

中国科学技术大学计算机学院

实验报告



实验题目：lab2 Multiboot2 myMain

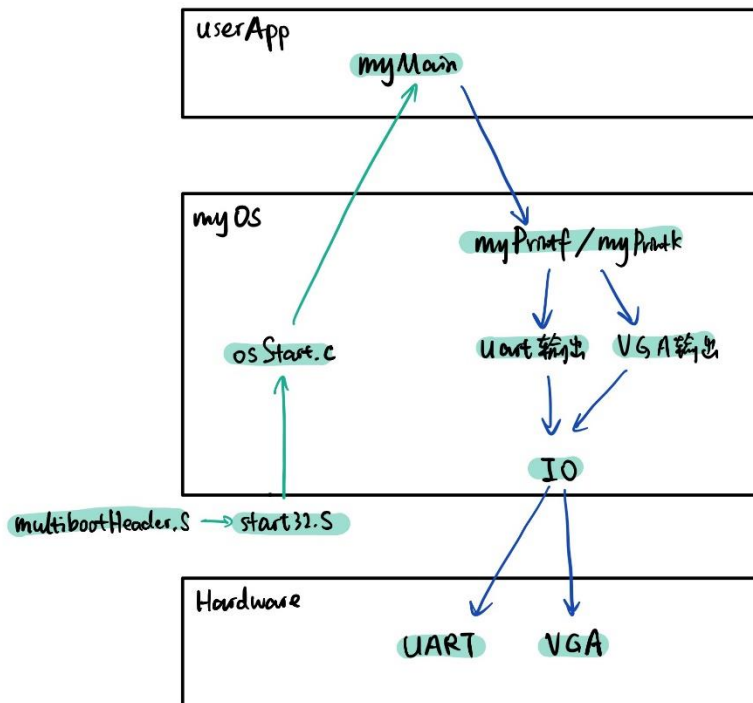
学生姓名：王舒

学生学号：PB19071472

完成日期：2021.4.7

一、软件框图

软件框图

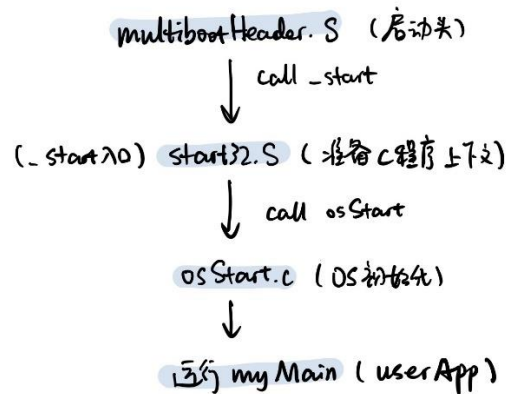


概述：

本次实验完成了一个简单的 OS 的初始化, 并通过它调用了一个用户态的程序 myMain。通过 multibootHeader.S 启动后, 进入 start32.S 为 C 语言环境准备上下文, 然后进入 osStart.c 来启动这个 OS。此时我们的操作系统唯一的功能就是调用一些简单的 myOS 的内部接口以及 userAPP。调用 myMain 后, 转入用户程序进行, 在其中可以调用用户自定义的函数和 OS 提供的接口, 比如 myPrintf/k 等。而这些函数, 将调用更底层的一些函数, 最终与硬件沟通, 完成功能。

二、主流程及其实现

主流程图



首先，用 multibootHeader.S 启动整个程序，在这段代码的最后有一句“call _start”，可以跳入到 start32.S 中的 _start 入口，从而用汇编语言为 C 语言准备上下文，包括建立栈、将 BSS 段初始化等。然后，利用“call osStart”跳转到 osStart.c 程序中。在 osStart.c 中，完成操作系统的初始化，清屏、输出特定提示字以及调用用户态程序 myMain.c 等功能。这样，就转入了用户态运行。

三、主要功能模块及其实现

1.I/O

端口输入输出采用嵌入式汇编。

1. I/O 端口输出

① outb 向端口写

value → register: eax

② inb 从端口读

value ← register: eax

①outb，向端口写一个字节

```
void outb (unsigned short int port_to, unsigned char value){
    __asm__ __volatile__ ("outb %b0,%w1"::"a" (value),"Nd" (port_to));
}
```

