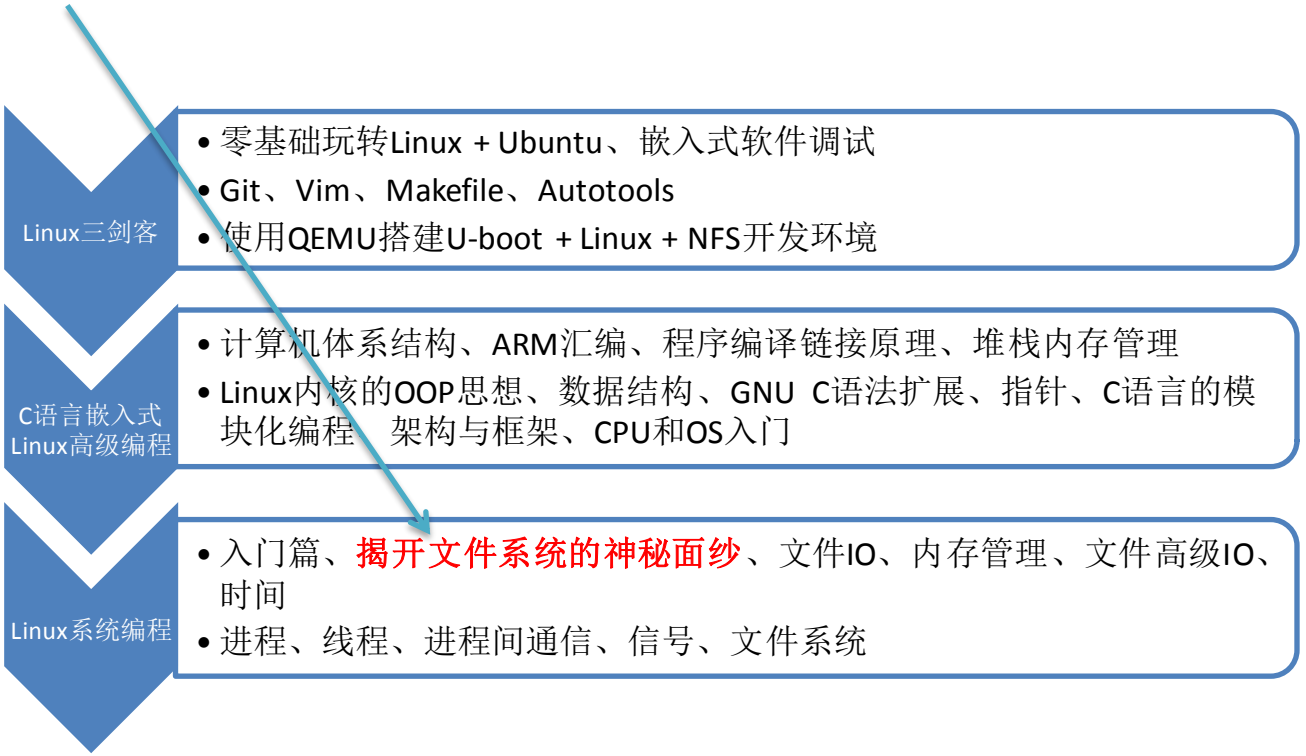
文档版本	说明	作者	创建日期
V0.1	Linux系统编程：入门篇视频配套PPT	王利涛	2018年10月14日
V0.2	第01期：揭开文件系统的神秘面纱	王利涛	2018年11月07日

《嵌入式工程师自修养》视频教程

- 第00步：Linux三剑客
- 第01步：C语言嵌入式Linux高级编程
- 第03步：Linux系统编程
- 第04步：Linux内核编程
- 第05步：嵌入式驱动开发
- 第06步：项目实战
- ----------
- 详情咨询QQ：3284757626
- 视频淘宝店：wanglitao.taobao.com
- 博客：www.zhaixue.cc
- 微信公众号：

Roadmap

- We are here...



Linux系统编程

第01期：揭开文件系统的神秘面纱

文件系统

- 文件系统基本概念
  - 一切皆文件：磁盘文件、设备文件、FIFO
  - 文件系统：对文件进行管理的程序

数据管理

- 数据封装
  - 基本数据类型：int、char、unsigned long
  - 组合数据类型：数组、结构体、联合
  - 缓冲区：指针
  - 文件：文件名、文件权限、时间戳

数据存储抽象

- 文件系统
  - 数据以文件形式封装
  - 以统一的文件接口进行读写

本期课程

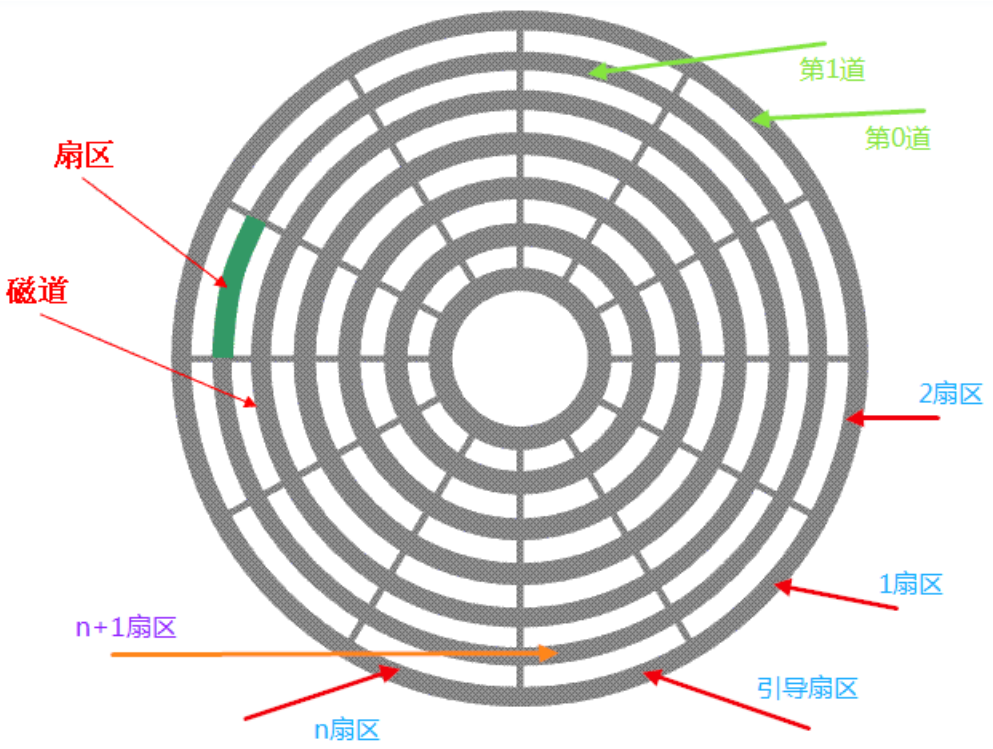
- 主要内容
  - 文件存储：磁盘分区、磁道、扇区、簇
  - NAND Flash存储电路原理、页、块
  - 文件系统：inode、superblock、data block
  - 目录、目录项、文件路径
  - 文件系统的挂载
  - 文件描述符、文件指针
  - 硬链接、软链接、文件删除与恢复
  - 实验：文件系统的制作与挂载

文件在磁盘上的存储(上)

磁盘

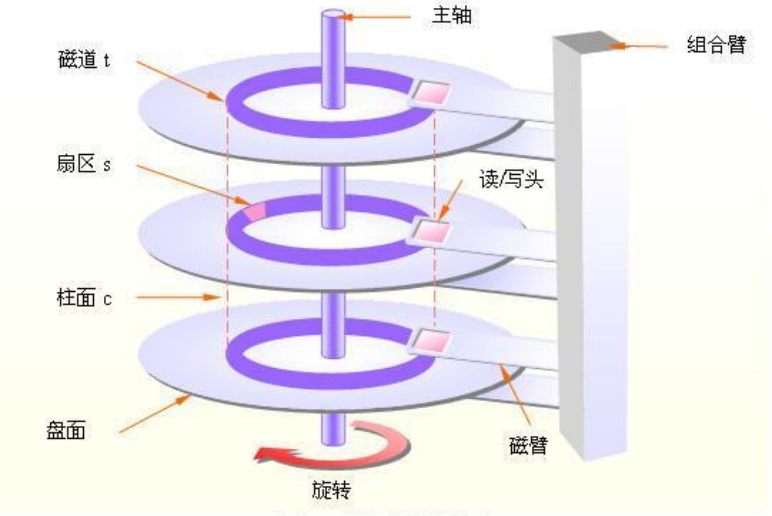
磁盘原理

- 盘面
  - 盘片：Platter
  - 磁头：Head
  - 磁道：Track
  - 扇区：Sector



磁盘原理

- 柱面
  - 柱面：Cylinder
  - 容量：由CHS决定，使用fdisk –l查看
  - 硬盘容量 = 柱面数(磁道数) * 磁头数 * 每个磁道扇区数 * 扇区大小
  - 3D寻址地址：XX磁头，XX磁道，XX扇区(每个磁道有相同扇区数)
  - 线性寻址：以扇区为单位进行线性寻址(等密度扇区)



磁盘控制器驱动

- 磁盘接口
  - SATA
  - IDE
  - SCSI

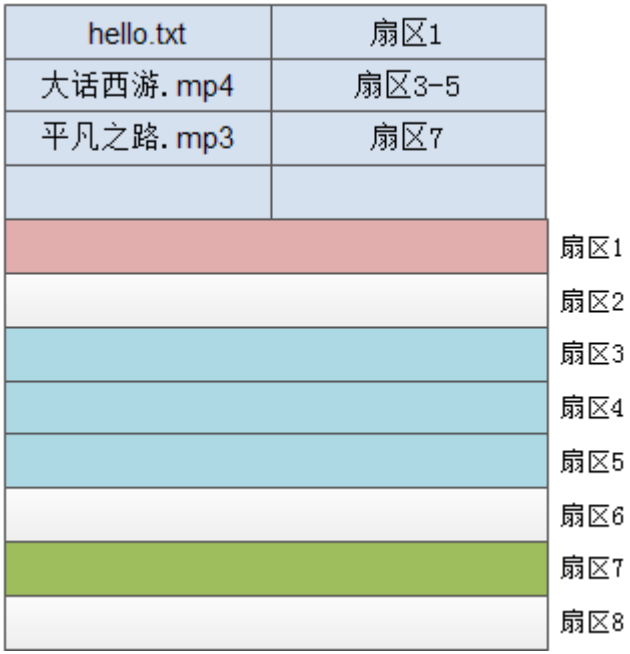
文件在磁盘上的存储(下)

