实验报告 实验五

姓名：王钦　学号：13349112　班级：计科二班

实验目的

1. 理解系统调用的实现方法。
2. 实现原型操作系统中一些基本的系统调用。
3. 设计并实现一测试系统调用的用户程序，利用系统调用实现用户界面和内部功能。
4. 在原型操作系统上建立一个初步C语言开发环境，理解操作系统与高级语言之间的关系。

实验内容

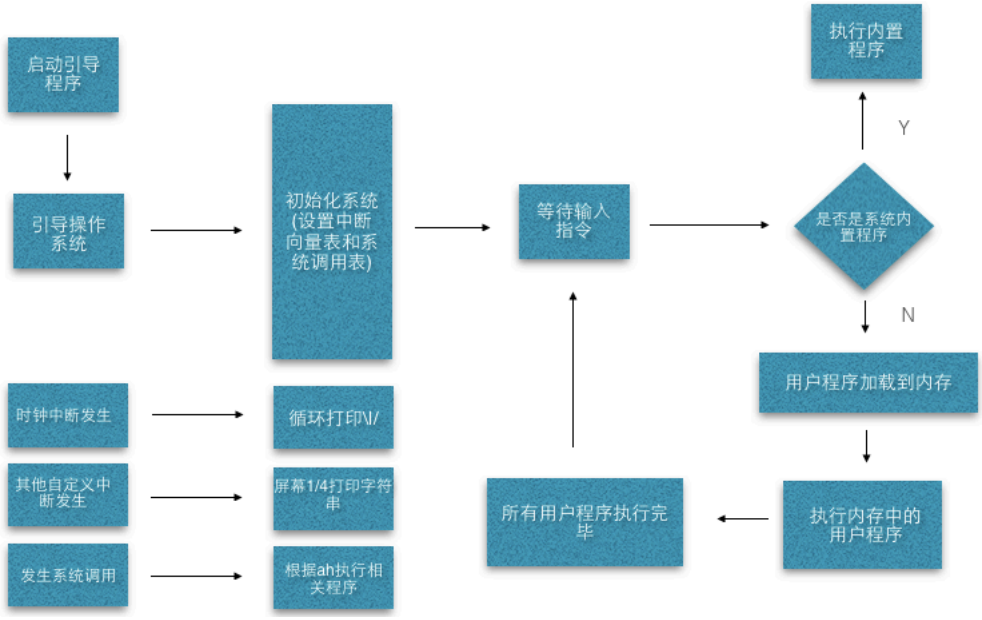
在实验四的基础上，进化你的原型操作系统，增加下列操作系统功能：
(1)参考下面的系统调用功能表，增加一些其他功能
系统调用表（局部设计）
(2)扩展MYOS内核，实现上表中的所有(包括你增加的)系统调用，并开发一个用户程序，展示这些系统调用的使用效果。
(3)设计一个C程序库，封闭getch(),gets(),putch(),puts(),scanf()和printf()等利用系统调用实现的细节，并参考下面程序，开发一个用户程序，测试这些函数功能。

```
include stdio.h
main(){
        char ch,str[80];
        int a;
        getch(&ch);
        gets(str);
        scanf “(a=%” d,&a);
        putch(ch);
        puts(str);
        printint “(ch=%c, a=%d, str=%” s, ch, a, str);
}
```

