Layout

The layout of a standard block group is approximately as follows (each of these fields is discussed in a separate section below):

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1024 bytes	1 block	many blocks	many blocks	1 block	1 block	many blocks	many more blocks

For the special case of block group 0, the first 1024 bytes are unused, to allow for the installation of x86 boot sectors and other oddities. The superblock will start at offset 1024 bytes, whichever block that happens to be (usually 0). However, if for some reason the block size = 1024, then block 0 is marked in use and the superblock goes in block 1. For all other block groups, there is no padding.

The ext4 driver primarily works with the superblock and the group descriptors that are found in block group 0. Redundant copies of the superblock and group descriptors are written to some of the block groups across the disk in case the beginning of the disk gets trashed, though not all block groups necessarily host a redundant copy (see following paragraph for more details). If the group does not have a redundant copy, the block group begins with the data block bitmap. Note also that when the filesystem is freshly formatted, mkfs will allocate "reserve GDT block" space after the block group descriptors and before the start of the block bitmaps to allow for future expansion of the filesystem. By default, a filesystem is allowed to increase in size by a factor of 1024x over the original filesystem size.

The location of the inode table is given by `grp.bg_inode_table_*`. It is continuous range of blocks large enough to contain `sb.s_inodes_per_group * sb.s_inode_size` bytes.

As for the ordering of items in a block group, it is generally established that the super block and the group descriptor table, if present, will be at the beginning of the block group. The bitmaps and the inode table can be anywhere, and it is quite possible for the bitmaps to come after the inode table, or for both to be in different groups (flex_bg). Leftover space is used for file data blocks, indirect block maps, extent tree blocks, and extended attributes.

Flexible Block Groups

Starting in ext4, there is a new feature called flexible block groups (flex_bg). In a flex_bg, several block groups are tied together as one logical block group; the bitmap spaces and the inode table space in the first block group of the flex_bg are expanded to include the bitmaps and inode tables of all other block groups in the flex_bg. For example, if the flex_bg size is 4, then group 0 will contain (in order) the superblock, group descriptors, data block bitmaps for groups 0-3, inode bitmaps for groups 0-3, inode tables for groups 0-3, and the remaining space in group 0 is for file data. The effect of this is to group the block group metadata close together for faster loading, and to enable large files to be continuous on disk. Backup copies of the superblock and group descriptors are always at the beginning of block groups, even if flex_bg is enabled. The number of block groups that make up a flex_bg is given by 2 ^ `sb.s_log_groups_per_flex`.

Meta Block Groups

Without the option META_BG, for safety concerns, all block group descriptors copies are kept in the first block group. Given the default 128MiB(2^27 bytes) block group size and 64-byte group descriptors, ext4 can have at most 2^27/64 = 2^21 block groups. This limits the entire filesystem size to 2^21 * 2^27 = 2^48bytes or 256TiB.

The solution to this problem is to use the metablock group feature (META_BG), which is already in ext3 for all 2.6 releases. With the META_BG feature, ext4 filesystems are partitioned into many metablock groups. Each metablock group is a cluster of block groups whose group descriptor structures can be stored in a single disk block. For ext4 filesystems with 4 KB block size, a single metablock group partition includes 64 block groups, or 8 GiB of disk space. The metablock group feature moves the location of the group descriptors from the congested first block group of the whole filesystem into the first group of each metablock group itself. The backups are in the second and last group of each metablock group. This increases the 2^21 maximum block groups limit to the hard limit 2^32, allowing support for a 512PiB filesystem.

The change in the filesystem format replaces the current scheme where the superblock is followed by a variable-length set of block group descriptors. Instead, the superblock and a single block group descriptor block is placed at the beginning of the first, second, and last block groups in a meta-block group. A meta-block group is a collection of block groups which can be described by a single block group descriptor block. Since the size of the block group descriptor structure is 32 bytes, a meta-block group contains 32 block groups for filesystems with a 1KB block size, and 128 block groups for filesystems with a 4KB blocksize. Filesystems can either be created using this new block group descriptor layout, or existing filesystems can be resized on-line, and the field s_first_meta_bg in the superblock will indicate the first block group using this new layout.

Please see an important note about `BLOCK_UNINIT` in the section about block and inode bitmaps.

Lazy Block Group Initialization

A new feature for ext4 are three block group descriptor flags that enable mkfs to skip initializing other parts of the block group metadata. Specifically, the INODE_UNINIT and BLOCK_UNINIT flags mean that the inode and block bitmaps for that group can be calculated and therefore the on-disk bitmap blocks are not initialized. This is generally the case for an empty block group or a block group containing only fixed-location block group metadata. The INODE_ZEROED flag means that the inode table has been initialized; mkfs will unset this flag and rely on the kernel to initialize the inode tables in the background.

By not writing zeroes to the bitmaps and inode table, mkfs time is reduced considerably. Note the feature flag is RO_COMPAT_GDT_CSUM, but the dumpe2fs output prints this as “uninit_bg”. They are the same thing.