

武汉大学国家网络安全学院教学实验报告

课程名称	操作系统设计与实践	实验日期	2024.11.11
实验名称	I/O 子系统	实验周次	第九周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

Table 1

* 一、实验目的及实验内容

(本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析)

i

(实验目的、内容及相关的理论知识)

1.1 实验目的

- 1 了解键盘输入的基本原理。
- 2 了解显示器输出的基本原理。
- 3 了解 TTY 终端的概念。
- 4 了解 I/O 相关系统调用的实现及其对应功能

1.2 实验内容

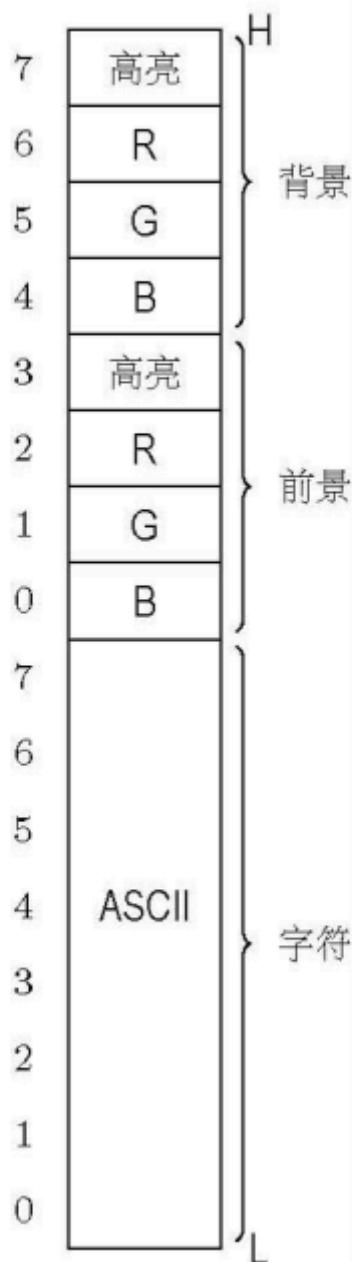
- ① 验证键盘扫描码与键盘中断的过程。
- ② 分析显示器的显示操控，以及输出方式。
- ③ 验证分析 tty 的处理过程，以及如何支持进程对 tty 的读写。
- ④ 结合 printf 的实现代码，理解 I/O 系统调用的实现机理

1.3 原理分析

- 键盘的敲击过程：在键盘中存在一枚叫做键盘编码器 Keyboard Encoder 的芯片，用于监视键盘的输入并把适当的数据传送给计算机。另外在计算机主板上，还有一个键盘控制器 Keyboard Controller，用来接收和解码来自键盘的数据并与 8259A 以及软件等进行通信。

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。当 8048 检测到一个键的动作后，会把相应的扫描码发送给 8042。8042 会把它转换成相应的 Scan code set 1 扫描码，并将其放置在输入缓冲区中，然后 8042 告诉 8259A 产生中断 IRQ1。如果此时键盘又有新的键被按下，8042 将不再接收，一直到缓冲区被清空，8042 才会收到更多的扫描码。

- 显示器的显示：在 Linux 下，开机看到的默认模式是 80×25 文本模式。在这种模式下显存大小为 32KB，占用的范围为 0xB8000 - 0xBFFFF。每 2 字节代表一个字符，其中低字节表示字符的 ASCII 码，高字节表示字符的属性。一个屏幕总共可以显示 25 行，每行 80 个字符。在默认情况下屏幕上每一个字符对应的 2 字节的定义如图所示：



为了让系统显示指定位置的内容，只需通过端口操作设置相应的寄存器。本实验假定系统使用的是 VGA 以上的视频子系统并假定不使用单色模式。

- **TTY:** 在 Linux 系统中，TTY（Teletypewriter）是一个代表终端设备的概念。TTY 是 Linux 操作系统中的一个重要部分，它允许用户与系统进行文本模式的交互。当按下 Alt+F1、Alt+F2、Alt+F3 等组合键时会切换到不同的屏幕。在这些不同的屏幕中可以分别有不同的输入和输出相互之间并不受到彼此影响。在某个终端中如果键入命令 **tty** 执行的结果将是当前的终端号。

✿ 二、实验环境及实验步骤

- ① (本次实验所使用的器件、仪器设备等的情况；具体实验步骤)

2.1 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- NASM 2.14.02
- Bochs 2.7
- Visual Studio Code (VSCode)

2.2 实验步骤

① 验证键盘扫描码与键盘中断：

- 验证键盘中断：编写键盘中断处理程序 `keyboard_handler`，实现按键触发中断时打印 `*`，初始化键盘中断处理程序，注册到中断向量表，开启键盘中断。
- 验证扫描码：`keyboard_handler` 中通过端口 `0x60` 读取扫描码并打印出来
- 设置键盘输入缓冲区：定义循环缓冲区 `kb_in`，存储扫描码，避免按键输入丢失。
- 解析扫描码：判断 `MakeCode` 和 `BreakCode`，解析特殊按键（`Pause/Break`、`PrintScreen`）。利用 `keymap` 映射扫描码到具体字符，处理组合键（`Shift`、`Ctrl`、`Alt`）。

② TTY 处理与控制台操作：

- TTY 初始化与处理框架：在 `TTY` 结构中定义缓冲区和控制台指针，初始化 `TTY` 和对应 `CONSOLE`。在 `task_tty` 中轮询各 TTY，执行读写操作。

- 读写操作的实现：实现 `tty_do_read`，从键盘缓冲区读取扫描码，调用 `in_process` 处理输入。实现 `tty_do_write`，从 TTY 缓冲区取字符，通过 `out_char` 输出到控制台。
- 控制台切换与光标操作：实现 `select_console`，通过 `alt + Fn` 切换控制台。使用 VGA 寄存器控制光标位置、屏幕滚动和显示内容起始地址。

③ `printf` 实现与 I/O 系统调用：

- 实现基本 `printf` 功能：定义 `printf`，使用变长参数解析格式化字符串，调用 `vsprintf` 将参数格式化到缓冲区，使用系统调用 `write` 输出缓冲区内容。

④ 实验拓展与优化（动手做）：

- 个性化 TTY：在 `TTY` 结构中增加标志 `is_special`，设置特定 TTY 启用个性功能。自由移动光标（上下左右方向键）、切换显示颜色（**F1** 键）、输出字符串图案（**F2** 键）。
- 扩展 `printf`：在 `vsprintf` 中增加对 `%s` 的支持，进一步拓展 `%u`、`%p` 等格式。分析 `printf` 格式化字符串漏洞（如 `%n` 的任意地址写入），提出输入验证与边界检查等安全措施。

* 三、实验过程分析



（详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等）

3.1 实验过程

3.1.1 验证键盘扫描码与键盘中断的过程

当我们在敲击键盘时会产生两个方面的含义：一个是敲击键盘的动作，另一个是敲击键盘的内容。敲击键盘的动作可以分为三种：按下按键、保持按住按键的状态和放开按键；敲击键盘的内容则是区分出按的是键盘上的哪一个键，键盘上的每一个按键都有自己对应的标识符。

对于敲击键盘的动作，在敲击键盘会产生一个编码，这个编码被称作扫描码（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。

敲击键盘的动作	产生的扫描码类型	产生扫描码的个数
按下	Make Code	一个
保持按下	Make Code	持续产生 n 个
弹起	Break Code	一个

Table 2

每一个按键都对应这一个扫描码，目前通用的扫描码总共有三套，分别是 Scan code set 1、Scan code set 2 和 Scan code set 3。Scan code set 1 是早期的 XT 键盘使用的，现在的键盘默认都支持 Scan code set 2，而 Scan code set 3 很少使用。但现在键盘默认支持的 Scan code set 2 在传给 8042 键盘控制器时也会自动转换为 Scan code set 1 的扫描码。

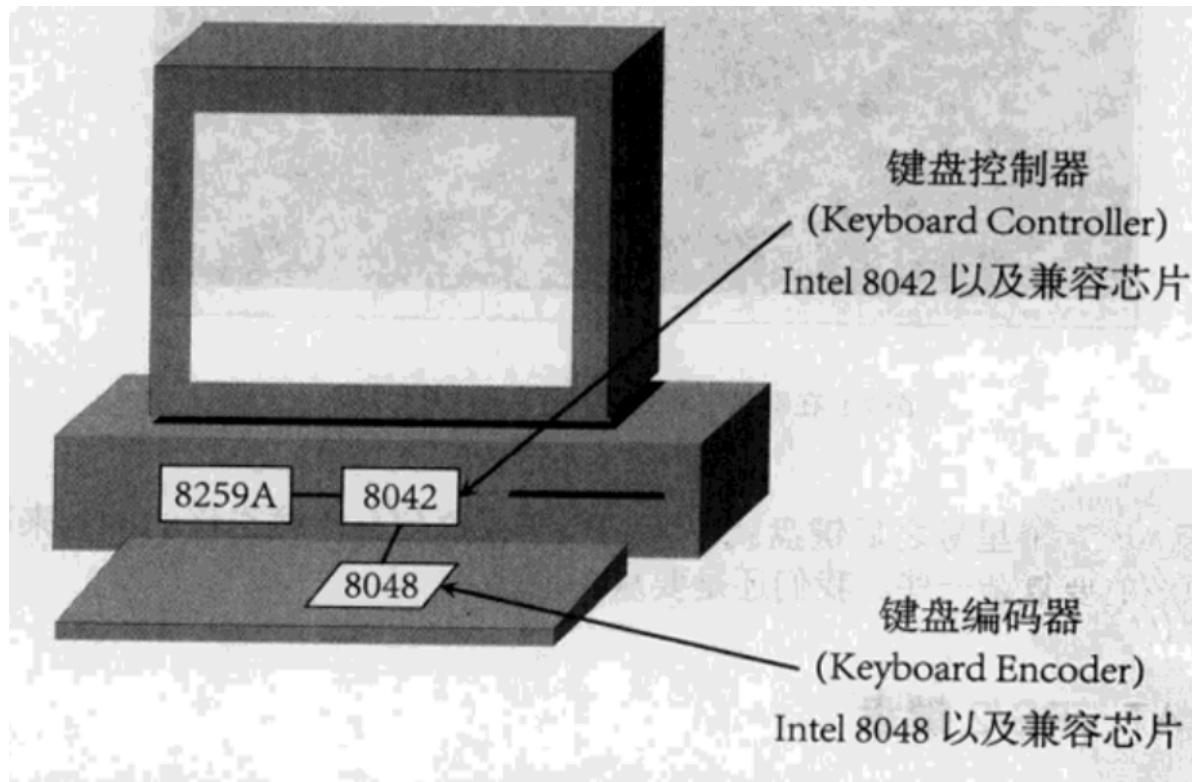


Figure 1

所以按键并弹起一个按键的全过程是：在键盘上按下一个按键由 8048 键盘编码器产生一个 Make Code 类型的按键扫描码，传送给 8042 键盘控制器，8042 将接收到的 Scan code set 2 中的扫描码转换为 Scan code set 1 扫描码的类型再传给 8259A 产生一个键盘中断。

表 7.1 8042 的寄存器				
寄存器名称	寄存器大小	端口	R/W	用法
输出缓冲区	1 BYTE	0x60	Read	读输出缓冲区
输入缓冲区	1 BYTE	0x60	Write	写输入缓冲区
状态寄存器	1 BYTE	0x64	Read	读状态寄存器
控制寄存器	1 BYTE	0x64	Write	发送命令

Figure 2

在 8042 的寄存器中，输入缓冲区的端口号为 0x60，所以我们就可以从这个端口中读取出键盘传来的按键扫描码。

3.1.1.1 验证键盘中断

接下来用代码实现一个键盘中断处理程序。先创建一个 keyboard.c 文件，在文件中实现中断处理程序 keyboard_handler，这个处理程序的作用触发键盘中断时就在屏幕上显示一个“*”。

```
/*=====
=====
PUBLIC void keyboard_handler(int irq)
{
    disp_str("*");
    /* u8 scan_code = in_byte(0x60);
    disp_int(scan_code); */
}
```

Fence 1

在之前的代码中，我们定义的 1 号中断（键盘中断）处理程序是通用的 spurious_irq 程序，作用是将中断号显示在屏幕上，而且并没有打开 8259A 中的键盘中断，所以过去我们并没有在有键盘输入时看到键盘中断处理程序的运行。现在既然我们已经定义好了我们自己的键盘中断处理程序，就不必使用原来的通用处理程序了，在 keyboard.c 文件中定义出初始化键盘中断处理程序，也就是将自己实现的中断处理程序放到中断向量表中，并且开启键盘中断。

```

/*=====
=====
    init_keyboard
=====

PUBLIC void init_keyboard()
{
    put_irq_handler(KEYBOARD_IRQ, keyboard_handler); /*设定键盘中断
处理程序*/
    enable_irq(KEYBOARD_IRQ);                      /*开键盘中断*/
}

```

Fence 2

在 main.c 中调用 init_keyboard 函数， 并修改 Makefile 就可以运行了。

```

PUBLIC int kernel_main()
{
...
    init_keyboard();
...
}

```

Fence 3

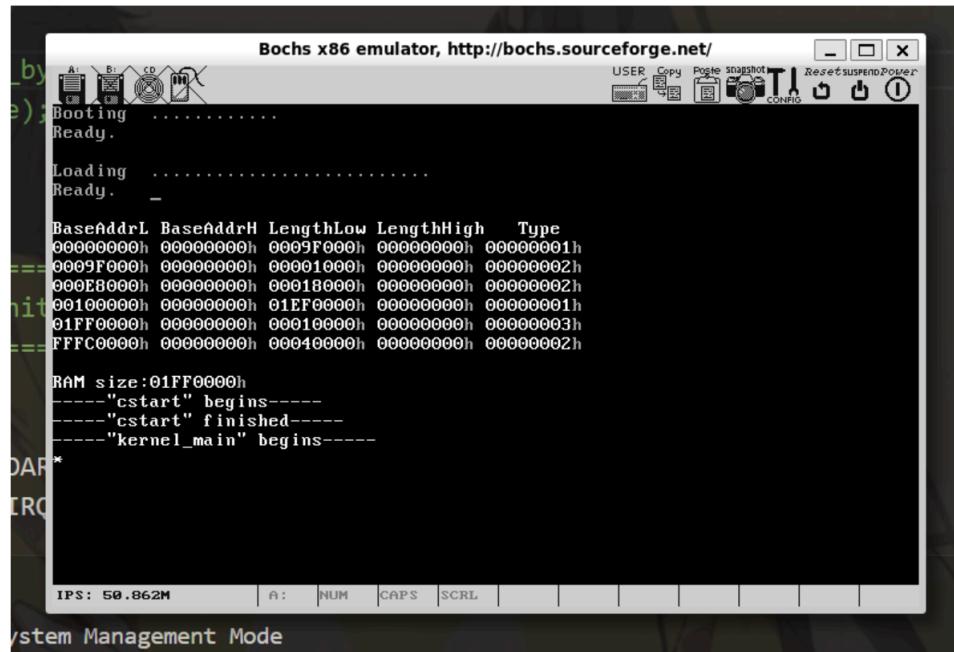


Figure 3

但可以看到屏幕上只显示了一个“*”号，这是因为 8048 检测到一个按键的动作之后，会把相应的扫描码发给 8042,8042 讲扫描码转换为 set 2 之后就把它放到缓冲区中，然后告诉 8259A 产生一个键盘中断，这时如果有新的按键被按下 8042 也不会再接收，而是等到缓冲区数据被清空之后才会继续接收下一个扫描码。所以在屏幕上就会出现只接受到一个扫描码显示出来的“*”号，而“卡住”的现象。

3.1.1.2 验证键盘扫描码

现在我们将缓冲区中的数据读取出来，就可以实现记录每一次按键事件的功能了。这个处理程序的作用就是从 0x60 端口中读取出一个扫描码，然后把这个扫描码的内容直接输出到屏幕上。

```
/*=====
=====
===== keyboard_handler
=====
=====

PUBLIC void keyboard_handler(int irq)
{
    /* disp_str("*"); */
    u8 scan_code = in_byte(0x60);
    disp_int(scan_code);
}
```

