

Lab2 系统调用

201300032 肖思远

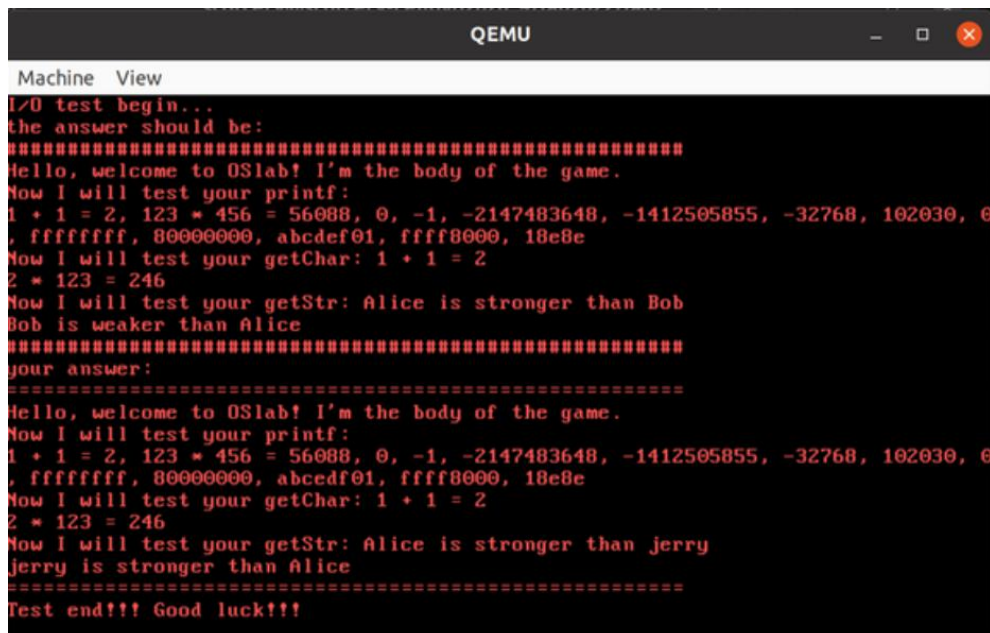
实验目的

1. 磁盘加载，即引入内核，bootloader 加载 kernel，由 kernel 加载用户程序
2. 开始区分内核态和用户态，完善中断机制
3. 通过实现用户态 I/O 函数介绍基于中断实现系统调用的全过程

实验进度

我完成了全部内容(但是有时会发生一些意外情况,将在后面实验结果分析讨论)

实验结果



```
Machine View
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 putchar: 1 + 1 = 2
2 * 123 = 246
Now I will test your gets: 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, abcdef01, ffff8000, 18e8e
Now I will test your putchar: 1 + 1 = 2
2 * 123 = 246
Now I will test your gets: Alice is stronger than jerry
jerry is stronger than Alice
=====
Test end!!! Good luck!!!
```

(注: gets 输入小写一切正常, 大写可能遇到 bug, 但是我没排查出问题)

关键代码实现方法

• 初始化中断门与陷阱门

```
11 static void setIntr(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl)
12 {
13     ptr->offset_15_0 = (uint16_t)offset;
14     ptr->segment = selector << 3; //TODO: check, which segment it is?
15     ptr->pad0 = 0;
16     ptr->type = INTERRUPT_GATE_32;
17     ptr->system = 0;
18     ptr->privilege_level = dpl;
19     ptr->present = 1;
20     ptr->offset_31_16 = (uint16_t)(offset >> 16);
21 }
22 /* 初始化一个陷阱门(trap gate) */
23 static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl)
24 {
25     ptr->offset_15_0 = (uint16_t)offset;
26     ptr->segment = selector << 3; //TODO: checked
27     ptr->pad0 = 0;
28     ptr->type = TRAP_GATE_32;
29     ptr->system = 0;
30     ptr->privilege_level = dpl;
31     ptr->present = 1;
32     ptr->offset_31_16 = (uint16_t)(offset >> 16);
33 }
```

• 填写 IDT 表项

```
61 setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
62 setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, DPL_KERN);
63 setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent, DPL_KERN);
64 setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault, DPL_KERN);
65 setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
66 setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
67 setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck, DPL_KERN);
68 setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException, DPL_KERN);
69
70 /* Exceptions with DPL = 3 */
71 // TODO: check some irq didn't use yet. 填好剩下的表项
72 setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
73 setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
74
```

