目录

[1.I/O模型 2](#_Toc49346164)

[1.1 阻塞与非阻塞 2](#_Toc49346165)

[1.2 多路复用 3](#_Toc49346166)

[1.3 异步IO 5](#_Toc49346167)

[(1) C库提供的Glibc-AIO 5](#_Toc49346168)

[(2) 内核提供的Kernel-AIO： 6](#_Toc49346169)

[1.4 Libevent事件触发 6](#_Toc49346170)

[2文件系统 7](#_Toc49346171)

[2.1 EXT2文件系统总体存储布局 7](#_Toc49346172)

[2.3数据块寻址 14](#_Toc49346173)

[2.4文件和目录操作的系统函数 15](#_Toc49346174)

[2.5 VFS: 17](#_Toc49346175)

[2.6 icache/dcache, 18](#_Toc49346176)

[2.7 fuse 18](#_Toc49346177)

[3 文件系统一致性 19](#_Toc49346178)

[3.1掉电与文件系统一致性 19](#_Toc49346179)

[3.2append一个文件的全流程 19](#_Toc49346180)

[3.3模拟文件系统不一致性案例 20](#_Toc49346181)

[3.4 fsck 22](#_Toc49346182)

[3.6文件系统的调试工具 24](#_Toc49346183)

[3.7 Copy On Write文件系统: btrfs 25](#_Toc49346184)

[4 块I/O流程与I/O调度器 26](#_Toc49346185)

[4.1一个块IO的一生 26](#_Toc49346186)

[4.2 O\_DIRECT和O\_SYNC 28](#_Toc49346187)

[4.3 BIO 流程blktrace 29](#_Toc49346188)

[4.4 IO调度算法，CFQ和ionice 31](#_Toc49346189)

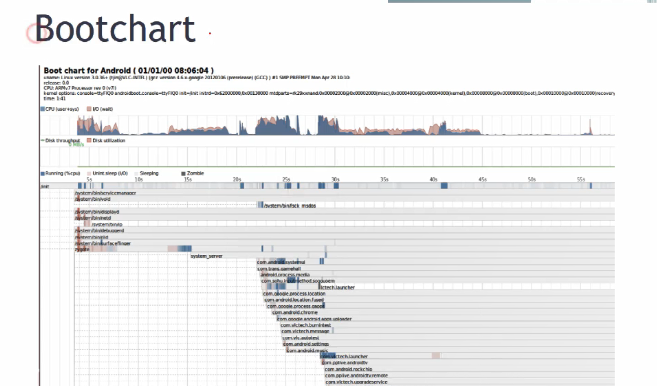
[4.5 cgroup与IO 32](#_Toc49346190)

[4.6 IO性能调试：iotop, iostat 32](#_Toc49346191)

# 1.I/O模型

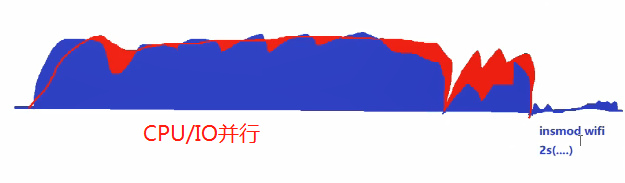
在业务执行过程中，常伴随大量的IO操作，如果IO操作和CPU消耗不能合理安排，将会导致整体业务执行效率低下，用户体验极差。

比如手机启动过程，有大量CPU消耗和IO操作。用Bootchart记录android启动过程的CPU/IO消耗如下图



**Systemd readahead:**

Systemd readahead-collect.service搜集系统启动过程中的文件访问信息，Systemd readahead-replay.service在后续启动过程中完成回放，即将IO操作与CPU并行；



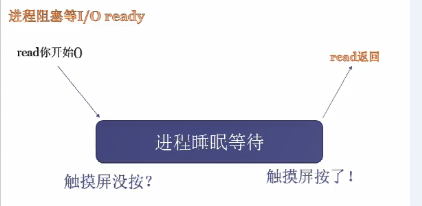
提高效率的一个宗旨，把CPU和IO的交替等，变为CPU和IO操作(不需要CPU参与)同时工作，充分利用系统资源；

为解决CPU/IO并行问题，Linux提供了很多IO模型；

## 1.1 阻塞与非阻塞

**(1)阻塞：**一般来说，进程阻塞，等待IO条件满足才返回；

有个例外，阻塞可以被信号打断；





若设置信号标记，act.sa\_flags |= SA\_RESTART; 接收信号，read阻塞不返回，但是信号响应函数还是会调用；相当于系统自动重新进入阻塞；

用signal()函数设置信号，其调用sigaction自动设置SA\_RESTART；

**（2）非阻塞**

read/write等IO调用，IO设备没就绪，立即返回，实际工程上用的不多；

## 1.2 多路复用

实际业务中，一般有多个IO请求，每个请求响应都用简单的阻塞模型效率太低，Linux提供了多路复用的的系统调用：

**(1) select**



**select()处理流程**

a.告诉系统，要关注哪些IO请求；

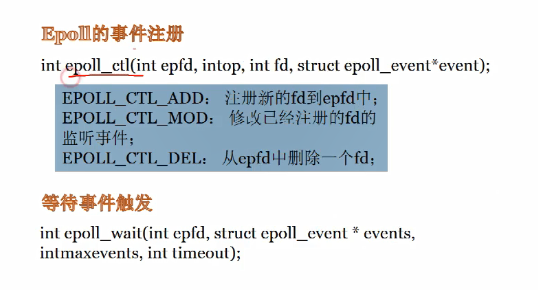
b.阻塞等待，直到有IO就绪，select返回；

c.主动查询是哪个IO就绪，然后响应该IO;

d.重新关注新的IO请求；

当IO请求过多时，这种查询的方式也很浪费资源，因此Linux提供了一个新的系统调用

**(2)epoll()**



**epoll与select的不同：**

a.将注册IO请求和等待事件触发分离开；

b.返回后，直接告诉哪些IO就绪，**不用再主动查询**；



当IO数量不多时，可以用select或epoll，但当IO非常多时，比如大型网络应用，响应多个IO请求时，用epoll效率远高于select。

signal io方式，都是read/write阻塞，底层实现，待IO就绪后，内核发送信号，唤醒阻塞；

比如读触摸屏应用，read被阻塞，只有触摸屏被按下，触发中断程序响应，读取触摸屏行为数据后，内核发送信号唤醒APP的等待，APP读到触摸动作信息，做相应业务处理。