Fence 4

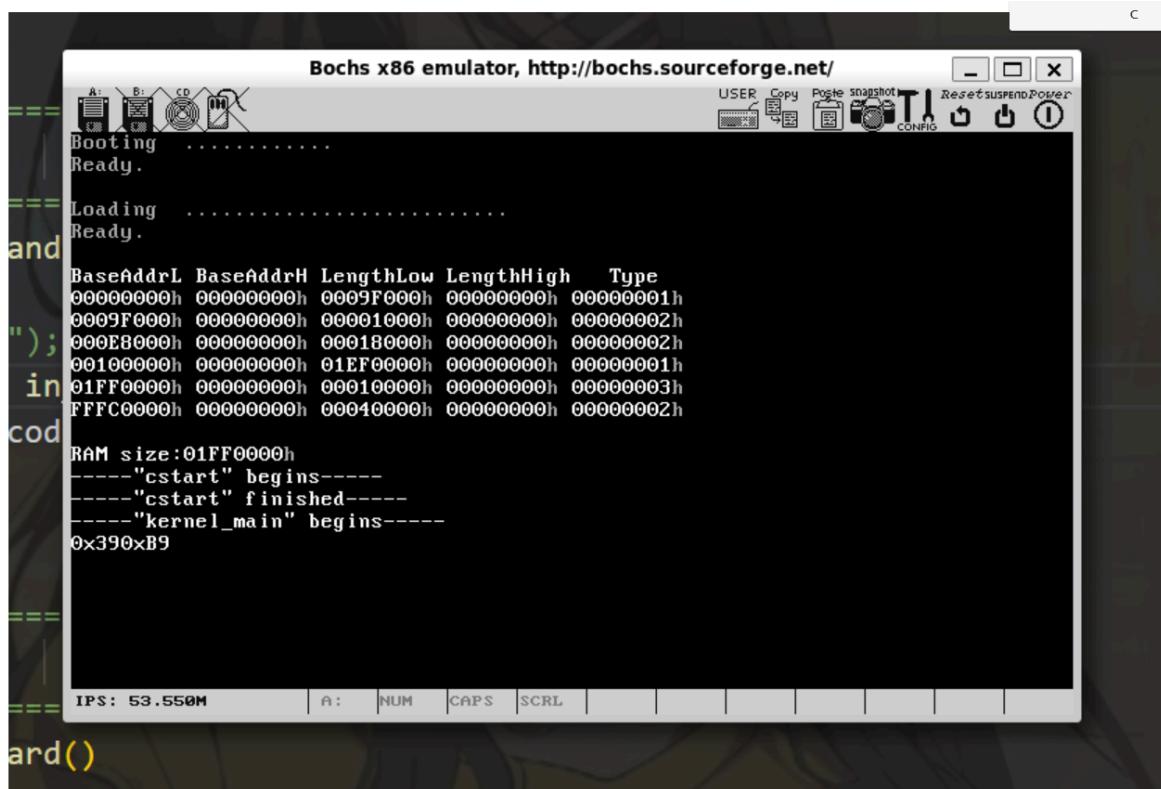


Figure 4

这里出现了两个扫描码，而且两个扫描码之间差一个 $0xB9 - 0x39 = 0x80$ ，这与我们预计的按下与弹起同一个按键会产生 Make Code 和 Break Code 的预期相同。

Key	Make	Break	Key	Make	Break
A	1E	9E	Tab	0F	8F
B	30	B0	C Lock	3A	BA
C	2E	AE	L Shift	2A	AA
D	20	A0	L Ctrl	1D	9D
E	12	92	L GUI	E0,5B	E0,DB
F	21	A1	L Alt	38	B8
G	22	A2	R Shift	36	B6
H	23	A3	R Ctrl	E0,1D	E0,9D
I	17	97	R GUI	E0,5C	E0,DC
J	24	A4	R Alt	E0,38	E0,B8
K	25	A5	APPS	E0,5D	E0,DD
L	26	A6	Enter	1C	9C
M	32	B2	Esc	1	81
N	31	B1	F1	3B	BB
O	18	98	F2	3C	BC
P	19	99	F3	3D	BD
Q	10	19	F4	3E	BE
R	13	93	F5	3F	BF
S	1F	9F	F6	40	C0
T	14	94	F7	41	C1
U	16	96	F8	42	C2
V	2F	AF	F9	43	C3
W	11	91	F10	44	C4
X	2D	AD	F11	57	D7
Y	15	95	F12	58	D8
Z	2C	AC	Print Screen	E0,2A	E0,B7
				E0,37	E0,AA

Figure 5

3.1.1.3 设置键盘输入缓冲区

8042 的缓冲区大小只有一个字节，但在输入“Shift+A”时会产生 4 次扫描码。分别是 0x2A 0X1E 0X9E 0XAA，这时我们就无法解析键盘的输入了，所以我们需要设置一个大一点的缓冲区，将接收到的信号先放到缓冲区中，再一次取出多个进行解析。

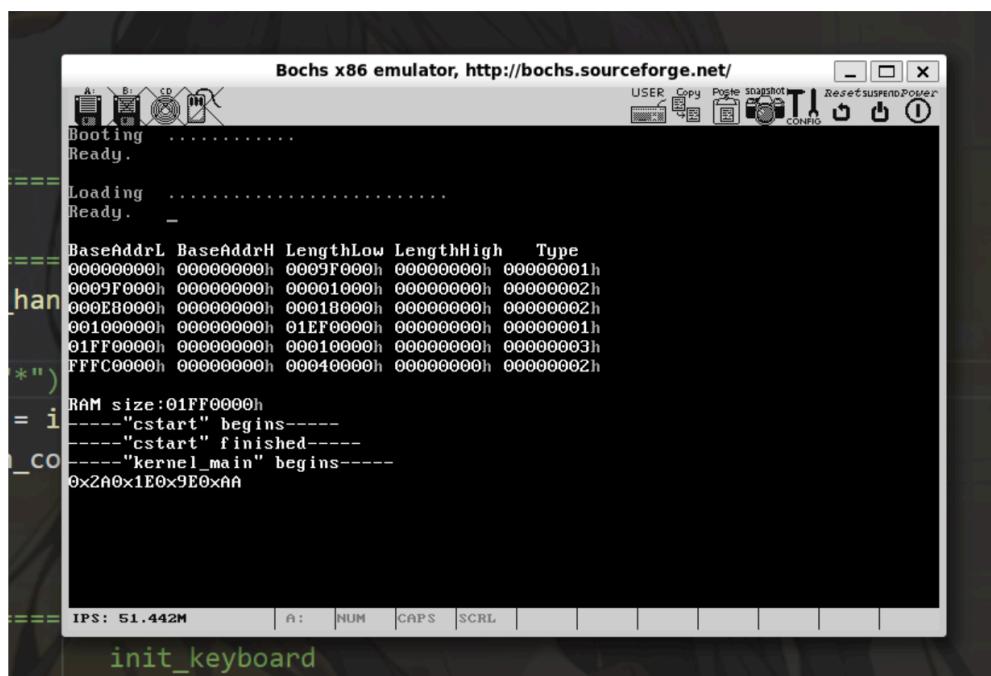


Figure 6

先在创建的 keyboard.h 中定义出一个键盘输入缓冲区的数据结构。

```
/*
 ****/
/*          Structure Definition
 */
/*
 ****/
/* Keyboard structure, 1 per console. */
typedef struct s_kb {
    char* p_head;           /* 指向缓冲区中下一个空闲位置 */
    char* p_tail;           /* 指向键盘任务应处理的字节 */
    int count;               /* 缓冲区中共有多少字节 */
    char buf[KB_IN_BYTES];   /* 缓冲区 */
}KB_INPUT;
```

Fence 5

可以看出这个定义出来的缓冲区是一个循环队列的形式，接下来将键盘中断处理程序改为将读取到的扫描码存入缓冲区。

```
PRIVATE KB_INPUT kb_in;

/*
=====
====*
        keyboard_handler
=====

=====
====*/
PUBLIC void keyboard_handler(int irq)
{
    u8 scan_code = in_byte(KB_DATA);

    if (kb_in.count < KB_IN_BYTES) {
        *(kb_in.p_head) = scan_code;
        kb_in.p_head++;
        if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_head = kb_in.buf;
        }
        kb_in.count++;
    }
}
```

Fence 6

然后还需要将 kb_in 初始化一下，这个过程放在 init_keyboard 中进行。

```

/*=====
=====
    init_keyboard
=====
=====*/
PUBLIC void init_keyboard()
{
    kb_in.count = 0;
    kb_in.p_head = kb_in.p_tail = kb_in.buf;

    put_irq_handler(KEYBOARD_IRQ, keyboard_handler); /*设定键盘中断
处理程序*/
    enable_irq(KEYBOARD_IRQ);                      /*开键盘中断*/
}

```

Fence 7

3.1.1.4 解析扫描码

先写一个读取缓冲区的程序，在这其中还涉及到了 disable_int 和 enable_int 的两个函数，这两个函数的功能就是分别集成了开、关中断。

```

/*=====
=====
    keyboard_read
=====
=====*/
PUBLIC void keyboard_read()
{
    u8 scan_code;

    if(kb_in.count > 0){
        disable_int();
        scan_code = *(kb_in.p_tail);
        kb_in.p_tail++;
        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }
        kb_in.count--;
        enable_int();

        disp_int(scan_code);
    }
}

```

```

;
=====

;
        void disable_int();
;

=====

=====
disable_int:
    cli
    ret

;

=====

;
        void enable_int();
;

=====

=====
enable_int:
    sti
    ret

```

Fence 8

3.1.1.4.1 让字符显示出来

想要显示字符其实要做的事情就是在读取到一个 Make Code 之后通过查找定义在 keymap.h 中的表来将它输出出来。

```

/*=====
=====
        keyboard_read
=====
=====

PUBLIC void keyboard_read()
{
    u8 scan_code;
    char output[2];
    int make; /* TRUE: make; FALSE: break. */

    memset(output, 0, 2);

    if(kb_in.count > 0){
        disable_int();
        scan_code = *(kb_in.p_tail);
    }
}
```

```

kb_in.p_tail++;
if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
    kb_in.p_tail = kb_in.buf;
}
kb_in.count--;
enable_int();

/* 下面开始解析扫描码 */
if (scan_code == 0xE1) {
    /* 暂时不做任何操作 */
}
else if (scan_code == 0xE0) {
    /* 暂时不做任何操作 */
}
else { /* 下面处理可打印字符 */

    /* 首先判断Make Code 还是 Break Code */
    make = (scan_code & FLAG_BREAK ? FALSE : TRUE);

    /* 如果是Make Code 就打印，是 Break Code 则不做处理 */
    if(make) {
        output[0] = keymap[(scan_code&0x7F)*MAP_COLS];
        disp_str(output);
    }
}
/* disp_int(scan_code); */
}
}

```

Fence 9

运行的效果如下图所示。

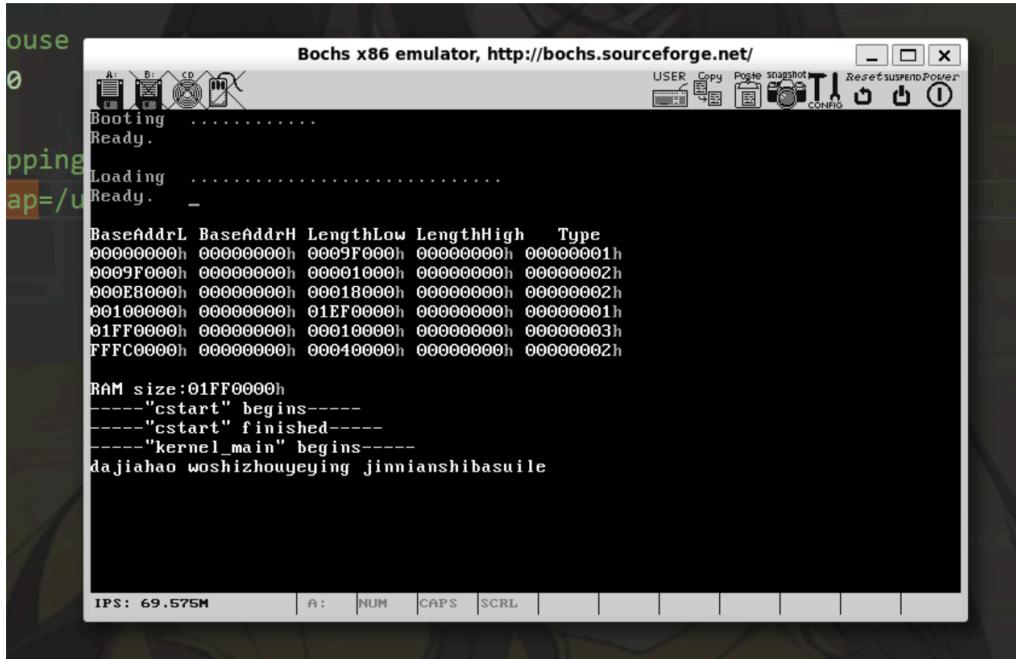


Figure 7

3.1.1.4.2 处理 Shift、Alt、Ctrl

先定义出检测这些按键的状态，因为在有的应用中左 Shift 和右 Shift 的功能是不一样的，所以在这里定义了 6 个变量来分别存储 Shift、Alt、Ctrl3 种按键各有左右 2 个的 6 种状态。

```

PRIVATE int code_with_E0 = 0;
PRIVATE int shift_l; /* l shift state */
PRIVATE int shift_r; /* r shift state */
PRIVATE int alt_l; /* l alt state */
PRIVATE int alt_r; /* r left state */
PRIVATE int ctrl_l; /* l ctrl state */
PRIVATE int ctrl_r; /* l ctrl state */
PRIVATE int caps_lock; /* Caps Lock */
PRIVATE int num_lock; /* Num Lock */
PRIVATE int scroll_lock; /* Scroll Lock */
PRIVATE int column;

```

Fence 10

当检测到按下左 Shift 键时，Shift_l 置为 TRUE，并未松开时又按下了“a”键，则 column 置为 1，表示读取 keymap [column] 中也就是第二列的值，对应的是大写字母的“A”。同时代码对以 0xE0 开头的扫描码进行了处理，这个字节意味着当前这个按键的扫描码不止一个字节，要联合着下一个字节的内容来解析按键类型。当检测到 0xE0 开头的扫描码时，code_with_E0 被置为 TRUE，会直接返回，退出这个程序，这样做的作用是使用下一个字节的内容来解析对应的按键，当下一个字节的内容来

时，code_with_E0 还为 TRUE，则 column 置为 2，key 变为 keymap [] 中的第二列的值。如果一个完整的操作还未结束，key 的值会被赋值为 0，等到下一次 keyboard_read() 执行时再处理。

```
/* 下面开始解析扫描码 */
if (scan_code == 0xE1) {
    /* 暂时不做任何操作 */
}
else if (scan_code == 0xE0) {
    code_with_E0 = 1;
}
else { /* 下面处理可打印字符 */

    /* 首先判断 Make Code 还是 Break Code */
    make = (scan_code & FLAG_BREAK ? 0 : 1);

    /* 先定位到 keymap 中的行 */
    keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];

    column = 0;
    if (shift_l || shift_r) {
        column = 1;
    }
    if (code_with_E0) {
        column = 2;
        code_with_E0 = 0;
    }

    key = keyrow[column];

    switch(key) {
    case SHIFT_L:
        shift_l = make;
        key = 0;
        break;
    case SHIFT_R:
        shift_r = make;
        key = 0;
        break;
    case CTRL_L:
        ctrl_l = make;
        key = 0;
        break;
    case CTRL_R:
        ctrl_r = make;
```

```
key = 0;
break;

case ALT_L:
    alt_l = make;
    key = 0;
    break;

case ALT_R:
    alt_l = make;
    key = 0;
    break;

default:
    if (!make) { /* 如果是 Break Code */
        key = 0; /* 忽略之 */
    }
    break;
}

/* 如果 Key 不为0说明是可打印字符，否则不做处理 */
if(key){
    output[0] = key;
    disp_str(output);
}
```

