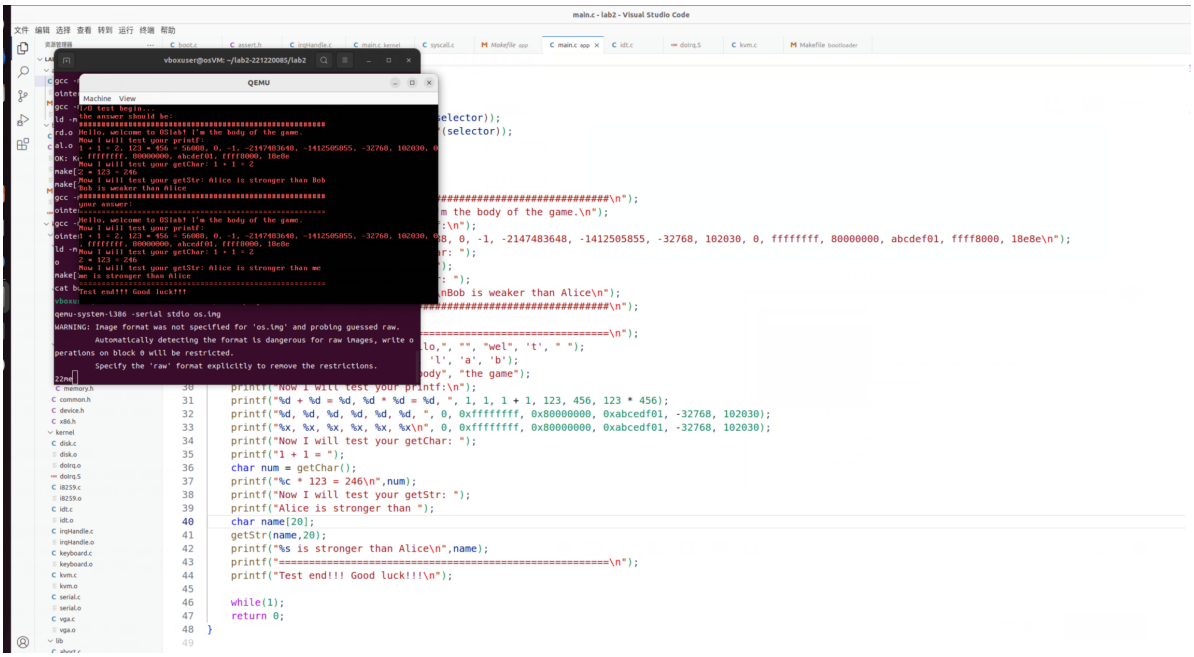
OS-lab2-report

姓名	时昌军
学号	221220085
邮箱	221220085@smail.nju.edu.cn

一、实验进度

3个实验要求全部完成

二、实验结果



三、实验修改的代码

1. 磁盘加载，即引入内核，bootloader加载kernel，由kernel加载用户程序

```
//boot.c
// TODO: 阅读boot.h查看elf相关信息，填写kMainEntry,装载内核
kMainEntry = (void(*)(void))((struct ELFHeader *)elf)->entry;

//kvm.c，由内核加载用户程序
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 offset = 0x1000; // .text section offset
    uint32_t elf = 0x200000; // physical memory addr to load
    uint32_t uMainEntry;
     // entry address of the program
    for (i = 0; i < 200; i++) {
        readSect((void*)(elf + i*512), 201+i);
    }
```

```
    // TODO: 阅读boot.h查看elf相关信息，填写kMainEntry
    uMainEntry = ((struct ELFHeader *)elf)->entry;
    for (i = 0; i < 200 * 512; i++) {
            *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i +
offset);
        }
    enterUserSpace(uMainEntry);
}
```

