

Comunicación interproceso

| | |
|------------------------------------|-----|
| Conductos | 352 |
| FIFO | 361 |
| Introducción del IPC del sistema V | 368 |
| Memoria compartida | 371 |
| Colas de mensaje | 377 |
| Semáforos | 386 |
| Resumen | 392 |

CAPÍTULO



Este capítulo cubre el conjunto de métodos de Linux que sirven para la *comunicación interproceso*, o IPC. IPC es un término genérico que describe los métodos que utilizan los procesos para comunicarse los unos con los otros. Sin IPC, los procesos pueden intercambiar datos u otra información sólo a través del sistema de archivos o, en el caso de los procesos que tienen un ancestro en común (como la relación padre-hijo después de un fork), a través de descriptores de archivo heredados. En particular, conoceremos los conductos, FIFO, memoria compartida, semáforos y colas de mensajes.

Conductos

En este capítulo explicamos dos tipos de conductos: sin nombre y con nombre. Los conductos con nombre normalmente se llaman FIFO (Primero en entrar, primero en salir).

Los *conductos sin nombre* son sin nombre porque nunca tienen un nombre de ruta y, por tanto, nunca existen en el sistema de archivos. Estrictamente hablando, son dos descriptores de archivos asociados con un ínodo del núcleo. El último proceso en cerrar uno de estos descriptores hace que el ínodo y, por tanto, el conducto, desaparezca. Por otra parte, los *conductos con nombre* están en el sistema de archivos. Se llaman FIFO porque los datos se leen en ellos en el orden en el que se escribieron, así que el primer dato en entrar en el FIFO es también el primero en salir. La Figura 17.1 ilustra las diferencias y semejanzas entre los conductos sin nombre y los FIFO.

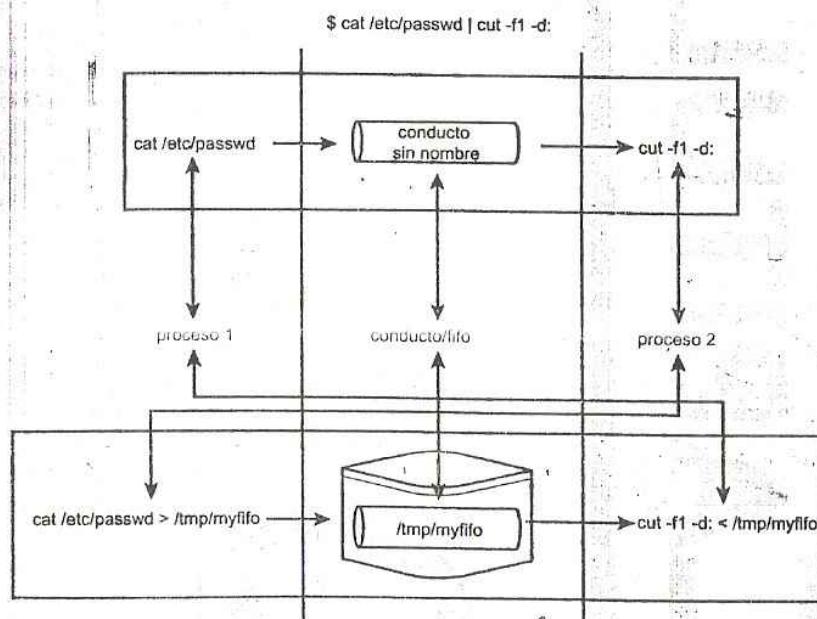


FIGURA 17.1. Los conductos y FIFO funcionan de forma similar pero tienen diferente semántica.

La parte superior de la Figura 17.1 ilustra la ejecución de la línea de conductos de shell `cat /etc/passwd cut -f1-d:`. La parte inferior muestra cómo se ejecutaría la línea

de conducto de la parte superior si utilizáramos un conducto con nombre en lugar de uno sin nombre. Las líneas verticales representan el punto en el que se leen o escriben en el conducto. La flechas dobles muestran la correspondencia entre la entrada y la salida de los dos tipos de conductos. Todos los detalles los explicaremos en secciones posteriores, así que es posible que prefiera volver a esta figura según vaya hacia delante en el capítulo. En la parte superior de la figura, la salida del comando `cat` se transmite a través de un conducto sin nombre creado en el núcleo. Su salida se convierte en la entrada de `cut`. Es una utilización típica de la línea de conductos *shell*.

En particular, observemos que la utilización de conductos con nombre requiere que se invierta el orden de los comandos `cat` y `cut`. Por razones que explicaremos más tarde, el comando `cut` debe ejecutarse primero (y en segundo plano) de forma que seamos capaces, en el mismo terminal o desde la misma consola, de llevar a cabo el comando `cat` que proporciona la entrada para el conducto con nombre.

El conducto con nombre, `/tmp/myfifo`, y el conducto sin nombre sirven para el mismo propósito: alimentar su entrada con su salida. Lo único que realmente difiere es el orden en el que se ejecutan los comandos y la utilización de los operadores de redirección *shell*, `>` y `<`, cuando tratamos con conductos con nombre.

La mayoría de los usuarios de Linux están familiarizados con los conductos sin nombre, aunque no lo sepan. Todo comando como el siguiente:

```
$ cut -f1 -d: </etc/group | sort
```

utiliza conductos sin nombre. En este ejemplo, la salida del comando `cut` (cuya entrada es redirigida desde `/etc/group`) se convierte en la entrada del comando `sort`. Como ya sabemos, `|` es el símbolo del conducto. Lo que no sabemos es que nuestro *shell* probablemente implementa `|` utilizando la función `pipe` con la que nos encontraremos en un momento.

NOTA

Un rápido consejo sobre la terminología: a no ser que se tenga que aclarar la distinción, cuando hablamos en este capítulo de un conducto nos referimos a los conductos sin nombre, y con FIFO nos referimos a los conductos con nombre. Sin embargo, ambos son *half duplex*: es decir, los datos fluyen en una sola dirección, al igual que el agua en un conducto de drenaje. Los conductos FIFO no se pueden buscar; es decir, no podemos utilizar llamadas como `lseek` para colocar el puntero del archivo.

Los conductos sin nombre tienen dos inconvenientes. Primero, son *half duplex*, así que los datos sólo pueden viajar en una dirección. Segundo, y más importante, sólo se pueden utilizar entre procesos relacionados, aquéllos que tienen un ancestro en común. Como recordaremos del Capítulo 13, "Control de procesos", los procesos hijo creados con un `fork` o un `exec` heredan los descriptores de su padre.

Cómo abrir y cerrar conductos

Naturalmente, antes de que podamos leer o escribir en un conducto, éste debe existir. El prototipo de la llamada para crear un conducto es:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Si pipe tiene éxito, abre dos descriptores de archivos y almacena sus valores en la matriz de enteros `filedes`. El primer descriptor, `filedes[0]`, se utiliza para leer, así que pipe lo abre utilizando el indicador `O_RDONLY` de `read`. El segundo, `filedes[1]`, se utiliza para escribir, así que pipe lo abre utilizando el indicador `O_WRONLY` de `open`. pipe devuelve 0 si tiene éxito y -1 si se produce un error, en cuyo caso también establece la variable global de error `errno`.

Possibles condiciones de error son `EMFILE`, que significa que el proceso que llama ya tiene abiertos demasiados descriptores de archivos, `EFAULT`, que significa que la matriz `filedes` no era válida, o `ENFILE`, que se produce cuando la tabla de archivos del núcleo está completa (definitivamente, *no* es una buena señal). De nuevo, debemos enfatizar que los descriptores de archivos no se corresponden con archivos de disco, sólo con un `inode` que reside en el núcleo. La Figura 17.2 ilustra esta afirmación.

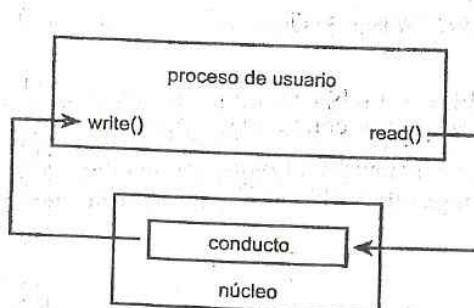


FIGURA 17.2. Un conducto de Linux existe sólo en el núcleo.

Para cerrar el conducto, cerramos sus descriptores de archivos asociados utilizando la llamada de sistema `close`.

El Listado 17.1 muestra cómo se abre y se cierra un conducto. Para construir este programa utilizando el archivo make proporcionado, ejecutamos el comando `make opipe`.

LISTADO 17.1. OPIPE.C

```
/*
 * opipe.c - Apertura y cierre de un conducto
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

(continúa)

LISTADO 17.1. OPIPE.C (CONTINUACIÓN)

```
int main(void)
{
    int fd[2]; /* Matriz de descriptores de archivos */

    if((pipe(fd)) < 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    printf("los descriptores son %d, %d\n", fd[0], fd[1]);
    close(fd[0]);
    close(fd[1]);
    exit(EXIT_SUCCESS);
}
```

La salida de este programa (véase más abajo) muestra que la llamada pipe ha tenido éxito (los valores de los descriptores puede ser diferentes según el sistema). El programa llama a la función pipe, pasándole la matriz de descriptores de archivos, fd. Si la llamada pipe tiene éxito, el programa imprime los valores enteros de los descriptores, los cierra y sale.

Una salida de este programa debería parecerse a ésta:

```
$ ./opipe
los descriptores son 3, 4
$
```

Cómo leer y escribir conductos

Para leer y escribir conductos, utilizamos sencillamente las llamadas `read` y `write`. Recordemos que `read` lee de `filedes[0]` y `write` escribe en `filedes[1]`.

Dicho esto, hay una pequeña razón para la apertura por parte de un proceso de un conducto para su propio uso. Los conductos se utilizan para intercambiar datos. Como un proceso ya ha accedido a los datos que compartiría a través de un conducto, no tiene sentido compartir los datos consigo mismo. Normalmente, un proceso llama a `pipe` y después a `fork` para generar un proceso hijo. Como el hijo hereda de su padre cualquier descriptor de archivos abierto, se ha establecido un canal IPC. El siguiente paso depende de qué proceso sea el de lectura y cuál el de escritura. La norma general es que el proceso de lectura cierra el lado de escritura del conducto y que el proceso de escritura cierra el lado de lectura. Los siguientes puntos hacen que el proceso sea más evidente:

- Si el proceso padre está enviando datos al hijo, el padre cierra `filedes[0]` y escribe en `filedes[1]`, mientras que el hijo cierra `filedes[1]` y lee de `filedes[0]`.
- Si el hijo está enviando datos al padre, el hijo cierra `filedes[0]` y escribe en `filedes[1]`, mientras que el padre cierra `filedes[1]` y lee en `filedes[0]`.

La Figura 17.3 debería ayudarnos a visualizar el procedimiento adecuado y a recordar la norma.

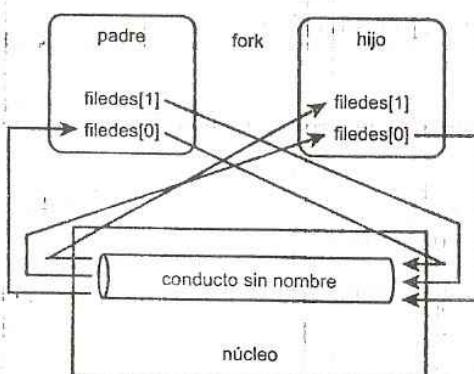


FIGURA 17.3. Lectura y escritura en un conducto después de un fork.

AVISO

Tentar leer y escribir en ambos lados de un solo conducto es un error de programación serio. Si dos procesos necesitan la funcionalidad de un conducto simultáneamente dúplex, el padre debe abrir dos conductos antes de hacer fork.

La parte superior de la Figura 17.3 muestra la disposición de los descriptores de archivos inmediatamente después del fork: tanto el padre como el hijo tienen descriptores abiertos. La Figura 17.3 asume que el padre será el escritor y que el hijo será el lector. La parte inferior de la figura ilustra el estado de los descriptores después de que el padre cierre su descriptor de lectura y el hijo su descriptor de escritura.

NOTA

Después de cerrarse el lado de escritura de un conducto, leer en él devolverá 0 para indicar el final del archivo. Sin embargo, si se ha cerrado el lado de lectura, cualquier intento de escribir creará una señal SIGPIPE para el escritor, y la misma llamada de escritura devolverá -1 y establecerá la variable global errno como EPIPE. Si el escritor no capta o ignora SIGPIPE, el proceso de escritura terminará.

El Listado 17.2, piperw.c, muestra el procedimiento correcto para abrir un conducto entre procesos relacionados. Construimos este programa ejecutando make piperw, definido en el archivo make proporcionado.

LISTADO 17.2. PIPERW.C

```
/*
 * piperw.c - El procedimiento correcto para abrir un conducto y hacer fork
 * a un proceso hijo.
```

(continúa)

LISTADO 17.2. PIPERW.C (CONTINUACIÓN)

```
/*
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(int argc, char *argv[])
{
    int fd[2]; /* Matriz de descriptores de archivos del conducto */
    int fdin; /* Descriptor para el archivo de entrada */
    char buf[BUFSZ];
    int pid, len;

    /* Crea el conducto */
    if((pipe(fd)) < 0)
        err_quit("pipe");

    /* Hace fork y cierra los descriptores apropiados */
    if((pid = fork()) < 0)
        err_quit("fork");
    if (pid == 0) {
        /* El hijo es el lector, cerramos el descriptor de escritura */
        close(fd[1]);
        while((len = read(fd[0], buf, BUFSZ)) > 0)
            write(STDOUT_FILENO, buf, len);
        close(fd[0]);
    } else {
        /* El padre es el escritor, cerramos el descriptor de lectura */
        close(fd[0]);
        if((fdin = open(argv[1], O_RDONLY)) < 0)
            perror("open");
        /* Enviamos algo ya que no podemos abrir la entrada */
        write(fd[1], "123\n", 4);
    } else {
        while((len = read(fdin, buf, BUFSZ)) > 0)
            write(fd[1], buf, len);
        close(fdin);
    }
    /* Cerramos el descriptor de escritura */
    close(fd[1]);
}
/* Recogemos el estado de salida */
waitpid(pid, NULL, 0);
exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
```

(continúa)

LISTADO 17.2. PIPERW.C (CONTINUACIÓN)

```

    perror(msg);
    exit(EXIT_FAILURE);
}

```

piperw espera el nombre de un archivo de entrada o envía 123\n al lector. Un ejemplo de ejecución utilizando opipe.c como entrada daría como resultado la salida:

```

$ ./piperw opipe.c
/*
 * opipe.c - Apertura y cierre de un conducto
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int fd[2]; /* Matriz de descriptores de archivos */
    if((pipe(fd)) < 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    printf("los descriptores son %d, %d\n", fd[0], fd[1]);
    close(fd[0]);
    close(fd[1]);
    exit(EXIT_SUCCESS);
}
$ 

```

Como podemos ver en la salida, piperw se comporta de forma muy parecida al comando cat, excepto en que utiliza un conducto en lugar de mostrar simplemente la entrada en stdout. Después del fork, el hijo cierra el descriptor de escritura que hereda, ya que sólo va a leer en el conducto. De forma similar, el padre es el escritor, así que cierra su descriptor de lectura.

En lugar de terminar el programa, si el padre es incapaz de abrir su archivo de entrada (`argv[1]`), envía la cadena 123\n a través del conducto antes de cerrar. Una vez que el padre termina de conducir los datos a su hijo, cierra su descriptor de escritura. Cuando un hijo lee 0 bits en el conducto, cierra el descriptor de lectura. Para terminar, aunque no está claro cuál de los dos sale primero, el padre llama a `waitpid` para recoger el estado de salida del hijo y evitar la creación de un *zombie* o de un huérfano.

NOTA

Si hay múltiples procesos escribiendo en un solo conducto, cada escritura de los procesos debe ser menor de PIPE_BUF bytes, una macro definida en `<sys/types.h>`.

(continúa)

(Continuación)

para asegurar escrituras atómicas; es decir, que los datos escritos por un proceso no se mezclan con los datos escritos por otro. Para describir este tema como una norma, para asegurar las escrituras atómicas, limitarímos la cantidad de datos escritos por una llamada de escritura a menos de PIPE_BUF bits.

Un método más sencillo

El programa piperw tiene que hacer gran cantidad de trabajo simplemente para hacer cat a un archivo: crear un conducto, hacer fork, cerrar los descriptores que no se necesitan en el padre y el hijo, abrir un archivo de entrada, escribir y leer en el conducto, cerrar los archivos y descriptores abiertos y recoger el estado de salida del hijo. Esta secuencia de acciones es tan común que ANSI/ISO C las ha codificado en las funciones de biblioteca estándar `popen` y `pclose`, cuyo prototipo es:

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

`popen` crea un conducto y hace fork con un proceso hijo seguido de un exec, que invoca a `/bin/sh -c` para ejecutar la cadena de comandos contenida en `command`. El argumento `mode` debe ser o r o w, que hacen la lectura y escritura en la biblioteca E/S. Es decir, si `mode` es r, el puntero de flujo FILE que devuelve `popen` está abierto para lectura, lo que significa que el flujo está adjunto a la salida estándar de `command`; leer del flujo es lo mismo que leer de la salida estándar de `command`. Así mismo, si `mode` es w, el flujo está adjunto a la entrada estándar de `command`, así que escribir en el flujo es lo mismo que escribir en la entrada estándar de `command`. Si la llamada `popen` falla, devuelve NULL. La condición de error que haya producido el fallo se establecerá en la variable de error `errno`.

Para cerrar el flujo, utilizamos `pclose`, que cierra el flujo E/S, espera que termine `command` y devuelve su estado de salida al proceso que llama. Si la llamada `pclose` falla devuelve -1.

El Listado 17.3, `popen.c` (construido utilizando el comando make `popen`), vuelve a escribir piperw para utilizar `popen` y `pclose`.

LISTADO 17.3. POPEN.C

```
/*
 * popen.c - Utilización de popen() para abrir un conducto
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
```

(continúa)

LISTADO 17.3. POPEN.C (CONTINUACIÓN)

```
#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(void)
{
    FILE *fp; /* flujo FILE para popen */
    char *cmdstring = "cat popen.c";
    char buf[BUFSZ]; /* Buffer for "input" */

    /* Crea el conducto */
    if((fp = popen(cmdstring, "r")) == NULL)
        err_quit("popen");

    /* Lee la salida de cmdstring */
    while(fgets(buf, BUFSZ, fp)) != NULL)
        printf("%s", buf);

    /* Cierra y recoge el estado de salida */
    pclose(fp);
    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Como podemos ver en el listado, `popen` y `pclose` hacen que trabajar con conductos sea mucho menos intensivo en código. El precio es perder cierta cantidad de control. Por ejemplo, estamos forzados a utilizar la biblioteca de flujo de C en lugar de las llamadas E/S de bajo nivel `read` y `write`. Además, `popen` obliga a nuestro programa a realizar un `exec` en segundo plano, lo que puede que no necesitemos o no queramos hacer. Para terminar, puede que la llamada a la función `pclose` utilizada para recoger el estado de salida del hijo no cuadre con los requerimientos de nuestro programa. No obstante esta pérdida de flexibilidad, `popen` ahorra de 10 a 15 líneas de código, mientras que el código para la lectura y la escritura se simplifica significativamente en comparación con el código de `piperw`. La salida, que mostramos en el siguiente ejemplo, no cambia. La semántica del argumento de modo no es intuitiva, así que tenemos que recordar que `r` significa que leemos de `stdout` y `w` que escribimos en `stdin`.

```
$ ./popen
/*
 * popen.v - Utilización de popen() para abrir un conducto.
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
```

```
#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(void)
{
    FILE *fp; /* Flujo FILE para popen */
    char *cmdstring = "cat popen.c";
    char buf[BUFSZ]; /* Buffer for "input" */

    /* Crea el conducto */
    if((fp = popen(cmdstring, "r")) == NULL)
        err_quit("popen");

    /* Lee la salida de cmdstring*/
    while((fgets(buf, BUFSZ, fp)) != NULL)
        printf("%s", buf);

    /* Cierra y recoge el estado de salida */
    pclose(fp);
    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

\$

FIFO

Como hemos dicho antes, a los FIFO también se les llama conductos con nombre porque son *persistentes*; es decir, existen en el sistema de archivos. Los FIFO son más útiles que los conductos sin nombre porque permiten que intercambien datos de procesos no relacionados y, además, porque son residentes permanentes en el sistema de archivos. *(persistentes)*

¿Qué son los FIFO?

Un simple ejemplo con la utilización de comandos *shell* nos puede ayudar a comprender los FIFO. El comando *mkfifo(1)* crea FIFO:

```
mkfifo [option] name [...]
```

mkfifo crea un FIFO con nombre *name*. *option* es normalmente *-m mode*, donde *mode* indica el modo octal del nuevo FIFO, sujeto a modificaciones por la *umask* del

proceso que llama. Una vez que creamos el FIFO, podemos utilizarlo como si fuera parte de una línea de conductos normal.

El siguiente ejemplo envía la salida de popen a través de un FIFO, con el nombre imaginativo de fifo1, que, en su momento, envía su salida a través del comando cut.

Primero, creamos el FIFO utilizando el siguiente comando:

```
$ mkfifo -m 600 fifo1
```

A continuación, después de construir popen, si es necesario, ejecutamos los dos siguientes comandos:

```
$ cat < fifo1 | cut -c1-5 &
$ ./popen > fifo1
```

La salida de estos comandos shell es:

```
/*
 * po
 */
#include
#include
#include
#include
#include

#define
void
int m
{
    F
    C
    C
    I
    D
    K
    W
    P
    E
}
void
{
    p
    exit
}
[1]+ Done
$ cat <fifo1 | cut -c1,5
```

Ejecutándose en segundo plano, el comando `cat` lee su entrada de FIFO, `fifo1`. La entrada de `cat` es la salida del comando `cut`, que elimina todo menos los cinco primeros caracteres de cada línea de la entrada. Para terminar, la entrada de `cut` es la salida de programa `newrw`.

La entrada real de los comandos `shell` es el código truncado de mal aspecto que completa el fragmento. Si `fifo1` fuera un archivo normal, el resultado sería un archivo normal relleno con la salida de `popen`.

Cómo crear un FIFO

La llamada de función para crear un FIFO tiene el mismo nombre que la interfaz `shell`, `mkfifo`. Su sintaxis es muy similar a la de la llamada de sistema `open`:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

`mkfifo` crea un FIFO con nombre `pathname` con los bits de permisos especificados (en octal) por `mode`. Como es normal, el valor de `mode` será modificado por la `umask` del proceso.

NOTA

Para determinar por adelantado cuál será el modo de un archivo o de un directorio después de su modificación por `umask`, simplemente aplicamos AND al modo que estamos utilizando con el complemento del valor de `umask`. En código, tendrá la siguiente apariencia:

```
mode_t mode = 0666;
mode & ~umask;
```

Así, dada una `umask` de 022, `mode & ~umask` devuelve 0644.

Si tiene éxito, `mkfifo` devuelve 0. En otro caso, establece el valor de la variable de error `errno` y devuelve -1 al que llama. Los errores potenciales son EACCESS, EEXIST, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR y EROFS.

El Listado 17.4 crea un FIFO en el directorio actual.

LISTADO 17.4. NEWFIFO.C

```
/*
 * newfifo.c - Crea un FIFO
 */
#include <sys/types.h>
```

(continúa)

LISTADO 17.4. NEWFIFO.C (CONTINUACIÓN)

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    mode_t mode = 0666;

    if(argc != 2) {
        puts("USAGE: newfifo <name>");
        exit(EXIT_FAILURE);
    }
    if((mkfifo(argv[1], mode)) < 0) {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Para construir este programa con el archivo make proporcionado, ejecutamos make newfifo. Si hemos seguido el ejemplo anterior, tenemos que asegurarnos de borrar fifo1 antes de ejecutar newfifo. Unas cuantas ejecuciones de ejemplo de este programa darán como resultado la siguiente salida:

```
$ ./newfifo
USAGE: newfifo <name>
$ ./newfifo fifo1
$ ./newfifo fifo1
mkfifo: File exists
```

La primera vez, newfifo protesta porque no se le llamó de forma correcta; espera que se le pase el nombre de FIFO como único argumento. La segunda proporciona un nombre de archivo, así que newfifo lo crea silenciosamente. Como el archivo ya existía, la tercera llamada a mkfifo falla y se establece errno como EEXIST. La configuración EEXIST se corresponde con la cadena que ha impreso perror: mkfifo: File exists.

Cómo abrir y cerrar FIFO

La apertura, el cierre, la eliminación, la lectura y la escritura de FIFO utilizan las mismas llamadas de sistema open, close, unlink, read y write, respectivamente, que ya hemos visto (una de las virtudes de la abstracción “todo son archivos” de Linux). Como la apertura y el cierre de FIFO son idénticas a la apertura y cierre de conductos, puede que queramos repasar el programa que abre y cierra un conducto, cuyo listado está en la sección “Cómo abrir y cerrar conductos” vista anteriormente en este capítulo.

Sin embargo, debemos recordar unas cuantas cosas cuando leamos y escribamos un FIFO. Primero, ambos lados de un FIFO deben estar abiertos antes de poder utilizarlos.

Segundo, y más importante, es el comportamiento de un FIFO si se ha abierto utilizando el indicador O_NONBLOCK. Recordemos que los indicadores O_WRONLY y O_RDONLY se pueden unir lógicamente con OR con O_NONBLOCK. Si hemos abierto un FIFO con O_NONBLOCK y O_RDONLY, la llamada vuelve inmediatamente, pero si lo hemos abierto con un O_NONBLOCK y O_WRONLY, open devuelve un error y establece errno como ENXIO si no se ha abierto el FIFO para lectura.

Por otra parte, si O_NONBLOCK no está especificado en los indicadores de open, O_RDONLY hará que se bloquee open (que no vuelva) hasta que otro proceso abra el FIFO para escritura. Así mismo, O_WRONLY bloqueará hasta que el FIFO se abra para lectura.

Como en los conductos, la escritura de un FIFO que no está abierto para lectura envía un SIGPIPE al proceso escritor y establece errno como EPIPE. Después de que el último escritor cierra un FIFO, un proceso lector detectará EOF en su siguiente lectura. Como mencionamos en relación con los conductos, para asegurar las escrituras atómicas cuando múltiples procesos están escribiendo en un solo FIFO, cada escritura de un proceso debe tener un tamaño menor que PIPE_BUF.

Cómo leer y escribir FIFO

Sujetas a las consideraciones explicadas al final de la última sección, la lectura y la escritura de FIFO son muy similares a la lectura y la escritura de conductos y archivos normales.

El siguiente ejemplo está, de alguna forma, relacionado. El Listado 17.5, rdfifo, crea y abre un FIFO para lectura, mostrando la salida del FIFO en stdout. El siguiente programa, wrfifo, mostrado en el Listado 17.6, abre el FIFO para escritura. Especialmente interesante es el proceso de ejecutar varias instancias del escritor en diferentes ventanas y vigilar sus salidas en la ventana del lector.

LISTADO 17.5. RDFIFO.C

```
/*
 * rdfifo.c - Creación de un FIFO y lectura desde él
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

int main(void)
{
    int fd; /* Descriptor del FIFO */
    int len; /* Bytes leídos del FIFO */
    char buf[PIPE_BUF];
```

(continúa)

LISTADO 17.5. RDFIFO.C (CONTINUACIÓN)

```

mode_t mode = 0666;

if((mkfifo("fifo1", mode)) < 0) {
    perror("mkfifo");
    exit(EXIT_FAILURE);
}

/* Apertura del FIFO de sólo lectura */
if((fd = open("fifo1", O_RDONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}

/* Lectura y visualización de la salida del FIFO hasta EOF */
while((len = read(fd, buf, PIPE_BUF - 1)) > 0)
    printf("rdfifo read: %s", buf);

close(fd);

exit(EXIT_SUCCESS);
}

```

LISTADO 17.6. WRIFO.C

```

/* wrfifo.c - Escritura de un FIFO bien conocido
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
#include <time.h>

int main(void)
{
    int fd;           /* Descriptor del FIFO */
    int len;          /* Bytes escritos en el FIFO */
    char buf[PIPE_BUF]; /* Se asegura de escrituras atómicas */
    time_t tp;        /* para la llamada de tiempo */

    /* Nos identificamos */
    printf("I am %d\n", getpid());

    /* Abre el FIFO de sólo escritura */
    if((fd = open("fifo1", O_WRONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
}

```

(continúa)

LISTADO 17.6. WRIFO.C (CONTINUACIÓN)

```

/* Genera algunos datos para escribir */
while(1) {
    /* Obtiene la hora actual */
    time(&tp);

    /* Crea la cadena para escribir */
    len = sprintf(buf, "wrfifo %d sends %s", getpid(), ctime(&tp));

    /*
     * Utiliza (len + 1) porque sprintf no cuenta
     * el null final
     */
    if((write(fd, buf, len + 1)) < 0) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }
    sleep(3);
}

