Francisco M. Santana Verona
José Évora Gómez

ÍNDICE

- Introducción
- Estructuras de datos
- Read
  - Prototipo
  - Funciones
- Write
  - Prototipo
  - Funciones
- Funciones auxiliares

Introducción

- ¿Cómo interactuar con el sistema de ficheros?
  1. Crear
  2. Abrir
  3. Leer / Escribir
  4. Cerrar
- ¿Qué veremos?

LEER / *ESCRIBIR* (Read / Write)

ESTRUCTURA FILE

Cuando un proceso de usuario abre un fichero el SVF crea una estructura file.

Los campos más importantes para el read y el write son:

```
struct file {
        …
    /* puntero al inodo de ese fichero */
    #define f_dentry   f_path.dentry
    /* conjunto de operaciones relacionadas con un fichero abierto */
    const struct file_operations *f_op;
    /*flag relacionado con los permisos sobre el fichero*/
    unsigned int f_flags;
    /* modo de apertura del fichero en la llamada open, conjunción
    * de FMODE_READ y FMODE_WRITE */
    fmode_t f_mode;
    /* posición actual desde el principio del fichero en bytes */
    loff_t f_pos;
        …
};
```

ESTRUCTURA FILE_OPERATIONS

Esta estructura define todas las operaciones que el núcleo ejecuta sobre ficheros. Los campos más destacados para el read y el write son:

```
struct file_operations
{
    …
   /* Posicionamiento en el fichero */
    loff_t (*llseek) (struct file *, loff_t, int);
    /*Lectura en el fichero */
    ssize_t (*read) (struct file *, char __user *, size_t,
    loff_t *);
     /*Escritura en el fichero */
     ssize_t (*write) (struct file *, const char __user *,
    size_t,     loff_t *);
    …

};
```

LA LLAMADA AL SISTEMA READ

- Nos permite leer de un archivo

read (fd,buf,count)

Fd = descriptor del fichero

Buf = puntero al buffer donde se recibirán los datos

Count = número máximo de bytes que se van a almacenar en el buffer

LA LLAMADA AL SISTEMA READ

- La llamada ***read*** devuelve el número de bytes que se han leído o -1 en caso de error.
- El tipo de error se almacena en ***errno***.
- Los tipos de errores 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.

