



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 2 实验入门_____

专业名称：_____计算机科学与技术(超级计算方向) _____

学生姓名：_____林天皓_____

学生学号：_____18324034_____

实验地点：_____

实验成绩：_____

报告时间：_____2021. 3. 20_____

Assignment 1

1.1 复现 example 1。说说你是怎么做的，并将结果截图

直接将示例代码复制，需要注意的是需要将示例代码第一行的 `org 7c00` 后面添加一个 `h`

```
nasm -g -f bin mbr.asm -o mbr.bin #nasm 编译
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc #创建镜像文件
qemu-system-i386 -hda hd.img -serial null -parallel stdio #qemu 运行
```

运行结果如下,左上角出现蓝色的 Hello world

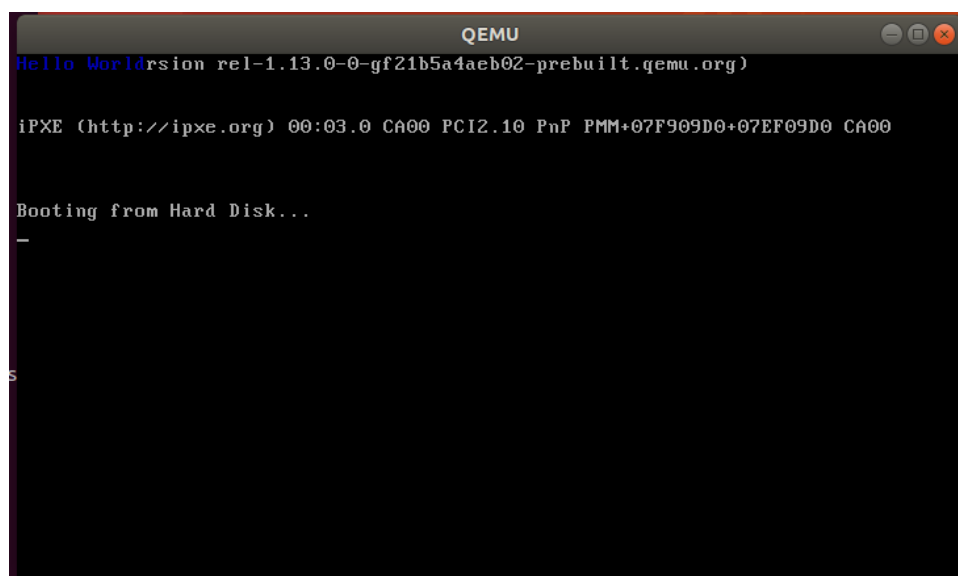


图 1-左上角出现蓝色的 Hello world

1.2 请修改 example 1 的代码，使得 MBR 被加载到 0x7C00 后在(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。说说你是怎么做的，并将结果截图。

颜色设置：这里我们设置文字颜色前景色为白色，背景色为红色，我们只需要修改 `ah` 的值为 `0b0100_0111` 换算为十六进制则为 `0x47`

位置设置：由于 qume 模拟器的屏幕为 80x25，根据换算规则一个(x,y)的坐标点，输出显存偏移量为 $2 \times (y \times 80 + x)$ ，也就是学号的第一位为 $2 \times (12 \times 80 + 12)$ ，好在 asm 汇编可以添加算式我们不需要把这个值算出来。

完整代码 <https://pastebin.ubuntu.com/p/jSz559QNbD/>

其中显示学号的重点代码如下，在其中运用 bp 做为偏移量寄存器，每次加 2 复用一部分代码。

```
mov ah, 0x47 ;改颜色
mov bp, 2*(12*80+12);首地址
mov al, '1'
mov [gs:bp], ax
add bp, 2
mov al, '8'
mov [gs:bp], ax
add bp, 2
mov al, '3'
mov [gs:bp], ax
add bp, 2
mov al, '2'
mov [gs:bp], ax
add bp, 2
mov al, '4'
mov [gs:bp], ax
add bp, 2
mov al, '0'
mov [gs:bp], ax
add bp, 2
mov al, '3'
mov [gs:bp], ax
add bp, 2
mov al, '4'
mov [gs:bp], ax
add bp, 2
```

