VFS 系统调用的实现

宝德技术研究院 李磊

# 几个重要的数据结构

在介绍文件描述符的具体操作之前,先让我们熟悉下这些重要的数据结构,其中task\_struct属于进程管理中的重要结构,不是本文的重点.

## task\_struct

由于task\_struct结构非常巨大,这里我只列出一些和本文相关的一些成员.

struct task\_struct {

uid\_t uid,euid,suid,fsuid;

gid\_t gid,egid,sgid,fsgid;

struct group\_info \*group\_info;

kernel\_cap\_t cap\_effective, cap\_inheritable, cap\_permitted;

int link\_count;

int total\_link\_count;

struct fs\_struct \*fs;

/\* open file information \*/

struct files\_struct \*files;

/\* namespace \*/

struct namespace \*namespace;

};

这些成员中:

**uid,gid**为运行该进程的用户的用户标识符和组标识符，通常是进程创建者的uid，gid.

**euid，egid**为有效uid,gid

**fsuid，fsgid**为文件系统uid,gid，这两个ID号通常与有效uid,gid相等，在检查对于文件系统的访问权限时使用他们。

**suid，sgid**为备份uid,gid

**cap\_effective**:当一个进程要进行某个特权操作时,操作系统会检查cap\_effective的对应位是否有效,而不再是检查进程的有效UID是否为0.

**cap\_permitted**:表示进程能够使用的能力,在cap\_permitted中可以包含cap\_effective中没有的能力,这些能力是被进程自己临时放弃的,也可以说cap\_effective是cap\_permitted的一个子集.

**cap\_inheritable**:表示能够被当前进程执行的程序继承的能力.

**link\_count**表示嵌套的符号连接数, 超过就放弃搜索(因为可能进入死循环).

**total\_link\_count**为总的符号连接数, 不能超过40.

## files\_struct

struct files\_struct {

/\*

\* read mostly part

\*/

atomic\_t count;

struct fdtable \*fdt;

struct fdtable fdtab;

/\*

\* written part on a separate cache line in SMP

\*/

spinlock\_t file\_lock \_\_\_\_cacheline\_aligned\_in\_smp;

//下一个可用的fd

int next\_fd;

//embedded\_fd\_set是位图,close\_on\_exec\_init记录执行exec

//需要关闭的文件描述符

struct embedded\_fd\_set close\_on\_exec\_init;

//最初的文件描述符集

struct embedded\_fd\_set open\_fds\_init;

//操作文件的数组

struct file \* fd\_array[NR\_OPEN\_DEFAULT];

};

## fdtable

struct fdtable {

//可以打开的最大文件数

unsigned int max\_fds;

//位图的最大长度

int max\_fdset;

//这是一个指针数组, 长度由max\_fds决定,每一个数组项指向一个file结构实例

//管理一个打开文件的所有信息,指向files\_struct的fd\_array

struct file \*\* fd; /\* current fd array \*/

//指向位域的指针, 该位域保存了所有在exec系统调用时需要关闭的文件描述符

//信息

fd\_set \*close\_on\_exec;

//open\_fds指向的bit位最大数目,由max\_fdset决定,指向位域的指针,该位域

//保存所有打开文件的描述符每个可能的文件描述符都对应一个比特位,如果比特位

//为1, 表示文件正在使用中.否则未使用

fd\_set \*open\_fds;

struct rcu\_head rcu;

struct files\_struct \*free\_files;

struct fdtable \*next;

};

## embedded\_fd\_set

struct embedded\_fd\_set {

unsigned long fds\_bits[1];

};

## file

struct file {

/\*

\* fu\_list becomes invalid after file\_free is called and queued

\* via fu\_rcuhead for RCU freeing

\*/

union {

struct list\_head fu\_list;

struct rcu\_head fu\_rcuhead;

} f\_u;

//提供了文件名和inode之间的联系

struct dentry \*f\_dentry;

struct vfsmount \*f\_vfsmnt;

//文件操作调用的各个函数

const struct file\_operations \*f\_op;

//访问本文件的的进程数目的计数器

atomic\_t f\_count;

//open系统调用时传递的额外标志

unsigned int f\_flags;

//打开文件时传递的模式参数(读,写或读写的访问模式)

mode\_t f\_mode;

//文件位置指针

loff\_t f\_pos;

struct fown\_struct f\_owner;

unsigned int f\_uid, f\_gid;

//是否预读数据,如何预读

struct file\_ra\_state f\_ra;

//检查file实例是否与相关inode兼容

unsigned long f\_version;

void \*f\_security;

/\* needed for tty driver, and maybe others \*/

void \*private\_data;

#ifdef CONFIG\_EPOLL

/\* Used by fs/eventpoll.c to link all the hooks to this file \*/

struct list\_head f\_ep\_links;

spinlock\_t f\_ep\_lock;

#endif /\* #ifdef CONFIG\_EPOLL \*/

//指向文件相关inode实例的地址空间映射

struct address\_space \*f\_mapping;

};

# 进程与打开文件之间的关系

进程和打开文件的关系如图1所示,在应用层调用open打开文件的返回值fd就是fd\_array的下标.



图1

# 文件描述符的管理

文件描述符的管理如图2所示,files\_struct指向的数组中最大的数目是由fdtable中的max\_fds决定的,位图的最大长度也由fdtable中的max\_fdset决定.

图2

# files\_struct的初始化

files\_struct的初始化是通过INIT\_FILES完成的.