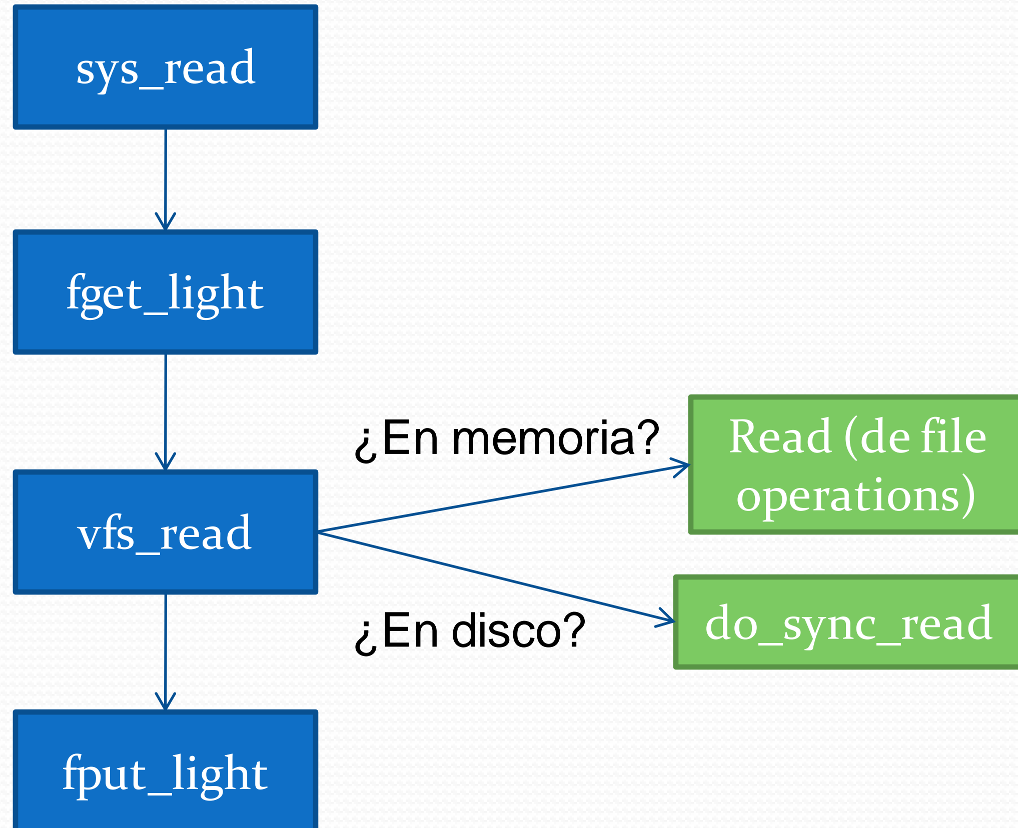
Flujo de ejecución del Read

Atiende la llamada al sistema Read.

Proporciona el descriptor del fichero.

Realiza comprobaciones. Efectúa la lectura.

Libera el descriptor del fichero.



FUNCIÓN SYS_READ

```
asmlinkage ssize_t sys_read(unsigned int fd, char
   __user * buf,       size_t count)
{
        struct file *file;

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

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

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

FUNCIÓN SYS_READ

Si fget_light devuelve un valor válido…

```
if (file) {
    /*Obtiene la posición actual en bytes desde el principio del fichero*/
        loff_t pos = file_pos_read(file);
    /*Llama a la función vfs_read que es la que hace la lectura*/
    ret = vfs_read(file, buf, count, &pos);
    /*actualiza la posición en la estructura file asociada al fd*/
    file_pos_write(file, pos);
    /*Liberamos la estructura de fichero asociada a fd*/
    fput_light(file, fput_needed);
}
return ret;
}
```

FUNCIÓN VFS_READ

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

/*Comprueba que la lectura se puede realizar*/

```
       if (!(file->f_mode & FMODE_READ))
          return -EBADF;
```
Descriptor no válido

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

/*Comprueba que la dirección del buffer es correcta*/

```
   if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
          return -EFAULT;
```
Dirección de buffer inválida

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

```
   ret = rw_verify_area(READ, file, pos, count);
```

FUNCIÓN VFS_READ

```
   if (ret>=0) {
        /*Si no esta bloqueada….*/
        count=ret;
        /*comprueba que tiene permiso de lectura*/
        if (file->f_op->read)
        /*se realiza la lectura desde memoria*/
                ret = file->f_op->read(file, buf, count, pos);
        /* sino fuerza lectura de disco */
        else
                ret = do_sync_read(file, buf, count, pos);
        /*Si se ha leído algo se notifica directorio padre que ha
        accedido al fichero */
        if (ret > 0) {
                fsnotify_access(file ->f_path.dentry);
                add_rchar(current,ret);
        }
        inc_syscr(current);
   }
   return ret;
}
```

do_sync_read

