邢森： 浅析Linux kernel的阅读方法

原创：邢森 Linux阅码场 2017-08-15

作者简介： 邢森，Java、Python码农，痴迷于Linux Kernel。自媒体:“写程序的康德”。现就职于上海电信信息网络部任职架构师，主要负责网络相关产品的研发工作。

●‥ 带着课题

分析任何代码都要都要带着课题，如果只是走马观花很难有具体的收获。“课题”可大、可小，大课题有大收获阅读分析时间也比较长。比如“搞清楚Linux是如何收发数据的”，“Linux是如何分配内存的”，这些都是比较大的题目；再比如“IP数据包如何被重组的”这就是比较具体的问题，属于“小课题”。

“大课题”一般是由多个彼此关联的“小课题”组成的，所以最终我们还是会去在内核中挨个寻找某个具体问题的具体答案，然后再回头来看整个问题。“小课题”是指某个具体问题，我们通常需要先找到一个切入点，然后顺藤摸瓜理出一个头绪来。

源代码的分析工具比较简单，一个编辑器（语法加亮）一个快捷的查找工具（比如grep）就可以开始干活了。通过查找工具找到切入点，然后分析代码的逻辑。如果代码量比较大，一般我们会选择一个IDE工具或者专门的代码阅读工具（Source Insight、Understand）来分析、阅读代码。

●‥ 观察数据流向

成熟的代码通常都很复杂，考虑的事情也比较全面，所以一个函数可能有几十行代码。阅读代码的时候我们要把握数据流向，比如我们知道函数的返回值是我们关注的数据，那么我们观察它在哪里执行了赋值语句，这样就可以理出个主脉络来。

●‥ 分析总结

我喜欢用两幅图来表示分析代码之后的收获，函数调用关系图、数据结构图。调用关系可以从宏观上告诉我们整个过程分成哪些步骤，步骤里面分为哪些子步骤；数据结构辅助说明了这些过程涉及到的数据操作。

●…分析实战

代码分析

“万物皆文件（everything is a file）”，Unix/Linux的一条著名的设计哲学。在Unix/Linux中很多硬件设备、进程运行信息、系统状态都被映射成文件系统中的某个文件，这种设计极大的简化了系统模型。

在Linux中每个文件都由“struct file”和“一个int类型的变量——file descriptor（文件描述符）”组成。下面通过分析Kernel代码来剖析file descriptor的分配过程。

我们是要探究file descriptor的分配过程，问题非常明确，切入点也比较好找——什么时候执行fd的分配？答案是执行`open`函数的时候。所以我们通过查找工具定位到`open`系统调用的代码(fs/open.c)

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);

    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        struct file *f = do_filp_open(dfd, tmp, &op);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            fsnotify_open(f);
            fd_install(fd, f);
        }
    }
    putname(tmp);
    return fd;
} « end do_sys_open »

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

分配文件描述符(fd)

分配struct file

分配struct file失败，释放fd

绑定fd和struct file

open函数是do_sys_open的封装

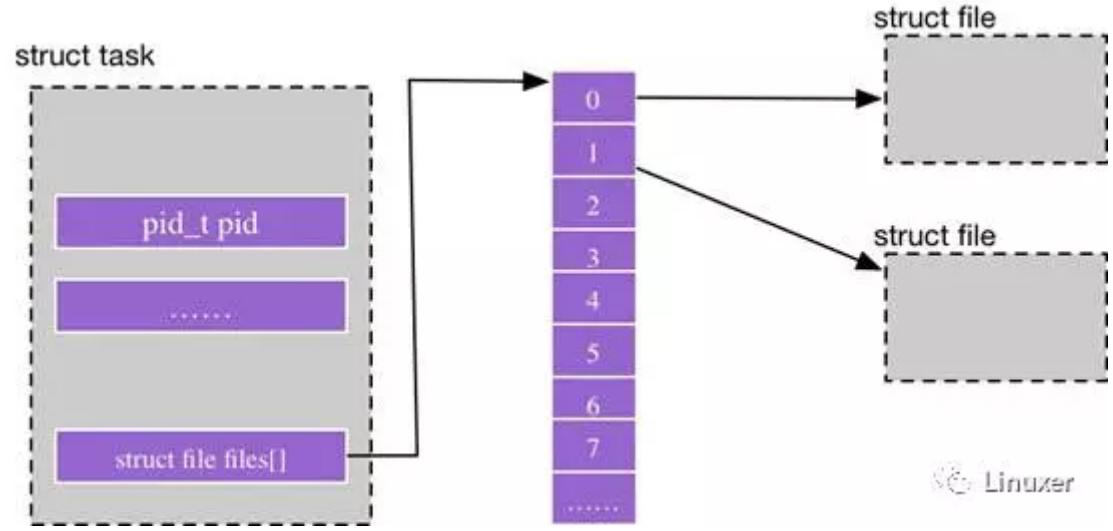
Linux的代码非常清晰，open函数实际上调用的是do_sys_open。“打开文件”的过程分为：

