

少帅的天空

深入理解linux 深入理解虚拟化 深入理解云计算

目录视图

摘要视图

RSS 订阅

个人资料



ustc_dylan

访问: 410164次

积分: 6632分

排名: 第862名

原创: 230篇 转载: 62篇

译文: 0篇 评论: 165条

文章搜索

文章分类

C/C++ (27)

linux (27)

GUI 图形库 (1)

linux2.6.xx内核代码分析 (72)

linux系统编程 (20)

Qt学习 (4)

学习漫谈 (47)

运维管理 (12)

数据结构学习 (3)

Virtualization && QEMU (25)

OpenNebula (9)

OpenStack (11)

文章存档

2014年06月 (1)

2014年04月 (4)

2014年03月 (1)

2014年02月 (1)

2014年01月 (2)

展开

阅读排行

有奖征资源, 博文分享有内涵 社区问答: 半峭iOS测试指南 专访阿里陶辉 2014 CSDN博文大赛 10月微软MVP申请

qemu中ELF文件的加载

分类: linux Virtualization && QEMU 2011-12-13 20:34 2215人阅读 评论(2) 收藏 举报

header

struct

image

table

数据结构

linux

前段时间分析了qemu中ELF文件的加载过程, 个人感觉通过这个分析不但可以加深对ELF文件格式的理解, 而且能够从侧面了解操作系统加载器的工作过程。

一、ELF相关的背景知识

1. ELF格式文件相关概念

ELF格式文件主要包括以下三种类型的文件:

• 可重定位的目标文件(.o文件) --> 用于链接生成可执行文件或动态链接库文件(.so)

• 可执行文件 --> 进程映像文件

• 共享库(.so文件) --> 用于链接

从链接和执行的角​​度来讲, ELF文件存在两种视图: 链接视图和执行视图。为了区分两种视图, 只需记住链接视图由多个section组成, 而执行视图由多个segment组成即可。另外, section是程序员可见的, 是给链接器使用的概念, 汇编文件中通常会显示的定义.text, .data等section, 相反, segment是程序员不可见的, 是给加载器使用的概念。下图形象的描述了ELF文件两种不同的视图的结构以及二者之间的联系。



linkable sections

executable segments

ELF header

program header table

describes segments

sections

describes sections

section header table

segments

(optional, ignored)

二者之间的联系在于: 一个segment包含一个或多个section。

注意: Section Header Table和Program Header Table并不是一定要位于文件的开头和结尾,其位置由ELF Header指出,上图这么画只是为了清晰。-- ELF文件每个部分的详细介绍参见《ELF 文件格式分析》。

TN05.ELF.Format.Summary.pdf

2. ELF文件主要数据结构

上面讲了ELF的相关概念, 但是要想用计算机语言(C语言)来实现, 必须对应相应的数据结构。linux下通过三个数据结构描述了ELF文件的相关概念。

http://blog.csdn.net/ustc_dylan/article/details/7068190

1/8

ubuntu下安装texlive 201	(8962)
8259A中断控制器详细介绍	(8416)
KVM虚拟机和QEMU（奇	(8407)
RCU 机制	(7282)
qemu的安装步骤详解(适	(7111)
Makefile常用函数总结	(5315)
QEMU源码分析系列(一)	(5178)
vim-latex 的安装与配置	(5040)
fcntl设置FD_CLOEXEC	(4988)
linux编程之mprotect	(4820)

评论排行

重大更正：CFS调度是没	(12)
8259A中断控制器详细介绍	(12)
ubuntu 11.10 安装system	(12)
ubuntu 12.04下devstack	(9)
OpenNebula 镜像管理分	(7)
RDO多节点部署OpenSt	(7)
openstack网络模式之vla	(7)
qemu源码分析之四--dyn	(6)
ubuntu 11.10 Texlive 20	(6)
qemu源码分析系列(二)	(5)

最新评论

openstack policy 鉴权过程分析
zx4052542: (1) context: 执行
resize操作的上下文，其内容包
括project_id, user_...

openstack网络模式之vlan分析
wodeyuer125: 很受益啊~~啥时
候能讲讲gre模式呢？

RDO多节点部署OpenStack Hav
rongzhuorong: 博主你好，我配
置一个两节点环境的时候，一直
提示下面这个错误，说重复的定
义了"ssh-rsa.local...

