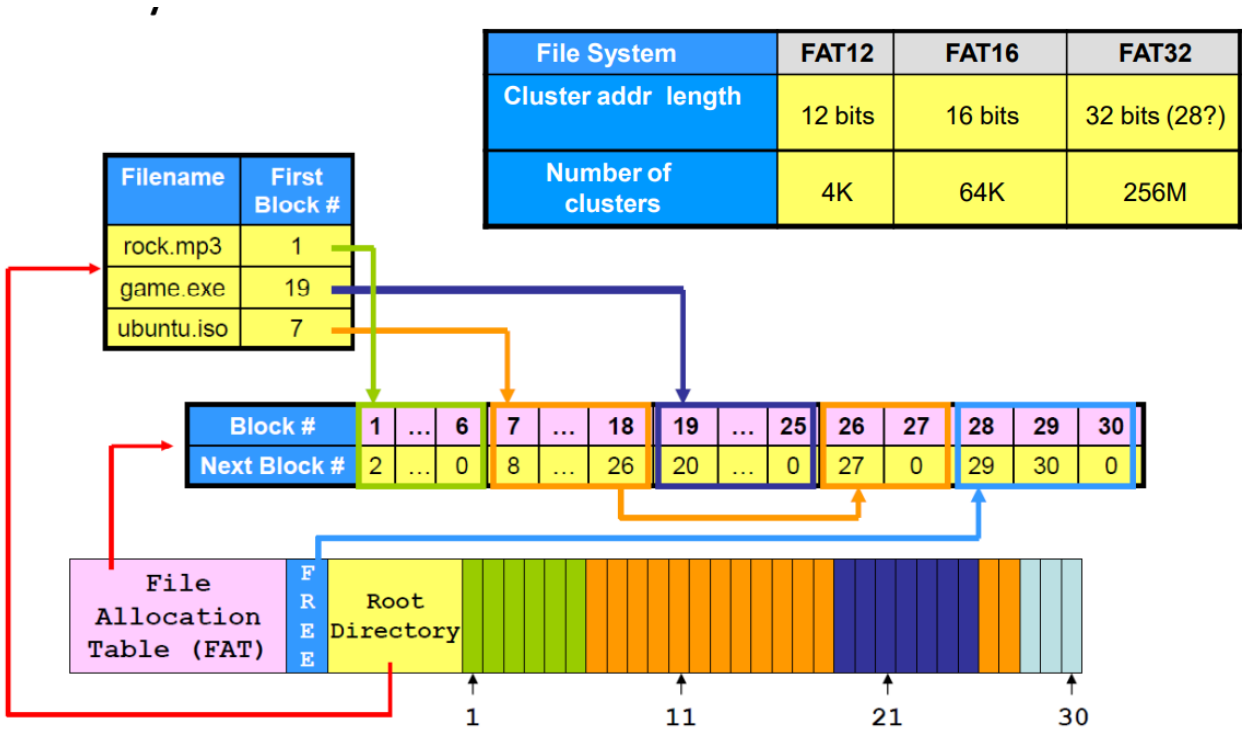
# CH10

## Details of FAT32

### 介绍

布局CH9有




| \ | PROPOSE | SIZE |
| --- | --- | --- |
| 启动扇区 | 储存该文件系统参数 | 1个扇区，512字节 |
| FSINFO | 空闲空间管理 | 同上 |
| 保留扇区 | | 不同格式大小不同 |
| 2个FAT表 | 提高可靠性 | 与硬盘有关 |
| 根目录 | 目录树的开始 | 至少一个簇，与目录项数目有关 |

