实验报告

【实验题目】

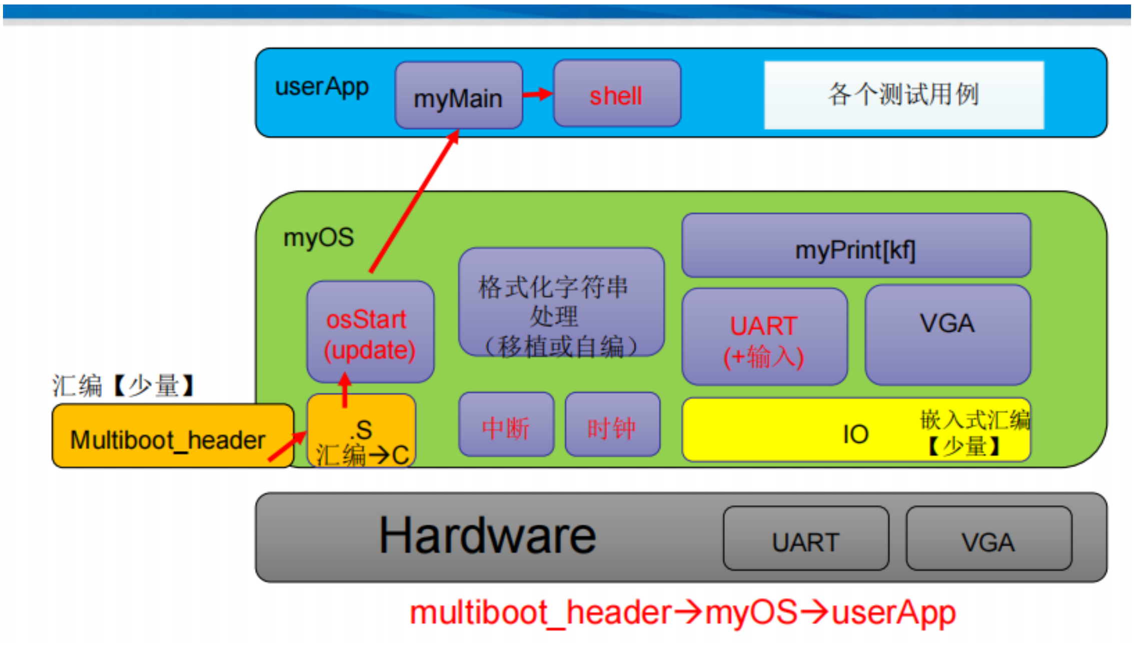
shell&interrupt

【实验要求】

```
1. 实现简单的shell程序，提供cmd和help命令，允许注册新的命令
2. 中断机制和中断控制器i8259A初始化
3. 时钟i8253和周期性时钟中断
4. VGA输出的调整：基于时钟中断以及其他中断的输出响应
5. 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
6. 提供脚本完成编译和执行
```

【实验原理】

- 软件架构和功能



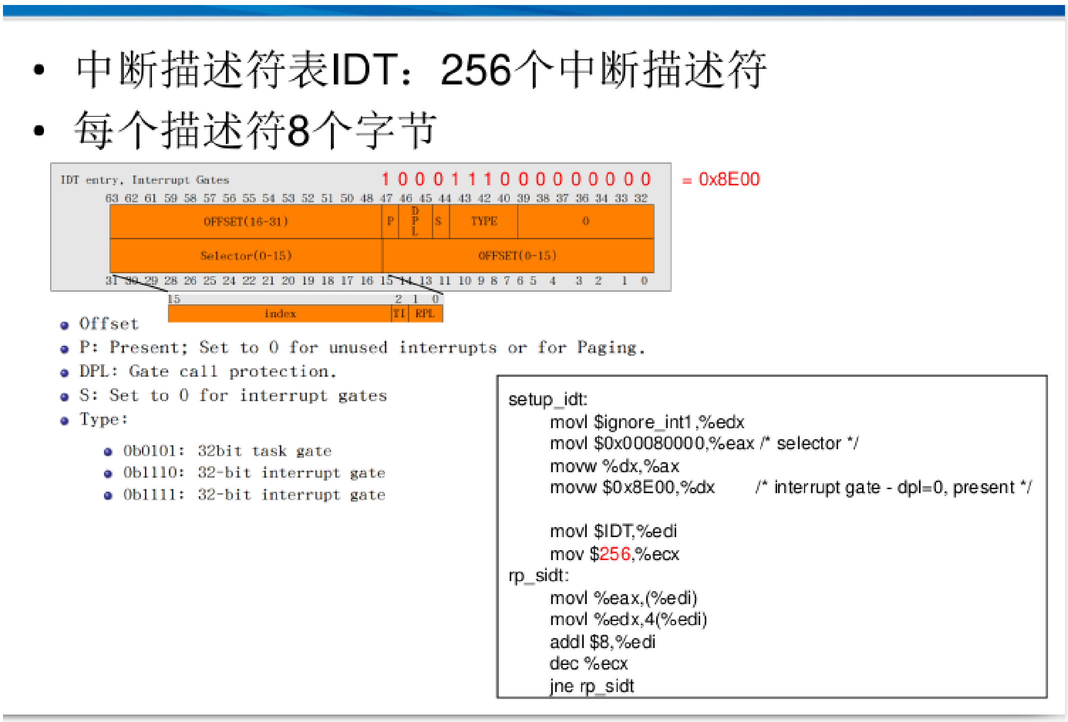

```
软件整个生命周期涉及三个层次：硬件层，操作系统层，用户应用层，其功能分别如下：
    1. Hardware层：
    ① 基于UART和VGA的IO功能
    ② 基于PIC i8259可中断控制器的中断处理功能
    ③ 基于PIT i8253可编程间隔定时器的定时功能
    2. myOS层：
    ① 系统初始化并为C语言准备环境：栈的建立以及BSS段的0初始化，为C程序的执行
       提供运行环境
    ② 硬件底层汇编的C语言函数封装库：UART和VGA的IO接口函数库，开关中断的接口函数
    ③ IDT的初始化及中断处理程序
    ④ 基于定时器的时钟显示模块
    3. UserApp层：
    实现shell终端：接受相关指令输入并执行
```

