# Inodo en sistemas de ficheros Ext2

**David Aguiar González**

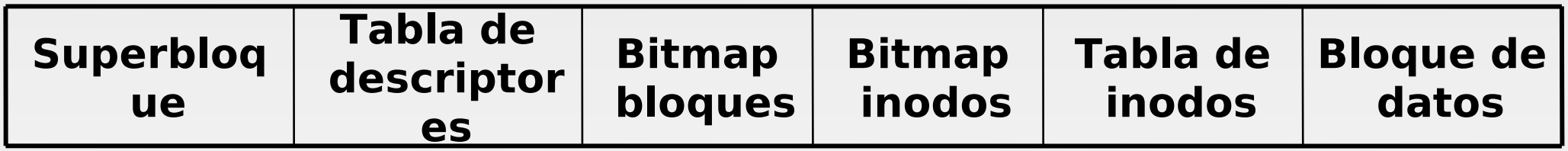
# CONTENIDO

- Estructura física Ext2
- Inodo:
  - ¿Qué es?
  - Estructuración
- Funciones
- ext2 Vs ext3 Vs ext4
- Bibliografía

# ESTRUCTURA FÍSICA EXT2 (I)

Todo dispositivo con ext2 estará compuesto por:

- ***Sector de boot***: código máquina necesario para cargar el núcleo en el arranque del sistema.
- ***Grupos de bloques*** de datos.

# ESTRUCTURA FÍSICA EXT2 (II)

Cada grupo de bloques se compone:

- **1 bloque superbloque**
  - Información de control de sistema de ficheros
  - Se copia en todos los grupos
- **N bloques tabla de descriptores**
  - Direcciones de bloques importantes
  - Se copia en todos los grupos
- **1 bloque bitmap bloques**
  - Vector de bits. 0=libre, 1=ocupado
- **1 bloque bitmap inodos**
  - Vector de bits. 0=libre, 1=ocupado
- **1 bloque tabla inodos**
  - Dividida entre todos los grupos de bloque
- **N bloques de datos**
  - Para almacenar los datos de archivos y directorios.

| Superbloque | Tabla de descriptores | Bitmap bloques | Bitmap inodos | Tabla de inodos | Bloque de datos |
|---|---|---|---|---|---|

# Inodo en sistemas de ficheros Ext2 - EL INODO

- **¿Qué es un INODO?**
  Una estructura. Única para cada objeto.

- **¿Cuál es su misión?**

Guardar información de archivos regulares.

- **¿Qué tipo de información?**

  - *Atributos*: permisos, propietario, grupo, tamaño, fechas....

Menos el NOMBRE.

  - *Localización*: direcciones de los bloques ocupados por el fichero.

# ESTRUCTURACIÓN

Cada sistema de ficheros tiene su propia estructura

**ext2**
**ext3**
**ext4**
**msdos**
**vfat**
**NTFS**
**iso9669**

(VFS también)

# struct ext2_inode (I)

## Fichero <include/linux/ext2_fs.h>

**i_mode** -> Formato del fichero y permisos de acceso

| | | |
|---|---|---|
| EXT2_S_IFSOCK | 0xC000 | socket |
| EXT2_S_IFLNK | 0xA000 | symbolic link |
| EXT2_S_IFREG | 0x8000 | regular file |
| ... | | |
| EXT2_S_IRUSR | 0x0100 | user read |
| EXT2_S_IWUSR | 0x0080 | user write |
| EXT2_S_IXUSR | 0x0040 | user execute |
| EXT2_S_IRGRP | 0x0020 | group read |
| EXT2_S_IWGRP | 0x0010 | group write |
| ... | | |

**i_uid** -> Identificador del propietario

**i_size** -> Tamaño en bytes del fichero asociado

**i_gid** -> Identificador de grupo

**i_links_count** -> Número de enlaces que apuntan al inodo

**i_blocks** -> Nº bloques asignados para el inodo

# struct ext2_inode (II)

| | | |
|---|---|---|
| **i_atime** | -> | **Fecha del último acceso al archivo** |
| **i_ctime** | -> | **Fecha de creación del inodo** |
| **i_mtime** | -> | **Fecha de última modificación del archivo** |
| **i_dtime** | -> | **Fecha de supresión del archivo** |

