ucore 可加载内核模块

蓝昶 2009011352

1 说明

本文档用于总结 ucore 可加载内核模块的实现原理，用于给后续的开发者参考。工程的源代码已经上传到 `https://github.com/TurfKids/ucore-kernel-module`，基于茅俊杰的 64 位基准代码。

由于本项目为操作系统课程大实验的题目，要求在 64 位 ucore 上实现，因此并没有在 32 位 ucore 上实现可加载内核模块功能。

我尽量在这份文档中体现我个人对可加载内核模块的学习和理解过程，但是可能对于初学者来说仍然会有难以理解之处。如果你觉得哪些地方需要补充，请发信给我，我会在 GitHub 上继续维护这份文档，我的邮箱地址是 `changlan9@gmail.com`。

2 文件列表

文件	说明
`kern-module/mod-add/*.[ch]`	加法模块，验证正确性用
`kern-module/mod-mul/*.[ch]`	乘法模块，验证依赖关系正确性用
`kern-module/mod-sfatfs/*.[ch]`	模块化的 SFATFS
`kern-module/Makefile`	
`kern-ucore/fs/vfs/vfs.[ch]`	文件系统注册与 `mount` 相关
`kern-ucore/module/mod.[ch]`	模块载入的初始化
`kern-ucore/module/mod_loader.[ch]`	ELF 载入和解析
`kern-ucore/module/mod_manager.[ch]`	散列表相关的辅助函数，用于管理符号表

3 可加载模块

3.1 背景

对于刚刚接触这个题目的开发者，IBM DeveloperWorks 上的一篇 Linux 可加载内核模块剖析 可以让你了解内核可加载模块的用处和背景。简单地说，系统有一部分功能只留一个函数或者变量指针作为接口，并不做实现。当用户将某个模块的代码（指令）载入到内存中时，修改系统的对应指针指向新载入的模块的地址，这就是加载模块的过程。

举个例子，Linux 的 VFS 提供了一组文件系统函数的接口，但是函数的内容在各个文件系统都不同。因此如果把 `ext4` 作为内核模块的话，加载器先把模块的二进制内容复制到内存中，然后将新的文件系统的接口指针指向二进制模块内容的正确位置。这样，系统就能动态调整支持某种文件系统与否。

简而言之，可加载内核模块的支持必需做到：

1. **导出系统内核的符号和地址到内核符号表里**，这样模块里面才能调用对应的系统符号，例如 `printf` 等等。
2. **将内核模块从文件读取到内存**。
3. **解析模块的 ELF 格式，将变量加到符号表里**。
4. **重定位**。因为加载之后 ELF 的符号地址已经发生改变，需要修正偏移量。
5. **操作系统有载入并注册模块的接口**。

3.2 导出内核符号表

继承自朱文雷学长的设计，内核符号用到的符号都在散列表里面，下面的函数用于将一个符号和地址存入散列表中。

```
void mod_touch_symbol(const char *name, uintptr_t ptr, uint32_t flags)
```

将必要的内核函数/变量符号导出到符号表里之后，模块才能调用对应函数/变量。实际上这样可以做一个访问权限控制的功能，例如为了提高安全系数，可以选择不将某些符号导出给可加载内核模块使用。

3.3 ELF 文件解析

我们尝试找现成的 ELF 文件解释器，调研了 `libElf`、`elfinfo`，但是发现编码定制和维护复杂程度并不比自己编写低。我认为 ELF 解析器的编写属于一劳永逸的，如果合乎规

范，后期基本不用对这个环节进行维护。

我写的解析器参考了 `inskmem`。`inskmem` 是个很有意思的小项目，它能在 Linux 内核不支持模块加载的情况下实现模块加载的功能，我很推荐有兴趣的读者看一下它短小的代码。当然，`inskmem` 也包含了 ELF 解析器的功能。

我们遍历各个 Section，遇到 `SH_TYPE_SYMTAB` 时，遍历各个符号项，用 `mod_touch_symbol` 把符号加到符号表，如果符号的名字是对应模块加载和卸载初始化的函数，记录下函数地址。如果符号项是 `SH_COMMON` 类型，说明符号在 ELF 里面没有空间，需要自己分配空间和地址。

遇到 `SHT_NOBITS` 段时，说明这个段在 ELF 中定位了长度但是并未分配空间，需要自己分配。

3.4 重定位

重定位是将 ELF 文件里面没有定义的符号关联到合法的地址上，例如目标文件里如果声明了 `printf`，在重定位的过程中，这个符号的引用必须指向机器内存地址中这个函数所在的地址。内核的这个功能需要我们自己实现。在 ELF 文件中，Relocation Table 就是指导解析器如何进行重定位的表，每个表项包含：

```
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
    Elf32_Sword     r_addend;
} Elf32_Rela;
```

