

## OSLAB2

191830064 姜纪文

注：未做 challenge。

**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）

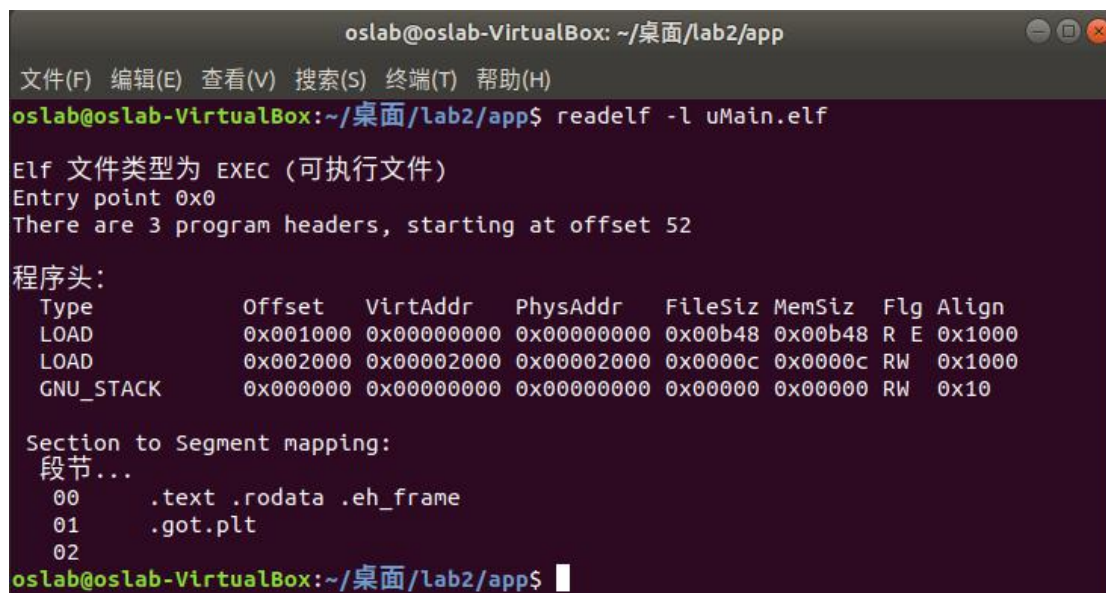
因为一次只能控制一个磁头进行读取，如果按题设存放所以我们必须控制所有磁头的工作以获得数据，如果只是同一盘面上，我们只需控制一个磁头

**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）

磁头数最大为 255 (用 8 个二进制位存储)，从 0 开始编号。柱面数最大为 1023(用 10 个二进制位存储)，从 0 开始编号。扇区数最大数 63(用 6 个二进制位存储)，从 1 始编号。LBA 从 0 开始编址。所以  $CHS \rightarrow LBA: (C * 255 + H) * 63 + S - 1$

