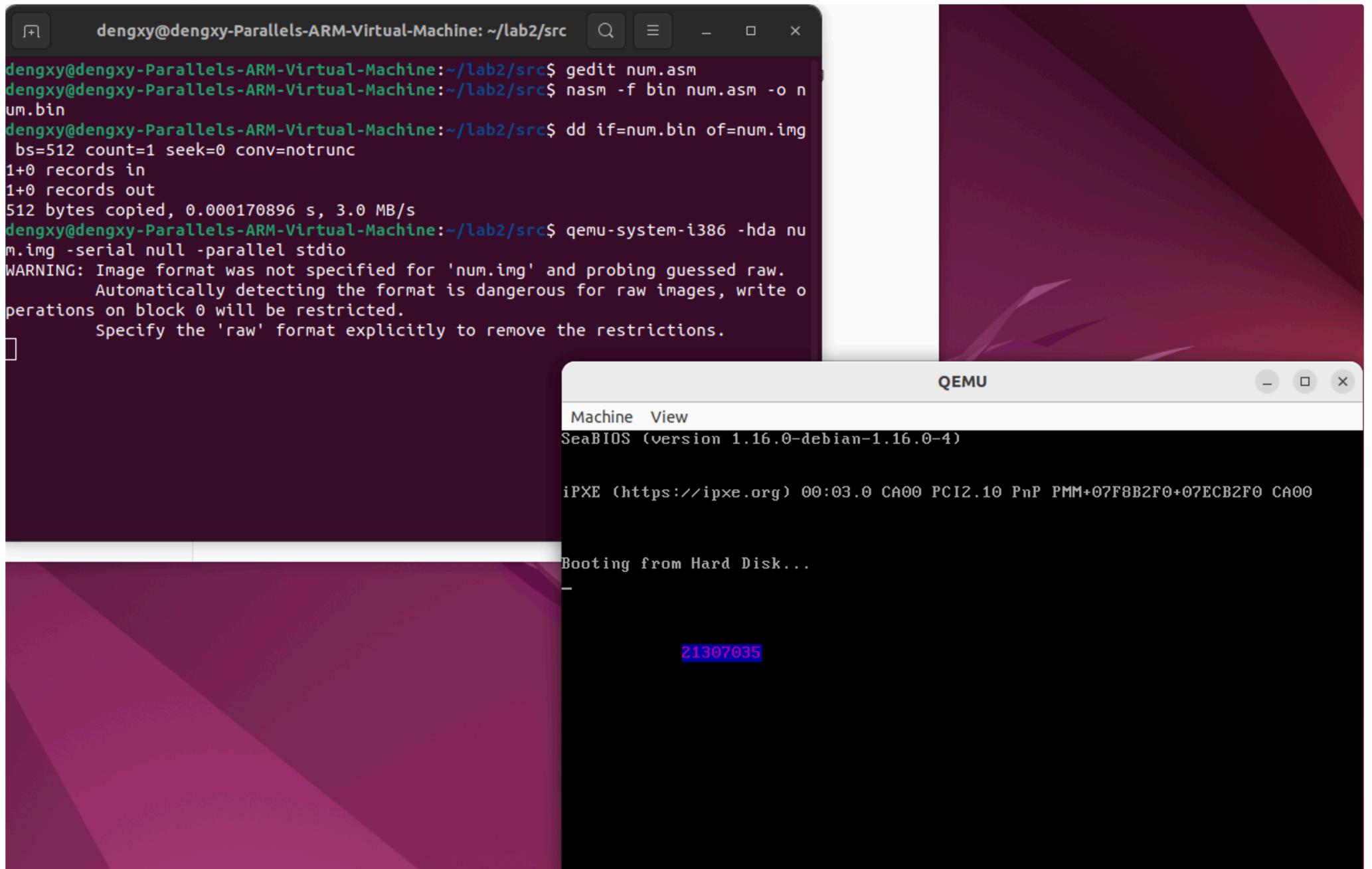
1.2：修改example 1的代码，使得MBR被加载到0x7C00后在(12,12)(12,12)处开始输出你的学号。注意，你的学号显示的前景色和背景色必须和教程中不同。

```
1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4 ; 初始化段寄存器，段地址全部设为0
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 ; 初始化栈指针
12 mov sp, 0x7c00
13 mov ax, 0xb800
14 mov gs, ax
15
16
17 mov ah, 0x15 ;背景蓝色，前景品红
18 mov al, '2'
19 mov [gs:2 * (12*80 + 12)], ax
20
21 mov al, '1'
22 mov [gs:2 * (12*80 + 13)], ax
23
24 mov al, '3'
```

```
25 mov [gs:2 * (12*80 + 14)], ax
26
27 mov al, '0'
28 mov [gs:2 * (12*80 + 15)], ax
29
30 mov al, '7'
31 mov [gs:2 * (12*80 + 16)], ax
32
33 mov al, '0'
34 mov [gs:2 * (12*80 + 17)], ax
35
36 mov al, '3'
37 mov [gs:2 * (12*80 + 18)], ax
38
39 mov al, '5'
40 mov [gs:2 * (12*80 + 19)], ax
41
42
43 jmp $ ; 死循环
44
45 times 510 - ($ - $$) db 0
46 db 0x55, 0xaa
```

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

其中，`[gs:2 * (12*80 + 12)]` 表示内存地址 `(12, 12)` 处，`2` 是因为每个字符占用两个字节，`80` 是因为屏幕每行有80个字符



Assignment2

2.1 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。

1. 光标位置获取：使用 `int 10h` 中断中的 `0x03` 功能号，调用该中断时，会返回光标当前所在行列数。返回的结果由寄存器 `AH` 和 `DL` 分别存储光标所在的行和列数。
2. 光标移动：使用 `int 10h` 中断中的 `0x02` 功能号，可以将光标移动到指定的行和列。需要将要移动到的行数和列数分别存入 `BH` 和 `DH` 寄存器中，然后调用中断 `int 10h` 即可。

