一、 概述

1. 什么是文件系统

文件系统是操作系统的重要组成部分，它含有大量的文件及其属性信息，负责对文件进行操纵和管理，并向用户提供一个使用文件的接口，不同的文件系统有不同的组织方式。

文件系统既要负责为用户提供对自己私有信息的方法，又要负责提供给用户访问共享信息的控制方式。

2. 常见文件系统的分类

- Windows：FAT，FAT16，FAT32，NTFS等
- Linux：EXT，EXT2，EXT3，EXT4等
- 其他：NFS, HPFS，HFS+等

3. EXT2的简单介绍

The Second Extended File System(ext2)文件系统于1993年1月加入Linux核心支持之中。最大可支持2TB的文件系统。

操作系统的文件数据除了文件实际内容外， 通常含有非常多的属性。文件系统通常会将这两部份的数据分别存放在不同的区块，权限与属性放置到inode中，至于实际数据则放置到 data block 区块中。 另外，还有一个超级区块 (superblock) 会记录整个文件系统的整体信息，包括inode与 block 的总量、使用量、剩余量等。

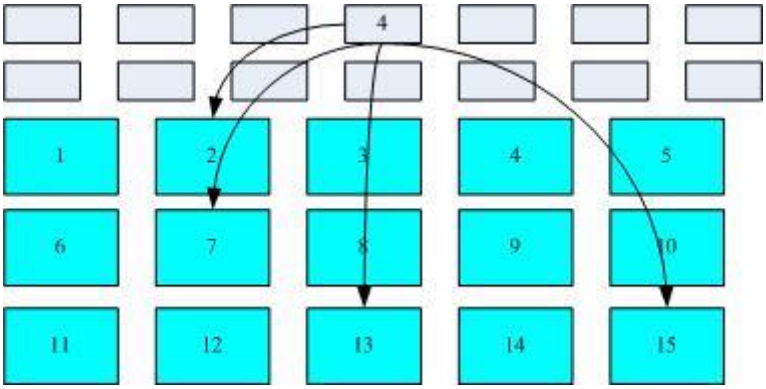
- superblock：记录此filesystem的整体信息，包括inode/block的总量、使用量、剩余量， 以及文件系统的格式与相关信息等；
- inode：记录文件的属性，一个文件占用一个inode，同时记录此文件的数据所在的 block 号码；
- block：实际记录文件的内容，若文件太大时，会占用多个 block。

由于每个inode与block都有编号，而每个文件都会占用一个inode，inode内则有文件数据放置的 block 号码。 因此，我们可以知道的是，如果能够找到文件的inode的话，那么自然就会知道这个文件所放置数据的block号码，当然也就能够读出该文件的实际数据了。这是个比较有效率的作法，因为如此一来我们的磁盘就能够在短时间内读取出全部的数据，读写的效能比较好。

4. EXT2和FAT的区别

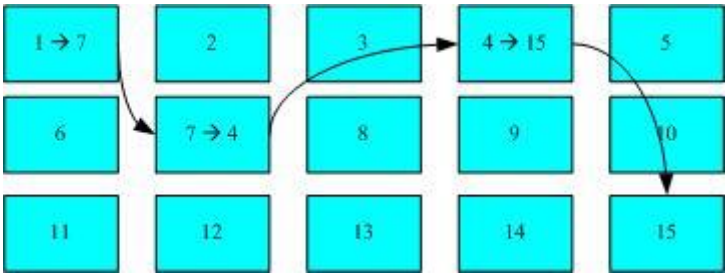
- EXT2

文件系统先格式化出 inode 与 block 的区块（inode 为灰色，block 为绿色），假设某一个文件的属性与权限数据是放置到 inode 4 号，而这个 inode 记录了文件数据的实际放置点为 2, 7, 13, 15 这四个 block 号码，此时我们的操作系统就能够据此来排列磁盘的阅读顺序，可以一口气将四个 block 内容读出来。

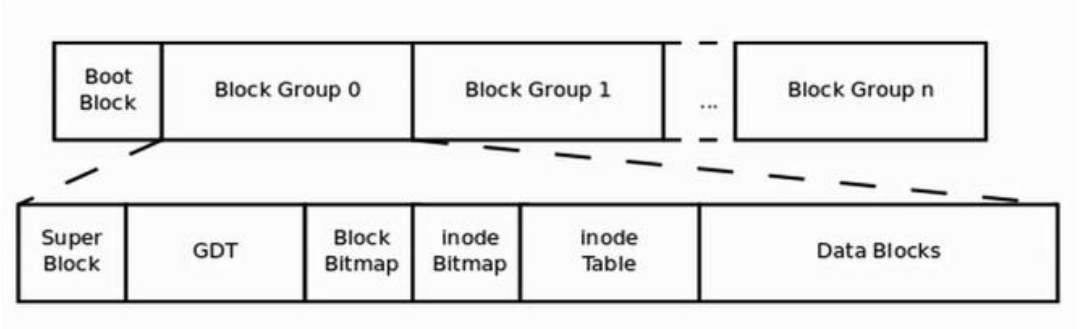


◆ FAT

我们假设文件的数据依序写入 1->7->4->15 号这四个 block 号码中， 但这个文件系统没有办法一口气就知道四个 block 的号码，他得要一个一个的将 block 读出后，才会知道下一个 block 在何处。 如果同一个文件数据写入的 block 分散的太过厉害时，则我们的磁盘读取头将无法在磁盘转一圈就读到所有的数据， 因此磁盘就会多转好几圈才能完整的读取到这个文件的内容！



5. EXT2 的结构



1) 总体描述

Ext2 文件系统在格式化的时候基本上是区分为多个区块群组 (block group) 的，每个区块群组都有独立的 inode/block/superblock 系统。

2) boot sector（启动扇区）

在整体的规划当中，文件系统最前面有一个启动扇区(boot sector)，这个启动扇区可以安装启动管理程序，如此一来我们就能够将不同的启动管理程序安装到个别的

文件系统最前端，而不用覆盖整颗硬盘唯一的 MBR，这样也才能够制作出多重引导的环境。

3) super block（超级块）

Superblock 是记录整个 filesystem 相关信息的地方，superblock 的大小一般为 1024bytes。记录的信息主要有：

- block 与 inode 的总量；
- 未使用与已使用的 inode / block 数量；
- block 与 inode 的大小 (block 为 1, 2, 4K，inode 为 128 bytes)；
- filesystem 的挂载时间、最近一次写入数据的时间、最近一次检验磁盘 (fsck) 的时间等文件系统的相关信息；
- 一个 valid bit 数值，若此文件系统已被挂载，则 valid bit 为 0 ，若未被挂载，则 valid bit 为 1 。

此外，每个 block group 都可能含有 superblock，而若含有 superblock 则该 superblock 主要是做为第一个 block group 内 superblock 的备份。

```
truct ext2_super_block {
    __le32    s_inodes_count;         /* 索引节点的总数 */
    __le32    s_blocks_count;         /* 块总数（所有的块） */
    __le32    s_r_blocks_count;       /* 保留的块数 */
    __le32    s_free_blocks_count;    /* 空闲块数 */
    __le32    s_free_inodes_count;    /* 空闲索引节点数 */
    __le32    s_first_data_block;     /* 第一个使用的块号（总为1 ?） */
    __le32    s_log_block_size;       /* 块的大小 */
    __le32    s_log_frag_size;        /* 片的大小 */
    __le32    s_blocks_per_group;     /* 每组中的块数 */
    __le32    s_frags_per_group;      /* 每组中的片数 */
    __le32    s_inodes_per_group;     /* 每组中的索引节点数 */
    __le32    s_mtime;                /* 最后一次安装操作的时间 */
    __le32    s_wtime;                /* 最后一次写操作的时间 */
    __le16    s_mnt_count;            /* 被执行安装操作的次数 */
    __le16    s_max_mnt_count;        /* 检查之前安装操作的次数 */
    __le16    s_magic;                /* 魔术签名 */
    __le16    s_state;                /* 文件系统状态标志 */
    __le16    s_errors;               /* 当检测到错误时的行为 */
    __le16    s_minor_rev_level;      /* 次版本号 */
    __le32    s_lastcheck;            /* 最后一次检查的时间 */
    __le32    s_checkinterval;        /* 两次检查之间的时间间隔 */
    __le32    s_creator_os;           /* 创建文件系统的操作系统 */
    __le32    s_rev_level;            /* 主版本号 */
    __le16    s_def_resuid;           /* 保留块的缺省 UID */
    __le16    s_def_resgid;           /* 保留块的缺省用户组 ID */
    __le32    s_first_ino;            /* 第一个非保留的索引节点号 */
    __le16    s_inode_size;           /* 磁盘上索引节点结构的大小 */
    __le16    s_block_group_nr;       /* 这个超级块的块组号 */
    __le32    s_feature_compat;       /* 具有兼容特点的位图 */
```

```
    __le32      s_feature_incompat;       /* 具有非兼容特点的位图 */
    __le32      s_feature_ro_compat;      /* 只读兼容特点的位图 */
    __u8        s_uuid[16];               /* 128位文件系统标识符 */
    char      s_volume_name[16];          /* 卷名 */
    char      s_last_mounted[64];         /* 最后一个安装点的路径名 */
    __le32    s_algorithm_usage_bitmap;   /* 用于压缩 */
    __u8      s_prealloc_blocks;          /* 预分配的块数 */
    __u8      s_prealloc_dir_blocks;      /* 为目录预分配的块数 */
    __u16     s_padding 1;                /* 按字对齐 */
    ……
    __u32     s_reserved[190];            /* 用null填充至1024字节 */
};
```

4) Group Description Table（块组描述表）

块组描述表是由一个个块组描述符组成的，有多少个块组就有多少个块组描述表。

- Block bitmap, inode bitmap, inode table的起始块
- Block 和 inode的剩余数量
- 目录的总量

```
struct ext2_group_desc{
    __le32    bg_block_bitmap;      /* 本块组的block bitmap从第几块开始*/
    __le32    bg_inode_bitmap;      /* 本块组的inode bitmap从第几块开始*/
    __le32    bg_inode_table;       /* 本块组的inode table从第几块开始*/
    __le16    bg_free_blocks_count; /* 本块组的空余块*/
    __le16    bg_free_inodes_count; /* 本块组的空余索引节点*/
    __le16    bg_used_dirs_count;   /*本块组的目录数量/
    __le16    bg_pad;
    __le32    bg_reserved[3];       /* 用null填充*/
};
```

5) Block bitmap

想要新增文件时总会用到 block ，当然选择空的 block 来记录新文件的数据。 那你怎么知道哪个 block 是空的？这就得要透过 block bitmap 的辅助了。从 block bitmap 当中可以知道哪些 block 是空的，block bitmap中的一个bit为标记一个block。 0代表未使用，1代表使用。

如果一个block的大小为1K，则一个block bitmap的大小为1024*8 = 8192，只能存放8192个block逻辑块，也就是8192*1K = 8M数据

6) Inode bitmap

功能和block bitmap相似，不过inode bitmap是记录inode是否可用。

一个inode保存一个文件信息，所以可以保存1024byte * 8bit = 8192个inode，也就是可以保存8192个文件

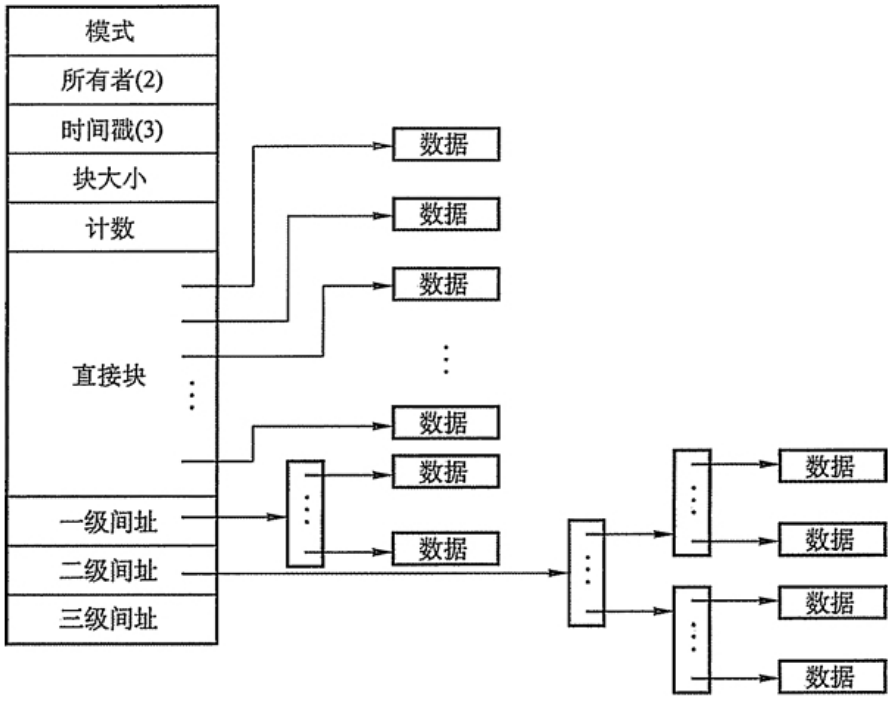
7) inode table（索引节点表）

Inode是在EXT2中指向数据块的“指针”，然后Inode保存着文件的属性；在EXT2中，文件的数据和属性是分开保存的，文件的属性保存在Inode表中，文件的数据保存在数据块block中。

- 每个 inode 大小均固定为 128 bytes（可以修改为128的倍数）

- 每个文件都仅会占用一个 inode 而已
- 该文件的存取模式(read/write/excute)
- 该文件的拥有者与群组(owner/group)
- 该文件的容量(size)
- 该文件创建或状态改变的时间(ctime)
- 最近一次的读取时间(atime)
- 最近修改的时间(mtime)
- 定义文件特性的旗标(flag)，如 SetUID...
- 该文件真正内容的指向 (pointer)

inode只有 128bytes ，假设我一个文件有 400MB 且每个 block 为 4K 时， 那么至少也要10万个 block 号码的记录呢，inode没有这么多可记录的信息。为此我们的系统将 inode 记录 block 号码的区域定义为12个直接，1个间接，1个双间接与1个三间接记录区。



我们来计算一下容量（假设block为1K，记录一个block号码需要4bytes）：

- 12个直接指向：12*1K = 12K
- 1个间接指向：1*(1024/4)*1K = 256K
- 1个双间接指向：256K*256K = 64M
- 1个三间接指向：256K*256K*256K = 16G

所以我们可以存放16G左右的数据

```
struct ext2_inode {
    __le16   i_mode;                /* File mode 文件模式*/
    __le16   i_uid;                 /* 拥有者 UID */
    __le32   i_size;                /* 文件大小*/
    __le32   i_atime;               /* Access time 最近访问时间*/
    __le32   i_ctime;               /* Creation time 创建时间*/
```

```
    __le32   i_mtime;                /* Modification time 修改时间*/
    __le32   i_dtime;                /* Deletion Time 删除时间*/
    __le16   i_gid;                  /* 用户组 ID*/
    __le16   i_links_count;          /* Links count  连接数*/
    __le32   i_blocks;               /* 物理块的数量 */
    __le32   i_flags;                /* 打开文件的方式 */
    __le32   i_block[EXT2_N_BLOCKS]; /* 至多可以有 15 个“指针”*/
    __le32   i_generation;           /* 文件版本 */
    __le32   i_file_acl;             /* 文件访问权限 */
    __le32   i_dir_acl;              /* 目录访问权限 */
    __u8     l_i_frag;               /* 每块中的片数*/
    __u8     l_i_fsize;              /* 片大小 */
    __u32    l_i_reserved;           /* 保留 */
};
```

