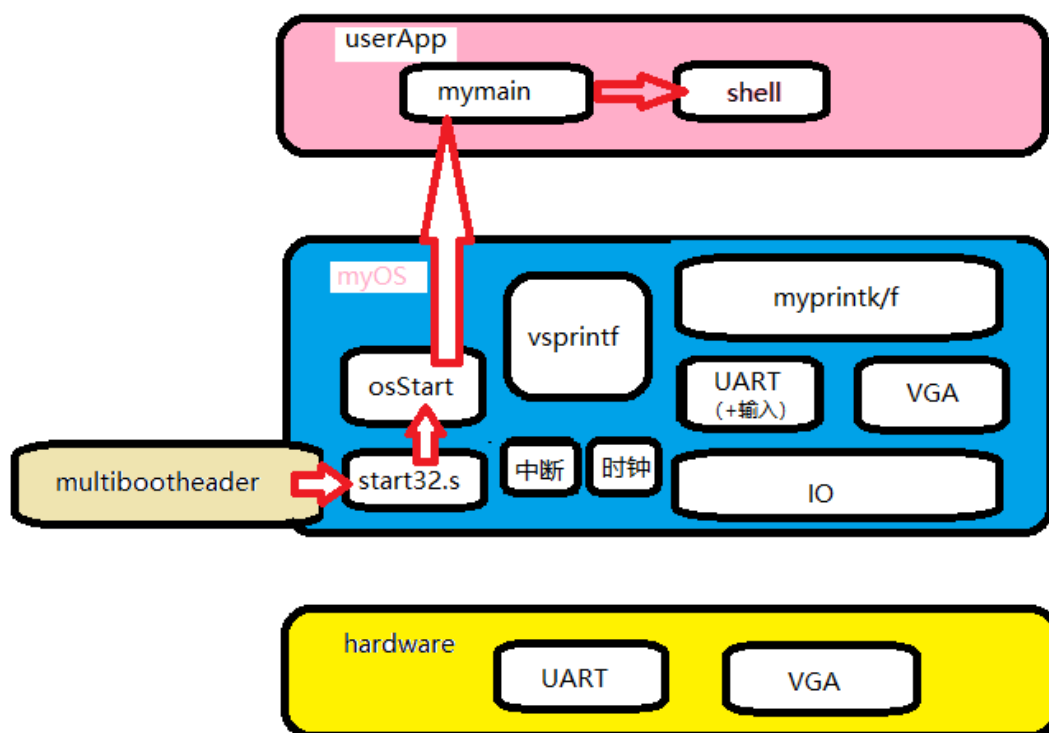


# Lab3 实验报告

PB18051098 徐碧涵

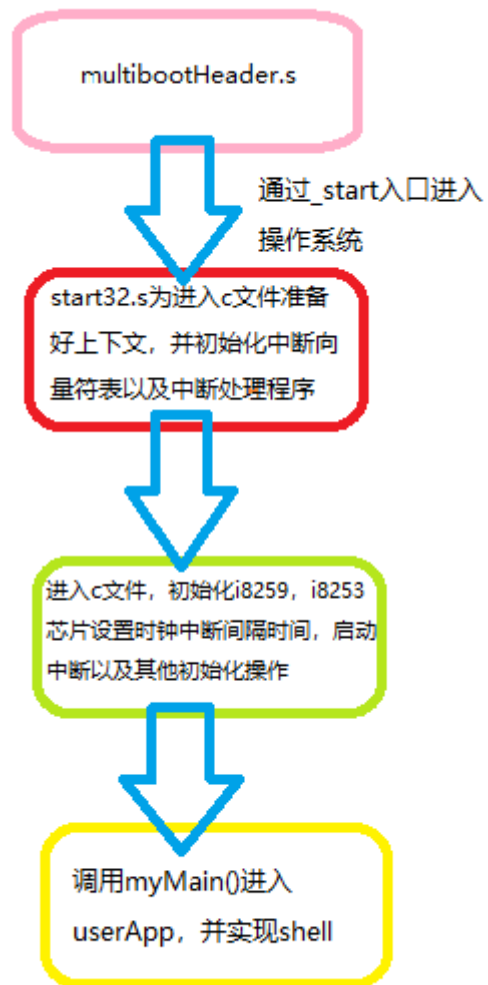
## 软件框图



本实验旨在 `myOS` 中实现中断，通过时钟中断实现时钟，启动中断，调用 `userApp` 实现 `shell`。

## 主流程及其实现

主流程如下：



通过遵循 multiboot 启动协议的 multibootheader.s 文件中的 multiboot\_header 头部启动内核，利用\_start 入口进入操作系统内部，执行 start32.s 文件，在其中为进入 c 文件准备好上下文，初始化中断向量表 IDT，将所有中断处理程序初始化为缺省处理函数 ignore\_int1，随后将时钟中断处理程序设置为 time\_interrupt 在其中完成中断处理，如时钟的维护。随后进入 c 文件对操作系统进行初始化，通过 init8259A(),init8253()初始化 i8259(可编程中断控制器)，i8253(可编程间隔定时器)芯片，然后启动时钟中断。随后调用 userApp 入口进入 shell，实现 shell 的功能。

## 主要功能模块及其实现

### 主要功能模块

#### 中断

*初始化中断描述符表 IDT 以及中断处理程序*

初始化中断描述符表

为 IDT 以及分配一块内存

```

.data
    .p2align 4
    .globl IDT
IDT:
    .rept 256
    .word 0,0,0,0
    .endr

idtptr:
    .word (256*8 - 1)
    .long IDT

```

初始化 IDT

```

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

```

中断处理程序初始化为缺省处理函数 ignore\_int1，ignore\_int1 如下：

```

ignore_int1:
    cld
    pusha
    call ignoreIntBody
    popa
    iret

```

为时钟中断设置中断处理程序：

```

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

```

time\_interrupt 内容如下：

```

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

```

### 初始化 i8259

通过 init8259A()初始化 i8259，识别中断