Field	Description
`r_offset`	要修改的地址值在相对于 section 起始地址的相对位置
`r_info >> 8`	符号表中符号的索引，这个符号在 `r_offset` 被引用
`r_info & 255`	重定位类型
`r_addend`	常数，计算新地址会用到

对于不同类型的符号，处理方法也有所不同：

1. **内核符号**：类型为 `SHN_UNDEF`，即未定义的变量。我们认为这是模块中调用内核中的符号，因此从导出的符号表里直接提取地址。
2. **普通符号**：用常规处理方法，新地址为 ELF 起始地址 + 符号段偏移 + 符号表项中的地址。

3. **SHN_COMMON**：这种类型的符号我还不确定是在时候会出现，因为取决编译器和平台，但是刚才提到，符号在 ELF 里面没有空间，需要自己分配空间和地址。所以计算方法是，分配内存的起始地址 + 符号表项中记录的地址。

重定位类型是非常 tricky 的一步，平台相关性很大。我之前从学长那里接手项目的时候，之所以难以实用，很大部分原因是对于重定位类型处理不过细致，因此我参考了 Linux 的实现，解决了问题。Linux Kernel 相关代码是 `source/arch/x86/kernel/module.c` 中的 `apply_relocate_add` 函数，代码片段如下：

```
int apply_relocate_add(Elf64_Shdr *sechdrs,
                   const char *strtab,
                   unsigned int symindex,
                   unsigned int relsec,
                   struct module *me)
{
        unsigned int i;
        Elf64_Rela *rel = (void *)sechdrs[relsec].sh_addr;
        Elf64_Sym *sym;
        void *loc;
        u64 val;

        DEBUGP("Applying␣relocate␣section␣%u␣to␣%u\n", relsec,
               sechdrs[relsec].sh_info);
        for (i = 0; i < sechdrs[relsec].sh_size / sizeof(*rel); i
           ++) {
                /* This is where to make the change */
                loc = (void *)sechdrs[sechdrs[relsec].sh_info].
                   sh_addr
                        + rel[i].r_offset;

                /* This is the symbol it is referring to.  Note
                   that all
                   undefined symbols have been resolved.  */
                sym = (Elf64_Sym *)sechdrs[symindex].sh_addr
                        + ELF64_R_SYM(rel[i].r_info);

                DEBUGP("type␣%d␣st_value␣%Lx␣r_addend␣%Lx␣loc␣%Lx\
                   n",
                        (int)ELF64_R_TYPE(rel[i].r_info),
                        sym->st_value, rel[i].r_addend, (u64)loc);
```

```
                val = sym->st_value + rel[i].r_addend;

                switch (ELF64_R_TYPE(rel[i].r_info)) {
                case R_X86_64_NONE:
                        break;
                case R_X86_64_64:
                        *(u64 *)loc = val;
                        break;
                case R_X86_64_32:
                        *(u32 *)loc = val;
                        if (val != *(u32 *)loc)
                                goto overflow;
                        break;
                case R_X86_64_32S:
                        *(s32 *)loc = val;
                        if ((s64)val != *(s32 *)loc)
                                goto overflow;
                        break;
                case R_X86_64_PC32:
                        val -= (u64)loc;
                        *(u32 *)loc = val;
#if 0
                        if ((s64)val != *(s32 *)loc)
                                goto overflow;
#endif
                        break;
                default:
                        printk(KERN_ERR "module␣%s:␣Unknown␣rela␣
                           relocation:␣%llu\n",
                               me->name, ELF64_R_TYPE(rel[i].
                                  r_info));
                        return -ENOEXEC;
                }
        }
        return 0;

overflow:
```

```
        printk(KERN_ERR "overflow␣in␣relocation␣type␣%d␣val␣%Lx\n"
           ,
               (int)ELF64_R_TYPE(rel[i].r_info), val);
        printk(KERN_ERR "`%s'␣likely␣not␣compiled␣with␣-mcmodel=
           kernel\n",
               me->name);
        return -ENOEXEC;
}
```

前面提到 32 位的版本需要重新修改，这里一并将 32 位的重定位代码贴出，可以看到如果需要移植的话是很容易的。

```
int apply_relocate(Elf32_Shdr *sechdrs,
                   const char *strtab,
                   unsigned int symindex,
                   unsigned int relsec,
                   struct module *me)
{
        unsigned int i;
        Elf32_Rel *rel = (void *)sechdrs[relsec].sh_addr;
        Elf32_Sym *sym;
        uint32_t *location;

        DEBUGP("Applying␣relocate␣section␣%u␣to␣%u\n", relsec,
               sechdrs[relsec].sh_info);
        for (i = 0; i < sechdrs[relsec].sh_size / sizeof(*rel); i
           ++) {
                /* This is where to make the change */
                location = (void *)sechdrs[sechdrs[relsec].sh_info
                   ].sh_addr
                        + rel[i].r_offset;
                /* This is the symbol it is referring to.  Note
                   that all
                   undefined symbols have been resolved.  */
                sym = (Elf32_Sym *)sechdrs[symindex].sh_addr
                        + ELF32_R_SYM(rel[i].r_info);

                switch (ELF32_R_TYPE(rel[i].r_info)) {
                case R_386_32:
```

```
                        /* We add the value into the location
                           given */
                        *location += sym->st_value;
                        break;
                case R_386_PC32:
                        /* Add the value, subtract its postition
                           */
                        *location += sym->st_value - (uint32_t)
                           location;
                        break;
                default:
                        printk(KERN_ERR "module␣%s:␣Unknown␣
                           relocation:␣%u\n",
                            me->name, ELF32_R_TYPE(rel[i].
                               r_info));
                        return -ENOEXEC;
                }
        }
        return 0;
}
```

