Directory Entries

In an ext4 filesystem, a directory is more or less a flat file that maps an arbitrary byte string (usually ASCII) to an inode number on the filesystem. There can be many directory entries across the filesystem that reference the same inode number--these are known as hard links, and that is why hard links cannot reference files on other filesystems. As such, directory entries are found by reading the data block(s) associated with a directory file for the particular directory entry that is desired.

Linear (Classic) Directories

By default, each directory lists its entries in an "almost-linear" array. I write "almost" because it's not a linear array in the memory sense because directory entries are not split across filesystem blocks. Therefore, it is more accurate to say that a directory is a series of data blocks and that each block contains a linear array of directory entries. The end of each per-block array is signified by reaching the end of the block; the last entry in the block has a record length that takes it all the way to the end of the block. The end of the entire directory is of course signified by reaching the end of the file. Unused directory entries are signified by inode = 0. By default the filesystem uses `struct ext4_dir_entry_2` for directory entries unless the "filetype" feature flag is not set, in which case it uses `struct ext4_dir_entry`.

The original directory entry format is `struct ext4_dir_entry`, which is at most 263 bytes long, though on disk you'll need to reference `dirent.rec_len` to know for sure.

Offset	Size	Name	Description
0x0	__le32	inode	Number of the inode that this directory entry points to.
0x4	__le16	rec_len	Length of this directory entry. Must be a multiple of 4.
0x6	__le16	name_len	Length of the file name.
0x8	char	name[EXT4_NAME_LEN]	File name.

Since file names cannot be longer than 255 bytes, the new directory entry format shortens the name_len field and uses the space for a file type flag, probably to avoid having to load every inode during directory tree traversal. This format is `ext4_dir_entry_2`, which is at most 263 bytes long, though on disk you'll need to reference `dirent.rec_len` to know for sure.

Offset	Size	Name	Description
0x0	__le32	inode	Number of the inode that this directory entry points to.
0x4	__le16	rec_len	Length of this directory entry.
0x6	__u8	name_len	Length of the file name.
0x7	__u8	file_type	File type code, see ftype table below.
0x8	char	name[EXT4_NAME_LEN]	File name.

The directory file type is one of the following values:

Value	Description
0x0	Unknown.
0x1	Regular file.
0x2	Directory.
0x3	Character device file.
0x4	Block device file.
0x5	FIFO.
0x6	Socket.
0x7	Symbolic link.

To support directories that are both encrypted and casefolded directories, we must also include hash information in the directory entry. We append `ext4_extended_dir_entry_2` to `ext4_dir_entry_2` except for the entries for dot and dotdot, which are kept the same. The structure follows immediately after `name` and is included in the size listed by `rec_len` If a directory entry uses this extension, it may be up to 271 bytes.

Offset	Size	Name	Description
0x0	__le32	hash	The hash of the directory name
0x4	__le32	minor_hash	The minor hash of the directory name

In order to add checksums to these classic directory blocks, a phony `struct ext4_dir_entry` is placed at the end of each leaf block to hold the checksum. The directory entry is 12 bytes long. The inode number and name_len fields are set to zero to fool old software into ignoring an apparently empty directory entry, and the checksum is stored in the place where the name normally goes. The structure is `struct ext4_dir_entry_tail`:

Offset	Size	Name	Description
0x0	__le32	det_reserved_zero1	Inode number, which must be zero.
0x4	__le16	det_rec_len	Length of this directory entry, which must be 12.
0x6	__u8	det_reserved_zero2	Length of the file name, which must be zero.

Offset	Size	Name	Description
0x7	__u8	det_reserved_ft	File type, which must be 0xDE.
0x8	__le32	det_checksum	Directory leaf block checksum.

The leaf directory block checksum is calculated against the FS UUID, the directory's inode number, the directory's inode generation number, and the entire directory entry block up to (but not including) the fake directory entry.

Hash Tree Directories

A linear array of directory entries isn't great for performance, so a new feature was added to ext3 to provide a faster (but peculiar) balanced tree keyed off a hash of the directory entry name. If the EXT4_INDEX_FL (0x1000) flag is set in the inode, this directory uses a hashed btree (htree) to organize and find directory entries. For backwards read-only compatibility with ext2, this tree is actually hidden inside the directory file, masquerading as "empty" directory data blocks! It was stated previously that the end of the linear directory entry table was signified with an entry pointing to inode 0; this is (ab)used to fool the old linear-scan algorithm into thinking that the rest of the directory block is empty so that it moves on.

