MEMORIA PRÁCTICA 1

SISTEMAS OPERATIVOS

Rubén García de la Fuente
Elena Cano Castillejo
Pareja 04
Grupo 2202

<u>Ejercicio 1:</u> Uso del Manual. *(0,25 ptos.)*

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"**. *(0,10 ptos.)*

```
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
       attri...
       pthread_attr_getdetachstate (3) - set/get detach state attribute in thread
       attri...
       pthread_attr_getguardsize (3) - set/get guard size attribute in thread
       attribute...
       pthread_attr_getinheritsched (3) - set/get inherit-scheduler attribute in
       thread...
       pthread_attr_getschedparam (3) - set/get scheduling parameter attributes in
       thre...
       pthread_attr_getschedpolicy (3) - set/get scheduling policy attribute in
       thread ...
       pthread_attr_getscope (3) - set/get contention scope attribute in thread
       attribu...
       pthread_attr_getstack (3) - set/get stack attributes in thread attributes
       object
       pthread_attr_getstackaddr (3) - set/get stack address attribute in thread
       attrib...
       pthread_attr_getstacksize (3) - set/get stack size attribute in thread
       attribute...
       pthread_attr_init (3) - initialize and destroy thread attributes object
       pthread_attr_setaffinity_np (3) - set/get CPU affinity attribute in thread
       attri...
       pthread_attr_setdetachstate (3) - set/get detach state attribute in thread
       attri...
       pthread_attr_setguardsize (3) - set/get guard size attribute in thread
       attribute...
       pthread_attr_setinheritsched (3) - set/get inherit-scheduler attribute in
       thread...
       pthread_attr_setschedparam (3) - set/get scheduling parameter attributes in
       thre...
       pthread_attr_setschedpolicy (3) - set/get scheduling policy attribute in
       thread ...
       pthread_attr_setscope (3) - set/get contention scope attribute in thread
       attribu...
       pthread_attr_setstack (3) - set/get stack attributes in thread attributes
       object
       pthread_attr_setstackaddr (3) - set/get stack address attribute in thread
       attrib...
       pthread_attr_setstacksize (3) - set/get stack size attribute in thread
       attribute...
       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 h...
```

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 ha...
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 mute...
pthread_mutexattr_getrobust_np (3) - get and set the robustness attribute of a m...
pthread_mutexattr_setpshared (3) - get/set process-shared mutex attribute
pthread_mutexattr_setrobust (3) - get and set the robustness attribute of a mute...
pthread_mutexattr_setrobust_np (3) - get and set the robustness attribute of a m...
pthread_rwlockattr_getkind_np (3) - set/get the read-write lock kind of the thre...
pthread_rwlockattr_setkind_np (3) - set/get the read-write lock kind of the thre...
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. *(0,15 ptos.)*

A través del comando `man man` accedemos al manual del manual, y en el apartado "Descripción" podemos encontrar una lista con las distintas secciones numeradas, se puede apreciar como la sección 2 corresponde a "Llamadas al sistema":

```
man man
       MAN(1)          Útiles de Páginas de Manual          MAN(1)

       NOMBRE
              man - una interfaz de los manuales de referencia electrónicos
       .
       .
       .
       DESCRIPCIÓN
              man es el paginador del manual  del  sistema.  Las  páginas  usadas
       como
                argumentos   al   ejecutar man suelen ser normalmente nombres de
       programas,
              útiles o funciones.  La página de manual associada con cada uno  de
       esos
               argumentos es buscada y presentada.  Si la llamada da también la
       sección,
                man buscará sólo en dicha sección del manual.  Normalmente,  la
       búsqueda
               se   lleva   a   cabo   en todas las secciones de manual disponibles
       según un
                 orden  predeterminado,  y  sólo  se  presenta  la   primera   página
       encontrada,
              incluso si esa página se encuentra en varias secciones.

              La  siguiente tabla muestra los números de sección del manual y los
       tipos de páginas que contienen.

       1   Programas ejecutables y guiones del intérprete de órdenes
       2  Llamadas del sistema (funciones servidas por el núcleo)
       3    Llamadas de la biblioteca (funciones contenidas  en  las  bibliotecas
       del sistema)
       4   Ficheros especiales (se encuentran generalmente en /dev)
       5   Formato de ficheros y convenios p.ej. I/etc/passwd
       6   Juegos
       7   Paquetes de macros y convenios p.ej. man(7), groff(7).
       8   Órdenes de admistración del sistema (generalmente solo son para root)
       9   Rutinas del núcleo [No es estándar]
       .
       .
       .
```

Utilizamos el comando `man 2 write` para acceder a la función de "write" asociada a las llamadas al sistema. Podemos ver que esta función tiene el siguiente formato:

`write(int fd, const void *buf, size_t count)`, la función write escribe `count` bytes del buffer empezando en el puntero `buf` al fichero `fd`.

```
man 2 write
       WRITE(2)        Linux Programmer's Manual       WRITE(2)

       NAME
           write - write to a file descriptor

       SYNOPSIS
           #include <unistd.h>

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

       DESCRIPTION
          write() writes up to count bytes from the buffer starting at buf to the
       file referred to by the file descriptor fd.

             The  number of bytes written may be less than count if, for example,
       there is insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)), or the call
       was interrupted by a signal handler after  having written less than count
       bytes.  (See also pipe(7).)
       .
       .
       .
```

<u>Ejercicio 2:</u> Comandos y Redireccionamiento. *(0,75 ptos.)*

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.** *(0,25 ptos.)*

El comando utilizado para realizar la actividad es el siguiente:

```
cat 'don quijote.txt' | grep -w molino >> aventuras.txt
```

El comando realiza las siguientes acciones, en primer lugar `cat 'don quijote.txt'` saca el fichero "don quijote.txt" por la salida estándar (la pantalla), a continuación con `| grep -w molino` utilizamos la salida generada (a través del pipeline) y busca aquellas líneas del fichero que contengan la palabra molino. Por último `>> aventuras.txt` escribe el resultado al final del fichero "aventuras.txt" sin sobreescribir lo que tuviese antes.

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.** *(0,25 ptos.)*

Utilizamos el siguiente comando:

```
ls -a | wc
```

Con el primer comando lo que hacemos es obtener todos los archivos, incluidos los ocultos y usando el pipeline contamos las líneas, palabra y letras en ese orden. La primera cifra nos dará el dato buscado.

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. *(0,25 ptos.)*

Hemos utilizado la siguiente línea de comando:

```
cat 'lista de la compra Elena.txt' 'lista de la compra Pepe.txt' 2> /dev/null | sort | uniq > 'num compra.txt'
```

En primer lugar, `cat 'lista de la compra Elena.txt' 'lista de la compra Pepe.txt'` concatena los ficheros "lista de la compra Elena.txt" y "lista de la compra Pepe.txt" en ese orden, si hay algún error, se redirige a /dev/null tal como queda especificado por `2> /dev/null`. A continuación se ordena y se seleccionan las líneas que no están repetidas, guardando el resultado en el fichero "num compra.txt"

d) 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}'. *(Opcional; 0,25 ptos)*

Para realizar esta tarea hemos utilizado el siguiente comando:

```
ps -A -L | awk '{print $1}' | uniq -c > hilos.txt
```

En primer lugar utilizamos `ps -A -L` para que muestre por pantalla la información de todos los procesos del sistema y los hilos del proceso. A continuación utilizamos `awk '{print $1}'` para extraer la primera columna del resultado (los identificadores de los procesos), si un proceso tiene más de un hilo el id del proceso aparece repetido. En tercer lugar utilizamos `uniq -c` que devuelve las líneas no repetidas y utilizamos -c para que devuelva el número de veces que se repite (que en este caso es el número de hilos que tiene cada proceso). Por último imprimimos el resultado en "hilos.txt", guardando así en un fichero el identificador del proceso junto al número de hilos que está ejecutando.

<u>Ejercicio 3:</u> Control de Errores. Escribir un programa que abra un fichero indicado por el primer parámetro en modo lectura usando la función fopen. En caso de error de apertura, el programa mostrará el mensaje de error correspondiente por pantalla usando perror. *(0,50 ptos.)*

Realizamos el programa indicado, que abra un fichero a través de un argumento de entrada e imprima un mensaje de error en caso de error de apertura.

a) ¿Qué mensaje se imprime al intentar abrir un fichero inexistente? ¿A qué valor de errno corresponde? *(0,15 ptos.)*

Para este caso pueden ocurrir dos mensajes de error.
Si no se introduce ningún argumento de entrada el error que se muestra por pantalla es:

```
: Bad address
```

Si por el contrario se introduce como argumento de entrada un fichero cuyo nombre no existe el mensaje que se muestra por pantalla:

```
: No such file or directory
```

El primer caso corresponde con el valor 14 de errno, el segundo caso corresponde al valor 2.

b) ¿Qué mensaje se imprime al intentar abrir el fichero /etc/shadow? ¿A qué valor de errno corresponde? *(0,15 ptos.)*

El mensaje de error que se imprime al abrir /etc/shadow es:

```
: Permission denied
```

Este caso corresponde con el valor 13 de errno.

c) Si se desea imprimir el valor de errno antes de la llamada a perror, ¿qué modificaciones se deberían realizar para garantizar que el mensaje de perror se corresponde con el error de fopen? *(0,20 ptos.)*

Habría que declarar una variable de tipo “int”, una vez se ejecute la función fopen, igualar la variable a errno. Justo antes de la llamada a perror habría que igualar errno al valor de la variable declarada, así este valor será el del error que produjo la función fopen.

