Lab2 实验报告

实验进度

本次实验我完成了所有内容。

实验步骤以及结果

1. 装载内核
   修改bootloader/boot.c中代码

```
// TODO: 阅读boot.h查看elf相关信息，填写kMainEntry、phoff、offset
 ELFHeader* elfhead = (ELFHeader*)elf;
 phoff = elfhead->phoff;
 ProgramHeader* phead = (void*)elf + elfhead->phoff;
 offset = phead->off;


 for (i = 0; i < 200 * 512; i++) {
                *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
        }
 kMainEntry = (void(*)(void))(elfhead->entry);
 kMainEntry();
```

2. 在内核中完善中断机制、提供系统服务函数
   中断描述符表：
   修改kernel/kernel/idt.c中代码：
   注意门描述符每个属性对应关系，注释有介绍。

```
/* 初始化一个中断门(interrupt gate) */
static void setIntr(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {
        // TODO: 初始化interrupt gate
        selector = (dpl == DPL_KERN) ? KSEL(selector) : USEL(selector);
        ptr->offset_15_0 = offset & 0xffff;//偏移量低15位
        ptr->segment = selector;//中断服务例程代码段选择子
        ptr->pad0 = 0;//not used
        ptr->type = INTERRUPT_GATE_32;//1110
        ptr->system = 0;//是否为系统描述符
        ptr->privilege_level = dpl;//所需特权级别
        ptr->present = 1;//门描述符是否有效
        ptr->offset_31_16 = (offset >> 16) & 0xffff;//偏移量高15位
}

/* 初始化一个陷阱门(trap gate) */
static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {
        // TODO: 初始化trap gate
        selector = (dpl == DPL_KERN) ? KSEL(selector) : USEL(selector);
        ptr->offset_15_0 = offset & 0xffff;
        ptr->segment = selector;
        ptr->pad0 = 0;
        ptr-> type = TRAP_GATE_32;
        ptr->system = 0;
        ptr->privilege_level = dpl;
        ptr->present = 1;
        ptr->offset_31_16 = (offset >> 16) & 0xffff;
}
//256 interrupt vector
void initIdt() {
        int i;
        /* 为了防止系统异常终止，所有irq都有处理函数(irqEmpty)。 */
        for (i = 0; i < NR_IRQ; i ++) {
                setTrap(idt + i, SEG_KCODE, (uint32_t)irqEmpty, DPL_KERN);
        }
        /*init your idt here 初始化 IDT 表，为中断设置中断处理函数*/

        // TODO: 参考上面第48行代码填好剩下的表项
        //将前面声明的那些函数逐个加进去(中断处理例程)
        setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
        setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, 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);
        setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
        setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);

        /* 写入IDT */
        saveIdt(idt, sizeof(idt));//use lidt
}
```

