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　实验三（开发独立内核的操作系统）

一、 实验目的: 用 C 和汇编实现操作系统内核

（一）用 **C** 和汇编实现操作系统内核

（二）增加批处理能力

1.2.1 提供用户返回内核的一种解决方案

1.2.2 一条在内核的 C 模块中实现

- 在磁盘上建立一个表，记录用户程序的存储安排
- 可以在控制台查到用户程序的信息，如程序名、字节数、在磁盘映像文件中的位置等
- 设计一种命令，命令中可加载多个用户程序，依次执行，并能在控制台发出命令
- 在引导系统前，将一组命令存放在磁盘映像中，系统可以解释执行

二、 实验原理:

（一）**c** 函数与 **nasm** 函数的相互调用

c 语言在实现比较复杂的程序逻辑上比较简单，而 asm 在实现底层 IO 功能上比较直接，本次实验的许多个功能我需要两者尽可能多地需要两者相互调用，汲取各自长处 c 语言函数的调用，在汇编中的实现方法，只是将参数简单压入栈，然后将返回出口压入栈后，直接跳向函数入口。知道这一点之后，汇编代码想要直接调用 c 函数，只需要将参数压入栈即可；asm 代码段想要使用 c 函数传过来的参数，也只需要从 $sp+4$ 之后的位置取出即可。在函数返回值在汇编层面也只是用 eax 来传递

（二）**c** 语言与 **nasm** 交叉编译

由于现阶段暂时还只能编译 16bin 文件，gcc 的参数需要指明-m32，同时需要关闭对关联库和 builtin 函数。

（三）obj 文件的链接

ld 对.o 文件先后顺序敏感，被依赖的文件需要放在后面；而且编译出来在 bin 文件中相对顺序也和命令中文件顺序有关

（四）执行用户程序前后的寄存器保护措施

所有在用户程序中被使用的寄存器，都需要使用栈加以保存。待到模块结束 ret 后需要弹出恢复。

三、 代码分析:

（一）shell 代码逻辑

整个 shell 的逻辑比较简单，不断输入字符并在屏幕上追踪显示；如果输入了删除符 0x8 则退格显示。

等到输入回车换行符之后，光标换行 Enterline()，并扫描命令行，跳进命令解释函数 exec_single()

```
void shell(){
    for (;;){
      char p[15]= "fuchang@1038 $\0";
      int len =strnlen(p, MAXLEN),i;
      puts(p, len, &row,&col);
      len=0;
      for (char ch=getchar(); ch!= '\r' && ch!='\n'; ch=getchar()){
        if ( ch==0x8 ){
          if ( len){
            --col;
            putchar(' ', &row,&col), s[--len] = 0;
            --col;setCursor(row, col);
          }
        }else{
          putchar(ch, &row,&col);s[len++] = ch;
```

```
16          }
17        }
18        Enterline(&row, &col);
19        for (i=0; s[i]==' ' && i<len;++i);
20        if (i<len)exec_single(s+i, len-i);
21      }
22    }
```

代码 1: shell 函数

（二）ls 现实当前文件信息

该 ls 命令会将表中的所有文件的文件名都输出

另外我设计的 ls 命令带有一个可选参数 $-al$，该参数下 ls 会将文件的所有详情信息全部输出。

```
1  void dir(int showall){
2          char *p;
3          for (volatile int i=0; i<32; ++i){
4                  p=readItem(i);
5                  if (p[0]==0) break;
6                  volatile int len = p[7]=='\0'?strnlen(p,8):8,num=0;
7                  if ( !showall){
8                          if ( i) {putchar(' ', &row,&col);}
9                          puts(p, len, &row, &col);
10                 }else{
11                         puts(p,len, &row, &col);
12                         puts(" section:",9, &row, &col);putnum(sectionLoc(p), &
                               row,&col);
13                         puts(" size:",6,&row, &col); putnum(fileSiz(p)<<9, &row
                               , &col);
```

```
                        num=createTim(p);
                        puts(" crtime:",8, &row, &col);
                        putnum(num/60,&row,&col);putchar(':',&row,&col);putnum(
                            num%60, &row,&col);
                        puts(" filetype:",10,&row, &col);
                        num=fileTyp(p);
                        if ( num==1) puts("bin",3, &row, &col); else
                        if ( num==2) puts("bat",3, &row, &col); else
                        if ( num==0) puts("ker",3, &row, &col);

                        Enterline(&row, &col);
                }
        }
        if ( !showall)Enterline(&row,&col);
}
```

代码 2: dir() 函数

为了方便读取文件信息，我对每一条文件信息都写了对应的函数

```

typedef char* itemPtr;
int sectionLoc(itemPtr p){return (((int)p[9])<<8) + p[8];}
int fileSiz(itemPtr p){ return (((int)p[11])<<8)+p[10];}
int createTim(itemPtr p){return ((  (int)p[13])<<8)+p[12];}
int fileTyp(itemPtr p){return (((int)p[15])<<8)+p[14];}
```

