2022spring《操作系统》lab2实验报告

姓名：武雅琛
学号：201220102

> 23个exercise,3个challenge，5个task,3个conclusion

一、PART1: exercise

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）**
  - 因为读取的过程中**盘面转动**，在一个扇区读写完成的时候就已经到达了这个**扇区的末尾部分**，如果此时选择同一柱面和同一扇区的另一个磁头读写就要**反向转动盘片使得磁头回到起点**，这个过程等于盘面需要来回两次旋转，耗费了更多的时间。
  - 而如果连续存储在同一柱面同一盘片的连续扇区内，则一个扇区**读写完成后磁头刚好位于下一个扇区的起始位置**，无需调整盘片就可以开始下一轮的读写。如果超过一个柱面内一个盘片的内容存储在下一个盘片的相同柱面内，相当于读取完一个盘片后磁头又回到了起始位置，只需要一个电信号选择读取下一个盘片的磁头即可。

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）。**
  - 磁盘的内容先在同一柱面号上同一盘面的扇区上连续存储，继而在同一柱面的连续盘面上存储，最后移动磁头寻找柱面。
  - **LBA表示法的编号 = C * 磁盘的磁头数 * 每个磁道的扇面数 + H * 每个磁道的扇面数 + 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）**
  - 写了一个简单的程序使用gcc编译成ELF文件（执行视图）。

```
#test.c
#include<stdio.h>

char* string = "Hello World!\n";

int main()
{
        printf("%s", string);
        return 0;
}
```

  - 输入命令

```
$ readelf -l test
```

- 得到

```
oslab@oslab-VirtualBox:~$ readelf -l test

Elf 文件类型为 DYN (共享目标文件)
Entry point 0x550
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
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000860 0x0000000000000860  R E    0x200000
  LOAD           0x0000000000000db8 0x0000000000200db8 0x0000000000200db8
                 0x0000000000000260 0x0000000000000268  RW     0x200000
  DYNAMIC        0x0000000000000dc8 0x0000000000200dc8 0x0000000000200dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000718 0x0000000000000718 0x0000000000000718
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000db8 0x0000000000200db8 0x0000000000200db8
                 0x0000000000000248 0x0000000000000248  R      0x1
```

  - 程序从文件偏移0x550开始执行, 并且程序头表从偏移64的地方开始(因为ELF头大小为64)。
  - 程序头表显示该可执行文件存在九个段（Segment），这些段的内容包括的对应的节为：

    （INTERP、DYNAMIC是和动态链接相关的，NOTE段保存了一些专有信息，第一个LOAD类型的段是只读代码段：包含了只读数据节和只读代码节等，第二个LOAD类型的段是可读写数据段：包括可读写数据节和未初始化数据节（.bss）等。**LOAD段存在ELF文件到虚拟空间的映射关系，也就是在执行时被加载的段**）

```
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
```

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）？**
  - **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）。~~
>
> 2、~~加载内核到内存某地址并跳转运行（lab1）。~~

1、在**bootloader中的start.S**文件中OS从实模式进入保护模式。

2、在**boot.c中的bootMain函数**中将kernel搬到内存上某地址，并且通过`kMainEntry`转入执行。

- **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。
>
> 1. 初始化串口输出（`initSerial`）
> 2. 初始化中断向量表（`initIdt`）
> 3. 初始化8259a中断控制器（`initIntr`）
> 4. 初始化 GDT 表、配置 TSS 段（`initSeg`）
> 5. 初始化VGA设备（`initVga`）
> 6. 配置好键盘映射表（`initKeyTable`）
> 7. 从磁盘加载用户程序到内存相应地址（`loadUMain`）

这一步在**kernel的main.c的**`kEntry`函数中实现（同时看Makefile可知kEntry是kernel的代码入口地址）。

至于具体的模块定义位于**kernel/kernel**目录下各个设备的驱动软件中。

- 步骤三

> 最后进入用户空间进行输出。
>
> 1. 进入用户空间（`enterUserSpace`）
> 2. 调用库函数，输出各种内容！

1、在**kernel/kvm.c中定义了**`loadUMain`函数，在把用户软件从磁盘加载到内存相应地址后调用`enterUserSpace`函数进入用户空间执行。

2、在**app/app.c**中调用库函数输出内容。

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）**
  - **可屏蔽中断和不可屏蔽中断：**可屏蔽中断的处理是取决于标志寄存器（`eflags`）中的中断控制位（`IF`）（`IF`=1,cpu接受可屏蔽中断，反之不处理），大多数由外部硬件引发的中断时可屏蔽中断。不可屏蔽中断的处理与中断控制位没有关系，中断向量号为2，即硬件自身的NMI中断，只有在特定严重的情况下才会产生，比如硬件故障和掉电等。
  - **软中断：**软中断是在cpu执行的用户软件指令序列中由于执行**访管指令**翻转到内核态来获取OS服务的方式。
  - **外部中断：**外部中断是cpu**处理外部设备事件**的一种机制。当一个外部设备事件到来时驱使cpu进入内核态OS的中断处理程序进行处理后再回到用户态执行用户代码。比如I/O设备，磁盘等等。
  - **异常：**异常是cpu正常运行软件的过程中**软件运行内部产生的**需要进入内核态由OS处理的特殊情况和故障：比如除数为0，缺页异常等等。