1． 分配文件描述符（fd）；

2． 分配struct file；

3． 绑定fd和structfile。

Linux中资源分配的对象是进程，用struct task表示进程的数据结构。“文件句柄”（内核中指向某个打开文件的指针数据结构是struct file）属于资源申请，所以按道理说Linux的struct task中应该定义一个struct file类型的数组，文件描述符则表示struct file数组中的索引。



实际上Linux2.1之前就是这么干的，但是这种实现方式有一个很明显的缺陷——files的大小是受限的，2.1之前它是一个固定的值——256。如果要突破限制那就不能使用“固定大小数组”的数字定义files，所以在后续的版本中就把“文件句柄”拆分成立两种内核资源——文件描述符（fd）、文件对象（struct file）。（后面会放上我们分析后的数据结构图——也是Linux正在用的数据结构）

回到我们的代码，分配文件描述符的代码是get_unused_fd_flags，我们跟踪下去发现它其实是__alloc_fd函数的封装，直接看__alloc_fd。

```
int __alloc_fd(struct files_struct *files,
               unsigned start, unsigned end, unsigned flags)
{
    unsigned int fd;
    int error;
    struct fdtable *fdt;

    spin_lock(&files->file_lock);
repeat:
    fdt = files_fdtable(files);
    fd = start;
    if (fd < files->next_fd)
        fd = files->next_fd;

    if (fd < fdt->max_fds)
        fd = find_next_fd(fdt, fd);

    /*
     * N.B. For clone tasks sharing a files structure, this test
     * will limit the total number of files that can be opened.
     */
    error = -EMFILE;
    if (fd >= end)
        goto out;

    error = expand_files(files, fd);
    if (error < 0)
        goto out;

    /*
     * If we needed to expand the fs array we
     * might have blocked - try again.
     */
    if (error)
        goto repeat;

    if (start <= files->next_fd)
        files->next_fd = fd + 1;

    __set_open_fd(fd, fdt);
    if (flags & O_CLOEXEC)
        __set_close_on_exec(fd, fdt);
    else
        __clear_close_on_exec(fd, fdt);
    error = fd;
```

