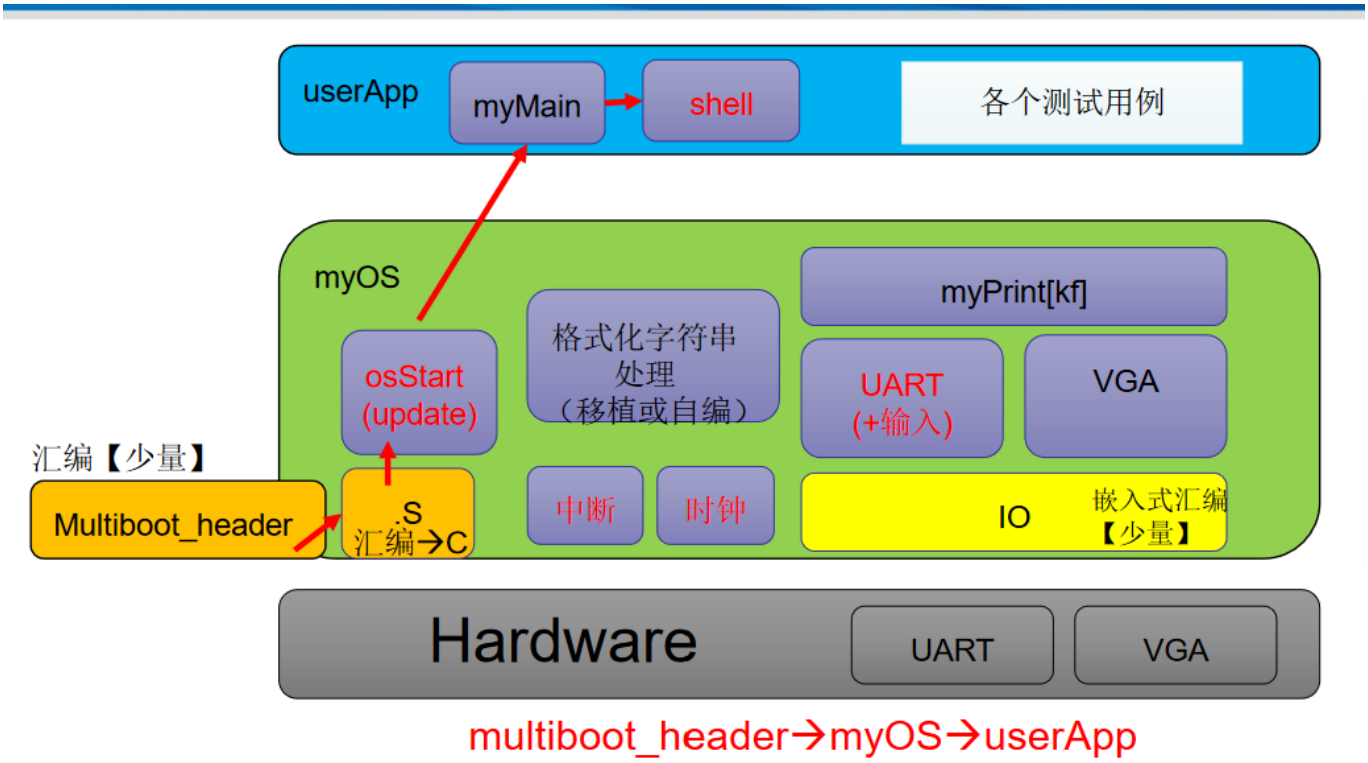
第三次实验报告

学号：PB20000024

姓名：陈奕衡

一、软件框图



相比于上一次实验，本次实验主要增加了三个新的模块：中断模块、时钟模块以及`Powershell`模块。由图可以看出，中断与时钟模块属于系统内部功能模块，而`Powershell`模块则属于用户程序。

二、主流程说明

主流程与实验二稍有不同，首先是`qemu`虚拟机启动并运行`Multiboot_header`文件，通过启动头进入`myOS`操作系统（访问`start.S`），在操作系统中通过`start32.S`文件运行`osStart.c`文件，在文件中进行中断控制器的初始化以及允许时钟中断。如下所示：

```
void osStart(void) {
    init8259A();
    init8253();
    tick();
```

```
    enable_interrupt();

    clear_screen();
    myPrintk(0x2, "START RUNNING......\n");
    myMain();
    myPrintk(0x2, "STOP RUNNING......ShutDown\n");
    while (1);
}
```

