实验报告

【实验题目】

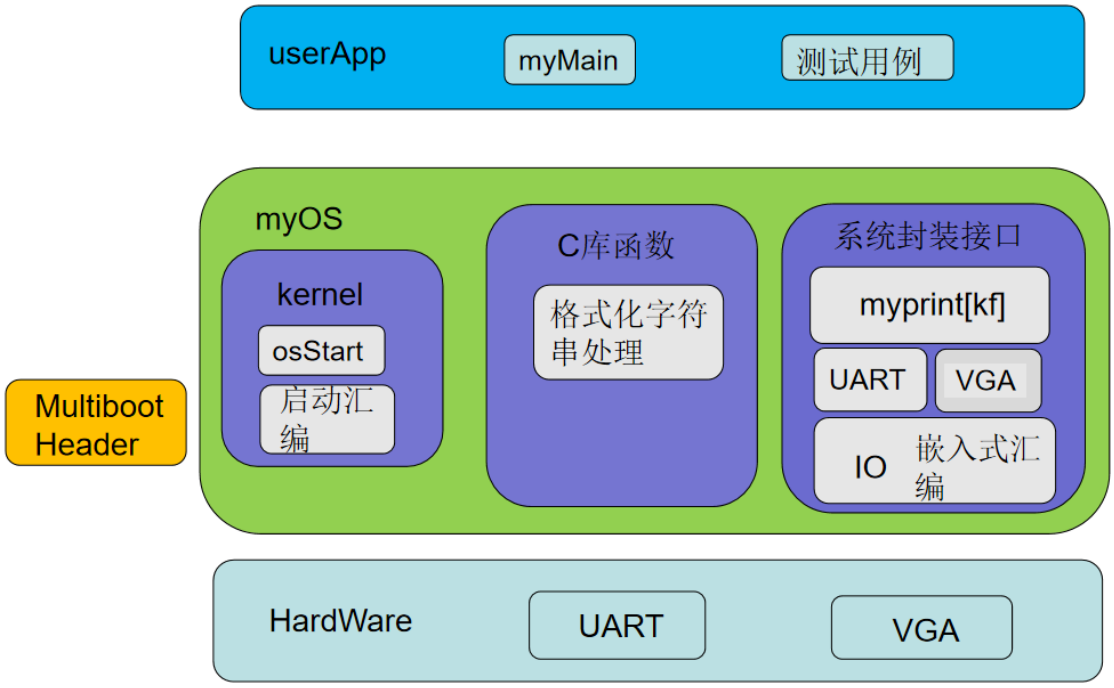
Multiboot2myMain

【实验要求】

```
1. 在源代码的语言层面，完成从汇编语言到C语言的衔接
2. 在功能上，实现清屏、格式化输入输出，设备包括VGA和串口，接口符合要求
3. 在软件层次和结构上，完成multiboot_header、myOS和userApp的划分，体现在文件目录组织和Makefile组织上
4. 采用自定义测试用例和用户（助教）测试用例相结合的方式进行验收
5. 提供脚本完成编译和执行
```

