The extended-2 filesystem overview

Gadi Oxman, tgud@tochnapc2.technion.ac.il

v0.1, August 3 1995

Contents

1 Preface

This document attempts to present an overview of the internal structure of the ext2 filesystem. It was written in summer 95, while I was working on the `ext2 filesystem editor project (EXT2ED)`.

In the process of constructing EXT2ED, I acquired knowledge of the various design aspects of the the ext2 filesystem. This document is a result of an effort to document this knowledge.

This is only the initial version of this document. It is obviously neither error-prone nor complete, but at least it provides a starting point.

In the process of learning the subject, I have used the following sources / tools:

- Experimenting with EXT2ED, as it was developed.
- The ext2 kernel sources:
  - The main ext2 include file, `/usr/include/linux/ext2_fs.h`
  - The contents of the directory `/usr/src/linux/fs/ext2`.
  - The VFS layer sources (only a bit).
- The slides: The Second Extended File System, Current State, Future Development, by `Remy Card`.

- The slides: Optimisation in File Systems, by `Stephen Tweedie`.
- The various ext2 utilities.

2 Introduction

The `Second Extended File System (Ext2fs)` is very popular among Linux users. If you use Linux, chances are that you are using the ext2 filesystem.

Ext2fs was designed by `Remy Card` and `Wayne Davison`. It was implemented by `Remy Card` and was further enhanced by `Stephen Tweedie` and `Theodore Ts´o`.

The ext2 filesystem is still under development. I will document here version 0.5a, which is distributed along with Linux 1.2.x. At this time of writing, the most recent version of Linux is 1.3.13, and the version of the ext2 kernel source is 0.5b. A lot of fancy enhancements are planned for the ext2 filesystem in Linux 1.3, so stay tuned.

3 A filesystem - Why do we need it ?

I thought that before we dive into the various small details, I'll reserve a few minutes for the discussion of filesystems from a general point of view.

A `filesystem` consists of two word - `file` and `system`.

Everyone knows the meaning of the word `file` - A bunch of data put somewhere. where ? This is an important question. I, for example, usually throw almost everything into a single drawer, and have difficulties finding something later.

This is where the `system` comes in - Instead of just throwing the data to the device, we generalize and construct a `system` which will virtualize for us a nice and ordered structure in which we could arrange our data in much the same way as books are arranged in a library. The purpose of the filesystem, as I understand it, is to make it easy for us to update and maintain our data.

Normally, by `mounting` filesystems, we just use the nice and logical virtual structure. However, the disk knows nothing about that - The device driver views the disk as a large continuous paper in which we can write notes wherever we wish. It is the task of the filesystem management code to store bookkeeping information which will serve the kernel for showing us the nice and ordered virtual structure.

In this document, we consider one particular administrative structure - The Second Extended Filesystem.

4 The Linux VFS layer

When Linux was first developed, it supported only one filesystem - The `Minix` filesystem. Today, Linux has the ability to support several filesystems concurrently. This was done by the introduction of another layer between the kernel and the filesystem code - The Virtual File System (VFS).

The kernel "speaks" with the VFS layer. The VFS layer passes the kernel's request to the proper filesystem management code. I haven't learned much of the VFS layer as I didn't need it for the construction of EXT2ED so that I can't elaborate on it. Just be aware that it exists.

5 About blocks and block groups

In order to ease management, the ext2 filesystem logically divides the disk into small units called `blocks`. A block is the smallest unit which can be allocated. Each block in the filesystem can be `allocated` or `free`.[1] The block size can be selected to be 1024, 2048 or 4096 bytes when creating the filesystem.

Ext2fs groups together a fixed number of sequential blocks into a `group block`. The resulting situation is that the filesystem is managed as a series of group blocks. This is done in order to keep related information physically close on the disk and to ease the management task. As a result, much of the filesystem management reduces to management of a single blocks group.

6 The view of inodes from the point of view of a blocks group

Each file in the filesystem is reserved a special `inode`. I don't want to explain inodes now. Rather, I would like to treat it as another resource, much like a `block` - Each blocks group contains a limited number of inode, while any specific inode can be `allocated` or `unallocated`.