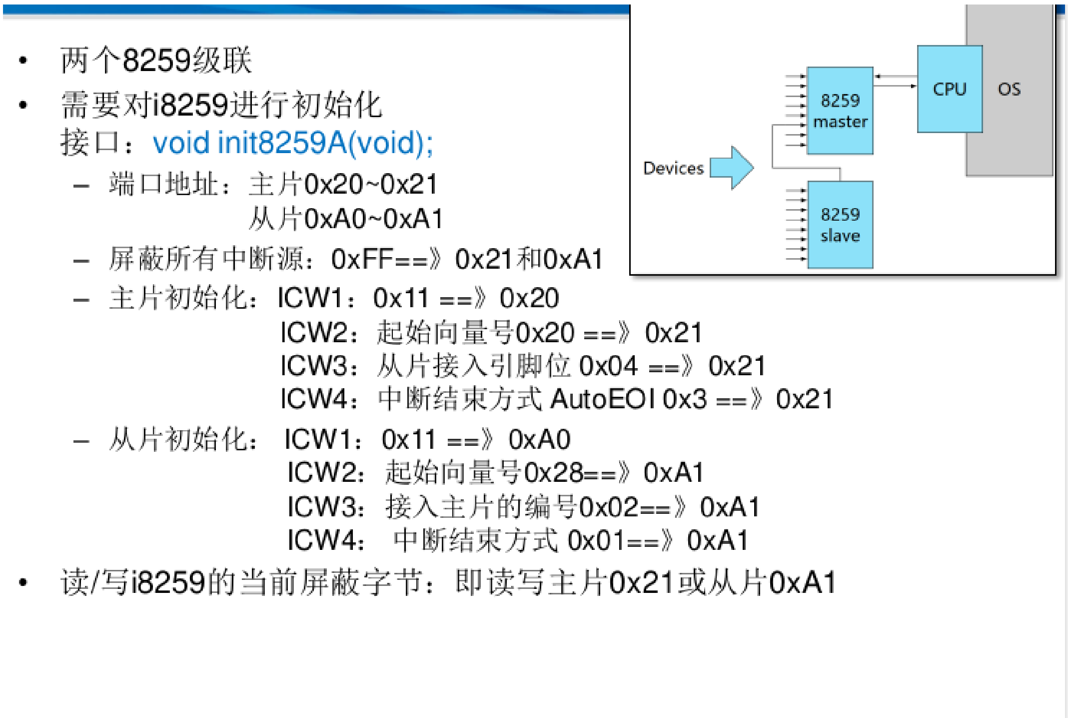
- 软件执行流程

- 主要功能模块及其实现

1. 中断处理模块

> 中断的处理需要软硬件协同，硬件产生相应的中断信号并调用软件预先设定好的中断处理程序，具体要求操作系统在内存的某个特定位置设定好IDT，并在系统初始化时初始化PIC，IDT与PIC的设定规则固定，具体如下：





- IDT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

    call setup_time_int_32

# Transfer control to main
to_main:
    call    osStart

shut_down:
    jmp shut_down    # Never here

    .p2align 4
time_interrupt:
    cld
    pushf
    pusha
    call tick
    popa
    popf
```

```
    iret

    .p2align 4
ignore_int1:
    cld
    pusha
    call ignoreIntBody
    popa
    iret


# ret /* if do not set timer*/
setup_time_int_32:
    movl $time_interrupt,%edx
    movl $0x00080000,%eax /* selector: 0x0010 = cs */
    movw %dx,%ax
    movw $0x8E00,%dx /* interrupt gate - dpl=0, present */
    movl $IDT,%edi
    addl $(32*8), %edi
    movl %eax,(%edi)
    movl %edx,4(%edi)
    ret


/* ===== data =========================== */
.data
# IDT
    .p2align 4
    .globl IDT
IDT:

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

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

- PIC的初始化

```
#include "io.h"

void init8253(void){
    //你需要填写这里
    const short int cnt = 11932;
    outb(0x43,0x34);
    outb(0x40,cnt);
    outb(0x40,cnt>>8);
    unsigned char mask1 = inb(0x21);
    outb(0x21,mask1&(0xfe));
    unsigned char mask2 = inb(0xa1);
    outb(0xa1,mask2&(0xfe));
}
```

- 中断处理程序

```
#汇编部分
time_interrupt:
    cld
    pushf
    pusha
    call tick
    popa
    popf
    iret

    .p2align 4
ignore_int1:
    cld
    pusha
    call ignoreIntBody
    popa
    iret
```

```
#C语言部分
void ignoreIntBody(void){
    myPrintk(0x07,"Unknown interrupt\n\0");
}

void tick(void){
    //你需要填写这里
    system_ticks++;
    int show_ticks = system_ticks/100;
    SS = show_ticks %60;
    MM = ((show_ticks -SS)/60)%60;
    HH = ((show_ticks-SS-MM*60-SS)/3600)%24;
    oneTickUpdateWallClock(HH, MM, SS);

    return;
}
```

2. 时钟显示模块

- PIT的初始化

```
PIT作为硬件的初始化方法固定，具体如下:
```


```
#include "io.h"

void init8259A(void){
    //你需要填写这里
    outb(0x21,0xff);
    outb(0xa1,0xff);
    outb(0x20,0x11);
    outb(0x21,0x20);
    outb(0x21,0x04);
    outb(0x21,0x3);
    outb(0xa0,0x11);
    outb(0xa1,0x28);
    outb(0xa1,0x02);
    outb(0xa1,0x1);
}
```

- 时钟的显示刷新

```
此处关于时钟的显示刷新采用hook机制，个人具体理解是：
    在设计初期不确定要实现的具体功能，如果直接设计出功能函数，后期如果不符合要求，就要删去重做，浪费不必要的时间，而hook机制允许预先设定一个容器，支持后期在容器内放入需要的处理函数，前期不影响程序的完备性，具体在本次实验中预先建立一个函数指针并指空，后期系统初始化时为其指定合适的处理函数(setWallClock)
```

```
#include "tick.h"
extern void oneTickUpdateWallClock(int HH, int MM, int SS);

void tick(void){
    //你需要填写这里
```

```
    system_ticks++;
    int show_ticks = system_ticks/100;
    SS = show_ticks %60;
    MM = ((show_ticks -SS)/60)%60;
    HH = ((show_ticks-SS-MM*60-SS)/3600)%24;
    oneTickUpdateWallClock(HH, MM, SS);

    return;
}

#include "wallClock.h"

void (*wallClock_hook)(int, int, int) = 0;
void oneTickUpdateWallClock(int HH, int MM, int SS){
    if(wallClock_hook) wallClock_hook(HH,MM,SS);
}

void setWallClockHook(void (*func)(int, int, int)) {
    wallClock_hook = func;
}

void setWallClock(int HH,int MM,int SS){
    //你需要填写这里
    char *ptr = (char*)0xB8F00;

    *ptr = (HH/10) +48;
    *(ptr+1) = 0x07;
    *(ptr+2) = (HH%10) +48;
    *(ptr+3) = 0x07;

    *(ptr+4) = ':';
    *(ptr+5) = 0x07;

    *(ptr+6)= (MM/10) +48;
    *(ptr+7) = 0x07;
    *(ptr+8) = (MM%10) +48;
    *(ptr+9) = 0x07;

    *(ptr+10) = ':';
    *(ptr+11) = 0x07;

    *(ptr+12) = (SS/10) +48;
    *(ptr+13) = 0x07;
    *(ptr+14) = (SS%10) +48;
    *(ptr+15) = 0x07;
    return;
}

void getWallClock(int *HH,int *MM,int *SS){
    //你需要填写这里
    char *ptr = (char*)0xB8F00;
    *HH = ((*ptr) -48)*10 + ((*(ptr+2))-48);
    *MM = ((*ptr+6) -48)*10 + ((*(ptr+8))-48);
    *SS = ((*ptr+12) -48)*10 + ((*(ptr+14))-48);
    return;
}
```