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。）**
  - 不是呢。
  - IRQ全称为Interupt Request，即中断请求。它是相对于外部中断请求的定义。它只包括了外部中断的编号，对应着8259A芯片的中断引脚。

- 中断号是一个相对于中断向量表/中断描述符表的定义。这意味着他包括外部中断和内部异常的所有情况，这也对应着对于cpu来说，中断处理的过程就是读取INTR引脚后用中断号查询中断描述符表找到对应OS中断处理程序的过程，二者时没有差异的。

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。）**
  - 为了保存当前程序断点让cpu在执行中断处理之后能正确返回断点部分执行。因为在硬件处理中断的过程中要根据中断号查找中断描述符表找到中断描述符，按照**中断描述符填写`cs:eip`，将段寄存器`cs`的`cpl`修改到最高特权级，转入内核态**，将`eip`指向中断处理程序的第一条指令。所以一但转入中断处理的软件部分就已经进入了内核态，**用户程序运行时`cs:eip`的内容已经被覆盖。**
  - 出于对程序状态的保护，eflags在IA-32i架构上只能通过一些**特定的指令修改其中的某些标志**，没有提供对这个寄存器修改或者操作的指令。

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）**
  - 比如说用户程序请求**I/O设备输入**，把当前程序信息放在了一块内存区域里，处理中又发现了一个**缺页异常**，然后把设备输入的程序信息存到了同一块区域内，那么第一次的信息就被冲刷了。等处理了缺页异常，回到输入程序，输入程序结束以后，之前存储用户程序的空间里存储的已经不是用户程序的信息了，而是输入程序的信息，**失去了用户程序运行的`cs:eip`**，cpu就无法回到用户程序的下一条指令继续执行了。

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?**

```
1 if(GDT[old_CS].DPL < GDT[CS].DPL)    //???
2  pop ESP
3  pop SS
```

  - `old_cs`是当前内核态的CS段寄存器的内容，它的DPL是00，也就是最高特权级。
  - `cs`是从栈里弹出的要返回的程序状态中的CS段寄存器的内容，如果GDT[CS].DPL为00，则结束中断**返回的程序依然在内核态**，那么**不需要切换**到用户栈。
  - 如果GDT[CS].DPL>00，那么要返回用户态，此时要将内核栈切换回中断之前的用户栈，此时，就要将**一开始压入内核栈**的**用户栈的寄存器内容弹出**，恢复栈帧现场。
  - plus：(粗浅的想法)这里也说明为什么TSS没有ring3的项，这是因为每次都是从ring3中断进入更高的特权级，**只需要将当前的栈帧相关的寄存器压入切换后的栈中**即可。也就是我们永远**没有直接进入ring3**的需求，也就不需要提前在TSS中存储表项记录栈帧。

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?**
  - 我觉得**会**。
  - 通用寄存器又被称之为程序运行的现场信息，表征着**cpu当前运行程序的状态。**
  - 如果不做保存和恢复，在中断处理程序结束之后，cpu即便可以回到之前程序的断点**程序状态也已经被破坏**，所以很有可能会产生不可恢复的错误。

exercise11

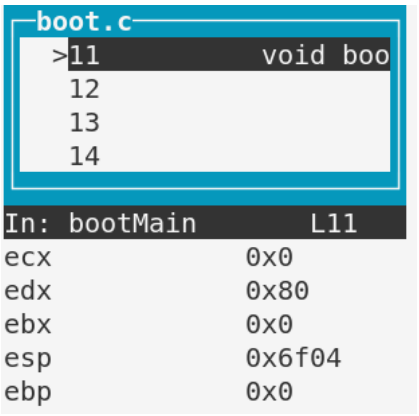
- **我们会发现软中断的过程和硬件中断的过程类似，他们最终都会去查找IDT，然后找到对应的中断处理程序。为什么会这样设计？（可以做完实验再回答这个问题）**
  - 中断是**操作系统**为用户程序运行**提供服务**的方式。
  - 外部中断是程序需要外部设备协作时外部设备**请求操作系统服务**的过程。
  - 软中断是程序正常在运行时主动**请求操作系统服务**的过程。
  - 外部中断通过终端控制器向cpu提供INTR引脚来提供中断号请求cpu响应提供服务，软中断是通过访管指令提供中断号请求服务。
  - 但是对于cpu来说，硬件中断和软中断是透明的，它只需要知道根据中断号查询IDT来切换到中断处理程序执行即可。
  - 这样使得cpu提供操作系统服务时更有**一致性**。

