

**软件工程**

**课程设计报告**

**课题名称：Linux文件系统分析**

**班 级：**

**姓 名：uknowho**

**学 号：201446xx**

**指导教师：csb**

2017.8.23

目录

[概述 2](#_Toc491382895)

[设计 2](#_Toc491382896)

[概要设计 2](#_Toc491382897)

[Ext文件系统分析流程： 2](#_Toc491382898)

[布局设置 5](#_Toc491382899)

[控件及功能设置设置 5](#_Toc491382900)

[详细设计 6](#_Toc491382901)

[1. 磁盘内容的读取与显示 6](#_Toc491382902)

[数据块寻址 12](#_Toc491382903)

[系统编码 14](#_Toc491382904)

[测试机分区分布 18](#_Toc491382905)

[参考 19](#_Toc491382906)

# 概述

Disknight 是一款基Qt Creator 开发的应用于Linux的磁盘浏览器。主要分析了Ubuntu文件系统 Ext4 的磁盘映像数据。并且提供以扇区为单位对磁盘数据进行浏览，数据修改、导出等功能。

位置在 /home/username/Desktop/my\_data.txt。每次导出会覆盖原有数据

并且通过对磁盘数据的分析，显示出了分区的基本信息，如inode总数，blocks总数等。

对已使用的inode、block 所占百分比以条形图的形式显示出来。

由于直接修改磁盘数据可能会导致严重异常情况出现，所以对软件的每一步操作都进行了记录，

在“操作日志”区可以浏览到所有对软件的可检测操作。如数据修改、扇区跳转等。并提供导出操作日志功能。位置在 /home/username/Desktop/my\_log.txt。每次导出会保留上一次导出内容。

# 设计

## 概要设计

### Ext文件系统分析流程：

1. 从MBR 找到分区的入口

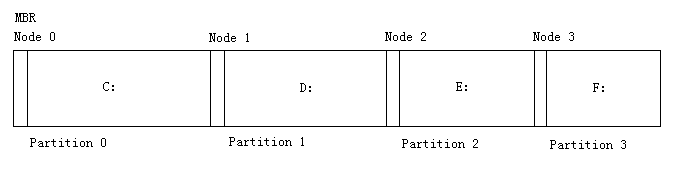


图1 MBR格式磁盘总体布局图

对磁盘进行格式化分区后，可以选择用什么文件系统初始化分区。如EXT、NTFS、FAT32等等。

1. 分析EXT 文件系统的结构

Ext文件系统把所分得磁盘空间初始化为N个数据块（data block）。一定适量的n个数据块构成了块组（Block Group），而整个分区就是以块组（Block Group）为单位进行管理的。为了方便，设置了数据结构块组描述符（GDT）对所有块组进行管理。GDT的位置在0号块组的超级块之后。

