# 基于Linux的基因组压缩文件系统

## 1工作任务

通过修改linux系统的vfs层，实现fastq文件的压缩和解压。当文件通过下载或拷贝进入系统时，我们直接将其压缩为fqz文件；当需要读取数据时，我们的系统内部先进行解压，然后再读取；需要写入数据时，我们需要先写入硬盘，然后再进行压缩。当然，对于复制，剪切，移动，重命名等文件常用的操作，我们也需要实现其功能。（即一个fastq文件，内部实际存放的是fqz格式，但是无论是进行什么操作，与普通的fastq文件没有差异，函数接口也完全一样）此外，我们还需要解决以下的问题：

1. 读后写，写后读，读后读的矛盾。
2. 较大文件的读取。
3. 并发操作（多进程，多线程）时进行加锁，防止系统多次解压。

## 2实现方式

我们通过修改vfs层的内核代码，在vfs层标注需要压缩的文件，并将进程阻塞。我们暂时使用存储到文件的方式进行处理，存储到文件时，文件名用原fastq文件的路径来设置，文件位置存储在统一的文件夹（现在我们也可以使用netlink来进行操作）。而在应用层，我们编写一个伺服端的程序，监控该文件夹中的文件的变化（或接收socket），当我们触发了伺服端的操作时，伺服端实现解压或者压缩的操作，并将标记写入文件（或发出socket）,然后vfs层接收到操作，阻塞的进程继续进行，vfs层完成原本需要的操作。

我们使用的文件主要存放在kernel\_op和kernel\_info这两个文件夹中，kernel\_info是vfs层负责写入需要进行文件操作和文件路径，伺服端负责监控，而kernel\_op由伺服端负责写入，0和1来标识伺服端的操作是否完成，vfs层负责读取。

此外，我们将压缩文件存放为.fq后缀，而解压后的文件我们存放为隐藏文件的形式，以.dcmp作为后缀。

此外，我们使用的压缩工具为fqz\_comp，下载的网址为

<https://sourceforge.net/projects/fqzcomp/files/latest/download>。

## 3调试方法

我们主要使用qemu虚拟机和直接编译到内核的方式的进行调试。

对于qemu，安装qemu的步骤不用赘述。安装完之后，首先我们需要先建立一个ramdisk。我们下载busybox，make menuconfig设置为静态编译，然后make，和make install，生成bin，sbin和usr三个文件夹，然后我们自己建立一个ramdisk文件夹，将这三个文件夹移入其中，并建立，dev，etc，proc，sys这四个文件夹，在etc文件夹下，我们需要自己写fstab和inittab这两个文件，然后在etc目录下创建init.d文件夹，在init.d文件夹内自己写rcS文件，最后我们需要在ramdisk文件夹中自己写一个init文件。上述四个自己需要编写的文件内容均写在附录中。

此外，你需要什么文件，你都可以放入ramdisk这个文件夹中，如果是c文件，你需要提前静态编译生成可执行文件。在执行完以上步骤后，我们在ramdisk中执行

find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../test.img

命令，将生成一个test.img的虚拟磁盘。

内核的代码我们使用的是linux-4.15.0版本，无论对于qemu还是直接编译到内核，我们都需要先make menuconfig，然后make bzImage –j8(线程数根据电脑确定)，具体步骤直接百度linux编译内核即可。bzImage存放在linux内核文件的arch/x86/boot目录下。对于qemu，我们执行

qemu-system-x86\_64 -kernel arch/x86/boot/bzImage -initrd ../test.img -m 2G -smp 4

这条命令，其中的../test.img是之前生成的test.img文件的路径，arch/x86/boot/bzImage是bzImage的路径，我们就可以在qemu虚拟机中测试内核。

