open, create y close

Enrique Ismael Mendoza Robaina
Javier Antonio Monzón Santana

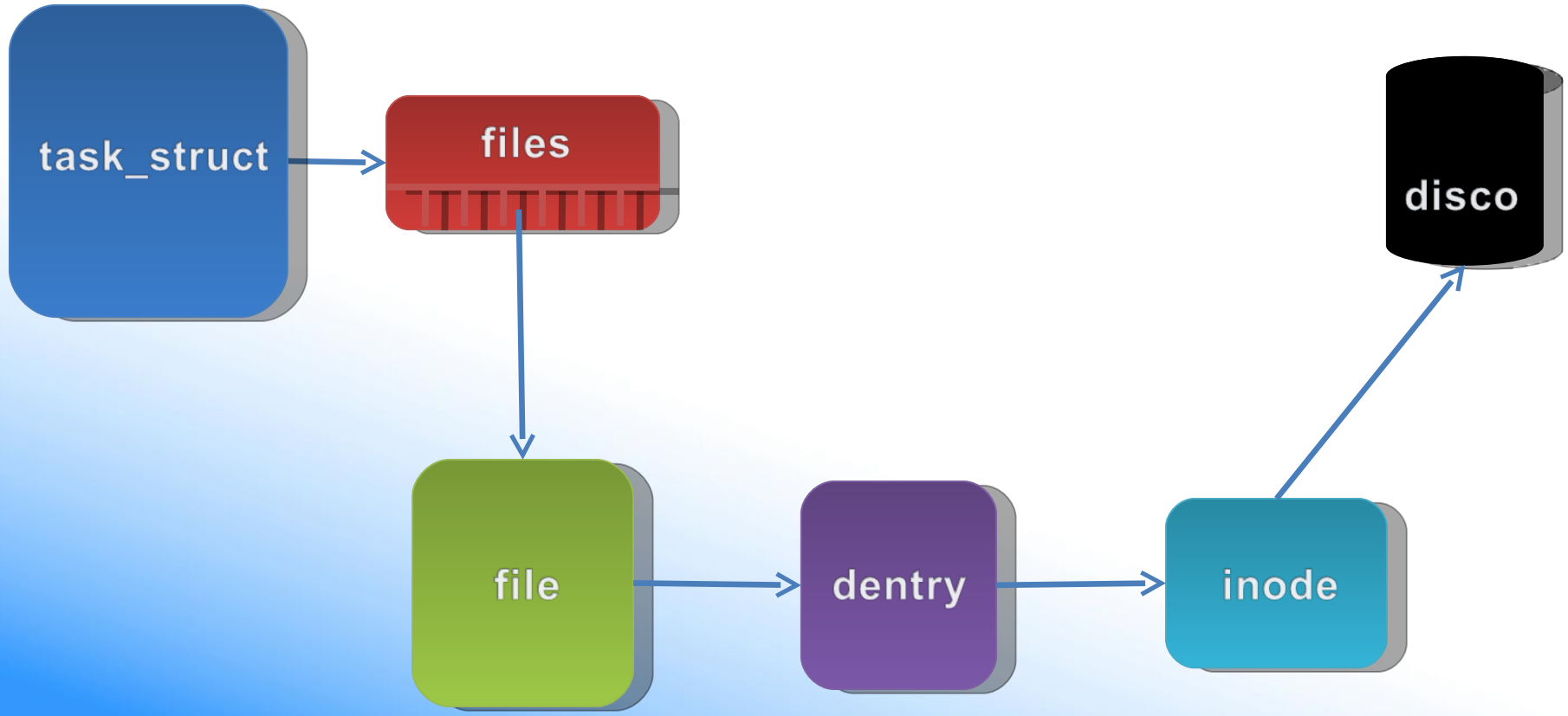
Índice de Contenidos

- Introducción
- Estructuras de datos
  - file
  - file_operations
  - files
  - nameidata
- Open
- Create
- Close