```
239 ssize_t do_sync_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos){
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;
245        init_sync_kiocb(&kiocb, filp);                              // Rellena la estructura kiocb con los campos de file
246         kiocb.ki_pos = *ppos;
247         kiocb.ki_left = len;
248
249         for (;;) {
// 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}
```

LA LLAMADA AL SISTEMA WRITE

- Nos permite escribir en un archivo

write (fd,buf,count)

Fd = descriptor del fichero

Buf = puntero al buffer donde se encuentran los datos a escribir

Count = número máximo de bytes que contiene el buffer

LA LLAMADA AL SISTEMA WRITE

- La llamada ***write*** devuelve el número de bytes que fueron transferidos correctamente o -1 en caso de error.
- El tipo de error se almacena en ***errno***.
- Los tipos de errores son:

TIPOS DE ERROR	
Valor de la variable errno	Causa del Error
EBADF	El descriptor de fichero no es válido
EFAULT	La dirección del buffer no es válida
EINTR	La llamada al sistema ha sido interrumpida
EINVAL	No se puede escribir en el fichero
EIO	Error de Entrada/Salida
EISDIR	El descriptor 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

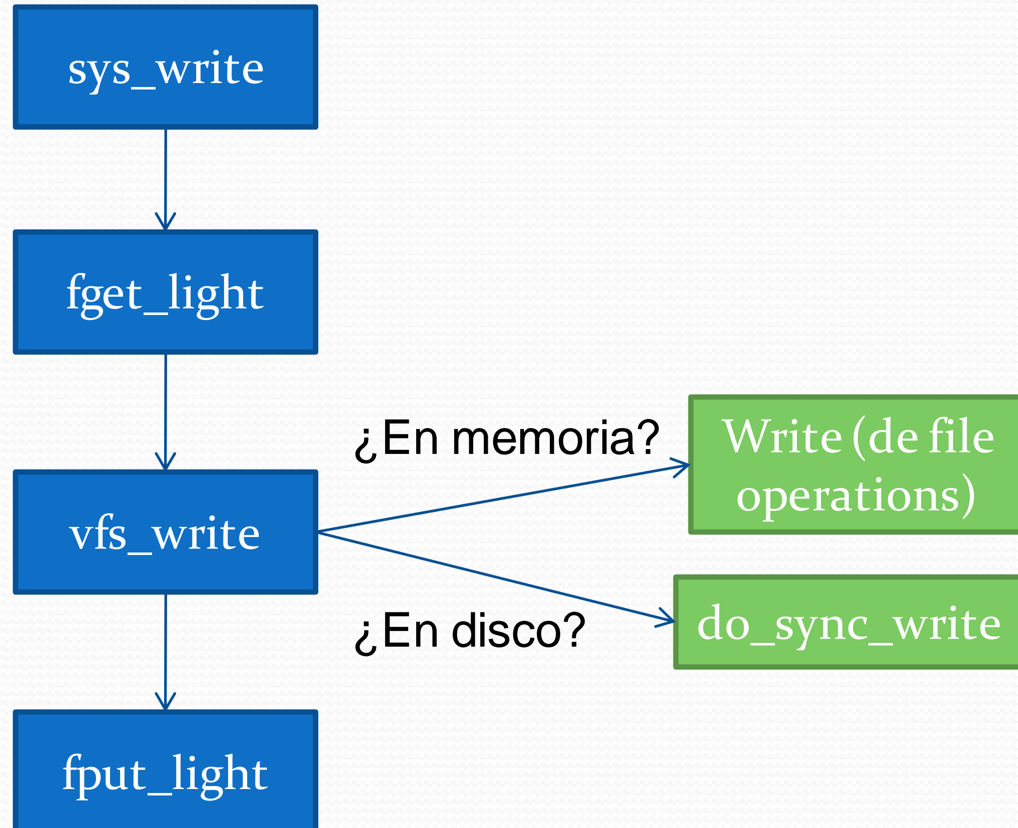
Flujo de ejecución del Write

Atiende la llamada al sistema Write.

Proporciona el descriptor del fichero.

Realiza comprobaciones. Efectúa la escritura.

Libera el descriptor del fichero.



sys_write (*fs/read_write.c*)

```
asmlinkage ssize_t sys_write(unsigned int fd, const char __user *     buf,
size_t count)
{
    Estructura file asociada a fd
    struct file *file;

    ret será la variable donde se almacene el valor de retorno. Se inicializa a "el fd
    especificado no es válido"
    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
    file = fget_light(fd, &fput_needed);
```

sys_write (*fs/read_write.c)*

Si fd ha sido válido file debería tener una dirección asociada

```
if (file) {
        Se obtiene la posición actual desde el principio del fichero en bytes
        loff_t pos = file_pos_read(file);

        Se llama a la función que realmente realiza la escritura de los datos en el fichero
        ret = vfs_write(file, buf, count, &pos);

        Se actualiza la posición actual en el fichero
        file_pos_write(file, pos);

        Se libera la estructura file asociada a fd
        fput_light(file, fput_needed);
}
return ret;
}
```

vfs_write (*fs/read_write.c)*

```
ssize_t vfs_write(struct file *file, const char __user *buf,      size_t count, loff_t
    *pos)
{
    ret será la variable donde se almacene el valor de retorno
    ssize_t ret;

    Se comprueba que se puede realizar la escritura del fichero
    if (!(file->f_mode & FMODE_WRITE))
            return -EBADF;                    error: fd especificado no es válido
    if (!file->f_op || (!file->f_op->write && !file->f_op->aio_write))
            return -EINVAL;                   error: no es posible realizar la escritura
    if (unlikely(!access_ok(VERIFY_READ, buf, count)))
            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)
    ret = rw_verify_area(WRITE, file, pos, count);
```

vfs_write (*fs/read_write.c)*