实验平台

gcc+ld+nasm+Linux+vim

算法流程图

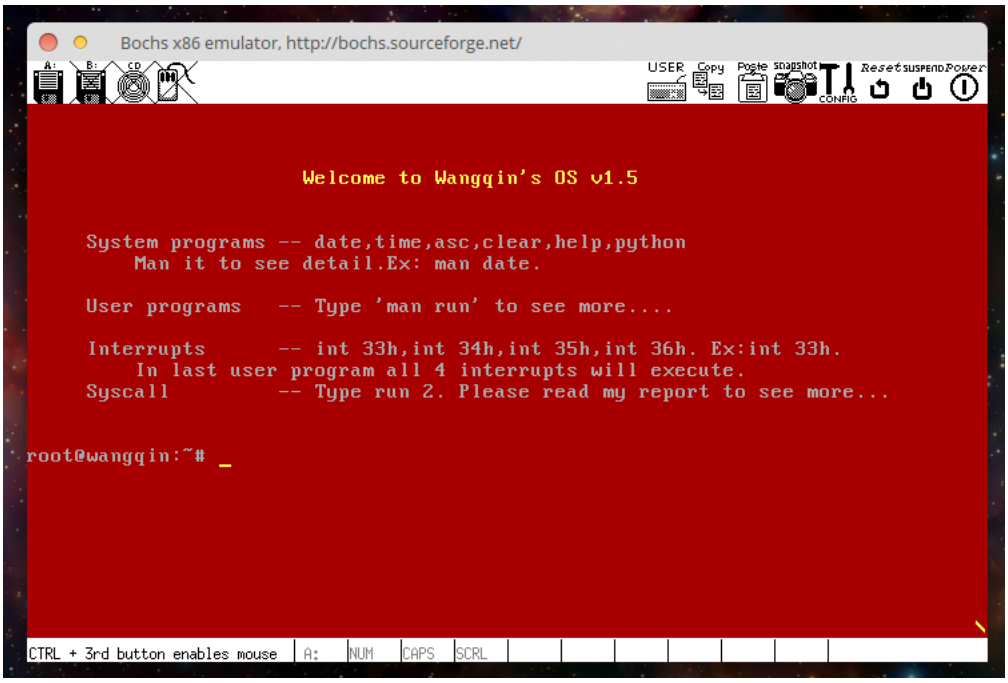


实验步骤及效果

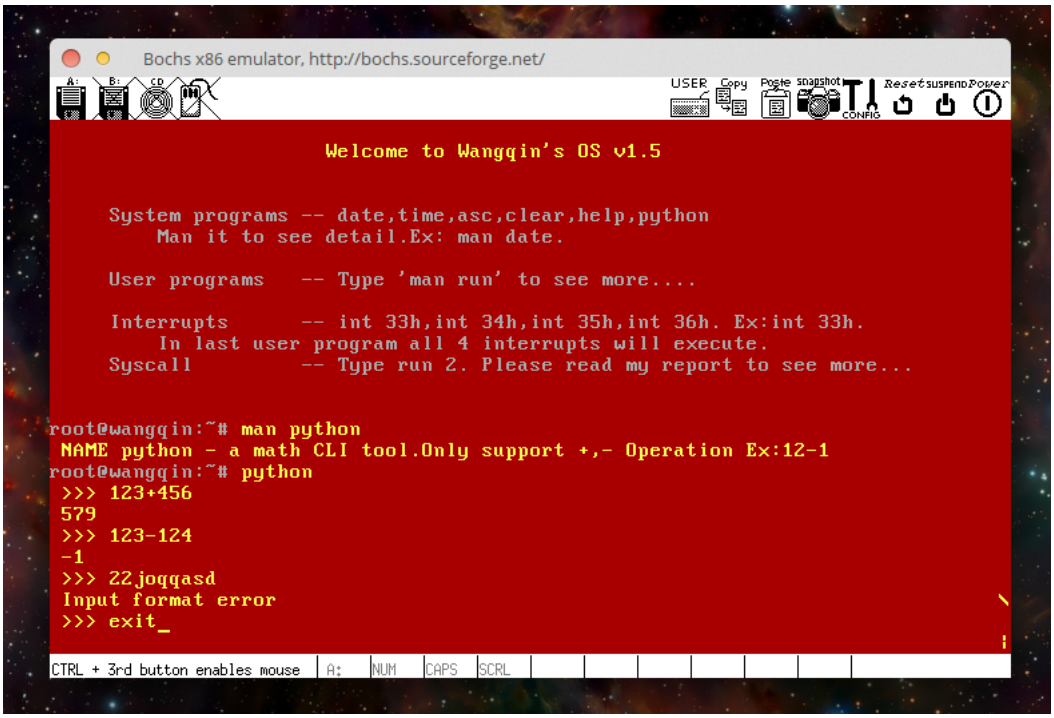
1. 编辑修改ASM 文件,和C文件

2. 使用make命令配合`makefile`文件进行编译

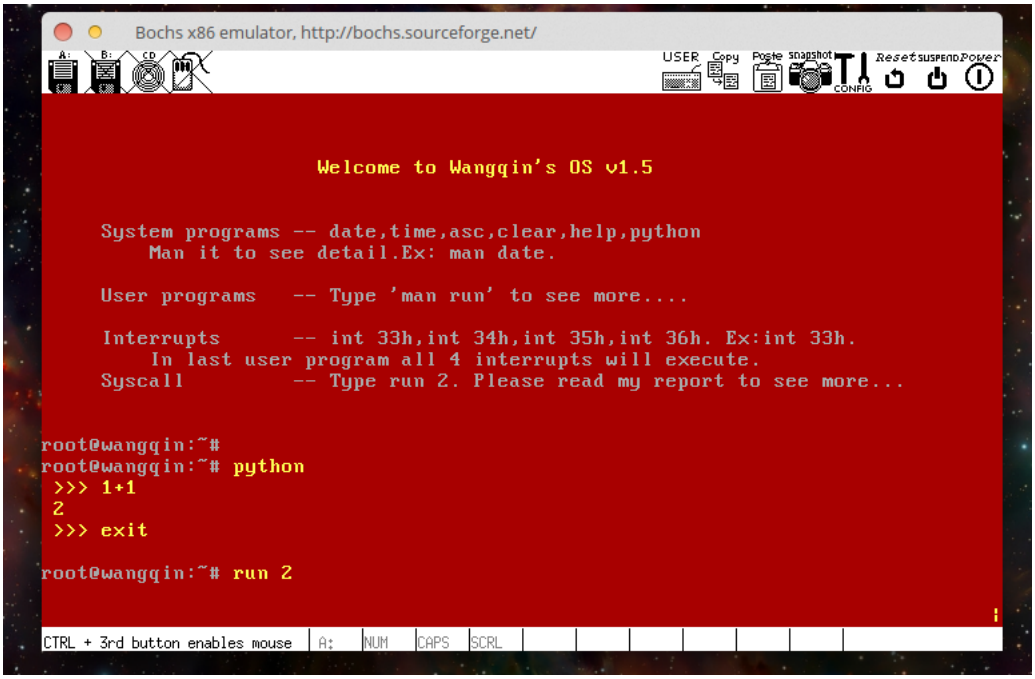
3. 运行`bochs or vmware`虚拟机进行测试,进入后所看到的欢迎界面



4. 这次比实验四多了一个系统内置程序python,我们可以执行man python查看他的作用，主要功能是输入一个数学表达式然后返回表达式的结果，类似python命令行的作用。但目前这个工具只支持加法和减法且只能有两个操作数（其实主要为了展示ah=3,4将字符串转为数值和将数值转为字符串的系统调用效果），我这里仿照linux系统的系统调用，设置 int 80h 为所有系统调用的入口。可以看到图中分别输入加法减法，返回计算结果。如果输入的不符合格式就会返回错误提示。最后输入exit退出python命令行工具。



5. 接下来测试ah等于0，1，2，5的系统调用(显示OUCH，字母大小写变化等)，输入 `run 2` ,运行第二个用户程序，



下面是第二个用户程序的代码：

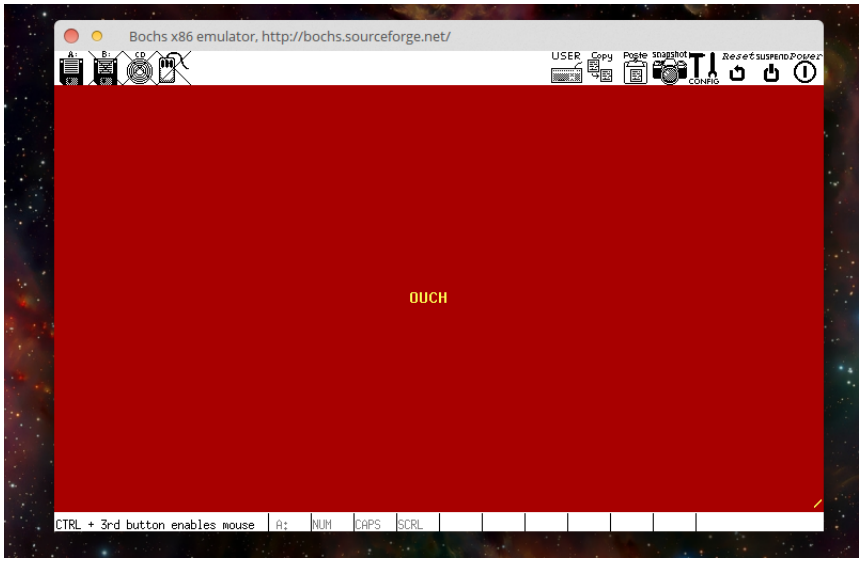
```
org 0x1000
;org 0x100

;#0
mov ah,0
int 80h

;LISTEN_EXIT——
listen0:
        mov ah,0
        int 16h


;#2
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,2
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h

;#1
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,1
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h



;#5
mov ax,cs
mov ds,ax
mov es,ax
```

```
42
43  mov cx,0317h   ;position
44  mov ah,5
45  mov dx,msg
46  int 80h
47
48
49  ;LISTEN_EXIT——
50  listen:
51          mov ah,0
52          int 16h
53
54  ret
55
56
57
58  msg:
59          db "hello world!"
60
61  times 512-($-$$) db 0    填充剩余扇区;0
```

