Lab2 实验报告

231880101 孙俊晖

邮箱：2033282932@qq.com

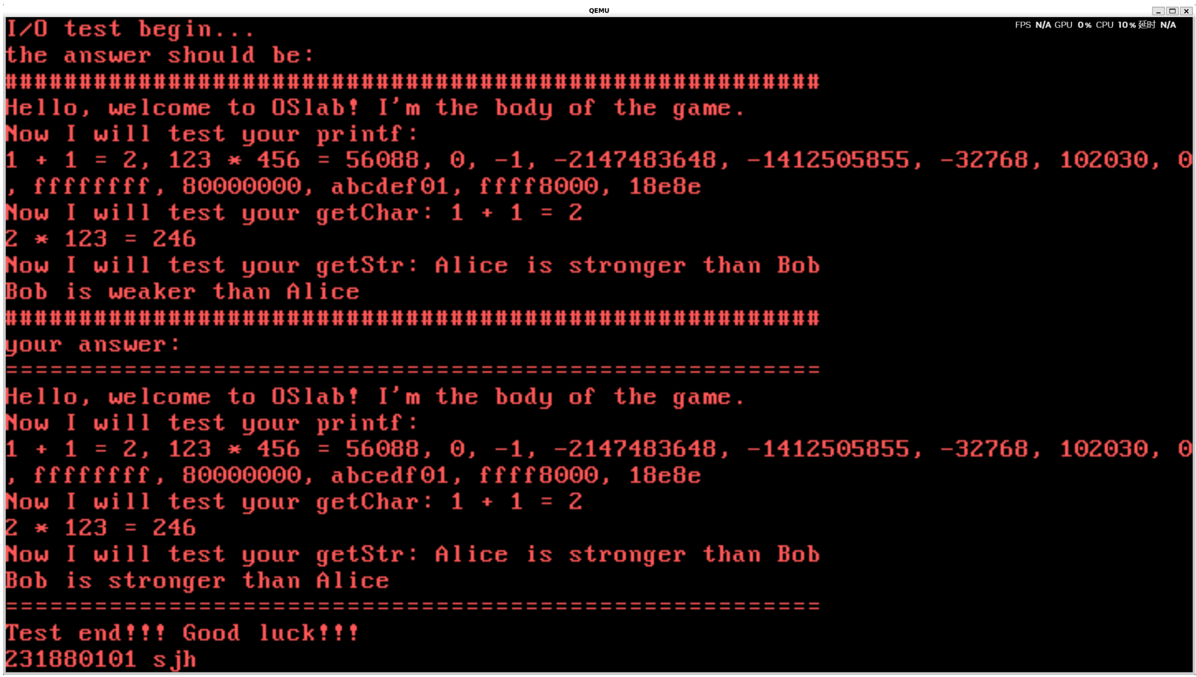
一、实验完成情况

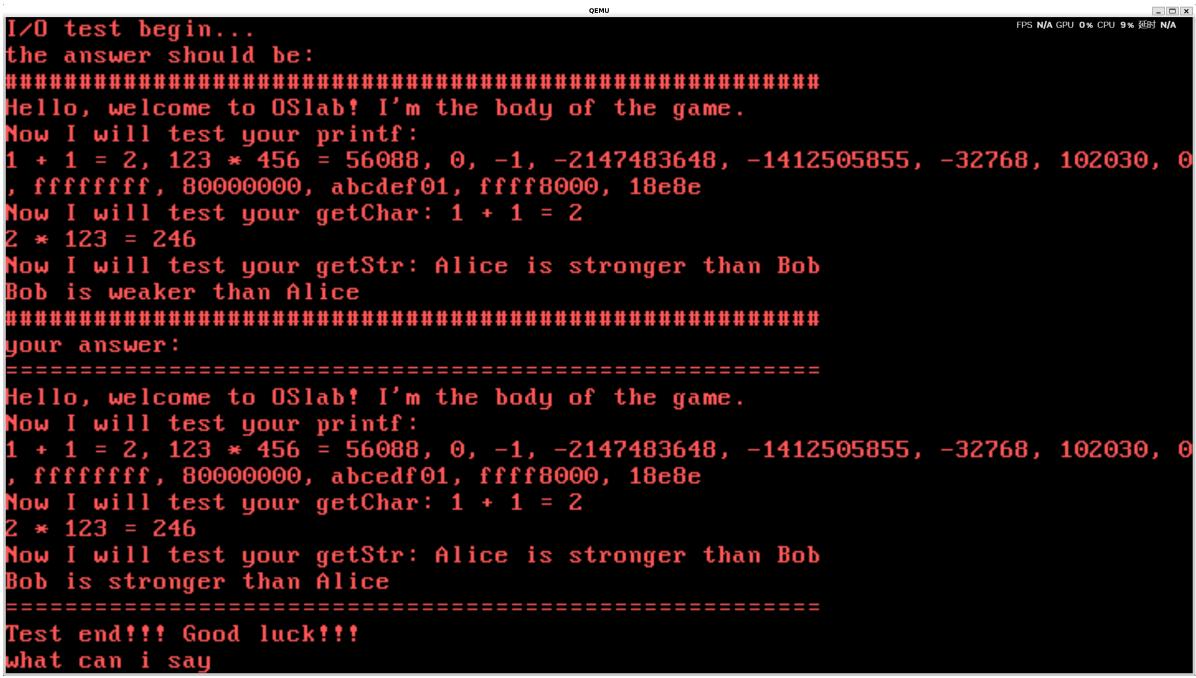
1.实验进度

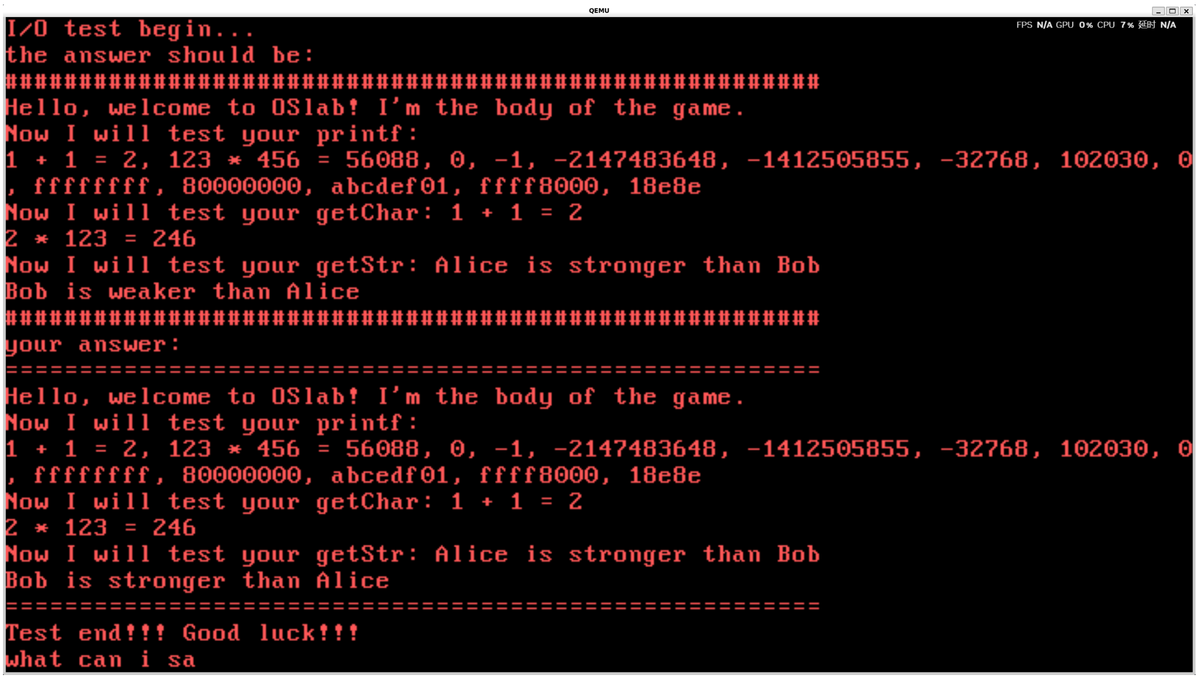
我完成了所有内容。

2.实验结果

我通过了用户程序主函数中的所有样例，实现了正常输入/输出字符，以及退格和换行功能。







3.实验修改的代码位置

（1）键盘按键的串口回显

1. 设置门描述符&各部分的初始化

在lab2/kernel/kernel/idt.c中，我首先完成了初始化中断门和陷阱门的函数setIntr和setTrap。

```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 = 0;
  ptr->privilege_level = dpl;
  ptr->present = 1;
  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 = 0;
  ptr->privilege_level = dpl;
  ptr->present = 1;
  ptr->offset_31_16 = (offset >> 16) & 0xffff;
}
```

