openEuler内核编程

课程讲稿

第九章 第2讲

《**F2FS文件系统详解**》

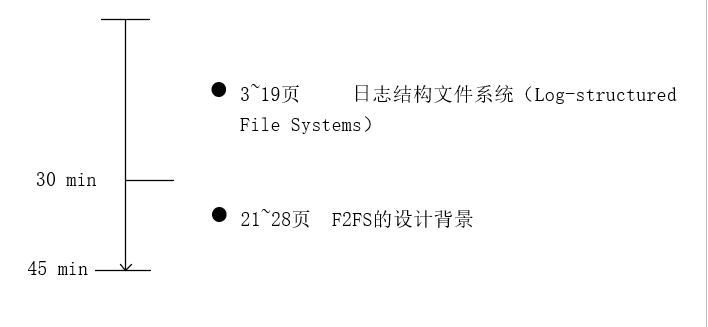
软件所制

第九章 第二讲 F2FS文件系统

**学时：**2学时

**教学目的：** 了解日志结构文件系统的概念，了解F2FS文件系统。

**课程时间线：**



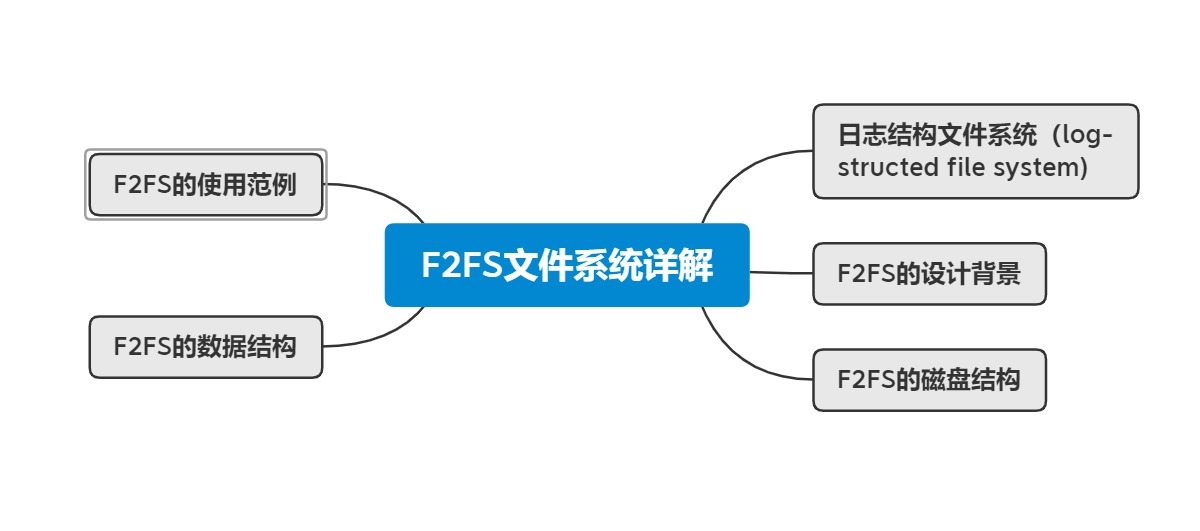
**课外参考读物：**

**1. https://blog.csdn.net/u011649400/article/details/107102072**

**2. 操作系统导论**

**3. https://bbs.huaweicloud.com/blogs/155722**

**知识框图：**

****

**PPT讲稿：**



略



这一节一共有2课时内容，和大家聊聊日志结构文件系统，然后讲一下F2FS。



首先我们简要讲一下日志结构文件系统，日志结构文化系统诞生于上世纪90年代早期，当时计算机的硬件性能开始了爆发式增长：CPU 的速度越来越快，内存也越来越大，用户可以在内存中缓存更多数据，因为读取可以在缓存中进行，所以写入操作占磁盘操作的比例不断增加，文件系统性能很大程度上取决于写入性能。另一方面，磁盘的传输带宽每年增加约50%-100%，寻道和旋转延迟成本下降的速度却只有每年5%-10%，这就造成了随机I/O对顺序I/O性能上的巨大性能优势。那么怎样才能更好地利用磁盘的顺序I/O并专注于写入性能



