fs实验

tips

- 元数据加锁规则
  - 元数据的加锁操作基本上都是VFS来做的
  - 在VFS中，对于大多数的元数据操作，可以通过制定统一的加锁顺序来避免死锁的发生
    - 先对父目录加锁，再对要操作的对象（目录或文件）加锁
      - 以创建操作为例，当我们要在一个目录下创建新的对象时，必须先对这个目录进行加锁，然后才能放心地进行创建对象的步骤，由此也不用担心该目录会被中途删除，或者其他操作对该创建操作所造成的干扰
  - 按照元数据名字空间的树状结构来看，**我们可以认为**
    - **先对父目录加锁，再对要操作的对象（目录或文件）加锁**
    - 是一种从上到下的加锁顺序。只要所有的元数据操作都遵循这个规则，就不会出现相反的加锁顺序（即从下到上的顺序），那么也就不会出现两个操作因为互相等待对方的锁而产生死锁的情况
  - 这个普遍的加锁规则对于绝大多数元数据操作都是适用的
    - 有两个元数据操作例外 **rename和link**
    - 因为它们的操作对象不止两个，而这些对象可能位于名字空间树状结构的任意几个位置，导致加锁的路径有可能不在名字空间树状结构**从上到下**的范畴内。对于这两个元数据操作，我们需要对其进行特殊的考虑和处理

go on

- https://zhuanlan.zhihu.com/p/183238194
  - 小林图解文件系统
- 核心 tips，在 ostips 中也会有一份
  - buffer head 代表的是 **磁盘块在内存中的映射**
    - bh
  - 文件系统中，只有真正的用户数据属于 data，**其它一切均属于 metadata**
    - 包括目录等
  - 驱动写数据到设备 **一定是异步的**
    - 上层想要同步阻塞的操作必须 **自己写 for 去轮询** ，等待内存中某位置处值的改变
      - 而这个值的改变一般是 DMA 控制器的中断把 IO 端口的数据 copy 到内存中的某位置的
  - ext4 fsync
    - 文件系统的一致性是由 metadata 来保证的
    - **data 是先于 metadata 落盘的**
    - 即使 data 与 metdata 持久化的过程中 crash，**数据落盘也没有原子性保证**
      - 但是只要 **元数据在数据之后落盘**，并且保证元数据的一下特性，ext4 从来没有保证过数据的一致性（除非用户指定模式 journal data）
        - 元数据的原子性
          - 比如说bitmap，inode等等要么全改，要么全不改，这个就需要事务来保证
          - 所以在 journal 中会保存 metadata 修改后的数据，事务执行期间 crash 的话，是需要根据这个做 redo log 的（**用于恢复真实位置处的 metadata**）
            - 事务实际保证的是 metadata 的持久性以及原子性
            - 隔离性也是事务来保证
        - 最终，**元数据的一致性** 由于 journal 机制而得以保证
          - 即事务是用于保证文件系统的 **元数据的一致性** 的
      - 现在 metadat 的一致性是有保障的，如果 metadata 先落盘，但是数据奔溃，如果journal是没有journal data 的，那这种情况下，文件系统是不可能回到一致性状态的
      - 反之，data 先落，如果 metadata （包括bitmap所在的块）落盘失败，那么 data 那部分的空间在 bitmap 看来也是可以继续使用的，一致性完全没有任何问题
  - ext4 目录操作 与 文件操作
    - 目录操作是一个纯的 metadata 操作，操作结束后会主动触发一个事务的提交
    - 文件操作例如写文件，是 data 操作，操作结束后会等待 timer 触发事务的提交
  - 需要读写文件的原因
    - 假设存储服务器盘上有 10 部电影 20G。OS 起来之后，给文件的 memory cache 只有一个视频的大小 2G
    - 现在这个存储服务器要并发的应对 20 个人的下载需求（**20个人各自下载一部电影**），数据 IO 路径是
      - OS 从文件系统读取电影 A
      - 电影 A 的数据分块（block）读取到 memory 中，当 block 通过网络发出去之后，memory 空出来了

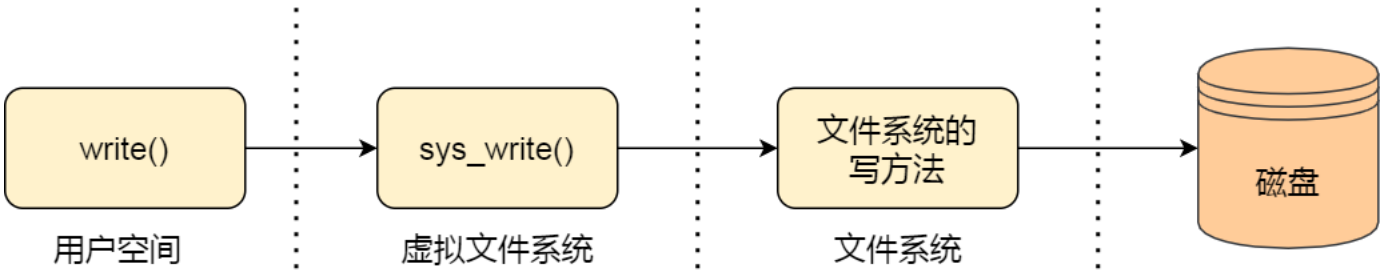
        - 可以继续 加载电影 A 的其它部分或 加载电影 B（以及其它电影）的部分，当 memory 满的时候，是需要等待 网络 把 memory 中的数据消化掉的
    - **确实是面临大量的 storage IO 过程的**
        - 20G的IO流量由storage到memory以及20G的网络流量由存储服务器到client是绝对无法避免的
            - 之前这个地方确实有没有想明白的地方
- handle
    - 这里要先聊一下 current 宏
        - https://kernelnewbies.org/FAQ/current
        - 我之前一直以为指向的是 task，即 thread，现在看起来这个理解是 **错误** 的
            - **current 指向的是 process**
    - 所以 handle 是属于 process 的，**自然 process 内的多 thread 自然就有并发了**
- 关于 **日志系统** 数据结构的 **大小关系**
    - `journal > transaction > handle > buffer head`
        - 一个 journal 属于一个文件系统
        - 一个 journal 中包含若干个 **事务**
        - 一个 事务 中包含若干 handle
            - 一个 handle 对应于 **一个process**，一个事务中有多个 handle，来自多个 **process**
        - 一个 handle 中包含若干个 **buffer head**
            - 一个 handle 的本意确实 是一个 **文件系统接口所涉及的内容**
            - 但是由于 多 thread 并发。并发 的 文件系统接口 会进入到一个 handle 中
- **文件系统接口的一致性** 再理解一下
    - 就以 ext4 为例，仅仅 journal metadata。这样是能够保证文件系统的一致性的
    - 但是，假设这样的一个情况，update data，且update已经持久化，在持久化 metadata 到 journal 的过程中，系统GG
        - 那么系统恢复之后，文件系统确实是一致的，即 inode 所指向的 block 以及 bitmap 所指向的 block 的使用情况，以及 inode 中文件大小的字段都没有改变，所以说文件系确实是一致的
    - 但是我们可以看到这个过程中，**是有数据被update的**，这个对于上层应用来说是灾难性的
    - 当然文件系统也提供了这个的解决方案，即代价最大的一致性操作，保证 metadata 一致性的前提下，保证 data 的一致性，即data也需要去 落盘到 journal 区
        - 系统奔溃后的重启中就可以 redo 这部分数据以保证数据的正确性
    - 文件系统的 **一致性与数据的正确性** 之前的理解是有问题的
        - 所以一些系统软件的应用是需要额外的机制（仅仅 journal metadata 的情况下）来保证数据的准确性的
        - 例如 数据库事务，日志系统的一些额外操作等等
            - 与佩哥聊日志系统有感
                - 日志系统一次写入的内容是 log entry
                - 智能网卡里跑的是一个 arm 核，arm 核上跑的是一个精简的 linux
                    - 如果想要改变智能网卡的一些行为，能够把 修改反应到 kernel 或 arm 上跑的 app 上

把小林的再捋一遍

- 文件系统的基本数据单位是文件，它的目的是对磁盘上的文件进行组织管理
    - 那组织的方式不同，就会形成不同的文件系统
- Linux 最经典的一句话是
    - 一切皆文件
    - 不仅 **普通的文件和目录**，就连**块设备**、**管道**、**socket** 等，也都是统一交给文件系统管理的
        - IPC 的抽象就是文件
- Linux 文件系统会为每个文件分配两个数据结构
    - 索引节点（index node）
        - 元数据
            - 编号，文件大小，访问权限，创建时间，修改时间
        - in disk
        - 文件的唯一标识
    - 目录项（directory entry）
        - 代表目录的层次结构

        - 文件系统目录树结构的关键数据结构，为什么不用 inode 呢，因为可能会创建硬链接，同一个 inode 会出现在树中的不同节点中
            - 所以必须引入 dentry
                - 多个 dentry 可能会指向同一个 inode
- 那文件数据是如何存储在磁盘的呢
    - 磁盘读写的最小单位是扇区，扇区的大小只有 512B 大小，很明显，如果每次读写都以这么小为单位，那这读写的效率会非常低
    - 文件系统把多个扇区组成了一个**逻辑块**，每次读写的最小单位就是逻辑块（数据块），Linux 中的逻辑块大小为 4KB，也就是一次性读写 8 个扇区，这将大大提高了磁盘的读写的效率
        - 文件系统看到的实际上是一个 **逻辑上的存储空间**
            - 线性的数组
- 磁盘进行格式化的时候
    - 会被分成三个存储区域，分别是
        - 超级块
            - 整个文件系统的元数据
            - 挂载时加载到内存，常驻内存
        - 索引节点区
            - 磁盘 inode 存储区
            - 文件被访问时加载
        - 数据块区
            - real data

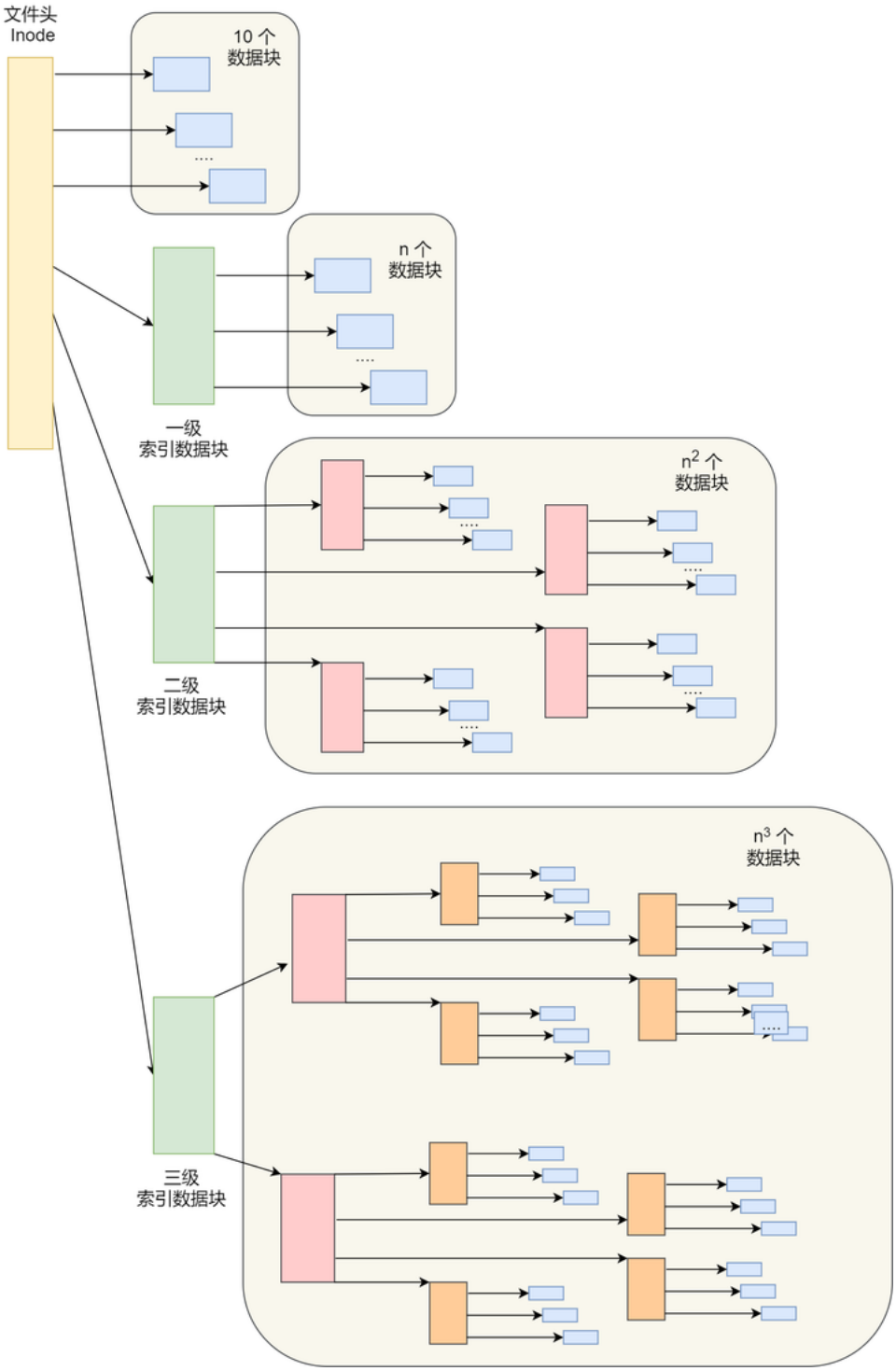
- 虚拟文件系统
    - 文件系统的种类众多，而操作系统希望对用户提供一个统一的接口，于是在用户层与文件系统层引入了中间层，这个中间层就称为虚拟文件系统（Virtual File System，VFS）
    - 采用 dynamic dispatch 的方式支持运行时绑定 **具体的 op 函数**
- Linux 支持的文件系统也不少，根据存储位置的不同，可以把文件系统分为三类
    - 磁盘的文件系统，它是直接把数据存储在磁盘中，比如 Ext 2/3/4、XFS 等都是这类文件系统
    - 内存的文件系统，这类文件系统的数据不是存储在硬盘的，而是占用内存空间，我们经常用到的 /proc 和 /sys 文件系统都属于这一类，读写这类文件，实际上是读写内核中相关的数据数据
    - 网络的文件系统，用来访问其他计算机主机数据的文件系统，比如 NFS、SMB 等等
        - 构建 NAS，文件系统在服务端



- 文件系统的使用
    - 操作系统为每个进程维护一个打开文件表
        - 文件表里的每一项代表 **文件描述符**，所以说 **文件描述符** 是 **打开文件的标识**
            - 句柄
    - 操作系统在 **打开文件表** 中维护着 **打开文件的状态和信息**
        - 内核为所有打开文件维持一张文件表项
            - entry
        - 两个独立进程各自打开同一个文件，打开该文件的每个进程都得到一个文件表项。每个进程都有自己的文件表项的一个理由是
            - 使每个进程都有它自己的对该文件的当前偏移量
- 用户和操作系统对文件的读写操作是有差异的
    - 用户习惯以 **字节** 的方式读写文件
    - 而操作系统则是以 **数据块** 来读写文件
    - 那屏蔽掉这种 **差异** 的工作就是 **文件系统了**
        - 当用户进程从文件读取 1 个字节大小的数据时，文件系统则需要获取字节所在的数据块，再返回数据块对应的用户进程所需的数据部分
        - 当用户进程把 1 个字节大小的数据写进文件时，文件系统则找到需要写入数据的数据块的位置，然后修改数据块中对应的部分，最后再把数据块写回磁盘
    - 文件系统的 **基本操作单位** 是 **数据块**

