LECCIÓN 21:

LLAMADAS AL SISTEMA

READ Y WRITE

ÍNDICE

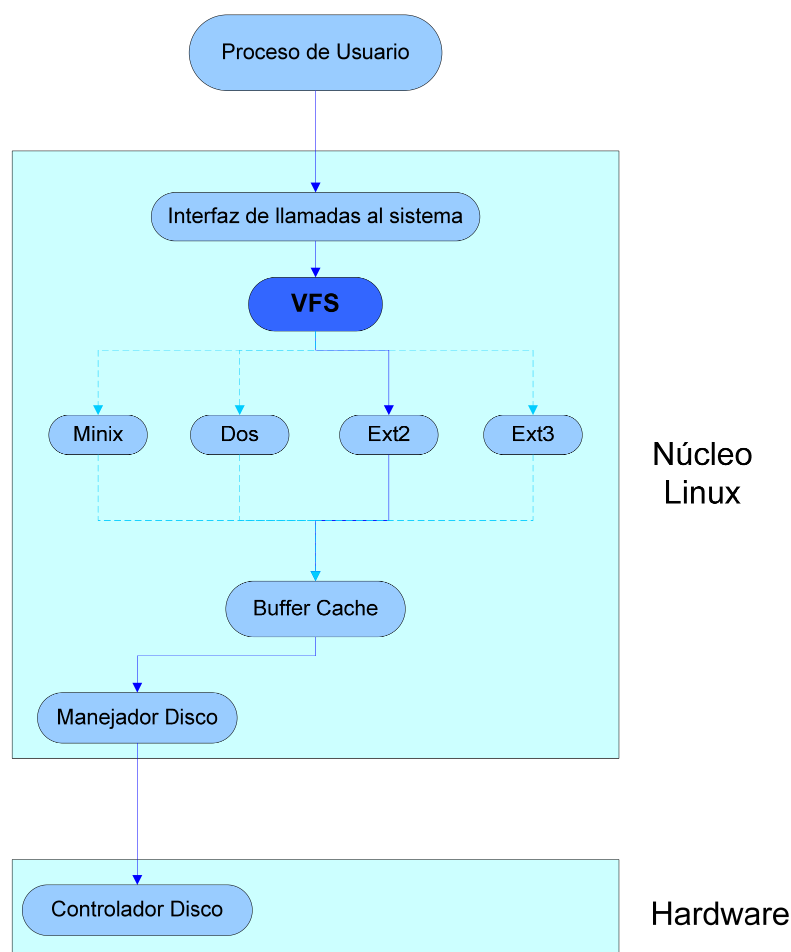
ÍNDICE 2
Introducción 3
Estructuras de datos 5
Estructura file 5
Estructura file_operations 6
Estructura ext2_file_operations 8
Estructura files_struct 8
Estructura fs_struct 9
Llamada al sistema READ 10
Función sys_read 11
Función vfs_read 12
do_sync_read 14
generic_file_aio_read 15
do_generic_file_read 17
readpage 21
do_mpage_readpage 22
Llamada al sistema WRITE 23
Función sys_write 24
Función vfs_write 25
Funciones auxiliares de read y write 27
Funciones fget_light y fcheck_files 27
Funciones fput_light y fput 29
Función rw_verify_area 29

Introducción

El núcleo de Linux dispone de dos llamadas al sistema, ***read*** y ***write***, que permiten a un proceso de usuario leer y escribir datos en un archivo que previamente debe haber sido abierto por medio de la llamada al sistema ***open***. Esta llamada open devuelve un descriptor de fichero *fd* el cual tendrán que utilizar el ***read*** y el ***write*** para acceder al archivo en cuestión.

Linux considera los archivos como una serie de bytes que carecen de tipo y de estructura. Por tanto, no impone ninguna restricción sobre los datos que pueda leer o escribir un proceso de cualquier tipo de ficheros. De modo que es capaz de soportar varios tipos de Sistemas de Ficheros como EXT2, Minix, Fat, etc.

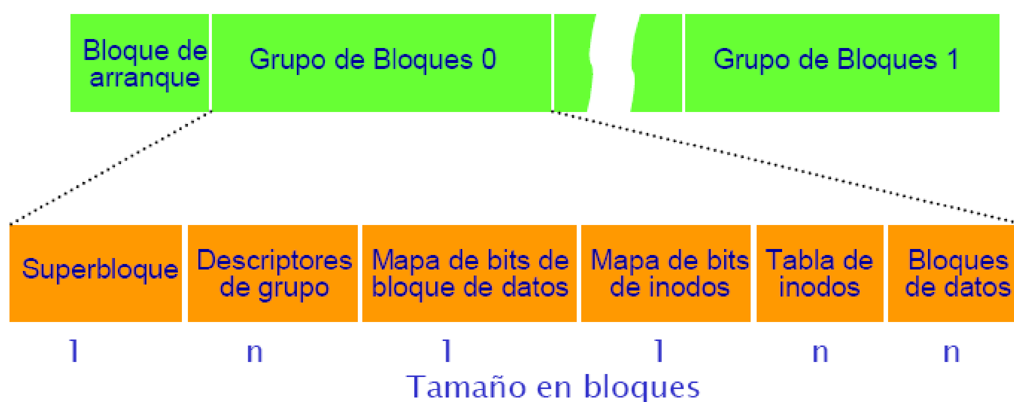
Por tanto, los procesos deben disponer de un acceso uniforme a los datos independientemente del tipo de Sistema de Fichero que se esté utilizando, para lo cual se sirven del Sistema Virtual de Ficheros (VFS) que es la interfaz entre las llamadas al sistema y el sistema real de ficheros que gestiona los datos. Esto es, cada vez que se ejecute una llamada al sistema, ésta se dirige al VFS el cual redirige la petición hacia el módulo que gestiona el fichero, como se puede observar con más detalle en la siguiente figura:



Las llamadas al sistema read y write actuarán sobre el sistema de ficheros EXT2 cuyas características principales son:

- Divide el disco en pequeñas unidades lógicas llamadas bloques que son las unidades mínimas que pueden ser asignadas y su tamaño se puede establecer a la hora de crear el sistema de ficheros.
- Divide las particiones lógicas que ocupa en grupo de bloques que es un conjunto de bloques secuenciales, que mantiene la información relacionada físicamente en el disco y facilita las tareas de gestión ya que el sistema de ficheros es manejado globalmente como una serie de grupos de bloques.

Los grupos de bloques contienen información como la que se muestra en la siguiente figura:



Superbloque: Copia del superbloque. Esta estructura contiene las informaciones de control del sistema de archivos y se duplica en cada grupo de bloques para permitir recuperar fácilmente una corrupción del sistema de archivos.

Descriptores de grupo: contienen las direcciones de bloques que contienen las informaciones cruciales, como los bloques y la tabla de inodos. También se duplican en cada grupo de bloques.

Mapa de bits de bloques de datos: a cada bloque se le asocia un bit indicando si está asignado o disponible.