显示效果如下：(12,12) 的位置开始红底白字显示学号

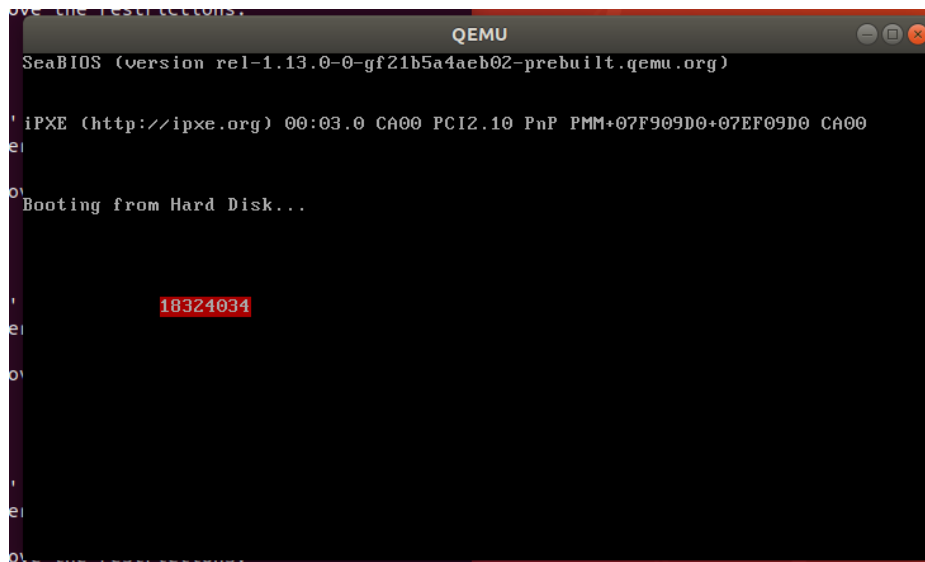


图 2-(12,12)出现红底白字显示 18324034

Assignment 2 实模式中断

2.1 实模式中断输出学号

请修改 1.2 的代码，使用实模式下的中断来输出你的学号，可以参考 [\[https://blog.csdn.net/lindorx/article/details/83957903\]](https://blog.csdn.net/lindorx/article/details/83957903)。说说你是怎么做的，并将结果截图。

根据参考文献，实模式下的中断向量表位于 0x0~0x3ff 中，这里我们采用 int 0 中断，因此只需要将主函数的地址写入主存地址的前两个字节。其他的部分不需要修改，直接使用参考文献中的代码即可。

完整代码 <https://pastebin.ubuntu.com/p/RJY5qQMx7h/>

部分重点代码如下

```
int_init:
    mov [es:2],bx
    mov bx,main_put;将 main_put 写入中断向量表
    mov [es:0],bx
    int 0
```

<code>jmp \$; 死循环</code>
<code>main_put:</code>
<code>push msg ;msg 为字符串首地址</code>
<code>push 8 ;字符数</code>
<code>push 0x4700 ;颜色设定为红底白字</code>
<code>call print</code>
<code>iret</code>

其中，需要将地址 0 写入 `main_put` 的地址，同时需要将内存地址 2 清零，然后通过 `int 0` 调用，最后在 `main_put` 中需要通过 `iret` 从中断态返回。

在运行到 `mov [es:0],bx` 时我们通过 `gdb` 中的 `x/1a 0` 命令查看位于第 0 个地址中的值。

下图中，我们将要执行的下一条指令是 `int 0` 的软中断

```
(gdb)
0x00007c21 in ?? ()
1: x/10i $pc
=> 0x7c21: int    $0x0
    0x7c23: jmp    0x7c23
    0x7c25: push   $0x86a7c00
    0x7c2a: push   $0x1e84700
    0x7c2f: add    %cl,%bh
    0x7c31: pusha
    0x7c32: push   %ebp
    0x7c33: mov    %esp,%ebp
    0x7c35: mov    $0xc08eb800,%eax
    0x7c3a: mov    0x18(%esi),%edi
```

图 3-进入中断前

下图中，通过 `gdb` 展示了当前内存地址为 0 内的值，为 `0x7c25`，联合上图和下图可得知 `0x7c25` 为 `main_put` 函数的首地址。

```
(gdb) x/1a 0
0x0: 0x7c25
```

图 4 内存地址 0 中存放中断向量表

```
24  √ int_init:
25      mov [es:2],bx
26      mov bx,main_put
27      mov [es:0],bx
28      int 0
29      jmp $ ; 死循环
30
31  √ main_put:
32      → push msg      ;msg为字符串首地址
33      push 8          ;字符数
34      push 0x4700     ;颜色设定为
35      call print
36      iret
37      # jmp $ ; 死循环
```

