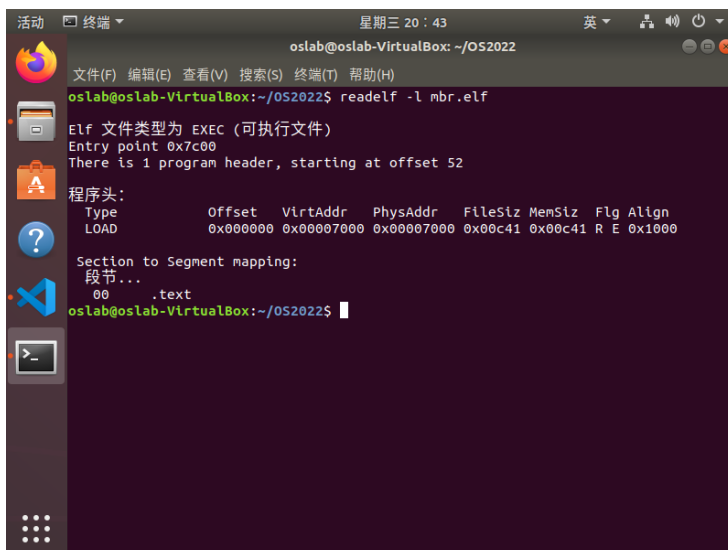


- 一. exercise1: 既然说确定磁头更快(电信号), 那么为什么不把连续的信息存在同一柱面号同一扇区号的连续的盘面上呢? (Hint: 别忘了在读取的过程中盘面是转动的)
- 因为在确定柱面和扇区的起始位置之后, 读取的过程中盘面在转动, 等到读完这个扇区之后, 磁头正好对到下一个扇区开始的地方, 如果这个时候下一个信息就在下一个扇区, 我们就可以不用让盘片进行旋转, 让磁头对准对应的扇区起始位置了。而如果我们把连续的信息存在同一柱面号同一扇区号的连续的盘面上, 那么这时候我们还需要将盘转回去, 并且改变磁头号, 这样浪费的时间更多。
- 二. exercise2: 假设 CHS 表示法中柱面号是 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)。
- 设 PS 表示每磁道有多少个扇区, PH 表示每柱面有多少个磁道;
- 则  $LBA = (C - 0) * PS * PH + (H - 0) * PS + (S - 1)$
- 三. 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)。



```
oslab@oslab-VirtualBox: ~/OS2022
oslab@oslab-VirtualBox:~/OS2022$ readelf -l mbr.elf
Elf 文件类型为 EXEC (可执行文件)
Entry point 0x7c00
There is 1 program header, starting at offset 52

程序头:
Type      Offset  VirtAddr  PhysAddr  FileSiz  MemSiz  Flg  Align
LOAD      0x000000 0x00007000 0x00007000 0x00c41  0x00c41  R E  0x1000

Section to Segment mapping:
段节...
00    .text
```

我这里观察的是 lab1 中的 mbr.elf 的程序头表,

可以看到只有一个段: 他的类型是需要装载的; 他相对于 ELF 起始位置的偏移量是 0; 应当被加载到 0x7000 的物理内存中; 文件大小和在内存中占据空间大小是 0xc41

- 四. 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)?
- 2.i kernel/kernel/serial.c
  - 2.ii kernel/kernel/idt.c
  - 2.iii kernel/kernel/i8259.c
  - 2.iv kernel/kernel/kvm.c
  - 2.v kernel/kernel/vga.c
  - 2.vi kernel/kernel/keyboard.c
  - 2.vii kernel/kernel/kvm.c
  - 3.i kernel/kernel/kvm.c

- 五. 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)

软中断即指令中断，由软件（int 0x80）主动执行系统调用发起；

硬中断由硬件发起，它可以被分为外中断和内中断。

外部中断包括外设请求和人为干预，包括时钟中断、键盘中断等等；（其中有些可以被屏蔽，有些不行）

内中断又称异常，是来自处理器内部的中断信号，通常由于在程序执行过程中给，发现与当前指令关联的，不正常的事件；例如除法错(#DE)，页错误(#PF)，常规保护错误(#GP)。