Fence 11

这时就可以打印出大写的字母了。

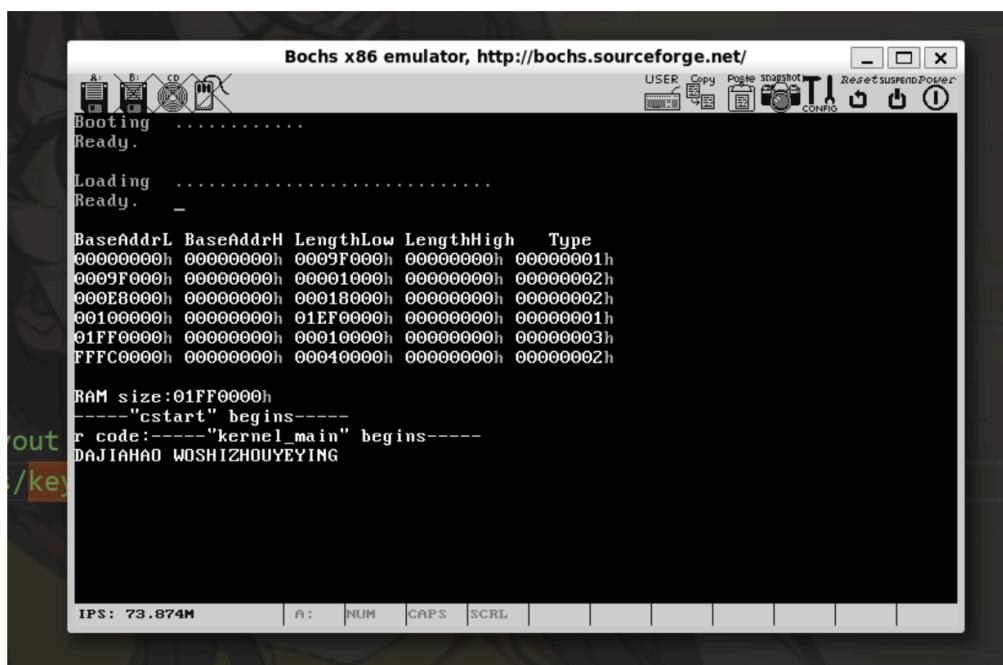


Figure 8

3.1.1.4.3 处理所有按键

目前的程序无法处理三个字节以上的扫描码，而且对于一些未定义的按键，例如“F1”这种功能键，还不能将其功能实现出来，而是打印一个奇怪的符号。接下来就将处理这些功能按键。

先将从缓冲区中取出一个字节的功能集成到单独的 `get_byte_from_kbuf` 中，这样就可以让每一次调用 `keyboard_read` 处理一个相对完整的过程，而不需要多次调用。然后单独处理 Pause 和 PrintScreen 两个按键。

```
PRIVATE u8 get_byte_from_kbuf();

/*
=====
*=*
          keyboard_read
=====
=*/
PUBLIC void keyboard_read()
{
    ...
    scan_code = get_byte_from_kbuf();
    ...

    if (scan_code == 0xE1) {
        int i;
        // 判断是不是Pause
        u8 pausebrk_scode[] = {0xE1, 0x1D, 0x45,
                               0xE1, 0x9D, 0xC5};
        int is_pausebreak = 1;
        for(i=1;i<6;i++){
            if (get_byte_from_kbuf() != pausebrk_scode[i]) {
                is_pausebreak = 0;
                break;
            }
        }
        if (is_pausebreak) {
            key = PAUSEBREAK;
        }
    }

    else if (scan_code == 0xE0) {
        scan_code = get_byte_from_kbuf();

        /* PrintScreen 被按下 */
        if (scan_code == 0x2A) {
            if (get_byte_from_kbuf() == 0xE0) {
                if (get_byte_from_kbuf() == 0x37) {

```

```

        key = PRINTSCREEN;
        make = 1;
    }
}
}

/* PrintScreen 被释放 */
if (scan_code == 0xB7) {
    if (get_byte_from_kbuf() == 0xE0) {
        if (get_byte_from_kbuf() == 0xAA) {
            key = PRINTSCREEN;
            make = 0;
        }
    }
}
/* 不是PrintScreen, 此时scan_code为0xE0紧跟的那个值. */
if (key == 0) {
    code_with_E0 = 1;
}
}

}

/*
=====
====*
get_byte_from_kbuf

=====
====*/
PRIVATE u8 get_byte_from_kbuf() /* 从键盘缓冲区中读取下一个字节 */
{
    u8 scan_code;

    while (kb_in.count < 0) {} /* 等待下一个字节到来 */

    disable_int();
    scan_code = *(kb_in.p_tail);
    kb_in.p_tail++;
    if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
        kb_in.p_tail = kb_in.buf;
    }
    kb_in.count--;
    enable_int();

    return scan_code;
}

```

上面的这份代码使用了单独的两个判断语句来对 PrintScreen 和 Pause 进行判断和操作还是显得过于臃肿，功能按键的具体功能实现不是在这个函数里进行的，在这里要实现的功能应该是识别出不同的按键，然后记录或输出这个按键。所以进一步将功能类按键进行封装。为了区分出不需要输出的按键，可以通过宏定义的方式区分出不需要输出的按键。

```
if (make) { /* 忽略 Break Code */
    key |= shift_l ? FLAG_SHIFT_L : 0;
    key |= shift_r ? FLAG_SHIFT_R : 0;
    key |= ctrl_l ? FLAG_CTRL_L : 0;
    key |= ctrl_r ? FLAG_CTRL_R : 0;
    key |= alt_l ? FLAG_ALT_L : 0;
    key |= alt_r ? FLAG_ALT_R : 0;

    in_process(key);
}

=====
====*
    in_process

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

    if (!(key & FLAG_EXT)) {
        output[0] = key & 0xFF;
        disp_str(output);
    }
}

/* Special keys */
#define ESC      (0x01 + FLAG_EXT) /* Esc      */
#define TAB      (0x02 + FLAG_EXT) /* Tab      */
#define ENTER    (0x03 + FLAG_EXT) /* Enter    */
#define BACKSPACE (0x04 + FLAG_EXT) /* BackSpace */

#define GUI_L     (0x05 + FLAG_EXT) /* L GUI    */
#define GUI_R     (0x06 + FLAG_EXT) /* R GUI    */
#define APPS      (0x07 + FLAG_EXT) /* APPS    */

/* Shift, Ctrl, Alt */
```

```

#define SHIFT_L      (0x08 + FLAG_EXT) /* L Shift */
#define SHIFT_R      (0x09 + FLAG_EXT) /* R Shift */
#define CTRL_L       (0x0A + FLAG_EXT) /* L Ctrl */
#define CTRL_R       (0x0B + FLAG_EXT) /* R Ctrl */
#define ALT_L        (0x0C + FLAG_EXT) /* L Alt */
#define ALT_R        (0x0D + FLAG_EXT) /* R Alt */

```

Fence 13

最后就可以正常地输出所有字符而不会出现乱码的现象了。

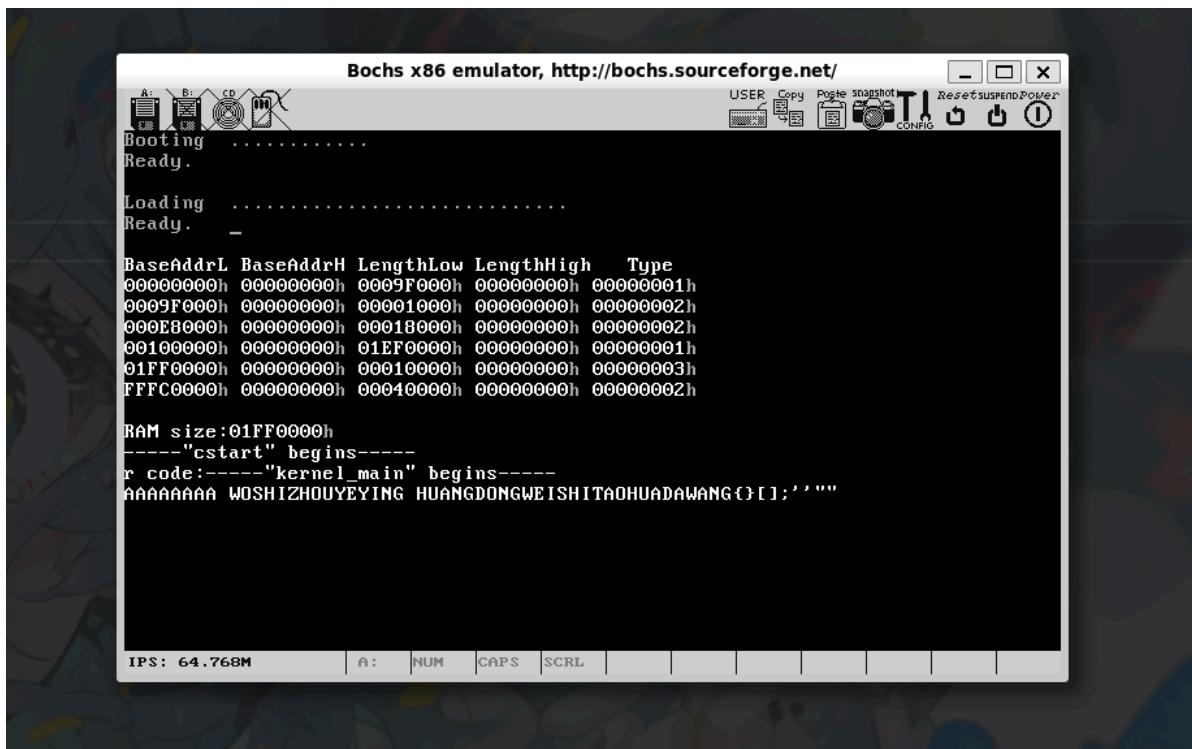


Figure 9

3.1.2 分析显示器的显示操控，以及输出方式

3.1.2.1 显存与显示模式

在 **80×25** 文本模式 下，显存地址从 **0xB8000** 开始，总大小为 32KB。每 2 字节表示一个字符：

- 低字节：ASCII 码，表示字符本身。
- 高字节：属性字节，表示字符的颜色、亮度和闪烁。

汇编语言示例：

```

mov ax, 0x0C41 ; 0x41 是 'A' 的 ASCII 码, 0x0C 表示红色字符。
mov [gs:edi], ax ; 将字符写入显存。

```

Fence 14

C 语言示例：

```
void display_char(char ch, char color, int position) {
    unsigned short *v_mem = (unsigned short *)0xB8000; // 显存地址
    v_mem[position] = (color << 8) | ch; // 写入字符和颜色
}
//参数 ch 是字符, color 是颜色, position 是显存中的位置。
display_char('A', 0x0C, 0); // 红色字符 'A'
```

Fence 15

一个屏幕可以显示 $80 \times 25 = 2000$ 个字符，因此占用的显存为 4000 字节（4KB），在 32KB 显存中，可以存储多个屏幕的数据。这种结构便于在多终端（TTY）系统中切换显示内容。

3.1.2.2 控制显示内容——使用 VGA 寄存器

VGA 系统有 6 组寄存器，各个寄存器读和写的端口不同。其中 **CRT (Cathode Ray Tube) Controller Registers** 在显示器的显示操控中扮演着重要角色，是操作系统的显示架构中的核心部分。

寄存器		读端口	写端口
General Registers	Miscellaneous Output Register	0x3CC	0x3C2
	Input Status Register 0	0x3C2	-
	Input Status Register 1	0x3DA	-
	Feature Control Register	0x3CA	0x3DA
	Video Subsystem Enable Register	0x3C3	
Sequencer Registers	Address Register	0x3C4	
	Data Registers	0x3C5	
CRT Controller Registers	Address Register	0x3D4	
	Data Registers	0x3D5	
Graphics Controller Registers	Address Register	0x3CE	
	Data Registers	0x3CF	
Attribute Controller Registers	Address Register	0x3C0	
	Data Registers	0x3C1	0x3C0
Video DAC Palette Registers	Write Address	0x3C8	
	Read Address	-	0x3C7
	DAC State	0x3C7	-
	Data	0x3C9	
	Pel Mask	0x3C6	-

Figure 10

CRT 控制器的寄存器分为两组：

- ① 地址寄存器（**CRTC Address Register**）：用于选择要操作的具体寄存器（类似数组的索引），位于 I/O 端口 **0x3D4**。
- ② 数据寄存器（**CRTC Data Register**）：用于读写寄存器的具体值，位于 I/O 端口 **0x3D5**。

每次操控寄存器需要通过向地址寄存器 **0x3D4** 写入寄存器的索引值，选择具体的寄存器。之后通过数据寄存器 **0x3D5** 写入或读取寄存器的值。

其中数据控制器支持多种功能寄存器，通过这些寄存器，可以灵活控制显示内容、光标位置及滚屏等功能。

常用寄存器的索引值如下：

索引值	寄存器名称	功能
0x0C	Start Address High	显存起始地址的高 8 位
0x0D	Start Address Low	显存起始地址的低 8 位
0x0E	Cursor Location High	光标位置的高 8 位
0x0F	Cursor Location Low	光标位置的低 8 位

Table 3

更多的寄存器名称及其索引如下表：

寄存器名称	索引	寄存器名称	索引
Horizontal Total Register	00h	Start Address Low Register	0Dh
End Horizontal Display Register	01h	Cursor Location High Register	0Eh
Start Horizontal Blanking Register	02h	Cursor Location Low Register	0Fh
End Horizontal Blanking Register	03h	Vertical Retrace Start Register	10h
Start Horizontal Retrace Register	04h	Vertical Retrace End Register	11h
End Horizontal Retrace Register	05h	Vertical Display End Register	12h
Vertical Total Register	06h	Offset Register	13h
Overflow Register	07h	Underline Location Register	14h
Preset Row Scan Register	08h	Start Vertical Blanking Register	15h
Maximum Scan Line Register	09h	End Vertical Blanking	16h
Cursor Start Register	0Ah	CRTC Mode Control Register	17h
Cursor End Register	0Bh	Line Compare Register	18h
Start Address High Register	0Ch		

Figure 11

以下分别以 设置光标位置 和 实现屏幕滚动 为示例说明，其中几个宏的定义在/h/include/const.h 文件

```

#define CRTC_ADDR_REG    0x3D4 /* CRT Controller Registers - Addr Register */
#define CRTC_DATA_REG    0x3D5 /* CRT Controller Registers - Data Register */
#define START_ADDR_H     0xC /* reg index of video mem start addr (MSB) */
#define START_ADDR_L     0xD /* reg index of video mem start addr (LSB) */
#define CURSOR_H         0xE /* reg index of cursor position (MSB) */
#define CURSOR_L         0xF /* reg index of cursor position (LSB) */
#define V_MEM_BASE       0xB8000 /* base of color video memory */
#define V_MEM_SIZE        0x8000 /* 32K: B8000H → BFFFFH */

```

Fence 16

① 设置光标位置：设置光标位置是通过修改光标位置寄存器实现的

- **Cursor Location High Register**（索引 0x0E）：光标位置的高 8 位
- **Cursor Location Low Register**（索引 0x0F）：光标位置的低 8 位

每次敲入一个新普通字符，更新光标位置在字符后。使用 `out_byte` 给地址寄存器分别写入光标位置寄存器索引号 0x0E、0x0F。再依次给两个光标位置寄存器写入当前光标位置。由于每个字符占用两个字节，所以 `disp_pos/2` 表示当前光标在屏幕上的字符位置。

```

disable_int(); // 禁止中断，确保原子操作

out_byte(CRTC_ADDR_REG, CURSOR_H);           // 设置光标高位
out_byte(CRTC_DATA_REG, (disp_pos/2 >> 8) & 0xFF);
out_byte(CRTC_ADDR_REG, CURSOR_L);           // 设置光标低位
out_byte(CRTC_DATA_REG, disp_pos/2 & 0xFF);

enable_int(); // 恢复中断

```

Fence 17

可以发现光标开始跟随字符，位于字符结尾处的下一个

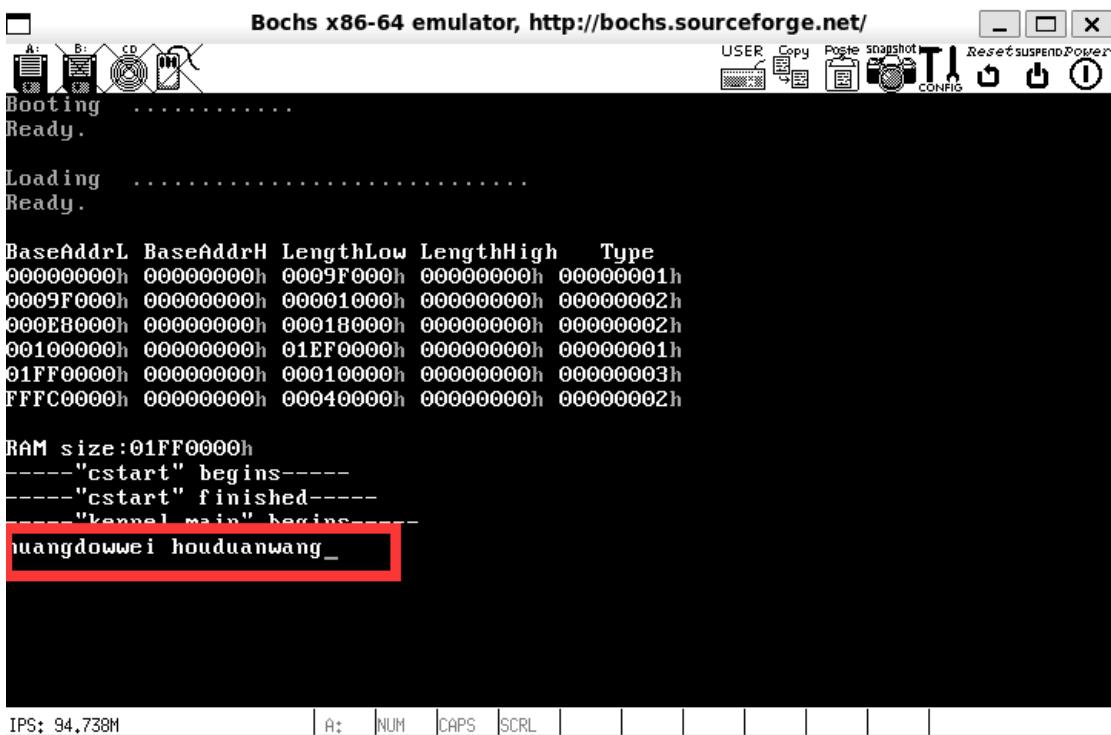


Figure 12

- ② 实现屏幕滚动：滚屏是通过修改 起始地址寄存器 实现的：

- **Start Address High Register** (索引 0x0C) : 起始地址的高 8 位。
- **Start Address Low Register** (索引 0x0D) : 起始地址的低 8 位。

