

# lab1

## 思考题

### Thinking 1.1

Q: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，观察相应地结果，并解释其中向 objdump 传入的参数的含义。

A:

上方为只经过编译的文件的反汇编结果

下方为经过编译和链接之后的文件的反汇编结果

- x86工具链(重复附录中的操作)

```
1 []
2 hello.o:      文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8    0: f3 0f 1e fa        endbr64
9    4: 55                 push   %rbp
10   5: 48 89 e5          mov    %rsp,%rbp
11   8: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # f <main+0xf>
12   f: 48 89 c7          mov    %rax,%rdi
13  12: e8 00 00 00 00     call   17 <main+0x17>
14  17: b8 00 00 00 00     mov    $0x0,%eax
15  1c: 5d                 pop    %rbp
16  1d: c3                 ret
17
18 Disassembly of section .rodata:
19
20 0000000000000000 <.rodata>:
"gcc_2" 96L, 3726B
21
22 0000000000001140 <frame_dummy>:
23  1140: f3 0f 1e fa        endbr64
24  1144: e9 77 ff ff ff        jmp    10c0 <register_tm_clones>
25
26 0000000000001149 <main>:
27  1149: f3 0f 1e fa        endbr64
28  114d: 55                 push   %rbp
29  114e: 48 89 e5          mov    %rsp,%rbp
30  1151: 48 8d 05 ac 0e 00 00  lea    0xeac(%rip),%rax      # 2004 <_IO_
+0x4>
31  1158: 48 89 c7          mov    %rax,%rdi
32  115b: e8 f0 fe ff ff        call   1050 <puts@plt>
33  1160: b8 00 00 00 00     mov    $0x0,%eax
34  1165: 5d                 pop    %rbp
35  1166: c3                 ret
36
37 Disassembly of section .fini:
38
39 0000000000001168 <fini>:
```

- MIPS交叉编译工具链

```

1 []
2 hello.o:      文件格式 elf32-tradbigmips
3
4
5 Disassembly of section .text:
6
7 00000000 <main>:
8    0: 27bdffe0      addiu   sp,sp,-32
9    4: afbf001c      sw      ra,28(sp)
10   8: afbe0018      sw      s8,24(sp)
11   c: 03a0f025      move    s8,sp
12  10: 3c1c0000      lui     gp,0x0
13  14: 279c0000      addiu   gp,gp,0
14  18: afbc0010      sw      gp,16(sp)
15  1c: 3c020000      lui     v0,0x0
16  20: 24440000      addiu   a0,v0,0
17  24: 8f820000      lw      v0,0(gp)
18  28: 0040c825      move    t9,v0
19  2c: 0320f809      jalr   t9
20  30: 00000000      nop

```

1,0-1

发送文本到所有ssh终端 ...

```

381
382 004006e0 <main>:
383 4006e0: 27bdffe0      addiu   sp,sp,-32
384 4006e4: afbf001c      sw      ra,28(sp)
385 4006e8: afbe0018      sw      s8,24(sp)
386 4006ec: 03a0f025      move    s8,sp
387 4006f0: 3c1c0042      lui     gp,0x42
388 4006f4: 279c9010      addiu   gp,gp,-28656
389 4006f8: afbc0010      sw      gp,16(sp)
390 4006fc: 3c020040      lui     v0,0x40
391 400700: 24440830      addiu   a0,v0,2096
392 400704: 8f828030      lw      v0,-32720(gp)
393 400708: 0040c825      move    t9,v0
394 40070c: 0320f809      jalr   t9
395 400710: 00000000      nop
396 400714: 8fdc0010      lw      gp,16(s8)
397 400718: 00001025      move    v0,zero
398 40071c: 03c0e825      move    sp,s8
399 400720: 8fbf001c      lw      ra,28(sp)
400 400724: 8fbe0018      lw      s8,24(sp)

```

```

1 ~/test_lab1 $ mips-linux-gnu-gcc -E hello.c > mips_1
2
3 ~/test_lab1 $ mips-linux-gnu-gcc -c hello.c
4 ~/test_lab1 $ mips-linux-gnu-objdump -DS hello.o > mips_2
5
6 ~/test_lab1 $ mips-linux-gnu-gcc -o misp.out hello.c
7 ~/test_lab1 $ mips-linux-gnu-objdump -DS mips.out > mips_3