8) data block（数据块）

data block 就是用来真正放置文件数据的地方，在 Ext2 文件系统中所支持的 block 大小有 1K,2K 及 4K 三种。

Block 大小	1KB	2KB	4KB
最大单一文件限制	16GB	256GB	2TB
最大文件系统总容量	2TB	8TB	16TB

- 每个 block 内最多只能够放置一个文件的数据；
- 如果文件大于 block 的大小，则一个文件会占用多个 block 数量；
- 若文件小于 block ，则该 block 的剩余容量就不能够再被使用了。

6. EXT2 的目录与文件

在 Linux 中不管建立一个文件还是目录都会占一个 inode。

1) 文件：

当我们在 Linux 下的 ext2 创建一个一般文件时， ext2 会分配一个 inode 与相对于该文件大小的 block 数量给该文件。例如：假设我的一个 block 为 1 Kbytes，而我要创建一个 100 KBytes 的文件，那么 linux 将分配一个 inode 与 100 个 block 来存储该文件。

2) 目录：

当我们在 Linux 下的 ext2 文件系统创建一个目录时，ext2 会分配一个 inode 与至少一块 block 给该目录。其中，inode 记录该目录的相关权限与属性，并可记录分配到的那块 block 号码；而 block 则是记录在这个目录下的文件名与该文件名占用的 inode 号码数据。

如果一个目录，则在该 block 应该存放如下数据结构

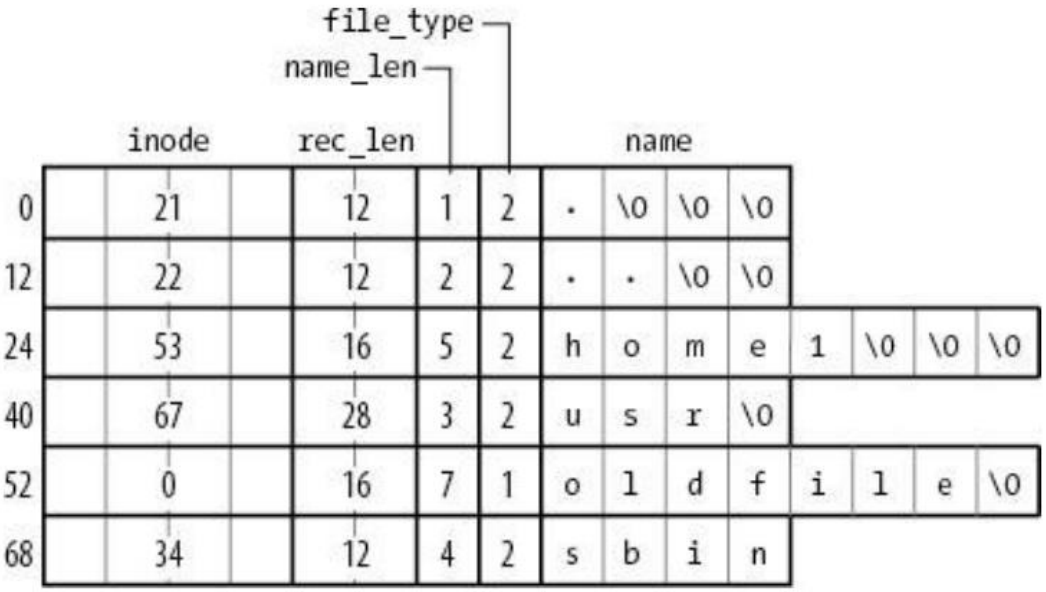
```
struct ext2_dir_entry{
    __u32    inode;      /* 索引节点号 */
    __u16    rec_len;    /* 目录项长度 */
    __u8     name_len;   /* 文件名长度 */
```

```
    __u8     file_type;  /* 文件类型 */
    char[]   name[255];  /* 文件名 */
};
```

文件类型有如下几种：

文件类型号	含义	文件类型	含义
0	未知	4	块设备
1	普通文件	5	管道(pipe)
2	目录	6	套接字
3	字符设备	7	符号指针

一个简单的组织方式



如果删除了某个文件，那么将其 inode 设置为 0，然后将 oldfile 的目录项的长度加到/usr 的目录项长度上。这样做可以避免因删除目录而产生的 IO 开销。

3) 目录树

如果我要寻找/etc/passwd 这个文件

a) 找到 GDT，获得 inode table 的起始块号

b) 找到 inode table 所在的这个块，EXT2 规定第二个 Inode 才属于根目录的

c) / 的 inode：以上得知根目录“/”的 inode 为 2

d) / 的 block：取得 block 的号码，并找到 etc/ 目录的 inode 号码

e) etc/ 的 inode：读取 inode 得知 etc/ 的 block 内容

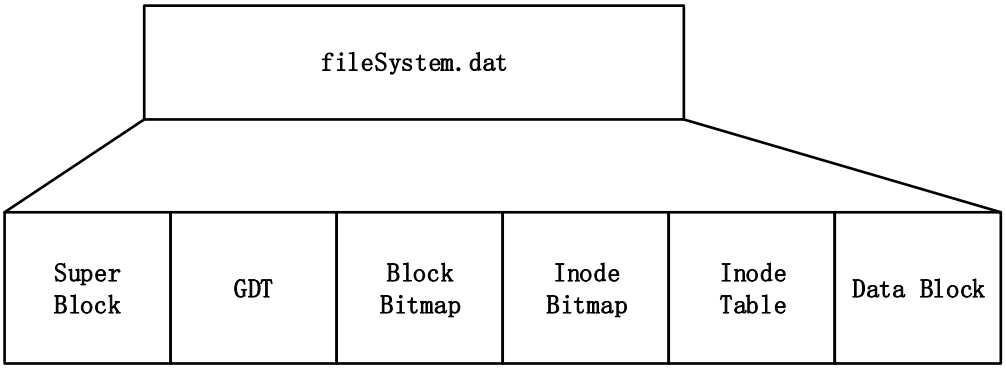
f) etc/ 的 block：取得 block 号码，并找到 passwd 文件的 inode 号码
g) passwd 的 inode：读取 inode 得知 passwd 的 block 内容
h) passwd 的 block：最后将该 block 内容的数据读出来

二、 数据结构

EXT2 文件系统是一个实际可用的文件系统，由以上真实系统的数据结构可以看出，体系实在是过于庞大，此次试验是为了进行简单的模拟，基于 EXT2 的思想和算法，设计一个简单的 Linux Ext2 的文件系统，实现 Ext2 文件系统的一个功能自己，并且用现有操作系统上的文件来代替硬盘进行硬件模拟。

将新建一个文件“fileSystem.dat”来模拟真实硬盘，将文件系统的所有信息都保存在此文件中，等再次启动程序的时候，只需要读取文件，以前在硬盘里建的东西都还存在，比较简单的真实的还原了 Linux Ext2 文件系统。

在此文件系统中没有设计 boot sector 启动扇区，因为没有引导程序，并且只设计了一个 super block 和一个 GDT 块组，然后文件系统的结构精简为下图：



1. 用户和组

为了简单而言，省去了权限控制。只设置了一个 root 用户和一个 root 用户组，也就是拥有所有权限，但是可以用命令改变权限。改变权限对 root 超级用户没有意义，所以没有权限的改变没有任何影响，只是输出不一样。方便以后的功能的扩充。

2. inode 和 block 的大小

- inode：64bytes，inode 号从 1 开始，具体请参考 inode table 设计
- block：1024byes，block 号从 0 开始，将逻辑块大小设置为 1K，由于用了一个 fileSystem.dat 文件来模拟一个磁盘，所以不用设计物理块大小（扇区大小为 512bytes）。

3. Super Block（超级块）

定义了一个超级块，大小为64bytes，占一个block，存放文件系统的基本信息，用命令“dumpe2fs”可以查看到这些信息：

- 卷名：“EXT2”
- inode和block的总数
  - inode：8192
    inode bitmap的大小为1个block，所以inode总数为1024*8=8192
  - block：1+1+1+1+512+8192 = 8708
    block总数需要把所有结构的block总数加起来，所以不仅包括data block的1024*8=8192个block数，还应该加上其他结构的block数，见下表（关于数据怎么来的请参考下面每个部分）：

	Super Block	GDT	Block Bitmap	Inode Bitmap	Inode Table	Data Block
单位bytes	64	32	1024	1024	64	1024
占用block块数	1	1	1	1	512	8192
占用的bytes总数	1K	1K	1K	1K	512K	8192K
存放的起始位置（bytes）	0	1024	2048	3072	4096	528384

- inode和block的剩余数
  因为设置的文件系统只有一个Super Block和一个GDT，所以两者的inode和block剩余数都应该一样的，并且是同步的。
  - inode：8191，假设系统的inode号从1开始
  - block：8192，假设系统的block号从0开始
- block和inode的大小
- 建立文件系统的时间：存放文件系统格式化的当时时间

```
struct super_block{
    char sb_volume_name[16];            //卷名 “EXT2”
    unsigned int sb_inodes_count;       //inode总数 = 8*1024=8192
    unsigned int sb_blocks_count;       //block总数 = 8708
    unsigned int sb_free_inodes_count;  //剩余inode数
    unsigned int sb_free_blocks_count;  //剩余block数
    unsigned int sb_block_size;         //1024byte
    unsigned int sb_inode_size;         //64byte
    char sb_wtime[20];                  //2016-06-01  00:00:00
    char sb_pad[4];                     //填充至64bytes
};
```

4. Group Description Table（块组描述表）

定义一个GDT，大小为32bytes，占用一个block，存放以下信息：

- block bitmap从第几个字节开始：2048
- inode bitmap从第几个字节开始：3072
- inode table从第几个字节开始：4096

- inode，block的剩余数：和super block的相同
- 本块组使用的目录总数（不包括“..”和“.”）

```
struct block_group_desc{
    unsigned int bgd_block_bitmap_start;  //block bitmap开始字节数地址
    unsigned int bgd_inode_bitmap_start;  //inode bitmap开始字节数地址
    unsigned int bgd_inode_table_start;   //inode table 开始字节数地址
    unsigned int bgd_free_inode_count;    //inode剩余数
    unsigned int bgd_free_block_count;    //block剩余数
    unsigned int bgd_used_dirs_count;     //目录总数
    char bgd_pad[8];                      //填充至32bytes
};
```

5. Block Bitmap

Block Bitmap占用一个block，大小即为1024bytes，可以存放1024bytes*8=8192个block号码，也就是一共有8192个数据块来存放数据（block编号直接从0开始）。我们计算一下一共可以保存多大的数据呢：8192*1K=8M。只能保存8M的数据。

所以一个block号最大为8192，存放一个block号的大小只需要2个字节足够。

6. Inode Bitmap

Inode Bitmap占用一个block，大小即为1024bytes，可以存放1024bytes*8=8192个inode号码，也就是一共可以建立8191个文件（编号0的inode号不使用，inode编号从1开始，Linux规定从2开始，这里方便简单就从1开始）。

7. Inode Table

由上面可以知道，我们将inode大小设置为64bytes，inode里面存放了如下信息：

- 文件的权限 rwx rwx rwx：

  r(read)可读，w(write)可写，x(execute)可执行

  每三个分别代表这个文件的所属用户(owner)，所属用户组(group)，以及其他人(other)对这个文件可执行的权限控制。由于这个文件系统只设置了一个root超级管理员用户，所以登录文件系统时，对任何文件都是可读可写可执行。但是任然可以修改此文件权限，方便以后功能的加入。

  为了方便，目录的权限初始化为755，文件的权限初始化为777
- 该文件或者目录的大小 size
  - 目录：因为目录下存放的都是inode指针，1个目录项的大小为16bytes，所以目录的大小也是16的倍数，但是最小也是32bytes，因为一个目录下默认包含“.”(指向本机目录)和“..”（指向上级目录）这两个特殊目录。一个目录的最大大小应该为1024*8 = 8192。

- 文件：存放的文件自己的真实大小，为了便于查看文件的真实大小，这里不设置为block大小的倍数。

➢ 访问时间 atime
- 此功能没有设置（待开发）
- 目录：读取文件或者执行文件时的更改（cd不会引起atime的改变，但ls读取了文件信息，所以会改变）
- 文件：读取文件或者执行文件时更改（任何对inode的访问都会使此处改变）

➢ 创建时间 ctime
- 目录：当目录的属性改变
- 文件：当文件的属性改变

➢ 修改时间 mtime
- 此功能没有设置（待开发）
- 目录：对目录下的文件新建，删除
- 文件：修改了文件的内容

➢ 删除时间 dtime
- 此功能没有设置（待开发）
- 目录和文件：保存删除时间

➢ 该inode占用的block的数量
- 目录：1个
- 文件：至少1个，以1K为单位增长

➢ 存放的block号：
- 目录：则在这个block存放的是目录项，一个目录项占16bytes，一个block可以存放1024/16 = 64个目录项，一共有8个block，所以一个目录项可以存放64*8 = 512个目录。具体请参考dir_entry目录项说明
- 文件：则在这里存放的是文件里的所有数据，由于系统设计的复杂性不高，所以我们取消了EXT2原有的inode指针方式（12个直接，1个间接，1个双间接与1个三间接记录区，可以存放16G的数据），那我们现在的inode指针方式如下：我们只有8个直接记录点，也就是直接指向空闲的8个block，一个block占1K，所以我们一个文件只能够存放1K*8 = 8K的数据，所以比原来的大大缩减了数据存放，但是不要紧，我们这个存放一些简短的txt文件还是没有什么问题的，我们只是为了做实验嘛。

```
struct inode{
    unsigned int i_mode;  //文件权限
    unsigned int i_size;  //文件或者目录大小
    unsigned int i_atime;  //访问时间
    char i_ctime[20];      //创建时间
    unsigned int i_mtime;  //修改时间
    unsigned int i_dtime;  //删除时间
    unsigned int i_blocks_count; //占用的block的数量
    unsigned short i_block[8];  //内容指向，存放block号
    char i_pad[4];  //填充
```

```
};
```

8. Data Block

block bitmap为1024bytes，真正存放数据的地方是data block这个数据块。1个bit位可以表示1个block的使用状态，现在通过block bitmap得知一共可以表示1024*8=8192个block的使用状态。所以data block的块数为8192个，一共占8192K=8M的容量。

9. 目录和文件

对于文件而言，这个数据结构可以忽略，文件没有目录体，文件只是属于目录项中的一项。

目录作为Linux的特殊文件，我们将根目录“/”的inode设置为1，直接指向的block数据块设置为0。所有的文件都在根目录下保存，所以每个文件都存放在当前目录下的目录项中。