最后通过myMain函数进行shell程序的调用，这样便可通过串口重定向来对myshell进行输出。

三、主要功能模块及其相应源代码

中断控制器

首先是中断描述符表IDT，分配内存过程如下：

```
.data
# IDT
    .p2align 4
    .globl IDT
IDT:

    .rept 256
    .word 0,0,0,0
    .endr
idtptr:

    .word (256*8 - 1)
    .long IDT
```

将所有中断处理程序初始化为合适的缺省处理函数，例如ignore_int1：

```
ignore_int1:
    cld
    pusha
    call ignoreIntBody
    popa
    iret
```

```
void ignoreIntBody(void){
    char str[] = "unknown interrupt\0";
    unsigned short int pos = VGA_SCREEN_HEIGHT * VGA_SCREEN_WIDTH;

    for (int i = 0; *(str + i) != '\0'; i++)
        put_char2pos(*(str + i), 0x2, pos + i);
```

```
    return;
}
```

此处，将未知中断提示放在了vga显存的左下角。

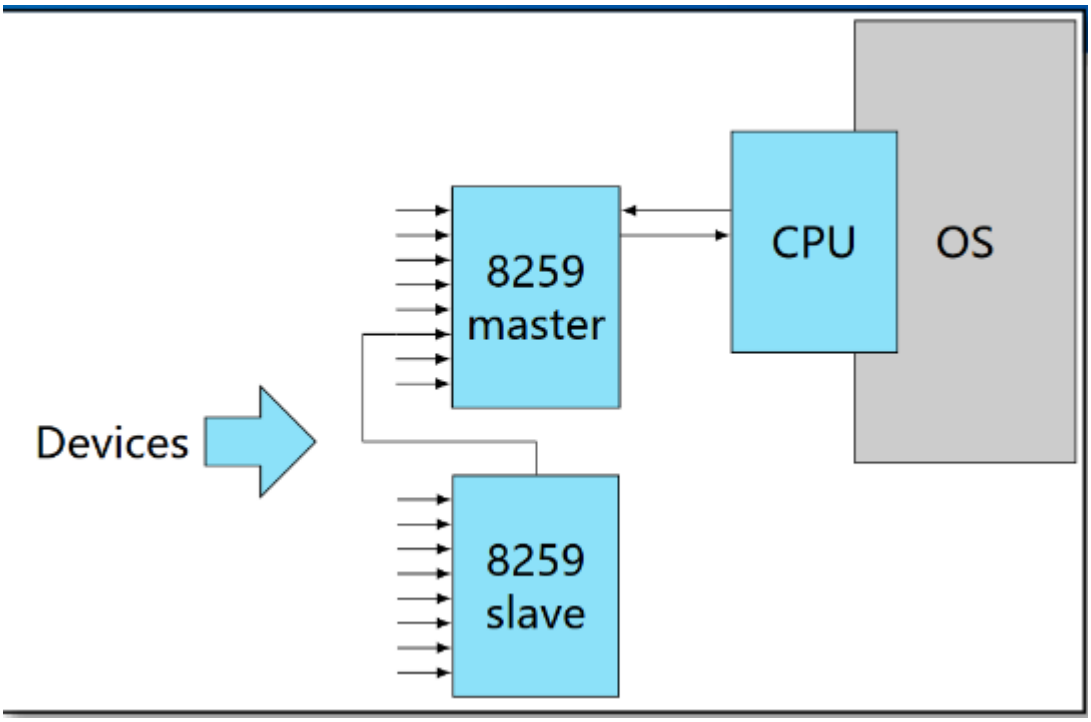
寄存器IDTR的初始化：

```
setup_idt:
    movl $ignore_int1,%edx
    movl $0x00080000,%eax
    movw %dx,%ax /* selector = 0x0010 = cs */
    movw $0x8E00,%dx /* interrupt gate - dpl=0, present */
    movl $IDT,%edi
    mov $256,%ecx

rp_sidt:
    movl %eax,(%edi)
    movl %edx,4(%edi)
    addl $8,%edi
    dec %ecx
    jne rp_sidt

    lidt idtptr
```

