211300097

实验要求

由于本次实验的要求在讲义中已经讲得很清楚了，所以我在此仅粗略地交代我的实现路径

1. 实现`bootloader`至`kernel`的跳转
2. 实现`kernel`的初始化
3. 实现`kernel`至`app`的跳转
4. 实现`printf`与`scanf`部分

实验情况

实验进度

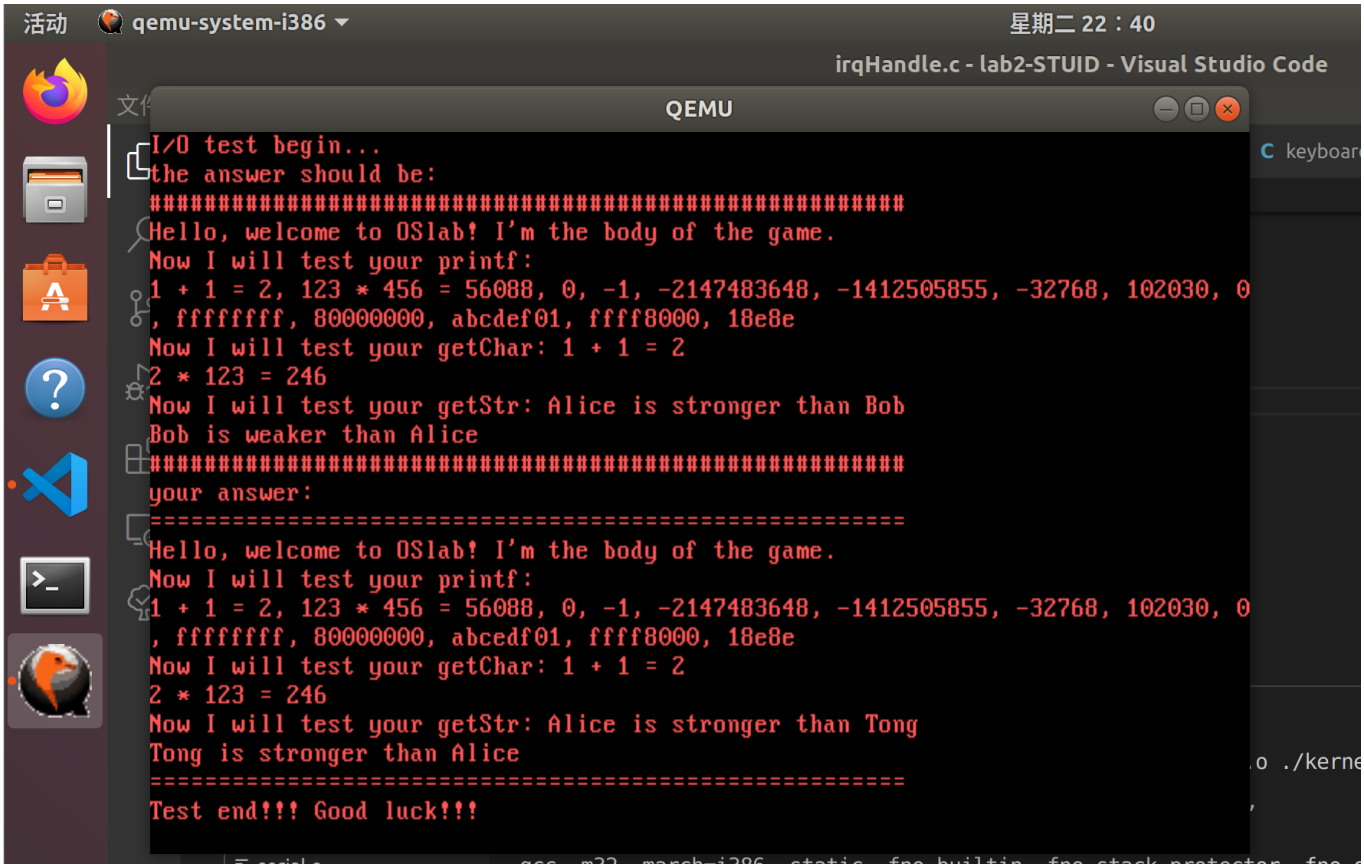
本人完成了此次实验的所有任务

代码修改处

除了框架中`TODO`字眼处，还修改了以下部分(具体修改下文提到)：

1. kvm.c - enterUserSpace
2. i8259.c - initInter

结果展示


活动
qemu-system-i386
星期二 22：40
irqHandle.c - lab2-STUID - Visual Studio Code
QEMU
I/O test begin...
the answer should be:
##
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
##
your answer:
==
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Tong
Tong is stronger than Alice
==
Test end!!! Good luck!!!

> Note
>
> 接下来的部分为:
> 1.实验心得 – 交代在实验中遇到的困难以及一些思考
> 2.(部分)思考题的回答

实验心得

Layer1

> 实现`bootloader`至`kernel`的跳转

这一部分的完成可谓一波三折: 本人在最开始使用的WSL系统, 后又辗转到linux**真机**上, 但都无法成功的完成这一部分.
起初怀疑故障是由于生成可执行文件时使用`objcopy`除去了除`.text段`外的其它段, 但联系理论知识很快就排除了这种可能性: `kernel`部分只存在局部变量, 也就是数据都保存在栈中, 并没有使用到数据段.

之后又怀疑是编译时`-O2`的选项使得编译出的文件没法正确跳转, 但取消优化后发现寸步难行: 只有进行优化后的可执行文件大小才能符合**512字节**的大小限制.