下面我们将上面代码分解一下，详细介绍。

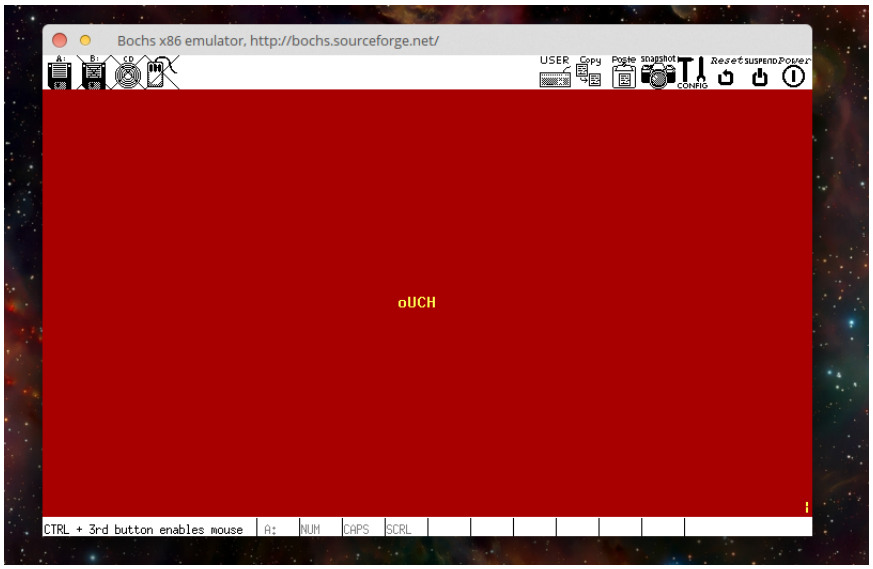
5.1 首先执行执行ah=0的系统调用将 OUCH 打印在屏幕中间

```
1   org 0x1000
2   ;org 0x100
3
4   ;#0
5   mov ah,0
6   int 80h
7   ;LISTEN_EXIT——
8   listen0:
9           mov ah,0
10          int 16h
```



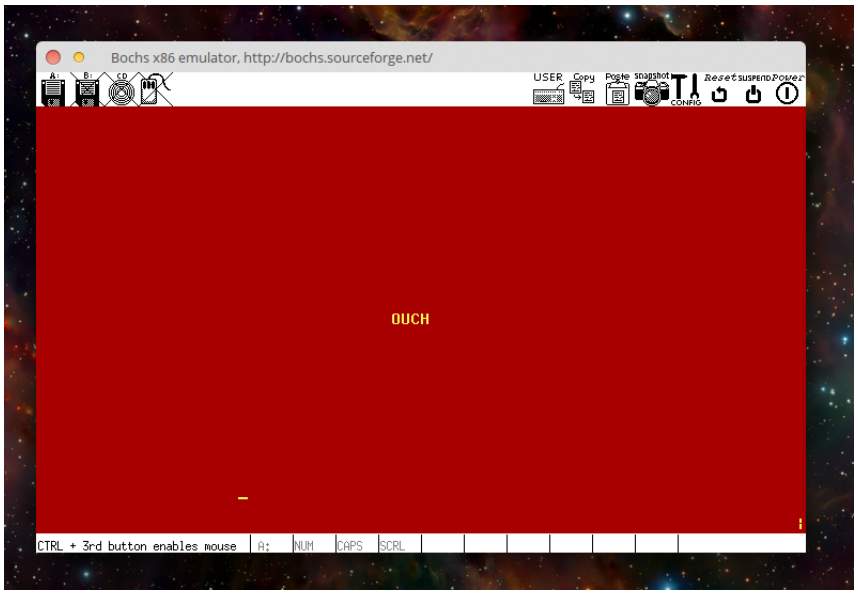
5.2 现在按下任意键，将执行ah=2的系统调用把 OUCH的第一个字母大写O变成小写o

```
1   ;#2
2   mov ax,0xb800
3   mov es,ax
4   mov dx,1994D
5   mov ah,2
6   int 80h
7
8   ;LISTEN_EXIT——
9           mov ah,0
10          int 16h
```



