操作系统 LAB1 实验报告

一． 思考题

1．请阅读附录中的编译链接详解，尝试分别使用实验环境中的原生 x86 工具链（gcc、ld、readelf、objdump 等）和 MIPS 交叉编译工具链（带有 mips-linux-gnu-前缀），重复其中的编译和解析过程，观察相应的结果，并解释其中向 objdump 传入的参数的含义。

原生环境是基于 x86 架构下的，而运用 MIPS 交叉编译工具则是虚拟了一个 MIPS 架构环境，编译链接反汇编等建立在 MIPS 架构下，二者有很大不同。

对于 Helloworld 文件，下图是用 GCC 编译后的结果:

```
 4 .LC0:
 5         .string "Hello,world!"
 6         .text
 7         .globl  main
 8         .type   main, @function
 9 main:
10 .LFB0:
11         .cfi_startproc
12         endbr64
13         pushq   %rbp
14         .cfi_def_cfa_offset 16
15         .cfi_offset 6, -16
16         movq    %rsp, %rbp
17         .cfi_def_cfa_register 6
18         leaq    .LC0(%rip), %rax
19         movq    %rax, %rdi
20         movl    $0, %eax
21         call    printf@PLT
22         movl    $0, %eax
23         popq    %rbp
24         .cfi_def_cfa 7, 8
25         ret
26         .cfi_endproc
27 .LFE0:
28         .size   main, .-main
29         .ident  "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
30         .section        .note.GNU-stack,"",@progbits
31         .section        .note.gnu.property,"a"
32         .align 8
33         .long   1f - 0f
```

下图是使用交叉编译器后的结果:

```
        .file   1 "helloworld.c"
        .section .mdebug.abi32
        .previous
        .nan    legacy
        .module fp=xx
        .module nooddspreg
        .abicalls
        .text
        .rdata
        .align  2
$LC0:
        .ascii  "Hello,world!\000"
        .text
        .align  2
        .globl  main
        .set    nomips16
        .set    nomicromips
        .ent    main
        .type   main, @function
main:
        .frame  $fp,32,$31              # vars= 0, regs= 2/0, args= 16, gp= 8
        .mask   0xc0000000,-4
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
```

Objdump 是反汇编命令，观察 LAB1 内 Makefile 的 objdump 部分，有一些常用的参数如下:

-a 显示档案库的成员信息,类似 ls -l 将 lib*.a 的信息列出。

-l --line-numbers 用文件名和行号标注相应的目标代码

-d 从 objfile 中反汇编那些特定指令机器码的 section。

-D --disassemble-all 与 -d 类似，但反汇编所有 section.

-s --full-contents 显示指定 section 的完整内容。默认所有的非空 section 都会被显示。

一般而言，我们使用 objdump -alds <目标文件或可执行文件名> -o <得到的反汇编文件> 即可满足需求。

2. 思考下述问题：

- 尝试使用我们编写的 readelf 程序，解析之前在 target 目录下生成的内核 ELF 文件。
- 也许你会发现我们编写的 readelf 程序是不能解析 readelf 文件本身的，而我们刚才介绍的系统工具 readelf 则可以解析，这是为什么呢？

内核的节头地址如下图所示：

用真正的 readelf -h 来查看我们编写的 readelf 头：

```
git@21373037:~/21373037/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
```

再来查看 hello 的头:

```
  Section header string table index: 30
git@21373037:~/21373037/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
```

Hello 的 Makefile 如图所示,注意参数-m32:

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

节选自 readelf.c:

```
int is_elf_format(const void *binary, size_t size) {
	Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
	return size >= sizeof(Elf32_Ehdr) && ehdr->e_ident[EI_MAG0] == ELFMAG0 &&
		ehdr->e_ident[EI_MAG1] == ELFMAG1 && ehdr->e_ident[EI_MAG2] == ELFMAG2 &&
		ehdr->e_ident[EI_MAG3] == ELFMAG3;
}
```

综上分析，Hello 在编译的过程中定义了类别为 ELF32，而 readelf 的类别为 ELF64，而可以看到 readelf 解析的是 ELF32 类别的文件，因此 readelf 无法解析它自身。