而直接编译到主机的话，make bzImage –j8需要make modules\_install和make install这两步操作，可以直接百度就有教程。

## 4具体方法

##### 4.1在vfs层

在vfs层我们主要修改了两个内核的c文件，一个是fs/read\_write.c，一个是fs/namei.c。其中，fs/read\_write.c中我们主要修改了vfs\_read()和vfs\_write()这两个函数，fs/namei.c中主要修改了vfs\_rename()这个函数。此外，我们还在这两个c文件中添加了几个函数，在fs/read\_write.c我们添加了getfullpath()，real\_vfs\_read()，read\_ready()，str\_replace()，real\_vfs\_write()，而在fs/namei.c中我们添加了getfullpath\_from\_dentry()这个函数，并用到了fs/read\_write.c中的函数，接下来，我们将对这些函数进行功能和代码的介绍。

getfullpath()函数的声明为char\* getfullpath(struct file\* file)，输入是一个file的结构体，返回的是一个字符串，里面存放的是该文件的路径，主要使用了d\_path()这个函数获取文件路径，然后放入到数组中，由于对于返回的字符串我们需要通过kmalloc() 申请的内存位于物理内存映射区域，并使用memset()初始化，所以我们需要将这整个过程封装为一个完整的函数。

real\_vfs\_read()函数的声明和原来的vfs\_read()一样，包含file结构体，缓存buf，计数count和偏移量pos，主要是为了除了fastq文件以外的文件方便操作，即为原本的vfs\_read()函数，同理，real\_vfs\_write()是原本的vfs\_write()函数。

read\_ready()函数的声明为bool read\_ready(char\* filename, struct file \*file)，主要判断fastq文件是否已经解压，我们假设10us判断一次，在现在的方法中，我们使用文件进行判断，伺服端若没有完成，即kernel\_op中的内容为0，则返回false，如果伺服端已经完成工作，即kernel\_op中的内容为1，则返回true。

str\_replace()函数的声明为void str\_replace(char\* des, char\* src, char dch, char sch)，这个函数的主要目的是将原src字符传中的sch字符，换成dch字符，并写入到新的des字符串中。

getfullpath\_from\_dentry()函数的声明为char\* getfullpath\_from\_dentry(struct dentry\* dentry)，这个函数的主要目的是通过dentry这个目录项获得文件的路径。

###### 4.1.1 vfs\_read()

对于vfs\_read()函数的修改主要如下。我们在进入vfs\_read()函数后，首先先判断文件的后缀，如果文件的后缀是fq，我们进行单独的处理，否则，我们调用real\_vfs\_read()进行处理。vfs\_read()读取文件的时候，一个进程会反复调用vfs\_read()，逐个包读取。

我们在遇到fastq文件的时候，首先要对文件加锁，文件的唯一标识符在inode节点上，我们在inode结构体中添加一个spinlock\_t类型的锁fqlock，这个类型通过spin\_lock()函数加锁，通过spin\_unlock()解锁。

在加锁之后，我们首先使用read\_ready判断是否解压完毕，如果没有解压完毕，我们使用real\_vfs\_write()将操作符rd和文件的路径写入kernel\_info的文件中，然后不断调用read\_ready()判断是否伺服端完成操作，返回为true跳出循环。（我们在read\_ready内部进行解锁）

此时，我们已经有了fastq文件对应的dcmp隐藏文件，我们将读取fastq文件的操作，重定向到dcmp文件，使用real\_vfs\_read()进行真正的读取即可。

###### 4.1.2 vfs\_write()

对于vfs\_write的修改主要如下。首先也是判断后缀是否为fq文件，如果不是，则调用real\_vfs\_write()执行即可。如果是fq文件，我们将重定向到其对应的dcmp文件中，即直接将内容写入到dcmp文件中，此后，我们将标识符wt和文件路径写入到kernel\_info文件中。压缩操作在伺服端执行。

###### 4.1.3 vfs\_rename()

对于vfs\_rename的修改主要如下。在vfs\_rename中，我们需要用到的参数只有 dentry类型的old\_dentry和new\_dentry，其他参数都是原本的vfs\_rename()函数需要使用的，我们并不关心。

当然，我们需要留一个后门，方便伺服端进行操作，我们允许cmp后缀的文件可以直接编程fq后缀的文件，不会调用伺服端进行压缩。而其他后缀的文件变成fq文件，我们会通过伺服端进行压缩。

如果原文件后缀不为cmp，现文件后缀为fq，我们将标识符mv和文件的oldpath以及现在的路径都写入到kernel\_info文件中，然后其他操作都是原来的vfs\_rename进行的。伺服端负责处理其他的操作。

##### 4.2 伺服端程序

伺服端程序我们使用python编写了一个watchdog程序watch.py，对kernel\_info中的文件进行监控，并将完成状态写入到kernel\_op中。

如果标识符为rd，即我们要执行读取操作，我们将fq文件重命名为对应的cmp文件，然后我们使用fqz\_comp进行解压，最后我们将cmp文件重命名为fq文件（这就是之前留的后门的作用）。完成后，我们在op文件中写1，标识dcmp文件存在。