代码 3: 文件条目读取

（三）对批处理文件与 **bin** 文件的执行

该函数的解释过程其实就是对两个指令字符串的匹配过程。整个过程比较冗余的还是对非法输入的判断。

对用户 bin 文件的执行和用户 batch 文件的执行还是有不同的，用户 batch 文件本质上还是一堆指令字符串

```
void exec_single(char *cmd, int len){
  if ( strncmp(cmd, "./",2)==0){
    int i=2,j,k;
    for (; i<len && cmd[i]==' ';++i);
    for (j=i; cmd[j]!=' ' && j<len; ++j);
    int exist=0;
    itemPtr t=readItem(k);
    for (k=0; k<32; ++k,t+=16) if ( strncmp(cmd+i,t, j-i)==0){ // bug ,t+=16?
      if ( fileTyp(t) ==1)run_user_prog(k);else
      if ( fileTyp(t) ==2)exec_batch(get_user_bat(k),fileSiz(t)<<9);
      exist = 1;
    }
    if ( !exist) puts("file not found!",15, &row,&col);

  }     else
  if ( strncmp(cmd, "ls",2)==0){
    int i=2, j;
    for (; i<len && cmd[i]==' '; ++i);
    for (j=i; cmd[j]!=' '&& j<len; ++j);
    if ( j-i>0){
      if (j-i==3  && strncmp(cmd+i, "-al",3)==0)
        dir(1);
      else puts("paramenter not found!",21, &row, &col);
    }else dir(0);
  }     else
    puts("command not found!",18, &row, &col);
```

```
}
```

代码 4: 解释并执行用户程序

用户所执行的批处理命令的执行程序，直接解析带有 $\backslash r$ 作为分隔符的字符串。

```
void exec_batch(char *batchs, int len){
  for (volatile int i=0,j; batchs[i] && i<len; i=j){
    for (; batchs[i]==' ' && i<len; ++i);
    for (j=i; batchs[j]!='\r'&&batchs[j]!='\n' && batchs[j] && j<len; ++j);
    if (i==j){ ++j;continue;}
    exec_single(batchs+i, j-i);
  }
}
```

代码 5: 执行用户批处理文件

```
    bat.sh_start:
    db "./prog2",0xa,"./prog1",0xa
    times 512-($-bat.sh_start) db 0
```

代码 6: 批处理程序内容

调用 nasm 中的用户程序，需要加载其到对应的内存并跳转执行，这里的 run_user_prog 和 get_user_bat 用 nasm 写的都是得到目的地地址或者命令地址然后直接执行

```
run_user_prog:          ;debug
        ; run user's bin prog
    push ax
    push bx

    mov bx, sp
    push word [bx+0x8]
    call dword Load_prog
    xor ebx, ebx
    mov bx, ax
    call dword ebx

    add sp, 0x2         ;bug , fail to match the push--pop bracket
    pop bx
    pop ax

o32 ret
```

```
18
19 get_user_bat:
20                          ;run user's bat order
21
22         push bx          ;each take 2 units of stack memo
23
24         mov bx, sp
25         push word [bx+0x6]
26         call dword Load_prog
27         add sp,0x2
28
29         pop bx
30
31 o32 ret
```

代码 7: 获取首地址之后，直接跳转执行

（四）**IO** 函数设计

我主要在 nasm 中封装了 4 个函数:

3.4.1 putchar

调用 10 号中断，将字符串输出到第 0 页的指定位置

```
1  asm_putchar:             ;debugged
2  push ax
3  push bx
4  push cx
5  push dx
6  push ds
7  push es
8  push bp   ; bug, every register used in the prog must be protected !!!
9
10   mov  ax, cs            ; 置其他段寄存器值与CS相同
11   mov  ds, ax            ; 数据段
12   mov  bp, sp       ; BP=当前串的偏移地址
13   add bp, 0x12
14   mov  ax, ds            ; ES:BP = 串地址
15   mov  es, ax            ; 置ES=DS
16   mov  cx,1  ; CX=串长
17   mov  ax, 1301h         ; AH = 13h（功能号）、AL = 01h（光标置于串尾）
18   mov  bx, 0007h         ; 页号为0(BH = 0) 黑底白字(BL = 07h)
19   mov  dh, [bp+0x4]              ; 行号=0
20   mov  dl, [bp+0x8]              ; bug , each is 64bit !!!
```

```
21    int   10h                    ; BIOS的10h功能：显示一行字符
22
23  pop bp
24  pop es
25  pop ds
26  pop dx
27  pop cx
28  pop bx
29  pop ax
30  o32 ret
```

代码 8: putchar

3.4.2 getchar

getchar 实现比较简单，调用 16h 中断后，所得 ascii 码直接存在了 al 中，作为返回值可以直接 ret

```
1  getchar:              ;debugged
2    mov ax,0x0
3    int 16h
4  o32 ret
```

代码 9: 几行 getchar

3.4.3 屏幕下滚

```
1  ScrollDown:           ;debugged
2
3      push ax
4      push bx
5      push cx
6      push dx
7
8      mov     ah,6      ;6=屏幕初始化或上卷 ， 7=屏幕初始化或下卷
9      mov     al,1;     AL = 上卷行数AL =0全屏幕为空白
10     mov     bh,0;     BH = 卷入行属性
11     mov     cx,0; CH = 左上角行号 CL = 左上角列号
12     mov     dx,0x184f     ;DH = 右下角行号 DL = 右下角列号(24,79)
13     int 10h
14
15     pop dx
16     pop cx
17     pop bx
```

```
18      pop ax
19    o32 ret
```