目前工程上，处理异步I/O更多用以下方法

## 1.3 异步IO



### (1) C库提供的Glibc-AIO

Glibc-AIO原理，aio\_read()立即返回，后台自动创建线程读取io，aio\_suspend()查询IO是否完成，完成立即返回，未完成，等待；

### (2) 内核提供的Kernel-AIO：

一般用来读取硬盘数据，比如数据库读取；

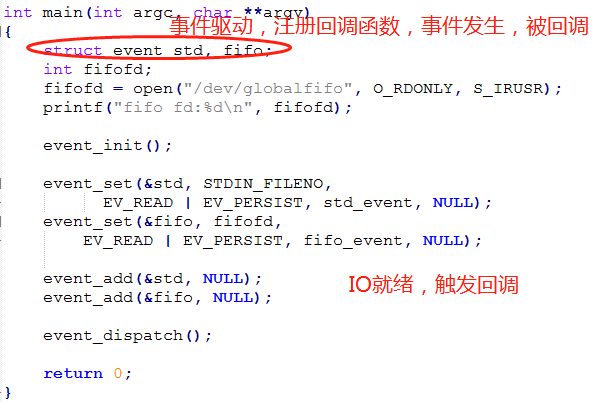
这些异步模型，天然的将IO与CPU消耗等待做并行处理；

## 1.4 Libevent事件触发

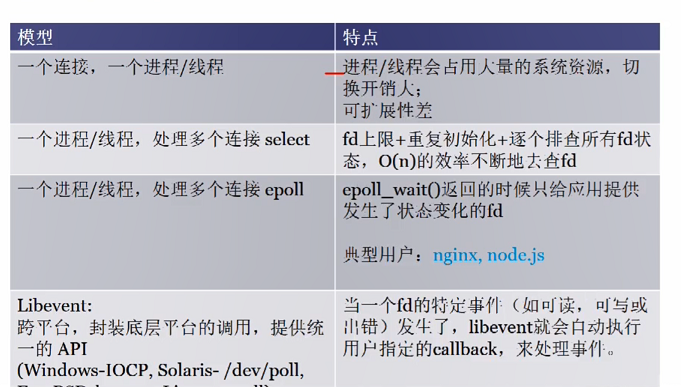
功能类似QT/VC的按钮，注册回调函数，当事件触发时，执行回调函数。

libevent是一个跨平台库，封装底层平台调用，提供统一API。Windows/Solaris/linux。

gcc xxx.c -levent



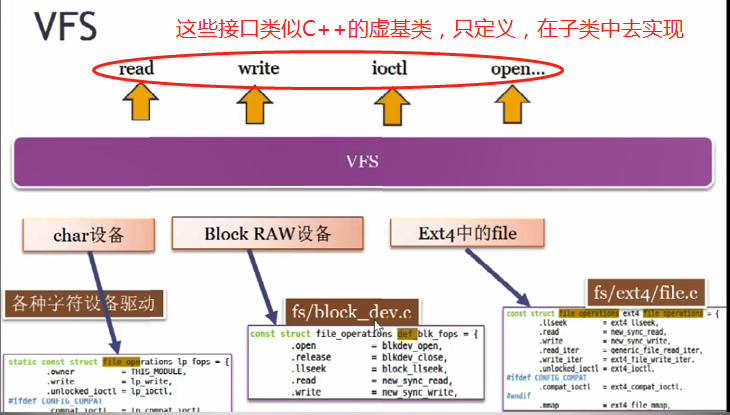
模型对比：



C10K问题：<http://www.kegel.com/c10k.html>

# 2文件系统

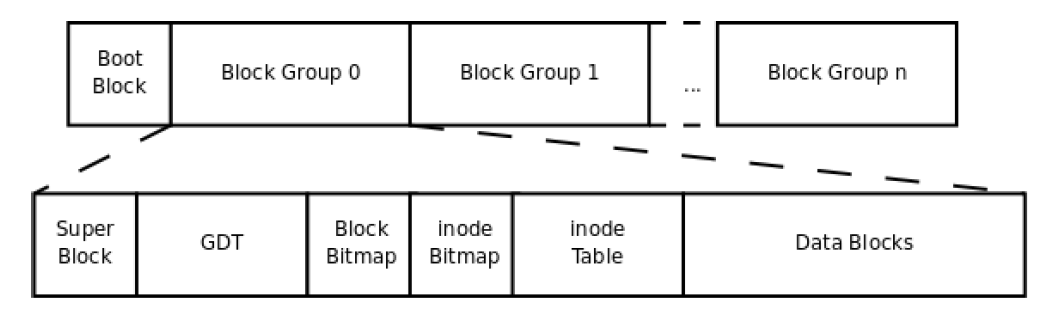
一切都是文件，Linux通过VFS中间层，支持多种文件系统，对APP统一接口；



文件系统的本质是将用户数据和元数据(管理数据的数据)，组织成有序的目录结构。

## 2.1 EXT2文件系统总体存储布局

一个磁盘可以划为多个分区，每个分区必须先用格式化工具(某种mkfs)格式化成某种格式的文件系统，然后才能存储文件，格式化的过程会在磁盘上写一些管理存储布局信息。一个典型的ext格式化文件系统存储布局如下：



文件系统最小存储单位是Block，Block大小格式化时确定，一般4K；

**启动块(BootBlock):**

大小1K，使由PC标准规定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使用启动块。启动块之后才是EXT文件系统的开始；

**超级块(Superblock)：**

描述整个分区的文件系统信息，比如块大小，文件系统版本号，上次mount时间等；

超级块在每个块组的开头都有一份拷贝；

**块组描述符表(GDT, Group Descriptor Table)**

由很多块组描述符组成，整个分区分成多少个块组就对应多少个GDT;

每个GD存储一个块组的描述信息，比如这个块组从哪里开始是inode表，哪里开始是数据块，空闲的inode和数据块还有多少个等。

和超级块类似，GDT在每个块组开头也有一份拷贝；

**块位图（Block Bitmap）**

用来描述整个块组中那些块空闲，本身占用一个块，每个bit代表本块组的一个块，bit为1表示对应块被占用，0表示空闲；

**Tips:**

df命令统计整个磁盘空间非常快，因为只需要查看每个块组的块位图即可；

du命令查看一个较大目录会很慢，因为需要搜索整个目录的所有文件；

**inode位图(inode Bitmap)**

和块位图类似，本身占用一个块，每个bit表示一个inode是否空闲可用；

**inode表(inode Table)**

每个文件对应一个inode，用来描述文件类型，权限，大小，创建/修改/访问时间等信息；

一个块组中的所有inode组成了inode表；

Inode表占用多少个块，格式化时就要确定，mke2fs工具默认策略是每8K分配一个inode。

就是说当文件全部是8K时，inode表会充分利用，当文件过大，inode表会浪费，文件过小，inode不够用；

硬链接指向同一个inode；

**数据块(Data Block)**

**a.常规文件：**

文件的数据存储在数据块中；

**b.目录**

该目录下所有文件名和目录名存储在数据块中；

文件名保存在目录的数据块中，ls –l看到的其他信息保存在该文件的inode中；

目录也是文件，是一种特殊类型的文件；

**c.符号链接**

如果目标路径名较短，直接保存在inode中以便查找，如果过长，分配一个数据块保存。

**d.设备文件、FIFO和socket等特殊文件**

没有数据块，设备文件的主，次设备号保存在inode中。

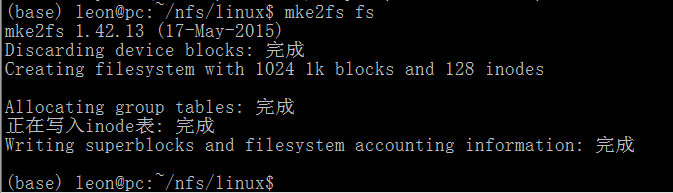
**2.2实例解析文件系统结构：**

用一个文件来模拟一个磁盘;

**1.创建一个1M文件，内容全是0；**

dd if=/dev/zero of=fs count=256 bs=4k

**2.对文件fs格式化**



格式化后的fs文件大小依然是1M，但内容已经不是全零。

**3.用dumpe2fs工具查看这个分区的超级块和块组描述表信息**

(base) leon@pc:~/nfs/linux$ dumpe2fs fs

dumpe2fs 1.42.13 (17-May-2015)

Filesystem volume name: <none>

Last mounted on: <not available>

Filesystem UUID: a00715b2-528b-4ca6-8c2b-953389a5ab00

Filesystem magic number: 0xEF53

Filesystem revision #: 1 (dynamic)

Filesystem features: ext\_attr resize\_inode dir\_index filetype sparse\_super large\_file

Filesystem flags: signed\_directory\_hash

Default mount options: user\_xattr acl

Filesystem state: clean

Errors behavior: Continue

Filesystem OS type: Linux

Inode count: 128

Block count: 1024

Reserved block count: 51

Free blocks: 986

Free inodes: 117

First block: 1

Block size: 1024

Fragment size: 1024

Reserved GDT blocks: 3

Blocks per group: 8192

Fragments per group: 8192

Inodes per group: 128

Inode blocks per group: 16

Filesystem created: Fri Aug 21 16:48:02 2020

Last mount time: n/a

Last write time: Fri Aug 21 16:48:02 2020

Mount count: 0

Maximum mount count: -1

Last checked: Fri Aug 21 16:48:02 2020

Check interval: 0 (<none>)

Reserved blocks uid: 0 (user root)

Reserved blocks gid: 0 (group root)

First inode: 11

Inode size: 128

Default directory hash: half\_md4

Directory Hash Seed: e5c519af-d42e-43b5-bc8d-c67c5a79bcbe

Group 0: (Blocks 1-1023)

主 superblock at 1, Group descriptors at 2-2

保留的GDT块位于 3-5

Block bitmap at 6 (+5), Inode bitmap at 7 (+6)

Inode表位于 8-23 (+7)

986 free blocks, 117 free inodes, 2 directories

可用块数: 38-1023

可用inode数: 12-128

(base) leon@pc:~/nfs/linux$

块大小1024字节，一共1024个块，第0块是启动块，第一个块开始才是EXT2文件系统，

Group0占据1到1023个块，共1023个块。

超级块在块1，GDT3-5，

块位图在块6，占用一个块，1024x8=8192bit，足够表示1023个块，只需一个块就够了；

inode bitmap在块11

inode表在8-23，占用16个块，默认每8K对应一个inode，共1M/8K=128个inode。每个inode占用128字节，128x128=16k

**4用普通文件制作的文件系统也可以像磁盘分区一样mount到某个目录**

$ sudo mount -o loop fs /mnt/

-o loop选项告诉mount这是一个常规文件，不是块设备，mount会把它的数据当作分区格式来解释；

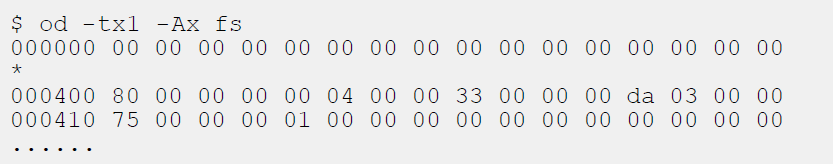
文件系统格式化后，在根目录自动生成三个字目录： ., .., lost+found

