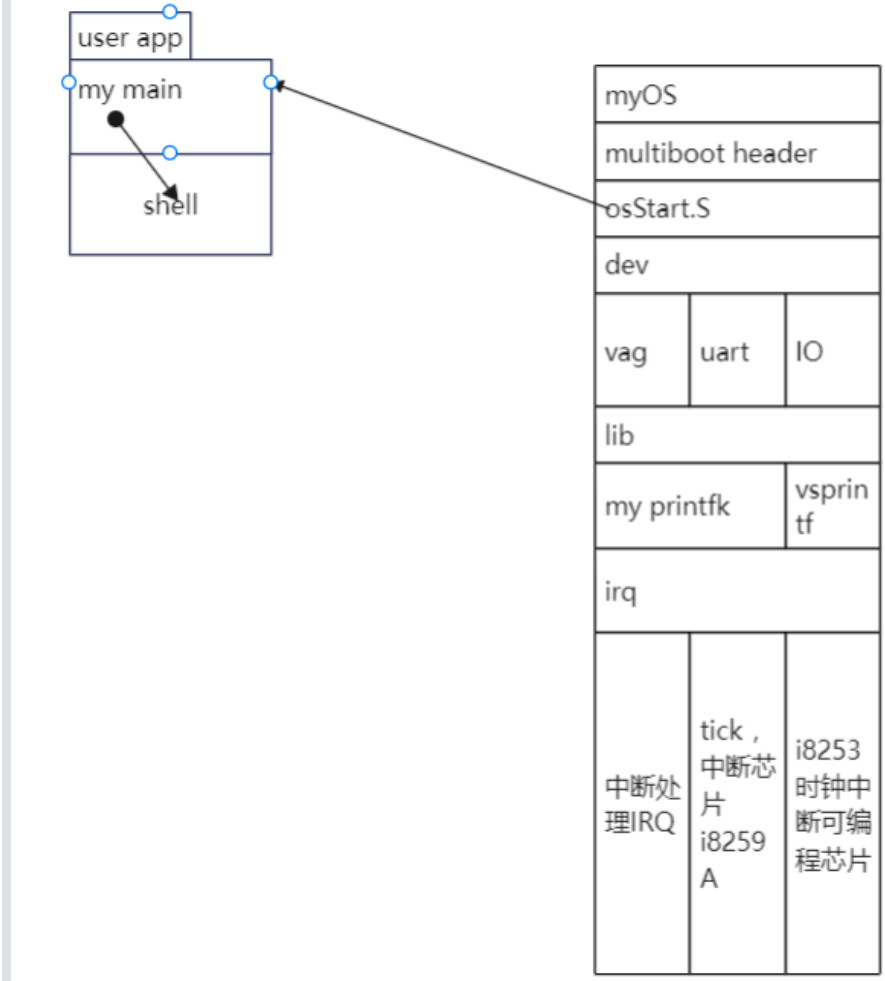
lab3

软件框图



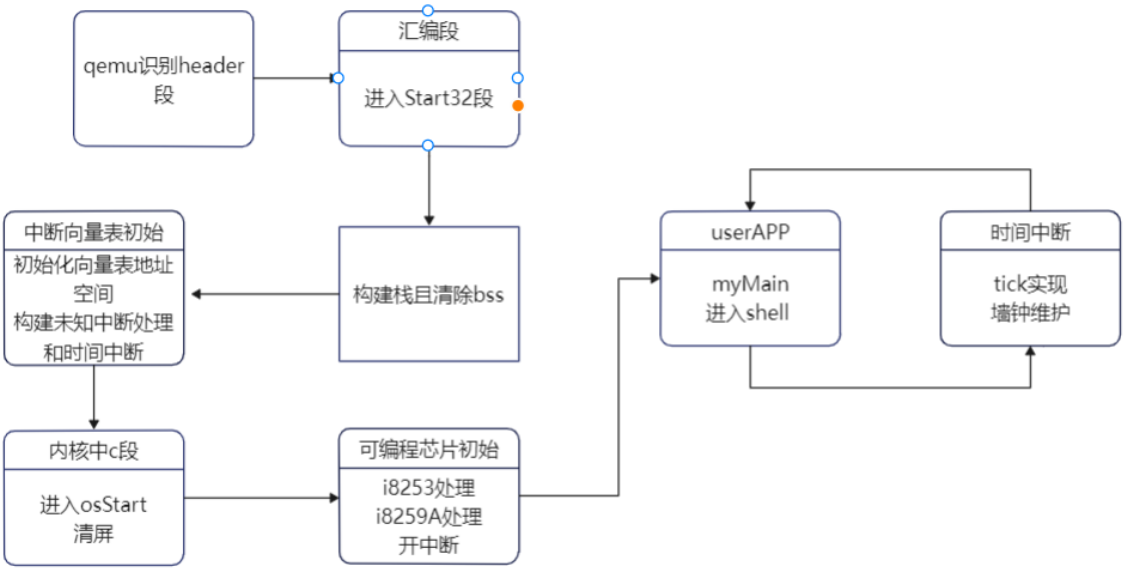
> 系统主要由三个部分组成
>
> user APP，OS核 和硬件接口（由qemu提供）组成
>
> OS 包含中断处理 irq， 时钟维护tick 和与输入输出的交互 IO与硬件 VGA uart 与串口和显存