我们现在用简单的16bytes字节来表示目录项的数据结构。1个block就可以存放1024/16 = 64个目录项

注意：只有目录才有目录项结构，并且目录的inode信息存放着目录项的block号，通过这个block号来找到目录下的指定文件/目录的inode。

- 索引节点号
  - 保存这个文件/目录的inode号
- 目录项长度
  - 这个代表这个目录项的长度。以我们这个数据结构为例：
    索引节点号为2个字节，文件名为2个字节，文件类型1个字节，文件名9个字节，所以目录项的长度为2+2+1+9=14个字节
  - 当然真正的目录项长度不可能定死为一个指定字节（最小为12bytes，以4个字节为单位增长，具体见概述中EXT2的目录和文件），我们这里只是为了方便简单，把文件名定为9个以内字节，一般来说都为255个字节，这样以4个字节为单位增长目录项长度（比如：6个字节的文件名为8个字节）
- 文件名长度
  - 一个文件名的实际长度字节数，并不是4个字节为单位
- 文件类型
  - 文件类型颇多，我们的文件系统只是用了其中两种
  - 文件：1
  - 目录：2
- 文件名
  - 存放文件名，但是有个缺陷，我们的文件名只能存9个字节，因为我们为了在一个block里面存放更多的目录项。就没有把目录项凑成32个字节

```
struct dir_entry{
    unsigned short inode;    //索引节点号
    unsigned short rec_len;  //目录项长度
    unsigned short name_len; //文件名长度
    char file_type;          //文件类型（1：普通文件，2：代表目录）
    char name[9];            //文件名
};
```

10. 宏定义

因为函数中要调用大量的数字，这样会很模糊不清，而且如果要改程序的数据结构会很不方便，所以对一些数据采用宏定义的方式，方便修改以及能够做到一目了然是什么意思。

```
#define BLOCK_SIZE 1024              // block 块大小
#define BLOCK_COUNT 8708             // block 块的总数量

#define SUPER_BLOCK_SIZE 64          // Super Block 的大小
#define BLOCK_GROUP_DES_SIZE 32      // GDT 的大小
#define INODE_TABLE_SIZE 524288      // Inode Table 的总大小
#define INODE_SIZE 64                // Inode Table 中 inode 的大小

#define SUPER_BLOCK_START 0          // Super Block 的起始位置
#define BLOCK_GROUP_DES_START 1024   // GDT 的起始位置
#define BLOCK_BITMAP_START 2048      // Block Bitmap 的起始位置
#define INODE_BITMAP_START 3072      // Inode Bitmap 的起始位置
#define INODE_TABLE_START 4096       // Inode Table 的起始位置
#define DATA_BLOCK_START 528384      // Block data 的起始位置

#define VOLUME_NAME "EXT2"           // 卷名

#pragma comment(lib, "WS2_32.lib")  // 这个是 windows API 所调用
                                     // 具体参考功能函数 ifconfig()(
```

11. 全局变量

有些变量调用很频繁，所以用全局变量来声明比较好。有一些全局变量是专门用来做缓冲区了，这样是为了避免不正当操作，直接把文件中的数据改掉，所以先缓存到一个变量中，然后再把变量写到文件系统中。

为什么缓冲区定义的是一个数组而不是一个指针？因为定义一个指针还需要用 malloc 分配空间才能使用，但是用一个数组表示就自动分配一个空间。并且和指针的作用是一样的。

```
static char block_buffer[1024];  // 1 个 block 块的内容缓存区域
char filebuffer[8192]="";        // 用 vi 保存文件时的缓存区域
```

```
struct super_block super_block_buffer[1];           // 超级块缓冲区
struct block_group_desc block_group_desc_buffer[1]; // 组描述符缓冲区
struct inode inode_buffer[1];                       // inode缓冲区 */

struct dir_entry dir[64];     //1个block中可以保存64个目录项

unsigned char block_bitmap_buffer[1024]=""; //block bitmap缓冲区
unsigned char inode_bitmap_buffer[1024]=""; //inode bitmap缓冲区

unsigned int last_inode_bit=1;  //记录最后一次的inode bit
unsigned int last_block_bit=0;  //记录最后一次的block bit
unsigned int current_dir_inode;  //当前目录的inode
unsigned int current_dir_length; //当前目录的长度

//路径名“[root@sking ~]# ”由以下三个部分拼接
char current_path[256]="";       // 当前路径名  [root@sking
char path_last[4]="";            // ]#
char path_name[241]="";          // 把中间~的替换为当前路径

char time_now[32] = "";          // 存储当前时间,再保存在文件属性当中
FILE *pf;                        // 定义文件指针，随时读写文件
```

三、 函数

对于用户而言，函数对于他们来说是隐形的，他们更关心的是如何使用命令来对文件系统进行操作，因为文件系统的函数比较多，所以分成两个部分，一个是系统函数，一个是功能函数。

系统函数一般是对于文件系统底层的功能，包括对数据的读写操作，索引节点的分配和释放，数据块的分配和释放，以及怎么找到Super Block，GDT这些的位置。这些函数是用户不能直接调用的。

功能函数和系统函数不同，功能函数代码更多，它相当于一个用户和文件系统底层的API接口，用户通过输入命令来调用专门的功能函数，然后功能函数调用各种系统函数，反馈给用户一个结果。所以功能函数相当于一个入口，来实现的特定的功能。

这次设计的简单文件系统会给用户提供一共16个Linux命令来进行调用，基本上还原了真实的Linux ext2文件系统的命令。可能交互性不是特别友好，也没有太多的合法性检查，所以不懂得直接用help进行查看命令，或者查看后续的用户手册，可以完成对此文件系统的检验。

1. 系统函数

1) read_super_block();

函数原型：void read_super_block(void)
函数功能：从磁盘中读出Super Block 的内容到缓冲区super_block_buffer

```
void read_super_block(void){
    fseek(pf, SUPER_BLOCK_START, SEEK_SET);
    fread(super_block_buffer, SUPER_BLOCK_SIZE, 1, pf);
}
```

2) write_super_block(); 将缓冲区的super_block 写入磁盘

函数原型：void write_super_block(void)
函数功能：将缓冲区super_block_buffer中的内容写入磁盘中的super_block

```
void write_super_block(void){
    fseek(pf, SUPER_BLOCK_START, SEEK_SET);
    fwrite(super_block_buffer, SUPER_BLOCK_SIZE, 1, pf);
    fflush(pf);
}
```

3) read_block_group_desc();

函数原型：void read_block_group_desc (void)
函数功能：从磁盘中读出GDT的内容到缓冲区block_group_desc_buffer

```
void read_block_group_desc(void){
    fseek(pf, BLOCK_GROUP_DES_START, SEEK_SET);
    fread(block_group_desc_buffer, BLOCK_GROUP_DES_SIZE, 1, pf);
}
```

4) write_block_group_desc();

函数原型：void write_block_group_desc (void)
函数功能：将缓冲区block_group_desc_buffer中的内容写入磁盘中的GDT

```
void write_block_group_desc(void){
    fseek(pf, BLOCK_GROUP_DES_START, SEEK_SET);
    fwrite(block_group_desc_buffer, BLOCK_GROUP_DES_SIZE, 1, pf);
    fflush(pf);
}
```

5) read_block_bitmap();

函数原型：void read_block_bitmap(void)
函数功能：从磁盘中读出Block Bitmap到缓冲区block_buffer

```
void read_block_bitmap(void){
    fseek(pf, BLOCK_BITMAP_START, SEEK_SET);
    fread(block_bitmap_buffer, BLOCK_SIZE, 1, pf);
}
```

6) write_block_bitmap();

函数原型：void write_block_bitmap(void)
函数功能：将缓冲区block_buffer中的内容写入磁盘的Block Bitmap

```
void write_block_bitmap(void){
    fseek(pf, BLOCK_BITMAP_START, SEEK_SET);
    fwrite(block_bitmap_buffer, BLOCK_SIZE, 1, pf);
    fflush(pf);
}
```

7) read_inode_bitmap();

函数原型：void read_inode_bitmap (void)
函数功能：从磁盘中读出Inode Bitmap到缓冲区inode_buffer

```
void read_inode_bitmap(void){
    fseek(pf, INODE_BITMAP_START, SEEK_SET);
    fread(inode_bitmap_buffer, BLOCK_SIZE, 1, pf);
}
```

8) write_inode_bitmap();

函数原型：void write_inode_bitmap (void)
函数功能：将缓冲区inode_buffer中的内容写入磁盘的Inode Bitmap

```
void write_inode_bitmap(void){
    fseek(pf, INODE_BITMAP_START, SEEK_SET);
    fwrite(inode_bitmap_buffer, BLOCK_SIZE, 1, pf);
    fflush(pf);
}
```

9) read_inode();

函数原型：void read_inode(int inode_num)
函数功能：inode_num为该文件的inode号码，从磁盘中读出Inode Table中该inode信息到缓冲区inode_buffer，注意inode号是从1开始

```
void read_inode(int inode_num){
    fseek(pf, INODE_TABLE_START+(inode_num-1)*INODE_SIZE, SEEK_SET);
    fread(inode_buffer, INODE_SIZE, 1, pf);
}
```

10) write_inode();

函数原型：void write_inode (int inode_num)
函数功能：inode_num为该文件的inode号，将inode_buffer中的内容写入磁盘中该inode的存储位置，注意inode号是从1开始

```
void write_inode(int inode_num){
    fseek(pf, INODE_TABLE_START+(inode_num-1)*INODE_SIZE, SEEK_SET);
    fwrite(inode_buffer, INODE_SIZE, 1, pf);
    fflush(pf);
}
```

11) read_block();

函数原型：void read_block(int block_num)
函数功能：block_num为block号，从磁盘中读出Data Block中block号里保存的内容到缓冲区block_buffer

```
void read_block(int block_num){
    fseek(pf, DATA_BLOCK_START+block_num*BLOCK_SIZE, SEEK_SET);
    fread(block_buffer, BLOCK_SIZE, 1, pf);
}
```

12) write_block();

函数原型：void write_block(int block_num)
函数功能：block_num为block号，将缓冲区block_buffer的内容写入指定的block块中

```
void write_block(int block_num){
    fseek(pf, DATA_BLOCK_START+block_num*BLOCK_SIZE, SEEK_SET);
    fwrite(block_buffer, BLOCK_SIZE, 1, pf);
```

```
    fflush(pf);
}
```

13) read_dir();

函数原型：void read_dir(int block_num)
函数功能：block_num为block号，目录的inode信息中i_block[8]存放着对应的8个block块号，每个块号存放着64个dir_entry目录项信息，此函数就是讲这个64个目录项信息读到缓冲区dir[64]这个结构体数组当中。

```
void read_dir(int block_num){
    fseek(pf, DATA_BLOCK_START+block_num*BLOCK_SIZE, SEEK_SET);
    fread(dir, BLOCK_SIZE, 1, pf);
}
```

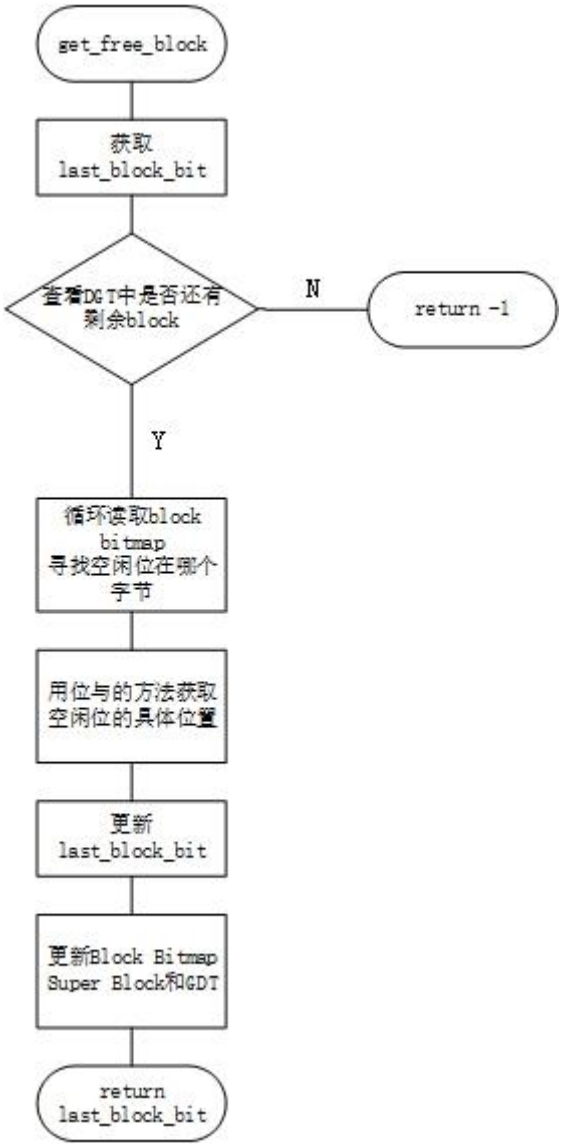
14) write_dir();

函数原型：void write_dir(int block_num)
函数功能：block_num为block块号，将缓冲区dir[64]这个结构体数组的信息写入到执行的block数据块中

```
void write_dir(int block_num){
    fseek(pf, DATA_BLOCK_START+block_num*BLOCK_SIZE, SEEK_SET);
    fwrite(dir, BLOCK_SIZE, 1, pf);
    fflush(pf);
}
```

15) get_free_block();

函数原型：unsigned int get_free_block(void)
函数功能：返回一个Data Block中空闲的block号码，从Block Bitmap寻找
函数说明：流程图如下