7 The group descriptors

Each blocks group is accompanied by a `group descriptor`. The group descriptor summarizes some necessary information about the specific group block. Follows the definition of the group descriptor, as defined in /usr/include/linux/ext2_fs.h:

```
struct ext2_group_desc
{
        __u32   bg_block_bitmap;        /* Blocks bitmap block */
        __u32   bg_inode_bitmap;        /* Inodes bitmap block */
        __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;
        __u32   bg_reserved[3];
};
```

[1] The Ext2fs source code refers to the concept of `fragments`, which I believe are supposed to be sub-block allocations. As far as I know, fragments are currently unsupported in Ext2fs.

The last three variables: `bg_free_blocks_count, bg_free_inodes_count and bg_used_dirs_count` provide statistics about the use of the three resources in a blocks group - The `blocks`, the `inodes` and the `directories`. I believe that they are used by the kernel for balancing the load between the various blocks groups.

`bg_block_bitmap` contains the block number of the `block allocation bitmap block`. This is used to allocate / deallocate each block in the specific blocks group.

`bg_inode_bitmap` is fully analogous to the previous variable - It contains the block number of the `inode allocation bitmap block`, which is used to allocate / deallocate each specific inode in the filesystem.

`bg_inode_table` contains the block number of the start of the `inode table of the current blocks group`. The `inode table` is just the actual inodes which are reserved for the current block.

The block bitmap block, inode bitmap block and the inode table are created when the filesystem is created.

The group descriptors are placed one after the other. Together they make the `group descriptors table`.

Each blocks group contains the entire table of group descriptors in its second block, right after the superblock. However, only the first copy (in group 0) is actually used by the kernel. The other copies are there for backup purposes and can be of use if the main copy gets corrupted.

8 The block bitmap allocation block

Each blocks group contains one special block which is actually a map of the entire blocks in the group, with respect to their allocation status. Each `bit` in the block bitmap indicated whether a specific block in the group is used or free.

The format is actually quite simple - Just view the entire block as a series of bits. For example,

Suppose the block size is 1024 bytes. As such, there is a place for 1024*8=8192 blocks in a group block. This number is one of the fields in the filesystem's `superblock`, which will be explained later.

- Block 0 in the blocks group is managed by bit 0 of byte 0 in the bitmap block.
- Block 7 in the blocks group is managed by bit 7 of byte 0 in the bitmap block.
- Block 8 in the blocks group is managed by bit 0 of byte 1 in the bitmap block.
- Block 8191 in the blocks group is managed by bit 7 of byte 1023 in the bitmap block.

A value of "`1`" in the appropriate bit signals that the block is allocated, while a value of "`0`" signals that the block is unallocated.

You will probably notice that typically, all the bits in a byte contain the same value, making the byte's value `0` or `0ffh`. This is done by the kernel on purpose in order to group related data in physically close blocks, since the physical device is usually optimized to handle such a close relationship.

9 The inode allocation bitmap

The format of the inode allocation bitmap block is exactly like the format of the block allocation bitmap block. The explanation above is valid here, with the work `block` replaced by `inode`. Typically, there are much less inodes then blocks in a blocks group and thus only part of the inode bitmap block is used. The number of inodes in a blocks group is another variable which is listed in the `superblock`.

10 On the inode and the inode tables

An inode is a main resource in the ext2 filesystem. It is used for various purposes, but the main two are:

- Support of files
- Support of directories

Each file, for example, will allocate one inode from the filesystem resources.

An ext2 filesystem has a total number of available inodes which is determined while creating the filesystem. When all the inodes are used, for example, you will not be able to create an additional file even though there will still be free blocks on the filesystem.

Each inode takes up 128 bytes in the filesystem. By default, `mke2fs` reserves an inode for each 4096 bytes of the filesystem space.

The inodes are placed in several tables, each of which contains the same number of inodes and is placed at a different blocks group. The goal is to place inodes and their related files in the same blocks group because of locality arguments.

The number of inodes in a blocks group is available in the superblock variable `s_inodes_per_group`. For example, if there are 2000 inodes per group, group 0 will contain the inodes 1-2000, group 2 will contain the inodes 2001-4000, and so on.

Each inode table is accessed from the group descriptor of the specific blocks group which contains the table.

