Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

为什么会有这一步

readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);

为什么是第一页，是8个扇区而不是一个简单的头部

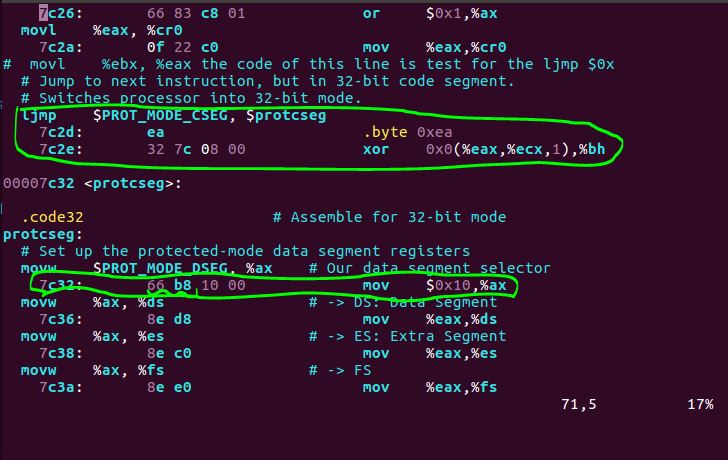
Be able to answer the following questions:

Q:what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

A:mov %eax,%cr0

ljmp $PROT\_MODE\_CSEG,$protcseg

经过编译之后变成了



而实际运行代码是：ljmp $0xb866,$0x87c32

按照一般情况下：ljmp应该翻译成：ljmp 0x8,0x7c00但是实际翻译成了如上

这是由于代码从16位转到了32位，下面的两个字节的指令被翻译到了上面。以下是论坛上的英文回答：

1. Some BIOS implementations go into protected mode before entering the bootloader. Most don't. It is possible that BIOS switches to protected mode for a short period and switches back before going to the bootloader, which would allow it to use some of the benefits of protected mode (such as 32 bit being the default address size). The reason that the bootloader should be in real mode is that most BIOS functions only work in real mode, so you need to be in real mode to use them.
2. ljmp specifies a code segment to switch to in addition to the address to jump to. They are so similar that (at least in GAS) the assembler will switch a jmp with 2 operands to a ljmp for you.
3. ljmp is one of the only ways to change the cs register. This needs to be done to activate protected mode, as the cs register needs to contain the selector for a code segment in the GDT. (In case you want to know, the other ways to change cs are far call, far return, and interrupt return)
4. See item 1. Either BIOS switched back to real mode, or this bootloader will not work with this BIOS.
5. See item 3. It changes cs to specify a 32 bit code segment, so the processor goes into 32 bit mode.
6. When you looked at the .asm file, the instruction was interpretted as if the address size was 32 bits, but GDB interpretted it as if the address size was 16 bits. The data at the address of the instruction would be 0xEA 32 7C 08 00 66 B8. EA is the long jump opcode. In a 32 bit address space, the address would be specified using the next four bytes, for an address of 0x87C32, but in a 16 bit address space, only 2 bytes are used, for an address of 0x7C32. The 2 bytes after the address specify the requested code segment, which would be 0xB866 in 32 bit mode and 0x0008 in 16 bit mode. The 0x66 B8 is the start of the next instruction, which is moving a 16 bit immediate value into the ax register, probably to set up the data segments for protected mode.

所以造成这个的原因就是代码由16位转为32而造成了。

Q:What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

A: ((void (\*)(void)) (ELFHDR->e\_entry))();和

if (ELFHDR->e\_magic != ELF\_MAGIC)

goto bad;

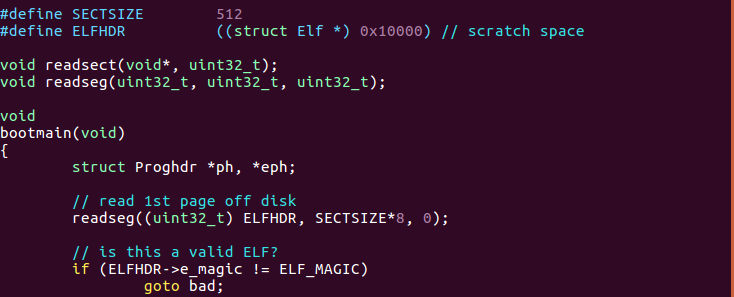
Q:Where is the first instruction of the kernel?

