Linux 2.6 内核模块载入过程解析

前言

本来是在整理以前读 2.4 内核的 Loadable Kernel Module(LKM)的载入过程的笔记的，为了与时俱进，就想看一下在 2.6 内核下 LKM 是怎样被载入的。不看不要紧，一看下一跳。不但整个载入过程被彻底重新写过，而且实现机制也完全变掉了。就连原来载入的工具包的名字也从“modutils”变成了“module-init-tools”。在 2.4 内核中载入的核心工作实际上是在 insmod utility 的用户态完成的，而到了 2.6，insmod 用户态的代码简单得不能再简单，99%的工作全部移到了内核中去完成。我没有去分析或到网上去查那些 Linux 大师们的想法，但单从代码上看，我觉得这次改变是成功的 --- 载入的思路比以前清晰多了，代码也比 2.4 时干净简洁（2.4 的 insmod 代码给人零乱而琐碎，当然或许是我的水平太低）。

在 2.4 中，为了支持 LKM，内核开放出如下的 system call

```
SYS(sys_create_module, 2)
SYS(sys_init_module, 5)
```

```
SYS(sys_delete_module, 1)
SYS(sys_get_kernel_syms, 1)
```

到了 2.6，由于实现机制的变化，内核只需要开放如下 systam call 即行。

```
cond_syscall(sys_init_module);
cond_syscall(sys_delete_module);
```

原来 2.4 中的 sys_create_module（）是一个莫名其妙的 system call，它实际上几乎不会被单独用到，它存在的目的就是为了给要载入的 LKM 分配内核空间的内存。而 sys_get_kernel_syms（）是为了获得内核及已载入 LKM 的输出 symbol。这两个 system call 的存在纯粹是因为载入 LKM 的工作在 2.4 内核中被拆分成 insmod 的用户态部分与内核部分（即上面的系统调用）。两者及其紧密的交互，在用户态没法干或没法获得的，则要委托内核来干。比如用户态没办法分配内核空间的内存，所以要有 sys_create_module（），用户态没办法获得在内核的 symbol，所以要有 sys_get_kernel_syms（）。而很大一部分工作在用户态工作在用户态完成。整个 2.4 中载入 LKM 的工作方式如下图中的左图，而 2.6 工作方式如下图中的右图。

2.4 的工作方式

2.6 的工作方式

用户空间

内核空间

由于整个的工作全部的移到内核中，不需要如 2.4 中那样的互动，所以索性

```
SYS(sys_create_module, 2)
SYS(sys_get_kernel_syms, 1)
```

两个 system call 被移走。

我觉得阅读 LKM 的载入过程的代码，对 Linux 下驱动程序的开发者是有很大好处的，尤其是刚开始学着写 Linux 下 LKM 的 programmer。因为我在论坛里老是看到一些刚开始学习 LKM Programming 的 Linux 爱好者提各种各样的与 LKM 开发相关的问题，有与开发环境相关的，有问我是按书上的例子输入的代码，编译也通过了，但就是载入失败，等等，不一而足。其实这些问题在 LKM 的载入过程的代码中都能找到答案。唉，有时候真的是求人不如求己!

Linux LKM 的载入过程代码比较琐碎（2.6 的代码实在已经简洁多了），也不难。关键是读者是否熟悉 ELF 文件中的重定位文件的格式。如果了解的话，应该阅读代码没什么大问题。

LKM是什么

LKM 的全称是 Loadable Kernel Module，可载入内核模块，也即是 Linux 下的驱动程序。它就相当于 Windows 下以.sys 为后缀名的文件。但它又实在与 Windows 下的驱动程序不一样。单从文件的形式上来说就大相径庭。Windows 下的.sys 文件也被称为内核态的动态链接库(kernel dll)，它毕竟是一个可执行文件，只不过是 PE(Portable Executable)格式的而已。但 Linux 下的驱动程序根本不是可执行文件，在 ELF Specification 中的标准学名叫“relocatable file”，即可重定位文件（在 Linux 下普通的可执行文件叫 executable file，类似 Windows 下 dll 的文件叫共享库）。其实就是 Programmer 在编译代码时的中间产物 obj 文件。是的，Linux 下的驱动程序的形式是 compiler 的产物，它没有经过 linker 的处理。最起码在刚开始接触 LKM 时，对这一点，我是蛮惊讶的。我们可以 dump 出 Linux 下三种 ELF 格式文件的汇编码来看看。

1. 普通可执行文件

以最简单的“Hello World”为例。

```
[wzhou@localhost hello]$ objdump -d hello

。。。

8048352:      c3                   ret
 8048353:      90                    nop

08048354 <main>:
 8048354:      8d 4c 24 04           lea    0x4(%esp),%ecx
 8048358:      83 e4 f0              and    $0xfffffff0,%esp
 804835b:      ff 71 fc              pushl  0xfffffffc(%ecx)
 804835e:      55                   push   %ebp
 804835f:      89 e5                 mov    %esp,%ebp
 8048361:      51                   push   %ecx
 8048362:      83 ec 04              sub    $0x4,%esp
 8048365:      c7 04 24 50 84 04 08    movl   $0x8048450,(%esp)
 804836c:      e8 23 ff ff ff         call   8048294 <puts@plt>
 8048371:      b8 00 00 00 00         mov    $0x0,%eax
 8048376:      83 c4 04              add    $0x4,%esp
 8048379:      59                   pop    %ecx
 804837a:      5d                   pop    %ebp
 804837b:      8d 61 fc              lea    0xfffffffc(%ecx),%esp
 804837e:      c3                   ret
 804837f:      90                   nop

08048380 <__libc_csu_fini>:
```

```
 8048380:      55                   push   %ebp
 8048381:      89 e5                mov    %esp,%ebp
 8048383:      5d                   pop    %ebp


。。。
```

在最左边的是该程序的某条指令的地址。linker 在生成可执行文件时已经为其安排好了固定的地址。不用载入该程序，我们都知道那条指令在那个位置。（最近听说有了可重定位的 ELF 可执行文件，不过我到目前为止还从来没看到过。）

2. 普通共享库

以 Linux 下标准 C 库为例。

```
00015d10 <__libc_fini>:
  15d10:      55                   push   %ebp
  15d11:      89 e5                mov    %esp,%ebp
  15d13:      56                   push   %esi
  15d14:      53                   push   %ebx
  15d15:      e8 56 00 00 00         call   15d70 <__i686.get_pc_thunk.bx>
  15d1a:      81 c3 da 32 12 00      add    $0x1232da,%ebx
  15d20:      8b 83 3c e2 ff ff      mov    0xffffe23c(%ebx),%eax
  15d26:      85 c0                test   %eax,%eax
  15d28:      74 12                je     15d3c <__libc_fini+0x2c>
  15d2a:      8d b3 3c e2 ff ff      lea    0xffffe23c(%ebx),%esi
  15d30:      ff d0                call   *%eax
  15d32:      8b 46 04             mov    0x4(%esi),%eax
  15d35:      83 c6 04             add    $0x4,%esi
  15d38:      85 c0                test   %eax,%eax
  15d3a:      75 f4                jne    15d30 <__libc_fini+0x20>
```

```
  15d3c:      5b                  pop   %ebx
  15d3d:      5e                  pop   %esi
  15d3e:      5d                  pop   %ebp
  15d3f:      c3                  ret
```

这里最左边的也是地址，虽然这地址只是 linker 给出的，以该共享库被载入 0 地址为基准的地址。但毕竟也是地址吗。

3. 可重定位文件

以例子 LKM hello-5.ko 为例。

```
hello-5.ko:    file format elf32-i386

Disassembly of section .text:

00000000 <export_sample>:
   0:   55                   push   %ebp
   1:   b8 01 00 00 00         mov    $0x1,%eax
   6:   89 e5                 mov    %esp,%ebp
   8:   5d                   pop    %ebp
   9:   c3                   ret
   a:   90                   nop
   b:   90                   nop
Disassembly of section .exit.text:

00000000 <cleanup_module>:
   0:   55                   push   %ebp
   1:   89 e5                 mov    %esp,%ebp
   3:   83 ec 04              sub    $0x4,%esp
   6:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
```

```
   d:   e8 fc ff ff ff         call   e <cleanup_module+0xe>
  12:   c9                   leave
  13:   c3                   ret
Disassembly of section .init.text:

00000000 <init_module>:
   0:   55                   push   %ebp
   1:   89 e5                mov    %esp,%ebp
   3:   83 ec 08              sub    $0x8,%esp
   6:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
   d:   e8 fc ff ff ff         call   e <init_module+0xe>
  12:   0f bf 05 0c 00 00 00    movswl 0xc,%eax
  19:   c7 04 24 24 00 00 00    movl   $0x24,(%esp)
  20:   89 44 24 04           mov    %eax,0x4(%esp)
  24:   e8 fc ff ff ff         call   25 <init_module+0x25>
  29:   a1 08 00 00 00         mov    0x8,%eax
  2e:   c7 04 24 15 00 00 00    movl   $0x15,(%esp)
  35:   89 44 24 04           mov    %eax,0x4(%esp)
  39:   e8 fc ff ff ff         call   3a <init_module+0x3a>
  3e:   a1 04 00 00 00         mov    0x4,%eax
  43:   c7 04 24 48 00 00 00    movl   $0x48,(%esp)
  4a:   89 44 24 04           mov    %eax,0x4(%esp)
  4e:   e8 fc ff ff ff         call   4f <init_module+0x4f>
  53:   a1 00 00 00 00         mov    0x0,%eax
  58:   c7 04 24 31 00 00 00    movl   $0x31,(%esp)
  5f:   89 44 24 04           mov    %eax,0x4(%esp)
```

```
 63:   e8 fc ff ff ff         call   64 <init_module+0x64>
 68:   31 c0                 xor    %eax,%eax
 6a:   c9                    leave
 6b:   c3                    ret
```

上面的代码还是以 section 为单位，最左边的只是该条指令在其所在 section 的偏移。而前面两种 ELF 文件中的指令地址都是单一于其文件载入首地址的，已经没有了 section 的概念。

列举了 ELF 的三种文件类型，主要想说明，可重定为文件（obj 文件）本来是一种中间产物文件，但 Linux（好象 Unix 都是这样，最起码我知道的 FreeBSD，Solaris 是这样的）中的设备驱动就是以它为最终产品形式的。自然其载入器也就必须充当 linker 的角色。所以你在了解了 LKM 的载入过程后，也会对 programmer 天天用到的 linker 会有所体会，毕竟开发过 linker 的 programmer 还是极少数人。

内核模块载入流程分析

由于 LKM 是可执行代码的半成品，所以要真正可执行自然要依赖于 LKM 载入器的帮助。在 2.6 版本的 Linux 中，LKM 载入器分成两部分，一个就是 insmod(install module) utility，另一个就是在内核中的 sys_init_module()的系统调用。

insmod 部分完成的工作非常简单，一言以蔽之就是把整个 LKM 文件读入内存，然后调用 sys_init_module()。相对 2.4 版本的 Linux 而言，简单得在初始时读代码时我几乎有点不敢相信。

下面是 sys_init_module()完成的工作。

1. 检查用户是否有载入 LKM 的权限。
2. 检查要载入的 LKM 不会太大，超过 64M。合法则在内核中通过 vmalloc（）分配 LKM 文件大小的内存。
3. 通过 copy_from_user（）把 LKM 从用户空间拷贝到内核空间。
4. 对已经在内核空间的 LKM 文件进行一系列合法性检查。比如是否是 ELF 文件，是否是可重定位文件，该文件中的代码是否是为正运行的 CPU 编译的，

section table 是否合法（该文件是完整的，没有被截断等）。

5. 更新 section header table 中的所有 header 的 sh_addr field，使它由原来指向该 section 在文件中的偏移变成指向现在被载入内核空间的 LKM 的 image 内的偏移。并且找到 symbol section 与 symbol section 中用到的字符串所在的 section。
6. 寻找.gnu.linkonce.this_module section，如果没有该 section 与 symbol section 则不能载入该 LKM。
7. 寻找各种各样的 section，有的是必须的，有的则是可选的。
8. 如果编译的内核 symbol 带版本信息，则要验证载入的 LKM 得 symbol 是否与该版本匹配（很多载入 LKM 失败原因即在此）。
9. 检查编译 LKM 的版本与当前内核的版本是否匹配。
10. 从用户空间拷贝传给 insmod 的参数列表到内核空间。
11. 检验相同模块名的 LKM 是否已经被载入。
12. LKM 支持 Per-CPU data。为 Per-CPU data 分配空间。Per-CPU data 是为系统中每个 CPU 都分配的空间，为的是减少数据的共享，以极大的减少访问数据时申请占有锁。
13. layout_sections（）统计需要载入内存的 LKM 的大小，分为常驻部分和初始化以后即可释放的部分。
14. 分别为常驻部分与初始化部分分配空间。
15. 从 LKM 文件 image 中拷贝相应的 section 到常驻部分和初始化部分。
16. 根据该 LKM 是否遵循类 GPL 来设置该模块是否被“污染”标志。Linux 内核认为不遵守类 GPL 则该 LKM 是个“污染源”。
17. “ndiswrapper”“ driverloader”是两个为 Windows 下的无限网卡驱动能在 Linux 下运行而载入的比较知名的“污染源”，这里特意点名注意。
18. simplify_symbols()搜索 LKM 的输入的 symbol，获得他们的地址。本来他们都悬空着。
19. 根据上面已经获得的输入 symbol 的地址来 fix(修补)obj 文件中的重定位项，即对内核和其他 LKM 中函数的调用。
20. 把 LKM 要输出的 symbol 准备好，以便外部 LKM 能引用。
21. 对 insmod 传递的参数的处理。
22. mod_sysfs_setup()应该是把该模块挂到/sys/modules 目录下吧？（对 2.6 才引入的 sys 系统不太了解）。
23. 通知系统内对模块的载入与卸载感兴趣的 component(通过向 module_notify_list 来注册)，有新模块载入了。
24. 调用该 LKM 中的初始化入口函数。
25. 释放 LKM 文件 image 在内核空间中的内存。

内核模块载入源代码注解

module-init-tools-3.2\insmod.c

```
33 #define streq(a,b) (strcmp((a),(b)) == 0)
34
35 extern long init_module(void *, unsigned long, const char *);
36
37 static void print_usage(const char *progname)
38 {
39     fprintf(stderr, "Usage: %s filename [args]\n", progname);
40     exit(1);
41 }
42
43 /* We use error numbers in a loose translation... */
44 static const char *moderror(int err)
45 {
46     switch (err) {
47     case ENOEXEC:
48         return "Invalid module format";
49     case ENOENT:
50         return "Unknown symbol in module";
51     case ESRCH:
52         return "Module has wrong symbol version";
```

```
53     case EINVAL:
54         return "Invalid parameters";
55     default:
56         return strerror(err);
57     }
58 }
59
```

