Lección 19: OPEN, CREATE Y CLOSE v2.6.29

19.1 Introducción

Un *fichero* es una abstracción muy importante en programación. Los ficheros sirven para almacenar datos de forma permanente y ofrecen un pequeño conjunto de primitivas muy potentes (abrir, leer, avanzar puntero, cerrar, etc.). El sistema de ficheros se organizan normalmente en estructuras de árbol, donde los nodos intermedios son directorios capaces de agrupar otros ficheros.

El sistema de ficheros es la forma en que el sistema operativo organiza, gestiona y mantiene la jerarquía de ficheros en los dispositivos de almacenamiento, normalmente discos duros. Cada sistema operativo soporta diferentes sistemas de ficheros. Para mantener la modularización del sistema operativo y proveer a las aplicaciones con una interfaz de programación (API) uniforme, los diferentes sistemas operativos implementan una capa superior de abstracción denominada *Sistema de Ficheros Virtual* (*VFS: Virtual File System*). Esta capa de software implementa las funcionalidades comunes de los diversos sistemas de ficheros implementados en la capa inferior.

En este capítulo describimos las operaciones de entrada/salida para el manejo del sistema de archivos como abrir ficheros y crearlos, además de manejarlos para poder leer de ellos y escribir. Este apartado se centra en la apertura de fichero, en su creación y en su cierre. Explicaremos como funciona las llamadas al sistema open(), create() y close(). No obstante, primero nos centraremos en las estructuras de datos que manejan las funciones, y luego pasaremos a explicar el funcionamiento de dichas funciones citadas.

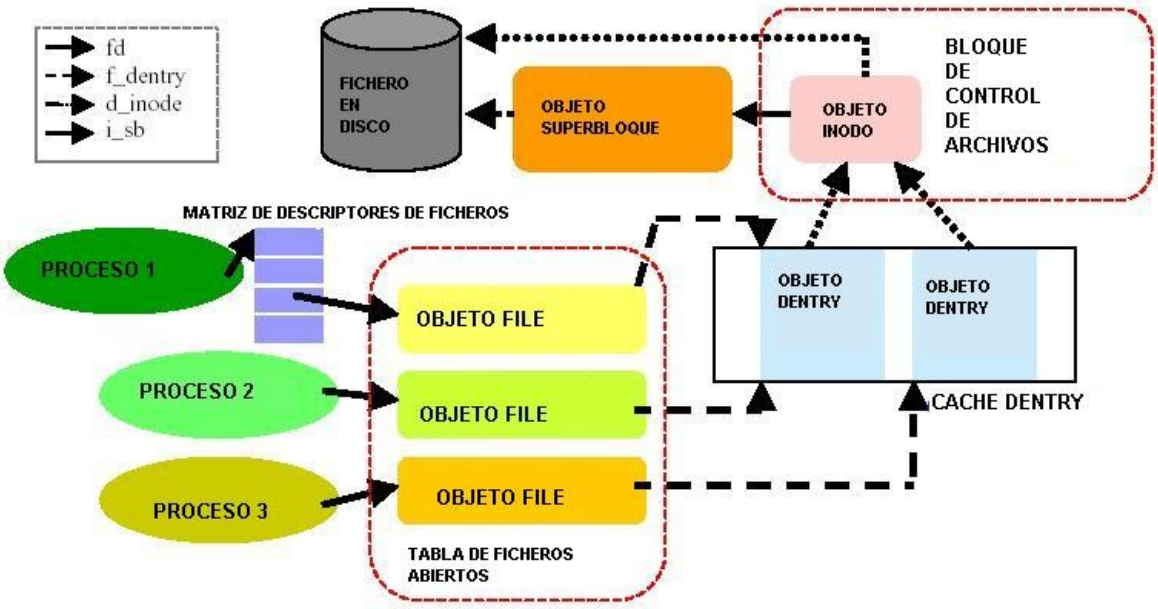
19.2 Estructuras

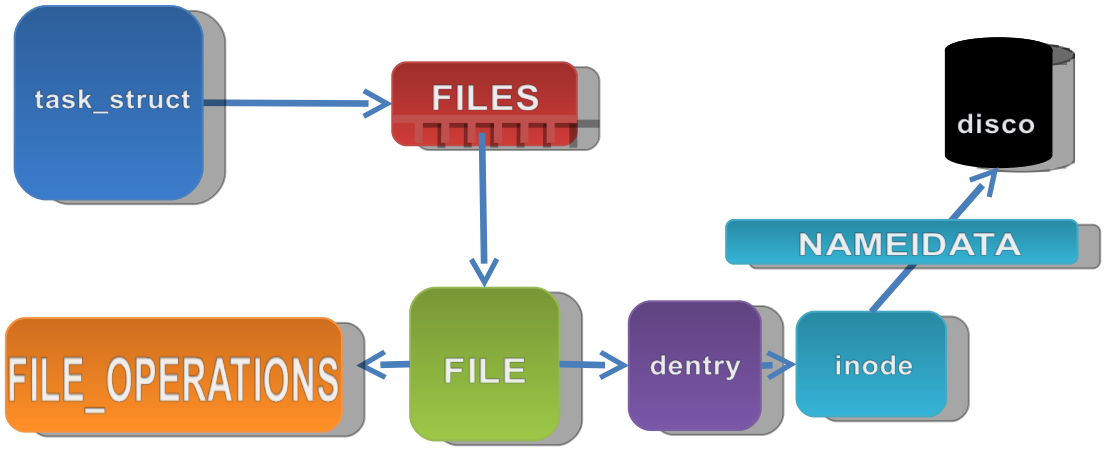
El modelo general de ficheros puede ser interpretado como orientado a objetos, donde los objetos son construcciones de software (estructura de datos y funciones y métodos asociados) de los siguientes tipos:

- **Estructura file:** mantiene la información relacionada a la interacción de un fichero abierto y un proceso. Este objeto existe sólo cuando un proceso mantiene abierto un fichero.
- **Estructura files:** es una estructura dentro de un campo de trask_struct, contiene la lista de archivos abiertos de un proceso y vincula los descriptores asignados por ese proceso con la estructura file correspondiente.

- **Estructura superbloque:** almacena información concerniente al sistema de archivos montado. Para sistemas de archivos basados en discos, este objeto corresponden normalmente al *"bloque del control de sistemas de archivos"* en disco.
- **Estructura inodo:** mantiene información relacionada a un fichero individual: propietario, grupo, fecha y hora de creación, modificación…
- **Estructura dentry:** enlaza una entrada de directorio (pathname) con su fichero correspondiente: inodo + ruta para encontrarlo.
- **Estructura nameidata**: describe la ubicación de un fichero en disco. De modo que se usa como una estructura intermediaria para posteriormente completar los campos de la estructura file.