可编程中断控制器PIC_i8259初始化：

```
void init8259A(void){
    outb(0x21, 0xFF); //shield the interrupt
    outb(0xA1, 0xFF);

    outb(0x20, 0x11); //initialize main chip
    outb(0x21, 0x20);
    outb(0x21, 0x04);
    outb(0x21, 0x3);

    outb(0xA0, 0x11); //initialize slave chip
    outb(0xA1, 0x28);
    outb(0xA1, 0x02);
    outb(0xA1, 0x01);

    return;
}
```

实际效果如下：



最后是开关中断指令：

```
.globl enable_interrupt
enable_interrupt:
    sti
    ret

.globl disable_interrupt
disable_interrupt:
    cli
    ret
```

Tick和tick维护

首先是可编程间隔定时器PIT_i8253的初始化：

```
void init8253(void){
    const short int fre = 1193180 /100;

    outb(0x43, 0x34); //PIT control  word

    outb(0x40, fre & 0xff); //fre constant
    outb(0x40, fre >> 8);

    unsigned char master = inb(0x21); //enable time interrupt
    unsigned char slave = inb(0xA1);

    master = master & 0xfe;
    slave = slave & 0xfe;
```

```
    outb(0x21, master);
    outb(0xA1, slave);

    return;
}
```

这里设定的时钟参数约为100Hz，最后通过8259允许时钟中断。之后是时钟计时，代码如下：

```
void tick(void){
    system_ticks++;

    int act_second = system_ticks / 100;
    int act_minute = act_second / 60;
    int act_hour = act_minute / 60;

    SS = act_second % 60;
    MM = act_minute % 60;
    HH = act_hour % 24;

    oneTickUpdateWallClock(HH, MM, SS);

    return;
}
```

`SS`，`MM`，`HH`分别是时钟的秒、分、时。这三个值为全局变量，会在墙钟模块被调用显示和利用hook机制在用户界面被修改。并且，这里使用`system_ticks`全局变量来维护`tick`调用次数。

维护墙钟和显示墙钟

维护墙钟代码如上，每次输出新的时、分、秒的值，显示墙钟的代码如下：

```
void setWallClock(int HH,int MM,int SS){
    unsigned short int pos = (VGA_SCREEN_HEIGHT + 1) * VGA_SCREEN_WIDTH - 8;
    int color = 0x2;

    put_char2pos(HH / 10 + '0', color, pos);
    put_char2pos(HH % 10 + '0', color, pos + 1);
    put_char2pos(':', color, pos + 2);
    put_char2pos(MM / 10 + '0', color, pos + 3);
    put_char2pos(MM % 10 + '0', color, pos + 4);
    put_char2pos(':', color, pos + 5);
    put_char2pos(SS / 10 + '0', color, pos + 6);
    put_char2pos(SS % 10 + '0', color, pos + 7);

    return;
}
```

可以看出这里将时钟设置在了终端右下角。下面是获取当前时钟的代码，只需读取右下角特定位置即可。

```
void getWallClock(int *HH,int *MM,int *SS){
    unsigned short int pos = (VGA_SCREEN_HEIGHT + 1) * VGA_SCREEN_WIDTH - 8;
    unsigned short int *ptr;
    ptr = (short int *)(VGA_BASE + pos);

    *HH = (ptr[0] - '0') * 10 + ptr[1] - '0';
    *MM = (ptr[3] - '0') * 10 + ptr[4] - '0';
    *SS = (ptr[6] - '0') * 10 + ptr[7] - '0';

    return;
}
```

Shell的实现

Shell的实现首先需要串口进行输入输出：

```
#include "uart.h"

unsigned char uart_get_char(void) {
    while (!(inb(UART_PORT + 5) & 1));
    return inb(UART_PORT);
}

/* 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 */
    unsigned short int port = UART_PORT;

    outb(port, ch);
    if (ch == '\n') outb(UART_PORT, '\r');

    return;
}

/* 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++)
        uart_put_char(*(str + i));

    return;
}
```

