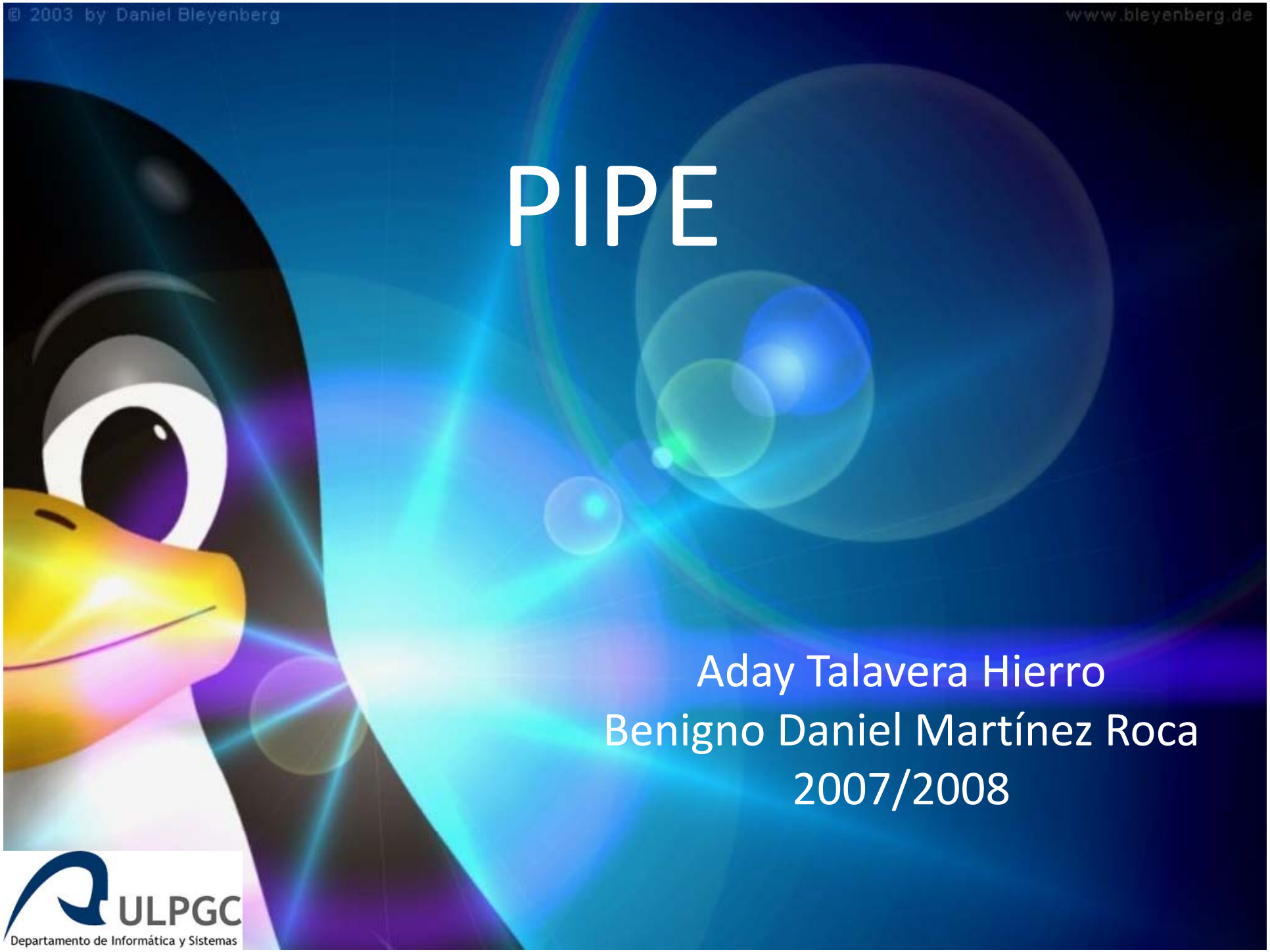


PIPE

Aday Talavera Hierro
Benigno Daniel Martínez Roca
2007/2008

ULPGC
Departamento de Informática y Sistemas

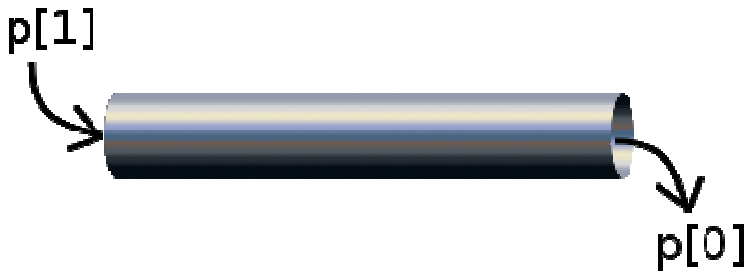
Índice

Introducción

Pipe es un mecanismo de comunicación entre procesos, utilizado en los sistemas Unix.

La transmisión de datos entre procesos se efectúa a través de un canal .

Los datos escritos en un extremo del canal se leen en el otro extremo del canal, actuando como si se tratase de una tubería.



Tipos de pipe

Existen dos tipos:

1. Sin nombre o **Pipe**.
2. Con nombre o **Fifo**.

❑ **PIPE:** Se crea por un proceso y la transmisión de los descriptores sólo se hace por herencia. Este mecanismo es restrictivo porque sólo permite la comunicación entre procesos cuyo antecesor común es el creador de la tubería.

❑ **FIFO:** Permiten salvar la restricción anterior porque se manipulan exactamente como archivos en lo que respecta a las operaciones. Esto es posible porque existen físicamente en el sistema de archivos.

Estructuras de datos

Las estructuras de datos del pipe están en el archivo:

<include/linux/pipe_fs_i.h>

Las operaciones están en el archivo:

<linux/fs/pipe.c>

Para cada pipe se crea un objeto *i-nodo* y dos objetos archivo, uno para la lectura y otro para la escritura.

Además cada pipe tiene su propio conjunto de 16 buffers. Estos 16 buffers se pueden ver como un gran buffer circular en el cual los procesos se escriben y añaden datos, mientras que los procesos que leen los van quitando.

Estructura: pipe_inode_info