流程实现

qemu识别header段后进入Start32段(汇编段)（data bss等段布局在gcc生成时已完成）

构建栈且清除bss， 初始化中断向量表，进入OSstart(C段)， 初始化i8259A 和i8253 两可编程芯片（开中断 时间中断开始）

先清屏再进入mymain函数调用shell



Linux 伪终端

> multiboot_header→myOS→userApp

> 通过 -pty 将qemu 串口重定向到伪终端 以伪终端作为中介 使用户和shell 读取和写入串口 实现与shell 的交互

主要功能模块实现

lab2

I/O实现

```
#include "io.h"

// inb，从端口号为 port_from 的端口读取一个字节
unsigned char inb(unsigned short int port_from) {
    unsigned char value;
    __asm__ __volatile__ ("inb %w1, %b0": "=a"(value): "Nd"(port_from));
    return value;
}

// outb，向端口号为 port_to 的端口输出一个字节
void outb(unsigned short int port_to, unsigned char value) {
    __asm__ __volatile__ ("outb %b0, %w1":: "a"(value), "Nd"(port_to));
}
```

基于内联汇编实现对特定地址（port： Nd）的内存实现访存

VGA实现

> 对显存的写入基于
>
> c语言类型转化
>
> 将int 型变量的地址量 转化为 指向short型变量的指针以达到修改显存地址

```
#include "io.h"
```

```c
#include "vga.h"

#define VGA_BASE 0xB8000 // vga 显存起始地址
#define VGA_END  0xB8FA0 // vga 显存结束地址
#define VGA_LAST_LINE  0xB8F00 //vga 显存倒数第一行
#define VGA_SCREEN_WIDTH 80 // vga 屏幕宽度（可容纳字符数）
#define VGA_SCREEN_HEIGHT 25 // vga 屏幕高度

#define CURSOR_LINE_REG 0xE // 高八位
#define CURSOR_COL_REG 0xF // 低八位
#define CURSOR_INDEX_PORT 0x3D4 // 光标行列索引端口号
#define CURSOR_DATA_PORT 0x3D5 // 光标数据端口号


/* 将光标设定到特定位置
 * 提示：使用 outb */
void set_cursor_pos(unsigned int pos) {
    outb(CURSOR_INDEX_PORT,0xE);
    outb(CURSOR_DATA_PORT,(pos>>8));//设定高八位
    outb(CURSOR_INDEX_PORT,0xF);
    outb(CURSOR_DATA_PORT,(pos&0xFF));//设定低八位
}

/* 获取光标当前所在位置
 * 提示：使用 inb */
unsigned int get_cursor_pos(void) {
    int unsigned pos;
    outb(CURSOR_INDEX_PORT,0xE);
    pos=(inb(CURSOR_DATA_PORT)<<8);
    outb(CURSOR_INDEX_PORT,0xF);
    pos+=(inb(CURSOR_DATA_PORT));
    return pos;
}

/* 滚屏，vga 屏幕满时使用。丢弃第一行内容，将剩余行整体向上滚动一行
 * 提示：使用指针修改显存 */
//最后一行内容写入前一行后附 foregroud 白 background 黑
void scroll_screen(void) {
    unsigned long int* cl=(unsigned long int*)VGA_BASE;
    do{
            *cl=*(cl+40);
    }while((int)(cl++)!=VGA_LAST_LINE);
    cl=(unsigned long int*)VGA_LAST_LINE;
    do{*cl=0x0F000F00;}while((int)(cl+=1)!=VGA_END);
}

/* 向 vga 的特定光标位置 pos 输出一个字符
 * 提示：使用指针修改显存 */
void put_char2pos(unsigned char c, int color, unsigned int pos) {
    short int *p;
    p=(short int *)(pos*2+0xB8000);
    *p=(short int)c+(color<<8);
}
//将偏移量和VGA_BASE相加得到VGA显存输出位置

/* 清除屏幕上所有字符，并将光标位置重置到顶格
 * 提示：使用指针修改显存 */
void clear_screen(void) {
```

```
    unsigned long int* cl=(unsigned long int*)VGA_BASE;
    do{*cl=0x0F000F00;}while((int)(cl+=1)!=VGA_END);
    set_cursor_pos(0);

}
//清屏时为显存赋初值foregroud 白 background 黑


/* 向 vga 的当前光标位置输出一个字符串，并移动光标位置到串末尾字符的下一位
 * 如果超出了屏幕范围，则需要滚屏
 * 需要能够处理转义字符 \n */
void append2screen(char *str, int color) {
    unsigned int pos=get_cursor_pos();
    while (*str!='\0')
    {
        if(*str=='\n'){
            if(pos>=(VGA_LAST_LINE-0xB8000)/2){scroll_screen();pos=
(VGA_LAST_LINE-0xB8000)/2;}
            else{pos=(pos/80+1)*80;}
            // set_cursor_pos(pos);
            str++;
            continue;
        }
        else{put_char2pos(*str,color,pos);}
        str++;
        pos+=1;
        if(pos==(VGA_END-0xB8000)/2){scroll_screen();pos=((VGA_LAST_LINE-
0xB8000)/2);}
    }
    set_cursor_pos(pos);
}
```

