Practica 1

SEMANA 1: INTRODUCCIÓN A LA SHELL. HILOS.

Ejercicio 1: Uso del Manual

a) Buscar en el manual la lista de funciones disponibles para el manejo de hilos y copiarla en la memoria junto con el comando usado para mostrarla. Las funciones de manejo de hilos comienzan por “pthread”.

```
$ man -k pthread
pthread_attr_destroy (3) - initialize and destroy thread attributes object
pthread_attr_getaffinity_np (3) - set/get CPU affinity attribute in thread
attributes object
pthread_attr_getdetachstate (3) - set/get detach state attribute in thread
attributes object
pthread_attr_getguardsize (3) - set/get guard size attribute in thread
attributes object
pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in thread
attributes object
pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in
thread attributes object
pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in thread
attributes object
pthread_attr_getscope (3) - set/get contention scope attribute in thread
attributes object
pthread_attr_getstack (3) - set/get stack attributes in thread attributes object
pthread_attr_getstackaddr (3) - set/get stack address attribute in thread
attributes object
pthread_attr_getstacksize (3) - set/get stack size attribute in thread
attributes object
pthread_attr_init (3) - initialize and destroy thread attributes object
pthread_attr_setaffinity_np (3) - set/get CPU affinity attribute in thread
attributes object
pthread_attr_setdetachstate (3) - set/get detach state attribute in thread
attributes object
pthread_attr_setguardsize (3) - set/get guard size attribute in thread
attributes object
pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in thread
attributes object
pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in
thread attributes object
pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in thread
attributes object
pthread_attr_setscope (3) - set/get contention scope attribute in thread
```

```
attributes object
pthread_attr_setstack (3) - set/get stack attributes in thread attributes object
pthread_attr_setstackaddr (3) - set/get stack address attribute in thread
attributes object
pthread_attr_setstacksize (3) - set/get stack size attribute in thread
attributes object
pthread_cancel (3)   - send a cancellation request to a thread
pthread_cleanup_pop (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_pop_restore_np (3) - push and pop thread cancellation clean-up
handlers while saving cancelability type
pthread_cleanup_push (3) - push and pop thread cancellation clean-up handlers
pthread_cleanup_push_defer_np (3) - push and pop thread cancellation clean-up
handlers while saving cancelability type
pthread_create (3)   - create a new thread
pthread_detach (3)   - detach a thread
pthread_equal (3)    - compare thread IDs
pthread_exit (3)     - terminate calling thread
pthread_getaffinity_np (3) - set/get CPU affinity of a thread
pthread_getattr_default_np (3) - get or set default thread-creation attributes
pthread_getattr_np (3) - get attributes of created thread
pthread_getconcurrency (3) - set/get the concurrency level
pthread_getcpuclockid (3) - retrieve ID of a thread's CPU time clock
pthread_getname_np (3) - set/get the name of a thread
pthread_getschedparam (3) - set/get scheduling policy and parameters of a thread
pthread_join (3)     - join with a terminated thread
pthread_kill (3)     - send a signal to a thread
pthread_kill_other_threads_np (3) - terminate all other threads in process
pthread_mutex_consistent (3) - make a robust mutex consistent
pthread_mutex_consistent_np (3) - make a robust mutex consistent
pthread_mutexattr_getpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_getrobust (3) - get and set the robustness attribute of a
mutex attributes object
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of a
mutex attributes object
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a
mutex attributes object
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of a
mutex attributes object
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the
thread read-write lock attribute object
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the
thread read-write lock attribute object
pthread_self (3)     - obtain ID of the calling thread
pthread_setaffinity_np (3) - set/get CPU affinity of a thread
pthread_setattr_default_np (3) - get or set default thread-creation attributes
pthread_setcancelstate (3) - set cancelability state and type
pthread_setcanceltype (3) - set cancelability state and type
pthread_setconcurrency (3) - set/get the concurrency level
pthread_setname_np (3) - set/get the name of a thread
```

```
pthread_setschedparam (3) - set/get scheduling policy and parameters of a thread
pthread_setschedprio (3) - set scheduling priority of a thread
pthread_sigmask (3)  - examine and change mask of blocked signals
pthread_sigqueue (3) - queue a signal and data to a thread
pthread_spin_destroy (3) - initialize or destroy a spin lock
pthread_spin_init (3) - initialize or destroy a spin lock
pthread_spin_lock (3) - lock and unlock a spin lock
pthread_spin_trylock (3) - lock and unlock a spin lock
pthread_spin_unlock (3) - lock and unlock a spin lock
pthread_testcancel (3) - request delivery of any pending cancellation request
pthread_timedjoin_np (3) - try to join with a terminated thread
pthread_tryjoin_np (3) - try to join with a terminated thread
pthread_yield (3)    - yield the processor
pthreads (7)         - POSIX threads
```