从串口读取数据并且写回至终端部分如下：

```
BUF_len=0;
int argc = 0;
char argv[8][8];
myPrintf(0x07,"Student>>\0");
while((BUF[BUF_len]=uart_get_char())!='\r') {
    myPrintf(0x07, "%c", BUF[BUF_len]);//将串口输入的数存入BUF数组中
    BUF_len++;  //BUF数组的长度加
}
myPrintf(0x07, "\n");
BUF[BUF_len] = '\0';
```

之后是对argc和argv的处理：

```
int j = 0;
for (int i = 0; i < BUF_len; i++) {
    if (BUF[i] == ' ' && j > 0){
        argv[argc++][j] = '\0';
        j = 0;
    } else {
        argv[argc][j++] = BUF[i];
    }
}
argv[argc][j] = '\0';
argv[++argc][0] = '\0';
```

最后是根据得到的argc和argv进行相应的指令执行：

```
if (strcmp(argv[0], cmd.name) == 0) {
    cmd.func(argc, argv);
}
else if (strcmp(argv[0], help.name) == 0) {
    help.func(argc, argv);
}
else if (strcmp(argv[0], "exit") == 0) {
    return;
}
else {
    myPrintf(0x2, "Command %s is not found\n", argv[0]);
}
```

这里说明一下，此处的strcmp函数是vsprint函数中自带的函数。并且，这里的vsprintf函数具备更多处理标准化字符串的能力。

```
int strcmp(const char *source,const char *dest)
{
    int ret = 0;
```

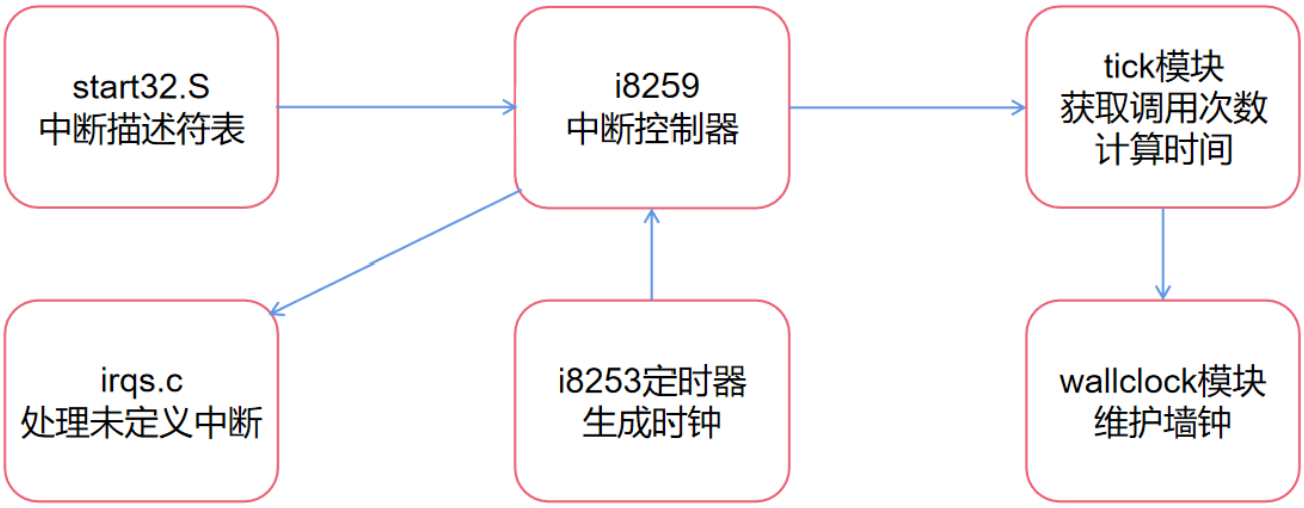
```
    if(!source || !dest) return -2;
    while( ! (ret = *( unsigned char *)source - *(unsigned char *)dest) && *dest)
    {
        source++;
        dest++;
    }

    if ( ret < 0 )
        ret = -1 ;
    else if ( ret > 0 )
        ret = 1 ;

    return(ret);
}
```