代码 10: 10h6h 功能，懒人上卷

3.4.4 设置光标

```
1   setCursor: ; debugged
2     push dx
3     push bx
4     push bp
5     push ax
6
7     xor ax, ax
8     xor bx, bx
9     mov ah, 2h
10    mov bp,sp
11    mov dh, [bp+0xc]  ; mad to calc the real site in stack
12    mov dl, [bp+0x10]
13    int 10h
14
15    pop ax
16    pop bp
17    pop bx
18    pop dx
19  o32 ret
```

代码 11: 根据用栈传进来的参数设置光标

然后在 c 语言中就可以比较方便地实现 puts,putnum,Enterline 等函数：

```
1  void putchar(char ch, int *r, int *c){  // used for
2    asm_putchar(ch, *r, *c);
3    ++(*c);
4  }
5
6  void puts(char *s, int len, int *r, int *c){
7    for (volatile int i=0; i<len && s[i]; ++i,++(*c))
8      asm_putchar(s[i], *r,*c);
9  } // print a string at the (r,c), used for terminal
```

```
10
11 extern void Enterline(int *r, int *c){
12   for (;(*r)>=24; --(*r))ScrollDown();
13   ++(*r);*c=0;
14 }
15 void putnum(int num,int *r, int *c){
16   tmplen=0;
17   if ( num==0) putchar('0', r, c);      else{
18     for (; num ;num/=10) tmp[tmplen++] = num%10;
19     for (volatile int i=tmplen-1; ~i; --i)
20       putchar(tmp[i]+'0',r, c);
21   }
22 }
```

代码 12: stdio.h

（五）**makefile**

makefile 在系统调试的时候给我的工作带来了极大的便捷

里面的内容大多数是参考老师 PPT 上的内容，只不过自己编译的时候会出现地址重定义的问题，gcc 中补上-fno-XXX 等参数就可以了

```
1 all:
2   gcc -m32 -Og -c ker.c -o ker.o -fno-PIC -fno-builtin -nostdinc -fno-stack-
      protector
3   nasm -f elf -g -F stabs  loader.asm -o loa.o
4   ld -m elf_i386 -N loa.o ker.o -s -Ttext 0x7c00 --oformat binary -o fd0 --
      entry=_start
5
6 clean:
7   rm *.o
```

