TEMA 4

COMUNICACIÓN BÁSICA ENTRE PROCESOS EN UNIX/LINUX

1 Introducción

En la programación tradicional de un solo proceso, los diferentes módulos que lo componen se pueden comunicar entre sí mediante variables globales, llamadas a funciones, y paso de argumentos entre las funciones y sus invocadores. Al tratar con procesos separados esto ya no es así, porque cada proceso se ejecuta en su *propio espacio de direcciones* y entonces hay que considerar otros detalles.

En los SSOO con multiprogramación y tiempo compartido un objetivo básico de diseño es asegurar que los diferentes procesos coexistentes no interfieran entre sí. Para que dos procesos se comuniquen el uno con el otro, deben ponerse de acuerdo y el SO facilitar algunos mecanismos que faciliten dicha comunicación.

En el tema anterior vimos una forma muy “simple” de interconectar procesos: las señales. Decimos simple porque no se traspasa información de datos de un proceso a otro. Solamente son estímulos o eventos que se envían a un proceso para indicarle que haga algo.

En cambio, en este tema veremos algunos mecanismos de comunicación considerados como tales, ya que sí permiten traspasar información de unos procesos a otros.

Dichos mecanismos son: cauces o tuberías y recursos IPC (*Inter Process Communication*), entre los que destacan memoria compartida, colas de mensajes y semáforos.

En el presente tema veremos únicamente **tuberías** uno de los mecanismos IPC, la **memoria compartida.**

2 Entrada/salida de bajo nivel

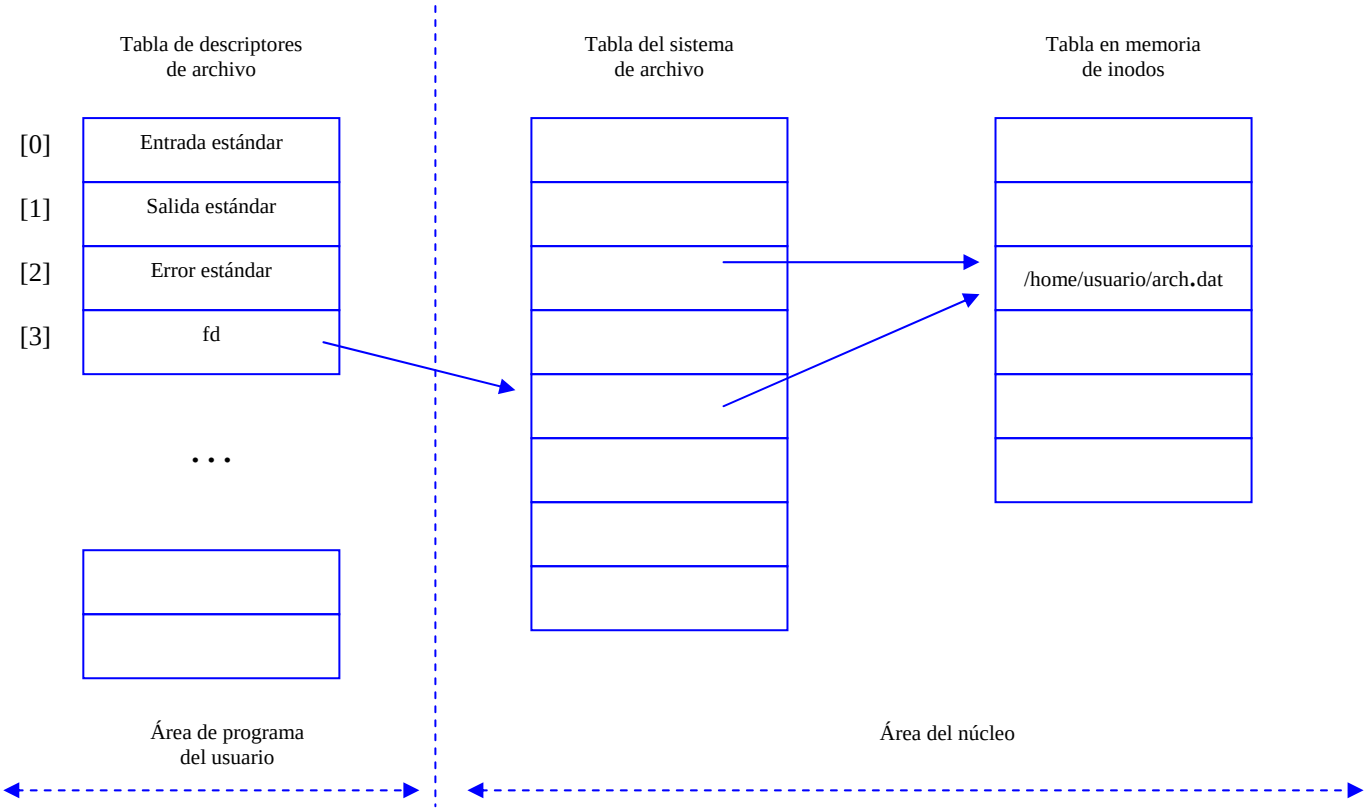
Antes de proseguir con la comunicación entre procesos vamos a ver una serie de funciones de entrada/salida estándar que van a ser comunes para el acceso a los archivos en Unix/Linux.

