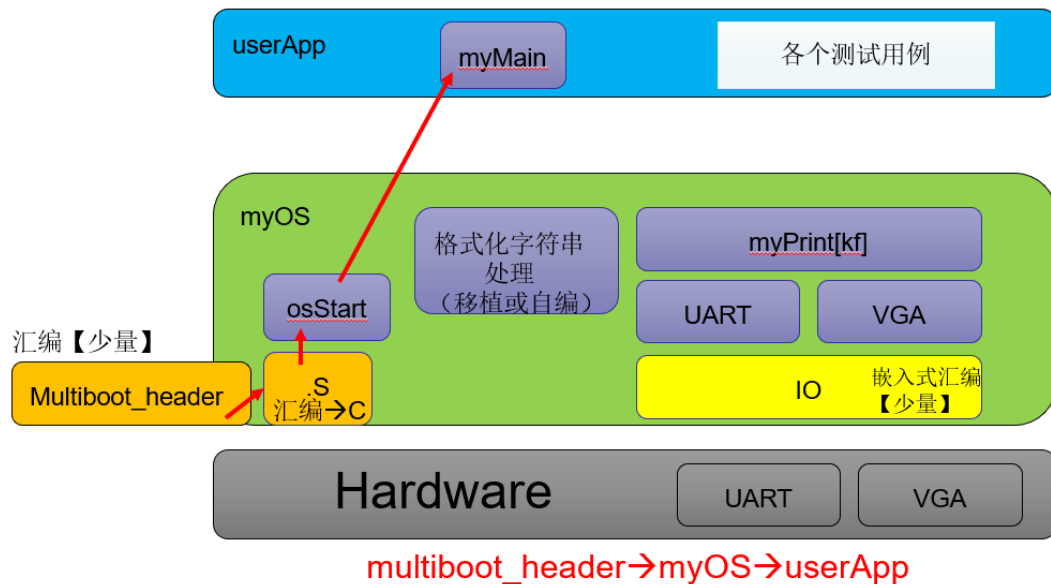


实验二

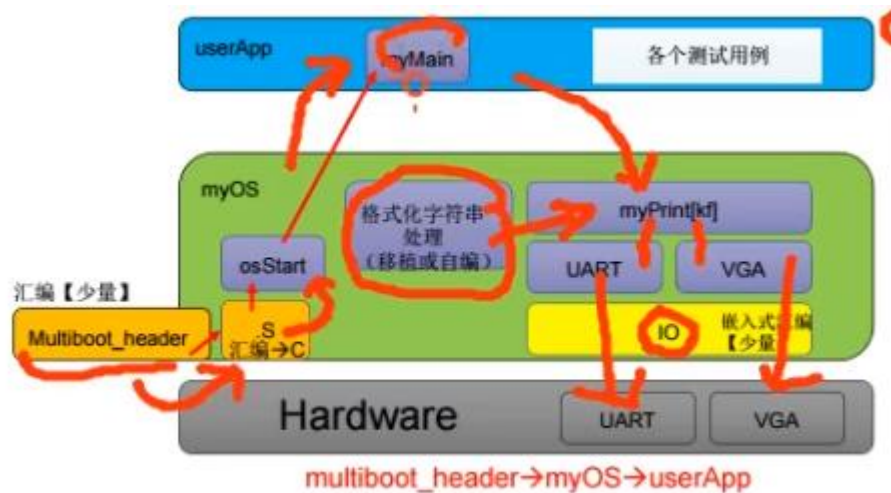
PB21020651 武宇星

软件框架



主流程

1. 通过 MultiBootHeader 启动 Qemu。
2. 跳转到内部的.S 文件，为 C 语言环境初始化。
3. 调用 `_start`，跳转到 `osstart` 函数。
4. 从 `osstart` 函数，进入到 `myMain` 用户程序。
5. 在 `myMain` 用户程序中，调用 `myPrint[kf]` 实现屏幕输出和串口输出。
6. 利用内嵌汇编完成与硬件的交互，实现 UART 和 VGA 的输入输出操作。



主要功能模块

src/myOS/start32.S 的编写：

观察 myOS.ld 文件，发现_end 表示 bss 结束地址

```
establish_stack:
    movl    $_end, %eax      # eax = end of bss/start of heap #填充
    addl    $STACK_SIZE, %eax # make room for stack
    andl    $0xffffffc0, %eax # align it on 16 byte boundary

    movl    %eax, %esp      # set stack pointer
    movl    %eax, %ebp      # set base pointer
```

