OS_lab1实验报告

思考题

Thinking 1.1 编译链接

gcc命令

使用gcc编译器的`gcc -E file.c`， `gcc -c file.c`，`gcc -o file file.c`命令可以分别生成编译器预处理、编译、链接之后产生的文件，通过查看相应代码可以看出，在实例`hello.c`生成可执行文件的过程中，`printf`的实现是在链接的过程中完成的。

ld命令

ld可用于将目标文件和库链接成可执行文件，它相当于执行了链接部分的功能。

命令格式：

```
ld [options] objfile...
```

gcc命令中，也会使用ld命令来完成链接工作。

readelf命令

readelf命令用于显示ELF可执行文件的相关信息，命令格式：

```
readelf -h <file> #显示头文件信息
readelf -S <file> #显示节表信息
readelf -s <file> #显示符号表信息
# ...
readelf -a <file> #显示所有信息
```

```
git@22373024:~/22373024/think1 (lab1)$ readelf -h hello
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:               0x1060
  程序头起点:          64 (bytes into file)
  Start of section headers:          13976 (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
```

objdump命令

objdump命令用于反汇编目标文件或者可执行文件，可用于查看预处理，编译，链接之后产生的二进制文件。

传入参数的含义：

```
# objdump --help
-D, --disassemble-all    Display assembler contents of all sections
    --disassemble=<sym>  Display assembler contents from <sym>
-S, --source             Intermix source code with disassembly
    --source-comment[=<txt>] Prefix lines of source code with <txt>
```

可以看出，`-D`表示反汇编，而`-S`用于将源代码和反汇编代码共同显示出来。

MIPS 交叉编译工具链

使用MIPS 交叉编译工具链重新进行编译链接的过程，也就是在上述命令上加上`mips-linux-gnu-`前缀：

```
mips-linux-gnu-gcc -E hello.c > hello_mips.i #预处理

mips-linux-gnu-gcc -c hello.c -o hello_mips.o #编译
mips-linux-gnu-objdimp -DS hello_mips.o > hello_mips_o_DS

mips-linux-gnu-gcc -o hello_mips hello.c #链接
mips-linux-gnu-objdump -DS hello_mips > hello_mips_DS
```

反汇编产生的文件中，文件格式由`elf64-x86-64`变为`elf32-tradbigmips`。

Thinking 1.2 readelf

使用`readelf`解析`target/mos`之后：

```
git@22373024:~/22373024 (lab1)$ readelf -S target/mos
There are 17 section headers, starting at offset 0x49cc:

节头:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        80400000 0000c0 0016f0 00 WAX  0   0 16
  [ 2] .reginfo          MIPS_REGINFO    804016f0 0017b0 000018 18   A  0   0  4
  [ 3] .MIPS.abiflags    MIPS_ABIFLAGS   80401708 0017c8 000018 18   A  0   0  8
  [ 4] .rodata           PROGBITS        80401720 0017e0 000230 00   A  0   0 16
  [ 5] .pdr              PROGBITS        00000000 001a10 000280 00      0   0  4
  [ 6] .comment          PROGBITS        00000000 001c90 000025 01  MS  0   0  1
  [ 7] .gnu.attributes   GNU_ATTRIBUTES  00000000 001cb5 000010 00      0   0  1
  [ 8] .debug_info       MIPS_DWARF      00000000 001cc5 000f9b 00      0   0  1
  [ 9] .debug_abbrev     MIPS_DWARF      00000000 002c60 0005c8 00      0   0  1
  [10] .debug_aranges    MIPS_DWARF      00000000 003228 000100 00      0   0  8
  [11] .debug_line       MIPS_DWARF      00000000 003328 00099a 00      0   0  1
  [12] .debug_str        MIPS_DWARF      00000000 003cc2 000502 01  MS  0   0  1
  [13] .debug_frame      MIPS_DWARF      00000000 0041c4 000338 00      0   0  4
  [14] .symtab           SYMTAB          00000000 0044fc 000300 10     15  27  4
  [15] .strtab           STRTAB          00000000 0047fc 000124 00      0   0  1
  [16] .shstrtab         STRTAB          00000000 004920 0000ac 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

使用`readelf -h`后:

```
git@22373024:~/22373024/tools/readelf (lab1)$ readelf -h hello
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:          746252 (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
```

```
git@22373024:~/22373024/tools/readelf (lab1)$ readelf -h 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
```

从`Makefile`文件中可以看出，在编译`hello`的时候，使用了`-m32 -static -g`选项，使得编译产生了一个32位的静态可执行文件，而`readelf`是64位的，因此无法解析`readelf`自己。