解释：__asm__表示后面的代码为内嵌汇编。__volatile__表示编译器不要优化代码。括号中的“outb %b0,%w1”是指令模板，代表指令的操作数，分别和后面的对应。“a”是寄存器 eax/ax/al 的简写，“Nd”中的 N 是用于 out 指令的 0-255 的立即数，d 表示寄存器 edx。括号中的 value 和 port_to 与 C 语言中的变量对应。

因此，这条代码的含义是将传入的 value 写入某个端口中。

②inb，从端口读一个字节

```
unsigned char inb(unsigned short int port_from){
    unsigned char _in_value;
    __asm__ __volatile__ ("inb %w1,%0"::"a"(_in_value): "Nd"(port_from));
    return _in_value;
}
```

解释：与上面所述类似，其中不一样的是“=”表示只读。

这个函数将某个端口中的值读到，并返回 (_in_value)。

2.串口 uart 输出

2. 串口 uart 输出

uart_put_char ← outb
↓
uart_put_chars

串口的输出比较简单。首先需要调用更底层的 outb 函数，将我们需要输出的字符从特定的端口地址 0x3F8 输出，也就是 uart_put_char 函数。

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

利用单个字符的输出函数，我们可以将字符串输出。其中，需要特判'\n'的情况，此时需要再打印一个'\r'。如图所示

```

void uart_put_chars(char *str){
    int i = 0;
    while (str[i] != '\0') {
        if (str[i] == '\n') { //特判
            uart_put_char('\n');
            uart_put_char('\r');
            i++;
        }
        else {
            uart_put_char(str[i]);
            i++;
        }
    }
}

```

3.VGA 输出

VGA 屏幕大小是 25*80，每个位置对应内存中的两个字节。为了代码编写方便，我写了一个简单的函数 calc_cord，将 VGA 输出时的行和列的信息转化成这个位置的内存地址。为了字符的操作方便，对内存的访问以两个字节为单位，也就是一个 short。如下：

```

int x = 0, y = 0; //x表示列，y表示行

//计算当前读取到的内存位置
short* calc_cord(int x, int y) {
    return (short*)0xB8000 + y * 80 + x;
}

```

其中 x、y 是全局变量。

①清屏功能

清屏功能比较简单，将所有显示部分的内存改为有背景色的空字符即可。

```

void clear_screen(void) {
    /* ... */
    for (int i = 0; i < 25; i++) {
        for (int j = 0; j < 80; j++) {
            (*calc_cord(j, i)) = 0x0f00;
        }
    }
    x = y = 0;
    cursor();
}

```

②屏幕输出功能，包括光标的显示，屏满时滚屏

(1) 光标显示函数 cursor

光标显示的原理是先将 0xE 或 0xF 写进寄存器 0x3D4 中，这样可以指定行号或者列号寄存器（类似于选择器 mux），然后再把对应的实际的行号或者列号写入端口 0x3D5 中。经过测试，发现光标位置与起始位置之差的后八位是列号，前八位是行号，因此代码如下

```
void cursor() {
    outb(0x3D4, 0xE); //行
    outb(0x3D5, (y * 80 + x) >> 8);
    outb(0x3D4, 0xF); //列
    outb(0x3D5, (y * 80 + x) % (1 << 8));
}
```

(2) 滚屏函数 add_oneline

该函数的功能是将屏幕上所有内容向上移动一行，原来的第一行消失，最后一行清空。实现方法是除了最后一行以外，将第 i 列第 (j+1) 行的内存中的内容覆盖第 j 行对应的内存，而最后一行类似于清屏的操作。代码如下

```
void add_oneline() {
    /* ... */
    for (int i = 0; i < N-1; i++) {
        for (int j = 0; j < 80; j++) {
            (*calc_cord(j, i)) = (*calc_cord(j, i+1));
        }
    }
    for (int i = 0; i < 80; i++) {
        (*calc_cord(i, N - 1)) = 0x0f00;
    }
}
```

