Linker Script in Linux

—— vmlinux_32.lds.S 与 vmlinux

Author: ZX_WING (xing5820@163.com)

（本文基于 **2.6.27-rc2** 内核版本）

写在前面的话

经常在内核中看到一些由链接脚本提供的全局变量，但一直不清楚链接脚本的工作原理，这是件非常不爽的事情。最近刚做完一个 task，于是抽了点时间学习了一下连接脚本的语法，配合 Linux 下 x86 平台的 vmlinux_32.lds.S 文件及编译生成的未压缩的内核——vmlinux，学习了内核是如何通过链接脚本生成最后的 image 的。写下这篇文章，供感兴趣的朋友参考。小弟对链接器知之甚少，文中难免有错，希望发现错误的朋友发信到 xing5820@163.com，我好及时纠正：）

内容提要

本文简单的介绍了理解内核链接脚本需要的链接器知识和链接脚本语法，从 vmlinux_32.lds.S 文件分析了内核 image 的构成，并着重讲解了使用自定义 section 配合链接脚本动态创建表的方法。这里或许有一些你经常看到但不了解原理的东西，例如“内核导出符号表是如何创建的？”、“__initcall 修饰的函数在什么时候被内核调用？”、“__initdata 是否会被释放？”等。

Revision History

日期	版本	描述
2008.9.18	1.0	最初发表版本

1. 什么是链接脚本

链接器主要有两个作用，一是将若干输入文件（.o 文件）根据一定规则合并为一个输出文件（例如 ELF 格式的可执行文件）；一是将符号与地址绑定（当然加载器也要完成这一部分工作）。关于链接器的工作机制可以参考《Linker and Loader》一书，本文只关心它的第一个功能，即如何根据一定规则将一个或多个输入文件合并成输出文件。这里的“一定规则”是通过链接脚本描述的。链接器有一个编译到其二进制代码中的默认链接脚本，大多数情况下使用它链接输入文件并生成目标文件。当然，我们也可以提供自定义的脚本以精确控制目标文件的格式，如同 Linux 内核做得那样，链接器“- T”参数用于指定自定义的脚本文件。

链接脚本有自己的一套语法，本文无意对它进行过多论述，后文描述 vmlinux_32.lds.S 内容时会对内核用到的语法进行解释。如果你希望了解完整的脚本语法，可以阅读参考文献 1。

2. 一些准备知识

说起链接器，ELF 文件格式通常是绕不开的，介绍它的文档多不胜数。实际上，对于了解链接脚本，我们完全没必要去学习 ELF 的具体格式，有一个全局的视图就足够了（当然，了解 ELF 格式会让事情变得轻松，你可以很轻易的将脚本中的某些元素和 ELF 格式中的一些字段联系起来，例如后面看到的 PHDRS 关键字就很容易和 ELF 的程序头部表关联）。

<table>
<tr><th>链接视图</th><th>执行视图</th></tr>
<tr><td>ELF 头部</td><td>ELF 头部</td></tr>
<tr><td>程序头部表（可选）</td><td>程序头部表</td></tr>
<tr><td>节区 1</td><td rowspan="2">段 1</td></tr>
<tr><td>...</td></tr>
<tr><td>节区 n</td><td rowspan="2">段 2</td></tr>
<tr><td>...</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>节区头部表</td><td>节区头部表（可选）</td></tr>
</table>

图 1. 链接器视图 overview（摘自《ELF 文件格式分析》，滕启明）

图 1 展示了从链接器的角度，如何看待输入文件和输出文件的视图。左边的“链接视图”

对应输入文件，它为链接器提供的主要内容是 section（节区）。随便找一个.o 文件，通过 objdump 后可以找到类似下面的内容：

```
 10 .init         00000030  080482b4  080482b4  000002b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000050  080482e4  080482e4  000002e4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         0000019c  08048340  08048340  00000340  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

这里的.init、.plt、.text 就是 section 名。不同的.o 文件可以有相同的 section，例如.text。编译器在生成.o 文件时会根据所生成二进制的不同性质把它们放入相应的 section 中。例如函数编译后的二进制代码通常放到.text，而 const 关键字修饰的全局数组会放到.rodata 中。GCC 有除了默认的 section，例如.text、.data、.bss、.debug、.dynsym 等，也支持用户自定义 section，在后面的内容中我们可以看到 Linux 大量使用 GCC 的扩展__attribute__((section(“section_name”))生成自定义 section。

链接器在进行链接时，会根据链接脚本从输入的.o 文件中挑选出感兴趣的 section，把它们合并生成新的 section，这些新产生的 section 归属于目标文件的某个 segment（段），并出现在目标文件中。例如 file1.o 和 file2.o 分别有两个.text，它们在链接后生产的目标文件也会有一个.text，而这个.text 既是由 file1.o 和 file2.o 的.text 合并而来的。这里提到了 segment 的概念，见图 1 的右部“执行视图”。Segment 可以看作一组具有相同属性（或部分相同属性）的 section 的集合，属性是指“读、写、执行”（通常用 rwx 或 rwe 表示）。例如.text 通常存放的是代码编译后的二进制，它具有 r-x 权限；.rodata 存放是的只读数据，如常量字符串，它通常具有 r—权限（实际上也可以具有 x 权限，例如用一个全局 const 数组存放可执行的机器码）；那么在生成目标文件时，.text 和.rodata 就可以通过一个具有 r-x 属性的 text segment 来包含它们，这就是我们通常说的“文本段”。经常看到有朋友在 C 版问“常量字符串的地址为什么在文本段？”、“常量字符串放哪儿？”之类的问题，其实写一个简单的程序，例如：