3. shell终端

此次实现基于串口接受用户输入，解析命令并执行的简单shell终端，shell程序具体执行流程如下：

startShell执行流程

1. 获取串口输入到输入缓冲区，并回显到vga和uart输出
2. 解析输入，获取有效命令及参数
    （1）curtail 删去先导的空白字符(' '，'\r','\n')
    （2）parse_command_from_buf 以空格作为分隔符将输入分为命令,输入参数
3. 确定有效命令并执行
    （1）get_cmd_index 利用compare_str在已注册命令中按字符串比较进行查找，确定有效命令，并返回命令编号
    (2) execute_command 根据命令编号调用相应命令的处理函数完成功能
4. 根据是否执行exit命令确定是否循环此流程

```
#include "io.h"
#include "myPrintk.h"
#include "uart.h"
#include "vga.h"
#include "i8253.h"
#include "i8259A.h"
#include "tick.h"
#include "wallClock.h"

#define MaxCmdNum 20

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

typedef struct cmdList {
    myCommand cmds[MaxCmdNum];
    int cmd_num;
} cmdList;

int func_cmd(int argc, char (*argv)[8]);
int func_help(int argc, char (*argv)[8]);
int func_exit(int argc, char (*argv)[8]);
int func_divZero(int argc, char (*argv)[8]);


cmdList cmd_list;
myCommand cmd={"cmd\0","List all command\n\0",func_cmd};
myCommand help={"help\0","Usage: help [command]\n\0Display info about
[command]\n\0",func_help};
myCommand exit = {"exit\0","exit shell\n\0",func_exit};
myCommand divZero = {"divZero\0","test divide 0 interrupt\n\0",func_divZero};

int func_cmd(int argc, char (*argv)[8]){
    for(int i=0;i<cmd_list.cmd_num;i++){
        myPrintk(0x07,(cmd_list.cmds)[i].name);
        myPrintk(0x07," ");
    }
```

```
    myPrintk(0x07,"\n");
    return 0;
}

int func_help(int argc, char (*argv)[8]){
    if(argc > 2)
        myPrintk(0x07,"The format is wrong,please check again!\n");
    else if(argc == 1)
        myPrintk(0x07,(cmd_list.cmds)[1].help_content);
    else {
        int i = get_cmd_index(cmd_list,argv[1]);
        myPrintk(0x07,(cmd_list.cmds)[i].help_content);
    }
     return 0;
}

int func_exit(int argc,char (*argv)[8]){
    return 0;
}

int func_divZero(int argc,char (*argv)[8]){
    int a = 1/0;
    return 0;
}

void startShell(void){
//我们通过串口来实现数据的输入
    char BUF[256]; //输入缓存区
    int BUF_len=0;  //输入缓存区的长度

    (cmd_list.cmds)[0] = cmd;
    (cmd_list.cmds)[1] = help;
    (cmd_list.cmds)[2] = exit;
    (cmd_list.cmds)[3] = divZero;
    cmd_list.cmd_num = 4;

    int argc;
    char argv[8][8];
    int cont = 1;

    do{
        BUF_len=0;
        myPrintk(0x07,"Student>>\0");
        while((BUF[BUF_len]=uart_get_char())!='\r'){
            uart_put_char(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
        }
        BUF[BUF_len] = 0;
        // myPrintk(0x07,BUF);
        uart_put_chars(" -pseudo_terminal\0");
        append2screen(BUF,0x07);
        append2screen(" -pseudo_terminal\0",0x07);
        myPrintk(0x07,"\n");
        parse_command_from_buf(BUF,&argc,argv);
        // myPrintk(0x07,BUF);
        cont = execute_command(cmd_list,argc,argv);
```

```
        //OK,助教已经帮助你们实现了“从串口中读取数据存储到BUF数组中”的任务，接下来你们要
做
        //的就是对BUF数组中存储的数据进行处理(也即，从BUF数组中提取相应的argc和argv参
        //数)，再根据argc和argv，寻找相应的myCommand ***实例，进行
***.func(argc,argv)函数
        //调用。

        //比如BUF中的内容为 “help cmd”
        //那么此时的argc为2 argv[0]为help argv[1]为cmd
        //接下来就是 help.func(argc, argv)进行函数调用即可

    }while(cont);

}

void parse_command_from_buf(char* buf,int *argc,char (*argv)[8]){
    curtail(buf);
    int i,j;
    // myPrintk(0x07,buf);
    if(buf[0] == 0){
        *argc = 0;
        argv[0][0] = 0;
    }
    else {
        *argc = 0;
        int index = 0;
        for(i=0;buf[index]!=0;i++){
            for(j=0;buf[index]!=' ' && buf[index] != 0;j++){
                argv[i][j] = buf[index];
                index++;
            }
            if(buf[index]==' ')
                index++;
            (*argc) ++;
            argv[i][j] = 0;
        }
    }
}

void curtail(char* buf){
    int i = 0;
    while(buf[i]==' ' || buf[i] == '\r' || buf[i] == '\n' || buf[i] == '\t' )
        i++;
    int j = 0;
    while(buf[i+j]!=0){
        buf[j] = buf[i+j];
        j++;
    }
}

int execute_command(cmdList cmd_list,int argc,char (*argv)[8]){
    int i = get_cmd_index(cmd_list,argv[0]);
    // myPrintk(0x07,argv[0]);
    // myPrintk(0x07,"\n");
    // myPrintk(0x07,"%d\n",i);
    if(i<0)
        myPrintk(0x07,"command not found!\n");
    else
```

```
        ((cmd_list.cmds)[i]).func(argc,argv);
    if(i == 2)
        return 0;
    else
        return 1;
}

int get_cmd_index(cmdList cmd_list,char* cmd_name){
    int i = 0;
    // myPrintk(0x07,cmd_name);
    while(i < cmd_list.cmd_num){
        if(compare_str(((cmd_list.cmds)[i]).name,cmd_name)==0)
            break;
        else
            i++;
    }
    // myPrintk(0x07,"%d\n",i);
    if(i==cmd_list.cmd_num)
        return -1;
    else
        return i;
}

int compare_str(char* str1,char* str2){
    int i = 0;
    if(str1[0]==0 && str2[0] == 0) return 0;
    else if(str1[0] == 0 || str2[0] == 0) return -1;
    else {
        for(;str1[i]!=0;i++){
            if(str1[i] != str2[i])
                break;
        }
        return str1[i] - str2[i];
    }
}
```

