Práctica 1

FECHA DE ENTREGA: DEL 21 DE FEBRERO AL 27 DE FEBRERO
(HORA LÍMITE: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

SEMANA 1: INTRODUCCIÓN A LA SHELL. HILOS.
Introducción a la Shell
Programación en C con POSIX

SEMANA 2: PROCESOS Y EJECUCIÓN DE PROGRAMAS.
Procesos
Ejecución de Programas

SEMANA 3: FICHEROS Y TUBERÍAS.
El Directorio `/proc`
Ficheros
Tuberías (*Pipes*)

SEMANA 1: INTRODUCCIÓN A LA SHELL. HILOS.

Introducción a la Shell

Una **shell** es una interfaz de usuario que permite usar los recursos del Sistema Operativo. De esta forma, aunque se pueden entender los entornos gráficos (como Windows, Mac OSX, Gnome, KDE, etc.) como shells, normalmente se usa el término shell para referirse a los Intérpretes de Líneas de Comandos (CLI).

Los sistemas Unix, como Linux o Mac OSX, disponen de diversas shells con distintas capacidades, sintaxis y comandos. Quizá la más común y extendida es la Bourne-Again Shell (o bash) desarrollada en 1989 por Brian Fox para el GNU Project, que hereda las características básicas y sintaxis de la Bourne Shell (o sh) desarrollada por Stephen Bourne en los Bell Labs en 1977, y que ha dado lugar a la familia más extensa de shells, la "Bourne shell compatible": `sh`, `bash`, `ash`, `dash`, `ksh`, etc. La siguiente familia más común de shells en Unix son las "C shell compatible" como `csh` o `tcsh`.

Las shells modernas son concebidas como intérpretes de comandos interactivos y como lenguajes de *scripting*. Las diferencias de sintaxis entre las distintas shells se encuentran fundamentalmente en las estructuras de control. Las "C compatible" tienen, por ejemplo, una sintaxis más parecida a C, no siendo tampoco muy difícil adaptarse a la sintaxis de la familia Bourne, más extendida, como se puede ver en el siguiente ejemplo:

```
#!/usr/bin/env sh
if [ $days -gt 365 ] then
    echo This is over a year
fi
```

```
#!/usr/bin/env csh
if ( $days > 365 ) then
    echo This is over a year
endif
```

La mayoría de comandos que permite ejecutar la shell no son comandos internos implementados en la propia shell, sino que son programas escritos en cualquier lenguaje de programación, como C o Python. Estos programas tienen en común que pueden comunicarse a través de 3 canales distintos:

0. Entrada estándar o `stdin`.
1. Salida estándar o `stdout`.
2. Salida de errores estándar o `stderr`.

Los programas que sigan la filosofía de Unix, salvo que se les indique lo contrario, deben leer su entrada (si la hay) de la entrada estándar, y escribir el resultado (de haberlo) por la salida estándar en formato de texto, de forma que sea comprensible por otros programas. La salida de errores se utiliza para comunicar mensajes al usuario humano, como mensajes de error, avisos, o mensajes para interactuar con el usuario. Un programa Unix que no tenga salida y se ejecute sin errores no escribirá nada por la salida estándar.

Seguir esta filosofía, junto con la idea de Unix de que "todo es un fichero", permite aplicar redirecciones al ejecutar programas, de modo que se puedan reutilizar programas de maneras que no fueron previstas por el autor original. Por ejemplo, se puede hacer que un programa lea de un fichero en lugar de la terminal, o que imprima el texto por la impresora en lugar de por pantalla, sin necesidad de cambiar el programa original. Los siguientes símbolos permiten establecer redirecciones al ejecutar un comando:

- `<` Redirige la entrada a un fichero. Por ejemplo, `<comando> < <fichero>` hace que `<comando>` reciba su entrada de `<fichero>` en lugar de la terminal.
- `>` Redirige la salida a un fichero. Si el fichero contenía algo se borrará.
- `2>` Redirige la salida de errores a un fichero. Si el fichero contenía algo se borrará.
- `>>` Redirige la salida a un fichero, manteniendo el contenido del fichero y añadiendo al final.
- `|` Tubería (en inglés, *pipe*). Redirige la salida del comando a la izquierda de la tubería a la entrada del comando a la derecha de la tubería. Así, `<comando1> | <comando2>` hace que `<comando2>` tenga como entrada la salida de `<comando1>`, permitiendo crear utilidades nuevas a base de encadenar utilidades ya existentes. Esto da pie a otra idea de la filosofía Unix: cada programa debe hacer una única cosa pero hacerla correctamente, para poder enlazarlo con otros de esta manera. Una serie de comandos comunicados entre sí mediante tuberías se denomina un *pipeline*.

A continuación se explican varios ejemplos de comandos útiles.

Comandos de Ayuda

- `man` - manual: Posiblemente el comando más útil para principiantes (y no tan principiantes). Precediendo a cualquier otro comando, abre el manual de uso del mismo, en el que se detallan el objetivo del comando y las opciones y parámetros de los que dispone. Por tanto, este será uno de los comandos más utilizados durante la práctica. Como todo buen comando, `man` también tiene su manual, se puede acceder a él mediante `man man`.
  - Una opción de gran utilidad de `man` es la de buscar una cierta palabra a lo largo de todo el manual. Para ello basta hacer `man -k <palabra_clave>`.
  - Cada página del manual está referida por un nombre y un número entre paréntesis (la sección). Para leer la página con un determinado nombre `<comando>` e incluida en la sección `<n_sec>`, se usaría: `man <n_sec> <comando>`. Así, `man passwd` muestra la ayuda para la utilidad `passwd` de Linux. Por el contrario, para conseguir la ayuda referida al fichero `/etc/passwd`, se usaría `man 5 passwd`.

0,25 ptos.
0,10 ptos.

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`".

0,15 ptos. 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.

