1. exercises

- ## exercise 1

  在读取数据的过程中，盘面会不断转动，磁头在盘面上的位置会从一个扇区的起始位置逐渐转移到该扇区的末尾。如果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，那么在读取下一个扇区的数据时，又需要转动盘片使得磁头再回到扇区的起始位置，这会导致读取磁盘数据的时间开销增加。

- ## exercise 2

  逻辑扇区号 = C * 磁头数 * 每个磁道的扇区数 + H * 每个磁道的扇区数 + S - 1

- ## exercise 3

该可执行文件只含有一个可装入的.text 段，该段的起始位置位于该文件的开头，在文件和内存中均占 0x84 个字节，应将其装入物理地址从 0x400000 处开始的内存中去。且该程序的入口地址为 0x40078.

- ## exercise 4

  - 步骤一：在 bootloader/start.S 和 bootloader/boot.c
  - 步骤二：
    - initSerial():kernel/kernel/serial.c
    - initIdt():kernel/kernel/idt.c
    - initIntr():kernel/kernel/i8259.c

        - initSeg():kernel/kernel/kvm.c
        - initVga():kernel/kernel/vga.c
        - initKeyTable():kernel/kernel/keyboard.c
        - loadUMain():kernel/kernel/kvm.c
    - 步骤三：在 kernel/main.c 中进入用户空间；各种库函数的实现位于 lib/syscall.c 中

- # exercise 5

    0. 内部中断是不可屏蔽的，外部中断可以屏蔽
    1. 外部中断是由外设引起的中断，而内部中断是在执行某条指令的过程中所发生的中断，可以是由 INT 指令引起的软中断，也可能是当前指令执行发生了异常（如缺页，除零错误等）而引起的中断

- # exercise 6

    - IRQ 是中断控制器的引脚，每个引脚对应一个外部设备
    - 中断号是由 IRQ+32 得到的，它与硬件无关，而且可以更改。每个中断号对应一个中断向量。

- # exercise 7

  因为在程序执行过程中，除了自陷，程序并不知道什么时候会发生中断。而且对于外部中断而言，中断发生时，由硬件 CPU 查询 idt 表然后跳转到异常处理程序或中断服务程序执行，这会导致 cs,eip 和 eflags 的变化，在这之前需要保存当前正在执行程序的 cs,eip 和 eflags，而这个过程软件无法参与，所以只能由硬件保存。

- # exercise 8

如果 CPU 在执行用户程序的过程中发生了中断，该用户程序的现场信息会被保存在某一位置，然后 cpu 跳转到相应的中断处理程序去进行中断处理，在允许中断嵌套的情况下，如果在当前中断处理过程中发生了优先级更高的中断，那么 cpu 就需要保存当前执行程序的现场信息然后转去响应更高级的中断，此时之前保存的用户程序的现场信息就会被覆盖掉。

- ## exercise 9

因为当前代码段的特权级和要返回到的代码段的特权级不属于同一特权级，因而要进行堆栈切换

- ## exercise 10

会。在进行系统调用时通用寄存器会用来进行某些参数的传递，如果不先进行保存，那么原来程序执行过程中的通用寄存器中的值就会丢失。

- ## exercise 11

因为软中断和硬中断的中断处理处理程序结构相似，

- ## exercise 12

设置栈的起始位置。不设置会导致栈的起始位置随意，可能位于内核代码区。设置为 0x1f0000 也可以。由于栈是由低地址向高地址增长的，内核所占内存不超过 512*200 = 0x19000 个字节，最多在 0x100000-0x119000。

- ## exercise 13

不符合对齐要求，第一个段的装入位置是正确的，之后的段紧接着前一个段装入内存，但这很可能和程序头表中要求的装入内存的位置不同。

- ## exercise 14

这条指令中“-e kEntry"把 kEntry 设置为程序的入口地址,在 boot.c 中根据 elf 头获得程序的入口地址（即 kEntry 函数的起始地址）然后赋值给 kMainEntry,所以二者是等价的。kMainEntry 函数就是跳转到 kernel 程序的入口地址。