Dentro de los programas C, la biblioteca estándar de E/S de UNIX emplea ***descriptores*** de archivo, que son *identificadores* que permiten manejar la entrada y salida independientemente del tipo de dispositivo de E/S.

El sistema mantiene una tabla de descriptores de archivo que apunta a una entrada en la tabla de archivos del sistema. Esta tabla de descriptores se encuentra en el área de usuario del proceso, pero éste no tiene acceso a ella más que a través de llamadas al sistema utilizando un descriptor de archivo. El identificador hace las veces de índice a la tabla de descriptores del archivo.

En la figura siguiente se puede ver que existe un descriptor identificado por *fd* cuyo valor es 3 y que se corresponde con el archivo */home/usuario/arch.dat*.

Los descriptores número 0, 1 y 2 se corresponden, respectivamente con la entrada, salida y error estándares. Se identifican por **STDIN_FILENO**, **STDOUT_FILENO** y **STDERR_FILENO** y están definidos en *unistd.h*.



2.1 Apertura de ficheros

El prototipo de la función C para abrir ficheros es el siguiente:

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

int open(const char *nomfich, int flags, mode_t modo);
```

- *nomfich*: nombre completo del archivo, es decir, incluye la ruta completa o path.
- *flags*: especifica los permisos de acceso al fichero.
- *modo*: modos permitidos de acceso.

Si la llamada se ejecuta correctamente, devolverá un *descriptor* (número entero no negativo) asociado al archivo. Si falla, devolverá el valor –1.

2.2 Cierre de ficheros

Si con la apertura de un archivo se crea una entrada en la tabla de descriptores del archivo, con el cierre del archivo se elimina dicha entrada. El prototipo de la función C para cerrar ficheros es el siguiente:

```
#include <unistd.h>

int close(int fd);
```

- *fd*: descriptor del fichero obtenido de una llamada previa a *open*.

Si la llamada se ejecuta correctamente devuelve el valor 0. Si falla, devuelve el valor –1.

Ejemplo

```
fd = open("Arch.txt", O_WRONLY, S_IROTH);  /* Archivo de escritura y con permiso de lectura para el resto de usuarios */

close(fd);  /* Cierra el archivo "Arch.txt"  */
```

El estándar **POSIX.1** define unos valores para los modos de acceso a los archivos y sus nombres simbólicos definidos en *fcntl.h* son los siguientes:

Modos de acceso para función open

FLAG DE ACCESO	SIGNIFICADO
O_RDONLY	Apertura como sólo lectura
O_WRONLY	Apertura como sólo escritura
O_RDWR	Apertura para lectura y escritura

Indicadores de acceso (flags) para función open

FLAG DE ACCESO	SIGNIFICADO
O_CREAT	Si no existe el fichero se crea
O_APPEND	Se abre el fichero para añadir datos al final del mismo
O_TRUNC	Si existe el fichero, se elimina su contenido
O_SYNC	Fuerza a que cualquier actualización se grabe en disco inmediatamente
O_NONBLOCK	El fichero se abre en modo compartido

Estos modos pueden combinarse con una operación OR a nivel de bits (en lenguaje C el operador | permite esto).

Ejemplo: fich = open("Arch.txt", O_WRONLY | O_CREAT);

Máscaras para comprobación del modo de acceso a un archivo

CONSTANTE	VALOR OCTAL	SIGNIFICADO
S_IRUSR	00400	Permiso de lectura para el usuario propietario
S_IWUSR	00200	Permiso de escritura para el usuario propietario
S_IXUSR	00100	Permiso de ejecución para el usuario propietario
S_IRWXU	00700	Permiso de lectura, escritura y ejecución para propietario
S_IRGRP	00040	Permiso de lectura para el grupo de usuarios
S_IWGRP	00020	Permiso de escritura para el grupo de usuarios
S_IXGRP	00010	Permiso de ejecución para el grupo de usuarios
S_IRWXG	00070	Permiso de lectura, escritura y ejecución para el grupo
S_IROTH	00004	Permiso de lectura para los otros usuarios
S_IWOTH	00002	Permiso de escritura para los otros usuarios
S_IXOTH	00001	Permiso de ejecución para los otros usuarios
S_IRWXO	00007	Permiso de lectura, escritura y ejecución para los otros usuarios

2.3 Lectura y escritura de ficheros

El estándar **POSIX.1** define unas funciones C de llamadas al sistema para el acceso *secuencial* a archivos UNIX.

El prototipo de la función de lectura es el sisguiente:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

- *fd*: descriptor del fichero obtenido de una llamada previa a *open*.
- *buf*: zona de memoria donde se coloca la información leída.
- *nbytes*: número de bytes leídos.
    - 0 si se ha encontrado el final del archivo
    - –1 si se ha producido algún error
    - Número de bytes leídos en caso de éxito

El prototipo de la función de escritura es el siguiente:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);
```