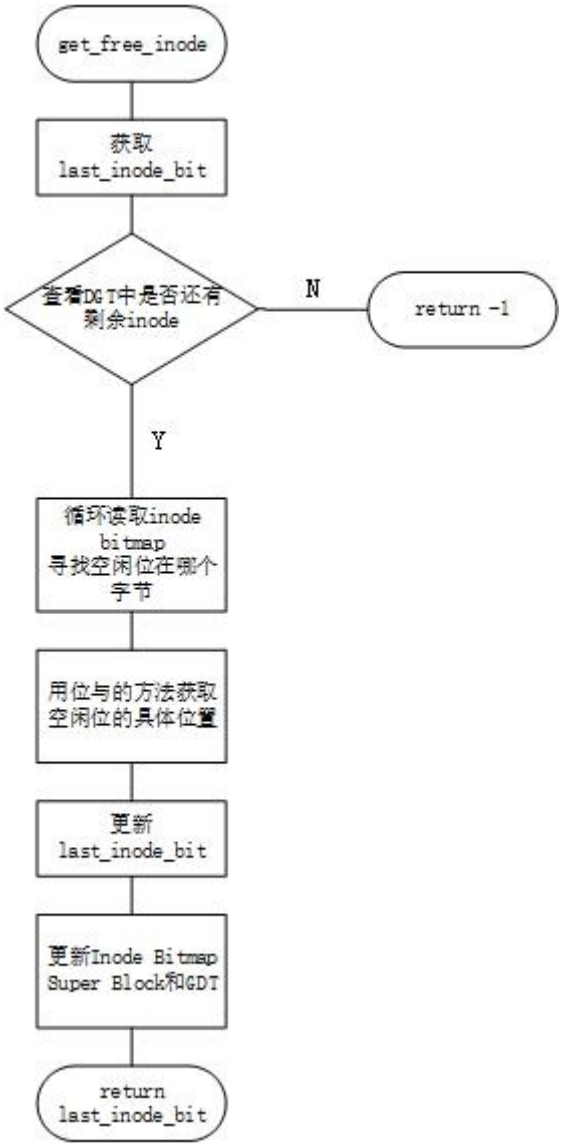



```
unsigned int get_free_block(void){

    unsigned int temp_block_bit = last_block_bit;  //记录上一次分配的block
号
    unsigned int temp = temp_block_bit/8;  //8bit = 1char
    char flag = 0;
    unsigned char con = 128;
    read_block_group_desc();

    //如果没有剩余block直接跳出
    if(block_group_desc_buffer->bgd_free_block_count==0){
        printf("there is no free block.\n");
        return -1;
    }

    //将block bitmap读出到  block_bitmap_buffer[]
```

```
    //查看是否哪一块有bit位空缺，因为一个文件一共有8个索引块
    read_block_bitmap();
    while(block_bitmap_buffer[temp] == 255){
        if(temp == 1023)
            temp=0;
        else
            temp++;
    }

    //用位与的方法，得到空闲的bit位
    while(block_bitmap_buffer[temp]&con){
        flag++;
        con /= 2;
    }

    //得到bit位之后，直接修改整个char值
    block_bitmap_buffer[temp] = block_bitmap_buffer[temp]+con;
    last_block_bit = temp*8+flag;
    write_block_bitmap(); //更新到block bitmap

    //更新到Super Block
    read_super_block();
    super_block_buffer->sb_free_blocks_count--;
    write_super_block();

     //更新到GDT
    block_group_desc_buffer->bgd_free_block_count--;
    write_block_group_desc();
    //printf("%u", last_block_bit);  //检验输出
    return last_block_bit;
}
```

16) get_free_inode();

函数原型：unsigned int get_free_inode(void)
函数功能：返回一个Inode Table中空闲的inode号码，从Inode Bitmap寻找
函数说明：过程和get_free_block相似，不过inode号是从1开始的




```
unsigned int get_free_inode(void){
    unsigned int temp_inode_bit = last_inode_bit; //最开始等于 1
    unsigned int temp = (temp_inode_bit-1)/8;  //8bit = 1char

    char flag = 0;
    unsigned char con = 128; //1000 0000b

    read_block_group_desc();
    if(block_group_desc_buffer->bgd_free_inode_count==0){
        printf("there is no free inode.\n");
        return -1;
    }

    //将 inode bitmap 读出到  inode_bitmap_buffer[]
    read_inode_bitmap();
    while(inode_bitmap_buffer[temp] == 255){
```

```
        if(temp == 1023)
            temp=0;
        else
            temp++;
    }

//  printf("temp_inode_bit: %d\n", temp_inode_bit);
//  printf("temp: %d\n", temp);
//  printf("inode_bitmap_buffer[temp]: %d\n", inode_bitmap_buffer[temp]);
//  printf("con: %d\n", con);
    //把while写成了if,醉了
    while(inode_bitmap_buffer[temp]&con){
        flag++;
        con /= 2;
        //printf("con: %d\n", con);
    }

    inode_bitmap_buffer[temp] = inode_bitmap_buffer[temp]+con;
    //printf("inode_bitmap_buffer[temp]: %d", inode_bitmap_buffer[temp]);
    last_inode_bit = temp*8+flag+1;
    write_inode_bitmap(); //更新到inode bitmap

    //更新到Super Block
    read_super_block();
    super_block_buffer->sb_free_inodes_count--;
    write_super_block();

    block_group_desc_buffer->bgd_free_inode_count--;
    write_block_group_desc(); //更新到块组描述

    //printf("%d\n", flag);
    //printf("%d\n", con);
    //printf("%u\n", last_inode_bit);
    return last_inode_bit;
}
```

17) remove_block();

函数原型：void remove_block(int remove_block_bit)
函数功能：remove_block_bit代表需要删除的block号码，从Block Bitmap中把指定的bit位改为0，表示该bit位代表的block没有使用。
函数说明：见下面流程图


```
void remove_block(int remove_block_bit){
    unsigned temp = remove_block_bit/8;

    read_block_bitmap();
    //找到bit位所属的字节然后和特定的数字相与就可以将它置0
    switch(remove_block_bit%8){
        case 0: block_bitmap_buffer[temp]&127;break; //0111 1111
        case 1: block_bitmap_buffer[temp]&191;break; //1011 1111
        case 2: block_bitmap_buffer[temp]&223;break; //1101 1111
        case 3: block_bitmap_buffer[temp]&239;break; //1110 1111
        case 4: block_bitmap_buffer[temp]&247;break; //1111 0111
        case 5: block_bitmap_buffer[temp]&251;break; //1111 1011
        case 6: block_bitmap_buffer[temp]&253;break; //1111 1101
        case 7: block_bitmap_buffer[temp]&254;break; //1111 1110
    }

    //更新Block Bitmap
    write_block_bitmap();

    //更新GDT
    read_block_group_desc();
    block_group_desc_buffer->bgd_free_block_count++;
    write_block_group_desc();

    //更新Super Block
    read_super_block();
    super_block_buffer->sb_free_blocks_count--;
    write_super_block();
```

```
}
```

18) remove_inode();

函数原型：void remove_inode(int remove_inode_bit)
函数功能：remove_inode_bit 代表需要删除的block 号码，从 Inode Bitmap 中把指定的bit 位改为0，表示该bit 位代表的 inode 没有使用。
函数说明：基本和remove_block()相似，但是 inode 号是从1开始的


```
void remove_inode(int remove_inode_bit){
    unsigned temp = (remove_inode_bit-1)/8;

    read_inode_bitmap();
    switch((remove_inode_bit-1)%8){
        case 0: inode_bitmap_buffer[temp]&127;break; //0111 1111
        case 1: inode_bitmap_buffer[temp]&191;break; //1011 1111
        case 2: inode_bitmap_buffer[temp]&223;break; //1101 1111
        case 3: inode_bitmap_buffer[temp]&239;break; //1110 1111
        case 4: inode_bitmap_buffer[temp]&247;break; //1111 0111
        case 5: inode_bitmap_buffer[temp]&251;break; //1111 1011
        case 6: inode_bitmap_buffer[temp]&253;break; //1111 1101
        case 7: inode_bitmap_buffer[temp]&254;break; //1111 1110
    }

    //更新 Inode Bitmap
    write_inode_bitmap();

    //更新 GDT
```

```
    read_block_group_desc();
    block_group_desc_buffer->bgd_free_inode_count++;
    write_block_group_desc();

    //更新Super Block
    read_super_block();
    super_block_buffer->sb_free_inodes_count++;
    write_super_block();
}
```

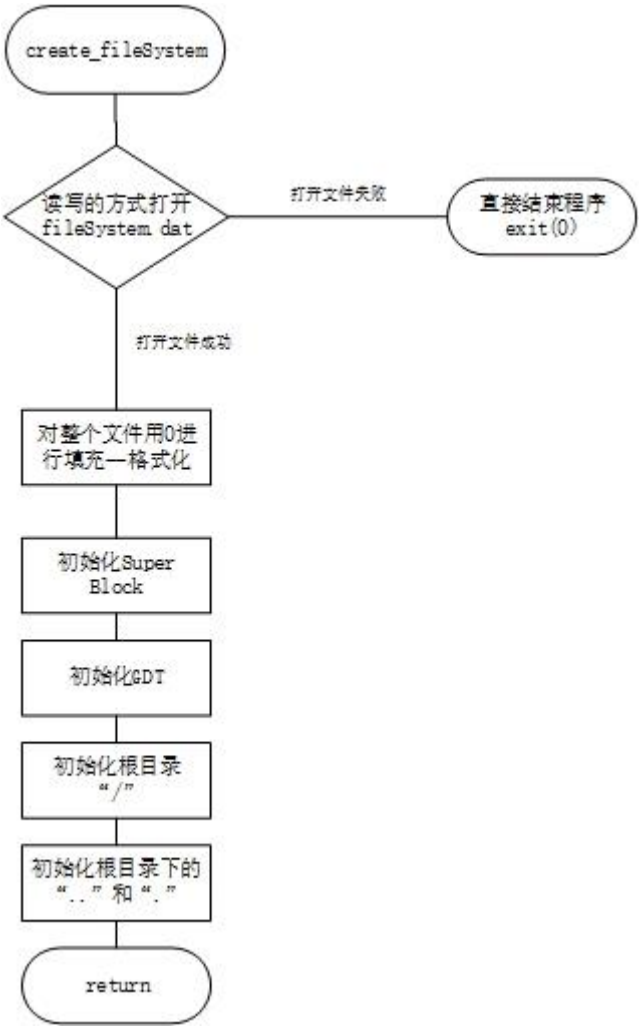
19) current_time();

函数原型：void current_time(void)
函数功能：将存储时间的变量time_now中的时间设置为当前本地时间
函数说明：这个函数需要用time()函数和strftime()函数，所以需要添加头文件<time.h>

```
void current_time(void){
    time_t t = time(0); //获取当前系统的时间
    strftime(time_now, sizeof(time_now),"%Y-%m-%d %H:%M:%S",localtime(&t));
}
```

20) create_fileSystem();

函数原型：void create_fileSystem(void)
函数功能：创建一个ext2文件系统
函数说明：若文件存在，则删除建立。若文件不存在，则直接建立文件系统
Block Btimap，Inode Bitmap，Inode Table，Data Block在文件初始化的时候全部都格式化了，所以不用进行专门初始化。而Super Block和GDT必须写入正确数据




```
void create_fileSystem(void){

    //创建文件系统的存储位置 ，虚拟磁盘
    //inode 从 1 开始，block 从 0 开始
    last_inode_bit=1;
    last_block_bit=0;
    int i = 0;

    printf("Please wait..\n");
    while(i<20){
        printf(".");
        Sleep(100);
        i++;
    }

    i=0;
    //将 fileSystem.dat 作为模拟磁盘
    pf = fopen("fileSystem.dat", "w+b");
```

```
    if(!pf){
        printf("open file filed!"); //打开文件失败
        exit(0);
    }


    //将buffer清空，好对磁盘进行格式化
    for(i=0; i<BLOCK_SIZE; i++){
        block_buffer[i] = 0;
        //printf("%d", i);
    }

    //格式化磁盘
    fseek(pf, 0, SEEK_SET);
    for(i=0; i < BLOCK_COUNT; i++){
        fwrite(block_buffer, BLOCK_SIZE, 1, pf);  //向磁盘写入0进行格式化
    }
    fflush(pf);


    //初始化super_block, 并写入磁盘中的block
    read_super_block();
    strcpy(super_block_buffer->sb_volume_name, VOLUME_NAME);
    super_block_buffer->sb_inodes_count = 8192;
    super_block_buffer->sb_blocks_count = 8708;
    super_block_buffer->sb_free_inodes_count = 8192-1; //inode号从1开始，所
以空闲的少一个
    super_block_buffer->sb_free_blocks_count = 8708;
    super_block_buffer->sb_block_size = BLOCK_SIZE;
    super_block_buffer->sb_inode_size = 64;
    current_time();
    strcpy(super_block_buffer->sb_wtime, time_now);
    write_super_block();


    //初始化block_group_desc, 并写入磁盘
    read_block_group_desc();
    block_group_desc_buffer->bgd_block_bitmap_start = BLOCK_BITMAP_START;
    block_group_desc_buffer->bgd_inode_bitmap_start = INODE_BITMAP_START;
    block_group_desc_buffer->bgd_inode_table_start = INODE_TABLE_START;
    block_group_desc_buffer->bgd_free_inode_count = 1024*8-1;
    block_group_desc_buffer->bgd_free_block_count = 1024*8;
    block_group_desc_buffer->bgd_used_dirs_count = 0;
```

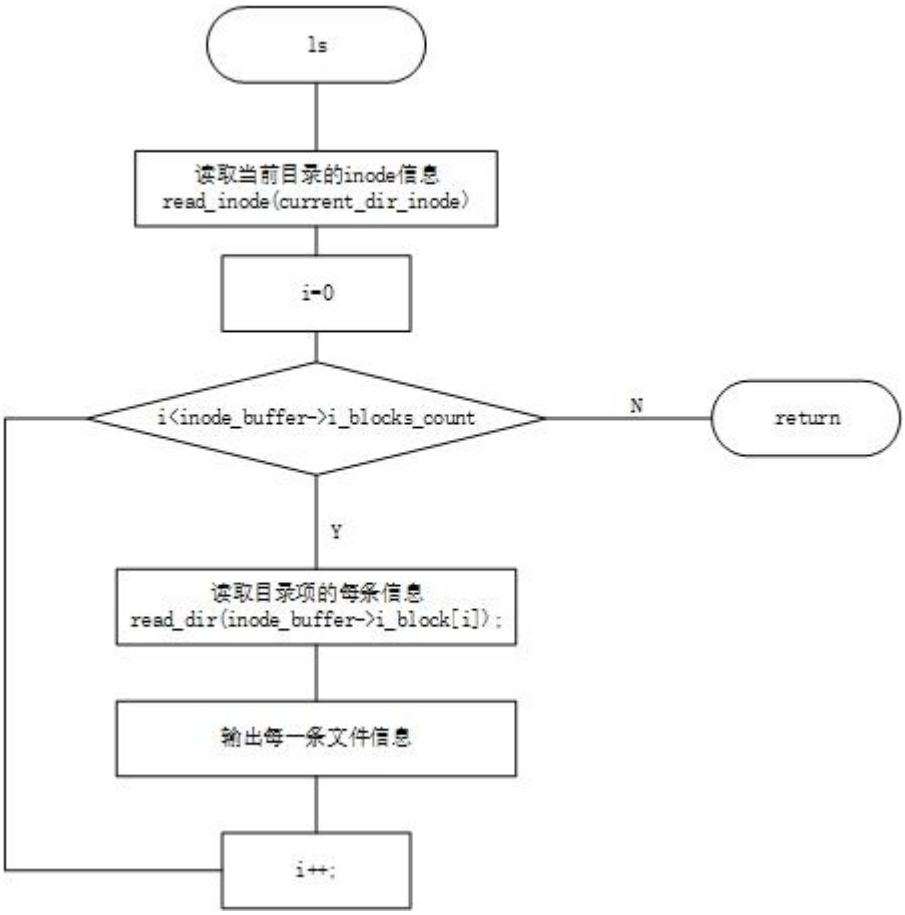
```
    write_block_group_desc();

    //格式时已初始化 block bitmap
    //读入到 block_bitmap_buffer 缓冲区
    read_block_bitmap();

    //格式时已初始化 inode bitmap
    //读入到 inode_bitmap_buffer 缓冲区
    read_inode_bitmap();

    //设置根目录
    read_inode(current_dir_inode);//将第一个 inode 读入 inode_buffer

    inode_buffer->i_mode = 755; //rwx r-x r-x
    inode_buffer->i_size = 16*2; //一个 block 占 1024bytes
    inode_buffer->i_atime = 0;
    current_time();
    strcpy(inode_buffer->i_ctime, time_now);
    inode_buffer->i_mtime = 0;
    inode_buffer->i_dtime = 0;
    inode_buffer->i_blocks_count = 1;
    inode_buffer->i_block[0] = get_free_block(); //分配一个空闲的 block bit

    current_dir_inode = get_free_inode(); //分配一个空闲的 inode bit
    current_dir_length = 1; //根分区的长度“/”


    //printf("%u\n", current_dir_inode);
    write_inode(current_dir_inode);

    read_dir(0); //写到缓冲区
    //初始化子目录“..”和“.”
    dir[0].inode = dir[1].inode = current_dir_inode;
    dir[0].name_len = 1;
    dir[1].name_len = 1;
    dir[0].file_type = dir[1].file_type = 2;
    strcpy(dir[0].name, ".");
    strcpy(dir[1].name, "..");
    for(i=2; i<64; i++){
        dir[i].inode = 0;
    }
    write_dir(inode_buffer->i_block[0]);

    printf("\nthe ext2 file system has been installed!\n\n\n");
```

```
    fclose(pf);
}
```