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）**
  - 因为在跳转到`bootMain`执行的时候需要创建若干局部变量，在c语言中局部变量保存在当前程序的栈中。所以要提前为程序准备号栈帧。

```
// void bootMain()
    int i = 0;
    int phoff = 0x34;
    int offset = 0x1000;
    unsigned int elf = 0x100000;
```

  - 不设置esp的内容相当于没有给下面运行的程序分配内核栈帧，很可能会发生非法的内存访问，导致程序严重错误。
    - 但我试了一下，没有出事。用之前的esp也不会冲突，虽然这样可能和程序虚拟空间的分配不符合，但不妨碍运行。



  - 设成`0x100000`发生冲突，装载kernel失败。

exercise13

- **上面那样写为什么错了？**
  - 它没有按照装载ELF文件的要求，通过遍历程序头表找到type为LOAD的项装入Physicaddr，而是将ELF文件从第一个段在文件开始的字节后的`200*512`个字节向前挪动了`ph->off`个字节。

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?**
  - ```
    //boot.c
    void (*kMainEntry)(void);
    kMainEntry = (void(*)(void))elfh->entry;
    ```

    可以看出KMainEntry是boot.c中的一个符号，但是它的类型是一个没有参数和返回值的函数指针，根据装载的kernel的elf文件包含的入口地址信息给它赋值。
  - ```
    #kernel/Makefile
    $(LD) $(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf $(KOBJS)
    ```

    可以看出在kernel的Makefile文件中使用了`ld`连接器链接，并且用参数`-e`指定了程序的入口地址为`kEntry`，这是在`main.c`定义的另一个符号，又用`-Ttext`指定了可执行程序代码段开始的地址为`0x00100000`。综上说明，链接后的kernel装载后第一个代程序字节在`0x00100000`，`KEntry`就是标识在这个地址的符号。而在`boot.c`中`KMainEntry`标识的也是这个地址。
  - 综上，它们**标识了同一个内存地址0x00100000**，可以通过`KMainEntry`跳转到同样的地址，但是本质上它们是**不同程序的不同符号。**

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?**
  - **asmDoIrq**
  - 因为切换到中断处理程序后在执行的过程中也会用到通用寄存器，为了**防止破坏此时的程序现场**，要在中断处理程序的准备阶段在内核栈保存通用寄存器，在恢复阶段恢复这些内容。
  - 也有用来作为**参数的一部分**传入中断处理程序的需求，方便中断处理中对程序进行分析。
  - 这是所有中断处理共同经过的过程。

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.....）**
  - 在`irq_handle.c`文件中找到了`irqHandle`的函数原型。

    ```
    void irqHandle(struct TrapFrame *tf) ;
    ```

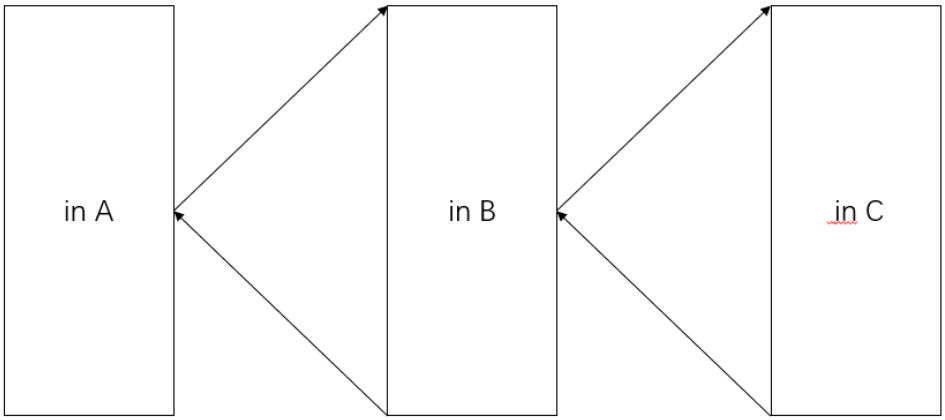
    - 这里发现是有一个参数`tf`，类型是`TrapFrame`，定义如下：

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

- 然后结合在`doirq.S`里的操作push irq, pusha，刚好对应了结构体的内容。而且由于在内核栈中地址从高地址向低地址长，所以会先push高地址的irq。
- 那么在push了`TrapFrame`的内容之后，esp刚好指向了结构体的第一个字节的地址，**push esp就相当于将结构体的地址压入栈里**，类似于c语言将调用参数压入栈中的操作，这样中断处理程序就可以根据`tf`传入的参数进一步处理了。