movw $0x1234,0x472 #warm boot

Q:How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

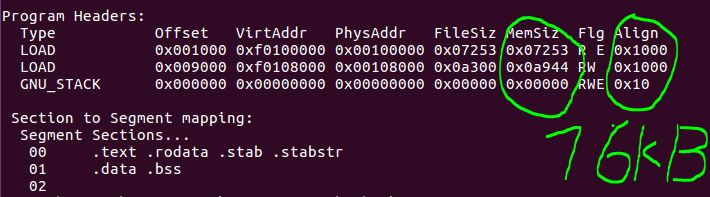
我们可以先知道一下的elf文件的格式。

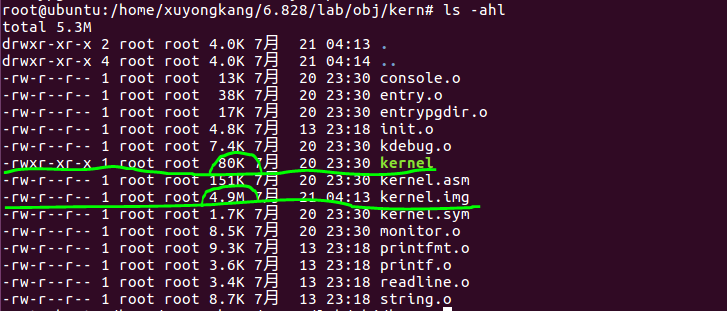
有多少的必须读入的扇区才能将完整的内核从磁盘上读入到内存中我们可以从源代码中看出：

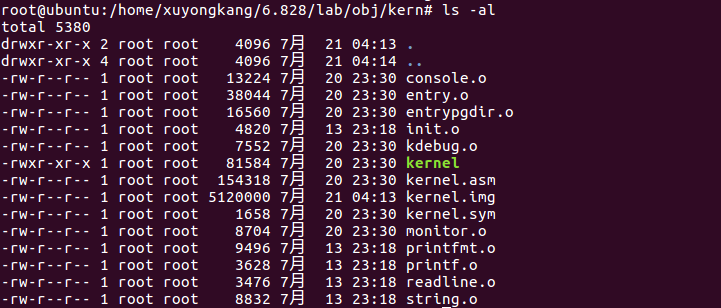


首先我们先从磁盘中的最开始的地方0位置开始读了512\*4 = 4KB的内容读到了内存中0x10000的位置，但是要注意的是：虽然readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);中偏移地址虽然为0，但是在函数readseg()中// translate from bytes to sectors, and kernel starts at sector 1

offset = (offset / SECTSIZE) + 1;这一句将自动跳过一个扇区（512字节进而加载kernel<elf格式>）从下一个扇区中加载4KB也就是一页的内容。这样也就把文件头和程序头加载到了内存中。至于为什么是4KB就能把所用到的头部信息加载到所有的内存中。我们可以计算。







而且我们知道kernel.img一共5120000字节所以根据obj/kern下makefile指令

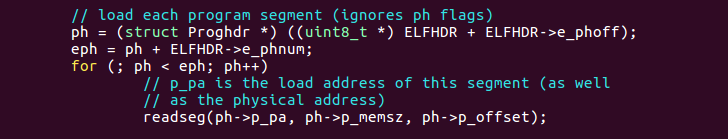
$(V)dd if=/dev/zero of=$(OBJDIR)/kern/kernel.img~ count=10000 2>/dev/null

我们可知命令默认的bs(blocksize)是：512字节，

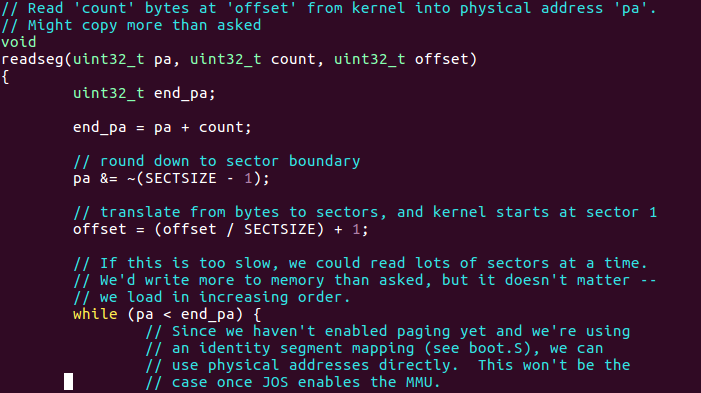
所以img共5MB大小

而整个从磁盘中国像内存中拷贝的扇区个数是（80KB/0.5KB）160个扇区

然后开始检查读入到内存中文件的magic\_number



所以根据文件头的指引将所有的程序段读入到内存中；这里我们需要注意从以下图中我们才能知道如何计算读入了多少扇区



end\_pa = pa + count;