每次输入 shift + 方向键，将会修改显示起始地址，显示器会从显存的新地址开始显示数据，从而实现滚动屏幕内容，而无需物理移动显存中的数据。

```

int raw_code = key & MASK_RAW;
switch(raw_code) {
    case UP:
        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
            disable_int();
            out_byte(CRTC_ADDR_REG, START_ADDR_H);
            out_byte(CRTC_DATA_REG, ((80*15) >> 8) & 0xFF);
            out_byte(CRTC_ADDR_REG, START_ADDR_L);
            out_byte(CRTC_DATA_REG, (80*15) & 0xFF);
            enable_int();
        }
        break;
    case DOWN:
        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
/* Shift+Down, do nothing */
        }
        break;
    default:
        break;
}

```

按下 Shift + ↑，可以看到屏幕向上滚动

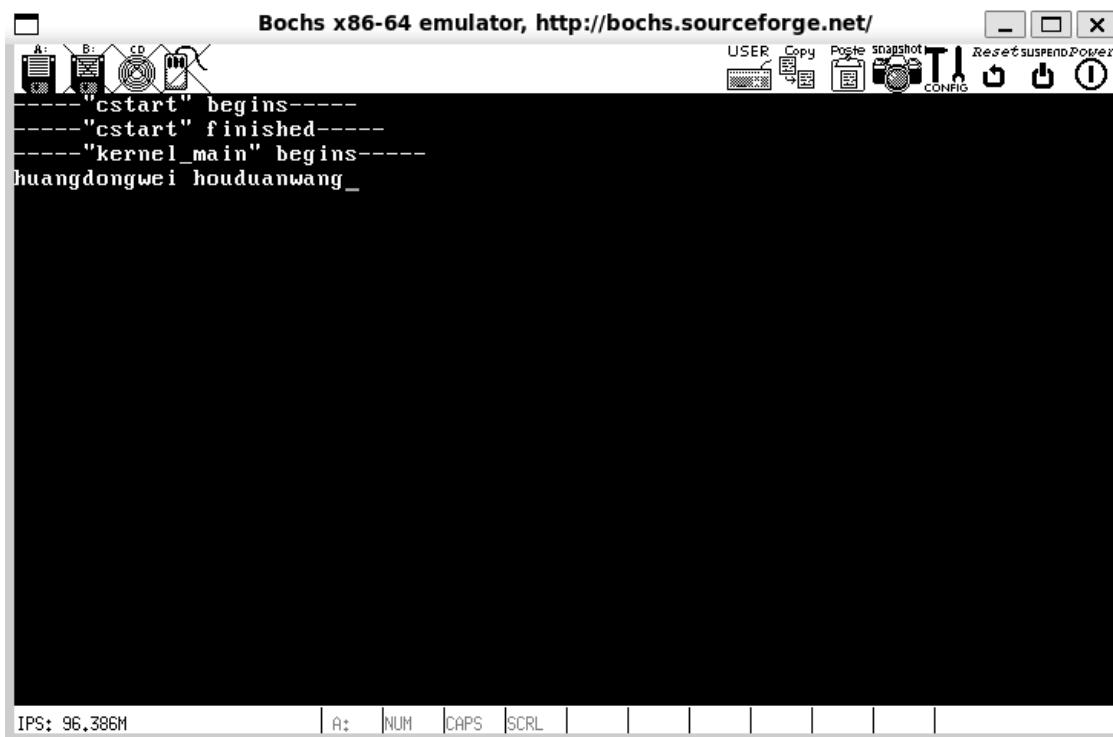
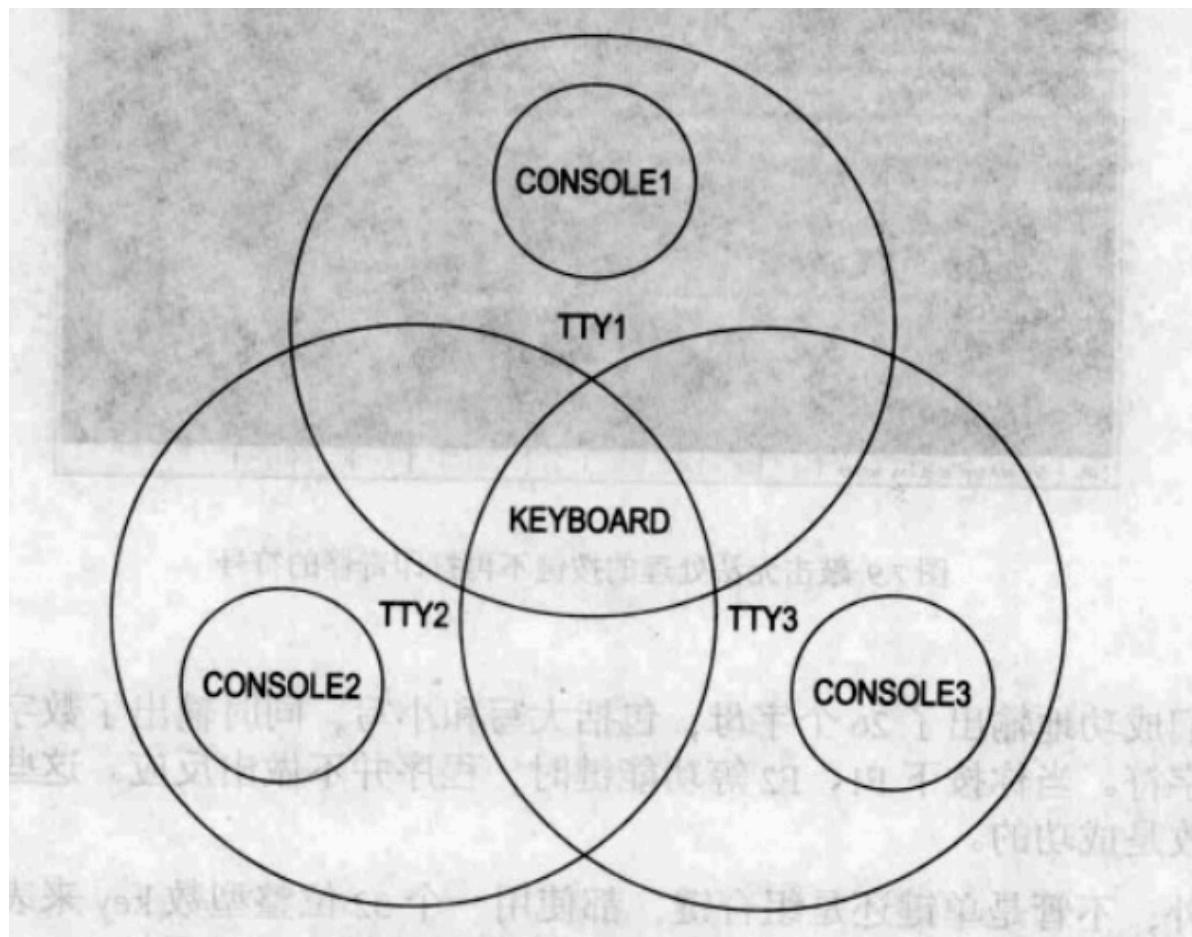


Figure 13

3.1.3 验证分析 tty 的处理过程，以及如何支持进程对 tty 的读写

3.1.3.1 TTY 处理的整体流程

TTY 的整体架构如下：



不同的 TTY 对应的输入设备是同一个键盘，但输出是在不同的屏幕上，即对应显存的不同位置。

切换 console 通过轮询的方式实现。在 TTY 任务中执行一个循环，这个循环将轮询每一个 TTY。处理它的事件包括从键盘缓冲区读取数据、显示字符等内容。

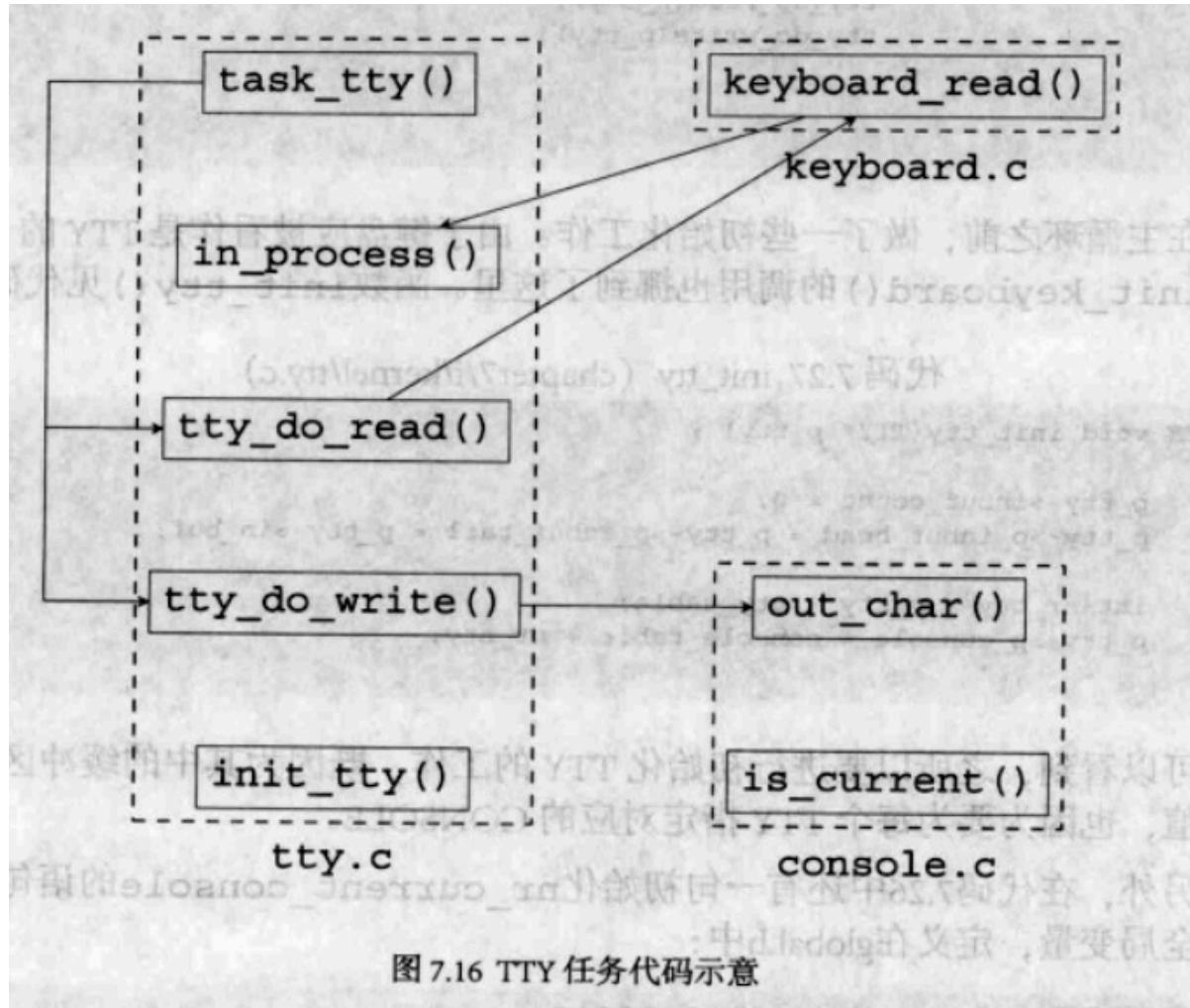
因此，轮询到每一个 TTY 时只需做两件事：

- 处理输入——查看是不是当前 TTY，如果是则从键盘缓冲区读取数据。
- 处理输出——如果有要显示的内容则显示它。



注意，当某个 TTY 对应的控制台是当前控制台时，它才可以读取键盘缓冲区。

TTY 的处理过程如下：



- ① 在主函数 `task_tty()` 中，通过循环来轮流处理每一个 TTY 的读和写操作；
- ② 读操作放入 `tty_do_read()` 函数中，它调用 `keyboard_read()`，
`keyboard_read()` 从键盘缓冲区得到数据后会调用 `in_process()` 将字符进行处理。
注意到为了实现读写分离，调用的 `in_process()` 不应该再直接回显字符，而
应该将回显的任务交给 TTY 来完成
- ③ 写操作放入 `tty_do_write()` 函数中，它会调用 `out_char()` 将字符写入指
定的 console。

TTY 和 console 的结构如下：

```
/* TTY */
typedef struct s_tty
{
    u32      in_buf[TTY_IN_BYTES];      /* TTY 输入缓冲区 */
    u32*     p_inbuf_head;            /* 指向缓冲区中下一个空闲位置 */
    u32*     p_inbuf_tail;           /* 指向键盘任务应处理的键值 */
    int      inbuf_count;             /* 缓冲区中已经填充了多少 */

    struct s_console *      p_console;
}TTY;

/* 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;
```

3.1.3.2 TTY 处理过程分析

- 主函数 `task_tty()` :

```
=====
task_tty
=====
PUBLIC void task_tty()
{
    TTY*      p_tty;

    init_keyboard();

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

打开键盘中断后，对所有 TTY 进行初始化，为缓冲区设置初值，设置对应的 console。

接着进入循环，对每个 TTY 进行轮询，其中，`nr_current_console` 记录当前的控制台是哪一个。只有当某个 TTY 对应的控制台是当前控制台时，它才可以读取键盘缓冲区。

- TTY 读操作函数 `tty_do_read()` :

```
PRIVATE void tty_do_read(TTY* p_tty)
{
    if (is_current_console(p_tty->p_console)) {
        keyboard_read(p_tty);
    }
}
```

TTY 首先对控制台进行判断，只有对应的控制台是当前控制台时，它才可以读取键盘缓冲区。当前控制台由 `nr_current_console` 记录，`is_current_console()` 只需将其与调用函数的 TTY 进行比较即可。

```
PUBLIC int is_current_console(CONSOLE* p_con)
{
    return (p_con == &console_table[nr_current_console]);
}
```

接着，调用 `keyboard_read()` 读取扫描码，结果进入 `in_process` 进行进一步处理。注意到为了实现读写分离，调用的 `in_process()` 不应该再直接回显字符，而应该将回显的任务交给 TTY 来完成。因此，在读操作过程中还需传入 `p_tty` 参数，将输出的字符写入 TTY 的缓冲区，而非直接打印。

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

    if (!(key & FLAG_EXT)) {
        if (p_tty->inbuf_count < TTY_IN_BYTES) {
            *(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++;
        }
    } else {
        int raw_code = key & MASK_RAW;
        switch(raw_code) {
        case UP:
            if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                disable_int();
                out_byte(CRTC_ADDR_REG, START_ADDR_H);
                out_byte(CRTC_DATA_REG, ((80*15) >> 8) & 0xFF);
                out_byte(CRTC_ADDR_REG, START_ADDR_L);
                out_byte(CRTC_DATA_REG, (80*15) & 0xFF);
                enable_int();
            }
            break;
        case DOWN:
            if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                /* Shift+Down, do nothing */
            }
            break;
        default:
            break;
        }
    }
}
```

- TTY 写操作函数 `tty_do_write()` :

```
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);
    }
}
```

它从 TTY 缓冲区中取出值，然后用 `out_char()` 显示在 console 中。其中，
`out_char()` 代码如下：

```
PUBLIC void out_char(CONSOLE* p_con, char ch)
{
    u8* p_vmem = (u8*)(V_MEM_BASE + disp_pos);

    *p_vmem++ = ch;
    *p_vmem++ = DEFAULT_CHAR_COLOR;
    disp_pos += 2;

    set_cursor(disp_pos/2);
}
```

`V_MEM_BASE` 为显存的起始地址 0xB8000，`V_MEM_BASE + disp_pos` 即为当前显示位置的地址，直接把字符写入特定地址，然后重新设置光标位置。