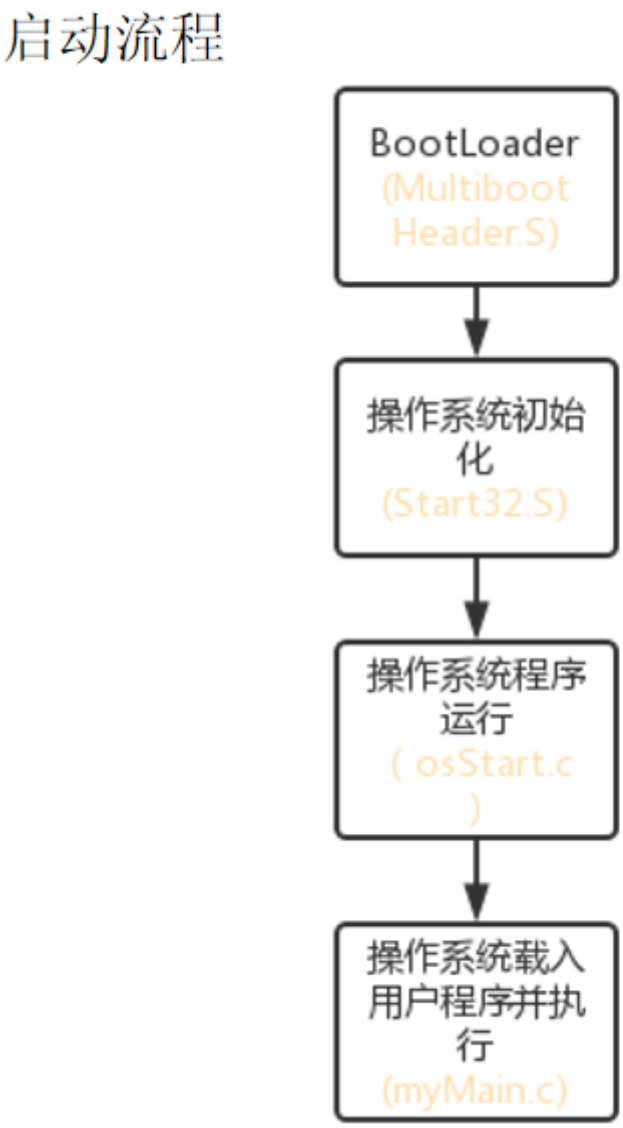
【实验原理】

- 软件架构和功能




```
软件整个生命周期涉及三个层次：硬件层，操作系统层，用户应用层，其功能分别如下：
    Hardware层：  提供基于UART和VGA的IO功能，另外提供程序执行的其他硬件资源
    myOS层：    本次实验中的操作系统层较为简单，仅实现栈的建立以及BSS段的0初始化，为C程序的执行提供运行环境，同时为C程序提供有关UART和VGA的IO接口函数库(myPrint[kf])
    UserApp层：    使用操作系统提供的IO接口，实现彩色格式化输入输出,完成测试用例
```

- 软件执行流程



- 主要功能模块及其实现

    1. vga输出模块

```
#include "io.h"
#include "vga.h"

#define VGA_BASE 0xB8000 // vga 显存起始地址
#define VGA_SCREEN_WIDTH 80 // vga 屏幕宽度（可容纳字符数）
#define VGA_SCREEN_HEIGHT 25 // vga 屏幕高度

#define CURSOR_LINE_REG 0xE // 行号寄存器
#define CURSOR_COL_REG 0xF // 列号寄存器
#define CURSOR_INDEX_PORT 0x3D4 // 光标行列索引端口号
#define CURSOR_DATA_PORT 0x3D5 // 光标数据端口号

/* ========= 以下函数仅供参考，可以根据自己的需求进行修改，甚至删除 ========= */

/* 将光标设定到特定位置
 * 提示: 使用 outb */
void set_cursor_pos(cursorPos pos) {
    /* todo */
```

```
    outb(CURSOR_INDEX_PORT,CURSOR_LINE_REG );
    outb(CURSOR_DATA_PORT ,pos.line);
    outb(CURSOR_INDEX_PORT,CURSOR_COL_REG );
    outb(CURSOR_DATA_PORT,pos.col);

}

/* 获取光标当前所在位置
 * 提示：使用 inb */
cursorPos get_cursor_pos(void) {
    /* todo */
    cursorPos pos;
    outb(CURSOR_INDEX_PORT,CURSOR_LINE_REG);
    pos.line = inb(CURSOR_DATA_PORT);
    outb(CURSOR_INDEX_PORT,CURSOR_COL_REG);
    pos.col = inb(CURSOR_DATA_PORT);
    return pos;
}

/* 滚屏，vga 屏幕满时使用。丢弃第一行内容，将剩余行整体向上滚动一行
 * 提示：使用指针修改显存 */
void scroll_screen(void) {
    /* todo */
    unsigned char* ptr = VGA_BASE;
    int i=0;
    for(;i<80*24;i+=1){
        *(ptr+2*i) = *(ptr+2*i+80*2);
        *(ptr+2*i+1) = *(ptr+2*i+80*2+1);
    }
     for(;i<80*25;i+=1){