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？）**
  - example：在键盘上连续敲下ABC，并且时间间隔**足以发生中断嵌套。**
    - 显示：**CBA。**
    - 运行过程be like：



    - 我觉得会在**内核栈里递归**，然后**C的输入先结束，继而是B，最后处理A。**

exercise18

- **阅读代码后回答，用户程序在内存中占多少空间?**
  - **4GB。**
  - 因为可以看到宏`SEG`将**粒度G设置为1**，并且用户数据段和代码段在操作系统中采用扁平化处理，**基地址都是**`0x00200000`。而limit = `0xfffff`，由此得知用户程序最多可以占用**1MB * 4KB = 4GB**空间。

exercise19

- **请看syscallPrint函数的第一行：**

  `int sel = USEL(SEG_UDATA);`

  请说说sel变量有什么用。（请自行搜索）
  - sel变量在函数一开始存储了用户态数据段的段选择子的内容。
  - 这是因为要打印的内容保存在**用户数据段**中，传入中断处理程序的只是**指向这段用户内存的一个指针，而不是字符的ascii码。**所以需要用户态数据段的段选择子在输出的时候找到用户段的位置，取出对应的输出的内容。
  - es是一个特殊的**附加段寄存器**，存放一些其他的段选择子。

```
//irqHandle.c
asm volatile("movw %0, %%es"::"m"(sel));//将段选择子的内容存入附加段寄存器（es）中
    for (i = 0; i < size; i++) {
        //指定es作为访问str的内容的段选择子，也就是去用户态数据段寻找str的ascii码
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
```

exercise20

- **paraList是printf的参数，为什么初始值设置为&format?**
  - `printf`可能会有不止一个参数，每个参数的类型都是`char*`。
  - `paraList`是一个**指针数组**，需要它可以按照下标或者指针访问传入的若干参数。
  - 所以`paraList`是一个**指向指针的指针**，也就是一个标准的**二级指针**，所以它**初始化为栈中第一个参数的地址**，以便向下继续访问第二，第三.....个参数。
- **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?**
  - (**&format**)(存放参数的首地址) + **4** + **4**
- **所以当我解析format遇到%的时候，需要怎么做?**
  - 将state**设置为1**, ++i;
  - 在下一次循环处读出%后的格式符，然后通过**paraList找到对应的参数。**
  - 用框架代码提供的转换函数转换为字符串并填写到buffer里，**index += 4，指向下一个参数。**
  - 将state**复原为0**,++i，进行接下来的处理。

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,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？）。**

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

  - `syscall`是封装好的系统调用API，我们找到`syscall`函数原型：

```
int32_t syscall(int num, uint32_t a1,uint32_t a2,uint32_t a3,
uint32_t a4, uint32_t a5);
```

    大概阅读函数的内容，就是将**当前程序运行状态**的寄存器**现场保存用户栈**里，然后用**系统调用的参数来替代通用寄存器的内容。**然后内联汇编访管指令发生系统调用，结束后**恢复现场**，并且通过函数返回值返回系统调用返回值。

    - **num：系统调用号。**`SYS_WRITE`是0号系统调用，即写入设备。
    - **a1：设备号。**`STD_OUT`是0号输出设备，也就是标准输出。
    - **a2：字符串的首地址。**将buffer中的字符输出，传入数组首地址。
    - **a3:输出字符的个数。**
    - 陷入系统调用以后，就可以和之前的部分衔接，查阅idt.....。比较重要的是**寄存器内容会压入内核栈组成重要的`Trapframe`参数结构体。**可以看出在下面的部分取出了**参数系统调用号**，判断系统调用为`write`，转入对应的部分。

```
//kernel/irqHandle.c
void syscallHandle(struct TrapFrame *tf) {
    switch(tf->eax) { // syscall number
        case 0:
            syscallWrite(tf);
            break; // for SYS_WRITE
```

- 然后又取出了**设备号**，选择**标准输出**：

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

- 关于访问字符串的细节在exercise19中有所涉略，不再赘述。

exercise22

- **记得前面关于串口输出的地方也有getChar和getStr吗？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。**
  - 大概结合前后文应该指的是前面也有`putChar`和`putStr`这两个函数。
  - 首先`syscall.c`中的函数均为**输入函数**，也就是从外部设备获取输入，依赖的是**系统调用的中断机制**，在底层取出**键盘维护的队列的内容**，而键盘维护的队列内容是通过处理键盘中断填写的，具体依赖的**键码来自于`0x60`号端口。**

    它可以在**用户态调用**，通过封装了操作系统提供的API来**触发中断**陷入内核处理即可。
  - `serial.c`里定义的是输出函数：

    ```
    //serial.c
    void putChar(char ch) {
        while (serialIdle() != TRUE);
        outByte(SERIAL_PORT, ch);//SERIAL_PORT = 0x3f8
    }
    ```