文件数据存储

- 扇区与地址
  - 每个文件存在一块连续的扇区上
  - 一个字节的文件也要占用扇区的整数倍

文件信息存储

- 文件属性
  - 文件在磁盘中的位置
  - 文件创建时间、修改时间
  - 文件大小
  - 文件的使用权限
  - 文件的所有者

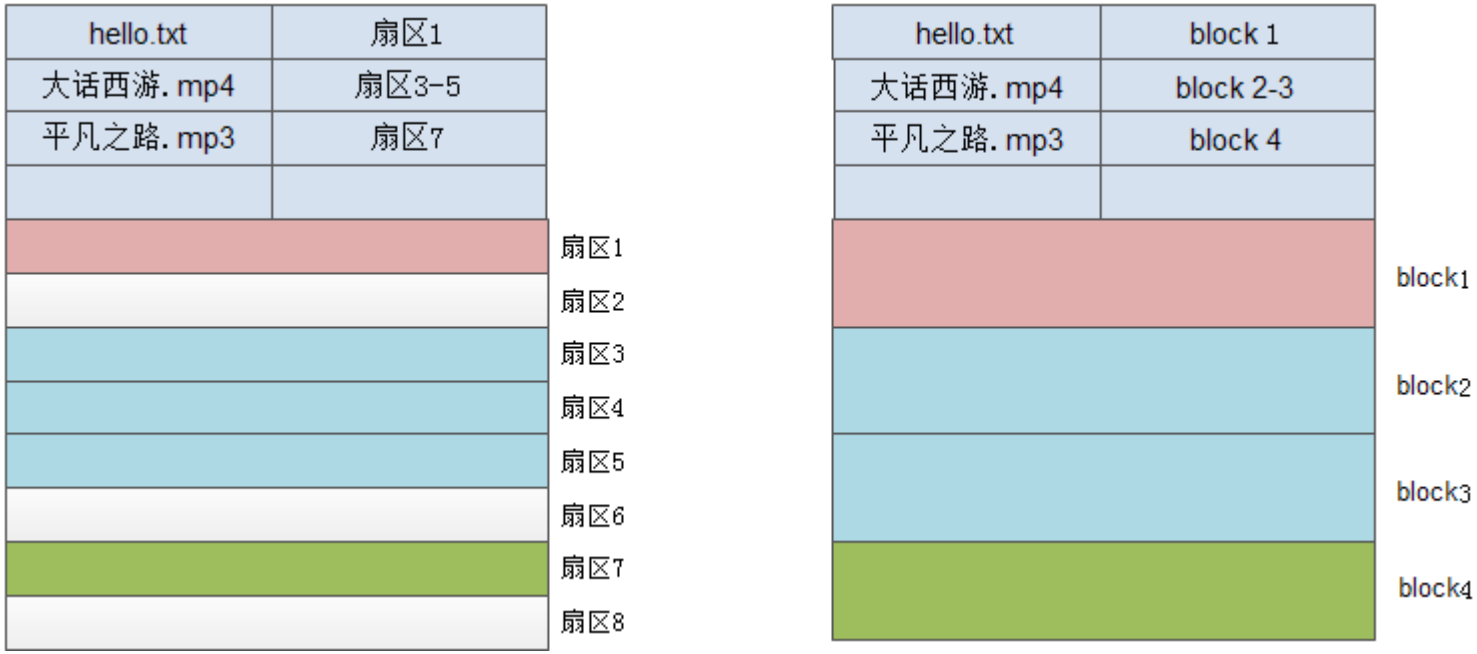


存储抽象

- 文件系统
  - 扇区：磁盘的最小读写物理单元
  - 簇：Windows下文件系统的最小读写逻辑单元
  - 块：Linux下文件系统的最小读写逻辑单元
    - 一个簇/块的大小等于扇区的2、4、8、16...倍
    - 一个文件占用的空间体积是簇/块的整数倍
    - 簇/块越小，磁盘空间利用率越高，但存储效率会降低
    - 文件体积越小，簇/块越大，磁盘空间利用率越低

存储抽象

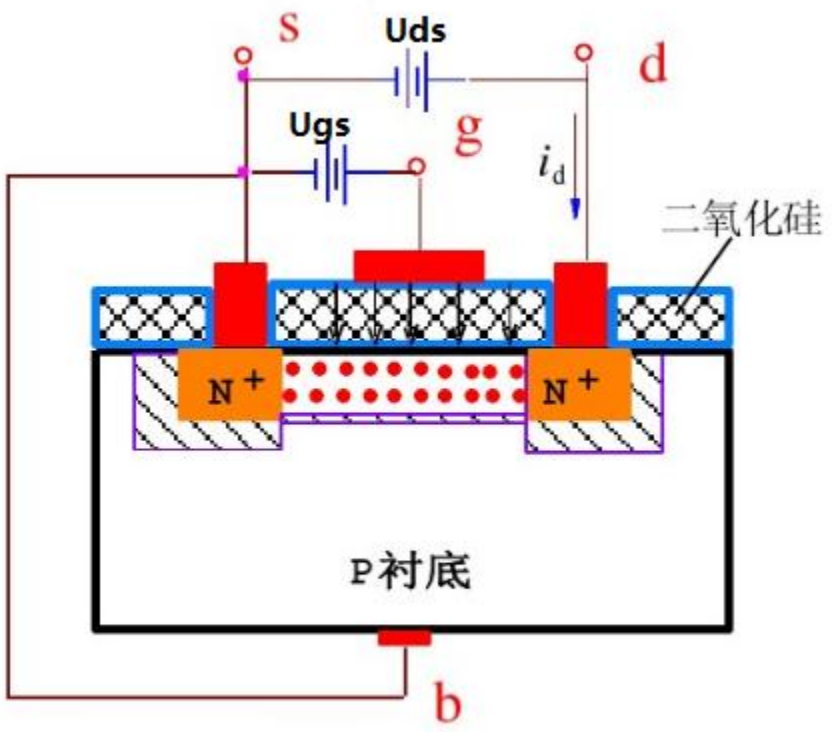
- 文件系统映射层



文件在Flash上的存储(上)

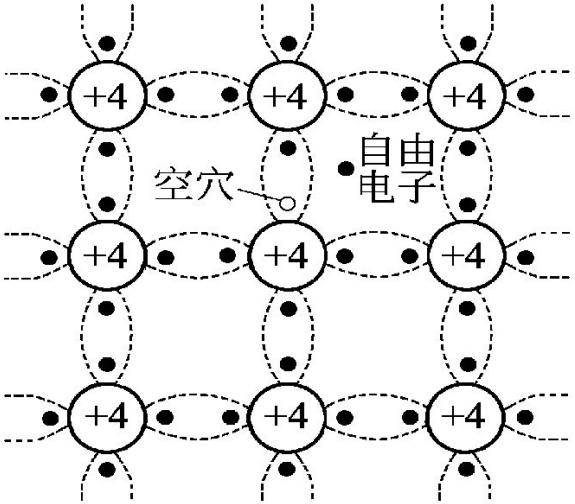
场效应管

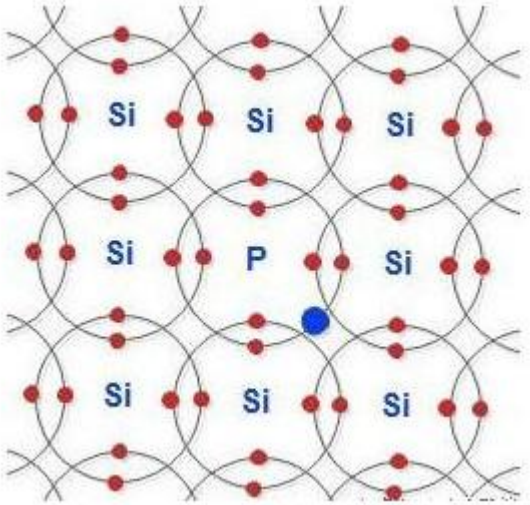
- 工作原理
  - 电压驱动型
  - 单极型晶体管
  - 沟道效应



半导体导电原理

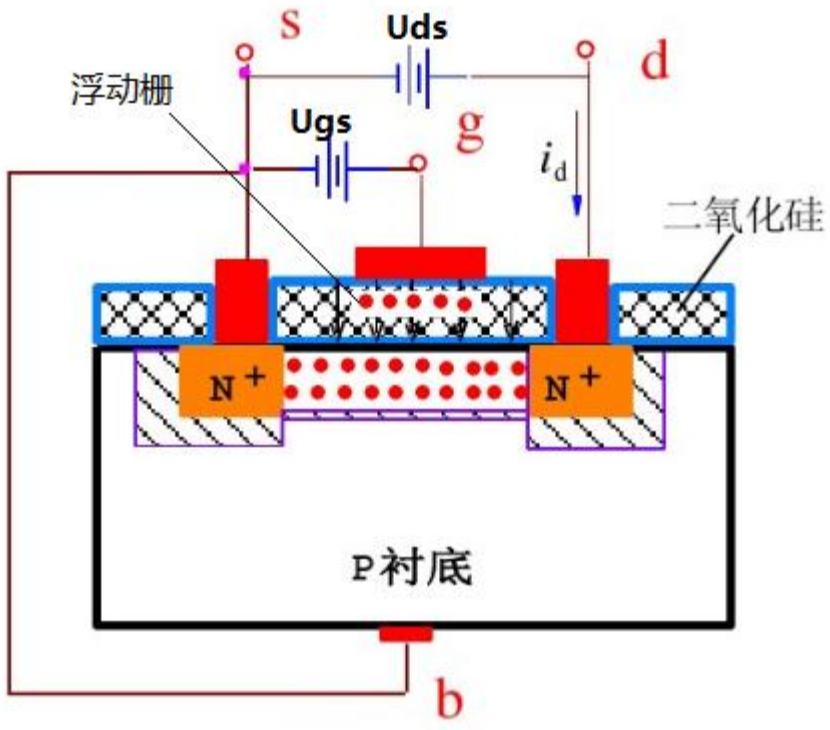
- 掺杂半导体
  - N型半导体：掺杂5价元素，电子浓度>>空穴浓度，导电介质为电子
  - P型半导体：掺杂3价元素，空穴浓度>> 电子浓度，导电介质为空穴





