(Último punto de Tema 4. GESTION DE ARCHIVOS)

4. Implementación de sistemas de archivos en Linux

4. 1 El Sistema de Archivos VFS (Virtual File System)

4. 2 El Sistema de Archivos Ext2 (Second Extended Filesystem)

4. 3 El Sistema de Archivos Ext3 (Third Extended Filesystem)

Bibliografía:

- R. Love, *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
- W. Mauerer, *Professional Linux Kernel Architecture*, Wiley, 2008.

4.1 El Sistema de Archivos VFS (Virtual File System)

Cada sistema de operativo (SO) implementa, al menos, un sistema de archivos (SA) estándar.

Clásicamente, Ext2 y Ext3 son los SA de Linux y aunque se ha comprobado que son tanto adecuados como robustos, es necesario dar soporte a otros SA distintos.

Con ese objetivo, el kernel incluye una capa entre los procesos de usuario (o la biblioteca estándar) y la implementación del SA.

Esta capa se conoce como Sistema de Archivos Virtual (VFS – Virtual File System). (Ver a continuación figura 8-1 de Mauerer)

La interfaz entre los procesos de usuario y la implementación del kernel de VFS la constituyen las llamadas al sistema (son más de 50).

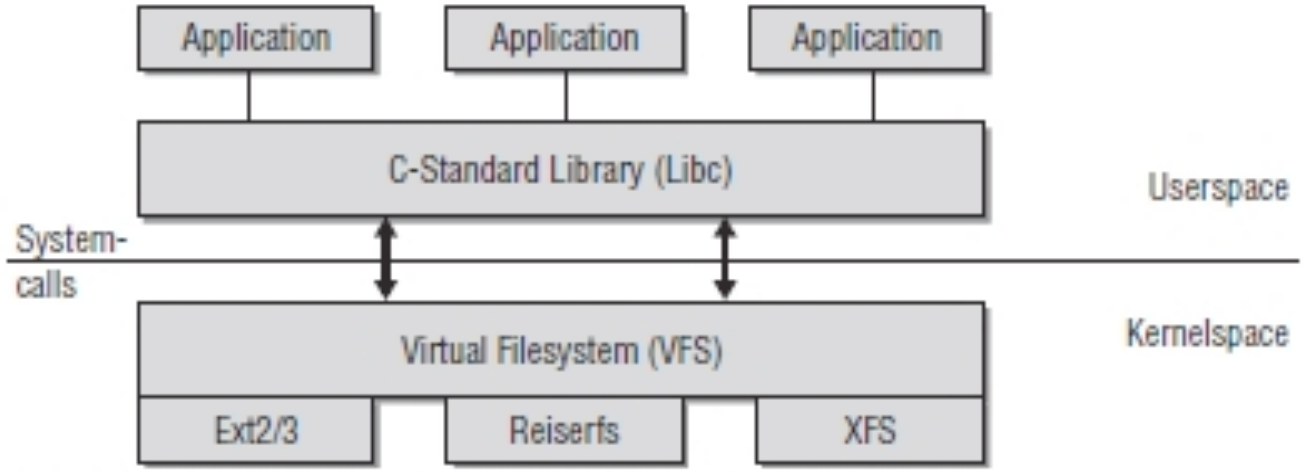


Figure 8-1: VFS layer for filesystem abstraction.

Sobre la complejidad de la tarea de VFS.....

Por una lado, tiene que proporcionar una forma uniforme para gestionar y manipular archivos, directorios y otros objetos.

Por otro, debe llevar a cabo estas funciones para implementaciones distintas de SA.

El kernel soporta más de 40 SA de distintos orígenes
(desde FAT, MS-DOS hasta UFS de Berkeley Unix y iso9660 para CD-ROMs).

(Love, pp. 261) VFS implementa la interfaz que necesitan los programas para trabajar con los distintos SA.

Los programas que usan las llamadas al sistema estándar de Unix pueden leer y escribir sobre diferentes SA, estén o no en diferentes medios físicos.

Todos los SA que permite VFS pueden coexistir y procesarse simultáneamente:

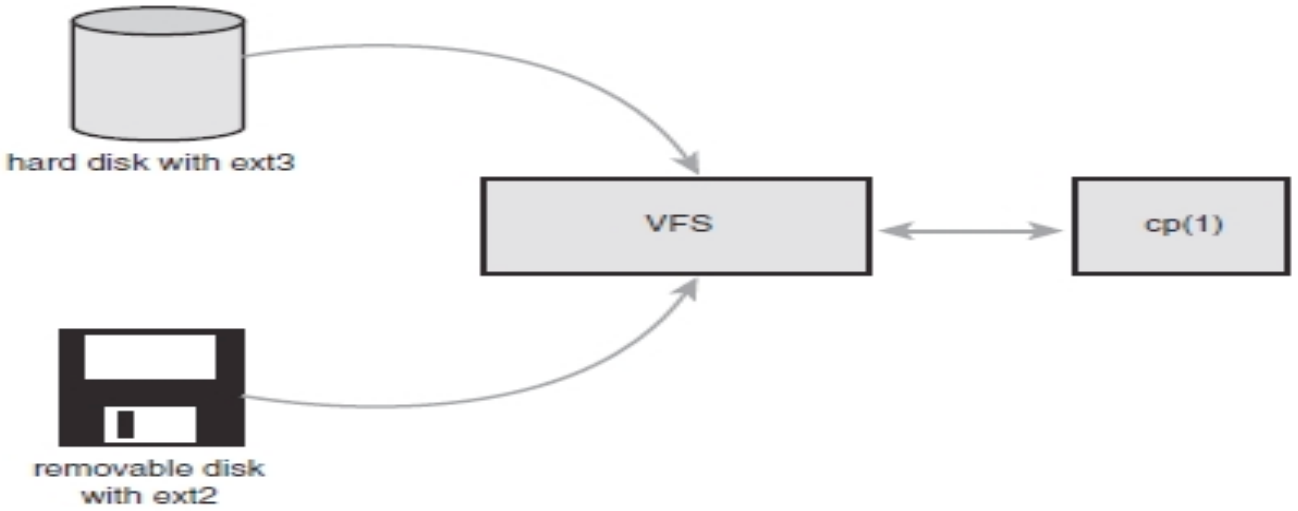


Figure 13.1 The VFS in action: Using the `cp(1)` utility to move data from a hard disk mounted as ext3 to a removable disk mounted as ext2. Two different filesystems, two different media, one VFS.

Linux abstrae el acceso a los archivos y a los SA mediante una interfaz virtual que lo hace posible, en la figura 13.2 de Love se muestra cómo ante una llamada al sistema de escritura (`write`) fluyen los datos por las distintas partes del sistema.

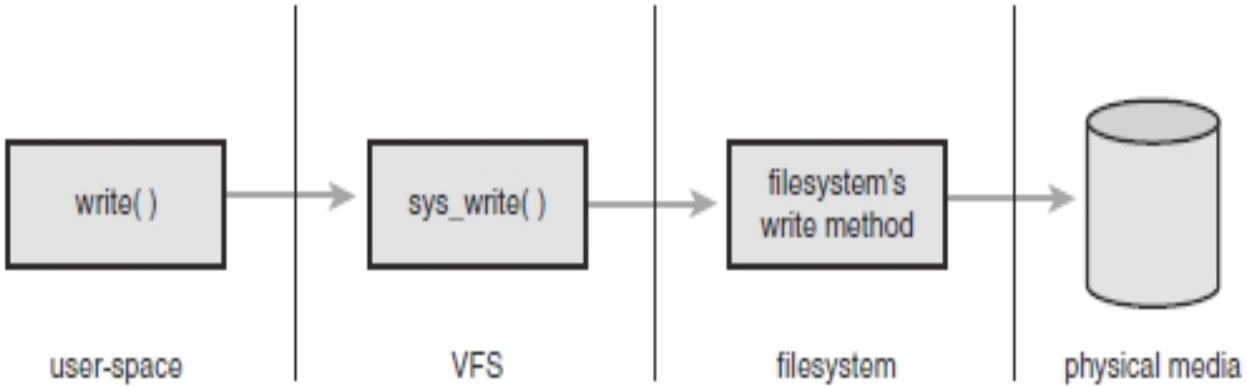


Figure 13.2 The flow of data from user-space issuing a `write()` call, through the VFS's generic system call, into the filesystem's specific write method, and finally arriving at the physical media.