    依赖的是向`0x3f8`**号串口输出字符。**并且由于我们在编译的时候指定了串口数据即时输出到标准输出`stdio`（宿主机），也就是**宿主机的终端输出。**

    它必须在**内核态**中调用，属于**内核库**的内容，比如在键盘中断输入回显的时候调用了`putChar`。

    plus：但是我忘记用了，全程*gdb*下来*QAQ*

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"。**
  - 查找`kvm.c`：

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

- 可以看出框架代码在操作系统和用户软件之间采用了**非扁平化处理**，在操作系统的代码都在`0x0`以上运行，而用户软件在`0x00200000`上运行。这意味着在**用户软件的逻辑地址为**`0x0`，结合用户段寄存器段基址'`0x00200000`。**实际上的线性地址为**`0x00200000`。但是用户软件眼中这个过程是**透明**的，它认为自己运行在`0x0`往上的**进程虚拟地址空间**中。

二、PART2:challenge

challenge1

- **既然错了，为什么不影响实验代码运行呢?**
  - 首先，他把elf文件按直接从**磁盘搬到了原本装载程序段***的内存地址上。
  - 然后这个`offset`就是这个**段相对于elf文件开头的文件偏移量。**
  - 所以这个过程就很滑稽，就是把elf文件从第一个段起始的内容**向开头挪动了**`offset`**的空间。**
  - 然后更巧的是，把代码段挪到了`0x100000`以后，**数据段和代码段在文件中的相对位置和装载以后内存的相对位置也一样**，然后稀里糊涂的让LOAD段到了正确的位置上。

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

请解释框架代码在干什么。

- **1.KeyboardHandle**
  - ```
    uint32_t code = getKeyCode();
    ```

    首先调用键盘的**驱动程序**`getKeyCode`（在这个程序当中主要是和端口交互，处理按键的按下和弹起，还有一些和键盘控制相关的键码，eg.caps lock）接收到一个**键码。**
  - ```
    if (code == 0xe)
        { // 退格符
            ...
        }
        else if (code == 0x1c)
        { // 回车符
            ...
        }
    ```

    首先对显示的特殊键码处理，包括**退格和回车。**

    - 退格比较粗暴，只能在当前行退格，光标退一列之后将之前位置**显存的内容清空。**所以也不能在退格中换行，也**没有在字符缓存中删除一个字符，就一个表面删了实际没删的操作**，可以测试出来这个问题。

- 回车精致一点，首先让**光标跑到了下一行开始的地方**，其次在字符缓存中放入了对应的ascii码（高度怀疑是为了判断输入结束），还兼顾了**滚屏**的问题

```
else if (code < 0x81)
    { // 正常字符
        // 注意输入的大小写的实现、不可打印字符的处理
        char character = getChar(code); //将键码转换为ascii码
        if (character != 0)//合法的输入内容
        {
            putChar(character); // put char into serial，在寄主机的终端显示
            uint16_t data = character | (0x0c << 8);
            keyBuffer[bufferTail++] = character;//写入字符缓存，等待程序输入时取出
            bufferTail %= MAX_KEYBUFFER_SIZE;   //采用循环队列，防止队列空间使用殆尽。FIFO替换缓存内容。
            int pos = (80 * displayRow + displayCol) * 2;//找到显示的显存位置
            asm volatile("movw %0, (%1)" ::"r"(data), "r"(pos + 0xb8000));//直接写到显存里显示到屏幕上
            //进行一些换行，滚屏的处理
            displayCol += 1;
            if (displayCol == 80)
            {
                displayCol = 0;
                displayRow++;
                if (displayRow == 25)
                {
                    scrollScreen();
                    displayRow = 24;
                    displayCol = 0;
                }
            }
        }
    }
    //更新光标的位置
    updateCursor(displayRow, displayCol);
}
```

