用户态文件系统框架分析

# 用户态文件系统介绍

用户态文件系统（filesystem in userspace， 简称FUSE），它能使用户在无需编辑和编译内核代码的情况下，创建用户自定义的文件系统。文件系统是操作系统的重要组成部分，一般在内核层面实现对于文件系统的支持，而通常内核态的代码难以调试，生产率较低。在用户态空间实现文件系统能够极大幅度的提高生产效率，简化为实现新的文件系统的工作量。FUSE主要包含两个部分，内核FUSE模块（Linux从2.6.14版本开始支持）和用户态Libfuse库。

目前FUSE支持的平台：

* Linux 完全支持
* BSD 部分支持
* OX-X参考[OSXFUSE](https://osxfuse.github.io/)

比较知名的用户态文件系统：

* ExpanDrive：商业文件系统，实现了SFTP/FTP/FTPS协议；
* GlusterFS：用于集群的分布式文件系统，可以扩展到PB级；
* SSHFS：通过SSH协议访问远程文件系统；
* GmailFS：通过文件系统方式访问GMail；
* EncFS：加密的虚拟文件系统
* NTFS-3G和Captive NTFS，在非Windows中对NTFS文件系统提供支持；
* WikipediaFS：支持通过文件系统接口访问Wikipedia上的文章；
* 升阳公司的Lustre：和GlusterFS类似但更早的一个集群文件系统
* ZFS：Lustre的Linux版；
* archivemount：
* HDFS: Hadoop提供的分布式文件系统。HDFS可以通过一系列命令访问，并不一定经过Linux FUSE；

在嵌入式开发平台上，我们利用FUSE实现unionfs，quota fs, RIP和temp sensor的文件系统。

FUSE官网：

<https://github.com/libfuse/libfuse>

# FUSE实现机制分析

在这个章节，我们首先对于虚拟文件系统做一个简单介绍，Linux下的文件系统都依赖于虚拟文件系统，要了解FUSE，首先要对虚拟文件系统有一个了解。然后我们对于FUSE做一个宏观框架的分析，先大致了解一下整个FUSE是如何工作的，最后两个小节分别从用户态和内核态具体分析FUSE的实现。

## 虚拟文件系统介绍（VFS）

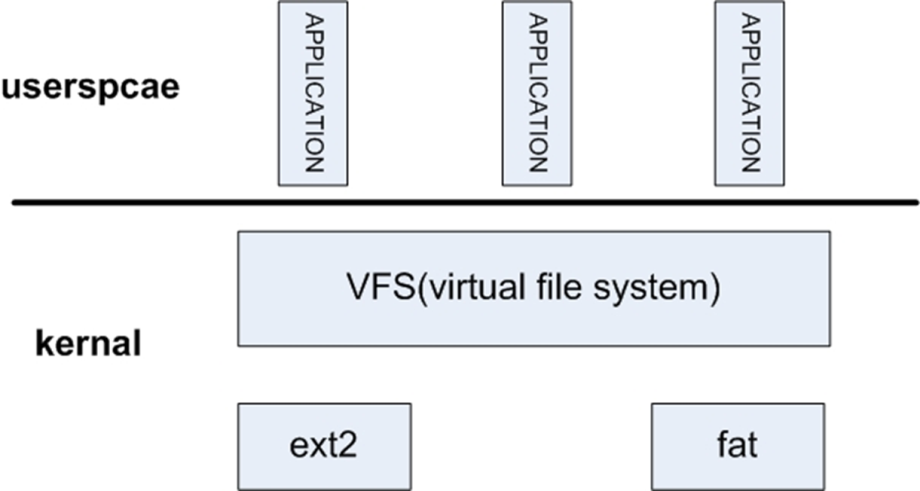
Linux支持ext，ext2，xia，minix，umsdos，msdes，fat32 ，ntfs，proc，stub，ncp，hpfs，affs 以及 ufs 等多种文件系统。为了实现这一目的，Linux 对所有的文件系统采用统一的文件界面，用户通过文件的操作界面来实现对不同文件系统的操作。对于用户来说，我们不要去关心不同文件系统的具体操作过程，而只是对一个虚拟的文件操作界面来进行操作，这个操作界面就是 Linux 的虚拟文件系统(VFS ) 。形象地说，Linux 的内核好像一个 PC 机的主板，VFS 就是上面的一个插槽，具体的文件系统就是外设卡。因此，每一个文件系统之间互不干扰，而只是调用相应的程序来实现其功能。在 Linux 的内核文件中，VFS 和具体的文件系统程序都放在 Linux\FS 中，其中每一种文件系统对应一个子目录，另外还有一些共用的 VFS 程序。在具体的实现上，每个文件系统都有自己的文件操作数据结构file-operations。所以，VFS 作为 Linux内核中的一个软件层，用于给用户空间的程序提供文件系统接口，同时也提供了内核中的一个抽象功能，允许不同的文件系统很好地共存。VFS 使 Linux 同时安装、支持许多不同类型的文件系统成为可能。VFS 拥有关于各种特殊文件系统的公共界面，如超级块、inode、文件操作函数入口等。实际文件系统的细节，统一由 VFS 的公共界面来索引，它们对系统核心和用户进程来说是透明的。