5.3 按下任意键，执行ah=1的系统调用把 oUCH的第一个字母小写o变回大写O

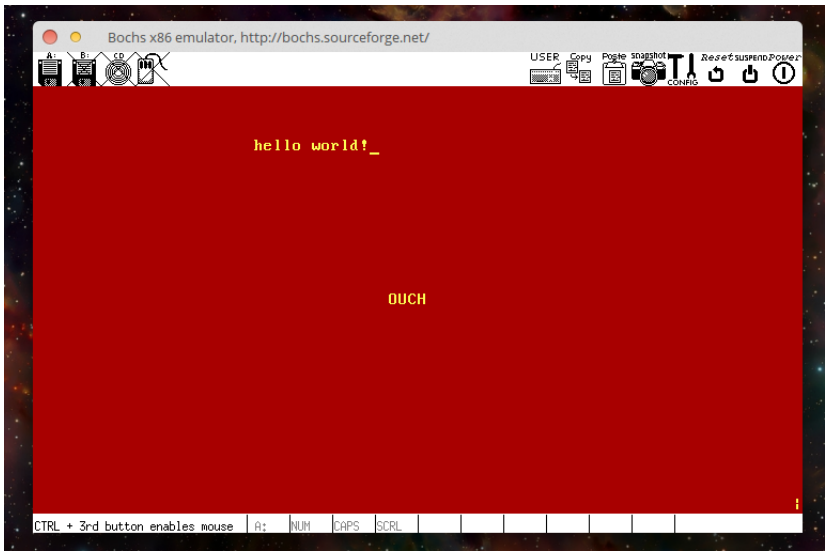
```
;#1
mov ax,0xb800
mov es,ax
mov dx,1994D
mov ah,1
int 80h

;LISTEN_EXIT——
        mov ah,0
        int 16h
```



5.4 按下任意键，执行ah=5的系统调用在屏幕3行17列的位置打印一个helloworld

```
;#5
mov ax,cs
mov ds,ax
mov es,ax

mov cx,0317h   ;position
```

```
mov ah,5
mov dx,msg
int 80h


;LISTEN_EXIT——
listen:
        mov ah,0
        int 16h

ret

msg:
        db "hello world!"
```

