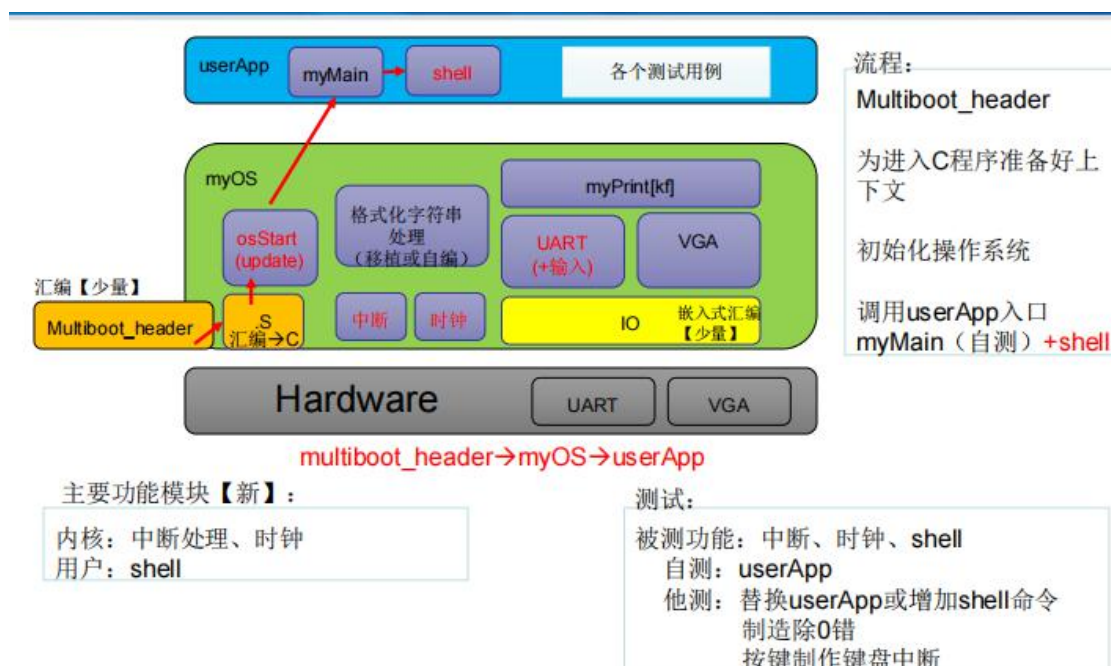


Lab3 时钟中断与 StartShell

1. 实验框架



Part1: uart.c 与 vga.c, 串口与显存读写结合 vsprintf.c 函数生成 myPrintf 函数实现双 I/O

Part2: 可编程间隔定时器定时输出时钟中断经由可编程中断控制器 i8259 生成中断, 查找中断向量表执行对应的中断程序, 本次实验在定时 tick 同时将全局变量 systemticks 自增, 再转化输出为墙钟

Part3: Shell: 伪终端输入字符串分成连续串, 分支处理, 输出相应信息

Part4: C 程序预处理为 i 文件, 编译为 x86 汇编指令(s 文件), 再转化为二进制码(o 文件), 链接为 exe 可执行文件, 由 QUMU 执行

2. 模块流程图 1 (内核: 中断控制, 时钟)

Part1: start32.S

IDT 的建立:

```

99  # IDT
100  .p2align 4
101  .globl IDT
102  IDT:
103
104  .rept 256
105  .word 0,0,0,0
106  .endr
107  idtptr: //idt尾指针
108
109  .word (256*8 - 1)
110  .long IDT
111

```

IDT 表每一项为 $4 \times 2 = 8$ 字节, 共 256 项



47:32 位为属性码，31: 16 为代码段（CS）地址，其余为偏移量（IP）
IDT 的填写：

```

39  setup_idt:
40      movl $ignore_int1,%edx
41      movl $0x00080000,%eax
42      movw %dx,%ax /* selector = 0x0010 = cs */
43      movw $0x8E00,%dx /* interrupt gate - dpl=0, present */
44      movl $IDT,%edi
45      mov $256,%ecx
46
47  rp_sidt:
48      movl %eax, (%edi)           #将ignore_int1地址写入IDT
49      movl %edx, 4(%edi)         #写入属性码8E
50      addl $8,%edi
51      dec %ecx
52      jne rp_sidt
53
54      lidt idtptr
55
56      call setup_time_int_32