```

void init8259A(void)
{
    outb(0x21,0xff);//屏蔽所有中断源
    outb(0xA1,0xff);
//主片初始化
    outb(0x20,0x11);
    outb(0x21,0x20);//起始向量号0x20 ==》 0x21
    outb(0x21,0x04);//从片接入引脚位 0x04 ==》 0x21
    outb(0x21,0x3);//中断结束方式 AutoEOI 0x3 ==》 0x21
//从片初始化
    outb(0xA0,0x11);
    outb(0xA1,0x28);//起始向量号0x28==》 0xA1
    outb(0xA1,0x02);//接入主片的编号0x02==》 0xA1
    outb(0xA1,0x01);//中断结束方式 0x01==》 0xA1
}

```

初始化 i8253，设置时钟中断频率，并允许时钟中断

设置时钟中断频率为 100HZ，得到分频参数并写入相应端口,随后通过 i8259 控制允许时钟中断

```
void init8253(void)
{
    int cut=11932;    //设定时钟中断的频率为100HZ，得到分频参数
    outb(0x43,0x34);
    outb(0x40,cut&0xff);// 将分频参数写入0x40端口先低八位后高八位
    outb(0x40,(cut>>8)&0xff);
    outb(0x21,inb(0x21&0xfe)); //通过8259控制，允许时钟中断，取消屏蔽
}
```

*利用 tick() 在周期性时钟中断发生时维护时钟*

由于每 10ms 产生一次时钟中断，可知产生 100 次时钟中断即相当于经过 1s，通过 tick 的发生次数维护时钟

```
void tick(void)
{
    count++; //全局变量count在每次tick后+1
    //每tick100次几乎经过了1s，用于时钟维护
    if(count==100){
        setWallClockHook();
        count=0;}
}
```

*开启中断*

```
enable_interrupt:
    sti
    ret
```

*关闭中断*

```
disable_interrupt:
    cli
    ret
```

*中断处理缺省函数 ignoreIntBody()*

```
void ignoreIntBody(void)
{
    putchar("Unknown interrupt",0x4); //在vga左下角输出内容用于处理除时钟中断外的其他中断
}
```

