

Práctica 1

FECHA DE ENTREGA: DEL 3 AL 6 DE MARZO
(HORA LÍMITE: ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

Introducción a la shell e hilos

Introducción a la shell
Programación en C con POSIX

Procesos y ejecución de programas

Procesos
Ejecución de programas

Hilos

Ficheros y tuberías

El directorio `/proc`
Ficheros
Tuberías (*Pipes*)

Ejercicio de codificación

***Nota.** La práctica consta de dos tipos de ejercicios diferentes. Los ejercicios cortos son ejercicios de aprendizaje sin puntuación asociada, que no es necesario entregar porque se evaluarán a través del examen correspondiente de evaluación continua. La calificación de la entrega corresponderá únicamente al ejercicio de codificación, incluyendo tanto el código como la documentación.*

Introducción a la shell e hilos

Introducción a la shell

Una **shell** es una interfaz de usuario que permite usar los recursos del sistema operativo. Aunque se pueden entender los entornos gráficos (como Windows, macOS, 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 macOS, 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, aunque no es difícil adaptarse a la sintaxis de la familia Bourne, más extendida,

como se muestra en el siguiente ejemplo:

1	<code>#!/usr/bin/env sh</code>	1	<code>#!/usr/bin/env csh</code>
2	<code>if [\$days -gt 365] then</code>	2	<code>if (\$days > 365) then</code>
3	<code>echo This is over a year</code>	3	<code>echo This is over a year</code>
4	<code>fi</code>	4	<code>endif</code>

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 siguen 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. 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 las prácticas. Como todo buen comando, **man** también tiene su manual, al que se puede acceder 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**.

Ejercicio 1: Uso del manual.

- a) Buscar en el manual la lista de funciones disponibles para el manejo de hilos. Las funciones de manejo de hilos comienzan por **<pthread>**.
- 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**.

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 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 mostrar las primeras **n** líneas de un fichero, basta utilizar **head -n <fichero>**.
- **tail**: El análogo de **head** pero para mostrar 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.
- **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>**.

Ejercicio 2: Comandos y redireccionamiento.

- a) Escribir un comando que busque las líneas que contengan «molino» en el fichero «don quijote.txt» y las añada al final del fichero «aventuras.txt».
- b) Elaborar un *pipeline* que cuente el número de ficheros en el directorio actual.
- 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**.

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. 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.

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`.

- ¿Qué mensaje se imprime al intentar abrir un fichero inexistente? ¿A qué valor de `errno` corresponde?
- ¿Qué mensaje se imprime al intentar abrir el fichero `/etc/shadow`? ¿A qué valor de `errno` corresponde?
- 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`?

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.

Ejercicio 4: Espera activa e inactiva.

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

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.

Notas.

- Los `PID` 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 lo 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`.
- 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 hijo que lanza; por tanto, todo proceso padre debe recoger el resultado de la ejecución de los procesos hijo 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 simplemente 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 **zombi**. Un proceso hijo zombi es un proceso que ha terminado y que está pendiente de que su padre recoja el resultado de su ejecución (por tanto, ha liberado los recursos que consumía pero sigue manteniendo una entrada en la tabla de procesos del sistema operativo).

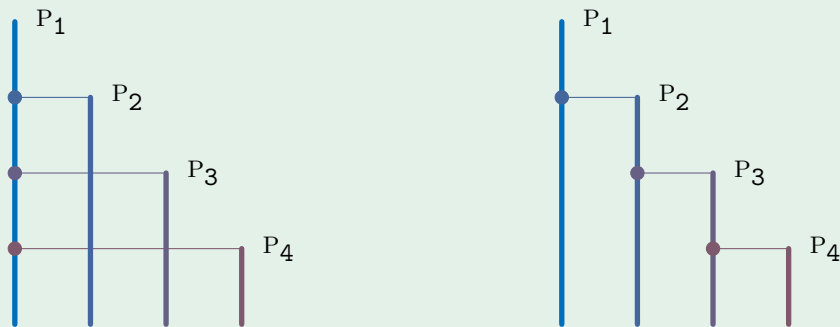
Si un proceso padre termina sin haber esperado a los procesos hijo creados, estos últimos quedan **huérfanos**. Históricamente era el proceso `init` (`PID = 1`) el que recogía a los procesos huérfanos, pero actualmente puede ser otro proceso ancestro (`PID > 1`) el encargado de recoger un proceso descendiente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

