操作系统实验报告 - Lab2

学号： 201220139

姓名： 艾泠州

邮箱： 201220139@smail.nju.edu.cn

日期： 2022年4月4日

1. Exercises

1. 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：因为磁盘一次只能读取一个扇区的数据，但是在读取数据的过程中盘面也在转动，因此如果把连续的信息存在同一柱面号同一扇区号的连续盘面上，会导致磁盘在读取完一个扇区后，需要等待磁盘再旋转一周后才能读取下一个扇区，导致读取速度减慢；而把信息集中存在同一柱面上，可以连续读取多个扇区的数据，保证读取速度。

2. 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：$LBA = C \times N_H \times N_S + H \times N_S + S - 1$。

3. 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：如下：

```
$ readelf -l /usr/bin/gcc

Elf 文件类型为 EXEC (可执行文件)
Entry point 0x467de0
There are 10 program headers, starting at offset 64

程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040
0x0000000000400040
                 0x0000000000000230 0x0000000000000230  R      0x8
  INTERP         0x0000000000000270 0x0000000000400270
0x0000000000400270
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000
0x0000000000400000
                 0x00000000000fa8f4 0x00000000000fa8f4  R E    0x200000
  LOAD           0x00000000000fb5e8 0x00000000006fb5e8
0x00000000006fb5e8
                 0x0000000000003b38 0x0000000000006490  RW     0x200000
```

```
  DYNAMIC        0x00000000000fddd8 0x00000000006fddd8
0x00000000006fddd8
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x000000000000028c 0x000000000040028c
0x000000000040028c
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x00000000000fb5e8 0x00000000006fb5e8
0x00000000006fb5e8
                 0x0000000000000000 0x0000000000000010  R      0x8
  GNU_EH_FRAME   0x00000000000e44ec 0x00000000004e44ec
0x00000000004e44ec
                 0x0000000000003714 0x0000000000003714  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000
0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x00000000000fb5e8 0x00000000006fb5e8
0x00000000006fb5e8
                 0x0000000000002a18 0x0000000000002a18  R      0x1

 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 .stapsdt.base .eh_frame_hdr .eh_frame
.gcc_except_table
   03     .init_array .fini_array .data.rel.ro .dynamic .got .got.plt
.data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .tbss
   07     .eh_frame_hdr
   08
   09     .init_array .fini_array .data.rel.ro .dynamic .got
```

由程序头表可知，该程序共有 10 个段。

以 02 段为例，该段的类型为 LOAD，距离文件头（`Offset`）0 字节，装载到内存中的虚拟地址（`VirtAddr`）为 `0x400000`，物理地址（`PhysAddr`）为 `0x400000`，文件大小（`FileSiz`）为 1,026,292（0xFA8F4）字节，在内存中占据的大小（MemSiz）也是 1,026,292 字节。标签为可读（`R`）和可执行（`E`），按 2 MB（2,097,152（0x200000）字节）对齐。

4. 上面的三个步骤（加载OS部分，进行系统的各种初始化工作，进入用户空间进行输出）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：第一步：`/bootloader/boot.c`；

   第二步：`/kernel/main.c`；

   第三步：`/app/app.c` 和 `/lib/syscall.c`。

5. 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：可屏蔽中断：与标志寄存器上的 IF 标志位相关的屏蔽类型，受中断优先级影响，与「不可屏蔽中断」相对应；

不可屏蔽中断：与标志寄存器上的 IF 标志位无关的屏蔽类型，与「可屏蔽中断」相对应。

软中断、异常：由程序执行时内部引起的中断，其中软中断由 `int` 指令引起，异常则是由程序执行时的产生的异常引起。二者均为「内部中断」，与「外部中断」相对应。

外部中断：由外部设备引起的中断，与「内部中断」相对应。

6. 请问：IRQ 和中断号是一个东西吗？它们分别是什么？

   答：不是。IRQ（Interrupt Request，中断请求）是一种信号，由硬件或软件产生；而中断号则是中断向量表上各个中断的索引，与 IRQ 之间是映射关系。