John K. Ousterhout教授和研究生Mendel Rosenblum领导的开发小组开发出一种新的文件系统-日志结构文件系统。在LFS中，磁盘上存储全部是日志，日志包含元数据信息和文件数据块。具体而言，当文件系统发起写操作时，LFS首先将更新的相关数据块（包括元数据信息）以内存段为单位缓存在主存，然后按顺序写入磁盘的一块连续区域。LFS不会覆盖或删除磁盘上已有数据，相反它将更新后的数据块（包括元数据）按顺序追加写入到空闲磁盘区域。通过将文件更新带来的随机写操作转化为顺序写到磁盘的方式，最大化了写操作效率。

这是LFS的磁盘布局，我们会在下面详细地讲解。

在LFS中，段是更新磁盘的基本单位，也就是说写入磁盘时，LFS会缓冲内存段内的更新，段满后将该段一次性写入磁盘，只要段足够大，写入就会很有效。

大家看ppt上的例子，LFS将两组更新缓冲到一个小段中，此次更新是对文件j的四个数据块和元数据写入以及对文件k的数据块添加以及源数据更新。之后LFS会把整个七个数据块和两个inode写入磁盘

我们知道在典型的文件系统（如FFS以及FFS中），因为这些文件系统的inode以数组形式组织，放在磁盘的固定位置，所以对inode的查找相对简单，比方说，在UNIX文件系统中，给定一个inode号和inode表的起始地址，要查找特定的inode，只需要将inode号乘以inode的大小加上inode表的起始地址，就可以获得确切的磁盘地址，即使在FFS中，因为FFS将inode表分块的缘故，知道每个inode块的大小和每个inode块的起始地址，再加上inode号，也容易计算特定inode的地址，然而在LFS中，我们把inode分散在整个磁盘中，而且永远不会覆盖，因此最新版本的inode会不断移动。

。

这个问题的解决方案相对比较复杂，LFS的设计者引入名为LFS映射的数据结构，在inode号和inode之间引入一个间接层（inode map），imap接受inode号为输入，输出最新版本的inode的磁盘地址。每次将inode写入磁盘，都会同时更新imap。

现在有一个新的问题，imap应该驻留在磁盘的哪一个位置？我们可以把它存储在磁盘的固定部分，但是由于imap需要定时更新，每次对inode的更新都要求写入imap，这会产生更多的磁盘寻道，造成额外的开销，我们显然不愿意看到这种情况发生，所以我们将imap的快放在他写入其他新信息附近的位置，因此数据块每次追加到文件K时，LFS将新的数据块，新的inode以及一段imap一起写入磁盘。

我们将imap和数据块和inode放在一块写入磁盘又诞生了一个新的问题，怎么查找imap，现在imap也分布在整个磁盘中，LFS在磁盘的固定位置存放一个检查点区域（Checkpoint region，CR）,检查点区域包含最新的imap指针，可以通过查找CR获得imap的位置，CR定期更新而不是随着IMAP更新而更新，这会降低对性能的影响。CR始终处于磁盘的开头，地址为0

为了更好的理解LFS是如何工作的，我们假设内存中没有任何东西，怎么从磁盘读取文件，首先要读入的第一个块是检查点区域，从检查点区域我们可以获得文件的imap地址，然后LFS将imap读入内存，在这之后给定inode号，我们可以获得文件的inode磁盘地址读入最新版本的inode，通常情况下，LFS执行与典型文件系统同样数量上的I/O,imap会缓存，LFS在读取过程的额外工作实在IMAP中查找inode地址。

大家看ppt，假设有一个inode号0引用的文件指向数据块D0，现在我们覆盖该块，产生新的inode号新的数据块，那么原有的就成为了垃圾文件，同理我们将一个块追加到该原始文件，这种情况会产生新的inode号，原有的inode就成为了垃圾，



