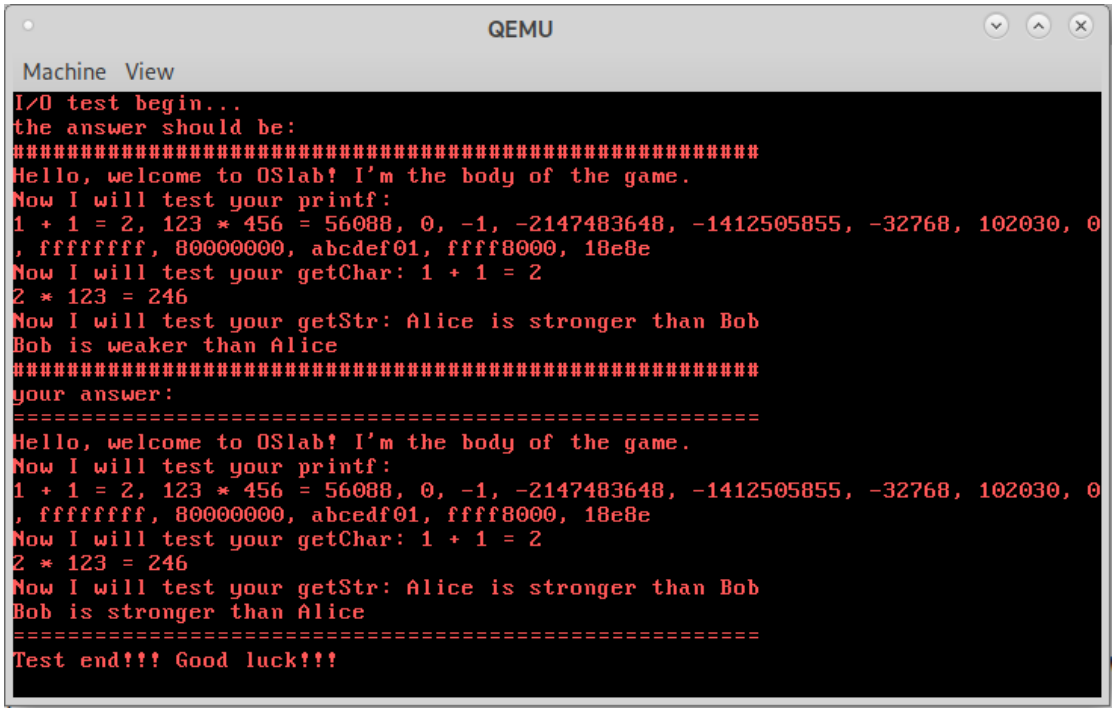
姓名：张涵之　　学号：191220154　　邮箱：1683762615@qq.com

注意事项：我用虚拟机打包传到课程网站以后，尝试把压缩包下载解压，直接 make 可能会报错 Permission denied，建议先对 lab 文件夹 chmod 修改权限以后再 make。