# 目录项

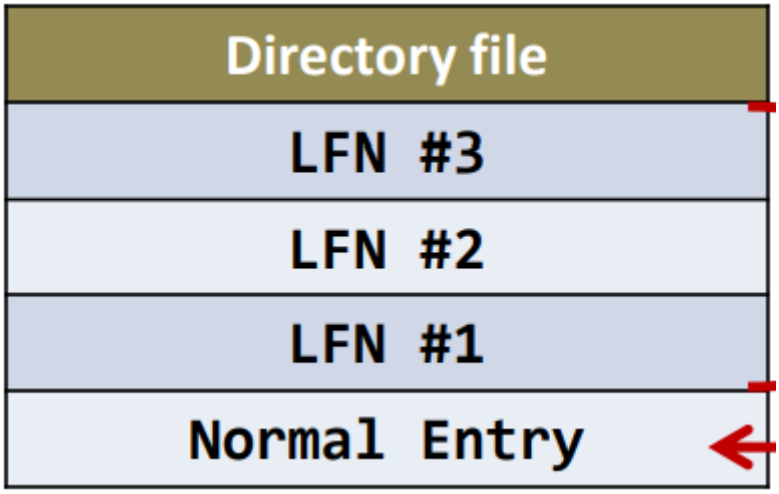
一个结构

| BYTE | DESCRIPTION |
| --- | --- |
| 0-0 | 文件名第一位(0xe5删除,0x00空闲) |
| 1-10 | 文件名(11位中8位文件名，3位扩展) |
| 11-11 | 文件属性 |
| 13-19 | 时间 |
| 20-21 | 簇号高两字节 |
| 26-27 | 簇号低两字节 |
| 28-31 | 文件大小 |

注意:小端系统**26**字节才是最低

文件最大大小4G-1bytes

长文件名(LFN)



| Bytes | Description |
| --- | --- |
| 0-0 | Sequence Number |
| 1-10 | File name characters<br>(5 characters in Unicode) |
| 11-11 | File attributes - always 0x0F |
| 12-12 | Reserved. |
| 13-13 | Checksum |
| 14-25 | File name characters<br>(6 characters in Unicode) |
| 26-27 | Reserved |
| 28-31 | File name characters<br>(2 characters in Unicode) |