2. 功能函数

1) ls();

函数原型：void ls(void)
函数功能：显示当前目录下的所有文件及其相关信息
函数说明：current_dir_inode 代表当前目录的 inode 号，读取该 inode 号中的 block 索引号，找到索引号指向的目录项信息，并依次读出来。



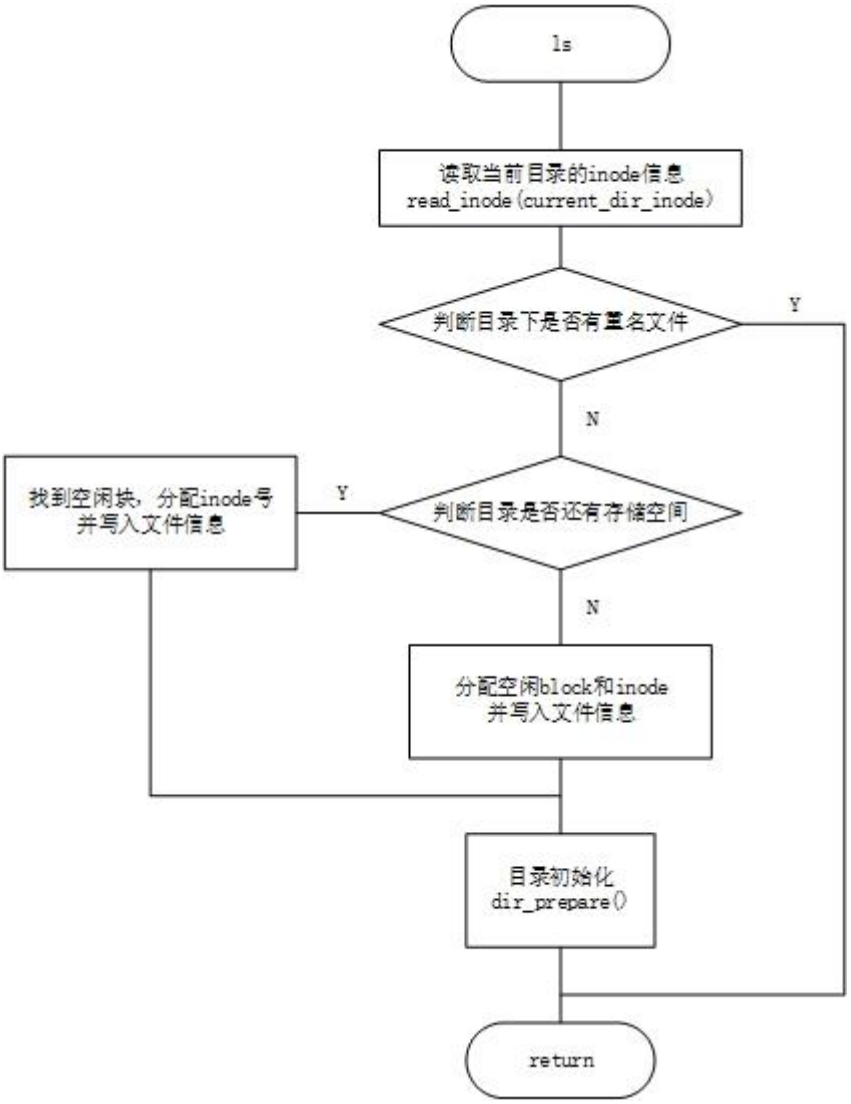

```
void ls(){
    printf("%-15s %-10s %-5s %-13s %-22s %-10s %-10s\n", "name", "type",
"user", "group", "create_time", "mode", "size");
    read_inode(current_dir_inode); //读取当前目录的节点信息

    unsigned int i=0, k=0, temp=0, j=0, n=0;
```

```
    char mode[9]="";

    i=0;
    while(i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode){ //判断inode是否存在
                printf("%-15s", dir[k].name);

                //判断是文件还是目录
                if(dir[k].file_type == 1){
                    printf("%-12s", "<File>");
                }else if(dir[k].file_type == 2){
                    printf("%-12s", "<Dir>");
                }

                printf("%-5s %-10s", "root", "root");

                //输出文件的c_time
                read_inode(dir[k].inode);
                printf("%-25s", inode_buffer->i_ctime);
                temp = inode_buffer->i_mode;
                //printf("%d", temp);
                j=0, n=100;

                //输出文件的权限
                while(1){
                    switch(temp/n){
                        case 0:strcpy(mode+j, "---");break;
                        case 1:strcpy(mode+j, "r--");break;
                        case 2:strcpy(mode+j, "-w-");break;
                        case 3:strcpy(mode+j, "rw-");break;
                        case 4:strcpy(mode+j, "--x");break;
                        case 5:strcpy(mode+j, "r-x");break;
                        case 6:strcpy(mode+j, "-wx");break;
                        case 7:strcpy(mode+j, "rwx");break;
                    }
                    if(n==1){
                        break;
                    }
                    temp %= n;
                    n /= 10;
                    j += 3;
```

```
                }
                printf("%-15s", mode);
                printf("%d\n", inode_buffer->i_size);

            }
            k++;
        }
        i++;
        read_inode(current_dir_inode);
    }
}
```

2) mkdir();

函数原型：void mkdir(char temp[9])
函数功能：新建一个目录
函数说明：参数为一个9个字节以内的文件名，然后判断是否有重名目录，以及是否有空间来新建一个目录，如果满足条件，则新建一个目录，并调用函数dir_prepare()来对目录进行初始化工作。




```
void mkdir(char temp[9]){
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, dir_inode=0, flag=1;

    //判断是否有同名，若有则退出
    while(i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            //如果节点存在并且名字相同
            //这里是不等于，需要注意一下
            if(dir[k].inode && !strcmp(dir[k].name, temp)){
                printf("filename has already existed! \n");
                return;
            }
            k++;
        }
```

```
        i++;
    }
    //k可以用来表示下一个可用的目录点inode， i表示下一个可用的block[i]
//  printf("k: %d\n", k);
//  printf("i: %d\n", i);

    //判断是否还有空闲空间，一个目录最大为8192大小
    if(inode_buffer->i_size == 1024*8){
        printf("Directory has no room to be alloced!\n");
        return;
    }

    flag=1;
    read_inode(current_dir_inode);
    //判断一个目录项中是否还有空闲
    if(inode_buffer->i_size != (inode_buffer->i_blocks_count)*1024){
        i=0;
        //printf("i: %d\n", i);
        //把空闲block找到
        while(flag && i < inode_buffer->i_blocks_count){
            read_dir(inode_buffer->i_block[i]);

            k=0;
            while(k < 64){
                if(dir[k].inode == 0){
                    flag=0; //如果有空闲空间直接跳出两重循环
                    break;
                }
                //printf("\nk: %d\n", k);
                k++;
            }
            i++;
        }

        //将该目录的inode信息补充完整
        dir_inode = get_free_inode();
        dir[k].inode = dir_inode;
        dir[k].name_len = strlen(temp);
        dir[k].file_type=2;
        strcpy(dir[k].name, temp);

        //printf("inode: %d\n", dir_inode);
        write_dir(inode_buffer->i_block[i-1]);
```

```
    }else{
        //一个block块中没有空闲目录项将新分配一个block
        inode_buffer->i_block[inode_buffer->i_blocks_count]=get_free_block();
        inode_buffer->i_blocks_count++;
        read_dir(inode_buffer->i_block[inode_buffer->i_blocks_count-1]);

        //新分配一个inode并写入信息
        dir_inode = get_free_inode();
        dir[0].inode = dir_inode;
        dir[0].name_len = strlen(temp);
        dir[0].file_type=2;
        strcpy(dir[0].name, temp);
        for(k=1; k<64; k++){
            dir[k].inode=0;
        }

        //printf("%d", dir_inode);
        write_dir(inode_buffer->i_block[i-1]);
    }

    inode_buffer->i_size += 16;
    write_inode(current_dir_inode);
    //为每个新加目录添加额外信息
    //目录初始化
    dir_prepare(dir_inode, strlen(temp));
}
```

3) dir_prepare(unsigned int, int);

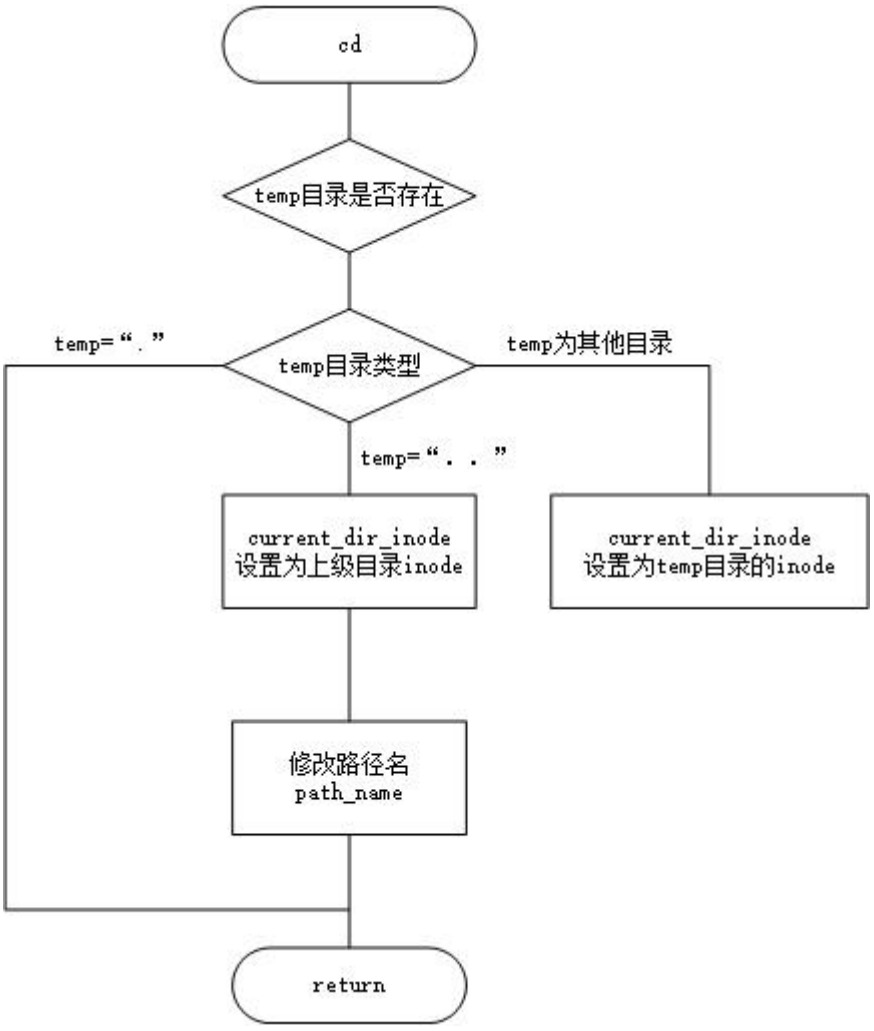
函数原型：dir_prepare(unsigned int dir_inode, int dir_len)
函数功能：对mkdir()建的目录进行初始化，并且在此目录下新建两个特殊目录“..”和“.”，分别代表本目录和上级目录
函数说明：目录权限设置为755


```
void dir_prepare(unsigned int dir_inode, int dir_len){
    read_inode(dir_inode);
    //初始化节点信息
    inode_buffer->i_mode=755;
    inode_buffer->i_size=32;
    inode_buffer->i_blocks_count=1; //占用的 block 的数量
    inode_buffer->i_block[0] = get_free_block();
    current_time();
    strcpy(inode_buffer->i_ctime, time_now);
    //printf("inode_buffer->i_block[0]:%d\n", inode_buffer->i_block[0]);

    //对两个特殊目录信息补充
    read_dir(inode_buffer->i_block[0]);
    dir[0].inode = dir_inode;
    dir[1].inode = current_dir_inode; //当前目录的 inode
    dir[0].name_len = dir_len;
    dir[1].name_len = current_dir_length;
    dir[0].file_type = dir[1].file_type = 2;
    //这是很关键的一步，不然一直会显示上次遗留的信息
    int i=0;
    for(i=2; i<64; i++){
        dir[i].inode=0;
    }
    strcpy(dir[0].name, ".");
    strcpy(dir[1].name, "..");
    write_dir(inode_buffer->i_block[0]);
    write_inode(dir_inode);
```

```
    //更新GDT
    read_block_group_desc();
    block_group_desc_buffer->bgd_used_dirs_count++;
    write_block_group_desc();
}
```

4) cd();

函数原型：void cd(char temp[9])
函数功能：切换目录
函数说明：temp表示传入的目录名，切换到子目录下，有三种目录，“..”，“.”和其他目录，有不同的处理方法，将current_dir_inode改为新的切换目录。如果没有找到此目录则提示未切换成功。




```
void cd(char temp[9]){
    int i=0, k=0;
    if(!strcmp(temp, "."){
        //printf("test");//啥都不做
        return;
```

```
    }else if(!strcmp(temp, ".")){
        read_inode(current_dir_inode); //读取当前目录的节点信息
        //切换到上级目录
        while(i < inode_buffer->i_blocks_count){
            read_dir(inode_buffer->i_block[i]); //读取每一个block指针所对应的
dir
            k=0;
            while(k < 64){
                if(!strcmp(dir[k].name, "..")){

                    //处理显示的路径名
                    current_dir_inode = dir[k].inode;
                    path_name[strlen(path_name)-dir[k-1].name_len-1] = '\0';
                    current_dir_length = dir[k].name_len;
                    //printf("%d", current_dir_inode);
                    //  / sking/123/123/
                    return;
                }
                k++;

            }
            i++;
        }
    }


    read_inode(current_dir_inode); //读取当前目录的节点信息
    while(i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]); //读取每一个block指针所对应的dir
        k=0;
        while(k < 64){
            if(!strcmp(dir[k].name, temp) && dir[k].inode&&dir[k].file_type==2){
                current_dir_inode = dir[k].inode;
                current_dir_length = dir[k].name_len;
                //printf("%d", current_dir_inode);

                strcat(path_name, temp);
                strcat(path_name, "/");
                return;
            }
            k++;

        }
        i++;
```

```
    }
    printf("Can't find this Directory! \n");
}
```