**时钟**

通过全局变量 hh，mm，ss 表示当前时间的时分秒

**setWallClock** 设置当前时间

```
void setWallClock(int h, int m, int s)
{
    hh=h;
    mm=m;
    ss=s;
}
```

**getWallClock** 获取当前时间

```
void getWallClock(int *h, int *m, int *s)
{
    *h = hh;
    *m = mm;
    *s = ss;
}
```

**setWallClockHook** 更新时间

更新时间，保证时间符合取值要求

```
ss++;
if(ss>=60){
    mm++;
    if(mm>=60){
        hh++;
        if(hh>=24) hh=0;
        mm=0;}
    ss=0;}
}
```

以 hh: mm: ss 格式在 vga 右下角显示时间

```
vgabuffer1[4000-16]=(char)(hh/10)+'0';
vgabuffer1[4000-14]=(char)(hh%10)+'0';
vgabuffer1[4000-12]=':.';
vgabuffer1[4000-10]=(char)(mm/10)+'0';
vgabuffer1[4000-8]=(char)(mm%10)+'0';
vgabuffer1[4000-6]=':.';
vgabuffer1[4000-4]=(char)(ss/10)+'0';
vgabuffer1[4000-2]=(char)(ss%10)+'0';
```

**shell**

**初始化命令**

cmd, 列出所有命令

help [cmd], 调用指定命令的 help 函数, 若没有指定命令, 则调用 help 的 help 函数

```

struct command {
    char *cmd;
    int (*func)(void);
    void (*help_func)(void);
    char *desc;
} cmds[] = {
    {"cmd", cmd_handler, NULL, "list all commands"},
    {"help", help_handler, help_help, "help [cmd]"},
    {"", NULL, NULL, ""}
};

```

### 命令处理函数

cmd\_handler()处理 cmd 命令，即列出所有命令

```

int cmd_handler(void)
{
    int i=0;
    char str1[]="cmd name",str2[]="description";
    myPrintk(0x6,"%s\n",cmds[i].desc);
    myPrintk(0x7,"%-20s%-20s\n",str1,str2);
    //输出所有指令
    while(strcmp(cmds[i].cmd,"")){
        myPrintk(0x6,"%-20s%-20s\n",cmds[i].cmd,cmds[i].desc);

        i++;
    }
    return 0;
}

```

help\_handler()处理 help[cmd]指令，即列出需要 help 的命令的描述。先比对需要 help 的命令，若无指定命令则调用 help\_help(), 否则若在命令集合中找到该命令，则输出该命令的描述信息，若找不到，则说明该命令不存在，无法识别，输出 “unknown cmd!”

```

int help_handler(void)
{
    int i=0;
    int flag=0;
    if(strcmp(argv[1],"")==0){ help_help(); flag=1;}//若没有指定命令，则调用help的help
    else
    {
        while(strcmp(cmds[i].cmd,"")){ //寻找指定命令
            if(strcmp(cmds[i].cmd,argv[1])==0){ //若找到指定命令则输出描述信息
                flag=1;
                myPrintk(0x6,"%s\n",cmds[i].desc);
                break;}
            i++;
        }
    }
    if(!flag)myPrintk(0x6,"unknown cmd!\n"); //无法识别的指令

    return 0;
}

```

help\_help()输出 help 的描述信息

```

void help_help(void)
{
    myPrintk(0x6,"%s\n",cmds[1].desc);
}

```

### 读取命令并显示

读取终端输入的命令并逐个显示在输入区域：

```

//逐个显示输入的字符，并将其储存在str中
while((c=uart_get_char())!='\r'){str[k++]=c;myPrintk(0x6,"%c",c);}
str[k]='\0';
k=0;

```

将输入的字符串保存至 argv[]中，并得到参数个数 argc

```

while(str[k]){
    while(str[k]!=' ')k++;
    if(str[k]){ argc++;
        while(str[k]!='&&str[k]){
            argv[i][j++]=str[k];k++;}
        argv[i][j]='\0';
        if(str[k]==' '){j=0;i++;}
    }
}

```

### 处理命令

当输入内容参数不为 0 时，对输入命令进行处理，比对第一个参数，如果是命令集中的命令，则调用相应的命令处理函数，否则该命令无法识别，输出 “unknown cmd!”

