

# 《操作系统》实验二报告

---

201220069 周心同

---

## 一、实验目的

---

在第一次实验中，我们熟悉了操作系统启动的过程，本次实验将继续完善我们的miniOS。我们知道，大部分的计算任务的过程：**接受输入->处理->输出**。要让他能够正常工作，我们需要使OS具备基本的**输入输出能力**。

我们在ICS课上学过，用户程序进行一些IO操作是通过一些系统级API，比如write和read函数。本次实验，我们会实现这些API然后赋予它IO能力。

## 二、实验内容

---

- **本次实验共有如下任务点需要完成（编号都是从1开始计数）：**

- 23个exercise
- 6个task
- 4个challenge
- 3个conclusion

- **完成内容：**

- 23个exercise（标蓝）
- 6个task（见代码）
- challenge1, 2（部分），3（部分）（标黄）
- 3个conclusion（标红）

## 三、实验过程和结果（按照手册顺序）

---

**exercise1：**既然说确定磁头更快（电信号），那么为什么不把连续的信息存在同一柱面号同一扇区号的连续的盘面上呢？（Hint：别忘了在读取的过程中盘面是转动的）

如果放在同一柱面号同一扇区号的连续的盘面，那么当一个磁头读完时，此时位置已经在该扇区的末尾了，要继续读就只能先回到最开始的位置，这段时间是浪费了的。想要一次性读完，除非有几个磁头同时读，这显然是做不到的。

而如果以柱面为单位连续放在一个个柱面上，一个磁头可以连续地读完一个柱面，且当该柱面读完时，下一个磁头可以无间隔地读下一个柱面的信息，没有浪费时间。

**exercise2: 假设CHS表示法中柱面号是C，磁头号是H，扇区号是S；那么请求一下对应LBA表示法的编号（块地址）是多少（注意：柱面号，磁头号都是从0开始的，扇区号是从1开始的）。**

假设柱面一共有c种，磁头有h种，扇区有s种

$$LBA = \text{扇区数} \times \text{磁头数} \times \text{柱面号} + \text{扇区数} \times \text{磁头号} + \text{当前所在扇区号} - 1 = shC + s * H + S - 1$$

**exercise3: 请自行查阅读取程序头表的指令，然后自行找一个ELF可执行文件，读出程序头表并进行适当解释（简单说明即可）。**

将hello.cc链接生成elf文件，用 `readelf -l hello` 查看程序头表

```
oslab@oslab-VirtualBox:~/OS2022$ readelf -l hello
```

Elf 文件类型为 DYN（共享目标文件）

Entry point 0x7b0

There are 9 program headers, starting at offset 64

程序头:

| Type        | offset             | VirtAddr             |
|-------------|--------------------|----------------------|
| PhysAddr    |                    |                      |
|             | FileSiz            | MemSiz               |
| Flags Align |                    |                      |
| PHDR        | 0x0000000000000040 | 0x0000000000000040   |
|             | 0x0000000000000040 |                      |
|             | 0x00000000000001f8 | 0x00000000000001f8 R |
| 0x8         |                    |                      |
| INTERP      | 0x0000000000000238 | 0x0000000000000238   |
|             | 0x0000000000000238 |                      |

```

                                0x0000000000000001c 0x0000000000000001c  R
0x1
    [Requesting program interpreter: /lib64/ld-linux-
x86-64.so.2]
    LOAD                        0x0000000000000000 0x0000000000000000
0x0000000000000000
                                0x00000000000000b78 0x00000000000000b78  R
E    0x200000
    LOAD                        0x00000000000000d78 0x000000000000200d78
0x000000000000200d78
                                0x00000000000000298 0x000000000000003c0  RW
    0x200000
    DYNAMIC                     0x00000000000000d90 0x000000000000200d90
0x000000000000200d90
                                0x00000000000000200 0x00000000000000200  RW
    0x8
    NOTE                        0x00000000000000254 0x00000000000000254
0x00000000000000254
                                0x00000000000000044 0x00000000000000044  R
    0x4
    GNU_EH_FRAME                0x000000000000009e4 0x000000000000009e4
0x000000000000009e4
                                0x0000000000000004c 0x0000000000000004c  R
    0x4
    GNU_STACK                    0x00000000000000000 0x00000000000000000
0x00000000000000000
                                0x00000000000000000 0x00000000000000000  RW
    0x10
    GNU_RELRO                    0x00000000000000d78 0x000000000000200d78
0x000000000000200d78
                                0x00000000000000288 0x00000000000000288  R
    0x1

```

#### Section to Segment mapping:

段节...