• KeyboardHandle

```
142 void KeyboardHandle(struct TrapFrame *tf){
143     uint32_t code = getKeyCode();
144     if(code == 0xe){ // 退格符
145         if(displayCol > 0){
146             // TODO: check. 要求只能退格用户键盘输入的字符串, 且最多退到当行行首
147             if (displayCol && deleteBuf()) {
148                 --displayCol;
149                 updateCursor(displayRow, displayCol);
150                 printChar(' ');
151             }
152         }
153     }
154 }
155
```

```

156 else if(code == 0x1c){ // 回车符
157     // TODO: check:稍微有些暴力。处理回车情况
158     if(displayRow == 24) {
159         scrollScreen();
160     }
161     else {
162         displayRow++;
163     }
164
165     displayCol = 0;
166     insertBuf('\n');
167     putChar('\n');
168 }
169
170 else if(code < 0x81){ // 正常字符
171     // TODO: 注意输入的大小写的实现、不可打印字符的处理
172     putChar(getChar(code));
173     printChar(getChar(code));
174     insertBuf(getChar(code));
175     if (displayCol == 79) {
176         displayCol = 0;
177         if (displayRow == 24)
178             scrollScreen();
179     }
180     ++displayRow;
181 }
182 else {
183     ++displayCol;
184 }
185
186 }
187 updateCursor(displayRow, displayCol);
188 }

```

- syscallPrint 中主要代码

```

223 // TODO: check 完成光标的维护和打印到显存
224 if(character == '\n') {
225     displayRow++;
226     displayCol = 0;
227
228     if(displayRow == 25) {
229         displayRow=24;
230         displayCol=0;
231         scrollScreen();
232     }
233 }
234
235 else {
236     data = character | (0x0c << 8);
237     pos = (80 * displayRow + displayCol) * 2;
238     asm volatile("movw %0,(%1)":"r"(data),"r"(pos + 0xb8000));
239     displayCol++;
240     if(displayCol == 80) {
241         displayRow++;
242         displayCol = 0;
243         if(displayRow == 25) {
244             displayRow = 24;
245             displayCol = 0;
246             scrollScreen();
247         }
248     }
249 }
250 }
251
252 updateCursor(displayRow, displayCol);
253 }
254

```

- 用户库函数层面 printf 函数

```
90 // TODO: in lab2 check挺逆天的
91 if(format[i]=='%') {
92     i++;
93     count--;
94     paraList += sizeof(format);
95     switch(format[i]) {
96         case 'c':
97             mychar = *(char*)paraList;
98             buffer[count++] = mychar;
99             break;
100         case 'x':
101             myhex = *(uint32_t*)paraList;
102             count = hex2Str(myhex,buffer,(uint32_t)MAX_BUFFER_SIZE,count);
103             break;
104         case 's':
105             mystring = *(char**)paraList;
106             count = str2Str(mystring,buffer,(uint32_t)MAX_BUFFER_SIZE,count);
107             break;
108         case 'd':
109             mydec = *(int *)paraList;
110             count = dec2Str(mydec,buffer,(uint32_t)MAX_BUFFER_SIZE,count);
111             break;
112         case '%':
113             paraList -= sizeof(format);
114             count++;
115             break;
116     }
117 }
118 }
119 if(count == MAX_BUFFER_SIZE) {
120     syscall(SYS_WRITE,STD_OUT,(uint32_t)buffer,(uint32_t)MAX_BUFFER_SIZE,0,0);
121     count = 0;
122 }
123 i++;
124 }
125 }
126 if(count!=0)
127     syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
128 }
```

- os 层面中实现 getChar 的主体部分

```
50 char osGetChar() {
51     asm volatile("sti");
52     while(inputSize == 0 || inputBuf[inputSize-1] != '\n') {
53         waitForInterrupt();
54     }
55     /*while(inputSize == 0) {*/
56     /*waitForInterrupt();*/
57     /*}*/
58     log("out of ans\n");
59     asm volatile("cli");
60     char ans = inputBuf[0];
61     inputSize = 0;
62     return ans;
63 }
```