Mapa de bits de bloques inodos: a cada inodo se le asocia un bit indicando si está asignado o disponible.

Tabla de inodos: contiene una parte de la tabla de inodos del sistema de archivos.

Bloques de datos: se utilizan para almacenar los datos contenidos en los archivos y los directorios.

Estructuras de datos

Explicaremos las estructuras de datos más relevantes que se utilizan en las llamadas al sistema **read** y **write**. Ambas llamadas acceden a las mismas estructuras del Sistema Virtual de Ficheros (VFS) aunque tan solo a unos campos en concreto, por ello solo nos centraremos en esos campos más significantes.

Estructura file

Cuando un proceso de usuario abre un fichero el sistema virtual de ficheros crea una estructura file que se define en *include/linux/fs.h*.

```
 812 struct file {

 817         union {
 818                 struct list_head        fu_list;
 819                 struct rcu_head         fu_rcuhead;
 820         } f_u;
 821         struct path             f_path;
```

Puntero al inodo de ese fichero

```
 822#define f_dentry        f_path.dentry
 823#define f_vfsmnt        f_path.mnt
```

Conjunto de operaciones relacionadas con un fichero abierto

```
 824         const struct file_operations    *f_op;
 825         atomic_long_t           f_count;
```

Flag relacionado con los permisos sobre el fichero

```
 826         unsigned int            f_flags;
```

Modo de apertura del fichero en la llamada open, conjunción de FMODE_READ y FMODE_WRITE

```
 827         fmode_t                 f_mode;
```

Posición actual desde el principio del fichero en bytes

```
 828         loff_t                  f_pos;
```

Proceso o grupo de procesos propietarios a los que se les envía la señal SIGIO

```
 829         struct fown_struct      f_owner;
```

```
 830         unsigned int            f_uid, f_gid;
```

Estructura que contiene parámetros relacionados con la lectura anticipada como: número máximo de bloques a leer, posición del primer byte en el archivo tras la lectura de la última página, tamaño de la lectura.

```
 831         struct file_ra_state    f_ra;
 832
 833         u64                     f_version;
 834#ifdef CONFIG_SECURITY
 835         void                    *f_security;
 836#endif
 837         /* needed for tty driver, and maybe others */
 838         void                    *private_data;
 839
 840#ifdef CONFIG_EPOLL
 841 /* Used by fs/eventpoll.c to link all the hooks to this file */
 842         struct list_head        f_ep_links;
 843         spinlock_t              f_ep_lock;
 844#endif /* #ifdef CONFIG_EPOLL */
 845         struct address_space    *f_mapping;
 846#ifdef CONFIG_DEBUG_WRITECOUNT
 847         unsigned long f_mnt_write_state;
 848#endif
 849};
```

Estas estructuras “file” se encadenan en una lista global doblemente encadenada, apuntada por la variable fu_list.

Estructura file_operations

La estructura *file_operations* se encuentra definida en el fichero *include/linux/fs.h*. En esta estructura se definen todas las operaciones que ejecuta el núcleo sobre ficheros.

```
1281struct file_operations {
1282         struct module *owner;
```

Se llama para modificar la posición actual del fichero

```
1283         loff_t (*llseek) (struct file *, loff_t, int);
```

Lectura en el fichero

```
1284         ssize_t (*read) (struct file *, char __user *, size_t,
loff_t *);
```

Escritura en el fichero

```
1285         ssize_t (*write) (struct file *, const char __user *,
size_t, loff_t *);
```

Lectura asíncrona

```
1286         ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
unsigned long, loff_t);
```

Escritura asíncrona

```
1287         ssize_t (*aio_write) (struct kiocb *, const struct iovec
*, unsigned long, loff_t);
```

Lectura de un directorio

```
1288         int (*readdir) (struct file *, void *, filldir_t);

1289         unsigned int (*poll) (struct file *, struct
poll_table_struct *);
```

Operación ioctl sobre un dispositivo

```
1290         int (*ioctl) (struct inode *, struct file *, unsigned int,
unsigned long);
1291         long (*unlocked_ioctl) (struct file *, unsigned int,
unsigned long);
1292         long (*compat_ioctl) (struct file *, unsigned int,
unsigned long);
```

Proyecta un fichero en memoria

```
1293         int (*mmap) (struct file *, struct vm_area_struct *);
```

Apertura de un fichero

```
1294         int (*open) (struct inode *, struct file *);
1295         int (*flush) (struct file *, fl_owner_t id);
```

Función para llamar en el último cierre de un fichero

```
1296         int (*release) (struct inode *, struct file *);
```

Escribe los bloques buffer de un fichero en el disco

```
1297         int (*fsync) (struct file *, struct dentry *, int
datasync);
1298         int (*aio_fsync) (struct kiocb *, int datasync);
```

Se utiliza cuando un proceso realiza la llamada fcntl para activar o desactivar las operaciones asíncronas

```
1299         int (*fasync) (int, struct file *, int);
1300         int (*lock) (struct file *, int, struct file_lock *);
1301         ssize_t (*sendpage) (struct file *, struct page *, int,
size_t, loff_t *, int);
1302         unsigned long (*get_unmapped_area)(struct file *, unsigned
long, unsigned long, unsigned long, unsigned long);
1303         int (*check_flags)(int);
1304         int (*dir_notify)(struct file *filp, unsigned long arg);
1305         int (*flock) (struct file *, int, struct file_lock *);
```

```
1306         ssize_t (*splice_write)(struct pipe_inode_info *, struct
file *, loff_t *, size_t, unsigned int);
1307         ssize_t (*splice_read)(struct file *, loff_t *, struct
pipe_inode_info *, size_t, unsigned int);
1308         int (*setlease)(struct file *, long, struct file_lock **);
1309};
```

Estructura ext2_file_operations

Se define en el fichero *fs/ext2/file.c*. Contiene todas las funciones genéricas que tratan con ficheros de usuarios de cualquier tipo. Es decir, se pueden tratar ficheros de cualquier tipo utilizando las mismas funciones.

```
  45const struct file_operations ext2_file_operations = {
```

Posicionamiento genérico según el tipo de fichero

```
  46         .llseek         = generic_file_llseek,
```

Lectura genérica según el tipo de fichero

```
  47         .read           = do_sync_read,
```

Escritura genérica según el tipo de fichero

```
  48         .write          = do_sync_write,

  49         .aio_read       = generic_file_aio_read,
  50         .aio_write      = generic_file_aio_write,
  51         .unlocked_ioctl = ext2_ioctl,
  52#ifdef CONFIG_COMPAT
  53         .compat_ioctl   = ext2_compat_ioctl,
  54#endif
  55         .mmap           = generic_file_mmap,
  56         .open           = generic_file_open,
  57         .release        = ext2_release_file,
  58         .fsync          = ext2_sync_file,
  59         .splice_read    = generic_file_splice_read,
  60         .splice_write   = generic_file_splice_write,
  61};
```