```

- objdump参数含义:

- d, --disassemble      Display assembler contents of executable sections  
反汇编那些特定指令机器码的section
- D, --disassemble-all      Display assembler contents of all sections  
反汇编所有的section
- disassemble=      Display assembler contents from
- S, --source      Intermix source code with disassembly
- source-comment[=]      Prefix lines of source code with
- s, --full-contents      Display the full contents of all sections requested

# Thinking 1.2

Q: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。

A:

```
git@22373474:~/22373474 (lab1)$ ./tools/readelf/readelf ./target/mos
0:0x0
1:0x80400000
2:0x804016f0
3:0x80401708
4:0x80401720
5:0x0
6:0x0
7:0x0
8:0x0
9:0x0
10:0x0
11:0x0
12:0x0
13:0x0
14:0x0
15:0x0
16:0x0
```

```
git@22373474:~/22373474 (lab1)$ readelf -S ./target/mos
There are 17 section headers, starting at offset 0x49cc:
Section Headers:
  [Nr] Name           Type        Addr     Off      Size     ES Flg Lk Inf Al
  [ 0] .null         PROGBITS    00000000 00000000 00000000 00      0 0 0 0
  [ 1] .text          PROGBITS    80400000 0000c0 0016f0 00 WAX 0 0 16
  [ 2] .reginfo       MPTPS_REGINFO 804016f0 0017b0 000018 18 A 0 0 4
  [ 3] .MIPS.abiflags MIPS_ABIFLAGS 80401708 0017c8 000018 18 A 0 0 8
  [ 4] .rodata         PROGBITS    80401720 0017e0 000230 00 A 0 0 16
  [ 5] .pdr           PROGBITS    00000000 001a10 000280 00 0 0 4
  [ 6] .comment        PROGBITS    00000000 001c90 000025 01 MS 0 0 1
  [ 7] .gnu.attributes GNU_ATTRIBUTES 00000000 001cb5 000010 00 0 0 1
  [ 8] .debug_info     MIPS_DWARF 00000000 001cc5 000f9b 00 0 0 1
  [ 9] .debug_abbrev   MIPS_DWARF 00000000 002c60 0005c8 00 0 0 1
  [10] .debug_aranges  MIPS_DWARF 00000000 003228 000100 00 0 0 8
  [11] .debug_line     MIPS_DWARF 00000000 003328 00099a 00 0 0 1
  [12] .debug_str      MIPS_DWARF 00000000 003c2c 000502 01 MS 0 0 1
  [13] .debug_frame    MIPS_DWARF 00000000 0041c4 000338 00 0 0 4
  [14] .symtab         SYMTAB     00000000 0044fc 000300 10 15 27 4
  [15] .strtab         STRTAB     00000000 0047fc 000124 00 0 0 1
  [16] .shstrtab       STRTAB     00000000 004920 0000ac 00 0 0 1

Key to Flags:
```

- 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` 的不同）

A:

## readelf头部文件信息：

```
git@22373474:~/22373474/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
```

hello头部文件信息：

```
git@22373474:~/22373474/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
```

系统工具readelf头部文件信息：

```
git@22373474:~/test_lab1 $ readelf -a $(which 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: 0x55310
  程序头起点: 64 (bytes into file)
  Start of section headers: 778816 (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: 30
```

tools/readelf/Makefile中的readelf与hello:

```
1 readelf: main.o readelf.o
2     $(CC) $^ -o $@
3 readelf: main.o readelf.o
4     $(CC) $^ -o $@
5 hello: hello.c
6     $(CC) $^ -o $@ -m32 -static -g
```

hello在编译链接时添加了-m32 -static -g选项，使得生成的目标文件是32位、静态链接并且包含调试信息的。32位指的是指令集架构和内存地址的位数。我们可以看到hello的头部文件信息中类别为**ELF32**，而默认情况下，在64位操作系统上使用GCC编译器生成的可执行文件通常是64位的，故我们课程组提供的readelf头部文件信息中类别为**ELF64**。

当一个 ELF 文件的类型被标识为 ELF32 时，表示该文件采用了32位的标识符、地址和偏移等字段来描述文件的各个部分，如节表、程序头部、符号表等。ELF64 同理。

