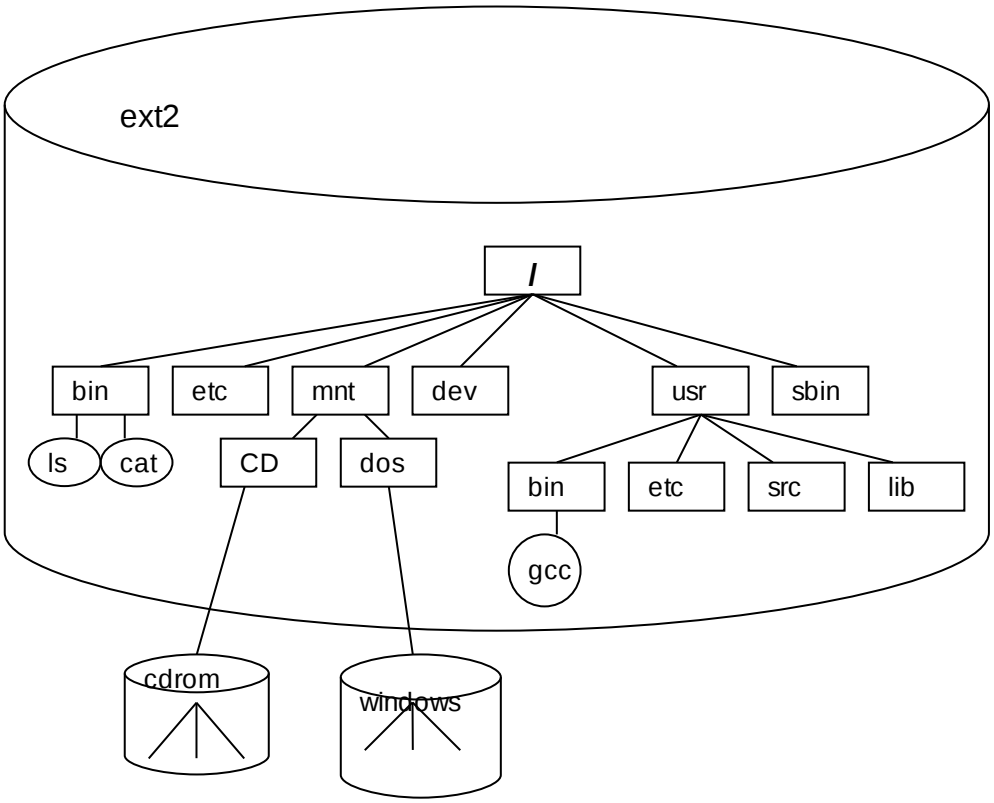
Lección 15: SISTEMA DE FICHEROS VIRTUAL - VFS

15.1 Introducción

El **sistema de ficheros** permite al núcleo mantener los ficheros (conjunto de datos) en una estructura de **un único árbol** del sistema en la que los nodos del árbol son directorios y los nodos hoja son los ficheros. También mantener la estructura del árbol con los enlaces. La información de los nodos del árbol se mantiene en las estructuras **inode**, y la estructura del sistema de ficheros en las estructuras **superbloque**.



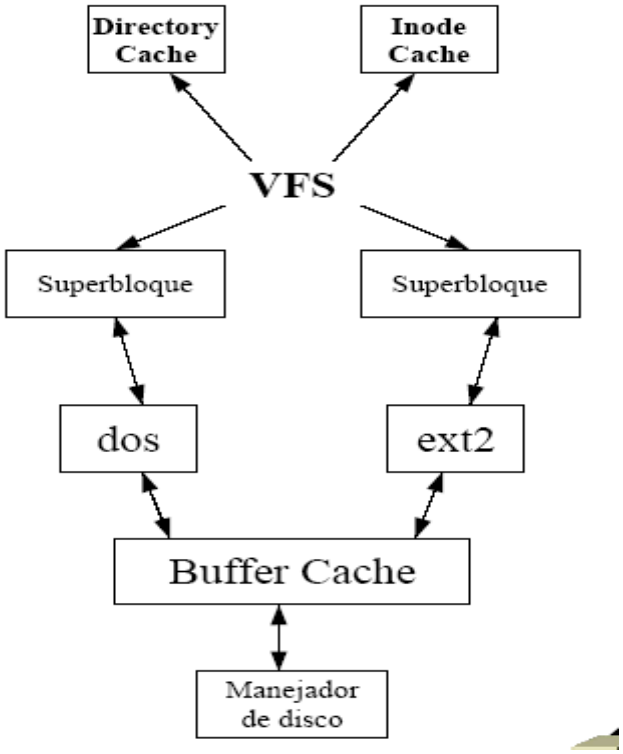
El sistema de ficheros diseñado por Remy Card en 1993 para Linux es el sistema de ficheros denominado “ext” (extensión de minix, actualmente la versión ext3) pero Linux permite trabajar con otros sistemas de ficheros reales como iso9660 que tienen los cdroms, vfat,o de otros sistemas operativos como minix, msdos, hpfs, etc. Para ello implementa una capa de software denominada sistema virtual de ficheros VFS por encima de los sistemas de ficheros reales, esta capa tiene un conjunto de estructuras en memoria principal para mantener el sistema de ficheros completo con un conjunto de funciones para manejarlas y una serie de estructuras en disco para mantener cada uno de los sistemas de ficheros reales con su conjunto particular de funciones para manejarlas. Todos los sistemas de ficheros reales para poder utilizarse tienen que engancharse (montarse) al árbol único del sistema de ficheros en un directorio denominado “mount directory”, directorio de montaje o también llamado “mount point”, punto de montaje. En la figura el sistema de ficheros del cdrom está montado en el directorio /mnt/CD y el sistema de ficheros

windows esta montado en el directorio /mnt/dos. Cada uno de estos sistemas de ficheros a su vez tienen sus propias estructuras de datos para mantenerse como el superbloque, las tablas de inodos, bitmaps, etc, que se ubican en el dispositivo de bloque, disco o partición, que sostiene a ese sistema de ficheros particular.