RDO多节点部署OpenStack Hav
linuxkernel: 您好,请问能发表您
的answer.conf 配置文件吗？谢
谢！

ubuntu下安装texlive 2010步骤和
u010060511: 按照步骤一步一步
来 终于可以用latex写中文文档
了！

softirq原理以及源码分析
oNightelf: 博主您好，dmsg 中
有NOHZ: local_softirq_pending
100。请问这是什...

OpenStack Icehouse error: Virtu
u012336994: 您好，我用
devstack在centos上装完
icehouse以后，启动云主机的时
候也遇到一样的问题...

Openstack压力测试(二) -- 结果过
wangweino01: 恩 Neutron 还是
单独部署比较好

OpenStack压力测试（批量创建
ustc_dylan: @wangweino01:现
在测试已经好多了，用的是
qpid，新的测试结果：
http://blog....

OpenStack压力测试（批量创建
wangweino01: 哪怕平时也经常
timeout，博主用的是哪个
message queue？rabbit mq
么？

友情链接

linuxsky.blog.chinaunix.net

(1) ELF Header

ELF Header描述了体系结构和操作系统等基本信息,并指出Section Header Table和Program Header Table在文件
中的什么位置，每个成员的解释参见注释及附件。

```
[cpp]
01. #define EI_NIDENT 16
02. typedef struct{
03. /*ELF的一些标识信息，固定值*/
04. unsigned char e_ident[EI_NIDENT];
05. /*目标文件类型：1-可重定位文件，2-可执行文件，3-共享目标文件等*/
06. Elf32_Half e_type;
07. /*文件的目标体系结构类型：3-intel 80386*/
08. Elf32_Half e_machine;
09. /*目标文件版本：1-当前版本*/
10. Elf32_Word e_version;
11. /*程序入口的虚拟地址，如果没有入口，可为0*/
12. Elf32_Addr e_entry;
13. /*程序头表(segment header table)的偏移量，如果没有，可为0*/
14. Elf32_Off e_phoff;
15. /*节区头表(section header table)的偏移量，没有可为0*/
16. Elf32_Off e_shoff;
17. /*与文件相关的，特定于处理器的标志*/
18. Elf32_Word e_flags;
19. /*ELF头部的大小，单位字节*/
20. Elf32_Half e_ehsize;
21. /*程序头表每个表项的大小，单位字节*/
22. Elf32_Half e_phentsize;
23. /*程序头表表项的个数*/
24. Elf32_Half e_phnum;
25. /*节区头表每个表项的大小，单位字节*/
26. Elf32_Half e_shentsize;
27. /*节区头表表项的数目*/
28. Elf32_Half e_shnum;
29. /**/
30. Elf32_Half e_shstrndx;
31. }Elf32_Ehdr;
```

下面通过一个具体实例来说明ELF header中每个数据成员对应的值，下面是hello world的ELF文件头，在linux下可
以通过"readelf -h ELF文件名"来获得。

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                         0
  Type:                                EXEC (Executable file)
  Machine:                             Intel 80386
  Version:                             0x1
  Entry point address:                  0x8048320
  Start of program headers:             52 (bytes into file)
  Start of section headers:            4412 (bytes into file)
  Flags:                                0x0
  Size of this header:                  52 (bytes)
  Size of program headers:              32 (bytes)
  Number of program headers:            9
  Size of section headers:              40 (bytes)
  Number of section headers:            30
  Section header string table index:    27
```

ELF Header用数据结构Elf32_Ehdr来表示，描述了操作系统是UNIX,体系结构是80386。Section Header Table中
有30个Section Header,从文件地址4412开始,每个Section Header占40字节,Segment Header Table中有9个
segment，每个segment header占32个字节，此ELF文件的类型是可执行文件(EXEC)，入口地址是0x8048320。

(2) Section Header Table Entry

从ELF Header中可知，每个ELF文件有个Section Header Table，其中每一个表项对应一个section，由数据结构
Elf32_Shdr来描述，每个成员的含义参见注释及附件。在linux下可以通过"readelf -S ELF文件名"来查看。

