

Lab1实验报告

Lab1实验报告

实验思考题

Thinking 1.1

Thinking 1.2

Thinking 1.3

Thinking 1.4

Thinking 1.5

Thinking 1.6

实验难点图示

Exercise 1.2

Exercise 1.5

体会与感想

实验思考题

Thinking 1.1

1. `objdump -DS` 输出可执行文件名 > 导出文本文件名

`-D, --disassemble-all` Display assembler contents of all sections 反汇编所有section

`-S, --source` Intermix source code with disassembly 将源代码与反汇编混合，尽可能反汇编出源代码

2. 课程平台的 MIPS 交叉编译器重复上述各步编译过程，如下图所示：

- 命令行内容

```

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage
Last login: Sun Mar 20 21:26:56 2022 from 10.134.170.231
git@19241027:~$ cd ..
git@19241027:/home$ cd ..
git@19241027:~$ ls
OSLAB bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var
git@19241027:~$ cd /OSLAB/compiler/usr/bin/
git@19241027:/OSLAB/compiler/usr/bin$ ls
cramfsck mips-linux-as mips-linux-gdbtui mips-linux-readelf mips_4KC-c++filt mips_4KC-objdump mkimage
depmod.pl mips-linux-c++ mips-linux-gprof mips-linux-rpmbuild mips_4KC-cpp mips_4KC-ranlib rpm2cpio
fix-embedded-paths mips-linux-c++filt mips-linux-gstack mips-linux-run mips_4KC-g++ mips_4KC-readelf rpmbuild
gcov mips-linux-cpp mips-linux-ld mips-linux-size mips_4KC-gcc mips_4KC-rpmbuild rpmbuild
gendiff mips-linux-g++ mips-linux-ldd mips-linux-strings mips_4KC-gdb mips_4KC-run rpmquery
genext2fs mips-linux-gcc mips-linux-make mips-linux-strip mips_4KC-ld mips_4KC-size rpmsign
jffs2reader mips-linux-gcc-4.0.0 mips-linux-nm mips_4KC-addr2line mips_4KC-ldd mips_4KC-strings rpmverify
makeinfo mips-linux-gccbug mips-linux-objcopy mips_4KC-ar mips_4KC-make mips_4KC-strip texi2dvi
mips-linux-addr2line mips-linux-gcov mips-linux-objdump mips_4KC-as mips_4KC-nm mkcramfs texindex
mips-linux-ar mips-linux-gdb mips-linux-ranlib mips_4KC-c++ mips_4KC-objcopy mkfs.jffs2
git@19241027:/OSLAB/compiler/usr/bin$ d ../../../../
-bash: d: command not found
git@19241027:/OSLAB/compiler/usr/bin$ cd ../../../../
git@19241027:~$ ls
OSLAB bin boot dev etc home lib lib32 lib64 libx32 media mnt opt proc root run sbin srv sys tmp usr var
git@19241027:~$ cd home/git/19241027/
git@19241027:~/19241027$ ls
Makefile boot drivers gxemul hello.c hello.o hello.txt include include.mk init lib readelf tools
git@19241027:~/19241027$ /OSLAB/compiler/usr/bin/mips-linux-gcc -c hello.c
mips-linux-gcc: CROSSL_COMPILE variable is not set or invalid
git@19241027:~/19241027$ /OSLAB/compiler/usr/bin/mips_4KC-gcc -c hello.c
git@19241027:~/19241027$ ls
Makefile boot drivers gxemul hello.c hello.o hello.txt include include.mk init lib readelf tools
git@19241027:~/19241027$ /OSLAB/compiler/usr/bin/mips_4KC-objdump -DS hello.o > hello.txt
git@19241027:~/19241027$ vim hello.txt

```

o hello.c 中内容

```

1 int main()
2 {
3     int a = 2333, b = 6666;
4     int c = a + b;
5     return 0;
6 }