示例：

`inter.asm`

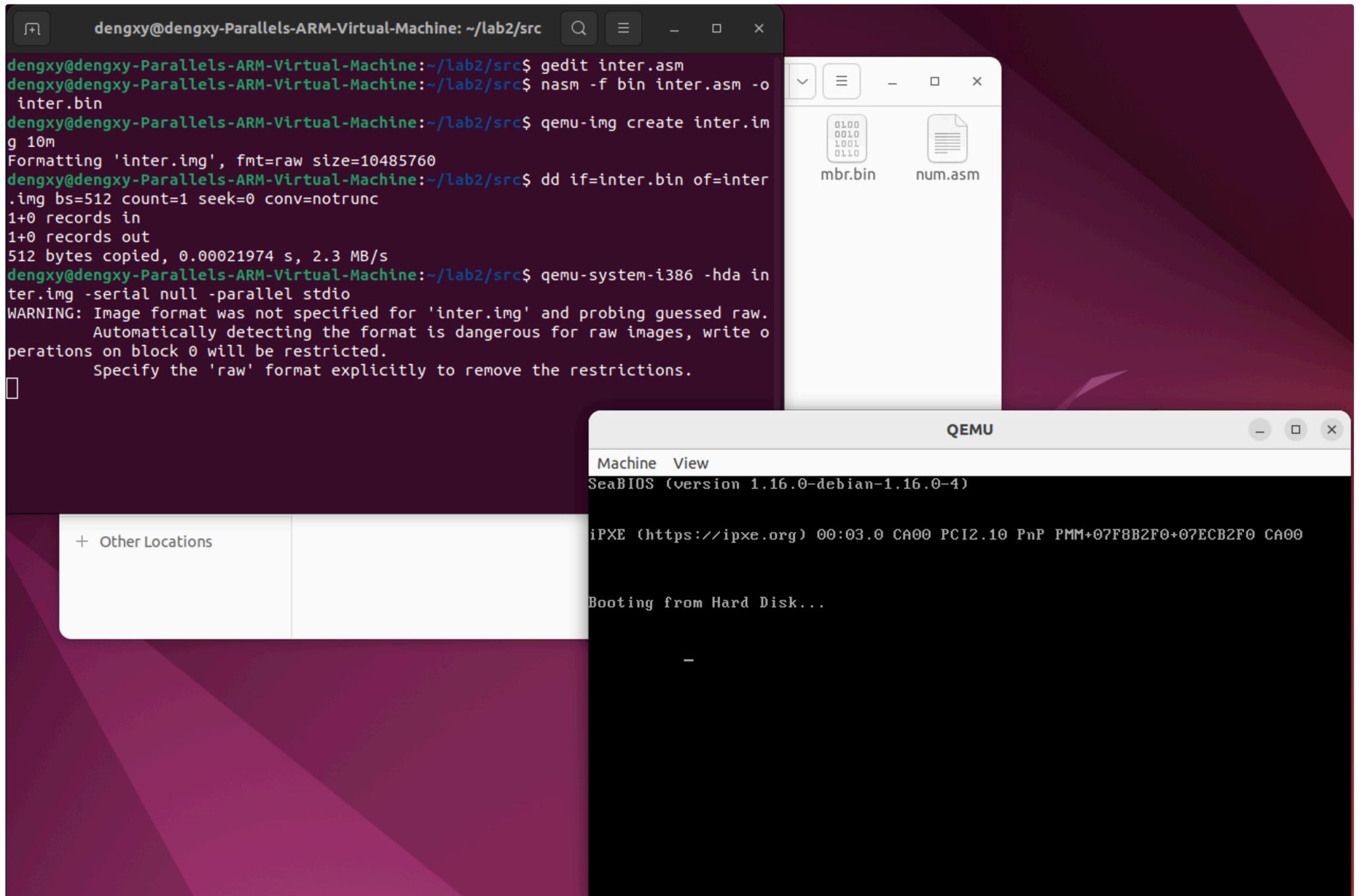
```

1 org 0x7c00
2
3 [bits 16]
4
5 mov ax, 0
6 mov ds, ax
7
8 ; 获取光标当前位置
9 mov ah, 0x03 ; 功能号为0x03
10 int 0x10      ; 调用int 10h中断
11 mov ah, 0x0e ; 功能号为0x0e
12 mov bh, 0x00 ; 页号为0
13 mov dl, ' ' ; 显示字符为空格

```

```
14 int 0x10      ; 调用int 10h中断, 将空格打印到光标所在位置
15 add dl, 0x30 ; 将光标所在列数转换成字符, 存放在DL中
16 int 0x10      ; 调用int 10h中断, 将光标所在列数显示在屏幕上
17 mov dl, ah    ; 将光标所在行数存放在DL中
18 add dl, 0x30 ; 将光标所在行数转换成字符, 存放在DL中
19 mov ah, 0x0e ; 功能号为0x0e
20 int 0x10      ; 调用int 10h中断, 将光标所在行数显示在屏幕上
21
22 ; 移动光标到指定位置
23 mov ah, 0x02 ; 功能号为0x02
24 mov bh, 0x00 ; 页号为0
25 mov dh, 0x0a ; 移动到第10行
26 mov dl, 0x0a ; 移动到第10列
27 int 0x10      ; 调用int 10h中断, 将光标移动到指定位置
28
29 jmp $
30
31 times 510 - ($ - $$) db 0
32 db 0x55, 0xaa
33
```

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



2.2 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。

```

1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4 ; 初始化段寄存器，段地址全部设为0
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 ; 初始化栈指针
12 mov sp, 0x7c00
13 mov ax, 0xb800
14 mov gs, ax
15
16
17 mov ah, 0x0e ; 设置光标显示属性
18 mov bh, 0x00 ; 页号为0
19 mov al, '2' ; 显示字符'2'
20 mov cx, 0x0002 ; 光标移动次数，为2
21 int 0x10 ; 调用int 10h中断，将字符'2'显示在屏幕上
22 mov al, '1' ; 显示字符'1'
23 int 0x10 ; 调用int 10h中断，将字符'1'显示在屏幕上