7. 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。）

   > 在 IA-32 中，A 的状态中比较重要的有如下几个：
   >
   > 首先当是 `EIP`（instruction pointer）了，它指示了 A 在被打断的时候正在执行哪条指令；然后就是 `EFLAGS`（各种标志位）和 `CS`（代码段，里面存着 CPL）。由于一些特殊的原因，这三个寄存器的内容必须由硬件来保存。

   答：在软件运行的过程中，这三个寄存器中保存的值会不断发生变化（如每次执行一条新指令会使 EIP 的值增加，且可能使 EFLAGS 中的标志位发生变化）。如果用软件保存，可能会保存的值发生错误，进而导致返回应用程序时发生错误。

8. 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）

   > 如果希望支持中断嵌套（即在进行优先级低的中断处理的过程中，响应另一个优先级高的中断，比如在 `printf` 的时候，发生除零异常），那么堆栈将是保存信息的唯一选择。如果选择把信息保存在一个固定的地方，发生中断嵌套的时候，第一次中断保存的状态信息将会被优先级高的中断处理过程所覆盖！

   答：比如在 `printf` 的过程中发生除零异常，如果保存的现场固定在同一个地方，则 `printf` 所保存的信息会被除零异常保存的信息覆盖掉。但如果保存在堆栈中，则不会发生信息覆盖的情况。

9. 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?

   ```
   ################### iret ####################
   old_CS = CS
   pop EIP
   pop CS
   pop EFLAGS
   if (GDT[old_CS].DPL < GDT[CS].DPL)    //???
       pop ESP
       pop SS
   ```

   答：因为在中断到来时，只有在 `target_CPL < GDT[old_CS].DPL`（即发生了优先级变化）的情况下才进行了 `push` `old_SS` 和 `old_ESP` 的操作，而中断结束后也需要在对应的情况下 `pop` ESP 和 SS。

10. 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：会。因为在进行 `syscall` 操作之前，这些寄存器中保存的是原先程序的局部变量。如果不进行保存和恢复，会导致程序在使用这些局部变量时产生一些不可预料的值，进而产生意料之外的错误。

11. 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？

    答：因为 IDT 中保存着处理所有类型的中断所需的门描述符，而通过门描述符可以获得处理中断所需的所有信息。通过相似的过程，可以减少机器处理中断过程所占用的资源。

12. 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。

    答：为栈帧分配足够的空间。不设置会导致栈溢出或导致栈把 OS 内核代码覆盖。

    可以把 ESP 设置为其他值。

13. 上面那样写为什么错了？

    ```
    // TODO: 填写 kMainEntry、phoff、offset
    ELFHeader* eh = (ELFHeader*)elf;
    kMainEntry = (void(*)(void))(eh->entry);
    phoff = eh->phoff;
    ProgramHeader* ph = (ProgramHeader*)(elf + phoff);
    offset = ph->off;

    for (i = 0; i < 200 * 512; i++) {
        *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
    }

    kMainEntry();
    ```

    答：因为这样的操作只是简单地把整个 ELF 文件复制到了另一个位置。

14. 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：`bootloader/boot.c` 的作用是解析 ELF 文件，并使 `kMainEntry` 函数指针指向 ELF 文件的程序入口。而 Kernel 中 Makefile 的链接指令有 `-e kEntry` 的选项，即将 `kEntry` 函数设置为了程序的入口函数。故 `kMainEntry` 函数指针指向的地址即为 `kEntry` 的地址。

15. 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：`irqHandle` 函数。

16. 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：因为 `irqHandle` 函数的参数为 `TrapFrame*` 指针，而在 `push esp` 指令前分别进行了 `push IRQ` 和 `pusha` 操作，它们共同构成了一个 `TrapFrame` 结构体，且栈顶指针 `esp` 即为结构体的地址。`push esp` 操作相当于为 `irqHandle` 函数传参。