        - 写放大实际上是比较大的

- 文件的存储，在内存中，**文件一定是按顺序存放的** （radix tree）
    - 连续空间（物理空间）存放方式
        - **读写效率高**，但是会造成磁盘碎片，并且文件不易扩展
            - **实际上用的很少**
    - 非连续空间（物理空间）存放方式
        - 链表方式
            - 隐式链表
                - 形式
                    - inode 包括 **第一块** 与 **最后一块** 的地址
                    - 每个数据块中的 **最后** 用于存放指向下一个块的 **指针**
                - 只能顺序访问，性能很差
            - 显式链表
                - 用 **空间** 来换 **时间**
                - 隐式连接中分别存储在每一个块中的 **块指针** 可以 **单独存放一张表**
                    - 能够弥补必须顺序访问的时间开销
                - **太大就不太行了**
        - 索引方式
            - 单独拿一个块来做一个文件的索引块
                - 比如 ext2 的 inode 就是有索引项的
                    - 大部分直接索引，最后几项间接索引
                - **索引块中存放指向文件数据块的指针**



- Unix 文件的实现方式
  - 它是根据文件的大小，存放的方式会有所变化
    - 如果存放文件所需的数据块小于 10 块，则采用 直接查找 的方式
    - 如果存放文件所需的数据块超过 10 块，则采用一级间接索引方式
    - 如果前面两种方式都不够存放大文件，则采用二级间接索引方式
    - 如果二级间接索引也不够存放大文件，这采用三级间接索引方式
  - 那么，文件头（Inode）就需要包含 13 个指针
    - 10 个指向数据块的指针
    - 第 11 个指向索引块的指针
    - 第 12 个指向二级索引块的指针
    - 第 13 个指向三级索引块的指针
  - 所以，这种方式能很灵活地支持 **小文件** 和 **大文件** 的存放
    - 对于小文件使用直接查找的方式可减少索引数据块的开销
    - 对于大文件则以多级索引的方式来支持，所以大文件在访问数据块时需要大量查询

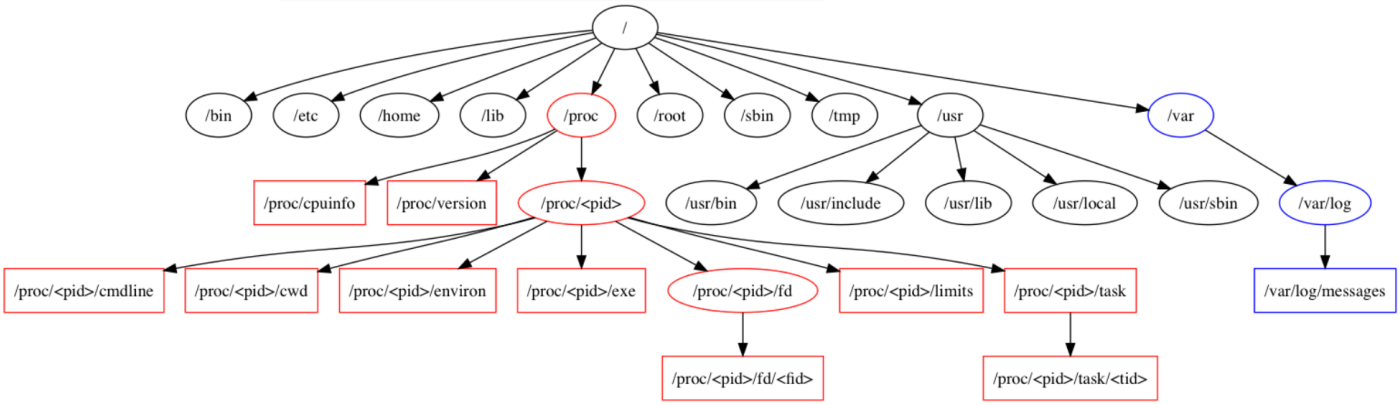
    - 这个方案就用在了 `Linux Ext 2/3` 文件系统里，虽然解决大文件的存储，但是对于大文件的访问，需要大量的查询，效率比较低
    - 为了解决这个问题，Ext 4 做了一定的改变，具体怎么解决的，本文就不展开了

- 空闲空间管理
    - 前面说到的文件的存储是针对已经被占用的数据块组织和管理，接下来的问题是，如果我要保存一个数据块，我应该放在硬盘上的哪个位置呢？难道需要将所有的块扫描一遍，找个空的地方随便放吗
    - 那这种方式效率就太低了，所以针对磁盘的空闲空间也是要引入管理的机制，接下来介绍几种常见的方法
        - 空闲表法
            - 建立一张表，表示哪些 **空间** 是空闲的
            - 额外的空间开销
                - 如果有大量的小空闲空间，那么分配，索引，空间开销的影响都非常大
        - 空闲链表法
            - 所有的空闲空间用 list 来组织，没有额外的空间开销
            - 修改指针都是 IO，效率很低
                - 还没法随机访问
                    - 内存中，数组能随机访问，但是插入删除效率低；链表必须遍历索引，但是插入删除的效率还是可以的
                    - 磁盘中，如果有list，修改指针就意味着磁盘 IO
        - 位图法
            - 前两种反正都不太行，位图就简单了，空间消耗最低了
            - 位图是利用二进制的一位来表示磁盘中一个盘块的使用情况，磁盘上所有的盘块都有一个二进制位与之对应
                - **位图所在的块更新的频率可能会比较高**
            - 位图块是若干个块
                - 不是仅仅一个 block

- 目录的存储
    - **可以通过 vim 打开它**
    - 在目录文件的块中，最简单的保存格式就是列表
        - 就是一项一项地将目录下的文件信息（如文件名、文件 inode、文件类型等）列在表里
        - 列表中每一项就代表该目录下的文件的文件名和对应的 inode
            - **通过这个 inode，就可以找到真正的文件**
    - 如果一个目录有超级多的文件，我们要想在这个目录下找文件，按照列表一项一项的找，效率就不高了
        - 引入了 hash
    - 目录查询是通过在磁盘上 **反复搜索完成**
        - **namei**
        - **需要不断地进行 I/O 操作，开销较大**
        - 为了减少 I/O 操作，把当前使用的文件目录缓存在内存

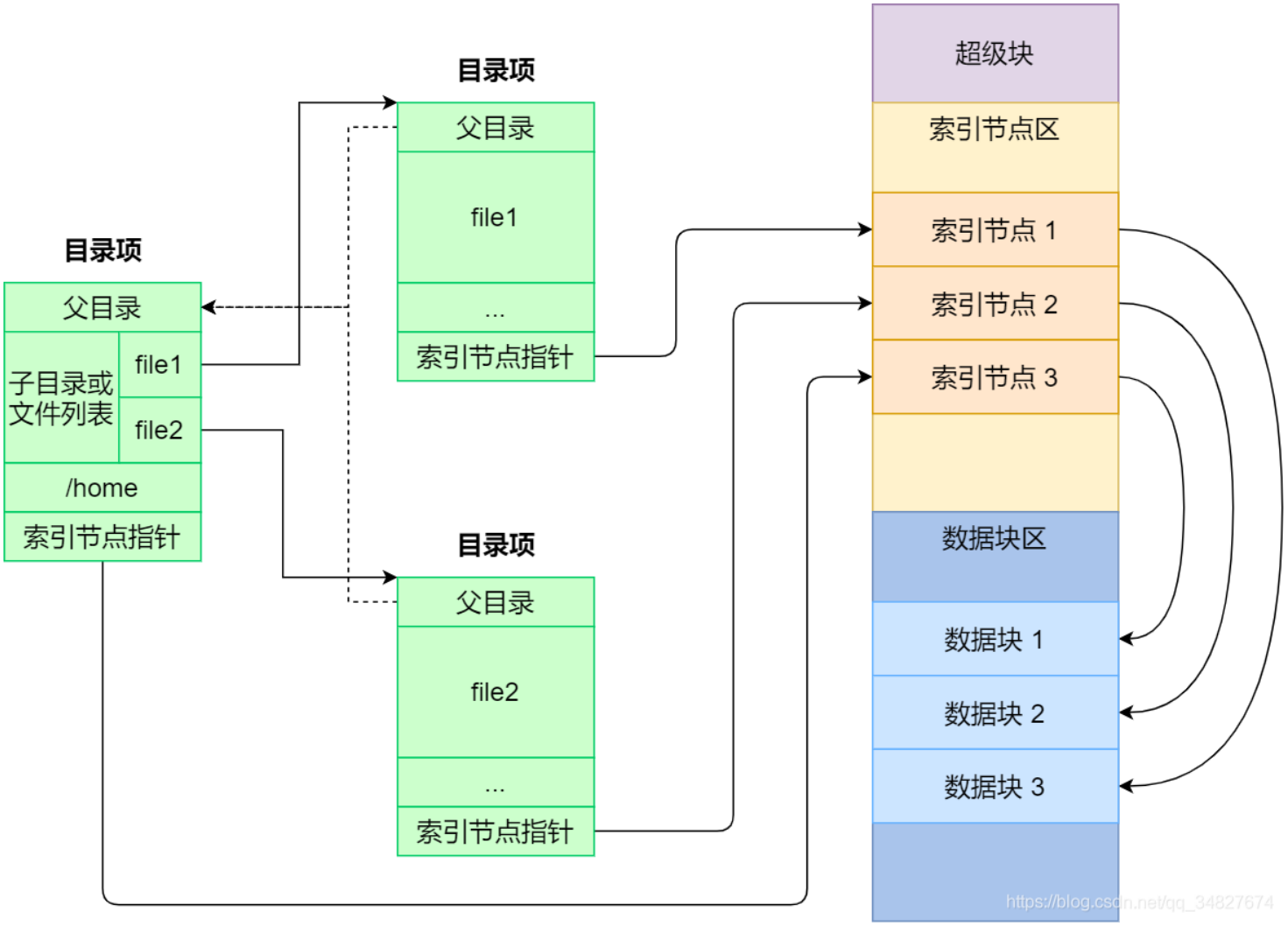
文件系的核心理解

> **内核维护的 struct inode（ 文件名只是 `inode` 号的一个别名，名字反而是记录在 `dentry` 中 ） 以及 struct dentry 才是理解的根本**



```
root@ab069b9d443a:~# ls -i
1444927 cli-pub-keys  1841356 codesnip  1446221 home_conf  1446084 storage_test  1444929 ucore
root@ab069b9d443a:/proc# ls -i
  10334375 1        10334385 66247    10334401 84385      4026532086 diskstats         64035 kcore        4026532163 mpt
  ...
root@ab069b9d443a:/dev# ls
console  core  fd  full  mqueue  null  ptmx  pts  random  shm  stderr  stdin  stdout  tty  urandom  zero
root@ab069b9d443a:/dev# ls -i
    3 console  64042 fd    63441 mqueue  64041 ptmx  64036 random  64045 stderr  64044 stdout  64040 urandom
64046 core     64037 full  64035 null        1 pts   64033 shm     64043 stdin   64038 tty     64039 zero
```

linux一切皆文件



inode

- 一个文件 **唯一对应** 一个inode（**文件的元数据**）
  - 无论是 设备文件 ，还是 存储介质 上的文件，亦或是 /proc伪文件系统 下的文件都会唯一对应一个 inode 号，即一个 inode 数据结构
    - 这是一个内核维护的数据结构 **struct inode in linux/include/linux/fs.h**
  - **不同文件系统 inode 号间是独立的**

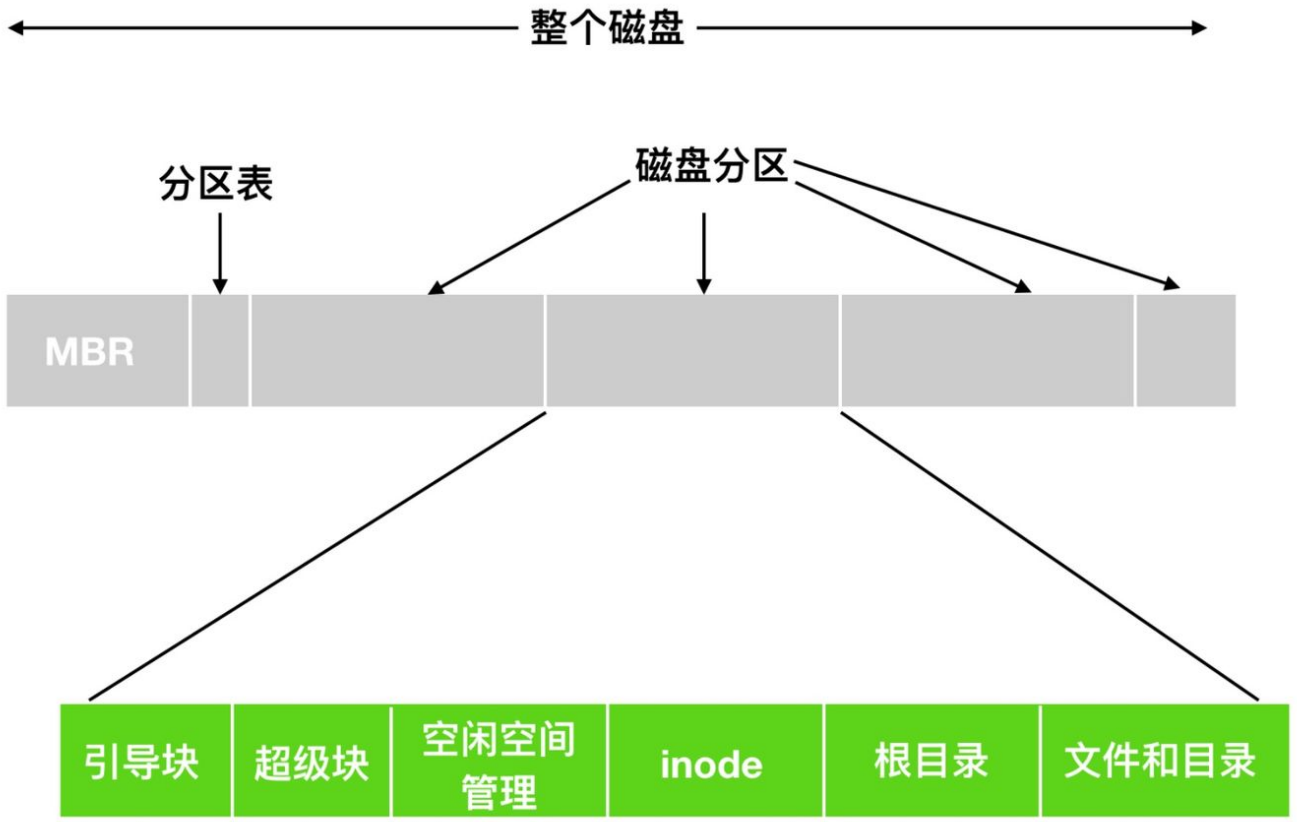
dentry

- 目录项
  - 操作系统目录树中的每一个节点都是**目录项**
    - **是目录项构成了树形结构**
    - /home/doubled

        - 其中 `/`，`home`，`doubled` 分别是三个目录项
    - 同样是内核维护的一个数据结构 **struct dentry in struct inode in linux/include/linux/dcache.h**
    - 为什么 inode 做目录树的节点不行
        - **hardlink 的存在会使得不同的目录对应相同的 inode**，所以目录项是目录树中的节点，**会指向一个 inode**
            - 一个 inode 可能会对应多个目录项
- **系统中，file 通过 dentry 去找 inode**
    - **理解为 dentry 更软**

磁盘文件系统 ext2/3/4

> 代表一个文件的数据结构是一个 **内核数据结构** 超级块 **struct super_block**，位于 **struct inode in linux/include/linux/fs.h**