```

```

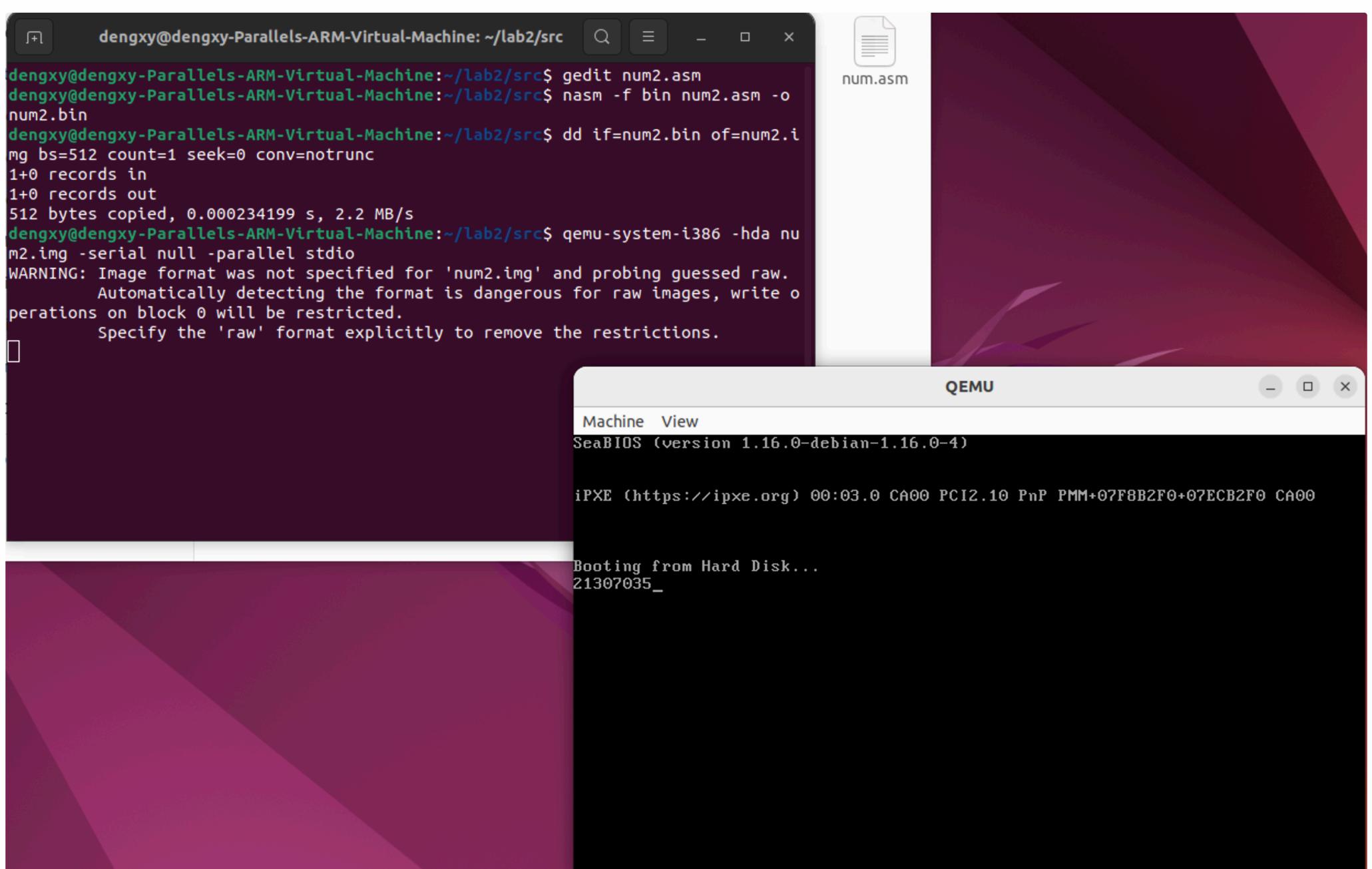
24 mov al, '3' ; 显示字符'3'
25 int 0x10      ; 调用int 10h中断, 将字符'3'显示在屏幕上
26 mov al, '0' ; 显示字符'0'
27 int 0x10      ; 调用int 10h中断, 将字符'0'显示在屏幕上
28 mov al, '7' ; 显示字符'7'
29 int 0x10      ; 调用int 10h中断, 将字符'7'显示在屏幕上
30 mov al, '0' ; 显示字符'0'
31 int 0x10      ; 调用int 10h中断, 将字符'0'显示在屏幕上
32 mov al, '3' ; 显示字符'3'
33 int 0x10      ; 调用int 10h中断, 将字符'3'显示在屏幕上
34 mov al, '5' ; 显示字符'5'
35 int 0x10      ; 调用int 10h中断, 将字符'5'显示在屏幕上
36
37 jmp $ ; 死循环
38
39 times 510 - ($ - $$) db 0
40 db 0x55, 0xaa

```

```

1 gedit num2.asm
2 nasm -f bin num2.asm -o num2.bin
3 qemu-img create num2.img 10m
4 dd if=num2.bin of=num2.img bs=512 count=1 seek=0 conv=notrunc
5 qemu-system-i386 -hda num2.img -serial null -parallel stdio