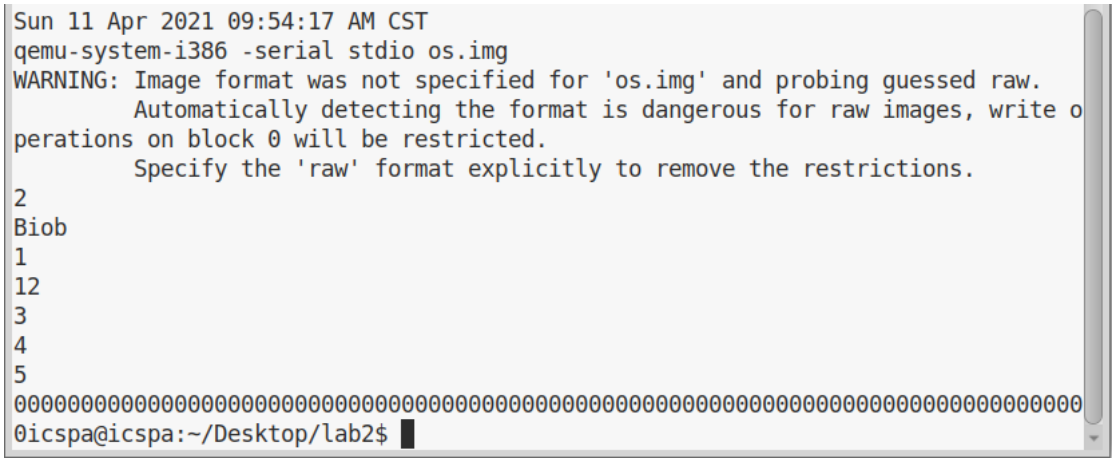
实验进度：我完成了所有内容

实验结果：下图为官方测试代码运行效果



下图为自动换行、换行符换行及滚屏测试

下图为控制台输出结果，其中测试了两次删除，第二行的 i 和第四行的 1



实验修改的代码位置：

bootloader/boot.c　　//对 kernel 代码的装载

```
30    // TODO: 填写kMainEntry phoff offset
31    kMainEntry = (void(*)(void))((struct ELFHeader *)elf)->entry;
32    phoff = ((struct ELFHeader *)elf)->phoff;
33    offset = ((struct ProgramHeader *)(elf + phoff))->off;
```

bootloader/start.S　　//跳转执行 bootMain 之前设置 esp

```
29    movl $0x1ffffff, %eax
30    movl %eax, %esp # TODO: setting esp
```

bootloader/Makefile　　//修改 makefile 以解决 boot block too large 问题

```
33    #objcopy -O binary bootloader.elf bootloader.bin
34    objcopy -S -j .text -O binary bootloader.elf bootloader.bin
```

kernel/kernel/doIrq.S　　//添加 irqKeyboard 中断向量号的处理

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

kernel/kernel/idt.c　　//初始化中断门和陷阱门

```
10 /* 初始化一个中断门(interrupt gate) */
11 static void setIntr(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset,
12    // TODO: 初始化interrupt gate
13    ptr->offset_15_0 = offset & 0xffff;
14    ptr->segment = selector << 3;
15    ptr->pad0 = 0;
16    ptr->type = INTERRUPT_GATE_32;
17    ptr->system = 0;
18    ptr->privilege_level = dpl;
19    ptr->present = 1;
20    ptr->offset_31_16 = (offset >> 16) & 0xffff;
```

```
23 /* 初始化一个陷阱门(trap gate) */
24 static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset,
25    // TODO: 初始化trap gate
26    ptr->offset_15_0 = offset & 0xffff;
27    ptr->segment = selector << 3;
28    ptr->pad0 = 0;
29    ptr->type = TRAP_GATE_32;
30    ptr->system = 0;
31    ptr->privilege_level = dpl;
32    ptr->present = 1;
33    ptr->offset_31_16 = (offset >> 16) & 0xffff;
```

//填写陷阱和中断表项，其中除了系统调用是用户态，其他均为内核态

```
64    // TODO: 填好剩下的表项
65    setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, DPL_KERN);
66    setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent, DPL_KERN);
67    setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault, DPL_KERN);
68    setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
69    setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
70    setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck, DPL_KERN);
71    setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException, DPL_KERN);
72    /* Exceptions with DPL = 3 */
73    // TODO: 填好剩下的表项
74    setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
75    setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
```

kernel/kernel/kvm.c　　//仿照内核加载方式加载用户程序

//其中只需修改 elf 首地址为 0x200000，以及扇区的加载从 201 而非 1 号开始

```
    int i = 0;
    int phoff = 0x34;
    int offset = 0x1000;
    unsigned int elf = 0x200000;
    unsigned int uMainEntry = 0x200000;

    for (i = 0; i < 200; i++) {
        readSect((void*)(elf + i*512), 201+i);
    }

    uMainEntry = ((struct ELFHeader *)elf)->entry;
    phoff = ((struct ELFHeader *)elf)->phoff;
    offset = ((struct ProgramHeader *)(elf + phoff))->off;

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

//以上代码完成了内核和用户代码的装载，实现了中断和陷阱门的初始化
//中断表的加载和调用，一下代码将实现 printf、getChar 和 getStr 这些函数

kernel/kernel/irqHandle.c　//中断处理程序调用

