OS Lab1 高海峰 21373512

思考题

Thinking 1.1

分别用`gcc`与`mips-linux-gnu-gcc`编译生成的`.o`文件，通过`readelf`与`mips-linux-gnu-readelf`解析后，输出比较：

对于.o文件分别反汇编的结果输出差异:
`gcc`：文件格式 `elf64-x86-64`
`mips-linux-gnu-gcc`：文件格式 `elf32-tradbigmips`
前者地址为64位，后者地址为32位

objdump传入的参数的含义(部分)：

参数	含义
-a	显示档案库的成员信息,类似ls -l将lib*.a的信息列出。
-f	显示objfile中每个文件的整体头部摘要信息。
-d	将代码段反汇编 反汇编那些应该还有指令机器码的section
-D	与 -d 类似，但反汇编所有section
-S	将代码段反汇编的同时，将反汇编代码和源代码交替显示，源码编译时需要加-g参数，即需要调试信息
-C	将C++符号名逆向解析
-l	反汇编代码中插入源代码的文件名和行号
-j	section: 仅反编译所指定的section，可以有多个-j参数来选择多个section
-x	显示所有的头部信息，实测基本包含了-a，-f，-p和-h参数的输出

Thinking 1.2

进入./tools/readelf文件夹，执行`make`指令，编译成功后，执行`./readelf ../../target/mos`后，可以观察到输出

```
0:0x0
1:0x80100000
2:0x80101d20
```

```
3:0x80101d38
4:0x80101d50
5:0x0
6:0x0
7:0x0
8:0x0
9:0x0
10:0x0
11:0x0
12:0x0
13:0x0
14:0x0
15:0x0
16:0x0
17:0x0
```

`make && readelf -h readelf`

```
cc -c main.c
cc -c readelf.c
cc main.o readelf.o -o readelf
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本：                         0
  类型：                             DYN (Position-Independent
Executable file)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x1180
  程序头起点：           64 (bytes into file)
  Start of section headers:          14488 (bytes into file)
  标志：             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

`make hello && readelf -h hello`