```



2.3 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

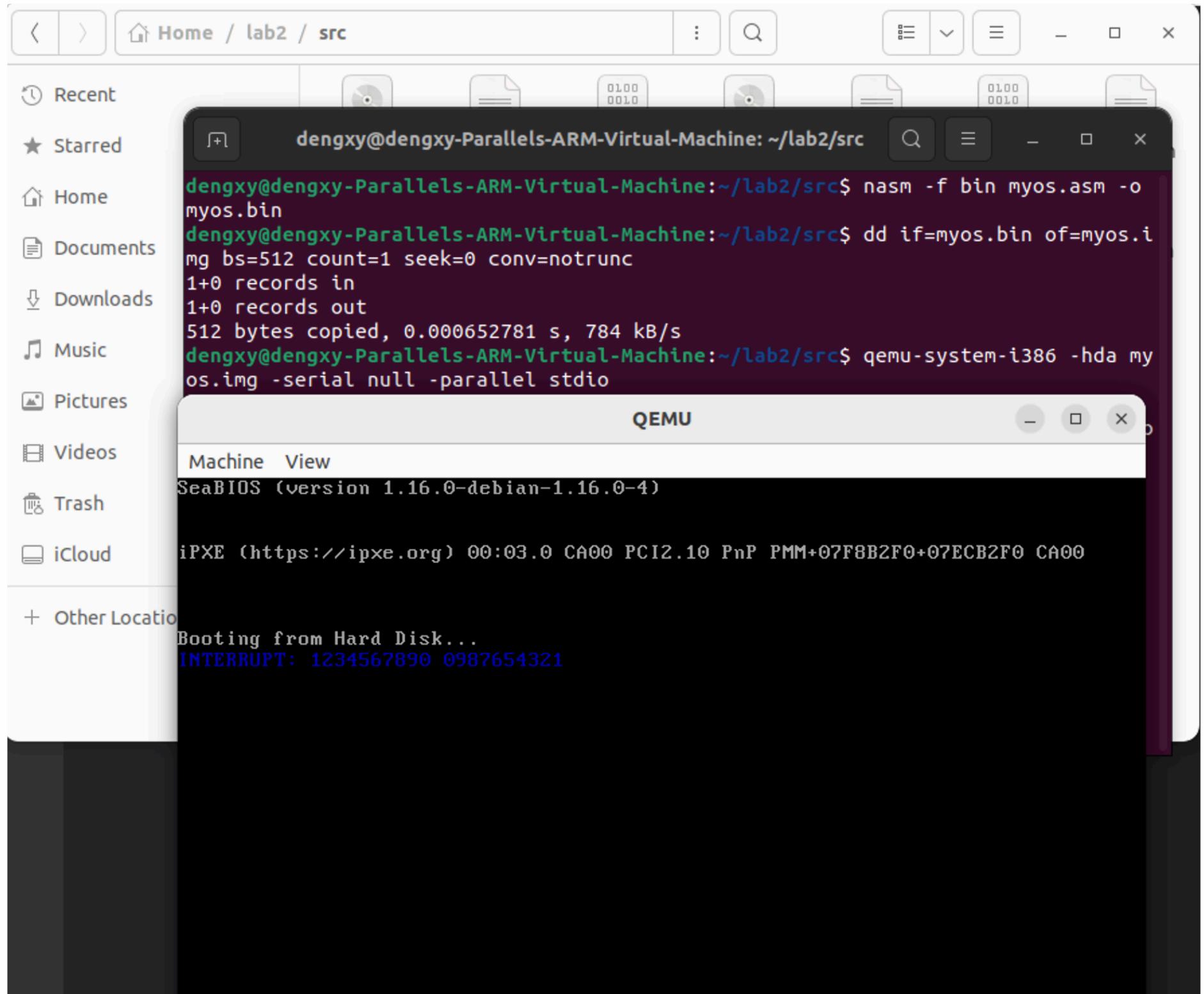
```
1 org 0x7c00
2 [bits 16]
3 xor ax, ax ; eax = 0
4 ; 初始化段寄存器，段地址全部设为0
5 mov ds, ax
6 mov ss, ax
7 mov es, ax
8 mov fs, ax
9 mov gs, ax
10
11 ; 初始化栈指针
12 mov sp, 0x7c00
13 ; mov ax, 0xb000
14 ; mov gs, ax
15
16 mov ah, 0x03 ; 功能码，表示调用读取光标位置的中断功能
17 mov bh, 0x00 ; 页码，文本状态设为0
18 int 0x10 ; interrupt
19
20 mov bl, 0x01 ; blue
21 mov al, 'I'
22 mov ah, 0x09 ; 功能码，表示在当前光标位置写字符和属性的中断功能
23 mov cx, 0x01 ; 输出字符的个数
24 int 0x10 ; interrupt
25
26 mov ah, 0x02 ; 功能码，表示调用设置光标位置的中断功能
27 mov bh, 0x00 ; 页码，文本状态设为0
28 add dl, 1 ;
29 int 0x10 ; interrupt
30
31 mov al, 'N'
32 mov ah, 0x09 ; 功能码，表示在当前光标位置写字符和属性的中断功能
33 mov cx, 0x01 ; 输出字符的个数
34 int 0x10 ; interrupt
35
36 mov ah, 0x02 ; 功能码，表示调用设置光标位置的中断功能
37 mov bh, 0x00 ; 页码，文本状态设为0
38 add dl, 1 ;
39 int 0x10 ; interrupt
40
41 mov al, 'T'
42 mov ah, 0x09 ; 功能码，表示在当前光标位置写字符和属性的中断功能
43 mov cx, 0x01 ; 输出字符的个数
44 int 0x10 ; interrupt
45
46 mov ah, 0x02 ; 功能码，表示调用设置光标位置的中断功能
47 mov bh, 0x00 ; 页码，文本状态设为0
48 add dl, 1 ;
```

```
49 int 0x10 ; interrupt
50
51 mov al, 'E'
52 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
53 mov cx, 0x01 ; 输出字符的个数
54 int 0x10 ; interrupt
55
56 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
57 mov bh, 0x00 ; 页码, 文本状态设为0
58 add dl, 1 ;
59 int 0x10 ; interrupt
60
61 mov al, 'R'
62 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
63 mov cx, 0x01 ; 输出字符的个数
64 int 0x10 ; interrupt
65
66 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
67 mov bh, 0x00 ; 页码, 文本状态设为0
68 add dl, 1 ;
69 int 0x10 ; interrupt
70
71 mov al, 'R'
72 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
73 mov cx, 0x01 ; 输出字符的个数
74 int 0x10 ; interrupt
75
76 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
77 mov bh, 0x00 ; 页码, 文本状态设为0
78 add dl, 1 ;
79 int 0x10 ; interrupt
80
81 mov al, 'U'
82 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
83 mov cx, 0x01 ; 输出字符的个数
84 int 0x10 ; interrupt
85
86 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
87 mov bh, 0x00 ; 页码, 文本状态设为0
88 add dl, 1 ;
89 int 0x10 ; interrupt
90
91 mov al, 'P'
92 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
93 mov cx, 0x01 ; 输出字符的个数
94 int 0x10 ; interrupt
95
96 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
97 mov bh, 0x00 ; 页码, 文本状态设为0
```

```
98 add dl, 1 ;
99 int 0x10 ; interrupt
100
101 mov al, 'T'
102 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
103 mov cx, 0x01 ; 输出字符的个数
104 int 0x10 ; interrupt
105
106 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
107 mov bh, 0x00 ; 页码, 文本状态设为0
108 add dl, 1 ;
109 int 0x10 ; interrupt
110
111 mov al, ':'
112 mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
113 mov cx, 0x01 ; 输出字符的个数
114 int 0x10 ; interrupt
115
116 mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
117 mov bh, 0x00 ; 页码, 文本状态设为0
118 add dl, 1 ;
119 int 0x10 ; interrupt
120
121 ; 键盘I/O中断调用 (INT 16H)
122 board_interrupt:
123     mov ah, 0x00 ;
124     int 0x16 ;
125     or al, 0x00 ;
126     jnz input
127     jmp board_interrupt
128
129 input:
130
131     mov ah, 0x02 ; 功能码, 表示调用设置光标位置的中断功能
132     mov bh, 0x00 ; 页码, 文本状态设为0
133     add dl, 1 ;
134     int 0x10 ; interrupt
135
136     mov ah, 0x09 ; 功能码, 表示在当前光标位置写字符和属性的中断功能
137     mov cx, 0x01 ; 输出字符的个数
138     int 0x10 ; interrupt
139
140     jmp board_interrupt ;
141
142     jmp $ ; 死循环
143
144 times 510 - ($ - $$) db 0
145 db 0x55, 0xaa
```

```
1 gedit myos.asm
2 nasm -f bin myos.asm -o myos.bin
3 #qemu-system-i386 myos.bin
4 qemu-img create myos.img 10m
5 dd if=myos.bin of=myos.img bs=512 count=1 seek=0 conv=notrunc
6 qemu-system-i386 -hda myos.img -serial null -parallel stdio
```