00

01 .interp

02 .interp .note.ABI-tag .note.gnu.build-id

.gnu.hash .dynsym .dynstr .gnu.version .gnu.version\_r

.rela.dyn .rela.plt .init .plt .plt.got .text .fini

.rodata .eh\_frame\_hdr .eh\_frame

```

03      .init_array .fini_array .dynamic .got .data .bss
04      .dynamic
05      .note.ABI-tag .note.gnu.build-id
06      .eh_frame_hdr
07
08      .init_array .fini_array .dynamic .got

```

由上图可以看出该程序头表列出了9个段以及各段的类型、相对于文件开头的偏移量、在虚拟地址空间和物理地址空间的位置、大小、标志和对齐信息。

PHDR保存程序头表。INTERP指定在程序已经从可执行映射到内存之后，必须调用解释器。LOAD表示一个从二进制文件映射到虚拟地址空间的段。其中保存了常量数据（如字符串），程序的目标代码等等。DYNAMIC段保存了其他动态链接器（即，INTERP中指定的解释器）使用的信息。NOTE保存了专有信息。

第二部分指定了哪些节载入到哪些段。

**exercise4: 上面的三个步骤都可以在框架代码里面找得到，请阅读框架代码，说说每步分别在哪个文件的什么部分（即这些函数在哪里）？**

1. （步骤一）先是加载OS部分（lab1，当然，这次要解析ELF格式）。
  1. ~~从实模式进入保护模式 (lab1)。~~ bootloader/start.S
  2. ~~加载内核到内存某地址并跳转运行 (lab1)。~~  
bootloader/boot.c/bootMain
2. （步骤二）然后是进行系统的各种初始化工作，这里我们采用模块化的方法。/kernel/main.c
  1. 初始化串口输出 (`initSerial`) kernel/kernel/serial.c/initSerial
  2. 初始化中断向量表 (`initIdt`) kernel/kernel/idt.c/initIdt
  3. 初始化8259a中断控制器 (`initIntr`)  
kernel/kernel/i8259.c/initIntr
  4. 初始化 GDT 表、配置 TSS 段 (`initSeg`)  
kernel/kernel/kvm.c/initSeg
  5. 初始化VGA设备 (`initVga`) kernel/kernel/vga.c/initVga
  6. 配置好键盘映射表 (`initKeyTable`)  
kernel/kernel/keyboard.c/initKeyTable

7. 从磁盘加载用户程序到内存相应地址 (`loadUMain`)

`kernel/kernel/kvm.c/loadUMain`

3. (步骤三) 最后进入用户空间进行输出。

1. 进入用户空间 (`enterUserSpace`)

`kernel/kernel/kvm.c/enterUserSpace`

2. 调用库函数, 输出各种内容! `app/main.c/uEntry` (库函数在 `lib/syscall.c`)

**exercise5: 是不是思路有点乱? 请梳理思路, 请说说“可屏蔽中断”, “不可屏蔽中断”, “软中断”, “外部中断”, “异常”这几个概念之间的区别和关系。(防止混淆)**

中断包括内中断和外中断, 外中断是指来自CPU外部的中断, 内中断是指在CPU内部的中断。

在内中断中分为两种, 通过int指令自愿产生的中断称为软中断, 通过地址越界, 虚地址缺页等引发的中断为强迫中断, 即异常。(手册上这样写, 网上说异常是内中断, 但是没有区分软中断和强迫中断。实际上, 不同计算机体系结构和教科书对异常和中断这两个概念规定了不同的内涵。)

不可屏蔽中断是CPU必须响应的中断, 包括外部 NMI 针脚产生的中断和内部软中断, 仅有几个特定的事件才能引起非屏蔽中断, 例如硬件故障以及或是掉电。

剩下的都是可屏蔽中断, 包括不在NMI线上的外部中断。

CPU 内部产生, 也可由外部 NMI 针脚产生。

**exercise6: 这里又出现一个概念性的问题, 如果你没有弄懂, 对上面的文字可能会有疑惑。请问: IRQ和中断号是一个东西吗? 它们分别是什么?(如果现在不懂可以做完实验再来回答。)**

IRQ对应的是8259芯片的一个中断引脚, 是硬件层面的, 只有硬件中断会用到, 8259通过被编程将引脚与对应的中断号一一映射。

中断号对应的是内存中断向量区中的中断类型, 是软件层面的, 中断号范围由硬件中断和软件中断所共享, CPU看到的只有中断号, 不清楚是硬件中断还是软件中断, 它所执行的是根据中断号到中断描述符表找到对应的中断向量然后调用中断处理程序。

**exercise7: 请问上面用斜体标出的“一些特殊的原因”是什么？**

**(Hint: 为啥不能用软件保存呢？注：这里的软件是指一些函数或者指令序列，不是gcc这种软件。)**

首先，软件本身是用指令一条条执行的，EIP和CS需要用来指示软件指令执行的位置，EFLAGS也会影响指令的执行。

而且如果用软件保存，可能软件自身的指令也会存在问题而引发中断，用硬件保存能确保进入中断处理程序并能正确返回原程序。

同时，用软件保存可能会涉及到安全方面的问题，比如软件可以篡改CS的CPL从而进入原本无权访问的代码区域中。

**exercise8: 请简单举个例子，什么情况下会被覆盖？（即稍微解释一下上面那句话）**

第一个中断发生时，若将状态信息保存到一个固定区域，在执行改中断服务时，若有另一个优先级更高的中断提出中断请求，此时会终止较低级别的中断服务程序，将优先级更高的中断状态信息保存到同一片固定区域中，那么原来的信息就被覆盖了。

**exercise9: 请解释我在伪代码里用“???”注释的那一部分，为什么要pop ESP和SS？**

比较两个CS段寄存器的DPL来切换堆栈防止越权访问，以这里的例子来说，若 $GDT[old\_CS].DPL < GDT[CS].DPL$ ，说明pop出来的CS是用户态的CS（而old\_CS显然是内核态的），也就是说之前是从用户态进入到了内核态，现在pop ESP和SS即返回用户态。