中断控制器:
main.c里调用initIntr()。

全局描述符表、任务状态段:
main.c里调用initSeg()。

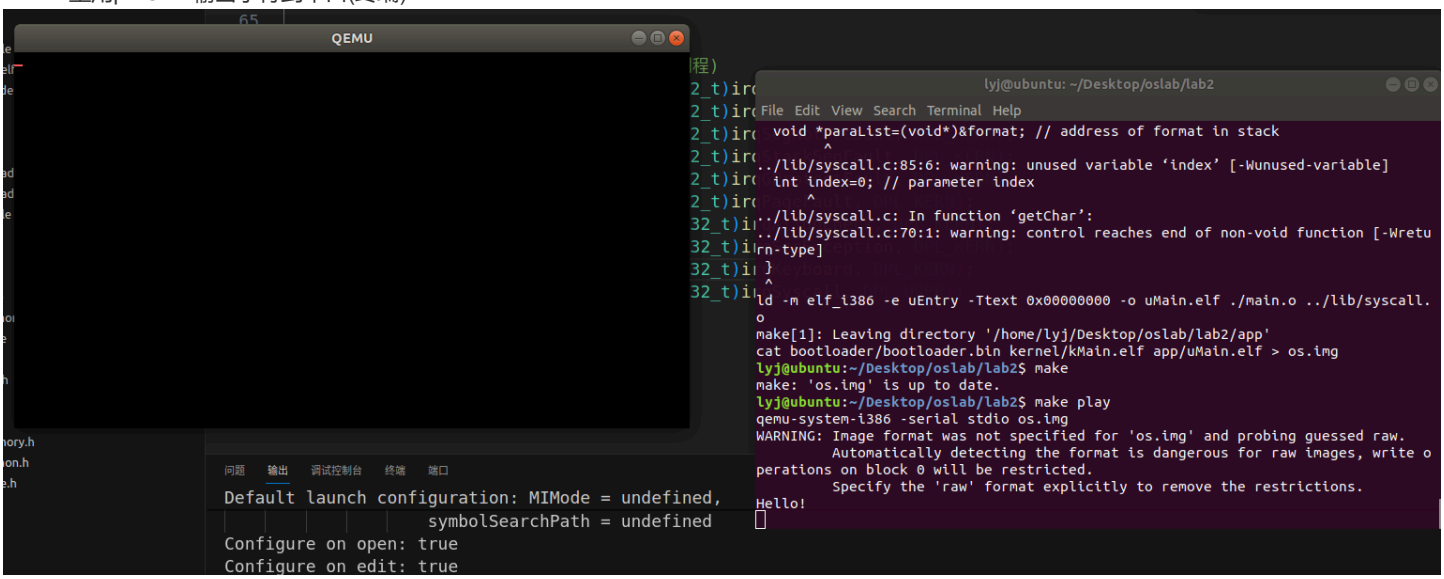
完善中断处理程序:
根据中断号来分类处理。

```
void irqHandle(struct TrapFrame *tf) { // pointer tf = esp
	/*
	 * 中断处理程序
	 */
	/* Reassign segment register */
	asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));

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

串口输出检查：

main.c里用putChar输出字符到串口(终端):



输出了"Hello!"

3. 内核中加载用户程序

修改kernel/kernel/kvm.c中代码:

类似bootloader加载内核的方式，也是读取elf相关信息并加载。

这里要注意，用户程序是加载至物理内存0x200000开始，用户程序从磁盘第201扇区开始。

```
void loadUMain(void) {
        // TODO: 参照bootloader加载内核的方式，由kernel加载用户程序
        int i = 0;
        int phoff = 0x34;
        int offset = 0x1000;
        unsigned int uelf = 0x200000;//用户程序加载至物理内存 0x200000 开始的位置
        unsigned int uMainEntry = 0x200000;

        for (i = 0; i < 200; i++) {
                readSect((void*)(uelf + i*512), 201+i);//201以后是用户程序部分
        }
        struct ELFHeader* elfhead = (void*)uelf;
        uMainEntry = elfhead->entry;
        phoff = elfhead->phoff;
        offset = ((struct ProgramHeader *)(uelf + phoff))->off;
        for (i = 0; i < 200 * 512; i++) {
                        *(unsigned char *)(uelf + i) = *(unsigned char *)(uelf + i + offset);
                }
        //putChar('i');
        enterUserSpace(uMainEntry);
}
```

4. printf格式化输出以及其他系统调用实现

对于printf格式化输出，采用va_list可变参数列表，这样实现更加方便。

```
void printf(const char *format,...){//照着上学期PA实验里的klib打
        int i=0; // format index
        char buffer[MAX_BUFFER_SIZE];
        int count=0; // buffer index
        //int index=0; // parameter index
        //void *paraList=(void*)&format; // address of format in stack
        //int state=0; // 0: legal character; 1: '%'; 2: illegal format
        int decimal=0;
        uint32_t hexadecimal=0;
        char *string=0;
        char character=0;
        va_list ap;
        va_start(ap, format);
        while(format[i]!=0){
                // TODO: support format %d %x %s %c
                if(format[i] == '%'){
                        ++i;
                        switch (format[i]){
                        case 'd':
                                decimal = va_arg(ap, int);//十进制整数
                                count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
                                break;
                        case 'x':
                                hexadecimal = va_arg(ap, uint32_t);//十六进制无符号整数
                                count = hex2Str(hexadecimal, buffer, MAX_BUFFER_SIZE, count);
                                break;
                        case 's':
                                string = va_arg(ap, char*);
                                count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
                                break;
                        case 'c':
                                character = va_arg(ap, char);
                                buffer[count++] = (char)character;
                                break;
                        default:
                                break;
                        }
                }
                else{
                        buffer[count++] = format[i];
                }

