OSlab1实验报告

Thinking1.1

请查阅并给出前述objdump 中使用的参数的含义。使用其它体系结构的编译器（如课程平台的MIPS交叉编译器）重复上述各步编译过程，观察并在实验报告中提交相应结果。

`objdump`参数定义：

```
OBJDUMP(1)                                                        GNU Development Tools

NAME
       objdump - display information from object files

SYNOPSIS
       objdump [-a|--archive-headers]
               [-b bfdname|--target=bfdname]
               [-C|--demangle[=style] ]
               [-d|--disassemble[=symbol]]
               [-D|--disassemble-all]
               [-z|--disassemble-zeroes]
               [-EB|-EL|--endian={big | little }]
               [-f|--file-headers]
               [-F|--file-offsets]
               [--file-start-context]
               [-g|--debugging]
               [-e|--debugging-tags]
               [-h|--section-headers|--headers]
               [-i|--info]
               [-j section|--section=section]
               [-l|--line-numbers]
               [-S|--source]
               [--source-comment[=text]]
               [-m machine|--architecture=machine]
               [-M options|--disassembler-options=options]
               [-p|--private-headers]
               [-P options|--private=options]
               [-r|--reloc]
               [-R|--dynamic-reloc]
               [-s|--full-contents]
               [-W[lLiaprmfFsoRtUuTgAckK]|
                --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,=frames-interp,=str,=loc,=Ranges,=pubtypes,=t
s,=gdb_index,=addr,=cu_index,=links,=follow-links]]
               [--ctf=section]
               [-G|--stabs]
               [-t|--syms]
               [-T|--dynamic-syms]
               [-x|--all-headers]
```

相应结果：

21210113

```
./gxemul/vmlinux:     file format elf32-tradbigmips

Disassembly of section .text:

80010000 <_start>:
80010000:   40806000    mtc0    zero,$12
80010004:   00000000    nop
80010008:   40809000    mtc0    zero,$18
8001000c:   00000000    nop
80010010:   40809800    mtc0    zero,$19
80010014:   00000000    nop
80010018:   40088000    mfc0    t0,$16
8001001c:   2401fff8    li  at,-8
80010020:   01014024    and t0,t0,at
80010024:   35080002    ori t0,t0,0x2
80010028:   40888000    mtc0    t0,$16
8001002c:   0c004014    jal 80010050 <main>
80010030:   3c1d8040    lui sp,0x8040
80010034:   00000000    nop

80010038 <loop>:
80010038:   0800400e    j   80010038 <loop>
8001003c:   00000000    nop
    ...

80010050 <main>:
80010050:   27bdffe8    addiu   sp,sp,-24
80010054:   afbf0010    sw  ra,16(sp)
80010058:   3c048001    lui a0,0x8001
8001005c:   0c0042a2    jal 80010a88 <printf>
80010060:   24840b48    addiu   a0,a0,2888
80010064:   0c004024    jal 80010090 <mips_init>
80010068:   00000000    nop
8001006c:   3c048001    lui a0,0x8001
80010070:   24840b64    addiu   a0,a0,2916
80010074:   24050014    li  a1,20
80010078:   3c068001    lui a2,0x8001
8001007c:   0c0042b2    jal 80010ac8 <_panic>
80010080:   24c60b6c    addiu   a2,a2,2924
    ...
```

Thinking1.2

也许你会发现我们的readelf程序是不能解析之前生成的内核文件(内核文件是可执行文件)的，而我们之后将要介绍的工具readelf则可以解析，这是为什么呢？(提示：尝试使用readelf -h，观察不同)

使用readelf文件夹下`./readelf ../gxemul/vmlinux`进行解析：

```
git@21210113:~/21210113/readelf$ ./readelf ../gxemul/vmlinux
Segmentation fault (core dumped)
```