Lost+found目录由e2fsck工具使用，如果在检查磁盘时发生错误，就把有错误的块挂在这个目录下。

现在可以在/mnt 读写文件，umount卸载后，确保所有改动都保存在fs文件中了。

**5.解读fs二进制文件**

od –tx1 –Ax fs

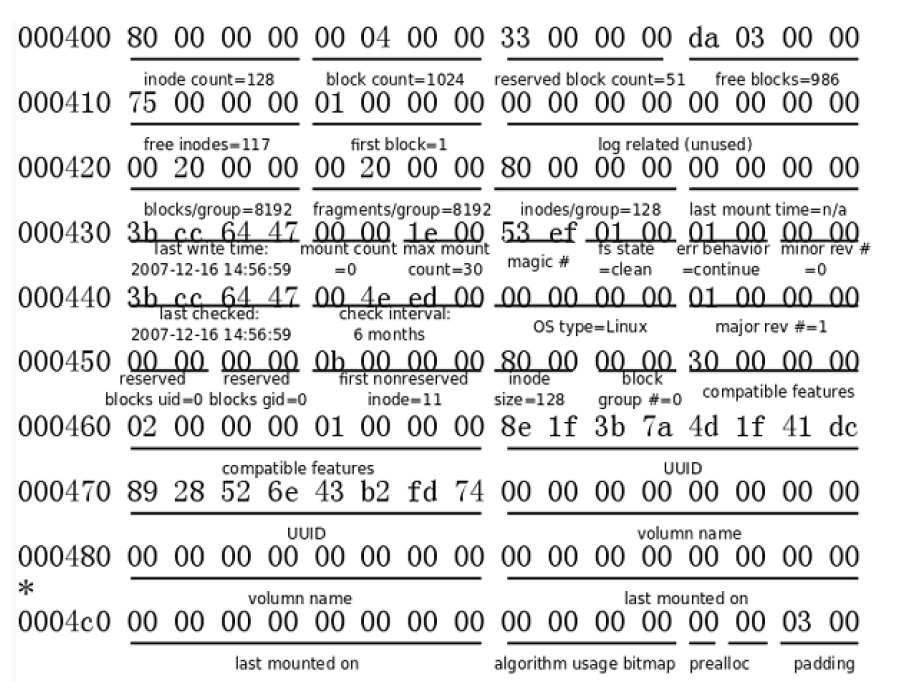


\*开头行表示省略全零数据。

000000开始的1KB是启动块，由于不是真正的磁盘，这里全零；

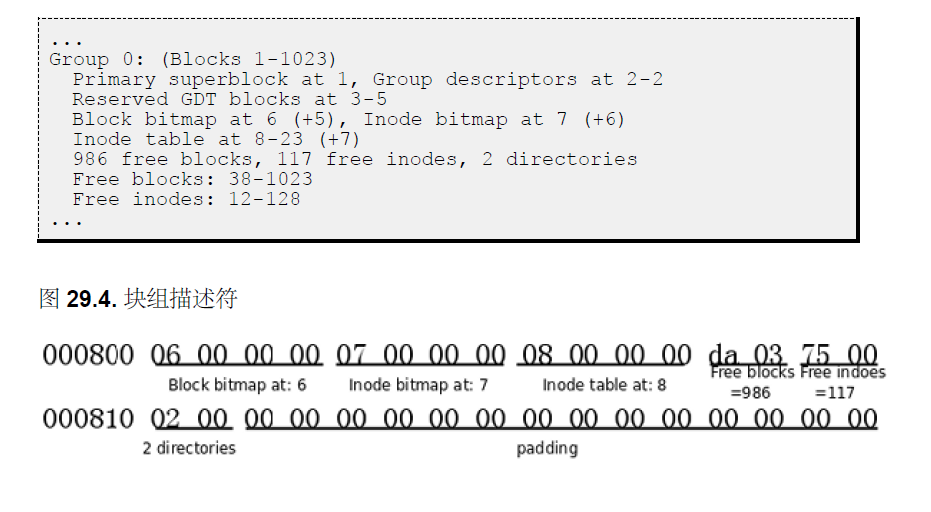
000400到0007ff是1KB的超级块，对照dumpe2fs输出信息对比如下：

超级块：



Ext2各字段按小端存储。

块组描述符



整个文件系统1M，每个块1KB，一共1024个块，除了启动块0，其他1-1023全部属于group0.

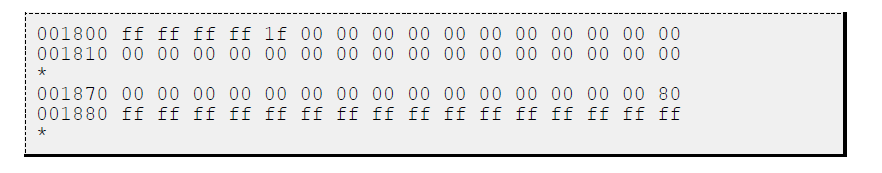
Block1是超级块，

块位图Block6，

inode位图Block7，

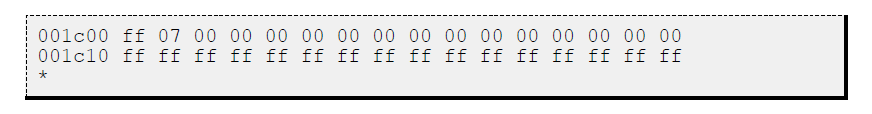
inode表从Block8开始，由于超级块中指出每个块组有128inode，每个inode大小128字节，因此共占用16个块（8-23）从Block24开始就是数据块。

查看块位图，6x1024=0x1800



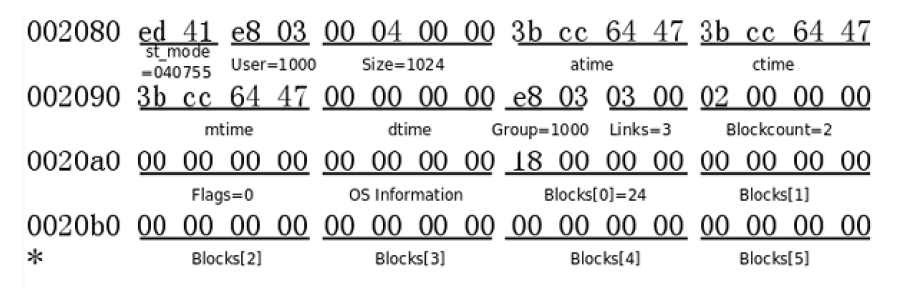
前37位（ff ff ff ff 1f）已经被占用，空闲块是连续的Block38-1023=986 free blocks

查看inode位图，7x1024=0x1c00



已用11个inode中，前10个inode是被ext2文件系统保留的，其中第二个inode是根目录，第11个inode是lost+found目录。块组描述符也指出该组有两个目录，就是根目录和lost+found目录。

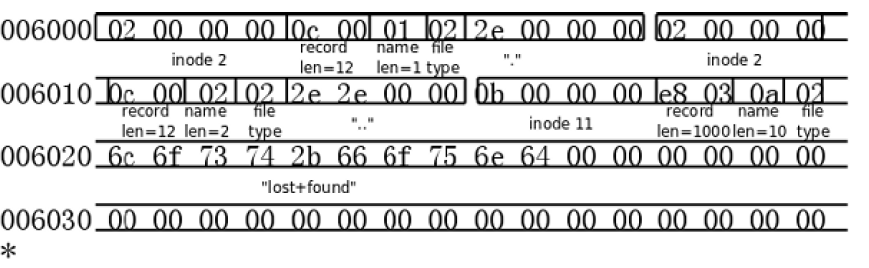
解析根目录的inode，地址Block8\*1024+inode2(1\*128)=0x2080



st\_mode用八进制表示，包含了文件类型和权限，最高位4表示为文件类型目录，755表示权限，size是大小，说明根目录现在只有一个块。Links=3表示根目录有三个硬链接，分别是根目录下的., ..和lost+found字目录下的..，

这里的Blockcount是以512字节为一个块统计的，磁盘最小读写单位一个扇区(Sector)512字节，而不是格式化文件系统时指定的块大小。所以Blockcount是磁盘的物理块数量，而不是分区的逻辑块数量。

根据上图Block[0]=24块，在文件系统中24x1024字节=0x6000，从od命令查找0x6000地址



目录数据块由许多不定长记录组成，每条记录描述该目录下的一个文件；

记录1，inode号为2，就是根目录本身，记录长12字节，文件名长度1(“.”)，类型2;

记录2，inode号为2，也是根目录本身，记录长12字节，文件名长度(“..”)，类型2;

记录3，inode号为11，记录长1000字节，文件名长度(”lost+found”)，类型2;

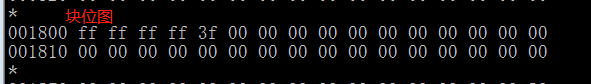
debugfs命令，不需要mount就可以查看这个文件系统的信息

debugfs fs

stat / cd ls 等

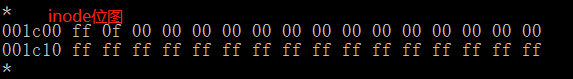
将fs挂载，在根目录创建一个hello.txt文件，写入内容”hello fs!”，重新解析根目录