LFS会反复将最新版本的文件写入磁盘的固定位置，这会提高写入效率，然而同时会在整个磁盘中产生大量的旧版本文件结构。大家知道，真实情况下的磁盘容量是有限的，如果不予以处理的话，最终旧版本文件结构会占满整个磁盘空间，这种情况下就会出现没有新的磁盘块可用。该如何处理这些旧版本文件结构？



为了处理这个问题，LFS 有一个清理(clean)线程，它会循环扫描日志并对日志进行压缩。首先，通过查看日志中第一部分的信息来查看其中存在哪些索引节点和文件。它会检查当前 inode 的映射来查看 inode 否在在当前块中，是否仍在被使用。如果不是，该信息将被丢弃。如果仍然在使用，那么 inode 和块就会进入内存等待写回到下一个段中。然后原来的段被标记为空闲，以便日志可以用来存放新的数据。用这种方法，清理线程遍历日志，从后面移走旧的段，然后将有效的数据放入内存等待写到下一个段中。由此一来整个磁盘会形成一个大的环形缓冲区，写线程将新的段写在前面，而清理线程则清理后面的段。



注意这里有一个容易混淆的概念，那就是日志结构文件系统和日志文件系统。前者的代表是F2FS，我们会在后面详细地讲，后者的代表就是ex3。日志结构文件系统，是一种为了append操作而特殊优化的文件系统。他将一部分的随机写转化为了顺序写，从而提高了写入性能，在大数据与云平台场景下有比较好的应用。而日志文件系统则是用日志来保证灾备恢复的一致性的。详细的说，日志结构文件系统在文件系统本身内部维护日志，。磁盘上的文件系统数据以日志形式组织。而日志文件系统在日志中跟踪文件系统的所有更改，日志与磁盘上的数据分开。这里要区分开。



基于LFS的文件系统如果使用平常的多级索引数据结构，会存在wandering tree问题。比如，如果使用ppt中的多级索引：

当我们对某个indirect指向的数据更新的时候，基于LFS实现，就需要更新对应的索引树。显然这样一次更新数据，需要更新三级索引结构，写放大太大。LFS使用平常的多级索引数据结构时，会出现文件数据更新时，直接指针指针、直接指针的间接指针会更新，然后保存间接指针的inode、inode map等结构也需要更新，导致了对meta数据频繁的更新。这个问题就是LFS的wandring tree 问题。

下面我们介绍一下F2FS的设计背景。



F2FS属于日志结构文件系统的一种，它专门针对SSD、eMMC、UFS等flash设备设计，可以说是移动互联网时代的文件系统。它支持32位块寻址空间，最大支持文件系统16TB，最大文件大小3.94TB。除此之外，它还具有下列特点：Flash 感知，解决Wandering Tree 问题，最小化Cleaning 开销，我们会在后面详细讲解这些特点。



在这之前，我们先讲一下什么是NAND Flash，Flash分为NAND Flash和NOR Flash，NOR型存储内容以编码为主，其功能多与运算相关；NAND型主要功能是存储资料，如数码相机中所用的记忆卡。NAND Flash有优秀的读写性能、较大的存储容量和性价比，因此在大容量存储领域得到了广泛的应用但是不具备随机访问性能。NAND Flash有着下列规则：首先Flash不支持覆盖，即写入操作只能在空或已擦除的单元内进行。其次以page为单位写入，以Block为单位擦除；擦除Block前需要先对里面的有效页进行搬迁。

然后每个Block都有擦除次数限制（有寿命），擦除次数过多会成为坏块（bad block）。



从上面我们知道，NAND Flash不能像内存那样随意的写入。NAND Flash在写入之前必须要将Page页所在的Block块擦除。这就引起了两个问题，第一，按照这种方式进行写操作，写入的性能将会很差；其二，不断的对同一Block块进行擦除操作，那么该块将会在短时间内磨损写坏，并且极易导致存储在该块上的数据丢失。