```
int main()
{
    printf(“%s\n”, “hello world”);
}
```

用 gcc –S 编译后可以看到：

```
    .section    .rodata
.LC0:
    .string "hello, world"
```

这里常量字符串”hello, world”放到了.rodata section，链接后该 section 通常会和.text section 一起放到目标文件的 text segment 中，这就是为什么字符串地址和 main()函数的地址如此相近的原因。

Segment 在 ELF 术语中称为 program headers，用来描述整个目标文件以什么样的方式加

载到内存中，方式是指加载的地址、segment 的长度和属性等等。用 objdump –p 命令可以查看目标文件的 segment，当然你也可以在通过 objdump –Dx 得到内容中找到它们。其内容如下所示（类似）：

```
Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x00000100 memsz 0x00000100 flags r-x
  INTERP off    0x00000134 vaddr 0x08048134 paddr 0x08048134 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000055c memsz 0x0000055c flags r-x
    LOAD off    0x0000055c vaddr 0x0804955c paddr 0x0804955c align 2**12
         filesz 0x000000fc memsz 0x00000104 flags rw-
 DYNAMIC off    0x00000570 vaddr 0x08049570 paddr 0x08049570 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000148 vaddr 0x08048148 paddr 0x08048148 align 2**2
         filesz 0x00000044 memsz 0x00000044 flags r--
EH_FRAME off    0x000004e8 vaddr 0x080484e8 paddr 0x080484e8 align 2**2
         filesz 0x0000001c memsz 0x0000001c flags r--
```

-

到这里，我们可以简单的对链接器的工作做一个概括。链接器从输入的.o 文件中挑选出感兴趣的 section（注意，我们再次提到了“感兴趣的 section”。是的，并不是所有出现在.o 文件中的 section 都会出现在最后的目标文件中。在后面我们会看到 Linux 如何把它不感兴趣的 section 排除在外），根据链接脚本提供的规则生成新的 section，再根据新 section 的属性把它们分为不同的 segment。

目标文件加载到内存的过程实际上就是若干不同 segment 被加载到内存的过程，下一节我们会看到 Linux 内核 image 是如何划分 segment 的。

3. 内核链接脚本 --- vmlinux_32.lds.S

3.1 Overview

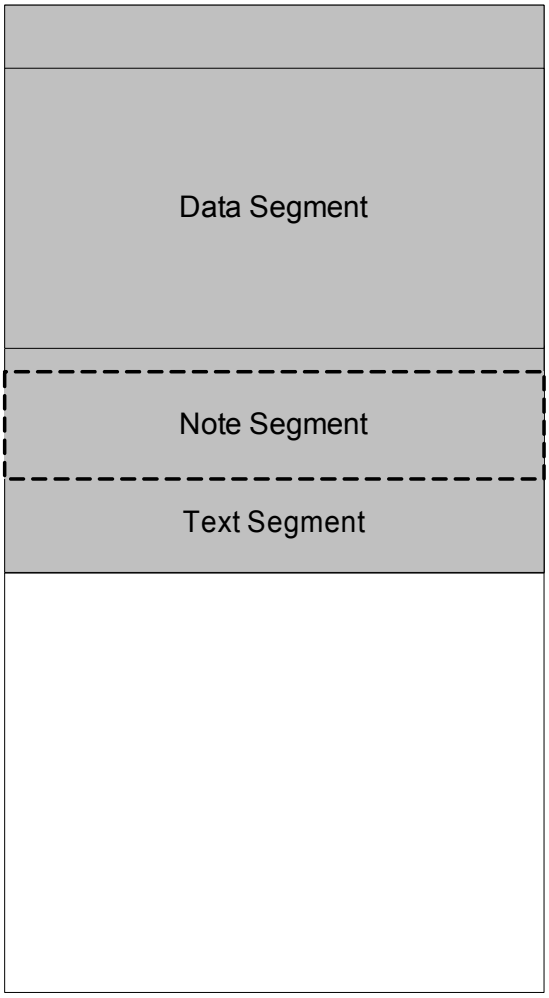

图 2. 内核 image 整体布局

图 2 显示了内核 image 的布局，其中深色部分位于内核的虚拟地址空间 3G~4G，共有 text、data、note 三个 segment，其中 note segment 又是包含在 text segment 中。每个 segment 包含多个 section，后面我们会讲到这些 section 是如何生成的。在这之前，需要了解链接脚本用到的两个地址：虚拟地址（VMA）和加载地址（LMA）。这里虚拟地址和我们平常说的虚拟地址是一样的，即 section[*]在目标文件加载后所在的虚拟地址。例如在一个可执行的 ELF 文件中，.text section 的 VMA 是 0x08048310，即.text section 的基地址位于虚拟地址空间的 0x08048310 处。加载地址指 section 被加载到内存中的地址，对于应用程序来说它通常和 VMA 相同，但对于内核来说，LMA 是指 section 被加载到的物理地址。例如内核.text 的 VMA 是 0xc1001000，则 LMA 是 0x01001000。很明显，这就是我们所熟知的内核虚拟地址

= 物理地址 + 0xC0000000（3G）的 identify mapping 关系。

*/*前面提到目标文件的加载是若干 segment 被加载到内存中的过程，这和 section 的加载并不冲突。实际上，当我们不指定 segment 的 LMA 和 VMA 时，这个两个值取 segment 中第一个 section 的 LMA、VMA。加载 segment 也就是将其包含的各个 section 加载到内存中的过程。

好了，下面我们来看看内核链接脚本是怎么干的。

3.2 链接脚本的开始

除去一些文件包含和宏定义，内核链接脚本以下面内容开始：

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(phys_startup_32)
jiffies = jiffies_64;

PHDRS {
      text PT_LOAD FLAGS(5);   /* R_E */
      data PT_LOAD FLAGS(7);   /* RWE */
      note PT_NOTE FLAGS(0);   /* ___ */
}
```