Cuando un proceso de usuario hace una llamada al sistema de ficheros, esta es atendida por el VFS, y posteriormente este dirige la llamada al sistema de ficheros particular de que se trate.

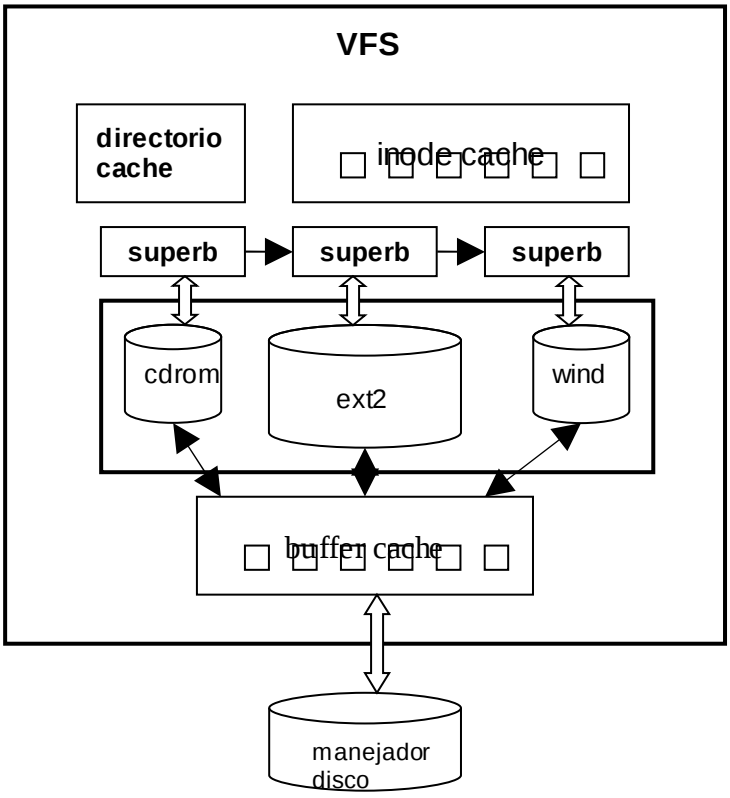
La filosofía de funcionamiento es:

1. Se produce una llamada al sistema que implica un acceso a un sistema de ficheros.
2. El VFS determina a qué sistema de ficheros pertenece.
3. Se comprueba si existe ya una entrada en el caché y si dicha entrada es válida
4. En el caso de que se requiera, se accede a la función del filesystem específico que realiza dicha función a través de las estructuras del VFS.
5. La función puede, o bien completar la llamada, o bien requerir una operación de E/S sobre un dispositivo, que en su caso puede provocar el paso a modo "sleep" del proceso que invocó la operación sobre el filesystem, hasta que la operación se complete.



Los sistemas de ficheros están almacenados físicamente en una partición de un dispositivo de bloques como son los discos, que pueden ser de distinto tipo: IDE, SATA, SCSI, CDROM... y con distintos manejadores. El

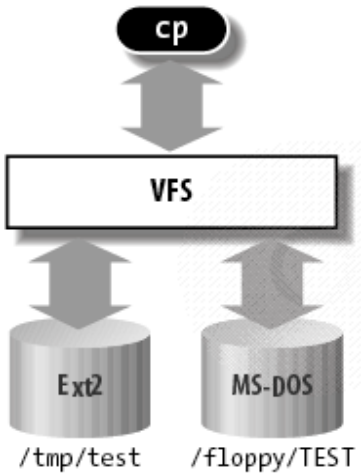
sistema de ficheros esconde todas las características físicas del disco como tipo de controladora, cabezas, cilindros, sectores y pistas y los trata como una secuencia de bloques de un cierto tamaño. Es el manejador del dispositivo el que se encarga de traducir las peticiones de bloques en cabeza, cilindro, pista.



Para que el resto del sistema operativo vea los distintos sistemas de ficheros reales como un único sistema de ficheros, el sistema de ficheros virtual VFS oculta todas las características de los sistemas de ficheros reales montados. Para ello mantiene unas estructuras de datos almacenadas en la memoria ram del núcleo y unos procedimientos que trabajan con ellas. Estas estructuras de datos serán llenadas con información almacenada en las estructuras de datos almacenadas en disco, especificas de cada sistema de fichero real. Los procedimientos de leer o escribir en disco también son específicos de cada sistema de ficheros real.

Estructuras de datos del sistema de ficheros virtual VFS en include/linux/fs.h	Estructuras de datos de un sistema de ficheros real (EXT3)
lista de sistemas soportados	superbloque
lista de sistemas montados	inode
superbloque	grupo de descriptores
inode	bitmap de bloques
buffer cache	bitmat de inode
cache de directorios	directorios
file	

Ejemplo de funcionamiento:



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

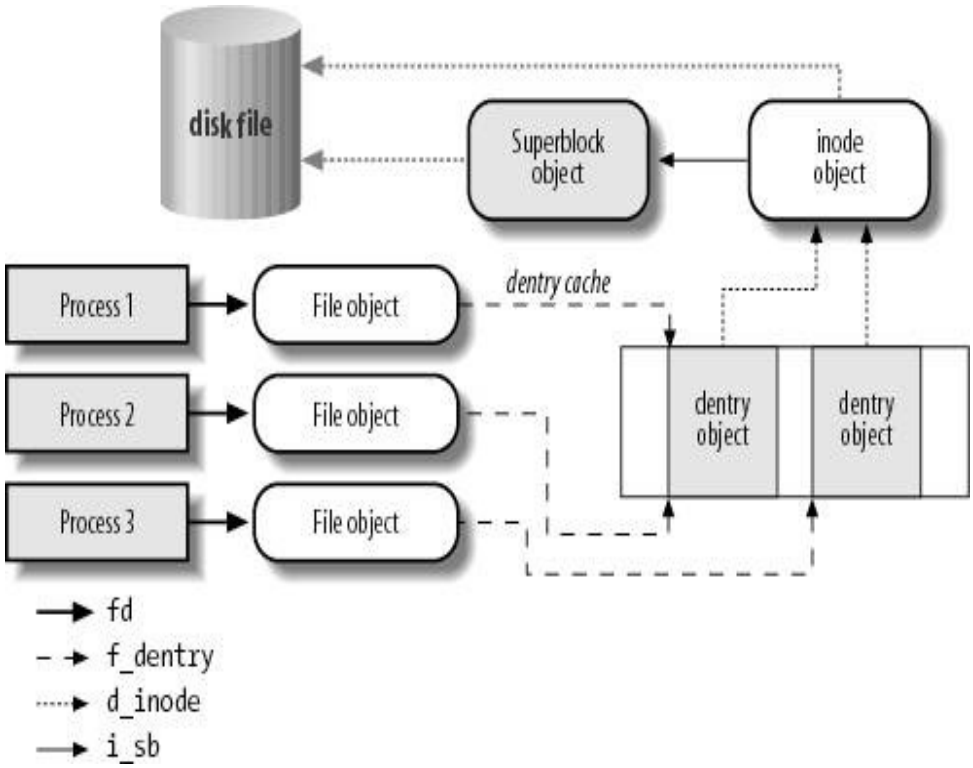
15.2 Estructura file_system_type

VFS tiene un conjunto de estructuras en memoria principal, definidas en el archivo include/linux/fs.h:

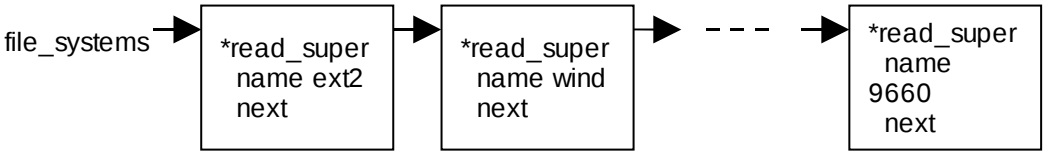
- lista de sistemas soportados
- lista de sistemas montados
- superbloque
- inode
- buffer cache
- cache de directorios
- File

Y una serie de funciones para poder manejarlas

Por su parte, cada sistema de ficheros tiene sus propias estructuras de datos en disco, y unas funciones propias para cada sistema para el manejo de dichas estructuras.



Cuando el sistema arranca cada sistema de ficheros se inicializa y se registra en el VFS llamando a la función **register_file_system**, en una lista de estructuras **file_system_type**, apuntadas por la variable **file_systems**.



Podemos ver los sistemas de ficheros registrados en el sistema leyendo el fichero **/proc/filesystems**. Se encuentra definida en el fichero: include/linux/fs.h

```
1565 struct file_system_type {
1566        const char *name;
1567        int fs_flags;
1568        int (*get_sb) (struct file_system_type *, int,
1569                       const char *, void *, struct vfsmount *);
1570        void (*kill_sb) (struct super_block *);
1571        struct module *owner;
1572        struct file_system_type * next;
1573        struct list_head fs_supers;
1574
1575        struct lock_class_key s_lock_key;
```

```
1576        struct lock_class_key s_umount_key;
1577
1578        struct lock_class_key i_lock_key;
1579        struct lock_class_key i_mutex_key;
1580        struct lock_class_key i_mutex_dir_key;
1581        struct lock_class_key i_alloc_sem_key;
1582};
```

name: Nombre del sistema de ficheros real.
fs_flags: Flags para indicar si el sistema de ficheros precisa un dispositivo de bloques para soportarlo.
***get_sb**: Función particular de cada sistema de ficheros real que lee su superbloque en disco y escribe en el superbloque del VFS. Se invoca en el montaje.
***kill_sb**: Deshace las inicializaciones y las asignaciones realizadas por get_sb(). Se invoca al desmontar el sistema de ficheros.
Next: Puntero al siguiente nodo de la lista.
fs_supers: Puntero al primer nodo de una lista que contiene todos los superbloques del mismo tipo.
Las demás estructuras tienen el propósito de cumplir el principio de exclusión mutua con respecto al sistema de ficheros.

15.3 Montar un sistema de ficheros real

Cuando un sistema de ficheros real se monta en el sistema, incluido el sistema de ficheros root, por ejemplo:

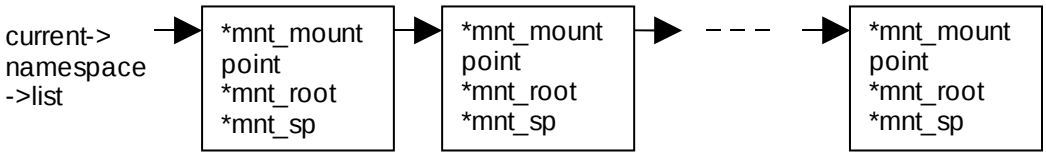
mount –t **iso9660** –o ro **/dev/cdrom** **/mnt/CD**

El tipo de sistema de ficheros: **iso9660**
El dispositivo físico de bloques que contiene el sistema de ficheros: **/dev/cdrom**
El directorio del sistema donde el sistema de ficheros se va a montar, llamado “mount directory” o “mount point” : **/mnt/CD**

1. El VFS debe buscar en la lista de estructura de sistemas de ficheros registrados file_system_type el sistema de ficheros [name].
2. Buscar en la cache de inodos el inodo del directorio donde el sistema de ficheros va a montarse ([mount directory]).
3. Llamar al procedimiento get_sb, para leer el superbloque almacenado en el dispositivo, además de información necesaria para crear el superbloque VFS en el vector “super_blocks”.

Bajo Linux, la información sobre los sistemas de ficheros montados es mantenida en dos estructuras separadas - super_block y vfsmount. El motivo para esto es que Linux permite montar el mismo sistema de ficheros (dispositivo de bloque) bajo múltiples puntos de montaje, lo cual significa que el mismo super_block puede corresponder a múltiples estructuras vfsmount.

Cada sistema de ficheros montado se describe por una estructura vfsmount. Todos ellos forman una lista apuntada por current->namespace->list, en /include/linux/mount.h.



