0.前言

本文是基于Linux-6.20内核源码中的FUSE和标准libfuse库的学习，主要从以下几个方面进行了总结：

1.FUSE模块加载

2.mount和open过程

3.对文件write

4.FUSE工作流程

5.FUSE数据结构

对于虚拟文件系统和设备驱动的相关概念本文仅作简要说明。需要说明的是，由于内核的复杂性及个人能力的有限，本文省略了包括内核同步，异常检查在内的诸多内容，希望可以突出重点。

Linux-6.2.0内核源码详见于：https://elixir.bootlin.com/linux/v6.2/source

标准libfuse库的源码详见于：https://libfuse.github.io/doxygen/files.html

1.FUSE模块加载

FUSE内核模块需要在用户空间使用insmod或者modprobe加载。它们通过系统调用init_module启动加载过程，注册过程比较简单，包括如下步骤：

1.创建高速缓存结构fuse_inode_cachep

2.遍历file_systems链表，如果未注册，则将fuseblk_fs_type链到file_systems链表尾部

3.遍历file_systems链表，如果未注册，则将fuse_fs_type链到file_systems链表尾部

4.创建fuse_kobj和connections_kobj两个kobject

5.遍历file_systems链表，如果未注册，则将fuse_ctl_fs_type链到file_systems链表尾部

模块成功加载以后，以下接口被注册：

```
static struct file_system_type fuseblk_fs_type = { //块设备
    .owner    = THIS_MODULE,
    .name     = "fuseblk",
    .mount    = fuse_mount_blk,
    .kill_sb  = fuse_kill_sb_blk,
    .fs_flags = FS_REQUIRES_DEV | FS_HAS_SUBTYPE,
};

 static struct file_system_type fuse_fs_type = {
    .owner    = THIS_MODULE,
    .name     = "fuse",
    .fs_flags = FS_HAS_SUBTYPE,
    .mount    = fuse_mount,
    .kill_sb  = fuse_kill_sb_anon,
};

 const struct file_operations fuse_dev_operations = {
    .owner        = THIS_MODULE,
    .llseek       = no_llseek,
    .read         = do_sync_read,
```

```
    .aio_read     = fuse_dev_read,
    .splice_read  = fuse_dev_splice_read,
    .write        = do_sync_write,
    .aio_write    = fuse_dev_write,
    .splice_write = fuse_dev_splice_write,
    .poll         = fuse_dev_poll,
    .release      = fuse_dev_release,
    .fasync       = fuse_dev_fasync,
};

 static struct miscdevice fuse_miscdevice = {
    .minor = FUSE_MINOR,
    .name  = "fuse",
    .fops  = &fuse_dev_operations,
};
```

2.mount和open过程

FUSE模块加载注册了fuseblk_fs_type和fuse_fs_type两种文件类型，默认情况下使用的是fuse_fs_type即mount 函数指针被初始化为fuse_mount，而fuse_mount实际调用mount_nodev，它主要由如下两步组成：

1.sget(fs_type)搜索文件系统的超级块对象(super_block)链表(type->fs_supers),如果找到一个与块设备相关的超级块，则返回它的地址。否则，分配并初始化一个新的超级块对象，把它插入到文件系统链表和超级块全局链表中，并返回其地址。

2.fill_super(此函数由各文件系统自行定义): 这个函数式各文件系统自行定义的函数，它实际上是fuse_fill_super。一般fill_super会分配索引节点对象和对应的目录项对象, 并填充超级块字段值，另外对于fuse还需要分配fuse_conn,fuse_req。需要说明的是，它在底层调用了fuse_init_file_inode用fuse_file_operations和fuse_file_aops分别初始化inode->i_fop和inode->i_data.a_ops。

```
static const struct file_operations fuse_file_operations = {
    .llseek         = fuse_file_llseek,
    .read           = do_sync_read,
    .aio_read       = fuse_file_aio_read,
    .write          = do_sync_write,
    .aio_write      = fuse_file_aio_write,
    .mmap           = fuse_file_mmap,
    .open           = fuse_open,
    .flush          = fuse_flush,
    .release        = fuse_release,
    .fsync          = fuse_fsync,
    .lock           = fuse_file_lock,
    .flock          = fuse_file_flock,
    .splice_read    = generic_file_splice_read,
    .unlocked_ioctl = fuse_file_ioctl,
    .compat_ioctl   = fuse_file_compat_ioctl,
    .poll           = fuse_file_poll,
    .fallocate      = fuse_file_fallocate,
};

static const struct address_space_operations fuse_file_aops  = {
    .readpage       = fuse_readpage,
    .writepage      = fuse_writepage,
    .launder_page   = fuse_launder_page,
    .readpages      = fuse_readpages,
    .set_page_dirty = __set_page_dirty_nobuffers,
```

```
    .bmap           = fuse_bmap,
    .direct_IO      = fuse_direct_IO,
};
```

open系统调用底层实现相当复杂，它的主要工作是实例化file对象。file->f_op就是在open中被赋值为inode->i_fop，这一过程读者可以在fs/open.c中的do_entry_open函数中找到。如上所述，inode->i_fop已经被fuse_init_file_inode初始化为fuse_file_operations。

