SUPERBLOQUE - EXT2 LINUX 2.6

Diseño de Sistemas Operativos

Universidad de Las Palmas de Gran Canaria

¿Qué estudiaremos en el tema?

- Historia y motivación de los distintos sistemas de archivos en Linux
- Conceptos básicos de sistemas de archivos tipo UNIX
- Aspectos clave del sistema de archivos Ext2
- El superbloque Ext2
- Estructura de datos -- </include/linux/ext2_fs.h>
- Funciones -- </fs/ext2/super.c>
- Referencias
- Para profundizar

¿Qué objetivos perseguimos?

- Conocer los sistemas de archivos más importantes en Linux
- Conocer las características principales de Ext2
- Conocer cómo Linux gestiona el superbloque Ext2
- Estructuras de datos
  - </include/linux/ext2_fs.h>
- Funciones
  - </fs/ext2/super.c>

Evolución de Linux y del sistema de archivos (1)

- Linux se basa en Minix ► Primer FS de Linux es Minix FS
- Minix es muy limitado
- Tamaño máximo del FS 64 Mb
- Nombres de archivo de 14 caracteres máximo
- Se incorpora VFS ► Facilita la integración de nuevos FS en el núcleo
- Aparece el sistema de archivos Ext
- Sistema de archivos de hasta 2 Gb
- Nombres de archivo de hasta 255 caracteres
- No soporta timestamps separados de acceso, modificación de inodo, modificación de datos
- Se emplean listas enlazadas para mantener información sobre bloques e inodos

Evolución de Linux y del sistema de archivos (2)

- Aparecen simultáneamente XIA y Ext2
- XIA ► Basado en código Minix FS. Pocas mejoras.
- Ext2 ► Basado en Ext. Aporta muchas mejoras. Posibilidad de ser extendido en el futuro

- Sistemas de archivos modernos ► Journaling
  - Ext3
  - RaiserFS
  - JFS
  - XFS

Sistemas de archivos en UNIX Conceptos (1)

- Un ***BLOQUE lógico*** es la unidad más pequeña de almacenamiento (se mide en bytes)
- Un ***VOLUMEN lógico*** es una ***PARTICIÓN*** de un dispositivo como un disco.
- Las particiones están divididas en bloques.
- La ***Asignación de bloques*** es un método por el cual el sistema de ficheros asigna los bloques libres.
- Una tabla de bits a cada bloque se mantiene y almacena en el sistema.

Sistemas de archivos en UNIX Conceptos: i-node (2)

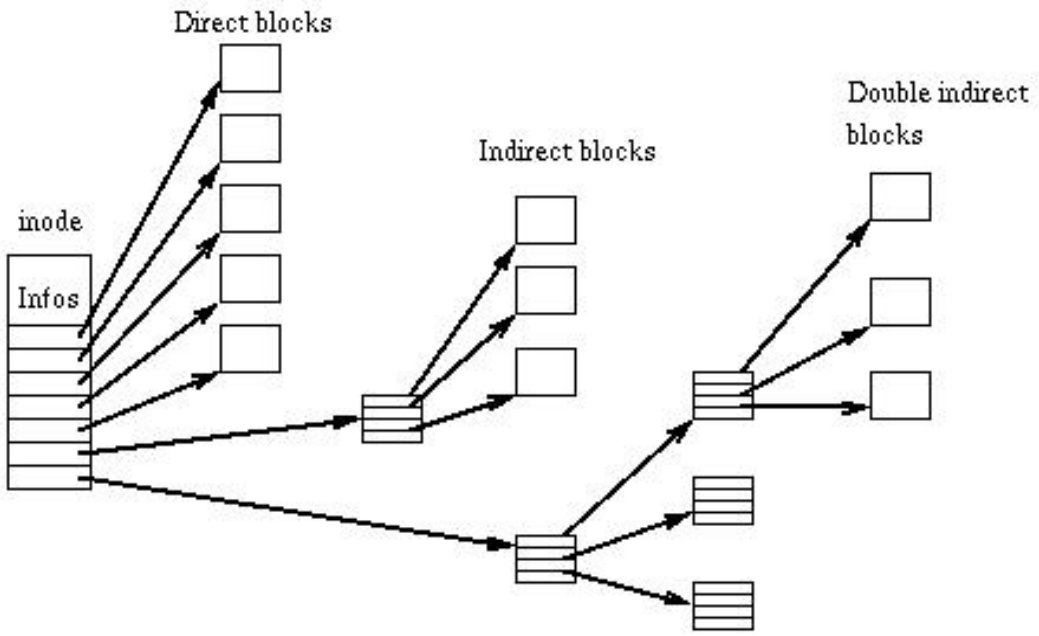
- Un ***I-NODE*** almacena toda la información sobre un fichero excepto la información del fichero propiamente dicha.

- Un **i-node** consume también bloques.

- Un **i-node** contiene:
  - Los permisos del fichero.
  - El tipo del fichero.
  - El número de links (enlaces) al fichero.

Sistemas de archivos en UNIX Conceptos: i-node (3)

- **I-nodes**
  - Punteros a bloques de datos
  - Tipo
  - Permisos
  - Propietarios
  - Marcas de tiemp
  - Tamaño



Sistemas de archivos en UNIX Conceptos (4)

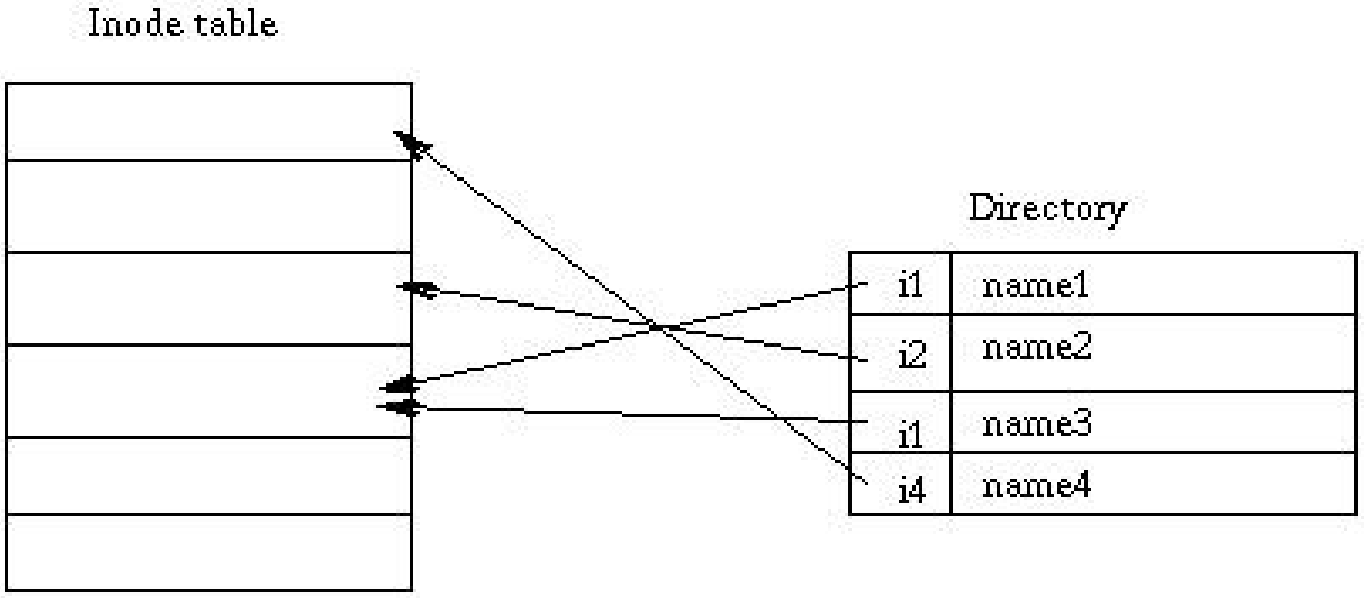
- Cada **i-node** tiene un número único que los identifica (offset).

- Un ***DIRECTORIO*** es un tipo específico de fichero que contiene punteros a otros ficheros.

- El **i-node** de un **fichero directorio** simplemente contiene el número identificador de los i-nodes.

Sistemas de archivos en UNIX Conceptos (5)

- **Directorios**
  - Número de inodo
  - Nombre de archivo



Second Extended File System
EXT2

Second Extended File System EXT2 (1)

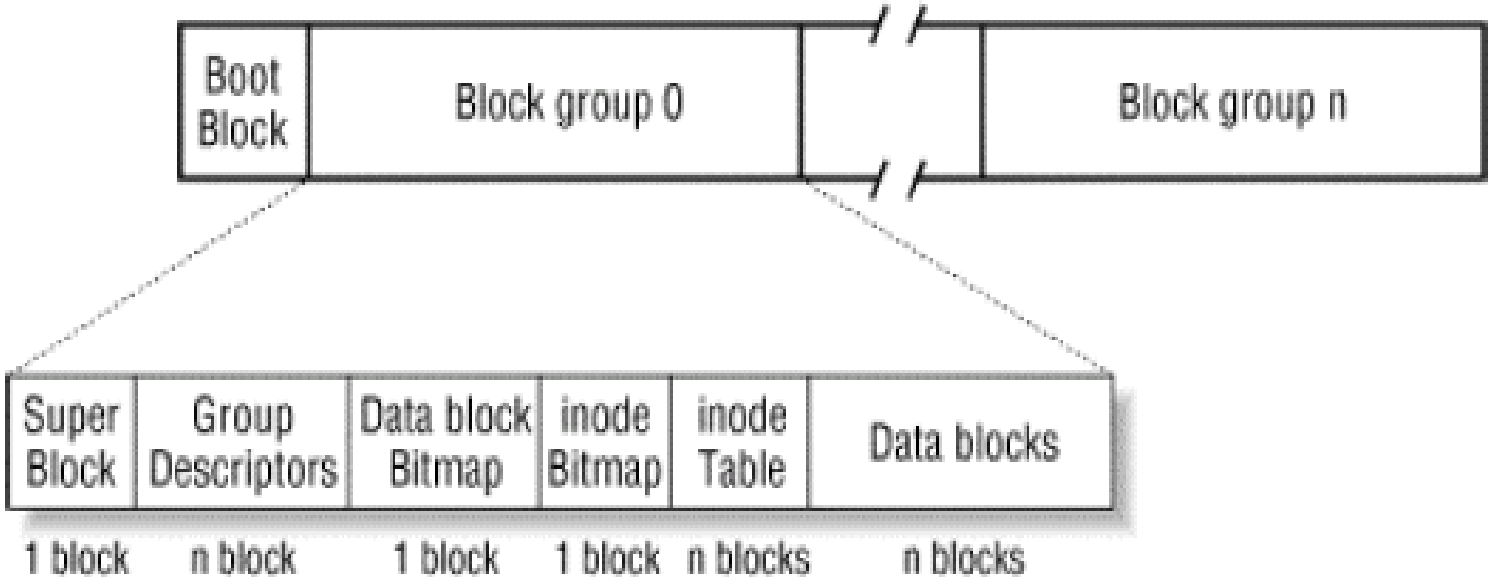
- Sucesor del sistema de archivos Ext
- Robusto
- Alto rendimiento
- Ampliable sin necesidad de formatear
- Implementa los conceptos UNIX
  - I-nodes
  - Directorios (entradas de longitud variable)
  - Enlaces duros y simbólicos (normales y rápidos)

Second Extended File System EXT2 (2)