b) Consultar en la ayuda en qué sección del manual se encuentran las “llamadas al sistema” y buscar información sobre la llamada al sistema write. Escribir en la memoria los comandos usados.

```
$ man man
```

En la descripción encontramos la siguiente línea:

```
       2   Llamadas del sistema (funciones servidas por el núcleo)
```

```
$ man 2 write
```

Esta función nos da la información de la llamada al sistema “write”

Ejercicio 2: Comandos y Redireccionamiento.

a) Escribir un comando que busque las líneas que contengan “molino” en el fichero “don quijote.txt” y las añada al final del fichero “aventuras.txt”. Copiar el comando en la memoria, justificando las opciones utilizadas.

```
grep molino 'don quijote.txt' >> aventuras.txt
```

Usamos grep para encontrar las frases con “molino”, y usamos >> para añadirlo al final del fichero (o crearlo si no existe)

b) Elaborar un pipeline que cuente el número de ficheros en el directorio actual. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

```
$ ls | wc -l
```

Utilizamos ls para ver los ficheros que hay en el directorio actual, luego | para construir el pipeline, y finalmente el número de ficheros mediante wc -l.

c) Elaborar un pipeline que cuente el número de líneas distintas al concatenar “lista de la compra Pepe.txt” y “lista de la compra Elena.txt” y lo escriba en “num compra.txt”. Si alguno de los ficheros no existe, hay que ignorar los mensajes de error, para lo cual se redirigirá la salida de errores a /dev/null. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados.

```
$ cat "lista de la compra Pepe.txt" "lista de la compra Elena.txt" 2> /dev/null/
| wc -l > "num compra.txt"
```

Al principio utilizamos cat para concatenar los dos archivos que vamos a utilizar, a continuación 2> para redireccionar si algun fichero no existe, luego el pipeline y finalmente >> para escribirlo.

d) (Opcional) Elaborar un pipeline que cuente el número de hilos de cada proceso del sistema y lo escriba en el fichero “hilos.txt”. Copiar el pipeline en la memoria, justificando los comandos y opciones utilizados. Los hilos del mismo proceso comparten la columna de identificador del proceso (la primera) en el comando ps. Para extraer la primera columna de cada línea de un fichero se puede usar el siguiente comando: awk '{print $1}'.

```
$ ps -L | wc -l >> hilos.txt
```

Utilizamos el comando ps -L para mostrarnos los hilos, después construimos el pipeline añadiendo wc -l para contar el número de hilos. Finalmente, escribimos el número de hilos en el archivo hilos.txt.

Ejercicio 3: Control de Errores.

Escribir un programa que abra un fichero indicado por el primer parametro en modo lectura usando la funcion fopen. En caso de error de apertura, el programa mostrar a el mensaje de error usando perror.

a) ¿Que mensaje se imprime al intentar abrir un fichero inexistente? ¿A que valor de errno corresponde?

El mensaje de error que se imprime es: “Error: No such file or directory”.
El valor de errno correspondiente es 2.

b) ¿Que mensaje se imprime al intentar abrir el fichero/etc/shadow? ¿A que valor de errno corresponde?