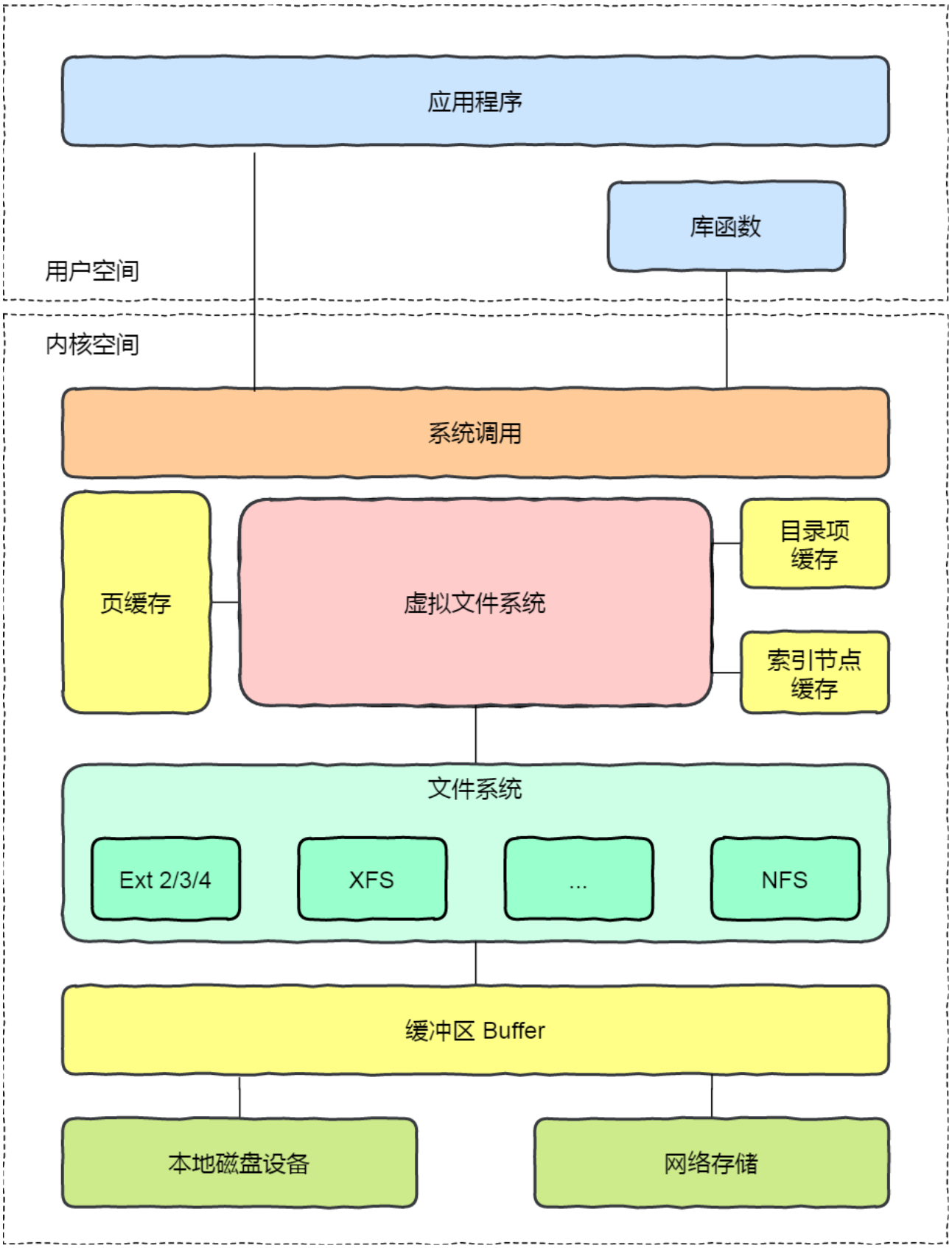
文件系统布局

- 同样在存储介质上，文件系统中也是一个 **目录树的格式**，但是仅仅 inode 是存盘数据，目录项仅存在于内存中
    - 中间节点代表目录文件
        - **map<文件名字, inode>**
    - 叶子节点代表普通文件
        - **二进制的有序字节流**
            - 这个序列在内存中的组织方式是有序的，但是在介质上是看文件系统如何管理的

```
struct ext4_super_block{...}
struct ext4_inode{...}
```

- **超级块** 与 **inode** 会存盘，这个数据结构的定义在对应的文件系统中，如上所示
    - **内存超级块与inode** 均包含对应的 **磁盘超级块与inode** 数据

VFS


应用程序
库函数
用户空间
内核空间
系统调用
页缓存
虚拟文件系统
目录项
缓存
索引节点
缓存
文件系统
Ext 2/3/4
XFS
...
NFS
缓冲区 Buffer
本地磁盘设备
网络存储

VFS初始化

文件系统基础知识

file_system_type与文件系统的注册

- in fs/filesystems.c
  - 系统中所有 **已注册** 的文件系统位于 /proc/filesystem

```
// 静态全局变量，文件系统类型的 list
static struct file_system_type *file_systems;
struct file_system_type {
        const char *name;  // 文件系统的名字
        int fs_flags;
        ...
        int (*init_fs_context)(struct fs_context *);  // 挂载文件系统时调用
        struct dentry *(*mount) (struct file_system_type *, int, const char *, void *);  // 挂载文件系统时调用
        void (*kill_sb) (struct super_block *);
        ...
        struct file_system_type * next;  // 每一个文件系统类型都是全局list上的一个node
        struct hlist_head fs_supers;  // 每一个文件系统都可以有多个实例，这个list用来保存系统中的文件系统的实例
        ...
};
// 文件系统的类型的实例是写在代码中的，如下所示，分别会在需要的时刻被注册到系统中
// in fs/mount.c
static struct file_system_type sysfs_fs_type = {
        .name                   = "sysfs",
        .init_fs_context        = sysfs_init_fs_context,
        .kill_sb                = sysfs_kill_sb,
        .fs_flags               = FS_USERNS_MOUNT,
};
int __init sysfs_init(void){
        ...
        err = register_filesystem(&sysfs_fs_type);
        ...
}
// in fs/ext4/super.c
static struct file_system_type ext4_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "ext4",
        .mount          = ext4_mount,
        .kill_sb        = kill_block_super,
        .fs_flags       = FS_REQUIRES_DEV,  // ext4表示需要后备存储设备
};

static int __init ext4_init_fs(void){
        ...
        err = ext4_init_sysfs();
        ...
}
```

- 部分文件系统的注册是系统初始化时指定的，主要是一些 **伪文件系统**。例如 **sysfs，procfs**，在 main.c 的 start_kernel 中会显示的调用 sysfs_init 以及 shmem_init
- 其它文件系统 **模块初始化** 的时候会 **主动注册** 对应文件系统到 OS，例如 ext4

```
# 系统初始化完成后，在挂载ext4文件系统前可观察到ext4文件系统已经被注册到了系统中，但是并无实例
# sysfs 不仅被注册到了文件系统中，且拥有一个对应的实例
$44 = (struct file_system_type *) 0xffffffff82573420 <ext4_fs_type>
<ext->next->next->next->next->next->next->next->next->next->next->fs_supers
$45 = {first = 0x0 <fixed_percpu_data>}

(gdb) p file_systems
$41 = (struct file_system_type *) 0xffffffff82573060 <sysfs_fs_type>
(gdb) p file_systems->fs_supers
$42 = {first = 0xffff88800494b0e0}
```

文件系统的挂载

```
struct fs_context {
        const struct fs_context_operations *ops;
        ...
        struct file_system_type *fs_type;
        ...
        struct dentry           *root;          /* The root and superblock */
        ...
};
```

- mount文件系统的时候，对应文件系统一定被注册到OS中
- 文件系统的挂载是由一个叫做**文件系统上下文**的实例**fc**来完成的
  - **挂载完成后，该实例会被销毁**
- 主要结果
  - 加入到 `file_system_type` list对应的节点中
  - 创建一个**super block**, **root dentry**以及**root vfs_inode**
  - 创建一个 `struct vfsmount` 的实例
    - `struct vfsmount` 用于在内核中表示一个 **已挂载的文件系统实例**
      - 通过它系统中所有的文件系统就可以形成一个 **挂载树**，也就形成了一个 **目录树**，因此通过根文件系统就可以遍历系统中所有的文件系统及其所有的文件或目录。须要特别强调的是
        - 每个挂载点在内核中都对应着 **两个目录项**
          - 一个属于 **被挂载的文件系统**
          - 另一个则属于当前文件系统 **根目录** 所对应的 **目录项**

root文件系统的挂载

- **root文件系统** 没有注册，直接通过调用 `init_mount_tree() in fs/dcache.c` 挂载

```
struct file_system_type rootfs_fs_type = {
        .name           = "rootfs",
        .init_fs_context = rootfs_init_fs_context,
        .kill_sb        = kill_litter_super,
};
```

其它文件系统的挂载

```
SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *, dir_name, char __user *, type, unsigned long, flags, void __user *, data){
        ...
        ret = do_mount(kernel_dev, dir_name, kernel_type, flags, options);
        ...
}
```

- 显示的调用 `do_mount` in `fs/namespace.c`

VFS初始化的基本流程

- start_kernel
  - vfs_caches_init
    - dcache_init
      - 创建 **dentry cache**
    - inode_init
      - 创建 **inode cache**
    - files_init
      - 创建 **file cache**
    - mnt_init
      - sysfs_init
        - 注册**sysfs**
      - shmem_init
        - 注册**tmpfs**

        - init_mount_tree
            - **挂载root文件系统**
                - 一般是一个ramdisk或tmpfs文件系统，就是**initrmfs**

/sys

- https://www.jeanleo.com/2020/03/30/linux设备驱动框架剖析/
- start_kernel
    - vfs_caches_init
        - **sysfs文件系统的注册**
    - proc_root_init
        - **proc文件系统的注册**
    - arch_call_rest_init
        - `pid = kernel_thread(kernel_init, NULL, CLONE_FS)`
        - kernel_init
            - kernel_init_freeable
                - do_basic_setup
                    - driver_init
                        - devices_init
                            - **sys下devices的目录项**，如下同理
                        - buses_init
                        - classes_init
                        - firmware_init
                        - platform_bus_init
                - do_one_initcall
                    - 所有编译进内核的驱动都会被执行 `module_init`
                        - **在这里会有大量的总线，设备以及驱动被注册到系统中**
            - run_init_process（**最后**）
                - 挂载真正的root文件系统
                - 将**sysfs**挂载到 `/sys` 目录下

/proc

- 部分内容
    - `/proc/devices`
        - 系统已经**加载**的所有块设备和字符设备的信息
            - **驱动被编译到内核**的话就会在内核启动时**被加载**
    - `/proc/filesystems`
        - 已 **注册** 文件系统，nodev 表示无需后备存储设备，**ext4就必须有dev**
        - 这个地方是非常全的
- 注册发生在内核初始化的 main 函数中，挂载由 init 进程完成

/dev

- 该目录下的文件是真实的设备文件，是应用层通过 **mknod** 创建的文件
    - 通常系统中是由udev在运行时创建的
    - 在busybox中可以通过mdev自动装配dev下的设备文件
        - **mdev -s**
            - `mdev -s` is to be run during boot to **scan /sys** and **populate /dev**

VFS四大数据结构

- VFS所需要的内存
    - **i-cache**与**d-cache**
        - 建立i-cache与d-cache的同时为它们建立hash索引
    - **pagecache**

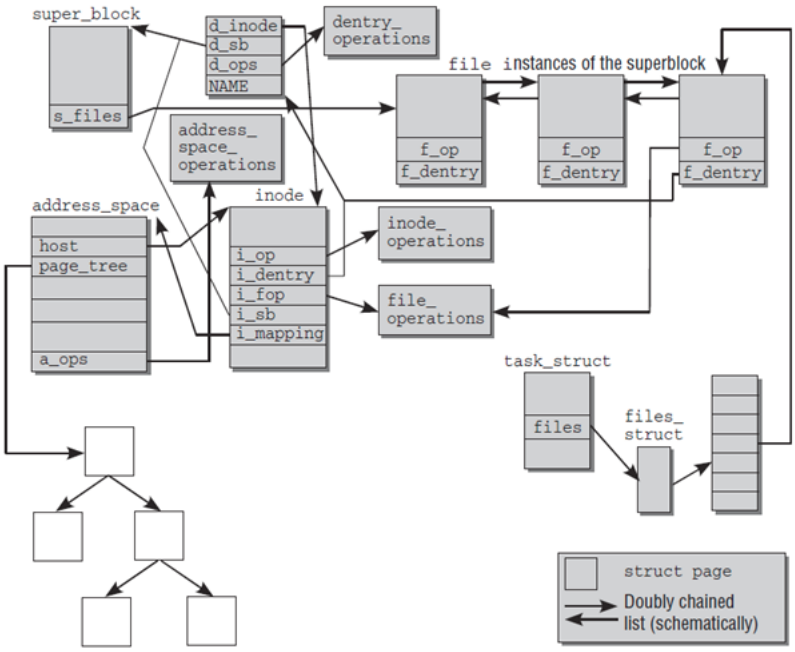
```
// 初始化操作大都在分配slab cache
void __init vfs_caches_init(void)
{
    ...
        dcache_init();  //
        inode_init();  // inode cache

        files_maxfiles_init();
        mnt_init();
        bdev_cache_init();
        chrdev_init();
}
```

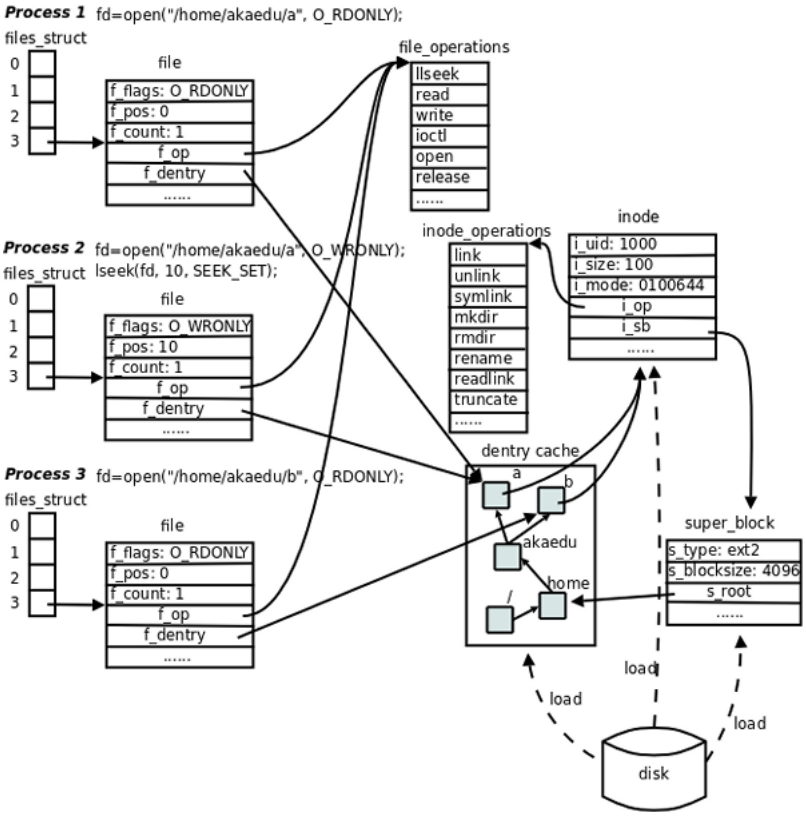
- VFS维护着一个全局的静态list表示系统内所有的文件系统类型

```
static struct file_system_type *file_systems;
void __init mnt_init(void)
{
    ...
        err = sysfs_init();  // register_filesystem(&sysfs_fs_type);
    ...
        shmem_init();  // tmpfs, /dev就是tmpfs类型的
        init_rootfs();
        init_mount_tree();
}
```

1. 向系统注册sysfs
    - https://www.jeanleo.com/2020/03/30/linux设备驱动框架剖析/
2. 向系统注册tmpfs
    - **/dev就是挂载在这里的**
3. 初始化rootfs
4. 通过**init_mount_tree**安装根文件系统