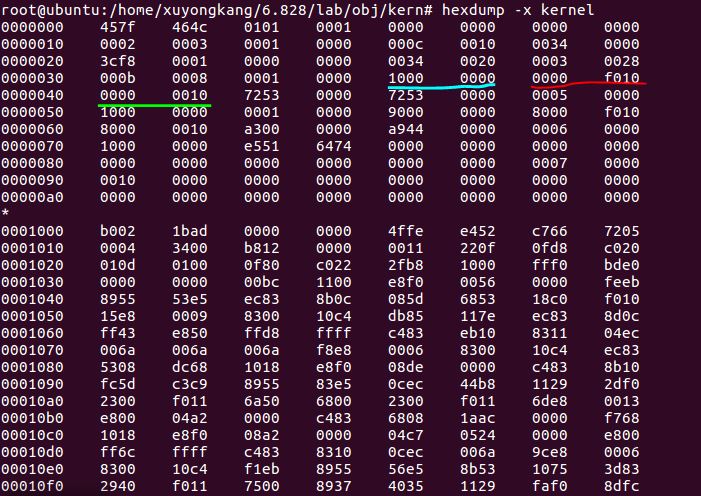
// round down to sector boundary

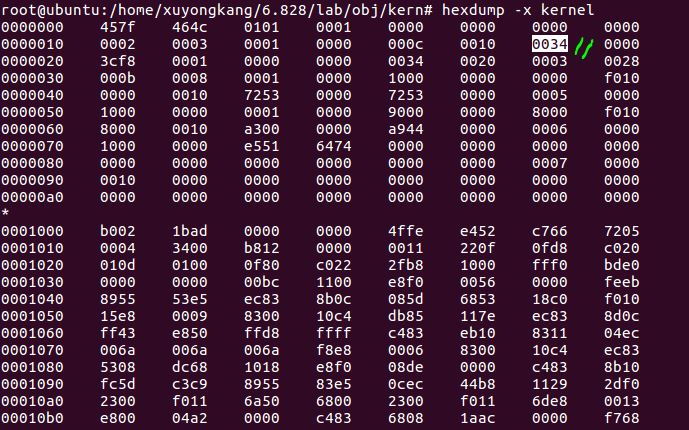
（1） pa &= ~(SECTSIZE - 1);

// translate from bytes to sectors, and kernel starts at sector 1

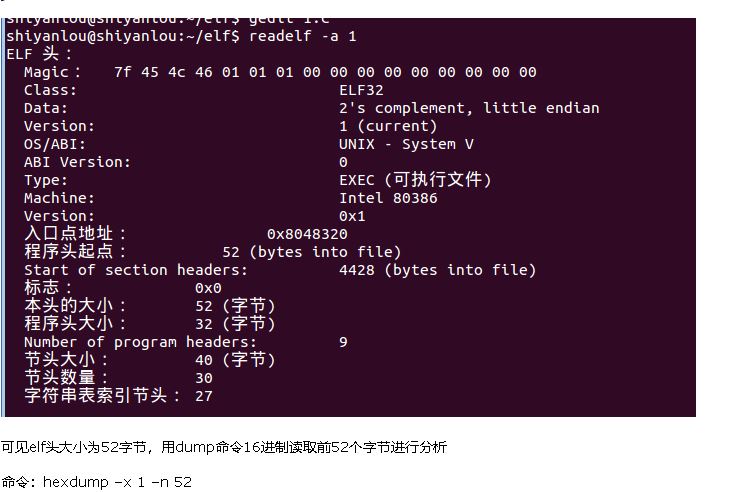
（2）offset = (offset / SECTSIZE) + 1;