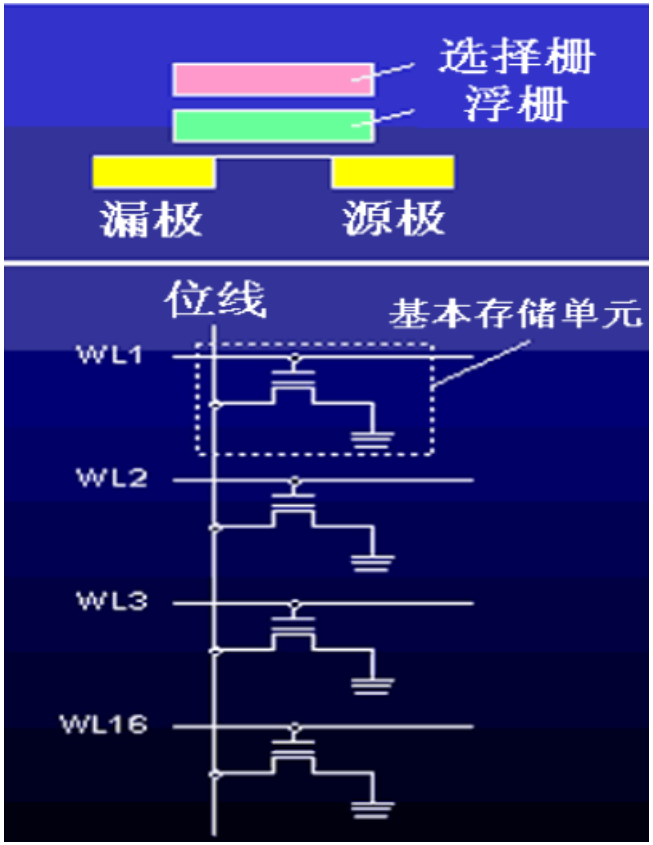
双栅极场效应管

- Flash存储
  - 浮动栅场效应管
    - FG充放电控制DS的通断
    - 成本低、适合存储连续大数据
    - NAND Flash、SSD、SD卡
  - Flash读写操作
    - 写入0：向浮栅中注入电荷
    - 写入1：不注入电荷
    - 擦除：将浮栅中的电荷释放掉
    - 读
      - Vgs=0, 源级接地
      - 漏级为低，读取数据为0
      - 漏级为高，读取数据为1



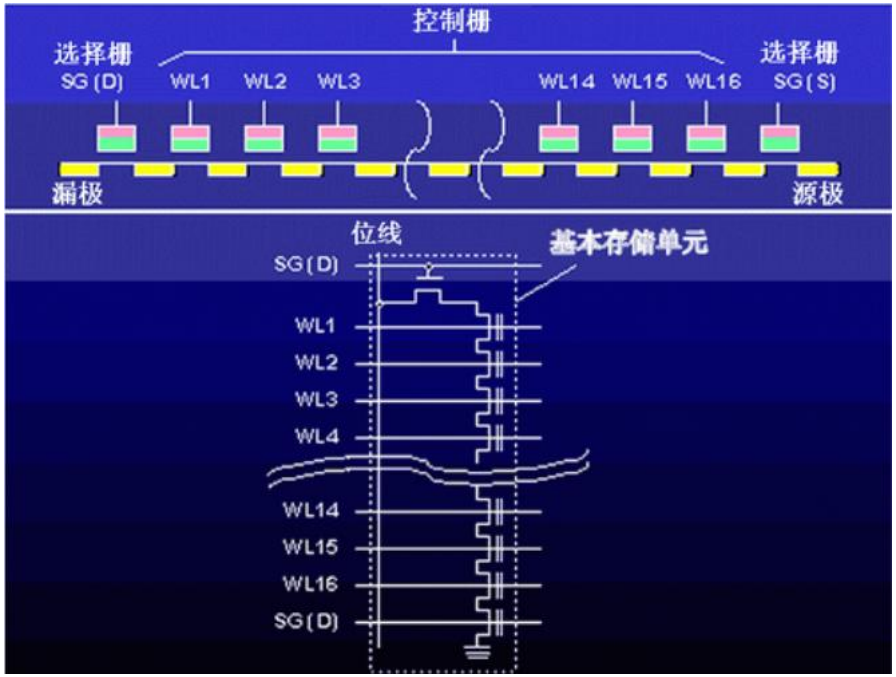
NOR Flash电路

- NOR Flash
  - 并联
    - 每个cell以并联方式连接到位线
    - 每一个cell可以随机存取
    - 专用地址线，直接随机寻址
  - 读：
    - 源级接地、漏级接位线
    - Vgs=0时位线为低，读出的数据为0
    - Vgs=0时位线为高，读出的数据为1



NAND Flash电路

- NAND Flash
  - 串联
    - 8/16/32个cell串联连接到位线
    - 每个cell各引出一个控制栅
  - 读
    - 位线锁定读取的cell
    - 给其余cell加电压导通
    - 锁定的cell不加偏置电压
    - 位线为低，读出的数据为0
    - 位线为高，读出的数据为1
  - 写：
    - 支持整块擦写操作、速度比NOR块



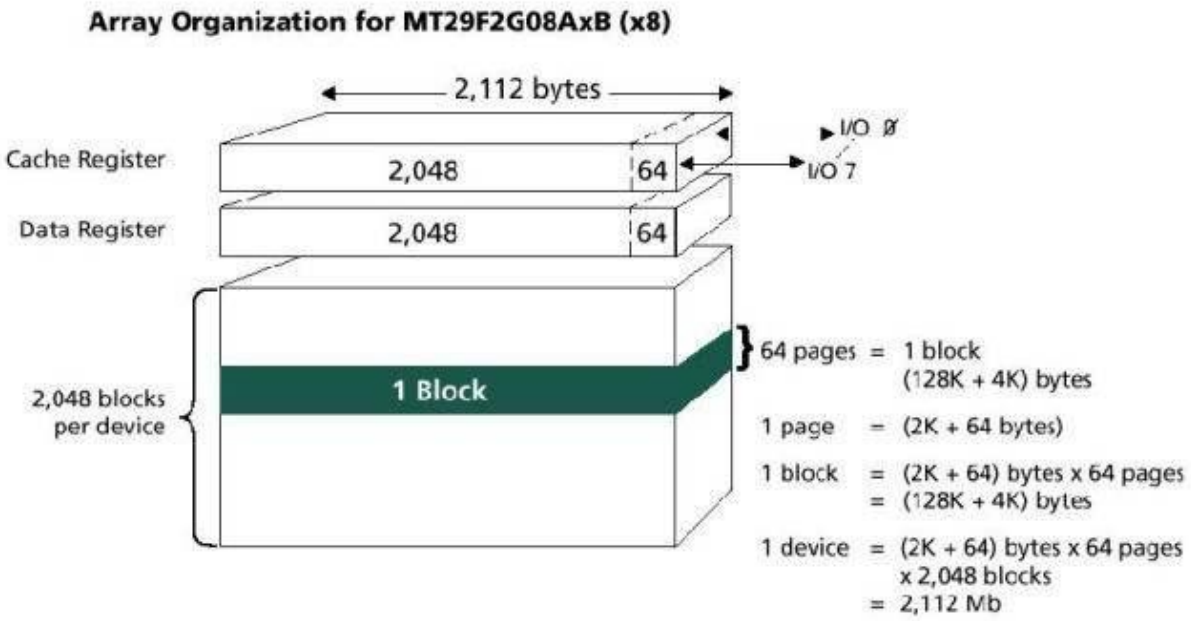
存储方案比较

- 优劣对比
  - NAND Flash
    - 共用位线，减少了芯片内位线数量，节省了存储成本，适合大容量存储
    - 支持整块擦除，写的速度比NOR要高
    - 不支持随机寻址，类似磁盘，要整片地读取
    - 由于串联，相邻之间的cell已发生位反转，导致坏块的产生
  - NOR Flash
    - 每个cell单独的位线，增加了存储成本
    - 支持随机寻址、支持XIP

文件在Flash上的存储(下)

NAND Flash芯片

- 基本构成
  - 每个位线连接8/16/32个基本的存储单元，可以存储1/2/4 Bytes
  - 2048条位线组成的电路构成一个页(page)，容量是2K/4K/8K Bytes
  - 2K个页组成一个块(block)，若干个块构成整个NAND Flash存储芯片

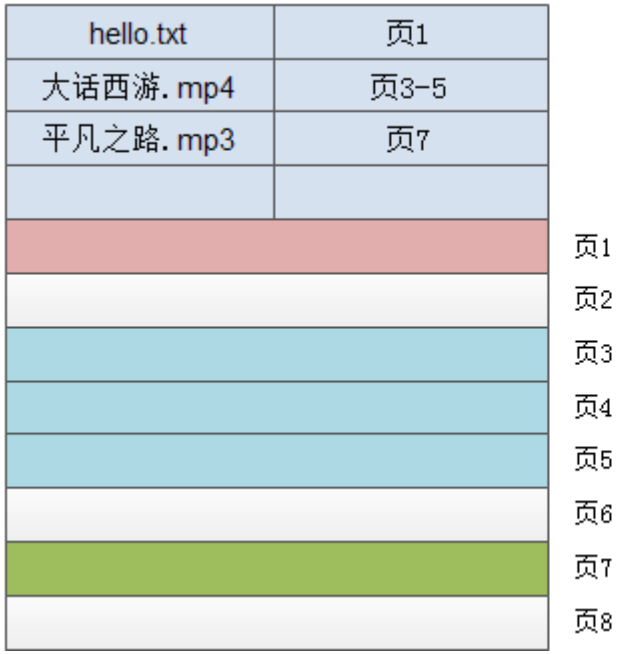


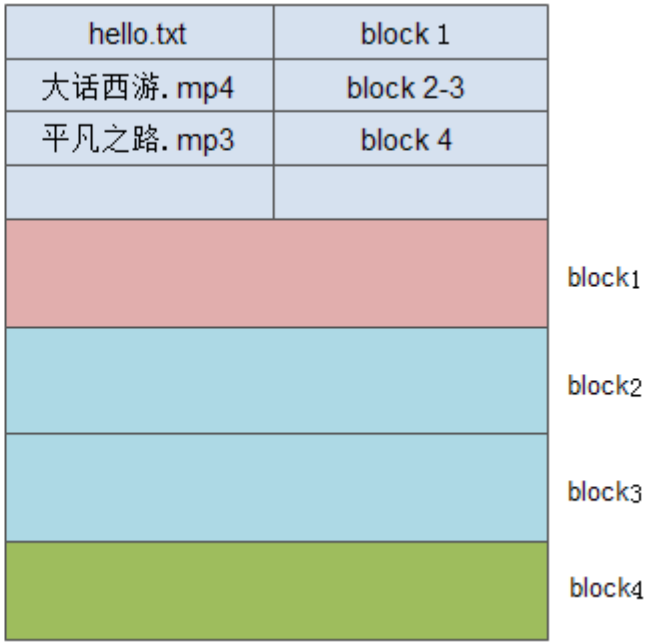
NAND Flash存储地址

- NAND Flash寻址方式
  - 数据线、地址线复用
  - 串行寻址：
    - 以页为最小读写单位，以块为单位擦除
    - 3次地址发送、5个时钟周期
    - 块号+块内页号+页内字节号