VFS中的四大数据结构





- super_block
- file
- dentry
- inode

VFS层相关的系统调用

- fs相关的系统调用位于 `linux/fs` 下的不同模块中
  - 例如open系统调用位于 `open.c`
  - 读写相关的系统调用位于 `read_write.c`
- 以下系统调用的相关介绍均来自最新版内核（**我写这个文档的时候的最新版内核**）
  - 自己的调试过程也在这里来说明
  - 系统调用的调试与 **starce** 对比着看
    - 有些bin下面的命令不确定会调用到什么系统调用，就用**strace**
- gdb中可以使用调用栈bt来查看函数调用关系

先要来一个综述，把 目录操作 文件操作 以及 文件系统的一致性保证都在这里说一下，一致性保证可以在 fs-dev 中说

- 关于持久化数据
  - 存盘数据包括 **正则文件** 与 **目录文件**，都有 inode 号
  - 目录文件中有记录 **inode号** 与 **子目录/子文件** 名字的 **map**
  - dentry 是如何 build 的呢
    - **超级块** 记录了文件系统的挂载点，即 root 目录的位置，即 root 是一个目录文件，inode 号是 1
    - inode 表的起始位置也是在 **超级块** 中所保存的，所以知道 inode 号是可以在 storage 中找到 inode 的
      - inode 号就是 **按顺序** 排列的
    - 找到 inode 号，并且读到内存中，变成 vfs_inode，vfs_inode 会指向实际数据的指向，所以可以把 目录文件 root 的内容读到内存中
      - 这个时候就可以创建 root 下一级目录 对应的 dentry 了，即 文件系统 tree 的第一层
      - 这个 inode 在 page_cache 中还有一份

            - raw_inode
        - **根据目录文件中的 inode 号就能够 build 了**
- 理解一下 vfs_inode 与 **raw_inode**
    - vfs_inode
        - 用来算的，位于内核栈或堆上，不会被用于存盘，kernel 中修改它 **什么用都不会有**
    - **raw_inode**
        - 保存在 buffer head 中的，用于存盘的部分，修改 inode 的时候要 update 这部分位于 page cache 的 inode

```
// ext4 文件系统中被 dispatch 函数的具体实现部分
inode->i_op = &ext4_file_inode_operations;
inode->i_fop = &ext4_file_operations;
inode->i_op = &ext4_dir_inode_operations;
inode->i_fop = &ext4_dir_operations;

// ext4文件系统
(S_ISREG(inode->i_mode)) {
        inode->i_op = &ext4_file_inode_operations;
        inode->i_fop = &ext4_file_operations;
} else if (S_ISDIR(inode->i_mode)) {
        inode->i_op = &ext4_dir_inode_operations;
        inode->i_fop = &ext4_dir_operations;
} else if (S_ISLNK(inode->i_mode)) {
        ...
}
```

- 首先 inode 是 文件 的 **唯一代言人**，而 文件 显然是 **具有不同类型** 的
    - data 文件都有一个称呼 **正则**
    - 此外，还有 **目录文件**，符号 **link文件等等**
        - 尽管文件的种类是多样的，但是组成是一样的
            - inode
            - data
- 最简单的理解，**文件** 与 **目录** ，都可能修改 inode 或 data，这些方法都是对应的函数指针，需要区别对待
    - 所以，ext4 文件系统不仅对 **inode** 与 **data** 的操作做区分，而且还对 **普通正则文件** 与 **目录文件** 以及 **符号连接文件** 做区分
    - 可以通过上面分类清晰的 **函数指针 感受下** 先
- 怪不得 **cat 打不开目录文件**，但是 vim 是可以打开目录的，**读取目录文件** 与 **读取正则文件** 的 函数是不一样的（**这些是与文件系统相关的**）
    - 在 ext4 中，并没有 **写目录文件** 这种单独操作

穿插一个 ext4 的 journal 机制 jbd2

- ref
    - https://bbs.huaweicloud.com/blogs/216911
        - 裸数据分析，fsdump 之类的方式直接去看 **裸journal**
    - http://www.cxyzjd.com/article/qq_32473685/103585546
    - ext4日志系统分析（一二）
        - CSDN 我又关注+收藏 ppingfann 这部分讲的很好
    - https://zhuanlan.zhihu.com/p/340990742
- 这个可以去类比 NOVA 的 **轻量级journal机制**
    - sb中保存有 **journal区** 的 **入队/出队** 指针
    - NOVA 的 journal 是 per-CPU 的，并发有优势
    - ext4 所使用的 就是 jbd2
        - 相比于 NOVA 的轻量级（只有一个block大小的 journal 区域），journal block device 这确实可以称之为一个系统
        - 不会按 core 来创建多个 journal，只有一个 journal，对并发的支持不是很好
            - 多 core，甚至多 fs 共享一个 journal 设备
    - jdb 基本看明白了，来总结一下 NOVA 吧，NOVA 基本套路与 jbd 差不多，非常核心的理解
- 所以 jbd2 是一个绝对的 **日志系统**
    - 是独立与文件系统的 **日志系统**
        - 并区别于 NOVA 的轻量级

    - 其源码位于 `fs/jbd2`
        - 阅读一下 kcongfig 可以 get 更多知识
            - jbd2 是一个 **面向块设备 通用** 的 **日志系统**
            - 现在有2个文件系统有采用
                - ext4 与 OCFS2
        - 无论是被编译为 **内建** 还是 **模块**，**init** 与 **exit** 就是 **主旋律**
- 关于 **日志系统** 数据结构的 **大小关系**
    - `journal > transaction > handle > buffer head`
        - 一个 journal 属于一个文件系统
        - 一个 journal 中包含若干个 **事务**
        - 一个 事务 中包含若干 handle
            - 一个 handle 对应于 **一个process**，一个事务中有多个 handle，来自多个 **process**
        - 一个 handle 中包含若干个 **buffer head**
            - 一个 handle 的本意确实 是一个 **文件系统接口所涉及的内容**
            - 但是由于 多 thread 并发。并发的 **文件系统调用** 会 **进入到一个 handle 中**

```
# dumpe2fs /dev/nvme0n1p2
# 部分结果，重点关注 journal 部分，说白了就是 sb 部分
Last mounted on:          /home/hwj/disk_mnt
Filesystem UUID:          1e051fd6-35da-4b7b-92e1-ee8e501ab5ef
Filesystem magic number:  0xEF53
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent flex_bg sparse_super large_file huge_file un
Inode count:              6553600
Block count:              26214400
Reserved block count:     1310720
Free blocks:              25611048
Free inodes:              6553587
First block:              0
Block size:               4096
# 时间戳
Filesystem created:       Sat Jul 17 11:32:24 2021
Last mount time:          Thu Jul 22 14:15:39 2021
Last write time:          Thu Jul 22 14:15:39 2021
...
First inode:              11  # 所有保留 inode 是不可用的，可用 inode 由 11 开始
Inode size:                   256
Journal inode:            8  # NOVA 果然是参考了 ext4的实现的，inode 8 指向了 jouranl 区域
Journal size:             128M  # 日志大小
Journal length:           32768  # 这个单位是 字节

Group 0: (Blocks 0-32767) [ITABLE_ZEROED]
  Checksum 0x0039, unused inodes 8180
  Primary superblock at 0, Group descriptors at 1-7
  Reserved GDT blocks at 8-1024
  Block bitmap at 1025 (+1025), Inode bitmap at 1041 (+1041)
  Inode table at 1057-1568 (+1057)  # inode 表的起始位置
  23513 free blocks, 8181 free inodes, 2 directories, 8180 unused inodes
  Free blocks: 9255-32767
  Free inodes: 12-8192

################################
# 日志基本信息
root@SGX-1604:/proc/fs/jbd2/nvme0n1p2-8# cat info
307 transactions (22 requested), each up to 8192 blocks
average:
  0ms waiting for transaction
  0ms request delay
  5600ms running transaction
  0ms transaction was being locked
  0ms flushing data (in ordered mode)
  4ms logging transaction
  154us average transaction commit time
  90078 handles per transaction
  6 blocks per transaction
  7 logged blocks per transaction
```

- 来检查一下 jbd2 的初始化过程
  - 实际上通常来说，管理块设备的是 文件系统，而 jdb2 作为日志系统管理 **一个块**（逻辑块，可能就是一个文件，也可以是一个逻辑分区，也可能是一个物理分区，甚至是log服务器集群）
    - 所以管理块设备的不仅有 **文件系统** ，还会有 **日志系统**，当然现在管理块设备的方式越来越多了，也不仅仅局限在这两点上了
  - 既然 jbd2 管理的是一个块设备，那么在设计上是与文件系统有相似之处的，比如 sb 的存在，**而 jbd2 区确实是有 sb 的存在的**
  - jbd2 模块 init
    - `BUILD_BUG_ON(sizeof(struct journal_superblock_s) != 1024);`
      - check 一下这个结构体 `journal_superblock_s` 对不对
      - 主要包含以下字段
        - 设备块的大小，比如 `1024/2048/4096` 等等
        - 日志文件 **块的总数**
        - **共享** 这个 log 的文件系统的数量
    - `ret = journal_init_caches();`
    - `jbd2_create_jbd_stats_proc_entry();`
      - 创建目录 `/proc/fs/jbd2` ，可以看一下 该目录 都包含哪些内容，有两个 **目录文件**
        - `nvme0n1p2-8` 与 `nvme1n1p2-8`
          - 表示日志保存在 nvme0n1p2 的 **inode号为8** 的 文件中，如上图所示
            - 内部会保存一些 **日志的基本信息**
          - 如果这里没有 inode 号，则 文件系统 用一个 **分区** 来作为日志

        - 这个显然是 **做文件系统** 的时候确定的
- ext4 的 journal 区域 是在 mke2fs 的时候确定的，会专门分配这么一片区域，以文件的形式
    - something like `mke2fs -J device`
    - 上面也提到了，可以是 **文件/分区/单独的盘/节点/分区** 的形式

```
// 日志相关 的 关键数据结构

// journal_t/journal_s 在概念上是最大的，代表的是 journal 文件/分区，理解为 mgr
// 每一个文件系统对应一个
/**
 * typedef journal_t - The journal_t maintains all of the journaling state information for a single filesystem.
 *
 * journal_t is linked to from the fs superblock structure.
 *
 * We use the journal_t to keep track of all outstanding transaction
 * activity on the filesystem, and to manage the state of the log
 * writing process.
 *
 * journal_s 这个是一个文件系统实例所拥有的，从 blocksize 与 blk_offset 这两个字段就可以看的出来一个文件系统的 journal
 * 在 设备上逻辑上是连续的，其实对于 disk，就可以认为在物理上也是连续的了，所以 一个 journal 文件/分区 可以被多文件系统共享 不对
 **/
typedef struct journal_s        journal_t;
struct journal_s
{
        ...
        int                     j_blocksize;  // Block size for the location where we store the journal
        unsigned long long      j_blk_offset;  // Starting block offset into the device where we store the journal.

    // 一个 journal 只能有一个运行着的事务
    transaction_t               *j_running_transaction;

    // 只有一个正在提交的事务
    transaction_t               *j_committing_transaction;

    // 等待 checkpoint 的事务可以比较多
    transaction_t               *j_checkpoint_transactions;

        // head 与 tail 就是入队出队指针
    // identifies the first unused block in the journal
        unsigned long       head;
    // identifies the oldest still-used block in the journal
        unsigned long       tail;
    // 空闲块的数量
    unsigned long               j_free;
}
```

- 看一下 ext4 在 **挂载的初始化阶段** 是如何 加载 log 的，这里要注意 jbd2 是作为一个单独的系统被 load 到 kernel 中的
- `ext4_mount`
    - `ext4_fill_super`
        - `err = ext4_load_journal(sb, es, journal_devnum);`
            - `journal_t *journal;`
                - 这个是概念上最大的数据结构，**代表的是整个 journal**
            - `if (journal_inum) {` —— inode=8 是存在的，由文件作为 journal。**重点分析一下这个**
                - `struct inode *journal_inode;`
                - `journal = ext4_get_journal(sb, journal_inum);`
                    - `journal_inode = ext4_get_journal_inode(sb, journal_inum);`
                        - 即 8号inode 内存中对应的 vfs_inode
                    - `journal = jbd2_journal_init_inode(journal_inode);` —— **jbd2_journal_init_inode creates a journal which maps an on-disk inode as the journal**
                        - `journal = journal_init_common(inode->i_sb->s_bdev, inode->i_sb->s_bdev, blocknr, inode->i_size >> inode->i_sb->s_bloc`
                            - **说白了这里是起一个 mgr**
                        - `jbd2_stats_proc_init(journal);`
                            - `proc_create_data("info", S_IRUGO, journal->j_proc_entry, &jbd2_info_proc_ops, journal);`
                                - 文件 **/proc/fs/jbd2/nvme0n1p2-8/info** 的由来
            - `} else {` —— 由单独的分区，即 dev 作为 journal
                - `journal = ext4_get_dev_journal(sb, journal_dev);`
            - `}`

        - `err = jbd2_journal_load(journal);` —— **Read journal from disk, journal 本身就是存盘的**
            - `journal_superblock_t *sb;`
            - `err = load_superblock(journal);`
            - `sb = journal->j_superblock;`
            - `jbd2_journal_recover(journal)`
                - 尝试恢复一下出问题的 log
                - 正常关机情况下，这里都是干净的
- 到此为止，**journal mgr** 就到位了
    - **它代表了 journal 那片区域**

```
/**
 * typedef handle_t - The handle_t type represents a single atomic update being performed by some process
 *
 * All filesystem modifications made by the process go through this handle.
 * Recursive operations (such as quota operations) are gathered into a single update.
 *
 **/
// 关于 handle，简单来说，是用来 隔离 process 的。每一个 process 对 文件系统的修改 是由该数据结构来表示的
// 这里确定 handle 对应的是 process，即进程
typedef struct jbd2_journal_handle handle_t;    /* Atomic operation type */

// 分界线
typedef struct transaction_s    transaction_t;  /* Compound (复合)  transaction type */
{
        ...
        /* Pointer to the journal for this transaction. [no locking] */
        journal_t               *t_journal;  // 事务是属于 journal 的
        /* Sequence number for this transaction [no locking] */
        tid_t                   t_tid;  // 事务的序列号
        /* Transaction's current state */
        enum {
                T_RUNNING,
                T_LOCKED,
                ...
                T_FINISHED
        } t_state;

        unsigned int    t_log_start;            // log中本transaction_t从日志中哪个块开始
        int                             t_nr_buffers;           // 本transaction_t中缓冲区的个数
        struct journal_head     *t_buffers;     // Doubly-linked circular list of all metadata buffers owned by this transaction
        struct journal_head     *t_shadow_list; // Doubly-linked circular list of metadata buffers being shadowed by log IO
        ...
}
```

- handle && transaction
    - JBD 将一组 handle 打包为一个事务（transaction），并将事务一次写入日志
        - 每一个 handle 属于一个 task，一个事务可能会包含若干个 handle，它们来自不同的 task，所以这里是没有 **按core去更好的支持并发的**
            - 一个 handle 可能涉及多个 **缓冲区bh** 的修改
                - 但是 handle 这个数据结构本身不包含 buffer head
                    - 事务与handle相互关联
                        - handle 不包含 bh，**主要目的是顺着它找到 事务**
    - JBD 保障事务本身是原子性的。这样，作为事务的组成部分的 handle 们自然也是原子性的
        - 事务管理 buffer head
        - 事务是有 list 来管理所有在这个事务内 dirty 的 bh 的
    - 事务提交的时候才会把 dirty buffer **持久化** 到 journal 中
- 相比于数据元数据更加重要，**元数据直接决定文件系统的一致性**
    - 代表一个原子操作，文件系统 的接口在 os 看来都是一个原子的操作

mkdir in ext4

```
/double_D/mnt/ext4point # mkdir test
# 所以说，mkdir 调用的是父目录的 mkdir 方法，这个是目录文件的 inode 方法，目录文件的 data 方法基本只有一个 read
# 父目录的 inode+data 都要 modify
# 子目录的 inode+data 都需要 new，子目录 有两个默认的 data .与..
```

