OS-lab2

211275009 陈铭浩
email：**211275009@smail.nju.edu.cn**
实验进度：我完成了所有内容

1. 中断机制

1.1 从实模式进入保护模式并加载内核（elf文件）到内存，跳转运行：`bootMain`

1. 修改boot.c，对ELF文件头魔数的检查，以确保加载的内容确实是一个ELF文件
2. kMainEntry 被更新为指向ELF头部中指定的入口点地址；phoff 更新为从ELF头部读取的程序头表偏移；offset 更新为第一个程序头（.text）中指定的段偏移。
3. 将从 elf + offset 开始的数据拷贝回 elf 开始的位置，覆盖200个扇区长度的数据。这个过程实际上是将ELF文件中偏移后的程序代码复制到其预期的运行位置。

```
// TODO: 阅读boot.h查看elf相关信息，填写kMainEntry、phoff、offset
ELFHeader *eh = (ELFHeader *)elf;

if (eh->magic == 0x464C457F) { // Check for ELF magic number
    phoff = eh->phoff;
    ProgramHeader *ph = (ProgramHeader *)((unsigned char *)eh + phoff);
    offset = ph->off;
    kMainEntry = (void(*)(void))(eh->entry);
}

for (i = 0; i < 200 * 512; i++) {
        *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
}

kMainEntry();
```

1.2 初始化串口输出：`initSerial`（框架中已实现）

1.3 初始化中断向量表：`initIdt`

1. 修改idt.c

- 完善setIntr()和setTrap()函数
    - 中断门和陷阱门的present（此位用于标示门描述符是否有效。设置为1表示有效，设置为0表示该中断门或陷阱门未使用或无效）都设为1
    - system（系统位，用于区分系统段（0）和代码或数据段（1））都设为0
    - 注意：段选择器（selector）本身并不直接放在中断描述符表（IDT）中。IDT中的条目使用的是段选择子（segment selector）和偏移量（offset）来定位中断处理程序的代码。因此idt表中的segment值应使用宏定义的KSEL（）处理selector后再填入（左移三位后与DPL进行按位或运算）
- 完善initIdt()函数
    - 利用上面定义好的两个函数，初始化IDT表项，为中断设置中断处理函数

2. 修改doIrq.S

- 完善irqKeyboard函数，将其中断向量号0x21压入栈

1.4 初始化8259a中断控制器：`initIntr`（框架中已实现）

1.5 初始化 GDT 表、配置 TSS 段：`initSeg`（框架中已实现）

> *IA-32提供了4个特权级, 但TSS中只有3个堆栈位置信息, 分别用于ring0, ring1, ring2的堆栈切换.为什么TSS中没有ring3的堆栈信息?*
>
> *硬件堆栈切换只会在目标代码特权级比当前堆栈特权级高的时候发生（低特权级->高特权级），而ring3是最低的特权级，故TSS没有存储ring3的堆栈信息的需求*

