

## 实验文档使用说明

目前前两个实验属于我们已经提供了完整的可实现的代码，并将其设计为实验，

实验一二应当是按照顺序进行实验的（随着对 os 的理解加深）

我们的 os 因为解耦性与模块的分离比较好，所以也可以通过把其中的部分算法挖掉一部分，来让学生填充展开实验。

实验三属于我们也没有实现的一些功能。可以由学生自己来补充。

最后的附录是由我们提供的 x86 架构下的一些常用端口

### 目录

实验一-----	p2-p6
实验二-----	p7-p16
实验三-----	p16
附录-----	p17-19

## 实验一：基于 NASM 的字符与颜色屏幕填充实验

---

### 使用方式:

建议可以让学生在我的分支代码（也就是只实现 bios 启动内核加载的代码）运行，调试，对底层代码有一定的了解与认知。做为第一个小实验。

难度：入门

---

### 实验背景

在操作系统底层开发中，直接操作显存是基础知识。通过显存操作，能够控制屏幕输出的内容、颜色以及字符显示的位置。方便学生快速上手并且有及时反馈的成就感，提升学习兴趣。

### 实验目的

1. **掌握显存操作原理：** 通过直接操作 VGA 显存，理解显存的内存布局。
  2. **控制屏幕输出：** 学习如何使用汇编代码动态修改字符和颜色信息。
  3. **颜色表和位置的使用：** 提供颜色变量表，让学生在实验中体验自定义字符与颜色的过程。
  4. **巩固地址计算与数据操作：** 练习通过计算位置偏移量来操作显存数据。
-

## 实验内容

### 功能要求

- 1. **屏幕字符填充：** 根据输入的字符、颜色和位置填充屏幕指定区域。
- 2. **提供颜色变量表：** 学生通过修改颜色变量表，自定义屏幕字符颜色。
- 3. **实现动态内容修改：** 学生通过修改代码中的字符变量或颜色变量，观察屏幕输出的变化。

---

### 实验实现逻辑

- 1. 定义一个颜色变量表和字符变量：

每个颜色和字符对应屏幕的某一行，支持多行不同颜色。
- 2. 通过地址计算填充屏幕：

根据行列位置动态计算显存偏移量。
- 3. 按照颜色表逐行填充：

根据颜色变量表，为每一行的字符填充不同的颜色。

---

### 实验步骤

#### 1. 项目目录结构

```
os /  
├── bootloader.asm    # 引导加载器  
├── kernel.asm        # 内核代码  
└── Makefile          # 编译和运行控制
```

---

#### 2. 代码实现

##### 2.1 引导加载器代码 (bootloader.asm)

引导加载器保持不变，与源代码一致。其功能是加载内核并跳转执行。

---

## 2.2 内核代码 (kernel.asm)

内核代码在屏幕上填充字符，并支持颜色变量表的配置。

[BITS 16]

[ORG 0x1000] ; 内核加载到 0x1000:0x0000 地址处

; VGA 显存的基础地址

VGA\_MEMORY equ 0xB800

VGA\_COLS equ 80 ; 每行 80 列字符

VGA\_ROWS equ 25 ; 屏幕行数

section .data

; 字符变量表：屏幕显示的字符（每行一个字符）

char\_table db 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'

; 颜色变量表：每行字符的颜色（VGA 颜色代码）

color\_table db 0x0F, 0x1F, 0x2F, 0x3F, 0x4F, 0x5F, 0x6F, 0x7F, 0x8F, 0x9F

section .text

start:

; 设置数据段寄存器指向 VGA 显存

mov ax, VGA\_MEMORY

mov ds, ax

; 初始化变量

mov si, char\_table ; SI 指向字符变量表

mov di, color\_table ; DI 指向颜色变量表

mov cx, 10 ; CX 表示行数（填充 10 行）

fill\_screen:

; 计算当前行的显存起始地址

mov bx, VGA\_COLS ; 每行 80 列字符

mov dx, cx ; 行号

sub dx, 10 ; 反向索引（行号从 10 开始到 1）

mul dx ; bx = 行号 \* VGA\_COLS

shl bx, 1 ; 每字符占用 2 字节，乘以 2

add bx, VGA\_MEMORY ; 显存起始地址 + 偏移量

mov es, bx ; 设置 ES 指向当前行的显存

; 填充该行的字符和颜色

```

        mov bx, 0                ; 列计数器
fill_row:
        mov al, [si]             ; 从字符变量表加载字符
        mov ah, [di]             ; 从颜色变量表加载颜色
        mov [es:bx], ax          ; 写入字符和颜色到显存
        add bx, 2                ; 下一个字符位置 (每次加 2)
        cmp bx, VGA_COLS * 2    ; 检查是否填充完整行
        jl fill_row              ; 如果没有, 则继续填充

        ; 移动到下一个字符和颜色
        inc si
        inc di
        loop fill_screen         ; 填充下一行

halt:
        jmp halt                 ; 无限循环

```

---

### 3. Makefile

Makefile 与源代码一致, 用于自动化编译和运行。

ASM = nasm

QEMU = qemu-system-i386

BOOTLOADER = bootloader.asm

KERNEL = kernel.asm

BOOTLOADER\_BIN = bootloader.bin

KERNEL\_BIN = kernel.bin

OS\_IMAGE = os-image.bin

\$(BOOTLOADER\_BIN): \$(BOOTLOADER)

\$(ASM) \$< -f bin -o \$@

\$(KERNEL\_BIN): \$(KERNEL)

\$(ASM) \$< -f bin -o \$@

\$(OS\_IMAGE): \$(BOOTLOADER\_BIN) \$(KERNEL\_BIN)

cat \$^ > \$@

run: \$(OS\_IMAGE)

\$(QEMU) -fda \$(OS\_IMAGE)

clean:

rm -f \$(BOOTLOADER\_BIN) \$(KERNEL\_BIN) \$(OS\_IMAGE)

---

### 4. 实验内容与要求

## 实验要求

### 1. 填充字符：

使用字符变量表中的内容填充屏幕前 10 行。

### 2. 自定义颜色：

修改颜色变量表，给每一行设置不同的颜色。

### 3. 扩展内容：

添加更多字符和颜色到表中，填充更多行内容。

## 实验观察

1. 学生通过修改 `char_table` 和 `color_table` 中的值，观察屏幕输出的变化。

2. 学生通过调整代码中的行数和列数参数，理解显存布局的规则。

---

## 实验原理分析

### 1. VGA 显存布局

VGA 显存从地址 `0xB8000` 开始，每个字符占用 2 字节：

**第 1 字节：** ASCII 字符。

**第 2 字节：** 字符颜色（低 4 位表示前景色，高 4 位表示背景色）。

每行有 80 列字符，每个字符占用 2 字节，因此每行显存占用  $80 \times 2 = 160$  字节。

### 2. 颜色代码

VGA 颜色代码由 4 位背景色和 4 位前景色组成：

颜色表（前景色）：

0x0: 黑色

0x1: 蓝色

0x2: 绿色

0x3: 青色

0x4: 红色

0x5: 洋红色

0x6: 棕色

0x7: 白色

0x8: 灰色

0x9: 浅蓝色

---

## 实验二\*：基于左右方向键实现光标移动功能

---

使用方式:

该实验对于我们的 os 系统的考察就比较全面,需要对键盘中断机制,端口机制,缓冲区的管理,有全面的认知,可作为大作业的形式展开。

难度：中等偏上。

---

实验背景与目的

实验背景

在操作系统开发中,键盘输入与屏幕显示是基本的交互方式之一。在更高层的终端系统中,支持用户通过方向键移动光标,编辑文本是非常常见的功能。本实验通过实现光标的左右移动功能,让学生进一步理解：

1. 键盘输入的中断处理机制。
2. 显存的操作与动态更新。
3. 如何管理文本缓冲区与光标位置。

实验目的

1. 学习键盘中断的基本处理流程,包括扩展扫描码的处理。
  2. 掌握显存光标操作的实现原理。
  3. 理解字符串缓冲区的操作,包括插入、删除字符,以及动态修改光标位置。
  4. 深入实践显示与输入交互机制,强化对操作系统输入输出机制的理解。
- 

实验内容

功能要求

1. 光标移动：

用户按下左右方向键时，光标可以在屏幕上向左或向右移动。

当光标移动超出文本范围或屏幕范围时，移动无效。

2. 文字输入与编辑：

用户可以在光标位置插入文字。

用户按下退格键时，删除光标左侧的字符并更新屏幕显示。

3. 键盘中断扩展：

处理键盘的扩展扫描码（如左右方向键）。

---

实验步骤

1. 项目目录结构

os/

——	bootloader.asm	# 引导加载器
——	kernel.asm	# 内核代码
——	drivers/keyboard.c	# 键盘驱动（包含光标移动功能）
——	drivers/display.c	# 显示模块（包含光标控制）
——	kernel/util.c	# 工具函数
——	Makefile	# 编译和运行控制

