Práctica 1: Programación en POSIX [*]

Programación y Administración de Sistemas

2020-2021

Juan Carlos Fernández Caballero
jfcaballero@uco.es

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

Febrero de 2021

Índice

[*]Parte de los contenidos de este guión se ha elaborado con la colaboración de los profesores Javier Sánchez Monedero [5], Pedro Antonio Gutiérrez Peña, David Guijo Rubio, Juan Carlos Fernández Caballero y Javier Barbero Gómez, además de las referencias y bibliografía asociada.

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 estándar 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 [7]. La última versión de la especificación POSIX fue publicada en el año 2018 y se le conoce como "POSIX.1-2017", "IEEE Std 1003.1-2017" y "The Open Group Technical Standard Base Specifications, Issue 7" [2].

POSIX recoge al Estándar 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 estándar 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 estándar, 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 estándar POSIX, por lo que incluye a su vez la implementación del estándar de C. Por tanto la biblioteca `glibc` sigue en su implementación todos los estándares más relevantes, ANSI C y POSIX.1-2017 [1]. `glibc` se distribuye bajo los términos de la licencia GNU LGPL[5]. Decir también que la implementación del estándar de C se encuentra en una biblioteca llamada `libc` [6].

`glibc` es muy *portable* y soporta gran cantidad de plataformas de *hardware* [6]. 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 términos que se acaban de exponer.

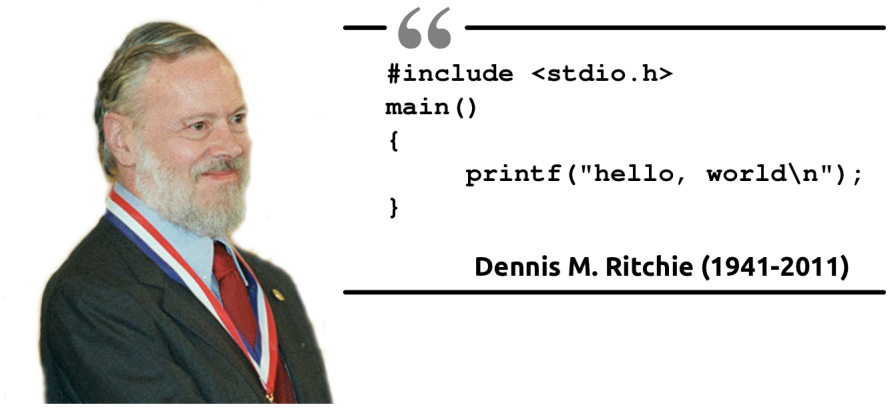
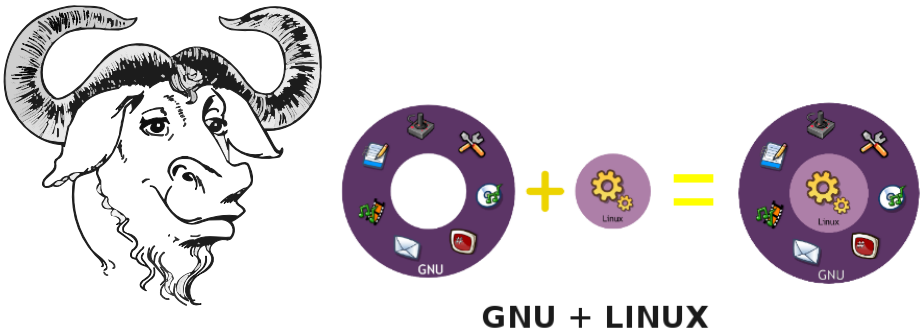


Figura 1: **Dennis MacAlistair Ritchie** (9 de septiembre de 1941 - 12 de octubre de 2011). Colaboró en el diseño y desarrollo de los sistemas operativos Multics y Unix, así como el desarrollo de varios lenguajes de programación como el C, tema sobre el cual escribió un célebre clásico de las ciencias de la computación junto a Brian Wilson Kernighan: *El lenguaje de programación C* [3].