Comandos de Navegación

- `cd` - change directory: Comando interno de la shell que permite cambiar el directorio que se considera como directorio actual. Todas las rutas relativas (aquellas que no comiencen por "`/`") se consideran partiendo del directorio actual.
  - `cd <dir>`: Lleva al directorio `<dir>`.
  - `cd /`: Lleva al directorio raíz.
  - `cd ~`: Lleva al directorio del usuario.
  - `cd ..`: Lleva al directorio que sea padre del directorio actual.
  - `cd -`: Lleva al último directorio visitado.
- `ls` - list: Muestra una lista con los ficheros de un directorio.
  - `ls`: Muestra los ficheros del directorio actual.
  - `ls <dir>`: Muestra los ficheros del directorio `<dir>`.
  - `ls -l`: Muestra información extendida de los ficheros.
  - `ls -a`: Muestra los ficheros que comienzan por "`.`". Por defecto estos ficheros no se muestran y se consideran ocultos.

Comandos de Manejo de Texto

- `cat` - concatenate: Concatena los ficheros que se le pasen y muestra su contenido por pantalla.
- `head`: Muestra las 10 primeras líneas de un fichero (por defecto, la entrada estándar). Para leer las primeras `n` líneas de un fichero, basta utilizar `head -n <fichero>`.
- `tail`: El análogo de `head` pero esta vez mostrando las últimas líneas.
- `more` y `less`: Muestran el contenido de un fichero (o de la entrada estándar) paginado. `more` es el comando que especifica POSIX, mientras que `less` es la versión de Linux, más avanzada.
- `grep` - global regular expression print: Potente comando para realizar búsquedas de patrones de texto. Se puede usar para buscar palabras concretas, pero su verdadera potencia viene cuando se usan expresiones regulares, un lenguaje de dominio específico para describir patrones de texto (y en general, para describir cualquier autómata finito).
- `wc` - word count: Cuenta las letras, palabras y líneas de un fichero.
- `sort`: Ordena las líneas de un fichero de texto. Mediante parámetros se puede especificar ordenación numérica o alfanumérica, y por qué columna se quieren ordenar las líneas, entre otras cosas.
- `uniq`: Por defecto, devuelve las líneas no repetidas de un fichero. Solo filtra las líneas repetidas consecutivas con lo que para filtrar todas las líneas repetidas es necesario pasar las líneas ordenadas. Con la opción `-c` cuenta además el número de ocurrencias repetidas de cada línea.

Comandos de Gestión de Procesos

- `top`: Muestra de forma dinámica información sobre los procesos en ejecución (CPU que consumen, memoria, etc.).
- `ps` - process status: Muestra información de los procesos del sistema.
  - `ps -A`: Muestra todos los procesos del sistema.
  - `ps -u <usuario>`: Muestra los procesos del usuario `<usuario>`.
  - `ps -fl`: Muestra información adicional de los procesos.
  - `ps -L`: Muestra hilos en lugar de procesos.
- `pstree`: Muestra la jerarquía de procesos en árbol. Al usarlo con `pstree <PID>` se usa como raíz del árbol el proceso con identificador `<PID>`.

0,75 ptos. **Ejercicio 2: Comandos y Redireccionamiento.**

0,25 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. 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. 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. 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}'`.

Programación en C con POSIX

Control de Errores

Aunque algunas funciones de C y POSIX (Portable Operating System Interface X) no pueden fallar (como `strlen`), la mayoría de ellas pueden retornar con error bajo alguna circunstancia. Normalmente las funciones que presentan errores devuelven como resultado algún valor que sea claramente distinguible de un resultado válido, como `-1` en caso de que se espere un entero positivo (esta es la razón por la que `fgetc` devuelve `int` y no `char`: `EOF`, el retorno en caso de error o fin de fichero, no es un carácter válido). El valor concreto devuelto en caso de error se puede consultar en el apartado "RETURN VALUE" del manual de la función correspondiente. En esta asignatura (y en general, en la programación en C) es importante comprobar el valor de retorno para asegurarse de que no se produce ningún error.

Además de detectar que una función ha devuelto un error, en ocasiones es útil diferenciar entre distintos tipos de errores, ya sea para mostrar un mensaje de error apropiado o para intentar solucionar la causa del error. Muchas de las funciones de C y POSIX guardan el tipo de error que se ha producido en la variable global `errno`, de tipo `int`. En la página de manual de cada función se indica si usa `errno` o no. Además, para cada función se listan los tipos de errores

más habituales en el apartado "ERRORS" del manual de dicha función, que se corresponden con identificadores declarados en `errno.h`.

Si se quiere obtener un mensaje de error apropiado para el tipo de error producido, se pueden usar las funciones de la familia `strerror`. También se puede imprimir el mensaje directamente usando la función `perror`. Es importante recalcar que la ejecución de cualquier otra función de C podría alterar el valor de `errno` incluso aunque se ejecute sin errores, así que será necesario copiar el valor de `errno` a otra variable de tipo `int` si se desea tratar el error en otro punto del programa.

0,50 ptos.

Ejercicio 3: 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`.

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

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

0,20 ptos. 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`?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Espera Inactiva

Las funciones `sleep` y `nanosleep` permiten esperar un tiempo sin consumir recursos de CPU. Para ello, el planificador de tareas del Sistema Operativo pasa a ejecutar otros hilos, despertando al hilo que llamó a estas funciones cuando haya pasado el tiempo indicado.

Nota. *El planificador no tiene por qué despertar al hilo inmediatamente pasado el tiempo indicado, sino que podría pasar algo más de tiempo, normalmente del orden de milisegundos.*

0,25 ptos.

Ejercicio 4: Espera Activa e Inactiva.

0,15 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?

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

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Hilos

Existen ocasiones en las que se desea diseñar software capaz de ejecutar distintas partes simultáneamente. Por ejemplo, que la interfaz gráfica siga respondiendo a acciones del usuario mientras la aplicación realiza los cálculos que se hayan solicitado, o que se divida el cálculo del producto de matrices de gran tamaño para que cada fila se compute por separado y se realice el cómputo más rápidamente.