---

2. 实现逻辑

2.1 键盘中断处理逻辑

1. 捕获方向键的扫描码：



左方向键的扩展扫描码为 0xE0 后接 0x4B。

右方向键的扩展扫描码为 0xE0 后接 0x4D。

## 2. 更新光标位置：

根据当前光标位置，更新其在屏幕上的偏移量。

限制光标移动范围，确保不会越界。

---

## 2.2 光标移动与缓冲区操作

### 1. 光标移动：

在屏幕上通过 `CURRENT_OFFSET` 管理光标位置。

更新光标时，重新计算显存地址并调整光标寄存器。

### 2. 字符串缓冲区管理：在光标移动后，确保光标始终与缓冲区位置同步。

提供操作字符串的辅助函数：

插入字符：在当前光标位置插入字符。

删除字符：删除光标左侧的字符。

---

## 3. 代码实现

### 3.1 键盘驱动代码 (keyboard.c)

```
#include "keyboard.h"
```

```
#include "ports.h"
```

```
#include "../cpu/isr.h"
```

```
#include "display.h"
```

```
#include "../kernel/util.h"
```

```
#include <stdint.h>
```

```
#include <stdbool.h>
```

```
#include "../kernel/kernel.h"
```

```
// 扩展扫描码
```

```
#define EXTENDED_LEFT_ARROW 0x4B
```

```
#define EXTENDED_RIGHT_ARROW 0x4D
```

```
// 当前输入位置
```

```
static int INPUT_POS = 0;
```

```
// 键盘回调函数
```

```
static void keyboard_callback(registers_t *regs) {
```

```
    uint16_t scancode = port_byte_in(0x60); // 获取扫描码
```

```
    uint32_t extended_scancode;
```

```
    bool is_extended = false;
```

```
    // 检测扩展扫描码
```

```
    if (scancode == 0xE0) {
```

```
        is_extended = true;
```

```
        extended_scancode = port_byte_in(0x60); // 读取扩展扫描码
```

```
    }
```

```

// 处理扩展扫描码

if (is_extended) {

    if (extended_scancode == EXTENDED_LEFT_ARROW) {

        // 左方向键

        if (move_cursor_left(key_buffer)) {

            set_input_pos(INPUT_POS - 1);

        }

    } else if (extended_scancode == EXTENDED_RIGHT_ARROW) {

        // 右方向键

        if (move_cursor_right(key_buffer)) {

            set_input_pos(INPUT_POS + 1);

        }

    }

} else {

    // 处理普通按键（省略，与之前一致）

}

}

// 初始化键盘

void init_keyboard() {

    register_interrupt_handler(IRQ1, keyboard_callback);

}

```

```
// 设置输入位置

void set_input_pos(int pos) {

    INPUT_POS = pos;

}
```

---

### 3.2 显示模块代码 (display.c)

```
#include "display.h"

#include "ports.h"

#include "../kernel/util.h"

static int CURRENT_OFFSET = 0;

// 左移光标

bool move_cursor_left(char* key_buffer) {

    if (CURRENT_OFFSET > 0) {

        CURRENT_OFFSET -= 2; // 每个字符占 2 字节

        set_cursor(CURRENT_OFFSET);

        return true;

    }

    return false;

}
```

// 右移光标

```
bool move_cursor_right(char* key_buffer) {  
    if (CURRENT_OFFSET < string_length(key_buffer) * 2) {  
        CURRENT_OFFSET += 2; // 每个字符占 2 字节  
        set_cursor(CURRENT_OFFSET);  
        return true;  
    }  
    return false;  
}
```

// 设置光标

```
void set_cursor(int offset) {  
    offset /= 2; // 将字节偏移量转换为字符偏移量  
    port_byte_out(REG_SCREEN_CTRL, 14);  
    port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset >> 8));  
    port_byte_out(REG_SCREEN_CTRL, 15);  
    port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset & 0xff));  
}
```