```
    switch(tf->irq) {
        // TODO: 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);
```

//KeyboardHandle 对退格符的处理：若当前列号大于 0
//则列号递减，输出空白符覆盖，清空 keyBuffer 尾部数据并修改尾指针

```
    if(code == 0xe){ // 退格符
        // TODO: 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--;
            char blank = ' ';
            keyBuffer[--bufferTail] = 0 | (0x0c << 8);
            uint16_t data = blank | (0x0c << 8);
            int pos = (displayRow * 80 + displayCol) * 2;
            asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));
```

//KeyboardHandle 对换行符的处理：行号递增，列数置 0，'\n'存入 buffer
//若当前行数大于 25 则调用滚屏函数处理，在控制台也输出换行符

```
    }else if(code == 0x1c){ // 回车符
        // TODO: 处理回车情况
        keyBuffer[bufferTail] = '\n';
        bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
        displayRow++;
        displayCol = 0;
        if (displayRow >= 25) {
            scrollScreen();
            displayRow = 24;
        }
        putChar('\n');
```

//KeyboardHandle 对正常字符的处理：不可打印字符按 0 处理
//大小写的实现键盘模块已提供，此处只需将对应 ascii 码存入 buffer 即可
//列号递增，若光标移至行尾（行号 80）则换行，仿照换行符处理即可

```
79      }else if(code < 0x81){ // 正常字符
80          // TODO: 注意输入的大小写的实现、不可打印字符的处理
81          char c = getChar(code);
82          if (c != 0) {
83              putChar(c);
84              keyBuffer[bufferTail] = c;
85              bufferTail = (bufferTail + 1) % MAX_KEYBUFFER_SIZE;
86              //putChar((char)(bufferTail + 48));
87              uint16_t data = c | (0x0c << 8);
88              int pos = (displayRow * 80 + displayCol) * 2;
89              asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));
90              displayCol++;
91              if (displayCol >= 80) {
92                  displayRow++;
93                  displayCol = 0;
94                  if (displayRow >= 25) {
95                      scrollScreen();
96                      displayRow = 24;
```

//syscallPrint 需根据参数从用户栈中读出字符数据，这部分框架代码已提供
//显示的实现原理、对换行和滚屏的处理与 KeyboardHandle 类似，不再赘述

lib/syscal.c　//printf 函数中格式化输出的实现
//对十进制数、十六进制数、字符串和单个字符分别进行了实现
//具体为调用提供的转换函数，将转换得到的字符存入 buffer
//需要注意的是此处单个字符也按四字节对其存储，若按 1 字节会出现问题

```
92          // TODO: in lab2
93          if (format[i] == '%') {
94              count--;
95              i++;
96              switch (format[i]) {
97              case '%':
98                  buffer[count] = '%';
99                  count++;
100                 break;
101             case 'd':
102                 index += 4;
103                 decimal = *(int *)(paraList + index);
104                 count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
105                 break;
106             case 'x':
107                 index += 4;
108                 hexadecimal = *(uint32_t *)(paraList + index);
109                 count = hex2Str(hexadecimal, buffer, MAX_BUFFER_SIZE, count);
110                 break;
111             case 's':
112                 index += 4;
113                 string = *(char **)(paraList + index);
114                 count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
115                 break;
116             case 'c':
117                 index += 4;
118                 character = *(char *)(paraList + index);
119                 buffer[count] = character;
120                 count++;
121                 break;
```

//sycallGetChar 的实现：查阅资料可知 getChar 的实现原理是有在用户按下换行符时
//取当前输入流中第一个字符，其余的字符（如果有的话）仍然留在输入流中
//因此当用户没有输入（buffer 头尾指针重合）或没有按下换行符时
//程序反复开中断来等待用户输入，输入完毕后，则取当前 buffer 的头部
//将返回值存入 eax，并将头指针向后移一位，此时若头尾指针没有重合
//说明输入流中还有数据，将尾指针向前移一位，避免将换行符也留在 buffer 内