(3) 显示输出函数 append2screen

把需要输出的字符和颜色打包成一个 2 字节的数，然后写入显存中的对应位置。要注意屏满的检查以及对 '\n' 的处理。在全部输出完成后，显示光标位置。代码如下

```

void append2screen(char *kBuf, int color) {
    //直接写VGA显存

    int j = 0;
    short temp;
    while (kBuf[j] != '\0') {
        if (kBuf[j] == '\n') {
            j++;
            x = 0;
            y++;
            if (y == N) {
                add_oneline();
                y = N-1;
            }
        }
        else {
            temp = (short)color << 8 | (short)kBuf[j]; //一个2bytes的颜色+字符打包
            j++;
            (*calc_cord(x, y)) = temp;
            x++;
            if (x == 80) {
                x = 0;
                y++;
                if (y == N) {
                    add_oneline();
                    y = N-1;
                }
            }
        }
    }
    cursor();
}

```

4.实现 myPrintf/k

4. myPrintk/f ← uart-put-chars.
 ↑
 处理格式化字符串 (自编)
 ↙
 append2screen

利用之前写好的函数, 就可以完成屏幕输出的功能了。其中, myPirntf 是在应用层打印, 而 myPrintk 是在内核打印。还需要实现的功能是格式化字符串的处理。参考已有的 vsprintf 函数源代码, 利用 C 语言可变参数的特点, 写了一个可以处理%d 和%c 的模块 (需要考虑正负数的问题), 代码如下:

```

va_list ap;
int d;
char c,*s;
va_start(ap,fmt);

int strlen=0;//字符串的已使用长度
int i=0;
while(fmt[i]!='\0'){
    if(fmt[i]!='%'){
        kBuf[strlen] = fmt[i];
        strlen++;
        i++;
    }
    else{//读取到了%
        i++;
        if(fmt[i]=='\0'){//%\0是不合法的，当他是普通的%添加进去
            kBuf[strlen] = '%';
            strlen++;
            break;//因为已经读到了\0所以直接结束
        }else if(fmt[i]=='d'){//读取到了%d
            int num=va_arg(ap,int);//从va_list中切一个int下来
            int t=1;
            if (num == 0) {
                uBuf[strlen] = '0';
                strlen++;
            }
            else if (num < 0) {
                uBuf[strlen] = '-';
                strlen++;
                num *= -1;
                int t = 1;
                //下面将数字转化为字符
                while (t <= num) {
                    t *= 10;
                }
                t /= 10;
                while (t >= 1) {
                    uBuf[strlen] = '0' + (num / t) % 10;
                    strlen++;
                    t /= 10;
                }
            }
        }
    }
}

```



```

        else {
            int t = 1;
            while (t <= num) {
                t *= 10;
            }
            t /= 10;
            while (t >= 1) {
                uBuf[strlen] = '0' + (num / t) % 10;
                strlen++;
                t /= 10;
            }
        }
        i++;

    }else if(fmt[i]=='c') { //读取到了%c
        char ch=(char)va_arg(ap, int);
        kBuf[strlen] = ch;
        strlen++;
        i++;
    }
}

kBuf[strlen] = '\0';
strlen++;
va_end(ap);

```

这样就把需要输入的字符都放入了数组里，然后调用 VGA 和 uart 的输出函数即可。

四、源代码说明

1.目录组织

```

├── Makefile
├── multibootheader
│   └── multibootHeader.S
├── myOS//操作系统中的内部文件
│   ├── dev//串口输出和 VGA 输出相关的部分
│   │   ├── Makefile
│   │   ├── uart.c
│   │   └── vga.c
│   └── i386//与硬件沟通的部分