En el siguiente esquema se puede apreciar cómo varios procesos interactúan con archivos. Cada proceso crea su propia estructura *file*, donde a su vez ésta referencia a la estructura *dentry* que se localiza en *dentry cache*. Como uno de los procesos abrió un enlace, se crea un objeto dentry diferente y ambos objetos dentry apuntan al mismo *inodo*, el cual identifica el superbloque y el archivo en disco.





Estructura file

Una estructura *file* describe cómo un proceso interacciona con el archivo que ha abierto. El objeto se crea cuando el archivo se abre y consiste en una estructura *file*, donde sus campos se apreciarán a continuación. Estos objetos no tienen correspondencia en disco, por tanto no tienen un campo que indique que ha sido modificado.

Campo	**Descripción**
fu_list	Puntero a la lista de objetos
f_dentry	Puntero al objeto dentry asociado
f_op	Puntero a la tabla de operaciones sobre archivos
f_mode	Modo de acceso del proceso
f_pos	Posición dentro del archivo
f_count	Contador de uso del objeto FILE
f_flags	Indicadores especificados en la apertura del archivo
f_owner	Datos para E/S asíncronas mediante señales
f_version	Número de versión, aumentado automáticamente después de cada uso
private_data	Necesario para manejadores tty

El campo f_pos almacena el *puntero a archivo*, es decir, la posición actual dentro del archivo donde tendrá lugar la siguiente operación. Dado

que varios procesos pueden acceder al mismo fichero concurrentemente, el puntero a archivo no puede mantenerse en el objeto i*nodo*.

La implementación de la estructura file se muestra a continuación, la cual se encuentra ubicada en la página linux/include/linux/fs.h

```
 838struct 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};
```

Cada sistema de ficheros incluye su propio conjunto de operaciones de archivo, que realizan actividades tales como leer y escribir un archivo,

etc. Cuando el núcleo carga un i*nodo* en memoria desde disco, almacena un puntero a esas operaciones de archivo en la estructura *file_operations*, la cual se encuentra ubicada en linux/include/linux/fs.h, cuya dirección está contenida en el campo *default_file_ops* de la estructura *inode_operations* del objeto i*nodo*. Cuando un proceso abre un archivo, el sistema de fichero inicializa el campo f_op del nuevo objeto con la dirección almacenada en el i*nodo,* de forma que las llamadas posteriores a las operaciones de archivo pueden usar estas funciones.

```
1310struct 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};
```

Estructura files

Esta tabla especifica qué archivos están actualmente abiertos por el proceso. Dicha información se almacena en la estructura task_struct en el campo files del descriptor del proceso. Un proceso no puede tener más de NR_OPEN (normalmente 1024) descriptores de archivos. Es posible definir un límite menor dinámico al máximo de archivos abiertos cambiando rlim[RLIMIT_NOFILE] en el descriptor del proceso.

La implementación de dicha estructura se muestra a continuación, y está ubicada en /include/linux/fdtable.h

```
 41struct files_struct {
 42  /*
 43   * read mostly part
 44   */
 45        atomic_t count;
 46        struct fdtable *fdt;
 47        struct fdtable fdtab;
 48  /*
 49   * written part on a separate cache line in SMP
 50   */
 51        spinlock_t file_lock ____cacheline_aligned_in_smp;
 52        int next_fd;
 53        struct embedded_fd_set close_on_exec_init;
 54        struct embedded_fd_set open_fds_init;
 55        struct file * fd_array[NR_OPEN_DEFAULT];
 56};
```

El significado de cada uno de los campos anteriores se explican en la siguiente tabla:

Campo	Descripcion
count	Número de procesos que comparten esta tabla
fdt	Puntero a la tabla de descriptores de ficheros file
fdtab	Matriz de de objetos file
next_fd	Descriptores de archivos max. asignados más 1
fd	Puntero a la matriz de puntero de objetos file
close_on_exec_init	Conjunto inicial de descriptores a cerrar con exec()
open_fds_init	Conjunto inicial de descriptores de archivos
fd_array[32]	vector de punteros a objetos file

El campo fd_array es un vector de punteros a estructuras de archivo de tipo *file*. El tamaño de dicho vector se almacena en el campo max_fds y, normalmente fd apunta al campo fd_array, que incluye 32 punteros a objetos file. Si el proceso abre más de 32 archivos, el núcleo asigna un nuevo vector mayor de punteros a archivos y almacena su dirección en el campo fd. No obstante, también actualiza el campo max_fds.

Para cada fichero con una entrada en el vector fd_array, el índice del vector es el *descriptor de fichero*. Normalmente, el primer elemento (índice 0) del vector se asocia con la entrada estándar del proceso el teclado, el segundo con la salida estándar la pantalla, y el tercero con el error estándar. Los procesos Unix utilizan el descriptor de fichero como el principal identificador de archivo.

Observar que, debido a las llamadas al sistema dup(), dup2(), y fcntl(), dos descriptores de ficheros pueden referenciar al mismo archivo abierto, es decir, dos elementos del vector pueden apuntar al mismo objeto archivo.

El campo open_fds contiene la dirección del campo open_fds_init, que es un mapa de bits que identifica los descriptores de archivo de los archivos abiertos actualmente. El campo max_fdset almacena el número de bits en el mapa de bits. Como la estructura de datos fd_set incluye 1024 bits, no es necesario expandir el tamaño del mapa de bits.

Sin embargo, el núcleo puede expandir dinámicamente el tamaño del mapa de bits si esto resulta necesario. El núcleo suministra una función fget() que se invoca cuando se empieza a utilizar un objeto archivo. Esta función recibe un descriptor de archivo, fd, como parámetro. Devuelve la dirección en current->files->fd[fd], es decir, la dirección del objeto archivo correspondiente, o NULL si no hay archivo que se corresponda con fd. En el primer caso, fget() incrementa en uno el contador de uso del objeto, f_count.

No obstante, el núcleo también suministra una función fput() que se invoca cuando un camino de control del núcleo termina de utilizar el objeto archivo. Esta función recibe como parámetro la dirección del objeto archivo y decrementa el contador de uso. Además, si el contador alcanza el valor cero, invoca a la función release de las operaciones de archivo, libera el objeto dentro asociado y decrementa el campo i_writeaccess del objeto i*nodo* (si el archivo estaba abierto para escritura), y finalmente mueve el objeto archivo de la lista en uso a la lista no usado.

Estructura dentry

Una vez que un directorio ha sido leído en memoria, el sistema de ficheros lo transforma en una estructura *dentry*, cuyos campos se describen a continuación. El núcleo crea una estructura *dentry* para cada componente de un camino de archivo al que accede un proceso y asocia la estructura con su correspondiente i*nodo*. Por ejemplo, cuando buscamos el camino /tmp/prueba, el núcleo crea una estructura *dentry* para el directorio raíz (/), un segundo para el directorio *tmp*, y un tercero dentro para el subdirectorio *prueba*.