<u>Ejercicio 4:</u> Espera Activa e Inactiva. *(0,25 ptos.)*

a) **Escribir un programa que realice una espera de 10 segundos usando la función clock en un bucle. Ejecutar en otra terminal el comando top. ¿Qué se observa?** *(0,15 ptos.)*

   Tras escribir el código con un bucle while que espera 10 segundos a través de clock(), se puede apreciar que en la terminal donde hemos ejecutado el comando top, el el porcentaje de CPU usada por el programa llega al 100%. Esto es debido a que con el bucle while el programa sigue ejecutándose hasta que pasan los 10 segundos, al contrario que con sleep y nanosleep.

b) **Reescribir el programa usando sleep y volver a ejecutar top. ¿Ha cambiado algo?** *(0,10 ptos.)*

   Tal y como cabía esperar, al ejecutar el nuevo programa que usa la función sleep, este queda en espera durante los 10 segundos sin consumir recursos del sistema y quedando en "segundo plano" hasta que vuelve a reactivarse.

<u>Ejercicio 5:</u> Finalización de Hilos. Dado el siguiente código en C, correspondiente al fichero "ejemplo_hilos.c": *(0,50 ptos.)*

a) **¿Qué hubiera pasado si el proceso no hubiera esperado a los hilos? Para probarlo basta eliminar las llamadas a pthread_join.** *(0,15 ptos.)*

   Si no se llama a pthread_join al ejecutarse el programa, la salida es la siguiente:

```
El programa ./Hilos termino correctamente
```

   Por lo que el programa no muestra la salida de cada uno de los hilos, que debería ser "Hola mundo" de manera intercalada y solo muestra el mensaje final.

b) **Con el código modificado del apartado anterior, indicar qué ocurre si se reemplaza la función exit por una llamada a pthread_exit.** *(0,10 ptos.)*

   Si se intercambia la sentencia exit(EXIT_SUCCESS) por pthread_exit(NULL) el programa se ejecuta en orden inverso de lo esperado, es decir, primero muestra por pantalla el siguiente mensaje:

```
El programa ./Hilos termino correctamente
```

   Y, a continuación, se muestra el mensaje "Hola mundo" de forma intercalada pes cada uno de los hilos se encarga por separado de imprimir letra a letra una de las palabras.

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 código que sería necesario añadir para que sea correcto no esperar a los hilos creados. *(0,25 ptos.)*

Habría que incluir la función pthread_detach() para cada uno de los hilos creados, para que se desliguen del hilo "principal" y así cada uno pueda ocuparse de una tarea sin tener que estar esperando al retorno del resto.

<u>Ejercicio 6:</u> Creación de Hilos y Paso de Parámetros. Escribir un programa en C ("ejercicio_hilos.c") que satisfaga los siguientes requisitos: *(1,00 ptos.)*

- **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.**

Creamos un programa que cumpla todos los requisitos y cuyo código es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

typedef struct NewStruct {
  int aleatnum;
  int x;
} NewStruct;

void* funcion(void *arg) {
  int* resultado;

  NewStruct* calculo=arg;
  sleep(calculo->aleatnum);
  resultado=malloc(sizeof(int));
  *resultado=(calculo->x)*(calculo->x)*(calculo->x);

  fprintf(stdout, "%d ", *resultado);

  free(resultado);
```

```
  return NULL;
}

int main(int argc, char** argv) {
  int error, i;
  int param=atoi(argv[1]);
  NewStruct* arg;
  pthread_t* h;

  srand(time(NULL));

  arg=(NewStruct*)malloc(param*sizeof(NewStruct));
  if (arg==NULL)
    return(EXIT_FAILURE);
  h=(pthread_t*)malloc(param*sizeof(pthread_t));
  if (h==NULL) {
    free(arg);
    return(EXIT_FAILURE);
  }

  for(i=0; i<param; i++) {
    arg[i].aleatnum=rand()%10;
    arg[i].x=i+1;
    error = pthread_create(h+i, NULL, funcion, &arg[i]);
       if(error != 0)
       {
              fprintf(stderr, "pthread_create: %s\n", strerror(error));
              exit(EXIT_FAILURE);
       }
  }

  for(i=0; i<param; i++) {
    error = pthread_join(*(h+i), NULL);
          if(error != 0) {
       fprintf(stderr, "pthread_join: %s\n", strerror(error));
       exit(EXIT_FAILURE);
     }
  }

  free(arg);
  free(h);
  exit(EXIT_SUCCESS);
}
```

<u>Ejercicio 7:</u> Creación de Procesos. Dado el siguiente código en C, correspondiente al fichero "ejemplo_fork.c": *(1,00 ptos.)*

a) Analizar el texto que imprime el programa. ¿Se puede saber a priori en qué orden se imprimirá el texto? ¿Por qué? *(0,10 ptos.)*

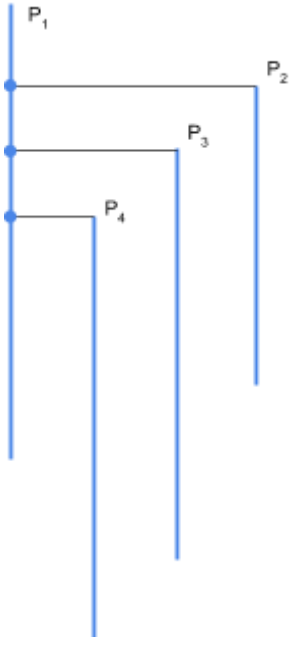
A priori no se puede saber el orden en el que se imprimirá el texto, la razón es que cuando se utiliza la función fork(), se realiza una llamada al sistema y, por tanto, es el planificador quien decide a quien dar prioridad a la hora de ejecutarse, por lo que un proceso puede acabar antes que otro siendo la impresión del programa incierta.

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. *(0,25 ptos.)*

Habrá que llevar a cabo una modificación únicamente en el 'if' que se ejecuta en el caso de que el proceso se el hijo. Usaremos las funciones getpid y getppid para obtener los datos requeridos e imprimirlos por pantalla.
Este sería el programa después de la modificación:

```
.
.
.
else if(pid ==  0)
        {   printf("PID hijo %d,PID padre %d\n", getpid(), getppid());
                        exit(EXIT_SUCCESS);
        }
.
.
.
```

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. *(0,25 ptos.)*



El dibujo anterior muestra lo que pasa con el código del programa, primero el proceso padre (P1) recorre un bucle de 3 iteraciones en las que crea 3 procesos hijos (que corresponderían con P2, P3, P4).
A continuación el proceso padre sale del bucle y se encuentra con un wait, por lo que tiene que esperar a que uno de los procesos termine para continuar, una vez

termina uno de los procesos hijo, el proceso padre puede continuar pudiéndose dar el caso de que éste acabe antes que el resto de los procesos hijo.

d) El código anterior deja procesos huérfanos, ¿por qué? *(0,15 ptos.)*

Este fenómeno se debe a que en nuestro programa tenemos un único 'wait' fuera del bucle por lo que solo esperamos que acabe el proceso uno de sus hijos, sin embargo tenemos tres. Por la tanto esos dos se quedarán zombies, pero como además nos salimos del main estos dos hijos restantes también se quedarán huérfanos porque su padre no ha esperado por ellos y ya ha finalizado su proceso.

e) Introducir el mínimo número de cambios en el código para que no deje procesos huérfanos. Copiar las modificaciones en la memoria y explicarlas. *(0,25 ptos.)*

El cambio que tendríamos que realizar para evitar esta situación es introducir un 'wait' por cada hijo. Así el padre esperaría a que acabaran todos sus procesos y ninguno se quedaría colgado.
Nuestro código solo se modifica en las últimas líneas las cuales serían:

```
        .
        .
        .
                        else if(pid >  0)
                {
                        printf("Padre %d\n", i);
                }
        }
        wait(NULL);
        wait(NULL);
        wait(NULL);
        exit(EXIT_SUCCESS);
}
```

<u>Ejercicio 8:</u> Árbol de Procesos. Escribir un programa en C ("ejercicio_arbol.c") que genere el siguiente árbol de procesos:
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. *(0,50 ptos.)*

Creamos un programa que cumpla todos los requisitos y cuyo código es el siguiente:

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

int main(int argc, char** argv) {
```

```
    int num_proc, i;
       pid_t pid;

    num_proc=atoi(argv[1]);

    for(i=0; i<num_proc-1; i++) {
              pid = fork();
              if(pid <  0) {
                     perror("fork");
                     exit(EXIT_FAILURE);
              }
              else if(pid ==  0 && i==num_proc-2) {
                     printf("Soy el último proceso\n");
              }
              else if(pid >  0) {
                     printf("Proceso %d\n", i+1);
            wait(NULL);
            exit(EXIT_SUCCESS);
               }
       }
       exit(EXIT_SUCCESS);
}
```

<u>Ejercicio 9:</u> Espacio de Memoria. Dado el siguiente código en C, correspondiente al fichero "ejemplo_malloc.c": *(0,50 ptos.)*

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? *(0,25 ptos.)*

Al ejecutar el código, el resultado que da por pantalla el siguiente programa es:

```
Padre:
```

Esto se debe a que, en primer lugar, se asigna el espacio de memoria para guardar la palabra "Hello". A continuación se ejecuta la función fork(), por lo que se hace una copia de los recursos del proceso padre al proceso hijo (incluyendo la memoria reservada), en otra zona de memoria.
El proceso hijo ejecuta la función strcpy(sentence, MESSAGE), guardando el mensaje "Hello" en la parte de su memoria que tenía reservada, pero no en la de su padre, puesto que son distintas.
Por eso, cuando el padre imprime el mensaje guardado en su zona de memoria, no hay nada.