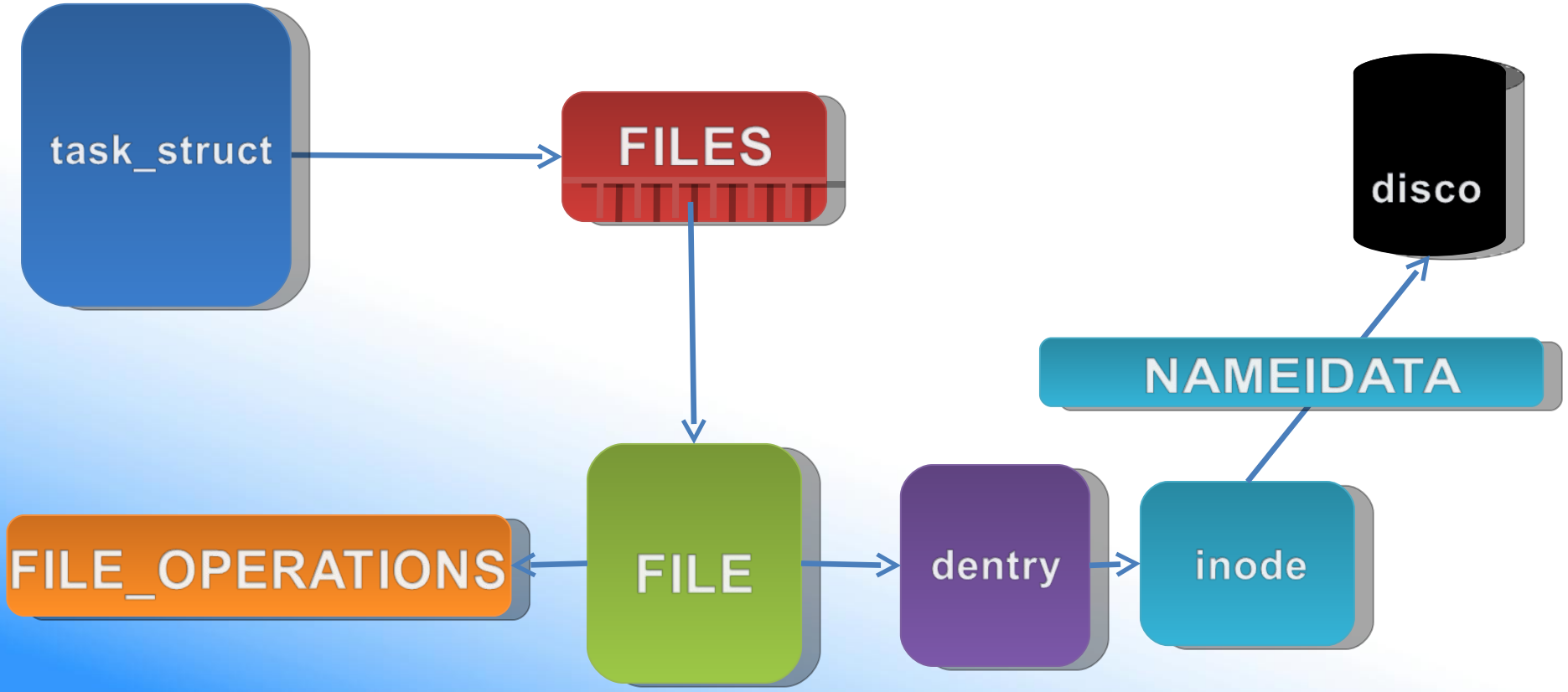
Introducción

- Un *fichero* es una abstracción muy importante en programación.
- Sirven para almacenar datos de forma permanente y ofrecen un pequeño conjunto de primitivas muy potentes
  - Abrir
  - Leer
  - avanzar puntero
  - Cerrar
  - etc.

Introducción



Introducción



FILE

- Describe cómo un proceso interacciona con el archivo que ha abierto.
- El objeto se crea cuando el archivo se abre
- Estos objetos no tienen correspondencia en disco

FILE

```
338struct file {
43        union {
44                struct list_head       fu_list;
45                struct rcu_head        fu_rcuhead;
46        } f_u;
47        struct path             f_path;
48#define f_dentry        f_path.dentry
49#define f_vfsmnt        f_path.mnt
50        const struct file_operations    *f_op;
51        atomic_long_t           f_count;
52        unsigned int            f_flags;
53        fmode_t                 f_mode;
54        loff_t                  f_pos;
55        struct fown_struct      f_owner;
56        const struct cred       *f_cred;
57        struct file_ra_state    f_ra;
59        u64                     f_version;
60#ifdef CONFIG_SECURITY
61        void                    *f_security;
62#endif
63        /* needed for tty driver, and maybe others */
64        void                    *private_data;
66#ifdef CONFIG_EPOLL
67        /* Used by fs/eventpoll.c to link all the hooks to this file */
68        struct list_head        f_ep_links;
69        spinlock_t              f_ep_lock;
70#endif /* #ifdef CONFIG_EPOLL */
71        struct address_space    *f_mapping;
72#ifdef CONFIG_DEBUG_WRITECOUNT
73        unsigned long f_mnt_write_state;
74#endif
75};
```

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

FILE_OPERATIONS

- Conjunto de operaciones asociadas al archivo
  - Abrir
  - Cerrar
  - ...
- Cuando el núcleo carga un inode, almacena un puntero a estas operaciones en file_operations
- Al abrir un archivo, se inicializa el campo f_op con la dirección almacenada en el inode para poder usar las operaciones posteriormente