- 代码组织及其实现
    - 目录组织

因文件过多，暂时不包含output文件夹下内容

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，设定IDT的汇编级程序
dev       提供硬件层UART,VGA的使用封装接口,i8253,i8259A的初始化接口
i386      提供基于i386架构的底层硬件IO的接口,中断处理程序
include   C程序的头文件
printk    myPrintk,vsprintf函数的实现
kernel    时钟显示模块的实现
```

3. userApp

```
此模块存放用户程序，本次实验包含main.c以及startShell.c，提供shell终端服务
```

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
    |   |   ├── 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
```

- 代码布局说明

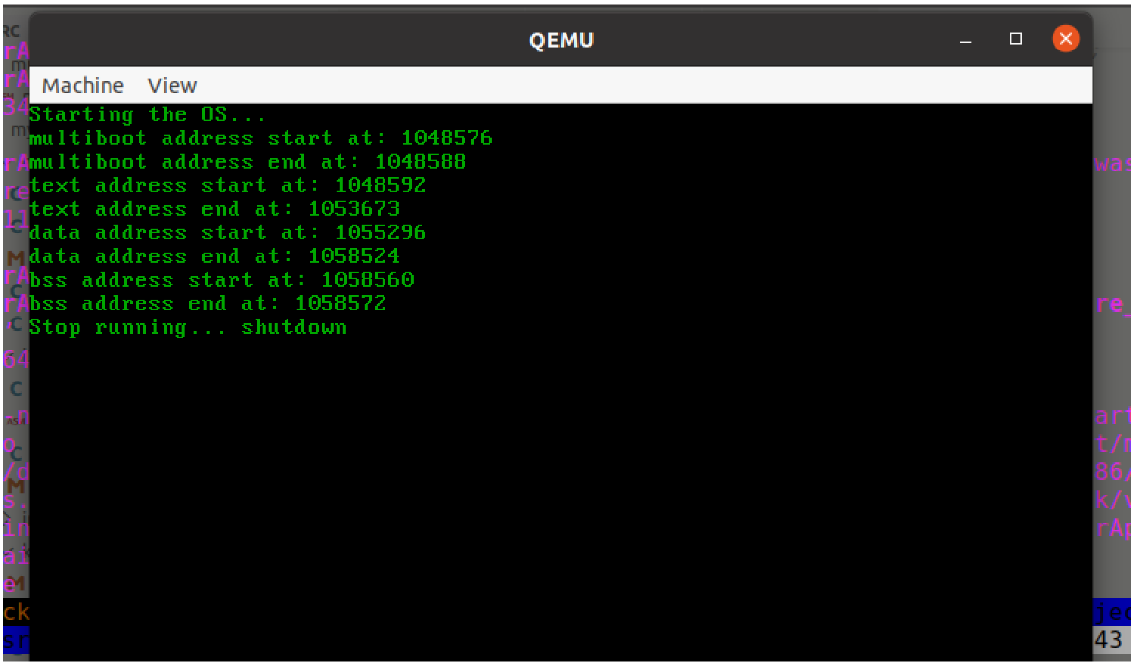
  借助C程序可查看各段的起止位置

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

```c
#include "myPrintk.h"
extern unsigned long __multiboot_start;
extern unsigned long __multiboot_end;
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;
/* 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,"multiboot address start at: %d\n",(unsigned
long)&__multiboot_start);
    myPrintk(0x2,"multiboot address end at: %d\n",(unsigned
long)&__multiboot_end);
    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, "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 = .;
}
```



Section	Offset(Base = 0)
.multiboot_header	0x00100000~0x0010000c
.text	0x00100010~0x001013ce
.data	0x00101a00~0x0010269c
.bss	0x001026c0~0x001026cc