使用`readelf -h`解析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:          37212 (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
```

我们错误是有无数据编码格式导致的问题，内核使用的是大端存储，而我们只能解析小段存储的文件，会在读取信息时发生错误。

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？

本次实验使用`start.S`作为入口，在其内部使用跳转指令`jal`跳转到指定的`init/main.c`中的`main`函数中，因此只需要传递函数地址就可以实现不同的main函数的调用。我们的操作系统运行在软件上，因此只需要找到相应的main函数入口加载到相应的地址就可以被正确的跳转到。

Thinking1.4

sg_size 和bin_size 的区别它的开始加载位置并非页对齐，同时bin_size的结束位置（va+i）也并非页对齐，最终整个段加载完毕的sg_size 末尾的位置也并非页对齐，请思考，为了保证页面不冲突（不重复为同一地址申请多个页，以及页上数据尽可能减少冲突），这样一个程序段应该怎样加载内存空间中。

不同的程序段在页目录表中使用不同的项，因此不冲突。段内可以进行页对齐，就可以按页取出使用。

Thinking1.5

内核入口在什么地方？main 函数在什么地方？我们是怎么让内核进入到想要的 main 函数的呢？又是怎么进行跨文件调用函数的呢？

内核的入口在`_start`函数在`boot/start.S`，`main`函数在`init/main.c`，内核启动后先执行入口函数，从入口函数中通过`jal`指令跳转到`main`函数中，因此只需要指定函数`main`的确切地址就可以实现不同位置的`main`函数的调用。

跨文件调用函数与之前所属类似，`jal`到指定函数的地址处即可。

Thinking1.6

查阅《See MIPS Run Linux》一书相关章节，解释 boot/start.S 中下面几行对 CP0 协处理器寄存器进行读写的意义。具体而言，它们分别读/写了哪些寄存器的哪些特定位，从而达到什么目的？

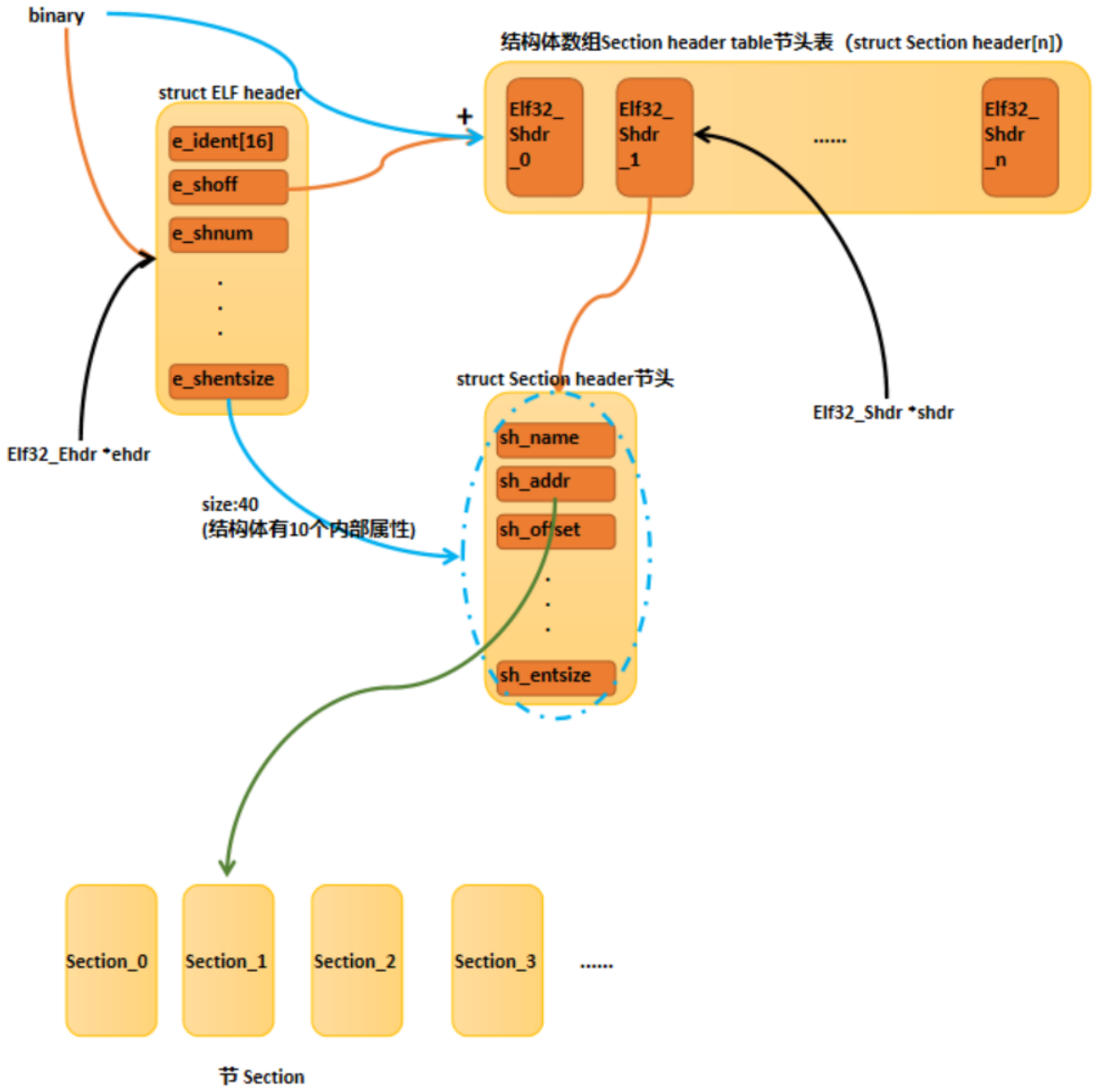
```
/* Disable interrupts */
mtc0 zero, CP0_STATUS # 把0送入协处理器0的status寄存器

......

/* disable kernel mode cache */
mfc0 t0, CP0_CONFIG  # 取出协处理器0 config寄存器中的数
and t0, ~0x7    # 将后三位清零
ori t0, 0x2     # 将倒数第二位置1
mtc0 t0, CP0_CONFIG # 得到的$t0寄存器数据重新写回协处理器0 config寄存器
```

实验难点

本次实验难点首先在于对于ELF文件的解析，首先ELF文件在编译链接与运行两个过程中会有两种不同的结构，我们通过阅读`readelf.c`得知ELF文件各个部分的结构，以及其中结构体包含的数据及其相应的含义，通过适当的处理取出结构体并获取所需数据完成相应的任务。



第二个难点就是页面冲突的发现与解决，这段我的理解实在是不太好，一会儿应该要放到残留难点里面去，有时间方便的话希望能讲讲比如课上Extra那道题怎么解决和思考题的答案。

指导书反馈

还是那句话，希望指导书能够不打哑谜，多举一些实例，毕竟是阅读代码是学习的过程之一，而不是面对文字坐那猜闷。

残留难点

页面冲突的发现与解决，这段我的理解实在是不太好，一会儿应该要放到残留难点里面去，有时间方便的话希望能讲讲比如课上Extra那道题怎么解决和思考题的答案。