El mensaje de error que se imprime es: “Error: Permission denied”.
El valor de errno correspondiente es 13.

c) Si se desea imprimir el valor de errno antes de la llamada a perror, ¿que modificaciones se deberıan realizar para garantizar que el mensaje de perror se corresponde con el error de fopen?
En el momento en el que se capta el error, guardamos el código de error en una variable de tipo int y luego usamos la función strerror.

Ejercicio 4: Espera Activa e Inactiva.

a) Escribir un programa que realice una espera de 10 segundos usando la funcion clock en un bucle. Ejecutar en otra terminal el comando top. ¿Que se observa?

Al realizar el programa y ejecutarlo, se puede observar como este tiene prácticamente el 100% de la CPU durante los 10 segundos de bucle.

```
31280 pablo 20   04372792728 R  99,7  0,0   0:03.08 a.out
```

b) Reescribir el programa usando sleep y volver a ejecutar top. ¿Ha cambiado algo?

Cuando se reescribe el programa con el comando sleep, este no aparece en la lista que se abre al ejecutar top, lo que quiere decir que no consume recursos de la CPU.

Ejercicio 5: Finalización de Hilos.

a) ¿Que hubiera pasado si el proceso no hubiera esperado a los hilos? Para probarlo basta eliminar las llamadas a pthread_join.

Al eliminar las llamadas a pthread_join, solo se escriben como output las dos primeras letras de las palabras que se escriben (H y M), ya que es lo que devuelve el pthread_create. Despues, sale el mensaje del final.

```
 H  M El  programa  ./a.out termino  correctamente
```

b) Con el codigo modificado del apartado anterior, indicar que ocurre si se reemplaza la funcion exit por una llamada a pthread_exit.

Si se reemplaza, no se escribe nada de las palabras que contiene pthread_create, solo el mensaje final.

```
El  programa  ./a.out termino  correctamente
```

c) Tras eliminar las llamadas a pthread_join en los apartados anteriores, el programa es ahora incorrecto porque no se espera a que terminen todos los hilos. Escribir en la memoria el codigo que seria necesario añadir para que sea correcto no esperar a los hilos creados.

Habría que añadir las siguientes líneas de código:

```
error = pthread_detach(h1);
   if (error != 0) {
       fprintf(stderr, "pthread_detach: %s\n", strerror(error));
   }
   error = pthread_detach(h2);
   if (error != 0) {
       fprintf(stderr, "pthread_detach: %s\n", strerror(error));
   }
```

Ejercicio 6: Creación de Hilos y Paso de Parámetros.

Escribir un programa en C (“ejercicio_hilos.c”) que satisfaga los siguientes requisitos:

⇒ Creará tantos hilos como se le indique por parámetro.
⇒ Cada hilo esperará un número aleatorio de segundos entre 0 y 10 inclusive, que será generado por el hilo principal. Después realizará el cálculo x^3, donde x será el número del hilo creado. Por último devolverá el resultado del cálculo en un nuevo entero, reservado dinámicamente.
⇒ El hilo principal deberá esperar a que todos los hilos terminen e imprimir todos los resultados devueltos por los hilos.
⇒ Como la función pthread_create solo admite el paso de un único parámetro habrá que crear un struct con ambos parámetros (tiempo de espera y valor de x).
⇒ El programa deberá finalizar correctamente liberando todos los recursos utilizados.
⇒ Deberá asimismo controlar errores, y terminar imprimiendo el mensaje de error correspondiente si se produce alguno.
ejercicio_hilos.c

SEMANA 2: PROCESOS Y EJECUCIÓN DE PROGRAMAS.

Ejercicio 7: Creación de Procesos.

a) Analizar el texto que imprime el programa. ¿Se puede saber a priori en qué orden se imprimirá el texto? ¿Por qué?