**exercise10: 我们在使用eax, ecx, edx, ebx, esi, edi前将寄存器的值保存到了栈中（注意第五行声明了6个局部变量），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？**

显然会。在syscall函数里先将各个参数分别赋值给EAX, EBX, ECX, EDX, EDI, ESI，此时会覆盖掉寄存器的原值，返回之后无法恢复。

**exercise11: 我们会发现软中断的过程和硬件中断的过程类似，他们最终都会去查找IDT，然后找到对应的中断处理程序。为什么会这样设计？（可以做实验再回答这个问题）**

CPU看到的只有中断号，中断号范围由硬件中断和软件中断所共享，CPU所需要执行的是根据中断号到中断描述符表找到对应的中断向量然后调用中断处理程序。

CPU不需要知道是硬件中断还是软中断，因为对于CPU的执行来说，中断的区别只在于是不可屏蔽还是可屏蔽的，这点通过IF标志位的设置即可知道。而硬件中断已被8259芯片映射到了对应的中断号上，这对CPU是透明的。

**exercise12: 为什么要设置esp? 不设置会有什么后果? 我是否可以把esp设置成别的呢? 请举一个例子, 并且解释一下。(你可以在写完实验确定正确之后, 把此处的esp设置成别的实验一下, 看看哪些可以哪些不可以)**

因为栈是从高地址向低地址增长的, 所以要先把栈的地址设置到一个高地址的地方, 否则后面没有足够的占空间。

可以设置成别的, 但是设置的越低, 所留的栈空间就越小, 而且可能会造成空间浪费。

比如设置成0x101000就会影响elf的装载的位置, 显然是不可以的。

**exercise13: 上面那样写为什么错了?**

因为上面加载ELF文件(即kernel)的方法是通过将从磁盘读进来的内容直接放到0x100000开始的地址上, 而没有按照elf文件读取的格式来做。

**challenge1: 既然错了, 为什么不影响实验代码运行呢?**

这个问题设置为challenge说明它比较难, 回答这个问题需要对ELF加载有一定掌握, 并且愿意动手去探索。Hint: 可以在写完所有内容并保证正确后, 改成这段错误代码, 进行探索, 并回答该问题。(做出来加分, 做不出来不扣分)



程序头:

| Type               | Offset   | VirtAddr   | PhysAddr   | FileSiz |
|--------------------|----------|------------|------------|---------|
| MemSiz Flg Align   |          |            |            |         |
| LOAD               | 0x001000 | 0x00100000 | 0x00100000 | 0x01568 |
| 0x01568 R E 0x1000 |          |            |            |         |
| LOAD               | 0x003000 | 0x00103000 | 0x00103000 | 0x00120 |
| 0x01f00 RW 0x1000  |          |            |            |         |
| GNU_STACK          | 0x000000 | 0x00000000 | 0x00000000 | 0x00000 |
| 0x00000 RWE 0x10   |          |            |            |         |

经过对kernel的ELF文件的程序头表的读取，可以发现前两个段是LOAD段，且第一个段就是从文件偏移1000的地方装到0x100000，第二个段就是从文件偏移3000装到0x103000，而错误的方法是将elf文件的内容移到了0x101000开始的地方，刚好装对了第一个LOAD段。第二个段是全局变量段，在kernel中几乎没用到，故不影响。第三个段是0字节，本就不需要装载。

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

**exercise14: 请查看Kernel的Makefile里面的链接指令，结合代码说明kMainEntry函数和Kernel的main.c里的kEntry有什么关系。kMainEntry函数究竟是啥？**

```
$(LD) $(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf  
$(KOBJS)
```

