Práctica 1

FECHA DE ENTREGA: Semana del 25-28 de febrero
HORA LÍMITE: Una hora antes de la clase de prácticas

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

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

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 (I)

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 *shell* modernas son concebidas como intérpretes de comandos interactivos y como lenguajes de scripting. Las diferencias de sintaxis entre las distintas *shell* 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
```

```
#!/bin/csh
```

```
if [ $days -gt 365 ] then
    echo This is over a year.
fi
```

```
if ( $days > 365 ) then
    echo This is over a year.
fi
```

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. 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/passwd`** haremos ***man 5 passwd***.

```
man -k <palabra_clave> # buscar <palabra clave en todo el manual>
```

```
man n_sec <orden sobre la que obtener ayuda>
man passwd # ayuda sobre la aplicación man
man 5 passwd # ayuda sobre el fichero /etc/passwd
```

Navegación y exploración de archivos

- **cd - Change directory:** Para navegar por el sistema de archivos.
    - **cd <dir>**: nos lleva al directorio dir
    - **cd /**: nos lleva al directorio raíz
    - **cd ~**: nos lleva al directorio del usuario
    - **cd ..**: nos lleva 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, dueño, 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*.

```
cd <dir> # nos lleva al directory <dir> en caso de que exista
cd / # nos lleva al directorio raíz
cd ~ # nos lleva al directorio del usuario
cd .. # nos lleva al directorio que contiene al directorio actual
```

```
ls -l # nos devuelve información detallada de cada fichero (permisos, dueño, tamaño, fecha de creación...)
ls -a # lista también los ficheros ocultos (aquellos cuyo nombre comienza con ".")
```

```
find /home -name tecmint.txt # Busca dentro del directorio /home todos los ficheros que se llamen tecmin.txt
find / -type d -name Tecmint # Busca todos los directorios que llamen Tecmint desde el directorio raíz
```

```
locate mozilla.pdf # Busca todos los ficheros que se llamen mozilla.pdf
```

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.

```
cat <fichero> # Imprime por pantalla el contenido de un fichero
```

```
head -n <fichero> # Imprime por pantalla las n primeras líneas de un fichero
```

```
tail -n <fichero> # Imprime por pantalla las n últimas líneas de un fichero
```

```
less <fichero> # permite visualizar ficheros y buscar en ellos, útil para ficheros grandes
```

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).

```
top # información sobre uso de recursos por procesos
```

```
ps # muestra procesos en ejecución
ps -a # muestra procesos en ejecución tuyos y de otros usuarios
ps -l # muestra más información de cada proceso
```

```
pstree # muestra los procesos como un árbol que relaciona procesos
padres e hijos
```

```
du # muestra en tamaño de los ficheros incluidos en <pwd>
du <dir> # muestra el tamaño de los ficheros incluidos en <dir>
```

```
df -h # muestra la cantidad de espacio libre en las distintas
particiones
```

```
free # muestra la cantidad de RAM y swap disponible
```

Procesos

- **&** 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.

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. 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.

Ejercicio 1. (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 2. (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 3. **(ENTREGABLE)** (1.5 puntos)

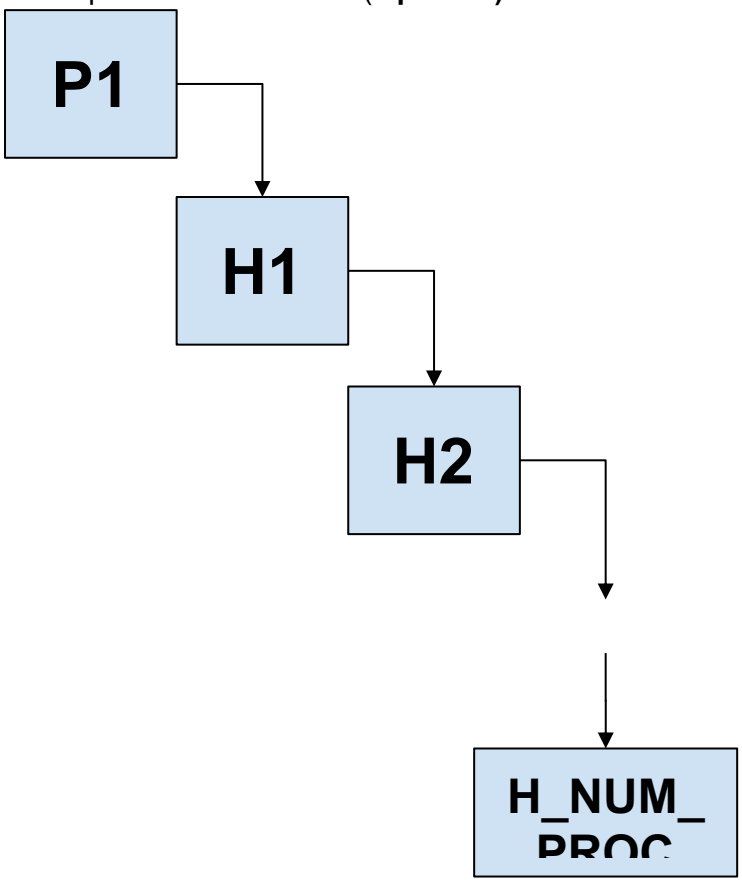
ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

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

#define NUM_PROC 3

int main (void) {
  pid_t pid;
  int i;

  for (i=0; i < NUM_PROC; i++) {
      pid = fork();

      if (pid < 0) {
        printf("Error al emplear fork\n");
        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);
}
```