图2-1 VFS示意图

FUSE内核模块的实现跟传统的文件系统实现既有相似点，也有差别的地方，FUSE内核模块实现了FUSE文件系统，只不过与传统的文件系统不同， FUSE需要**把**VFS层的请求传到用户态的fuse app，在用户态处理，然后再返回到内核态，把结果返回给VFS层。更多细节，且看下文。

## FUSE宏观框架

当用户自定义一个新的用户态文件系统被挂载之后，我们在访问该文件系统的文件的方式与访问其他文件系统的文件是一样的，VFS保证了这一点。不同的是，FUSE文件系统下面的访问行为是可以用户自定义的。我们从一个简单的例子出发，先宏观上理解一下整个FUSE工作的流程。

以open为例，整个调用的过程如下：

1. 用户态app调用glibc open接口，触发sys\_open系统调用。
2. sys\_open 调用fuse中inode节点定义的open方法。
3. inode中open生成一个request消息，并通过/dev/fuse发送request消息到用户态libfuse。
4. Libfuse调用fuse\_application用户自定义的open的方法，并将返回值通过/dev/fuse通知给内核。
5. 内核收到request消息的处理完成的唤醒，并将结果放回给VFS系统调用结果。
6. 用户态app收到open的返回结果。

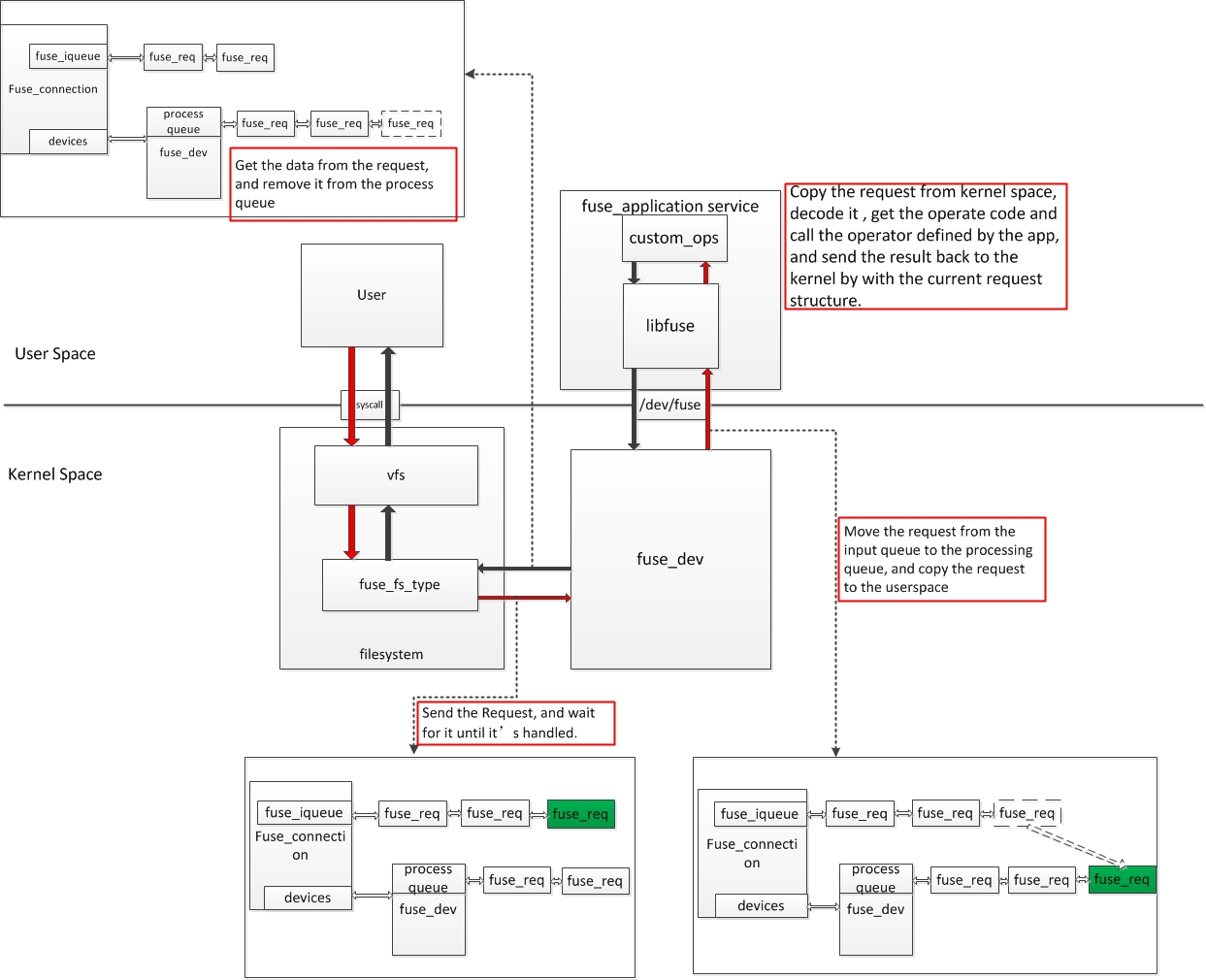


图2-2 FUSE实现框架图

## Libfuse实现分析

对于Libfuse的分析，我们从一个简单的例子开始(example/hello.c)。

这个例子实现了一个最简单的用户态文件系统fuse.hello。

# hello /mnt/

# cd /mnt/

# ls -l

total 0

-r--r--r-- 1 root root 13 Jan 1 1970 hello

# cat hello

Hello World!

# touch a

touch: a: Function not implemented

# echo 0 > hello

-sh: can't create hello: Permission denied

# mkdir x

mkdir: can't create directory 'x': Function not implemented

static struct fuse\_operations hello\_oper = {

.getattr = hello\_getattr,

.readdir = hello\_readdir,

.open = hello\_open,

.read = hello\_read,

};

int main(int argc, char \*argv[])

{

return fuse\_main(argc, argv, &hello\_oper, NULL);

}

上面是测试的结果，可以看到：

1. 可以读取目录
2. 可以读取文件属性
3. 可以读文件，不可以写文件
4. 不可以创建目录
5. 不可以创建文件

我们再结合hello.c中定义的方法，不难看出它们之间的关联。要使用FUSE实现自己的文件系统，我们需要定义一个fuse\_operations类型的结构体变量，并将它传递给fuse\_main，剩下的交给libfuse去处理，实现一个文件系统简单了很多。