Cuando un objeto *i-nodo* se refiere a un pipe, el campo *i_pipe del i-nodo* apunta a una estructura *pipe_inode_info* como la siguiente:

```
struct pipe_inode_info {
        wait_queue_head_t wait;
        unsigned int nrbufs, curbuf;
        struct page *tmp_page;
        unsigned int readers;
        unsigned int writers;
        unsigned int waiting_writers;
        unsigned int r_counter;
        unsigned int w_counter;
        struct fasync_struct *fasync_readers;
        struct fasync_struct *fasync_writers;
        struct inode *inode;
        struct pipe_buffer bufs[PIPE_BUFFERS];
};
```

Estructura: pipe_inode_info

Tipo	Campo	Descripción
wait_queue_head_t	wait	Cola wait del pipe.
unsigned int	nrbufs	Número de buffers que contienen datos a leer.
unsigned int	curbuf	Índice del primer buffer que contiene datos para leer.
struct page *	tmp_page	Puntero a un marco de pagina.
unsigned int	readers	Número de procesos que pueden leer.
unsigned int	writers	Número de procesos que pueden escribir.
unsigned int	waiting_writers	Número de procesos escritores esperando en la cola *wait*
unsigned int	r_counter	Como *readers* para procesos que leen FIFO.
unsigned int	w_counter	Como *writers* para procesos que escriben FIFO.
struct fasync_struct *	fasync_readers	Se usa para notificaciones asíncronas de E/S a través de señales.
struct fasync_struct *	fasync_writers	Se usa para notificaciones asíncronas de E/S a través de señales.
struct inode *	inode	Puntero a una estructura i-nodo
struct pipe_buffer	bufs[PIPE_BUFFERS]	Vector de buffers de pipe (16).