如果标识符为wt，即我们要执行写操作，我们在op中写1，标识dcmp文件可以写。

如果标识符为mv，即重命名操作，我们分为两种情况，一种是将fastq文件重命名或移动，另一种是将其他文件重命名为fastq文件。第一种情况，我们要分为dcmp存在和不存在两种情况，如果存在，就将dcmp移动到新的文件夹，再进行压缩，不存在，就直接移动fq文件，当然我们需要将kernel\_op中的文件进行变更的处理保证新的文件对应的kernel\_op是正确的。而对于第二种情况，我们需要进行移动，压缩和重命名这三步操作。当然我们还需要向op中写1进行标识。

此外，我们谢了一个check\_dcmp\_file()的函数，保证一段时间如果fq文件没有访问，那么我们将对应的dcmp隐藏文件删除。

##### 4.3 使用netlink的方法

使用netlink的时候，我们暂时只实现了vfs\_read()部分。在vfs层，我们又添加了send\_usrmsg()和netlink\_rcv\_msg()这两个函数。

send\_usrmsg()的函数声明是int send\_usrmsg(char\* pbuf, uint16\_t len)，这个函数负责向伺服端发送socket，首先使用nlmsg\_new()，nlmsg\_new()会新申请一个socket buffer ，其大小为socket消息头大小+netlink 消息头大小+用户消息大小。然后使用nlmsg\_put()函数，nlmsg\_put()函数向skb缓冲区中获取消息头空间并且初始化netlink消息头。然后使用memcpy()将pbuf的内容写入nlmsg\_data中。最后使用netlink\_unicast()函数发送socket。

netlink\_rcv\_msg()的函数声明是static void netlink\_rcv\_msg(struct sk\_buff \*skb)。我们首先要通过nlmsg\_hdr()函数获取到netlink报头，然后用NLMSG\_DATA()获取socket的数据。当然，我们需要通过get\_fs()和set\_fs(get\_ds())使得我们可以在内核中访问任意的地址。然后调用copy\_from\_user()从用户端拷贝数据。最后调用set\_fs()，使得恢复到原来的状态。

当然我们声明一个全局的大于64K的缓存fq\_data\_buf，和fq\_data\_flag标记。之后我们要对每个进程都要拥有一套buf和flag。此外，我们使用struct netlink\_kernel\_cfg cfg={.input = netlink\_rcv\_msg}注册回调函数，告诉内核socket传过来的包这个函数处理，将netlink\_rcv\_msg()接收到的数据存放在fq\_data\_buf中。

在vfs\_read()中先进行发送请求操作，发送socket到伺服端，然后负责接收数据，把之前存到fq\_data\_buf中的数据，拷贝到真正的vfs\_read使用的buf中。

伺服端的程序是一个handler.c，我们没有使用python，改用c来编写。这个业务逻辑就很清楚，我们首先接收到内核发出的解压请求，然后进行解压，最后将解压后的文件存入socket由内核再接收。

## 5 接下来需要进行的工作

对于使用netlink()的方式，vfs\_read()我们暂时实现了，但是我们只考虑了单个进程的情况，只设置了一个缓冲区，需要更改每个进程一个缓冲区。此外，使用netlink的时候，我们暂时只实现了vfs\_read()部分，还有vfs\_write()和vfs\_rename()需要我们实现。当然我们在多进程多线程操作的时候，我们也需要进行加锁。

## 附录

##### 1 ramdisk中的几个文件

###### 1.1 ramdisk/etc/fstab

# /etc/fstab

proc /proc proc defaults 0 0

sysfs /sys sysfs defaults 0 0

devtmpfs /dev devtmpfs defaults 0 0

###### 1.2 ramdisk/etc/inittab

::sysinit:/etc/init.d/rcS

::askfirst:-/bin/sh

::restart:/sbin/init

::ctrlaltdel:/sbin/reboot

::shutdown:/bin/umount -a -r

::shutdown:/sbin/swapoff -a

###### 1.3 ramdisk/etc/init.d/rcS

#!/bin/sh

mount proc

mount -o remount,rw /

mount -a

clear

###### 1.4 ramdisk/init

#!/bin/busybox sh

mount –t proc none /proc

mount –t sysfs none /sys

exec /sbin/init