该函数把 LKM 载入内存，并在*size 中记录该 LKM 的大小。

```
60 static void *grab_file(const char *filename, unsigned long *size)
61 {
62     unsigned int max = 16384;
63     int ret, fd;
64     void *buffer = malloc(max);
65     if (!buffer)
66         return NULL;
67
68     if (streq(filename, "-"))
69         fd = dup(STDIN_FILENO);
70     else
71         fd = open(filename, O_RDONLY, 0);
72
73     if (fd < 0)
74         return NULL;
75
76     *size = 0;
77     while ((ret = read(fd, buffer + *size, max - *size)) > 0) {
```

```
78          *size += ret;
79          if (*size == max)
80              buffer = realloc(buffer, max *= 2);
81      }
82      if (ret < 0) {
83          free(buffer);
84          buffer = NULL;
85      }
86      close(fd);
87      return buffer;
88  }
89
90  int main(int argc, char *argv[])
91  {
92      unsigned int i;
93      long int ret;
94      unsigned long len;
95      void *file;
96      char *filename, *options = strdup("");
97      char *progname = argv[0];
98
99      if (strstr(argv[0], "insmod.static"))
100         try_old_version("insmod.static", argv);
101     else
102         try_old_version("insmod", argv);
103
```

```
104    if (argv[1] && (streq(argv[1], "--version") || streq(argv[1], "-V"))) {
105       puts(PACKAGE " version " VERSION);
106       exit(0);
107    }
108
109    /* Ignore old options, for backwards compat. */
110    while (argv[1] && (streq(argv[1], "-p")
111              || streq(argv[1], "-s")
112              || streq(argv[1], "-f"))) {
113       argv++;
114       argc--;
115    }
116
117    filename = argv[1];                    要载入的 LKM
118    if (!filename)
119       print_usage(progname);
120
121    /* Rest is options */
122    for (i = 2; i < argc; i++) {
123       options = realloc(options,
124                strlen(options) + 1 + strlen(argv[i]) + 1);
125       strcat(options, argv[i]);
126       strcat(options, " ");
127    }
       把在载入 LKM 时的参数 merge 在 options 字符串中。
128
```

```
129    file = grab_file(filename, &len);    把整个要载入的 LKM 文件读入内存，该文件的长度在 len 中
130    if (!file) {
131       fprintf(stderr, "insmod: can't read '%s': %s\n",
132          filename, strerror(errno));
133       exit(1);
134    }
135
136    ret = init_module(file, len, options);
       调用 system call，即内核中的 sys_init_module()中去处理，传递 3 个参数
       1。file 为要载入 LKM 的内存映像
       2。len 为 LKM 的内存映像的长度
       3。在载入 LKM 时传递的参数

137    if (ret != 0) {
138       fprintf(stderr, "insmod: error inserting '%s': %li %s\n",
139          filename, ret, moderror(errno));
140       exit(1);
141    }
142    exit(0);
143 }
```

src\linux-2.6.20\kernel\module.c

```
1955   /* This is where the real work happens */
1956   asmlinkage long
1957   sys_init_module(void __user *umod,
```

```
1958          unsigned long len,
1959          const char __user *uargs)
1960  {
1961      struct module *mod;
1962      int ret = 0;
1963
1964      /* Must have permission */
1965      if (!capable(CAP_SYS_MODULE))
1966          return -EPERM;
          检查是否用户拥有载入 LKM 的权利。一般只有 root 才拥有该权利。
1967
1968      /* Only one module load at a time, please */
1969      if (mutex_lock_interruptible(&module_mutex) != 0)
1970          return -EINTR;
1971
1972      /* Do all the hard work */
1973      mod = load_module(umod, len, uargs);
          该函数几乎完成载入 LKM 的所有工作。
1974      if (IS_ERR(mod)) {
1975          mutex_unlock(&module_mutex);
1976          return PTR_ERR(mod);
1977      }
1978
1979      /* Now sew it into the lists.  They won't access us, since
1980             strong_try_module_get() will fail. */
1981      stop_machine_run(__link_module, mod, NR_CPUS);
```

```
1982
1983      /* Drop lock so they can recurse */
1984      mutex_unlock(&module_mutex);
1985
1986      blocking_notifier_call_chain(&module_notify_list,
1987              MODULE_STATE_COMING, mod);
1988
1989      /* Start the module */
1990      if (mod->init != NULL)
1991         ret = mod->init();
```

这儿就是调用 LKM 的初始化函数的地方。也就是你在代码中用 module_init（function）表示的 function。在调试时由于要 debug 的 LKM 还未载入，debugger(KDB 或 kgdb)自然不会认识你的初始化函数。一个比较有效的办法就是在这里设断点，这样可以调试你的 LKM 的初始化函数。

```
1992      if (ret < 0) {
1993         /* Init routine failed: abort.  Try to protect us from
1994                  buggy refcounters. */
1995         mod->state = MODULE_STATE_GOING;
1996         synchronize_sched();
1997         if (mod->unsafe)
1998            printk(KERN_ERR "%s: module is now stuck!\n",
1999                  mod->name);
2000         else {
2001            module_put(mod);
2002            mutex_lock(&module_mutex);
2003            free_module(mod);
2004            mutex_unlock(&module_mutex);
```

```
2005          }
2006          return ret;
2007      }
2008
2009      /* Now it's a first class citizen! */
2010      mutex_lock(&module_mutex);
2011      mod->state = MODULE_STATE_LIVE;
2012      /* Drop initial reference. */
2013      module_put(mod);
2014      unwind_remove_table(mod->unwind_info, 1);
2015      module_free(mod, mod->module_init);
2016      mod->module_init = NULL;
2017      mod->init_size = 0;
2018      mod->init_text_size = 0;
2019      mutex_unlock(&module_mutex);
2020
2021      return 0;
2022  }
```

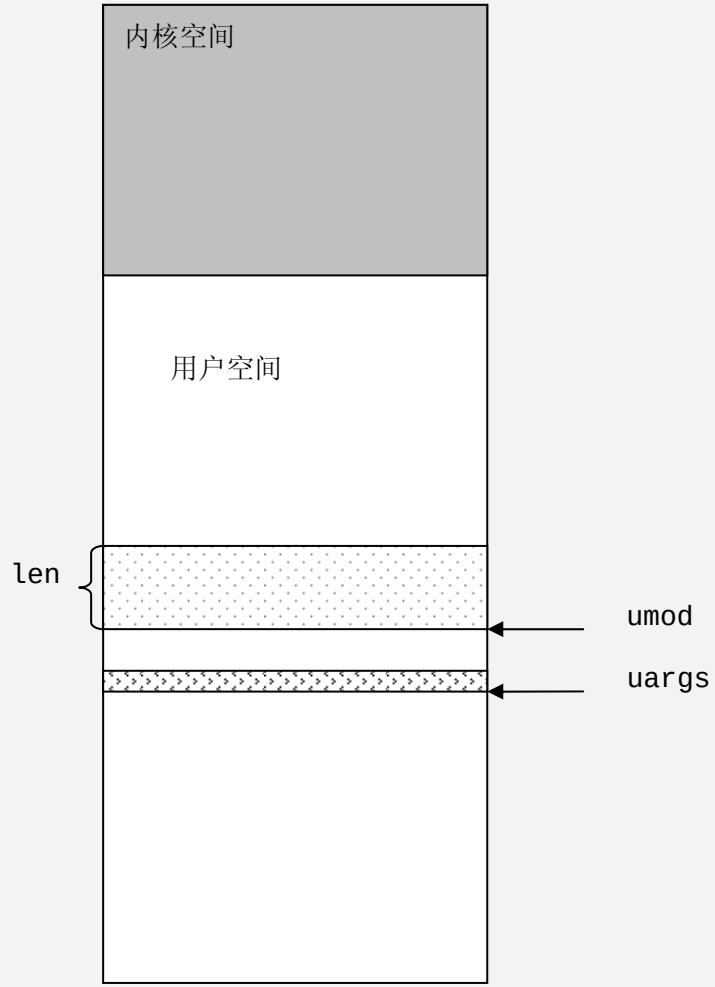
src\linux-2.6.20\kernel\module.c

这个函数是载入 LKM 的最核心函数。其实际上是一个 LKM 的 dynamic loader，负责把一堆没有地址空间概念的可重定位代码嵌入到 kernel 空间中，并解决代码中对其他模块或内核的函数调用，同时如果它本身又提供输出函数以供其它 LKM 调用，则还要输出这些函数。
本函数完全是过程性的，比较琐碎，如果你对 ELF 格式比较了解，整个代码还是蛮好理解的。

```
1534  /* Allocate and load the module: note that size of section 0 is always
```

```
1535     zero, and we rely on this for optional sections. */
1536   static struct module *load_module(void __user *umod,
1537                 unsigned long len,
1538                 const char __user *uargs)
1539   {
1540      Elf_Ehdr *hdr;
1541      Elf_Shdr *sechdrs;
1542      char *secstrings, *args, *modmagic, *strtab = NULL;
1543      unsigned int i;
1544      unsigned int symindex = 0;
1545      unsigned int strindex = 0;
1546      unsigned int setupindex;
1547      unsigned int exindex;
1548      unsigned int exportindex;
1549      unsigned int modindex;
1550      unsigned int obsparmindex;
1551      unsigned int infoindex;
1552      unsigned int gplindex;
1553      unsigned int crcindex;
1554      unsigned int gplcrcindex;
1555      unsigned int versindex;
1556      unsigned int pcpuindex;
1557      unsigned int gplfutureindex;
1558      unsigned int gplfuturecrcindex;
1559      unsigned int unwindex = 0;
1560      unsigned int unusedindex;
```

```
1561      unsigned int unusedcrcindex;
1562      unsigned int unusedgplindex;
1563      unsigned int unusedgplcrcindex;
1564      struct module *mod;
1565      long err = 0;
1566      void *percpu = NULL, *ptr = NULL; /* Stops spurious gcc warning */
1567      struct exception_table_entry *extable;
1568      mm_segment_t old_fs;
1569
          在整个函数开工以前，初始状态如下图（figure 1）
```



insmod 已经把要载入的 LKM 读入内存，由 umod 指针指向，其长度为 len，传递给该 LKM 的参数被 merge 在 uargs 指向的字符串中

```
1570      DEBUGP("load_module: umod=%p, len=%lu, uargs=%p\n",
1571             umod, len, uargs);
```

下面对 LKM 进行一系列的合法性检查。这些检查完全基于 ELF 格式。

```
1572      if (len < sizeof(*hdr))
1573         return ERR_PTR(-ENOEXEC);
```

LKM 是 ELF 格式中的可重定位格式，也就是俗语中的 obj 文件，即 compiler 的输出，注意是 compiler 的输出，而非 linker 的输出。我们常规的可执行文件是 linker 的输出。

比如：

```
[wzhou@dcmp10 ~]$ file /lib/modules/2.6.9-5.EL/kernel/fs/fat/fat.ko
/lib/modules/2.6.9-5.EL/kernel/fs/fat/fat.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
not stripped
```

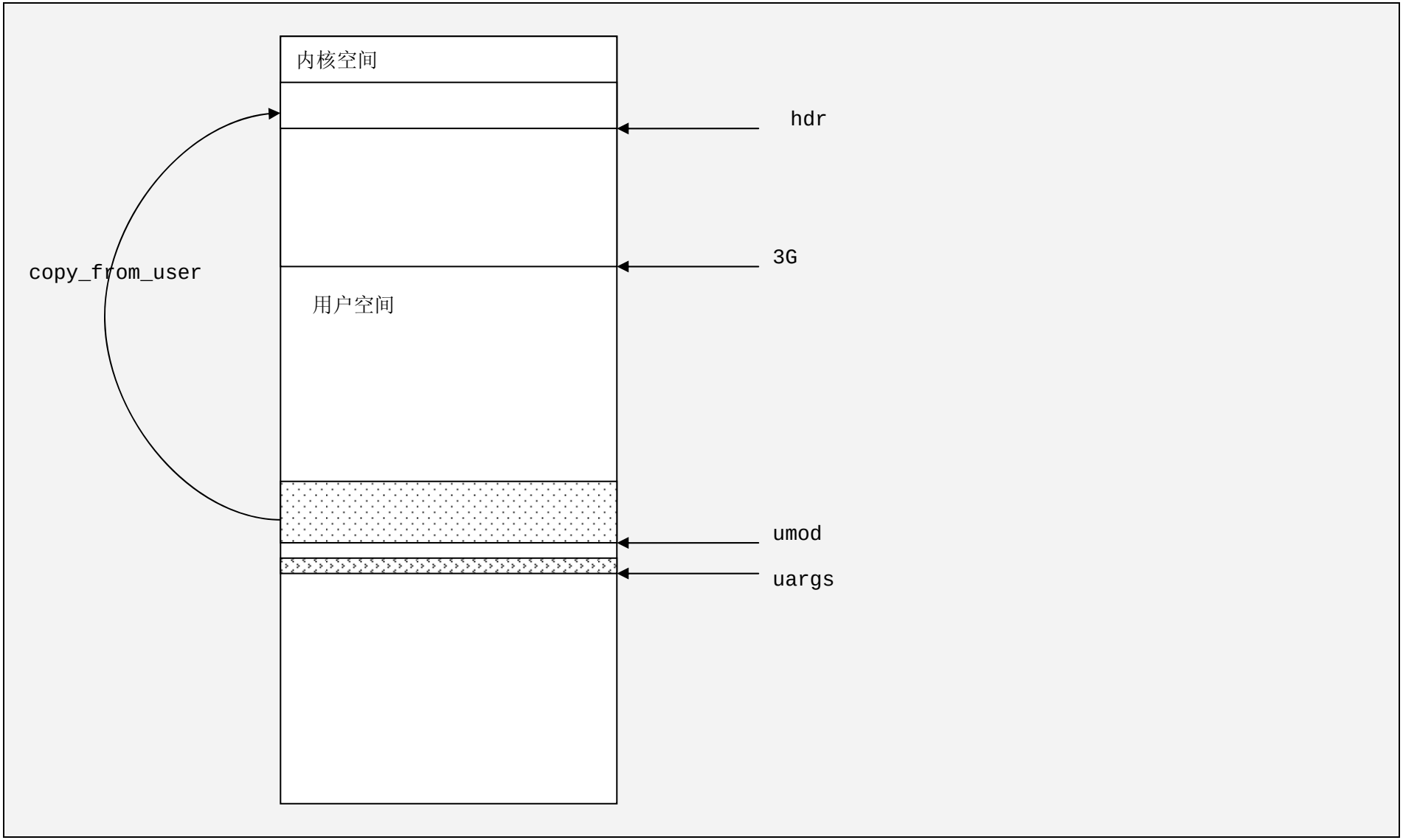
sizeof(*hdr)，即 ELF header，ELF 文件长度小于 ELF header 的长度，当然非法。

```
1574
1575      /* Suck in entire file: we'll want most of it. */
1576      /* vmalloc barfs on "unusual" numbers.  Check here */
1577      if (len > 64 * 1024 * 1024 || (hdr = vmalloc(len)) == NULL)
1578         return ERR_PTR(-ENOMEM);
```

