# Práctica 2: Programación en POSIX *

Programación y Administración de Sistemas

2018-2019

Juan Carlos Fernández Caballero
jfcaballero@uco.es

2º curso de Grado en Ingeniería Informática
Departamento de Informática y Análisis Numérico
Escuela Politécnica Superior
Universidad de Córdoba

Febrero de 2019

## Índice



*Parte de los contenidos de este guión corresponden al preparado por Javier Sánchez Monedero en el curso académico 2011/2012 [1] y por los sucesivos profesores de prácticas (Pedro Antonio Gutiérrez Peña, Juan Carlos Fernández Caballero, David Guijo Rubio).

# 1. Introducción

POSIX [1] es el acrónimo de *Portable Operating System Interface*; la *X* viene de UNIX como señal de identidad de la API (*Application Programming Interface*, interfaz de programación de aplicaciones). Son una familia de estándares de llamadas al sistema operativo (*wrappers*) definidos por el IEEE (*Institute of Electrical and Electronics Engineers*, Instituto de Ingenieros Eléctricos y Electrónicos) y especificados formalmente en el IEEE 1003. Persiguen generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas. De esta forma, si una aplicación hace un buen uso de estas funciones deberá compilar y ejecutarse *sin problemas* en cualquier sistema operativo que siga el estandar POSIX.

Estos estándares surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces a nivel de aplicación adaptables a una gran variedad de implementaciones de sistemas operativos [2]. La última versión de la especificación POSIX es del año 2008 y se le conoce como "POSIX.1-2008", "IEEE Std 1003.1-2008" y "The Open Group Technical Standard Base Specifications, Issue 7" [3].

POSIX recoge al Estandar de C, también nombrado como ANSI C o ISO C, el cual ha ido evolucionando a lo largo de los años [2], es decir, mientras que el estandar de C aporta un conjunto de definiciones, nomenclaturas, ficheros de cabecera y bibliotecas con rutinas básicas que debería implementar todo sistema operativo que siga dicho estandar, POSIX es una ampliación de lo anterior, aportando más rutinas y más ficheros de cabecera, lo cual amplia la funcionalidad de un sistema.

Cuando hablamos de Linux como sistema operativo completo debemos referirnos a él como "GNU/Linux" para reconocer que **el sistema lo compone tanto el núcleo Linux como las bibliotecas de C y otras herramientas de GNU** que hacen posible que exista como sistema operativo [3]. GNU (**G**NU's **N**ot **U**nix) es el nombre elegido para sistemas que siguen un diseño tipo Unix y que se mantiene compatible con éste, pero se distinguen de Unix por ser software libre y por no contener código de Unix (que era privativo).

*GNU C Library*, comúnmente conocida como `glibc` [4], es una biblioteca en lenguaje C para sistemas GNU que implementa el estandar POSIX, por lo que incluye a su vez la implementación del estandar de C. Por tanto la biblioteca *glibc* sigue en su implementación todos los estándares más relevantes, ANSI C y POSIX.1-2008 [4]. *glibc* se distribuye bajo los términos de la licencia GNU LGPL[5]. Decir también que la implementación del estandar de C se encuentra en una biblioteca llamada `libc` [6].

`glibc` es muy *portable* y soporta gran cantidad de plataformas de *hardware* [5]. En los sistemas GNU/Linux se instala normalmente con el nombre de `libc6`. No debe confundirse con GLib[7], otra biblioteca que proporciona estructuras de datos avanzadas como árboles, listas enlazadas, tablas hash, etc, y un entorno de orientación a objetos en C. Algunas distribuciones de GNU/Linux como Debian o Ubuntu, utilizan una variante de `glibc` llamada

[1] https://en.wikipedia.org/wiki/POSIX
[2] https://en.wikipedia.org/wiki/ANSI_C
[3] http://es.wikipedia.org/wiki/Controversia_por_la_denominaci%C3%B3n_GNU/Linux
[4] https://es.wikipedia.org/wiki/Glibc
[5] http://es.wikipedia.org/wiki/GNU_General_Public_License
[6] https://es.wikipedia.org/wiki/Biblioteca_est%C3%A1ndar_de_C
[7] http://library.gnome.org/devel/glib/,http://es.wikipedia.org/wiki/GLib

`eglibc`[8], adaptada para sistemas empotrados, pero a efectos de programación no hay diferencias.

A modo de resumen, es importante no confundir a POSIX con un lenguaje de programación, ya que es un estándar que siguen (implementan) bibliotecas como *glibc* (incluye a *libc*), y no un lenguaje como tal.

Consulte los enlaces proporcionados en la práctica y en el propio Moodle para discernir y diferenciar entre los terminos que se acaban de exponer.

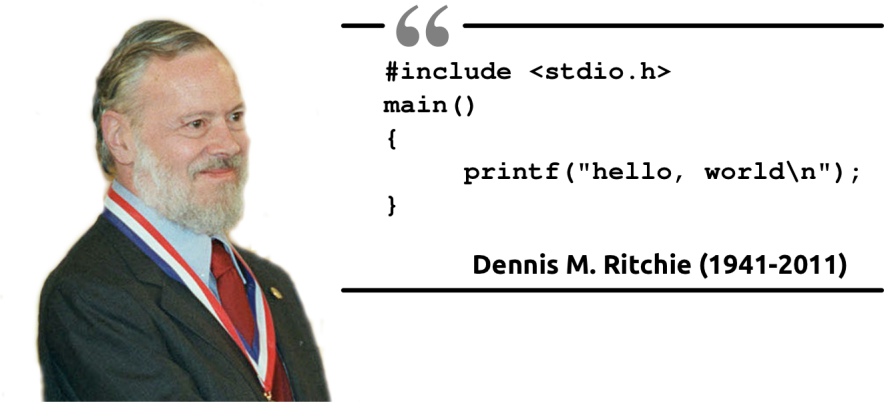
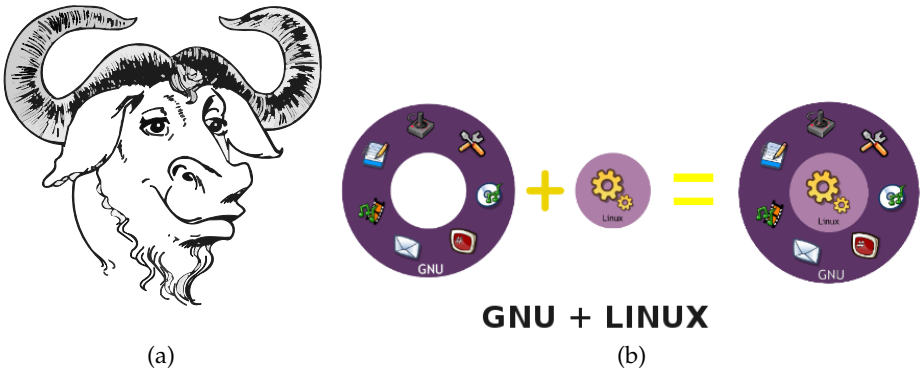



Figura 1: **Dennis MacAlistair Ritchie**. Colaboró en el diseño y desarrollo de los sistemas operativos Multics y Unix, así como el desarrollo de varios lenguajes de programación como el C, tema sobre el cual escribió un célebre clásico de las ciencias de la computación junto a Brian Wilson Kernighan: *El lenguaje de programación C* [6].




(a) (b)

Figura 2: (a) Mascota del proyecto GNU (`http://www.gnu.org/`). (b) GNU + Linux = GNU/Linux.

[8] `http://www.eglibc.org/home`

## 2. Objetivos

Los objetivos que se persiguen en esta práctica son los siguientes:

- Conocer algunas rutinas POSIX relacionadas con la temática concreta de esta práctica, y su implementación `glibc`.
- Aprender a utilizar bibliotecas externas en nuestros programas y a consultar su documentación asociada.
- Aprender cómo funcionan internamente algunas partes de GNU/Linux.
- Mejorar la programación viendo ejemplos hechos por los desarrolladores de las bibliotecas.
- Aprender a cómo gestionar el procesado de la linea de argumentos de un programa.
- Aprender a utilizar variables de entorno e información de los usuarios del sistema.
- Aprender a comunicar aplicaciones utilizando paso de mensajes.