-e kEntry: 使用符号kENTRY作为你的程序的开始执行点。

-Ttext 0x00100000: 设置.text节的位置。

在这里kENTRY成为了文件的开始执行点，而kMainEntry函数调用后会跳转到0x00100000,即代码节的起始位置，这里执行的第一个函数即kENTRY。

**exercise15: 到这里，我们对中断处理程序是没什么概念的，所以请查看doirq.S，你就会发现idt.c里面的中断处理程序，请回答：所有的函数最后都会跳转到哪个函数？请思考一下，为什么要这样做呢？**



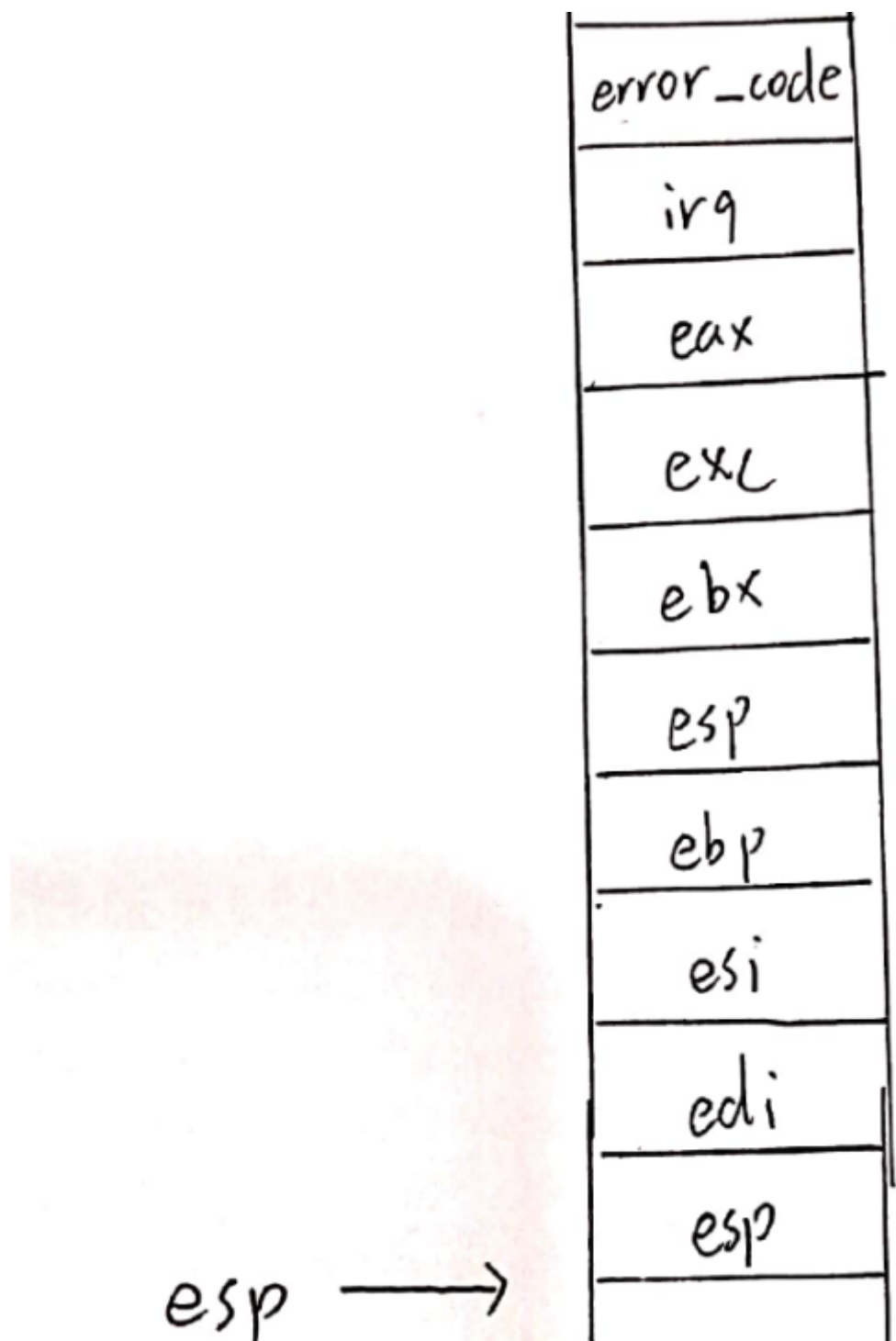
跳转到asmDoIrq。统一处理所有中断请求，每一次中断处理都只需根据传入的类型号和中断向量调用对应的中断处理程序，调用同一个函数可以减少代码冗余，减少出bug的可能性。

**exercise16: 请问doirq.S里面asmDoirq函数里面为什么要push esp? 这是在做什么? (注意在push esp之前还有个pusha, 在pusha之前.....)**

根据irq\_handle函数可以发现pushl %esp其实就是准备参数 TrapFrame \*tf。该参数是一个指向 TrapFrame 结构体的指针，通过这个指针来获取之前所有压栈的寄存器的值，之前所有压栈的值的的首地址就是当前的esp寄存器的值，所以只要将esp寄存器的值压栈， irq\_handle 函数就可以访问所有之前压栈的寄存器的值。

根据TrapFrame的结构以及之前的pusha等等操作，我们可以推断出栈如下：

```
struct TrapFrame {
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    int32_t irq;
};
```



**exercise17: 请说说如果keyboard中断出现嵌套, 会发生什么问题?**  
(Hint: 屏幕会显示出什么? 堆栈会怎么样?)

如果keyboard中断允许嵌套，那么在执行打印一个字母的时候，有可能因为有一个keyboard中断的中断请求而被打断，去执行第二个中断处理程序，打印另一个字母，这会导致打印字母的顺序相反，比如输入abc，打cba。由于堆栈需要保存现场信息，故而在处理完中断前会累积起来，如果长按可能会造成堆栈溢出。

### exercise18: 阅读代码后回答，用户程序在内存中占多少空间？

用户程序大小不大于 $200 \times 512$ 字节，即100KB。

### exercise19: `int sel = USEL(SEG_UDATA);`请说说sel变量有什么用。 (请自行搜索)

```
int sel = USEL(SEG_UDATA);

asm volatile("movw %0, %%es" :: "m"(sel));

asm volatile("movb %%es:(%1), %0"
              : "=r"(character)
              : "r"(str + i));
```

sel变量即用户的数据段选择子，后面内联汇编中，“m”表示使用内存方式输入或输出，结合前面movw指令，可知是把sel的值赋给es，再根据后一个内联汇编，可知赋值给es段寄存器说为了找到character的地址。因为原来传过来的 `char *str = (char *)tf->edx;` 并不足以找到其地址，还需要加上用户数据段的基地址才能找到。