• os 层面中实现 getStr 的主体部分（灰色部分是我尝试按照分段，且内核和数据段基址不同的情况下的代码，后面会提及）

```
65 void osGetSTR(char *dst,int size) {
66     /* i think this will meet the address transition*/
67     /*int sel = USEL(SEG_UDATA);*/
68
69     asm volatile("sti");
70     while(inputSize == 0 || inputBuf[inputSize-1] != '\n') {
71         waitForInterrupt();
72     }
73     asm volatile("cli");
74     int i = 0;
75     /*asm volatile("movw %0,%es" ::"m"(sel));*/
76     /*char mychar = 0;*/
77     /*int addr = (int)dst;*/
78     for(i = 0; i<inputSize; i++) {
79         //TODO:check
80         dst[i] = inputBuf[i];
81         /*addr = (int)(dst+i);*/
82         /*mychar = inputBuf[i];*/
83         /*log("mychar is\n");*/
84         /*putChar(mychar);*/
85         /*asm volatile("movb %1,%es:(%0)\n\t"*/
86             /*:="r"(addr)*/
87             /*:="r"(mychar)*/
88             /*:);*/
89     }
90     /*addr = (int)(dst+inputSize);*/
91     /*asm volatile("movb $0,%es:(%0)"*/
92         /*:="r"(addr)*/
93         /*:);*/
94     dst[inputSize-1] = '\0';
95     log("inputBuf is\n");
96     log(inputBuf);
97
98     inputSize = 0;
99 }
```

实验思考题

1. 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？

中断机制发挥了帮助内核与硬件外设的 I/O 的交互，节约时间等作用（相比于轮询），且更加普适（相比于 DMA（个人认为））。以本次实验为例子，在这次实验中硬件外设 I/O 引起中断的主要有：keyboard 产生的中断。keyboard 的中断是当我们按下键盘时才产生，按照中断机制，由硬件接收到信号，按照中断向量表与中断号经过多步最终跳转到此中断的处理程序，然后执行处理程序获得 I/O 的信息，这样内核就完成了一次外设的处理。而且目前我们实现的单进程的 cpu，有了中断机制我们可以不用像轮询一样一直等待，从而节约时间，提高效率。

2. 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?

TSS 是用于从低特权级到高特权级的切换的, 只有这种情况下, 新堆栈才会从 TSS 中获取。ring3 是最低的特权级, 所以没有其堆栈信息。(那么从高特权级到低特权级有没有必要用 TSS 呢? 查了一下发现高特权级会将低特权级的 ss, esp 等压入栈, 所以也没有必要借助 TSS)

3. 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?

我觉得是有可能的。如果不保存和恢复, 那么等执行回来的时候各个寄存器里什么值理论上来说都是有可能出现的。万一原来的程序接下来要用某个寄存器的值作为除数, 然后在系统调用中这个寄存器被填入了 0, 那么接下来就出现了不可恢复的错误——除零错误了。

4. 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

- `%d`: 表示以十进制形式输出带符号整数
- `%x`: 以十六进制形式输出无符号整数
- `%s`: 输出字符串
- `%c`: 输出单个字符

实验结果分析

在最后的实验结果中, 我遇到了比较奇怪的情况。在最后 `getStr` 的实现中, 我出现了在输入大写字符的时候, 由可能会获取不到 `Str` 的情况, 但是小写字符却完全可以。我在阅读源代码的过程中, 认为大小写的区别应该只是取值不同, 出现这个结果我并不能理解。由于当我连上 `gdb` 后就无法向 `qemu` 中进行键盘 I/O, 我也难以观察到执行中寄存器的存储情况来分析原因。

实验思考

1. 关于段寄存器

这次我在两个地方遇到了段寄存器的问题。第一个地方是跳到用户程序的部分。这次实验中 `enterUserSpace` 将用户的段选择子等等压入栈中, 然后 `iret` 由硬件切换至用户态。但是我总是会跳到一片空白的区域。我通过 `gdb`

确定了当时的状态，从我目前学的理论上应该是段基址变为 0x200000，偏移量变为 0x0，就是正确的跳转（当然段基址是框架给的，我确认过是对的）。怀疑是不是我没有理解 entry，这个 entry 会不会是在内核态的用户代码的偏移量，我去查阅了 i386 的手册。最后确定了：手册上对于压入栈的参数处理中最后是将：EIP←Pop(),CS←Pop(),那确定是对于用户的偏移量了。最后没有找出原因，我还是选择了规避这个问题：修改 Makefile 中用户的.text 开始地址为 0x200000，然后修改填写的 GDT 表。虽然这个方式是 Linux 用的，也确实解决了这个问题，但是失去了分段的美感（这个保护模式开了跟没开类似的感觉，又没有用分页.....），其实还是挺难过的，感觉自己或许对于分段理解的还不够深入。