文件在Flash上存储

- 存储管理



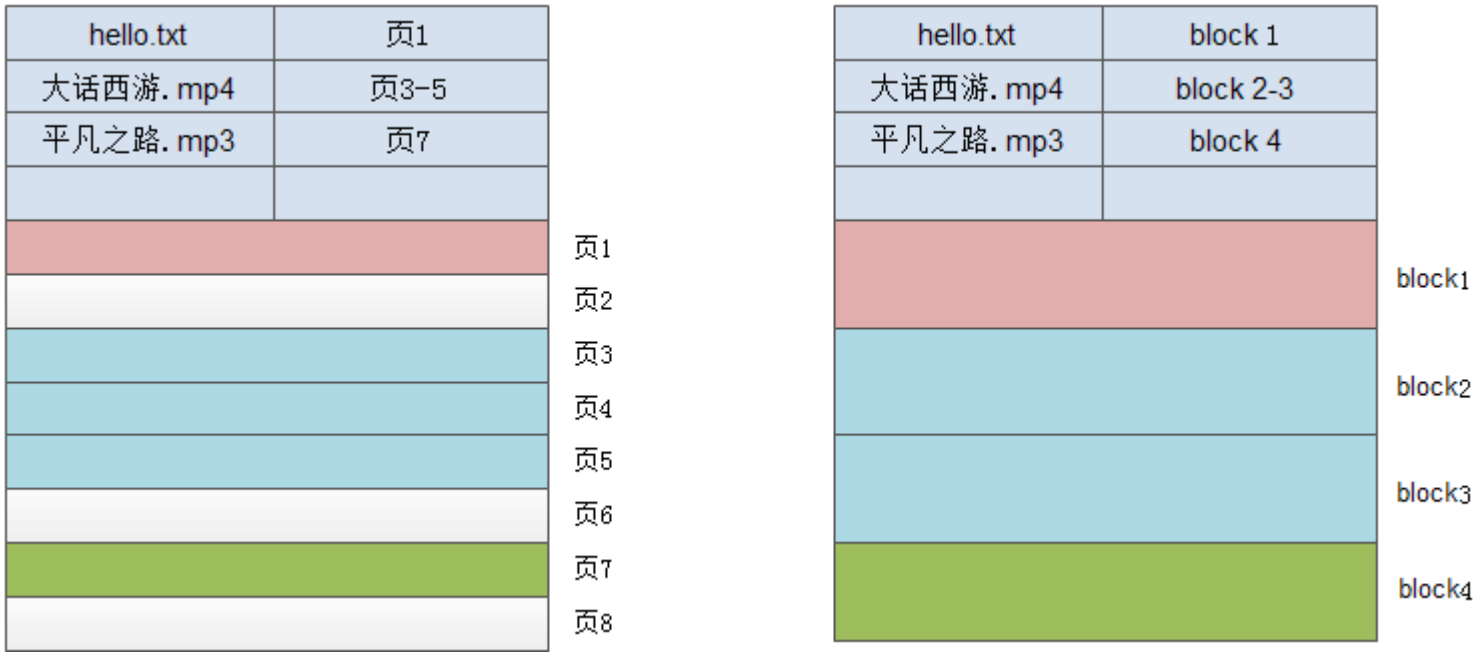


磁盘与Flash存储对比

- 异同点
  - 由控制器驱动进行底层读写
    - 磁盘：SCSI命令、SATA命令
    - NAND Flash：芯片读写指令、周期
  - 寻址方式都是多级地址
    - 磁盘：xx磁道， xx磁头， xx扇区
    - Flash： xx块号， xx块内页号， xx页内字节号
  - 都是整片为单位进行读写
    - 磁盘以扇区为最小读写单位，Nand Flash以页为最小读写单位
    - 都可以被文件系统抽象为逻辑块，通过映射层进行地址映射

文件索引节点：inode

逻辑块


hello.txt
页1
大话西游.mp4
页3-5
平凡之路.mp3
页7
页1
页2
页3
页4
页5
页6
页7
页8
hello.txt
block 1
大话西游.mp4
block 2-3
平凡之路.mp3
block 4
block1
block2
block3
block4

文件的存储

- 两部分
  - 纯数据区
    - 文件真正的数据存储区、基本存储单位为block
  - 元数据区
    - 文件属性：磁盘中的存储位置、文件长度等信息
    - 时间戳：创建时间、修改时间
    - 读写权限：使用read/write系统调用时，要首先要进行权限检查
    - 所属组、所有者
    - 链接数

索引节点:inode

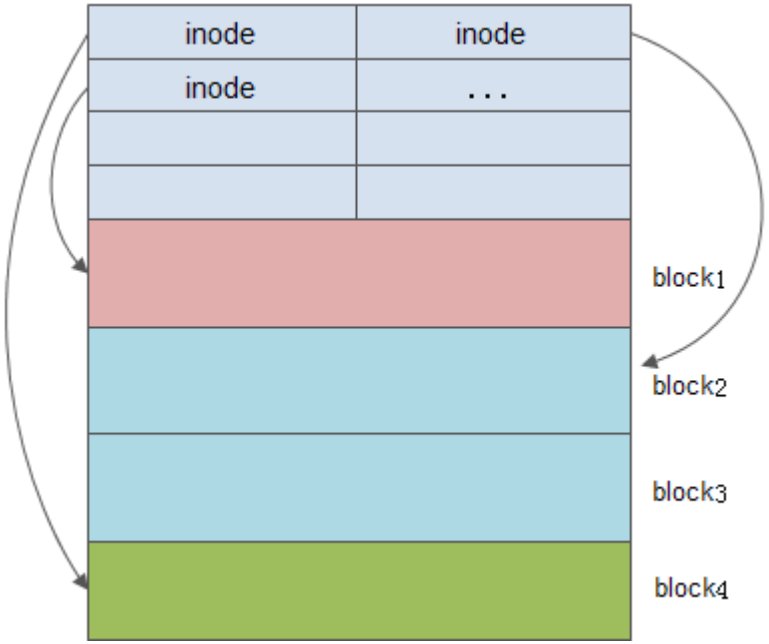
- 用来存储文件信息
  - inode：每个文件使用一个inode结构体来描述
  - 每个inode有固定编号、有单独的存储空间
  - 每个inode的大小为128/256B
  - Linux系统根据inode来查找文件的存储位置
- TIPS
  - 查看文件的inode信息：$stat xx.c
  - 查看某个分区的inode总数：$df -i
  - 查看inode大小：$dumpefs -h /dev/sda1