从这个检查，可以看出 LKM 不能大于 64M。 这里为该 LKM 分配内核空间。通过 vmalloc()分配的内核空间是虚拟地址连续，但物理地址并不一定连续。

```
1579      if ((hdr, umod, len) != 0) {
1580         err = -EFAULT;
1581         goto free_hdr;
1582      }
```

把 LKM 从用户空间拷贝到内核空间。见下图。


内核空间
hdr
3G
copy_from_user
用户空间
umod
uargs

```
1583
1584      /* Sanity checks against insmoding binaries or wrong arch,
1585            weird elf version */
1586      if (memcmp(hdr->e_ident, ELFMAG, 4) != 0
1587          || hdr->e_type != ET_REL
1588          || !elf_check_arch(hdr)
1589          || hdr->e_shentsize != sizeof(*sechdrs)) {
1590          err = -ENOEXEC;
1591          goto free_hdr;
1592      }
          memcmp(hdr->e_ident, ELFMAG, 4) != 0 检查是否有 ELF 的签名
          hdr->e_type != ET_REL 检查是否是可重定位文件
          elf_check_arch(hdr) 检查是否运行在合适的 CPU 上
          hdr->e_shentsize != sizeof(*sechdrs) ELF 文件头中的 section header size 合法
1593
1594      if (len < hdr->e_shoff + hdr->e_shnum * sizeof(Elf_Shdr))
1595          goto truncated;
          ELF 文件中的 section table 没有被截断。hdr->e_shoff 为 section header table 在文件中的偏移。hdr->e_shnum 为有多少
          项 section header。

          合法性检查完毕，开始分析 LKM 中的各个 section。由于 LKM 只是 obj 文件，所以没有反映链接特性的 segment(段)。举例如下：


[wzhou@localhost example]$ readelf -S hello-5.ko
There are 35 section headers, starting at offset 0xf3c8:
```

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000000 00  AX  0   0  4
  [ 2] .exit.text        PROGBITS        00000000 000040 000014 00  AX  0   0 16
  [ 3] .rel.exit.text    REL             00000000 00f940 000010 08     33   2  4
  [ 4] .init.text        PROGBITS        00000000 000060 00006c 00  AX  0   0 16
  [ 5] .rel.init.text    REL             00000000 00f950 000070 08     33   4  4
  [ 6] .rodata.str1.1    PROGBITS        00000000 0000cc 000053 01 AMS  0   0  1
  [ 7] .rodata.str1.4    PROGBITS        00000000 000120 00006a 01 AMS  0   0  4
  [ 8] .modinfo          PROGBITS        00000000 0001a0 000144 00   A  0   0 32
  [ 9] __param           PROGBITS        00000000 0002e4 000050 00   A  0   0  4
  [10] .rel__param       REL             00000000 00f9c0 000080 08     33   9  4
  [11] .data             PROGBITS        00000000 000334 00002c 00  WA  0   0  4
  [12] .rel.data         REL             00000000 00fa40 000008 08     33  11  4
  [13] .gnu.linkonce.thi PROGBITS        00000000 000380 001200 00  WA  0   0 128
  [14] .rel.gnu.linkonce REL             00000000 00fa48 000010 08     33  13  4
  [15] .bss              NOBITS          00000000 001580 000000 00  WA  0   0  4
  [16] .comment          PROGBITS        00000000 001580 00005c 00      0   0  1
  [17] .debug_aranges    PROGBITS        00000000 0015dc 000030 00      0   0  1
  [18] .rel.debug_arange REL             00000000 00fa58 000020 08     33  17  4
  [19] .debug_pubnames   PROGBITS        00000000 00160c 00006a 00      0   0  1
  [20] .rel.debug_pubnam REL             00000000 00fa78 000010 08     33  19  4
  [21] .debug_info       PROGBITS        00000000 001676 008668 00      0   0  1
  [22] .rel.debug_info   REL             00000000 00fa88 003ca0 08     33  21  4
  [23] .debug_abbrev     PROGBITS        00000000 009cde 0004fe 00      0   0  1
```

```
  [24] .debug_line      PROGBITS       00000000 00a1dc 0007e3 00     0   0  1
  [25] .rel.debug_line   REL           00000000 013728 000010 08    33  24  4
  [26] .debug_frame     PROGBITS       00000000 00a9c0 000044 00     0   0  4
  [27] .rel.debug_frame  REL           00000000 013738 000020 08    33  26  4
  [28] .debug_str       PROGBITS       00000000 00aa04 004838 01  MS  0   0  1
  [29] .debug_loc       PROGBITS       00000000 00f23c 000058 00     0   0  1
  [30] .rel.debug_loc    REL           00000000 013758 000060 08    33  29  4
  [31] .note.GNU-stack   PROGBITS       00000000 00f294 000000 00     0   0  1
  [32] .shstrtab        STRTAB         00000000 00f294 000132 00     0   0  1
  [33] .symtab          SYMTAB         00000000 0137b8 0003d0 10    34  49  4
  [34] .strtab          STRTAB         00000000 013b88 000253 00     0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
[wzhou@localhost example]$
```

```
1596
1597      /* Convenience variables */
1598      sechdrs = (void *)hdr + hdr->e_shoff;
1599      secstrings = (void *)hdr + sechdrs[hdr->e_shstrndx].sh_offset;
1600      sechdrs[0].sh_addr = 0;
          secstrings 指向.shstrtab section，该 section 中是各个 section 的 name 的字符串。像上例中“Name”一列的字符串全部记
          录在该 section 中。
1601
```

```
1602      for (i = 1; i < hdr->e_shnum; i++) {     对整个 section table 进行枚举
1603         if (sechdrs[i].sh_type != SHT_NOBITS
1604             && len < sechdrs[i].sh_offset + sechdrs[i].sh_size)
1605             goto truncated;
          SHT_NOBITS 表示该 section 更本不占用空间。这里是对当前枚举 section 的合法性检查。如果该 section 没有全部包含在 LKM 文件
          内，则很显然该文件是被截断了。
1606
1607         /* Mark all sections sh_addr with their address in the
1608            temporary image. */
1609         sechdrs[i].sh_addr = (size_t)hdr + sechdrs[i].sh_offset;
            sechdrs[i].sh_addr 为正枚举的 section 的首地址。
            使得 section  header 中的 sh_addr 指向其在载入内核空间的 LKM 的各个对应 section 的地址。原本该 field 只是指向该
            section 在 LKM 文件中的偏移。
1610
1611         /* Internal symbols and strings. */
1612         if (sechdrs[i].sh_type == SHT_SYMTAB) {
            如果是 symbol section，则记录下来。“symbol”涉及两个 section，一个是纪录 symbol 信息的 section，也就是当前正枚举
            的 section，另一个是该 symbol 相关的字符串的 section，也就是下面 strtab 所指向的。这个具体的数据结构要参考 ELF
            Specification。
1613             symindex = i;
1614             strindex = sechdrs[i].sh_link;
1615             strtab = (char *)hdr + sechdrs[strindex].sh_offset;
1616         }
1617   #ifndef CONFIG_MODULE_UNLOAD
      即如果编译内核时没有选择可以卸载 LKM，即只能载入，不能卸载，则.exit section 实际上就不需要了，所以不需要为它分配内存，这里
      就是把是否要分配内存的标志给去掉。
```

```
1618         /* Don't load .exit sections */
1619         if (strncmp(secstrings+sechdrs[i].sh_name, ".exit", 5) == 0)
1620            sechdrs[i].sh_flags &= ~(unsigned long)SHF_ALLOC;
1621   #endif
1622      }
1623
1624      modindex = find_sec(hdr, sechdrs, secstrings,
1625                ".gnu.linkonce.this_module");
1626      if (!modindex) {
1627         printk(KERN_WARNING "No module found in object\n");
1628         err = -ENOEXEC;
1629         goto free_hdr;
1630      }
         LKM 必须有.gnu.linkonce.this_module section。？？？

1631      mod = (void *)sechdrs[modindex].sh_addr;
         取得.gnu.linkonce.this_module section 在内存中的地址。
1632
1633      if (symindex == 0) {
1634         printk(KERN_WARNING "%s: module has no symbols (stripped?)\n",
1635               mod->name);
1636         err = -ENOEXEC;
1637         goto free_hdr;
1638      }
         LKM 必须要有 symbol。
1639
```

```
1640      /* Optional sections */
1641      exportindex = find_sec(hdr, sechdrs, secstrings, "__ksymtab");
1642      gplindex = find_sec(hdr, sechdrs, secstrings, "__ksymtab_gpl");
1643      gplfutureindex = find_sec(hdr, sechdrs, secstrings, "__ksymtab_gpl_future");
1644      unusedindex = find_sec(hdr, sechdrs, secstrings, "__ksymtab_unused");
1645      unusedgplindex = find_sec(hdr, sechdrs, secstrings, "__ksymtab_unused_gpl");
1646      crcindex = find_sec(hdr, sechdrs, secstrings, "__kcrctab");
1647      gplcrcindex = find_sec(hdr, sechdrs, secstrings, "__kcrctab_gpl");
1648      gplfuturecrcindex = find_sec(hdr, sechdrs, secstrings, "__kcrctab_gpl_future");
1649      unusedcrcindex = find_sec(hdr, sechdrs, secstrings, "__kcrctab_unused");
1650      unusedgplcrcindex = find_sec(hdr, sechdrs, secstrings, "__kcrctab_unused_gpl");
1651      setupindex = find_sec(hdr, sechdrs, secstrings, "__param");
1652      exindex = find_sec(hdr, sechdrs, secstrings, "__ex_table");
1653      obsparmindex = find_sec(hdr, sechdrs, secstrings, "__obsparm");
1654      versindex = find_sec(hdr, sechdrs, secstrings, "__versions");
1655      infoindex = find_sec(hdr, sechdrs, secstrings, ".modinfo");
1656      pcpuindex = find_pcpusec(hdr, sechdrs, secstrings);
1657  #ifdef ARCH_UNWIND_SECTION_NAME
1658      unwindex = find_sec(hdr, sechdrs, secstrings, ARCH_UNWIND_SECTION_NAME);
1659  #endif
1660
1661      /* Don't keep modinfo section */
1662      sechdrs[infoindex].sh_flags &= ~(unsigned long)SHF_ALLOC;
          infoindex 指向.modinfo section，该 section 不需要处理，即无须载入内存。modinfo utility 就是读取 LKM 中的该 section
          的内容来显示该 LKM 相关信息的。
1663  #ifdef CONFIG_KALLSYMS
```

如果编译内核时打开 CONFIG_KALLSYMS 选项，则表示 symbol 都要输出，所以 LKM 的 symbol 自然必须载入内存，所以相关的 section 要分配内存。

```
1664      /* Keep symbol and string tables for decoding later. */
1665      sechdrs[symindex].sh_flags |= SHF_ALLOC;
1666      sechdrs[strindex].sh_flags |= SHF_ALLOC;
1667   #endif
1668      if (unwindex)
1669         sechdrs[unwindex].sh_flags |= SHF_ALLOC;
1670
1671      /* Check module struct version now, before we try to use module. */
1672      if (!check_modstruct_version(sechdrs, versindex, mod)) {
1673         err = -ENOEXEC;
1674         goto free_hdr;
1675      }
1676
1677      modmagic = get_modinfo(sechdrs, infoindex, "vermagic");
```

读取.modinfo section 中 vermagic=XXX 的 value。比如：

```
[wzhou@dcmp10 ~]$ /sbin/modinfo /lib/modules/2.6.9-5.EL/kernel/fs/fat/fat.ko
filename:      /lib/modules/2.6.9-5.EL/kernel/fs/fat/fat.ko
license:       GPL
vermagic:      2.6.9-5.EL 686 REGPARM 4KSTACKS gcc-3.4
depends:
```

也就是在这里 modmagic 可能是如这样的字符串“2.6.9-5.EL 686 REGPARM 4KSTACKS gcc-3.4”。

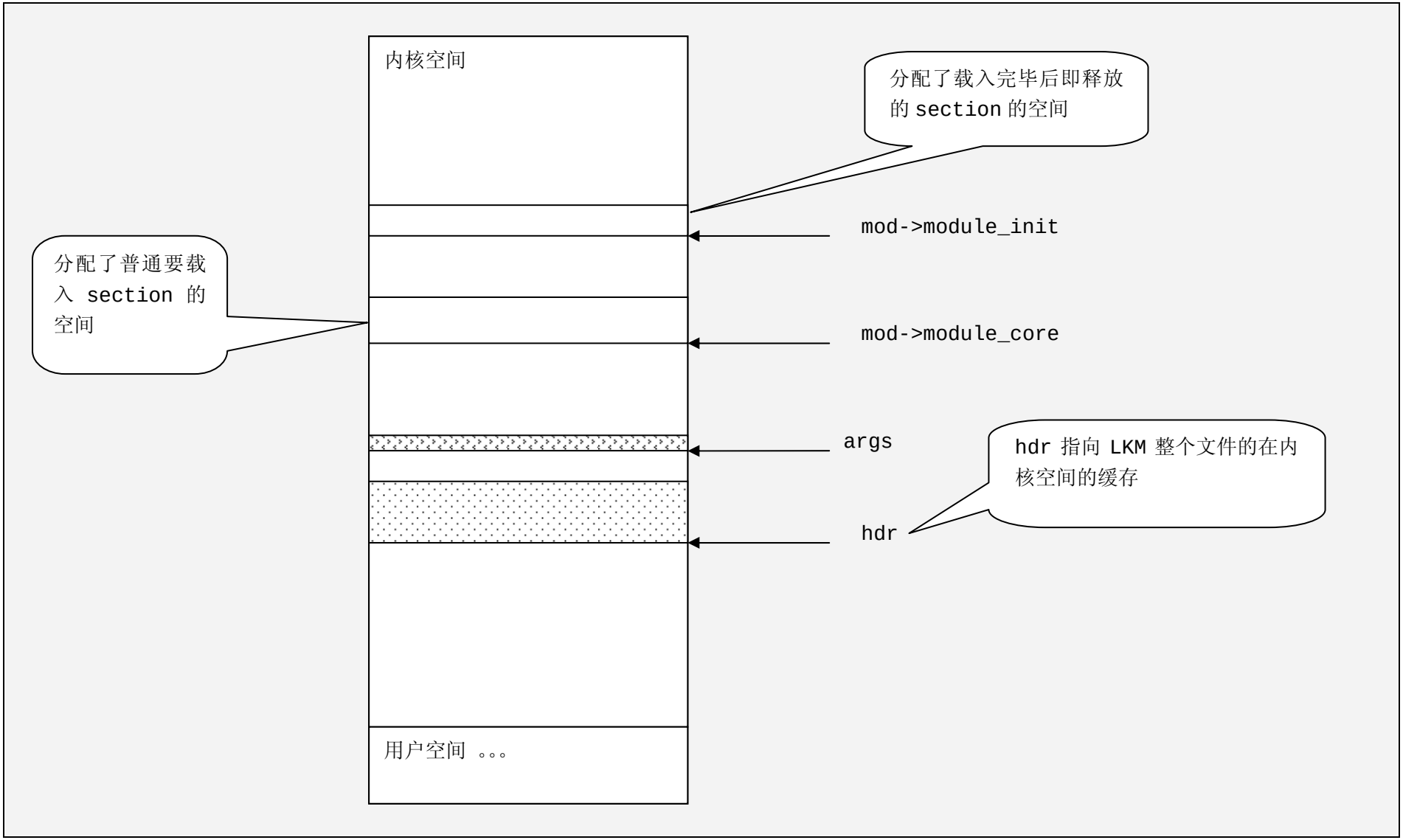
```
1678      /* This is allowed: modprobe --force will invalidate it. */
1679      if (!modmagic) {
```