Una forma de realizar esta ejecución simultánea de tareas dentro de un mismo programa es mediante el uso de **hilos**. Los hilos son unidades de trabajo de cuya ejecución es responsable

el Sistema Operativo. Normalmente el Sistema Operativo intentará mantener todas las CPUs de la máquina ocupadas ejecutando un hilo en cada una de ellas (no necesariamente del mismo programa). También se encargará de alternar la ejecución de los hilos en el caso de que haya más hilos que CPUs disponibles, de modo que todos los hilos se ejecuten regularmente.

Los hilos creados dentro del mismo programa tienen acceso a todos los recursos del mismo. Esto quiere decir que los hilos verán la misma memoria y los mismos ficheros abiertos, entre otras cosas. Aunque por un lado esto permite una comunicación directa y rápida entre hilos, también requiere un mayor cuidado por parte del programador para evitar que los hilos manipulen recursos que otro hilo pueda estar usando.

Para escribir programas multihilo en C se puede hacer uso de la biblioteca de hilos `pthread` (POSIX threads) que implementa el estándar POSIX. Para ello el programa deberá incluir la cabecera correspondiente (`#include <pthread.h>`) y a la hora de compilar es necesario enlazar el programa con la biblioteca de hilos, añadiendo a la llamada al compilador `gcc` el parámetro `-lpthread` (o `-pthread` según la plataforma).

Al comienzo del programa, éste tendrá un único **hilo principal** que ejecutará la función `main`. Cuando esta función retorne, o se llame a una función de la familia `exit`, o el programa finalice como resultado de una señal del Sistema Operativo (por ejemplo, por una violación de segmento), todos los hilos del programa finalizarán.

Se pueden crear hilos para ejecutar una función concreta con la función `pthread_create`. Esta función permite establecer los atributos del hilo y pasar un argumento a la función a ejecutar mediante un puntero. La función `pthread_create` devuelve en su primer argumento un identificador del hilo creado. Además, cada hilo puede acceder a su propio identificador llamando a `pthread_self`. Cada hilo adicional constará con su propia pila local (*stack*), donde se crearán las variables automáticas, de un tamaño fijo que puede especificarse al crear el hilo. El hilo finalizará al retornar de la función, o al ejecutar `pthread_exit`.

Por defecto, los hilos creados con `pthread_create` devuelven un puntero como resultado. Cualquier otro hilo del programa puede llamar a la función `pthread_join` para esperar a que un hilo concreto termine y recuperar el valor de retorno. Un programa correcto debería llamar a `pthread_join` una vez por cada hilo creado de esta manera. También es posible crear hilos "desligados" que no puedan ser esperados y no retornen ningún valor a otro hilo. Para ello, un hilo puede desligarse llamando a `pthread_detach` con el identificador del hilo. También es posible especificar en los atributos del hilo, al llamar a `pthread_create`, que el hilo debe ser desligado.

Nota. *Las funciones de la biblioteca de hilos retornan directamente el valor de error en lugar de usar* `errno`*. En los programas multihilo,* `errno` *no es realmente una variable global, sino que cada hilo tiene su propio* `errno` *(es* thread-*local).*

0,50 ptos.

Ejercicio 5: Finalización de Hilos. Dado el siguiente código en C, correspondiente al fichero "`ejemplo_hilos.c`":

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

void * slow_printf(void * arg)
{
    const char * msg = arg;

```

```c
    for(size_t i = 0; i < strlen(msg); i++)
    {
        printf("%c ", msg[i]);
        fflush(stdout);
        sleep(1);
    }

    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t h1;
    pthread_t h2;
    char * hola = "Hola ";
    char * mundo = "Mundo";
    int error;

    error = pthread_create(&h1, NULL, slow_printf, hola);
    if(error != 0)
    {
        fprintf(stderr, "pthread_create: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }

    error = pthread_create(&h2, NULL, slow_printf, mundo);
    if(error != 0)
    {
        fprintf(stderr, "pthread_create: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }

    error = pthread_join(h1, NULL);
    if(error != 0)
    {
        fprintf(stderr, "pthread_join: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }

    error = pthread_join(h2, NULL);
    if(error != 0)
    {
        fprintf(stderr, "pthread_join: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }

    printf("El programa %s termino correctamente \n", argv[0]);
    exit(EXIT_SUCCESS);
}
```

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

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

0,25 ptos. 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.

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

1,00 ptos.

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.

SEMANA 2: PROCESOS Y EJECUCIÓN DE PROGRAMAS.

Procesos

Un **proceso** es un programa en ejecución. Todo proceso en un Sistema Operativo tipo Unix:

- Tiene un identificador de proceso (Process ID, PID).
- Tiene un proceso padre, y a su vez puede disponer de ninguno, uno o más procesos hijo.
- Tiene un propietario, el usuario que ha lanzado dicho proceso.
- El proceso `init` (PID = 1) es el padre de todos los procesos. Es la excepción a la norma general, pues no tiene padre.

Para mostrar la relación actual de procesos en el sistema se puede emplear la orden en línea de comandos `ps`.