Ejercicio 5: Creación de procesos. Dado el siguiente código en C:

`proc_example.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 #define NUM_PROC 3
8
9 int main(void) {
10     int i;
11     pid_t pid;
12
13     for (i = 0; i < NUM_PROC; i++) {
14         pid = fork();
15         if (pid < 0) {
16             perror("fork");
17             exit(EXIT_FAILURE);
18         } else if (pid == 0) {
19             printf("Child %d\n", i);
20             exit(EXIT_SUCCESS);
21         } else if (pid > 0) {
22             printf("Parent %d\n", i);
23         }
24     }
25     wait(NULL);
26     exit(EXIT_SUCCESS);
27 }
```

- Analizar el texto que imprime el programa. ¿Se puede saber *a priori* en qué orden se imprimirá el texto? ¿Por qué?
- Cambiar el código para que el proceso hijo imprima su PID y el de su padre en vez de la variable `i`.
- Dados los dos siguientes diagramas de árbol:



¿A cuál de los dos árboles de procesos se corresponde el código de arriba, y por qué? ¿Qué modificaciones habría que hacer en el código para obtener el otro árbol de procesos?

- d) El código original deja procesos huérfanos, ¿por qué?
- e) Introducir el mínimo número de cambios en el código para que no deje procesos huérfanos.

Ejercicio 6: Espacio de memoria. Dado el siguiente código en C:

proc_malloc.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7
8 #define MESSAGE "Hello"
9
10 int main(void) {
11     pid_t pid;
12     char *sentence = calloc(sizeof(MESSAGE), 1);
13
14     pid = fork();
15     if (pid < 0) {
16         perror("fork");
17         exit(EXIT_FAILURE);
18     } else if (pid == 0) {
19         strcpy(sentence, MESSAGE);
20         exit(EXIT_SUCCESS);
21     } else {
22         wait(NULL);
23         printf("Parent: %s\n", sentence);
24         exit(EXIT_SUCCESS);
25     }
26 }
```

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

Nota. Con lo aprendido en esta sección se puede realizar la parte a) del ejercicio de programación.

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 carácter «/» 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 `fxexecve` permite proporcionar directamente el programa, en lugar de la ruta al mismo.

Ejercicio 7: Ejecución de programas. Dado el siguiente código en C:

proc_exec.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 int main(void) {
7     char *argv[3] = {"ls", "./", NULL};
8     pid_t pid;
9
10    pid = fork();
11    if (pid < 0) {
12        perror("fork");
13        exit(EXIT_FAILURE);
```

```

14 } else if (pid == 0) {
15     if (execvp("ls", argv)) {
16         perror("execvp");
17         exit(EXIT_FAILURE);
18     }
19 } else {
20     wait(NULL);
21 }
22 exit(EXIT_SUCCESS);
23 }

```

- ¿Qué sucede si se sustituye el primer elemento del array `argv` por la cadena `"mi-ls"`? ¿Por qué?
- ¿Qué modificaciones habría que hacer en el programa anterior para utilizar la función `execl` en lugar de `execvp`?

Nota. Se puede obtener la ruta completa de `ls` usando el comando `which ls`.

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 cálculo más rápido.

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 CPU 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 CPU 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, este 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 contará 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, se puede desligar un hilo llamando a `pthread_detach` con su identificador. 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*).

Ejercicio 8: Finalización de hilos. Dado el siguiente código en C:

thread_example.c

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  void *slow_printf(void *arg) {
8      const char *msg = arg;
9      size_t i;
10
11     for (i = 0; i < strlen(msg); i++) {
12         printf("%c ", msg[i]);
13         fflush(stdout);
14         sleep(1);
15     }
16
17     return NULL;
18 }
19
20 int main(int argc, char *argv[]) {
21     pthread_t h1;
22     pthread_t h2;
23     char *hello = "Hello ";
24     char *world = "World";
25     int error;
26
27     error = pthread_create(&h1, NULL, slow_printf, hello);
28     if (error != 0) {
29         fprintf(stderr, "pthread_create: %s\n", strerror(error));
30         exit(EXIT_FAILURE);
31     }
32
33     error = pthread_create(&h2, NULL, slow_printf, world);
34     if (error != 0) {
35         fprintf(stderr, "pthread_create: %s\n", strerror(error));
36         exit(EXIT_FAILURE);
37     }
38 }