3. 在理论课上我们了解到，MIPS 体系结构上电时，启动入口地址为 0xBFC00000（其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址），但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？（提示：思考实验中启动过程的两阶段分别由谁执行。）

我们在 kernel.lds 中设置好了各个节被加载的位置，即最终的 segment 地址，同时使用了 ENTRY(_start)指定了程序入口（即内核入口）。而在 stage1 和 2 阶段时由 bootloader 掌管执行，在完成必要的初始化后，会将内核引导装填进内存中的指定位置(即 lds 中设置好的各个节被加载的位置)，最后将 PC 指向_start 内核入口处，将控制权交给内核，最终保证了内核入口能够被正确跳转。

二． 实验难点

本次实验难点有二，其一是有关 ELF 文件的理解，其二是填写 printk 函数。

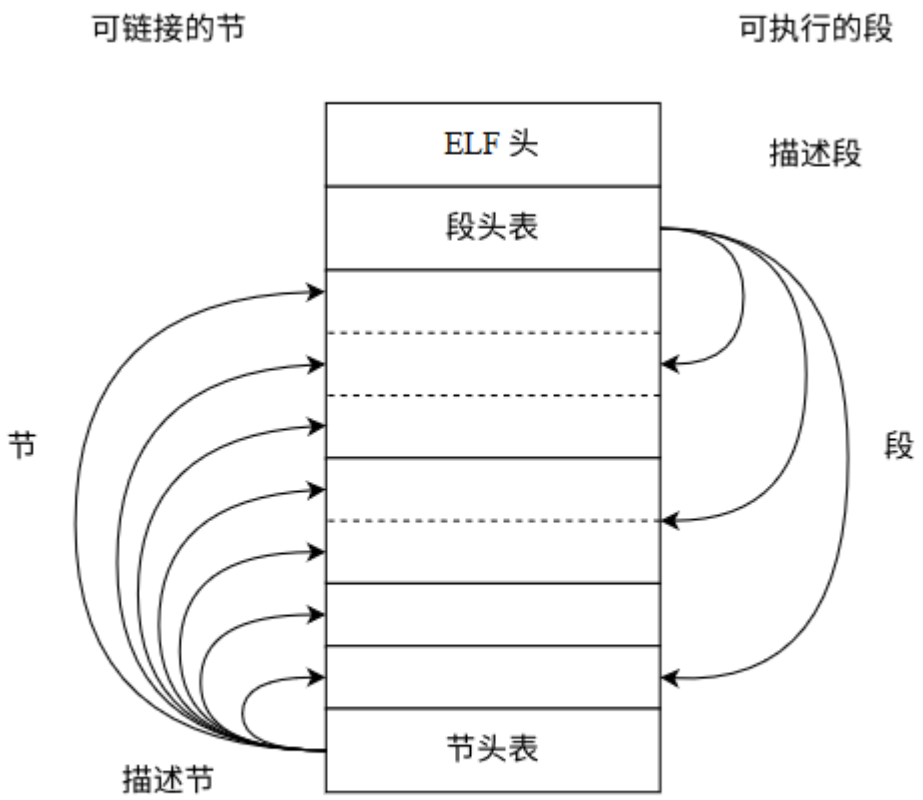


图 1.1: ELF 文件结构

(1)引用教程中的一张图，重点在于理解 ELF 文件内部的结构，首先一个满足条件的程序(可执行，共享，目标)即对应了一个 ELF 文件，他的主要组成包括:

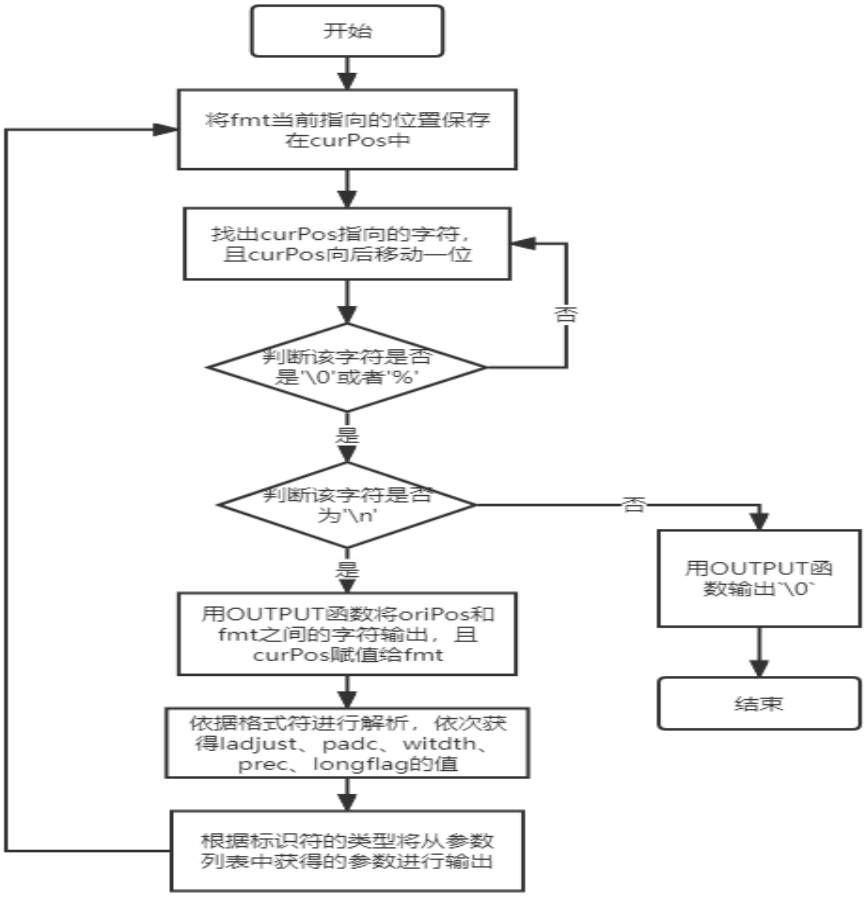
ELF 头，由一个结构体组成，包括程序的基本信息，比如体系结构和操作系统，同时也包含了节头表和段头表相对文件的偏移量 (offset)。

节头表，是由数个节头结构体组成的一个结构体数组(笔者一开始便云里雾里了很久)，每个节头称为一个节头表项，有固定的 size。

段头表也同上，以及各个节的代码数据内容。

我们可根据 ELF 头地址+节头表偏移量寻到节头表的首地址，然后再根据节头表的项数和节头的 size 找到每个节头的首地址，再指向其对应的节地址即可。需要注意对 c 语言中指针的理解，将一个 int 型指针加 1 实际则是在地址上加了 4，**它是以数据类型的大小为单位进行计算的**。同时 const void*是万能指针，由它可以转化为任何类型的指针(加上强转更好)，也可由任何类型的指针转为它，因为指针本质都是一个四字节的地址。

(2)有关 printk 函数的实现，有些与 OO 中 Lexer 词法分析器的实现类似，开始时字符串指针处于字符串头，用指针扫描每个字符，针对扫描到的指定字符进行分析，直至指针扫描到'\0'停止。



值得注意的是在八进制，十六进制，二进制中负数皆被看作补码形式，

而十进制下有关负数的处理不同，它会保留负号输出，因此若从参数列表中提取出来的为负数，则应将其绝对值进行输出，再在前面加上负号。

三．实验总结

本次 LAB1 实验带我们完成了内核的启动，readelf 文件的编写，以及 printk 函数的实现，相比较 LAB0，可以说这才是真正进入了操作系统世界的大门。而随着实验的深入，笔者明显察觉到的是指导书内容篇幅的增加以及读起来会有更多不理解的磕绊，同时 C 语言代码部分难度开始提高，令笔者不得不重新复习一年前学过的数据结构与指针等等基础知识(原来大一上了一年用处在这里)；同时做一道题依赖的文件数也变得连分屏都难以看过来，这令我不得不去尝试 Vscode 的使用，当然还有英文注释带来不方便感。然而我们更愿意相信如今苦难的背后一定是光明，克服烦躁，去享受操作系统带给我们的快乐吧！