图 5-进入对应 msg 中断函数

由以上两个图中可以得知，`main_put` 中第一条指令的地址位于 `0x7c25` 下面我们执行 `ni`

```
1: x/10i $pc
⇒ 0x7c28:      push    $0x8
   0x7c2a:      push    $0x1e84700
   0x7c2f:      add     %cl,%bh
   0x7c31:      pusha
   0x7c32:      push    %ebp
   0x7c33:      mov     %esp,%ebp
   0x7c35:      mov     $0xc08eb800,%eax
   0x7c3a:      mov     0x18(%esi),%edi
   0x7c3d:      mov     0x16(%esi),%ecx
   0x7c40:      mov     0x14(%esi),%edx
```

图 6-成功进入 msg 函数

pc 已经跳转到 `main_put` 函数内。整体运行效果如下，输出学号 18324034。

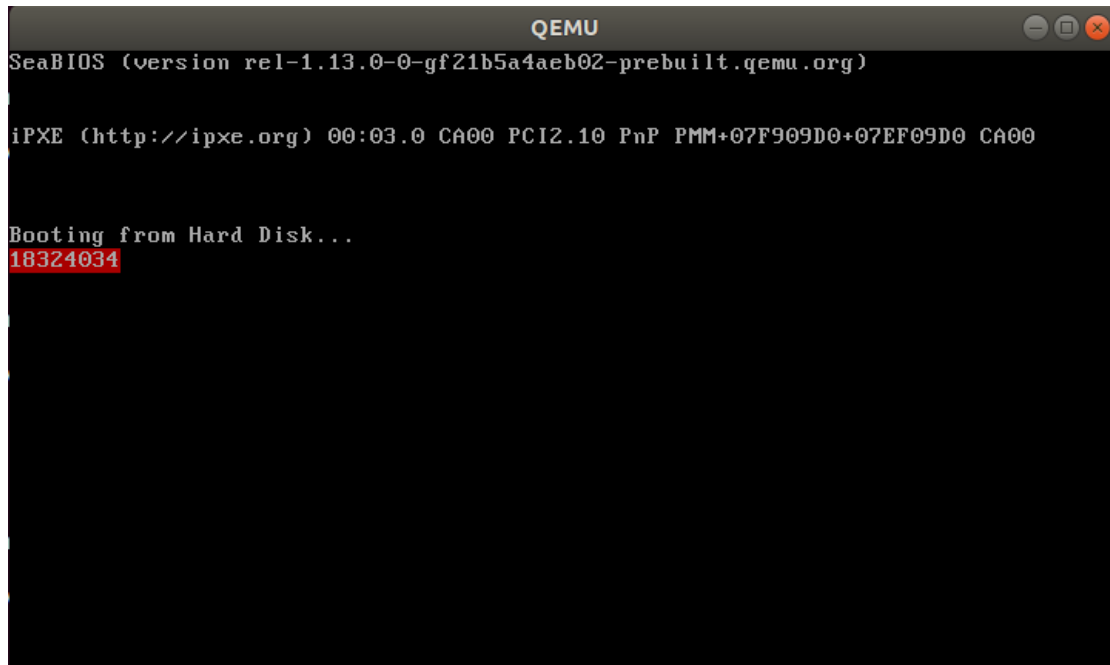


图 7-输出学号效果

2.2 中断实现光标的位置获取和光标的移动

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，可以参考[\[https://blog.csdn.net/lindorx/article/details/83957903\]](https://blog.csdn.net/lindorx/article/details/83957903)。说说你是怎么做的，并将结果截图。

在 2.1 中我们已经学会了自定义中断，这里继续按照其中的方法，将 `get_pos` 和 `set_pos` 写入中断向量表。同时需要将 `get_pos` 中的 `ret` 改为 `iret` 指令

完整代码 <https://pastebin.ubuntu.com/p/g7mTNmsGBn/>

重点代码：下图为将 `main_put`, `set_pos`, `get_pos` 写入中断向量表的初始化过程。

```
int_init:
    mov [es:2],bx
    mov [es:6],bx
    mov [es:10],bx
    mov bx,main_put
    mov [es:0],bx
    mov bx,set_pos
    mov [es:4],bx
```

```
mov bx,get_pos
mov [es:8],bx
int 0
jmp $; 死循环
```

通过这种方式，我们就可以通过 `int 1` 来实现原来 `call set_pos` 的功能，通过 `int 2` 来实现原来 `call get_pos` 的功能。如下移动光标的函数已经是通过 `int 1` 中断实现。

```
mov_cursor:
inc ax
int 1
```

