中山大学数据科学与计算机学院
操作系统实验课程

实 验 报 告

教　　师　凌应标

学　　号　17341038

姓　　名　傅畅

实验名称　实验五 (实现系统调用)

实 验 五

实 现 系 统 调 用

姓名：傅畅 **学号：**17341038

邮箱：fuch8@mail2.sysu.edu.cn

实验时间：周五（3-4 节）

目录

一、 实验目的

1) 解释系统调用的意义及规划一组系统调用功能

2) 完成一个系统调用实现

3) 包括增加系统调用，高级语言库中即 C 程序库设计与系统调用的封装的方法和测试程序设计。

二、 配置方法及相关工具说明

（一）实验支撑环境

- 硬件：个人计算机
- 主机操作系统：Linux 4.18.0-16-generic 17-Ubuntu
- 虚拟机软件: Bochs 2.6.9

（二）实验开发工具

- 汇编语言工具：x86 汇编
- 程序编辑器：VScode
- x86 汇编器：NASM_v2.07
- 磁盘映像文件浏览编辑工具：wxHexEditor 0.23 Beta For Linux

本次实验主要在 VScode 编辑平台上使用汇编语言模式完成。汇编程序使用 NASM 的汇编命令生成 bin 文件，然后使用 wxHexEditor 检查格式完整，完成引导盘的设计。

三、 实验过程

这次实验我将 5 个系统调用（主要是一些 IO 函数），封装在软中断 int 0x11 中，通过寄存器传递指令参数的方式调用系统函数。这些函数大多都是 c 和 asm 交叉汇编而成。为了便于在 C 程序中进行读写操作，我将 in，out 的操作封装成一个可以供 C 调用的函数：

代码 1: In ,Out for C

```
Out:
                ;Out(port, val)
        push eax
        push edx
        mov dx, [esp+0xc]
        mov al, [esp+0x10]      ;从栈中取出端口和参数,
        out dx, al                      ;向端口送数
        pop edx
        pop eax
        ret
In:
                ;int In(port)
        push edx
        xor eax, eax
        mov dx , [esp +8]   ;从栈中区处于端口号
        in al, dx                       ;从段口取得返回值, 用eax返回值
        pop edx
        ret
```

（一）系统调用详情

3.1.1 0x0 , putchar

类似于 stdio.h 中的 putchar，该函数用于在屏幕上输出一个可视字符（包括回车和换行），然后光标移动一个单位。该调用兼顾了对光标越位时的滚屏操作。

3.1.2 0x01 , getchar

类似于 stdlib.h 中的 getchar()，该函数用于读取键盘缓冲区中的一个字符，如果缓冲区为空则阻塞忙等待。该键盘缓冲区由键盘中断例程进行维护。

3.1.3 0x02 , simple_puts

用于在一个特定的位置打印一个字符串，这过程不对光标进行移动。

3.1.4 0x03 , BCD clock()

用于返回当前时间（24 小时制）的 BCD 码，即 3 字节 24=4*6 位；该格式只是为了方便打印在屏幕上。

3.1.5 0x04 , clear screen

该中断用于清屏，清除屏幕上的所有字符，并将光标放置在 (0,0) 处。

（二）系统调用代码分析

3.2.1 0x0 , putchar

putchar 中的 C 语言实现部分

```
1  void putchar(u8 c){
2          Out(0x3d4, 0x0e);
3          int pos=In(0x3d5)<<8;            //两次读取光标位置的低字和高字，
4          Out(0x3d4, 0x0f);
5          pos|= In(0x3d5);
6          if ( c==0x0d) pos=(pos/80+1)*80;else     //判断换行和回车
7          if (c==0x0a)pos+=80;else{
8                  buf[0] = c; buf[1]=0;
9                  simple_puts(buf, (pos<<16)+0x7);        //其他可视字符，直接在
                       该位置打印
10                 ++pos;
11         }
```

```
        while ( pos>=25*80)roll_screen(), pos-=80;      //处理光标越位而滚屏的操作
        Out(0x3d4, 0x0e);
        Out(0x3d5, pos>>8);                                 //将新的坐标位置写回端口
        Out(0x3d4, 0x0f);
        Out(0x3d5, pos&255);
        return ;
}
```

puchar 的封装过程如下

```
        cmp al, 0                                       ;0  putchar , cl=char
        jne     .endhandle0
        push ecx
        call putchar                                ;C风格调用函数,
        add esp, 4
        jmp .endsyscall
.endhandle0:
```

其中 roll_screen 的代码如下

```
        roll_screen:
        pushad

        cld
        mov esi, VideoSite+0xa0          ; 定向后, 将后24*80的字符向前挪动80个单位
        mov edi, VideoSite
        mov ecx, 1920
        rep movsd
        mov bx, 3840
```

```
        mov ecx, 80
.cls:
        mov word [VideoSite+ebx], 0x0720        ;将最后一行的80个位置清除
        add bx, 2
        loop .cls

        popad
        ret
```

