The Second Extended File System (ext2fs)

CSCI780: Linux Kernel Internals
Fall 2001

Anne Shepherd

Background

- Early versions of Linux were based on Tannenbaum's Minix filesystem, but Minix proved inadequate as Linux matured.
- The Extended File System was created by Remy Card, Theodore T'so, and Stephen Tweedie. It removed Minix's limitations on max fs size and max file name size, but still had problems.
- In 1993/1994 Card, T'so, and Tweedie introduced the Second Extended Filesystem(ext2), which has become the most widely used Linux filesystem.

Ext2fs features

- Ext2fs supports standard Unix file types, and added support for long file names and very large partition sizes (on some architectures). It reserves some blocks for the superuser (5 per cent is the default) to allow sysadmins to recover from situations where user processes fill up the filesystem.
- Ext2fs also allows users to set file attributes, choose file semantics (BSD or SVR4), and logical block size (1k, 2k, or 4k).
- Fast symbolic links (which use no data blocks) are implemented: the target name is stored in the inode itself, not in a data block, up to a certain length (currently 60 characters).
- The fs keeps track of the filesystem state, so the filesystem checker, `ext2fsck`, can decide whether to check the filesystem on boot.

Ext2fs performance optimizations

- The ext2fs code was written with a view toward speeding up i/o.
- Ext2fs uses readaheads to take advantage of buffer cache management. Readaheads are done both on regular files and on directory reads.
- Allocation optimizations are used to reduce disk head seeks. Related inodes and data blocks are clustered into Block Groups; the kernel tries to allocate disk blocks for a file within that file's own block group.
- The filesystem preallocates a group of adjacent blocks when writing data to a file. The fs's authors claim that this results in good write performance under heavy load; and the fact that it allocates contiguous blocks to files speeds up reads as well.

Physical Layout of an ext2 partition

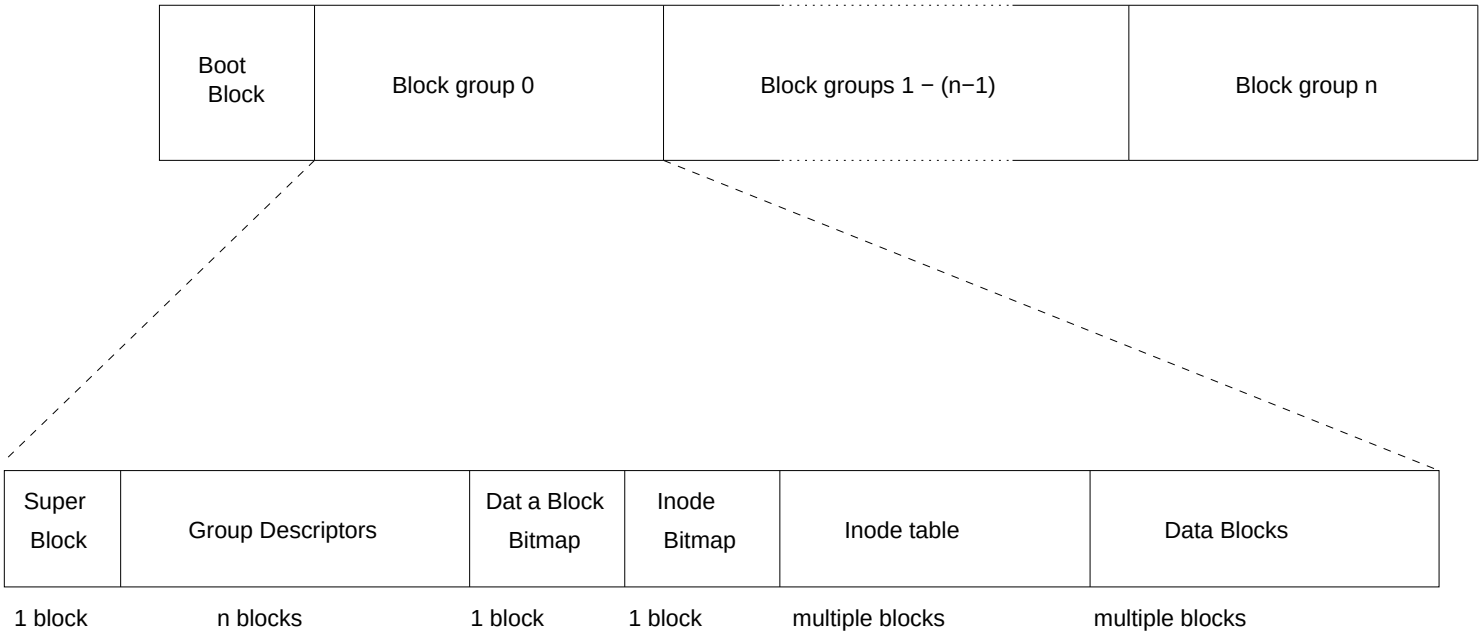
The first block in any ext2 partition is reserved for the partition's boot sector. (The rest of the partition is made up of block groups). All block groups are the same size. Because they are stored sequentially, the kernel can easily find a block group on a disk just from its index number.

