openEuler内核编程

课程讲稿

第九章 第3讲

《**EROFS介绍**》

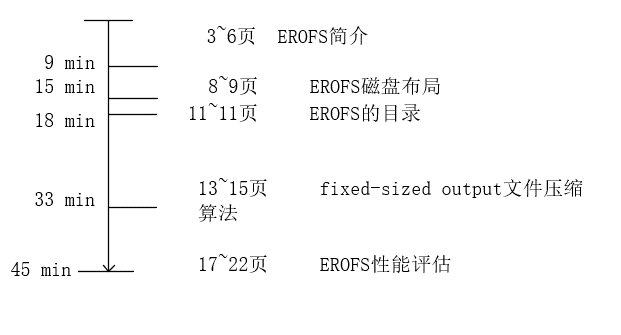
软件所制

第九章 第三讲 EROFS

**学时：**1学时

**教学目的：** 了解EROFS文件系统。

**课程时间线：**



**课外参考读物：**

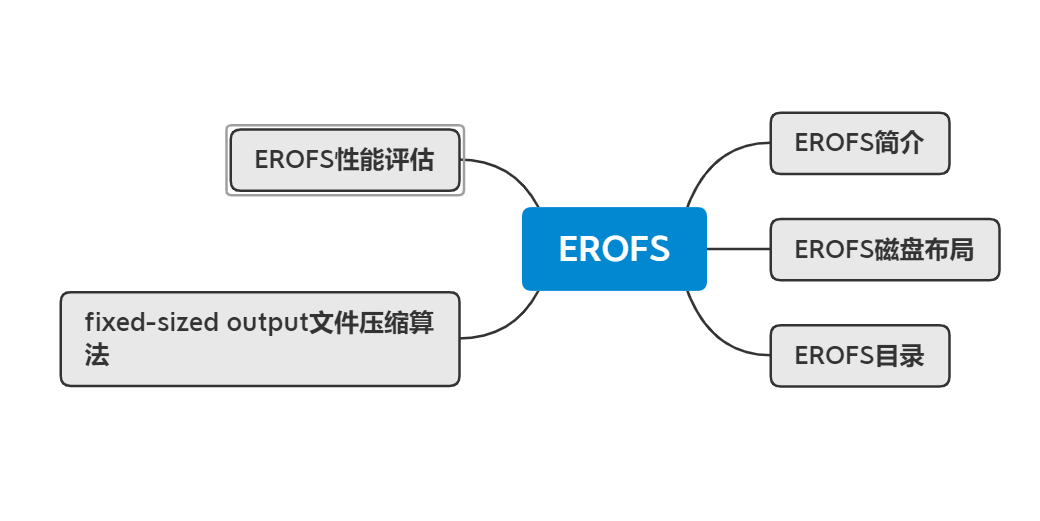
**1. https://www.kernel.org/doc/html/latest/filesystems/erofs.html**

**2. https://www.usenix.org/system/files/atc19-gao.pdf**

**3. 操作系统导论**

**4.** [**https://nan01ab.github.io/2019/11/File-Systems.html/**](https://nan01ab.github.io/2019/11/File-Systems.html/)

**知识框图：**

****

**PPT讲稿：**



略



这一节和大家聊聊由华为设计的超级文件系统-EROFS



大家应该都用过智能手机，android是眼下最流行的手机操作系统，有十亿级别的设备搭载Android系统，但是其对存储资源的需求却随着版本号的迭代逐渐增加，在2.3.6时代，系统分区可能只需要512MB，相当于0.5GB，但是到了8.0时代，仅仅系统分区就需要3072MB，相当于3GB。Android就像一个怪兽一样吞噬着系统资源，为了解决这个问题，华为发布了EROFS文件系统，可提供比其他文件读取系统方案更好的性能，同时还可以节省存储空间。

EROFS文件系统是华为公司自研的一项提升手机随机读写性能的系统及应用编译和运行机制，全称为Extendable Read-Only File System（可扩展读取文件系统）。他有两个比较重要的特点：提供一种不同的压缩模式设计fixed-sized output文件压缩算法，相比于以固定大小为输入，变长输出的fixed-size input文件压缩算法（如squashfs文件系统），前者的压缩率更高，并且能够有效的减少读放大问题，降低IO读取量。其次，EROFS加强了对数据的安全防护，EROFS是天然只读的，没法通过任何手段挂载成可写的，这就相当于筑了一道难以逾越的墙，使用EROFS的系统分区更加安全了。EROFS已经并入内核主线4.19分支，代码可以在driver/staging/erofs目录上获取