## 文件操作

### 读文件

例子:C:\windows\explorer.exe

1. 先找到explorer.exe的簇号32，再读取内容
2. 在FAT表32号簇位置读取下一个簇号
3. 1,2步骤往复直到FAT表的内容为EOF

写文件

1. 查找到文件的最后一个簇
2. 未满则继续写入
3. 满则从FSINFO中分配空闲簇
4. 更新FAT和FSINFO
5. 写完后更新文件大小

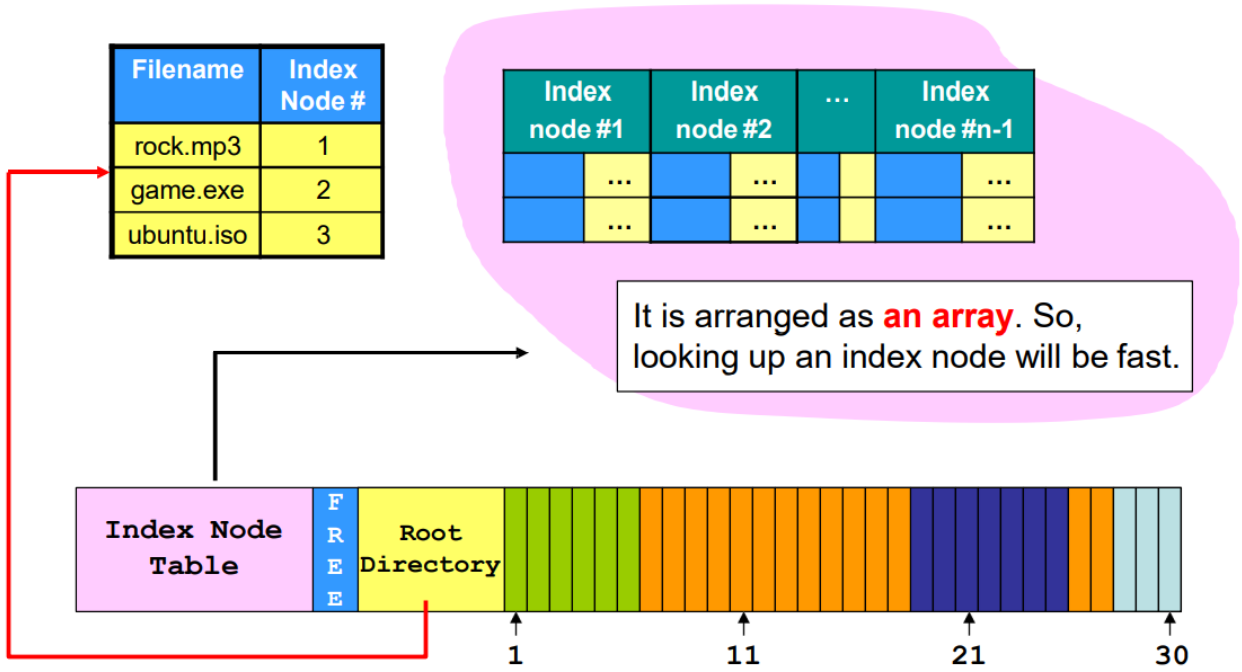
删除文件

1. 找到目录项
2. 把name[0]改为0xe5

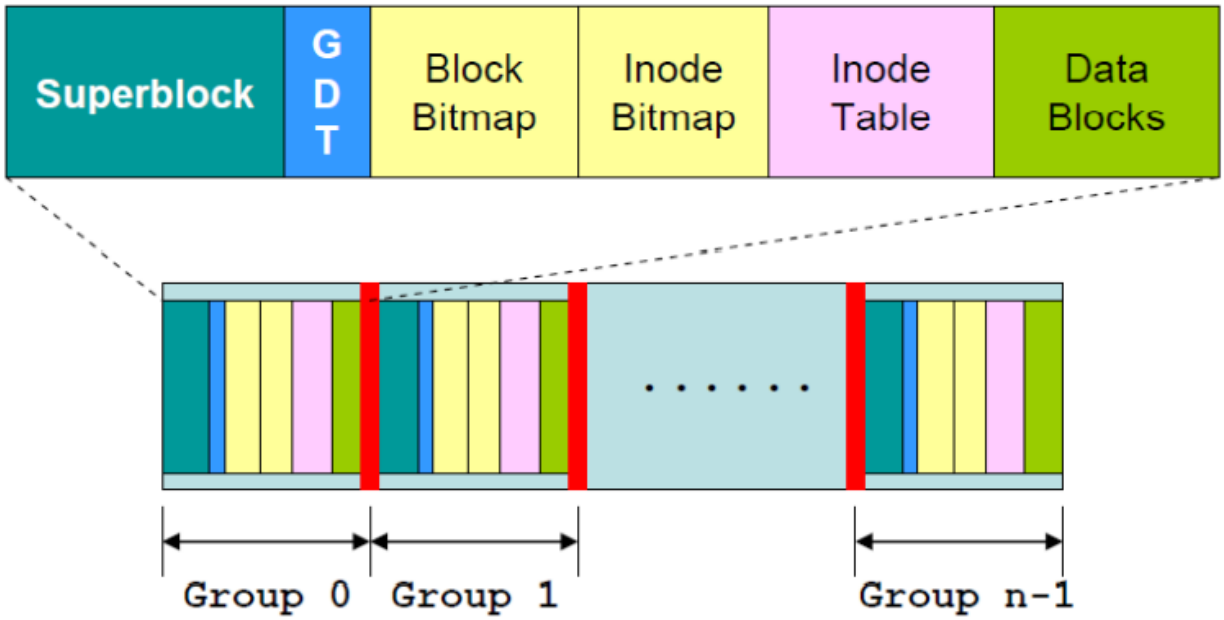
恢复文件

1. 单个簇容易找回
2. 多个簇可以依照未修改的FAT查找，但已修改则永远丢失

## Details of Ext2/3/4




存在分组现象




每一组的superblock一样、其他不一样(冗余带来可靠性)

分组原因:分割Inode以减小文件系统

superblock内容:

1. Total number of inodes in the system.
2. Total number of blocks in the system.
3. Number of reserved blocks
4. Total number of free blocks.
5. Total number of free inodes.
6. Location of the first block.
7. The size of a block.

