OSLab2

191220083 蒙芷露 mzl0830@sina.com

阅读框架代码：

> 把需要实现的加粗了

\bootloader：

lab1中保护模式的开启是在start.s中完成的，lab2已经提供，考虑复用；

boot.c: bootloader 加载kernel到磁盘； kMainEntry应该是kernel的开始（的确是，在kernel的main.c中可以看到），认为已经完成todo中要求的填写kmainentry phoff offset

boot.h中还有I/O相关的串口函数

start.s: 设置esp 具体数值？0x1fffff

\kernel

assert.h 提供了一个宏来调试判断

serial.c:

initSerial 串口的初始化已经完成

putChar 调试用

main.c:

初始化idt（中断向量表）--> idt.c

8259a（中断控制器）--> i8259a.c

初始化gdt表和tss段--> kvm.c/initSeg()

初始化vga -->vga.c

初始化键盘 --> keyboard.c

加载用户程序到内存地址 -->kvm.c/loadUMain() √

doIrq.S: 中断处理函数相关的汇编代码irqKeyboard 需要填写

irqHandle.c: 主要是中断处理程序的调用和系统调用的函数实现

irqHandle(): 中断处理程序的调用

KeyboardHandle(): 键盘的处理

syscallPrint(): 显示打印

syscallGetChar(): 自由任务：返回键盘输入的一个字符，作为系统中断处理的一部分

syscallGetStr(): 自由任务：返回键盘输入的一条字符串作为系统中断处理的一部分

该文件中调用到的与设备相关的部分都可以通过头文件的提示在kernel中找到

\lib

syscall.c:

printf(): 利用系统中断，先压参数到栈中再启动中断来进行系统调用 上述两函数大概同理

getChar()/getStr(): 参考printf的实现，考虑利用syscallGetChar/syscallGetStr来实现

具体实现

加载kernel/用户程序到内存地址：

加载kernel：磁盘1-200号是内核的部分 装载进入后将kMainEntry指向程序真正开始的地方；此时需要借助elf文件的格式信息：

```
struct ELFHeader {
        unsigned int   magic;
        unsigned char  elf[12];
        unsigned short type;
        unsigned short machine;
        unsigned int   version;
        unsigned int   entry;
        unsigned int   phoff;
        unsigned int   shoff;
        unsigned int   flags;
        unsigned short ehsize;
        unsigned short phentsize;
        unsigned short phnum;
        unsigned short shentsize;
        unsigned short shnum;
        unsigned short shstrndx;
};

/* ELF32 Program header */
struct ProgramHeader {
        unsigned int type;
        unsigned int off;
        unsigned int vaddr;
        unsigned int paddr;
        unsigned int filesz;
        unsigned int memsz;
        unsigned int flags;
        unsigned int align;
};
```

elf->entry才是跳转的目标位置；通过得到phoff读取程序头表找到实际程序开始的偏移量。第二个for循环是在将程序整体前移，去掉程序实际开始之前的部分。

```
oid bootMain(void) {
    int i = 0;
    int phoff = 0x34;
    int offset = 0x1000;
    unsigned int elf = 0x100000;
    void (*kMainEntry)(void);
    kMainEntry = (void(*)(void))0x100000;

    for (i = 0; i < 200; i++) {
        readSect((void*)(elf + i*512), 1+i);
    }

    //填写kMainEntry、phoff、offset
    kMainEntry=(void(*)(void))((struct ELFHeader*)elf)->entry;
    phoff=((struct* ELFHeader*)elf)->phoff;//程序头表开始的位置
    offset = ((struct ProgramHeader*)(elf+phoff))->off;//程序实际部分的首字节在文件中
的位置
    for (i = 0; i < 200 * 512; i++) {
```

```
        *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
    }//整体前移

    kMainEntry();
}
```