- `fs/namei.c:SYSCALL_DEFINE2(mkdir, const char __user *, pathname, umode_t, mode)`
  - `do_mkdirat(AT_FDCWD, pathname, mode)` —— pathname is "test"
    - `dentry = user_path_create(dfd, pathname, &path, lookup_flags);`
      - pathname 仅仅是 test/path是一个空的dentry对象（**主要内容是dentry**）,**它代表的其实是父目录**
      - 创建 test 的 dentry，这个 test 显然是要在文件 tree 中的
      - 在这一步，dentry的父子关系就已经建立了
        - `dentry->d_parent` 已经到位了
    - `error = vfs_mkdir(path.dentry->d_inode, dentry, mode);`
      - `path.dentry->d_inode` 这个是 `/double_D/mnt/ext4point` 的 inode
      - dentry 是新目录文件的 dentry，即代表 test 的 entry
      - `error = dir->i_op->mkdir(dir, dentry, mode);`
        - 调用具体文件系的方法，即 mkdir
- `ext4_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)`
  - `inode = ext4_new_inode_start_handle(dir, S_IFDIR | mode, &dentry->d_name, 0, NULL, EXT4_HT_DIR, credits);` —— **怪不得不理解，最开始把这一步跳过去了，回来认真重看**
    - `struct inode *__ext4_new_inode(handle_t *handle, struct inode *dir, umode_t mode, const struct qstr *qstr, __u32 goal, uid_t *owner, _`
      - `inode = new_inode(sb);`
        - 这里分配的是内存 inode
        - 在gdb例子中
          - `inode->i_ino=12`
      - `inode_init_owner(inode, dir, mode);`
        - 初始化 inode 的归属
      - `handle = __ext4_journal_start_sb(dir->i_sb, line_no, handle_type, nblocks, 0, ext4_trans_default_revoke_credits(sb));`
        - tips
          - **handle的初衷** 看样子是 os 所看到的文件系统的 **接口的原子性**
          - 尽管 handle 可能已经存在，但是任何一个由 **文件系统标准接口** 进入的 op 都应该 **尝试创建自己的handle**
            - **在创建过程中，如果已经存在，则直接返回**
        - `journal_t *journal;` in `fs/ext4/ext4_jbd2.c`
        - `journal = EXT4_SB(sb)->s_journal;` in `fs/ext4/ext4_jbd2.c`
          - **journal 挂在文件系统的 sb 上**
        - `jbd2__journal_start(journal, blocks, rsv_blocks, revoke_creds, GFP_NOFS, type, line);` in `fs/jbd2/transaction.c`
          - **jbd2__journal_start**
          - `handle_t *handle = journal_current_handle();`
            - **当前 task 可能创建过 handle**
          - `if (handle) { handle->h_ref++; return handle; }`
            - 返回一个已经可以用的 handle、如果 handle 不可用，则需要 `new && init`
              - 就是下面的这一套操作
          - `handle = new_handle(nblocks);`
            - `handle_t *handle = jbd2_alloc_handle(GFP_NOFS);`
              - slab 的内存分配
            - `handle->h_total_credits = nblocks;`
              - 可能涉及到的 buffer head（**文件系统块**）的 **个数**
                - 该原子操作预期将修改 nblocks 个缓冲区
              - Number of remaining buffers we are allowed to add to
                - **没有针对 nblocks 的分配操作**
            - `handle->h_ref = 1;`
          - `err = start_this_handle(journal, handle, gfp_mask);`
            - `transaction_t *transaction, *new_transaction = NULL;`
            - `int blocks = handle->h_total_credits;`
            - `if (!journal->j_running_transaction)`
              - `new_transaction = kmem_cache_zalloc(transaction_cache, gfp_mask);`

        - 为当前的 handle 关联一个 running 的事务，这个事务是属于 journal 的，而 journal 是属于一个 文件系统的，所以 **journal 不会被多文件系统实例共享**
    - `transaction = journal->j_running_transaction;`
        - 最终得到一个 running 的事务，在一个 journal 中的某时刻只能有一个 running 的 transaction
    - `handle->h_transaction = transaction;`
        - handle 关联当前 running 事务
        - handle 需要通过 transaction 找到 bh
            - bh 在事务中记录
    - `handle->h_requested_credits = blocks;`
    - `current->journal_info = handle;`
        - 当前 handle 被赋值，每 task 一个 handle
- `return handle;`
- **关键分隔符**
    - 这里得到一个与 running transaction 所关联的 handle
    - 我们假设这个 mkdir 是这个 ext4 唯一遇到的 **第一个也是唯一一个** 系统调用
        - 则 handle 与 transaction 均为 `new && init`
- `err = ext4_handle_dirty_metadata(handle, NULL, inode_bitmap_bh);` —— **这里是非常关键的需要理解的地方**
    - tips
        - 因为新分配了一个 inode，所以 bitmap 所在的 buffer head 也需要 dirty
            - 这是第一个脏的 buffer head，即将 **挂到事务** 上去
        - 在某种配置下，**糟蹋的第一个块**
    - `if (ext4_handle_valid(handle))`
        - `err = jbd2_journal_dirty_metadata(handle, bh);` —— **mark a buffer as containing dirty metadata**
            - tips
                - 有一个 buffer head is dirty，需要被 journaled 到当前 事务中，但是它是带着 handle 来的
                - jdb2 必须拥有对 buffer head 的写权限
                - buffer head 是挂在 transaction 上的
                    - 所以 handle 没有 bh，handle 能够找到 transaction，然后 transaction 管理 buffer head
                - **dirty journal head 挂到了当前的 transaction 的 list 中**
            - `transaction_t *transaction = handle->h_transaction;`
                - 目标事务
            - `jh = bh2jh(bh);`
                - 一个 journal head 对应一个 buffer head，实际上 transaction 管理的是 journal head
            - `__jbd2_journal_file_buffer(jh, transaction, BJ_Metadata);` —— **指定这里修改的是元数据 BJ_Metadata**
                - `jh->b_transaction = transaction;`
                - `switch (jlist)`
                    - `case BJ_Metadata`
                        - `transaction->t_nr_buffers++;`
                        - `list = &transaction->t_buffers;`
                            - **记着**，这些被修改的 **元数据 是挂在 t_buffers 上的**
                - `__blist_add_buffer(list, jh);`
                    - Append a buffer to a transaction list, given the transaction's list head
        - **说白了只是当前事务 多了一个要管理的 journal head**，目前全部都还在内存中
            - **只是一个 add to list 的操作**
            - **提交** 的时候这个 journal head 才会被 **持久化到 journal 中**
    - `else`
        - `if (inode) mark_buffer_dirty_inode(bh, inode); else mark_buffer_dirty(bh);`
            - 这里 **只能** 起到一个 **mark** 的作用
        - `if (inode && inode_needs_sync(inode))`
            - `sync_dirty_buffer(bh);`
                - `return __sync_dirty_buffer(bh, REQ_SYNC);` —— **REQ_SYNC** 需要同步
                - lock_buffer(bh);
                - `ret = submit_bh(REQ_OP_WRITE, op_flags, bh);`
                    - 只有这样的一个 submit 下去是无法保证 函数返回时 bh 已经持久化到 disk 的
                        - **所以才加了下面的内容**
                - `wait_on_buffer(bh);`
                    - `might_sleep();`

- `if (buffer_locked(bh)) { __wait_on_buffer(bh) };`
  - `wait_on_bit_io(&bh->b_state, BH_Lock, TASK_UNINTERRUPTIBLE);`
    - tips
      - Block until a buffer comes unlocked
      - wait for a bit to be cleared
      - 参数
        - &bh->b_state: the word being waited on, a kernel virtual address
        - BH_Lock: the bit of the word being waited on
        - TASK_UNINTERRUPTIBLE: the task state to sleep in
          - **不可被中断**
    - `return out_of_line_wait_on_bit(word, bit, bit_wait_io, mode);`
      - `return __wait_on_bit(wq_head, &wq_entry, action, mode);`
        - `do {`
          - `test`
        - `} while(...)`
      - 最下面是 **死循环的轮询** 等待 **指定内存** 被修改
        - 核心是 **wait_on_bit**
        - DMA设备会完成数据的传输，storage中的数据被DMA传输到内存后，storage会发中断给CPU
          - 在对应的中断handler中会set bit，在轮询的task看到之后就可以返回了，因为数据已经到位了
- 这里说白了就是在做 sync，这里真的需要等待 返回，**即确保数据一定已经落盘**
  - 就是 while + poll
- `err = ext4_handle_dirty_metadata(handle, NULL, group_desc_bh);`
  - 在某种配置下，**糟蹋的第二个块**
  - group_desc_bh 这个不知道是哪个 buffer head，但是只要知道它脏了就可以了
- `ei = EXT4_I(inode);`
- `inode->i_ino = ino + group * EXT4_INODES_PER_GROUP(sb);`
- `ei->i_disksize = 0;`
  - 这里在内存里，对 **vfs_inode** 以及 **raw inode** 一顿 dirty，反正它们已经在 transaction 上挂上号了
    - **想怎么折腾怎么折腾了**
- `err = ext4_mark_inode_dirty(handle, inode);`
  - `err = ext4_mark_iloc_dirty(handle, inode, &iloc);`
    - `err = ext4_do_update_inode(handle, inode, iloc);`
      - `rc = ext4_handle_dirty_metadata(handle, NULL, bh);`
        - **糟蹋的第三个块**
          - 这个就是新 inode 本身

- **关键分隔符**
  - ****
  - ****
  - ****
  - ****
  - ****
  - 在某选项下，至少三个 journal head 被挂到到了 transaction 上
    - `ext4_handle_dirty_metadata(handle, NULL, inode_bitmap_bh);`
      - **inode_bitmap_bh**
    - `ext4_handle_dirty_metadata(handle, NULL, group_desc_bh);`
      - **group_desc_bh**
    - `ext4_handle_dirty_metadata(handle, NULL, bh);`
      - **bh**
        - 这个 bh 就是代表着新的 inode 所在的块
  - ****
  - ****
  - ****
- `handle = ext4_journal_current_handle();`
- `inode->i_op = &ext4_dir_inode_operations;`
  - test 目录的 inode 操作方法，与父目录保持一致

- `inode->i_fop = &ext4_dir_operations;`
  - test 目录的 data 的操作方法，与父目录保持一致
- `err = ext4_init_new_dir(handle, dir, inode);`
  - dir 是 `/double_D/mnt/ext4point` 的 inode，inode 是 `test` 的 inode
  - `dir_block = ext4_append(handle, inode, &block);`
    - `bh = ext4_bread(handle, inode, *block, EXT4_GET_BLOCKS_CREATE);`
      - `bh` 是 **一个块** 在内存中的映射，即一个 page 中有若干个 `buffer_head`
      - `ext4_bread means block read`
      - `bh = ext4_getblk(handle, inode, block, map_flags);`
        - `err = ext4_map_blocks(handle, inode, &map, map_flags);`
          - 或 map 或 分配，一定会分配磁盘上的一个块
      - 这个函数返回之后，**逻辑块号就到位了**，即资源分配到位了
    - 总之， `dir_block` 是新分配的一个磁盘块。block 是 **逻辑块号**（**需要被返回**）
      - 为 test 的 data 部分分配空间，最小操作单位是块，因为 `.`与`..` 的存在，只要 new 的目录文件立刻要分配 空间
    - 到此为止，test 的 **元数据+数据** 都到位了，都在内存中
  - `err = ext4_handle_dirty_dirblock(handle, inode, dir_block);`
    - `return ext4_handle_dirty_metadata(handle, inode, bh);`
      - **糟蹋的第四个块**
- **手动分隔符**
  - `****`
  - 到此为止，对于新目录，**元数据与数据** 的修改现在都在内存中 buffer head
    - 此外，它们都已经 挂到了 **当前** 的 **事务** 中
  - `****`
- `err = ext4_add_entry(handle, dentry, inode);`
  - adds a **file entry** to the **specified directory**
  - `dentry->d_parent` 早已到位了，所以根据 parent dentry 能够直接找到 父目录（ `/double_D/mnt/ext4point` ）的 inode
    - 但是这个 link 关系是不存盘的，关键还是要在 目录文件 `/double_D/mnt/ext4point` 中写入 test 项
      - 即 **file entry** `{test inode: "test"}`
  - `bh = ext4_bread(handle, dir, block, EXT4_GET_BLOCKS_CREATE);`
    - bh 是父目录（ `/double_D/mnt/ext4point` ）所在的数据块在内存中的映射
  - `retval = add_dirent_to_buf(handle, &fname, dir, inode, NULL, bh);` —— **Add a new entry into a directory (leaf) block**
    - `ext4_find_dest_de(dir, inode, bh, bh->b_data, blocksize - csum_size, fname, &de);`
      - 在 page buffer 的 buffer head 中找到 new entry 应该写入的位置
    - `ext4_insert_dentry(inode, de, blocksize, fname);`
      - 将 new entry 的对应 value 写入 buffer head 中
    - `dir->i_mtime = dir->i_ctime = current_time(dir);`
    - `err = ext4_handle_dirty_dirblock(handle, dir, bh);`
      - `return ext4_handle_dirty_metadata(handle, inode, bh);`
        - 这个很可能在前面已经 dirty 过了，即已经在 当前事务的 mgr 之上了
- `err = ext4_mark_inode_dirty(handle, dir);` —— 最后是父目录的 inode 所在的块
  - ...
    - `rc = ext4_handle_dirty_metadata(handle, NULL, bh);`
      - 父目录 inode 所在的 buffer head 也被当前事务所管理了
- `****`
- `****`
- `****`
- 在 dirty 并 add 一个 buffer head 到当前事务的过程中
  - 上述过程可能有若干的重复工作，但是最终在 事务 看来
    - `t_buffers` 所管理的显然是不会重复的
  - mkdir 整个这个过程都是只涉及 metadata 的操作
    - 只有 file data 才算是 data，其余均为 metadata，包括目录自己
      - 在 **树形结构** 的文件系统中，确实只有最底层的叶子节点上的那个才是 **数据**
        - 中间节点均为 **元数据**
- `****`
- `****`
- `****`
- `if (handle)`

- `ext4_journal_stop(handle);`
  - `tid_t tid;`
  - `tid = transaction->t_tid;`
    - 这个代表的是当前事务的序列号，即 sequence num
  - `rc = jbd2_journal_stop(handle);`
    - tips
      - complete a transaction
      - **All done for a particular handle**
      - `handle->h_sync` 的 **赋值**
        - `if (file->f_flags & O_SYNC)`
          - `handle->h_sync = 1;`
        - `if (IS_SYNC(inode))`
          - `handle->h_sync = 1;`
        - `if (IS_DIRSYNC(inode))`
          - `handle->h_sync = 1;`
      - **jbd2_journal_stop**
    - `transaction_t *transaction = handle->h_transaction;`
    - `pid = current->pid;`
    - `if (handle->h_sync && journal->j_last_sync_writer != pid && journal->j_max_batch_time)`
      - tips
        - 这段代码实现的是 **同步事务的 batching**
          - If the handle was synchronous, **don't force a commit immediately**. Let's **yield** and let another thread piggyback(**肩抗式运输**) onto this transaction
      - `set_current_state(TASK_UNINTERRUPTIBLE);`
        - 当前 task 的状态变成 **不可中断**
      - `schedule_hrtimeout(&expires, HRTIMER_MODE_ABS);`
        - 准备 **sleep一段时间** 等待 其它线程 append 内容到 transaction 中
    - 同步 **事务提交task** 被 **唤醒** 准备继续操作
    - `if (handle->h_sync)`
      - `transaction->t_synchronous_commit = 1;`
    - `if (handle->h_sync || time_after_eq(jiffies, transaction->t_expires))`
      - **h_sync** 或 一个事务 **足够old** 也都要 force commit 了
      - `jbd2_log_start_commit(journal, tid);`
        - `ret = __jbd2_log_start_commit(journal, tid);`
          - tips
            - Allocation code for the journal file. Manage the space left in the journal, so that we can begin checkpointing when appropriate
              - **事务提交的时候才会需要 journal file 的 space**
              - 但是 **提交事务的** 依然是 **另一个thread**
            - 这里应该只是说明在 **jouranl中 有可以提交的事务了**
            - **目录操作并不是要触发 transaction 的 commit，或者说在某种少见的条件满足时，才碰巧在这里完成事务的提交**
              - 这里的函数调用栈有做 **简化**，**这里并不会主动触发事务的提交，在大多数情况下**
          - `journal->j_commit_request = tid;`
            - 申请提交事务的事务 id
          - `wake_up(&journal->j_wait_commit);`
            - 唤醒阻塞在 j_wait_commit 等待队列上的 **内核线程kjournald**
              - 说白了就是 **事务提交线程**
    - `stop_this_handle(handle);`
      - finish
    - `jbd2_free_handle(handle);`
      - finish