至此，普通文件和设备文件的操作接口都已成功初始化。

3.FUSE下write的一般流程

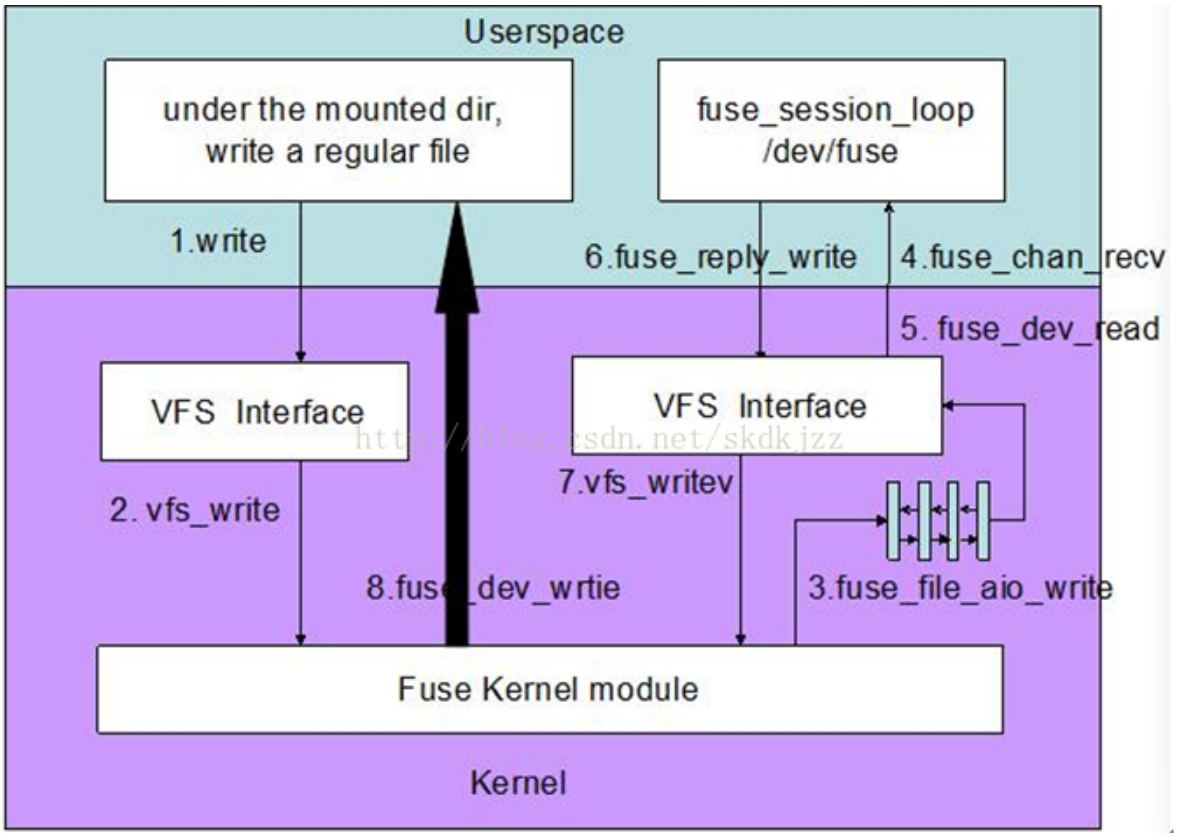

图1

在基于FUSE的用户空间文件系统中执行write操作的流程如图1所示(图中部分函数是缩写，请参考源码)：

1.客户端在mount目录下面，对一个regular file调用write, 这一步是在用户空间执行

2.write内部会调用虚拟文件系统提供的一致性接口vfs_write

3.根据FUSE模块注册的file_operations信息，vfs_write会调用fuse_file_write_iter，将写请求放入fuse connection的request pending queue, 随后进入睡眠等待应用程序reply

4.用户空间的libfuse有一个守护进程通过函数fuse_session_loop轮询杂项设备/dev/fuse, 一旦request queue有请求即通过fuse_kern_chan_receive接收

5.fuse_kern_chan_receive通过read读取request queue中的内容，read系统调用实际上是调用的设备驱动接口fuse_dev_read

6.在用户空间读取并分析数据，执行用户定义的write操作，将状态通过fuse_reply_write返回给kernel

7.fuse_reply_write调用VFS提供的一致性接口vfs_write

8.vfs_write最终调用fuse_dev_write将执行结果返回给第3步中等待在waitq的进程，此进程得到reply后，write返回

4.FUSE工作流程

4.1 FUSE用户空间流程

FUSE在用户空间提供了fuse userspace library和mount /unmount。fuse usespace library提供了一组API供用户开发用户空间文件系统。用户要做的就是实现fuse_operations 或fuse_lowlevel_ops定义的操作, 这两个结构类似于VFS中的struct file_operations。

mount工具fusermount用于挂载用fuse实现的文件系统。

用户在使用fuse的时候有两种开发模式：一种是high-level模式，此模式下fuse的入口函数为fuse_main，它封装了一系列初始化操作，使用简单，但是不灵活。另一种是low-level模式，用户可以利用fuse提供的底层函数灵活开发应用程序。