OUTPUT_FORMAT 和 OUTPUT 都是链接脚本的关键字，它们指定了目标文件的格式和所运行平台的架构，这些公式化的东西我们不关心它，具体内容详见参考文献 1。ENTRY 指定了整个目标文件的入口点（或入口函数），这里 phys_startup_32 是个地址，从名字我们就可以看出它是 startup_32()函数的物理地址，在后面会看到该地址是如何计算得到的。jiffies = jiffes_64 的魔术与本文无关，感兴趣的朋友可以参见 ULK3 的 6.2.1.2 节。

下面进入正题。PHDRS 关键字描述了 3 个 segment：text、data 和 note，它们分别具有 PT_LOAD 和 PT_NOTE 类型，并指定了每个 segment 的属性。PT_LOAD 类型表示该 segment 是从文件加载入内存的，在这个上下文中文件应该指最后生成的内核 image。FLAG 关键字指定 segment 的属性，如注释所示，text segment 为可读可执行、data segment 为可读可写可执行，note 段留到后面再说。至此，PHDRS 定义了内核 image 的大体框架，它包含两个最主要的 segment —— text 和 data，并确定了它们的属性，后面的代码就是向两个 segment 填充 section 了。

链接脚本知识：

PHDRS 关键字的完整格式如下：

```
PHDRS
{
```

name type [FILEHDR] [PHDRS] [AT (*address*)]
[FLAGS (*flags*)] ;
}

其中 name 表示 segment 的名字，它位于单独的名字空间，不会和后面的 section name 冲突。type 即 segment 的类型，如上的 PT_LOAD，详细列表参见参考文献 1。FILEHDR 和 PHDRS 指定是否要包含 ELF 文件头和 ELF 程序头。AT 指定 segment 的加载地址，FLAGS 指定 segment 的属性。

3.3 构造 Section

在构建了基本的 segment 后，就可以从输入.o 文件中获取感兴趣的 section 以生成新的 section 并放入相应的 segment。在这里，输入的 section 称为 input section，生成的新 section 称为 output section。除此之外，有一个重要的链接脚本符号“.”需要了解。”.”是个位置计数器，记录着当前位置在目标文件中的虚拟地址(VMA)。”.”是个自动增加的计数器，当一个 output section 生成后，”.”的值自动加上该 output section 的长度。我们也可以显式的给”.”赋值以改变当前位置的地址，这在内核链接脚本中被大量使用。一个例子可以很好的描述”.”的作用：

```
. = 0x100000;
_start_addr = .;
.text : { *(.text) }
_end_addr = . ;
```

这里我们首先给”.”赋了一个初值，将地址指定到 0x100000 处，并将该值赋给变量 _start_addr，它是.text section 的起始地址；接着我们生成了一个.text section，此时”.”自动加上该 section 的长度，可描述为 . = . + SIZEOF(.text)；最后将”.”赋值给_end_addr，记录下.text 的结束地址。此时”.”的值变成了 0x100000 + SIZEOF(.text)。有了”.”的帮助，我们可以灵活的控制目标文件中各个 section 所在的虚拟地址(VMA)。

3.3.1 Text Segment 的构造

内核首先构造的是 text segment，该 segment 又由若干个.text.*节构成，除此之外，它还包含了 note segment 的内容以及只读数据 section。下面的代码完成了这些工作：

```
SECTIONS
{
  . = LOAD_OFFSET + LOAD_PHYSICAL_ADDR;
  phys_startup_32 = startup_32 - LOAD_OFFSET;

  .text.head : AT(ADDR(.text.head) - LOAD_OFFSET) {
    _text = .;              /* Text and read-only data */
    *(.text.head)
  } :text = 0x9090

  /* read-only */
  .text : AT(ADDR(.text) - LOAD_OFFSET) {
    . = ALIGN(PAGE_SIZE); /* not really needed, already page aligned
*/
    *(.text.page_aligned)
    TEXT_TEXT
    SCHED_TEXT
    LOCK_TEXT
    KPROBES_TEXT
    *(.fixup)
    *(.gnu.warning)
    _etext = .;                 /* End of text section */
  } :text = 0x9090

  NOTES :text :note

  . = ALIGN(16);          /* Exception table */
  __ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {
    __start___ex_table = .;
     *(__ex_table)
    __stop___ex_table = .;
  } :text = 0x9090

  RODATA
```