再后来又通过`objdump`和`readelf`察看反汇编信息和解析elf, 这让我察觉到了反汇编结果与实际想要实现的效果并不一致, 这让开始怀疑是版本带来的问题. 这之后由于暂时没有什么头绪,所以暂时将实验搁置了一星期左右.

> Cite
>
> 这里要感谢尤同学在群里的提醒, 让我发现版本的问题. 最终更换到提供的虚拟环境中得以成功实现了跳转到内核的功能.

Layer2

> 实现`kernel`的初始化

这一部分主要难度在于要理解各种操作系统中的机制和数据组织的方式, 所以这一部分绝大多数时间花在恶补提供的讲义和演示ppt之中.
在代码实现方面来说, 由于框架代码中已经提供了许多有效的代码, 大大降低了实现的困难程度, 并且这些代码也减轻了一部分理解的负担(还可以询问chatgpt代码的涵义).

这一部分个人觉得最难以把握的其实是硬件与软件之间的配合关系.
这一部分是由于上学期ics课的pa大作业使用的是`riscv`的架构, 和oslab中使用的`x86`不太一样, 前者是RISC, 倾向于让软件完成更多的事情; 而后者是CISC, 倾向于硬件来完成大多数任务.

就拿软中断时寄存器的切换来说，这些在pa大作业中是一段编写好的汇编代码来完成的，而在此处可以发现从`TSS`中取出和放回`esp/ss`其实是硬件就帮助完成了．所以在这一部分许多时间都在纠结：这个任务应不应该写一段代码来完成，还是已经又硬件完成了？
我解决这部分疑惑的方式是运行一下代码(实验)或是查阅手册寻找其中符合什么规约．

在这一部分我还遇到了一些坑和修改了一部分的代码，这些在接下来的部分中会提到．

Layer3

> 实现`kernel`至`app`的跳转

这部分代码量并不是很多，并且几乎可以完全copy第一部分的代码，不过由于在第二部分给自己埋了一个坑，debug花了不少实现．

在实现了这部分之后，qemu就会直接崩溃．通过二分的方式发现问题出在`iret`之后，也就是跳转过程中出现了问题．

最初怀疑问题出在了跳转目的地，也就是跳转到的位置不是正确的`entry`．所以怀疑是将`.text`段的开头设置为0x0的问题．经过仔细分析后发现由于设置了用户空间代码段为的`offset`为0x200000，在访问用户代码的时候跳转到的正是第一条指令的位置．