```
                如果 modmagic 为空，则表示不太符合标准，所以置不安全标志。
1680            add_taint_module(mod, TAINT_FORCED_MODULE);
1681            printk(KERN_WARNING "%s: no version magic, tainting kernel.\n",
1682                   mod->name);
1683        } else if (!same_magic(modmagic, vermagic)) {
                如果 modmagic 与当前系统的“vermagic”字符串不一致，一般表示该 LKM 不是在当前的系统和平台上编译的，所以可能有问题，
                系统在这里索性拒绝载入。这一点也是 Linux 有点令人讨厌的地方。它编译的 LKM 太依赖于特定的编译平台。而不象 Windows 下，
                我在 Windows 2000 下编译的 driver 完全可以拷贝到 Windows XP 或者 Windows 2003 下来运行。
1684            printk(KERN_ERR "%s: version magic '%s' should be '%s'\n",
1685                   mod->name, modmagic, vermagic);
1686            err = -ENOEXEC;
1687            goto free_hdr;
1688        }
1689
1690        /* Now copy in args */
1691        args = strndup_user(uargs, ~0UL >> 1);
1692        if (IS_ERR(args)) {
1693            err = PTR_ERR(args);
1694            goto free_hdr;
1695        }
            把用户态的参数也拷贝到内核空间。由 args 指向。
1696
1697        if (find_module(mod->name)) {
1698            err = -EEXIST;
1699            goto free_mod;
1700        }
```

```
        查找要载入的 LKM 是否已经被载入，如果是则退出。
1701
1702    mod->state = MODULE_STATE_COMING;
1703
1704    /* Allow arches to frob section contents and sizes.  */
1705    err = module_frob_arch_sections(hdr, sechdrs, secstrings, mod);
1706    if (err < 0)
1707       goto free_mod;
1708
        在 SMP 的情况下，LKM 还能指定 per-CPU-data。
1709    if (pcpuindex) {
1710       /* We have a special allocation for this section. */
1711       percpu = percpu_modalloc(sechdrs[pcpuindex].sh_size,
1712                 sechdrs[pcpuindex].sh_addralign,
1713                 mod->name);
1714       if (!percpu) {
1715          err = -ENOMEM;
1716          goto free_mod;
1717       }
1718       sechdrs[pcpuindex].sh_flags &= ~(unsigned long)SHF_ALLOC;
1719       mod->percpu = percpu;
1720    }
1721
1722    /* Determine total sizes, and put offsets in sh_entsize.  For now
1723       this is done generically; there doesn't appear to be any
1724       special cases for the architectures. */
```

```
1725      layout_sections(mod, hdr, sechdrs, secstrings);
```
该函数实际上是统计“linker”该 LKM 真正要用到的内存。内存分为两大部分：1。LKM 运行期间一直占用的（记录在 mod->core_size 中）；2。该 LKM 载入期间会用到，但在初始化以后就可以释放的（记录在 mod->init_size 中）。具体见该函数注释。

```
1726
1727      /* Do the allocs. */
1728      ptr = module_alloc(mod->core_size);
1729      if (!ptr) {
1730         err = -ENOMEM;
1731         goto free_percpu;
1732      }
1733      memset(ptr, 0, mod->core_size);
1734      mod->module_core = ptr;
1735
1736      ptr = module_alloc(mod->init_size);
1737      if (!ptr && mod->init_size) {
1738         err = -ENOMEM;
1739         goto free_core;
1740      }
1741      memset(ptr, 0, mod->init_size);
1742      mod->module_init = ptr;
```
为真正做载入准备空间，见下图。

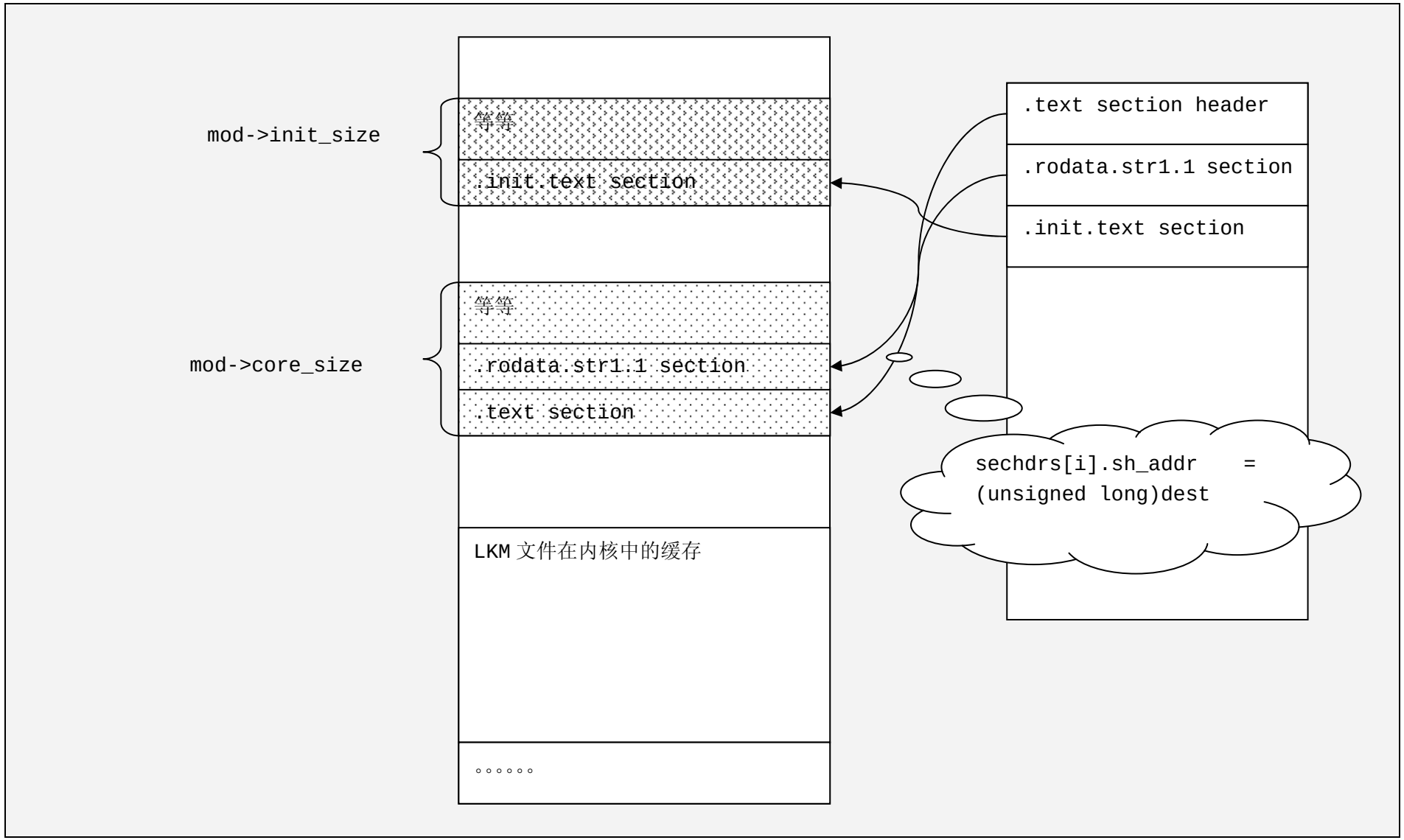

内核空间
分配了载入完毕后即释放的 section 的空间
mod->module_init
分配了普通要载入 section 的空间
mod->module_core
args
hdr 指向 LKM 整个文件的在内核空间的缓存
hdr
用户空间 。。。

```
1743
1744      /* Transfer each section which specifies SHF_ALLOC */
1745      DEBUGP("final section addresses:\n");
          下面的循环即是把符合要求的 section 从 LKM 在内核中的缓存中拷贝到上面分配的两块内存中。
1746      for (i = 0; i < hdr->e_shnum; i++) {
1747         void *dest;
1748
1749         if (!(sechdrs[i].sh_flags & SHF_ALLOC))
1750            continue;
             如果该 section 不需要载入内存则忽略。
1751
             /* If this is set, the section belongs in the init part of the module */
             #define INIT_OFFSET_MASK (1UL << (BITS_PER_LONG-1))
             在 32 位 CPU 上 BITS_PER_LONG = 32，即 INIT_OFFSET_MASK = 1 << 31，即最高为置 1 表示该 section 为初始化 section。

1752         if (sechdrs[i].sh_entsize & INIT_OFFSET_MASK)
1753            dest = mod->module_init
1754               + (sechdrs[i].sh_entsize & ~INIT_OFFSET_MASK);
1755         else
1756            dest = mod->module_core + sechdrs[i].sh_entsize;
1757
1758         if (sechdrs[i].sh_type != SHT_NOBITS)
1759            memcpy(dest, (void *)sechdrs[i].sh_addr,
1760                   sechdrs[i].sh_size);
             SHT_NOBITS 表示不占用 LKM 的文件空间，但内存空间还是要占用的，象应用程序的 bss 段，这里对这种 section，不需要处理，
             因为空间已经留出来了，而且在上面被初始化为零。对普通的 section，则要拷贝了。
```

```
1761          /* Update sh_addr to point to copy in image. */
1762          sechdrs[i].sh_addr = (unsigned long)dest;
              该 section header 的 sh_addr 成员变量指向该 section 被载入的地址。

1763          DEBUGP("\t0x%lx %s\n", sechdrs[i].sh_addr, secstrings + sechdrs[i].sh_name);
1764      }
```


mod->init_size
等等
.init.text section
等等
mod->core_size
.rodata.str1.1 section
.text section
LKM 文件在内核中的缓存
。。。。。。
.text section header
.rodata.str1.1 section
.init.text section
sechdrs[i].sh_addr =
(unsigned long)dest

```
1765      /* Module has been moved. */
1766      mod = (void *)sechdrs[modindex].sh_addr;
          指向.gnu.linkonce.this_module section。？？？
1767
1768      /* Now we've moved module, initialize linked lists, etc. */
1769      module_unload_init(mod);
          ？？？
1770
1771      /* Set up license info based on the info section */
1772      set_license(mod, get_modinfo(sechdrs, infoindex, "license"));
          在源代码上可以通过 MODULE_LICENSE("XXX")宏来设置 LKM 作者想在自己的作品中应用的 License。比如
          MODULE_LICENSE("GPL")就是应用 GPL License。而该信息会被编译进 LKM 的.modinfo section 中。让我们来读取我们例子 LKM
          中的.modinfo section 来看看。

[wzhou@localhost example]$ hexdump -C -s 0x1a0[1] -n 324[2] hello-5.ko
000001a0  70 61 72 6d 3d 6d 79 73  74 72 69 6e 67 3a 41 20  |parm=mystring:A |
000001b0  63 68 61 72 61 63 74 65  72 20 73 74 72 69 6e 67  |character string|
000001c0  00 70 61 72 6d 74 79 70  65 3d 6d 79 73 74 72 69  |.parmtype=mystri|
000001d0  6e 67 3a 63 68 61 72 70  00 70 61 72 6d 3d 6d 79  |ng:charp.parm=my|
000001e0  6c 6f 6e 67 3a 41 20 6c  6f 6e 67 20 69 6e 74 65  |long:A long inte|
000001f0  67 65 72 00 70 61 72 6d  74 79 70 65 3d 6d 79 6c  |ger.parmtype=myl|
00000200  6f 6e 67 3a 6c 6f 6e 67  00 70 61 72 6d 3d 6d 79  |ong:long.parm=my|
00000210  69 6e 74 3a 41 6e 20 69  6e 74 65 67 65 72 00 70  |int:An integer.p|
00000220  61 72 6d 74 79 70 65 3d  6d 79 69 6e 74 3a 69 6e  |armtype=myint:in|
```

[1] 在该例子 LKM 中.modinfo section 从文件偏移的 0x1a0 开始。
[2] 在该例子 LKM 中.modinfo section 所占大小为 324 bytes。

```
00000230  74 00 70 61 72 6d 3d 6d  79 73 68 6f 72 74 3a 41  |t.parm=myshort:A|
00000240  20 73 68 6f 72 74 20 69  6e 74 65 67 65 72 00 70  | short integer.p|
00000250  61 72 6d 74 79 70 65 3d  6d 79 73 68 6f 72 74 3a  |armtype=myshort:|
00000260  73 68 6f 72 74 00 61 75  74 68 6f 72 3d 50 65 74  |short.author=Pet|
00000270  65 72 20 4a 61 79 20 53  61 6c 7a 6d 61 6e 00 6c  |er Jay Salzman.l|
00000280  69 63 65 6e 73 65 3d 47  50 4c 00 00 00 00 00 00  |icense=GPL[3].....|
00000290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002a0  64 65 70 65 6e 64 73 3d  00 00 00 00 00 00 00 00  |depends=........|
000002b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002c0  76 65 72 6d 61 67 69 63  3d 32 2e 36 2e 32 30 20  |vermagic=2.6.20 |
000002d0  53 4d 50 20 6d 6f 64 5f  75 6e 6c 6f 61 64 20 33  |SMP mod_unload 3|
000002e0  38 36 20 00                                       |86 .|
```

从.modinfo section 中取出 license=”XXX”中的 XXX，然后比较该 LKM 用的 license 是否是如下的一种：

GPL

GPL v2

GPL and additional rights

Dual BSD/GPL

Dual MIT/GPL

Dual MPL/GPL

（以上 license 从 license_is_gpl_compatible（）函数提取。）

如果不是，则认为该 module 被“tainted”，即被污染了。GPLer 们把所有专有软件都看成“污染源”了。

```
1773
1774      if (strcmp(mod->name, "ndiswrapper") == 0)
1775         add_taint(TAINT_PROPRIETARY_MODULE);
```

[3] 这就是源代码中 MODULE_LICENSE("GPL")在最终 LKM 中的反映。