```

IDT 前 256 为写入 ignore_int 段地址与属性码 8E;

```

85  setup_time_int_32:              #IDT末尾放入time_int
86      movl $time_interrupt,%edx
87      movl $0x00080000,%eax /* selector: 0x0010 = cs */
88      movw %dx,%ax
89      movw $0x8E00,%dx /* interrupt gate - dpl=0, present */
90      movl $IDT,%edi
91      addl $(32*8), %edi
92      movl %eax, (%edi)
93      movl %edx, 4(%edi)
94      ret

```

IDT 末尾为 time_interrupt 短地址，属性码为 8E;
时钟中断与其他中断程序：

```

66  time_interrupt:
67      cld
68      pushf
69      pusha
70      call tick
71      popa
72      popf
73      iret
74
75      .p2align 4
76  ignore_int1:
77      cld
78      pusha
79      call ignoreIntBody
80      popa
81      iret

```

保护现场——进入中断——恢复现场——返回原地址

Part2:可编程中断控制器 i8259:

可编程中断控制器

PIC i8259

中国科学技术大学
University of Science and Technology of China

- 两个8259级联
- 需要对i8259进行初始化
接口: `void init8259A(void);`
 - 端口地址: 主片0x20~0x21
 从片0xA0~0xA1
 - 屏蔽所有中断源: 0xFF==》0x21和0xA1
 - 主片初始化: ICW1: 0x11 ==》0x20
 ICW2: 起始向量号0x20 ==》0x21
 ICW3: 从片接入引脚位 0x04 ==》0x21
 ICW4: 中断结束方式 AutoEOI 0x3 ==》0x21
 - 从片初始化: ICW1: 0x11 ==》0xA0
 ICW2: 起始向量号0x28==》0xA1
 ICW3: 接入主片的编号0x02==》0xA1
 ICW4: 中断结束方式 0x01==》0xA1
- 读/写i8259的当前屏蔽字节: 即读写主片0x21或从片0xA1

初始化 i8259, 将主片最低位设置为 0 允许时钟中断, 其他位置置 0 允许其他中断, 从片全置 0 为个性化

```

3  void init8259A(void) {
4      outb( 0x21, 0xff);
5      outb( 0xA1, 0xff);
6      outb( 0x20, 0x11);
7      outb( 0xA0, 0x11);
8      outb( 0x21, 0x20);
9      outb( 0xA1, 0x28);
10     outb( 0x21, 0x04);
11     outb( 0xA1, 0x02);
12     outb( 0x21, 0x03);
13     outb( 0xA1, 0x01); //初始化完成
14     outb( 0x21, 0x00); //主片最低位置0,允许时钟中断
15     //其他位置置0,允许其他中断
16     outb( 0xA1, 0xff); //屏蔽从8259所有中断
17 }

```

Part3: 可编程间隔定时器

可编程间隔定时器 PIT: i8253



中国科学技术大学
University of Science and Technology of China

- 需要对PIT: i8253进行初始化，接口：void init8253(void)
 - 端口地址 0x40~0x43;
 - 14,3178 MHz crystal
4,772,727 Hz system clock
1,193,180 Hz to 8253
 - 设定时钟中断的频率为100HZ，分频参数是多少？
 - 初始化序列为：
 - 0x34 ==》端口0x43（参见控制字说明）
 - 分频参数==》端口0x40，分两次，先低8位，后高8位
 - 通过8259控制，允许时钟中断
 - 读取原来的屏蔽字，将最低位置0

```

1  #include "io.h"
2
3  void init8253(void) {
4      outb( 0x43, 0x34);
5      outb( 0x40, 156);
6      outb( 0x40, 46);
7
8

```

初始化后设置频率为 $1193180 / (46 * 256 + 156) \approx 100\text{HZ}$

- Tick（嘀嗒）：周期性时钟中断
 - 频率：100HZ（可以不必是100HZ）
 - 通过对8253编程来触发周期性时钟中断（见8253初始化）

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

- Tick发生时，做些什么

接口： `void tick(void);`

- Tick的维护：
 - 使用一个全局变量来维护tick发生的次数
 - 初始化为0；每次tick，加1
- 随时间变化而进行的维护（包括其他模块所需）
 - 如wallClock