Aunque en realidad son los segundos que han pasado desde 01/01/1970

| | | |
|---|---|---|
| **i_flags** | -> | **Comportamiento al acceder al inodo** |
| **osd1** | -> | **Depende del sistema operativo** |
| **i_generation** | -> | **Número de versión asociado al inodo** |
| **i_file_acl** | -> | **Dirección del descriptor de la lista de control de acceso asociada al archivo** |
| **i_dir_acl** | -> | **Dirección del descriptor de la lista de control de acceso asociada a un directorio** |

# struct ext2_inode (III)

**i_block[]** -> **Direcciones de bloques de datos asociados al inodo**

```
#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)
```

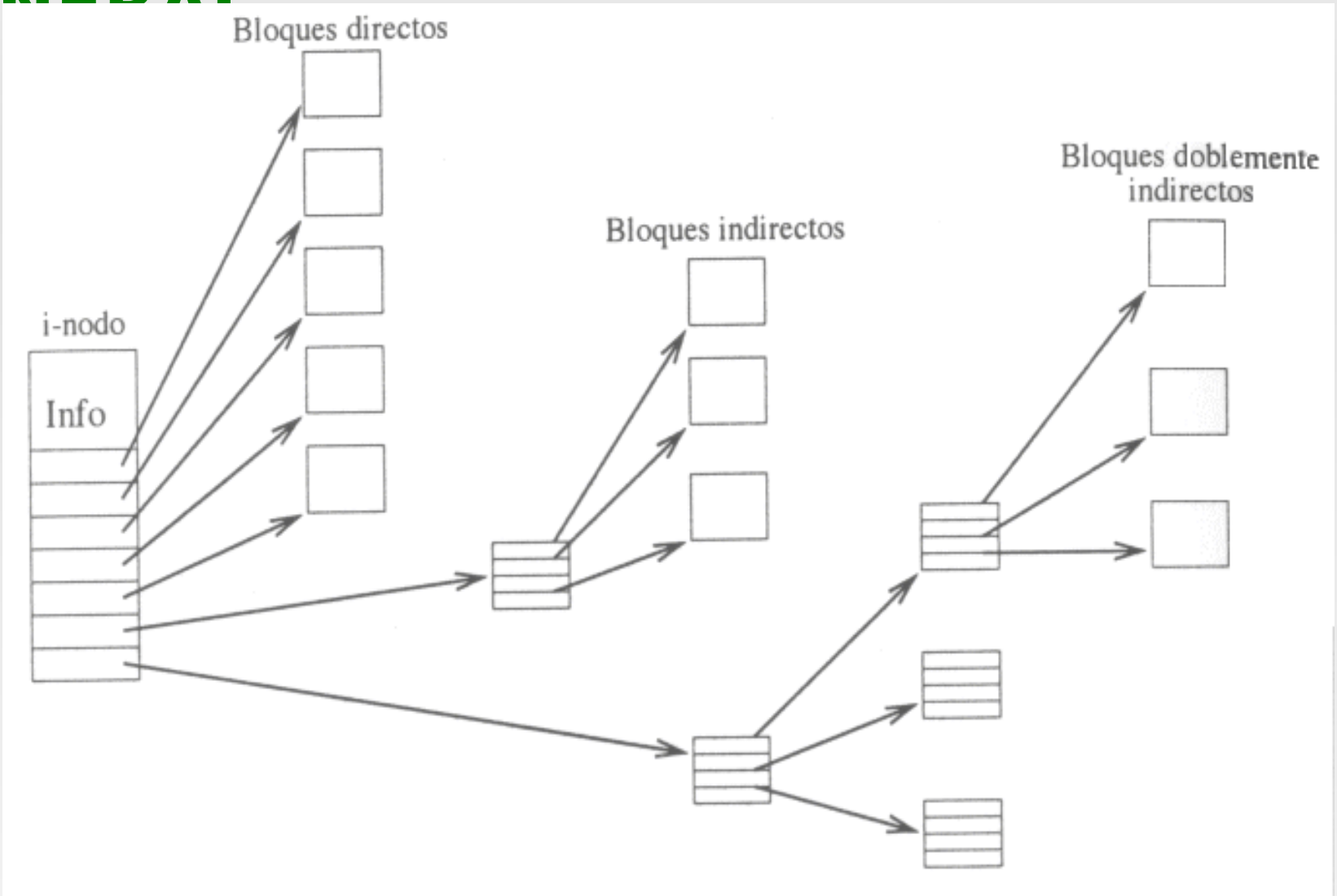
**i_faddr** -> **Dirección del fragmento del archivo**