然后我完成了initIdt函数，初始化 IDT 表的同时为中断设置中断处理函数。

```
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);
    }
    setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
    // TODO: 填好剩下的表项
    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);
    /* Exceptions with DPL = 3 */
    // TODO: 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);
    /* 写入IDT */
    saveIdt(idt, sizeof(idt));
}
```

然后我顺便完成了lab2/kernel/main.c中的内核main函数，这部分很简单，只要加入各个初始化函数即可。

```
void kEntry(void) {
    initSerial();// initialize serial port
    // TODO: 做一系列初始化
    initIdt(); // initialize idt
    initIntr(); // iniialize 8259a
    initSeg();  // initialize gdt, tss
    initVga(); // initialize vga device
    initKeyTable(); // initialize keyboard device
    loadUMain(); // load user program, enter user space

    while(1);
    assert(0);
}
```

在lab2/kernel/kernel/kvm.c中，我参照bootloader加载内核的方式完成了加载用户程序的函数，注意与boot.c中不同，加载用户程序时应从磁盘的第201个区域开始读取程序。

```
void loadUMain(void) {
  int i = 0;
  int offset = 0x1000;
  unsigned int elf = 0x200000;
  uint32_t uMainEntry = 0x200000;
  for (i = 0; i < 200; i++) {
      readSect((void*)(elf + i*512), 201 + i);
  }
  struct ELFHeader *ehdr = (void *)elf;
  uMainEntry = (uint32_t)(ehdr->entry);
  for (i = 0; i < 200 * 512; i++) {
```

```
            *(uint8_t *)(elf + i) = *(uint8_t *)(elf + i + offset);
        }
        enterUserSpace(uMainEntry);
}
```

已知键盘中断号为0x21，在lab2/kernel/kernel/doIrq.s中，我将irqKeyboard的中断向量号0x21压入栈。

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

在lab2/bootloader/start.s中设置esp寄存器的值，这部分和lab1中一模一样，入口地址为0x8000。

```
.code32
start32:
    movw $0x10, %ax # setting data segment selector
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %fs
    movw %ax, %ss
    movw $0x18, %ax # setting graphics data segment selector
    movw %ax, %gs

    movl $0x8000, %eax # TODO: setting esp
  movl %eax, %esp
    jmp bootMain # jump to bootMain in boot.c
```

2. 完善中断服务例程

根据各中断给出的中断号, 在lab2/kernel/kernel/irqHandle.c的irqHandle函数中, 根据不同的中断填充其对应调用的处理函数。

```
void irqHandle(struct TrapFrame *tf) { // pointer tf = esp
    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;
    case 0xd:
      GProtectFaultHandle(tf);
      break;
    case 0x21:
      KeyboardHandle(tf);
      break;
    case 0x80:
      syscallHandle(tf);
      break;
        default:
      assert(0);
    }
}
```

GProtectFaultHandle函数和syscallHandle函数已经写好了，只需完成KeyboardHandle函数即可。根据输入字符的种类（退格符/回车符/正常字符）分情况进行处理，在这个过程中得注意维护光标的位置。当输入字符为可打印字符时，直接将其打印到显存中。

```
void KeyboardHandle(struct TrapFrame *tf){
    uint32_t code = getKeyCode();
    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(bufferTail > bufferHead && keyBuffer[bufferTail] != '\n')
    {
      int currentHead = bufferTail - 1;
            while (currentHead > bufferHead && keyBuffer[currentHead - 1] != '\n')
      {
        currentHead--;
      }
            if(displayCol > 0)
      {
        displayCol--;
      }
            keyBuffer[bufferTail] = '\0';
      bufferTail--;
        int sel = USEL(SEG_UDATA);
        char character = 0;
        uint16_t data = 0;
        int pos = 0;
        asm volatile("movw %0, %%es"::"m"(sel));
        for (int i = 0; i < bufferTail - currentHead; i++) {
            asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(keyBuffer+currentHead+i));
            data = character | (0x0c << 8);
            pos = (80 * displayRow + displayCol) * 2;
            asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));
        }
    }
    }else if(code == 0x1c){ // 回车符
        // TODO: 处理回车情况
    displayCol = 0;
        displayRow ++;
        keyBuffer[bufferTail] = '\n';
    bufferTail++;
    }else if(code < 0x81){ // 正常字符
        // TODO: 注意输入的大小写的实现、不可打印字符的处理
    char ch = getChar(code);
        if (ch >= 0x20) {
            putChar(ch);
            keyBuffer[bufferTail] = ch;
      bufferTail++;
            int sel = USEL(SEG_UDATA);
            char character = ch;
            uint16_t data = 0;
            int pos = 0;
            asm volatile("movw %0, %%es"::"m"(sel));
            data = character | (0x0c << 8);
            pos = (80 * displayRow + displayCol) * 2;
```

```
            asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));
            displayCol ++;
            if (displayCol >= 80) {
                displayCol = 0;
                displayRow ++;
            }
            while (displayRow >= 25) {
                scrollScreen();
                displayRow --;
                displayCol = 0;
            }
        }
    }
    updateCursor(displayRow, displayCol);
}
```

