

SESIÓN 4 MODULO II SO

Ejercicio 1. Consulte en el manual las llamadas al sistema para la creación de archivos especiales en general (mknod) y la específica para archivos FIFO (mkfifo). Pruebe a ejecutar el siguiente código correspondiente a dos programas que modelan el problema del productor/consumidor, los cuales utilizan como mecanismo de comunicación un cauce FIFO. Determine en qué orden y manera se han de ejecutar los dos programas para su correcto funcionamiento y cómo queda reflejado en el sistema que estamos utilizando un cauce FIFO. Justifique la respuesta.

Los programas dados en el enunciado son los siguientes:

Programa consumidor

```
1  //consumidorFIFO.c
2  //Consumidor que usa mecanismo de comunicacion FIFO
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <errno.h>
10 #include <string.h>
11 #define ARCHIVO_FIFO "ComunicacionFIFO"
12
13 int main(void){
14     int fd;
15     char buffer[80]; // Almacenamiento del mensaje del cliente
16     int leidos;
17
18     //Crear el cauce con nombre (FIFO) si no existe
19     umask(0);
20     mknod(ARCHIVO_FIFO, S_IFIFO|0666, 0);
21     //también vale: mkfifo(ARCHIVO_FIFO, 0666);
22
23     //Abrir el cauce para lectura-escritura
24     if ( (fd=open(ARCHIVO_FIFO, O_RDWR)) < 0 ) {
25         perror("open");
26         exit(-1);
27     }
28
29     //Aceptar datos a consumir hasta que se envíe la cadena fin
30     while(1) {
31         leidos=read(fd, buffer, 80);
32         if(strcmp(buffer, "fin")==0){
33             close(fd);
34             return 0;
35         }
36         printf("\nMensaje recibido: %s\n", buffer);
37     }
38
39     return 0;
40 }
```

Programa productor

```
1 //productorFIFO.c
2 //Productor que usa mecanismo de comunicacion FIFO
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <unistd.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <errno.h>
11 #define ARCHIVO_FIFO "ComunicacionFIFO"
12
13 int main(int argc, char *argv[]){
14     int fd;
15
16     //Comprobar el uso correcto del programa
17     if(argc != 2) {
18         printf("\nproductorFIFO: faltan argumentos (mensaje)");
19         printf("\nPruebe: productorFIFO <mensaje>, donde <mensaje> es una cadena de
20         exit(-1);
21     }
22
23     //Intentar abrir para escritura el cauce FIFO
24     if( (fd=open(ARCHIVO_FIFO,O_WRONLY)) <0){
25         perror("\nError en open");
26         exit(-1);
27     }
28
29     //Escribir en el cauce FIFO el mensaje introducido como argumento
30     if( (write(fd,argv[1],strlen(argv[1])+1)) != strlen(argv[1])+1) {
31         perror("\nError al escribir en el FIFO");
32         exit(-1);
33     }
34     close(fd);
35     return 0;
36 }
37
38
```

Programa productor:

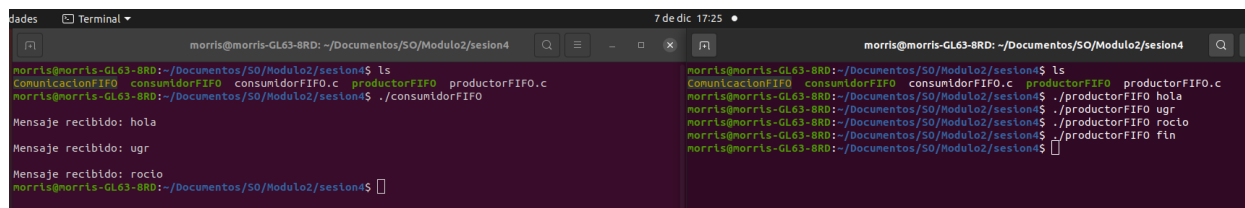
- Si el número de argumentos no es 2, o sea si no se le pasa ningún string, faltarían argumentos. El primer argumento es el ejecutable y el segundo argumento es el string que se le quiere pasar al consumidor.
- Abrimos la escritura en el cauce FIFO con fd, de modo que solo pueda escribirse, y como siempre comunicamos su error.
- Escribimos en fd, en el cauce FIFO, argv[1] que sería el segundo argumento que se le pasa, o sea el string que queremos pasar al consumidor, y si el String que le pasamos es menor que el número de bytes que se van a escribir, o sea si es menor que el size del string +1, hay un error.

Programa consumidor:

- creamos el cauce con nombre llamado ARCHIVO_FIFO, que se puede hacer de 2 formas, y antes de eso ponemos la máscara a 0 para que no haya ningún tipo de problema.

- abrimos el cauce en modo lectura-escritura.
- mientras sea 1, con la variable leídos hacemos un read de fd (archivo fifo abierto) y copiamos 80 bytes en el buffer. Si el buffer no contiene la palabra “fin”, aparecerá por pantalla mensaje leído y a continuación el string que le hemos pasado a través del productor.