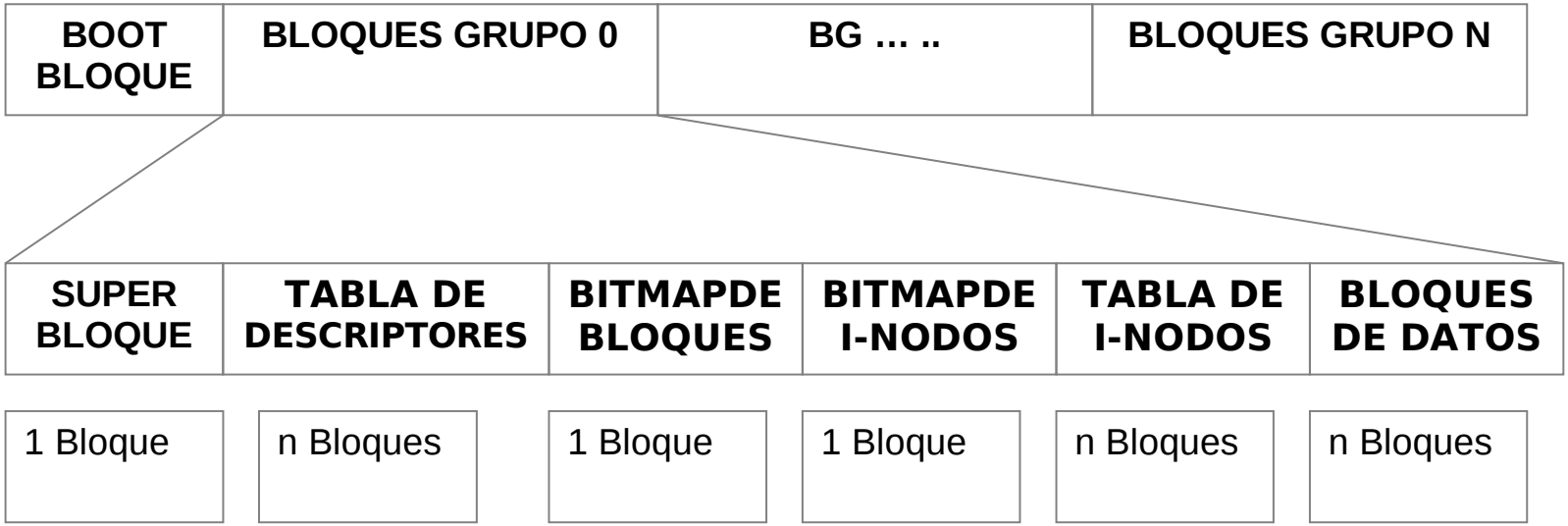
- Particiones de hasta 4Tb
- Nombres de archivo de hasta 255 o extendido hasta 1012 caracteres
- Bloques reservados para el superusuario
- Permite seleccionar el tamaño del bloque lógico
- Almacenamiento redundante de información crítica para el FS
  - Superbloque
  - Descriptores de grupo

Second Extended File System EXT2 (3)

- **Estructura física de un sistema Ext2**

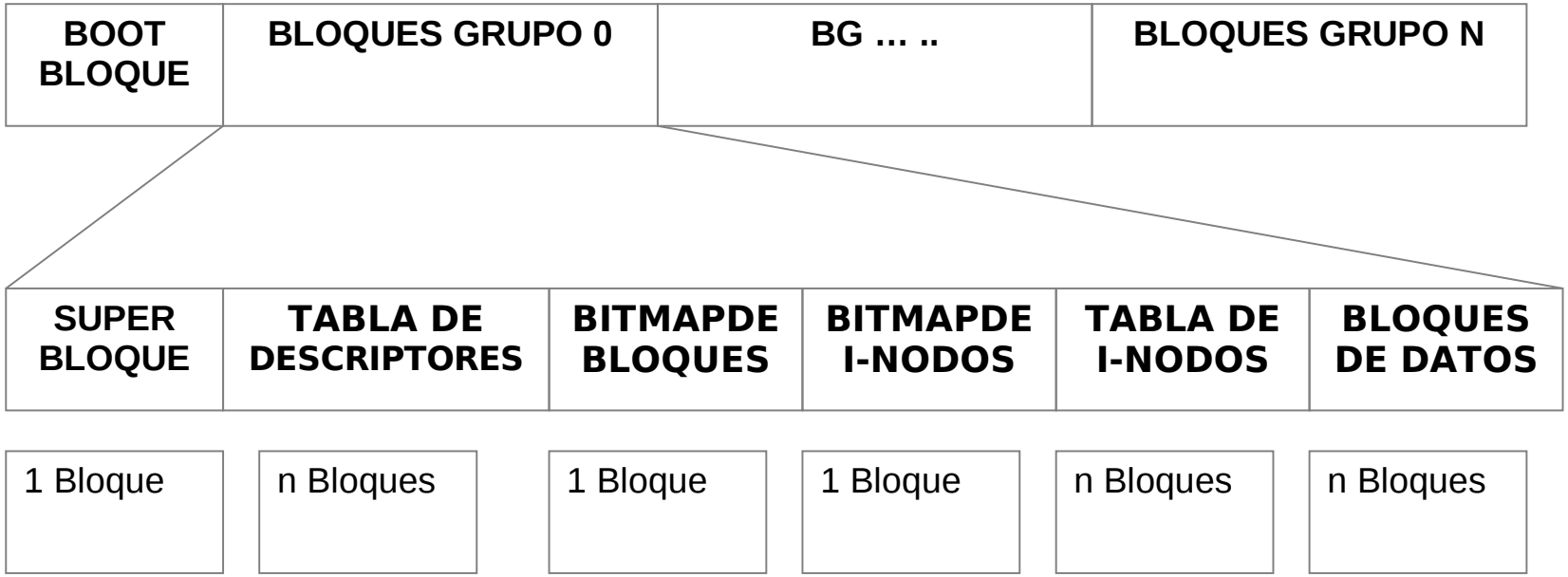


EXT2: Estructura física (4)



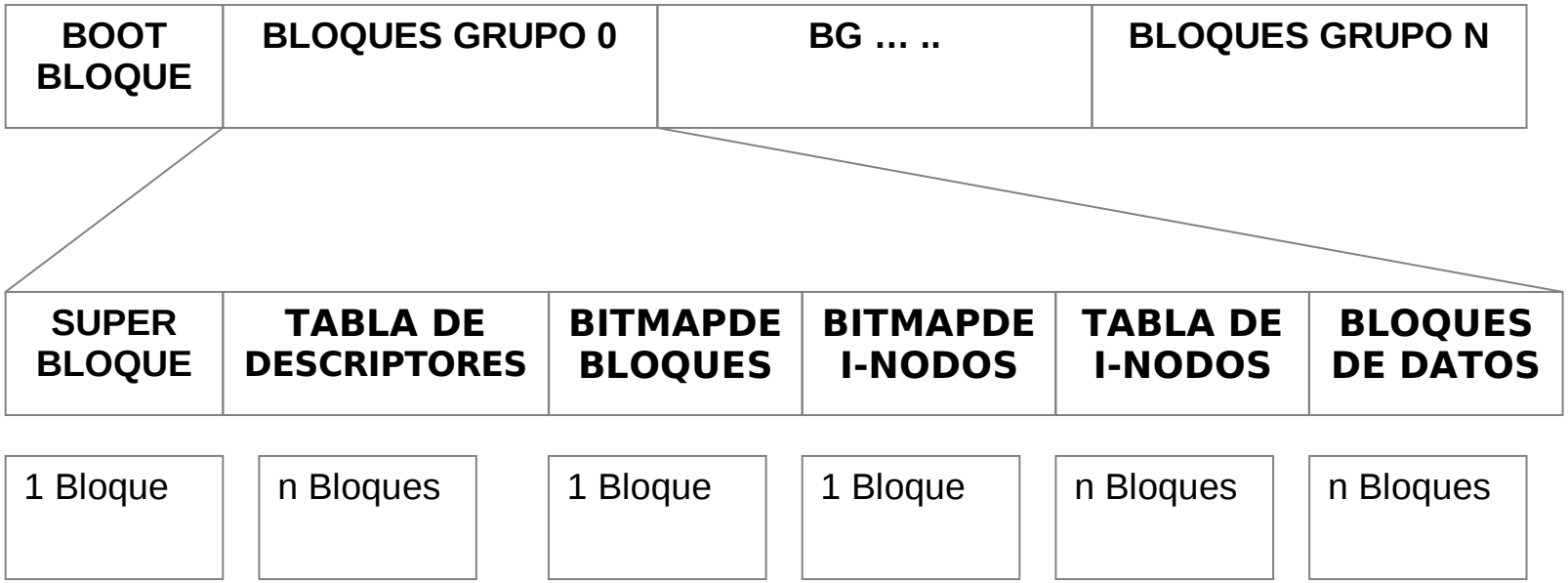
- Está formado por una colección de bloques secuenciales
- El sector de *boot* contiene el código necesario para cargar el núcleo en el arranque del sistema.
- El sistema de ficheros es manejado globalmente como una serie de *Grupos de Bloques* (GB)

EXT2: Estructura física (5)



- Los GB ayudan a mantener la información relacionada físicamente cercana en el disco, simplificando las tareas de gestión.
- Todos los grupos de bloques tienen el mismo tamaño y están colocados de manera secuencial.
- De esta manera se reduce la fragmentación externa.

EXT2: Estructura física (6)



Cada GB contiene a su vez distintos elementos:

- Una copia del superbloque
- Una copia de la tabla de descriptores de grupo
- Un bloque de bitmap para los bloques
- Un bloque de bitmap para los i-nodos
- Una tabla de i-nodos
- Bloques de datos

Estructura ext2 SUPERBLOQUE(1)

- **Estructura ext2_super_block en el fichero </include/linux/ext2_fs.h>**

```
334 /*
335  * Structure of the super block
336  */
337 struct ext2_super_block {
338         __le32  s_inodes_count;         /* Inodes count */
339         __le32  s_blocks_count;         /* Blocks count */
340         __le32  s_r_blocks_count;       /* Reserved blocks count */
341         __le32  s_free_blocks_count;    /* Free blocks count */
342         __le32  s_free_inodes_count;    /* Free inodes count */
343         __le32  s_first_data_block;     /* First Data Block */
344         __le32  s_log_block_size;       /* Block size */
345         __le32  s_log_frag_size;        /* Fragment size */
...
...
```

Estructura ext2 SUPERBLOQUE(2)

- El superbloque contiene 3 tipos de parámetros
  - 1- Parámetros no modificables ►Se fijan en la creación del FS
    - Tamaño de bloque
    - Número de bloques disponibles
    - Número de inodos disponibles
  - 2- Parámetros modificables (se modifica durante el uso del FS)
    - Bloques reservados al superusuario
    - UID (identificador usuario) del superusuario por defecto
    - GID (identificador de grupo) del superusuario por defecto

Estructura ext2 SUPERBLOQUE (3)

- El superbloque contiene 3 tipos de parámetros
  - 3- Parámetros de estado y uso (varian durante el uso fs)
    - Inodos y bloques libres
    - Estado del sistemas de archivos (limpio/no limpio)
    - Número de montajes
    - Máximo número de montajes entre comprobaciones
    - Máximo tiempo entre comprobaciones
    - Comportamiento frente a errores
    - Directorio del último montaje

Ext2 SUPERBLOQUE (4)

s_inodes_count	Número de inodos
s_blocks_count	Número de bloques
s_r_blocks_count	Número de bloques reservados
s_free_blocks_count	Número de bloques libres
s_free_inodes_count	Número de inodos libres
s_first_data_block	Primer bloque de datos
s_log_block_size	Tamaño de bloque
s_log_frag_size	Tamaño de fragmento
s_blocks_per_group	Número de bloques por grupo
s_frags_per_group	Número de fragmentos por grupo
s_inodes_per_group	Número de inodos por grupo
s_mtime	Fecha del último montaje

Ext SUPERBLOQUE (5)

- s_wtime Fecha de la última escritura
- s_mnt_count Número de montajes realizados desde la última comprobación
- s_max_mnt_count Máximo número de montajes entre comprobaciones
- s_magic Número mágico
- s_state Estado del sistema de archivos
- s_errors Comportamiento en caso de error
- s_minor_rev_level Número menor de revisión
- s_last_check Fecha de la úlitma coprobación
- s_check_interval Máximo intervalo de tiempo entre comprobaciones
- s_creator_os Sistema operativo que creó el FS
- s_rev_level Número de revisión
- s_def_resuid UID del superusuario por defecto
- s_def_resuid GID del superusuario por defecto

Ext2 SUPERBLOQUE (6)

Los siguientes campos son para superbloques EXT2_DINAMIC_REV sólamente

- s_first_ino Primer inodo no reservado
- s_inode_size Tamaño de la estructura inodo
- s_block_group_nr Número de grupo que contiene éste superbloque
- s_feature_compat Indicador de características compatibles
- s_feature_incompat Indicador de características incompatibles
- s_feature_ro_compat Compatibilidad con sólo lectura
- s_uuid[16] UID de 128 bits, para el volumen
- s_volume_name[16] Nombre del volumen
- s_last_mounted`[64] directorio sobre el que se montó la última vez
- s_algorithm_usage_bitmap Para compresión

Ext2 SUPERBLOQUE (7)

Campos relacionado con el rendimiento. Preasignación de directorios únicamente debe realizarse si el flag EXT2_COMPAT_PREALLOC está activo.