比较可以知道，32 位的重定位比 64 位的更加简单。事实上，64 位重定位类型中，有的需要将计算结果截断成 32 位，这就要求内核模块的所有符号都在地址

`0xffffffff00000000-0xffffffffffffffff`

否则，在重定位时如果发生 32 位截断，重定位后的地址错误，引起 Page Fault，这是 64 位特有的，32 位系统不受此限制。

4 文件系统模块化

到此为止可加载内核模块的支持已经基本完成，为了说明有效性和展示要求，我以文件系统为例，将 SFAT 文件系统做成模块并成功让运行时的系统加载文件系统模块、挂载分区。

4.1 文件系统注册

首先系统必须支持载入文件系统模块后，能知道自己可以支持新的文件系统类型了。我维护一个双向链表，管理一系列 `struct file_system_type` 的结构，每一个代表一种文件系统。

```
struct file_system_type
{
    const char *name;
    int (*mount)(const char *devname);
    list_entry_t file_system_type_link;
};
```

相关 API 如下:

```
void file_system_type_list_init(void);
int register_filesystem(const char *name, int (*mount)(const char
    *devname)); // 注册一个新的文件系
    统
int unregister_filesystem(const char *name); // 取消注册文件系统
int do_mount(const char *devname, const char *fsname); // 系统调用,
    挂载设备
int do_umount(const char *devname); // 系统调用, 取消挂载
```

此外我编写了用户态的程序 `mount` 和 `umount`，挂载与取消挂载设备，使用方法为

```
mount -t filesystem device
umount device
```

例如

```
mount -t sfs disk2
umount disk2
```

4.2 内核符号表导出

其次是导出内核符号到符号表，供模块调用。例如在 `mod.c` 中的 `mod_init()` 函数中添加 `EXPORT(kprintf)` 即可将 `kprintf` 函数导出。可以在该文件中看到我导出的所有内核符号。

4.3 练习：将 SFS 改编成模块

本实验中我将 SFAT 文件系统制作成可加载模块，但是 SFS 同样可以成为模块，事实上在最初我就是将 SFS 制作成模块并测试的。由于 QEMU 最多只支持 3 个硬盘设备，前两个分别交换分区和根分区，只能有一个多余的设备供我们测试，所以在最后的演示中无法演示同时挂载两个文件系统并互传文件。最终代码版本也删去了这一个模块（因为没有意义）。

虽然如此，后来的开发者可以以此作为了解文件系统模块加载机制的实践机会，我在此给出将 SFS 改编成模块的方法。

首先将 `sfs` 目录移到 `kern-module` 目录并修改 Makefile 编译出 mod-sfs.ko 文件。对于代码，只要在 `sfatfs.c` 中添加 `init_module()` 和 `cleanup_module()`，里面调用 `register_filesystem("sfs", sfatfs_mount)` 和 `unregister_filesystem("sfs")` 即可。要修改的地方就这么简单!

5 测试

在提交的代码包中，首先载入 SFAT 模块

```
$ insmod mod-sfatfs
```

其次，挂载设备

```
$ mount -t sfatfs disk2
```

完成，可以到设备下面

```
$ cd disk2:
```

也可以取消挂载

```
$ umount disk2
```

卸载模块

```
$ rmmod mod-sfatfs
```

6 总结与注意事项

本项目实现了可用的可加载内核模块功能，并且可以在文件系统上应用。

前面提到，64 位的重定位要求内核模块的所有符号都在地址

```
0xffffffff00000000-0xffffffffffffffff
```

否则，在重定位时如果发生 32 位截断，重定位后的地址错误，引起 Page Fault，这是 64 位特有的，32 位系统不受此限制。而 ucore 目前的 `kmalloc` 并不满足这一要求，因此我暂时在代码中静态地开辟一段空间用于存放载入的模块。如果 ucore 下一版本中修复问题以后，可以直接切换回 `kmalloc` 方式。

此外，如果需要实现 32 位支持，主要在于 ELF32 和 ELF64 文件格式的区别。就本项目而言，差异只体现在字段的长度，以及重定位的类型。值得一提的是，32 位下不会有 `kmalloc` 的问题，因此可以直接采用 `kmalloc` 方式。