Esta estructura contiene:

```
 38 struct vfsmount {
 39        struct list_head mnt_hash;
 40        struct vfsmount *mnt_parent;     /* fs donde estamos montado */
 41        struct dentry *mnt_mountpoint;  /* dentry del punto de montaje
*/
 42        struct dentry *mnt_root;         /* raiz del arbol montado */
 43        struct super_block *mnt_sb;      /* puntero al superbloque del fs
montado */
 44       struct list_head mnt_mounts;      /* lista de fs montados por
debajo de nuestro punto de montaje */
 45        struct list_head mnt_child;      /* and going through their
mnt_child */
 46        int mnt_flags;
 47
 48        const char *mnt_devname;         /* Nombre del dispositivo
ej. /dev/dsk/hda1 */
 49        struct list_head mnt_list;
 50        struct list_head mnt_expire;     /* link in fs-specific expiry list */
 51        struct list_head mnt_share;      /* lista circular de montajes
compartidos */
 52        struct list_head mnt_slave_list;  /* lista de montajes esclavos */
 53        struct list_head mnt_slave;
 54        struct vfsmount *mnt_master;
 55        struct mnt_namespace *mnt_ns;          /* namespace */
 56        int mnt_id;                      /* identificador de montaje */
 57        int mnt_group_id;
 58        /*
 59         * We put mnt_count & mnt_expiry_mark at the end of struct
vfsmount
 60         * to let these frequently modified fields in a separate cache line
```

```
 61        * (so that reads of mnt_flags wont ping-pong on SMP machines)
 62        */
 63       atomic_t mnt_count;
 64       int mnt_expiry_mark;              /* verdadero si está marcado
para expiración */
 65       int mnt_pinned;
 66       int mnt_ghosts;
 67       /*
 68        * This value is not stable unless all of the mnt_writers[] spinlocks
 69         * are held, and all mnt_writer[]s on this mount have 0 as their -
>count
 70        */
 71       atomic_t __mnt_writers;
 72};
```

Para desmontar un sistema de ficheros mediante “umount” se debe cumplir que no se debe estar usando ninguno de sus ficheros. Si el superbloque de este sistema de ficheros tiene el campo dirty activado, se debe actualizar el superbloque en disco y posteriormente se libera la estructura super_block de memoria y la estructura vfsmount.

Existen varias funciones que trabajan con esta estructura vfsmount:

lookup_vfsmnt – Busca un sistema de archivos en la lista.
add_vfsmnt – Añade un sistema de ficheros montado en la lista.
remove_vfsmnt – Quita de la lista un sistema de ficheros que se desmonta.

15.4 El superbloque

Cuando los sistemas ficheros reales se montan en el sistema son representados en el sistema en la **tabla super_blocks** de NR_SUPER elementos de estructura **super_block**. Esta tabla se encuentra en la memoria principal del núcleo. La estructura se encuentra definida en el fichero: include/linux/fs.h

```
1132 struct super_block {
1133       struct list_head       s_list;        /* Keep this first */
1134       dev_t                  s_dev;              /*  search  index;  _not_
kdev_t */
1135       unsigned long          s_blocksize;
1136       unsigned char          s_blocksize_bits;
1137       unsigned char          s_dirt;
1138       unsigned long long s_maxbytes;     /* Max file size */
1139       struct file_system_type    *s_type;
1140       const struct super_operations   *s_op;
1141       struct dquot_operations   *dq_op;
1142       struct quotactl_ops        *s_qcop;
```

```
1143        const struct export_operations  *s_export_op;
1144        unsigned long               s_flags;
1145        unsigned long               s_magic;
1146        struct dentry                       *s_root;
1147        struct rw_semaphore         s_umount;
1148        struct mutex                        s_lock;
1149        int                         s_count;
1150        int                         s_need_sync_fs;
1151        atomic_t                    s_active;
1152#ifdef CONFIG_SECURITY
1153        void                        *s_security;
1154#endif
1155        struct xattr_handler        **s_xattr;
1156
1157        struct list_head            s_inodes;       /* all inodes */
1158        struct list_head                    s_dirty;        /* dirty inodes */
1159        struct list_head            s_io;           /* parked for writeback */
1160        struct list_head                    s_more_io;          /*  parked  for
more writeback */
1161        struct hlist_head           s_anon;      /* anonymous dentries for
(nfs) exporting */
1162        struct list_head                    s_files;
1163             /*  s_dentry_lru  and  s_nr_dentry_unused  are  protected  by
dcache_lock */
1164        struct list_head                    s_dentry_lru;    /* unused dentry
lru */
1165        int                         s_nr_dentry_unused;     /* # of dentry
on lru */
1166
1167        struct block_device         *s_bdev;
1168        struct mtd_info             *s_mtd;
1169        struct list_head                    s_instances;
1170        struct quota_info           s_dquot;            /* Diskquota  specific
options */
1171
1172        int                         s_frozen;
1173        wait_queue_head_t           s_wait_unfrozen;
1174
1175        char s_id[32];                      /* Informational name */
1176
1177        void                    *s_fs_info;     /* Filesystem private info */
1178        fmode_t                 s_mode;
1179
1180        /*
1181             * The next field is for VFS *only*. No filesystems have any
business
1182         * even looking at it. You had been warned.
1183         */
1184        struct mutex s_vfs_rename_mutex;        /* Kludge */
1185
```

```
1186        /* Granularity of c/m/atime in ns.
1187           Cannot be worse than a second */
1188        u32             s_time_gran;
1189
1190        /*
1191         * Filesystem subtype.  If non-empty the filesystem type field
1192         * in /proc/mounts will be "type.subtype"
1193         */
1194        char *s_subtype;
1195
1196        /*
1197         * Saved mount options for lazy filesystems using
1198         * generic_show_options()
1199         */
1200        char *s_options;
1201
1202        /*
1203         * storage for asynchronous operations
1204         */
1205        struct list_head s_async_list;
1206};
```