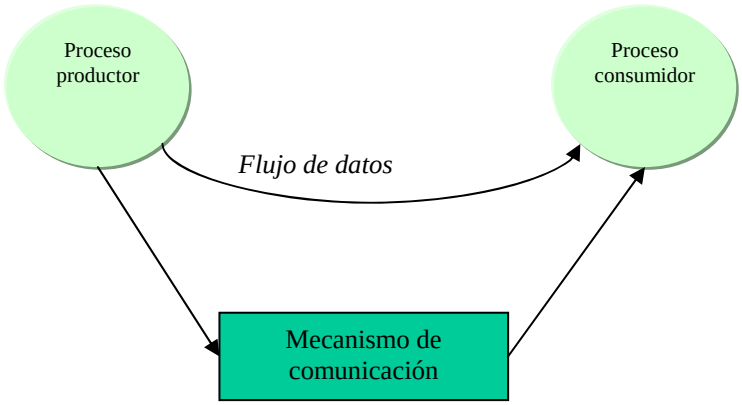
Si la llamada se ejecuta correctamente devuelve el número de bytes realmente escritos. Si falla, devuelve el valor –1.

Ejemplo:

```
/* fdread.c: Uso de funciones read y write, realizando una copia de un
archivo en otro */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

main (void)
{
   int fd, fdcop, nbytes;
   char arch[] = “Archivo“, arch_cop[]=“Archivo_copia”;
   int flags = O_CREAT | O_TRUNC | O_WRONLY;
   char buf[10];

   /* Apertura de archivo de sólo lectura */
   if ( (fd = open(arch, O_RDONLY) ) < 0 ) {
      perror("Abriendo archivo entrada");
      close(fd);
      exit(EXIT_FAILURE);
   }

      /* Apertura de archivo de escritura. Si el archivo no existe, lo
      crea  y si existe, trunca su contenido. Una vez creado, el modo de
      acceso es de lectura y escritura para el usuario y solamente
      lectura para el resto */
   if ( (fdcop = open(arch_cop, flags, 0644) ) < 0 ) {
      perror("Abriendo archivo copia de salida");
      close(fdcop);
      exit(EXIT_FAILURE);
   }

   /* Lee y escribe un máximo de 10 bytes en cada vez) */
   while ( (nbytes = read(fd, buf, 10) ) > 0)
      if ( write(fdcop, buf, nbytes) < 0 ) {
            perror("Escribiendo en salida");
            close(fdcop);
            exit(EXIT_FAILURE);
      }

   /* Cierre de ficheros */
   close(fd);
   close(fdcop);
}
```

3 Problema del productor-consumidor

El problema del **productor-consumidor** es uno de los problemas que surge cuando se programan aplicaciones utilizando procesos son **concurrentes**. En este tipo de problemas, uno o más procesos, que se denominan *productores*, generan cierto tipo de datos que son utilizados o consumidos por otros procesos denominados *consumidores*.



Estructura clásica de procesos productor-consumidor.

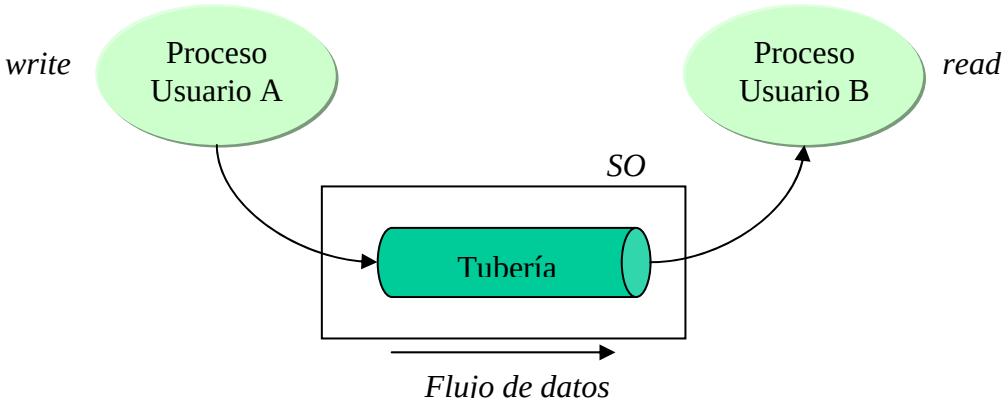
En esta clase de problemas es necesario disponer de algún tipo de mecanismo de comunicación que permita a los procesos productor y consumidor *intercambiar información*. Ambos procesos, además, deben ***sincronizar*** su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática: *cuando un mecanismo de comunicación se llene, el proceso productor se deberá quedar bloqueado hasta que haya espacio para seguir añadiendo elementos*. A su vez, el proceso consumidor deberá quedarse *bloqueado* cuando el mecanismo de comunicación esté vacío, ya que en este caso no podrá continuar su ejecución al no disponer de información a consumir.