接下来我们看一下Fuse\_operations的定义：

*struct fuse\_operations {*

*int (\*getattr) (const char \*, struct stat \*);*

*int (\*readlink) (const char \*, char \*, size\_t);*

*int (\*getdir) (const char \*, fuse\_dirh\_t, fuse\_dirfil\_t);*

*int (\*mknod) (const char \*, mode\_t, dev\_t);*

*int (\*mkdir) (const char \*, mode\_t);*

*int (\*unlink) (const char \*);*

*int (\*rmdir) (const char \*);*

*int (\*symlink) (const char \*, const char \*);*

*int (\*rename) (const char \*, const char \*);*

*int (\*link) (const char \*, const char \*);*

*int (\*chmod) (const char \*, mode\_t);*

*int (\*chown) (const char \*, uid\_t, gid\_t);*

*int (\*truncate) (const char \*, off\_t);*

*int (\*utime) (const char \*, struct utimbuf \*);*

*int (\*open) (const char \*, struct fuse\_file\_info \*);*

*int (\*read) (const char \*, char \*, size\_t, off\_t,*

*struct fuse\_file\_info \*);*

*int (\*write) (const char \*, const char \*, size\_t, off\_t,*

*struct fuse\_file\_info \*);*

*int (\*statfs) (const char \*, struct statvfs \*);*

*int (\*flush) (const char \*, struct fuse\_file\_info \*);*

*int (\*release) (const char \*, struct fuse\_file\_info \*);*

*int (\*fsync) (const char \*, int, struct fuse\_file\_info \*);*

*int (\*setxattr) (const char \*, const char \*, const char \*, size\_t, int);*

*int (\*getxattr) (const char \*, const char \*, char \*, size\_t);*

*int (\*listxattr) (const char \*, char \*, size\_t);*

*int (\*removexattr) (const char \*, const char \*);*

*int (\*opendir) (const char \*, struct fuse\_file\_info \*);*

*int (\*readdir) (const char \*, void \*, fuse\_fill\_dir\_t, off\_t,*

*struct fuse\_file\_info \*);*

*int (\*releasedir) (const char \*, struct fuse\_file\_info \*);*

*int (\*fsyncdir) (const char \*, int, struct fuse\_file\_info \*);*

*void \*(\*init) (struct fuse\_conn\_info \*conn);*

*void (\*destroy) (void \*);*

*int (\*access) (const char \*, int);*

*int (\*create) (const char \*, mode\_t, struct fuse\_file\_info \*);*

*int (\*ftruncate) (const char \*, off\_t, struct fuse\_file\_info \*);*

*int (\*fgetattr) (const char \*, struct stat \*, struct fuse\_file\_info \*);*

*int (\*lock) (const char \*, struct fuse\_file\_info \*, int cmd,*

*struct flock \*);*

*int (\*utimens) (const char \*, const struct timespec tv[2]);*

*int (\*bmap) (const char \*, size\_t blocksize, uint64\_t \*idx);*

*unsigned int flag\_nullpath\_ok:1;*

*unsigned int flag\_nopath:1;*

*unsigned int flag\_utime\_omit\_ok:1;*

*unsigned int flag\_reserved:29;*

*int (\*ioctl) (const char \*, int cmd, void \*arg,*

*struct fuse\_file\_info \*, unsigned int flags, void \*data);*

*int (\*poll) (const char \*, struct fuse\_file\_info \*,*

*struct fuse\_pollhandle \*ph, unsigned \*reventsp);*

*int (\*write\_buf) (const char \*, struct fuse\_bufvec \*buf, off\_t off,*

*struct fuse\_file\_info \*);*

*int (\*read\_buf) (const char \*, struct fuse\_bufvec \*\*bufp,*

*size\_t size, off\_t off, struct fuse\_file\_info \*);*

*int (\*flock) (const char \*, struct fuse\_file\_info \*, int op);*

*int (\*fallocate) (const char \*, int, off\_t, off\_t,*

*struct fuse\_file\_info \*);*

*};*

在fuse\_operations中所有的方法都是可选的，但是为了实现一个有价值的文件系统，有些方法是必须实现的(比如getattr)。

getattr() 类似于stat()

readlink() 读取链接文件的真实文件路径

getdir() 已经过时，使用readdir()替代

mknod() 创建一个文件节点

mkdir() 创建一个目录

unlink() 删除一个文件

rmdir() 删除一个目录

syslink() 创建一个软链接

rename() 重命名文件

link() 创建一个硬链接

chmod() 修改文件权限

chown() 修改文件的所有者和所属组

truncate() 改变文件的大小

utime() 修改访问和修改文件的时间，已经过时，使用utimens()替代

open() 打开文件

read() 读取文件

write() 写文件

statfs() 获取文件系统状态

flush() 刷缓存数据

release() 释放打开的文件

fsync() 同步数据

setxattr() 扩展属性接口， 下同

getxattr()

listxattr()

removexattr()

opendir() 打开一个目录

readdir() 读取目录

releasedir() 释放打开的目录

fsyncdir() 同步目录

init() 初始化文件系统

destroy() 清理文件系统

access() 检查访问权限

create() 创建并打开文件

ftruncate() 修改文件的大小

fgetattr() 获取文件属性

lock()

utimens()

bmap()

ioctl()

poll()

write\_buf()

read\_buf()

flock()

fallocate()

