Lab1 实验报告

游子诺 2019.3.20

1 实验思考题

Think 1.1

> 也许你会发现我们的readelf 程序是不能解析之前生成的内核文件(内核文件是可执行文件) 的，而我们之后将要介绍的工具readelf 则可以解析，这是为什么呢？(提示：尝试使用readelf -h，观察不同)

通过readelf -h 比较vmlinux和testELF两个文件的文件头差异，我认为无法解析内核文件的原因是因为内核文件中的存储方式是解析程序所不支持的，解析程序只支持小端存储的多字节数据读取，而vmlinux程序则有大端存储的多字节数据。

```
File: testELF
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian

File: vmlinux
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian

```

Think 1.2

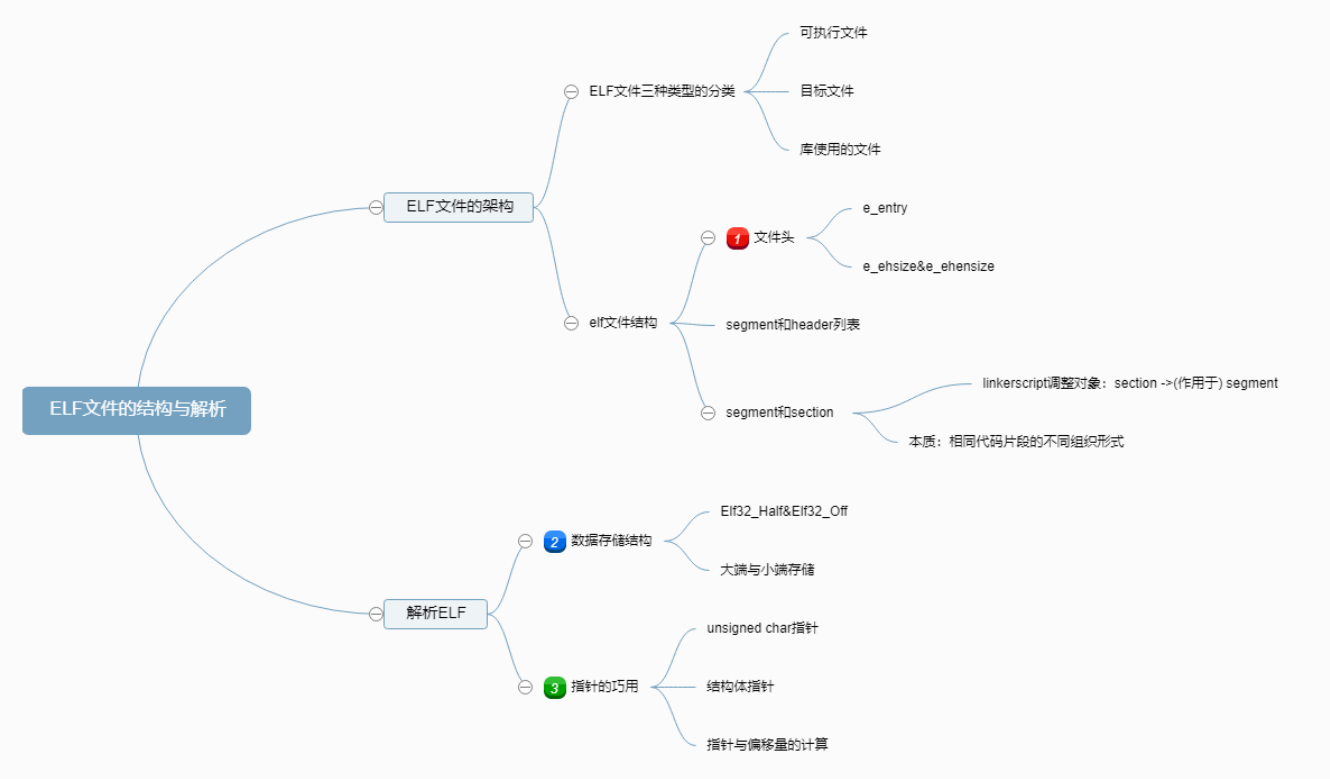
> main 函数在什么地方？我们又是怎么跨文件调用函数的呢？

1. 由顶层的Makefile文件可知，main函数在init目录下的main.c和main.o中。
2. 编译与链接是生成可执行文件时两个重要的步骤，将每个c语言或汇编语言文件编译为obj文件时，start.o文件中跳转到main的代码所对应的地址还是空白的，在顶层Makefile生成vmlinux时才会调用链接器将各个.o文件进行链接，这时的start.o中跳转main的地址就被赋予正确的值，由此完成跨文件调用函数。

