

# Lab3

## 1 准备

实验以《orange's: 一个操作系统的实现》的代码为基础

### 1.1 运行

#### 问题1

make image失败

```
gcc -I include/ -c -fno-builtin -o kernel/proc.o kernel/proc.c
nasm -I include/ -f elf -o lib/kliba.o lib/kliba.asm
gcc -I include/ -c -fno-builtin -o lib/klib.o lib/klib.c
nasm -I include/ -f elf -o lib/string.o lib/string.asm
ld -s -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/syscall.o kernel/start.o kernel/main.o kernel/clock.o kernel/keyboard.o kernel/tty.o kernel/console.o kernel/i8259.o kernel/global.o kernel/protect.o kernel/proc.o lib/kliba.o lib/klib.o lib/string.o
kernel/keyboard.o: In function `keyboard_read':
keyboard.c:(.text+0x5a5): undefined reference to `__stack_chk_fail'
kernel/tty.o: In function `in_process':
tty.c:(.text+0x1f2): undefined reference to `__stack_chk_fail'
kernel/protect.o: In function `exception_handler':
protect.c:(.text+0x64f): undefined reference to `__stack_chk_fail'
lib/klib.o: In function `disp_int':
klib.c:(.text+0xe8): undefined reference to `__stack_chk_fail'
Makefile:70: recipe for target 'kernel.bin' failed
make: *** [kernel.bin] Error 1
zhhc@zhhc-VirtualBox:~/OS/lab3/n$
```

解决方法

```
13 # Programs, flags, etc.
14 ASM      = nasm
15 DASM     = ndisasm
16 CC       = gcc
17 LD       = ld
18 ASMBFLAGS = -I boot/include/
19 ASMKFLAGS = -I include/ -f elf
20 CFLAGS    = -I include/ -c -fno-builtin -fno-stack-protector
21 LDFLAGS   = -s -Ttext $(ENTRYPOINT)
22 DASMFLAGS = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)
23
```

结果

```
1+0 records in
1+0 records out
512 bytes copied, 0.000253721 s, 2.0 MB/s
sudo mount -o loop a.img /mnt/floppy/
[sudo] password for zhhc:
sudo cp -fv boot/loader.bin /mnt/floppy/
'boot/loader.bin' -> '/mnt/floppy/loader.bin'
sudo cp -fv kernel.bin /mnt/floppy
'kernel.bin' -> '/mnt/floppy/kernel.bin'
sudo umount /mnt/floppy
zhhc@zhhc-VirtualBox:~/OS/lab3/n$
```