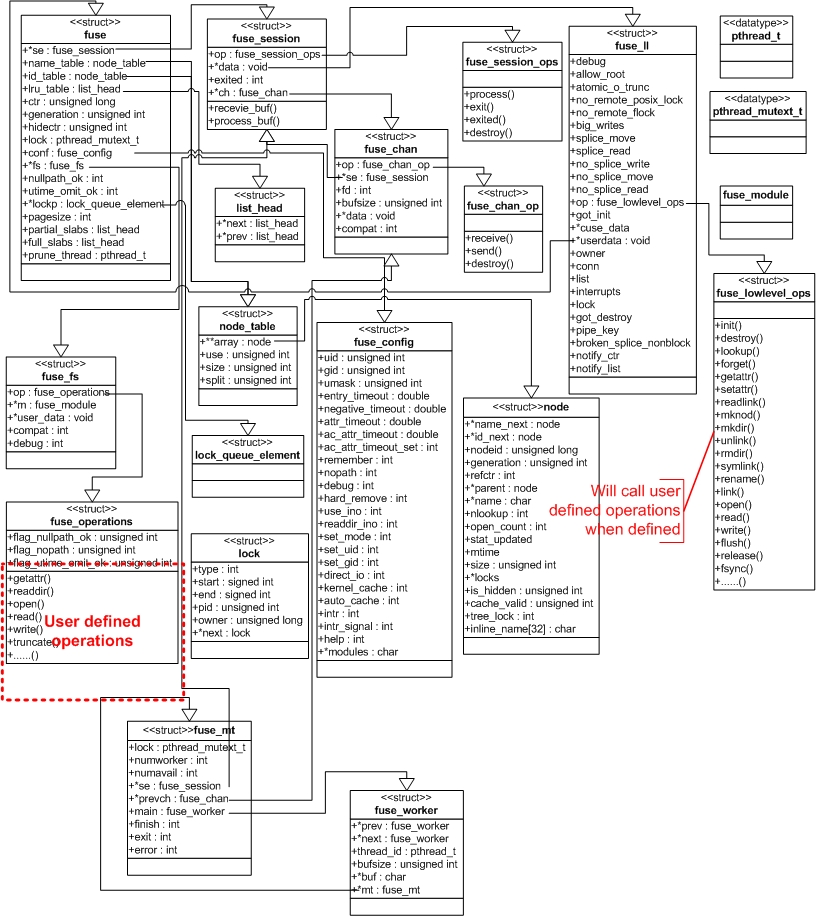
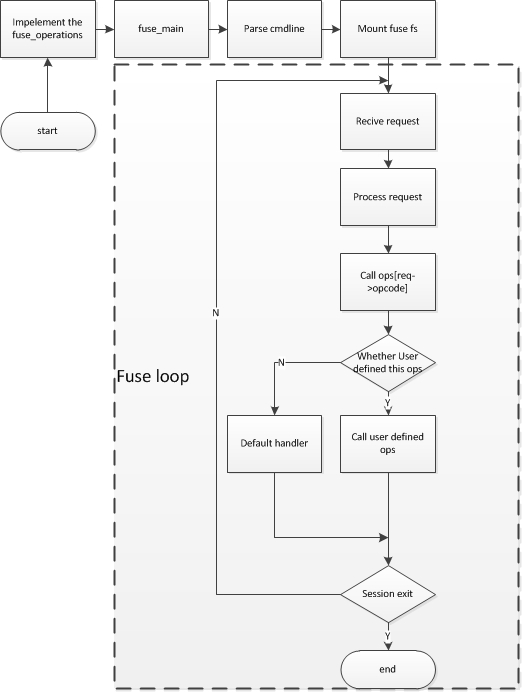
在探究libfuse的实现之前，我们先给出libfuse的核心的数据结构框架图。这幅图可以我们在阅读代码的时候作为参考。