可屏蔽中断包括软中断和部分的外部中断；

内中断（即异常）是不可屏蔽中断

六． 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 号指的是选用 8259 芯片的哪个中断引脚，而中断号是指内存中断向量区中的中断类型（共 256 个）。

I/O 设备发出的 IRQ（属于）由 8259A 这个可编程中断控制器（PIC）统一处理，并转化为 8-Bits 中断向量由 INTR 引脚输入 CPU。

每个中断（前面提到了，包括外部中断，异常或软中断）都由一个向量来标识，Intel 称其为中断向量。而每个中断向量，都有一个 8bits 的向量号，8Bits 表示一共有 256 个中断向量，与 256 个中断向量对应，IDT 中存有 256 个表项，表项称为门描述符（Gate Descriptor），每个描述符占 8 个字节。

七． 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。）

因为如果不是硬件自动将他们保存到堆栈而是用一个比如 pusha 指令来保存的话，是可能改变这三个寄存器的值的（CS, Eip 和 Eflags），所以必须用硬件保存

八． 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）

如果这些寄存器的信息被保存在一个固定的地方，那么如果当一个优先级更高的中断到来的时候，他也会把那时候的寄存器信息保存在这个地方，当处理完这个优先级跟高的中断的时候，继续执行第一次的中断的时候，它保存的寄存器信息已经被覆盖了，会发生异常。

补充：思考一下（不必回答）：ring3 的堆栈在哪里？

（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）

因为从特权级低的状态转向特权级高的状态的时候，需要借助 TSS 来切换栈；但是从特权级高的状态返回特权级低的状态的时候，因为先前转移的时候已经在新的栈内保存过旧栈的位置，所以直接恢复现场就行了，不需要借助 TSS，因此没有 ring3 的信息。

九． 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？

（假设一开始是从用户态发生中断，到内核态处理）因为如果一开始 GDT[target\_CS].DPL < GDT[SS].DPL，那么就会将 ss 和 esp push 进入内核堆栈，保存信息，然后进行堆栈的转换；现在要恢复用户态的堆栈，就必须将原本 push 进去的 old\_SS 和 old\_Esp 给 pop 出来

十． 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？

是会的，因为从内核恢复之后的 iret 指令只恢复 EIP, CS, EFLAGS 寄存器，至于这六个

普通寄存器是不会恢复的。所以我们必须进行认为的保存与恢复，否则 cpu 恢复这个程序的执行之后，寄存器的值可能被内核程序修改，会产生不可预知的错误。

十一. 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）

软中断就是指令中断,相当于我在指令中直接给出中断号,然后到 IDT 中进行查找,找到中断处理函数之后执行(最后都跳转到 irq\_handle 去).这样设计的好处就是不用区分软硬中断了,直接都放在一个数组里,按照中断号去找 IDT 的对应表项就完事了,很方便.

十二. 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)

设置 esp, 就是设定刚开始的时候内核栈顶的值, 应该设定一个较大的值, 否则无法向下扩展, 栈空间会比较小。如果不设置的话整个 bootloader 都用不了栈了。(甚至因为后面也没有设定 esp, 所以连内核都没有栈空间了)

而且还需要合理规划内存空间, 不能与其他要用的地方重叠了(比如刚刚被 boot.c 加载进来的 elf 文件的位置)

十三. exercise13: 上面那样写为什么错了?