首先是 SECTIONS 关键字，官方的解释是“The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory.”，实际上可以把它看成一个描述符，所有的工作要在它的内部完成。就像你在 C 中定义一个结构体要以 struct 关键字开头一样。

构造的第一步是为”.”指定初值，之后所有 section 虚拟地址（VMA）都由该值计算得来（前面我们讲过，生成一个 section 后”.”的值会自动加上改 section 的长度）。这里初始值为

LOAD_OFFSET + LOAD_PHYSICAL_ADDR，前者是我们熟知的内核虚拟地址空间起始地址 0xC0000000，LOAD_PHYSICAL_ADDR 是内核 image 加载的物理地址，由 CONFIG_PHYSICAL_START 计算得到。该物理地址是可以指定的，你可以在.config 文件中找到它，也可以由 make menuconfig 得到，具体解释参考 arch/x86/Kconfig 文件的 PHYSICAL_START 条目。对于一般的 x86 架构，内核被加载到物理地址 0x100000 处，故"."的初值为 0xC0100000。接着

```
phys_startup_32 = startup_32 - LOAD_OFFSET;
```

计算了内核 image 的入口地址，这在前面已经提到。

开始构造 section 了。由于使用的语法是固定的，我们只需要了解一个例子，其余的就可举一反三。以第一个 section 为例：

```
.text.head : AT(ADDR(.text.head) - LOAD_OFFSET) {
        _text = .;              /* Text and read-only data */
        *(.text.head)
    } :text = 0x9090
```

.text.head 指定了生成的 section 的名字，后面的冒号是固定语法。AT 关键字前面介绍过，指定该 section 的加载地址（LMA），它的完整表达是

```
AT(expression)
```

括号中 expression 表达式指定 LMA 的值。在此例中该表达式由

```
ADDR(.text.head) - LOAD_OFFSET
```

计算得到。这里

```
ADDR(section)
```

计算 section 的虚拟地址，故.text.head 的加载地址（LMA）是它的物理地址。在大括号内部，

```
_text = .;
```

定义了一个全局变量，它的值为"."的当前值，记录了整个 text segment 的起始地址.。在这里，由于_text 变量前还没有任何 section 被创建，故_text 有如下等价关系：

```
_text = ADDR(.text.head) = . = LOAD_OFFSET + LOAD_PHYSICAL_ADDR;
```

(.text.head)完成了具体的 section 创建工作，""代表所有输入的.o 文件，括号中的.text.head 指定了链接器感兴趣的 section 名。

```
*(text.head)
```

表示从所有输入文件中抽取名为.text.head 的 section 并填充到目标文件的.text.head section 中。

```
: text
```

指定了新生成 section 所在的 segment，这里冒号后的 text 是 segment 名，可见内核的第一个 section 被放到了 text segment。

```
= 0x9090
```

指定 section 的填充内容。从输入文件中抽取来的 section 由于代码对齐的缘故，其二进制的存放可能是不连续的，这里指定对 section 中的空隙用 0x9090 进行填充。0x90 是汇编指令 NOP 的机器码，故相当于在不连续代码间填充空操作。至此，内核的第一个 section 就创建好了，它名为.text.head，由输入文件的.text.head section 构成（并非所有文件都有.text.head section，链接器只从具有该 section 的文件中抽取内容），该 section 的虚拟地址（VMA）由"."的值确定，加载地址(LMA)为其物理地址，section 中不连续区域产生的间隙由 0x9090 填充，最后该 section 被放入了内核的 text segment 中。

通过 objdump 内核，我们可以看到关于该 section 的最终内容：

```
Sections:
Idx Name              Size      VMA       LMA       File off  Algn
  0 .text.head        00000375  c1000000  01000000  00001000  2**2
                      CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

  ……..
Disassembly of section .text.head:

c1000000 <_text>:
c1000000:>--f6 86 11 02 00 00 40 >--testb   $0x40,0x211(%esi)
c1000007:>--75 14                >--jne     c100001d <_text+0x1d>
c1000009:>--0f 01 15 1a e1 4d 01 >--lgdtl   0x14de11a
>--->--->---c100000c: R_386_32>-boot_gdt_descr
c1000010:>--b8 18 00 00 00       >--mov     $0x18,%eax
c1000015:>--8e d8                >--mov     %eax,%ds

……
c10013d5:>--5b                   >--pop     %ebx
c10013d6:>--5e                   >--pop     %esi
c10013d7:>--c9                   >--leave--
c10013d8:>--c3                   >--ret----
c10013d9:>--90                   >--nop----
c10013da:>--90                   >--nop----
c10013db:>--90                   >--nop----
c10013dc:>--90                   >--nop----
```

其中最后一部分显示了填充 0x9090 产生的 nop 指令。

链接脚本知识：

创建一个 section 的完整格式是：

section [*address*] [(*type*)] : [AT(*lma*)]

{

output-section-command
output-section-command
...
} [>*region*] [:*phdr* :*phdr* ...] [=*fillexp*]

其中[address]参数在上例中没有提到，它指定了 section 的虚拟地址(VMA)，如果没有指定该参数及 region 参数，section 的虚拟地址由当前”.”的值确定，正如上例我们看到的一样。Region 用于将 section 分配给通过 MEMORY 关键字创建的内存描述块，内核链接脚本没使用它，本文也不关注，具体内容详见参考文献 1 的 MEMORY command 一节。