```
1776      if (strcmp(mod->name, "driverloader") == 0)
1777         add_taint_module(mod, TAINT_PROPRIETARY_MODULE);
          如果载入的是“ndiswrapper”或“driverloader”LKM 则设“tainted”标志。
          NdisWrapper 是一个开源的驱动(从技术上讲,是内核的一个模块，即 LKM),它能够让 Linux 使用标准的 Windows XP 下的无线网络驱
          动。你可以认为 NdisWrapper 是 Linux 内核和 Windows 驱动之间的一个翻译层。Windows 驱动可以通过 NdisWrapper 的配置工具进
          行安装。在 Linux 上运行 Windows 下的驱动，当然是被污染了。

          driverloader 完成的功能于 ndiswrapper 类似，但它是商业软件。
1778
1779      /* Set up MODINFO_ATTR fields */
1780      setup_modinfo(mod, sechdrs, infoindex);
          ？？？
1781
1782      /* Fix up syms, so that st_value is a pointer to location. */
1783      err = simplify_symbols(sechdrs, symindex, strtab, versindex, pcpuindex,
1784                             mod);
          解决在 LKM 中对 kernel 与其他 LKM 的函数调用问题。这有点像普通应用编程中你的程序中调用了共享库中的函数，比如 C 库中的
          strcmp()函数，在程序被载入内存执行以前要由可执行文件载入器（在 Linux 上是 ELF loader，在 Windows 上是 PE loader）
          “resolve”该悬空的调用。simplify_symbols（）函数就是根据纪录在.symtab section 中的对非本身模块的函数调用来 fix。
          也就是根据函数名称（name），来找到该函数所在的地址。在例子代码中我们调用了内核提供的输出函数 printk()。
static int __init hello_5_init(void)
{
    printk(KERN_ALERT "Hello, world 5\n=============\n");
    printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
    printk(KERN_ALERT "myint is an integer: %d\n", myint);
    printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
```

```
    printk(KERN_ALERT "mystring is a string: %s\n", mystring);
    return 0;
}
```

我们看一下对应的 obj 代码（这还真不能称其为可执行代码）。

```
00000000 <init_module>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
   d:   e8 fc ff ff ff          call   e <init_module+0xe>[4]
  12:   0f bf 05 0c 00 00 00    movswl 0xc,%eax
  19:   c7 04 24 24 00 00 00    movl   $0x24,(%esp)
  20:   89 44 24 04             mov    %eax,0x4(%esp)
  24:   e8 fc ff ff ff          call   25 <init_module+0x25>[5]
  29:   a1 08 00 00 00          mov    0x8,%eax
  2e:   c7 04 24 15 00 00 00    movl   $0x15,(%esp)
  35:   89 44 24 04             mov    %eax,0x4(%esp)
  39:   e8 fc ff ff ff          call   3a <init_module+0x3a>[6]
  3e:   a1 04 00 00 00          mov    0x4,%eax
  43:   c7 04 24 48 00 00 00    movl   $0x48,(%esp)
  4a:   89 44 24 04             mov    %eax,0x4(%esp)
  4e:   e8 fc ff ff ff          call   4f <init_module+0x4f>[7]
```

[4] 对应源代码中第一个 printk()调用
[5] 对应源代码中第二个 printk()调用
[6] 对应源代码中第三个 printk()调用
[7] 对应源代码中第四个 printk()调用

```
 53:   a1 00 00 00 00         mov    0x0,%eax
 58:   c7 04 24 31 00 00 00    movl   $0x31,(%esp)
 5f:   89 44 24 04           mov    %eax,0x4(%esp)
 63:   e8 fc ff ff ff         call   64 <init_module+0x64>[8]
 68:   31 c0                 xor    %eax,%eax
 6a:   c9                    leave
 6b:   c3                    ret
```

“e8” 是 call 指令，而后面的“fc ff ff ff”肯定不是 printk()函数所在的地址。因为那都快到达 32 位地址空间的顶端（ff ff ff ff）了。在 LKM 中这些对 printk 的调用都是悬空的，用 ELF Specification 中的说法就是"UND"的。而在.rel.init.text section 中则有对此的解答（因为 init_module 函数位于.init.text section，所以其指示未定函数调用的信息在.rel.init.text section 中，即对应的 section 名称前加.rel 前缀即可）。

```
Relocation section '.rel.init.text' at offset 0xf950 contains 14 entries:
 Offset     Info    Type           Sym.Value  Sym. Name
00000009  00000501 R_386_32          00000000   .rodata.str1.4
0000000e  00003702 R_386_PC32        00000000   printk
00000015  00000801 R_386_32          00000000   .data
0000001c  00000501 R_386_32          00000000   .rodata.str1.4
00000025  00003702 R_386_PC32        00000000   printk
0000002a  00000801 R_386_32          00000000   .data
00000031  00000401 R_386_32          00000000   .rodata.str1.1
0000003a  00003702 R_386_PC32        00000000   printk
0000003f  00000801 R_386_32          00000000   .data
00000046  00000501 R_386_32          00000000   .rodata.str1.4
```

[8] 对应源代码中第五个 printk()调用

```
0000004f  00003702 R_386_PC32       00000000   printk
00000054  00000801 R_386_32         00000000   .data
0000005b  00000401 R_386_32         00000000   .rodata.str1.1
00000064  00003702 R_386_PC32       00000000   printk
```

以第一个 printk()为例。

```
Offset     Info    Type          Sym.Value  Sym. Name
0000000e  00003702 R_386_PC32       00000000   printk
```

offset 0000000e 表示该重定位项是针对.init.text section 中的偏移 0x0e 的。

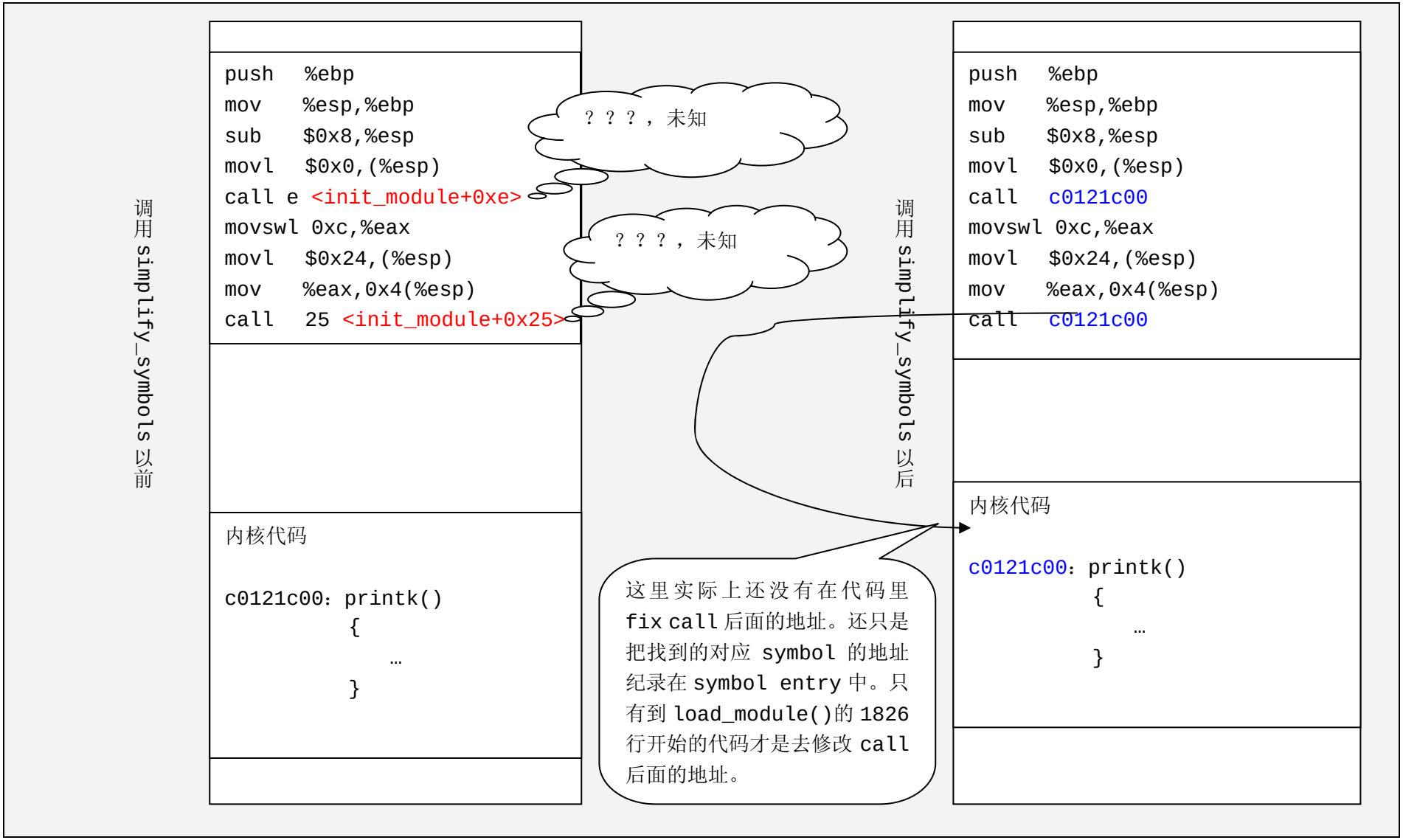
```
00000000 <init_module>:
   0:   55                    push   %ebp
   1:   89 e5                 mov    %esp,%ebp
   3:   83 ec 08              sub    $0x8,%esp
   6:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
   d:   e8 ▲fc ff ff ff         call   e <init_module+0xe>
  12:   0f bf 05 0c 00 00 00    movswl 0xc,%eax
  19:   c7 04 24 24 00 00 00    movl   $0x24,(%esp)
```

上面▲所在的位置即离开.text section 的头部 0x0e 个字节。

Info 00003702 是类型信息，这里表示是函数调用，而非象全局变量等，具体可参考 ELF Specification。
Sym.Value 是 00000000，未定吗，当然是 0
Sym. Name 指向字符串“printk”。表示.text section 偏移 0x0e 出的函数名称。

当执行完 simplify_symbols（）以后，原本“UND”的 printk 函数调用应该不再“UND”。

具体解释见对该函数的注释。


调用 simplify_symbols 以前
push %ebp
mov %esp,%ebp
sub $0x8,%esp
movl $0x0,(%esp)
call e <init_module+0xe>
movswl 0xc,%eax
movl $0x24,(%esp)
mov %eax,0x4(%esp)
call 25 <init_module+0x25>
内核代码
c0121c00: printk()
{
…
}
？？？，未知
？？？，未知
这里实际上还没有在代码里fix call后面的地址。还只是把找到的对应 symbol 的地址纪录在 symbol entry 中。只有到 load_module()的 1826 行开始的代码才是去修改 call 后面的地址。
调用 simplify_symbols 以后
push %ebp
mov %esp,%ebp
sub $0x8,%esp
movl $0x0,(%esp)
call c0121c00
movswl 0xc,%eax
movl $0x24,(%esp)
mov %eax,0x4(%esp)
call c0121c00
内核代码
c0121c00: printk()
{
…
}

```
1785      if (err < 0)
1786         goto cleanup;
1787
1788      /* Set up EXPORTed & EXPORT_GPLed symbols (section 0 is 0 length) */
1789      mod->num_syms = sechdrs[exportindex].sh_size / sizeof(*mod->syms);
1790      mod->syms = (void *)sechdrs[exportindex].sh_addr;
```

__ksymtab section 中包含了本 LKM 中要输出(export)的 symbol，在例子 LKM 中没有该 section。如果你需要输出函数或变量给其他 LKM 使用，则需要用到 EXPORT_SYMBOL 宏。比如在例子代码中加入如下代码：

```
EXPORT_SYMBOL(myshort);           //输出 myshort 这个全局变量
static int export_sample()
{
   return 1;
}
EXPORT_SYMBOL(export_sample);     //输出 export_sample 这个函数
```

编译以后，再看一下例子 LKM 的 section table。

```
[wzhou@localhost example]$ readelf -S hello-5.ko
There are 38 section headers, starting at offset 0xf604:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000040 00000c 00  AX  0   0 16
  [ 2] .exit.text        PROGBITS        00000000 000050 000014 00  AX  0   0 16
```

```
  [ 3] .rel.exit.text    REL             00000000 00fbf4 000010 08     36   2  4
  [ 4] .init.text        PROGBITS        00000000 000070 00006c 00  AX  0   0 16
  [ 5] .rel.init.text    REL             00000000 00fc04 000070 08     36   4  4
  [ 6] .rodata.str1.1    PROGBITS        00000000 0000dc 000053 01 AMS  0   0  1
  [ 7] .rodata.str1.4    PROGBITS        00000000 000130 00006a 01 AMS  0   0  4
  [ 8] __ksymtab         PROGBITS        00000000 00019c 000010 00   A  0   0  4
  [ 9] .rel__ksymtab     REL             00000000 00fc74 000020 08     36   8  4
  [10] .modinfo          PROGBITS        00000000 0001c0 000144 00   A  0   0 32
  [11] __param           PROGBITS        00000000 000304 000050 00   A  0   0  4
  [12] .rel__param       REL             00000000 00fc94 000080 08     36  11  4
  [13] __ksymtab_strings PROGBITS        00000000 000354 000016 00   A  0   0  1
  [14] .data             PROGBITS        00000000 00036c 00002c 00  WA  0   0  4
  [15] .rel.data         REL             00000000 00fd14 000008 08     36  14  4
  [16] .gnu.linkonce.thi PROGBITS        00000000 000400 001200 00  WA  0   0 128
  [17] .rel.gnu.linkonce REL             00000000 00fd1c 000010 08     36  16  4
  [18] .bss              NOBITS          00000000 001600 000000 00  WA  0   0  4
  [19] .comment          PROGBITS        00000000 001600 00005c 00      0   0  1
  [20] .debug_aranges    PROGBITS        00000000 00165c 000030 00      0   0  1
  [21] .rel.debug_arange REL             00000000 00fd2c 000020 08     36  20  4
  [22] .debug_pubnames   PROGBITS        00000000 00168c 00006a 00      0   0  1
  [23] .rel.debug_pubnam REL             00000000 00fd4c 000010 08     36  22  4
  [24] .debug_info       PROGBITS        00000000 0016f6 00871b 00      0   0  1
  [25] .rel.debug_info   REL             00000000 00fd5c 003d18 08     36  24  4
  [26] .debug_abbrev     PROGBITS        00000000 009e11 000511 00      0   0  1
  [27] .debug_line       PROGBITS        00000000 00a322 000801 00      0   0  1
  [28] .rel.debug_line   REL             00000000 013a74 000018 08     36  27  4
```

```
  [29] .debug_frame     PROGBITS        00000000 00ab24 00005c 00      0   0  4
  [30] .rel.debug_frame  REL            00000000 013a8c 000030 08     36  29  4
  [31] .debug_str       PROGBITS        00000000 00ab80 0048ae 01  MS  0   0  1
  [32] .debug_loc       PROGBITS        00000000 00f42e 000084 00      0   0  1
  [33] .rel.debug_loc    REL            00000000 013abc 000090 08     36  32  4
  [34] .note.GNU-stack   PROGBITS        00000000 00f4b2 000000 00      0   0  1
  [35] .shstrtab        STRTAB          00000000 00f4b2 000152 00      0   0  1
  [36] .symtab          SYMTAB          00000000 013b4c 000440 10     37  56  4
  [37] .strtab          STRTAB          00000000 013f8c 0002b5 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