**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）。

`readelf -l` 选项为读取程序头表，如下图有三个 segment，两个类型为 load，一个为类型 stack。我们加载可执行文件时就要依靠这些表项，offset 指明该段在文件的偏移，virtaddr 指明加载的虚拟地址，filesize 以及 memsize 指明数据代码等段，以及 bss 段大小。



```
oslab@oslab-VirtualBox: ~/桌面/lab2/app
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
oslab@oslab-VirtualBox:~/桌面/lab2/app$ readelf -l uMain.elf

Elf 文件类型为 EXEC (可执行文件)
Entry point 0x0
There are 3 program headers, starting at offset 52

程序头:
  Type           Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  LOAD           0x001000   0x00000000  0x00000000  0x00b48 0x00b48  R E 0x1000
  LOAD           0x002000   0x00002000  0x00002000  0x0000c 0x0000c  RW 0x1000
  GNU_STACK      0x000000   0x00000000  0x00000000  0x00000 0x00000  RW 0x10

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

**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，加载 os。即在 bootmain 中，对 os 加载。

步骤二，系统初始化。在 `kernel` 中的 `main.c` 中的 `kentry` 函数中。

步骤三，加载用户空间，库函数。`kentry` 最后一步，`loadumain` 函数为用户空间准备。用户函数在 `app` 中的 `main.c` 中，函数为 `uentry`。

**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）

可屏蔽中断与不可屏蔽中断：二者是从是否能被屏蔽的角度划分的，不可屏蔽中断与 `IF` 寄存器无关，是系统硬件上的特殊的保护性的中断，如果不处理则会导致系统崩溃。而可屏蔽中断则是与 `PIC` 相关的，一般是外设的中断。

下面三者则是从内外以及是否自愿来划分的中断类型。

外部中断：外部硬件产生的，是可以屏蔽的，一般是强迫的

软中断：指的是程序内部的自愿的指令中断

异常：执行指令时产生的错误，是强迫的中断

**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。）

不是一个东西，`IRQ` 是 `PIC` 中的引脚编号，而中断号则是对应内存中断向量表的索引。

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

首先软件是不知道中断的到来的，中断到来会通知 `cpu`，`cpu` 会改变执行流，那么 `cs:eip` 会指向所谓的软件来保存 `eip,cs` 等等，但是此时 `eip` 大概率不是上一个程序的下一个指令，即已经被改变许多了，而且这个软件大概率是无法知道被改变之前的 `eip,cs,eflags` 等全部信息的，即还未到软件执行，`eip` 已经与上一个程序下一条指令的地址相差很多了，那么在此过程中就必须要有硬件来支持保存上一个程序的状态，而此时所谓保存 `eip,cs` 等信息的软件也就没有必要了。

**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）

用户程序软中断时，在任务切换时，还未 `push old_eip,old_cs` 时发生优先级更高的中断，于是 `old_eip,old_cs` 被改变，并且 `push` 了，保存的是用户程序的状态，优先级更高的中断返回时是直接 `pop`，从而执行流回到用户程序而不是次一级的中断程序。

**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`？

因为发生了特权级不同的堆栈切换，所以结束前要恢复 `esp,ss`。

**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？

应该是不会发生不可恢复的错误，因为这些只是通用寄存器，如果这些寄存器错误会导致程序运行错误，有可能导致内核杀死该进程，进而也就没有错误了。

**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？

可能是为了中断程序以及接口的统一，便于管理。

**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)

设置内核栈, 不设置则会导致内核栈与用户栈的交叠, 容易错误, 可以设置成别的数值, 操作系统内核空间为 0x100000-0x200000, 所以可以选择别的数值, 例如 0x1ddddd。只有空间合适即可。

**exercise13:** 上面那样写为什么错了?

加载 segment 的时候会覆盖程序头表的内容。

**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?

kMainEntry 与 kEntry 的地址是一样的, kMainEntry 函数是 jmp 到 kEntry 的一个 c 函数。

**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?

最后通过 asmdoirq 跳转到 irqhandle 函数, 便于中断处理与中断二者的扩展, irqhandle 作为中间唯一的接口。

**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.....)

push esp 是将 esp 所指向的位置以 trapframe 结构解释与解析, 即 esp 传递该结构的地址。

**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?)

会先处理优先级高, 或者最近的中断, 例如连续键入 abc, 很有可能输出 cba。

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

我们加载的是  $200 \times 512 = 100\text{KB}$ , 用户程序占 100KB。用户空间是基址 0x200000, 界限 0xffff。

**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。

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

sel 变量即 selector, 框架中用于将 sel 的值赋给 es 寄存器, 为下一步显利用 es 作为选择子, 寻找 GDT 的表项, 进而以 es 为基址操作显存。

**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?

ParaList 设置为第一个参数字符串的地址, 用于解析栈上的压入的数据。

数字 2 的地址应该是 `(char*)&format + 2*sizeof(int)`;遇到%时, 就去解析后面一个字符作出相应的动作。

**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?)。

第一个参数为在 `lib.h` 中宏定义的常量 0, 具体在 `syscallHandle()` 函数中利用 `switch` 判别去执行内核的 `syscallWrite()`;

第二个参数同样是 `lib.h` 宏定义的常量 0, 具体用在 `syscallWrite()` 函数中去判别该项去执行内核的 `syscallPrint()`;

第三个是 `Buffer` 空间的地址

第四个是此时 `buffer` 的计数, 若满则执行写操作。

传递参数则是通过汇编将栈上的参数 `mov` 到相应的寄存器中的, 然后内核的中断处理程序的从相应寄存器读取参数。

**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。

串口是 `putChar`, `putStr` 两个函数, 区别在于串口函数输出到我们的终端上, `getChar` 是输出到 `qemu` 上的机器上。`putChar` 用于内核调试; `getChar` 用于用户程序, 向内核请求服务。

**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"`。

因为此时已经设置好了用户空间的 GDT, `cs` 为 `0x200000`, 那么用户空间的地址 `eip` 则是从 `0x00000000` 开始的, 即 `cs:eip` 寻址。

**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)

`Syscall`-->软件保存寄存器, 并把参数传入寄存器-->`int 0x80` 软中断->硬件保存 `eip`, `cs` -->找到相应的中断例程

-->`irqSyscall`-->`asmDolrq`-->`irqHandle`-->`syscallHandle`-->`syscallWrite`-->`syscallPrint`-->`scrollScreen&updateCursor`-->逐级返回直至 `asmIrq`-->`popal&iret`-->`syscall`-->返回值付给 `eax`, 恢复应用程序的通用寄存器值。

**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。

执行程序的时候, 当硬件发现例如权限不对等问题时发生保护错, 会根据错误类型转向中断

程序处理，中断程序会反馈给用户错误并且进行一定的处理，然后返回，硬件取出之前 **push** 的寄存器值，恢复。

**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。

疑惑是显存问题，显示设备是怎么利用显存显示的，每一次的刷新以及页面的上下浏览是如何操作的。