3.2.2 0x01 , getchar

getchar 中用于维护键盘缓冲区的部分 C 代码如下，

```
u8 keybuf=0, boolkeybuf=0;
void flush_to_keyb(u8 keyval){
        if ( keyval>=0x080) return ;
        u8 ch=0;
        for (int i=0; i<KEYNUMS; ++i)
                if ( keyval == keymp[3*i+1])ch=keymp[3*i];
        keybuf=ch; boolkeybuf=1;
}
u8 getchar(){
        while (!boolkeybuf);
        boolkeybuf=0;
        return keybuf;
}
```

键盘中断响应的处理过程如下

```

keyboard_interrupt_handle:
        pushad
```

```
        mov al, 0x20
        out 0xa0, al
        out 0x20, al

        xor eax, eax
        in al, 0x60
        push eax
        call flush_to_keyb
        pop eax
```

getchar 的封装如下，为了能够在忙等待的过程中能够及时使用键盘中断响应例程，以更新键盘缓冲区

```
        cmp al, 1                                       ;1      getchar ,
            return al=getchar()
        jne     .endhandle1
        xor eax, eax
        sti                                                     ; ??? must be
            open to let keyboard handle to flush the keybuf
        call getchar
        cli
        jmp .endsyscall
```

3.2.3 0x02 , simple_puts

本来这是个给内核使用的简陋代码，不过考虑到以后还是可能会遇到需要在不移动光标的情况下打印字符串的情况，还是将这个打印代码封装了。

```
simple_puts:
pushad              ; 简单的输出字符串，不涉及光标移动
                                                ; arg1 is string pointer
                                                ; arg2 的低16位表示颜色，高16为
```

```
                                                表示显示的启示位置，即x*80+
                                                y (col,xy)
5                       ; simple_puts(string pointer , color_and_site)
6                                               ; C function call is near call,
                                                    only push cs
7               mov ebx , [esp+0x28] ;from 44
8               xor eax , eax
9               mov ax  , bx
10              shr ebx , 15          ; shr 16  ,, shl 1
11
12              mov ebp , [esp+0x24]  ; from 40
13              .enumchar:
14                              mov cl,[ebp]
15                              cmp cl, 0x0   ; 字符串默认以0结尾，
16                              je .endenum
17                              mov [VideoSite+ebx], cl
18                              inc ebx
19                              mov [VideoSite+ebx], al
20                              inc ebx
21                              inc ebp
22                              jmp .enumchar
23              .endenum:
24              popad
25  ret
```

封装过程

```
1       cmp al, 2                                       ;2      simple_puts ,
            ebx=*str  , ecx=pos<<16+col
2       jne     .endhandle2
3       push ecx
```

```
        push ebx
        call simple_puts
        add esp , 8
        jmp .endsyscall
.endhandle2
```

3.2.4 0x03 , BCD clock()

该例程用于获取 BCD 码格式的当前时间，时间中断处理例程如下，

```
rtm_0x70_interrupt_handle:
        push eax
        mov al,0x20                         ;中断结束命令EOI
        out 0xa0,al                         ;向8259A从片发送
        out 0x20,al                         ;向8259A主片发送

        mov al,0x0c                         ;寄存器C的索引。且开放NMI
        out 0x70,al
        in al,0x71                          ;读一下RTC的寄存器C，否则只发生一次
            中断;此处不考虑闹钟和周期性中断的情况
        pop eax
        call c_rtm_0x70_interrupt_handle    ;调用
        iretd
```

封装

```
        cmp al, 3                                       ;3  clock() return BCD
            code 0x00HourMinSec
        jne     .endhandle3
        call curr_clock
        jmp .endsyscall
.endhandle3:
```

时间处理函数的主体部分在 C 中，包括继续现实风火轮和当前时间，软中断所封装的是 current_clock()

```
1  //clock interruption
2  const char message_cyc[4]="\|/-";
3  int curcyc=0;
4  void c_rtm_0x70_interrupt_handle(){
5          // Out(0xa0, 0x20); Out(0x20, 0x20);
6          // Out(0x70, 0x0c); In(0x71);
7
8          buf[0] = message_cyc[curcyc++];buf[1]=0;        //每次循环打印一个字符
9          curcyc&=3;
10         simple_puts(buf, (24*80+79<<16) + 4);
11         show_current_clock();
12 }
13 u32 curr_clock(){       // BCD code ,total 3 byte 12:34:56 <---> 0x00123456
14         u32 res=0;
15         Out(0x70, 0x84);        res=In(0x71);
16         Out(0x70, 0x82);        res=(res<<8)|In(0x71);
17         Out(0x70, 0x80);        res=(res<<8)|In(0x71);          //读取三个端
               口，所读取的已是bcd编码的数字
18         return res;
19 }
20 void show_current_clock(){
21         u32 clk= curr_clock(), len=0;
22         for (int i=0; i<6; ++i, clk>>=4){                       //不断右移4位，
               取出低4位的BCD数
23                 buf[len++] = (clk&0x0f)+'0';
24                 if ( i<5 && i%2==1) buf[len++]=':';
25         }
```

```
        reverse(buf, buf+len); buf[len]=0;
        simple_puts(buf, (24*80+70<<16)|8);
}
```