s_list: una lista doblemente enlazada de todos los superbloques activos.
s_dev: para sistemas de archivos que requieren un bloque para ser montado en él.
s_blocksize, s_blocksize_bits: tamaño del bloque y log2(tamaño del bloque).
s_lock: indica cuando un superbloque está actualmente bloqueado por lock_super() / unlock_super().
s_dirt: establece cuando superbloque está modificado, y limpiado cuando es vuelto a ser escrito a disco.
s_type: puntero a struct file_system_type del sistema de archivos correspondiente.
s_op: puntero a la estructura super_operations, la cual contiene métodos específicos del sistema de archivos para leer/escribir inodos, etc.
dq_op: operaciones de cuota de disco.
s_flags: banderas de superbloque.
s_magic: número mágico del sistema de archivos. Usado por el sistema de archivos de minix para diferenciar entre múltiples tipos de él mismo.
s_root: dentry de la raiz del sistema de archivos.
s_wait: cola de espera de los procesos esperando para que el superbloque sea desbloqueado.
s_dirty: una lista de todos los inodos sucios.
s_files: una lista de todos los archivos abiertos en este superbloque.
s_bdev: para FS_REQUIRES_DEV, esto apunta a la estructura block_device describiendo el dispositivo en el que el sistema de archivos está montado.
s_mounts: una lista de todas las estructuras vfsmount, una por cada instancia montada de este superbloque.
s_dquot: más miembros de diskquota.

15.4.1 Operaciones genéricas de vfs sobre el superbloque

Esta estructura contiene un campo s_op con punteros a funciones que trabajan con la estructura **super_block**, y la estructura inode, estas funciones estan definidas y son propias de cada sistema de ficheros real.

```
1382 struct super_operations {
1383       struct inode *(*alloc_inode)(struct super_block *sb);
1384       void (*destroy_inode)(struct inode *);
1385
1386       void (*dirty_inode) (struct inode *);
1387       int (*write_inode) (struct inode *, int);
1388       void (*drop_inode) (struct inode *);
1389       void (*delete_inode) (struct inode *);
1390       void (*put_super) (struct super_block *);
1391       void (*write_super) (struct super_block *);
1392       int (*sync_fs)(struct super_block *sb, int wait);
1393       int (*freeze_fs) (struct super_block *);
1394       int (*unfreeze_fs) (struct super_block *);
1395       int (*statfs) (struct dentry *, struct kstatfs *);
1396       int (*remount_fs) (struct super_block *, int *, char *);
1397       void (*clear_inode) (struct inode *);
1398       void (*umount_begin) (struct super_block *);
1399
1400       int (*show_options)(struct seq_file *, struct vfsmount *);
1401       int (*show_stats)(struct seq_file *, struct vfsmount *);
1402#ifdef CONFIG_QUOTA
1403         ssize_t (*quota_read)(struct super_block *, int, char *, size_t,
loff_t);
1404         ssize_t (*quota_write)(struct super_block *, int, const char *,
size_t, loff_t);
1405#endif
1406       int (*bdev_try_to_free_page)(struct super_block*, struct page*,
gfp_t);
1407};
```

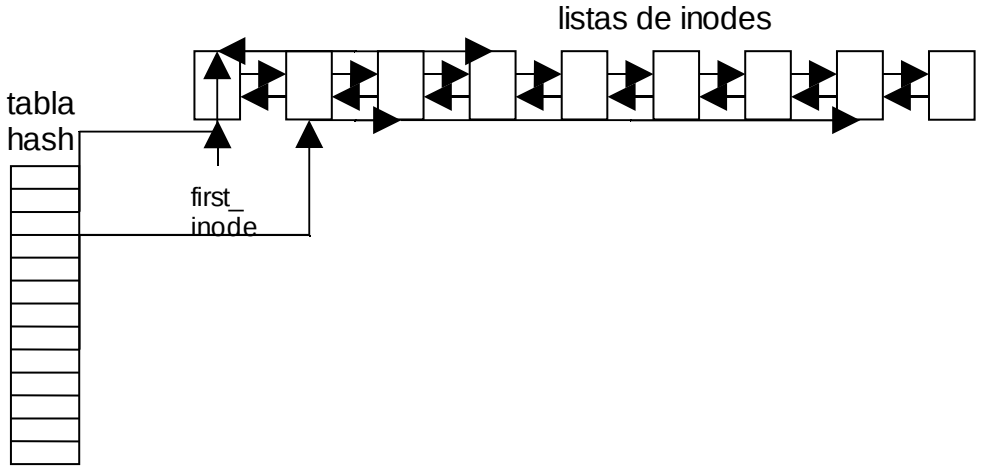
read_inode: lee un inodo desde el sistema de archivos.
write_inode: escribe un inodo de vuelta al disco.
put_inode: llamado cuando la cuenta de referencia es decrementada.
delete_inode: llamado cuando inode->i_count y inode->i_nlink llegan a 0.
put_super: llamado en las últimas etapas de la llamada al sistema.
umount para notificar al sistema de archivos que cualquier información mantenida por el sistema de archivos sobre esa instancia tiene que ser liberada.
write_super: llamado cuando el superbloque necesita ser vuelto a escribir en el disco.

statfs: implementa las llamadas al sistema **fstatfs / statfs**.
remount_fs: llamado cuando el sistema de archivos está siendo remontado.
clear_inode: llamado desde el nivel VFS clear_inode(). Los sistemas que incluyen datos privados a la estructura del inodo (a través del campo generic_ip) deben liberarse aquí.
umount_begin: llamado durante el desmontaje forzado para notificarlo al sistema de archivos de antemano, por lo tanto puede ser lo mejor para asegurarse que nada mantiene al sistema de archivos ocupado.

15.5 Estructura inode

Todo fichero o directorio que está siendo accedido, independiente del sistema de ficheros real donde esté ubicado, se representado dentro de VFS por un **struct inode** (/include/linux/fs.h) en la memoria principal del núcleo. Cada inode se llena, mediante funciones específicas del sistema de ficheros real, con la información del inode en disco más información del sistema. Dado el número de inodes que maneja el sistema, para un acceso rápido, se estructuran en listas en el **inode cache**.