【实验过程】

- 编译过程及运行过程说明

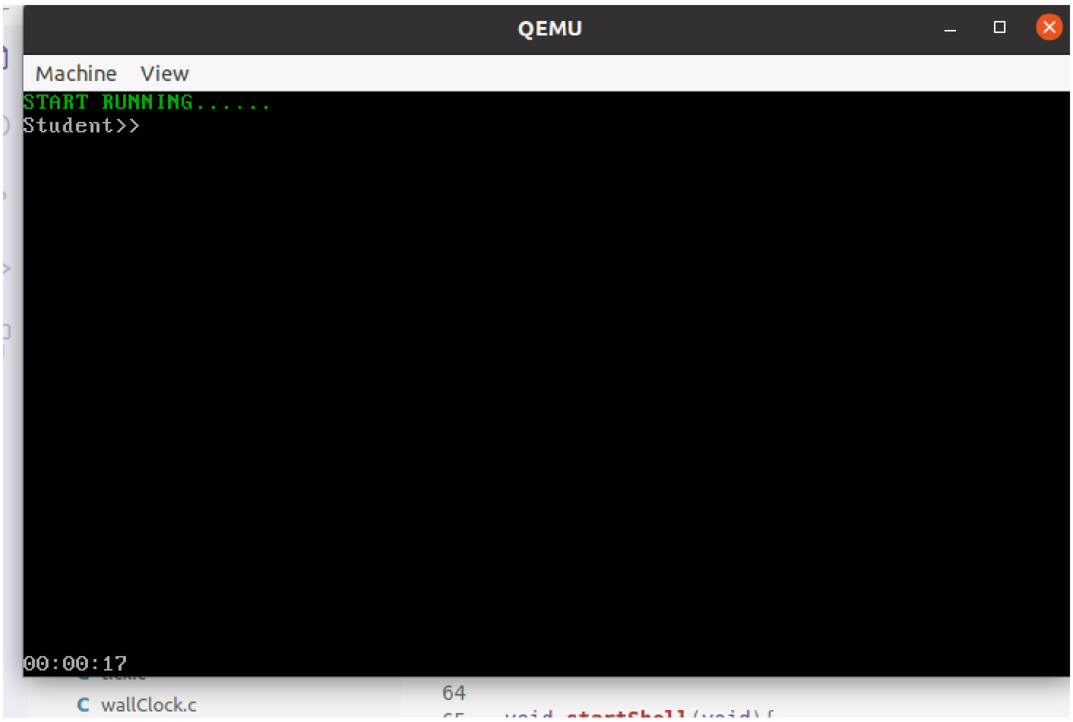
```
直接执行脚本文件source2img.h实现编译链接及运行
```

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=i686-elf-
CROSS_COMPILE=
ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
C_FLAGS =  -m32 -fno-stack-protector -g
INCLUDE_PATH = myOS/include

.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 ${C_FLAGS} -I${INCLUDE_PATH} -c -o $@ $<