相应的指令函数如下：

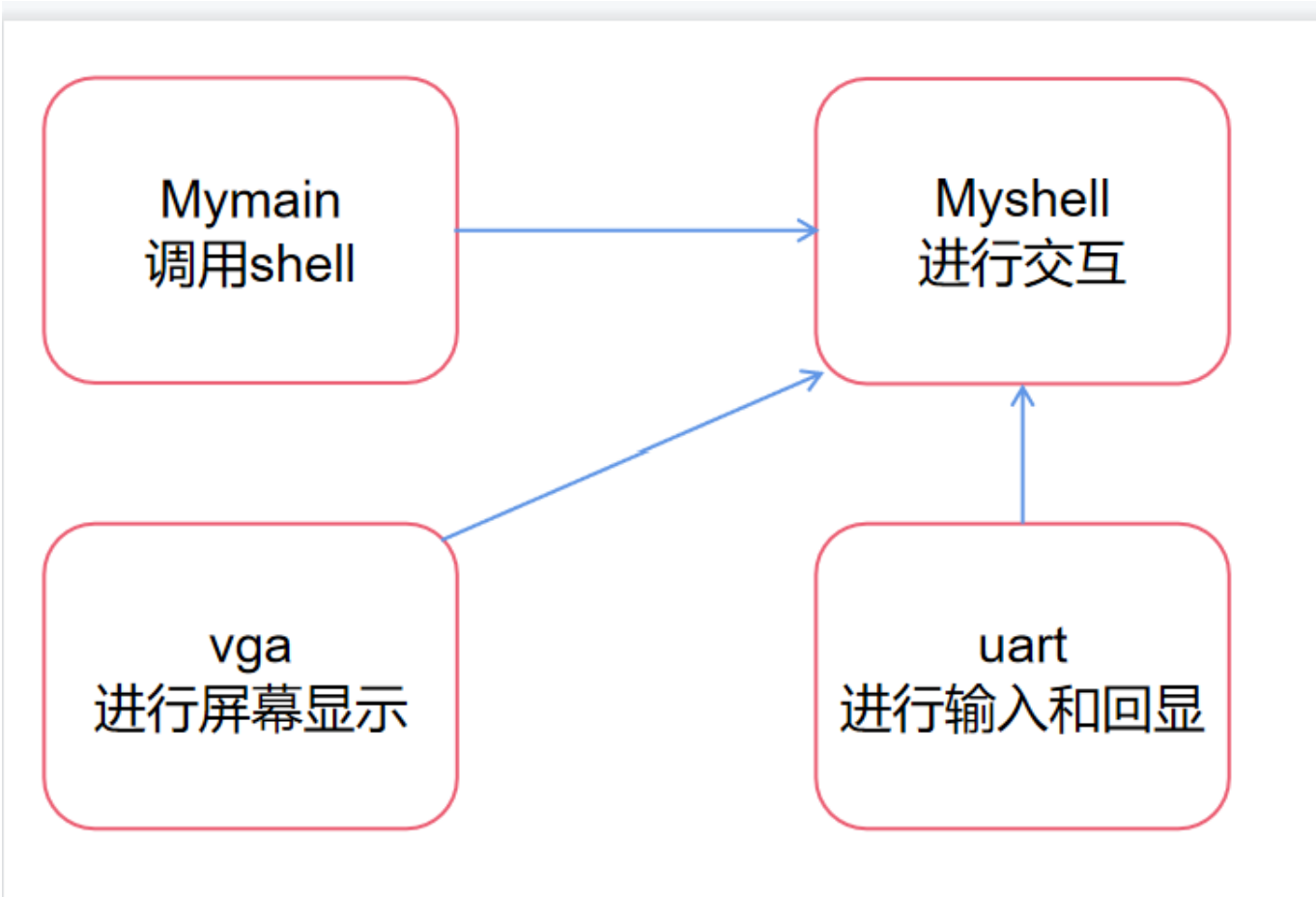
```
int func_cmd(int argc, char (*argv)[8]){
    myPrintf(0x7, "%s\n", cmd.name);
    myPrintf(0x7, "%s\n", help.name);

    return 0;
}
int func_help(int argc, char (*argv)[8]){
    if (argc == 1){
        myPrintf(0x7, "%s", help.help_content);
        return 0;
    }
    else if (strcmp(argv[1], cmd.name) == 0) {
        myPrintf(0x7, "%s", cmd.help_content);
        return 0;
    }
    else if(strcmp(argv[1], help.name) == 0) {
        myPrintf(0x7, "%s", help.help_content);
        return 0;
    }

    myPrintf(0x7, "command %s is not found\n", argv[1]);

    return 0;
}
```