```

"hello.c" 6L, 73C

o 反汇编出的内容

```

1
2 hello.o: file format elf32-tradbigmips
3
4 Disassembly of section .text:
5
6 00000000 <main>:
7 0: 27bdffe0 addiu sp,sp,-32
8 4: afbe0018 sw s8,24(sp)
9 8: 03a0f021 move s8,sp
10 c: 2402001d li v0,2333
11 10: afc20010 sw v0,16(s8)
12 14: 24021a0a li v0,6666
13 18: afc2000c sw v0,12(s8)
14 1c: 8fc30010 lw v1,16(s8)
15 20: 8fc2000c lw v0,12(s8)
16 24: 00621021 addu v0,v1,v0
17 28: afc20008 sw v0,8(s8)
18 2c: 00001021 move v0,zero
19 30: 03c0e821 move sp,s8
20 34: 8fbc0018 lw s8,24(sp)
21 38: 27bd0020 addiu sp,sp,32
22 3c: 03e00008 jr ra
23 40: 00000000 nop
24 ...
25 Disassembly of section .reginfo:
26
27 00000000 <.reginfo>:
28 0: e000000c sc zero,12(zero)
29 ...
30 Disassembly of section .pdr:
31
32 00000000 <.pdr>:
33 0: 00000000 nop
34 4: 40000000 mfc0 zero,c0_index
"hello.txt" 52L, 1248C
1,0-1 Top

```

Thinking 1.2

用 `readelf -h` 解析testELF与vmlinux文件，解析结果分别如下

```

git@19241027:~/19241027$ readelf -h ./readelf/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
  Version:     1 (current)
  OS/ABI:      UNIX - System V
  ABI Version: 0
  Type:        EXEC (Executable file)
  Machine:     Intel 80386
  Version:     0x1
  Entry point address: 0x8048490
  Start of program headers: 52 (bytes into file)
  Start of section headers: 4440 (bytes into file)
  Flags:       0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 9
  Size of section headers: 40 (bytes)
  Number of section headers: 30
  Section header string table index: 27
git@19241027:~/19241027$

```

```

git@19241027:~/19241027$ readelf -h ./gxemul/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
  Version:            1 (current)
  OS/ABI:             UNIX - System V
  ABI Version:        0
  Type:               EXEC (Executable file)
  Machine:            MIPS R3000
  Version:            0x1
  Entry point address: 0x80010000
  Start of program headers: 52 (bytes into file)
  Start of section headers: 37148 (bytes into file)
  Flags:              0x1001, noreorder, o32, mips1
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 2
  Size of section headers: 40 (bytes)
  Number of section headers: 14
  Section header string table index: 11
git@19241027:~/19241027$
  
```

由解析结果可看出，testELF 文件为小端存储，vmlinux 为大端存储，而我们的 readelf 文件只能对小端存储的文件进行解析，因此无法解析vmlinux。

Thinking 1.3

一般情况下，计算机系统刚上电时是没有正常的 C 语言环境，不能读取解析 ELF 文件。所以需要从电启动地址开始，依靠 bootloader 引导操作系统内核启动。

我们的 MOS 操作系统的目标是在 GXemul 仿真器上运行，GXemul 仿真器支持直接加载 ELF 格式的内核，也就是说，GXemul 已经提供了 bootloader 全部功能。GXemul 支持加载 ELF 格式内核，所以启动流程中 stage1 不需要执行，stage2 被简化为加载内核到内存，之后跳转到内核的入口即可。能保证内核入口被正确跳转到。

