

实验 2： Multiboot2myMain

目录

- 软件结构概述
- 主流程及其实现
- 主要功能模块及其实现
- 源代码说明(目录组织、 Makefile组织)
- 代码布局说明(地址空间)
- 编译过程说明
- 运行和运行结果说明
- 遇到的问题和解决方案说明

原理说明

- uart端口输入输出

COM1端口的基地址是0x3F8，基地址同时也是数据地址。0x3F8+5是该端口的Line Status Register，该寄存器有8位，对应8个标志，本实验只用到了它的第0位(Data Ready)，用于判断输入该端口的数据是否可用。

输出时，直接向基地址写出数据即可；输入时，需要循环判断Line Status Register的Data Ready位是否为1，如果是1才读入基地址的数据并返回。

- 光标位置的读写

光标位置是16位无符号整数，高8位存在14号寄存器，低8位存在15号寄存器。这些寄存器需要通过端口映射访问，通过设置索引端口的数据(0x3d4)可以选择要访问的寄存器。将0x3d4端口设为0xe访问的是14号寄存器，即光标位置的高8位；同理，设置为0xf就可以访问光标的低8位。设置完索引端口，即可通过数据端口(0x3d5)读写相应内容。

- VGA输出

VGA显存的起始地址是0xB8000，因此从该地址开始写入要显示的文本。VGA显存约定每个字符需要两个字节，一个字节用于存放ASCII码，另外一个用于存放该字符的显示样式。具体格式如下表：

Attribute	Charactor
15~8	7~0

其中字符样式Attribute还有如下格式：

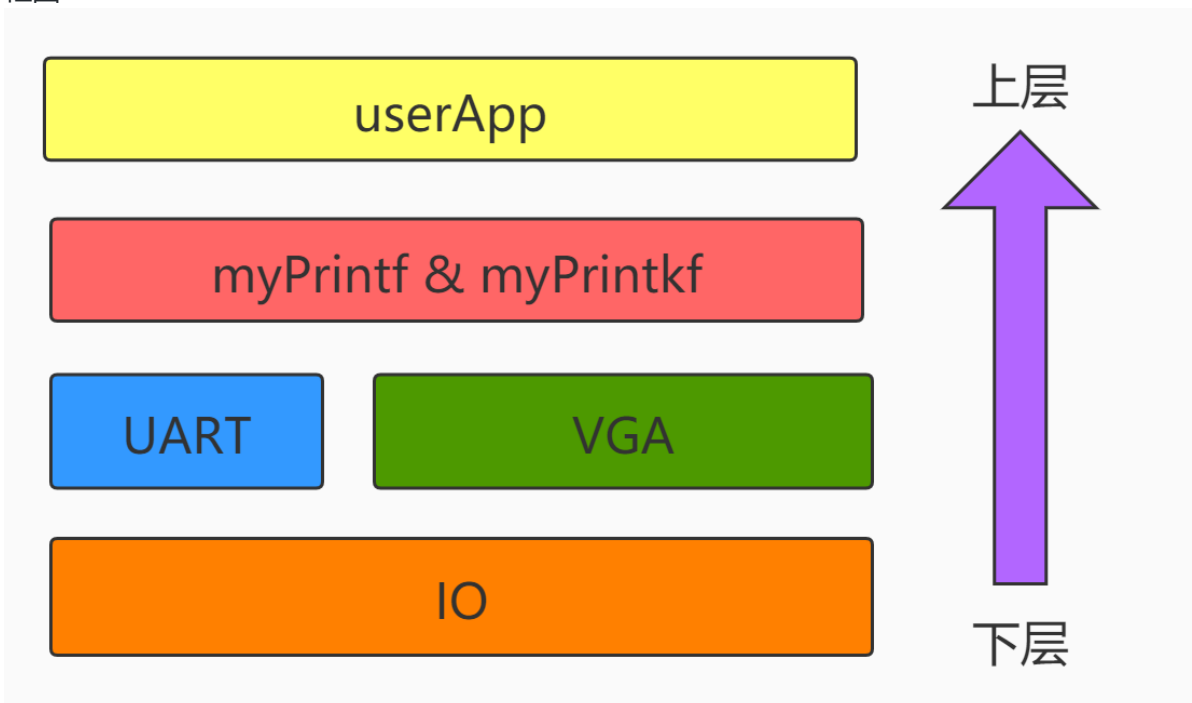
Blink	Background Color	Foreground Color
7	6~4	3~0