通过这个例子，我们很容易就可以理解 text segment 中其它 section 的创建。例如接下来的第二个.text section，它的创建方法和.text.head 类似，唯一不同的是这里多了一句：

```
. = ALIGN(PAGE_SIZE);
```

ALIGN(exp)关键字计算当前”.”值对齐到 exp 边界后的地址，即：

```
ALIGN(exp) = ( . + exp – 1) & ~(exp – 1);
```

此处在创建.text section 前，将”.”对齐到了页边界，从第一个输入 section 的名字.text.page_aligned 就可以看出，输入 section 的内容是有对齐要求的。内核使用了 TEXT_TEXT 等宏将不同类型的输入 section 进行了封装，展开后可以看到它们都是：

```
*(section_name)
```

的形式，和我们前面讲的一样，不再多做介绍。

从上面内容可以看出，输入文件中的 section 有各种各样的名字，如.text.head、.text.page_aligned、.text.hot 等，并不是所有的 section 名都是标准的，绝大部分是内核使用 GCC 扩展生成的自定义名。举个例子，我们常见的__init 宏，展开后如下：

```
#define __init __attribute__ ((__section__(“.init.text”)))
```

这里.init.text 是个自定义的 section，用__init 修饰的函数编译后会被放到名为.init.text section 中。

自定义的 section 极大的发挥了链接脚本的作用，让我们可以对代码中的函数、数据进行归类操作，同时还可以完成一些在程序中不易完成的功能。这很容易理解，如果我们都用 GCC 内置的 section，何必要自定义链接脚本，用默认的不就好了。

链接脚本向我们展示了大量的自定义 section，本人水平有限，无法一一弄清每个 section 的用途，但通过几个常见的典型例子，我们可以了解它们的用法。首先就以 text segment 中的 exception table 举例。

3.3.1.1 Exception Table

此 exception table 不是用于处理硬件异常的（那是 IDT 表的工作），但它确实和硬件异

常有一点关系，具体来说是和 Page Fault 有关系。Exception Table 的具体机制在内核文档”Exception”中有详细介绍，你可以在/path_to_your_kernel_src/ Documentation/exception.txt 中找到它。这里为了说明问题做一点简要介绍。

我们尊敬的 Linus 大神为了避免内核在访问用户态地址时进行有效性检查带来的开销（我们总是需要这样的检查，虽然大部分情况下结果是成功的），利用了 page fault 的处理函数来完成这项任务，这样只有在真正访问了一个坏的用户态地址时检查才会发生。或许你会问：此时检查有什么用？一个例子就很容易说明问题，假设我们有一个函数叫 is_user_addr_ok()，用于检查传入的用户态地址是否合法。那么，当地址非法时它能干什么？什么都不能干，仅仅是告诉内核：“这是个非法地址，你不要访问”。这样便带来了个问题，让它在 90%的时间里告诉内核：“这是个合法地址，去吧！”是件很无聊的事情。既然该函数对非法地址无能为力，我们干脆就什么都不要干，直到内核真访问到一个非法地址时再告诉调用者：“噢，抱歉，您访问到一个非法地址。”不管用哪种方法，调用者遇到非法地址最终结果都是获得一个错误码，但后者明显省下了对合法地址进行检查的开销。让我们来看看如何用自定义 section 完成这个任务。

如果你顺着 copy_from_user()向下找几层，会看到__get_user_asm 宏，该宏展开后可读性太差，我们用下面的伪代码来描述它：

```
1:          movb (%from),(%to)    /* 这里访问用户态地址，当地址非法时会产生一个
page fault*/
2:
/*注意，后面的代码在最终的目标文件中不是跟在标号 2 后的*/
.section .fixup,"ax"
3:          movl $ERROR_CODE,%eax
            xorb %dl,%dl
            jmp 2b
.section __ex_table,"a"
            .align 4
            .long 1b,3b
```

上面的伪代码描述了__get_user_asm 宏的用途，它将用户态地址 from 中的内容拷贝到内核地址 to。当 from 是个非法地址时，会产生 page fault 从而执行内核的 do_page_fault()，在进行一系列检查处理后 fixup_exception()被调用，该函数会调用 search_exception_tables()查找 exception table，将 EIP 设置成对应 handler 的地址并返回。至此该非法地址造成的错误就交由 exception table 中的 handler 处理了。

所有问题的归结到了 exception table 的建立和错误处理 handler 的设置。其实上面的伪代码已经告诉我们答案了。首先，标号”1”代表了可能产生 page fault 的 EIP，当 page fault 产生时这个地址会被记录在 struct pt_regs 的 ip 字段中（不知道的看看 do_page_fault()的参

数）；其次，标号”3”是错误处理 handler 的地址，很明显，它只是返回了一个错误码(EAX 是 x86 的返回值寄存器)。jmp 2b 跳到了产生 page fault 的指令的下一条指令继续执行。这里

```
.section .fixup,"ax"
```

创建了名为.fixup 的自定义 section，并将整个 handler 放入其中。标号”1”后的代码是位于.text section 的，故你看到它们在源代码里写在了一起，但在目标文件中去是分开的，它们在不同的 section。