采用Log-structured的方式之后，NAND Flash可以采用out-of-place的数据更新方式。所有的数据更新都不会写入原来的page页，而是重映射写入一个新的Page页。在这个思路的引导，很显然所有NAND Flash的存储资源可以按照物理Page页的方式管理起来，而用户可见的空间则是一个连续逻辑Page页连接起来的地址空间。FTL的一个关键任务就是建立逻辑Page和物理Page之间的映射关系，并且在数据写入时重新分配物理Page页。大家是不是想起了imap，很多时候技术是相同的。



F2FS扩大随机写区域（元数据区域，two-location）以获取更好的性能

尽量使F2FS 的数据结构与FTL 的运算部件对齐



我们之前说LFS存在wandering Tree问题，但是F2FS很好的解决了它，怎么解决的呢。造成wandring tree的关键是上级索引直接指向了下级索引，基于LFS索引也不能原地更新，这样一旦下级索引有改动，上级索引也需要随之更新。为此，可以在上级索引和下级索引直接引入一层防火墙，来隔离这两者的相互影响，避免更新向上层传播。F2FS中把block分为两类：NODE和DATA。NODE存储元数据，DATA存储文件数据，NODE类型block包括三类元数据：inode、直接dnode、间接dnode。直接dnode的表项指向的是DATA block的地址，所以DATA page变脏了之后，DATA block就要变更了，所以被殃及的直接dnode当然也脏喽。但是注意，此时火势并不会蔓延到间接dnode上，因为间接dnode表项指针指的并不是直接dnode的block地址，而是NAT表中的一个表项，所以NAT就像防火枪一样防止了tree任意滋生：你只要把NAT中相应的block\_addr域给更新掉就可以了，我间接dnode的指针还是指向这个NAT表项



引入NAT，实现node id和node block的地址映射。我们要访问某个文件第11个block的数据块，根据pointer11,找到对应的indirect block，基于inode的索引层次，算出对应物理块地址在这个indirect block内由哪个entry索引，读取对应entry记录的NodeID,查找NAT,得到最终的物理地址。最后基于这个物理地址，访问数据。



F2FS通过支持后台Cleaning 进程、支持greedy 和 低成本(转移数据最少)的待清理Section 选择算法、支持multi-head logs 用于动态/静态hot 与cold 数据分离、引入自适应logging 用于有效块分配实现了最小化cleaning开销。



接下来我们讲一下F2FS的磁盘布局



F2FS的数据存储的基本单位是block，大小为4KB，整个flash设备被格式化为多个block组成的结构。很多数据结构也被设计为4KB的大小，这是因为很多flash设备单次IO的读写都是基于4KB的倍数进行。segment: segment是管理block的结构，一个segment的大小是512个block，也就是2MB。section: 默认情况下一个segment等于一个section，section是垃圾回收的基本操作单位，每次垃圾回收都会从section中选出特定的segment进行回收。默认情况一个zone等于一个section，与物理设备有关，大部分情况下用不上。



大家看ppt上的视图可能更容易理解

整个存储设备区域被F2FS格式化为6个区域，分别是Superblock，Checkpoint，Segment Info Table，Node Address Table，Segment Summary Area，以及Main Area。前5个区域总称为元数据区域(Metadata Area)，保存的是跟F2FS直接相关的元信息，而最后一个区域是保存文件数据的主要区域，主要保存了node数据、文件data数据、目录数据



我们看一下F2FS中六个元数据区域的作用：

Superblock :保存了核心元数据的结构，包括磁盘大小，元区域的各个部分的起始地址等

Checkpoint :维护了系统当前的状态，例如segment的分配情况，node的分配情况，以及当前的active segment的状态等

Segment Information Table(SIT) :维护segment的block的使用状态以及有效无效状态

Node Address Table(NAT) : 维护了一张表，记录了每node在flash设备的物理地址

Segment Summary Area(SSA) :集中管理物理地址到逻辑地址的映射关系，同时也具有通过journal缓存sit或者nat的操作用于数据恢复的作用