- s_prealloc_blocks — Número de bloques que debe intentar preasignarse
- s_prealloc_dir_blocks — Número de bloques a preasignar para directorios
- s_padding1 — Número de bloques de relleno

Ext2 SUPERBLOQUE (8)

Campos relacionados con journaling, válidos si EXT3_FEATURE_COMPAT_HAS_JOURNAL está activo.

- s_journal_uuid[16] UID del superbloque journal
- s_journal_inum Número de inodo del archivo journal
- s_journal_dev Número del dispositivo journal
- s_last_orphan Inicio de la lista de inodos a borrar
- s_hash_seed[4] Semilla hash de HTREE
- s_def_hash_version Versión por defecto de hash a usar
- s_reserved_char_pad Caracter de relleno por defecto
- s_reserved_word_pad Palabra de relleno por defecto
- s_default_mount_opts Opciones de montaje por defecto
- s_first_meta_bg Primer grupo de bloques
- s_reserved[190] Reserva de bloques

ext2_sb_info SUPERBLOQUE (1)

- **Estructura ext2_sb_info en el fichero** </include/linux/ext2_fs_sb.h>

```
22 /*
23  * second extended-fs super-block data in memory
24  */
25 struct ext2_sb_info {
26     unsigned long s_frag_size;      /* Size of a fragment in bytes */
27     unsigned long s_frags_per_block;/* Number of fragments per block */
28     unsigned long s_inodes_per_block;/* Number of inodes per block */
29     unsigned long s_frags_per_group;/* Number of fragments in a group */
30     unsigned long s_blocks_per_group;/* Number of blocks in a group */
31     unsigned long s_inodes_per_group;/* Number of inodes in a group */
...
```

ext2_sb_info SUPERBLOQUE (2)

- s_frag_size — Tamaño de un fragmento en bytes
- s_frags_per_block — Número de fragmentos por bloque
- s_inodes_per_block — Número de inodos por bloque
- s_frags_per_group — Número de fragmentos en un grupo
- s_blocks_per_group — Número de bloques en un grupo
- s_inodes_per_group — Número de inodos en un grupo
- s_itb_per_group — Nº bloques de la tabla de inodos por grupo
- s_gdb_count — Nº bloques del descriptor de grupo
- s_desc_per_block — Nº de descriptores de grupo por bloque
- s_groups_count — Número de grupos en el fs
- s_overhead_last — Último overhead calculado
- s_blocks_last — Último bloque contado

ext2_sb_info SUPERBLOQUE (3)

- s_sbh Buffer que contiene el superbloque
- s_es Puntero al buffer superbloque
- s_group_desc Puntero al descriptor de grupo
- s_mount_opt Opciones de montaje
- s_sb_block Numero de bloque del superbloque
- s_resuid Superusuario por defecto
- s_resgid Grupo del superusuario por defecto
- s_mount_state Estado del montaje
- s_pad
- s_addr_per_block_bits
- s_desc_per_block_bits
- s_inode_size Tamaño de inodo

ext2_sb_info SUPERBLOQUE (4)

- s_first_ino — Bloque del primer inodo
- s_next_gen_lock
- s_next_generation
- s_dir_count — Número de directorios
- s_debts
- s_freeblocks_counter — Número de bloques libres
- s_freeinodes_counter — Número de inodos libres
- s_dirs_counter — Número de directorios
- s_blockgroup_lock
- s_rsv_window_lock
- s_rsv_window_root — Ventana reserva para root
- s_rsv_window_head — Principio de la reserva

ext2_group_desc
Descriptores de grupo

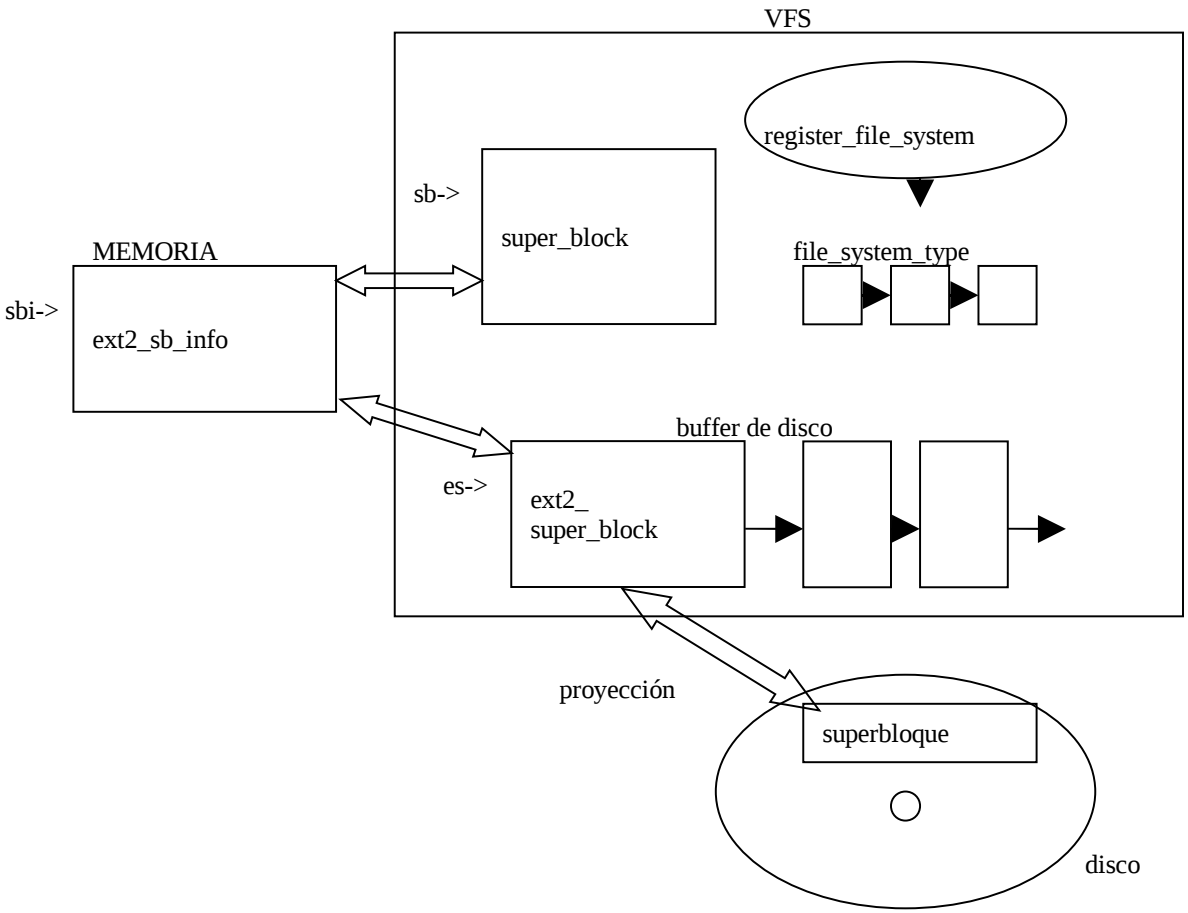
```
struct ext2_group_desc
 {
        __le32  bg_block_bitmap;        /* Dirección del Bloque del bitmap de bloques */
        __le32  bg_inode_bitmap;        /*Dirección del  Bloque del bitmap de inodos  */
        __le32  bg_inode_table;         /* Dirección del Bloque de la tabla de inodos */
        __le16  bg_free_blocks_count;  /* número  de bloques libres */
        __le16  bg_free_inodes_count;  /* número de inodos libres */
        __le16  bg_used_dirs_count;     /* número de directorios */
        __le16  bg_pad;                 /* no utilizado */
        __le32  bg_reserved[3];         /* reservado para futura extensión */
 };
```

Ext2 estructuras de datos

Hemos visto tres estructuras relacionadas con el superbloque:

- En el capitulo de sistema de ficheros virtual VFS, **super_block**, la variable que lo referencia es “**sb**”.
-
- En este capitulo superbloque para el sistema de ficheros ext2, **ext2_super_block**, esta estructura se proyecta en el buffer de disco, la variable que lo referencia es “**es**”.
-
- Estructura **ext2_sb_info**, superbloque ext2 en memoria, se referencia con la variable “**sbi**”.

Ext2 estructuras de datos



VFS funciones vinculadas al superbloque

struct ***super_operations*** *{*

struct inode *(*****alloc_inode****)(struct super_block* *******sb); /*Lee un inode de disco */*

void (****write_inode****) (struct inode *, int);/*escribe el inode del VFS en el disco*/*

void (****delete_inode****) (struct inode *);/*borra un inode */*

void (*****put_super****) (struct super_block *);/*libera la memoria usada por superb*/*

void (*****write_super****) (struct super_block *);/*escribe el superbloque en disco*/*

int (****sync_fs****)(struct super_block *sb, int wait);*

int (*****statfs****) (struct super_block *, struct statfs *); /*lee información de control*/*

int (*****remount_fs****) (struct super_block *, int *, char *);/*nuevo montaje*/*