## 问题2

make image 结束之后，执行下面的命令失败

```
1 bochs -f bochsrc
```

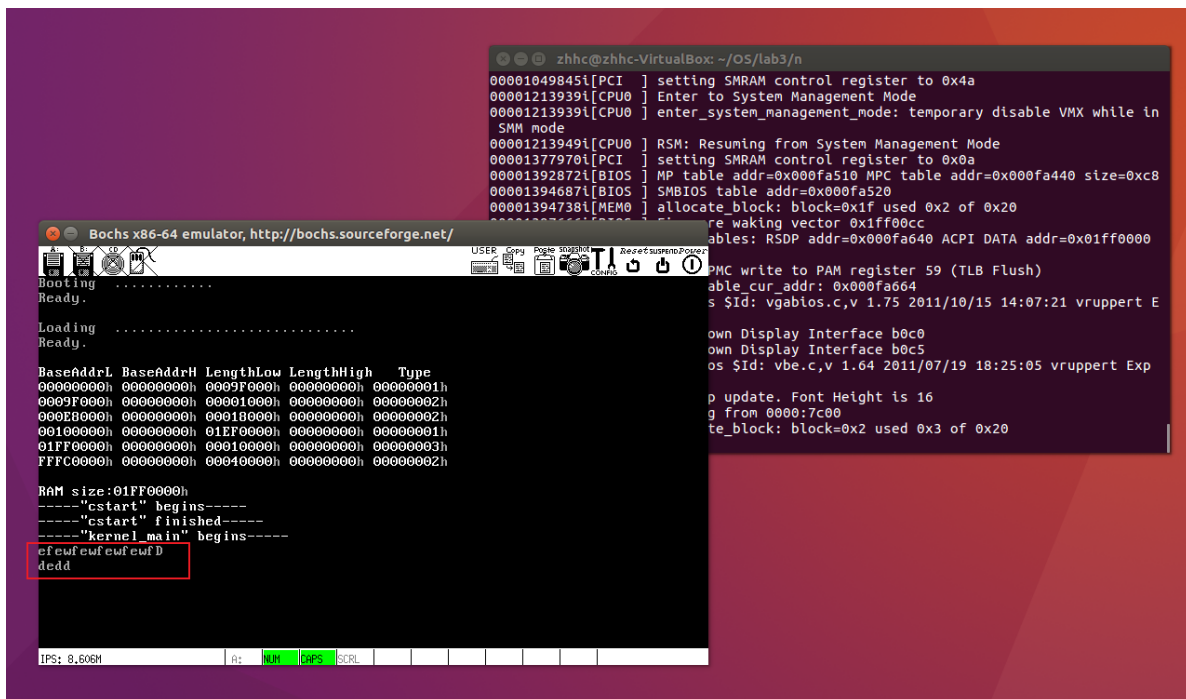
报错信息

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

解决方法

```
1 sudo apt-get install bochs-x
```

结果显示



## 1.2 已实现和待实现

### 已实现

1. 从屏幕左上角开始，以白色显示键盘输入的字符，可以输入并显示a-z, A-Z和0-9字符。
2. 支持大小写切换包括 Shift 组合键以及大写锁定两种方式，大写锁定后再用 Shift 组合键将会输入小写字母
3. 支持回车键换行
4. 支持用退格键删除输入内容
5. 支持空格键
6. 有光标显示
7. 输入字符无上限