```

```

39  error = pthread_join(h1, NULL);
40  if (error != 0) {
41      fprintf(stderr, "pthread_join: %s\n", strerror(error));
42      exit(EXIT_FAILURE);
43  }
44
45  error = pthread_join(h2, NULL);
46  if (error != 0) {
47      fprintf(stderr, "pthread_join: %s\n", strerror(error));
48      exit(EXIT_FAILURE);
49  }
50
51  printf("Program %s finished correctly\n", argv[0]);
52  exit(EXIT_SUCCESS);
53 }

```

- ¿Qué hubiera pasado si el proceso no hubiera esperado a los hilos? Para probarlo basta eliminar las llamadas a `pthread_join`.
- Con el código modificado del apartado anterior, indicar qué ocurre si se reemplaza la función `exit` por una llamada a `pthread_exit`.
- Tras eliminar las llamadas a `pthread_join` en los apartados anteriores, el programa no espera a que terminen todos los hilos. ¿Qué código habría que añadir para que los recursos del sistema operativo se gestionen correctamente a pesar de que no se espere a los hilos creados?

Nota. Con lo aprendido en esta sección se puede realizar la parte b) del ejercicio de programación.

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.

Ejercicio 9: Directorio de información de procesos. Buscar para alguno de los procesos la siguiente información en el directorio `/proc`. Hay que tener en cuenta que tanto las variables de entorno como la lista de comandos delimitan los elementos con `\0`, así que puede ser conveniente convertir los `\0` a `\n` usando `tr '\0' '\n'`.

- El nombre del ejecutable.
- El directorio actual del proceso.

- c) La línea de comandos que se usó para lanzarlo.
- d) El valor de la variable de entorno `LANG`.
- 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 este 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 se desee 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 descriptors de fichero

La **tabla de descriptors** de fichero es un array dentro del proceso gestionado por el sistema operativo. Los descriptors de fichero son índices dentro de esta tabla. Al comenzar un programa típicamente tendrá tres descriptors 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 descriptors 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 descriptors 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 esta 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.

Ejercicio 10: Visualización de descriptors de fichero. Dado el siguiente código en C:


```
file_descriptors.c
1 #include <fcntl.h>
```

```
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/stat.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7
8 #define MESSAGE "Hello"
9
10 #define FILE1 "file1.txt"
11 #define FILE2 "file2.txt"
12 #define FILE3 "file3.txt"
13
14 int main(void) {
15     int file1, file2, file3, file4;
16     size_t target_size;
17     ssize_t size_written, total_size_written;
18
19     fprintf(stderr, "PID = %d\nStop 1\n", getpid());
20     getchar();
21
22     if ((file1 = open(FILE1, O_CREAT | O_TRUNC | O_RDWR,
23                     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)) == -1) {
24         perror("open");
25         exit(EXIT_FAILURE);
26     }
27
28     /* Write message. */
29     total_size_written = 0;
30     target_size = sizeof(MESSAGE);
31     do {
32         size_written = write(file1, MESSAGE + total_size_written,
33                             target_size - total_size_written);
34         if (size_written == -1) {
35             perror("write");
36             exit(EXIT_FAILURE);
37         }
38         total_size_written += size_written;
39     } while (total_size_written != target_size);
40
41     fprintf(stderr, "Stop 2\n");
42     getchar();
43
44     if ((file2 = open(FILE2, O_CREAT | O_TRUNC | O_RDWR,
45                     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)) == -1) {
46         perror("open");
47         exit(EXIT_FAILURE);
48     }
49
50     fprintf(stderr, "Stop 3\n");
51     getchar();
52
53     if (unlink(FILE1) != 0) {
54         perror("unlink");
55         exit(EXIT_FAILURE);
56     }
57
58     fprintf(stderr, "Stop 4\n");
59     getchar();
60
61     close(file1);
```

```

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

```

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.