void (****umount_begin****) (struct super_block *);*

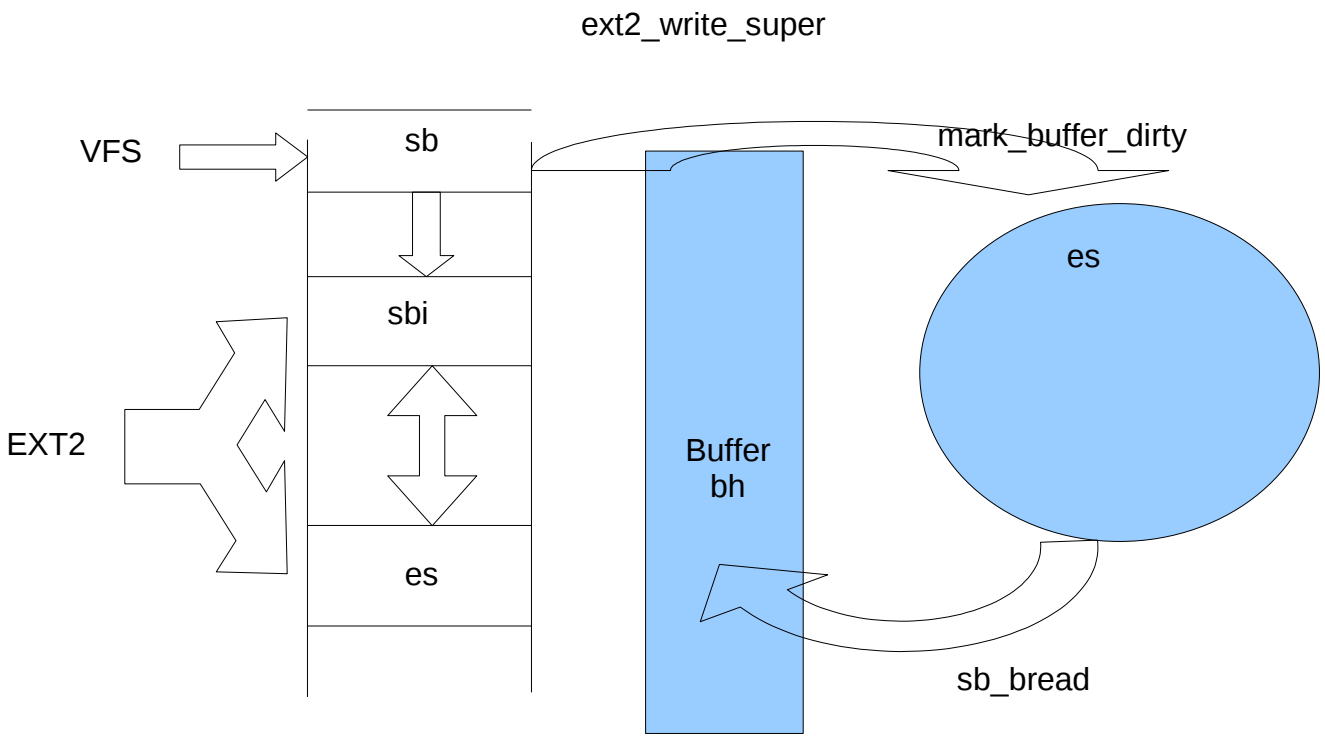
int (****show_options****)(struct seq_file *, struct vfsmount *);*

- El VFS realiza la correspondencia entre las funciones llamadas por procesos y la específica para ese SF. P. ej.: ext2_write_inode -> write_inode
- El siguiente array de punteros a funciones, indica esta correspondencia

Ext2 funciones vinculadas al superbloque/fs/ext2/super.c

```
static struct super_operations ext2_sops = {
        .alloc_inode            = ext2_alloc_inode,
        .destroy_inode          = ext2_destroy_inode,
        .write_inode            = ext2_write_inode,
        .delete_inode           = ext2_delete_inode,
        .put_super              = ext2_put_super,
        .write_super            = ext2_write_super,
        .statfs                 = ext2_statfs,
        .remount_fs             = ext2_remount,
        .clear_inode            = ext2_clear_inode,
    .show_options               = ext2_show_options,
 #ifdef CONFIG_QUOTA
        .quota_read   = ext2_quota_read,
        .quota_write  = ext2_quota_write,
 #endif
 };
```

Ext2 estructuras de datos y funciones

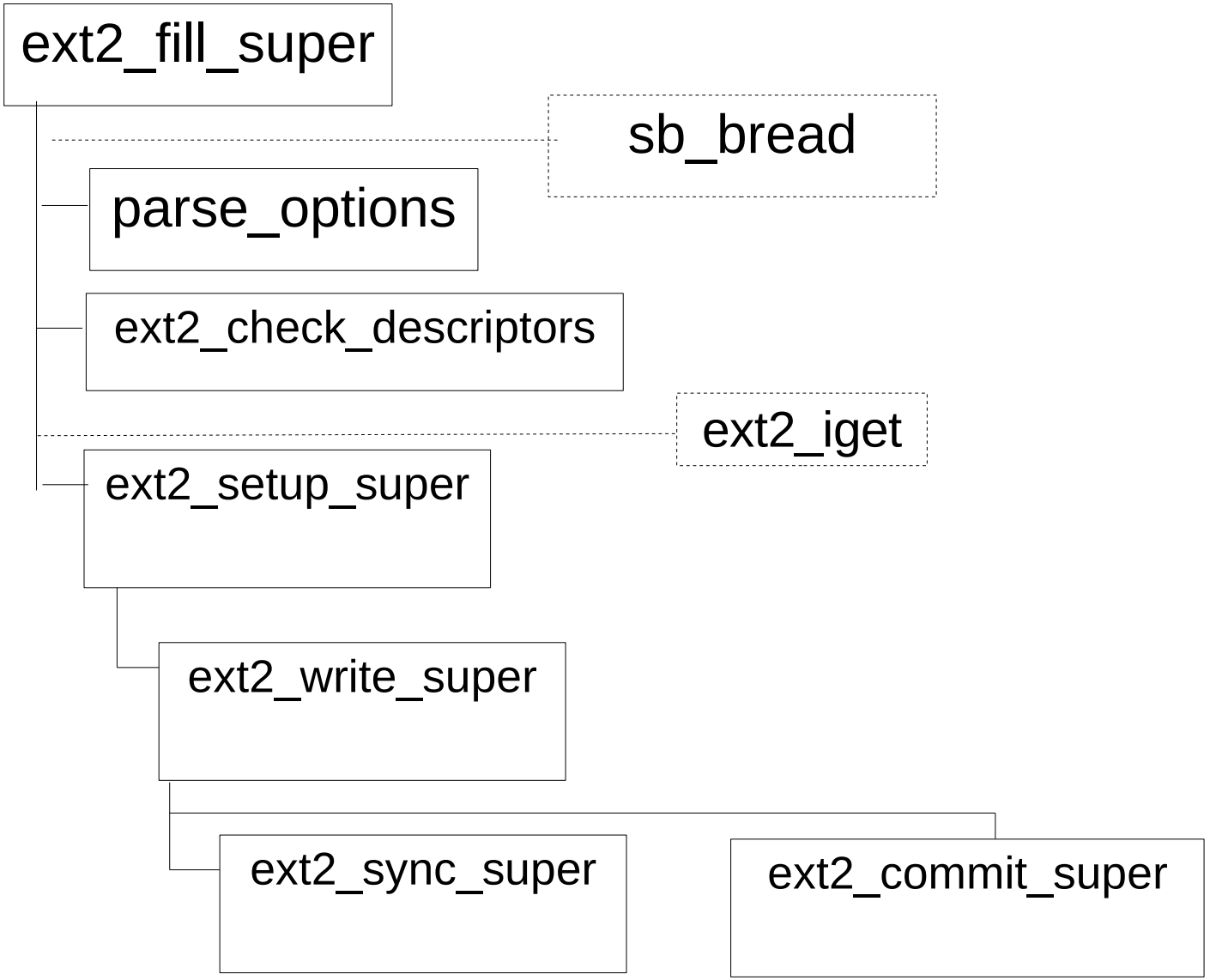


SUPER.C: Funciones

- Principales funciones vinculadas al superbloque en el sf ext2:

  - ext2_error
  - ext2_warning
  - ext2_parse_options
  - ext2_setup_super
  - ext2_check_descriptors
  - ext2_commit_super
  - ext2_remount
  - ext2_write_super
  - ext2_put_super
  - ext2_sync_super
  - ext2_fill_super
  - ext2_statfs
  - ext2_init_ext2_fs

Dependencias entre funciones



Ext2_fill_super

- **¿Qué hace?**

  Lee del disco el superbloque del sistema de ficheros a montar. Se llama cuando se monta un sistema de ficheros.

Se lee el superbloque desde el disco, se guarda en memoria (en buffer bh, en es y en sbi).
Comprueba su validez mediante el número mágico (*s_magic*).
Comprueba las opciones de montaje con la función *parse_options*.
Comprueba el tamaño de bloque, y en caso de ser incorrecto, se libera y se lee de nuevo el superbloque de disco y se almacena en bh, es y sbi.
Se inicializan campos del superbloque sbi, y se hace una comprobación de que se vaya a montar un sistema de ficheros ext2 mediante su número mágico.

Ext2_fill_super

¿Qué hace?

Comprueba que los siguientes campos son correctos: Tamaño de bloque, tamaño de fragmento de bloques por grupo, fragmentos por grupo e inodos por grupo.

Lee de disco, se chequean y almacena en sbi los campos de descriptores de grupo.

Rellena la estructura sb->s_op con las operaciones propias del sitema de ficheros ext2 leidas del disco.

Realiza el mount, grabando el directorio raiz del nuevo sistema de ficheros en sb_root.

Actualiza el superbloque en disco.

Ext2_fill_super (1)

```
739 static int ext2_fill_super(struct super_block *sb, void *data, int silent)
 740 {//bh es un puntero al buffer que contiene el ext2_superblock
 741        struct buffer_head * bh;
 742        struct ext2_sb_info * sbi;
 743        struct ext2_super_block * es;
 744        struct inode *root;
 745        unsigned long block;
 746        unsigned long sb_block = get_sb_block(&data);
 747        unsigned long logic_sb_block;
 748        unsigned long offset = 0;
 749        unsigned long def_mount_opts;
 750        long ret = -EINVAL;
 751        int blocksize = BLOCK_SIZE;
 752        int db_count;
 753        int i, j;
 754        __le32 features;
 755        int err;
```

Ext2_fill_super (2)

```
756   //Se obtiene memoria para el "sbi"
757         sbi = kzalloc(sizeof(*sbi), GFP_KERNEL);
758         if (!sbi)
759                 return -ENOMEM;
760         sb->s_fs_info = sbi;
761         sbi->s_sb_block = sb_block;
 764          * Se calcula el tamaño de bloque (blocksize) para el dispositivo,
765          *  y  se utiliza como el blocksize. De lo contrario,
766          *  (o si el blocksize es menor que el de por defecto)
770         blocksize = sb_min_blocksize(sb, BLOCK_SIZE);
771         if (!blocksize) {
772                 printk ("EXT2-fs: unable to set blocksize\n");
773                 goto failed_sbi;
774         }
```

Ext2_fill_super (3)

```
776       /*
777        * Si el superbloque no comienza en el principio de un
778        * sector hardware, calcula el desplazamiento
779        */
780       if (blocksize != BLOCK_SIZE) {
781             logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
782             offset = (sb_block*BLOCK_SIZE) % blocksize;
783       } else {
784             logic_sb_block = sb_block;
785       }
786//Se lee y almacena el superbloque en una estructura buffer_head "bh"
787       if (!(bh = sb_bread(sb, logic_sb_block))) {
788             printk ("EXT2-fs: unable to read superblock\n");
789             goto failed_sbi;
790       }
```

Ext2_fill_super (4)

```
791         /* Nota: s_es debe inicializarse tan pronto como sea posible
792          * ya que algunas macro-instructiones de ext2 dependen de su
793          * valor  "es" apunta al buffer bh corregido con el offset que
794          */ contiene el superbloque ext2 leido del disco
795         es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);

//Se almacena el superbloque ext2 "es" en memoria intermedia "sbi"
796         sbi->s_es = es;
797         sb->s_magic = le16_to_cpu(es->s_magic);
798//Se verifica la validez del SB comparándolo con el número mágico
799         if (sb->s_magic != EXT2_SUPER_MAGIC)
800                 goto cantfind_ext2;
801
```

