

OS Lab

Lab2

191220008 陈南瞳

924690736@qq.com

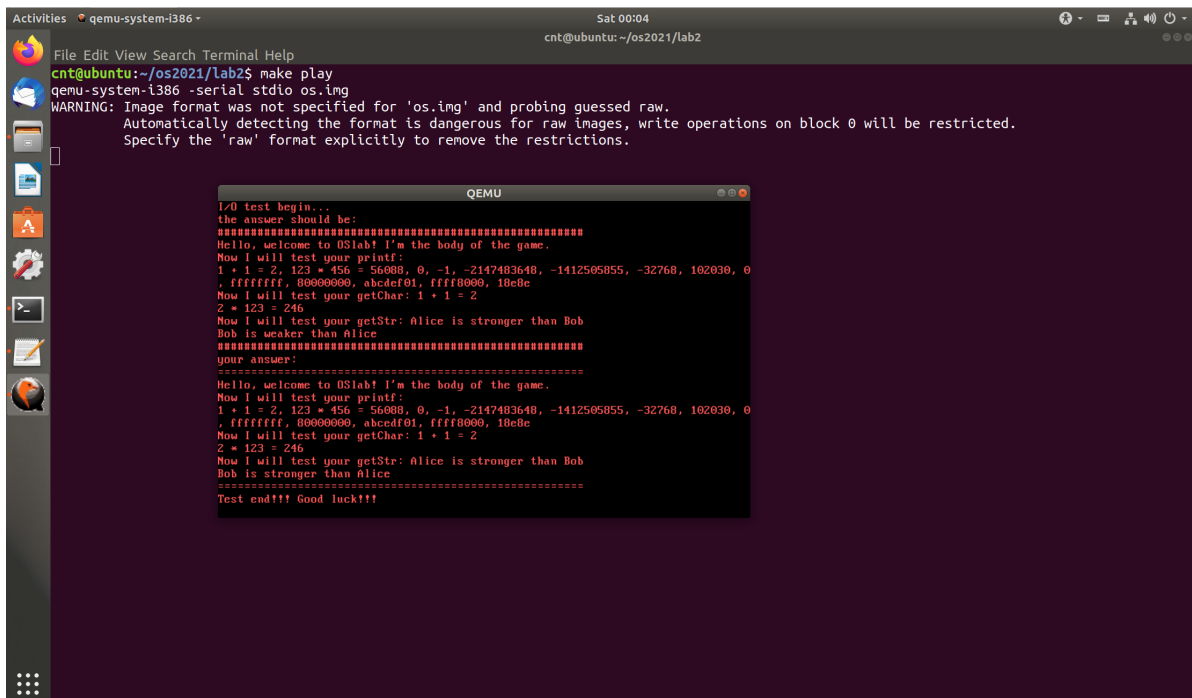
一、实验进度

完成了Lab2中的所有内容：

- 中断机制
- 实现系统调用库函数printf和对应的处理例程
- 键盘按键的串口回显
- 实现系统调用库函数getChar、getStr和对应的处理例程

二、实验结果

加载用户程序，实验结果如下：



```
Activities qemu-system-i386 Sat 00:04
cnt@ubuntu: ~/os2021/lab2
File Edit View Search Terminal Help
cnt@ubuntu:~/os2021/lab2$ 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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

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, 00000000, abcdef01, ffff8000, 18e8c
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, 00000000, abcdef01, ffff8000, 18e8c
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!!!
```

三、实验修改的代码位置

本次实验主要是对 TODO 部分的代码的填写。

(一) 程序的加载

与lab1类似，系统通过 `bootloader/start.s` 转到 `bootloader/boot.c` 的 `bootMain` 函数，由于用户程序的 `.text` 段的起始地址为 `0x200000`，所以此处的 `esp` 就应设为 `0x1ffffff`。

```
movl $0x1ffffff, %eax # TODO: setting esp
movl %eax, %esp
```