Campo	Descripcion
d_count	Contador de uso del objeto dentry
d_flags	Indicadores dentry
d_inode	Inodo asociado con el nombre de archivo

d_parent	Objeto dentry del directorio padre
d_mounts	Para un punto de montaje, dentro del raíz del sistema montado
d_covers	Para el sistema archivos raíz, el raíz del sistema
d_hash	Punteros para la lista en una entrada de la tabla hash
d_lru	Puntero para lista de “no en uso”
d_child	Punteros para la lista de objetos dentry incluidos en el directorio padre
d_subdirs	Para directorios, lista de objetos dentry de los subdirectorios
d_alias	Lista de Inodos asociados (alias)
d_name	Nombre de archivo
d_time	Utilizado por el método d_revalidate
d_op	Métodos dentro
d_sb	Objeto superbloque del archivo
d_reftime	Tiempo de cuando fue descargado el objeto dentry
d_fsdata	Datos dependientes del sistema de archivos
d_iname[16]	Espacio para nombres cortos

Observar que las estructuras *dentry* no tiene su correspondiente imagen en disco, y por tanto no se incluye un campo en la estructura *dentry* para especificar que ha sido modificado. Las estructuras *dentry* se almacenan en una caché, denominada *dentry*_cahe.

```
  89struct 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};


 134struct dentry_operations {
 135        int (*d_revalidate)(struct dentry *, struct nameidata *);
 136        int (*d_hash) (struct dentry *, struct qstr *);
 137        int (*d_compare) (struct dentry *, struct qstr *, struct qstr
*);
 138        int (*d_delete)(struct dentry *);
 139        void (*d_release)(struct dentry *);
 140        void (*d_iput)(struct dentry *, struct inode *);
 141        char *(*d_dname)(struct dentry *, char *, int);
 142};
```

La dentry cache.

Dado que leer una entrada de directorio desde disco y construir el correspondiente *dentry* requiere un tiempo considerable, tiene sentido mantener en memoria estos objetos que aunque hayamos finalizado con ellos, pueden servir con posterioridad. Por ejemplo, podemos editar un archivo y compilarlo, y volverlo a editar, hacer una copia de él, etc. En estos casos, el mismo archivo es accedido repetidamente.

Para maximizar la eficiencia en la gestión de entradas de directorio, Linux utiliza una *caché dentry* que consiste en dos clases de estructuras de datos:

• Un conjunto de objetos dentro de los estados en uso, no usado, o negativo.
• Una tabla hash para derivar rápidamente el objeto *dentry* asociado con un nombre de archivo dado y un directorio dado. En caso de que los objetos requeridos no están incluidos en la caché, la función hash devuelve un valor nulo.

La *caché dentry* funciona como un controlador de la *caché de inodos*. Los i*nodo*s en memoria núcleo que están asociados con entradas de directorio no usadas no se descargan, dado que la *caché dentry* las está

utilizando y, por tanto, los campos i_count no son nulos. Así, los objetos i*nodo* se mantienen en RAM y pueden referenciarse con rapidez mediante sus correspondientes entradas *dentry*.

En el fichero linux/include/Linux/dcache.h encontramos la estructura dentry.

Estructura nameidata

Se encuentra definida en el fichero: /incluye/linux/namei.h

Se utiliza como estructura intermedia para realizar las inicializaciones pertinentes a la hora de crear o abrir un fichero.

```
 18struct nameidata {
 19        struct path     path;
 20        struct qstr     last;
 21        unsigned int    flags;
 22        int             last_type;
 23        unsigned        depth;
 24        char *saved_names[MAX_NESTED_LINKS + 1];
 25
 26        /* Intent data */
 27        union {
 28                struct open_intent open;
 29        } intent;
 30};
```

Estructura inode

Se encuentra definida en el fichero: /incluye/linux/fs.h

Todo fichero o directorio está asociado a su inode correspondiente. El inode estará ubicado siempre en disco y en memoria principal cuando está siendo accedido. Cada inode contiene información sobre el sistema de ficheros real, ubicación del fichero en disco, e información del fichero.

Los inode en memoria se organizan en listas en el *inode cache*, para un acceso rápido. Todos los inodos se combinan en una lista global doblemente encadenada. Mediante una tabla hash se accede a listas con los inodos que tienen el mismo valor de hash, este valor se obtiene con el número de dispositivo más número de inode.

Si un inode se necesita y no está en la caché se busca en el disco y se almacena en la caché. La política de gestión de la caché se realiza mediante el uso del atributo count de cada inode. Los inodes modificados (dirty) tienen que escribirse en disco. Las funciones de lectura/escritura de inodes en disco son específicas de cada sistema de ficheros, estas funciones se encuentran en el struct “inode_operations”.

```
 649struct 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};
```

19.3 Apertura de un archivo

La llamada al sistema open() se utiliza para a partir de una ruta/nombre de fichero obtener un descriptor de fichero (fd), que se utilizará en las operaciones de E/S posteriores como en read, write, etc.

fd = open(pathname, flags, mode)

Los parámetros que presenta la función open() son los siguientes:

Pathname: nombre del fichero a abrir.

Flags: define el modo de apertura.

Mode: especifica los permisos del usuario.

El modo de apertura, **flags**, establece la forma en que se va a trabajar con el fichero. Las constantes que definen los modos se encuentran en el archivo de cabecera <fcntl.h>:

O_RDONLY: sólo lectura.

O_WRONLY: sólo escritura.

O_RDWR: lectura y escritura.

O_CREAT: crea el fichero y lo abre. Si existía lo sobrescribe.

O_EXCL: si el fichero ya existía retorna un error.

O_APPEND: empieza a escribir al final del fichero.

O_TRUNC: abre el fichero y trunca su longitud a cero.

O_NONBLOCK: en modo compartido.

O_SYNC: modo síncrono. Toda actualización se escribe inmediatamente en disco.

Los permisos de usuario-grupo-otros, **mode**, son macros definidas que se corresponden con un número octal de tres dígitos 0xxx:

0[rwx][rwx][rwx]

El primero indica permisos para usuarios.

El segundo indica permisos para grupos.

El tercero para otros.

Con la instrucción anterior se realiza la llamada al sistema de open (número 5 para sys_call), y a continuación se conmuta de modo usuario a modo núcleo mediante las tablas IDT y GDT.

