lab1实验报告

1. 思考题：

Thinking 1.1

> Thinking 1.1 在阅读 附录中的编译链接详解 以及本章内容后，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，如 mips-linux-gnu-gcc、mips-linux-gnu-ld），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。

- objdump命令的常用参数如下：

参数	含义
-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

- 先新建一个文件 `hello.c` ：

```c
#include <stdio.h>
int main() {
        printf("Hello World:\n");
        return 0;
}
```

- 然后使用原生 x86 工具链编译和解析：

  编译并链接：

```
# 生成目标文件
gcc -c hello.c -o hello_x86.o
# 生成可执行文件
gcc hello_x86.o -o hello_x86
```

  使用 `objdump` 反汇编：

```
objdump -d hello_x86.o      # 反汇编目标文件的代码段
objdump -d hello_x86        # 反汇编可执行文件的代码段
```

  可以得到：

```
git@23371331:~/test $ objdump -d hello_x86.o

hello_x86.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
```

```
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 05 ac 0e 00 00    lea    0xeac(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    1158:       48 89 c7                mov    %rax,%rdi
    115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
    1160:       b8 00 00 00 00          mov    $0x0,%eax
    1165:       5d                      pop    %rbp
    1166:       c3                      ret

Disassembly of section .fini:
```

Thinking 1.2

Thinking 1.2 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（提示：尝试使用readelf-h，并阅读tools/readelf 目录下的 Makefile，观察 readelf 与 hello 的的不同）

- 使用编写的 `readelf` 解析内核 `ELF` 文件 `mos`

```
git@23371331:~/23371331/tools/readelf (lab1)$ ./readelf ../../target/mos
0:0x0
1:0x80020000
2:0x800220b0
3:0x800220c8
4:0x800220e0
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
```

- 使用编写的 `readelf` 文件不能解析本身

```
git@23371331:~/23371331/tools/readelf (lab1)$ ./readelf readelf
git@23371331:~/23371331/tools/readelf (lab1)$
```

但是系统工具 `readelf` 可以

```
git@23371331:~/23371331/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
```

接着再解析 `hello` 文件

通过观察 `readelf` 和 `hello` 类别，发现前者是64位文件而后者是32位文件，所以我们自己编写的 `readelf` 不能解析自己，只能解析32位文件 `hello` 。

Thinking 1.3

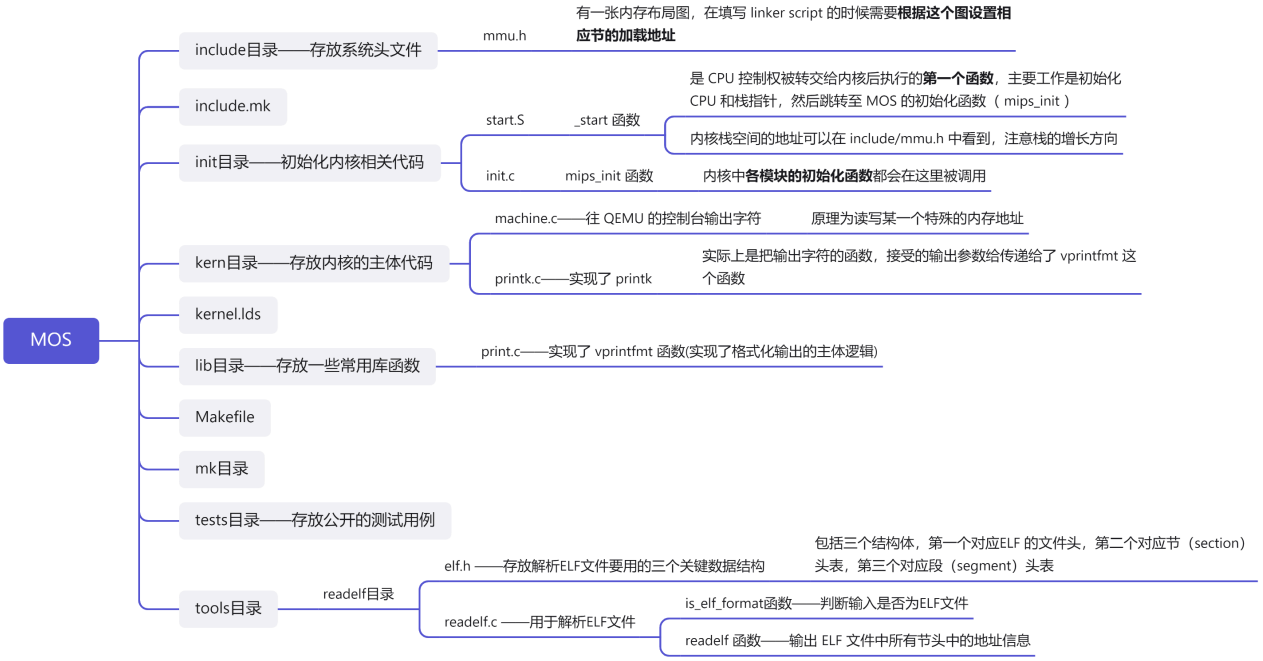
> Thinking1.3在理论课上我们了解到，MIPS体系结构上电时，启动入口地址为0xBFC00000（其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址），但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？（提示：思考实验中启动过程的两阶段分别由谁执行。）