Estructura files_struct

Se define en el fichero *include/linux/fdtable.h*. El sistema le asocia a cada proceso una tabla de descriptores de ficheros abiertos en el campo “files” dentro de la estructura task_struct. Esta tabla también contiene el número máximo de descriptores que tiene la tabla y el máximo de descriptores que pueden estar asignados en un determinado momento. Además, contiene un semáforo para bloquear la ejecución cuando realice los accesos atómicos que quedarán reflejados en un contador.

```
  41struct files_struct {
  42  /*
```

```
43   * read mostly part
44   */
```

Contador de accesos atómicos

```
45         atomic_t count;
46         struct fdtable *fdt;
47         struct fdtable fdtab;
48   /*
49    * written part on a separate cache line in SMP
50    */
```

Bloquea todos los miembros situados por debajo

```
51         spinlock_t file_lock ____cacheline_aligned_in_smp;
```

Siguiente descriptor a asignar

```
52         int next_fd;
```

Cadena de bits utilizados por algunas tareas

```
53         struct embedded_fd_set close_on_exec_init;
54         struct embedded_fd_set open_fds_init;
```

Vector actual de descriptores de ficheros

```
55         struct file * fd_array[NR_OPEN_DEFAULT];
56};
```

Estructura fs_struct

El sistema para cada proceso dentro de su task_struct, mantiene una estructura fs_struct, que contiene información sobre el sistema de ficheros, como la ruta del root y del directorio actual, para la gestión de los ficheros de cada proceso. Se encuentra definida en include/linux/fs_struct.h

```
6struct fs_struct {
```

Para accesos atómicos

```
7         atomic_t count;
8         rwlock_t lock;
```

Derechos de acceso predeterminados para la creación de ficheros

```
9         int umask;
```

Estructura de directorio para accecer al inodo de root y al directorio actual del proceso

```
10         struct path root, pwd;
11};
```

Llamada al sistema READ

La llamada al sistema *read* se realiza desde un proceso de usuario y permite la lectura de datos de un archivo. Esta llamada se define en *fs/read_write.c*.

Prototipo

```
# include < unistd.h >
ssize_t read(unsigned int fd, char __user * buf, size_t count)
```

Parámetros

fd	descriptor de fichero, creado en una llamada previa *open*, del cual se leerán los datos.
buf	puntero al buffer de caracteres donde se recibirán los datos leídos.
count	número máximo de bytes que se van a almacenar en el buffer de caracteres.

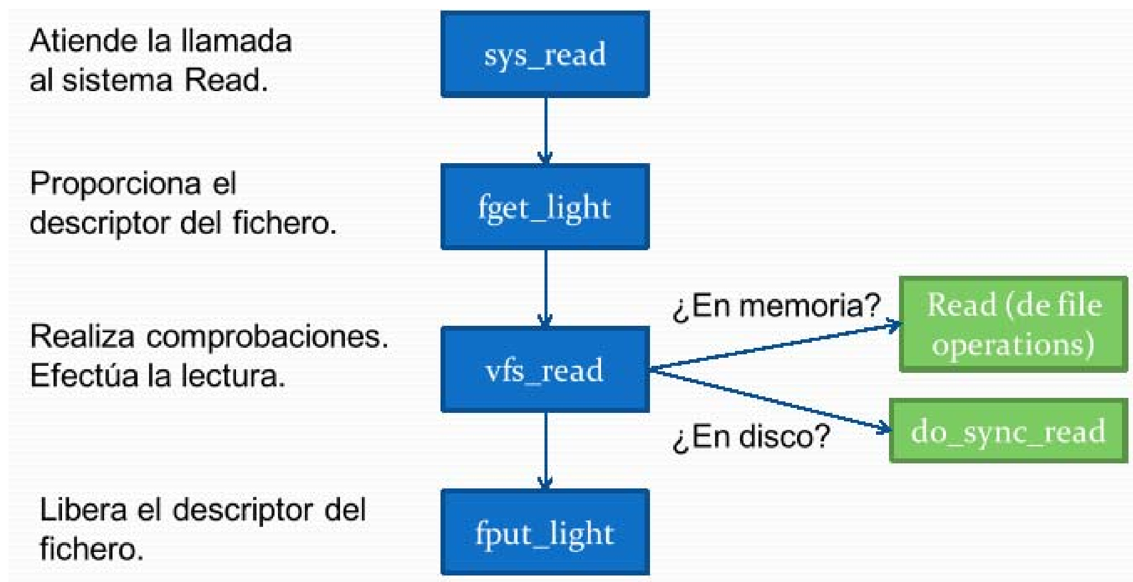
Esta llamada lee los datos de los dispositivos de bloques a la caché del núcleo para luego copiarlos al espacio de direcciones del proceso.

La llamada al sistema *read* devuelve el número de bytes que se han podido leer del archivo y en caso de error devuelve -1. Si esto último ocurre, se almacena el tipo de error en la variable **errno.** Los valores que puede tomar son:

Tipo de error

Valor de errno	Causa del error
EBADF	El descriptor de fichero especificado no es válido.
EFAULT	buf contiene una dirección no válida.
EINTR	La llamada al sistema ha sido interrumpida por la recepción de una señal.
EINVAL	fd se refiere a un objeto sobre el que no es posible una lectura.
EIO	Se ha producido un error de entrada/salida.
EISDIR	fd se refiere a un directorio.

Flujo de ejecución Read



Función sys_read

Como cualquiera de las llamadas al sistema ya vistas, utiliza la tabla sys_call_table que contiene las direcciones de las funciones que implementan las llamadas al sistema. En este caso, la función que implementa la llamada al sistema *read* se llama **sys_read**, definida en *fs/read_write.c*, cuya dirección se encontrará en la cuarta entrada de la tabla debido a que *read* tiene el número de llamada al sistema 3. Por tanto, accedemos a la dirección definida por sys_call_table + (3 * word_size) de donde obtenemos la dirección de la función **sys_read.** A continuación se muestra su código:

```
 359 asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf,
size_t count)
 360{
      /*Declara un puntero de tipo file*/
 361         struct file *file;

      /*declara el valor de retorno y lo inicializamos*/
 362         ssize_t ret = -EBADF;

/*variable que determina si es necesario liberar la estructura file
asociada a fd*/

 363         int fput_needed;

      /*obtiene la estructura file asociada a fd*/
 365         file = fget_light(fd, &fput_needed);

      Si fget_light devuelve un valor válido…

 366         if (file) {
/*Obtiene la posición actual en bytes desde el principio del fichero*/
```

```
 367                    loff_t pos = file_pos_read(file);
/*Llama a la función vfs_read que es la que hace la lectura*/
 368                    ret = vfs_read(file, buf, count, &pos);
/*actualiza la posición en la estructura file asociada al fd*/
 369                    file_pos_write(file, pos);
/*Liberamos la estructura de fichero asociada a fd*/
 370                    fput_light(file, fput_needed);
 371            }
 372
 373            return ret;
 374}
```

Esta función busca la estructura de fichero file asociada al número fd que se le ha pasado a la función *read*. Esa estructura contiene un puntero a la función que se debe usar para leer datos de ese tipo de fichero en concreto. Esto se lleva a cabo en la función **vfs_read**, que tras hacer varias comprobaciones, llama a esa función *read* específica del fichero para leer realmente los datos del fichero. Por último, actualiza la nueva posición de lectura del fichero y deja la estructura de fichero asociada al número fd para posteriormente retornar.