Main Area : F2FS的主要数据保存区域



F2FS 采用 Checkpoint 机制维护文件系统的一致性。在挂载文件系统的时候，F2FS 首先尝试通过扫描 CP 区域找到最新的有效的 Checkpoint。为减少扫描时间， CP仅使用两个备份，其中一个备份中包含的是最新的有效数据。这种两个备份，且仅有一个包含最新版本有效数据的技术也称为 shadow copy 机制。除了 CP，NAT 和 SIT 也采用 shadow copy 机制保证数据一致性。为保持文件系统一致性，每个 CP 指向有效的 NAT 和 SIT 备份



F2FS 有929个直接索引地址，2个一级间接索引块，2个二级间接索引块，1个三级间接索引块。允许寻址接近4TB的文件。大家知道怎么计算么？4 KB \* (929 + 2\*1018 + 2\*1018\*1018 + 1018\*1018\*1018) := 3.94 TB



一个目录项 block 包含214个目录项槽位(slot)和文件名，其中用一个 bitmap 表示每个目录项是否有效，1个目录项块占用 4 KB，结构如下：

Dentry block(4 Kb) = bitmap(27bytes) + Reserved(3bytes) + Dentries(11\*214bytes) + filename(8\*214bytes)



所有的哈希表按如下方式组织：第一个哈希表恰好有1个 bucket，大小是两个数据块，因而对于最开始的少数几百个目录项，使用简单的线性搜索。第二个哈希表有2个 bucket，第三个哈希表有4个 bucket，第四个哈希表有8个 bucket，然后是16 个 bucket，……，直到第31个哈希表有大约10亿（2^30）个 bucket，每个 bucket 都是两个数据块大小。从第32个哈希表开始，都与第31个哈希表一样，但是不同的是每个 bucket 的大小是四个数据块。

F2FS 在目录中查找文件名时，首先计算文件名的哈希值，然后在 level #0 中扫描哈希值查找包含文件名和文件的 inode 的目录项。如果没有找到，F2FS在 level #1 中扫描下一个哈希表，用这种方式，F2FS 以递增的方式扫描每个 level 的哈希表（如果上一 level 中没有查找到结果），在每个 level中，F2FS 仅需要扫描一个bucket，而该 bucket 的号是由文件名的哈希值与该 level 中的 buckets 个数的相除取余得到的。也就是每个 level 中需要扫描的一个 bucket 由下式确定的，查找复杂度是 0：

bucket number to scan in level #n = (hash value) % (# of buckets in level #)

所以哈希表的最终结果就是需要线性搜索几百个目录项，如果目录项很大的话，需要通过几个数据块的搜索。搜索长度仅随着目录文件中目录项的个数对数级增长，所以容易扩展。这种搜索当然比完全的线性搜索算法要好，但是看起来好像比实际需要做的工作更多。但是它保证了每次加入或删除一个文件名的时候仅需要更新一个数据块，因为目录项没有移动，目录项在文件中的偏移对于 telldir() 来说就是一个稳定的地址，这是非常有意义的特性。

在创建文件的情况下，F2FS 在哈希表的buckets 中找到该创建文件名对应的空的连续的槽位(slots)（这一句的原文是：F2FS finds empty consecutive slots that cover the file name）。F2FS 自哈希表的 level #0 到level #N 从哈希表中给搜索该空闲的槽位，与查找文件系统中已有文件的查找操作一样。



F2FS在“Main Area”区域管理6个活跃的log，分别是Hot/Warm/Cold node log和Hot/Warm/Cold data logs

根据数据所属的类型，利用对应的log分配的空间将数据写入该空间



LFS有两种机制用于空闲空间管理：Threaded log和copy-and-compaction。Copy-and-compaction机制就是log-structured文件系统中常常提到的cleaning操作，Copy-and-compaction机制非常适用于具有非常好的连续写性能的设备，但是该机制的cleaning操作开销对性能影响很大。相反，Threaded log机制的弊端是要承受随机写，但是不需要cleaning过程。F2FS采用混合机制，其中Copy-and-compaction机制是默认采取的机制。但是F2FS会根据文件系统的状态(在没有足够的clean的segments )动态地更改为Threaded log机制



F2FS可以在需要的时候(on-demand)或者空闲的时候以后台处理的方式进行clean操作。On-demand cleaning操作是由于没有足够的空闲segments服务VFS调用而触发开启的cleaning操作，后台cleaner由内核线程操作，当系统I/O Idle的时候触发cleaning操作。