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? *(0,25 ptos.)*

La memoria se ha liberar en ambos procesos, puesto que cuando se ejecuta fork() se hace una copia de los recursos (incluyendo la memoria reservada) en ambos procesos, siendo ahora distinta. Por lo que habría que liberar la memoria en todos los procesos ejecutados.
El programa liberando correctamente la memoria en todos los procesos es el siguiente:

```
int main(void)
{
	pid_t pid;
	char * sentence = calloc(sizeof(MESSAGE), 1);

	pid = fork();
	if (pid < 0)
	{
		perror("fork");
		free(sentence);
		exit(EXIT_FAILURE);
	}
	else if (pid == 0)
	{
		strcpy(sentence, MESSAGE);
		free(sentence);
		exit(EXIT_SUCCESS);
	}
	else
	{
		wait(NULL);
		printf("Padre: %s\n", sentence);
		free(sentence);
		exit(EXIT_SUCCESS);
	}
}
```

<u>Ejercicio 10:</u> Shell. Escribir un programa en C (“ejercicio_shell.c”) que implemente una shell sencilla (sin redirecciones ni estructuras de control). *(1,50 ptos.)*

a) Escribir el programa, satisfaciendo los siguientes requisitos: *(1,00 ptos.)*

- **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 o Terminated by signal , en función de cómo terminó el hijo.**
- **A continuación se realizará la siguiente iteración del bucle, leyendo el siguiente comando.**

Escribimos un programa cumpliendo los requisitos establecidos, cuyo código es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wordexp.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv) {
    int status=0;
    char* command=NULL;
    int tam;
    size_t size=0;
    wordexp_t p;

    while((tam=getline(&command, &size, stdin))!=EOF) {
        command[tam-1]='\0';
        if(fork()) {
            wait(&status);
            if (WIFEXITED(status)) {
                fprintf(stderr, "\nExited with value %d\n\n",
WEXITSTATUS(status));
            }
            if (WIFSIGNALED(status)) {
                fprintf(stderr, "\nTerminated by signal %d\n\n",
WTERMSIG(status));
            }
        }
        else {
            fprintf(stdout, "\nExecuting new process...\n\n");
            wordexp(command, &p, 0);
            execvp(p.we_wordv[0], p.we_wordv);
            wordfree(&p);
            exit(EXIT_FAILURE);
        }
    }
    free(command);
    exit(EXIT_SUCCESS);
}
```

El programa tiene un bucle general que recoge la línea introducida como input (y sale de él cuando se lee un EOF).
Para que el último de los argumentos se pase correctamente, sustituimos el caracter \n por \0.

A continuación se realiza un fork() y el padre espera a que termine el hijo, recogiendo el status del mismo e imprimiendo por pantalla el valor de regreso o la señal de salida en función de la ejecución del programa.
El hijo guarda en la variable p (de tipo wordexp_t) el directorio del programa junto con los argumentos para ejecutarse y llama a la función execvp, pasando como primer argumento el el nombre del programa (si está en el directorio PATH) (o el directorio completo si no se encuentra en este directorio) y como segundo un puntero a una tabla con los argumentos que va a utilizar el programa para ejecutarse.
Antes del retorno al proceso padre, se llama a la función exit(EXIT_FAILURE) del hijo en el caso de que salga de la función execvp para indicar al padre un fallo en la ejecución.
Por último, se libera la memoria de command y se termina el programa.

b) Explicar qué función de la familia exec se ha usado y por qué. ¿Podría haberse usado otra? ¿Por qué? *(0,25 ptos.)*

Se ha utilizado la función execvp debido a que esta recibe como argumento el nombre el nombre del programa (que va buscando en la variable PATH del proceso), o el directorio del programa si no se encuentra en esta variable, y como segundo argumento un array con lo argumentos de entrada del proceso. Podría haberse utilizado otra función pero la implementación habría sido demasiado complicada e innecesario.
Hemos pensado que es la mejor de las funciones a usar debido a que otras nos limitan su uso a la hora de implementar el programa.
Por ejemplo, en las funciones execl, execlp y execle se pasan individualmente cada uno de los argumentos, sin embargo no sabemos a priori el número de argumentos que tendrá el comando. En las funciones execv y execl, hay que meter como primer argumento el directorio completo en el que se encuentra el programa que se quiere ejecutar.

c) Ejecutar con la shell implementada el comando sh -c inexistente. ¿Qué imprime? *(0,10 ptos.)*

Ejecutamos el programa e introducimos la siguiente línea para que ejecute el comando:

```
sh: 0: -c requires an argument

Exited with value 2
```

d) Hacer un programa en C que finalice llamando a abort y ejecutarlo con la shell implementada. ¿Qué se imprime ahora? *(0,15 ptos.)*

Implementamos un programa que termine los procesos con la función abort(), cuando lo intentamos ejecutar con la shell, termina por una señal (que recogemos en la variable status) y muestra el siguiente resultado por pantalla:

```
Terminated by signal 6
```

e) 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. *(Opcional; 0,50 ptos.)*

Escribimos un programa cumpliendo los requisitos establecidos, cuyo código es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <spawn.h>
#include <wordexp.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv) {
    int status=0;
    char* command=NULL;
    int tam;
    size_t size=0;
    wordexp_t p;
    pid_t pid;

    while((tam=getline(&command, &size, stdin))!=EOF) {
        command[tam-1]='\0';

        wordexp(command, &p, 0);
        status=posix_spawnp(&pid, p.we_wordv[0], NULL, NULL, p.we_wordv, NULL);

        if(status == 0) {
            wait(&status);
            if (WIFEXITED(status)) {
                fprintf(stderr, "\nExited with value %d\n\n", WEXITSTATUS(status));
            }
            if (WIFSIGNALED(status)) {
                fprintf(stderr, "\nTerminated by signal %d\n\n", WTERMSIG(status));
            }
        }
    }
    free(command);
    exit(EXIT_SUCCESS);
}
```

