LECCIÓN 18: I-NODE

LECCIÓN 18: I-NODE

18.1 Introducción

El sistema de ficheros permite al núcleo mantener los ficheros en una estructura de un único árbol, en la que los nodos son directorios y las hojas son ficheros.

El sistema de ficheros esconde todas las características físicas del disco como: tipo de controladora, cabezas, cilindros, sectores y pistas; y los trata como una secuencia de bloques de un cierto tamaño.

La información de los nodos del árbol se mantienen en la estructura i-node y la del sistema de ficheros en las estructuras superbloque.

Cuando un proceso de usuario hace una llamada al sistema de ficheros, ésta es atendida por el V.F.S. (Virtual File System) y luego éste dirige la llamada al sistema de ficheros particular que se trate.

El VFS provee de una capa de abstracción para la lectura, escritura y ejecución de ficheros.

Para que el resto del S.O. vea los distintos sistemas de ficheros, el VFS oculta todas las características de los sistemas de ficheros reales montados y para ello mantiene unas estructuras de datos almacenadas en Memoria RAM del núcleo y unos procedimientos que trabajan con ellas.

Estructura de un sistema de archivos Ext2

Un sistema de archivos, de tipo Ext2, debe estar presente sobre un dispositivo físico (disquete, disco duro,...) y el contenido de este dispositivo se descompone lógicamente en varias partes, como muestra la siguiente figura:

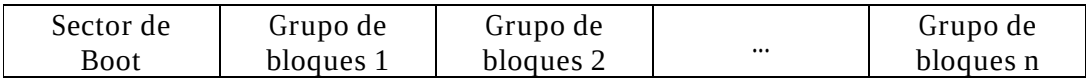


Figura 1

El sector de boot contiene la tabla de particiones y un trozo del código máquina necesario para cargar el núcleo en el arranque del sistema.

Estructura de un grupo de bloques

Cada uno de los grupos de bloques se descompone a su vez en varios elementos, como muestra la siguiente figura:

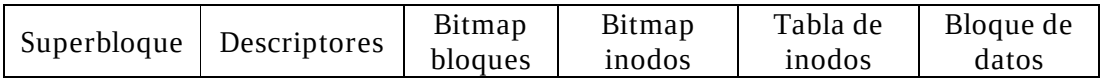


Figura 2

El superbloque: esta estructura contiene la información de control del sistema de archivos y se duplica en cada grupo de bloques para permitir paliar fácilmente una corrupción del sistema de archivos.

Una tabla de descriptores: contienen información sobre el grupo, las direcciones de bloques que contienen las informaciones cruciales, como los bloques de *bitmap* y la tabla de inodos; también se duplican en cada grupo de bloques.

Un bloque de *bitmap* para los bloques: este bloque contiene una tabla de bits: a cada bloque del grupo se le asocia un bit indicando si el bloque está asignado (el bit está entonces a 1) o disponible (el bit está a 0).

Una tabla de inodos: estos bloques contienen una parte de la tabla de inodos del sistema de archivos.

Bloques de datos: el resto de los bloques del grupo se utiliza para almacenar los datos contenidos en los archivos y los directorios.

Cuando un bloque debe asignarse, el núcleo intenta asignarlo en el mismo grupo de su inodo para minimizar el desplazamiento de las cabezas del disco en el acceso al archivo.

18.2 Estructura Inode

Un inodo, nodo-i, nodo índice o i-node en inglés, es una estructura de datos propia de los sistemas de archivos tradicionalmente empleados en los sistemas operativos tipo UNIX como es el caso de Linux. Un inodo contiene las características (permisos, fechas, ubicación, pero NO el nombre) de un archivo regular, directorio, o cualquier otro objeto que pueda contener el sistema de ficheros.

Se refiere generalmente a inodos en discos (dispositivos en modo bloque) que almacenan archivos regulares, directorios, y enlaces simbólicos.

Cada inodo queda identificado por un número entero, único dentro del sistema de ficheros, y los directorios recogen una lista de parejas formadas por un número de inodo y nombre identificativo que permite acceder al archivo en cuestión: cada archivo tiene un único inodo, pero puede tener más de un nombre en distintos o incluso en el mismo directorio para facilitar su localización.

En general en una estructura inodo se pueden distinguir dos partes:

Atributos

Son todos los campos que caracterizan al objeto en cuestión (tipo de objeto, permisos, propietario, grupo, tamaño, número de enlaces, fechas de creación, modificación y acceso).

Localización

Son las direcciones de los bloques ocupados por el fichero (objeto en general) en el sistema de ficheros. Es una tabla indexada, cuyo nivel de indexación aumenta con el tamaño del objeto. Dispone de una serie de direcciones directas (alrededor de 10) y de otras tres entradas de nivel de indexación creciente (simple, doble y triple indirecta) que apuntan a otros bloques de datos (que denominamos bloques indirectos) usados para contener direcciones de bloque.
La estructura inode es diferente para cada tipo de sistema de ficheros. Nosotros hemos elegido el sistema ext2 (second extended file system), por ser este más extensamente conocido y ser pocas sus varaciones con respecto a su sucesor ext3 en cuanto a lo que inode se refiere (han añadido 2 nuevos campos a la estructura).

La estructura ext2_inode (include/linux/ext2_fs.h) define el formato de un inodo:

TIPO	CAMPO	DESCRIPCIÓN
__le16	i_mode	Formato del fichero y permisos de acceso
__le16	i_uid	Identificador del propietario
__le32	i_size	Tamaño del archivo en bytes
__le32	i_atime	Fecha del último acceso al archivo
__le32	i_ctime	Fecha de creación del inodo
__le32	i_mtime	Fecha de última modificación del archivo
__le32	i_dtime	Fecha de supresión del archivo
__le16	i_gid	Identificador de grupo
__le16	i_links_count	Número de enlaces que apuntan al inodo
__le32	i_blocks	Número de bloques asignados para contener los datos del inodo
__le32	i_flags	Comportamiento al acceder al inodo
union	osd1	Variable de tipo unión que depende del Sistema Operativo (para Linux, contiene un campo actualmente reservado)

__le32	i_block [EXT2_N_BLOCKS]	Direcciones de bloque de datos asociados al inodo
__le32	i_generation	Número de versión asociado al inodo
__le32	i_file_acl	Dirección del descriptor de la lista de control de acceso asociada al archivo*
__le32	i_dir_acl	Dirección del descriptor de la lista de control de acceso asociada a un directorio*
__le32	i_faddr	Dirección del fragmento del archivo
union	osd2	Variable de tipo unión que depende del Sistema Operativo (para Linux, contiene una estructura con campos como el número de fragmentos, tamaño de fragmentos,...)

Tabla 1. Estructura ext2_inode

*Muchos elementos dependen del tipo de sistema operativo que se esté usando. Como en nuestro caso estamos hablando de Linux, hemos puesto directamente los campos para este sistema.

El campo i_block contiene las direcciones de bloques de datos asociadas al inodo. Es de tamaño EXT2_N_BLOCKS.

```
#define EXT2_NDIR_BLOCKS        12
#define EXT2_IND_BLOCK          EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK         (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK         (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS           (EXT2_TIND_BLOCK + 1)
```

EXT2_NDIR_BLOCKS: esta macro, nos da a conocer el valor del número total de elementos que contienen direcciones de bloques de datos comenzando por el principio de la tabla. Por lo tanto, tenemos 12 bloques al principio totalmente reservados y fijos de acceso directo.
EXT2_IND_BLOCK: dirección de un bloque que direcciona a otros bloques de datos indirectamente.
EXT2_DIND_BLOCK: dirección de un bloque que direcciona doblemente otros bloques de datos.
EXT2_TIND_BLOCK: dirección de un bloque que direcciona triplemente otros bloques de datos.

Esta explicación se comprenderá mejor atendiendo a la siguiente ilustración (limitándose a dos niveles de indirección, por razones de claridad).

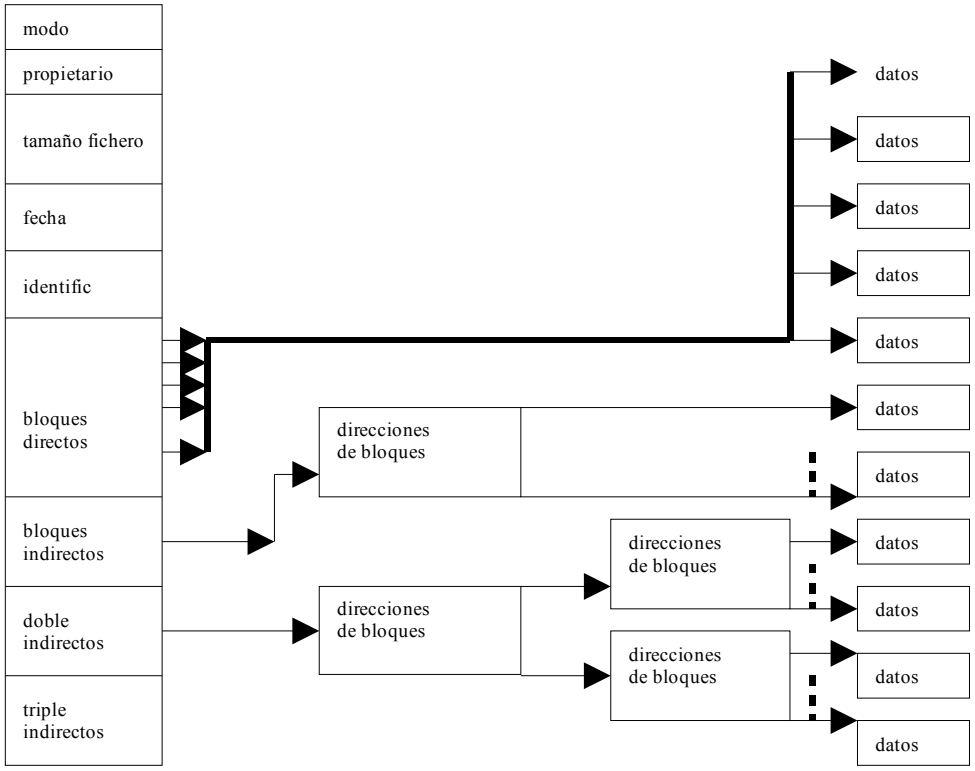


Ilustración 1. Punteros contenidos en un inodo

Estructura inodo en memoria RAM

Todo directorio o fichero que está siendo accedido, independientemente del sistema de ficheros real donde esté ubicado, se representa dentro del VFS por una estructura inode dentro de la Memoria Principal.

Cada inode se llena mediante funciones específicas del sistema de ficheros real con la información del inode en el disco más información del sistema.

Todos los inodes se combinan en una lista doblemente encadenada. Además mediante una tabla de Hash se accede a listas con los inodos que tienen el mismo valor de Hash (valor = nº disp. + nº inodos).

