LECCIÓN 26: PIPE

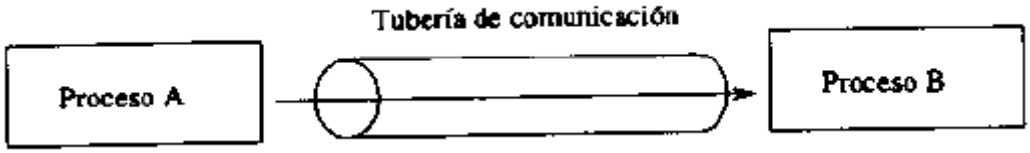
- Versión del núcleo de Linux analizada: 2.6.25.4

LECCIÓN 26: PIPE

26.1 Introducción

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

El concepto de Pipe también es conocido como Tubería. La transmisión de datos entre procesos se efectúa a través de un canal de comunicación: los datos escritos en un extremo del canal se leen en el otro extremo del canal, actuando como si se tratase de una tubería.



-Comunicación entre procesos mediante Pipe o Tubería-

Se puede comparar el Pipe con una cola de caracteres de tipo FIFO (First In First Out, primero en entrar, primero en salir) para conseguir un concepto en su funcionamiento.

Pipe no ofrece una comunicación estructurada. La lectura de los datos es independiente de la escritura, por lo que no es posible a nivel de las llamadas al sistema conocer el tamaño, el emisor o el destinatario de los datos contenidos en el Pipe. Por el contrario, una ventaja de este método de comunicación es que permite leer de una sola vez datos que se han podido escribir en varias ocasiones.

El método utilizado para la construcción de un pipe, es el uso de descriptores de ficheros (uno para lectura y otro para escritura) y de un área intermedia de memoria, en la cual se alojará el Pipe propiamente dicho.

26.2 Tuberías anónimas y tuberías con nombre

La gestión de las tuberías está integrada en el sistema de archivos. El acceso a las tuberías se realiza por los descriptores de entrada/salida (un descriptor para la lectura en la tubería y un descriptor para la escritura en la tubería).

Existen dos tipos de Tuberías: tuberías sin nombre o Pipe, y tuberías con nombre o Fifo.

El primer tipo de tuberías es la tubería anónima, que corresponde con la denominación de PIPE. Se crea por un proceso y la transmisión de los descriptores sólo se hace por herencia hacia sus descendientes. Este mecanismo es restrictivo porque sólo permite la comunicación entre procesos cuyo antecesor común es el creador de la tubería.

Las tuberías con nombre, correspondientes a la denominación de FIFO. Permiten salvar la restricción anterior porque se manipulan exactamente como archivos en lo que respecta a las operaciones de apertura, cierre, lectura y escritura. Esto es posible porque existen físicamente en el sistema de archivos. Es posible crear una tubería con nombre bajo el intérprete de mandatos mediante el mandato *mkfifo*.

Este documento se centra en las tuberías sin nombre.

Para comprender el funcionamiento de la implementación de las tuberías es necesario haber consultado el capítulo sobre el sistema de archivos, ya que una tubería es un archivo particular del sistema de archivos.

22.3 Creación de una tubería mediante llamadas al sistema

Existe una llamada al sistema estrictamente dedicada a la creación de tuberías anónimas: **pipe**. Las tuberías con nombre se crean utilizando la función **mkfifo**, o bien la llamada al sistema **mknod**.

Creación de una tubería anónima

La llamada **pipe** permite crear una tubería anónima. Para efectuar esta operación desde un programa de usuario se tiene la siguiente interfaz, a la que hay que pasar como parámetro una tabla de dos enteros.

```
int pipe (int filedes[2]);
```

Si la llamada se efectúa con éxito, la tabla de enteros contendrá el descriptor de lectura (filedes[0]) y de escritura (filedes[1]) de la tubería creada. Esta llamada al sistema puede generar los errores siguientes:

Error	Significado
EFAULT	La tabla pasada como parámetro no es válida
EMFILE	Se ha alcanzado el número máximo de archivos abiertos para el proceso actual
ENFILE	Se ha alcanzado el número máximo de archivos abiertos en el sistema

Es posible utilizar funciones de alto nivel para las operaciones de lectura y escritura como *printf, sprintf, scanf...* Sin embargo, estas operaciones utilizan memorias intermedias, por lo que todo carácter escrito no está inmediatamente disponible en el otro extremo de la tubería, a menos que cada operación de escritura vaya seguida de una llamada a *fflush*.

Los descriptores inutilizados pueden cerrarse por la llamada al sistema **close**. Una vez cerrado, el descriptor no permite ya acceder a al tubería.

De modo predeterminado, la lectura en una tubería es bloqueadora. El proceso lector queda bloqueado hasta que lee la cantidad de datos precisa en la llamada **read**. Dos procesos comunicantes pueden adoptar pues un protocolo para evitar bloquearse mutuamente.