```
Si no estaba bloqueada… Se comprueba si se tiene permiso de escritura
ret = security_file_permission (file, MAY_WRITE);
if (ret>=0) {  Si tiene permiso de escritura…
        count=ret;
        if (file->f_op->write)
                Se escriben los datos en memoria
                ret = file->f_op->write(file, buf, count, pos);
        else
                Se fuerza la escritura en disco
                ret = do_sync_write(file, buf, count, pos);
        if (ret > 0) {
                Se notifica al directorio padre la modificación del archivo
        fsnotify_modifyt (file->f_path, dentry);
                Se incrementa el número de bytes escritos
        add_wchar(current,ret);
        }
        Se incrementa el número de llamadas al sistema write
        inc_syscw(current);
}
return ret;
}
```

do_sync_write

```
239 ssize_t do_sync_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos){
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;
245        init_sync_kiocb(&kiocb, filp);                        // Rellena la estructura kiocb con los campos de file
246         kiocb.ki_pos = *ppos;
247         kiocb.ki_left  = len;
248
249         for (;;) {
// llama aio_write  pasándole iovec y kiocb.  Retorna un valor con los  bytes leidos que es lo que retorna la
      función
250                 ret = filp->f_op->aio_write(&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}
```

Funciones auxiliares

Las funciones auxiliares comunes a ambas llamadas son:

-fget_light

-fput_light

-rw_verify_area

fget_light (linux/fs/file_table.c)

```
struct file fastcall *fget_light(unsigned int fd, int    *fput_needed)
{
    struct file *file;
    struct files_struct *files = current->files;
    *fput_needed = 0;

    Si la zona está bloqueada por el proceso actual...
    if (likely((atomic_read(&files->count) == 1))) {
            Obtiene dirección estructura de fichero
            file = fcheck_files(files, fd);179

    } else { En caso contrario ...
            ... La bloquea y la coge
        rcu_read_lock();
        file = fcheck_files(files, fd);
            if (file) {
                    if (atomic_long_inc_not_zero(&file->f_count))
                            *fput_needed = 1;
                    else
                            file = NULL;
          }
          rcu_read_unlock();
     }
     return file; }
```

fcheck_files (include/linux/file.h)

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

Puntero a estructura de fichero a devolver (inicialmente nula)

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

Comprueba que el f des válido

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

Devuelve estructura fichero asociada a fd

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

fput_light (include/linux/file.h)

```
static inline void fput_light(struct file *file, int fput_needed)
{
        if (unlikely(fput_needed))
                fput(file);
}
```

rw_verify_area (fs/read_write.c)

- Encargada de comprobar que la sección a leer del fichero no esté bloqueada.

- Bloquea según el tipo de operación que se vaya a efectuar:

  - - Si es una lectura:
    - sólo se bloquea la zona que se va a leer.

  - -Si es una escritura:
    - se bloquea todo el fichero.

rw_verify_area (fs/read_write.c)

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

    inode = file->f_path.dentry->d_inode;

    Se comprueba que el tamaño de los datos a leer o escribir no debe ser mayor que
    tamaño máximo permitido en estructura fichero
    if (unlikely(count > file->f_maxcount))
            return retval; Va a la etiqueta Einval en caso de error

    Se asigna a pos loa posición actual del fichero
    pos = *ppos;

    Comprueba que no se intenta acceder fuera del fichero
    if (unlikely((pos < 0) || (loff_t) (pos + count) < 0))
            return retval; Va a la etiqueta Einval en caso de error
```

rw_verify_area (fs/read_write.c)

Comprueba que no se intenta acceder fuera del fichero

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

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.

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

Se devuelve el resultado de la llamada a Write.

```
        if (retval)
                return retval;
        return count > MAX_RW_COUNT ? MAX_RW_COUNT : count;
}
```

Bibliografía

- Linux Cross Reference (http://lxr.linux.no)
- Understanding the Linux Kernel (3ª Edición) Daniel P. Bovet, Marco Cesati Ed. O'Reilly [2005]