图2-3 Libfuse数据结构图

接下来我们来看一下libfuse是如何实现的。

Libfuse从fuse\_main这个入口开始，从这里我们注册进去定义的文件操作方法来实现我们自己的文件系统。 在fuse\_main中，首先会完成参数解析，注册用户定义的operations, 实现文件系统的挂载（系统调用mount）， 填充fuse相关的数据结构，消息的处理。消息的处理部分是libfuse最核心的部分，实现用户态与内核的互动（/dev/fuse），从内核接收req消息，解析，调用用户自定义的ops，完成处理后，把结果通过/dev/fuse返回给内核，内核再返回给VFS层的系统调用，获得结果。

图2-4 libfuse实现流程图

## FUSE内核实现分析

对于内核部分又为两个部分，一个部分是文件系统部分，另一个部分是字符设备部分。两部分建立关联是在文件系统挂载的时候。

我们首先从挂载部分看起，利用strace工具，截取mount系统调用相关的信息：

# strace hello /mnt/

…

open("/dev/fuse", O\_RDWR|O\_LARGEFILE) = 3

…

mount("hello", "/mnt", "fuse.hello", MS\_NOSUID|MS\_NODEV, "fd=3,rootmode=40000,user\_id=0,gr"...) = 0

…

mount参数部分需要关注一下，fd=3，这个是关联字符设备和文件系统关键纽带。

上面是用户态的系统调用，接下来我们再来看一下内核态中mount系统调用的处理过程。

Sys\_mount

|-> do\_mount

|-> do\_new\_mount

|-> get\_fs\_type

|-> vfs\_kern\_mount

|-> mount\_fs

|-> type->mount() [fuse\_mount]

|-> mount\_nodev

|-> fuse\_fill\_super

|-> file = fget(d.fd);

|-> fc->sb = sb

|-> sb->s\_fs\_info = fc

|-> fud = fuse\_dev\_alloc(fc)

|-> file->private\_data = fud

在最后的fuse\_fill\_super部分，file就是通过mount传进来的参数”fd=3”得到的，对应于打开的“/dev/fuse”。在挂载时候创建的superblock，fuse\_conn, fuse\_dev，file在这里关联起来了，具体的可以看一下图2-5更清楚一些。

 图2-5 Linux FUSE模块数据结构图

接下来我们以删除一个文件为例，看一下FUSE是如何工作的，图2-6摘自libfuse官方文档内核部分。

首先fuse\_app会阻塞在读/dev/fuse, 当挂载点下面有新的行为（删除文件）触发时，会通过系统调用调用fuse文件系统内核接口，并生成request消息，同时唤醒阻塞的fuse\_app读操作，fuse\_app读到request之后，到用户态利用libfuse进行解析，根据request中的opcode找到对应的ops并执行，执行之后通过/dev/fuse把处理的结果传回。VFS阻塞的行为会被唤醒，然后完成VFS的访问。

