Lab1实验报告

思考题

Thinking 1.1

- 对于 `objdump` 指令
  - `-D`：反汇编所有 section 的内容
  - `-S`：反汇编的时候一同输出源代码

```
git@22375080:~ $ objdump --help
用法：objdump <选项> <文件>
 显示来自目标 <文件> 的信息。
 至少必须给出以下选项之一：
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -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>
  -s, --full-contents      Display the full contents of all sections requested
  -Z, --decompress         Decompress section(s) before displaying their contents
  -g, --debugging          Display debug information in object file
  -e, --debugging-tags     Display debug information using ctags style
  -G, --stabs              Display (in raw form) any STABS info in the file
  -W, --dwarf[a/=abbrev, A/=addr, r/=aranges, c/=cu_index, L/=decodedline,
              f/=frames, F/=frames-interp, g/=gdb_index, i/=info, o/=loc,
              m/=macro, p/=pubnames, t/=pubtypes, R/=Ranges, l/=rawline,
              s/=str, O/=str-offsets, u/=trace_abbrev, T/=trace_aranges,
              U/=trace_info]
                           Display the contents of DWARF debug sections
  -Wk,--dwarf=links        Display the contents of sections that link to
                            separate debuginfo files
  -WK,--dwarf=follow-links
                           Follow links to separate debug info files (default)
  -WN,--dwarf=no-follow-links
                           Do not follow links to separate debug info files
```

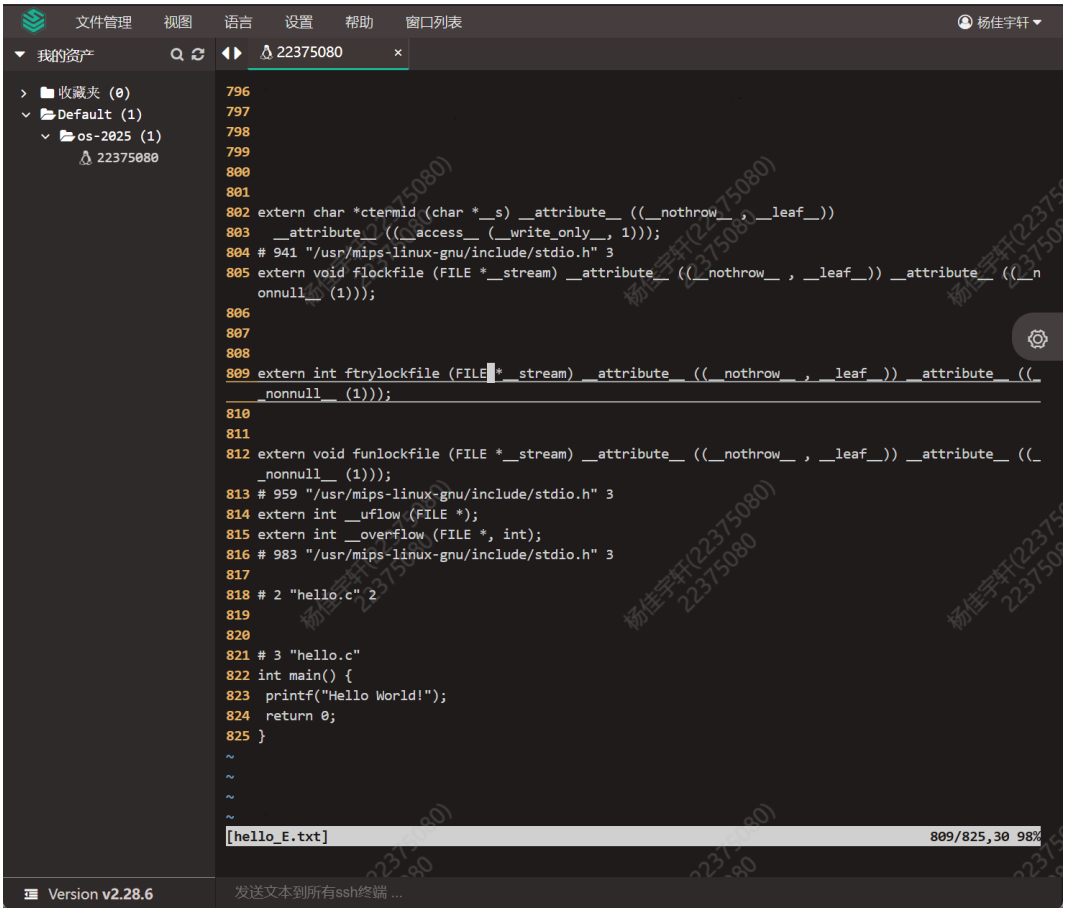
- 实验
  - `hello.c` 文件

```c
#include <stdio.h>

int main() {
        printf("Hello World!");
        return 0;
}
```