### challenge2: 比较关键的几个函数

1. KeyboardHandle函数是处理键盘中断的函数
  2. syscallPrint函数是对应于“写”系统调用
  3. syscallGetChar和syscallGetStr对应于“read”系统调用
- 有以下两个问题

1. 请解释框架代码在干什么。
2. 阅读前面人写的烂代码是我们专业同学必备的技能。很多同学会觉得这三个函数给的框架代码写的非常烂，你是否有更好的实现方法？可以替换掉它（此题目难度很大，修改哪个都行）。这一问可以不做，但是如果有同学实现得好，可能会有隐藏奖励（也可能没有）。

KeyboardHandle 函数：

- 1.通过getKeyCode获取键盘输入的扫描码。
- 2.分别处理退格符，回车符和正常字符。
- 3.通过内存映射将要显示的字符写到vga显存上。

`syscallPrint` 函数：

- 1.通过用户段的段选择子和传进来的参数获取需要打印的内容。
- 2.将 `edx` 指向的 `ebx` 个字符输出到VGA显存。

`syscallGetChar` 函数：

- 1.初始化缓冲区。
- 2.重复读取缓冲区第一个字符，直到其不为空，则为读到。
- 3.后面wait再读一个。

`syscallGetStr` 函数：

- 1.初始化缓冲区。
- 2.用上述方式重复读取缓冲区的一个个字符，直到读到回车。
- 3.将取得的字符串（加上`\0`）放到对应的用户数据区中。

顺便，写的烂吗，比我数电实验写的好多了。。。。

**exercise20: `paraList`是`printf`的参数，为什么初始值设置为`&format`？ 假设我调用`printf("%d = %d + %d", 3, 2, 1);`，那么数字2的地址应该是多少？ 所以当我解析`format`遇到`%`的时候，需要怎么做？**

在调用`printf`时，`printf`的参数会从右往左依次入栈，将`paraList`初始值设置为`&format`，即`printf`的最左边参数的地址，这样就可以访问`printf`中的其他参数。

数字2的地址应该是`&format+2*sizeof(format)`。（栈区宽度为4，`sizeof(format)`即4）

每次遇到`%`号时，说明要使用下一个参数了，故将`paraList += sizeof(format)`即可读取对应参数的值。

**exercise21: 关于系统调用号, 我们在printf的实现里给出了样例, 请找到阅读这一行代码**

```
syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
```

**说一说这些参数都是什么 (比如SYS\_WRITE在什么时候会用到, 又是怎么传递到需要的地方呢?) 。**

在执行syscall后, 首先会保存寄存器旧值, 然后将参数存入对应的寄存器, 通过 `asm volatile("int $0x80");` 进入内核, 跳转到doIrq.S的irqSyscall, 这里会准备好irqHandle的参数TrapFrame, 所以这些参数都是TrapFrame的值, 然后irqHandle根据中断类型号跳转到syscallHandle。

SYS\_WRITE: 被存入了eax, 在syscallHandle中有判断, 如果是0 (即SYS\_WRITE), 调用syscallWrite, 如果是1(即SYS\_READ), 调用syscallread。

STD\_OUT: 被存入了ecx, 在syscallWrite中有判断, 如果是0 (即STD\_OUT), 调用syscallPrint

(uint32\_t)buffer: 被存入了edx, 在syscallPrint用 `char *str = (char*)tf->edx;` 读取了该缓冲区

(uint32\_t)count: 被存入了ebx, 在syscallPrint用 `int size = tf->ebx;` 读取了缓冲区字符计数

**exercise22: 记得前面关于串口输出的地方也有putChar和putStr吗? (助教打错了) 这里的两个函数和串口那里的两个函数有什么区别? 请详细描述它们分别在什么时候能用。**

这里的两个函数是用户程序通过系统调用进入内核态, 内核态再通过传进来的参数来进行, 故只能在用户程序里使用 (即app) 。

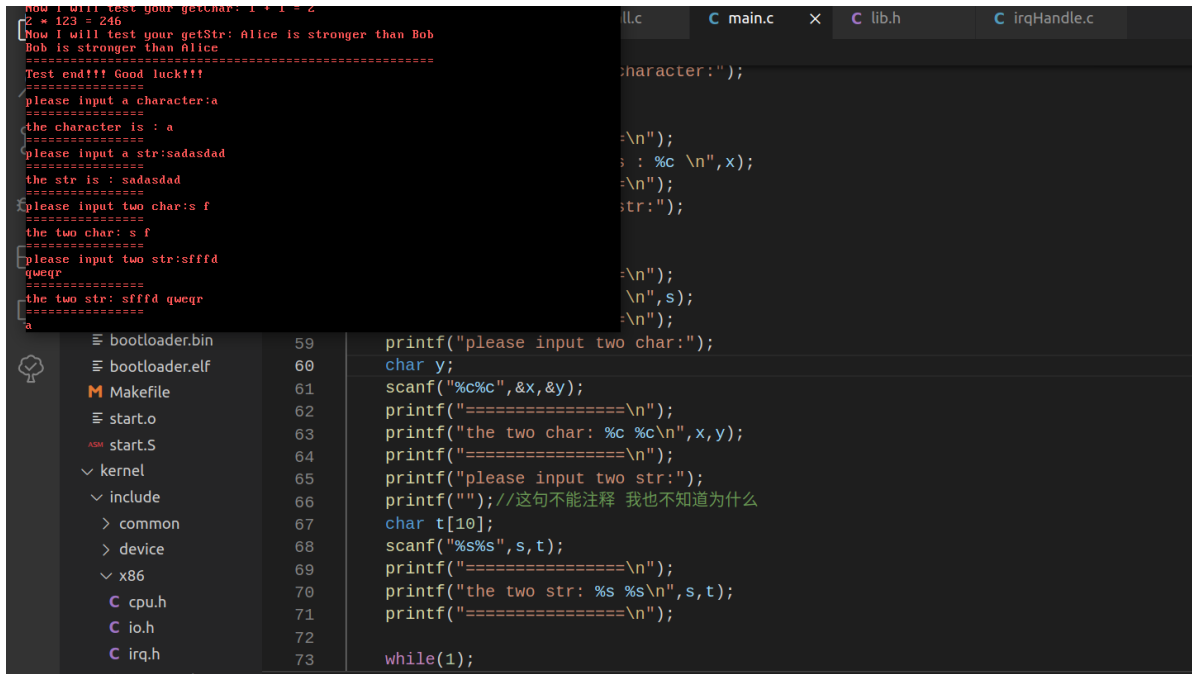
串口输出的putChar和putStr本就是在内核态进行的, 故只能在内核使用 (即kernel, 而且要在初始化串口之后) 。