                if(count == MAX_BUFFER_SIZE){//缓冲区满，输出
                        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
                        count = 0;
                }
                i++;
        }
        if(count!=0){//缓冲区还有内容，输出
                syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
        }
        va_end(ap);
        return;
}
```

那么与之对应，需要完善kernel/kernel/irqHandle.c中相关内容

KeyboardHandle中完善处理正常字符的部分，还是一样用内敛汇编把键码写入vga对应地址(基址0xb8000)。注意换行和满屏的特判。syscallPrint中打印到显存和光标维护与上面所述类似。

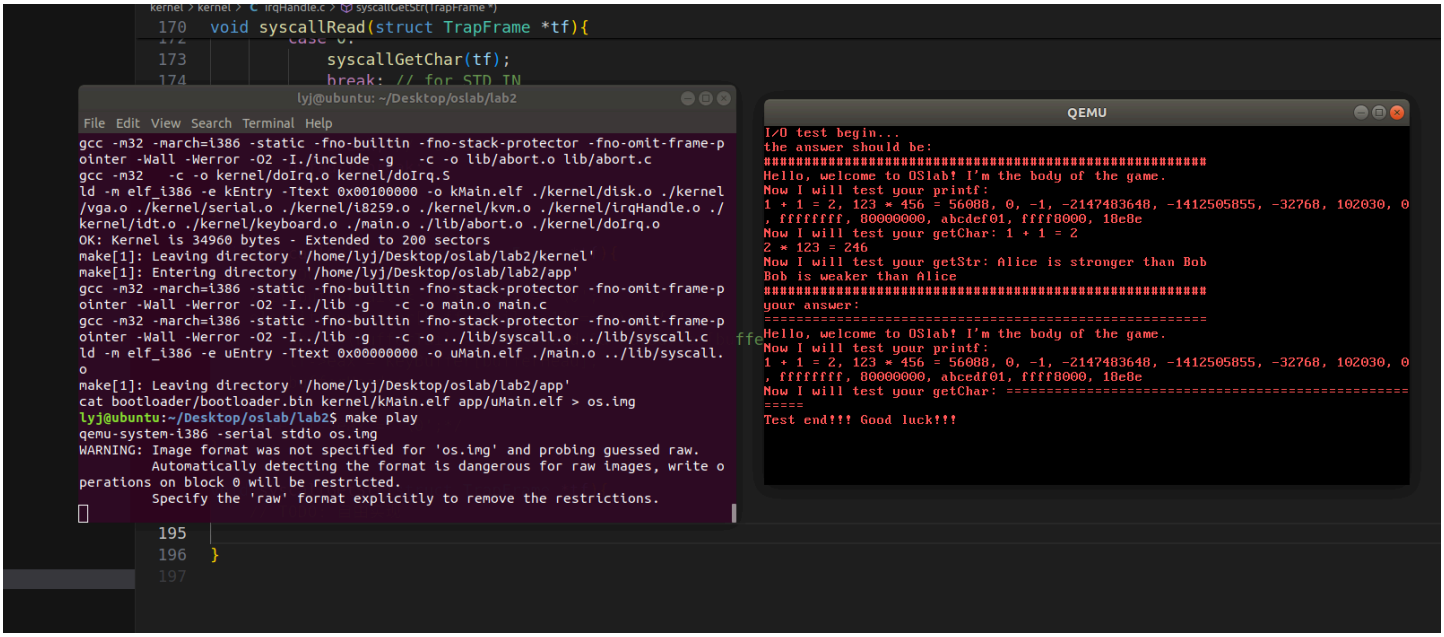
下面就是完成getChar和getStr了。

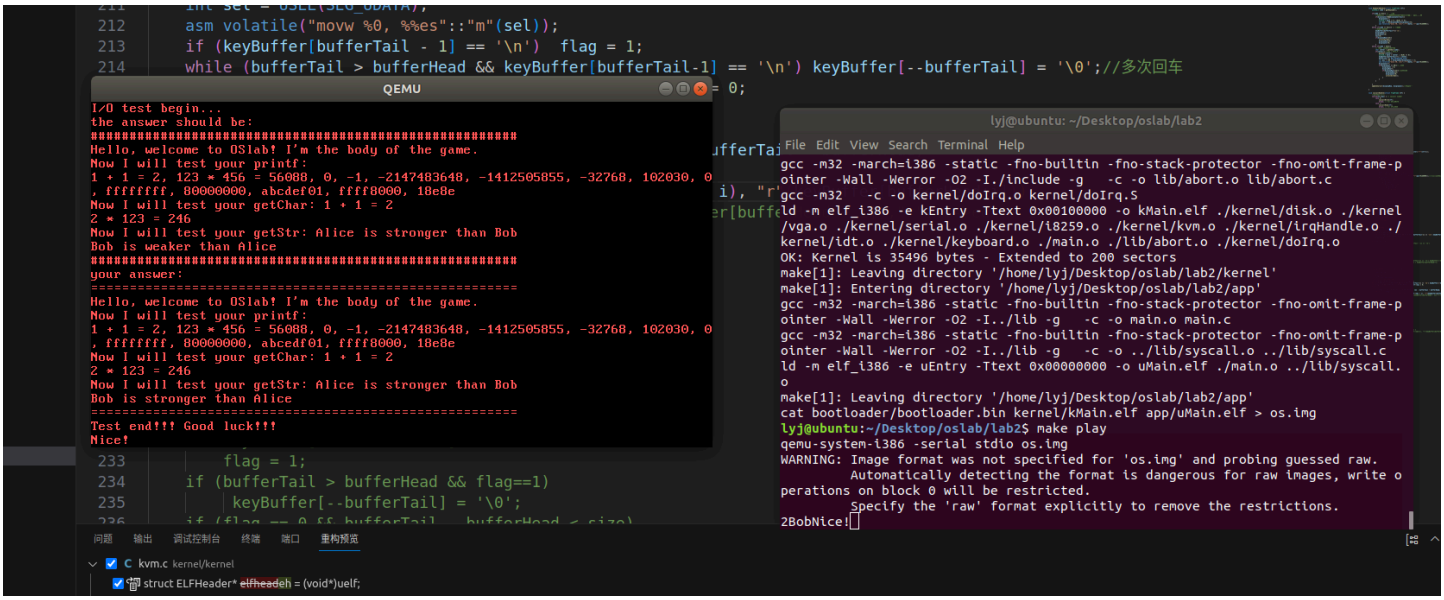
我选择用$syscall(SYS_READ, STD_IN, \cdots)$系统调用的返回值来判断是否输入完成，为0就是还未完成(等待输入)，否则输入完成，getChar和getStr返回。