1 #define INIT\_FDTABLE \

2 { \

3 .max\_fds = NR\_OPEN\_DEFAULT, \

4 .max\_fdset = EMBEDDED\_FD\_SET\_SIZE, \

5 .fd = &init\_files.fd\_array[0], \

6 .close\_on\_exec = (fd\_set \*)&init\_files.close\_on\_exec\_init, \

7 .open\_fds = (fd\_set \*)&init\_files.open\_fds\_init, \

8 .rcu = RCU\_HEAD\_INIT, \

9 .free\_files = NULL, \

10 .next = NULL, \

11 }

12

13 #define INIT\_FILES \

14 { \

15 .count = ATOMIC\_INIT(1), \

16 .fdt = &init\_files.fdtab, \

17 .fdtab = INIT\_FDTABLE, \

18 .file\_lock = \_\_SPIN\_LOCK\_UNLOCKED(init\_task.file\_lock), \

19 .next\_fd = 0, \

20 .close\_on\_exec\_init = { { 0, } }, \

21 .open\_fds\_init = { { 0, } }, \

22 .fd\_array = { NULL, } \

23 }

struct embedded\_fd\_set {

unsigned long fds\_bits[1];

};

/\*

\* More than this number of fds: we use a separately allocated fd\_set

\*/

#define EMBEDDED\_FD\_SET\_SIZE (BITS\_PER\_BYTE \* sizeof(struct

embedded\_fd\_set))

files\_struct中的fdtab的初始状态可以查看INIT\_FDTABLE,其中max\_fds表示最大的文件数, 初始值为NR\_OPEN\_DEFAULT, 也就是BITS\_PER\_LONG, 在32位机上是32,64位机上是64.max\_fdset表示最大的fd数量,初始值为EMBEDDED\_FD\_SET\_SIZE, 也就是一个字长的位数, 在32位机上是32,64位机上是64.fdtable中的fd指向了files\_struct的fd\_array,close\_on\_exec指向了close\_on\_exec\_init, open\_fds指向了open\_fds\_init.

# 通过open打开文件的过程

应用程序通过标准库的open实现一个文件的打开过程,该函数返回一个文件描述符.该函数使用了同名的系统调用,调用了fs/open.c中的sys\_open函数,该函数的大体流程如图3所示.



图3

force\_o\_large检查是否应该设置O\_LARGEFILE标志,如果处理器的字长不是32位(即64位),就应该设置这个标志.下面将分别分析在这过程中使用到的函数.

## do\_sys\_open

1 long do\_sys\_open(int dfd, const char \_\_user \*filename, int flags, int mode)

2 {

3 char \*tmp = getname(filename);

4 int fd = PTR\_ERR(tmp);

5

6 if (!IS\_ERR(tmp)) {

7 //找到一个未使用的fd

8 fd = get\_unused\_fd();

9 if (fd >= 0) {

10 struct file \*f = do\_filp\_open(dfd, tmp, flags, mode);

11 if (IS\_ERR(f)) {

12 put\_unused\_fd(fd);

13 fd = PTR\_ERR(f);

14 } else {

15 fsnotify\_open(f->f\_dentry);

16 fd\_install(fd, f);

17 }

18 }

19 putname(tmp);

20 }

21 return fd;

22 }

函数第3行通过getname从用户空间获取文件名.如果tmp指针有错,即获取用户空间文件名失败,就将tmp指针代表的错误码赋值给fd, 直接将错误码返回.第8行,如果获取文件名成功,就调用get\_unused\_fd获取一个未使用的fd.如果fd合法,就调用do\_filp\_open找到文件名所属的file结构.然后就在16行用fd\_install将fd和通过do\_filp\_open找到的file结构关联起来.

## get\_unused\_fd

1 int get\_unused\_fd(void)

2 {

3 struct files\_struct \* files = current->files;

4 int fd, error;

5 struct fdtable \*fdt;

6

7 error = -EMFILE;

8 spin\_lock(&files->file\_lock);

9

10 repeat:

11 //获取files的fdtable

12 fdt = files\_fdtable(files);

13 //在位图偏移next\_fd的位置开始找第一个为0的bit

14 fd = find\_next\_zero\_bit(fdt->open\_fds->fds\_bits,

15 fdt->max\_fdset,

16 files->next\_fd);

17

18 /\*

19 \* N.B. For clone tasks sharing a files structure, this test

20 \* will limit the total number of files that can be opened.

21 \*/

22 //检查是否超过当前进程限定的最大可打开文件数

23 if (fd >= current->signal->rlim[RLIMIT\_NOFILE].rlim\_cur)

24 goto out;

25

26 /\* Do we need to expand the fd array or fd set? \*/

27 //error < 0错误

28 error = expand\_files(files, fd);

29 if (error < 0)

30 goto out;

31

32 if (error) {

33 /\*

34 \* If we needed to expand the fs array we

35 \* might have blocked - try again.

36 \*/

37 error = -EMFILE;

38 goto repeat;

39 }

40

41 FD\_SET(fd, fdt->open\_fds);

42 FD\_CLR(fd, fdt->close\_on\_exec);

43 files->next\_fd = fd + 1;

44 #if 1

45 /\* Sanity check \*/

46 if (fdt->fd[fd] != NULL) {

47 printk(KERN\_WARNING "get\_unused\_fd: slot %d not NULL!\n", fd);

48 fdt->fd[fd] = NULL;

49 }

50 #endif

51 error = fd;

52

53 out:

54 spin\_unlock(&files->file\_lock);

55 return error;

56 }

函数第3行获取当前进程的files\_struct指针,12行进一步获取了files\_struct指针中的位图信息.然后调用find\_next\_zero\_bit获取了一个未使用的fd.第23行检查fd有没有超过进程打开文件数量的限制.

### find\_next\_zero\_bit

这个函数跟体系架构有关,这里我们具体分析i386构架的.

1 int find\_next\_zero\_bit(const unsigned long \*addr, int size, int

offset)

2 {

3 unsigned long \* p = ((unsigned long \*) addr) + (offset >> 5);

4 int set = 0, bit = offset & 31, res;

5

6 if (bit) {

7 /\*

8 \* Look for zero in the first 32 bits.

9 \*/

10 \_\_asm\_\_("bsfl %1,%0\n\t"

11 //zf = 0 转移

12 "jne 1f\n\t"

13 "movl $32, %0\n"

14 "1:"

15 : "=r" (set)

16 : "r" (~(\*p >> bit)));

17 if (set < (32 - bit))

18 return set + offset;

19 set = 32 - bit;

20 p++;

21 }

22 /\*

23 \* No zero yet, search remaining full bytes for a zero

24 \*/

25 res = find\_first\_zero\_bit (p, size - 32 \* (p - (unsigned long

\*) addr));

26 return (offset + set + res);

27 }

首先我们看一下这个函数需要的参数:

◆addr:这是位图的地址,表示函数搜索开始的位置.

