一 Fuse简要介绍

1 什么是Fuse

传统的文件系统是操作系统的一部分，放在操作系统内核里面实现。Fuse(Filesystem in Userspace), 一个用户空间文件系统框架，提供给我们一组用于实现一个文件系统的API，使我们可以在用户态实现自已的文件系统。目前fuse已集成在Linux2.6以上版本的内核中。

(注：操作系统中的用户态指权限等级中的一般级别，与之相对的是超级用户或者管理员的特权级别。用户态启动的每个进程，根据运行该进程的登录用户，都被系统赋予一定的权限，另外也有一些限制。)

2 优缺点

1. 传统文件系统都是定义在操作系统内核层面上的，要操作系统识别一种新的文件系统，必需重写内核，而内核态代码难以调试，生产率较低；但是用户空间编程和调试难度较小，有更多的语言可以选择（目前FUSE已经绑定了很多语言，比如c++、java等），还可以复用已有的库），从而能够大幅提高生产率，极大地简少了为操作系统提供新的文件系统的工作量。
2. 一些服务可以通过统一的文件系统接口来进行访问，比如说ftp、sftp、samba
3. 可以把非文件的服务当做文件来实现，比如把gmail提供的巨大的空间用来进行文件存储的Gmail Filesystem。

在用户态实现文件系统必然会引入额外的内核态/用户态切换带来的开销，对性能会产生一定影响。

二 Fuse的结构

fuse包括三个模块：用户空间库，内核模块以及mount工具

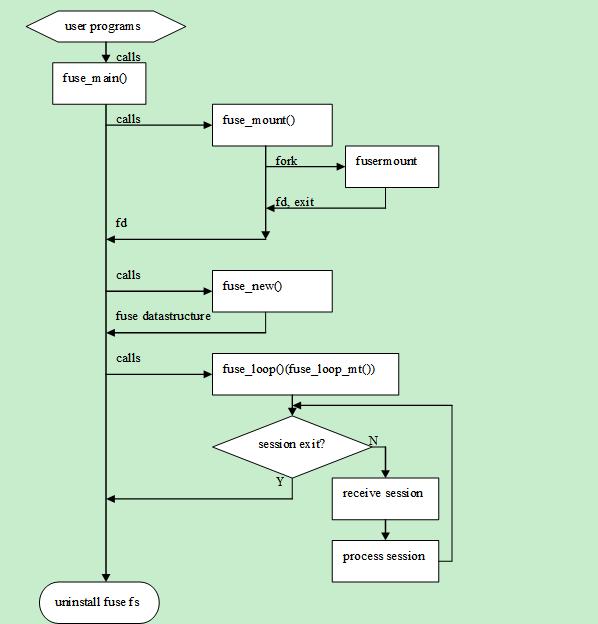
1）用户空间库给程序员提供编程接口，程序员通过实现fuse提供的两组接口fuse\_lowlevel\_ops， fuse\_operations之一即可实现一个用户空间文件系统

2）内核模块实现了一个完整文件系统的框架，但具体操作没有实现（由程序员在用户空间实现）

3）mount工具fusermount用于挂载基于fuse的文件系统

1 Fuse在用户空间工作的流程图

通过这幅图可以看到三个模块在fuse工作时所起的作用



fuse\_main() (lib/helper.c)——fuse用户空间主函数，用户程序调用它时，fuse\_main()函数解析相关参数（如mountpoint，multithreaded），并调用fuse\_mount()函数，接着调用fuse\_new()函数，为fuse文件系统数据分配存储空间。最后调用fuse\_loop()函数实现会话的接受与处理。

fuse\_mount() (lib/mount.c)——创建UNIX本地套接口，创建并运行子进程fusermount。

fusermount (util/fusermount.c)——确保fuse模块已经加载，通过UNIX套接口返回fuse模块的文件fd给fuse\_mount()函数。

fuse\_new() (lib/fuse.c)——为fuse创建数据结构空间，用来存储文件系统数据。

fuse\_loop() (lib/fuse.c)( fuse\_loop\_mt() (lib/fuse\_mt.c))——从/dev/fuse (/dev 设备文件存储目录)读取文件系统调用，调用fuse\_operations或fuse\_lowlevel\_ops结构中的处理函数，返回调用结果给/dev/fuse