Follows the structure of an inode in Ext2fs:

```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        union {
```

```
                struct {
                        __u32  l_i_reserved1;
                } linux1;
                struct {
                        __u32  h_i_translator;
                } hurd1;
                struct {
                        __u32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __u32   i_version;      /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        union {
                struct {
                        __u8    l_i_frag;       /* Fragment number */
                        __u8    l_i_fsize;      /* Fragment size */
                        __u16   i_pad1;
                        __u32   l_i_reserved2[2];
                } linux2;
                struct {
                        __u8    h_i_frag;       /* Fragment number */
                        __u8    h_i_fsize;      /* Fragment size */
                        __u16   h_i_mode_high;
                        __u16   h_i_uid_high;
                        __u16   h_i_gid_high;
                        __u32   h_i_author;
                } hurd2;
                struct {
                        __u8    m_i_frag;       /* Fragment number */
                        __u8    m_i_fsize;      /* Fragment size */
                        __u16   m_pad1;
                        __u32   m_i_reserved2[2];
                } masix2;
        } osd2;                         /* OS dependent 2 */
};
```

10.1 The allocated blocks

The basic functionality of an inode is to group together a series of allocated blocks. There is no limitation on the allocated blocks - Each block can be allocated to each inode. Nevertheless, block allocation will usually be done in series to take advantage of the locality principle.

The inode is not always used in that way. I will now explain the allocation of blocks, assuming that the current inode type indeed refers to a list of allocated blocks.

It was found experimently that many of the files in the filesystem are actually quite small. To take advantage of this effect, the kernel provides storage of up to 12 block numbers in the inode itself. Those blocks are called `direct blocks`. The advantage is that once the kernel has the inode, it can directly access the file's blocks, without an additional disk access. Those 12 blocks are directly specified in the variables `i_block[0]` to `i_block[11]`.

`i_block[12]` is the `indirect block` - The block pointed by i_block12 will `not` be a data block. Rather, it will just contain a list of direct blocks. For example, if the block size is 1024 bytes, since each block number is 4 bytes long, there will be place for 256 indirect blocks. That is, block 13 till block 268 in the file will be accessed by the `indirect block` method. The penalty in this case, compared to the direct blocks case, is that an additional access to the device is needed - We need `two` accesses to reach the required data block.

In much the same way, `i_block[13]` is the `double indirect block` and `i_block[14]` is the `triple indirect block`.

`i_block[13]` points to a block which contains pointers to indirect blocks. Each one of them is handled in the way described above.

In much the same way, the triple indirect block is just an additional level of indirection - It will point to a list of double indirect blocks.

10.2 The i_mode variable

The i_mode variable is used to determine the `inode type` and the associated `permissions`. It is best described by representing it as an octal number. Since it is a 16 bit variable, there will be 6 octal digits. Those are divided into two parts - The rightmost 4 digits and the leftmost 2 digits.

10.2.1 The rightmost 4 octal digits

The rightmost 4 digits are `bit options` - Each bit has its own purpose.

The last 3 digits (Octal digits 0,1 and 2) are just the usual permissions, in the known form `rwxrwxrwx`. Digit 2 refers to the user, digit 1 to the group and digit 2 to everyone else. They are used by the kernel to grant or deny access to the object presented by this inode.[2]

Bit number 9 signals that the file (I'll refer to the object presented by the inode as file even though it can be a special device, for example) is `set VTX`. I still don't know what is the meaning of "VTX".

Bit number 10 signals that the file is `set group id` - I don't know exactly the meaning of the above either.

Bit number 11 signals that the file is `set user id`, which means that the file will run with an effective user id root.

10.2.2 The leftmost two octal digits

Note the the leftmost octal digit can only be 0 or 1, since the total number of bits is 16.

[2]A `smarter` permissions control is one of the enhancements planned for Linux 1.3 - The ACL (Access Control Lists). Actually, from browsing of the kernel source, some of the ACL handling is already done.

Those digits, as opposed to the rightmost 4 digits, are not bit mapped options. They determine the type of the "file" to which the inode belongs:

- `01` - The file is a `FIFO`.
- `02` - The file is a `character device`.
- `04` - The file is a `directory`.
- `06` - The file is a `block device`.
- `10` - The file is a `regular file`.
- `12` - The file is a `symbolic link`.
- `14` - The file is a `socket`.

10.3 Time and date

Linux records the last time in which various operations occured with the file. The time and date are saved in the standard C library format - The number of seconds which passed since 00:00:00 GMT, January 1, 1970. The following times are recorded:

- `i_ctime` - The time in which the inode was last allocated. In other words, the time in which the file was created.
- `i_mtime` - The time in which the file was last modified.
- `i_atime` - The time in which the file was last accessed.
- `i_dtime` - The time in which the inode was deallocated. In other words, the time in which the file was deleted.