2. 开始区分内核态和用户态，完善中断机制

涉及文件 kernel/kernel/idt.c , 需要实现中断门和陷阱门的设置，并仿照已有表项填好IDT中断表中的剩余表项。

```
                            80386 INTERRUPT GATE
 31                23                15                7                0
+-----------------+-----------------+---+---+---------+-----+-----------+
|           OFFSET 31..16           | P |DPL|0 1 1 1 0|0 0 0|(NOT USED) |4
|-----------------------------------+---+---+---------+-----+-----------|
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+

                            80386 TRAP GATE
 31                23                15                7                0
+-----------------+-----------------+---+---+---------+-----+-----------+
|          OFFSET 31..16            | P |DPL|0 1 1 1 1|0 0 0|(NOT USED) |4
|-----------------------------------+---+---+---------+-----+-----------|
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+
```

```
//idt.c
/* 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

    ptr->offset_15_0 = offset & 0xFFFF;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = INTERRUPT_GATE_32;
    ptr->system = FALSE;
    ptr->privilege_level = dpl;
    ptr->present = TRUE;
    ptr->offset_31_16 = (offset >> 16) & 0xFFFF;
}

/* 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
    ptr->offset_15_0 = offset & 0xFFFF;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = TRAP_GATE_32;
    ptr->system = FALSE;
    ptr->privilege_level = dpl;
    ptr->present = TRUE;
```

```
    ptr->offset_31_16 = (offset >> 16) & 0xFFFF;
}
// TODO: 参考上面第48行代码填好剩下的表项
    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);
```

3. 通过实现用户态I/O函数介绍基于中断实现系统调用的全过程

```
//irqHandle.c
else if(code < 0x81){
// TODO: 处理正常的字符
char character=getChar(code);
if(character!=0){
    putChar(character);
    keyBuffer[bufferTail++]=character;
    bufferTail%=MAX_KEYBUFFER_SIZE;
    //将字符 character 显示在屏幕的 displayRow 行 displayCol 列
    uint16_t data = character | (0x0c << 8);
    int pos = (80*displayRow+displayCol)*2;
    asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));

    displayCol+=1;
    if(displayCol==80){
        displayCol=0;
        displayRow++;
        if(displayRow==25){
            scrollScreen();
            displayRow=24;
            displayCol=0;
        }
    }
}

// TODO: 完成光标的维护和打印到显存
if (character == '\n')
{
    displayRow++;
    displayCol = 0;
    if (displayRow == 25){
        displayRow = 24;
        displayCol = 0;
        scrollScreen();
    }
}
else
{
```

```
    data = character | (0x0c << 8);
    pos = (80 * displayRow + displayCol) * 2;
    asm volatile("movw %0, (%1)" ::"r"(data), "r"(pos + 0xb8000));
    displayCol++;
    if (displayCol == 80){
        displayRow++;
        displayCol = 0;
        if (displayRow == 25){
            displayRow = 24;
            displayCol = 0;
            scrollScreen();
        }
    }
}

void syscallGetChar(struct TrapFrame *tf){
    // TODO: 自由实现
    keyBuffer[0]=0;
    keyBuffer[1]=0;
    char c=0;
    while(c == 0){
        enableInterrupt();/* 打开外部中断 */
        c = keyBuffer[0];
        putChar(c);
        disableInterrupt();/* 关闭外部中断 */
    }
    tf->eax=c;
    char wait=0;
    while(wait==0){
        enableInterrupt();
        wait = keyBuffer[1];//等待用户按下回车键来确认输入
        disableInterrupt();
    }
    return;
}

void syscallGetStr(struct TrapFrame *tf){
    // TODO: 自由实现
    char* str=(char*)(tf->edx);//str pointer
    int size=(int)(tf->ebx);//str size
    bufferHead=0;
    bufferTail=0;
    //for(int j=0;j<MAX_KEYBUFFER_SIZE;j++)keyBuffer[j]=0;//init
    int j=0;
    while(j<MAX_KEYBUFFER_SIZE){
        keyBuffer[j]=0;
        j++;
    }
    int i=0;
    //该循环会从键盘缓冲区中读取字符，直到遇到换行符 \n 或者达到指定的字符数 size
    char c=0;
    while(c!='\n' && i<size){
//在内部 while 循环中,我们等待键盘缓冲区中的字符不再为零（即有输入）。一旦有输入,我们将其存储在 c 变量中,并递增计数器 i。
        while(keyBuffer[i]==0){
```

```
            enableInterrupt();
        }
        c=keyBuffer[i];
        i++;
        disableInterrupt();
    }

    int selector=USEL(SEG_UDATA);//初始化一个整数变量 selector，其值与用户数据段相关
    asm volatile("movw %0, %%es"::"m"(selector));//将 selector 的值移动到额外段寄存器（ES）中。
    j=0;
    for(int p=0;p<i-1;p++){
        asm volatile("movl %0, %%es:(%1)"::"r"(keyBuffer[p]),"r"(str+j));//这行将 keyBuffer[p] 的一个字节复制到 str+k
        j++;
    }
    asm volatile("movl $0x00, %%es:(%0)"::"r"(str+i));//在 str 缓冲区的末尾写入一个空终止符（0x00）
    return;
}

//实现用户层面函数调用
char getChar(){ // 对应SYS_READ STD_IN
    // TODO: 实现getChar函数，方式不限
    char c = 0;
    return syscall(SYS_READ, STD_IN,(uint32_t)c,1,0,0);
}

void getStr(char *str, int size){ // 对应SYS_READ STD_STR
    // TODO: 实现getStr函数，方式不限
    syscall(SYS_READ, STD_STR, (uint32_t)str,  (uint32_t)size, 0, 0);
    return;
}

// TODO: support format %d %x %s %c
//%d表示按整型数据的实际长度输出数据。
//%c用来输出一个字符。
//%s用来输出一个字符串。
//%x表示以十六进制数形式输出整数。
    buffer[count] = format[i];
    count++;

    if (format[i] == '%'){
        count--;
        i++;
        paraList += sizeof(char *);
        switch (format[i]){
            case 'c':
                character = *(char *)paraList;
                buffer[count++] = character;
                break;
            case 's':
                string = *(char **)paraList;
```

```
143                 count = str2Str(string, buffer, (uint32_t)MAX_BUFFER_SIZE,
    count);
144                 break;
145             case 'x':
146                 hexadecimal = *(uint32_t *)paraList;
147                 count = hex2Str(hexadecimal, buffer,
    (uint32_t)MAX_BUFFER_SIZE, count);
148                 break;
149             case 'd':
150                 decimal = *(int *)paraList;
151                 count = dec2Str(decimal, buffer, (uint32_t)MAX_BUFFER_SIZE,
    count);
152                 break;
153             case '%'://输出%
154                 paraList -= sizeof(format);
155                 count++;
156                 break;
157         }
158     }
159     //clean buffer
160     if (count == MAX_BUFFER_SIZE)
161     {
162         syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer,
    (uint32_t)MAX_BUFFER_SIZE, 0, 0);
163         count = 0;
164     }
165     i++;
166
```