close(fd);
exit(EXIT_SUCCESS);
}

```

Podemos construir estos programas ejecutando make rdfifo wrfifo en el directorio de código fuente de este capítulo. Mostramos la salida de estos programas en la Figura 17.4. El lector, rdfifo, se está ejecutando en la xterm grande. Para volver a crear el ejemplo, eliminamos primero fifo1 si existe. Después, abrimos cuatro xterms. En una, iniciamos wrfifo. Después, en las otras tres, ejecutamos rdfifo. Cada una de las tres xterms más pequeñas está ejecutando una instancia del escritor, wrfifo. El PID de cada escritor se muestra en su pantalla. Cada tres segundos, los escritores envían un mensaje que consiste en sus PID y la hora actual del FIFO, fifo1. Como podemos ver, el lector muestra el mensaje recibido y lo preajusta con rdfifo read para diferenciar su salida de la lectura entrada del FIFO.

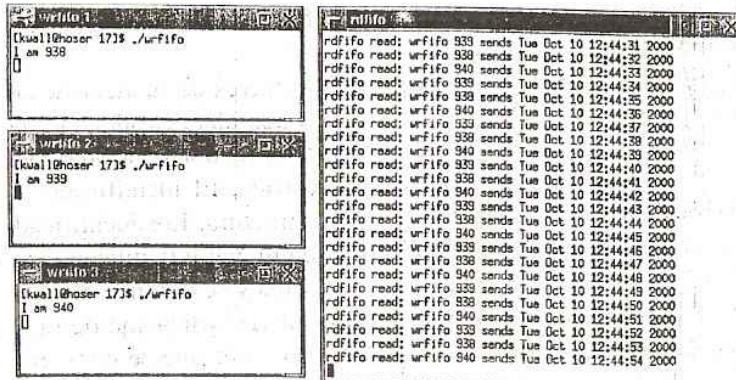


FIGURA 17.4. Múltiples procesos que escriben en el mismo FIFO.

Está claro que estos programas constituyen una aplicación cliente-servidor primitiva. El servidor es `rdfifo`; procesa mensajes que se le envían desde FIFO. Los clientes son los procesos `wrfifo`, cuyo único propósito es enviar mensajes al servidor.

Una aplicación cliente-servidor más sofisticada realizaría algún tipo de procesamiento en los datos que recibe y enviaría algún tipo de notificación o dato al cliente o clientes. Una explicación en profundidad de las aplicaciones cliente-servidor está fuera del alcance de este libro. No obstante, hemos escrito una de estas aplicaciones.

Introducción del IPC del sistema V

El IPC del sistema V tiene una utilización escasa en las nuevas aplicaciones porque ha sido sobrepasado por el IPC de POSIX. No obstante, lo explicamos en este libro porque probablemente nos lo encontraremos en programas más antiguos que fueron escritos antes de que apareciera el estándar IPC POSIX. Los tres tipos tienen la misma interfaz básica y el mismo diseño general. Esta sección introduce los conceptos fundamentales del IPC del sistema V y revisa las características y expresiones de programación comunes de los semáforos, colas de mensajes y memoria compartida.

Conceptos claves del IPC del sistema V

Las estructuras IPC (semáforos, colas de mensajes y segmentos de memoria compartida) están en el núcleo, como los conductos, en lugar de en el sistema de archivos, como los FIFO. A veces nos referimos a ellas colectivamente como a objetos IPC para evitar decir o escribir "semáforos, colas de mensajes y segmentos de memoria compartida". De forma similar, el término objeto IPC se utiliza para referirnos a uno de los tipos de estructuras sin tener que especificar cuál. La Figura 17.5 muestra cómo se comunican procesos no relacionados mediante un objeto IPC.

Como podemos ver en la Figura 17.5, los objetos IPC se mantienen en el núcleo (en realidad en su memoria), permitiendo así que procesos no relacionados (procesos que no tienen un padre común) se comuniquen utilizando uno de los mecanismos IPC, memoria compartida, semáforos o colas de mensajes. Los datos fluyen libremente entre los procesos utilizando los mecanismos IPC.

Nos referimos y accedemos a cada objeto a través de su identificador, un entero positivo único que identifica el objeto y su tipo. Cada identificador es único para cada tipo, pero podemos utilizar el mismo valor de identificador para una cola de mensajes, un semáforo y un segmento de memoria compartida. El identificador se convierte en el manipulador de todas las operaciones de la estructura. Los identificadores de estructura IPC no son enteros positivos pequeños que se utilizan y reutilizan como los descriptores de archivos. De hecho, según la estructura se crea y se elimina, el número del identificador, formalmente llamado número de secuencia de utilización de ranura, se incrementa hasta que alcanza un valor máximo, momento en el que se convierte en 0 y vuelve a comenzar. El valor máximo depende del sistema operativo y del hardware. En Linux, los identificadores se declaran como enteros, así que el máximo valor posible es 65.535.

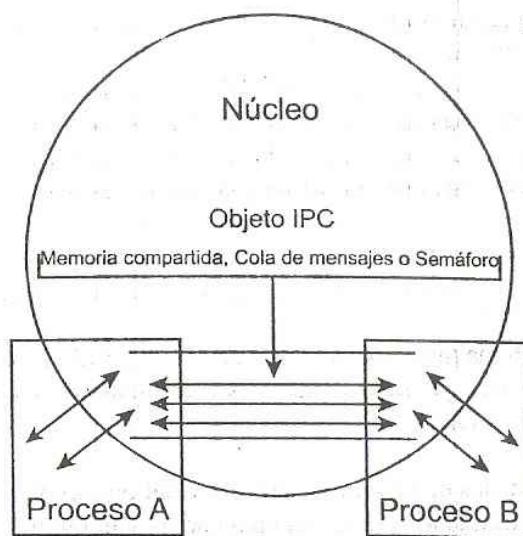


FIGURA 17.5. Los objetos IPC permiten que procesos no relacionados intercambien datos.

Cada estructura IPC se crea con una función `get`: `semget` para semáforos, `msgget` para colas de mensajes y `shmget` para memoria compartida. Cada vez que se crea un objeto utilizando una función `get`, el proceso que llama debe especificar una clave de tipo `key_t` (declarada en `<sys/types.h>`) que utiliza el núcleo para generar el identificador. El núcleo Linux 2.2.x define `key_t` como un entero.

Después de haberse creado la estructura IPC, llamadas posteriores a la función `get` utilizando la misma clave no crean una estructura, sino que devuelven el identificador asociado a la existente. Esto permite que dos o más procesos llamen a una función `get` con la misma clave para establecer un canal IPC.

El desafío pasa a ser cómo asegurar que todos los procesos que quieran utilizar la misma estructura IPC utilicen la misma clave. En un método, el proceso que realmente crea la estructura pasa la clave `IPC_PRIVATE` a la función `get`, que garantiza que se creará una nueva estructura. El creador de la estructura IPC almacena el identificador devuelto en el sistema de archivos donde otros procesos pueden acceder a él. En el caso de que el padre haga `fork` y `exec` a un nuevo proceso, éste pasa el identificador a sus procesos hijos como uno de los argumentos de la función `exec` que los crea.

Otro método de comunicar la clave es almacenarla en un archivo de encabezamiento común, de forma que todos los programas que incluyan dicho archivo tengan acceso a la misma clave. Un problema de este método es que ningún proceso sabe si está creando una estructura o sencillamente accediendo a una que ya ha sido creada por otro proceso. Otro problema es que la clave podría estar ya en uso en un programa no relacionado. Como resultado, el proceso que utiliza la clave debe tener código para manipular esto como una posibilidad.

Un tercer método implica la utilización de la función `ftok`, que tiene un nombre de ruta y un solo carácter `char` llamado identificador de proyecto y que devuelve una clave, que se pasa a la función `get` apropiada. Es nuestra responsabilidad asegurarnos de que el nombre de ruta y el identificador de proyectos son conocidos por todos los proyectos por adelantado. Podemos hacerlo utilizando uno de los métodos descritos anteriormente:

incluir ambos en un archivo de encabezamiento común o almacenarlos en un archivo de configuración predefinido.

Por desgracia, `ftok` tiene un serio defecto: No está garantizado que genere una clave única, lo que crea los mismos problemas que el segundo método mencionado anteriormente. Debido a los problemas potenciales derivados de la utilización de `ftok`, este capítulo lo descarta. `ftok` genera una clave no única en las siguientes situaciones:

- Cuando dos enlaces simbólicos diferentes se enlazan con el mismo archivo.
- Cuando los primeros 16 bits del número de ínidio del nombre de ruta tienen el mismo valor.
- Cuando un sistema tiene dos unidades de disco con el mismo número menor, lo que ocurre en sistemas que tienen múltiples controladores de disco. El número de unidad mayor será diferente, pero el menor puede ser el mismo.

Dadas las debilidades de la implementación `ftok` de Linux, recomendamos encarecidamente a los lectores que lo consideren un anatema y lo ignoren.

Además de crear una clave, las funciones `get` también aceptan un argumento `flags` que controla su comportamiento. Si la clave especificada no está ya en uso para el tipo de estructura deseada y el bit `IPC_CREAT` está establecido en `flag`, se creará una nueva estructura.

Cada estructura IPC tiene un modo, un conjunto de permisos que se comportan de forma análoga al modo del archivo (según se pasan a `open`), excepto que las estructuras IPC no tienen el concepto de permisos de ejecución. Cuando creamos una estructura IPC, debemos unir permisos específicos con OR, utilizando la notación octal definida para las llamadas de sistema `open` y `creat`, en el argumento `flags` pasado a `get` o puede que no podamos acceder a la estructura creada. Veremos ejemplos específicos posteriormente. Como era de esperar, el IPC del sistema V incluye una llamada para cambiar los permisos de acceso y la propiedad de las estructuras IPC.

Problemas con el IPC del sistema V

El IPC del sistema V tiene varios inconvenientes. Primero, la interfaz de programación es compleja comparada con los beneficios que proporciona. Segundo, las estructuras IPC son un recurso más limitado que algo como el número de archivos que puede soportar un sistema o el número de procesos activos que permite el sistema. Tercero, además de ser un recurso limitado, las estructuras IPC no mantienen una cuenta de referencia, que es un contador del número de procesos que están utilizando la estructura. Como resultado, el IPC del sistema V no tiene un método automático para reclamar estructuras IPC abandonadas.

Por ejemplo, si un proceso crea una estructura, coloca datos en ella y termina sin eliminarla apropiadamente a ella y a sus datos asociados, la estructura permanece hasta que ocurre uno de los siguientes casos:

- Se reinicia el sistema.
- Se elimina específicamente utilizando el comando `ipcrm`.
- Otro proceso, con permisos de acceso adecuados, lee los datos, la elimina o hace ambas cosas.

Los inconvenientes constituyen un defecto de diseño significativo.

Para terminar, las estructuras IPC, como dijimos anteriormente, están sólo en el núcleo; son desconocidas para el sistema de archivos. Como resultado, realizar E/S en ellas requiere aprender otra interfaz de programación. A falta de descriptores de archivos, no podemos utilizar E/S *multiplex* a través de una llamada de sistema *select*. Si un proceso necesita esperar E/S en una estructura IPC, debe utilizar algún tipo de bucle de espera ocupada. Un bucle de espera ocupada, un bucle que comprueba permanentemente alguna condición de cambio, es casi siempre una práctica de programación pobre, porque consume ciclos de CPU innecesariamente. Estos bucles son especialmente perniciosos en Linux, que tiene varios métodos para implementar esperas no ocupadas, como el bloqueo de E/S, la llamada de sistema *select* y las señales.

Linux y el IPC del sistema V

El IPC del sistema V es bien conocido y se utiliza con frecuencia, pero su implementación en Linux es mala. La versión Linux también precede al IPC POSIX, pero algunos programas Linux la implementan incluso aunque está disponible en los núcleos 2.2.x. El IPC POSIX tiene una interfaz muy similar a la del sistema V explicado en este capítulo y en el anterior, pero elimina algunos de los problemas que tenía el sistema V y simplifica la interfaz. El problema es que aunque el IPC del sistema V es estándar, también está implementado pobemente en Linux.

El resultado de esta situación es que Linux, que se esfuerza (y tiene gran éxito) en ser compatible con POSIX, implementa una versión temprana del IPC POSIX y la versión del sistema V. La dificultad es que el sistema V está bien establecido y es más común, pero la versión POSIX es mejor y más fácil de utilizar y tiene una interfaz más uniforme para interactuar programáticamente con los tres tipos de objetos IPC. ¿El resultado? Nosotros hemos elegido romper nuestra regla y cubrir lo que es más posible que encontraremos tanto en los programas existentes como en los nuevos en lugar de explicar lo correcto; es decir, el IPC POSIX.

Además, surgen problemas adicionales cuando tratamos con los semáforos. Los semáforos del sistema V se creaban en las Eras Oscuras para afrontar la horda de problemas que surgían cuando múltiples hilos de ejecución de un solo proceso (y de múltiples procesos) necesitaban acceder a los mismos recursos de sistema de forma casi simultánea. Los programadores con experiencia, incluso los programadores *Über*, buscarán otra solución antes de dirigirse a la multilectura, porque es difícil de implementar correctamente.

La explicación de los semáforos en este capítulo se ha simplificado porque la versión del sistema V se creó con los procesos multihilo en mente. No obstante, los semáforos son bastante útiles en programas de un solo hilo y estándares, como veremos en este capítulo. La interfaz del semáforo POSIX es más sencilla pero no muy utilizada en estos momentos en los programas de Linux.

Memoria compartida

La memoria compartida es el primero de los tres tipos de verdaderos IPC que vamos a aprender. En su conjunto, a los tres tipos normalmente se les llama IPC del sistema V,

porque se originaron con el sistema V UNIX, originalmente lanzado por AT&T. También los soportan las implementaciones derivadas del UNIX BSD y otros sistemas operativos semejantes a UNIX, incluido Linux.

La memoria compartida es una región de memoria (un segmento) configurada por el núcleo con el propósito de servir para el intercambio de información entre múltiples procesos. Suponiendo que los permisos del segmento estén establecidos de forma adecuada, cada proceso que desee acceder al segmento puede hacer correspondencia con él en su propio espacio de direcciones privado. Si un proceso actualiza los datos del segmento, dicha actualización es visible inmediatamente para los demás procesos. Un segmento creado por un proceso puede ser leído o escrito (o ambos) por otros procesos. El mismo nombre de *memoria compartida* indica que múltiples procesos comparten el acceso al segmento y a la información que contiene.

Cada proceso recibe su propia correspondencia de la memoria compartida en su espacio de direcciones. De hecho, la memoria compartida representa conceptualmente a los archivos de correspondencia de memoria. La mostramos en la Figura 17.6.

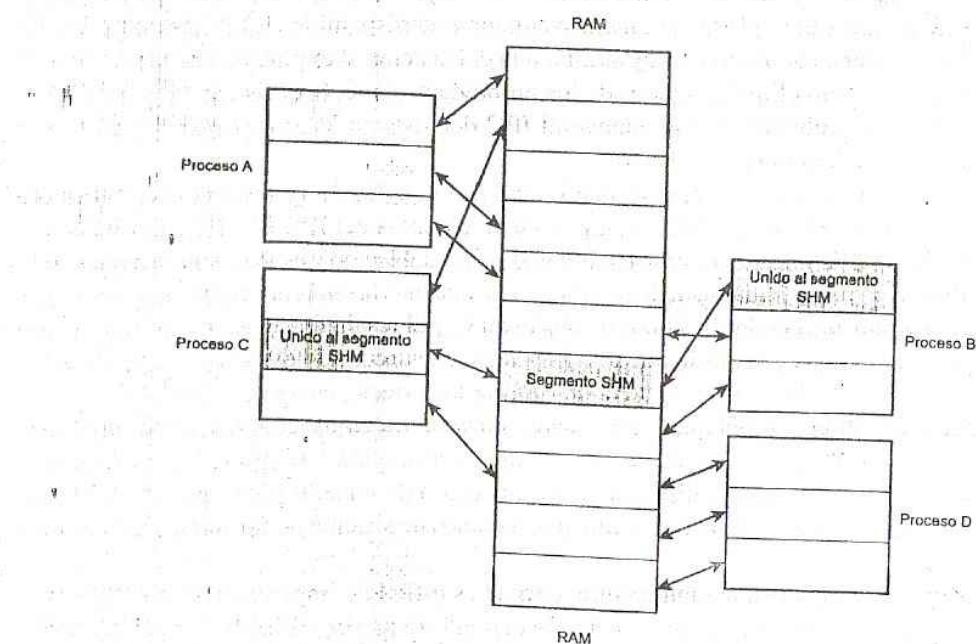


FIGURA 17.6. Los procesos hacen corresponder segmentos de memoria compartida en su propio espacio de direcciones.

La Figura 17.6 simplifica mucho el tema de la memoria compartida porque el segmento de memoria compartida puede consistir en datos en la RAM física así como en páginas de memoria que se han cambiado al disco. Lo mismo se puede aplicar al espacio de direcciones de los procesos que se adjunta al segmento de la memoria compartida.

Aun así, la figura muestra un segmento de memoria compartida (SHM) creado en la memoria principal (mostrado como un cuadro sombreado). Los cuadros sombreados de

los procesos B y C muestran que los dos procesos han hecho corresponder el segmento en sus propios espacios de direcciones. La figura también muestra que cada uno de los cuatro procesos tiene su propio espacio de direcciones que se hace corresponder con alguna parte de la RAM física. Sin embargo, el espacio de direcciones de un proceso no está disponible para los demás.

Naturalmente, como la transferencia de datos se produce estrictamente en la memoria (ignorando la posibilidad de que una o más páginas se hayan cambiado al disco), la memoria compartida es un método rápido de comunicación entre procesos. Tiene muchas de las ventajas de los archivos de correspondencia de memoria.

Cómo crear un segmento de memoria compartida

La llamada para crear un segmento de memoria compartida es `shmget`. Su prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flags);
```