- 少量（思考：hook机制） VS 大量？用户需要？

每 10ms 产生时钟中断查找中断向量进入 time_interrupt 段执行一次 tick 函数

```
1  #include "tick.h"
2  extern void oneTickUpdateWallClock(int HH, int MM, int SS);
3  extern int system_ticks;
4  void tick(void){
5      //你需要填写这里
6      system_ticks++;
7      oneTickUpdateWallClock(HH, MM, SS);
8      return;
9  }
10
11
```

```

1  #include "wallClock.h"
2  int system_ticks;
3  void (*wallClock_hook)(int, int, int) = 0;
4  void oneTickUpdateWallClock(int HH, int MM, int SS) {           //wallClock_hook初始=0:如果MyMain中
5      if (wallClock_hook) wallClock_hook(HH, MM, SS):           //setWallClockHook(setWallClock):执行, WallClock_hook执行
6      }                                                         //更新时wallClock_hook=setWallClock, 机制与策略分离
7
8  void setWallClockHook(void (*func)(int, int, int)) {
9      wallClock_hook = func;
10 }
11
12 void setWallClock(int HH, int MM, int SS) {
13     //你需要填写这里
14     int h1, h2, m1, m2, s1, s2;
15     char clock[100];
16     HH=(system_ticks/100)/3600;
17     MM=((system_ticks/100)%3600)/60;
18     SS=(system_ticks/100)%60;
19     h1= (system_ticks / 100) / 3600 / 10;    h2= (system_ticks / 100) / 3600 % 10;
20     m1= ((system_ticks / 100) % 3600) / 60 / 10;    m2= ((system_ticks / 100) % 3600) / 60 % 10;
21     s1= (system_ticks / 100) % 60 / 10;    s2= (system_ticks / 100) % 60 % 10;
22     //clock[8]=((48+h1), (48+h2), ':', (48+m1), (48+m2), ':', (48+s1), (48+s2));
23     clock[0] = 48 + h1; clock[1] = 48 + h2; clock[2] = ':'; clock[3] = 48 + m1;    //例如48+3为3对应ASCII码
24     clock[4]=48+m2;clock[5]=':';clock[6]=48+s1;clock[7]=48+s2;
25     short *i = (short*) (0xB8000+(24*80+72)*2);
26     char *p;    p=clock;
27     while ((*p)!='\0') {
28         *i=256*2+(*p);
29         p++;i++;
30     }
31     //myPrintk2(0x02, "%d:%d:%d", HH, MM, SS);
32     return;
33 }
34
35 void getWallClock(int *HH, int *MM, int *SS) {
36     //你需要填写这里
37
38     return;
39 }
40
41

```

结合 main.c 函数

```

12 void myMain(void) {
13     setWallClockHook(setWallClock); //
14     //
15     startShell();
16     return;
17 }
18

```

每次 tick,将 setWallClock 函数赋给函数指针 hook,hook 变为 setWallClock,生成墙钟

3. 模块流程图 2(Shell)

助教已经帮我们字符串从串口读入并且放入 BUF 数组并用 BUF_len 记录其长度

Step1: 将 BUF 划分为若干个连续字符串, 放入 argc,argv

```

60
61
62
63
64
65
66
67
68
69
70
71

```

```

int i, j, k;
i = 0; j = 0;
for (k = 0; k < BUF_len; k++) {
    if (BUF[k] == ' ') {
        i++; j = 0;
    }
    else {
        argv[i][j] = BUF[k];
        j++;
    }
}
argc = i + 1;

```

Step2:对 argv 进行判断，为此定义 teststr 函数

```
82 int teststr(char *a, char *b) { //input str
83     int cout=0;
84     while (*b != '\0') {
85         if ((*a) == (*b)) {
86             a++; b++;
87         }
88         else
89         {
90             cout++; a++; b++;
91         }
92     }
93     return ((cout==0));
94 }
95
```