Los inodos modificados (bit dirty a 1) tienen que escribirse en disco.

```
 623struct inode {
 624        struct hlist_node       i_hash;
 625        struct list_head        i_list;
 626        struct list_head        i_sb_list;
 627        struct list_head        i_dentry;
 628        unsigned long           i_ino;
 629        atomic_t                i_count;
 630        unsigned int            i_nlink;
 631        uid_t                   i_uid;
 632        gid_t                   i_gid;
 633        dev_t                   i_rdev;
 634        u64                     i_version;
 635        loff_t                  i_size;
 636#ifdef __NEED_I_SIZE_ORDERED
 637        seqcount_t              i_size_seqcount;
 638#endif
 639        struct timespec         i_atime;
 640        struct timespec         i_mtime;
 641        struct timespec         i_ctime;
 642        unsigned int            i_blkbits;
 643        blkcnt_t                i_blocks;
 644        unsigned short          i_bytes;
 645        umode_t                 i_mode;
 646        spinlock_t              i_lock; /* i_blocks, i_bytes, maybe
i_size */
 647        struct mutex            i_mutex;
 648        struct rw_semaphore     i_alloc_sem;
 649        const struct inode_operations   *i_op;
 650        const struct file_operations    *i_fop; /* former ->i_op-
>default_file_ops */
 651        struct super_block      *i_sb;
 652        struct file_lock        *i_flock;
 653        struct address_space    *i_mapping;
 654        struct address_space    i_data;
 655#ifdef CONFIG_QUOTA
 656        struct dquot            *i_dquot[MAXQUOTAS];
 657#endif
 658        struct list_head        i_devices;
 659        union {
 660                struct pipe_inode_info  *i_pipe;
 661                struct block_device     *i_bdev;
 662                struct cdev             *i_cdev;
 663        };
```

```
 664        int                     i_cindex;
 665
 666        __u32                   i_generation;
 667
 668#ifdef CONFIG_DNOTIFY
 669        unsigned long           i_dnotify_mask; /* Directory notify
events */
 670        struct dnotify_struct   *i_dnotify; /* for directory
notifications */
 671#endif
 672
 673#ifdef CONFIG_INOTIFY
 674        struct list_head        inotify_watches; /* watches on this
inode */
 675        struct mutex            inotify_mutex;  /* protects the
watches list */
 676#endif
 677
 678        unsigned long           i_state;
 679        unsigned long           dirtied_when;   /* jiffies of first
dirtying */
 680
 681        unsigned int            i_flags;
 682
 683        atomic_t                i_writecount;
 684#ifdef CONFIG_SECURITY
 685        void                    *i_security;
 686#endif
 687        void                    *i_private; /* fs or device private
pointer */
 688};
 689
 690/*
 691 * inode->i_mutex nesting subclasses for the lock validator:
 692 *
 693 * 0: the object of the current VFS operation
 694 * 1: parent
 695 * 2: child/target
 696 * 3: quota file
 697 *
 698 * The locking order between these classes is
 699 * parent -> child -> normal -> xattr -> quota
 700 */
 701enum inode_i_mutex_lock_class
 702{
 703        I_MUTEX_NORMAL,
 704        I_MUTEX_PARENT,
 705        I_MUTEX_CHILD,
 706        I_MUTEX_XATTR,
 707        I_MUTEX_QUOTA
 708};
 709
 710extern void inode_double_lock(struct inode *inode1, struct inode
*inode2);
 711extern void inode_double_unlock(struct inode *inode1, struct inode
*inode2);
 712
```

Como vemos, dentro de la estructura inode encontramos un campo de importancia como es el struct inode_operations *i_op, que es un struct con los punteros a las diferentes operaciones que se van a permitir sobre el fichero. Podemos ver como existen las operaciones típicas de un fichero: el rename (cambiar el nombre), unlink (eliminar un enlace), etc.

```
1311struct inode_operations {
1312        int (*create) (struct inode *,struct dentry *,int, struct
nameidata *);
1313        struct dentry * (*lookup) (struct inode *,struct dentry *,
struct nameidata *);
1314        int (*link) (struct dentry *,struct inode *,struct dentry *);
1315        int (*unlink) (struct inode *,struct dentry *);
1316        int (*symlink) (struct inode *,struct dentry *,const char *);
1317        int (*mkdir) (struct inode *,struct dentry *,int);
1318        int (*rmdir) (struct inode *,struct dentry *);
1319        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
1320        int (*rename) (struct inode *, struct dentry *,
1321                        struct inode *, struct dentry *);
1322        int (*readlink) (struct dentry *, char __user *,int);
1323        void * (*follow_link) (struct dentry *, struct nameidata *);
1324        void (*put_link) (struct dentry *, struct nameidata *, void
*);
1325        void (*truncate) (struct inode *);
1326        int (*permission) (struct inode *, int);
1327        int (*setattr) (struct dentry *, struct iattr *);
1328        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct
kstat *);
1329        int (*setxattr) (struct dentry *, const char *,const void
*,size_t,int);
1330        ssize_t (*getxattr) (struct dentry *, const char *, void *,
size_t);
1331        ssize_t (*listxattr) (struct dentry *, char *, size_t);
1332        int (*removexattr) (struct dentry *, const char *);
1333        void (*truncate_range)(struct inode *, loff_t, loff_t);
1334        long (*fallocate)(struct inode *inode, int mode, loff_t
offset,
1335                          loff_t len);
1336        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64
start,
1337                      u64 len);
1338};
```

Los inode directorios

Los directorios se componen también de bloques de datos. Estos bloques se estructuran lógicamente en una serie de entradas. La estructura que los define es ext2_dir_entry (include/linux/ext2_fs.h):

TIPO	CAMPO	DESCRIPCIÓN
_le32	inode	Número del inodo correspondiente al archivo
_le16	rec_len	Tamaño en bytes de la entrada del directorio
_le16	name_len	Número de caracteres que componen el nombre del archivo
char[ ]	name	Nombre del archivo

18.3 Funciones

- *Funciones de creación*

Función ext2_new_inode. Fichero ialloc.c

Crea un nuevo inode ext2. La prioridad es que el inode se encuentre en el grupo del inode padre. Se intenta además balancear el número de ficheros regulares y directorios.
Dos parámetros:

*dir: dirección del directorio donde el inode tiene que ser insertado.
mode: tipo de inodo.

```
438struct inode *ext2_new_inode(struct inode *dir, int mode)
439{
440         struct super_block *sb;
441         struct buffer_head *bitmap_bh = NULL;
442         struct buffer_head *bh2;
443         int group, i;
444         ino_t ino = 0;
445         struct inode * inode;
446         struct ext2_group_desc *gdp;
447         struct ext2_super_block *es;
448         struct ext2_inode_info *ei;
449         struct ext2_sb_info *sbi;
450         int err;
451
452         sb = dir->i_sb;
```

Creamos el nuevo inodo. Si no hay memoria suficiente para crearlo, devolvemos el código de error correspondiente.

```
453         inode = new_inode(sb);
454         if (!inode)
455                 return ERR_PTR(-ENOMEM);
456
457         ei = EXT2_I(inode);
458         sbi = EXT2_SB(sb);
459         es = sbi->s_es;
460         if (S_ISDIR(mode)) {
461                 if (test_opt(sb, OLDALLOC))
462                         group = find_group_dir(sb, dir);
463                 else
```

Heurística orlov para almacenar inodes tipo fichero:
Se busca por un grupo de bloques que tengan un cierto número de inodos libres y de bloques libres. Si pasa esta condición, se mira que no existan muchos directorios, y que queden suficientes inodos libres (para almacenar ficheros). Se comprueba además el registro debt, para que sea mínimo.
Si no se consiguen estos puntos, comienza la búsqueda en el grupo del padre que contenga simplemente un número de inodos mínimo

```
464                         group = find_group_orlov(sb, dir);
465         } else
```

Primero se hace una búsqueda rápida mirando los campos free_inodes_count y free_blocks_count en busca de un grupo. Si esto falla, se hace una búsqueda lineal en todo el grupo.

```
466                 group = find_group_other(sb, dir);
467
```

Si no se ha podido encontrar el grupo, se devuelve un código de error y se marca el inodo como corrupto.

```
468         if (group == -1) {
469                 err = -ENOSPC;
470                 goto fail;
471         }
472
```

Comenzamos la búsqueda en el grupo seleccionado para almacenar el inodo.

```
473         for (i = 0; i < sbi->s_groups_count; i++) {
474                 gdp = ext2_get_group_desc(sb, group, &bh2);
475                 brelse(bitmap_bh);
```

Se carga el bitmap de inodos. Se busca por el primer bit no nulo.

```
476                 bitmap_bh = read_inode_bitmap(sb, group);
477                 if (!bitmap_bh) {
478                         err = -EIO;
479                         goto fail;
480                 }
481                 ino = 0;
482
483repeat_in_this_group:
```

Puede ser que en el grupo que hemos preseleccionado ya no queden inodos libres, con lo cual tendremos que buscar en otro grupo (incrementamos variable group y continuamos en el bucle).

```
484                 ino = ext2_find_next_zero_bit((unsigned long *)bitmap_bh-
>b_data,
485                 EXT2_INODES_PER_GROUP(sb), ino);
486                 if (ino >= EXT2_INODES_PER_GROUP(sb)) {
487                         /*
488                          * Rare race: find_group_xx() decided that there
were
489                          * free inodes in this group, but by the time we
tried
490                          * to allocate one, they're all gone.  This can
also
491                          * occur because the counters which
find_group_orlov()
492                          * uses are approximate.  So just go and search
the
493                          * next block group.
494                          */
495                         if (++group == sbi->s_groups_count)
496                                 group = 0;
497                         continue;
498                 }
499                 if (ext2_set_bit_atomic(sb_bgl_lock(sbi, group),
500                                         ino, bitmap_bh->b_data)) {
501                         /* we lost this inode */
502                         if (++ino >= EXT2_INODES_PER_GROUP(sb)) {
503                                 /* this group is exhausted, try next
group */
```

Este grupo está completo y buscamos en el siguiente.

```
504                                 if (++group == sbi->s_groups_count)
505                                         group = 0;
506                                 continue;
507                         }
508                         /* try to find free inode in the same group */
509                         goto repeat_in_this_group;
510                 }
511                 goto got;
512         }
513
514         /*
515          * Scanned all blockgroups.
516          */
517         err = -ENOSPC;
518         goto fail;
519got:
```

Marcamos el buffer como modificado.

```
520         mark_buffer_dirty(bitmap_bh);
```