需要说明的是high-level模式其实是对low-level的封装，因此这里分析lowlevel模式。

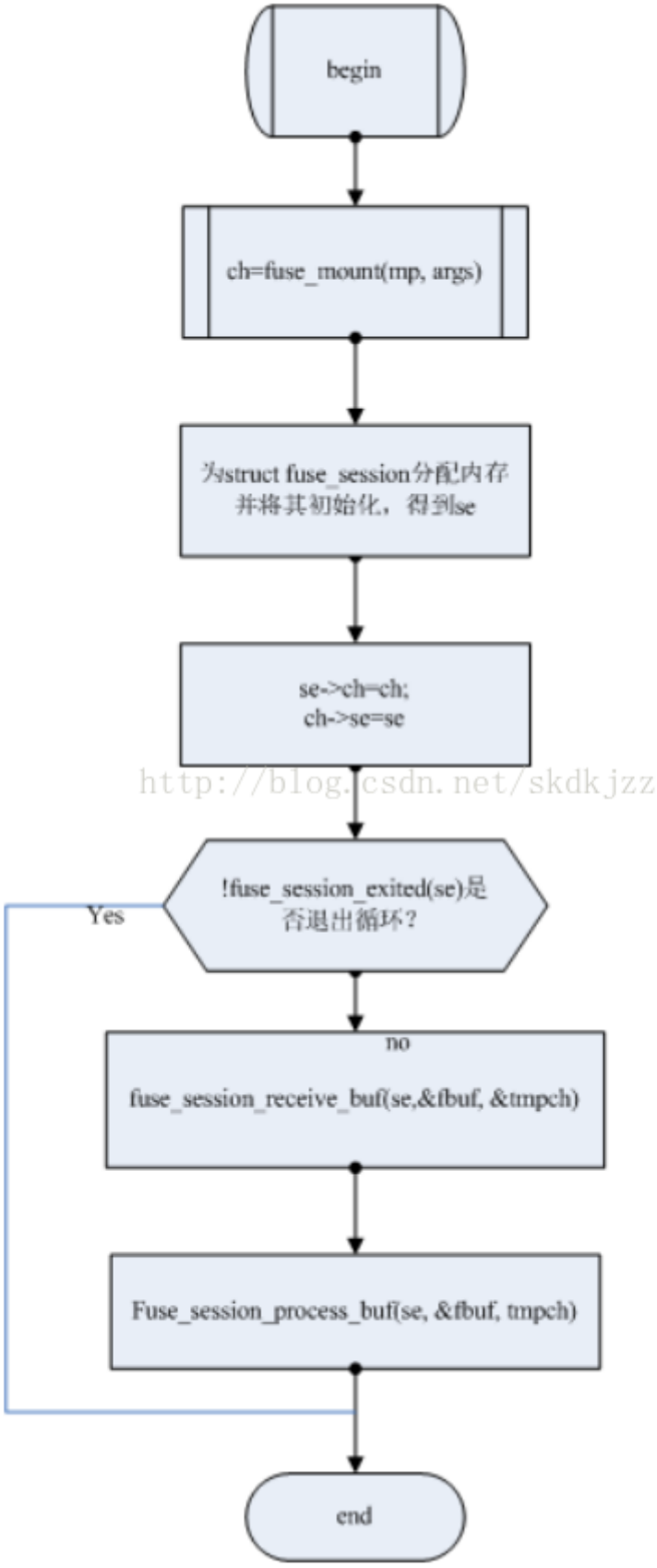

图2

图2展示FUSE在用户空间总体工作流程：

1.调用fuse_mount实例化struct fuse_chan为ch, 将指定目录mount到挂载点

2.实例化struct fuse_session为se，并且将se和ch关联

3.进入循环，从/dev/fuse读取数据，处理以后执行响应的操作

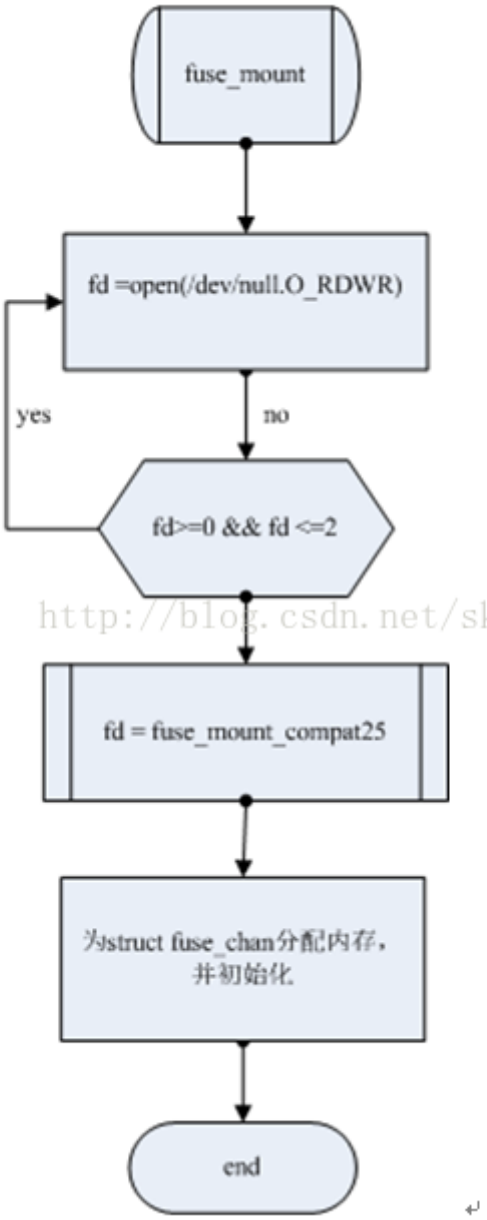

图3

图3展示了fuse_mount函数内部流程：

1. 确保打开的文件描述符至少大于2
2. 分析并检查用户传入的参数
3. 打开/dev/fuse 得到fd，用户空间与内核通过/dev/fuse通信
4. mount源目录到挂载点
5. 用fd实例化struct fuse_chan为ch
6. 返回ch