在 `bootMain` 函数中，对 `kernel` 的可执行文件进行加载，和lab1一样使用读磁盘的形式将前1~200扇区中的ELF文件读入内存。然后从ELF头中获取 `kernel` 程序入口和程序头表的位置，再据此得到程序偏移量。

```
kMainEntry = (void (*)(void))((struct ELFHeader *)elf)->entry;
phoff = ((struct ELFHeader *)elf)->phoff;
offset = ((struct ProgramHeader *)elf + phoff)->off;
```

然后就根据得到的偏移量找到 `kernel` 的 `.text` 部分，从磁盘拷到相应地址再进入执行即可。

(二) 中断机制与系统调用

进入到 `kernel/main.c`，进行一系列初始化，其中需要自行完成的是 `idt` 的初始化，因而在 `idt.c` 中根据需要的中断向量号，完成对中断向量表的填写。

```
void initIdt() {
    ...
    /*
     * init your idt here
     * 初始化 IDT 表，为中断设置中断处理函数
     */
    /* Exceptions with error code */
    setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault, DPL_KERN);
    // TODO: 填好剩下的表项
    setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS, DPL_KERN);
    setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent, DPL_KERN);
    setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault, DPL_KERN);
    setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
    setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
    setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck, DPL_KERN);
    setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException, DPL_KERN);

    /* Exceptions with DPL = 3 */
    // TODO: 填好剩下的表项
    setTrap(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
    setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER); // for int
    0x80, interrupt vector is 0x80, Interruption is disabled
    ...
}
```

接着进入 `kernel/kernel/kvm.c` 的 `loadUMain` 函数中，模仿 `bootloader` 加载内核完成了代码填写。

(三) printf的实现例程

printf 相关函数在 lib/syscall.c 中被定义。printf 的机理是基于中断陷入内核，经过中断处理函数完成字符串的打印；此外，还需要实现 printf 的格式化输出，即格式串的解析。

printf 函数采用了可变参数，而 paraList 指向了第一个指定参数 format，根据参数在栈中连续存储的机制，通过 paraList += 4 的方式获取每个参数的地址。当遇到 format[i] = '%' 时表示此处需要格式化输出，接下来读取的工作就是根据 format 中出现的格式字符串类型，读取相应类型的参数，读完这个 paraList 再移到下一个参数中。

利用框架代码提供的 dec2Str，hex2Str 和 rstr2Str 函数将十进制整数，十六进制整数和字符串加入输出字符串中并修正 count，十分便捷。

处理完 printf 中的字符串之后，执行 syscallPrint 函数，把处理完毕的字符串的内容输出到显存，而需要填写的是维护光标和打印字符串的相关代码，主要是一些边界条件（换行、行末）的处理稍复杂，但利用提供的 scrollScreen 函数，实现起来并不困难。在 irqHandle 执行完后，回到 doIrq.s 的 asmDoIrq 中，此时相应的内容出栈，然后执行 iret 函数，回到用户态。

```
int pos = (80 * displayRow + displayCol) * 2;
if (pos >= 80 * 25 * 2) {
    scrollScreen();
    displayRow = 24;
    pos = (80 * displayRow + displayCol) * 2;
}
...
if(displayRow > 24) {
    scrollScreen();
    displayRow = 24;
}
```

(四) getChar和getStr的实现例程

首先，需要实现键盘按键的串口回显。在 initIdt 函数中加入中断向量号为 0x21，中断处理函数为 irqKeyboard 的门描述符。

```
setTrap(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
```

然后参照 printf 的部分逻辑，完成 KeyboardHandle 函数的部分填写，注意特殊字符和屏幕显示的变化，还需要有一个缓存区记录输入的字符。

在 lib/syscall.c 中完成函数定义。

```

char getChar(){ // 对应SYS_READ STD_IN
    // TODO: 实现getChar函数, 方式不限
    char *str;
    syscall(SYS_READ, STD_IN, (uint32_t)str, (uint32_t)1, 0, 0);
    return *str;
}

void getStr(char *str, int size){ // 对应SYS_READ STD_STR
    // TODO: 实现getStr函数, 方式不限
    syscall(SYS_READ, STD_STR, (uint32_t)str, (uint32_t)size, 0, 0);
    return;
}