Tipos de SA (Mauerer, pp. 520)

Podemos agrupar los distintos SA en tres clases generales:

1. **SA basados en disco** (Disk-based filesystems): son la forma clásica de almacenar archivos en medios no volátiles. Ejemplos: Ext2/3, Reiserfs, FAT, e iso9660.

2. **SA Virtuales** (Virtual filesystems): son generados por el kernel y constituyen una forma simple para permitir la comunicación entre los programas y los usuarios.

   No requieren espacio de almacenamiento en ningún dispositivo hardware.

   Ejemplo: **/proc**.

   ```
   $ cat /proc/version
   Linux version 2.6.4 (wolfang@shroedinger)  (gcc version 4.2.-...
   ```

   La orden "cat /proc/version" muestra información del procesador del sistema, se extrae de las estructuras de datos que tiene el kernel en memoria principal (MP).

3. **SA de Red** (Network filesystems): los datos están ubicados en un dispositivo harware de otro ordenador.

   Cuando un proceso escribe datos en un archivo, los datos se envian al computador remoto usando un determinado protocolo; el ordenador remoto será el responsable de almacenarlos e informar de que los datos han llegado.

(Mauerer, pp. 521) VFS proporciona una visión uniforme de los objetos en el SA.

No todos los SA soportan las mismas abstracciones (p. ej. los pipes).

Los archivos de dispositivo no pueden almacenarse en SA como FAT porque FAT no proporciona objetos de este tipo.

VFS contempla todos los componentes del SA más poderoso o complejo.

Naturalmente, este modelo existe sólo virtualmente, y debe adaptarse a cada SA usando una variedad de objetos con punteros a funciones.

Cuando trabajamos con archivos, los objetos centrales difieren si estamos en el espacio del kernel o si estamos en el espacio de usuario.

Para un programa de usuario, un archivo se identifica por un descriptor de archivo (nº entero usado como parámetro para identificar el archivo en las operaciones relacionadas con él). El descriptor lo asigna el kernel cuando se abre el archivo y es válido sólo dentro de un proceso. Dos procesos diferentes pueden usar el mismo descriptor pero no apuntan al mismo archivo.

En el espacio del kernel, se mantendrán diversas estructuras de datos para representar el contenido de los archivos, la jerarquía de directorios, información de administración (permisos, usuarios, grupos...) y otros metadatos para gestionar información interna del SA.
Estas estructuras de datos residen en bloques de disco para su almacenamiento secundario, y se leerán en memoria principal (MP) para su procesamiento; habrá normalmente dos versiones de una determinada estructura de datos: una para su almacenamiento persistente en disco, y otra para trabajar con ella en MP.

Estructura de VFS

(Love pp. 265) Existen 4 tipos de **objetos primarios** del VFS:

- **objeto `superblock`**: representa a un SA montado
- **objeto `inode`**: representa a un archivo
- **objeto `dentry`**: representa a una entrada de un directorio que es cada uno de los componentes de un path (cualquier tipo de archivo).
- **objeto `file`**: representa a un archivo abierto y está asociado a un proceso.

Al tratar los directorios como archivos, no existe un objeto especial para representar a un directorio.

Cada uno de estos objetos primarios tiene un objeto "`operations`". Estos objetos describen los métodos que el kernel invoca sobre los objetos primarios.

- **objeto `super_operations`**:
  contiene los métodos que el kernel puede invocar sobre un SA concreto, tal como write_inodo() y sync_fs().
- **objeto `inode_operations`**:
  contiene los métodos que el kernel puede invocar sobre un archivo concreto, tal como crete() y link().
- **objeto `dentry_operations`**:
  contiene los métodos que el kernel puede invocar sobre una entrada de directorio tal como d_compare() y d_delete().
- **objeto `file_operations`**:
  contiene los métodos que un proceso puede invocar sobre un archivo abierto como read() y write().

Los objetos "operations" se implementan como un estructura de punteros a funciones que operan sobre

el objeto padre.

Objeto `superblock`: representa a un SA montado (Mauerer, pp. 552)

NO ENTRA

El kernel crea una lista de instancias de superbloques, una para cada SA montado.

```
<fs.h>
struct super_block {
     struct list_head s_list; /* Keep this first */
     dev_t s_dev; /* search index; _not_ kdev_t */
     unsigned long s_blocksize;
     unsigned char s_blocksize_bits;
     unsigned char s_dirt;
     unsigned long long s_maxbytes; /* Max file size */
     struct file_system_type *s_type;
     struct super_operations *s_op;
     unsigned long s_flags;
     unsigned long s_magic;
     struct dentry *s_root;
     struct xattr_handler **s_xattr;
     struct list_head s_inodes; /* all inodes */
     struct list_head s_dirty; /* dirty inodes */
     struct list_head s_io; /* parked for writeback */
     struct list_head s_more_io; /* parked for more writeback */
     struct list_head s_files;
     struct block_device *s_bdev;
     struct list_head s_instances;
     char s_id[32]; /* Informational name */
     .....}
```

NO ENTRA

Un elemento importante de la **estructura superbloque** es una lista con todos los inodos modificados del SA (el kernel los llama *dirty inodes*).

Los archivos y directorios que han sido modificados se identifican fácilmente mediante esta lista.

Se almacena el tamaño de bloque en Kbytes (`s_blocksize` que toma el valor estandar 1 en Ext2).

`s_root` asocia a este SA con la dentry asociada a su directorio raiz dentro del sistema jerarquico global de archivos.

El kernel comprueba la lista de bloques sucios en intervalos de tiempo periódicos y transfiere los cambios al hardware (por si se cae el sistema).

NO ENTRA

Objeto `inode` (Mauerer pp. 527)

```
<fs.h>
struct inode {
     struct hlist_node i_hash;
     struct list_head i_list;
     struct list_head i_sb_list;
     struct list_head i_dentry;
     unsigned long i_ino;     // n° de inodo; único dentro de su SA
     atomic_t i_count;        // n° de proc usando esta estructura
     unsigned int i_nlink;    // n° de enlaces duros
     uid_t i_uid;             // UID asociado al archivo
     gid_t i_gid;             // GID asociado al archivo
     dev_t i_rdev;            // si el archivo representa un dispositivo
     unsigned long i_version;
     loff_t i_size;               // tamaño en bytes del archivo
     struct timespec i_atime;     // tiempo del último acceso
     struct timespec i_mtime;     // tiempo última mod. del contenido
     struct timespec i_ctime;     // tiempo última mod. del inodo
     unsigned int i_blkbits;
     blkcnt_t i_blocks;           // tamaño en bloques del archivo
     umode_t i_mode;              // tipo de archivo y permisos
     struct inode_operations *i_op;
     const struct file_operations *i_fop;
     struct super_block *i_sb;
     struct address_space *i_mapping;
```

```
        struct address_space i_data;
        struct dquot *i_dquot[MAXQUOTAS];
        struct list_head i_devices;


        union {
                struct pipe_inode_info *i_pipe;    //para archivos tipo pipe
                struct block_device *i_bdev;       //para dispositivos de bloques
                struct cdev *i_cdev;               //para dispositivos de caract.
        };
        int i_cindex;
        __u32 i_generation;
        unsigned long i_state;
        unsigned long dirtied_when; /* jiffies of first dirtying */
        unsigned int i_flags;
        atomic_t i_writecount;
        void *i_security;
};
```

NO ENTRA

Esta estructura inode contiene los datos en memoria que el kernel guarda sobre un inodo que se está procesando, contiene algunos datos más que los datos de un inodo en su soporte de almacenamiento secundario.

Hay SA como FAT que no tienen inodos y, por tanto, se debe generar la información de esta estructura extrayéndola de las usadas por estos SA.