Todos los inode se combinan en una **lista** global doblemente encadenada. Además mediante una tabla de **hash** se accede a listas con los inodos que tienen el mismo valor de hash. El valor de hash se obtiene con el número de dispositivo más el número de inode.



Si un inodo se necesita y no esta en la cache, se busca en el disco el inodo y se almacena en la cache. Existe una política para mantener o sacar un inodo de la cache basándose en el uso "**count**". Los inodes

modificados “**dirty**” tienen que escribirse en disco. Las funciones de lectura y escritura de inodos en disco son específicas de cada sistema de ficheros “**inode_operations**”.

```
649 struct inode {
650       struct hlist_node      i_hash;
651       struct list_head       i_list;
652       struct list_head       i_sb_list;
653       struct list_head       i_dentry;
654       unsigned long          i_ino;
655       atomic_t               i_count;
656       unsigned int           i_nlink;
657       uid_t                  i_uid;
658       gid_t                  i_gid;
659       dev_t                  i_rdev;
660       u64                    i_version;
661       loff_t                 i_size;
662#ifdef __NEED_I_SIZE_ORDERED
663       seqcount_t             i_size_seqcount;
664#endif
665       struct timespec        i_atime;
666       struct timespec        i_mtime;
667       struct timespec        i_ctime;
668       unsigned int           i_blkbits;
669       blkcnt_t               i_blocks;
670       unsigned short         i_bytes;
671       umode_t                i_mode;
672       spinlock_t             i_lock; /* i_blocks, i_bytes, maybe i_size */
673       struct mutex           i_mutex;
674       struct rw_semaphore    i_alloc_sem;
675       const struct inode_operations   *i_op;
676       const struct file_operations    *i_fop; /* former ->i_op->default_file_ops
*/
677       struct super_block     *i_sb;
678       struct file_lock       *i_flock;
679       struct address_space    *i_mapping;
680       struct address_space    i_data;
681#ifdef CONFIG_QUOTA
682       struct dquot           *i_dquot[MAXQUOTAS];
683#endif
684       struct list_head       i_devices;
685       union {
686             struct pipe_inode_info  *i_pipe;
687             struct block_device     *i_bdev;
688             struct cdev             *i_cdev;
689       };
690       int                    i_cindex;
691
692       __u32                  i_generation;
693
694#ifdef CONFIG_DNOTIFY
695       unsigned long          i_dnotify_mask; /* Directory notify events */
696       struct dnotify_struct   *i_dnotify; /* for directory notifications */
```

```
697#endif
698
699#ifdef CONFIG_INOTIFY
700     struct list_head     inotify_watches; /* watches on this inode */
701     struct mutex         inotify_mutex;  /* protects the watches list */
702#endif
703
704     unsigned long        i_state;
705     unsigned long        dirtied_when;   /* jiffies of first dirtying */
706
707     unsigned int         i_flags;
708
709     atomic_t             i_writecount;
710#ifdef CONFIG_SECURITY
711     void                 *i_security;
712#endif
713     void                 *i_private; /* fs or device private pointer */
714};
```

i_hash: lista de inodos con el mismo valor de hash.
i_list: lista global de inodos.
i_ino: número de inode (único).
i_count: número de usos del inode.
i_mode: flags.
i_nlink: número de enlaces a este inode.
i_uid: identificador de usuario propietario.
i_gid: identificador de grupo propietario.
i_size: tamaño del fichero asociado.
i_time: fecha del último acceso.
i_mtime: fecha de última modificación.
i_ctime: fecha de creación.
i_blkbits: tamaño del bloque en bits
i_blksize: tamaño del bloque
i_blocks: número de bloques que ocupa el inode.
i_lock: cerrojo para asegurar exclusión mutua.
i_op: puntero a estructura de operaciones de inode.
i_fop: puntero a estructura de operaciones de ficher.
i_sb: puntero al superbloque.
i_mapping / i_data: puntero a las áreas y páginas de memoria de este inode.
i_dquot[MAXQUOTAS]: cuotas de disco.
i_pipe: puntero a estructura pipe si el inode representa un pipe.
i_cdev: puntero a block_device si el inode representa un dispositivo de modo bloque.
dirtied_when: fecha de la primera modificación.
i_writecount: número de accesos de escritura.

15.5.1 Estructura address_space

A esta estructura se hace referencia dentro de cada inode con el puntero *i_mapping y la variable i_data, contiene las virtual áreas VMA y las páginas de este inodo.

```
549 struct address_space {
550       struct inode          *host;         /* owner: inode, block_device */
551       struct radix_tree_root  page_tree;      /* radix tree of all pages */
552       spinlock_t            tree_lock;      /* and lock protecting it */
553       unsigned int          i_mmap_writable;/* count VM_SHARED mappings */
554       struct prio_tree_root   i_mmap;         /* tree of private and shared
mappings */
555       struct list_head        i_mmap_nonlinear;/*list VM_NONLINEAR mappings
*/
556       spinlock_t            i_mmap_lock;    /* protect tree, count, list */
557       unsigned int          truncate_count; /* Cover race condition with
truncate */
558       unsigned long           nrpages;        /* number of total pages */
559       pgoff_t               writeback_index;/* writeback starts here */
560       const struct address_space_operations *a_ops;   /* methods */
561       unsigned long           flags;          /* error bits/gfp mask */
562       struct backing_dev_info *backing_dev_info; /* device readahead, etc */
563       spinlock_t            private_lock;   /* for use by the address_space */
564       struct list_head        private_list;   /* ditto */
565       struct address_space    *assoc_mapping; /* ditto */
566} __attribute__((aligned(sizeof(long))));
```