输出INTERRUPT: 之后，实现键盘输入并回显，如图，为1234567890 0987654321



assignment 3

3.1: 分支逻辑的实现：请将下列伪代码转换成汇编代码，并放置在标号 `your_if` 之后。

```
1 if a1 < 12 then
2   if_flag = a1 / 2 + 1
3 else if a1 < 24 then
4   if_flag = (24 - a1) * a1
5 else
6   if_flag = a1 << 4
7 end
```

```

1 | your_if:
2 |     mov eax, [a1];将内存地址a1中的值移动到寄存器eax中
3 |     cmp eax, 0xc ;比较eax(a1)与0xc(12)的值
4 |     jl less_than_12 ;eax<0xc跳转less_than_12
5 |     jmp more_than_12 ;eax≥0xc跳转more_than_12
6 |
7 | less_than_12:
8 |     mov eax, [a1];将内存地址a1中的值再次移动到寄存器eax中
9 |     mov ebx, 0x2; 将0x2的值放入寄存器ebx
10 |    div ebx;将寄存器eax中的值除以寄存器ebx中的值
11 |    add eax, 1;将除法结果加1, 并将其存回eax中
12 |    mov [if_flag], eax;将寄存器 eax 中的值移动到内存位置 if_flag 中
13 |    jmp your_if_end
14 | more_than_12:
15 |     mov eax, [a1];将内存地址a1中的值再次移动到寄存器eax中
16 |     cmp eax, 24;将寄存器 eax 中的值与 24 进行比较
17 |     jl less_than_24;如果寄存器eax中的值小于24, 则跳转less_than_24
18 |     jmp more_than_24;否则跳转more_than_24
19 | less_than_24:
20 |     mov ecx, 0x18;将值 0x18 (十进制24) 移动到寄存器 ecx 中
21 |     sub ecx, eax;将寄存器 eax 中的值从寄存器 ecx 中的值中减去, 存在ecx中
22 |     imul ecx, eax;将寄存器 ecx 中的值乘以寄存器 eax 中的值
23 |     mov [if_flag], ecx;将乘法结果移动到内存位置 if_flag 中
24 |     jmp your_if_end
25 | more_than_24:
26 |     mov eax, [a1];将内存地址a1中的值再次移动到寄存器eax中
27 |     shl eax, 4;将寄存器eax中的值向左移动4位
28 |     mov [if_flag], eax;将移位操作的结果移动到内存位置 if_flag 中
29 |     jmp your_if_end
30 | your_if_end:
31 |

```

- `cmp` 指令比较两个操作数的大小。
- `jl` 指令是 "jump if less than" 的缩写, 如果上一次比较结果中小于标志位 (即第一个操作数小于第二个操作数), 则跳转到指定的代码块。同理, `jmp` 指令是无条件跳转。
- `inc` 指令是加 1 操作。
- `sub` 指令将两个操作数相减。
- `imul` 指令是有符号整数乘法指令, 将两个操作数相乘, 并将结果存储在第一个操作数中。
- `shl` 指令是逻辑左移指令, 将一个寄存器或内存位置的值向左移动指定的位数。

3.2: 循环逻辑的实现：请将下列伪代码转换成汇编代码，并放置在标号 `your_while` 之后。

```
1 while a2 >= 12 then
2     call my_random          // my_random将产生一个随机数放到eax中返回
3     while_flag[a2 - 12] = eax
4     --a2
5 end
```

```
1 your_while:
2
3     mov ebx, [a2];将a2数组的第一个元素的地址存储在ebx寄存器中
4     cmp ebx, 0xc;比较ebx寄存器中存储的值与十六进制数0xc (十进制数12)的大小关系
5     jl your_while_end
6
7     while:
8         call my_random;调用一个函数my_random, 获取一个随机数, 并将随机数保存在eax寄存器中
9         mov edx, [while_flag];将while_flag变量的值存储在edx寄存器中
10        sub ebx, 0xc;将ebx寄存器中的值减去12, 即将a2数组中下一个元素的地址存储在ebx寄存器中
11        mov [edx + ebx], eax;将eax寄存器中的值 (即获取到的随机数) 存储在另一个数组中对应的位置上
12        dec ebx;将ebx寄存器中的值减1, 指向a2数组的下一个元素
13        mov [a2], ebx;将ebx寄存器中的值存储回a2数组中, 即更新下一个要处理的元素的地址
14        cmp ebx, 12;比较ebx寄存器中的值与十六进制数0xc (十进制数12)的大小关系
15        jge while;如果ebx寄存器中的值大于等于12, 则跳转到while标签, 继续执行循环体
16
17 your_while_end:
```

- `cmp` 指令比较两个操作数的大小。
- `jl` 指令是 "jump if less than" 的缩写，如果上一次比较结果中小于标志位（即第一个操作数小于第二个操作数），则跳转到指定的代码块。
- `call` 指令调用一个函数，并将控制转移到该函数的起始地址。
- `mov` 指令将一个操作数的值复制到另一个操作数中。
- `dec` 指令将一个寄存器或内存位置的值减 1。
- `jmp` 指令是无条件跳转，将控制转移到指定的代码块。
- `jge` 指令的含义是"jump if greater or equal"，它的功能是在比较两个值后，如果第一个值大于或等于第二个值，则跳转到指定的目标地址执行程序，否则继续顺序执行下一条指令。

3.3: 函数的实现：请编写函数 `your_function` 并调用之，函数的内容是遍历字符数组 `string`。

```
1 your_function:  
2     for i = 0; string[i] ≠ '\0'; ++i then  
3         pushad  
4         push string[i] to stack  
5         call print_a_char  
6         pop stack  
7         popad  
8     end  
9     return  
10 end
```

```
1 your_function:  
2  
3     mov eax, 0; 将eax寄存器清零  
4     mov edx, [your_string]; 将字符串的地址加载到edx寄存器  
5     mov ebx, 0; 将ebx寄存器清零  
6     cmp [edx], ebx; 比较字符串的第一个字符是否为0, 即判断是否为空字符串  
7     je your_function_end; 空字符串, 结束  
8  
9 Loop:  
10    pushad ; 压入所有寄存器的值到堆栈中  
11    mov edx, [edx + eax] ; 将edx寄存器设置为字符串中当前位置的字符  
12    push edx ; 将当前字符压入堆栈  
13    call print_a_char ; 调用打印字符的函数  
14    pop edx ; 将堆栈中的值弹出到edx寄存器  
15    popad ; 弹出所有寄存器的值  
16    inc eax ; 将eax寄存器加1, 即将指针移到字符串的下一个字符  
17  
18    cmp byte [edx + eax], 0 ; 比较字符串的下一个字符是否为0, 即判断是否到达字符串末尾  
19    jne Loop ; 如果不是, 则跳转到循环标签继续打印下一个字符  
20 your_function_end:
```