```

```

|   |   |—— io.c
|   |   |—— io.h
|   |   |—— Makefile
|   |—— Makefile
|   |—— myOS.ld
|   |—— osStart.c
|   |—— printk//实现 printf/k 功能的部分
|   |   |—— Makefile
|   |   |—— myPrintk.c
|   |—— start32.S
|—— output//所有编译链接生成的文件
|   |—— multibootheader
|   |   |—— multibootHeader.o
|   |—— myOS
|   |   |—— dev
|   |   |   |—— uart.o
|   |   |   |—— vga.o
|   |   |—— i386
|   |   |   |—— io.o
|   |   |—— osStart.o
|   |   |—— printk
|   |   |   |—— myPrintk.o
|   |   |—— start32.o
|   |—— myOS.elf
|   |—— userApp
|   |   |—— main.o
|—— README.txt
|—— source2run.sh
|—— userApp//用户态的文件
|   |—— main.c
|   |—— Makefile

```

2.Makefile 组织

src 目录下的 Makefile 将各种其他子 Makefile 文件串联起来。在 output/myOS.elf 中, 将 OS_OBJS 包含进去, 而 OS_OBJS 将 MYOS_OBJS 和 USER_APP_OBJS 包含进去, MYOS_OBJS 中又把 DEV_OBJS、i386_OBJS 和 PRINTK_OBJS 包含进去。这样就把所有需要编译成可执行文件的内容包含进去了。

五、代码布局说明（地址空间）

这个 ld 文件将各可执行文件中的 text、data 和 bss 段分别拆开，然后拼接到一起。

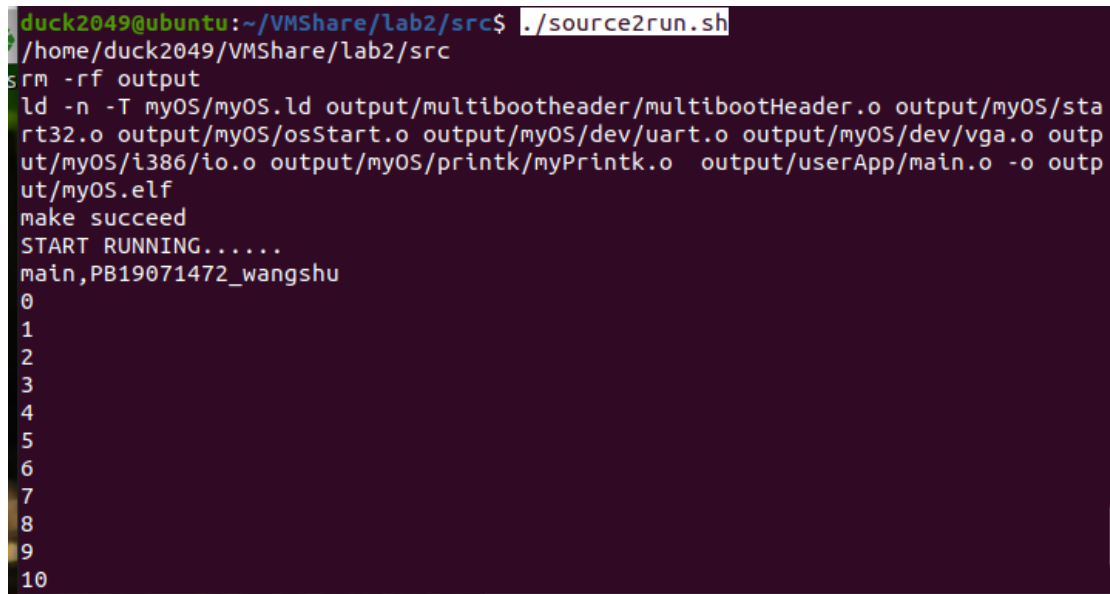
在 .text 段中，先把 multiboot_header 的启动头写进去，然后对齐，再把 text 部分写入。再对齐后，把 data 部分写入。再次对齐，把 bss 段写入，再经过一些对齐的处理。经过编译，就可以按照 ld 文件生成 elf 文件。