串口实现

> 通过outb向0x3F8输入字符即可

```
#include "io.h"
#include "uart.h"

#define UART_PORT 0x3F8 // 串口端口号

/* 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) {
    outb(UART_PORT,ch);
}

// 向串口输出一个字符串
void uart_put_chars(char *str) {
    while(*str!='\0'){
        uart_put_char(*str);
        str++;
    }
}
```

字符串处理实现

> 该部分太长 不贴全
>
> 仅解释重要部分

```
static size_t strnlen(const char *s, size_t count);
//实现对字符串大小识别
static int skip_atoi(const char **s);
//将字符串数字连接起来（无大小），转化为有大小的int型变量
static char *number(char *str, long num, int base, int size, int precision, int
type);
//进制转化
int vsprintf(char *buf, const char *fmt, va_list args);
//*buf格式化处理后的字符串 *fmt函数用户定义输出格式 args基于格式需调用的参数
//通过寻找 % 后的标识符 对参数列表进行相应的处理后与fmt 写入buf
```

lab3

中断向量表的构建

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

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

    .word (256*8 - 1)
    .long IDT
#为中断向量表构建地址空间
time_interrupt:
cld
pushf
pusha
call tick
popa
popf
iret

    .p2align 4
ignore_int1:
cld
pusha
call ignoreIntBody
popa
iret
#中断处理机制的编写
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
#初始化时间中断向量表
```

irq.S 开关中断

```
.text
.code32
_start:
.globl enable_interrupt
enable_interrupt:
sti;
ret;

.globl disable_interrupt
disable_interrupt:
cli;
ret;
```

可编程芯片的处理i8259&i8253

```
#include "io.h"

void init8259A(void){
    //设置全mask
    outb(0x21,0xFF);
    outb(0xA1,0xFF);
    //主片
    outb(0x20,0x11);
    outb(0x21,0x20);
    outb(0x21,0x04);
    outb(0x21,0x3);

    //从片
    outb(0xA0,0x11);
    outb(0xA1,0x28);
    outb(0xA1,0x02);
    outb(0xA1,0x01);
    //初始化mask
    outb(0x21,inb(0x21)&0b010);
}



#include "io.h"
#define clkfrq 1193180/100
void init8253(void){
    // set 8253 timer-chip
    outb(0x43, 0x34);
    outb(0x40,clkfrq % 256);
    outb(0x40, clkfrq / 256);
```

```
}
```

0x34的含义是?

> 可编程芯片
>
> 00 11 0100
>
> 00 选则通道0
>
> > 通道0用作定时器,为系统提供一个恒定的时间标准.
>
> 11 控制芯片的写入方式
>
> > 读写16位
>
> 010 选则工作方式2
>
> > 不需要重新写入 周期 便顺着周期循环产生中断

时钟的维护