索引节点：inode

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;
        const  struct           inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;
        unsigned short          i_bytes;
        blkcnt_t                i_blocks;
        struct mutex            i_mutex;
        unsigned long           dirtied_when;
        unsigned long           dirtied_time_when;
        atomic_t                i_count;
        atomic_t                i_dio_count;
        atomic_t                i_writecount;
        const  struct           file_operations   *i_fop;
        struct  address_space   i_data;
        void *i_private;
};
```

索引节点表

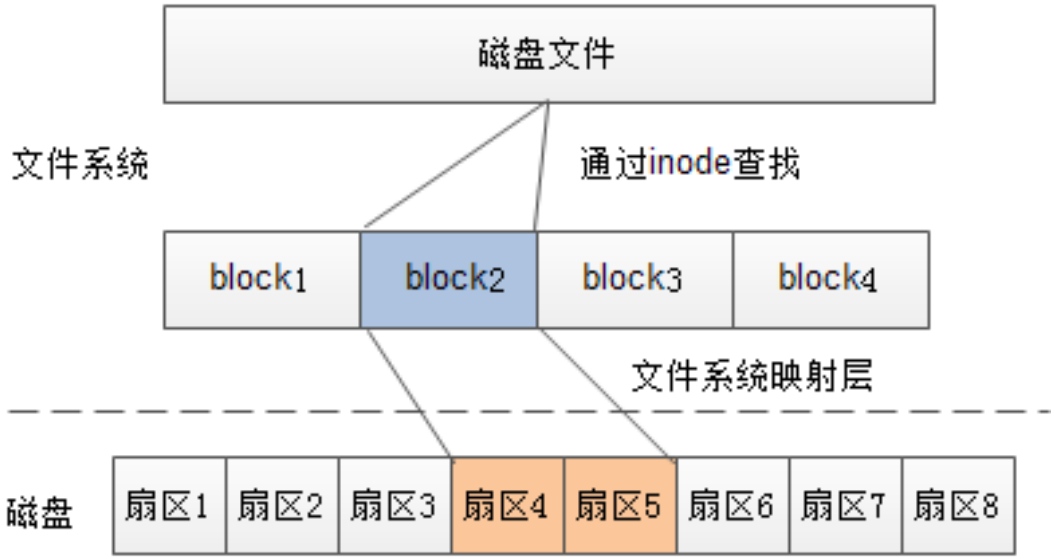
- inode table



data block

- 数据块(逻辑块)
  - 格式化磁盘时划分的文件系统的最小逻辑读写单位
  - 每个block都有自己的编号，inode中存放文件的block地址信息
  - 一个block一般是扇区或页的整数倍：1K/2K/4K
- TIPS
  - 查看某个分区的block信息：$df

存储映射


磁盘文件
文件系统
通过inode查找
block1
block2
block3
block4
文件系统映射层
磁盘
扇区1
扇区2
扇区3
扇区4
扇区5
扇区6
扇区7
扇区8

思考

- 用户可以不断地往文件系统添加文件、删除文件、修改文件，那文件系统是如何维护的呢？

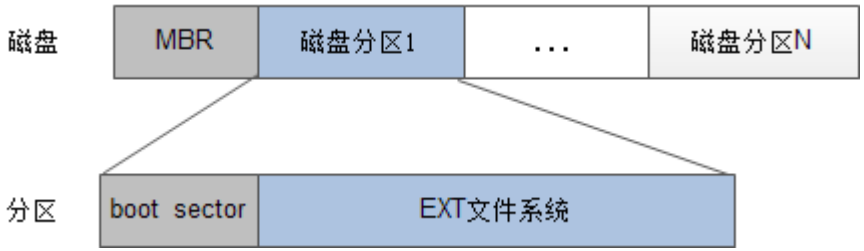
超级块：superblock

超级块

- super block
  - 记录整个文件系统信息
  - 一个inode、block的大小
  - inode使用情况：已使用数量、未使用数量
  - block使用情况：已使用数量、未使用数量
  - 文件系统挂载情况
  - 文件系统的挂载时间、最后一次写入数据、检验磁盘的时间
  - 当文件系统挂载时，这部分信息会加载到内存，并常驻内存

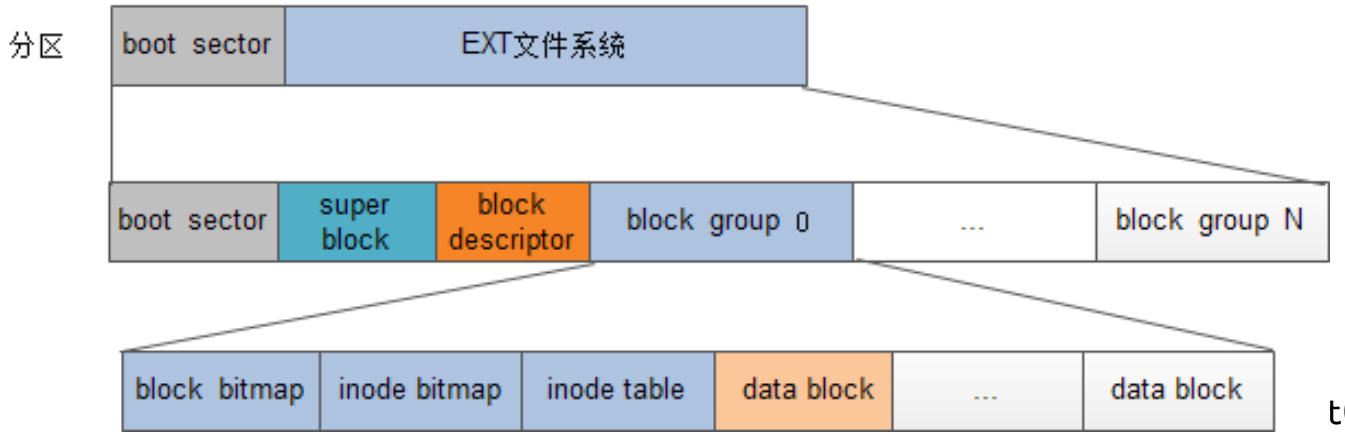
磁盘格式化

- 两种格式化
  - 物理格式化
    - 磁盘在使用前要进行分区和格式化：MBR中存放分区信息、开机代码
    - 出厂前厂家已经做好的工作：划分磁道、扇区
  - 逻辑格式化
    - 使用格式化工具，在磁盘上安装文件系统
    - 将磁盘划分为不同的block
    - 将磁盘划分为不同的区段



磁盘格式化

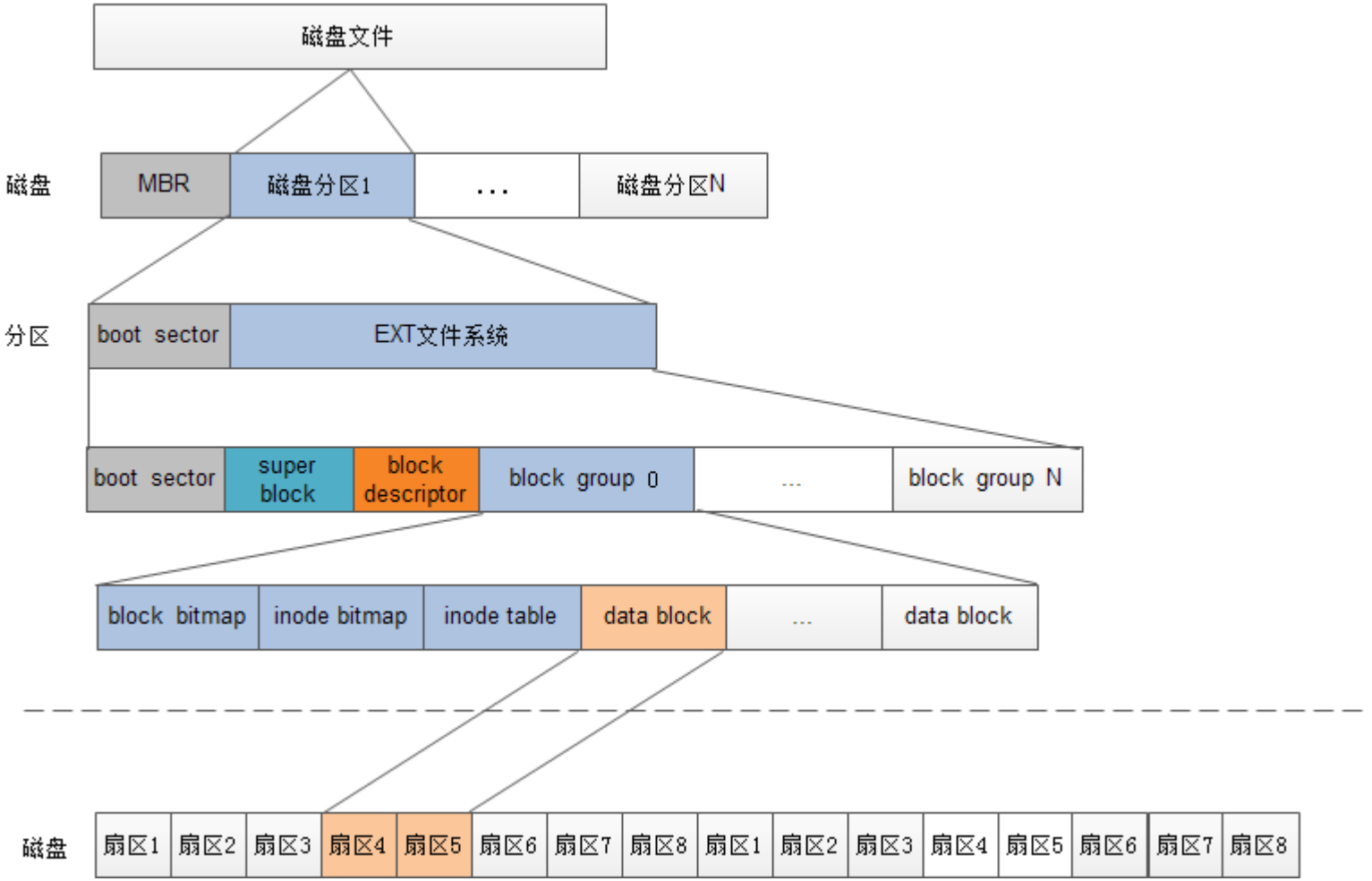
- 不同的区段
  - boot sector：引导扇区
  - Superblock：记录文件系统的整体信息、inode和block信息
  - block group
    - 每个block group都有一个group descriptor，聚集存储在分区开头位置
    - block bitmap：记录block的使用情况，哪些在使用，哪些是空的
    - inode bitmap：记录inode的使用情况
    - inode table、data block



块组

- block group
  - 一个分区在格式化时，可以划分为多个block group
  - 每个block group包含block bitmap、 inode bitmap、 inode table、 data block
  - 每个block bitmap大小为一个block，每bit表示一个block
  - 每个inode bitmap大小为一个block，每一个bit表示一个inode
- group descriptor
  - 存储在superblock的后面
  - 有一个block指针，指向block bitmap
  - 有一个block指针，指向inode bitmap
  - 有一个block指针，指向inode table
  - group descriptor信息存储在superblock中： group descriptor总数等信息

文件在磁盘上的存储


磁盘文件
磁盘
MBR
磁盘分区1
...
磁盘分区N
分区
boot sector
EXT文件系统
boot sector
super block
block descriptor
block group 0
...
block group N
block bitmap
inode bitmap
inode table
data block
...
data block
磁盘
扇区1
扇区2
扇区3
扇区4
扇区5
扇区6
扇区7
扇区8
扇区1
扇区2
扇区3
扇区4
扇区5
扇区6
扇区7
扇区8

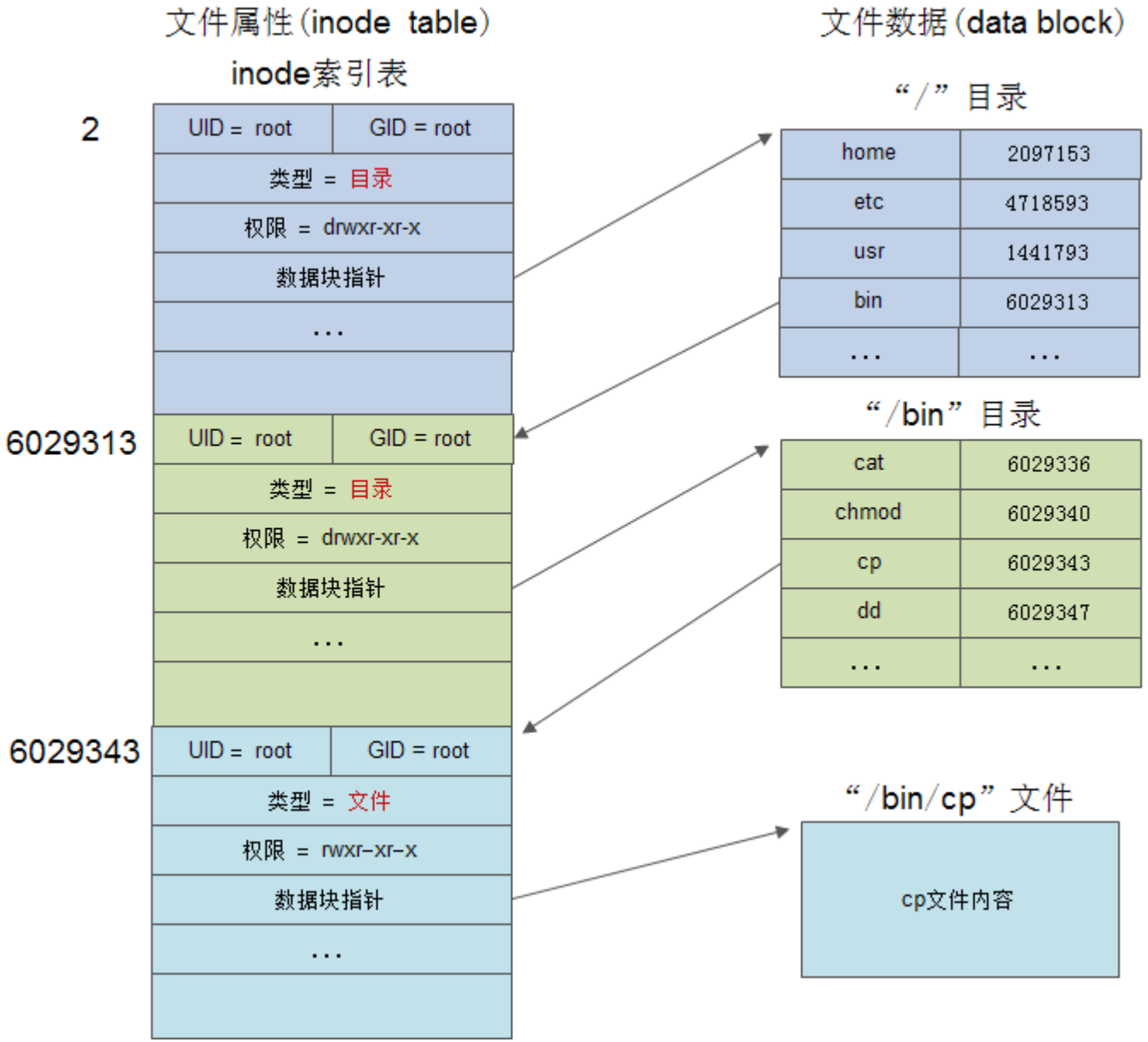
思考

- 通过inode和block，我们已经知道了一个文件和实际物理磁盘之间的关系，而实际开发过程中，用户都是直接通过文件名来操作文件的？文件名是怎么回事？

目录和目录项

目录和目录项

- 目录
  - 是一个文件，有自己的inode，在inode中将该文件类型标记为“目录”
  - 目录存储在data block中
  - 目录本质上是一个表格：由若干个目录项组成
    - 一个目录下面可以多个文件：文件名和文件对应的inode
    - 一个目录文件有多个子目录：目录名和其对应的inode
    - 多个子目录构成树状的文件系统结构
- 目录项
  - 一个目录项由文件名和inode编号组成，根据inode编号可以找到inode table中真正的inode节点
  - 目录项存储在data block中


文件属性(inode table)
inode索引表
文件数据(data block)
"/" 目录
2
UID = root
GID = root
类型 = 目录
权限 = drwxr-xr-x
数据块指针
...
home
2097153
etc
4718593
usr
1441793
bin
6029313
...
...
"/bin" 目录
6029313
UID = root
GID = root
类型 = 目录
权限 = drwxr-xr-x
数据块指针
...
cat
6029336
chmod
6029340
cp
6029343
dd
6029347
...
...
6029343
UID = root
GID = root
类型 = 文件
权限 = rwxr-xr-x
数据块指针
...
"/bin/cp" 文件