```
[cpp]
01. typedef struct{
```

```
02.  /*节区名称*/
03.  Elf32_Word sh_name;
04.  /*节区类型: PROGBITS-程序定义的信息, NOBITS-不占用文件空间(bss), REL-重定位表项*/
05.  Elf32_Word sh_type;
06.  /*每一bit位代表一种信息, 表示节区内的内容是否可以修改, 是否可执行等信息*/
07.  Elf32_Word sh_flags;
08.  /*如果节区将出现在进程的内存影响中, 此成员给出节区的第一个字节应处的位置*/
09.  Elf32_Addr sh_addr;
10.  /*节区的第一个字节与文件头之间的偏移*/
11.  Elf32_Off sh_offset;
12.  /*节区的长度, 单位字节, NOBITS虽然这个值非0但不占文件中的空间*/
13.  Elf32_Word sh_size;
14.  /*节区头部表索引链接*/
15.  Elf32_Word sh_link;
16.  /*节区附加信息*/
17.  Elf32_Word sh_info;
18.  /*节区带有地址对齐的约束*/
19.  Elf32_Word sh_addralign;
20.  /*某些节区中包含固定大小的项目, 如符号表, 那么这个成员给出其固定大小*/
21.  Elf32_Word sh_entsize;
22.  }Elf32_Shdr;
```

(3) Program Header Table Entry

从ELF Header中可知, 每个ELF文件有个Program Header Table, 其中每一个表项对应一个segment, 由数据结构Elf32_phdr来描述, 每个成员的含义参见注释及附件。在linux下可以通过“readelf -l ELF文件名”来查看。

```
[cpp]
01.  typedef struct
02.  {
03.      /*segment的类型: PT_LOAD= 1 可加载的段*/
04.      Elf32_Word p_type;
05.      /*从文件头到该段第一个字节的偏移*/
06.      Elf32_Off p_offset;
07.      /*该段第一个字节被放到内存中的虚拟地址*/
08.      Elf32_Addr p_vaddr;
09.      /*在linux中这个成员没有任何意义, 值与p_vaddr相同*/
10.      Elf32_Addr p_paddr;
11.      /*该段在文件映像中所占的字节数*/
12.      Elf32_Word p_filesz;
13.      /*该段在内存映像中占用的字节数*/
14.      Elf32_Word p_memsz;
15.      /*段标志*/
16.      Elf32_Word p_flags;
17.      /*p_vaddr是否对齐*/
18.      Elf32_Word p_align;
19.  } Elf32_phdr;
```

二、qemu中ELF文件的加载过程

在了解了ELF文件的基本结构之后, 大体可以想到ELF文件的加载过程就是一个查表的过程, 即通过ELF Header得到ELF文件的基本信息-Section Header Table和program Header Table,然后再根据Section Header Table和program Header Table的信息加载ELF文件中的相应部分。上面也提到过, section是从链接器的角度来讲的概念, 所以, ELF文件的加载过程中, 只有segment是有效的, 加载器根据program Header Table中的信息来负责ELF文件的加载。

首先, 从感性上认识一下segment, 还是以上面的hello world为例, 其对应的program header table如下。

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x005c4	0x005c4	R E	0x1000
LOAD	0x000f14	0x08049f14	0x08049f14	0x00100	0x00108	RW	0x1000
DYNAMIC	0x000f28	0x08049f28	0x08049f28	0x000c8	0x000c8	RW	0x4
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x0004d0	0x080484d0	0x080484d0	0x00034	0x00034	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f14	0x08049f14	0x08049f14	0x000ec	0x000ec	R	0x1

第一列type即每个segment的类型，每个类型的具体含义详见附件。通常我们之关心程序的代码段(.text section)和数据段(.data section)，这两个section组成LOAD类型的segment。

Offset: 当前segment加载到的地址的偏移

VirtAddr: 当前segment加载到的虚拟地址

PhysAddr: 当前segment加载到的物理地址（x86平台上，此值没有意义，并不指物理地址）

FileSiz: 当前segment在ELF文件中的偏移

MemSiz: 当前segment在内存页中的偏移

Flg: segment的权限，R-可读，W-可写，E-可执行

Align: x86平台内存页面的大小

在了解了segment的相关信息后，分析下qemu代码中ELF文件的加载过程，印证下上面提到的ELF文件的加载的思想。

[cpp]