Estructura: pipe_buffer

Pipe_buffer definido en archivo: *</define/kernel/pipe_fs_i.h.c>*

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Estructura: pipe_buffer

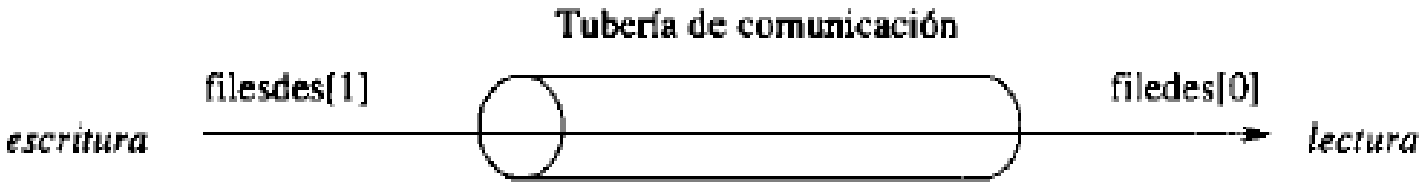
Tipo	Campo	Descripción
struct page *	page	Descriptor del marco de página dentro del buffer.
unsigned int	offset	Posición actual de los datos dentro del marco de página
unsigned int	len	Longitud de los datos significativos del pipe_buffer.
const struct pipe_buf_operations *	ops	Tabla de operaciones
unsigned int	flags	Flags

Creación de Pipe

La llamada al sistema ***pipe*** permite crear un pipe. Para efectuar esta operación, hay que pasar cómo parámetro una tabla de dos enteros.

int pipe (int filedes [2]);

La tabla de enteros contendrá el descriptor de lectura (filedes[0]) y de escritura (filedes[1]) de la tubería creada.



pipe

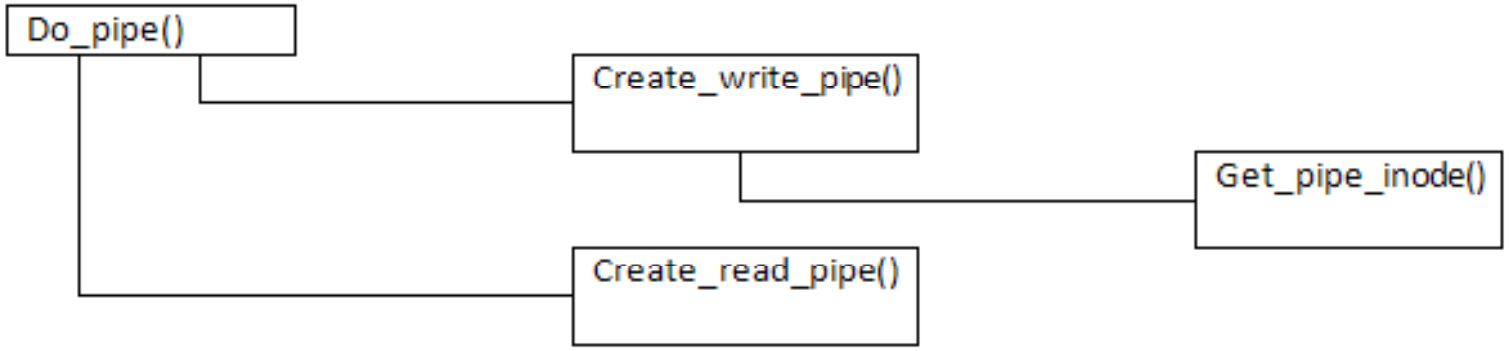
Llamada al sistema: sys_pipe

La llamada al sistema ***pipe*** es tratada por la llamada al sistema ***sys_pipe***, la cual invoca la función ***do_pipe***.

```
asmlinkage int sys_pipe(nabi_no_regargs volatile struct pt_regs regs)
{
        int fd[2];
        int error, res;
        error = do_pipe(fd);
        if (error) {
                res = error;
                goto out;
        }
        regs.regs[3] = fd[1];
        res = fd[0];
out:
        return res;
}
```