```
readelf: main.o readelf.o
	$(CC) $^ -o $@
hello: hello.c
	$(CC) $^ -o $@ -m32 -static -g
```

Thinking 1.3 启动入口地址

大部分bootloader括stage1和stage2，stage1在ROM或者FLASH中执行，初始化硬件设备并为加载stage2准备RAM空间，在stage2中，会执行GRUB，GRUB会一步一步的加载自身代码，从而识别文件系统，从而将内核加载到内存并跳转，保证能够跳转到正确的内核位置，所以启动入口地址不一定是内核入口地址。

难点分析

本次实验的难点我觉得有两点：**理解ELF文件的结构**和**补全`vprintfmt()`函数。**

ELF文件

首先，我们应该明确ELF文件的结构：

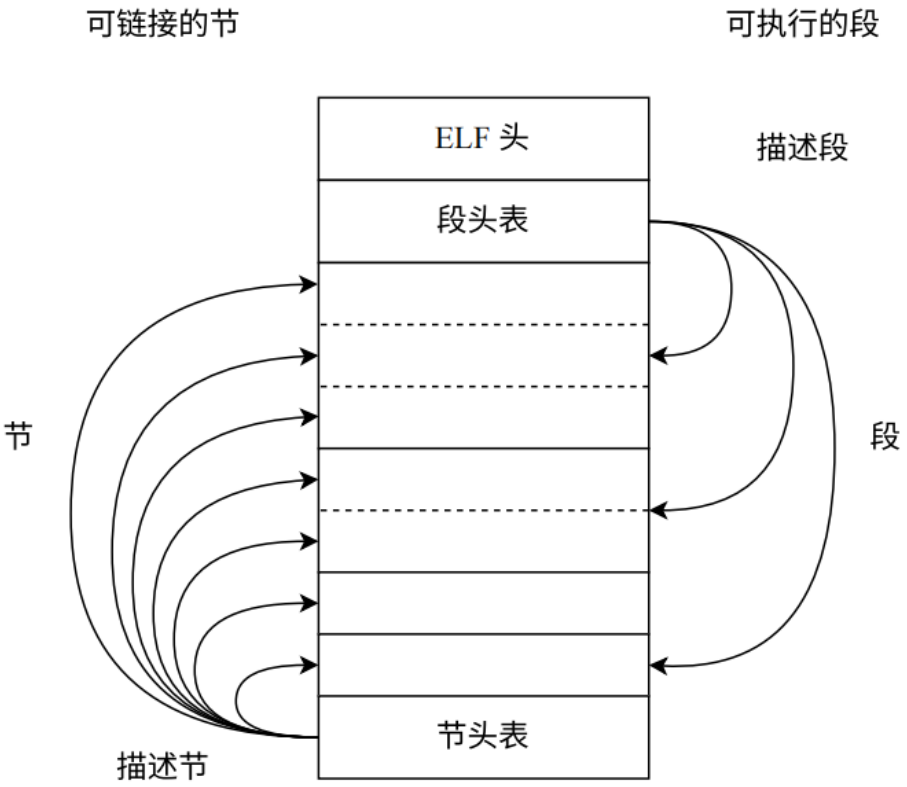


图 1.1: ELF 文件结构

明确在一个ELF文件中，ELF头、段头表（程序头表，program header table）、节头表（section header table）等概念，程序头表中包含`segment`的信息，节头表中包含`section`信息。

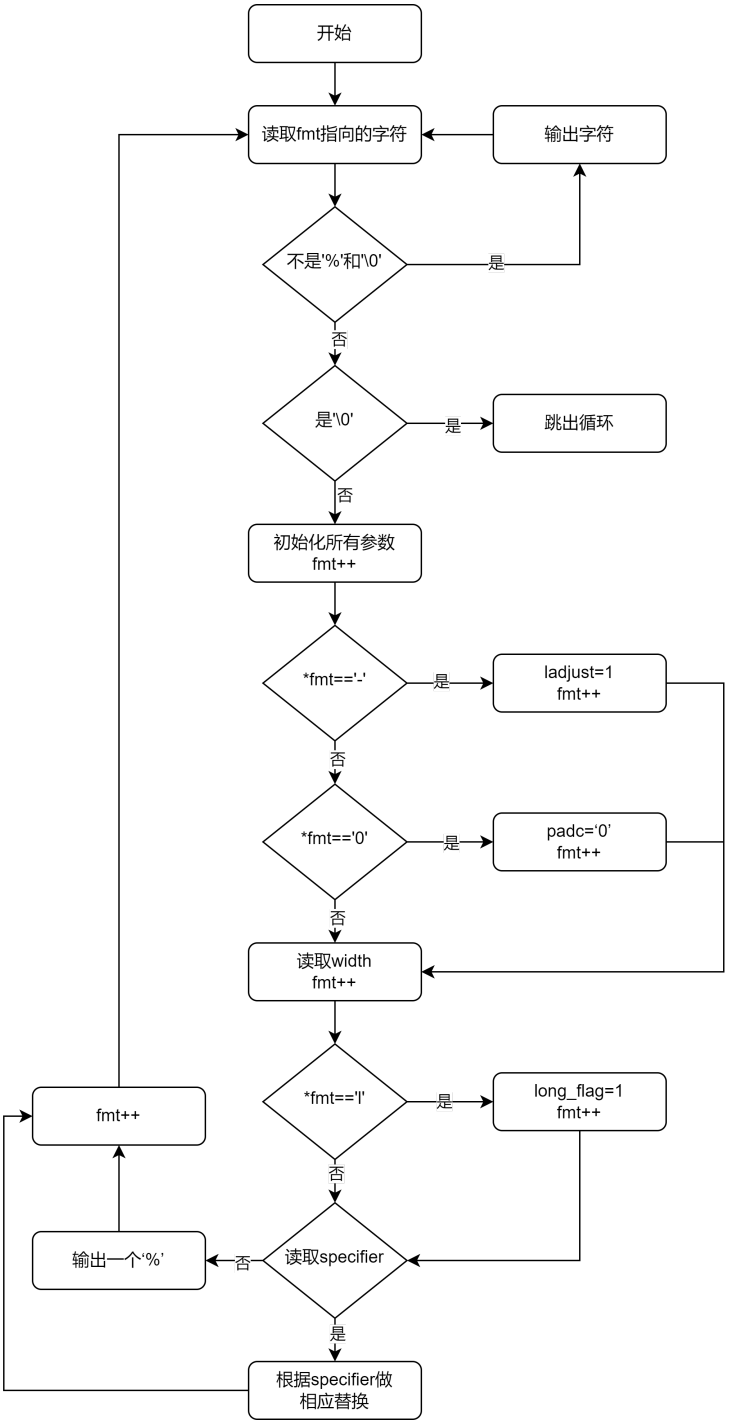
在exercise1.1中，我们需要在给定ELF头地址的情况下，输出文件中所有节头的地址信息。根据`elf.h`文件中ELF头结构体中的内容，我们发现可以在ELF头中找到节头表的地址偏移量和节头的数量以及大小，根据给出的ELF头地址，我们可以访问出以上信息，然后去遍历每一个节头即可。