  En la asignatura de Sistemas Operativos ya estudió algunas **IPC** (Inter-Process Communication) o formas de comunicar y/o sincronizar procesos e hilos, concretamente los semáforos y el uso de memoria compartida. En esta práctica ampliará esos conocimientos con:

  1. Tuberías o *pipes*.
  2. Colas de mensajes.

## 3. Directrices

Tenga en cuenta las siguientes directrices:

- En Moodle se adjunta el fichero `codigo-ejemplos.zip`, que contiene código de ejemplo de las funciones que se irán estudiando.
- No hay que entregar obligatoriamente los programas propuestos en los **Ejercicios Resumen**, ya que no se someterán a evaluación, pero es aconsejable que los realice todos, los comprenda perfectamente e incluso haga modificaciones y mejoras propias, ya que tendrá que examinarse en ordenador para superar las prácticas. La asistencia y la realización de las prácticas es fundamental para la preparación de los examenes en ordenador.
- Acostumbrese a una buena modularidad del código en funciones, a la comprobación de errores en los argumentos de los programas y la claridad en las salidas generadas, es fundamental para generar programas de calidad, tanto para superar la asignatura como para su trabajo como Ingeniero Informático.
- Todos los programas deben funcionar correctamente en la máquina `ts.uco.es`, ya que es ahí donde se examinará. Compruebe que los comportamientos de los programas son similares a los esperados en los ejemplos de ejecución.

- A vista de los examenes en ordenador, para que un ejercicio se corrija es absolutamente necesario que: 1) Compile correctamente, sin errores. 2) Ejecute correctamente, aportando la salida esperada, usando las técnicas y conceptos que se han estudiado durante la asignatura, y no otros. El alumnado debe tener claro que a partir de que se cumplan los items anteriores, el profesorado otorgará a un ejercicio más o menos puntuación dependiendo de: Control de errores utilizado, invocación y uso correcto de las funciones, identación y claridad de la programación.

- **Documentación POSIX.1-2008:** Especificación del estándar POSIX. Dependiendo de la parte que se documente es más o menos pedagógica[9]. Téngala siempre en cuenta y consúltela a lo largo de la práctica, es lo que tendrá como documentación en los examenes en ordenador, junto con la ayuda de *man*.

- **Documentación *GNU C Library* (`glibc`)**: Esta documentación incluye muchos de los conceptos que ya ha trabajado en asignaturas de Introducción a la programación o de Sistemas operativos. Es una guía completa de programación en el lenguaje C, pero sobre todo incluye muchas funciones que son esenciales para programar, útiles para ahorrar tiempo trabajando o para garantizar la portabilidad del código entre sistemas POSIX[10]. Recuerde que *glibc* sigue el estándar Posix nombrado anteriormente, es decir, lo implementa.

# 4. Procesado de línea de comandos tipo POSIX

## 4.1. Introducción y documentación

Los parámetros que procesa un programa en sistemas POSIX deben seguir un estándar de formato y respuesta esperada[11]. Un resumen de lo definido en el estándar es lo siguiente:

- Una opción es un guión (-) seguido de un carácter alfanumérico, por ejemplo, `-o`.

- En una opción que requiere un parámetro, este debe aparecer inmediatamente después de la opción, por ejemplo, `-o parámetro` o `-oparámetro`.

- Las opciones que no requieren parámetros pueden agruparse detrás de un guión, por ejemplo, `-lst` es equivalente a `-t -l -s`.

- Las opciones pueden aparecer en cualquier orden, así `-lst` es equivalente a `-tls`.

La función `getopt()` del estándar [12] [13] ayuda a desarrollar el manejo de las opciones siguiendo las directrices POSIX.1-2008.

Puede consultar la sección *Using the `getopt` function* de la documentación[14] para saber cómo funciona `getopt`, qué valores espera y qué comportamiento tiene. También puede ver

[9] http://pubs.opengroup.org/onlinepubs/9699919799/

[10] http://www.gnu.org/software/libc/manual/

[11] 12.1 Utility Argument Syntax, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

[12] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getopt.html

[13] http://www.gnu.org/software/libc/manual/html_node/Getopt.html

[14] http://www.gnu.org/software/libc/manual/html_node/Using-Getopt.html

un código de ejemplo [15] simple dentro del fichero `ejemplo-getopt.c` de los que hay en Moodle.

Por otro lado, para permitir especificar opciones en formato corto o largo (por ejemplo, `--help` y `-h` como órdenes compatibles), se dispone en *glibc* de la función *getopt_long()* [16]. Esta función no está descrita en POSIX, pero la implementa *glibc* y por lo tanto los sistemas GNU/Linux que la usan. En el fichero `ejemplo-getoptlong.c` contiene un ejemplo [17] de procesado de órdenes al estilo de GNU, le será útil para los ejercicios de la práctica.

No olvide **consultar todos los enlaces que aparecen en las notas al pie antes de continuar**. Estos enlaces contienen la información tanto a nivel teórico como a nivel práctico, a partir de la cual podrá implementar sus ejercicios en C y prepararse para los examenes en ordenador.

### 4.2. Ejercicio

Lea el código del fichero `ejemplo-getopt.c`, compílelo y ejecútelo para comprobar que admite las opciones de parámetros POSIX. Trate de entender el código (consultando los enlaces proporcionados en los pie de página) y añada más opciones (por ejemplo una sin parámetros y otra con parámetros) y modificaciones que se le ocurran para entender su comportamiento.

## 5. Variables de entorno

### 5.1. Introducción y documentación

Una variable de entorno es un objeto designado para contener información usada por una o más aplicaciones. La variables de entorno se asocian a toda la máquina, pero también a usuarios individuales.

Si utiliza `bash`, puede consultar las variables de entorno de su sesión con el comando `env`. También puede consultar o modificar el valor de una variable de forma individual para la sesión actual:

```
1 $ env
2 $ ...
3 $ echo $LANG
4 es_ES.UTF-8
5 $ export LANG=en_GB.UTF-8
```

En el fichero `ejemplo-getenv.c` hay un ejemplo de uso de la función `getenv()` [18]. Este programa, dependiendo del idioma de la sesión de usuario, imprime un mensaje con el nombre de su carpeta personal en castellano o en inglés.

[15] http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html
[16] https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html
[17] https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Option-Example.html
[18] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getenv.html

# 6. Obtención de información de un usuario

## 6.1. Introducción y documentación

En los sistemas operativos, la base de datos de usuarios que hay en el sistema puede ser local y/o remota. En GNU/Linux puede ver los usuarios y grupos locales en los siguientes ficheros (consulte los enlaces antes de continuar):

- Mantiene información sobre los usuarios: `/etc/passwd`
- Mantiene información sobre los grupos: `/etc/group` [19].

A modo de información, si los usuarios no son locales, normalmente se encuentran en una máquina remota a la que se accede por un protocolo específico. Algunos ejemplos son el servicio de información de red (NIS, *Network Information Service*) o el protocolo ligero de acceso a directorios (LDAP, *Lightweight Directory Access Protocol*). En la actualizad NIS se usa en entornos exclusivos UNIX y LDAP es el estándar para autenticar usuarios tanto en sistemas Unix o GNU/Linux, como en sistemas Windows.

En el caso de GNU/Linux, la autenticación local de usuarios la realizan los módulos de autenticación PAM (*Pluggable Authentication Module*). PAM es un mecanismo de autenticación flexible que permite abstraer las aplicaciones del proceso de identificación. La búsqueda de su información asociada la realiza el servicio NSS (*Name Service Switch*), que provee una interfaz para configurar y acceder a diferentes bases de datos de cuentas de usuarios y claves como `/etc/passwd`, `/etc/group`, `/etc/hosts`, LDAP, etc.

POSIX presenta una interfaz para el acceso a la información de usuarios que abstrae al programador de dónde se encuentran los usuarios (en bases de datos locales y/o remotas, con distintos formatos, etc.). Puede ver las funciones y estructuras de acceso a la información de usuarios y grupos en los siguientes ficheros de cabecera:

- Funciones y estructuras de acceso a la información de usuarios:
  `/usr/include/pwd.h` [20]
- Funciones y estructuras de acceso a la información de grupos:
  `/usr/include/grp.h` [21]

## 6.2. Ejercicio

La llamada `getpwuid()` devuelve una estructura con información de un usuario previo paso de su `uid` como parámetro. La implementación POSIX de esta función se encarga de intercambiar información con NSS para conseguir la información del usuario. NSS leerá ficheros en el disco duro o realizará consultas a través de la red para conseguir esa información.

Por otro lado, la función `getpwnam()` [22] devuelve una estructura con información de un usuario previo paso de su `login` como parámetro.