3.2.5 0x04 , clear screen

清屏函数，用于用户程序执行前

```
clear_screen:
                pushad
                mov ecx, 2000
                mov ebx, 0
        .clall:
                mov word [VideoSite+ebx], 0x0720
                add bx, 2
                loop .clall

                xor eax, eax
                mov al, 0x0e
                mov dx, 0x3d4
                out dx, al
                mov al, 0
                inc dx
                out dx, al
                mov al ,0x0f
                dec dx
                out dx, al
                mov al, 0
                inc dx
                out dx, al

```

```
24                  popad
25                  ret
```

封装过程

```
1         cmp al, 4                                         ;4      clear screen()
2         jne     .endsyscall
3
4         call clear_screen
5
6   .endsyscall:
```

（三）内核代码测试系统调用

在开启中断之后，进入用户程序之前对中断进行测试

```
1         call cmain
2         mov al, 0x01              ;getchar,
3         int 0x11
4         mov cl, al
5         mov al, 0x0                         ;并打印出来
6         int 0x11
7
8         mov al, 0x4               ;清屏
9         int 0x11
10        mov ebx, selfMessage ;并显示一条个人信息
11        mov ecx, 0xf00003
12        mov al, 2
13        int 0x11
```

其中在 main 函数中的个人调用

```
1   void cmain(){
2         simple_puts("Hello  world!",0x7);
```

```
3         simple_puts("I'm from C language",(1*80+0<<16)+6);
4         return ;
5 }
```

（四）用户代码测试系统调用

为了方便 C 语言调用系统中断，在用户程序 asm 中对其再次封装

```
1  global putchar
2  global getchar
3  global get_curr_time
4  global _start
5  extern user_main
6  _start:
7                  mov al, 4                        ; clear _screen
8                  int 0x11
9                  call user_main
10                 jmp $
11 putchar:
12                 push eax
13                 push ecx
14                 mov al, 0
15                 mov cl, [esp+0xc]
16                 int 0x11
17                 pop ecx
18                 pop eax
19                 ret
20 getchar:
21                 xor eax, eax
22                 mov al, 1
23                 int 0x11
```

```
                ret
get_curr_time:
                xor eax, eax
                mov al, 3
                int 0x11
                ret
```

然后用户 C 代码部分利用的再次封装好的中断，测试读入字符清屏并显示进入时间。

```
#define N 20
const char input_info[]="Please input a char:", enter_info[]="Enter user
    program clock:";

char time_str[N];
extern void user_main(){
        for (int i=0; input_info[i]; ++i) putchar(input_info[i]);
        char ch= getchar();
        putchar(ch);
        putchar('\r');
        int clk = get_curr_time(),len=0;
        for (int i=0; enter_info[i]; ++i) putchar(enter_info[i]);
        for (int i=0; i<6; ++i , clk>>=4){
                time_str[len++] =(clk&0x0f)+'0';
                if ( i%2==1 && i<5) time_str[len++]=':';
        }
        for (int i=len ;i; --i) putchar(time_str[i-1]);
}
```

