课 程 实 验 报 告

实验名称 系统调用的实现

课程名称 操作系统

院　　系 计算机科学与技术系

学　　号 191220129

姓　　名 邢尚禹

邮　　箱 191220129@smail.nju.edu.cn

实验日期 2021 年 3 月

目录

1 实验进度

已完成所有内容。

2 实验思路和过程

2.1 加载并初始化内核

由于内核是一个 elf 文件，需要在 bootloader 中实现对 elf 文件的解析。elf 文件的整体结构如下：

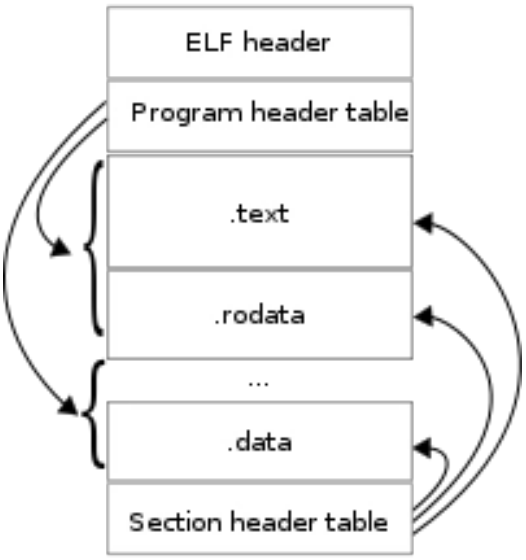

图 1: elf 文件结构

可执行的 elf 文件中，elfheader 指定了程序的 entry point，program header 指定了代码段在文件中的偏移量。直接读取对应数据，将文件中的代码和数据段加载到内存 0x100000 处，再跳转至 entry point 执行即可。

在加载和运行用户程序前，内核需要初始化串口，idt，中断，段寄存器，vga 和键盘设备。其中大部分内容框架代码已实现，只需要补充 idt 的初始化。通过查阅相关资料，知键盘中断号是 0x21，系统调用中断号是 0x80，据此填写即可。需要注意系统调用的 dpl 为 3。

```
setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
```

完成初始化后，就可以加载用户程序了。加载用户程序的过程与加载内核的过程基本相同，此处不再赘述。之后，内核通过 iret 指令进入用户空间，执行用户程序。

2.2 中断服务和系统调用

首先阅读实验代码中对中断处理的框架。各个中断服务程序将中断相关信息保存至栈中，然后统一调用 asmDoIrq，保存现场并调用对应的 irqHandle，通过 TrapFrame 的数据结构传递保存在通用寄存器内的参数。irqHandle 再根据保存的中断号调用对应的中断处理函数。我们需要在此基础上实现对不同中断的特定处理程序。

由于框架代码已给出了 putChar 函数，它可以将指定的字符从串口输出。由此可以封装自己的 log 函数，供调试使用。

```
void log(const char *str)
{
    for (int i = 0; i < 100 && str[i]; ++i)
        putChar(str[i]);
}
```

2.2.1 键盘按键输入的处理

由于填写好了键盘中断的 idt，按下键盘的按键后最终会执行到函数 KeyboardHandle。按键回显实现非常简单，直接对按键转换后的 ascii 码调用 putChar 即可。下面重点阐述键盘输入在 vga 上显示的处理方法。

打印单个字符的方法已在指导文件中给出，可以将其封装为一个函数，方便后续使用：

```
void printChar(char c)
{
    uint16_t data = c | (0x0c << 8);
    int pos = (80 * displayRow + displayCol) * 2;
    asm volatile("movw %0, (%1)" ::"r"(data), "r"(pos + 0xb8000)
        );
```

```
}
```

将所有的按键分为以下 4 类：

- 普通按键。这一类主要包含字母，数字和符号，处理方法也很简单，直接输出即可。
- 功能按键。这一类包含 shift, control, capslock, tab 等，这些按键可以不用显示。
- enter。需要注意光标的移动，要能够正确维护光标的位置，不需要输出字符。
- backspace。需要注意光标的移动，还要输出一个空格来覆盖原来可能存在的字符。而根据要求，backspace 还只能删除自己输入的字符，因此还涉及输入缓冲区的维护（输入缓冲区相关内容将在最后一小节详细阐述）。