（2）实现printf的处理例程

我在lab2/kernel/kernel/irqHandle.c中完成了与写显存内容密切相关的函数syscallPrint，根据实验手册维护段选择子，并实现了光标的维护和打印到显存，同时要考虑换行和翻页问题。

```
void syscallPrint(struct TrapFrame *tf) {
    int sel = USEL(SEG_UDATA); //TODO: segment selector for user data, need further modification
    char *str = (char*)tf->edx;
    int size = tf->ebx;
    int i = 0;
    int pos = 0;
    char character = 0;
    uint16_t data = 0;
    asm volatile("movw %0, %%es"::"m"(sel));
    for (i = 0; i < size; i++) {
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
        // TODO: 完成光标的维护和打印到显存
        if(character == '\n')
        {
            displayCol = 0;
            displayRow++;
            if (displayRow >= 25) {
                scrollScreen();
                displayRow = 24;
                displayCol = 0;
            }
        }
        else
        {
            data = character | (0x0c << 8);
            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) {
                scrollScreen();
                displayRow = 24;
                displayCol = 0;
            }
        }
    }
    updateCursor(displayRow, displayCol);
}
```

(3) 完善printf的格式化输出

我在lab2/lib/syscall.c中完成了printf函数，支持%d, %x, %s,%c四种格式转换说明符。该部分需要用到文件中已封装好的转换函数, 根据不同的格式转换说明符进行相应处理即可。

```
void printf(const char *format,...){
    int i=0; // format index
    char buffer[MAX_BUFFER_SIZE];
    int count=0; // buffer index
    //int index=0; // parameter index
    void *paraList=(void*)&format+sizeof(uint32_t); // 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;
    while(format[i]!=0){
        char c = format[i++];
        // TODO: in lab2
    if(state == 0)
    {
      if(c == '%')
      {
        state = 1;
      }
      else
      {
        buffer[count] = c;
        count++;
      }
      continue;
    }
    else if(state == 1)
    {
      if(c == 'd')
      {
        decimal = *(int *)paraList;
                paraList += 4;
                count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
      }
      else if(c == 'x')
      {
        hexadecimal = *(uint32_t *)paraList;
                paraList += 4;
```

```
                count = hex2Str(hexadecimal, buffer, MAX_BUFFER_SIZE, count);
      }
      else if(c == 'c')
      {
        character = *(char *)paraList;
                paraList += 4;
                buffer[count++] = character;
                if (count == MAX_BUFFER_SIZE) {
                    syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
                    count = 0;
                }
      }
      else if(c == 's')
      {
        string = *(char **)paraList;
                paraList += 4;
                count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
      }
      else
      {
        state = 2;
        return;
      }
      state = 0;
    }
    else if(state == 2)
    {
      return;
    }
    }
    if(count!=0)
        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
}
```