四、 实验结果及分析

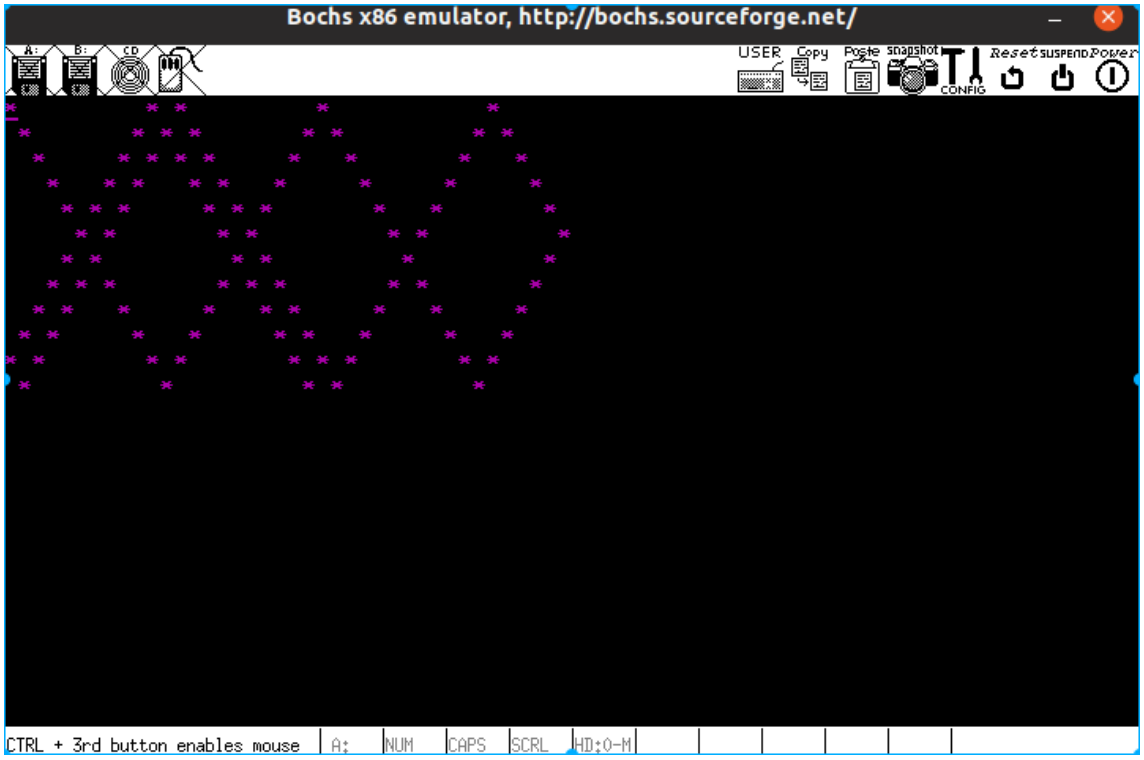

图 1: 显示一个弹球弹幕

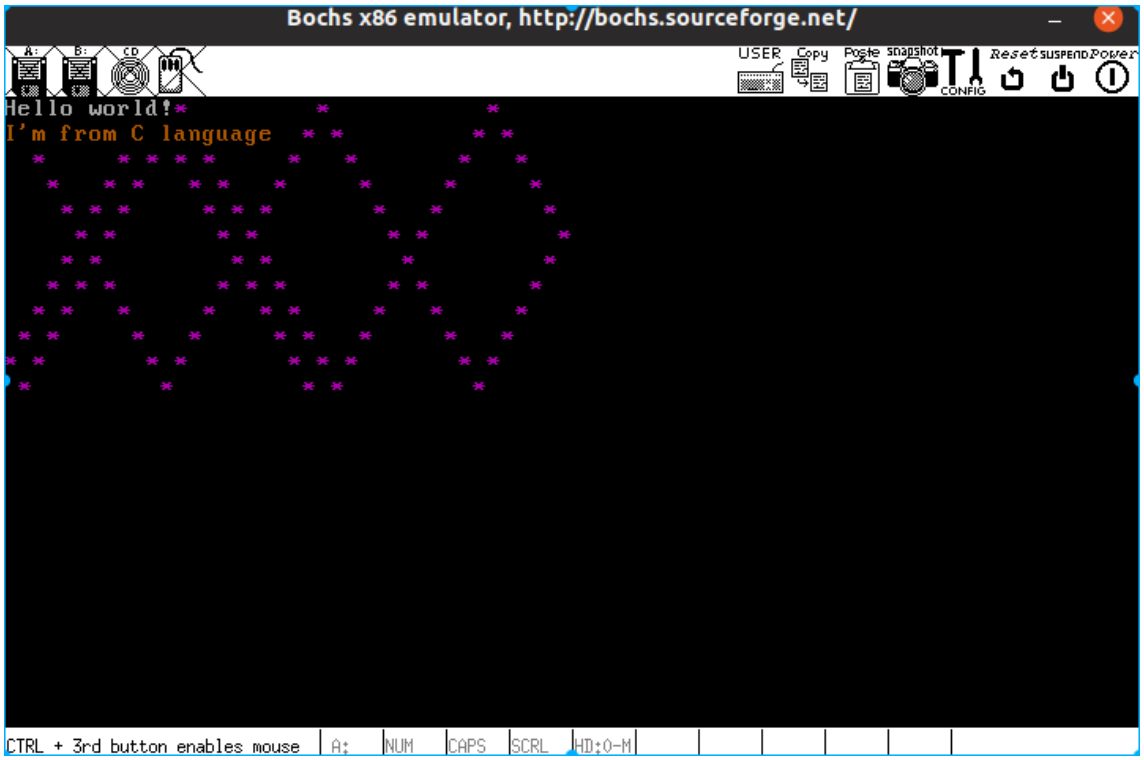

图 2: 显示两个字符串

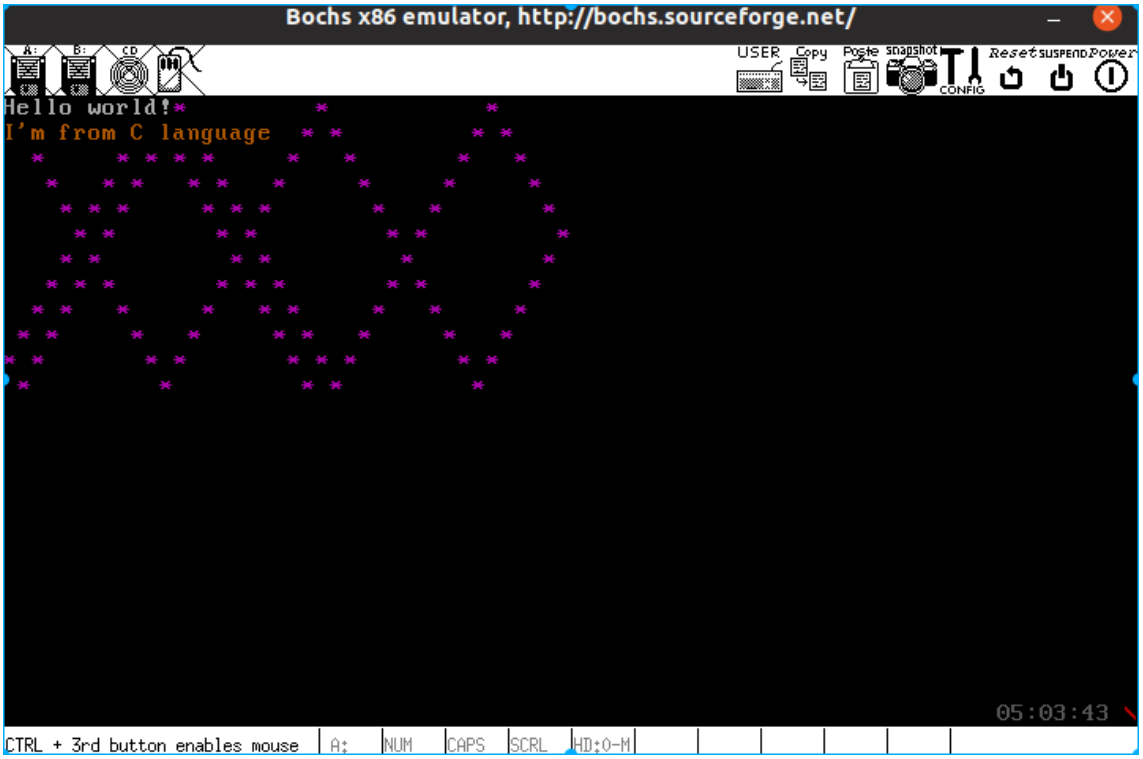