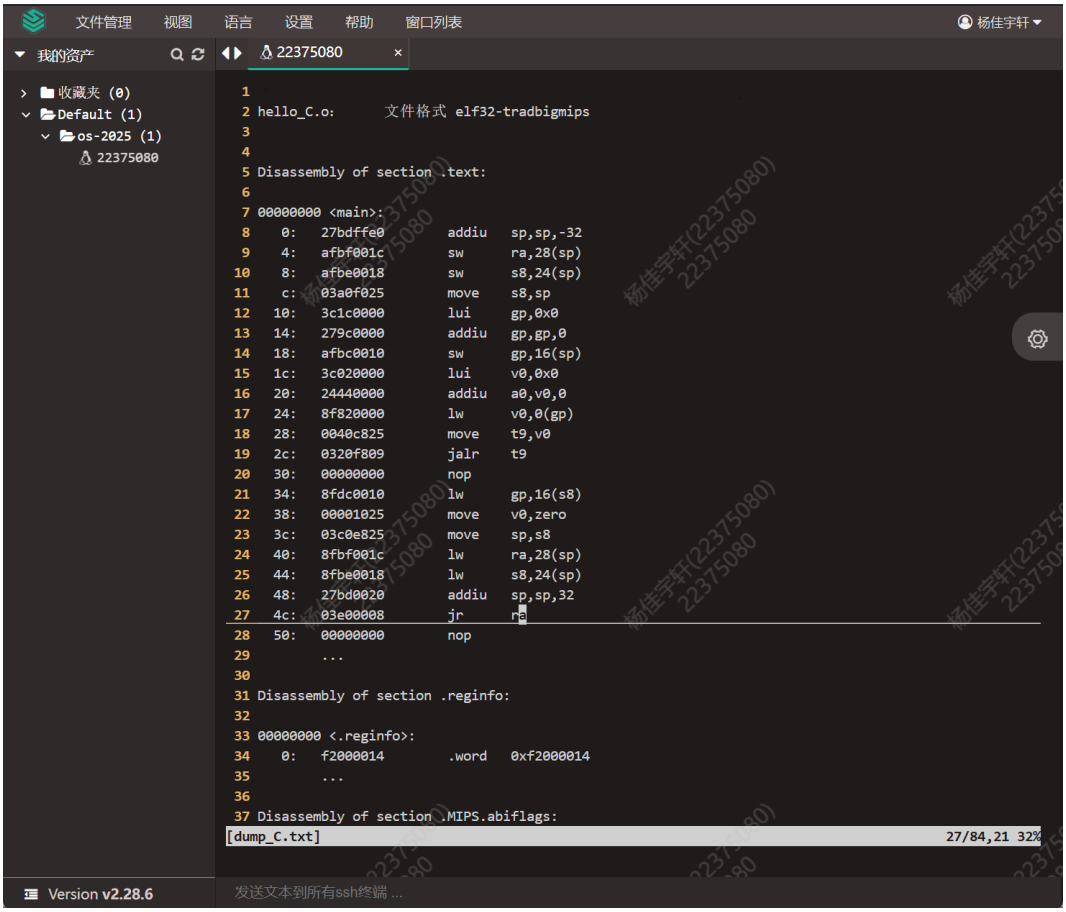
- `Makefile` 文件

```
CROSS_COMPILE := mips-linux-gnu-
CC            := $(CROSS_COMPILE)gcc
LD            := $(CROSS_COMPILE)ld
OBJDUMP       := $(CROSS_COMPILE)objdump
TARGET        := hello.c

all:
        $(CC) -E $(TARGET) > hello_E.txt
        $(CC) -c $(TARGET) -o hello_C.o
        $(OBJDUMP) -DS hello_C.o > dump_C.txt
        $(CC) $(TARGET) -o hello
        $(OBJDUMP) -DS hello > dump_EXE.txt
```

