



《操作系统》实验报告

lab2



2020-3-22

薛飞阳 181860121
502126785@qq.com

1. 实验进度

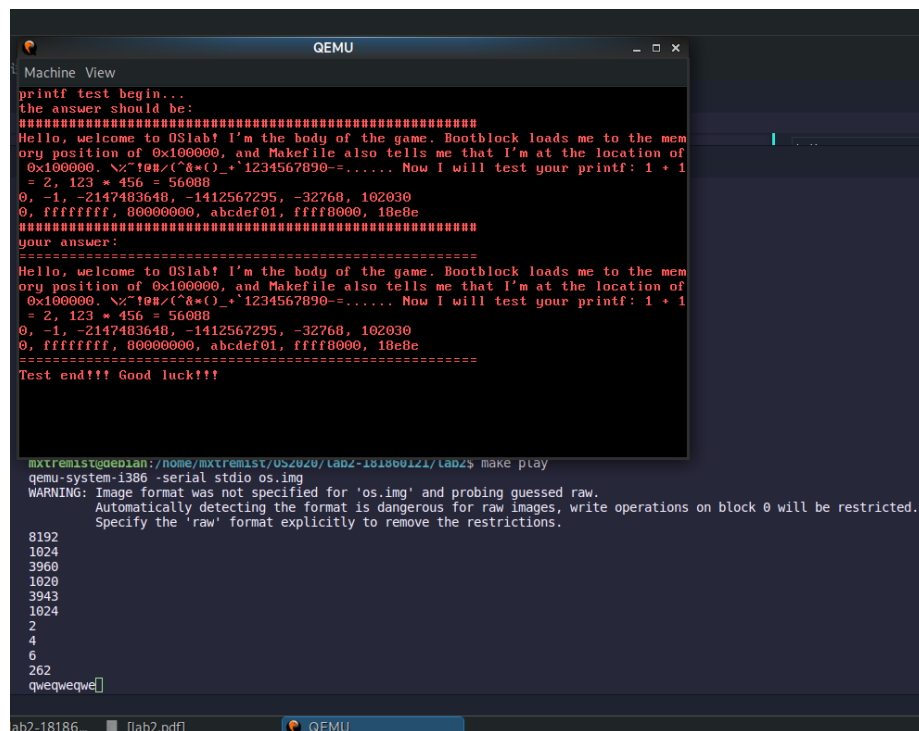
我完成了所有内容，包括格式化程序、键盘按键的串口回显、实现系统调用库函数 `printf` 和对应的处理例程、完善 `printf` 的格式化输出。

2. 实验结果

```
ld -m elf_i386 -e uEntry -Ttext 0x00000000 -o uMain.elf ./main.o ../lib/syscall.o
format fs.bin -s 8192 -b 2
FORMAT success.
1023 inodes and 3959 data blocks available.
mkdir /boot
MKDIR success.
1022 inodes and 3958 data blocks available.
copy from uMain.elf to /boot/initrd
CP success.
1021 inodes and 3944 data blocks available.
mkdir /usr
MKDIR success.
1020 inodes and 3943 data blocks available.
ls /
Name: .., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: ., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: boot, Inode: 2, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: usr, Inode: 4, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
LS success.
1020 inodes and 3943 data blocks available.
ls /boot
Name: .., Inode: 2, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: ., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
Name: initrd, Inode: 3, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13544.
LS success.
1020 inodes and 3943 data blocks available.
ls /usr
Name: .., Inode: 4, Type: 2, LinkCount: 2, BlockCount: 1, Size: 1024.
Name: ., Inode: 1, Type: 2, LinkCount: 4, BlockCount: 1, Size: 1024.
LS success.
1020 inodes and 3943 data blocks available.
make[1]: Leaving directory '/home/mxtremist/052020/lab2-181860121/lab2/app'
cat bootloader/bootloader.bin kernel/kMain.elf app/fs.bin > os.img
mxtremist@debian:/home/mxtremist/052020/lab2-181860121/lab2$
```

如上图所示，在 `lab2` 目录下执行 `make`，终端中依次显示 `FORMAT`、`MKDIR`、`CP`、`MKDIR`、`LS success`，并成功创建 `os.img` 文件。

接下来执行 `make play`，`qemu` 中将调用测试样例中的代码执行数个 `printf` 语句，同时当用户按键时将通过串口回显显示在终端上，下图中 `qemu` 窗口是 `qemu` 的执行结果，其测试代码包含在 `lab2/app/main.c` 中，终端窗口有用户输入的回显“`qweqweqwe`”。