F2FS支持两种选择待清理segments的策略：greedy和cost-benefit算法。在greedy算法中，F2FS选择具有最少有效数据块个数的segment；在cost-benefit算法中，F2FS根据segment的年龄以及有效数据个数选择segment，以解决greedy算法中的log block颠簸问题(频繁的cleaning操作，频繁的数据块迁移，极端情况，选中的segment可能在很长一段时间内一直是greedy算法的选择结果)。F2FS对on-demand cleaner采用greedy算法选择待清理的segment，而对于后台cleaner采用cost-benefit算法。



下面我们讲一下F2FS的数据结构。



F2FS中的一个inode，包含两个主要部分: metadata部分，和数据块寻址部分。我们重点观察数据块寻址部分，分析inode时如何将数据块索引出来。在图中，数据块寻址部分包含direct pointers，single-indirect，double-indirect，以及triple-indirect。它们的含义分别是：

direct pointer: inode内直接指向数据块(图右上角Data)的地址数组，即inode->data模式。

single-indirect pointer: inode记录了两个single-indirect pointer(图右上角Direct node)，每一个single-indirect pointer存储了多个数据块的地址，即inode->direct\_node->data模式。

double-indirect: inode记录了两个double-indirect pointer(图右上角indirect node)，每一个double-indirect pointer记录了许多single-indirect pointer，每一个single-indirect pointer指向了数据块，即inode->indirect\_node->direct\_node->data模式。

triple-indirect: inode记录了一个triple-indirect pointer(图右上角indirect node)，每一个triple-indirect pointer记录了许多double-indirect pointer，每一个double-indirect pointer记录了许多single-indirect pointer，最后每一个single-indirect pointer指向了数据块。即inode->indirect\_node->indirect\_node->direct\_node->data模式。

因此，我们可以发现，F2FS的inode结构采取indirect\_node，首先在inode内部寻找物理地址，如果找不到再去direct\_node找，层层深入。



我们知道对于一个较大的文件，它可能包含inode以外的node，去保存一些间接寻址的信息。single-indirect pointer记录的是数据块的地址，而double-indirect pointer记录的是single-indirect pointer的地址，triple-indirect pointer记录的double-indirect pointer地址。在F2FS中，

inode对应的是f2fs\_inode结构，包含了多个direct pointer指向数据块物理地址；

single-indirect pointer对应的是direct\_node结构，包含了多个direct pointer指向物理地址；

double-indirect pointer对应的是indirect\_node结构，包含了多个指向direct\_node的地址；

triple-indirect pointer对应的也是indirect\_node结构，包含了多个指向indirect\_node的地址

接下来我们逐个分析F2FS每一个node的具体数据结构。f2fs\_inode和direct\_node以及indirect\_node都使用了同一个数据结构f2fs\_node进行描述，并通过union的方式，将f2fs\_node初始化成不同的node形式



我们看f2fs\_inode的结构，省略其他元数据的信息，重点关注文件如何索引的

i\_addr数组就是前面提及的direct pointer，数组的下标是文件的逻辑位置，数组的值就是flash设备的物理地址。例如文件的第一个页就对应i\_addr[0]，第二个页就对应i\_addr[1]，而i\_addr[0]和i\_addr[1]所记录的物理地址，就是文件第一个页(page)和第二个页的数据的物理地址，系统可以将两个物理地址提交到flash设备，将数据读取出来。

我们可以发现i\_addr的数组长度只有923，即一个f2fs\_inode只能直接索引到923个页/块的地址(约3.6MB)，对于大于3.6MB的文件，就需要使用间接寻址。f2fs\_inode的i\_nid数组就是为了间接寻址而设计，i\_nid数组是一个长度为5的数组，可以记录5个node的地址。其中

i\_nid[0]和i\_nid[1]记录的是direct\_node的地址，即对应前述的single-indirect pointer。

i\_nid[2]和i\_nid[3]记录的是indirect\_node的地址，这两个indirect\_node记录的是direct\_node的地址，即对应前述的double-indirect pointer。

i\_nid[4]记录的是indirect\_node的地址，但是这个indirect\_node记录的是indirect\_node的地址，即前述的triple-indirect pointer。

direct\_inode以及indirect\_inode的结构如下所示，direct\_node记录的是数据块的地址，indirect\_inode记录的是node的id，系统可以通过nid找到对应的node的地址。