图 3: 阻塞等待读入字符，此时右下角的时间开始显示

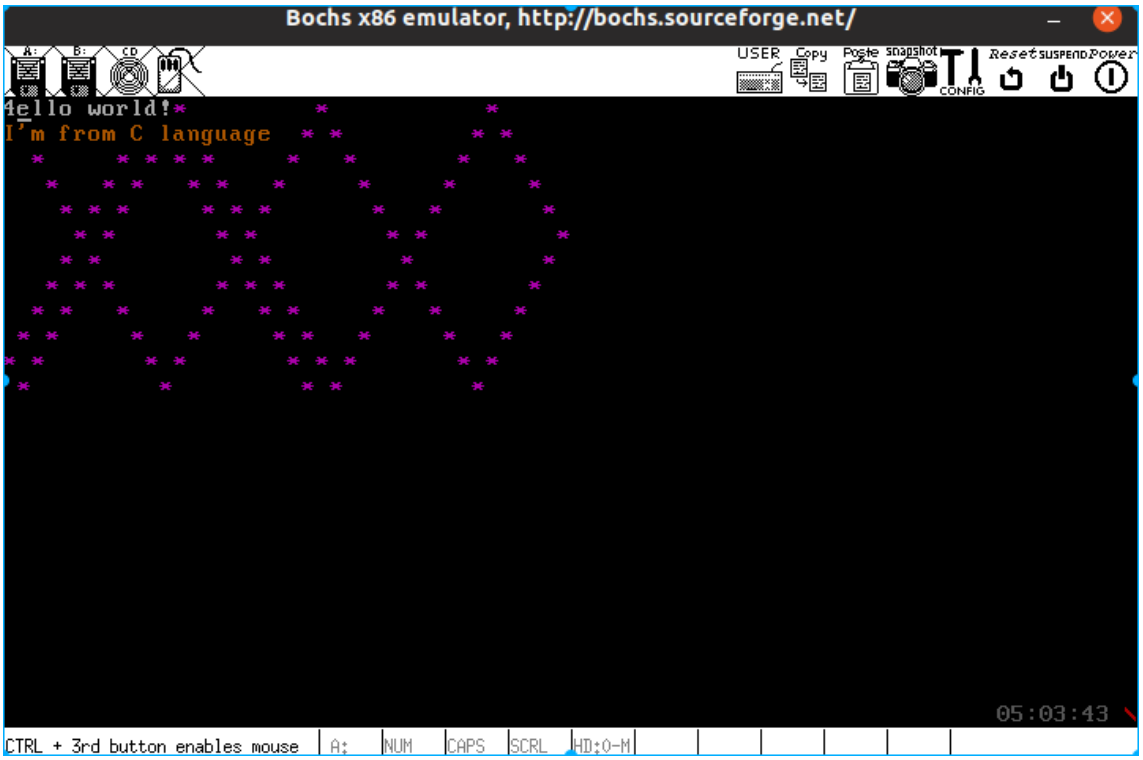

图 4: 成功读入字符 4，并显示在第一个位置

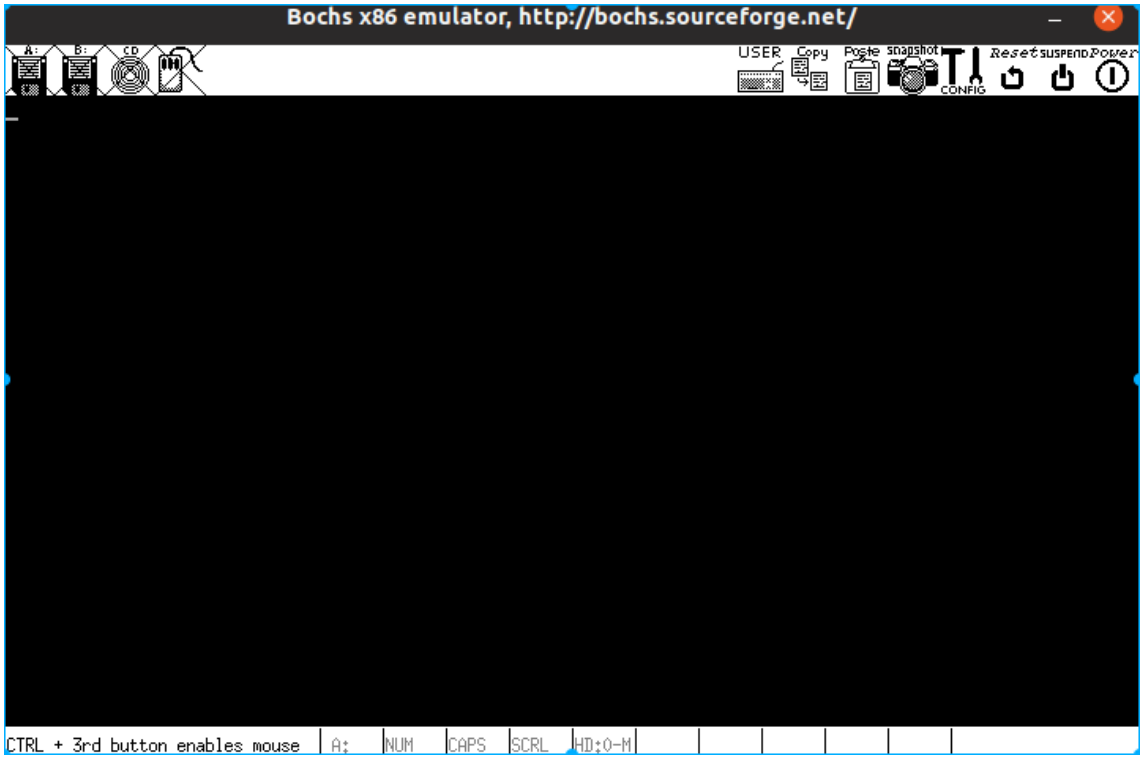

图 5: 第一次调用清屏

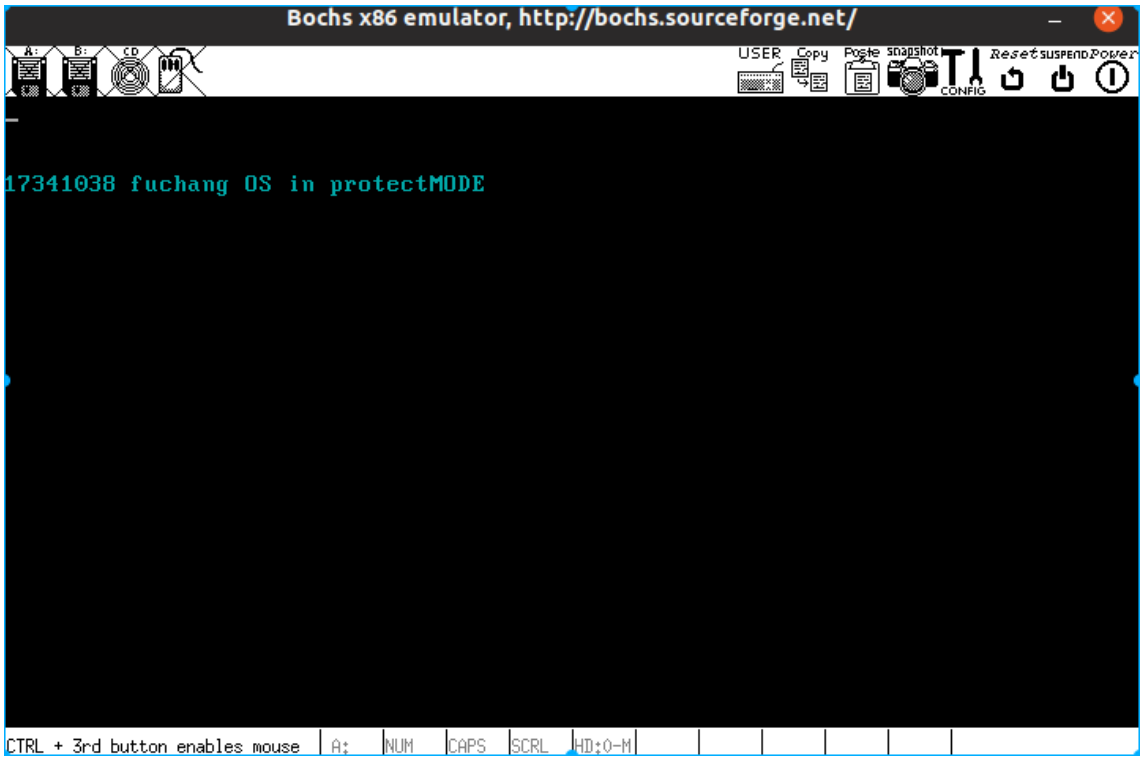