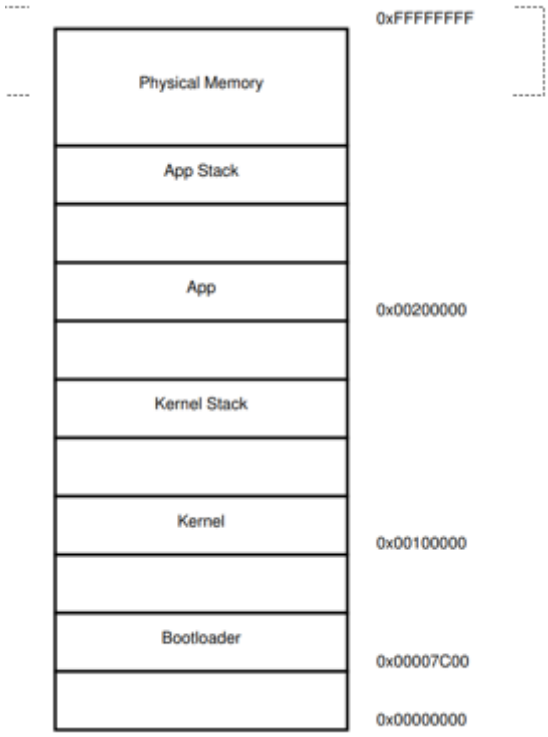
加载用户程序：在kvm.c中完成，关键是知道程序所在磁盘的位置；具体可以参见\app\Makefile 是 **0x00200000**

esp需要被更改吗 是在何时切换到用户栈的？ --需要，在enterUserspace中有一句0x2fffff就是用户栈的指针被压入作为跳转的信息。

初始化栈顶指针

start.S的任务是加载内核，所以在其中设定的栈顶指针应当是kernel stack的顶部

由图可以设定为 **0x1FFFFF**



初始化中断向量表

setIntr/setTrap：初始化interrupt gate的通用方法，其中GateDescriptor的定义在memory.h

```
struct GateDescriptor {
        uint32_t offset_15_0      : 16;
        uint32_t segment          : 16;
        uint32_t pad0             : 8;
        uint32_t type             : 4;
        uint32_t system           : 1;
        uint32_t privilege_level  : 2;
        uint32_t present          : 1;
        uint32_t offset_31_16     : 16;
};
```

除了给定的offset，segment，DPL信息以及默认的位数信息之外，还有present位和type需要确定；type可以见手册；present位如下：

> **I386 Interrupt Gate**
>
> The Interrupt Gate is used to specify an interrupt service routine. When you do `INT 50` in assembly, running in protected mode, the CPU looks up the 50th entry (located at 50 * 8) in the IDT. Then the Interrupt Gates selector and offset value is loaded. The selector and offset is used to call the interrupt service routine. When the `IRET` instruction is read, it returns. If running in 32 bit mode and the specified selector is a 16 bit selector, then the CPU will go in 16 bit protected mode after calling the interrupt service routine. To return you need to do `032 IRET`, else the CPU doesn't know that it should do a 32 bit return (reading 32 bit offset of the stack instead of 16 bit).
>
> | type_attr | Type |
> |---|---|
> | 0b1110=0xE | 32-bit interrupt gate |
> | 0b0110=0x6 | 16-bit interrupt gate |
>
> Here are some pre-cooked type_attr values people are likely to use (assuming DPL=0):
>
> - 32-bit Interrupt gate: 0x8E (P=1, DPL=00b, S=0, type=1110b => type_attr=1000_1110b=0x8E)

I386 Trap Gate

When an exception occurs, that should correspond to a Trap Gate, and the CPU places the return info o interrupt service routine can resume the interrupted instruction when it calls IRET.

Then, execution is transferred to the given selector:offset from the gate descriptor.

