冯亚林 191850036 南京大学软件学院

1、获取源码

从gitee仓库获取了第七章的源码，我选择文件夹 m 中的部分用作框架。

oranges: 《ORANGE'S：一个操作系统的实现》源码 (gitee.com)

2、前期准备

1.大概阅读助教给的参考资料和源代码

阅读几份参考资料；

2.调试程序环境，运行样例代码，准备乱改改

3、修改makefile，运行源码

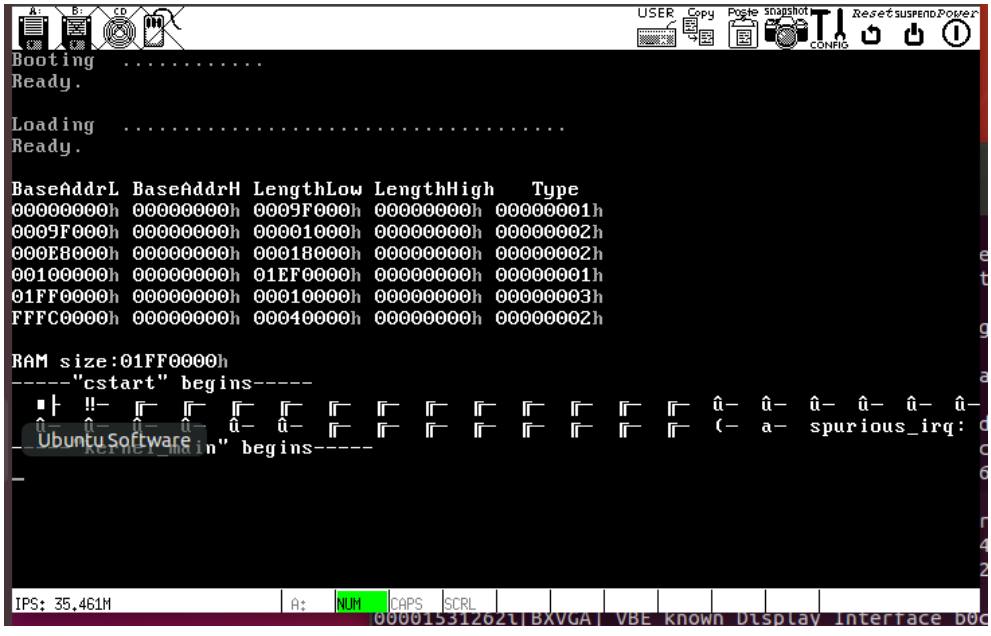
从gitee处下载到源码以后，要修改makefile文件才能用'make image'指令编译-链接成功，在终端 Terminal 输入:

```
bochs #启动虚拟机bochs
```

在终端Terminal 输入

```
c
```

退出bochs的调试模式；程序正式运行，截图如下：



至于怎样修改makefile：

- Bug1：

  __stack_chk_fail_local

  > **编译遇到“__stack_chk_fail_local”错误**
  >
  > refer—link：https://blog.csdn.net/duanbeibei/article/details/11890929
  >
  > 若在ubuntu上编译代码遇到"__stack_chk_fail_local"错误时，在makefile CFLAGS中加入"-fno-stack-protector"

- Fix1：我的虚拟机是64位的，从gitee上下载下来的是32位的源程序，要在makefile中指明用32位的方式编译和链接；

  > 修改 makefile文件：

```
# Programs, flags, etc.
ASM             = nasm
DASM            = ndisasm
CC              = gcc
LD              = ld
ASMBFLAGS       = -I boot/include/
ASMKFLAGS       = -I include/ -f elf32
CFLAGS          = -I include/ -c -fno-builtin -m32 -fno-stack-protector
LDFLAGS         = -s -Ttext $(ENTRYPOINT) -m elf_i386
DASMFLAGS       = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)
```

- Fix2：修改makefile中的挂载目录

```
# 注意这里我用: use '/home/fyl/HW/3/aMount' to replace '/mnt/floppy/'
buildimg :
        dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
        sudo mount -o loop a.img /mnt/floppy/
        sudo cp -fv boot/loader.bin /mnt/floppy/
        sudo cp -fv kernel.bin /mnt/floppy
        sudo umount /mnt/floppy
```