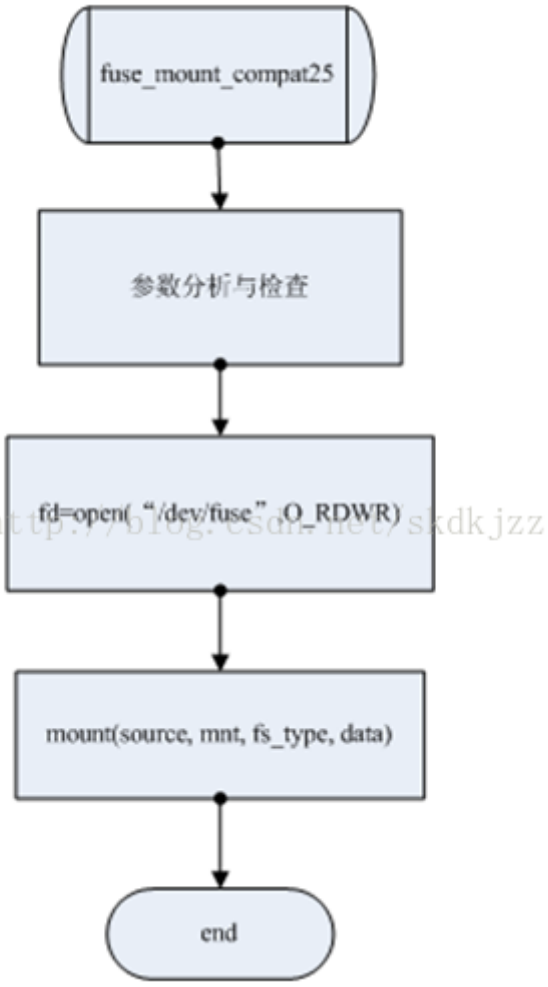

图4

图5展示了fuse_mount_compat25内部细节，进入循环以后，函数fuse_session_receive_buf实际通过fuse_ll_receive_buf从/dev/fuse中读取数据，其通过fbuf返回。

fuse_ll_receive_buf是通过read或者splice系统调用从内核request队列中读取数据。函数fuse_session_process_buf实际通过fuse_ll_process_buf处理数据，fuse_ll_process_buf会根据数据类型最后执行用户定义的操作fuse_ll_ops[in->opcode].func(req, in->nodeid, inarg)。

执行完用户定义的操作以后需要向内核返回执行结果，fuse提供了一组类似fuse_reply_XXX的API, 这些API最后实际通过系统调用writev将结果传入内核。

4.2 FUSE内核部分流程

FUSE在内核空间执行的部分主要包括FUSE模块加载以及杂项设备驱动。模块加载过程已经在第1节介绍，这一节主要描述从request队列读写请求的流程。

FUSE设备驱动程序本质上是一个生产者——消费者模型。生产者为用户在挂载目录下对普通文件(regular file)执行的系统调用，每一次系统调用会产生一个request然后将去放入pending list。pending list能存放的元素个数只和系统内存有关；消费者为用户对设备文件/dev/fuse或者/dev/fuseblk的read，这一操作会去pending list或interrupt list取request，当list为空时，进程主动schedule让出CPU。