我们知道ph是从已经读入0x10000（64KB）处的4KBELf头部变量得到的程序头表的入口地址。而ph->p\_pa就是程序各段的即将读入的内存的物理地址。如下图的三条线分别表示的意义是程序的偏移地址程序的虚拟地址程序的物理地址也就是p\_pa（我们可以有程序头的数据结构类型来进行判断），而偏移量刚好是0x1000也就是4KB这样就刚好是接着上回磁盘读取的位置接着读取（为什么偏移量是0x1000是有编译的时候自动生成的还是有相关的命令使之对齐不知道！！！！好像这个偏移地址是自动生成的因为读取程序段的时候程序是自动加上这个段的地址从而是磁盘磁头移动到相应的位置进行读写，而头部大小是自动计算生成的），而最终将程序读取到位置为0x100000（1M）字节的地方，而这此读过去的内容是按照程序段的方式读过去的没有ELF格式所以是直接能够运行。同理我们可以推断在生成kernel.img的时候boot.o被objcopy命令操作生成了/boot/boot，因此也是将所有与elf文件格式的相关信息去除而只是生成一个512字节的放到指定位置就能运行的代码而非elf文件因此如果是elf文件相应的操作系统就应当有相应的解析能力来使elf文件运行，但是在这512字节被载入到内存中之前我们知道显然操作系统此时没有这个能力。之后将obj/kernel写入到这512字节的后面，而这个时候kernel是一个elf文件这个文件包含这操作系统中相关运行代码的信息，这些信息也是bootmain依靠来把操作系统立起来的基础。而最终512字节的mbr被读入到0x7c00位置，之后的4KB的kernel头部表的相关信息《到底是不是4KB还有疑问但所有的程序段之前的所有内容就是4KB而剩下的所有内容都是程序段见kernel.ld<为什么这里有kernel.ld而boot的时候没有~~可能是因为程序太过于复杂？~~还是我们需要让相应的程序段映射到我们想要的位置```这个好像是正确的```>也有相关描述对链接操作的描述》被读入0x10000的内存位置，而最终我们依靠这4KB的内容将剩余的几个程序段全部按照elf格式的要求读入到了地址：0x100000的位置并最终跳转到这个位置至此将控制权交给了内核程序，最操作系统立了起来。



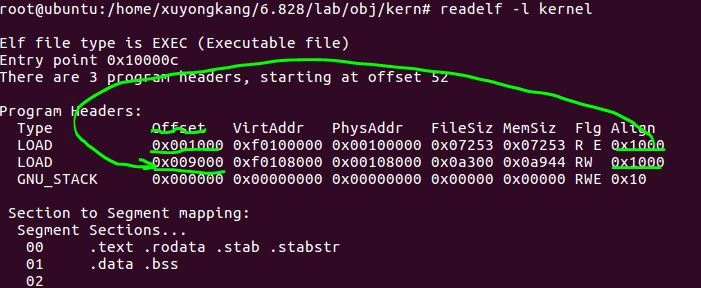


我们使用hexdump命令对kernel二进制文件进行查看，相关的结果我们可以查看



（下面的图片见链接https://www.cnblogs.com/EliteDci/p/5578901.html）





一共三个程序段：

偏移量分别为：0x1000,0x9000,0x0000

由于是0x1000对齐的所以整个内核的占的空间的大小为：0xb000+0x8000=0x13000=80KB,所以最终拷贝的扇区个数是：160个扇区共80KB.其中Program Header部分占用76KB.