I/O 端口：

根据实验文档代码完成 inb, outb 读写端口的字节数据

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

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

inb 函数使用 __asm__ __volatile__ 描述内嵌汇编代码。

"inb %w1, %b0" 是汇编指令，用于从端口 %w1 读取一个字节，并将其存储到 %b0。

%w1 表示读取的端口，对应 C 语言变量 port_from, w 表示长度为 16 位，N 表示立即数，d 表示先存入寄存器 %edx，再作为指令的操作数。

%b0 表示读取内容的存放处，对应 C 语言变量 value, b 表示长度为 8 位，=a 表示先读入寄存器 %eax，再将其写入 C 语言变量 value 中。

outb 函数：与 inb 函数类似，使用 __asm__ __volatile__ 描述内嵌汇编代码。

"outb %b0, %w1" 是汇编指令，用于将字节 %b0 写入端口 %w1。

%b0 表示要写入的字节，对应 C 语言变量 value。

%w1 表示目标端口，对应 C 语言变量 port_to。

UART 输出模块：

先调用 outb 完成 uart_put_char，而在 uart_put_chars 中只需循环调用 uart_put_char，并对回车换行进行处理，遇到 '\n' 实际输出 '\r'。而 uart_get_char 则检查需先 UART 状态寄存器的第 0 位（值为 0x01）来判断接收缓冲区是否有数据可读。如果有数据可读，则使用 inb 读取一个字符并返回。

```

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) & 0x01) == 0);
    return inb(uart_base);
}

void uart_put_chars(char *str){
    while (*str) {
        if (*str == '\n') {
            uart_put_char('\r');
        }
        uart_put_char(*str);
        str++;
    }
}

```

VGA 输出模块:

宏定义 VGA 基地址, 屏幕
宽高及字体颜色

```

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

#define VGA_BASE_ADDR 0xb8000
#define ROWS 25
#define COLS 80

#define BLACK    0x0
#define BLUE     0x1
#define GREEN    0x2
#define CYAN     0x3
#define RED      0x4
#define MAGENTA  0x5
#define BROWN   0x6
#define LGRAY    0x7

#define BRIGHT   0x8
#define DGRAY    BRIGHT & BLACK
#define LBLUE    BRIGHT & BLUE
#define LGREEN   BRIGHT & GREEN
#define LCYAN    BRIGHT & CYAN
#define LRED     BRIGHT & RED
#define PINK     BRIGHT & MAGENTA
#define YELLOW   BRIGHT & BROWN
#define WHITE    BRIGHT & LGRAY

#define BLINK    0x80

```

更新当前光标的位置：

```
static void update_cursor(unsigned int row, unsigned int col) {
    unsigned int cursor_loc = row * 80 + col;

    outb(0x3D4, 14);
    outb(0x3D5, cursor_loc >> 8);
    outb(0x3D4, 15);
    outb(0x3D5, cursor_loc & 0xff);
}
```

设置行号：将行号寄存器写入索引端口，然后将光标位置的高八位写入数据端口。

设置列号：将列号寄存器写入索引端口，然后将光标位置的低八位写入数据端口。

获取当前光标的位置：

```
unsigned char* get_cursor_position(int row, int col) {
    unsigned char* addr = (unsigned char*)VGA_BASE_ADDR;
    return addr + row * COLS * 2 + col * 2;
}
```

屏幕内存是一个二维数组，每个字符占两个字节，故根据 VGA 屏幕内存的基地址，行数和列数计算光标在屏幕内存中的地址。

输出字符串到屏幕：

初始化一个字符指针 `current`，指向输入字符串 `str`。

从 VGA 控制器获取当前光标位置，并计算光标所在的行和列。

使用一个 `while` 循环遍历字符串中的每个字符。

在每次循环迭代中，检查字符是否为换行符（`'\n'`）：

如果字符不是换行符，使用 `write_char` 函数将字符写入屏幕，并将列值加 1。

如果字符是换行符，将列值设置为 0，行值加 1。

在每次迭代之后，检查列值是否达到屏幕宽度：

如果是，将列值设置为 0，行值加 1。

检查行值是否达到屏幕高度：

如果是，调用 `scroll_one_row` 函数滚动屏幕，并将行值设置为最后一行。

循环结束后，使用 `update_cursor` 函数更新光标位置。

```
void append2screen(char* str, int color) {
    char ch, *current = str;
    unsigned int cursor_loc;
    int row, col;
    outb(0x3D4, 14);
    cursor_loc = inb(0x3D5) << 8;
    outb(0x3D4, 15);
    cursor_loc |= inb(0x3D5);

    row = cursor_loc / 80;
    col = cursor_loc % 80;

    while (ch = *current++) {
        if (ch != '\n') {
            write_char(ch, color, row, col++);
        } else {
            col = 0;
            row++;
        }

        if (col >= COLS) {
            col = 0;
            row++;
        }
        if (row >= ROWS) {
            scroll_one_row();
            row = ROWS - 1;
        }
    }
    update_cursor(row, col);
}
```

滚屏:

滚屏的原理就是不断将下一行的字符复制到本行，最终清空最后一行。故首先完成清一行功能，即用空字符填满该行。

```
void clear_char(int row, int col) {
    unsigned char* addr = get_cursor_position(row, col);
    *addr++ = 0;
    *addr = 0x7;
}

void clear_last_row(void) {
    int col;
    for (col = 0; col < COLS; col++) clear_char(ROWS - 1, col);
}

void scroll_one_row(void) {
    int i;
    unsigned char* ptr = (unsigned char*)VGA_BASE_ADDR;
    unsigned char* next_row_ptr = get_cursor_position(1, 0);

    for (i = 0; i < (ROWS - 1) * COLS; i++) {
        *ptr++ = *next_row_ptr++;
        *ptr++ = *next_row_ptr++;
    }
    clear_last_row();
}
```