request结构的细节在第3节已经介绍，此处不赘述。enmu fuse_req_state定义了request的6种状态，其含义分别为：

FUSE_REQ_INIT：请求被初始化

FUSE_REQ_PENDING：请求挂起待处理

FUSE_REQ_READING：请求正在读

FUSE_REQ_SENT：请求被发送

FUSE_REQ_WRITING：请求正在写

FUSE_REQ_FINISHED：请求已经完成

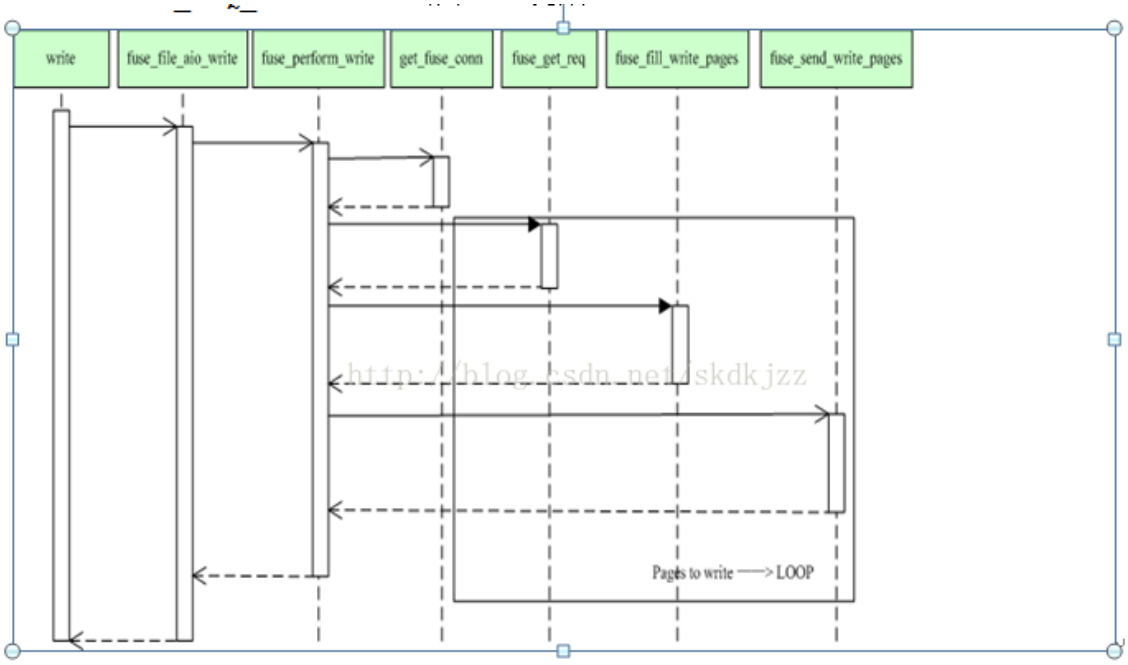

图5

图5是在mount目录下面执行write以后触发的一个函数调用序列，图中省略了VFS层的函数调用。

fuse_file_write_iter是在mount过程中注册到fuse_file_operations.write_iter的函数指针，它会调用fuse_perform_write，fuse_perform_write调用get_fuse_conn得到struct fuse_conn实例fc，它保存在struct super_block的私有数据成员中s_fs_info中，而struct super_block是struct inode的一个成员。

接下来是循环从用户空间拷贝数据到内核，数据实际保存在struct pages中，内核fuse_req保存了pages指针，然后调用fuse_send_write_pages。

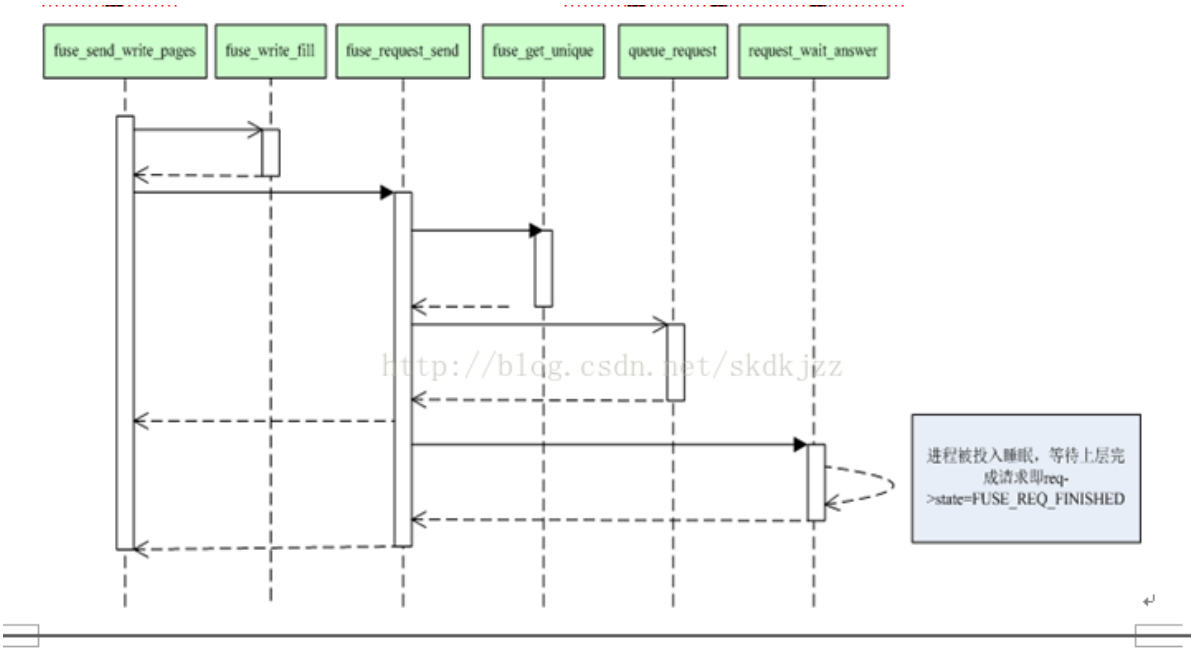

图6

fuse_send_write_pages调用会等待脏数据写回到磁盘上，然后调用fuse_write_fill将包括操作码FUSE_WRITE在内的信息写入request。

随后fuse_request_send(fc, req)，它先通过fuse_get_unique获取唯一请求号，请求号是一个64位无符号整数，请求号从1开始随请求依次递增。然后调用queue_request(fc, req)，它主要完成4件事情：