好了，我们已经有了会产生错误的代码地址，也有了错误处理 handler 的地址，

```
.section __ex_table,"a"
```

将它们放到了自定义的__ex_table section 中(.long 1b,3b)，以如下格式存放：

```
出错地址，处理函数地址
```

内核用结构体 struct exception_table_entry 表示该格式，定义如下：

```
struct exception_table_entry {
    unsigned long insn, fixup;
};
```

很明显，exception table 的格式简单，表项的前 4 个字节是出错地址，后 4 个字节是处理函数的地址。下图展示了通过 exception table 解决一次访问用户态非法地址产生的错误。

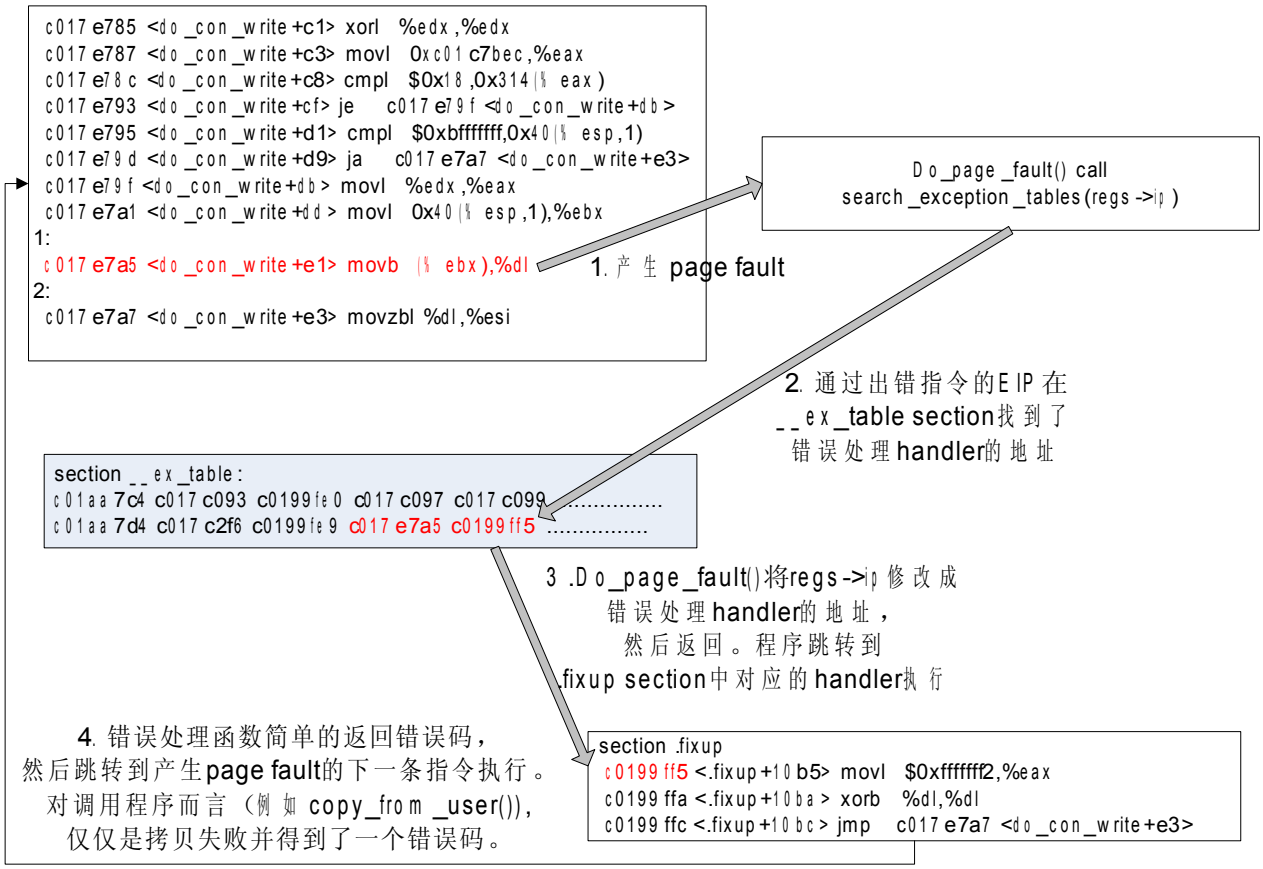

图 3. 通过 execption table 处理非法用户态地址访问的过程

怎样，所有的事实都清楚了。当在内核在不同位置调用 copy_from_user()时，展开的

__get_user_asm 宏都会将可能出错的地址和处理函数的地址存入该源文件对应.o 文件的__ex_table section 中。链接脚本的如下代码：

```
__ex_table  :  AT(ADDR(__ex_table)  -
LOAD_OFFSET) {
      __start___ex_table = .;
       *(__ex_table)
      __stop___ex_table = .;
   } :text = 0x9090
```

将分散的__ex_table 收集起来产生一张完整的 exception 表，并将表的起始地址和结束地址记录在__start___ex_table 和__stop___ex_table 两个全局变量中，从而 search_exception_tables() 函数可以顺利的索引该表。