**osd2** -> **Depende del sistema operativo. En nuestro caso (linux), contiene una estructura cuyos campos son el número de fragmentos y el**

# Inodo en sistemas de ficheros Ext2 - EL INODO – ESTRUCTURACIÓN

## struct ext2_inode (IV) Figura Aclaratoria

**INODO GENERAL**



# struct ext2_inode (V) Caso especial

## INODO DIRECTORIO

Tienen una estructura diferente => **ext2_dir_entry.**

Es una tabla donde cada entrada es una estructura que asocia inodos con archivos.

UNA por cada archivo.

```
inode -> Número del inodo del archivo
rec_len  -> Tamaño en bytes de la entrada del directorio
name_len -> Nº de caracteres del nombre del archivo
name     -> Nombre del archivo
```

# struct ext2_inode (VI) Figura Aclaratoria

## INODO DIRECTORIO

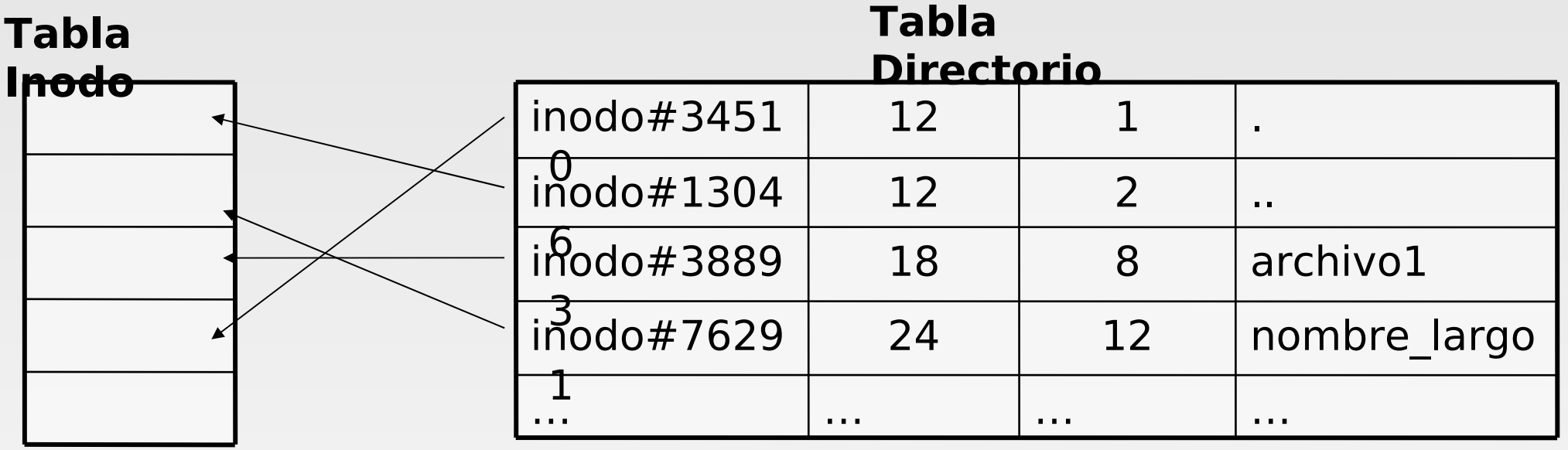
```
$ ls –a

.

..

archivo1

nombre_largo
```



# Campo de operaciones i_op

En la estructura inode, se define un campo llamado i_op, que contendrá un puntero a todas las operaciones posibles que se le pueden aplicar a un inodo:

```
int (*create) (struct inode *,struct dentry *,int,
struct nameidata *);

struct dentry * (*lookup) (struct inode *,struct dentry
*, struct nameidata *);

int (*link) (struct dentry *,struct inode *,struct
dentry *);

int (*unlink) (struct inode *,struct dentry *);

…
```