要回答这个问题首先就是要知道ELF的文件结构

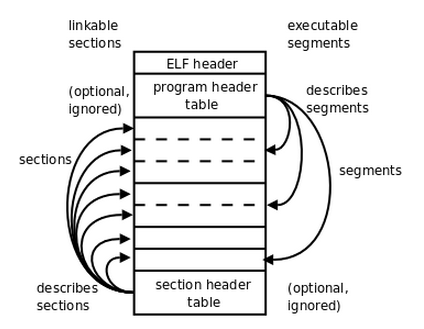
1.首先Elf文件一共分为三种类型：

1. 可重定位的对象文件(Relocatable file)   
   这是由汇编器汇编生成的 .o 文件。后面的链接器(link editor)拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件 (Executable file) 或者一个可被共享的对象文件(Shared object file)。我们可以使用 ar 工具将众多的 .o Relocatable object files 归档(archive)成 .a 静态库文件。如何产生 Relocatable file，你应该很熟悉了，请参见我们相关的基本概念文章和JulWiki。另外，可以预先告诉大家的是我们的内核可加载模块 .ko 文件也是 Relocatable object file。
2. 可执行的对象文件(Executable file)   
   这我们见的多了。文本编辑器vi、调式用的工具gdb、播放mp3歌曲的软件mplayer等等都是Executable object file。你应该已经知道，在我们的 Linux 系统里面，存在两种可执行的东西。除了这里说的 Executable object file，另外一种就是可执行的脚本(如shell脚本)。注意这些脚本不是 Executable object file，它们只是文本文件，但是执行这些脚本所用的解释器就是 Executable object file，比如 bash shell 程序。
3. 可被共享的对象文件(Shared object file)   
   这些就是所谓的动态库文件，也即 .so 文件。如果拿前面的静态库来生成可执行程序，那每个生成的可执行程序中都会有一份库代码的拷贝。如果在磁盘中存储这些可执行程序，那就会占用额外的磁盘空 间；另外如果拿它们放到Linux系统上一起运行，也会浪费掉宝贵的物理内存。如果将静态库换成动态库，那么这些问题都不会出现。动态库在发挥作用的过程 中，必须经过两个步骤：

a) 链接编辑器(link editor)拿它和其他Relocatable object file以及其他shared object file作为输入，经链接处理后，生存另外的 shared object file 或者 executable file。   
b)在运行时，动态链接器(dynamic linker)拿它和一个Executable file以及另外一些 Shared object file 来一起处理，在Linux系统里面创建一个进程映像。

**2. ELF文件格式**

首先，ELF文件格式提供了两种视图，分别是链接视图和执行视图。



链接视图是以节（section）为单位，执行视图是以段（segment）为单位。链接视图就是在链接时用到的视图，而执行视图则是在执行时用到的视图。上图左侧的视角是从链接来看的，右侧的视角是执行来看的。总个文件可以分为四个部分：

- ELF header： 描述整个文件的组织。

- Program Header Table: 描述文件中的各种segments，用来告诉系统如何创建进程映像的。

- sections 或者 segments：segments是从运行的角度来描述elf文件，sections是从链接的角度来描述elf文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序（所以很多加固手段删除了section header table）。从图中我们也可以看出，segments与sections是包含的关系，一个segment包含若干个section。

- Section Header Table: 包含了文件各个segction的属性信息，我们都将结合例子来解释。



ELF头部：

struct Elf {

uint32\_t e\_magic; // must equal ELF\_MAGIC

uint8\_t e\_elf[12];

uint16\_t e\_type;

uint16\_t e\_machine;

uint32\_t e\_version;

uint32\_t e\_entry;

uint32\_t e\_phoff;

uint32\_t e\_shoff;

uint32\_t e\_flags;

uint16\_t e\_ehsize;

uint16\_t e\_phentsize;

uint16\_t e\_phnum;

uint16\_t e\_shentsize;

uint16\_t e\_shnum;

uint16\_t e\_shstrndx;

};

程序头部表：

struct Proghdr {

uint32\_t p\_type;

uint32\_t p\_offset;

uint32\_t p\_va;

uint32\_t p\_pa;

uint32\_t p\_filesz;

uint32\_t p\_memsz;

uint32\_t p\_flags;

uint32\_t p\_align;

};

节区头部表：

struct Secthdr {

uint32\_t sh\_name;

uint32\_t sh\_type;

uint32\_t sh\_flags;

uint32\_t sh\_addr;

uint32\_t sh\_offset;

uint32\_t sh\_size;

uint32\_t sh\_link;

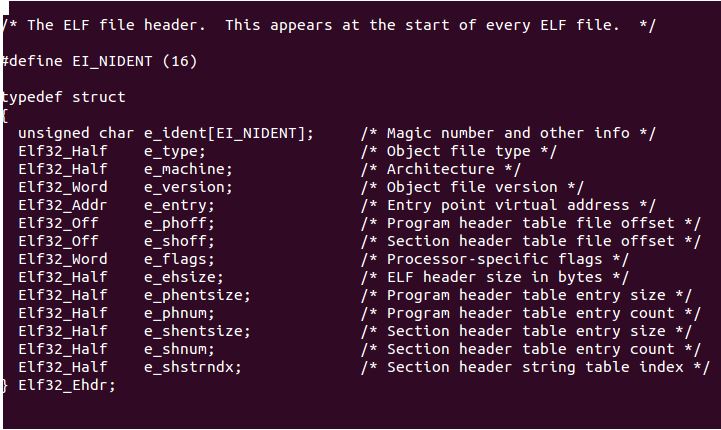
uint32\_t sh\_info;