```
#include "tick.h"
int ticks=0;
int HH,MM,SS;
extern void oneTickUpdateWallClock(int HH, int MM, int SS);

void tick(void){
    ticks++;
    if (ticks %100 ==0)
    {   ticks=0;
        SS++;
        if(SS%60==0){
            SS=0;
            MM++;
            if(MM%60==0)
            {
            MM=0;
            HH++;
            }
        }
    }
    oneTickUpdateWallClock(HH, MM, SS);

    return;
}
```

设置墙钟

```
#include "wallClock.h"
void put_char2pos(unsigned char c, int color, unsigned int pos);
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){
    unsigned char* time[8];
    int pos=1960;
    time[0]=(char)(HH/10+48);
    time[1]=(char)(HH%10+48);
    time[2]=':';
    time[3]=(char)(MM/10+48);
    time[4]=(char)(MM%10+48);
    time[5]=':';
    time[6]=(char)(SS/10+48);
    time[7]=(char)(SS%10+48);
    for(int i=0;i<8;i++,pos++)
    {put_char2pos(time[i],0x04,pos);
    }
    return;
}
```

hooking

> 指通过拦截软件模块间的函数调用、消息传递、事件传递来修改或扩展操作系统、应用程序或其他软件组件的行为的各种技术。处理被拦截的函数调用、事件、消息的代码，被称为**钩子**（hook）。
>
> -wiki
>
> 在这次实验中 hooking机制 指本该由内核设置的墙钟工作方式
>
> 通过hook机制拦截 set 使userapp 段可以定义和设置墙钟工作方式

shell

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

int strcmp(char *a,char *b){
        while((*a==*b)&&*a!='\0'){a++;b++;}
        return *a-*b;
}
```

```
// 字符串比较
int strcpy(char *a,char *b){
    while (*b!='\0'){
        *a=*b;
        a++;b++;
    }
}
//字符串拷贝
typedef struct myCommand {
    char name[80];
    char help_content[200];
    void (*func)(int argc, char (*argv)[8]);
}myCommand;
int cmdnums=3;
myCommand all[];

int func_cmd(int argc, char (*argv)[8]){
    for(int i=0;i<cmdnums;i++){
        append2screen(all[i].name,0x04);
        append2screen("\x20",0x04);
    }
    append2screen("\n",0x04);
}

// myCommand cmd={"cmd\0","List all command\n\0",func_cmd};

int func_help(int argc, char (*argv)[8]){
    for(int i=1;i<argc;i++){
        for(int j=0;j<cmdnums;j++){
            if(strcmp(argv[i],all[j].name)==0)
{append2screen(all[j].help_content,0x04);append2screen("\n",0x04);}
        }
    }
}
int func_hello(int argc, char (*argv)[8]){
    append2screen("Hello world\n",0x12);
}


myCommand all[100]={{"cmd\0","cmd:List all command",func_cmd},
{"help\0","help:Usage: help [command]\nDisplay info about [command]",func_help},
{"hello","hello:Hello world",func_hello}};
// 静态定义


int osCmdReg(char *name,char* help,void (*func)()){
            all[cmdnums].func=func;
            strcpy(all[cmdnums].name,name);
            strcpy(all[cmdnums].help_content,help);
            cmdnums++;
}
char *a="test";


//动态注册
void startShell(void){
    osCmdReg(a,a,func_hello);
//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

    int argc=0;
    char argv[8][8];
    do{
        BUF_len=0;
        argc=0;
        myPrintk(0x07,"Student>>");
        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]='\n';
        BUF[BUF_len+1]='\0';
        uart_put_chars(" -pseudo_terminal\n");
        uart_put_char('\r');
        append2screen(BUF,0x04);
        //向vga中输出伪终端内容
        for (int i = 0,k=0; i < BUF_len+1; i++)
        {
            if((int)BUF[i]==32|BUF[i]=='\n'){
                argv[argc][k]='\0';
                argc++;
                k=0;
                continue;
            }
            argv[argc][k]=BUF[i];
            k++;
        }
        // 参数处理
        for(int j=0;j<cmdnums;j++)
        {
            if(strcmp(all[j].name,argv[0])==0){all[j].func(argc,argv);break;}
        }
        //命令调用
    }while(1);

}
```