软件结构概述

- 概述

软件结构自底向上是端口IO模块、UART和VGA模块、格式化输出模块以及用户模块。UART和VGA模块是基于端口IO实现的，在UART和VGA实现了字符串输出的基础上实现格式化输出模块。最顶层的用户模块调用格式化输出模块的接口即可通过UART或VGA输出格式化字符串。

- 框图

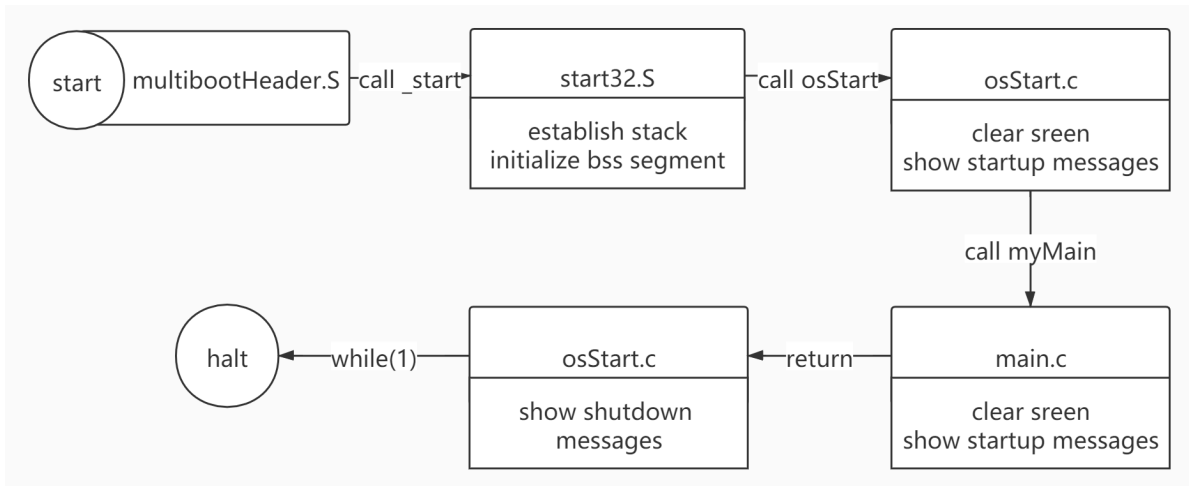


主流程及其实现

- 流程说明

- 1.根据链接描述文件，程序的入口在multibootHeader的start位置。
- 2.multibootHeader调用start32的_start函数，进行栈空间和bss空间的初始化。
- 3.start32调用onStart，执行系统启动时命令。
- 4.osStart调用用户程序入口函数myMain，执行用户命令。
- 5.用户程序执行完毕返回osStart，osStart执行系统关闭时命令，然后停机（死循环）。

- 主流程图



主要功能模块及其实现

- IO模块：读写指定端口的字节数据

```

// read byte from a certain port
unsigned char inb(unsigned short int port_from){
    unsigned char c;
    __asm__ __volatile__ ("inb %w1,%b0":"=a"(c):"Nd" (port_from));
    return c;
}

// write byte to a certain port
void outb (unsigned short int port_to, unsigned char value){
    __asm__ __volatile__ ("outb %b0,%w1::" "a" (value), "Nd" (port_to));
}

```

- uart模块：串口输入输出模块，用于调试

```

extern unsigned char inb(unsigned short int port_from);
extern void outb (unsigned short int port_to, unsigned char value);

#define uart_base 0x3F8

void uart_put_char(unsigned char c){
    outb(uart_base, c);
}

unsigned char uart_get_char(void){
    while(!(inb(uart_base+5)&1)); // loop until data is ready
    return inb(uart_base);
}

void uart_put_chars(char *str){
    for(int i=0; str[i]!='\0'; i++){
        outb(uart_base, str[i]);
    }
}