- ## exercise 15

  所有的程序最后都会跳转到 asmDoIrq。因为中断处理程序的结构类似，可以划分为三个阶段：

  - 准备阶段：保存现场信息
  - 处理阶段：跳转到对应的中断服务例程处执行
  - 恢复阶段：恢复通用寄存器的值，返回到用户态当前进程的逻辑控制流的断点处继续执行

- ## exercise 16

  为后续调用 irqHandle 提供参数。在调用 irqHandle 之前 push esp，

  相当于把 TrapFrame 结构体的首地址作为参数传递给 irqHandle,因而后续可以根据该地址得到相应的中断号。

- ## exercise 17

  允许键盘中断嵌套可能导致输出到屏幕上的字符顺序与实际不符，以及如果连续快速按键，cpu 还未处理完当前中断，便要去响应新的中断，需要不停地保存现场，会不断消耗栈空间，很容易导致栈满。

- ## exercise 18

  用户程序在内存中占据的空间为 2MB.

- # exercise 19

  由于内核空间和用户空间的段基址不一样，用 sel 用于存储用户数据段的段选择子，用来得到所要输出字符串所在段的基地址。

- # exercise 20

  - printf 可以有多个参数，所有参数按顺序存放在内存中，第一个参数为 format，将 paraList 设为&format，就可以通过 paraList 来访问 printf 的所有参数，paraList[0]是 format, paraList[1]..... 为依次的剩余参数。
  - 数字 2 地址为&paraList[2]
  - 遇到%时，将 state 置位 1，然后读取%后面的一位字符，根据该字符的值来调用不同的辅助函数，之后再把 state 设为 0.

- # exercise 21

  所有的参数都通过通用寄存器传递

  - SYS_WRITE 是系统调用号，保存在 eax 寄存器中
  - STD_OUT 是标准输出设备的文件描述符，表示输出到标准设备上，该值保存在 ecx 寄存器中
  - buffer 表示要输出内容存放位置的首地址，保存在 edx 寄存器中
  - count 表示要输出的字符的个数，保存在 ebx 寄存器中
  - 剩下两个参数对于 sys_write 调用没有意义

- # exercise 22

  lib/syscall.c 中的 getChar 和 getStr 是对系统调用的封装，在用户空间中调用来获取键盘输入的字符，而最终。

kernel/kernel/serial.c 在内核空间中使用。

- ## exercise 23

  保护模式下，物理地址由段基址+段内偏移量确定，用户代码段的基地址为 0x200000，加上.text 起始位置相对于段的偏移量 0，用户代码实际被加载到物理地址为 0x200000 的位置处。

2. Challenges

- ## challenge 1

  错误代码的问题是从第二段开始，加载到内存的位置不正确。通过查看 kMain.elf 的程序得知，该程序需要加载的有两段，其中第二段主要是.bss 和.data，用于存放全局变量或静态变量，由于 kernel 程序中没有全局变量和静态变量，因此第二段并不影响程序的执行，所以该错误代码可以正确执行。

- ## challenge3

  视频见打包文件。

3. Conclusions

- ## conclusion 1

  0. 保存通用寄存器的值，将参数放入通用寄存器中，然后通过 int 指令从用户态进入内核态

1. cpu 根据中断号查询 IDT 表跳转到对应的中断处理程序 irqSyscall（在这之前需要保存原来 cs,eip 等寄存器的值）
2. 将中断号和各通用寄存器压栈，然后根据中断号选择对应的中断服务例程 syscallHandle 执行
3. 执行结束后，恢复通用寄存器的值并从内核态返回用户态
4. 在用户态，将之前保存的通用寄存器的值恢复

- ## conclusion 2

0. cpu 根据中断号查询 idt 表跳转到 irqGProtectFault 执行
1. 将中断号和各通用寄存器压栈，根据中断号选择对应的中断服务例程 GProtectFaultHandle 执行
2. 通过 assert 结束原用户程序的运行

- ## conclusion 3

疑惑：

  - 在处理异常（如除零等）时也是由硬件来查询 idt 表的吗