运行程序，输出正常。

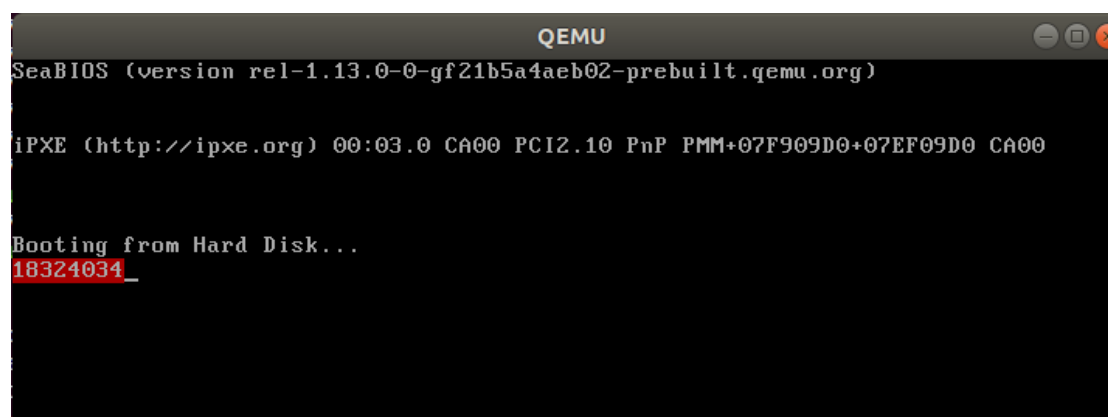


图 8-输出学号效果

2.3 键盘中断实现键盘输入并回显

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，可以参考 [\[https://blog.csdn.net/deniece1/article/details/103447413\]](https://blog.csdn.net/deniece1/article/details/103447413)。关于键盘扫描码，可以参考[\[http://blog.sina.com.cn/s/blog_1511e79950102x2b0.html\]](http://blog.sina.com.cn/s/blog_1511e79950102x2b0.html)。说说你是怎么做的，并将结果截图。

2.3.1 一个简单的方法

最简单的方法我们只需要 6 行即可完成。

```
;cursor_echo.asm
org 7c00h
[bits 16]
echo:
    mov ah,0
    int 0x16
    mov dl,al
    mov ah,0x0e
    int 0x10
    jmp echo

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

读取键盘输入：通过设置 `ah=0` 然后调用 `16h` 的输入中断实现读取键盘输入的字符

输出字符：通过设置 `ah=0x0e` 然后调用 `10h` 的输入中断实现输出字符到屏幕上

效果加截图如下

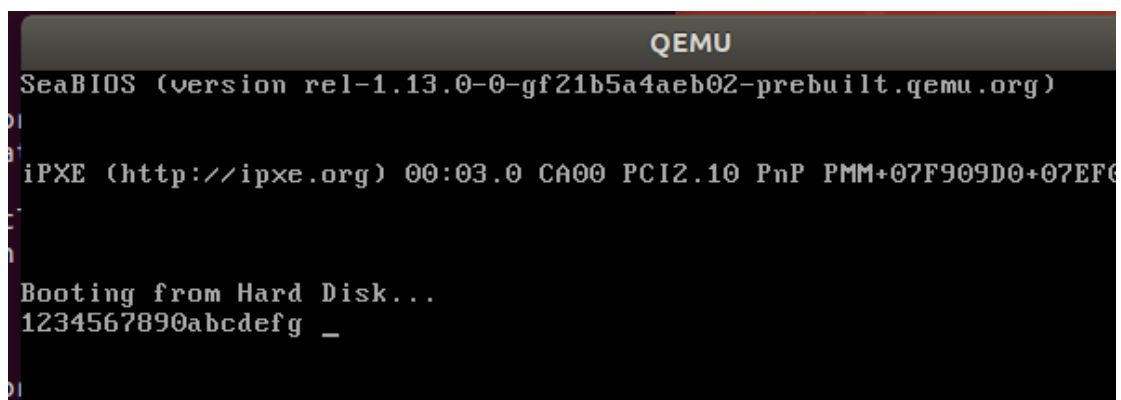


图 9-键盘输入回显

以上的实现方式缺点较多，在输入退格键的时候不能删除已经输出的字符，同时不能自定义输入键盘字符对应的处理方式。

2.3.2 一个更完善的方法

不使用 `int 10h` 做为输出字符的手段，而和显示学号任务中一样通过 `judeg` 函数通过操作显存来显示字符。由于原程序中使用 `ax` 储存光标的位置，而读取字符时候我们需要改变 `ax` 的值，因此，修改文中储存光标的寄存器为 `bp` 同时在其他需要用到 `ax` 的地方通过 `push` 和 `pop` 的方式使用栈暂时储存。

在这个程序中

1. 可以使用退格键删除已经输出的字符
2. 退格到开头之后也可以让光标返回上一行。
3. 对于非可输出字符不会输出一个空白

完整代码 <https://pastebin.ubuntu.com/p/83HKknzdWr/>

程序实现的主要代码(节选)如下