这种通过自定义 section 和链接脚本构造表的技巧被大量使用，后面我们还会看到两个例子。在此先告一段落。

3.3.2 Note Segment

前面提到内核 image 分为三个 segment，其中就有 note segment，它是包含在 text segment 中的。NOTE segment 被用于不同的 vendor 在 ELF 文件中添加一些标识，让运行这些二进制代码的系统确定能否为该 ELF 提供其所需要的系统调用接口。它对我们了解内核用处不大，详细内容参见参考文献 2。

```
NOTES :text :note
```

上面代码中，NOTE 是一个宏，展开的格式和构建其它 section 的格式一样，这里”:text:note”表示把生成的 section 即加入 text segment 又加入 note segment。从 objdump 的内容可以看到后者包含在前者之中，如下：

```
LOAD off    0x00001000 vaddr 0xc1000000 paddr 0x01000000 align 2**12
      filesz 0x004de000 memsz 0x004de000 flags r-x
NOTE off    0x0037b844 vaddr 0xc137a844 paddr 0x0137a844 align 2**2
      filesz 0x00000024 memsz 0x00000024 flags ---
```

3.3.3 .rodata section 的构造

前面提到，只读数据被放到了 text segment，链接脚本中的 RODATA 宏完成了这项工作 。RODATA 创建了大量不同名称的 section，它们有些是内置的，有些则是自定义的。创建方式并无特别之处，有了前面的知识，你可以轻易的看懂它们。这里要说的是关于自定义 section 的第二个例子——内核符号表。

读过 LDD 的朋友都知道，在 module 中导出符号给内核其它部分应该使用__ksymtab，

我们也经常在内核中看到类似的代码，如：

```
EXPORT_SYMBOL(boot_cpu_data);
```

但，内核是怎么做的？符号表如何被创建？如果你看了/path_to_your_kernel_src/include/linux/module.h 中 EXPORT_SYMBOL 的定义，再配合自定义 section 的知识，很快就能明白内核只是创建了一个名为__ksymtab 的自定义 section，当调用 EXPROT_SYMBOL 宏时会生成一个 struct kernel_symbol 变量记录下函数/数据的名称和地址，最后将这个变量存入__ksymtab section 中。RODATA 宏的如下代码：

```
/* Kernel symbol table: Normal symbols */                \
__ksymtab               : AT(ADDR(__ksymtab) - LOAD_OFFSET) {        \
        VMLINUX_SYMBOL(__start___ksymtab) = .;                \
        *(__ksymtab)                                          \
        VMLINUX_SYMBOL(__stop___ksymtab) = .;                 \
}                                                             \
```

将输入文件中的__kysmtab section 合并生成新的__ksymtab section，这就是内核最终的导出符号表，同样，__start___ksymtab 和__stop___ksymtab 记录下了表的起始地址和结束地址。如此一来，动态加载 module 时内核如何将 module 中调用的函数替换成相应的地址就不难理解了吧。

链接脚本知识：

或许你已经注意到，上述创建__ksymtab section 的代码中，并没有在最后加上:text 标明将该 section 放到 text segment。实际上这是链接脚本的一个简化，当没有为 section 指定 segment 时，以上一个明确指定的 segment 为准。例如之前最后一次明确指定 segment 的__ex_table section 指定了 text segment，则其后没有指定 segment 的 section 也被放到了 text segment，直到下一次明确指定 segment 的 section 出现为止。

3.3.4 Data Segment 的构建

从现在开始，所有的section都归属于data segment。与text segment不同的是，data segment的所有 section 都是可写的。实际上我已经不需要继续写下去，因为 data segment 的创建中并没有新奇的链接脚本语法出现。有趣的是，我们发现大量的代码编译后产生的二进制也被放到了 data segment（从常理看，它们应该被放到 text segment）。这些代码在内核里非常常见，都被__init 宏或__initcall 宏修饰。内核把它们放到 data segment 的原因很简单，它们只在初始化阶段有用，一旦进入正常运行阶段，这些代码所在的页面将被回收以作它用。同样会被回收的 section 还有被__initdata、__setup_param 等宏修饰的变量。这些宏会生产名为.init.*或*.init 的自定义 section，内核在初始化完成后回收它们占用的页面。如何回收这些页面的内容不在本文讨论范围之内，感兴趣的朋友可以 grep free_initmem()函数，看看内核

在何时调用它。这里我们举自定义 section 的第三个例子，从技术上来说它和前两个例子并没有什么差别，它较常见于内核代码但多数人不一定了解它的原理，故这里特别提出来说一下。

我们经常在驱动或内核子系统的代码中看到由__initcall、fs_initcall、arch_initcall 等类似的宏修饰的函数名，例如：

```
fs_initcall(acpi_event_init);
```

很多资料告诉我们这些宏定义了函数的初始化级别，内核会在初始化的不同阶段调用它们，级别从 0~7 不等。实际上这也是自定义 section 的应用，原理跟内核符号表的创建一样。寻根究底，这些宏都是由宏__define_initcall 生成的，其定义如下：

```
#define __define_initcall(level,fn,id) \
        static initcall_t __initcall_##fn##id __used \
        __attribute__((__section__(".initcall" level ".init"))) = fn
```

其中 level 参数即 0~7（实际是 0~7s）共 14 个级别，这样内核在编译时会生产 14 个名为".initcall.(0~7s).init"的 section，例如.initcall0.init、.initcall2s.init 等。被不同宏修饰的函数被放到对应的 section 中，例如上例的 fs_initcall(acpi_event_init)最终会被放到.initcall5.init。