1.6 初始化VGA设备：`initVga`（框架中已实现）

1.7 配置好键盘映射表：`initKeyTable`（框架中已实现）

1.8 从磁盘加载用户程序到内存相应地址，进入用户空间：`loadUMain`、`enterUserSpace`

1. 参照bootloader加载内核的方式，由kernel加载用户程序，用户程序在内存中的位置为0x200000

---

2. 完善键盘按键中断服务例程

2.1 完善irqHandle函数：`irqHandle`

1. 填好对应中断处理程序的调用，根据中断向量号分别调用不同的中断处理程序

2.2 完善KeyboardHandle函数：`KeyboardHandle`

1. 特殊处理回车和退格的情况（框架中已实现）（但此处处理退格时还要注意维护好keyBuffer，以便getChar和getStr的正常使用）
2. 处理正常的字符，利用putChar串口输出接口完成按键回显，此处还要注意换行与滚动屏幕的逻辑
3. 有一些字符没有按键回显（如ctrl，shift，capslock等）

```
}else if(code < 0x81){
    // TODO: 处理正常的字符
    if(code==0x3a||code==0x2a||code==0x36||code==0x1d){
        return;
    }
    char ch=getChar(code);
    keyBuffer[bufferTail++]=ch;
    //putNum(bufferHead);
    //putNum(bufferTail);
    uint16_t data = ch | (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++;
        tail=0;
        if(displayRow==25){
            scrollScreen();
            displayRow=24;
        }
    }
    //putChar(keyBuffer[bufferTail-1]);
    //putChar('\n');
```

---

3. 实现系统调用库函数printf和对应的处理例程

3.1 实现printf的处理例程：`syscallPrint`

1. 单独考虑换行符（\n）
2. 其他字符直接打印到显存，同样注意换行与滚动屏幕的逻辑

3.2 完善printf的格式化输出

1. 使用宏定义好的va_list记录可变参数列表
2. 在遇到字符串终止符前（\0），分四种情况处理格式化字符串（整数、十六进制数、字符串和字符），对每个遇到的格式说明符（%d, %x, %s, %c），执行相应的转换函数（如 dec2Str, hex2Str, str2Str），这些函数负责将数据转换为字符串并存入 buffer 中

3. 输出控制：当 buffer 填满或处理完所有输入后，通过系统调用 syscall 发送到标准输出

---

4. 实现getChar，getStr的处理例程

> *getChar函数返回键盘输入的一个字符，getStr返回键盘输入的一条字符串*
>
> *此处参考了c语言中相关函数的处理逻辑*

4.1 实现getChar的处理例程：`getChar`，`syscallGetChar`

1. syscallGetChar()中以换行符为输入结束标志，识别到换行符后，首先将键盘缓冲区尾部的换行符全部去掉，然后将缓冲区头部第一个字符返回即可
2. getChar()中直接返回调用syscall即可

```
void syscallGetChar(struct TrapFrame *tf){
    // TODO: 自由实现
    int end=0;
    //扔掉回车符
    while(keyBuffer[bufferTail-1]=='\n' && bufferTail>bufferHead){
        //putNum(bufferHead);
        //putNum(bufferTail);
        keyBuffer[--bufferTail]='\0';
        end=1;
    }
    if(bufferTail>bufferHead&&end==1){
        char ch=keyBuffer[bufferHead];
        tf->eax=ch;
        bufferHead++;
        return;
    }
    tf->eax=0;
```

4.2 实现getStr的处理例程：`getStr`，`syscallGetStr`

1. syscallGetStr()中以换行符为输入结束标志，识别到换行符后，首先将键盘缓冲区尾部的换行符全部去掉，然后检查缓冲区中是否已经达到了最大长度或者已经输入了换行符，使用内联汇编指令将缓冲区中的字符复制到用户空间的字符串中即可
2. getStr()中调用syscall