源代码组织

目录组织

```
.
├── makefile
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── dev
│   │   ├── i8253.c
│   │   ├── i8259A.c
│   │   ├── Makefile
```

```
|   |   ├── uart.c
|   |   └── vga.c
|   ├── i386
|   |   ├── io.c
|   |   ├── irq.S
|   |   ├── irqs.c
|   |   └── Makefile
|   ├── include
|   |   ├── i8253.h
|   |   ├── i8259A.h
|   |   ├── io.h
|   |   ├── irqs.h
|   |   ├── myPrintk.h
|   |   ├── tick.h
|   |   ├── uart.h
|   |   ├── vga.h
|   |   ├── vsprintf.h
|   |   └── wallClock.h
|   ├── kernel
|   |   ├── Makefile
|   |   ├── tick.c
|   |   └── wallClock.c
|   ├── Makefile
|   ├── myOS.ld
|   ├── osStart.c
|   ├── printk
|   |   ├── Makefile
|   |   ├── myPrintk.c
|   |   └── vsprintf.c
|   └── start32.S
├── source2img.sh
└── userApp
    ├── main.c
    ├── Makefile
    └── startShell.c

8 directories, 35 files
```

makefile 组织

```
├── multibootheader
|   └── multibootHeader.o
├── myOS
|   ├── dev
|   |   ├── i8253.o
|   |   ├── i8259A.o
|   |   ├── uart.o
|   |   └── vga.o
|   ├── i386
|   |   ├── io.o
|   |   ├── irq.o
|   |   └── irqs.o
|   ├── kernel
|   |   ├── tick.o
|   |   └── wallClock.o
|   ├── osStart.o
|   ├── printk
```

```
|   |   ├── myPrintk.o
|   |   └── vsprintf.o
|   └── start32.o
├── myOS.elf
└── userApp
     ├── main.o
     └── startShell.o
```

代码布局

```
There are 19 section headers, starting at offset 0xccb8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00100000 000080 0015d1 00  AX  0   0 16
  [ 2] .rodata           PROGBITS        001015d4 001654 00017f 00   A  0   0  4
  [ 3] .eh_frame         PROGBITS        00101754 0017d4 0004c8 00   A  0   0  4
  [ 4] .text.__x86.get_p PROGBITS        00101c1c 001c9c 000004 00  AX  0   0  1
  [ 5] .text.__x86.get_p PROGBITS        00101c20 001ca0 000004 00  AX  0   0  1
  [ 6] .text.__x86.get_p PROGBITS        00101c24 001ca4 000004 00  AX  0   0  1
  [ 7] .data             PROGBITS        00101c40 001cc0 007714 00  WA  0   0 32
  [ 8] .got.plt          PROGBITS        00109354 0093d4 00000c 04  WA  0   0  4
  [ 9] .bss              NOBITS          00109360 0093e0 00035c 00  WA  0   0 32
  [10] .debug_line       PROGBITS        00000000 0093e0 000dde 00      0   0  1
  [11] .debug_info       PROGBITS        00000000 00a1be 000eaf 00      0   0  1
  [12] .debug_abbrev     PROGBITS        00000000 00b06d 0008c4 00      0   0  1
  [13] .debug_aranges    PROGBITS        00000000 00b938 000200 00      0   0  8
  [14] .debug_str        PROGBITS        00000000 00bb38 0004a2 01  MS  0   0  1
  [15] .comment          PROGBITS        00000000 00bfda 00002a 01  MS  0   0  1
  [16] .symtab           SYMTAB          00000000 00c004 0007b0 10     17  71  4
  [17] .strtab           STRTAB          00000000 00c7b4 00041f 00      0   0  1
  [18] .shstrtab         STRTAB          00000000 00cbd3 0000e4 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)

Elf file type is EXEC (Executable file)
Entry point 0x100010
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000080 0x00100000 0x00100000 0x09360 0x096bc RWE 0x20
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .eh_frame .text.__x86.get_pc_thunk.bx
.text.__x86.get_pc_thunk.ax .text.__x86.get_pc_thunk.cx .data .got.plt .bss
   01
```

编译过程说明

编译所用指令

```
gcc -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
gcc -m32 -fno-stack-protector -fno-builtin -g
```

编译大致过程

编译汇编文件和c语言文件形成可重定向的二进制文件，再通过ld 命令将可重定向文件重新组织链接形成elf文件

运行结果