- 控制台切换 `select_console` 函数：

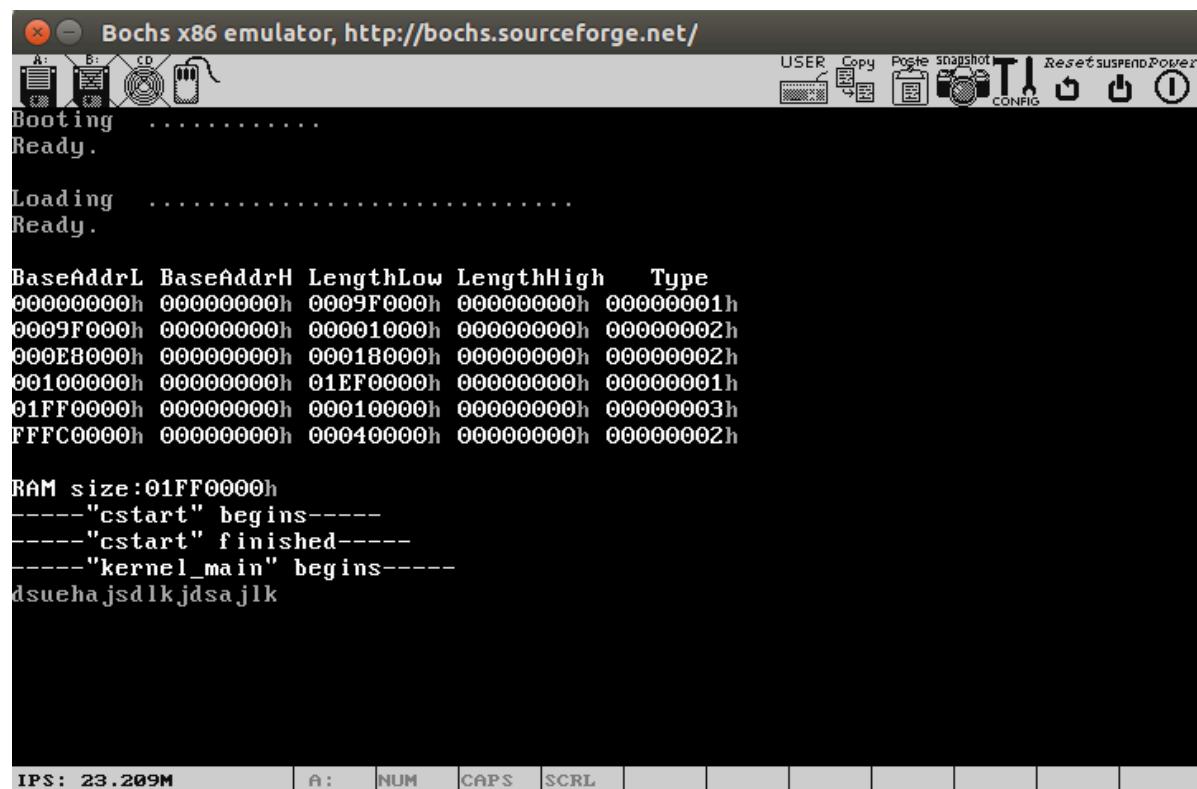
```
PUBLIC void select_console(int nr_console)      /* 0 ~ (NR_CONSOLES - 1) */
{
    if ((nr_console < 0) || (nr_console >= NR_CONSOLES)) {
        return;
    }

    nr_current_console = nr_console;
    set_cursor(console_table[nr_console].cursor);
    set_video_start_addr(console_table[nr_console].current_start_addr);
}
```

控制台切换主要分为三步：设置 `nr_current_console` 记录修改后的当前控制台，设置光标位置，设置起始显示的显存地址。

3.1.3.3 TTY 处理过程验证

编译运行程序，键盘输入一些字符，可以看到只有控制台 0 打印字符，使用 **alt + F1/2** 切换到其他控制台，由于 `nr_current_console` 初始化后没有变过，所以可以看到其他控制台没有动作。



Bochs x86 emulator, http://bochs.sourceforge.net/

Booting
Ready.

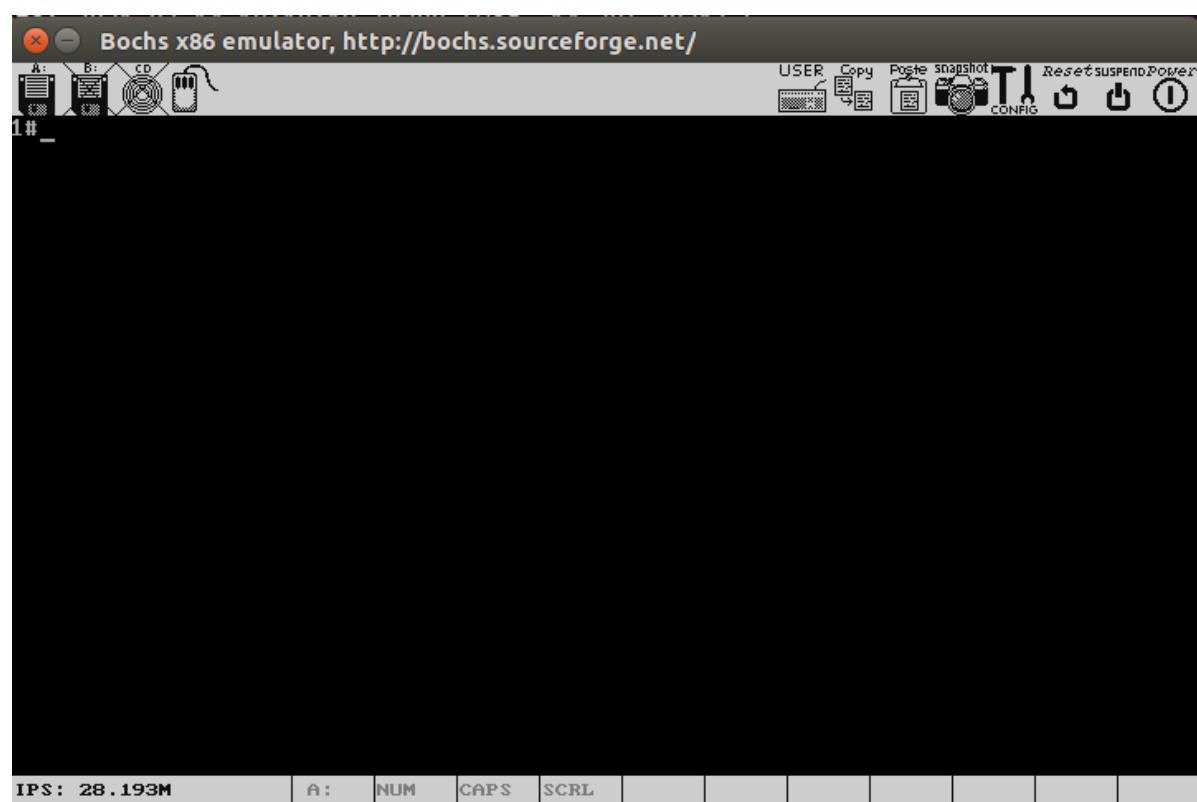
Loading
Ready.

BaseAddrL	BaseAddrH	LengthLow	LengthHigh	Type
00000000h	00000000h	0009F000h	00000000h	000000001h
0009F000h	00000000h	00001000h	00000000h	00000002h
000E8000h	00000000h	00018000h	00000000h	00000002h
00100000h	00000000h	01EF0000h	00000000h	00000001h
01FF0000h	00000000h	00010000h	00000000h	00000003h
FFFC0000h	00000000h	00040000h	00000000h	00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
dsueha jsd lkJdsajlk

IPS: 23.209M A: NUM CAPS SCRL

控制台 0



Bochs x86 emulator, http://bochs.sourceforge.net/

1# -

IPS: 28.193M A: NUM CAPS SCRL

Figure 14

控制台 1

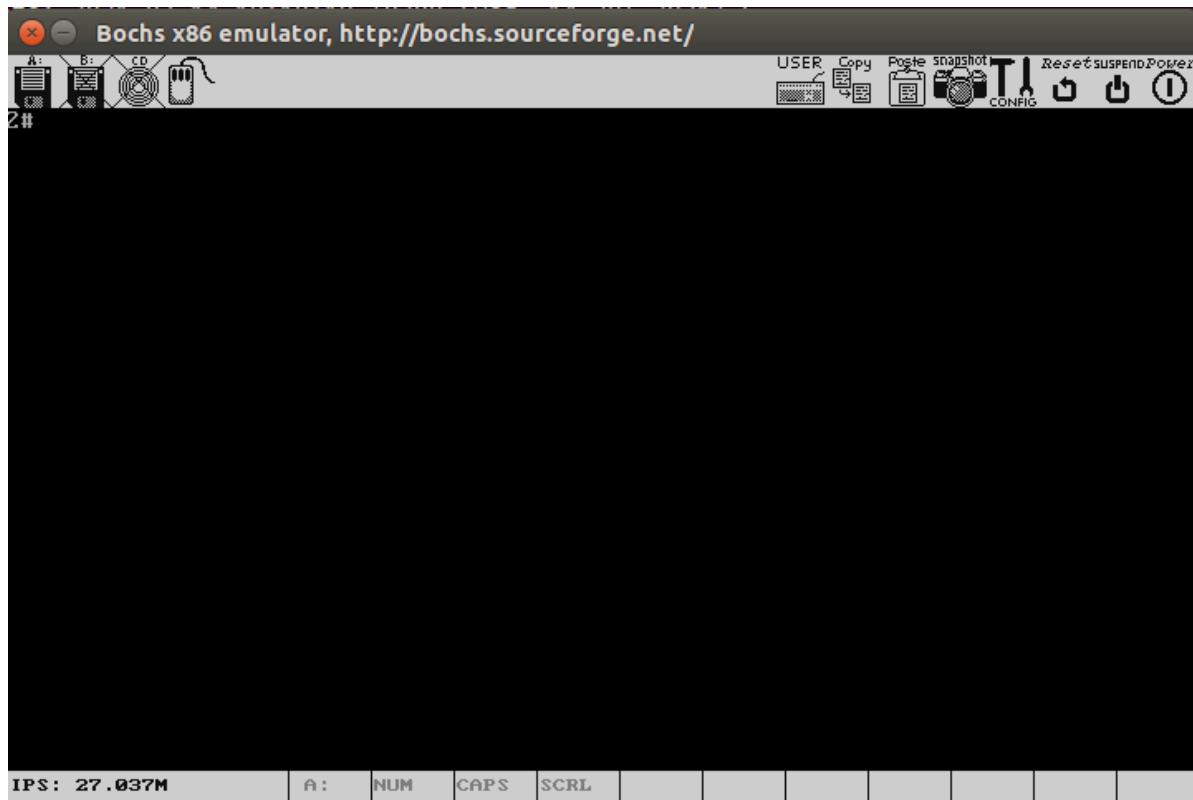


Figure 15

控制台 2

3.1.3.4 如何支持进程对 TTY 的读写

每个进程都是和一个 TTY 绑定的。通过上面的分析，我们可以知道进程对 TTY 的读写主要是通过 `tty_do_read()` 和 `tty_do_write()` 两个函数实现的。

- `tty_do_read()` : 它调用 `keyboard_read()` 处理扫描码并把读入的字符交给 `in_process()` 做进一步处理。如果是要输出的字符，`in_process()` 把它放入当前 TTY 的缓冲区中。
- `tty_do_write()` : 它从 TTY 缓冲区读取数据，在本实验中，函数直接讲读取的数据送入 `out_char()` 进行打印。实际上，TTY 可以对数据进行更多处理。

3.1.4 结合 `printf` 的实现代码，理解 I/O 系统调用的实现机理

`printf()` 要完成屏幕输出的功能，需要调用控制台模块的相应代码，因此它必须通过系统调用才能完成。

`printf()` 的实现过程如下：

3.1.4.1 指定进程对应的 TTY

在系统调用过程中，系统只知道当前系统调用是由哪个进程触发的，但是 `printf()` 时，操作系统必须知道往哪个控制台输出，因此在实现 `printf()` 之前必须在进程表中增加一个成员变量 `nr_tty`，指定进程对应的 TTY。

```
typedef struct s_proc {
    STACK_FRAME regs;           /* process registers saved in stack frame */

    u16 ldt_sel;               /* gdt selector giving ldt base and limit */
    DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */

    int ticks;                  /* remained ticks */
    int priority;

    u32 pid;                   /* process id passed in from MM */
    char p_name[16];           /* name of the process */

    int nr_tty;
}PROCESS;
```

在本实验中，所有进程在初始化时默认指定为对应第 0 个 TTY。接着，将 B 和 C 两个进程与第一个 TTY 绑定。

```
proc_table[1].nr_tty = 0;
proc_table[2].nr_tty = 1;
proc_table[3].nr_tty = 1;
```

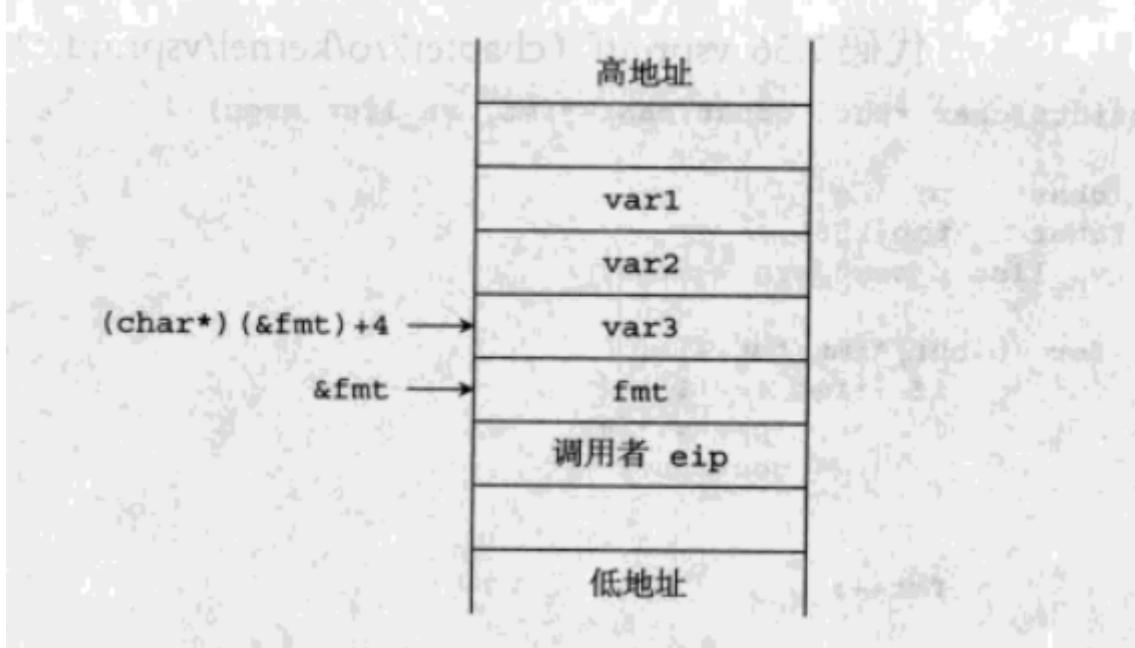
3.1.4.2 搭建 `printf()` 基本框架

`printf()` 与一般的函数不同，其参数的个数及类型都可变，而且其表示格式的参数（`%s`、`%x`）形式多样，在本实验中，首先以 `%x` 格式为例分析 `printf()` 的实现方法。

由于参数个数和类型是可变的，因此除第一个参数格式化字符串 `fmt` 外其他参数都用省略号进行接收，表示表示参数不知道有多少，也不知道是什么。

在 C 调用约定中，约定后面的参数先入栈，并且由调用者清理堆栈。因此当需要使用这些参数时，可以通过基准 `fmt` 确定，例如，相对 `fmt` 的第一个参数地址

为 `(char *)&fmt+4`，以此类推，栈的结构可概括如下：



为了确定参数的个数和输出格式，`printf()` 首先需要对格式化字符串进行解析，根据格式化符号来确定，这个过程由 `vsprintf()` 函数来完成，解析后需要打印的字符串放入缓冲区 `buf`。

接着，通过一个系统调用 `write()` 来将 `buf` 中的字符串输出到对应的 TTY 中。

函数基本框架如下：

```
int printf(const char *fmt, ...)  
{  
    int i;  
    char buf[256];  
  
    va_list arg = (va_list)((char *)(&fmt) + 4); /*4是参数fmt所占堆栈中的大小*/  
    i = vsprintf(buf, fmt, arg);  
    write(buf, i);  
  
    return i;  
}
```