cp文件内容

思考

- 根目录“/”的inode编号为2，那么inode编号为1的文件是什么？

文件路径解析

文件路径

- 构成
  - 由各个目录、子目录构成
  - 各个路径构成树状结构的文件系统
- 分类
  - 相对路径
  - 绝对路径

绝对路径

- 根目录
  - 绝对路径的参考起点
    - Linux内核中的“/”
    - Windows系统中的盘符
    - 文件系统预留的inode编号：2

相对路径

- 当前目录
  - 相对路径的参考起点
    - . ：当前目录的硬链接
    - .. ：上级目录的硬链接
    - 根目录下的.和..
    - 查看当前目录的inode编号：$ls –i –d .

小结

- 一个目录下可以包含多个文件、或者嵌套多个子目录
- 各级目录构成一个路径，应用程序根据该路径来找到文件
- 路径分为绝对路径和相对路径
- 路径的本质：各级目录文件中的目录构成的一个inode链

思考

- 想使用一个磁盘、U盘，除了要格式化、安装文件系统外
- 还需要做什么操作？

文件系统的挂载

文件系统的挂载

- 基本原理
  - 一个磁盘格式化、安装文件系统后，可以通过文件接口访问存储空间
  - 用户通过路径名来访问文件
  - 挂载：让磁盘与Linux根文件系统某个目录建立关联、加入全局文件系统树
  - 挂载点(mount point)是进入该挂载设备文件系统的入口

实验

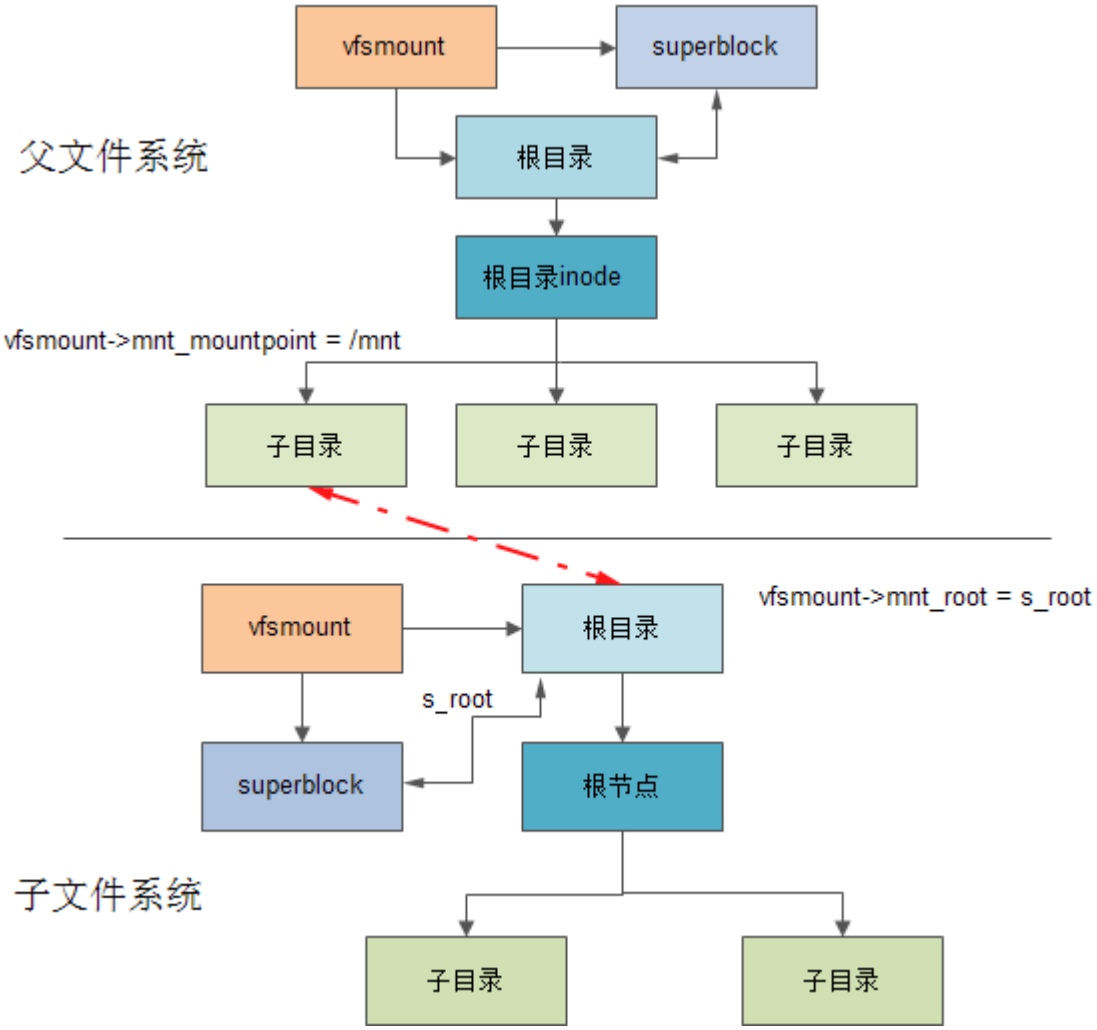
- 目录挂载
- 读写读写

命令: mount --bind 需要挂载的目录 挂载到的目录 自动屏蔽挂载到的目录的文件
umount 被挂载的目录 : 卸载该目录挂载的文件

挂载过程

- 结构体：vfsmount\superblock
  - 每个挂载的文件系统，VFS都会创建一个vfsmount、super_block对象
  - 该对象描述了文件系统mount的所有信息
  - 父文件系统的挂载点：vfsmount->mnt_mountpoint = /mnt
  - 子文件系统的根目录：vfsmount->mnt_root = superblock->s_root
  - 初始化好vfsmount对象后，将该对象添加到VFSMOUNT hash table 中。

文件路径解析


vfsmount
superblock
父文件系统
根目录
根目录inode
vfsmount->mnt_mountpoint = /mnt
子目录
子目录
子目录
vfsmount
根目录
vfsmount->mnt_root = s_root
s_root
superblock
根节点
子文件系统
子目录
子目录

文件路径解析

- 目录项
  - 若目录项dentry标记为DCACHE_MOUNTED，路径解析时对该目录/mnt项屏蔽
  - 计算该目录的HASH值，根据值去VFSMOUNT hash table查找对应的vfsmount对象
  - 根据vfsmount->mnt_root，找到子文件系统的根目录
  - 查找子文件系统指定目录下的文件

文件系统类型

文件系统类型

- 基于操作系统
  - 基于Linux/android
    - ext/ext2/ext3/ext4、XFS、brtfs、iso9660、JFFS/JFFS2
    - minix、proc、NFS、SMB、swap
    - CRAMFS、yaffs、yaffs2、UBIFS
  - 基于Windows
    - FAT16/FAT32
    - NTFS
  - 基于Mac OS X
    - HFS

Linux下的文件系统

- 按存储介质划分
  - 基于磁盘/Flash
    - Ext2/ext3/ext4
    - JFFS2、UBIFS、CRAMFS
    - 基于Flash的文件系统一般基于MTD驱动：坏块管理、磨损均衡
  - 基于内存
    - ramdisk：ramfs、tmpfs
  - 其它特殊文件系统
    - procfs、sysfs
    - NFS： network file system
    - devfs

文件系统选择

- 性能指标
  - 挂载时间
  - IO性能：顺序/随机读写能力、IO等待时间、大/小文件读写能力
  - 资源利用率：存储空间利用率、CPU利用率
  - 功耗

嵌入式文件系统

- 组合文件系统
  - 根据存储数据不同划分不同分区：系统、配置、数据、多媒体数据
  - 内存文件系统：ramdisk、proc
  - 特殊文件系统：devfs、sysfs

问题

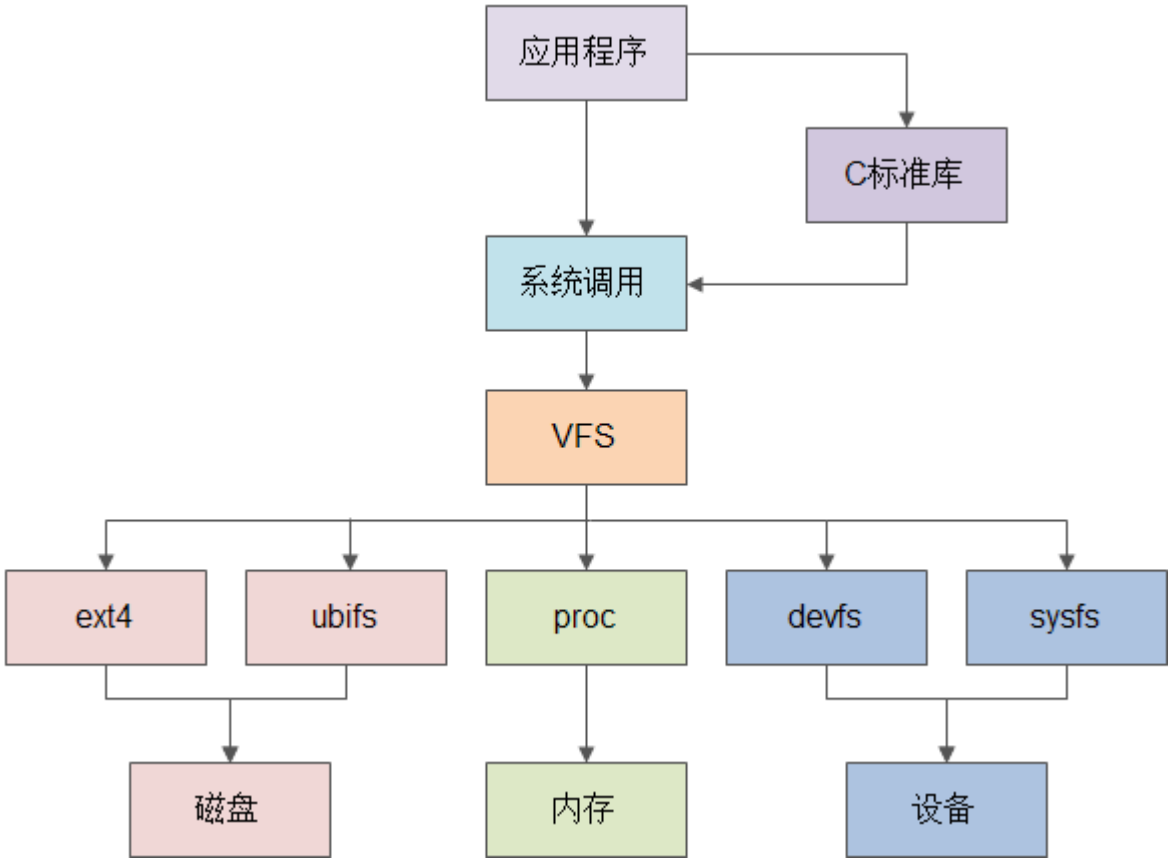
- 不同的文件系统读写接口、读写方式存在一定的差别
- Linux系统如何通过系统调用接口完成统一？

虚拟文件系统：VFS

VFS

- 虚拟文件系统
  - visual file system
  - 基于内核和存储设备之间的抽象层
  - 向上：统一封装了不同设备、文件系统的读写接口：系统调用API
  - 向下：新的设备、文件系统添加到Linux内核，实现VFS的接口即可
  - Linux内核支持60+种不同类型的文件系统

Linux系统中的VFS



VFS的对象类型

- 对象属性
  - super_block：已经安装的具体的文件系统
  - inode：代表一个具体的文件
  - dentry：目录项，目录项路径的组成部分
  - file：进程打开的文件
- 对象方法
  - super_operations：alloc_inode、write_inode、destroy_inode
  - inode_operations：link、mknod、mkdir、rename、create
  - dentry_operations：d_hash、d_compare、d_delete、d_release
  - file_operations：read、write、open、close、fsync、mmap

通过OOP理解VFS

- VFS中的面向对象思想
  - 封装：file->file_operations
  - 继承：write_inode、ext2_write_inode
  - 多态：callback

文件描述符

复习

- 文件的存储
  - 磁盘：扇区、簇、superblock、block、inode、dentry
  - Flash：页、块、block、inode、dentry
  - VFS：统一操作接口
  - 系统调用接口：open、close、read、write、fsync
  - C标准库函数：fopen、fclose、fread、fwrite、fsync

程序中如何操作文件

- 文件描述符
  - Linux进程使用文件描述符(*file descriptors*,简称*fd*)来操作文件
  - int fd = open (“/home/wit/hello.txt”, O_WRONLY, 0666);
  - read (fd, buf, 100);

文件描述符

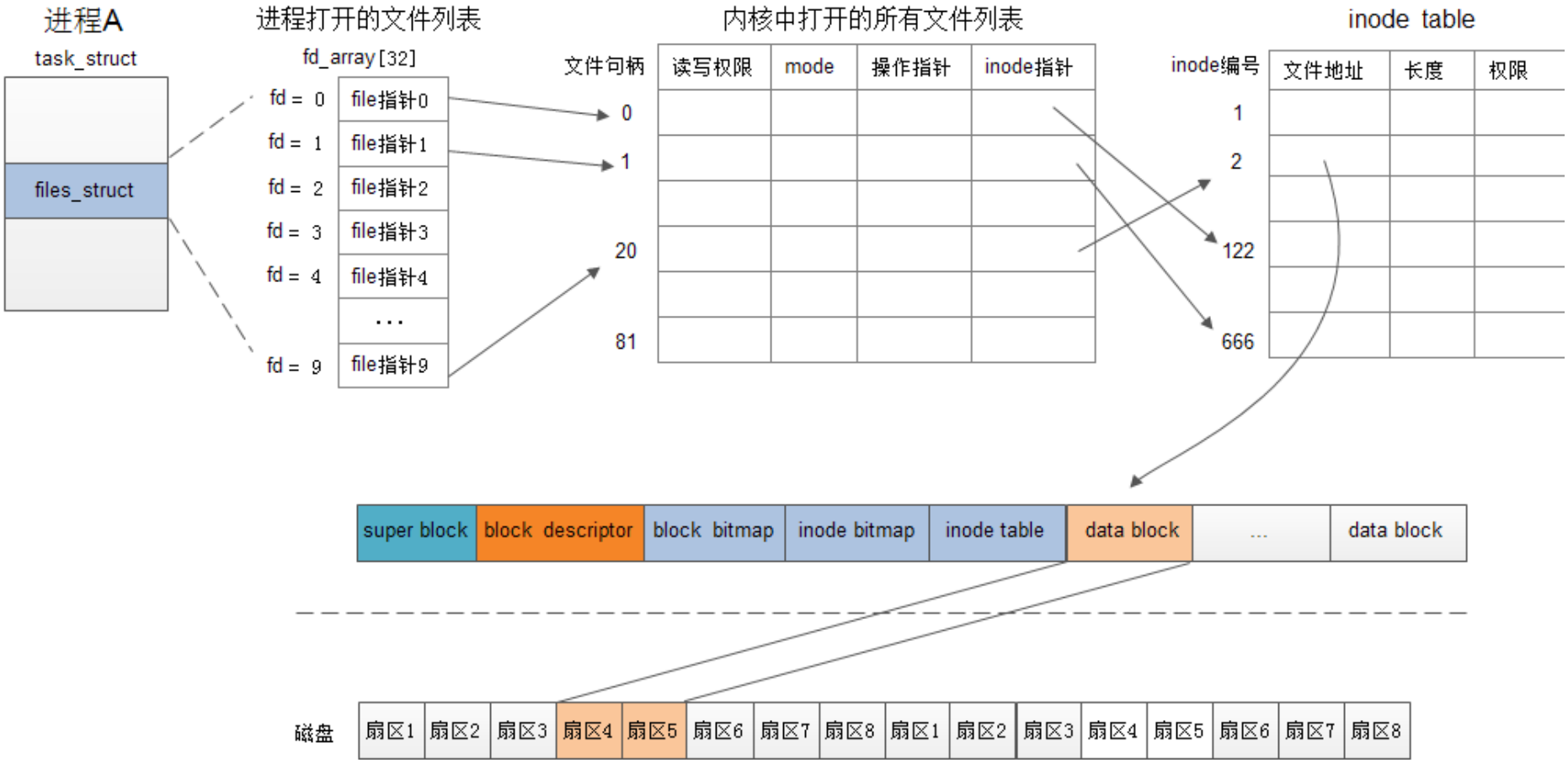
- 结构体files_struct
- 用来表示一个进程打开的文件列表

