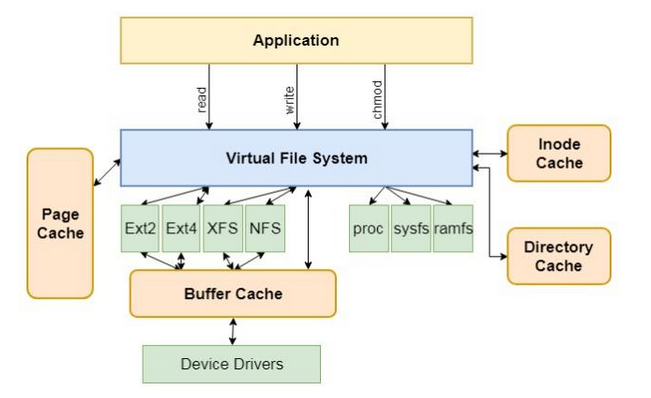
简单介绍该通过800行代码实现的一个文件系统.

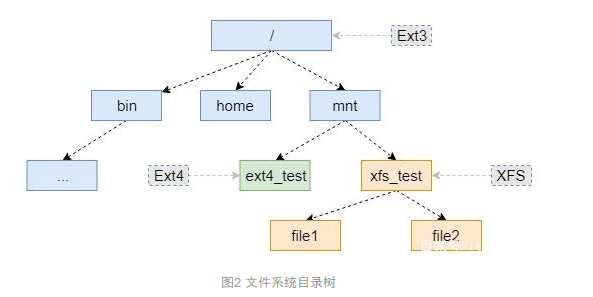
首先来聊下Linux文件系统,linux一切皆为文件,想必了解过Linux系统的人都知道.个人认为文件系统的就是实现linux系统和用户的文件高效的组织管理起来.让用户很直观的看到各个文件的名字,路径,权限.

了解Linux文件系统首先要知道Linux中的虚拟文件系统(VFS),为什么有VFS的出现呢,让我们想象一下,linux的文件系统种类非常多,例如ext2,ext3,ext4.VFAT,XFS,FUSE等.想必每一个文件系统提供的接口函数肯定有不一样的.好比你写了一个应用,里面创建了一个文件调用了文件系统里的函数.当后面你想移植到另外一个平台上用的不同的文件系统上时,你原先调用的先前文件系统的函数可能就用不了.linux想必不会这样干,为了实现用户程序的跨平台性.LINUX提供了统一的文件操作函数.也就是vfs.让用户完全不必关系底层的实现细节.VFs还实现了Inode cache,directory cache,避免多个文件系统重复实现.



VFS虚拟文件系统还可以让多个文件系统关联起来.比如该系统本身是EXT3文件系统.

在它mnt下分别挂载了ext4文件系统和,xfs文件系统.具体实现原理可以去深入研究,本人还是个菜鸟对这块不熟悉,就不分析了.



接下来来说下文件.linux文件,了解文件要了解以下3个结构体,inode,file.和file\_operation

struct inode {

umode\_t i\_mode;

unsigned short i\_opflags;

kuid\_t i\_uid;

kgid\_t i\_gid;

unsigned int i\_flags;

const struct inode\_operations \*i\_op;

struct super\_block \*i\_sb;

struct address\_space \*i\_mapping; /\*

union {

const unsigned int i\_nlink;

unsigned int \_\_i\_nlink;

};

dev\_t i\_rdev;

loff\_t i\_size;

struct timespec i\_atime;

struct timespec i\_mtime;

struct timespec i\_ctime;

spinlock\_t i\_lock; /\* i\_blocks, i\_bytes, maybe i\_size \*/

unsigned short i\_bytes;

unsigned int i\_blkbits;

blkcnt\_t i\_blocks;

struct hlist\_node i\_hash;

struct list\_head i\_io\_list; /\* backing dev IO list \*/

#ifdef CONFIG\_CGROUP\_WRITEBACK

struct bdi\_writeback \*i\_wb; /\* the associated cgroup wb \*/

/\* foreign inode detection, see wbc\_detach\_inode() \*/

int i\_wb\_frn\_winner;

u16 i\_wb\_frn\_avg\_time;

u16 i\_wb\_frn\_history;

#endif

struct list\_head i\_lru; /\* inode LRU list \*/

struct list\_head i\_sb\_list;

union {

struct hlist\_head i\_dentry;

struct rcu\_head i\_rcu;

};

u64 i\_version;

atomic\_t i\_count;

atomic\_t i\_dio\_count;

atomic\_t i\_writecount;

const struct file\_operations \*i\_fop; /\* former ->i\_op->default\_file\_ops \*/

struct address\_space i\_data;

union {

struct pipe\_inode\_info \*i\_pipe;

struct block\_device \*i\_bdev;

struct cdev \*i\_cdev;

char \*i\_link;