`flags` puede ser uno o varios de `IPC_CREAT`, `IP_EXCL` y un conjunto de bits de permisos (modos) unidos por OR. Estos bits se especifican en octal. `IPC_EXCL` se asegura de que si el segmento ya existe, la llamada falla en lugar de devolver el identificador de un segmento existente.

`IPC_CREAT` indica que se debería crear un nuevo segmento si no hay ya un segmento asociado con `key`. `key` es `IPC_PRIVATE` o la clave devuelta por la función `ftok`. El argumento `size` especifica el tamaño del segmento, pero se redondeará al valor de `PAGE_SIZE`, que es el tamaño natural de una página para un procesador dado (4 KB para los procesadores Intel actuales, 8 KB en Alpha). `shmget` devuelve el identificador de segmento si tiene éxito o -1 si se produce un error.

El Listado 17.7, `mkshm.c`, crea un segmento de memoria compartida y muestra el identificador que devuelve `shmget`. Para construir este programa utilizando el archivo make facilitado, ejecutamos `make mkshm`.

LISTADO 17.7. MKSHM.C

```
/*
 * mkshm.c - Crea e inicia un segmento de memoria compartida
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 4096 /* Tamaño del segmento */
```

(continúa)

LISTADO 17.7. MKSHM.C (CONTINUACIÓN)

```

int main(void)
{
    int shmid;

    if((shmid = shmget(IPC_PRIVATE, BUFSZ, 0666)) < 0) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    printf("segmento creado: %d\n", shmid);
    system("ipcs -m");

    exit(EXIT_SUCCESS);
}

```

Mostramos a continuación un ejemplo de ejecución de este mkshm:

```

$ ./mkshm
segment created: 40833

----- Segmentos de memoria compartida -----
key      shmid   owner   perms   bytes   nattch   status
0x0000000000 11375   kwall    666     4096       0

```

Como muestra la salida, mkshm ha creado con éxito un segmento de memoria compartida. La penúltima columna de la salida de ipcs -m, nattch, muestra el número de procesos que tienen el segmento adjunto (lo que significa que han hecho corresponder el segmento dentro de su propio espacio de direcciones). Observemos que no hay ningún proceso que tenga el segmento adjunto. Lo único que hace shmget es crear el segmento de memoria compartida; los procesos que quieran hacerlo corresponder en su propio espacio de memoria, lo que se llama adjuntar al segmento, deben hacerlo explícitamente, utilizando la llamada shmat que explicamos en la siguiente sección.

Cómo adjuntar un segmento de memoria compartida

No podemos utilizar un segmento de memoria compartida hasta que lo adjuntemos. De forma similar, cuando terminemos de trabajar con un segmento de memoria compartida, debemos quitarlo de nuestro espacio de direcciones. Se adjunta utilizando la llamada shmat, y quitarlo requiere una llamada shmdt. Estas rutinas tienen los siguientes prototipos:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int flags);
int shmdt(char *shmaddr);

```

shmid es el identificador del segmento al que queremos adjuntar. En shmat, si shmaddr es 0, el núcleo hará correspondencia del segmento en el espacio de direcciones del proceso que llama en una dirección de su elección. Si shmaddr no es 0, esto indica la dirección en la que el núcleo deberá hacer la correspondencia del segmento (obviamente, esto es un tiro al azar, así que tenemos que establecer shmaddr siempre como 0). flags puede ser SHM_RDONLY, lo que significa que se adjuntará como de sólo lectura. En otro caso, el comportamiento predeterminado es que el segmento se adjunte como de lectura y escritura. Si la llamada shmat tiene éxito, devuelve la dirección del segmento adjunto. En otro caso, devuelve -1 y establece errno.

shmctl desconecta el segmento que está adjunto en shmaddr del espacio de direcciones del proceso que llama. La dirección debe haber sido devuelta previamente por una llamada a shmget.

El primer ejemplo, el Listado 17.8, adjunta y quita un segmento de memoria compartida. Construimos este programa ejecutando make atshm.

LISTADO 17.8. ATSHM.C

```
/*
 * atshm.c - Adjunta un segmento de memoria compartida.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int shmid; /* ID del segmento */
    char *shmbuf; /* Dirección del proceso */

    /* Espera un id de segmento en la línea de comandos */
    if(argc != 2) {
        puts("UTILIZACIÓN: atshm <identifier>");
        exit(EXIT_FAILURE);
    }
    shmid = atoi(argv[1]);

    /* Adjunta el segmento */
    if((shmbuf = shmat(shmid, 0, 0)) < (char *)0) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    /* ¿Dónde está adjunto? */
    printf("segmento adjunto en %p\n", shmbuf);

    /* ¡Realmente estamos adjuntos! */
    system("ipcs -m");

    /* Lo quita */
    if((shmctl(shmbuf)) < 0) {
```

(continúa)

LISTADO 17.8: ATSHM.C (CONTINUACIÓN)

```

    perror("shmdt");
    exit(EXIT_FAILURE);
}
puts("segmento quitado");

/* Si, lo hemos hecho */
system("ipcs -m");

exit(EXIT_SUCCESS);
}

```

shmat devuelve un puntero a char, de forma que cuando comprueba su código de retorno, atshm difunde el cero como un char * para evitar avisos de compilador. El ejemplo también utiliza el comando ipcs para confirmar que el proceso que llama ha adjuntado y quitado de forma satisfactoria el segmento de memoria. Para ejecutar atshm, le pasamos el identificador devuelto por mkshm. La siguiente salida lo ilustra. Observemos que el número de segmentos adjuntos, nattch, crece y disminuye.