Ext2_fill_super (5)

```
802       /* Se analizan las opciones del mount  y se colocan los valores por
    defecto en "sbi"
803       def_mount_opts = le32_to_cpu(es->s_default_mount_opts);
804       if (def_mount_opts & EXT2_DEFM_DEBUG)
805             set_opt(sbi->s_mount_opt, DEBUG);
806       if (def_mount_opts & EXT2_DEFM_BSDGROUPS)
807             set_opt(sbi->s_mount_opt, GRPID);
808       if (def_mount_opts & EXT2_DEFM_UID16)
809             set_opt(sbi->s_mount_opt, NO_UID32);
810#ifdef CONFIG_EXT2_FS_XATTR
811       if (def_mount_opts & EXT2_DEFM_XATTR_USER)
812             set_opt(sbi->s_mount_opt, XATTR_USER);
813#endif
814#ifdef CONFIG_EXT2_FS_POSIX_ACL
815       if (def_mount_opts & EXT2_DEFM_ACL)
816             set_opt(sbi->s_mount_opt, POSIX_ACL);
817#endif
```

Ext2_fill_super (6)

```
819       if (le16_to_cpu(sbi->s_es->s_errors) == EXT2_ERRORS_PANIC)
820             set_opt(sbi->s_mount_opt, ERRORS_PANIC);
821       else if (le16_to_cpu(sbi->s_es->s_errors) ==
   EXT2_ERRORS_CONTINUE)
822             set_opt(sbi->s_mount_opt, ERRORS_CONT);
823       else
824             set_opt(sbi->s_mount_opt, ERRORS_RO);
825
826       sbi->s_resuid = le16_to_cpu(es->s_def_resuid);
827       sbi->s_resgid = le16_to_cpu(es->s_def_resgid);
828
829       set_opt(sbi->s_mount_opt, RESERVATION);
830//Se analizan las opciones del montaje con la función parse_options
831       if (!parse_options ((char *) data, sbi))
832             goto failed_mount;
833
```

Ext2_fill_super (7)

```
834       sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
835             ((EXT2_SB(sb)->s_mount_opt & EXT2_MOUNT_POSIX_ACL) ?
836              MS_POSIXACL : 0);
838       ext2_xip_verify_sb(sb); /* see if bdev supports xip, unset
839                                  EXT2_MOUNT_XIP if not */
840//Verificaciones de otras características generales
841       if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV &&
842          (EXT2_HAS_COMPAT_FEATURE(sb, ~0U) ||
843           EXT2_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
844           EXT2_HAS_INCOMPAT_FEATURE(sb, ~0U)))
845             printk("EXT2-fs warning: feature flags set on rev 0 fs, "
846                    "running e2fsck is recommended\n");
```

Ext2_fill_super (8)

```
848         * Check feature flags regardless of the revision level, since we
849         * previously didn't change the revision level when setting the flags,
850         * so there is a chance incompat flags are set on a rev 0 filesystem.
851         */
852        features = EXT2_HAS_INCOMPAT_FEATURE(sb,
    ~EXT2_FEATURE_INCOMPAT_SUPP);
853        if (features) {
854                printk("EXT2-fs: %s: couldn't mount because of "
855                       "unsupported optional features (%x).\n",
856                       sb->s_id, le32_to_cpu(features));
857                goto failed_mount;
858        }
```

Ext2_fill_super (9)

```
859       if (!(sb->s_flags & MS_RDONLY) &&
860           (features = EXT2_HAS_RO_COMPAT_FEATURE(sb,
    ~EXT2_FEATURE_RO_COMPAT_SUPP))){
861               printk("EXT2-fs: %s: couldn't mount RDWR because of "
862                      "unsupported optional features (%x).\n",
863                      sb->s_id, le32_to_cpu(features));
864               goto failed_mount;
865       }
866 //Se inicializa el tamaño de bloque
//Se realiza la correspondencia 0->1kB; 1->2kB; 2->4kB
867       blocksize = BLOCK_SIZE << le32_to_cpu(sbi->s_es-
    >s_log_block_size);
868
869       if (ext2_use_xip(sb) && blocksize != PAGE_SIZE) {
870               if (!silent)
871                       printk("XIP: Unsupported blocksize\n");
872               goto failed_mount;
873       }
```

Ext2_fill_super (10)

```
875       /* Se comprueba que el tamaño del blocksize sea correcto
876       if (sb->s_blocksize != blocksize) {
//Si no se liberan las memorias buffer que contienen los descriptores del fs
877               brelse(bh);
 879               if (!sb_set_blocksize(sb, blocksize)) {
880                       printk(KERN_ERR "EXT2-fs: blocksize too small for
   device.\n");
881                       goto failed_sbi;
882               }
 884               logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
885               offset = (sb_block*BLOCK_SIZE) % blocksize;
886               bh = sb_bread(sb, logic_sb_block);
887               if(!bh) {
888                       printk("EXT2-fs: Couldn't read superblock on "
889                              "2nd try.\n");
890                       goto failed_sbi;
891               }
```

Ext2_fill_super (11)

```
//Se vuelve a asignar el superbloque a las variables "es" y "sbi"
 892                es = (struct ext2_super_block *) (((char *)bh->b_data) +
    offset);
 893                sbi->s_es = es;
//Se comprueba la validez del superbloque
 894                if (es->s_magic != cpu_to_le16(EXT2_SUPER_MAGIC)) {
 895                        printk ("EXT2-fs: Magic mismatch, very weird !\n");
 896                        goto failed_mount;
 897                }
 898        }
 899
 900        sb->s_maxbytes = ext2_max_size(sb->s_blocksize_bits);
```

Ext2_fill_super (12)

```
901  //Comprobaciones acerca de los niveles de revisión
902       if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
903            sbi->s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
904            sbi->s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
905       } else {
906            sbi->s_inode_size = le16_to_cpu(es->s_inode_size);
907            sbi->s_first_ino = le32_to_cpu(es->s_first_ino);
908            if ((sbi->s_inode_size < EXT2_GOOD_OLD_INODE_SIZE) ||
909                !is_power_of_2(sbi->s_inode_size) ||
910                (sbi->s_inode_size > blocksize)) {
911                printk ("EXT2-fs: unsupported inode size: %d\n",
912                        sbi->s_inode_size);
913                goto failed_mount;
914            }
915       }
```

Ext2_fill_super (13)

```
916  //Se rellenan los campos del SB en memoria intermedia "sbi"
917       sbi->s_frag_size = EXT2_MIN_FRAG_SIZE <<
918                                 le32_to_cpu(es->s_log_frag_size);
919       if (sbi->s_frag_size == 0)
920            goto cantfind_ext2;
921       sbi->s_frags_per_block = sb->s_blocksize / sbi->s_frag_size;
922
923       sbi->s_blocks_per_group = le32_to_cpu(es->s_blocks_per_group);
924       sbi->s_frags_per_group = le32_to_cpu(es->s_frags_per_group);
925       sbi->s_inodes_per_group = le32_to_cpu(es->s_inodes_per_group);
926
927       if (EXT2_INODE_SIZE(sb) == 0)
928            goto cantfind_ext2;
929       sbi->s_inodes_per_block = sb->s_blocksize / EXT2_INODE_SIZE(sb);
930       if (sbi->s_inodes_per_block == 0 || sbi->s_inodes_per_group == 0)
931            goto cantfind_ext2;
```

Ext2_fill_super (14)

```
932       sbi->s_itb_per_group = sbi->s_inodes_per_group /
933                                   sbi->s_inodes_per_block;
934       sbi->s_desc_per_block = sb->s_blocksize /
935                                   sizeof (struct ext2_group_desc);
936       sbi->s_sbh = bh;
937       sbi->s_mount_state = le16_to_cpu(es->s_state);
938       sbi->s_addr_per_block_bits =
939             ilog2 (EXT2_ADDR_PER_BLOCK(sb));
940       sbi->s_desc_per_block_bits =
941             ilog2 (EXT2_DESC_PER_BLOCK(sb));
942//Si no es SF ext2(número mágico) --> ERROR
943       if (sb->s_magic != EXT2_SUPER_MAGIC)
944             goto cantfind_ext2;
```

Ext2_fill_super (15)

```
945  //Se comprueba que sea correcto el tamaño de bloque
946        if (sb->s_blocksize != bh->b_size) {
947              if (!silent)
948                    printk ("VFS: Unsupported blocksize on dev "
949                          "%s.\n", sb->s_id);
950              goto failed_mount;
951        }
952
953        if (sb->s_blocksize != sbi->s_frag_size) {
954              printk ("EXT2-fs: fragsize %lu != blocksize %lu (not supported
    yet)\n",
955                    sbi->s_frag_size, sb->s_blocksize);
956              goto failed_mount;
957        }
```

Ext2_fill_super (16)

Se comprueba que sea correcto el tamaño de fragmento de bloques de grupo

```
959       if (sbi->s_blocks_per_group > sb->s_blocksize * 8) {
960             printk ("EXT2-fs: #blocks per group too big: %lu\n",
961                   sbi->s_blocks_per_group);
962             goto failed_mount;
963       }//Se comprueba que sea correcto los fragmentos por grupos
964       if (sbi->s_frags_per_group > sb->s_blocksize * 8) {
965             printk ("EXT2-fs: #fragments per group too big: %lu\n",
966                   sbi->s_frags_per_group);
967             goto failed_mount;
968       }     //Se comprueba que sea correcto los inodos por grupos
969       if (sbi->s_inodes_per_group > sb->s_blocksize * 8) {
970             printk ("EXT2-fs: #inodes per group too big: %lu\n",
971                   sbi->s_inodes_per_group);
972             goto failed_mount;
973       }
```

Ext2_fill_super (17)

```
975       if (EXT2_BLOCKS_PER_GROUP(sb) == 0)
976             goto cantfind_ext2;
977       sbi->s_groups_count = ((le32_to_cpu(es->s_blocks_count) -
978                               le32_to_cpu(es->s_first_data_block) - 1)
979                                    / EXT2_BLOCKS_PER_GROUP(sb)) + 1;
980       db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /
981                  EXT2_DESC_PER_BLOCK(sb);
//Se crea memoria para buffer_head para s_group_desc
982       sbi->s_group_desc = kmalloc (db_count * sizeof (struct buffer_head *),
    GFP_KERNEL);
983       if (sbi->s_group_desc == NULL) {
984             printk ("EXT2-fs: not enough memory\n");
985             goto failed_mount;
986       }
```

Ext2_fill_super (18)

```
987       bgl_lock_init(&sbi->s_blockgroup_lock);
     //Se crea memoria para s_debts
988       sbi->s_debts = kcalloc(sbi->s_groups_count, sizeof(*sbi->s_debts),
   GFP_KERNEL);
989       if (!sbi->s_debts) {
990             printk ("EXT2-fs: not enough memory\n");
991             goto failed_mount_group_desc;
992       }//Se lee cada uno de los descriptores de grupo desde disco
993       for (i = 0; i < db_count; i++) {
994             block = descriptor_loc(sb, logic_sb_block, i);
995             sbi->s_group_desc[i] = sb_bread(sb, block);
//Se chequean los descriptores de grupo Si  error se libera la memoria asignada
996             if (!sbi->s_group_desc[i]) {
997                   for (j = 0; j < i; j++)
998                         brelse (sbi->s_group_desc[j]);
999                   printk ("EXT2-fs: unable to read group descriptors\n");
1000                   goto failed_mount_group_desc;
1001             }
```

Ext2_fill_super (19)

```
1002        }
1003        if (!ext2_check_descriptors (sb)) {
1004                printk ("EXT2-fs: group descriptors corrupted!\n");
1005                goto failed_mount2;
1006        }
1007        sbi->s_gdb_count = db_count;
1008        get_random_bytes(&sbi->s_next_generation, sizeof(u32));
1009        spin_lock_init(&sbi->s_next_gen_lock);
1010
1011        /* per fileystem reservation list head & lock */
1012        spin_lock_init(&sbi->s_rsv_window_lock);
1013        sbi->s_rsv_window_root = RB_ROOT;
1014        /*
```

