Lab1实验报告

Thinking

Thinking1

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：

- D：即--disassemble-all ， 反汇编所有section
- S: 即 --source ， 尽可能反汇编出源代码

hello.c文件内容

```
int main(){
    int a = 1;
    return 0;
}
```

编译过程：

- 预处理

```
/OSLAB/compiler/usr/bin/mips_4KC-gcc -E hello.c

输出结果：
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
int main(){
 int a = 1;
 return 0;
}
```

- 编译

```
/OSLAB/compiler/usr/bin/mips_4KC-gcc -c hello.c
/OSLAB/compiler/usr/bin/mips-linux-objdump -DS hello.o > out.txt
```

反汇编结果

```
hello.o:     file format elf32-tradbigmips

Disassembly of section .text:

00000000 <main>:
   0:   27bdffe8    addiu   sp,sp,-24
   4:   afbe0010    sw  s8,16(sp)
   8:   03a0f021    move    s8,sp
   c:   24020001    li  v0,1
```

```
  10:   afc20008    sw  v0,8(s8)
  14:   00001021    move    v0,zero
  18:   03c0e821    move    sp,s8
  1c:   8fbe0010    lw  s8,16(sp)
  20:   27bd0018    addiu   sp,sp,24
  24:   03e00008    jr  ra
  28:   00000000    nop
  2c:   00000000    nop
```

- 链接

```
/OSLAB/compiler/usr/bin/mips_4KC-ld -o hello hello.o
/OSLAB/compiler/usr/bin/mips-linux-objdump -DS hello > out.txt
```

反汇编结果

```
004000b0 <main>:
  4000b0:   27bdffe8    addiu   sp,sp,-24
  4000b4:   afbe0010    sw  s8,16(sp)
  4000b8:   03a0f021    move    s8,sp
  4000bc:   24020001    li  v0,1
  4000c0:   afc20008    sw  v0,8(s8)
  4000c4:   00001021    move    v0,zero
  4000c8:   03c0e821    move    sp,s8
  4000cc:   8fbe0010    lw  s8,16(sp)
  4000d0:   27bd0018    addiu   sp,sp,24
  4000d4:   03e00008    jr  ra
  4000d8:   00000000    nop
  4000dc:   00000000    nop
```

Thinking2

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*)*

用自己写的readelf程序解析：

```
git@20373159:~/20373159$ ./readelf/readelf gxemul/vmlinux
Segmentation fault (core dumped)
```

官方版的readelf解析：

```
git@20373159:~/20373159$ 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:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          36652 (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
```

发现原因在于自己的程序不支持解析segment，但是官方版支持

Thinking3

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？

MIPS加电后，操作系统内核启动前，BootLoader会做准备性工作，其目标就是能正确地调用内核。其分为两个阶段，stage1会初始基本的硬件，为stage2准备RAM空间，方便内核镜像文件加载到内存。stage2就把内核从存储器读到RAM中了，并将CPU取值寄存器PC设置为内核入口函数的地址，便保证了内核入口被正确跳转到。

Thinking4

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。

1. 若出现程序段开始的地址页不对齐，则申请一页内存，并分配。
2. 之后一页一页的申请内存，并将ELF文件内容拷贝到内存，用一个变量`i`记录申请内存的大小，当`i`大于`bin_size`停止
3. 继续一页一页申请内存，但是此时不需要复制，当`i`大于`sg_size`时停止。
4. 如果此时末地址占用的页面地址为`vi`，那么之后的程序段的首地址应该从下一个页面`vi+1`开始申请。

Thinking5

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？

内核入口在boot/start.S的_start中，main函数在init文件夹下的main.c中，通过start.S的跳转jal指令跳转到main中，虽然main由C语言编写，但是被编译成汇编之后，其入口点会被翻译为一个标签，经过链接后可以实现跨文件调用函数。

Thinking6

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
//将状态寄存器SR置0，主要是为了将IEc(SR[0])置0，使CPU不会响应任何中断。
......
/* disable kernel mode cache */
mfc0 t0, CP0_CONFIG
//取Config寄存器，存入t0中
and t0, ~0x7
//将第0、第2位置0，第1位置为1。K0代表最低3位，决定kseg0区是否经过cache。该操作禁止经过cache
ori t0, 0x2
mtc0 t0, CP0_CONFIG
```

在include/asm/cp0regdef.h下，有有关对寄存器的宏定义