La función del núcleo que implementa la llamada open es **sys_open**. Esta función se define como una macro SYSCALL_DEFINE3, que nos lleva a sys_open, definida en el fichero include/linux/syscall.h.

```
1049 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags,
int, mode)
1050{
1051            long ret;
1052
1053            if (force_o_largefile())  //Comprueba si es un fichero
largo
1054                    flags |= O_LARGEFILE;  //Si lo es asigna flgas

1055            //Llamada a do_sys_open
1056            ret = do_sys_open(AT_FDCWD, filename, flags, mode);
1057            /* avoid REGPARM breakage on x86: */
1058            asmlinkage_protect(3, ret, filename, flags, mode);
1059            return ret;
1060}
```

Esta función probará primero si el fichero abierto es un fichero grande y entonces a flags se le añadirá el bit de “O_LARGEFILE”, llamando a la función **do_sys_open**, con los parámetros de entrada del sys_open, los flags en caso de que sea fichero largo y la macro AT_FDCWD que indica que la apertura ha de realizarse en el directorio actual.

Si la llamada tiene éxito devolverá un descriptor del fichero y en otro caso devuelve -1.

```
1027long do_sys_open(int dfd, const char __user *filename, int flags, int
mode)
1028{
1029        char *tmp = getname(filename); /* Obtiene memoria en el
núcleo y pasa el nombre del fichero al núcleo. */
1030        int fd = PTR_ERR(tmp);
1031
1032        if (!IS_ERR(tmp)) {
1033                fd = get_unused_fd_flags(flags); /* Obtiene un
descriptor libre para ese fichero. */
1034                if (fd >= 0) {
1035                        struct file *f = do_filp_open(dfd, tmp,
flags, mode); /* Abre o crea el fichero. */
1036                        if (IS_ERR(f)) {
1037                                put_unused_fd(fd); /* Si se produce un
error en la apertura libera el descriptor del fichero.*/
1038                                fd = PTR_ERR(f);
1039                        } else {
1040                                fsnotify_open(f->f_path.dentry); /*
Comprueba que el archivo este abierto. */
1041                                fd_install(fd, f); /* Vincula el
descriptor de fichero con la lista de ficheros.*/
1042                        }
1043                }
1044                putname(tmp);
1045        }
1046        return fd;
1047}
```

Dentro de **do_sys_open()** veamos las partes destacadas:

La primera es la invocación de **getname(),** la cual lee el nombre del archivo, lo copia en la memoria del núcleo y devuelve la dirección de memoria.

Cuando ha copiado en memoria del núcleo el nombre de fichero y sabe que no hay error, invoca a **get_unused_fd()**, dicha función realiza la búsqueda de una posición libre en el vector de descriptores de ficheros (files), esta posición determinara el descriptor fd.

Cuando consigue la posición libre llama a **do_filp_open()** que hace todo el proceso de apertura del fichero además de crear su estructura file correspondiente.

Llama a la función **fsnotify_open()**, notifica en el inode que se ha abierto un fichero.

Llama a la función **fd_install()**, para introducir la nueva estructura file en la tabla y actualiza el puntero a la tabla.

Si se crea bien la estructura fichero, **putname()** libera la memoria del núcleo en donde alojamos el pathname.

Devuelve el descriptor de fichero que le ha asignado **return fd**.

Ahora profundizamos un poco más en las funciónes que utiliza **do_sys_open()**.

Función do_getname()

La función **do_getname()** realiza la copia de dicha ristra definida por el usuario y la aloja en el espacio de memoria que hemos obtenido en **getname()**.

/fs/namei.c

```
 140char * getname(const char __user * filename)
 141{
 142        char *tmp, *result;
 143
 144        result = ERR_PTR(-ENOMEM);
 145        tmp = __getname();/* Asigna memoria en el núcleo para el
nombre del fichero */
 146        if (tmp)  {
/*Copia el nombre del fichero desde el espacio del usuario al espacio del
núcleo*/
 147                int retval = do_getname(filename, tmp);
 148
 149                result = tmp;
 150                if (retval < 0) {/* Si algo ha ido mal */
 151                        __putname(tmp); /* libera el espacio de
memoria del núcleo */
 152                        result = ERR_PTR(retval); /* y devuelve el
código de error del fallo. */
 153                }
 154        }
 155        audit_getname(result);
 156        return result; /*Si todo ha ido bien, devuelve un puntero al
nuevo fichero. */
 157}
 118static int do_getname(const char __user *filename, char *page)
 119{
 120        int retval;
 121        unsigned long len = PATH_MAX;
 122         /* Verifica si cabe en memoria. */
 123        if (!segment_eq(get_fs(), KERNEL_DS)) {
 124                if ((unsigned long) filename >= TASK_SIZE)
 125                        return -EFAULT;
 126                if (TASK_SIZE - (unsigned long) filename < PATH_MAX)
 127                        len = TASK_SIZE - (unsigned long) filename;
 128        }
 129
 130        retval = strncpy_from_user(page, filename, len); /* Realiza
la copia. */
```

```
 131        if (retval > 0) {
 132                if (retval < len)
 133                        return 0;
 134                return -ENAMETOOLONG;
 135        } else if (!retval)
 136                retval = -ENOENT;
 137        return retval;
 138}
```

Volvemos a la función **do_sys_open()** y ahora comprueba que la dirección de memoria devuelta por **getname()** no es errónea.

Función get_unused_fd_flags()

Luego localiza un descriptor de fichero libre con **get_unused_fd_flags().** La función realiza la búsqueda de una ranura vacía en *current->files->fd*, lo marca como usado y define el siguiente descriptor de fichero que esté libre y lo devuelve si todo ha ocurrido bien. Y finalmente invoca **find_next_zero_bit()** para encontrar una ranura vacía en *current->files->fd*.

```
  39#define get_unused_fd_flags(flags) alloc_fd(0, (flags))

 437/*
 438 * allocate a file descriptor, mark it busy.
 439 */
 440int alloc_fd(unsigned start, unsigned flags)
 441{
 442        struct files_struct *files = current->files;
 443        unsigned int fd;
 444        int error;
 445        struct fdtable *fdt;
 446
 447        spin_lock(&files->file_lock); /* Comienza la exclusión mutua.
*/
 448repeat: /* Busca el primer descriptor de fichero libre. */
 449        fdt = files_fdtable(files); /* Busca un descriptor. */
 450        fd = start;
 451        if (fd < files->next_fd)
 452                fd = files->next_fd;
 453
 454        if (fd < fdt->max_fds)
 455                fd = find_next_zero_bit(fdt->open_fds->fds_bits,
 456                                        fdt->max_fds, fd); /*Si se
ha superado el límite, no ha encontrado ningún descriptor y va a la
etiqueta out*/

 457       /* Mira si se puede expandir el vector de descriptores. */
 458        error = expand_files(files, fd);
 459        if (error < 0) /* Ha ocurrido un error, así que va a out con
el código en error. */
```

```
 460                goto out;
 461
 462        /*
 463         * If we needed to expand the fs array we
 464         * might have blocked - try again.
 465         */
 466        if (error) /* Si error vale 1, se ha podido expandir. */
 467                goto repeat; /* Seguimos con la búsqueda del
descriptor. */
 468
 469        if (start <= files->next_fd)
 470                files->next_fd = fd + 1;
 471
 472        FD_SET(fd, fdt->open_fds); /* Encuentra el descriptor y lo
marca como usado*/
 473        if (flags & O_CLOEXEC)
 474                FD_SET(fd, fdt->close_on_exec);
 475        else
 476                FD_CLR(fd, fdt->close_on_exec);
 477        error = fd;
 478#if 1
 479        /* Sanity check */
 480        if (rcu_dereference(fdt->fd[fd]) != NULL) { /* Comprobación
de que el descriptor es correcto*/
 481                printk(KERN_WARNING "alloc_fd: slot %d not NULL!\n",
fd);
 482                rcu_assign_pointer(fdt->fd[fd], NULL);
 483        }
 484#endif
 485
 486out:
 487        spin_unlock(&files->file_lock);
 488        return error; /* Fin de la exclusión mutua y retorna error o
el descriptor. */
 489}
 490
 491int get_unused_fd(void)
 492{
 493        return alloc_fd(0, 0);
 494}
```