每组的设计

| TYPE | CONTENT |
| --- | --- |
| superblock | 文件系统信息 |
| INODE table | 一个INODE数组 |
| GDT | 储存Block bitmap、INODE bitmap、inode tabl的起始块 |
| data block | 储存文件数据的块 |
| block bitmap | 标记块是否被分配 |
| INODE bitmap | 标记inode是否被分配 |

因此FS中文件最大数目是固定的

## inode 结构

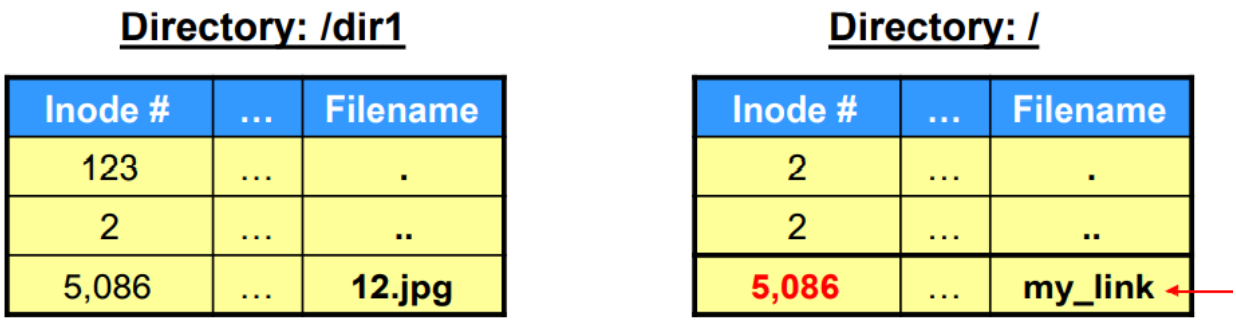
| BYTE | VALUE |
| --- | --- |
| 0-1 | File type and permission |
| 2-3 | User ID |
| 4-7 | Lower 32 bits of file sizes in bytes |
| 8-23 | Time information |
| 24-25 | Group ID |
| 26-27 | Link count |
| 40-87 | 12 direct data block pointers |
| 88-91 | Single indirect block pointer |
| 92-95 | Double indirect block pointer |
| 96-99 | Triple Indirect block pointer |
| 108-111 | Upper 32 bits of file sizes in bytes |

## linkfile

两种:hard link和symbolic link

### hard link

仅仅是一个目录项指向一个已经存在的文件，未创建新文件




相当于创建了一个具有两个路径名的文件

在inode有一个link count域储存这个链接数

特别的两个`.`,`..`

删除文件:调用unlink()系统调用

将link  count减一，当link count为0时释放空间

### symbolic link

相当于创建了一个新的inode，这个inode储存路径名

## buffer cache

内核buffer cache

定义:内核保留一些最近被访问的数据块的备份，这些空间被叫做kernel buffer cache

3种buffer cache

| TYPE | CONTENT |
| --- | --- |
| 页面缓存 | 缓存一些文件的数据块 |
| 目录项缓存 | 把目录项储存在内核 |
| inode缓存 | inode缓存到内核 |
| | |

替换算法 LRU

配合buffer cache 的读写

### read

读文件时会自动cache，通过调用readahead()

```
ssize_t readahead(int fd, off64_t offset, size_t count);
//读取接下来的一定范围的数据
```

工作原理:当读取块x时，读取块x+1概率也很大(基于之前读取是随机访问还是顺序访问)

cache完减少访问硬盘:

1. 硬盘头不会一直停在想要的位置
2. 硬盘的顺序读写性能优秀

### write

分为写直达和写返回，与COD同

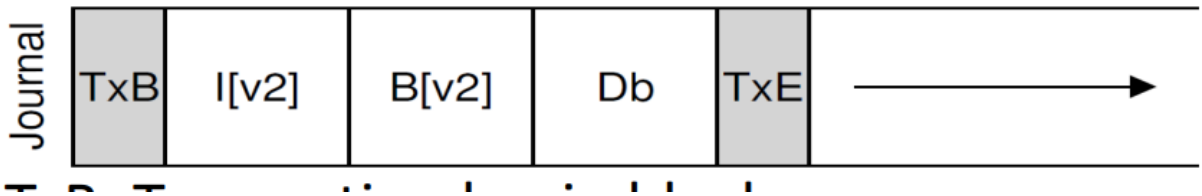
## journaling(日志系统)

日志是文件系统的日记本，在磁盘(如下图)中

文件系统操作会被分割成一系列的原子操作，这些操作标志着所有对文件系统的操作，每一个操作都被写入到日志


有如下几种原子操作:

1. 更新inode
2. 更新bitmap
3. 更新数据块




### 策略1——data journaling

执行顺序:

1. 先写日志

   把原子操作写到日志(包括TXB,TXE,元数据,需写入的数据)，并等待写入完成

2. checkpoint

   把待写入的数据写到正确的位置、更新元数据

写日志的几种方法

1. 一个一个原子操作写，安全、速度慢