这两页ppt是关于F2FS的inode的，inode是用来和外部用户交互的，inode包括和VFS交互，包括ACL、time等相关数据信息。其中上面的i\_addr 直接可以指向数据块，如果数据块的数量超过了DEF\_NIDS\_PER\_INODE，就需要使用i\_nid。 i\_nid 数组可以用来分别指向2个direct、2个indirect、1个double indirect的 block地址索引块。



上面f2fs\_inode数据结构是一个inode块里面的内容。那么这个inode块的地址如何确定呢？这就是f2fs\_nat\_entry的职责了, 每个f2fs\_nat\_entry 记录了每个inode编号和其inode块数据地址的对应关系。而专门存储f2fs\_nat\_entry的block，组成了f2fs\_nat\_block。可是上面问题又来了， inode number如何确定？NAT block起始地址在哪，有多少个？



f2fs\_dir\_entry 回答了上面的第一个问题，它把inode number和文件名通过hash关联起来了。同样，也有专门存储f2fs\_dir\_entry的块，叫做f2fs\_dentry\_block.



下面是F2FS的使用范例，我们将讲一下F2FS的读流程与写流程



F2FS的读流程包含了以下几个子流程:

vfs\_read函数

generic\_file\_read\_iter函数: 根据访问类型执行不同的处理

generic\_file\_buffered\_read: 根据用户传入的文件偏移，读取尺寸等信息，计算起始位置和页数，然后遍历每一个page，通过预读或者单个读取的方式从磁盘中读取出来

f2fs\_read\_data\_page&f2fs\_read\_data\_pages函数: 从磁盘读取1个page或者多个page

f2fs\_mpage\_readpages函数: f2fs读取数据的主流程

我们之前已经讲过vfs\_read函数，这里不再赘述

1. generic\_file\_read\_iter 函数的作用是处理普通方式访问以及direct方式访问的读行为

我们先讲一下先介绍一种VFS提高读取速度的机制: 预读(readahead)机制。它的核心原理是，当用户访问page 1，系统就会将page 1后续的page 2，page 3，page 4一起读取到page cache(减少与磁盘这种速度慢设备的交互次数，提高读性能)。之后用户再连续读取page 2，page 3，page 4时，由于已经读取到内存中，因此可以快速地返回给用户。eneric\_file\_buffered\_read函数的主要作用是循环地从磁盘或者内存读取用户需要的page，同时也会在某些情况调用page\_cache\_sync\_readahead函数进行预读，page\_cache\_sync\_readahead有两种情况：情况1: 预读(readahead)机制成功预读到用户需要接下来访问的page

ind\_get\_page: 系统无法在cache中找到用户需要的page

page\_cache\_sync\_readahead: 系统执行该函数进行预读，一次性读取多个page

find\_get\_page: 再重新在cache获取一次page，获取成功后跳转到page ok区域

page\_ok: 复制page的数据去用户传入的buffer中，然后判读是否为最后一个page，如果是则退出读流程

情况2: 预读(readahead)机制错误预读到用户需要接下来访问的page

find\_get\_page: 系统无法在cache中找到用户需要的page

page\_cache\_sync\_readahead: 系统执行该函数进行预读，一次性读取多个page

find\_get\_page: 再重新在cache获取一次page，获取失败，跳转到no\_cached\_page区域

no\_cached\_page: 创建一个page cache结构，加入到LRU后，跳转到readpage区域

readpage: 执行mapping->a\_ops->readpage函数从磁盘读取数据，成功后跳转到page ok区域

page\_ok: 复制page的数据去用户传入的buffer中，然后判读是否为最后一个page，如果是则退出读流程。预读机制会调用f2fs\_read\_pages函数一次性读取多个page。而当预读失败时，也会调用f2fs\_read\_page读取单个page。