The system tries to keep all the data blocks belonging to a given file in its home block group. The idea here is to reduce file fragmentation.

Every block group contains some redundant information:

- a copy of the file system superblock
- copies of all group descriptors

The kernel uses only the versions of these items in block 0. They are replicated in the other blocks for the use of the filesystem checker. If, when it performs a consistency check on the file system, it finds that the block 0 superblock has been corrupted, it can use one of the other copies to replace it (and the group descriptors) and bring the file system back to an acceptable state.



* Figure adapted from Bovet et al, Understanding the Linux Kernel

The ext2 physical superblock

(from `fs/ext2_fs.h`:

```
struct ext2_super_block {
   __u32   s_inodes_count;         /* Inodes count */
   __u32   s_blocks_count;         /* Blocks count */
   __u32   s_r_blocks_count;       /* Reserved blocks count */
   __u32   s_free_blocks_count;    /* Free blocks count */
   __u32   s_free_inodes_count;    /* Free inodes count */
   __u32   s_first_data_block;     /* First Data Block [always 1 --pls]*/
   __u32   s_log_block_size;       /* Block size */
/* s_log_block_size expresses block size as a power of 2, using 1024
 * bytes as the unit. So 0 would be a 1024-byte block, 1 a 2048-byte
 * block, etc. Note that the fragmentation stuff (below) is not yet
 * implemented  --pls */
   __s32   s_log_frag_size;        /* Fragment size */
   __u32   s_blocks_per_group;     /* # Blocks per group */
   __u32   s_frags_per_group;      /* # Fragments per group */
   __u32   s_inodes_per_group;     /* # Inodes per group */
   __u32   s_mtime;                /* [last] Mount time */
   __u32   s_wtime;                /* Write time */
```

```
/* fields below cause the filesystem checker (ext2fsck) to
 * run after a predefined number of mounts or a certain amount of
 * time has passes since the last check.  --pls */
   __u16   s_mnt_count;            /* Mount count */
   __s16   s_max_mnt_count;        /* Maximal mount count [before check]*/
   __u16   s_magic;                /* Magic signature */
   __u16   s_state;                /* File system state */
   __u16   s_errors;               /* Behaviour when detecting errors */
   __u16   s_minor_rev_level;      /* minor revision level */
   __u32   s_lastcheck;            /* time of last check */
   __u32   s_checkinterval;        /* max. time between checks */
   __u32   s_creator_os;           /* OS */
   __u32   s_rev_level;            /* Revision level */
   __u16   s_def_resuid;           /* Default uid for reserved blocks*/
   __u16   s_def_resgid;           /* Default gid for reserved blocks */
  . . .
  __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/
  __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
  __u16   s_padding1;             /* Padding to the end of the block */
  __u32   s_reserved[204];
};
```

ext2 superblock operations (from `/include/linux/ext2_fs.h`)

```
static struct super_operations ext2_sops = {
        read_inode:     ext2_read_inode,
        write_inode:    ext2_write_inode,
        put_inode:      ext2_put_inode,
        delete_inode:   ext2_delete_inode,
        put_super:      ext2_put_super,
        write_super:    ext2_write_super,
        statfs:         ext2_statfs,
        remount_fs:     ext2_remount,
};
```

The ext2 group descriptor (from `/include/linux/ext2_fs.h`)

```
struct ext2_group_desc
{
    __u32   bg_block_bitmap;        /* [number of] Blocks bitmap block */
    __u32   bg_inode_bitmap;        /* [number of]Inodes bitmap block */
/* the next field actually holds the block number of the first block
 * of the inode table --pls*/
    __u32   bg_inode_table;         /* Inodes table block */
    __u16   bg_free_blocks_count;   /* Free blocks count */
    __u16   bg_free_inodes_count;   /* Free inodes count */
    __u16   bg_used_dirs_count;     /* Directories count */
    __u16   bg_pad;                 /* alignment to word boundary --pls*/
    __u32   bg_reserved[3];         /* nulls to pad out 24 bytes  --pls*/
};
```

Each bitmap must be stored in a single block, so the size of the block is relevant here: a 1k block can contain a bitmap mapping the states of 8192 blocks; a 4k block, 32,768 blocks.