Utiliza las funciones file_pos_read y file_pos_write, para leer el puntero dentro del fichero y para actualizarlo.

```
static inline loff_t file_pos_read(struct file *file)
 350{
 351            return file->f_pos;
 352}
 353
 354static inline void file_pos_write(struct file *file, loff_t pos)
 355{
 356            file->f_pos = pos;
 357}
```

Función vfs_read

Se define en el fichero *fs/read_write.c*. Esta función es la que realmente se encarga de realizar la lectura de los datos del fichero, para ello se basa en el sistema virtual de ficheros (**VFS**).

Lo primero que hace esta función es comprobar ciertos permisos y opciones para que la lectura se pueda ejecutar sobre el fichero, comprobaciones como que el descriptor sea válido, que el fd apunte a un objeto sobre el que sea posible realizar la lectura y que buf contenga una dirección válida.

Una vez comprobado que se puede realizar la lectura, se verifica que la sección del fichero a leer está disponible, es decir que no esté bloqueada. En caso de que no esté bloqueada la toma y comprueba que tiene los permisos adecuados para la lectura, en cuyo caso se realiza la lectura del fichero teniendo en cuenta que éste puede estar en memoria, con lo que la lectura se realiza sin problemas, en caso contrario se fuerza a que haga la lectura de disco.

Por último, si ha conseguido leer los datos del fichero notifica al directorio padre que ha accedido al fichero e incrementa la cantidad de bytes leídos en la estructura correspondiente a la tarea actual que está realizando la llamada al sistema. Además, aunque no haya conseguido leer nada, incrementa el número de llamadas al sistema *read* que se han realizado en lo que lleva ejecutándose la tarea actual.

```
 264ssize_t vfs_read(struct file *file, char __user *buf, size_t
count, loff_t *pos)
 265{
 266         ssize_t ret;

      /*Comprueba que la lectura se puede realizar*/
 268         if (!(file->f_mode & FMODE_READ))
 269                 return -EBADF; Descriptor no válido

 270         if (!file->f_op || (!file->f_op->read && !file->f_op-
>aio_read))
 271                 return -EINVAL; No es posible realizar lectura

      /*Comprueba que la dirección del buffer es correcta*/
 272         if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
 273                 return -EFAULT; Dirección de buffer inválida

/*Comprueba que no haya otra llamada accediendo a la misma zona del
fichero*/

 275         ret = rw_verify_area(READ, file, pos, count);
 276         if (ret >= 0) {
                  /*Si no esta bloqueada….*/
 277                 count = ret;
                  /*comprueba que tiene permiso de lectura*/
 278                 if (file->f_op->read)
                        /*se realiza la lectura desde memoria*/
 279                   ret = file->f_op->read(file, buf, count, pos);
                        /* sino fuerza lectura de disco */
 280                 else
 281                         ret = do_sync_read(file, buf, count, pos);

/*Si se ha leído algo se notifica directorio padre que ha accedido al
fichero */

 282                 if (ret > 0) {
 283                         fsnotify_access(file->f_path.dentry);
 284                         add_rchar(current, ret);
 285                 }
 286                 inc_syscr(current);
 287         }
 288
 289         return ret;
```

```
 290}
```

La función read contenida en el file->f_op->**read**(file, buf, count, pos) es la que se encuentra en la estructura struct **file_operations ext2_file_operations** que nos lleva a la lectura genérica según el tipo de fichero

```
  26         .read               = do_sync_read,
```

do_sync_read

Se encuentra en el fichero fs/read_write.c y utiliza la estructura kiocb
filp – dirección de la estructura file del fichero
buf – dirección de memoria en el espacio del usuario donde los datos van a depositarse
len – número de caracteres a ser leidos
ppos – posición del puntero dentro del fichero desde donde comienza la lectura

```
 239ssize_t do_sync_read(struct file *filp, char __user *buf, size_t
len, loff_t *ppos)
 240{
```

Inicializa la estrutura iovec con la dirección del buffer del usuario y la cuenta de caracteres

```
 241         struct iovec iov = { .iov_base = buf, .iov_len = len };
```

La estructura kiocb se utiliza para llevar la gestión y el estado de las operaciones de entrada/salida

```
 242         struct kiocb kiocb;
 243         ssize_t ret;
 244
```

Rellena la estructura kiocb con los campos de file

```
 245         init_sync_kiocb(&kiocb, filp);
 246         kiocb.ki_pos = *ppos;
 247         kiocb.ki_left = len;
 248
 249         for (;;) {
```

Mientras queden datos por leer llama aio_read pasándole iovec y kiocb. Retorna un valor con los bytes leidos que es lo que retorna la función

```
 250                 ret = filp->f_op->aio_read(&kiocb, &iov, 1,
kiocb.ki_pos);
 251                 if (ret != -EIOCBRETRY)
 252                         break;
 253                 wait_on_retry_sync_kiocb(&kiocb);
```

```
 254         }
 255
 256         if (-EIOCBQUEUED == ret)
 257                 ret = wait_on_sync_kiocb(&kiocb);
 258         *ppos = kiocb.ki_pos;
 259         return ret;
 260}
```

filp->f_op->**aio_read**(&kiocb, &iov, 1, kiocb.ki_pos), en la estructura **ext2_file_operations** nos lleva a la function aio_read = **generic_file_aio_read** que se encuentra en mm/filemap.c

generic_file_aio_read

Es una función de propósito general para implementar tanto las lecturas síncronas como las asíncronas.

```
/**
1130  * generic_file_aio_read - rutina generica read para el
filesystem
1131  * @iocb:       estructura kiocb para que el kernel controle I/O
1132  * @iov:        vector con las solicitudes de io
1133  * @nr_segs:    número de segmentos en el iovec
1134  * @pos:        posición actual dentro del archivo
1135  *
1136  * Esta es la función  "read()" para todos los sistemas de
ficheros
1137  * puede utilizar la cache de páginas directamente.
1138  */
1293ssize_t
1294generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,
1295                 unsigned long nr_segs, loff_t pos)
1296{
1297         struct file *filp = iocb->ki_filp;
1298         ssize_t retval;
1299         unsigned long seg;
1300         size_t count;
1301         loff_t *ppos = &iocb->ki_pos;
1302
1303         count = 0;
1304         retval = generic_segment_checks(iov, &nr_segs, &count,
VERIFY_WRITE);
1305         if (retval)
1306                 return retval;
1307
```