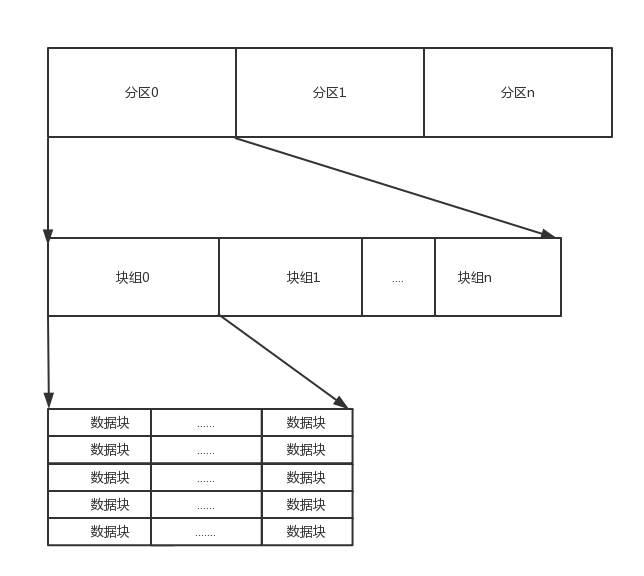


图2 Ext4文件系统组织形式

需要注意的是，在0分区起始处，会有大小为1K的数据，成为启动块或启动扇区（Boot Sector）

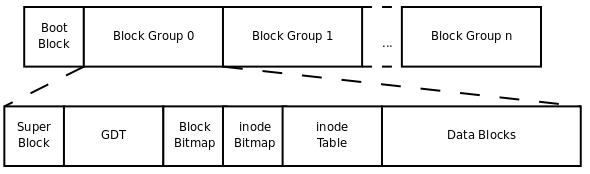


图3 磁盘分区上的Ext4总体结构

其中，块位图（Block Bitmap）、inode位图、（inode Bitmap）、和inode表（indoe Table）之间可能并不是连续存放，即不是完全无缝连接在一起。

要读取某一特定目录的数据（以根目录为例）就必根据根目录（/）的inode号找到对应块组，深入块组，找到对应的inode表（inode Table），根据inode Table找出真正保存根目录数据的数据块。

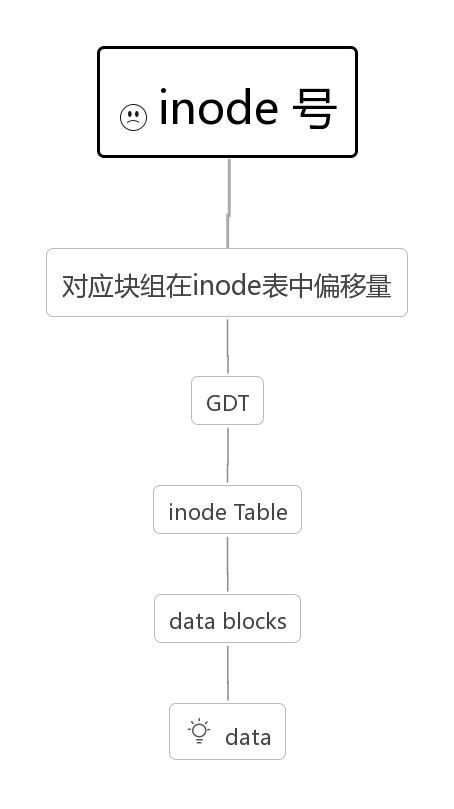


图4 根目录寻找思维导图

类Linux操作系统磁盘设备属于块设备，放置于 /usr/sd\_/ 文件中。(sda、sdb等).。

Disknight 使用底层IO函数进行读写磁盘文件 /usr/blocks/8:0/

### 布局设置



图五 界面原形

### 控件及功能设置设置

表1 基本控件

|  |  |  |  |
| --- | --- | --- | --- |
| 主要控件名称 | 用途 | 主要控件名称 | 用途 |
| TextEdit | 显示磁盘数据 | CommandLinkButton | 导出磁盘数据 |
| TextEdit | 记录操作日志 | CommandLinkButton | 导出操作日志 |
| CheckBox | 开启/关闭编辑模式 | ProcessBar | 已用inode比例 |
| TextEdit | 显示扇区数/更改扇区数 | ProcessBar | 已用block比例 |
| Button | 跳转 | Labels若干 | 显示基本数据 |
| Button | 上一扇区 | Labels若干 | 显示标题 |
| Button | 下一扇区 | Containers若干 | 控制布局 |

## 详细设计

### 磁盘内容的读取与显示

由于Linux系统的权限限制，在使用 /dev/block/8:0/ 文件时应当先把相应权限赋予。这里赋予所有权限，即可读可写可执行。



图六 修改文件权限

可以看到， 实际上 /dev/block/8:0/ 文件是一个指向 /dev/sda/ 的链接文件。

利用底层IO函数 open 和 read 和 write 即可打开文件，进行读、写操作。读取出的数据为二进制数据，不方便进行浏览。这里提一个格式化二进制数据的方法 disk\_formater(char\*, int) 将二进制数据格式化后，显示在相应区域。