[19] gestión de usuarios y grupos en GNU/Linux

[20] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pwd.h.html

[21] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/grp.h.html

[22] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getpwnam.html
http://www.gnu.org/software/libc/manual/html_node/User-Database.html

Con respecto a los grupos, la llamada a la función `getgrgid()` [23] [24], devuelve una estructura con información de un grupo previo paso de su `gid` como parámetro; y la función y `getgrnam()` [25] devuelve una estructura con información de un grupoprevio paso de su `nombre` de grupo como parámetro respectivamente.

Estudie el programa `ejemplo-infousuario.c`, es un ejemplo de implementación que utiliza las funciones que se acaban de nombrar. Ejecútelo y haga los cambios que considere oportunos para entender su funcionamiento. Puede ampliarlo para para utilizar `getgrgid()` para obtener el nombre del grupo del usuario a través del identificador del grupo.

En el programa `ejemplo-infousuario.c` verá el uso de la función `getlogin()`. Dicha función puede tener comportamientos inesperados, por ejemplo en la UCO devuelve el usuario por defecto que usa el terminal, pero en otros sistemas puede que no sea así (problemas a la hora de mirar un fichero que aloja el usuario asociado a los terminales). Modifique el programa de forma que pueda obtener el `login` de un usuario de otra manera.

## 7. Ejercicio resumen 1

El fichero de código de este ejercicio será `ejercicio1.c` y el ejecutable `ejercicio1`. Implemente un programa que obtenga e imprima información sobre usuarios del sistema (*todos* los campos de la estructura `passwd`) e información sobre grupos del sistema (*GID* y *nombre del grupo* mediante la estructura `group`), según las opciones recibidas por la línea de argumentos.

- La opción `-u/--username` servirá para especificar el nombre de un usuario del sistema (p.ej. `jfcaballero`) del cual hay que mostrar la información correspondiente a su estructura `passwd`.
- La opción `-i/--useruid` servirá para especificar el identificador de un usuario del sistema (p.ej. `17468`) del cual hay que mostrar la información correspondiente a su estructura `passwd` .
- La opción `-g/--groupname`, servirá para especificar el nombre de un grupo del sistema (p.ej. `adm`) del cual hay que mostrar la información correspondiente a su estructura `group` (GID).
- La opción `-d/--groupuid`, servirá para especificar el identificador de un grupo del sistema (p.ej. `4`) del cual hay que mostrar la información correspondiente a su estructura `group` (Nombre).
- Si se invoca al programa con la opción `-s` o con `--allgroups`, se mostrarán todos los grupos del sistema, junto con su identificador. Para ello recorra el fichero correspondiente (le permitirá recordar como gestionar y buscar en cadenas) y luego vaya extrayendo información como si se invocase la opción `--groupname` o `--groupuid`. No muestre el contenido del fichero, recorralo y muestre la información por cada grupo que haya.

[23] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getgrgid.html
[24] http://www.gnu.org/software/libc/manual/html_node/Group-Database.html
[25] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getgrnam.html

- La opción `-a/--allinfo` servirá para especificar el nombre de un usuario del sistema (p.ej. `jfcaballero`) del cual hay que mostrar la información correspondiente a su estructura `passwd`, y además mostrará la información correspondiente a su estructura `group` del grupo al cual pertenece ese usuario.
- La opción `-b/--bactive`, deberá buscar e imprimir la información del grupo del usuario ACTUAL (GID del grupo y nombre del grupo).
- Si se invoca al programa sin ninguna opción se mostrará la información del usuario actual y del grupo al que pertenece. Si la variable de entorno LANG estuviera en Ingles, la información se mostrará en Inglés.
- Se creará una opción de ayuda `-h/--help` para mostrar información sobre cada uno de los usos del programa. Esa información también se mostrará cuando el usuario cometa **cualquier error en la invocación del programa**.

Tenga también en cuenta el siguiente control de errores:

- Asegurar que se pasan nombres e identificadores de usuarios o grupos válidos que existan en la máquina.
- Asegurar que no se puedan pasar por linea de comandos opciones que sean incompatibles, por ejemplo, las opciones `-u` y `-i` no pueden activarse a la vez, o las opciones `-u` y `-a`
- Asegurar que las opciones tengan el numero correcto de argumentos (hay opciones que no necesitan argumentos, por ejemplo `-b/--bactive`, y otras que si). Haga un control de errores tan exahustivo como considere oportuno.

Ejemplos de llamadas válidas serían la siguientes:

```
# Obtener la información del grupo del usuario actual
jfcaballero@NEWTS:~$ ./ejercicio1 -b
Main Group Number: 1000
Main Group Name: jfcaballero

# Obtener la información del usuario actual
jfcaballero@NEWTS:~$ ./ejercicio1
Nombre de usuario: Juan Carlos Fernández Caballero,,,
Identifador de usuario: 1000
Contraseña: x
Carpeta inicio: /home/jfcaballero
Intérprete por defecto: /bin/bash
Login de usuario:jfcaballero
Numero de grupo: 1000
Nombre de grupo: jfcaballero

jfcaballero@NEWTS:~$ ./ejercicio1 --groupname adm
Main Group Number: 4

# Llamadas incorrectas
jfcaballero@NEWTS:~$ ./ejercicio1 -u jfcaballero -i 1000
No se pueden activar estas dos opciones a la vez.
Uso del programa: ejercicio1 [opciones]
Opciones:
```

```
-h, --help                    Imprimir esta ayuda
-u, --username          Nombre de Usuario
-i, --useruid           Identificador de Usuario
-g, --groupname             Nombre de Grupo
-d, --groupuid                  Identificador de Grupo
-s, --allgroups                 Muestra info de todos los grupos del sistema
-a, --allinfo               Nombre de Usuario
-b, --bactive               Muestra info grupo usuario Actual
```

# 8. Creación de procesos (fork y exec)

## 8.1. Introducción y documentación

En general, en sistemas operativos y lenguajes de programación, se llama *bifurcación* o *fork* a la creación de un subproceso copia del proceso que llama a la función. El subproceso creado, o "proceso hijo", proviene del proceso originario, o "proceso padre". Los procesos resultantes son idénticos, salvo que tienen distinto número de proceso (PID) [26].

En GNU/Linux, esto ocurre al crear cualquier proceso, por ejemplo, al utilizar tuberías o *pipes* desde la terminal, las cuáles son esenciales para la comunicación inter-procesos. Así, el siguiente comando mostraría el contenido del fichero *ejemplo-fork.c* y la salida serviría de entrada para el proceso *wc*, que contaría las lineas mostradas en la salida.

```
$  cat ./ejemplo-fork.c | wc -l
```

A nivel de programación, en C se crea un subproceso llamando a la función `fork()` [27] [28]. Tiene un pequeño manual y mucho código de ejemplo en [7].

El nuevo proceso hereda muchas propiedades del proceso padre (variables de entorno, descriptores de ficheros, etc. ). Después de una llamada *exitosa* a `fork`, habrá dos copias del código original ejecutándose a la vez (o multiplexando si se tiene un solo procesador con un solo núcleo o las condiciones del sistema no permiten la ejecución en paralelo).

En el proceso original, el valor devuelto de `fork` será el identificador del proceso hijo, sin embargo, en el proceso hijo el valor devuelto por `fork` será `0`.

El fichero (`ejemplo-fork.c`) muestra un ejemplo de uso de `fork` que controla qué proceso es el que ejecuta determinada parte del código, usando funciones POSIX para obtener información de los procesos (puede ver un esquema de los subprocesos creados en la Figura 3). Un ejemplo de la salida de la ejecución de ese código sería:

```
$ ./ejemplo-fork
Soy el padre, mi PID es 23455 y el PID de mi hijo es 23456
Soy el hijo, mi PID es 23456 y mi PPID es 23455
Final de ejecución de 23456
Final de ejecución de 23455
```

[26] http://www.gnu.org/software/libc/manual/html_node/Processes.html

[27] http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html

[28] Puede ver un código con muchos comentarios en la siguiente entrada de Wikipedia http://es.wikipedia.org/wiki/Bifurcaci%C3%B3n_%28sistema_operativo%29

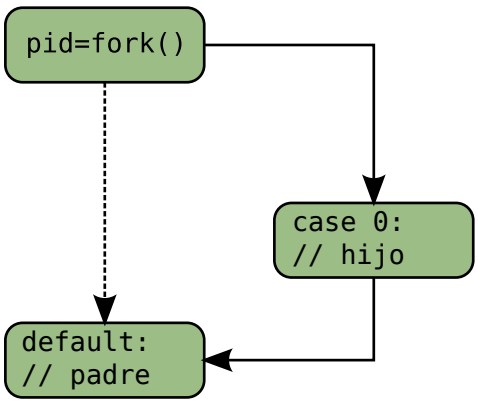



Figura 3: Esquema de llamadas y procesos generados por `fork()` en el ejemplo.