Si la lectura es directa (O_DIRECT), realiza la lectura directamente sobre el dispositivo sin utilizar el cache de páginas

```
1309         if (filp->f_flags & O_DIRECT) {
1310                 loff_t size;
1311                 struct address_space *mapping;
1312                 struct inode *inode;
1313
1314                 mapping = filp->f_mapping;
1315                 inode = mapping->host;
1316                 if (!count)
```

```
1317                            goto out; /* skip atime */
1318                    size = i_size_read(inode);
1319                    if (pos < size) {
1320                            retval = filemap_write_and_wait(mapping);
1321                            if (!retval) {
1322                              retval = mapping->a_ops->direct_IO(READ,
iocb,
1323            iov, pos, nr_segs);
1324                            }
1325                            if (retval > 0)
1326                                    *ppos = pos + retval;
1327                            if (retval) {
1328                                    file_accessed(filp);
1329                                    goto out;
1330                            }
1331                    }
1332            }
```

Rellena la estructura desc con el estado de la operación de lectura solicitada, bytes que han sido copiados al usuario, Dirección del buffer, bytes que quedan por transferir y error

```
1334            for (seg = 0; seg < nr_segs; seg++) {
1335                    read_descriptor_t desc;
1336
1337                    desc.written = 0;
1338                    desc.arg.buf = iov[seg].iov_base;
1339                    desc.count = iov[seg].iov_len;
1340                    if (desc.count == 0)
1341                            continue;
1342                    desc.error = 0;
```

Llama a la función de lectura pasándole la estructura file, la estructura desc y la dirección de la función manejadora que escribe los datos en la dirección de usuario

```
1343                    do_generic_file_read(filp, ppos, &desc,
file_read_actor);
1344                    retval += desc.written;
1345                    if (desc.error) {
1346                            retval = retval ?: desc.error;
1347                            break;
1348                    }
1349                    if (desc.count > 0)
1350                            break;
1351            }
```

Devuelve el número de bytes copidos en el bufer del usuario, que es el valor del campo desc.writen

```
1352 out:
1353            return retval;
1354}
```

do_generic_file_read

Se encuentra en el fichero include/linux/fs.h. Implementa la operación read de datos de un archivo, utilizando la cache de páginas, lee las páginas de disco, las escribe en la cache de páginas y las escribe en el buffer del usuario.

```
 998static void do_generic_file_read(struct file *filp, loff_t *ppos,
 999                  read_descriptor_t *desc, read_actor_t actor)
1000{
1001         struct address_space *mapping = filp->f_mapping;
1002         struct inode *inode = mapping->host;
1003         struct file_ra_state *ra = &filp->f_ra;
1004         pgoff_t index;
1005         pgoff_t last_index;
1006         pgoff_t prev_index;
1007         unsigned long offset;     // Desplazamiento dentro de la
                                   // caché de páginas
1008         unsigned int prev_offset;
1009         int error;
```

El fichero se encuentra dividido en páginas (4096 bytes). Calcula utilizando ppos el indice a la pagina que contiene el primer byte a leer

```
1011         index = *ppos >> PAGE_CACHE_SHIFT;
1012         prev_index = ra->prev_pos >> PAGE_CACHE_SHIFT;
1013         prev_offset = ra->prev_pos & (PAGE_CACHE_SIZE-1);
1014         last_index = (*ppos + desc->count + PAGE_CACHE_SIZE-1) >>
PAGE_CACHE_SHIFT;
```

Calcula el desplazamiento dentro de la página

```
1015         offset = *ppos & ~PAGE_CACHE_MASK;
```

Comienza un bucle para leer todas las paginas que contienen los datos requeridos

```
1017         for (;;) {
1018                 struct page *page;
1019                 pgoff_t end_index;
1020                 loff_t isize;
```

nr - numero maximo de bytes a copiar desde la página
nr = PAGE_CACHE_SIZE;

```
1021                 unsigned long nr, ret;
```

Comprueba si el proceso tiene activado el flag TIF_NEED_RESCHED para invocar a schedule()

```
1023                 cond_resched();
```

Busca en la cache de páginas el descriptor de la página que contiene los datos a leer pasandole las direcciones y el número de página

```
1024 find_page:
1025                 page = find_get_page(mapping, index);
```

Si la página no se encuentra en la cache

```
1026                 if (!page) {
```

Prepara los parámetros que necesita read-ahead para iniciar una lectura a disco

```
1027                         page_cache_sync_readahead(mapping,
1028                                         ra, filp,
1029                                         index, last_index -
index);
1030                         page = find_get_page(mapping, index);
1031                         if (unlikely(page == NULL))
```

Hay que traer la pagina de disco

```
1032                                 goto no_cached_page;
1033                 }
1034                 if (PageReadahead(page)) {
1035                         page_cache_async_readahead(mapping,
1036                                         ra, filp, page,
1037                                         index, last_index -
index);
1038                 }
```

La página esta en la cache comprueba si esta actualizada (PG_uptodate flag)

```
1039                 if (!PageUptodate(page)) {
1040                         if (inode->i_blkbits == PAGE_CACHE_SHIFT
||
1041                                         !mapping->a_ops-
>is_partially_uptodate)
```

Si no está actualizada leerla de disco

```
1042                                 goto page_not_up_to_date;
1043                         if (!trylock_page(page))
1044                                 goto page_not_up_to_date;
1045                         if (!mapping->a_ops-
>is_partially_uptodate(page,
1046
desc, offset))
1047                                 goto page_not_up_to_date_locked;
1048                         unlock_page(page);
1049                 }
1050 page_ok:
```

La página esta en la cache y esta actualiza

```
1060                 isize = i_size_read(inode);
1061                 end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
1062                 if (unlikely(!isize || index > end_index)) {
1063                         page_cache_release(page);
1064                         goto out;
1065                 }
```

```
1066
1067    /* nr is the maximum number of bytes to copy from this page */
1068                  nr = PAGE_CACHE_SIZE;
1069                  if (index == end_index) {
1070                          nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
```

Si no quedan bytes para leer en la página salimos

```
1071                          if (nr <= offset) {
1072                                  page_cache_release(page);
1073                                  goto out;
1074                          }
1075                  }
1076                  nr = nr - offset;
1077
```

Comprobar si otros usuarios estan escribiendo en este momento en la página.

```
1082                  if (mapping_writably_mapped(mapping))
1083                          flush_dcache_page(page);
1084
```

Cuando parte de la misma página es leida multiples veces sucesivamente, solo se marca como que se esta accediendo la primera vez activando los flags PG_referenced o PG_active.

```
1089                  if (prev_index != index || offset != prev_offset)
1090                          mark_page_accessed(page);
1091                  prev_index = index;
```

Una vez leída la página desde la cache o desde el disco, su contenido se copia en la memoria del usuario proporcinada por el proceso que llama con la función actor()cuya dirección ha sido pasada como parámetro