Si la opción MS_SYNCHRONOUS está activa, se espera hasta que termine la operación de escritura del buffer.

```
521         if (sb->s_flags & MS_SYNCHRONOUS)
522                 sync_dirty_buffer(bitmap_bh);
523         brelse(bitmap_bh);
```

```
524
525         ino += group * EXT2_INODES_PER_GROUP(sb) + 1;
526         if (ino < EXT2_FIRST_INO(sb)
                                || ino > le32_to_cpu(es->s_inodes_count)) {
527                 ext2_error (sb, "ext2_new_inode",
528                             "reserved inode or inode > inodes count - "
529                             "block_group = %d, inode=%lu", group,
530                             (unsigned long) ino);
531                 err = -EIO;
532                 goto fail;
533         }
534
```

Decrementamos el número de inodos libres del grupo.

```
535         percpu_counter_add(&sbi->s_freeinodes_counter, -1);
```

Si es de tipo directorio, aumentamos la cuenta de directorios.

```
536         if (S_ISDIR(mode))
537                 percpu_counter_inc(&sbi->s_dirs_counter);
538
539         spin_lock(sb_bgl_lock(sbi, group));
```

Y decrementamos además el número de inodos libres del grupo.

```
540         le16_add_cpu(&gdp->bg_free_inodes_count, -1);
```

“debts” permite tener una relación entre directorios y ficheros regulares. Dependiendo del tipo incrementamos o decrementamos.

```
541         if (S_ISDIR(mode)) {
542                 if (sbi->s_debts[group] < 255)
543                         sbi->s_debts[group]++;
544                 le16_add_cpu(&gdp->bg_used_dirs_count, 1);
545         } else {
546                 if (sbi->s_debts[group])
547                         sbi->s_debts[group]--;
548         }
549         spin_unlock(sb_bgl_lock(sbi, group));
550
551         sb->s_dirt = 1;
552         mark_buffer_dirty(bh2);
553         inode->i_uid = current->fsuid;
554         if (test_opt (sb, GRPID))
555                 inode->i_gid = dir->i_gid;
556         else if (dir->i_mode & S_ISGID) {
557                 inode->i_gid = dir->i_gid;
558                 if (S_ISDIR(mode))
559                         mode |= S_ISGID;
560         } else
561                 inode->i_gid = current->fsgid;
562         inode->i_mode = mode;
563
564         inode->i_ino = ino;
565         inode->i_blocks = 0;
566         inode->i_mtime = inode->i_atime =
                                        inode->i_ctime = CURRENT_TIME_SEC;
567         memset(ei->i_data, 0, sizeof(ei->i_data));
568         ei->i_flags = EXT2_I(dir)->i_flags & ~EXT2_BTREE_FL;
569         if (S_ISLNK(mode))
```

```
570                 ei->i_flags &= ~(EXT2_IMMUTABLE_FL|EXT2_APPEND_FL);
571         /* dirsync is only applied to directories */
572         if (!S_ISDIR(mode))
573                 ei->i_flags &= ~EXT2_DIRSYNC_FL;
```

Se terminan de inicializar el resto de campos de la estructura inode.

```
574         ei->i_faddr = 0;
575         ei->i_frag_no = 0;
576         ei->i_frag_size = 0;
577         ei->i_file_acl = 0;
578         ei->i_dir_acl = 0;
579         ei->i_dtime = 0;
580         ei->i_block_alloc_info = NULL;
581         ei->i_block_group = group;
582         ei->i_dir_start_lookup = 0;
583         ei->i_state = EXT2_STATE_NEW;
584         ext2_set_inode_flags(inode);
585         spin_lock(&sbi->s_next_gen_lock);
586         inode->i_generation = sbi->s_next_generation++;
587         spin_unlock(&sbi->s_next_gen_lock);
588         insert_inode_hash(inode);
589
590         if (DQUOT_ALLOC_INODE(inode)) {
591                 err = -EDQUOT;
592                 goto fail_drop;
593         }
594
595         err = ext2_init_acl(inode, dir);
596         if (err)
597                 goto fail_free_drop;
598
599         err = ext2_init_security(inode,dir);
600         if (err)
601                 goto fail_free_drop;
602
603         mark_inode_dirty(inode);
604         ext2_debug("allocating inode %lu\n", inode->i_ino);
```

Se usa ext2_preread_inode para leer del disco el bloque que contiene el inodo. Un inodo que se ha acaba de crear seguramente será accedido en el instante siguiente.

```
605         ext2_preread_inode(inode);
606         return inode;
607
608fail_free_drop:
609         DQUOT_FREE_INODE(inode);
610
611fail_drop:
612         DQUOT_DROP(inode);
613         inode->i_flags |= S_NOQUOTA;
614         inode->i_nlink = 0;
615         iput(inode);
616         return ERR_PTR(err);
617
618fail:
619         make_bad_inode(inode);
620         iput(inode);
621         return ERR_PTR(err);
622}
```

Función ext2_get_blocks. Fichero inode.c

Buscamos en el árbol las hojas para alojar una serie de bloques.
La estrategia de asignación es simple: si tenemos que asignar algo, tendremos que ir hasta el final de la hoja. Así que lo hacemos antes de conectar nada al árbol, establecemos el vínculo entre los nuevos bloques, los escribimos si sync lo indica, recomprobamos de nuevo la ruta, liberamos y repetimos si la comprobación falla, en otro caso, ubicamos el último enlace perdido y, posiblemente, forzamos la escritura en el bloque padre.
Esto tiene una gran propiedad adicional: ninguna recuperación desde alojamientos fallidos es necesaria - simplemente liberamos bloques y no se toca nada accesible desde el inodo.

```
572static int ext2_get_blocks(struct inode *inode,
573                            sector_t iblock, unsigned long maxblocks,
574                            struct buffer_head *bh_result,
575                            int create)
576{
577        int err = -EIO;
578        int offsets[4];
579        Indirect chain[4];
580        Indirect *partial;
581        ext2_fsblk_t goal;
582        int indirect_blks;
583        int blocks_to_boundary = 0;
584        int depth;
585        struct ext2_inode_info *ei = EXT2_I(inode);
586        int count = 0;
587        ext2_fsblk_t first_block = 0;
588
```

Con ext2_blocks_to_path traducimos el número del bloque a una ruta del árbol.

```
589        depth = ext2_block_to_path(inode, iblock, offsets,
&blocks_to_boundary);
590
591        if (depth == 0)
592                return (err);
593reread:
594        partial = ext2_get_branch(inode, depth, offsets, chain, &err);
595
```

Entramos en el caso más simple, hemos encontrado un bloque en las hojas.

```
596        /* Simplest case - block found, no allocation needed */
597        if (!partial) {
598                first_block = le32_to_cpu(chain[depth - 1].key);
599                clear_buffer_new(bh_result); /* What's this do? */
600                count++;
601                /* map more blocks */
602                while (count < maxblocks && count <= blocks_to_boundary) {
603                        ext2_fsblk_t blk;
604
605                        if (!verify_chain(chain, partial)) {
606                                /*
607                                 * Indirect block might be removed by
608                                 * truncate while we were reading it.
609                                 * Handling of that case: forget what we've
610                                 * got now, go to reread.
611                                 */
612                                count = 0;
```

```
613                                     goto changed;
614                             }
615                             blk = le32_to_cpu(*(chain[depth-1].p + count));
616                             if (blk == first_block + count)
617                                     count++;
618                             else
619                                     break;
620                     }
621                     goto got_it;
622             }
623
```

Siguiente caso, búsqueda simple o lectura fallida del bloque indirecto.

```
624             /* Next simple case - plain lookup or failed read of indirect block
*/
625             if (!create || err == -EIO)
626                     goto cleanup;
627
```

Cerramos el cerrojo, esta parte la haremos en exclusión mutua.

```
628             mutex_lock(&ei->truncate_mutex);
629
630             /*
631              * Okay, we need to do block allocation.  Lazily initialize the
block
632              * allocation info here if necessary
633              */
634             if (S_ISREG(inode->i_mode) && (!ei->i_block_alloc_info))
635                     ext2_init_block_alloc_info(inode);
636
```

Buscamos el mejor lugar para alojar un bloque.

```
637             goal = ext2_find_goal(inode, iblock, partial);
638
639             /* the number of blocks need to allocate for [d,t]indirect blocks */
640             indirect_blks = (chain + depth) - partial - 1;
641             /*
642              * Next look up the indirect map to count the total number of
643              * direct blocks to allocate for this branch.
644              */
```

Contamos el número de bloques directos necesarios para ser alojados en una rama dada.

```
645             count = ext2_blks_to_allocate(partial, indirect_blks,
646                                           maxblocks, blocks_to_boundary);
647             /*
648              * XXX ???? Block out ext2_truncate while we alter the tree
649              */
```

Ahora alojamos e inicializamos el grupo de bloques. Si se producen errores, salimos de la exclusión mutua, limpiamos y salimos. Si no, conectamos la rama al inodo.

```
650             err = ext2_alloc_branch(inode, indirect_blks, &count, goal,
651                                     offsets + (partial - chain), partial);
652
653             if (err) {
654                     mutex_unlock(&ei->truncate_mutex);
```

```
655                 goto cleanup;
656         }
657
658         if (ext2_use_xip(inode->i_sb)) {
659                 /*
660                  * we need to clear the block
661                  */
662                 err = ext2_clear_xip_target (inode,
663                         le32_to_cpu(chain[depth-1].key));
664                 if (err) {
665                         mutex_unlock(&ei->truncate_mutex);
666                         goto cleanup;
667                 }
668         }
669
670         ext2_splice_branch(inode, iblock, partial, indirect_blks, count);
```

Abrimos el cerrojo, fin de la exclusión mutua.

```
671         mutex_unlock(&ei->truncate_mutex);
672         set_buffer_new(bh_result);
673got_it:
674         map_bh(bh_result, inode->i_sb, le32_to_cpu(chain[depth-1].key));
675         if (count > blocks_to_boundary)
676                 set_buffer_boundary(bh_result);
677         err = count;
678         /* Clean up and exit */
679         partial = chain + depth - 1;    /* the whole chain */
680cleanup:
681         while (partial > chain) {
682                 brelse(partial->bh);
683                 partial--;
684         }
685         return err;
686changed:
687         while (partial > chain) {
688                 brelse(partial->bh);
689                 partial--;
690         }
691         goto reread;
692}
```