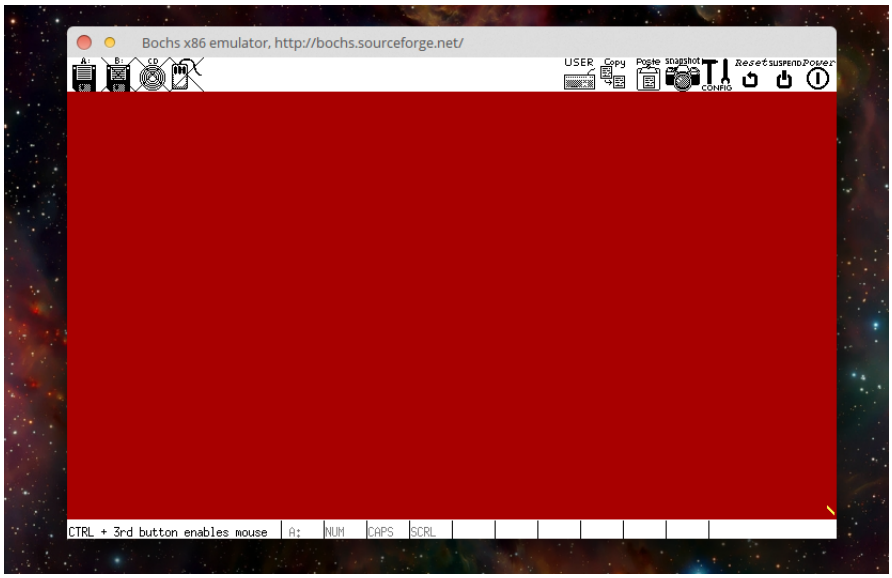


5.5 返回操作系统

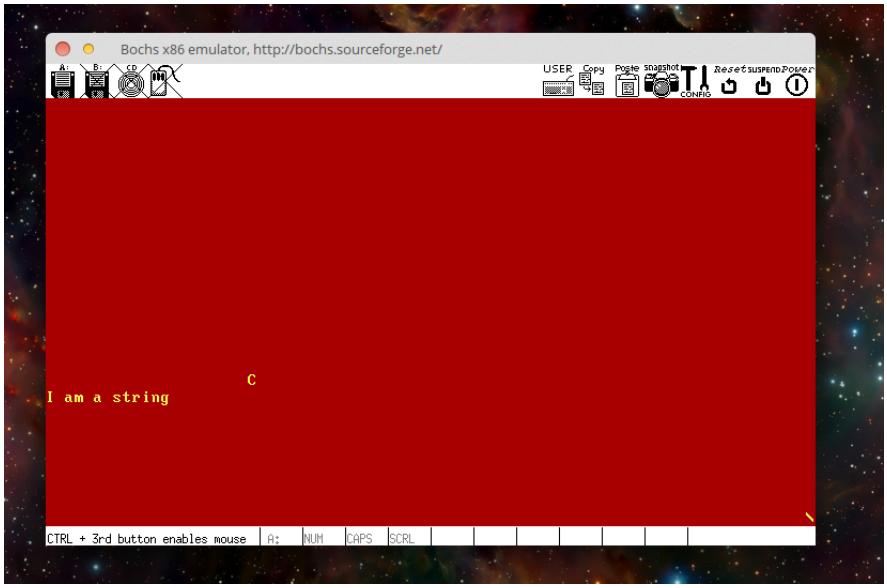
6. 接下来我们测试封装的osclib_share.c库里面实验所要求的`getch,gets,scanf,putch,putch,printint`,输入`run 1`执行第一个用户程序
代码如下:

```
void main(){
        screen_init();
        getch( &ch);
        gets( key);
        putch('\r');
        putch('\n');
        scanf("a=%d",&a);

        putch( ch);
        puts( key);
        printint( "ch=%c,a=%d,str=%s",ch,a,key);

        wait_key();
}
```