10.4 i_size

`i_size` contains information about the size of the object presented by the inode. If the inode corresponds to a regular file, this is just the size of the file in bytes. In other cases, the interpretation of the variable is different.

10.5 User and group id

The user and group id of the file are just saved in the variables `i_uid` and `i_gid`.

10.6 Hard links

Later, when we'll discuss the implementation of directories, it will be explained that each `directory entry` points to an inode. It is quite possible that a `single inode` will be pointed to from `several` directories. In that case, we say that there exist `hard links` to the file - The file can be accessed from each of the directories.

The kernel keeps track of the number of hard links in the variable `i_links_count`. The variable is set to "1" when first allocating the inode, and is incremented with each additional link. Deletion of a file will delete the current directory entry and will decrement the number of links. Only when this number reaches zero, the inode will be actually deallocated.

The name `hard link` is used to distinguish between the alias method described above, to another alias method called `symbolic linking`, which will be described later.

10.7 The Ext2fs extended flags

The ext2 filesystem associates additional flags with an inode. The extended attributes are stored in the variable `i_flags`. `i_flags` is a 32 bit variable. Only the 7 rightmost bits are defined. Of them, only 5 bits are used in version 0.5a of the filesystem. Specifically, the `undelete` and the `compress` features are not implemented, and are to be introduced in Linux 1.3 development.

The currently available flags are:

- bit 0 - Secure deletion.

  When this bit is on, the file's blocks are zeroed when the file is deleted. With this bit off, they will just be left with their original data when the inode is deallocated.

- bit 1 - Undelete.

  This bit is not supported yet. It will be used to provide an `undelete` feature in future Ext2fs developments.

- bit 2 - Compress file.

  This bit is also not supported. The plan is to offer "compression on the fly" in future releases.

- bit 3 - Synchronous updates.

  With this bit on, the meta-data will be written synchronously to the disk, as if the filesystem was mounted with the "sync" mount option.

- bit 4 - Immutable file.

  When this bit is on, the file will stay as it is - Can not be changed, deleted, renamed, no hard links, etc, before the bit is cleared.

- bit 5 - Append only file.

  With this option active, data will only be appended to the file.

- bit 6 - Do not dump this file.

  I think that this bit is used by the port of dump to linux (ported by `Remy Card`) to check if the file should not be dumped.

10.8 Symbolic links

The `hard links` presented above are just another pointers to the same inode. The important aspect is that the inode number is `fixed` when the link is created. This means that the implementation details of the filesystem are visible to the user - In a pure abstract usage of the filesystem, the user should not care about inodes.

The above causes several limitations:

- Hard links can be done only in the same filesystem. This is obvious, since a hard link is just an inode number in some directory entry, and the above elements are filesystem specific.
- You can not "replace" the file which is pointed to by the hard link after the link creation. "Replacing" the file in one directory will still leave the original file in the other directory - The "replacement" will not deallocate the original inode, but rather allocate another inode for the new version, and the directory entry at the other place will just point to the old inode number.

`Symbolic link`, on the other hand, is analyzed at `run time`. A symbolic link is just a `pathname` which is accessible from an inode. As such, it "speaks" in the language of the abstract filesystem. When the kernel reaches a symbolic link, it will `follow it in run time` using its normal way of reaching directories.

As such, symbolic link can be made `across different filesystems` and a replacement of a file with a new version will automatically be active on all its symbolic links.

The disadvantage is that hard link doesn't consume space except to a small directory entry. Symbolic link, on the other hand, consumes at least an inode, and can also consume one block.

When the inode is identified as a symbolic link, the kernel needs to find the path to which it points.

10.8.1 Fast symbolic links

When the pathname contains up to 64 bytes, it can be saved directly in the inode, on the `i_block[0] - i_block[15]` variables, since those are not needed in that case. This is called `fast` symbolic link. It is fast because the pathname resolution can be done using the inode itself, without accessing additional blocks. It is also economical, since it allocates only an inode. The length of the pathname is stored in the `i_size` variable.

10.8.2 Slow symbolic links

Starting from 65 bytes, additional block is allocated (by the use of `i_block[0]`) and the pathname is stored in it. It is called slow because the kernel needs to read additional block to resolve the pathname. The length is again saved in `i_size`.

10.9 i_version

`i_version` is used with regard to Network File System. I don't know its exact use.

10.10 Reserved variables

As far as I know, the variables which are connected to ACL and fragments are not currently used. They will be supported in future versions.