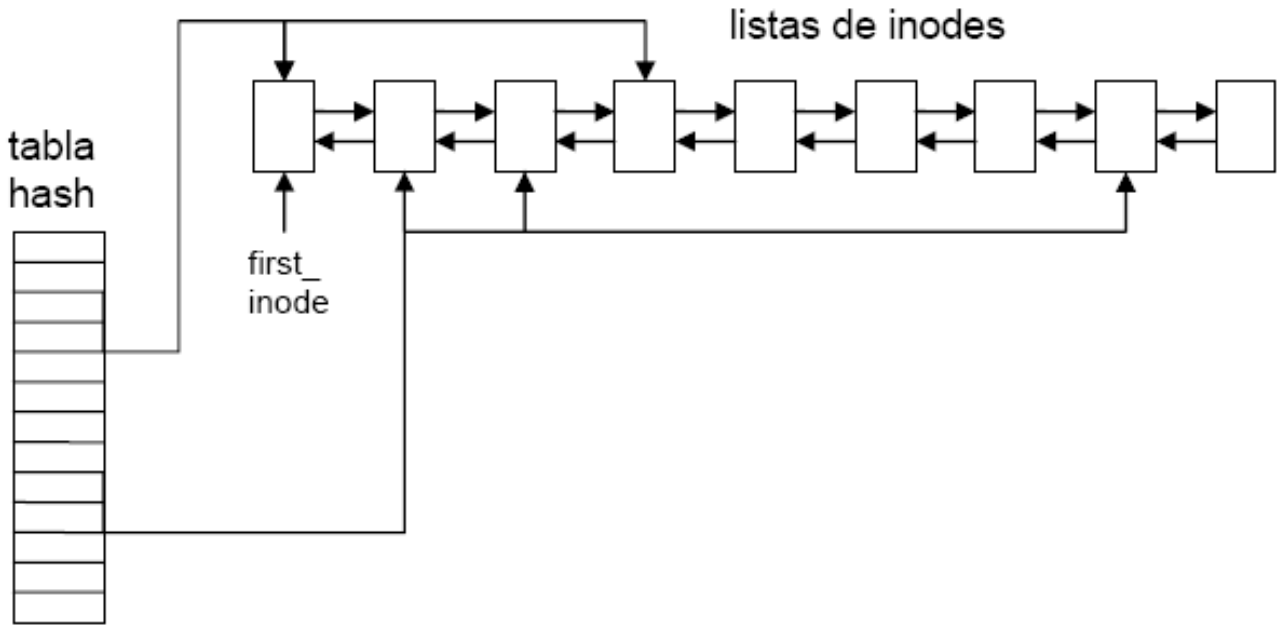
NO ENTRA

Además del tamaño del archivo en bytes, se almacena también el número de bloqes que ocupa el archivo.

Podría calcularse a partir del tamaño del archivo en bytes y del tamaño del bloque del SA donde se ubica el archivo.

Cache de inodos en memoria:

Se denomina **Inode Cache** o **Cache de inodos en memoria** al conjunto de inodos en memoria que gestiona el kernel.
Todos los inodes se combinan en una lista global doblemente enlazada. Además mantiene una tabla hash en que se acccede a las listas con los inodos que tienen el mismo valor de hash. El valor de hash se obtiene con el nº de dispositivo mas el nº de inode.



Si un inodo se necesita y no está en la cache, se lee desde disco a memoria almacenándose en la cache.

Operaciones sobre inodos (Mauerer pp. 529) NO ENTRA

Se define un conjunto de punteros a función para abstraer las operaciones ya que los datos se manipularán por la implementación de un SA específico.

La interfaz es siempre la misma aunque el trabajo es realizado por funciones específicas de la implementación.

La estructura inodo tiene dos punteros (`i_op` y `i_fop`) a matrices que implementan la abstracción anterior.

`i_op` está relacionado con las operaciones sobre el inodo

`i_fop` está relacionado con las las operaciones para trabajar con el archivo.

Todas las operaciones se agrupan juntas en la estructura `inode_operations` siguiente:

<fs.h>

```
struct inode_operations {
      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 *);
      int (*symlink) (struct inode *,struct dentry *,const char *);
      int (*mkdir) (struct inode *,struct dentry *,int);
      int (*rmdir) (struct inode *,struct dentry *);
      int (*mknod) (struct inode *,struct dentry *,int,dev_t);
      int (*rename) (struct inode *, struct dentry *,
      struct inode *, struct dentry *);
      int (*readlink) (struct dentry *, char __user *,int);
      void * (*follow_link) (struct dentry *, struct nameidata *);
      void (*put_link) (struct dentry *, struct nameidata *, void *);
```

```
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
        long (*fallocate)(struct inode *inode, int mode, loff_t offset,
        loff_t len);
}
```

NO ENTRA

NO ENTRA

Objeto `dentry` (Love pp. 275; Mauerer pp. 542)

Un objeto dentry es una estructura que almacena información sobre un archivo y su inodo. Representa a un elemento del arbol de archivos y directorio. Establece la relación entre un archivo y su inodo.

Por ejemplo, cuando se procesa la ruta /home/lopez/prog1 se crea un objeto denty para /, otro para /home, otro para /home/lopez, y otro para /home/lopez/prog1

El conjunto de objetos dentry existentes en un momento dado se denomina **Cache de objetos dentry**

Al contrario de lo que ocurría con los objetos superblock e inode, el objeto dentry no tiene su correspondencia en disco, y por lo tanto no contiene un flag “dirty” que indicara que la copia en disco no esté actualizada.

NO ENTRA

<dcache.h>

```
struct dentry {
     atomic_t d_count; /* usage count */
     unsigned int d_flags; /* dentry flags */
     spinlock_t d_lock; /* per-dentry lock */
     int d_mounted; /* is this a mount point? */
     struct inode *d_inode; /* associated inode */
     struct hlist_node d_hash; /* list of hash table entries */
     struct dentry *d_parent; /* dentry object of parent */
     struct qstr d_name; /* dentry name */
     struct list_head d_lru; /* unused list */
     union {
          struct list_head d_child; /* list of dentries within */
          struct rcu_head d_rcu; /* RCU locking */
     } d_u;
     struct list_head d_subdirs; /* subdirectories */
     struct list_head d_alias; /* list of alias inodes */
     unsigned long d_time; /* revalidate time */
     struct dentry_operations *d_op; /* dentry operations table */
     struct super_block *d_sb; /* superblock of file */
     void *d_fsdata; /* filesystem-specific data */
     unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
};
```

NO ENTRA

Objeto `file` (Love pp. 279)

Representa un archivo abierto por un proceso.

> Cuando se piensa en VFS desde la perspectiva del espacio de usuario, el objeto file juega un papel central, puesto que los procesos tratan directamente con archivos, y no tanto con superbloques, inodos o dentrys.

El objeto file (no el archivo físico) se crea en respuesta a la llamada al sistema open, y se destruye en respuesta a la llamada al sistema close.

Puesto que múltiples procesos pueden abrir y manipular un archivo al mismo tiempo, puede haber múltiples objetos file existiendo al mismo tiempo.

(Love pp. 280) Al igual que el objeto dentry, el objeto file no tiene su correspondencia en disco y por tanto no tiene un flag “dirty” para expresar que deba escribirse en disco.

NO ENTRA

```
struct file {
      struct list_head       f_list;        /* list of file objects */
      struct dentry          *f_dentry;     /* objeto dentry asociado*/
      struct vfsmount        *f_vfsmnt;     /* associated mounted fs */
      struct file_operations *f_op;         /* file operations table */
      atomic_t               f_count;       /* file object's usage count
      unsigned int           f_flags;       /* flags specified on open */
      mode_t                 f_mode;        /* file access mode */
      loff_t                 f_pos;         /* file offset (file pointer
      struct fown_struct     f_owner;       /* owner data for signals */
      unsigned int           f_uid;         /* user's UID */
      unsigned int           f_gid;         /* user's GID */
      int                    f_error;       /* error code */
      struct file_ra_state   f_ra;          /* read-ahead state */
      unsigned long          f_version;     /* version number */
      void                   *f_security;   /* security module */
      void                   *private_data; /* tty driver hook */
      struct list_head       f_ep_links;    /* list of eventpoll links */
      spinlock_t             f_ep_lock;     /* eventpoll lock */
      struct address_space   *f_mapping;    /* page cache mapping */ };
```

NO ENTRA

Estructura `file_system_type`: Cada tipo de SA está representado por una estructura `file_system_type`.

Estructura `vfsmount`: Cada punto de montaje está representado por una estructura `vfsmount` contiene información acerca del punto de montaje, tal como sus localizaciones y flags de montaje.

