第三次作业

> 191250128 孙钰昇

第三次作业

- 实验背景知识
  - 如何进入保护模式?
  - 描述符
- 作业实现
  - 1.基础代码
  - 2. make run
  - 3.TAB支持
    - 输入
    - 删除（即对退格键的修改）
  - 4.清屏
    - 初始化的清屏
    - 每20秒的清屏
  - 5.查找模式
    - 按ESC进入查找模式和退出查找模式
    - 取消在查找模式下的清屏
    - 查找模式下的红色输入
    - 按下ESC撤销红色输入
    - 查找
    - 按下ENTER后屏蔽其他输入
  - 6.撤销操作
    - 识别
    - 撤销
- 完成中的一些问题
  - bochs无法输入?
    - 一些问题与解决
      - 报错
      - 解决
      - 报错
      - 解决

实验背景知识

如何进入保护模式?

- 准备GDT
- 用lgdt加载gdtr
- 打开A20
- 置cr0的PE位置
- 跳转，进入保护模式

描述符

- P位：存在位，用来标示段在内存中是否存在
- DPL位：描述符特权级，数字越小特权级越大
- S位：指明描述符是数据段/代码段描述符还是系统段/门描述符
- TYPE：用于定义描述符类型

表 3.1 描述符类型

TYPE 值	数据段和代码段描述符	系统段和门描述符
0	只读	<未定义>
1	只读，已访问	可用 286TSS
2	读/写	LDT
3	读/写，已访问	忙的 286TSS
4	只读，向下扩展	286 调用门
5	只读，向下扩展，已访问	任务门
6	读/写，向下扩展	286 中断门
7	读/写，向下扩展，已访问	286 陷阱门
8	只执行	<未定义>
9	只执行、已访问	可用 386TSS
A	执行/读	<未定义>
B	执行/读、已访问	忙的 386TSS
C	只执行、一致码段	386 调用门
D	只执行、一致码段、已访问	<未定义>
E	执行/读、一致码段	386 中断门
F	执行/读、一致码段、已访问	386 陷阱门

- G位：段界限粒度
- DB位
- AVL位

作业实现

1.基础代码

使用Orange‘s中的7n代码作为基础代码，当它正常运行时，可以发现，代码已经实现了输入回显功能、光标功能、退格功能、回车换行功能、大小写切换功能。因此，需要完成的任务只有以下几点：

- make run
- 增加对于TAB的支持
- 每二十秒和程序最开始进行一次清屏
- 对于退格键的修改，让退格键不再从行末删起
- 全部的查找功能
- 附加功能

2. make run

先前已经有了比较完备的makefile，只需要在基础上进行一些修改，添加make run命令依赖于make image。并运行bochs -f bochsrc。

添加的代码如下所示：

```
BOCHS = bochs
BOCHSFLAGS = -f bochsrc
run : image
    $(BOCHS) $(BOCHSFLAGS)
```

3.TAB支持

输入

首先在tty中添加能够解析TAB的方式。即添加一个case

```
case TAB:
    put_key(p_tty,'\t');
    break;
```

然后在console.c中完成对应的out_char

```
case '\t':
        if(p_con->cursor < p_con->original_addr + p_con->v_mem_limit -
TAB_WIDTH){
            for(int i=0;i<TAB_WIDTH;i++){
                *p_vmem++ = ' ';
                *p_vmem++ = DEFAULT_CHAR_COLOR;
                p_con->cursor++;
            }
        }
        break;
```

删除（即对退格键的修改）

考虑到要让退格键精准退到应有的位置，我们使用一个cursorStack来保存每一个字符输入后的位置。

在console.h中定义数据结构

```
typedef struct i_stack{
    unsigned int idx;//当前的下标
    unsigned int len;//数组的长度
    unsigned int array[SCREEN_SIZE];//储存用数组
}STACK;

/* CONSOLE */
typedef struct s_console
{
    unsigned int    current_start_addr; /* 当前显示到了什么位置      */
    unsigned int    original_addr;      /* 当前控制台对应显存位置 */
    unsigned int    v_mem_limit;        /* 当前控制台占的显存大小 */
    unsigned int    cursor;          /* 当前光标位置 */
    STACK *cursorStack; //用于记录每一个光标位置
}CONSOLE;
```