Ext2fs is being ported to other operating systems. As far as I know, at least in linux, the os dependent variables are also not used.

10.11 Special reserved inodes

The first ten inodes on the filesystem are special inodes:

- Inode 1 is the `bad blocks inode` - I believe that its data blocks contain a list of the bad blocks in the filesystem, which should not be allocated.
- Inode 2 is the `root inode` - The inode of the root directory. It is the starting point for reaching a known path in the filesystem.
- Inode 3 is the `acl index inode`. Access control lists are currently not supported by the ext2 filesystem, so I believe this inode is not used.
- Inode 4 is the `acl data inode`. Of course, the above applies here too.
- Inode 5 is the `boot loader inode`. I don't know its usage.
- Inode 6 is the `undelete directory inode`. It is also a foundation for future enhancements, and is currently not used.
- Inodes 7-10 are `reserved` and currently not used.

11 Directories

A directory is implemented in the same way as files are implemented (with the direct blocks, indirect blocks, etc) - It is just a file which is formatted with a special format - A list of directory entries.

Follows the definition of a directory entry:

```
struct ext2_dir_entry {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u16   name_len;               /* Name length */
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

Ext2fs supports file names of varying lengths, up to 255 bytes. The `name` field above just contains the file name. Note that it is `not zero terminated`; Instead, the variable `name_len` contains the length of the file name.

The variable rec_len is provided because the directory entries are padded with zeroes so that the next entry will be in an offset which is a multiplition of 4. The resulting directory entry size is stored in rec_len. If the directory entry is the last in the block, it is padded with zeroes till the end of the block, and rec_len is updated accordingly.

The inode variable points to the inode of the above file.

Deletion of directory entries is done by appending of the deleted entry space to the previous (or next, I am not sure) entry.

12 The superblock

The superblock is a block which contains information which describes the state of the internal filesystem.

The superblock is located at the fixed offset 1024 in the device. Its length is 1024 bytes also.

The superblock, like the group descriptors, is copied on each blocks group boundary for backup purposes. However, only the main copy is used by the kernel.

The superblock contain three types of information:

- Filesystem parameters which are fixed and which were determined when this specific filesystem was created. Some of those parameters can be different in different installations of the ext2 filesystem, but can not be changed once the filesystem was created.
- Filesystem parameters which are tunable - Can always be changed.
- Information about the current filesystem state.

Follows the superblock definition:

```
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 */
        __u32   s_log_block_size;       /* Block size */
        __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;                /* Mount time */
        __u32   s_wtime;                /* Write time */
        __u16   s_mnt_count;            /* Mount count */
        __s16   s_max_mnt_count;        /* Maximal mount count */
        __u16   s_magic;                /* Magic signature */
        __u16   s_state;                /* File system state */
```

