Enhanced Read-Only File System - EROFS

Overview

EROFS file-system stands for Enhanced Read-Only File System. Different from other read-only file systems, it aims to be designed for flexibility, scalability, but be kept simple and high performance.

It is designed as a better filesystem solution for the following scenarios:

- read-only storage media or
- part of a fully trusted read-only solution, which means it needs to be immutable and bit-for-bit identical to the official golden image for their releases due to security and other considerations and
- hope to minimize extra storage space with guaranteed end-to-end performance by using compact layout, transparent file compression and direct access, especially for those embedded devices with limited memory and high-density hosts with numerous containers;

Here is the main features of EROFS:

- Little endian on-disk design;
- Currently 4KB block size (nobh) and therefore maximum 16TB address space;
- Metadata & data could be mixed by design;
- 2 inode versions for different requirements:

Inode metadata size	32 bytes	64 bytes
Max file size	4 GB	16 EB (also limited by max. vol size)
Max uids/gids	65536	4294967296
Per-inode timestamp	no	yes (64 + 32-bit timestamp)
Max hardlinks	65536	4294967296
Metadata reserved	4 bytes	14 bytes

- Support extended attributes (xattrs) as an option;
- Support xattr inline and tail-end data inline for all files;
- Support POSIX.1e ACLs by using xattrs;
- Support transparent data compression as an option: LZ4 algorithm with the fixed-sized output compression for high performance;
- Multiple device support for multi-layer container images.

The following git tree provides the file system user-space tools under development (ex, formatting tool mkfs.erofs):

- git://git.kernel.org/pub/scm/linux/kernel/git/xiang/erofs-utils.git

Bugs and patches are welcome, please kindly help us and send to the following linux-erofs mailing list:

- linux-erofs mailing list <linux-erofs@lists.ozlabs.org>

Mount options

(no)user_xattr	Setup Extended User Attributes. Note: xattr is enabled by default if CONFIG_EROFS_FS_XATTR is selected.
(no)acl	Setup POSIX Access Control List. Note: acl is enabled by default if CONFIG_EROFS_FS_POSIX_ACL is selected.
cache_strategy=%s	Select a strategy for cached decompression from now on: disabled: In-place I/O decompression only; readahead: Cache the last incomplete compressed physical cluster for further reading. It still does in-place I/O decompression for the rest compressed physical clusters; readaround: Cache the both ends of incomplete compressed physical clusters for further reading. It still does in-place I/O decompression for the rest compressed physical clusters.
dax={always,never}	Use direct access (no page cache). See Documentation/filesystems/dax.rst.
dax	A legacy option which is an alias for `dax=always`.
device=%s	Specify a path to an extra device to be used together.

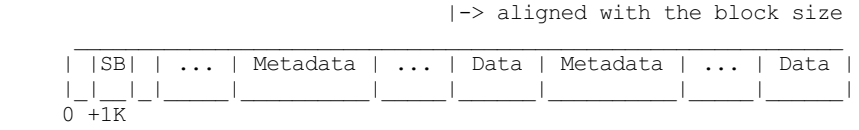
Sysfs Entries

Information about mounted erofs file systems can be found in /sys/fs/erofs. Each mounted filesystem will have a directory in /sys/fs/erofs based on its device name (i.e., /sys/fs/erofs/sda). (see also Documentation/ABI/testing/sysfs-fs-erofs)

On-disk details

Summary

Different from other read-only file systems, an EROFS volume is designed to be as simple as possible:



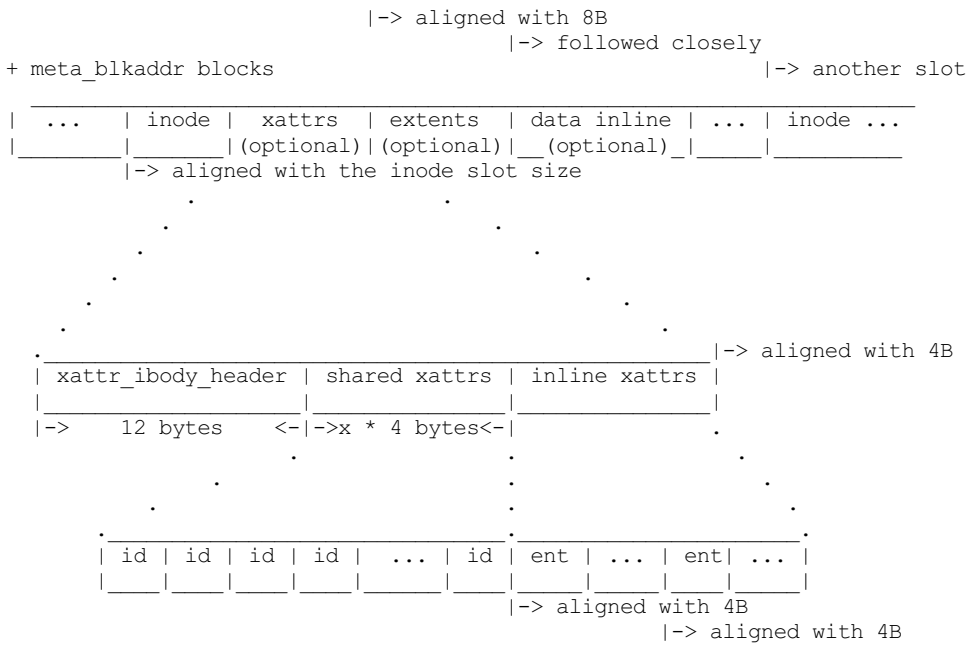
All data areas should be aligned with the block size, but metadata areas may not. All metadatas can be now observed in two different spaces (views):

1. Inode metadata space

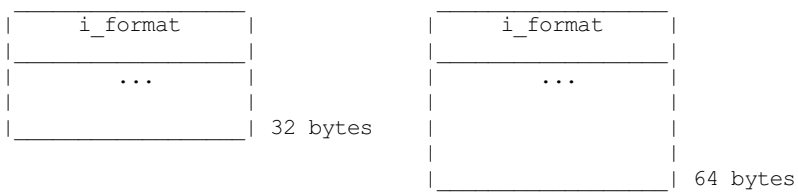
   Each valid inode should be aligned with an inode slot, which is a fixed value (32 bytes) and designed to be kept in line with compact inode size.

   Each inode can be directly found with the following formula:

   inode offset = meta_blkaddr * block_size + 32 * nid



   Inode could be 32 or 64 bytes, which can be distinguished from a common field which all inode versions have -- i_format:



   Xattrs, extents, data inline are followed by the corresponding inode with proper alignment, and they could be optional for different data mappings. _currently_ total 5 data layouts are supported:

0	flat file data without data inline (no extent);
1	fixed-sized output data compression (with non-compacted indexes);
2	flat file data with tail packing data inline (no extent);
3	fixed-sized output data compression (with compacted indexes, v5.3+);
4	chunk-based file (v5.15+).

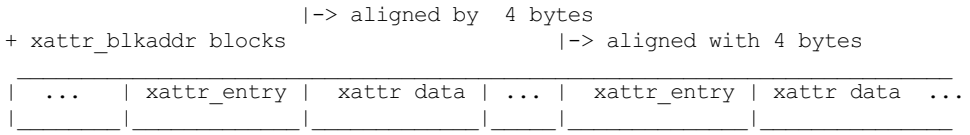
   The size of the optional xattrs is indicated by i_xattr_count in inode header. Large xattrs or xattrs shared by many

different files can be stored in shared xattrs metadata rather than inlined right after inode.

2. Shared xattrs metadata space

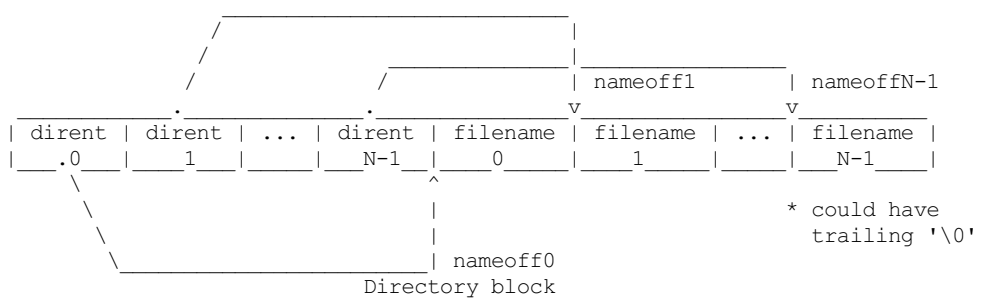
   Shared xattrs space is similar to the above inode space, started with a specific block indicated by xattr_blkaddr, organized one by one with proper align.