Función ext2_alloc_blocks. Fichero inode.c

Se usa para asignar un grupo de bloques necesarios para una rama.
Chequea que el bloque lógico esté dentro de uno de los bloques reservados del fichero. Si ocurre esto lo obtiene y devuelve la dirección del bloque lógico. En caso de que no se corresponda con los bloques reservados, se hace una búsqueda con el ext2_new_blocks.

```
361static int ext2_alloc_blocks(struct inode *inode,
362                         ext2_fsblk_t goal, int indirect_blks, int blks,
363                         ext2_fsblk_t new_blocks[4], int *err)
364{
365         int target, i;
366         unsigned long count = 0;
367         int index = 0;
368         ext2_fsblk_t current_block = 0;
369         int ret = 0;
```

```
370
371         /*
372          * Here we try to allocate the requested multiple blocks at once,
373          * on a best-effort basis.
374          * To build a branch, we should allocate blocks for
375          * the indirect blocks (if not allocated yet), and at least
376          * the first direct block of this branch.  That's the
377          * minimum number of blocks need to allocate (required)
378          */
```

Deberemos asignar bloques a los bloques indirectos (si no están alojados aún) y, al menos, el primer bloque directo de esta rama. Ese es el mínimo número de bloques requeridos. Si no ha habido problemas, devolvemos el número de bloques directos alojados.

```
379         target = blks + indirect_blks;
380
381         while (1) {
382                 count = target;
383                 /* allocating blocks for indirect blocks and direct
blocks */
384                 current_block = ext2_new_blocks(inode,goal,&count,err);
385                 if (*err)
386                         goto failed_out;
387
388                 target -= count;
389                 /* allocate blocks for indirect blocks */
390                 while (index < indirect_blks && count) {
391                         new_blocks[index++] = current_block++;
392                         count--;
393                 }
394
395                 if (count > 0)
396                         break;
397         }
398
399         /* save the new block number for the first direct block */
400         new_blocks[index] = current_block;
401
402         /* total number of blocks allocated for direct blocks */
403         ret = count;
404         *err = 0;
405         return ret;
```

Si no se pudieron asignar nuevos bloques, liberamos los que ya habíamos asignados y devolvemos el código de error.

```
406failed_out:
407         for (i = 0; i <index; i++)
408                 ext2_free_blocks(inode, new_blocks[i], 1);
409         return ret;
410}
```

Función ext2_new_blocks. Fichero balloc.c

Función básica de asignación de bloque(s).
ext2_new_blocks usa un bloque objetivo parar indicar dónde alojarlo. Si el objetivo está libre o existe un bloque libre dentro de los 32 bloques del objetivo, ese bloque será alojado. De lo contrario, dentro de cada grupo de bloques, la búsqueda se fija primero en bytes libres enteros en el bitmap del bloque y, si falla, por cualquier bit libre del bitmap.

```
1217ext2_fsblk_t ext2_new_blocks(struct inode *inode, ext2_fsblk_t goal,
1218                     unsigned long *count, int *errp)
1219{
1220        struct buffer_head *bitmap_bh = NULL;
1221        struct buffer_head *gdp_bh;
1222        int group_no;
1223        int goal_group;
1224        ext2_grpblk_t grp_target_blk;   /* blockgroup relative goal
block */
1225        ext2_grpblk_t grp_alloc_blk;    /* blockgroup-relative allocated
block*/
1226        ext2_fsblk_t ret_block;         /* filesyetem-wide allocated
block */
1227        int bgi;                        /* blockgroup iteration index */
1228        int performed_allocation = 0;
1229        ext2_grpblk_t free_blocks;      /* number of free blocks in a
group */
1230        struct super_block *sb;
1231        struct ext2_group_desc *gdp;
1232        struct ext2_super_block *es;
1233        struct ext2_sb_info *sbi;
1234        struct ext2_reserve_window_node *my_rsv = NULL;
1235        struct ext2_block_alloc_info *block_i;
1236        unsigned short windowsz = 0;
1237        unsigned long ngroups;
1238        unsigned long num = *count;
1239
1240        *errp = -ENOSPC;
1241        sb = inode->i_sb;
```

Se comprueba la existencia del superbloque de inodos.

```
1242        if (!sb) {
1243                printk("ext2_new_blocks: nonexistent device");
1244                return 0;
1245        }
1246
1247        /*
1248         * Check quota for allocation of this block.
1249         */
1250        if (DQUOT_ALLOC_BLOCK(inode, num)) {
1251                *errp = -EDQUOT;
1252                return 0;
1253        }
1254
1255        sbi = EXT2_SB(sb);
1256        es = EXT2_SB(sb)->s_es;
1257        ext2_debug("goal=%lu.\n", goal);
1258        /*
1259         * Allocate a block from reservation only when
```

```
1260          * filesystem is mounted with reservation(default,-o reservation),
and
1261          * it's a regular file, and
1262          * the desired window size is greater than 0 (One could use ioctl
1263          * command EXT2_IOC_SETRSVSZ to set the window size to 0 to turn off
1264          * reservation on that particular file)
1265          */
1266         block_i = EXT2_I(inode)->i_block_alloc_info;
1267         if (block_i) {
1268                 windowsz = block_i->rsv_window_node.rsv_goal_size;
1269                 if (windowsz > 0)
1270                         my_rsv = &block_i->rsv_window_node;
1271         }
1272
```

Si no hay bloques libres, se devuelve el valor de “no hay espacio” en la variable de error y se deshace la asignación, en caso de que se hayan asignado algunos bloques.

```
1273         if (!ext2_has_free_blocks(sbi)) {
1274                 *errp = -ENOSPC;
1275                 goto out;
1276         }
1277
1278         /*
1279          * First, test whether the goal block is free.
1280          */
```

Comprobamos que el bloque objetivo está sin ocupar.

```
1281         if (goal < le32_to_cpu(es->s_first_data_block) ||
1282             goal >= le32_to_cpu(es->s_blocks_count))
1283                 goal = le32_to_cpu(es->s_first_data_block);
1284         group_no = (goal - le32_to_cpu(es->s_first_data_block)) /
1285                         EXT2_BLOCKS_PER_GROUP(sb);
1286         goal_group = group_no;
1287retry_alloc:
1288         gdp = ext2_get_group_desc(sb, group_no, &gdp_bh);
1289         if (!gdp)
1290                 goto io_error;
1291
1292         free_blocks = le16_to_cpu(gdp->bg_free_blocks_count);
1293         /*
1294          * if there is not enough free blocks to make a new resevation
1295          * turn off reservation for this allocation
1296          */
```

Si después de la asignación, no restasen bloques para otra futura asignación, se fija la variable my_rsv a NULL.

```
1297         if (my_rsv && (free_blocks < windowsz)
1298                 && (free_blocks > 0)
1299                 && (rsv_is_empty(&my_rsv->rsv_window)))
1300                 my_rsv = NULL;
1301
1302         if (free_blocks > 0) {
1303                 grp_target_blk = ((goal - le32_to_cpu(es-
>s_first_data_block)) %
1304                                         EXT2_BLOCKS_PER_GROUP(sb));
1305                 bitmap_bh = read_block_bitmap(sb, group_no);
1306                 if (!bitmap_bh)
1307                         goto io_error;
```

Alojamos los bloques con ext2_try_to_allocate_with_rsv(...).

```
1308                    grp_alloc_blk = ext2_try_to_allocate_with_rsv(sb,
group_no,
1309                                                  bitmap_bh, grp_target_blk,
1310                                                  my_rsv, &num);
1311                    if (grp_alloc_blk >= 0)
1312                            goto allocated;
1313            }
1314
1315            ngroups = EXT2_SB(sb)->s_groups_count;
1316            smp_rmb();
1317
1318            /*
1319             * Now search the rest of the groups.  We assume that
1320             * group_no and gdp correctly point to the last group visited.
1321             */
```

No hemos encontrado bloques disponibles. Buscamos en otros grupos.

```
1322            for (bgi = 0; bgi < ngroups; bgi++) {
1323                    group_no++;
1324                    if (group_no >= ngroups)
1325                            group_no = 0;
1326                    gdp = ext2_get_group_desc(sb, group_no, &gdp_bh);
1327                    if (!gdp)
1328                            goto io_error;
1329
1330                    free_blocks = le16_to_cpu(gdp->bg_free_blocks_count);
1331                    /*
1332                     * skip this group if the number of
1333                     * free blocks is less than half of the reservation
1334                     * window size.
1335                     */
1336                    if (my_rsv && (free_blocks <= (windowsz/2)))
1337                            continue;
1338
1339                    brelse(bitmap_bh);
1340                    bitmap_bh = read_block_bitmap(sb, group_no);
1341                    if (!bitmap_bh)
1342                            goto io_error;
1343                    /*
1344                     * try to allocate block(s) from this group, without a
goal(-1).
1345                     */
```

Encontramos sitio donde alojar los bloques. Por tanto, los alojamos.

```
1346                    grp_alloc_blk = ext2_try_to_allocate_with_rsv(sb,
group_no,
1347                                                  bitmap_bh, -1, my_rsv, &num);
1348                    if (grp_alloc_blk >= 0)
1349                            goto allocated;
1350            }
1351            /*
1352             * We may end up a bogus ealier ENOSPC error due to
1353             * filesystem is "full" of reservations, but
1354             * there maybe indeed free blocks avaliable on disk
1355             * In this case, we just forget about the reservations
1356             * just do block allocation as without reservations.
1357             */
```

```
1358        if (my_rsv) {
1359                my_rsv = NULL;
1360                windowsz = 0;
1361                group_no = goal_group;
1362                goto retry_alloc;
1363        }
1364        /* No space left on the device */
1365        *errp = -ENOSPC;
1366        goto out;
1367
1368allocated:
1369
1370        ext2_debug("using block group %d(%d)\n",
1371                        group_no, gdp->bg_free_blocks_count);
1372
1373        ret_block = grp_alloc_blk + ext2_group_first_block_no(sb,
group_no);
1374
1375        if (in_range(le32_to_cpu(gdp->bg_block_bitmap), ret_block, num)
||
1376            in_range(le32_to_cpu(gdp->bg_inode_bitmap), ret_block, num)
||
1377            in_range(ret_block, le32_to_cpu(gdp->bg_inode_table),
1378                     EXT2_SB(sb)->s_itb_per_group) ||
1379            in_range(ret_block + num - 1, le32_to_cpu(gdp-
>bg_inode_table),
1380                     EXT2_SB(sb)->s_itb_per_group)) {
1381                ext2_error(sb, "ext2_new_blocks",
1382                            "Allocating block in system zone - "
1383                            "blocks from "E2FSBLK", length %lu",
1384                            ret_block, num);
1385                /*
1386                 * ext2_try_to_allocate marked the blocks we allocated
as in
1387                 * use.  So we may want to selectively mark some of the
blocks
1388                 * as free
1389                 */
1390                goto retry_alloc;
1391        }
1392
1393        performed_allocation = 1;
1394
1395        if (ret_block + num - 1 >= le32_to_cpu(es->s_blocks_count)) {
1396                ext2_error(sb, "ext2_new_blocks",
1397                            "block("E2FSBLK") >= blocks count(%d) - "
1398                            "block_group = %d, es == %p ", ret_block,
1399                        le32_to_cpu(es->s_blocks_count), group_no, es);
1400                goto out;
1401        }
1402
```

Ajustamos los contadores de bloques libres del grupo.

```
1403        group_adjust_blocks(sb, group_no, gdp, gdp_bh, -num);
1404        percpu_counter_sub(&sbi->s_freeblocks_counter, num);
1405
```

Marcamos el buffer como modificado y si procede (MS_SYNCHRONOUS activo) lo escribimos en disco.

```
1406         mark_buffer_dirty(bitmap_bh);
1407         if (sb->s_flags & MS_SYNCHRONOUS)
1408                 sync_dirty_buffer(bitmap_bh);
1409
1410         *errp = 0;
1411         brelse(bitmap_bh);
1412         DQUOT_FREE_BLOCK(inode, *count-num);
1413         *count = num;
1414         return ret_block;
1415
1416io_error:
1417         *errp = -EIO;
1418out:
1419         /*
1420          * Undo the block allocation
1421          */
1422         if (!performed_allocation)
1423                 DQUOT_FREE_BLOCK(inode, *count);
1424         brelse(bitmap_bh);
1425         return 0;
1426}
```

Función ext2_alloc_branch. Fichero: inode.c.