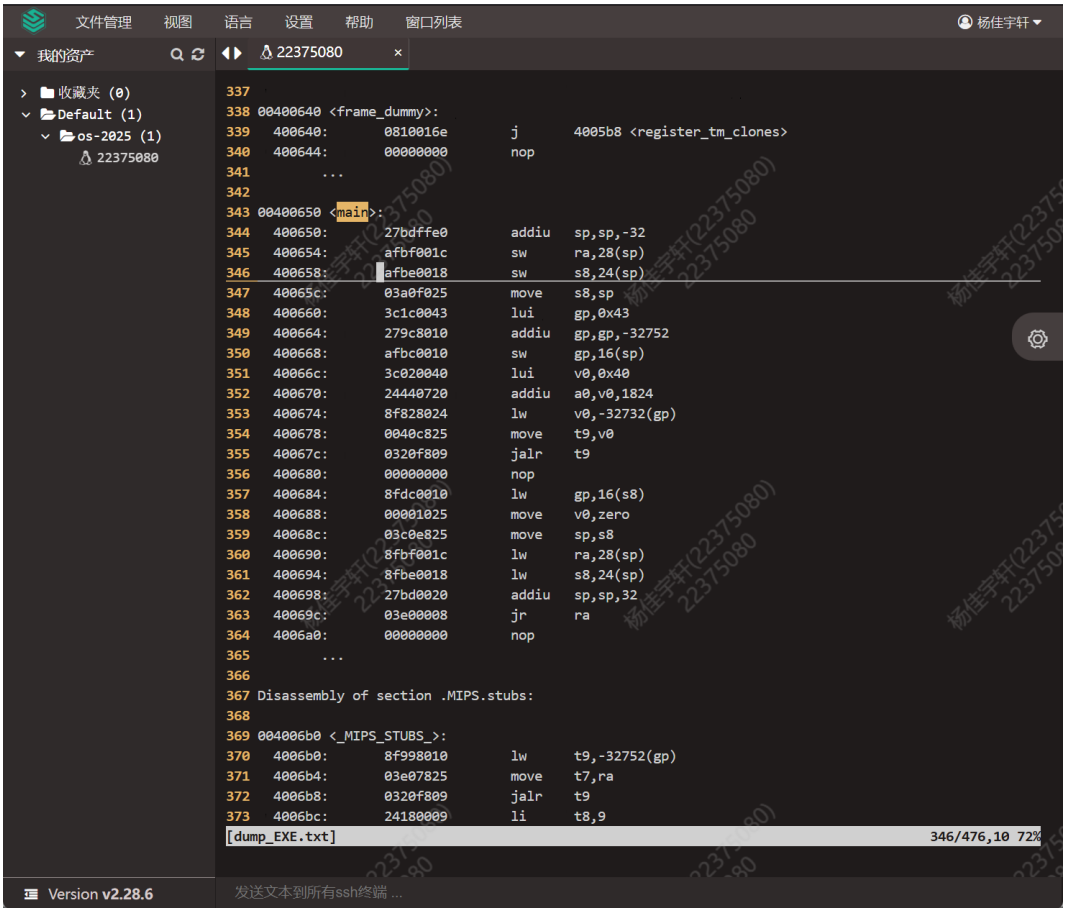
- `-E` 预处理



- `-c` 只编译不链接，之后进行目标文件反汇编



- 编译链接后进行反汇编



- 对两次的函数比较可以发现，函数调用的地址在链接后进行了全局指针的传递分配

`dump_C.txt`

```
24:    8f820000        lw      v0,0(gp)
28:    0040c825        move    t9,v0
2c:    0320f809        jalr    t9
```

`dump_EXE.txt`

```
400674:        8f828024        lw      v0,-32732(gp)
400678:        0040c825        move    t9,v0
40067c:        0320f809        jalr    t9
```

Thinking 1.2

- 使用 `readelf` 解析
  - 使用系统工具 `readelf` 进行解析

```
git@22375080:~/22375080 (lab1)$ readelf -S ./target/mos
There are 19 section headers, starting at offset 0x4f30:

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        80020000 0000c0 001930 00 WAX  0   0 16
  [ 2] .reginfo          MIPS_REGINFO    80021930 0019f0 000018 18   A  0   0  4
  [ 3] .MIPS.abiflags    MIPS_ABIFLAGS   80021948 001a08 000018 18   A  0   0  8
  [ 4] .rodata           PROGBITS        80021960 001a20 000230 00   A  0   0 16
  [ 5] .pdr              PROGBITS        00000000 001c50 000280 00      0   0  4
  [ 6] .comment          PROGBITS        00000000 001ed0 000026 01  MS  0   0  1
  [ 7] .gnu.attributes   GNU_ATTRIBUTES  00000000 001ef6 000010 00      0   0  1
  [ 8] .debug_info       MIPS_DWARF      00000000 001f06 001017 00      0   0  1
  [ 9] .debug_abbrev     MIPS_DWARF      00000000 002f1d 0006e3 00      0   0  1
  [10] .debug_aranges    MIPS_DWARF      00000000 003600 000100 00      0   0  8
  [11] .debug_line       MIPS_DWARF      00000000 003700 000aac 00      0   0  1
  [12] .debug_str        MIPS_DWARF      00000000 0041ac 0004fe 01  MS  0   0  1
  [13] .debug_frame      MIPS_DWARF      00000000 0046ac 00033c 00      0   0  4
  [14] .debug_rnglists   MIPS_DWARF      00000000 0049e8 000016 00      0   0  1
  [15] .debug_line_str   MIPS_DWARF      00000000 0049fe 000020 01  MS  0   0  1
  [16] .symtab           SYMTAB          00000000 004a20 000320 10     17  29  4
  [17] .strtab           STRTAB          00000000 004d40 000124 00      0   0  1
  [18] .shstrtab         STRTAB          00000000 004e64 0000cc 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)
```