另外这里是不是应该问printf和put的区别 (而不是get的区别) ?

因为串口的输出是用in和out指令 (端口映射) 实现的, 而printf是通过vga内存映射的方式实现的。

**challenge3: 比较关键的几个函数**challenge3: 如果你读懂了系统调用的实现, 请仿照printf, 实现一个scanf库函数。并在app里面编写代码自行测试。最后录一个视频, 展示你的scanf。(写出来加分, 不写不扣分)

时间所限, 实现了%c%s, 其他差不多同理, 视频放在文件夹里。



```
Now I will test your getChar: 1 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!!!
=====
please input a character:a
=====
the character is : a
=====
please input a str:sadasdad
=====
the str is : sadasdad
=====
please input two char:s f
=====
the two char: s f
=====
please input two str:sfffd queqr
=====
the two str: sfffd queqr
=====
a
```

```
59 printf("please input two char:");
60 char y;
61 scanf("%c%c",&x,&y);
62 printf("=====\\n");
63 printf("the two char: %c %c\\n",x,y);
64 printf("=====\\n");
65 printf("please input two str:");
66 printf("");//这句不能注释 我也不知道为什么
67 char t[10];
68 scanf("%s%s",s,t);
69 printf("=====\\n");
70 printf("the two str: %s %s\\n",s,t);
71 printf("=====\\n");
72
73 while(1);
```

**exercise23: 请结合gdt初始化函数, 说明为什么链接时用"-Ttext 0x00000000"参数, 而不是"-Ttext 0x00200000"。**

```
//gdt[SEG_UCODE] = SEG(STA_X | STA_R, 0,
0xffffffff, DPL_USER);
gdt[SEG_UCODE] = SEG(STA_X | STA_R,
0x00200000, 0x000fffff, DPL_USER);
//gdt[SEG_UDATA] = SEG(STA_W, 0,
0xffffffff, DPL_USER);
gdt[SEG_UDATA] = SEG(STA_W,
0x00200000, 0x000fffff, DPL_USER);
```

因为在初始化gdt, tss这一步骤中, initseg函数将用户的代码段和数据段的基地址设置为0x200000, 所以链接时只需要用逻辑地址0x0, 实际地址即0x200000, 如果用0x200000, 那么实际地址就变成了0x400000。

**conclusion1: 请回答以下问题: 请结合代码, 详细描述用户程序app调用printf时, 从lib/syscall.c中syscall函数开始执行到lib/syscall.c中syscall返回的全过程, 硬件和软件分别做了什么? (实际上就是中断执行和返回的全过程, syscallPrint的执行过程不用描述)**

调用printf后经过一系列操作到syscall函数,

```
int32_t syscall(int num, uint32_t a1, uint32_t a2,
                uint32_t a3, uint32_t a4, uint32_t a5)
{
    int32_t ret = 0;
    //Generic system call: pass system call number in AX
    //up to five parameters in DX,CX,BX,DI,SI
    //Interrupt kernel with T_SYSCALL
    //
    //The "volatile" tells the assembler not to optimize
    //this instruction away just because we don't use the
    //return value
    //
    //The last clause tells the assembler that this can
    potentially
    //change the condition and arbitrary memory locations.

    /*
        Note: ebp shouldn't be flushed
        May not be necessary to store the value of eax,
        ebx, ecx, edx, esi, edi
    */
    uint32_t eax, ecx, edx, ebx, esi, edi;
    uint16_t selector;

    asm volatile("movl %%eax, %0":"=m"(eax));
    asm volatile("movl %%ecx, %0":"=m"(ecx));
    asm volatile("movl %%edx, %0":"=m"(edx));
    asm volatile("movl %%ebx, %0":"=m"(ebx));
    asm volatile("movl %%esi, %0":"=m"(esi));
    asm volatile("movl %%edi, %0":"=m"(edi));
    asm volatile("movl %0, %%eax": : "m"(num));
    asm volatile("movl %0, %%ecx": : "m"(a1));
    asm volatile("movl %0, %%edx": : "m"(a2));
    asm volatile("movl %0, %%ebx": : "m"(a3));
```



```

asm volatile("movl %0, %%esi" :: "m"(a4));
asm volatile("movl %0, %%edi" :: "m"(a5));
asm volatile("int $0x80");
.....