下面我们看一下EROFS的主要特性：小端磁盘布局、4KB块大小、最大文件系统16TB、元数据和数据混合设计、提供两种inode类型、支持LZ4文件压缩节约存储空间、支持xattr。我们将在后面陆续介绍这些特性。



EROFS支持两种inode，第一种inode元数据大小为32位，最大文件大小4GB，最大用户ids和组ids为65536，不支持文件修改时间，最大硬链接数65536，为元数据保留4位。第二种64位，最大支持16EB，最大用户ids和组ids为4294967296，支持文件修改时间，最大硬链接数4294967296，为元数据保留14位。



接下来我们看一下EROFS的磁盘布局。



EROFS被应用于Android的只读系统分区。以往Android只读系统分区使用ext4文件系统的只读挂载参数和挂载dm-verity虚拟块设备（用于文件系统校验）来配合实现只读。这并非是一种结构上的限制，通过去除Ext4只读挂载选项并且不挂载dm-verity设备，就可以对系统分区进行写操作。

而EROFS在结构上即为天然只读，存储单元间通过一种类似链表的结构进行连接，前面的存储单元保存着其后一个存储单元的位置，不通过全局bitmap管理所有可写区域。因此外来者无法知道哪个区域是有效可写入的，即使强行写入，其内容也不会被其它存储单元承认。

保障只读为整个系统的设计带来了很多好处，由于数据空间大小已经事先确定，文件系统可以消除直接inode/间接inode这样的设计，也省略掉了inode bitmap和block bitmap（记录空的block）这种区域来节省空间。直接把每个文件压缩后的数据顺序储存，然后保留压缩后每个簇对应的原始数据范围作为元数据即可。这样一来对文件的顺序访问变成了顺序读取，性能得到了很大提升。实现细节上，EROFS的inode table是变长的，里面包含了inode结构体和少量的内联文件数据。xattr和压缩特性都是可选的，未选择的情况下inode table空间将进一步缩小。



EROFS目前基本以4K大小为block，一个文件4K不对齐的结尾的部分会作为内联文件数据和inode head连接在一起，以提高page cache的利用和命中率。



下面我们看一下EROFS的目录项



EROFS的目录项每项12字节，文件名被联合在一起放在目录项的后面节约存储空间，inode number用64字节大小保存。紫色部分是一个inode结构体的启始，可以简单的通过inode number \* 32 + super block offset = paddr来得到inode结构体的分区物理存放位置，缩小了dirent结构体的大小，因为不用保存inode结构体的位置。



下面我们看一下之前提到的EROFS的fixed-sized output文件压缩算法



首先我们了解一下什么是读放大问题。一般存储器以4K为最小的读写单元，要找到一个读写单元就要寻址。如果有一组数据大小在5K左右，压缩后占用3.5K的空间。当这3K的数据全部在一个单元内时，一次读写就能完成我们的操作，但是如果这3K的数据前1.5K在第一个存储单元内，后2K在另一个存储单元内，就需要进行至少两次读写，会导致性能的下降。在以往Linux内核使用的压缩只读文件系统squashfs中，每4K数据进行一次压缩。数据不同，压缩率也不一样，有的数据可以压缩到很小，有的数据几乎没办法压缩。这样就会压缩出很多大小不一的数据块。尽管这些数据块全部小于4K，然而可以说，这些数据块几乎完全不能和4K单元对齐，因此读写时会造成很多的额外消耗。例如一个128k的文件， squashfs会采用固定大小（比如32K）为输入，即会将每32k数据压缩成不确定的大小，如可能是12K；那么读取文件中的任意4k数据，都需要至少读取完整的12k压缩数据，然后解压再读取其中的4k，这多读取的8K就是所谓的“读放大问题”。