Creación de una tubería y ejecución de un programa

Es frecuente utilizar tubería para comunicarse con un programa externo. Se proporciona dos funciones de biblioteca para simplificar esta tarea:

```
#include <stdio.h>

FILE *popen (const char *command, const char *type);

int pclose (FILE *stream);
```

La función **popen** crea una tubería, y luego un proceso hijo. Este proceso ejecuta el mandato especificado por el parámetro **mandato**. Se devuelve un descriptor de entradas/salidas correspondiente a la tubería, según el valor de parámetro **type**:

- Si **type** es la cadena <<r>>, el descriptor es accesible en lectura, y permite acceder a la salida estándar del mandato.
- Si **type** es la cadena <<w>>, el descriptor es accesible en escritura, y permite acceder a la entrada estándar del mandato.

La función **pclose** puede llamarse para cerrar la tubería, y utiliza la función **fclose** para el cierre. Luego provoca la finalización del proceso hijo asociado, que había sido creado por la llamada a **popen**.

Creación de una tubería con nombre

La creación de una tubería con nombre se efectúa mediante la función de biblioteca **mkfifo**.

El prototipo **mkfifo** es el siguiente:

```
#include <sys/stat.h>

int mkfifo (const char *path, mode_t mode);
```

Sin embargo, no de trata de una llamada al sistema. En realidad, esta función se implementa de una manera muy simple, con la llamada **mknod**. La implementación de esta función se realiza con la llamada siguiente

```
mknod (path, mode | S_IFIFO, 0);
```

Hay que destacar que el usuario puede crear una tubería con nombre usando **mkfifo** o bien el mandato **mknod**.

Como todo archivo, las tuberías con nombre permiten las entradas/salidas con bloqueo (de modo predeterminado) o sin bloqueo (especificando la opción O_NODELAY o O_NONBLOCK al abrir con **open**). Todas las operaciones de manipulación de un archivo son válidas para una tubería con nombre, salvo las llamadas a **lseek**.

26.4 Estructuras de datos

El proceso utiliza las llamadas al sistema *read()* y *write()* para acceder al Pipe. Por lo tanto, para cada Pipe, el núcleo crea un objeto *i-nodo* más dos objetos de archivo, uno para lectura y otro para escritura. Cuando el objeto *i-nodo* referencia un Pipe, su campo *i_pipe* señala a una estructura *pipe_inode_info*, que es en la que se va a apoyar la llamada al sistema pipe.

El archivo donde se puede encontrar la estructura *pipe_inode_info* es <include/Linux/pipe_fs_i.h>.

```
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];
};
```

Tipo	**Campo**	**Descripción**
wait_queue_head_t *	wait	Cola wait del pipe.
unsigned int	nrbufs	Número de buffers que contienen datos para leer.
unsigned int	curbuf	Indice del primer buffer que contiene datos para leer.
struct page *	tmp_page	Puntero a un marco de página.
unsigned int	readers	Número de procesos que tienen acceso en lectura a la tubería
unsigned int	writers	Número de procesos que tienen acceso en escritura a la tubería
unsigned int	waiting_writers	Número de procesos de escritura dormidos en la cola de espera.
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 esctructura i-nodo
Struct pipe_buffer	Bufs [PIPE_BUFFERS]	Vector de buffers de pipe (16).

- Estructura *Pipe_inode_info* -

Constante PIPE_BUFFERS con un valor definido a 16, indicando que hay 16 buffer.

```
#define PIPE_BUFFERS (16)
```

El campo *bufs* de la estructura almacena un vector de 16 (PIPE_BUFFERS) objetos *pipe_buffer*, donde cada uno describe un buffer del pipe.

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

Los campos de estos objetos son los siguientes:

Tipo	**Campo**	**Descripción**
struct page *	page	Dirección del descriptor del marco de página dentro del buffer.
unsigned int	offset	Posición actual de los datos significativos dentro del marco de página.
unsigned int	len	Longitud de los datos significativos de buffer pipe.

struct pipe_buf_operations *	ops	Dirección de la tabla de operaciones.
unsigned int	flags	Flags, indican las opciones de funcionamiento.

26.5 Funciones para creación y destrucción del Pipe

La creación de una tubería anónima se efectúa en un primer momento creando un nuevo *i-nodo* en el sistema de archivos. Este *i-nodo* se inicializa con los diferentes campos útiles y dado que posee un número de enlaces nulos este archivo no es visible en el sistema de archivos, y la única manera de acceder a él, desde el punto de vista del programador, es poseer un descriptor sobre este archivo.

La llamada al sistema *pipe* permite crear un Pipe. Para efectuar esta operación, hay que pasar cómo parámetro un vector, que contiene los descriptores de fichero de lectura (filedes[0]) y de escritura (filedes[1]).

```
int pipe (int filedes [2]);.
```

-Tubería anónima – Pipe –

La llamada al sistema *pipe* es tratada por la función *sys_pipe*, la cual invoca la función *do_pipe* para crear un nuevo Pipe. La función *sys_pipe* puede encontrarse referenciada en la tabla *sys_call_table*, al igual que todas las llamadas al sistema. Tal tabla se encuentra en el archivo <arch/i386/kernel/syscall_table.s>.

```
.long sys_pipe
```

El archivo que contiene la función *sys_pipe* es <arch/i386/kernel/sys_i386c>.