至此，主模块描述完毕，下面为主模块流程图：

- 时钟模块

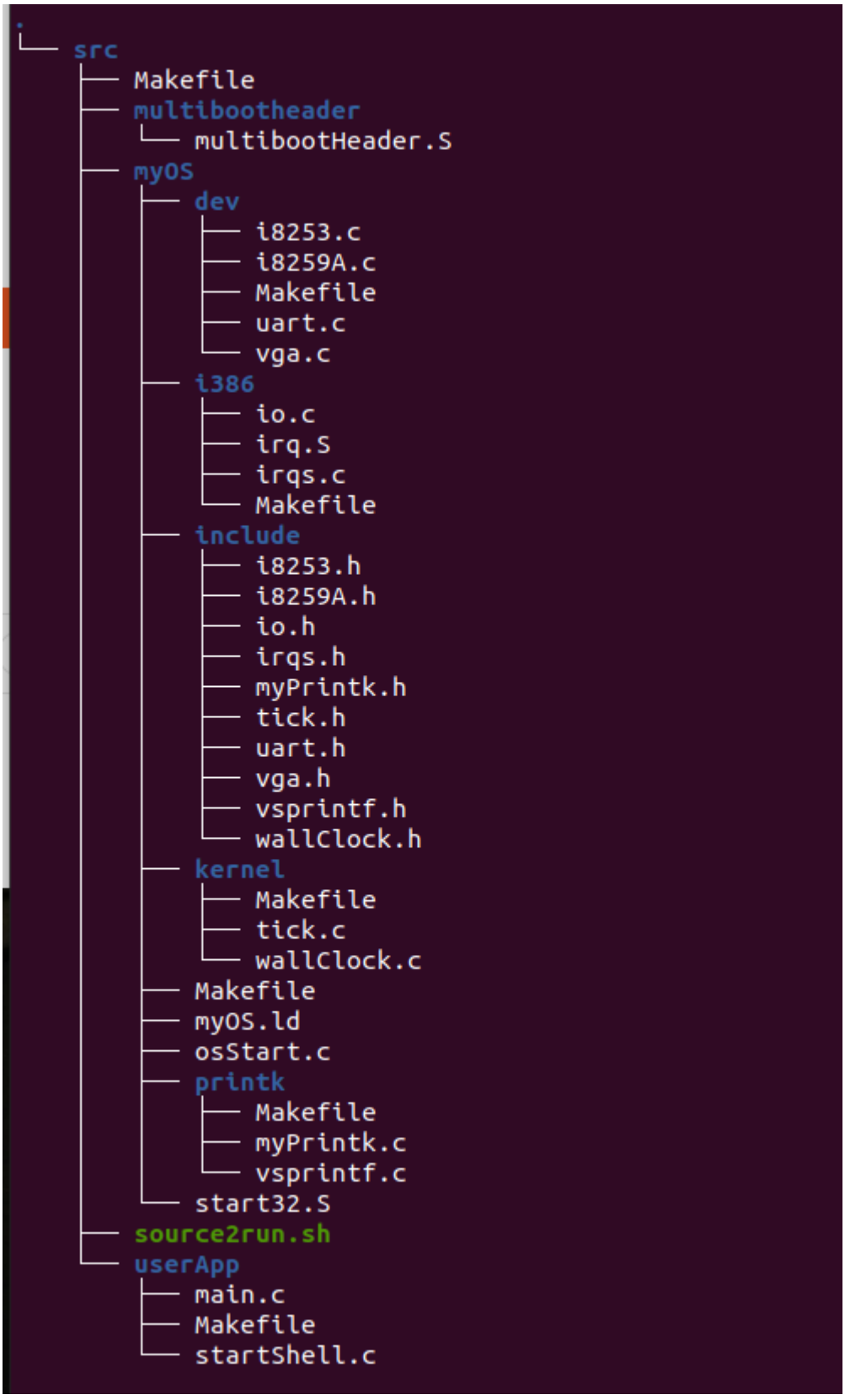


- shell程序



四、组织说明

本实验组织架构如下:



Makefile组织架构如下：

```
MULTI_BOOT_HEADER = output/multibootheader/multibootHeader.o

MYOS_OBJS = output/myOS/start32.o output/myOS/osStart.o \
          DEV_OBJS = output/myOS/dev/uart.o \
                     output/myOS/dev/vga.o \
                     output/myOS/dev/i8259A.o \
                     output/myOS/dev/i8253.o
          I386_OBJS = output/myOS/i386/io.o \
                     output/myOS/i386/irqs.o \
                     output/myOS/i386/irq.o
       PRINTK_OBJS = output/myOS/printk/myPrintk.o \
                     output/myOS/printk/vsprintf.o
       KERNEL_OBJS = output/myOS/kernel/tick.o \
                     output/myOS/kernel/wallClock.o

USER_APP_OBJS =  output/userApp/main.o \
                 output/userApp/startShell.o
```

五、代码布局说明

代码空间分布如下：

```
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);
}
```

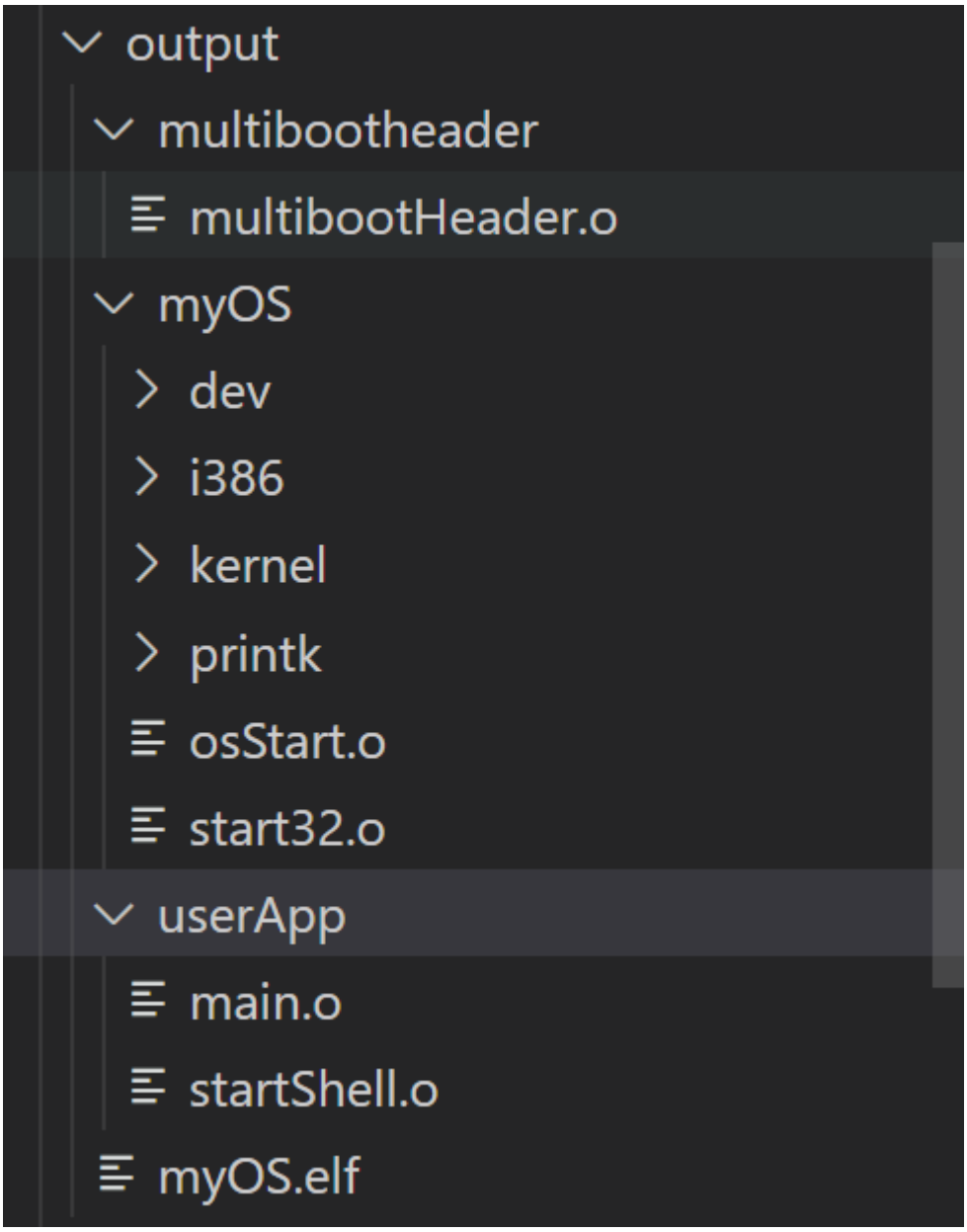
其中.text代码段在1m位置进行存储，以8字节形式对齐；.data段、.bss段以16字节形式对齐；最后的_end以及后续代码以512字节形式对齐。