Ext2_fill_super (20)

```
1015        * Add a single, static dummy reservation to the start of the
1016        * reservation window list --- it gives us a placeholder for
1017        * append-at-start-of-list which makes the allocation logic
1018        * _much_ simpler.
1020       sbi->s_rsv_window_head.rsv_start =
    EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
1021       sbi->s_rsv_window_head.rsv_end =
    EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
1022       sbi->s_rsv_window_head.rsv_alloc_hit = 0;
1023       sbi->s_rsv_window_head.rsv_goal_size = 0;
1024       ext2_rsv_window_add(sb, &sbi->s_rsv_window_head);
1025 //Se rellenan campos del superbloque de memoria intermedia "sbi"
1026       err = percpu_counter_init(&sbi->s_freeblocks_counter,
1027                                 ext2_count_free_blocks(sb));
1028       if (!err) {
1029             err = percpu_counter_init(&sbi->s_freeinodes_counter,
1030                                 ext2_count_free_inodes(sb));
1031       }
```

Ext2_fill_super (21)

```
1032    if (!err) {
1033            err = percpu_counter_init(&sbi->s_dirs_counter,
1034                    ext2_count_dirs(sb));
1035    }
1036    if (err) {
1037            printk(KERN_ERR "EXT2-fs: insufficient memory\n");
1038            goto failed_mount3;
1039    }
1040    /* Se rellena la estructura "sb" en las operaciones
1041     * relacionadas con el superbloque (sb->s_op) con las
1042     */ operaciones propias del sistema de ficheros ext2
1043    sb->s_op = &ext2_sops;
1044    sb->s_export_op = &ext2_export_ops;
1045    sb->s_xattr = ext2_xattr_handlers;
```

Ext2_fill_super (22)

```
        //Lee el nodo raiz del nuvevo fs a montar
1046      root = ext2_iget(sb, EXT2_ROOT_INO);
1047      if (IS_ERR(root)) {
1048            ret = PTR_ERR(root);
1049            goto failed_mount3;
1050      }
1051      if (!S_ISDIR(root->i_mode) || !root->i_blocks || !root->i_size) {
1052            iput(root);
1053            printk(KERN_ERR "EXT2-fs: corrupt root inode, run e2fsck\n");
1054            goto failed_mount3;
1055      }
1056
        //Almacena en root el nodo raiz del nuvevo fs a montar en root
1057      sb->s_root = d_alloc_root(root);
//Si error --> se elimina el nodo raiz
```

Ext2_fill_super (23)

```
1058        if (!sb->s_root) {
1059                iput(root);
1060                printk(KERN_ERR "EXT2-fs: get root inode failed\n");
1061                ret = -ENOMEM;
1062                goto failed_mount3;
1063        }
1064        if (EXT2_HAS_COMPAT_FEATURE(sb,
    EXT3_FEATURE_COMPAT_HAS_JOURNAL))
1065                ext2_warning(sb, __FUNCTION__,
1066                        "mounting ext3 filesystem as ext2");
     //Se llama a ext2_setup_super que incrementará el numero de montajes
     //en "es"y guardará "es" en disco
1067        ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
1068        return 0;
1069
1070  cantfind_ext2:
1071        if (!silent)
1072                printk("VFS: Can't find an ext2 filesystem on dev %s.\n",
1073                        sb->s_id);
```

Ext2_fill_super (24)

```
1074        goto failed_mount;
1075  failed_mount3:    //Etiquetas en caso de error
1076        percpu_counter_destroy(&sbi->s_freeblocks_counter);
1077        percpu_counter_destroy(&sbi->s_freeinodes_counter);
1078        percpu_counter_destroy(&sbi->s_dirs_counter);
1079  failed_mount2:
1080        for (i = 0; i < db_count; i++)
1081              brelse(sbi->s_group_desc[i]);
1082  failed_mount_group_desc:
1083        kfree(sbi->s_group_desc);
1084        kfree(sbi->s_debts);
1085  failed_mount:
1086        brelse(bh);
1087  failed_sbi:
1088        sb->s_fs_info = NULL;
1089        kfree(sbi);
1090        return ret;
1091}
```

_Ext2_parse_options (1)

- **¿Qué hace?**

Realiza un análisis de las opciones de montaje especificadas. Analiza la cadena de caracteres pasada como parámetro e inicializa las opciones de montaje.
Comprueba si se han pasado opciones de montaje. Si no se le pasa ninguna opción se sale de la función.
Se comprueba cada una de las opciones pasadas como parámetro y va inicializando las opciones de montaje.
La función devuelve 1 si es correcto, 0 en caso contrario.

_Ext2_parse_options (2)

```
430  static int parse_options (char * options,
431                            struct ext2_sb_info *sbi)
432  {
433      char * p;
434      substring_t args[MAX_OPT_ARGS];
435      int option;
436       //Se comprueba si se han pasado opciones de montaje
437      if (!options)
438              return 1;
439
440      while ((p = strsep (&options, ",")) != NULL) {
441              int token;
442              if (!*p)
443                      continue;
444
```

_Ext2_parse_options (3)

```
445             token = match_token(p, tokens, args);
446             switch (token) {
447             case Opt_bsd_df:
448                     clear_opt (sbi->s_mount_opt, MINIX_DF);
449                     break;
450             case Opt_minix_df:
451                     set_opt (sbi->s_mount_opt, MINIX_DF);
452                     break;
453             case Opt_grpid:
454                     set_opt (sbi->s_mount_opt, GRPID);
455                     break;
456             case Opt_nogrpid:
457                     clear_opt (sbi->s_mount_opt, GRPID);
458                     break;
459             case Opt_resuid:
460                     if (match_int(&args[0], &option))
461                             return 0;
462                     sbi->s_resuid = option;
463                     break;
```

_Ext2_parse_options (4)

```
464             case Opt_resgid:
465                     if (match_int(&args[0], &option))
466                             return 0;
467                     sbi->s_resgid = option;
468                     break;
469             case Opt_sb:
470                     /* handled by get_sb_block() instead of here */
471                     /* *sb_block = match_int(&args[0]); */
472                     break;
473             case Opt_err_panic:
474                     clear_opt (sbi->s_mount_opt, ERRORS_CONT);
475                     clear_opt (sbi->s_mount_opt, ERRORS_RO);
476                     set_opt (sbi->s_mount_opt, ERRORS_PANIC);
477                     break;
478             case Opt_err_ro:
479                     clear_opt (sbi->s_mount_opt, ERRORS_CONT);
480                     clear_opt (sbi->s_mount_opt, ERRORS_PANIC);
481                     set_opt (sbi->s_mount_opt, ERRORS_RO);
482                     break;
```

_Ext2_parse_options (5)

```
483             case Opt_err_cont:
484                     clear_opt (sbi->s_mount_opt, ERRORS_RO);
485                     clear_opt (sbi->s_mount_opt, ERRORS_PANIC);
486                     set_opt (sbi->s_mount_opt, ERRORS_CONT);
487                     break;
488             case Opt_nouid32:
489                     set_opt (sbi->s_mount_opt, NO_UID32);
490                     break;
491             case Opt_nocheck:
492                     clear_opt (sbi->s_mount_opt, CHECK);
493                     break;
494             case Opt_debug:
495                     set_opt (sbi->s_mount_opt, DEBUG);
496                     break;
497             case Opt_oldalloc:
498                     set_opt (sbi->s_mount_opt, OLDALLOC);
499                     break;
```

_Ext2_parse_options (6)

```
500             case Opt_orlov:
501                     clear_opt (sbi->s_mount_opt, OLDALLOC);
502                     break;
503             case Opt_nobh:
504                     set_opt (sbi->s_mount_opt, NOBH);
505                     break;
506 #ifdef CONFIG_EXT2_FS_XATTR
507             case Opt_user_xattr:
508                     set_opt (sbi->s_mount_opt, XATTR_USER);
509                     break;
510             case Opt_nouser_xattr:
511                     clear_opt (sbi->s_mount_opt, XATTR_USER);
512                     break;
513 #else
514             case Opt_user_xattr:
515             case Opt_nouser_xattr:
516                     printk("EXT2 (no)user_xattr options not supported\n");
517                     break;
518 #endif
```

_Ext2_parse_options (7)

```
519#ifdef CONFIG_EXT2_FS_POSIX_ACL
520             case Opt_acl:
521                     set_opt(sbi->s_mount_opt, POSIX_ACL);
522                     break;
523             case Opt_noacl:
524                     clear_opt(sbi->s_mount_opt, POSIX_ACL);
525                     break;
526#else
527             case Opt_acl:
528             case Opt_noacl:
529                     printk("EXT2 (no)acl options not supported\n");
530                     break;
531#endif
532             case Opt_xip:
533#ifdef CONFIG_EXT2_FS_XIP
534                     set_opt (sbi->s_mount_opt, XIP);
535#else
536                     printk("EXT2 xip option not supported\n");
537#endif
538                     break;
```

_Ext2_parse_options (8)

```
540#if defined(CONFIG_QUOTA)
541             case Opt_quota:
542             case Opt_usrquota:
543                     set_opt(sbi->s_mount_opt, USRQUOTA);
544                     break;
545
546             case Opt_grpquota:
547                     set_opt(sbi->s_mount_opt, GRPQUOTA);
548                     break;
549#else
550             case Opt_quota:
551             case Opt_usrquota:
552             case Opt_grpquota:
553                     printk(KERN_ERR
554                             "EXT2-fs: quota operations not supported.\n");
555
556                     break;
557#endif
```

_Ext2_parse_options (9)

```
558
559             case Opt_reservation:
560                     set_opt(sbi->s_mount_opt, RESERVATION);
561                     printk("reservations ON\n");
562                     break;
563             case Opt_noreservation:
564                     clear_opt(sbi->s_mount_opt, RESERVATION);
565                     printk("reservations OFF\n");
566                     break;
567             case Opt_ignore:
568                     break;
569             default:
570                     return 0;
571             }
572     }
573     return 1;
574 }
```

Ext2_check_descriptors (1)

¿Qué hace?

Verifica la validez de los descriptores de conjuntos leídos desde el disco. Para cada descriptor, verifica que los bloques de bitmap y la tabla de inodos están contenidos en el grupo.

Comprueba si se han pasado opciones de montaje. Si no se le pasa ninguna opción se sale de la función.

Comprueba cada una de las opciones pasadas como parámetro y va inicializando las opciones de montaje.

