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）

答：因为读取的过程中盘面是转动的，如果将信息存在同一柱面同一扇区，就会导致转至其他扇区的时候，无法读入消息

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）。

答：C*（一个柱面扇区的数量）*（磁头的数量）+H*（一个柱面扇区的数量）+S

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）。

答:

读 ELF 头：　　$ readelf -h xxx

读程序头表：　$ readelf -l xxx

如下图是我读出的 kernel 里的 kMain.elf

```
oslab@oslab-VirtualBox:~/桌面/lab2/lab2$ readelf -l kernel/kMain.elf

Elf 文件类型为 EXEC (可执行文件)
Entry point 0x1000a8
There are 3 program headers, starting at offset 52

程序头:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x01480 0x01480 R E 0x1000
  LOAD           0x003000 0x00103000 0x00103000 0x00120 0x01f00 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  段节...
   00     .text .rodata .eh_frame
   01     .got.plt .data .bss
   02
```

可以看到需要进行装载的为两个段，分别装至 0x100000 处 0x1480 字节，0x103000 处 0x120 字节,并将 0x103000+0x1f00 的其余位置置为 0

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）？

答:

步骤一：在 bootloader 文件夹中的 boot.c 里的 void bootMain(void) 函数中进行 OS 的加载

步骤二：在 kernel 文件夹中的 main.c 里的 void kEntry(void) 函数中进行一系列初始化
步骤三：在 app 文件夹中的 main.c 里的 int uEntry(void) 函数中进入用户空间进行输出

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）

答:

中断可分为 内中断 和 外中断
内中断又称为异常
外中断可分为可屏蔽中断和不可屏蔽中断
内中断中有包含了 自愿中断和强迫中断 ，软中断属于自愿中断，由 int 等指令产生

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。）

答:
外部中断重新设置到了 `0x20-0x2F` 号中断
IRQ：即中断请求，实际上就是中断请求信号线，对应了 8259A 芯片的一个引脚。若硬件设备想要向 CPU 发送中断信号，则必须申请一条可用的中断请求信号线，也就是申请一个 IRQ 号。一个 IRQ 号对应了一个中断号
中断号：中断号是系统分配给每个中断源的代号，以便识别和处理。

exercise7：请问上面用斜体标出的“一些特殊的原因”是什么？(EIP,CS,EFLAGS)

（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。）

答：
因为这三个寄存器，在跳转至中断程序后就会改变，所以必须在跳转前保存。而既然已经跳转，自然不会执行下面的指令，也就不能软件保存了。
而其他寄存器可以在跳到了中断处理程序后在进行保存，因为它们跳转后不会立即改变

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）

答：

如上所说，如果选择将信息存放至相同地方，那么当中断嵌套时，第二次中断会覆盖掉第一次中断存储的信息。例如 printf 是一次中断，将信息保存后跳至中断处理程序，结果需要打印的内容里面出现了除数为 0，那么更高优先级的中断会立即执行，将目前的信息保存至原先保存信息的位置，发生了覆盖。

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？

答：

因为当目标代码特权级更高时，CPU 才会进行硬件堆栈的切换，所以 ESP 和 SS 需要恢复。

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？

答：

可能会出现，因为原来六个寄存器的值会发生变化，导致后续可能出现不可预料的错误。

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）

答：

因为无论是软中断还是硬中断，它都有一个中断向量号，然后进而在 IDT 中找到对应的中断描述符，经过一系列检查和准备，进入中断处理程序。这样的设计更加方便。

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）

答：

因为如果不设置，esp 就默认为 0，而 esp 应该指向的时堆栈的顶部，所以这既有可能覆盖原来那个位置存储的信息，又会导致无法正常使用堆栈。

经过实验，如果把 esp 设置为 0x100000,程序将不能正确运行。如果设置为 0x120000,却可以正确运行。这是因为我们将 kernel 装载至了 0x100000,如果将 esp 设置为相同地址，会对 kernel 造成覆盖。而 0x120000 处却能正确

运行，因为我们可以查看 kernel 的程序头表发现只装载到了 0x100000~0x104f00 处，所以其余位置覆盖了不会造成影响。

当然，有关用户程序的装载也是同理

exercise13：上面那样写为什么错了？

答：

因为正确的加载步骤应该将程序头表中各表项中所描述的部分，加载在对应的位置。而上述代码是暴力的将第一个表项以后的所有东西全部一股脑地装载了过来，不管装载的地址是否正确、每个段需要分配多少空间等。

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？

答：

makefile 中有如下一句：

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

意思就是将 0x00100000 作为.text 节，然后指定了 kEntry 作为入口，然后生成了 elf 文件。我们在 boot.c 中读取了 elf 文件，并找到了程序入口的位置，并且让函数指针指向了它，通过这个函数指针调用了指向的函数 kMainEntry()，这实际上就是指向了程序入口，即为 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？

答：

都会跳转到 asmDoIrq。因为所有中断程序都需要进行一些相同的操作，即保护现场，然后进行跳转处理后恢复现场。

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.....）

答：

是在传递栈顶指针作为 Irqhandle 函数的参数。

因为 irqhandle 需要的参数是一个 TrapFrame 类型的指针，而 push esp 之前刚好 push 了 TrapFrame 结构里的数据结构，即 esp 等等寄存器以及 irq 号，此时的 esp 便相当于这个指针。

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？）

答：

会显示出原来想要打印出来的序列的倒序。

堆栈会一直叠加叠加，直到松开后在缓慢逆序退栈。

exercise18：阅读代码后回答，用户程序在内存中占多少空间？

答：

从代码中可获知用户进程以下消息：

Base:0x00200000

Limit:0x000fffff

故占据了 1MB 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）

答：

sel 是 用户数据段、特权级为用户级(3)的 段选择子。通过段选择子便可以在 GDT 中找到对应的段描述符，进而找到用户数据段，进而进行显存数据的修改以实现 print 功能。