a) Analiza el texto que imprime el programa. ¿Se puede saber a priori en qué orden se imprimirá el texto ? ¿Por qué? **(0.5 punto)**
b) Cambia el código para que el proceso hijo imprima su **`pid y el de su padre`** en vez de la variable **`i. (0.5 puntos)`**
c) Analiza el árbol de procesos que genera el código de arriba. Muestralo en la memoria como un diagrama de árbol (como el que aparece en el ejercicio 4) explicando porqué es así. **(0.5 punto)**

Ejercicio 4. **(ENTREGABLE)** (1.5 puntos)

a) El código del ejercicio 3 deja procesos huérfanos, ¿Por qué?. **(0.25 puntos)**
b) Introduce el mínimo número de cambios en el código del ejercicio 3 para que no deje procesos huérfanos. **(0.25 punto).**
c) Escribe un programa en C (ejercicio4.c) que genere el siguiente árbol de procesos. El proceso padre genera un proceso hijo que a su vez generará otro hijo así hasta llegar a **`NUM_PROG`** procesos hijos. Asegurate de que cada padre espera a que termine su hijo y no queden procesos huérfanos. (**1 puntos)**



Ejercicio 5. **(ENTREGABLE)** (1 puntos)

ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

```
/* wait and return process info */
```

```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 pid;
  char* sentence = malloc(5*sizeof(char));

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

a) En el programa anterior se reserva memoria en el proceso padre y la inicializa en el proceso hijo usando el método ***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 ejecutamos el código? ¿Es este programa correcto? ¿Por qué? justifica tu respuesta. **(0.5 puntos)**

b) El programa anterior contiene un ***memory leak*** ya que el array ***sentence*** nunca se libera. Corrige el código para eliminar este ***memory leak.*** ¿Dónde hay que liberar la memoria en el padre, en el hijo o en ambos?. Justifica tu respuesta. **(0.5 puntos)**

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.

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.

Ejercicio 6. (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[]);

Ejercicio 7. **(ENTREGABLE) (2 puntos)**

Completa el siguiente fragmento de código para que primero ejecute el comando **`ls -l`** en un proceso hijo, usando la llamada **`execlp`.** A continuación pida al usuario una lista de ficheros de esa carpeta separados por comas(“,”) para luego imprimirlos por pantalla creando un proceso hijo y usando la llamada **`execvp`** para ejecutar el comando **`cat`**.

Ejemplo de ejecución.

Los ficheros **uno.txt** y **dos.txt** tienen el texto “uno” y “dos” respectivamente.

```
total 36
drwxrwxr-x  2 angel angel  4096 ene 24 12:36 .
drwxr-xr-x 25 angel angel  4096 ene 23 17:44 ..
-rwxrwxr-x  1 angel angel 13504 ene 24 12:36 a.out
-rw-rw-r--  1 angel angel     4 ene 23 21:13 dos.txt
-rw-rw-r--  1 angel angel  3571 ene 24 12:31 practica1-ejercio7.c
-rw-rw-r--  1 angel angel     4 ene 23 21:13 uno.txt
Introduzca los ficheros que quiere mostrar separados por ','
uno.txt,dos.txt
uno
dos
```

Esqueleto del Código

ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

```c
/* wait and return process info */
#include <sys/types.h>
#include <sys/wait.h>
/* standard input/output */
#include <stdio.h>
```