## 待实现

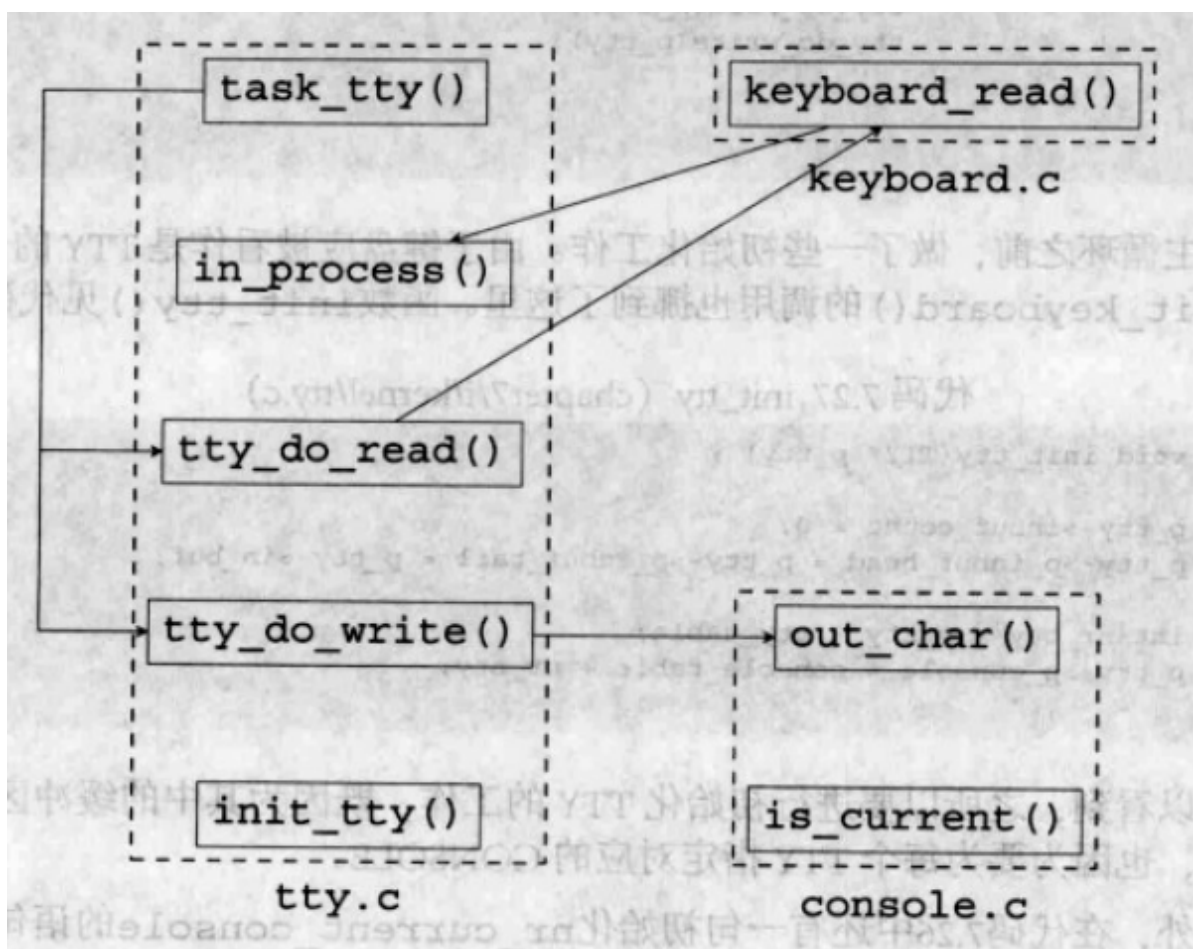
- ✓ make run 直接运行
- ✓ 支持Tab键
- ✓ 清空屏幕以及每隔20秒清空屏幕
- ✓ 退格换行和TAB需要一次完成
- ✓ 查找功能
- ✓ control + z 组合键撤回

## 2 理解代码

敲击键盘所产生的编码被称作扫描码 (Scan Code)，它分为 Make Code 和 Break Code 两类。当一个键被按下或者保持住按下时，将会产生 Make Code，当键弹起时，产生 Break Code。除了 Pause 键之外，每一个按键都对应一个 Make Code 和一个 Break Code。