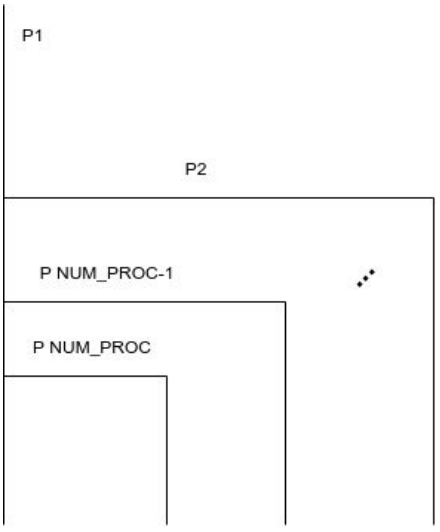
No, porque los procesos se ejecutan de forma paralela, lo que hace imprevisible saber cuál será el orden de los procesos.

b) Cambiar el código para que el proceso hijo imprima su PID y el de su padre en vez de la variable i. Copiar las modificaciones en la memoria y explicarlas.

```
printf("PID:    %d    PPID:    %d\n",    getpid(),
getppid());
```

Utilizamos getpid() para obtener el PID del proceso actual (concretamente del proceso hijo) y getppid para obtener el proceso padre.

c) Analizar el árbol de procesos que genera el código de arriba. Mostrarlo en la memoria como un diagrama de árbol (como el que aparece en el Ejercicio 8) explicando por qué es así.



P1 es el padre de todos los procesos. Al entrar en el bucle for lo primero que se hace es un fork (pid = fork()), luego se genera un proceso casi idéntico al primero (P2) y ese pid se va comprobando en los diferentes if else que tiene el bucle (cuando es menor que 0, es igual a cero o es mayor que cero) lo que da lugar a tres procesos hijo. Como el for se repite hasta NUM_PROC que está definido como 3, se generarán 3 procesos hijos con el fork, y otros 3 en cada hijo.

d) El código anterior deja procesos huérfanos, ¿por qué?

Porque solo hay una sentencia wait en el código, lo que significa que el proceso padre espera al primer hijo que termine y no al resto, dejándolos huérfanos.

e) Introducir el mínimo número de cambios en el código para que no deje procesos huérfanos. Copiar las modificaciones en la memoria y explicarlas.

```
while(wait(NULL) != -1);
```

Utilizamos un bucle while y dentro utilizamos la condición wait() != -1. Sabemos que wait devolverá-1 si no existe ningún proceso hijo, por lo tanto, una vez salga del bucle estaremos seguros de que todos los procesos hijo habrán terminado y no quedará un proceso huérfano.

Ejercicio 8: Arbol de Procesos.

Escribir un programa en C (“ejercicio_arbol.c”) que genere el árbol de procesos de la figura.

El proceso padre genera un proceso hijo, que a su vez generará otro hijo, y así hasta llegar a NUM_PROC procesos en total. El programa debe garantizar que cada padre espera a que termine su hijo, y no quede ningún proceso huérfano.

ejercicio_arbol.c

Ejercicio 9: Espacio de Memoria.

a) En el programa anterior se reserva memoria en el proceso padre y se inicializa en el proceso hijo usando la función strcpy (que copia un string a una posición de memoria). Una vez el proceso hijo termina, el padre lo imprime por pantalla. ¿Qué ocurre cuando se ejecuta el código? ¿Es este programa correcto? ¿Por qué?

Cuando se ejecuta el código, el programa tiene como salida “Padre: “, es decir, no hay ningún resultado, luego el programa es incorrecto. Esto se debe a que los procesos no comparten memoria, por eso, al hacer la función strcpy, el mensaje no lo tiene el padre, ya que la memoria no es compartida.

b) El programa anterior contiene una fuga de memoria ya que el array sentence nunca se libera. Corregir el código para eliminar esta fuga y copiar las modificaciones en la memoria. ¿Dónde hay que liberar la memoria, en el proceso padre, en el hijo o en ambos? ¿Por qué?