◦ 到此为止，这个 thread 的工作就结束了。可以看到在这个过程中，**全部的修改均在内存中**
  - 该过程中没有 **同步持久化** 的过程
  - 目录操作结束后的 **journal_stop** 不会主动触发 **事务的提交**，但是会 在 **journal_stop** 的过程中检查是否满足提交的条件
    - **journal_stop** 的主要目的是 **断开 handle 与 transaction 的关联**

简述一下基本过程

- 创建一个 **目录或文件** 都是 **在 dentry tree 上 insert 一个节点**，以 **目录文件** 为例
  - get handle
    - if not 创建 handle
    - handle 找不到 buffer head
  - get transaction
    - if not 创建 transaction
      - transaction 能够找到 buffer head
    - 找到在当前 journal 上 running 的事务
  - **handle 只是为了与 transaction 相关联**
  - 先为新成员 **分配资源**，以下两个资源其实都需要 **写或更新磁盘**
    1. 一个新的 inode
       - hold 新目录的 元数据，inode number 等等
    2. 一个新的 block
       - hold 新目录的 `./..`，关键是记录 `.目录` 与 `..目录` 的 inode 号
    3. inode bitmap 所在的 block
  - 然后要 **修改父目录** 的 **元数据**
    1. inode
       - 父目录的数据都变了，元数据显然也要改变
         - 大小
         - 访问时间
         - ...
    2. dentry（**目录项算是元数据**）
       - 需要将 **新目录的entry** append 下来，包括 inode 号与目录名（test）等
  - 其中每 dirty 一个 buffer head 都对应了以下的一个操作
    - `ext4_handle_dirty_metadata`
      - journal head(buffer head) 挂到 transaction 的 list 上
  - 最后 stop handle 并尝试提交 running 事务
  - 到此结束，所有操作都在内存中
    - 事务的提交以及 bio 的提交，是一个内核线程

简单看一下 wirte 简单比对一下

- ...
  - 前面的部分均未涉及 metadata 的修改，包括 **数据的修改** 也是 **没有的**
- `ext4_buffered_write_iter(iocb, from);`
  - `if(err)`
    - `ret = generic_write_sync(iocb, ret);` —— **Sync the bytes written** if this was a **synchronous write**
      - `int ret = vfs_fsync_range(iocb->ki_filp, iocb->ki_pos - count, iocb->ki_pos - 1, (iocb->ki_flags & IOCB_SYNC) ? 0 : 1);` —— 如果 IOCB_SYNC是真的，那么这个就是0，那么就要把 **数据&元数据** 全部 sync 下去
        - tips
          - helper to sync a range of **data & metadata** to disk
          - 参数
            - file: file to sync
            - start: offset in bytes of the beginning of data range to sync
            - end: offset in bytes of the end of data range (inclusive)
            - datasync: perform only datasync
          - **fsync系统调用** 走的也是这条路
        - `if (!datasync && (inode->i_state & I_DIRTY_TIME))` —— **需要 sync metadata，简单理解这里调用 fsync 而非 fdatasync**
          - `mark_inode_dirty_sync(inode);`
            - `__mark_inode_dirty(inode, I_DIRTY_SYNC);`
              - **Put the inode on the super block's dirty list**
                - 所以脏的 inode 也不全是在 journal 上管理着的，这里由超级快来维护 list 管理

- `file->f_op->fsync(file, start, end, datasync);`
  - `ext4_sync_file(struct file *file, loff_t start, loff_t end, int datasync)`
    - `ret = file_write_and_wait_range(file, start, end);` —— **write out & wait on a file range，sync data 的关键**
      - `struct address_space *mapping = file->f_mapping;`
      - `err = __filemap_fdatawrite_range(mapping, lstart, lend, WB_SYNC_ALL);` —— **start writeback on mapping dirty pages in range**
        - `ret = do_writepages(mapping, &wbc);`
          - `while (1)`
            - `if (mapping->a_ops->writepages)`
              - `ret = mapping->a_ops->writepages(mapping, wbc);`
            - `else`
              - `ret = generic_writepages(mapping, wbc);`
            - 以上两个命令实际做的就是 submit bio，但是不论 wbc 中的 mode 是什么，都无法改变 一个事实
              - **对应函数返回时，数据还未达到 storage**
            - `might_sleep();`
              - 对于慢速设备，实际数据 IO 确实是需要 sleep 的
            - `cond_resched();`
              - `schedule();`
              - **这玩意会主动让出CPU**
            - `congestion_wait(BLK_RW_ASYNC, HZ/50);`
              - 这个是用来处理**拥塞的**
                - congestion means 拥塞
                - 如果CPU下发的IO太多了，**那么需要处理一下拥塞**
      - **所以这个函数结束之后数据是无法到达 storage 的**
      - `__filemap_fdatawait_range(mapping, lstart, lend);`
        - `while (index <= end)`
          - `for (i = 0; i < nr_pages; i++)`
            - `wait_on_page_writeback(page);`
              - `wait_on_page_bit(page, PG_writeback);`
                - 等待内存中的某一个位置的标志被修改（**被IO完成的中断修改**），这些位置不是 page 的，就是 buffer head 的
    - 在函数 fdatawait 返回后，数据算是真的完成持久化
      - 这里 while 确实就是在test内存中一个bit的变化
        - **同步设备，基本就一个方法，那就是 poll**
  - *****
  - *****
  - 前面已经 sync 了 data，接下来准备 sync metadata
    - The caller's `filemap_fdatawrite()/wait` will **sync the data**
    - Metadata is in the journal, **we wait for proper transaction to commit here**
    - 下面是典型的 **3种粒度** 的 **一致性保证**
  - *****
  - *****
  - `if (!sbi->s_journal)`
    - **ext4 没有开 journal，根本没有事务那一说**
    - `ret = ext4_fsync_nojournal(inode, datasync, &needs_barrier);`
      - ret = sync_mapping_buffers(inode->i_mapping);
        - Starts I/O against the buffers at **mapping->private_list**, and waits upon that I/O
        - 这里显然不是 metadata 部分
      - `if (!(inode->i_state & I_DIRTY_ALL)) return ret;`
      - `if (datasync && !(inode->i_state & I_DIRTY_DATASYNC)) return ret;`
        - 对应系统调用 `fdatasync()`， sync data 而不 sync metadata
      - 如果执行到这里了，说明 **需要 sync metadata**
      - `err = sync_inode_metadata(inode, 1);` —— **write an inode to disk**
        - `sync_inode(inode, &wbc);`
          - `writeback_single_inode(inode, wbc);` — Write out an inode's dirty pages
            - `ret = __writeback_single_inode(inode, wbc);`
              - `ret = do_writepages(mapping, wbc);`

- `int err = filemap_fdatawait(mapping);`
- **wait on bit**
- `inode_sync_complete(inode);`
- `wake_up_bit(&inode->i_state, __I_SYNC);`
- **唤醒等待在bit上的task**
- 在没有journal的情况下，到此为止，**data + metadata 均已落盘**
- `else if (ext4_should_journal_data(inode))`
- **ext4 在 journal 中 也写了data**
- 那么前面的操作 **filemap_fdatawrite do nothing**
- **data + metadata 都通过 force commit sync**
- `ret = ext4_force_commit(inode->i_sb);` —— **Force the running and committing transactions to commit, and wait on the commit**
- `journal = EXT4_SB(sb)->s_journal;`
- `ext4_journal_force_commit(journal);`
- `jbd2_journal_force_commit(journal);`
- `ret = __jbd2_journal_force_commit(journal);`
- `jbd2_log_start_commit(journal, tid);`
- `ret = jbd2_log_wait_commit(journal, tid);`
- 同样的，函数返回意味着 **data + metadat 落盘完成**
- `else`
- **最通常的情况，ext4 开启了 journal，且 journal 中只有 metadata**
- data 通过 filemap_fdatawrite 落盘，而 metadata 通过 commit journal 落盘
- `ret = ext4_fsync_journal(inode, datasync, &needs_barrier);`
- `ext4_fc_commit(journal, commit_tid);` —— **Performs a fast commit for transaction, fc means fast commit**
- `ret = jbd2_fc_begin_commit(journal, commit_tid);`
- `ret = ext4_fc_perform_commit(journal);`
- `list_for_each(pos, &sbi->s_fc_q[FC_Q_MAIN])`
- `iter = list_entry(pos, struct ext4_inode_info, i_fc_list);`
- `inode = &iter->vfs_inode;`
- `ret = ext4_fc_write_inode_data(inode, &crc);`
- `ret = ext4_fc_write_inode(inode, &crc);`
- 这里也体现了一个思想，**data 比 metadata 先持久化**
- 反正就是一顿 **落盘** 操作
- `jbd2_fc_end_commit(journal);`
- **反正一直等待落盘后才返回**
- 总结一下同步操作
- 就是要硬硬的等 IO 全部完成之后再返回，**data+metadata 都会持久化finish**

- `else`
- `ret = generic_perform_write(iocb->ki_filp, from, iocb->ki_pos);`
- `struct address_space *mapping = file->f_mapping;`
- `write_begin`
- `copy`
- `write_end`
- tips
- 数据写完之后要尝试去修改 metadata 即 inode
- `handle_t *handle = ext4_journal_current_handle();`
- 找到当前的 handle 并且绑定 handle 到一个 running transaction
- `ret = ext4_mark_inode_dirty(handle, inode);`
- 这个 inode dirty 会被记录到 journal 的 事务中
- 看到这里就明白为什么 **事务提交破事那么多了**
- **正常情况下，journal 中需要 持久化的是 metadata，但是其关联的 data 也必须先于 metadata 持久化下去**
- 写数据到 page cache 就直接返回了
- 同步的原因就是确实是同步的
- 对于这个 app 来说，同步等待写完成后返回，这里认为写到 page cache 即意味着写完成

事务提交线程

- `static int kjournald2(void *arg)` in `linux/fs/jbd2/journal.c`
  - tips
    - The main thread function used to manage a logging device journal
    - This kernel thread is responsible for two things
      - COMMIT
        - The journal thread is responsible for writing all of the metadata buffers to disk
      - CHECKPOINT
        - We cannot reuse a used section of the log file until all of the data in that part of the log has been rewritten elsewhere on the disk
        - Flushing these old buffers to reclaim space in the log is known as checkpointing, and this thread is responsible for that job
    - kjournald2
      - 上面的说法比较官方，之前理解有问题，事务提交线程 与 flsuh data 到 storage 的线程我给搞混了，以为有两个。但是其实 **这是一个线程**
      - ext4 数据的持久化所依赖的就是 **事务的提交**
        - 要么是 **定时发生**，要门是某些操作结束之后 **主动触发** 事务的提交
  - `timer_setup(&journal->j_commit_timer, commit_timeout, 0);`
    - 要么被想要提交的事务唤醒，要么定时唤醒
  - loop
    - `if (journal->j_commit_sequence != journal->j_commit_request)`
      - tips
        - tid_t j_commit_sequence;
          - Sequence number of the most recently committed transaction
        - tid_t j_commit_request;
          - Sequence number of the most recent transaction wanting commit
        - **一股脑提交一下**
      - `jbd2_journal_commit_transaction(journal);` —— **真正的事务提交部分，比较复杂，单独来说一下还是要**

jbd2_journal_commit_transaction

- 当前状态
  - `touch txt && echo "123" > txt`
    - 文件操作
      - 分配一个磁盘块，修改 bitmap，**bitmap 所在的 block 属于这个操作的元数据**
        - 有一个元数据块 **即将** 变脏
      - 数据写到 page cache 中
        - 暂时不涉及元数据的改变
        - data 位于 page cache，很干净，**没有任何标记行为**
          - 是通过挂在 **事务上的 脏的** inode 去找到这部分 data 的
      - 写操作完成之后，数据由 **用户态缓存 copy 到内核中**
        - bitmap所在的块 与 inode所在的块 均为 metadata，这两个块都会被 **按序** add 到事务的 list 中
          - 它俩的持久化 **一定是一个整体**，这个如果没有事务是不可能实现的
          - 所以再次强调，事务保证的是 metadata 的一致性
            - 而 metadata 的修改也满足 ACID
              - A: 如果有不完整的事务，就全部 recovery 掉以保证原子性，但是这个是不是要记录 old value
              - C: 其它三点来保证
              - I: 由锁机制隔离
              - D: 实际位置的 metadata 可以不用 fsync 死等，可以给 block layer 空间来调度
                - 如果 crash，journal 中的 metadata 的备份（new value）能够保证 持久性
                  - redo log
        - metadata 也位于 page cache，**对应块** 也已经被 **标记为脏**，并且被挂到了 **事务的队列中**
          - 这个 **metadata** 可能包括 **inode 所在的块，bitmap 所在的块，目录文件所在的块等等**
  - `mkdir test`
    - 目录操作，操作涉及的所有数据块均为 metadata