3.1.4.3 `vsprintf()` 的实现

对格式化字符串 `fmt` 逐个检查，若不为 `%` 说明可直接打印，直接输出到 `buf` 中；否则说明后面跟着一个占位符，进一步判断占位符类型。由于本实验只实现 `%x` 占位符，因此只需实现 `%x` 对应的动作。

使用 `p_next_arg` 记录当前将要处理的参数，每次碰到 `%x` 占位符，取出 `p_next_arg` 对应的参数，转换为十六进制形式显示的字符串，然后 `p_next_arg` 指向下一个参数。

`vsprintf()` 代码如下：

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    for (p=buf; *fmt; fmt++) {
        if (*fmt != '%') {
            *p++ = *fmt;
            continue;
        }

        fmt++;

        switch (*fmt) {
        case 'x':
            itoa(tmp, *((int*)p_next_arg));
            strcpy(p, tmp);
            p_next_arg += 4;
            p += strlen(tmp);
            break;
        case 's':
            break;
        default:
            break;
        }
    }

    return (p - buf);
}
```

3.1.4.4 系统调用 `write()` 的实现

首先需要实现一个对外的系统调用接口 `write()`，并在代码中函数声明。这个函数传入必要的参数，指定功能号 `eax` 为 `_NR_write`，表示需要调用的函数是 `sys_write()`，然后用 `int` 命令触发系统调用中断。

```
write:
    mov     eax, _NR_write
    mov     ebx, [esp + 4]
    mov     ecx, [esp + 8]
    int     INT_VECTOR_SYS_CALL
    ret
```

然后在 `sys_call_table` 中增加成员 `sys_write()`，使得它可以通过 `eax` 功能号找到。

```
PUBLIC system_call sys_call_table[NR_SYS_CALL] = {sys_get_ticks, sys_write};
```

注意，`sys_write()` 比 `write()` 多了一个参数指向当前的进程，这个参数是在修改的 `sys_call` 中压栈的。

```

`sys_call:
    call    save
    push    dword [p_proc_ready]
    sti

    push    ecx
    push    ebx
    call    [sys_call_table + eax * 4]
    add    esp, 4 * 3

    mov     [esi + EAXREG - P_STACKBASE], eax
    cli
    ret

```

由于当前运行的进程就是通过设置 `p_proc_ready` 来恢复执行的，所以当进程切换到未发生之前，`p_proc_ready` 的值就是指向当前进程的指针。把它压栈，就将当前进程（即 `write()` 的调用者指针）传递给了 `sys_write()`。

内核部分的函数 `sys_write()` 通过调用新增加的简单函数 `tty_write()` 来实现字符的输出。

```

PUBLIC void tty_write(TTY* p_tty, char* buf, int len)
{
    char* p = buf;
    int i = len;
|
    while (i) {
        out_char(p_tty->p_console, *p++);
        i--;
    }
}

/*=====
 *===== sys_write
 *=====*/
PUBLIC int sys_write(char* buf, int len, PROCESS* p_proc)
{
    tty_write(&tty_table[p_proc->nr_tty], buf, len);
    return 0;
}

```

3.1.4.5 实验结果

这样，`printf()` 就成功实现了。接着在 3 个用户进程中调用它。

```

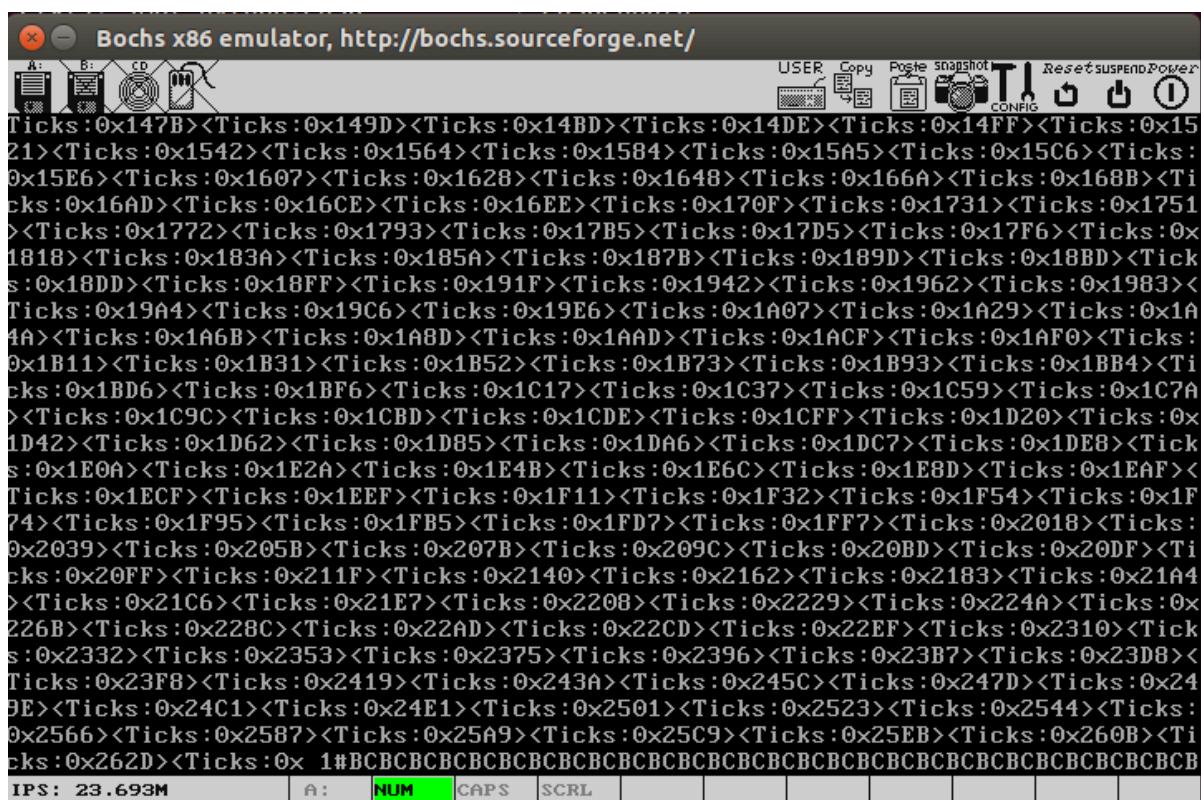
/*=====
        TestA
=====
void TestA()
{
    int i = 0;
    while (1) {
        printf("<Ticks:%x>", get_ticks());
        milli_delay(200);
    }
}

/*=====
        TestB
=====
void TestB()
{
    int i = 0x1000;
    while(1){
        printf("B");
        milli_delay(200);
    }
}

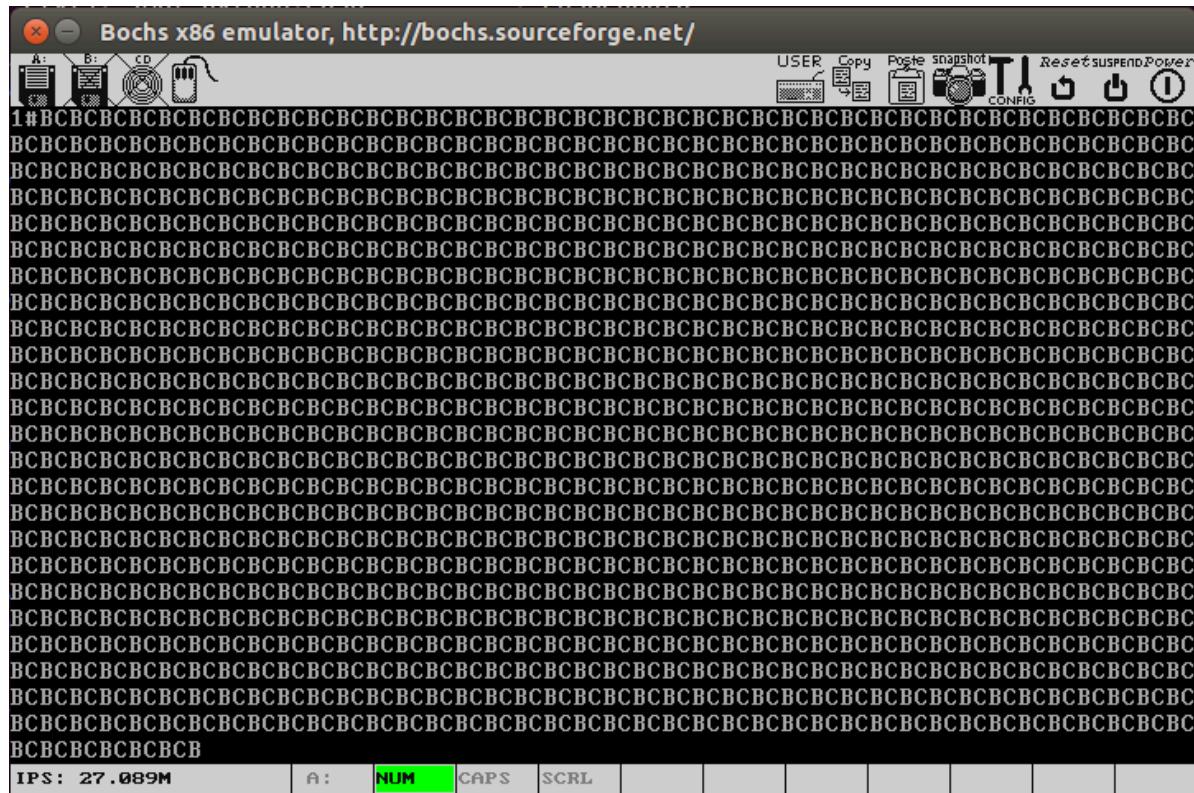
/*=====
        TestB
=====
void Testc()
{
    int i = 0x2000;
    while(1){
        printf("C");
        milli_delay(200);
    }
}

```

运行，控制台 0 查看效果：



控制台 1 查看效果：



因此，根据对 `printf()` 实现代码分析，可以概括 I/O 系统调用的实现机理为：

表 7.6 增加一个系统调用的过程

步骤	内容	文件
1	<code>NR_SYS_CALL</code> 加一	const.h
2	为 <code>sys_call_table[]</code> 增加一个成员，假设是 <code>sys_foo</code>	global.c
3	<code>sys_foo</code> 的函数体	因具体情况而异
4	<code>sys_foo</code> 的函数声明	proto.h
5	<code>foo</code> 的函数声明	proto.h
6	<code>_NR_foo</code> 的定义	syscall.asm
7	<code>foo</code> 的函数体	syscall.asm
8	添加 <code>global foo</code>	syscall.asm
9	如果参数个数与以前的系统调用比有所增加，则需要修改 <code>sys_call</code>	kernel.asm

3.2 实验问题与故障分析

3.2.1 库函数错误

使用 `strlen` 时出现错误：

```
undefined reference to 'strlen'
```

Fence 20

原因： `strlen` 是标准库函数，但内核通常不链接标准 C 库。因此，使用 `strlen` 会导致链接错误。

修复方案：用自定义的字符串长度计算函数替代 `strlen`，例如：

```
PRIVATE int string_length(const char* str) {
    int len = 0;
    while (*str++) len++;
    return len;
}
```

Fence 21

3.2.2 未声明函数

在自定义 TTY 时出现如下错误：

```
error: static declaration of 'switch_color' follows non-static
declaration
```

Fence 22

原因： `switch_color` 在调用之前没有正确声明，因此编译器假设它是非静态函数（隐式声明）。然后在定义时，被声明为 `static` 或 `PRIVATE`（即静态函数），导致冲突。

修复方案：因此，需要在 `tty.c` 的顶部，添加 `switch_color` 的声明：

```
PUBLIC void switch_color(CONSOLE* p_con);
```

Fence 23

* 四、实验结果总结



(对实验结果进行分析，完成思考题目，并提出实验的改进意见)

4.1 实验总结

- 实验通过验证键盘扫描码与键盘中断的过程，成功观察到键盘能够相应基本键盘按键并打印，验证成功。
- 实验通过验证分析 tty 的处理过程，观察到只有当前控制台读取键盘输入，并能够通过 **alt + Fn** 切换到其他控制台。
- 实验通过分析验证 printf 的实现代码，成功观察到进程调用的 printf 信息打印在对应控制台上，通过 printf 的实现理解了 I/O 系统调用的实现机理。

4.2 思考题

4.2.1 键盘输入的处理程序

4.2.1.1 解释 Scancode、MakeCode、BreakCode 的区别

- **Scancode**（扫描码）：扫描码是敲击键盘所产生的编码，表示按键的唯一标识符。每次按键操作都会在键盘编码器（如 Intel 8048）中生成相应的扫描码，并传输到计算机的键盘控制器，然后再传递给 CPU。具体过程如下：

当 8048 检测到一个键的动作后会把相应的扫描码发送给 8042。8042 会把它转换成相应的 Scancode set 1 扫描码，并将其放置在输入缓冲区中。接下来 8042 向主机中的可编程控制器 8259A 请求中断，在该过程中如果键盘产生了新的扫描码则不会被 8042 接受，直到缓冲区清空。

- **MakeCode**：当按下一个按键或保持按住状态时，会生成一个 MakeCode，表示“按下”这一动作。每个按键都有唯一的 MakeCode。
- **BreakCode**：当松开一个按键时，会生成一个 BreakCode，表示“松开”这一动作。BreakCode 通常是通过将 MakeCode 加上一个标志位（0x80）来生成的。

因此，在键盘中断处理中，Makecode 和 Breakcode 同属于扫描码 Scancode，是扫描码的两个子类，MakeCode 和 BreakCode 分别用来识别当前按键是按下还是松开。除了特殊键 Pause，其他按键都有对应的 MakeCode 和 BreakCode。使用 Makecode 与 80H 进行按位或操作即可得到 Breakcode。

4.2.1.2 解释本实验如何解析扫描码

- ① 从键盘缓冲区读取扫描码：每次调用 `keyboard_read` 时，从键盘输入缓冲区 `kb_in` 中读取一个扫描码。`scan_code = get_byte_from_kbuf();` 用于从键盘缓冲区中获取下一个扫描码。
- ② 处理特殊扫描码（Pause/Break 和 PrintScreen）：
 - **Pause/Break 键：**当扫描码为 `0xE1` 时，代码检查后续的字节序列。如果序列符合 `0xE1, 0x1D, 0x45, 0xE1, 0x9D, 0xC5`，则将 `key` 设置为 PAUSEBREAK，表示这是 Pause/Break 键。

```
if (scan_code == 0xE1) {  
    int i;  
    u8 pausebrk_scode[] = {0xE1, 0x1D, 0x45,  
                           0xE1, 0x9D, 0xC5};  
    int is_pausebreak = 1;  
    for(i=1;i<6;i++){  
        if (get_byte_from_kbuf() != pausebrk_scode[i]) {  
            is_pausebreak = 0;  
            break;  
        }  
    }  
    if (is_pausebreak) {  
        key = PAUSEBREAK;  
    }  
}
```

Fence 24

- **PrintScreen 键：**如果扫描码以 `0xE0` 开头且匹配 `0xE0, 0x2A, 0xE0, 0x37`，表示按下了 PrintScreen 键；如果是 `0xE0, 0xB7, 0xE0, 0xAA`，则表示松开了 PrintScreen 键。

```
else if (scan_code == 0xE0) {  
    scan_code = get_byte_from_kbuf();  
  
    /* PrintScreen 被按下 */  
    if (scan_code == 0x2A) {  
        if (get_byte_from_kbuf() == 0xE0) {
```

```

        if (get_byte_from_kbuf() == 0x37) {
            key = PRINTSCREEN;
            make = 1;
        }
    }
/* PrintScreen 被释放 */
if (scan_code == 0xB7) {
    if (get_byte_from_kbuf() == 0xE0) {
        if (get_byte_from_kbuf() == 0xAA) {
            key = PRINTSCREEN;
            make = 0;
        }
    }
}

```

Fence 25

- ③ 判断 MakeCode 和 BreakCode：通过位与 `0x80`，若值为 0 则是 MakeCode，否则为 BreakCode。

```
make = (scan_code & FLAG_BREAK ? 0 : 1);
```

Fence 26

- ④ 将扫描码映射到字符：

在实验中，为了将扫描码（Scancode）转换成实际的字符或键功能，建立了一个数组 `keymap`。在 `keymap` 中，扫描码作为索引，数组中的元素表示相应的字符或键功能。为了适应不同的组合键和前缀标志，`keymap` 的每一行包含三个值：

- 默认：使用第一列（单独按下的字符）
- Shift 键：如果 `shift_l` 或 `shift_r` 标志为 `1` 或大写锁定启用，`column = 1`，使用第二列的字符。
- 0xE0 前缀：如果 `code_with_E0` 为 `1`，设置 `column = 2`，选择第三列的字符或功能。

数组的结构定义如下：