最后就是普通字符的处理，出于方便，我直接增加了代码注释解释代码。

- **2.syscallprint**

  注释解释。

```
void syscallPrint(struct TrapFrame *tf)
{
    //拿到用户态的数据段选择子，方便之后访问给出了用户段地址的字符串内容
    int sel = USEL(SEG_UDATA); // segment selector for user data, need further modification
    //即将输出的字符串首地址
    char *str = (char *)tf->edx;
    //即将输出的字符串字符数目
    int size = tf->ebx;
    int i = 0;
    int pos = 0;
    char character = 0;
    uint16_t data = 0;
```

```
    //将保存下来的用户态段选择子内容放入附加段选择子
    asm volatile("movw %0, %%es" ::"m"(sel));
    //通过循环逐个打印字符串字符，共执行size次，输出size个字符
    for (i = 0; i < size; i++)
    {
        //从用户地址中取出一个字符（注意这里使用了附加段选择子，不然在非扁平模式中找不到这个用户软件地址）
        asm volatile("movb %%es:(%1), %0"
                     : "=r"(character)
                     : "r"(str + i));
        //  完成光标的维护和打印到显存

        if (character != '\n')
        {
            //普通字符
            data = character | (0x0c << 8);
            pos = (80 * displayRow + displayCol) * 2;
            //直接修改对应的显存内容,写入数据
            asm volatile("movw %0, (%1)" ::"r"(data), "r"(pos + 0xb8000));
            //处理了一下光标的移动，换行和滚屏问题
            displayCol += 1;
            if (displayCol == 80)
            {
                displayCol = 0;
                displayRow++;
                if (displayRow == 25)
                {
                    scrollScreen();
                    displayRow = 24;
                    displayCol = 0;
                }
            }
        }
        else
        {
            //回车字符的输出，把光标挪到下一行起始，注意一下滚屏即可
            displayCol = 0;
            displayRow++;
            if (displayRow == 25)
            {
                scrollScreen();
                displayRow = 24;
                displayCol = 0;
            }
        }
    }
    tail = displayCol;
    //更新光标，这是和vga设备的端口通信
    updateCursor(displayRow, displayCol);
}
```

- **syscallGetChar和syscallGetStr对应于“read”系统调用**

  注释解释。

```
void syscallGetChar(struct TrapFrame *tf)
```

```
{
    //清空标准输入缓存（可能是怕之前误触键盘影响吧，虽然我觉得未必有必要）
    bufferHead = 0;
    bufferTail = 0;
    keyBuffer[bufferHead] = 0;
    keyBuffer[bufferHead + 1] = 0;
    char c = 0;

    //如果不发生输入，按照初始化的内容，队列头部始终为0
    while (c == 0)
    {
        //enable和disalbeInterrupt是一对简单的内联汇编，负责打开IF位和清空IF位
        //enableInterrupt开启IF位，允许cpu处理键盘中断
        enableInterrupt();
        c = keyBuffer[bufferHead];
        //串口显示一下输入的字符，在实验中会显示在寄主机终端上
        putChar(c);
        disableInterrupt();
    }
    //通过寄存器把结果传回用户态（popal）
    tf->eax = c;

    char wait = 0;
    //说实话啊，我没意识到这是在干啥QAQ
    while (wait == 0)
    {
        enableInterrupt();
        wait = keyBuffer[bufferHead + 1];
        disableInterrupt();
    }
    return;
}

void syscallGetStr(struct TrapFrame *tf)
{
    char *str = (char *)(tf->edx); // str pointer
    int size = (int)(tf->ebx);     // str size
    //清空缓存，初始化
    bufferHead = 0;
    bufferTail = 0;
    for (int j = 0; j < MAX_KEYBUFFER_SIZE; j++)
        keyBuffer[j] = 0; // init
    int i = 0;

    char tpc = 0;
    //进行size个字符的输入或者回车结束输入
    while (tpc != '\n' && i < size)
    {
        //和GetChar一样，得到一个字符输入
        while (keyBuffer[i] == 0)
        {
            enableInterrupt();
        }
        tpc = keyBuffer[i];
        i++;
        disableInterrupt();
    }

```

```
    int selector = USEL(SEG_UDATA);
    //将用户数据段选择子写入附加段选择子供使用
    asm volatile("movw %0, %%es" ::"m"(selector));
    int k = 0;
    for (int p = bufferHead; p < i - 1; p++)
    {
        //把输入缓存的内容挪到用户软件提供的地址上，记得用附加段选择子找到用户段的位置
        asm volatile("movb %0, %%es:(%1)" ::"r"(keyBuffer[p]), "r"(str + k));
        k++;
    }
    //出于稳妥，写个'\0'进去，符合c语言对字符串结束的定义
    asm volatile("movb $0x00, %%es:(%0)" ::"r"(str + i));
    return;
}
```

- 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）。
    - 修改了键盘处理**退格符**的一个小小小小问题：可以同时对**字符缓存做了处理**。

```
//KeyboardirqHandle()
...
if (bufferTail != bufferHead)
    {
        keyBuffer[--bufferTail] = 0xe;//存入退格符的键码提示字符缓存处理
    }
...
//syscallGetstr()
...
    if (keyBuffer[i] == 0xe)//如果不再行首则
    {
        if (i > 0)
        {
            --i;
        }
        keyBuffer[i] = 0x0;
    }
...
```