La comunicación de procesos va a habilitar mecanismos para que los procesos puedan intercambiarse datos y sincronizarse. Hasta ahora hemos vistos dos formas elementales para que dos procesos puedan comunicarse: *el envío de señales para la sincronización y el uso de ficheros ordinarios para el intercambio de datos*. Las señales no deben considerarse parte de la forma habitual de comunicar dos procesos y su uso debe restringirse a la comunicación de eventos o situaciones excepcionales.

A continuación veremos un mecanismo más eficiente para resolver el problema del productor-consumidor, denominado **tubería** o **pipe**.

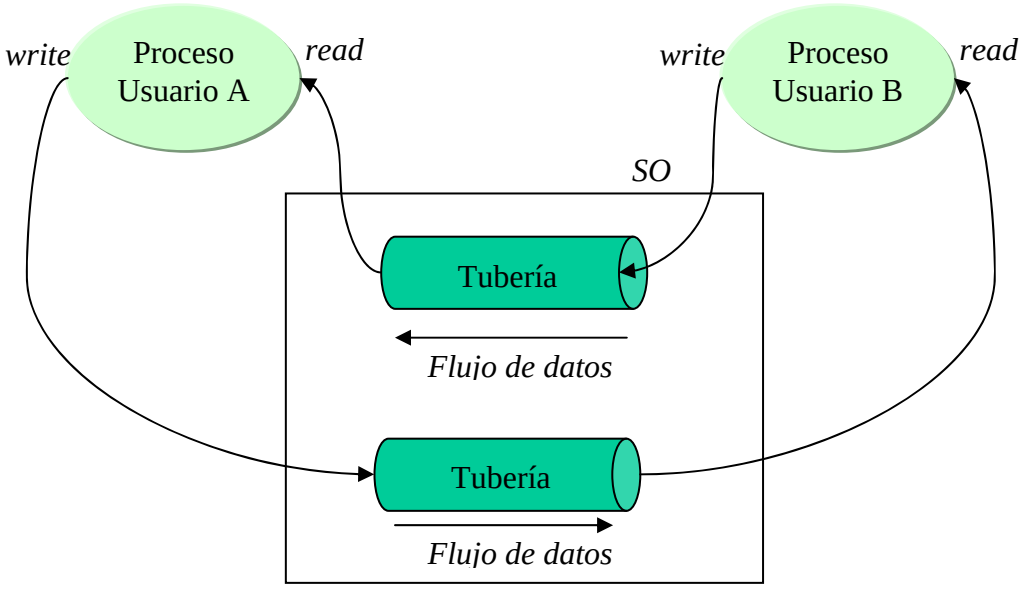
4 Tuberías o Pipes

Una ***tubería*** se puede considerar como un *canal de comunicación y sincronización entre dos procesos.* Las hay de dos tipos: *tuberías con nombre (fifos)* y *tuberías sin nombre*. La comunicación se realiza en un solo sentido, es decir, es **half-duplex**. Cada proceso ve la tubería como un conducto con dos extremos, uno para escribir datos en él y otro para leerlo de él. Además, su característica FIFO hace los datos se lean en el mismo orden en que son escritos.



Comunicación unidireccional empleando una tubería.

Si se quiere que un proceso A pueda simultáneamente enviar y recibir información de otro B, se debe recurrir a crear dos tuberías, una para enviar información desde A hasta B y otra para enviar desde B hasta A.



Comunicación biridireccional empleando una tubería.

4.1 Tuberías sin nombre o anómimas

Una tubería en C se puede crear utilizando la función *pipe*. A este tipo de tuberías se las denomina **pipes sin nombre o anónimas**.

El prototipo de función C para crear una tubería sin nombre es el siguiente:

```
#include <unistd.h>

int pipe(int *tuberia);
```

Donde *tubería* es un array que almacena dos ***descriptores*** de fichero, *tubería[0]* contiene el descriptor de *lectura* y *tubería[1]* el de *escritura*.

La función devuelve 0 en caso de éxito y –1 en caso de error.

Al ser la tubería un tipo de fichero, vamos a poder acceder a ella con las primitivas de gestión de archivos proporcionadas por el SO: cescribir y leer de ella con las llamadas *write* y *read*.

Los ficheros asociados con estos descriptores son streams abiertos tanto para lectura como para escritura. Una lectura en *tuberia[0]* accede a los datos escritos en *tuberia[1]* y una lectura en *tuberia[1]* accede a los datos escritos en *tuberia[0].* Lo normal es que *tuberia[0]* se emplee para lectura y *tuberia[1]* para escritura.

Existen otras funciones para el acceso a un pipe que son similares a las funciones de E/S vistas anteriormente: write, read y close.

