Sistemas Operativos – Práctica 1

FECHA DE ENTREGA: SEMANA 27 FEBRERO -3 MARZO (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS).

La primera práctica se va a desarrollar en tres semanas con el siguiente cronograma:

- Semana 1: Introducción a la Shell, Expresiones Regulares y Descriptores de Ficheros
- Semana 2: Procesos, Procesos Padre y Procesos Hijo, Llamada a sistemas con la familia de funciones exec()
- Semana 3: Comunicación entre Procesos (IPC) mediante Tuberías

Los ejercicios correspondientes a esta primera práctica se van a clasificar en:

- **APRENDIZAJE**, se refiere a ejercicios altamente recomendable realizar para el correcto seguimiento de los conocimientos teóricos que se presentan en esta unidad didáctica.
- **ENTREGABLE**, estos ejercicios son obligatorios y deben entregarse correctamente implementados y documentados.

SEMANA 1

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 usaremos el término *shell* para referirnos a los Intérpretes de Líneas de Comandos (CLI).

Los sistemas Unix, como Linux o Mac OSX, disponen de diversas *shell* 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 *1hell* modernas son concebidas como intérpretes de comandos interactivos y como lenguajes de scripting. Las diferencias de sintaxis entre las distintas *1hell* se encuentran fundamentalmente en las estructuras de control. Las "C compatibles" 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:

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

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

```
fi                                          Endif
```

En Unix el flujo de datos y el conjunto de operaciones quedan definidos mediante ficheros. Es decir, en Unix *todo* es un fichero. Durante las prácticas de Sistemas Operativos exploraremos y aprovecharemos está forma de interpretar el conjunto de operaciones que nos ofrece Unix.

Para empezar, la mayoría de los sistemas Unix proporcionan una serie de utilidades o programas que pueden ser invocados desde la *shell* para realizar tareas concretas. El hecho de que podamos acceder directa y fácilmente a estos programas a través de la consola no debe llevarnos a concluir la simplicidad de los mismos. En efecto, la mayoría son programas con una potente funcionalidad y una gran eficiencia fruto de una cuidada programación. A continuación mostramos unos cuantos ejemplos de programas *shell* de gran utilidad:

- 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. Si durante estas prácticas un alumno no usa *man* al menos 30 veces puede considerar que está haciendo algo mal. 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, el número de sesión. Para leer la página con un determinado nombre e incluida en la sección *n_sec*, haríamos
      man n_sec <orden sobre la que obtener ayuda>.
      Así, si escribimos *man passwd* obtendremos la ayuda para la utilidad *passwd* de Linux. Por el contrario, si queremos conseguir la ayuda referida al fichero /etc/passd haremos *man 5 passwd.*
- Navegación y exploración de archivos
    - cd – Change directory
      Para navegar por el sistema de archivos. *cd <dir>* nos lleva al directorio <dir>, en caso de que exista en el directorio actual. *cd /* nos lleva al directorio raíz; *cd* ~ al directorio de usuario y *cd ..* al directorio que contiene al directorio actual.
    - ls – list directory contents

      Nos devuelve una lista con el contenido del directorio actual. Algunos parámetros útiles son:

      -l: nos devuelve información detallada de cada fichero (permisos, owner, tamaño, fecha de creación…)

      -a: lista también los ficheros ocultos (aquellos cuyo nombre comienza con “.”).

    - find – walk a file hierarchy

Sirve para buscar ficheros. Comando realmente potente que desciende recursivamente desde el directorio actual buscando los archivos que se correspondan con una expresión dada. Merece la pena hojear su manual.

- locate – find files by name
  Este comando es de gran utilidad cuando se desea buscar un fichero incluido en nuestro sistema de ficheros. La búsqueda del nombre de fichero se efectúa en una o más bases de datos generadas mediante updatedb.

- Visualización de ficheros
  - cat – concatenate and print files
    Imprime por pantalla el contenido de un fichero.
  - head – print first lines
    Imprime por pantalla las primeras líneas de un fichero (5 por defecto, pero se puede especificar el número de líneas por parámetros). Si queremos leer las primeras *n* líneas de un fichero, basta utilizar *head –n <nombre_fichero>*
  - tail – print last lines
    Como head, pero con las últimas líneas.
  - less – visualizador de archivos
    Permite navegar por un fichero. man abre los manuales, normalmente, con este visualizador. Aparte de moverse arriba y abajo, permite buscar en el documento, escribiendo una barra invertida ("/") seguida de la expresión a buscar y presionando "Enter". Presionando *n (b), less* se mueve al siguiente (anterior) resultado de la búsqueda. less no lee el archivo completo antes de empezar a mostrarlo con lo que es realmente rápido y útil para visualizar archivos de gran tamaño.
- Gestión de procesos y sistema
  - top – display dynamic information about processes

    Muestra de forma dinámica información sobre los procesos en ejecución (ej., cpu que consumen, memoria…).