```

syscall函数首先会保存寄存器旧值，然后将传来的参数存入对应的寄存器，通过 `asm volatile("int $0x80");` 进入内核，跳转到doIrq.S的irqSyscall，

```

irqSyscall:
    pushl $0 // push dummy error code
    pushl $0x80 // push interruption number into kernel
stack
    jmp asmDoIrq

```

这里会先把中断类型号和中断向量入栈，跳转到asmDoIrq，

```

.global asmDoIrq
asmDoIrq:
    pushal // push process state into kernel stack
    pushl %esp // esp is treated as a parameter
    call irqHandle
    .....

```

准备好irqHandle的参数TrapFrame \*tf，然后调用irqHandle，

```

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;
    case 0xd:
        GProtectFaultHandle(tf);
        break;
    case 0x21:

```

```

        keyboardHandle(tf);
        break;
    case 0x80:
        syscallHandle(tf);
        break;
    default:assert(0);
}
}

```

再根据中断类型号跳转到syscallHandle,

```

void syscallHandle(struct TrapFrame *tf) {
    switch(tf->eax) { // syscall number
        case 0:
            syscallWrite(tf);
            break; // for SYS_WRITE
        case 1:
            syscallRead(tf);
            break; // for SYS_READ
        default:break;
    }
}

```

再根据tf->eax的值也就是先前准备在eax寄存器的参数SYS\_WRITE=0跳转到syscallWrite,

```

void syscallWrite(struct TrapFrame *tf) {
    switch(tf->ecx) { // file descriptor
        case 0:
            syscallPrint(tf);
            break; // for STD_OUT
        default:break;
    }
}

```

再根据tf->ecx也就是先前准备在ecx寄存器的参数STD\_OUT=0调用syscallPrint,

```

void syscallPrint(struct TrapFrame *tf)//代码略

```

syscallPrint执行完打印后返回,

```

call irqHandle
addl $4, %esp //esp is on top of kernel stack
popal
addl $4, %esp //interrupt number is on top of kernel
stack
addl $4, %esp //error code is on top of kernel stack
iret

```

返回到asmDoIrq时，将指针移到一开始没push终端类型号和中断向量的地方（这中间要有一步popal对应前面的pushal），然后执行iret指令将当前栈顶的数据依次Pop至EIP，CS，EFLAGS寄存器，若Pop出的CS寄存器的CPL数值上大于当前的CPL，则继续将当前栈顶的数据依次Pop至ESP，SS寄存器，返回syscall。

```

asm volatile("movl %eax, %0":"=m"(ret));
asm volatile("movl %0, %%eax::"m"(eax));
asm volatile("movl %0, %%ecx::"m"(ecx));
asm volatile("movl %0, %%edx::"m"(edx));
asm volatile("movl %0, %%ebx::"m"(ebx));
asm volatile("movl %0, %%esi::"m"(esi));
asm volatile("movl %0, %%edi::"m"(edi));

asm volatile("movw %%ss, %0":"=m"(selector)); //ds is
reset after iret
//selector = 16;
asm volatile("movw %%ax, %%ds::"a"(selector));

return ret;

```

在syscall将eax寄存器的值作为返回值传出去，并恢复寄存器旧值。

**conclusion2: 请回答下面问题：请结合代码，详细描述当产生保护异常错误时，硬件和软件进行配合处理的全过程。**

在中断到来时，硬件行为如下：

```

old_CS = CS
old_EIP = EIP
old_SS = SS
old_ESP = ESP

```

```

target_CS = IDT[vec].selector           //target_cs是中断向量号为
vec的中断描述符里的selector
target_CPL = GDT[target_CS].DPL         //target_CPL是目标代码段
的DPL
if(target_CPL < GDT[old_CS].DPL)         //特权级检查!!! (请翻找
手册前面)
    TSS_base = GDT[TR].base             //检查通过, 获得TSS段的基
地址 (结构体地址)
    switch(target_CPL)                   //我要切换到ring几 (在本实
验中只有ring0)
    case 0:
        SS = TSS_base->SS0
        ESP = TSS_base->ESP0
    case 1:
        SS = TSS_base->SS1
        ESP = TSS_base->ESP1
    case 2:
        SS = TSS_base->SS2
        ESP = TSS_base->ESP2
    push old_SS
    push old_ESP
    push EFLAGS                           //把原来的eflags, cs,
eip都push进堆栈 (思考: 谁的堆栈)
    push old_CS
    push old_EIP

```

硬件会先保存CS, EIP, SS, ESP, 然后切换堆栈, 从硬件处理转入内核。

因为先前初始化过GDT, 所以这里会跳转到doirq.S的irqGProtectFault, 之后的处理过程与conclusion类似, 不赘述。

返回时硬件所做如下:

```
##### iret #####
//这个和前面的流程顺序上.....
old_CS = CS
pop EIP
pop CS
pop EFLAGS
if(GDT[old_CS].DPL < GDT[CS].DPL)    //???
    pop ESP
    pop SS
```