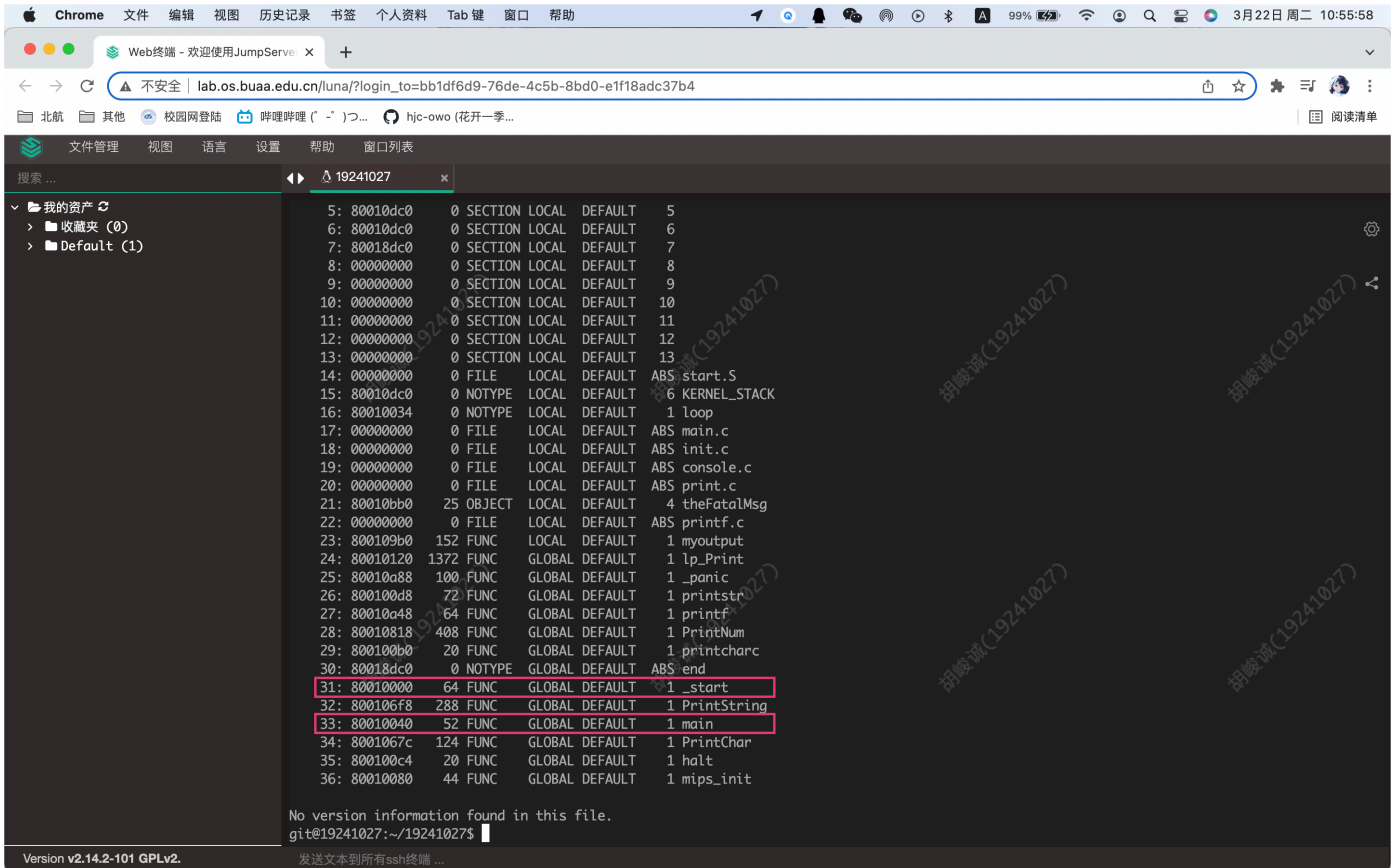
Thinking 1.4

如果 S1 和 S2 两个段在某页的中间相接，先后被加载，如果每次都获取新页，会导致这页对应虚地址关联到了两个实页。

加载段的时候，如果两边不是页对齐的，那么会先检查这个地址是否已经被映射到一个实地址，如果没有映射到实地址才申请新页，防止为一个地址申请多个页产生冲突。

Thinking 1.5

通过命令 `readelf -a gxemul/vmlinux` 可知，如图所示
 内核的入口 `_start` 的位置在 `0x80010000` 处，main 函数的位置在 `0x80010040` 处。



通过 exercise 1.4 可知，在 start.S 文件里，通过跳转指令 jal 跳转到 main 函数地址进入。
在跨文件调用函数时，每个函数会被分配一个固定的地址，调用过程为 将需要存储的值进行保护（比如入栈等方式），再用 jal 跳转到相应函数的地址。