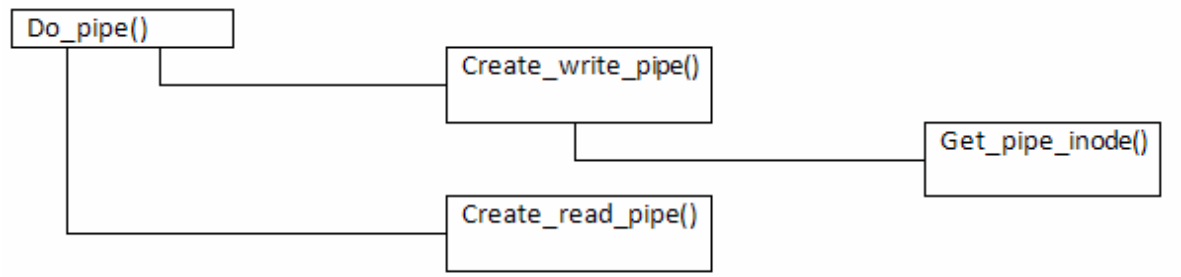
<linux/arch/arm/kernel/sys_arm.c>

```
43 asmlinkage int sys_pipe(nabi_no_regargs volatile struct pt_regs regs)
44{
45      int fd[2];
46      int error, res;
48      error = do_pipe(fd);
49      if (error) {
50              res = error;
51              goto out;
52      }
```

```
53        regs.regs[3] = fd[1];
54        res = fd[0];
55out:
56        return res;
57}
```

La función *do_pipe* es la encarga de crear el Pipe, se encuentra en el archivo <fs/pipe.c>. El objetivo de esta función, sin olvidar la creación del *i-nodo* y la estructura de pipe, es crear dos descriptores de archivo asociados a dos estructuras *file* que controlarán la escritura y lectura del Pipe que se encuentra en memoria. Esto lo hace buscando, en primer lugar, ranuras para los ficheros de entrada y salida, a continuación escribe en la tabla de *filp* los descriptores de ambos ficheros, dejándolos listos para ser utilizados.

En la creación del Pipe es necesaria la creación de un *i-nodo* referente al Pipe a crear y una estructura *pipe_inode_info*. Estas labores son realizadas por la función *get_pipe_inode()* explicada anteriormente. La inicialización y creación del *i-nodo,* es decir, el llamamiento a la función *get_pipe_inode()*, no es realizado directamente por la función *do_pipe*, sino que la función *create_write_pipe()* es quien realiza tal acción.



<fs/pipe.c>

```
1027int do_pipe(int *fd)
1028{
1029    struct file *fw, *fr;
1030    int error;
1031    int fdw, fdr;
        /*Se crea el pipe para procesos que quieren escribir*/
1033    fw = create_write_pipe();
1034    if (IS_ERR(fw))
1035            return PTR_ERR(fw);
        /* Se crea el pipe para procesos que quieren leer*/
```

```
1036    fr = create_read_pipe(fw);
1037    error = PTR_ERR(fr);
1038    if (IS_ERR(fr))
1039            goto err_write_pipe;
1041    error = get_unused_fd();
1042    if (error < 0)
1043            goto err_read_pipe;
1044    fdr = error;
1045
1046    error = get_unused_fd();
1047    if (error < 0)
1048            goto err_fdr;
1049    fdw = error;
1050
1051    error = audit_fd_pair(fdr, fdw);
1052    if (error < 0)
1053            goto err_fdw;
1054
/* Se almacenan los descriptores de fichero de los lectores y escritores en la estructura file de lectura y escritura respectivamente.*/
1055    fd_install(fdr, fr);
1056    fd_install(fdw, fw);
/* Se asignan los descriptores de ficheros de los procesos lectores y escritores, en el vector que se pasa por parámetro para obtener los descriptores de fichero del pipe.*/
1057    fd[0] = fdr;
1058    fd[1] = fdw;
1060    return 0;
1062 err_fdw:
1063    put_unused_fd(fdw);
1064 err_fdr:
1065    put_unused_fd(fdr);
1066 err_read_pipe:
1067    dput(fr->f_dentry);
1068    mntput(fr->f_vfsmnt);
1069    put_filp(fr);
1070 err_write_pipe:
1071    free_write_pipe(fw);
1072    return error;
1073}
```

<fs/pipe.c>

```
952struct file *create_write_pipe(void)
953{
954     int err;
955     struct inode *inode;
956     struct file *f;
957     struct dentry *dentry;
958     struct qstr name = { .name = "" };
960     err = -ENFILE;
961     inode = get_pipe_inode();
```

```
/* Se crea el inode con la estructura pipe, sobre este inode es con el que trabajará la llamada al sistema pipe.*/
962     if (!inode)
963             goto err;
965     err = -ENOMEM;
        /*En dentry se almacenan referencias a directorios usados. D_alloc se asigna una dentry a un nombre */
966     dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &name);
967     if (!dentry)
968             goto err_inode;
        /* Se asignan las operaciones del pipe*/
970     dentry->d_op = &pipefs_dentry_operations;
976     dentry->d_flags &= ~DCACHE_UNHASHED;
/* Se añade una dentry a la tabla de hash del inode, actualizando el campo d_inode de la dentry*/
977     d_instantiate(dentry, inode);
979     err = -ENFILE;
        /* Se asigna e inicializa la estructura file con los parámetros para la escritura*/
980     f = alloc_file(pipe_mnt, dentry, FMODE_WRITE, &write_pipe_fops);
981      if (!f)
982             goto err_dentry;
        /* Se asignan campos para la escritura*/
983     f->f_mapping = inode->i_mapping;
984
985     f->f_flags = O_WRONLY;
986     f->f_version = 0;
987
988     return f;
990 err_dentry:
991     dput(dentry);
 992 err_inode:
993     free_pipe_info(inode);
994      iput(inode);
995 err:
996     return ERR_PTR(err);
997}
```

En la creación de una estructura *file* para lectura, mediante la función *create_read_pipe,* se utiliza la estructura *file* creada por la subrutina create_write_pipe, puesto que tienen campos en común y los únicos que cambian son los que especifica que se trata de una lectura o una escritura.

