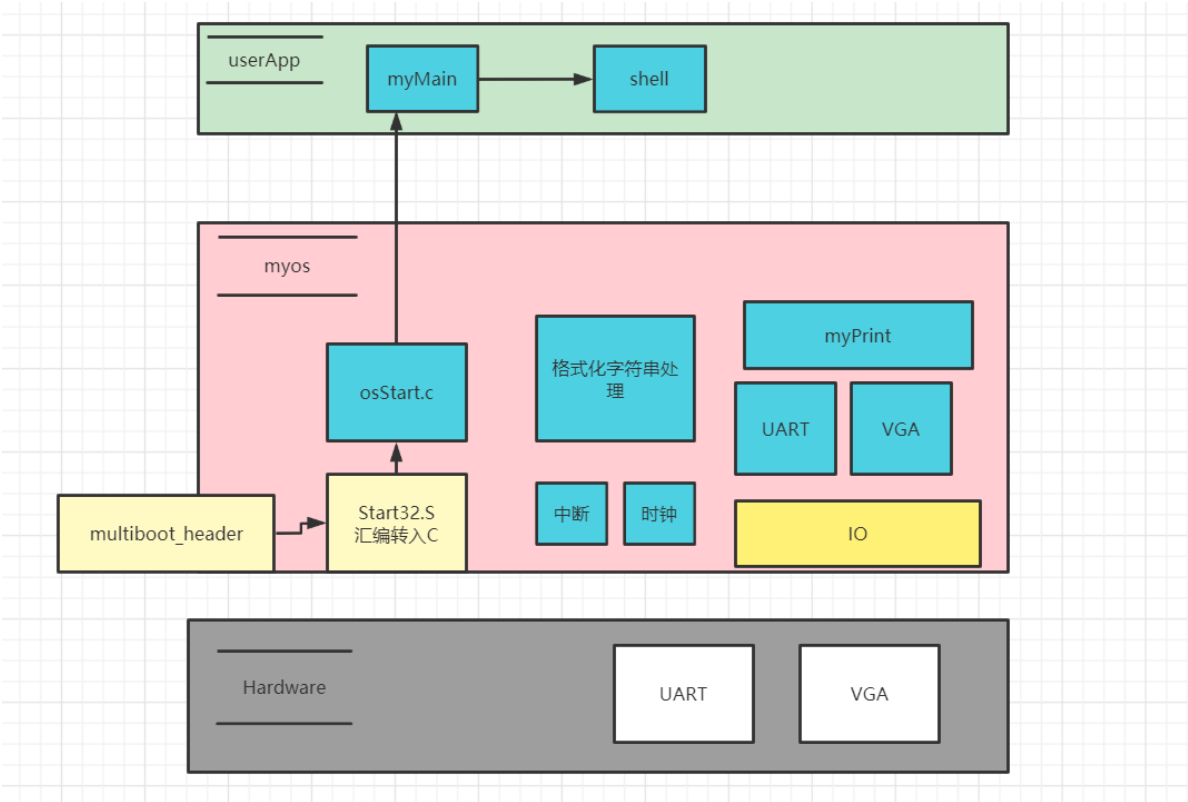
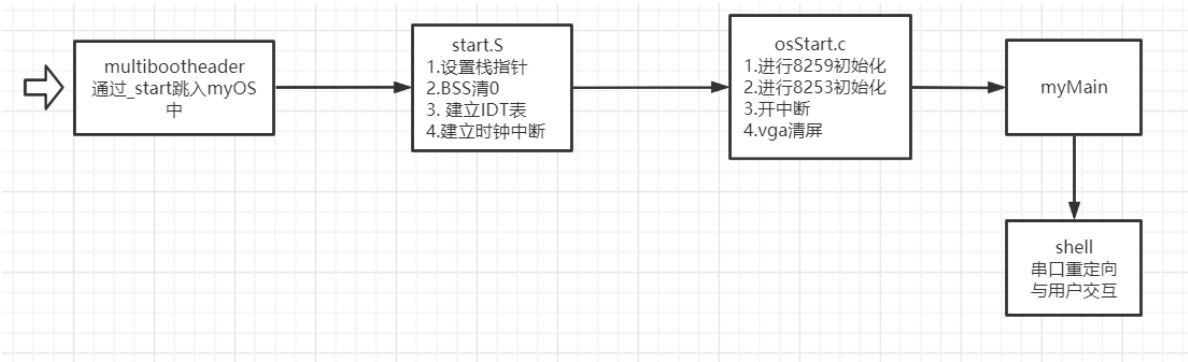
lab3实验报告