◆size:位图的大小,也就是函数搜索的范围,以bit为单位

◆offset:位图中的偏移,即函数从位图内什么位置开始搜索

由于函数中用到了汇编,这里简单介绍下用到的这两个汇编指令.

◆bsfl:正向位扫描(bit scan forward)

格式: bsf reg, ops

功能: 从0位开始扫描ops各位，若均为0，则zf=1，否则将第一个为1的位置值→reg，zf=0.

◆jne:不相等转移(jump if not equal)

格式: jne 标号

功能: 当zf=0时,就跳转到标号处.

接下来,开始分析代码.第3行,找到离offset最近的那个unsigned long型指针并赋值给p.bit初始化为offset在一个unsigned long中的bit号.如图4所示.



图4

在函数中的一段内嵌汇编中,%0代表c语言中的set变量,%1代表(~(\*p >> bit)),举个例子,这里假设\*p = 10,bit = 3,%1就是0xffff fffe了.第10行的就可以表示为 bsfl 0xffff fffe,set.这里要注意的是kernel中的内嵌汇编一般是AT&T汇编,不是80x86的汇编语言, AT&T和80x86汇编的源操作数和目标操作数相反.故第10行的意思为在0xffff fffe这个数中找出第一个为1的bit, 并把这个bit所在的位置传给set变量.很显然,本例中,set就为1了.既然找到了一个这样的bit, 那么寄存器zf就等于0, 程序运行到第12行后就跳出汇编了.很显然第17行成立.set + offset就是我们要找的那一位,函数返回.

这里还要注意的一点是要找的这一个bit不一定查找一个unsigned long就可以完成的,可能当前查找的unsigned long的数据是0xffff ffff, 这就需要将p偏移一个unsigned long的长度,继续调用find\_next\_zero\_bit查找.函数find\_first\_zero\_bit实际上是对find\_next\_zero\_bit的一个封装.

### expand\_files

1 int expand\_files(struct files\_struct \*files, int nr)

2 {

3 int err, expand = 0;

4 struct fdtable \*fdt;

5

6 //通过进程的打开文件列表获得文件描述符位图结构

7 fdt = files\_fdtable(files);

8 if (nr >= fdt->max\_fdset || nr >= fdt->max\_fds) {

9 if (fdt->max\_fdset >= NR\_OPEN ||

10 fdt->max\_fds >= NR\_OPEN || nr >= NR\_OPEN) {

11 err = -EMFILE;

12 goto out;

13 }

14 expand = 1;

15 if ((err = expand\_fdtable(files, nr)))

16 goto out;

17 }

18 err = expand;

19 out:

20 return err;

21 }

这个函数的实现相对简单,第8行的判断如果为真,说明需要重新分配fdtable.接着在15行调用expand\_fdtable进行fdtable的重新分配,并将files的将分配好的fdtable的fdt指向新fdtable.返回是否分配(0表示未分配新fdtable,1表示分配了fdtable,<0表示分配错误).

### expand\_fdtable

1 static int expand\_fdtable(struct files\_struct \*files, int nr)

2 \_\_releases(files->file\_lock)

3 \_\_acquires(files->file\_lock)

4 {

5 int error = 0;

6 struct fdtable \*fdt;

7 struct fdtable \*nfdt = NULL;

8

9 spin\_unlock(&files->file\_lock);

10 nfdt = alloc\_fdtable(nr);

11 if (!nfdt) {

12 error = -ENOMEM;

13 spin\_lock(&files->file\_lock);

14 goto out;

15 }

16

17 spin\_lock(&files->file\_lock);

18 fdt = files\_fdtable(files);

19 /\*

20 \* Check again since another task may have expanded the

21 \* fd table while we dropped the lock

22 \*/

23 //这里保证nr不小于 fdt->max\_fds和fdt->max\_fdset

24 if (nr >= fdt->max\_fds || nr >= fdt->max\_fdset) {

25 copy\_fdtable(nfdt, fdt);

26 } else {

27 /\* Somebody expanded while we dropped file\_lock \*/

28 spin\_unlock(&files->file\_lock);

29 \_\_free\_fdtable(nfdt);

30 spin\_lock(&files->file\_lock);

31 goto out;

32 }

33 rcu\_assign\_pointer(files->fdt, nfdt);

34 free\_fdtable(fdt);

35 out:

36 return error;

37 }

第10行通过alloc\_fdtable获取重新分配的fdtable.18行获取当前file正在使用的fdtable.若24行成立说明有必要使用扩大后的fdtable,需要调用copy\_fdtable将旧的fdtable中的数据拷贝到新的fdtable中.33行将file的fdt指向新的fdtable.

### alloc\_fdtable

1 static struct fdtable \*alloc\_fdtable(int nr)

2 {

3 struct fdtable \*fdt = NULL;

4 int nfds = 0;

5 fd\_set \*new\_openset = NULL, \*new\_execset = NULL;

6 struct file \*\*new\_fds;

7 fdt = kzalloc(sizeof(\*fdt), GFP\_KERNEL);

8 if (!fdt)

9 goto out;

10

11 nfds = max\_t(int, 8 \* L1\_CACHE\_BYTES, roundup\_pow\_of\_two(

nr + 1));

12 if (nfds > NR\_OPEN)

13 nfds = NR\_OPEN;

14

15 new\_openset = alloc\_fdset(nfds);

16 new\_execset = alloc\_fdset(nfds);

17 if (!new\_openset || !new\_execset)

18 goto out;

19 fdt->open\_fds = new\_openset;

20 fdt->close\_on\_exec = new\_execset;

21 fdt->max\_fdset = nfds;

22

23 nfds = NR\_OPEN\_DEFAULT;

24 /\*

25 \* Expand to the max in easy steps, and keep expanding it until

26 \* we have enough for the requested fd array size.