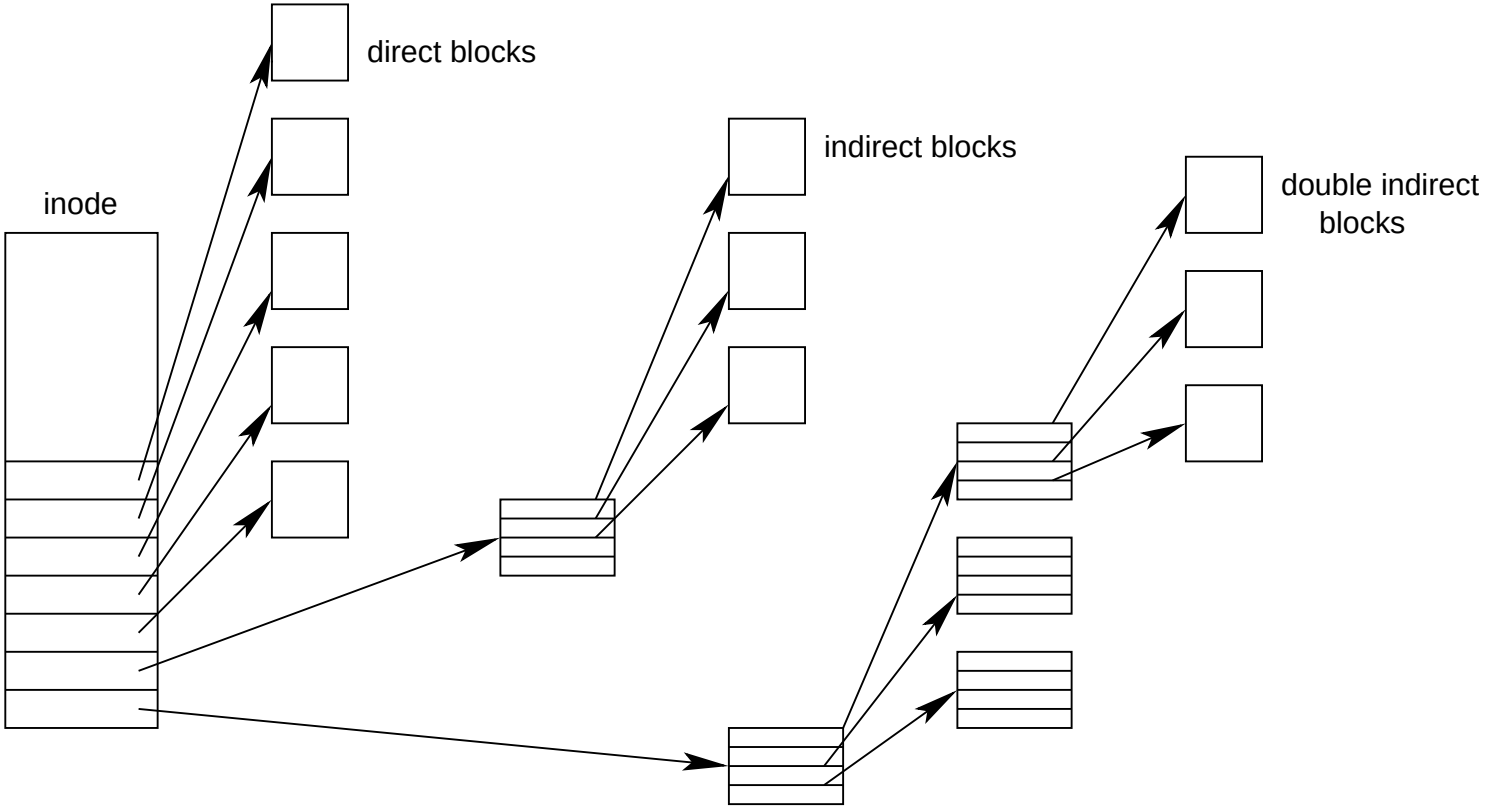
The ext2 physical inode (from `/include/linux/ext2_fs.h`)

```
 struct ext2_inode {
   __u16   i_mode;         /* File mode */
   __u16   i_uid;          /* Low 16 bits of Owner Uid */
/* The highest-order bit of i_size is not used, effectively limiting
 * the file size to 2GB, though there are ways of getting around this
 * on certain 64-bit architectures. --pls */
   __u32   i_size;         /* Size in bytes */
   __u32   i_atime;        /* [last] Access time */
   __u32   i_ctime;        /* Creation time */
   __u32   i_mtime;        /* Modification time */
   __u32   i_dtime;        /* Deletion Time */
   __u16   i_gid;          /* Low 16 bits of Group Id */
   __u16   i_links_count;  /* Links count */
/* Note that each file is currently stored in a whole number of
 * blocks, so an empty file will start out with one block.
 * This may change when/if fragmentation is implemented --pls*/
   __u32   i_blocks;       /* Blocks count */
   __u32   i_flags;        /* File flags */
   union...osd1;           /* big union to hold os-specific stuff --pls*/
   __u32   i_block[EXT2_N_BLOCKS];  /* Pointers to [data] blocks */
```

```
    __u32   i_generation;   /* File version (for NFS) */
    __u32   i_file_acl;     /* File ACL */
    __u32   i_dir_acl;      /* Directory ACL */
    __u32   i_faddr;        /* Fragment address */
    union...osd2,           /* big union to hold os-specific stuff --pls*/
};
```

Note that there is no inode number stored in the physical inode, even though the VFS layer required each file to have a unique inode number. This is because ext2 just uses the inode number as a key to retrieve the inode descriptor from disk–its value can be derived from the block group number and relative position inside the inode table.