我们举一个具体的例子了解一下EROFS是怎么解决这个问题的。一个文件4K不对齐的结尾的部分会作为内联文件数据和inode head连接在一起，故压缩后的物理块大小为固定值（4K）。相比于以固定大小为输入，变长输出的fixed-size input能够有效的缓解读放大问题，降低I/O读取量。对于上面的情况，EROFS采用的是固定大小为压缩输出，即可能会将这128k数据的0-7k压缩成4k，7-25k也压缩成4k，25-32k也压缩成4k，以此类推。那么读取文件中的任意4k数据，只要读取最少4k，最多8k（任意4k数据压缩后最多分布到两个block中）的压缩数据。相比squashfs，读放大问题明显减少了。



压缩之后保存数据之后的文件系统要处理的另外一个问题就是在读取的时候的解压缩策略。EROFS使用了两种策略，cached I/O 和 in-place I/O，前者会将数据先解压缩到inode对应到的一个Page Cache中，在读取的时候直接将请求导向这个Cache的数据即可。而后者的策略是利用了读取请求带有的Buffer，在读取的时候，VFS会先分配一些内存Pages来作为Buffer，EROFS这里就直接利用这些Buffer来保存解压缩之后的数据，避免额外的内存分配。

EROFS在解压缩的处理上面也使用了不少优化策略，下图是EROFS，(a)中原始的数据被压缩为两个压缩块，D5在压缩的过程中被拆分。如果是为了读取D3 和 D4。这个操作的步骤如下，1. 读取C0，这里只要解压缩这个C0就可以了，2. EROFS会分配三个连续的Pages来保存D0-D2，而D3和D4会直接利用请求使用的Page Cache，3. 压缩算法要求连续的内存，这里就使用vmap将两段内存映射一段连续的内存，4. 这里在处理的时候，要拷贝C0一次，避免C0在这个过程中被修改，另外C0也是使用Page Cache的Page来保存的，5. 解压缩操作完成之后，就可以vunmap，D0-D2的数据也可以释放。这样的基本策略有两个问题，一个是仍然需要分配物理Pages，另外一个就是使用vmap机制并不十分高效，EROFS有使用了另外的一些优化策略，

Per-CPU buffers，在解压缩后的数据不大于4个Page的时候，会将这些数据解压缩到per-CPU buffers中，然后在拷贝到Page Cache中，如下图中的(c)所示。通过重复利用这个per-CPU buffers来减少内存分配的动作；

Rolling decompression，这里的基本思路分配一个64KB/16Pages的大小。一些情况下吗在解压缩了64KB之后，可以重新利用这段内存，即一种循环利用的方式。



下面我们看一下对EROFS的性能评估

参与评估文件系统一共有4个，分别是EROFS、Squashfs、Btrfs、Ext4。在这里面EROFS和Squarshfs是可压缩的只读文件系统，btrfs虽然有对可压缩的支持，但并不是只读文件系统，ext4是android默认的文件系统。



第一个测试项目Fio性能测试比较, 测试项目分别是顺序读，随机读和分散读。需要解释一下：enwik9是标准测试数据，android是以android framework为主要内容的数据，两者差别主要是在压缩比上。大家可以看到顺序读表现最好的依然是老牌的ext4，但是erofs明显好于squashfs， squashfs在加上谷歌打patch之后，性能提升很多（红色柱子），但依然还是比erofs差。



随机读上由于有缓存的作用，squashfs看起来表现更好，但实质会造成更大的读放大问题，特别是块比较大的128KB squashfs。4KB的erofs和ext4还是比较相当的。



分散读主要是要排除掉读缓存的效果，测试方法是每128kb只读取前4kb，结果是显然squashfs因为读放大很多，导致了性能最差。但是erofs的表现没有比ext4差很多，考虑到它本身还压缩了，已经非常不错。



第二个测试项目是压缩比率测试，可以看到如果作为android系统分区使用的话，erofs因为使用LZ4压缩算法，压缩比率在70%；如果面对纯文本数据enwik9，erofs的压缩比率会更高一些达到56%



第三个测试项目是读放大测试，测试内容是文件系统读取10MB文件内容，测试实际块设备读取数量。数值大于10表示有读放大，越大则读放大问题越明显。数值小于10则表示没有读放大问题。顺序读由于采用了压缩，因此压缩文件系统都没有读放大问题，所以顺序读是最有利于压缩文件系统。随机读和分散读，压缩文件系统都有不同程度的读放大，但是erofs表现明显好于squashfs。