# Ejercicio 9 (T4)

La Figura 1 muestra el código del programa "prog"(por simplicidad, se omite el código de tratamiento de errores). Contesta de forma **JUSTIFICADA** a las siguientes preguntas, suponiendo que ejecutamos este programa con el siguiente comando:

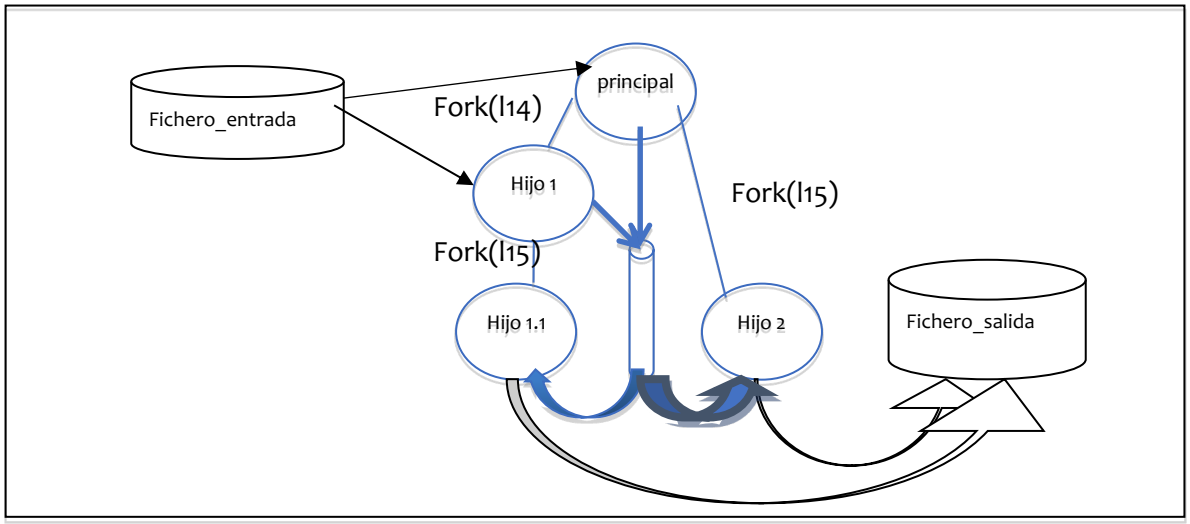
%./prog fichero_salida < fichero_entrada

Y Suponiendo que "fichero_entrada" existe y su contenido es "abcdefghijklmnñopqrstuvwxyz"

1. ¿Cuántos procesos crea este programa? ¿Cuántas pipes? Indica en qué línea de código se crea cada proceso y cada pipe de las que especifiques.

Crea 3 procesos: el proceso principal ejecuta el primer fork para crear un hijo, y luego ambos procesos ejecutan el siguiente fork creando otro hijo cada 1. Se crea una sola pipe, lo hace el proceso principal antes de crear ningún proceso.

2. Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Representa también la(s) pipe(s) que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una. En el dibujo indica también que procesos acceden a los ficheros "fichero_entrada" y "fichero_salida" y el tipo de acceso (lectura/escritura). Represéntalo mediante flechas que indiquen el tipo de acceso.



```
1.  main(int argc char *argv[]) {
2.  int fd_pipe[2];
3.  int fd_file;
4.  int ret, pid1, pid2;
5.  char c;
6.  char buf[80];
7.  int size;
8.
9.  close(1);
10. fd_file = open (argv[1], O_WRONLY|O_TRUNC|O_CREAT, 0660);
11.
12. pipe(fd_pipe);
13.
14. pid1 = fork();
15. pid2 = fork();
16.
17. if (pid2 == 0) {
18.       while ((ret = read(fd_pipe[0],&c, sizeof(c))) > 0)
19.             write (1,&c, sizeof(c));
20. } else {
21.       while ((ret=read(0,&c,sizeof(c)) > 0)
22.             write(pipe_fd[1], &c, ret);
23.
24.       while (waitpid(-1, NULL, 0) > 0);
25.       sprintf(buf, "Fin ejecución\n");
26.       write(2,buf,strlen(buf));
27. }
28. }
```

*Figura 1 Código de prog*

3. Completa la siguiente figura con los campos que faltan para que represente el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodes al inicio de la ejecución de este programa (línea 1).