```

$ ./atshm 11137
segmento adjunto en 0x2aabf000

----- Segmentos de memoria compartidas -----
key      shmid     owner     perms     bytes     nattch    status
0x00000000 11137     kwall     666       4096        1

segment detached

----- Segmentos de memoria compartida -----
key      shmid     owner     perms     bytes     nattch    status
0x00000000 11137     kwall     666       4096        0

$ 

```

El Listado 17.9, opshm.c, muestra cómo escribir en la lectura de un segmento de memoria compartida. Se adjunta al segmento, escribe datos en él y el *buffer* pasa al archivo opshm.out. Construimos este programa utilizando el archivo make suministrado ejecutando make opshm. Lo ejecutamos pasando el identificador devuelto por mkshm.

LISTADO 17.9. OPSHM.C

```

/*
 * opshm.c - Lectura y escritura de un segmento de memoria compartida
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

```

(continúa)

LISTADO 17.9. OPSHM.C (CONTINUACIÓN)

```
#include <ctype.h>

#define BUFSZ 4096

int main(int argc, char *argv[])
{
    int shmid; /* ID de segmento */
    char *shmbuf; /* Dirección del proceso */
    int fd; /* Descriptor del archivo */
    int i; /* Contador */

    /* Se espera un id de segmento en la línea de comandos */
    if(argc != 2) {
        puts("UTILIZACIÓN: opshm <identifier>");
        exit(EXIT_FAILURE);
    }
    shmid = atoi(argv[1]);

    /* Se adjunta al segmento */
    if((shmbuf = shmat(shmid, 0, 0)) < (char *)0) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    /* Tamaño de shmbuf apropiado */
    if((shmbuf = malloc(sizeof(char) * BUFSZ)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    for(i = 0; i < BUFSZ; ++i)
        shmbuf[i] = rand();

    /* Se escribe el contenido en bruto del segmento en el archivo */
    fd = open("opshm.out", O_CREAT | O_WRONLY, 0600);
    write(fd, shmbuf, BUFSZ);
    free(shmbuf); /* No quiere pérdidas de memoria */

    exit(EXIT_SUCCESS);
}
```

Después de ejecutar el programa, encontraremos opshm.out en el directorio actual. Sin embargo, no le hagamos cat, porque está lleno de basura aleatoria (ya que opshm utilizó la llamada a rand para generar datos y llenar el segmento de memoria compartida). Hacer cat probablemente hará que nuestro terminal quede inutilizable.

Colas de mensajes

Una cola de mensajes es una lista de mensajes enlazada almacenada con el núcleo e identificada por procesos por un *identificador de colas de mensajes*, del mismo tipo que

el del segmento de memoria compartida. Por brevedad y conveniencia, este capítulo utiliza los términos *cola* e *ID de cola* para referirse a colas de mensajes y a *identificadores de colas de mensajes*. Si añadimos un mensaje a una cola, parece un FIFO porque los nuevos mensajes se añaden al final de la cola. Sin embargo, a diferencia de los FIFO, los mensajes de la cola se pueden recuperar en un orden relativamente arbitrario porque, como veremos más adelante, podemos utilizar un tipo de mensaje para recuperar un mensaje fuera de orden. En este aspecto, las colas son similares programáticamente a las matrices asociativas.

Todas las funciones de manipulación de colas están declaradas en `<sys/msg.h>`, pero también debemos incluir `<sys/types.h>` y `<sys/IPC.h>` para acceder a las declaraciones de constante y tipo que contienen estos archivos de encabezamiento. Para crear una nueva cola o para abrir una existente, utilizamos la función `msgget`. Para añadir un nuevo mensaje al final de una cola, utilizamos `msgsnd`. Para sacar un mensaje de una cola, utilizamos la llamada `msgrcv`. `msgctl` nos permite manipular las características de la cola y eliminarla, suponiendo que el proceso que llama sea su creador o tenga permisos de superusuario.

Cómo crear y abrir una cola

La función `msgget` crea una nueva cola o abre una existente. Su prototipo es:

```
#include <sys/types.h>
#include <sys/IPC.h>
#include <sys/msg.h>
int msgget(key_t key, int flags);
```

Si la llamada tiene éxito, devuelve el ID de cola de la cola nueva o existente y correspondiente al valor contenido en `key`. Si `key` es `IPC_PRIVATE`, se crea una nueva cola utilizando el valor de clave que genera la implementación subyacente; utilizando `IPC_PRIVATE` se garantiza la creación de una nueva cola si no se han excedido los límites del sistema en los números de colas o el número total de bits de todas las colas.

De forma similar, si `key` no es `IPC_PRIVATE`, si no se corresponde con una cola existente que tenga la misma `key` y si el bit `IPC_CREAT` está establecido en el argumento `flags`, se crea una nueva cola. En otro caso, es decir, si `key` no es `IPC_PRIVATE` y el bit `IPC_CREAT` no está establecido en `flags`, `msgget` devuelve el ID de cola de la cola existente asociada con `key`. Si `msgget` falla, devuelve -1 y establece la variable de error `errno`.

El Listado 17.10, `mkq.c`, crea una nueva cola de mensajes. Si lo ejecutamos una segunda vez, en lugar de crear la cola especificada, simplemente abre la existente. Podemos construir este programa utilizando el archivo `make` suministrado ejecutando `make mkq`.

LISTADO 17.10. MKQ.C

```
/*
 * mkq.c - Crea una cola de mensajes de IPC SysV.
 */
#include <sys/types.h>
#include <sys/IPC.h>
```

(continúa)

LISTADO 17.10. MKQ.C (CONTINUACIÓN)

```
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int qid; /* El identificador de la cola */
    key_t key; /* La clave de la cola */

    key = 123;

    /* Crea la cola */
    if((qid = msgget(key, IPC_CREAT | 0666)) < 0) {
        perror("msgget:create");
        exit(EXIT_FAILURE);
    }
    printf("creado ID de cola = %d\n", qid);

    /* Vuelve a abrir la cola */
    if((qid == msgget(key, 0)) < 0) {
        perror("msgget:open");
        exit(EXIT_FAILURE);
    }
    printf("abierto. ID de cola = %d\n", qid);

    exit(EXIT_SUCCESS);
}
```

cola crear, escribir, leer, borrar

La salida de este programa debería parecerse a la siguiente:

```
$ ./mkq
created ID de cola = 128
opened ID de cola = 128
$
```

Si la primera llamada a `msgget` tiene éxito, `mkq` muestra el ID de cola recién creado y llama a `msgget` una segunda vez. Si la segunda llamada tiene éxito, `mkq` también informa. Sin embargo, la segunda llamada sólo abre la cola existente en lugar de crear una nueva. Observemos que la primera llamada especifica permisos de lectura-escritura para todos los usuarios que utilicen la notación octal estándar.

CONSEJO

Cuando creamos un objeto del IPC del sistema V, la `umask` del proceso no modifica los permisos de acceso del objeto. Si no establecemos permisos de acceso, el modo predeterminado es 0, lo que significa que ni siquiera el creador de la estructura tiene acceso de lectura o escritura a ella.

Cómo escribir un mensaje en una cola

Para añadir un nuevo mensaje al final de una cola, utilizamos la función `msgsnd`, cuyo prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flags);
```

`msgsnd` devuelve 0 si tiene éxito, pero si falla, devuelve -1 y establece la variable de error global `errno` con uno de los valores de la siguiente lista:

- `EGAIN`
- `EACCES`
- `EFAULT`
- `EIDRM`
- `EINTR`
- `EINVAL`
- `ENONEM`

El argumento de `msqid` debe ser un ID de cola devuelto por una llamada previa a `msgget`. `nbytes` es el número de bytes del mensaje enviado, que no debería terminar en nulo. `ptr` es un puntero a una estructura `msgbuf`, que consiste en un tipo de mensaje y los bytes de datos que conforman dicho mensaje. `msgbuf` está definida en `<sys/msg.h>` así:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Esta declaración de estructura es realmente sólo una plantilla porque `mtext` debe tener el tamaño de los datos que se están almacenando, que se corresponde con el valor pasado en el argumento `nbytes`, menos cualquiera que termine en nulo. `mtype` puede ser cualquier entero `long` mayor que cero. El proceso que llama debe tener también acceso de lectura a la cola. Para terminar, la variable `flags` es o bien 0 ó `IPC_NOWAIT`. `IPC_NOWAIT` produce un comportamiento similar al del indicador `O_NONBLOCK` pasado a la llamada de sistema `open`: Si el número total de mensajes individuales de la cola o el tamaño de la cola, en bytes, es igual al límite especificado en el sistema, `msgsnd` vuelve inmediatamente y establece `errno` como `EAGAIN`. Como resultado, no seremos capaces de añadir ningún mensaje más a la cola hasta que al menos uno de ellos haya sido leído.

Si `flags` es 0 y se ha escrito el máximo número de mensajes o el número total de bytes de datos en la cola, la llamada `msgsnd` se bloquea (no vuelve) hasta que la condición haya desaparecido. Para hacer desaparecer la condición, se deben leer los mensajes de la cola, se debe eliminar la cola (lo que establece `errno` como `EIDRM`) o se debe capturar una señal y el manipulador devuelto (lo que establece `errno` como `EINTR`).

CONSEJO

La plantilla de la estructura msgbuf puede expandirse para que se ajuste a las necesidades de nuestra aplicación. Por ejemplo, si queremos mandar un mensaje que consista en un entero y en una matriz de caracteres de 10 bytes, sólo tenemos que declarar msgbuf así:

```
struct msgbuf {
    long mtype;
    int i;
    char c[10];
};
```

msgbuf es simplemente un long seguido de los datos del mensaje, que pueden ser formateados como queramos. El tamaño de la estructura declarada en el ejemplo es sizeof(msgbuf) - sizeof(long).

El Listado 17.11, qsnd, añade un mensaje al final de una cola que ya existe. Pasamos el ID de cola devuelto por mqopen como el único argumento de línea de comandos del programa. Construimos el programa utilizando el archivo make suministrado y ejecutando make qsnd.