# FUNCIONES (I)

## EXT2_NEW_INODE <linux/fs/ext2/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.**

```
struct inode *ext2_new_inode(struct inode *dir, int mode)
{
        ...
        sb = dir->i_sb;

        inode = new_inode(sb);
        if (!inode)
                return ERR_PTR(-ENOMEM);
        ...
        if (S_ISDIR(mode)) {
                if (test_opt(sb, OLDALLOC))
                        group = find_group_dir(sb, dir);
                else
                        group = find_group_orlov(sb, dir);
        } else
                group = find_group_other(sb, dir);
```

**Obtiene la estructura superbloque donde se encuentra el inodo padre**

**Crea un nuevo inode en VFS y inicializa el valor i_sb**

**Heurísticas que dependen del tipo de fichero (directorio o fichero regular)**

# FUNCIONES (II)

## EXT2_NEW_INODE <linux/fs/ext2/ialloc.c>

```
for (i = 0; i < sbi->s_groups_count; i++) {
      gdp = ext2_get_group_desc(sb, group, &bh2);
      ...

      bitmap_bh = read_inode_bitmap(sb, group);
      ...
      ino = ext2_find_next_zero_bit((unsigned long bitmap_bh->b_data,
                                    EXT2_INODES_PER_GROUP(sb), ino);

...

percpu_counter_mod(&sbi->s_freeinodes_counter, -1);
...
inode->i_mode = mode;
inode->i_ino = ino;
inode->i_blocks = 0;
inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME_SEC;
...
mark_inode_dirty(inode);

ext2_preread_inode(inode);
return inode;
```

Comenzamos la búsqueda en el grupo seleccionado para almacenar el inode

Buscamos en el bitmap de inodes alguno a 0

Decrementamos el número de inodos libres totales

Vamos rellenando los campos del inodo

Marcamos el inodo como modificado

Hacemos una prelectura del inodo, ya que seguramente después habrá que crear

# FUNCIONES (III)

## EXT2_IGET <linux/fs/ext2/inode.c>

**Copia un inodo desde disco a otro en memoria.**

```
struct inode *ext2_iget (struct super_block *sb, unsigned long ino)
{
        ...
        inode = iget_locked(sb, ino);
        if (!inode)
                return ERR_PTR(-ENOMEM);
        if (!(inode->i_state & I_NEW))
                return inode;
        ...
        raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
        ...
        inode->i_mode = le16_to_cpu(raw_inode->i_mode);
        inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
        inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
        ...
        inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
        inode->i_size = le32_to_cpu(raw_inode->i_size);
        inode->i_atime.tv_sec = (signed)le32_to_cpu(raw_inode->i_atime);
```

**Obtenemos el inodo de disco.**
**Si no se pudo alojar en memoria, se devuelve un error y si es nuevo, simplemente se devuelve.**

**Obtenemos el inodo en la estructura ext2**

**Copiamos sus campos**

# FUNCIONES (IV)

## EXT2_IGET <linux/fs/ext2/inode.c>

```
if (S_ISREG(inode->i_mode)) {
      inode->i_op = &ext2_file_inode_operations;
...
} else if (S_ISDIR(inode->i_mode)) {
      inode->i_op = &ext2_dir_inode_operations;
...
} else if (S_ISLNK(inode->i_mode)) {
      if (ext2_inode_is_fast_symlink(inode)) {
            inode->i_op = &ext2_fast_symlink_inode_operations;
...

return inode;
```

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

**Devolvemos el inodo**

# FUNCIONES (V)

## EXT2_UPDATE_INODE <linux/fs/ext2/inode.c>

**Actualizar o modificar un inodo en disco.**

```
static int ext2_update_inode(struct inode * inode, int do_sync)
{
        ...
        struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
        ...
        raw_inode->i_mode = cpu_to_le16(inode->i_mode);
        ...
        raw_inode->i_links_count = cpu_to_le16(inode->i_nlink);
        raw_inode->i_size = cpu_to_le32(inode->i_size);
        raw_inode->i_atime = cpu_to_le32(inode->i_atime.tv_sec);
        raw_inode->i_ctime = cpu_to_le32(inode->i_ctime.tv_sec);
        raw_inode->i_mtime = cpu_to_le32(inode->i_mtime.tv_sec);
        raw_inode->i_blocks = cpu_to_le32(inode->i_blocks);
        raw_inode->i_dtime = cpu_to_le32(ei->i_dtime);
        raw_inode->i_flags = cpu_to_le32(ei->i_flags);
        raw_inode->i_faddr = cpu_to_le32(ei->i_faddr);
        raw_inode->i_frag = ei->i_frag_no;
        raw_inode->i_fsize = ei->i_frag_size;
```