Tabla Canales

| | **Ent. TFA** |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 2 | 0 |
| | |
| | |
| | |
| | |

Tabla Ficheros Abiertos

| | **#refs** | **Mod** | **Punt l/e** | **Ent t. inodes** |
|---|---|---|---|---|
| 0 | 2 | rw | -- | 0 |
| 1 | 1 | r | 0 | 1 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Tabla i-nodes

| | **#refs** | **inode** |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |
| | | |
| | | |
| | | |
| | | |
| | | |

4. Completa ahora la siguiente figura para representar el estado de las tablas de canales de cada proceso, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos se encuentran en la línea 16.

Tabla Canales

| | Ent. TFA |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |

Tabla Ficheros Abiertos

| | #refs | Mod | Punt l/e | Ent t. inodes |
|---|---|---|---|---|
| 0 | 4 | rw | -- | 0 |
| 1 | 4 | r | 0 | 1 |
| 2 | 4 | w | 0 | 2 |
| 3 | 4 | r | -- | 3 |
| 4 | 4 | w | -- | 3 |

Tabla i-nodes

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |
| 2 | 1 | i_fichSal |
| 3 | 2 | i_pipe |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |
| |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |
| |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |

5. ¿Qué procesos acabarán la ejecución?

Ninguno. Los procesos hijos ejecutan un bucle que lee de la pipe mientras la lectura no devuelva 0. Pero eso nunca pasará porque nunca se cierran los canales de escritura en la pipe. Y los procesos padre se quedan bloqueados esperando

6. ¿Qué líneas de código y en qué posición las añadirías para conseguir que todos los procesos acaben la ejecución?

Hay que añadir close(fd_pipe[1]) entre las líneas 17 y 18, y en la línea 23

7. ¿Qué procesos leen "fichero_entrada"? ¿Podemos saber qué fragmento del fichero lee cada proceso? Si repetimos más veces la ejecución de este programa, con el mismo comando, ¿podemos garantizar que los mismos procesos leerán los mismos fragmentos del fichero?

El padre e hijo 1 pueden leer del fichero. El fragmento que lea cada uno dependerá del orden en el que se intercalen en la cpu: sabemos que se lee hasta el final del fichero, que comparten el puntero de lectura/escritura. Por tanto cada carácter será leído sólo por uno, pero no podemos saber cuántos caracteres leerá cada proceso. Y esto puede cambiar entre ejecuciones.

8. Al final de la ejecución, ¿cuál será el contenido de "fichero_salida"? Si repetimos más veces la ejecución de este programa, con el mismo comando, ¿podemos garantizar que el contenido será siempre el mismo?

No podemos garantizar el contenido del fichero ya que puede haber un cambio de contexto entre la lectura de la pipe y la escritura en el fichero. Y también entre la lectura del fichero y la escritura en la pipe.

# Ejercicio 12 (T4)

Tenemos el programa "prog" que se genera al compilar el siguiente código (por simplicidad, se omite el código de tratamiento de errores):