Así mismo, existen dos llamadas al sistema que, aunque no son específicas de *pipes*, suelen usarse en combinación con estos para lograr las redirecciones de ficheros. Son las funciones *dup* y *dup2* y que veremos en un apartado posterior.

Sin embargo, es preciso hacer algunas consideraciones en cuanto a las operaciones de lectura y escritura sobre una tubería:

- Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.
- Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error.
- Una operación de escritura sobre una tubería se realiza de forma ***atómica***, es decir, si dos procesos intentan escribir de forma simultánea en una tubería, sólo uno de ellos lo hará. El otro se bloqueará hasta que finalice la primera escritura.
- Si la tubería está vacía, la llamada bloquea el proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.
- Si la tubería almacena M bytes y se quieren leer n bytes, entonces:

- Si M ≥ n, la llamada devuelve n bytes y elimina de la tubería los datos solicitados.

- Si M < n, la llamada devuelve M bytes y elimina los datos disponibles de la tubería.

- Si no hay escritores y la tubería está vacía, la operación de lectura devuelve fin de archivo y no se bloquea el proceso lector.

Como puede apreciarse, una lectura de una tubería nunca bloquea al proceso si hay datos disponibles en la misma.

Los descriptores de fichero se heredan de padres a hijos tras la llamada a *fork* o a *exec*, por lo que la misma tubería puede ser utilizada por el padre y sus procesos hijo.

La *sincronización* entre los accesos de escritura y lectura la lleva a cabo el *núcleo* del Sistema Operativo. Así pues, hasta que no haya datos en la tubería, el proceso lector no tomará el control y permanecerá bloqueado. Además, el *núcleo* se encarga de gestionar la tubería para dotarla de un mecanismo de acceso tipo ***FIFO*** (*First-In First-Out*). En consecuencia, *el proceso receptor saca los datos en el mismo orden en que los escriba el proceso emisor*. Si se realiza una operación con *write* en una tubería que ha sido cerrada, se genera la señal ***SIGFPE***, *write* devuelve el valor 0 y la variable *errno* toma el valor ***EPIPE***.

El tamaño del bloque datos que se puede escribir en una tubería depende del sistema, pero no es inferior a 4 Mb.

El siguiente ejemplo C ilustra el envío de mensajes entre un proceso emisor y otro receptor a través de una tubería sin nombre.

Ejemplo:

```
#include <stdio.h>
#include <stdilb.h>
#include <string.h>
#include <unistd.h>

#define MAX 128

int main()
{
   int pid, tuberia[2];
   char mensaje[MAX];

  /* Creación de una tubería sin nombre. */
   if ( pipe(tuberia) == -1 )
   {
      perror("pipe");
      exit(-1);
   }
  /* Creación del proceso hijo. */
   if ( (pid = fork()) == -1 )
   {
      perror("fork");
      exit(-1);
   }
   if ( pid == 0 )    /* Código del proceso hijo. */
   {
    /* El proceso hijo (receptor) se va a encargar de leer un mensaje
       de la tubería y presentarlo en pantalla. */

      close(tuberia[1]);
      while ( read(tuberia[0], mensaje, MAX) > 0  &&
              strcmp(mensaje, "Q") != 0 )
      {
         fprintf(stdout,"Proceso Receptor. Mensaje: %s\n", mensaje);
         memset(mensaje, 0, MAX); // Rellena todo el mensaje con nulos
      }

      close(tuberia[0]);
      exit(0);
   }
   else      /* Código del proceso padre. */
   {
      /* El proceso padre (emisor) se va a encargar de leer un mensaje
      de la entrada estándar y lo escribe en la tubería para que lo
      reciba el proceso hijo. */

      close(tuberia[0]);
      do {
          scanf("%s%", mensaje);
          write(tuberia[1], mensaje, strlen(mensaje));
      } while ( strcmp(mensaje, "Q") != 0) );

      close(tuberia[1]);
      exit(0);
   }

}
```

4.2 Tuberías con nombre o FIFO

Por medio de *tuberías sin nombre* podemos comunicar procesos padre y sus hijos, ya que el proceso que crea la tubería y sus hijos tienen acceso a la misma. En cambio, para los procesos que no guardan ninguna relación de parentesco no sirven los canales abiertos mediante tuberías sin nombre. Para comunicar este tipo de procesos recurrimos a las ***tuberías con nombre*** o **fifo**.

El funcionamiento de un *fifo* es similar a una tubería sin nombre pero accediendo a ella como si fuera un archivo en disco. Ello significa que lo primero que hay que hacer es abrirlo con una instrucción *open*. Cuando un proceso abre un *fifo* para escribir en él, se queda a la espera hasta que otro proceso lo abre para leer de él. Si es el proceso lector el primero en abrir el *fifo*, se pone a la espera hasta que un proceso lo abre para escribir.

Este tipo de *pipes* se pueden crear tanto desde la línea de órdenes del sistema operativo como a través de una función C.