Creación de Pipe

La función ***do_pipe*** es la encarga de crear el Pipe. La inicialización y creación del *i-nodo* (función *get_pipe_inode*) no es realiza directamente por la función ***do_pipe***, sino que esta invoca a la función ***create_write_pipe****, quien se encarga de crear el i-nodo y crear un objeto de archivo de escritura.*



Creación de Pipe

La función do_pipe crear el pipe, tanto para poder realizar la lectura como la escritura sobre el mismo. Los descriptores de fichero para poder hacerlo se almacenan en fd.

```
int do_pipe(int *fd)
{
        struct file *fw, *fr;
        int error;
        int fdw, fdr;
        /*Se crea el pipe para procesos que quieren escribir*/
        fw = create_write_pipe();
        if (IS_ERR(fw))
                return PTR_ERR(fw);
        /* Se crea el pipe para procesos que quieren leer*/
        fr = create_read_pipe(fw);
        error = PTR_ERR(fr);
        if (IS_ERR(fr))
                goto err_write_pipe;
        error = get_unused_fd();
```

Creación de Pipe

do_pipe

```
if (error < 0)
        goto err_read_pipe;
fdr = error;

error = get_unused_fd();
if (error < 0)
        goto err_fdr;
fdw = error;

error = audit_fd_pair(fdr, fdw);
if (error < 0)
        goto err_fdw;

/* Se almacenan los descriptores de fichero de los lectores y
   escritores en la estructura file*/
fd_install(fdr, fr);
fd_install(fdw, fw);
```

pipe

Creación de Pipe

```
/* Se asignan los descriptores de ficheros de los procesos lectores y
escritores*/
        fd[0] = fdr;
        fd[1] = fdw;
        return 0;
err_fdw:
        put_unused_fd(fdw);
err_fdr:
        put_unused_fd(fdr);
err_read_pipe:
        dput(fr->f_dentry);
        mntput(fr->f_vfsmnt);
        put_filp(fr);
err_write_pipe:
        free_write_pipe(fw);
        return error;
}
```

Creación de Pipe

```
struct file *create_write_pipe(void)
{
        int err;
        struct inode *inode;
        struct file *f;
        struct dentry *dentry;
        struct qstr name = { .name = "" };
        err = -ENFILE;
        inode = get_pipe_inode(); /* Se crea el inode con la
                                     estructura pipe*/
        if (!inode)
                goto err;
        err = -ENOMEM;
        /*En dentry se almacenan referencias a directorios usados.
        D_alloc se asigna una dentry a un nombre */
        dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &name);
```

Creación de Pipe

```
if (!dentry)
        goto err_inode;
/* Se asignan las operaciones del pipe*/
dentry->d_op = &pipefs_dentry_operations;
dentry->d_flags &= ~DCACHE_UNHASHED;
/* Se añade una dentry a la tabla de hash del inode, actualizando
   el campo d_inode de la dentry*/
d_instantiate(dentry, inode);
err = -ENFILE;
/* Se asigna e inicializa la estructura file con los parámetros*/
f = alloc_file(pipe_mnt, dentry, FMODE_WRITE, write_pipe_fops);
 if (!f)
        goto err_dentry;
```

Creación de Pipe

create_write_pipe

```
        /* Se asignan campos para la escritura*/
        f->f_mapping = inode->i_mapping;

        f->f_flags = O_WRONLY;
        f->f_version = 0;

        return f;
err_dentry:
        dput(dentry);
err_inode:
        free_pipe_info(inode);
         iput(inode);
err:
        return ERR_PTR(err);
}
```

Creación de Pipe

```
struct file *create_read_pipe(struct file *wrf)
{    /* wrf es la estructura creada mediante el create_write_pipe.
                Toma una fichero que no está siendo usado.*/
     struct file *f = get_empty_filp();
     if (!f)
             return ERR_PTR(-ENFILE);
     /* Como se apoya en el parámetro pasado, se copian
                              algunos campos */
     f->f_path.mnt = mntget(wrf->f_path.mnt);
     f->f_path.dentry = dget(wrf->f_path.dentry);
     f->f_mapping = wrf->f_path.dentry->d_inode->i_mapping;
     f->f_pos = 0;
     /* Indica en el campo f_flags que se trata de lectura. */
     f->f_flags = O_RDONLY;
      /* Almacena una estructura con las operaciones de lectura en pipe
           y el modo*/
     f->f_op = &read_pipe_fops;
     f->f_mode = FMODE_READ;
     f->f_version = 0;
     return f;
}
```