```
// Get the address of the section table, the number of section headers and the
size of a
// section header.
const void *sh_table;
Elf32_Half sh_entry_count;
Elf32_Half sh_entry_size;
/* Exercise 1.1: Your code here. (1/2) */
sh_table = binary + ehdr->e_shoff;
sh_entry_count = ehdr->e_shnum;
sh_entry_size = ehdr->e_shentsize;
// For each section header, output its index and the section address.
// The index should start from 0.
for (int i = 0; i < sh_entry_count; i++) {
    const Elf32_Shdr *shdr;
    unsigned int addr;
    /* Exercise 1.1: Your code here. (2/2) */
    shdr =(const Elf32_Shdr *)(sh_table + i * sh_entry_size);
    addr = shdr->sh_addr;
    printf("%d:0x%x\n", i, addr);
}
```

我认为这部分内容中的难点在于搞清楚ELF文件的结构，以及去读懂`readelf.c`、`main.c`、`elf.h`文件中代码的功能，然后根据给出的提示将代码补全即可。

`vprintfmt()`函数

我认为补全该函数的难点在于，需要去阅读附录和代码内容，明确printk格式，然后确定函数中是如何解析待输出的字符串的。以下是流程图：



实验体会

这次实验我认为还是有一定难度的，主要在于我对指导书的很多内容在第一次阅读的时候都不太理解，在进行实验的时候几乎都是摸索着去完成的，我认为想要做好这次的实验，必须要将指导书和代码相结合，在学习指导书内容是时候也要自己动手去试一试，明确不同文件之间的关系，然后多加尝试，同时还可以多和同学们进行讨论，互相分享心得。虽然过程比较痛苦，但是在完成实验，看到`print.c`文件正常运行的时候，我还是感到非常开心的，这时候再看一看指导书上的内容，感觉比刚开始理解的也更加深入了。