```
CP0_STATUS $12 SR
CP0_CONFIG $16 Config
```

实验难点

本实验重点和难点就在于对ELF文件的解读和printf函数的实现

ELF文件

ELF 文件是一种对可执行文件、目标文件和库使用的文件格式。

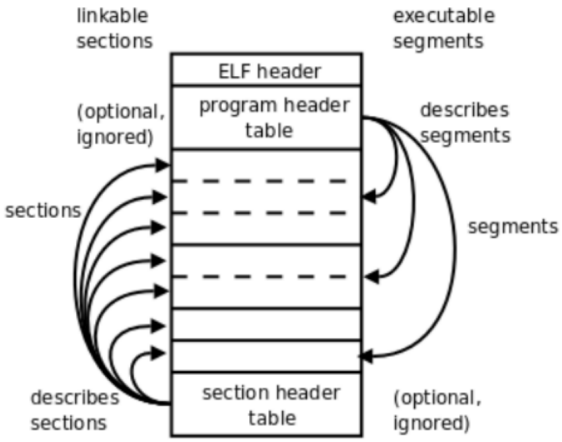


图 1.4: ELF 文件结构

ELF文件包含5个部分，分别为：

1. ELF Header，包括程序的基本信息，如下表

英文	中文	命名
Program Header Table file offset	程序头表相对ELF文件偏移量	e_phoff
Program header table entry size	程序头表每项(指针)的大小	e_phentsize
Program header table entry count	程序头表项数	e_phnum
Section header table file offset	节头表相对ELF文件偏移量	e_shoff
Section header table entry size	节头表每项的大小	e_shentsize
Section header table entry count	节头表项数	e_shnum

2. Section header table，在Elf文件中并不真实存在一个这样的结构体。但它的地址可以寻找section header。它由section header(节头表项)组成，节头表项的结构为Elf32_Shdr(section header)，给出section header table address，逐个加上entry size （从0开始加）就是每个section header的地址。
3. Program header table，类比section header table
4. Sections for **Linkable**，Section节记录了程序的**代码段、数据段等各个段的内容**，**主要是编译和链接要用**，其中最为重要的3个Section节为：
   - .text 保存可执行文件的操作指令。
   - .data 保存已初始化的全局变量和静态变量。
   - .bss 保存未初始化的全局变量和静态变量。

   记录信息如下表：

英文	中文	变量名
section addr	如果本节的内容需要映射到进程空间中去，地址为映射的起始地址；如果不需要映射，此值为 0。	sh_addr

5. Segments for **executable**，记录了每一段数据需要**被载入到内存的哪个位置**.**

英文	中文	变量
offset from elf file head of this entry	该段相对ELF表头的偏移量	p_offset
virtual addr of this segment	该段最终需要被加载到内存的哪个位置	p_vaddr
file size of this segment	该段数据在文件中的大小，也就是现在的大小	p_filesz
memory size of this segment	该段数据未来在内存中的大小	p_memsz

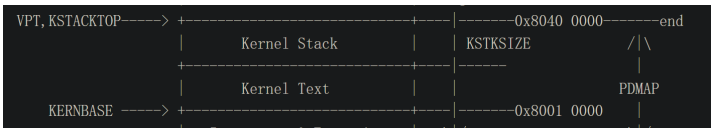
   注意，segment包含一个或多个section。程序头只对可执行文件或共享目标 文件有意义，对于其它类型的目标文件，该信息可以忽略。

操作系统的启动

Lab1主要讲解了操作系统的启动，内核如何启动，并在内核中实现了printf函数。内核，虽然是一个可执行文件vmlinux，但是是由xxx/文件夹下诸多模块、.o文件链接成的，所以本实验中修改各文件其实就是在修改内核。