LISTADO 17.11. Q SND.C

```
/*
 * qsnd.c - Envía un mensaje a una cola previamente abierta.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BUFSZ 512

/* Estructura del mensaje */
struct msg {
    long msg_type;
    char msg_text[BUFSZ];
};

int main(int argc, char *argv[])
{
    int qid; /* El identificador de la cola */
    int len; /* El tamaño de los datos enviados */
    struct msg pmsg; /* Puntero a la estructura de mensajes */
    /* Espera el ID de cola pasado en la linea de comandos */
    if(argc != 2) {
```

(continúa)

LISTADO 17.11. QSND.C (CONTINUACIÓN)

```

    puts("USAGE: qsnd <ID de cola>");
    exit(EXIT_FAILURE);
}
qid = atoi(argv[1]);

/* Obtiene el mensaje a añadir en la cola */
puts("Introducir el mensaje a enviar:");
if((fgets(&pmsg->msg_text, BUFSZ, stdin)) == NULL) {
    puts("no hay mensaje para enviar");
    exit(EXIT_SUCCESS);
}

/* Asocia el mensaje con este proceso */
pmsg.msg_type = getpid();
/* Añade el mensaje a la cola */
len = strlen(pmsg.msg_text);
if((msgsnd(qid, &pmsg, len, 0)) < 0) {
    perror("msgsnd");
    exit(EXIT_FAILURE);
}
puts("mensaje enviado");

exit(EXIT_SUCCESS);
}

```

Una ejecución de ejemplo de este programa tiene como resultado la siguiente salida. Observemos que el programa utiliza el ID de cola devuelto por mkq.

```

$ ./qsnd 128
Introduzca el mensaje a enviar:
Añadiendo el mensaje al ID de cola 128.
Mensaje enviado

```

El comando qsnd 128 tiene como resultado una línea de introducción de comandos donde introduce el mensaje a enviar y almacena la respuesta escrita (indicada en negrita) directamente en la estructura msg declarada anteriormente. Si msgsnd termina con éxito, muestra un mensaje a tal efecto. Observemos que qsnd establece el tipo de mensaje con el PID del proceso que llama. Esto nos permite recuperar más tarde (utilizando msgrcv) sólo los mensajes que envió este proceso.

Cómo leer mensajes de la cola

Para sacar un mensaje de la cola, utilizamos msgrcv, que tiene la sintaxis indicada aquí:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flags);

```

Si tiene éxito, `msgrcv` elimina el mensaje devuelto de la cola. Los argumentos son los mismos que acepta `msgsnd`, excepto que `msgrcv` rellena la estructura `ptr` con el tipo de mensaje y hasta `nbytes` de datos. El argumento adicional, `type`, corresponde al miembro `msg_type` de la estructura `msg` previamente explicada. El valor de `type` determina qué mensaje se devuelve, como podemos ver en la siguiente lista:

- Si `type` es 0, se devuelve el primer mensaje de la cola (el de más arriba).
- Si `type` es mayor que 0, se devuelve el primer mensaje cuyo `msg_type` sea igual a `type`.
- Si `type` es menor que 0, se devuelve el primer mensaje cuyo `msg_type` sea el menor valor menor o igual que el valor absoluto de `type`.

El valor de `flags` también controla el comportamiento de `msgrcv`. Si el mensaje devuelto es mayor que `nbytes`, se trunca el mensaje a `nbytes` si el bit `MSG_NOERROR` está establecido en `flags` (pero no se genera ninguna notificación de que se haya truncado el mensaje). En otro caso, `msgrcv` devuelve -1 para indicar un error y establecer `errno` como `E2BIG`. El mensaje también permanece en la cola.

Si el bit `IPC_NOWAIT` está establecido en `flags`, `msgrcv` vuelve inmediatamente y establece `errno` como `ENOMSG` si no hay disponibles mensajes del tipo especificado. En otro caso, `msgrcv` se bloquea hasta que se produzca una de las condiciones descritas anteriormente `msgsnd`.

NOTA

Podemos utilizar un valor negativo para el argumento `type` para crear un LIFO, un tipo de cola el último en entrar-el primero en salir, familiarmente conocido como una pila. Pasar un valor negativo en `type` nos permite recuperar mensajes de un tipo dado en un orden inverso respecto al de entrada en la cola.

El Listado 17.12, `qrd.c`, lee un mensaje de una cola poblada y creada previamente. La cola de la que leer se pasa como argumento de línea de comandos. El comando `make qrd` utiliza el archivo `make` proporcionado para construir este programa.

LISTADO 17.12. QRD.C

```
/*
 * qrd.c - Lee todos los mensajes de una cola de mensajes
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 512
```

(continúa)

LISTADO 17.12. QRD.C (CONTINUACIÓN)

```

/* Estructura del mensaje */
struct msg {
    long msg_type;
    char msg_text[BUFSZ];
};

int main(int argc, char *argv[])
{
    int qid; /* El identificador de cola */
    int len; /* El tamaño del mensaje */
    struct msg pmsg; /* Una estructura de mensajes */

    /* Espera el ID de cola pasado en la linea de comandos */
    if(argc != 2) {
        puts("USAGE: qrd <ID de cola>");
        exit(EXIT_FAILURE);
    }
    qid = atoi(argv[1]);

    /* Recupera y muestra un mensaje de la cola */
    len = msgrecv(qid, &pmsg, BUFSZ, 0, 0);
    if(len > 0) {
        (&pmsg)->msg_text[len] = '\0';
        printf("leyendo el ID de cola: %05d\n", qid);
        printf("tipo de mensaje: %05ld\n", (&pmsg)->msg_type);
        printf("tamaño del mensaje: %d bytes\n", len);
        printf("texto del mensaje: %s\n", (&pmsg)->msg_text);
    } else {
        perror("msgrecv");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}

```

A continuación tenemos la salida de una ejecución de este programa. Como antes, utiliza el ID de cola creado por la ejecución de mkq anteriormente en esta sección. Además, lee el mensaje enviado por el ejemplo anterior de qsnd.

```

$ ./qrd 640
tipo de mensaje: 06360
tamaño del mensaje: 34 bits
texto del mensaje: Añadiendo un mensaje al ID de cola 128.
$ 

```

En el código del listado podemos ver que la lectura de una cola de mensajes es más sencilla que la escritura y que requiere menos código. Es de particular interés en el programa de ejemplo que recoge el primer mensaje de la cima de la cola porque pasa un type de 0. En este caso, como el PID del proceso de escritura del mensaje es conocido o pue-

de ser descubierto fácilmente, qrd podría haber pasado un type de 06360 y haber recibido el mismo mensaje de la cola.

Cómo eliminar colas de mensajes

La función msgctl ofrece un cierto grado de control sobre las colas de mensajes. Su prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Como es normal, msqid es el ID de cola de una cola existente. cmd puede ser uno de:

- IPC_RMID. Elimina la cola msqid.
- IPC_STAT. Rellena buf con la estructura msqid_ds de la cola y nos permite ver su contenido sin eliminar ningún mensaje. Como es una lectura no destructiva, podríamos considerarla similar a msgrcv.
- IPC_SET. Nos permite cambiar el UID, GID, modo de acceso y máximo número de bites permitidos de la cola.

El Listado 17.13, qctl.c, utiliza la llamada msgctl para eliminar una cola cuyo ID se pasa en la línea de comandos. Construimos qctl utilizando el archivo make suministrado y ejecutando make qctl.

LISTADO 17.13. QCTL.C

```
/*
 * qctl.c - Elimina una cola de mensajes.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int qid;

    if(argc != 2) {
        puts("USAGE: qctl <qid>");
        exit(EXIT_FAILURE);
    }
    qid = atoi(argv[1]);
```

(continúa)

LISTADO 17.13. QCTL.C (CONTINUACIÓN)

```

if((msgctl(qid, IPC_RMID, NULL)) < 0) {
    perror("msgctl");
    exit(EXIT_FAILURE);
}
printf("cola %d eliminada\n", qid);

exit(EXIT_SUCCESS);
}

```

Como el ejemplo de qrd también borró la cola previamente creada, tenemos que ejecutar mkq para crear una nueva antes de utilizar qctl. Mostramos un ejemplo:

```

$ ./mkq
creado ID de cola = 384
abierto ID de cola = 384
$ ipcs -q
----- Colas de mensajes -----
key      msqid      owner      perms  used-bytes messages
0x00000007b 384        kwall      666        0          0

$ ./qctl 384
cola 384 eliminada
$ ipcs -q
----- Colas de mensajes -----
key      msqid      owner      perms  used-bytes messages
$ 

```

La salida del programa muestra que se ha eliminado la cola especificada. mkq crea la cola. La llamada ipcs confirma que se ha creado la cola. Utilizando el ID de cola que devuelve mkq, qctl llama a msgctl, especificando el indicador IPC_RMID para eliminar la cola. Una segunda ejecución de ipcs confirma que qctl eliminó la cola.

El comando ipcs, utilizado en varios de los ejemplos, es parte del software IPC instalado por la mayoría de las distribuciones de Linux. Cuando lo ejecutamos, muestra el número y el estado de todas las estructuras del IPC del sistema V que hay en el sistema. El comando ipcrm eliminará la estructura IPC cuyo tipo e ID se especifiquen en la línea de comandos. Diríjase a las páginas del manual para obtener más información.

Semáforos

Los semáforos controlan el acceso a recursos compartidos, así como las luces de tráfico controlan el flujo del tráfico en una intersección. Son diferentes que otros métodos de IPC que hemos visto, porque no proporcionan información entre procesos, sino que sincronizan el acceso a recursos compartidos a los que no se debería acceder simultánea-

mente. En este aspecto, la utilización de un semáforo se parece más al bloqueo de un registro o de un archivo, excepto en que los semáforos se pueden aplicar a más recursos aparte de los archivos. Esta sección explica sólo el tipo de semáforo más sencillo; el *semáforo binario*. Tiene dos posibles valores: 0 cuando un proceso está bloqueado y no deberían acceder otros procesos y 1 cuando el recurso está desbloqueado.

¿Cómo se utilizan los semáforos? Cuando un proceso necesita acceder a un recurso controlado, como un archivo, primero comprueba el valor del semáforo, como un conductor mira si la luz está verde. Si el valor del semáforo es 0, lo que se corresponde con la luz roja, el recurso está en uso, así que el proceso se bloquea hasta que el recurso esté disponible (es decir, el valor del semáforo deje de ser cero). En terminología de IPC del sistema V, este bloqueo recibe el nombre de una espera. Si el semáforo tiene un valor positivo, lo que se corresponde con la luz verde de una intersección, el recurso asociado está disponible, así que el proceso disminuye el valor del semáforo, realiza sus operaciones en el recurso e incrementa el valor del semáforo para eliminar el bloqueo.

Cómo crear un semáforo

Naturalmente, antes de poder incrementar o disminuir el valor de un semáforo, éste debe existir. La llamada de función para la creación de un nuevo semáforo o para el acceso a uno existente es *semget*, cuyo prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flags);
```

semget devuelve el identificador del semáforo asociado con un conjunto de semáforos *nsems*. Se crea un nuevo conjunto de semáforos si *key* es *IPC_PRIVATE* o si *key* no está ya en uso y el bit *IPC_CREAT* está establecido en *flags*. Como con los segmentos de memoria compartida y las colas de mensajes, *flags* también puede unirse con OR con bits de permisos (en octal) para establecer los modos de acceso para el semáforo. Sin embargo, observemos que los semáforos tienen permisos de lectura y alteración, en lugar de lectura y escritura. Utilizan la noción de alteración en lugar de la de escritura porque realmente nunca les escribimos datos, sencillamente cambiamos (o alteramos) su estado incrementando o disminuyendo su valor. *semget* devuelve -1 y establece *errno* si se produce un error. En otro caso, devuelve el identificador del semáforo asociado con el valor de *key*.