Creación de Pipe

En un primer momento se crea un nuevo *i-nodo* en el sistema de archivos, y éste tiene enlaces nulos. De este modo, este archivo no es visible en el sistema de archivos, y la única manera de acceder a él es teniendo un descriptor del fichero del mismo. Asimismo, se crea la estructura de Pipe (*pipe_inode_info*). *Todo esto lo hace get_pipe_inode*

```
static struct inode * get_pipe_inode(void)
{
        /*Crea una una nueva estructura inode*/

        struct inode *inode = new_inode(pipe_mnt->mnt_sb);

        /*Crea la estructura a usar por el pipe*/

        struct pipe_inode_info *pipe;

        if (!inode)

                goto fail_inode;
```

Creación de Pipe

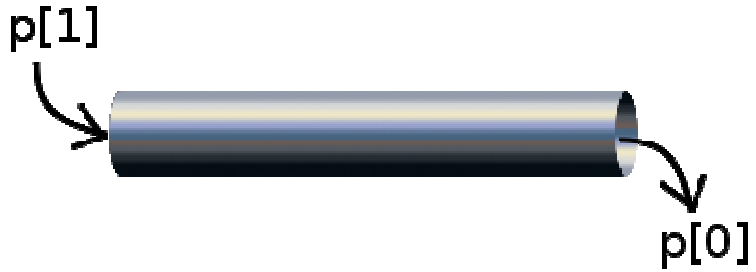
```
        /*Mediante alloc_pipe se inicializan campos de la estructura
            pipe_inodo_info y se le asocia al  i-nodo creado.*/
pipe = alloc_pipe_info(inode);
if (!pipe)
        goto fail_iput;
      /* Asigna la dirección de la estructura de Pipe al campo del i-nodo
      i_pipe. */
inode->i_pipe = pipe;

        /* Inicializa los campos readers y writers. */
pipe->readers = pipe->writers = 1;
      /* Almacena una estructura (file_operations) con las operaciones del
      Pipe.  */
inode->i_fop = &rdwr_pipe_fops;
```

Creación de Pipe

```
    /*  Inicialización de campos del i-nodo. */
    inode->i_state = I_DIRTY;

    inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;

    inode->i_uid = current->fsuid;

    inode->i_gid = current->fsgid;

    inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;

    return inode
fail_iput:

    iput(inode);
     fail_inode:
    return NULL;
}
```

Destrucción de Pipe

Cada proceso posee un descriptor de lectura y un descriptor de escritura en la misma tubería. Los procesos deben elegir un sentido de transmisión, utilizando solo uno de los dos descriptores. Los descriptores inutilizados pueden cerrarse por la llamada al sistema ***close***. Cada proceso debe cerrar un descriptor antes de usar el otro.



Siempre que un proceso invoca la llamada al sistema ***close*** sobre un descriptor, dependiendo si el archivo es asociado con lectura o con escritura, se invoca la función *pipe_read_release()* o *pipe_write_release()*. Ambas funciones invocan a *pipe_release().*

Destrucción de Pipe

pipe_read_release
pipe_write_release

```
static int pipe_read_release(struct inode *inode, struct file *filp)
{
        pipe_read_fasync(-1, filp, 0);
        return pipe_release(inode, 1, 0);
}


static int pipe_write_release(struct inode *inode, struct file *filp) {
        pipe_write_fasync(-1, filp, 0);
        return pipe_release(inode, 0, 1);
}
```