后来转变思路：由于在这一部分之前全是初始化的操作，所以问题应该是由于初始化过程的纰漏．
所以将目光落在了最有可能的两个地方：`idt`和`gdt`的设置．经过排查，发现是在初始化`idt`时`selector`段的参数有问题：`selector`应该包含`DPL`和`index`，而我在此处没有考虑到`DPL`．

经过以上分析，就可以发现问题是如何发生的了：`iret`指令需要切换回`ring3`的代码段，而此时并不能根据表中的`segment`段找到正确的段位置，无法跳转到用户程序．

也正是这一个不太起眼的小细节导致了问题的出现，并大大增加了本人debug的时长．这让我不得不感叹：**计算机里处处是细节，一定不能忽视"规范"的重要性!**

Layer4

> 实现`printf`与`scanf`部分

这一部分在我看来是本次实验中比较困难的一部分，涉及到了从`printf`到`sysPrint`/`getStr/getChar`到`syscallGetStr/syscallGetChar`逐层深入的实现和外部中断`KeyboardHandle`的实现．

- 实现`printf`：由于上学期已经涉及过这个内容，所以实现起来也算是轻车熟路．并且由于框架代码中已经提供了非常有用的`Hex2Str`等函数的接口，大大降低了实现者的工程量．之前实现的时候使用了`va_args`系列的宏定义，经过此次实验发现这部分过程不过是根据约定在栈上取参数，根据`paraList`变量逐个索引即可．
- 实现`syscallPrint`：这一部分比较多的是繁琐的细节如光标的维护/打印到显存等．框架代码中提供了最为重要的`asm volatile("movw %0, %%es" ::"m"(sel))`，提示了实现者如何完成内核部分向用户部分写入数据，同时也见识到了辅助段寄存器存在的重大意义．

- 实现`KeyboardHandle`：这个函数也涉及同上个函数一样的光标的维护等问题，在此不再赘述．还有另一处比较繁琐的细节是需要对不同的字符做出不同的处理，在此处使用了`keyboard.c`中定义的`keyBuffer`变量，来实现字符的存储，以便实现后面的`syscallWrite`系列函数 -- 对于**退格**情况，根据当前字符是否在`keyBuffer`中来判断是否可以进行退格，否则会造成由`printf`打印的字符也被无故退格；对于**可显示字符**来说，需要去查阅`keyboard.c`文件之中的定义，我们知道不可打印字符取出的值为`0`，会造成在打印的终止(`\0`被视为字符串的终止)，所以要防止不可打印字符进入`keyBuffer`之中．
- 实现`syscallWrite`系列函数：由于要协调键盘这个外部中断，这部分实现起来bug尤其多，~~经过多次的微调~~才最终实现．最后采用的方式是定义了全局变量`int volatile readOrnot = 0`，用来指示当前的键盘终端是否能够往`keyBuffer`中写入数据(由于并不是多线程，在此处使用一个全局变量来协调全局是可行的)．当进入`syscallGetChar/syscallGetStr`时则置为1，表示能够往写入；一旦遇见`\n`则置为0，表示当前不能写入．最后只要从`keyBuffer`中逐个取出键入的字符即可．**一个有趣的细节是定义`readOrnot`变量时一定要加上`volatile`关键字，否则被编译优化后会始终为0!!!**
- 一些修改：在实验过程中发现在跳转进入`app.c`之前，在`irqHandle`函数之中会跳转进入`default`分支并且触发`abort`，发现如果`IF`位关闭则不会出现这个问题．最后经过控制变量法发现为"Slave 8259A"部分引起的．查阅资料，最终使用`outByte(PORT_PIC_SLAVE + 1, 0xFF)`关闭了这一部分的中断，就可以正确跳转并且正确执行后续的代码了．