(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 en nuestros programas y a consultar su documentación asociada.
- Mejorar la programación viendo ejemplos hechos por los desarrolladores de las bibliotecas y creando nuevos programas.
- Aprender cómo funcionan algunas partes de GNU/Linux.
- 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 algunas metodologías de paso de mensajes.

  En la asignatura de Sistemas Operativos ya estudió algunas **IPC** (Inter-Process Communication) [9] o formas de comunicar y/o sincronizar procesos e hilos, como los semáforos, señales y la 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 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 exámenes en ordenador.
- Acostúmbrese a una buena modularidad del código en funciones, a la comprobación de errores en los argumentos de los programas y a la claridad y formato del código fuente y las salidas generadas. Esto es fundamental para generar programas de calidad, tanto para superar la asignatura como para su trabajo como Ingeniero Informático.

[9] https://es.wikipedia.org/wiki/Comunicaci%C3%B3n_entre_procesos

- Todos los programas deben funcionar correctamente en la máquina `ts.uco.es`[10], 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 exámenes 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 ítems anteriores, el profesorado otorgará a un ejercicio más o menos puntuación dependiendo de: control de errores utilizado, invocación y uso correcto de las funciones, indentación y claridad de la programación.

- **Documentación POSIX.1-2017:** Especificación del estándar POSIX. Dependiendo de la parte que se documente es más o menos pedagógica[11]. Téngala siempre en cuenta y consúltela a lo largo de la práctica, es lo que tendrá como documentación en los exámenes en ordenador, junto con la ayuda del comando `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[12]. 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[13]. 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 `-o<parámetro>`.

- 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`.

[10] Acceso en las aulas o a través de SSH (instrucciones en el siguiente enlace, diapositiva 16: http://www.uco.es/servicios/informatica/images/documentos/conexion_a_escritorios_docentes.pdf)

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

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

[13] 12.1 Utility Argument Syntax, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

La función `getopt()` del estándar [14] [15] ayuda al manejo de las opciones siguiendo las directrices POSIX.1-2017.

Puede consultar también la sección «*Using the `getopt` function*» de la documentación[16] para saber cómo funciona `getopt()`, qué valores espera y qué comportamiento tiene. También puede ver un código de ejemplo [17] 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()` [18]. 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 [19] 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 exámenes en ordenador.

4.2. Ejercicio

Lea el código de los ficheros `ejemplo-getopt.c` y `ejemplo-getoptlong.c` y los comentarios que aparecen en los mismos, compílelos y ejecútelos 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