代码 13: makefile

四、 实验结果与分析

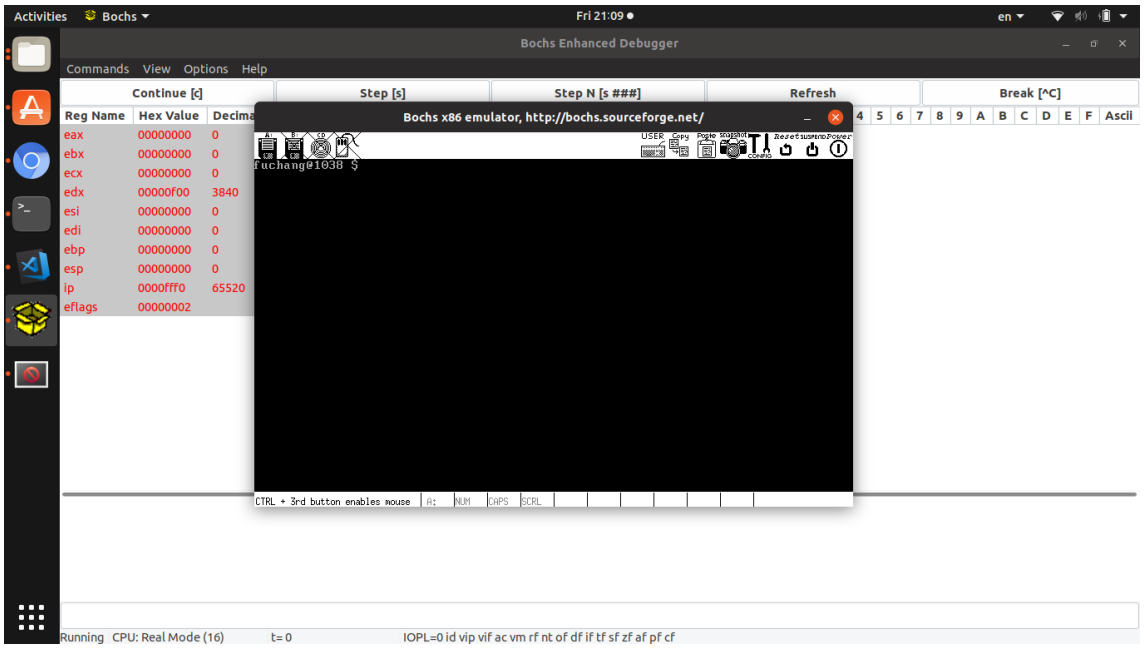

图 1: 打开程序显示终端

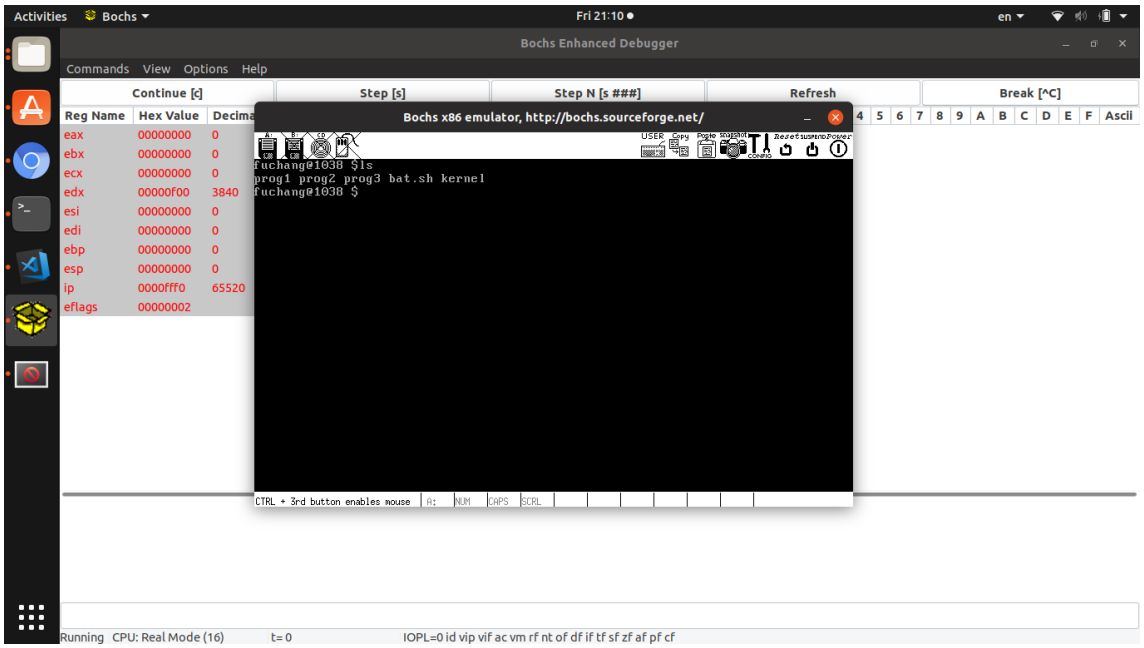

图 2: 输入 **ls** 指令

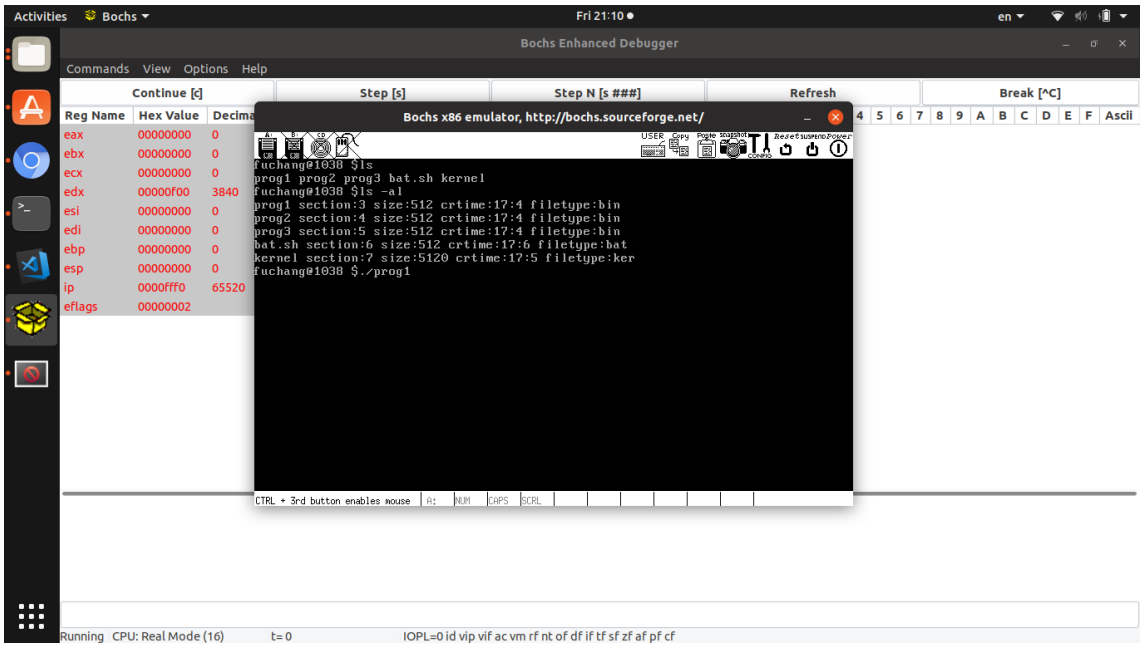

图 3: 输入 ls 以及 ls -al 指令之后，显示了文件名及详情信息