**Obtenemos el inodo de disco**

**Copiamos cada uno de los campos del inodo a la estructura ext2_inode**

# FUNCIONES (VI)

## EXT2_UPDATE_INODE <linux/fs/ext2/inode.c>

```
        ...
        mark_buffer_dirty(bh);
        if (do_sync) {
                sync_dirty_buffer(bh);
        ...
}
```

**Marcamos el buffer como modificado y si la sincronización está activa, lo escribimos en disco**

# FUNCIONES (VII)

## EXT2_DELETE_INODE <linux/fs/ext2/inode.c>

**Elimina un inodo y limpia toda la información asociada a este.**

```
void ext2_delete_inode (struct inode * inode)
{
      truncate_inode_pages(&inode->i_data, 0);
      if (is_bad_inode(inode))
            goto no_delete;
      EXT2_I(inode)->i_dtime  = get_seconds();
      mark_inode_dirty(inode);
      ext2_update_inode(inode, inode_needs_sync(inode));

      inode->i_size = 0;
      if (inode->i_blocks)
            ext2_truncate (inode);
      ext2_free_inode (inode);

      return;
no_delete:
      clear_inode(inode);
}
```

Vacía las páginas de datos asociados

Inodo inválido por estar corrupto

Liberamos el inodo

El inodo no será útil

# FUNCIONES (VIII)

## EXT2_FREE_INODE <linux/fs/ext2/ialloc.c>

**Libera un inodo.**

```
void ext2_free_inode (struct inode * inode)
{
      ...
      if (!is_bad_inode(inode)) {
            ext2_xattr_delete_inode(inode);
      ...
      clear_inode (inode);
      ...
      bitmap_bh = read_inode_bitmap(sb, block_group);
      ...
      if (!ext2_clear_bit_atomic(sb_bgl_lock(EXT2_SB(sb), block_group),
                                       bit, (void *) bitmap_bh->b_data))
            ext2_error (sb, "ext2_free_inode",
                  "bit already cleared for inode %lu", ino);
      else
            ext2_release_inode(sb, block_group, is_directory);
      mark_buffer_dirty(bitmap_bh);
   ...
}
```

**Borramos toda su información**

**Inodo no útil**

**Leemos el bitmap de inodos y lo restablecemos a libre (0)**

# FUNCIONES (IX)

## EXT2_FREE_BLOCKS <linux/fs/ext2/balloc.c>

**Elimina uno o varios bloques. Se le pasa el inodo, el bloque y el número de bloques a borrar.**

```
void ext2_free_blocks (struct inode * inode, unsigned long block,
                                                unsigned long count)
{
        ...
        block_group = (block - le32_to_cpu(es->s_first_data_block)) /
                        EXT2_BLOCKS_PER_GROUP(sb);
        bit = (block - le32_to_cpu(es->s_first_data_block)) %
                        EXT2_BLOCKS_PER_GROUP(sb);
        ...
        bitmap_bh = read_block_bitmap(sb, block_group);
        if (!bitmap_bh)
                goto error_return;
        ...
```

**Obtenemos el número del grupo de bloques y el bit**

# FUNCIONES (X)

## EXT2_FREE_BLOCKS <linux/fs/ext2/balloc.c>

```
for (i = 0, group_freed = 0; i < count; i++) {
      if (!ext2_clear_bit_atomic(sb_bgl_lock(sbi, block_group),
                                       bit + i, bitmap_bh->b_data)) {
            ext2_error(sb, __func__,
                  "bit already cleared for block %lu", block + i);
      } else {
            group_freed++;
      }
}

mark_buffer_dirty(bitmap_bh);
if (sb->s_flags & MS_SYNCHRONOUS)
      sync_dirty_buffer(bitmap_bh);

group_adjust_blocks(sb, block_group, desc, bh2, group_freed);
...
}
```