For some exceptions, an error code is also pushed on the stack, which must be POPped before doing IR

Trap and Interrupt gates are similar, and their descriptors are structurally the same, they differ only in th interrupt gates, interrupts are automatically disabled upon entry and reenabled upon IRET which restore

Choosing type_attr values: (See Descriptors#type_attr)

type_attr	Type
0b1111=0xf	32-bit trap gate
0b0111=0x7	16-bit trap gate

Here are some pre-cooked type_attr values people are likely to use (assuming DPL=0):

- 32-bit Trap gate: 0x8F (P=1, DPL=00b, S=0, type=1111b => type_attr=1000_1111b=0x8F)

Thus, Trap and Interrupt gate descriptors hold the following data (other than type_attr):

- 16-bit selector of a code segment in GDT or LDT
- 32-bit offset into that segment - address of the handler, where execution will be transferred

initIdt():

读懂示例：

```
/* Exceptions with error code */
setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
```

在idt表中的位置，代码位于的段；中断处理函数的偏移量；特权等级；

每个中断描述符的具体的信息可以在手册中查到；同时关于第三个参数可以在doIrq.S/idt.c下方的函数定义中查到。

```
 * init your idt here
 * 初始化 IDT 表，为中断设置中断处理函数
 */
/* Exceptions with error code */
setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
setTrap(idt+0xa,SEG_KCODE,(uint32_t)ifqInvalidTSS,DPL_KERN);
setTrap(idt+0xb,SEG_KCODE,(uint32_t)irqSegNotPresent,DPL_KERN);
setTrap(idt+0xc,SEG_KCODE,(uint32_t)irqStackSegFault,DPL_KERN);
setTrap(idt+0xd,SEG_KCODE,(uint32_t)irqGProtectFault,DPL_KERN);
setTrap(idt+0xe,SEG_KCODE,(uint32_t)irqPageFault,DPL_KERN);
setTrap(idt+0x11,SEG_KCODE,(uint32_t)irqAlignCheck,DPL_KERN);
setTrap(idt+0x1e,SEG_KCODE,(uint32_t)irqSecException,DPL_KERN);

/* Exceptions with DPL = 3 */
//填好剩下的表项
setIntr(idt+0x21,SEG_KCODE,(uint32_t)irqKeyboard,DPL_KERN);
setIntr(idt+0x80,SEG_KCODE,(uint32_t)irqSyscall,DPL_USER);
```

下表是关于段/特权等级的宏信息，可以参照示例填写。

```
#define DPL_KERN                0
#define DPL_USER                3

// Application segment type bits
#define STA_X       0x8         // Executable segment
#define STA_W       0x2         // Writeable (non-executable segments)
#define STA_R       0x2         // Readable (executable segments)

// System segment type bits
#define STS_T32A    0x9         // Available 32-bit TSS
#define STS_IG32    0xE         // 32-bit Interrupt Gate
#define STS_TG32    0xF         // 32-bit Trap Gate

// GDT entries
#define NR_SEGMENTS     7           // GDT size
#define SEG_KCODE   1           // Kernel code
#define SEG_KDATA   2           // Kernel data/stack
#define SEG_UCODE   3           // User code
#define SEG_UDATA   4           // User data/stack
#define SEG_TSS     5           // Global unique task state segement
```

doIrq的填写：参照手册

```
.global irqKeyboard
irqKeyboard:
        pushl $0
        pushl $0x21     # TODO: 将irqKeyboard的中断向量号压入栈
        jmp asmDoIrq
```

kernel/main.c 初始化函数的调用

上述部分各自完成之后，此时kernel已经具有初始化的能力，在main.c函数中调用对应功能的函数即可

irqHandle(): 中断处理程序的调用

memory.h 中结构如下：

```
struct TrapFrame {
        uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
        int32_t irq;
};
```

irq是中断号，可以看到文件中给了有限的相关的处理函数；其中GP错误根据手册表格可知中断号是0xd；系统调用的中断号是0x80。

```
switch(tf->irq) {
        //填好中断处理程序的调用
        case -1:break;
        case 0xd: GProtectFaultHandle(tf);break;
        case 0x21:KeyboardHandle(tf);break;
        case 0x80: syscallHandle(tf);break;
        default:assert(0);
    }
```

输出/打印相关 中断服务例程

printf()

读printf()代码可以看到最后都是调用syscall来完成对屏幕的输出的；重点是实现格式符的对应和参数的读取，已经给出了buffer/format/paraList等一系列参数，扫描format并逐个读取参数列表并且判断即可