查看块位图



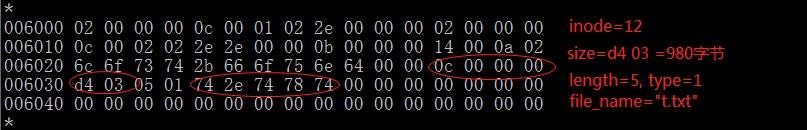
可见前38bit被占用，第38块地址38x1027=0x9800

查看inode位图，7x1024=0x1c00



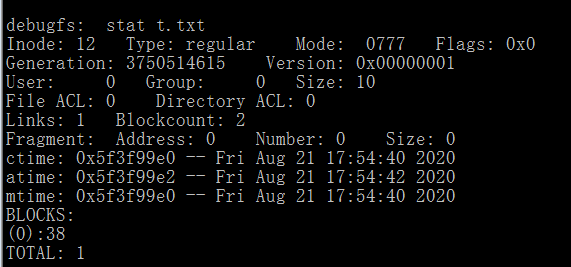
由图知，用掉了12个inode

查看根目录的数据块内容



debug fs查看t.txt属性

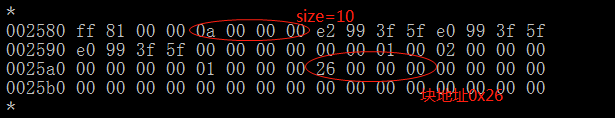
stat t.txt



t.txt文件inode号=12

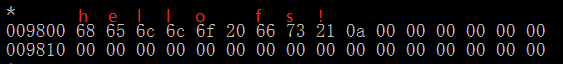
Inode12的地址=Block8\*1024+inode12(11\*128)=0x2580

查看t.txt的inode



文件大小10字节=stlen(“hello fs!”)，数据块地址0x26x1024 = 0x9800

查看内容



## 2.3数据块寻址

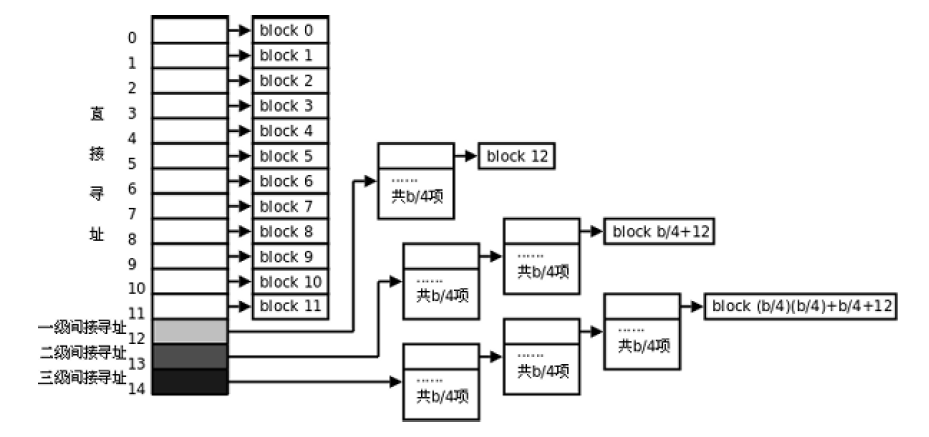
如果一个文件很大，有多个数据块，这些块可能不是连续存放的，那如何寻址所有块呢？

在上面根目录数据块是通过inode的索引项Blocks[0]找到的，实际上这样的索引项一共有15个，从Blocks[0]到Blocks[14]，每个索引项占4字节，前12个索引项都表示块编号，例如上面Blocks[0]保存块24，如果块大小是1KB，这样就可以表示12KB的文件，剩下的三个索引项Blocks[12]~ Blocks[14]，如果也这么用，就只能表示最大15KB文件，这远远不够。

实际上剩下的这3个索引项都是间接索引，Blocks[12]所指向的间接寻址块(Indirect Block)，其中存放类似Blocks[0]这种索引，再由索引项指向数据块。假设块大小是b，那么一级间接寻址加上之前的12个索引项，最大可寻址b/4+12个数据块=1024/4+12=268KB的文件。

同理Blocks[13]作为二级寻址，最大可寻址(b/4)\*(b/4)+12=64.26MB

Blocks[14]作为三级寻址，最大可寻址(b/4)\*(b/4) \*(b/4)+12=16.06GB

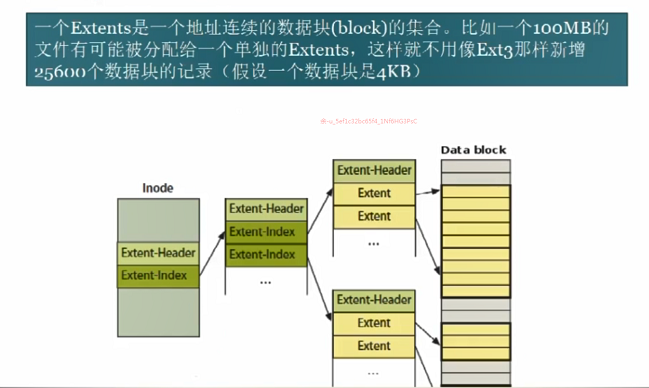


可见，这种寻址方式对于访问不超过12数据块的小文件，是非常快的。访问任意数据只需要两次读盘操作，一次读Inode，一次读数据块。

而访问大文件数据最多需要5次读盘操作，inode, 一级寻址块、二级寻址块、三级寻址块、数据块。

实际上磁盘中的inode和数据块往往会被内核缓存，读大文件的效率也不会太低。

在EXT4，支持Extents，其描述连续数据块的方式，可以节省元数据空间。



## 2.4文件和目录操作的系统函数

Linux提供一些文件和目录操作的常用系统函数，文件操作命令比如ls, cp,mv等都是基于这些系统调用实现的。

**stat:**

读取文件的inode, 把inode中的各种文件属性填入struct stat结构体返回；

假如读一个文件/opt/file，其查找顺序是

1.读出inode表中第2项，也就是根目录的inode，从中找出根目录数据块的位置

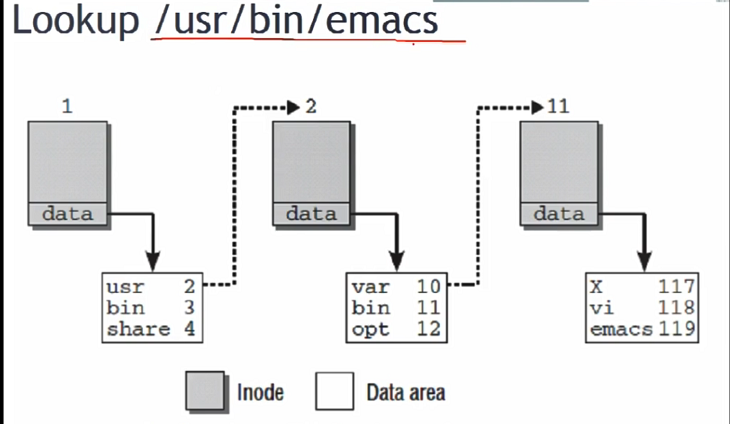
2.从根目录的数据块中找出文件名为opt的记录，从记录中读出它的inode号

3. 读出opt目录的inode，从中找出它的数据块的位置

4. 从opt目录的数据块中找出文件名为file的记录，从记录中读出它的inode号

5.读出file文件的inode

还有另外两个类似stat的函数：fstat(2)函数，lstat(2)函数



**access(2)：**

检查执行当前进程的用户是否有权限访问某个文件，access去取出文件inode中的st\_mode字段，比较访问权限，返回0表示允许访问，-1不允许。

**chmod(2)和fchmod(2)：**

改变文件的访问权限，也就是修改inode中的st\_mode字段。

chmod(1)命令是基于chmod(2)实现的。

**chown(2)/fchown(2)/lchown(2):**

改变文件的所有者和组，也就是修改inode中的User和Group字段。

**utime(2):**

改边文件访问时间和修改时间，也就是修改inode中的atime和mtime字段。touch(1)命令是基于utime实现的。

**truncate(2)/ftruncate(2)：**

截断文件，修改inode中的Blocks索引项以及块位图中的bit.

**link(2)：**

创建**硬链接**，就是在目录的数据块中添加一条记录，其中的inode号字段与源文件相同。

**syslink(2)：**

**创建符号链接，需要创建一个新的inode**，其中st\_mode字段的文件类型是符号链接。指向路径名，不是inode，替换掉同名文件，符号链接依然可以正常访问。

ln(1)命令是基于link和symlink函数实现的。

**unlink(2):**

删除一个链接，如果是符号链接则释放符号链接的inode和数据块，清除inode位图和块位图中相应位。如果是硬链接，从目录的数据块中清除文件名记录，如果当前文件的硬链接数已经是1，还要删除它，同时释放inode和数据块，清除inode位图和块位图相应位，这时文件就真的删除了。

**rename(2):**

修改文件名，就是修改目录数据块中的文件名记录，如果新旧文件名不在一个目录下，则需要从原目录数据中清楚记录，然后添加到新目录的数据块中。mv(1)命令是基于rename实现的。

**readlink(2):**

从符号链接的inode或数据块中读出保存的数据。

**rmdir(2):**

删除一个目录，目录必须是空的(只含.和..)才能删除，释放它的inode和数据块，清除inode位图和块位图的相应位，清除父目录数据块中的记录，父目录的硬链接数减1，rmdir(1)命令是基于rmdir函数实现的。