Step3:根据不同的情况调用不同的 func 输出相应内容

```
73     if (teststr(argv[0], "cmd"))
74         cmd.func(argc, argv);
75     else if (teststr(argv[0], "help"))
76         help.func(argc, argv);
77     else myPrintf(0x02, "%s\n", "unknown command");
78 }while(1);
79
80
```

```
19 int func_cmd(int argc, char (*argv)[8]){
20     myPrintf(0x02, "%s\n%s\n", cmd.name, help.name);
21 }
22
23 myCommand cmd={"cmd\0", "List all command\n\0", func_cmd};
24
25 int func_help(int argc, char (*argv)[8]){ // *argv=a[0]=&a[0][0]
26     /*(argv+1)=a[1]=&a[1][0]: *(argv+1)+i=a[1][i];
27     if (argc > 2)
28         myPrintf(0x02, "%s\n", "error:too large parameter number");
29     else
30     {
31         if(teststr(argv[1], "help"))
32             myPrintf(0x02, "%s\n", help.help_content);
33         else if(teststr(argv[1], "cmd"))
34             myPrintf(0x02, "%s\n", cmd.help_content);
35         else myPrintf(0x02, "%s\n", "no such help");
36     }
37 }
38
39 myCommand help={"help\0", "Usage: help [command]\n\0Display info about [command]\n\0", func_help};
40
```

4. 测试图

伪终端依次输入:

cmd

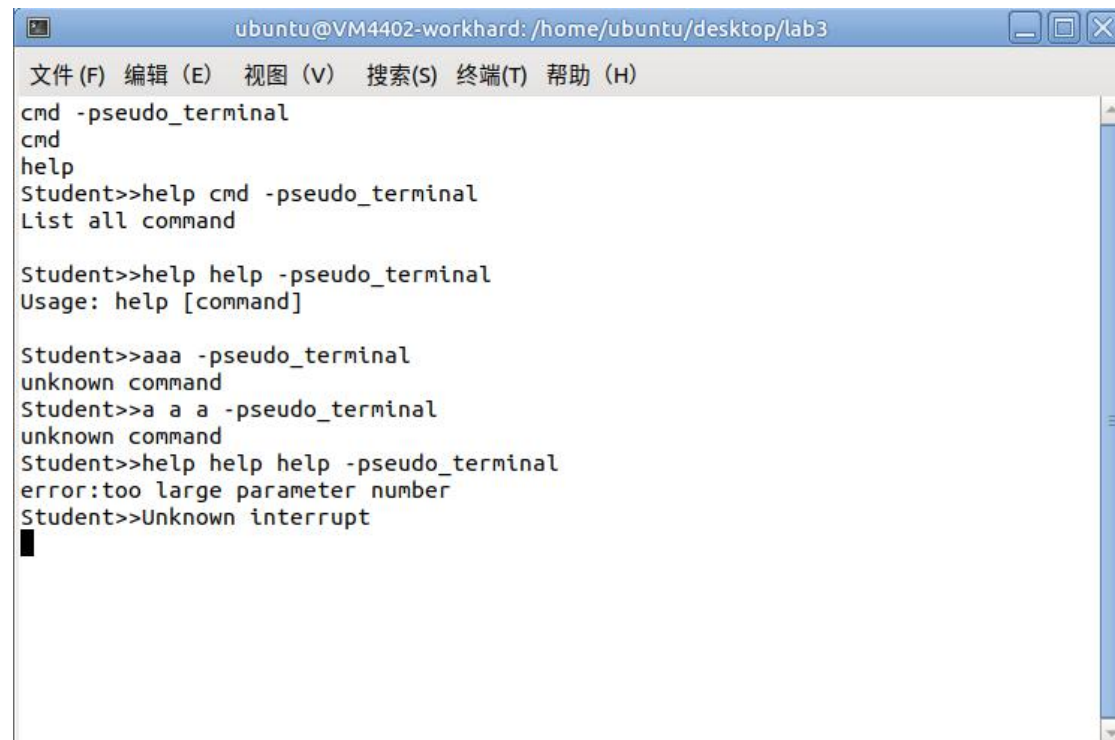
help cmd

help help

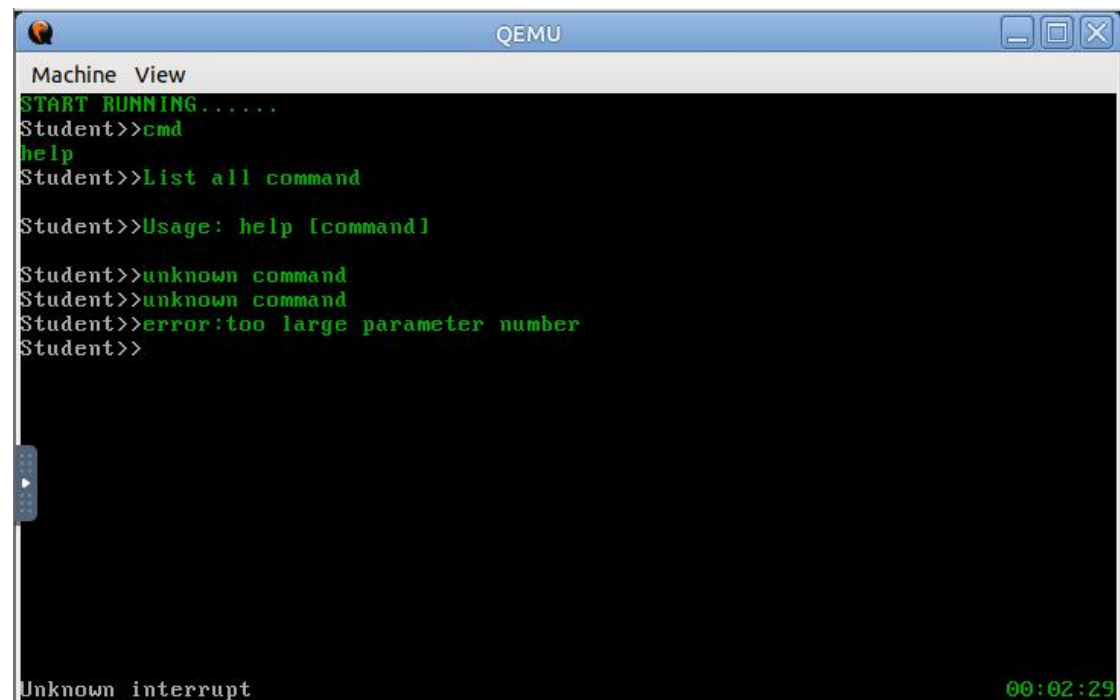
aaa

a a a

help help help 最后鼠标移入 QEMU，随意按键出现 Unknown interrupt