**Comprobamos que no estén ya libres**

**Marcamos el buffer como modificado y lo actualizamos si procede**

**Disminuye el valor de bloques ocupados del grupo**

# FUNCIONES (XI)

## EXT2_GET_BLOCKS <linux/fs/ext2/inode.c>

**Buscamos en el árbol hojas donde alojar una serie de bloques.**

```
static int ext2_get_blocks(struct inode *inode,
                           sector_t iblock, unsigned long maxblocks,
                           struct buffer_head *bh_result, int create)
{
        ...
        depth = ext2_block_to_path(inode, iblock, offsets, &blocks_to_boundary);
        ...
        partial = ext2_get_branch(inode, depth, offsets, chain, &err);
        if (!partial) {
                first_block = le32_to_cpu(chain[depth - 1].key);
                clear_buffer_new(bh_result);
                count++;
                ...
                goto got_it;
        ...
        mutex_lock(&ei->truncate_mutex);
        ...
        goal = ext2_find_goal(inode, iblock, partial);
```

**Traducimos el número de bloque a una ruta en el árbol**

**Buscamos un bloque en las ramas**

**No hemos encontrado bloques libres, buscamos el mejor lugar donde alojarlo**

# FUNCIONES (XII)

## EXT2_GET_BLOCKS <linux/fs/ext2/inode.c>

```
...
count = ext2_blks_to_allocate(partial, indirect_blks, maxblocks,
                                                blocks_to_boundary);

err = ext2_alloc_branch(inode, indirect_blks, &count, goal,
                        offsets + (partial - chain), partial);

if (err) {
      mutex_unlock(&ei->truncate_mutex);
      goto cleanup;
}
...
ext2_splice_branch(inode, iblock, partial, indirect_blks, count);
mutex_unlock(&ei->truncate_mutex);
```

**Bloques directos necesarios para alojar en la rama**

**Alojamos e inicializamos el grupo de bloques**

**Conectamos la rama al inodo**

# FUNCIONES (XIII)

## EXT2_GET_BLOCKS <linux/fs/ext2/inode.c>

```
got_it:
      map_bh(bh_result, inode->i_sb, le32_to_cpu(chain[depth-1].key));
      if (count > blocks_to_boundary)
            set_buffer_boundary(bh_result);
      err = count;
      partial = chain + depth - 1;
cleanup:
      while (partial > chain) {
            brelse(partial->bh);
            partial--;
      }
      return err;
changed:
      while (partial > chain) {
            brelse(partial->bh);
            partial--;
            }
      goto reread;
}
```

**Número de bloques alojados**

# FUNCIONES (XIV)

## EXT2_FIND_GOAL <linux/fs/ext2/inode.c>

**Busca el mejor lugar para alojar un bloque. Usa una heurística para buscar el inodo con la mejor posición.**

```
static inline ext2_fsblk_t ext2_find_goal(struct inode *inode, long block,
                                          Indirect *partial)
{
        struct ext2_block_alloc_info *block_i;
        block_i = EXT2_I(inode)->i_block_alloc_info;

        if (block_i && (block == block_i->last_alloc_logical_block + 1)
                    && (block_i->last_alloc_physical_block != 0)) {
                return block_i->last_alloc_physical_block + 1;
        }
        return ext2_find_near(inode, partial);
}
```

**Intenta buscar el bloque consecutivo al último bloque del inode**

# FUNCIONES (XV)

## EXT2_FIND_NEAR <linux/fs/ext2/inode.c>

**Busca un lugar para alojar un grupo de bloques que sea lo más próximo al inodo.**

```
static ext2_fsblk_t ext2_find_near(struct inode *inode, Indirect *ind)
{
        ...
        for (p = ind->p-1; p >= start; p--)
                if (*p)
                        return le32_to_cpu(*p);


        if (ind->bh)
                return ind->bh->b_blocknr;



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

**Buscamos en un bloque anterior**

**Apunta a un bloque indirecto? Lo alojamos cerca del mismo**

**Apunta al inodo mismo. Lo asignamos al cilindro**

# FUNCIONES (XVI)

## EXT2_NEW_BLOCKS <linux/fs/ext2/balloc.c>

**Función básica de asignación de bloques. Usa un bloque objetivo para indicar dónde alojarlo.**