- Stop 1.* Inspeccionar los descriptores de fichero del proceso. ¿Qué descriptores de fichero se encuentran abiertos? ¿A qué tipo de fichero apuntan?
- Stop 2 y Stop 3.* ¿Qué cambios se han producido en la tabla de descriptores de fichero?
- 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?
- 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`?

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.

Ejercicio 11: Problemas con el *buffer*. Dado el siguiente código en C:

file_buffer.c

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

```

2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int main(void) {
8     pid_t pid;
9
10    printf("I am your father");
11
12    pid = fork();
13    if (pid < 0) {
14        perror("fork");
15        exit(EXIT_FAILURE);
16    } else if (pid == 0) {
17        printf("Noooooooo");
18        exit(EXIT_SUCCESS);
19    }
20
21    wait(NULL);
22    exit(EXIT_SUCCESS);
23 }

```

- ¿Cuántas veces se escribe el mensaje «I am your father» por pantalla? ¿Por qué?
- En el programa falta el terminador de línea (\n) en los mensajes. Corregir este problema. ¿Sigue ocurriendo lo mismo? ¿Por qué?
- Ejecutar el programa redirigiendo la salida a un fichero. ¿Qué ocurre ahora? ¿Por qué?
- ¿Cómo se puede corregir definitivamente este problema sin dejar de usar printf?

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
...	...
<code>fd[0]</code>	Acceso a la tubería en modo lectura
<code>fd[1]</code>	Acceso a la tubería en modo escritura
...	...

Para que pueda existir comunicación entre procesos, la creación de la tubería siempre debe ser 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.

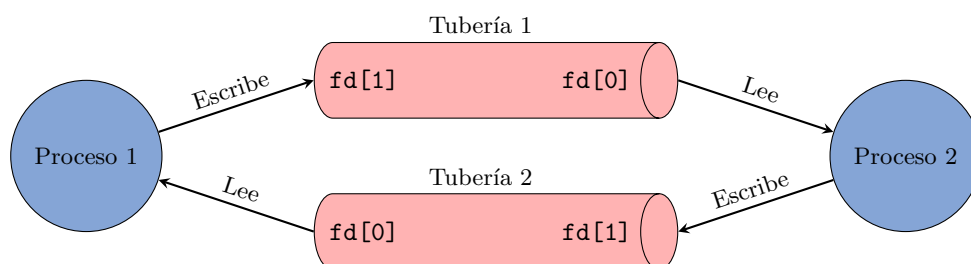


Figura 1: Comunicación bidireccional utilizando tuberías.

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 de la tubería, si está vacía 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`.

Ejercicio 12: Ejemplo de tuberías. Dado el siguiente código en C:

pipe_example.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7
8 int main(void) {
9     int fd[2];
10
11     const char *string = "Hi all!\n";
```