除此之外，在实现中还碰到了一些别的小细节上的纰漏例如在从指向字符串的指针取字符的的时候，使用了`*(*char)`，最后修改为`*(char**)`．在此其他的小错误就不一一列出了．

Miscellaneous

除了上述的主体实验部分，实验中还遇到了许多小插曲，其中耗费了主要精力的部分当属"get well with the Virtual Machine"(~~解决这些问题的耗时甚至超过了写代码的时间~~)

由于上学期使用的是linux真机完成的pa大作业，所以对于虚拟机的操作不甚了解(说来惭愧，对linux系统本身也不是特别的熟悉)，缺少很多配置虚拟机的经验．常常就是写代码写到一半就卡死了，或是频繁提示磁盘空间不足导致连文件都无法保存．

最开始由于对未知事物的恐惧，并没有尝试着去解决根本问题如重新分配cpu和磁盘大小，采用了关闭系统中不必要的软件和删除一些东西尽量腾出磁盘空间的方式，还是反复出现以上问题，严重影响了本人主体实验的进行．所以最后决定采取"制本"的方式来解决这些问题．在解决的过程中往往会牵扯出一些别的问题(例如对磁盘分区要用到一些之前未安装的工具，之后发现虚拟机无法联网，又牵扯出一些我很少了解的计算机网络方面的知识)，所以整个过程算不上轻松，也算是付出了之前没有认真关注过这些问题的代价．不禁感慨：**计算机处处是学问，由于各种原因落下的知识迟早要补上．**

思考题

> Ex1：计算机系统的中断机制在内核处理硬件外设的 I/O 这一过程中发挥了什么作用?

答：当一个外设需要向 CPU 发送数据或者请求处理时，它会通过相应的控制器向 CPU 发送一个中断请求信号，这个信号会中断 CPU 正在执行的程序，并且将控制权转移到操作系统内核的中断处理程序上.

中断处理程序会根据中断类型，查询中断向量表，跳转执行相应的处理程序. 在完成中断处理后，中断处理程序会将控制权返回给原来被中断的程序.

通过中断机制，计算机系统可以在外设请求处理时及时响应，并且在完成处理后立即返回到原来的程序，从而保证了系统的实时性和响应性能. 由此可见，中断机制对外设的处理是不可或缺的.

> Ex2：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?

答：从低优先级进入高优先级时，会将低优先级的SS和ESP值压入堆栈. 返回时，直接从堆栈中恢复即可，不需要特殊记录.

> Ex3：我们在使用eax，ecx，edx，ebx，esi，edi前将寄存器的值保存到了栈中，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. 首先，这些信息只能保存在不被多个进程共用的位置，那么理想的位置就是栈；其次，这些寄存器中保存了一个进程运行过程中必要的信息，倘如不进行保存和恢复，那么当此进程再次运行时，这些寄存器中存的将是无用信息(属于别的进程或者内核)，会产生难以预期的结果.

> Ex4：查阅相关资料，简要说明一下 %d ，%x ，%s ，%c 四种格式转换说明符的含义。

答：四者分别对应十进制数/十六进制数/字符串/字符的转换标志.

总结

总体来说，本次实验可谓是一波三折，在很多节点都遭遇了一些以前没有遇见过的困难，不过在最后也都一一解决了.有几个比较深刻的感受：一是计算机上处处是门道，学会掌握一种工具是不可或缺的能力；二是在完成一些稍微上规模的代码时，有各种debug的方式(即便是 printf)很重要；三是先完成后完美的原则(~~虽然本人的实现也远算不上完美~~)，先去实现基本的部件再去考虑更远的事情.

在此还需要感谢实验提供的讲义，pptx和框架代码，这些资料很大程度上降低了实现的困难，并且能够了解到一些作业之外的知识.