硬件会恢复寄存器旧值并返回内核态。

**conclusion3: 请回答下面问题 如果上面两个问题你不会，在实验过程中你一定会出现很多疑惑，有些可能到现在还没有解决。请说说你的疑惑。**

```
unsigned int elf = 0x100000;
void (*kMainEntry)(void);
kMainEntry = (void(*) (void))0x100000;

for (i = 0; i < 200; i++) {
    readSect((void*)(elf + i*512), 1+i);
}
```

一开始把ELF文件读入到指定位置的时候读到了0x100000，但是这个也是装载位置，装载的时候要是没有覆盖掉原elf文件的内容，应该会出问题，但是我目前没碰到问题，所以这个地方到底要不要处理，如果要处理，应该读到哪比较好？（后面loadumain我也是读到了0x200000）

手册中所述硬件对中断部分的执行如何在代码中体现？是通过int指令和iret指令吗？

另外在测试scanf的时候发现胡乱printf也会出bug，也不知道是我的scanf问题还是printf问题。。。

```

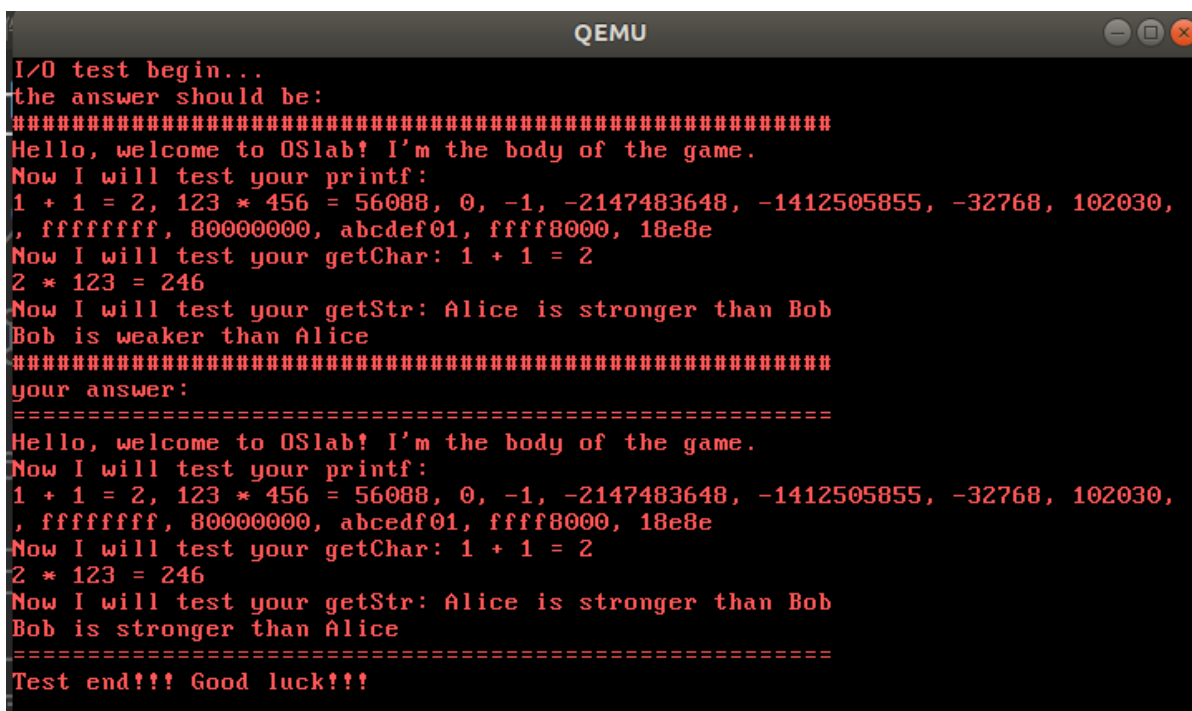
53     printf("please input a str: ");
54     char s[10];
55     scanf("%s",s);
56     printf("=====\n");
57     printf("the str is : %s \n",s);
58     printf("=====\n");
59     printf("please input two char:");
60     char y;
61     scanf("%c%c",&x,&y);
62     printf("=====\n");
63     printf("the two char: %c %c",x,y);
64     printf("=====\n");
65     printf("please input two str:");
66     printf(""); //这句不能注释 我也不知道为什么
67     char t[10];
68     scanf("%s%s",s,t);
69     printf("=====\n");
70     printf("the two str: %s %s",s,t);
71     printf("=====\n");
72
73     while(1);

```

**challenge4:** 根据框架代码，我们设计了一个比较完善的中断处理机制，而这个框架代码也仅仅是实现中断的海量途径中的一种设计。请找到框架中你认为需要改进的地方进行适当的改进，展示效果（非常灵活的一道题，不写不扣分）。

无。

## 实验结果



```

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,
, 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,
, 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 stronger than Alice
=====
Test end!!! Good luck!!!

```

## 四、实验的启示/意见和建议

bootloader一开始因为超过510字节，我按照lab1的方式改makefile优化，结果make play报错bootloader有问题，但是弄半天boot也不对，最后根据群里的提示改成这样就过了，这里手册里也没提到。。。

```
CFLAGS = -m32 -march=i386 -static \
        -fno-builtin -fno-stack-protector -fno-omit-frame-pointer \
        -Wall -Werror -Os -g -fno-asynchronous-unwind-tables
ASFLAGS = -m32
LDFLAGS = -m elf_i386
```