La función actor realiza los siguientes pasos:

1. Invoca a kmap() para realizar una proyección de la pagina
2. Lama a __copy_to_user(), para copiar los datos al buffer del usuario
3. Invoca a kunmap() para quitar la proyección de la página
4. Actualiza los campos count, griten y buf del read_descriptor_t

La función actor devuelve el número de bytes copiados, tiene que actualizar los punteros al buffer de usuario y el resto de la cuenta

```
1103                  ret = actor(desc, page, offset, nr);
1104                  offset += ret;
1105                  index += offset >> PAGE_CACHE_SHIFT;
1106                  offset &= ~PAGE_CACHE_MASK;
1107                  prev_offset = offset;
```

Decrementa el contador de uso de la página

```
1109                  page_cache_release(page);
```

Si count no es cero quedan datos por leer del fichero continúa para leer otra página del fichero

```
1110                    if (ret == nr && desc->count)
1111                            continue;
1112                    goto out;
1113
1114page_not_up_to_date:
```

Los datos de la página no están actualizados hay que leer de disco

```
1116                    error = lock_page_killable(page);
1117                    if (unlikely(error))
1118                            goto readpage_error;
1119
1120page_not_up_to_date_locked:
1121                    /* Did it get truncated before we got the lock? */
1122                    if (!page->mapping) {
1123                            unlock_page(page);
1124                            page_cache_release(page);
1125                            continue;
1126                    }
1127
1128                    /* Did somebody else fill it already? */
1129                    if (PageUptodate(page)) {
1130                            unlock_page(page);
1131                            goto page_ok;
1132                    }
```

Ahora se puede realizar la lectura de disco, llamando a la función **readpage** asociada al inodo llena la pagina con los datos de disco

```
1134 readpage:
1135            /* Start the actual read. The read will unlock the page. */
1136                    error = mapping->a_ops->readpage(filp, page);
1137
1138                    if (unlikely(error)) {
1139                            if (error == AOP_TRUNCATED_PAGE) {
1140                                    page_cache_release(page);
1141                                    goto find_page;
1142                            }
1143                            goto readpage_error;
1144                    }
1145
1146                    if (!PageUptodate(page)) {
```

Si el flag PG_update esta a cero, espera hasta que la página ha sido leida y el proceso que solicitó la lectura duerme hasta que se complete la transferencia

```
1147                            error = lock_page_killable(page);
1148                            if (unlikely(error))
1149                                    goto readpage_error;
1150                            if (!PageUptodate(page)) {
1151                                    if (page->mapping == NULL) {

1155                                            unlock_page(page);
1156                                            page_cache_release(page);
1157                                            goto find_page;
1158                                    }
1159                                    unlock_page(page);
```

```
1160                                    shrink_readahead_size_eio(filp,
ra);
1161                                    error = -EIO;
1162                                    goto readpage_error;
1163                            }
1164                            unlock_page(page);
1165                    }
1166
1167                    goto page_ok;
1168
1169readpage_error:
1170            /* UHHUH! A synchronous read error occurred. Report it */
1171                    desc->error = error;
1172                    page_cache_release(page);
1173                    goto out;
1174
```

Si la página no se encuentra en la cache, se crea una nueva página (alloc) se añade a la cache de páginas y se activa el flag PG_locked de la nueva página

```
1175 no_cached_page:
1180                    page = page_cache_alloc_cold(mapping);
1181                    if (!page) {
1182                            desc->error = -ENOMEM;
1183                            goto out;
1184                    }
```

El nuevo descriptor de página se inserta en la lista lru cache de páginas

```
1185                    error = add_to_page_cache_lru(page, mapping,
1186                                                    index, GFP_KERNEL);
1187                    if (error) {
1188                            page_cache_release(page);
1189                            if (error == -EEXIST)
1190                                    goto find_page;
1191                            desc->error = error;
1192                            goto out;
1193                    }
```

Página creada, salta a leer los datos de la página

```
1194                    goto readpage;
1195            }
1196
```

Todos los bytes han sido leidos

```
1197 out:
1198            ra->prev_pos = prev_index;
1199            ra->prev_pos <<= PAGE_CACHE_SHIFT;
1200            ra->prev_pos |= prev_offset;
1201
1202            *ppos = ((loff_t)index << PAGE_CACHE_SHIFT) + offset;
1203            file_accessed(filp);
1204}
```

readpage

La estructura:

```
static const struct address_space_operations ext3_writeback_aops = {
1785         .readpage         = ext3_readpage,
```

Nos lleva a la función ext3_readpage que no es más que un envoltorio para la función mpage_readpage en fs/ext3/inode.c que a su vez llama a do_mpage_readpage

```
1647 static int ext3_readpage(struct file *file, struct page *page)
1648 {
1649         return mpage_readpage(page, ext3_get_block);
1650 }
```

do_mpage_readpage

Es la que lee los bloques físicos de disco y los escribe en la cache de páginas llamando a get_block

Llamada al sistema WRITE

La escritura de datos que un proceso puede realizar en un archivo se lleva a cabo mediante la llamada al sistema *write*, cuyo código está en *fs/read_write.c.*

Prototipo

```
#include <unistd.h>
ssize_t write(unsigned int fd, char __user * buf, size_t count)
```

Consta de tres Parámetros:

Fd: descriptor del fichero en el que se quiere escribir
Buf: puntero que apunta al buffer donde se encuentran los datos a escribir.
Count: indica el número de bytes que contiene el buffer.

Esta llamada puede **retornar**:

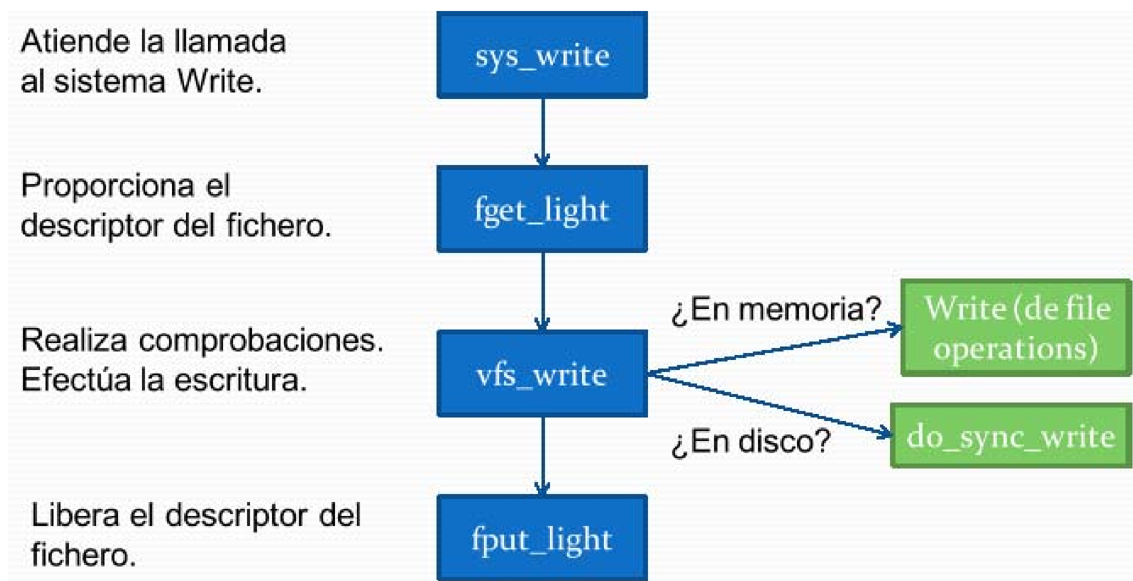
-Si se ha ejecutado correctamente: el **número de bytes que fueron transferidos**.
-Si ha habido algún error : **-1**.