```
struct files_struct {
        atomic_t                count;
        bool                    resize_in_progress;
        wait_queue_head_t       resize_wait;
        struct  fdtable __rcu   *fdt;
        struct  fdtable         fdtab;
        int                     next_fd;
        unsigned long           close_on_exec_init[1];
        unsigned long           open_fds_init[1];
        unsigned long           full_fds_bits_init[1];
        struct  file __rcu      * fd_array[NR_OPEN_DEFAULT];
};
```

文件

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        }f_u; //所有打开的文件构成一个系统级的全局双链表
        struct    path                  f_path;
        struct    inode                 *f_inode;   //inode节点
        const   struct  file_operations *f_op; //该文件的读写方法
        spinlock_t                      f_lock;
        atomic_long_t                   f_count;
        unsigned int                    f_flags;
        fmode_t                         f_mode;  //打开模式
        struct mutex                    f_pos_lock;
        loff_t                          f_pos;  //文件当前位置
        struct      fown_struct         f_owner;
        const       struct   cred       *f_cred;
        struct      file_ra_state       f_ra;
        u64                             f_version;
        void                            *private_data;
        struct    address_space         *f_mapping;
};
```

一个进程打开的文件


进程A
task_struct
files_struct
进程打开的文件列表
fd_array[32]
fd = 0
fd = 1
fd = 2
fd = 3
fd = 4
fd = 9
file指针0
file指针1
file指针2
file指针3
file指针4
...
file指针9
内核中打开的所有文件列表
文件句柄
读写权限
mode
操作指针
inode指针
0
1
20
81
inode table
inode编号
文件地址
长度
权限
1
2
122
666
super block
block descriptor
block bitmap
inode bitmap
inode table
data block
...
data block
磁盘
扇区1
扇区2
扇区3
扇区4
扇区5
扇区6
扇区7
扇区8
扇区1
扇区2
扇区3
扇区4
扇区5
扇区6
扇区7
扇区8

文件指针

通过C库函数操作文件

- 文件指针
  - C标准库函数
    - C标准库函数使用文件指针来操作文件
    - FILE *fp = fopen(“/home/wit/hello.txt”, “w+”);
    - fread (buf, 4, 100, fp);

C语言中的文件指针FILE*

- 对文件描述符的封装
  - 文件描述符：fd
  - 文件位置:f_fops
  - 缓冲区
  - 文件标志:f_mod

```
typedef struct  _IO_FILE FILE;
struct _IO_FILE {
  int _flags;
  char* _IO_read_ptr;          /* Current read pointer */
  char* _IO_read_end;          /* End of get area. */
  char* _IO_read_base;         /* Start of putback+get area. */
  char* _IO_write_base;        /* Start of put area. */
  char* _IO_write_ptr;         /* Current put pointer. */
  char* _IO_write_end;         /* End of put area. */
  char* _IO_buf_base;          /* Start of reserve area. */
  char* _IO_buf_end;           /* End of reserve area. */
  char *_IO_save_base;         /* Pointer to start of non-current get area. */
  char *_IO_backup_base;      /* Pointer to first valid character of backup area */
  char *_IO_save_end;          /* Pointer to end of non-current get area. */
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int          _fileno;
  _IO_off_t _old_offset;       /* This used to be _offset but it's too small.  */
};
```