清屏:

清屏即遍历行列全部置空字符，最后将光标移到左上角位置。

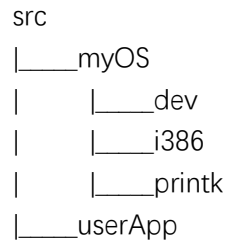
```
void clear_screen(void) {
    unsigned char* ptr = (unsigned char*)VGA_BASE_ADDR;
    unsigned int row, col;
    for (row = 0; row < ROWS; row++) {
        for (col = 0; col < COLS; col++) {
            (*ptr++) = 0;
            (*ptr++) = 0x7;
        }
    }
    update_cursor(0, 0);
    return;
}
```

源代码说明：

目录组织：



Makefile 组织：



代码布局说明（地址空间）：

SECTIONS 部分定义了程序在内存中的各个段的布局。以下是各个段的说明：

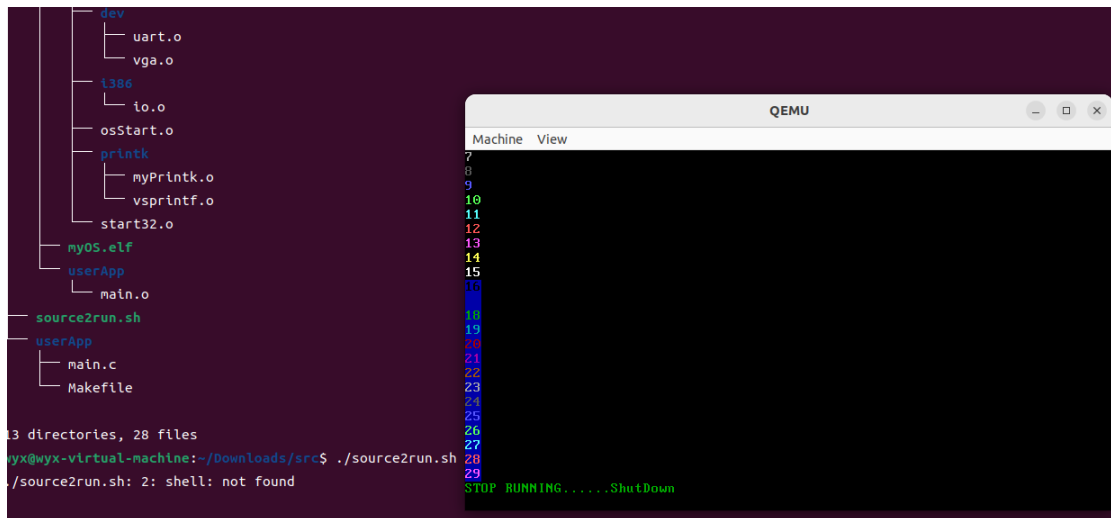
1. **. = 1M;** 将位置计数器设置为 1MB。这意味着程序将从 1MB 处开始加载。
2. **.text** 段包含程序的代码。首先，它包括名为 **".multiboot_header"** 的段，然后是 **".text"** 段。**.multiboot_header** 是多引导标头，它使程序与多引导协议兼容，允许诸如 GRUB 等引导加载器加载它。**ALIGN(8)** 确保在 **.text** 段之前有足够的对齐空间。
3. **.data** 段包含程序的已初始化数据。它在内存中对齐为 16 字节边界。
4. **.bss** 段包含程序的未初始化数据。这些数据在程序启动时被初始化为零。该段在内存中也对齐为 16 字节边界。**__bss_start** 和 **_bss_start** 符号表示 **.bss** 段的开始，而 **__bss_end** 符号表示它的结束。
5. **_end** 符号表示整个程序的结束地址。**ALIGN(512)** 确保结束地址对齐为 512 字节边界。

编译过程说明:

1. 编译汇编.S 源文件与 c 语言.c 源文件, 生成.o 文件。这一步需要包括各目录下的 makefile 文件。
2. 将各.o 文件进行链接, 生成 myOS.elf 文件。

运行和运行结果说明:

首先 `chmod +x source2run.sh` 赋予权限, 然后 `source2run.sh` 运行



```
tree
.
├── dev
│   ├── uart.o
│   └── vga.o
├── i386
│   └── io.o
├── osStart.o
├── printk
│   ├── myPrintk.o
│   └── vsprintf.o
├── start32.o
├── myOS.elf
├── userApp
│   └── main.o
├── source2run.sh
├── userApp
│   ├── main.c
│   └── Makefile
└── 13 directories, 28 files

wyx@wyx-virtual-machine:~/Downloads/src$ ./source2run.sh
./source2run.sh: 2: shell: not found

QEMU
Machine View
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
```