2. 全部一起写入，不安全
3. 先把除了TXE的原子操作写入，最后写入TXE

详细的操作顺序

1. 写日志

   向日志中写入操作的内容，包括TXB、数据和元数据

2. 提交

   元数据和数据，还有TXE

3. 更新到文件系统相应位置

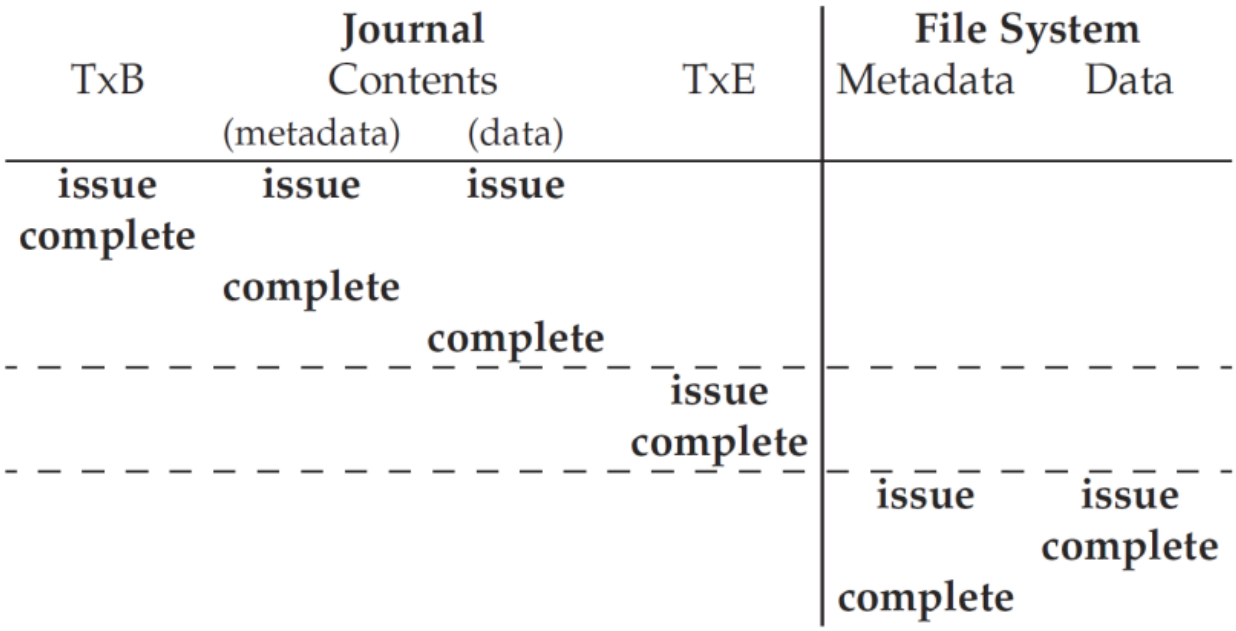
数据恢复

1. 在commit之前崩溃

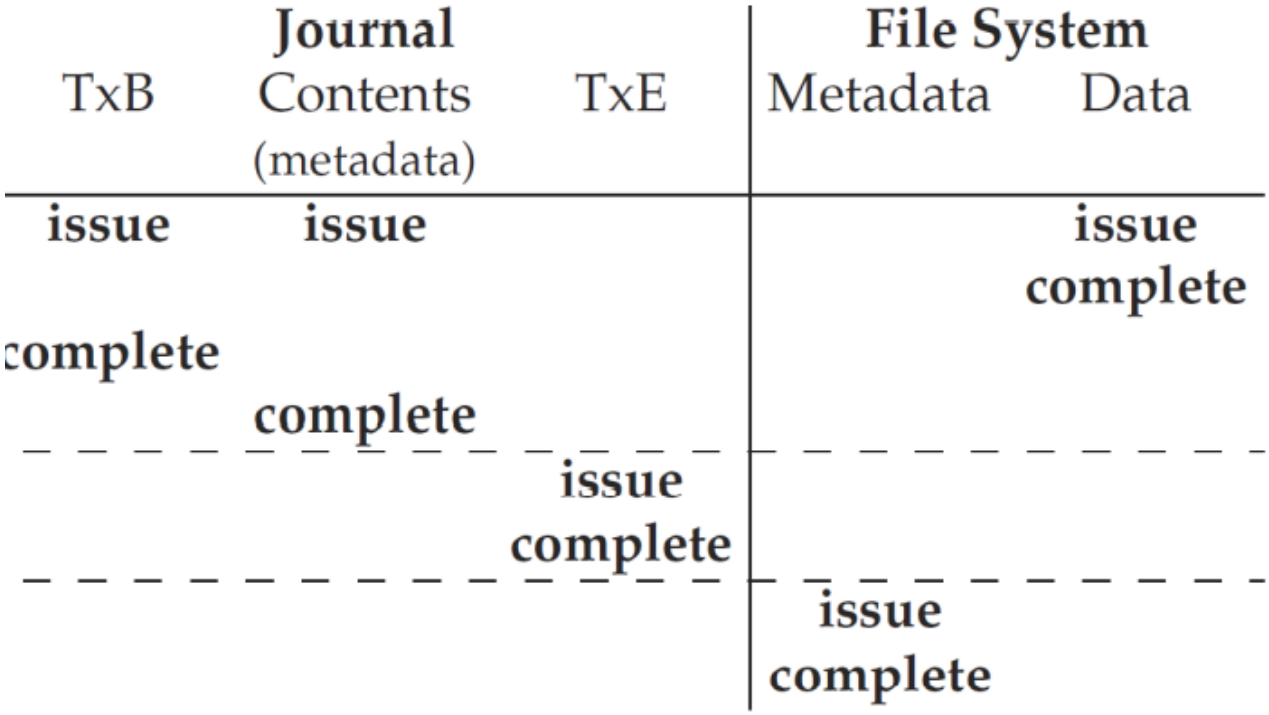
   跳过更新

2. commit之后

   执行日志相关内容



策略**2——metadata journaling**




数据和元数据的写入是并发的，不用写两次数据

1. 写数据和写日志并发执行
2. 提交日志
3. 写入元数据

## VFS

linux提供了一个统一的文件系统接口支持不同的文件系统