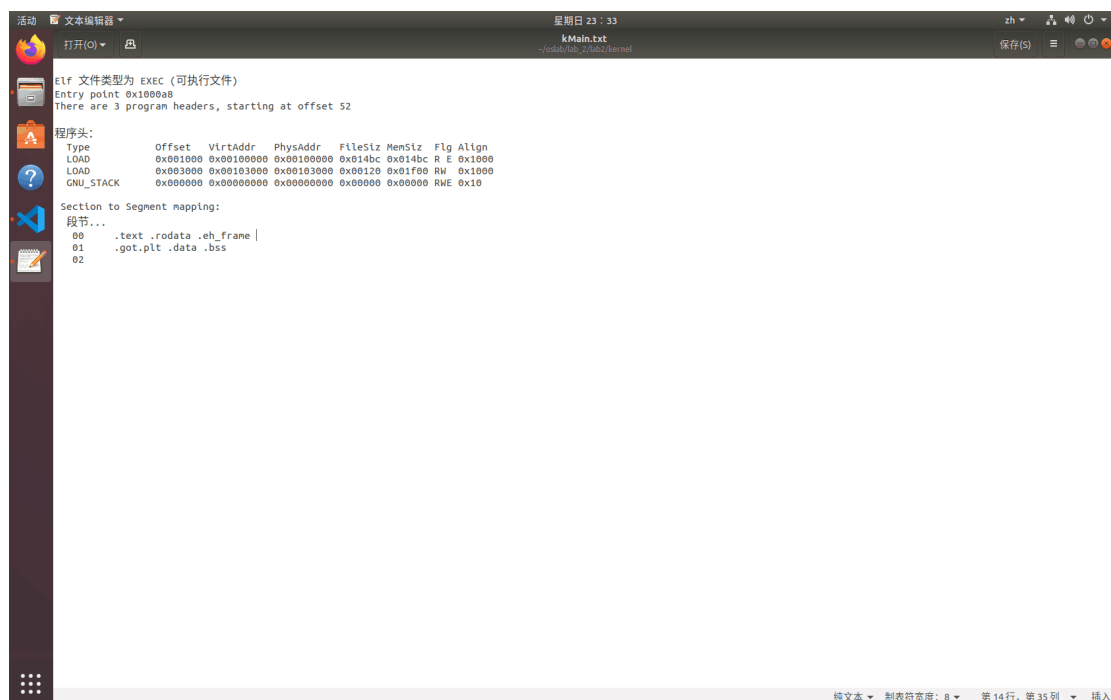
因为加载 kernel 的过程应该是把 kernel 的程序头表中的每个 type 为 Load 的表项装载到他的 elf 文件中指定的 paddr 的位置, 而这里并不是这样, 而是乱七八糟瞎搞。

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)

是因为它这边 elf+offset 指向了程序头表的第一个段, 他相当于将整个 elf 从第一个段开始全部搬到了 0x10000 的位置; 这样操作第一个段的装载位置是正确的 (makefile 指定第一个段的装载位置确实应该是 0x10000), 但是从第二个开始就可能出现问。

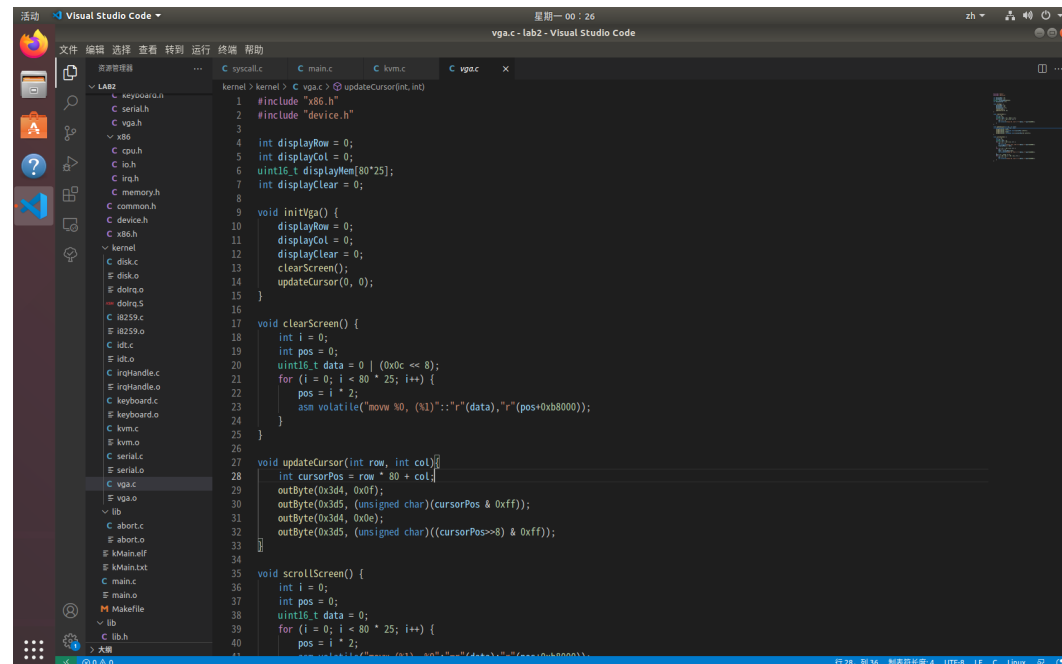
比如下图是我 kMain.elf 的程序头表:



从图中可以看出第二个段如果按照祖传代码, 是会被加载到 0x102000 去, 但是我们要他在

0x103000; 而且 kernel 中明显有全局变量, 那么为什么还可以正常运行呢?