```
$  ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 11:48 ?        00:00:00 /sbin/init
...
practica    1712     1 18 12:08 ?        00:00:00 gnome-terminal
practica    1713  1712  0 12:08 ?        00:00:00 gnome-pty-helper
practica    1714  1712 20 12:08 pts/0    00:00:00 bash
practica    1731  1714  0 12:08 pts/0    00:00:00 ps -ef
```

La columna `UID` indica el identificador del usuario que lanzó el proceso, la columna `PID` contiene el identificador del proceso, la columna `PPID` el PID del proceso padre (Parent PID, PPID), y la columna `CMD` muestra el comando que se está ejecutando.

Padres, Hijos, Zombies y Huérfanos

Todo proceso puede lanzar un proceso hijo en cualquier momento. Para ello, el Sistema Operativo ofrece la llamada al sistema `fork`. Inmediatamente después de una llamada a `fork` con éxito, se tendrán dos procesos casi idénticos ejecutándose concurrentemente, el **proceso**

padre (el original) y el **proceso hijo**, una copia exacta exceptuando su PID y su PPID. Sin embargo, procesos padre e hijo no comparten memoria, son completamente independientes. El proceso hijo hereda alguno de los recursos del padre, tales como los archivos y dispositivos abiertos.

Tras la creación del proceso hijo, ambos procesos comenzarán a ejecutar el código inmediatamente posterior al `fork`. Para distinguir el proceso padre del proceso hijo, `fork` retorna en el proceso padre el PID del hijo, y en el proceso hijo retornará `0`, que no puede ser el PID de un proceso hijo. Además, todos los procesos pueden acceder a su PID y al de su proceso padre con las funciones `getpid` y `getppid`, respectivamente.

Nota. *Los PIDs de procesos se almacenan en variables de tipo* `pid_t`*, que es un entero de un tamaño no especificado. Por tanto, para poder imprimirlo se debe hacer un casting a otro tipo de entero suficientemente grande como para contenerlo. Aunque típicamente funcionará imprimirlo como un* `int` *usando* `%d`*, el único método fiable sería hacer una conversión a* `intmax_t` *(definido en* `stdint.h`*) e imprimirlo con* `%jd`*.*

Nota. *En un programa multihilo, el nuevo proceso solo tendrá un hilo. Debido a que esto podría dejar el proceso en un estado inconsistente, hay un número reducido de acciones que se pueden realizar en el proceso hijo después de* `fork` *y antes de llamar a una función de la familia* `exec`*.*

Todo proceso padre es responsable de los procesos hijos que lanza, por ello, todo proceso padre debe recoger el resultado de la ejecución de los procesos hijos para que estos finalicen adecuadamente (al igual que se hacía en hilos con `pthread_join`). Para ello, el Sistema Operativo ofrece la llamada `wait` que permite esperar hasta obtener el resultado de la ejecución de un proceso hijo. Si se desea especificar el proceso hijo a esperar, se puede usar la función `waitpid`.

El valor obtenido mediante `wait` y `waitpid` no es siempre el valor retornado por `main` o `exit`. En ocasiones el proceso termina por una señal del Sistema Operativo (por ejemplo, por una violación de segmento). Las macros `WIFEXITED` y `WIFSIGNALED` permiten determinar si el proceso terminó por sí mismo o a causa de una señal del Sistema Operativo. Además, las macros `WEXITSTATUS` y `WTERMSIG` permiten obtener el valor de retorno o la señal que causó la muerte, en cada caso. Todas estas macros se encuentran documentadas en la página de manual de `wait`.

Si un proceso padre no recupera el resultado de la ejecución de su hijo, se dice que el proceso hijo queda en estado **zombie**. Un proceso hijo zombie es un proceso que ha terminado su ejecución (ha liberado los recursos que consumía pero sigue manteniendo una entrada en la tabla de procesos del Sistema Operativo) y que está pendiente de que su padre recoja el resultado de su ejecución.

Si un proceso padre termina sin haber esperado a los procesos hijos creados, estos últimos quedan **huérfanos**. Históricamente era el proceso `init` (PID = 1) el que recogía a los procesos huérfanos, pero desde la versión 3.4 del kernel de linux puede ser otro proceso ancestro (PID > 1) el encargado de recoger un proceso descendiente huérfano. En la generación de código hay que evitar dejar procesos hijo huérfanos. Todo proceso padre debe esperar por los procesos hijo creados.

Nota. *No existe equivalente a* `pthread_detach` *para procesos, pero se puede lograr un efecto similar haciendo un doble* `fork`*.*

1,00 ptos.

Ejercicio 7: Creación de Procesos. Dado el siguiente código en C, correspondiente al fichero "`ejemplo_fork.c`":

```
1 #include <stdio.h>
```

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

#define NUM_PROC 3

int main(void)
{
    pid_t pid;
    for(int i = 0; i < NUM_PROC; i++)
    {
        pid = fork();
        if(pid <  0)
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        else if(pid ==  0)
        {
            printf("Hijo %d\n", i);
            exit(EXIT_SUCCESS);
        }
        else if(pid >  0)
        {
            printf("Padre %d\n", i);
        }
    }
    wait(NULL);
    exit(EXIT_SUCCESS);
}
```

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

0,25 ptos. 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.

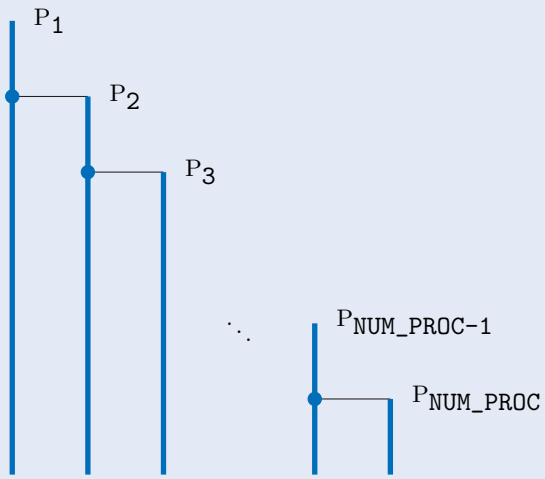
0,25 ptos. c) Analizar el árbol de procesos que genera el código de arriba. Mostrarlo en la memoria como un diagrama de árbol (como el que aparece en el Ejercicio 8) explicando por qué es así.

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

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

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

0,50 ptos. **Ejercicio 8: 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.