但是在课程组提供的readelf.c文件中，表示ELF头部信息的结构体大小为32位，故课程组的readelf可以用来解析hello而不可以用来解析它自身。

如果课程组的readelf的编译方式和hello一样，即如下述形式，则readelf\_temp可以解析自身。

```

git@22373474:~/22373474/tools/readelf (lab1)$ make readelf_temp
cc main.c readelf.c -o readelf_temp -m32 -static -g
git@22373474:~/22373474/tools/readelf (lab1)$ readelf_temp readelf_temp
-bash: readelf_temp: 未找到命令
git@22373474:~/22373474/tools/readelf (lab1)$ ./readelf_temp readelf_temp
0:0x0
1:0x8048134
2:0x8048158
3:0x8048178
4:0x8049000
5:0x8049028
6:0x80490a0
7:0x80b80c0
8:0x80b8c68
9:0x80b9000
10:0x80d5080
11:0x80ead4c

```

而之所以系统工具 readelf 都能够解析，是因为 readelf 是一个通用的 ELF 文件解析工具，可以处理不同版本的 ELF 文件。当用户使用 readelf 命令解析一个 ELF 文件时，readelf 会根据文件的格式自动识别是 ELF32 还是 ELF64，并相应地解析文件的结构和内容。

```

1 //readelf.c
2 int is_elf_format(const void *binary, size_t size) {
3     Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
4     return size >= sizeof(Elf32_Ehdr) && ehdr->e_ident[EI_MAG0] == ELFMAG0 &&
5         ehdr->e_ident[EI_MAG1] == ELFMAG1 && ehdr->e_ident[EI_MAG2] ==
6         ELFMAG2 &&
7         ehdr->e_ident[EI_MAG3] == ELFMAG3;

```

## Thinking 1.3

**Q:** 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。）

**A:** 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，内存和一些外围设备都可以正常使用。QEMU 支持加载 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。

1. Makefile 中使用的 \$(link\_script) 来生成的内核

```

1 link_script      := kernel.lds
2
3 $(mos_elf): $(modules)
4     $(LD) $(LDFLAGS) -o $(mos_elf) -N -T $(link_script) $(objects)

```

2. kernel.lds 文件中有 ENTRY(\_start) 这一行命令，这就是内核的入口

3. start.s 中 sp 寄存器设置到内核栈空间的位置上，随后跳转到 mips\_init 函数（C 语言的主函数）

## 难点分析

# 1.0 常用操作

```
1 objdump -DS 要反汇编的目标文件名 > 导出文本文件名  
2 file 要查看类型的目标文件名 #获得文件类型  
3 grep -R printcharc #-R的意思是在多个子目录中找
```

## 1.1 ELF相关

- MemSiz与FileSize(ELF头文件中段相关信息)

Offset:该段 (segment) 的数据相对于 ELF文件的偏移。

VirtAddr:该段最终需要被加载到内存的那个位置。

FileSize:该段的数据在文件中的长度。

MemSiz:该段的数据在内存中所应当占的大小。

MemSiz永远大于等于FileSize。若MemSiz大于FileSize，则操作系统在加载程序的时候，会首先将文件中记录的数据加载到对应的VirtAddr处。之后，向内存中填0，直到该段在内存中的大小达到MemSiz为止。那么为什么MemSiz有时候会大于FileSize 呢？这里举这样一个例子：C语言中未初始化的全局变量，我们需要为其分配内存，但它又不需要被初始化成特定数据。因此，在可执行文件中也只记录它需要占用内存(MemSiz)，但在文件中却没有相应的数据（因为它并不需要初始化成特定数据）。故而在这种情况下，MemSiz会大于FileSize。这也解释了，为什么C语言中全局变量会有默认值0。这是因为操作系统在加载时将所有未初始化的全局变量所占的内存统一填了0。

- ELF头部标识符

当一个 ELF 文件的类型被标识为 ELF32 时，表示该文件采用了32位的标识符、地址和偏移等字段来描述文件的各个部分，如节表、程序头部、符号表等。 ELF64 同理。