15.5.2 Estructura inode_operations

Contiene punteros a funciones que trabajan con la estructuras inode, directorio y fichero. Son propias y se implementan en el sistema de ficheros real.

```
1339 struct inode_operations {
1340       int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
1341       struct dentry * (*lookup) (struct inode *,struct dentry *, struct
nameidata *);
1342       int (*link) (struct dentry *,struct inode *,struct dentry *);
1343       int (*unlink) (struct inode *,struct dentry *);
1344       int (*symlink) (struct inode *,struct dentry *,const char *);
1345       int (*mkdir) (struct inode *,struct dentry *,int);
1346       int (*rmdir) (struct inode *,struct dentry *);
1347       int (*mknod) (struct inode *,struct dentry *,int,dev_t);
1348       int (*rename) (struct inode *, struct dentry *,
1349                   struct inode *, struct dentry *);
1350       int (*readlink) (struct dentry *, char __user *,int);
1351       void * (*follow_link) (struct dentry *, struct nameidata *);
1352       void (*put_link) (struct dentry *, struct nameidata *, void *);
1353       void (*truncate) (struct inode *);
1354       int (*permission) (struct inode *, int);
1355       int (*setattr) (struct dentry *, struct iattr *);
```

```
1356        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
1357        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
1358        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
1359        ssize_t (*listxattr) (struct dentry *, char *, size_t);
1360        int (*removexattr) (struct dentry *, const char *);
1361        void (*truncate_range)(struct inode *, loff_t, loff_t);
1362        long (*fallocate)(struct inode *inode, int mode, loff_t offset,
1363                          loff_t len);
1364        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
1365                      u64 len);
1366};
```

15.6 Estructura dentry

Consideramos cada directorio como un fichero que contiene una lista de ficheros y otros directorios.
Cuando leemos una entrada de directorio, el kernel crea un objeto dentry para cada componente de su inode.
Dentry no tiene imagen en disco

Ejemplo: Cuando buscamos en la ruta /tmp/test, el kernel crea un objeto dentry para el directorio / , un segundo objeto para la entrada tmp del directorio raíz y un tercer objeto dentry para la entrada test del directorio /tmp/
Definida en: include/linux/dcache.h

```
struct dentry {
 90        atomic_t d_count;
 91        unsigned int d_flags;          /* protected by d_lock */
 92        spinlock_t d_lock;             /* per dentry lock */
 93        int d_mounted;
 94        struct inode *d_inode;         /* Where the name belongs to - NULL is
 95                                        * negative */
 96        /*
 97         * The next three fields are touched by __d_lookup.  Place them here
 98         * so they all fit in a cache line.
 99         */
100        struct hlist_node d_hash;      /* lookup hash list */
101        struct dentry *d_parent;       /* parent directory */
102        struct qstr d_name;
103
104        struct list_head d_lru;        /* LRU list */
105        /*
106         * d_child and d_rcu can share memory
107         */
108        union {
109                struct list_head d_child;      /* child of parent list */
110                struct rcu_head d_rcu;
111        } d_u;
```

```
112        struct list_head d_subdirs;     /* our children */
113        struct list_head d_alias;       /* inode alias list */
114        unsigned long d_time;           /* used by d_revalidate */
115        struct dentry_operations *d_op;
116        struct super_block *d_sb;       /* The root of the dentry tree */
117        void *d_fsdata;                 /* fs-specific data */
118
119        unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
120};
```

15.7 Cache de directorios

Dado un nombre de fichero o directorio, el VFS tiene que encontrar el número de dispositivo y el número de inode que le corresponde. Para que la busqueda sea más rápida el VFS mantiene dos tablas cache de directorio: **level1_dcache** y **level2_dcache**, con las entradas de directorio más utilizadas y sus inodos. Cada cache se estructura como una tabla de hash con varias listas, donde cada entrada apunta a una lista que tiene el mismo valor de hash. La función de hash utiliza el número de dispositivo que tiene el sistema de ficheros, y el nombre del directorio. Se implementa en el fichero fs/dcache.c

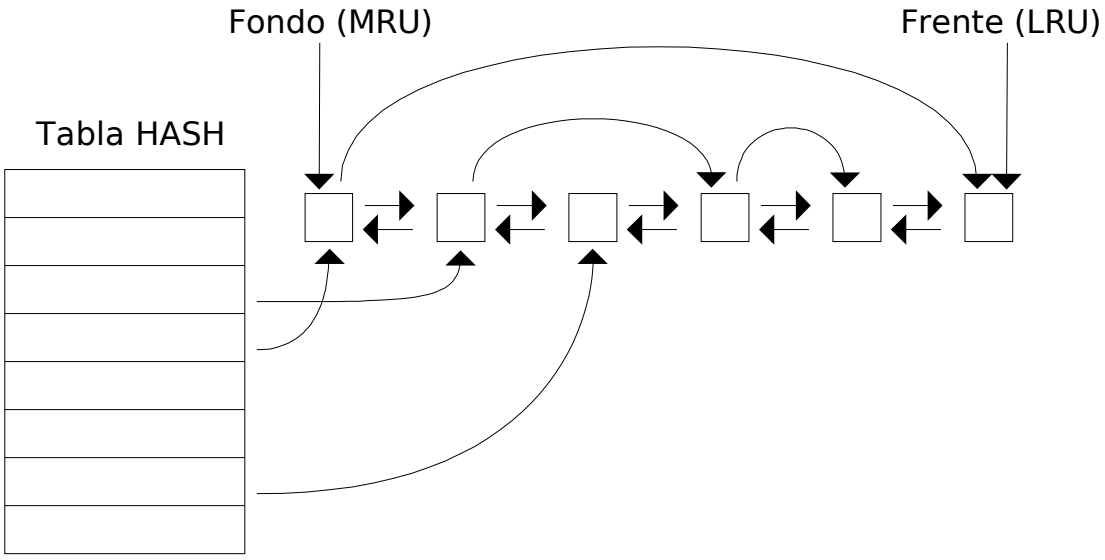
15.8 Buffer cache

Para acelerar la E/S a disco se utilizan memorias intermedias estructuradas en listas y ubicadas en memoria principal (buffer-cache).
La lectura de un bloque de disco se guarda en un buffer mientras se esté utilizando y no se necesite ese espacio de memoria. Los sucesivos accesos no se realizan sobre el disco, sino sobre el buffer.
Una modificación del buffer no se vuelca sobre la marcha a disco. Periódicamente, el proceso “update” llama a la primitiva “sync()” para forzar la reescritura de todos los buffers modificados con esto el sistema experimenta una notable reducción de las E/S a disco.