        *(ptr+2*i) = 0;
        *(ptr+2*i+1) = 0x2;
    }

}

/* 向 vga 的特定光标位置 pos 输出一个字符
 * 提示：使用指针修改显存 */

// 只处理常规输出
void put_char2pos(unsigned char c, int color, cursorPos pos) {
    /* todo */
    unsigned char* ptr = VGA_BASE + pos.line*80*2+pos.col*2;
    *ptr = c;
    *(ptr+1) = color;
}

/* ========= 以下函数接口禁止修改 ========= */

/* 清除屏幕上所有字符，并将光标位置重置到顶格
 * 提示：使用指针修改显存 */
void clear_screen(void) {
    /* todo */
    unsigned char* ptr = VGA_BASE;
    cursorPos pos = {0,0};
    *(ptr) = 0;
    *(ptr+1) = 0x2;
    for(int i=1;i<80*25;i+=1){
```

```
        *(ptr+2*i) = 0;
        *(ptr+2*i+1) = 0;
    }
    set_cursor_pos(pos);
}

/* 向 vga 的当前光标位置输出一个字符串，并移动光标位置到串末尾字符的下一位
 * 如果超出了屏幕范围，则需要滚屏
 * 需要能够处理转义字符 \n */
void append2screen(char *str, int color) {
    /* todo */
    cursorPos pos = get_cursor_pos();
    char ch;
    for(int i=0;str[i]!=0;i++){
        ch = str[i];

        if(ch == '\n') {
            if(pos.line == 24) {
                scroll_screen();
                pos.line = 24;
                pos.col = 0;
            }
            else {
                pos.col = 0;
                pos.line += 1;
            }

        }
        else {
            put_char2pos(ch,color,pos);
            if(pos.col == 79){
                if(pos.line == 24) {
                    scroll_screen();
                }
                else {
                     pos.line += 1;
                     pos.col = 0;
                }
            }
            else pos.col += 1;
        }

    }
    set_cursor_pos(pos);

}
```