---

### 3.3 Makefile

Makefile 用于编译和运行实验。

makefile

复制代码

# 汇编器与编译器

ASM = nasm

CC = i686-elf-gcc

# 项目文件

KERNEL = kernel.o

DISPLAY = drivers/display.o

KEYBOARD = drivers/keyboard.o

UTIL = kernel/util.o

OS\_IMAGE = os-image.bin

BOOTLOADER = bootloader.asm

# 编译规则

all: \$(OS\_IMAGE)

\$(OS\_IMAGE): \$(BOOTLOADER) \$(KERNEL) \$(DISPLAY) \$(KEYBOARD) \$(UTIL)

cat bootloader.bin kernel.bin > \$(OS\_IMAGE)

%.o: %.c

\$(CC) -c \$< -o \$@

```
%bin: %.asm
```

```
$(ASM) $< -f bin -o $@
```

```
run: $(OS_IMAGE)
```

```
qemu-system-i386 -fda $(OS_IMAGE)
```

```
clean:
```

```
rm -f *.bin *.o $(OS_IMAGE)
```

---

## 4. 实验要求与扩展

### 实验要求

#### 1. 光标移动功能：

按左方向键时，光标应左移一个字符，直到到达行首。

按右方向键时，光标应右移一个字符，直到缓冲区末尾。

#### 2. 文字输入与编辑：

用户可以在光标位置插入字符或删除字符。

### 实验扩展

#### 1. 多行输入支持：

添加换行逻辑，允许用户在多行输入内容。

#### 2. 边界行为：

当光标到达行末或屏幕末尾时，支持滚屏功能。

---

### 实验意义

### 1. 深入理解中断：

通过实现键盘扩展扫描码的解析，理解硬件中断的底层原理。

### 2. 文本缓冲区管理：

掌握动态管理字符串缓冲区与光标位置的方法。

### 3. 输入输出的交互：

实现光标与文本输入的交互逻辑，为未来实现更复杂的终端功能打下基础。

## 实验三：自由拓展

可由学生拓展我们的 os 做为加分项。

包括;文件管理系统的完善(1.我们目前的 os 不支持用户输入字符进行文件管理，只能在代码层面完善。2.多级目录管理等接近现代完善的文件管理)

Tips：可以尝试学习 xv6 的文件管理来模仿并改进。输入字符的功能，则是要对缓冲区改进。

鼠标的接入。(较为复杂，我也没啥思路)

内存管理的实现。

硬盘写入改成用中断实现。

Tips：添加中断向量表里的内容，在 keyboard 里添加额外的绑定事件。



## 附录：OS 常用端口指令汇总

---

### 1. BIOS 视频端口 (0x3D4 和 0x3D5)

#### 用途

用于控制 VGA 文本模式下的光标位置。

#### 相关端口

端口地址	功能
0x3D4	光标控制寄存器选择端口
0x3D5	光标位置数据端口

---

### 2. 键盘输入端口 (0x60 和 0x64)

#### 用途

键盘控制器通过这些端口与 CPU 交互，用于读取按键扫描码和控制键盘状态。

#### 相关端口

端口地址	功能
0x60	数据端口，用于读取扫描码或写入数据
0x64	状态端口，用于获取键盘状态

---

### 3. 硬盘（ATA）控制端口

#### 用途

ATA（Advanced Technology Attachment）端口用于与硬盘交互，实现扇区的读取与写入。

#### 相关端口

端口地址	功能
0x1F0	数据寄存器（读/写扇区数据）
0x1F1	错误寄存器（读）或特性寄存器（写）
0x1F2	扇区计数寄存器
0x1F3	LBA 地址（低 8 位）
0x1F4	LBA 地址（中间 8 位）
0x1F5	LBA 地址（高 8 位）
0x1F6	驱动器选择寄存器
0x1F7	命令寄存器（写）或状态寄存器（读）

---

### 4. 定时器（PIT）控制端口

#### 用途

通过定时器控制端口，设置系统时钟频率或实现简单的延

时功能。

相关端口

端口地址

功能

0x40

通道 0 数据寄存器

0x41

通道 1 数据寄存器（通常未使用）

0x42

通道 2 数据寄存器

0x43

模式控制寄存器

---

## 5. 常用工具函数操作的端口

用途

这些端口用于与硬件设备交互，是对 I/O 设备的基本操作。

相关端口

操作类型

功能

port\_byte\_in 从指定端口读取一个字节

port\_byte\_out 向指定端口写入一个字节

port\_word\_in 从指定端口读取一个字

port\_word\_out 向指定端口写入一个字