El buffer-cache mantiene copias de bloques de disco individuales. Las entradas del caché están identificadas por el dispositivo y número de bloque. Cada *buffer* se refiere a cualquier bloque en el disco y consiste de una cabecera y un área de memoria *igual* al tamaño del bloque del dispositivo. Para minimizar la sobrecarga, los *buffers* se mantienen en una de varias listas enlazadas: sin usar (*unused*), libres (*free*), no modificadas (*clean*), modificadas (*dirty*), bloqueadas (*locked*), etc.

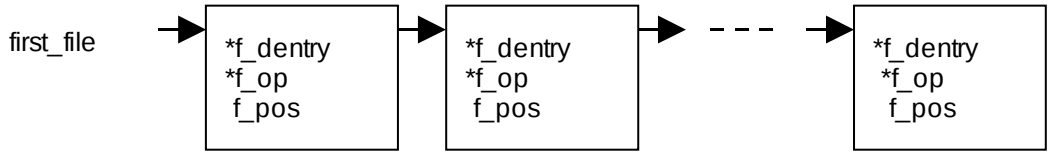
Por cada operación de lectura, el subsistema de *buffer-cache* debe buscar si el bloque en cuestión ya está copiado en la memoria. Para hacerlo de forma eficiente se mantienen tablas de dispersión para todos los *buffers*

presentes en la caché. El *buffer-cache* es también usado para mejorar las operaciones de escritura, en vez de copiar todos los bytes inmediatamente al disco, el núcleo los almacena temporalmente en el *buffer-cache* e intenta agrupar varias operaciones para escribirlas al disco simultáneamente. Un *buffer* que está esperando por ser copiado al disco se denomina *modificado* (*sucio* o *dirty*).



15.9 Estructura file

Cuando un proceso de usuario abre un fichero, el VFS, crea una estructura **file**, definida en include/linux/fs.h, estas estructuras se encadenan en una lista global apuntada por la variable **first_file**.




```
struct file {
 839        /*
 840         * fu_list becomes invalid after file_free is called and queued via
 841         * fu_rcuhead for RCU freeing
 842         */
 843        union {
 844                struct list_head        fu_list;
 845                struct rcu_head         fu_rcuhead;
```

```
 846        } f_u;
 847        struct path             f_path;
 848#define f_dentry        f_path.dentry
 849#define f_vfsmnt        f_path.mnt
 850        const struct file_operations    *f_op;
 851        atomic_long_t           f_count;
 852        unsigned int            f_flags;
 853        fmode_t                 f_mode;
 854        loff_t                  f_pos;
 855        struct fown_struct      f_owner;
 856        const struct cred       *f_cred;
 857        struct file_ra_state    f_ra;
 858
 859        u64                     f_version;
 860#ifdef CONFIG_SECURITY
 861        void                    *f_security;
 862#endif
 863        /* needed for tty driver, and maybe others */
 864        void                    *private_data;
 865
 866#ifdef CONFIG_EPOLL
 867        /* Used by fs/eventpoll.c to link all the hooks to this file */
 868        struct list_head        f_ep_links;
 869        spinlock_t              f_ep_lock;
 870#endif /* #ifdef CONFIG_EPOLL */
 871        struct address_space    *f_mapping;
 872#ifdef CONFIG_DEBUG_WRITECOUNT
 873        unsigned long f_mnt_write_state;
 874#endif
 875};
```

f_path: Estructura path que representa la ruta del fichero.
f_path.dentry: Estructura dentry del path del archivo.
f_op: Puntero a la estructura file_operations.
f_mode: Modo de apertura del fichero.
f_pos: Desplazamiento dentro del fichero.
f_owner: Propietario del fichero.
f_mapping: Contine lasVMA del fichero.

15.9.1 Operaciones sobre ficheros abiertos

Esta estructura tiene un campo denominado f_op, con punteros a funciones que trabajan con ficheros mediante la estructura **file** y son propias y se implementan en el sistema de ficheros real.

```
1310 struct file_operations {
1311        struct module *owner;
```

```
1312        loff_t (*llseek) (struct file *, loff_t, int);
1313        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1314        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1315        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned
long, loff_t);
1316        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
unsigned long, loff_t);
1317        int (*readdir) (struct file *, void *, filldir_t);
1318        unsigned int (*poll) (struct file *, struct poll_table_struct *);
1319        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
1320        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
long);
1321        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1322        int (*mmap) (struct file *, struct vm_area_struct *);
1323        int (*open) (struct inode *, struct file *);
1324        int (*flush) (struct file *, fl_owner_t id);
1325        int (*release) (struct inode *, struct file *);
1326        int (*fsync) (struct file *, struct dentry *, int datasync);
1327        int (*aio_fsync) (struct kiocb *, int datasync);
1328        int (*fasync) (int, struct file *, int);
1329        int (*lock) (struct file *, int, struct file_lock *);
1330        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t
*, int);
1331        unsigned long (*get_unmapped_area)(struct file *, unsigned
long, unsigned long, unsigned long, unsigned long);
1332        int (*check_flags)(int);
1333        int (*flock) (struct file *, int, struct file_lock *);
1334        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
loff_t *, size_t, unsigned int);
1335        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info
*, size_t, unsigned int);
1336        int (*setlease)(struct file *, long, struct file_lock **);
1337};
```

Existen otro conjunto de funciones:

internas o auxiliaries en fs/file_table.c	gestion de nombras en fs/namei
insert_file_free	getname
remove_file_free	putname
put_last_free	dir_namei
grow_files	namei
file_table_init	lnamei
get_empty_filp	

Todo proceso tiene un campo files en la tabla de procesos task_struct, que apunta a una tabla de descriptores de fichero (df) abiertos, esta tabla

tiene la estructura files_struct y se encuentra definida en /include/linux/sched.h, con los siguientes campos:

files_struct

count – número de descriptores de ficheros asociados.
close_on_exec – indicadores de “close_on_exec” como una cadena de bits.
fd – puntero a los descriptores de fichero abiertos.x