首先，需要理解代码，可以照着《orange's：一个操作系统的实现》第七章的讲解一起看。

输出一个字符的函数调用逻辑

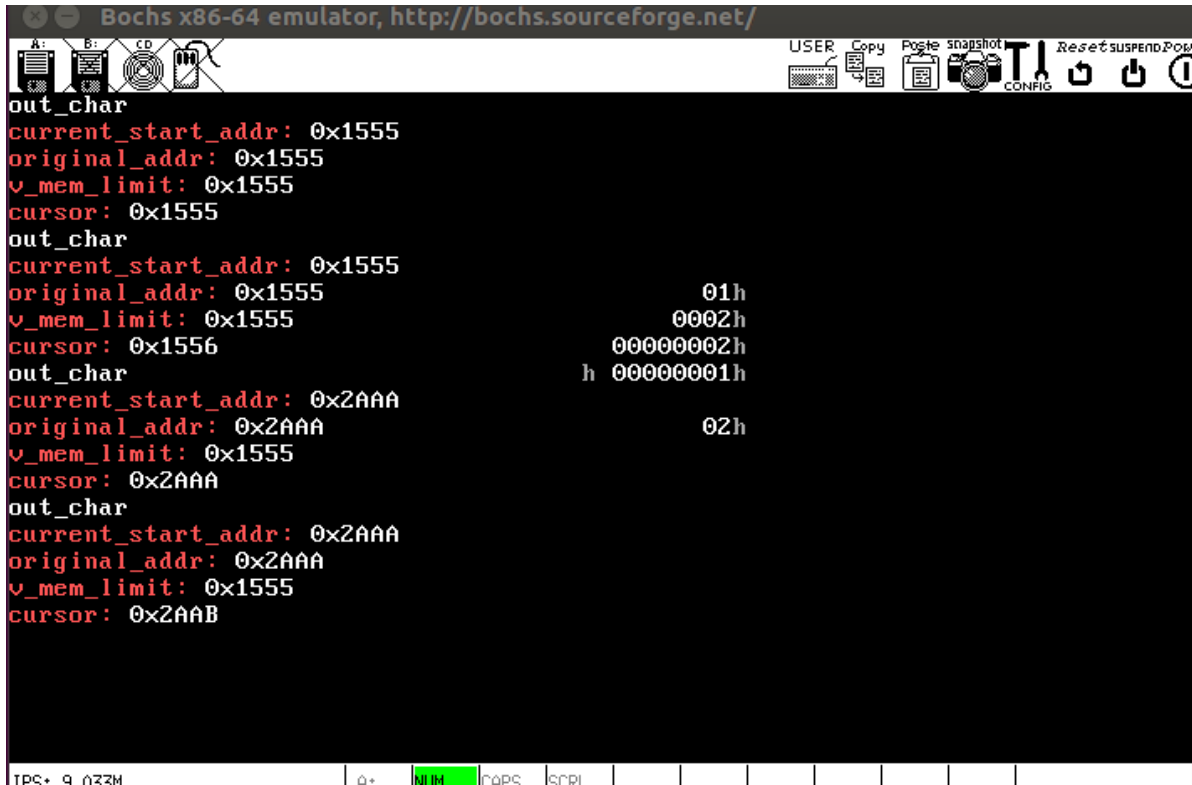


当初始化结束后，就会进入到 `task_tty()` 函数中，进行不断的循环。

## 探究CONSOLE结构体中四个变量的含义

```
/* 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 */
}CONSOLE;
```

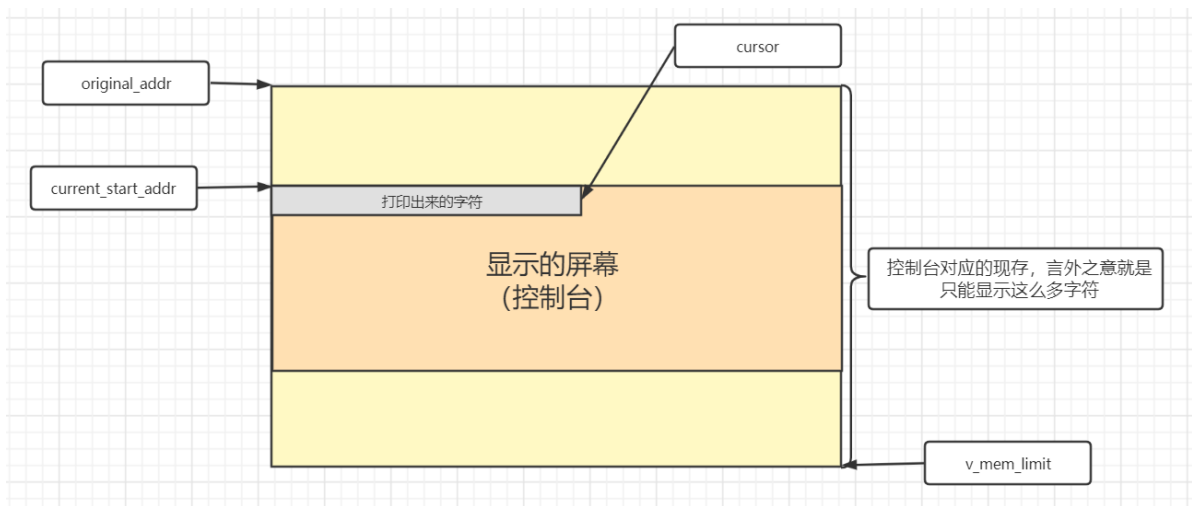
### 分析过程



屏幕一行80个字符，屏幕总共25行。

在敲字符过程中，cursor 在不断加1，original\_addr 始终为 0x0，current\_start\_addr 是当前屏幕可见范围的开始位置距离最开始显示位置的偏移。