```c
/* malloc, free */
#include <stdlib.h> /* library for exec */
#include <unistd.h>
/* for comparing strings */
#include<string.h>

/*
* Pide al usuario una serie de ficheros separados por comas, los almacena en
* un vector de strings, crea un nuevo proceso y ejecuta el comando cat con el
* vector de strings como vector de argumentos
/
void  processCat () {

    /* Variables que usa el metodo getline para leer la entrada del usuario */
    char *fileName = NULL;
    size_t fileLen = 0;
    ssize_t fileRead;

    /* pide al usuario una linea de texto con todos los ficheros separados por
comas */
    printf("Introduzca los ficheros que quiere mostrar separados por ',' \n");
    while((fileRead = getline(&fileName, &fileLen, stdin)) < 1)
    {
        printf("Por favor inserte al menos un fichero \n");
    }

    /* Cuenta el número de ficheros */
    size_t fileCount = 0;
    for(ssize_t i = 0; i < fileRead; i++)
    {
        if(fileName[i] == ',' || fileName[i] == '\n')
        {
                fileCount++;
        }
    }

    size_t nArgs = fileCount + 2;
    /* Reserva espacio para argumentos */
    char ** args = malloc(nArgs * sizeof(*args));
    if(args == NULL)
    {
        exit(EXIT_FAILURE);
    }

    args[0] = "cat";

    char * filePtr = fileName;
    size_t argIndex = 1;
    for(ssize_t i = 0; i < fileRead; i++)
    {
```

```
        if(fileName[i] == ',' || fileName[i] == '\n')
        {
                fileName[i] = '\0';
                args[argIndex] = filePtr;
                argIndex++;
                filePtr = &fileName[i + 1];
        }
    }

    args[nArgs - 1] = NULL;

    if (nArgs >  1) {
        /* METER CODIGO */
        /* Creamos un nuevo proceso hijo y en el ejecutamos execv para ejecutar
el
         comando cat con el vector de argumentos args. El padre debe esperar a
que
         el hijo termine */

    }

    /* Liberamos la memoria dinamica reservada por el proceso */
    free (args);
    /* liberamos la memoria reservada por getline */
    free (fileName);
}

void  showAllFiles () {
    /* METER CODIGO */
   /*
    * Creamos un nuevo proceso hijo usando la llamada execlp y en el ejecutamos
el
    * comando ls -l. El proceso padre debe de esperar a que el hijo termine.
    */
}

int  main(void) {
    showAllFiles();
    processCat();
    exit (EXIT_SUCCESS);
}
```

SEMANA 2

Introducción a la shell (II)

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 esta forma de interpretar el conjunto de operaciones que nos ofrece Unix.

Trabajo con ficheros

- **grep – file pattern search.** 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

```
grep angel paswords.txt # muestra todas las lineas del fichero password.txt que contengan el texto angel
```

```
wc f.txt # muestra el nº de lineas, nº de palabras y numero de bytes del fichero f.txt
```

```
sort  f.txt # muestra por pantalla todas las lineas del fichero f.txt ordenadas
```