```

u32 keymap[NR_SCAN_CODES * MAP_COLS] = {
    /* scan-code           !Shift      Shift      E0 XX*/
/*=====*/
    /* 0x00 - none          */     0,         0,         0,
    /* 0x01 - ESC            */     ESC,       ESC,       0,
    /* 0x02 - '1'            */     '1',       '! ',      0,
    /* 0x03 - '2'            */     '2',       '@ ',      0,
    ...
};


```

Fence 27

通过扫描码 `scan_code` 定位到 `keymap` 数组的对应行。使用 `0x7F` 去除高位标志，以确保索引在 `keymap` 范围内。`column` 用于选择键映射的列。

```

keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];
key = keyrow[column];

```

Fence 28

⑤ 处理组合键（Shift、Ctrl、Alt）：

使用 `switch` 语句根据 `key` 值处理特殊键（Shift、Ctrl、Alt、CapsLock、NumLock、ScrollLock）：

- 当检测到功能键时，使用 `make` 标识赋值给功能键的状态（1 表示按下，0 表示松开）

```

switch(key) {
    case SHIFT_L:
        shift_l = make;
        break;
    case SHIFT_R:
        shift_r = make;
        break;
    case CTRL_L:
        ctrl_l = make;
        break;
    case CTRL_R:
        ctrl_r = make;
        break;
    case ALT_L:
        alt_l = make;
        break;
    case ALT_R:
        alt_l = make;
        break;
}

```

- 其中 `shift_l` 或者 `shift_r` 可用于定位 keymap 中的列，从而得到大写字母

```
int caps = shift_l || shift_r;
```

- 最后对于扫描码为 makecode 的 key，会与功能键相应的 FLAG 按位或操作，便于扫描码处理程序对功能键进行识别。

```
key |= shift_l ? FLAG_SHIFT_L : 0;
key |= shift_r ? FLAG_SHIFT_R : 0;
key |= ctrl_l ? FLAG_CTRL_L : 0;
key |= ctrl_r ? FLAG_CTRL_R : 0;
key |= alt_l ? FLAG_ALT_L : 0;
key |= alt_r ? FLAG_ALT_R : 0;
key |= pad ? FLAG_PAD : 0;
```

4.2.1.3 解释键盘输入缓冲区的作用，以及处理过程

键盘缓冲区的作用：用于临时存储从键盘接收到的扫描码，以确保按键信息的完整性和顺序性。这种设计使得键盘输入可以以中断的方式被处理，避免了按键信息丢失，并能够实现非阻塞的键盘输入处理。

- ① **防止丢失按键输入：**键盘输入是异步的，每次按键（包括按下和释放）都会触发一个中断并生成一个扫描码。如果处理程序不能快速地将扫描码传递到应用层，按键信息可能会丢失。缓冲区的存在可以有效避免这种情况，将每次按键的扫描码都暂存到缓冲区，确保处理完当前按键后下一个扫描码不会被遗漏。
- ② **保障扫描码的顺序处理：**键盘缓冲区是一个先进先出（FIFO）的循环缓冲区，保证按键按下和释放的顺序一致。因此，缓冲区可以按顺序依次存储扫描码，并在应用层按正确的顺序处理，避免了因多个按键输入导致的扫描码顺序混乱。
- ③ **提高系统响应效率：**由于键盘缓冲区支持异步存储，主程序不必阻塞等待按键处理完成。扫描码存入缓冲区后，主程序可以继续执行其他任务，从而提高了系统的响应效率。

键盘输入缓冲区的处理过程：

- ① 每次按键动作都会产生一个键盘中断，键盘中断处理程序 `keyboard_handler` 被触发。
- ② `keyboard_handler` 通过从键盘控制器的 `0x60` 端口读取扫描码，并将其存入键盘缓冲区 `kb_in` 中。
- ③ 如果缓冲区未满 (`count < KB_IN_BYTES`)，将扫描码写入缓冲区位置 `p_head`，然后更新 `p_head` 和 `count`。
- ④ 如果 `p_head` 达到缓冲区末尾，则循环到开头继续存储（即循环缓冲区的特性）。

4.2.1.4 为什么在/c/的 `klib.asm`, 代码 7.12 中, 需要 `disable_int`, `enable_int`?

在 `get_byte_from_kbuf` 函数中使用了 `disable_int` 和 `enable_int`，即在读取时临时关闭中断，读取完成后重新开启中断。

`disable_int` 和 `enable_int` 二者用于临界区的保护：键盘缓冲区 `kb_in` 是一个共享资源，由 键盘中断处理程序 和 读取函数 `get_byte_from_kbuf` 共同访问。中断处理程序会不断地将新的扫描码写入 `kb_in`，而 `get_byte_from_kbuf` 则从中读取数据。如果在 `get_byte_from_kbuf` 读取缓冲区时发生键盘中断，导致缓冲区的数据结构（如 `count`、`p_tail`）被修改，那么读取操作可能会不完整，甚至导致数据不一致。

- `disable_int` 使用 `cli` 指令来关闭中断，防止在读取 `kb_in` 的过程中发生中断，确保缓冲区的状态不会被其他进程或中断处理程序更改。
- `enable_int` 则使用 `sti` 指令来开启中断，使系统可以继续响应其他中断。

4.2.2 解释一下，重新设置显示开始地址的原理

在文本模式下，显示器显示的内容是从显存中读取的。默认情况下，屏幕从显存的起始地址（`0xB8000`）开始读取数据。改变开始地址能够显示出不同显存段的字符内容。

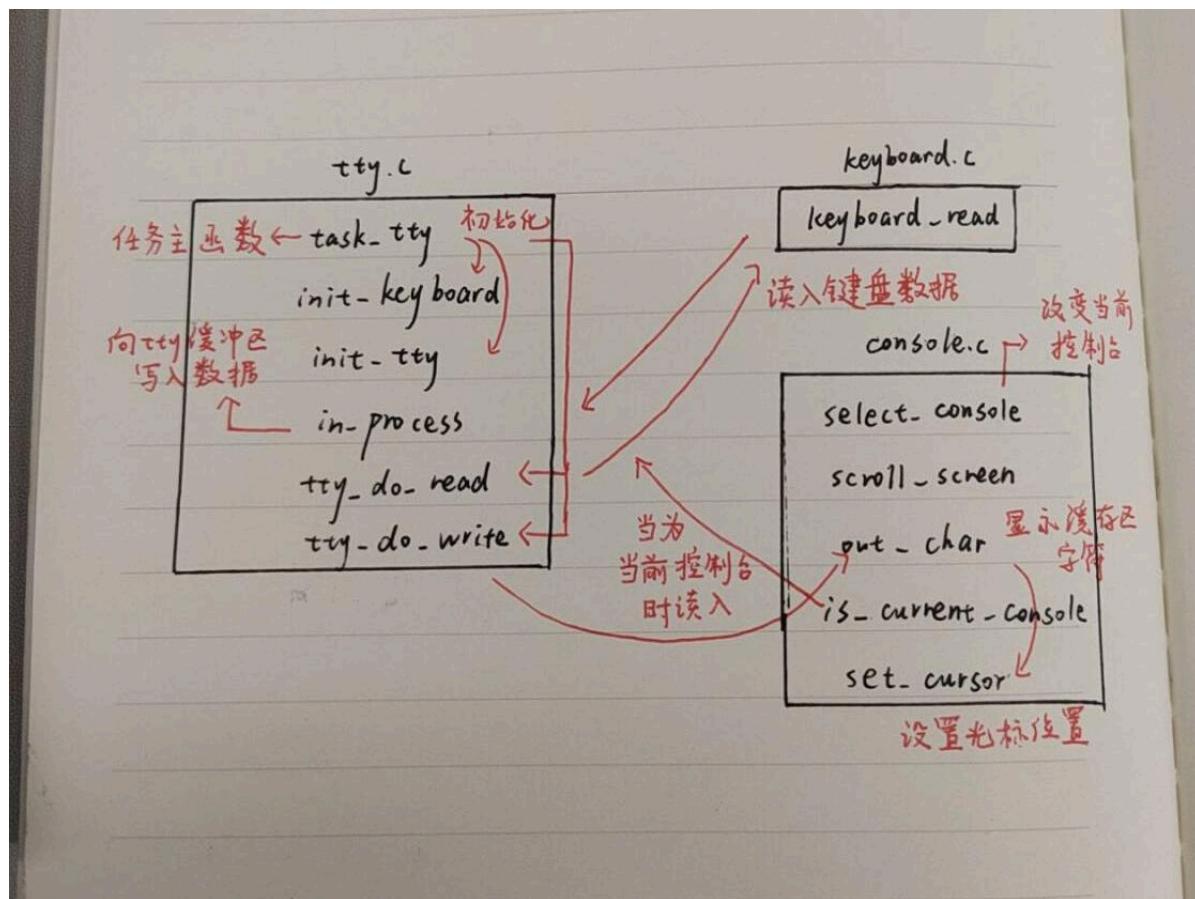
VGA 提供了地址寄存器和数据寄存器。每次操控寄存器需要通过向地址寄存器 `0x3D4` 写入寄存器的索引值，选择具体的寄存器。之后通过数据寄存器 `0x3D5` 写入或读取寄存器的值。

其中数据寄存器 Start Address High Register 和 Start Address Low Register 来控制显示内容的起始地址。通过修改 显示开始地址寄存器，可以动态改变屏幕显示内容对应的显存起始位置，从而实现屏幕内容的切换或滚动。

4.2.3 解释多个 tty，能够并发处理输入输出的原理，结合代码给出一个比书上框图更为细致的流程图。

原理： 在 TTY 结构体中，为每个 TTY 定义了 TTY 缓冲区。`task_tty()` 对每个 TTY 进行遍历，只有当前控制台才能够读取键盘缓冲区，因此在每个时刻总是最多只有一个控制台在进行读入。而由于写入过程是从 TTY 自己的缓冲区中读取数据，因此写入也是不冲突的。综上，多个 TTY 能够并发处理输入输出的过程。

流程图：



4.2.4 动手做 1：请添加一个你自己个性化的 TTY，在这个 TTY 上，你可以根据键盘输入或者移动光标的某种规律，运行一个你的彩蛋程序，而在其他 TTY 中不会有这个效果

4.2.4.1 个性化 TTY

- 首先需要修改 **TTY** 结构：在 **TTY** 结构中添加一个 **is_special** 标志，用来标记这个 TTY 是否启用彩蛋功能。

```
typedef struct s_tty {  
    u32 in_buf[TTY_IN_BYTES]; /* 输入缓冲区 */  
    u32* p_inbuf_head; /* 指向缓冲区的下一个空闲位置 */  
    u32* p_inbuf_tail; /* 指向缓冲区中下一个可读位置 */  
    int inbuf_count; /* 缓冲区中已存数据的数量 */  
    struct s_console* p_console; /* 指向对应的控制台 */  
    int is_special; /* 是否是特殊 TTY */  
} TTY;
```

Fence 32

- 初始化 **TTY** 时启用特定 TTY 的彩蛋

```
PRIVATE void init_tty(TTY* p_tty) {  
    p_tty->inbuf_count = 0;  
    p_tty->p_inbuf_head = p_tty->p_inbuf_tail = p_tty->in_buf;  
  
    int nr_tty = p_tty - tty_table; // 当前 TTY 的编号  
    // 设置是否为特殊 TTY  
    p_tty->is_special = (nr_tty == 2) ? 1 : 0;  
  
    init_screen(p_tty);  
}
```

Fence 33

4.2.4.2 自由移动光标

教材代码并未实现方向键的自由移动光标，而是通过 `shift + 方向键` 实现屏幕滚动。因此为实现上下左右箭头键控制光标自由移动，可以封装一个函数 `move_cursor`，并在 `in_process` 函数中处理箭头键的输入。下面是具体步骤：

- 定义方向宏：

```
#define SCR_UP 0 /* scroll forward */  
#define SCR_DN 1 /* scroll backward */  
#define SCR_LT 2 /* scroll left */  
#define SCR_RT 3 /* scroll right */
```

Fence 34

② 新增 **move_cursor** 函数：

- 边界判断：不越过屏幕顶部或者底部。
- 根据不同方向移动光标：若是向上移动。则让 **p_con->cursor** 减去 **SCREEN_WIDTH**，向左移动光标则 **p_con->cursor** 自减，其他同理。
- 屏幕滚动跟随光标：当屏幕光标超过显示范围，需要调整屏幕显示起始地址，向上或者向下挪动一行以适应光标。
- 更新光标和屏幕状态：最后使用封装的 **flush** 函数更新光标和屏幕状态

```
=====
=====
          move_cursor
-----
-----
  根据方向移动光标

=====
=====
PUBLIC void move_cursor(CONSOLE* p_con, int direction)
{
    switch (direction) {
        case SCR_UP: // 向上移动光标
            if (p_con->cursor > p_con->original_addr) { // 不越过屏幕
                p_con->cursor -= SCREEN_WIDTH;
            }
            break;
        case SCR_DN: // 向下移动光标
            if (p_con->cursor + SCREEN_WIDTH <
                p_con->original_addr + p_con->v_mem_limit) { // 不越
                p_con->cursor += SCREEN_WIDTH;
            }
            break;
        case SCR_LT: // 向左移动光标
            if (p_con->cursor > p_con->original_addr) { // 不越过屏幕
                p_con->cursor--;
            }
            break;
        case SCR_RT: // 向右移动光标
            if (p_con->cursor <
```

```

        p_con->original_addr + p_con->v_mem_limit - 1) { //  

    不越过屏幕右侧  

        p_con->cursor++;  

    }  

    break;  

default:  

    break;  

}  
  

// **调整屏幕显示起始地址，使屏幕滚动跟随光标**  

if (p_con->cursor < p_con->current_start_addr) {  

    // 光标在当前显示范围之上  

    p_con->current_start_addr -= SCREEN_WIDTH;  

} else if (p_con->cursor >= p_con->current_start_addr +  

SCREEN_SIZE) {  

    // 光标在当前显示范围之外（下方）  

    p_con->current_start_addr += SCREEN_WIDTH;  

}  

flush(p_con); // 更新光标和屏幕状态
}

```

Fence 35

③ 更新 `in_process` 函数：

在 `in_process` 函数中，根据上下左右键的扫描码调用 `move_cursor`：

```

if(p_tty->is_special)
{
    switch(raw_code)
    {
        case UP:
            if (!(key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
            {
                move_cursor(p_tty->p_console, SCR_UP);
            }
            break;
        case DOWN:
            if (!(key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
            {
                move_cursor(p_tty->p_console, SCR_DN);
            }
            break;

        case LEFT:

```

```
        if (!(key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
        {
            move_cursor(p_tty→p_console, SCR_LT);
        }
        break;
    case RIGHT:
        if (!(key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
        {
            move_cursor(p_tty→p_console, SCR_RT);
        }
        break;
}
}
```

效果如下：

Fence 36

我们在 2 号 TTY 可以自由移动光标，当向下移动超出显示范围，屏幕将会向下滚动一行，2# 的字符串不在显示范围内。



Figure 16



Figure 17

4.2.4.3 切换显示颜色

为了实现按 **F1** 键顺序切换显示颜色（如 **WHITE**、**RED**、**GREEN**、**BLUE** 等），可以在 **CONSOLE** 结构中添加一个用于记录当前颜色索引的变量，并根据该索引切换颜色。

- ① 颜色常量定义：在 **const.h** 中定义一些常见颜色：

```
#define BLACK    0x0      /* 0000 */
#define WHITE    0x7      /* 0111 */
#define RED     0x4      /* 0100 */
#define GREEN   0x2      /* 0010 */
#define BLUE    0x1      /* 0001 */
#define FLASH   0x80     /* 1000 0000 */
#define BRIGHT  0x08     /* 0000 1000 */
#define YELLOW  0x0E     /* 黄色 */
#define CYAN    0x0B     /* 青色 */
#define PURPLE  0x5      /* 紫色 */
```

Fence 37

- ② 修改 **CONSOLE** 结构：在 **CONSOLE** 结构中添加 **char_color** 和 **color_index** 属性，用于存储当前字符颜色和索引：

```
typedef struct s_console {
    unsigned int current_start_addr; /* 当前显示起始地址 */
    unsigned int original_addr;     /* 显存的起始地址 */
    unsigned int v_mem_limit;       /* 显存大小 */
    unsigned int cursor;           /* 光标位置 */
    u8 char_color;                /* 当前字符颜色 */
    u8 color_index;               /* 当前颜色索引 */
} CONSOLE;
```

Fence 38

③ 定义颜色数组：

在 `console.c` 文件中定义颜色数组，按顺序存储颜色值，其中 FLASH 是闪烁