六、编译过程

编译所用脚本文件如下，在终端输入`./source2run.sh`便可一键编译加重定向串口：

```
#!/bin/bash
make clean
make

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

编译完成之后，还需使用screen命令重定向串口至伪终端才能进行相应shell的交互操作。编译过程中由专属的elf文件对相应的Makefile进行连接并最后统一输出至output文件夹中。

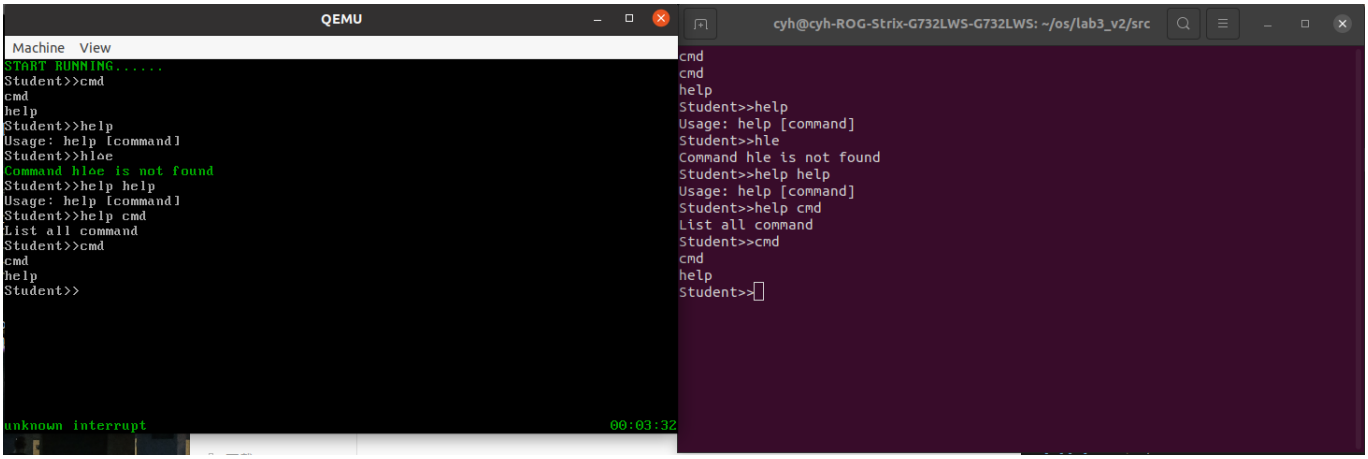


上图为编译过后的部分.o文件结构。

实验结果

运行后的实验结果如下：

注意运行时还需输入`sudo screen /dev/pts/1`，以重定向至伪终端实现输入操作。



首先，串口重定向是成功的，可以看到回显和指令执行也是成功的。之后，也能够成功应对未知中断（这里测试的是对于键盘的中断相应，可见左下角显现出了unknown_interrupt字样）。最后是时钟右下角的时钟，正常显示。

实验中遇到的问题

字符串的处理问题，在没有string.h库的支持下，需要自己寻找或者写出相应的字符串对比，复制函数。此处我采取的办法是从标准字符串处理函数vsprintf.c寻找相应函数strcmp、strcpy，可以从头文件中体现出来。

```
int strlen(const char * str);
int strncmp(const char *first,const char *last,int count);
unsigned long strnlen(const char *s, int count);
int strcmp(const char *source,const char *dest);
```