## 3 实现功能

### 3.1 TAB

在 `keyboard.h` 中可以发现TAB是一个不可显示的字符，所以在 `tty.c` 的 `in_process` 中要增加对于TAB 的识别，将其放进缓冲区内。然后在 `console.c` 的 `out_char` 函数中增加对 `\t` 的输出处理。

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

```
case '\t':
    if (p_con->cursor <
        p_con->original_addr + p_con->v_mem_limit - 1) {
        p_cbuf->buf[p_cbuf->buf_cur_idx].before_cursor = p_con->cursor;
        for (int i = 0; i < TAB_SIZE; i++) {
            *p_vmem++ = ' ';
            *p_vmem++ = p_cbuf->buf[p_cbuf->buf_cur_idx].color;
            p_con->cursor++;
        }
        p_cbuf->buf[p_cbuf->buf_cur_idx].after_cursor = p_con->cursor;
    }
    break;
```

### 3.2 清屏

清屏的功能实现在逻辑上比较简单，只要不断地调用 `out_char()` 函数，传入 `\b`，直到 `cursor` 回到 `original_addr` 位置。

但是如何在实现每20秒清屏一次呢？要理解 `kernel_main()` 函数中的任务，可以新增一个任务，`task_clear_screen`，然后在其中执行清屏任务，并且每次执行完毕后延迟20s。

1、将 `task_clear_screen` 声明成一个任务，不断执行

在 `global.c` 中做相应修改，并修改对应的宏



```

PUBLIC PROCESS proc_table[NR_TASKS + NR_PROCS];

PUBLIC TASK task_table[NR_TASKS] = {
    {task_tty, STACK_SIZE_TTY, "tty"},
    {task_clear_screen, STACK_SIZE_CLEAR, "clear_screen"}
};

PUBLIC TASK user_proc_table[NR_PROCS] = {
    {TestA, STACK_SIZE_TESTA, "TestA"},
    {TestB, STACK_SIZE_TESTB, "TestB"},
    {TestC, STACK_SIZE_TESTC, "TestC"}};

PUBLIC char task_stack[STACK_SIZE_TOTAL];

```

2、在 `task_clear_screen` 任务中完成清屏逻辑，并且延迟20s

```

PUBLIC void task_clear_screen(){
    while(1){
        clean_screen();
        milli_delay(50000);
    }
}

PRIVATE void clean_screen(){
    TTY* p_tty;
    for(p_tty = TTY_FIRST; p_tty < TTY_END; p_tty++){
        if(is_current_console(p_tty->p_console)){
            while(p_tty->p_console->cursor != p_tty->p_console->original_addr){
                out_char(p_tty->p_console, '\b');
            }
        }
    }
}

```

每个任务中while循环不能break，否则执行会报错。

### 3.3 退格一次完成

源代码的删除功能只实现了一个一个删除，即使是对于 `\n` 和 `\t` 也是如此，不符合实际情况，需要做修改，实现一次删除 `\t` 和 `\n`。

#### 思路

要想实现一次退格，就需要知道前一个字符是什么，如果是普通字符那么让光标位置减一即可，如果是特殊的 `\n` 和 `\t` 就需要特殊处理，使光标回到按下 `\n` 和 `\t` 前的位置。

