OS —— Lab1实验报告

22373407 王飞阳

一、思考题

Thinking 1.1

编译与反汇编(原生x86工具链)

main.c

```
#include<stdio.h>
int main(){
        printf("hello objdump");
        return 0;
}
```

反汇编 main.o (直截取main代码段)

```
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:   b8 00 00 00 00          mov    $0x0,%eax
  17:   e8 00 00 00 00          call   1c <main+0x1c>
  1c:   b8 00 00 00 00          mov    $0x0,%eax
  21:   5d                      pop    %rbp
  22:   c3                      ret
```

反汇编 main (直截取main代码段)

```
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:       b8 00 00 00 00          mov    $0x0,%eax
    1160:       e8 eb fe ff ff          call   1050 <printf@plt>
    1165:       b8 00 00 00 00          mov    $0x0,%eax
    116a:       5d                      pop    %rbp
    116b:       c3                      ret
```

可以发现，`call` 后面已经被填入了一个地址

编译与反汇编(MIPS交叉编译工具链)

main.c

```
#include<stdio.h>
int main(){
        printf("hello objdump");
        return 0;
}
```

反汇编 main.o (直截取main代码段)

```
00000000 <main>:
   0:   27bdffe0        addiu   sp,sp,-32
   4:   afbf001c        sw      ra,28(sp)
   8:   afbe0018        sw      s8,24(sp)
   c:   03a0f025        move    s8,sp
  10:   3c1c0000        lui     gp,0x0
  14:   279c0000        addiu   gp,gp,0
  18:   afbc0010        sw      gp,16(sp)
  1c:   3c020000        lui     v0,0x0
  20:   24440000        addiu   a0,v0,0
  24:   8f820000        lw      v0,0(gp)
  28:   0040c825        move    t9,v0
  2c:   0320f809        jalr    t9
  30:   00000000        nop
  34:   8fdc0010        lw      gp,16(s8)
  38:   00001025        move    v0,zero
  3c:   03c0e825        move    sp,s8
  40:   8fbf001c        lw      ra,28(sp)
  44:   8fbe0018        lw      s8,24(sp)
  48:   27bd0020        addiu   sp,sp,32
  4c:   03e00008        jr      ra
  50:   00000000        nop
```

反汇编 main (直截取main代码段)

```
004006e0 <main>:
  4006e0:       27bdffe0        addiu   sp,sp,-32
  4006e4:       afbf001c        sw      ra,28(sp)
  4006e8:       afbe0018        sw      s8,24(sp)
  4006ec:       03a0f025        move    s8,sp
  4006f0:       3c1c0042        lui     gp,0x42
  4006f4:       279c9010        addiu   gp,gp,-28656
  4006f8:       afbc0010        sw      gp,16(sp)
  4006fc:       3c020040        lui     v0,0x40
  400700:       24440830        addiu   a0,v0,2096
  400704:       8f828030        lw      v0,-32720(gp)
  400708:       0040c825        move    t9,v0
  40070c:       0320f809        jalr    t9
  400710:       00000000        nop
  400714:       8fdc0010        lw      gp,16(s8)
  400718:       00001025        move    v0,zero
  40071c:       03c0e825        move    sp,s8
```

```
400720:       8fbf001c        lw      ra,28(sp)
400724:       8fbe0018        lw      s8,24(sp)
400728:       27bd0020        addiu   sp,sp,32
40072c:       03e00008        jr      ra
400730:       00000000        nop
```

在链接时进行了重定位，`main` 地址不再是0，而是0x4006e0

objdump 传入参数意义

`-D : Display assembler contents of all sections`，反汇编所有节的内容

`-S : Intermix source code with disassembly`，显示与反汇编结合的源代码

Thinking 1.2

1. 使用自编 `readelf` 解析 `mos` ELF 文件

解析结果

```
0:0x0
1:0x80020000
2:0x80022060
3:0x80022078
4:0x80022090
5:0x0
6:0x0
7:0x0
8:0x0
9:0x0
10:0x0
11:0x0
12:0x0
13:0x0
14:0x0
15:0x0
16:0x0
```

2. 不同的原因

在Makefile中，有两个目标：`readelf`和`hello`。它们的编译指令有明显的差别：

- `readelf`是通过链接`main.o`和`readelf.o`对象文件来生成的，没有指定任何特殊的编译选项。
- `hello`是直接从`hello.c`编译生成的，但它使用了`-m32`、`-static`，这个选项指示编译器生成32位代码，且意味着`hello`程序是静态链接的。

我们自己写的`readelf`程序没有被设计来支持32位的ELF文件，那么它可能无法解析用这个选项编译生成的程序；且我们的`readelf`主要支持解析动态链接的程序，那么它可能无法正确解析静态链接的程序。