下表列出了对磁盘操作的主要函数

表2 主要函数列表

|  |  |  |
| --- | --- | --- |
| **磁盘操作** | | |
| 返回值 | 函数名 | 用途 |
| void | read\_disk\_data(const char \* file\_name, long sector\_number) | 读取指定扇区数据 |
| void | static data\_formater(char\* buf, int size) | 格式化二进制数据 |
| void | jmp2\_mbr() | 移动文件指针到MBR |
| void | jmp2\_super\_block() | 移动文件指针到超级块 |
| long | jmp2\_boot\_block() | 移动文件指针到启动块（扇区） |
| void | jmp2\_gdt() | 移动文件指针到GDT |
| void | jmp2\_inode\_table(int n\_bg) | 移动文件指针到块组n\_bg的inode-table |
| void | jmp2\_n\_block\_group(int n\_bg) | 移动文件指针到n\_bg块组 |
| char\* | get\_a\_buffer() | 获得全局缓冲区 |
| long | get\_inodes() | 获取inodes总数 |
| long | get\_blocks() | 获取block总数 |
| long | grab\_n\_bytes(int fd, int len\_hex) | 截取长度为\_len\_hex(16进制)的数据并返回 |
| long | get\_blocks\_free() | 返回未使用的block的数量 |
| long | get\_blocks\_reserved() | 返回系统保留的block的数量 |
| long | get\_first\_datablock\_num() | 返回第一块数据块块号 |
| long | get\_inodes\_free() | 返回未使用的inode的数量 |
| long | get\_inodes\_per\_gp() | 返回一个块组中inode的数量 |
| long | get\_blocks\_per\_gp() | 返回一个块组中block的数量 |
| long | get\_inode\_size() | 返回inode大小(16进制) |
| long | get\_n\_bg\_descriptor(int n\_bg) | 返回第n\_bg块组的块组描述符 |
| long | get\_n\_bg\_block\_bitmap(int n\_bg) | 返回第n\_bg块组的块位图 |
| long | get\_n\_bg\_inode\_bitmap(int n\_bg) | 返回第n\_bg块组的inode位图 |
| long | get\_n\_bg\_inode\_table(int n\_bg) | 返回第n\_bg块组的inode表 |
| QString | get\_n\_bg\_number(int inode) | 返回inode号对应的块组号 |
| QString | block\_num\_2\_sector\_num(long block\_num) | 块号转为扇区号 |
| QString | get\_n\_inode\_offset\_in\_inode\_table(int inode) | 返回inode号在inode表中的偏移量 |
| long | jmp2\_data\_block\_in\_inode\_table(int inode) | 跳转到inode号所在inode表中的位置 |
| QString | get\_page\_data(int) | 返回指定扇区的数据， |
| **事件响应** | | |
| void | on\_btn\_prev\_clicked() | 响应上一页按钮点击 |
| void | on\_btn\_next\_clicked() | 响应下一页按钮点击 |
| void | on\_btn\_jump\_clicked(); | 响应跳转按钮点击 |
| void | on\_cmdbtn\_export\_data\_clicked() | 导出当前扇区数据 |
| void | on\_btn\_boot\_sector\_clicked() | 跳转到启动扇区并显示数据 |
| void | on\_btn\_super\_block\_clicked() | 跳转到超级块并显示数据 |
| void | on\_btn\_gdt\_clicked() | 跳转到GDT并显示数据 |
| void | call\_alert(int alert\_code) | 响应错误代码 |
| void | on\_cb\_edit\_mode\_clicked() | 开启/关闭 编辑模式 |
| long | w\_log(QString msg) | 记录操作日志 |
| void | on\_cmdbtn\_export\_log\_clicked() | 导出操作日志 |
| void | set\_processbar() | 设置用量比例 |

要使二进制数据可读，需要对二进制数据进行格式化处理，这里了提供一种格式化处理方式：

QString Dialog::data\_formater(char \*buf, int size)

/\*

Format 'size' data in 'buf' to hexadecimal and readble.