答案就是我们所有的全局变量都是先写后用的: 意思是我们不管是.bss 节还是.data 节的全局变量, 我们在使用他们之前都会给他们赋值然后再使用他们。下图是一个例子。



```
1 #include "x86.h"
2 #include "device.h"
3
4 int displayRow = 0;
5 int displayCol = 0;
6 uint16_t displayMem[80*25];
7 int displayClear = 0;
8
9 void initVga() {
10     displayRow = 0;
11     displayCol = 0;
12     displayClear = 0;
13     clearScreen();
14     updateCursor(0, 0);
15 }
16
17 void clearScreen() {
18     int i = 0;
19     int pos = 0;
20     uint16_t data = 0 | (0x0c << 8);
21     for (i = 0; i < 80 * 25; i++) {
22         pos = i * 2;
23         asm volatile("movw %0, (%1)";::"r"(data),"r"(pos+0xb8000));
24     }
25 }
26
27 void updateCursor(int row, int col) {
28     int cursorPos = row * 80 + col;
29     outByte(0x3d4, 0x0f);
30     outByte(0x3d5, (unsigned char)(cursorPos & 0xff));
31     outByte(0x3d4, 0x0e);
32     outByte(0x3d5, (unsigned char)((cursorPos>>8) & 0xff));
33 }
34
35 void scrollScreen() {
36     int i = 0;
37     int pos = 0;
38     uint16_t data = 0;
39     for (i = 0; i < 80 * 25; i++) {
40         pos = i * 2;
```

在这种情况下, 我们的.data 和.bss 节被加载到了 0x102000 之后, 我们在对全局变量赋值的过程中, 会写到 0x103000 中的对应位置 (因为我们的程序不知道他被加载到了 0x102000), 在这种情况下, 我们再次使用这个全局变量的时候, 根据符号表到对应位置查看, 可以看到我们刚刚写入的值, 可以正常使用 (而且因为没有后续的段, 也不会发生重叠之类的问题), 所以可以正常运行。

十四. 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?

我认为这两个指的是一个函数; 在 makefile 中可以看到 -e 参数后面是 kEntry 标志, 意思是程序入口从 kEntry 函数开始执行; 因此我们的 elfheader 指向的 entry 的位置也就是我们的 kMainEntry 函数其实就是 main.c 中的 kEntry 函数。

十五. 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?

都会跳转到 asmDolrq 函数; 这样的话, 每个处理程序在跳转到这个函数之前只需要把各自的中断号 push 进栈, 剩下的步骤交给这个公共的函数就可以了

(补充: task3:

Selector 结构!

```

15                                     3   1 0
+-----+-----+-----+
|                                     |T|  |
|           INDEX                   | |RPL|
|                                     |I|  |
+-----+-----+-----+
TI - TABLE INDICATOR, 0 = GDT, 1 = LDT
RPL - REQUESTOR'S PRIVILEGE LEVEL

```

这里不管是 Interrupt gate 还是 trap gate RPL 应该都是 0，因为这里的 selector 最终是要跳转到 kernel 的代码段中去的，必须是 0，否则去 GDT 中找表项会有特权级的出错；

)

十六. 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.....)

因为我们查看它调用的 irqhandler 函数, 我们可以看到这个函数只有一个参数就是一个 Trapframe\* 的指针; 我们这里 push 的 esp 其实就是这个参数, 它指向的地址就是我们上一步 pusha 进去的各个寄存器的位置, 这就是 Trapframe\*

十七. 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?)