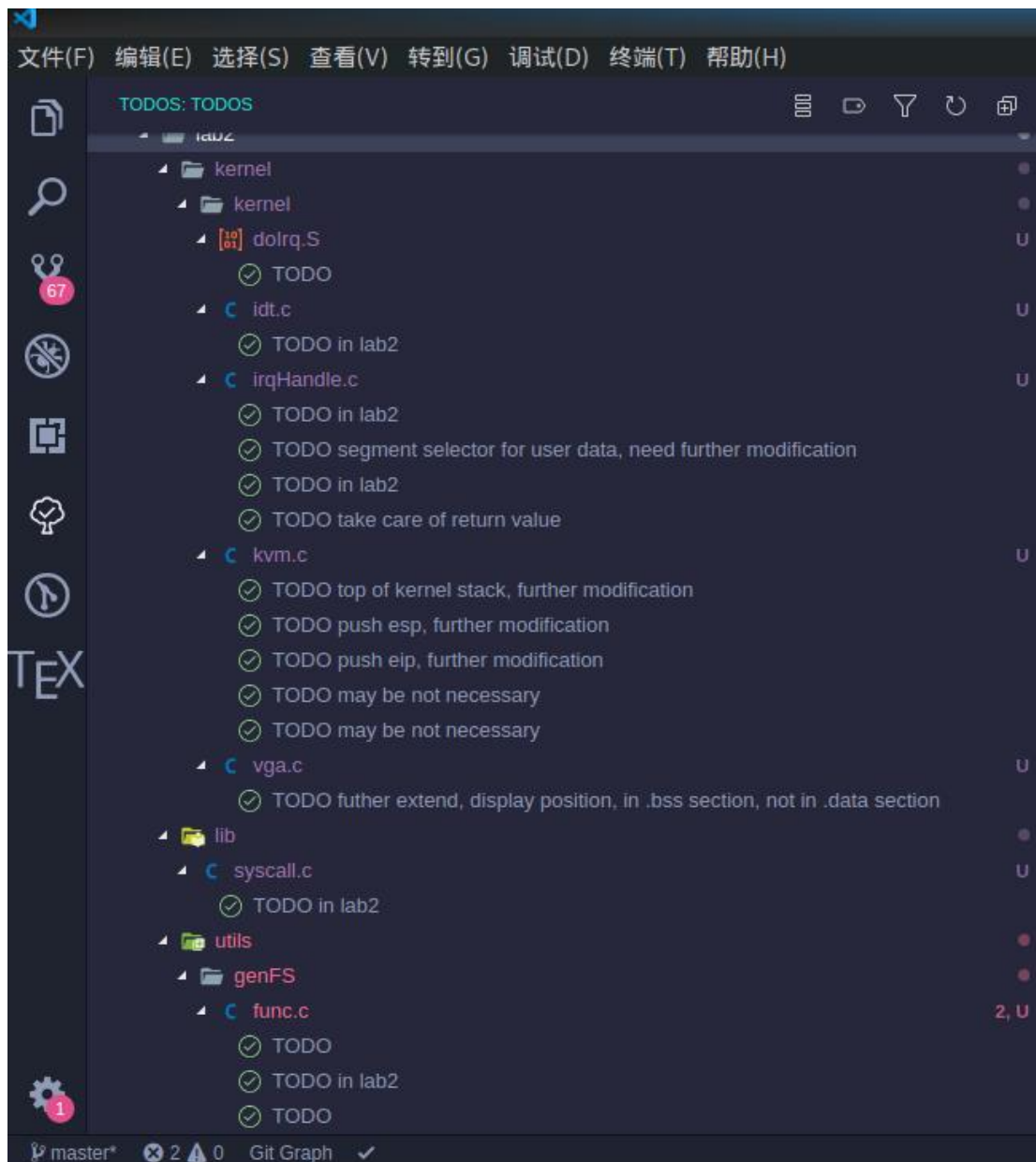
```
Machine View
printf test begin...
the answer should be:
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. Now I will test your printf: 1 * 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
your answer:
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. Now I will test your printf: 1 * 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Test end!!! Good luck!!!

mxtremist@debian:/home/mxtremist/052020/lab2-181860121/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.

8192
1024
3960
1020
3943
1024
2
4
6
262
qweqweqwe
```

3. 实验修改的代码位置

本次实验中，我们需要修改的代码位置包括：`utils/genFS/func.c`、`kernel/kernel/idt.c`、`kernel/kernel/irqHandle.c`、`lib/syscall.c`，如下图中 TODO-tree 所示。



(1) `utils/genFS/func.c`:

为了实现格式化程序，首先我们需要补全 `genFS/main.c` 中为了构建文件系统使用的函数，在 `lab_2` 中仅需要补全 `cp` 函数即可。

`copy` 的过程与 `mkdir` 相比有一定的相似之处，因为其所做的第一步是创建一个新文件，而 `mkdir` 是创建一个新目录。因此可以直接复用 `mkdir` 前半部分的创建过程，即在 `intDir` 函数之前的全部内容。这部分内容完成了读取 `superBlock` 信息、找到父

目录的 inode 表项、在父目录中找到空的 DirEntry 以及一个空的 Inode 并填写相关信息。完成这部分内容之后，实际上就是已经在正确的路径位置创建了一个对应类型的文件。在 mkdir 中，该类型为 DIRECTORY_TYPE，即目录类型，而在 cp 中需要修改为 REGULAR_TYPE。然后，需要打开 srcfile 并且将其内容复制到该文件中。同时需要修改 superBlock、inode 等的一些信息。这里可以使用提供好的 copyData 函数，该函数完成了复制内容并修改的功能。配置形参 ret = copyData(file, srcfile, &superBlock, &destInode, destInodeOffset); 其中 srcfile 是通过只读方式打开的 srcFilePath 路径的文件。

完成了 cp 函数之后，通过 make 指令就能够生成一个 os.img 文件了。由于这部分代码前半部分和 mkdir 重复率过高，后半部分的主要内容也就是调用 copyData 函数，故在实验报告中就不展示代码内容了。

(2) kernel/kernel/idt.c:

为了实现键盘输入的串口回显，首先要在 IDT 表中加上键盘中断对应的门描述符。这部分的位置在 idt.c 的 initIdt 函数中，该函数完成了初始化 idt 表，添加语句 “setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);” 完成添加一个中断号为 0x21，处理程序为 irqKeyboard、特权级为内核级 DPL_KERN 的 idt 表项。

irqKeyboard 程序在 doIrq.S 汇编文件里实现，该程序和 asmDoIrq 一起创建了 TrapFrame 并跳转到 irqHandle 程序，而 irqHandle 程序又通过中断号跳转到了 keyboardHandle 程序。这样，接下来就只需要实现 keyboardHandle 程序中具体的操作即可。

(3) kernel/kernel/irqHandle.c Part 1:

如上文中提到的，我们需要补全 irqHandle.c 中的 keyboardHandle 函数。由于需要接收键盘的信息并且通过串口显示，所以，显然需要调用这两个部分的处理函数。同路径下的 keyboard.c 中提供了读取扫描码的 getKeycode 函数和将扫描码转换为 ASCII 码的 getChar 函数。通过这两个函数，我们将能够获取到需要显示的 ASCII 字符，接下来需要将其通过串口显示。阅读 serial.c 文件。该文件的 putChar 函数可以将单个字符通过 outByte 系统调用输出在串口上。至此，如何补全 keyboardHandle 函数就变得一目了然了。

如下图所示，首先读取扫描码，通过 getChar 转换为 ASCII 码之后通过 putChar 输出在串口上。

```
void keyboardHandle(struct TrapFrame *tf) {
    // TODO in lab2
    uint32_t code = getKeyCode();
    if(code > 0)
    {
        putChar(getChar(code));
    }
    return;
}
```

(4) kernel/kernel/irqHandle.c Part 2:

我们刚刚完成了利用键盘驱动接口和串口输出接口实现的键盘按键的串口回显，接下来则需要实现 `printf` 的处理流程，从而通过调用 `printf` 显示东西在显示屏上。

阅读提示可以知道，在 `syscallPrint` 函数的 `for` 循环中已经得到了当前需要输出的单个字符 `character`，我们需要具体实现将其显示在显示屏中。

```
data = character | (0x0c << 8);
pos = (80*displayRow+displayCol)*2;
asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000));
```

上图所示代码可以实现将字符输出在显示屏的 `displayRow` 行、`displayCol` 列，而这两个参数已经在 `vga.c` 中完成了初始化为 0。因此，我们只需要在每个 `for` 循环中调用该代码，并且完成行、列的修改即可。

一般情况下行列的修改规则为：每输出一个字符，列++，当列增长到 80 时，列=0，行++。如果行=25，则需要将其置为 24 并进行一次滚屏操作。

当输出 `\n` 时，其效果应该与列增长到 80 之后相同，因此，可以将这部分相同代码整理成简单的函数 `do_newline`，该函数需要调用 `vga.c` 中的滚屏处理函数 `scrollScreen`，如下图所示：

```
void do_newline()
{
    displayCol = 0;
    displayRow++;
    if(displayRow == 25)
    {
        displayRow--;
        scrollScreen();
    }
}
```