Destrucción de Pipe

pipe_release

```
static int pipe_release(struct inode *inode, int decr, int decw)
{
        struct pipe_inode_info *pipe;

        mutex_lock(&inode->i_mutex);
        pipe = inode->i_pipe;
        pipe->readers -= decr;
        pipe->writers -= decw;
        /*Si ya no hay escritores ni lectores se librea el inode*/
        if (!pipe->readers && !pipe->writers) {
                free_pipe_info(inode);
        } else {
        /*Sino, se fuerza a que terminen los escritores y lectores*/
                wake_up_interruptible_sync(&pipe->wait);
                kill_fasync(&pipe->fasync_readers, SIGIO,
POLL_IN);
                kill_fasync(&pipe->fasync_writers, SIGIO,
POLL_OUT);
        }
        mutex_unlock(&inode->i_mutex);

        return 0;
}
```

Escritura en pipe

Un proceso que desea escribir datos en un pipe utiliza la llamada al sistema *write() sobre el descriptor de fichero p[1]*.

El núcleo invoca la función *pipe_write* de acuerdo con el *struct file_operations write_pipe_fops.*

```
static const struct file_operations write_pipe_fops ={
        .llseek = no_llseek,
        .read = bad_pipe_r, ← no se puede leer
        .write = do_sync_write,
        .aio_write = pipe_write,
        .poll = pipe_poll,
        .unlocked_ioctl = pipe_ioctl,
        .open = pipe_write_open,
        .release = pipe_write_release,
        .fasync = pipe_write_fasync,
};
```

Lectura en pipe

Un proceso que desea leer datos de un pipe utiliza la llamada al sistema *read() sobre el descriptor de fichero p[0]*.

El núcleo invoca la función *pipe_read* de acuerdo con el *struct file_operations read_pipe_fops.*

```
static const struct file_operations read_pipe_fops = {
        .llseek = no_llseek,
        .read = do_sync_read,
        .aio_read = pipe_read,
        .write = bad_pipe_w, ← no se puede escribir
        .poll = pipe_poll,
        .unlocked_ioctl = pipe_ioctl,
        .open = pipe_read_open,
        .release = pipe_read_release,
        .fasync = pipe_read_fasync,
};
```

pipe_write (i)

```
pipe_write(struct kiocb *iocb, const struct iovec * _iov, unsigned long nr_segs, loff_t
ppos)
{
        ssize_t ret;  //Valor de retorno
        int do_wakeup; //Flag para despertar a los lectores
        size_t total_len; //Tamaño de los datos que queremos escribir

         //Se obtiene el semáforo
        mutex_lock(&inode->i_mutex);

        //Si no hay lectores, enviar señal SIGPIPE y retornar
        if (!pipe->readers) {
                send_sig(SIGPIPE, current, 0);
                ret = -EPIPE;
                goto out;
        }
```

pipe_write (ii)

```
//Si hay espacio libre en el último buffer, se escribe en él
//¡En este caso no se crea ningún buffer nuevo!
chars = total_len & (PAGE_SIZE-1);
if (pipe->nrbufs && chars != 0) {
        {…} //Obtiene datos del buffer
        //Si se pueden combinar los buffers, se escribe
        if (ops->can_merge && offset + chars <= PAGE_SIZE) {
                //Mapea el marco de página del buffer
                addr = ops->map(pipe, buf, atomic);
                //Copia del espacio de usuario los bytes
                error = pipe_iov_copy_from_user(offset + addr, iov, chars, atomic);
                //Desmapea el marco de página del buffer
                ops->unmap(pipe, buf, addr);
                {…} //Actualiza datos del buffer
                //Si se han escrito todos los bytes, retornar
                total_len -= chars;
                if (!total_len)
                        goto out;
        }
}
```

pipe_write (iii)

```
//Bucle infinito: en este bucle se crearán nuevos buffers siempre
for (;;) {
//Si hay algún buffer libre
bufs = pipe->nrbufs;
if (bufs < PIPE_BUFFERS) {
        {…} //Obtiene datos del buffer
        //Despertar a los lectores ya que se van a escribir datos
        do_wakeup = 1;
        //Mapea el marco de página del buffer
        kmap(page);
        //Copia del espacio de usuario los bytes
        error = pipe_iov_copy_from_user(src, iov, chars,atomic);
        //Desmapea el marco de página del buffer
        kunmap(page);
        //Inserta la página en el vector de buffers
        buf->page = page;
        {…} //Se actualizan varios datos del buffer y se incrementa el nº de buffers
        //Se actualiza el contador de bytes escritos
        total_len -= chars;
```