四、实验时遇到的问题以及解决方法

1. 问题：make os.img的时候显示bootloader.bin太大，kernel加载不进来

```
gcc -m32   -c -o start.o start.S
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-p
ointer -Wall -Werror -O2 -g   -c -o boot.o boot.c
ld -m elf_i386 -e start -Ttext 0x7c00 -o bootloader.elf ./start.o ./boot.o
objcopy -O binary bootloader.elf bootloader.bin
ERROR: boot block too large: 1000 bytes (max 510)
make[1]: *** [Makefile:34: bootloader.bin] Error 1
```

2. 解决：删去boot.c和loadUMain里面的phoff,并更改bootloader的makefile

五、思考题

1. **ring3的堆栈在哪里? 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?**

答：RING0（内核态）：操作系统内核运行在这个最高特权级别上。它可以执行特权指令，控制中断、修改页表、访问设备等等。内核代码运行在这里。

RING1和RING2：这两个级别在实际操作系统中很少使用，通常保留给特定硬件或虚拟化方案。它们的权限介于RING0和RING3之间。

RING3（用户态）：应用程序的代码运行在这个最低特权级别上。它不能执行特权指令，不能直接访问硬件资源，必须通过系统调用来请求内核执行特权操作。

堆栈位置信息：

- IA-32体系结构中的TSS（任务状态段）只包含了三个堆栈位置信息，用于RING0、RING1和RING2的堆栈切换。这是因为RING3的堆栈切换不需要TSS来管理。
- RING3的堆栈切换是由操作系统内核负责的。当应用程序执行系统调用时，CPU会从RING3切换到RING0，内核会为其分配一个新的堆栈，执行相应的内核代码，完成特权操作，然后再切换回RING3。

总之，RING3的堆栈切换不需要TSS来管理，因为它是由操作系统内核动态分配和管理的。

2. **保存寄存器的旧值。我们在使用eax, ecx, edx, ebx, esi, edi前将寄存器的值保存到了栈中，如果去掉保存和恢复的步骤，从内核返回之后会不会产生不可恢复的错误?**

答：会。例如 eax 可能会保存函数返回值，而响应中断会使用 eax ，从而会丢失 eax 的 信息。

六、实验心得

和这次实验相比，上次实验就是宝宝巴士。。。这次实验被卡住了好几次，首先是 四 中遇到的问题，和好几个同学交流后才得出了解决方案，应该是虚拟机版本导致的。还有就是实现 `syscallGetChar` 和 `syscallGetStr` 时遇到了较大的困难，查了很多资料才把写出来。希望下次实验能顺利一些。(●'◡'●)