1.将request->list插入fc维护的pending链表尾部

2.置req->state为FUSE_REQ_PENDING

3.wake_up唤醒等待队列fc->waitq

4.kill_fasync异步通知用户进程数据到达

从queue_request返回以后调用request_wait_answer:进程被投入睡眠，等待请求完成(wait_event(req->state == FUSE_REQ_FINISHED))。

如果用户程序处理完了请求，它会reply，进程被唤醒，到此可以向上层调用返回处理结果(错误码或者写入字节数)。

在第4.1节我们提到了用户空间有个daemon进程会循环read设备文件/fuse/dev以便处理内核请求，图7展示了该read调用触发的函数调用序列。

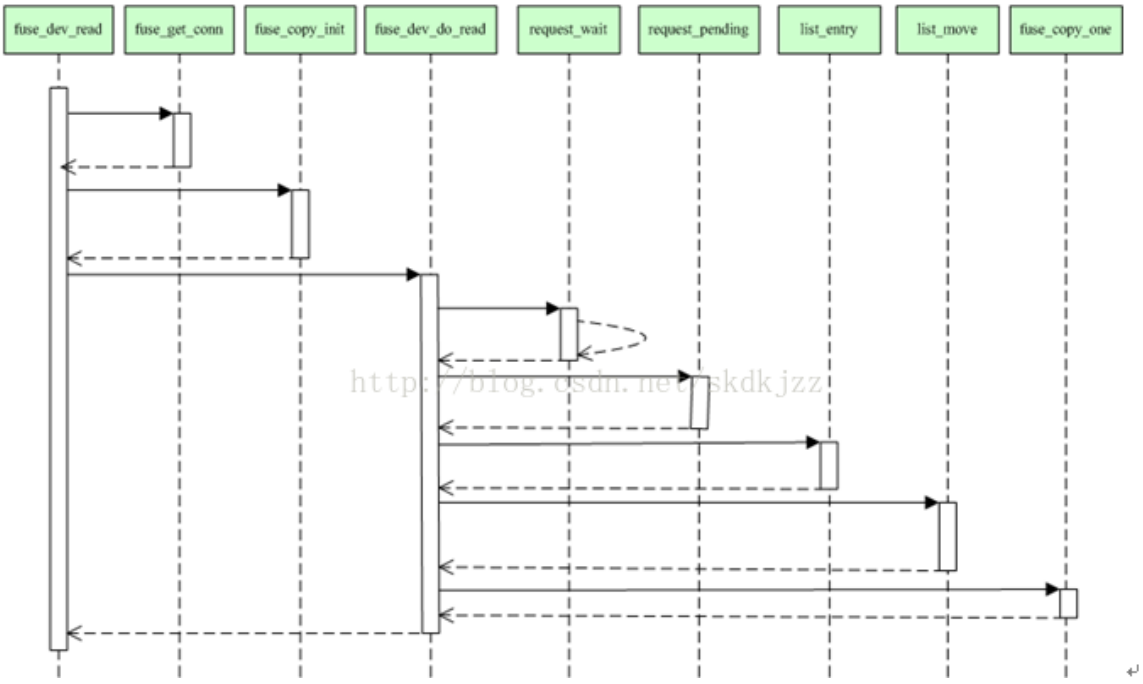

图7

从第1节可知，FUSE模块加载过程注册了对设备文件/dev/fuse的操作接口fuse_dev_operations。由此可知，read底层实际调用的是fuse_dev_read

fuse_dev_read首先通过fuse_get_conn获得struct fuse_conn的实例fc，通过fuse_copy_init为struct fuse_copy_state分配内存并将其实例化。主要的数据读取在fuse_dev_do_read中分4步完成：

1.request_wait:在挂起的列表上等待一个请求到达:

(1).DECLARE_WAITQUEUE(wait, current): 创建等待队列项，并将其初始化为current

(2).add_wait_queue_exclusive(&fc->waitq, &wait): 将wait加入fc->waitq,当有请求发送到

FUSE文件系统时，这个等待队列上的进程会被唤醒

(3).如果没有request，一直循环检查pending list和interrupt list, 直到有请求；

如果有请求则将state设置为TASK_RUNNING

(4).将wait从等待队列中移除

2.list_entry(fc->pending.next, struct fuse_req, list)：从fc->pending.next中取出request,req->state状态设为FUSE_REQ_READING,

3. 将req->list移到fc->io
4. fuse_copy_one:将数据拷贝到struct fuse_copy_state的buf中(此buf指针指向应用层的void *buf)， 返回。

阅读代码时需要注意:fuse_dev_read：struct fuse_copy_state成员write为1；fuse_dev_write：struct fuse_copy_state成员write为0。

用户读取request，分析并执行以后需要调用fuse_write_reply回复内核，这个函数最终调用write写/dev/fuse。图8是write触发的函数调用序列。