```
ext2_fsblk_t ext2_new_blocks(struct inode *inode, ext2_fsblk_t goal,
                                                  unsigned long *count, int *errp)
{
        ...
        sb = inode->i_sb;
        if (!sb) {
                printk("ext2_new_blocks: nonexistent device");
                return 0;
        }
        ...
        if (!ext2_has_free_blocks(sbi)) {
                *errp = -ENOSPC;
                goto out;
        }
        ...
        if (goal < le32_to_cpu(es->s_first_data_block) ||
                                        goal >= le32_to_cpu(es->s_blocks_count))
                goal = le32_to_cpu(es->s_first_data_block);
        ...
```

**Comprobamos que exista el superbloque**

**Comprueba que el bloque objetivo está sin ocupar**

# FUNCIONES (XVII)

## EXT2_NEW_BLOCKS <linux/fs/ext2/balloc.c>

```
...
gdp = ext2_get_group_desc(sb, group_no, &gdp_bh);
if (!gdp)
        goto io_error;
free_blocks = le16_to_cpu(gdp->bg_free_blocks_count);
```

Obtenemos los bloques libres del grupo de bloques

```
if (my_rsv && (free_blocks < windowsz) && (free_blocks > 0)
                                        && (rsv_is_empty(&my_rsv->rsv_window)))
        my_rsv = NULL;
```

Si después de la asignación, no restasen bloques

# FUNCIONES (XVIII)

## EXT2_NEW_BLOCKS <linux/fs/ext2/balloc.c>

```
if (free_blocks > 0) {
      grp_target_blk = ((goal - le32_to_cpu(es->s_first_data_block)) %
                                                  EXT2_BLOCKS_PER_GROUP(sb));

      bitmap_bh = read_block_bitmap(sb, group_no);
      if (!bitmap_bh)
            goto io_error;

      grp_alloc_blk = ext2_try_to_allocate_with_rsv(sb, group_no,
                                                  bitmap_bh, grp_target_blk,
                                                  my_rsv, &num);


      if (grp_alloc_blk >= 0)
            goto allocated;
}
```

**Se alojan los bloques en el árbol rojo-negro y si se consiguió, saltamos**

# FUNCIONES (XIX)

## EXT2_NEW_BLOCKS <linux/fs/ext2/balloc.c>

```
allocated:
        ret_block = grp_alloc_blk + ext2_group_first_block_no(sb, group_no);

        ...
        group_adjust_blocks(sb, group_no, gdp, gdp_bh, -num);
        percpu_counter_sub(&sbi->s_freeblocks_counter, num);

        mark_buffer_dirty(bitmap_bh);
        if (sb->s_flags & MS_SYNCHRONOUS)
                sync_dirty_buffer(bitmap_bh);
        ...
        return ret_block;

out:
        if (!performed_allocation)
                DQUOT_FREE_BLOCK(inode, *count);
        brelse(bitmap_bh);
        return 0;
}
```

**Bloques alojados**

**Disminuye el número de bloques libres del superbloque**

**Si no se pudieron alojar todos los bloques, deshacemos las asignaciones que teníamos hechas**

# ext2 Vs ext3 Vs ext4

## ext3 agrega a ext2...

- Registro por diario
- Índices en árbol para directorios que ocupan múltiples bloques
- Crecimiento en línea

## ext4 agrega a ext3...

- Soporte de volúmenes de hasta 1024PB
- Soporte añadido de extent
- Menor uso de CPU
- Mejoras en la velocidad de lectura y escritura

**Capacidad de reservar un área contigua para un archivo. Puede reducir y hasta eliminar completamente la fragmentación de archivos**

# BIBLIOGRAFÍA


- **The Second Extended File System: Internal Layout. Dave Poirier**
- **Understanding Linux Kernel. 3erd edition. O'Reilly**
- **http://lxr.linux.no/linux+v2.6.28**
- **http://es.wikipedia.org/wiki/Sistema_de_archivos_virtual**
- **http://en.wikipedia.org/wiki/Inode**
- **http://es.wikipedia.org/wiki/Ext2**
- **http://es.wikipedia.org/wiki/Ext3**
- **http://es.wikipedia.org/wiki/Ext4**