  - ps – Process status

    Muestra información sobre los procesos que son controlados por una terminal. Algunas opciones útiles:

    - –a muestra también procesos de otros usuarios
    - –l muestra más información de cada proceso
  - pstree – show the running process as a tree
    Muestra todos los procesos siguiendo una estructura de árbol. La raíz de ese árbol es el pid dado como entrada de pstree, aunque también es posible pasar como argumento de entrada el nombre de un usuario. En este caso pstree dará lugar a varios árboles, siendo la raíz de cada árbol un proceso creado por el usuario dado como argumento de entrada. Finalmente, si se usa pstree sin argumento se muestra el árbol de procesos que tiene por raíz el proceso init.
  - df – display free disk space
    Muestra la cantidad de espacio libre en las distintas particiones de los discos duros. Con la opción –h muestra la información en formato human readable (Gb, Mb…)

- du – estimate file space usage
  Muestra el tamaño de los ficheros incluidos en un directorio. Si se ejecuta *du* sin aportar ningún nombre de directorio como argumento, devuelve la distribución de espacio para el actual directorio (el directorio que nos devuelve la orden *pwd)*.
- free - Display amount of free and used memory in the system
  Informa sobre el consume de memoria (RAM y memoria swap, así como el uso de buffers).

- Trabajo con ficheros
  - grep – file pattern searcher

    Busca un patrón (expresión regular) en un fichero (o en los ficheros de un directorio). Potentísima aplicación de la que merece la pena leerse el manual

  - wc – word count

    Cuenta las letras, palabras y líneas de un fichero.

  - sort – sort lines of text files

    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 – report or filter out repeated lines in a file

    Por defecto, devuelve las líneas no repetidas de un fichero. Sólo filtra las líneas repetidas consecutivas con lo que si queremos filtrar todas las líneas repetidas es necesario, primero, pasar el archivo por un sort: *sort file | uniq*

- Redirección. Procesos y ficheros
  - &

  Se escribe al final de la línea a ejecutar. Provoca que el comando se ejecute en segundo plano, quedando la terminal libre para nuevas interacciones.

  - |

    Tubería (en inglés, pipe). Redirige la salida del comando anterior al pipe y como entrada del posterior al pipe. Así, *sort file | uniq | wc -l* ordena el fichero *file* y se lo pasa como argumento de entrada a *uniq* que devuelve exclusivamente las líneas únicas, y este resultado es pasado como entrada a *wc –l* que devuelve el número de líneas. De esta forma, en conjunto los tres comandos sirven para contar el número de líneas únicas de un fichero.

  - >

    Redirecciona la salida, que por defecto es *stdout,* al fichero deseado. Así *sort file | uniq > file_uniq* guarda en *file_uniq* las líneas únicas del fichero *file*

- <

  Redirección de entrada. Redirecciona la entrada, que por defecto es *stdin,* desde el fichero deseado

Expresiones Regulares

Una *expresión regular* nos sirve para definir lenguajes imponiendo restricciones sobre las secuencias de caracteres que se permiten. De modo más concreto, una *expresión regular* puede estar constituida por caracteres del alfabeto normal, más un pequeño conjunto de caracteres extra (*meta-caracteres)* que nos permitirán definir aquellas restricciones. El conjunto de meta-caracteres que está más extendido es el siguiente

Nombre	**Carácter**	**Significado**
Cierre	*	El elemento precedente debe aparecer 0 o más veces
Cierre positivo	+	El elemento precedente debe aparecer 1 o más veces
Comodín	.	Un carácter cualquiera excepto salto de línea
Condicional	?	Operador unario. El elemento precedente es opcional
OR	|	Operador binario. Operador OR entre dos elementos. En el lenguaje aparecerá o uno u otro.
Comienzo de línea	^	Comienzo de línea
Fin de línea	$	Fin de línea
	[…] (caracteres entre corchetes)	Conjunto de caracteres admitidos
	[^…]	Conjunto de caracteres no admitidos
Operador de rango	-	Dentro de un conjunto de caracteres escrito entre corchetes podemos especificar un rango (ej., [a-zA-Z0-9]
	(…) (elementos entre paréntesis)	Agrupación de varios elementos
Carácter de escape	\ (barra inversa)	Debido a que algunos caracteres del alfabeto coinciden con metacaracteres, el carácter de escape permite indicar que un meta-carácter se interprete como un símbolo del alfabeto.
Salto de línea	'\n'	Carácter de salto de línea
Tabulador	'\t'	Carácter de tabulación