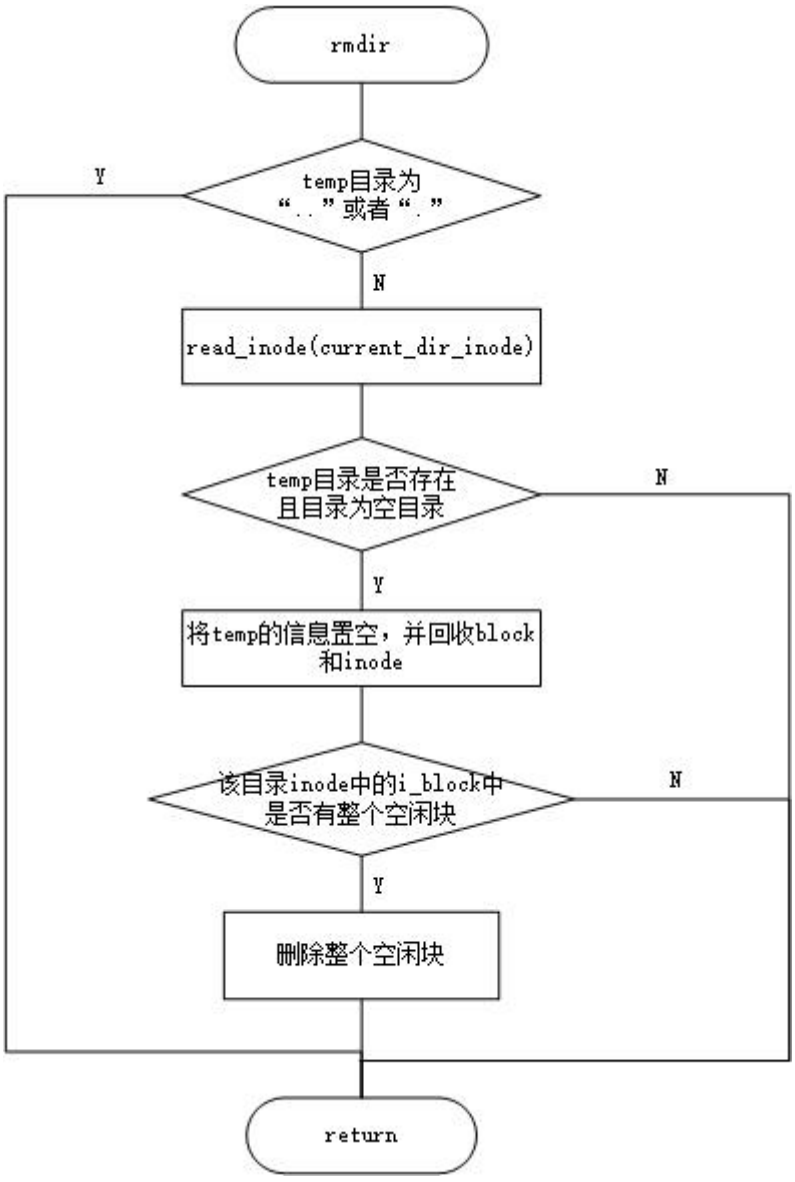
5) format();

函数原型：void format()
函数功能：将磁盘删除，并调用create_fileSystem()重新格式化文件系统

```
void format(){
    char answer = 'Y';
    while(1){
        scanf("%c", &answer);
        if(toupper(answer) == 'Y'){
            printf("Ready to format......\n");

            //格式化文件系统并初始化一些重要信息
            create_fileSystem();
            pf = fopen("fileSystem.dat", "r+b");
            last_inode_bit=1;
            last_block_bit=0;
            read_super_block();
            read_block_group_desc();

            //目录名
            strcpy(current_path, "[root@sking ");
            strcpy(path_name, "/");
            strcpy(path_last, "]# ");

            current_dir_inode=1;
            current_dir_length=1;
            return;
        }else if(toupper(answer) == 'N'){
            return;
        }else{
            fflush(stdin);
        }
        printf("Are you sure you want to format the disk ?[y / n]: ");
    }
}
```

6) void rmdir(char temp[9]); //删除空目录

函数原型：void rmdir(char temp[9])
函数功能：temp表示需要删除的空目录名，若目录为空，则将此空目录删除掉
函数说明：删除前先判断目录是否为空，也就是只有“..”和“.”目录，然后将此目录的inode信息置空并调用remove_inode和remove_block将inode号和block号回收。因为删除了一个目录block号，所以有可能本目录的i_block存放的索引中有一个很可能里面没有存放任何数据。我们需要循环检查这个索引数组并将这个block号回收。




```
void rmdir(char temp[9]){
    int i=0, k=0, flag=0;

    if(!strcmp(temp, "..") || !strcmp(temp, ".")) {
        printf("This directory is not allowed to be deleted!\n");
        return;
    }
```

```
    read_inode(current_dir_inode);

    while(!flag && i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode && !strcmp(dir[k].name, temp) && dir[k].file_type==2){
                flag = 1; //找到此目录了
                break;
            }
            k++;
        }
        i++;
    }

    if(!flag){
        printf("Please enter the correct directory name!\n");
        return;
    }else{
        //加载需要删除的节点信息
        read_inode(dir[k].inode);
        //判断是否为空目录
        //只有 .. 和 . 目录
        if(inode_buffer->i_size == 32){
            inode_buffer->i_mode = 0;
            inode_buffer->i_size = 0;
            inode_buffer->i_blocks_count = 0;
            inode_buffer->i_size = 0;
            //将子目录下的两个文件删除
            read_dir(inode_buffer->i_block[0]);
            dir[0].inode=0;
            dir[1].inode=0;
            write_dir(inode_buffer->i_block[0]);
            //删除 block 号
            remove_block(inode_buffer->i_block[0]);
            //删除本目录下的这个目录
            read_inode(current_dir_inode);
            read_dir(inode_buffer->i_block[i-1]);
            remove_inode(dir[k].inode)
            dir[k].inode=0;
            write_dir(inode_buffer->i_block[i-1]);

            inode_buffer->i_size -= 16;
            //把哪些整个 block 都是空的这种块去掉
```

```
            // 循环检查是否有没有存放数据的block块索引
            i=1;
            flag=0;
            while(flag<64 && i<inode_buffer->i_blocks_count){
                k=0;
                read_dir(inode_buffer->i_block[i]);
                //dir[k].inode=0说明没有分配inode号
                while(k<64){
                    if(!dir[k].inode){
                        flag++;
                    }
                    k++;
                }

                //flag=64表示找到了空闲索引节点
                if(flag==64){
                    remove_block(inode_buffer->i_block[i]);
                    inode_buffer->i_blocks_count--;
                     //如果回收的是中间的索引节点，就把后面的所有节点往前移动一个
                    while(i<inode_buffer->i_blocks_count){
                        inode_buffer->i_block[i] = inode_buffer->i_block[i+1];
                        i++;
                    }
                }

            }
             write_inode(current_dir_inode);

        }else{
            printf("Can't delete, directory is not null! ");
            return;
        }
    }

}
```

7) logout();

函数原型：void logout(void)
函数功能：退出登录
函数说明：因为只有一个root用户，所以就是退出程序

```
void logout(void){
    char answer = 'Y';
```

```
    while(1){
        scanf("%c", &answer);
        if(toupper(answer) == 'Y'){
            printf("\nbye bye ~~\n");
            exit(0);
        }else if(toupper(answer) == 'N'){
            return;
        }else{
            fflush(stdin);
        }
        printf("Are you sure you want to quit ?[y / n]: ");
    }
}
```

8) help();

函数原型：void help()
函数功能：显示帮助命令信息，以防不知道命令如何使
注：这里代码贴出来格式有点凌乱，我贴图片更直观

```
void help(){
    printf("      ********************************************************************************\n");
    printf("      *                                  command help                                *\n");
    printf("      *                                                                              *\n");
    printf("      * 01.command help  : help                09.format disk       : format           *\n");
    printf("      * 02.create dir    : mkdir + dir_name     10.delete empty dir  : rmdir + dir_name *\n");
    printf("      * 03.list dir      : ls                   11.chang dir         : cd + dir_name    *\n");
    printf("      * 04.edit file     : vi + file_name       12.read file         : cat + file_name  *\n");
    printf("      * 05.remove file   : rm + file_name       13.ping IP/Address   : ping + ip/host   *\n");
    printf("      * 06.list local IP : ifconfig            14.traceroute        : traceroute + ip/host *\n");
    printf("      * 07.show data now : data                15.modify mode       : chmod + mode     *\n");
    printf("      * 08.logout        : logout              16.display disk info : dumpe2fs         *\n");
    printf("      *                                                                              *\n");
    printf("      ********************************************************************************\n\n");
}
```

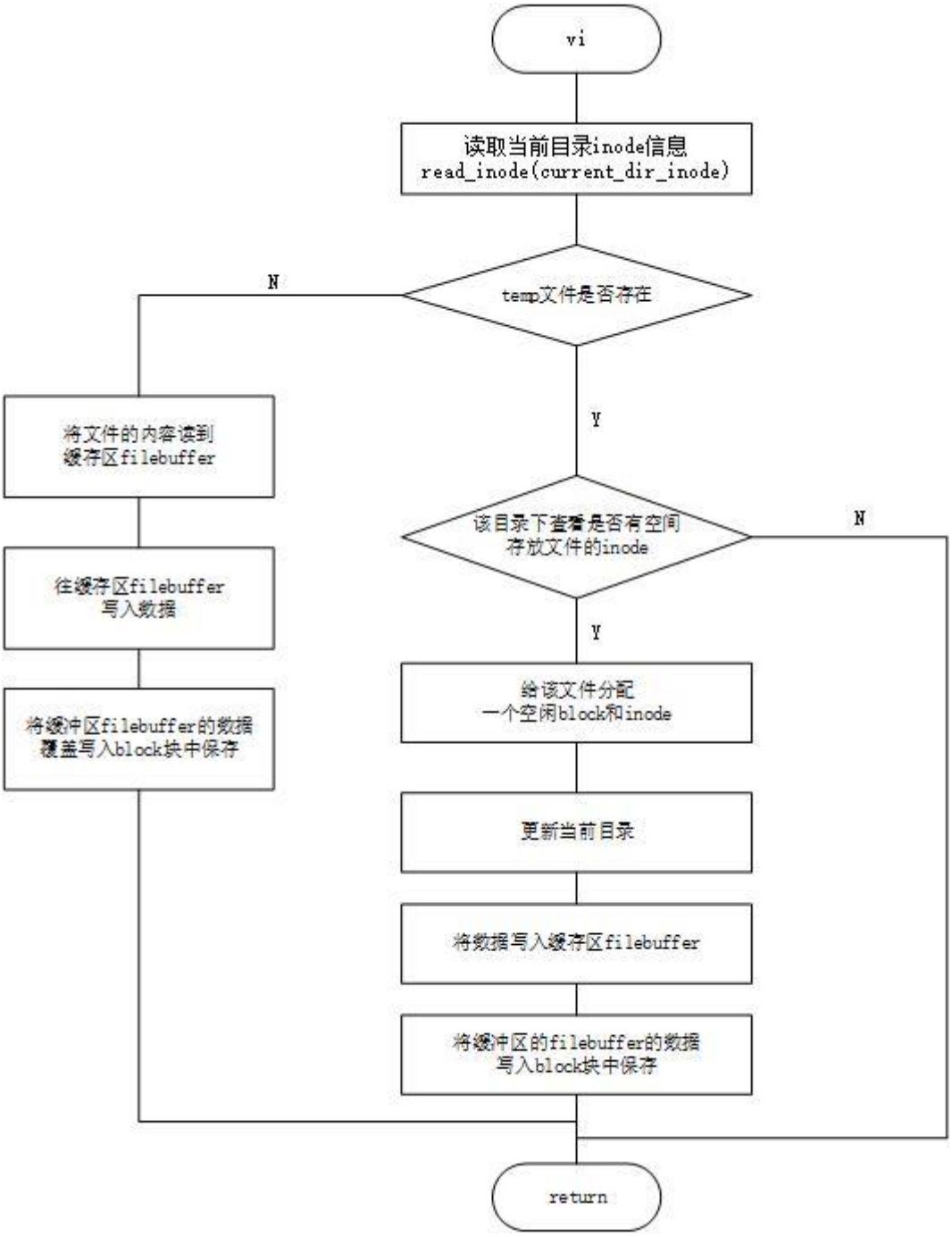
9) dumpe2fs();

函数原型：void dumpe2fs(void)
函数功能：显示磁盘信息，也就是显示 Super Block 里存放的所有信息
注：代码还是贴图片。

```
void dumpe2fs(){
    read_super_block();
    printf("volume name            : %s\n", super_block_buffer->sb_volume_name);
    printf("inodes counts          : %d\n", super_block_buffer->sb_inodes_count);
    printf("blocks counts          : %d\n", super_block_buffer->sb_blocks_count);
    printf("free inodes counts     : %d\n", super_block_buffer->sb_free_inodes_count);
    printf("free blocks counts     : %d\n", super_block_buffer->sb_free_blocks_count);
    printf("inode size             : %d(kb)\n", super_block_buffer->sb_inode_size);
    printf("block size             : %d(kb)\n", super_block_buffer->sb_block_size);
    printf("create time            : %s\n\n", super_block_buffer->sb_wtime);

}
```

10) vi();

函数原型：void vi(char temp[9])
函数功能：temp 表示传入的文件名，此函数可以对文件进行编辑保存或者新建文件保存
函数说明：若文件存在，则是对文件进行编辑，若文件不存在，则是新建文件。但是注意文件的大小不能超过 8K，原因见目录和文件里面有说明。




```
void vi(char temp[9]){
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, flag=1, m=0, file_inode=0;
```

```
    int local=0, file_length=0, file_block_count=0;

    //寻找文件是否存在，如果是目录，则不能进行编辑
    //如果是文件，则存在进行编辑，不存在进行创建
    while(flag && i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode && !strcmp(dir[k].name, temp)){
                if(dir[k].file_type == 2){
                    printf("Directory can't be edit! \n");
                    return;
                }else{
                    flag = 0; //存在文件
                    break;
                }
            }
            k++;
        }
        i++;
    }

    //如果文件不存在, 将创建新文件
    if(flag){
        printf("Can't find the file name, will create a new file!\n");
        printf("Please input '\\q' to quit! \n\n");
        read_inode(current_dir_inode); //读取当前节点信息
        i=0, k=0, file_inode=0, flag=1;

        //是否还有空闲block
        if(inode_buffer->i_size == 1024*8){
            printf("Directory has no room to be alloced!\n");
            return;
        }

        m=1; //用来终止循环
        //判断一个目录项中是否还有空闲
        if(inode_buffer->i_size != (inode_buffer->i_blocks_count)*1024){
            i=0;
            while(m && i < inode_buffer->i_blocks_count){
                read_dir(inode_buffer->i_block[i]);
                k=0;
                while(k < 64){
                    if(dir[k].inode == 0){
```

```
                    m=0; //如果有空间直接跳出两重循环
                    break;
                }
                //printf("\nk: %d\n", k);
                k++;
            }
            i++;
        }

        //printf("k: %d\n", k);
        file_inode = get_free_inode();
        dir[k].inode = file_inode;
        dir[k].name_len = strlen(temp);
        dir[k].file_type=1;
        strcpy(dir[k].name, temp);

        //printf("inode: %d\n", file_inode);
        write_dir(inode_buffer->i_block[i-1]);

    }else{
        //没有空闲则分配一个 block 空间
        inode_buffer->i_block[inode_buffer->i_blocks_count] = get_free_block();
        inode_buffer->i_blocks_count++;

        read_dir(inode_buffer->i_block[inode_buffer->i_blocks_count-1]);
        file_inode = get_free_inode();
        dir[0].inode = file_inode;
        dir[0].name_len = strlen(temp);
        dir[0].file_type=1;
        strcpy(dir[0].name, temp);
        for(k=1; k<64; k++){
            dir[k].inode=0;
        }

        //printf("%d", file_inode);
        write_dir(inode_buffer->i_block[i-1]);
    }

    //写入当前目录
    inode_buffer->i_size += 16;
    write_inode(current_dir_inode);

    //写入文件初始化信息
    read_inode(file_inode);
```