All inodes are 128 bytes long, so a 1k block holds 8 inodes, and a 4k block wil contain 32 inodes. To calculate the size of your inode table, divide the total number of inodes in a group (found in the superblock's `s_inodes_per_group`) by the number of inodes per block.



* figure copied from Remy Card, "Design and implementation of the second extended file system"

The `i_block` field of the inode is an array that contains the addresses of the inode's associated data blocks. It has `EXT2_N_BLOCKS` = 15 (the default) entries. Indirection goes from direct through triple-indirect as follows:

- The first `EXT2_NDIR_BLOCK` = 12 of these directly contain addresses of data blocks.
- The next entry, at index `EXT2_IND_BLOCK` = 12, holds the address of the first indirect block, a block that contains addresses of data blocks.
- The entry found at index `EXT2_DIND_BLOCK` = 13 contains the address of the double indirect block, which in turn holds addresses of indirect blocks which hold addresses of data blocks.
- Finally, at index `EXT2_TIND_BLOCK` = 15 there is an address of a block containing addresses to blocks containing addresses to blocks.

A couple of things to note about block addressing:

- The way it is set up favors small files. If your file has 12 blocks or less, you need only look at the `i_block` array and go straight to the data block you need. With long files, several disk accesses may be necessary, depending on the level of indirection.
- Block size, of course, has a direct effect on the amount of indirection. Though triple-indirection may be necessary for a very large file on a filesystem with blocks of size 1k, with 4k block sizes, any file of 2GB or less (the maximum file size allowed by ext2 on 32 bit architectures) can be addressed with at most double indirection.

Data Blocks Addressing: Example

In this example, we will assume a block size of 1kb, so $b = 1024$ in the calculations below. We divide by 4 in the formulas below because each logical block number is stored in 4 bytes.

- The first 12 entries in the `i_block[]` array contain addresses to logical block numbers 0 through 11.
- The single-indirect entry at index 12 will eventually get you to blocks beginning with block 12 and going up to block $(\frac{b}{4} + 11)$, with b being the block size; so in our example, the single indirect block addresses blocks 12 - 267.
- The double-indirect entry at index 13 will address blocks from $(\frac{b}{4} + 12)$ through $((\frac{b}{4})^2 + \frac{b}{4} + 11)$; with a 1k block that's blocks 268 through 65803
- The triple-indirect entry at index 14 can address blocks from $((\frac{b}{4})^2 + \frac{b}{4} + 12)$ through $((\frac{b}{4})^3 + (\frac{b}{4})^2 + \frac{b}{4} + 12)$–that's a lot of blocks: from 65804 through (theoretically) 16843019.

Bitmaps

Ext2fs keeps track of allocated inodes and blocks using bitmaps. Each bitmap must fit in a single block.

The Group Descriptor contains fields containing the block numbers of the blocks bitmap and the inode bitmap for its group. The bitmaps encode the availability of blocks and inodes.

Because it would take up too much RAM to keep all bitmaps cached at all times, the filesystem instead uses two bitmap caches (one for blocks, one for inodes), each holding `EXT2_MAX_GROUP_LOADED` (currently defined as 8) bitmaps. The replacement algorithm is LRU, but this the LRU cache is maintained only if there are $>$ `EXT2_MAX_GROUP_LOADED` Block Groups in the filesystem.

Keeping filesystem information handy

A lot of ext2 disk data structure information is copied into RAM when the file system is mounted. Data that will be frequently used (for example, some superblock information and the group descriptors) will be kept always cached; it will not be removed from the buffer cache till the partition is unmounted. The kernel ensures this by keeping the usage counter on the buffers that holds the information greater than 0 at all times.

Other information, like block and inode bitmaps, is kept in the buffer cache in fixed-limit mode: there is room for a certain number of these in the cache. Old ones are flushed to disk when we run out of room.

Inodes and data blocks are cached while in use, then removed from cache in the usual course of events when they are no longer being used.