```
sort f.txt | uniq # muestra las lineas no repetidas del fichero f.txt
```

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
	[...]	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

Redirección. Procesos y ficheros

- **|** 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.

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.

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 8 (APRENDIZAJE)

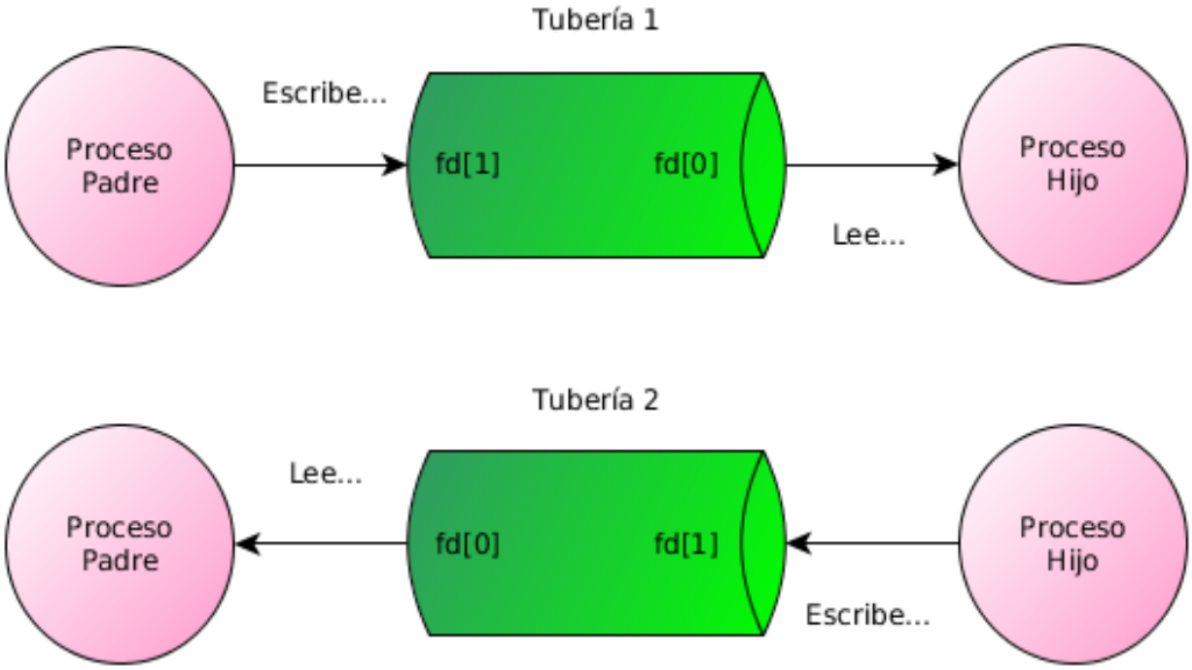
Estudia las siguientes funciones de manejo de ficheros (#include <fcntl.h>):

- 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.

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. La función devuelve -1 en caso de error.

```
int pipe(int fd[2]); //intenta inicializar una tuberia, 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:

ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

```
#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];
  int nbytes, pipe_status;
  pid_t childpid;

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

  pipe_status=pipe(fd);
  if(pipe_status == -1) {
```

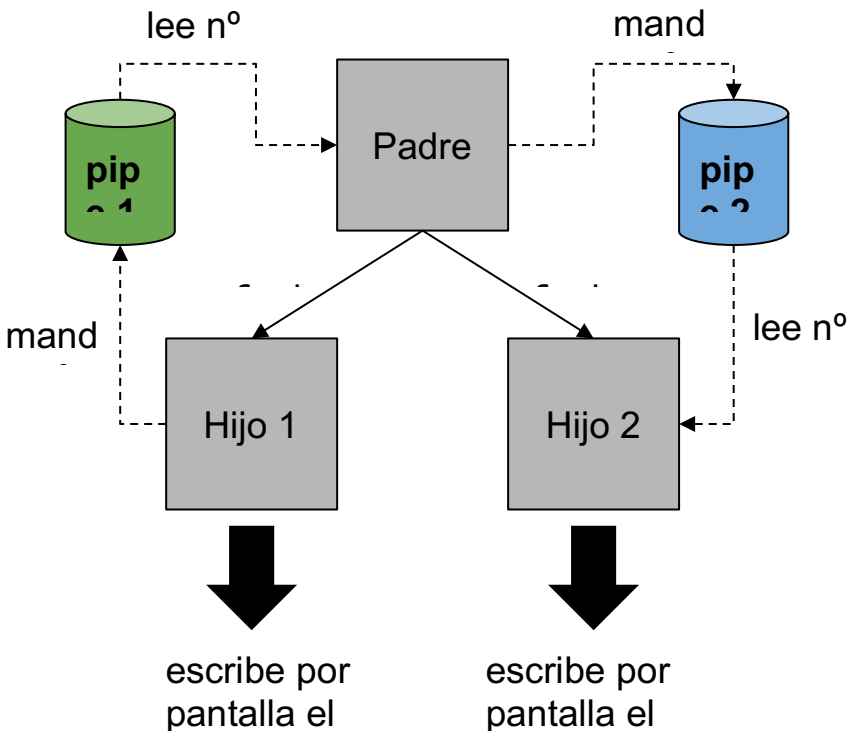
```
        perror("Error creando la tuberia\n");
        exit(EXIT_FAILURE);
    }

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

Ejercicio 9 **(ENTREGABLE) (2 puntos)**

Escribe un programa en lenguaje C (ejercicio9.c) para que cree dos procesos hijos. Mediante pipes el proceso padre se debe comunicar con uno de sus hijos para leerá el ***un número aleatorio*** que generaŕa dicho proceso y lo enviará a través del pipe. El proceso hijo antes de finalizar debe imprimir por pantalla el ***número aleatorio*** que ha generado. Una vez el proceso padre tenga el ***número aleatorio*** del hijo se lo enviará al otro proceso restante a través de un pipe. Este último proceso debe leer el número del pipe e imprimirlo por pantalla.

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.

DIagrama de lo que debe hacer el programa



ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