在console.c中定义push和pop

```
PRIVATE void push(CONSOLE* p_con,unsigned int pos){
    if(p_con->cursorStack->idx<p_con->cursorStack->len){
```

```
        p_con->cursorStack->array[p_con->cursorStack->idx] = pos;
        p_con->cursorStack->idx++;
    } else{
        disp_str("stackOverFlow");
    }
}
PRIVATE unsigned int pop(CONSOLE* p_con){
    if(p_con->cursorStack->idx-1>=0){
        unsigned int res = p_con->cursorStack->array[p_con->cursorStack->idx-1];
        p_con->cursorStack->idx--;
        return res;
    } else{
        disp_str("stackOverFlow");
        return p_con->cursorStack->len+1;
    }
}
```

最后对out_char中添加对应的调用

```
switch(ch) {
    case '\n':
        if (p_con->cursor < p_con->original_addr +
            p_con->v_mem_limit - SCREEN_WIDTH) {
            push(p_con,p_con->cursor);
            p_con->cursor = p_con->original_addr + SCREEN_WIDTH *
                ((p_con->cursor - p_con->original_addr) /
                 SCREEN_WIDTH + 1);
        }
        break;
    case '\b':
        if (p_con->cursor > p_con->original_addr&&p_con->cursorStack->idx!=0) {
            unsigned int idx = pop(p_con);
            if(idx!=(p_con->cursorStack->len+1)){
                int i=0;
                while(p_con->cursor>idx){
                    p_con->cursor--;
                    *(p_vmem-2-2*i) = ' ';
                    *(p_vmem-1-2*i) = DEFAULT_CHAR_COLOR;
                    i++;
                }
            }
        }
        break;
    case '\t':
        if(p_con->cursor < p_con->original_addr + p_con->v_mem_limit -
TAB_WIDTH){
            push(p_con,p_con->cursor);
            for(int i=0;i<TAB_WIDTH;i++){
                *p_vmem++ = ' ';
                *p_vmem++ = DEFAULT_CHAR_COLOR;
                p_con->cursor++;
            }
        }
        break;
    default:
        if (p_con->cursor <
            p_con->original_addr + p_con->v_mem_limit - 1) {
```

```
            push(p_con,p_con->cursor); //在操作之前保存，可以在需要时候pop出来
            *p_vmem++ = ch;
            *p_vmem++ = DEFAULT_CHAR_COLOR;
            p_con->cursor++;
        }
        break;
    }
```

4.清屏

初始化的清屏

只要在main.c中添加调用CleanScreen即可。输出一整个屏幕的空格，并将光标置于开始处

```
PUBLIC int cleanScreen(){
    disp_pos = 0;
    for (int i = 0 ; i < SCREEN_SIZE; ++i){
        disp_str(" ");
    }
    disp_pos = 0;
}
```

每20秒的清屏

在定时进程中添加清屏函数，并初始化所有的屏幕,延时20秒

```
void TestA()
{
    int i = 0;
    while (1) {
        cleanScreen();
        init_all_screen();
        milli_delay(100000);
    }
}
```

注意，用户任务是不能操作其他tty的，需要将TASKA从任务变为进程。

global.c中

```
PUBLIC  TASK    task_table[NR_TASKS] = {
    {task_tty, STACK_SIZE_TTY, "tty"},
    {TestA, STACK_SIZE_TESTA, "TestA"}};

PUBLIC  TASK    user_proc_table[NR_PROCS] = {

    {TestB, STACK_SIZE_TESTB, "TestB"},
    {TestC, STACK_SIZE_TESTC, "TestC"}};
```

proc.h中

```
/* Number of tasks & procs */
#define NR_TASKS    2
#define NR_PROCS    2
```

5.查找模式

按ESC进入查找模式和退出查找模式

在global中定义mode，其中1代表查找模式，0代表一般模式

在tty.c的in_process中针对ESC做出解析

```
case ESC:
     mode = (mode == 1)?0:1;
     break;
```

取消在查找模式下的清屏

在main.c中进行对应的修改

```
void TestA()
{
    int i = 0;
    while (1) {
        if(mode==0){
            cleanScreen();
            init_all_screen();
            milli_delay(100000);
        } else{
            milli_delay(10);
        }
    }
}
```

查找模式下的红色输入

在console.c中设置对应的输出颜色

```
default:
        if (p_con->cursor <
            p_con->original_addr + p_con->v_mem_limit - 1) {
            push(p_con,p_con->cursor); //在操作之前保存，可以在需要时候pop出来
            if(mode==0&&p_con->cursor>p_con->endOfNormalCursor){
                p_con->endOfNormalCursor = p_con->cursor;
            }
            *p_vmem++ = ch;
            if(mode==0||ch==' '){
                //注意把空格颜色设置成白色，便于和蓝色的TAB区分
                *p_vmem++ = DEFAULT_CHAR_COLOR;
            } else{
                *p_vmem++ = RED;
            }
            p_con->cursor++;
        }
        break;
    }
```