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
```

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

En el fichero `ejemplo-getenv.c` hay un ejemplo de uso de la función `getenv()`[20]. 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.

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` [21].

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` [22]
- Funciones y estructuras de acceso a la información de grupos:
  `/usr/include/grp.h` [23]

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.

[20] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getenv.html
[21] gestión de usuarios y grupos en GNU/Linux
[22] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pwd.h.html
[23] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/grp.h.html

Por otro lado, la función `getpwnam()` [24] devuelve una estructura con información de un usuario previo paso de su `login` como parámetro.

Hay **algunos campos de la información del usuario que POSIX no indica**, pero que si implementa `glibc`, puede consultarlos en la documentación asociada [25], a partir de la sección de la documentación de `glibc` «*User Database*». [26]

Con respecto a los grupos, la llamada a la función `getgrgid()` [27] [28], devuelve una estructura con información de un grupo previo paso de su `gid` como parámetro; y la función y `getgrnam()` [29] devuelve una estructura con información de un grupo previo paso de su `nombre` de grupo como parámetro respectivamente.

6.2. Ejercicio

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.

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` del usuario por ejemplo a partir de la variable de entorno `USER` (`getenv("USER")`), para pasárselo posteriormente a `getpwnam()`. No use la función `getlogin()`.

7. Ejercicio resumen

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/--user` se utilizará para indicar un usuario. Si le pasamos como argumento un número, lo interpretará como UID, en caso contrario como el nombre del usuario. Se mostrará la información correspondiente a su estructura `passwd`.
- La opción `-g/--group` se utilizará para indicar un grupo. Si le pasamos como argumento un número, lo interpretará como GID, en caso contrario como el nombre del grupo. Se mostrará la información correspondiente a su estructura `group`.
- La opción `-a/--active` será equivalente a especificar`--user` con el usuario actual.
- La opción `-m/--maingroup` modifica a `--user` o `--active` y hace que imprima la información de su grupo principal (mismo formato que `--group`).

[24] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getpwnam.html
[25] https://www.gnu.org/software/libc/manual/html_node/User-Data-Structure.html#User-Data-Structure
[26] https://www.gnu.org/software/libc/manual/html_node/User-Database.html
[27] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getgrgid.html
[28] http://www.gnu.org/software/libc/manual/html_node/Group-Database.html
[29] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getgrnam.html

- 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 `--group`. No muestre directamente el contenido del fichero, recórralo y muestre la información por cada grupo que haya.
- Si se invoca al programa sin ninguna opción se mostrará la información del usuario actual (con el mismo formato que que `-u/--user`) y del grupo al que pertenece (con el mismo formato que que `-g/--group`).
- 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 línea de comandos opciones que sean incompatibles. Las únicas combinaciones posibles son:
  - `--help`, junto con cualquiera (se ignoran las demás)
  - vacío
  - `--user <usuario>`
  - `--user <usuario> --maingroup`
  - `--active`
  - `--active --maingroup`
  - `--group <grupo>`
  - `--allgroups`

Ejemplos de llamadas válidas serían la siguientes:

```
i32bagoj@NEWTS:~$ ./a.out --user i42vayuv
Usuario:
Nombre: VICTOR MANUEL VARGAS YUN
Login: i42vayuv
Password: *
UID: 79892
Home: /home/i42vayuv
Shell: /bin/bash
Número de grupo principal: 700

i32bagoj@NEWTS:~$ ./a.out -a -m
Usuario:
Nombre: JAVIER BARBERO GOMEZ
Login: i32bagoj
Password: *
UID: 74064
Home: /home/i32bagoj
```

```
Shell: /bin/bash
Número de grupo principal: 700
Grupo:
Nombre del grupo: upi0
GID: 700
Miembros secundarios:

i32bagoj@NEWTS:~$ ./a.out --group upi0
Grupo:
Nombre del grupo: upi0
GID: 700
Miembros secundarios:

i32bagoj@NEWTS:~$ ./a.out --allgroups -m
La opción --maingroup sólo puede acompañar a --user o --active
Uso del programa: ejercicio1 [opciones]
Opciones:
-h, --help                    Imprimir esta ayuda
-u, --user (<nombre>|<uid>)   Información sobre el usuario
-a, --active                  Información sobre el usuario activo actual
-m, --maingroup               Además de info de usuario, imprimir la info de su
    grupo principal
-g, --group (<nombre>|<gid>) Información sobre el grupo
-s, --allgroups               Muestra info de todos los grupos del sistema

i32bagoj@NEWTS:~$ ./a.out --allgroups
Grupo:
Nombre del grupo: root
GID: 0
Miembros secundarios:
Grupo:
Nombre del grupo: daemon
GID: 1
Miembros secundarios:
Grupo:
Nombre del grupo: bin
GID: 2
Miembros secundarios:
[...]
```

8. Creación de procesos (**fork** y **exec**)

En la asignatura de Sistemas Operativos ya estudió el uso de `fork()` para crear procesos y el uso de la familia de funciones `exec()`, por lo que aquí solo se hará un breve recordatorio.

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) [30].

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

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

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`.

Un proceso padre debe esperar a que un proceso hijo termine, para ello se utilizan las funciones `wait()` y `waitpid()` [31] [32]. 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.

A nivel de programación, en C se crea un subproceso llamando a la función `fork()` [33] [34]. Tiene un pequeño manual y mucho código de ejemplo en [4].

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 del código del programa `ejemplo-fork.c` sería el siguiente:

```
$ ./ejemplo-fork
Soy el Padre, mi PID es 3740 y el PID de mi hijo es 3741
Soy el Hijo, mi PID es 3741 y mi PPID es 3740
Proceso Padre, Hijo con PID 3741 finalizado, status = 0
Proceso Padre 3740, no hay mas hijos que esperar. Valor de errno =
   10, definido como: No child processes
```

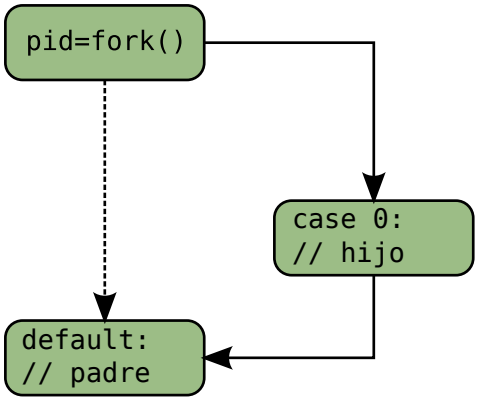


Figura 3: Esquema de llamadas y procesos generados por `fork()` en el ejemplo.

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()` [35] 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

[31] www.gnu.org/software/libc/manual/html_node/Process-Completion.html
[32] http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html
[33] http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html
[34] Puede ver un código con muchos comentarios en la siguiente entrada de Wikipedia http://es.wikipedia.org/wiki/Bifurcaci%C3%B3n_%28sistema_operativo%29
[35] http://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

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`

Un ejemplo de la salida de la ejecución del código del programa `ejemplo-fork-exec.c`, el cual hace un listado del directorio actual por parte de un hijo, sería el siguiente:

```
Hijo 3979 ejecutando comando ls...
total 28
-rwxr-xr-x 1 jfcaballero jfcaballero 12816 feb 11 11:29 ejemplo-
   fork-exec
-rwxrwxrwx 1 jfcaballero jfcaballero  1565 feb 11 11:27 ejemplo-
   fork-exec.c
-rwxrwxrwx 1 jfcaballero jfcaballero  1343 feb 11 11:20
   esperadehijoswaitpid.c
-rwxrwxrwx 1 jfcaballero jfcaballero  1477 feb 11 11:18 ejemplo-
   fork.c
Proceso Padre, Hijo con PID 3979 finalizado, status = 0
Proceso Padre 3978, no hay mas hijos que esperar. Valor de errno =
   10, definido como: No child processes
```

9. Señales entre procesos

En la asignatura de Sistemas Operativos ya estudió el uso de señales para comunicación entre procesos, por lo que aquí solo se hará un breve recordatorio.

Las señales [36] entre procesos 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 nivel del *software* llame a la subrutina o función definida en una capa superior (ver Figura 4, fuente Wikipedia[37]).

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()`[38] 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);
```

[36] http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html
[37] http://en.wikipedia.org/wiki/Callback_%28computer_science%29
[38] http://pubs.opengroup.org/onlinepubs/9699919799/functions/signal.html

```
// 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
  ...

}
```