Un proceso padre debe esperar a que un proceso hijo termine, para ello se utiliza la función `wait` y `waitpid` [29] [30]. El valor devuelto por la función `waitpid` es el PID del proceso hijo que terminó y recogió el padre. El estado de terminación del proceso (código de error), se recoge en la variable `status` pasada como argumento.

En ocasiones puede interesar ejecutar un programa distinto, no diferentes partes de él, y se quiere iniciar este segundo proceso diferente desde el programa principal. La familia de funciones `exec()` [31] permiten iniciar un programa dentro de otro programa. En lugar de crear una copia del proceso, `exec()` provoca el reemplazo total del programa que llama a la función por el programa llamado. Por ese motivo se suele utilizar `exec()` junto con `fork()`, de forma que sea un proceso hijo el que cree el nuevo proceso para que el proceso padre no sea destruido. Puede ver un ejemplo en el fichero `ejemplo-fork-exec.c`

### 8.2. Ejercicio

Ejecute, estudie y modifique los ficheros de ejemplo `ejemplo-fork.c` y `ejemplo-fork-exec.c`, hasta que comprenda totalmente su comportamiento y el uso de las funciones Posix que en ellos se utilizan.

## 9. Señales entre procesos

### 9.1. Introducción y documentación

Las señales [32] entre programas son interrupciones *software* que se generan para informar a un proceso de la ocurrencia de un evento. Otras formas alternativas de comunicación entre procesos son las que veremos en la sección 10.

Los programas pueden diseñarse para capturar una o varias señales proporcionando una función que las maneje. Este tipo de funciones se llaman técnicamente *callbacks* o *retrollamadas*. Una *callback* es una referencia a un trozo de código ejecutable, normalmente una función, que se pasa como parámetro a otro código. Esto permite, por ejemplo, que una capa de bajo

[29] www.gnu.org/software/libc/manual/html_node/Process-Completion.html
[30] http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html
[31] http://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html
[32] http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html

nivel del *software* llame a la subrutina o función definida en una capa superior (ver Figura 4, fuente Wikipedia[33]).

Por ejemplo, cuando se apaga GNU/Linux, se envía la señal SIGTERM a todos los procesos, así los procesos pueden capturar esta señal y terminar de forma adecuada (liberando recursos, cerrando ficheros abiertos, etc.).

La función `signal`[34] permite asociar una determinada función (a través de un puntero a función) a una señal identificada por un entero (SIGTERM, SIGKILL, etc.).

```
#include <signal.h>

// El prototipo de la función manejadora es el siguiente
// sighandler_t signal(int signum, sighandler_t handler);
// sighandler_t representa un puntero a una función que devuelve
// void y recibe un entero
...

// Función que va a manejar la señal TERM
void mifuncionManejadoraTerm(int signal)
{
  ....
}

int main(void){

  ...
  // Vinculacion de la señal concreta SIGTERM a una funcion
     manejadora
  signal(SIGTERM, mifuncionManejadoraTerm);
  // Donde SIGTERM es 15, y mifuncionManejadoraTerm es un manejador
      de la señal, un puntero a función
  ...

}
```

## 9.2. Ejercicio

El código del fichero `ejemplo-signal.c`[35] contiene ejemplos de captura de señales POSIX enviadas a un programa. Recuerde que la función `signal()` no llama a ninguna función, lo que hace es asociar una función del programador a eventos que se generan en el sistema, esto es, pasar un puntero a una función. Modifique el código de este programa hasta que entienda completamente su funcionamiento y el de las funciones que se utilizan.

A continuación se muestra un ejemplo de ejecución del programa `ejemplo-signal.c`, al que se le mandan las señales SIGHUP y SIGTERM desde otro terminal. Lo primero que se muestra es que no se puede capturar la señal KILL:

[33] http://en.wikipedia.org/wiki/Callback_%28computer_science%29
[34] http://pubs.opengroup.org/onlinepubs/9699919799/functions/signal.html
[35] Adaptado de http://www.amparo.net/ce155/signals-ex.html

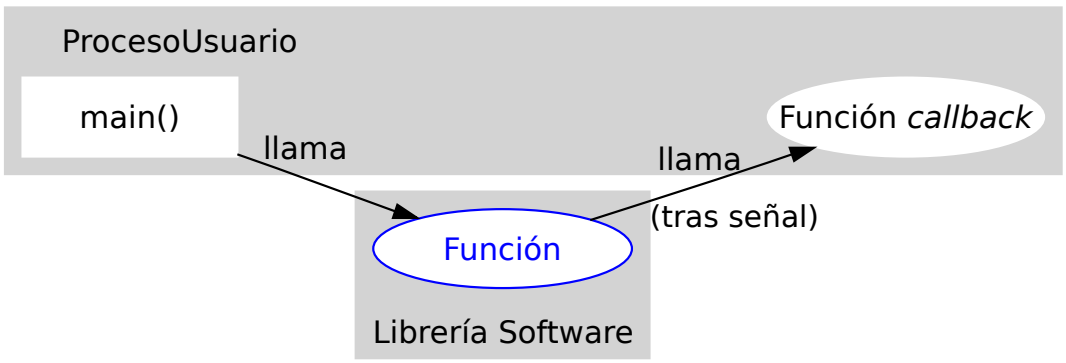



Figura 4: Esquema del funcionamiento de las *callbacks* o *retrollamadas*.

```
$  ./ejemplo-signal
No puedo asociar la señal SIGKILL al manejador!
Capturé la señal SIGHUP y no salgo!
Capturé la señal SIGTERM y voy a salir de manera ordenada
Hasta luego... cerrando ficheros...
Hasta luego... cerrando ficheros...
Hasta luego... cerrando ficheros...
```

En otro terminal podemos consultar rápidamente el `id` de nuestro proceso y enviarle las señales SIGHUP y SIGTERM con los siguientes comandos, de manera que se reproduciría la salida del ejemplo de ejecución anterior:

```
$  ps -a
PID TTY          TIME CMD
737 tty1     00:00:00 syslog-ng
1414 tty1     00:00:00 xrdp
1416 tty1     00:00:00 xrdp-sesman
1826 tty1     00:00:00 bash
19774 pts/47   00:00:00 ejemplo-signal
19993 pts/52   00:00:00 ps

$ kill -SIGHUP 19774
$ kill -SIGTERM 19774
```

En el fichero `ejemplo-signal-division.c` se muestra un programa que con dos números calcule la división del primero entre el segundo. Dicho programa captura la excepción de división por cero (sin comprobar que el segundo argumento es cero) y, en el caso de que la haya, divide por uno. Modifique el código de este programa hasta que entienda completamente su funcionamiento y el de las funciones que se utilizan.

```
$ ./ejemplo-signal-division
Introduce el dividendo: 1
Introduce el divisor: 2
Division=0
$ ./ejemplo-signal-division
```

```
Introduce el dividendo: 1
Introduce el divisor: 0
Capturé la señal DIVISIÓN por cero
Division=1
```

# 10. Comunicación entre procesos POSIX

El estándar POSIX contempla distintos mecanismos de comunicación entre varios procesos que están ejecutándose en un sistema operativo. Todos los mecanismos de comunicación entre procesos se recogen bajo el término *InterProcess Communication* (IPC), de forma que el POSIX IPC hereda gran parte de sus mecanismos del System V IPC (que era la implementación propuesta en Unix).

Los mecanismos IPC fundamentales son:

- Semáforos.
- Memoria compartida.
- **Tuberías (*pipes*).**
- **Colas de mensajes.**

## 10.1. Semáforos

Un semáforo es, básicamente, una variable entera (contador) que se mantiene dentro del núcleo del sistema operativo. El núcleo bloquea a cualquier proceso que intente decrementar el contador por debajo de cero. Los incrementos nunca bloquean al proceso. Esto permite realizar una sincronización entre los distintos procesos.

Los semáforos ya se han estudiado en la asignatura de Sistemas Operativos y, por tanto, no se tratarán en esta práctica, pero es importante tener en cuenta que también están especificados en el estándar POSIX.

## 10.2. Memoria compartida

Este tipo de comunicación implica que dos procesos del sistema operativo van a compartir una serie de páginas de la memoria principal. Esto permite que la comunicación se limite a copiar datos a y leer datos de dicho fragmento de memoria. Es un mecanismo muy eficiente, ya que cualquier otro mecanismo hace que tengamos que realizar cambios de contexto (modo usuario $\Rightarrow$ modo núcleo $\Rightarrow$ modo usuario). Como contrapartida, se debe realizar una sincronización de las lecturas y escrituras y usar semáforos, que implica una mayor dificultad en la programación.