一、软件框图



二、主流程

1.流程图



2.文字说明

计算机开机之后，经过一些初始化步骤后，首先读取multibootheader文件，在校验对接完成之后，结尾处将借助myOS提供的_start 入口跳入myOS中。

进入myOS中首先执由汇编语言构成的start.S文件，这个文件会进行设置栈和将BSS 段清0两个操作，为下面的C语言执行提供必要的环境。接着进行IDT表的建立工作，为256个中断向量描述符开辟空间，初始化IDTR，并将每个向量均指向缺省函数。之后正式建立时间中断向量，跳入osStart.c中

进入osStart.c中调用库函数进行初始化工作（包括对8259，8253的硬件初始化），初始化完成之后开中断，调用库函数进行VGA清屏，输出开始信息，之后便跳入userApp执行main.c函数。

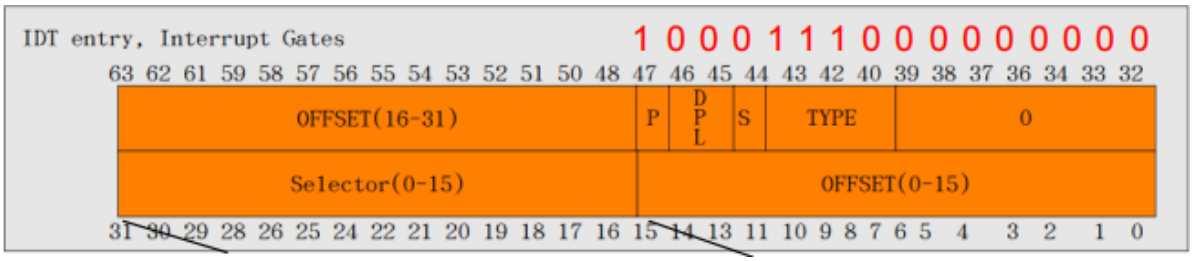
mian.c函数通过调用os提供的函数接口调用startShell()函数启动简易shell程序，启动完成后在终端与用户进行交互，读取用户命令，并进行处理。

三、功能模块

1.IDT的建立

```
# set up the IDT table
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              # edi= mem[IDT]
    mov  $256,%ecx              # ecx is counter
```

这段汇编代码负责拼接好一个下图所示的64位中断描述符



由于32位偏移量分布到两个地方，在使用movl获取偏移量后要用movw进行32位的切割，再经过若干mov指令进行重组，最后得到一个符合格式的描述符。此过程重复进行256次便可建立起IDT，并且其中每一个都指向缺省函数。