（4）实现getChar, getStr的处理例程

参照printf函数的实现过程，我首先在lab2/kernel/kernel/irqHandle.c中实现了getChar和getStr的系统调用函数syscallGetChar和syscallGetStr，同样使用keyBuffer数组来辅助相应功能的实现。

```
void syscallGetChar(struct TrapFrame *tf){
    // TODO: 自由实现
  int flag = 0;
  if(keyBuffer[bufferTail - 1] == '\n')
  {
    flag = 1;
  }
    while (bufferTail > bufferHead && keyBuffer[bufferTail-1] == '\n')
  {
    bufferTail--;
    keyBuffer[bufferTail] = '\0';
  }
    if (bufferTail > bufferHead && flag == 1)
```

```
  {
    tf->eax = keyBuffer[bufferHead];
    bufferHead++;
  }
    else
  {
    tf->eax = 0;
  }
}

void syscallGetStr(struct TrapFrame *tf){
    // TODO: 自由实现
  int flag = 0;
    int sel = USEL(SEG_UDATA);
    asm volatile("movw %0, %%es"::"m"(sel));
    if (keyBuffer[bufferTail - 1] == '\n')
  {
    flag = 1;
  }
    while (bufferTail > bufferHead && keyBuffer[bufferTail-1] == '\n')
  {
    bufferTail--;
    keyBuffer[bufferTail] = '\0';
  }
    if (flag == 0 && bufferTail - bufferHead < tf->ebx)
  {
    tf->eax = 0;
  }
    else
  {
    //参考syscall.c中的dec2Str函数
        char str[256];
      int count = 0, i = 0;
    int decimal = bufferTail - bufferHead;
    int temp;
    int number[16];
    // 处理负数
    if (decimal < 0) {
      str[count++] = '-';
      if (count == 256)
      {
        putChar('\n');
        count = 0;
      }
      temp = decimal / 10;
      number[i++] = temp * 10 - decimal;
      decimal = temp;
      while (decimal != 0)
      {
        temp = decimal / 10;
        number[i++] = temp * 10 - decimal;
        decimal = temp;
      }
    }
    else
    {
```

```
      temp = decimal / 10;
      number[i++] = decimal - temp * 10;
      decimal = temp;
      while (decimal != 0)
      {
        temp = decimal / 10;
        number[i++] = decimal - temp * 10;
        decimal = temp;
      }
    }
    // 将数字转换为字符串
    while (i != 0)
    {
      str[count++] = number[i - 1] + '0';
      if (count == 256)
      {
        putChar('\n');
        count = 0;
      }
      i--;
    }
    str[count] = '\0';
    // 输出字符串长度
    while (str[i] != '\0') putChar(str[i++]);
    putChar('\n');
        for (int i = 0; i < tf->ebx && i < bufferTail-bufferHead; i++) {
            asm volatile("movb %1, %%es:(%0)"::"r"(tf->edx+i), "r"
(keyBuffer[bufferHead+i]));
        }
        tf->eax = 1;
    }
}
```

最后在lab2/lib/syscall.c中实现了getChar和getStr函数，这两个函数通过判断返回值是否为0来进行返回。

```
char getChar(){ // 对应SYS_READ STD_IN
    // TODO: 实现getChar函数，方式不限
  char c = 0;
    while (c == 0)
  {
    c = (char)syscall(SYS_READ, STD_IN, 0, 0, 0, 0);
  }
    return c;
}

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

```
}
```

二、问题回答

1.ring3的堆栈在哪里?

ring3的堆栈信息由用户程序自行管理，通常保存在用户程序的堆栈段中，其堆栈指针（ESP）和段寄存器（SS）在用户程序运行时通过代码设置。

2.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?

TSS中仅存储更高特权级的堆栈信息，用于特权级提升时的堆栈切换。在IA-32体系架构中，ring3是用户态，是最低特权级，任何一个向高特权级进行的转移都不可能转移到ring3。因此，不需要在TSS中保存ring3的堆栈信息。

3.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?

会产生不可恢复的错误。因为内核在处理系统调用或中断时，可能会修改这些通用寄存器的值。如果不保存它们的原始值，这些修改将直接影响到用户态程序的寄存器状态。用户态程序依赖于这些寄存器的值来继续执行。如果这些值被内核意外修改，用户态程序的行为将变得不可预测，可能导致程序崩溃、数据错误或其他异常行为。

三、实验心得

本次实验让我深入理解了计算机系统从输入到输出的底层机制，尤其是中断机制和系统调用的实现原理。通过手动实现库函数printf、getChar、getStr，我不仅对这些常用函数有了更直观的认识，还体会到从理论到实践的挑战。在实验过程中，我遇到了各种各样的问题，一开始make时不同文件的代码老是会频繁报错，经常改完还报错，导致我得频繁地make/make clean，甚至第一次弹出窗口时由于IDT表初始化错误，窗口一直在闪。最终，当实验成功运行，屏幕上显示出预期的输出时，我感受到了前所未有的成就感。