Desde la línea de órdenes

 # mknod [opciones] nombre_pipe p

ó

 # mkfifo [opciones] nombre_pipe

Ejemplos:

 # mknod –m 644 tub1 p (crea una tubería denominada *tub1* con permisos de lectura y escritura para el propietario y de sólo lectura para el resto)

 # mkfifo 444 tub2 (crea una tubería denominada *tub2* con permisos de sólo lectura)

Con una función C

```
int mknod(char *nombre, mode_t modo, dev_t dev);
```

La función devuelve 0 en caso de éxito y –1 en caso de error.

Ejemplo: *mknod(“tub”, S_IFIFO | 0666, 0);* /* Crea una tubería de nombre *tub* y permisos 0666 */

Una vez creada la tubería, se puede leer o escribir en ella como en cualquier fichero a través de las funciones *read* y *write*. Además, persisten en el sistema aún después de que todos los procesos los hayan cerrado, pues son un tipo de archivo más.

El siguiente ejemplo muestra el uso de tuberías con nombre. Ahora se tienen dos procesos: *wrfifo* que escribe mensajes en la tubería y *rdfifo* que los lee de la misma.

Ejemplo:

```
/***
    PROGRAMA: wrfifo.c
    DESCRIPCION:
       Escritura en un FIFO.
    FORMA DE USO:
       wrfifo (Necesita la ejecución del programa rdfifo)
***/

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <limits.h>  // Para PIPE_BUF = 4 Mb
#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 albergar la hora actual. */
   int i;

   printf("Soy el proceso %d\n", getpid());

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

   /* Genera algunos datos para escribir en el FIFO */
   for (i=0; i<5; i++)
   {
      time(&tp);  // Obtiene la hora actual
      len = sprintf(buf, "wrfifo: %d envía %s", getpid(), ctime(&tp));

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

  close(fd);
  exit(EXIT_SUCCESS);
}
```

```
/***
    PROGRAMA: rdfifo.c
    DESCRIPCION:
       Lectura desde un FIFO.
    FORMA DE USO:
       rdfifo (Necesita la ejecución del programa wrfifo)
***/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <limits.h>  // Para PIPE_BUF = 4Mb


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


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

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

   close(fd);

   exit(EXIT_SUCCESS);
}
```

Nota: Para probar este último ejemplo, previamente ha de crearse la tubería con nombre. Por ejemplo, desde la línea de comandos con *mkfifo 644 fifo1*.

Además, se deben ejecutar ambos programas al mismo tiempo desde diferentes ventanas shell.

5 Mecanismos IPC

La comunicación entre procesos es una función básica de los sistemas operativos. Ya hemos visto unos mecanismos de comunicación de procesos como son las tuberías. En este apartado desarrollaremos uno de los tres mecanismos denominados **IPC** (*InterProcess Communication*) existentes:

- Memoria Compartida
- Semáforos
- Colas de Mensajes.

Estos tres mecanismos tienen una estructura similar. En todos ellos cada objeto IPC se identifica por un *número entero no negativo* de forma parecida a los descriptores de los ficheros. Para obtener este entero se hace uso de una determinada clave y una determinada función: *shmget*, *semget* y *msgget* para memoria compartida, semáforos y colas de mensajes, respectivamente.

Para extraer el entero, dicha clave puede ser:

- Elegida por el núcleo del sistema: IPC_PRIVATE
- Elegida por el usuario
- Generada por el sistema a través de la función ***ftok*** a la que se le pasan unos parámetros elegidos por el usuario o programador. Este caso permite la comunicación entre procesos independientes (no emparentados).

El prototipo de la función es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t  ftok(char *camino, int id);
```

Donde *camino* es un puntero al nombre de un path de un fichero e *id* es un carácter que identifica el proyecto. Devuelve la llave **IPC** si tiene éxito o –1 en caso de error. Esta llave es única para cada par de valores (camino, id).

Pueden consultarse los objetos IPC existentes en nuestro sistema a través de la orden **ipcs**.

ipcs <opcion>

*donde **opcion** puede tomar los valores: -m (memoria compartida) , -s (semáforo) o -q (cola de mensajes).*

En el caso de que no se introduzca opcion mostrará los tres tipos de mecanismos IPC existentes en el sistema.

 # ipcs

```
------ Segmentos memoria compartida --------
key        shmid      propietario   perms      bytes      nattch     estado
0x4b06a7bb 22937      jcarlos       600        40         0

------ Matrices semáforo --------
key        semid      propietario   perms      nsems
0x4c01df1c 32766      jcarlos       600        2
0x00000000 35431      root          600        1