};

void \*i\_private; /\* fs or device private pointer \*/

};

i\_mode 文件权限

const struct inode\_operations inode的操作函数,creat,link,unlink,mkdir,rmdir,反正就是对文件的各种操作函数.

struct super\_block \*i\_sb 指向超级块的指针.

struct address\_space \*i\_mapping; 指向地址空间的指针,address\_space结构体中有一个radix\_tree radix数据结构的树.主要用来标记读取的文件在pagecache中是否命中.

const unsigned int i\_nlink 硬链接的计数.

struct timespec i\_atime; 最后access的时间

struct timespec i\_mtime; 最后modify的时间

struct timespec i\_ctime; 最后change的时间

atomic\_t i\_writecount; 写该文件的计数.

union {

struct pipe\_inode\_info \*i\_pipe;

struct block\_device \*i\_bdev;

struct cdev \*i\_cdev;

char \*i\_link;

};

特殊文件,指向这些特殊文件结构的指针.pipe ,blockdevice, cdev,link文件

void \*i\_private; /\* fs or device private pointer \*/ iNode 私有数据

struct file {

union {

struct llist\_node fu\_llist;

struct rcu\_head fu\_rcuhead;

} f\_u;

struct path f\_path;

struct inode \*f\_inode; /\* cached value \*/

const struct file\_operations \*f\_op;

/\*

\* Protects f\_ep\_links, f\_flags.

\* Must not be taken from IRQ context.

\*/

spinlock\_t f\_lock;

atomic\_long\_t f\_count;

unsigned int f\_flags;

fmode\_t f\_mode;

struct mutex f\_pos\_lock;

loff\_t f\_pos;

struct fown\_struct f\_owner;

const struct cred \*f\_cred;

struct file\_ra\_state f\_ra;

u64 f\_version;

#ifdef CONFIG\_SECURITY

void \*f\_security;

#endif

/\* 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;

struct list\_head f\_tfile\_llink;

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

struct address\_space \*f\_mapping;

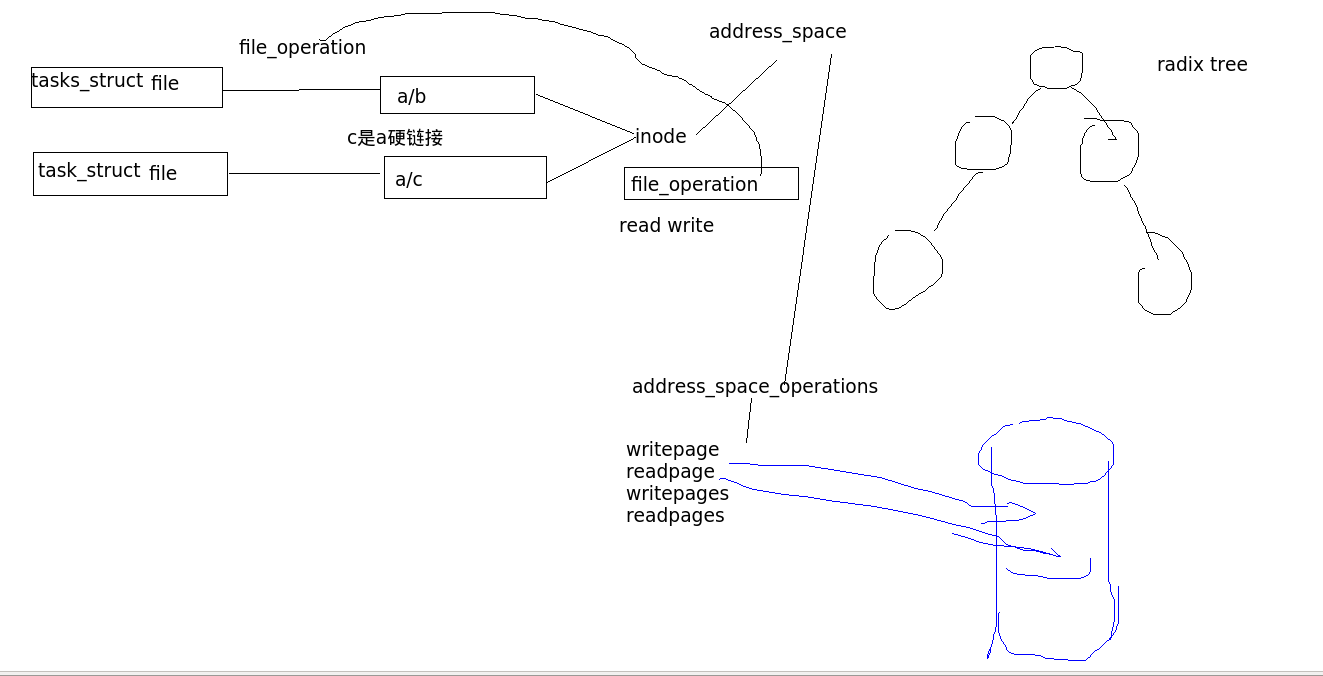
}

Linux中每打开一个文件都会对应创建一个file结构体.即便有两个进程打开同一个文件.即便是同一个文件,但是访问的偏移地址不同.用同一个file结构体无法满足记录当前访问的偏移地址