<fs/pipe.c>

```
1007struct file *create_read_pipe(struct file *wrf)
1008{   /* wrf es la estructura creada mediante el create_write_pipe. Toma una fichero que no está siendo usado.*/
1009    struct file *f = get_empty_filp();
1010    if (!f)
1011            return ERR_PTR(-ENFILE);
```

```
1013    /* Dado se apoya en el parámetro pasado, se copian los campos comunes*/
1014    f->f_path.mnt = mntget(wrf->f_path.mnt);
1015    f->f_path.dentry = dget(wrf->f_path.dentry);
1016    f->f_mapping = wrf->f_path.dentry->d_inode->i_mapping;
1018    f->f_pos = 0;
                /* Indica en el campo f_flags que se trata de lectura. */
1019    f->f_flags = O_RDONLY;
        /* Almacena una estructura con las operaciones de lectura en pipe y el modo*/
1020    f->f_op = &read_pipe_fops;
1021    f->f_mode = FMODE_READ;
1022    f->f_version = 0;
1024    return f;
1025}
```

La utilización de una tubería por un proceso único tiene un interés limitado. La creación de una tubería anónima va seguida de la creación de un proceso que hereda descriptores sobre la tubería y puede por tanto comunicarse con el creador de la tubería. Es importante realizar estas operaciones en este orden porque si no la tubería sólo podría ser accedida por uno solo de los dos procesos.

Cada proceso posee entonces un descriptor en lectura y un descriptor en escritura en la misma tubería. A fin de comunicarse correctamente, los procesos deben elegir un sentido de transmisión. Cada uno utiliza entonces uno solo de los dos descriptores. Uno de los procesos escribe en la tubería, y el otro lee. Los descriptores inutilizados pueden cerrarse por la llamada al sistema *close*. Una vez cerrado, el descriptor no permite ya acceder al Pipe. Cada proceso debe cerrar un descriptor antes de usar el otro. Si un proceso necesita tanto escribir como leer usando Pipe, ha de utilizar dos Pipes diferentes, creándolos mediantes dos llamadas diferentes de *pipe.*

Siempre que un proceso invoca la llamada al sistema *close* sobre un descriptor asociado a un Pipe, el núcleo ejecuta la función *fput()* correspondiente al objeto del archivo, decrementando el contador en uso. Si el contador se hace 0, la función invoca al método de liberación *release.*

Dependiendo si el archivo es asociado con lectura o con escritura, el método *release* es implementado por *pipe_read_release()* o por *pipe_write_release()*. Ambas funciones invocan a *pipe_release()*, la cual pone los campos *reader* y *writers* del *pipe_inode_info* a 0, dependiendo de si se trata de el cierre de un descriptor del fichero que usa el pipe para lectura o la escritura. La función comprueba si estos últimos campos son iguales a 0, en este caso invoca el método de liberación de los buffers del pipe.

fs/pipe.c>

```
637static int pipe_release(struct inode *inode, int decr, int decw)
639{
640     struct pipe_inode_info *pipe;
641
642     mutex_lock(&inode->i_mutex);
643     pipe = inode->i_pipe;
644     pipe->readers -= decr;
645     pipe->writers -= decw;
646     /*Si ya no hay escritores ni lectores se librea el inode*/
647     if (!pipe->readers && !pipe->writers) {
648               free_pipe_info(inode);
649     } else {
          /*Sino, se fuerza a que terminen los escritores y lectores*/
650               wake_up_interruptible_sync(&pipe->wait);
651               kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
652               kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
653     }
654     mutex_unlock(&inode->i_mutex);
655
656 return 0;
657}
```

<fs/pipe.c>

```
717pipe_read_release(struct inode *inode, struct file *filp)
718{
719     pipe_read_fasync(-1, filp, 0);
720     return pipe_release(inode, 1, 0);
721}

723static int pipe_write_release(struct inode *inode, struct file *filp) 725{
726     pipe_write_fasync(-1, filp, 0);
727     return pipe_release(inode, 0, 1);
728}
```

26.6 Funciones de lectura y escritura en Pipe

El pipe puede verse reflejado como un el típico problema de concurrencia: lectores y escritores. Inicialmente el buffer está vacío y los procesos escritores corresponden a aquellos que realizan operaciones de escritura en el buffer. Lógicamente, los procesos lectores corresponden a aquellos que realizan operaciones de lectura en el buffer.

Las lecturas y las escrituras se efectúan en una memoria intermedia de tamaño PIPE_BUF. En función de las lecturas y escrituras de datos en la tubería, la

ventana que contiene los datos de la tubería se desplaza de manera circular en el interior de la memoria intermedia.