A continuación mostramos algunos ejemplos de expresiones regulares mediante grep

- Para mostrar todas las líneas de un fichero con un solo carácter
  - grep '^.$' <nombre_fichero>
- Para mostrar todas las líneas de un fichero con un solo punto
  - grep '^\.$' <nombre_fichero>
- Para mostrar todas las líneas de un fichero que contengan al menos una letra (mayúscula o minúscula)
  - grep '[a-zA-Z]' <nombre_fichero>

Tabla de descriptores de fichero

La tabla de descriptores de fichero es una estructura de datos propia de cada proceso existente en el sistema. Por lo tanto, hay una tabla de descriptores de fichero por proceso.

Un descriptor es un número entero que identifica una cierta operación con un fichero o dispositivo. El valor entero de un descriptor lo selecciona el sistema operativo y es arbitrario. La tabla de descriptores de fichero tiene un número limitado de descriptores definido por el sistema operativo, esto quiere decir que un proceso puede tener un número limitado de ficheros abiertos.

Los tres primero descriptores están abiertos de partida: 0 –stdin-, 1 –stdout- y 2 –stderr. El descriptor 0 nos permite operar con la entrada estándar (el teclado) como si de un fichero se tratase. Lo mismo sucede con los descriptores 1 y 2, que nos permiten imprimir mensajes por pantalla.

La tabla de descriptores de fichero la gestiona el sistema operativo y las funciones *open( )* y *close()* permiten obtener nuevos descriptores y liberarlos cuando ya no se necesiten. De forma que una vez obtenido el descriptor podemos realizar operaciones de lectura y escritura sobre el fichero.

Esquema de una tabla de descriptores de fichero de un proceso

Descriptor	Significado
0	Entrada estándar (teclado)
1	Salida estándar (pantalla)
2	Salida estándar de errores (pantalla)
. . .	
87	Fichero "datos.txt" en modo lectura (R)
. . .	

Ejercicio 1. (APRENDIZAJE) Estudia las siguientes funciones de manejo de ficheros (#include <fcntl.h>):

- int open(const char *path, int oflags);
- int open(const char *path, int oflags, mode_t mode); /*apertura extendida*/
- int close(int fildes);
- int read(int handle, void *buffer, int nbyte);
- int write(int handle, void *buffer, int nbyte);

Del mismo modo estudiar los posibles oflags: O_RDONLY, O_WRONLY, O_RDWR , O_APPEND, O_TRUNC, O_CREAT y O_EXCL.

SEMANA 2

Procesos

Un proceso es un programa en ejecución. Todo proceso en un sistema operativo tipo Unix:

- 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 primera columna indica el identificador del usuario (UID) del proceso, la segunda el identificador del proceso (PID), la tercera el PID del proceso padre (PPID). Por último, aparece el nombre del proceso en cuestión.

Procesos Padre y Procesos Hijo

Todo proceso (padre) puede lanzar un proceso hijo en cualquier momento, para ello el sistema operativo nos ofrece una llamada al sistema que se denomina *fork()*.

Un proceso hijo es un proceso clon del padre, es una copia exacta del padre exceptuando su PID. 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.

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. Para ello, el sistema operativo ofrece la llamada *wait()* que nos permite obtener el resultado de la ejecución de uno o varios procesos hijo. Si queremos que el proceso padre espere hasta que la ejecución del proceso hijo termine hay que hacer uso de las funciones *wait()* o *waitpid()* en el proceso padre junto con *exit()* en el proceso hijo.

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 y es el proceso init el que adopta a los procesos que se han quedado huérfanos. En la generación de código hay que evitar dejar procesos hijo huérfanos. Todo proceso padre debe esperar por los procesos hijo creados.