Las modificaciones en el código son:

```
strcpy(sentence, MESSAGE);
free(sentence);
exit(EXIT_SUCCESS);
```

```
wait(NULL);
printf("Padre: %s\n", sentence);
free(sentence);
exit(EXIT_SUCCESS);
```

Como se puede comprobar, la liberación de memoria se hace tanto en el proceso padre como en el proceso hijo, esto ocurre ya que, aunque los procesos no comparten memoria, el proceso hijo si hereda recursos del padre y por ello hereda ese alojamiento de memoria que, al igual que el padre, tendrá que liberar.

Ejercicio 10: Shell.

Escribir un programa en C (“ejercicio_shell.c”) que implemente una shell sencilla (sin redirecciones ni estructuras de control).

a) Escribir el programa, satisfaciendo los siguientes requisitos:
⇒ Tendrá un bucle principal que pida una línea al usuario para cada comando, hasta leer EOF de la entrada estándar. Se puede introducir manualmente EOF en la entrada estándar mediante la combinación de teclas Ctrl+D. Para leer líneas se puede usar fgets o getline.
⇒ Cada línea deberá trocearse para separar el ejecutable (primer argumento) y los argumentos del programa (todos los argumentos, incluido el propio ejecutable). Esta tarea de troceado puede realizarse ayudándose de funciones de librería como strtok o, si se desea, se puede usar wordexp, que además realiza las tareas adicionales que haría la shell (expandir variables de entorno, permitir comillas, etc.).
⇒ Cada comando debe ejecutarse realizando un fork seguido de un exec en el hijo.
⇒ El proceso padre debe esperar al hijo y a continuación imprimir por la salida de errores Exited with value <valor> o Terminated by signal <señal>, en función de cómo terminó el hijo.
⇒ A continuación se realizará la siguiente iteración del bucle, leyendo el siguiente comando.

ejercicio_shell.c

b) Explicar qué función de la familia exec se ha usado y por qué. ¿Podría haberse usado otra? ¿Por qué?

Se ha usado la función execvp porque al contener la letra “v” recibe los argumentos que se pasarán a la función main a ejecutar en un array y, al contener la letra “p”, buscarán el nombre del fichero del ejecutable en los directorios que se encuentren en la variable de entorno PATH.
No podemos usar otra función ya que execvp ejecuta los directorios listados en la variable PATH mientras que otras funciones como execv solo son ejecutadas si se encuentra en el directorio actual.

c) Ejecutar con la shell implementada el comando sh -c inexistente. ¿Qué imprime?

Se imprime `sh: 1: inexistente: not found`

d) Hacer un programa en C que finalice llamando a abort y ejecutarlo con la shell implementada. ¿Qué se imprime ahora?

Se imprime `Abortado ('core' generado)`

e) (Opcional)Las funciones de POSIX posix_spawn y posix_spawnp permiten realizar la acción combinada de fork + exec de forma más sencilla y eficiente. Entregar una copia del ejercicio anterior (“ejercicio_shell_spawn.c”) en la que se reemplace fork+exec por una de estas funciones.

SEMANA 3: FICHEROS Y TUBERÍAS.

Ejercicio 11: Directorio de información de procesos.

Buscar para alguno de los procesos la siguiente información en el directorio /proc y escribir tanto la información como el fichero utilizado en la memoria. Hay que tener en cuenta que tanto las variables de entorno como la lista de comandos delimitan los elementos con \0, así que puede ser conveniente convertir los \0 a\n usando tr'\0''\n'.

El proceso sobre el que trabajamos es self.

a) El nombre del ejecutable.

Tras hacer `cd self` en el directorio /proc, para obtener el nombre utilizamos el comando `more status`, el cual, entre otra información nos dio el nombre:

```
Name:    exe
Umask:    0022
State:    S (sleeping)
Tgid:    10421
Ngid:    0
Pid:    10421
PPid:    10333
```

Como se puede ver, la primera información que nos da es el nombre, en este caso, exe.
b) El directorio actual del proceso.

El enlace (simbólico ya que /proc son ficheros virtuales) se obtiene mediante el comando `cd cwd` que nos lleva al mismo directorio, lo cual es lógico ya que self es el mismo proceso.

c) La línea de comandos que se usó para lanzarlo.