La memoria compartida ya se ha estudiado en la asignatura de Sistemas Operativos y, por tanto, no se tratarán en esta práctica, pero es importante tener en cuenta que también están especificados en el estándar POSIX.

## 10.3. Tuberías

Las tuberías son ficheros temporales que actúan como *buffer* y en los que se pueden enviar y recibir una secuencia de *bytes*. Una tubería es de **una sola dirección** (de forma que un proceso escribe sobre ella y otro proceso lee el contenido) y no permite *acceso aleatorio*.

Por ejemplo, el comando:

```
$ ls | wc -l
44
```

conecta la salida de `ls` con la entrada de `wc`, tal y como se indica en la Figura 5.

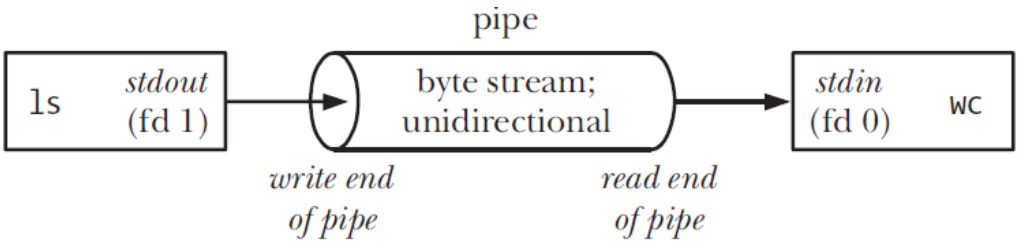



Figura 5: Intercomunicación entre procesos utilizando la tubería `ls | wc -l`. "*write end*" significa extremo de escritura y "*read end*" extremo de lectura.

Existen dos tipos de tuberías: **tuberías anónimas** y **tuberías con nombre**. La tubería que se ha visto en el ejemplo anterior sería una **tubería anónima**, ya que se crea desde `bash` de forma temporal para intercomunicar dos procesos.

Podemos crear tuberías anónimas en un programa en C mediante la función `pipe` de `unistd.h`[36]:

```
#include <unistd.h>
int pipe(int fildes[2]);
```

Esta función crea una tubería anónima y devuelve (por referencia, en el vector que se pasa como argumento) dos descriptores de fichero ya abiertos, uno para **leer (`fildes[0]`)** y otro para **escribir (`fildes[1]`)**.

Para leer o escribir en dichos descriptores, utilizaremos las funciones `read`[37] y `write`[38], cuyo uso es similar a `fread` y `fwrite`.

Una vez utilizados los extremos de lectura y/o escritura, los podemos cerrar con `close`[39].

En el esquema [40] que se muestra a continuación se representa como se escribe y se lee una cadena "Hola mundo" en un *pipe* anónimo, utilizando para ello `fork()`.

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

...

```

[36] http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html
[37] http://pubs.opengroup.org/onlinepubs/9699919799/functions/read.html
[38] http://pubs.opengroup.org/onlinepubs/9699919799/functions/write.html
[39] http://pubs.opengroup.org/stage7tc1/functions/close.html
[40] Extraído de http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

```
6   int fildes[2];
7   const int BSIZE = 100;
8   char buf[BSIZE];
9   ssize_t nbytes;
10  int status;
11
12  status = pipe(fildes);
13  if (status == -1 ) {
14    // Ocurrió un error al crear la tubería
15    ...
16  }
17
18  switch (fork()) {
19    // Ocurrió un error al hacer fork()
20    case -1:
21      break;
22
23    // El hijo lee desde la tubería
24    case 0:
25      // No necesitamos escribir
26      close(fildes[1]);
27
28      // Leer usando READ
29      // -> Habría que comprobar errores!
30      nbytes = read(fildes[0], buf, BSIZE);
31
32      // En este punto una lectura adicional hubiera llegado a FEOF
33
34      // Cerrar el extremo de lectura
35      close(fildes[0]);
36      exit(EXIT_SUCCESS);
37
38    // El padre escribe en la tubería
39    default:
40      // No necesitamos leer
41      close(fildes[0]);
42
43      // Escribimos datos en la tubería
44      // -> Habría que comprobar errores!
45      write(fildes[1], "Hola Mundo!!\n", 14);
46
47      // El hijo vería FEOF ante una lectura adicional
48      close(fildes[1]);
49      exit(EXIT_SUCCESS);
50  }
```

En los ficheros `pipe.c` y `pipe2.c` dispone de ejemplos, consultelos y modifiquelos hasta que entienda completamente su funcionamiento y el de las funciones que se utilizan.

Por otro lado, también disponemos de lo que se llaman *named pipes* (tuberías con nombre) o FIFOs, que permiten crear una tubería dentro del sistema de archivos para que pueda ser accedida por distintos procesos. Desde C, la función `mkfifo(pathname, permissions)`[41]

[41] http://pubs.opengroup.org/onlinepubs/9699919799/functions/mkfifo.html

permitiría crear una tubería con nombre en el sistema de archivos. Luego abriríamos un extremo para lectura mediante `open(pathname, O_RDONLY)` y otro para escritura mediante `open(pathname, O_WRONLY)`, de manera que la primera llamada a `open` dejaría bloqueado el proceso hasta que se produzca la segunda. Tengalas presente, aunque no se utilizarán en esta práctica.

## 10.4. Colas de mensajes

Las colas de mensajes POSIX suponen otra forma alternativa de comunicación entre procesos. Se basan en la utilización de una **comunicación por paso de mensajes**, es decir, los procesos se comunican e incluso se sincronizan en función de una serie de mensajes que se intercambian entre si. Las colas de mensajes POSIX permiten una comunicación indirecta y simétrica, de forma síncrona o asíncrona.

El sistema operativo pone a disposición de los procesos una serie de colas de mensajes o buzones. Un proceso tiene la posibilidad de depositar mensajes en la cola o de extraerlos de la misma. Algunas de las características a destacar sobre este mecanismo de comunicación son las siguientes:

- La cola está gestionada por el núcleo del sistema operativo y la sincronización es responsabilidad de dicho núcleo. Como programadores, esto **evita que tengamos que preocuparnos de la sincronización** de los procesos.
- Las colas van a tener un determinado identificador y los mensajes que se mandan o reciben a las colas son de **formato libre**.
- Al contrario que con las tuberías, en una cola podemos tener múltiples lectores o escritores. Las colas de mensajes se gestionan mediante la política FIFO (*First In First Out*), sin embargo se puede hacer uso de prioridades de mensajes, para hacer que determinados mensajes se salten este orden FIFO.

Existen dos familias de funciones para manejo de colas de mensajes incluidas en el estándar POSIX y que se pueden acceder desde C:

- Funciones `msg*` (heredadas de System V).
- Funciones `mq_*` (algo más modernas). En nuestro caso, nos vamos a centrar en las funciones `mq_*` por ser más simples de utilizar y aportar algunas ventajas[42].

Como programadores, serán tres las operaciones que realizaremos con las colas de mensajes[43]:

1. Crear o abrir una cola: `mq_open`.
2. Recibir/mandar mensajes desde/a una cola en concreto: `mq_send` y `mq_receive`.
3. Cerrar y/o eliminar una cola: `mq_close` y `mq_unlink`.

**Ojo**: para compilar los ejemplos relacionados con colas, es necesario incluir la librería *real time*, es decir, incluir la opción `-lrt` .

[42] Más información en http://stackoverflow.com/questions/24785230/difference-between-msgget-and-mq-open

[43] Se puede obtener más información en http://www.filibeto.org/unix/tru64/lib/rel/4.0D/APS33DTE/DOCU_011.HTM

### 10.4.1. Creación o apertura de colas

La función a utilizar es `mq_open`[44]:

```
1 #include <mqueue.h>
2 mqd_t mq_open(const char *name, int oflag, mode_t mode, struct
    mq_attr *attr);
```

- `name` es una cadena que identifica a la cola a utilizar (el nombre siempre tendrá una barra al inicio, "`/nombrecola`").
- `oflag` corresponde a la forma de acceso a la cola.
  - En `oflag` tenemos una serie de *flags* binarios que se pueden especificar como un OR a nivel de *bits* de distintas *macros*.

    Por ejemplo, si indicamos `O_CREAT | O_WRONLY` estaremos diciendo que la cola debe crearse si no existe ya y que vamos a utilizarla solo para escritura. Para lectura o para lectura-escritura los *flags* serían `O_RDONLY` y `O_RDWR` respectivamente.
  - Al crear la cola con `mq_open`, podemos incluir el *flag* `O_NONBLOCK` en `oflag`, que hace que la recepción de mensajes sea **no bloqueante**, es decir, la función devuelve un error si no hay ningún mensaje en la cola en lugar de esperar.

    El comportamiento por defecto (sin incluir el *flag*) es **bloqueante**, es decir, si la cola está vacía, el proceso se queda esperando en esa línea de código, hasta que haya un mensaje en la cola.
- `mode` corresponde a los permisos con los cuales creamos la cola.
  - Solo en aquellos casos en que indiquemos que queremos crear la cola (`O_CREAT`), tendrán sentido los argumentos opcionales `mode`. Sirve para especificar los permisos (por ejemplo, `0644` son permisos de lectura y escritura para el propietario y de sólo lectura para el grupo y para otros).
- `attr` es un puntero a una estructura `struct mq_attr` que contiene propiedades de la cola.
  - Solo en aquellos casos en que indiquemos que queremos crear la cola (`O_CREAT`), tendrán sentido los argumentos opcionales `attr`.

    Nos especifica diferentes propiedades de una cola mediante una estructura con varios campos (los campos que vamos a usar son `mq_maxmsg` para el número máximo de mensajes acumulados en la cola y `mq_msgsize` para el tamaño máximo de dichos mensajes).
- La función devuelve un descriptor de cola (parecido a los identificadores de ficheros), que me permitirá realizar operaciones posteriores sobre la misma.

  Si la creación o apertura falla, se devuelve `-1` y `errno` me indicará el código de error (el cuál puede interpretarse haciendo uso de `perror`).

[44] http://pubs.opengroup.org/stage7tc1/functions/mq_open.html, http://linux.die.net/man/3/mq_open

### 10.4.2. Recepción de mensajes desde colas

Para recibir un mensaje desde una cola utilizaremos la función `mq_receive`[45]:

```
#include <mqueue.h>
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
   unsigned *msg_prio);
```

- La función intenta leer un mensaje de la cola `mqdes` (identificador de cola devuelto por `mq_open`).
- El mensaje se almacena en la cadena apuntada por el puntero `msg_ptr`.
- Se debe especificar el tamaño del mensaje a leer en *bytes* (`msg_len`).
- El último argumento (`msg_prio`) es un argumento de salida, un puntero a una variable de tipo `unsigned`, que, a la salida de la función, contendrá la prioridad del mensaje leído.

  El motivo es que, por defecto, siempre se lee el mensaje más antiguo (política FIFO) de máxima prioridad en la cola. Es decir, durante el envío, se puede incrementar la prioridad de los mensajes y esto hará que se adelanten al resto de mensajes antiguos (aunque, en empate de prioridad, el orden sigue siendo FIFO).
- La función devuelve el número de *bytes* que hemos conseguido leer de la cola. Si hubiese cualquier error, devuelve -1 y el código de error en `errno`.

### 10.4.3. Envío de mensajes a colas

Para mandar un mensaje a una cola utilizaremos la función `mq_send`[46]:

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
   unsigned msg_prio);
```

- La función enviará el mensaje apuntado por `msg_ptr` a la cola indicada por `mqdes` (recordad que este identificador es el devuelto por `mq_open`).
- El tamaño del mensaje a enviar (número de *bytes*) se indica mediante `msg_len`.
- Finalmente, el valor `msg_prio` permite indicar la prioridad del mensaje.

  Tal y como se indicó antes, una prioridad mayor que 0, hará que los mensajes se adelanten en la cola a la hora de la recepción.
- Se devuelve un 0 si el envío tiene éxito y un -1 en caso contrario (de nuevo, el código de error vendría en `errno`).

[45] http://pubs.opengroup.org/stage7tc1/functions/mq_receive.html
[46] http://pubs.opengroup.org/stage7tc1/functions/mq_send.html

### 10.4.4. Cierre de colas

Para cerrar una cola (dejar de utilizarla pero que siga existiendo) utilizaremos la función `mq_close`[47]:

```
1 #include <mqueue.h>
2 int mq_close(mqd_t mqdes);
```

- `mqdes` es el descriptor de cola devuelto por `mq_open`.

  La función elimina la asociación entre `mqdes` y la cola correspondiente, es decir, cierra la cola de forma ordenada, pero seguirá disponible para otros procesos, manteniendo sus mensajes si es que los tuviera.

- La función devuelve 0 si no hay ningún error y -1 en caso contrario (con el valor correspondiente de `errno`).

### 10.4.5. Eliminación de colas

Si queremos eliminar una cola de forma permanente ya que estamos seguros que ningún proceso la va a utilizar más, podemos emplear la función `mq_unlink`[48]:

```
1 #include <mqueue.h>
2 int mq_unlink(const char *name);
```

- `name` es el nombre de la cola a eliminar (por ejemplo, "`/nombrecola`"). Antes de eliminarse, se borran todos los mensajes.

- La función devuelve 0 si no hay ningún error y -1 en caso contrario (con el valor correspondiente de `errno`).

## 10.5. Ejemplo simple de uso de colas

A continuación se verá un primer ejemplo simple en el que se hace uso de dos elementos de POSIX: `fork` y colas de mensajes. Concretamente el ejemplo permite comunicarse mediante colas de mensajes a un proceso principal o *main()* con un proceso hijo. El código correspondiente se encuentra en el fichero `ejemplo-mq.c`. Ábralo y consúltelo mientras lee esta sección.

Las primeras líneas de código (previas a la llamado a `fork`) son ejecutadas por el proceso original o padre (antes de clonarse):

- Se definen las propiedades de la cola a utilizar (número máximo de mensajes en la cola en un determinado instante y tamaño máximo de cada mensaje).

- Se hace la llamada al `fork`.

Tras la llamada al `fork`, siguiendo la rama del `switch` correspondiente, el proceso hijo realiza las siguientes acciones:

[47] http://pubs.opengroup.org/stage7tc1/functions/mq_close.html
[48] http://pubs.opengroup.org/stage7tc1/functions/mq_unlink.html

- Abre o crea la cola en modo solo escritura (el hijo solo va a escribir).

  Si hay que crearla, se le ponen permisos de lectura y escritura al usuario actual y de solo lectura al resto.

- Construye el mensaje dentro de la variable `buffer`, introduciendo un número aleatorio entre 0 y 4999.

  En lugar de transformar el número a cadena, se podría haber enviado directamente, realizando un *casting* del puntero correspondiente (`(char *) &numeroAleatorio`). Esto habría que haberlo tenido en cuenta también en el proceso padre.

- Envía el mensaje por la cola `mq`, cierra la cola y sale del programa.

En el caso del proceso padre:

- Abre o crea la cola en modo solo lectura.

  Si hay que crearla, se le ponen permisos de lectura y escritura al usuario actual y de solo lectura al resto. Recuerde que tanto el padre como el proceso hijo están ejecutando en paralelo en el sistema, por lo que cualquiera de los dos puede ser el primero en crear la cola.

- Esperamos a recibir un mensaje por la cola `mq`. La espera (bloqueante) se prolonga hasta que haya un mensaje en la cola, es decir, hasta que el proceso hijo haya realizado el envío.

- Imprimimos el número aleatorio que viene en el mensaje.

- Cierra la cola y, como sabe que nadie más va a utilizarla, la elimina. Por último, esperamos a que el hijo finalice y salimos del programa.

A continuación, se muestra un ejemplo de ejecución de este programa:

```
jfcaballero@NEWTS:~$ ./ejemplo-mq
[PADRE]: mi PID es 8807 y el PID de mi hijo es 8808
[PADRE]: recibiendo mensaje (espera bloqueante)...
[HIJO]: mi PID es 8808 y mi PPID es 8807
[HIJO]: generado el mensaje "4501"
[HIJO]: enviando mensaje...
[HIJO]: Mensaje enviado!
[PADRE]: el mensaje recibido es "4501"
[PADRE]: Cola cerrada.
Hijo PID:8808 finalizado, estado=0
No hay más hijos que esperar
status de errno=10, definido como No child processes
```

## 10.6. Ejemplo cliente-servidor de uso de colas

Seguidamente se estudiará un segundo ejemplo[49] que puede encontrar en los ficheros de código `common.h`, `servidor.c` y `cliente.c`.

[49] Adaptado de http://stackoverflow.com/questions/3056307

Este caso contempla dos procesos independientes, de forma que el `servidor` crea una cola y espera a que el `cliente` introduzca mensajes en esa cola.