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？

答：

因为按照 printf 传参顺序和函数调用的压栈顺序，paralist 应该比 format 先压栈，所以根据 format 的地址就可以得到逐个得到每个参数的地址。

2 的地址应该是(&format)+8。

遇到%就继续解析下一个字符，分别处理 x\c\d\s 的情况。

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？）。

答：

SYS_WRITE 是 write 的系统调用号，在进行系统调用的时候会作为参数传递给系统调用服务函数。

STD_OUT 的意思是将 buffer 写入到 STD 标准输出上。

Buffer 则是一个临时存放缓冲区，存储了需要输出的内容

Count 是需要输出内容的长度

exercise22：记得前面关于串口输出的地方也有 putChar 和 putStr 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。

答：

串口输出处的 putChar 和 putStr 是通过 outbyte 函数写到对应的 IO 端口以实现输出。

而此处的两个函数则是通过在用户态中调用了 getStr 或 getChar 函数，进而进行了相应的系统调用，然后转至了 sys_read,进而获取键盘输入的字符或字符串。

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"。

答：

因为对于用户程序本身而言，就是从 0x0 处开始的，再链接接后从 gdt 初始化函数如下，可以看到用户代码段是从 0x200000 开始，所以链接后对于该程序则会加上了 0x200000,相当于我们装载的位置

```
//init GDT and LDT
void initSeg() { // setup kernel segements
    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, 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);
    gdt[SEG_TSS] = SEG16(STS_T32A,      &tss, sizeof(TSS)-1, DPL_KERN);
    gdt[SEG_TSS].s = 0;
    setGdt(gdt, sizeof(gdt)); // gdt is set in bootloader, here reset gdt in kernel

    /*
     * 初始化TSS
     */
    tss.esp0 = 0x1fffff;
    tss.ss0 = KSEL(SEG_KDATA);
    asm volatile("ltr %%ax":: "a" (KSEL(SEG_TSS)));

    /*设置正确的段寄存器*/
    asm volatile("movw %%ax,%%ds":: "a" (KSEL(SEG_KDATA)));
    //asm volatile("movw %%ax,%%es":: "a" (KSEL(SEG_KDATA)));
    //asm volatile("movw %%ax,%%fs":: "a" (KSEL(SEG_KDATA)));
    //asm volatile("movw %%ax,%%gs":: "a" (KSEL(SEG_KDATA)));
    asm volatile("movw %%ax,%%ss":: "a" (KSEL(SEG_KDATA)));

    lLdt(0);
```

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）

答：我们可以先阅读 kernel 的程序头表。

```
程序头：
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x01480 0x01480 R E 0x1000
  LOAD           0x003000 0x00103000 0x00103000 0x00120 0x01f00 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```

可以发现只有两个段需要装载。错误代码可以正确的装载第一个段，但第二段则会直接被装载在紧跟着第一个段的位置。

而最终能够正确的执行程序，说明第二段装载在哪个位置不会对程序造成影响。

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）。

答：

(1)

① KeyboardHandle 函数首先从 IO 设备中读出输入的字符 code，然后根据具体是什么字符，对显存的对应位置进行修改，可看出显存起始位置应该在 0xb8000 处。

② 同理 syscallPrint 也是将字符存储在显存的对应位置

③ Syscallread 中的 getchar 和 getstr，顾名思义，则是从外部输入中读取字符或者字符串。当系统调用 syscall_read 之后，进入 getchar 或 getstr，然后进行一个开中断，然后从键盘中读取字符或字符串，然后关中断。其中 getchar 会将字符存入 eax 中作为返回值；而 getstr 则会根据回车来判断输入结束，然后将 keyBuffer 中的字符串拷贝到传递进来的指针 str 的位置，过程中有一个向用户数据段的切换。

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）

答：

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）

答：

首先 syscall 函数开始执行，进行 eax, ecx, edx, ebx, esi, edi 值的保存，然后将传进来的六个参数保存在上述寄存器中，然后通过 int0x80 指令陷入内核态。

根据中断向量号在 IDT 中找到对应的门描述符，进而进入到了 doirq.S 中的 irqSyscall，在进入 asmDoIrq 函数，然后将通用寄存器的值 push 到栈中，然后将 Esp 指针作为参数跳转至了 irqHandle 中，。

根据中断向量号进入到了 systemWrite 函数和 syscallPrint 函数，完成了 print 的执行，最后返回到 syscall 函数，保存返回值，恢复现场，切换回用户态。

硬件主要进行保护现场和加载至中断服务程序，以及从中断服务程序中恢复执行被暂停的程序。

而软件则负责根据中断向量进入对应的中断服务程序，以及程序的执行

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。

答：

- CPU 执行指令过程中产生常规保护错误(#GP)，根据中断向量号在 IDT 中找到对应的门描述符，进而进入到了 doirq.S 中的 irqGProtectFault 当中，首先 push 中断向量 0xd,再进入 asmDoIrq 函数，在之中调用 pusha 将通用寄存器的值全部压入 kernel 的栈中，然后将 Esp 指针作为参数跳转至了 irqHandle 中，根据中断向量号进入 GProtectFaultHandle 函数，程序终止。

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。

答：根据手册，在进行系统调用的时候，会有一个找到对应中断描述符然后检查 CPL 和 DPL 的过程，但我找了很久都未能找到这个过程。另外有关 TR 寄存器,TSS 的一系列操作，以及压入 SS 和 ESP 的步骤，修改 IF 位。。。都未能找到，只看到直接跳转到了 irqHandle 中进行中断处理程序了，所以很迷惑。

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）。

答：