- get_cursor_pos与set_cursor_pos

  预先定义结构体cursorPos,其中包含两个变量x,y分别代表光标的横纵坐标,通过myOS提供的底层硬件IO接口读取，写入光标的行号，列号寄存器实现

- scroll_screen

> 通过C语言的指针再借助myOS提供的底层硬件IO接口，完成vga显示数据的地址前移，达到丢弃第一行内容，将剩余行整体向上滚动一行的效果，最后再将光标设置在最后一行的首位

- put_char2pos

> 直接使用C指针修改vga显示位置的数据(字符+颜色)即可,另外此函数只完成输出功能，不处理例外或光标的调整功能

- clear_screen

> 直接使用C指针修改vga显示位置的所有数据(统一将字符数据修改为0，颜色数据修改为0x2),另外此函数只实现清屏功能，不处理例外或光标的调整功能

- append2screen

> 将字符串逐个字符取出调用put_char2pos将其输出到vga显示屏，另外此函数根据当前光标位置与当前输出的字符完成特殊字符的效果(如换行),滚屏，清屏，光标位置的调整

2. 串口输出模块

```
#include "io.h"
#include "uart.h"

#define UART_PORT 0x3F8 // 串口端口号

/* 向串口输出一个字符
 * 使用封装好的 outb 函数 */
void uart_put_char(unsigned char ch) {
    /* todo */
    outb(UART_PORT,ch);
}

/* 向串口输出一个字符串
 * 此函数接口禁止修改 */
void uart_put_chars(char *str) {
    /* todo */
    for(int i=0;str[i]!=0;i++){
        outb(UART_PORT,str[i]);
    }

}
```

> 直接调用底层硬件IO接口实现uart的输入输出即可

3. 可变长参数格式化输出模块

```
#include "vsprintf.h"

/* 将 printf 格式串转化成可直接输出的字符串，存储在 buf 中
 * 可以移植或自编，自编至少需要支持 %d （建议移植）
 */
typedef enum bool{false,true} bool ;
```

```
int vsprintf(char *buf, const char *fmt, va_list args) {
    char chs[2];
    int buf_index =0;
    int fmt_index = 0;

    Buffer buffer;
    bool new = true;

    for(;fmt[fmt_index];fmt_index++){
        if(new) {
            chs[0] = fmt[fmt_index];
            new = false;
        }
        else {
            if(chs[0]=='%') {
                if(fmt[fmt_index]=='d') {
                    int num = va_arg(args,int);
                        buffer.char_num = 0;
                        if(num == 0){
                            buffer.char_buf[buffer.char_num] = '0';
                            buffer.char_num++;
                        }
                        else {
                            int rem;
                            while(num){
                                rem = num % 10;
                            num = num /10;
                            buffer.char_buf[buffer.char_num] = rem+48;
                                buffer.char_num++;
                            }
                        }
                        for(int i=0;i<buffer.char_num;i++){
                            buf[buf_index] = buffer.char_buf[buffer.char_num-
i-1];

                            buf_index++;
                            }

                }
                else {
                    chs[1] = fmt[fmt_index];
                        buf[buf_index] = chs[0];
                        buf[buf_index+1] = chs[1];
                        buf_index += 2;
                }
                new = true;
            }
            else if(fmt[fmt_index]=='%') {
                buf[buf_index] = chs[0];
                buf_index += 1;
                chs[0] = '%';
            }
            else {
                chs[1] = fmt[fmt_index];
                    buf[buf_index] = chs[0];
                    buf[buf_index+1] = chs[1];
                    buf_index += 2;
                    new = true;
```

```
            }


        }
    }
    if(!new){
        buf[buf_index] = chs[0];
        buf_index += 1;
    }
    buf[buf_index] = 0;
    if (buf_index) return buf_index;
    else return -1;
}
```