\*/

{

QString data="Oset 00 01 02 03 04 05 "

"06 07 -08 09 0A 0B "

"0C 0D 0E 0F\n0000 ";

QString text="";

for(int i=0; i<size; i++)

{

if(i!=0 && i%16==0)

{

data+=" "+text;

data+=QString::asprintf("\r\n%04X ", i);

text="";

}

else if(i!=0 && i%8==0)

{

data+="-";

text+=" ";

}

data+=QString::asprintf("%02X ", buf[i]&0xFF);

if(isprint(buf[i]))

text+=QString::asprintf("%c", buf[i]);

else

text+=".";

}

return (data+" "+text);

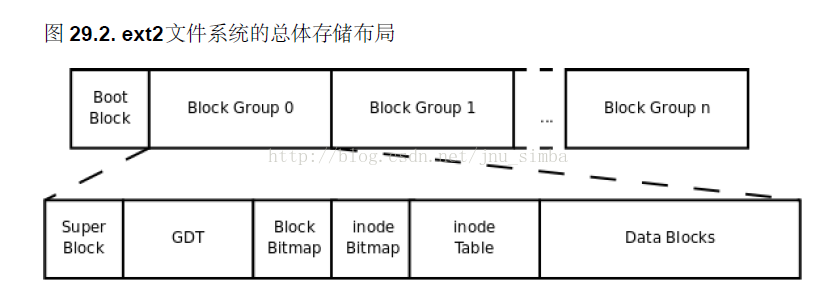
}

磁盘上第一个扇区的数据通常都是MBR，从MBR中可以获得第一个分区的起始扇区号。

表3 MBR硬盘分区表数据结构

|  |  |  |
| --- | --- | --- |
| **偏移** | **长度（字节）** | **意义** |
| 00H | 1 | 分区状态：00-->非活动分区；80-->活动分区； 其它数值没有意义 |
| 01H | 1 | 分区起始磁头号（HEAD），用到全部8位 |
| 02H | 2 | 分区起始扇区号（SECTOR），占据02H的位0－5； 该分区的起始磁柱号（CYLINDER），占据 02H的位6－7和03H的全部8位 |
| 04H | 1 | [文件系统](https://zh.wikipedia.org/wiki/%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F)标志位 |
| 05H | 1 | 分区结束磁头号（HEAD），用到全部8位 |
| 06H | 2 | 分区结束扇区号（SECTOR），占据06H的位0－5； 该分区的结束磁柱号（CYLINDER），占据 06H的位6－7和07H的全部8位 |
| 08H | 4 | 分区起始相对扇区号 |
| 0CH | 4 | 分区总的扇区数 |

因为分区起始扇区处1KB 数据是启动块（BootBlock）是由 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 ext3文件系统的开始。



图七 启动块在块组中的位置

由于分区是由块组进行管理，每个块组都由以下部分组成：

* 超级块(Super Block)

超级块描述整个文件系统分区的信息，如果超级块损坏将会丢失整个分区的数据，所以为了保证数据安全从ext2 文件系统中，每一个块组的开头都有一份超级块的备份

超级块的数据结构如下：

struct ext2\_super\_block {

\_\_le32 s\_blocks\_count; //Blocks数量:

\_\_le32 s\_inodes\_count; //I-nodes数量:

\_\_le32 s\_r\_blocks\_count; //保留的 blocks数量:

\_\_le32 s\_free\_blocks\_count; //空闲的 blocks数量:

\_\_le32 s\_free\_inodes\_count; //空闲的 i-nodes数量:

\_\_le32 s\_first\_data\_block; //第一个数据块块号:

\_\_le32 s\_log\_block\_size; //日志数据块大小:

\_\_le32 s\_log\_frag\_size; //日志片段大小：

\_\_le32 s\_blocks\_per\_group; //每一个块组中数据块的数量：

\_\_le32 s\_frags\_per\_group; //每一个块组中片段的数量

\_\_le32 s\_inodes\_per\_group; //每一个块组中I-nodes的数量：

\_\_le32 s\_mtime; //挂载时间：

\_\_le32 s\_wtime; //写时间：

\_\_le16 s\_mnt\_count; //挂载次数：

\_\_le16 s\_max\_mnt\_count; //最大挂载次数：

\_\_le16 s\_magic; //Magic签名：

\_\_le16 s\_state; //文件系统状态：

\_\_le16 s\_errors; //文件系统出错：

\_\_le16 s\_minor\_rev\_level; //次版本号：

\_\_le32 s\_lastcheck; //上一次检查时间：

\_\_le32 s\_checkinterval; //检查间隔值：

\_\_le32 s\_creator\_os; //创建操作系统：

\_\_le32 s\_rev\_level; //版本号：

\_\_le16 s\_def\_resuid; //默认保留uid：

\_\_le16 s\_def\_resgid; //默认保留gid：

\_\_le32 s\_first\_ino; //第一个非保留的i-node：

\_\_le16 s\_inode\_size; // i-node结构体的大小:

\_\_le16 s\_block\_group\_nr; //当前超级块所在的块组号:

\_\_le32 s\_feature\_compat; //兼容性特征：

\_\_le32 s\_feature\_incompat; //非兼容性特征：

\_\_le32 s\_feature\_ro\_compat; //行兼容性特征：

\_\_u8 s\_uuid[16]; //分区uid:

char s\_volume\_name[16]; //分区卷名:

char s\_last\_mounted[64]; //最后一次挂载的目录：

\_\_le32 s\_algorithm\_usage\_bitmap; //位图算法使用：

\_\_u8 s\_prealloc\_blocks; //预分配块数：

\_\_u8 s\_prealloc\_dir\_blocks; //预分配目录块数：

\_\_u16 s\_padding1; //填充数据1：

\_\_u8 s\_journal\_uuid[16]; //日志文件的uuid:

\_\_u32 s\_journal\_inum; //日志文件i-node号:

\_\_u32 s\_journal\_dev; //日志文件设备号:

\_\_u32 s\_last\_orphan; //分区未使用的剩余块:

\_\_u32 s\_hash\_seed[4]; // hash种子值:

\_\_u8 s\_def\_hash\_version; //默认hash版本:

\_\_u8 s\_reserved\_char\_pad; //保留填充字符:

\_\_le32 s\_default\_mount\_opts; //默认登录选项:

\_\_le32 s\_first\_meta\_bg; //第一个元背景:

\_\_u32 s\_reserved[190]; //保留:

};

* 块组描述符（GDT）

由很多块组描述符组成，整个分区分成多少个块组就对应有多少个块组描述符。每个块组描述符（ Group Descriptor）用20h存储一个块组的描述信息，例如在这个块组中从哪里开始是 inode表，从哪里开始是数据块，空闲的 inode和数据块还有多少个等等。和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第0 个块组中的拷贝，当执行 e2fsck检查文件系统一致性时，第 0个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第 0个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

块组描述符的数据结构如下：

struct ext2\_group\_desc

{

\_\_le32 bg\_block\_bitmap; //块位图所在的数据块号 ：00004001h，2808h扇区

\_\_le32 bg\_inode\_bitmap; //i-node位图所在的块号：00000411h，2888h扇区

\_\_le32 bg\_inode\_table; //i-node表所在的数据块号：00000421h，2908h扇区

\_\_le16 bg\_free\_blocks\_count; //空闲块数量：1354h

\_\_le16 bg\_free\_inodes\_count; //空闲i-node数量：0008h

\_\_le16 bg\_used\_dirs\_count; //目录数：0239h

\_\_le16 bg\_pad; //GDT填充：0004h

\_\_le32 bg\_reserved[3]; //GDT保留：00000000h，00000000h,C8200000h

};

* 快位图（Block Bitmap）

一个块组中的块是这样利用的：数据块存储所有文件的数据。超级块、块组描述符表、块位图、 inode位图、 inode表这几部分存储该块组的描述信息。那么如何知道哪些块已经用来存储文件数据或其它描述信息，哪些块仍然空闲可用呢？块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个 bit代表本块组中的一个块，这个 bit为 1表示该块已用，这个 bit为 0表示该块空闲可用。