对于 `\t` 还好说，只要把光标减4即可；但是对于 `\n` 来说，就有点麻烦了。

一开始，想让光标一直往回移动直到遇到不是空格的字符。但是如果在输入 `\n` 之前刚输入了空格呢？这样显然就不对，而且实现起来还很复杂。

所以需要借助新的数据结构来存储已经显示的字符，并且对显示的字符做一个包装，做成一个结构体。

```

26     typedef struct char_in_screen {
27         u32 val; // 字符的值
28         unsigned int before_cursor; // 输出该字符前的光标位置
29         unsigned int after_cursor; // 输出该字符后的光标位置
30     } CHAR;
31

```

再构建一个存储当前屏幕中所有显示的字符的结构体

```

32     typedef struct char_buf {
33         CHAR buf[BYTES_IN_BUF]; // 存储当前所有显示的字符
34         int buf_cur_idx;         // 指向当前最新显示的字符
35     } C_BUF;
36

```

然后需要在适当的地方将这个数据结构插入

## 过程

### 1、初始化C\_BUF结构体

```

PUBLIC void task_tty() {
    disp_str("task_tty... \n");
    TTY* p_tty;

    init_keyboard();

    for (p_tty = TTY_FIRST; p_tty < TTY_END; p_tty++) {
        init_tty(p_tty);
    }

    init_char_buf(p_cbuf);

    select_console(0);
    while (1) {
        for (p_tty = TTY_FIRST; p_tty < TTY_END; p_tty++) {
            tty_do_read(p_tty);
            tty_do_write(p_tty);
        }
    }
}

```

### 2、在将字符放入TTY中的同时，也将封装起来的字符放进C\_BUF中



```

PRIVATE void put_key(TTY* p_tty, u32 key) {
    if (p_tty->inbuf_count < TTY_IN_BYTES) {
        // 将字符送到char_buf中
        CHAR c;
        c.val = key;
        p_cbuf->buf[p_cbuf->buf_cur_idx] = c;

        *(p_tty->p_inbuf_head) = key;
        p_tty->p_inbuf_head++;
        if (p_tty->p_inbuf_head == p_tty->in_buf + TTY_IN_BYTES) {
            p_tty->p_inbuf_head = p_tty->in_buf;
        }
        p_tty->inbuf_count++;
    }
}

```

### 3、在 out\_char 函数中实现对应逻辑

在其他输出字符的位置需要加上对CHAR结构体中before\_cursor和after\_cursor的赋值，例如：

```

default:
    if (p_con->cursor <
        p_con->original_addr + p_con->v_mem_limit - 1) {
        p_cbuf->buf[p_cbuf->buf_cur_idx].before_cursor = p_con->cursor;
        *p_vmem++ = ch;
        *p_vmem++ = DEFAULT_CHAR_COLOR;
        p_con->cursor++;
        p_cbuf->buf[p_cbuf->buf_cur_idx].after_cursor = p_con->cursor;
    }
    break;

```

然后在处理退格键时就可以用如下一段代码解决

```

case '\b':
    // 光标位置需要在多于起始位置
    if (p_con->cursor > p_con->original_addr) {
        if (p_cbuf->buf_cur_idx >= 1)
            p_con->cursor = p_cbuf->buf[p_cbuf->buf_cur_idx - 1].before_cursor;
        else
            p_con->cursor--;
        *(p_vmem - 2) = ' ';
        *(p_vmem - 1) = DEFAULT_CHAR_COLOR;
    }
    break;

```

### 4、显示字符结束后，需要修改C\_BUF中的 buf\_cur\_idx

```

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--;

        out_char(p_tty->p_console, ch);

        if (ch == '\b') {
            if(p_cbuf->buf_cur_idx > 0)
                p_cbuf->buf_cur_idx--;
        } else {
            p_cbuf->buf_cur_idx++;
        }
    }
}

```

这么设计就可以把所有的字符统一起来处理，在退格时只要回到按下该键前的光标位置即可。

注意需要把p\_cbuf作为全局变量声明，这样可以在所有文件中使用，而不用作为函数参数传来传去。

在global.c和global.h中做声明即可

### 3.4 查找功能

#### 思路

明确几个状态(代码也是依据状态编写的)