27 \*/

28 do {

29 #if NR\_OPEN\_DEFAULT < 256

30 if (nfds < 256)

31 nfds = 256;

32 else

33 #endif

34 if (nfds < (PAGE\_SIZE / sizeof(struct file \*)))

35 nfds = PAGE\_SIZE / sizeof(struct file \*);

36 else {

37 nfds = nfds \* 2;

38 if (nfds > NR\_OPEN)

39 nfds = NR\_OPEN;

40 }

41 } while (nfds <= nr);

42 new\_fds = alloc\_fd\_array(nfds);

43 if (!new\_fds)

44 goto out2;

45 fdt->fd = new\_fds;

46 fdt->max\_fds = nfds;

47 fdt->free\_files = NULL;

48 return fdt;

49 out2:

50 nfds = fdt->max\_fdset;

51 out:

52 if (new\_openset)

53 free\_fdset(new\_openset, nfds);

54 if (new\_execset)

55 free\_fdset(new\_execset, nfds);

56 kfree(fdt);

57 return NULL;

58 }

参数:

◆nr:文件描述符

这个函数的作用是分配一个fdtable,并给其中的一些成员变量赋值.第7行,通过kmalloc分配fdt.第8行计算nfds对于x86的构架显然多余,因为8\*L1\_CACHE\_BYTES = 1024, 而fd是不能超过1024的, 这样就没必要再和roundup\_pow\_of\_two(nr+1)比较了.15,16行分配new\_openset和new\_execset,这两个变量是fd\_set指针,一个fd\_set占用128字节, alloc\_fdset刚好也分配128字节(1024bit/8),数组new\_openset, new\_execset内实际上也就一个fd\_set成员.19-21行就对fdt的成员open\_fds,close\_on\_exec,max\_fdset赋予新值.位图相关的成员处理完成后,接着就要处理文件对象相关的成员了.23行给nfds一个初始值.如果nfds <= nr的话(这里一定会成立,如果不成立根本不会调用alloc\_fdtable分配fdtable)就增加nfds.接着调用alloc\_fd\_array根据nfds分配new\_fds.最后fdt的fd(这个fd实际上指向的是files\_struct的fd\_array)就是new\_fds.

### 验证位图对fd分配的作用

为了验证fdtable中的位图open\_fds对文件描述符的作用,笔者对代码做了如下修改,其中”+”代表是增加的部分,由于改动的这部分函数会被调用多次,笔者加了一个全局变量用于控制输出.当open的是”/tmp/rocklee”时,才会有加入的debug信息.

1 diff --git a/fs/open.c b/fs/open.c

2 index 0b946eb..3d34f6c 100755

3 --- a/fs/open.c

4 +++ b/fs/open.c

5 @@ -30,7 +30,7 @@

6 #include <linux/audit.h>

7

8 #include <asm/unistd.h>

9 -

10 +int rocklee\_flag = 0;

11 int vfs\_statfs(struct dentry \*dentry, struct kstatfs \*buf)

12 {

13 int retval = -ENODEV;

14 @@ -984,6 +984,8 @@ int get\_unused\_fd(void)

15 struct files\_struct \* files = current->files;

16 int fd, error;

17 struct fdtable \*fdt;

18 + int i;

19 + unsigned long test = 0xffffffffffffffff;

20

21 error = -EMFILE;

22 spin\_lock(&files->file\_lock);

23 @@ -992,6 +994,21 @@ repeat:

24 //获取files的fdtable

25 fdt = files\_fdtable(files);

26 //在位图偏移next\_fd的位置开始找第一个为0的bit

27 + if(rocklee\_flag == 1){

28 + printk("xspace, fdt->max\_fdset is %d\n",fdt->max\_fdset);

29 +

30 + for(i = 0; i < fdt->max\_fdset / (8 \* sizeof(unsigned long));i++){

31 + printk("xspace, fdt->open\_fds->fds\_bits[%d] is

0x%lx\n", i, fdt->open\_fds->fds\_bits[i]);

32 + }

33 +

34 + fdt->open\_fds->fds\_bits[0] = test;

35 + printk("###############after change###############\n");

36 +

37 + for(i = 0; i < fdt->max\_fdset / (8 \* sizeof(unsigned

long));i++){

38 + printk("xspace, fdt->open\_fds->fds\_bits[%d] is

0x%lx\n", i, fdt->open\_fds->fds\_bits[i]);

39 + }

40 + }

41 +

42 fd = find\_next\_zero\_bit(fdt->open\_fds->fds\_bits,

43 fdt->max\_fdset,

44 files->next\_fd);

45 @@ -1087,9 +1104,19 @@ long do\_sys\_open(int dfd, const char \_\_user

\*filename, int flags, int mode)

46 char \*tmp = getname(filename);

47 int fd = PTR\_ERR(tmp);

48

49 + if(strcmp(tmp, "/tmp/rocklee") == 0){

50 + rocklee\_flag = 1;

51 + }else{

52 + rocklee\_flag = 0;

53 + }

54 +

55 if (!IS\_ERR(tmp)) {

56 //找到一个未使用的fd

57 fd = get\_unused\_fd();

58 +

59 + if(rocklee\_flag == 1)

60 + printk("xspace, fd is %d\n", fd);

61 +

62 if (fd >= 0) {

63 struct file \*f = do\_filp\_open(dfd, tmp, flags, mode);

64 if (IS\_ERR(f)) {

相应的应用层代码:

1 #include <stdio.h>

2 #include <sys/stat.h>

3 #include <sys/types.h>

4 #include <fcntl.h>

5

6 #define size 2048

7

8 int main(void)

9 {

10 int fd[size];

11 fd[0] = open("/tmp/rocklee",O\_RDWR);

12 if(fd[0] < 0){

13 printf("open failed!\n");

14 return 0;

15 }

16

17 printf("app fd is %d\n", fd[0]);

18

19 close(fd[0]);

20

21 return 0;