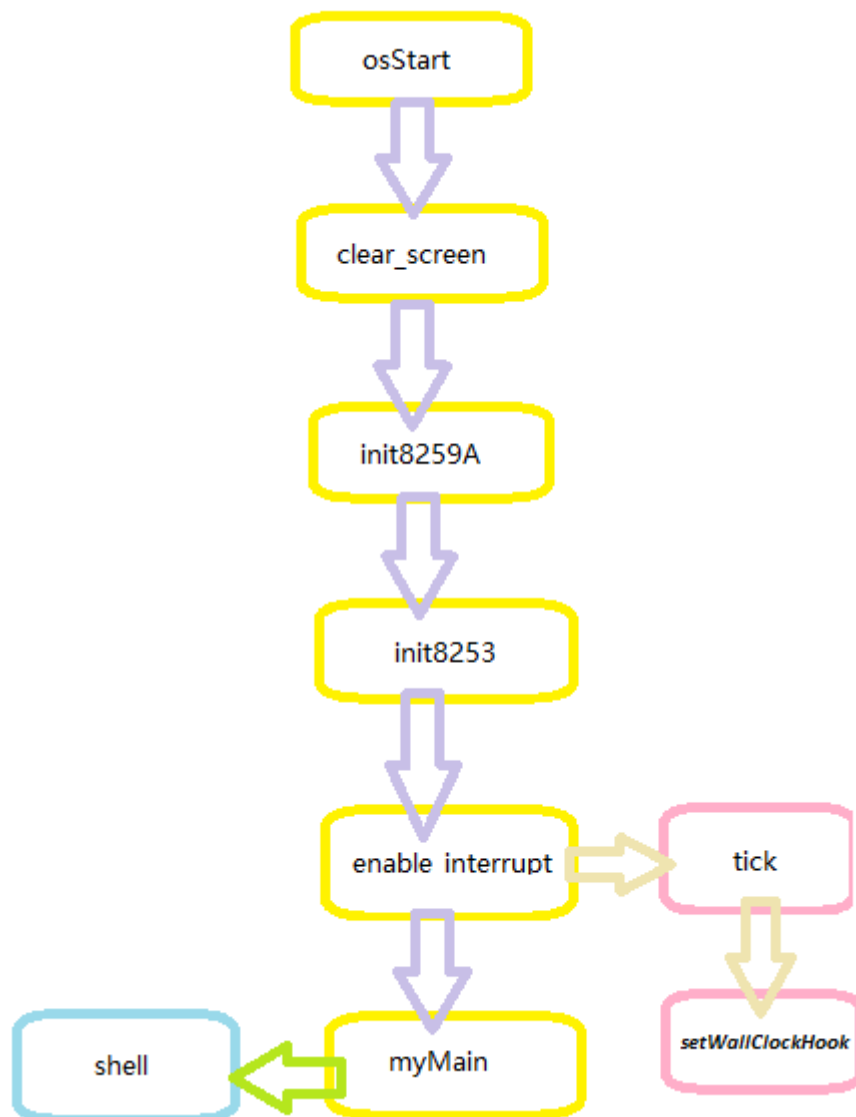


```

if(argc==1||argc==2){
    i=0;
    //判断输入命令是否是合法命令
    while(strcmp(cmds[i].cmd,"")){
        if(strcmp(cmds[i].cmd,argv[0])==0){
            flag=1;
            cmds[i].func();//调用相应命令处理程序
            break;}
        i++;
    }
    if(!flag)myPrintk(0x6,"unknown cmd!\n");//该命令非法，无法识别
}

```

流程图如下：



## Makefile 组织

基本上每一个文件夹下都有一个 **Makefile** 文件将目标文件生成中间文件，便于后期整体组织

*目录组织如下：*

```
multibootheader
    multibootheader.s
myOS
    i386
        io.c
        irq.s
        irqs.c
        Makefile
    dev
        i8253.c
        i8259A.c
        uart.c
        vga.c
        Makefile
    printk
        vsprintf.c
        myprintk.c
        Makefile
    kernel
        tick.c
        WallClock.c
        Makefile
    lib
        string.c
        Makefile
    Makefile
    myOS.ld
    osStart.c
    start32.s
userApp
    main.c
    shell.c
    Makefile
Makefile
source2img.sh
```