Función do_filp_open()

Cuando ha obtenido el nuevo descriptor de fichero, lo almacena en una variable local “fd” y comprueba si no ha habido ningún error, y si puede seguir llamando a **do_filp_open.** La función **do_filp_open()** se le pasa como parámetros el “pathname”, el modo de acceso, y los permisos. Esta función, ejecuta los pasos siguientes antes de llamar a **open_namei().** Copia el registro de modo de acceso en “namei_flags”, pero codificado con las banderas O_RDONLY, O_WRONLY, y O_RDWR (que es la macro O_ACCMODE) en un formato especial: el primer bit de “namei_flags” se fijará a cero solamente si el acceso del archivo requiere

privilegios de lectura; de otra forma, el bit se fijará a uno si el acceso del archivo requiere privilegios de escritura. Nótese que no es posible especificar en la llamada del sistema del **open()** que un acceso del archivo no requiere permisos de lectura ni de escritura, es decir, en una operación de búsqueda del pathname que implica links.

```
1625/*
1626 * Note that the low bits of the passed in "open_flag"
1627 * are not the same as in the local variable "flag". See
1628 * open_to_namei_flags() for more details.
1629 */
1630struct file *do_filp_open(int dfd, const char *pathname,
1631                int open_flag, int mode)
1632{
1633        struct file *filp; //Estructura filp que se devulve
1634        struct nameidata nd;
1635        int acc_mode, error;
1636        struct path path;
1637        struct dentry *dir; //Estructura dentry donde se encuentre el
inode asociado
1638        int count = 0;
1639        int will_write;
1640        int flag = open_to_namei_flags(open_flag); //Se copian las
flags
1641
1642        acc_mode = MAY_OPEN | ACC_MODE(flag);
1643
1644        /* O_TRUNC implies we need access checks for write
permissions */
1645        if (flag & O_TRUNC)
1646                acc_mode |= MAY_WRITE;
1647
1648        /* Allow the LSM permission hook to distinguish append
1649           access from general write access. */
1650        if (flag & O_APPEND)
1651                acc_mode |= MAY_APPEND;
1652
1653        /*
1654         * Si sólo hay que abrir el fichero se llama a la función y
se especificara que no hace falta crear el fichero y Devuelve el inode
correspondiente al fichero. */
1655         */
1656        if (!(flag & O_CREAT)) {
1657                error = path_lookup_open(dfd, pathname,
lookup_flags(flag), &nd, flag);
1659                if (error)
1660                        return ERR_PTR(error);
1661                goto ok;
1662        }
1663
1664        /*
1665         * Si se crea el fichero llamada a la función que lo crea y
rellena los campos y se devuelve el inode
1666         */
1667        error = do_path_lookup(dfd, pathname, LOOKUP_PARENT, &nd);
```

```
1668        if (error)
1669                return ERR_PTR(error);
1670
1671        /*
1672         * We have the parent and last component. First of all, check
1673         * that we are not asked to creat(2) an obvious directory -
that
1674         * will not do.
1675         */
1676        error = -EISDIR;
1677        if (nd.last_type != LAST_NORM || nd.last.name[nd.last.len])
1678                goto exit_parent;
1679
1680        error = -ENFILE;
1681        filp = get_empty_filp();
1682        if (filp == NULL)
1683                goto exit_parent;
1684        nd.intent.open.file = filp;
1685        nd.intent.open.flags = flag;
1686        nd.intent.open.create_mode = mode;
1687        dir = nd.path.dentry;
1688        nd.flags &= ~LOOKUP_PARENT;
1689        nd.flags |= LOOKUP_CREATE | LOOKUP_OPEN;
1690        if (flag & O_EXCL)
1691                nd.flags |= LOOKUP_EXCL;
1692        mutex_lock(&dir->d_inode->i_mutex);
1693        path.dentry = lookup_hash(&nd);
1694        path.mnt = nd.path.mnt;
1695
1696do_last:
1697        error = PTR_ERR(path.dentry);
1698        if (IS_ERR(path.dentry)) {
1699                mutex_unlock(&dir->d_inode->i_mutex);
1700                goto exit;
1701        }
1702
1703        if (IS_ERR(nd.intent.open.file)) {
1704                error = PTR_ERR(nd.intent.open.file);
1705                goto exit_mutex_unlock;
1706        }
1707
1708        /* Se debe realizar la Creación del fichero */
1709        if (!path.dentry->d_inode) {
1710                /*
1711                 * This write is needed to ensure that a
1712                 * ro->rw transition does not occur between
1713                 * the time when the file is created and when
1714                 * a permanent write count is taken through
1715                 * the 'struct file' in nameidata_to_filp().
1716                 */
1717                error = mnt_want_write(nd.path.mnt);
1718                if (error)
1719                        goto exit_mutex_unlock;
1720                error = __open_namei_create(&nd, &path, flag, mode);
1721                if (error) {
1722                        mnt_drop_write(nd.path.mnt);
1723                        goto exit;
```

```
1724                }
1725                filp = nameidata_to_filp(&nd, open_flag); /* Realiza
la apertura del fichero después de crearlo. */
1726                mnt_drop_write(nd.path.mnt);
1727                return filp; /*devuelve la estructura FILE
1728        }
1729
1730        /*
1731         * It already exists.
1732         */
1733        mutex_unlock(&dir->d_inode->i_mutex);
1734        audit_inode(pathname, path.dentry);
1735
1736        error = -EEXIST;
1737        if (flag & O_EXCL)
1738                goto exit_dput;
1739
1740        if (__follow_mount(&path)) {
1741                error = -ELOOP;
1742                if (flag & O_NOFOLLOW)
1743                        goto exit_dput;
1744        }
1745
1746        error = -ENOENT;
1747        if (!path.dentry->d_inode)
1748                goto exit_dput;
1749        if (path.dentry->d_inode->i_op->follow_link)
1750                goto do_link;
1751
1752        path_to_nameidata(&path, &nd);
1753        error = -EISDIR;
1754        if (path.dentry->d_inode && S_ISDIR(path.dentry->d_inode-
>i_mode))
1755                goto exit;
1756ok:
1757        /*
1758         * Consider:
1759         * 1. may_open() truncates a file
1760         * 2. a rw->ro mount transition occurs
1761         * 3. nameidata_to_filp() fails due to
1762         *    the ro mount.
1763         * That would be inconsistent, and should
1764         * be avoided. Taking this mnt write here
1765         * ensures that (2) can not occur.
1766         */
1767        will_write = open_will_write_to_fs(flag, nd.path.dentry-
>d_inode);
1768        if (will_write) {
1769                error = mnt_want_write(nd.path.mnt);
1770                if (error)
1771                        goto exit;
1772        }
1773        error = may_open(&nd.path, acc_mode, flag);
1774        if (error) {
1775                if (will_write)
1776                        mnt_drop_write(nd.path.mnt);
1777                goto exit;
```

```
1778        }
1779        filp = nameidata_to_filp(&nd, open_flag); /* Realiza la
apertura del fichero sin que se haya creado porque ya existia */
1780        /*
1781         * It is now safe to drop the mnt write
1782         * because the filp has had a write taken
1783         * on its behalf.
1784         */
1785        if (will_write)
1786                mnt_drop_write(nd.path.mnt);
1787        return filp; //Devuelve la estructura FILE
1788
1789exit_mutex_unlock:
1790        mutex_unlock(&dir->d_inode->i_mutex);
1791exit_dput:
1792        path_put_conditional(&path, &nd);
1793exit:
1794        if (!IS_ERR(nd.intent.open.file))
1795                release_open_intent(&nd);
1796exit_parent:
1797        path_put(&nd.path);
1798        return ERR_PTR(error);
1799
1800do_link:
1801        error = -ELOOP;
1802        if (flag & O_NOFOLLOW)
1803                goto exit_dput;
1804        /*
1805         * This is subtle. Instead of calling do_follow_link() we do
the
1806         * thing by hands. The reason is that this way we have zero
link_count
1807         * and path_walk() (called from ->follow_link) honoring
LOOKUP_PARENT.
1808         * After that we have the parent and last component, i.e.
1809         * we are in the same situation as after the first
path_walk().
1810         * Well, almost - if the last component is normal we get its
copy
1811         * stored in nd->last.name and we will have to putname() it
when we
1812         * are done. Procfs-like symlinks just set LAST_BIND.
1813         */
1814        nd.flags |= LOOKUP_PARENT;
1815        error = security_inode_follow_link(path.dentry, &nd);
1816        if (error)
1817                goto exit_dput;
1818        error = __do_follow_link(&path, &nd);
1819        if (error) {
1820                /* Does someone understand code flow here? Or it is
only
1821                 * me so stupid? Anathema to whoever designed this
non-sense
1822                 * with "intent.open".
1823                 */
1824                release_open_intent(&nd);
1825                return ERR_PTR(error);
```

```
1826        }
1827        nd.flags &= ~LOOKUP_PARENT;
1828        if (nd.last_type == LAST_BIND)
1829                goto ok;
1830        error = -EISDIR;
1831        if (nd.last_type != LAST_NORM)
1832                goto exit;
1833        if (nd.last.name[nd.last.len]) {
1834                __putname(nd.last.name);
1835                goto exit;
1836        }
1837        error = -ELOOP;
1838        if (count++==32) {
1839                __putname(nd.last.name);
1840                goto exit;
1841        }
1842        dir = nd.path.dentry;
1843        mutex_lock(&dir->d_inode->i_mutex);
1844        path.dentry = lookup_hash(&nd);
1845        path.mnt = nd.path.mnt;
1846        __putname(nd.last.name);
1847        goto do_last;
1848}
```