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）

- 只写了%s和%d，其他的实现原理大差不差。
- 视频见附件。

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）。

空。

三、PART3:task

task1

- **填写boot.c，加载内核代码。**
  - **step1**：查看kernel的Makefile文件，可以看到在连接时用参数`-Ttext`指定了代码段的第一个字节从`0x00100000`开始，然后发现在原始版本里把从磁盘读elf文件的内存地址设置为`0x00100000`，那么装载的时候就会把文件的内容覆盖掉。

kernel > Makefile

```
8   LDFLAGS = -m elf_i386
9
10  KCFILES = $(shell find ./ -name "*.c")
11  KSFILES = $(shell find ./ -name "*.S")
12  KOBJS = $(KCFILES:.c=.o) $(KSFILES:.S=.o)
13  #KOBJS = $(KSFILES:.S=.o) $(KCFILES:.c=.o)
14
15  kmain.bin: $(KOBJS)
16      $(LD) $(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf $(KOBJS)
17      @#objcopy -S -j .text -j .rodata -j .eh_frame -j .data -j .bss -O binary kMain.elf kMain.bin
18      @#objcopy -O binary kMain.elf kMain.bin
19      @../utils/genKernel.pl kMain.elf
20      @#../utils/genKernel.pl kMain.bin
```

  - **step2:**解析elf文件头的内容，修改`kMainEntry`并且找到程序头表在内存的起始和结尾位置。

```
1  //boot.c
2      ELFHeader* elfh = (ELFHeader*)elf;//elfh:the head of elf
3      phoff = elfh->phoff; //程序头表在文件中的偏移量
4      kMainEntry = (void(*)(void))(elfh->entry);//函数指针：指向kernel的第一行指令
5
6      ProgramHeader* ph, *eph;//ph:entry of programheader; eph:the next of the last ehtry of programheader
7      ph = (ProgramHeader*)(elf + phoff);
8      eph = ph +elfh->phnum;
```

  - **step2:**根据程序头表项的内容装载段（*ps:注意.bss段的装载*）

```
1  //boot.c
2      for (; ph < eph; ++ ph)
3      {
4          if (ph->type == PT_LOAD)
5              {
6                  for (unsigned i = 0; i < ph->filesz; ++ i)
7                      *((char*)(ph->paddr + i)) = * ((char*)(elf + ph->off + i));
8                  if (ph->memsz > ph->filesz)
9                  {
10                     //.bss段
11                     for (unsigned i = ph->filesz; i < ph->memsz; ++ i)
12                     {
13                         *((char*)(ph->paddr + i)) = 0;
14                     }
```

```
                }
            }
    }
```

task2

- **完成Kernel的初始化，这应该是本实验最简单的地方了。**
  - 找到kernel对于各个外部设备的软件所在的文件，然后在`main.c`中填写初始化函数即可。

```
//kernel/main.c
// TODO: 做一系列初始化
    // initialize idt
    initIdt();
    // iniialize 8259a
    initIntr();
    // initialize gdt, tss
    initSeg();
    // initialize vga device
    initVga();
    // initialize keyboard device
    initKeyTable();
```

task3

- **完成setIntr和setTrap函数。**
  - 以setIntr先看函数原型。

```
//idt.c
static void setIntr(struct GateDescriptor *ptr, uint32_t selector,
uint32_t offset, uint32_t dpl);
```

  - ptr指向填写的中断描述符的地址，通过ptr访问结构体。

```
//memory.h
struct GateDescriptor {
    uint32_t offset_15_0      : 16;
    uint32_t segment          : 16;
    uint32_t pad0             : 8;
    uint32_t type             : 4;
    uint32_t system           : 1;
    uint32_t privilege_level  : 2;
    uint32_t present          : 1;
    uint32_t offset_31_16     : 16;
};
```

  - 此处的selector给的并非16位的段选择子的内容，而是对应**段选择子的`index`索引部分**（索引到GDT的段描述符的内容），因此可以通过框架代码给的**宏`KSEL`和`USEL`**分别来填写**中断门（内核态）**和**陷阱门（用户态）**中断描述符的段选择子的其它内容即可（**主要是填写了DPL**）。
  - offset（实际便是**中断处理程序的地址**）使用从中间折断填入两个部分即可，dpl填写到privilege_level。
  - 其他部分根据实验手册固定填写内容。

- 代码部分以setInstr为例，setTrap大差不差，只需要修改type的值即可：

```
//idt.c
//static void setInstr(...)

    ptr->offset_15_0 = offset & 0xffff;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = 0xe;
    ptr->system = 0;
    ptr->privilege_level = dpl;
    ptr->present = 1;
    ptr->offset_31_16 = (offset >> 16) & 0xffff;
```

- **为initIdt填空。**
  - 看着`Doirq.S`里的函数一顿填就可以啦。

task4

- **填充中断处理程序，保证系统在发生中断是能合理处理。**

  参考上面给出的中断程序声明和`Doirq.S`中压入栈的中断号即可。

```
//irqHandle.c
        case 0xd : GProtectFaultHandle(tf);break;
        case 0x21 : KeyboardHandle(tf);break;
        case 0x80: syscallHandle(tf);break;
```

task5

- **完成lib/syscall.c（不是kernel的lib，是外部库函数）里的库函数，分别对应输入和输出函数。**
  - **完成getChar()**
    - 程序获取外部设备输入，必然需要系统调用，所以参考`syscall.c`提供的系统调用API可以得到：