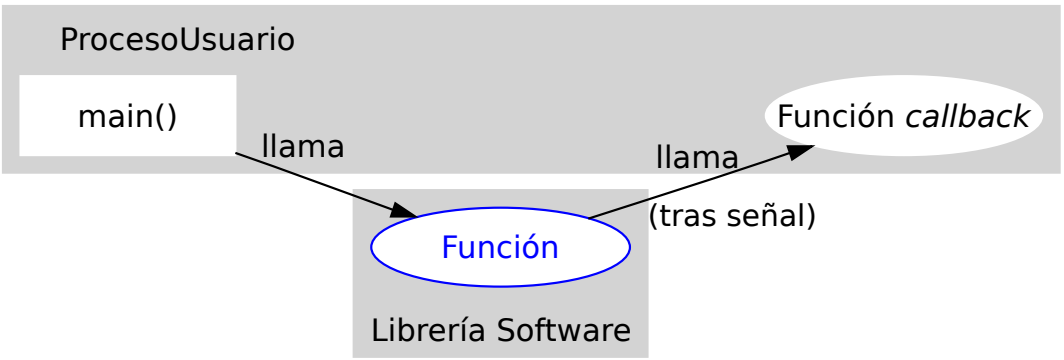


Figura 4: Esquema del funcionamiento de las *callbacks* o *retrollamadas*.

El código del fichero `ejemplo-signal.c`[39] contiene ejemplos de captura de señales POSIX enviadas a un proceso. 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.

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:

```
$  ./ejemplo-signal
No puedo asociar la señal SIGKILL al manejador!
Capturé la señal SIGHUP y no salgo!
```

[39] Adaptado de http://www.amparo.net/ce155/signals-ex.html

```
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:

```
$ ./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 los siguientes, algunos de ellos ya los ha estudiado en la asignatura de Sistemas Operativos, aquí se ampliarán con las tuberías y las colas de mensajes:

- Semáforos.

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

10.1. 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:

```
1 $ ls | wc -l
2 44
```

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

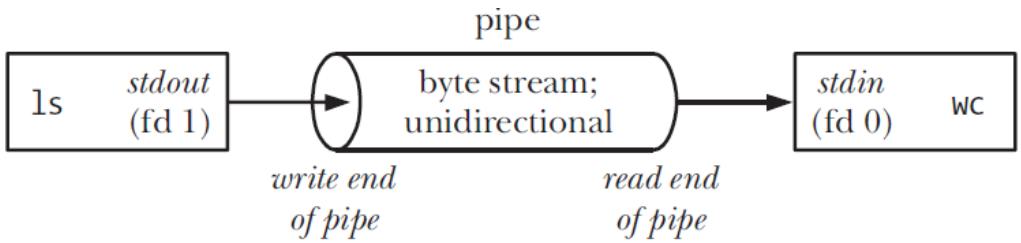


Figura 5: Intercomunicación entre procesos utilizando la tubería `ls | wc -l`. "*write end*" significa extremo de escritura y "*read end*" extremo de lectura.

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** (son las que se estudiarán en esta práctica), ya que se crea desde `bash` de forma temporal para intercomunicar dos procesos.

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)`[40] 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. Téngalas presente, aunque no se utilizarán en esta práctica.

Podemos crear tuberías anónimas en un programa en C mediante la función `pipe()` de `unistd.h`[41]:

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

```
1 #include <unistd.h>
2 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`[42] y `write`[43], cuyo uso es similar a `fread` y `fwrite`.

Una vez utilizados los extremos de lectura y/o escritura, los podemos cerrar con `close`[44].

En el fichero `pipe.c` se escribe y se lee una cadena "Hola mundo" en un *pipe* anónimo [45], utilizando para ello `fork()`. En el fichero `pipe2.c` dispone de un ejemplo en el que dos procesos se envían por una tubería números aleatorios. Estudie y analice ambos programas.

10.1.1. Ejercicio

En el fichero `pipebidireccinal.c` se encuentra un programa que crea dos procesos que se comunican mediante tuberías de manera bidireccional. Estúdielo hasta que lo entienda y describa que es lo que hace el programa.

10.2. 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:

[42] http://pubs.opengroup.org/onlinepubs/9699919799/functions/read.html
[43] http://pubs.opengroup.org/onlinepubs/9699919799/functions/write.html
[44] http://pubs.opengroup.org/stage7tc1/functions/close.html
[45] http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

- 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[46].

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

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` .

10.2.1. Creación o apertura de colas

La función a utilizar es `mq_open()`[47]:

```
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.