---

2 实验难点图示

本次实验我认为比较困难的内容是ELF文件结构的理解和解析程序的编写，下面的思维导图将点出难点，而后将使用选择重要的辨析点进一步阐释说明。



[辨析点一] elf文件头有关数据

ELF文件头Header中的数据有两点需要重点辨析：

1. e_entry：e_entry标志着可执行程序的入口位置，以vmlinux内核为例，gexmul模拟器将其加载至正确的位置后，解析了入口位置，便将CPU的PC指针移动至入口以开始系统的运行。但是请注意，**gexmul模拟内核启动和readelf.c解析ELF是不同的行为**，在解析时，ELF文件在内存中只是一段要被处理的数据，而不是需要被运行，因此在利用Table的偏移量计算Table地址`table_addr = base + offset`时，**基址不是e_entry，而是在C语言解析程序中指向ELF文件的指针。**
2. e_ehsize & e_ehensize: 此处需要明晰entry一词与table之间的关系，前者表示了整个table的大小字节，后者表示了table中一横栏的大小。当逐个遍历table中的entry时，有两种方式进行指针的跳跃：
    1. 以字节为单位，指针每次移动时增大e_ehensize。
    2. 以table对应的entry结构体为单位，指针每次移动时+1即可。

[辨析点二] elf文件存储结构

在实际解析ELF文件的操作中，宏定义的变量和大小端存储是重点需要注意的地方：当ELF文件大小存储方式与解析程序运行环境存在差异时，需要使用辨析点三中的指针技巧进行转换，而转换时的依据又和宏定义变量类型时字节大小有关。

[辨析点三] elf文件解析时指针的巧用

1. u_char指针的使用：unsigned char类型的大小和计算机存储单元字节同等大小，在解析程序中多次**使用u_char指针指向尚未被具体定性的数据块**；同时，u_char指针用于处理多字节数据的大小端存储转换。
2. 结构体指针：结构体指针的使用**实际定义了一种对后续数据的定位和解析方法**：一是使用结构体指针访问结构体中某个数据时，会根据此数据在结构体中排列顺序定位数据的首个字节位置；二是对于多字节数据，会根据数据的字节大小抓取对应的后续字节进行解析。

---

3 体会与感想

本次lab应该算是我接触OS的第一个实验了，经过在Unix系统下编程和调试，我也逐渐地意识到lab0中的各种工具的使用情景和重点方法。

对于本章的知识环节，我花费时间最多去思考的内容是有关elf文件的，回过头来思考自身知识的欠缺，其实主要在编译原理和系统如何运行程序还理解得不够透彻，达到gcc编译器在编译一个可执行程序时所涉及的多个步骤，小到elf文件中各部分数据的组成结构，当从可编辑代码到内存中的已加载好的执行代码这一条过程走完后，许多其他语言（如Java）程序的运行机理就可以通过类比得到了。

对于本章的编程能力环节，我也有机会见识了一番系统工程代的维护和操作方法：定义的各种elf_数据类使得当参数发生变化时仅需调整宏定义即可，指针与结构体让我见识到了指针使用的灵活性，把我带到了一个全靠指针处理内存数据的世界，加深了对象和引用的思想。

当然，经过两次课上测试，我也发现我对练习以外的课程内容和代码没有引起足够的重视，因此需要在今后得以主动加强，对于加强的方法，我认为最重要的是理论要跟上，课上的PPT需要温习，SMRL和教材要拓展性阅读，不是用原来的知识去实现操作系统中某个残缺功能就罢，而是要全方位地弄清楚操作系统。

4&5 指导书反馈与残留难点

[.PHONY] 指令的解释

在指导书1.3.1节中具体地介绍了一个工程型Makefile文件的架构，其中提到了.PHONY的作用是“让指令执行不受时间的约束即一旦被调用总是被执行”，我在此处的理解和尝试遇到了很多的问题，在此想给出以下建议。

1. 明晰Makefile在工程文件修改时维护工程文件的原理：MakeFile在lab0中简短地提到了运用时间戳进行维护，说明了维护所依赖的数据，但没有说清数据怎么用？这对理解目标文件和伪目标文件十分重要。
2. 给.PHONY语句声明的目标定性：.PHONY可避免文件名冲突、可让指令执行不受时间约束、可提升性能……都是语句产生的功能，但我在记忆这些知识时不便于记忆，因为我并没有得到一个定义，也没有发现这些功能的原理和它们之间潜在的关联（逻辑解释不通）。后来经过查阅博客时，有作者将.PHONY后面的目标定义为**伪目标**，其像子程序的函数名一样：无需实际存在的目标，一旦调用则像程序一样一定执行。

> 参考链接：MakeFile教程

> 其中具体讨论了`伪目标和实体目标依赖`、`首位默认的伪目标`、`伪目标的时间戳`等问题，并对每个问题都附有样例和解释。

3. 说明all和无参数时执行第一个目标文件的默认原则。

LinkerScript编写的疑问

1. 启动地址的变更：1.3.3节中提到gxemul加载默认生成的可执行文件时加载位置是不正确的，但是又提到其会根据ELF文件将机器码加载至正确地址，并从ELF文件中知晓程序入口，位置可以自定义加载，入口也可以自定义指定，看起来内核代码在哪里执行没有什么约束呀？**“不正确”**的根源究竟在哪里呢？

   是硬件上的约束？：gxemul模拟时内存段有功能要求？

   是软件上的约束？：编写的vmlinux内核中有绝对地址的情况？

   期待解答！

2. .bss, .text, .data段加载的顺序：指导书中并没有说明在创建自定义LinkerScript时三段加载的顺序要求，我自行尝试了三种顺序都能够通过测评，但不是太明确其中的原理是什么，“究竟是那一步**自适应地**解决了这样的问题”？