        - 会按照顺序将需要修改的 metadata 所在的 buffer head 挂在 transaction 上
        - 保守估计 6 个 buffer head 会 dirty
    - **最后会触发事务的提交**
  - **所有数据都在内存中目前**
    - data: 单纯的位于 page cache 中
      - **不被事务管理**
    - metadata: 同样位于 page cache 中
      - **被事务管理**
        - 这个被事务管理的 inode 能够找到 上面 不被事务所管理的 buffer head（**for data**）
  - 按照这个思路去理一下 ext4 的事务提交过程。其实也没有那么复杂
    - **事务提交** 或 **sync的文件系统调用** 意味着 **data+metadata 的持久化**
    - 首先，文件系统对外暴露的接口都是 **原子的**，都是能够保证 **一致性** 的
    - 假设两个 **并发线程** 都在执行 **大量** 的 `mkdir` 以及 `create && write`，假设只有一个 core
      - journal 中的一个 transaction 内部的 handle 可能是来自于不同的**进程**的
      - 因为要保证 handle 的原子性，所以 并发写 handle 到 transaction 是需要 lock 的
        - `|handle1|handle2|handle3|...|handle1|handle2|`
        - jdb 日志系统 保证一个 transaction 的一致性，所以 handle 是在事务内部的，handle 的 **一致性** 自然被保证
    - 然后一顿操作之后，metadata 的脏数据被 挂在了 transaction 上，纯粹的数据现在仅仅是位于 page cache 中
    - 某时刻，事务提交被触发
      - 5s 的 flush 周期到了
      - 上层的 sync 操作 触发了 commit
        - 现在假设其中一个 thread 执行了 fsync 操作，准备提交事务的时候，其实是会先 sleep 一下，然后等待 其它线程继续向该 事务 中 append dirty metadata 的
        - 对应 batch 处理，是用来提高 **多线程** 的性能的
    - 现在准备提交事务了
      - 数据先 **同步阻塞** 落盘
        - `submit_bio` 操作无法保证阻塞同步的等待
        - 实际上 `submit_bio` 这类函数执行完毕之后后面都会加一个 while 循环去 poll 内存中某一个位置的改变，它标志这 IO 的完成，以实现同步过程
          - 同步只有 **早晚** 问题，对于驱动与设备，读写请求的操作一定都是异步的，然后由DMA控制器去处理中断，修改内存中的某标志位去指示 IO 的完成
    - 总结一下
      - **事务的提交**标志着数据的落盘
      - 数据**先于**元数据
      - A txn is not considered committed until all its log records have been written to stable storage
- `jbd2_journal_commit_transaction(journal);`
  - `commit_transaction = journal->j_running_transaction;`
  - **phase 1**
  - `commit_transaction->t_state = T_LOCKED;`
    - 调整事务状态，这个事务无法再增加新的 handle，理解为 **关门**
  - **phase 2a**
  - `commit_transaction->t_state = T_FLUSH;`
  - `journal->j_committing_transaction = commit_transaction;`
  - `journal->j_running_transaction = NULL;`
    - journal 上没有 running transaction，但是有了一个 committing transaction
    - 这时如果有其它 handle 的进入，则会触发新的 running transaction 的 **创建与初始化**
  - **phase 2b**
  - `err = journal_submit_data_buffers(journal, commit_transaction);`
    - tips
      - Now start **flushing things to disk**, **in the order they appear on the transaction lists**
      - **Data blocks go first**
        - 实际上这里也只是只 flush data, **data 就是 事务中所管理的那些 inode->mapping 所关联的data**
          - 比如说 inode 2 是对应了数据文件 txt，**这里 submit 的便是 txt 所在的数据块**
          - tmd，终于理解到这里了，这也是为啥 data 要先于 metadata 持久化的原因
      - **Submit all the data buffers of inode associated with the transaction to disk**
    - `err = journal->j_submit_inode_data_buffers(jinode);`
      - `ret = jbd2_journal_submit_inode_data_buffers(jinode);` —— **仅考虑只 journal metadata 的版本**

        - `struct writeback_control wbc = {.sync_mode = WB_SYNC_ALL}` —— **WB_SYNC_ALL means 一定要同步持久化到 storage**
        - `generic_writepages(mapping, &wbc);`
      - **注意，这里的 data 数据持久化还未完成**
      - 此时，**所有的 metadata 均位于内存中**
        - 接下来要准备写 metadata 了，但是写 metadata 中间 crash 的话，就会导致不一致，**所以 metadata 的落盘是需要 事务的支持的**
    - **phase 3**
      - tips
    - `commit_transaction->t_state = T_COMMIT;`
    - `while (commit_transaction->t_buffers)` —— **t_buffers 代表的就是 dirty 的 metadata 所在的 buffer head**
      - `jh = commit_transaction->t_buffers;`
      - `struct buffer_head *descriptor;`
      - `descriptor = jbd2_journal_get_descriptor_buffer(commit_transaction, JBD2_DESCRIPTOR_BLOCK);`
        - tips
          - We play buffer_head aliasing tricks to write data/metadata blocks to the journal **without copying their contents**
          - but for journal descriptor blocks we do need to generate **real fide buffers**
        - `err = jbd2_journal_next_log_block(journal, &blocknr);`
        - `bh = __getblk(journal->j_dev, blocknr, journal->j_blocksize);`
          - 现在要创建一个 buffer head 作为 metadata 的 **运载体**
            - **这个 bh 是要 持久化到 journal 中的**
        - `wbuf[bufs++] = descriptor;`
      - `err = jbd2_journal_next_log_block(journal, &blocknr);` —— **Where is the buffer to be written?**
        - `blocknr = journal->j_head;`
        - `journal->j_head++;`
        - `journal->j_free--;`
        - `jbd2_journal_bmap(journal, blocknr, retp);`
          - **Conversion of logical to physical block numbers for the journal**
    - `err = journal_finish_inode_data_buffers(journal, commit_transaction);`
      - **while + wait_on_bit** 等待data持久化完成
    - **到此为止，data持久化完成了**
    - `update_tail = jbd2_journal_get_log_tail(journal, &first_tid, &first_block);`
      - **get当前log尾指针**
    - `commit_transaction->t_state = T_COMMIT_DFLUSH;`
    - `err = journal_submit_commit_record(journal, commit_transaction, &cbh, crc32_sum);`
    - **phase 4**
    - **phase 5**
    - **phase 6**
    - `commit_transaction->t_state = T_COMMIT_JFLUSH;`
    - `if (update_tail)`
      - `jbd2_update_log_tail(journal, first_tid, first_block);`
    - **phase 7**
    - `commit_transaction->t_state = T_COMMIT_CALLBACK;`
    - `trace_jbd2_end_commit(journal, commit_transaction);`
    - `free`
- **tips**
  - 能够确保持久化的方式一定是 **wait_on_bit + while**
  - 一般 **bio/bh/writepage** 的提交都无法做到持久化，除非下面显示的调用 **wait类的函数**

open系统调用

- `do_sys_open` in `fs/open.c`
  - `long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)`
    - `return do_sys_openat2(dfd, filename, &how);`
    - 无论是打开一个文件还是创建一个文件实际最终调用的就是**do_sys_open**

        - 如果 **不是create**，mode会被忽略
- open 可以与 **数据库的连接** 相对应
    - 起的作用是一样的，都是打通一条路先
    - **其实就是对应的VFS数据结构都创建好**
- **终极简述**
    - 创建文件描述符fd，fd的域有一个dentry的指针，要填充，dentry的域有inode的指针要填充，inode有private域，需要与disk上的匹配
    - 在open的过程中是需要解析路径的，**路径与dentry是对应的**，这个过程可能会涉及到disk IO
- open还有一个重要的左右，**就是维护文件的写位置**，与数据库的游标是相似的，打开就要维护操作位置
    - open文件并且完成写操作如果不关闭文件，文件的写位置是不会变的，**除非调用lseek或直接关闭文件**
        - 否则会继续在后面追加

c标准库的open函数

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
// open函数申明
int open(const char *pathname, int flags, ...);
```

- open 函数可以打开或创建一个文件
    - 成功返回新分配的文件描述符，出错返回-1并设置errno
- 最后的可变参数可以是0个或1个，由flags参数中的标志位决定，见下面的详细说明
    - **O_CREAT若此文件不存在则创建它。**使用此选项时需要提供第三个**参数mode**，**表示该文件的访问权限**

简述

- 打开文件返回的是 **文件描述符fd**，代表一个新的 `struct file`
    - `file` 依赖于 `dentry` ， `dentry` 依赖于 `vfs_inode`
- 所以想要 `file` 在内存中出现， `dentry` 与 `vfs_inode` 必须存在
    - open 过程可能会对应着 `dentry` 与 `vfs_inode` 的创建（**inode可能由后端存储介质加载**）
- 有一个**namei**（即**解析文件路径**）的过程
    - `/dev/dax0.0` 就需要解析**3个dentry**最终找到 `dax0.0`
- file 中关键域的数据结构的赋值在 `do_dentry_open` in `fs/open.c`
    - `struct inode *inode` 被作为参数传入
        - `f->f_inode = inode;`
        - `f->f_mapping = inode->i_mapping;`
        - `f->f_op = fops_get(inode->i_fop);`：`inode` 的 `i_fop` 是一个关键因素
            - `if(open){open = f->f_op->open; error = open(inode, f);}`
                - 会调用**f_op**指定的**open**方法

strace

```
root@ab069b9d443a:~# strace touch open
execve("/usr/bin/touch", ["touch", "open"], 0x7ffd5e2be3d8 /* 15 vars */) = 0
...
openat(AT_FDCWD, "open", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3  # 0666表示8进制的权限
dup2(3, 0)                              = 0
close(3)                                = 0
utimensat(0, NULL, NULL, 0)             = 0
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
...
```

- `mode=0666` ,10进制的 `438`
- `flags=0x8042` ,10进制的 `32834`

由touch open 调试 do_sys_open

- 调试
  - 比较filename
    - 直接在gdbinit中断点了，**内核启动过程调用了无数次~**，必须用**条件断点**，要不这个没法玩
      - 比较filename
      - `b do_sys_open if strcmp(filename, "open") == 0`
        - `char *strstr(char *str, char *substr);`
          - 子串首次出现的地址
      - `int strcmp(const char *s1,const char *s2);`
        - 相等为0
      - 但是比较这个都失败了
        - `Error in testing breakpoint condition: Could not fetch register "fs_base"; remote failure reply 'E14'`
  - 比较mode与flags
    - `b do_sys_open if mode==438 && flags==32834`
      - `Breakpoint 2, do_sys_open (dfd=-100, filename=0x7ffc59464f62 "open", flags=32834, mode=438) at fs/open.c:1186`

在ext4文件系统下 touch file

- ext4 目录下创建一个新文件: `touch /double_D/mnt/ext4point/test.c`
  - 在 open 的过程中，为新文件 `test.c` 创建 `dentry`， `test.c` 的父目录是 `/ext4point` ，它是一个目录dentry，对应一个inode。在 `/ext4point` 创建新文件实际上是要调用 `/ext4point` 所对应inode的 `i_op` ， `/ext4point` 本身是目录，所以 `inode->i_op = &ext4_dir_inode_operations;`
    - 最终调用的是 `ext4point` 的 `ext4_create` in `fs/ext4/namei.c`
      - 它会创建新的inode，并为inode的一些域赋值，下一次打开 `test.c` 进行**读写mmap等操作**的时候就可以直接执行对应的操作了
- ext4目录下创建一个新文件夹
  - 基本流程和上面是一样的，最终**调用父目录**的**ext4_mkdir**去创建子目录
- in `fs/ext4/inode.c` 这部分内容是 **关键的便于理解** 的，是很好的理解资料，这部分内容见 **ostips**
  - `inode->i_op = &ext4_file_inode_operations;`
  - `inode->i_fop = &ext4_file_operations;`
  - `inode->i_op = &ext4_dir_inode_operations;`
  - `inode->i_fop = &ext4_dir_operations;`

- 所以说 创建文件与目录 是两个不同的系统调用
  - `fs/namei.c:SYSCALL_DEFINE2(mkdir, const char __user *, pathname, umode_t, mode)`
  - `fs/open.c:SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)`

lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

- open文件之后，文件有一个写入的游标，并且没有外力干扰是不会改变的，如果有继续写入的行为，那也是追加
  - **而lseek则可以修改一个打开文件的offset**
- 其中whence含义如下
  - SEEK_SET: The file offset is set to offset bytes
  - SEEK_CUR: The file offset is set to its current location plus offset bytes
  - SEEK_END: The file offset is set to the size of the file plus offset bytes
- 返回值
  - 错误返回-1，具体错误类型在errno
  - 否则返回当前的offset
    - `offset = lseek(fd,8192,0);`
      - 如果当前文件只有4K，**那么执行lseek并执行写入操作后文件确实可能超过8K**
      - **lseek确实是可能导致文件变大的**

mknod系统调用

- do_mknodat in fs/namei.c
  - long do_mknodat(int dfd, const char __user *filename, umode_t mode, unsigned int dev)

mknod /dev/myep c 245 0

- do_mknodat
  - vfs_mknod in fs/namei.c
    - error = dir->i_op->mknod(dir, dentry, mode, dev); , 即调用**dev inode**的**mknod**方法。 /dev is in shmem fs (**tmpfs**)
      - init_special_inode in fs/inode.c
        - inode->i_mode = mode;
        - if (S_ISCHR(mode)) {inode->i_fop = &def_chr_fops; inode->i_rdev = rdev; // 这个是设备号}

```
(gdb) p dir->i_op->mknod
$12 = (int (*)(struct inode *, struct dentry *, umode_t,
    dev_t)) 0xffffffff81211300 <shmem_mknod>
```

- dev确实是tmpfs在管的
  - tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)

```
// in mm/shmem.c
static const struct inode_operations shmem_dir_inode_operations = {
        ...
        .mknod          = shmem_mknod,  // 关键方法
        ...
};
```

open("/dev/myep")并执行读写操作

```
// in fs/char_dev.c
/*
 * Dummy default file-operations: the only thing this does
 * is contain the open that then fills in the correct operations
 * depending on the special file...
 */
const struct file_operations def_chr_fops = {
        .open = chrdev_open,
        .llseek = noop_llseek,
};
// in fs/char_dev.c
/* Called every time a character special file is opened */
static int chrdev_open(struct inode *inode, struct file *filp){
        ...
        p = inode->i_cdev;
        if(!p){
                kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);  // open的时候如果没有设备会拿着设备号i_rdev（创建时赋值的）去找设备的
        }
        fops = fops_get(p->ops);
        replace_fops(filp, fops);
        if (filp->f_op->open)
                ret = filp->f_op->open(inode, filp);  // 重新调用真正的open方法
}
struct cdev {
        ...
        const struct file_operations *ops;
        ...
};
```

mmap

- ksys_mmap_pgoff in linux/mm/mmap.c
  - unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len, unsigned long prot, unsigned long flags, unsigned long fd, unsigned lo

fd = open("/dev/dax0.0", O_RDWR) and mmap(NULL, 1024*1024*1024, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_POPULATE, fd, 0)

- ksys_mmap_pgoff
  1. 分配vma，即指定映射的虚拟地址的 start 与 end
     - vma_start：140242073944064
     - vma_end：140243147685888

        - 140243147685888 - 140242073944064 = 1G
    2. 调用 dax0.0 中 file->f_op 的mmap方法（**file->f_op->mmap**）
        - 即 dax_fops 中的 dax_mmap

read or write 系统调用

- 以ext4文件系统为例
    - touch test: open
    - echo 2 > test: write
    - cat test: read

```
const struct file_operations ext4_file_operations = {
        ...
        .read_iter      = ext4_file_read_iter,
        .write_iter     = ext4_file_write_iter,
        ...
};
```

munmap —— linux/mm/mmap.c

```
static int __vm_munmap(unsigned long start, size_t len, bool downgrade)
{
        int ret;
        struct mm_struct *mm = current->mm;
        LIST_HEAD(uf);

        if (mmap_write_lock_killable(mm))
                return -EINTR;

        ret = __do_munmap(mm, start, len, &uf, downgrade);
        /*
         * Returning 1 indicates mmap_lock is downgraded.
         * But 1 is not legal return value of vm_munmap() and munmap(), reset
         * it to 0 before return.
         */
        if (ret == 1) {
                mmap_read_unlock(mm);
                ret = 0;
        } else
                mmap_write_unlock(mm);

        userfaultfd_unmap_complete(mm, &uf);
        return ret;
}
/* ... */
SYSCALL_DEFINE2(munmap, unsigned long, addr, size_t, len)
{
        addr = untagged_addr(addr);
        profile_munmap(addr);
        return __vm_munmap(addr, len, true);
}
```

write