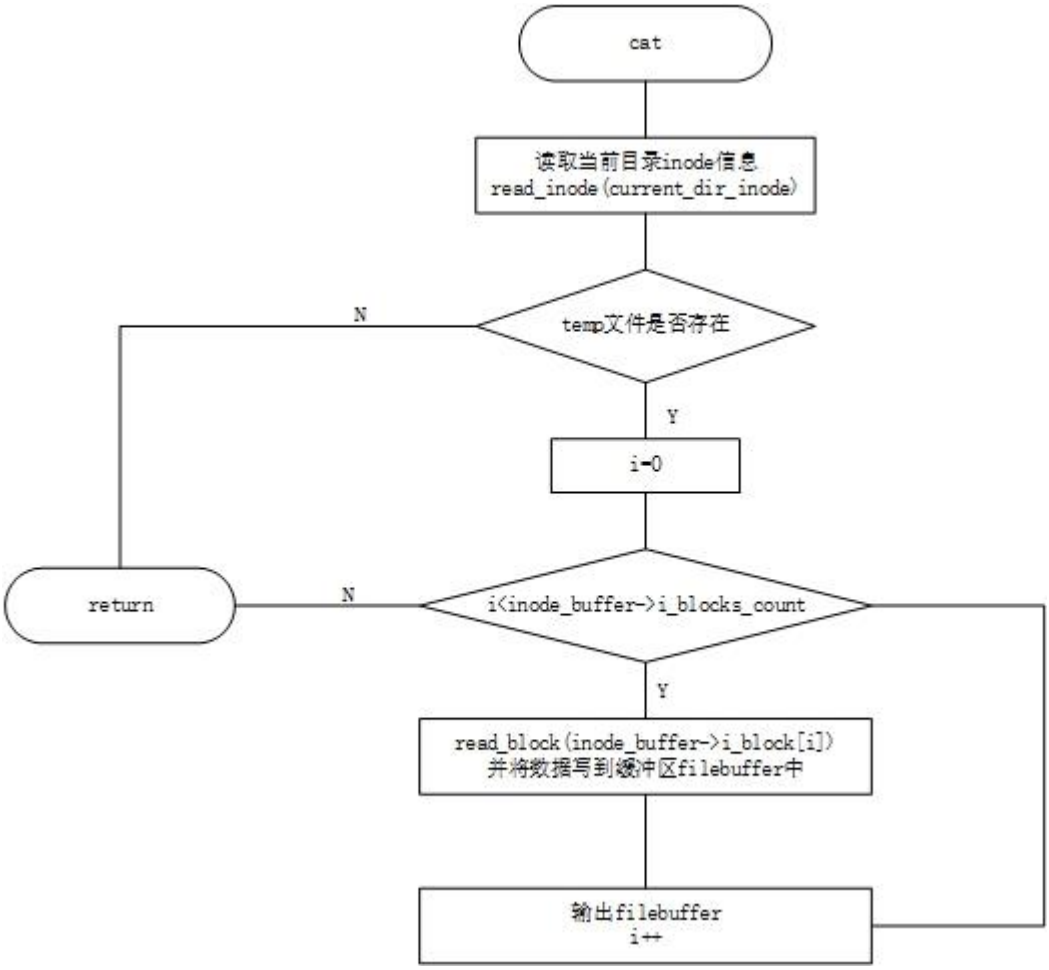
```
        inode_buffer->i_mode = 777;
        inode_buffer->i_size = 0;
        inode_buffer->i_blocks_count = 0;
        current_time();
        strcpy(inode_buffer->i_ctime, time_now);

        //以下才是对文件内容进行编辑
        //开始写入文件缓存 filebuffer;
        while(1){
            filebuffer[local] = getchar();
            if(filebuffer[local] == 'q' && filebuffer[local-1]=='\\'){
                filebuffer[local-1] = '\0';
                break;
            }

            if(local>=8191){
                printf("Sorry,the max size of a file is 8KB!\n");
                break;
            }

            local++;
        }

        file_length = strlen(filebuffer); //文件内容的长度
        file_block_count = file_length/1024;
        if(file_length%1024){
            file_block_count++;
        }
        //直接覆盖写入
        for(i=0; i<file_block_count; i++){
            inode_buffer->i_blocks_count++;
            inode_buffer->i_block[i] = get_free_block();
            read_block(inode_buffer->i_block[i]);

            //直接将内容用函数 memcpy 将文件内容拷贝到内存中
            if(i==file_block_count-1)
                memcpy(block_buffer, filebuffer+i*BLOCK_SIZE, file_length-
i*BLOCK_SIZE);
            else
                memcpy(block_buffer, filebuffer+i*BLOCK_SIZE, BLOCK_SIZE);
            write_block(inode_buffer->i_block[i]);
        }

        inode_buffer->i_size=file_length;
```

```
        write_inode(file_inode);
        printf("\nSave as ");
        for(i=0; i<strlen(temp); i++){
            printf("%c", temp[i]);
        }
        printf("!\n");
    //文件存在时
    }else{
        fflush(stdin);
        printf("The file is exist!!\n");
        printf("Please input '\\q' to quit! \n\n");
        read_inode(dir[k].inode); //读取当前节点信息,存入 inode_buffer

        //现将文件读出来并显示
        for(i=0; i<inode_buffer->i_blocks_count; i++){
            read_block(inode_buffer->i_block[i]); //存入 block_buffer
            if(i == inode_buffer->i_blocks_count-1){
                memcpy(filebuffer+i*BLOCK_SIZE,block_buffer,inode_buffer->i_size-
i*BLOCK_SIZE);
            }else{
                memcpy(filebuffer+i*BLOCK_SIZE, block_buffer, i*BLOCK_SIZE);
            }
            remove_block(inode_buffer->i_block[i]);
        }

        for(i=0; i<inode_buffer->i_size; i++){
            printf("%c", filebuffer[i]);
        }

        inode_buffer->i_blocks_count=0;

        //将读入缓冲区的文件内容末尾指针记录下来
        local = inode_buffer->i_size;
        while(1){
            if(local>=8191){
                printf("Sorry,the max size of a file is 8KB!\n");
                break;
            }

            filebuffer[local] = getchar();
            if(filebuffer[local] == 'q' && filebuffer[local-1]=='\\'){
                filebuffer[local-1] = '\0';
                break;
```

```
            }

            local++;
        }

        file_length = strlen(filebuffer); //文件内容的长度
        file_block_count = file_length/1024;
        if(file_length%1024){
            file_block_count++;
        }

        //直接将原来的内容覆盖写入
        for(i=0; i<file_block_count; i++){
            inode_buffer->i_blocks_count++;
            inode_buffer->i_block[i] = get_free_block();
            read_block(inode_buffer->i_block[i]); //将数据读入缓冲区
            if(i==file_block_count-1)
                memcpy(block_buffer,filebuffer+i*BLOCK_SIZE,file_length-
i*BLOCK_SIZE);
            else
                memcpy(block_buffer, filebuffer+i*BLOCK_SIZE, BLOCK_SIZE);
            write_block(inode_buffer->i_block[i]);
        }

        inode_buffer->i_size=file_length;
        //更新时间
        current_time();
        strcpy(inode_buffer->i_ctime, time_now);
//      printf("file_inode: %d", file_inode);
//      printf("inode_buffer->i_blocks_count: %d\n", inode_buffer->i_blocks_count);
        write_inode(dir[k].inode);

        printf("\nSave!\n");
    }
}
```

11) cat();

函数原型：void cat(char temp[9])
函数功能：temp代表文件名，查看该文件的内容并输出到屏幕上

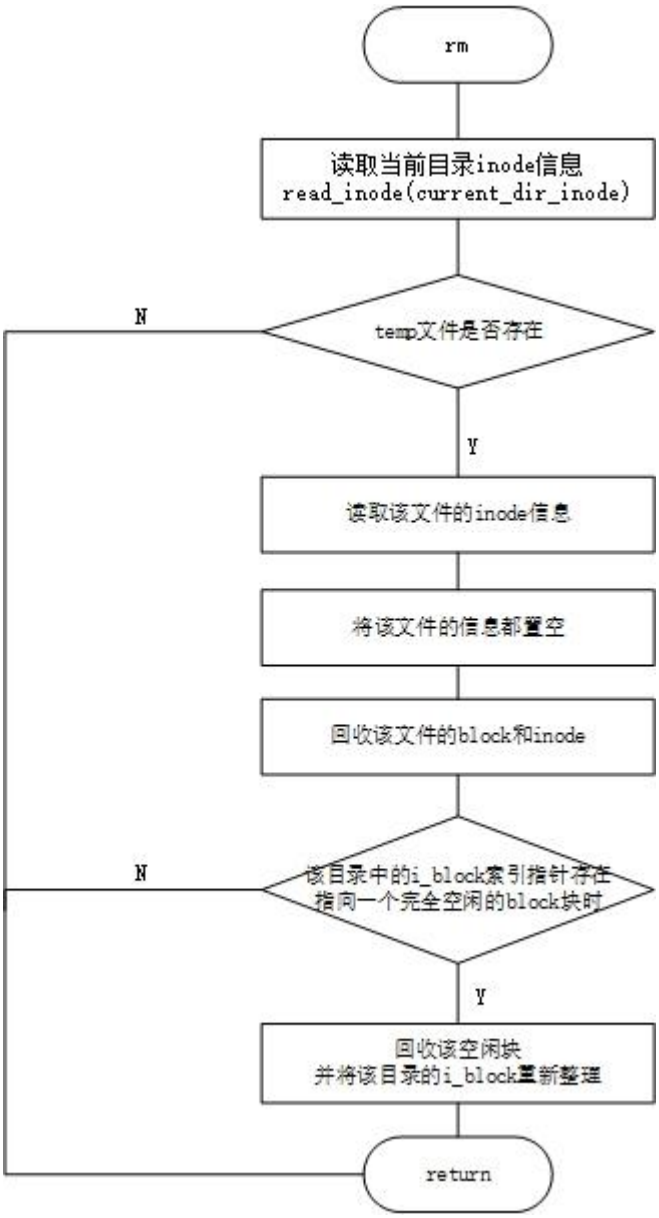



```
void cat(char temp[9]){
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, flag=1;

    //寻找文件是否存在，如果是目录，则不能进行查看
    while(flag && i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode && !strcmp(dir[k].name, temp)){
                if(dir[k].file_type == 2){
                    printf("That is a directory! \n");
                    return;
                }else{
                    flag = 0; //存在文件
                    break;
                }
            }
            k++;
```

```
        }
        i++;
    }
    if(!flag){
        read_inode(dir[k].inode); //读取当前节点信息,存入 inode_buffer
        for(i=0; i<inode_buffer->i_blocks_count; i++){
            read_block(inode_buffer->i_block[i]); //存入 block_buffer
            if(i == inode_buffer->i_blocks_count-1){
                memcpy(filebuffer+i*BLOCK_SIZE, block_buffer,
inode_buffer->i_size-i*BLOCK_SIZE);
            }else{
                memcpy(filebuffer+i*BLOCK_SIZE, block_buffer, i*BLOCK_SIZE);

            }
            remove_block(inode_buffer->i_block[i]);
        }

        for(i=0; i<inode_buffer->i_size; i++){
            printf("%c", filebuffer[i]);
        }
        printf("\n");
    }else{
        printf("Can't find the filename!\n");
        return;
    }

}
```

12) rm();

函数原型：void rm(char temp[9])
函数功能：temp 为文件名，该函数删除本目录下的文件名， 不能删除目录，只能删除文件名。
函数说明：见一下流程图




```
void rm(char temp[9]){
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, flag=1;
    int m=0, n=0;

    //判断文件是否存在，若文件存在才可删除
    while(flag && i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode && !strcmp(dir[k].name, temp)){
                if(dir[k].file_type == 2){
                    printf("'rm' only delete file! \n");
                    return;
```

```
                }else{
                    flag = 0; //存在文件
                    break;
                }
            }
            k++;
        }
        i++;
    }

    //flag = 1 文件不存在
    if(flag){
        printf("Can't find this filename!\n");
        return;
    }else{
        read_dir(inode_buffer->i_block[i]);
        read_inode(dir[k].inode); //加载inode
        //把当前文件的inode信息删除
        for(i=0; i<inode_buffer->i_blocks_count; i++){
            remove_block(inode_buffer->i_block[i]);
        }
        inode_buffer->i_mode = 0;
        inode_buffer->i_size = 0;
        inode_buffer->i_blocks_count=0;

        //删除本目录下此文件的信息
        read_inode(current_dir_inode);
        inode_buffer->i_size -= 16;
        read_dir(inode_buffer->i_block[i]);
        dir[k].inode = 0;
        write_dir(inode_buffer->i_block[i]);

        //删除后如果有整个block块空闲，则回收该空闲块，并将后面的空闲块的数
据往前移动
        m=1;
        while(m < inode_buffer->i_blocks_count){
            read_dir(inode_buffer->i_block[m]);
            flag=n=0;
            //空闲目录项的inode=0
            while(n<64){
                if(!dir[n].inode){
                    flag++;
                }
                n++;
```

```
            }

            //flag=64表示有64个空闲目录项，也就是一个block空闲
            if(flag == 64){
                remove_block(inode_buffer->i_block[m]);
                inode_buffer->i_blocks_count--;
                while(m < inode_buffer->i_blocks_count){
                    inode_buffer->i_block[m] = inode_buffer->i_block[m+1];
                    m++;
                }
            }
            m++;

        }
        write_inode(current_dir_inode);
        printf("The file has been deleted!\n");

    }
}
```

13) ifconfig();