六、编译过程说明

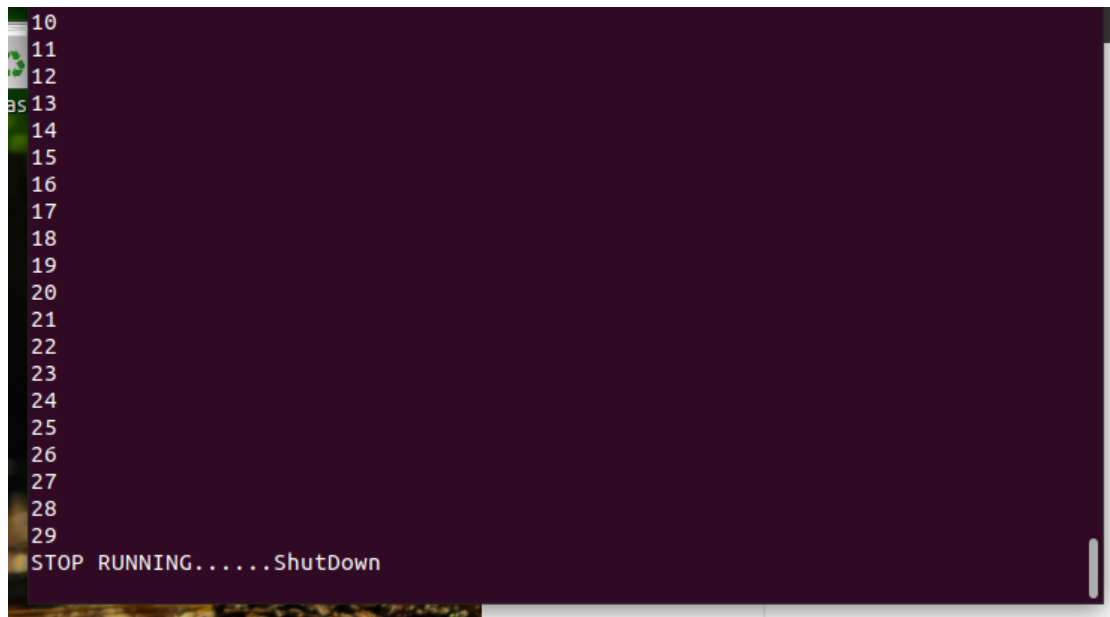
Src 目录下的 Makefile 将所有 .S 文件和 .c 文件按照 gcc 编译为可执行文件，然后按照 ld 文件的指示生成 elf 文件。（所有生成的可执行文件和 elf 文件都被放到了 output 文件夹中）。

七、运行结果

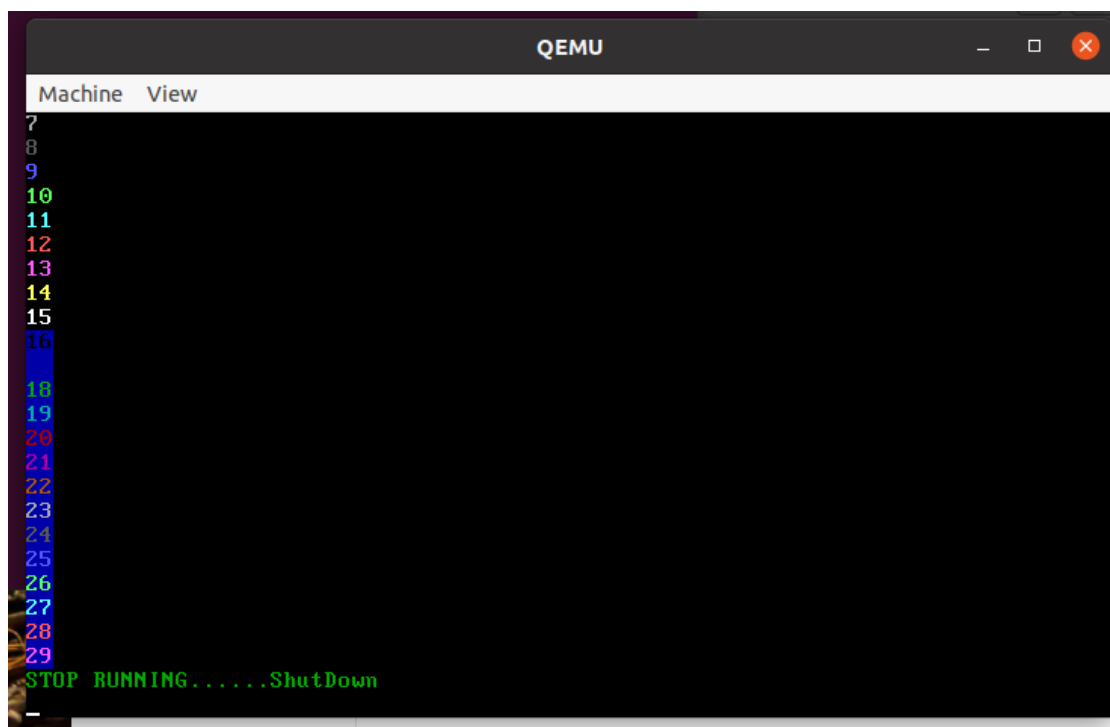
在 terminal 中敲入脚本的运行命令 ./source2run.sh，成功编译并运行。在控制台中，输出了内核态和用户态的目标内容，如下



```
duck2049@ubuntu:~/VMShare/lab2/src$ ./source2run.sh
/home/duck2049/VMShare/lab2/src
rm -rf output
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start32.o output/myOS/osStart.o output/myOS/dev/uart.o output/myOS/dev/vga.o output/myOS/i386/io.o output/myOS/printk/myPrintk.o output/userApp/main.o -o output/myOS.elf
make succeed
START RUNNING.....
main,PB19071472_wangshu
0
1
2
3
4
5
6
7
8
9
10
```