Con esta función lo que hacemos es alojar y cargar una serie de bloques.

inode: inode que contendrá los bloques
offsets: desplazamientos en los bloques
branch: dónde almacenar la cadena de bloques

Utilizamos como función auxiliar ext2_alloc_blocks para almacenar los bloques.

```
438static int ext2_alloc_branch(struct inode *inode,
439                         int indirect_blks, int *blks, ext2_fsblk_t goal,
440                         int *offsets, Indirect *branch)
441{
442         int blocksize = inode->i_sb->s_blocksize;
443         int i, n = 0;
444         int err = 0;
445         struct buffer_head *bh;
446         int num;
447         ext2_fsblk_t new_blocks[4];
448         ext2_fsblk_t current_block;
449
```

Asignamos los bloques. Si no se ha podido, devolvemos un código de error y salimos. En otro caso, los inicializamos.

```
450         num = ext2_alloc_blocks(inode, goal, indirect_blks,
451                                 *blks, new_blocks, &err);
452         if (err)
453                 return err;
454
```

Se han podido asignar los bloques.

```
455         branch[0].key = cpu_to_le32(new_blocks[0]);
456         /*
457          * metadata blocks and data blocks are allocated.
458          */
459         for (n = 1; n <= indirect_blks;  n++) {
460                 /*
461                  * Get buffer_head for parent block, zero it out
462                  * and set the pointer to new one, then send
463                  * parent to disk.
464                  */
465                 bh = sb_getblk(inode->i_sb, new_blocks[n-1]);
466                 branch[n].bh = bh;
467                 lock_buffer(bh);
468                 memset(bh->b_data, 0, blocksize);
469                 branch[n].p = (__le32 *) bh->b_data + offsets[n];
470                 branch[n].key = cpu_to_le32(new_blocks[n]);
471                 *branch[n].p = branch[n].key;
472                 if (n == indirect_blks) {
473                         current_block = new_blocks[n];
474                         /*
475                          * End of chain, update the last new metablock of
476                          * the chain to point to the new allocated
477                          * data blocks numbers
478                          */
479                         for (i=1; i < num; i++)
480                                 *(branch[n].p + i) =
cpu_to_le32(++current_block);
481                 }
482                 set_buffer_uptodate(bh);
483                 unlock_buffer(bh);
484                 mark_buffer_dirty_inode(bh, inode);
485                 /* We used to sync bh here if IS_SYNC(inode).
486                  * But we now rely upon generic_osync_inode()
487                  * and b_inode_buffers. But not for directories.
488                  */
489                 if (S_ISDIR(inode->i_mode) && IS_DIRSYNC(inode))
490                         sync_dirty_buffer(bh);
491         }
492         *blks = num;
493         return err;
494}
```

- *Funciones de liberación*

Función ext2_delete_inode. Fichero: inode.c

La función ext2_delete_inode se encarga de eliminar un inodo (y limpiar toda la información asociada a ella).

```
60void ext2_delete_inode (struct inode * inode)
61{
```

Vaciamos las páginas de datos asociadas.

```
62        truncate_inode_pages(&inode->i_data, 0);
63
```

Si el nodo ha sido invalidado porque tiene errores, saltamos a no_delete y, mediante clear_inode(...), lo marcamos como inútil.

```
64        if (is_bad_inode(inode))
65                goto no_delete;
66        EXT2_I(inode)->i_dtime  = get_seconds();
67        mark_inode_dirty(inode);
```

Actualizamos el inodo en disco, borrando sus campos.

```
68        ext2_update_inode(inode, inode_needs_sync(inode));
69
70        inode->i_size = 0;
```

Si tiene bloques asignados, se eliminan.

```
71        if (inode->i_blocks)
72                ext2_truncate (inode);
```

Ahora se elimina el inodo después de haber liberado los bloques asociados.

```
73        ext2_free_inode (inode);
74
75        return;
76no_delete:
77        clear_inode(inode);  /* We must guarantee clearing of inode... */
78}
```

Función ext2_free_inode. Fichero ialloc.c

Se libera un inodo. El único parámetro que se le pasa es la dirección del propio inodo.

```
104void ext2_free_inode (struct inode * inode)
105{
106         struct super_block * sb = inode->i_sb;
107         int is_directory;
108         unsigned long ino;
109         struct buffer_head *bitmap_bh = NULL;
110         unsigned long block_group;
111         unsigned long bit;
112         struct ext2_super_block * es;
113
114         ino = inode->i_ino;
115         ext2_debug ("freeing inode %lu\n", ino);
116
117         /*
118          * Note: we must free any quota before locking the superblock,
119          * as writing the quota to disk may need the lock as well.
120          */
121         if (!is_bad_inode(inode)) {
122                 /* Quota is already initialized in iput() */
123                 ext2_xattr_delete_inode(inode);
124                 DQUOT_FREE_INODE(inode);
125                 DQUOT_DROP(inode);
126         }
127
128         es = EXT2_SB(sb)->s_es;
129         is_directory = S_ISDIR(inode->i_mode);
130
```

Usa la función clear_inode del superbloque (del VFS). Cuando dicho inodo se encuentra en un dispositivo, lo elimina de la lista de inodos de dichos dispositivos.
Pone el inodo a I_CLEAR (puede ser eliminado sin problemas).

```
131         /* Do this BEFORE marking the inode not in use or returning an
error */
132         clear_inode (inode);
133
134         if (ino < EXT2_FIRST_INO(sb) ||
135             ino > le32_to_cpu(es->s_inodes_count)) {
136                 ext2_error (sb, "ext2_free_inode",
137                             "reserved or nonexistent inode %lu", ino);
138                 goto error_return;
139         }
140         block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
141         bit = (ino - 1) % EXT2_INODES_PER_GROUP(sb);
142         brelse(bitmap_bh);
```

Leemos la tabla bitmap.

```
143         bitmap_bh = read_inode_bitmap(sb, block_group);
144         if (!bitmap_bh)
145                 goto error_return;
146
147         /* Ok, now we can actually update the inode bitmaps.. */
148         if (!ext2_clear_bit_atomic(sb_bgl_lock(EXT2_SB(sb), block_group),
149                                    bit, (void *) bitmap_bh->b_data))
```

```
150                 ext2_error (sb, "ext2_free_inode",
151                                 "bit already cleared for inode %lu", ino);
152         else
```

Incrementa el free_inodes_count del grupo de bloques. Si es un directorio, además decrementa el registro bg_used_dirs_count, actualizando también el registro s_dirs_counter en el superbloque. Finalmente marca el bit a 1.

```
153                 ext2_release_inode(sb, block_group, is_directory);
```

Se limpia el bit que corresponde al inodo en el bitmap y se marca como sucio en el buffer.

```
154         mark_buffer_dirty(bitmap_bh);
```

Si la opción MS_SYNCHRONOUS está activa, se espera hasta que termine la operación de escritura del buffer.

```
155         if (sb->s_flags & MS_SYNCHRONOUS)
156                 sync_dirty_buffer(bitmap_bh);
157error_return:
158         brelse(bitmap_bh);
159}
```

Función ext2_free_blocks. Fichero balloc.c

Cuando se quiere eliminar un fichero, se debe marcar todos los bloques de datos como no marcados. Esto lo hace ext2_truncate(), que recibe el objeto inode. Lo que hace ext2_truncate es buscar en el vector i_block eliminando todos los bloques con la función ext2_free_blocks().
ext2_free_blocks se encarga de eliminar uno o varios bloques. Se le pasa el inodo donde se encuentran los bloques, el bloque lógico que queremos eliminar, y en count el número de bloques adyacentes.

```
486void ext2_free_blocks (struct inode * inode, unsigned long block,
487                         unsigned long count)
488{
489         struct buffer_head *bitmap_bh = NULL;
490         struct buffer_head * bh2;
491         unsigned long block_group;
492         unsigned long bit;
493         unsigned long i;
494         unsigned long overflow;
495         struct super_block * sb = inode->i_sb;
496         struct ext2_sb_info * sbi = EXT2_SB(sb);
497         struct ext2_group_desc * desc;
498         struct ext2_super_block * es = sbi->s_es;
499         unsigned freed = 0, group_freed;
500
501         if (block < le32_to_cpu(es->s_first_data_block) ||
502             block + count < block ||
503             block + count > le32_to_cpu(es->s_blocks_count)) {
504                 ext2_error (sb, "ext2_free_blocks",
505                             "Freeing blocks not in datazone - "
506                             "block = %lu, count = %lu", block, count);
507                 goto error_return;
508         }
509
510         ext2_debug ("freeing block(s) %lu-%lu\n", block, block + count - 1);
```

```
511
512do_more:
513         overflow = 0;
```

Sacamos el número del grupo de bloque donde se encuentra el bloque que queremos desalojar y el bit del bitmap.

```
514         block_group = (block - le32_to_cpu(es->s_first_data_block)) /
515                       EXT2_BLOCKS_PER_GROUP(sb);
516         bit = (block - le32_to_cpu(es->s_first_data_block)) %
517                       EXT2_BLOCKS_PER_GROUP(sb);
518         /*
519          * Check to see if we are freeing blocks across a group
520          * boundary.
521          */
522         if (bit + count > EXT2_BLOCKS_PER_GROUP(sb)) {
523                 overflow = bit + count - EXT2_BLOCKS_PER_GROUP(sb);
524                 count -= overflow;
525         }
526         brelse(bitmap_bh);
527         bitmap_bh = read_block_bitmap(sb, block_group);
528         if (!bitmap_bh)
529                 goto error_return;
530
```