将格式化字符串按两个划分，如果其中一个为'%'则另行处理，否则直接写入输出字符串，如果其中一个为'%'判断其后紧跟着的字符是否为d，如果满足，使用C库提供的可变长参数的获取方法，将数字还原成字符串插入到结果字符串中即可，本函数只实现了'%d'功能

4. myPrint[fk]模块

```
#include "vga.h"
#include "uart.h"
#include "vsprintf.h"

/* 内核 print 函数
 * 调用已经完成的 vga 和 串口 输出接口，补全此函数
 * 禁止修改此函数接口 */
char kBuf[400];
int myPrintk(int color, const char *format, ...) {
    va_list args;

    va_start(args, format);
    int cnt = vsprintf(kBuf, format, args);
    va_end(args);

    /* todo */
    append2screen(kBuf, color);

    return cnt;
}

/* 用户 print 函数
 * 调用已经完成的 vga 和 串口 输出接口，补全此函数
 * 禁止修改此函数接口 */
char uBuf[400];
int myPrintf(int color, const char *format, ...) {
    va_list args;

    va_start(args, format);
    int cnt = vsprintf(uBuf, format, args);
    va_end(args);

    /* todo */
    append2screen(uBuf, color);
    return cnt;
}
```

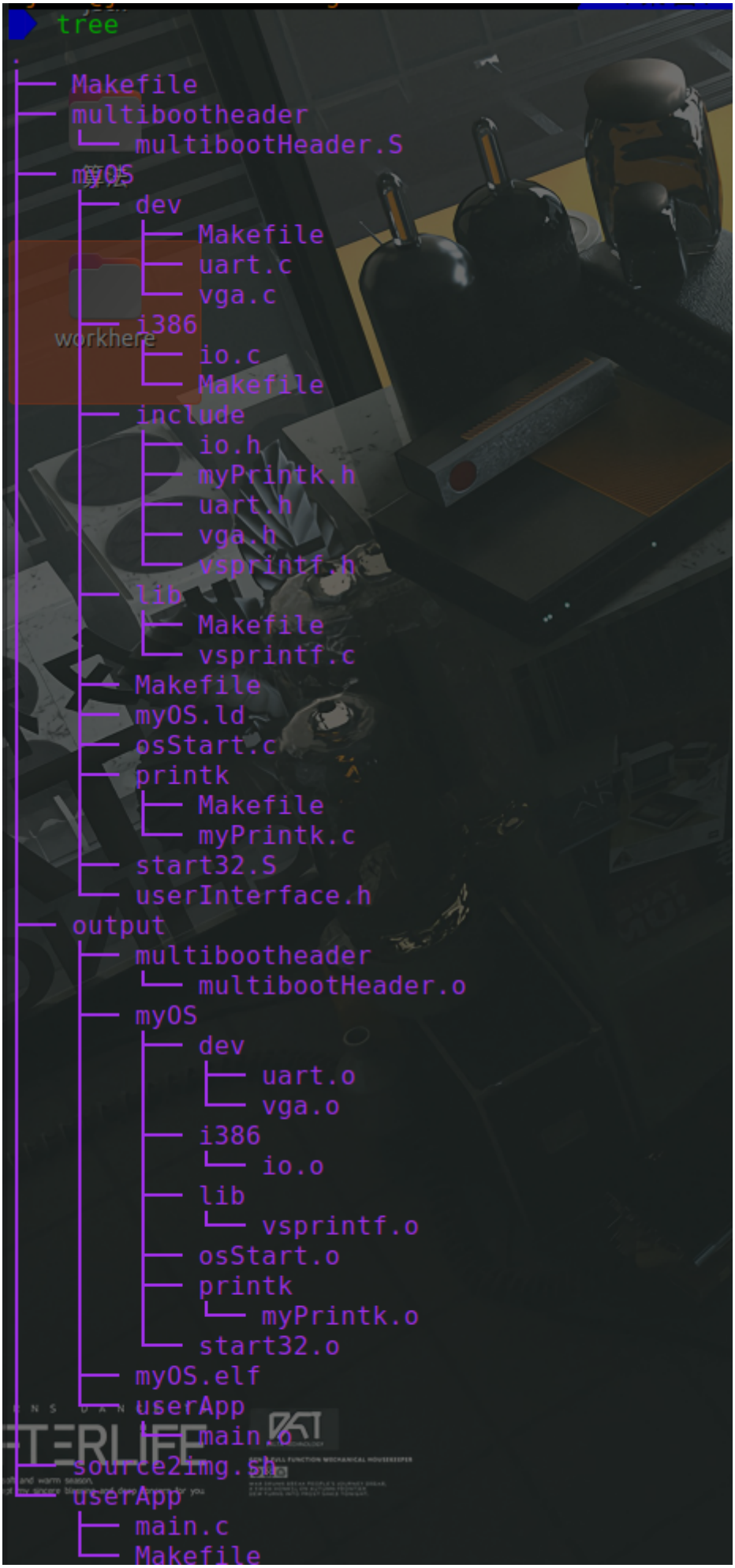
调用vsprintf模块与vga,uart输出模块完成功能

5. 用户程序与测试模块

```
#include "userInterface.h"

  void myMain(void) {

      int i;
      for (i = 1; i < 30; i++)
          myPrintf(i, "%d\n", i);

      myPrintk(0x7, "PB20061338_kezhiwei\n"); // 你的学号、姓名

      return;
  }
```

调用myPrintk函数完成测试要求

- 代码组织及其实现
  - 目录组织



代码按模块主要分为四部分：

1. multibootheader

此模块提供multibootHeader段的代码，使得bootloader成功将操作系统载入内存

2. myOS

此模块提供操作系统的代码，包括

```
start32.S 初始化C程序的运行环境的汇编级程序
dev       提供硬件层UART,VGA的使用封装接口
i386      提供基于i386架构的底层硬件IO的接口
include   C程序的头文件
lib       C程序的库函数，本次实验只有自己实现的vsprintf函数
printk    借助库函数自行实现的printf与printk的C语言函数
```

3. userApp

此模块存放用户程序，本次实验只含main.c

4. output

此模块存放含操作系统及用户程序在内的所有程序根据Makefile,ld文件编译链接后的生成的可执行文件，结构与myOS类似，不再赘述

- Makefile组织

```
.
├── MULTI_BOOT_HEADER
|   └── output/multibootheader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
    |   ├── output/myOS/start32.o
    |   ├── output/myOS/osStart.o
    |   ├── DEV_OBJS
    |   |   ├── output/myOS/dev/uart.o
    |   |   └── output/myOS/dev/vga.o
    |   ├── I386_OBJS
    |   |   └── output/myOS/i386/io.o
    |   ├── PRINTK_OBJS
    |   |   └── output/myOS/printk/myPrintk.o
    |   └── LIB_OBJS
    |       └── output/myOS/lib/vsprintf.o
    └── USER_APP_OBJS
        └── output/userApp/main.o
```