第二个问题是在写 getStr 中遇到的。我认为其他的参数（比如 SYS_READ 等等）是数值，可以直接用。但是 str 是一个用户态下的指针，其值是用户态下的地址。那么内核访问的时候，我认为必须要进行一些转化，就像 printf 中，将用户段选择子加载到某个数据段寄存器，然后再用内联汇编访问。（我查了下实际的操作系统想要访问用户空间要通过某些封装的接口，我们这么处理还是怪暴力的）。但是这样却传不到我想要的地址中。（很可惜开了 gdb

```
75     asm volatile("movw %0,%%es" :: "m"(sel));
76     char mychar = 0;
77     int addr = (int)dst;
78     for(i = 0; i<inputSize; i++) {
79         //TODO:check
80         addr = (int)(dst+i);
81         mychar = inputBuf[i];
82         log("mychar is\n");
83         putchar(mychar);
84         asm volatile("movb %1,%%es:(%0)\n\t"
85                     : "=r"(addr)
86                     : "r"(mychar)
87                     );
88     }
89     addr = (int)(dst+inputSize);
90     asm volatile("movb $0,%%es:(%0)"
91                 : "=r"(addr)
92                 :);
```

后 qemu 会无法输入，也就会卡死在上面的循环到不了这里，所以我没能通过 gdb 实时观察状态。）最后的解决方案是直接访问，因为我把内核和用户的段基址设置成是一样的，那么用户的线性地址就和内核的线性地址就可以了，那么就可以直接访问了。但是这样就像上面说的，我规避了这个问题，很暴力.....

2. 关于硬件与软件

刚开始我有一个疑问：那个 TSS 似乎花了很大篇幅来讲是如何执行的，但是在做的时候我似乎只填了个表，以及键盘我似乎也只是填了个表，但是我好像没有写到也没有看到调用的全过程？但是后来我慢慢地想明白了，这些都是硬件在执行的，而我写的只是一个软件，所以我只需要根据硬件的 API 进行相应的填写就够了。硬件和软件的区别其实在做实验中有很多的体现。比如说，我在写的时候遇到了不确定的问题，我需要思考查哪一本手册。像指令含义等等属于硬件，去查 i386 的手册往往就能查到。而比如填 GDT，IDT 等表，一般要查软件（IDT

硬件只能查到前面的不可屏蔽，后面的 intel 保留的给操作系统）。我常常经历看手册看得眼花缭乱，看完之后发现想要查的嘿！没有！（查错手册了 Q-Q）。在这种经历下对于硬件和软件的思考和区别就慢慢多起来了。（但是讲义我觉得比较难受的就是硬件和软件的区分不是很明显，两个都讲了但是没有强调于是我们真正要干什么就模糊了）（好吧其实有部分讲了但是我自动忽略了....就像 TSS 我自己感悟到硬件干的然后再回去看才发现通篇都写着硬件上下文切换）

3. 关于抽象层

这次的代码我花了很长时间理解，有时候会想我是谁我在哪我要干什么？有时候我会想是这个代码真的不太星吗？我觉得有些部分是我没有理解设计的用意。比如说 `bootMain` 加载 `kernel` 的时候，突然出现了一个 ELF 头，我记得我当时感到非常匪夷所思地去看 `makefile`，想为什么不直接弄成 `bin` 简单方便。但是之后学到 `jyy` 多次强调的一个概念：操作系统对于硬件来说其实也只是一个可执行文件，我就在想包括之后让我们仿照这个加载用户程序的时候是不是就是让我们感受到操作系统其实也就是个可执行文件这一点？（但我还是很好奇加载操作系统时真正机器怎么加载，准备以后去查查）。

言归正传（？），我在被代码折磨时也在思考为什么这个这么难读，我觉得这个代码其实就像上面说的很多地方不是写的不好，是我们没有理解。

（仅个人观点）我觉得可能大概也许我理解的有部分原因是抽象层。如果有 m 个类型硬件， n 个类型需求接口，这个代码我（仅个人水平下的个人观点 x ）感觉能写 $m*n$ 个操作系统（抽象好一些的可能只要写 $m+n$ 个）。似乎有些接口设计的有些直白，有些有些不够抽象（`printf` 和 `syscall` 虽然都是提供给用户的接口但是不太像一个抽象级，都放在了 `syscall.c` 文件里）。而我写的部分的抽象与封装也做的不好，我至今能想起我直接将用户的段基址和内核段基址都设为 0 的暴力做法。总的来说，还是很遗憾的。希望下一次实验我能做得更好！