2. i8253&i8259初始化

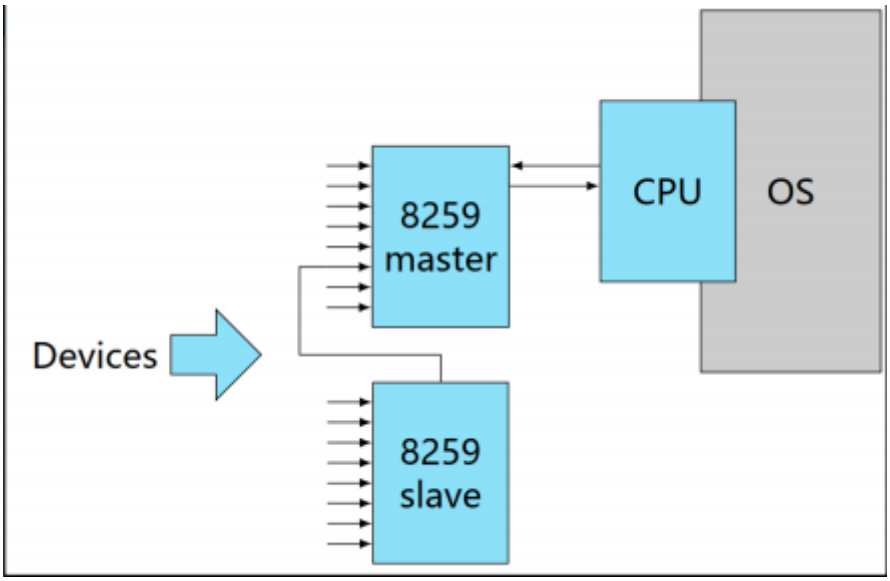
```
//i8253
void init8253(void){
    outb(0x43, 0x34);
    outb(0x40, 0x9B);
    outb(0x40, 0x2E);
}
```

```
#define PORT_M0 0x20
#define PORT_M1 0x21
#define PORT_S0 0xA0
#define PORT_S1 0xA1

void init8259A(void){
    //屏蔽中断源
    // outb(PORT_M1, 0XFF);
    // outb(PORT_S1, 0XFF);
    //初始化主片
    outb(PORT_M0, 0X11);
    outb(PORT_M1, 0X20);
    outb(PORT_M1, 0X04);
    outb(PORT_M1, 0X3);
    //初始化从片
```

```
    outb(PORT_S0, 0X11);
    outb(PORT_S1, 0X28);
    outb(PORT_S1, 0X02);
    outb(PORT_S1, 0X01);
}
```