Obtenemos la descripción del grupo.

```
531         desc = ext2_get_group_desc (sb, block_group, &bh2);
532         if (!desc)
533                 goto error_return;
534
535         if (in_range (le32_to_cpu(desc->bg_block_bitmap), block, count)
||
536             in_range (le32_to_cpu(desc->bg_inode_bitmap), block, count)
||
537             in_range (block, le32_to_cpu(desc->bg_inode_table),
538                       sbi->s_itb_per_group) ||
539             in_range (block + count - 1, le32_to_cpu(desc-
>bg_inode_table),
540                       sbi->s_itb_per_group)) {
541                 ext2_error (sb, "ext2_free_blocks",
542                             "Freeing blocks in system zones - "
543                             "Block = %lu, count = %lu",
544                             block, count);
545                 goto error_return;
546         }
547
548         for (i = 0, group_freed = 0; i < count; i++) {
549                 if (!ext2_clear_bit_atomic(sb_bgl_lock(sbi, block_group),
550                                                 bit + i, bitmap_bh-
>b_data)) {
551                         ext2_error(sb, __func__,
552                                 "bit already cleared for block %lu",
block + i);
553                 } else {
```

Hemos conseguido liberar un bloque y seguimos en caso de tener más bloque que liberar.

```
554                         group_freed++;
555                 }
556         }
```

```
557
```

Marcamos los bloques como modificados.

```
558         mark_buffer_dirty(bitmap_bh);
559         if (sb->s_flags & MS_SYNCHRONOUS)
560                 sync_dirty_buffer(bitmap_bh);
561
```

La función group_adjust_blocks se encargará de disminuir el valor de los registros implicados en contar el número de bloques en el descriptor de grupo.

```
562         group_adjust_blocks(sb, block_group, desc, bh2, group_freed);
563         freed += group_freed;
564
565         if (overflow) {
566                 block += count;
567                 count = overflow;
568                 goto do_more;
569         }
570error_return:
571         brelse(bitmap_bh);
572         release_blocks(sb, freed);
573         DQUOT_FREE_BLOCK(inode, freed);
574}
```

- *Otras funciones generales*

Función read_inode_bitmap. Fichero ialloc.c

Lee la estructura bitmap de inodos de un determinado grupo, usando para ello la información de descriptor de fichero.

```
45 static struct buffer_head *
46read_inode_bitmap(struct super_block * sb, unsigned long block_group)
47 {
48         struct ext2_group_desc *desc;
49         struct buffer_head *bh = NULL;
50
```

Obtenemos el descriptor del grupo de bloques. Si no es posible obtenerlo, devolvemos un código de error.

```
51desc = ext2_get_group_desc(sb, block_group, NULL);
52         if (!desc)
53                 goto error_out;
54
```

Leemos el bloque especificado y retornamos el buffer de cabeza que lo contiene.

```
55bh = sb_bread(sb, le32_to_cpu(desc->bg_inode_bitmap));
56         if (!bh)
57ext2_error(sb, "read_inode_bitmap",
58"Cannot read inode bitmap - "
59"block_group = %lu, inode_bitmap = %u",
60                            block_group, le32_to_cpu(desc-
```

```
>bg_inode_bitmap));
61 error_out:
62         return bh;
63 }
```

Función ext2_block_to_path. Fichero inode.c.

Esta función traduce el número de bloque en una ruta dentro del árbol. Devuelve la longitud de la ruta y en la variable offsets estará el puntero al siguiente nodo desde el puntero al nodo actual.

```
129static int ext2_block_to_path(struct inode *inode,
130                         long i_block, int offsets[4], int *boundary)
131{
132        int ptrs = EXT2_ADDR_PER_BLOCK(inode->i_sb);
133        int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(inode->i_sb);
134        const long direct_blocks = EXT2_NDIR_BLOCKS,
135                indirect_blocks = ptrs,
136                double_blocks = (1 << (ptrs_bits * 2));
137        int n = 0;
138        int final = 0;
139
```

Comprobamos el nivel de indirección al que se encuentra el bloque indicado.

```
140        if (i_block < 0) {
141                ext2_warning (inode->i_sb, "ext2_block_to_path", "block <
0");
142        } else if (i_block < direct_blocks) {
```

Bloques de acceso directo.

```
143                offsets[n++] = i_block;
144                final = direct_blocks;
145        } else if ( (i_block -= direct_blocks) < indirect_blocks) {
```

Bloques de acceso indirecto.

```
146                offsets[n++] = EXT2_IND_BLOCK;
147                offsets[n++] = i_block;
148                final = ptrs;
149        } else if ((i_block -= indirect_blocks) < double_blocks) {
```

Bloques de acceso doblemente indirecto.

```
150                offsets[n++] = EXT2_DIND_BLOCK;
151                offsets[n++] = i_block >> ptrs_bits;
152                offsets[n++] = i_block & (ptrs - 1);
153                final = ptrs;
154        } else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) <
ptrs) {
```

Bloques de acceso triplemente indirecto.

```
155                offsets[n++] = EXT2_TIND_BLOCK;
156                offsets[n++] = i_block >> (ptrs_bits * 2);
157                offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
158                offsets[n++] = i_block & (ptrs - 1);
```

```
159                 final = ptrs;
160         } else {
161                 ext2_warning (inode->i_sb, "ext2_block_to_path", "block >
big");
162         }
163         if (boundary)
164                 *boundary = final - 1 - (i_block & (ptrs - 1));
165
166         return n;
167}
```

Función ext2_find_goal. Fichero inode.c

Busca el mejor lugar para alojar un bloque. Se usa una heurística para buscar el nodo con la mejor posición. Intenta buscar el bloque consecutivo al último bloque del inode. Si no lo consigue coge el primer bloque del inode.

```
 294static inline ext2_fsblk_t ext2_find_goal(struct inode *inode, long block,
 295                                           Indirect *partial)
 296{
 297         struct ext2_block_alloc_info *block_i;
 298
 299         block_i = EXT2_I(inode)->i_block_alloc_info;
 300
 301         /*
 302          * try the heuristic for sequential allocation,
 303          * failing that at least try to get decent locality.
 304          */
 305         if (block_i && (block == block_i->last_alloc_logical_block + 1)
 306                 && (block_i->last_alloc_physical_block != 0)) {
 307                 return block_i->last_alloc_physical_block + 1;
 308         }
 309
```

Si no encontramos un objetivo óptimo, simplemente llamamos a ext2_find_near.

```
 310         return ext2_find_near(inode, partial);
 311}
```

Función ext2_find_near. Fichero inode.c

Buscamos un lugar para alojar un bloque que sea lo más próximo al inodo. Sus reglas son:
- Si existe un bloque anterior a la posición actual, lo aloja cerca
- Si el puntero apunta a bloques indirectos, lo aloja cerca de ese bloque
- Si el puntero apunta a un inodo, lo aloja en el mismo grupo de cilindros.

```
258static ext2_fsblk_t ext2_find_near(struct inode *inode, Indirect *ind)
259{
260         struct ext2_inode_info *ei = EXT2_I(inode);
261         __le32 *start = ind->bh ? (__le32 *) ind->bh->b_data : ei-
>i_data;
262         __le32 *p;
263         ext2_fsblk_t bg_start;
264         ext2_fsblk_t colour;
265
```

```
266         /* Try to find previous block */
```

Buscamos un bloque anterior.

```
267         for (p = ind->p - 1; p >= start; p--)
268                 if (*p)
269                         return le32_to_cpu(*p);
270
271         /* No such thing, so let's try location of indirect block */
```

Si esto último falla, miramos si se apunta a un bloque indirecto, y lo alojamos cerca del mismo.

```
272         if (ind->bh)
273                 return ind->bh->b_blocknr;
274
275         /*
276          * It is going to be refered from inode itself? OK, just put it into
277          * the same cylinder group then.
278          */
```

Si el puntero apunta al inodo, lo que hacemos es asignarlo al cilindro donde se encuentra.

```
279         bg_start = ext2_group_first_block_no(inode->i_sb, ei-
>i_block_group);
280         colour = (current->pid % 16) *
281                         (EXT2_BLOCKS_PER_GROUP(inode->i_sb) / 16);
282         return bg_start + colour;
283}
```

- *Funciones de lectura.*

Función ext2_iget. Fichero inode.c

Esta función permite leer un inode del disco y copiarlo a otro en memoria. Se le pasa una estructura inode que como recordamos pertenece al VFS.

```
1191struct inode *ext2_iget (struct super_block *sb, unsigned long ino)
1192{
1193         struct ext2_inode_info *ei;
1194         struct buffer_head * bh;
1195         struct ext2_inode *raw_inode;
1196         struct inode *inode;
1197         long ret = -EIO;
1198         int n;
1199
```

Obtenemos el inodo de disco.

```
1200         inode = iget_locked(sb, ino);
1201         if (!inode)
1202                 return ERR_PTR(-ENOMEM);
1203         if (!(inode->i_state & I_NEW))
1204                 return inode;
1205
1206         ei = EXT2_I(inode);
1207#ifdef CONFIG_EXT2_FS_POSIX_ACL
1208         ei->i_acl = EXT2_ACL_NOT_CACHED;
```

```
1209        ei->i_default_acl = EXT2_ACL_NOT_CACHED;
1210#endif
1211        ei->i_block_alloc_info = NULL;
1212
```

Obtenemos el inodo ext2. Si no se pudo obtener, marca el inodo como muerto y retorna un código de error.

```
1213        raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
1214        if (IS_ERR(raw_inode)) {
1215                ret = PTR_ERR(raw_inode);
1216                goto bad_inode;
1217        }
1218
```