Función fsnotify_open()

Devuelve la dirección del objeto file a la función do_sys_open(). Si hay errores en el objeto file se libera espacio en el descriptor de ficheros y se devuelve error si no se llama a **fsnotify_open()** para notificar en el inode que se ha abierto un fichero.

```
/*
 169 * fsnotify_open - file was opened
 170 */
 171static inline void fsnotify_open(struct dentry *dentry)
 172{
 173        struct inode *inode = dentry->d_inode;
 174        u32 mask = IN_OPEN;
 175
 176        if (S_ISDIR(inode->i_mode))
 177                mask |= IN_ISDIR;
 178
 179        inotify_dentry_parent_queue_event(dentry, mask, 0, dentry-
>d_name.name);
 180        inotify_inode_queue_event(inode, mask, 0, NULL, NULL);
 181}
 182
```

Función fd_install()

Se llama a **fd_install()**, que introduce la nueva estructura file en la tabla fdtable y actualiza el puntero a la tabla fdt.

```
1012  void fastcall fd_install(unsigned int fd, struct file * file)
1013  {
1014        struct files_struct *files = current->files;
1015        struct fdtable *fdt;
1016        spin_lock(&files->file_lock);
1017        fdt = files_fdtable(files);
1018        BUG_ON(fdt->fd[fd] != NULL);
1019        rcu_assign_pointer(fdt->fd[fd], file);
1020        spin_unlock(&files->file_lock);
1021  }
```

Función putname()

Putname(), realiza la función contraria a getname, libera espacio en el núcleo.

```
void putname(const char *name)
 161{
 162        if (unlikely(!audit_dummy_context()))
 163                audit_putname(name);
 164        else
 165                __putname(name);
 166}
```

```
#define __putname(name) kmem_cache_free(names_cachep, (void *)(name))
```

```
/**
3685 * kmem_cache_free - Deallocate an object
3686 * @cachep: The cache the allocation was from.
3687 * @objp: The previously allocated object.
3688 *
3689 * Free an object which was previously allocated from this
3690 * cache.
3691 */
3692void kmem_cache_free(struct kmem_cache *cachep, void *objp)
3693{
3694        unsigned long flags;
3695
3696        local_irq_save(flags);
3697        debug_check_no_locks_freed(objp, obj_size(cachep));
```

```
3698        if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
3699                debug_check_no_obj_freed(objp, obj_size(cachep));
3700        __cache_free(cachep, objp);
3701        local_irq_restore(flags);
3702}
```

19.4 FUNCIONES AUXILIARES DE SEGUNDO NIVEL

Función path_lookup_open()