Si ocurre algún **error**, dicho error se almacenará en la variable **errno**. La siguiente tabla muestra los valores que puede tomar:

Tipo de error

Valor de errno	Causa del error
EBADF	El descriptor de fichero especificado no es válido
EFAULT	*buf* contiene una dirección no válida
EINTR	La llamada al sistema ha sido interrumpida por la recepción de una
EINVAL	*fd* se refiere a un objeto sobre el que no es posible la escritura
EIO	Se ha producido un error de entrada/salida
EISDIR	*fd* se refiere a un directorio
EPIDE	*fd* se refiere a una tubería sobre la que no existe ya proceso lector
ENOSPC	El sistema de archivos está saturado

Flujo de ejecución Write



Función sys_write

La llamada al sistema *write* se encuentra en la tabla sys_call_table, la quinta entrada contiene la dirección de la función del núcleo **sys_write,** que es la que lleva a cabo la llamada *write*. Esto es así debido a que el *write* tiene el número de llamada 4; por tanto accedemos a la dirección definida por sys_call_table + (4 * word_size) de donde obtenemos la dirección de la función **sys_write** definida en el fichero *fs/read_write.c*

Esta función en primer lugar se encarga de obtener la dirección de la estructura del fichero (struct file) asociada al número fd que le ha sido pasado por parámetro. Esta estructura contiene un puntero a la función específica para escribir los datos en el fichero según su tipo.

La siguiente acción que se lleva a cabo en esta función es obtener la posición actual en el fichero almacenándola en la variable local pos, que será pasada como parámetro a la función **vfs_write**. Esta última, es la que lleva a cabo la escritura de los datos en el fichero (tras comprobar que la escritura es viable).

Para terminar, la función **sys_write**, actualiza la nueva posición de escritura en el fichero y deja la estructura de fichero asociada al fd para luego retornar.

```
 376asmlinkage ssize_t sys_write(unsigned int fd, const char __user *
buf, size_t count)
 377{
```

Estructura file asociada a fd

```
 378        struct file *file;
```

ret será la variable donde se almacene el valor de retorno. Se inicializa “el fd especificado no es válido”

```
379         ssize_t ret = -EBADF;
```

Variable que utilizan las funciones fget_light y fput_light para indicar si es necesario liberar la estructura file asociada a fd

```
        int fput_needed;
```

Se obtiene la dirección de la estructura file asociada a fd

```
 380         int fput_needed;
```

Si fd ha sido válido file debería tener una dirección asociada

```
 382         file = fget_light(fd, &fput_needed);
 383         if (file) {
```

Se obtiene la posición actual desde el principio del fichero en bytes

```
 384                 loff_t pos = file_pos_read(file);
```

Se llama a la función que realmente realiza la escritura de los datos en el fichero

```
 385                 ret = vfs_write(file, buf, count, &pos);
```

Se actualiza la posición actual en el fichero

```
 386                 file_pos_write(file, pos);
```

Se libera la estructura file asociada a fd

```
 387                 fput_light(file, fput_needed);
 388         }
 389 
 390         return ret;
 391}
```

Función vfs_write

Se define en el fichero *fs/read_write.c*. Esta función es la que realmente se encarga de realizar la escritura de los datos en el fichero apoyándose en el sistema virtual de ficheros.

La primera acción que hace esta función es comprobar ciertos permisos y opciones para que la escritura en el fichero sea posible, las comprobaciones son exactamente las mismas que las realizadas para la lectura, pero referidas a escritura.

Una vez comprobado que la escritura es viable, se verifica que la sección del fichero a escribir no esté bloqueada por otra llamada al sistema. En caso de que no esté bloqueada se bloquea. El siguiente paso es comprobar

que tiene los permisos adecuados para la escritura (función security_file_permission), en cuyo caso se realiza la escritura en el fichero teniendo en cuenta también si se encuentra en memoria, si no es así se fuerza a que haga la escritura sobre el fichero en disco.

Si ha conseguido escribir los datos en el fichero notifica al directorio padre que ha modificado el contenido del fichero e incrementa la cantidad de bytes escritos en la estructura correspondiente a la tarea actual que está realizando la llamada al sistema.

Por último, aunque no haya conseguido escribir nada, incrementa el número de llamadas al sistema *write* que se han realizado en lo que lleva ejecutándose la tarea actual.

```
 319ssize_t vfs_write(struct file *file, const char __user *buf,
size_t count, loff_t *pos)
 320{
```

ret será la variable donde se almacene el valor de retorno

```
 321        ssize_t ret;
```

Se comprueba que se puede realizar la escritura del fichero

```
 323        if (!(file->f_mode & FMODE_WRITE))
 324                return -EBADF;
```

error: fd especificado no es válido

```
 325        if (!file->f_op || (!file->f_op->write && !file->f_op-
>aio_write))
 326          return -EINVAL; // error: no es posible realizar la
escritura
 327        if (unlikely(!access_ok(VERIFY_READ, buf, count)))
 328          return -EFAULT; // error: dirección de buffer no válida
```

Se comprueba que la sección a escribir no está bloqueada (en el caso de no estarlo, se bloquea)

```
 330        ret = rw_verify_area(WRITE, file, pos, count);
```

Si no estaba bloqueada. Se comprueba si se tiene permiso de escritura

```
 331        if (ret >= 0) {//Si tiene permiso de escritura
 332                count = ret;
 333                if (file->f_op->write)
```

Se escriben los datos en memoria

```
 334                        ret = file->f_op->write(file, buf, count,
pos);
 335                else
```

Se fuerza la escritura en disco

```
 336                        ret = do_sync_write(file, buf, count,
pos);
```

```
 337                    if (ret > 0) {
```

Se notifica al directorio padre la modificación del archivo

```
 338                            fsnotify_modify(file->f_path.dentry);
```

Se incrementa el número de bytes escritos

```
 339                            add_wchar(current, ret);
 340                    }
```

Se incrementa el número de llamadas al sistema write

```
 341                    inc_syscw(current);
 342            }
 343
 344            return ret;
 345}
```

Funciones auxiliares de read y write

En esta sección se explican las funciones auxiliares que son llamadas en las funciones sys_read, sys_write, vfs_read y vfs_write. Estas funciones son exactamente las mismas tanto para la lectura como para la escritura.