- `je` 相等则跳转
- `jne` 不等则跳转
- `pushad` 压入所有值到堆栈中
- `popad` 弹出寄存器中所有的值



dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab2/assignment

```
[ 1.462359] Segment Routing with IPv6
[ 1.464408] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[ 1.466530] NET: Registered protocol family 17
[ 1.467065] Key type dns_resolver registered
[ 1.467259] mce: Unable to init MCE device (rc: -5)
[ 1.467622] IPI shorthand broadcast: enabled
[ 1.467930] sched_clock: Marking stable (1445365828, 22334690)->(1484679777, )
[ 1.468984] registered taskstats version 1
[ 1.469055] Loading compiled-in X.509 certificates
[ 1.472828] PM: Magic number: 15:726:571
[ 1.473296] printk: console [netcon0] enabled
[ 1.473380] netconsole: network logging started
[ 1.475082] cfg80211: Loading compiled-in X.509 certificates for regulatory e
[ 1.482925] kworker/u2:0 (59) used greatest stack depth: 7180 bytes left
[ 1.491648] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[ 1.492812] platform regulatory.0: Direct firmware load for regulatory.db fa2
[ 1.493243] cfg80211: failed to load regulatory.db
[ 1.493915] ALSA device list:
[ 1.494044]   No soundcards found.
[ 1.550444] Freeing unused kernel image (initmem) memory: 680K
[ 1.553488] Write protecting kernel text and read-only data: 15604k
[ 1.553855] Run test as init process
[ 1.557663] process '/test' started with executable stack
>>> begin test
>>> if test pass!
>>> while test pass!
Mr.Chen, students and TAs are the best!
[ 1.581841] Kernel panic - not syncing: Attempted to kill init! exitcode=0x00
[ 1.582168] CPU: 0 PID: 1 Comm: test Not tainted 5.10.172 #1
[ 1.582259] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.164
[ 1.582480] Call Trace:
[ 1.583127]   dump_stack+0x54/0x68
[ 1.583201]   panic+0xaf/0x270
[ 1.583245]   do_exit.cold+0x52/0xcd
[ 1.583332]   do_group_exit+0x2a/0x90
[ 1.583382]   __ia32_sys_exit_group+0x10/0x10
[ 1.583437]   __do_fast_syscall_32+0x45/0x80
[ 1.583521]   do_fast_syscall_32+0x29/0x60
[ 1.583571]   do_SYSENTER_32+0x15/0x20
[ 1.583642]   entry_SYSENTER_32+0x9f/0xf2
[ 1.583797]   EIP: 0xb7fdb549
[ 1.584016] Code: 03 74 c0 01 10 05 03 74 b8 01 10 06 03 74 b4 01 10 07 03 76
[ 1.584349] EAX: ffffffd a EBX: 00000000 ECX: 000000fc EDX: 00000001
[ 1.584446] ESI: ffffffd c EDI: 09360380 EBP: 08221900 ESP: bfa7f300
[ 1.584536] DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b EFLAGS: 00000212
[ 1.584984] Kernel Offset: disabled
[ 1.585208] ---[ end Kernel panic - not syncing: Attempted to kill init! exi-
```