Información específica de un proceso (Mauerer pp. 532)

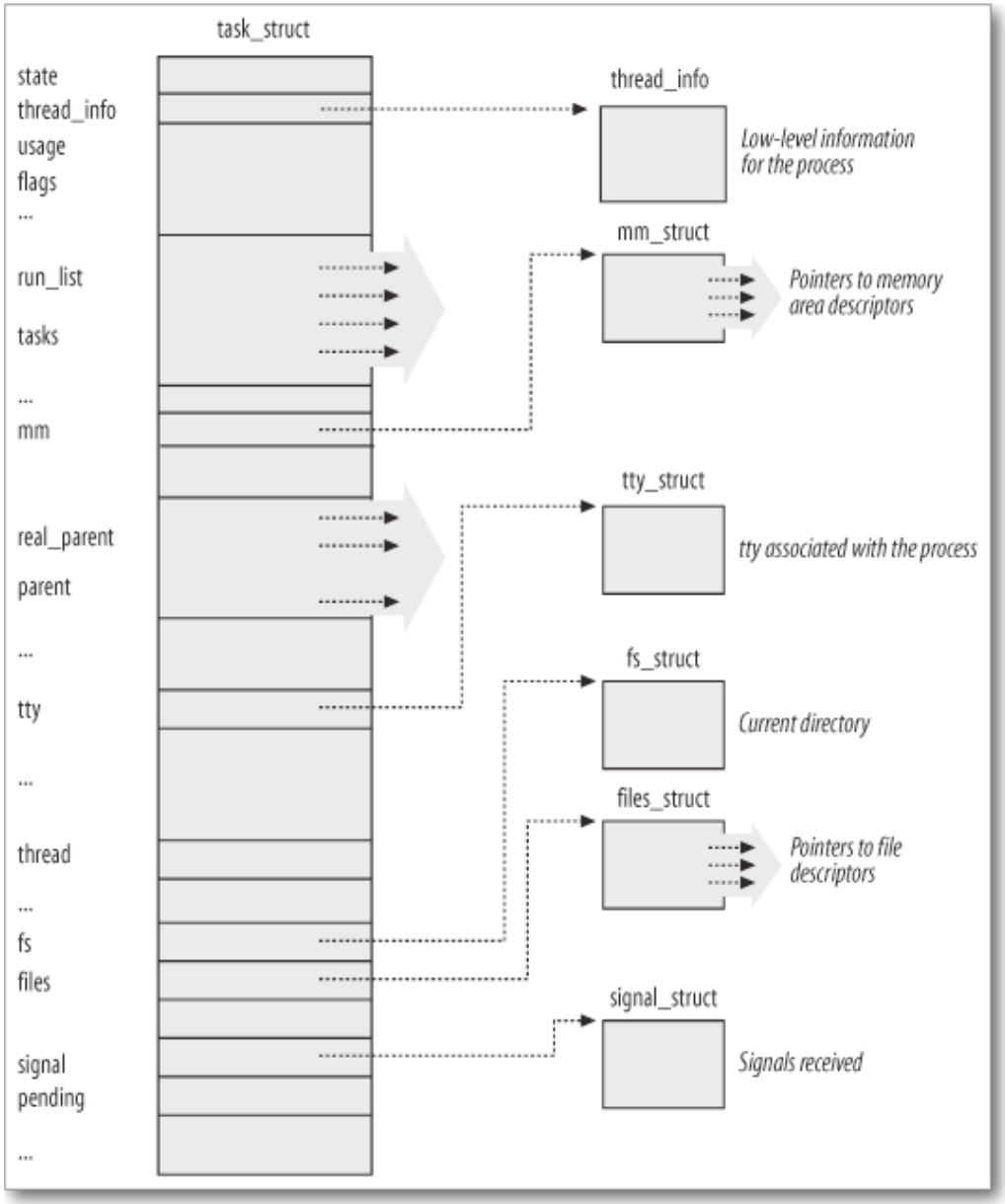
Finalmente, existen dos estructuras por proceso que describen el SA y los archivos asociados con un proceso: `fs_struct` y `file`.

<sched.h>

```
struct task_struct {
......
   /* filesystem information */
   struct fs_struct *fs;

   /* open file information */
   struct files_struct *files;
......

}
```



Estructura files_struct (Love 287)

NO ENTRA

Contiene toda la información específica de un proceso sobre archivos abiertos y descriptores de archivo.

```
 <fs_struct.h>
struct files_struct {
     atomic_t count; /* structure's usage count */
     spinlock_t file_lock; /* lock protecting this structure */
     int max_fds; /* maximum number of file objects */
     int max_fdset; /* maximum number of file descriptors */
     int next_fd; /* next file descriptor number */
     struct file **fd; /* array of all file objects */
     fd_set *close_on_exec; /* file descriptors to close on exec() */
     fd_set *open_fds; /* pointer to open file descriptors */
     fd_set close_on_exec_init; /*initial files to close on exec() */
     fd_set open_fds_init; /* initial set of file descriptors */
     struct file *fd_array[NR_OPEN_DEFAULT]; /* default array
                                                of file objs */
};
```

El array fd apunta a la lista de objetos file abiertos.

Estructura fs_struct (Love pp. 287) (Mauerer pp.540)