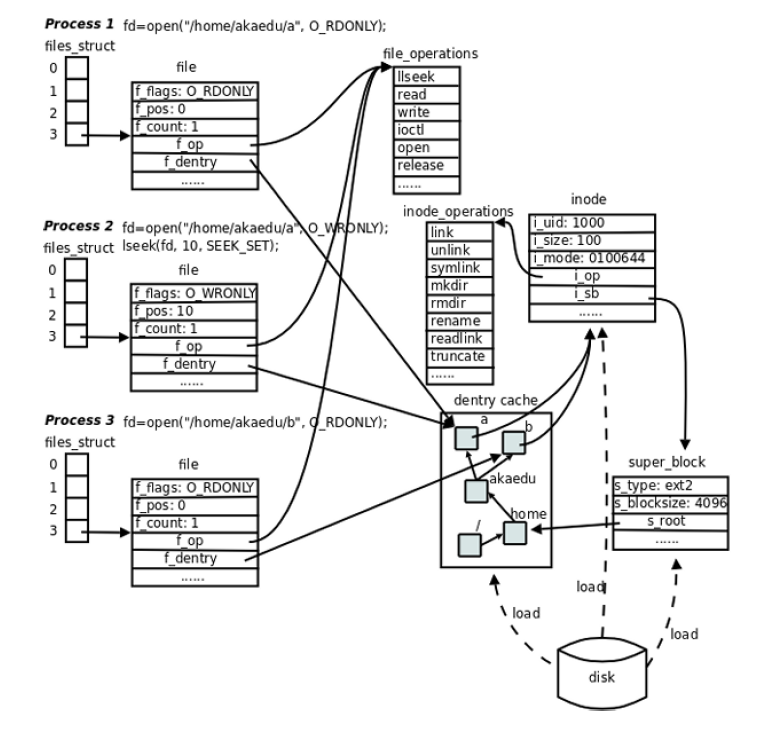
**opendir(3)/readdir(3)/closedir(3)：**

用于遍历目录数据块中的记录。

目录，是一个特殊的文件，其存放inode号与文件名的映射关系；

## 2.5 VFS:

Linux支持各种文件系统格式，ext2,ext3,ext4,fat,ntfs,yaffs等，内核在不同的文件系统格式之上做了一个抽象层，使得文件目录访问等概念成为抽象层概念，对APP提供统一访问接口，由底层驱动去实现不同文件系统的差异，这个抽象层叫虚拟文件系统（VFS, Virtual Filesystem）。



File,dentry,inode,super\_block这几个结构体组成了VFS的核心概念。

2.6 icache/dcache,

访问过的文件或目录，内核都会做cache;

inode\_cachep = kmem\_cache\_create()

dentry\_cache=KMEM\_CACHE()

这两个函数申请的slab可以回收，内存自动释放；

Linux配置回收优先级

(1).free pagecache:

echo 1 >> /proc/sys/vm/drop\_caches

(2)free reclaimable slab objects (includes dentries and inodes)

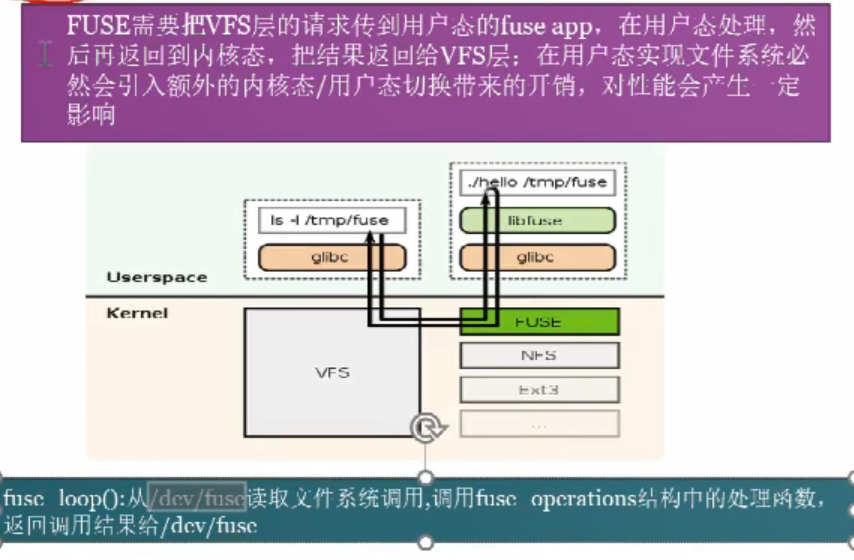
echo 2 >> /proc/sys/vm/drop\_caches

(3)free slab objects and pagecache:

echo 3 >> /proc/sys/vm/drop\_caches

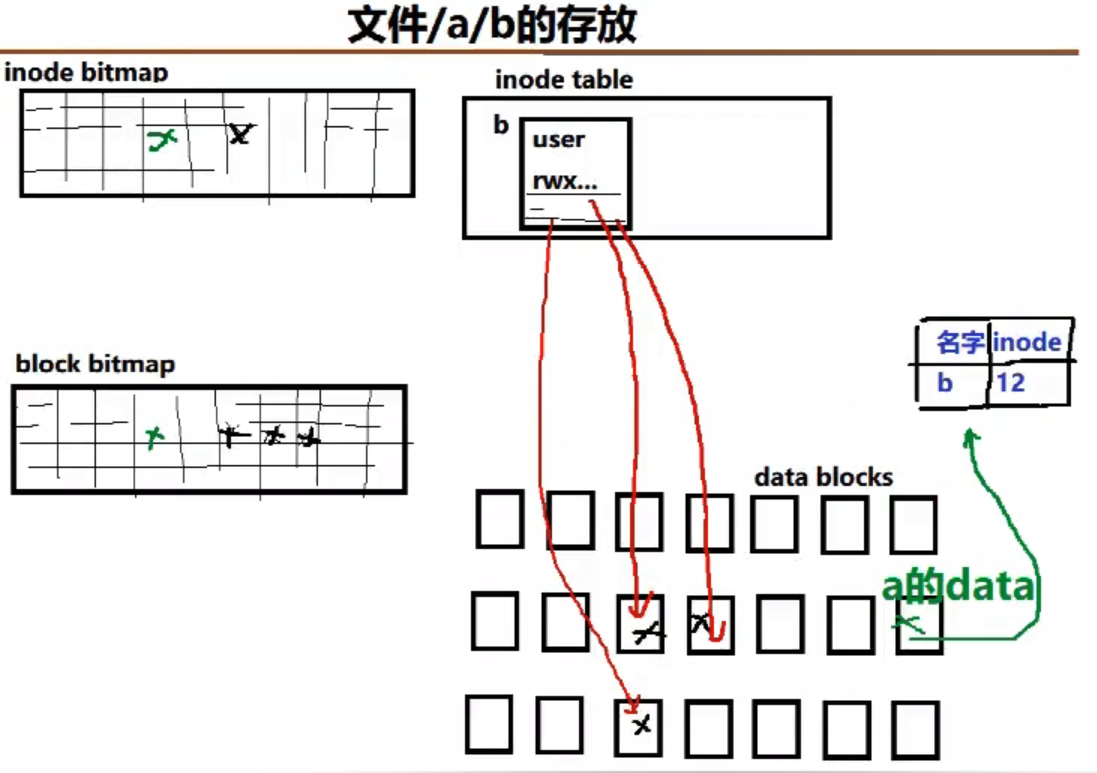
## 2.7 fuse

Linux支持用户空间实现文件系统，fuse实际上是把内核空间实现的VFS支持接口，放到用户层实现。

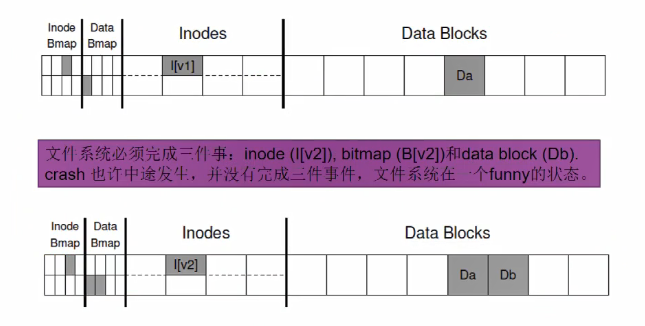


# 3 文件系统一致性

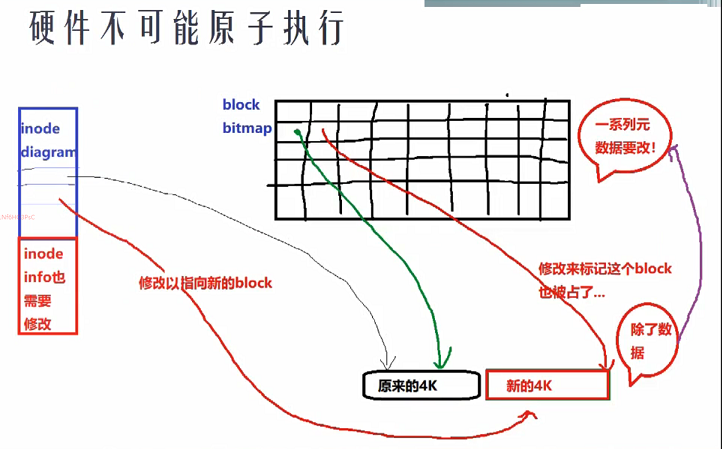
3.1掉电与文件系统一致性  
由上一节文件系统的布局分析可知，当操作一个文件时，比如往/a目录下添加一个b，即添加/a/b文件，需要修改inode bitmap, inode table, block bitmap, data block。