在syscallGetChar和syscallGetStr中，我进行多次回车判断。两个函数都以回车键为输入终止标准。syscallGetChar将输入字符写入eax寄存器；syscallGetStr则是通过内敛汇编将键缓冲区中的字符串输入到指定地址，eax设置为syscall系统调用返回值，用来判断输入是否结束。

最后测试效果:

打印测试以及输入测试全部正确。测试完成后依然可以输入字符("Nice!")，说明键盘输入中断实现正确。





自由报告

由于一些奇奇怪怪的原因(bug)，我写了两天半(时间正确)才写完这个实验。开始idt.c里面有个地方的高位offset右移十六位写成了左移十六位。这里特别感谢助教老师的帮助！

一个很奇怪的bug：一般都用串口输出函数putChar来进行调试，之前我一直认为没有进入用户空间，所以printf没有执行，为此在系统调用的一些列代码里加putChar串口输出来看是否执行。结果非常离谱，开始printf没执行。在我改正idt.c后，printf函数有部分执行，部分没执行，并且若注释掉部分printf，剩下printf又能执行。这种“薛定谔的printf”使我百思不得其解。之后各种尝试，6.25小时后(2.5^2)，我一次偶然地把系统调用相关函数里的putChar注释调后，我的printf竟然奇迹般地恢复了！经过我不大的大脑思考后，我认为可能是putChar会与printf竞争中断(?)，毕竟这里的实现的中断是没有优先级的，也就是没有中断队列。

此次实验加深了我对中断、异常的理解，更深入地理解了系统调用的过程。