2 Fuse内核模块

FUSE Kernel模块由两部分组成：

第一部分——proc文件系统组件：Kernel/dev.c——回应io请求到/dev/fuse。fuse\_dev\_read()函数负责读出文件，并将来自“list of request”结构体的命令返回到调用函数。fuse\_dev\_write ()负责文件写入，并将写入的数据置放到“req→out”数据结构中。

第二部分——文件系统调用部分：kernel/file.c，kernel/inode.c，kernel/dir.c——调用request\_send()，将请求加入到“list of request”结构体中，等待回复(reply)。

三 Fuse调用流程

由于fuse处理请求过程涉及的内容较多，如果从采用从外到内逐层深入的方法来讲，虽然符合逻辑但会增加理解难度，因为到最后大家会迷失在一个个的函数调用里，而且也难以抓住其本质与核心。所以我由其核心——队列管理讲起，向外扩散；再从最外层的函数调用向内讲；最后瞻前顾后，整个fuse处理请求的流程就明白了。

我们先利用下面一幅图简要了解下fuse文件系统工作时的调用路径。

在shell里输入命令，请求通过vfs到达fuse，然后通过用户实现的fuse给出的API返回调用。



1 Fuse处理请求的核心工作就是进行队列管理

1）两个重要的数据结构

fc的定义如下

/\* A Fuse connection.

\* This structure is created, when the filesystem is mounted, and is destroyed, when the

\* client device is closed and the filesystem is unmounted.

\*/

Struct fuse\_conn

{

/\*\* Readers of the connection are waiting on this \*/

wait\_queue\_head\_t waitq; // 等待执行请求的进程的队列

/\*\* The list of pending requests \*/

struct list\_head pending; // 被挂起的请求 的队列

/\*\* The list of requests being processed \*/

struct list\_head processing; // 正在被处理的请求的 队列

/\*\* Pending interrupts \*/

struct list\_head interrupts; // 执行中被中断的请求的 队列

...

}

req的定义如下：

/\*

\*A request to the client

\*/

struct fuse\_req

{

/\*\* Used to wake up the task waiting for completion of request\*/

wait\_queue\_head\_t waitq; // 请求的等待队列

…

}

2）队列管理的过程如下

fuse队列管理.emf

3）队列管理的相关代码

①（左列一至五行） fuse通过fuse\_session\_loop来启动守护程序，守护程序最终会调用fuse\_dev\_readv, fuse\_dev\_readv调用request\_wait，使得进程在fc的waitq队列上睡眠。

Static size\_t fuse\_dev\_readv(struct file \*file, const struct iovec \*iov, unsigned long nr\_segs, loff\_t \*off)

{

….

request\_wait(fc);

….

}

②/\* Wait until a request is available on the pending list

\*当前进程一直等待，直到挂起队列中有一个请求

\*/

static void request\_wait(struct fuse\_conn \*fc)

{

DECLARE\_WAITQUEUE(wait, current); //定义一个队列节点变量wait，其与当前进程相关联

add\_wait\_queue\_exclusive(&fc->waitq, &wait); //将wait加入到fc->waitq等待队列中

//不断的检查fc的pending队列及interrupts队列，看是否有请求，没有请求一直while循环

while (fc->connected && !request\_pending(fc))

{

set\_current\_state(TASK\_INTERRUPTIBLE);

if (signal\_pending(current)) break;

spin\_unlock(&fc->lock);

schedule(); //选择一个进程运行

spin\_lock(&fc->lock);