## 1.2 printk实现相关

- 可变参数列表

当函数参数列表末尾有省略号时，该函数即有变长的参数表。

由于需要定位变长参数表的起始位置，函数需要含有至少一个固定参数，且变长参数必须在参数表的末尾。

stdarg.h 头文件中为处理变长参数表定义了一组宏和变量类型如下：

va\_list, 变长参数表的变量类型；

va\_start(va\_list ap, lastarg), 用于初始化变长参数表的宏；

va\_arg(va\_list ap, 类型), 用于取变长参数表下一个参数的宏；

va\_end(va\_list ap), 结束使用变长参数表的宏。

- 回调函数

回调函数就是一个参数，将这个函数作为参数传到另一个函数里面，当那个函数执行完之后，再执行传进去的这个函数。这个过程就叫做回调。

```
1 //定义主函数，回调函数作为参数  
2 function A(callback) {  
3     callback();
```

```
4     console.log('我是主函数');
5 }
6
7 //定义回调函数
8 function B(){
9     setTimeout("console.log('我是回调函数')", 3000); //模仿耗时操作
10 }
11
12 //调用主函数，将函数B传进去
13 A(B);
14
15 //输出结果
16 我是主函数
17 我是回调函数
```

## 1.3 tmux操作

```
1 tmux      #启动
2 tmux set mouse on #嘿嘿启用鼠标
3 exit      #退出/ctrl+d
```

- 会话

```
1 $ tmux new -s <session-name>      #新建会话
2 $ tmux detach                      #分离会话/ctrl+b d
3 $ tmux ls
4 # or
5 $ tmux list-session                #查看所有会话
6 # 使用会话编号
7 $ tmux attach -t 0
8
9 # 使用会话名称
10 $ tmux attach -t <session-name>    #接入会话
11 # 使用会话编号
12 $ tmux kill-session -t 0
13
14 # 使用会话名称
15 $ tmux kill-session -t <session-name>  #杀死会话
16 # 使用会话编号
17 $ tmux switch -t 0
18
19 # 使用会话名称
20 $ tmux switch -t <session-name>      #切换会话
21 $ tmux rename-session -t 0 <new-name> #重命名会话
```

- 窗口

```

1 $ tmux new-window
2
3 # 新建一个指定名称的窗口
4 $ tmux new-window -n <window-name>          #新建窗口
5 # 切换到指定编号的窗口
6 $ tmux select-window -t <window-number>
7
8 # 切换到指定名称的窗口
9 $ tmux select-window -t <window-name>          #切换窗口

```

- 窗格

```

1 # 划分上下两个窗格
2 $ tmux split-window
3
4 # 划分左右两个窗格
5 $ tmux split-window -h          #划分窗格

```

## 实验体会

lab1上机题目本身难度不大

- lab1-exam:更改输出格式

考察点在于读入具有变长参数列表的函数参数

- lab1-extra:实现简化的scanf(%d %x %s %c)

考察点在于编写具有变长参数列表的函数，练习使用回调函数。

编写逻辑并不复杂，识别百分号后的输入控制符规定的格式→根据格式获取指针→由回调函数in从控制台读取参数→存入指针指向的内存区域。

但是很可惜，没有通过lab1-extra。原因也很可惜，主要有两个问题，第二个十分严重的问题没有来得及在课上解决。。。

第一个问题:判断是否有负号之后，没有再读入下一个字符。

第二个问题:在明晃晃的注释下我又从控制台读取了下一个字符。

```

295     while (*fmt) {
296         if (*fmt == '%') {
297             ret++;
298             fmt++; // 跳过 '%'
299             do {
300                 in(data, &ch, 1);
301             } while (ch == ' ' || ch == '\t' || ch == '\n'); // 跳过空白符
302             // 注意，此时 ch 为第一个有效输入字符
303             switch (*fmt) {
304                 case 'd': // 十进制
305                     // Lab 1-Extra: Your code here. (2/5)
306                     neg = 0;
307                     ip = (int*)va_arg(ap,int*);
308
309                     //in(data, &ch, 1);
310                     if (ch == '-') {
311                         neg = 1;
312                         in(data,&ch,1);
313                     }

```

309行注释掉的部分是实验课上错误的实现方式，每一种case都犯了类似的错误。

希望下次会做得好一些吧

# Reference

---

[可能是东半球最全面易懂的 Tmux 使用教程！\(强烈建议收藏\)-腾讯云开发者社区-腾讯云\(tencent.com\)](#)