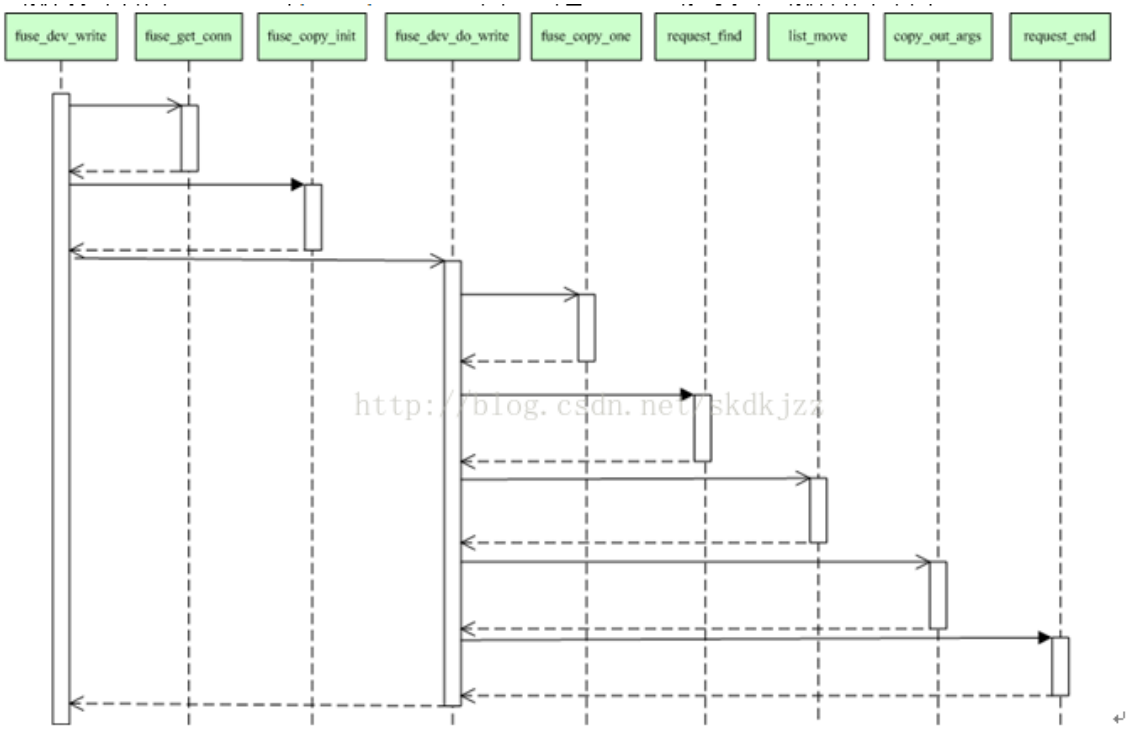

图8

write前两步和read类似即获取fc(struct fuse_conn)和实例化cs(struct fuse_copy_state)实际的写数据操作在fuse_dev_do_write中执行，可以分为7步完成数据的写入：

1.fuse_copy_one(cs, &oh, sizeof(oh)): 将数据从cs(struct fuse_copy_state)拷贝到oh(struct fuse_out_header)

2.request_find(fc, oh.unique)，根据unique id在fc(fuse_conn)中找到相应的request

3.设置req->state为FUSE_REQ_WRITING

4.将req->list移到fc->io队列

5.req被赋予cs->req

6.copy_out_args(cs, &req->out, nbytes)：从cs(struct fuse_copy_state)中拷贝参数到req->out

7.request_end：请求处理完成，设置req->state=FUSE_REQ_FINISHED, 唤醒等待在waitq的进程wake_up(&req->waitq)