Copiamos los campos del inodo.

```
1219        inode->i_mode = le16_to_cpu(raw_inode->i_mode);
1220        inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
1221        inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
1222        if (!(test_opt (inode->i_sb, NO_UID32))) {
1223                inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) <<
16;
1224                inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) <<
16;
1225        }
1226        inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
1227        inode->i_size = le32_to_cpu(raw_inode->i_size);
1228        inode->i_atime.tv_sec = (signed)le32_to_cpu(raw_inode->i_atime);
1229        inode->i_ctime.tv_sec = (signed)le32_to_cpu(raw_inode->i_ctime);
1230        inode->i_mtime.tv_sec = (signed)le32_to_cpu(raw_inode->i_mtime);
1231        inode->i_atime.tv_nsec = inode->i_mtime.tv_nsec = inode-
>i_ctime.tv_nsec = 0;
1232        ei->i_dtime = le32_to_cpu(raw_inode->i_dtime);
1233        /* We now have enough fields to check if the inode was active or
not.
1234         * This is needed because nfsd might try to access dead inodes
1235         * the test is that same one that e2fsck uses
1236         * NeilBrown 1999oct15
1237         */
1238        if (inode->i_nlink == 0 && (inode->i_mode == 0 || ei->i_dtime))
{
1239                /* this inode is deleted */
1240                brelse (bh);
1241                ret = -ESTALE;
1242                goto bad_inode;
1243        }
1244        inode->i_blocks = le32_to_cpu(raw_inode->i_blocks);
1245        ei->i_flags = le32_to_cpu(raw_inode->i_flags);
1246        ei->i_faddr = le32_to_cpu(raw_inode->i_faddr);
1247        ei->i_frag_no = raw_inode->i_frag;
1248        ei->i_frag_size = raw_inode->i_fsize;
1249        ei->i_file_acl = le32_to_cpu(raw_inode->i_file_acl);
1250        ei->i_dir_acl = 0;
1251        if (S_ISREG(inode->i_mode))
1252                inode->i_size |= ((__u64)le32_to_cpu(raw_inode-
>i_size_high)) << 32;
1253        else
1254                ei->i_dir_acl = le32_to_cpu(raw_inode->i_dir_acl);
1255        ei->i_dtime = 0;
1256        inode->i_generation = le32_to_cpu(raw_inode->i_generation);
```

```
1257        ei->i_state = 0;
1258        ei->i_block_group = (ino - 1) / EXT2_INODES_PER_GROUP(inode-
>i_sb);
1259        ei->i_dir_start_lookup = 0;
1260
1261        /*
1262         * NOTE! The in-memory inode i_data array is in little-endian
order
1263         * even on big-endian machines: we do NOT byteswap the block
numbers!
1264         */
```

Se copian los bloques del inodo en memoria.

```
1265        for (n = 0; n < EXT2_N_BLOCKS; n++)
1266                ei->i_data[n] = raw_inode->i_block[n];
1267
```

Según el tipo de inodo, se cambia el valor del registro i_op para poder aplicar las operaciones sobre el inodo.

```
1268        if (S_ISREG(inode->i_mode)) {
1269                inode->i_op = &ext2_file_inode_operations;
1270                if (ext2_use_xip(inode->i_sb)) {
1271                        inode->i_mapping->a_ops = &ext2_aops_xip;
1272                        inode->i_fop = &ext2_xip_file_operations;
1273                } else if (test_opt(inode->i_sb, NOBH)) {
1274                        inode->i_mapping->a_ops = &ext2_nobh_aops;
1275                        inode->i_fop = &ext2_file_operations;
1276                } else {
1277                        inode->i_mapping->a_ops = &ext2_aops;
1278                        inode->i_fop = &ext2_file_operations;
1279                }
1280        } else if (S_ISDIR(inode->i_mode)) {
1281                inode->i_op = &ext2_dir_inode_operations;
1282                inode->i_fop = &ext2_dir_operations;
1283                if (test_opt(inode->i_sb, NOBH))
1284                        inode->i_mapping->a_ops = &ext2_nobh_aops;
1285                else
1286                        inode->i_mapping->a_ops = &ext2_aops;
1287        } else if (S_ISLNK(inode->i_mode)) {
1288                if (ext2_inode_is_fast_symlink(inode)) {
1289                        inode->i_op =
&ext2_fast_symlink_inode_operations;
1290                        nd_terminate_link(ei->i_data, inode->i_size,
1291                                sizeof(ei->i_data) - 1);
1292                } else {
1293                        inode->i_op = &ext2_symlink_inode_operations;
1294                        if (test_opt(inode->i_sb, NOBH))
1295                                inode->i_mapping->a_ops =
&ext2_nobh_aops;
1296                        else
1297                                inode->i_mapping->a_ops = &ext2_aops;
1298                }
1299        } else {
1300                inode->i_op = &ext2_special_inode_operations;
1301                if (raw_inode->i_block[0])
1302                        init_special_inode(inode, inode->i_mode,
1303                           old_decode_dev(le32_to_cpu(raw_inode-
>i_block[0])));
1304                else
```

```
1305                        init_special_inode(inode, inode->i_mode,
1306                           new_decode_dev(le32_to_cpu(raw_inode-
>i_block[1])));
1307        }
1308        brelse (bh);
1309        ext2_set_inode_flags(inode);
1310        unlock_new_inode(inode);
1311        return inode;
1312
1313bad_inode:
1314        iget_failed(inode);
1315        return ERR_PTR(ret);
1316}
```

Función ext2_get_inode. Fichero inode.c

Con esta función devolvemos una estructura ext2_inode al pasarle un superbloque y una dirección del inode.

```
1113static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
1114                                        struct buffer_head **p)
1115{
1116        struct buffer_head * bh;
1117        unsigned long block_group;
1118        unsigned long block;
1119        unsigned long offset;
1120        struct ext2_group_desc * gdp;
1121
```

Comprueba si el inodo es correcto.

```
1122        *p = NULL;
1123        if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
1124            ino > le32_to_cpu(EXT2_SB(sb)->s_es->s_inodes_count))
1125                goto Einval;
1126
```

Se calcula el bloque en el que se encuentra el inodo.

```
1127        block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
1128        gdp = ext2_get_group_desc(sb, block_group, NULL);
1129        if (!gdp)
1130                goto Egdp;
1131        /*
1132         * Figure out the offset within the block group inode table
1133         */
1134        offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) *
EXT2_INODE_SIZE(sb);
1135        block = le32_to_cpu(gdp->bg_inode_table) +
1136                (offset >> EXT2_BLOCK_SIZE_BITS(sb));
```

Leemos el descriptor del grupo usando la función sb_bread. Una vez conseguido esto, obtenemos el inodo junto con el desplazamiento.

```
1137        if (!(bh = sb_bread(sb, block)))
1138                goto Eio;
1139
1140        *p = bh;
```

```
1141        offset &= (EXT2_BLOCK_SIZE(sb) - 1);
1142        return (struct ext2_inode *) (bh->b_data + offset);
1143
1144Einval:
1145        ext2_error(sb, "ext2_get_inode", "bad inode number: %lu",
1146                   (unsigned long) ino);
1147        return ERR_PTR(-EINVAL);
1148Eio:
1149        ext2_error(sb, "ext2_get_inode",
1150                   "unable to read inode block - inode=%lu, block=%lu",
1151                   (unsigned long) ino, block);
1152Egdp:
1153        return ERR_PTR(-EIO);
1154}
```

- *Funciones de actualización de inode.*

Función ext2_update_inode. Fichero inode.c

Se encarga de actualizar el inodo en disco. Para ello usa ext2_get_inode. Luego se modifica dicho inodo y se escribe.

```
1317static int ext2_update_inode(struct inode * inode, int do_sync)
1318{
1319        struct ext2_inode_info *ei = EXT2_I(inode);
1320        struct super_block *sb = inode->i_sb;
1321        ino_t ino = inode->i_ino;
1322        uid_t uid = inode->i_uid;
1323        gid_t gid = inode->i_gid;
1324        struct buffer_head * bh;
```

Obtenemos el inodo del disco.

```
1325        struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
1326        int n;
1327        int err = 0;
1328
1329        if (IS_ERR(raw_inode))
1330                return -EIO;
1331
1332        /* For fields not not tracking in the in-memory inode,
1333         * initialise them to zero for new inodes. */
```

A partir de ahora se van copiando cada uno de los campos del inodo a la estructura ext2_inode.

```
1334        if (ei->i_state & EXT2_STATE_NEW)
1335                memset(raw_inode, 0, EXT2_SB(sb)->s_inode_size);
1336
1337        ext2_get_inode_flags(ei);
1338        raw_inode->i_mode = cpu_to_le16(inode->i_mode);
1339        if (!(test_opt(sb, NO_UID32))) {
1340                raw_inode->i_uid_low = cpu_to_le16(low_16_bits(uid));
1341                raw_inode->i_gid_low = cpu_to_le16(low_16_bits(gid));
1342/*
1343 * Fix up interoperability with old kernels. Otherwise, old inodes get
1344 * re-used with the upper 16 bits of the uid/gid intact
1345 */
1346                if (!ei->i_dtime) {
```

```
1347                        raw_inode->i_uid_high =
cpu_to_le16(high_16_bits(uid));
1348                        raw_inode->i_gid_high =
cpu_to_le16(high_16_bits(gid));
1349                } else {
1350                        raw_inode->i_uid_high = 0;
1351                        raw_inode->i_gid_high = 0;
1352                }
1353        } else {
1354                raw_inode->i_uid_low = cpu_to_le16(fs_high2lowuid(uid));
1355                raw_inode->i_gid_low = cpu_to_le16(fs_high2lowgid(gid));
1356                raw_inode->i_uid_high = 0;
1357                raw_inode->i_gid_high = 0;
1358        }
1359        raw_inode->i_links_count = cpu_to_le16(inode->i_nlink);
1360        raw_inode->i_size = cpu_to_le32(inode->i_size);
1361        raw_inode->i_atime = cpu_to_le32(inode->i_atime.tv_sec);
1362        raw_inode->i_ctime = cpu_to_le32(inode->i_ctime.tv_sec);
1363        raw_inode->i_mtime = cpu_to_le32(inode->i_mtime.tv_sec);
1364
1365        raw_inode->i_blocks = cpu_to_le32(inode->i_blocks);
1366        raw_inode->i_dtime = cpu_to_le32(ei->i_dtime);
1367        raw_inode->i_flags = cpu_to_le32(ei->i_flags);
1368        raw_inode->i_faddr = cpu_to_le32(ei->i_faddr);
1369        raw_inode->i_frag = ei->i_frag_no;
1370        raw_inode->i_fsize = ei->i_frag_size;
1371        raw_inode->i_file_acl = cpu_to_le32(ei->i_file_acl);
1372        if (!S_ISREG(inode->i_mode))
1373                raw_inode->i_dir_acl = cpu_to_le32(ei->i_dir_acl);
1374        else {
1375                raw_inode->i_size_high = cpu_to_le32(inode->i_size >>
32);
1376                if (inode->i_size > 0x7fffffffULL) {
1377                        if (!EXT2_HAS_RO_COMPAT_FEATURE(sb,
1378
EXT2_FEATURE_RO_COMPAT_LARGE_FILE) ||
1379                                EXT2_SB(sb)->s_es->s_rev_level ==
1380                                        cpu_to_le32(EXT2_GOOD_OLD_REV))
{
1381                                /* If this is the first large file
1382                                 * created, add a flag to the superblock.
1383                                 */
1384                                lock_kernel();
1385                                ext2_update_dynamic_rev(sb);
1386                                EXT2_SET_RO_COMPAT_FEATURE(sb,
1387
EXT2_FEATURE_RO_COMPAT_LARGE_FILE);
1388                                unlock_kernel();
1389                                ext2_write_super(sb);
1390                        }
1391                }
1392        }
1393
1394        raw_inode->i_generation = cpu_to_le32(inode->i_generation);
```