这就是 EXPORT_SYMBOL 宏的作用。

而 EXPORT_SYMBOL_GPL(XXX)中输出的 symbol 只能由标志为“GPL”的 LKM 才能输入。Linux 内核鼓励你开发的 LKM 都能用“GPL”来 release。

```
1791      if (crcindex)
1792          mod->crcs = (void *)sechdrs[crcindex].sh_addr;
```

使 mod->crcs 指向__kcrctab section。

```
1793      mod->num_gpl_syms = sechdrs[gplindex].sh_size / sizeof(*mod->gpl_syms);
1794      mod->gpl_syms = (void *)sechdrs[gplindex].sh_addr;
```

除了 EXPORT_SYMBOL 宏可以输出 symbol 外，用 EXPORT_SYMBOL_GPL 宏也可以输出。

```
1795      if (gplcrcindex)
```

```
1796          mod->gpl_crcs = (void *)sechdrs[gplcrcindex].sh_addr;
1797      mod->num_gpl_future_syms = sechdrs[gplfutureindex].sh_size /
1798                    sizeof(*mod->gpl_future_syms);
1799      mod->num_unused_syms = sechdrs[unusedindex].sh_size /
1800                    sizeof(*mod->unused_syms);
1801      mod->num_unused_gpl_syms = sechdrs[unusedgplindex].sh_size /
1802                    sizeof(*mod->unused_gpl_syms);
1803      mod->gpl_future_syms = (void *)sechdrs[gplfutureindex].sh_addr;
1804      if (gplfuturecrcindex)
1805          mod->gpl_future_crcs = (void *)sechdrs[gplfuturecrcindex].sh_addr;
1806
1807      mod->unused_syms = (void *)sechdrs[unusedindex].sh_addr;
1808      if (unusedcrcindex)
1809          mod->unused_crcs = (void *)sechdrs[unusedcrcindex].sh_addr;
1810      mod->unused_gpl_syms = (void *)sechdrs[unusedgplindex].sh_addr;
1811      if (unusedgplcrcindex)
1812          mod->unused_crcs = (void *)sechdrs[unusedgplcrcindex].sh_addr;
1813
1814  #ifdef CONFIG_MODVERSIONS
```

如果在编译内核时打开该选项，则在所有的 symbol 后面会带上由当前版本生成的数字。

```
1815      if ((mod->num_syms && !crcindex) ||
1816          (mod->num_gpl_syms && !gplcrcindex) ||
1817          (mod->num_gpl_future_syms && !gplfuturecrcindex) ||
1818          (mod->num_unused_syms && !unusedcrcindex) ||
1819          (mod->num_unused_gpl_syms && !unusedgplcrcindex)) {
1820          printk(KERN_WARNING "%s: No versions for exported symbols."
```

```
1821                " Tainting kernel.\n", mod->name);
1822          add_taint_module(mod, TAINT_FORCED_MODULE);
1823       }
1824    #endif
1825
          刚才在 simplify_symbols（）中是找到了所有未定义的 symbol 对应的地址，并记录在该 symbol entry 中。现在要真正去修改 call
          调用后面的真正的地址了。
1826       /* Now do relocations. */
1827       for (i = 1; i < hdr->e_shnum; i++) {
          对 LKM 中所有 section 进行枚举。
1828          const char *strtab = (char *)sechdrs[strindex].sh_addr;
              strindex 的赋值在本函数 load_module()中的第 1614 行

              1614            strindex = sechdrs[i].sh_link;

              即.symtab section 中的 symbol name 的字符串所在的 string section。
              strtab 指向该 section，在具体 fix relocation 时当然要用到 symbol name。

              下面是合法性检查。
1829          unsigned int info = sechdrs[i].sh_info;
1830
1831          /* Not a valid relocation section? */
1832          if (info >= hdr->e_shnum)
1833             continue;
1834
1835          /* Don't bother with non-allocated sections */
```

```
1836          if (!(sechdrs[info].sh_flags & SHF_ALLOC))
1837             continue;
              对不占用内存的 section 当然 可以忽略。
1838
1839          if (sechdrs[i].sh_type == SHT_REL)
              i386 CPU 符合这条分支，即调用 apply_relocate（）。
1840             err = apply_relocate(sechdrs, strtab, symindex, i,mod);
                 在该函数就是实实在在的去修改 call 后面的那四个 byte 了。具体见该函数的注释。
1841          else if (sechdrs[i].sh_type == SHT_RELA)
              好像 SUN SPARC CPU 要走这条分支，不过我不敢很肯定。
1842             err = apply_relocate_add(sechdrs, strtab, symindex, i,
1843                        mod);
1844          if (err < 0)
1845             goto cleanup;
1846       }
1847
          到此代码中对内核或其他 LKM 的函数调用和数据引用已经解决，即依赖于外部的问题解决了。就像自私固然是人的天性，但毕竟不能太自
          私了，你用了人家的东西，自己总得也有所贡献吧。下面就是本 LKM 要把自身可以输出(export)的函数或数据的 symbol 注册到系统中，
          以便其他 LKM 也能“利用”你。毕竟只有“One for All”，才能“All for One”吗!

1848          /* Find duplicate symbols */
1849       err = verify_export_symbols(mod);
          这里先要检查一下，本 LKM 可以贡献出的 symbol 是否会与系统中已有的 symbol 冲突，也就是重名。具体见对该函数的注释。
1850
1851       if (err < 0)
1852          goto cleanup;
```

```
1853
1854      /* Set up and sort exception table */
1855      mod->num_exentries = sechdrs[exindex].sh_size / sizeof(*mod->extable);
1856      mod->extable = extable = (void *)sechdrs[exindex].sh_addr;
1857      sort_extable(extable, extable + mod->num_exentries);
          在 LKM 中可以定义 exception table，具体含义请看我的关于 Linux 内核中 exception table 的详细分析文章《Linux 与 SEH》。
1858
1859      /* Finally, copy percpu area over. */
1860      percpu_modcopy(mod->percpu, (void *)sechdrs[pcpuindex].sh_addr,
1861                sechdrs[pcpuindex].sh_size);
1862
          把要输出的 symbol 纪录在 mod 指向的 struct module 结构中。这样以便其他 LKM 能利用。
1863      add_kallsyms(mod, sechdrs, symindex, strindex, secstrings);
1864
1865      err = module_finalize(hdr, sechdrs, mod);
1866      if (err < 0)
1867         goto cleanup;
1868
1869      /* flush the icache in correct context */
1870      old_fs = get_fs();
1871      set_fs(KERNEL_DS);
          这里与 1885 行的代码构成一对。这里
          #define get_fs() (current_thread_info()->addr_limit)
          是当前进程的用户空间的边界，即 3G。而 KERNEL_DS 的定义如下
          #define KERNEL_DS   MAKE_MM_SEG(0xFFFFFFFFUL)
          这里先临时把当前 process 的空间放大到 4G 的整个地址空间，在 1885 行的 set_fs(old_fs)中又马上恢复。这里的当前 process 就
```

是 insmod utility。这显然是为 flush_icache_range（）来准备的。

```
1872
1873      /*
1874       * Flush the instruction cache, since we've played with text.
1875       * Do it before processing of module parameters, so the module
1876       * can provide parameter accessor functions of its own.
1877       */
1878      if (mod->module_init)
1879         flush_icache_range((unsigned long)mod->module_init,
1880                    (unsigned long)mod->module_init
1881                    + mod->init_size);
1882      flush_icache_range((unsigned long)mod->module_core,
1883                (unsigned long)mod->module_core + mod->core_size);
```

在 i386 下 flush_icache_range（）是空的宏定义，可能在其他体系结构 CPU 上要做点什么事吧。

```
1884
1885      set_fs(old_fs);
1886
1887      mod->args = args;
```

传递给 insmod utility 的参数。比如例子代码中：

```
insmod hello-5.o mystring="bebop" myshort=255 myint=-1
```

这里的 mystring="bebop" myshort=255 myint=-1 即是参数列表。这些参数信息被放在__param section 中。
这些参数的作用是设置 LKM 中的初始值。在例子的源代码文件 hello-5.c 中有如下的代码：

```
        static short int myshort = 1;
        static int myint = 420;
        static long int mylong = 9999;
        static char *mystring = "blah";
```

即如果你是这样载入本 LKM，insmod hello-5.o，则在载入后这些全局变量的初始值即是上面在源代码中静态赋的值。但 Linux 内核还给你在动态载入该 LKM 时去动态初始化全局变量。像例子中的 mystring="bebop" myshort=255 myint=-1，即表示在 LKM 开 始执行时，mystring 被覆写成指向“bebop”，myShort 为 255，myint 为-1。具体怎么应用请看相应的 Linux 下 Device Driver 开发的资料。

```
1888      if (obsparmindex)
1889         printk(KERN_WARNING "%s: Ignoring obsolete parameters\n",
1890                mod->name);
1891
1892      /* Size of section 0 is 0, so this works well if no params */
1893      err = parse_args(mod->name, mod->args,
1894             (struct kernel_param *)
1895             sechdrs[setupindex].sh_addr,
1896             sechdrs[setupindex].sh_size
1897             / sizeof(struct kernel_param),
1898             NULL);
```

分析并处理传入的参数。见对该函数的分析。

```
1899      if (err < 0)
1900         goto arch_cleanup;
```

```
1901
1902      err = mod_sysfs_setup(mod,
1903                  (struct kernel_param *)
1904                  sechdrs[setupindex].sh_addr,
1905                  sechdrs[setupindex].sh_size
1906                  / sizeof(struct kernel_param));
1907      if (err < 0)
1908         goto arch_cleanup;
1909      add_sect_attrs(mod, hdr->e_shnum, secstrings, sechdrs);
1910
1911      /* Size of section 0 is 0, so this works well if no unwind info. */
1912      mod->unwind_info = unwind_add_table(mod,
1913                                    (void *)sechdrs[unwindex].sh_addr,
1914                                    sechdrs[unwindex].sh_size);
1915
1916      /* Get rid of temporary copy */
1917      vfree(hdr);
          释放整个 LKM 在 kerenl 空间的缓存。

          下面是出错时，需要释放的一些资源。
1918
1919      /* Done! */
1920      return mod;
1921
1922    arch_cleanup:
1923      module_arch_cleanup(mod);
```

```
1924   cleanup:
1925     module_unload_free(mod);
1926     module_free(mod, mod->module_init);
1927   free_core:
1928     module_free(mod, mod->module_core);
1929   free_percpu:
1930     if (percpu)
1931        percpu_modfree(percpu);
1932   free_mod:
1933     kfree(args);
1934   free_hdr:
1935     vfree(hdr);
1936     return ERR_PTR(err);
1937
1938   truncated:
1939     printk(KERN_ERR "Module len %lu truncated\n", len);
1940     err = -ENOEXEC;
1941     goto free_hdr;
1942  }
```

src\linux-2.6.20\kernel\module.c

```
1331  /* Lay out the SHF_ALLOC sections in a way not dissimilar to how ld
1332     might -- code, read-only data, read-write data, small data.  Tally
1333     sizes, and place the offsets into sh_entsize fields: high bit means it
1334     belongs in init. */
```

```
1335  static void layout_sections(struct module *mod,
1336                const Elf_Ehdr *hdr,
1337                Elf_Shdr *sechdrs,
1338                const char *secstrings)
1339  {
1340     static unsigned long const masks[][2] = {
1341        /* NOTE: all executable code must be the first section
1342         * in this array; otherwise modify the text_size
1343         * finder in the two loops below */
1344        { SHF_EXECINSTR | SHF_ALLOC, ARCH_SHF_SMALL },
1345        { SHF_ALLOC, SHF_WRITE | ARCH_SHF_SMALL },
1346        { SHF_WRITE | SHF_ALLOC, ARCH_SHF_SMALL },
1347        { ARCH_SHF_SMALL | SHF_ALLOC, 0 }
1348     };
```

在 elf.h 中，上面的常数定义如下：

```
351 #define SHF_WRITE       (1 << 0)  /* Writable */
352 #define SHF_ALLOC       (1 << 1)  /* Occupies memory during execution */
353 #define SHF_EXECINSTR       (1 << 2)  /* Executable */
```

而 ARCH_SHF_SMALL 的定义如下：

```
#ifndef ARCH_SHF_SMALL
   #define ARCH_SHF_SMALL 0
#endif
```

这里用该二维数组中的第一维（masks[0]）来筛选 LKM 中的某个 section 是否要 load 入内存，而第二维（masks[1]）中的顺序来表示 load 入内存的先后次序。从该数组的定义可以看出，具有如下属性的 section 需要载入内存。

1。设置了 SHF_ALLOC 标志的 section。

而载入内存的次序（反映在地址空间上就是次序越前面，载入的地址越低）则是
1。可执行代码 section
2。只读数据 section
3。可写数据 section
4。其他任何要分配空间的 section