而系统提供的`readelf`工具是一个成熟、全面的工具，它支持解析多种类型的ELF文件，包括但不限于32位与64位、静态与动态链接，以及包含各种节和段信息的ELF文件。它能够解析的广泛性部分来源于其对ELF格式深入和全面的支持。

Thinking 1.3

因为`QEMU`模拟器支持直接加载ELF格式的内核，也就是说，`QEMU`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MOS操作系统的运行第一行代码前，我们就已经拥有一个正常的程序运行环境，而只需要将内核加载后跳转到内核函数入口就可以启动完毕。

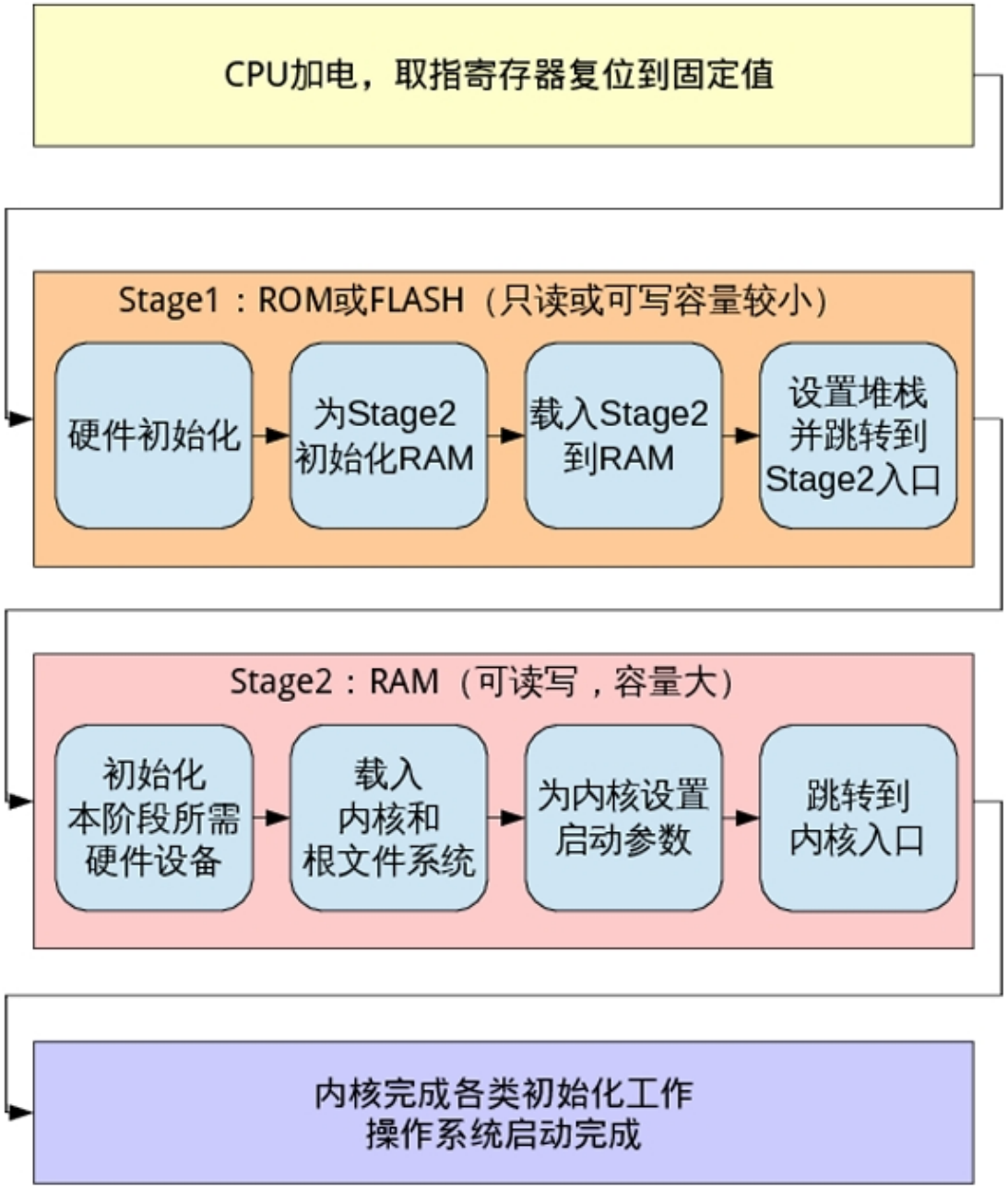
在链接器中，我们指定了内核加载的地址，并通过`start.S`中的代码，初始化硬件设备，设置堆栈入口，然后跳转到了内核函数入口处。