}

// 有请求，将进程设为TASK\_RUNNING状态(被唤醒，被赋予CPU使用权)

set\_current\_state(TASK\_RUNNING);

remove\_wait\_queue(&fc->waitq, &wait); // 将wait（当前进程）从等待队列中移除

}

③// fc的pending队列及interrupts队列，看是否有请求

static int request\_pending(struct fuse\_conn \*fc)

{

return !list\_empty(&fc->pending) || !list\_empty(&fc->interrupts);

}

④（右列一到四）request\_send是用户请求经过vfs（如上面的图），再到fuse operation中被调用的，它向/dev/fuse发送请求

void request\_send(struct fuse\_conn \*fc, struct fuse\_req \*req)

{

……

queue\_request(fc, req);

request\_wait\_answer(fc, req);

……

}

⑤static void queue\_request(struct fuse\_conn \*fc, struct fuse\_req \*req)

{

list\_add\_tail(&req->list, &fc->pending); //将请求加入到pending队列

req->state = FUSE\_REQ\_PENDING;

if (!req->waiting)

{

req->waiting = 1;

atomic\_inc(&fc->num\_waiting);

}

wake\_up(&fc->waitq); //唤醒等待等列

kill\_fasync(&fc->fasync, SIGIO, POLL\_IN);

}

⑥/\* Called with fc->lock held. Releases, and then reacquires it. \*/

//该调用会在req的waitq上睡眠，fuse守护程序处理完请求后，会将其唤醒

static void request\_wait\_answer(struct fuse\_conn \*fc, struct fuse\_req \*req)

{

if (!fc->no\_interrupt)

{

/\* Any signal may interrupt this \*/

wait\_answer\_interruptible(fc, req);

if (req->aborted)

goto aborted;

if (req->state == FUSE\_REQ\_FINISHED)

return;

req->interrupted = 1;

if (req->state == FUSE\_REQ\_SENT)

queue\_interrupt(fc, req);

}

if (req->force) {

spin\_unlock(&fc->lock);

wait\_event(req->waitq, req->state == FUSE\_REQ\_FINISHED);

spin\_lock(&fc->lock);

} else {

sigset\_t oldset;

/\* Only fatal signals may interrupt this \*/

block\_sigs(&oldset);

wait\_answer\_interruptible(fc, req);

restore\_sigs(&oldset);

}

if (req->aborted)

goto aborted;

if (req->state == FUSE\_REQ\_FINISHED) return;

req->out.h.error = -EINTR;

req->aborted = 1;

aborted:

if (req->locked) {

/\* This is uninterruptible sleep, because data is

being copied to/from the buffers of req. During

locked state, there mustn't be any filesystem

operation (e.g. page fault), since that could lead

to deadlock \*/

spin\_unlock(&fc->lock);

wait\_event(req->waitq, !req->locked);

spin\_lock(&fc->lock);

}

if (req->state == FUSE\_REQ\_PENDING) {

list\_del(&req->list);

\_\_fuse\_put\_request(req);

} else if (req->state == FUSE\_REQ\_SENT) {

spin\_unlock(&fc->lock);

wait\_event(req->waitq, req->state == FUSE\_REQ\_FINISHED);

spin\_lock(&fc->lock);

}

}

}

(左列七行)fuse守护程序处理完请求，最终通过fuse\_dev\_writev写回/dev/fuse，它将唤醒相应req中waitq的等待队列元素，从而让文件系统请求完成request\_wait\_answer，获取到结果。

⑦/\*\*Write a single reply to a request. First the header is copied from the write buffer. The request is then \*searched on the processing list by the unique ID found in the header. If found, then remove it from the list \*and copy the rest of the buffer to the request. The request is finished by calling request\_end()

\*/

static ssize\_t fuse\_dev\_writev(struct file \*file, const struct iovec \*iov, unsigned long nr\_segs, loff\_t \*off)

{

……..

req = request\_find(fc, oh.unique);

request\_end(fc, req);

….