下面的代码分成两部分
1。第一个循环用来处理普通的 section(非初始化 section)，即在该 LKM 被卸载出去以前，一直占有内存。
2。第一个循环用来处理初始化 section，即该 LKM 初始化完成以后，就可以释放所占内存。

```
1349      unsigned int m, i;
1350
1351      for (i = 0; i < hdr->e_shnum; i++)
1352         sechdrs[i].sh_entsize = ~0UL;
```
先初始化所有的 section 的 sh_entsize 为-1。
```
1353
1354      DEBUGP("Core section allocation order:\n");
1355      for (m = 0; m < ARRAY_SIZE(masks); ++m) {
1356         for (i = 0; i < hdr->e_shnum; ++i) {
1357            Elf_Shdr *s = &sechdrs[i];
1358
1359            if ((s->sh_flags & masks[m][0]) != masks[m][0]
1360               || (s->sh_flags & masks[m][1])
1361               || s->sh_entsize != ~0UL
1362               || strncmp(secstrings + s->sh_name,
1363                     ".init", 5) == 0)
```

```
1364                continue;
                满足上面的条件的则忽略该 section，即不需要载入内存。
                上面的代码用
                s->sh_flags & masks[m][0]) != masks[m][0]
                (s->sh_flags & masks[m][1])
                来实现载入的优先级，这个思路挺妙的。
1365            s->sh_entsize = get_offset(&mod->core_size, s);
                在 s->sh_entsize 中记录当前处理的 section 的载入内存后占用内存的大小，而 mod->core_size 中累加需要载入 section
                的大小总和。
1366            DEBUGP("\t%s\n", secstrings + s->sh_name);
1367        }
1368        if (m == 0)
1369            mod->core_text_size = mod->core_size;
            如果只循环了一次，表示该 LKM 只有代码 section，所以 core_text_size 即整个 LKM 载入大小。
1370    }
1371
        同上面几乎完全一样，只是对初始化 section 的处理。
1372    DEBUGP("Init section allocation order:\n");
1373    for (m = 0; m < ARRAY_SIZE(masks); ++m) {
1374        for (i = 0; i < hdr->e_shnum; ++i) {
1375            Elf_Shdr *s = &sechdrs[i];
1376
1377            if ((s->sh_flags & masks[m][0]) != masks[m][0]
1378                || (s->sh_flags & masks[m][1])
1379                || s->sh_entsize != ~0UL
1380                || strncmp(secstrings + s->sh_name,
```

```
1381                      ".init", 5) != 0)
1382                continue;
1383             s->sh_entsize = (get_offset(&mod->init_size, s)
1384                    | INIT_OFFSET_MASK);
1385             DEBUGP("\t%s\n", secstrings + s->sh_name);
1386          }
1387          if (m == 0)
1388             mod->init_text_size = mod->init_size;
1389       }
1390    }
```

```
1321   /* Update size with this section: return offset. */
1322   static long get_offset(unsigned long *size, Elf_Shdr *sechdr)
1323   {
1324       long ret;
1325
1326       ret = ALIGN(*size, sechdr->sh_addralign ?: 1);
       如果该 section 对载入地址有对齐(alignment)的要求，则考虑对齐因素。
1327       *size = ret + sechdr->sh_size;
       返回值为当前处理的 section 的载入内存所占的大小（考虑了对齐因素），而*size 是已经处理过的 section 的大小总和。
1328       return ret;
1329   }
```

src\linux-2.6.20\kernel\module.c

下面例子 LKM 中的 symbol。

```
[wzhou@localhost example]$ readelf -s hello-5.ko

Symbol table '.symtab' contains 61 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 SECTION LOCAL  DEFAULT    6
     5: 00000000     0 SECTION LOCAL  DEFAULT    7
     6: 00000000     0 SECTION LOCAL  DEFAULT    8
     7: 00000000     0 SECTION LOCAL  DEFAULT    9
     8: 00000000     0 SECTION LOCAL  DEFAULT   11
     9: 00000000     0 SECTION LOCAL  DEFAULT   13
    10: 00000000     0 SECTION LOCAL  DEFAULT   15
    11: 00000000     0 SECTION LOCAL  DEFAULT   16
    12: 00000000     0 SECTION LOCAL  DEFAULT   17
    13: 00000000     0 SECTION LOCAL  DEFAULT   19
    14: 00000000     0 SECTION LOCAL  DEFAULT   21
    15: 00000000     0 SECTION LOCAL  DEFAULT   23
    16: 00000000     0 SECTION LOCAL  DEFAULT   24
    17: 00000000     0 SECTION LOCAL  DEFAULT   26
    18: 00000000     0 SECTION LOCAL  DEFAULT   28
    19: 00000000     0 SECTION LOCAL  DEFAULT   29
```

```
    20: 00000000     0 SECTION LOCAL  DEFAULT   31
    21: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello-5.c
    22: 00000000    20 FUNC    LOCAL  DEFAULT    2 hello_5_exit
    23: 00000000   108 FUNC    LOCAL  DEFAULT    4 hello_5_init
    24: 0000000c     2 OBJECT  LOCAL  DEFAULT   11 myshort
    25: 00000008     4 OBJECT  LOCAL  DEFAULT   11 myint
    26: 00000004     4 OBJECT  LOCAL  DEFAULT   11 mylong
    27: 00000000     4 OBJECT  LOCAL  DEFAULT   11 mystring
    28: 00000000    33 OBJECT  LOCAL  DEFAULT    8 __mod_mystring33
    29: 00000021    24 OBJECT  LOCAL  DEFAULT    8 __mod_mystringtype32
    30: 00000000    20 OBJECT  LOCAL  DEFAULT    9 __param_mystring
    31: 0000000e     9 OBJECT  LOCAL  DEFAULT   11 __param_str_mystring
    32: 00000039    27 OBJECT  LOCAL  DEFAULT    8 __mod_mylong31
    33: 00000054    21 OBJECT  LOCAL  DEFAULT    8 __mod_mylongtype30
    34: 00000014    20 OBJECT  LOCAL  DEFAULT    9 __param_mylong
    35: 00000017     7 OBJECT  LOCAL  DEFAULT   11 __param_str_mylong
    36: 00000069    22 OBJECT  LOCAL  DEFAULT    8 __mod_myint29
    37: 0000007f    19 OBJECT  LOCAL  DEFAULT    8 __mod_myinttype28
    38: 00000028    20 OBJECT  LOCAL  DEFAULT    9 __param_myint
    39: 0000001e     6 OBJECT  LOCAL  DEFAULT   11 __param_str_myint
    40: 00000092    29 OBJECT  LOCAL  DEFAULT    8 __mod_myshort27
    41: 000000af    23 OBJECT  LOCAL  DEFAULT    8 __mod_myshorttype26
    42: 0000003c    20 OBJECT  LOCAL  DEFAULT    9 __param_myshort
    43: 00000024     8 OBJECT  LOCAL  DEFAULT   11 __param_str_myshort
    44: 000000c6    25 OBJECT  LOCAL  DEFAULT    8 __mod_author11
    45: 000000df    12 OBJECT  LOCAL  DEFAULT    8 __mod_license10
```

```
    46: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello-5.mod.c
    47: 00000100     9 OBJECT  LOCAL  DEFAULT    8 __module_depends
    48: 00000120    36 OBJECT  LOCAL  DEFAULT    8 __mod_vermagic5
    49: 00000000  4608 OBJECT  GLOBAL DEFAULT   13 __this_module
    50: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_set_short
    51: 00000000    20 FUNC    GLOBAL DEFAULT    2 cleanup_module
    52: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_set_charp
    53: 00000000   108 FUNC    GLOBAL DEFAULT    4 init_module
    54: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_get_long
    55: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printk
    56: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_get_charp
    57: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_set_int
    58: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_get_short
    59: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_set_long
    60: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND param_get_int
```

这里的symbol包括了“输入”与“输出”的symbol。所谓输入，即调用内核或其他LKM的函数的symbol，而所谓输出，则是本LKM提供的symbol，可以被其他LKM作为输入。上面标红的“UND”[9]的即是输入的symbol，需要被"Fix"的。

```
1260   /* Change all symbols so that sh_value encodes the pointer directly. */
1261   static int simplify_symbols(Elf_Shdr *sechdrs,
1262                               unsigned int symindex,
1263                               const char *strtab,
1264                               unsigned int versindex,
1265                               unsigned int pcpuindex,
```

[9] "UND"表示 Undefined，未定义。

```
1266                struct module *mod)
1267  {
1268      Elf_Sym *sym = (void *)sechdrs[symindex].sh_addr;
          取得.symtab section 在内存中的首地址。
1269      unsigned long secbase;
1270      unsigned int i, n = sechdrs[symindex].sh_size / sizeof(Elf_Sym);
          每一个 symbol entry 由 structure Elf_Sym 表示，所有的 symbol entry 组成 symbol 的数组，其项数为 n。
1271      int ret = 0;
1272
1273      for (i = 1; i < n; i++) {
          对 symbol entry 数组进行枚举。
1274          switch (sym[i].st_shndx) {
              sym[i].st_shndx 纪录了该 symbol entry 的类型
1275          case SHN_COMMON:
              在 LKM 的重定位 ELF 文件中不应该有如此类型的 symbol entry。
1276              /* We compiled with -fno-common.  These are not
1277                 supposed to happen.  */
1278              DEBUGP("Common symbol: %s\n", strtab + sym[i].st_name);
1279              printk("%s: please compile with -fno-common\n",
1280                     mod->name);
1281              ret = -ENOEXEC;
1282              break;
1283
1284          case SHN_ABS:
1285              /* Don't need to do anything */
1286              DEBUGP("Absolute symbol: 0x%08lx\n",
```

```
1287                  (long)sym[i].st_value);
1288              break;
1289
1290          case SHN_UNDEF:
              这里只需要处理“Undefined”类型的symbol。
1291              sym[i].st_value
1292                = resolve_symbol(sechdrs, versindex,
1293                        strtab + sym[i].st_name, mod);
              根据symbol name来得到该name所对应的地址。见resolve_symbol（）的注释。
1294
1295              /* Ok if resolved.  */
1296              if (sym[i].st_value != 0)
1297                  break;
1298              /* Ok if weak.  */
1299              if (ELF_ST_BIND(sym[i].st_info) == STB_WEAK)
1300                  break;
                  ？？？
1301
1302              printk(KERN_WARNING "%s: Unknown symbol %s\n",
1303                    mod->name, strtab + sym[i].st_name);
                  在要载入的LKM中有函数调用没办法找到实际的调用体，载入失败。
1304              ret = -ENOENT;
1305              break;
1306
1307          default:
1308              /* Divert to percpu allocation if a percpu var. */
```

```
1309            if (sym[i].st_shndx == pcpuindex)
1310                secbase = (unsigned long)mod->percpu;
1311            else
1312                secbase = sechdrs[sym[i].st_shndx].sh_addr;
1313            sym[i].st_value += secbase;
1314            break;
1315        }
1316    }
1317
1318    return ret;
1319  }
```

src\linux-2.6.20\kernel\module.c

```
944 /* Resolve a symbol for this module.  I.e. if we find one, record usage.
945   Must be holding module_mutex. */
946 static unsigned long resolve_symbol(Elf_Shdr *sechdrs,
947                 unsigned int versindex,
948                 const char *name,
949                 struct module *mod)
950 {
951    struct module *owner;
952    unsigned long ret;
953    const unsigned long *crc;
954
```

该函数只是对__find_symbol（）的简单包装。

```
       参数解释如下：
       name   --- 要查找的输入函数或变量名，像 printk。
       owner  --- 若 resolve 该 symbol，则带回该 symbol 所在的模块
       crc    --- CRC 校验值，？？？
       再后一个是标志值，表示该 LKM 是否被污染。

955    ret = __find_symbol(name, &owner, &crc,
956           !(mod->taints & TAINT_PROPRIETARY_MODULE));
957    if (ret) {
958       /* use_module can fail due to OOM, or module unloading */
959       if (!check_version(sechdrs, versindex, name, mod, crc) ||
960          !use_module(mod, owner))
961          ret = 0;
962    }
963    return ret;
964 }
```

src\linux-2.6.20\kernel\module.c

```
171 /* Find a symbol, return value, crc and module which owns it */
    该函数在系统中查找由参数 name 标示的 symbol，然后返回与该 symbol 相关的信息：该 symbol 对应的地址，其所属的模块，该 symbol 的 CRC 校验值。
172 static unsigned long __find_symbol(const char *name,
173                 struct module **owner,
174                 const unsigned long **crc,
175                 int gplok)
```

```
176 {
177    struct module *mod;
178    const struct kernel_symbol *ks;
179
       首先在内核本身输出的 symbol table 中进行查找。
180    /* Core kernel first. */
181    *owner = NULL;
182    ks = lookup_symbol(name, __start___ksymtab, __stop___ksymtab);
183    if (ks) {
184       *crc = symversion(__start___kcrctab, (ks - __start___ksymtab));
185       return ks->value;
186    }
187    if (gplok) {
188       ks = lookup_symbol(name, __start___ksymtab_gpl,
189                  __stop___ksymtab_gpl);
190       if (ks) {
191          *crc = symversion(__start___kcrctab_gpl,
192                   (ks - __start___ksymtab_gpl));
193          return ks->value;
194       }
195    }
196    ks = lookup_symbol(name, __start___ksymtab_gpl_future,
197               __stop___ksymtab_gpl_future);
198    if (ks) {
199       if (!gplok) {
200          printk(KERN_WARNING "Symbol %s is being used "
```

```
201                "by a non-GPL module, which will not "
202                "be allowed in the future\n", name);
203           printk(KERN_WARNING "Please see the file "
204                "Documentation/feature-removal-schedule.txt "
205                "in the kernel source tree for more "
206                "details.\n");
207        }
208        *crc = symversion(__start___kcrctab_gpl_future,
209                (ks - __start___ksymtab_gpl_future));
210        return ks->value;
211     }
212
213     ks = lookup_symbol(name, __start___ksymtab_unused,
214                __stop___ksymtab_unused);
215     if (ks) {
216        printk_unused_warning(name);
217        *crc = symversion(__start___kcrctab_unused,
218                (ks - __start___ksymtab_unused));
219        return ks->value;
220     }
221
222     if (gplok)
223        ks = lookup_symbol(name, __start___ksymtab_unused_gpl,
224                __stop___ksymtab_unused_gpl);
225     if (ks) {
226        printk_unused_warning(name);
```

```
227         *crc = symversion(__start___kcrctab_unused_gpl,
228                 (ks - __start___ksymtab_unused_gpl));
229         return ks->value;
230     }
231
        如果在内核中找不到，则一次枚举系统中载入的 LKM 输出的 symbol 中查找。
232     /* Now try modules. */
233     list_for_each_entry(mod, &modules, list) {
234         *owner = mod;
235         ks = lookup_symbol(name, mod->syms, mod->syms + mod->num_syms);
236         if (ks) {
237             *crc = symversion(mod->crcs, (ks - mod->syms));
238             return ks->value;
239         }
240
241         if (gplok) {
242             ks = lookup_symbol(name, mod->gpl_syms,
243                     mod->gpl_syms + mod->num_gpl_syms);
244             if (ks) {
245                 *crc = symversion(mod->gpl_crcs,
246                         (ks - mod->gpl_syms));
247                 return ks->value;
248             }
249         }
250         ks = lookup_symbol(name, mod->unused_syms, mod->unused_syms + mod->num_unused_syms);
251         if (ks) {
```

```
252          printk_unused_warning(name);
253          *crc = symversion(mod->unused_crcs, (ks - mod->unused_syms));
254          return ks->value;
255       }
256
257       if (gplok) {
258          ks = lookup_symbol(name, mod->unused_gpl_syms,
259                   mod->unused_gpl_syms + mod->num_unused_gpl_syms);
260          if (ks) {
261             printk_unused_warning(name);
262             *crc = symversion(mod->unused_gpl_crcs,
263                      (ks - mod->unused_gpl_syms));
264             return ks->value;
265          }
266       }
267       ks = lookup_symbol(name, mod->gpl_future_syms,
268                 (mod->gpl_future_syms +
269                  mod->num_gpl_future_syms));
270       if (ks) {
271          if (!gplok) {
272             printk(KERN_WARNING "Symbol %s is being used "
273                   "by a non-GPL module, which will not "
274                   "be allowed in the future\n", name);
275             printk(KERN_WARNING "Please see the file "
276                   "Documentation/feature-removal-schedule.txt "
277                   "in the kernel source tree for more "
```

```
278                     "details.\n");
279           }
280           *crc = symversion(mod->gpl_future_crcs,
281                   (ks - mod->gpl_future_syms));
282           return ks->value;
283       }
284    }
285    DEBUGP("Failed to find symbol %s\n", name);
286    return 0;
287 }
```

