

操作系统实验

Lab2:

内核及用户程序的加载、idt 的初始化、printf 参数处理、
中断处理程序的分发(irqHandle)、键盘中断、
getChar 及 getStr 实现

院系：人工智能学院

姓名：石睿

学号：211300024

班级：操作系统-2023 春季学期

邮箱：211300024@smail.nju.edu.cn

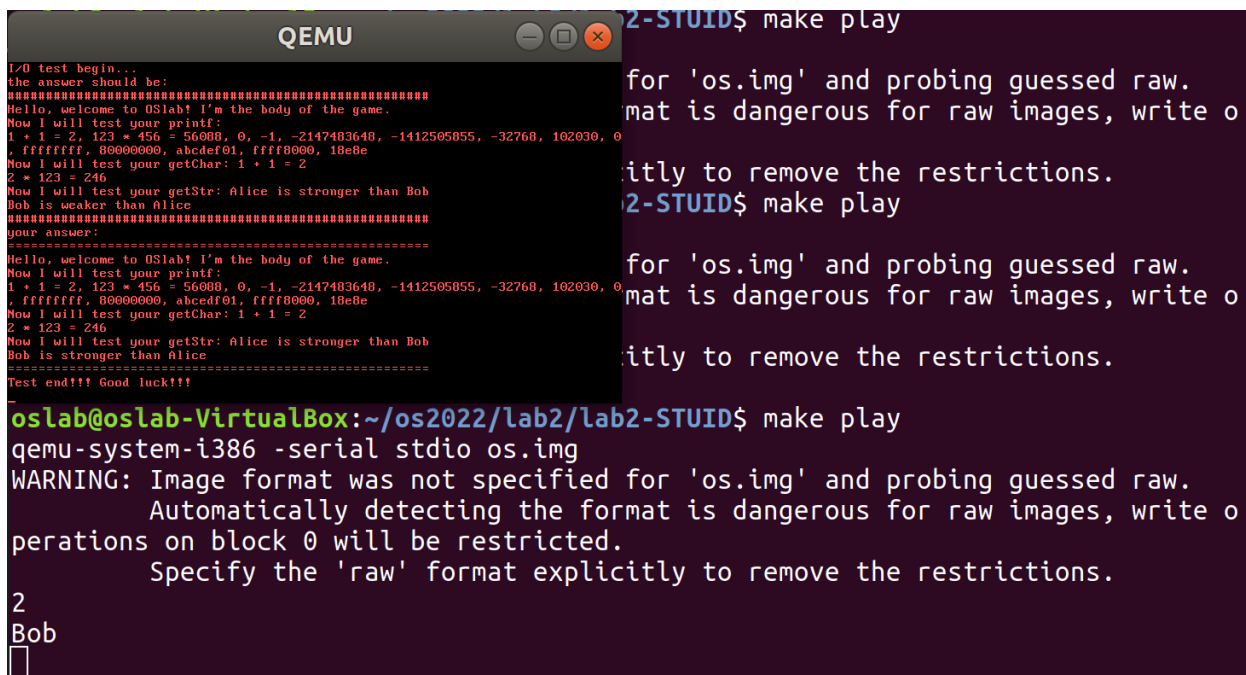
实验时间：2022.4.4

一、 实验进度

我已经完成了 Lab2 的所有内容。

二、 实验结果

结果如下图所示，具体实现过程在第三部分介绍。



```
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 = 56000, 0, -1, -2147483648, -1412505055, -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 = 56000, 0, -1, -2147483648, -1412505055, -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 stronger than Alice
Test end!!! Good luck!!!

2-STUID$ make play
for 'os.img' and probing guessed raw.
mat is dangerous for raw images, write o
itly to remove the restrictions.
2-STUID$ make play
for 'os.img' and probing guessed raw.
mat is dangerous for raw images, write o
itly to remove the restrictions.

oslab@oslab-VirtualBox:~/os2022/lab2/lab2-STUID$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
2
Bob
```

可检测的 Task1: printf 参数处理、系统调用、字符串的显存打印

可检测的 Tak2: getChar 和 getStr 实现 - 键盘中断、两者的系统调用

三、 实验修改的代码位置

① 内核的加载

正确装载 elf 文件以及内核的步骤有以下 3 步:

Step1: 通过 readSect 函数把 ELF 文件整体读入正确的固定位置

Step2: 找到 ELF 头、程序头表的位置, 并得知表项的数量

Step3: 通过程序头表中的 type 信息, 逐个判断当前段是不是需要被装入。

3.1 段的装入。如果当前段的 type 是 PL_load, 需要知道相较于 elf 起始位置的偏移量 ph->off、被加载到的物理内存中的位置 ph->paddr。

【具体实现】`*(unsigned char*)(ph->paddr+j) = *(unsigned char*)(elf+ph->off+j);`

3.2 bss 的初始化。 .bss 段在进程的虚拟存储映象中没有被分配空间，而在真实内存空间中是被分配空间的了。其大小是 `memsz-filesz`。

【具体实现】`*(unsigned char*)(ph->paddr+j) = (unsigned char)0;`

而本次框架代码中给出的代码未实现 step2 和 step3，没有根据程序头表中的内容进行装载，导致数据段的装载是错误的。

```
oslab@oslab-VirtualBox:~/os2022/lab2/lab2-STUID$ readelf -l kernel/kMain.elf
Elf 文件类型为 EXEC (可执行文件)
Entry point 0x1000a8
There are 3 program headers, starting at offset 52

程序头:
  Type           Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  LOAD           0x001000  0x00100000  0x00100000  0x0130c 0x0130c  R E  0x1000
  LOAD           0x003000  0x00103000  0x00103000  0x00120 0x01f00  RW  0x1000
  GNU_STACK      0x000000  0x00000000  0x00000000  0x00000 0x00000  RWE  0x10

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

本次实验编译后 elf 文件中程序头表的信息

```
48 void bootMain(void) {
49     unsigned int elf = 0x200000;
50     for (int i = 0; i < 200; i++) {
51         readSect((void*)(elf + i*512), 1+i);
52     }
53     // TODO: 填写 kMainEntry、phoff、offset, 加载 Kernel
54     struct ELFHeader *eh = (struct ELFHeader *)elf;
55     struct ProgramHeader *ph = (struct ProgramHeader *)elf + eh->phoff;
56     struct ProgramHeader *eph = (struct ProgramHeader *)ph + eh->phnum;
57     int j = 0;
58     for (; ph < eph; ph++){
59         if(ph->type == 1){ //同PA啦，只装载可以装载的段！
60             for(j = 0; j < ph->filesz; j++){
61                 *(unsigned char*)(ph->paddr+j) = *(unsigned char*)(elf+ph->off+j);
62             }
63             for(j < ph->memsz; j++){ //在上面可装入段所有内容完成装入后，内存中剩余的其它地方【.bss节】都初始化成0
64                 *(unsigned char*)(ph->paddr+j) = (unsigned char)0;
65             }
66         }
67     }
}
```

修正后我的实现如上

② 完善中断机制、提供系统服务函数（idt 的初始化）

本阶段需要做一些列的初始化：idt、8259a、gdt、tss、vga、keyboard device 的初始化，其中只需要补全 idt 初始化函数 `initIdt()` 即可。

限于篇幅，此处仅以陷阱门（Trap Gate）为例进行分析，具体 GateDescriptor 中的各个元素和陷阱门描述符的对应关系如下图所示。其中需要注意的是段选择子 selector，一共 16 位寄存器中保存段选择子的下标+第三位的特权级，而门描述符中只保存它的下标（idt 的什么位置）。所以在实现的时候需要把 selector 右移动 3 位后才可以赋值给 segment！

```

struct GateDescriptor {
    uint32_t offset_15_0 : 16;
    uint32_t segment : 16;
    uint32_t pad0 : 8;
    uint32_t type : 4;
    uint32_t system : 1;
    uint32_t privilege_level : 2;
    uint32_t present : 1;
    uint32_t offset_31_16 : 16;
};

```

TI - TABLE INDICATOR, 0 = GDT, 1 = LDT
RPL - REQUESTOR'S PRIVILEGE LEVEL