pipe_write (iv)

```
        //Si se han escrito todos los bytes, retornar
        if (!total_len) break;
        //Si quedan buffers libres, continuar escribiendo
        if (bufs < PIPE_BUFFERS) continue;
        //Si es necesario, despertar a los lectores
        if (do_wakeup) {
                wake_up_interruptible_sync(&pipe->wait);
                kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
                do_wakeup = 0;
        }
        //Se duerme este escritor en espera de espacio en el buffer
        pipe->waiting_writers++;
        pipe_wait(pipe);
        pipe->waiting_writers--;
} //Fin del bucle infinito
out:   //Libera el semáforo
        mutex_unlock(&inode->i_mutex);
        //Despierta a los lectores si es necesario
        return ret;
}
```

pipe_read (i)

```
pipe_read(struct kiocb *iocb, const struct iovec * _iov,unsigned long nr_segs, loff_t pos)
{
        ssize_t ret;  //Valor de retorno
        int do_wakeup; //Flag para despertar a los lectores
        size_t total_len; //Tamaño de los datos que queremos escribir
//Coge el semáforo
mutex_lock(&inode->i_mutex);
//Bucle infinito
for (;;) {
//Si hay bytes para leer
int bufs = pipe->nrbufs;
if (bufs){
        {…} //Obtiene datos del buffer
        //Mapea el marco de página del buffer
        addr = ops->map(pipe, buf, atomic);
        //Copia al espacio de usuario los bytes
        error = pipe_iov_copy_to_user(iov, addr + buf->offset, chars, atomic);
        //Desmapea el marco de página del buffer
        ops->unmap(pipe, buf, addr);
```

pipe_read (ii)

```
{..} //Actualiza datos del buffer
//Si el buffer está vacío, liberarlo y activar el flag de avisar escritores
if (!buf->len) {
        ops->release(pipe, buf);
        do_wakeup = 1;
}
//Si no falta nada por leer, lectura finalizada, terminar el bucle
if (!total_len) break;
//Si queda por copiar y se puede seguir copiando, iterar de nuevo
if (bufs) continue;
//Si no hay escritores, terminar
if (!pipe->writers) break;
//Si se activo el flag, despertar a los escritores
if (do_wakeup) {
        wake_up_interruptible_sync(&pipe->wait);
        kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
}
//Duerme al lector
pipe_wait(pipe);
```

pipe_read (iii)

```
} //Fin del bucle infinito
//Libera el semáforo
mutex_unlock(&inode->i_mutex);
//Despierta a los escritores dormidos
if (do_wakeup) {
        wake_up_interruptible_sync(&pipe->wait);
        kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
}
return ret;
}
```

pipe_wait

```
void pipe_wait(struct pipe_inode_info *pipe){
        //Añade el proceso a la cola de espera del pipe
        prepare_to_wait(&pipe->wait, &wait, TASK_INTERRUPTIBLE);
        //Libera el semáforo
        if (pipe->inode)
                mutex_unlock(&pipe->inode->i_mutex);
        //Invoca al planificador
        schedule();
        //Retira el proceso de la cola de espera del pipe
        finish_wait(&pipe->wait, &wait);
        //Adquiere el semáforo
        if (pipe->inode)
                mutex_lock(&pipe->inode->i_mutex);
}
```

Funciones de copia entre buffers

Copia los datos desde el buffer de usuario al buffer del pipe:
pipe_iov_copy_from_user(void *to, struct iovec *iov, unsigned long len,int atomic)

Copia los datos desde el buffer del pipe al buffer del usuario:
pipe_iov_copy_to_user(struct iovec *iov, const void *from, unsigned long len,int atomic)