```
PRIVATE u8 colors[] = {
    WHITE,
    BRIGHT | WHITE,
    RED,
    GREEN,
    PURPLE,
    YELLOW,
    CYAN,
    FLASH | RED
};

#define NUM_COLORS (sizeof(colors) / sizeof(colors[0]))
```

Fence 39

④ 实现颜色切换函数：创建一个函数 `switch_color`，切换到下一种颜色

```
PUBLIC void switch_color(CONSOLE* p_con) {
    // 切换到下一个颜色
    p_con->color_index = (p_con->color_index + 1) % NUM_COLORS;
    p_con->char_color = colors[p_con->color_index];
}
```

Fence 40

⑤ 按 `F1` 切换颜色：在 `in_process` 函数中，为了与 `ALT` + `F1` 功能分开，选择单独处理 `F1` 键时调用 `switch_color`

```

case F1:
    if (!(key & FLAG_ALT_L) || (key & FLAG_ALT_R))
    {
        switch_color(p_tty→p_console);
    }
    break;

```

Fence 41

最后效果如下，其中感叹号实现了闪烁的效果：



Figure 18

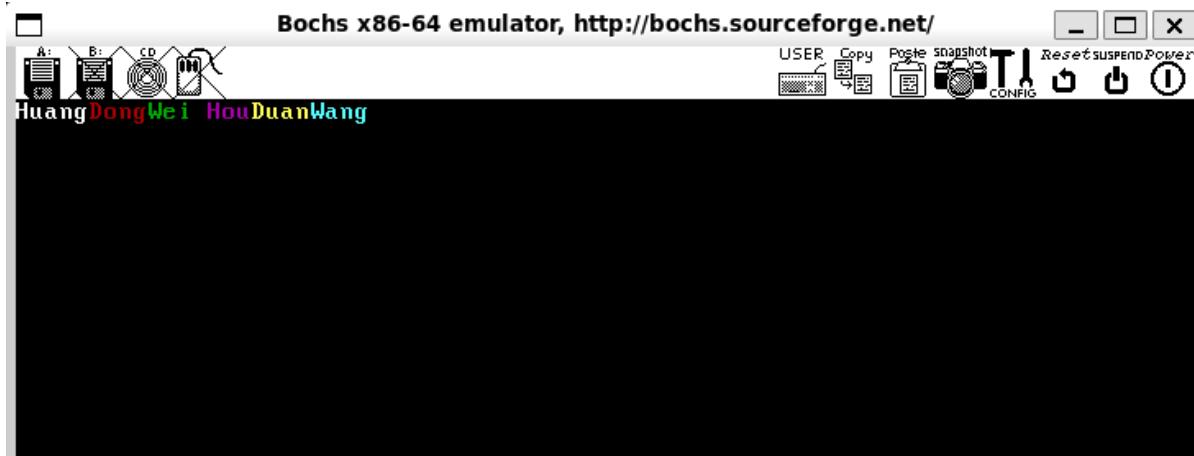


Figure 19

4.2.4.4 输出字符串图案

- ① 得到字符：使用 `figlet` 生成字符串图案。`figlet` 是一个命令行工具，可以将普通文本转换为 ASCII 字符串图案。我们可以使用现成的 `figlet` 工具生成字符串。

```
figlet -f slant "HDW"
```

Fence 42

- ② 输出字符：添加一个函数 `draw_pattern`，在屏幕上打印一个简单的字符图案：

```

PUBLIC void draw_pattern(CONSOLE* p_con) {
    static char* art[] = {
        "      _ ----- _      _      ",
        " / / / / \ \ |   / /      ",
        " / / / / / / | / /      ",
        " / _ / / / / | / / /      ",
        "/_ / / / / / / | / / /      ",
        "                                     // 保留空行用于格式完整
    };
}

int start_row = (p_con->cursor / SCREEN_WIDTH) + 1; // 从当前行下一行开始
for (int i = 0; i < 6; i++) {
    int tmp = p_con->cursor;
    for (int j = 0; j < string_length(art[i]); j++) {
        if (art[i][j] == '\0') break;
        out_char(p_con, art[i][j]);
    }
    p_con->cursor = tmp + SCREEN_WIDTH; // 跳到下一行
}
flush(p_con); // 刷新屏幕
}

```

Fence 43

- ③ 添加功能键处理：在 `in_process` 函数中处理功能键：

```

case F2:
    if (!(key & FLAG_ALT_L) || (key & FLAG_ALT_R))
    {
        draw_pattern(p_tty->p_console);
    }
    break;

```

Fence 44

最后配合前面的切换颜色，效果如下：



Figure 20

此外，如果要在程序中对用户输入的字符串调用 `figlet` 并显示输出，可以使用 `popen` 函数在程序中调用系统命令 `figlet`，捕获 `figlet` 的输出并存储为字符串。再显示。

```
void generate_and_display_art(const char* input) {
    char command[MAX_INPUT + 20];
    char output[MAX_OUTPUT];
    FILE* fp;

    // 构建 figlet 命令
    sprintf(command, sizeof(command), "figlet '%s'", input);

    // 使用 popen 执行 figlet 命令并捕获输出
    fp = popen(command, "r");
    if (fp == NULL) {
        fprintf(stderr, "Error executing figlet command.\n");
        return;
    }

    // 读取 figlet 输出
    printf("\nGenerated ASCII Art:\n");
    while (fgets(output, sizeof(output), fp) != NULL) {
        // 将 figlet 输出到屏幕
        printf("%s", output);
    }
}
```

```
// 关闭管道  
pclose(fp);  
}
```

Fence 45

4.2.5 动手做 2：尝试扩展一下 `printf`，让它支持 `%s`，想想目前的 `printf` 实现是否有什么安全漏洞？可以怎么解决。

4.2.5.1 拓展 `printf` 函数

要支持 `%s` 占位符，只需修改 `vsprintf` 函数，使得其解析格式化字符串时能够解析 `%s` 占位符。解析方式与 `%x` 一致，即根据 `%` 后的字符就能判断占位符的类型，从而知道如何从堆栈中取出参数。

对于 `%s` 占位符，由于对应的字符串是变长的，因此对应入栈的参数实际上是该字符串的指针。由于入栈的为字符串指针，故 `p_next_arg` 为指针的指针，其类型为 `char **`，以 `char *` 的方式读取参数，放入 `str` 变量，存放该字符串指针。

接着对字符串进行遍历，将 `str` 中的字符复制到 `buf` 中，直至遇到 `\0`。

修改后的 `vsprintf()` 如下：

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    char* str;

    for (p=buf; *fmt; fmt++) {
        if (*fmt != '%') {
            *p++ = *fmt;
            continue;
        }

        fmt++;

        switch (*fmt) {
        case 'x':
            itoa(tmp, *((int*)p_next_arg));
            strcpy(p, tmp);
            p_next_arg += 4;
            p += strlen(tmp);
            break;
        case 's':
            str = *((char**)p_next_arg);
            char *cur_ch = str;
            for(; *cur_ch; cur_ch++)
            {
                *p = *cur_ch;
                p++;
            }
            p_next_arg += 4;
            break;
        default:
            break;
        }
    }

    return (p - buf);
}
```

最后，在 `main.c` 中修改进程函数，来测试 `printf()` 支持 `%s` 的效果。为了确定设计好的 `printf` 能同时支持多个 `%x` 和 `%s` 占位符，使用两个 `%s` 和一个 `%x` 进行打印。

进程 A 的函数代码修改如下：

```
=====
TestA
=====
void TestA()
{
    int i = 0;
    char str1[32] = "hello world,A.";
    char str2[32] = "end\n";
    while (1) {
        //printf("<Ticks:%x>", get_ticks());
        printf("%s Ticks:%x %s", str1, get_ticks(), str2);
        milli_delay(200);
    }
}
```

编译运行，实验结果如下：

```
000E8000h 00000000h 00018000h 00000000h 00000002h  
00100000h 00000000h 01EF0000h 00000000h 00000001h  
01FF0000h 00000000h 00010000h 00000000h 00000003h  
FFFC0000h 00000000h 00040000h 00000000h 00000002h  
  
RAM size:01FF0000h  
----"cstart" begins----  
----"cstart" finished----  
----"kernel_main" begins----  
hello world,A. Ticks:0xF end  
hello world,A. Ticks:0x4F end  
hello world,A. Ticks:0x91 end  
hello world,A. Ticks:0xD3 end  
hello world,A. Ticks:0x115 end  
hello world,A. Ticks:0x156 end  
hello world,A. Ticks:0x195 end  
hello world,A. Ticks:0x1D6 end  
hello world,A. Ticks:0x216 end  
hello world,A. Ticks:0x257 end  
hello world,A. Ticks:0x299 end  
hello world,A. Ticks:0x2D7 end  
hello world,A. Ticks:0x319 end  
hello world,A. Ticks:0x35A end  
hello world,A. Ticks:0x39B end  
  
IPS: 15.720M | A: NUM CAPS SCRL
```

可以看到，成功输出了字符串，拓展成功。

4.2.5.2 `printf` 实现的格式化字符串漏洞

`printf` 的参数个数和类型都是可变的，因此函数并不知道实际传入参数个数，只是根据格式化字符串中的占位符来确定。确定参数个数后，函数依次读取栈上数据，直到满足参数个数为止，实现变长参数读取。

在正常调用函数过程中，占位符和参数个数往往是对应的，格式化字符串中有 3 个占位符，就额外提供 3 个参数。但是，当攻击者不按规则输入字符串时，就将导致漏洞。

攻击者故意不填充某占位符，而函数根据对格式化字符串的解析，仍然以为调用者提供了该参数，从栈上往下寻找，导致泄露了正常参数之外的栈数据。

例如，对于 `printf(buf);`，当 `buf = "%d %d"` 时，函数将在栈上往下寻找两个参数来对应占位符，导致相对于格式化字符串 1 个字和 2 个字位置的内存数据泄露。

在 C 语言中，还存在一些特殊的占位符。在通常情况下，程序遇到一个占位符，就按顺序向后寻找参数，但是我们可以使用 `数字+$` 的形式，直接指定参数相对于格式化字符串的偏移。例如，对于 `printf("%3$s %2$s %1$s", a, b, c);`，当程序看到 `%3$s` 的时候，就不是直接找相对于格式化字符串的第一个参数了，而是去找相对于格式化字符串的第三个参数，即为 `c`。

此外，`%n` 可以实现将已经输出的字符数写入指定参数。因此，利用格式化字符串漏洞，可以看出，`%n$s` 可以实现任意地址的数据泄露，`%n$n` 可以实现任意地址写，造成严重安全威胁。

4.2.5.3 `printf` 安全漏洞的解决方法

从分析过程可以看到，格式化字符串漏洞的根本原因是用户不按规则输入数据。因此，相应的防御方法是永远不信任用户输入，对所有用户输入进行检查。当用户输入带有占位符的文本时，发出安全警告。

此外，还可以对格式化函数的写操作指令进行边界检查，防御格式化串写越界。

4.3 实验改进意见

4.3.1 `printf` 的进一步拓展

在动手做环节中，我们对 `printf` 进行拓展，使其能够支持 `%s` 占位符。可以继续修改代码，使其能够支持的占位符更多，例如 `%u`、`%p` 等等。

4.3.2 键盘中断处理过程的补充

在本实验中，我们对所有基本的扫描码及其组合进行了处理，但是对于部分特殊按键（例如 F3 - F12）我们只进行了扫描码的读取，在 `in_process` 中没有对其进行处理。可以查询资料，探讨现代操作系统中，F11 按键的截屏，F1 按键的降低亮度等功能是怎么实现的。

4.3.3 课程内容的知识梳理

实验涉及的课本内容较多，且与之前所学的中断与系统调用知识有交叉，老师可以对之前所学知识进行一个简单回顾和梳理，帮助同学们加深对知识的理解。

* 五、各人实验贡献与体会

1 程序：

- 承担任务：独立完成所有实验，负责实验过程的 TTY 和 printf 部分，思考题的 TTY 部分和动手做 2，完成实验总结部分，补充部分实验故障和实验改进建议。
- 个人体会：本次实验中我们学习了键盘中断的处理过程，由于键盘按键的不同组合达到的效果不同，中断处理时需要涉及到大量的判断。参考资料以循序渐进的方式，从最简单的判断开始不断增加代码，降低了读者阅读的难度。

此外，实验还涉及了显示器输出的原理，并实现了屏幕滚动、操作光标等功能来验证所学的知识。在大一学习汇编语言课程时，对光标的操作只是打印/清空一个 | 来模拟光标的闪烁，对屏幕滚动也只是粗暴的清空整行，再把下一行复制上来。而在本实验中，我们学习了一种更为便捷，更符合编程习惯的方式，通过读写 VGA 寄存器端口来进行显示器的设置，进一步加深了我对于显示器输出基本原理的理解。

同时，实验还介绍了 printf 函数的实现机制，将之前 TTY 和 显示器显示部分知识进行串联，同时进一步巩固了实现系统调用的方法，通过拓展 printf 的动手做环节，加深了对实验内容的理解。

② 黄东威：

- 承担任务：此次实验本人独立完成了所有的实验内容，并在和小组成员讨论的情况下，共同参与了实验问题与故障分析、动手做 1 等部分实验报告的撰写。
- 个人体会：本次实验我深入地参与了代码的阅读和调试，过程中也遇到了程序异常崩溃的情况。经过老师的指导明白过来是自己的 gcc 版本更高导致的 proc 数组未赋足够的初值 导致，提高了我对 C 语言的理解。实验过程中键盘中断的实验和完善给了我很大的兴趣，在更早的实验中，我和小组成员已经就按下键盘是响应几次中断进行了讨论和尝试，这次的理论知识补充验证了我们的猜想，令我感到很满足。对于 tty 部分知识的了解我此前从未有过，跟着 oranges 书一起 褪去 tty 神秘的面纱 的过程也充满乐趣，最终我们完成了更好的打印函数 “printf” 也算是 标志着我们的 IO 部分的一个里程碑。整体而言，通过对汇编和 C 代码的阅读和分析，我增添了更多对底层知识的认知，对操作系统和计算机的运作原理有了更深的了解。

③ 周业营：

- 承担任务：独立完成所有实验，验证、分析了键盘输入的处理程序部分并撰写对应的实验报告，补充部分实验故障和实验改进建议。
- 个人体会：这次的实验在原有的基础上分析键盘输入的原理并实现了键盘中断处理程序，分析了显示器显示的原理，分析 tty 的原理并实现了多个 tty 的控制和切换，还实现了光标的显示和移动、printf 函数功能的实现，总的来说这次实验内容是实现和完善 I/O 子系统。

这次实验让我收获颇多，实验中涉及到的到的键盘输入和显示器显示相关功能的实现与之前在实验中对这些部分的体会相互印证。但涉及到 tty 的基本架构和相关功能的部分对我而言相对陌生，处理多个 tty 同时运行，并发实现输入输出调度的过程十分有趣，进一步加深了我对操作系统内部运作过程的理解。

在跟着教材由浅入深，由易到难的过程中不断发现问题，解决问题，既逐步提升了我对操作系统内部运作原理的理解，又提高了我编程和调试代码的能力。在未来的学习中，我将继续努力深化这些知识，并将它们应用到实践当中。

④ 王浚杰：

- 承担任务：独立完成所有实验，负责实验过程和思考题各前半部分的撰写，完成动手做 1 的实现，并补充了实验问题与故障分析部分。
- 个人体会：本次实验真正的开始体验键盘中断，深入了解键盘输入输出与显示系统的底层机制。通过实验中对扫描码（Scancode）、MakeCode 和 BreakCode 的解析，我明确了键盘输入的底层工作原理。其中，让我印象最深的是处理功能键（如 Shift、Alt、Ctrl）的过程中，左右按键需要分离设计，不能把左右两个键不加区分，例如超级玛丽中左右 Shift 功能被定义为不同的操作。设置键盘缓冲区时，实验用到了大一数据结构课程的环形队列，填补了我对理论知识如何落地实践的空缺。

通过使用 VGA 控制寄存器操作显存，教材代码实现了字符的显示和屏幕滚动。受此启发，在接触到显存的底层工作方式后，我便一发不可收拾，成功在动手做 1 定制属于自己的个性化 TTY，实现光标的方向键移动、字符颜色切换和字符串图案输出的功能。

最后，通过本次实验，我对 TTY 的角色和操作系统整体架构的认识有了进一步的加深。在多用户系统中，TTY 是用户和系统之间的桥梁，其背后的设计不仅涉及到键盘输入和屏幕显示的底层操作，还需要考虑如何协调多个用户任务的输入输出。这次动手实验不仅提升了我对操作系统底层机制的理解，也让我体验到操作系统开发的强大魅力和无限创意空间。