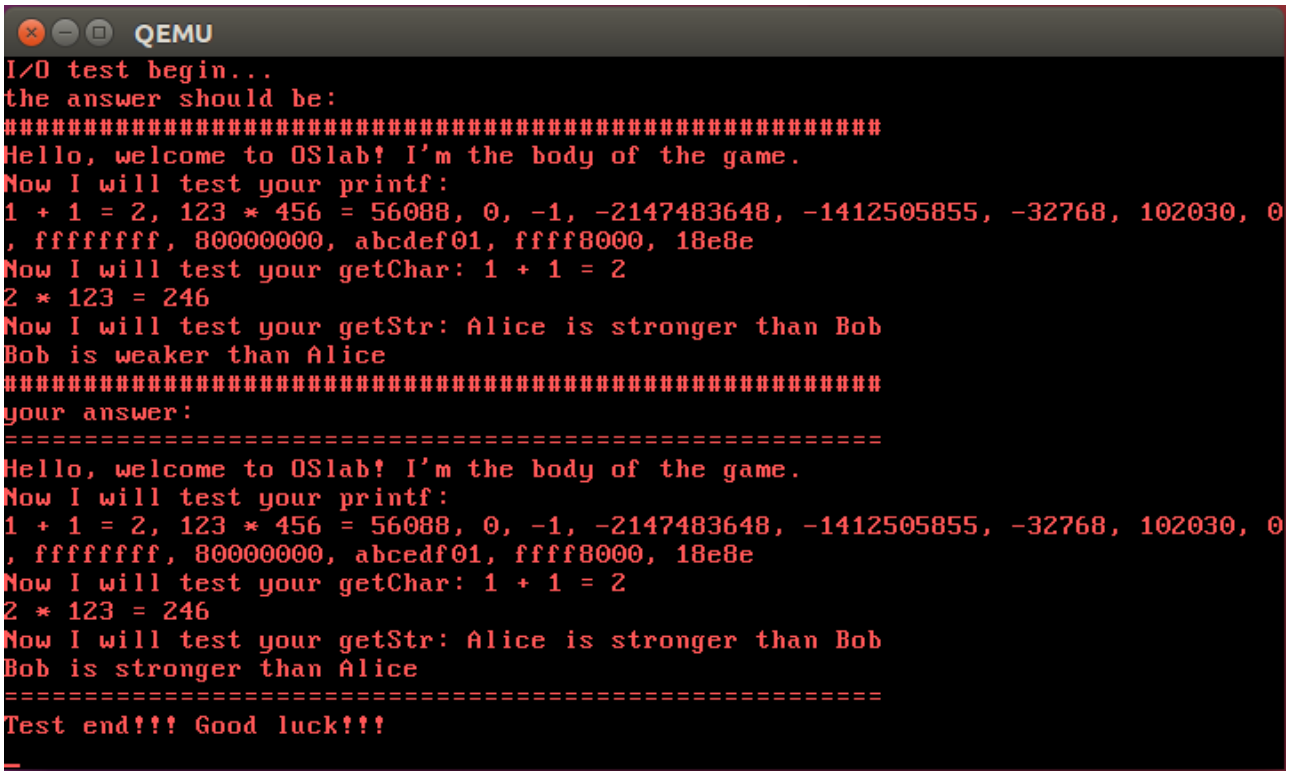
```
void syscallGetStr(struct TrapFrame *tf){
    // TODO: 自由实现
    char *str = (char *)tf->edx;
    int maxSize = tf->ebx;
    int end=0;
    //int sel = USEL(SEG_UDATA);
    //asm volatile("movw %0, %%es"::"m"(sel));
    //putNum(bufferHead);
    //putNum(bufferTail);
    //扔掉回车符
    while(keyBuffer[bufferTail-1]=='\n' && bufferTail>bufferHead){
        //putNum(bufferHead);
        //putNum(bufferTail);
        keyBuffer[--bufferTail]='\0';
        end=1;
    }
    //两种退出情况
    if(end==1 || bufferTail-bufferHead>=maxSize){
        int size=0;
        if(bufferTail-bufferHead<maxSize){
            size=bufferTail-bufferHead;
        }else{
            size=maxSize;
        }
        for(int i=0;i<size;i++){
            asm volatile("movb %1, %%es:(%0)"::"r"(str+i), "r"(keyBuffer[bufferHead+i]));
        }
        tf->eax = 1;
        return;
    }
    tf->eax=0;
```

---

5. 实验结果

在lab2文件夹下执行如下命令：

```
chmod +x utils/genBoot.pl # 需要首先将genBoot.pl文件提权
chmod +x utils/genKernel.pl # 需要首先将genKernel.pl文件提权
make
make play
```


QEMU
I/O test begin...
the answer should be:
##
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
##
your answer:
==
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is stronger than Alice
==
Test end!!! Good luck!!!

测试结果正常