根据上述特征，KeyboardHandle 应该这样实现：

```
void KeyboardHandle(struct TrapFrame *tf)
{
    uint32_t code = getKeyCode();
    if (code == 0xe)
    { // 退格符
        // TODO: 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
        if (displayCol && deleteBack(&inputBuf))
        {
            --displayCol;
            updateCursor(displayRow, displayCol);
            printChar(' ', displayCol, displayCol);
        }
    }
    else if (code == 0x1c)
    { // 回车符
        // TODO: 处理回车情况
        if (displayRow == 24)
            scrollScreen();
```

```
19          else
20              ++displayRow;
21          displayCol = 0;
22          insertBuf(&inputBuf, '\n');
23          putChar('\n');
24      }
25      else if (code < 0x81 && (code > 1 && code < 0xe || code > 0
            xf && code != 0x1d && code != 0x2a && code != 0x36 &&
            code != 0x38 && code != 0x3a && code < 0x45))
26      { // 正常字符
27          // TODO: 注意输入的大小写的实现、不可打印字符的处理
28          putChar(getChar(code));
29          printChar(getChar(code), displayRow, displayCol);
30          insertBuf(&inputBuf, getChar(code));
31          if (displayCol == 79)
32          {
33              displayCol = 0;
34              if (displayRow == 24)
35                  scrollScreen();
36              else
37                  ++displayRow;
38          }
39          else
40              ++displayCol;
41      }
42      updateCursor(displayRow, displayCol);
43  }
```

2.2.2 打印字符串和 printf 的实现

实现了键盘按键输入的 vga 显示后，打印字符串的功能就很容易实现了。只需要特殊处理换行符号，剩下的直接输出。光标维护的方法与上一小节完全相同，代码也高度相似，此处省略。

函数 printf 的实现并不困难，只需要根据 4 种格式，调用对应的函数即可。需要注意计数必须准确。

```
1 void printf(const char *format, ...)
```

```
{
    int i = 0; // format index
    char buffer[MAX_BUFFER_SIZE];
    int count = 0;                      // buffer index
    void *paraList = (void *)&format; // address of format in
        stack
    int decimal = 0;
    uint32_t hexadecimal = 0;
    char *string = 0;
    char character = 0;
    for (; format[i] && count <= MAX_BUFFER_SIZE; ++i)
    {
        buffer[count] = format[i];
        count++;
        //TODO in lab2
        if (format[i] == '%')
        {
            --count;
            paraList += sizeof(format);
            switch (format[++i])
            {
            case 'c':
                character = *(char *)paraList;
                buffer[count++] = character;
                break;
            case 's':
                string = *(char **)paraList;
                count = str2Str(string, buffer, (uint32_t)
                    MAX_BUFFER_SIZE, count);
                break;
            case 'x':
                hexadecimal = *(uint32_t *)paraList;
                count = hex2Str(hexadecimal, buffer, (uint32_t)
                    MAX_BUFFER_SIZE, count);
                break;
            case 'd':
                decimal = *(int *)paraList;
                count = dec2Str(decimal, buffer, (uint32_t)
```

```
                    MAX_BUFFER_SIZE, count);
                break;
            case '%':
                paraList -= sizeof(format);
                ++count;
            }
        }
    }
    if (count != 0)
        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)
            count, 0, 0);
}
```

2.2.3 输入字符串的实现

为实现输入的功能，必须维护一个输入缓冲区，键盘按键时将对应的字符加入缓冲区，系统调用输入字符串时将取出的字符（串）从缓冲区中删除。因此需要建立数据结构 InputBuf。

```
typedef struct
{
    int size;
    char buf[MAX_INPUT_SIZE];
} InputBuf;
void clearBuf(InputBuf *buf)
{
        buf->size = 0;
}
void insertBuf(InputBuf *buf, char c)
{
        buf->buf[buf->size++] = c;
        if (buf->size > MAX_INPUT_SIZE)
        {
                log("Input buf overflow!\n");
                assert(0);
        }
}
int deleteBack(InputBuf *buf)
```

```
{
        if (buf->size)
        {
                --buf->size;
                return 1;
        }
        return 0;
}
```

但仅仅有上述插入删除功能还不够，还需要能够在缓冲区为空且执行 getChar 或 getStr 的系统调用时阻塞用户进程，直到用户在终端中输入字符（串）。阻塞解除的条件是输入回车，因此可以先开中断再执行 hlt 指令，直到检测到用户输入回车，再关中断，执行接下来的流程。由此，可以这样实现从缓冲区中获取字符（串）的方法：

```
char retriveChar(InputBuf *buf)
{
    // retrive a single char
    asm volatile("sti");
    while (!buf->size || buf->buf[buf->size - 1] != '\n')
        waitForInterrupt();
    asm volatile("cli");
    char res = buf->buf[0];
    buf->size = 0;
    return res;
}
void retriveStr(InputBuf *buf, char *dst)
{
    // retrive until \n
    asm volatile("sti");
    while (!buf->size || buf->buf[buf->size - 1] != '\n')
        waitForInterrupt();
    asm volatile("cli");
    int i = 0;
    for (; i < buf->size && buf->buf[i] != '\n'; ++i)
        dst[i] = buf->buf[i];
    dst[i] = 0;
    buf->size = 0;
```

```
24 }
```