- 1、正常状态。以黑底白字显示字符，一切正常
- 2、搜索状态。在正常状态按下esc后，以黑底红字显示字符，并记录这段时间敲下的字符。
- 3、匹配状态。搜索状态按下enter后，进行match，思路是从头遍历C\_BUF结构体数组，找到匹配的字符串后把相应的字符颜色改成黑底红字，再从头显示。
- 4、从匹配态退出。删去搜索字符串，把黑底红字的改成黑底白字，重新显示。

#### 过程

- 1、定义一些全局变量和常量，对CHAR结构体进行调整(增加color字段)

```

typedef struct char_in_screen {
    char val; // 字符的值
    char color; // 字符颜色
    unsigned int before_cursor; // 输出该字符前的光标位置
    unsigned int after_cursor; // 输出该字符后的光标位置
} CHAR;

```

```
PUBLIC int cur_mode; // 记录当前处于什么状态 用于搜索 INIT SEARCH MATCH
PUBLIC char match_str[MATCH_LEN]; // 记录搜索字符串
PUBLIC int match_str_idx;
```

```
/* search mode */
#define INIT 0 // 正常状态
#define SEARCH 1 // 搜索状态
#define MATCH 2 // 匹配状态
#define MATCH_LEN 256 // 搜索字符串最大长度
```

状态的变化

```
1 INIT(按ESC) -> SEARCH(按Enter) -> MATCH(按ESC) -> INIT
```

2、在每个状态编写相应逻辑

### INIT

在 `in_process` 函数中增加对ESC的识别

```
case ESC:
    if(cur_mode == INIT) {
        cur_mode = SEARCH;
        init_search();
    }
    break;
```

这个初始化要小心，每次进入搜索状态都要进行初始化

```
PRIVATE void init_search(){
    match_str_idx = 0;
}
```

### SEARCH

改变in\_process中对Enter键的处理

```
case ENTER:
    if(cur_mode == SEARCH){
        cur_mode = MATCH;
        do_match(p_tty->p_console);
    } else {
        put_key(p_tty, '\n');
    }
    break;
```

```

PRIVATE void do_match(CONSOLE* p_con){
    int i = 0; int len = p_cbuf->buf_cur_idx;
    while(i < len){
        int k = i; int j = 0;
        // 匹配搜索字符串
        for(; j < match_str_idx; j++){
            if(p_cbuf->buf[i].val != match_str[j]){
                break;
            } else {
                i++;
            }
        }
        if(j == match_str_idx){
            // 改颜色
            for(int m = k; m < i; m++){
                p_cbuf->buf[m].color = RED_IN_BLACK;
            }
        } else {
            i = k + 1;
        }
    }
    // 重新输出
    p_con->cursor = p_con->original_addr;
    p_cbuf->buf_cur_idx = 0;
    for(int t = 0; t < len; t++){
        out_char(p_con, p_cbuf->buf[t].val);
    }
}

```

## MATCH

在in\_process函数开头加上判断，只接收ESC

```

PUBLIC void in_process(TTY* p_tty, u32 key) {
    // 如果处于匹配阶段 不处理除esc外任何输入
    if(cur_mode == MATCH){
        if((key & MASK_RAW) == ESC){
            exit_search(p_tty->p_console);
            cur_mode = INIT;
        }
        return;
    }
}

```

```

PRIVATE void exit_search(CONSOLE* p_con){
    // 删除搜索字符串
    for(int i = 0; i < match_str_idx; i++){
        out_char(p_con, '\b');
    }
    // 改变字体颜色
    int len = p_cbuf->buf_cur_idx;
    for(int i = 0; i < len; i++){
        if(p_cbuf->buf[i].color == RED_IN_BLACK){
            p_cbuf->buf[i].color = DEFAULT_CHAR_COLOR;
        }
    }
    // 从头显示
    p_con->cursor = p_con->original_addr;
    p_cbuf->buf_cur_idx = 0;
    for(int i = 0; i < len; i++){
        out_char(p_con, p_cbuf->buf[i].val);
    }
}