- 代码布局说明

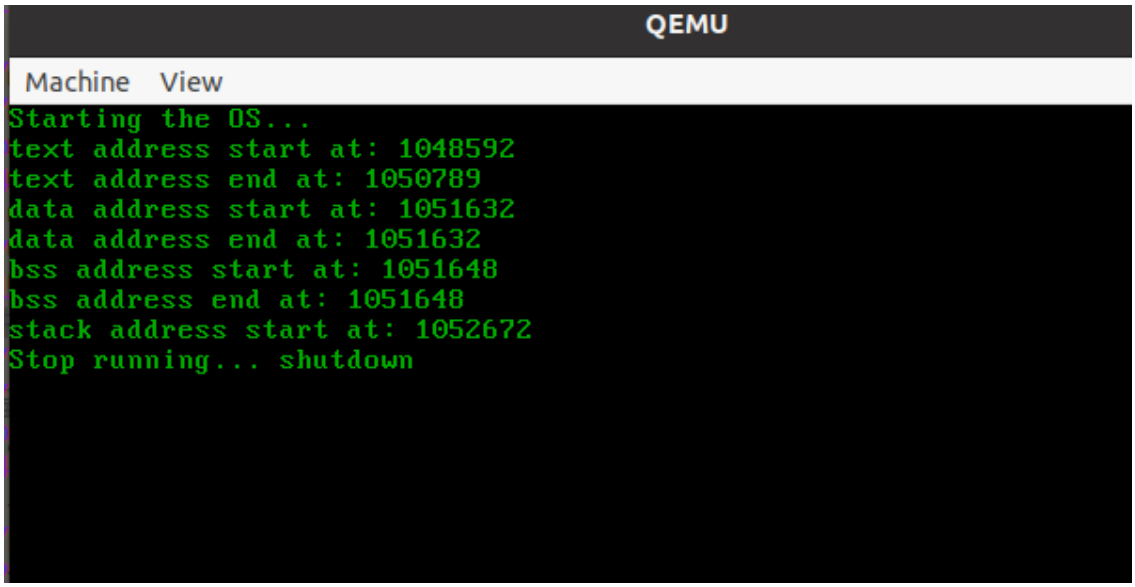
  借助C程序可查看各段的起止位置

```
#include "vga.h"
#include "myPrintk.h"
```

```
extern unsigned long __text_start;
extern unsigned long __text_end;
extern unsigned long __data_start;
extern unsigned long __data_end;
extern unsigned long __bss_start;
extern unsigned long __bss_end;
extern unsigned long __stack_start;
/* 此文件无需修改 */

// 用户程序入口
void myMain(void);

void osStart(void) {
    clear_screen();
    myPrintk(0x2, "Starting the OS...\n");
    //myMain();
    myPrintk(0x2,"text address start at: %d\n",(unsigned long)&__text_start);
    myPrintk(0x2,"text address end at: %d\n",(unsigned long)&__text_end);
    myPrintk(0x2,"data address start at: %d\n",(unsigned long)&__data_start);
    myPrintk(0x2,"data address end at: %d\n",(unsigned long)&__data_end);

    myPrintk(0x2,"bss address start at: %d\n",(unsigned long)&__bss_start);
    myPrintk(0x2,"bss address end at: %d\n",(unsigned long)&__bss_end);
    myPrintk(0x2,"stack address start at: %d\n",(unsigned
long)&__stack_start);

    myPrintk(0x2, "Stop running... shutdown\n");
    while(1);
}
```

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

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

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

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

```
    . = ALIGN(16);
    _end = .;
    . = ALIGN(512);
    __stack_start = .;
}
```



Section	Offset(Base = 0)
.multiboot_header	0x0
.text	0x100010~0x1008a5
.data	0x100bf0~0x100bf0
.bss	0x100c00~0x100c00
stack	0x101000~0x105000