```

26  /* 初始化一个陷阱门(trap gate) */
27  static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {
28      // TODO: 初始化 trap gate
29      ptr->offset_15_0 = offset & 0xFFFF; // 类型是 uint32_t, 右移动高位补 0.
30      ptr->offset_31_16 = (offset >> 16) & 0xFFFF; // 通过与上一个长度更小的【显示地把】长度给截断-----自己写的，不加 0xffff
31      ptr->segment = selector << 3; // 自动截断，回顾 lab1 中的 gdt 表，选择子存入寄存器后，低三位是留给特权级的!!!
32      ptr->present = 0x1; // 在 memory.h 文件中，定义了宏来完成 selector 的处理呢！
33      ptr->privilege_level = dpl;
34      ptr->system = 0;
35      ptr->type = 0xF; // 也就是上面定义的 TRAP_GATE_32
36      ptr->pad0 = 0;
37  }

```

陷阱门 trap gate 的初始化如上所示

之后需要对 idt 表项中设置中断处理函数，首先要完成对特定中断向量的设置。

9.10 Error Code Summary

Table 9-7 summarizes the error information that is available with each exception.

Table 9-7. Error-Code Summary

Description	Interrupt Number	Error Code
Divide error	0	No
Debug exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

在查阅了 Intel 80386 手册后，找到了对于特定的中断向量的中断号，以及他们对应的出错码（它们的出错码会在 `doIrq.S` 中被压入栈中）。其次，完成本次实验中后续中断处理程序 `irqKeyboard`, `irqSyscall` 的们描述符的初始化。它们分别是陷阱门和中段门，实验手册中提到了它

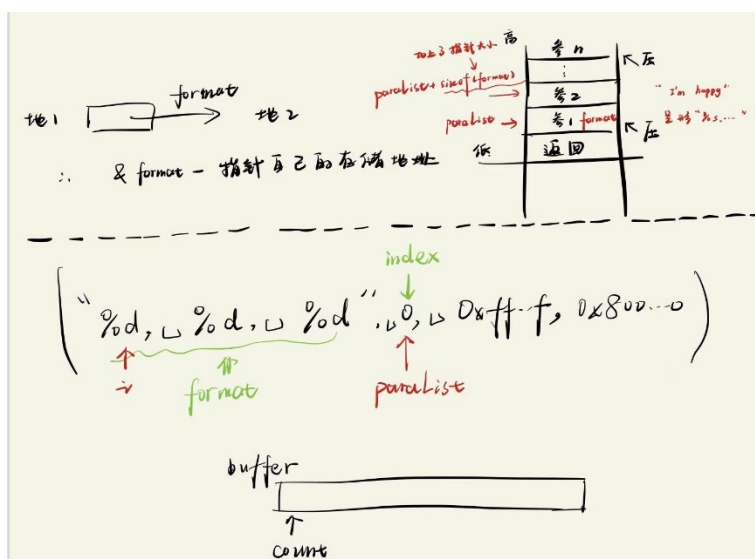
们的中断向量号分别为 0x21 和 0x80, 并且特权级 (DPL) 分别是内核态 (ring0) 和用户态 (ring3), 具体实现如下所示。