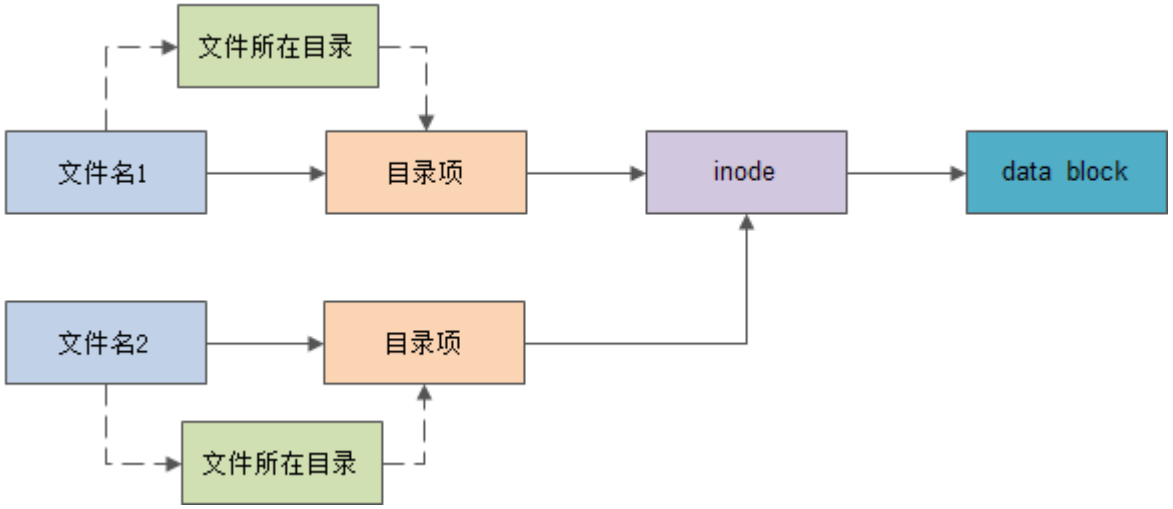
标准流

文件描述符	用途	POSIX名称	stdio流
0	标准输入	STDIN_FILENO	stdin
1	标准输出	STDOUT_FILENO	stdout
2	标准错误	STDERR_FILENO	stderr

硬链接与软链接

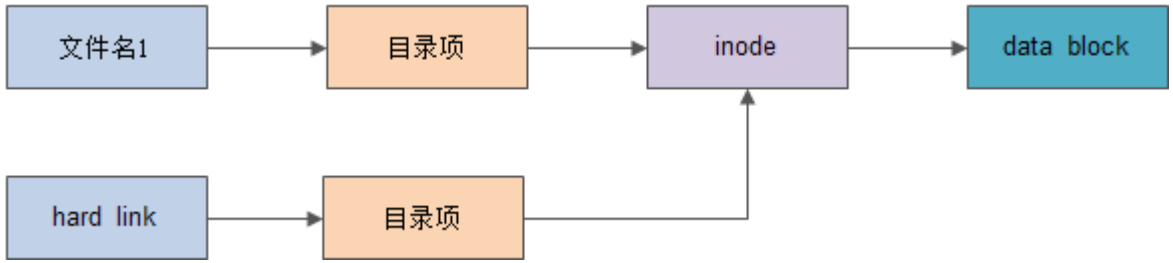
链接

- 基本概念
  - Linux下一种文件共享的方式，类似于Windows下的快捷方式
  - 一个文件使用多个别名：多个文件名共享一个inode



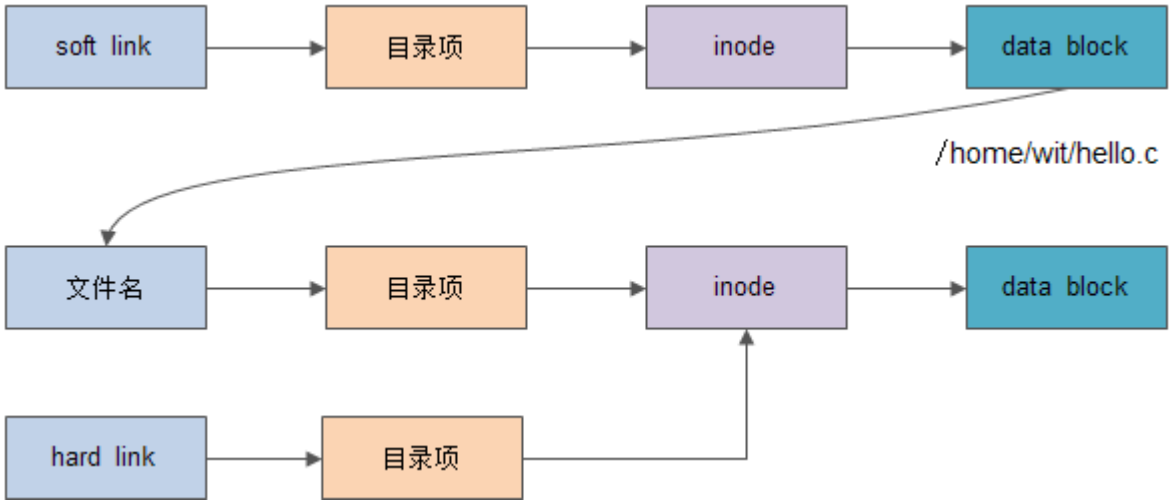
硬链接

- 基本概念
  - 硬链接和文件有相同的inode和datablock



软链接

- 基本概念
  - 软链接是一个普通文件，文件内容为：指向文件的路径名
  - 有自己的inode编号和data block、文件权限、属性…
  - 删除软链接不会影响到其指向的文件本身



实验

- 给一个文件创建一个硬链接　　ln 源文件 硬链接名称
- 给一个文件创建一个软链接　　ln -s 源文件 软链接(文件名称)
- TIPS：$ ln 源文件 目标文件

硬链接与软链接区别

- 硬链接
  - 只能对存在的文件创建硬链接
  - 不能对目录创建硬链接
  - 创建硬链接不能跨越文件系统分区
- 软链接
  - 可以对不存在的文件或目录创建软链接
  - 可以对目录创建软链接
  - 可以跨越文件系统分区创建软链接

思考

- 为什么硬链接不能跨越文件系统或分区创建？ 跨文件分区inode冲突
- 为什么软链接可以跨越文件系统或分区创建？ 有自己的inode,不存在跨越冲突

目录不能有硬链接:目录硬链接产生了很多回环依赖的关系，并且导致了很多歧义性。 a目录下有b目录,在b目录下创建一个a目录的硬链接c,出现了回环依赖,删除a目录,这个时候操作系统没法处理了。

一些命令

跟FS相关

- 常用命令
  - mount
  - mkdir/rmdir/chmod/chown/
  - df、du、wc

如何使用直接 man 命令名称即可

磁盘管理

- 统计磁盘使用率：$ df -h
- 统计目录：$ du 统计目录大小

文件统计

- 当前目录下的C文件个数： $ find . -name “*.c” | wc -l
- 当前目录下(包括子目录)的文件个数：$ ls –lR | grep “^-” | wc –l
- 一个项目的总目录个数：$ ls –lR | grep “^d” | wc –l
- 一个项目的代码总行数：$ find . -name “*.c” |xargs cat | wc -l

作业

- 下载Linux内核最新版本源代码，统计Linux内核：
  - 一共有多少个文件？
  - 有多少个C文件？H文件？汇编文件？
  - 一共有多少个目录？
  - 一共有多少行代码？

实验：磁盘格式化及挂载

使用磁盘

- 基本步骤
  - 格式化、分区
  - 安装文件系统
  - 挂载：将磁盘挂载到我们电脑上的某个目录上

格式化:fdisk 磁盘如:/dev/sdb
分区:输入n添加分区,输入回车(默认主分区),设置分区号,默认1则回车,第一个扇区大小,回车默认,继续默认,最后输入w保存

mkfs.ext4 磁盘如:/dev/sdb1 给sdb1分区安装ext4文件系统

mount -t ext4 /dev/sdb1 /abc/1 将sdb1挂载挂载到/abc/1 指定类型ext4

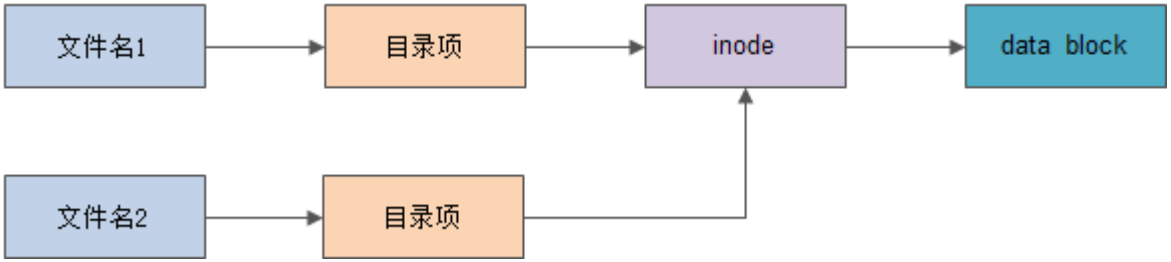
问题

- 为什么磁盘要先分区、格式化才能使用

恢复删除的文件

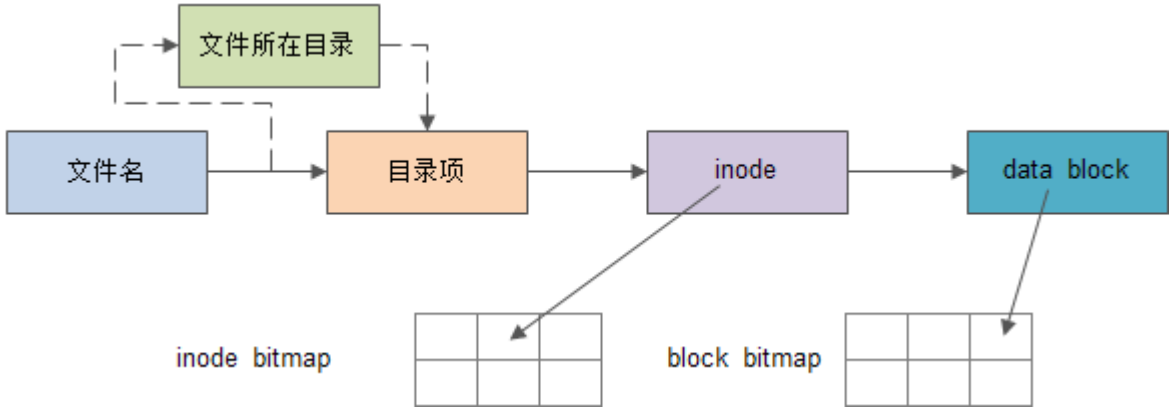
rm命令…

- 文件删除的背后
  - 并没有真正删除数据，当inode的链接计数>1时
  - 仅删除了文件名和inode之间的关联：清除了目录项中inode指针信息，



真正的文件删除

- 当inode的链接计数为1时
- 将目录项中的文件名和inode对应关系删除
- 将该文件inode中的block指针删除
- 在inode bitmap中将该文件的inode标记为未用
- 将block bitmap中将该文件的data block标记为未用
- ext日志文件系统：把删除文件的inode信息和文件名写入日志



恢复删除的文件

- 基本步骤
  - $ apt-get install extundelete
  - $ extundelete --inode 2 /dev/sdb1　　查看某个分区日志
  - $ extundelete /dev/sdb1 --restore-file /home/wit/hello.c
  - $ extundelete /dev/sda1 --restore-inode 1122　　在日志中找到删除的inode
  - $ extundelete /dev/sdb1 --restore-directory /home/wit/test

恢复操作最好在其他分区上,避免需要恢复磁盘的影响导致恢复失败

sync 命令强制将缓冲区写到磁盘