【实验过程】

- 编译过程及运行过程说明

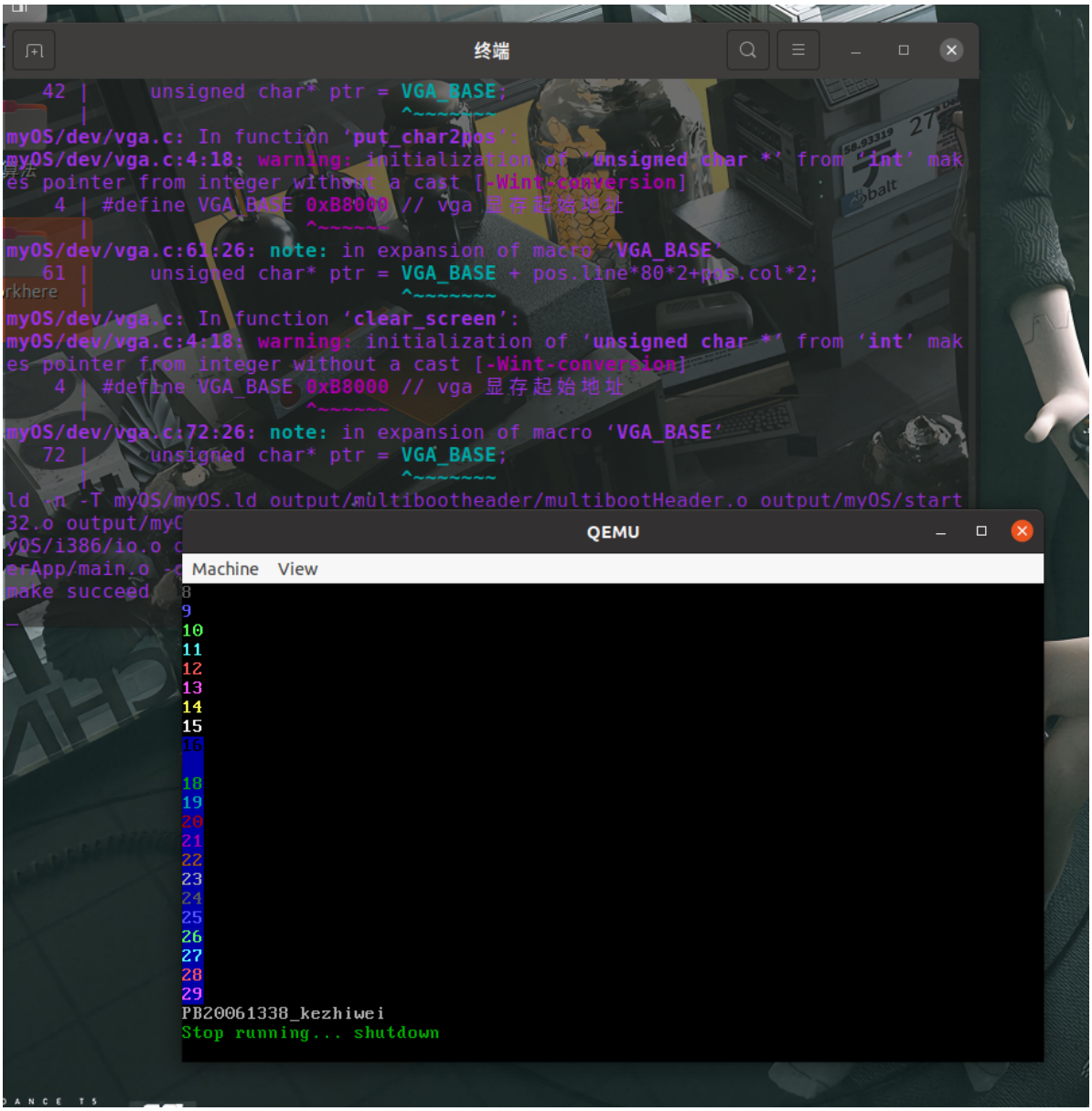
> 直接执行脚本文件source2img.h实现编译链接及运行

脚本文件及外层Makefile如下：

```
#!/bin/bash
make clean

make

if [ $? -ne 0 ]; then
    echo "make failed"
else
    echo "make succeed"
    qemu-system-i386 -kernel output/myOS.elf -serial stdio
fi
```

```
SRC_RT = $(shell pwd)
```

```
CROSS_COMPILE=
ASM_FLAGS = -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
C_FLAGS = -m32 -fno-stack-protector -fno-builtin -g

.PHONY: all
all: output/myOS.elf

MULTI_BOOT_HEADER = output/multibootheader/multibootHeader.o
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/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 -I myOS/include -I myOS ${C_FLAGS} -c -o $@ $<

clean:
    rm -rf output
```



【问题与解决】

- 文件组织问题

  > 在自行将文件分类存放时，修改了文件名导致运行编译脚本出错，找不到输入的文件，最后发现问题修改文件名成功

- 操作系统初始化时初始化栈的地址出错

  > 一开始认为将栈底的地址设置的足够大一定可以，发现输出内容被截断，只有部分内容输出，寻求助教帮助后，发现栈的地址不能过打，依据代码布局正确设置栈底地址后成功

- 使用C语言强制类型转换失败

  > 在自行实现C语言`vsprintf`函数的`%d`格式输出时，将一个数字模`10`取余后使用`'char()'`强制类型转换出错，后来意识到应该通过`'+48'`实现数字到字符的转变