22 }

当执行应用程序打开”/tmp/rocklee”时,就会有如下内核输出:

xspace, fdt->max\_fdset is 1024

xspace, fdt->open\_fds->fds\_bits[0] is 0x7

xspace, fdt->open\_fds->fds\_bits[1] is 0x0

xspace, fdt->open\_fds->fds\_bits[2] is 0x0

xspace, fdt->open\_fds->fds\_bits[3] is 0x0

xspace, fdt->open\_fds->fds\_bits[4] is 0x0

xspace, fdt->open\_fds->fds\_bits[5] is 0x0

xspace, fdt->open\_fds->fds\_bits[6] is 0x0

xspace, fdt->open\_fds->fds\_bits[7] is 0x0

xspace, fdt->open\_fds->fds\_bits[8] is 0x0

xspace, fdt->open\_fds->fds\_bits[9] is 0x0

xspace, fdt->open\_fds->fds\_bits[10] is 0x0

xspace, fdt->open\_fds->fds\_bits[11] is 0x0

xspace, fdt->open\_fds->fds\_bits[12] is 0x0

xspace, fdt->open\_fds->fds\_bits[13] is 0x0

xspace, fdt->open\_fds->fds\_bits[14] is 0x0

xspace, fdt->open\_fds->fds\_bits[15] is 0x0

###############after change###############

xspace, fdt->open\_fds->fds\_bits[0] is 0xffffffffffffffff

xspace, fdt->open\_fds->fds\_bits[1] is 0x0

xspace, fdt->open\_fds->fds\_bits[2] is 0x0

xspace, fdt->open\_fds->fds\_bits[3] is 0x0

xspace, fdt->open\_fds->fds\_bits[4] is 0x0

xspace, fdt->open\_fds->fds\_bits[5] is 0x0

xspace, fdt->open\_fds->fds\_bits[6] is 0x0

xspace, fdt->open\_fds->fds\_bits[7] is 0x0

xspace, fdt->open\_fds->fds\_bits[8] is 0x0

xspace, fdt->open\_fds->fds\_bits[9] is 0x0

xspace, fdt->open\_fds->fds\_bits[10] is 0x0

xspace, fdt->open\_fds->fds\_bits[11] is 0x0

xspace, fdt->open\_fds->fds\_bits[12] is 0x0

xspace, fdt->open\_fds->fds\_bits[13] is 0x0

xspace, fdt->open\_fds->fds\_bits[14] is 0x0

xspace, fdt->open\_fds->fds\_bits[15] is 0x0

xspace, fd is 64

下面是应用程序输出:

[root@localhost ~]# ./app

app fd is 64

在没添加以上那段debug的代码前,应用程序输出的fd是3,前3个fd默认被用做标准输入,标准输出,错误输出.添加debug代码后,即修改fdt->open\_fds->fds\_bits,使数组第0号元素全部为1,fd的分配从数组1号元素开始分配,得到的fd就是64了.

## do\_filp\_open

1 static struct file \*do\_filp\_open(int dfd, const char \*filename,

2 int flags,int mode)

3 {

4 int namei\_flags, error;

5 struct nameidata nd;

6

7 namei\_flags = flags;

8 if ((namei\_flags+1) & O\_ACCMODE)

9 namei\_flags++;

10

11 error = open\_namei(dfd, filename, namei\_flags, mode, &nd);

12 if (!error)

13 return nameidata\_to\_filp(&nd, flags);

14

15 return ERR\_PTR(error);

16 }

这个函数主要是通过文件名,获取相应的file指针.

参数:

◆dfd:用于路径查找的标志

◆filename:文件名

◆flags:inode中用的标志,表示文件类型,访问权限,打开方式等

◆mode:open是的权限

第8行加1是由于open\_flag的低两位的含义和该函数内部的flag变量中的是不同的. fcntl.h中O\_RDONLY为0,O\_WRONLY为1.而在fs.h中FMODE\_READ是1,FMODE\_WRITE是2. open\_namei用于获取目标文件的nd, nameidata\_to\_filp将nd转化成file指针.

### open\_namei

1 int open\_namei(int dfd, const char \*pathname, int flag,

2 int mode, struct nameidata \*nd)

3 {

4 int acc\_mode, error;

5 struct path path;

6 struct dentry \*dir;

7 int count = 0;

8

9 acc\_mode = ACC\_MODE(flag);

10

11 /\*O\_TRUNC implies we need access checks for write permissions\*/

12 if (flag & O\_TRUNC)

13 acc\_mode |= MAY\_WRITE;

14

15 /\* Allow the LSM permission hook to distinguish append

16 access from general write access. \*/

17 if (flag & O\_APPEND)

18 acc\_mode |= MAY\_APPEND;

19

20 /\*

21 \* The simplest case - just a plain lookup.

22 \*/

23

24 if (!(flag & O\_CREAT)) {

25 error = path\_lookup\_open(dfd, pathname,

26 lookup\_flags(flag), nd, flag);

27

28 if (error)

29 return error;

30 goto ok;

31 }

32

33 /\*

34 \* Create - we need to know the parent.

35 \*/

36 error = path\_lookup\_create(dfd,pathname,LOOKUP\_PARENT, nd, flag, mode);

37 if (error)

38 return error;

39

40 /\*

41 \* We have the parent and last component. First of all, check

42 \* that we are not asked to creat(2) an obvious directory - that

43 \* will not do.

44 \*/

45 error = -EISDIR;

46

47 if (nd->last\_type != LAST\_NORM || nd->last.name[nd->last. len])

48 goto exit;

49

50 dir = nd->dentry;

51 nd->flags &= ~LOOKUP\_PARENT;

52 mutex\_lock(&dir->d\_inode->i\_mutex);

53 path.dentry = lookup\_hash(nd);

54 path.mnt = nd->mnt;

55

56 do\_last:

57 error = PTR\_ERR(path.dentry);

58 if (IS\_ERR(path.dentry)) {

59 mutex\_unlock(&dir->d\_inode->i\_mutex);

60 goto exit;

61 }

62

63 if (IS\_ERR(nd->intent.open.file)) {

64 mutex\_unlock(&dir->d\_inode->i\_mutex);

65 error = PTR\_ERR(nd->intent.open.file);

66 goto exit\_dput;

67 }

68

69 /\* Negative dentry, just create the file \*/

70 if (!path.dentry->d\_inode) {

71 if (!IS\_POSIXACL(dir->d\_inode))

72 mode &= ~current->fs->umask;

73

74 error = vfs\_create(dir->d\_inode, path.dentry, mode, nd);

75 mutex\_unlock(&dir->d\_inode->i\_mutex);

76 dput(nd->dentry);

77 nd->dentry = path.dentry;

78 if (error)

79 goto exit;

80 /\* Don't check for write permission, don't truncate \*/

81 acc\_mode = 0;

82 flag &= ~O\_TRUNC;

83 goto ok;

84 }

85

86 /\*

87 \* It already exists.