Ejercicio 9: Espacio de Memoria. Dado el siguiente código en C, correspondiente al fichero "`ejemplo_malloc.c`":

```c
/* wait and return process info */
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MESSAGE "Hello"

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

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

0,25 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. 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?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Ejecución de Programas

Las llamadas al sistema `exec` son una familia de funciones que permiten reemplazar el código del proceso actual por el código del programa que se pasa como parámetro. Un proceso hijo puede ejecutar un programa diferente al proceso padre, es decir, código máquina diferente al del padre. Para ello debe invocar a otros programas ejecutables.

Nota. *No se debe realizar ningún tipo de tratamiento tras una llamada* `exec` *pues dicho código nunca llega a ejecutarse si* `exec` *se ejecuta con éxito. Únicamente debe realizarse el tratamiento de errores tras invocar a dicha llamada, puesto que no existe retorno después de la ejecución de* `exec`*, a menos que haya un error.*

Las funciones de la familia `exec` tienen esencialmente el mismo objetivo, que es la ejecución de un nuevo programa. Para ello han de recibir el programa a ejecutar, junto con los parámetros que se le pasarán. A pesar de ser muy parecidas, estas funciones presentan algunas diferencias:

- Las funciones que contienen la letra "`v`" (`execv`, `execvp` y `execve`) reciben los argumentos que se pasarán a la función `main` del programa a ejecutar en un array, con `NULL` como último elemento. Las funciones que contienen la letra "`l`" (`execl`, `execlp` y `execle`) aceptan esos argumentos como parámetros separados (como `printf`), acabando con `(char *)NULL`.
- Las funciones que terminan en la letra "`e`" (`execve` y `execle`) reciben un nuevo conjunto de variables de entorno que reemplaza al existente. Las variables de entorno son pares clave–valor, que se copian normalmente de un proceso padre a un proceso hijo a menos que se usen estas funciones. Un proceso puede acceder a sus variables de entorno a través de la variable global `environ`, o usando la función `getenv`. Típicamente estas variables se usan para establecer opciones de configuración, como el idioma de los programas o el editor de texto por defecto.
- Las funciones que contienen la letra "`p`" (`execvp` y `execlp`) buscan el nombre del fichero a ejecutar (si el caracter "`/`" no aparece en dicho nombre) en los directorios que se encuentren en la variable de entorno `PATH` del proceso. Esta variable de entorno contiene una serie de rutas separadas por "`:`" en las que buscar ficheros ejecutables.
- La función `fexecve` permite proporcionar directamente el programa, en lugar de la ruta al mismo.

1,50 ptos. **Ejercicio 10: Shell.** Escribir un programa en C ("`ejercicio_shell.c`") que implemente una shell sencilla (sin redirecciones ni estructuras de control).

1,00 ptos. 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.

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

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

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

+0,50 ptos. 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.

El Directorio `/proc`

Un concepto que Linux implementa basado en ideas de Plan 9 en lugar de basado en Unix es el del directorio de ficheros `/proc`. Este directorio no contiene ficheros reales. En lugar de eso, contiene información del Sistema Operativo, simulando ser ficheros, con el fin de que se pueda manipular con las mismas herramientas que se usan para manipular ficheros (una vez más, se sigue la filosofía de Unix de que "todo es un fichero").

Al ejecutar `ls` en este directorio, lo primero que se observa es que hay una carpeta con el PID de cada proceso del sistema. Esta carpeta contiene toda la información que el Sistema Operativo ofrece sobre cada uno de estos procesos, y es de donde los comandos como `ps` obtienen la información que muestran. Se puede acceder a la información de aquellos procesos para los que se tiene permiso, como por ejemplo los del propio usuario. El fichero `self` es un enlace a la carpeta correspondiente al PID del proceso que accede a él.

0,50 ptos. **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'`.

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

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

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

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

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

Ficheros

Una de las formas más comunes de abrir y cerrar ficheros en C es usando las funciones `fopen` y `fclose`. Del mismo modo, se puede imprimir con formato usando `fprintf`, y leer líneas usando `fgets`; también se pueden realizar escrituras y lecturas con un tamaño conocido usando las funciones `fwrite` y `fread`. Todas estas funciones manipulan ficheros a través de un tipo abstracto de datos llamado `FILE`, proporcionado por la biblioteca estándar de C. Por defecto, la mayoría de programas comienzan con tres objetos de tipo `FILE` abiertos por la biblioteca estándar: `stdin`, `stdout` y `stderr`. Sin embargo, las funciones que manipulan objetos `FILE` en un sistema Unix están construidas sobre otras de más bajo nivel proporcionadas por el Sistema Operativo, que se describen a continuación.

La función que ofrece el Sistema Operativo para abrir ficheros del sistema de archivos es `open`. Con respecto a `fopen`, se observan varias diferencias:

- La función `open` no retorna un objeto de tipo `FILE`. En lugar de eso retorna un valor de tipo `int`, que las funciones del Sistema Operativo utilizan para referirse a un fichero abierto. Este valor entero se conoce como **descriptor de fichero** y es un índice en la llamada **tabla de descriptores de fichero** del proceso, que se explicará a continuación.
- La función `open` tiene un parámetro que permite especificar una serie de *flags* para indicar cómo debe abrirse el fichero. Algunos de estos *flags* permiten usar opciones que en `fopen` podían especificarse a través de una cadena de texto, mientras que otras opciones son exclusivas de `open`. Entre los *flags* se debe especificar obligatoriamente una de las siguientes opciones: `O_RDONLY`, `O_WRONLY` y `O_RDWR`. Estos *flags* indican si el fichero abierto va a manipularse únicamente para leer o para escribir, o si se van a realizar ambas acciones. Además, los siguientes *flags* pueden ser útiles:
  - `O_APPEND`: Hace que las escrituras en el fichero se realicen siempre al final (se usa cuando se redirige usando `>>`).