按下ESC撤销红色输入

利用没有使用的/r标记ESC，在out_char中解析ESC，并将光标退格到进入查找模式前的地方

```
case '\r':
        //作为ESC的代表字符
        if(mode==1){
            p_con->endOfNormalCursor = p_con->cursor;
        } else{
            if (p_con->cursor > p_con->original_addr){
                int i=0;
                while(p_con->cursor>p_con->endOfNormalCursor){
                    unsigned int idx = pop(p_con);
                    //将光标栈出栈到合适的位置
                    while(p_con->cursor>idx){
                        p_con->cursor--;
                        *(p_vmem-2-2*i) = ' ';
                        *(p_vmem-1-2*i) = DEFAULT_CHAR_COLOR;
                        i++;
                    }
                }
            }
            for(int i=0;i<p_con->cursor;i++){
                *(u8*)(V_MEM_BASE + i * 2+1)=DEFAULT_CHAR_COLOR;
            }
        }
        break;
```

查找

通过显存查找

```
PRIVATE void searchString(CONSOLE *p_con){
    int len = p_con->cursor-p_con->endOfNormalCursor;//待匹配字符串长度
    if(len==0){
        return;
    }
    u8* p_vmem;
    u8* tar_vmem;
    u8* p_color;
    u8* tar_color;
    for(int i=0;i<p_con->endOfNormalCursor;i++){
        int found = 1;
        for(int j=0;j<len;j++){
            p_vmem = (u8*)(V_MEM_BASE + i * 2 + j*2);
            p_color = (u8*)(V_MEM_BASE + i * 2 + j*2+1);
            tar_vmem = (u8*)(V_MEM_BASE + p_con->endOfNormalCursor * 2 + j*2);
            tar_color = (u8*)(V_MEM_BASE + p_con->endOfNormalCursor * 2 +
j*2+1);
            if(*p_vmem!=*tar_vmem||(*p_vmem==' '&&*p_color!=*tar_color)){
                //通过颜色的比较来判断空格和TAB
                found = 0;
                break;
            }
        }
        if(len>p_con->endOfNormalCursor){
            found = 0;
```

```
            //排除匹配字符比原文长的情况
        }
        if(found==1){
            for(int k=0;k<len;k++){
                *(u8*)(V_MEM_BASE + i * 2+k*2+1)=RED;
            }
        }
    }
}
```

按下ENTER后屏蔽其他输入

将mode==2设为屏蔽模式。对mode==2只接受ESC输入

```
if(mode==2){
        if(ch=='\r'){
            mode = 0;
        } else{
            return;
        }
    }
```

6.撤销操作

识别

在global中定义，在keyboard.c中实现

```
// global.h
extern int ctrl;

// global.c
PUBLIC int ctrl;
//用0表示正常，1表示按下ctrl

//keyboard.c
ctrl = ctrl_l||ctrl_r;
```

撤销

本质上是记录每一步输出什么字符，组成一个队列。当撤销时，就从队尾排除一个，再重新执行输出操作。

```
//console.h
typedef struct c_stack{
    int idx;//当前的下标
    int seperator;//记录/r的下标
    char ch[SCREEN_SIZE];//储存每一步的字符
}CHARSTACK;
typedef struct s_console
{
    unsigned int    current_start_addr; /* 当前显示到了什么位置      */
    unsigned int    original_addr;      /* 当前控制台对应显存位置 */
    unsigned int    v_mem_limit;        /* 当前控制台占的显存大小 */
    unsigned int    cursor;         /* 当前光标位置 */
```

```
    STACK *cursorStack; //用于记录每一个光标位置
    CHARSTACK charStack;//用以记录每一步进行了什么操作
    unsigned int    endOfNormalCursor;//记录进入查找模式前的光标位置
}CONSOLE;
```

在tty.c中完成记录

```
PRIVATE void tty_do_write(TTY* p_tty)
{
    if (p_tty->inbuf_count) {
        char ch = *(p_tty->p_inbuf_tail);
        p_tty->p_inbuf_tail++;
        if (p_tty->p_inbuf_tail == p_tty->in_buf + TTY_IN_BYTES) {
            p_tty->p_inbuf_tail = p_tty->in_buf;
        }
        p_tty->inbuf_count--;
        if(ch!='\r'){
            push_charStack(p_tty->p_console,ch);
        } else{
            set_charStackSeperator(p_tty->p_console);
        }
        out_char(p_tty->p_console, ch);
    }
}
PUBLIC void push_charStack(CONSOLE* p_con,char c){
    p_con->charStack.ch[p_con->charStack.idx]=c;
    p_con->charStack.idx++;
}
PUBLIC void set_charStackSeperator(CONSOLE* p_con){
    p_con->charStack.seperator=p_con->charStack.idx;
}
```