Para que esto funcione, primero se debe ejecutar el programa consumidor, para esperar la llegada del string que se le pasa a través del cauce del archivo productor. De esta forma, hasta que no se escriba la palabra fin, esto se producirá continuamente, como se puede ver a continuación:



```
morris@morris-GL63-BRD: ~/Documentos/50/Modulo2/sesion4
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ls
ComunicacionFIFO  consumidorFIFO  consumidorFIFO.c  productorFIFO  productorFIFO.c
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ./consumidorFIFO
Mensaje recibido: hola
Mensaje recibido: ugr
Mensaje recibido: rocto
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$

morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ls
ComunicacionFIFO  consumidorFIFO  consumidorFIFO.c  productorFIFO  productorFIFO.c
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ./productorFIFO hola
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ./productorFIFO ugr
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ./productorFIFO rocto
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$ ./productorFIFO fin
morris@morris-GL63-BRD:~/Documentos/50/Modulo2/sesion4$
```

Ejercicio 2. Consulte en el manual en línea la llamada al sistema pipe para la creación de cauces sin nombre. Pruebe a ejecutar el siguiente programa que utiliza un cauce sin nombre y describa la función que realiza. Justifique la respuesta.

```
1  /*
2  tarea6.c
3  Trabajo con llamadas al sistema del Subsistema de Procesos y Caudes conforme a
4  POSIX 2.10
5  */
6  #include <sys/types.h>
7  #include <fcntl.h>
8  #include <unistd.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <errno.h>
12 #include <string.h>
13
14 int main(int argc, char *argv[]){
15
16     int fd[2], numBytes;
17     pid_t PID;
18     char mensaje[] = "\nEl primer mensaje transmitido por un cauce!!\n";
19     char buffer[80];
20     pipe(fd); // Llamada al sistema para crear un cauce sin nombre
21
22     if ((PID= fork()) < 0){
23         perror("fork");
24         exit(-1);
25     }
26
27     if (PID == 0) {
28         //Cierre del descriptor de lectura en el proceso hijo
29         close(fd[0]);
30         // Enviar el mensaje a través del cauce usando el descriptor de escritura
31         write(fd[1], mensaje, strlen(mensaje)+1);
32         exit(0);
33     }
34     else { // Estoy en el proceso padre porque PID != 0
35         //Cerrar el descriptor de escritura en el proceso padre
36         close(fd[1]);
37         //Leer datos desde el cauce.
38         numBytes = read(fd[0], buffer, sizeof(buffer));
39         printf("\nEl número de bytes recibidos es: %d", numBytes);
40         printf("\nLa cadena enviada a través del cauce es: %s", buffer);
41     }
42
43     return(0);
44 }
```

Lo que hace el programa es lo siguiente:

- con la orden pipe, crea un cauce sin nombre, sobre fd,
- Se hace una comprobación del fork() para ver si es correcto o no.
- Si PID==0, significa que estamos en el proceso hijo, entonces como el hijo quiere enviar datos al padre (ya que usa write), hacemos close fd[0] (que es el descriptor de lectura, ya que ahora estamos con el de escritura). Enviamos el mensaje a través del cauce creado con write, y lo almacenamos en fd[1]

- En otro caso, estamos en el proceso padre, ya que el pid es distinto de 0. Cerramos el descriptor de escritura, fd[1], ya que estamos en proceso de leer con fd[0]. Con read lee el mensaje de fd[0] y lo almacena en el buffer usando el size de ese buffer.
- Por último, con la función read imprime por pantalla el número de bytes recibidos y muestra la cadena que se ha guardado en fd[0], o sea, el mensaje que se ha enviado antes, y se almacena en buffer con la función read.

Al ejecutarlo se muestra lo siguiente:

```
El primer mensaje transmitido por un cauce!!
morris@morris-GL63-8RD:~/Documentos/S0/Modulo2/sesion4$ ./tarea6

El número de bytes recibidos es: 47
La cadena enviada a través del cauce es:
El primer mensaie transmitido por un cauce!!
```

Ejercicio 3. Redirigiendo las entradas y salidas estándares de los procesos a los cauces podemos escribir un programa en lenguaje C que permita comunicar órdenes existentes sin necesidad de reprogramarlas, tal como hace el shell (por ejemplo `ls | sort`). En particular, ejecute el siguiente programa que ilustra la comunicación entre proceso padre e hijo a través de un cauce sin nombre redirigiendo la entrada estándar y la salida estándar del padre y el hijo respectivamente.

```
10 #include<stdlib.h>
11 #include<errno.h>
12
13 int main(int argc, char *argv[]){
14
15     int fd[2];
16     pid_t PID;
17     pipe(fd); // Llamada al sistema para crear un pipe
18
19     if ((PID= fork()) < 0){
20         perror("fork");
21         exit(-1);
22     }
23
24     if(PID == 0){ // ls
25
26         //Establecer la dirección del flujo de datos en el cauce cerrando
27         // el descriptor de lectura de cauce en el proceso hijo
28         close(fd[0]);
29
30         //Redirigir la salida estándar para enviar datos al cauce
31         //-----
32         //Cerrar la salida estándar del proceso hijo
33         close(STDOUT_FILENO);
34
35         //Duplicar el descriptor de escritura en cauce en el descriptor
36         //correspondiente a la salida estándar (stdout)
37         dup(fd[1]);
38         execlp("ls","ls",NULL);
39     }
40     else{ // sort. Estoy en el proceso padre porque PID != 0
41
42         //Establecer la dirección del flujo de datos en el cauce cerrando
43         // el descriptor de escritura en el cauce del proceso padre.
44         close(fd[1]);
45
46         //Redirigir la entrada estándar para tomar los datos del cauce.
47         //Cerrar la entrada estándar del proceso padre
48         close(STDIN_FILENO);
49
50         //Duplicar el descriptor de lectura de cauce en el descriptor
51         //correspondiente a la entrada estándar (stdin)
52         dup(fd[0]);
53         execlp("sort","sort",NULL);
54     }
55
56     return(0);
57 }
```