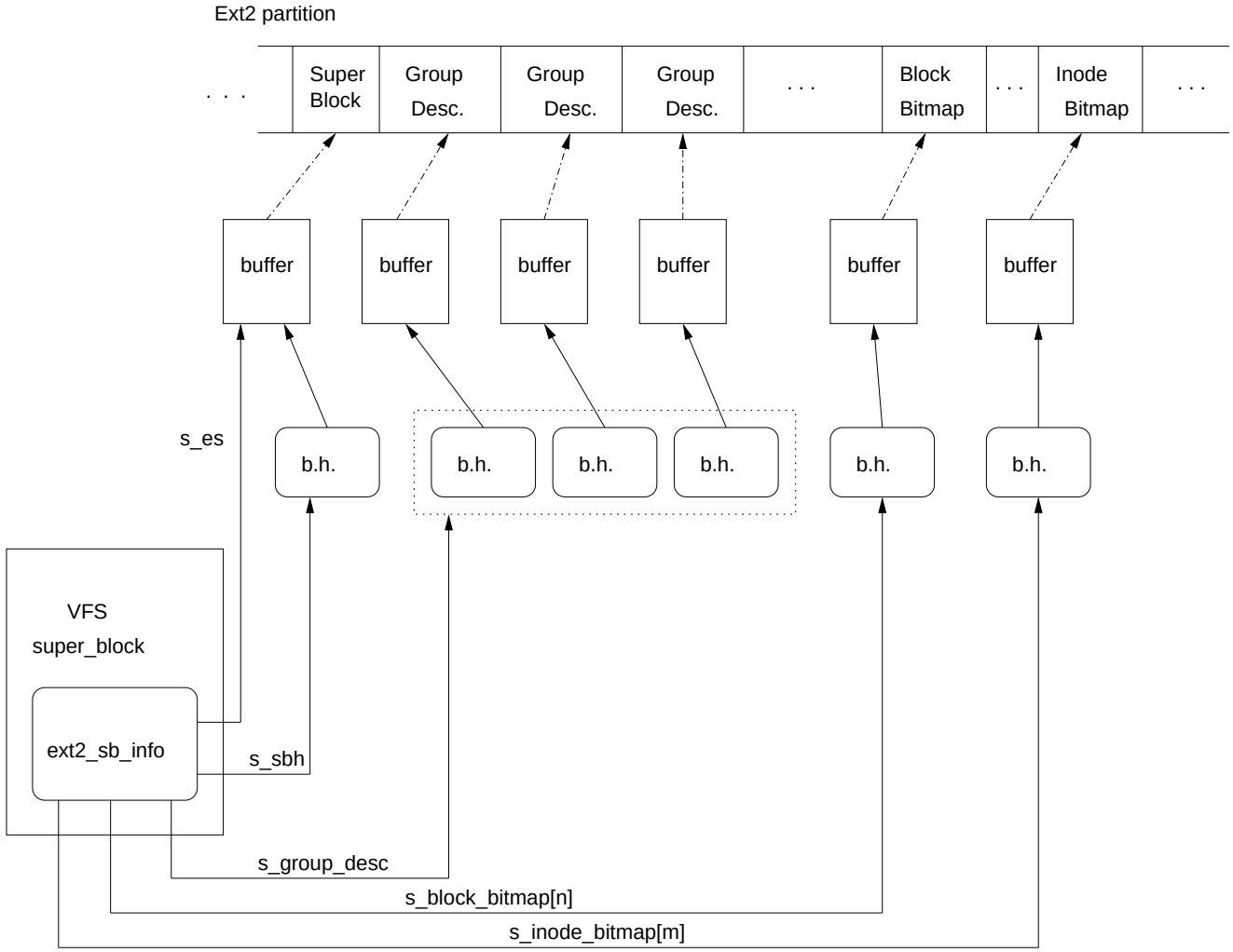
In some instances, special memory data structures are used to hold the information in the buffer cache. To contain needed superblock information the kernel uses the `ext2_sb_info` struct; for inode information, the `ext2_inode_info` struct. VFS data structures associated with these structs will have pointers pointing to them.

The `ext2_sb_info` struct

An ext2 partition is associated not only with an ext2 superblock, but with a VFS superblock as well. The `u` field of this structure is used to store filesystem-specific data. When the ext2 partition is mounted, the field will point to a struct `ext2_sb_info`, defined in `/include/linux/ext2_fs_sb.h`.

```
struct ext2_sb_info {
    unsigned long s_frag_size;      /* Size of a fragment in bytes */
    unsigned long s_frags_per_block;/* Number of fragments per block */
    unsigned long s_inodes_per_block;/* Number of inodes per block */
    unsigned long s_frags_per_group;/* Number of fragments in a group */
    unsigned long s_blocks_per_group;/* Number of blocks in a group */
    unsigned long s_inodes_per_group;/* Number of inodes in a group */
    unsigned long s_itb_per_group;
                  /* Number of inode table blocks per group */
    unsigned long s_gdb_count;      /* Number of group descriptor blocks */
    unsigned long s_desc_per_block; /* Number of group descriptors per block */
    unsigned long s_groups_count;   /* Number of groups in the fs */
    struct buffer_head * s_sbh;     /* Buffer containing the super block */
    struct ext2_super_block * s_es;
                   /* Pointer to the super block in the buffer */
```

```
    struct buffer_head ** s_group_desc;
    unsigned short s_loaded_inode_bitmaps;
    unsigned short s_loaded_block_bitmaps;
    unsigned long s_inode_bitmap_number[EXT2_MAX_GROUP_LOADED];
    struct buffer_head * s_inode_bitmap[EXT2_MAX_GROUP_LOADED];
    unsigned long s_block_bitmap_number[EXT2_MAX_GROUP_LOADED];
    struct buffer_head * s_block_bitmap[EXT2_MAX_GROUP_LOADED];
    unsigned long  s_mount_opt;
    uid_t s_resuid;
    gid_t s_resgid;
    unsigned short s_mount_state;
    unsigned short s_pad;
    int s_addr_per_block_bits;
    int s_desc_per_block_bits;
    int s_inode_size;
    int s_first_ino;
};
```