i8259的工作原理如图所示：



初始化过程首先关中断，然后对主从片对应端口输入相应数据进行初始化。

对i8253的初始化同理，向相应端口输入相应的数据即可。

3. tick与wallClock的建立

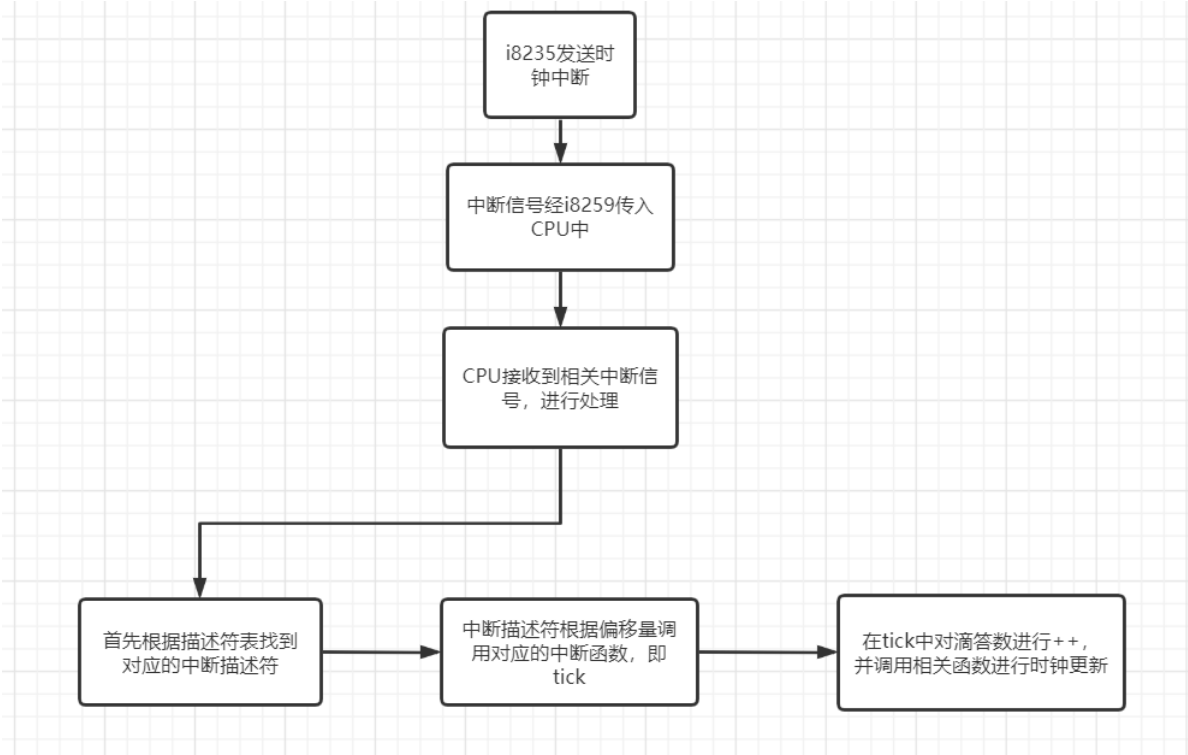
tick段代码如下：

```
// 宏参来代表HMS分别需要多少tick
#define H_TICKNUM 360000
#define M_TICKNUM 6000
#define S_TICKNUM 100
int system_ticks = 0;
int HH,MM,SS;

/*由 i8253 引起的时钟中断而引起的中断子程序
time_interrupt 处理中调用tick 函数*/

void tick(void){
    int temp1, temp2;
    system_ticks++;
    //对HH,MM,SS进行处理
    if(system_ticks == 24 * H_TICKNUM)
        system_ticks = 0;
    HH = system_ticks / H_TICKNUM;
    temp1 = system_ticks % H_TICKNUM;
    MM = temp1 / M_TICKNUM;
    temp2 = temp1 % M_TICKNUM;
    SS = temp2 / S_TICKNUM;
    oneTickUpdateWallClock(HH,MM,SS);
    return;
}
```

在每一次时钟中断时均会调用tick函数，tick的调用流程如下：



tick对时钟的更新策略为根据滴答数分别计算出系统运行的时，分，秒，调用相关函数进行显示。

wallClock代码如下：

```
typedef unsigned short int int16;

int col;

void (*wallClock_hook)(int, int, int) = 0;

void oneTickUpdatewallClock(int HH, int MM, int SS){
    if(wallClock_hook) wallClock_hook(HH,MM,SS);
}

//决定设置墙钟所用的函数
void setwallClockHook(void (*func)(int, int, int)) {
    wallClock_hook = func;
}

void setwallClock(int HH,int MM,int SS){
    //向VGA的右下角输出时钟
    //要将显示格式设置为00：00：00
    if(HH <= 9)
        showMsg(24, 71, 0x4, "0%d:",HH);
    else
        showMsg(24, 71, 0x4, "%d:",HH);
    if(MM <= 9)
        showMsg(24, 74, 0x4, "0%d:",MM);
    else
        showMsg(24, 74, 0x4, "%d:",MM);
    if(SS <= 9)
        showMsg(24, 77, 0x4, "0%d",SS);
    else
        showMsg(24, 77, 0x4, "%d",SS);

}
```

```
//获得墙钟时间
void getWallClock(int *HH,int *MM,int *SS){
    //获取时钟
    col = 71;
    (*HH) = getTimesegment();
    col++;
    (*MM) = getTimesegment();
    col++;
    (*SS) = getTimesegment();
}
```

setWallClock()的实现流程是调用printk中的showMsg()函数进行字符串处理，再调用vga的showMessage往特定位置显示字符串。获取墙钟是通过获取VGA对应显存位置数据进行反字符串化处理。