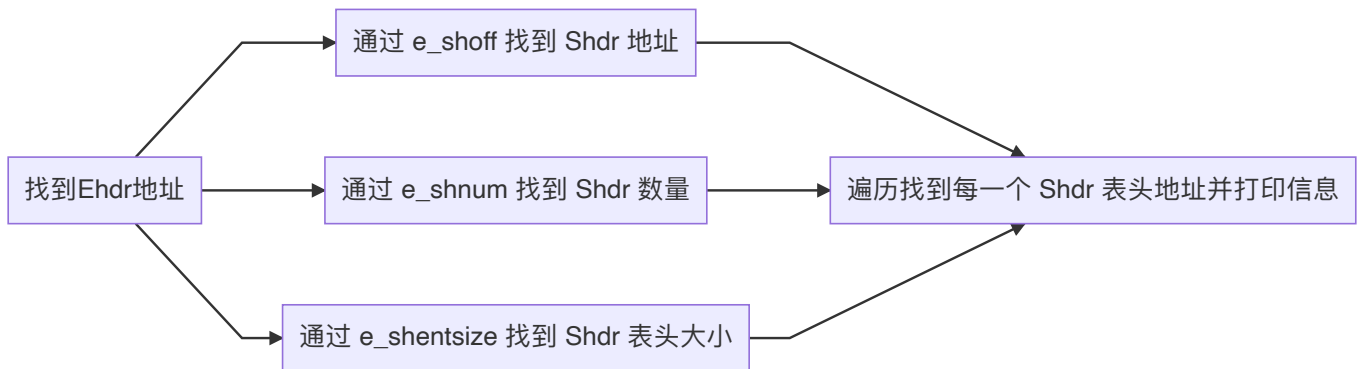
Thinking 1.6

```
1  /* Disable interrupts 禁用中断 */
2  mtc0 zero, CP0_STATUS # 把 CPU 通用寄存器 zero 的内容传送到协处理器 0 寄存器 CP0_STATUS
3  .....
4  /* disable kernel mode cache 禁用内核模式缓存 */
5  mfc0 t0, CP0_CONFIG # 通用寄存器 t0 装入 CPU 控制寄存器 CP0_CONFIG 的值
6  and t0, ~0x7 # ~0x7 给出要清零的位的反码，将 t0 相应位清零（清零低 3 位）
7  ori t0, 0x2 # 0x2 给出要置 1 的位，将 t0 相应位置 1（将第 2 位置 1）
8  mtc0 t0, CP0_CONFIG # 把 CPU 通用寄存器 t0 的内容传送到协处理器 0 寄存器 CP0_CONFIG
9  #将 CP0_CONFIG 寄存器后三位设定为 010，此时 coherency algorithm = uncached，即在内核态模
   式中禁用缓存。
```

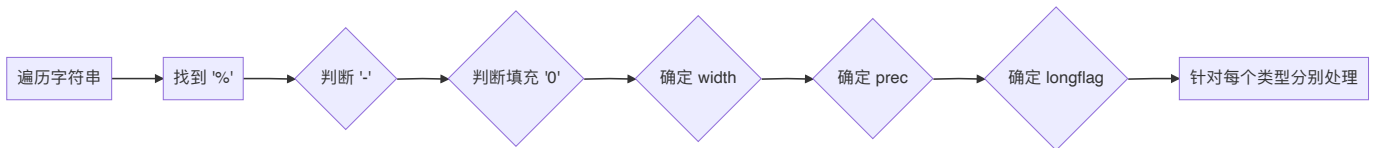
实验难点图示

Exercise 1.2

readelf 函数需要输出 ELF 文件的所有 section header 的序号和地址信息的流程：



Exercise 1.5



体会与感想

这次实验，让我对 ELF 文件有了更深刻的认识。同时自己实现 `printf` 函数的相应功能，要求我们对本身的 `printf` 函数比较熟悉。同时想要完成本次实验需要阅读很多的代码。可能这是我们第一次自行阅读一个工程文件，对于各种文件的寻找还是磕磕绊绊的。回想起刚开始拿到代码的时候，根本不知道从何下手，还是让我心有余悸。

感觉这次实验的难度比 lab0 的难度提升不少。按照指导书的话来说，就是：

相信聪明的你一定已经知道我是什么意思了吧～