```
const char *buf_7 = "";
ret = write(fd,buf_7,1024);  // 会将文件的大小扩展到1024
```

- 先看一下how to use的含义，即 man write
    - writes **up to count bytes** from the buffer starting at buf to the file referred to by the file descriptor fd
    - 自己在实现fuse write功能的过程中，在buffer为空的情况下，直接返回0了，与write含义不符
- `ksys_write` in `fs/read_write.c`
    - `ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)`
- `ksys_write`
    - `vfs_write`
        - `if(file->f_op->write) ret = file->f_op->write(file, buf, count, pos);`

        - 这里会直接调用文件系统的写方法，取决于文件系统的实现
        - 如果文件系统实现的是 write_iter而不是write，则走下面的路
    - `if(file->f_op->write_iter) ret = new_sync_write(file, buf, count, pos);`
        - 构造一个 `struct kiocb` 对象（**kiocb is kernel io control block**，反正代表了一个**IO请求**），再去调用相应文件系统的 `write_iter` 方法
            - `if(IS_DAX(inode)) return ext4_dax_write_iter(iocb, from);`
            - `if(iocb->ki_flags & IOCB_DIRECT) return ext4_dio_write_iter(iocb, from);`
                - **direct IO**
            - `else return ext4_buffered_write_iter(iocb, from);`
                - **buffered IO**
                - `generic_perform_write` 直接写到了VFS的 `pagecache` 中就返回了
        - 所以说写操作**一定不会被阻塞的**
- **buffered IO**需要被送到驱动层，是由一个内核线程**kjournald2**来完成的 `in fs/jbd2/journal.c`
    - `141 0 0:00 [jbd2/pmem0-8]`
- `kjournald2`：函数调用栈有**省略**
    - `jbd2_journal_commit_transaction`
        - `submit_bh`：**bh means bufer head**
            - `submit_bh_wbc`：**wbc means writeback control**
                - `new bio`
                - `bio_set_dev(bio, bh->b_bdev);`
                    - 与设备关联，即确定了**设备文件的fops**
                    - 本例测试的结果是
                        - `(gdb) p bio->bi_disk->fops`
                        - `$7 = (const struct block_device_operations *) 0xffffffff822bd680 <pmem_fops>`
                - `submit_bio`
                    - `struct gendisk *disk = bio->bi_disk;`
                    - `ret = disk->fops->submit_bio(bio);`

```
struct iovec
{
        void __user *iov_base;
        __kernel_size_t iov_len;
};
static ssize_t new_sync_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos)
{
        // io vector 用于说明用户空间的地址与长度
        struct iovec iov = { .iov_base = (void __user *)buf, .iov_len = len };
        struct kiocb kiocb;  // 控制块
        struct iov_iter iter;  // 看名字，这个称为迭代器
        ssize_t ret;

        init_sync_kiocb(&kiocb, filp);  // 主要就是把文件描述符丢到 kiocb 对象中
        kiocb.ki_pos = (ppos ? *ppos : 0);  // 所操作file的offset
        iov_iter_init(&iter, WRITE, &iov, 1, len);  // count，即操作的数量会在iter中记录

        // 反正之前传过来的值这里都接收到了
        ret = call_write_iter(filp, &kiocb, &iter);  // 调用对应文件系统的ops
        BUG_ON(ret == -EIOCBQUEUED);
        if (ret > 0 && ppos)
                *ppos = kiocb.ki_pos;
        return ret;
}
// iocb 与from反正是包含了buff，len，offset等信息
static ssize_t ext4_buffered_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
        ...
        current->backing_dev_info = inode_to_bdi(inode);
        ret = generic_perform_write(iocb->ki_filp, from, iocb->ki_pos);  // 核心写操作函数
        ...
        if (likely(ret > 0)) {
                iocb->ki_pos += ret;
                ret = generic_write_sync(iocb, ret);
        }

        return ret;
}
```

- 重点来看一下 **kiocb: kernel io control block, 即函数 ret = new_sync_write(file, buf, count, pos);**

- sd_fops
- 磁盘驱动提交IO的操作

```
struct file {
    ...
    struct path         f_path;  // 有包含目录项
    const struct file_operations        *f_op;  // 有包含OP
    loff_t                      f_pos;  // 当前的读写位置
    ...
}
```

- 关注一下内核对write的并发控制（首先要关注文件描述符fd中的一个数据结构**f_pos**，**类比数据库中的游标**，表示的是**当前文件的读写位置**，**偏移量直接决定文件写入到哪里**）
  - 如果以append的方式open文件，那么**写位置**就会被set到文件的最后
    - 执行写操作的时候就会直接追加
  - 普通模式打开的文件，操作位置是0
    - 并发情况下的写操作会覆盖
- 测试场景
  1. 多线程并发写
     - 先创建fd，所有并发线程用一个fd
     - os保证write接口的原子性，所以一个线程在写操作期间不会被打断，而一个线程写操作完成之后**f_pos**也会有对应的修改
       - 所以后续线程可以自然的完成追加操作
     - 在该场景下，**是否以追加的形式打开不重要**
  2. 多进程并发写，父子进程
     - fd在fork之前创建
       - 子进程会复制父进程的fd
         - 实际上子进程继承的是**父进程**的**打开文件描述符表**
       - **这个fd是共享的**
         - **与多线程并发差不多**
     - fd在fork之后创建
       - **父子进程拥有各自的打开文件描述符**
         - 与第三种场景是一致的
  3. 多进程并发，无父子关系
     - 若2个进程同时打开一个文件做**读操作**，每个进程都有自己相对于文件的偏移量，而且读入整个文件是**独立于另一个进程的**
     - 如果2个进程打开同一个文件做**写操作**，写操作是相互独立的，**每个进程都可以重写另一个进程写入的内容**
       - **说白了就是会相互覆盖**
     - append的影响
       - 如果fd有append的flag，就不会相互覆盖了
       - 核心原因就是fd虽然是独立的，但是后端的资源总是共享的
         - **即后端的文件就一个**
       - 怪不得GFS有提供一个**原子追加**的操作
         - **原来这都是本地文件系统都支持的内容**

read

truncate系统调用（截断）

- `truncate` 和 `ftruncate`函数 会将由path指定的常规文件或由fd引用的常规文件截断为长度精确的字节（**理解为重新set一个长度**）
  - 如果以前的文件大于此大小，**额外的数据将丢失**
  - 如果以前的文件较短，**则扩展**，扩展部分读取为空字节 `'\0'`
- `do_sys_truncate` in `fs/open.c`
  - `long do_sys_truncate(const char __user *pathname, loff_t length)`

```
long do_sys_truncate(const char __user *pathname, loff_t length){
        ...
        error = vfs_truncate(&path, length);
        ...
}
SYSCALL_DEFINE2(truncate, const char __user *, path, long, length){
        return do_sys_truncate(path, length);
}
#ifdef CONFIG_COMPAT  // 是否支持64位机上运行32位app
COMPAT_SYSCALL_DEFINE2(truncate, const char __user *, path, compat_off_t, length){
        return do_sys_truncate(path, length);
}
#endif
}
```

- ftruncate, truncate
  - truncate or extend a file to a specified length

inotify

- `do_inotify_init` in `linux/fs/notify/inotify/inotify_user.c`
  - `static int do_inotify_init(int flags)`

sync/fsync系列

- ref
  - http://byteliu.com/2019/03/09/Linux-IO同步函数-sync、fsync、fdatasync/
- 还是来学习一下 sync 以及 fsync 等操作
  - 当调用 write() 函数写出数据时，数据一旦写到该缓冲区（仅仅是写到缓冲区），函数便马上返回。用 write() 写出的文件数据与外部存储设备并非全然同步的。不同步的时间间隔非常短，一般仅仅有 **几秒或十几秒**
  - 内核将缓冲区中的数据 **写** 到标准输入磁盘文件里，不是 move 而是 copy，**也就说此时缓冲区内容还没有被清除**
    - **内核会等待写入磁盘动作完毕后，才放心的将缓冲区的数据删除掉**
  - 当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓冲区高速缓存中内容的一致性，UNIX系统提供了 sync、fsync 和 fdatasync 三个函数
- **要理解 fsync 是非常昂贵的，这也是为什么 write 与 fsync 是两个独立的函数**

sync

- `sync` in `fs/sync.c`
  - `ksys_sync();`
    - ...
    - 有一个遍历的过程。**依然保证能够落盘**
      - **同步阻塞**

fsync

- `fsync` in `fs/sync.c`
  - `do_fsync(fd, 0);`
    - `vfs_fsync(f.file, datasync);`
      - `vfs_fsync_range(file, 0, LLONG_MAX, datasync);`
        - `file->f_op->fsync`
          - `ext4_sync_file`
            - **内部的实现在vfs**

fdatasync

- `fdatasync` in `fs/sync.c`
  - `do_fsync(fd, 1);`
    - ...
      - **其函数调用栈与fsync是一致的**

fsync && fdatasync 在表现上实际就是一个参数的问题

- datasync=1 的话意味着只需要 sync data 就可以了无需 sync 元数据
  - 如果有一个操作是 update，这个本身只会修改数据的value，对于 metadata 只会修改一些 time 等 无关痛痒的 数据，所以这里只sync data，而 metadata 即使没有 sync 下去也不会对一致性造成影响
- 同步阻塞写 data
  - ret = file_write_and_wait_range(file, start, end);
    - while循环中轮询
- 同步阻塞写 metadata
  - ret = ext4_fsync_journal(inode, datasync, &needs_barrier);
    - while循环中轮询

再总结一下 sync && fsync && fdatasync

```
#include <unistd.h>
int fsync(int fd);

#incude <sys/mman.h>
int msync(void *addr, size_t length, int flags)
```

- fsync 针对单个文件，**sync data + metadata**
  - while循环中轮询，阻塞直到设备报告IO完成
    - 如果文件是 mmap 的 IO 方式，有对应的 msync
    - fsync sync 的是整个文件，而 msync sync 的粒度更小。只 sync dirty data
  - sync data + metadata
    - 需要两次 IO 过程
      - IO 过程是昂贵的
    - Unfortunately fsync() will always initialize **two write operations**
      - one for the newly written data
      - another one in order to update the modification time stored in the inode

```
#include <unistd.h>
int fdatasync(int fd);
```

- fdatasync 针对的也是单个文件 **sync data**
  - 但是如果只是 data 的update 操作，元数据只会影响 time 等，所以为了 性能，这里是没有必要去同步 metadata 的，即使 GG 也没有任何问题
  - 实际上就是 **放宽同步的语义** 来提高性能
    - 减少了一次 IO
    - fdatasync does not flush modified metadata unless that metadata is **needed** in order to allow a subsequent data retrieval to be corretly handled
      - 比如 **时间戳** 大部分时间都无伤大雅
      - 但是这种是要求 update 之后， `inode->i_size` 是不变的，文件系统的一致性并不 care 文件的内容是什么
- sync
  - **针对全系统的**

函数 open 的参数 O_SYNC|O_DSYNC

- O_SYNC
  - 与 fsync 具有相同的语义
- O_DSYNC
  - 与 fdatasync 具有相同的语义
- 有机会用到再说吧

使用 fdatasync 优化日志同步

- 为了满足事务要求，数据库的日志文件是常常需要同步IO的。由于需要同步等待硬盘IO完成，所以事务的提交操作常常十分耗时
- 在Berkeley DB下，如果开启了 AUTO_COMMIT（**所有独立的写操作自动具有事务语义**）并使用默认的同步级别（**日志完全同步到硬盘才返回**）
  - 写一条记录的耗时大约为 **5~10ms** 级别，基本和一次IO操作（**10ms**）的耗时相同
- 我们已经知道，在同步上 fsync 是低效的

    - 但是如果需要使用 fdatasync 减少对 metadata 的更新，**则需要确保文件的尺寸在 write 前后没有发生变化**
    - 日志文件天生是追加型（append-only）的，总是在不断增大，似乎很难利用好 fdatasync
- 且看 Berkeley DB 是怎样处理日志文件的（简单来说就是要使 **inode->i_size 变大**，这样之后的写操作就不会涉及到 metadata 的变化（时间戳这种无关痛痒的不 care））
    - 1. 每个log文件固定为10MB大小，从1开始编号，名称格式为 `log.%010d`
    - 2. 每次log文件创建时，先写文件的最后1个page，将log文件扩展为10MB大小
    - 3. 向log文件中追加记录时，由于文件的尺寸不发生变化，使用 fdatasync 可以大大优化写log的效率
    - 4. 如果一个log文件写满了，则新建一个log文件，也只有一次同步metadata的开销

分割线

mkfs(对理解文件系统有帮助)

- 文件系统的代码是写在操作系统中的，我拿到一个空设备，是需要初始化为某种文件系统的，这个过程是**mkfs的工具所执行的**
    - 是要去写磁盘块的，**就是写成空的SFS**
- mkfs
    - 某次需要简单看一下 SSD 的 IO 路径，就准备在 debug 环境中做一下 f2fs 文件系统，将 f2fs 编译到内核后准备安装的时候发现，之前小看 mkfs 了，或者说把 mkfs 看的简单了
    - 首先之前没理解 mkfs 是个啥，以为 mkfs.ext4 是啥黑魔法，但是去 check which 了一下之后发现就是 一个符号连接
        - lrwxrwxrwx 1 root root 6 Feb 14 2020 mkfs.ext2 -> mke2fs
        - lrwxrwxrwx 1 root root 6 Feb 14 2020 mkfs.ext3 -> mke2fs
        - lrwxrwxrwx 1 root root 6 Feb 14 2020 mkfs.ext4 -> mke2fs
    - 但是并不全是符号连接，比如 mkfs.f2fs 就是一个简单的可执行文件，ldd 看一下可以发现
        - `libf2fs.so.0 => /lib/x86_64-linux-gnu/libf2fs.so.0 (0x00007fe26609f000)`
            - 确实是需要一个用户态的 lib 库去支持的
    - 最后还是由 oblidb 的机器上 copy 了一个 `mkfs.f2fs` 到 kernel 的 debug 环境中
        - 当然已经变成了 **静态的可执行文件**

VFS数据结构中所拥有的关键结构

- **file->dentry->inode**

```
// file
struct file {
    ...
    struct path         f_path;  // 有包含目录项
    const struct file_operations        *f_op;  // 有包含OP
    loff_t                      f_pos;  // 当前的读写位置
    ...
}
// 此外，inode还指向实际存储数据的数据块，时间戳等等，所有指向自己目录项的list
struct inode {
    ...
        const struct inode_operations   *i_op;  // inode的操作方法
        struct super_block      *i_sb;  // 超级块
        struct address_space    *i_mapping;  // page cache
        unsigned long           i_ino;  // inode号
        const unsigned int i_nlink;  // 指向这个inode的dentry数量
    const struct file_operations        *i_fop;
    // 这个指的是由disk读上来的inode
    void                        *i_private; /* fs or device private pointer */
    ...
}
// lookup，link，mkdir这些均为目录操作，是定义来inode中的，注意与file op的区别
// shell中操作的时候一般用这个
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
    ...
    int (*create) (struct inode *,struct dentry *, umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
            struct inode *, struct dentry *, unsigned int);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
    ...
};
// 文件的操作函数，注意区分一下，核心功能是读写，编程的时候一般用这个
struct file_operations {
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        ...
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
}


// 目录项的操作不是很常见，先不讨论
struct dentry {
    ...
        struct dentry *d_parent;  // 任何目录项一定都只有一个父目录，是tree
        struct inode *d_inode;          // 多个目录项会公用一个inode
        unsigned char d_iname[DNAME_INLINE_LEN];  // 目录项的名字，很多文件系统都会限制目录名的长度
        const struct dentry_operations *d_op;  // 目录项的操作
        struct super_block *d_sb;       // 文件系统对应的超级块
        struct list_head d_child;       // child of parent list
        struct list_head d_subdirs;     // our children
        ...
}
```

```
struct super_block {
    ...
        unsigned long           s_blocksize;  // 块大小，以字节为单位
    loff_t                      s_maxbytes;     // Max file size
        struct file_system_type *s_type;  // 文件系统类型
        const struct super_operations   *s_op;  // sb的op
        struct dentry           *s_root;  // 文件系统的根目录项
    struct block_device *s_bdev;
    // Filesystem private info
    // 这是一个非常关键的数据结构，以ext4为例
    //  struct ext4_sb_info *sbi
    //  sb->s_fs_info = sbi;
    //  这个地方可以理解为文件系统位于disk上的那一部分
    void                        *s_fs_info;
    ...
}

struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*free_inode)(struct inode *);
    ...
        void (*dirty_inode) (struct inode *, int flags);
        int (*write_inode) (struct inode *, struct writeback_control *wbc);
        ...
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*umount_begin) (struct super_block *);
}
```