如果输入很快的话, 屏幕的输出与键盘的输入会呈现反向, 因为上一个键盘输入的中断还未被处理完成, 下一个键盘输入就已经进入了, 会引起下一个中断处理程序, 最终呈现的屏幕输出和键盘输入顺序相反。堆栈会不停累积, 因为每产生一个中断, 都会给 irq\_handle 传送一个 Trapframe, 这是每次中断处理程序的必备资源, 需要存放在内核堆栈中。如果这种信息过多 (即如果很多中断嵌套在一起), 内核堆栈可能溢出。

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

gdt[SEG\_UCODE] = SEG(STA\_X | STA\_R, 0x00200000, 0x000ffff, DPL\_USER); 从这句代码可以看出, 用户程序所占用空间为 0xffff

十九. 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)

通过观察这段代码, 我们可以发现首先这句话是将 sel 这个变量变成一个用户数据区的段选择符; asm volatile("movw %0, %%es::"m(sel)); 后面的这行代码意思就是将这个 16bit 的段选择符的值赋给 es 寄存器, 从而可以让 es 段就代表用户数据段; asm volatile("movb %%es:(%1), %0::"=r(character):"r"(str+i)); 因此这句代码的意思就是将用户数据段中偏移量为 str+i 的字符存入 character 变量, 即这个字符串的第 i 个字符

**challenge2:** 比较关键的几个函数:

KeyboardHandle 函数是处理键盘中断的函数

syscallPrint 函数是对应于“写”系统调用

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。

首先在 irq\_handle 函数中, 我们根据 trapframe 中 irq 号的不同调用不同的中断处理函

数。

如果中断号为 0x80，那么调用的是 syscallHandle 函数。这个函数会根据 eax 的值，调用 syscallwrite 或者是 syscallread 函数。(1) 如果 eax 是 0，那么调用 syscallwrite，如果 ecx 也是 0，那么会调用 syscallPrint 函数，在这个函数里面，挨个字符向显存(从 0xb8000 开始)的对应位置输出给定字符串的字符。(2) 如果 eax 是 1，那么调用 syscallread，分两种情况；如果 ecx 是 0，那么调用 syscallgetchar 函数，否则如果是 1，调用 syscallGetstr 函数。

(1) KeyboardHandle 函数

如果是正常字符，那么会调用 getchar 函数获取这个字符；然后将该字符输出到串口，并且存入 keybuffer 中，同时也会将这个字符写入显存中，并且调整输出之后的 displayRow 和 displayCol

如果是回车，就把 displayRow 加 1，displayCol 置为 0 就可以，然后写入 keybuffer 中

如果是退格符，就把 displayCol--，然后用 0 覆盖显存的当前字符。

(2) syscallPrint 函数

这个函数还是比较简单的，就是把给定字符串的每个字符输出到显存的对应位置，然后再调整输出之后的 displayRow 和 displayCol

(3) syscallGetChar 和 syscallGetStr 函数

syscallGetChar 函数是从 keybuffer 中接收到第一个非 0 字符，将其输出到串口上，并保存在 tf->eax 中。然后等待直到下一个字符进入 keybuffer

syscallGetStr 函数是首先等待，直到所有的字符(size 存在 ebx 中)全都已经输入 keybuffer 中之后，将这些输入的字符从 keybuffer 存入用户数据段的指定位置(就是给定的 str 指针开始的地方)最后填上一个'\0'字符表示字符串结束

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)。

二十. 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？

数字 2 的地址应该是 paraList+8; 所以我们设置 para = paraList + 4; 每当我遇到一个百分号，将 para 用作当前参数的地址，然后根据百分号后面字符的类型，将 para 加上 1 (%c) 或者 4(%s,%x,%d)

二十一. 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?)。

这行代码把 SYS\_Write 存入 eax，把 STD\_OUT 存入 ecx，把 buffer 存入 edx，把 count 存入 ebx，然后把 esi 和 edi 置为 0 之后开始执行 int 0x80。那么 irq\_handle 函数就会接收到一个 trapframe，然后调用 syscallhandle 函数，因为 eax 为 0，所以会调用 syscallwrite 函数，又因为 ecx 是 0，所以会调用 syscallPrint 函数，最终把字符串打印到显存上。

二十二. 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。

区别是这两个函数是通过系统调用实现的，最后会调用 syscallGetChar 和 syscallGetStr 函数，从 keybuffer 中读取数据；而串口输出的那两个函数则是直接使用了 outByte 的特权指



令.因此,这两个函数只有内核态才能使用,而 get 的两个函数在用户态也可以使用.