El tamaño de una tubería esta limitado a 4KB por razones de rendimiento. Este valor se encuentra en la constante PIPE_BUF definida en el archivo <include/Linux/limits.h>. Este límite corresponde al valor límite para la realización de una escritura atómica en la tubería. El usuario puede escribir más datos en la tubería, pero dejará de tener garantizada la atomización de la operación.

```
15 #define PIPE_BUF        4096    /* # bytes in atomic write to a pipe */
```

Tanto en read_pipe como en write_pipe, si una copia atómica falla, se intenta una vez más hacer la copia de forma no atómica.

Otra característica importante de estas dos funciones es que pueden ser llamadas con un flag `O_NONBLOCK` que indicará que la llamada a las funciones read y write no debe ser bloqueante. Esto implica que la llamada no dormirá al proceso en espera de que un lector o escritor opere con el buffer. Por defecto la llamada es bloqueante.

Todas las operaciones de entrada/salida, ya se trate de tuberías con nombre o anónimas, se encuentran en el archivo <fs/pipe.c>.

26.7 Escritura en Pipe: pipe_write()

Cuando se llama a la función write sobre el descriptor de fichero p[1], el núcleo invoca a la función pipe_write de acuerdo con el struct file_operations write_pipe_fops que se le habrá asignado al descriptor en la creación del mismo.

Pueden diferenciarse dos partes importantes:

1. En la cual se intentan escribir todos los datos posibles en el último buffer que se haya creado si no está lleno. Esto sólo puede hacerse si en la estructura `struct pipe_buf_operations` se ha activado el flag can_marge. Si es posible se escriben todos los datos que quepan en el buffer. Si con esto se lograron escribir todos los datos pasados a la función write, la función termina. Si no, pasa a la siguiente parte.

   Nota: el flag can_merge está activado por defecto puesto que a los buffers se les asigna siempre la estructura `anon_pipe_buf_ops` que lo tiene activado por defecto.

2. Es un bucle infinito en el cual se crean buffers para escribir los datos pasados a la función wirte. Por cada iteración se crea un buffer nuevo y se vuelcan los datos en él. Si aún quedan datos por escribir, se itera otra vez. Si no quedan buffers libres, y la función puede bloquearse (no se ha usado el flag O_NONBLOCK), se bloquea y será despertada cuando un lector libere espacio de un buffer con la función release.