而在 qemu 中，实现了滚屏和光标的显示（由于有换行，光标显示在下一行）



八、遇到的问题和解决方案

相较于第一次实验来说，本次实验的难度明显加大，在实验过程中也遇到了很多问题。

(1) 对于实验内容的理解比较困难。由于文件的数量变多了，之间的内部逻辑也更加复杂，刚入手时非常困惑，需要一点点摸索，每个文件都要弄清楚是在干什么，还要弄明白自己要

做些什么、调用哪些函数。

(2) 由于对 Linux 系统的操作不熟悉，内部环境也没有调好，面对那个配色非常严肃的 terminal 窗口经常不知所措。比如在一开始修改 Makefile 文件里的路径时，就因为不知道还有 pwd 这种奇妙的指令，浪费了很多时间。又比如由于没有安装链接库，在调用很多头文件时出现了很大的问题，以至于晚上做梦都梦到被 `sys/types.h` 追着跑。

(3) 在正式摸清自己的任务，开始写代码的时候，还是很茫然。在处理格式化字符串的时候，我一开始以为只要从网上随便找一个 `vsprintf` 函数就可以用了，后来却因为头函数的问题，挣扎了很久还是放弃了。最后找了很多相关资料（和靠谱的大腿），自己写了一个简单的处理，才把这个问题解决。再比如由于对 C 语言中如何直接写内存以及地址的计算很陌生，要查阅很多资料学习，然后摸索怎么才能写对。回想起来，每个函数思路都其实不难，但常常缺乏放手尝试的勇气。从一个直白的功能过渡到如何实现它，实际上还是有很大距离，需要想很久，这或许也是我欠缺的能力吧。

总而言之，当我做完实验时，再去回忆近十天的实验过程，觉得每一步都好像很显然。但在刚开始接触一个自己不熟悉的系统、还要用它实现很多奇怪的功能的时候，可以说是举步维艰。面对接踵而至大大小小的问题，只能自己查阅资料（以及求助大腿），尝试去理解它们。通过这个实验，我明显感觉到自己的学习能力得到了很大的提升，奇怪的知识了解了很多，对于理论课的理解也加深了。更重要的或许是，正如 Linux 系统的包容与开放性一样，想去做什么就放手去做的那份勇气。我第一次感觉到作为一个没有头发的程序员，在自己的电脑上肆意妄为的快乐和骄傲。

注：本次实验使用的 lab1 基础来自于老师提供的代码（multibootHeader.S）。回顾自己的 lab1 实现情况，感觉良好，理解比较到位。