Funciones fget_light y fcheck_files

La function **fget_light** se encuentra en el fichero *linux/fs/file_table.c*. Esta función obtiene la estructura de fichero asociada al descriptor de ficheros fs que se le pasa por parámetro. Devuelve la dirección a dicha estructura de fichero. Hace uso de la función **fcheck_files**.

En primer lugar tenemos que se declara un puntero a una estructura *files_struct* que apuntará a la estructura de este tipo correspondiente a la tarea actual que está realizando la llamada al sistema.

La función prosigue comprobando que la zona a la que se pretende acceder esté bloqueada ya por el proceso actual, en cuyo caso se limita a llamar a la función **fcheck_files** que se encarga de devolver la dirección de la estructura de fichero asociada a fd (que se le pasará como parámetro).

Si la zona no está bloqueada por el proceso actual, se procede al bloqueo, se vuelve a realizar la llamada a la función **fcheck_files** y posteriormente se llama a la función **get_file** para obtener la estructura apuntada por nuestro puntero local file. En este caso será necesario poner la variable de entrada/salida *fput_needed* a 1 para indicar que será necesaria una liberación de la zona.

```
 321struct file *fget_light(unsigned int fd, int *fput_needed)
 322{
 323        struct file *file;
```

```
 324          struct files_struct *files = current->files;
 325
 326          *fput_needed = 0;
```

Si la zona está bloqueada por el proceso actual...

```
 327          if (likely((atomic_read(&files->count) == 1))) {
```

Obtiene dirección estructura de fichero

```
 328                  file = fcheck_files(files, fd);

 329          } else {
```

En caso contrario La bloquea y la coge

```
 330                  rcu_read_lock();
 331                  file = fcheck_files(files, fd);
 332                  if (file) {
 333                          if (atomic_long_inc_not_zero(&file-
>f_count))
 334                                  *fput_needed = 1;
 335                          else
 336                                  /* Didn't get the reference,
someone's freed */
 337                                  file = NULL;
 338                  }
 339                  rcu_read_unlock();
 340          }
 341
 342          return file;
 343}
```

La función **fcheck_files** se encuentra definida en *include/linux/fdtable.h*. Es la que se encarga de comprobar que el descriptor de fichero fd es válido (es menor al máximo permitido por la estructura files), en cuyo caso devuelve la dirección de la estructura de fichero asociada al mismo.

```
  75static inline struct file * fcheck_files(struct files_struct
*files, unsigned int fd)
  76{
```

Puntero a estructura de fichero a devolver (inicialmente nula)

```
  77          struct file * file = NULL;
  78          struct fdtable *fdt = files_fdtable(files);
```

Comprueba que el f des válido

```
  80          if (fd < fdt->max_fds)
```

Estructura fichero asociada a fd

```
  81                  file = rcu_dereference(fdt->fd[fd]);
  82          return file;
  83}
```

Funciones fput_light y fput

La función **fput_light** se encuentra definida en el fichero *include/linux/file.h*. Es la encargada de liberar la estructura fichero asociada al descriptor de fichero fd. Hace uso de la función **fput**.

```
  27static inline void fput_light(struct file *file, int fput_needed)
  28{
  29        if (unlikely(fput_needed))
  30                fput(file);
  31}
```

```
 220void fput(struct file *file)
 221{
 222        if (atomic_long_dec_and_test(&file->f_count))
 223                __fput(file);
 224}
```

Realmente quien realiza la acción de liberar la zona es la función **fput**. La función **fput_light** simplemente comprueba mediante el parámetro *fput_needed* si es necesario la liberación de dicha zona. Será obligatoria la liberación de la zona cuando *fput_needed = 1* (caso visto en la función **fget_light** cuando la zona no estaba bloqueada anteriormente por el proceso actual).

La función **fput** se define en el fichero *linux/fs/file_table.c*. Esta a su vez va a llamar a la función **__fput**, siempre que se esté accediendo de forma atómica a dicha operación. Es esta última función la que se encarga de liberar la estructura de fichero que se ha venido utilizando.

Función rw_verify_area

Está definida en el fichero *fs/read_write.c*. Es la encargada de comprobar que la sección a leer del fichero no esté bloqueada, es decir que no haya otra llamada al sistema accediendo a la misma zona en el mismo instante.

Para ello lo primero que hace es comprobar que el tamaño de los datos a leer o escribir no supere el tamaño máximo de bloque establecido en la estructura file asociada al descriptor de fichero. En caso de que el tamaño sea correcto comprueba que la posición del fichero de la que se pretende leer o en la que se va a escribir y esa posición más la cantidad de bytes a leer o escribir no caigan fuera del bloque ocupado por el fichero.

Una vez realizadas las comprobaciones pertinentes, lo siguiente que hace es coger la estructura inodo correspondiente a la entrada del fichero accedido. Con la información obtenida del inodo comprueba que la zona accedida no está bloqueada por otro proceso, en cuyo caso la bloquea según el tipo de operación que se vaya a efectuar: si es una lectura solo se bloquea la zona que

se va a leer (acceso compartido, no se modifican datos), mientras que si es una escritura se bloquea todo el fichero (acceso exclusivo, se modifica contenido del fichero).

```
 202int rw_verify_area(int read_write, struct file *file, loff_t
*ppos, size_t count)
 203{
 204          struct inode *inode;
 205          loff_t pos;
 206          int retval = -EINVAL;
```

Retval coge el valor de la dirección al error Einval.

```
 207
 208          inode = file->f_path.dentry->d_inode;
 209          if (unlikely((ssize_t) count < 0))
 210                  return retval;
 211          pos = *ppos;
```

Se comprueba que el tamaño de los datos a leer o escribir no debe ser mayor que tamaño máximo permitido en estructura fichero.

```
 212          if (unlikely((pos < 0) || (loff_t) (pos + count) < 0))
 213                  return retval;
```

Va a la etiqueta Einval (por medio de retval). En caso de error comprueba que no se intenta acceder fuera del fichero

```
 215          if (unlikely(inode->i_flock && mandatory_lock(inode))) {
 216                  retval = locks_mandatory_area(
 217                          read_write == READ ? FLOCK_VERIFY_READ :
FLOCK_VERIFY_WRITE,
 218                          inode, file, pos, count);
 219                  if (retval < 0)
 220                          return retval;
 221          }
```

Security file permission se encarga de comprobar si el acceso al fichero no está bloqueado. También observa el modo en el que fue abierto (escritura o lectura). En caso de poder, escribe.

```
 222          retval = security_file_permission(file,
 223                                  read_write == READ ? MAY_READ :
MAY_WRITE);
```

Se devuelve el resultado de la llamada a Write.

```
 224          if (retval)
 225                  return retval;
 226          return count > MAX_RW_COUNT ? MAX_RW_COUNT : count;
 227}
```