关于hook机制的思考

> hook机制本质上是给了用户修改某个策略下机制的具体实现方式。hook就像一个程序钩子，可以在相应的位置挂上不同的代码。具体到本次实验中：
>
> 首先在myMain函数中进行setWallClockHook操作，如果用户无修改的需要，钩子将挂上系统默认的内核实现方式setWallClock。若有修改需要则传入mySetWallClock函数。
>
> 在wallClock.c中，使用函数指针指向传入的函数，可能是setWallClock或者mySetWallClock。这个时候真正实现了hook，用一个函数指针充当钩子挂上具体的实现函数。并由其它函数调用挂载函数实现最终功能。
>
> 这个过程体现了机制与策略相分离的原则。

4.shell的实现

数据结构如下：

```
typedef struct myCommand
{
    char name[80];
    char help_content[200];
    int (*func)(int argc, char (*argv)[8]); //函数指针
} myCommand;
```

每个命令包括该命令的名称，命令的帮助文档，执行命令要调用的函数。

读入并处理命令的代码段如下：

```
        BUF_len = 0;
        myPrintk(0x07, "Student>>");
        //从终端获得命令
        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1
        }
        myPrintf(0x07, " -pseudo_terminal\n");

        //从BUF数组中提取相应的argc和argv参数
        i = 0; j = 0;
        while(i < BUF_len)
```

```
        {
            k = 0;
            j ++;
            while (i < BUF_len && BUF[i] != ' ')
            {
                argv[j-1][k++] = BUF[i++];
            }
            argv[j-1][k] = '\0';
            while (i < BUF_len && BUF[++i] == ' ');//处理多空格
        }
        argc = j;
        //寻找相应的myCommand实例
        i = findCommand(argc, argv[0]);
        //调用相关函数
        if(i != -1)
            cmd_list[i].func(argc, argv);
        else
            myPrintf(0x07,"command error!\n");
```

shell的工作流程为从终端读入用户输入的命令，解析该命令，调用相应的函数执行命令。

四、源代码说明

1.目录组织

```
$ tree
.
|-- Makefile
|-- multibootheader
|   `-- multibootHeader.S
|-- myOS
|   |-- Makefile
|   |-- dev
|   |   |-- Makefile
|   |   |-- i8253.c
|   |   |-- i8259A.c
|   |   |-- uart.c
|   |   `-- vga.c
|   |-- i386
|   |   |-- Makefile
|   |   |-- io.c
|   |   |-- irq.S
|   |   `-- irqs.c
|   |-- include
|   |   |-- i8253.h
|   |   |-- i8259A.h
|   |   |-- io.h
|   |   |-- irqs.h
|   |   |-- myPrintk.h
|   |   |-- mymath.h
|   |   |-- mystring.h
|   |   |-- tick.h
|   |   |-- uart.h
|   |   |-- vga.h
|   |   |-- vsprintf.h
|   |   `-- wallClock.h
|   |-- kernel
```

```
|   |   |-- Makefile
|   |   |-- tick.c
|   |   `-- wallClock.c
|   |-- lib
|   |   |-- Makefile
|   |   |-- myPrintk.c
|   |   |-- mymath.c
|   |   |-- mystring.c
|   |   `-- vsprintf.c
|   |-- myOS.ld
|   |-- osStart.c
|   |-- start32.S
|   |-- userInterface.h
|   |-- userApp
|       |-- main.o
|       `-- startShell.o
|-- source2run.sh
`-- userApp
    |-- Makefile
    |-- main.c
    `-- startShell.c
```

2.Makefile组织

```
.
├── MULTI_BOOT_HEADER
| └── output/multibootheader/multibootHeader.o
├── MYOS_OBJS
| ├── 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
| ├── KERNEL_OBJS
| | ├── output/myOS/kernel/tick.o
| | └── output/myOS/kernel/wallClock.o
| ├── LIB_OBJS
| | ├── output/myOS/lib/vsprintf.o
| | |—output/myOS/lib/myPrintk.o
| | ├── output/myOS/lib/mymath.o
| | └── output/myOS/lib/mystring.o
| ├── output/myOS/osStart.o
| └── output/myOS/start32.o
└── USER_APP_OBJS
 ├── main.o
 └── startShell.o
```