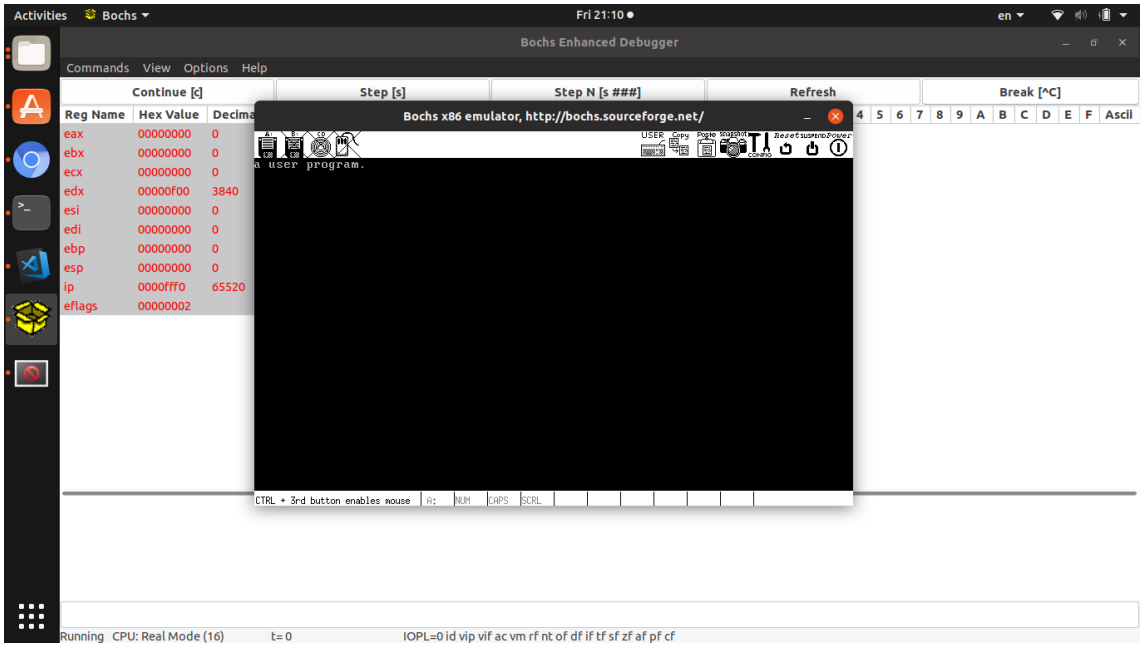

图 4: 输入"./prog1 之后，程序运行在新的窗口"，可以

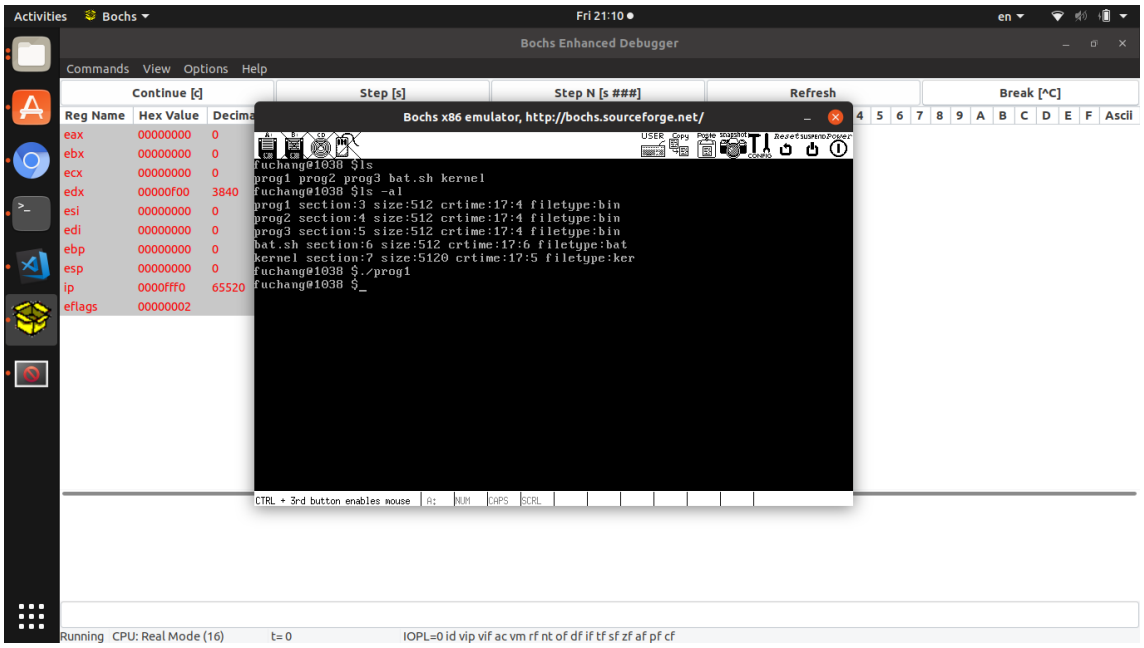

图 5: 用户程序 1 按回车之后，回到了主界面

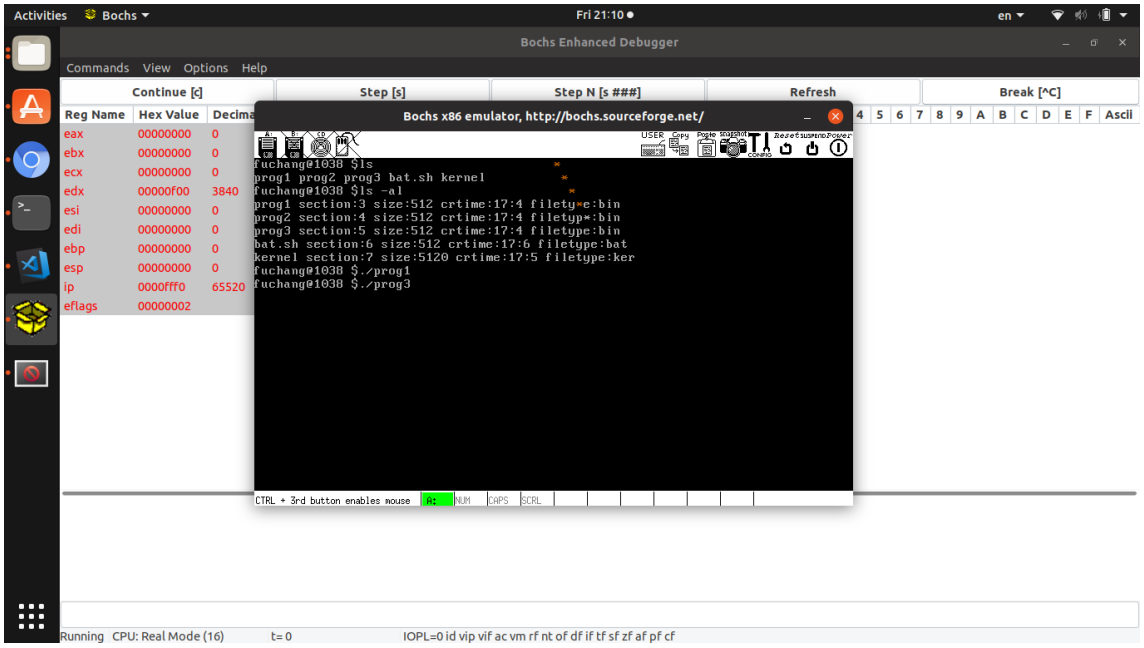

图 6: 执行用户三，该小球在原来界面执行