```
/* Exceptions with DPL = 3 */
// TODO: 填好剩下的表项
setTrap(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN); //irqKeyboard的调用号是0x21
setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER); //irqSyscall的调用号是0x80
//系统调用的 int 0x80, 中断向量是 0x80, 在用户态, ring3发出的中断 DPL_USER=3
```

③ 加载用户程序、printf 解析参数

加载用户程序和加载内核几乎完全相同, 只需要注意一下程序入口偏移量需要减去 0x200000.

Printf 解析参数和 PA 中实现的略有不同, 但大体一致。PA 中采用了 va_list, 是 C 语言中解决函数读入不定参数所定义的宏, 可以解决 printf 参数不定的问题。本 Lab 中除了采用了 C 语言调用惯例中实参从右向左逐步压栈的方式查找参数, 其他和 PA 中的实现相同, 代码中相关变量的图示如下。



④ 中断处理程序分发、键盘处理与打印到显存、getChar、getStr

中断处理程序的分发和 PA 中也类似, 在 irqHandle 根据中断向量号调用不同处理函数即可

键盘的处理根据读入的键盘码分别对读入是退格符、回车符、正常字符分三种情况进行解析。

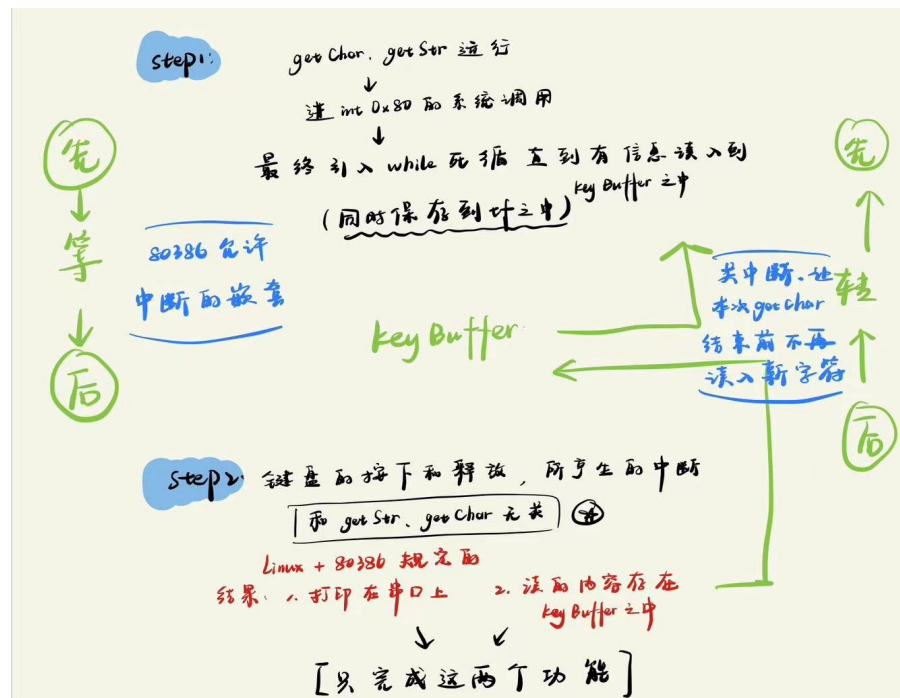
在解析键盘之前, 首先要明白 getChar 和键盘中断之间的关系。也就是 app/main.c 中 `char num = getChar();` 的处理流程 (此处仅仅以 getChar 为例)。

Step1: getChar 处理。syscall->int 0x80->irqHandle->syscallGetChar。此时如果键盘没有输入就一直等待, 直到有信息读入 keyBuffer 之中。这一步可以 work 的基础是 Intel 80386 支持中断的嵌套, 在 syscall (一个用户级 ring3 的系统调用) 中可以优先处理中间插入的键盘中断 (一

个内核级的中断)，并在键盘中断处理完成后返回继续处理 syscall!

Step2: 键盘中断。在 getChar 的 while(1)循环中敲击键盘，硬件产生了键盘中断，由于优先级会先处理这个键盘中断。在个人实现的时候，把键盘读入的信息打印到串口上，并且把读入的内容放到了 keyBuffer 之中，中断处理完毕，iret 返回。

Step1 和 step2 用流程图说明可能会更加清楚这个过程。



此外还有两点需要格外注意：

1、 putChar 只有在内核级（Kernel 文件夹下）中才可以使用（因为框架的声明）。

getChar、getStr 在用户级和内核级都可以使用，也就是在 app/main.c 中使用的。

2、 为了避免在输入完成的时候，由于在键盘按下到松开的时候有一段时间，会读入很多个相同按键的内容到 keyBuffer 之中。为了避免这个问题，根据两个函数完成输入状态的方法：getChar 以读入一个字符判完成，getStr 以读入\n 判完成，以及框架代码中提供的开中断函数 enableInterrupt()和 disableInterrupt()两个函数。以 getChar 为例，让在【没输入的时候】开中断->关中断->开中断->关中断…，保证了一直可以输入进来。在【输入完成的时候】保持关中断出 while 循环，这样就无法再继续读入键盘仍然按下时候的信息啦！syscallGetChar 的部分实现实现如下图所示。