NO ENTRA

```
<fs_struct.h>
struct fs_struct {
    atomic_t count;          // nº procesos que apuntan a esta tabla
    rwlock_t lock;
    int umask;               // máscara para la creación de permisos
    struct dentry * root, * pwd, * altroot;
                  // directorio raiz; directorio de trabajo
    struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

Esta estructura almacena información sobre el directorio de trabajo y directorio raiz del proceso en cuestión.

Visión global de las estructuras de datos de VFS (Mauerer pp. 526)

La siguiente figura de Mauerer muestra los distintos componentes de VFS y su interconexión:

NO ENTRA

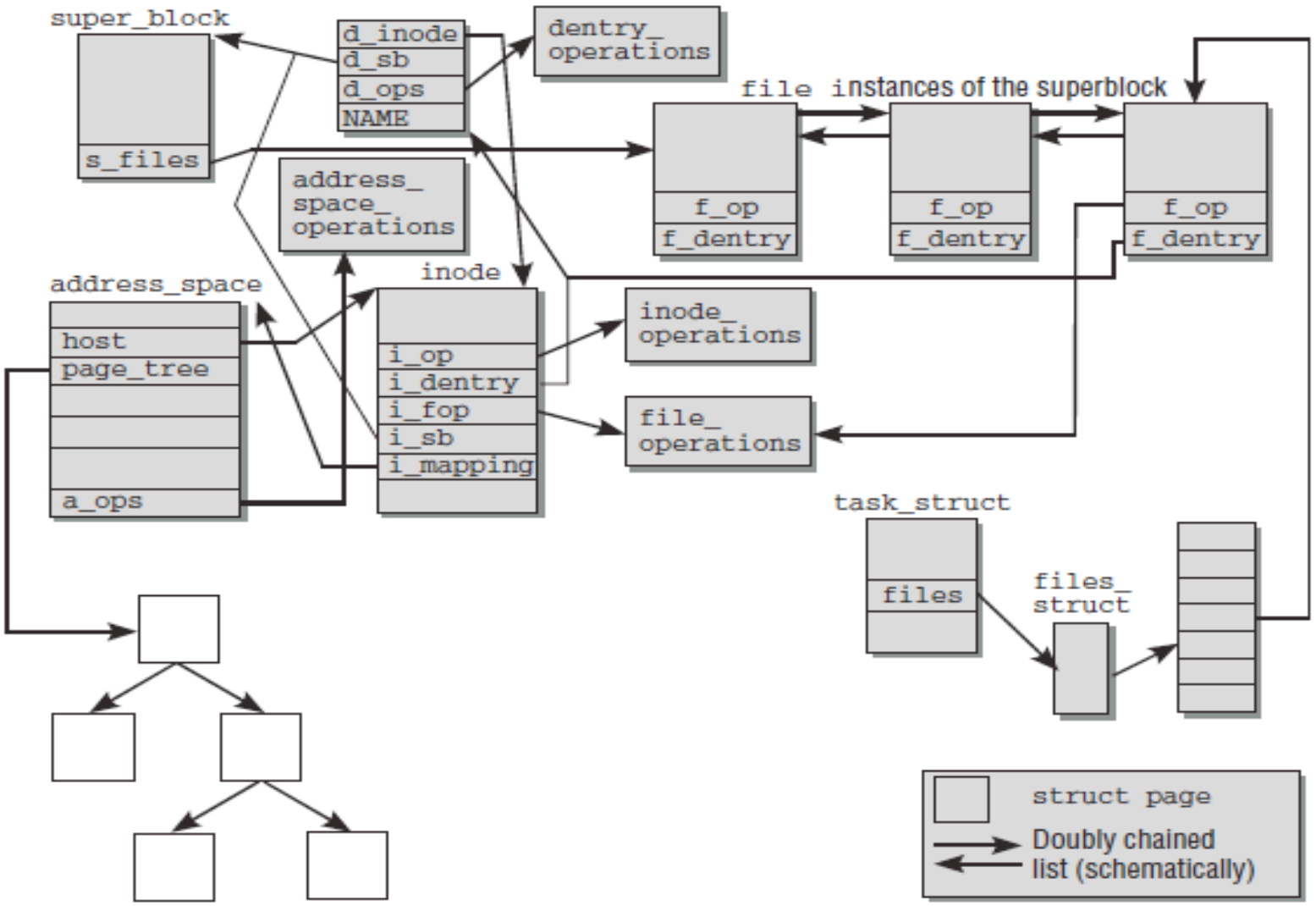
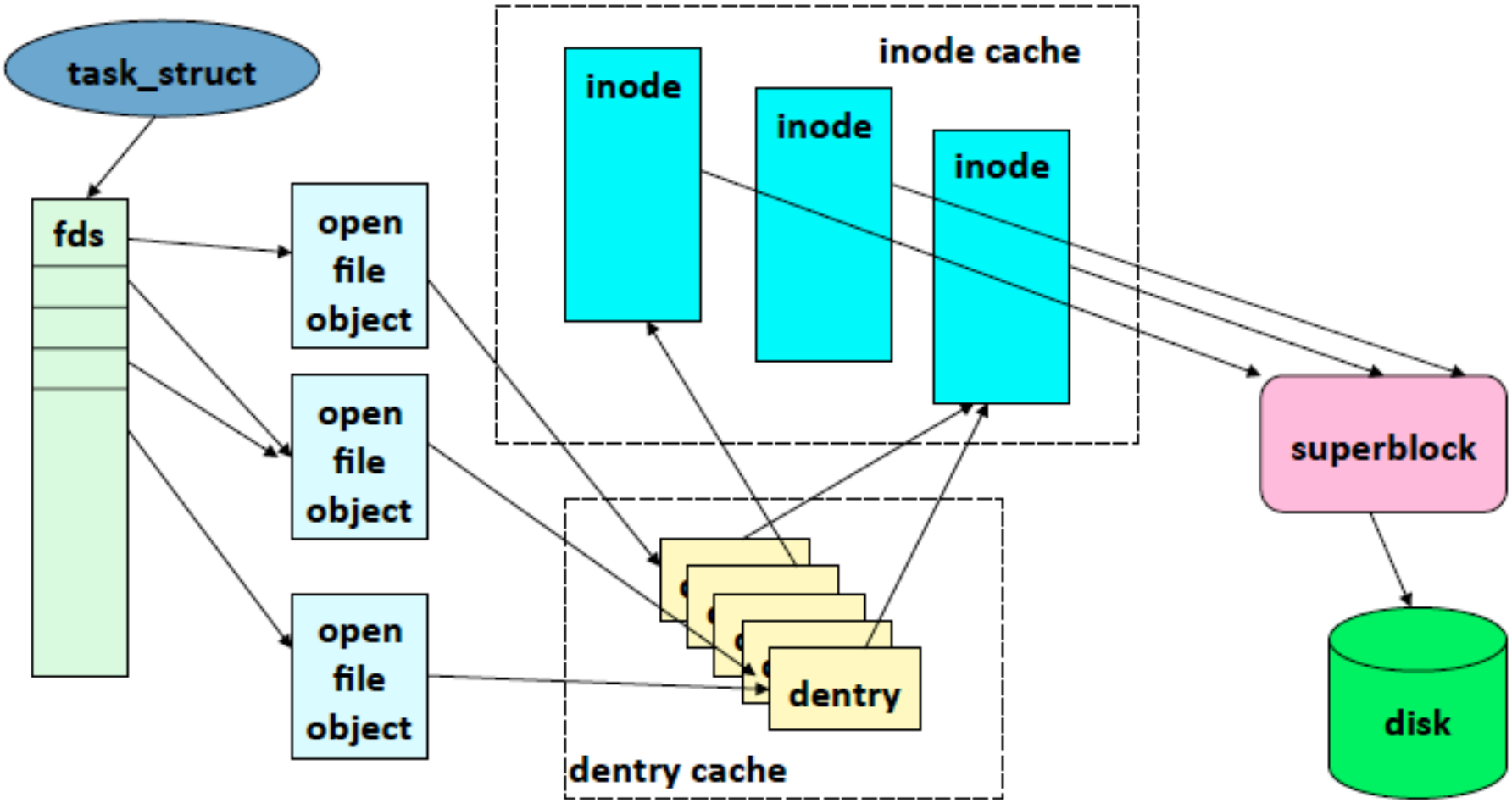


Figure 8-3: Interplay of the VFS components.


task_struct
fds
open file object
open file object
open file object
inode cache
inode
inode
inode
dentry
dentry cache
superblock
disk

4.2 El Sistema de Archivos Ext2 (Second Extended Filesystem)

(Mauerer pp. 584) Tiene un alto rendimiento; fue diseñado para cubrir los siguientes objetivos:

- Soportar tamaños de bloque variables para que el SA pueda hacer frente tanto al hecho de tener muchos archivos grande como muchos archivos pequeños.

- Implementar enlaces simbólicos rápidos, cuyas direcciones de enlace estén almacenadas en sus propios inodos y no en un bloque de datos (habrá un determinado límite en el número de caracteres).

- Integrar las extensiones en el diseño de forma que no se tenga que formatear y recargar el disco duro cuando se migre a una versión nueva.

- Minimizar los efectos de las caídas del sistema mediante una sofisticada estrategia de manipulación de datos en el medio de almacenamiento. Deben existir herramientas que permitan restaurar el SA en un estado anterior (aunque se pueden perder datos).

- Usar atributos especiales para etiquetar archivos como inmutables. Esto permite proteger los archivos importantes, como los de configuración, de cambios no deseables, incluso del superusuario.

Una característica de ext2 es que su código es muy compacto comparado con los SA modernos (menos de 10.000 líneas son suficientes).

a) Estructura física.

Estudiamos las estructuras en C usadas para gestionar los datos del SA.
El elemento central de Ext2 es el "grupo de bloques" (block group).
Estructura de un grupo de bloques:

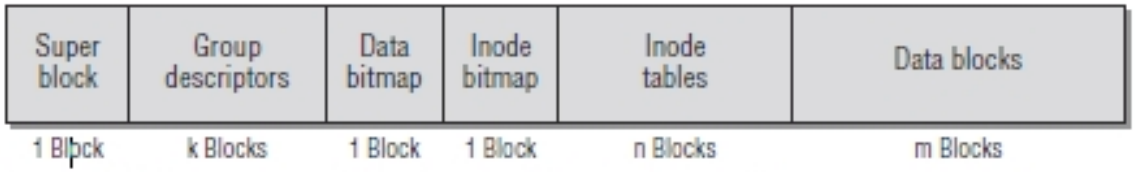


Figure 9-2: Block group of the Second Extended Filesystem.

Cada SA consta de un gran número de grupos de bloques secuenciales:

Figure 9-3: Boot sector and block groups on a hard disk.

El boot sector (Boot block) es una zona del disco duro cuyo contenido se carga automáticamente por la BIOS y se ejecuta cuando el sistema arranca. Incluye un cargador (boot loader), como LILO por ejemplo, que permite seleccionar uno de los sistemas instalados en el computador y que será responsable de continuar con el proceso de arranque.

Como se ve en la figura, cada grupo de bloques contiene información redundante. Justificación.....

Cada grupo de bloques contiene información redundante. Este gasto de espacio se permite por las razones siguientes:

1. Si el superbloque se destruye, toda la información sobre la estructura y contenidos del SA se pierde y sólo se puede recuperar si hay copias de seguridad.

2. Mantener los archivos y los datos para su gestión próximos, esto permite reducir los movimientos de la cabeza de lectura/escritura y así mejorar el rendimiento del SA.

En la práctica, los datos del superbloque no están duplicados en cada grupo de bloques; el kernel trabaja sólo con la primera copia del superbloque.

Cuando se realiza una comprobación del SA, se copia el contenido del primer superbloque en los demás.

Como esto consume bastante tiempo, las últimas versiones de Ext2 adoptaron la técnica "sparse superblock" que consiste en que el superbloque sólo se mantiene en los grupos 0 y 1 y en aquellos cuyo ID sea potencia de 3, 5 y 7.

Cuando se usa un SA, los datos del superbloque se almacenan en MP de forma que el kernel no tenga que estar continuamente leyendo esta información de disco.

Descripción de **cada elemento de un grupo de bloques**:

- **Superbloque** (Superblock): estructura central para almacenar meta-información del SA.
Incluye información sobre ...
    el número de bloque libres y usados
    tamaño de bloque
    estado actual del SA (usado cuando se levanta el sistema para así poder detectar inconsistencias si hubo una caída previamente)
    distintos tiempos (última vez que se montó el SA, instante de la última escritura, etc.)
    número mágico para que la rutina de montaje pueda comprobar su tipo.

- **Descriptores de grupo** (Group descriptors): contienen información que refleja el estado de los grupos de bloques individuales del SA., por ejemplo, el número de bloques libres e inodos libres del grupo. Cada grupo de bloques incluye descriptores para todos los grupos de bloques del SA.

- **Mapa de bits de datos y de inodos** (Data bitmap, Inode bitmap): contienen un bit por bloque de datos y por inodo respectivamente para indicar si están libres o no.

- **Tabla de inodos** (Inode tables): contiene todos los inodos del grupo de bloques.

Reflexión general sobre implementación

Un problema en la implementación del SA es que los archivos difieren mucho en tamaño.

> Algunos son muy grandes (contienen vídeos, por ejemplo) y otros sólo consumen unos pocos bytes (archivos de configuración).
>
> Además, hay distintos tipos de metainformación: por ejemplo, la información almacenada para dispositivos difiere de la almacenada para directorios, o archivos regulares, o pipes....

Las estructuras usadas para almacenar datos deben diseñarse para asegurar una óptima gestión de almacenamiento, por ejemplo....

* buena relación (tamaño metadatos) frente a (tamaño archivo)
* no demasiado espacio ocupado en la gestión del espacio libre
* permitir las ventajas de asignación no contigua

....

No siempre es fácil tener la más optima gestión de almacenamiento y, a la vez, tener una alta velocidad de acceso.

La forma de almacenar la información sobre los bloques de datos (BD) de un archivo es la que se muestra en la figura:

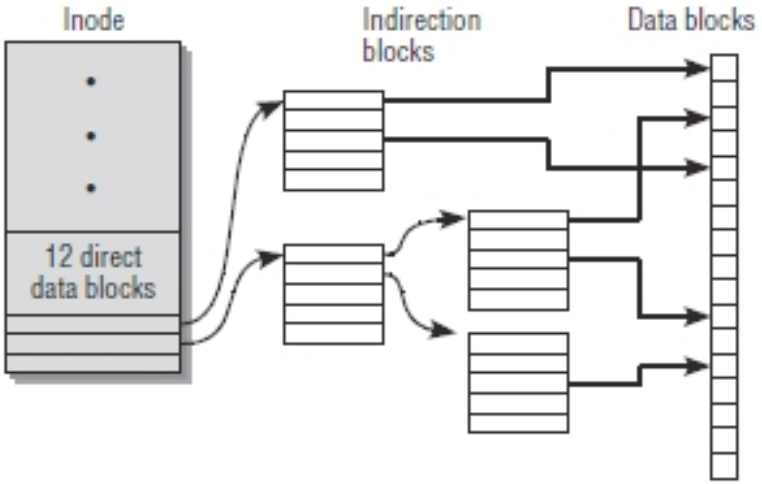


Figure 9-4: Simple and double indirection.

En el inodo tenemos almacenada la información para localizar los 12 primeros BD de un archivo. Después, si es necesario, se usa un sistema de bloques índices.

De esta forma, para archivos pequeños no necesitamos nada más que usar el inodo y para grandes, los bloques índices (el número dependerá del tamaño del archivo).

Existe también una triple indirección para archivos realmente grandes.
En arquitecturas de 32 bits , puesto que una dirección dentro de un archivos sólo puede representarse con 32 bits, usando las bibliotecas con objetos long, la capacidad máxima de un archivo es de 2^32 bytes => 2GB. Por eso, normalmente no se usa triple indirección, ya que con la doble ya se pueden direccionar los archivos más grandes.

A la hora de asignar bloques, Ext2 intenta minimizar la fragmentación

(fragmentación: grado en que los bloques de un archivo estén dispersos, reduciendo la velocidad de acceso)

se intenta mantener los bloques de un archivo en el mismo grupo de bloques

b) Estructuras de datos en el dispositivo de almacenamiento (Mauerer pp. 592) Punto (b) NO ENTRA

Se usan junto con las estructuras de datos del VFS.

***Estructura para el superbloque**: `struct ext2_super_block`*

Es la estructura central en que se mantienen los datos característios del SA. Su contenido es lo que primero ve el kernel cuando se monta un SA.