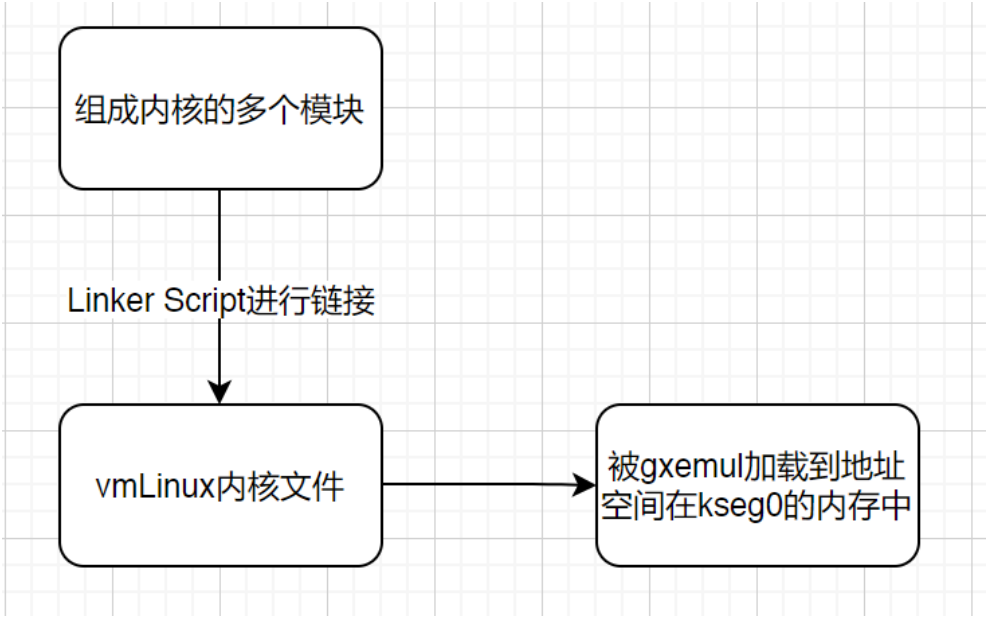
首先，Makefile是内核的地图，里面提到的modules为模块，模块也是目标对象文件，但是是链接到内核中，为了扩充内核功能的。使用Makefile的总目标就是**vmlinux可执行文件**，也就是我们的内核。

内核vmlinux准备好之后，gxemul需要根据内核的各segment中的virtual address 将其加载到内存的相应位置上。由于载入内核时未建立虚存机制，不能使用MMU，而且kseg1一般用于访问外部设备，故放在kseg0的这个位置。



那如何放在这个位置呢？当内核文件的.o文件写好后，需要链接，在链接过程中section的地址就可以改变！*(由于segment是由section组成，section地址改变后，segment也跟着变了)* 而在链接过程中，.o文件被看成section的集合，最关心的3个section就是.data,.text,.bss了，因此，只要在链接过程中控制这三个section的位置，就可以将内核加载到正确的位置。那如何控制呢？通过自定义链接脚本Linker Script即可控制3个section链接后在内存中的排布信息。

用一个流程图表示就是这样：



丰富内核：printf

这个printf函数是在要内核中实现。让控制台输出一个字符,实际上是对某一个内存地址写一串数据，这个操作由drivers/gxconsole/gxconsole.c 中的printcharc函数实现，由lib/printf.c的myoutput函数调用，具体的细节由print.c中的**lp_Print()**函数实现。

易错点主要是：

1. 牢记fmt指针处理完要执行++的操作
2. flag不一定只有一个，可能出现`printf("%0-0-0-0-0-0-0-0-023d%d", 12, 10);`这种flags的情况。以下测试了一些数据

感想

本章内容极其琐碎，本人通过所有exercise还是有点迷糊，重新捋了一遍才有所清晰，同时又由于对内存、地址的概念理解不深入，有些地方还是一知半解，甚至完全不解。总时长可能大概2天时间，但是还需要更长时间巩固消化，拓展些内存管理的知识 。

疑问

何为进程空间? 何为section的地址?

这里要补充一个知识点，程序的装载，装载是指由装载程序将可执行文件装载到内存。

程序的装载

1. 装载前准备工作

   shell 调用fork()系统调用，创建出一个子进程

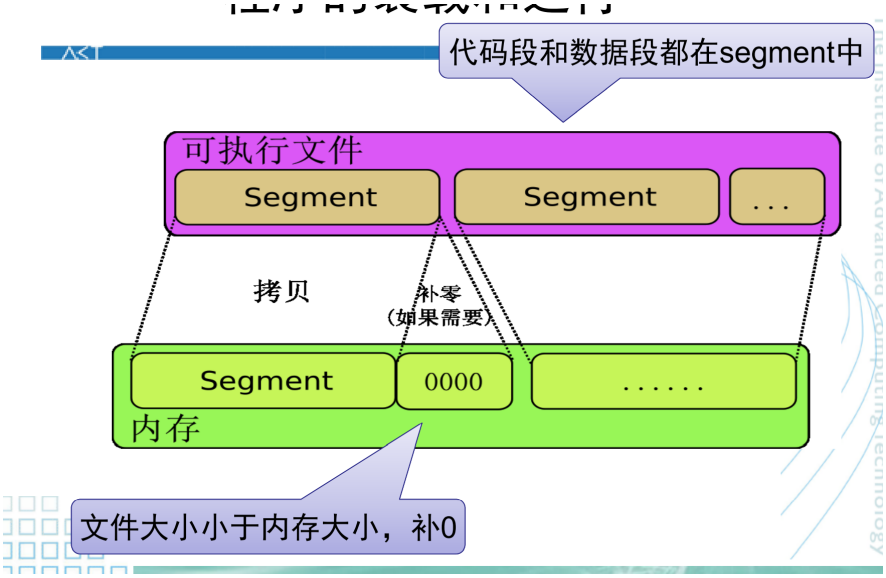
2. 装载工作

   子进程调用execve()加载program(即要执行的程序)。

3. 程序如何被装载

   加载器在加载程序的时候只需要看ELF文件中和 segment相关的信息即可。我们用readelf工具将segment读取出来，只需要装载p_type为Load的segment, segment会被从文件中拷贝到 内存中。具体装载流程如下：

   1. 找到ELF Header，根据其找到Segment。
   2. 对每一个要Load的Segment, 根据其memory size，为它分配足够的物理页，并映射到指定的虚地址上。再把文件内容拷贝到内存



   3. 若memory size > file size, 多出来的部分用0填充
   4. 设置进程控制块中的PC为ELF文件里记载的入口地址
   5. 进程开始执行

因此，Section header中记录的section的地址addr，就是在装入内存后的地址。header中的offset，是指该section在文件中的位置。文件不在内存里，文件在磁盘上。