链接脚本的如下代码：

```
.initcall.init : AT(ADDR(.initcall.init) - LOAD_OFFSET) {
    __initcall_start = .;
    INITCALLS
    __initcall_end = .;
}
```

生成了 initcall 表，起始地址和结束地址存在__initcall_start 和__initcall_end 中。很多资料说根据 level 参数的值不同，内核在不同阶段调用这些函数。但从代码来看，我认为并非如此，内核只分两个阶段调用，即 early initcall 阶段和剩余阶段（level = 0~7s）。感兴趣的朋友可以看看上面 INITCALLS 宏的展开以及 do_initcalls()、do_pre_smp_initcalls()两个函数，很容易就能明白。

在构建 data segment 的最后部分，我们看到如下代码：

```
.bss : AT(ADDR(.bss) - LOAD_OFFSET) {
   __init_end = .;
   __bss_start = .;            /* BSS */
   *(.bss.page_aligned)
   *(.bss)
   . = ALIGN(4);
   __bss_stop = .;
   _end = . ;
   /* This is where the kernel creates the early boot page tables */
   . = ALIGN(PAGE_SIZE);
   pg0 = . ;
}
```

它告诉我们.bss section 位于 data segment 的最后，变量 pg0 存放的是“Provisional kernel Page Tables”的地址，不熟悉的朋友可以阅读 ULK3 的 2.5.5.1 节。

最后，我们列出几个著名的由链接脚本提供的全局变量：

名称	描述
_text	text segment 的起始地址，也是内核 image 的起始地址
_etext	内核代码段的结束地址，仅仅是代码段，因为 text segment 还包含.rodata、exception table、note segment
_edata	不好描述具体含义，见下图
_end	内核 image 的结束地址

上述描述不一定准确，实际上你只要在链接脚本中一看它们出现的位置就能很快知道其含义，也可以在合适的位置打印它们的值验证一下，例如 setup_arch()函数中。读过 ULK 的朋友一定想起一副熟悉的图，把它粘贴如下：

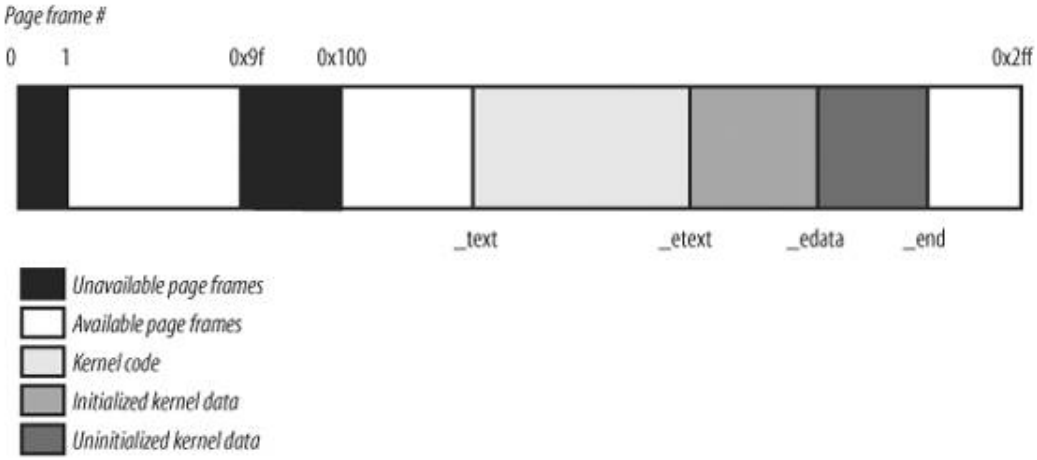

图 4. 著名的全局变量布局（摘自《Understanding Linux Kernel》）

前面我们提过并非所有输入文件中的 section 都会出现在目标文件中，对于不感兴趣的 section，链接脚本用下列代码抛弃它们。

```
/* Sections to be discarded */
   /DISCARD/ : {
      *(.exitcall.exit)
      }
```

4. 做一些尝试

通过学习内核链接脚本，笔者最大的收获不是了解了内核 image 的布局，而是通过自定义的 section 并配合链接脚本来构建动态表的方式（之所以说是动态表，是因为它的长度由加入该 section 的元素个数决定，并非事先定义好的）。也许你说一个全局的大数组也可以做到，但这样坏处是数组要预定义到最够大，其次它占用的内存在内核运行时释放不掉，最糟糕的是这个数组必须在整个内核空间共享，这样你才能在需要往里添加元素的时候访问到它。这种污染整个名字空间的设计无疑是糟糕的，把工作交给链接器是最好的选择。

最后我建议感兴趣的朋友尝试试试这种方式，把你自定义的 section 放到 data segment，你会发现数据在 section 中的排列顺序和链接器从输入文件中抽取 section 的顺序有关。嗯，exception 文档提到.text section 没有这个问题，还需要研究研究 ……

参考文献

[1]. Linker Scripts, http://www.bilmuh.gyte.edu.tr/gokturk/introcpp/gcc/ld_3.html#SEC38

[2]. Vendor-specific ELF Note Elements, http://netbsd.org/docs/kernel/elf-notes.html