Para obtener la línea de comandos se utiliza el comando `more cmdline`, que, en este caso, es simplemente `./exe`

d) El valor de la variable de entorno LANG.

Para obtener la variable de entorno LANG se utiliza el comando `echo $LANG`, dando de vuelta `es_ES.UTF-8`, que en este caso quiere decir que es el idioma español.

También se puede ejecutar `more environ` donde nos aparecerá una lista con todas las variables de entorno, y donde coincide la variable LANG que hemos ejecutado con el comando anterior.

e) La lista de hilos del procesos.

Para saber la lista de hilos, utilizamos el siguiente pipeline: `cat status | grep Threads`.
Tras ejecutarlo, devuelve lo siguiente: `Threads:	1`

Ejercicio 12: Visualización de Descriptores de Fichero.

El programa se para en ciertos momentos para esperar a que el usuario pulse. Se pueden observar los descriptores de fichero del proceso en cualquiera de esos momentos si en otra terminal se inspecciona el directorio /proc/<PID>/fd, donde <PID> es el identificador del proceso. A continuación se indica qué hacer en cada momento.

a) Stop 1. Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos? ¿A qué tipo de fichero apuntan?

Los descriptores de fichero abiertos son: 0, 1, 2, 13, 14, 15, 16, 17, 18, 19, 36, 43, 47, 48, 49, 51, 52, 53, 54, 55, 103 y 104.
La mayoría de los ficheros son de tipo binario, pero hay un par de excepciones. Los ficheros 0, 1, 2 son de un tipo especial, y se tiene que utilizar -f para poder leerlos. Los ficheros 15 y 43 en cambio, cuando son inspeccionados por less, revelan una serie de funciones y constantes propias de un programa.

b) Stop 2 y Stop 3. ¿Qué cambios se han producido en la tabla de descriptores de fichero?

En el Stop 2 se ha creado un nuevo fichero, con el número 3, el cual al inspeccionar con more se puede leer “Hello”. Además, se crea un fichero llamado “file1.txt”, que es de tipo binario.

En el Stop 3, de nuevo se crea un fichero, con el número 6, pero a diferencia del anterior este no contiene nada si se inspecciona con more. Además, se crea un fichero llamado “file2.txt”, que si se abre, se puede comprobar que no contiene nada.

c) Stop 4. ¿Se ha borrado de disco el fichero FILE1? ¿Por qué? ¿Se sigue pudiendo acceder al fichero a través del directorio/proc? ¿Hay, por tanto, alguna forma sencilla de recuperar los datos?

El fichero “file1.txt” se borra de la carpeta de la que se encuentra, sin embargo no se borra del disco. Esto se debe a que hay más de una ruta a la que acceder a ese fichero y, por lo tanto, se puede seguir accediendo. De esta forma, se pueden recuperar los datos del fichero mediante el directorio /proc y buscando dicho fichero.

d) Stop 5, Stop 6 y Stop 7. ¿Qué cambios se han producido en la tabla de descriptores de fichero? ¿Qué se puede deducir sobre la numeración de un descriptor de fichero obtenido tras una llamada a open?

En el Stop 5, ya no aparece disponible el fichero “file1.txt” ya que no aparece su descriptor, deduciendo así que se han eliminado todas las rutas mediante la función unlink.

En el Stop 6, el descriptor de “file1.txt” vuelve a estar disponible, es decir, el número 3 vuelve a ser visible cuando se hace ls; sin embargo, cuando se inspecciona este no contiene nada, ya que no es el mismo fichero que se había creado anteriormente sino que tiene el mismo identificador. Además, este identificador se corresponde con el fichero “file3.txt”, el cual se puede comprobar que no contiene nada.

En el Stop 7, aparece un nuevo descriptor con el número 7, sin embargo no se crea ningún archivo file, lo que quiere decir que no se abre ningún fichero nuevo.

Sobre la numeración de un descriptor de fichero tras llamar a open se puede deducir que los descriptores se irán creando por orden y, si se borra un descriptor de fichero, el próximo open obtendrá el número del que ha sido eliminado.