struct path f\_path. 当打开一个路径文件后,该指针存储传入的路径.

struct inode \*f\_inode; 指向该文件的inode

const struct file\_operations \*f\_op; 文件操作函数,当文件OPEN时候会被填充,如果是字符设备,该指针就指向字符设备的驱动函数.



进程A和进程B打开了目录A下面的B和C文件.但是BC软件是个硬链接.它们对应同一个inode, inode结构体有个addressspace结构体,addressspace里有一个radix tree ,用来标记读取的文件是否在pagecache中命中.如果命中了直接返回数据给用户,如果没有命中,调用addressspace 里面的address\_space\_operation有读取页面的操作函数.去进行IO读写.

用户层

Oepn() Read() write()

VFS(虚拟文件系统)

块设备

块设备驱动

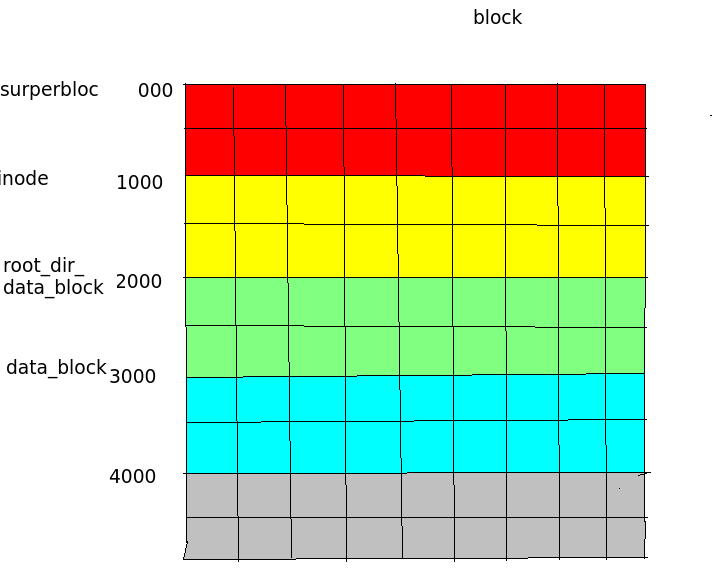
通用块层

文件系统(EXT4,EXT3,EXT2,FUSE,VFAT)

pagecache

Sys\_read sys\_write sys\_open

Simplefs 是由国外的一个人写的一个文件系统,该文件系统只用了800多行代码实现了,文件的管理,不过目前还未实现文件删除的功能,现在我准备在这个文件系统添加一些代码,实现文件删除功能.



**该文件系统在物理上的存储结构如图.surperblock(超级块)在block的0地址结构体如下**

**struct simplefs\_super\_block {**

**uint64\_t version; //版本号**

**uint64\_t magic; //对应的镜像号**

**uint64\_t block\_size; //block 大小**

**/\* FIXME: This should be moved to the inode store and not part of the sb \*/**

**uint64\_t inodes\_count; //总inode数量**

**uint64\_t free\_blocks; //空闲的blocks**

**char padding[SIMPLEFS\_DEFAULT\_BLOCK\_SIZE - (5 \* sizeof(uint64\_t))]; //预留**

**};**

**该文件系统inode结构在1000位置**

**struct simplefs\_inode {**

**mode\_t mode; //权限**

**uint64\_t inode\_no; //inode号**

**uint64\_t data\_block\_number; //数据块对应的block号**

**union {**

**uint64\_t file\_size; //文件大小**

**uint64\_t dir\_children\_count; //如果是目录对应的目录下的文件数,如果不是目录则为0**

**};**

**};**

**目录对应的数据结构**

**struct simplefs\_dir\_record {**

**char filename[SIMPLEFS\_FILENAME\_MAXLEN]; //文件名字**

**uint64\_t inode\_no; //文件inode号**

**};**

创建一个文件.首先判断是否是个目录,还是文件,如果是文件.

ret = simplefs\_sb\_get\_objects\_count(sb, &count);

获取当前总inode数量.

inode = new\_inode(sb);

创建新的一个inode