```c
/*
 * Ejemplo de codigo que genera un numero aleatorio y lo muestra por pantalla
 */
 #include <sys/types.h>
 #include <time.h>
 #include <stdlib.h>
 #include <stdio.h>

int main(int argc, char *argv[]) {
  /* Inicializa el generador con una semilla cualquiera, OJO! este metodo solo
  se llama una vez */
  srand(time(NULL));
  /* Devuelve un numero aleatorio en 0 y MAX_RAND(un número alto que varia
  segun el sistema) */
  int r = rand();
  printf("%d\n", r);

  exit(EXIT_SUCCESS);
}
```

```
# Ejemplo de salida por la consola
HIJO 1: numero aleatorio 128959393
HIJO 2: el numero aleatorio recibido es: 128959393
```

SEMANA 3

Hilos

Los hilos son unidades de trabajo que se pueden expedir para su ejecución. En general, un proceso puede tener asociados varios hilos, que compartirán entre ellos los recursos que el sistema asigne al proceso, permitiendo desarrollar programación concurrente de diversas acciones dentro del proceso.

A diferencia de los procesos, que son completamente independientes (aunque puedan ejecutar el mismo código fuente), los hilos mantienen una relación de dependencia entre ellos que les da ciertas ventajas frente a los procesos: duplican menos recursos, se crean y se destruyen más rápido y pueden comunicarse de forma más rápida e intuitiva ya que comparten el mismo área de memoria (espacio de direcciones).

Los hilos se crean para ejecutar una función concreta, con su propia pila local (stack), donde se crearán las variables automáticas. No obstante, compartirán los recursos asignados por el sistema operativo, como los ficheros abiertos, así como la misma área de memoria dinámica (heap) y las mismas variables globales.

Para escribir programas multihilo en C podemos hacer uso de la biblioteca de hilos pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir la cabecera correspondiente (`#include <pthread.h>`) y a la hora de compilar es necesario enlazar el programa con la biblioteca de hilos, es decir se debe añadir a la llamada al compilador *`gcc`* el parámetro ***`-pthread`***:

Ejercicio 10. (APRENDIZAJE)

Estudia las siguientes funciones de la biblioteca (#include <pthread.h>):

- int pthread_create(pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg);

- void pthread_exit(void *retval);
- int pthread_join(pthread_t thread, void **retval);
- int pthread_detach(pthread_t thread);
- int pthread_cancel(pthread_t thread);
- pthread_t pthread_self(void);

Ejercicio 11. (APRENDIZAJE)

Estudia qué hace el siguiente programa en C.

ATENCIÓN! CUANDO TENGAS QUE EDITAR EL CÓDIGO NO COPIES Y PEGUES EL CÓDIGO DIRECTAMENTE DE ESTE CUADRO YA QUE QUEDARÁ MAL FORMATEADO USA LOS FICHEROS .C QUE ACOMPAÑAN A LA PRÁCTICA

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

void *slowprintf (void *arg) {
   char *msg;
   int i;
   msg = (char *)arg;

   for ( i = 0 ; i < strlen(msg) ; i++ ) {
    printf(" %c", msg[i]);
    fflush(stdout);
    usleep (1000000) ;
   }
   pthread_exit(NULL);
}

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

   pthread_create(&h1, NULL , slowprintf , (void *)hola);
   pthread_create(&h2, NULL , slowprintf , (void *)mundo);

   pthread_join(h1,NULL);
   pthread_join(h2,NULL);

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

¿Qué hubiera pasado si el proceso no hubiera esperado por los hilos? Para probarlo elimina las dos líneas de la llamada a la función ***int pthread_join(pthread_t thread, void **retval)***.

Ejercicio 12 **(ENTREGABLE) (2ptos)**

Paso de parámetros en hilos. Crea un programa en C (ejercicio12.c) que cree `N` hilos (`N` puede definirse en el código usando una directiva `#define`). Cada hilo realizará el cálculo 2^x , dónde `X` será un parámetro que reciba el hilo. Además del parámetro de entrada `X` el hilo debe recibir un parámetro de salida en el que guardar el resultado de la operación. El padre de los hilos debe esperar a que todos 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.