Ejercicio 13: Problemas con el Buffer.

a) ¿Cuántas veces se escribe el mensaje “Yo soy tu padre” por pantalla? ¿Por qué?

Se ha escrito dos veces, esto se debe a que a que la salida se almacena en el búfer, por lo que al hacer fork(), el hijo también hereda el mensaje de “Yo soy tu padre” en su búfer y, por lo tanto, a la salida estándar, se volverá a repetir el mensaje por el hijo, tras hacer wait.

b) En el programa falta el terminador de línea (\n) en los mensajes. Corregir este problema. ¿Sigue ocurriendo lo mismo? ¿Por qué?

Al introducir /n deja de suceder que se escriba dos veces el mensaje porque es un carácter especial que indica el final de una línea, y, por lo tanto, hace que el búfer perciba que su array está lleno dando lugar a que este no se copie en el proceso hijo al hacer el fork.

c) Ejecutar el programa redirigiendo la salida a un fichero. ¿Qué ocurre ahora? ¿Por qué?

Al redirigir la salida a un fichero, vuelve a ocurrir que el mensaje de “Yo soy tu padre” se escribe dos veces. La razón de esto es porque estamos utilizando una función que utiliza objetos FILE. Debido a esto, el búfer guarda el contenido en el array y por lo tanto es heredado al hijo.

d) Indicar en la memoria cómo se puede corregir definitivamente este problema sin dejar de usar printf.

La forma de corregir este problema es utilizando fflush justo después del printf(“Yo soy tu padre”), para así para vaciar de manera manual el array y, de esta forma, evitar la repetición del mensaje.

Ejercicio 14: Ejemplo de Tuberías.

a) Ejecutar el código. ¿Qué se imprime por pantalla?

```
He escrito en el pipe
He recibido el string: Hola a todos!
```

b) ¿Qué ocurre si el proceso padre no cierra el extremo de escritura? ¿Por qué?

Al no cerrar el programa, aunque no da errores, se queda en constante ejecución. Esto ocurre porque al no cerrar el proceso de escritura el lector no reconoce que haya finalizado dicho proceso, dando lugar a que el lector se quede esperando de forma sempiterna.

c) (Opcional) Modificar el proceso hijo para que espere 1 segundo antes de escribir. Modificar el proceso padre para que finalice sin leer de la tubería y sin esperar al proceso hijo. ¿Qué se imprime ahora? ¿Por qué?

Al hacer las modificaciones requeridas, se imprime:

```
Process terminating with default action of signal 13 (SIGPIPE)
```

Esto se debe a que cuando el hijo escribe, no hay lectores al otro lado de la tubería, de tal forma que el Sistema Operativo manda el error de SIGPIPE, además, el proceso se cierra de forma automática.

Ejercicio 15: Comunicación entre Tres Procesos.

Escribir un programa en C(“ejercicio_pipes.c”) que satisfaga los siguientes requisitos:

⇒ El proceso inicial debe crear dos procesos hijos.

⇒ Mediante tuberías, el proceso padre se debe comunicar con uno de sus hijos para leer un número aleatorio x que generará dicho proceso hijo, y que enviará al padrea través de una tubería.

⇒ Este primer proceso hijo, antes de finalizar, debe imprimir por pantalla el número aleatorio que ha generado.

⇒ Una vez que el proceso padre tenga el número aleatorio del primer hijo,
se lo enviará al segundo proceso hijo a través de otra tubería distinta.

⇒ Este segundo proceso hijo debe leer el número de la tubería y escribirlo en el fichero “numero_leido.txt”, usando las funciones que suministra el Sistema Operativo para ello. Se puede usar la función dprintf para hacer escritura formateada usando un descriptor de fichero. El esquema de este programa se resume en la figura.

El programa debe tener en cuenta: (a) control de errores, (b) cierre de la tuberías pertinentes, y (c) espera del proceso padre a sus procesos hijo.

ejercicio_pipes.c