    ```
    mq_server = mq_open(serverQueue, O_CREAT | O_RDWR, 0644, &attr)
    ```

  - Al crear la cola con `mq_open`, podemos incluir el *flag* `O_NONBLOCK` en `oflag`, que hace que el envío/recepción de mensajes sea **no bloqueante**. Es decir, la función devuelve un error si no hay espacio libre para escribir o ningún mensaje para leer en la cola, en lugar de esperar.
    El comportamiento por defecto (sin incluir el *flag*) es **bloqueante**, es decir, si la cola está vacía y se intenta leer, el proceso se queda esperando en esa línea de código hasta que haya un mensaje en la cola. De la misma forma, si la cola está llena y se intenta escribir, el proceso se queda esperando hasta que se libere espacio.

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

[47] https://pubs.opengroup.org/onlinepubs/9699919799/, http://linux.die.net/man/3/mq_open

- `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 permitirá realizar operaciones posteriores sobre la misma.

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

10.2.2. Recepción de mensajes desde colas

Para recibir un mensaje desde una cola utilizaremos la función `mq_receive()`[48]:

```
1 #include <mqueue.h>
2 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`.

[48] http://pubs.opengroup.org/stage7tc1/functions/mq_receive.html

10.2.3. Envío de mensajes a colas

Para mandar un mensaje a una cola utilizaremos la función `mq_send()`[49]:

```
1 #include <mqueue.h>
2 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`).

10.2.4. Cierre de colas

Para cerrar una cola (dejar de utilizarla pero que siga existiendo) utilizaremos la función `mq_close()`[50]:

```
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`).

10.2.5. 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()`[51]:

```
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.

[49] http://pubs.opengroup.org/stage7tc1/functions/mq_send.html
[50] http://pubs.opengroup.org/stage7tc1/functions/mq_close.html
[51] http://pubs.opengroup.org/stage7tc1/functions/mq_unlink.html

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

10.2.6. Ejercicio 1

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 concienzudamente.

Las primeras líneas de código (previas a la llamada 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).
- 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 código del ejemplo, los programas de ambos usuarios interactuarían entre si y los resultados no serían los deseados. Para evitar esto, el nombre de la cola que utilicéis podría ser el nombre original seguido vuestro nombre de usuario, es decir, "`nombre_original-usuario`". Para obtener el nombre de usuario, se puede consultar la variable de entorno correspondiente.
- Se hace la llamada a `fork()`.

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

- 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.
- 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`, y la longitud correcta para un `int`). 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:

```
[PADRE]: El nombre de la cola es: /una_cola-jfcaballero
[PADRE]: El descriptor de la cola es:3
[PADRE]: Mi PID es 14140 y el PID de mi hijo es 14141
[PADRE]: Recibiendo mensaje (espera bloqueante)...
[HIJO]: El nombre de la cola es: /una_cola-jfcaballero
[HIJO]: El descriptor de la cola es: 3
[HIJO]: Mi PID es 14141 y mi PPID es 14140
[HIJO]: Generado el mensaje "4804"
[HIJO]: Enviando mensaje...
[HIJO]: Mensaje enviado!
[HIJO]: Cola cerrada.
[PADRE]: El mensaje recibido es "4804"
[PADRE]: Cola cerrada.
Proceso Padre, Hijo con PID 14141 finalizado, status = 0
Proceso Padre 14140, no hay mas hijos que esperar. Valor de errno =
    10, definido como: No child processes
```

10.2.7. Ejercicio 2

Seguidamente se estudiará un ejemplo [52] cliente-servidor que usa colas y que puede encontrar en los ficheros de código `common.h`, `servidor.c` y `cliente.c`. Analice y estudie el código de los dos programas hasta que entienda su funcionamiento y el de las funciones que se utilizan.