- Feat1：新增对make run的支持；

  其实就是加一个run指令的解释：

  因为源码编译后，要输入bochs启动虚拟机；所以“run”指令其实就是在make image后，在终端继续输入 “bochs”；

```
#todo new feature make-run
run :
        make image
        bochs
```

4、确定任务

经过运行样例以后，我发现任务实际上已经完成了大半……，只剩下实现 Tab键、20秒清屏、按ESC进行查找、Ctrl+Z撤回；感谢作者和助教！QAQ

> ### 1.1 功能要求
>
> **基本功能**
>
> 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. 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. 支持空格键和 Tab 键（4 个空格，可以被统一的删除）
> 6. 每隔 20 秒左右，清空屏幕。输入的字符重新从屏幕左上角开始显示。
> 7. 要求有光标显示，闪烁与否均可，但一定要跟随输入字符的位置变化。
> 8. 不要求支持屏幕滚动翻页，但输入字符数不应有上限。
> 9. 不要求支持方向键移动光标。
>
> **查找功能**

完成任务难度：

Tab<清屏<查找

5、阅读源码和《orange's》

先阅读《orange's》再读源码吧；

书P242——P252

7.1.1 keyboard.

keyboard.c: 键盘输入相关函数

keyboard_handler（）：键盘中断处理

init_keyboard()方法：打开键盘中断

7.1.3 键盘敲击过程

键盘编码器8048及兼容芯片——键盘控制器8042——中断处理芯片8259A

敲击键盘=输入（动作，内容）；内容就是按了哪个键；动作有3类：按下、按住、放开；

按下、放开都会产生扫描码；Scan Code；按下是make code，放开是break code；

扫描码一共有三套，set1 set2 set3；其中1是过去的键盘；3很少使用；主要是 Scan Code Set2；

键盘控制器8042的4个寄存器

输出缓冲、输入缓冲、状态寄存器、控制寄存器；

只有缓冲区被清空，才能写入和读入更多扫描码；

8048检测到动作，传递扫描码给8042——8042转换成Scan Code set1，然后存到缓冲区，告诉8259A产生了中断。

7.1.4 扫描码与字符的映射关系

keymap.h里

7.1.5 键盘输入缓冲区

keyboard.h里类似栈的数据结构s_kb；

s_kb的初始化交给init_keyboard()方法；

init_clock():时钟中断的设定和开启；

在kernel_main()中可以调用init_keyboard()和init_clock()完成键盘和时钟相关方法和中断机制，增强可修改性和代码简洁度。

7.1.6 新加任务处理键盘操作

终端任务；

阅读keyboard_read()的代码；

在这里，作者写了一个将输入缓冲区s_kb的数据不断读出的程序。

7.1.7 解析扫描码

7.1.7.1 解析字母数字

继续丰富keyboard_read()，在上一步的基础上，加入了if-else嵌套的用于解析的过程（也就是将扫描码和字符用于映射的过程）

但只能解析简单的扫描码，如字母数字；

7.1.7.2 完善shift ctrl alt5

增加了6个全局变量，用以标识左边右边各3个的shift、alt、ctrl；

增加了变量make，用以标识是make code还是 break code；

遇到复杂扫描码时（即遇到多字节扫描码）时，前面的控制用的字节用来设置全局变量，后面的配合控制用的变量进行打印和输出。

换句话说，一个完整的操作被分成了多次keyboard.read()