文件中已经定义好了一些可以把非字符型的变量转换为字符并且填入buffer的函数（dec2Str,hex2Str,etc.），所以找到对应参数后，调用这些函数即可。

```
while(format[i]!=0){
        //in lab2
        ch=format[i++];
        switch(state)
        {
            case 0:
                if(ch=='%')
                    state=1;
                else
                    buffer[count++]=ch;
                break;
            case 1:
                if(ch=='d')
                {
                    decimal=*(int*)paraList;
                    paraList+=4;//指针移动
                    count = dec2Str(decimal,buffer,MAX_BUFFER_SIZE,count);
                    index++;
                }
```

```
                else if(ch=='x')
                {
                    hexadecimal=*(uint32_t*)paraList;
                    paraList+=4;
                    count=hex2Str(hexadecimal,buffer,MAX_BUFFER_SIZE,count);
                    index++;
                }
                else if(ch=='s')
                {
                    string=*(char**)paraList;
                    paraList+=4;
                    count=str2Str(string,buffer,MAX_BUFFER_SIZE,count);
                    index++;
                }
                else if(ch=='c')
                {
                    character=*(char*)paraList;
                    paraList+=4;
                    buffer[count++]=character;
                    if(count==MAX_BUFFER_SIZE)
                    {
                        syscall(SYS_WRITE,STD_OUT,
(uint32_t)buffer,MAX_BUFFER_SIZE,0,0);
                        count=0;//满了就一次性输出
                    }
                    index++;
                }
                else
                {
                    state=2;
                    return ;
                }
                state=0;
                break;
            case 2: return ;
            default : break;

        }

    }
    if(count!=0)
        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
```

syscallPrint()

利用手册给出的内联汇编代码可以在屏幕指定位置上输出，关键在于调整displayCol和displayRow。根据手册可以知道屏幕大小是80*25；即displayCol<=80 displayRow<=25;行满换行，整个页面满了就调用scrollScreen（见vga.c)

```
for (i = 0; i < size; i++) {
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
        if(chracter!='\n')
        {
            data = character | (0x0c << 8);
            pos = (80*displayRow+displayCol)*2;
            asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));//print a
character
```

```
            displayCol++;
            if(displayCol>=80)
            {
                displayCol=0;
                displayRow++;
            }
            if(displayRow>=25)
            {
                displayRow=24;
                displayCol=0;
                scrollScreen();
            }
        }
        else
        {
            displayCol=0;
            displayRow++;
            if(displayRow>=25)
            {
                displayRow=24;
                displayCol=0;
                scrollScreen();
            }
        }
        //完成光标的维护和打印到显存
```

KeyboardHandle()

一边要将读取的字符输出到显示屏上，一边要将字符存入buffer（注意检查buffer是否溢出，buffer当成循环队列来使用）。遇到退格/换行时还需要进行处理

关于退格，可以利用bufferTail实现:

```
uint32_t keyBuffer[MAX_KEYBUFFER_SIZE];
int bufferHead;
int bufferTail;
```

关于大小写: getKeyCode已经完成了切换keyTable的任务，只需要getchar() 就可以读取出对应的值。

```
void KeyboardHandle(struct TrapFrame *tf){
    uint32_t code = getKeyCode();
    if(code == 0xe){ // 退格符
        //要求只能退格用户键盘输入的字符串，且最多退到当行行首
        if(displayCol!=0)
        {
            displayCol--;
            if(bufferTail!=bufferHead)
            {
                keyBuffer[bufferTail]=0;
                bufferTail=(MAX_KEYBUFFER_SIZE+bufferTail-1)%MAX_KEYBUFFER_SIZE;
            }
    }else if(code == 0x1c){ // 回车符
        //处理回车情况
        displayRow++;
```

```
            displayCol=0;
            if(displayRow>=25)
            {
                displayRow=24;
                scrollScreen();
            }
            if((bufferTail+1)%MAX_KEYBUFFER_SIZE!=bufferHead)
            {
                keyBuffer[bufferTail]='\n';
                bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
            }
        }else if(code < 0x81){ // 正常字符
            //注意输入的大小写的实现、不可打印字符的处理
            if((bufferTail+1)%MAX_KEYBUFFER_SIZE!=bufferHead)
            {

                char character=getChar(code);
                keyBuffer[bufferTail]=character;
                bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
                //putChar(character);//print character
                uint16_t data = character | (0x0c << 8);
                int pos = (80*displayRow+displayCol)*2;
                asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
                displayCol++;
                if(displayCol>=80)
                {
                    displayCol=0;
                    displayRow++;
                    if(displayRow>=25)
                    {
                        displayRow=24;
                        displayCol=0;
                        scrollScreen();
                    }
                }
            }
        }

        updateCursor(displayRow, displayCol);
}
```