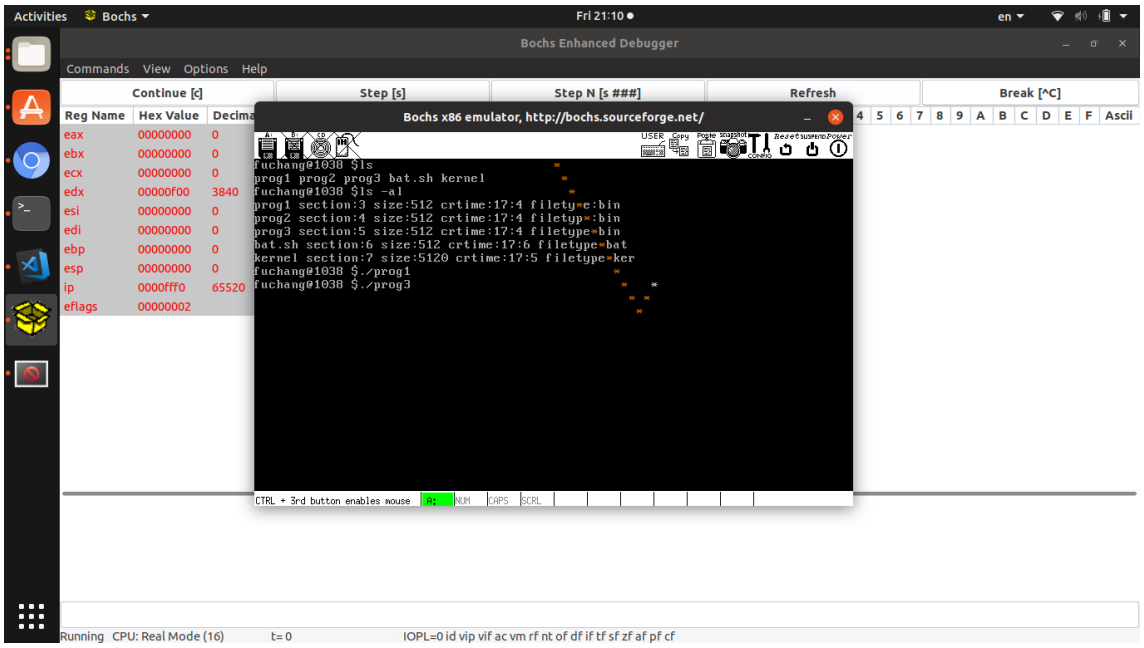

图 7: 用户三继续执行

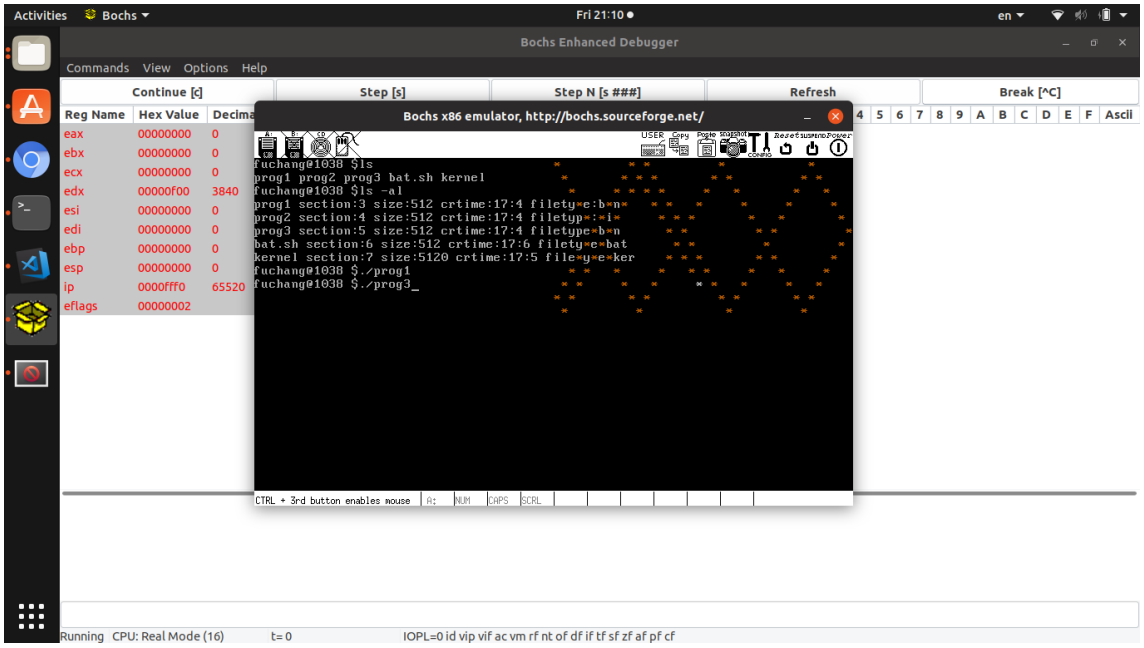

图 8: 用户三继续执行

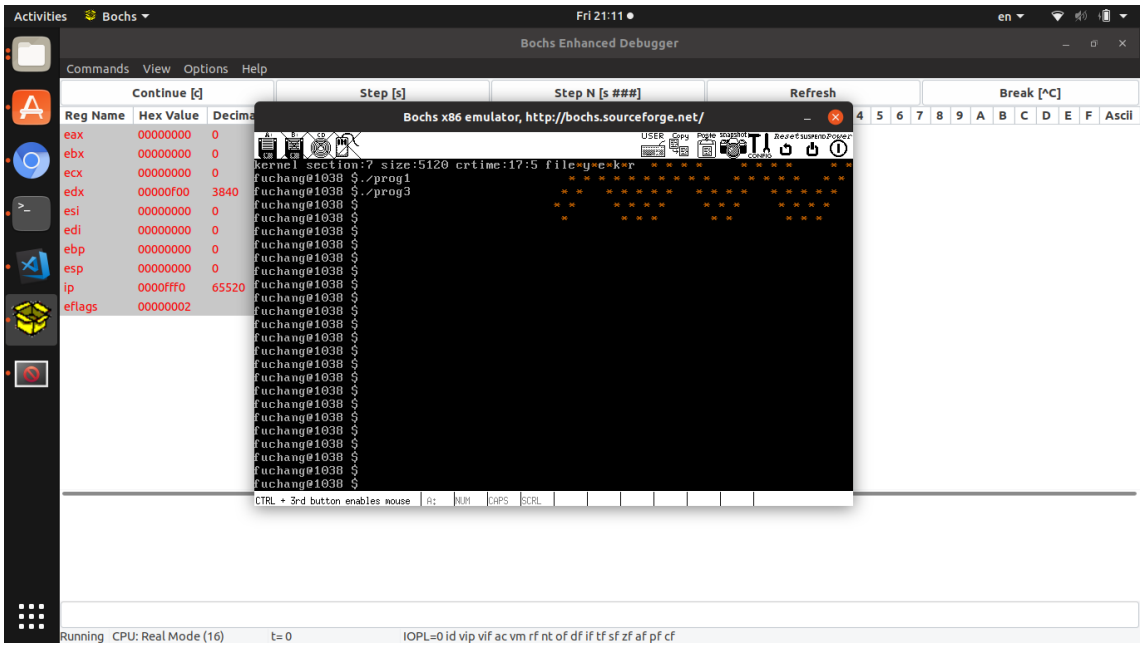

图 9: 这里的下卷功能非常方便，输入若干回车之后，保持下端标识完整

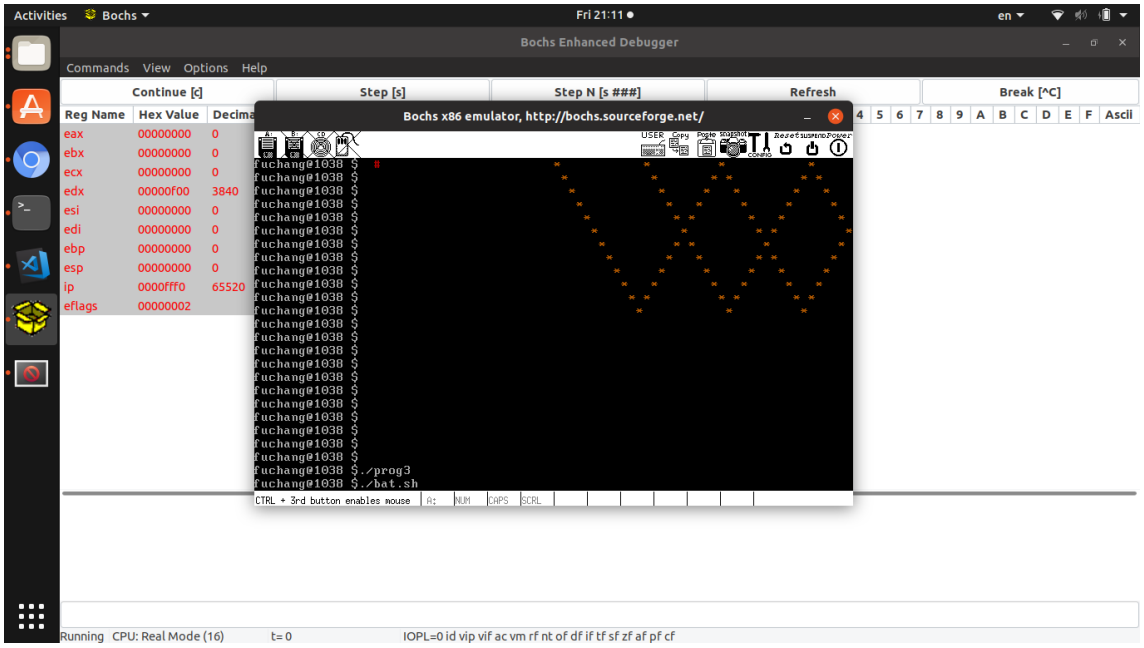