- 在MIPS架构中，尽管硬件上电时从固定地址（如0xBFC00000）启动，但内核仍能正确跳转到其入口地址的原因在于启动过程分为两个阶段，并通过链接脚本（Linker Script）控制内存布局。
- 启动过程的两阶段
  - 第一阶段：Bootloader/模拟器初始化
    - 硬件启动地址（如0xBFC00000）：此处存放的是Bootloader或固件代码（如实验中的GXemul模拟器逻辑）。
    - Bootloader的作用：初始化硬件，将内核的可执行文件从存储介质（如磁盘）加载到内存的指定位置。
    - 模拟器的简化流程：GXemul直接解析ELF格式的内核文件，根据ELF头信息将其代码段、数据段等加载到链接脚本指定的内存地址，无需手动拷贝。
  - 第二阶段：内核执行

    - 跳转到内核入口：Bootloader/模拟器完成加载后，主动跳转到内核的入口地址（如`_start`函数），该地址由链接脚本定义，与上电启动地址无关。
- 链接器根据脚本将内核的各个段（`.text`、`.data`等）分配到指定地址，生成的可执行文件（ELF）中会记录这些地址信息。ELF文件的入口地址（`e_entry`）被设为`_start`，Bootloader/模拟器跳转至此地址。内核的物理地址由链接脚本和加载器（Bootloader/模拟器）共同保证，只要加载到指定位置，无论原始上电地址如何，均可正确执行。

2. 难点分析

- 实验的目的：
    - 从操作系统角度理解MIPS 体系结构
    - 完成 `printk` 函数的`k`的编写
- 因为此前对于数据类型的理解不够灵活，导致我理解这一部分花了好些时间。我认为这里的理解关键在于，**数据类型并不是某段内存数据的固有属性，而是对它的解释方式。**只要符合类型的大小和对齐要求，就可以把这段数据看做某种类型。
- 操作系统的启动时，在`QEMU`中的模拟器，启动流程被简化为加载内核到内存，之后跳转到内核的入口。实验时难点是利用代码上下部分完成对`k`来输出键值对。而主要的代码架构如下：



3. 实验体会

- 实验的`exam`部分较为简单，主要分为三个部分，打印键即输出字符串，接着输出`>=`字符串，最后在打印数字即可。只需要依次利用`%s`和`%d`即可。在ELF文件的解析过程中，用到了3个重要的结构体，相当于定义了3种新的数据类型（实际不止3种，其他部分代码里也有定义）。而在实际编写中，就涉及到数据类型的转换。比如在做Exercise 1.1的时候，把 void *类型的 binary 强制转换为了 ELF32_Ehdr * 。因为此前对于数据类型的理解不够灵活，导致我理解这一部分花了好些时间。数据类型并不是某段内存数据的固有属性，而是对它的解释方式。只要符合类型的大小和对齐要求，就可以把这段数据看做某种类型。
-

```
s = (char *)va_arg(ap, char *);
```

```
print_str(out, data, s, width, ladjust);
print_str(out, data, tt, 4, ladjust);
if (long_flag) {
    num = va_arg(ap, long int);
} else {
    num = va_arg(ap, int);
}
if (num < 0) {
    num = -num;
    neg_flag = 1;
}
print_num(out, data, num, 10, neg_flag, width, ladjust, padc, 0);
break;
```

- 然而实验的 `extra` 比较困难，需要补全四个函数，这次我并没有作出这部分，但是仍然理解到对不同部分文件的理解。`vprintfmt` 函数实现的是格式化输出的主体逻辑，被 `printk` 调用，参数列表中的ap参数也是从上层函数中得来。结合 `printk` 的用法，比如 `printk("%d%c%ld", a, b, c)`，很容易明白可变参数就是需要输出的一系列变量，而 `fmt` 就是引号中的字符串。