```

12 char readbuffer[80];
13 int pipe_status;
14 pid_t childpid;
15 ssize_t nbytes;
16
17 pipe_status = pipe(fd);
18 if (pipe_status == -1) {
19     perror("pipe");
20     exit(EXIT_FAILURE);
21 }
22
23 childpid = fork();
24 if (childpid == -1) {
25     perror("fork");
26     exit(EXIT_FAILURE);
27 }
28
29 if (childpid == 0) {
30     /* Close of the read end in the child. */
31     close(fd[0]);
32     /* The message is written in the write end. */
33     /* strlen(string) + 1 < PIPE_BUF, there are no short writes. */
34     nbytes = write(fd[1], string, strlen(string) + 1);
35     if (nbytes == -1) {
36         perror("write");
37         exit(EXIT_FAILURE);
38     }
39     printf("I have written in the pipe\n");
40
41     exit(EXIT_SUCCESS);
42 } else {
43     /* Close of the write end in the parent. */
44     close(fd[1]);
45     /* The message is read. */
46     nbytes = 0;
47     do {
48         nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
49         if (nbytes == -1) {
50             perror("read");
51             exit(EXIT_FAILURE);
52         }
53         if (nbytes > 0) {
54             printf("I have received the string: %.*s", (int)nbytes, readbuffer
55 );
56         }
57     } while (nbytes != 0);
58
59     wait(NULL);
60     exit(EXIT_SUCCESS);
61 }

```

- Ejecutar el código. ¿Qué se imprime por pantalla?
- ¿Qué ocurre si el proceso padre no cierra el extremo de escritura? ¿Por qué?

Nota. Con lo aprendido en esta sección se puede realizar la parte c) del ejercicio de programación.

Ejercicio de codificación

En este ejercicio se comenzará a programar parte del proyecto incremental (llamado "Miner Rush"), que una vez completo permitirá crear una red de mineros de bloques inspirada en la tecnología **Blockchain**. La red se compone de un conjunto de procesos denominados *mineros* que tendrán como misión la resolución de una prueba de esfuerzo (*Proof of Work*, POW). Todos los mineros tendrán que intentar resolver de forma concurrente la misma prueba, y solo uno, el primero en encontrarla, recibirá la recompensa: una moneda.

Una prueba de esfuerzo o POW consiste en resolver un reto matemático, habitualmente hallar el operando que permite obtener un determinado resultado tras aplicar una función hash no invertible, es decir que la única forma de encontrar el operando es por fuerza bruta. En particular, dado un objetivo (*target*) t y la función hash f se debe encontrar la solución s tal que $f(s) = t$. Puesto que la función f no es invertible, la única forma de encontrar s es probar todos los posibles valores hasta hallar aquel que satisface la ecuación.

Se proporcionan los ficheros `pow.c` y `pow.h` con la implementación de la POW. En concreto, la función `pow_hash` realiza el cálculo de la función hash, y la constante `POW_LIMIT` es el límite superior del rango de búsqueda (es decir, habrá que probar los valores entre 0 y `POW_LIMIT - 1`).

En esta primera iteración se implementará un único minero, que resolverá un número dado de POW de forma iterativa, como se describe a continuación. De igual manera, se implementarán los procesos

10,00 ptos.

Ejercicio 13: Sistema de minero único multihilo. Escribir un programa en C que implemente un minero que resuelva un número dado de POW por fuerza bruta, usando tantos hilos como se especifiquen. El ejercicio se divide en tres partes bien diferenciadas, de manera que se puedan ir implementado y probando secuencialmente.

2,00 ptos.

a) Sistema multiproceso:

- El sistema se debe ejecutar con tres parámetros:

```
./miner <TARGET_INI> <ROUNDS> <N_THREADS>
```

donde `miner` es el nombre del ejecutable, `<TARGET_INI>` es el objetivo del problema inicial, `<ROUNDS>` es el número de rondas que se van a realizar y `<N_THREADS>` es el número de hilos que se van a utilizar.

- El proceso principal, **Minero**, creará un nuevo proceso usando `fork`: el proceso **Registrador**.
 - El proceso **Minero** realizará las tareas especificadas en el Ejercicio 13b).
 - El proceso **Registrador** realizará las tareas especificadas en el Ejercicio 13c).
- El proceso **Minero** será el que determine cuándo ha completado la última ronda, y mandará una orden por la tubería para la finalización del proceso **Registrador** al que esperará (por ejemplo una solución -1 encontrada). Además, **Minero** y **Registrador** indicarán por pantalla el código de salida de su proceso hijo a través de las cadenas `"Miner exited with status %d"/"Logger exited with status %d"/"Miner exited unexpectedly"/"Logger exited unexpectedly"`, según los procesos terminen normalmente o no.

Nota. Para comprobar que esta parte funciona correctamente, se puede implementar un **Minero** y un **Registrador** trivial que simplemente impriman un mensaje y terminen su ejecución.

4,00 ptos.

b) Minero multihilo:

- Cada minero será un proceso, **Minero**, encargado de resolver la POW usando múltiples hilos en paralelo.
- El proceso **Minero** recibe el número de rondas de minado que debe realizar, el número de hilos que debe utilizar, y el objetivo del problema inicial que debe resolver.
- Para cada ronda, el proceso **Minero**:
 - Divide el espacio de búsqueda entre tantos hilos como se haya especificado, de manera que realicen búsquedas independientes para paralelizar la tarea.
 - Crea los hilos, y espera a que terminen.
 - Cuando un hilo encuentra la solución, terminan todos los hilos y se devuelve la solución al hilo principal del proceso **Minero**.
 - El proceso **Minero** fija como siguiente objetivo la solución obtenida en esta ronda.
- El proceso **Minero** sale con código **EXIT_FAILURE** si se produce algún error, y con código **EXIT_SUCCESS** si todo fue correctamente.

***Nota.** Se puede comprobar que este apartado es correcto imprimiendo la solución tras cada ronda de minado.*

3,00 ptos.

c) Registrador del sistema:

- El proceso **Registrador** será el encargado de registrar en un fichero las soluciones encontradas por el proceso **Minero**.
- Para ello, **Minero** enviará un mensaje a través de una tubería a **Registrador**, indicándole el objetivo y la solución obtenida.
- Cuando **Registrador** detecte que se ha cerrado la tubería para recibir mensajes de **Minero**, terminará su ejecución saliendo con código **EXIT_FAILURE** si se produce algún error, y con código **EXIT_SUCCESS** si todo fue correctamente.

Mientras la tubería que lo conecta con **Minero** se mantenga abierta, lee la información recibida, y los escribe usando la función `dprintf` en un fichero cuyo nombre dependa del PID de su proceso padre.

Formato del fichero Registrador El fichero generado tendrá una estructura de Bloque que iremos completando en subsiguientes prácticas que es como sigue:

```
Id:      [NUMERO RONDA]
Winner:   [ID PROCESO PADRE]
Target:   [TARGET RONDA]
Solution: [SOLUCION ENCONTRADA] [(validated)/(rejected)]
Votes:    [NUMERO RONDA]/[NUMERO RONDA]
Wallets:  [ID PROCESO PADRE]:[NUMERO RONDA]
```

Donde **[NUMERO RONDA]** es el número de Ronda que se está ejecutando, **[ID PROCESO PADRE]** es el identificador del proceso padre, **[TARGET RONDA]** es el objetivo a encontrar, **[SOLUCION ENCONTRADA]** es la solución encontrada por el **Minero** y **[(validated)/(rejected)]** será el valor si se ha validado o no la solución. Como esta funcionalidad de validación se hará en posteriores versiones de las prácticas con la red block chain funcionando, de momento se asume que será **(validated)** y el estudiante realizará pruebas para forzar que a veces la validación dé valor **(rejected)** (por ejemplo, con una periodicidad random o para determinados targets).

Este sería un ejemplo de ronda con solución validada.

```
Id:      19
Winner:  6837
Target:  27818400
Solution: 53980520 (validated)
Votes:   19/19
Wallets: 6837:19
```

Y este para una solución incorrecta:

```
Id:      19
Winner:  7072
Target:  27818400
Solution: 00000011 (rejected)
Votes:   19/19
Wallets: 7042:19
```

1,00 ptos.

- d) Pruebas y Razonamiento: Una vez programadas las diferentes partes, hacer pruebas con diferente número de rondas y diferente número de hilos, tomando tiempos de ejecución. ¿Qué conclusiones se pueden tomar respecto al número óptimo de hilos?

El esquema global del programa se resume en la Figura 2.

Se deberán tener en cuenta los siguientes aspectos: (a) entrega de la documentación en formato .pdf requerida en la normativa de prácticas, (b) control de errores, (c) cierre de las tuberías pertinentes, (d) espera de cada proceso padre a sus procesos hijo, y (e) gestión adecuada de los hilos creados.

Algunos ejemplos de ejecución del sistema final son los siguientes:

```
$ ./mrush 0 5 3
Solution accepted: 00000000 --> 38722988
Solution accepted: 38722988 --> 82781454
Solution accepted: 82781454 --> 59403743
Solution accepted: 59403743 --> 44638907
Solution accepted: 44638907 --> 98780967
Logger exited with status 0
Miner exited with status 0
```

En el caso de que se fuerce a que **Minero** envíe una solución incorrecta en la tercera ronda la salida esperada sería:

```
$ ./mrush 0 5 3
Solution accepted: 00000000 --> 38722988
Solution accepted: 38722988 --> 82781454
Solution rejected: 82781454 !-> 00000011
Solution accepted: 59403743 --> 44638907
Solution accepted: 44638907 --> 98780967
Logger exited with status 0
Miner exited with status 0
```

En el directorio del proceso donde se ejecuta **Registrador** deberá haber un fichero llamado **ppid.log** (donde ppid es el id del proceso padre) con una estructura similar a la siguiente:

```
$ cat 2345.log
Id:      1
Winner:   2345
Target:   00000000
Solution: 38722988 (validated)
Votes:    1/1
Wallets:  2345:1

Id:      2
Winner:   2345
Target:   38722988
Solution: 82781454 (validated)
Votes:    2/2
Wallets:  2345:2

Id:      3
Winner:   2345
Target:   82781454
Solution: 59403743 (validated)
Votes:    3/3
Wallets:  2345:3

Id:      4
Winner:   2345
Target:   59403743
Solution: 44638907 (validated)
Votes:    4/4
Wallets:  2345:4

Id:      5
Winner:   2345
Target:   44638907
Solution: 98780967 (validated)
Votes:    5/5
Wallets:  2345:5
```

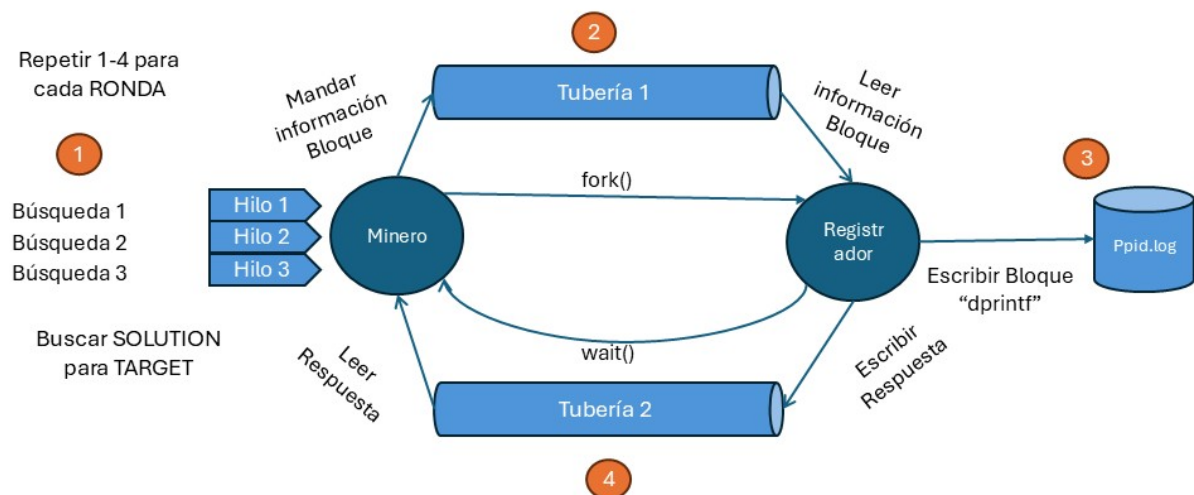



Figura 2: Descripción del funcionamiento general