```
01. ret = loader_exec(filename, target_argv, target_environ, regs, info, &bprm);
02.
03. filename: 要加载的ELF文件的名称
04. target_argv: qemu运行的参数，在这里即hello (hello是生成的可执行文件名， $qemu hello)
05. target_environ: 执行qemu的shell的环境变量
06. regs, info, bprm是ELF文件加载过程中涉及三个重要数据结构，下面会详细分析。
07.
08. er_exec函数的功能及含义参见代码注释。
09.
10. int loader_exec(const char* filename, char** argv, char** envp,
11.                 struct target_pt_regs * regs, struct image_info* info,
12.                 struct linux_binprm *bprm)
13. {
14.     int retval;
15.     int i;
16.     bprm->p = TARGET_PAGE_SIZE * MAX_ARG_PAGES - sizeof(unsigned int); /*MAX_ARG_PAGES= 33*/
17.     memset(bprm->page, 0, sizeof(bprm->page));
18.     retval = open(filename, O_RDONLY); /*返回打开文件的fd*/
19.     if (retval < 0)
20.         return retval;
21.     bprm->fd = retval;
22.     bprm->filename = (char *)filename;
23.     bprm->argc = count(argv);
24.     bprm->argv = argv;
25.     bprm->envc = count(envp);
26.     bprm->envp = envp;
27.     /*1. 要加载文件的属性判断: 是否常规文件, 是否可执行文件, 是否ELF文件; 2. 读取ELF文件的前1024个字节*/
28.     retval = prepare_binprm(bprm);
29.     if (retval >= 0) { /*prepare_binprm函数已经读出了目标文件的前1024个字节, 先判断下这个文件是否是ELF文件, 即前4个字节*/
30.         if (bprm->buf[0] == 0x7f
31.             && bprm->buf[1] == 'E'
32.             && bprm->buf[2] == 'L'
33.             && bprm->buf[3] == 'F') {
34.             retval = load_elf_binary(bprm, regs, info);
35. #if defined(TARGET_HAS_BFLT)
36.         } elseif (bprm->buf[0] == 'b'
37.             && bprm->buf[1] == 'F'
38.             && bprm->buf[2] == 'L'
39.             && bprm->buf[3] == 'T') {
40.             retval = load_flt_binary(bprm, regs, info);
41.         }
42.     }
43. }
```