clean:
	rm -rf output
```

- 运行测试
  - 时钟显示



  - 执行命令

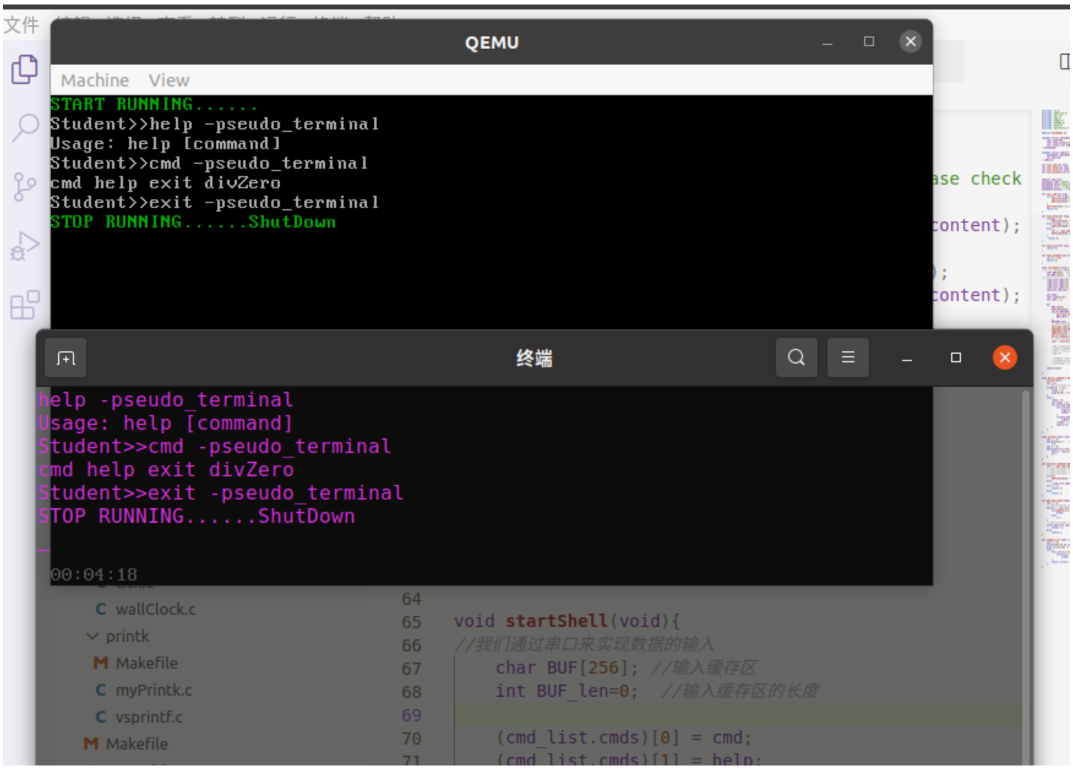
此次共实现四个命令

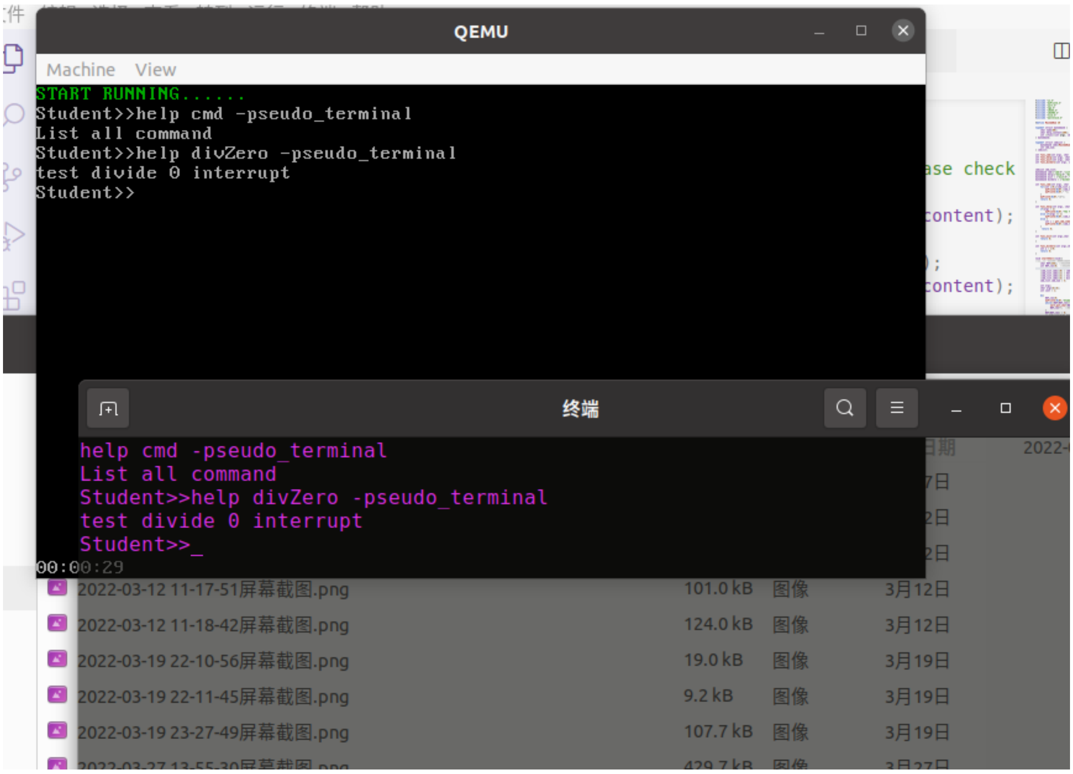
① help 获取相关指令帮助信息

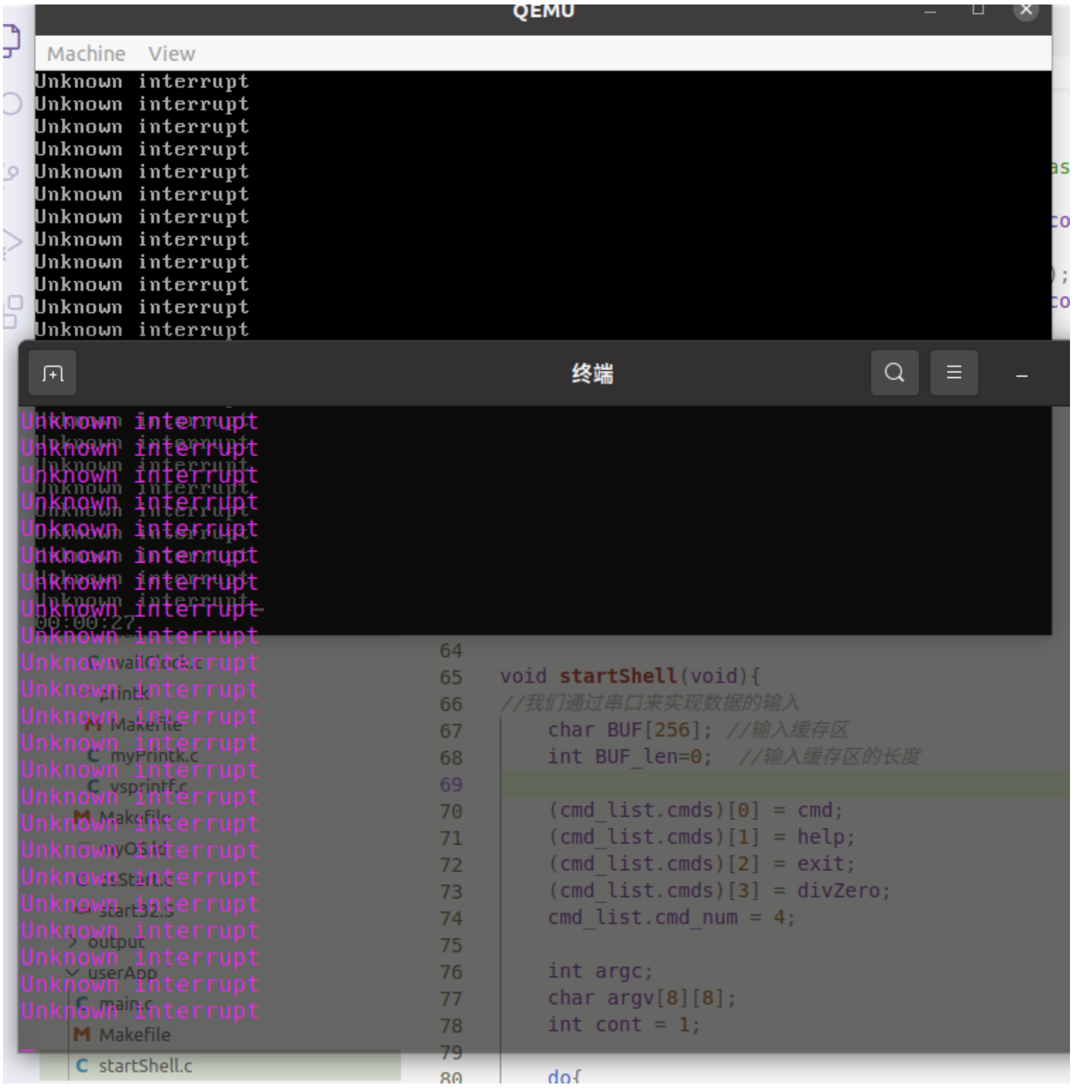
② cmd 打印当前所有注册指令

③ exit 退出终端程序