      SYS_READ：系统调用号`0x1`，读取

      STD_IN：设备号`0x0`，字符输入

      ```
      char character = syscall(SYS_READ, STD_IN , 0, 0, 0, 0);
      ```

  - **完成getStr()**
    - 同样参考API可以得到：

      SYS_READ：系统调用号`0x1`，读取

      STD_STR：设备号`0x1`，字符串输入

      但是要注意每次系统调用返回的字符缓存长度存在最大值，所以最好使用循环，如果输入过多可以分割为多次系统调用依次输入。

四、PART4：conclusion

conclusion1

- 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）
  - **step 1**：将**当前用户程序状态**的通用寄存器现场存放在**用户栈**中，防止丢失（通过**用户态函数局部变量**的形式）。
  - **step 2：**将传入的**系统调用号**（SYS_WRITE == 0x0（eax））和相应的**参数**（STD_OUT == 0x0（ecx），字符串在用户空间的地址（edx）以及字符串的长度（ebx））**存入通用寄存器**中，以达到传入内核态的目标。
  - **step 3**：执行**访管指令**`int $0x80`，转入**硬件系统调用服务。**
  - **step 4：**硬件基于中断号**查询中断描述符表**idt取出0x80号表项，取出中断描述符的dpl，**判断当前cpl <= dpl**（系统调用的中断描述符dpl = 3，可以从用户态访问）。因为是系统调用，此时的**cpl > dpl**（此处为中断处理程序段选择子的dpl），要进行**模式翻转。**
  - **step 5**：首先将**用户栈**切换到**内核栈。**读**TR**寄存器访问到**TSS段**得到**ring0的段选择子和栈指针填写到ss和esp**，然后将ring3的段选择子和栈指针压入内核栈等待最后弹出恢复。
  - **step 6**：将用户态的**cs、eip和eflags寄存器**压入内核栈，即用户程序断点和运行状态。
  - **step 7**：根据中断描述符中的**段选择子**和**异常处理程序的基地址**重新填写`cs:eip`，转入异常处理程序，进入操作系统**软件处理服务。**
  - **step 8：**首先转入`irqSyscall`函数，**压入中断号**`0x80`，又在`asmirqHandle`函数中将**通用寄存器中保留的参数一并压入内核栈**中，最后压入**栈顶指针**`esp`，从而构造**结构体指针参数**`Trapframe* tf`，调用`irqHandle`中断处理程序。
  - **step 9**：根据结构体保存的中断号`0x80`，转入**系统调用处理程序**`syscallHandle`。
  - **step 10：**根据结构体保存的**系统调用号**`0x0`，转入`syscallWrite`执行，又查询得到**设备号**`0x0`为标准输出，转到`syscallprint`执行。
  - **step 11**：`syscallprint`执行结束后层层返回到`asmirqHandle`，将内核栈中的**参数恢复到通用寄存器**中，并且恢复栈帧状态，最后**执行**`iret`**指令**进行模式翻转恢复状态，再次转入**硬件处理。**
  - **step 12**：恢复在内核栈里先前保存的**断点、程序状态和用户栈帧**内容，转入**用户程序执行。**
  - **step 13**：回到`syscall`中弹出**用户栈中保存的用户程序现场**，返回。

conclusion2

- 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。
  - 发生保护异常错误时，硬件首先在一条指令的执行时检测到了程序内部发生的保护异常错误。cpu判断到异常事件后立即**关中断（清空IF，防止破坏断点），保存程序断点（**`cs:eip`**）和程序状态（eflags）。**然后根据发生异常的中断号`0xd`**查询idt填写**`cs:eip`**转入中断处理程序**（这一段过程已经在conclusion1涉及）。
  - 首先进入`irqGProtectFault`函数，压入**中断号**`0xd`，转入`asmirqHandle`压入其余的参数（这一段在conclusion1有涉及）。
  - 在`irqHandle`处理程序中根据中断号`0xd`转入对应的中断处理程序`GProtectFaultHandle`，这里就直接`assert(0)`自己摆烂了。

conclusion3

- 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。
  - `syscallGetchar`，代码的末尾部分出现了`wait`变量，没明白在干什么。因为每次syscallGetchar都会清空输入缓存，为什么还要等下一个字符来。（我猜是为了在输入的时候创造一种回车的效果，因为连续输入两个字符也会结束字符输入）

五、PART5：feeling

- 这框架代码真的让我写的充满眼泪，尤其是在写scanf的地方无数次想跑路不写这个challenge了。
- 不过对了解中断和异常很有帮助，教学意义还是很大的。