El código de la función pipe_write es el siguiente:

```
 393static ssize_t
 394pipe_write(struct kiocb *iocb, const struct iovec *_iov,
 395           unsigned long nr_segs, loff_t ppos)
 396{
 397        struct file *filp = iocb->ki_filp;
 398        struct inode *inode = filp->f_path.dentry->d_inode;
 399        struct pipe_inode_info *pipe;
       //ret es el valor de retorno que puede ser el número de bytes
      leídos o un código de
       //error
 400        ssize_t ret;
       //Esta variable es un flag que indicará si se deben despertar a
los escritores
 401        int do_wakeup;
 402        struct iovec *iov = (struct iovec *)_iov;
 403        size_t total_len;
 404        ssize_t chars;
 405
       //Se calcula el total de bytes que se quieren escribir
       //total_len será un contador de los bytes que faltan por
escribir
 406        total_len = iov_length(iov, nr_segs);
 407        /* Null write succeeds. */
 408        if (unlikely(total_len == 0))
 409                return 0;
 410
 411        do_wakeup = 0;
 412        ret = 0;
       //Coge el semáforo
 413        mutex_lock(&inode->i_mutex);
 414        pipe = inode->i_pipe;
 415
       //Si no hay lectores, enviar señal SIGPIPE y retornar
 416        if (!pipe->readers) {
 417                send_sig(SIGPIPE, current, 0);
 418                ret = -EPIPE;
 419                goto out;
 420        }
 421
 422        /* We try to merge small writes */
       //Si hay espacio libre en el último buffer, se escribe en él
       //¡En este caso no se crea ningún buffer nuevo!
 423        chars = total_len & (PAGE_SIZE-1); /* size of the last
buffer */
 424        if (pipe->nrbufs && chars != 0) {
           //Se obtiene el buffer
 425                int lastbuf = (pipe->curbuf + pipe->nrbufs - 1) &
```

```
 426
(PIPE_BUFFERS-1);
 427                 struct pipe_buffer *buf = pipe->bufs + lastbuf;
 428                 const struct pipe_buf_operations *ops = buf->ops;
                //Se obtiene el offset del buffer
 429                 int offset = buf->offset + buf->len;
 430
                //Si se pueden combinar los buffers, se escribe
 431                 if (ops->can_merge && offset + chars <= PAGE_SIZE)
{
 432                         int error, atomic = 1;
 433                         void *addr;
 434
 435                         error = ops->confirm(pipe, buf);
 436                         if (error)
 437                                 goto out;
 438
 439                         iov_fault_in_pages_read(iov, chars);
 440redo1:
                //Mapea el marco de página del buffer
 441                         addr = ops->map(pipe, buf, atomic);
                //Copia del espacio de usuario los bytes
                        //iov: buffer del usuario
                        //offset + addr: comienzo de los bytes del
buffer donde se escribirá
                        //chars: cantidad de bytes a escribir
                        //atomic: operación atómica o no

 442                         error = pipe_iov_copy_from_user(offset +
addr, iov,
 443                                                         chars,
atomic);
                //Desmapea el marco de página del buffer
 444                         ops->unmap(pipe, buf, addr);
 445                         ret = error;
 446                         do_wakeup = 1;
 447                         if (error) {
 448                                 if (atomic) {
 449                                         atomic = 0;
 450                                         goto redo1;
 451                                 }
 452                                 goto out;
 453                         }
                //Actualiza la longitud del buffer
 454                         buf->len += chars;
                //Actualiza los bytes que faltan por escribir
 455                         total_len -= chars;
 456                         ret = chars;
                //Si se han escrito todos los bytes, retornar
 457                         if (!total_len)
 458                                 goto out;
 459                 }
 460         }
 461
        //Bucle infinito: en este bucle se crearán nuevos buffers
siempre
 462         for (;;) {
 463                 int bufs;
 464
        //Si no hay lectores, enviar la señal SIGPIPE y terminar
 465                 if (!pipe->readers) {
```

```
 466                        send_sig(SIGPIPE, current, 0);
 467                        if (!ret)
 468                                ret = -EPIPE;
 469                        break;
 470                }
        //Si hay algún buffer libre
 471                bufs = pipe->nrbufs;
 472                if (bufs < PIPE_BUFFERS) {
 473                        int newbuf = (pipe->curbuf + bufs) &
(PIPE_BUFFERS-1);
 474                        struct pipe_buffer *buf = pipe->bufs +
newbuf;
 475                        struct page *page = pipe->tmp_page;
 476                        char *src;
 477                        int error, atomic = 1;
 478
                //Si el buffer no tiene página, obtenerla
 479                        if (!page) {
 480                                page = alloc_page(GFP_HIGHUSER);
 481                                if (unlikely(!page)) {
 482                                        ret = ret ? : -ENOMEM;
 483                                        break;
 484                                }
 485                                pipe->tmp_page = page;
 486                        }
 487                        /* Always wake up, even if the copy fails.
Otherwise
 488                         * we lock up (O_NONBLOCK-)readers that
sleep due to
 489                         * syscall merging.
 490                         * FIXME! Is this really true?
 491                         */
                //Despertar a los lectores
 492                        do_wakeup = 1;
                //chars = tamaño de página
 493                        chars = PAGE_SIZE;
                //Si los bytes a escribir son menores que el tamaño
máximo de la página
                //Se ajusta la cantidad a escribir a ese tamaño
 494                        if (chars > total_len)
 495                                chars = total_len;
 496
 497                        iov_fault_in_pages_read(iov, chars);
 498redo2:
                //Mapea el marco de página del buffer
 499                        if (atomic)
 500                                src = kmap_atomic(page, KM_USER0);
 501                        else
 502                                src = kmap(page);
 503
                //Copia del espacio de usuario los bytes
                        //iov: buffer del usuario
                        //offset + addr: comienzo de los bytes del
buffer donde se escribirá
                        //chars: cantidad de bytes a escribir
                        //atomic: operación atómica o no
 504                        error = pipe_iov_copy_from_user(src, iov,
chars,
 505                                                        atomic);
                //Desmapea el marco de página del buffer
 506                        if (atomic)
```

```
 507                                    kunmap_atomic(src, KM_USER0);
 508                            else
 509                                    kunmap(page);
 510
 511                            if (unlikely(error)) {
 512                                    if (atomic) {
 513                                            atomic = 0;
 514                                            goto redo2;
 515                                    }
 516                                    if (!ret)
 517                                            ret = error;
 518                                    break;
 519                            }
                        //Añade los bytes al número de bytes leídos
 520                            ret += chars;
 521
 522                            /* Insert it into the buffer array */
                        //Inserta la página en el vector de buffers
 523                            buf->page = page;
                        //Se le asignan las operaciones al buffer
 524                            buf->ops = &anon_pipe_buf_ops;
 525                            buf->offset = 0;
 526                            buf->len = chars;
 527                            pipe->nrbufs = ++bufs;
 528                            pipe->tmp_page = NULL;
 529
                        //Se actualiza el contador de bytes escritos
 530                            total_len -= chars;
                        //Si se han escrito todos los bytes, retornar
 531                            if (!total_len)
 532                                    break;
 533                    }
                //Si quedan buffers libres, continuar escribiendo
 534                    if (bufs < PIPE_BUFFERS)
 535                            continue;
 536                    if (filp->f_flags & O_NONBLOCK) {
 537                            if (!ret)
 538                                    ret = -EAGAIN;
 539                            break;
 540                    }
 541                    if (signal_pending(current)) {
 542                            if (!ret)
 543                                    ret = -ERESTARTSYS;
 544                            break;
 545                    }
                //Si es necesario, despertar a los lectores
 546                    if (do_wakeup) {
 547                            wake_up_interruptible_sync(&pipe->wait);
 548                            kill_fasync(&pipe->fasync_readers, SIGIO,
POLL_IN);
 549                            do_wakeup = 0;
 550                    }
                //Se duerme este escritor en espera de espacio en el buffer
 551                    pipe->waiting_writers++;
 552                    pipe_wait(pipe);
 553                    pipe->waiting_writers--;
 554            }
 555out:
                //Libera el semáforo
 556            mutex_unlock(&inode->i_mutex);
                //Despierta a los lectores si es necesario
```

```
 557         if (do_wakeup) {
 558                 wake_up_interruptible_sync(&pipe->wait);
 559                 kill_fasync(&pipe->fasync_readers, SIGIO,
POLL_IN);
 560         }
 561         if (ret > 0)
 562                 file_update_time(filp);
 563         return ret;
 564}
```

26.8 Escritura en Pipe: pipe_read()

Cuando se llama a la función read sobre el descriptor de fichero p[0], el núcleo invoca a la función pipe_read de acuerdo con el struct file_operations read_pipe_fops que se le habrá asignado al descriptor en la creación del mismo.