无论是f2fs\_read\_page函数还是f2fs\_read\_pages函数，都是调用f2fs\_mpage\_readpages函数进行读取，区别仅在于传入参数。大家可以在ppt看一下f2fs\_mpage\_readpages的定义

第二个参数表示一个链表头，这个链表保存了多个page，因此需要写入多个page的时候，就要传入一个List。

第三个参数表示单个page，在写入单个page的时候，通过这个函数写入。

第四个参数表示需要写入page的数目。

因此

在写入多个page的时候，需要设定第二个参数，和第四个参数，然后设定第三个参数为NULL。

在写入单个page的时候，需要设定第三个参数，和第四个参数，然后设定第二个参数为NULL。

这个函数的执行流程如下：遍历传入的page，得到每一个page的index以及inode

将page的inode以及index传入 f2fs\_map\_blocks 函数获取到该page的物理地址

将物理地址通过 submit\_bio 读取该page在磁盘中的数据



F2FS的写流程主要包含了以下几个子流程:

调用vfs\_write函数

调用f2fs\_file\_write\_iter函数: 初始化f2fs\_node的信息

调用f2fs\_write\_begin函数: 创建page cache，并填充数据

写入到page cache: 等待系统触发writeback回写到磁盘

调用f2fs\_write\_end函数: 将page设置为最新状态

调用f2fs\_write\_data\_pages函数: 系统writeback或者fsync触发的时候执行这个函数写入到磁盘

第一步的vfs\_write函数是VFS层面的流程，下面仅针对涉及F2FS的写流程，且经过简化的主要流程进行分析。



f2fs\_file\_write\_iter的主要作用是在数据写入文件之前进行预处理，核心流程就是将该文件对应f2fs\_inode或者direct\_node对应写入位置的i\_addr或者addr的值进行初始化。例如用户需要在第4个page的位置写入数据，那么f2fs\_file\_write\_iter函数会首先找到该文件对应的f2fs\_inode，然后找到第4个page对应的数据块地址记录，即f2fs\_inode->i\_addr[3]。如果该位置的值是NULL\_ADDR则表示当前是添加写(Append Write)，因此将值初始化为NEW\_ADDR;如果是该位置的值是一个具体的block号，那么表示为覆盖写(Overwrite)，不需要做处理。



VFS中write\_begin和write\_end函数分别是数据写入page cache前以及写入后的处理。写入page cache后，系统会维护一段时间，直到满足一定条件后(如fsync和writeback会写)，VFS会调用writepages函数，将这些缓存在内存中的page一次性写入到磁盘中。分析VFS的write\_begin和write\_end对应的功能，write\_begin在F2FS中对应的是f2fs\_write\_begin，它的作用是将根据用户需要写入的数据类型，对page进行初始化。



将用户数据写入到page cache之后，进行write\_end处理，在F2FS中对应的是f2fs\_write\_end函数，f2fs\_write\_end将page设置为最新状态。



系统会将用户写入的数据先写入到page cache，然后等待时机回写到磁盘中。page cache的回写是通过f2fs\_write\_data\_pages函数进行。系统会将page cache中dirty的pages加入到一个list当中，然后传入到` f2fs\_write\_data\_pages进行处理。针对F2FS文件系统，它包含如下步骤:

f2fs\_write\_data\_pages&\_\_f2fs\_write\_data\_pages函数: 做一些不那么重要的预处理

f2fs\_write\_cache\_pages函数: 从inode->mapping的radix tree中取出page

\_\_write\_data\_page函数: 判断文件类型(内联文件，目录文件，普通文件)进行不同的写入

f2fs\_do\_write\_data\_page: 根据F2FS的状态选择进行就地回写(在原物理地址更新)还是异地回写(在其他物理地址更新)

f2fs\_outplace\_write\_data: 执行回写，更新f2fs\_inode的状态

do\_write\_page: 从CURSEG分配物理地址，然后写入到磁盘

1. 完