------ Colas de mensajes --------
key        msqid      propietario   perms      bytes utilizados   mensajes
0x0207f613 32769      root          600        60                 6
```

Aquí puede verse que existe un segmento de memoria compartida con identificación 22937 y permisos de lectura y escritura para el propietario y que contiene 40 bytes.
Asimismo puede verse que existe una cola de mensajes con identificación 32769 y permisos de lectura y escritura para el propietario y que contiene 6 mensajes no leídos.
También existen dos grupos de semáforos con los mismos permisos de lectura y escritura. Uno con dos semáforos creado a partir de una llamada a ftok() (key distinta de 0) y el otro con un semáforo creado por el núcleo del sistema operativo (key igual a 0).

Puede eliminarse cualquier objeto IPC mediante el comando **ipcrm**.

ipcrm tipo_ipc identificador

*donde **tipo_ipc** puede tomar los valores: shm (memoria compartida), sem (semáforo) o msq (cola de mensajes)*

***identificador** es el identificador del objeto IPC*

 # ipcrm msq 32769 (eliminaría la cola de mensajes con msqid = 32769)

6 Memoria Compartida

Cuando un proceso crea otro proceso hijo, éste **duplica** su contexto. Esto implica que los cambios que un proceso realiza sobre sus variables no son visibles en el otro proceso, dado que son copias distintas. Si bien en multitud de ocasiones esto es precisamente lo que se quiere, en otras muchas sería deseable que tanto el padre como el hijo pudieran tener *variables en común* para compartir información. Existen mecanismos vistos previamente como las tuberías que permiten compartir información pero están limitadas a una comunicación 1 a 1. Una manera de comunicar procesos sin la limitación 1 a 1 es a través de una *zona común de memoria*.

Este mecanismo se denomina ***memoria compartida***. Para enviar datos de un proceso a otro, sólo hay que escribir en memoria y automáticamente esos datos están disponibles para que los lea cualquier otro proceso. Además, tiene la particularidad de que es *la forma más rápida de comunicar procesos*

El sistema operativo *Unix* (y, por extensión, Linux) permite crear zonas de memoria que pueden ser direccionadas por varios procesos simultáneamente. Esta memoria va a ser ***virtual***, por lo que los procesos emplearán direcciones virtuales y será el núcleo del sistema operativo el encargado de traducir las unas en las otras.

Las llamadas para poder manipular la memoria compartida son:

- **shmget**, para crear una zona de memoria compartida o habilitar el acceso a una ya creada.

- **shmat**, para unir o acoplar una zona de memoria compartida a un proceso, es decir, *pega* el segmento de memoria identificada por *shmid* al segmento de datos del proceso que llama a la función.

- **shmdt**, para separar o desacoplar una zona previamente unida.

- **shmctl**, para acceder y modificar la información administrativa y de control que el kernel le asocia a cada zona de memoria compartida (establecer propietario, permisos, destruir el propio segmento, etc.).

Nota: *después de una llamada a* ***fork(),*** *el hijo* ***hereda*** *los segmentos de memoria compartidos acoplados. Después de una llamada a* ***exit()*** *o a cualquier función de la familia* ***exec()****, todos los segmentos son* ***desacoplados*** *pero* ***no destruidos****.*

6.1 Acceso al segmento de memoria compartida

El prototipo de la función para poder crear y/o acceder a un segmento de memoria compartida es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t  key, int size, int shmflg);
```

Donde:

- ***key*** puede ser uno de los siguientes:
  - clave definida por el programador
  - IPC_PRIVATE para que sea el propio sistema operativo el que se encargue de obtener el identificador
  - valor obtenido por una llamada previa a la función *ftok*.

- ***size*** es el tamaño en bytes del segmento a crear.

- ***shmflg*** es una máscara de bits que indica el modo de adquisición del identificador.

  Si *key* no es IPC_PRIVATE, entonces hay que añadir en *shmflg* el bit IPC_CREAT para poder crear el segmento.

Para detectar si el segmento de memoria ya existe entonces también hay que añadir en *shmflg* el bit IPC_EXCL. En este caso, la variable de entorno *errno* pasa a valer EEXIST.

Si la llamada se ejecuta correctamente, devolverá el identificador (número entero no negativo) asociado a la zona de memoria. Si falla, devolverá el valor –1 y la variable de entorno *errno* tendrá el valor del error producido.

Por ejemplo:

```
int   shmid1, shmid2;

shmid1 = shmget(IPC_PRIVATE, 10*sizeof(int), IPC_CREAT | 0600);

shmid2 = shmget(ftok("/bin/ls", 'K'), 5*sizeof(float), IPC_CREAT | 0644);
```

El primer caso reservaría una zona de memoria compartida para albergar 10 números enteros. Es el propio núcleo del sistema operativo el que se encarga de obtener el identificador (IPC_PRIVATE en primer argumento). A este segmento únicamente pueden acceder en modo lectura y escritura los procesos del usuario propietario y no podrían acceder ningún proceso de cualquier otro usuario del sistema.
En el segundo caso se reservaría una zona de memoria para albergar 5 números del tipo float. Ahora el identificador se obtiene mediante una llave generada por una llamada a la función ftok(). El flag IPC_CREAT indica que si no existe el segmento entonces se creará. Cualquier otro proceso que obtenga el identificador con la misma clave accederá al mismo segmento de memoria compartida.