  - `O_CLOEXEC`: Cierra el fichero al realizar una llamada con éxito a una función de la familia `exec`. En programas multihilo que vayan a crear procesos es importante usarlo para todos los ficheros que no deseemos que se copien en posibles descendientes, ya que otros hilos podrían crear un hijo en cualquier momento.
  - `O_CREAT`: El fichero será creado si no existe.
  - `O_EXCL`: Debe combinarse con `O_CREAT`. Hace que `open` retorne un error si el fichero ya existe.
  - `O_TRUNC`: Trunca el fichero a tamaño 0 borrando el contenido, asumiendo que el fichero exista, sea un fichero normal y se haya abierto para escribir.
- El último parámetro de `open` es opcional, y solo es necesario si `O_CREAT` se encuentra entre los *flags*. Especifica los permisos que tendrá el nuevo fichero creado. En Unix, al crear un fichero éste adquirirá un usuario y grupo, copiado a partir del usuario y el grupo efectivos del proceso creador. Los ficheros tienen permisos distintos para el usuario del fichero, los miembros (distintos del usuario) del grupo del fichero, y todos los demás usuarios del

sistema. Para cada uno de estos grupos es posible especificar tres tipos de permisos: lectura, escritura y ejecución.

Lectura y Escritura

Para leer y escribir en el fichero apuntado por un descriptor de fichero se usarán las funciones `read` y `write`. Estas funciones permiten la lectura y escritura de un número fijo de bytes, al igual que las funciones `fread` y `fwrite` hacían sobre objetos `FILE`.

Hay que tener en cuenta que al usar estas funciones se puede leer o escribir un tamaño menor al especificado. Esto puede producirse, por ejemplo, si el número de bytes disponible para leer era menor al especificado, o si se produce un error al leer o escribir después de que se hayan leído o escrito varios bytes con éxito. Por ello, es importante comprobar el número devuelto por estas funciones, que será el número de bytes leídos o escritos realmente. En caso de que deseemos completar la lectura o escritura deberá reintentarse la operación con los datos faltantes. Si se produjo una lectura o escritura "corta" como resultado de un error, es probable que la siguiente llamada produzca el error sin llegar a leer o escribir nada (no es necesariamente así, ya que el error podría haberse solucionado en ese tiempo, por ejemplo por mediación de otro proceso).

Tabla de Descriptores de Fichero

La **tabla de descriptores** de fichero es un array dentro del proceso, gestionado por el Sistema Operativo. Los descriptores de fichero son índices dentro de esta tabla. Al comenzar un programa típicamente tendrá tres descriptores de fichero ya abiertos: `STDIN_FILENO` (0), `STDOUT_FILENO` (1) y `STDERR_FILENO` (2):

Descriptor	Significado
0	Entrada estándar
1	Salida estándar
2	Salida de errores
...	...

Nota. *El hecho de que el descriptor de fichero de la salida de errores tenga el valor* 2 *es la razón por la que se puede redirigir en la shell con el operador* `2>`*. De hecho, se pueden usar los operadores* `<`*,* `>` *y* `>>` *con cualquier número de descriptor de fichero.*

Las estructuras que se almacenan en este array constan de dos partes: una serie de *flags* independientes para cada descriptor de fichero y un puntero a una **descripción de fichero abierto** gestionada por el Sistema Operativo. Varios descriptores de fichero, incluso aquellos de procesos diferentes, pueden referirse a la misma descripción de fichero abierto. Esto es lo que ocurre, por ejemplo, al realizar un `fork`, ya que los descriptores del proceso hijo se referirán a las mismas descripciones que el padre tenga abiertas (por ejemplo, si el padre tenía abierto un descriptor para un fichero en disco, escribir en ese mismo descriptor en el hijo avanzará la posición de escritura en el padre, ya que ésta se guarda en la descripción de fichero abierto).

Para cerrar el descriptor de fichero se usa la función `close`. Si se desea borrar una de las rutas para acceder al fichero desde el sistema de archivos se usa `unlink`. El fichero solo se borrará realmente de disco cuando todas las rutas que se usan para acceder a él sean borradas y todos los procesos que lo tuvieran abierto (o que mapeen su contenido a memoria, como se verá en la tercera práctica) lo cierren.

0,50 ptos.

Ejercicio 12: Visualización de Descriptores de Fichero. Dado el siguiente código en C, correspondiente al fichero "`ejemplo_descriptores.c`":

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

#define MESSAGE "Hello"

#define FILE1 "file1.txt"
#define FILE2 "file2.txt"
#define FILE3 "file3.txt"

int main(void)
{
    fprintf(stderr, "PID = %d\nStop 1\n", getpid());
    getchar();

    int file1 = open(FILE1, O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR
        | S_IRGRP | S_IWGRP);
    if(file1 == -1)
    {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Escribir mensaje */
    ssize_t total_size_written = 0;
    size_t target_size = sizeof(MESSAGE);
    do {
        ssize_t size_written = write(file1, MESSAGE + total_size_written,
            target_size - total_size_written);
        if(size_written == -1)
        {
            perror("write");
            exit(EXIT_FAILURE);
        }

        total_size_written += size_written;
    } while(total_size_written != target_size);

    fprintf(stderr, "Stop 2\n");
    getchar();

    int file2 = open(FILE2, O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR
        | S_IRGRP | S_IWGRP);
    if(file2 == -1)
    {
        perror("open");
        exit(EXIT_FAILURE);
    }

    fprintf(stderr, "Stop 3\n");
    getchar();

    int status = unlink(FILE1);
    if(status != 0)
```

```
55      {
56          perror("unlink");
57          exit(EXIT_FAILURE);
58      }
59
60      fprintf(stderr, "Stop 4\n");
61      getchar();
62
63      close(file1);
64
65      fprintf(stderr, "Stop 5\n");
66      getchar();
67
68      int file3 = open(FILE3, O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR
            | S_IRGRP | S_IWGRP);
69      if(file3 == -1)
70      {
71          perror("open");
72          exit(EXIT_FAILURE);
73      }
74
75      fprintf(stderr, "Stop 6\n");
76      getchar();
77
78      int file4 = open(FILE3, O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP |
            S_IWGRP);
79      if(file4 == -1)
80      {
81          perror("open");
82          exit(EXIT_FAILURE);
83      }
84
85      fprintf(stderr, "Stop 7\n");
86      getchar();
87
88      close(file2);
89      close(file3);
90      close(file4);
91
92      status = unlink(FILE2);
93      if(status != 0)
94      {
95          perror("unlink");
96          exit(EXIT_FAILURE);
97      }
98
99      status = unlink(FILE3);
100     if(status != 0)
101     {
102         perror("unlink");
103         exit(EXIT_FAILURE);
104     }
105 }
```