```
180 void syscallGetChar(struct TrapFrame *tf){
181     // TODO: 自由实现
182     char character = 0;
183     while(bufferHead == bufferTail || keyBuffer[bufferTail - 1] != '\n') {
184         //putChar((char)(bufferTail + 48));
185         asm volatile("sti");
186     }
187     //putChar('F');
188     character = keyBuffer[bufferHead];
189     uint32_t data = character | (0x00000c << 8);
190     asm volatile("movl %0, %%eax"::"m"(data));
191     bufferHead++;
192     if (bufferTail != bufferHead)
193         bufferTail--;
194     tf->eax = data;
```

//在 lib/syscall/c/getChar()中调用 syscall，间接调用上面的函数
//由于系统函数返回值存储在 eax 中，此处用汇编指令将其取出

```
58 char getChar(){ // 对应SYS_READ STD_IN
59     // TODO: 实现getChar函数，方式不限
60     char c = 0;
61     uint32_t data = 0;
62     syscall(SYS_READ, STD_IN, 0, 0, 0, 0);
63     asm volatile("movl %%eax, %0":"=m"(data));
64     c = data;
65     return c;
```

//对于取输入流中第一个字符，其余留在缓冲区中的功能作如下测试
//首先输入 222，按下换行符后，再输入 Bob，按下换行符
//此处 i 为打字时的手误，同时再次验证了退格键功能
//可见第一个 2 被 getChar 返回了，后面两个则与 Bob 一同被 getStr 捕获

```
icspa@icspa:~/Desktop/lab2$ make play
------make------
1618106473393
Sun 11 Apr 2021 10:01:13 AM CST
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
222
Biob
```

```
Now I will test your getChar: 1 + 1 = 222
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
22Bob is stronger than Alice
==========================================================
Test end!!! Good luck!!!
```

//syscallGetStr 的实现：仿照 syscallPrint 对参数进行处理，获得字符串地址和长度
//当没有输入或没有换行时开中断等待用户，原理与 syscallGetCar 相同
//完毕后从输入流中循环取出 size-1 个字符存入用户栈（最后一位必须预留给'\0'）
//如果未取满 size-1 个字符便出现头尾指针相遇，则提前跳出循环
//如果取满 size-1 个字符后输入流中还有数据，则将这部分数据丢弃
//完成后将头尾指针置于一处，并将'\0'存入字符串末端位置

```
197 void syscallGetStr(struct TrapFrame *tf){
198     //putChar('-');
199     // TODO: 自由实现
200     char *str = (char*)tf->edx;
201     int size = tf->ebx;
202     int i = 0;
203     char character = 0;
204     //putChar(size + 48);
205     while(bufferHead == bufferTail || keyBuffer[bufferTail - 1] != '\n') {
206         asm volatile("sti");
207     }
208     while (i < size - 1) {
209         if (bufferHead == bufferTail)
210             break;
211         else {
212             character = keyBuffer[bufferHead];
213             bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
214             if (character != 0 && character != '\n') {
215                 //putChar(character);
216                 asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str + i));
217                 i++;
218             }
219         }
220     }
221     bufferHead = bufferTail;
222     asm volatile("movb $0x00, %%es:(%0)"::"r"(str + i));
```

//在 lib/syscall/c/getStr()中调用 syscall，间接调用上面的函数
//系统函数将字符串存入对应的地址，故此处无需其他操作

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

//针对 getStr 返回字符串长度进行了如下两次测试
//第一次输入了 30 个字符，输出长度为 19，另有一个隐式存在的'\0'