为什么用df命令统计整个磁盘的已用空间非常快呢？因为只需要查看每个块组的块位图即可，而不需要搜遍整个分区。相反，用 du命令查看一个较大目录的已用空间就非常慢，因为不可避免地要搜遍整个目录的所有文件。

设块大小指定为 b字节，那么一个块可以有 8b个 bit，这样大小的一个块位图就可以表示 8b个块的占用情况，因此一个块组最多可以有 8b个块。

* Inode位图（inode Bitmap）

和块位图类似，本身占一个块，其中每个 bit表示一个 inode是否空闲可用。

* Inode表（inode-Table）

一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等），权限，文件大小，创建 /修改 /访问时间等。这些信息存在 inode中而不是数据块中。每个文件都有一个 inode，一个块组中的所有 inode组成了 inode表。

* 数据块(data Block)

根据不同的文件类型有以下几种情况：

对于常规文件，文件的数据存储在数据块中。

对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外， ls -l命令看到的其它信息都保存在该文件的inode中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。

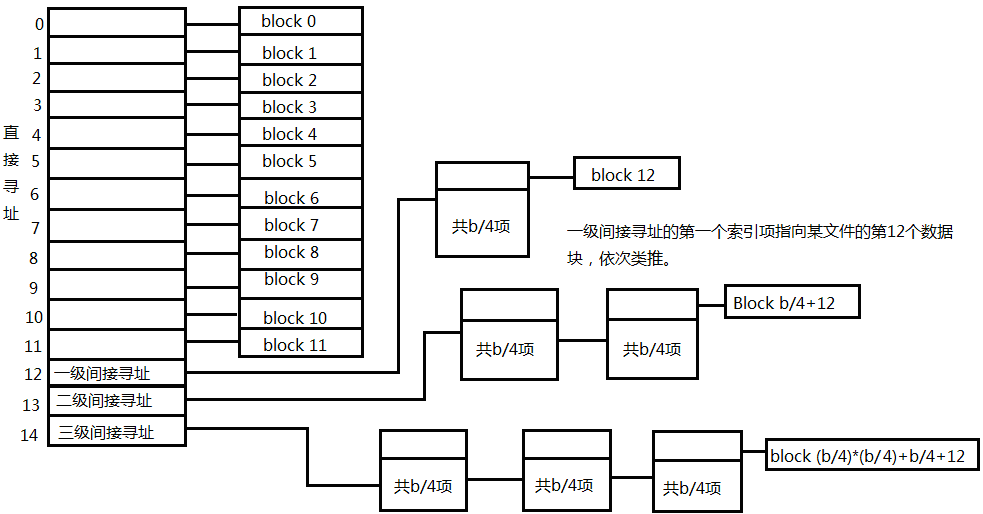
对于符号链接，如果目标路径名较短则直接保存在 inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

设备文件、FIFO和socket 等特殊文件没有数据块，即文件大小为0，设备文件的主设备号和次设备号保存在 inode中。

### 数据块寻址

如果一个文件有多个数据块，这些数据块很可能不是连续存放的，应该如何寻址到每个块呢？事实上，每个文件的inode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，例如若Blocks[0]字段保存着 24，就表示块号为24的块是该文件的数据块，如果块大小是 1KB，这样可以表示从 0字节到 12KB的文件。如果剩下的三个索引项 Blocks[12]到 Blocks[14]也是这么用的，就只能表示最大 15KB的文件了，这是远远不够的，事实上，剩下的三个索引项都是间接索引。

索引项Blocks[12]所指向的块并非数据块，而是称为间接寻址块（ Indirect Block），其中存放的都是类似 Blocks[0]这种索引项，再由索引项指向数据块。设块大小是 b，那么一个间接寻址块中可以存放 b/4个索引项，指向 b/4个数据块。所以如果把 Blocks[0]到 Blocks[12]都用上，最多可以表示 b/4+12个数据块，对于块大小是 1K的情况，最大可表示 268K的文件。如下图所示，注意文件的数据块编号是从 0开始的， Blocks[0]指向第 0个数据块， Blocks[11]指向第 11个数据块， Blocks[12]所指向的间接寻址块的第一个索引项指向第 12个数据块，依此类推。