6.1 首先程序等待输入一个字符



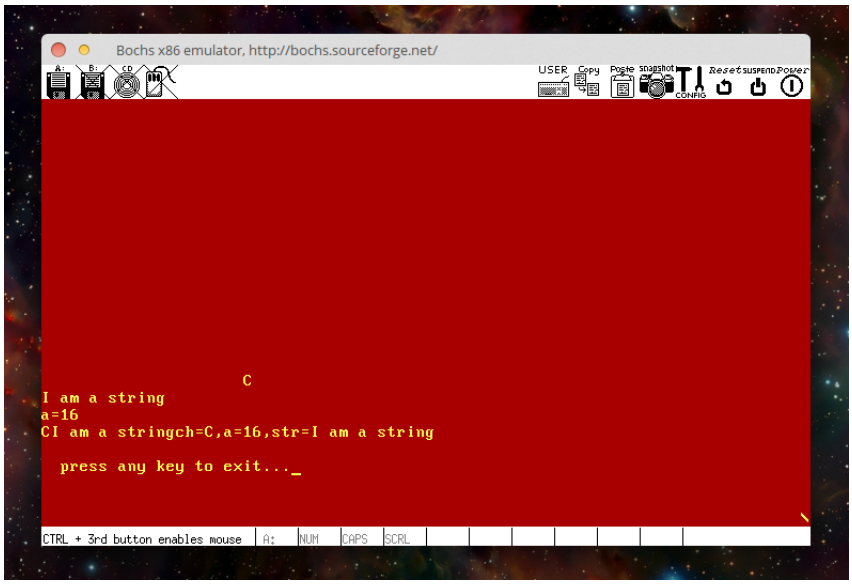
6.2 输入字符 C,回车再输入一个字符串`I am a string`回车



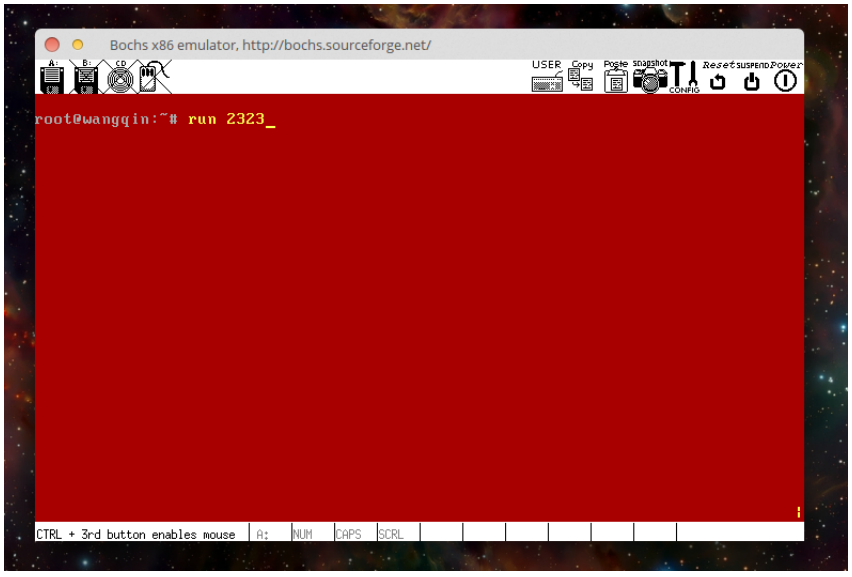
6.3 再输入一个int类型的数字16,回车后执行下面代码

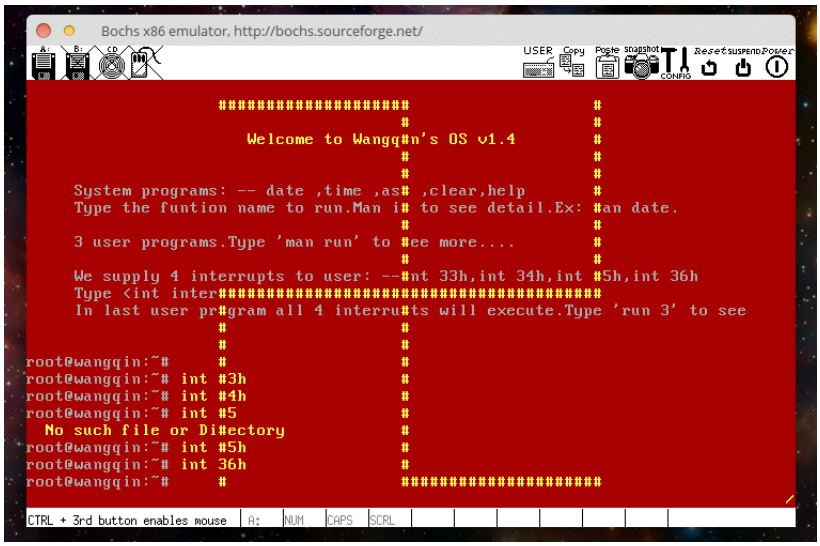
```
1           putch( ch);
2           puts( key);
3           printint( "ch=%c,a=%d,str=%s",ch,a,key);
```

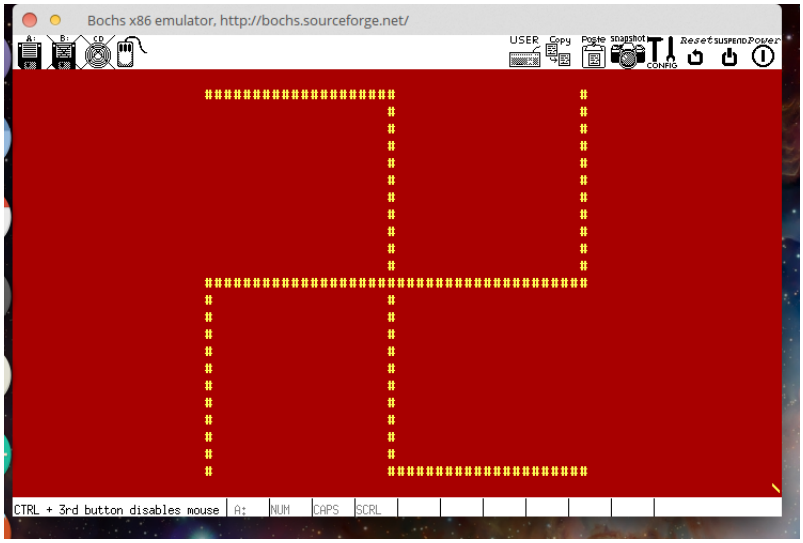
可以看到相继打印出了之前输入的内容



7. 和之前的实验一样，内核也有设置中断,系统内置程序和重复运行多个用户程序，这里就不再赘述了。







内存和软盘存储管理

1. 引导程序加载到内存0x7c00处运行
2. 引导程序将操作系统加载到0x7e00处运行
3. 操作系统讲用户程序加载到0x1000处运行
4. 软盘第1个柱面的第一个扇区存储操作系统引导程序
5. 软盘第1个柱面剩下所有扇区2~36扇区存储操作系统内核
6. 软盘第2,3,4柱面分别存储三个用户程序

更多细节信息请阅读我的Makefile文件

主要函数模块解释

内核架构解释:
os.c为内核主要控制模块
osclib.c os.asm主要为os.c提供函数实现.
oslib.asm 为osclib.c提供更底层的函数封装
osclib_share.c,oslib_share.asm 为用户程序中所需要用到的函数，从osclib.c oslib.asm中取出一部分作为内核和用户的共享库(使用户程序体积减少)。
os_syscall.asm 初始化系统调用和设置系统调用相关模块
更多细节信息请阅读我的Makefile文件

1. os.c: main 函数模块，这个在之前报告中已经解释这里就不在赘述

```
void main(){
        //--------------------init
        init_ss();
        screen_init();
        interrupt_init();
        syscall_init();
        print_welcome_msg();
        print_message();
        print_flag(); //root@wangqin4377@:    position
        //--------------------init_end

        while(1){
                char length = listen_key();
                if_screen_scroll(); //bottom of screen
                flag_scroll();//move flag to next line
                print_flag();
        }
}
```