7.1.7.3 处理所有按键；

解决两个问题：

超过3字符的扫描码；

F1\F2等功能键会被打印；

解决第一个问题：

抽离出get_byte_from_kbuf()方法，便于通过一次keyboard_read()读完一个相对完整的扫描码。

解决第二个问题：

将keyboard.read()处理扫描码的功能剥离出来，交给in_process()处理。

通过变量key保存shift、alt、ctrl的状态信息。

作者通过给不可打印的特殊字符加一个FLAG_EXT的偏移量，再利用位与运算，以更好的在判断中识别可打印还是不可打印字符。

7.2 显示器 & 7.3 TTY终端和Console控制台

5.1 Feat1: 修改Tab键，让Tab可打印；

通过读书，我们知道了要让Tab有用，主要修改keyboard.c tty.c console.c和 .h 文件，主要是修改对Tab的解析。

直接看源码。

总结IO流程

> 流程：
>
> 1.main是程序的总入口
>
> main通过tty.c的函数task_tty()来启动IO模块
>
> 2.task_tty()会初始化键盘相关、终端相关，然后以轮询的方式监控各个终端、让各个终端执行tty_do_read、tty_do_write
>
> 3.tty_do_read()
>
> tty_do_read实际是做一个选中终端的判断，他踢出当前没有工作的终端，然后让keyboard_read()为当前选中的中断服务。
>
> 4.keyboard_read()
>
> keyboard_read()从键盘读取一次或多次键盘动作，形成有意义的扫描符串，将扫描符串通过keymap.h映射进行一轮转换以后，将两个字节、一个字符的信息合成到一个32位的变量key里，并将key传给tty.c里的in_process();
>
> 5.in_process()
>
> in_process里收到1个key；

> 我们可以首先判断这个字符是不是需要特殊处理的，如果可以直接打印，那就用put_key()处理
>
> 如果需要特殊处理，那就规定好以后，再传给put_key();
>
> **注意，我们的TAB特殊处理可以在这里新增！我们可以将\t'作为TAB的代表哦！**
>
> 6.put_key()
>
> put_key收到key以后，将之读入到终端的数据结构TTY 里作暂存。
>
> 7.tty_do_write()
>
> 接下来我们会看task_tty()，会发现控制权转交给了tty_do_write();
>
> tty_do_write()干的事也很简单，他并不自己处理输出，他将输出缓冲区（其实就是类栈结构TTY）中的元素一个一个取出，交给out_char()去输出;
>
> 8.out_char()
>
> out_char有1个指针p_vmem指向当前显示位置的地址。
>
> 然后是1个switch+1个while（）+1个flush（）
>
> 这个后面的while和flush的作用是让屏幕能正确显示光标所在。
>
> out_char()实际输出字符到屏幕上，靠的是switch语句，switch里又有一堆的特判，
>
> **显然，这里又是可以将TAB进行特判的时候，我们用字符'\t'和TAB建立了一一映射关系，这里无非就是遇到'\t'，输出4个空格罢了。**
>
> 9.out_char事实上也没有完成输出，但是就完成TAB功能而言，明显我们已经不再需要关注之后的了

```
./configure --with-x11 --enable-debugger --enable-disasm --enable-x86-64 --
enable-debugger-gui --enable-x86-debugger
```

2处更改实现

序号	文件	函数/宏/变量
1	tty.c	in_process()
2	console.c	out_char()的switch-case。新增case \t； 修改case \b 修改删除的情况，多一次特判

> 第1处：tty.c 的 in_process()

```
/*======================================================================*
                in_process
 *======================================================================*/
PUBLIC void in_process(TTY* p_tty, u32 key)
{
        char output[2] = {'\0', '\0'};

        if (!(key & FLAG_EXT)) {
        put_key(p_tty, key);
        }
        else {
            int raw_code = key & MASK_RAW;
            switch(raw_code) {

            //TODO New:新增对 TAB的特殊处理
            case TAB:
                put_key(p_tty, '\t');
                break;
            //End New
```

第2处:

```
switch(ch) {

/* TODO New
*  \t输出4个0x00,0x00不能显示,看上去就是空格,注意书上P265页图7.11;
*  这个不显示的字符是一个标识
*/
case '\t':
    if (p_con->cursor  <
        p_con->original_addr + p_con->v_mem_limit - 4) {
        for(int i = 0;i < 4 ;++i){
            //0x00不能显示,看上去就是空格
            *p_vmem++ = 0x00;
            *p_vmem++ = DEFAULT_CHAR_COLOR;
            p_con->cursor++;
        }
    }
    break;
//End New
```

```
case '\b':
    //TODO New
    //判断是不是遇上了Tab类的空格,是的话连续删除4次
    if ( (*(p_vmem - 2) == 0x00) &&
        (p_con->cursor >= p_con->original_addr + 4) ) {
        for (int i = 0; i < 4; i++) {
            p_con->cursor--;
            *(p_vmem - 2
            *(p_vmem - 1
        }
    }

    //下面加了一个else
    else if (p_con->cursor > p_con->original_addr) {
        p_con->cursor--;
        *(p_vmem-2) = ' ';
        *(p_vmem-1) = DEFAULT_CHAR_COLOR;
    }
    //End New
    break;
```