  
这一系列的操作是非原子的，假如任何一个环节掉电，造成某些步骤丢失，就会造成数据的不完整，文件将无法正常访问。

## **3.2append一个文件的全流程**



而硬件是不可能原子执行的，因此会造成不一致性。



## **3.3模拟文件系统不一致性案例**

**(1). 做一个image，用来模拟磁盘**

dd if=/dev/zero of=image bs=1024 count=4096

**(2).格式化为ext4文件系统**

mkfs.ext4 -b 4096 image

**(3).mount到test目录，写入一个ok.txt文件**

sudo mount -o loop image test/

cd test/

sudo touch ok.txt

cd ..

sudo umount test

**(4).查看磁盘详细信息**

(base) leon@pc:~/io$ dumpe2fs image

dumpe2fs 1.42.13 (17-May-2015)

Filesystem volume name: <none>

Last mounted on: /home/leon/io/test

Filesystem UUID: 759835e3-9508-4c57-b511-9c4f7e13f0ad

Filesystem magic number: 0xEF53

Filesystem revision #: 1 (dynamic)

…

Inode count: 1024

Block count: 1024

Reserved block count: 51

Free blocks: 982

Free inodes: 1012

First block: 0

Block size: 4096

Fragment size: 4096

Blocks per group: 32768

Fragments per group: 32768

Inodes per group: 1024

…

First inode: 11

Inode size: 128

Default directory hash: half\_md4

Directory Hash Seed: 9cf91d57-8528-4d39-b6ba-5f8e2e86fcb7

Group 0: (Blocks 0-1023) [ITABLE\_ZEROED]

Checksum 0x240a, unused inodes 1012

主 superblock at 0, Group descriptors at 1-1

Block bitmap at 2 (+2), Inode bitmap at 18 (+18)

Inode表位于 34-65 (+34)

982 free blocks, 1012 free inodes, 2 directories, 1012个未使用的inodes

可用块数: 8-17, 19-33, 67-1023

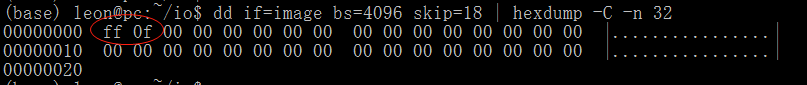
可用inode数: 13-1024

(base) leon@pc:~/io$

可以看到inode bitmap在18个块。

**(5).查看inodebitmap块**

dd if=image bs=4096 skip=18 | hexdump -C -n 32



由于ext4默认用掉11个inode，新创建的ok.txt文件后，inode bitmap用掉12位。

**(6).现在模拟掉电，修改inode bitmap**

vim -b image

:%!xxd –g 1

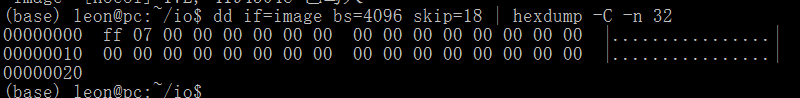
:%!xxd –r

找到inode bitmap块对应地址4096\*18=0x12000



将bitmap改为ff 07



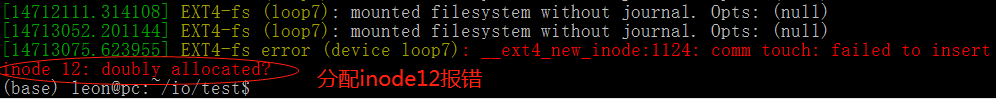


**(7).重新mount访问，查看出错信息**

(base) leon@pc:~/io/test$ sudo touch bad.txt

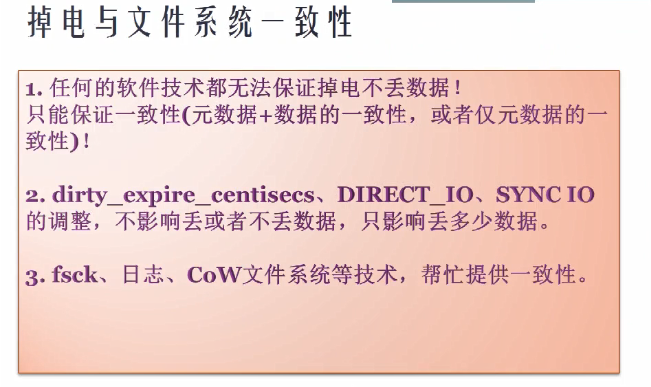
touch: 无法创建'bad.txt': 输入/输出错误

(base) leon@pc:~/io/test$ dmesg



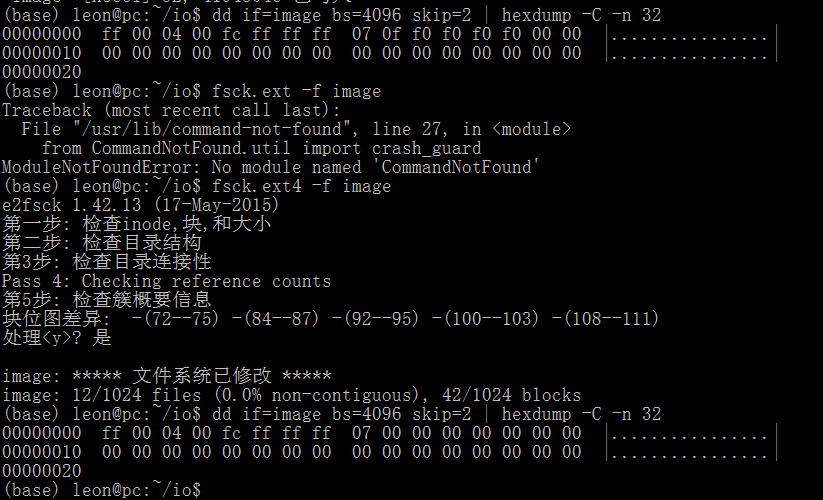
掉电导致的不一致性，会出现各种奇怪的问题，甚至都无法修复；

**任何软件的手段只能保持一致性，无法保证不丢失数据。**



## **3.4 fsck**

人为破坏data block，用fsck修复

  
修复原理，扫描bitmap和inode table的一致性。

早期Linux/Windows系统异常掉电后启动，都用fsck修复磁盘，速度很慢。

为提高速度，新系统都采用日志系统方式。  
3.5 文件系统的日志

将要修改的行为，记录为一个日志，若操作磁盘过程掉电，开机根据日志回放，将磁盘操作全部重做一遍。磁盘操作完成，删除日志。

优点：保持文件系统的一致性，也提高速度。

EXT2/3/4都采用日志系统。

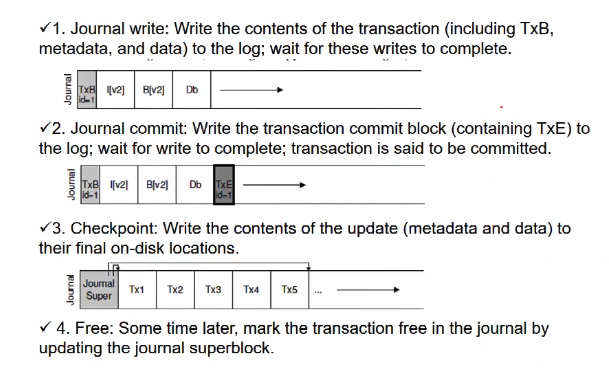
日志的几个阶段：

1.开始写日志

2.日志区写完日志，commited;

3.执行完一条日志，磁盘操作完成,checkpoint。

4.操作完成，free日志。



完整的日志方式，相当于每个数据都写了两遍，让系统变很慢，实际工程上会根据数据情况，做部分日志，即日志方式分为三种：速度递增，安全性递减

data=journal: 完整日志；

data = ordered: 只写元数据，且先写完数据块，再写元数据

data=writeback 只写元数据，循序不确定；ubuntu默认方式；

这样日志就分为5个阶段：



3.6文件系统的调试工具

创建一个文件t.txt，df –h

(base) leon@pc:~/io$ **sudo debugfs -R 'stat /home/leon/io/t.txt' /dev/sda7**

sudo: 无法解析主机：pc: 连接超时

debugfs 1.42.13 (17-May-2015)

Inode: 13896517 Type: regular Mode: 0664 Flags: 0x80000

Generation: 507799365 Version: 0x00000000:00000001

User: 1000 Group: 1000 Size: 17

File ACL: 0 Directory ACL: 0

Links: 1 Blockcount: 8

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x5f40e439:2182bba0 -- Sat Aug 22 17:24:09 2020

atime: 0x5f40e43b:c9589800 -- Sat Aug 22 17:24:11 2020

mtime: 0x5f40e439:2182bba0 -- Sat Aug 22 17:24:09 2020

crtime: 0x5f40e439:208e98b8 -- Sat Aug 22 17:24:09 2020

Size of extra inode fields: 32

EXTENTS:

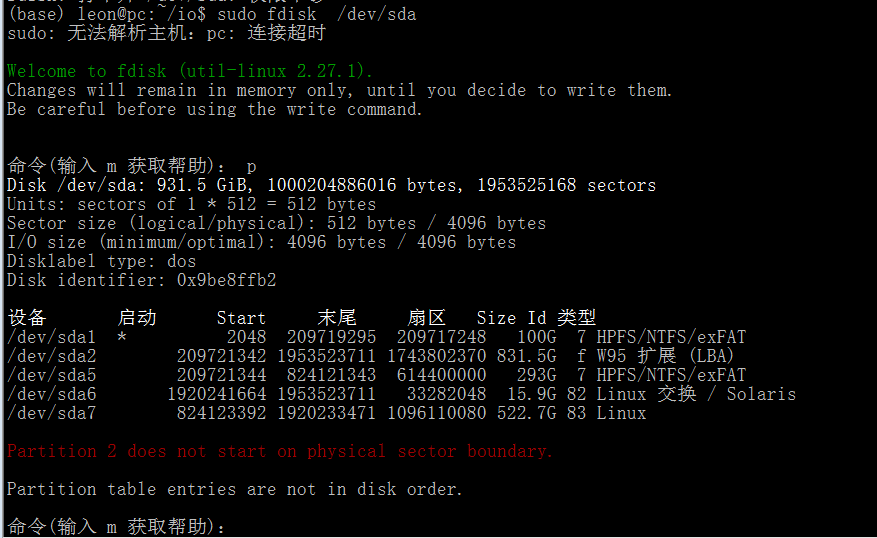
(0):55636171

(END)

**得到文件的数据块55636171**

查看数据块内容

sudo **blkcat** /dev/sda7 55636171



sudo dd if=/dev/sda of=1 skip=$((55636171\*8+824123392)) bs=512c count=1

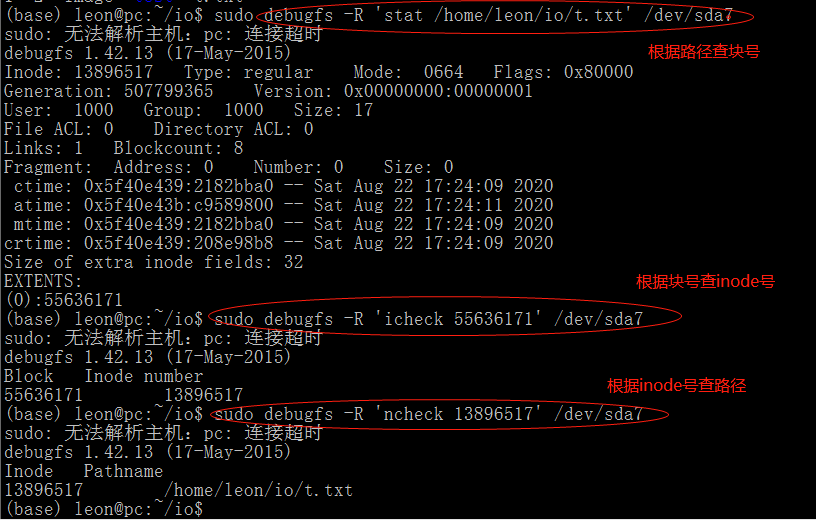


**debugfs 根据块号查inode号**

sudo debugfs -R 'icheck 55636171' /dev/sda7

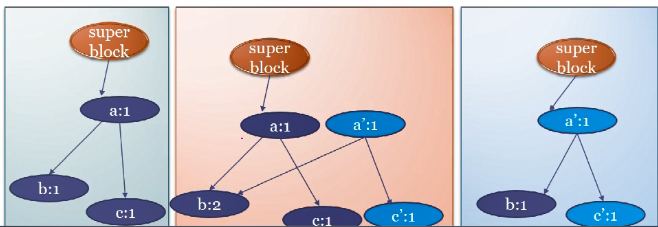
**根据inode号，查文件路径**

sudo debugfs -R 'ncheck 13896517' /dev/sda7



## **3.7 Copy On Write文件系统: btrfs**

不用日志，实现文件系统一致性。每次写磁盘时，先将更新数据写入一个新的block，当新数据写入成功之后，再更新相关的数据结构指向新block。



COW稳健性系统的实现方式，有利于实现子卷和snapshot，类似git的思想：



# 4 块I/O流程与I/O调度器

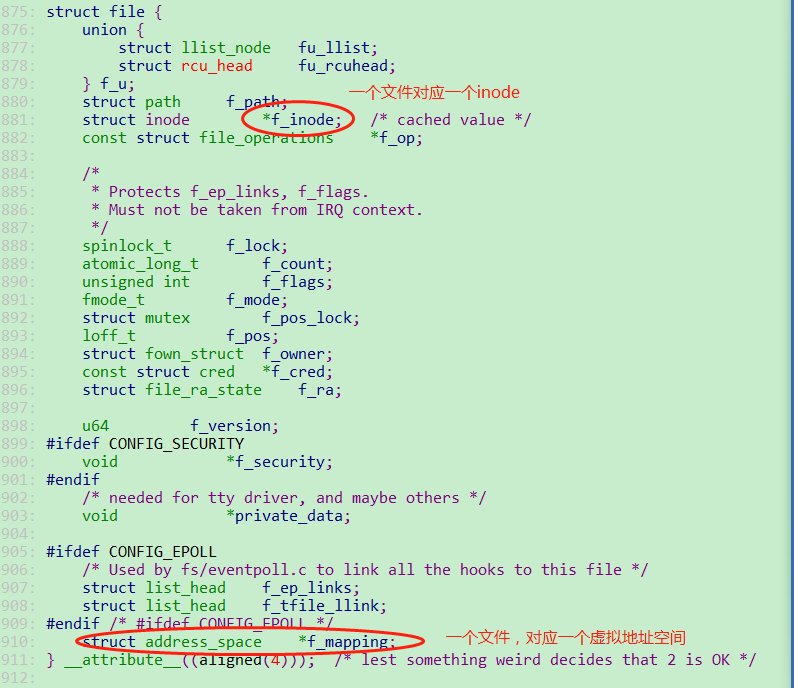
## **4.1一个块IO的一生**

从page cache到bio到request

当APP打开一个文件，内核为文件维护一个pagecache(磁盘的一个副本)；

读写文件时如果page cache命中，只会读写内存不操作磁盘；没有命中，才操作磁盘。

在内核用file结构体表示，可见其包含一个inode结构体，一个地址空间；

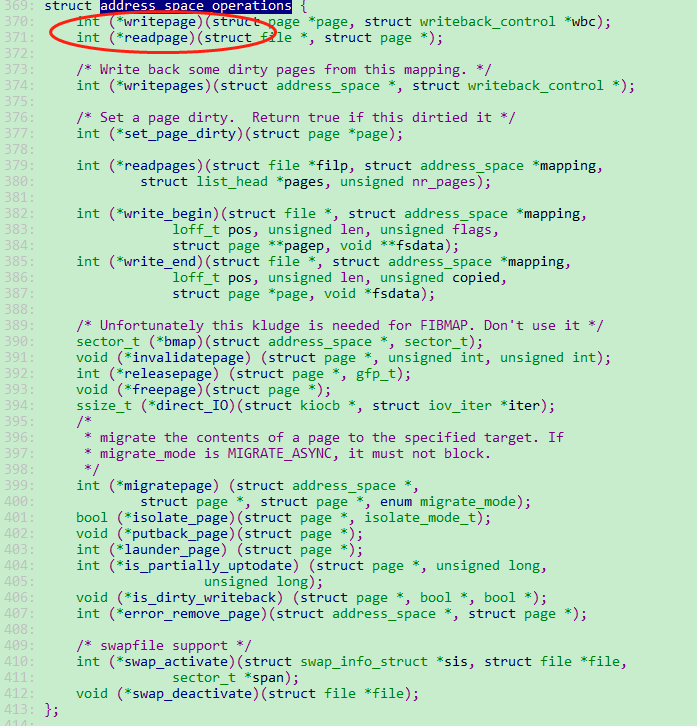


相关的几个结构体在内核对应关系如下：



可见，当多个进程同时打开同一个文件时，不同的file结构体对应同一个inode和同一个地址空间，地址空间是由一颗radix tree维护(即pagecache)，读写文件时，查看对应内存页在page cache中是否命中，若命中直接从内存空间读写；若不命中，申请一个内存页，从磁盘读入数据，挂到page cache的radix tree中。