```

`syscallGetChar` 函数等待输入直到按回车才结束, 回车前的所有输入字符都会逐个显示在屏幕上。但只有第一个字符作为函数的返回值 (需要有一个缓存区记录输入的字符)。

```

while(1) {
    enableInterrupt();
    if(keyBuffer[bufferTail] == 13) {
        keyBuffer[bufferTail] = '\n';
        bufferTail--;
        break;
    }
}

```

`syscallGetStr` 函数需要一直读到回车, 并将最后的 `'\n'` 改为 `'\0'`。

```

while(1) {
    enableInterrupt();
    if(keyBuffer[bufferTail] == 13) {
        keyBuffer[bufferTail] = 0;
        break;
    }
}

```

四、遇到的问题和对这些问题的思考

1、`printf`的参数: 一开始对于 `printf` 的参数有些迷惑, 不明白是什么意思。

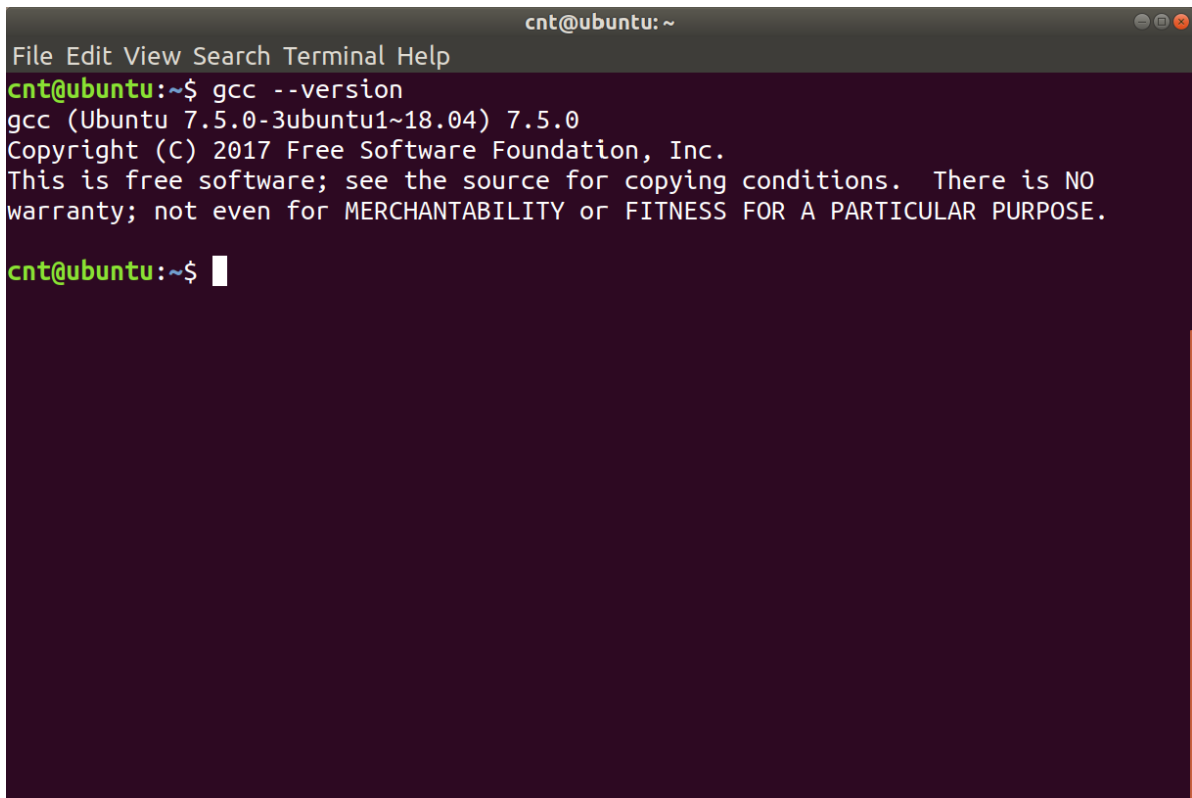
```

void printf(const char *format,...) {
    ...
}