El programa es muy parecido al original, con la diferencia que tras realizar la división del comando en ejecutable y argumentos utilizamos la función posix_spawnp que realiza la tarea de fork y execvp a la vez, hemos utilizado posix_spawnp y no posix_spawn porque busca el nombre del fichero en la variable de entorno PATH del proceso (al igual que execvp), a esta función le pasamos como primer argumento un puntero a identificador de proceso, como segundo argumento el nombre del

ejecutable y como quinto argumento un array de argumentos con las que ejecutar el programa, el resto de argumentos están a NULL puesto que no los utilizamos.
El resto del código es exactamente igual que el programa original.

<u>Ejercicio 11:</u> 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'. *(0,50 ptos.)*

Primero ejecutamos un proceso y vemos que id tiene, en nuestro caso tomamos el proceso cuyo id es el 1513, que es la shell (el proceso bash) que estamos ejecutando. Vemos la lista de directorios que contiene con ls -l.

a) **El nombre del ejecutable.** *(0,10 ptos.)*

   Una vez vemos los directorios podemos observar que el ejecutable se encuentra dentro de exe y su nombre es: `/usr/bin/gnome-shell`.

b) **El directorio actual del proceso.** *(0,10 ptos.)*

   Lo podemos encontrar en el enlace simbólico cwd que significa 'current working directory' y el directorio de nuestro proceso es: `/home/elena`.

c) **La línea de comandos que se usó para lanzarlo.** *(0,10 ptos.)*

   Para buscar la línea de comandos que se utilizó ejecutamos: `cat cmdline | tr '\0' '\n'` buscando así el command line y cambiando los \0 por \n para poderlo leer con mayor facilidad y el resultado que buscamos es el siguiente: `/usr/bin/gnome-shell`.

d) **El valor de la variable de entorno LANG.** *(0,10 ptos.)*

   Para encontrar el valor de la variable ejecutamos: `cat environ | tr '\0' '\n' | grep LANG` y el resultado obtenido es: `LANG=es_ES.UTF-8`

e) **La lista de hilos del proceso.** *(0,10 ptos.)*

   Para ver la lista de hilos del proceso utilizamos el comando `cd task` que nos muestra la siguiente lista de procesos:

```
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1513
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1515
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1516
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1517
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1538
```

```
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1562
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1565
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1566
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1567
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1568
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1569
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1570
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1571
dr-xr-xr-x 7 elena elena 0 feb 15 12:01 1572
```

<u>Ejercicio 12:</u> Visualización de Descriptores de Fichero. Dado el siguiente código en C, correspondiente al fichero "ejemplo_descriptores.c":
El programa se para en ciertos momentos para esperar a que el usuario pulse ENTER. 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. *(0,50 ptos.)*

a) **Stop 1. Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos? ¿A qué tipo de fichero apuntan?** *(0,10 ptos.)*

   Al ejecutar el programa observamos su pid (que en este caso resulta ser 7350) y ejecutamos la instrucción: ls -l /proc/7350/fd y así podemos ver la información deseada:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
```

   Como podemos observar los descriptores de ficheros que se encuentran abiertos son el 0, el 1 y el 2 que corresponden respectivamente a la Entrada estándar, la Salida estándar y Salida de errores.

b) **Stop 2 y Stop 3. ¿Qué cambios se han producido en la tabla de descriptores de fichero?** *(0,10 ptos.)*

   Para Stop 2 el descriptor de fichero es el siguiente:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:16 3 ->
/home/rubgarfue/Descargas/Práctica1/file1.txt
```

   Podemos ver que, al abrirse el fichero file1.txt, aparece un nuevo fichero abierto con el número 3 por el proceso.

   Para Stop 3 el descriptor de fichero es el siguiente:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:16 3 ->
/home/rubgarfue/Descargas/Práctica1/file1.txt
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:17 4 ->
/home/rubgarfue/Descargas/Práctica1/file2.txt
```

Tras producirse el Stop 4, se abre el fichero file2.txt cuyo identificador dentro del descriptor de fichero es 4.

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? *(0,20 ptos.)*

Para Stop 4:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:16 3 ->
'/home/rubgarfue/Descargas/Práctica1/file1.txt (deleted)'
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:17 4 ->
/home/rubgarfue/Descargas/Práctica1/file2.txt
```