   Each share xattr can also be directly found by the following formula:
   xattr offset = xattr_blkaddr * block_size + 4 * xattr_id



Directories

All directories are now organized in a compact on-disk format. Note that each directory block is divided into index and name areas in order to support random file lookup, and all directory entries are _strictly_ recorded in alphabetical order in order to support improved prefix binary search algorithm (could refer to the related source code).



Note that apart from the offset of the first filename, nameoff0 also indicates the total number of directory entries in this block since it is no need to introduce another on-disk field at all.

Chunk-based file

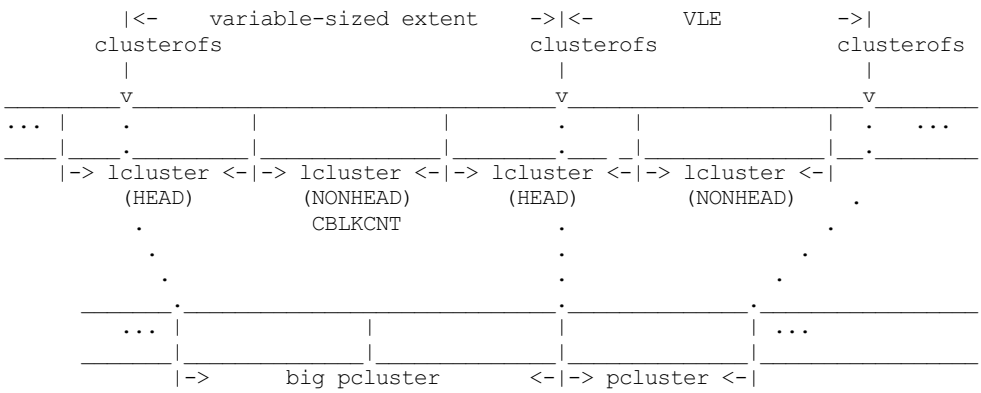
In order to support chunk-based data deduplication, a new inode data layout has been supported since Linux v5.15: Files are split in equal-sized data chunks with `extents` area of the inode metadata indicating how to get the chunk data: these can be simply as a 4-byte block address array or in the 8-byte chunk index form (see struct erofs_inode_chunk_index in erofs_fs.h for more details.)

By the way, chunk-based files are all uncompressed for now.

Data compression

EROFS implements LZ4 fixed-sized output compression which generates fixed-sized compressed data blocks from variable-sized input in contrast to other existing fixed-sized input solutions. Relatively higher compression ratios can be gotten by using fixed-sized output compression since nowadays popular data compression algorithms are mostly LZ77-based and such fixed-sized output approach can be benefited from the historical dictionary (aka. sliding window).

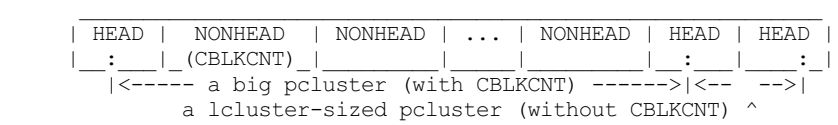
In details, original (uncompressed) data is turned into several variable-sized extents and in the meanwhile, compressed into physical clusters (pclusters). In order to record each variable-sized extent, logical clusters (lclusters) are introduced as the basic unit of compress indexes to indicate whether a new extent is generated within the range (HEAD) or not (NONHEAD). Lclusters are now fixed in block size, as illustrated below:



A physical cluster can be seen as a container of physical compressed blocks which contains compressed data. Previously, only lcluster-sized (4KB) pclusters were supported. After big pcluster feature is introduced (available since Linux v5.13), pcluster can be a multiple of lcluster size.

For each HEAD lcluster, clusterofs is recorded to indicate where a new extent starts and blkaddr is used to seek the compressed data. For each NONHEAD lcluster, delta0 and delta1 are available instead of blkaddr to indicate the distance to its HEAD lcluster and the next HEAD lcluster. A PLAIN lcluster is also a HEAD lcluster except that its data is uncompressed. See the comments around "struct z_erofs_vle_decompressed_index" in erofs_fs.h for more details.

If big pcluster is enabled, pcluster size in lclusters needs to be recorded as well. Let the delta0 of the first NONHEAD lcluster store the compressed block count with a special flag as a new called CBLKCNT NONHEAD lcluster. It's easy to understand its delta0 is constantly 1, as illustrated below:



If another HEAD follows a HEAD lcluster, there is no room to record CBLKCNT, but it's easy to know the size of such pcluster is 1 lcluster as well.