```

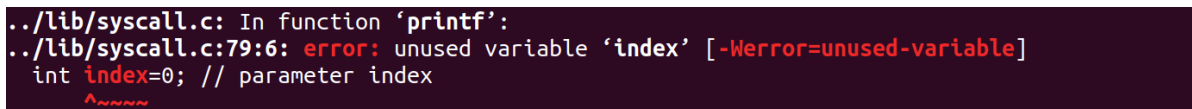
经查阅, 才知道是可变参数的形式。因此, 对于 `paraList` 指针的移动, 可以采取不断 `paraList += 4` 的方法来获取各个参数的地址。

2、gcc版本的影响：由于使用的Ubuntu是18.04版本，默认gcc是7.5.0，不会对实验造成影响。但身边有同学的Ubuntu是16.04版本，默认的gcc版本是4.5.0，会导致 boot failed。

A terminal window titled 'cnt@ubuntu: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'cnt@ubuntu:~\$'. The command 'gcc --version' has been executed, resulting in the following output: 'gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0', 'Copyright (C) 2017 Free Software Foundation, Inc.', 'This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.', and the prompt 'cnt@ubuntu:~\$' with a cursor.

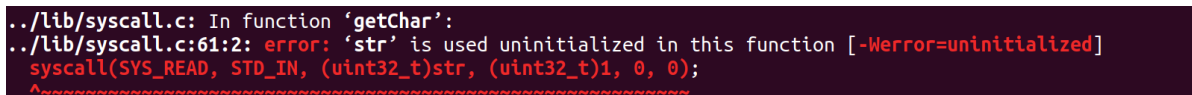
3、编译错误：

① 定义了使用的变量index。

A terminal snippet showing a GCC error: '..../lib/syscall.c: In function 'printf':', '..../lib/syscall.c:79:6: error: unused variable 'index' [-Werror=unused-variable]', and the code line 'int index=0; // parameter index' with a red squiggly line under 'index'.

解决方法：注释（或删除）掉该语句即可。

② 定义未初始化的变量str，无法通过编译。

A terminal snippet showing a GCC error: '..../lib/syscall.c: In function 'getChar':', '..../lib/syscall.c:61:2: error: 'str' is used uninitialized in this function [-Werror=uninitialized]', and the code line 'syscall(SYS_READ, STD_IN, (uint32_t)str, (uint32_t)1, 0, 0);' with a red squiggly line under 'str'.

发现下面还写着一句话：

cc1: all warnings being treated as errors

解决方法：将Makefile中的 -werror 选项删除即可。

五、对讲义或框架代码中某些思考题的看法

思考题

1、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?

因为低特权级向高特权级切换时, ring3的SS和ESP值会被压入堆栈, 当从内环返回外环时, 从堆栈中恢复即可, 因此TSS不需要保存最外层的ring3堆栈信息。

2、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?

会产生不可恢复的错误。因为内核态也会使用到寄存器, 会把寄存器的旧值覆盖。若未进行保存和恢复, 则当从内核态回到用户态的时候就失去了原来寄存器里的数据。

六、实验心得或对提供帮助的同学的感谢

于我而言, 本次lab2的难度特别大, 一是因为对于ics中相关知识遗忘较多, 相关概念的记忆存在偏差; 二是因为框架代码较为复杂, 难以理解。

因此, 在做实验的前期特别痛苦, 无法下手、进展很慢, 且由于ics相关知识的掌握较为薄弱, 实验教程不太能看懂, 阅读框架代码也很吃力, 直到有其他很厉害的同学指导才能推进实验。虽然助教jj很贴心地高亮了每个部分的TODO, 但对于我这种基础很薄弱同学来说, 还是难以理清实验的脉络和步骤。但在做完实验, 进一步理解框架代码, 进行多次的debug后, 还是感觉受益颇多。

(一点小小的建议: 可以把实验教程做得更清晰一点, 对于实验的每个阶段需要做什么, 进行一个简单的引导(类似ics的PA教程), 不至于一头雾水, 特别是对于基础薄弱的同学...理解起来可能比较吃力)