88 \*/

89 mutex\_unlock(&dir->d\_inode->i\_mutex);

90 audit\_inode\_update(path.dentry->d\_inode);

91

92 error = -EEXIST;

93 if (flag & O\_EXCL)

94 goto exit\_dput;

95

96 if (\_\_follow\_mount(&path)) {

97 error = -ELOOP;

98 if (flag & O\_NOFOLLOW)

99 goto exit\_dput;

100 }

101

102 error = -ENOENT;

103 if (!path.dentry->d\_inode)

104 goto exit\_dput;

105

106 if (path.dentry->d\_inode->i\_op && path.dentry->d\_inode ->i\_op->follow\_link)

107 goto do\_link;

108

109 path\_to\_nameidata(&path, nd);

110 error = -EISDIR;

111

112 if (path.dentry->d\_inode && S\_ISDIR(path.dentry->d\_inode ->i\_mode))

113 goto exit;

114 ok:

115 error = may\_open(nd, acc\_mode, flag);

116

117 if (error)

118 goto exit;

119 return 0;

120

121 exit\_dput:

122 dput\_path(&path, nd);

123 exit:

124 if (!IS\_ERR(nd->intent.open.file))

125 release\_open\_intent(nd);

126 path\_release(nd);

127 return error;

128 do\_link:

129

130 error = -ELOOP;

131 if (flag & O\_NOFOLLOW)

132 goto exit\_dput;

133 /\*

134 \* This is subtle. Instead of calling do\_follow\_link() we do

135 \*the thing by hands. The reason is that this way we have

136 \* thing by hands. The reason is that this way we have zero

137 \* link\_count and path\_walk() (called from ->follow\_link)

138 \* honoring LOOKUP\_PARENT. After that we have the parent

139 \* and last component, i.e. we are in the same situation as

140 \* after the first path\_walk().Well, almost - if the last

141 \* component is normal we get its copy stored in nd->last.name

142 \* and we will have to putname() it when we are done. Procfs-like \* symlinks just set LAST\_BIND.

\*/

143 nd->flags |= LOOKUP\_PARENT;

144 error = security\_inode\_follow\_link(path.dentry, nd);

145 if (error)

146 goto exit\_dput;

147 error = \_\_do\_follow\_link(&path, nd);

148 if (error) {

149 /\* Does someone understand code flow here? Or it is only

150 \* me so stupid? Anathema to whoever designed this non-sense

151 \* with "intent.open".

152 \*/

153 release\_open\_intent(nd);

154 return error;

155 }

156 nd->flags &= ~LOOKUP\_PARENT;

157 if (nd->last\_type == LAST\_BIND)

158 goto ok;

159 error = -EISDIR;

160 if (nd->last\_type != LAST\_NORM)

161 goto exit;

162 if (nd->last.name[nd->last.len])

163 \_\_putname(nd->last.name);

164 goto exit;

165 }

166 error = -ELOOP;

167 if (count++==32) {

168 \_\_putname(nd->last.name);

169 goto exit;

170 }

171 dir = nd->dentry;

172 mutex\_lock(&dir->d\_inode->i\_mutex);

173 path.dentry = lookup\_hash(nd);

174 path.mnt = nd->mnt;

175 \_\_putname(nd->last.name);

176 goto do\_last;

177 }

这个函数很长,并且实现有些复杂,参照图5可以对函数的整体流程有个把握.第9行到18行对访问时flag操作.24行到31行对flag进行检查,如果flag中的O\_CREAT未置位,这就是最简单的访问操作,调用path\_lookup\_open填充nd中的intent成员,并且通过路径查询,获取目标文件的dentry和mnt成员,并赋值给nd.

如果flag中的O\_CREAT置位,情况就复杂了.这时候不能获取目标文件的dentry和mnt成员,只能获取其父目录的dentry和mnt成员,这是因为当目标文件不存在时,需要创建目标文件.36行调用path\_lookup\_create获取目标文件父目录的dentry和mnt成员.然后判断nd对目标文件的类型记录,如果目标文件是一个目录或者类型不是LAST\_NORM,这时候就应该退出函数.第53,54行获取目标文件的path结构.其中利用lookup\_hash获取目标文件的dentry时, 如果目标文件的dentry不存在,这个函数就会为目标文件创建一个dentry.57行是对目标文件dentry的正确性检查.63行是对intent正确性的检查.70行,如果目标文件的dentry存在,但是inode不存在,这时候需要调用vfs\_create为目标文件创建一个inode.96行,如果目标文件是文件系统的挂载点,这时目标文件的dentry和mnt成员就需要切换到文件系统的根目录.106行,如果目标文件是一个符号链接,这时就需要跳转到do\_link标记处.在标记do\_link中进行符号链接跟踪调用的是\_\_do\_follow\_link,为什么不调用do\_follow\_link?

经过笔者的分析,如果调用do\_follow\_link就会使nd->depth增加,进入\_\_link\_path\_walk后就会一直跟踪符号链接,直到获取目标文件.这与我们的初衷是相违背的, O\_CREAT置位的情况下我们应该仅仅获取父目录的dentry和mnt成员,当目标文件不存在时还需要创建目标文件.这样的话我们就不能调用do\_follow\_link,只能调用\_\_do\_follow\_link获取目标文件指向的路径,然后调用\_\_link\_path\_walk获取这个指向路径的最后分量的父目录.将最后分量的dentry和mnt传给nd,然后跳转到do\_last标签处继续对nd的子文件检查,如果这个子文件还是符号链接,继续重复上面的流程.在do\_follow\_link中有对符号链接的嵌套个数限制,防止无限嵌套,故在do\_link标签中也要对嵌套的个数进行限制,在167行中对count的判断就是出于此目的.

如果最终的目标文件是一个普通文件,在109行就获取此文件的dentry和mnt成员.接下来判断是否能打开,如果能,就返回成功了.



图5

### path\_lookup\_open

int path\_lookup\_open(int dfd, const char \*name, unsigned int lookup\_flags,

struct nameidata \*nd, int open\_flags)

{

return \_\_path\_lookup\_intent\_open(dfd, name, lookup\_flags, nd,

open\_flags, 0);

}

这个函数上是\_\_path\_lookup\_intent\_open的简单封装.