6.2 Acoplamiento y desacoplamiento de un segmento de memoria compartida

Para poder utilizar una zona de memoria compartida, es preciso *mapearla* al espacio de direcciones virtuales del proceso (recordar que los procesos trabajan con direcciones virtuales que no coincide con las direcciones físicas de la memoria). Este hecho se denomina habitualmente como unirse o **acoplarse** *al segmento de memoria*.

Cuando el proceso ya no necesita el segmento de memoria, se realiza el proceso inverso que se denomina **desacoplarse** del segmento de memoria y éste deja de estar accesible al proceso.

Los prototipos de las funciones C que realizan estos dos cometidos son los siguientes:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void  *shmat(int shmid, char *shmaddr, int shmflg);

int shmdt(char *shmaddr);
```

Donde:

- ***shmid*** es un identificador válido devuelto por una llamada previa a *shmget*.
- ***shmaddr*** es la dirección virtual donde queremos que empiece la zona de memoria compartida. Lo normal es que *shmaddr* valga 0 con lo cual se deja en manos del núcleo la elección de la dirección de inicio.
- ***shmflg*** es una máscara de bits que indica el modo de adquisición del identificador. Por defecto, el segmento está accesible en modo lectura/escritura. Si el bit SHM_RDONLY está activo, la memoria será accesible para leer, pero no para escribir.

Si las llamadas funcionan correctamente, *shmat* devuelve la dirección a la que está unido el segmento de memoria compartida y *shmdt* devuelve 0. Si algo falla, ambas llamadas devuelven –1.

Siguiendo con el ejemplo anterior:

```
int  * buffer;

buffer = (int *) shmat(shmid1, NULL, 0);
```

6.3 Control del segmento de memoria compartida

Ciertas operaciones sobre el segmento de memoria compartida tales como obtener información administrativa y de estado o simplemente eliminar el segmento, se realizan con la función ***shmctl*** cuyo prototipo es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Donde:

- ***shmid*** es un identificador válido devuelto por una llamada previa a *shmget*.
- ***cmd*** indica el tipo de operación de control a realizar. Algunos valores posibles son (consultar el manual para más opciones):
  - IPC_STAT: lee el estado de la estructura de control de la memoria y lo devuelve a través de la zona de memoria apuntada por *buf*.
  - IPC_RMID: borra del sistema la zona de memoria compartida identificada por *shmid*.

- ***shmid_ds*** es una estructura con datos del segmento tales como PID del proceso creador, PID del proceso que hizo la última operación sobre el segmento, número de procesos unidos al segmento, fechas de acoplamiento y desacoplamiento, etc.
  Lo habitual es llamar a la función con este valor puesto a 0.

Si algo falla, la función devuelve el valor –1.

Tras realizar con éxito el acoplamiento al segmento de memoria compartida, el acceso a la misma por parte del proceso se realiza a través de punteros, como cualquier otra memoria de datos asignada al programa.

Ejemplo:

```
/*  shm1.c :  creación y escritura en un segmento de memoria */

#define MAX 10

int main()
{
      int shmid, i;
      int *array;
      key_t llave;

      llave = ftok(".", 'C');   // Creación de la llave

      /* Petición de una zona de memoria compartida para albergar hasta
MAX números enteros. Si no existe, se crea */
      shmid = shmget(llave, MAX * sizeof(int), IPC_CREAT | 0600);

      /* Unión de la zona de memoria compartida al espacio de direcciones
virtuales del proceso */
      array = (int *)shmat(shmid, 0, 0);

      /* Manipulación de la zona de memoria compartida */
      for (i=0; i < MAX; i++)
          array[i] = 2 * i;

      /* Separación de la zona de memoria compartida de nuestro espacio
de direcciones virtuales. Esto NO es un borrado del segmento */
      shmdt(array);

}
```

Ejemplo:

```
/*  shm2.c :  acceso a un segmento de memoria compartida existente */

#define MAX 10

int main()
{
      int shmid, i;
      int *array;
      key_t llave;

      /* Creación de la llave. Debería ser la misma que en shm1.c */
      llave = ftok(".", 'C');

      /* Petición de la zona de memoria compartida */
      shmid = shmget(llave, MAX * sizeof(int), 0600);
      if ( shmid == -1 )
      {
          perror("Acceso segmento");
          exit(1);
      }

      /* Unión de la zona de memoria compartida al espacio de direcciones
virtuales del proceso */
      array = (int *)shmat(shmid, 0, 0);

      /* Comprobación de los datos leídos del segmento */
      for (i=0; i < MAX; i++)
          fprintf(stdout, "%2d ", array[i]);

      /* Separación de la zona de memoria compartida del espacio de
direcciones virtuales del proceso. El espacio puede ser aún accesible
desde otro proceso que la tenga acoplada */
      shmdt(array);

      /* Borrado de la zona de memoria compartida. A partir de ahora ya
no será accesible por ningún proceso */
      shmctl(shmid, IPC_RMID, 0);

}
```