```

```

    }
}

```

- vga模块：屏幕文字输出模块，用于调试

```

extern void disable_interrupt(void);
extern void enable_interrupt(void);

extern void uart_put_chars(unsigned char* str);
extern unsigned char uart_get_char(void);
extern unsigned char inb(unsigned short int port_from);
extern void outb(unsigned short int port_to, unsigned char value);

#define vga_base 0xB8000
#define vga_size 0x1000
#define index_port 0x3d4
#define data_port 0x3d5
#define buffer_width 80
#define buffer_height 24

int addr = vga_base; // video memory address
int cursor_pos; // cursor position

void set_cursor_pos(int pos) {
    outb(index_port, 0xe);
    outb(data_port, (pos >> 8) & 0xff); // set high 8 bits
    outb(index_port, 0xf);
    outb(data_port, pos & 0xff); // set low 8 bits
}

//put char with raw data
void append_char_raw(char c, int color) {
    if (cursor_pos / buffer_width == buffer_height) { // need to roll screen
        // copy data to previous lines
        addr = vga_base;
        for (int i = 0; i < buffer_height - 1; i++) {
            for (int j = 0; j < buffer_width / 2; j++) {
                __asm__ __volatile__ ("movl (%0),%%ebx"::"a"(addr + buffer_width *
2));
                __asm__ __volatile__ ("movl %%ebx, (%0)"::"a"(addr));
                addr += 4;
            }
        }
        // erase data in the last line
        for (int i = 0; i < buffer_width / 2; i++) {
            __asm__ __volatile__ ("movl $0, (%0)"::"a"(addr));
            addr += 4;
        }
        cursor_pos -= buffer_width;
    }
    // put char to cursor position
    addr = vga_base + cursor_pos * 2;
    __asm__ __volatile__ ("movb %0, (%1)"::"a"(c), "b"(addr++));
    __asm__ __volatile__ ("movb %0, (%1)"::"a"(color), "b"(addr++));
}

```

```

    set_cursor_pos(++cursor_pos); // set new cursor position
}

//put char with processed data
void append_char(char c, int color) {
    int tmp = buffer_width - cursor_pos % buffer_width;
    switch (c) {
        case '\t': // \t = put 4 space
            for (int i = 0; i < 4; i++) {
                append_char_raw(' ', color);
            }
            break;
        case '\n': // \n = fill the rest of current line with space
            for (int i = 0; i < tmp; i++) {
                append_char_raw(0x20, 0);
            }
            break;
        default: // directly put char
            append_char_raw(c, color);
            break;
    }
}

void clear_screen(void) {
    for (int i = 0; i < vga_size; i++) {
        __asm__ __volatile__ ("movw $0xf20, (%0)":"a"(addr));
        addr += 2;
    }

    set_cursor_pos(0); // initialize cursor position
    cursor_pos = 0;
    addr = vga_base; // initialize data cursor
}

void append2screen(char* str, int color) {
    for (int i = 0; str[i] != 0; i++) {
        append_char(str[i], color);
    }
}

```

- 格式化输出模块,用于调试。其中的vsprintf移植自linux内核。

```

#include <stdarg.h>

extern void append2screen(char *str, int color);
extern void uart_put_chars(char *str);
extern int vsprintf(char *buf, const char *fmt, va_list args);

char kbuf[400];
int myPrintk(int color, const char* format, ...) {
    va_list args;
    int i;
    va_start(args, format);
    //write formatted string to buffer

```

```

    i = vsprintf(kbuf, format, args);
    //release args
    va_end(args);
    //append formatted string to screen
    append2screen(kbuf,color);
    return i;
}

char ubuf[400];
int myPrintf(int color, const char* format, ...) {
    va_list args;
    int i;
    va_start(args, format);
    //write formatted string to buffer
    i = vsprintf(ubuf, format, args);
    //release args
    va_end(args);
    //append formatted string to screen
    append2screen(ubuf,color);
    return i;
}

char buf[400];
int uartPrintf(const char* format, ...){
    va_list args;
    int i;
    va_start(args, format);
    //write formatted string to buffer
    i = vsprintf(buf, format, args);
    //release args
    va_end(args);
    //append formatted string to serial
    uart_put_chars(buf);
    return i;
}

```

源代码说明

- 目录组织

```

__src
|__multibootheader
| |__multibootHeader.S
|__userApp
| |__main.c
|__myOS
| |__osStart.c
| |__i386
| | |__io.c
| | |__io.h
| |__myOS.ld
| |__dev
| | |__uart.c
| | |__vga.c
| |__start32.S

```

```
| | ____printf
| | | ____vsprintf.c
| | | ____myPrintf.c
| ____source2run.sh
```

- Makefile组织

```
____src
| ____Makefile
| ____userApp
| | ____Makefile
| ____myOS
| | ____Makefile
| | ____i386
| | | ____Makefile
| | ____dev
| | | ____Makefile
| | ____printf
| | | ____Makefile
```

代码布局说明

- 链接描述文件

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
    . = 1M;
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