五、代码布局说明

地址空间排布如下：

1. 从1M的内存地址开始，首先放multiboot_header，写入12个字节 。
2. 要求8字节对齐，.text代码从16字节开始。
3. 放完.text后，进行16字节对齐。接着放入.data数据。
4. 再次进行16字节对齐后为全局变量分配空间。
5. 16字节对齐后，最后为栈分配空间。进行512字节对齐。

在.data数据段对IDT的分配代码如下

```
/* ===== data ========================== */
.data
# IDT
    .p2align 4
    .globl IDT
IDT:
    .rept 256
    .word 0,0,0,0
    .endr
idtptr:
    .word (256*8 - 1)
    .long IDT
```

.p2align 4的意思为按照2的4次方进行对齐，即16字节对齐。

关于.S中的.p2align 4与.ld中ALIGN()的区别和联系

> .ld中的对齐方式是在段层面，它只作用于该段最开始的数据。
>
> 而每个段都是有许多部分构成的，如data段由IDT与其他数据构成。若IDT不在该段的最开始的地方，则无法保证其16字节对齐，所以要在.S相应位置加入.p2align 4描述内存分布，保证该部分被16字节对齐。

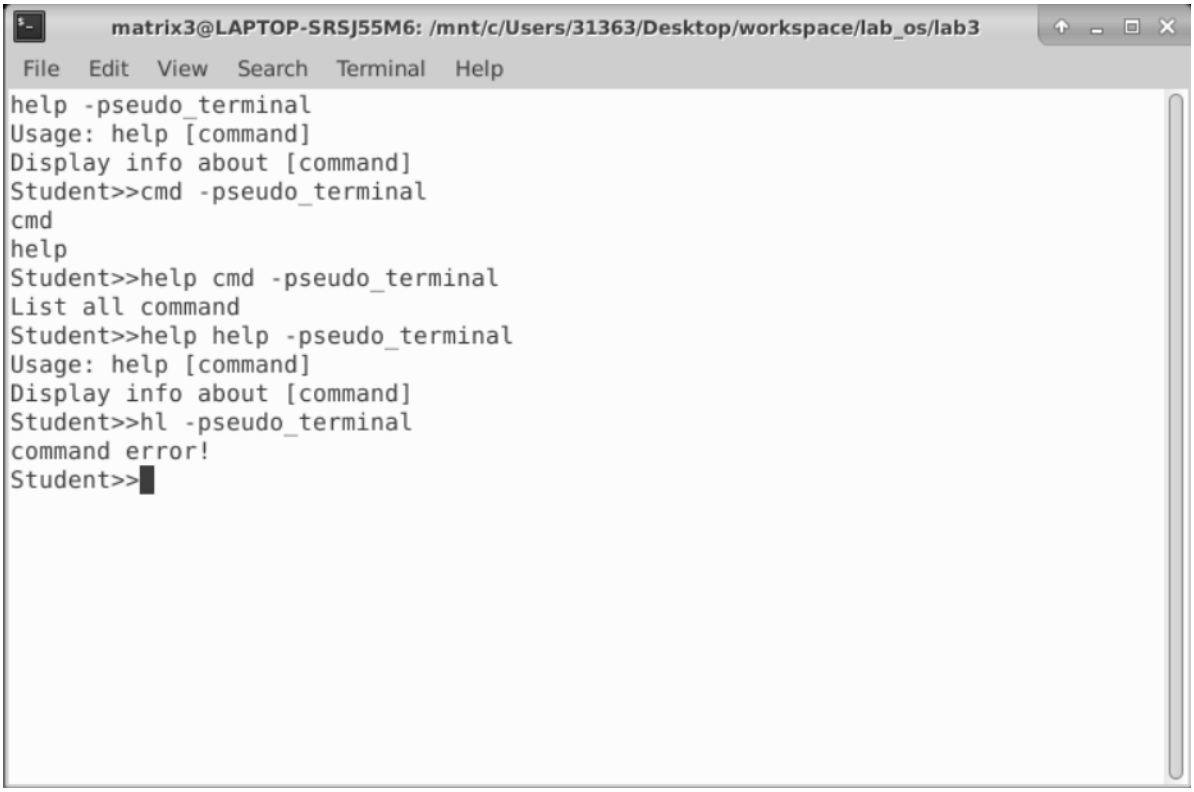
对VGA显存部分，每两个字节间放一个字符，第一个放到0xB8000,第二个从0xB8002开始。