  - 使用我们编写的 `readelf` 尝试解析

```
git@22375080:~/22375080/tools/readelf (lab1)$ ./readelf ../../target/mos
0:0x0
1:0x80020000
2:0x80021930
3:0x80021948
4:0x80021960
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
18:0x0
```

- 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?
  - `readelf -h readelf`

```
git@22375080:~/22375080/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
```

  - `readelf -h hello`

```
git@22375080:~/22375080/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:               0x8049750
  程序头起点:          52 (bytes into file)
  Start of section headers:          707128 (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:         30
  Section header string table index: 29
```

可以发现，`hello` 文件使用了 `-m32 -static -g` 编译选项，是 `ELF32` 类型的，而 `readelf` 是 `ELF64` 类型的

说明我们的 `hello` 文件是 32 位格式，但 `readelf` 文件是 64 位的，而 `readelf.c` 文件中的所有数据类型都是基于 32 位进行设置的

- `-m32`：编译出来的文件为 32 位程序，可以在 32 位操作系统以及 64 位操作系统运行

Thinking 1.3

- MIPS系统启动时由 `Bootloader` 接管，通过存放的一小段代码进行硬件初始化，其运行并通过 `Linker Script` 映射形成内存映像，内核文件也在这时得知需要被加载的合适的地址

  所以内核代码会被加载到 `0x80010000` 这段地址，之后通过 `ENTRY(_start)` 便可以跳转到内核运行位置

实验难点

- QEMU 支持记载 ELF 格式内核，所以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，启动就完成了
- 内核镜像文件 mos 的 make 过程，如何进行交叉编译
- 实验代码中各个目录的组织以及其中的重要文件

- 根目录下还存在 `kernel.lds` 这个 linker script 文件，我们会在下面的小节中详细讲解。
- `init` 目录中主要有两个代码文件 `start.S` 和 `init.c`，其作用是初始化内核。`start.S` 文件中的 `_start` 函数是 CPU 控制权被转交给内核后执行的第一个函数，主要工作是初始化 CPU 和栈指针，为之后的内核初始化做准备，最后跳转到 `init.c` 文件中定义的 `mips_init` 函数。在本章中 `mips_init` 函数只是简单的打印输出，而在之后的实验中会逐步添加新的内核功能，内核中各模块的初始化函数都会在这里被调用。
- `include` 目录中存放系统头文件。在本章中需要用到的头文件是 `mmu.h` 文件，这个文件中有一张内存布局图，我们在填写 linker script 的时候需要根据这个图来设置相应节的加载地址。
- `lib` 目录存放一些常用库函数，本章中主要存放用于格式化输出的函数。
- `kern` 目录中存放内核的主体代码，本章中主要存放的是终端输出相关的函数。
- `tests` 目录中存放公开的测试用例，我们在进行本地测试时会用到它。

- 理解交叉编译的过程以及使用方式
- ELF 文件整体上分为 5 个部分

  段头表在运行时刻使用

  节头表在编译和链接时候使用

  1. ELF 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（或程序头表，program header table），主要包含程序中各个段（segment）的信息，段的信息需要在运行时刻使用。
  3. 节头表（section header table），主要包含程序中各个节（section）的信息，节的信息需要在程序编译和链接的时候使用。
  4. 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。
  5. 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。

- ELF 的文件头就是一个存 ELF 文件信息的结构体
- 内核通过 `virtAddr` 指示的地址加载到正确的位置上，内核就可以运行起来
- 我们将内核放在 `kseg0` 段，运行在 `kseg1` 中的 `bootloader` 在载入内核前会进行 cache 初始化
- 程序执行的第一条指令的地址被称为入口地址（Entrypoint）

  我们实验中的 `kernel.lds` 中通过 `ENTRY(_start)` 来设置如何为 `_start`

体会与感想

Lab1 并不是过一次就能够完全掌握的，我相信之后的实验也是这样。对于第一次跟着 guidebook 过一次，只能够进行简单的 Exercise 解题。但更重要的是做完之后进行不断的回顾，才能知道每一步之间的连接关系，为什么以及如何做

Thinking 部分我是在第二次阅读 guidebook 的时候完成的，相较于第一次，对于整体的结构有了更多的知解

操作系统凝结了前人的智慧，我们需要不断地一步一步理解它