底层文件夹下的 Makefile 文件使当前文件夹下的.c 文件生成中间文件,处于上层文件夹的 Makefile 将下层文件夹中生成的中间文件进行组织最后由最顶层文件夹下的 Makefile 汇总生成目标文件。

## 代码布局说明(地址空间)

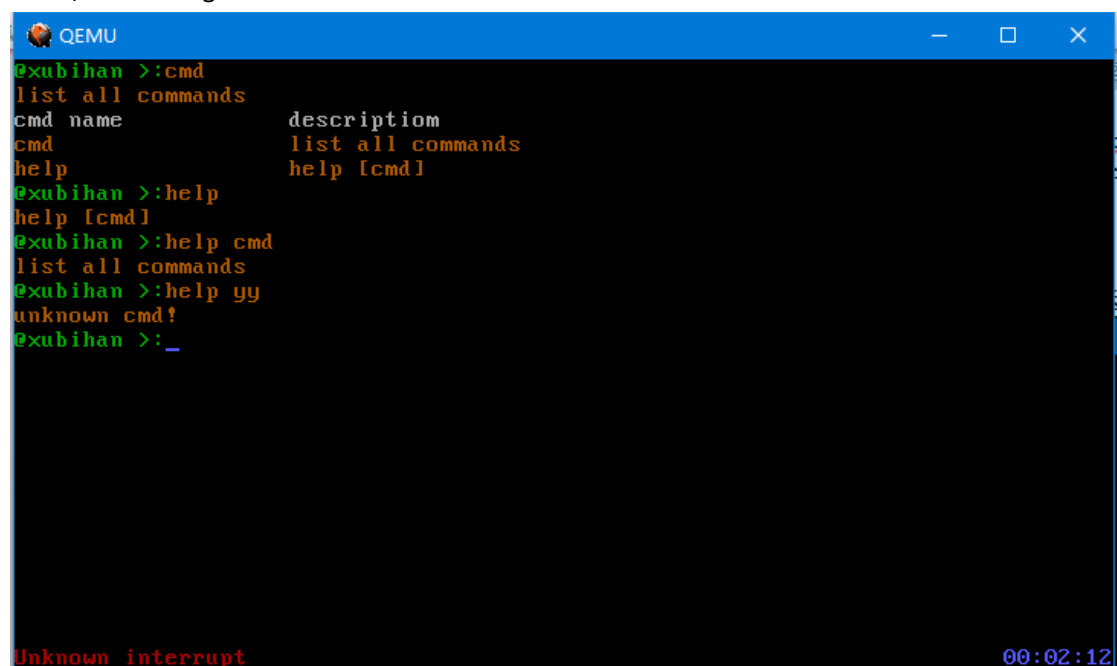
代码在内存空间中的地址以及布局由 myOS.ld 文件所确定,代码将从物理内存 1M 的位置开始存放,代码段 8 个字节对齐,之后存放程序中已经初始化的全局变量并按 16 字节对齐,在之后存放未初始化的全局变量按 16 字节对齐,下一个存放的代码按 512 个字节对齐

## 编译过程说明

通过 ./source2img.sh 命令执行 make 命令在 Makefile 的组织下使所有.c.s 文件生成中间文件最终按照 myOS.ld 文件代码布局以及地址空间的要求在内存空间生成并存放.elf 文件。

## 运行和运行结果说明

输入 ./source2img.sh 命令运行脚本文件,编译后进而执行.elf 文件,运行结果如下:



```
QEMU
exubihan >:cmd
list all commands
cmd name      description
cmd           list all commands
help          help [cmd]
exubihan >:help
help [cmd]
exubihan >:help cmd
list all commands
exubihan >:help yy
unknown cmd!
exubihan >:_

Unknown interrupt 00:02:12
```

## 遇到的问题和解决方案说明

问题: 输入命令时,无法看到输入的命令,认为出了故障无法输入。

解决方案: 实际上命令已经输入,只是没有打印来。将输入的命令逐个字符打印出来即可看到输入的命令。