Es llamada dentro de la función do_file_open. Cuando se llama a **path_lookup_open() o directamente a la función do_path_lookup** (), se le pasa como parémtro la opción *create_mode* a cero si es mediante **path_lookup_open()** y sino con la bandera *mode* para crear el fichero, es decir, si el fichero va ha ser abierto no necesita los permisos, si el fichero va a ser creado necesita los permisos para su creación.

```
1129int path_lookup_open(int dfd, const char *name, unsigned int
lookup_flags,
1130                struct nameidata *nd, int open_flags)
1131{
1132        struct file *filp = get_empty_filp();
1133        int err;
1134
1135        if (filp == NULL)
1136                return -ENFILE;
1137        nd->intent.open.file = filp;
1138        nd->intent.open.flags = open_flags;
1139        nd->intent.open.create_mode = 0;
1140        err = do_path_lookup(dfd, name, lookup_flags|LOOKUP_OPEN,
nd);
1141        if (IS_ERR(nd->intent.open.file)) {
1142                if (err == 0) {
1143                        err = PTR_ERR(nd->intent.open.file);
1144                        path_put(&nd->path);
1145                }
1146        } else if (err != 0)
1147                release_open_intent(nd);
1148        return err;
1149}
```

El procedimiento de **do_path_lookup()** consiste en analizar el nombre de camino en una secuencia de nombres de archivos. Todos los nombres de archivos excepto el último deben identificar directorios.

Si el primer carácter del nombre de camino es /, el nombre de camino es absoluto, y la búsqueda comienza desde el directorio identificado por

current->fs->root (el directorio raíz del proceso). En otro caso, el nombre es relativo, y la búsqueda se inicia en el directorio identificado por current->fs->pwd (el directorio actual del proceso).

Teniendo el *inodo* del directorio inicial, el procedimiento do_path_lookup examina la entrada que iguala el primer nombre para derivar el correspondiente *inodo*. Entonces, lee el archivo de directorio que tiene ese *inodo* desde disco y la entrada que iguale el siguiente nombre se examina para derivar al *inodo* correspondiente. Este procedimiento se repite por cada nombre incluido en el camino.

La caché *dentry* acelera considerablemente el procedimiento, ya que mantiene los objetos *dentry* utilizados más recientemente. Como hemos visto antes, cada objeto asocia un nombre de archivo en un directorio específico a su correspondiente *inodo*. En muchos casos, el análisis del nombre de camino puede evitar la lectura de directorios de disco intermedios.

```
1015/* Returns 0 and nd will be valid on success; Retuns error,
otherwise. */
1016static int do_path_lookup(int dfd, const char *name,
1017                                unsigned int flags, struct nameidata
*nd)
1018{
1019        int retval = 0;
1020        int fput_needed;
1021        struct file *file;
1022        struct fs_struct *fs = current->fs;
1023
1024        nd->last_type = LAST_ROOT; /* if there are only slashes... */
1025        nd->flags = flags;
1026        nd->depth = 0;
1027
1028        if (*name=='/') {
1029                read_lock(&fs->lock);
1030                nd->path = fs->root;
1031                path_get(&fs->root);
1032                read_unlock(&fs->lock);
1033        } else if (dfd == AT_FDCWD) {
1034                read_lock(&fs->lock);
1035                nd->path = fs->pwd;
1036                path_get(&fs->pwd);
1037                read_unlock(&fs->lock);
1038        } else {
1039                struct dentry *dentry;
1040
1041                file = fget_light(dfd, &fput_needed);
1042                retval = -EBADF;
1043                if (!file)
1044                        goto out_fail;
1045
1046                dentry = file->f_path.dentry;
1047
```

```
1048                retval = -ENOTDIR;
1049                if (!S_ISDIR(dentry->d_inode->i_mode))
1050                        goto fput_fail;
1051
1052                retval = file_permission(file, MAY_EXEC);
1053                if (retval)
1054                        goto fput_fail;
1055
1056                nd->path = file->f_path;
1057                path_get(&file->f_path);
1058
1059                fput_light(file, fput_needed);
1060        }
1061
1062        retval = path_walk(name, nd);
1063        if (unlikely(!retval && !audit_dummy_context() && nd-
>path.dentry &&
1064                                nd->path.dentry->d_inode))
1065                audit_inode(name, nd->path.dentry);
1066out_fail:
1067        return retval;
1068
1069fput_fail:
1070        fput_light(file, fput_needed);
1071        goto out_fail;
1072}
```

Función nameidata_to_filp()

Es llamada dentro de la función do_file_open. Si esta parte del código finaliza correctamente se traducirá el fichero **nameidata** a una estructura **file** mediante la función comentada anteriormente **nameidata_to_filp().**

```
 926/**
 927 * nameidata_to_filp - convert a nameidata to an open filp.
 928 * @nd: pointer to nameidata
 929 * @flags: open flags
 930 *
 931 * Note that this function destroys the original nameidata
 932 */
/* Traduce la estructura nameidata a un fichero a través de la función
dentry. */
 933struct file *nameidata_to_filp(struct nameidata *nd, int flags)
 934{
 935        const struct cred *cred = current_cred();
 936        struct file *filp;
 937
 938        /* Pick up the filp from the open intent */
 939        filp = nd->intent.open.file;
 940        /* Has the filesystem initialised the file for us? */
 941        if (filp->f_path.dentry == NULL)
```

```
 942                filp = __dentry_open(nd->path.dentry, nd->path.mnt,
flags, filp, NULL, cred); //Funcion encargada de rellenar la estructura
FILE que se devuelve
 944        else
 945                path_put(&nd->path);
 946        return filp;
 947}
```

Función __dentry_open()

Ésta a su vez invoca a **__*dentry*_open()** pasándole las direcciones del objeto *dentry*, el sistema de ficheros montado, modo de acceso, el fichero objeto y la operación. Inicializa los campos del objeto archivo. En particular ajusta el campo f_op al contenido del campo i_op->default_file_ops del objeto i*nodo* (f->f_op = fops_get(inode->i_fop)). Esto activa todas las funciones adecuadas para futuras operaciones sobre archivos. Si el método open de las operaciones de archivo (defecto) está definido, lo invoca y limpia los indicadores O_CREAT, O_EXCL, O_NOCTTY, y O_TRUNC de f_flags.