La operación implementa un bucle en el que en cada iteración se leen datos de un buffer en el que previamente se hayan escrito datos. La cantidad de datos que se leen será la del número de bytes que falten por leer o, si esta cantidad es mayor que los datos que hay en el buffer, leerá todo lo que pueda del buffer y si no se ha activado el flag `O_NONBLOCK` se bloquea en espera de se escriban nuevos datos con los que satisfacer su demanda. Si al leer de un buffer lo vaciamos, se libera con la operación release y se despiertan a los escritores para que puedan introducir nuevos datos.

El código de la función pipe_read es el siguiente:

```
 280static ssize_t
 281pipe_read(struct kiocb *iocb, const struct iovec *_iov,
 282           unsigned long nr_segs, loff_t pos)
 283{
 284         struct file *filp = iocb->ki_filp;
 285         struct inode *inode = filp->f_path.dentry->d_inode;
 286         struct pipe_inode_info *pipe;
        //Esta variable es un flag que indicará si se deben despertar a
los escritores
 287         int do_wakeup;
        //ret es el valor de retorno que puede ser el número de bytes
      leídos o un código de
        //error
 288         ssize_t ret;
 289         struct iovec *iov = (struct iovec *)_iov;
 290         size_t total_len;
 291
        //Se calcula el total de bytes que se quieren leer
        //total_len será un contador de los bytes que faltan por leer
 292         total_len = iov_length(iov, nr_segs);
 293         /* Null read succeeds. */
```

```
 294        if (unlikely(total_len == 0))
 295                return 0;
 296
 297        do_wakeup = 0;
 298        ret = 0;
       //Coge el semáforo
 299        mutex_lock(&inode->i_mutex);
 300        pipe = inode->i_pipe;
       //Bucle infinito
 301        for (;;) {
              //nrbufs contiene el número de buffers con datos para
leer
              //Si hay bytes para leer
 302                int bufs = pipe->nrbufs;
 303                if (bufs)
                   {
              //curbuf contiene el primer buffer con datos para leer
 304                        int curbuf = pipe->curbuf;
              //buf es el vector de buffers, se le suma curbuf y
obtenemos
              //el buffer de lectura que se guarda en buf
 305                        struct pipe_buffer *buf = pipe->bufs +
curbuf;
              //Se obtienen las operaciones del buffer
 306                        const struct pipe_buf_operations *ops =
buf->ops;
 307                        void *addr;
              //Se obtiene la cantidad de datos del buffer
 308                        size_t chars = buf->len;
 309                        int error, atomic;
 310
              //Si hay más datos para leer de los que ha solicitado
la operación
              //de lectura, se leerá sólo la cantidad
requerida(total_len)
              //Si no, se leerá el máximo posible, chars bytes
 311                        if (chars > total_len)
 312                                chars = total_len;
 313
 314                        error = ops->confirm(pipe, buf);
 315                        if (error) {
 316                                if (!ret)
 317                                        error = ret;
 318                                break;
 319                        }
 320
 321                        atomic = !iov_fault_in_pages_write(iov,
chars);
 322redo:
              //Mapea el marco de página del buffer
 323                addr = ops->map(pipe, buf, atomic);
              //Copia al espacio de usuario los bytes
                       //iov: buffer del usuario
                       //addr + buf->offset: comienzo de los bytes de
lectura
                       //chars: cantidad de bytes a leer
                       //atomic: operación atómica o no
 324                   error = pipe_iov_copy_to_user(iov, addr + buf-
>offset, chars, atomic);
              //Desmapea el marco de página del buffer
```

```
 325                   ops->unmap(pipe, buf, addr);
 326                         if (unlikely(error)) {
 327                                 /*
 328                                  * Just retry with the slow path
if we failed.
 329                                  */
 330                                 if (atomic) {
 331                                         atomic = 0;
 332                                         goto redo;
 333                                 }
 334                                 if (!ret)
 335                                         ret = error;
 336                                 break;
 337                         }
                 //Actualiza la longitud, el offset del buffer y el nº
de caracteres leidos
 338                         ret += chars;
 339                         buf->offset += chars;
 340                         buf->len -= chars;
                 //Si el buffer está vacío, liberarlo y activar el flag
de avisar escritores
 341                         if (!buf->len) {
 342                                 buf->ops = NULL;
 343                                 ops->release(pipe, buf);
 344                                 curbuf = (curbuf + 1) &
(PIPE_BUFFERS-1);
 345                                 pipe->curbuf = curbuf;
 346                                 pipe->nrbufs = --bufs;
 347                                 do_wakeup = 1;
 348                         }
                 //Actualiza el número de caracteres que faltarían por
leer
 349                         total_len -= chars;
                 //Si no falta nada por leer, lectura finalizada,
terminar el bucle
 350                         if (!total_len)
 351                                 break;  /* common path: read
succeeded */
 352                 }
                 //Si queda por copiar y se puede seguir copiando,
iterar de nuevo
 353                 if (bufs)       /* More to do? */
 354                         continue;
                 //Si no hay escritores, terminar
 355                 if (!pipe->writers)
 356                         break;
 357                 if (!pipe->waiting_writers) {
 358                         /* syscall merging: Usually we must not
sleep
 359                          * if O_NONBLOCK is set, or if we got some
data.
 360                          * But if a writer sleeps in kernel space,
then
 361                          * we can wait for that data without
violating POSIX.
 362                          */
 363                         if (ret)
 364                                 break;
 365                         if (filp->f_flags & O_NONBLOCK) {
 366                                 ret = -EAGAIN;
 367                                 break;
```

```
 368                     }
 369                }
 370                if (signal_pending(current)) {
 371                     if (!ret)
 372                          ret = -ERESTARTSYS;
 373                     break;
 374                }

               //Si se activo el flag, despertar a los escritores
 375                if (do_wakeup) {
 376                     wake_up_interruptible_sync(&pipe->wait);
 377                     kill_fasync(&pipe->fasync_writers, SIGIO,
POLL_OUT);
 378                }
               //Duerme al lector
 379                pipe_wait(pipe);
 380           }
          //Libera el semáforo
 381           mutex_unlock(&inode->i_mutex);
 382
 383           /* Signal writers asynchronously that there is more room.
*/
          //Despierta a los escritores dormidos
 384           if (do_wakeup) {
 385                wake_up_interruptible_sync(&pipe->wait);
 386                kill_fasync(&pipe->fasync_writers, SIGIO,
POLL_OUT);
 387           }
 388           if (ret > 0)
 389                file_accessed(filp);
 390           return ret;
 391}
```