* Figure taken from Bovet et al, Understanding the Linux Kernel

Likewise, the inode is associated with the memory data structure `ext2_inode_info`, defined in `/include/linux/ext2_fs_i.h`. This struct includes, among other things:

- Fields in the physical inode that aren't mirrored in the VFS inode;
- The index into `block_group` at which the inode belongs;
- Fields used for preallocation;
- As-yet-unimplemented fragmentation information.

How files use disk blocks

Different types of files use disk blocks differently:

- Regular files start out empty–they need data blocks only when they begin to have data. They can also be emptied by a `truncate()` operation.
- Directories use data blocks to store information about the files under them. This information is stored in structs `ext2_dir_entry_2`.
- A symbolic link requires a data block only if the path name is longer than 60 characters (it uses a single block in this case); if the path name is less than 60 characters, it is stored in the inode and no date blocks are needed.
- Pipes, sockets, and device files keep all necessary information in the inode, so they use no data blocks.

The `ext2_dir_entry_2` struct

From `include/linux/ext2_fs.h`:

```
struct ext2_dir_entry_2 {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* [actual] Name length */
        __u8    file_type;       /* for example. 1 is regular file,
                                  *  2 is directory, 5 is named pipe--pls*/
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

The `rec_len` field represents the actual record length, rounded up to the next 4-byte value. It serves the same purpose as a "next" pointer in a linked list, by signifying the offset where we can find the next entry in the directory.

Example: reading a directory entry

Say you want to list all the entries in a directory, with their inode numbers:

```
/* toyls.c
 *
 * adapted from Stevens, Advanced Programming in the
 *    Unix Environment, p. 4
 *
 * Note that in the case of a symlink, the inode number stored in the
 * dirent is the link inode, not the target inode.  If you need
 * the target inode, use the stat utility with -l opiton:
 *
 *  stat -l <linkfilename>
 */
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
```

```
  DIR *dp;
  struct dirent *dirp;

  if(argc != 2){
    printf("\nUsage: toyls <directory name> \n");
    exit(1);
  } /* print usage and exit */

  if( (dp = opendir(argv[1])) != NULL){
    printf("\n");
    while( (dirp = readdir(dp)) != NULL) {
      printf("%s,  %li \n", dirp->d_name, dirp->d_ino);
    } /* while there's more to read */

    closedir(dp);
  } /* if we can open the dir */
  return 0;
}
```

The `dirent` structs looks like this:

```
struct dirent {
```

```
    long            d_ino;
    __kernel_off_t  d_off;
    unsigned short  d_reclen;   // --length of whole struct + k,
                                //   (0 <= k <= 3)  --pls
    char            d_name[256]; /* We must not include limits.h! */
};
struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
```

...and this one is used by the system to keep track of where we are in the list of directory entries:

```
struct getdents_callback {
        struct linux_dirent * current_dir;
        struct linux_dirent * previous;
        int count;
        int error;
};
```

The `readdir(3)` call is handled by glibc by repeated calls to `getdents()`. (It stores the current information statically and clobbers it at each call, so it is not thread-safe). Once in the kernel, we find ourselves in (from `/fs/readdir.c`):

```
asmlinkage long sys_getdents
(unsigned int fd, void * dirent, unsigned int count)
{
        struct file * file;
        struct linux_dirent * lastdirent;
        struct getdents_callback buf;
        int error;

        error = -EBADF;
        file = fget(fd);
        if (!file)
                goto out;

        buf.current_dir = (struct linux_dirent *) dirent;
        buf.previous = NULL;
        buf.count = count;
        buf.error = 0;
/* Note that filldir() is a function, defined in /fs/readdir.c --pls*/
```

```
        error = vfs_readdir(file, filldir, &buf);
```

Now we are in `vfs_readdir()`:

```
int vfs_readdir(struct file *file, filldir_t filler, void *buf)
{
        struct inode *inode = file->f_dentry->d_inode;
        int res = -ENOTDIR;
        if (!file->f_op || !file->f_op->readdir)
                goto out;
        down(&inode->i_sem);
        down(&inode->i_zombie);
        res = -ENOENT;
        if (!IS_DEADDIR(inode)) {
                lock_kernel();
                res = file->f_op->readdir(file, buf, filler);
                unlock_kernel();
        }
        up(&inode->i_zombie);
        up(&inode->i_sem);
out:
        return res;
```