(字段) unsigned int s_console::cursor
当前光标位置
联机搜索

5.2 Feat2: 清屏

每隔 20 秒清空屏幕。**输入的字符**重新**从屏幕左上角**开始显示。

分析：

每隔20秒清屏——它是一个周期性任务，执行完以后delay(20s)；我们在main.c 里可以看到3个周期性任务，改一改其中一个的内容就行了

清空字符——就是将显存的内容覆盖为空格或者0x00;

实现

序号	文件	函数/宏/变量
1	main.c	TestA() 修改周期，调用clear()
2	tty.c	新增tty_clear()，它负责找到当前选中的控制台，然后调用clear_console()
3	console.c	新增clear_console(),它负责清除当前控制台对应的显存内容

> 第1处：

```c
/*============================================================================*
                                TestA
 *============================================================================*/
void TestA()
{
    int i = 0;
    while (1) {
        /*
        * TODO New
        * 在这里实现清屏——输入重新显示的功能;
        * 注意milli_delay(int mili_second)形参的单位是 0.1 毫秒,1s = 1000ms;
        */
        milli_delay(20 * 1000 * 10);
        tty_clear();
        // New End
    }
}
```

第2处:

```c
/* TODO New function:
 *============================================================================*
                    tty_clear()
 *============================================================================*
 */
PUBLIC void tty_clear()
{
    TTY* p_tty;

    /* 通过for循环,找到当前选中的控制台 */
    for (p_tty = TTY_FIRST; p_tty < TTY_END; p_tty++) {
        if (is_current_console(p_tty->p_console)) {
            break;
        }
    }

    clear_console(p_tty->p_console);

}
//End New
```

第3处:

```
/*TODO New function :
*========================================================================*
                            clear_console
*========================================================================*
* 清除当前console的屏幕,然后重新显示所有输入
*/
PUBLIC void clear_console(CONSOLE* p_con)
{
    //cursor_pos: 未清空前的游标地址
    int cursor_pos = p_con->cursor;

    /* current_start_addr 只清除当前屏幕; original_addr 清空显存所有内容;  */

    p_con->cursor = p_con->original_addr; /* p_con->cursor = p_con->current_start_addr; */
    for (u8* p_vmem = (u8*)(V_MEM_BASE + p_con->cursor * 2);
        p_con->cursor < cursor_pos;
        p_con->cursor++, flush(p_con) ) {
        *p_vmem++ = ' ';
        *p_vmem++ = DEFAULT_CHAR_COLOR;
    }

    p_con->cursor = p_con->original_addr; /* p_con->cursor = p_con->current_start_addr; */

    flush(p_con);
}
//End New
```

5.3 Feat3: 查找功能;

查找功能阶段划分:

阶段1: √

读到ESC的make code，进入查找模式，此外不清空屏幕;

阶段2: √

输入关键字，将输入的关键字以红色显示;

阶段3: 部分实现

3.1: 按回车后，确认匹配格式，根据格式找到所有匹配文本；单个的时候有问题；Tab不行;

3.2: 将匹配文本用红色显示 ×

3.3: 屏蔽除ESC外任何输入 √

动作: 按回车 匹配 红色显示 屏蔽输入

阶段4: √

4.1: 解除屏蔽，恢复原来模式: 不行，启动一次以后就一直不能输入了; √

4.2: 文本恢复白色; √

4.3: 删除关键字，光标回退; √

```
//TODO New
//ESC 查找模式匹配到的字串颜色
#define ESC_COLOR 0x04  /*  黑底红字 */
//End New
```