}

⑧/\* \* This function is called when a request is finished. Either a reply has arrived or it was aborted (and not yet \*sent) or some error occurred during communication with userspace, or the device file was closed. The \*requester thread is woken up (if still waiting), the 'end' callback is called if given, else the reference to the \*request is released Called with fc->lock, unlocks it

\*/

static void request\_end(struct fuse\_conn \*fc, struct fuse\_req \*req)

{

….

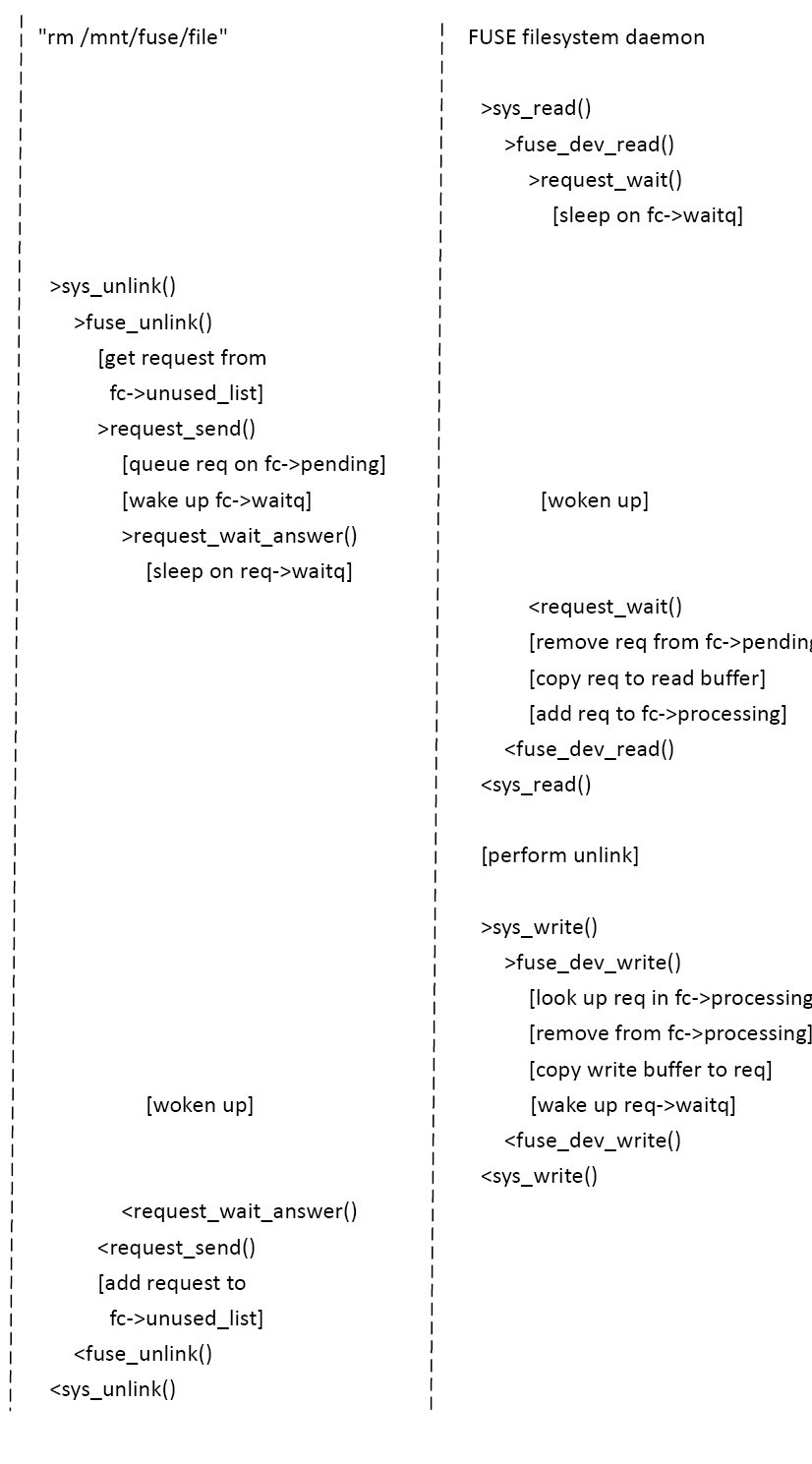
wake\_up(&req->waitq); //唤醒req上的等待队列

……

}

2 以unlink操作为例，根据流程图结合源代码分析fuse处理请求流程

处理unlink操作的整个流程如下图所示。其中“>”表示调用，”<”表示返回，[]表示调用中所做的工作。



①fuse通过fuse\_session\_loop（或对应多线程的方法）来启动fuse守护程序，守护程序不断的从/dev/fuse上读取请求，并处理。

int fuse\_session\_loop(struct fuse\_session \*se) //在fuse\_main中会被调用，或其多线程版本

{

int res = 0;

struct fuse\_chan \*ch = fuse\_session\_next\_chan(se, NULL);

size\_t bufsize = fuse\_chan\_bufsize(ch);

char \*buf = (char \*) malloc(bufsize); //为channel分配好缓冲区

if (!buf) {

fprintf(stderr, "fuse: failed to allocate read buffer\n");

return -1;

}

//fuse daemon, loops

while (!fuse\_session\_exited(se)) {

struct fuse\_chan \*tmpch = ch;

// 从/dev/fuse读请求，会等待一直到有请求为止

res = fuse\_chan\_recv(&tmpch, buf, bufsize);

if (res == -EINTR) continue;

if (res <= 0) break;

fuse\_session\_process(se, buf, res, tmpch); //处理读到的请求

}

free(buf);

fuse\_session\_reset(se);

return res < 0 ? -1 : 0;