Tras borrarse el fichero file1.txt e intentar acceder a él por el directorio `/home/rubgarfue/Descargas/Práctica1/file1.txt` se nos indica que el fichero que se intenta abrir no existe, por lo que ha sido borrado del disco.
Sin embargo se puede acceder a él a través del directorio /proc, en concreto el fichero sigue almacenado en /proc/7350/fd/3, por lo que podemos guardar los datos de este fichero antes de que desaparezca completamente. Por ejemplo si queremos guardar los datos en el fichero “datos.txt”, podemos introducir por la terminal el siguiente comando: `cat /proc/7350/fd/3 > datos.txt`.

d) Stop 5, Stop 6 y Stop 7. ¿Qué cambios se han producido en la tabla de descriptores de fichero? ¿Que se puede deducir sobre la numeración de un descriptor de fichero obtenido tras una llamada a open? *(0,10 ptos.)*

Para Stop 5:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:17 4 ->
/home/rubgarfue/Descargas/Práctica1/file2.txt
```

Una vez se ha terminado de eliminar por completo los datos del fichero file1.txt, podemos ver en el descriptor de ficheros que ya no existe, pero no realiza una reasignación de identificadores de ficheros.

Para Stop 6:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:16 3 ->
/home/rubgarfue/Descargas/Práctica1/file3.txt
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:17 4 ->
/home/rubgarfue/Descargas/Práctica1/file2.txt
```

Al abrir un nuevo fichero (file2.txt), el descriptor de fichero asigna como identificador el primero número que no identifica a ninguno, en este caso es 3.

Para Stop 7:

```
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 0 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 1 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:07 2 -> /dev/pts/0
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:16 3 ->
/home/rubgarfue/Descargas/Práctica1/file3.txt
lrwx------ 1 rubgarfue rubgarfue 64 feb 15 13:17 4 ->
/home/rubgarfue/Descargas/Práctica1/file2.txt
lr-x------ 1 rubgarfue rubgarfue 64 feb 15 13:21 5 ->
/home/rubgarfue/Descargas/Práctica1/file3.txt
```

Por último abre el fichero y, tal y como se esperaba, le asigna el identificador 5 pues es el primero de los identificadores que no está asignado.

Podemos deducir que cuando se ejecuta un proceso, automáticamente se abren los ficheros 0, 1 y 2 que corresponden con la Entrada estándar, la Salida estándar y la Salida de errores y que, a partir de ahí, todos los ficheros que se abren recibirán como identificador el primer número en orden ascendente que no está asignado a ninguno de los identificador.
Por eso cuando se cierra un fichero deja su identificador ‘libre’ por lo que cuando se abra un nuevo proceso buscará entre los identificadores que estén sin usar el menor de ellos y se lo asignará.

<u>Ejercicio 13:</u> Problemas con el Buffer. Dado el siguiente código en C, correspondiente al fichero "ejemplo_buffer.c": *(0,50 ptos.)*

a) ¿Cuantas veces se escribe el mensaje "Yo soy tu padre" por pantalla? ¿Por qué? *(0,10 ptos.)*

El mensaje "Yo soy tu padre" se escribe dos veces por pantalla. Esto se debe a que, al hacer el printf del mensaje, no se escribe directamente en stdout, si no en el buffer del proceso asociado al descriptor de fichero 1.
Cuando se ejecuta fork(), se crea otro proceso y se copia toda la información del proceso padre al proceso hijo, incluyendo el buffer en el que está almacenado el mensaje "Yo soy tu padre", por eso cuando ambos procesos vacían el buffer, el mensaje aparece dos veces por pantalla.

b) En el programa falta el terminador de línea (\n) en los mensajes. Corregir este problema. ¿Sigue ocurriendo lo mismo? ¿Por qué? *(0,15 ptos.)*

Al incluir el \n lo que conseguimos es que se imprima lo que contiene el buffer en el descriptor de ficheros. Por lo tanto se imprime el mensaje de "yo soy tu padre" y el buffer queda vacío. Al hacer el fork() tanto el buffer del padre como el del hijo están vacíos. En el caso de hijo también se escribirá "Noooooo" debido a que hemos incluido un \n en este último mensaje y el buffer del hijo se vaciará.

c) Ejecutar el programa redirigiendo la salida a un fichero. ¿Qué ocurre ahora? ¿Por qué? *(0,10 ptos.)*

Al redirigir la salida del programa a un fichero externo con los cambios ya solucionados (el \n al final de cada mensaje) y abrirlo nos encontramos con lo siguiente:

```
Yo soy tu padre
Noooooo
Yo soy tu padre
```

Esto se debe a que el carácter \n sólo vacía el buffer cuando se está escribiendo en la terminal, pero al estar escribiendo en un fichero externo el buffer no se vacía y tras ejecutar el fork() ocurre el mismo problema con el que nos encontrábamos en el apartado a).

d) Indicar en la memoria como se puede corregir definitivamente este problema sin dejar de usar printf. *(0,15 ptos.)*

Si justo después de la función `printf("Yo soy tu padre\n");` incluimos la sentencia `fflush(stdout);` obligamos al buffer a vaciarse y, por tanto, a escribirse sea cual sea el fichero de salida.
Al introducir los cambios y ejecutar el programa redirigiendo la salida a un fichero externo comprobamos que el problema se soluciona.