The root of the tree always lives in the first data block of the directory. By ext2 custom, the '.' and '..' entries must appear at the beginning of this first block, so they are put here as two `struct ext4_dir_entry_2`s and not stored in the tree. The rest of the root node contains metadata about the tree and finally a hash->block map to find nodes that are lower in the htree. If `dx_root.info.indirect_levels` is non-zero then the htree has two levels; the data block pointed to by the root node's map is an interior node, which is indexed by a minor hash. Interior nodes in this tree contains a zeroed out `struct ext4_dir_entry_2` followed by a minor_hash->block map to find leafe nodes. Leaf nodes contain a linear array of all `struct ext4_dir_entry_2`; all of these entries (presumably) hash to the same value. If there is an overflow, the entries simply overflow into the next leaf node, and the least-significant bit of the hash (in the interior node map) that gets us to this next leaf node is set.

To traverse the directory as a htree, the code calculates the hash of the desired file name and uses it to find the corresponding block number. If the tree is flat, the block is a linear array of directory entries that can be searched; otherwise, the minor hash of the file name is computed and used against this second block to find the corresponding third block number. That third block number will be a linear array of directory entries.

To traverse the directory as a linear array (such as the old code does), the code simply reads every data block in the directory. The blocks used for the htree will appear to have no entries (aside from '.' and '..') and so only the leaf nodes will appear to have any interesting content.

The root of the htree is in `struct dx_root`, which is the full length of a data block:

Offset	Type	Name	Description
0x0	__le32	dot.inode	inode number of this directory.
0x4	__le16	dot.rec_len	Length of this record, 12.
0x6	u8	dot.name_len	Length of the name, 1.
0x7	u8	dot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).
0x8	char	dot.name[4]	".\0\0\0"
0xC	__le32	dotdot.inode	inode number of parent directory.
0x10	__le16	dotdot.rec_len	block_size - 12. The record length is long enough to cover all htree data.
0x12	u8	dotdot.name_len	Length of the name, 2.
0x13	u8	dotdot.file_type	File type of this entry, 0x2 (directory) (if the feature flag is set).
0x14	char	dotdot_name[4]	"..\0\0"
0x18	__le32	struct dx_root_info.reserved_zero	Zero.
0x1C	u8	struct dx_root_info.hash_version	Hash type, see dirhash table below.
0x1D	u8	struct dx_root_info.info_length	Length of the tree information, 0x8.
0x1E	u8	struct dx_root_info.indirect_levels	Depth of the htree. Cannot be larger than 3 if the INCOMPAT_LARGEDIR feature is set; cannot be larger than 2 otherwise.
0x1F	u8	struct dx_root_info.unused_flags	
0x20	__le16	limit	Maximum number of dx_entries that can follow this header, plus 1 for the header itself.
0x22	__le16	count	Actual number of dx_entries that follow this header, plus 1 for the header itself.
0x24	__le32	block	The block number (within the directory file) that goes with hash=0.
0x28	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

The directory hash is one of the following values:

Value	Description
0x0	Legacy.

Value	Description
0x1	Half MD4.
0x2	Tea.
0x3	Legacy, unsigned.
0x4	Half MD4, unsigned.
0x5	Tea, unsigned.
0x6	Siphash.

Interior nodes of an htree are recorded as `struct dx_node`, which is also the full length of a data block:

Offset	Type	Name	Description
0x0	__le32	fake.inode	Zero, to make it look like this entry is not in use.
0x4	__le16	fake.rec_len	The size of the block, in order to hide all of the dx_node data.
0x6	u8	name_len	Zero. There is no name for this "unused" directory entry.
0x7	u8	file_type	Zero. There is no file type for this "unused" directory entry.
0x8	__le16	limit	Maximum number of dx_entries that can follow this header, plus 1 for the header itself.
0xA	__le16	count	Actual number of dx_entries that follow this header, plus 1 for the header itself.
0xE	__le32	block	The block number (within the directory file) that goes with the lowest hash value of this block. This value is stored in the parent block.
0x12	struct dx_entry	entries[0]	As many 8-byte `struct dx_entry` as fits in the rest of the data block.

The hash maps that exist in both `struct dx_root` and `struct dx_node` are recorded as `struct dx_entry`, which is 8 bytes long:

Offset	Type	Name	Description
0x0	__le32	hash	Hash code.
0x4	__le32	block	Block number (within the directory file, not filesystem blocks) of the next node in the htree.

(If you think this is all quite clever and peculiar, so does the author.)

If metadata checksums are enabled, the last 8 bytes of the directory block (precisely the length of one dx_entry) are used to store a `struct dx_tail`, which contains the checksum. The `limit` and `count` entries in the dx_root/dx_node structures are adjusted as necessary to fit the dx_tail into the block. If there is no space for the dx_tail, the user is notified to run e2fsck -D to rebuild the directory index (which will ensure that there's space for the checksum. The dx_tail structure is 8 bytes long and looks like this:

Offset	Type	Name	Description
0x0	u32	dt_reserved	Zero.
0x4	__le32	dt_checksum	Checksum of the htree directory block.

The checksum is calculated against the FS UUID, the htree index header (dx_root or dx_node), all of the htree indices (dx_entry) that are in use, and the tail block (dx_tail).