实现这样的数据结构之后，只需要简单地调用它的方法就可以实现 getChar 和 getStr 了。键盘输入时维护缓冲区的相关代码在本节第 1 小节。

3 实验结果

kernel 可以正确加载用户程序运行，并提供打印字符串，键盘输入并在串口和 vga 上回显，读取输入字符的中断服务程序。用户程序中的 printf 工作正常。

```
xsy@ASUS-VivoBook:/media/xsy/SSD/MEGA/assignments/os/lab2/lab2$ 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.
Enter kernel main.
Initialization successful.
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall print
syscall getchar
2
syscall print
syscall print
syscall print
syscall getstr
Bob
syscall print
syscall print
syscall print
absjiahndjsdisjcxoijjxkczjiodijsijdiojaskcfmjkxzmvopaskdfjiewjfkdlsjcmkxlskjdfasddk
```

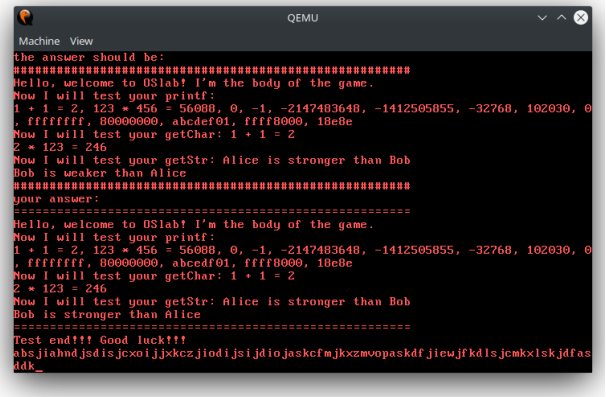

图 2: 实验结果

4 问题与思考

1. 一开始直接使用框架代码给出的 makefile 进行编译并执行，但发现报错”boot block too large”，于是根据实验 0 的相关指导将命令改为

objcopy -S -j .text -O binary bootloader.elf bootloader.bin，即可正确完成编译。

2. 在命令行中运行 qemu-system-i386 os.img，窗口显示"no bootable device"。通过询问助教，得知原因可能是 gcc 版本问题。安装 gcc6 并编译可成功运行。另外，一位学长（计网助教）给出的解决方案是加一个编译参数-fno-asynchronous-unwind-tables，经过尝试也可以成功。查询相关资料知，"This option determines whether unwind information is precise at an instruction boundary or at a call boundary. If -fno-asynchronous-unwind-tables is specified, the unwind table is precise at call boundaries only." 新版本的 gcc 默认行为是"-fasynchronous-unwind-tables",而旧版的 gcc 默认"-fno-asynchronous-unwind-tables"。猜想 unwind information 的位置不同将影响加载过程，而 qemu 模拟的是老式的硬件，可能会产生不匹配。

3. kernel 按照-O1 编译可以正常工作，但如果用-O2 编译就会出现问题（getStr 函数无法响应回车输入，一直保持阻塞状态）。根据以往经验，猜想可能是因为存在未初始化的变量，但仔细查找后并没有发现。有可能是因为 kernel 作为操作系统内核有特殊性，不能盲目开启优化。

5 建议

1. 框架代码修改建议：在框架代码中初始化 gdt 时，用户段的段描述符是将 base 设置为 0x200000，而 Makefile 中-Ttext 是 0。因此，框架代码是想通过段基地址转换的偏移量访问用户代码和数据。但是，这样的想法和 linux 的实现方式相违背的。在 linux 中，用户段的 base=0, limit=0xffffffff，即不使用段基地址转换。我一开始阅读框架代码，以为实现方式与 linux 相同，于是直接将-Ttext 改为 0x200000，没有仔细看 gdt 的初始化部分，导致后面运行出现问题，浪费了很多时间。而且，像我这样操作的同学还很多，可见这一想法是自然的，是框架代码的实现方式不佳。因此，我希望能进行如下修改：将 app/Makefile 中的-Ttext 改为 0x200000，在 kernel/kernel/kvm.c 中把用户代码和

数据段描述符改为

```
gdt[SEG_UCODE] = SEG(STA_X | STA_R, 0, 0xffffffff, DPL_USER
    );
gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```

2. 修改 Makefile：

   - 不要使用-O2 优化，可能会出现问题；
   - 编译参数里添加-fno-asynchronous-unwind-tables。