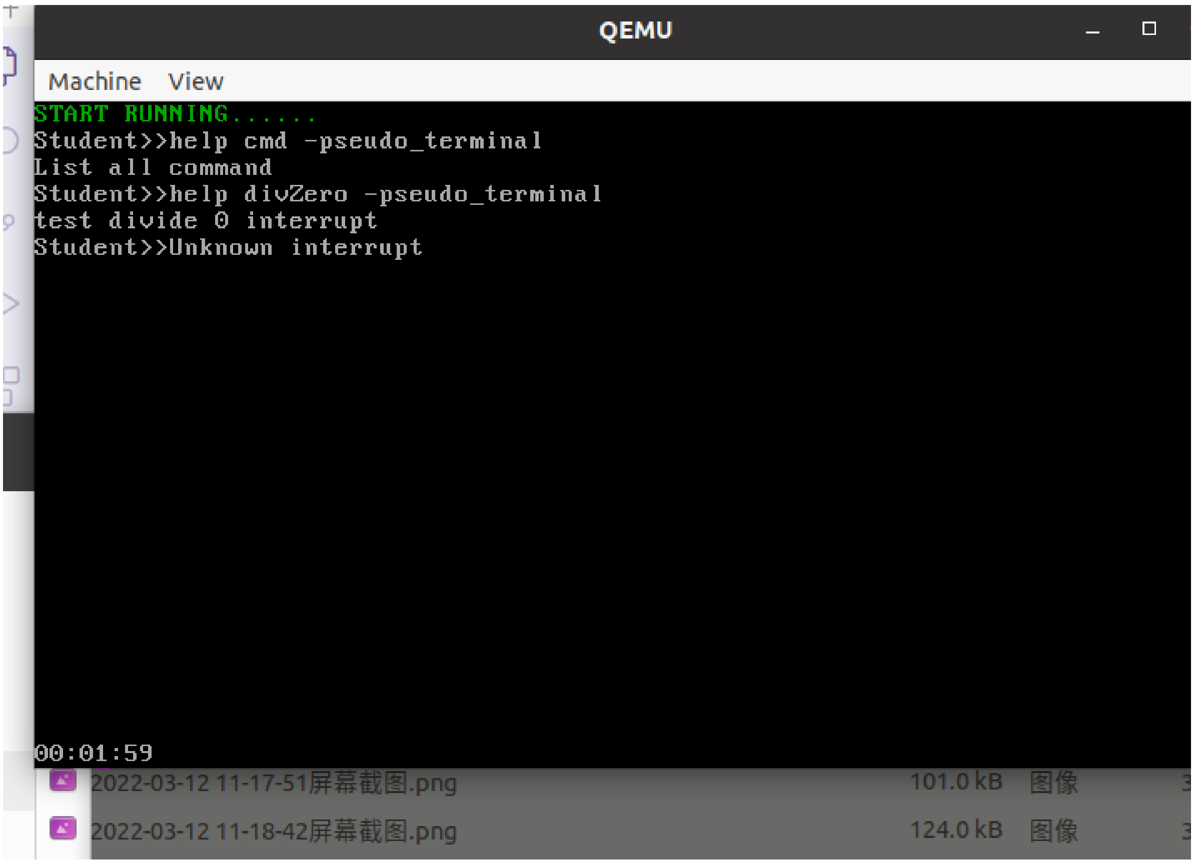
④ divZero 测试除0引起的中断







此处为执行命令divZero导致的除0中断



此处为键盘按键产生的中断

【问题与解决】

- 在完成shell程序时，在程序外初始化变量，无法执行

```
C语言中变量可以在程序外定义，但一旦定义以后无法在赋值
```

- 在解析命令并执行时，输入缓冲区采用覆盖的方式，导致执行多条指令在进行指令比对时受前面输入的影响

```
在每次输入的末尾补0作为本次输入的结束，后面的数据不参与命令的解析执行
```