Si el inodo es de un tipo especial, que solo contiene una serie de nodos específicos, se copian solamente esos.

```
1395        if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode)) {
1396                if (old_valid_dev(inode->i_rdev)) {
1397                        raw_inode->i_block[0] =
```

```
1398                                    cpu_to_le32(old_encode_dev(inode-
>i_rdev));
1399                            raw_inode->i_block[1] = 0;
1400                    } else {
1401                            raw_inode->i_block[0] = 0;
1402                            raw_inode->i_block[1] =
1403                                    cpu_to_le32(new_encode_dev(inode-
>i_rdev));
1404                            raw_inode->i_block[2] = 0;
1405                    }
```

En todos los demás casos, se copian los bloques que hayan definidos.

```
1406            } else for (n = 0; n < EXT2_N_BLOCKS; n++)
1407                    raw_inode->i_block[n] = ei->i_data[n];
1408            mark_buffer_dirty(bh);
```

Si la sincronización está activa, esperamos a que se copie el inodo en disco mediante sync_dirty_buffer (espera a que todos los procesos I/O acaben para comenzar un nuevo proceso I/O).

```
1409            if (do_sync) {
1410                    sync_dirty_buffer(bh);
1411                    if (buffer_req(bh) && !buffer_uptodate(bh)) {
1412                            printk ("IO error syncing ext2 inode
[%s:%08lx]\n",
1413                                    sb->s_id, (unsigned long) ino);
1414                            err = -EIO;
1415                    }
1416            }
1417            ei->i_state &= ~EXT2_STATE_NEW;
1418            brelse (bh);
1419            return err;
1420}
```

Función ext2_write_inode. Fichero: inode.c

Llama a ext2_update_inode para hacer la escritura del inodo en el disco. La reescritura además no se hace de forma inmediata.

```
1422int ext2_write_inode(struct inode *inode, int wait)
1423{
1424            return ext2_update_inode(inode, wait);
1425}
```

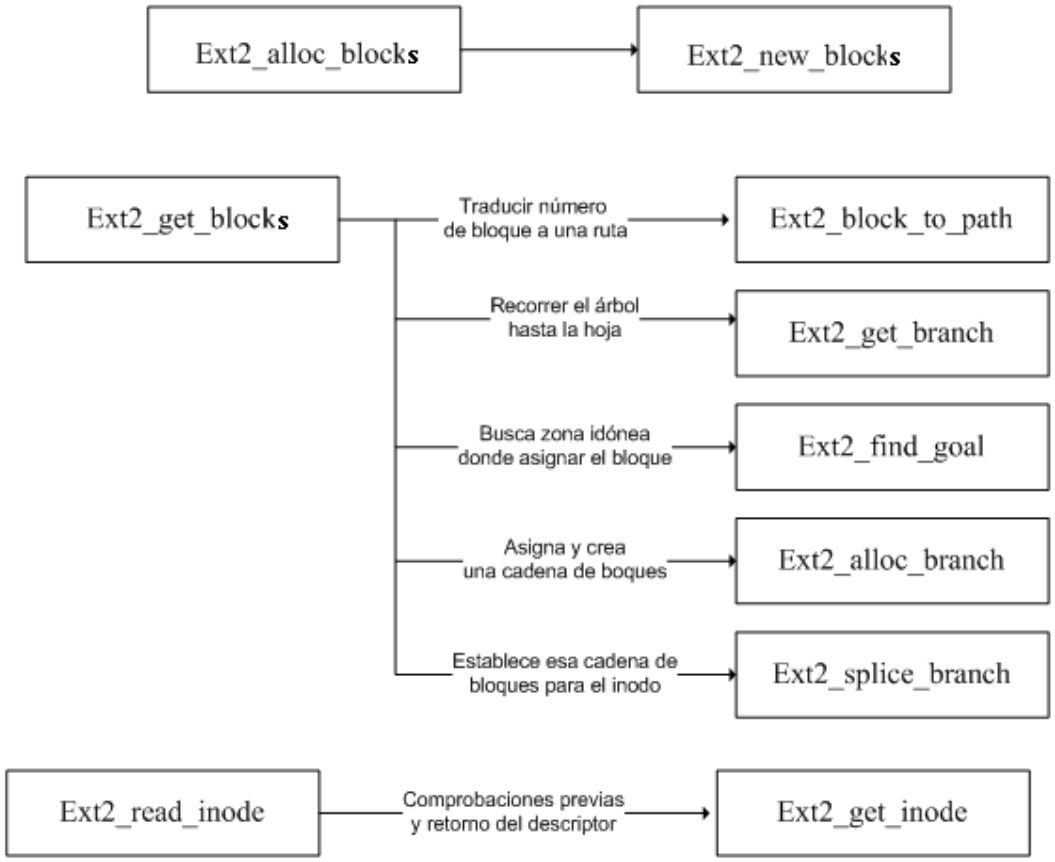
Función ext2_sync_inode. Fichero: inode.c

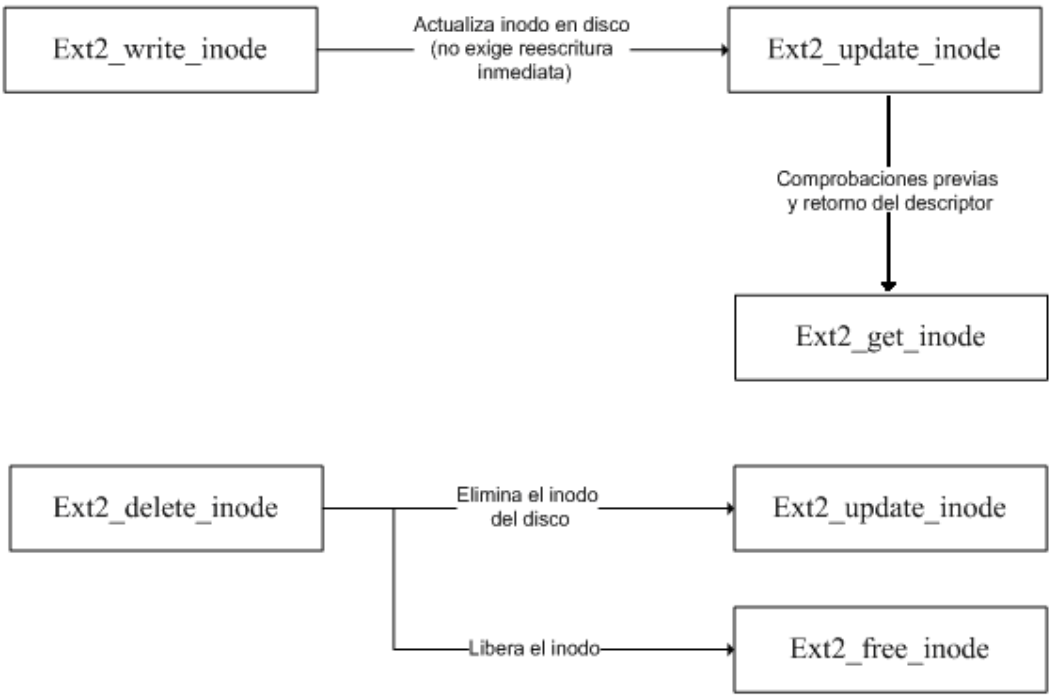
La función ext2_sync_inode llama a la función del VFS sync_inode (se encuentra en fs/fs-writeteback.c.). Produce una reescritura inmediata.

```
1427int ext2_sync_inode(struct inode *inode)
1428{
1429            struct writeback_control wbc = {
1430                    .sync_mode = WB_SYNC_ALL,
1431                    .nr_to_write = 0,       /* sys_fsync did this */
```

```
1432        };
1433        return sync_inode(inode, &wbc);
1434}
```

- Dependencia de funciones





18.4 Glosario

Superbloque.- Esta estructura es la que, entre otras cosas, identifica al sistema de ficheros tipo UNIX como tal, a través de un número mágico o similar. Asimismo contiene información de detalle sobre la disposición y tamaño del resto de las partes (campos) del sistema de ficheros. Sin el superbloque, el sistema de ficheros no puede ser reconocido como tal, ni puede ser manejado convenientemente. Por esta razón es común que existan varías copias del mismo dispersas por la partición en puntos preestablecidos. Su tamaño es, como su nombre indica, un bloque, que en su mayor parte estará vacío.

Mapa de bits de bloques.- Esta estructura de datos es tratada como un vector de bits donde cada bit informa sobre el estado de ocupación (libre u ocupado) de cada bloque del dispositivo. Realmente sólo tendría que informar sobre los bloque que componen el campo “Bloques de datos” y no sobre todo el dispositivo, pero esta optimización no suele realizarse.

Precisa un bit por bloque, luego su tamaño dependerá del número total de bloques del dispositivo, o lo que es lo mismo, del tamaño del dispositivo y del tamaño del bloque. Finalmente su tamaño se redondeará por exceso a un número entero de bloques.

Mapa de bits de inodos.- Al igual que la estructura anterior, se trata de un vector de bits, donde cada bit informa sobre el estado de ocupación (libre u ocupado) de cada inodo del sistema de ficheros.

Precisa un bit por inodo, luego su tamaño dependerá del número total de inodos del sistema de ficheros (véase el apartado siguiente). Finalmente su tamaño se redondeará por exceso a un número entero de bloques.

Inodos.- Este campo equivale a un vector de inodos, es decir, contiene todos los inodos del sistema de ficheros uno detrás de otro.

Cada inodo contiene toda la información relativa a un objeto (fichero, directorio, fichero especial, etc.) del sistema de ficheros, exceptuando su nombre. En general en el inodo se pueden distinguir dos partes, atributos y localización.

Atributos.- Son todos los campos que caracterizan al objeto en cuestión (tipo de objeto, permisos, propietario, grupo, tamaño, número de enlaces, fechas de creación, modificación y acceso).

Localización.- Son las direcciones de los bloques ocupados por el fichero (objeto en general) en el sistema de ficheros. Es una tabla indexada, cuyo nivel de indexación aumenta con el tamaño del objeto. Dispone de una serie de direcciones directas (alrededor de 10) y de otras tres entradas de nivel de indexación creciente (simple, doble y triple indirecta) que apuntan a otros bloques de datos (que denominamos bloques indirectos) usados para contener direcciones de bloque.

Fichero regular:- fichero que contiene información habitual, datos o programas.

Bibliografía

- The Second Extended File System: Internal Layout. Dave Poirier
- Understanding Linux Kernel. 3erd edition. O'Reilly
- The Linux Kernel Book. Rémy Card
- http://lxr.linux.no/linux+v2.6.28
- http://es.wikipedia.org/wiki/Sistema_de_archivos_virtual
- http://en.wikipedia.org/wiki/Inode