Se le pasa a la función las direcciones del objeto *dentry*, el sistema de ficheros montado, modo de acceso, el fichero objeto y la operación. Inicializa los campos del objeto archivo; en particular, ajusta el campo f_op al contenido del campo i_op->default_file_ops del objeto *inodo* (f->f_op = fops_get (inode-> i_fop)). Esto activa todas las funciones adecuadas para futuras operaciones sobre archivos. Si el método open de las operaciones de archivo (defecto) esta definido, invocarlo. Limpia los indicadores O_CREAT, O_EXCL, O_NOCTTY, y O_TRUNC de f_flags.

```
805static struct file *__dentry_open(struct dentry *dentry, struct
vfsmount *mnt,
 806                                  int flags, struct file *f,
 807                                  int (*open)(struct inode *,
struct file *),
 808                                  const struct cred *cred)
 809{
 810        struct inode *inode;
 811        int error;
 812
 813        f->f_flags = flags; /* Comienzan las asignaciones a los
campos del file. */
 814        f->f_mode = (__force fmode_t)((flags+1) & O_ACCMODE) |
FMODE_LSEEK |
 815                                FMODE_PREAD | FMODE_PWRITE;
 816        inode = dentry->d_inode;
 817        if (f->f_mode & FMODE_WRITE) {/* Comprobación de permisos. */
 818                error = __get_file_write_access(inode, mnt);
```

```
 819                 if (error)
 820                         goto cleanup_file; /* Si falla destruye la
estructura file. */
 821                 if (!special_file(inode->i_mode))
 822                         file_take_write(f);
 823         }
 824
 825         f->f_mapping = inode->i_mapping; /* Asignación del inode. */
 826         f->f_path.dentry = dentry;
 827         f->f_path.mnt = mnt;
 828         f->f_pos = 0;
 829         f->f_op = fops_get(inode->i_fop); /* Activa todas las
funciones futuras */
 830         file_move(f, &inode->i_sb->s_files); /*operaciones. */
 831
 832         error = security_dentry_open(f, cred);
 833         if (error)
 834                 goto cleanup_all;
 835
 836         if (!open && f->f_op) )    /* Si el modo->abierto no está
establecido, lo pone. */
 837                 open = f->f_op->open;
 838         if (open) {
 839                 error = open(inode, f); /* APERTURA DEL FICHERO */
 840                 if (error)
 841                         goto cleanup_all; /* Error de apertura, mata
estructura file y pone los  objetos a null. */
 842         }
 843/* Elimina todos los flags. */
 844         f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
 845
 846         file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);
 847
 848         /* NB: we're sure to have correct a_ops only after f_op->open
*/
 849         if (f->f_flags & O_DIRECT) {
 850                 if (!f->f_mapping->a_ops ||
 851                     ((!f->f_mapping->a_ops->direct_IO) &&
 852                     (!f->f_mapping->a_ops->get_xip_mem))) {
 853                         fput(f);
 854                         f = ERR_PTR(-EINVAL);
 855                 }
 856         }
 857
 858         return f;
 859
 860cleanup_all:
 861         fops_put(f->f_op);
 862         if (f->f_mode & FMODE_WRITE) {
 863                 put_write_access(inode);
 864                 if (!special_file(inode->i_mode)) {
 865                         /*
 866                          * We don't consider this a real
 867                          * mnt_want/drop_write() pair
 868                          * because it all happenend right
 869                          * here, so just reset the state.
 870                          */
```

```
 871                        file_reset_write(f);
 872                        mnt_drop_write(mnt);
 873                }
 874        }
 875        file_kill(f);
 876        f->f_path.dentry = NULL;
 877        f->f_path.mnt = NULL;
 878cleanup_file:
 879        put_filp(f);
 880        dput(dentry);
 881        mntput(mnt);
 882        return ERR_PTR(error);
 883}
```

19.5 Creación de un archivo

La llamada al sistema create() crea un fichero y lo abre en modo escritura, independientemente del modo:

fd = CREAT (char *nombre, int permisos)

Dicha llamada es establecida por la función del sistema sys_create, la cual hace una llamada a sys_open con los parámetros de creación de ficheros además del modo de acceso al fichero, que serán con las banderas (O_CREAT | O_WRONLY | O_TRUNC) para crear archivo o si existe suprimir su contenido. Por tanto, la función sys_create() presenta los siguientes parámetros de entrada:

- **Pathname**: nombre del fichero creado.
- **Mode**: permisos del fichero.

```
1082SYSCALL_DEFINE2(creat, const char __user *, pathname, int, mode)
1083{
1084        return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC,
mode);
1085}
```

19.6 Cierre de un archivo

La llamada al sistema close() (número 6) se encarga de cerrar el fichero y es establecida por la llamada sys_close() que recibe como parámetro el descriptor del archivo que va a cerrarse. Dicha función se encuentra está contenida en el fichero fuente: fs/open.c

```
1118SYSCALL_DEFINE1(close, unsigned int, fd)
1119{
1120         struct file * filp;
        /*Asigna la estructura del fichero que hizo la llamada a open. */
1121         struct files_struct *files = current->files;
1122         struct fdtable *fdt;
1123         int retval;
1124
1125         spin_lock(&files->file_lock); /* Comienza la exclusión
mutua.*/
1126         fdt = files_fdtable(files); /* Busca el descriptor del
fichero*/
1127         if (fd >= fdt->max_fds) /* Si no lo encuentra desbloquea*/
1128                 goto out_unlock;
1129         filp = fdt->fd[fd]; /*Si lo encuentra se le asigna a filp*/
1130         if (!filp) /* Si no lo encontró desbloquea. */
1131                 goto out_unlock;
1132         rcu_assign_pointer(fdt->fd[fd], NULL); /* Limpia los bits. */
1133         FD_CLR(fd, fdt->close_on_exec);
1134         __put_unused_fd(files, fd); /* Libera el descriptor. */
1135         spin_unlock(&files->file_lock); /* Desbloquea. */
1136         retval = filp_close(filp, files); /* CIERRA */
1137
1138         /* Comprueba que se ha cerrado correctamente. */
1139         if (unlikely(retval == -ERESTARTSYS ||
1140                      retval == -ERESTARTNOINTR ||
1141                      retval == -ERESTARTNOHAND ||
1142                      retval == -ERESTART_RESTARTBLOCK))
1143                 retval = -EINTR;
1144
1145         return retval;
1146
1147out_unlock:
1148         spin_unlock(&files->file_lock);
1149         return -EBADF;
1150}
```

Filp_close:

1º.- Invoca al método flush de las operaciones de archivo, si está definido.

2º.- Libera cualquier cerrojo obligatorio sobre el archivo.

3º.- E invoca a fput() para liberar el objeto archivo.

Para finalizar, **close()** devuelve el código de error del método flush (normalmente 0).

```
1093int filp_close(struct file *filp, fl_owner_t id)
1094{
1095        int retval = 0;
1096
1097        if (!file_count(filp)) {
1098                printk(KERN_ERR "VFS: Close: file count is 0\n");
1099                return 0;
1100        }
1101
1102        if (filp->f_op && filp->f_op->flush)
1103                retval = filp->f_op->flush(filp, id); /* Libera las
operaciones asignadas al file. */
1104
1105        dnotify_flush(filp, id);
1106        locks_remove_posix(filp, id); /* Libera cerrojos. */
1107        fput(filp); /* Libera el objeto. */
1108        return retval;
1109}
```