```
<ext2_fs.h>
struct ext2_super_block {
        __le32 s_inodes_count;          /* Inodes count */
        __le32 s_blocks_count;          /* Blocks count */
        __le32 s_r_blocks_count;        /*Reserved blocks count*/
        __le32 s_free_blocks_count;     /* Free blocks count */
        __le32 s_free_inodes_count; /* Free inodes count */
        __le32 s_first_data_block; /* First Data Block */
        __le32 s_log_block_size; /* Block size */
        __le32 s_log_frag_size; /* Fragment size */
        __le32 s_blocks_per_group; /* # Blocks per group */
        __le32 s_frags_per_group; /* # Fragments per group */
        __le32 s_inodes_per_group; /* # Inodes per group */
        __le32 s_mtime; /* Mount time */
```

NO ENTRA

```
        __le32 s_wtime; /* Write time */
        __le16 s_mnt_count; /* Mount count */
        __le16 s_max_mnt_count; /* Maximal mount count */
        __le16 s_magic; /* Magic signature */
        __le16 s_state; /* File system state */
        __le16 s_errors; /* Behaviour when detecting errors */
        __le16 s_minor_rev_level; /* minor revision level */
        __le32 s_lastcheck; /* time of last check */
        __le32 s_checkinterval; /* max. time between checks */
        __le32 s_creator_os; /* OS */
        __le32 s_rev_level; /* Revision level */
        __le16 s_def_resuid;/*Default uid for reserved blks */
        __le16 s_def_resgid;/*Default gid for reserved blks */
* These fields are for EXT2_DYNAMIC_REV superblocks only.
*
* Note: the difference between the compatible feature set and
* the incompatible feature set is that if there is a bit set
* in the incompatible feature set that the kernel doesn't
* know about, it should refuse to mount the filesystem.
*
* e2fsck's requirements are more strict; if it doesn't know
* about a feature in either the compatible or incompatible
* feature set, it must abort and not try to meddle with
* things it doesn't understand...
*/
        __le32 s_first_ino; /* First non-reserved inode */
        __le16 s_inode_size; /* size of inode structure */
        __le16 s_block_group_nr; /* block group # of this superblock */
        __le32 s_feature_compat; /* compatible feature set */
        __le32 s_feature_incompat; /* incompatible feature set */
        __le32 s_feature_ro_compat; /* readonly-compatible feature set */
        __u8 s_uuid[16]; /* 128-bit uuid for volume */
```

NO ENTRA