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.

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

0,20 ptos. 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?

0,10 ptos. 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`?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Descriptores de Fichero y Objetos `FILE`

Es posible crear un objeto `FILE` asociado a un descriptor de fichero arbitrario, usando la función `fdopen`. También es posible obtener el descriptor de fichero asociado a un objeto `FILE` usando la función `fileno`.

Nota. *Algunos objetos* `FILE`*, como los retornados por* `fmemopen` *y* `open_memstream`*, no se refieren a ficheros reales y no tienen un descriptor de fichero asociado.*

Dada la aparente similitud entre las funciones que trabajan con descriptores de fichero y las que utilizan los objetos `FILE`, es posible preguntarse por qué son necesarios ambos tipos de funciones. La razón es que presentan diferencias sustanciales y no pueden ser intercambiadas en todas las situaciones. La principal diferencia entre ambos conjuntos de funciones es que los objetos `FILE` tienen un buffer privado en el espacio de memoria del proceso, mientras que los descriptores de fichero no lo tienen.

Cuando una función de lectura lee de un objeto `FILE`, típicamente lee más caracteres de los que se le piden, y copia los caracteres sobrantes en un array asociado al objeto. Cuando vuelva a tener que leer, leerá del array el contenido disponible, y solo volverá a leer del descriptor de fichero cuando los caracteres de dicho array, o *buffer*, se agoten. De este modo se minimizan las llamadas al sistema que se deben ejecutar si, por ejemplo, el programador está leyendo carácter a carácter.

Del mismo modo, cuando se utilizan las funciones de escritura de un objeto `FILE`, en lugar de escribir directamente en el descriptor de fichero, se escribe en un array en memoria. Solo cuando el array esté completamente lleno (o, si se escribe en terminal, al introducir un carácter `\n`) se escribirá en el descriptor de fichero. También puede vaciarse manualmente el buffer de escritura usando `fflush`.

Estas funciones, por tanto, tienen ventajas en varias circunstancias, pero también desventajas. Si se requiere que el contenido esté disponible inmediatamente tras escribir, el uso del buffer puede ser un inconveniente. Además, si el proceso termina de forma abrupta, es posible que el contenido del buffer no llegue a escribirse. Por último, en el caso de programas multihilo, las funciones de objetos `FILE` garantizan que los hilos no van a acceder simultáneamente al buffer, pero no sin cierta penalización en el rendimiento.

Nota. *Es posible configurar el uso del buffer para cada objeto* `FILE` *usando la función* `setvbuf`*. El objeto* `stderr` *normalmente no usa buffer por defecto, para que los mensajes de diagnóstico se escriban cuanto antes.*

Es importante recalcar que aunque los descriptores de fichero no tengan buffer en la memoria del proceso, cuando se refieren a ficheros en disco el Sistema Operativo mantiene una caché de las partes recientemente usadas (o cuyo uso se prevé) del contenido del fichero en memoria, común a todos los procesos, de modo que las lecturas y escrituras no tengan que acceder directamente al disco en la mayoría de los casos.

Nota. *Por tanto, si el Sistema Operativo termina de forma abrupta es posible que parte del contenido supuestamente escrito de un fichero no haya alcanzado el disco y se pierda.*

0,50 ptos. **Ejercicio 13: Problemas con el Buffer.** Dado el siguiente código en C, correspondiente al fichero "`ejemplo_buffer.c`":

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

int main(void)
{
    pid_t pid;

    printf("Yo soy tu padre");

    pid = fork();
    if(pid <  0)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if(pid ==  0)
    {
        printf("Noooooo");
        exit(EXIT_SUCCESS);
    }

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

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

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

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

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

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Tuberías (*Pipes*)

Para comunicar dos procesos con relación parental-filial es posible emplear el mecanismo de tuberías. Este mecanismo permite crear un canal de comunicación unidireccional.

Básicamente, una tubería consiste en dos descriptores de fichero. Uno de ellos permite leer de la tubería (`fd[0]`) y el otro permite escribir en la tubería (`fd[1]`). Al tratarse de descriptores de fichero, se pueden emplear las llamadas `read` y `write` para leer y escribir de una tubería como si de un fichero se tratase.

Nota. *Esta es la funcionalidad que se utiliza para implementar el operador* `|` *en la shell.*

Para crear una tubería simple en lenguaje C, se usa la llamada al sistema `pipe`, que tiene como argumento de entrada un array de dos enteros, y si tiene éxito, la tabla de descriptores de fichero contendrá dos nuevos descriptores de fichero para ser usados por la tubería. La función devuelve `-1` en caso de error.

El resultado de la llamada `pipe` sobre la tabla de descriptores del fichero es el siguiente:

Descriptor	Significado
0	Entrada estándar
1	Salida estándar
2	Salida de errores
...	...
`fd[0]`	Acceso a la tubería en modo lectura
`fd[1]`	Acceso a la tubería en modo escritura
...	...