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <string.h>
4.
5.  main() {
6.
7.  int fd[2];
8.  int ret, pid1, pid2;
9.  char c;
10. char buf[80];
11.
12.
13. pid1 = fork();
14. pipe(fd);
15. pid2 = fork();
16.
17. if (pid2 == 0) {
18.       while ((ret = read(fd[0],&c, sizeof(c))) > 0)
19.             write (1,&c, sizeof(c));
20. } else {
21.       sprintf(buf, "Te notifico mi pid: %d\n",getpid());
22.       write(fd[1], buf, strlen(buf));
23.       while (waitpid(-1, NULL, 0) > 0);
24. }
25.
26. }
27.
```

Contesta de forma justificada a las siguientes preguntas.

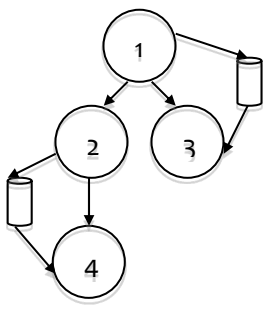
**Suponiendo que el programa se ejecuta de la siguiente manera:**

**%./prog**

1. Análisis del código:

   a) Indica cuántos procesos crea este programa y cuántas pipes. Además representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Indica claramente que procesos se comunican entre sí y con qué pipe (asígnale a las pipes alguna etiqueta si crees que puede ayudar). Representa también el uso concreto que hace cada proceso de cada pipe: qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una.

Crea 3 procesos (a parte del inicial) y 2 pipes, ya que la llamada pipe está justo después del fork y no está condicionada al resultado del fork.



b) ¿Qué proceso(s) ejecutará(n) las líneas de código 18 y 19? ¿Y las líneas entre la 21 y la 23? (justifícalo)

La 18 y la 19 los últimos procesos creados (3 y 4), y de la 21 a la 23 el proceso inicial y su primer hijo (1 y 2)

c) ¿Es necesario añadir algún código para sincronizar a los procesos en el acceso a la(s) pipe(s)?

No, porque en cada pipe sólo hay un lector y un escritor. Y entre parejas no hace falta que se sincronicen.

d) ¿Qué mensajes aparecerán en la pantalla?

Aparecerá "Te notifico mi pid:1 " y "Te notifico mi pid: 2". Ya que los procesos 3 y 4 escriben byte a byte, los textos podrían aparecer mezclados en la salida std. Ya que es compartida.

e) ¿Qué procesos acabaran la ejecución? ¿Por qué? ¿Qué cambios añadirías al código para conseguir que todos los procesos acaben la ejecución sin modificar la funcionalidad del código? Se valorará que el número de cambios sea el menor posible.

Ninguno. Los procesos lectores de las pipes se quedarán bloqueados esperando a que alguien escriba algo en los canales de escritura de las pipes. Y los procesos escritores (que son los padres de los lectores) se quedarán bloqueados en el waitpid esperando a que sus hijos acaben. Sólo hace falta cerrar los canales de escritura de la pipe: los lectores lo deben hacer antes de entrar en el bucle de lectura y los escritores después de escribir.

2. Acceso a las estructuras de datos del kernel:

La siguiente figura representa el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodes al inicio de la ejecución de este programa. Completa la figura representando el estado de las tablas de canales de todos los procesos que intervienen, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos se encuentran en la línea 16. (en la figura solo se muestra la tabla de canales del proceso inicial)

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | |

| TFA | #ref | modo | Puntl/e | Ent.T. inodo |
|---|---|---|---|---|
| 0 | 12 | rw | | 0 |
| 1 | 2 | R | | 1 |
| 2 | 2 | W | | 1 |
| 3 | 2 | R | | 2 |
| 4 | 2 | W | | 2 |
| 5 | | | | |

| T.inodo | #ref | inodo |
|---|---|---|
| 0 | 1 | tty |
| 1 | 2 | Pipe1 |
| 2 | 2 | Pipe2 |
| 3 | | |
| 4 | | |
| 5 | | |

Proceso 2

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |
| 5 | |

Proceso 3

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | |

Proceso 4

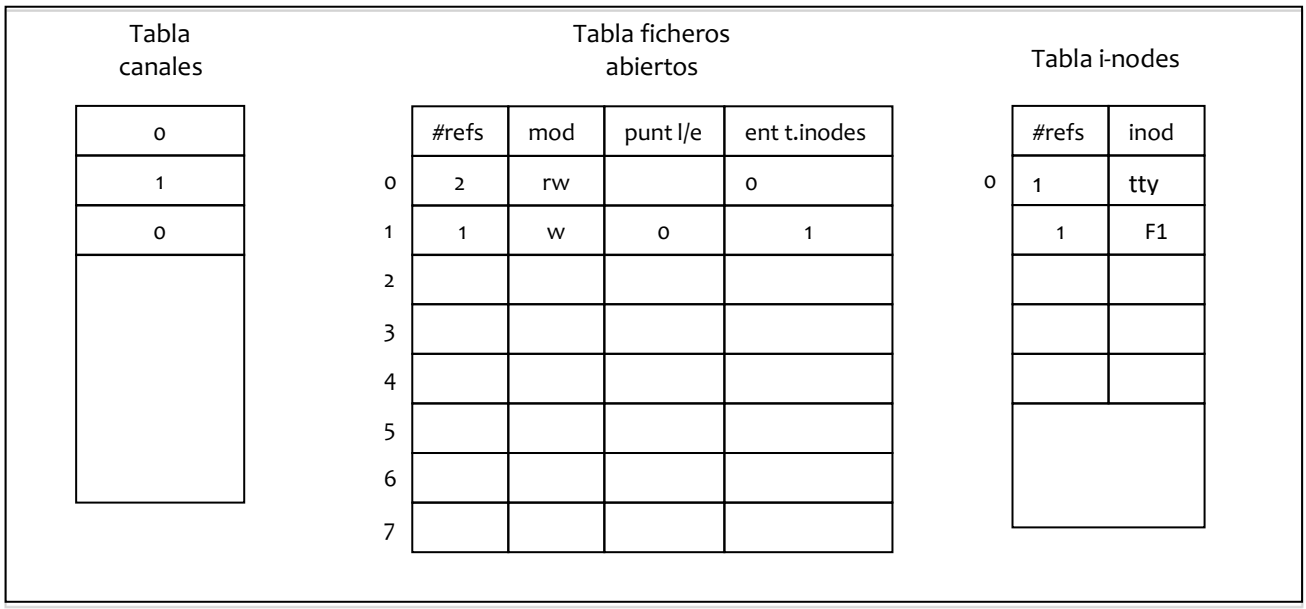
| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |
| 5 | |

3. Modificamos su comportamiento

**Suponiendo que el programa se ejecuta de la siguiente manera:**

**%./prog > f1**

a) ¿Cambiaría de alguna manera el estado inicial representado en la figura del apartado anterior? Representa en la siguiente figura el estado inicial que tendríamos al ejecutar de esta manera el programa.




b) ¿Qué mensajes aparecerán en pantalla?

Ninguno. Hemos redireccionado la salida estándar para asociarla a f1 y todos los procesos la heredarán.

c) ¿Qué procesos acabarán la ejecución? ¿Por qué? ¿Qué cambios añadirías al código para conseguir que todos los procesos acaben la ejecución sin modificar la funcionalidad del código? Se valorará que el número de cambios sea el menor posible.

Ninguno. El cambio de la redirección no afecta a la causa del bloqueo del ejercicio.

## Ejercicio 16 (T4)

Tenemos los siguientes códigos prog1.c y prog2.c, de los que omitimos la gestión de errores para facilitar la legibilidad:

```
1.  / * codigo de prog1.c */
2.  main() {
3.
4.  int pid_h;
5.  int pipe_fd[2];
6.  char buf [80];
7.
8.  pipe(pipe_fd);
9.
10. pid_h =fork();
11.
12. dup2(pipe_fd[0], 0);
13. dup2(pipe_fd[1], 1);
14.
15. close(pipe_fd[0]);
16. close(pipe_fd[1]);
17.
18. sprintf(buf, "%d", pid_h);
19.
20. execlp("./prog2", "prog2", buf, (char *) NULL);
21.
22.
23. }
24.
```

```
1.  / * codigo de prog2.c */
2.  int turno_escr;
3.
4.  void trat_sigusr1(int signum) {
5.      turno_escr = 1;
6.  }
7.
8.  main (int argc, char *argv[]) {
9.
10. char buf [80];
11. int pid_dest;
12. int i,ret,valor_rec;
13. int valor  = getpid();
14. struct sigaction trat;
15. trat.sa_flags = 0;
16. trat.sa_handler=trat_sigusr1;
17. sigemptyset(&trat.sa_mask);
18.
19. sigaction(SIGUSR1, &trat, NULL);
20.
21. pid_dest = atoi(argv[1]);
22.
23. if (pid_dest == 0) {
24.     pid_dest = getppid();
25.     write(1, &valor,sizeof(valor));
26.     turno_escr = 0;
27.     kill(pid_dest,SIGUSR1);
28.     valor ++;
29.
30. } else {
31.     turno_escr = 0;
32. }
33.
34. for (i = 0; i< 5; i++) {
35.     while (!turno_escr);
36.     ret=read (0,&valor_rec,sizeof(valor_rec));
37.     sprintf(buf,"%d",valor_rec);
38.     write(2,buf,ret);
39.     write(2,"\n",1);
40.     write(1,&valor,sizeof(valor));
41.     turno_escr = 0;
42.     kill(pid_dest,SIGUSR1);
43.     valor ++;
44.
```

Supón que en el directorio actual de trabajo tenemos los ejecutables prog1 y prog2, y que ejecutamos el siguiente comando: %./prog1. Contesta razonadamente a las siguientes preguntas

1. ¿Cuántos procesos se crearán? ¿Cuántas pipes?

   2 procesos contanto el inicial. Una pipe

4. Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Representa también las pipes que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una.


5. ¿Qué proceso(s) ejecutará(n) el código de prog2?

Los dos procesos, ya que ambos ejecutan la llamada a sistema exec que tenemos en prog1

6. ¿Qué mensajes aparecerán en el terminal?

El proceso inicial mostrará 5 mensajes, que será un número entero partiendo del pid de su hijo

Y el proceso hijo mostrará 5 mensajes, que también serán números enteros partiendo del pid de su padre

7. Describe brevemente el funcionamiento de este código. ¿Para qué se están utilizando los signals?

Los dos procesos hacen el mismo trabajo pero de manera alternada. El proceso que tiene el turno escribe un valor en la pipe y queda a la espera de recibir de nuevo el turno. Cuando recibe el turno (signal SIGUSR1) lee de la pipe, muestra en salida de errores estándar lo que ha recibido, incrementa el valor y cede el turno enviando al otro proceso un SIGUSR1. El primer proceso en tener el turno es el hijo, ya que cada proceso recibe como parámetro del main de prog2 el valor de retorno del fork. Los signals se están utilizando para sincronizar el acceso a la pipe.

8. Supón que el código prog1 ocupa 1KB y el código de prog2 ocupa 4KB. La máquina en la que ejecutamos estos códigos tiene un sistema de memoria basado en paginación, las páginas miden 4KB y utiliza la optimización copy-on-write en el fork. ¿Cuánta memoria necesitaremos para soportar el código de todos los procesos simultáneamente, suponiendo que cada proceso se encuentra en la última instrucción antes de acabar su ejecución?

Cada proceso carga el programa prog2, que ocupa 4 KB. Como las páginas son de 4KB necesitaremos 8KB para el código de los dos procesos. Justo antes de acabar la ejecución no tiene efecto el copy-on-write porque ambos procesos, padre e hijo, han mutado después del fork.