5.FUSE数据结构

5.1 内核部分

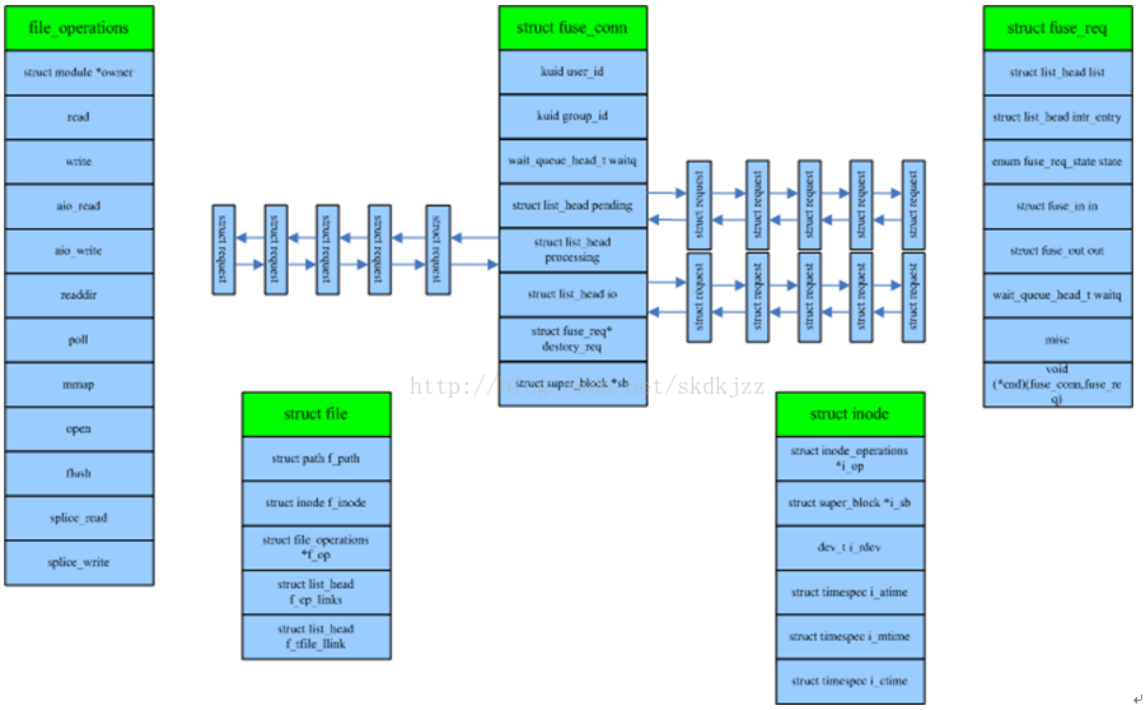

图9

struct fuse_conn：每一次mount会实例化一个struct fuse_conn即fuse connection, 它代表了用户空间和内核的通信连接。fuse connection维护了包括pending list, processing list和io list在内的request queue，fuse connection通过这些队列管理用户空间和内核空间通信过程。

struct fuse_req：每次执行系统调用时会生成一个struct fuse_req, 这些fuse_req依据state被组织在不同的队列中，struct fuse_conn维护了这些队列.

struct file: 存放打开文件与进程之间进行交互的有关信息，描述了进程怎样与一个打开的文件进行交互，这类信息仅当进程访问文件期间存在于内核内存中。

struct inode：文件系统处理文件所需要得所有信息都放在一个名为inode(索引节点)的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随着文件的存在而存在。

struct file_operation：定义了可以对文件执行的操作。

5.2 用户空间部分

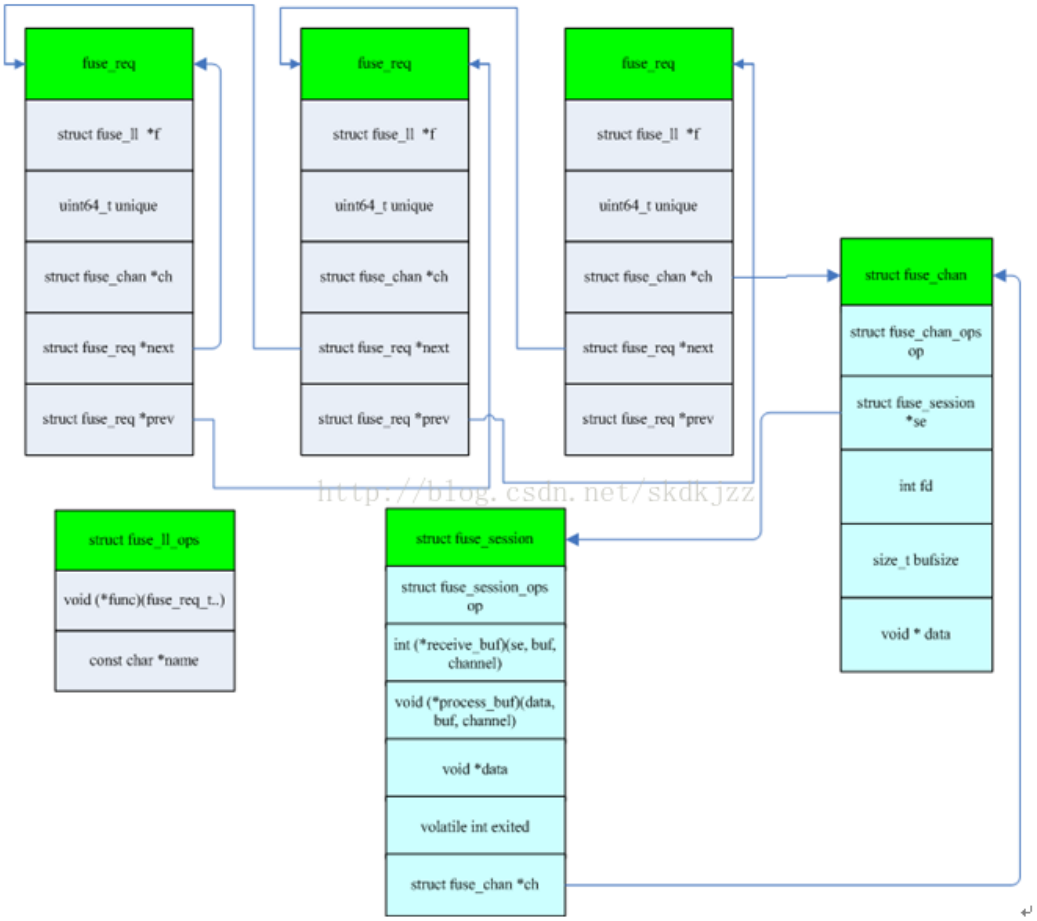

图10

struct fuse_req：这个结构和上文中内核的fuse_req同名，有着类似的作用，但是数据成员不同。

struct fuse_session：定义了客户端管理会话的结构体，包含了一组对session可以执行的操作。

struct fuse_chan：定义了客户端与FUSE内核连接通道的结构体，包含了一组对channel可以执行的操作。

struct fuse_ll_ops：结构的成员为一个函数指针func和命令名字符串name，内核中发过来的每一个request最后都映射到以此结构为元素的数组中。