```

42.     } else{
43.         fprintf(stderr, "Unknown binary format\n");
44.         return -1;
45.     }
46. }
47. if(retval>=0){
48.     /* success. Initialize important registers*/
49.     do_init_thread(regs, infop);
50.     return retval;
51. }
52. /* Something went wrong, return the inode and free the argument pages*/
53. for (i=0; i<MAX_ARG_PAGES; i++){
54.     g_free(bprm->page[i]);
55. }
56. return(retval);
57. }
58.
59. int load_elf_binary(struct linux_binprm* bprm, struct target_pt_regs* regs,
60.                    struct image_info * info)
61. {
62.     struct image_info interp_info;
63.     struct elfhdr elf_ex;
64.     char *elf_interpreter = NULL;
65.     info->start_mmap= (abi_ulong)ELF_START_MMAPP;          /*ELF_START_MMAPP= 0x80000000*/
66.     info->mmap= 0;
67.     info->rss= 0;
68.     /*主要工作就是初始化info, 申请进程虚拟地址空间, 将ELF文件映射到这段虚拟地址空间上*/
69.     load_elf_image(bprm->filename, bprm->fd, info,
70.                   &elf_interpreter, bprm->buf);
71.
72.     ... ..
73.
74.
75.     return 0;
76. }
77.
78. static void load_elf_image(const char*image_name,int image_fd,
79.                            struct image_info *info, char**pinterp_name,
80.                            char bprm_buf[BPRM_BUF_SIZE])
81. {
82.     struct elfhdr *ehdr = (struct elfhdr *)bprm_buf;
83.     struct elf_phdr *phdr;
84.     abi_ulong load_addr, load_bias, loadaddr, hiaddr,error;
85.     int i, retval;
86.     const char *errmsg;
87.     /* First of all, some simple consistency checks*/
88.     errmsg = "Invalid ELF image for this architecture";
89.     if (!elf_check_ident(ehdr)){/*ELF头检查*/
90.         goto exit_errmsg;
91.     }
92.     bswap_ehdr(ehdr); /*当前为空, 是不是主机和目标机大小尾端不一致时才会swap*/
93.     if (!elf_check_ehdr(ehdr)){
94.         goto exit_errmsg;
95.     }
96.     /*下面的代码即读出ELF文件的程序头表, 首先判断下是否已经被完全读出*/
97.     i = ehdr->e_phnum* sizeof(struct elf_phdr); /*program header 表的大小*/
98.     if (ehdr->e_phoff+ i <= BPRM_BUF_SIZE){
99.         phdr = (struct elf_phdr *) (bprm_buf+ ehdr->e_phoff);
100.     } else{
101.         phdr = (struct elf_phdr *) alloca(i); /*申请i个程序头部*/
102.         retval = pread(image_fd, phdr, i, ehdr->e_phoff); /*从文件image_id的偏移为ehdr-
103.         >e_phoff处读取i个字节到phdr中, 即phdr存放program header*/
104.         if (retval!= i){
105.             goto exit_read;
106.         }
107.     }
108.     bswap_phdr(phdr, ehdr->e_phnum);
109. #ifdef CONFIG_USE_FDPIC
110.     info->nsegs= 0;
111.     info->pt_dynamic_addr= 0;
112. #endif
113.     /* Find the maximum size of the image and allocate an appropriate
114.     amount of memory to handle that.*/
115.     loadaddr = -1, hiaddr= 0;
116.     for (i= 0; i < ehdr->e_phnum; ++i){/*遍历每一个program header*/
117.         if (phdr[i].p_type== PT_LOAD){
118.             abi_ulong a = phdr[i].p_vaddr;
119.             if (a < loadaddr){ /*loadaddr= -1而且是unsigned 类型的, 所以loadaddr是个很大的数
120.

```