9. La siguiente figura representa el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodos al inicio de la ejecución de este programa. Completa la figura representando el estado de las tablas de canales de cada proceso, la tabla de ficheros abiertos y la tabla de inodos, suponiendo que todos los procesos están ejecutando la última instrucción antes de acabar su ejecución.

T. canales

| | |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| | |
| | |

T. Ficheros abiertos

| | #refs | Modo | Punt l/e | Ent T. inodos |
|---|---|---|---|---|
| 0 | 3 | rw | -- | 0 |
| | | | | |
| | | | | |
| | | | | |

T. inodos

| | #refs | inodo |
|---|---|---|
| 0 | 1 | i-tty |
| | | |
| | | |
| | | |

| | T. canales |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 0 |
| | |
| | |

| | T. Ficheros abiertos | | | |
|---|---|---|---|---|
| | #refs | Modo | Punt l/e | Ent T. inodes |
| 0 | 1 | rw | -- | 0 |
| 1 | 1 | r | -- | 1 |
| 2 | 1 | w | -- | 1 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

| | T. inodes | |
|---|---|---|
| | #refs | inode |
| 0 | 1 | i-tty |
| 1 | 2 | i-pipe |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

| | T. canales |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 0 |

# Ejercicio 20 (T2, T4)

La *Figura 2* contiene el código del programa "prog" (por simplicidad, se omite el código de tratamiento de errores). Contesta de forma **JUSTIFICADA** a las siguientes preguntas, suponiendo que ejecutamos este programa con el siguiente comando y que ninguna llamada a sistema provoca error:

%./prog < fichero_entrada > fichero salida

Y suponiendo que "fichero_entrada" existe y su contenido es "abcdefghijklmnñopqrstuvwxyz"

1. ¿Cuántos procesos crea este programa? ¿Cuántas pipes? Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un identificador para poder referirte a ellos en el resto de preguntas. Representa también las pipes que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una. En el dibujo indica también (mediante flechas) que procesos acceden a los ficheros "fichero_entrada" y "fichero_salida" y el modo de acceso (lectura/ escritura).

Se crean 2 procesos (contando al inicial en total hay 3 procesos) y se crean dos pipes. El proceso inicial crea una pipe en la línea 6 y a continuación crea a un proceso en la línea 7. Este proceso hijo creará otra pipe en la línea 13 y a otro proceso en la línea 14. El proceso inicial empieza la ejecución con la entrada estándar asociada a fichero_entrada y no modifica esa asociación. Por tanto cuando mute para ejecutar cat y lea de canal 0 estará leyendo de fichero_entrada. También empieza la ejecución con la salida estándar redireccionada , en este caso asociada a fichero_salida y así la hereda su hijo. Sin embargo, despúes de crear al hijo, asocia el canal 1 al extremo de escritura de la pipe 1. Por lo que cuando mute y el código de cat escriba en canal 1 lo hará en esa pipe. El primer hijo (hijo 1) crea a otro hijo antes de redireccionar ningún canal, por lo que este nuevo proceso hereda los canales estándar tal y como los tenía el proceso inicial al empezar la ejecución. El Hijo 1, una vez creado a hijo 1.1., modifica la asociación de los canales estándar de la siguiente manera: asocia canal 0 al extremo de lectura de pipe1 y el canal 1 al extremo de escritura de pipe 2. Por lo que cuando mute para ejecutar cat, leerá de una pipe y escribirá en la otra. Por último Hijo 1.1. únicamente modifica su canal 0 para asociarlo al extremo de lectura de la pipe 2, por lo que cuando mute para ejecutar el cat leerá lo que reciba por esa pipe y escribirá ese contenido en fichero_salida.