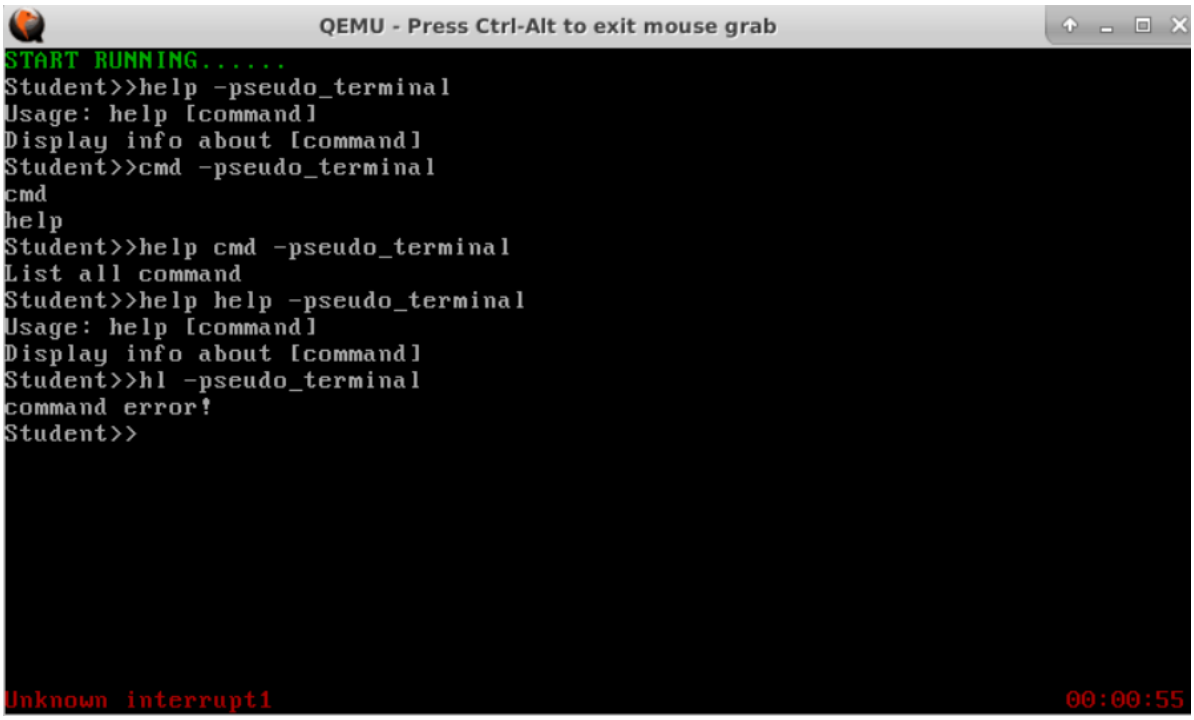
六、编译过程说明

具体操作为直接运行 source2run.sh 脚本，进行 make和串口重定向

编译原理为: 通过Makefile 将S 和.c 文件转换为.o 文件，利用.o文件生成myOS.elf。

七、运行和运行结果说明





运行结果符合要求。

八、遇到的问题

在实现一些基础功能时发现需要实现一些库函数的功能，于是写了mymath.c，mystring.c库函数，之后还会继续扩充。