El programa `cliente` lee por teclado los mensajes a enviar y realiza un envío cada vez que pulsamos `INTRO`.

Por cada mensaje recibido, el `servidor` imprime su valor en consola.

La comunicación finaliza y los programas terminan, cuando el `cliente` manda el mensaje de salida (establecido como "`exit`" en `common.h`).

Se ha considerado que el `servidor` sea el que cree la cola, para que así quede bloqueado hasta que el `cliente` arranque y mande su mensaje. Por tanto, es también el `servidor` el que la elimina cuando la comunicación finaliza.

Primero se debe lanzar el `servidor`, quedando a la espera de los mensajes del `cliente`:

```
jfcaballero@NEWTS:~$ ./servidor
```

Posteriormente, se lanza el `cliente` desde otra terminal, quedando a la espera de que escribamos un mensaje:

```
jfcaballero@NEWTS:~$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
>
```

Escribimos "`hola`" y pulsamos `INTRO`:

```
jfcaballero@NEWTS:~$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
> hola
>
```

El mensaje ya se ha enviado. Si se vuelve a la terminal del `servidor`, se podrá comprobar lo siguiente en cuanto a su recepción:

```
jfcaballero@NEWTS:~$ ./servidor
Recibido el mensaje: hola
```

Si ahora se envía el mensaje "`exit`" desde el `cliente` se observa que que el `servidor` se para:

```
jfcaballero@NEWTS:~$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
> hola
> exit
```

## 10.7. Ejercicio

Analice y estudie el código de los dos programas de los fichero `servidor.c` y `cliente.c`, y modifiquelos hasta que entienda su funcionamiento y el de las funciones que se utilizan.

## 11. Ejercicio resumen 2

El fichero de código de este ejercicio será `ejercicio2.c` y el ejecutable `ejercicio2`. Cuando la comunicación entre procesos es simple y ambos extremos de la comunicación han sido generados con un `fork()` o pertenecen al mismo proceso, se pueden utilizar tuberías anónimas (*pipes*).

Las tuberías son unidireccionales, por lo que para hacer una comunicación full-duplex se necesitarían dos tuberías. Implemente un programa en C que cree un proceso hijo usando la primitiva `fork()`, de forma que el padre y el hijo se comunicarán mediante dos tuberías de la siguiente manera:

- El padre pedirá por pantalla dos números enteros, y los enviará a su hijo por la **tubería 1**, con el siguiente formato: "num1;num2", es decir, un único mensaje con los dos números separados por ";".
- El hijo leerá de la **tubería 1** y recogerá los enteros, los separará haciendo uso de la función *strtok*[50] y calculará si ambos enteros son números primos gemelos[51], primos no gemelos o alguno de los dos no es un número primo.
- A continuación el padre recibirá por la **tubería 2** una palabra enviada por el hijo, que puede ser: "gemelos", "primos", "no-primos", y lo imprimirá por pantalla. Eso dependerá de si son primos gemelos, primos no gemelos o alguno de los dos no es un número primo, respectivamente .

No olvide que el padre debe esperar al hijo para una finalización correcta del programa.
Ejemplo de salida esperada para dos números primos gemelos:

```
jfcaballero@NEWTS:~$ ./ejercicio2.out
[PADRE]: Inserte dos números enteros:
[PADRE]: 3
[PADRE]: 5
[PADRE]: He escrito los dos números en la tuberia 1.
[PADRE]: Tuberia 1 cerrada.
[HIJO]: Leido 3;5 de la tuberia 1.
[HIJO]: Tuberia 1 cerrada.
[HIJO]: He escrito en la tuberia 2.
[HIJO]: Tuberia 2 cerrada.
[PADRE]: Leido gemelos de la tuberia 2.
[PADRE]: Tuberia 2 cerrada.
Proceso Padre, Hijo con PID 7475 finalizado, status = 0
Proceso Padre, valor de errno = 10, definido como No child
   processes, no hay más hijos que esperar!
```

Ejemplo de salida esperada para dos números primos no gemelos:

```
jfcaballero@NEWTS:~$ ./ejercicio2.out
[PADRE]: Inserte dos números enteros:
```

[50] Función *strtok* http://c.conclase.net/librerias/?ansifun=strtok
[51] Números primos gemelos https://en.wikipedia.org/wiki/Twin_prime

```
[PADRE]: 5
[PADRE]: 3
[PADRE]: He escrito los dos números en la tuberia 1.
[PADRE]: Tuberia 1 cerrada.
[HIJO]: Leido 5;3 de la tuberia 1.
[HIJO]: Tuberia 1 cerrada.
[HIJO]: He escrito en la tuberia 2.
[HIJO]: Tuberia 2 cerrada.
[PADRE]: Leido primos de la tuberia 2.
[PADRE]: Tuberia 2 cerrada.
Proceso Padre, Hijo con PID 7483 finalizado, status = 0
Proceso Padre, valor de errno = 10, definido como No child
   processes, no hay más hijos que esperar!
```

Ejemplo de salida esperada para dos números que no son primos:

```
jfcaballero@NEWTS:~$ ./ejercicio2.out
[PADRE]: Inserte dos números enteros:
[PADRE]: 3
[PADRE]: 6
[PADRE]: He escrito los dos números en la tuberia 1.
[PADRE]: Tuberia 1 cerrada.
[HIJO]: Leido 3;6 de la tuberia 1.
[HIJO]: Tuberia 1 cerrada.
[HIJO]: He escrito en la tuberia 2.
[HIJO]: Tuberia 2 cerrada.
[PADRE]: Leido no-primos de la tuberia 2.
[PADRE]: Tuberia 2 cerrada.
Proceso Padre, Hijo con PID 7485 finalizado, status = 0
Proceso Padre, valor de errno = 10, definido como No child
   processes, no hay más hijos que esperar!
```

# 12. Expresiones regulares

A continuación se muestra un uso básico de las expresiones regulares. Una expresión regular (***regex***) describe un conjunto de cadenas de texto, de forma que en determinadas aplicaciones, ahorran mucho tiempo y hacen el código más robusto. Por ejemplo se pueden utilizar en:

- En entornos UNIX, con comandos como *grep*, *sed*, *awk*. Trabajará con ellos en posteriores prácticas.
- De manera intensiva, en lenguajes de programación como perl, python,ruby, XML...
- En bases de datos.

La expresión regular más simple sería la que busca una secuencia fija de caracteres literales. Se dice que una cadena cumple o empareja un expresión regular si contiene esa secuencia.

Por ejemplo, dado el literal "ola":

- Ella me dijo hola → Empareja.
- Ella me dijo mola → Empareja.
- Lola me dijo hola → Empareja 2 veces.
- Ella me dijo, ¡adiós, eres un pesado! → No Empareja.

En expresiones regulares, el carácter "." empareja cualquier cosa, por ejemplo, dado la expresión "ola.", un emparejamiento en una cadena podría ser:

- Lola me dijo hola. → Empareja 2 veces.

Para el manejo de expresiones regulares, la *GNU C Library* incluye una serie de funciones (`regcomp`, `regerror`, `regexec` y `regfree`) que permiten comprobar si una cadena empareja una expresión regular. Estas funciones están incluidas en `regex.h`[52]. Consulte el ejemplo del enlace de la nota al pie. Todo esto forma parte del estándar POSIX.

También dispone de información general [53] sobre `regex.h` e información completa del mismo [54].

## 13. Ejercicio resumen 3

Los ficheros de código utilizados en este ejercicio serán `ejercicio3-servidor.c`, `ejercicio3-cliente.c` y `common.h`. Los ejecutables generados tendrán como nombre `ejercicio3-servidor` y `ejercicio3-cliente`.

Implemente un programa en C que utilice colas de mensajes y comunique dos procesos, de forma que cumpla los siguientes requisitos (puede utilizar como base el código de los ficheros `common.h`, `servidor.c` y `cliente.c` que se le han proporcionado como ejemplo):

1. Hay un proceso cliente que enviará cadenas leidas desde teclado y las envía mediante mensajes a un proceso servidor cada vez que pulsamos INTRO.
2. El servidor comprobará si los mensajes enviados por el cliente emparejan o no una determinada expresión regular, indicada al arrancar al servidor.

   Tras esto, el servidor mandará un mensaje al cliente, por otra cola distinta, con la cadena "Empareja" o "No Empareja", según el resultado del emparejamiento.

   Solo es necesario que el servidor diga si empareja o no empareja un cadena recibida, no es necesario que se compruebe el número de veces que se hace un emparejamiento, ni en que posición de la cadena recibida comienza el emparejamiento.