```

119.         loadaddr = a;           /*loadaddr记录segment的起始地址*/
120.     }
121.     a += phdr[i].p_memsz;        /*这个segment在内存中的偏移地址*/
122.     if (a > hiaddr){            /*hiaddr记录segment的结束地址*/
123.         hiaddr = a;
124.     }
125. #ifdef CONFIG_USE_FDPIC
126.     ++info->nsegs;
127. #endif
128. }
129. }
130. load_addr = loadaddr;           /*计算出来的需要加载的起始地址*/
131. if (ehdr->e_type== ET_DYN){      /*共享目标文件(.so)*/
132.     /* The image indicates that it can be loaded anywhere. Find a
133.        location that can hold the memoryspace required.If the
134.        image is pre-linked, LOADADDR will be non-zero. Since wedo
135.        not supply MAP_FIXED here we'll use that addressif and
136.        only if it remains available.*/
137.     load_addr = target_mmap(loadaddr, hiaddr- loadaddr, PROT_NONE,
138.                             MAP_PRIVATE | MAP_ANON| MAP_NORESERVE,
139.                             -1, 0);
140.     if (load_addr== -1) {
141.         goto exit_perror;
142.     }
143. } elseif (pinterp_name!= NULL) {
144.     /* Thisis the main executable. Make sure that the low
145.        address does not conflict with MMAP_MIN_ADDRor the
146.        QEMU application itself. */
147.     probe_guest_base(image_name, loadaddr, hiaddr);
148. }
149. load_bias = load_addr - loadaddr;
150. #ifdef CONFIG_USE_FDPIC
151. {
152.     struct elf32_fdpic_loadseg *loadsegs= info->loadsegs=
153.         g_malloc(sizeof(*loadsegs)* info->nsegs);
154.     for (i= 0; i < ehdr->e_phnum;++i){
155.         switch (phdr[i].p_type){
156.             case PT_DYNAMIC:
157.                 info->pt_dynamic_addr= phdr[i].p_vaddr+ load_bias;
158.                 break;
159.             case PT_LOAD:
160.                 loadsegs->addr= phdr[i].p_vaddr+ load_bias;
161.                 loadsegs->p_vaddr= phdr[i].p_vaddr;
162.                 loadsegs->p_memsz= phdr[i].p_memsz;
163.                 ++loadsegs;
164.                 break;
165.         }
166.     }
167. }
168. #endif
169. info->load_bias= load_bias;        /*真实的加载地址和计算出来(读ELF头信息)的加载地址之差*/
170. info->load_addr= load_addr;        /*真实的加载地址*/
171. info->entry= ehdr->e_entry+ load_bias; /*重新调整下程序的入口*/
172. info->start_code= -1;
173. info->end_code= 0;
174. info->start_data= -1;
175. info->end_data= 0;
176. info->brk= 0;
177. for (i= 0; i < ehdr->e_phnum; i++){
178.     struct elf_phdr *eppnt = phdr + i;
179.     if (eppnt->p_type== PT_LOAD){
180.         abi_ulong vaddr, vaddr_po, vaddr_ps, vaddr_ef, vaddr_em;
181.         int elf_prot = 0;
182.         /*记录PT_LOAD类型segment的权限: 读/写/可执行*/
183.         if (eppnt->p_flags& PF_R) elf_prot= PROT_READ;
184.         if (eppnt->p_flags& PF_W) elf_prot|= PROT_WRITE;
185.         if (eppnt->p_flags& PF_X) elf_prot|= PROT_EXEC;
186.         vaddr = load_bias + eppnt->p_vaddr;
187.         vaddr_po = TARGET_ELF_PAGEOFFSET(vaddr);/*((vaddr)& ((1<< 12)-1)), 目的是取页内偏
移*/
188.         vaddr_ps = TARGET_ELF_PAGESTART(vaddr); /*((vaddr)& ~(unsigned long)
((1<< 12)-1)), 向下页对齐, 目的取页对齐的地址*/
189.         /*将ELF文件映射到进程地址空间中*/
190.         error= target_mmap(vaddr_ps, eppnt->p_filesz+ vaddr_po, /*映射的时候从页内偏移
vaddr_po开始映射, 即保持原来的偏移量*/
191.                             elf_prot, MAP_PRIVATE| MAP_FIXED,
192.                             image_fd, eppnt->p_offset- vaddr_po);
193.         if (error ==-1) {
194.             goto exit_perror;

```