```
int_init:
    mov dh,0x47
    mov bp,es
    mov [es:2],bx
    mov [es:6],bx
    mov [es:10],bx
    mov bx,main_put
    mov [es:0],bx
    mov bx,set_pos
    mov [es:4],bx
    mov bx,get_pos
    mov [es:8],bx
    mov ax,0xb800
    mov es,ax
    int 0
    jmp $; 死循环
```

main_put:
call echo
iret
echo:
call get_key;读取键盘输入
call judeg;输出键盘字符以及处理光标控制
jmp echo;无限循环
get_key:
mov ah,0
int 0x16
mov dl,al
ret

运行结果如下。

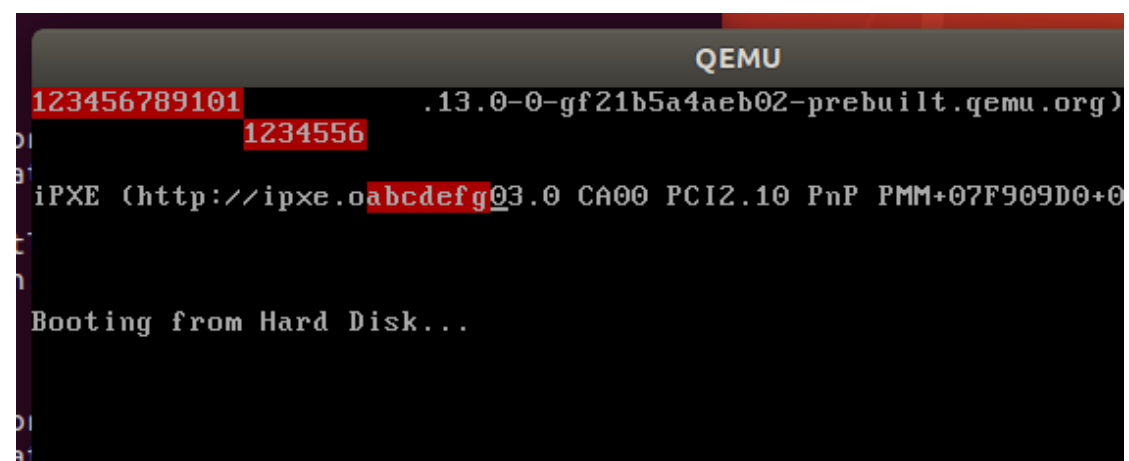


图 10-键盘输入回显（功能完善版）

上图中第一行后的黑色空白通过退格键实现，第二行以及第四行通过 ctrl+enter 键换行实现

Assignment 3 汇编

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。

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

实现逻辑:

1. 将 a1 存入 ebx
2. 判断 ebx 的值进入对应分支
3. 小于 12 分支 : eax=ebx 右移一位+1
4. 小于 24 分支 : eax=ebx-24, ebx 取负数, ebx 乘以 eax
5. else 分支 : eax=eax 左移 4 位
6. 将 eax 写回 a1

完整代码如下

```
your_if:
    mov ebx,[a1]
    cmp ebx,12
    jl your_if_st12
    cmp ebx,24
    jl your_if_st24
    jmp your_if_other
your_if_st12:
    shr ebx,1
    mov eax,ebx
    inc eax
    jmp your_if_end
your_if_st24:
    mov ecx,ebx
    sub ebx,24
    neg ebx
    imul ebx,ecx
```

```

    mov eax,ebx
    jmp your_if_end
your_if_other:
    shl ebx,4
    mov eax,ebx
    jmp your_if_end
your_if_end:
    mov [if_flag],eax

```

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。

```

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

```

实现逻辑如下

1. a2 存入 eax
2. ebx 存入 while_flag 首地址
3. ecx=eax-11 即储存循环次数
4. 进入 loop 循环，在循环中 保护寄存器，调用 my_random
5. 将 al 存入[ecx+ebx-1],即为*(a2-11-1)
6. eax-1 并写回 a2
7. 循环直到 ecx=0

实现代码如下：