在console.c中完成撤销

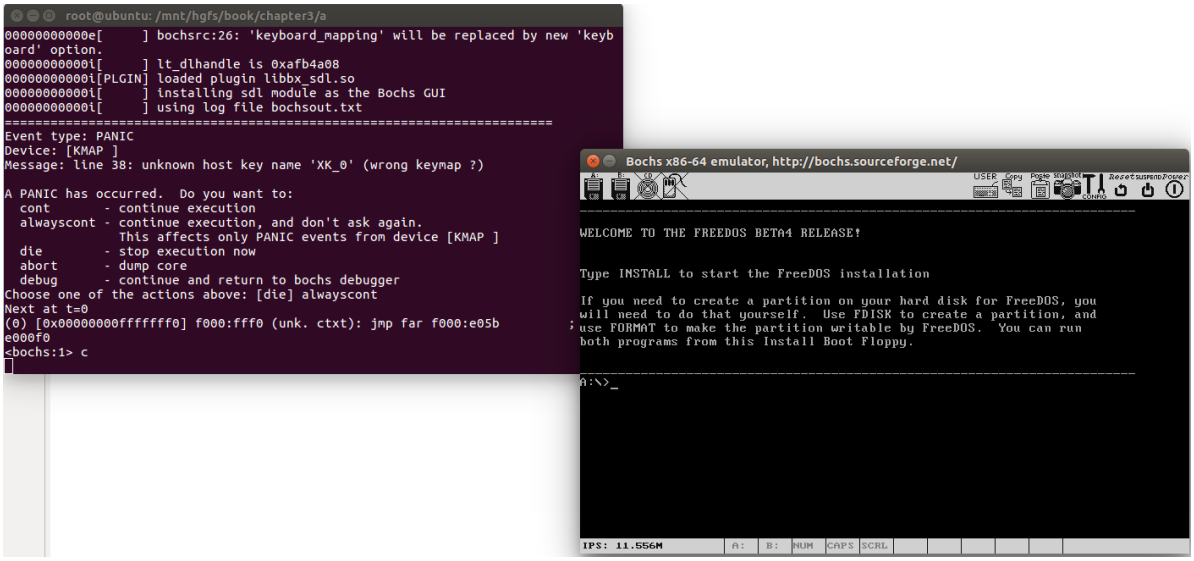
```
/*======================================================================*
                撤销操作
 *======================================================================*/
PUBLIC void doCtrlZ(CONSOLE *p_con){
    if(mode==0){
        cleanScreen();
        p_con->cursorStack->idx=0;
        p_con->cursor = disp_pos / 2;
        //初始化指针
        flush(p_con);
        redo(p_con);
    }
    if(mode==1){
        p_con->cursorStack->idx=p_con->charStack.seperator;
        disp_pos = p_con->endOfNormalCursor * 2;
        for (int i = 0 ; i < SCREEN_SIZE; ++i){
            disp_str(" ");
        }
        disp_pos = p_con->endOfNormalCursor * 2;
        p_con->cursor = disp_pos / 2;
        //初始化指针
```

```
            flush(p_con);
            redo(p_con);
        }
    }
    PUBLIC void redo(CONSOLE *p_con){
        int start = 0;
        if(mode==1){
            start = p_con->charStack.seperator;
        }
        p_con->charStack.idx-=2;
        if(p_con->charStack.idx<=0){
            p_con->charStack.idx=0;
            return;
            //已经清空
        }
        for(int i=start;i<p_con->charStack.idx;i++){
            out_char(p_con,p_con->charStack.ch[i]);
        }
    }
```

完成中的一些问题

bochs无法输入?

刚刚试图用freeDos启动后，无法获取键盘输入。



一些问题与解决

> 一些前人的攻略：(45条消息) 操作系统内核Hack：(一)实验环境搭建_西代零零发-CSDN博客

报错

`Message: dlopen failed for module 'x': file not found`

解决

在安装的时候，少安装了个bochs-x包，安装即可。

`apt-get install bochs-x`

报错

`Message: unkonwn host key name “XK”`

解决

注释掉

`#keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map`