```

195.     }
196.     vaddr_ef = vaddr + eppnt->p_filesz;
197.     vaddr_em = vaddr + eppnt->p_memsz;
198.     /*If the load segment requests extra zeros (e.g. bss), map it.*/
199.     if (vaddr_ef < vaddr_em){
200.         zero_bss(vaddr_ef, vaddr_em, elf_prot);
201.     }
202.     /* Find the full program boundaries.*/
203.     if (elf_prot & PROT_EXEC){
204.         if (vaddr < info->start_code){
205.             info->start_code= vaddr;    /*代码段的起始虚拟地址 (页对齐的地址) */
206.         }
207.         if (vaddr_ef > info->end_code){
208.             info->end_code= vaddr_ef;    /*代码段的结束虚拟地址 (页对齐的地址) */
209.         }
210.     }
211.     if (elf_prot & PROT_WRITE){
212.         if (vaddr < info->start_data){
213.             info->start_data= vaddr;    /*数据段的起始虚拟地址*/
214.         }
215.         if (vaddr_ef > info->end_data){
216.             info->end_data= vaddr_ef;    /*数据段的起始虚拟地址 (包括bss段的大小) */
217.         }
218.         if (vaddr_em > info->brk){
219.             info->brk= vaddr_em;    /*程序内存映像的顶端 (代码段+数据段+bss段) */
220.         }
221.     }
222. } elseif (eppnt->p_type== PT_INTERP&& pinterp_name){/*内部解释程序名称: /lib/ld-
linux.so.2*/
223.     char *interp_name;
224.     if (*pinterp_name){
225.         errmsg = "Multiple PT_INTERP entries";
226.         goto exit_errmsg;
227.     }
228.     interp_name = malloc(eppnt->p_filesz);
229.     if (!interp_name){
230.         goto exit_perror;
231.     }
232.     if (eppnt->p_offset+ eppnt->p_filesz<= BPRM_BUF_SIZE){
233.         memcpy(interp_name, bprm_buf+ eppnt->p_offset,
234.             eppnt->p_filesz);
235.     } else {
236.         retval = pread(image_fd, interp_name, eppnt->p_filesz,
237.             eppnt->p_offset);
238.         if (retval != eppnt->p_filesz){
239.             goto exit_perror;
240.         }
241.     }
242.     if (interp_name[eppnt->p_filesz- 1] != 0){
243.         errmsg = "Invalid PT_INTERP entry";
244.         goto exit_errmsg;
245.     }
246.     *pinterp_name = interp_name;
247. }
248. }
249. if (info->end_data== 0){
250.     info->start_data= info->end_code;
251.     info->end_data= info->end_code;
252.     info->brk= info->end_code;
253. }
254. if (qemu_log_enabled()){
255.     load_symbols(ehdr, image_fd, load_bias);
256. }
257. close(image_fd);
258. return;
259. exit_read:
260.     if (retval>= 0){
261.         errmsg = "Incomplete read of file header";
262.         goto exit_errmsg;
263.     }
264. exit_perror:
265.     errmsg = strerror(errno);
266. exit_errmsg:
267.     fprintf(stderr,"%s: %s\n", image_name, errmsg);
268.     exit(-1);
269. }

```

上一篇 elf转化成bin后，bin文件变大的问题

下一篇 ubuntu 11.10 安装systemtap

顶 3

踩 0

主题推荐 qemu 操作系统 数据结构 程序员 application

猜你在找

- 论计算机叫兽们与林纳斯·托瓦兹
- 网卡驱动1-移植snll到linux-3.0.8
- configure: error: GRUB requires a working absolute objcopy;
- 让你不再害怕指针 系列
- 关于Mips--基本认识
- android 游戏开发之物理小球的应用
- 关于__irq 的使用
- Android常用适配器分析(如何制作简易Launcher)
- python的datetime模块功能详解
- NSData跟struct之间的转换方法

学习最新的 ANDROID 开发技术

程序员想月薪1W+,埋头改Bug,肯定实现不了,不如看看这些新技术学了没?

查看评论

2楼 sns1984 2012-11-30 15:51发表



写的很不错，学习了。

1楼 azru0512 2012-01-19 12:07发表



嗨! 您寫的 Qemu 系列文章很不錯。我在 http://www.hellogcc.org 也發了幾篇文章。希望有機會能跟您多交流。:-)

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- 全部主题
- Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC
- WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS
- Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE
- Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace
- Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate
- ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django
- Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2014, CSDN.NET, All Rights Reserved