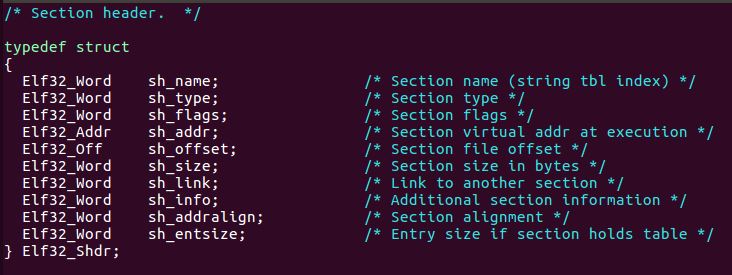
uint32\_t sh\_addralign;

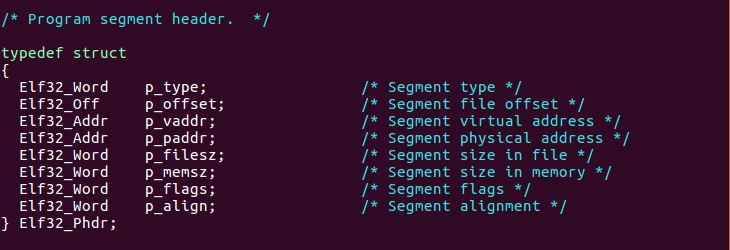
uint32\_t sh\_entsize;

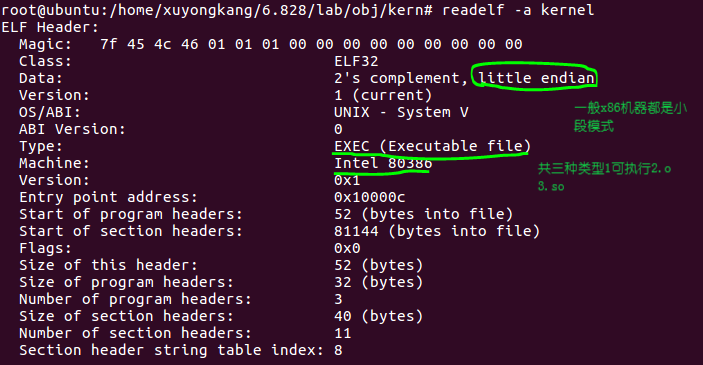
};

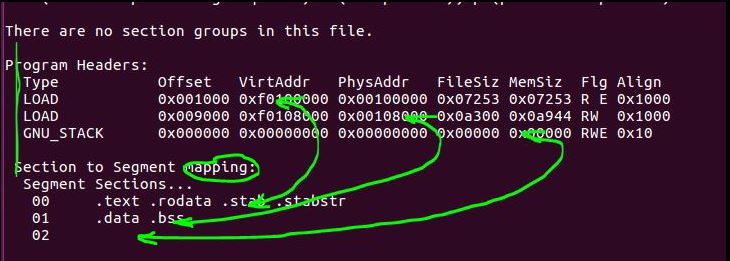
JOS ELF文件的魔数：#define ELF\_MAGIC 0x464C457FU /\* "\x7FELF" in little endian \*/



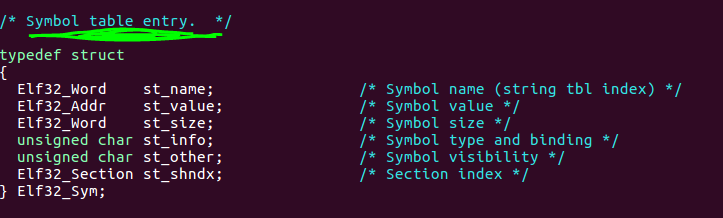








至此我们只是总结了作为二进制文件的elf文件的结构还有(.o)和（.so）格式这些文件中内容都是以段为单位的例如下面的：



具体更多信息见：<https://blog.csdn.net/feglass/article/details/51469511>

由于文中内容只是自己的学习过程的记录，并给都是原创。

#查看elf文件的相关命令：

Readelf –h kernel

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: 0x10000c

Start of program headers: 52 (bytes into file)

Start of section headers: 81144 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

Size of program headers: 32 (bytes)

Number of program headers: 3

Size of section headers: 40 (bytes)

Number of section headers: 11

Section header string table index: 8

Readelf –a kernel 查看全部的相关信息

Readelf –l kernel 查看programmer headers和section headers