17. 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：如果快速地按下多个按键，会导致在前一个按键还尚未处理完毕就直接开始处理下一个按键，造成屏幕显示的按键顺序与实际操作不符；同时会导致堆栈叠加过多造成栈溢出。

18. 阅读代码后回答，用户程序在内存中占多少空间？

    答：100 KB。

19. 请看 `syscallPrint` 函数的第一行：

    ```
    1 int sel = USEL(SEG_UDATA);
    ```

    请说说 `sel` 变量有什么用。（请自行搜索）

    答：执行 `syscallPrint` 函数时，进程处于内核态，而此时 `DS` 段寄存器中存储着指向内核数据段的段选择子。而打印字符需要访问用户数据段的内容，其中 `sel` 变量保存的就是用户数据段索引的段选择子。因此需要通过 `sel` 变量来访问需要输出的内容。

20. `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：进入函数前，参数从右到左依次入栈，故左侧参数在低地址处，右侧参数在高地址处，且 `&format` 是所有参数的最低地址。初始值设置为 `&format` 有利于从左到右逐个访问各个参数。

    数字 2 的地址为 `paraList + 8`。

    将状态机设为状态 1，并解析下一个字符，找到对应的参数地址（`paraList + index * 4`），并通过提供的函数处理对应参数。

21. 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：

    ```
    1 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` 为系统调用号 `0`，`STD_OUT` 为 `stdout` 的文件号 `1`，`buffer` 为字符串存储地址，`count` 为输出字符的个数，后两个参数在 `syscallWrite` 中不使用。6 个参数分别通过 EAX、ECX、EDX、EBX、ESI、EDI 寄存器传递给系统调用函数。

22. 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：串口输出的函数用于将需要显示的字符或字符串输出到串口，并通过 `-serial` 参数使得串口数据能够输出到主机上，供 Kernel 使用。

此处的两个函数用于调用 `syscall` 来直接在 QEMU 的显存上进行输出操作，供用户程序使用。

23. 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：因为在装载用户程序时，系统已经进入了保护模式，故指令中的地址为逻辑地址而不是物理地址。由于在初始化 `gdt` 时，用户态代码段的 `base` 被初始化为了 `0x200000`，故代码段开头的逻辑地址为 0。

2. Tasks

1. 填写 `boot.c`，加载内核代码。
2. 完成 Kernel 的初始化。
3. 完成 IDT 相关初始化：完成 `setIntr` 和 `setTrap` 函数；为 `initIdt` 填空。
4. 填充中断处理程序，保证系统在发生中断是能合理处理。（`irqHandle` 通过相应的 IRQ 号跳转到对应的Handle 函数）
5. 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`；
   - 完成 `getStr`；
   - 完成 `printf`。

   提示：他们都要调用 `syscall` 函数。不要忘记 `getChar` 有返回值。
6. 填空：`kvm.c` 里面的 `loadUMain`。

3. Challenges

1. 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。）

```
// TODO: 填写 kMainEntry、phoff、offset
ELFHeader* eh = (ELFHeader*)elf;
kMainEntry = (void(*)(void))(eh->entry);
phoff = eh->phoff;
ProgramHeader* ph = (ProgramHeader*)(elf + phoff);
offset = ph->off;

for (i = 0; i < 200 * 512; i++) {
    *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
}

kMainEntry();
```

   答:

2. 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）。

3. 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` 。

4. Conclusions

1. 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：硬件：保存系统调用前通用寄存器的值；通过寄存器向系统调用函数传参；通过查 idt 表进入 `irqSyscall`，将异常号、中断号、寄存器的值、tf 首地址压入栈；接着跳转到 `irqHandle` 函数，进入内核态；之后跳转到 `syscallHandle`，根据通用寄存器中的值去调用其他对应的函数；最后恢复现场。

2. 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：当产生保护异常错误时，直至进入 `irqHandle` 之前，过程与 Conclusion 1 中的过程都相同。之后，`irqHandle` 函数会调用 `GProtectFaultHandle` 函数，之后直接 `assert(0)` 退出 QEMU。

3. 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：

4. 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：