### \_\_path\_lookup\_intent\_open

1 static int \_\_path\_lookup\_intent\_open(int dfd, const char \*name,

2 unsigned int lookup\_flags, struct nameidata \*nd,

3 int open\_flags, int create\_mode)

4 {

5 struct file \*filp = get\_empty\_filp();

6 int err;

7

8 if (filp == NULL)

9 return -ENFILE;

10 nd->intent.open.file = filp;

11 nd->intent.open.flags = open\_flags;

12 nd->intent.open.create\_mode = create\_mode;

13 err = do\_path\_lookup(dfd, name, lookup\_flags|LOOKUP\_OPEN,

nd);

14 if (IS\_ERR(nd->intent.open.file)) {

15 if (err == 0) {

16 err = PTR\_ERR(nd->intent.open.file);

17 path\_release(nd);

18 }

19 } else if (err != 0)

20 release\_open\_intent(nd);

21 return err;

22 }

这个函数主要是根据文件名及一些标志,获取nameidata对象.

参数:

◆dfd:用于路径查找的标志.

◆name:文件名.

◆lookup\_flags:lookup过程中用到的标志.

◆nd:需要填充的nameidata对象.

◆open\_flags:open时用到的标志.

◆create\_mode:文件创建时用到的标志.

函数首先通过get\_empty\_filp获取一个file指针.第10行至12行填充nd的intent成员.然后通过do\_path\_lookup获取nd中其他的成员. do\_path\_lookup这个函数的分析请见文档--vfs中的路径查询.

### may\_open

int may\_open(struct nameidata \*nd, int acc\_mode, int flag)

2 {

3 struct dentry \*dentry = nd->dentry;

4 struct inode \*inode = dentry->d\_inode;

5 int error;

6

7 if (!inode)

8 return -ENOENT;

9

10 if (S\_ISLNK(inode->i\_mode))

11 return -ELOOP;

12

13 if (S\_ISDIR(inode->i\_mode) && (flag & FMODE\_WRITE))

14 return -EISDIR;

15

16 error = vfs\_permission(nd, acc\_mode);

17 if (error)

18 return error;

19

20 if (S\_ISFIFO(inode->i\_mode) || S\_ISSOCK(inode->i\_mode)) {

21 flag &= ~O\_TRUNC;

22 } else if (S\_ISBLK(inode->i\_mode) || S\_ISCHR(inode->i\_mode)) {

23 if (nd->mnt->mnt\_flags & MNT\_NODEV)

24 return -EACCES;

25

26 flag &= ~O\_TRUNC;

27 } else if (IS\_RDONLY(inode) && (flag & FMODE\_WRITE))

28 return -EROFS;

29

30 if (IS\_APPEND(inode)) {

31 if ((flag & FMODE\_WRITE) && !(flag & O\_APPEND))

32 return -EPERM;

33 if (flag & O\_TRUNC)

34 return -EPERM;

35 }

36

37 if (flag & O\_NOATIME)

38 if (current->fsuid != inode->i\_uid && !capable (CAP\_FOWNER))

39 return -EPERM;

40

41 /\*

42 \* Ensure there are no outstanding leases on the file.

43 \*/

44 error = break\_lease(inode, flag);

45 if (error)

46 return error;

47

48 if (flag & O\_TRUNC) {

49 error = get\_write\_access(inode);

50 if (error)

51 return error;

52

53 error = locks\_verify\_locked(inode);

54 if (!error) {

55 DQUOT\_INIT(inode);

56 error = do\_truncate(dentry, 0, ATTR\_MTIME|ATTR\_CTIME, NULL);

57 }

58 put\_write\_access(inode);

59 if (error)

60 return error;

61 } else

62 if (flag & FMODE\_WRITE)

63 DQUOT\_INIT(inode);

64

65 return 0;

66 }

函数首先判断inode是否存在.然后判断目标文件是否是一个符号链接.在open\_namei中调用may\_open之前,符号链接均被处理完成,故在此处不可能再有符号链接.接着判断目标文件是否是一个目录.在16行对访问权限进行检查.设备文件的内容是不能被清空的,如块设备,socket, fifo,及字符设备.在只读文件系统对文件写入也是非法的.30行,如果一个文件只能以追加的方式写入,如果操作以非追加方式写入就会报错.如果将文件内容清空也是不允许的. break\_lease用于检测是否有其他进程在对本次操作目标进行操作.若open时设置了O\_TRUNC标志,还需要对文件加锁.

这个函数中的锁机制目前笔者不太熟悉,后续将会完善

### nameidata\_to\_filp

1 struct file \*nameidata\_to\_filp(struct nameidata \*nd, int flags)

2 {

3 struct file \*filp;

4

5 /\* Pick up the filp from the open intent \*/

6 filp = nd->intent.open.file;

7 /\* Has the filesystem initialised the file for us? \*/

8 if (filp->f\_dentry == NULL)

9 filp = \_\_dentry\_open(nd->dentry, nd->mnt, flags, filp, NULL);

10 else

11 path\_release(nd);

12 return filp;

13 }

函数第6行获取想要打开的文件的文件指针.然后判断这个文件指针的dentry是否已经被文件系统初始化.如果没有初始化就调用\_\_dentry\_open对文件对象初始化.

### \_\_dentry\_open

1 static struct file \*\_\_dentry\_open(struct dentry \*dentry,

2 struct vfsmount \*mnt,

3 int flags, struct file \*f,

4 int (\*open)(struct inode \*, struct file \*))

5 {

6 struct inode \*inode;

7 int error;

8

9 //系统调用时传入的标志

10 f->f\_flags = flags;

11 f->f\_mode = ((flags+1) & O\_ACCMODE) | FMODE\_LSEEK |

12 FMODE\_PREAD | FMODE\_PWRITE;

13 inode = dentry->d\_inode;

14 //如果用户要求写

15 if (f->f\_mode & FMODE\_WRITE) {

16 //写文件之前要加锁

17 error = get\_write\_access(inode);

18 if (error)

19 goto cleanup\_file;

20 }

21

22 //地址空间对象

23 f->f\_mapping = inode->i\_mapping;

24 f->f\_dentry = dentry;

25 f->f\_vfsmnt = mnt;

26 //文件指针位置

27 f->f\_pos = 0;