```

your_while:
    mov eax,[a2]
    mov ebx,[while_flag]

```

```

mov ecx,eax
sub ecx,11
my_loop:
    push eax
    push ecx
    push ebx
    call my_random
    pop ebx
    pop ecx
    mov [ecx+ebx-1],al ;不能赋值 eax
    pop eax
    dec eax
    mov [a2],eax
    loop my_loop

```

【小心错误!】需要注意的是这里需要特别注意写入的方式，一定要是单字节寄存器 al。由于这里是倒序写入，如果直接赋值 eax 会导致后续单元的值被覆盖，造成只有第一个值是正确。

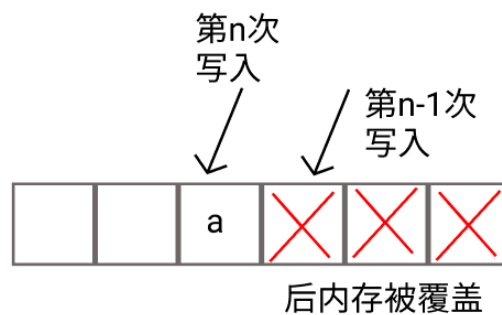


图 11-通过 eax 赋值的错误

下图是一个错误示范：

```

➔ assignment3 git:(main) x make run
>>> begin test
>>> if test pass!
while_flag 1
random_buffer lwn
>>> test failed

```

图 12-错误效果：只有第一个值正确

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`。

```
your_function:
for i = 0; string[i] != '\0'; ++i then
pushad
push string[i] to stack
call print_a_char
pop stack
popad
end
return
end
```

有了前面两题的基础，这里的的大部分内容已经比较明确。实现逻辑如下

1. `edx` 储存 `your_string` 首地址，`ecx` 作为循环变量置 0
2. 进入循环，循环中读取字符，判断字符是否等于 0，是否跳出
3. 对 `ax` 压栈，调用 `print_a_char`
4. 跳转到第 2 步

代码实现如下：

```
your_function:
mov ecx,0
mov edx,[your_string];储存 string 首地址
func_loop:
mov al,[ecx+edx]
mov ah,0
cmp ax,0
je func_end
pushad
push ax;不能换为 al
call print_a_char
pop ax
popad
```

```
add ecx,1
jmp func_loop
func_end:
ret
```

由于 x86 指令的限制，我们不能只压栈 al,只能对 ax 整体压栈。

Assignment 3 实验测试结果

测试三组数据，为了方便观察将随机字符串 random_buffer 和 while_flag 字符串

输出查看

第一组测试

a1=24 a2=14

```
→ assignment3 git:(main) x make run
>>> begin test
>>> if test pass!
while_flag lwn
random_buffer lwn
>>> while test pass!
Mr.Chen, students and TAs are the best!
```

图 13-测试结果 1

生成三个长度的字符串，结果正确

第二组测试

a1=16 a2=18

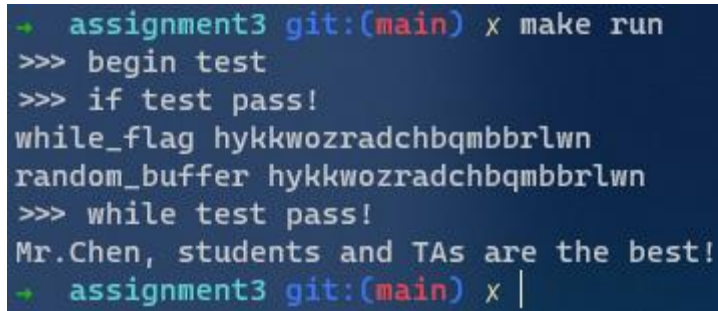
```
→ assignment3 git:(main) x make run
>>> begin test
>>> if test pass!
while_flag mbbrrlwn
random_buffer mbbrrlwn
>>> while test pass!
Mr.Chen, students and TAs are the best!
→ assignment3 git:(main) x |
```


图 13-测试结果 2

生成 7 个长度的字符串，结果正确

第三组测试

a1=7 a2=32



```
→ assignment3 git:(main) x make run
>>> begin test
>>> if test pass!
while_flag hykkwozradchbqmbbbrlwn
random_buffer hykkwozradchbqmbbbrlwn
>>> while test pass!
Mr.Chen, students and TAs are the best!
→ assignment3 git:(main) x |
```

图 14-测试结果 3

生成 21 个长度的字符串，结果正确

综上，通过三组测试数据，测试均正确。

Assignment 4

程序流程图如下：

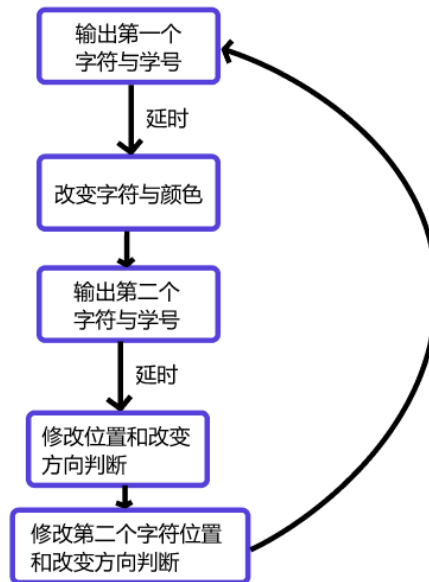


图 15-paint 流程图

完整代码 <https://pastebin.ubuntu.com/p/PjFfRmQmkW/>

其中寄存器使用与初始化如下：