```
1.  main() {
2.  int fd_pipe1[2];
3.  int fd_pipe2[2];
4.  int ret, i;
5.  char buf[80];
6.      pipe(fd_pipe1);
7.      ret = fork();
8.      if (ret > 0){
9.          dup2(fd_pipe1[1],1); close(fd_pipe1[1]); close(fd_pipe1[0]);
10.         execlp("cat","cat",(char *) 0);
11.     } else {
12.         close(fd_pipe1[1]);
13.         pipe(fd_pipe2);
14.         ret = fork();
15.
16.         if (ret > 0) {
17.             dup2(fd_pipe1[0],0); close(fd_pipe1[0]);
18.             dup2(fd_pipe2[1],1); close(fd_pipe2[1]); close(fd_pipe2[0]);
19.             execlp("cat","cat",(char *) 0);
20.
21.         } else {
22.             close(fd_pipe1[0]);
23.             dup2(fd_pipe2[0],0), close (fd_pipe2[0]); close(fd_pipe2[1]);
24.             execlp("cat", "cat", (char *) 0);
25.
26.         }
27.     }
28.
29.     write (2, "Fin", 3);
30. }
```

*Figura 2 Código de prog*

2. Completa la siguiente figura con toda la información necesaria para representar el estado de las tablas de canales de todos los procesos, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos están justo antes de acabar la ejecución (nota: en el campo inode de la tabla i-nodes puedes poner algo que represente al i-node del dispositivo).

Tabla Canales Tabla Ficheros Abiertos Tabla i-nodes

| | Ent. TFA |
|---|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 0 |
| 3 | |
| 4 | |
| | |
| | |

| | #refs | Mod | Punt l/e | Ent t. inodes |
|---|---|---|---|---|
| 0 | 3 | rw | -- | 0 |
| 1 | 1 | r | end | 1 |
| 2 | 1 | w | end | 2 |
| 3 | 1 | r | -- | 3 |
| 4 | 1 | w | -- | 3 |
| 5 | 1 | r | -- | 4 |
| 6 | 1 | w | -- | 4 |

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |
| 2 | 1 | i_fichSal |
| 3 | 2 | i_pipe1 |
| 4 | 2 | i_pipe2 |
| | | |
| | | |

Tabla Canales

| | |
|---|---|
| 0 | 3 |
| 1 | 6 |
| 2 | 0 |
| | |
| | |
| | |

Tabla Canales

| | |
|---|---|
| 0 | 5 |
| 1 | 2 |
| 2 | 0 |
| | |
| | |
| | |

3. ¿Qué contendrá fichero_salida al final de la ejecución? ¿Qué mensajes aparecerán en pantalla?

Fichero_salida contiene una copia exacta de fichero_entrada. El proceso inicial lee fichero entrada y escribe su contenido en la primera pipe. El proceso hijo 1 lee de esa pipe todo ese contenido y lo escribe en la segunda pipe. Por último el proceso hijo 1.1. lee de esa segunda pipe todo el contenido y lo escribe en fichero_salida. En pantalla no aparece ningún mensaje porque la única línea que intenta escribir en terminal es la 29 (escribe en salida de errores estándar que está asociada al terminal) pero ningún proceso la llega a ejecutar porque todos mutan antes.

4. ¿Acabarán la ejecución todos los procesos?

Sí que acaban todos. La única situación que podría provocar que un proceso no acabara es que no acabara el bucle de lectura que ejecuta el comando cat. En cuanto un proceso acaba de escribir en la pipe muere y por lo tanto se cierran todos sus canales. Y ningún proceso se deja abierto ningún canal de escritura en la pipe.

5. (*T2)* Queremos modificar el código para que cada cierto tiempo (por ejemplo cada segundo) aparezca un mensaje en la pantalla. Se nos ocurre la siguiente modificación del código (los cambios aparecen marcados en negrita):