    . = ALIGN(16);
    .data : { *(.data*) }

    . = ALIGN(16);
    .bss :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
    . = ALIGN(512);
}
```

- 代码地址空间

Offset	Field	Macro
1M	.text	
ALIGN(16)	.data	
ALIGN(16)	.bss	__bss_start, _bss_start
		_bss_end
ALIGN(16)		_end

编译过程说明

- Makefile

```
OS_OBJS      = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

output/%.o : %.S
    @mkdir -p $(dir $@)
    @$${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c
    @mkdir -p $(dir $@)
    @$${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

- 说明

根据Makefile分为两步：编译和链接。

第一步，编译汇编代码(*.S)和c代码(*.c)并输出对象文件(*.o)。

第二步，将这些对象文件链接并输出可执行可链接文件(myOS.elf)。

运行和运行结果说明

- 运行

执行命令：`qemu-system-i386 -kernel output/myOS.elf -serial stdio`

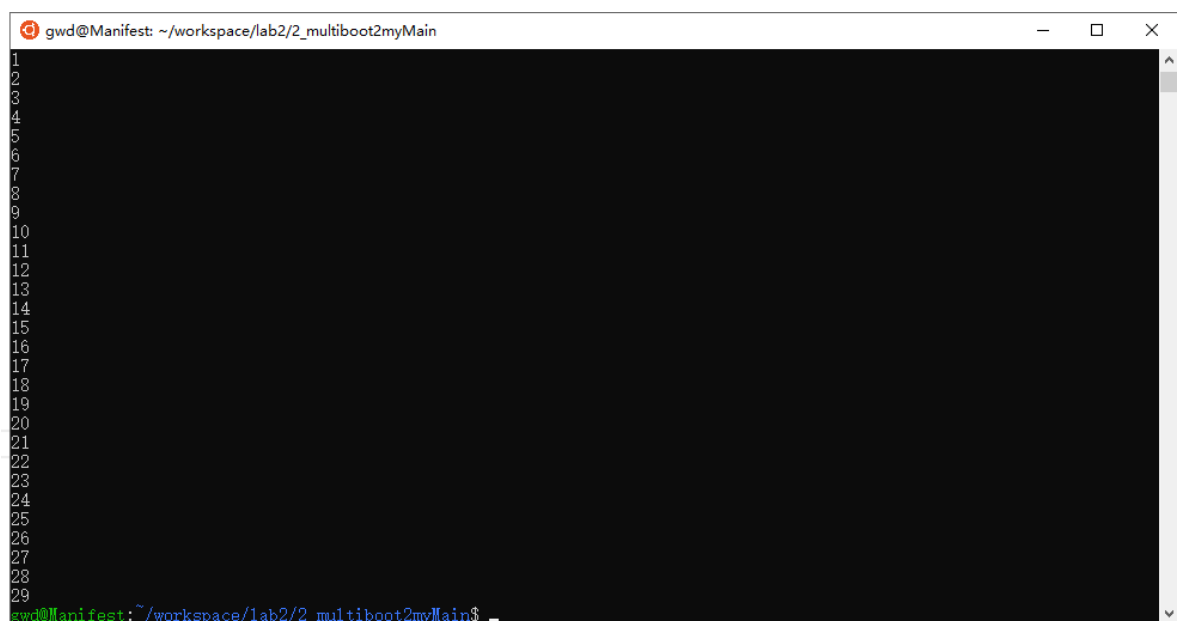
将之前编译链接生成的elf文件，加载到qemu中运行。

- 运行结果



```
QEMU - Press Ctrl-Alt to exit mouse grab
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
STOP RUNNING.....ShutDown
```

可以看到，滚屏输出正常，光标显示正常，程序运行结束。



```
gwd@Manifest: ~/workspace/lab2/2_multiboot2myMain
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
gwd@Manifest:~/workspace/lab2/2_multiboot2myMain$
```

另外，串口输出也正常。

遇到的问题 and 解决方案说明

- uart_get_char

问题描述：直接调用inb读不到uart输入的字符。

解决方案：经过求助师兄，了解到line status register有标志位可以指示通过uart输入的数据 是否已经可用。于是只要等到数据可用后再返回 读入的数据就可以了。

- 光标设置问题

问题描述：逐字符输出设置光标时出现光标跳跃。

解决方案：通过查阅资料，得知光标寄存器数据格式约定。光标位置是16位无符号整数，高8位存在14号寄存器，低8位存在15号寄存器。而不是起初认为的14号存行，15号存列。