```

在out\_char函数中也要做相应处理

- 1、MATCH状态下是不需要向C\_BUF数组中增添CHAR的，只做输出。
- 2、SEARCH状态下要记录搜索字符串，并且注意SEARCH状态下是黑底红色。

```

PUBLIC void out_char(CONSOLE* p_con, char ch) {

    if(cur_mode != MATCH){
        CHAR c;
        c.val = ch;
        c.color = cur_mode == INIT ? DEFAULT_CHAR_COLOR : RED_IN_BLACK;
        p_cbuf->buf[p_cbuf->buf_cur_idx] = c;
    }

    if(cur_mode == SEARCH){
        if(ch == '\b'){
            match_str_idx--;
        } else {
            match_str[match_str_idx++] = ch;
        }
    }
}

```

注意：在退格搜索字符串( SEARCH 状态)，不能将原来的字符删除

### 3.5 control+z撤销

这个也很简单，首先要判断出是否是按下 `ctrl+z` 组合键，然后执行退格操作即可。

似乎没有这么简单，因为既要撤销显示出来的字符，还要撤销删除，以及需要能够一直撤销直到初始状态。

想法1：再建立一个ACTION的列表，记录所有的操作。但是实现起来可能比较复杂，因为需要同步好几个数据结构数组之间的关系。

想法2：改造C\_BUF数据结构。其实撤销的主要难点在于撤销退格，所以在C\_BUF中使用两个指针。也不行，可操作性不高，指针移动比较复杂。

最终采用想法1，需要理清ACTION数组在什么位置更新！

### 1、创建数据结构

其中 MAX\_ACTION 为1000，即支持大约1000次撤销。

```
typedef struct action {
    CHAR this;
    CHAR relate;
} ACTION;

typedef struct actoin_list{
    ACTION list[MAX_ACTION];
    int idx;
} ACT_LIST;
```

### 2、初始化ACTION列表

在task\_tty函数中调用初始化函数即可

```
PRIVATE void init_act_list(){
    p_act_list = &act_list;
    p_act_list->idx = 0;
}
```

### 3、向列表中增加ACTION

经过思考，选在tty\_do\_write函数的out\_char后面

```
out_char(p_tty->p_console, ch);

ACTION action;
if(ch == '\b'){
    action.this = p_cbuf->buf[p_cbuf->buf_cur_idx+1];
    action.relate = p_cbuf->buf[p_cbuf->buf_cur_idx];
} else {
    action.this = p_cbuf->buf[p_cbuf->buf_cur_idx-1];
}
p_act_list->list[p_act_list->idx] = action;
p_act_list->idx++;
```

#### 4、撤销操作

在in\_process函数中增加以下逻辑，就是在判断出是 `Ctrl+z` 组合键后，回退ACTION数组，做逆操作。

```
// 可显示字符
if (!(key & FLAG_EXT)) {
    char c = key;
    if( c == 'z' ){
        if((key & FLAG_CTRL_L) || (key & FLAG_CTRL_R)){
            if(p_act_list->idx > 0){
                ACTION last_action = p_act_list->list[p_act_list->idx-1];
                if(last_action.this.val == '\b'){
                    out_char(p_tty->p_console, last_action.relate.val);
                } else {
                    out_char(p_tty->p_console, '\b');
                }
                p_act_list->idx --;
            }
            return;
        }
    }
}
put_key(p_tty, key);
```

注意虚拟机中的热键 VirtualBox默认是 Right Ctrl

导致Right Ctrl按不出来

### 3.6 总结

由于上面的描述是写完一个功能后立刻写下的，所以导致代码前后会出现不一致。

十分重要的调整是【将字符加进C\_BUF结构体中buf】的位置和【更新C\_BUF结构体中buf\_cur\_idx】的位置，最后是都放在了out\_char函数中。