Assignment 4

```
1 org 7c00h
2
3 _DR equ 1
4 _UR equ 2
5 _UL equ 3
6 _DL equ 4
7 delay equ 400
8 double_delay equ 200
9
```

```
10 START:  
11     mov ax, cs  
12     mov es, ax  
13     mov ds, ax  
14     mov ax, 0b800h  
15     mov es, ax  
16     mov si, 0  
17     mov di, 0  
18  
19 PRINT:  
20     mov bx, name ; 将字符串的起始地址 name 存储到寄存器 bx 中  
21     mov al, [bx+si]; 将寄存器 bx+si 所指向的内存中的字节读取到寄存器 al 中  
22     cmp al, 0 ; 比较寄存器 al 中的值是否为 0, 即是否到达字符串结束符  
23     jz LOOP1 ; 如果是, 则跳转到标签 LOOP1 继续执行其他操作  
24     mov bx, 52; 将偏移地址 52 存储到寄存器 bx 中, 此处用于指定打印字符串的位置  
25     mov byte[es:bx+di], al ; 将寄存器 al 中的值存储到 es:bx+di 指向的内存中  
26     mov byte[es:bx+di+1], 1; 将数字 1 存储到 es:bx+di+1 指向的内存中, 用于指定字符属性  
27     inc si ; 将寄存器 si 的值加 1, 指向下一个字符  
28     add di, 2; 将寄存器 di 的值加 2, 指向下一个字符的输出位置  
29     jmp PRINT ; 跳转回标签 PRINT, 继续打印下一个字符  
30  
31 LOOP1:  
32     dec word[count]; 将变量“count”的值减1  
33     jnz LOOP1; 如果“count”不为零, 则跳转回循环的开头  
34  
35     mov word[count], delay; 将“delay”的值存储在变量“count”中  
36     dec word,double_count]; 将变量“double_count”的值减1  
37     jnz LOOP1; 如果“double_count”不为零, 则跳转回循环的开头  
38  
39     mov word[count], delay; 再次将“delay”的值存储在变量“count”中  
40     mov word,double_count], double_delay ; 将“double_delay”的值存储在变量  
“double_count”中  
41     ; 以下指令根据变量“RightDownUpLeft”的值使用条件跳转:  
42     mov al,1 ; 将值1移动到寄存器“al”中  
43     cmp al, byte[RightDownUpLeft]; 将“al”的值与“RightDownUpLeft”的值进行比较  
44     jz DownRight; 如果它们相等, 则跳转到标签“DownRight”  
45  
46     mov al, 2; 将值2移动到寄存器“al”中  
47     cmp al, byte[RightDownUpLeft]; 将“al”的值与“RightDownUpLeft”的值进行比较  
48     jz UpRight; 如果它们相等, 则跳转到标签“UpRight”  
49  
50     mov al, 3; 将值3移动到寄存器“al”中  
51     cmp al, byte[RightDownUpLeft]; 将“al”的值与“RightDownUpLeft”的值进行比较  
52     jz UpLeft; 如果它们相等, 则跳转到标签“UpLeft”  
53  
54     mov al, 4; 将值4移动到寄存器“al”中  
55     cmp al, byte[RightDownUpLeft]; 将“al”的值与“RightDownUpLeft”的值进行比较  
56     jz DownLeft; 如果它们相等, 则跳转到标签“DownLeft”
```

```
58     jmp $  
59  
60 ;往右下移动, 判断是否碰壁并显示字符  
61 DownRight:  
62     inc word[x]; 将变量“x”的值增加1  
63     inc word[y]; 将变量“y”的值增加1  
64     mov bx, word[x]; 将变量“x”的值移动到寄存器“bx”中  
65     mov ax, 25; 将值25移动到寄存器“ax”中  
66     sub ax, bx; 计算“25 - x”的结果并存储在寄存器“ax”中  
67     jz DownRightToUpRight; 如果“ax”的值为零, 则跳转到标签“DownRightToUpRight”  
68  
69     mov bx, word[y]; 将变量“y”的值移动到寄存器“bx”中  
70     mov ax, 80; 将值80移动到寄存器“ax”中  
71     sub ax, bx; 计算“80 - y”的结果并存储在寄存器“ax”中  
72     jz DownRightToLeft; 如果“ax”的值为零, 则跳转到标签“DownRightToLeft”  
73  
74     jmp show  
75  
76 DownRightToUpRight:  
77     mov word[x], 23; 将值23移动到变量“x”中  
78     mov byte[RightDownUpLeft], _UR ; 将值“_UR”(表示向上和向右) 移动到变量  
“RightDownUpLeft”中  
79     jmp show; 跳转到标签“show”  
80 DownRightToLeft:  
81     mov word[y], 78; 将值78移动到变量“y”中  
82     mov byte[RightDownUpLeft], _DL ; 将值“_DL”(表示向下和向左) 移动到变量  
“RightDownUpLeft”中  
83     jmp show; 跳转到标签“show”  
84  
85 ;往右上移动, 判断是否碰壁并显示字符  
86 UpRight:  
87     dec word[x]; 将变量“x”的值减1  
88     inc word[y]; 将变量“y”的值增加1  
89     mov bx, word[y]; 将变量“y”的值移动到寄存器“bx”中  
90     mov ax, 80; 将值80移动到寄存器“ax”中  
91     sub ax, bx; 计算“80 - y”的结果并存储在寄存器“ax”中  
92     jz UpRightToUpLeft; 如果“ax”的值为零, 则跳转到标签“UpRightToUpLeft”  
93  
94     mov bx, word[x]; 将变量“x”的值移动到寄存器“bx”中  
95     mov ax, 0; 将值0移动到寄存器“ax”中  
96     sub ax, bx; 计算“0 - x”的结果并存储在寄存器“ax”中  
97     jz UpRightToLeft; 如果“ax”的值为零, 则跳转到标签“UpRightToLeft”  
98  
99     jmp show; 如果以上两个条件都不满足, 则跳转到标签“show”  
100  
101 UpRightToUpLeft:  
102     mov word[y], 78; 将值78移动到变量“y”中  
103     mov byte[RightDownUpLeft], _UL ; 将值“_UL”(表示向上和向左) 移动到变量  
“RightDownUpLeft”中
```

```

104         jmp show; 跳转到标签“show”
105 UpRightToLeft:
106     mov word[x], 1; 将值1移动到变量“x”中
107     mov byte[RightDownUpLeft], _DR ; 将值“_DR”（表示向下和向右）移动到变量
108     “RightDownUpLeft”中
109
110 ;往左上移动，判断是否碰壁并显示字符
111 UpLeft:
112     dec word[x]; 将变量“x”的值减1
113     dec word[y]; 将变量“y”的值减1
114     mov bx, word[x]; 将变量“x”的值移动到寄存器“bx”中
115     mov ax, 0; 将值0移动到寄存器“ax”中
116     sub ax, bx; 计算“0 - x”的结果并存储在寄存器“ax”中
117     jz UpLeftToDownLeft; 如果“ax”的值为零，则跳转到标签“UpLeftToDownLeft”
118
119     mov bx, word[y]; 将变量“y”的值移动到寄存器“bx”中
120     mov ax, -1; 将值-1移动到寄存器“ax”中
121     sub ax, bx; 计算“-1 - y”的结果并存储在寄存器“ax”中
122     jz UpLeftToUpperRight; 如果“ax”的值为零，则跳转到标签“UpLeftToUpperRight”
123
124     jmp show; 如果以上两个条件都不满足，则跳转到标签“show”
125
126 UpLeftToDownLeft:
127     mov word[x], 1; 将值1移动到变量“x”中
128     mov byte[RightDownUpLeft], _DL ; 将值“_DL”（表示向下和向左）移动到变量
129     “RightDownUpLeft”中
130     jmp show; 跳转到标签“show”
131 UpLeftToUpperRight:
132     mov word[y], 1; 将值1移动到变量“y”中
133     mov byte[RightDownUpLeft], _UR; 将值“_UR”（表示向上和向右）移动到变量
134     “RightDownUpLeft”中
135     jmp show; 跳转到标签“show”
136
137 ;往左下移动，判断是否碰壁并显示字符
138 DownLeft:
139     inc word[x]; 将变量“x”的值增加1
140     dec word[y]; 将变量“y”的值减1
141     mov bx, word[y]; 将变量“y”的值移动到寄存器“bx”中
142     mov ax, -1; 将值-1移动到寄存器“ax”中
143     sub ax, bx; 计算“-1 - y”的结果并存储在寄存器“ax”中
144     jz DownLeftToDownRight; 如果“ax”的值为零，则跳转到标签“DownLeftToDownRight”
145
146     mov bx, word[x]; 将变量“x”的值移动到寄存器“bx”中
147     mov ax, 25; 将值25移动到寄存器“ax”中
148     sub ax, bx; 计算“25 - x”的结果并存储在寄存器“ax”中
149     jz DownLeftToUpperLeft; 如果“ax”的值为零，则跳转到标签“DownLeftToUpperLeft”
150
151     jmp show; 如果以上两个条件都不满足，则跳转到标签“show”