En este programa se hace lo siguiente:

- En primer lugar se crea el cauce con `pipe`.
- Se comprueba el `fork`.

- Si el `pid == 0`, estamos en el proceso hijo, por lo que se cierra el descriptor de lectura `fd[0]`, para redirigir la salida estándar (es la escritura por así decirlo) para enviar datos al cauce, cerramos la salida estándar (`STDOUT_FILENO`), que corresponde al descriptor de archivo 1 y posteriormente hacemos una copia del descriptor de escritura. Por tanto, ahora `fd[1]` tendrá el mismo valor que `STDOUT_FILENO`. (creo). Con `execlp` se ejecuta la orden `ls`.
- Si no es 0, estamos en el proceso padre, por lo que se cierra el descriptor de escritura `fd[1]`, redirigimos la entrada estándar, que sería la lectura por así decirlo, de tal forma que, se cierra la entrada estándar (`STDIN_FILENO`), que corresponde al descriptor de archivo 0, y a continuación duplicamos el descriptor de lectura, que tendrá el mismo valor que `STDIN_FILENO`. Con `execlp` ejecutamos la orden `sort`.

Con esto ya tendríamos el cauce formado con padre e hijo, que realiza la orden `ls | sort`.

Como podemos ver, esta tarea realiza la misma función que la orden `ls | sort` → lista los ficheros del directorio actual y los ordena por orden alfabético.

```

morris@morris-6L63-8RD:~/Documentos/S0/Modulo2/sesion4$ ./tarea7
ComunicacionFIFO
consumidorFIFO
consumidorFIFO.c
productorFIFO
productorFIFO.c
tarea6
tarea6.c
tarea7
tarea7.c
morris@morris-6L63-8RD:~/Documentos/S0/Modulo2/sesion4$ ls | sort
ComunicacionFIFO
consumidorFIFO
consumidorFIFO.c
productorFIFO
productorFIFO.c
tarea6
tarea6.c
tarea7
tarea7.c

```

Ejercicio 4. Compare el siguiente programa con el anterior y ejecútelo. Describa la principal diferencia, si existe, tanto en su código como en el resultado de la ejecución.

```
1 // Programa ilustrativo del uso de pipes y la redireccion de entrada y
2 salida estándar: "ls | sort", utilizando la llamada dup2.
3 */
4
5 #include <sys/types.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <errno.h>
11
12
13 int main(int argc, char *argv[]){
14     int fd[2];
15     pid_t PID;
16     pipe(fd); // Llamada al sistema para crear un pipe
17
18     if ((PID= fork()) < 0){
19         perror("\nError en fork");
20         exit(-1);
21     }
22
23     if (PID == 0){ // ls
24
25         //Cerrar el descriptor de lectura de cauce en el proceso hijo
26         close(fd[0]);
27
28         //Duplicar el descriptor de escritura en cauce en el descriptor
29         //correspondiente a la salida estándar (stdout), cerrado previamente en
30         //la misma operación
31         dup2(fd[1], STDOUT_FILENO);
32
33         execlp("ls", "ls", NULL);
34     }
35
36     else{ // sort. Proceso padre porque PID != 0.
37
38         //Cerrar el descriptor de escritura en cauce situado en el proceso padre
39         close(fd[1]);
40
41         //Duplicar el descriptor de lectura de cauce en el descriptor
42         //correspondiente a la entrada estándar (stdin), cerrado previamente en
43         //la misma operación
44         dup2(fd[0], STDIN_FILENO);
45
46         execlp("sort", "sort", NULL);
47     }
48
49     return(0);
50 }
```

La ejecución es la siguiente:

```
morris@morris-GL63-8RD:~/Documentos/S0/Modulo2/sesion4$ ./tarea8
ComunicacionFIFO
consumidorFIFO
consumidorFIFO.c
productorFIFO
productorFIFO.c
tarea6
tarea6.c
tarea7
tarea7.c
tarea8
tarea8.c
morris@morris-GL63-8RD:~/Documentos/S0/Modulo2/sesion4$
```

Como se puede apreciar, sale el mismo resultado que en el ejercicio anterior. La única diferencia es que se ha usado la orden dup2 en vez de dup, que cierra y duplica los descriptors del tirón. Esto evita por tanto condiciones de carrera, ya que lo hace todo a la vez. Los 2 programas hacen lo mismo aun así.