}

②int fuse\_chan\_recv(struct fuse\_chan \*\*chp, char \*buf, size\_t size)

{

struct fuse\_chan \*ch = \*chp;

if (ch->compat)

return ((struct fuse\_chan\_ops\_compat24 \*) &ch->op)->receive(ch, buf, size);

else

return ch->op.receive(chp, buf, size); //由下面的一段代码可以发现，receive最终是通过

// fuse\_kern\_chan\_receive实现的，代码片段3分析该请求

}

③#define MIN\_BUFSIZE 0x21000

struct fuse\_chan \*fuse\_kern\_chan\_new(int fd)

{

//channel的读写方法

struct fuse\_chan\_ops op = {

.receive = fuse\_kern\_chan\_receive,

.send = fuse\_kern\_chan\_send,

.destroy = fuse\_kern\_chan\_destroy,

};

//设置bufsize大小

size\_t bufsize = getpagesize() + 0x1000;

bufsize = bufsize < MIN\_BUFSIZE ? MIN\_BUFSIZE : bufsize;

return fuse\_chan\_new(&op, fd, bufsize, NULL);

}

④static int fuse\_kern\_chan\_receive(struct fuse\_chan \*\*chp, char \*buf, size\_t size)

{

struct fuse\_chan \*ch = \*chp;

int err;

ssize\_t res;

struct fuse\_session \*se = fuse\_chan\_session(ch);

assert(se != NULL);

// 一直轮询，直到读到请求为止

restart:

//fuse\_chan\_fd获取到/dev/fuse的文件描述符，调用read系统调用从设备读取请求

res = read(fuse\_chan\_fd(ch), buf, size);

//根据fuse设备驱动程序file结构的实现（dev.c），read将调用fuse\_dev\_read,该方法最终通过fuse\_dev\_readv

//实现，根据代码中的注释，fuse\_dev\_read做了如下工作：

// Read a single request into the userspace filesystem's buffer. This function waits until a request is available,

// then removes it from the pending list and copies request data to userspace buffer.