```
        __u16   s_errors;               /* Behaviour when detecting errors */
        __u16   s_pad;
        __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 */
        __u32   s_reserved[235];        /* Padding to the end of the block */
};
```

12.1 superblock identification

The ext2 filesystem's superblock is identified by the `s_magic` field. The current ext2 magic number is 0xEF53. I presume that "EF" means "Extended Filesystem". In versions of the ext2 filesystem prior to 0.2B, the magic number was 0xEF51. Those filesystems are not compatible with the current versions; Specifically, the group descriptors definition is different. I doubt if there still exists such a installation.

12.2 Filesystem fixed parameters

By using the word `fixed`, I mean fixed with respect to a particular installation. Those variables are usually not fixed with respect to different installations.

The `block size` is determined by using the `s_log_block_size` variable. The block size is 1024*pow (2,s_log_block_size) and should be between 1024 and 4096. The available options are 1024, 2048 and 4096.

`s_inodes_count` contains the total number of available inodes.

`s_blocks_count` contains the total number of available blocks.

`s_first_data_block` specifies in which of the `device block` the `superblock` is present. The superblock is always present at the fixed offset 1024, but the device block numbering can differ. For example, if the block size is 1024, the superblock will be at `block 1` with respect to the device. However, if the block size is 4096, offset 1024 is included in `block 0` of the device, and in that case `s_first_data_block` will contain 0. At least this is how I understood this variable.

`s_blocks_per_group` contains the number of blocks which are grouped together as a blocks group.

`s_inodes_per_group` contains the number of inodes available in a group block. I think that this is always the total number of inodes divided by the number of blocks groups.

`s_creator_os` contains a code number which specifies the operating system which created this specific filesystem:

- `Linux` :-) is specified by the value `0`.
- `Hurd` is specified by the value `1`.
- `Masix` is specified by the value `2`.

`s_rev_level` contains the major version of the ext2 filesystem. Currently this is always `0`, as the most recent version is 0.5B. It will probably take some time until we reach version 1.0.

As far as I know, fragments (sub-block allocations) are currently not supported and hence a block is equal to a fragment. As a result, `s_log_frag_size` and `s_frags_per_group` are always equal to `s_log_block_size` and `s_blocks_per_group`, respectively.

12.3 Ext2fs error handling

The ext2 filesystem error handling is based on the following philosophy:

1. Identification of problems is done by the kernel code.
2. The correction task is left to an external utility, such as `e2fsck by Theodore Ts´o` for `automatic` analysis and correction, or perhaps `debugfs by Theodore Ts´o` and `EXT2ED by myself`, for `hand` analysis and correction.

The `s_state` variable is used by the kernel to pass the identification result to third party utilities:

- `bit 0` of s_state is reset when the partition is mounted and set when the partition is unmounted. Thus, a value of 0 on an unmounted filesystem means that the filesystem was not unmounted properly - The filesystem is not "clean" and probably contains errors.
- `bit 1` of s_state is set by the kernel when it detects an error in the filesystem. A value of 0 doesn't mean that there isn't an error in the filesystem, just that the kernel didn't find any.

The kernel behavior when an error is found is determined by the user tunable parameter `s_errors`:

- The kernel will ignore the error and continue if `s_errors=1`.
- The kernel will remount the filesystem in read-only mode if `s_errors=2`.
- A kernel panic will be issued if `s_errors=3`.

The default behavior is to ignore the error.

12.4 Additional parameters used by e2fsck

Of-course, `e2fsck` will check the filesystem if errors were detected or if the filesystem is not clean.

In addition, each time the filesystem is mounted, `s_mnt_count` is incremented. When s_mnt_count reaches `s_max_mnt_count`, `e2fsck` will force a check on the filesystem even though it may be clean. It will then zero s_mnt_count. `s_max_mnt_count` is a tunable parameter.

E2fsck also records the last time in which the file system was checked in the `s_lastcheck` variable. The user tunable parameter `s_checkinterval` will contain the number of seconds which are allowed to pass since `s_lastcheck` until a check is reforced. A value of `0` disables time-based check.

12.5 Additional user tunable parameters

`s_r_blocks_count` contains the number of disk blocks which are reserved for root, the user whose id number is `s_def_resuid` and the group whose id number is `s_deg_resgid`. The kernel will refuse to allocate those last `s_r_blocks_count` if the user is not one of the above. This is done so that the filesystem will usually not be 100% full, since 100% full filesystems can affect various aspects of operation.

`s_def_resuid` and `s_def_resgid` contain the id of the user and of the group who can use the reserved blocks in addition to root.

12.6 Filesystem current state

`s_free_blocks_count` contains the current number of free blocks in the filesystem.

`s_free_inodes_count` contains the current number of free inodes in the filesystem.

`s_mtime` contains the time at which the system was last mounted.

`s_wtime` contains the last time at which something was changed in the filesystem.

13 Copyright

14 Acknowledgments

I would like to thank the following people, who were involved in the design and implementation of the ext2 filesystem kernel code and support utilities:

- `Remy Card`

  Who designed, implemented and maintains the ext2 filesystem kernel code, and some of the ext2 utilities. `Remy Card` is also the author of several helpful slides concerning the ext2 filesystem. Specifically, he is the author of `File Management in the Linux Kernel` and of `The Second Extended File System - Current State, Future Development`.

- `Wayne Davison`

  Who designed the ext2 filesystem.

- `Stephen Tweedie`

  Who helped designing the ext2 filesystem kernel code and wrote the slides `Optimizations in File Systems`.

- `Theodore Ts´o`

  Who is the author of several ext2 utilities and of the ext2 library `libext2fs` (which I didn't use, simply because I didn't know it exists when I started to work on my project).

Lastly, I would like to thank, of-course, `Linus Torvalds` and the `Linux community` for providing all of us with such a great operating system.

Please contact me in a case of an error report, suggestions, or just about anything concerning this document.

Enjoy,

Gadi Oxman <tgud@tochnapc2.technion.ac.il>

Haifa, August 95