28 f->f\_op = fops\_get(inode->i\_fop);

29 //把file加入sb维护的file链表中去

30 file\_move(f, &inode->i\_sb->s\_files);

31

32 //普通文件open为空

33 if (!open && f->f\_op)

34 //使用inode中的i\_fop

35 open = f->f\_op->open;

36 /\*如果是设备文件，需要打开\*/

37 if (open) {

38 error = open(inode, f);

39 if (error)

40 goto cleanup\_all;

41 }

42

43 //这里将所有创建标志全部清空

44 f->f\_flags &= ~(O\_CREAT | O\_EXCL | O\_NOCTTY | O\_TRUNC);

45

46 /\*预读初始化\*/

47 file\_ra\_state\_init(&f->f\_ra, f->f\_mapping->host->i\_mapping

);

48

49 if (f->f\_flags & O\_DIRECT) {

50 if (!f->f\_mapping->a\_ops ||

51 ((!f->f\_mapping->a\_ops->direct\_IO) &&

52 (!f->f\_mapping->a\_ops->get\_xip\_page))) {

53 fput(f);

54 f = ERR\_PTR(-EINVAL);

55 }

56 }

57

58 return f;

59 ...

}

这个函数用nd填充file指针,对象是一个设备文件,则调用设备的open函数.

## fd\_install

1 void fastcall fd\_install(unsigned int fd, struct file \* file)

2 {

3 struct files\_struct \*files = current->files;

4 struct fdtable \*fdt;

5 spin\_lock(&files->file\_lock);

6 fdt = files\_fdtable(files);

7 BUG\_ON(fdt->fd[fd] != NULL);

8 rcu\_assign\_pointer(fdt->fd[fd], file);

9 spin\_unlock(&files->file\_lock);

10 }

这个函数将fd和file指针关联起来,vfs中的open 过程就全部分析完毕.下面分析vfs中的read和write.

## sys\_read

1 asmlinkage ssize\_t sys\_read(unsigned int fd, char \_\_user \* buf,

size\_t count)

2 {

3 struct file \*file;

4 ssize\_t ret = -EBADF;

5 int fput\_needed;

6

7 file = fget\_light(fd, &fput\_needed);

8 if (file) {

9 loff\_t pos = file\_pos\_read(file);

10 ret = vfs\_read(file, buf, count, &pos);

11 file\_pos\_write(file, pos);

12 fput\_light(file, fput\_needed);

13 }

14

15 return ret;

16 }

函数第7行调用fget\_light获取fd对应的file对象.然后调用file\_pos\_read获取此文件对象内的读写指针的位置.第10行是这个函数的核心, 通过vfs\_read,将文件内容读取到buf中.接下来的工作就是写入读写指针的位置.以及对引用引用计数减1.

### fget\_light

1 struct file fastcall \*fget\_light(unsigned int fd, int

\*fput\_needed)

2 {

3 struct file \*file;

4 struct files\_struct \*files = current->files;

5

6 \*fput\_needed = 0;

7 //如果file\_struct只有一个进程在使用, 不需要RCU.

8 //同一个进程多次读取,这里不会自加1

9 if (likely((atomic\_read(&files->count) == 1))) {

10 file = fcheck\_files(files, fd);

11 //如果有多个进程在使用,就加锁访问

12 } else {

13 rcu\_read\_lock();

14 file = fcheck\_files(files, fd);

15 if (file) {

16 //计数器不为0的时候自加,open过程中f\_count初始化为1

17 if (atomic\_inc\_not\_zero(&file->f\_count))

18 \*fput\_needed = 1;

19 else

20 /\* Didn't get the reference, someone's freed \*/

21 file = NULL;

22 }

23 rcu\_read\_unlock();

24 }

25

26 return file;

27 }

第9行,判断file\_struct是否只有当前进程访问.如果没有其他进程访问,就调用fcheck\_files通过fd获取file对象.如果除了当前进程还有其他进程访问这个file\_struct对象.这时就需要使用rcu包含临界区了.rcu在读者多余写者的情况效果非常好,因为读者之间不需要锁机制.第14行也是调用fcheck\_files获取file对象.第17行检查file对象的引用计数,如果file->f\_count不是0, file->f\_count就自加,并且将标志fput\_needed置1,表示在read完之后需要调用fput将引用计数减1.

### fcheck\_files

1 static inline struct file \* fcheck\_files(struct files\_struct

\*files, unsigned int fd)

2 {

3 struct file \* file = NULL;

4 struct fdtable \*fdt = files\_fdtable(files);

5

6 if (fd < fdt->max\_fds)

7 file = rcu\_dereference(fdt->fd[fd]);

8 return file;

9 }

这个函数在第4行获取了fdtable, 然后将fd作为下标,在file指针数组中取出对应的file对象.

### vfs\_read

1 ssize\_t vfs\_read(struct file \*file, char \_\_user \*buf, size\_t count,

loff\_t \*pos)

2 {

3 ssize\_t ret;

4

5 if (!(file->f\_mode & FMODE\_READ))

6 return -EBADF;

7 if (!file->f\_op || (!file->f\_op->read && !file->f\_op->

aio\_read))

8 return -EINVAL;

9 //检查用户空间指针是否可用

10 if (unlikely(!access\_ok(VERIFY\_WRITE, buf, count)))

11 return -EFAULT;

12

13 ret = rw\_verify\_area(READ, file, pos, count);

14 if (ret >= 0) {

15 count = ret;

16 ret = security\_file\_permission (file, MAY\_READ);

17 if (!ret) {

18 if (file->f\_op->read)

19 //调用设备的read函数

20 ret = file->f\_op->read(file, buf, count, pos);

21 else

22 ret = do\_sync\_read(file, buf, count, pos);

23 if (ret > 0) {

24 fsnotify\_access(file->f\_dentry);

25 current->rchar += ret;

26 }

27 current->syscr++;

28 }

29 }

30

31 return ret;

32 }

真正的读取操作就在这个函数中完成的.跳过一些参数检测代码,在函数13行对pos和count进行检查.并处理强制锁的情况.这里先不做讨论.在18行,如果注册了read函数,就调用此回调函数.如果没有,就执行通用的do\_sync\_read.

### sys\_write

和sys\_read几乎完全一样,略