   Por tanto habrá dos colas, ambas creadas por el servidor:

   *a*) Una cola servirá para que el cliente le envie al servidor las cadenas de texto donde éste último tiene que buscar emparejamientos.
   De esta cola leerá el servidor para obtener dichas cadenas y analizarlas, usando para ello una serie de funciones para expresiones regulares.

[52] Tienes un ejemplo de uso de `regex.h` en `http://www.peope.net/old/regex.html`
[53] `http://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html`
[54] `http://pubs.opengroup.org/stage7tc1/functions/regexec.html`

*b*) Otra cola por la que el servidor eviará al cliente si una expresión regular empareja con la cadena de texto recibida por la primera cola.
De esta segunda cola leera el cliente para mostrar si la cadena que envió al servidor tiene emparejamiento.

Se han de tener en cuenta los siguientes items:

- La expresión regular a buscar se indicará por línea de argumentos del servidor, utilizando la opción `-r/--regex`.
- La cola de mensajes adicionales de tipo "Empareja" o "No Empareja", enviados desde el servidor al cliente, se creará y eliminará por parte del servidor (que siempre es el primero en lanzarse) y la abrirá el cliente.
- Si el servidor tiene cualquier problema en su ejecución, por ejemplo errores al compilar la expresión regular, deberá mandar el mensaje de salida, para forzar al cliente a parar.

3. En un sistema compartido, debemos asegurar que la cola de mensajes que estamos utilizando es única para el usuario. Por ejemplo, si dos de vosotros os conectaseis por `ssh` a `ts.uco.es` y utilizarais el cliente servidor del ejemplo, los programas de ambos usuarios interactuarían entre si y los resultados no serían los deseados. Para evitar esto, en este ejercicio se pide que como nombre para la cola utilicéis el nombre original seguido vuestro nombre de usuario, es decir, "`nombre_original-usuario`". Para obtener el nombre de usuario, deberás consultar la variable de entorno correspondiente.

4. En el código de que se dispone en Moodle (ficheros `common.h`, `servidor.c` y `cliente.c`), tanto el cliente como el servidor tienen incluidas unas funciones de *log*. Estas funciones implementan un pequeño sistema de registro o *log*. Utilizándolas se registran en ficheros de texto los mensajes que los programas van mostrando por pantalla (`log-servidor.txt` y `log-cliente.txt`).

   Por ejemplo, si queremos registrar en el cliente un mensaje simple, haríamos la siguiente llamada:

```
1 funcionLog("Error al abrir la cola del servidor");
```

   Si quisiéramos registrar un mensaje más complejo (por ejemplo, donde incluimos el mensaje recibido a través de la cola), la llamada podría hacerse del siguiente modo:

```
1 char msgbuf[100];
2 ...
3 sprintf(msgbuf,"Recibido el mensaje: %s\n", buffer);
4 funcionLog(msgbuf);
```

   Utilice estas llamadas para dejar registro en fichero de texto de todos los mensajes que se muestren por pantalla en la ejecución del cliente y el servidor, incluidos los errores que se imprimen por consola.

5. Captura las señales `SIGTERM`, `SIGINT` que podrá **enviar el cliente** para gestionar adecuadamente el fin del programa servidor y de el mismo. Puede asociar estas señales con una misma función que pare el programa.

- Dicha función deberá, en primer lugar, registrar la señal capturada (y su número entero) en el fichero de *log* del cliente.
- El cliente, antes de salir, deberá mandar a la cola correspondiente, un mensaje de fin de sesión (que debe interpretar el servidor), que hará que el otro extremo deje de esperar mensajes. Este mensaje también se registrará en los logs.
- Se deberá cerrar, en caso de que estuvieran abiertas, aquellas colas que se estén utilizando y el fichero de *log*.

A continuación, se muestran ejemplos de invocación del cliente:

```
$ ejercicio3-cliente
> Tengo un perro en mi casa
< Empareja
> Viva el vino!
< No Empareja
> Capturada señal SIGINT (2) por parte del cliente
> Se finalizará la sesión, enviando fin de sesión al servidor...
> Cerrando y eliminando las estructuras pertinentes. Fin del
   programa...

$ ./ejercicio3-cliente -h
Uso del programa: ejercicio3-cliente [opciones]
Opciones:
-h, --help        Imprimir esta ayuda

Para finalizar el programa envie señal SIGINT o SIGTERM.
```

A continuación, se muestran ejemplos de invocación del servidor:

```
# Buscar 'perro'
$ ./ejercicio3-servidor --regex 'perro'
< Recibido "Tengo un perro en mi casa"
> Empareja
< Recibido "Viva el vino!"
> No empareja
< Recibido mensaje de finalización por parte del cliente por
   captura de señal SIGINT (2)
> Cerrando y eliminando las estructuras pertinentes. Fin del
   programa...

# Ayuda del programa
$ ./ejercicio3-servidor -h
Uso del programa: ejercicio4-servidor [opciones]
Opciones:
-h, --help        Imprimir esta ayuda
-r, --regex=EXPR  Expresión regular a utilizar
```

## 14. Ejercicio resumen 4

Realice el mismo programa anterior (el cliente manda mensajes al servidor y el servidor responde indicando si hay o no hay emparejamiento con una expresión regular) pero en este caso vamos a utilizar la primitiva fork(), el padre será el servidor y el hijo será el cliente. El fichero de código de este ejercicio será `ejercicio4.c` y el ejecutable `ejercicio4`.

- Todo el paso de mensajes deberá de ser resuelto haciendo uso de una **única cola** en la que tanto padre como hijo puedan **leer** y **escribir** alternándose. El hijo enviará una cadena al padre y esperará a recibir el procesado antes de volver a enviar. El padre esperará una cadena del hijo, la procesará y devolverá el resultado del procesado para ponerse de nuevo a esperar. Y así consecutivamente.
- No es necesario que realice la parte de captura de señales.
- No es necesario que realice la parte de escritura en el *log*.

A continuación, se muestran ejemplos de invocación:

```
jfcaballero@NEWTS:~$ ./ejercicio4.out -r hola
rvalue = hola, hflag = 0
[HIJO] Mandando mensajes al servidor (escribir "exit" para parar):
> hola!
[PADRE] Recibido el mensaje: hola!
[HIJO] Recibido el mensaje: Empareja
> adios
[PADRE] Recibido el mensaje: adios
[HIJO] Recibido el mensaje: No empareja
> que tal todo?
[PADRE] Recibido el mensaje: que tal todo?
[HIJO] Recibido el mensaje: No empareja
> holaaaaa
[PADRE] Recibido el mensaje: holaaaaa
[HIJO] Recibido el mensaje: Empareja
> exit
Proceso Padre, Hijo con PID 5861 finalizado, status = 0
Proceso Padre, valor de errno = 10, definido como No child
   processes, no hay más hijos que esperar!
```

## Referencias

[1] Javier Sánchez Monedero. Programación posix, 2012. URL: http://www.uco.es/~i02samoj/docencia/pas/practica-POSIX.pdf.

[2] Wikipedia. Posix – wikipedia, la enciclopedia libre, 2012. [Internet; descargado 12-abril-2012]. URL: http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603.

[3] The IEEE and The Open Group. Posix.1-2008 – the open group base specifications issue 7, 2008. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[4] Proyecto GNU. Gnu c library, 2015. URL: http://www.gnu.org/software/libc/libc.html.

[5] Wikipedia. Glibc – wikipedia, la enciclopedia libre, 2015. [Internet; descargado 22-marzo-2015]. URL: http://es.wikipedia.org/wiki/Glibc.

[6] Brian W. Kernighan, Dennis Ritchie, and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Pearson Educación, 2 edition, 1991.

[7] Tim Love. Fork and exec, 2008. URL: http://www-h.eng.cam.ac.uk/help/tpl/unix/fork.html.

[8] Wikipedia. Dennis ritchie – wikipedia, la enciclopedia libre, 2012. URL: http://es.wikipedia.org/wiki/Dennis_Ritchie.

[9] chuidiang.com. Programación de sockets en c de unix/linux, 2007. URL: http://www.chuidiang.com/clinux/sockets/sockets_simp.php.

[10] Andrew Gierth Vic Metcalfe and other contributers. Programming UNIX Sockets in C - Frequently Asked Questions. 4.2 Why don't my sockets close?, 1996. URL: http://www.softlab.ntua.gr/facilities/documentation/unix/unix-socket-faq/unix-socket-faq-4.html#ss4.2.