NOTA

Las llamadas de semáforo del Sistema V operan en realidad en una matriz, o conjunto, de semáforos, en lugar de en solo un semáforo. Sin embargo, el objetivo de este capítulo es simplificar la explicación e introducirnos materia

(continúa)

(continuado)

en lugar de cubrir los semáforos en toda su complejidad. Independientemente de que estemos trabajando con un solo semáforo o con múltiples, el método básico es el mismo, pero es importante que comprendamos que los semáforos del Sistema V vienen en grupos. El IPC de POSIX IPC estandariza una interfaz de semáforos más sencilla, pero igualmente funcional.

La función **semop** es el caballo de batalla de la rutina de semáforos. Realiza operaciones en uno o más de los semáforos creados o a los que se ha accedido mediante la llamada **semget**. Su prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *semops, unsigned nops);
```

semid es un identificador de semáforo previamente devuelto por **semget** y señala al conjunto de semáforos a manipular. **nops** es el número de elementos de la matriz de la estructura **sembuf** a la que señala **semops**. La estructura **sembuf** se define así:

```
struct sembuf {
    short sem_num; /* Número de semáforo */
    short sem_op; /* La operación a realizar */
    short sem_flg; /* Indicadores que controlan la operación */
};
```

En la estructura **sembuf**, **sem_num** es un número de semáforo, de 0 a **nsems** - 1, **sem_op** es la operación a realizar y **sem_flg** modifica el comportamiento de **semop**. **sem_op** puede ser positivo, negativo o 0.

- Si **sem_op** es positivo, el recurso controlado por el semáforo es liberado y el valor del semáforo se incrementa.
- Si **sem_op** es negativo, el proceso que llama está indicando que quiere esperar hasta que el recurso controlado esté disponible, en cuyo momento el valor del semáforo disminuirá y el recurso será bloqueado por el proceso.
- Si **sem_op** es 0, el proceso que llama se bloqueará hasta que el semáforo sea 0; si ya es 0, la llamada vuelve inmediatamente. **sem_flg** puede ser **IPC_NOWAIT**, que tiene el comportamiento explicado anteriormente, o **SEM_UNDO**, lo que significa que la operación realizada se deshará cuando salga el proceso que llamaba al semáforo.

El Listado 17.14, **mksem.c**, crea un semáforo e incrementa su valor, marcando el recurso asociado como desbloqueado o disponible. Ejecutamos **make mksem** para construir este programa utilizando el archivo **make** suministrado.

LISTADO 17.14. MKSEM.C

```
/*
 * mksem.c - Crea e incrementa el valor de un semáforo.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int semid; /* Identificador del semáforo */
    int nsems = 1; /* Cuántos semáforos crear */
    int flags = 0666; /* Modo de lectura-alteración general */
    struct sembuf buf; /* Comportamiento del semáforo */

    /* Crea el semáforo con permisos de lectura-alteración general */
    semid = semget(IPC_PRIVATE, nsems, flags);
    if(semid < 0) {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    printf("semáforo creado: %d\n", semid);

    /* Configura la estructura */
    buf.sem_num = 0; /* Un solo semáforo */
    buf.sem_op = 1; /* Incrementa el valor del semáforo */
    buf.sem_flg = IPC_NOWAIT; /* No lo bloquea */

    if((semop(semid, &buf, nsems)) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    system("ipcs -s");
    exit(EXIT_SUCCESS);
}
```

A continuación tenemos un ejemplo de salida de mksem. Es probable que los valores de identificadores mostrados en otros sistemas sean diferentes.

```
$ ./mksem
semáforo creado: 0

----- Matrices de semáforo -----
key      semid     owner     perms   nsems   status
0x0000000000 0       kwall     666      1

$
```

El ejemplo utiliza **IPC_PRIVATE** para asegurar que se crea el semáforo como se ha solicitado y muestra el valor de retorno de **semget** para mostrar el identificador del semáforo.

La llamada `semop` inicia el semáforo de forma adecuada: Como se ha creado sólo un semáforo, `sem_num` es cero. Como el recurso imaginario no está en uso (más precisamente, el semáforo no está asociado por programa a ningún recurso en particular), `mksem` inicia su valor como 1, el equivalente a desbloqueado. Al no necesitar el comportamiento de bloqueo en esta instancia, el `sem_flg` está establecido como `IPC_NOWAIT`, así que la llamada vuelve inmediatamente. Para terminar, el ejemplo utiliza la llamada `system` para invocar el comando `ipcs` de nivel de usuario para confirmar que la estructura IPC solicitada existe.

Cómo controlar y eliminar semáforos

Ya hemos visto `msgctl` y `shmctl`, las llamadas que manipulan las colas de mensajes y los segmentos de memoria compartida. Como era de esperar, la función equivalente para los semáforos es `semctl`, cuyo prototipo es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);
```

`semid` identifica el conjunto de semáforos que queremos manipular y `semnum` especifica el semáforo en particular en el que estamos interesados. Este libro ignora las situaciones en las que hay múltiples semáforos en un conjunto, así que `semnum` (en realidad un índice dentro de una matriz de semáforos) siempre será 0.

Los valores posibles del argumento `cmd` están en la Tabla 17.1.

TABLA 17.1. VALORES DE CMD EN LA LLAMADA SEMCTL

| Comando | Description |
|----------|---|
| GETVAL | Devuelve el estado actual del semáforo (bloqueado o desbloqueado). |
| SETVAL | Establece el estado actual del semáforo como <code>arg.val</code> (el argumento <code>semun</code> lo explicamos en otro sitio en este capítulo). |
| GETPID | Devuelve el PID del proceso que llamó por última vez a <code>semop</code> . |
| GETNCNT | Hace que el valor de retorno de <code>semctl</code> sea el número de procesos que están esperando a que el valor del semáforo se incremente, es decir, el número de procesos esperando en el semáforo. |
| GETZCNT | Hace que el valor de retorno de <code>semctl</code> sea el número de procesos que están esperando a que el valor del semáforo sea cero. |
| GETALL | Devuelve el valor de todos los semáforos del conjunto asociado con <code>semid</code> . |
| SETALL | Establece el valor de todos los semáforos del conjunto asociado con <code>semid</code> con los valores almacenados en <code>arg.array</code> . |
| IPC_RMID | Elimina el semáforo con <code>semid</code> . |
| IPC_SET | Establece el modo (bits de permiso) del semáforo. |
| IPC_STAT | Cada semáforo tiene una estructura de datos, <code>semid_ds</code> , que describe completamente su configuración y comportamiento. <code>IPC_STAT</code> copia esta información de configuración en el miembro <code>arg.buf</code> de la estructura <code>semun</code> . |

Si la llamada `semctl` falla, devuelve -1 y establece `errno` apropiadamente. En otro caso, devuelve un valor entero de `GETNCNT`, `GETPID`, `GETVAL` o `GETZCNT`, dependiendo del valor de `cmd`.

Como es de imaginar, el argumento `semun` juega un papel vital en la rutina `semctl`. Debemos definirlo en nuestro código de acuerdo con la siguiente plantilla:

```
union semun {
    int val;                      /* Valor de SETVAL */
    struct semid_ds *buf;          /* buffer de IPC_STAT */
    unsigned short int *array;     /* buffer de GETALL y SETALL */
};
```

Por ahora, deberíamos empezar a comprender que la interfaz de semáforos del Sistema V es demasiado complicada para los simples mortales. No obstante esta dificultad, el Listado 17.15, `sctl.c`, utiliza la llamada `semctl` para eliminar un semáforo del sistema. Para hacerlo, querremos ejecutar `mksem` para crear un semáforo y después utilizar su identificador como el argumento de `sctl`. Construimos este programa ejecutando `make sctl`. Lo ejecutamos pasando el ID de semáforo devuelto por `mksem`.

LISTADO 17.15. SCTL.C

```
/*
 * sctl.c - Manipulación y eliminación de un semáforo.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int semid; /* Identificador del semáforo */

    if(argc != 2) {
        puts("USAGE: sctl <semáforo id>");
        exit(EXIT_FAILURE);
    }
    semid = atoi(argv[1]);

    /* Elimina el semáforo */
    if((semctl(semid, 0, IPC_RMID)) < 0) {
        perror("semctl IPC_RMID");
        exit(EXIT_FAILURE);
    } else {
        puts("semáforo eliminado");
        system("ipcs -s");
    }

    exit(EXIT_SUCCESS);
}
```

El siguiente extracto ilustra la operación de salida de sctl en mi sistema:

```
$ ./sctl 0  
semáforo eliminado
```

```
----- Matrices de semáforos -----  
key      semid      owner      perms      nsems      status
```

El código de sctl simplemente intenta eliminar el semáforo cuyo identificador pasamos en la línea de comandos. También utiliza la llamada system para ejecutar ipcs -s para confirmar que el semáforo en efecto se ha eliminado.

Resumen

En este capítulo hemos explicado los distintos métodos de comunicación interproceso disponibles en Linux: conductos y FIFO, memoria compartida, semáforos y colas de mensajes IPC del sistema V. Aunque más complicada que los conductos y FIFO, la memoria compartida IPC es más capaz y flexible y es utilizada normalmente en aplicaciones mayores y más sofisticadas, como en sistemas de administración de bases de datos relacionales (RDBMS) como Informix y Oracle. Las colas de mensajes permiten a las aplicaciones una considerable libertad a la hora de pasar mensajes entre procesos no relacionados. Para terminar, los semáforos son un método fácil de controlar el acceso a los diversos recursos del sistema.