```
1.  int trat_sigarlm(int signum) {
2.  char buf[80];
3.      strcpy(buf, "Ha llegado alarma!");
4.      write(2,buf,strlen(buf));
5.      alarm(1);
6.  }
7.  main() {
8.  int fd_pipe1[2];
9.  int fd_pipe2[2];
10. int ret, i;
11. char buf[80];
12. struct sigaction trat;
13.    trat.sa_flags = 0;
14.     trat.sa_handler = trat_sigalrm;
15.     sigemtpyset(&trat.sa_mask);
16.     sigaction(SIGARLM ,trat_sigalrm,NULL);
17.     alarm(1);
18.     pipe(fd_pipe1);
19.     ret = fork();
20.     // A PARTIR DE AQUI NO CAMBIA NADA
21.    //...
```

Suponiendo que el proceso inicial tarda más de 1 segundo en acabar, ¿funcionará? ¿Qué mensajes aparecerán ahora por pantalla? ¿Qué procesos los escribirán? ¿Cambiará en algo el resultado de la ejecución?

No funcionará porque al mutar para ejecutar el cat perdemos la reprogramación de la alarma. El único proceso que recibe la alarma es el inicial, si recibiera alguna alarma antes de mutar escribiría el mensaje "ha llegado alarma" y continuaría la ejecución. Pero en cuanto reciba una alarma una vez mutado se ejecutará el tratamiento por defecto de la alarma y el proceso morirá. El fichero de salida contendrá la parte del fichero de entrada que hayamos tenido tiempo de transmitir.