src/linux-2.6.20/arch/i386/kernel/module.c

```
57 int apply_relocate(Elf32_Shdr *sechdrs,
58           const char *strtab,
59           unsigned int symindex,
60           unsigned int relsec,
61           struct module *me)
62 {
63     unsigned int i;
64     Elf32_Rel *rel = (void *)sechdrs[relsec].sh_addr;
65     Elf32_Sym *sym;
66     uint32_t *location;
67
68     DEBUGP("Applying relocate section %u to %u\n", relsec,
69           sechdrs[relsec].sh_info);
```

```
70     for (i = 0; i < sechdrs[relsec].sh_size / sizeof(*rel); i++) {
71        /* This is where to make the change */
72        location = (void *)sechdrs[sechdrs[relsec].sh_info].sh_addr
73           + rel[i].r_offset;
74        /* This is the symbol it is referring to.  Note that all
75          undefined symbols have been resolved.  */
76        sym = (Elf32_Sym *)sechdrs[symindex].sh_addr
77           + ELF32_R_SYM(rel[i].r_info);
78
79        switch (ELF32_R_TYPE(rel[i].r_info)) {
80        case R_386_32:
81           /* We add the value into the location given */
82           *location += sym->st_value;
83           break;
84        case R_386_PC32:
85           /* Add the value, subtract its postition */
86           *location += sym->st_value - (uint32_t)location;
87           break;
88        default:
89           printk(KERN_ERR "module %s: Unknown relocation: %u\n",
90                 me->name, ELF32_R_TYPE(rel[i].r_info));
91           return -ENOEXEC;
92        }
93     }
94     return 0;
95 }
```

src\linux-2.6.20\kernel\module.c

```
1227  /*
1228   * Ensure that an exported symbol [global namespace] does not already exist
1229   * in the Kernel or in some other modules exported symbol table.
1230   */
      该函数用以验证载入的 LKM 中的 symbol 没有与当前系统中已有的 symbol 冲突的。
1231  static int verify_export_symbols(struct module *mod)
1232  {
1233      const char *name = NULL;
1234      unsigned long i, ret = 0;
1235      struct module *owner;
1236      const unsigned long *crc;
1237
          依次枚举普通输出 symbol table 中的 symbol
1238      for (i = 0; i < mod->num_syms; i++)
1239              if (__find_symbol(mod->syms[i].name, &owner, &crc, 1)) {
1240              name = mod->syms[i].name;
1241              ret = -ENOEXEC;
1242              goto dup;
1243          }
1244
          依次枚举只输出给符合 GPL 发行的 LKM 的 GPL symbol table 中的 symbol
1245      for (i = 0; i < mod->num_gpl_syms; i++)
1246              if (__find_symbol(mod->gpl_syms[i].name, &owner, &crc, 1)) {
```

```
1247            name = mod->gpl_syms[i].name;
1248            ret = -ENOEXEC;
1249            goto dup;
1250        }
1251
1252  dup:
1253     if (ret)
1254        printk(KERN_ERR "%s: exports duplicate symbol %s (owned by %s)\n",
1255           mod->name, name, module_name(owner));
1256
1257     return ret;
1258  }
```

src\linux-2.6.20\kernel\module.c

```
1454  #ifdef CONFIG_KALLSYMS
1455  int is_exported(const char *name, const struct module *mod)
1456  {
1457     if (!mod && lookup_symbol(name, __start___ksymtab, __stop___ksymtab))
1458        return 1;
1459     else
1460        if (mod && lookup_symbol(name, mod->syms, mod->syms + mod->num_syms))
1461           return 1;
1462        else
1463           return 0;
1464  }
1465
```

```
1466  /* As per nm */
1467  static char elf_type(const Elf_Sym *sym,
1468               Elf_Shdr *sechdrs,
1469               const char *secstrings,
1470               struct module *mod)
1471  {
1472     if (ELF_ST_BIND(sym->st_info) == STB_WEAK) {
1473        if (ELF_ST_TYPE(sym->st_info) == STT_OBJECT)
1474           return 'v';
1475        else
1476           return 'w';
1477     }
1478     if (sym->st_shndx == SHN_UNDEF)
1479        return 'U';
1480     if (sym->st_shndx == SHN_ABS)
1481        return 'a';
1482     if (sym->st_shndx >= SHN_LORESERVE)
1483        return '?';
1484     if (sechdrs[sym->st_shndx].sh_flags & SHF_EXECINSTR)
1485        return 't';
1486     if (sechdrs[sym->st_shndx].sh_flags & SHF_ALLOC
1487        && sechdrs[sym->st_shndx].sh_type != SHT_NOBITS) {
1488        if (!(sechdrs[sym->st_shndx].sh_flags & SHF_WRITE))
1489           return 'r';
1490        else if (sechdrs[sym->st_shndx].sh_flags & ARCH_SHF_SMALL)
1491           return 'g';
```

```
1492         else
1493            return 'd';
1494      }
1495      if (sechdrs[sym->st_shndx].sh_type == SHT_NOBITS) {
1496         if (sechdrs[sym->st_shndx].sh_flags & ARCH_SHF_SMALL)
1497            return 's';
1498         else
1499            return 'b';
1500      }
1501      if (strncmp(secstrings + sechdrs[sym->st_shndx].sh_name,
1502            ".debug", strlen(".debug")) == 0)
1503         return 'n';
1504      return '?';
1505   }
1506
1507   static void add_kallsyms(struct module *mod,
1508               Elf_Shdr *sechdrs,
1509               unsigned int symindex,
1510               unsigned int strindex,
1511               const char *secstrings)
1512   {
1513      unsigned int i;
1514
1515      mod->symtab = (void *)sechdrs[symindex].sh_addr;
1516      mod->num_symtab = sechdrs[symindex].sh_size / sizeof(Elf_Sym);
1517      mod->strtab = (void *)sechdrs[strindex].sh_addr;
```

```
1518
1519      /* Set types up while we still have access to sections. */
1520      for (i = 0; i < mod->num_symtab; i++)
1521         mod->symtab[i].st_info
1522            = elf_type(&mod->symtab[i], sechdrs, secstrings, mod);
1523   }
1524   #else
1525   static inline void add_kallsyms(struct module *mod,
1526                Elf_Shdr *sechdrs,
1527                unsigned int symindex,
1528                unsigned int strindex,
1529                const char *secstrings)
1530   {
1531   }
1532   #endif /* CONFIG_KALLSYMS */
```

src/linux-2.6.20/arch/i386/kernel/module.c

```
108 int module_finalize(const Elf_Ehdr *hdr,
109           const Elf_Shdr *sechdrs,
110           struct module *me)
111 {
112    const Elf_Shdr *s, *text = NULL, *alt = NULL, *locks = NULL,
113       *para = NULL;
114    char *secstrings = (void *)hdr + sechdrs[hdr->e_shstrndx].sh_offset;
115
```

```
116    for (s = sechdrs; s < sechdrs + hdr->e_shnum; s++) {
       对 LKM 中的 section table 进行枚举。
117       if (!strcmp(".text", secstrings + s->sh_name))
118          text = s;
119       if (!strcmp(".altinstructions", secstrings + s->sh_name))
120          alt = s;
121       if (!strcmp(".smp_locks", secstrings + s->sh_name))
122          locks= s;
123       if (!strcmp(".parainstructions", secstrings + s->sh_name))
124          para = s;
125    }
       通过枚举，获得分别指向.text section，.altinstructions section，.smp_locks section，.parainstructions section
       的指针。

       .altinstructions section 是用来用新的比较好的指令来替换掉老的指令的。
       #define mb() alternative("lock; addl $0,0(%%esp)", "mfence", X86_FEATURE_XMM2)

       .parainstructions section 不知道是干啥用的，唉，Linux 发展太快，总赶不上趟。
126
127    if (alt) {
128       /* patch .altinstructions */
129       void *aseg = (void *)alt->sh_addr;
130       apply_alternatives(aseg, aseg + alt->sh_size);
131    }
132    if (locks && text) {
133       void *lseg = (void *)locks->sh_addr;
```

```
134       void *tseg = (void *)text->sh_addr;
135       alternatives_smp_module_add(me, me->name,
136                     lseg, lseg + locks->sh_size,
137                     tseg, tseg + text->sh_size);
138    }
139
140    if (para) {
141       void *pseg = (void *)para->sh_addr;
142       apply_paravirt(pseg, pseg + para->sh_size);
143    }
144
145    return module_bug_finalize(hdr, sechdrs, me);
146 }
```

src/linux-2.6.20/kernel/params.c

处理传递给 insmod utility 的参数。
name 是模块名
args 是已经拷贝到内核空间的参数列表字符串。
params 指向 LKM image 中的__param section。在 LKM 源代码中用 module_param 宏说明的参数信息在该 section 中。如例子代码中的

```
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
```

```
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
```

在__param section 中是 kernel_param 结构的数组，num 为数组的大小。

这里传入的 unknown 参数为 null。该参数是个 callback 函数，用于当内核没法找到匹配的参数时，调用该函数。比如在 LKM 源代码里，programmer 没有提供如下定义：

```
module_param(mylong, long, S_IRUSR);
```

但却在载入时的命令行上传入对该变量的设置。

```
$ insmod hello-5.ko mylong=50
```

则内核在__param section 中将找不到对该变量的说明。并不是通过命令行上可以订制的初始化 LKM 中的任何全局变量，而是必须用 module_param 宏说明的才行。原因即是在 parse_args 函数中获得订制的初始化变量所需要的信息只是从__param section 中去获得，而不是从其他更全面的渠道获得，比如 symbol section 等。

```
129 /* Args looks like "foo=bar,bar2 baz=fuz wiz". */
130 int parse_args(const char *name,
131          char *args,
132          struct kernel_param *params,
133          unsigned num,
134          int (*unknown)(char *param, char *val))
135 {
136    char *param, *val;
137
138    DEBUGP("Parsing ARGS: %s\n", args);
```

```
139
140   /* Chew leading spaces */
141   while (*args == ' ')
142      args++;
143
144   while (*args) {
145      int ret;
146      int irq_was_disabled;
147
148      args = next_arg(args, &param, &val);
149      irq_was_disabled = irqs_disabled();
150      ret = parse_one(param, val, params, num, unknown);
151      if (irq_was_disabled && !irqs_disabled()) {
152         printk(KERN_WARNING "parse_args(): option '%s' enabled "
153                "irq's!\n", param);
154      }
155      switch (ret) {
156      case -ENOENT:
157         printk(KERN_ERR "%s: Unknown parameter `%s'\n",
158                name, param);
159         return ret;
160      case -ENOSPC:
161         printk(KERN_ERR
162                "%s: `%s' too large for parameter `%s'\n",
163                name, val ?: "", param);
164         return ret;
```

```
165       case 0:
166          break;
167       default:
168          printk(KERN_ERR
169                "%s: `%s' invalid for parameter `%s'\n",
170                name, val ?: "", param);
171          return ret;
172       }
173    }
174
175    /* All parsed OK. */
176    return 0;
177 }
```

```
49  static int parse_one(char *param,
50              char *val,
51              struct kernel_param *params,
52              unsigned num_params,
53              int (*handle_unknown)(char *param, char *val))
54  {
55     unsigned int i;
56
    与__param section 中的信息进行比较，看有没有这参数，如有则初始化之。
57     /* Find parameter */
58     for (i = 0; i < num_params; i++) {
59        if (parameq(param, params[i].name)) {
```

```
60            DEBUGP("They are equal!  Calling %p\n",
61                   params[i].set);
62            return params[i].set(val, &params[i]);
63        }
64    }
65
66    if (handle_unknown) {
67        DEBUGP("Unknown argument: calling %p\n", handle_unknown);
68        return handle_unknown(param, val);
69    }
70
71    DEBUGP("Unknown argument `%s'\n", param);
72    return -ENOENT;
73 }
74
75 /* You can use " around spaces, but can't escape ". */
76 /* Hyphens and underscores equivalent in parameter names. */
```

整个代码如果用 regular express 表达会及其简单，就是在提取 param=value 对。当看到具体 parse 字符串时，才发觉 regular express 是多么的有用。内核中是否应该弄个 regular express 的引擎呢。我觉得是应该的。在现在物理内存动则快上 G 的情况下，借口内核要小，好像不是很有说服力。毕竟有了它，象如下那种难看的字符串分析（我觉得几乎所有字符串分析都难看）就可以避免。对写代码与读代码的人都是种解脱。

```
77 static char *next_arg(char *args, char **param, char **val)
78 {
79    unsigned int i, equals = 0;
80    int in_quote = 0, quoted = 0;
81    char *next;
```

```
82
83     if (*args == '"') {
84        args++;
85        in_quote = 1;
86        quoted = 1;
87     }
88
89     for (i = 0; args[i]; i++) {
90        if (args[i] == ' ' && !in_quote)
91           break;
92        if (equals == 0) {
93           if (args[i] == '=')
94              equals = i;
95        }
96        if (args[i] == '"')
97           in_quote = !in_quote;
98     }
99
100    *param = args;
101    if (!equals)
102       *val = NULL;
103    else {
104       args[equals] = '\0';
105       *val = args + equals + 1;
106
107       /* Don't include quotes in value. */
```

```
108       if (**val == '"') {
109          (*val)++;
110          if (args[i-1] == '"')
111             args[i-1] = '\0';
112       }
113       if (quoted && args[i-1] == '"')
114          args[i-1] = '\0';
115    }
116
117    if (args[i]) {
118       args[i] = '\0';
119       next = args + i + 1;
120    } else
121       next = args + i;
122
123    /* Chew up trailing spaces. */
124    while (*next == ' ')
125       next++;
126    return next;
127 }
```

附录

例子代码

```
[wzhou@localhost example]$ cat hello-5.c
/*
 *  hello-5.c - Demonstrates command line argument passing to a module.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";

/*
 * module_param(foo, int, 0000)
```

```
 * The first param is the parameters name
 * The second param is it's data type
 * The final argument is the permissions bits,
 * for exposing parameters in sysfs (if non-zero) at a later stage.
 */

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

static int __init hello_5_init(void)
{
       printk(KERN_ALERT "Hello, world 5\n=============\n");
       printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
       printk(KERN_ALERT "myint is an integer: %d\n", myint);
       printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
       printk(KERN_ALERT "mystring is a string: %s\n", mystring);
       return 0;
}

static void __exit hello_5_exit(void)
```

```
{
	printk(KERN_ALERT "Goodbye, world 5\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

Makefile

在 2.6 内核下，编译 LKM 的 Makefile 的书写比 2.4 内核下要简单多了。就一行。

```
[wzhou@localhost example]$ cat Makefile
obj-m += hello-5.o
```

编译命令

```
[wzhou@localhost example]$ make -C ~wzhou/src/linux-2.6.20/ SUBDIRS=$PWD modules
```

联系

Walter Zhou

mailto:z-l-dragon@hotmail.com