图八 数据寻址过程

从上图可以看出，索引项 Blocks[13]指向两级的间接寻址块，总共最多可表示 (b/4)^2 +b/4+12个数据块，对于 1K的块大小最大可表示 64.26MB的文件。索引项 Blocks[14]指向三级的间接寻址块，总共最多可表示 (b/4)^3 +(b/4)^2 +b/4+12个数据块，对于 1K的块大小最大可表示 16.06GB的文件。

可见，这种寻址方式对于访问不超过 12个数据块的小文件是非常快的，访问文件中的任意数据只需要两次读盘操作，一次读 inode（也就是读索引项）一次读数据块。而访问大文件中的数据则需要最多五次读盘操作： inode、一级间接寻址块、二级间接寻址块、三级间接寻址块、数据块。实际上，磁盘中的 inode（索引节点高速缓存）和数据块（块高速缓存）往往已经被内核缓存了，读大文件的效率也不会太低。

## 系统编码

提供一些主要函数源代码，其他代码请参见源程序。

1. 读取数据

void Dialog::read\_disk\_data(const char \*file\_name, long sector\_number)

{

fd=::open(file\_name, O\_RDONLY);

if(fd>0&&sector\_number>=0)

{

::lseek64(fd,(sector\_number)\*SECTOR\_SIZE,SEEK\_SET);

if(read(fd, get\_a\_buffer(), SECTOR\_SIZE)>0)

ui->te\_show\_data->

setText(data\_formater(get\_a\_buffer(), SECTOR\_SIZE));

}

else

return;

}

1. 格式化数据

QString Dialog::data\_formater(char \*buf, int size)

{

//参见上一节

}

1. 跳转

void Dialog::on\_btn\_jump\_clicked()

/\*

Text a sector number and jump to it.

\*/

{

bool ok=false;

sector\_num=ui->et\_page\_num->toPlainText().toLong(&ok, 16);

if(sector\_num<0)

Dialog::call\_alert(0x950913);

if(ok)

{

::lseek64(fd,sector\_num\*SECTOR\_SIZE,SEEK\_SET);

if(read(fd,get\_a\_buffer(),SECTOR\_SIZE))

{

ui->te\_show\_data->setText(data\_formater(get\_a\_buffer(), SECTOR\_SIZE));

Dialog::w\_log(QString("Success!\JUMP clicked."

"\nRead number %1 psysicial sector.").arg(sector\_num));

}

else

{

Dialog::w\_log(QString("Failed!\JUMP clicked."

"\nCannot read %1 file.").arg(DISKFILE));

Dialog::call\_alert(0x950702);

}

}

current\_showed\_data=ui->te\_show\_data->toPlainText();

}

1. 跳转到启动扇区

long Dialog::jmp2\_boot\_block()

/\*

Move DISKFILE cusor to Boot Block head.

Offset Description:

Disk number

-- 0x1BE:Usually mark 80(hex).

Sector number(+0x08 from disk number)

-- 0x1C6: where partion0 starts. 00 08 00 00 (this disk)

\*/

{

Dialog::jmp2\_mbr();

::lseek64(fd,0x1C6,SEEK\_CUR); //00 00 08 00

char buf[0x04];

long l\_jmp2;

bool ok=false;

if(read(fd,buf,0x04))

{

QString str\_jmp2="";

for(int i=3; i>=0; i--)

{

qDebug("buf[%d]=%02X",i,buf[i]&0xFF);

str\_jmp2+=QString::asprintf("%02X",buf[i]&0xFF);//00000800

}

l\_jmp2=str\_jmp2.toLong(&ok,16); //800

if(&ok)

::lseek64(fd,l\_jmp2\*SECTOR\_SIZE,SEEK\_SET);

}

return l\_jmp2;

}

1. 截取指定长度字节

long Dialog::grab\_n\_bytes(int fd, int length)

/\*

function is as good as it's name.