FILE_OPERATIONS

```
1310struct file_operations {
1311        struct module *owner;     // Propietario de la estructura
1312        loff_t (*llseek) (struct file *, loff_t, int); // Para posicionar el puntero dentro del fichero
1313        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); // Operación de lectura
1314        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);  // Operación de escritura
...

1322        int (*mmap) (struct file *, struct vm_area_struct *); // Operación de mapeo
1323        int (*open) (struct inode *, struct file *); // Operación de abrir fichero
1324        int (*flush) (struct file *, fl_owner_t id);  // Flush
...
```

FILES

- Tabla de archivos actualmente abiertos
- Un proceso no puede tener más de NR_OPEN (~1024) descriptores de fichero
  - Se puede definir un límite inferior: rlim[RLIM_NOFILE]
- Fichero: /include/linux/fdtable.h

FILES

```
41struct files_struct {
•/*
43   * read mostly part
44   */
45      atomic_t count;   // Número de procesos que comparten esta tabla

46      struct fdtable *fdt; // Puntero a la tabla de descriptores de ficheros

47      struct fdtable fdtab; //Puntero a la matriz de puntero de objetos archivos

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; // Descriptores de archivos max. asignados más 1

53      struct embedded_fd_set close_on_exec_init;  // Conjunto inicial de descriptores a cerrar con exec()

54      struct embedded_fd_set open_fds_init;  // Conjunto inicial de descriptores de archivos

55      struct file * fd_array[NR_OPEN_DEFAULT];  // vector de punteros a objetos archivo

56};
```

NAMEIDATA

- Estructura intermedia para realizar las inicializaciones pertinentes a la hora de crear o abrir un fichero

```
 18struct nameidata {
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};
```

OPEN

- La llamada al sistema open() se utiliza para convertir una ruta/nombre de fichero en 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)

- **Pathname:** nombre del fichero a abrir.
- **Flags:** define el modo de apertura.
- **Mode:** especifica los permisos del usuario.

OPEN

– Flags

- **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.

– Modos

0[rwx][rwx][rwx]

- El primero indica permisos para usuarios.
- El segundo indica permisos para grupos.
- El tercero para otros.

OPEN

– La función del núcleo que implementa la llamada open es **sys_open**

```
1049 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
1050{
•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}
```

DO_SYS_OPEN

```
1027long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
1028{
•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 */
1038                                fd = PTR_ERR(f); /* descriptor del fichero.*/
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}
```

GET_UNUSED_FD_FLAGS

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

440int alloc_fd(unsigned start, unsigned flags)
441{
…
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;
…
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*/
```

GET_UNUSED_FD_FLAGS

```
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}
```

DO_FILP_OPEN

```
1630struct file *do_filp_open(int dfd, const char *pathname,
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;
```

DO_FILP_OPEN

```
1   * 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       }


1   /*
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);
```

DO_FILP_OPEN

```
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;
```

DO_FILP_OPEN

```
e debe realizar la Creación del fichero */
9       if (!path.dentry->d_inode) {
0               /*
1                * This write is needed to ensure that a
2                * ro->rw transition does not occur between
3                * the time when the file is created and when
4                * a permanent write count is taken through
5                * the 'struct file' in nameidata_to_filp().
6                */
7               error = mnt_want_write(nd.path.mnt);
8               if (error)
9                       goto exit_mutex_unlock;
0               error = __open_namei_create(&nd, &path, flag, mode);
1               if (error) {
2                       mnt_drop_write(nd.path.mnt);
3                       goto exit;
4               }
5                filp = nameidata_to_filp(&nd, open_flag); /* Realiza la apertura del fichero después de crearlo. */
6               mnt_drop_write(nd.path.mnt);
7                return filp; /*devuelve la estructura FILE
8       }



1779       filp = nameidata_to_filp(&nd, open_flag); /* Realiza la apertura del fichero sin que se haya creado porque



  1787       return filp; //Devuelve la estructura FILE
```

NAMEI_TO_FILP

```
/* 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
945path_put(&nd->path); 946        return filp;
947}
```

_DENTRY_OPEN

```
805static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
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       }


 825       f->f_mapping = inode->i_mapping; /* Asignación del inode. */
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. */


    839             error = open(inode, f); /* APERTURA DEL FICHERO */
```

CREATE

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

fd = CREAT (char *nombre, int permisos)

Parámetros:

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

- Función sys_create que llama a sys_open con los parámetros de creación de ficheros y con el modo de acceso al mismo

Flags: (O_CREAT | O_WRONLY | O_TRUNC)

CREATE

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

CLOSE

- 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
- **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

CLOSE

```
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;
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 */
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;
1145        return retval;
1147out_unlock:
1148        spin_unlock(&files->file_lock);
1149        return -EBADF;
1150}
```

FILP_CLOSE

```
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 páginas asociadas 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}
```