```

```
150
151 DownLeftToDownRight:
152     mov word[y], 1; 将值1移动到变量“y”中
153     mov byte[RightDownUpLeft], _DR ; 将值“_DR”（表示向下和向右）移动到变量
154     “RightDownUpLeft”中
155     jmp show; 跳转到标签“show”
156 DownLeftToUpLeft:
157     mov word[x], 23; 将值23移动到变量“x”中
158     mov byte[RightDownUpLeft], _UL ; 将值“_UL”（表示向上和向左）移动到变量
159     “RightDownUpLeft”中
160     jmp show; 跳转到标签“show”
161
162 ;在屏幕上显示字符
163 show:
164     xor ax, ax ; 将寄存器“ax”清零
165     mov ax, word[x]; 将变量“x”的值移动到寄存器“ax”中
166     mov bx, 80; 将值80移动到寄存器“bx”中
167     mul bx; 计算“x * 80”的结果并存储在寄存器“ax”中
168     add ax, word[y]; 将变量“y”的值加到寄存器“ax”中
169     mov bx, 2; 将值2移动到寄存器“bx”中
170     mul bx; 计算“ax * 2”的结果并存储在寄存器“ax”中
171     mov bx, ax; 将寄存器“ax”的值移动到寄存器“bx”中
172     mov ah, byte[color] ; 将变量“color”的值移动到寄存器“ah”中
173     mov al, byte[char] ; 将变量“char”的值移动到寄存器“al”中
174     mov [es:bx], ax ; 将寄存器“ax”的值存储到屏幕内存中
175
176     inc byte[char]; 将变量“char”的值加1
177     cmp byte[char], 'z'+1; 比较变量“char”的值是否超过字符“z”的ASCII码值
178     jnz keep; 如果没有超过，则跳转到标签“keep”
179     mov byte[char], '0'; 如果超过了，则将变量“char”的值设置为字符“0”的ASCII码值
180
181 keep:
182     inc byte[color]; 将变量“color”的值加1
183     cmp byte[color], 0x10; 比较变量“color”的值是否超过16，即0x10的十六进制值
184     jnz LOOP1; 如果没有超过，则跳转到标签“LOOP1”
185     mov byte[color], 0x40 ; ;如果超过了，则将变量“color”的值设置为字符“@”的ASCII码值，表
186     示循环显示不同样式的字符
187     jmp LOOP1; 跳转到标签“LOOP1”继续循环
188
189 end:
190     jmp $
191
192 count dw delay ;一层延迟
193 double_count dw double_delay ;二层延迟
194 RightDownUpLeft db _DR ;方向变量
195 color db 0x15 ;样式（颜色）变量
196 x dw 0 ;横坐标
197 y dw 0 ;纵坐标
198 char db '1' ;要显示的字符
```

```

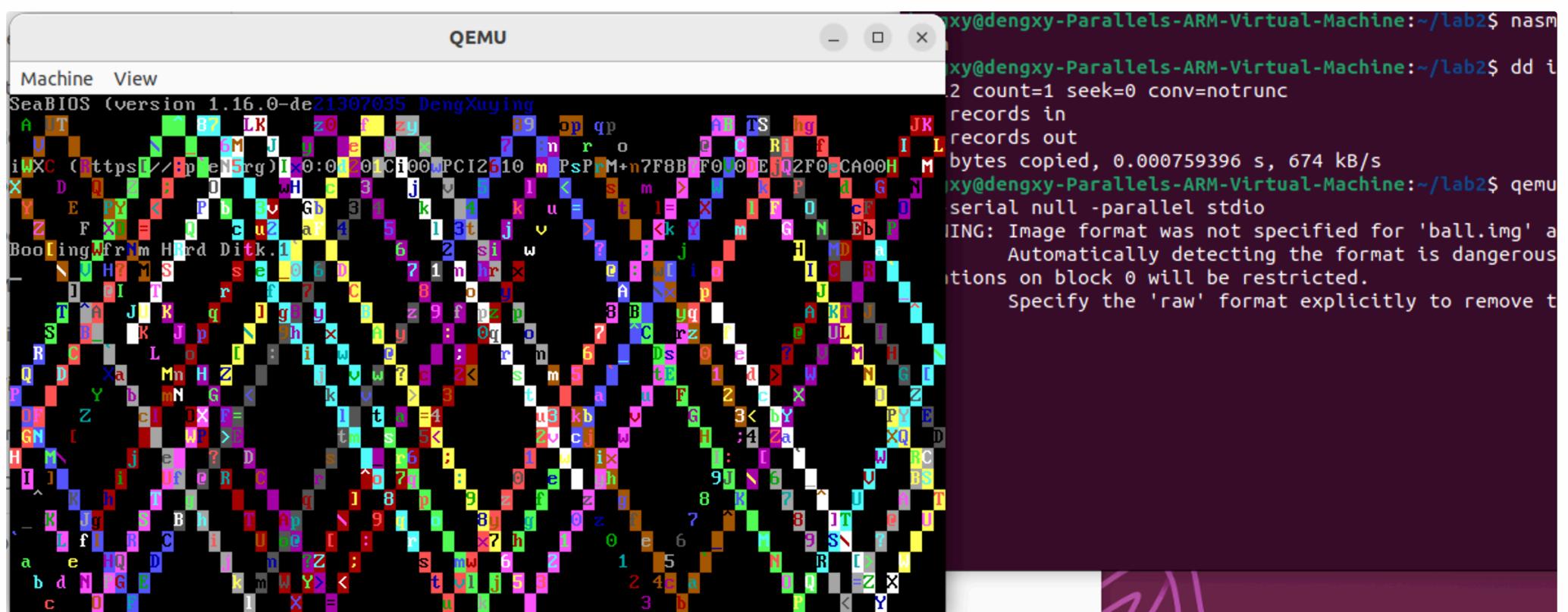
196     name db '21307035 DengXuying', 0      ;学号姓名
197
198     times 510-($-$) db 0
199     dw 0aa55h

```

```

1 gedit ball.asm
2 nasm -f bin ball.asm -o ball.bin
3 qemu-img create ball.img 10m
4 dd if=ball.bin of=ball.img bs=512 count=1 seek=0 conv=notrunc
5 qemu-system-i386 -hda ball.img -serial null -parallel stdio

```



3、总结

- 熟悉了x86汇编
- 熟悉了对 .asm 编译
- 熟悉写入虚拟磁盘、使用qemu启动虚拟磁盘等内容