2. 本次试验主要为了实现系统调用的工作，故在 os.asm中实现了下列函数供设置系统调用使用

```
        ;---PARAM: ah is syscall num   ebx is address of syscall  bx:temp cx:function ax:sysnum
        setting_up_syscall:
        mov bx,0
        mov es,bx
        mov al,ah
        mov ah,0
        shl al,2
        mov bx,0xfe00
        add bx,ax
        mov [es:bx],ecx
        ret
```

3. 初始化设置系统调用,设置全部6个系统调用功能，调用上面的函数实现

```
syscall_init:

;----#0 syscall
mov ah,0
mov ecx,0
mov cx,display_center_ouch
call setting_up_syscall

;----#1 syscall
mov ah,1
mov ecx,0
mov cx,letter_upper
call setting_up_syscall

;----#2 syscall
mov ah,2
mov ecx,0
mov cx,letter_lower
call setting_up_syscall

;----#3 syscall
mov ah,3
mov ecx,0
```

```
mov cx,atoi_syscall
call setting_up_syscall

;----#4 syscall
mov ah,4
mov ecx,0
mov cx,itoa_syscall
call setting_up_syscall

;----#5 syscall
mov ah,5
mov ecx,0
mov cx,display_str
call setting_up_syscall
ret
```

4. 对scanf,printint,gets..等函数的封装: 其实这些函数在之前的实验中已经实现，只是名字不一样罢了。本次实验单独抽出来相关代码放在osclib_share.c中。

5. python命令行工具：存放在python_extension.c中,主要功能的就是使用系统调用实现字符串和数值之间的转换。

```
        unsigned short int itoa_temp;
        char * itoa_ans;
        char * itoa( short int x){
                itoa_temp = x;
                __asm__("mov $4,%ah");   // syscall num
                __asm__("push %bp");
                __asm__("int $0x80");    // syscall
                __asm__("pop %bp");
                return itoa_ans;
        }

        char *atoi_temp;
        unsigned short int atoi_ans;
        unsigned short int atoi( char * str){
                atoi_temp = str;
                __asm__("mov $3,%ah");   // syscall num
                __asm__("int $0x80");    //syscall
                return atoi_ans;
        }
```

实验心得及仍需改进之处

实验心得:

通过本次试验我手动编写仿照linux的系统调用通过设置功能号ah然后使用int 80h中断来调用系统服务，让我了解了用户调用内核的系统服务的机制和原理 在实验的过程中遇到了很多问题，比如设置好系统服务程序后，使用int 80h 调用的时候却无法工作。通过使用bochs来调试解决了。

总结一下所有的bug分为两类：一类是因为自己缺乏相关知识或经验错误的使用了一些指令导致的，另一类就是自己粗心大意，在细节上没有处理好，结果调试很久才发现是一个小细节上疏忽了。对我 而言后一类是经常遇到的，以后的实验一定在编写代码的时候一定要非常谨慎，必须每一步都要考虑操作系统全局的实现，必须要先规划好层次和模块再去实现,不能茫然的 直接开始写代码。其中如何将实验

要求的实现系统服务功能展示出来我就换了很多种方式，最后确定功能好为3，4 的在python 命令行工具使用的过程中展示，其他的 在用户程序2中展示。

在做封装c的一些函数的时候，因为之前实验已经做好了那些函数，但是这些函数都和庞大的osclib.c os.asm oslib.asm密不可分，我就直接将这三个很大的 类库与用户1的程序联合编译，结果可想而知导致用户1的程序非常大，只能改变用户程序的存储方式，原来是一个扇区存储一个用户程序，改为一个柱面存储一个用户程序。后来 我将用户1程序中c函数所用到的一些代码从庞大的三个文件中分离出来，建立一个操作系统和用户的共享库osclib_share.c oslib_share.asm。使得用户1的体积减小了五倍, 只占两个扇区。希望在以后的实验中能实现在内核中直接可以解析执行elf文件，这样用户程序就可以已elf的格式存在，动态链接操作系统的类库，而不是自己也带一份一模一样的。

实验仍需改进之处：

仍需完善细节，比如说python的命令行工具加入乘法，除法完全是几行代码的问题
可以考虑将用户程序做成elf格式，动态链接系统的代码库。目前的情况是用户程序自己带着一份和操作系统一样的代码库，分别联合编译
考虑精简操作系统代码，减少冗余代码
继续优化调整内核架构和内存磁盘管理