```
cc hello.c -o hello -m32 -static -g
```

```
ELF 头：
  Magic：   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  类别：                             ELF32
  数据：                             2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI 版本：                         0
  类型：                             EXEC (可执行文件)
  系统架构：                         Intel 80386
  版本：                             0x1
  入口点地址：               0x8049600
  程序头起点：          52 (bytes into file)
  Start of section headers:          746260 (bytes into file)
  标志：             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 34
```

可以看出来，`readelf`的类别是`ELF64`，而`hello`的类别是`ELF32`，我们可以知道我们自己写的`readelf.c`编译链接后的可执行文件只可以对32位文件进行使用，而不能对64位文件进行使用。

Thinking 1.3

GXemul 已经提供了bootloader 的引导（启动）功能。MOS 操作系统不需要再实现 bootloader 的功能。在`kernel.lds`中，设置了`_start`为程序的入口，并且将`.text`、`.data`、`.bss`设置到了特定的加载位置，因此在执行make run之后，PC会直接来到`_start`，并且执行第一个指令`mtc0  zero, CP0_STATUS`，接着会初始化内核栈指针`li sp, 0x80400000`，并且跳转到`mips_init`。因此内核入口能够被正确的跳转。

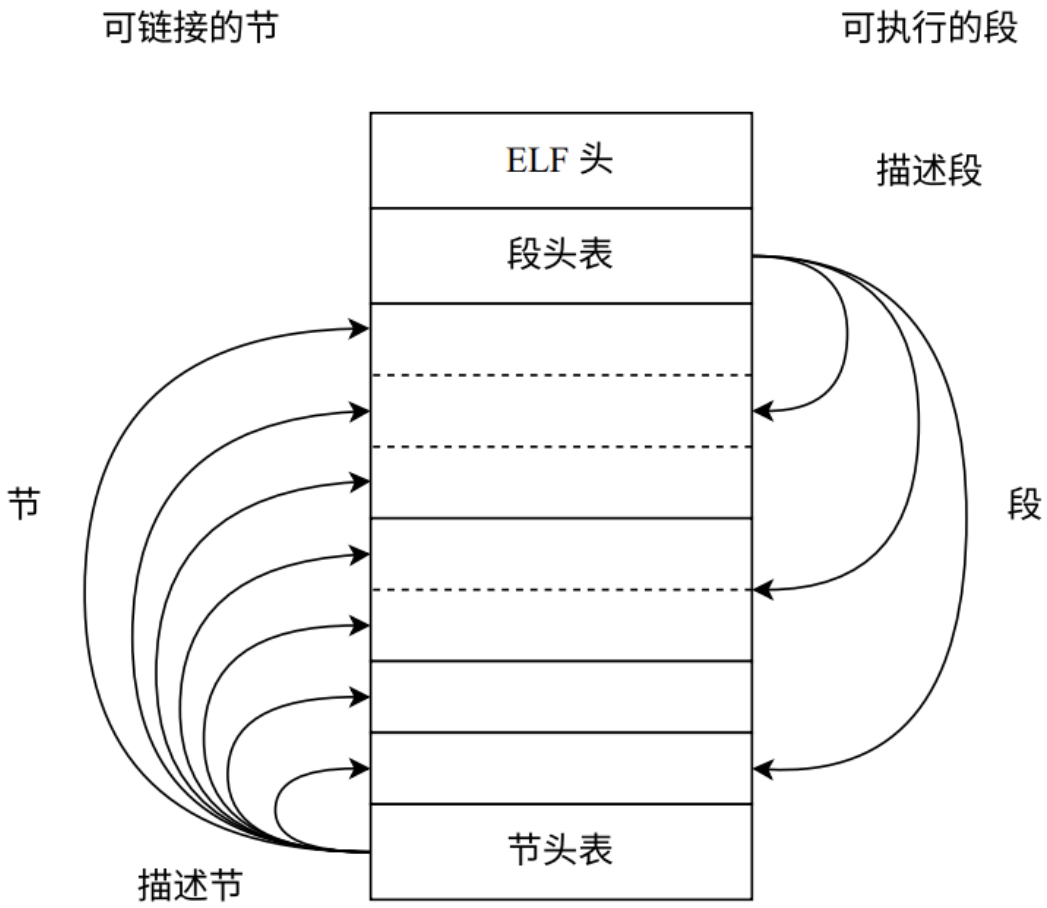
难点分析

本次实验的难点主要为以下两点两点

1. ELF文件的结构，如何通过ELF文件来获取ELF各个内部数据数据。
2. vprintfmt如何实现。

1.ELF文件的结构与解析

在教程当中我们有这样的图ELF的结构图如下：



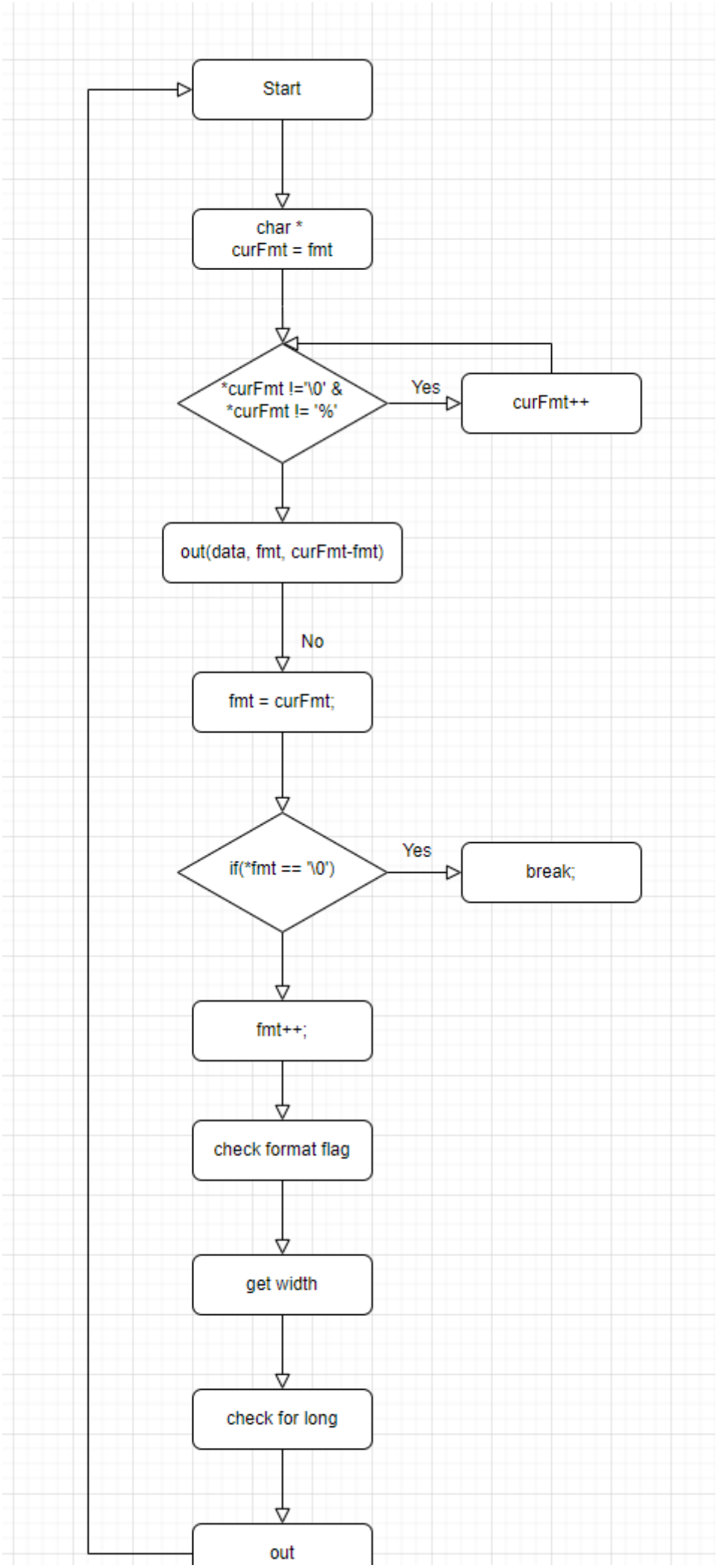
1. ELF 头，包括程序的基本信息，比如体系结构和操作系统，同时也包含了节头表和段头表相对文件的偏移量（offset）。
2. 段头表（或程序头表，program header table），主要包含程序中各个段（segment）的信息， 段的信息需要在运行时刻使用。
3. 节头表（section header table），主要包含程序中各个节（section）的信息，节的信息需要在程序编译和链接的时候使用。