\*/

{

char buf[length];

long l\_bytes;

if(read(fd,buf,length))

{

QString str\_bytes="";

bool ok=false;

for(int i=length-0x01;i>=0x00;i--)

{

str\_bytes+=QString::asprintf("%02X",buf[i]&0xFF);

}

l\_bytes=str\_bytes.toLong(&ok,16);

if(ok)

return l\_bytes;

}

else

return 0x961206;

}

1. 导出数据

void Dialog::on\_cb\_edit\_mode\_clicked()

{

bool is\_editable=false;

if(ui->cb\_edit\_mode->isChecked())

{

QMessageBox::StandardButton choice=

QMessageBox::warning(this,"Active edit mode",

"Are you sure active edit mode?\n",

QMessageBox::Yes|QMessageBox::No,

QMessageBox::No);

if(choice==QMessageBox::Yes)

{

ui->te\_show\_data->setReadOnly(false);

ui->te\_show\_data->setOverwriteMode(true);

is\_editable=true;

Dialog::w\_log("Edit mode opened!");

}

else

{

ui->cb\_edit\_mode->setChecked(false);

is\_editable=false;

}

}

else

{

ui->te\_show\_data->setReadOnly(true);

//check content wheather it has changed

if(ui->te\_show\_data->document()->isModified())

{

QMessageBox::StandardButton write\_or\_not=

QMessageBox::warning(this,"Text Modified",

"Text is Modified.\nDo you want to write it?",

QMessageBox::Yes|QMessageBox::No,

QMessageBox::No);

if(write\_or\_not==QMessageBox::Yes)

{

//write to disk.

ui->te\_show\_data->document()->undoAvailable(true);

Dialog::w\_log(QString("Writing data into \"%1\"....").arg(DISKFILE));

}

else

{

//undo the modification.

/\*Only can undo 1 step.

\* this need to be strengthened.

\* considering using Stack(Commands stack)

\*/

ui->te\_show\_data->undo();

}

}

Dialog::w\_log("Edit mode closed!");

}

}

## 测试机分区分布

表4 分区地址分布

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 物理扇区号 | 数据块号 | 长度(扇区) | 数据项目 | 备注 |
| 00000000h | - | 00000800h | 系统数据 | - |
| 分区00h开始 | | | | |
| 00000800h | 00000000h | 00000002h | Boot Block | 分区0引导数据块 |
| 00000802h | - | 00000002h | Super Block | 超级块 |
| 00000804h | - | 00000002h | 保留 | - |
| 00000806h | - | 00000002h | 保留 | - |
| 00000808h | 00000001h | 00000008h | Group Descriptor Table | 块组描述符表，20h |
| 00000810h | 00000002h | 00000008h | Group Descriptor Table | 块组描述符表，20h |
| 00000818h | 00000003h | 00000008h | Group Descriptor Table | 块组描述符表，20h |
| 块组00h开始 | | | | |
| 00000820h | 00000004h | 03FD\*08h | Data Blocks | 数据块1 |
| 00002808h | 00000401h | 00000008h | Block Bitmap | 块位图，1bit |
| 00002810h | 00000402h | 000Fh\*08h | Data Blocks | 数据块2、3..18 |
| 00002888h | 00000411h | 00000008h | i-node Bitmap | i-node位图，1bit |
| 00002890h | 00000412h | 000Fh\*08h | Data Blocks | 数据块19、20..34 |
| 00002908 | 00000421h | 0200h\*08h | i-node Table | i-node表,80h |
| 00003908 | 00000621h | 01E2h\*08h | Data Blocks | 数据块35.. |
| 块组01h开始 | | | | |
| 00040820h | 00008004h | ... |  |  |
| 块组02h开始 | | | | |
| 00080820h |  | ... |  |  |
| 块组12Fh开始 | | | | |
| 4BC0820h |  | ... |  |  |
|  |  |  |  |  |
| 分区00h结束，分区01h开始 | | | | |
| 4C00800h |  |  |  |  |

## 参考

* Linux EXT文件系统分析案例-Ubuntu
* http://blog.csdn.net/jnu\_simba/article/details/11759809