next_fd是最小可用fd，这里并不是把next_fd直接使用，而是作为“下标”。

去位图中搜寻可用fd，从“下标”next_fd开始

fdtable当前fd已经用完，则尝试扩充fd

fd分配成功，标记位图为已使用

Linuxer

直接读这么一大段代码很难理清楚头绪，这里有个技巧推荐给大家。直接看它的返回值，它的返回值就是文件描述符，所以我们只要注意在哪里给它赋值就能理出关键头绪。

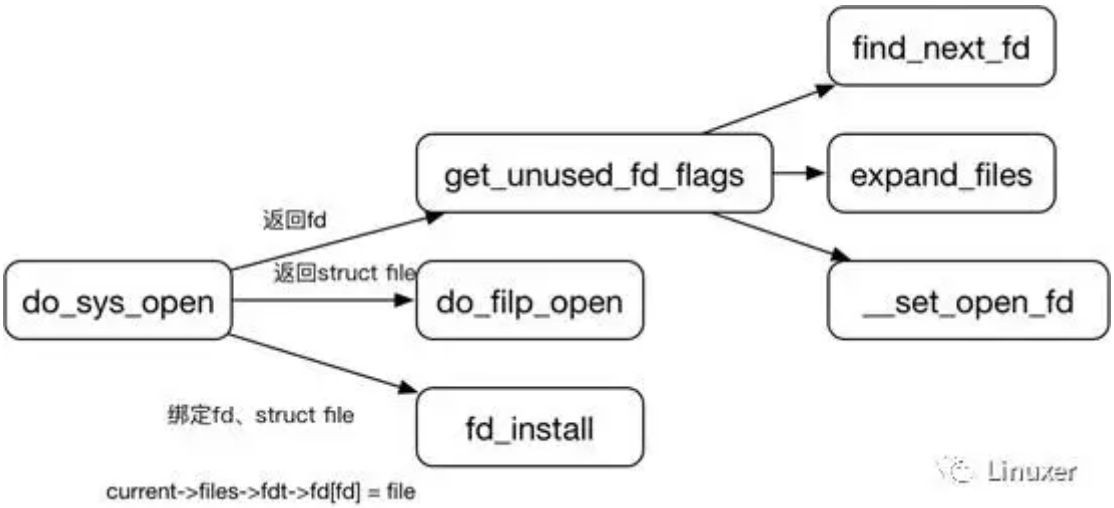
__alloc_fd函数的start，end参数是指文件描述符的**可用**范围，get_unused_fd_flags在传递start参数的时候是0，所以不设置下标范围。Linux用一个位图记录fd的分配状态，需要注意的是next_fd并不能直接作为fd返回，它仅仅是标识“未使用的fd中最小值”，这是为了防止位图中“空隙”（位图中1、2、4、5、6都是空闲的，3已经被使用了，我们搜索未使用fd的时候很显然应该从1开始搜索。所以一定要保存这个“下标”）。

fdtable是内核中用来表示文件描述符表格的数据结构，表示fd分配状态的位图就是它的成员变量（full_fds_bits），max_fds记录的是当前表格可用的最大文件描述符，这个值是可以通过expand_files增加的（如果你打开`expand_files`会发现fd最大值是不能超过`sysctl_nr_open`的，这个就是fs.nr_open的值）。

上面的代码只是寻找可用fd而没有修改位图，所以代码最后通过__set_open_fd来修改位图。

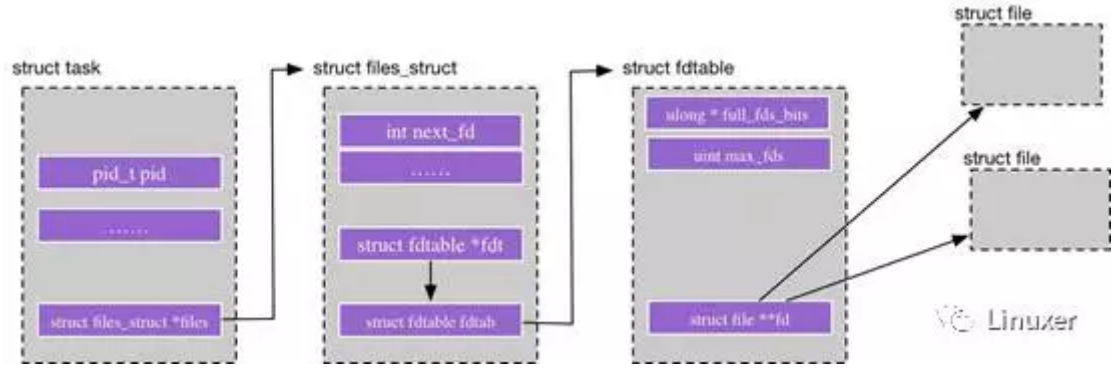
●…总结经验

“一图胜千言”，代码分析是一件非常难“表达出来”的事情，如果是像上面的文字估计没有多少人会有兴趣看。所以我一般分析完代码后画两张图，一张表示数据结构关系的图，一张表示函数调用关系的图。



do_sys_open所做的都是为了最后执行fd_install（成功打开文件），而fd_install可以被简化为一个简单的赋值语句（图中的那句赋值语句）。所以前面的get_unused_fd_flags其实是为了返回合适的fd、do_filp_open则是为struct file分配一块内存空间。

结合数据结构图来看



get_unused_fd_flags的主要操作对象其实就是struct files_struct。

往期精彩回顾

2017.8.14 《深入探究Linux的设备树》的ppt分享

《Linux总线、设备、驱动模型》直播PPT分享

何晔： 当ZYNQ遇到Linux Userspace I/O（UIO）
黄伟亮：ext4文件系统之裸数据的分析实践
陈然： 容器生态系统的发展与演变之我见
徐西宁： 码农小马与Docker不得不说的故事
让天堂的归天堂，让尘土的归尘土——谈Linux的总线、设备、驱动模型
...

快，关注这个公众号，一起涨姿势～