<u>Ejercicio 14:</u> Ejemplo de Tuberías. Dado el siguiente código en C, correspondiente al fichero "ejemplo_pipe.c": *(0,25 ptos.)*

a) Ejecutar el código. ¿Qué se imprime por pantalla? *(0,10 ptos.)*

Al ejecutar el programa se imprime por la terminal el siguiente mensaje:

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

El programa ejecuta la función pipe(), que crea una tubería entre los procesos padre e hijo. A continuación se cierra el extremo de escritura y el extremo de lectura en el padre y el hijo respectivamente. El hijo escribe el mensaje en el pipe y, a continuación, el padre lo lee y lo escribe por pantalla

b) ¿Qué ocurre si el proceso padre no cierra el extremo de escritura? ¿Por qué? *(0,15 ptos.)*

El proceso padre al no tener cerrado el proceso de escritura no es capaz de reconocer cuando la escritura ha finalizado, pues sigue habiendo escritores en esa tubería (que es precisamente el propio padre).

c) 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? *(Opcional: 0,25 ptos.)*

Cuando introducimos las modificaciones indicadas, el programa no devuelve ninguna salida por pantalla.
Esto se debe a que el hijo antes de escribir debe esperar un segundo, en este tiempo el padre acaba el proceso ya que como hemos borrado el 'wait' no espera a su hijo y finaliza dejándolo huérfano.
Cuando termina la espera, el hijo intenta escribir en el descriptor de fichero, pero debido a que no hay lectores en la tubería (pues el padre ha finalizado), el Sistema Operativo manda la señal SIGPIPE que finalizará el proceso hijo antes de que se imprima por pantalla la frase "He escrito en el pipe\n".

<u>Ejercicio 15:</u> 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 padre a 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 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. *(1,50 ptos.)*

Escribimos un programa cumpliendo los requisitos establecidos, cuyo código es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

int main() {
    int fd1[2];
    int fd2[2];
    int aleatnum;
    pid_t pid;
    ssize_t nbytes;
    char buffer[80];
    int filedesc;

    srand(time(NULL));
       if(pipe(fd1) == -1)
       {
              perror("pipe 1");
              exit(EXIT_FAILURE);
       }
       if(pipe(fd2) == -1)
       {
              perror("pipe 2");
              exit(EXIT_FAILURE);
       }

    pid=fork();
    if (pid == -1) {
        perror("fork 1");
        exit(EXIT_FAILURE);
    }

    if(pid == 0) {
        close(fd1[0]);

        aleatnum=rand()%11;
        sprintf(buffer, "%d", aleatnum);

        nbytes=write(fd1[1], buffer, strlen(buffer)+1);
        if (nbytes == -1) {
            perror("write 1");
            exit(EXIT_FAILURE);
```

```
        }
        fprintf(stdout, "He escrito el número %d\n", aleatnum);

        exit(EXIT_SUCCESS);
    }
    else {
        pid=fork();
        if (pid == -1) {
            perror("fork 2");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) {
            close(fd2[1]);

            filedesc=open("numero_leido.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR |
S_IWUSR | S_IRGRP | S_IWGRP);
            if (filedesc == -1) {
                perror("open");
                exit(EXIT_FAILURE);
            }

            nbytes=read(fd2[0], buffer, sizeof(buffer));
            if (nbytes == -1) {
                perror("read 2");
                exit(EXIT_FAILURE);
            }

            dprintf(filedesc, "%s", buffer);

            close(filedesc);

            exit(EXIT_SUCCESS);
        }
        else {
            close(fd1[1]);
            close(fd2[0]);

            nbytes=read(fd1[0], buffer, sizeof(buffer));
            if (nbytes == -1) {
                perror("read 1");
                exit(EXIT_FAILURE);
            }

            nbytes=write(fd2[1], buffer, strlen(buffer)+1);
            if (nbytes == -1) {
                perror("write 2");
                exit(EXIT_FAILURE);
            }

            wait(NULL);
            wait(NULL);
        }
    }
    exit(EXIT_SUCCESS);
}
```

En el programa creamos dos tuberías y, a continuación, realizamos dos fork() en el bucle padre para que éste cree dos hijos. Lo primero que hacemos en cada uno de los procesos (padre, hijo 1, hijo 2) es cerrar los descriptores de ficheros de las tuberías que no vamos a utilizar y a continuación realizamos cada una de las tareas que deben hacer cada proceso por separado:
El hijo 1 generará un número aleatorio (entre 0 y 10), lo escribirá en la tubería e imprimirá por pantalla el valor del número aleatorio generado.
El padre leerá de la tubería el número aleatorio y lo escribirá en la segunda tubería para que pueda leerlo el segundo hijo.
El hijo 2 creará un nuevo fichero llamado “numero_leido.txt”, leerá el número aleatorio de la tubería y lo escribirá en el fichero.
Todos los procesos realizan comprobación de errores y liberan toda la memoria utilizada.