另外，page cache与磁盘的同步由地址空间操作函数readpage/writepage完成

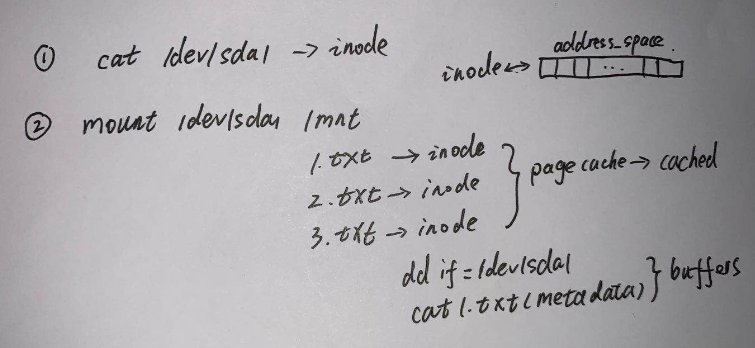


对磁盘访问，有两种方法：

a.裸磁盘直接访问；

b.通过文件系统访问；

它们在内核page cached对应关系如下：



一个address\_space对应一个inode。

**free命令统计的buffer/cached，只是统计上的区别；**

buffer=操作裸分区的地址空间+文件系统的元数据地址空间；

cached=文件系统的地址空间(page cached)缓存；

但是对同一个磁盘，裸磁盘和文件系统两种方式同时操作的时候，同一个数据块会被映射到不同的address\_space，会产生同步的问题；

在用dd的时候，最好不要操作文件系统数据。

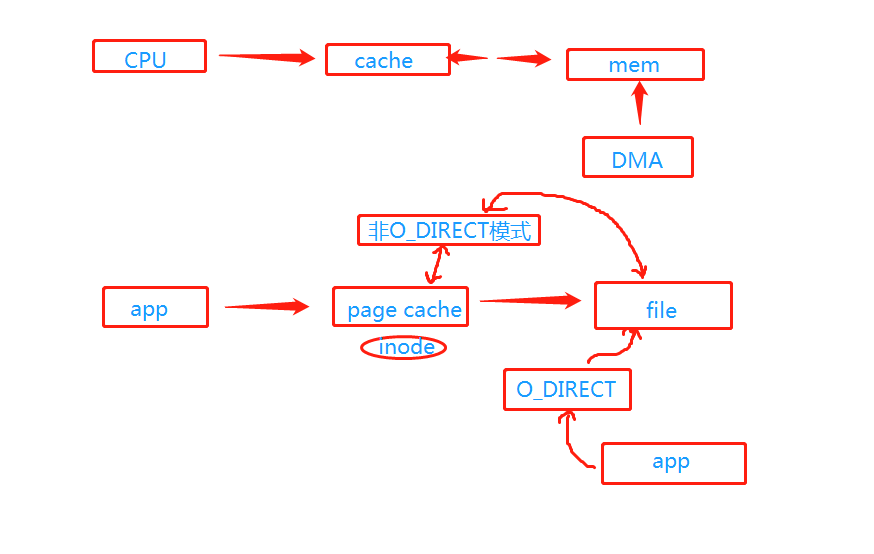
## **4.2 O\_DIRECT和O\_SYNC**

直接操作裸磁盘分区用O\_DIRECT，内核没有任何cache，直接操作磁盘；用户可以根据数据特点，在用户空间做cache。O\_DIRECT申请内存要用posix\_memalign接口；

而O\_SYNC依然通过page cache，但是会立即写入同步磁盘；



App通过page cache访问文件和直接操作裸磁盘的模型，与CPU通过cache访问内存和DMA直接访问内存的模型非常类似；

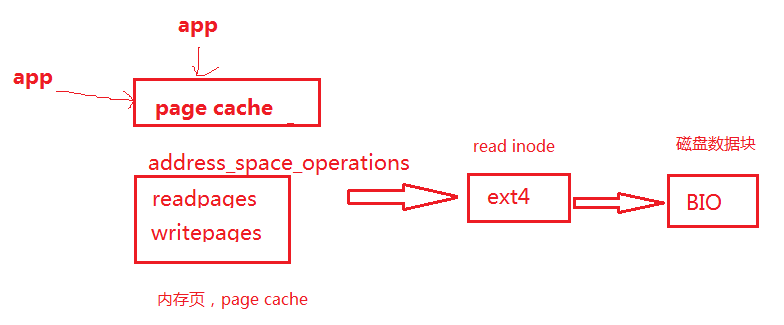


这里page cache是内存，file是磁盘分区数据块；

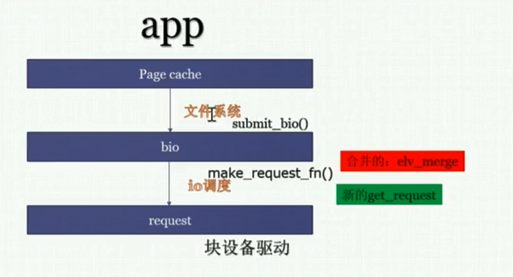
当有一个进程启用O\_DIRECT模式，其他进程最好也用O\_DIRECT；

## **4.3 BIO 流程blktrace**

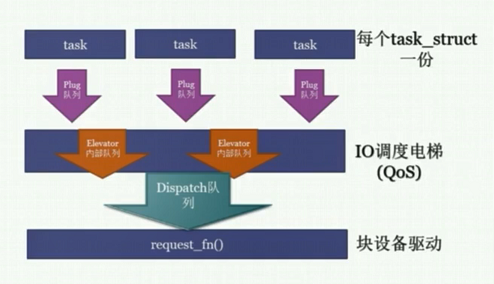
对于一个pagecache地址空间，指向的是page页，文件系统ext4读取inode，将page转化为磁盘数据块，变成BIO请求；



BIO最终要转化成request，然后request被块设备驱动程序调用完成；



Bio经过三个队列变成request，三进三出



**step1:原地蓄势**

把bio转化为request，把request放入进程本进程的**plug队列**；蓄势多个request后，再进行泄洪。

可能合并相邻bio为一个request；

Bio数量>=request数量；

多个进程，泄洪request到电梯；

**ste2.电梯排序**

进程本地的plug队列的request进入到**电梯队列**，进行再次的合并、排序，执行QoS的排队，之后按照QoS的结果，分发给**dispatch队列**。电梯内部的实现，可以有各种各样的队列。

比如两个进程需要读邻近的数据块，可以合并为一个request

电梯调度层可以做QoS，设定进程访问IO的优先级；

**3.分发执行dispatch**

电梯分发的request，被设备驱动的request\_fn()挨个取出来，派发真正的硬件读写命令到硬盘。这个分发的队列，一般就是我们在块设备驱动里面见到的request\_queue了。request\_queue完成真正的硬件操作；

工具ftrace

do.sh

#!/bin/bash

debugfs=/sys/kernel/debug

echo nop > $debugfs/tracing/current\_tracer

echo 0 > $debugfs/tracing/tracing\_on

echo `pidof read` > $debugfs/tracing/set\_ftrace\_pid

echo function\_graph > $debugfs/tracing/current\_tracer

echo vfs\_read > $debugfs/tracing/set\_graph\_function

echo 1 > $debugfs/tracing/tracing\_on

执行./read读文件

查看trace过程

sudo cat /sys/kernel/debug/tracing/trace > t.txt

用VIM查看t.txt，用.funcgrahp.vim插件打开可以合并/展开对应函数调用

vim -S ~/.funcgrahp.vim

## **4.4 IO调度算法，CFQ和ionice**

查看当前系统的IO调度算法

(base) leon@pc:/sys/block/sda/queue$ cat scheduler

noop deadline [cfq]

修改调度算法:

sudo echo cfg > scheduler

CFQ调度算法:类似进程调度的CFS算法

ionice –help

ionice –c 2 –n 0 dd if=/dev/sda of=/dev/null & //设置nice值=0

ionice –c 2 –n 8 dd if=/dev/sda of=/dev/null & //设置nice值=8

iotop查看IO读写速度，有明显差异

改成deadline策略

echo deadline > scheduler

此时尽管优先级不同，但两个进程IO速度相近了；

## 4.5 cgroup与IO

cd /sys/fs/cgroup/blkio

**a.创建群组A,B**

Mkdir A;makedir B

cat blkiolweight //默认是500

分别把两个进程放到A和B执行

cgexec -g blkio:A dd if=/dev/sda of=/dev/null iflag=direct &

cgexec -g blkio:B dd if=/dev/sda of=/dev/null iflag=direct &

查看io占用iotop

默认是相近

**b.修改群权重**

echo 50 > blkio.weight

两个进程优先级一样，但IO速度差别很大；

**c.cgroup还可以控制阀门，限制最大读速度：**

8：0磁盘的主次设备号

echo "8:0 1048576" > /sys/fs/cgroup/blkio/A/blkio.throttle.read\_bps\_device

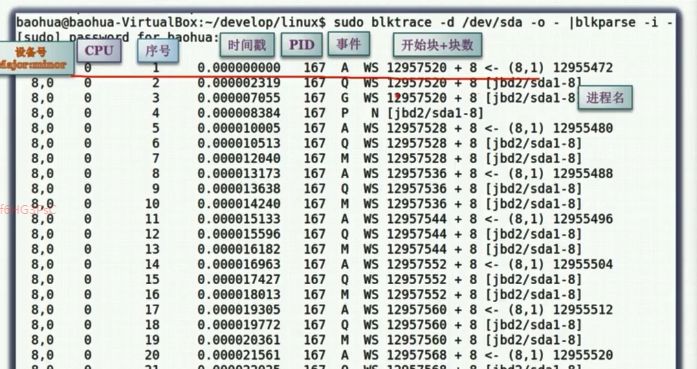
带cache读写，这里的限速不起作用；direct方式才生效；

cgroup v2版本支持带cache限速

## 4.6 IO性能调试：iotop, iostat

blktrace跟踪硬盘的各个读写点

blktrace –d /dev/sda –o - |blkparse –I –



dd if=main.c of=t.txt oflag=sync

debugfs –R ‘ickeck xxxx’ /dev/sda1

debugfs –R ‘nckeck inode’ /dev/sda1

blkcat /dev/sda1 xxx