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)

(已完成,见视频)

二十三. 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"

因为在 gdt 初始化函数中,已经把用户代码段和用户数据段的 base 全都设置为了 0x200000,所以在用户态的视角下,这段代码其实是被加载到了 0x00000000 的位置上.因此我们在链接的时候使用的是-Ttext 0x00000000"参数(感觉像是程序头表的第一个需要被装载到内存的表项的 paddr 就是这里-Ttext 后面填上的地址,也就是 0;这也是为什么我们在 load 用户程序的时候需要在 paddr 后面额外加上 0x200000,转化为真正的物理地址)

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)

Printf 最后的指令是这一句代码: `syscall(SYS_WRITE, STDOUT, (uint32_t)buffer, (uint32_t)count, 0, 0);`

进入 syscall 函数之后,他会先保存 eax, ecx, edx, ebx, esi, edi 这几个寄存器到对应的局部变量里面,然后将传入的参数对应的赋值给这几个寄存器,然后执行 `int 0x80` 指令;机器接收到一个中断号为 0x80 的中断,这时候就会进入 IDT 表中查询对应的表项,找到了以后,因为是用用户态转到内核态,所以会借助 TSS 转变 SS 和 ESP 的值到内核栈;然后硬件会保存 CS, eip 和 EFLAGS,软件会保存通用寄存器到内核栈中,然后进行指令的跳转;执行对应的处理程序 `irqSyscall` 函数,这个函数最后会跳转到 `asmDoIrq` 这个函数中,然后会进行保存现场(pusha)操作,并且将 `trapframe` 作为参数,调用 `irq_handle` 函数处理对应的中断。在 `irq_handle` 函数中,查询到对应的中断号为 0x80,会调用 `syscallHandle` 函数,然后会根据一开始传入的寄存器的值(这个时候保存在 `trapframe` 中了)进行调用 `syscallWrite` 和 `syscallPrint` 函数,在现存中输出信息,全部处理完之后,返回 `asmDoIrq` 函数,恢复现场并且进行 `iret` 指令的执行,返回用户态(返回一开始调用的 `syscall` 函数)。返回之后会将 `eax` 存入 `ret`,当作返回值,然后恢复 `eax, ecx, edx, ebx, esi, edi` 这一系列寄存器的值,最后将 `ss` 段寄存器的值存入 `ds` 段寄存器的值,返回 `ret` 就可以了。(因为 `iret` 会 reset `ds` 寄存器的值,又因为是扁平模式,所以我们可以用 `ss` 寄存器的值给 `ds` 寄存器赋值,反正都是用户态了)

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。

当产生保护异常错误的时候,中断号为 13,进入 IDT 表进行查询找到对应的处理程序(注意这里 GP 异常因为不是 `int` 指令,所以相当于是硬中断中的内中断,而所有硬中断是不受 IDT 表项的 DPL 影响的,所以这里表项是 `KERNEL` 级别是没关系的)之后执行 `irqGProtectFault` 函数,因为该中断硬件会产生一个 `error code`,所以就不用我们自己手动 push 一个了,然后我们要做的就是 push 一个中断号进去,调用 `asmDoIrq` 函数即可,剩下的与上面的描述差不多,进入 `irq_handle` 函数,执行 `GProtectFaultHandle` 函数,最后 `assert(0)`,调用 `abort` 函数输出报错信息,并等待下一个中断的到来

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。

虽然问题很多但是后面都基本被解决了。比如函数的参数占用空间是 4 的倍数，这点我以前完全不知道。然后还有这里 elf 文件到底应该被加载到什么地方也是非常值得考虑的，比如 load app 的时候，如果我把 elf 置为 0x200000，把这个 elf 从磁盘上搬上来，然后同时我又把应该被 load 的段也 load 到这个地址开始的地方，那样就可能冲掉程序头表的信息，造成很奇怪的异常（我加载 kernel 的时候就出这样的问题）

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）