图 6: 显示个人信息

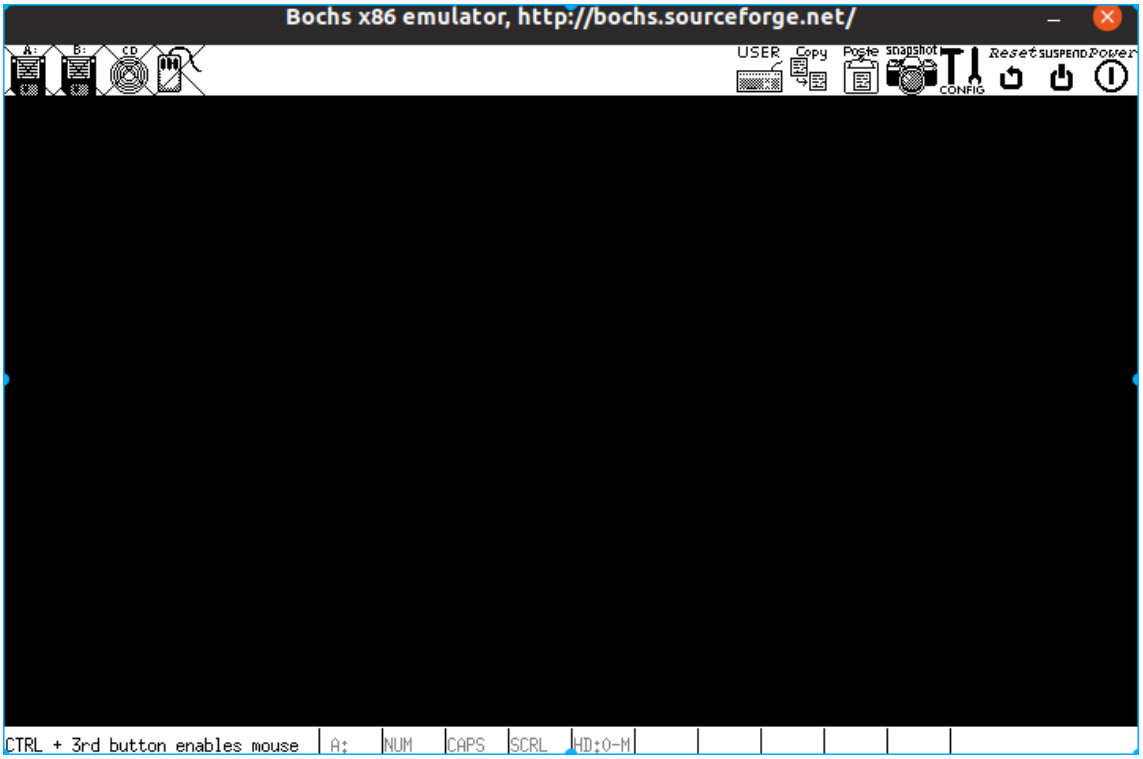

图 7: 第二次清屏

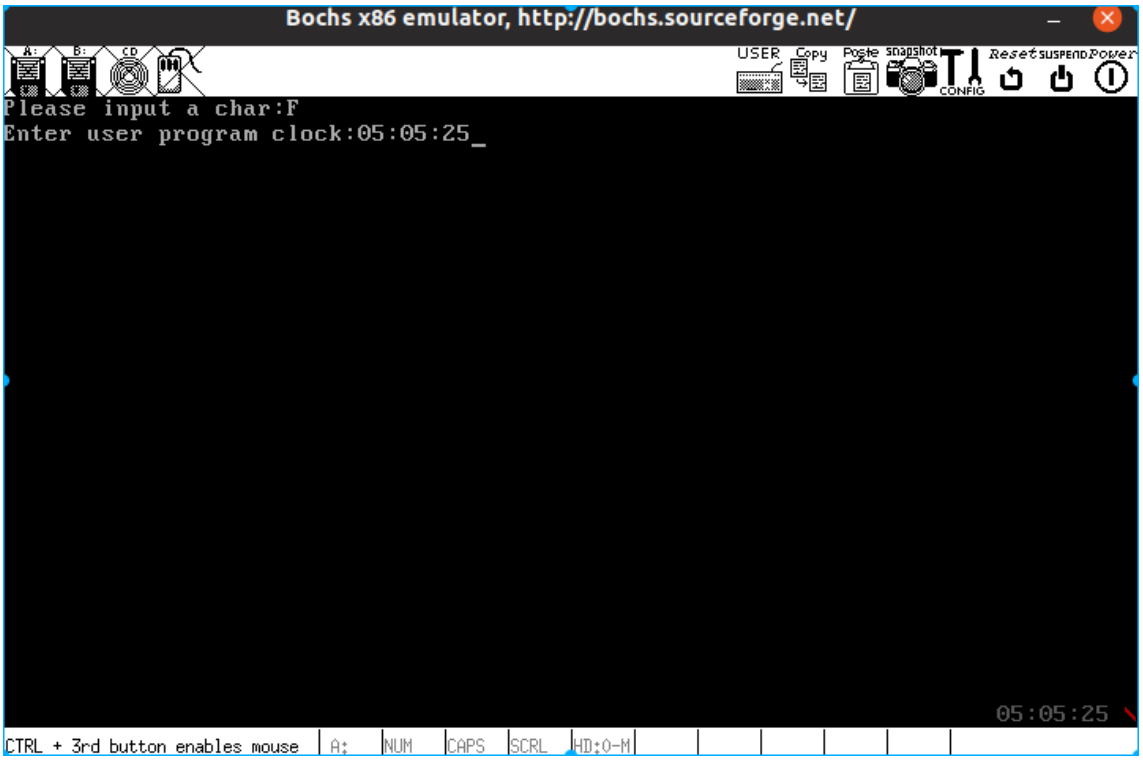

图 8: 读入一个字符 F 并显示，接着便显示进入用户程序的时间，和右下角一致

五、 实验总结

本次系统调用的实验是为以后实现多用户甚至多进程的模型做准备。日益增长的用户功能急需封装更多的系统调用。本次实验我掌握的通过软中断封装系统调用的方式实现了一些非常基本的 IO 函数

尽管 getchar 还非常简陋，因为我还没能够支持 shift 输入转义字符。不过相比之前只能输出的程序，还是迈出了一大步

因为系统调用是一个个添加的，并且做足了系统测试与模块测试，一些常见的不常见的 bug 总是很快地被调出来了。而且和以前不一样，由于贯彻了“先阅读手册和书籍”再写代码的教训，遇到问题总是能很快地摸清解决的方向并很快地找到解决方案，这相比起以前实验给我更有趣的体验。

实验的不足还是有的，其中一个就是不能动态的确定被加载代码块的长度，这在以后代码模块动态变化很大的时候会造成隐患，这个问题应该在下一个实验得到解决。