```
}
```

...which takes us at last to (in `/fs/ext2/dir.c`):

```
static int ext2_readdir(struct file * filp,
                        void * dirent, filldir_t filldir)
{
     int error = 0;
     unsigned long offset, blk;
     int i, num, stored;
     struct buffer_head * bh, * tmp, * bha[16];
     struct ext2_dir_entry_2 * de;

     struct super_block * sb;
     int err;
     struct inode *inode = filp->f_dentry->d_inode;

     sb = inode->i_sb;

     stored = 0;
     bh = NULL;
```

```
/* mask to get just the offset --pls*/
     offset = filp->f_pos & (sb->s_blocksize - 1);

/* remember that the file in question is (or should be) a directory.
 * as we go into the loop, stored == 0 --pls*/

     while (!error && !stored && filp->f_pos < inode->i_size) {

/* right-shift by sb->s_blocksize_bits to get the block number (blk is
 * a long int) --pls*/
          blk = (filp->f_pos) >> EXT2_BLOCK_SIZE_BITS(sb);
/* going into the call below, the 0 means create = false --pls*/
          bh = ext2_bread (inode, blk, 0, &err);
```

ext2_bread() reads a data block assigned to an inode. It calls ext2_getblk() (from /fs/ext2/inode.c to find the appropriate memory block.

```
/* Going into this call, create == 0. Other params are as they
 * were going into ext2_bread(). --pls*/

struct buffer_head * ext2_getblk(struct inode * inode,
                         long block, int create, int * err)
```

```
{
        struct buffer_head dummy;
        int error;

        dummy.b_state = 0;
        dummy.b_blocknr = -1000;
/* upon return from the following call, dummy should hold the buffer
 * head we want. create is still 0. --pls*/
        error = ext2_get_block(inode, block, &dummy, create);
```

ext2_get_block() calls ext2_block_to_path() to find how deep the indirection tree is (if it is, e.g., at a double or triple indirect address) and get the appropriate addresses; then fills dummy with the appropriate information.

```
        *err = error;
/* buffer_mapped is a flag.  It should be true here. --pls*/
        if (!error && buffer_mapped(&dummy)) {
                struct buffer_head *bh;
```

The /fs/buffer.c function getblk(), as Joy ably explained in the last presentation, accesses the buffer for us and puts the result in bh.

```
                bh = getblk(dummy.b_dev, dummy.b_blocknr,
                            inode->i_sb->s_blocksize);
                . . .
                return bh;
        }
        return NULL;
}
```

back in `ext2_readdir()`. We do a readahead because users are quite likely to read more than one or two directory entries–in fact, very often this is used for `getdents()`, where all the entries will be read, so it makes sense to read ahead.

```
          . . .
          /* Do the readahead */
/* offset == 0 means that we just started, so now is the time to
 * read ahead. --pls*/
          if (!offset) {
/* EXT2_BLOCK_SIZE_BITS is blocksize in number of bits. For
 * a 1k block, it will = 10, for a 4k block, 12 So we will
 * end up reading ahead 8 blocks if we are  using 1k blocks, 2
 * blocks if using 4k blocks. --pls*/
                  for (i = 16 >> (EXT2_BLOCK_SIZE_BITS(sb) - 9),
```

```
                                        num = 0; i > 0; i--) {
                                tmp = ext2_getblk (inode, ++blk, 0, &err);
                                if (tmp && !buffer_uptodate(tmp) &&
                                    !buffer_locked(tmp))
                                        bha[num++] = tmp;
                                else
                                        brelse (tmp);
                        }
 /* if there are any that need to be marked up-to-date, ll_rw_block
  * will do that. --pls */

                        if (num) {
                                ll_rw_block (READA, num, bha);
                                for (i = 0; i < num; i++)
                                        brelse (bha[i]);
                        }
                }

revalidate:
                /* If the dir block has changed since the last call to
                 * readdir(2), then we might be pointing to an invalid
                 * dirent right now.  Scan from the start of the block
```

```
           * to make sure. */

/* version is set whent the inode is read and when an lseek has to
 * change the f_pos indicator to match the offset  --pls*/
          if (filp->f_version != inode->i_version) {
                  for (i = 0; i < sb->s_blocksize && i < offset; ) {
                          de = (struct ext2_dir_entry_2 *)
                                  (bh->b_data + i);
                          /* It's too expensive to do a full
                           * dirent test each time round this
                           * loop, but we do have to test at
                           * least that it is non-zero.  A
                           * failure will be detected in the
                           * dirent test below. */
                          if (le16_to_cpu(de->rec_len) <
                               EXT2_DIR_REC_LEN(1))
                                  break;
                          i += le16_to_cpu(de->rec_len);
                  }
                  offset = i;
/* the & ~ masks off everything below the block size level.  This is
 * then OR-ed with offset, to set the f_pos pointer so we can find the
```