```

197 while(c == 0){//一直等待用户输入，只有用户输入了c才不是0!
198     enableInterrupt();
199     c = keyBuffer[bufferHead];
200     disableInterrupt();
201 }
202 tf->eax=c;//保存到tf【也就是函数的返回值-eax中!】|
203 char wait_enter=0;
204 while(wait_enter==0){//如果下一次拿到的keyBuffer中的内容是0，那么就不输出（用户输入完成但没有按enter呢!）
205     //如果下一次拿到了enter，那么就关闭中断，直接让printf输出（中断的嵌套）
206     enableInterrupt();
207     wait_enter = keyBuffer[bufferHead+1];
208     disableInterrupt();
209 }

```

四、思考题的看法

EX1：计算机系统的中断机制在内核处理硬件外设的 I/O 这一过程发挥了什么作用？

A1：因为外设的 I/O 的速度会比 CPU 速度慢很多很多。引入中断机制有以下两个作用：

- 1、让外设来做 I/O 的同时,CPU 可以在这个过程中同时完成其他的任务，而不必一直等待 I/O 的完成.
- 2、在硬件完成 I/O 的时候再使用中断让 CPU 不做现在的工作而来处理原来的进程 I/O 之后的工作呢！

并且，相比较 轮询和 DMA 的方式，中断是一种比较灵活高效的方式

EX2：IA-32 提供了四个特权级，但 TSS 中只有 3 个堆栈位置信息，为什么没有 ring3 的信息？

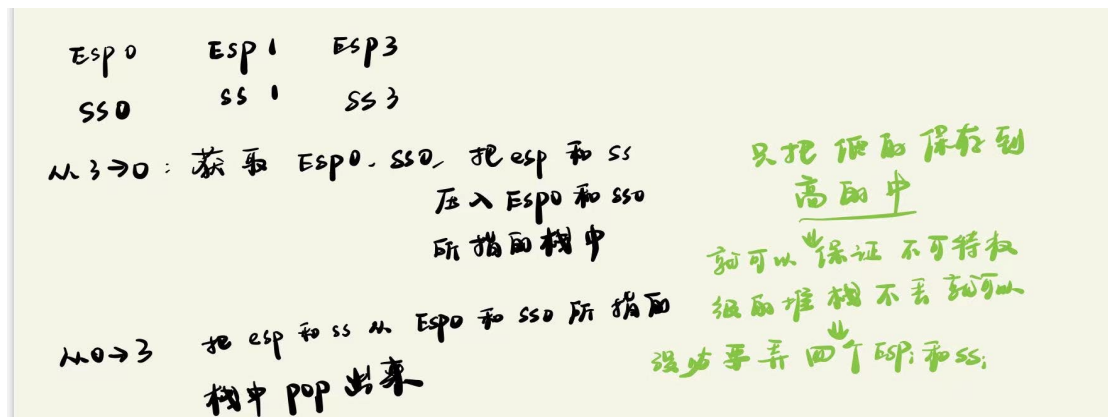
A2：特权级变化的时候栈的变化

Situation1：低-高（如 ring3->ring0）：从 TSS 获取高特权级目标栈段，将低特权级的信息压入高特权级栈中（ss 和 esp）

Situation2：高-低（如 ring0->ring3）：将低特权级栈中取出并恢复到 ss 和 esp。

所以在 TSS 中，从高-低的时候，CPU 是不需要在 TSS 中寻找低特权级目标栈的，所以最低特权级 ring3 的栈段是没有必要保存在 TSS 中的呢！（没有人比他级别更低，所以没有人会把它的栈段保存到“ss3”和“esp3”之中呢!）。

下图通过 ring0<->ring3 的栈转换过程详细说明一下。



EX3: 系统调用前后若不保存被调用者保存寄存器会不会产生不可恢复的错误?

A3: 会的! TSS 的功能只能把栈段恢复, 但是并没有恢复原来级别进程中的全部状态, 如在计算过程中的寄存器, 它们的值还是错误的数值, 这样就会出现不可恢复的错误。

EX4: printf 的%d、%x、%s、%c 转换说明符的含义。

A4:

%d – 转换成有符号的十进制整数

%x – 转换成无符号十六进制整数

%s – 转换成字符串

%c – 转换成一字符