syscallGetChar()/syscallGetStr()

函数定义位于在中断处理模块中；syscallRead()最终调用的是这两个函数；或许可以参照syscallPrint()；用到tf；一边写入keyBuffer一边将字符存入tf寄存器中关于字符（串）所在地址。

键码的部分参照了手册中的链接。

```
void syscallGetChar(struct TrapFrame *tf){
    //自由实现
    int sel = USEL(SEG_UDATA); // segment selector for user data, need further
modification
    char* str = (char*)tf->edx;
    //int size = tf->ebx;
    int i=0;
    char character=0;
```

```
    uint32_t code=0;
    asm volatile("movw %0, %%es"::"m"(sel));
    while(code==0)
    {
        code=getKeyCode();
        character=getChar(code);
        putChar(character);
    }
    asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
    i++;
    asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
    bufferTail=bufferHead;
    return;
}
/*syscallGetStr  一开始希望直接在内部调用getkeycode来获取键码，但是发现会重复。考虑在函数中开中断，由于sti只是控制是否接受中断请求，需要人为构造死循环来等待键盘输入*/
void syscallGetStr(struct TrapFrame *tf){
    //自由实现
    asm volatile("sti");
    int sel = USEL(SEG_UDATA);
    char *str = (char*)tf->edx;
    int size=tf->ebx;
    int i=0;
    char character=0;
    uint32_t code=0;
    asm volatile("movw %0, %%es"::"m"(sel));
    asm volatile("sti"::);
    while(1)
    {
        code=getKeyCode();
        character=keyBuffer[bufferTail-1];
        if(character=='\n' || code==0x1c || code==0x9c)
            break;
    }
    //KeyboardHandle(tf);
    while(i < size-1){
        if(bufferHead!=bufferTail) {
            bufferHead=(bufferHead+1)%MAX_KEYBUFFER_SIZE;//把指针后移，读下一个
            character=keyBuffer[bufferHead];
            putChar(character);
            if(character != 0 && character !='\n') {
                asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str+i));
                i++;
            }
        }
        else
            break;
        }
    asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
    return ;
}
```

getChar()/getStr()

库函数的一种；需要在内核态和用户态之间切换；参考printf 调用思路应当是先调用syscall，syscall根据调用的信息调用syscallGetChar/syscallGetStr

getChar最终实现的效果是输入后立刻返回第一个字符，getStr是会返回回车键前出现的字符

syscall会用到的参数如下：

```
#define SYS_WRITE 0 # 可修改 可自定义
#define STD_OUT 0 # 可修改 可自定义

#define SYS_READ 1 # 可修改 可自定义
#define STD_IN 0 # 可修改 可自定义
#define STD_STR 1 # 可修改 可自定义
```

\lib\lib.h

采用的方式是直接调用syscal，将必要的参数传入，剩下的交给sysGetChar和sysGetStr

Bug与解决方案

闪屏：段选择符没有向左移位 低三位应该是0

printf字符显示部分出错: printf的参数列表开始的位置应当避开format的长度

键盘按下没有反应:中断处理程序中没有填写键盘对应的处理程序

getstr中调用keyCode出现满屏重复字符：改用其他实现方式

总结和感想

在较为复杂的项目面前，读懂框架代码是磨刀不误砍柴工的；把各部分的关系，需要补充的部分，可能用到的信息搞明白可以方便更快的完成实验。

通过实验以及和同学的交流，对中断和利用中断的I/O机制有了更深入的认识。

> 实验用时：3+5+3+2+3+3+1=20h