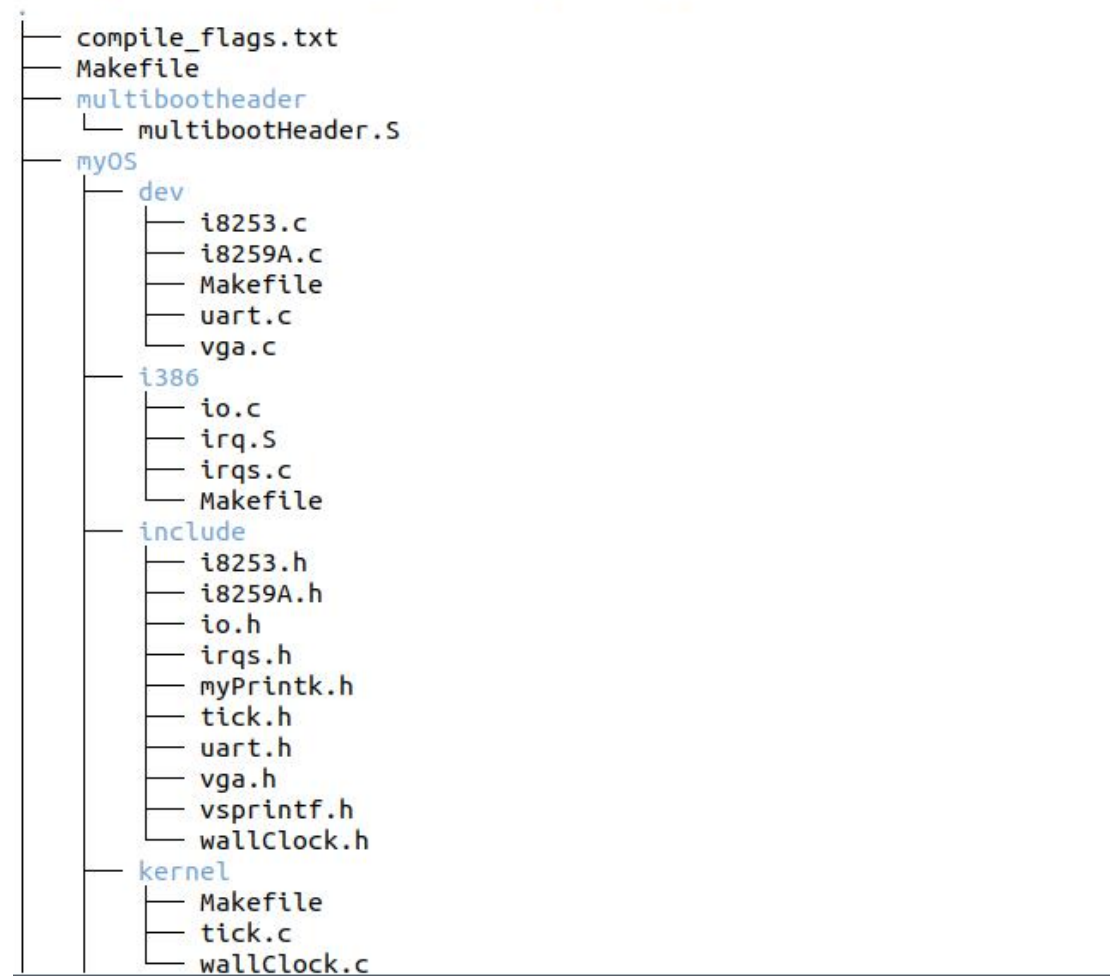


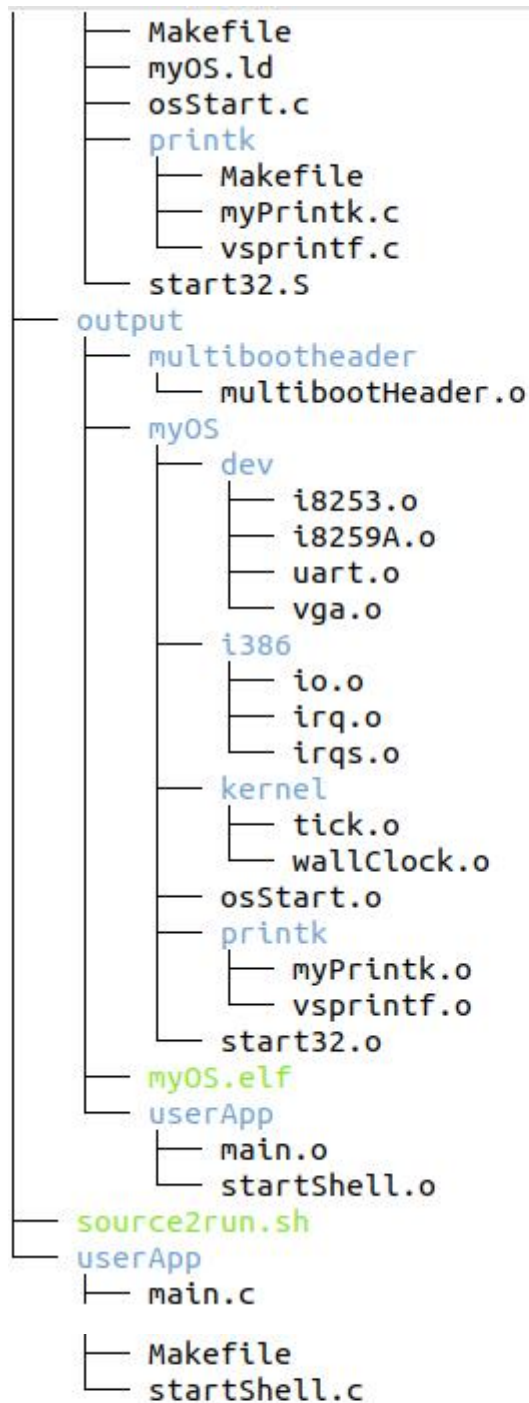
VGA 输出，墙钟如图：



5. 文件组织

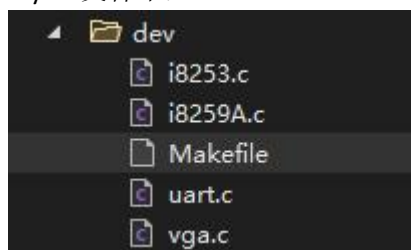
ubuntu@VM4402-workhard:~/home/ubuntu/desktop/lab3\$ tree