```
        char s_volume_name[16]; /* volume name */
        char s_last_mounted[64]; /* directory where last mounted */
        __le32 s_algorithm_usage_bitmap; /* For compression */
        /*
        * Performance hints. Directory preallocation should only
        * happen if the EXT2_COMPAT_PREALLOC flag is on.
        */
        __u8 s_prealloc_blocks; /* Nr of blocks to try to preallocate*/
        __u8 s_prealloc_dir_blocks; /* Nr to pre-allocate for dirs */
        __u16 s_padding1;
/*
* Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
*/
...
        __u32 s_reserved[190]; /* Padding to the end of the block */
};
```

Los campos `s_def_resuid` y `s_def_resgid` especifican el usuario y el grupo al que se le reserva cierto número de bloques de forma exclusiva.

El número de bloques reservados los tenemos en `s_r_blocks_count`.

Esto tiene sentido para que ciertos procesos, por ejemplo, un demonio que necesita espacio en disco para empezar a hacer su trabajo no pare porque no exista espacio en disco. Por esta razón se guarda más o menos un 5% de espacio para reservarlo normalmente para el superusuario y sus procesos.

NO ENTRA

La consistencia del SA es comprobada con ayuda de 3 variables:

1. `s_state` especifica el estado actual del SA.

   Cuando la partición se ha desmontado correctamente, toma el valor `EXT2_VALID_FS` (en `ext2_fs.h`) indicando así a `mount` que la partición está en un estado correcto.

   Si se desmontó incorrectamente (p.ej. se cayó el sistema) tendrá el valor `EXT2_ERROR_FS`. En este caso, se disparará `e2fsck` (comprobador de consistencia) automáticamente.

2. La fecha de la última comprobación se almacena en `s_lastcheck`. Si `s_checkinterval` ha transcurrido desde esta fecha, se fuerza una comprobación incluso si el SA está en un estado correcto.

3. Otra forma de forzar la comprobación de consistencia es la implementada con la ayuda de los contadores:

   `s_max_mnt_count` (máximo nº de montajes que se pueden hacer hasta forzar el chequeo)
   `s_mnt_count` (nº de veces que se ha montado desde el último chequeo).

   Cuando el número del 2º contador excede del 1º, entonces se fuerza la comprobación.

NO ENTRA

***Estructura para el descriptor de grupo**: `ext2_group_desc`*

La figura (Mauerer Fig 9.2) anteriormente vista muestra que cada grupo de bloques contiene justo tras el superbloque un conjunto de descriptores de grupo (normalmente un nº altode descriptores de grupo).

> En estos descriptores se almacena información no solo de los bloques asociados con el grupo de bloques donde está ubicado, sino también de los restantes grupos de bloques.

La estructura de datos usada para definir **cada descriptor de grupo** es la siguiente:

```
<ext2_fs.h>
struct ext2_group_desc
{
      __le32 bg_block_bitmap; /* Blocks bitmap block */
      __le32 bg_inode_bitmap; /* Inodes bitmap block */
      __le32 bg_inode_table; /* Inodes table block */
      __le16 bg_free_blocks_count; /* Free blocks count */
      __le16 bg_free_inodes_count; /* Free inodes count */
      __le16 bg_used_dirs_count; /* Directories count */
      __le16 bg_pad;......};
```

El kernel usa una copia de esta estructura para cada grupo de bloques.

NO ENTRA

Cada descriptor contiene entradas indicando:

- **`bg_block_bitmap:`** puntero al bloque que contiene un mapa de bits indicando el estado de ocupación de cada bloque de este grupo (usado/libre).
- **`bg_inode_bitmap:`** puntero al bloque que contiene un mapa de bits indicando el estado de ocupación de cada inodo dentro de la tabla de inodos de este grupo (usado/libre). Puesto que se conoce en qué bloque comienza la tabla de inodos y qué tamaño tiene cada inodo, el kernel puede convertir la posición de un determinado bit de este maña en su correspondiente posición en el disco.
- **`bg_free_blocks_count`**: El nº de bloques libres en ese grupo.
- **`bg_free_inodes_count`**: El nº de inodos libres en ese grupo.
- **`bg_used_dirs_count`**: El nº de directorios.

Desde cada grupo de bloques es posible determinar la siguiente información sobre cualquier grupo de bloques del SA:

- Posición del mapa de bits de bloques
- Posición del mapa de bits de inodos
- nº de bloques libres
- nº de inodos libres

Los bloques usados para almacenar los mapas de bits de bloques e inodos de cada grupo de bloques no están duplicados en cada grupo de bloques. Hay solo una ocurrencia de ellos en el SA.

Table 9-2: Maximum Sizes in a Block Group

Block size	Number of blocks
1,024	8,192
2,048	16,384
4,096	32,768

(Mauerer Tabla 9-2: Magnitudes máximas en un grupo de bloques

Se intenta mantener los bloques de un archivo en el mismo grupo de bloques

así se minimiza el recorrido de las cabezas de lectura/escritura entre mapas de bits de bloques, mapas de bits de inodos y bloques de datos

NO ENTRA

Estructura para el inodo: *struct ext2_inode*

Cada grupo de bloques tiene un mapa de bits de inodos y una tabla de inodos local que puede ocupar distintos bloques. La tabla de inodos consiste en una gran fila de estructuras inodo una tras otra.

```
<ext2_fs.h>
struct ext2_inode {
      __le16 i_mode; /* File mode */
      __le16 i_uid; /* Low 16 bits of Owner Uid */
      __le32 i_size; /* Size in bytes */
      __le32 i_atime; /* Access time */
      __le32 i_ctime; /* Creation time */
      __le32 i_mtime; /* Modification time */
      __le32 i_dtime; /* Deletion Time */
      __le16 i_gid; /* Low 16 bits of Group Id */
      __le16 i_links_count; /* Links count */
      __le32 i_blocks; /* Blocks count */
      __le32 i_flags; /* File flags */
      union {
                  struct {
                        __le32 l_i_reserved1;
                  } linux1;
                  struct {
                        ...
                  } hurd1;
                  struct {
                  ...
                  } masix1;
            } osd1; /* OS dependent 1 */
```

```
        __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32 i_generation; /* File version (for NFS) */
        __le32 i_file_acl; /* File ACL */
        __le32 i_dir_acl; /* Directory ACL */
        __le32 i_faddr; /* Fragment address */
        union {
              struct {
                    __u8 l_i_frag; /* Fragment number */
                    __u8 l_i_fsize; /* Fragment size */
                    __u16 i_pad1;
                    __le16 l_i_uid_high; /* these 2 fields */
                    __le16 l_i_gid_high; /* were reserved2[0] */
                    __u32 l_i_reserved2;
              } linux2;
              struct {
                    ...
              } hurd2;
              struct {
                    ...
              } masix2;
        } osd2; /* OS dependent 2 */
};
```

NO ENTRA

El tamaño de un inodo es constante y es de 120 bytes.

El número de inodos por grupo de bloques se especifica en la creación del SA.

Estructura para directorios y archivos

NO ENTRA

Cada directorio es representado por un inodo que tiene asignados bloques de datos. Estos bloques contienen una serie de **entradas de directorio**. Cada entrada de directorio tiene asociada una estructura llamada `ext2_dir_entry_2`.

```
<ext2_fs.h>
struct ext2_dir_entry_2 {
        __le32 inode; /* Inode number */
        __le16 rec_len; /* Directory entry length */
        __u8 name_len; /* Name length */
        __u8 file_type;
        char name[EXT2_NAME_LEN]; /* File name */
};
```

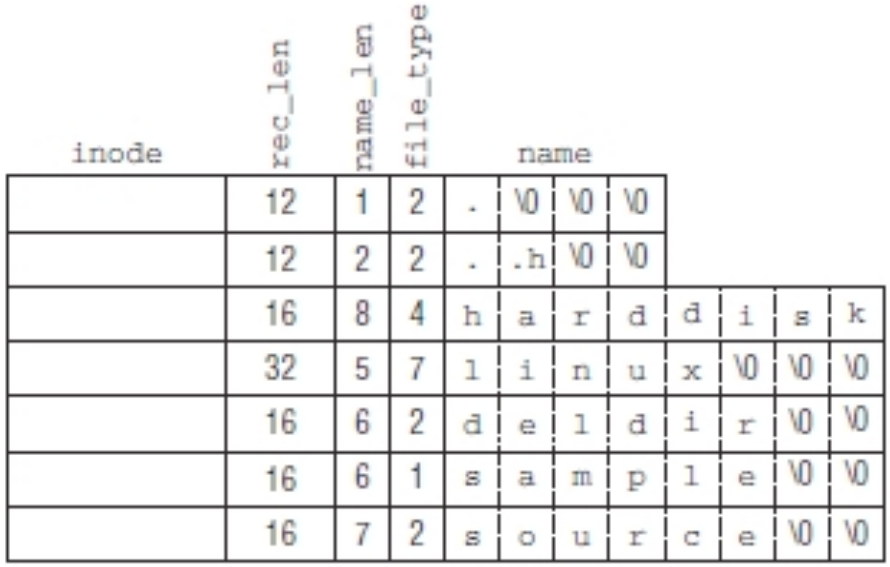
`file_type` especifica el tipo de archivo, es decir, de esa entrada de directorio. Esta variable acepta uno de los valores que se definen a continuación en la estructura `enum`:

```
<ext2_fs.h>
enum {
        EXT2_FT_UNKNOWN,
        EXT2_FT_REG_FILE,
        EXT2_FT_DIR,
        EXT2_FT_CHRDEV,
        EXT2_FT_BLKDEV,
        EXT2_FT_FIFO,
        EXT2_FT_SOCK,
        EXT2_FT_SYMLINK,
        EXT2_FT_MAX
};
```

Las dos primeras entradas de un directorio son "." y ".." (directorio actual y padre respectivamente). Las entradas gastan un nº de bytes múltiplo de 4, si no son necesarios los últimos bytes se rellenan con ceros (`\0`). Podemos ver en la siguiente figura un ejemplo del contenido de un directorio.

Ejemplo de representación de directorios en Ext2: (Mauerer)

```
01.02.03.04.05.06.07.08.09.10.11.12.13.14.15.16.....
```



inode	rec_len	name_len	file_type	name							
	12	1	2	.	\0	\0	\0				
	12	2	2	.	.h	\0	\0				
	16	8	4	h	a	r	d	d	i	s	k
	32	5	7	l	i	n	u	x	\0	\0	\0
	16	6	2	d	e	l	d	i	r	\0	\0
	16	6	1	s	a	m	p	l	e	\0	\0
	16	7	2	s	o	u	r	c	e	\0	\0

Figure 9-6: Representation of files and directories in the Second Extended Filesystem.

Significado de **rec_len (**2Bytes):

nº Bytes entre el fin de este `rec_len` y el final del siguiente `rec_len`

o bien:

nº Bytes entre el fin de este `rec_len` y el comienzo del siguiente `name_len` existente

Borrar una entrada de directorio:

Cuando una entrada de directorio se desea borrar el campo `rec_len` de la entrada anterior se establece al valor que apunte a la entrada siguiente de la que se quiere borrar.

En el ejemplo anterior, el resultado de `ls` no incluye `deldir` porque se borró anteriormente.

El tipo de archivo no está definido en el inodo sino en el campo `file_type` de la entrada del directorio padre (el que lo contiene).

Sólo los directorios y archivos regulares ocupan bloques de datos, los otros tipos de archivos tienen toda su información en el inodo (los contenidos de un inodo difieren según el tipo de archivo).

Enlaces simbólicos

El nombre del archivo al cual apunta se almacena en el inodo (si es de menos de 60 caracteres).

Se utiliza para ello la estructura `i_block` que contiene 15 entradas de 32 bits (que se utiliza normalmente para almacenar la información de la dirección de los bloques de datos)
(32 bits * 15 = 60 bytes).

Si el nombre del archivo a enlazar es mayor de 60 caracteres, el SA le asigna un bloque de datos donde se almacenará dicho nombre.

NO ENTRA

c) Estructuras de datos en Memoria Principal (pp. 604)

Linux mantiene mucha información de las estructuras de disco en MP para no tener que estar constantemente leyendo de disco. Ext2 mantiene ciertos campos que indiquen si una estructura de datos en MP se modificó desde que llegó a MP o no, es decir, si la copia en disco está actualizada o no.

Para incrementar el rendimiento de la asignación de bloques, Ext2 usa un mecanismo conocido como pre-asignación (pre-allocation). Si se requieren nuevos bloques para un archivo, no sólo se le asignan los que son necesarios sino un conjunto más de bloques que además sean del mismo grupo.

d) Crear un SA (Mauerer pp. 608)

Un SA se crea con la herramienta `mke2fs`.

Se crea....

- el directorio raíz **/**
- los subdirectorios **.** y **..**
- el subdirectorio `lost+found` para mantener los bloques defectuosos

NO ENTRA

e) Acciones del SA (Mauerer pp. 610)

Como se ha dicho antes, la asociación entre el VFS y la implementaciones específicas se establece gracias a tres estructuras que incluyen una serie de punteros a funciones. Esta asociación debe implementarse por todos los SA.

- Operaciones para gestionar los contenidos de un archivo: almacenadas en `file_operations`
- Operaciones para procesar objetos `file`: `inode_operations`
- Operaciones con espacios de direcciones generalizados: `address_space_operations`

Ext2 cuenta con diversas instancias de `file_operations` para diferentes tipos de archivos. La variante más usada es para archivos regulares.

fs/ext2/file.c

```
struct file_operations ext2_file_operations = {
      .llseek = generic_file_llseek,
      .read = do_sync_read,
      .write = do_sync_write,
      .aio_read = generic_file_aio_read,
      .aio_write = generic_file_aio_write,
      .ioctl = ext2_ioctl,
      .mmap = generic_file_mmap,
      .open = generic_file_open,
      .release = ext2_release_file,
      .fsync = ext2_sync_file,
      .readv = generic_file_readv,
```

```
        .splice_read = generic_file_splice_read,
        .splice_write = generic_file_splice_write,
};
```

NO ENTRA

Muchas de sus entradas mantienen punteros a funciones estándar de VFS. Todos los tipos tienen su propia instancia de `file_operations` (los directorios, los enlaces simbólicos, etc.).

fs/ext2/dir.c

```
struct file_operations ext2_dir_operations = {
        .llseek = generic_file_llseek,
        .read = generic_read_dir,
        .readdir = ext2_readdir,
        .ioctl = ext2_ioctl,
        .fsync = ext2_sync_file,
};
```

También están las operaciones sobre inodos para archivos y directorios y una cuarta estructura (`super_operations`) que se usa para interaccionar con el superbloque (conjunto de operaciones posibles como leer, escribir, borrar, etc.).

4.3 El Sistema de Archivos Ext3 (Third Extended Filesystem)

Concepto de transacción.

Una transacción es un conjunto de operaciones sobre el sistema de archivos que ha de realizarse de forma atómica con la semántica de "todo o nada":

> es decir, o bien la totalidad de las operaciones que forman la transacción tienen efecto sobre el SA y son por tanto visibles ante otros accesos, o bien no tiene efecto ninguna.

Determinados conjuntos de operaciones sobre un SA deben ser ejecutados con las características anteriores para garantizar la **consistencia de los datos**.

Ante una interrupción de una transacción en un punto intermedio.....

> deben quedar anuladas las operaciones hechas hasta ese momento...
> ..... o bien deben completarse las operaciones que no se llegaron a realizar antes de que el sistema de archivos sea accesible

(Mauerer pp.637) El sistema de archivos Ext3 cuenta con un mecanismo de "journal" en que almacenando adecuadamente qué operaciones se realizan sobre el SA se intenta implementar el concepto de transacción definido anteriormente.

En Ext3 cada operación sobre el SA se trata como una transacción, y se guarda en un journal antes de realizarse.

Cuando la transacción finaliza (cuando las modificaciones se han hecho), la información asociada es eliminada del journal.

Si ocurre un error, las operaciones pendientes se realizan cuando se vuelva a montar el SA.

Si la interrupción ocurre antes de que la transacción se escriba en el journal, la operación no se realiza ya que la información se ha perdido pero el sistema sigue en un estado consistente.

Ext3 no realiza milagros, es posible perder datos cuando un sistema se cae. Sin embargo, el SA siempre puede restaurarse después rápidamente a un estado consistente.

La sobrecarga adicional necesaria para almacenar las transacciones se refleja en el rendimiento de Ext3, que no se ajusta exactamente al de Ext2.

El sistema de archivos Ext3 puede trabajar de 3 formas diferentes para lograr un equilibrio entre la integridad de los datos y el rendimiento en cualquier situación:

1. **Modo Writeback** (reescritura): sólo se registran en el journal los cambios de los metadatos. Este modo garantiza una alto rendimiento pero una baja protección de datos.

2. **Modo Ordered** (ordenado): sólo se registran en el journal los cambios de los metadatos, pero se llevan acabo los cambios en los datos de especial interés o significado antes de que se lleven a cabo las operaciones sobre los metadatos. Es decir, hay más accesos a disco que en 1. Por tanto este modo es más lento que 1.

3. **Modo Journal**: todos los cambios, tanto a los datos como a los metadatos, se almacenan en el journal. Este modo garantiza el nivel más alto de protección de datos pero es el más lento. La posibilidad de perder datos es mínima.

El modo deseado se especifica cuando se monta el SA, su valor defecto es “Ordered”.