La función devuelve 1 si es correcto, 0 en caso contrario

Ext2_check_descriptors (2)

```
621static int ext2_check_descriptors(struct super_block *sb)
622{
623       int i;
624       struct ext2_sb_info *sbi = EXT2_SB(sb);
625       unsigned long first_block = le32_to_cpu(sbi->s_es->s_first_data_block);
626       unsigned long last_block;
627
628       ext2_debug ("Checking group descriptors");
629//Se van recorriendo todos los descriptores de grupo
630       for (i = 0; i < sbi->s_groups_count; i++) {
631              struct ext2_group_desc *gdp = ext2_get_group_desc(sb, i, NULL);
632//Se selecciona un descriptor de Grupo de Bloques
633              if (i == sbi->s_groups_count - 1)
634                     last_block = le32_to_cpu(sbi->s_es->s_blocks_count) - 1;
```

Ext2_check_descriptors (3)

```
635             else
636                     last_block = first_block +
637                             (EXT2_BLOCKS_PER_GROUP(sb) - 1);
638//Si bitmap de bloque no está contenido en el grupo --> ERROR
639             if (le32_to_cpu(gdp->bg_block_bitmap) < first_block ||
640                 le32_to_cpu(gdp->bg_block_bitmap) > last_block)
641             {
642                     ext2_error (sb, "ext2_check_descriptors",
643                             "Block bitmap for group %d"
644                             " not in group (block %lu)!",
645                             i, (unsigned long) le32_to_cpu(gdp->bg_block_bitmap));
646                     return 0;
647             }
```

Ext2_check_descriptors (4)

```
//Si bitmap de inodo no está contenido en el grupo --> ERROR
 648            if (le32_to_cpu(gdp->bg_inode_bitmap) < first_block ||
 649                le32_to_cpu(gdp->bg_inode_bitmap) > last_block)
 650            {
 651                    ext2_error (sb, "ext2_check_descriptors",
 652                                "Inode bitmap for group %d"
 653                                " not in group (block %lu)!",
 654                                i, (unsigned long) le32_to_cpu(gdp->bg_inode_bitmap));
 655                    return 0;
 656            }
//Si tabla de inodo no está contenido en el grupo --> ERROR
 657            if (le32_to_cpu(gdp->bg_inode_table) < first_block ||
 658                le32_to_cpu(gdp->bg_inode_table) + sbi->s_itb_per_group - 1 >
 659                last_block)
```

Ext2_check_descriptors (5)

```
660             {
661                     ext2_error (sb, "ext2_check_descriptors",
662                             "Inode table for group %d"
663                             " not in group (block %lu)!",
664                             i, (unsigned long) le32_to_cpu(gdp->bg_inode_table));
665                     return 0;
666             }
667             first_block += EXT2_BLOCKS_PER_GROUP(sb);
668     }//Si no se produjo ningún error se devuelve 1
669     return 1;
670 }
```

Ext2_setup_super (1)

¿Qué hace?
Esta función actualiza la estructura superbloque de ext2 (es) incrementado el número de montajes.

Llama a ext2_write_super para actualizar campos de (es) a partir de (sb) y escribir en el disco. Se llama cuando se monta el sistema de ficheros.

Almacena el "sb" en la memoria intermedia "sbi".

Comprueba que la versión del super bloque no es mayor que la máxima permitida, en cuyo caso se producirá un error.

Si el sistema de fichero se monto de solo lectura se sale de la función, no tiene sentido escribir en disco.

Ext2_setup_super (2)

En caso de no haberse montado correctamente el sistema de ficheros la última vez, se mostrará un mensaje con printk y se recomienda ejecutar e2fsck.

Se comprueba si el sistema de ficheros ha sido testeado, si tiene errores, si se ha superado el número máximo de montajes, si se ha superado el tiempo de revisión.

Incrementa el número de veces que se ha montado el sistema de ficheros.

Llama a la función ext2_write_super que se encargará de actualizar los campos de "es" a partir de "sb" y posterior escitura en disco.

_Ext2_setup_super (3)

```
576  static int ext2_setup_super (struct super_block * sb,
 577                               struct ext2_super_block * es,
 578                               int read_only)
 579{
 580       int res = 0;
         //Se almacena el "sb" en la memoria intermedia sbi
 581       struct ext2_sb_info *sbi = EXT2_SB(sb);
 582//Si la versión de SB es mayor que la máxima permitida --> ERROR
 583       if (le32_to_cpu(es->s_rev_level) > EXT2_MAX_SUPP_REV) {
 584               printk ("EXT2-fs warning: revision level too high, "
 585                       "forcing read-only mode\n");
 586               res = MS_RDONLY;
 587       }//Si es de solo lectura
 588       if (read_only)
 589               return res;
```

_Ext2_setup_super (4)

```
  //Se comprueba que el sistema se montó correctamente la última vez
590       if (!(sbi->s_mount_state & EXT2_VALID_FS))
591             printk ("EXT2-fs warning: mounting unchecked fs, "
592                   "running e2fsck is recommended\n");
593       else if ((sbi->s_mount_state & EXT2_ERROR_FS))
594             printk ("EXT2-fs warning: mounting fs with errors, "
595                   "running e2fsck is recommended\n");
596       else if ((__s16) le16_to_cpu(es->s_max_mnt_count) >= 0 &&
597              le16_to_cpu(es->s_mnt_count) >=
598              (unsigned short) (__s16) le16_to_cpu(es->s_max_mnt_count))
599             printk ("EXT2-fs warning: maximal mount count reached, "
600                   "running e2fsck is recommended\n");
601       else if (le32_to_cpu(es->s_checkinterval) &&
602             (le32_to_cpu(es->s_lastcheck) + le32_to_cpu(es->s_checkinterval)
   <= get_seconds()))
603             printk ("EXT2-fs warning: checktime reached, "
604                   "running e2fsck is recommended\n");
```

_Ext2_setup_super (5)

```
 605        if (!le16_to_cpu(es->s_max_mnt_count))
//Se incrementa el número de veces que se ha montado el FS
 606                es->s_max_mnt_count = cpu_to_le16(EXT2_DFL_MAX_MNT_COUNT);
 607        es->s_mnt_count=cpu_to_le16(le16_to_cpu(es->s_mnt_count) + 1);
//Actualiza los campos de "es" a partir de "sb" y escribe en disco.
 608        ext2_write_super(sb);
          //Opciones de depuración y chequeo
 609        if (test_opt (sb, DEBUG))
 610                printk ("[EXT II FS %s, %s, bs=%lu, fs=%lu, gc=%lu, "
 611                        "bpg=%lu, ipg=%lu, mo=%04lx]\n",
 612                        EXT2FS_VERSION, EXT2FS_DATE, sb->s_blocksize,
 613                        sbi->s_frag_size,
 614                        sbi->s_groups_count,
 615                        EXT2_BLOCKS_PER_GROUP(sb),
 616                        EXT2_INODES_PER_GROUP(sb),
 617                        sbi->s_mount_opt);
 618        return res;
 619}
```

_Ext2_write_super (1)

¿Qué hace?

Escribe el superbloque en el buffer asignado a disco para su posterior escritura en disco.

Se comprueba si el FS está en modo lectura/escritura.

Si es un FS válido (libre de errores), se actualizan los campos de estado (*s_state*), números de bloques, inodos libres y fecha del último montaje (*s_mtime*).

Se escribe el superbloque en disco llamando a la función ext2_sync_super.

Si no es fs valido entonces se llama a la función *ext2_commit_super, para que se escriba en disco.*

_Ext2_write_super (2)

```
1122void ext2_write_super (struct super_block * sb)
1123{
1124        struct ext2_super_block * es;
 1125        lock_kernel(); //Se bloquea el kernel
//Se comprueba que el FS no esté en modo de solo lectura
1126        if (!(sb->s_flags & MS_RDONLY)) {
1127                es = EXT2_SB(sb)->s_es;
1128//Se comprueba que sea un sistema de fichero válido
1129                if (le16_to_cpu(es->s_state) & EXT2_VALID_FS) {
1130                        ext2_debug ("setting valid to 0\n");
//Se actualiza es->s_state del SF montado
1131                        es->s_state = cpu_to_le16(le16_to_cpu(es->s_state) &
1132                                                  ~EXT2_VALID_FS);
```

_Ext2_write_super (3)

```
//Se actualiza el número de bloques y de inodos libres
1133                    es->s_free_blocks_count =
    cpu_to_le32(ext2_count_free_blocks(sb));
1134                    es->s_free_inodes_count =
    cpu_to_le32(ext2_count_free_inodes(sb));
//Se actualiza el s_mtime (fecha de la última modificación del SB
1135                    es->s_mtime = cpu_to_le32(get_seconds());
//Se escribe el superbloque "es" en disco
1136                    ext2_sync_super(sb, es);
1137            } else
//Si no es un sistema de fichero válido se actualiza el SB
1138                    ext2_commit_super (sb, es);
1139        }
1140        sb->s_dirt = 0;
      //Se desbloquea el kernel
1141        unlock_kernel();
1142}
```

Ext2_sync_super (1)

¿Qué hace?

Se encarga de actualizar el superbloque desde memoria a disco.
Actualiza campos de "es", número de bloques e inodos libres, fecha de la última escritura (s_wtime).
Llama a las funciones *mark_buffer_dirty* y *sync_dirty_buffer para escribir en el buffer y luego que se escriba en disco.*
Pone sb->s_dirt a 0 a limpio como que se actualizó.

Ext2_sync_super (2)

```
1101  static void ext2_sync_super(struct super_block *sb, struct
        ext2_super_block *es)
1102  {
1103        es->s_free_blocks_count =
                cpu_to_le32(ext2_count_free_blocks(sb));
1104        es->s_free_inodes_count =
                cpu_to_le32(ext2_count_free_inodes(sb));
      // actualiza fecha de escritura
1105        es->s_wtime = cpu_to_le32(get_seconds());
      // marca el buffer como sucio para su escritura en disco
1106        mark_buffer_dirty(EXT2_SB(sb)->s_sbh);
      // escribe el buffer en disco
1107        sync_dirty_buffer(EXT2_SB(sb)->s_sbh);
1108        sb->s_dirt = 0;
1109}
```

Ext2_commit_super (1)

¿Qué hace?

Esta función es llamada para guardar en disco las modificaciones efectuadas en el superbloque.
Guarda la fecha de la última escritura en el superbloque.
Marca como modificado el buffer que contiene el superbloque, mediante la llamada a *mark_buffer_dirty*.
De esta manera, el contenido del buffer será escrito a disco en la siguiente operación de guardar en el buffer cache.

Ext2_commit_super(2)

```
1093  static void ext2_commit_super (struct super_block * sb,
1094                                 struct ext2_super_block * es)
1095  {
     //Obtiene la hora de la última escritura en el superbloque
1096        es->s_wtime = cpu_to_le32(get_seconds());
    //Se marca como modificado el buffer que contiene el superbloque
  //De ello se encarga la función mark_buffer_dirty
1097        mark_buffer_dirty(EXT2_SB(sb)->s_sbh);
1098        sb->s_dirt = 0;
1099}
```

Ext2_put_super (1)

¿Qué hace?

Es llamada por el VFS cuando un sistema de archivos se desmonta.
Guarda el superbloque ext2 (es) en disco en caso de haberse modificado, para posteriormente eliminar las memorias ocupadas por los descriptores de grupos de bloques y la memoria intermedia del superbloque (sbi).
Comprueba en primer lugar si se ha modificado el superbloque en disco, en cuyo caso se ha de guardar.
Libera las memorias intermedias que contienen los descriptores del sistema de archivos, llamando a la función *brelse.*
Libera los punteros a esas memorias llamando a *kfree_s*.
Libera las memorias intermedias asociadas a los bloques de *bitmap* cargados en memoria.
Libera la memoria intermedia sbi que contiene el superbloque del sistema de archivos.

Ext2_put_super(2)

```
111static void ext2_put_super (struct super_block * sb)
 112{
 113        int db_count;
 114        int i;
 115        struct ext2_sb_info *sbi = EXT2_SB(sb);
 116
 117        ext2_xattr_put_super(sb);
//Se comprueba si hay que actualizar el SB en disco
 118        if (!(sb->s_flags & MS_RDONLY)) {
//Se actualiza 'es' con la memoria intermedia sbi
 119                struct ext2_super_block *es = sbi->s_es;
 120
 121                es->s_state = cpu_to_le16(sbi->s_mount_state);
      //Se actualiza en disco
 122                ext2_sync_super(sb, es);
 123        }
```

Ext2_put_super(3)

```
 //Se liberan los descriptores de los grupos mediante la función brelse
124         db_count = sbi->s_gdb_count;
125         for (i = 0; i < db_count; i++)
126                 if (sbi->s_group_desc[i])
127                         brelse (sbi->s_group_desc[i]);
  //Se liberan los punteros a esas memorias
128         kfree(sbi->s_group_desc);
129         kfree(sbi->s_debts);
  //Se liberan las memorias asociadas a los bitmaps de bloques
130         percpu_counter_destroy(&sbi->s_freeblocks_counter);
 //Se liberan las memorias asociadas a los bitmaps de inodos
131         percpu_counter_destroy(&sbi->s_freeinodes_counter);
132         percpu_counter_destroy(&sbi->s_dirs_counter);
 //Se libera la memoria intermedia sbi que contiene el superbloque
133         brelse (sbi->s_sbh);
134         sb->s_fs_info = NULL;
135         kfree(sbi);
137         return;
138}
```

Ext2_remount (1)

¿Qué hace?