```
Now I will test your getStr: Alice is stronger than 123456789012345678901234567890
12345678901234567890 is stronger than Alice
============================================================
Test end!!! Good luck!!!
```

//第二次先输入 123，换行，在输入 20 个字符，可见输出层的仍为 19
//其中前两个字符为 getChar 遗留在缓冲区的，后 17 个为 getStr 输入的

```
Now I will test your getChar: 1 + 1 = 123
1 * 123 = 246
Now I will test your getStr: Alice is stronger than 12345678901234567890
2312345678901234567 is stronger than Alice
============================================================
Test end!!! Good luck!!!
```

思考和总结:

本次实验帮助我更好地理解了 ICS 中陷阱、中断和函数调用的介绍。其中不同部分（如内核框架、用户程序、装载程序、函数库都分别放在不同的子目录下，不同模块之间的结构和调用关系非常清晰，对我理解操作系统各模块的划分和相互配合有很大的帮助。

在实验过程中，我主要遇到了两次困难。第一次是实现 printf 并测试成功后，对 getChar 和 getStr 的“自主实现，方式不限”感到一筹莫展。首先我观察了 printf 的代码，认为 getStr 类似简化版的（不带格式输入的）scanf，可以仿照实现。可是 getStr 涉及到与用户交互、对键盘模块的调用和对 buffer 数组的读取，较 printf 有更多需要考虑的地方。

在 syscall 系列函数中，该如何隐式地调用 KeyboardHandle 模块呢？测试很容易得出，显示地调用该模块会造成大量重复乱码的输出。观察发现在用户程序装载之前（在写完这部分代码之前测试 printf）或者之后（所有测试用例运行完）的 while(1)循环中，KeyboardHandle 均可用于从键盘获取输入，并在控制台和屏幕输出。故 KeyboardHandle 应该不需要显式调用，而是在没有其他中断的情况下受键盘控制自动调用的。因此设置 while 循环，在获得满足条件的输入（换行符）前反复调用内联函数开中断，每次调用 sti 都开启下一条指令的硬件中断，KeyboardHandle 每次处理一个按键输入，则重复和乱码的问题得到了解决。

syscallGetStr 可以仿照 syscallPrint，根据输入的参数（字符串地址）将获得的字符依次写入内存中，作为 void 函数，它不需要考虑返回值的处理。可是 getChar 没有任何参数，却有返回值 char，它对应的系统函数也是 void 类型。我们知道 getChar 首先会调用 syscall，然后到 doIrq 里寻找对应的 irqHandle，由 irqHandle 调用 syscallHandle，调用 sysRead，最后调用 syscallGetChar，那么此时返回值该如何经由如此多的步骤传递到 getChar 中呢？推测由于途径的函数都是 void 类似，应该对 eax 没有改动，或改动又复原，可以在系统函数中通过对 tf 的 eax 进行修改传递返回值，再到 getChar 中调用内联汇编指令取出。

此外内联汇编指令的编写也让我有些困惑，如各种 mov 指令使用的是变量名、寄存器还是地址指针，其中使用的参数，限定的操作数类型都有什么规范。我在网上阅读了一些介绍性的文章，感觉说得都很笼统，例子很少，自己仿照框架代码中的去写，遇见报错或者结果与预期不符的情况再取修改，也能实现想要的功能，但总有一种不太满意的感觉。希望在本学期的其他实验和以后其他科目的学习中，我能够比较熟练地了解和使用内联汇编。

另外，官方版本的 getChar 似乎会把换行符也留在输入流中，但如果这样的话，根据提供的测试用例，测试 getChar 以后继续测试 getStr，此时会直接读到之前调用 getChar 遗留的换行符，而不会给用户额外输入任何内容的机会，为了折衷，这个版本的 getChar 保留了输入流缓冲区的特性，但把换行符排除在外，从而能够通过提供的测试代码。