`for` 循环的处理流程也就十分简单了：首先判断是否为换行，如果不是则通过之前提到的语句块输出字符并完成行、列信息的修改：

```
for (i = 0; i < size; i++) {
    asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
    // TODO in lab2
    if(character == '\n')
    {
        do_newline();
    }
    else
    {
        data = character | (0x0c << 8);
        pos = (80*displayRow+displayCol)*2;
        asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000));
        displayCol++;
        if(displayCol == 80)
        {
            do_newline();
        }
    }
}
```

至此，已经实现了 `printf` 的基本输出流程，即将一个字符串输出在显示屏上，接下来需要完善其格式化输出。

(5) lib/syscall.c:

在 `syscall.c` 的 `printf` 函数中，我们需要完善其处理流程。通过读取 `printf` 的第一个参数（即 `format` 字符串），我们有三种不同的状态需要做三种不同的操作（类似一个状态机），如下表格所示：

| state | 解释 | 操作 |
|-------|-------------|------------|
| 0 | 普通字符 | 直接读入字符串 |
| 1 | 需要格式转换，或者%% | 调用格式转换函数 |
| 2 | 非法输入，例如%b | 忽略%，读入剩余字符 |

情况 0 很简单，直接正常读入即可。如果是情况 1，即说明是 % 后跟一个需要处理的字符，如果是 `s`、`x`、`c`、`d` 等，则需要调用提供的格式转换函数从栈中读取对应的参数后加入到待输出字符串中，如果是 %，则说明需要输出一个 %。情况 2 虽然可以并入情况 1 中的 `else` 内，即 % 后跟着一个未被解释的字符，例如 %b，但是考虑到后续可能有扩展，加入其他的非法情况例如 \+ 一个未被解释的字符等等，故将其单独列出为一种非法的输入。新建一个 `c` 文件，加入 `stdio` 库并执行下列语句：`printf("%b")`，发现结果为输出单个字符 `b`，故对于这种情况，我们可以直接忽略掉 %，读取剩余字符。

在梳理清晰后，“状态机”部分的代码就很容易写出了。如下图所示，`percent` 为标志位，仅当输入非法时为 1。

```
if (format[i] == '%' && percent == 0)
{
    state = 1;
}
else if (percent == 1) //illegal
{
    state = 2;
}
else
{
    state = 0;
}
```

对于 `state 0` 和 `state 2`，他们的处理都是将单个字符读入并跳转到下一个，不同的是 `state 2` 需要将 `percent` 重新置为 0，方便下次读入。对于 `state 1`，除了 %% 这种类似转义符的操作，其余情况较为类似，都是需要从栈中读取特定的参数。

我们利用 `index` 参数说明已经从栈中读取了多少额外的参数，再利用初始时指向 `format` 的 `void*` 类型指针 `paraList`，则 `paraList + 4 * index` 即为指向当前需要读入的参数的 `void*` 类型指针，将其强制类型转换后即可使用。对于 `%s`、`%x`、`%d`、`%c`，我们分别需要强制类型转换为 `char **`、`int *`、`int *`、`char *`，这样就得到了指向对应类型参数的指针。将其值读出后调用转换函数，就得到了转换后的字符串，将其全部加入待输出字符串即可。下图为其中一种情况的例子，其他情况大致相同。

```
else if (format[i] == 's')
{
    string = *(char **)(paraList + index * 4);
    count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
}
```

至此，实验部分已经完成。

4. 问题的思考

"幽灵"空间

在磁盘上创建一个新文本文件, 往里面写入一个字符后, 查看该文件的大小:

- 在Windows中, 有一项叫Size on disk的数据
- 在Linux中, 使用 `ls -ls` 命令, 输出的第一列即为文件占用的磁盘空间(单位:KB)
这些显示的数值远远大于一个字节, 你知道为什么吗?

文件大小与其占用空间是两个概念, 在实际应用中, 即使文件实际大小不足一个块, 也会单独占用这个块, 故其大小则为一个块的大小。

对目录的硬链接

Linux不允许对目录创建硬链接, 你知道为什么吗? 如果允许对目录创建硬链接, 你能否举出一个影响系统工作的例子?

目录均自带两个目录项(.)和(..)。如果允许对目录的硬链接, 当执行 `ls` 遍历的时候就会产生循环。

ring3的堆栈在哪里?

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?

当从 ring3 切换到 ring0 时会将 ring3 的堆栈信息保存在结构 `TrapFrame` 内, 当切换回 ring3 时, 相关数据都可以从 `TrapFrame` 读出。

保存寄存器的旧值

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。因为内核态很可能会使用这些寄存器, 导致其原先的值丢失。