// 而fuse\_dev\_read又调用request\_wait,使得进程在fc->waitq上睡眠

if no data： goto restart

………

}

以上的分析对应了fuse filesystem daemon做的第一部分工作。当用户从控制台输入"rm /mnt/fuse/file"时，通过VFS（sys\_unlink），再到fuse（dir.c中实现的inode\_operations，file.c中实现的file\_operations中的方法都会最终调用request\_send，后面会讲到），这个请求最终被发到了/dev/fuse中，该请求的到达会唤醒正在等待的fuse守护程序，fuse守护程序读取该请求并进行处理，接下来介绍处理请求所作的工作。

⑤struct fuse\_session \*fuse\_lowlevel\_new\_common(struct fuse\_args \*args,

const struct fuse\_lowlevel\_ops \*op,

size\_t op\_size, void \*userdata)

{

//fuse\_lowlevel\_ops在之前的文章

http://blog.chinaunix.net/u2/87570/showart\_2166461.html中已经介绍//过了，开发者实现了fuse\_lowlevel\_ops并传递给fuse\_lowlevel\_common

struct fuse\_ll \*f;

struct fuse\_session \*se;

struct fuse\_session\_ops sop = {

//最终调用的处理方法

.process = fuse\_ll\_process, //分析见代码片段5

.destroy = fuse\_ll\_destroy,

};

…….

}

⑥static void fuse\_ll\_process(void \*data, const char \*buf, size\_t len, struct fuse\_chan \*ch)

{

struct fuse\_ll \*f = (struct fuse\_ll \*) data;

struct fuse\_in\_header \*in = (struct fuse\_in\_header \*) buf;

const void \*inarg = buf + sizeof(struct fuse\_in\_header);

struct fuse\_req \*req;

//创建并初始化一个请求

req = (struct fuse\_req \*) calloc(1, sizeof(struct fuse\_req));

if (req == NULL) {

fprintf(stderr, "fuse: failed to allocate request\n");

return;

}

req->f = f;

req->unique = in->unique;

……

//根据opcode调用fuse\_ll\_ops中相应的方法，fuse\_ll\_ops的介绍

// http://blog.chinaunix.net/u2/87570/showart\_2166461.html

fuse\_ll\_ops[in->opcode].func(req, in->nodeid, inarg);

}

}

以上代码对应中流程中perform unlink的工作，实际上就是执行开发者实现的一组方法来完成相关的工作，接下来就是把执行完请求后需要的数据返回，最终是通过send\_reply实现的，

⑦static int send\_reply(fuse\_req\_t req, int error, const void \*arg, size\_t argsize)

{

struct iovec iov[2];

int count = 1;

if (argsize) {

iov[1].iov\_base = (void \*) arg;

iov[1].iov\_len = argsize;

count++;

}

return send\_reply\_iov(req, error, iov, count);

}

⑧static int send\_reply\_iov(fuse\_req\_t req, int error, struct iovec \*iov, int count)

{

……

res = fuse\_chan\_send(req->ch, iov, count);

free\_req(req);

return res;

}

⑨static int fuse\_kern\_chan\_send(struct fuse\_chan \*ch, const struct iovec iov[], size\_t count)

{

if (iov) {

//将数据写到/dev/fuse上,最终会调用fuse\_dev\_write

ssize\_t res = writev(fuse\_chan\_fd(ch), iov, count);

……

return 0;

}

另外fuse接收到VFS的请求时，通过request\_send将请求发送到/fuse/dev，并调用request\_wait\_answer等待返回结果。至于fuse使用的队列的管理，在流程图中也做了简单的说明，下一篇文章将详细分析队列的管理。

⑩void request\_send(struct fuse\_conn \*fc, struct fuse\_req \*req)

{

req->isreply = 1;

spin\_lock(&fc->lock);

if (!fc->connected)

req->out.h.error = -ENOTCONN;

else if (fc->conn\_error)

req->out.h.error = -ECONNREFUSED;

else {

//将请求加入请求队列

queue\_request(fc, req);

/\* acquire extra reference, since request is still needed after request\_end() \*/

\_\_fuse\_get\_request(req);

//等待结果

request\_wait\_answer(fc, req);

}

spin\_unlock(&fc->lock);

}