Para que pueda existir comunicación entre procesos, la creación de la tubería siempre es anterior a la creación del proceso hijo. Tras la llamada a `fork`, el proceso hijo, que es una copia del padre, se lleva también una copia de la tabla de descriptores de fichero. Por tanto, padre e hijo disponen de acceso a los descriptores que permiten operar con la tubería. Dado que las tuberías son un mecanismo unidireccional, es necesario que únicamente uno de los procesos escriba, y que el otro únicamente lea. En el caso de querer establecer una comunicación bidireccional deberán utilizarse dos tuberías, como se muestra en la Figura 1

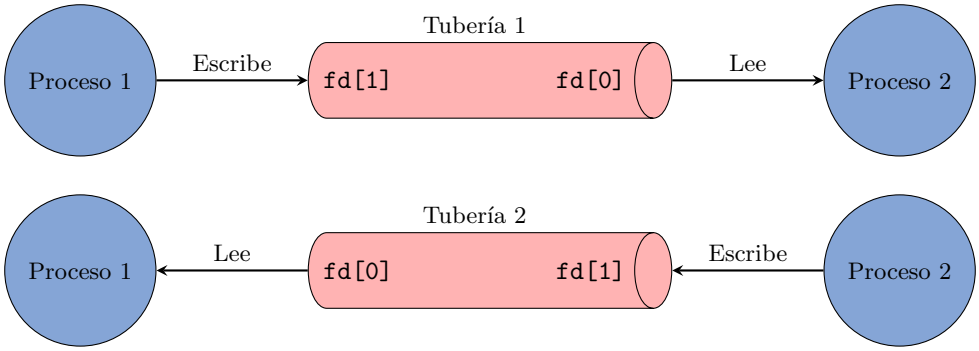


Figura 1: Comunicación bidireccional utilizando tuberías.

Las escrituras en una tubería de tamaño menor a `PIPE_BUF` caracteres se realizan de forma atómica. Esto quiere decir que no serán escrituras cortas ni podrán mezclarse con escrituras de otros hilos. Además el lector de la tubería verá la escritura completa.

La tubería tiene un número máximo de bytes que es capaz de aceptar. En caso de que un hilo intente escribir más contenido del que cabe en la tubería, el hilo esperará automáticamente a que quede el hueco suficiente. En caso de que quede algo de hueco y la escritura sea de tamaño superior a `PIPE_BUF` se producirá una escritura corta de lo que quepa. En caso de leer del pipe, si este está vacío se esperará a que haya contenido. Por tanto, lectores y escritores se esperan mutuamente.

En caso de que un hilo intente leer de una tubería vacía y no haya escritores (ningún proceso tiene abierto el extremo de escritura), entonces leerá 0 caracteres, indicando `EOF`. Por ello, es importante que el proceso lector cierre el extremo de escritura, y viceversa, con el fin de poder reconocer cuando la escritura ha finalizado. En caso de que un hilo intente escribir en una tubería sin lectores, el Sistema Operativo mandará la señal `SIGPIPE` al proceso al que pertenezca el hilo. Por defecto esta señal finalizará el proceso.

Nota. *El envío de la señal* `SIGPIPE` *tiene como objetivo finalizar un proceso de un* pipeline *si el siguiente proceso ha terminado, ya que en ese caso continuar ejecutándose no altera el resultado final. Este es el caso si un proceso no lee toda su entrada, como por ejemplo el comando* `head`*.*

0,25 ptos.

Ejercicio 14: Ejemplo de Tuberías. Dado el siguiente código en C, correspondiente al fichero "`ejemplo_pipe.c`":

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

int main(void) {
    int fd[2];

    const char *string = "Hola a todos!\n";
    char readbuffer[80];

    int pipe_status = pipe(fd);
    if(pipe_status == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

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

    if (childpid == 0) {

        /* Cierre del descriptor de entrada en el hijo */
        close(fd[0]);

        /* Enviar el saludo vía descriptor de salida */
        /* strlen(string) + 1 < PIPE_BUF así que no hay escrituras cortas
           */
        ssize_t nbytes = write(fd[1], string, strlen(string) + 1);
        if(nbytes == -1)
        {
            perror("write");
            exit(EXIT_FAILURE);
        }

        printf("He escrito en el pipe\n");

```

```
44          exit(EXIT_SUCCESS);
45      } else {
46          /* Cierre del descriptor de salida en el padre */
47          close(fd[1]);
48          /* Leer algo de la tubería... el saludo! */
49          ssize_t nbytes = 0;
50          do {
51              nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
52              if(nbytes == -1)
53              {
54                  perror("read");
55                  exit(EXIT_FAILURE);
56              }
57              if(nbytes > 0)
58              {
59                  printf("He recibido el string: %.*s", (int) nbytes,
                        readbuffer);
60              }
61          } while(nbytes != 0);
62
63          wait(NULL);
64          exit(EXIT_SUCCESS);
65      }
66  }
```

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

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

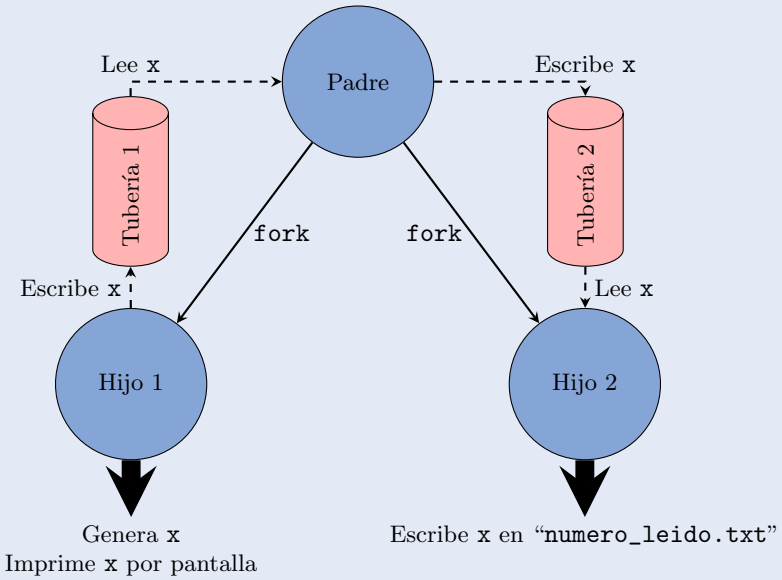
+0,25 ptos. c) *(Opcional)* Modificar el proceso hijo para que espere 1 segundo antes de escribir. Modificar el proceso padre para que finalice sin leer de la tubería y sin esperar al proceso hijo. ¿Qué se imprime ahora? ¿Por qué?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

1,50 ptos. **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 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 esquema de este programa se resume en la siguiente 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.