Ejercicio 2. **(APRENDIZAJE)** Estudia las siguientes funciones (#include <sys/types.h> #include <sys/wait.h> #include <stdio.h>):

- pid_t fork(void);
- pid_t wait(int *status);
- pid_t getpid(void);
- pid_t getppid(void);
- pid_t waitpid(pid_t pid, int *status, int options);
- void exit(int status);

Ejercicio 3. **(APRENDIZAJE)** Analiza la información de estado almacenada en la variable status con las siguientes macros:

- WIFEXITED (*status)
- WEXITSTATUS (*status)
- WIFSIGNALED (*status)
- WTERMSIG (*status)
- WIFSTIPPED (*status)
- WSTOPSIG (*status)

Ejercicio 4. **(ENTREGABLE)** (2,25 puntos)

a) Analiza el árbol de procesos vinculado al siguiente código:

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

#define NUM_PROC 3

int main (void)
{
  int pid;
  int i;
  for (i=0; i < NUM_PROC; i++){
```

```
    if ((pid=fork()) <0 ){
      printf("Error al emplear fork\n");
      exit(EXIT_FAILURE);
    }else if (pid ==0){
      printf("HIJO  %d\n”);
    }else{
      printf ("PADRE %d \n”);

    }
  }

  exit(EXIT_SUCCESS);
}
```

Modifica el código anterior de forma que cada hijo imprima su pid y el pid de su proceso padre.

b) Explica la diferencia entre el código anterior y el siguiente:

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

#define NUM_PROC 3

int main (void)
{
  int pid;
  int i;
  for (i=0; i < NUM_PROC; i++){
    if ((pid=fork()) <0 ){
      printf("Error haciendo fork\n");
      exit(EXIT_FAILURE);
    }else if (pid ==0){
      printf("HIJO %d\n", i);
    }else{
      printf ("PADRE %d\n", i);

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

¿Existen procesos huérfanos en alguno de los dos programas analizados? Al igual que en el código anterior, modifica este programa para que cada proceso hijo imprima su pid y el pid de su proceso padre. De cara a analizar la existencia de procesos huérfanos es de utilidad el comando pstree. Ejecuta pstree para todos pid de procesos padre que obtienes al ejecutar los dos programas de este ejercicio. Analiza la salida que obtienes para cada uno de los casos correspondientes.

Ejercicio 5. (ENTREGABLE) (2,25 puntos)

a) Introduce el mínimo número de cambios en el código del segundo programa del ejercicio de forma que se generen un conjunto de procesos de modo secuencial (cada proceso tiene un único hijo y ha de esperar a que concluya la ejecución de su proceso hijo). Todos los cambios introducidos han de explicarse adecuadamente.

b) Introduce el mínimo número de cambios en el código del segundo programa del ejercicio anterior de forma que exista un único proceso padre que dar lugar a un conjunto de procesos hijo. El proceso padre ha de esperar a que termine la ejecución de todos sus procesos hijo. Todos los cambios introducidos han de explicarse convenientemente.

Ejercicio 6. (ENTREGABLE) (1,5 puntos) Escribe un programa en C (ejercicio6.c) que reserve en el proceso padre memoria dinámica para una cadena de 80 caracteres de longitud y después genere un proceso hijo. Si en el proceso hijo se pide al usuario que introduzca un nombre para guardar en la cadena, ¿el proceso padre tiene acceso a ese valor? ¿Dónde hay que liberar la memoria reservada y por qué?

Ejecución de Programas – Llamadas a sistema con la familia de funciones exec

Las llamadas al sistema exec son una familia de funciones que nos permiten reemplazar el código del proceso actual por el código del programa que se pasa como parámetro.

Normalmente un proceso hijo puede ejecutar un programa diferente al proceso padre, es decir código máquina diferente al del padre. Esto implica que debe invocar a otros programas ejecutables.

Ejercicio 7. (APRENDIZAJE) Estudiar las funciones (#include <unistd.h>):

- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);

Nota: No se debe realizar ningún tipo de tratamiento tras una llamada exec pues dicho código nunca llega a ejecutarse si exec devuelve 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 surgiera un error.

Ejemplo de uso:

```
char *prog[] = { "ls", "-la", NULL };
execvp("ls", prog);
perror("fallo en exec");
exit(EXIT_FAILURE);
```

Ejercicio 8. **(ENTREGABLE)** (1,75 puntos)

Escribe un programa en C (ejercicio8.c) que cree tantos procesos hijo como el proceso padre reciba como argumentos de entrada, y que serán cualquier tipo de programa ejecutable.

Cada proceso hijo mediante, la llamada a una función exec, debe ejecutar cada programa pasado por argumento. Para saber que función exec se debe ejecutar, se pasará una única opción que lo indicará, de la siguiente forma:

- -l: se ejecutará la función execl
- -lp: se ejecutará la función execlp
- -v: se ejecutará la función execv
- -vp: se ejecutará la función execvp

Ejemplo de ejecución del ejercicio8:

- $ ejercicio8 ls df du -l

En este caso, se crearán 3 procesos hijos; el primero, mediante execl, ejecutará el programa "ls"; el segundo, también mediante execl, ejecutará el programa el "df"; y el tercero, también mediante execl, ejecutará el programa el "du".

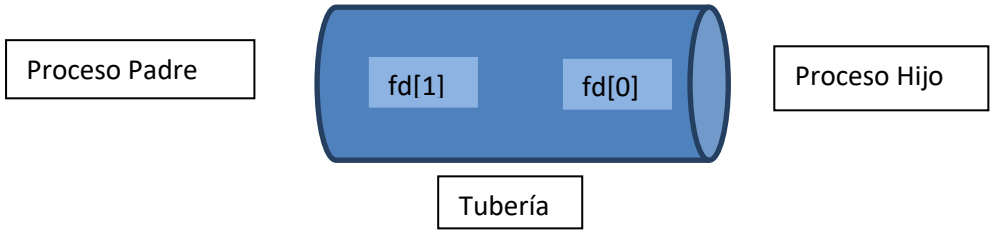
Otros ejemplos:

- $ ejercicio8 ls df du -lp
- $ ejercicio8 ls df du -v
- $ ejercicio8 ls df du -vp

SEMANA 3

Comunicación entre procesos con Tuberías

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 otro de ellos 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.
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.

```
int pipe(int fd[2]);
```

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 (teclado)
1	Salida estándar (pantalla)
2	Salida estándar de errores (pantalla)
. . .	
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 *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.

Si el proceso padre quiere recibir datos del proceso hijo, debe cerrar fd[1], y el proceso hijo debe cerrar fd[0]. Si el proceso padre quiere enviarle datos al proceso hijo, debe cerrar fd[0], y el proceso hijo debe cerrar fd[1]. Como los descriptores se comparten entre el padre e hijo, siempre se debe cerrar el extremo de la tubería que no interesa, nunca se devolverá EOF si los extremos innecesarios de la tubería no son explícitamente cerrados.

Ejemplo de uso:

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

int main(void)
{
        int fd[2], nbytes, pipe_status;
        pid_t childpid;
        char string[] = "Hola a todos!\n";
        char readbuffer[80];
```

```
        pipe_status=pipe(fd);
        if(pipe_status=-1) {
                perror("Error creando la tuberia\n");
                exit(EXIT_FAILURE);
        }

        if((childpid = fork()) == -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 */
                write(fd[1], string, strlen(string));
                exit(0);
        }else{
                /* Cierre del descriptor de salida en el padre */
                close(fd[1]);
                /* Leer algo de la tubería... el saludo! */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("He recibido el string: %s", readbuffer);
        }
        return(0);
}
```

Ejercicio 9. (ENTREGABLE) (2.25 puntos) Escribe un programa en lenguaje C (ejercicio9.c) para que cree 4 procesos hijo, que se van a encargar de realizar las operaciones matemáticas básicas: suma, resta, multiplicación y división. Cada uno de los procesos hijo realizará un operación matemática con los dos operandos que reciba del proceso padre y que este último los deberá obtener por pantalla. El primer proceso hijo realizará la suma de los operandos, el segundo la resta, el tercero la multiplicación y el cuarto, la división.

La comunicación será bidireccional, por lo que el proceso padre enviará a cada uno de los procesos hijo el mensaje compuesto por los dos operandos, separados por comas, por ejemplo: "5,6", y cada proceso hijo enviará al proceso padre el resultado de la operación, incluyendo su PID. Por ejemplo, el primer hijo enviará el mensaje "Datos enviados a través de la tubería por el proceso PID=XX. Operando 1: 5. Operando 2: 6. Suma: 11" (con XX el PID del proceso hijo correspondiente). Por último, será el proceso padre el que muestre por pantalla la información que reciba de los procesos hijos.

El programa debe tener en cuenta: (1) control de errores, (2) cierre de la tuberías pertinentes y (3) espera del proceso padre por sus procesos hijo y en consecuencia los procesos hijo deben enviar al proceso padre su terminación.