```
* next entry. --pls*/
                filp->f_pos = (filp->f_pos & ~(sb->s_blocksize - 1))
                        | offset;
                filp->f_version = inode->i_version;
        }

        while (!error && filp->f_pos < inode->i_size
               && offset < sb->s_blocksize) {
                de = (struct ext2_dir_entry_2 *) (bh->b_data + offset);
                if (!ext2_check_dir_entry ("ext2_readdir", inode, de,
                                           bh, offset)) {
                        /* On error, skip the f_pos to the
                           next block. */
                        filp->f_pos =
                   (filp->f_pos | (sb->s_blocksize -
                                           1)) + 1;
                        brelse (bh);
                        return stored;
                }
                offset += le16_to_cpu(de->rec_len);
                if (le32_to_cpu(de->inode)) {
                        /* We might block in the next section
```

```
                         * if the data destination is
                         * currently swapped out.  So, use a
                         * version stamp to detect whether or
                         * not the directory has been modified
                         * during the copy operation.
                         */
                        unsigned long version = filp->f_version;
                        unsigned char d_type = DT_UNKNOWN;
                        . . .
/* filldir() fills in the blanks of the dirent structure. --pls*/
                        error = filldir(dirent, de->name,
                                        de->name_len,
                                        filp->f_pos,
                                        le32_to_cpu(de->inode),
                                        d_type);
                        if (error)
                                break;
                        if (version != filp->f_version)
                                goto revalidate;
                        stored ++;
                }
  /* now move the pointer down to the next record. --pls*/
```

```
                filp->f_pos += le16_to_cpu(de->rec_len);
        }
        offset = 0;
        brelse (bh);
    }
    UPDATE_ATIME(inode);
    return 0;
}
```

Back in sys_getdents():

```
        if (error < 0)
                goto out_putf;
        error = buf.error;
        lastdirent = buf.previous;
        if (lastdirent) {
/* put_user eventually gets us to an assembler routine that
 * will get this stuff into the user's space --pls*/
                put_user(file->f_pos, &lastdirent->d_off);
                error = count - buf.count;
        }
```

```
out_putf:
        fput(file);
out:
        return error;
}
```

Making and mounting an ext2 partition

Here's how to make and mount your very own ext2 partition on a floppy disk:

1. First, you have to format the floppy. Take a regular 1.4M DOS-formatted (or unformatted, but most of them are DOS these days) floppy, insert into the floppy drive, and type at the prompt:

   ```
   > fdformat /dev/fd0h1440
   ```

   (see the `fdformat` man page for more details).
2. When the formatting has finished, tell `mke2fs` to make the file system using default settings. On one of the department systems, type:

   ```
   > /sbin/mke2fs /dev/fd0
   ```

   These are the messages I got when I tried it:

   ```
   mke2fs 1.23, 15-Aug-2001 for EXT2 FS 0.5b, 95/08/09
   Filesystem label=
   OS type: Linux
   Block size=1024 (log=0)
   Fragment size=1024 (log=0)
   ```

```
184 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
184 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

3. Now mount the floppy:

```
> mount /mnt/floppy
```

   You can use `mount` to see what filesystems are currently mounted:

```
> mount
/dev/hda4 on / type ext2 (rw)
none on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
/dev/hda3 on /scratch type ext2 (rw)
```

```
none on /dev/pts type devpts (rw,gid=5,mode=620)
va:/var/spool/mail on /var/spool/mail type nfs (rw,addr=128.239.2.31)
automount(pid744) on /home type autofs  \
  (rw,fd=5,pgrp=744,minproto=2,maxproto=3)
mom:/export/home/mom2 on /home/mom2 type nfs
  (rw,rsize=8192,wsize=8192,soft,quota,nfsvers=2,addr=128.239.2.57) \
mom:/export/home/mom0 on /home/mom0 type nfs
  (rw,rsize=8192,wsize=8192,soft,quota,nfsvers=2,addr=128.239.2.57) \
mom:/export/home/scratch4 on /home/scratch4 type nfs
  (rw,rsize=8192,wsize=8192,soft,nfsvers=2,addr=128.239.2.57) \
va:/home/va on /home/va type nfs
  (rw,rsize=8192,wsize=8192,soft,addr=128.239.2.31)  \
/dev/fd0 on /mnt/floppy type ext2 (rw,nosuid,nodev,user=plshep)
```

4. When you want to unmount, type:

```
> umount /mnt/floppy
```

References

- Tin Siladin, *The Proc and Ext2 file systems*, Summer 1997
- Carter Rabase, *The Ext2 file system*, Spring 1999
- M. Beck, et al, *Linux Kernel Internals*, 2/e
- Remy Card, et al, *The Linux Kernel Book*
- Remy Card, et al, *Design and Implementation of the Second Extended Filesystem*
- Daniel Bovet, et al, *Understanding the Linux Kernel*
- Linux Cross Reference