二、难点分析

通过阅读内核实验指导书以及对 Lab1 的完成，个人认为本实验的难点如下：

难点一：操作系统的启动

真实的操作系统的内核以及启动较为复杂。实验中我们采用`QEMU`模 拟器上直接运行`MOS`操作系统，使得操作系统的启动部分大大简化。真实的操作系统启动基本步骤如下图表示：

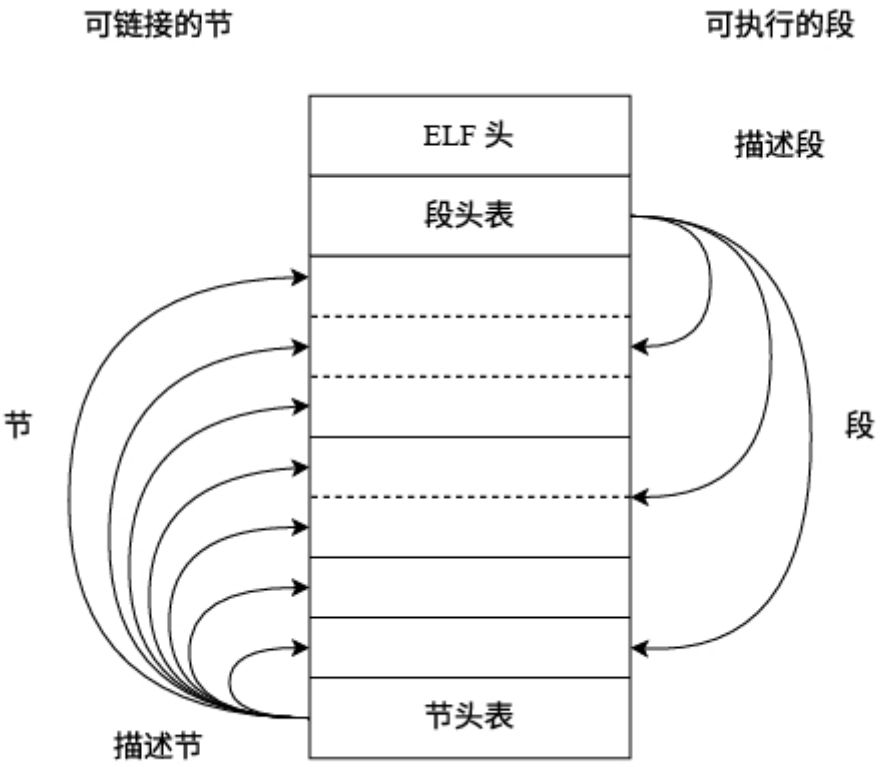


难点二：ELF 文件的结构和功能

ELF 文件的结构解析和功能理解也是本次实验的一大重点。

- **ELF文件的结构**

ELF文件的结构可表示如下：



1. **EFL 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**：段头表（或程序头表，主要包含程序中各个段的信息， 段的信息需要在运行时刻使用。段头表中的每一个表项，记录了该段数据载入内存时的目标位置等，记录了用于指导应用 程序加载的各类信息。
3. **节头表**：节头表，主要包含程序中各个节的信息，节的信息需要 在程序编译和链接的时候使用。节头表中的每一个表项，记录了该节程序的代码段、数据段等各个段的内容，主要是链接 器在链接的过程中需要使用。

- **功能**

1. 组成可重定位文件，参与可执行文件和可共享文件的链接。此时使用节头表。
2. 组成可执行文件或者可共享文件，在运行时为加载器提供信息。此时使用段头表。

难点三：`printk` 函数的编写

此部分的主要难点在于对实验关键代码的阅读和理解，以及C语言编写程序的练习（注意对题目所给函数中参数的初始化）

三、实验体会

本次 Lab1 实验相较于 Lab0 实验来讲，难度提升了一些。在刚开始阅读内核实验指导书时，理解起来较为困难。但随着反复的阅读和查阅资料，以及实验题目、思考题目的完成，让我逐渐对整个项目有了初步理解。

以下是对整个 Lab1 的结构理解：

整个实验以总的 `Makefile` 为基础，将各个文件夹串联起来：

- `init` 目录：内核初始化相关代码
- `include` 目录：存放系统头文件
- `lib` 目录：存放一些常用的库函数，包括 `vprintfmt`
- `kern` 目录：存放内核的主体代码
- `tests` 目录：存放测试程序
- `tools` 目录：存放一些实用工具，包括 `readelf` ，该目录下的 C 程序使用原生工具链构建（而非交叉编译），在宿主环境（而非 QEMU）下运行
- `target` 目录：存放编译的产物
- `Makefile` ：用于编译 MOS 内核的 `Makefile` 文件

以上就是整个 Lab1 的文件结构，在梳理过后会清晰很多，同时也方便之后实验的理解。