```
sh_table = binary + ehdr->e_shoff;     //段表头地址
sh_entry_count = ehdr->e_shnum;        //段表项的数量
sh_entry_size = ehdr->e_shentsize;     //每个段表项的大小
shdr = (Elf32_Shdr *)(sh_table + i *
                      sh_entry_size); //某一特定段
addr = shdr->sh_addr;                  //某一特定段的地址
```

2.vprintfmt的实现

当前的字符遇到`%`或者`\0`的时候，需要将之前的字符原样打印。并且进一步判断：如果遇到的是`\0`，则表示已经对fmt打印完毕，退出即可，如果遇到的是`%`，那么则需要根据文法分析打印的类型。

vprintfmt的流程图可以粗略的表示成以下形式


Start
char *
curFmt = fmt
*curFmt !='\0' &
*curFmt != '%'
Yes
curFmt++
out(data, fmt, curFmt-fmt)
No
fmt = curFmt;
if(*fmt == '\0')
Yes
break;
fmt++;
check format flag
get width
check for long
out

在厘清逻辑以后，我们就可以针对性地完成vprintfmt的设计了。

check format flag

```
ladjust = 0;
padc = ' ';
if(*fmt == '-') {
    ladjust = 1;
    fmt++;
}
if(*fmt == '0') {
    padc = '0';
    fmt++;
}
```

get width

```
width = 0;
while ((*fmt >= '0') & (*fmt <= '9')){
    width = width * 10 + (*fmt - '0');
    fmt++;
}
```

check for long

```
long_flag = 0;
if (*fmt == 'l') {
    long_flag = 1;
    fmt ++;
}
```

case 'd'/'D'

```
if (num < 0) {
    num = -num;
    neg_flag = 1;
}
print_num(out, data, num, 10, neg_flag,
          width, ladjust, padc, 0);
```

注意在%d / %D的时候的时候需要提前判断num的正负即可。

实验体会

Lab1的内容主要是了解操作系统启动的基本流程，掌握**ELF**的结构以及解析过程，并且针对具体的**C**库函数(printf)上手编写练习。

在具体学习的过程中，需要注意细枝末节。

虽然Lab1的内容容量并不小，但实际上Lab1的需要编写的内容很短，很大程度上课程组已经将Mos的雏形做的比较完善了，我们只需要对其中的很少部分的功能进行修修补补即可。因此仅仅通过完成作业，我们对于操作系统的运行的了解还是远远不够的，还是应该更加了解课设代码。