16 directories, 53 files

myOS 文件中,



```
1 DEV_OBJS = output/myOS/dev/uart.o \
2           output/myOS/dev/vga.o \
3           output/myOS/dev/i8259A.o \
4           output/myOS/dev/i8253.o
```

dev 下 makefile 中将 dev 目录下的 c.s 文件转化成的 o 文件链接

i386,kernel,printk 同理;

myOS 下 makefile 将 dev,i386,kernel,printk 中链接文件再次链接,

```
1  # 不需要修改
2  include $(SRC_RT)/myOS/dev/Makefile
3  include $(SRC_RT)/myOS/i386/Makefile
4  include $(SRC_RT)/myOS/printk/Makefile
5  include $(SRC_RT)/myOS/kernel/Makefile
6
7  MYOS_OBJS = output/myOS/start32.o output/myOS/osStart.o \
8             ${DEV_OBJS} \
9             ${I386_OBJS} \
10            ${PRINTK_OBJS} \
11            ${KERNEL_OBJS}
12
```

最后再和 userApp 中链接文件链接,

multibooheader 进入引导程序,

myOS.ld 分配 text 段, data 段,

Start32.S 分配栈段并写入 IDT,并写入时钟中断和其他中断

PS: myOS 中 include 文件夹中为 c 函数对应头文件声明,

需要在 myOS 目录下的文件中调用某函数, 只要在首部加入#include 对应 h 文件即可

```

1  SRC_RT=$(shell pwd)
2
3  # CROSS_COMPILE=i686-elf-
4  CROSS_COMPILE=
5  ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
6  C_FLAGS = -m32 -fno-stack-protector -g
7  INCLUDE_PATH = myOS/include
8
9  .PHONY: all
10 all: output/myOS.elf
11
12 MULTI_BOOT_HEADER=output/multibootheader/multibootHeader.o
13 include $(SRC_RT)/myOS/Makefile
14 include $(SRC_RT)/userApp/Makefile
15
16 OS_OBJS      = ${MYOS_OBJS} ${USER_APP_OBJS}
17
18 output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
19 | | ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${O
20
21 output/%.o : %.S
22 | | @mkdir -p $(dir $@)
23 | | @$ {CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<
24
25 output/%.o : %.c
26 | | @mkdir -p $(dir $@)
27 | | @$ {CROSS_COMPILE}gcc ${C_FLAGS} -I${INCLUDE_PATH} -c -o $@ $<
28
29 clean:
30 | | rm -rf output
31

```

6. 总结:

1. 多维数组与函数指针

1. 多维数组Argv[8][8]

Argv	Argv[0]首地址
Argv+1	Argv[1]首地址
*(Argv+1)	Argv[1][0]地址
((Argv+1)+1)	Argv[1][1]

```
Char cb[2][3][4];
Char(*p)[4];
p=cb[2];
*p=cb[2][0];
*(p+1)=cb[2][1];
Char ***p=cb; 类型不匹配, error!
```

1 函数指针的定义

1.1 普通函数指针定义

```
1. int (*pf)(int,int);
```

1.2 使用typedef定义函数指针类型

```
1. typedef int (*PF)(int,int);
2. PF pf; //此时, 为指向某种类型函数的函数指针类型, 而不是具体指针, 用它可定义具体指针
```

2 函数指针的普通使用

```
1. pf = add;
2. pf(100,100); //与其指向的函数用法无异
3. (*pf)(100,100); //此处*pf两端括号必不可少
```

注意: add类型必须与pf可指向的函数类型完全匹配

2. 调试正确姿势——增加断点 vga 输出