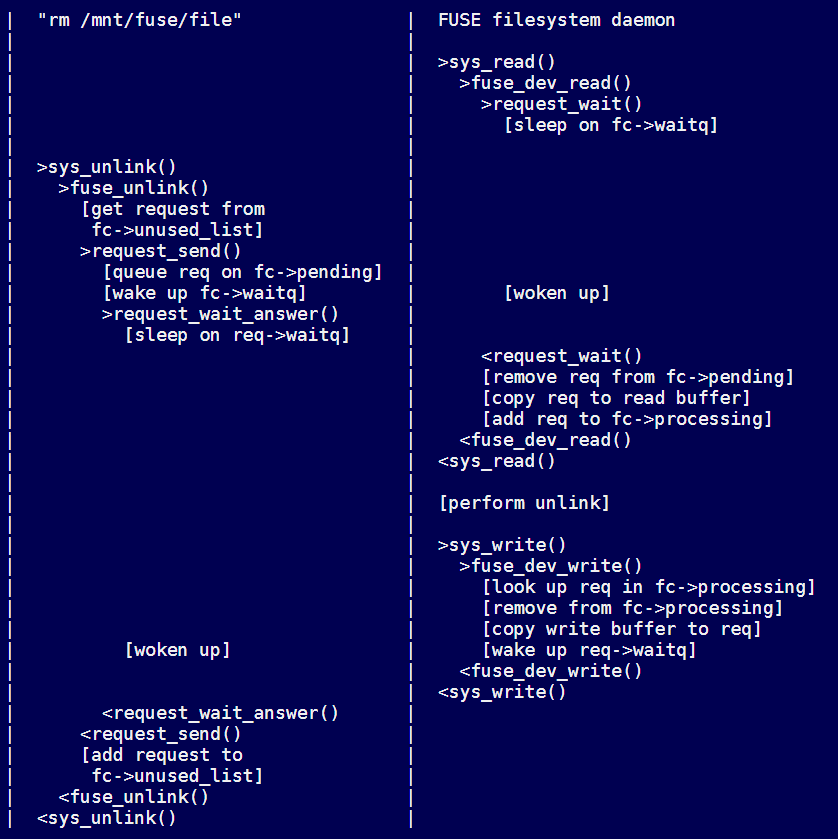


图2-6 用户态和内核态交互过程示例

# FUSE实践过程记录

在实践章节，我们准备在QEMU环境中演示一下一个实用的用户态文件系统，实现用户态配额文件系统。这里我们需要用到buildroot和QEMU，本文主要还是为了演示FUSE，对于buildroot和QEMU本身不做详细介绍，只介绍一些用到的命令。

Buildroot是一个开源组件，广泛用于嵌入式开发平台，集toolchain，rootfs，bootloader，kernel，open source package等于一身，方便开发者定制自己的linux系统。

QEMU是一个虚拟机，可以做到指令集的仿真，支持x86，ARM，powerpc等架构，可以用于模拟实体板卡。

HOST平台： Ubuntu14.04

TARGET平台：qemu\_vexpress

GITHUB repo：

<https://github.com/JinhuaW/buildroot.git>

<https://github.com/JinhuaW/target-apps.git>

1. 安装实验必须的组件：

sudo apt-get install qemu git g++

1. 从github上克隆buildroot库。

git clone <https://github.com/JinhuaW/buildroot.git>

1. 编译，切换到buildroot根目录

jinhuawu@UbuntuPC:~/buildroot$ make qemu\_arm\_vexpress\_defconfig

#

# configuration written to /home/jinhuawu/buildroot/.config

#

jinhuawu@UbuntuPC:~/buildroot$ make

在编译完成那个以后，我们可以得到下面的image

jinhuawu@UbuntuPC:~/buildroot/output/images$ ls

rootfs.ext2 vexpress-v2p-ca9.dtb zImage

1. 运行QEMU环境

jinhuawu@UbuntuPC:~/buildroot/output/images$ ls

rootfs.ext2 vexpress-v2p-ca9.dtb zImage

jinhuawu@UbuntuPC:~/buildroot/output/images$ qemu-system-arm -M vexpress-a9 -m 512M -nographic -append "root=/dev/mmcblk0 console=ttyAMA0" -kernel zImage -sd rootfs.ext2 -dtb vexpress-v2p-ca9.dtb

1. 测试app。

在用QEMU把target环境启动起来以后，我们可以测试我们quotafs，quotafs相应的代码可以从<https://github.com/JinhuaW/target-apps.git> 获取。

Welcome to Buildroot

buildroot login: root

# quotafs -h

Usage: quota --src source\_dirctory --size quota\_size mount\_point [OPTIONS]

Mount a user space quota filesytem.

-h, show fuse option help

--src, the source dircotry is going to setup the quota

--size, the size of the quota

# quotafs --src /var/ --size 1024000 /mnt/

# mount

/dev/root on / type ext2 (rw,relatime)

devtmpfs on /dev type devtmpfs (rw,relatime,size=256244k,nr\_inodes=64061,mode=755)

proc on /proc type proc (rw,relatime)

devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620)

tmpfs on /dev/shm type tmpfs (rw,relatime,mode=777)

tmpfs on /tmp type tmpfs (rw,relatime)

tmpfs on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)

sysfs on /sys type sysfs (rw,relatime)

quotafs on /mnt type fuse.quotafs (rw,nosuid,nodev,relatime,user\_id=0,group\_id=0)

# cd /mnt/

# ls

cache lock run tmp used\_size

lib log spool total\_size www

# cat total\_size

1024000

# cat used\_size

492866

# dd if=/dev/zero of=test bs=1 count=1000

1000+0 records in

1000+0 records out

# cat used\_size

493866