图 10: 执行用户批处理

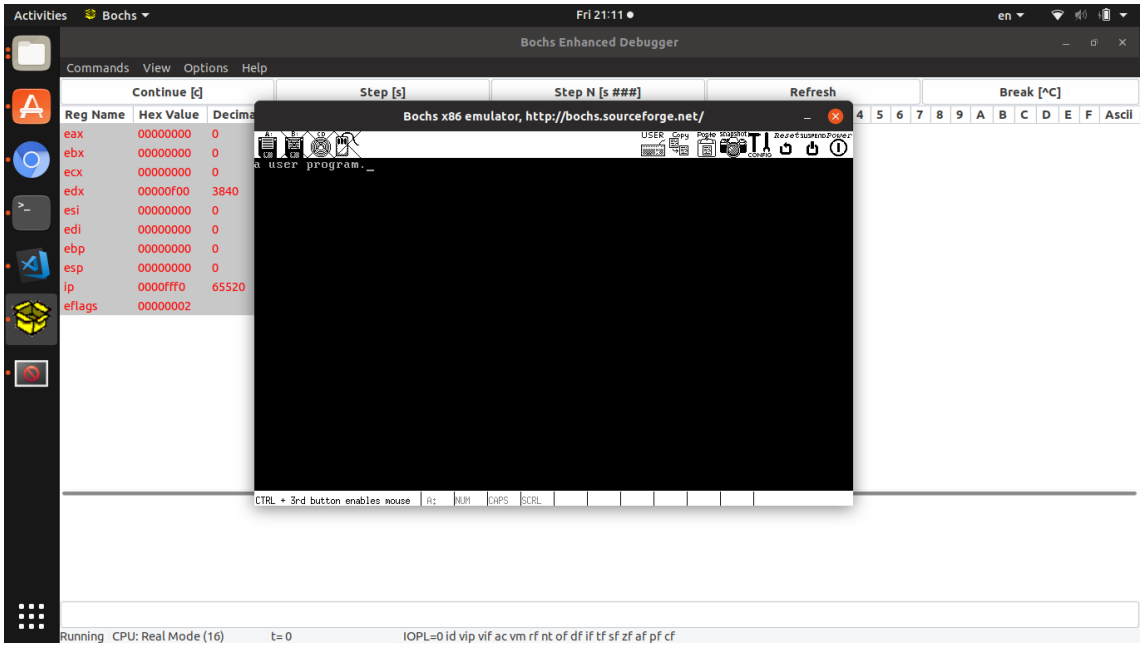

图 11: 程序跳入执行，并成功返回

五、 总结及心得体会:

第三次写操作系统实验，我的汇编代码第一次超过了 500+ 行，代码的组织和管理提高工作效率的作用上变得非常重要

第一次使用 nasm 和 gcc 交叉编译，很多问题在相互调用以及链接的过程中暴露无遗；我不得不补上一些相关的编译原理的知识，才能最终理解程序代码的 bug 所在

调试最久的一个 bug 是在 c 程序调用汇编然后从汇编 ret 回 c 函数内的时候，由于没有 call dword，导致从栈中取出了错误的返回地址，陷入了死循环。

另外一个错误是在 ld 的时候，由于我错误地使用了 section 语法，导致 ld 出来的 bin 文件中我的分区被全部打乱。我当时也被其他 bug 搞高的焦头烂额，没有心思再思考这个问题，只是现在觉得所有语法功能没有想象中的那么简单，不因当直接望文生义。