Implementa la operación del sistema de ficheros *remount_fs*. Se monta de nuevo el sistema de ficheros ya montado. Se reconfigura el sistema de ficheros, no lo lee de disco de nuevo.

Decodifica las opciones de montaje con *parse_options.*

Actualizan varios campos del descriptor de superbloque.

Llama a *ext2_setup_super* para inicializar el superbloque.

Ext2_remount (2)

```
1144  static int ext2_remount (struct super_block * sb, int * flags, char * data)
1145 {
1146        struct ext2_sb_info * sbi = EXT2_SB(sb);
1147        struct ext2_super_block * es;
1148        unsigned long old_mount_opt = sbi->s_mount_opt;
1149        struct ext2_mount_options old_opts;
1150        unsigned long old_sb_flags;
1151        int err;
1152
1153        /* Store the old options */
1154        old_sb_flags = sb->s_flags;
1155        old_opts.s_mount_opt = sbi->s_mount_opt;
1156        old_opts.s_resuid = sbi->s_resuid;
1157        old_opts.s_resgid = sbi->s_resgid;
1158
```

Ext2_remount (3)

```
1161        */* Se comprueban las opciones de montaje
1162        if (!parse_options (data, sbi)) {
1163                err = -EINVAL;
1164                goto restore_opts;
1165        }
1166//Se actualizan los siguientes campos del descriptor de fichero
1167        sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
1168                ((sbi->s_mount_opt & EXT2_MOUNT_POSIX_ACL) ? MS_POSIXACL : 0);
1169
1170        ext2_xip_verify_sb(sb); /* see if bdev supports xip, unset
1171                                   EXT2_MOUNT_XIP if not */
1172
1173        if ((ext2_use_xip(sb)) && (sb->s_blocksize != PAGE_SIZE)) {
1174                printk("XIP: Unsupported blocksize\n");
1175                err = -EINVAL;
1176                goto restore_opts;
1177        }
```

Ext2_remount (4)

```
1179        es = sbi->s_es;
1180        if (((sbi->s_mount_opt & EXT2_MOUNT_XIP) !=
1181            (old_mount_opt & EXT2_MOUNT_XIP)) &&
1182            invalidate_inodes(sb))
1183                ext2_warning(sb, __FUNCTION__, "busy inodes while remounting "\
1184                            "xip remain in cache (no functional problem)");
        //Si se está en modo solo lectura se sale de la función
1185        if ((*flags & MS_RDONLY) == (sb->s_flags & MS_RDONLY))
1186                return 0;
        //Sistema en modo lectura y estaba en modo lectura/escritura
1187        if (*flags & MS_RDONLY) {
        //Se comprueba que el estado del FS sea válido
1188                if (le16_to_cpu(es->s_state) & EXT2_VALID_FS ||
1189                    !(sbi->s_mount_state & EXT2_VALID_FS))
1190                        return 0;
```

Ext2_remount (5)

```
//Se monta de nuevo una partición válida lect/esc como sólo lectura
//Luego se marca de nuevo como una partición válida
1195            es->s_state = cpu_to_le16(sbi->s_mount_state);
1196            es->s_mtime = cpu_to_le32(get_seconds());
1197        } else {
1198            __le32 ret = EXT2_HAS_RO_COMPAT_FEATURE(sb,
1199                                    ~EXT2_FEATURE_RO_COMPAT_SUPP);
1200            if (ret) {
1201                printk("EXT2-fs: %s: couldn't remount RDWR because of "
1202                       "unsupported optional features (%x).\n",
1203                       sb->s_id, le32_to_cpu(ret));
1204                err = -EROFS;
1205                goto restore_opts;
1206            }
```

Ext2_remount (6)

```
        * Si se va a volver a montar una partición lect/esc a solo lectura RDONLY,
        * se vuelve a leer el flag de válidez  (puede haber sido cambiado por e2fsck
        * desde que se montó originalmente la partición).
1212                sbi->s_mount_state = le16_to_cpu(es->s_state);
//Se llama a ext2_setup_super para incrementar el número de montajes
//en "es" y guardar en disco
1213                if (!ext2_setup_super (sb, es, 0))
1214                        sb->s_flags &= ~MS_RDONLY;
1215        }
Se actualiza el superbloque en disco con la función ext2_sync_super
1216        ext2_sync_super(sb, es);
1217        return 0;
1218restore_opts:
1219        sbi->s_mount_opt = old_opts.s_mount_opt;
1220        sbi->s_resuid = old_opts.s_resuid;
1221        sbi->s_resgid = old_opts.s_resgid;
1222        sb->s_flags = old_sb_flags;
1223        return err;
1224}
```

Ext2_statfs(1)

- **¿Qué hace?**

Implementa la operación del sistema de ficheros *statfs*.
Copia las estadísticas de uso del sistema de ficheros desde el descriptor del superbloque en la variable pasada por parámetro.
Calcula el tamaño de las cabeceras de un SB.
Lee la información del superbloque (número mágico, tamaño de bloque, número de bloque y número de bloques libres).
Almacena los datos en el buffer pasado por parámetro.

Ext2_statfs(2)

```
1226  static int ext2_statfs (struct dentry * dentry, struct kstatfs * buf)
1227{
1228        struct super_block *sb = dentry->d_sb;
1229        struct ext2_sb_info *sbi = EXT2_SB(sb);
1230        struct ext2_super_block *es = sbi->s_es;
1231        u64 fsid;
1233        if (test_opt (sb, MINIX_DF))
1234                sbi->s_overhead_last = 0;
1235        else if (sbi->s_blocks_last != le32_to_cpu(es->s_blocks_count)) {
1236                unsigned long i, overhead = 0;
1237                smp_rmb();
1240                 * Calcula los bloques que ocupan los metadatos del FS
1246                 * Todos los bloques anteriores a first_data_block son
1247                 * los metadatos
1248
1249                overhead = le32_to_cpu(es->s_first_data_block);
1250
```

Ext2_statfs(3)

```
1252            * Añade el superbloque y los grupos de bloques
1255            */
1256           for (i = 0; i < sbi->s_groups_count; i++)
1257                   overhead += ext2_bg_has_super(sb, i) +
1258                           ext2_bg_num_gdb(sb, i);
1260
1261           * Cada grupo de bloques tiene un bitmap de inodo,
1262           * un bitmap de bloque y una tabla de inodo.
1264             overhead += (sbi->s_groups_count *
1265                             (2 + sbi->s_itb_per_group));
1266             sbi->s_overhead_last = overhead;
1267             smp_wmb();
1268             sbi->s_blocks_last = le32_to_cpu(es->s_blocks_count);
1269     }
```

Ext2_statfs(4)

```
//Se lee información del sistema de ficheros a partir
1270 del superbloque En buf se almacena la salida
1271        buf->f_type = EXT2_SUPER_MAGIC;
1272        buf->f_bsize = sb->s_blocksize;
//Bloque de datos(no metadatos)
1273        buf->f_blocks = le32_to_cpu(es->s_blocks_count) – sbi->s_overhead_last;
/Los bloques disponibles, no se cuentan los reservados para el
//superusuario
1274        buf->f_bfree = ext2_count_free_blocks(sb);
1275        es->s_free_blocks_count = cpu_to_le32(buf->f_bfree);
1276        buf->f_bavail = buf->f_bfree - le32_to_cpu(es->s_r_blocks_count);
```

Ext2_statfs(5)

```
1277        if (buf->f_bfree < le32_to_cpu(es->s_r_blocks_count))
1278                buf->f_bavail = 0;
1279        buf->f_files = le32_to_cpu(es->s_inodes_count);
1280        buf->f_ffree = ext2_count_free_inodes(sb);
1281        es->s_free_inodes_count = cpu_to_le32(buf->f_ffree);
1282        buf->f_namelen = EXT2_NAME_LEN;
1283        fsid = le64_to_cpup((void *)es->s_uuid) ^
1284              le64_to_cpup((void *)es->s_uuid + sizeof(u64));
1285        buf->f_fsid.val[0] = fsid & 0xFFFFFFFFUL;
1286        buf->f_fsid.val[1] = (fsid >> 32) & 0xFFFFFFFFUL;
1287        return 0;
1288}
```

Ext2_init_ext2_fs (1)

- **¿Qué hace?**
  - Registra el sistema de ficheros EXT2.
  - Llama a la función *register_filesystem.*

Ext2_init_ext2_fs (2)

```
1409  static int __init init_ext2_fs(void)
1410  {
1411        int err = init_ext2_xattr();
1412        if (err)
1413              return err;
1414        err = init_inodecache();
1415        if (err)
1416              goto out1;
1417        err = register_filesystem(&ext2_fs_type);
1418        if (err)
1419              goto out;
1420        return 0;
1421  out:
1422        destroy_inodecache();
1423  out1:
1424        exit_ext2_xattr();
1425        return err;
1426  }
```

Ext2_error (1)

- **¿Qué hace?**
  - Imprime un mensaje de error mediante la llamada a la función printk. relacionado con el sistema de archivos
  - En función del comportamiento definido frente a errores, aborta el montaje o monta en modo de sólo lectura

Ext2_error (2)

```
43 void ext2_error (struct super_block * sb, const char * function,
44 const char * fmt, ...)
45 {
46      va_list args;
47      struct ext2_sb_info *sbi = EXT2_SB(sb);
48      struct ext2_super_block *es = sbi->s_es;
```

Si el sistema estaba montado en modo lectura/escritura se marca como erróneo y se vuelca a disco con ext2_sync_super

```
49
50       if (!(sb->s_flags & MS_RDONLY)) {
51                 sbi->s_mount_state |= EXT2_ERROR_FS;
52                 es->s_state =
53                 cpu_to_le16(le16_to_cpu(es->s_state) | EXT2_ERROR_FS
);
54                 ext2_sync_super(sb, es);
55      }
56
57      va_start(args, fmt);
58      printk(KERN_CRIT "EXT2-fs error (device %s): %s: ",sb->s_id,
function);
59      vprintk(fmt, args);
60      printk("\n");
61      va_end(args);
```

Ext2_error (3)

Se imprime información del error y en función del comportamiento ante errores se lanza un PANIC o se remonta el FS en modo solo lectura

```
62
63      if (test_opt(sb, ERRORS_PANIC))
64                      panic("EXT2-fs panic from previous error\n");
65       if (test_opt(sb, ERRORS_RO)) {
66                      printk("Remounting filesystem read-only\n");
67                     sb->s_flags |= MS_RDONLY;
68      }
69 }
```

Ext2_warning

- **¿Qué hace?**
  - Imprime un aviso mediante la llamada a la función printk.
  - La operación se ha llevado a cabo, pero hay aspectos que tener en cuenta

SUPER.C: Ext2_warning

```
71 void ext2_warning (struct super_block * sb, const char *
function,
72 const char * fmt, ...)
73 {
74     va_list args;
75
76     va_start(args, fmt);
77     printk(KERN_WARNING "EXT2-fs warning (device %s): %s: ",

78     sb->s_id, function);
79     vprintk(fmt, args);
80     printk("\n");
81     va_end(args);
82 }
```

BIBLIOGRAFÍA

- **The Linux Kernel Book**. Rémy Card, Éric Dumas y Frank Mével

- Cross-Referencing Linux (http://lxr.linux.no/blurb.html)