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`:

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

```
jfcaballero@NEWTS:~$ ./servidor
[Servidor]: El nombre de la cola es: /server_queue-jfcaballero
[Servidor]: El descriptor de la cola es: 3
```

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

```
jfcaballero@NEWTS:~$ ./cliente
[Cliente]: El nombre de la cola es: /server_queue-jfcaballero
[Cliente]: El descriptor de la cola es: 3
Mandando mensajes al servidor (escribir "exit" para parar):
>
```

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

```
jfcaballero@NEWTS:~$ ./cliente
[Cliente]: El nombre de la cola es: /server_queue-jfcaballero
[Cliente]: El descriptor de la cola es: 3
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
[Servidor]: El nombre de la cola es: /server_queue-jfcaballero
[Servidor]: El descriptor de la cola es: 3
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
[Cliente]: El nombre de la cola es: /server_queue-jfcaballero
[Cliente]: El descriptor de la cola es: 3
Mandando mensajes al servidor (escribir "exit" para parar):
> hola
> exit
```

11. Ejercicio resumen

Implemente un programa en C usando tuberías similar a los existentes en los ficheros `pipe.c` y `pipe2.c`, pero en este caso que un proceso genere dos números aleatorios flotantes y envie la suma de ellos al otro proceso para que éste muestre su resultado.

12. Ejercicio resumen

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 leídas desde teclado y las envía mediante mensajes a un proceso servidor cada vez que pulsamos INTRO.

2. El servidor recibirá los mensajes y contará el número de caracteres recibidos exceptuando el fin de cadena (un espacio en blanco se considerará un carácter). Tras esto, el servidor mandará un mensaje al cliente, **por otra cola distinta**, con la cadena "Número de caracteres recibidos: X", siendo X el número de caracteres calculados.

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

   a) Una cola servirá para que el cliente le envíe al servidor las cadenas de texto.
   De esta cola leerá el servidor para obtener dichas cadenas y analizarlas para contar el número de caracteres que tienen.

   b) Otra cola por la que el servidor enviará al cliente el número de caracteres calculados en la cadena de texto recibida por la primera cola.
   De esta segunda cola leerá el cliente para mostrar el número de caracteres calculados que le ha enviado el servidor.

   Se han de tener en cuenta los siguientes items:

   - La cola de mensajes para el texto "Número de caracteres recibidos: X", 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 deberá mandar el mensaje de salida, para forzar al cliente a parar.

3. Asegurar que el nombre de las colas sea diferente para su ejecución en un sistema compartido. Puede usar la idea de anexar el *login* al nombre de la cola.

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. Capture 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*.

13. Ejercicio resumen

Realice el mismo programa anterior pero en este caso vamos a utilizar la primitiva fork(), el padre será el servidor y el hijo será el cliente.

Tenga en cuenta también las siguientes directrices:

- 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*.

Referencias

[1] Proyecto GNU. *GNU C Library*. 2015. URL: http://www.gnu.org/software/libc/libc.html.

[2] The IEEE y The Open Group. *POSIX.1-2017 – The Open Group Base Specifications Issue 7, 2018 edition*. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[3] Brian W. Kernighan, Dennis Ritchie y Dennis M. Ritchie. *C Programming Language (2nd Edition)*. 2.ª ed. Pearson Educación, 1991. ISBN: 968-880-205-0.

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

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

[6] Varios colaboradores. *Glibc — Wikipedia, La enciclopedia libre*. [Internet; descargado el 12 de marzo de 2021]. 2020. URL: https://es.wikipedia.org/w/index.php?title=Glibc&oldid=128337445.

[7] Varios colaboradores. *POSIX — Wikipedia, The Free Encyclopedia*. [Internet; descargado el 12 de marzo de 2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=POSIX&oldid=1007100348.

[8] Wikipedia. *Dennis Ritchie – Wikipedia, La enciclopedia libre*. 2021. URL: http://es.wikipedia.org/wiki/Dennis_Ritchie.