```

;初始化
;字符 1 位置 bh,bl
;字符 2 位置 ch,cl
;字符与颜色 al,ah
;当前方向 dh,dl
;方向 00: 右下, 方向 01: 右上, 方向 10: 左下, 方向 11: 左上
mov bh,9;字符 1 初始行
mov bl,0;字符 1 初始列
mov ch,16;字符 2 初始行
mov cl,79;字符 2 初始列
mov dh,0;字符 1 初始方向
mov dl,3;字符 2 初始方向
call clear_screen
  
```

1.主函数 print:

同流程图

```

print:
    pushad
    mov dh,0
    mov dl,bh
    imul dx,80
    mov bh,0
    add dx,bx
    imul dx,2
    mov bp,dx
    mov [gs:bp],ax
    popad
    call print_id ;输出学号
    call pause ;延时函数
    call change_text ;改变输出的字符和颜色

    pushad
    mov dh,0
    mov dl,ch
    imul dx,80
    mov ch,0
    add dx,cx
    imul dx,2
    mov bp,dx
    mov [gs:bp],ax
    popad
    call print_id ;输出学号
    call pause ;延时函数
    call change_text ;改变输出的字符和颜色
    call change_b_position ;改变第一个字符的位置
    call change_c_position ;改变第二个字符的位置
    jmp print ;回到开头继续循环

```

2.改变字符颜色和数字函数 change_text:

当字符超过‘9’的时候需要重新设置为 0

```

change_text: ;改变颜色和数字
    inc ah
    inc al
    cmp al,'9'
    jge set_al_zero
    jmp change_text_end
set_al_zero: ;超过 9 重新设置为 0

```

```
    mov al,'0'  
change_text_end:  
    ret
```

3.改变字符 1 位置函数:

首先判断当前需要移动的方向，进入对应分支，改变之后判断是否遇到边界需要通过 xor 位运算转换方向。对于字符 2 同理，此处不再赘述

```
change_b_position:  
    cmp dh,0  
    je right_down_b ;方向右下  
    cmp dh,1  
    je right_up_b ;方向右上  
    cmp dh,2  
    je left_down_b ;方向左下  
    cmp dh,3  
    je left_up_b ;方向左上  
    right_down_b:  
        inc bh  
        inc bl  
        jmp check_bl  
    right_up_b:  
        dec bh  
        inc bl  
        jmp check_bl  
    left_down_b:  
        inc bh  
        dec bl  
        jmp check_bl  
    left_up_b:  
        dec bh  
        dec bl  
        jmp check_bl  
    check_bl:  
        cmp bl,0  
        jle change_left_right  
        cmp bl,79  
        jge change_left_right  
        jmp check_bh  
    change_left_right: ;改变方向左右  
        xor dh,0x02
```

```

check_bh:
    cmp bh,0
    jle change_up_down
    cmp bh,24
    jge change_up_down
    jmp change_end
change_up_down:
    xor dh,0x01 ;改变方向上下
change_end:
    ret

```

4.延时函数 pause:

通过一个 5000 次循环嵌套 10 次循环函数延时

```

pause: ;延时函数
    pushad
    mov ecx,delay
in_loop:
    call pause_2
    loop in_loop
in_end:
    popad
    ret
pause_2:
    pushad
    mov ecx,delay2
in_loop_2:
    loop in_loop_2
in_end_2:
    popad
    ret

```

5.显示姓名和学号函数 print_id:

```

print_id:
    pushad
    mov si,stuid ; 显示姓名和学号
    mov di,64 ;显示位置
    mov cx,16 ;姓名与学号的长度
print_char:
    mov ah,0x47
    mov al,[si]

```

```
inc si
mov word [gs:di],ax
add di,2
loop print_char
popad
ret
```

6.清屏函数 clear_screen:

```
clear_screen:
pushad
mov ax,0
mov ecx,2000 ;循环 80x25=2000 次
mov di,0
clc_loop:
    mov word [gs:di],ax
    add di,2
    loop clc_loop
popad
ret

stuid db 'Avarpow 18324034'
times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

实验结果

运行效果如下： 开场清屏，学号显示在最上方，双向发射，边界反弹。

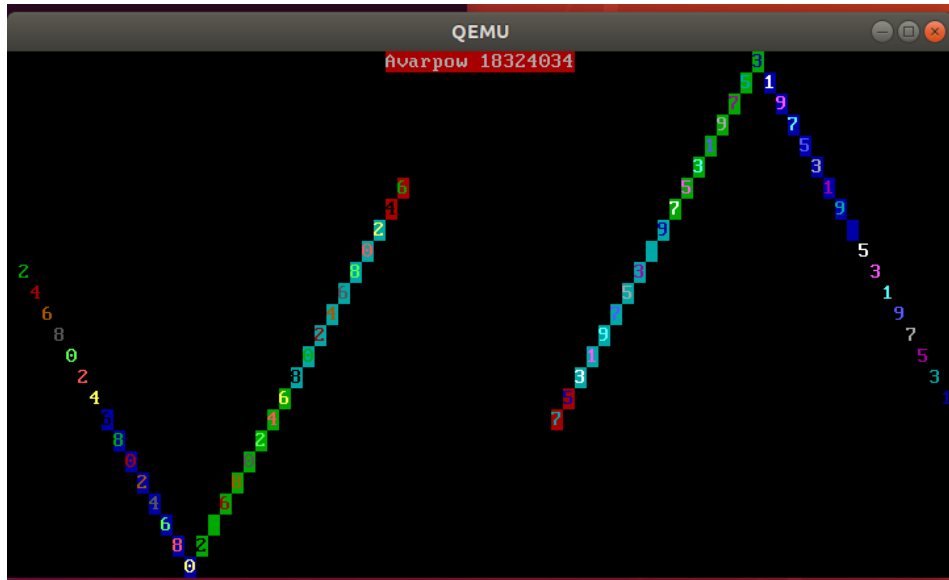


图 16-运行开始效果



图 17-运行中途效果

实验感想

在本次实验 assignment 1 中，了解了 x86 架构中的启动过程，执行的第一条用户指令地址为 0x7c00，因此在使用 gdb debug 时候首先需要设置 0x7c00 的断点。了解了 x86 中 bios 的显存地址 0xb800,并能输出各种颜色的字符。

assignment 2 中，了解了 x86 实模式中的软中断使用，需要将对应中断程序的入口地址存放在内存地址 0x00-0x3f 中对应的地址中。了解了 x86 实模式中如何使用默认中断 0x16 接收键盘输入和使用中断 0x10 输出字符。

assignment 3 中，学会了如何将伪代码翻译成为 x86 的汇编代码。在编写的过程中，由于 x86 对于寄存器的使用有一些约定俗成的规定经常导致编译失败，同时寄存器太少，写起来很麻烦，需要很好的规划。对比起来，以前写 mips 汇编的时候有充足的寄存器则容易得多。但是 x86 也有一个好，作为一个 CISC 指令集，它有着一个操作数可以来自主存的特点，不用向 RISC 指令集一样需要频繁的使用 sw 和 lw 指令完成寄存器与主存之间的数据同步。

在 assignment 3 中，有两点在开始的时候感到困惑，一是在汇编中引用 C++ 语言中定义的指针到底该怎么定义。一开始直接使用 your_string 当作首地址实际上是不正确的，需要通过一次取地址之后的[your_string],才是真正的字符首地址。二是在 C++语言中进行了有参数函数的调用，这个参数通过什么样的方式传递给我们所编写的汇编函数？通过查阅参考资料，了解到通过函数中的参数自右向左压栈传递参数，通过 eax 寄存器返回值。

assignment 4 中，得益于在之前对于 X86 汇编的初步熟悉，先合理分配寄存器，考虑流程实现，按照自顶向下一步一步实现就可以完成了，好在所需的寄存器不多，没有需要使用栈传递参数，否则该实验的难度就要大大提升了。

由于每次编译，导入镜像，qemu 启动需要三条语句较麻烦，通过编写脚本快速调用


```
#run.sh
```

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

我们就可以通过 `./run.sh xxx` 来快速对 `xxx.asm` 执行三条语句

参考资料

<https://www.itzhai.com/assembly-int-10h-description.html>

<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html#instructions>

https://en.wikipedia.org/wiki/X86_calling_conventions