26.9 Funciones auxiliares a read_pipe y write_pipe

Existen dos funciones de copia de datos que se limitan a realizar el verdadero intercambio de datos entre el buffer del usuario y el buffer del pipe. Se trata de las operaciones `pipe_iov_copy_from_user` y `pipe_iov_copy_to_user`. pipe_iov_copy_from_user copia datos de un buffer del usuario hacia el buffer del pipe. pip_iov_copy_to_user copia datos desde el buffer del pipe hacia el buffer del usuario.

```
  57static int
  58pipe_iov_copy_from_user(void *to, struct iovec *iov, unsigned long
len,
  59                        int atomic)
  60{
  61        unsigned long copy;
  62
  63        while (len > 0) {
  64                while (!iov->iov_len)
```

```
  65                        iov++;
  66                copy = min_t(unsigned long, len, iov->iov_len);
  67
  68                if (atomic) {
  69                        if (__copy_from_user_inatomic(to, iov-
>iov_base, copy))
  70                                return -EFAULT;
  71                } else {
  72                        if (copy_from_user(to, iov->iov_base,
copy))
  73                                return -EFAULT;
  74                }
  75                to += copy;
  76                len -= copy;
  77                iov->iov_base += copy;
  78                iov->iov_len -= copy;
  79        }
  80        return 0;
  81}
```

```
  83static int
  84pipe_iov_copy_to_user(struct iovec *iov, const void *from,
unsigned long len,
  85                      int atomic)
  86{
  87        unsigned long copy;
  88
  89        while (len > 0) {
  90                while (!iov->iov_len)
  91                        iov++;
  92                copy = min_t(unsigned long, len, iov->iov_len);
  93
  94                if (atomic) {
  95                        if (__copy_to_user_inatomic(iov->iov_base,
from, copy))
  96                                return -EFAULT;
  97                } else {
  98                        if (copy_to_user(iov->iov_base, from,
copy))
  99                                return -EFAULT;
 100                }
 101                from += copy;
 102                len -= copy;
 103                iov->iov_base += copy;
 104                iov->iov_len -= copy;
 105        }
 106        return 0;
 107}
```

Además ambas funciones, read_pipe y write_pipe, hacen uso de una función auxiliar para dormir al proceso actual y llamar al planificador consiguiendo de esta forma dormir a escritores y a lectores. Se trata de la función `pipe_wait`.

```
void pipe_wait(struct pipe_inode_info *pipe)
  41{
  42        DEFINE_WAIT(wait);
  43
  44        /*
  45         * Pipes are system-local resources, so sleeping on them
  46         * is considered a noninteractive wait:
  47         */
       //Añade el proceso a la cola de espera del pipe
  48        prepare_to_wait(&pipe->wait, &wait, TASK_INTERRUPTIBLE);
       //Libera el semáforo
  49        if (pipe->inode)
  50                mutex_unlock(&pipe->inode->i_mutex);
       //Invoca al planificador
  51        schedule();
       //Retira el proceso de la cola de espera del pipe
  52        finish_wait(&pipe->wait, &wait);
       //Adquiere el semáforo
  53        if (pipe->inode)
  54                mutex_lock(&pipe->inode->i_mutex);
  55}
```