函数原型：void ifconfig(void)
函数功能：显示当前电脑上所有的ip地址以及对应的hostname
函数说明：这里是使用的windows API,并且需要加入头文件<winsock2.h>才能够使用，配置dev c++的相关运行环境方能使用。对windows API的掌握不是特别清楚，所以这段代码我是从网上借鉴并且封装修改了一下。

```
void ifconfig(){
     char host_name[256];
     int WSA_return, i;
     WSADATA WSAData;
     HOSTENT *host_entry;
     WORD wVersionRequested;

     wVersionRequested = MAKEWORD(2, 0);
     WSA_return = WSAStartup(wVersionRequested, &WSAData);

     if (WSA_return == 0){
         gethostname(host_name, sizeof(host_name));
         host_entry = gethostbyname(host_name);
         printf("\t%-15s\t  %s\n", "IP", "hostname");
         for(i=0;host_entry!=NULL&&host_entry->h_addr_list[i]!=NULL;++i){
```

```
            const char *pszAddr =inet_ntoa(*(struct in_addr
*)host_entry->h_addr_list[i]);
            printf("   %-24s%s\n", pszAddr, host_name);
        }
    }
    else{
        printf("Please check network!\n");
    }
    WSACleanup();
}
```

14) ping();

函数原型：void ping(char ip[128])
函数功能：ip可以代表IP地址或者HOST地址(www.baidu.com)，然后可以对改地址ping 4次，显示相关的信息
函数说明：此函数只是建立了一个管道pipe,调用了windows cmd的ping命令，并将cmd的反馈信息给窗口显示

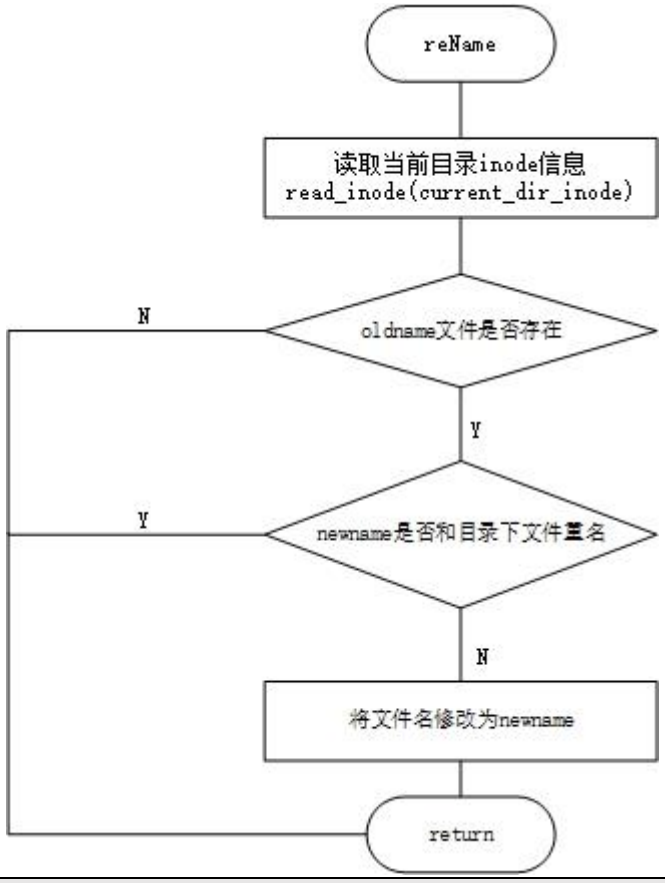
```
void ping(char ip[128]){
    char address[150];
    strcpy(address, "ping -n 4 ");
    strcat(address, ip);
    char buffer[128];

    FILE *pipe = _popen(address, "r");
    //如果管道建立失败
    if(!pipe){
        printf("cmd failed\n");
    }

    while(!feof(pipe)){
        fgets(buffer, 128, pipe);
        printf("%s", buffer);
    }
    _pclose(pipe);
    printf("\n");
}
```

15) reName();

函数原型：void reName(char oldname[128], char newname[128])

函数功能：oldname代表传入的准备修改的文件名，newname代表修改后的文件名，将文件名修改为newname

函数说明：rename和系统函数名冲突，所以改为reName




```
void reName(char oldname[128], char newname[128]){
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, flag=1;
    int m=0, n=0;
    //判断文件是否存在，若文件存在才可修改
    while(flag && i < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i]);
        k=0;
        while(k < 64){
            if(dir[k].inode && !strcmp(dir[k].name, oldname)){
                if(dir[k].file_type == 2){
                    printf("The directory name can't be modified!\n");
                    return;
                }else{
                    flag = 0; //存在
                    break;
                }
            }
            k++;
        }
```

```
        i++;
    }

    //文件不存在 flag=1
    if(flag){
        printf("Can't find the filename!\n");
        return;
    }else{
        flag=1;
        //判断 newname 是否重名
        while(flag && m < inode_buffer->i_blocks_count){
            read_dir(inode_buffer->i_block[m]);
            n=0;
            while(n < 64){
                if(dir[n].inode && !strcmp(dir[n].name, newname)){
                        flag = 0; //存在
                        break;
                }
                n++;
            }
            m++;
        }
        if(!flag){
            printf("The filename \"%s\" is existed\n", newname);
            return;
        }else{
            read_inode(current_dir_inode); //读取当前节点信息
            read_dir(inode_buffer->i_block[i]);
            strcpy(dir[k].name, newname);
            dir[k].name_len=strlen(newname);
            write_dir(inode_buffer->i_block[i]);
        }
    }
}
```

16) data();

函数原型：void data(void)

函数功能：将当前的时间输出显示到目录

```
void data(){
    char data_time[128];
    time_t t = time(0); //获取当前系统的时间
```

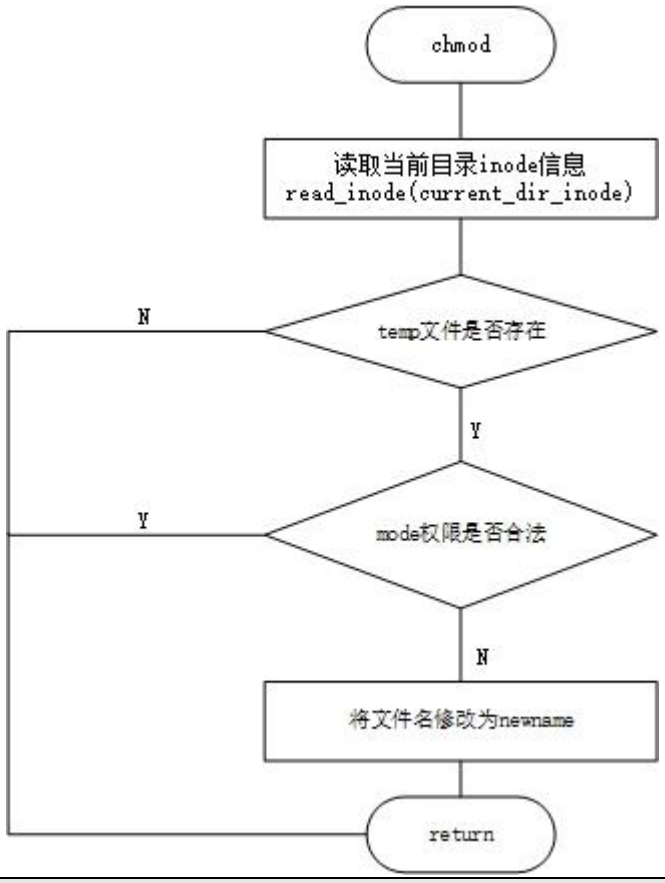
```
    strftime(data_time, sizeof(data_time), "%Y-%m-%d %H:%M:%S %A",
localtime(&t));
    printf("%s\n", data_time);
}
```

17) chmod();

函数原型：void chmod(char temp[9], int mode)

函数功能：temp 代表目录或者文件，mode 代表权限值，此函数改变目录或者文件的权限

函数说明：




```
void chmod(char temp[9], int mode){
    if(mode>777 || mode < 0){
        printf("mode <000-777>!\n");
        return;
    }
    read_inode(current_dir_inode); //读取当前节点信息
    int i=0, k=0, flag=1;

    //判断文件是否存在，若文件存在才可修改
    flag = search_file(temp, &i, &k);
```

```
    //flag = 1 文件不存在
    if(flag){
        printf("Can't find this filename!\n");
        return;
    }else{
        read_dir(inode_buffer->i_block[i]);
        read_inode(dir[k].inode);
        inode_buffer->i_mode = mode;
        current_time();
        strcpy(inode_buffer->i_ctime, time_now);
        write_inode(dir[k].inode);
        printf("The file's mode has been modify!\n");

    }
}
```

18) search_file();

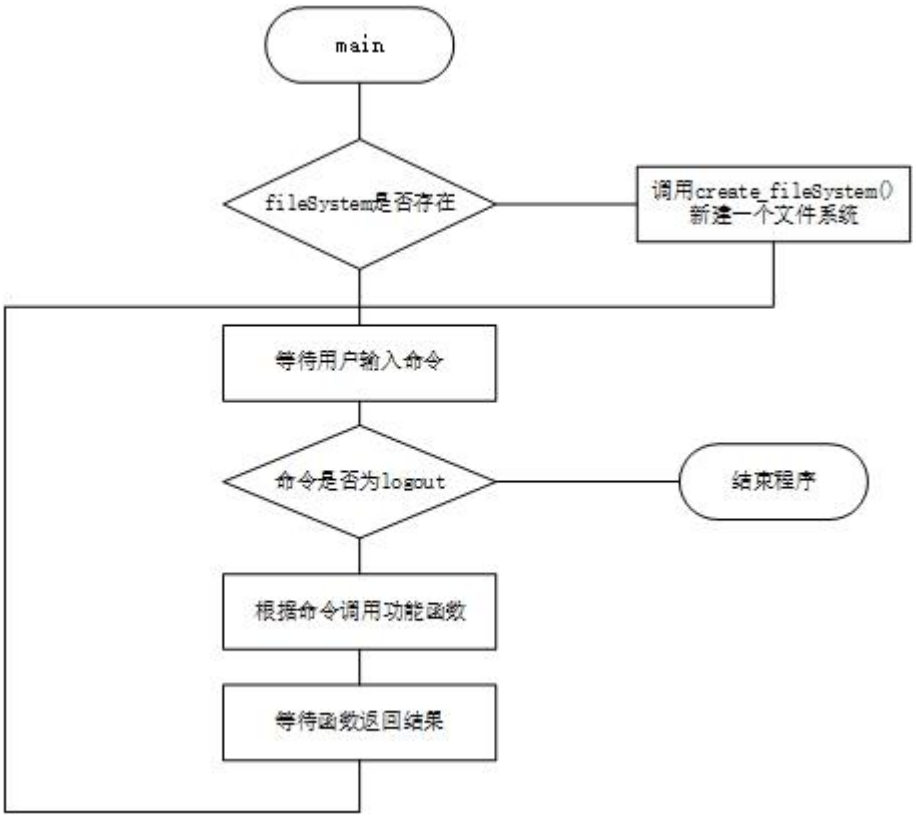
函数原型：search_file(char temp[9], int *i, int int *k)
函数功能：i,k 都是传入的地址，如果在本目录下找到了文件则返回 0，没找到则返回 1
函数说明：

```
int search_file(char temp[9], int *i, int *k){
    read_inode(current_dir_inode); //读取当前节点信息
    int flag=1, i_temp=0, k_temp=0; //设置临时变量

    while(flag && i_temp < inode_buffer->i_blocks_count){
        read_dir(inode_buffer->i_block[i_temp]);
        k_temp=0;
        while(k_temp < 64){
            if(dir[k_temp].inode && !strcmp(dir[k_temp].name, temp)){
                    flag = 0; //存在
                    *i=i_temp;
                    *k=k_temp;
                    return 0;
            }
            k_temp++;
        }
        i_temp++;
    }
    return 1;
}
```

3. 主函数main

程序运行时，调用的第一个函数就是主函数main，先检查是否有文件系统存在，也就是查看文件“fileSystem.dat”是否存在，如果存在，则读出文件系统的相关数据到内存中，如果文件不存在，则将调用create_fileSystem()函数新建一个文件系统。

流程图如下：




```
int main(void){
    printf("Welcome to sking's simple ext2\n");
    printf("********************************\n");
    printf("**  Author: @sking        ****\n");
    printf("**  Create: 2016-6-8      ****\n");
    printf("**  Blog  : www.sking.xin ****\n");
    printf("********************************\n\n");

    pf = fopen("fileSystem.dat", "r+b");
    if(!pf){
        printf("The File system does not exist!, it will be creating
now!\n");
        create_fileSystem();
    }

    pf = fopen("fileSystem.dat", "r+b");
    last_inode_bit=1;
```

```
    last_block_bit=0;
    read_super_block();
    read_block_group_desc();

    //目录名
    strcpy(current_path, "[root@sking ");
    strcpy(path_name, "/");
    strcpy(path_last, "]# ");

    current_dir_inode=1;
    current_dir_length=1;

    int flag = 1;
//  printf("[root@sking /]# ");
    while(1){
        strcpy(current_path, "[root@sking ");
        strcat(current_path, path_name);
        strcat(current_path, path_last);
        char command[10]="", temp[9]="";  //存储命令和变量
        fflush(stdin); //清除 stdin，就不会出现两个提示符了
        printf("%s", current_path);
            //如果是回车，则重显
        if(flag){
            command[0] = getchar();
            if(command[0] == '\n'){
                continue;
            }
        }
        scanf("%s", command+1);
        if(!strcmp(command, "ls")){
            ls();
        }else if(!strcmp(command, "mkdir")){
            scanf("%s", temp);
            mkdir(temp);
        }else if(!strcmp(command, "cd")){
            scanf("%s", temp);
            cd(temp);
        }else if(!strcmp(command, "format")){
            format();
        }else if(!strcmp(command, "rmdir")){
            scanf("%s", temp);
            rmdir(temp);
        }else if(!strcmp(command, "help")){
            help();
```

```
        }else if(!strcmp(command, "logout")){
            logout();
        }else if(!strcmp(command, "dumpe2fs")){
            dumpe2fs();
        }else if(!strcmp(command, "vi")){
            scanf("%s", temp);
            vi(temp);
        }else if(!strcmp(command, "cat")){
            scanf("%s", temp);
            cat(temp);
        }else if(!strcmp(command, "rm")){
            scanf("%s", temp);
            rm(temp);
        }else if(!strcmp(command, "ifconfig")){
            ifconfig();
        }else if(!strcmp(command, "ping")){
            char ip[128];
            scanf("%s", ip);
            ping(ip);
        }else if(!strcmp(command, "rename")){
            char oldname[9];
            char newname[9];
            scanf("%s %s", oldname, newname);
            reName(oldname, newname);
        }else if(!strcmp(command, "data")){
            data();
        }else if(!strcmp(command, "chmod")){
            int mode;
            scanf("%s", temp);
            scanf("%d", &mode);
            chmod(temp, mode);
        }else{
            printf("Can't find this command! \n");
        }
    }
    return 0;
}
```

4. EXT2相关命令对应的功能函数

EXT2命令	功能函数	EXT2命令	功能函数
ls	ls()	dumpe2fs	dumpe2fs()
mkdir	mkdir()	vi	vi()
cd	cd()	cat	cat()
format	format()	rm	rm()
rmdir	rmdir()	ifconfig	ifconfig()
help	help()	ping	ping()
logout	logout()	rename	reName()
data	data()	chmod	chmod()