Práctica3: Programación POSIX

Programación y Administración de Sistemas

2014-2015

Pedro Antonio Gutiérrez Peña [*]
pagutierrez@uco.es

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

22 de marzo de 2015

Resumen

Entre otras cosas, el estándar POSIX define una serie de funciones (comportamiento, parámetros y valores devueltos) que deben proporcionar todos los sistemas operativos que quieran satisfacer el estándar. De esta forma, si una aplicación hace un buen uso de estas funciones deberá compilar y ejecutarse *sin problemas* en cualquier sistema operativo POSIX. En esta práctica vamos a conocer qué es el estándar POSIX, y una implementación de la biblioteca C de POSIX, la biblioteca libre *GNU C Library* o *glibc*. En Moodle se adjunta el fichero `pas-practica3.tar.gz` que contiene código de ejemplo de las funciones que iremos viendo en clase. También se ha subido un fichero `Makefile` que os puede servir para preparar los ejercicios que se piden en esta práctica. Se entregará el código de los programas propuestos en los **Ejercicios Resumen**, junto con el `Makefile` mencionado y un fichero de texto que aclare las particularidades de los mismos (todo comprimido en un único archivo). Se valorará la utilización de comentarios, la máxima modularidad en el código, la comprobación de errores en los argumentos de los programas y la claridad en las salidas generadas. Todos los programas deben prepararse para funcionar correctamente en la máquina `ts.uco.es`. El día tope para la entrega de este guión de prácticas es el **domingo 26 de abril a las 23.55h**. La entrega se hará utilizando la tarea en Moodle habilitada al efecto. En caso de que dos alumnos entreguen códigos copiados, no se puntuarán ninguno de los dos. Comprueba que los comportamientos de los programas son similares a los esperados en los ejemplos de ejecución.

[*]Parte de los contenidos de este guión corresponden al preparado por Javier Sánchez Monedero en el curso académico 2011/2012 [1]

Índice

1. Introducción a POSIX **3**

2. Objetivos **4**

3. Entrega de prácticas **4**

4. Documentación de POSIX y las bibliotecas **5**

5. Procesado de línea de comandos tipo POSIX **5**
5.1. Introducción y documentación . . . 5
5.2. Ejercicios . . . 6

6. Variables de entorno **6**
6.1. Introducción y documentación . . . 6
6.2. Ejercicio de ejemplo . . . 6

7. Obtención de información de un usuario **6**
7.1. Introducción y documentación . . . 6
7.2. Ejercicios . . . 7

8. Ejercicio resumen 1 **7**

9. Ejercicio resumen 2 **9**

10. Creación de procesos (`fork` y `exec`) **10**
10.1. Introducción y documentación . . . 10
10.2. Ejercicios y ejemplos . . . 11

11. Señales entre procesos **14**
11.1. Introducción y documentación . . . 14
11.2. Ejercicios y ejemplos . . . 15

12. Comunicación entre procesos POSIX **16**
12.1. Tuberías . . . 16
12.2. Memoria compartida . . . 18
12.3. Semáforos . . . 18
12.4. Colas de mensajes . . . 19
12.4.1. Creación o apertura de colas . . . 19
12.4.2. Recepción y envío de mensajes desde/a colas . . . 20
12.4.3. Cierre y/o eliminación de colas . . . 21
12.5. Ejemplo simple de uso de colas . . . 21
12.6. Ejemplo cliente-servidor de uso de colas . . . 25

13. Ejercicio resumen 3 **26**

14. Ejercicio resumen 4 **27**

Referencias **28**

1. Introducción a POSIX

Figura 1: Mascota del proyecto GNU (http://www.gnu.org/)

POSIX es el acrónimo de *Portable Operating System Interface*; la *X* viene de UNIX como seña de identidad de la API (*Application Programming Interface*, interfaz de programación de aplicaciones). Son una familia de estándares de llamadas al sistema operativo 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. Estos estándares surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces 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].

GNU C Library, comúnmente conocida como `glibc` es la biblioteca estándar de lenguaje C de GNU. Se distribuye bajo los términos de la licencia GNU LGPL[1]. Esta biblioteca sigue todos los estándares más relevantes como ISO C99 y POSIX.1-2008 [4]. `gnulib`, también conocida como "Biblioteca de portabilidad de GNU" es una colección de subrutinas diseñadas para usarse en distintos sistemas operativos y arquitecturas. El objetivo de esta segunda biblioteca es facilitar el desarrollo multi-plataforma de aplicaciones de *software* libre.

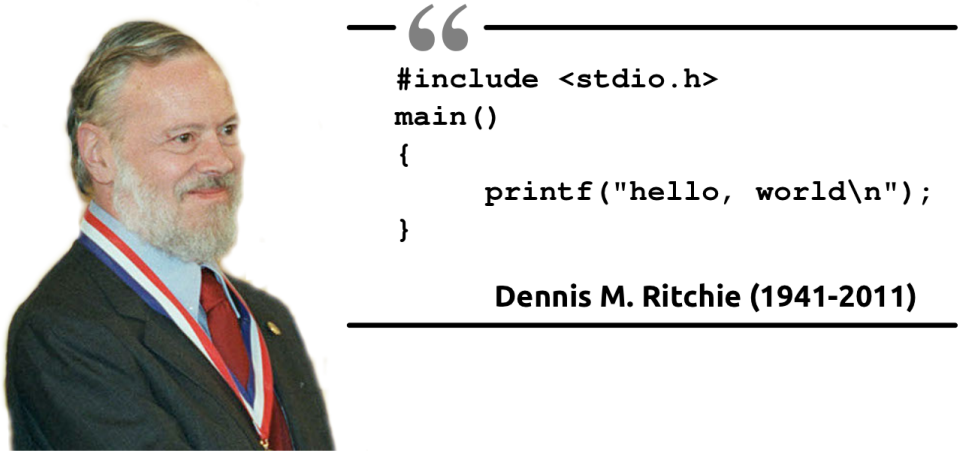

Figura 3: **Dennis MacAlistair Ritchie**. Colaboró en el diseño y desarrollo de los sistemas operativos Multics y Unix, así como el desarrollo de varios lenguajes de programación como el C, tema sobre el cual escribió un célebre clásico de las ciencias de la computación junto a Brian Wilson Kernighan: *El lenguaje de programación C* [5].

[1] http://es.wikipedia.org/wiki/GNU_General_Public_License

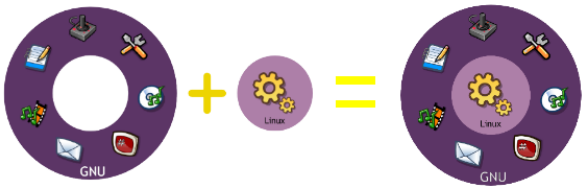

Figura 2: GNU + Linux = GNU/Linux

En los sistemas en los que se usan, estas bibliotecas de C proporcionan y definen las llamadas al sistema y otras funciones básicas, y son utilizadas por casi todos los programas. Sobre todo, es muy usada en los sistemas GNU y en el kernel de Linux. 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[2].

`glibc` y `gnulib` son muy *portables* y soportan gran cantidad de plataformas de *hardware* [6]. En los sistemas Linux se instalan normalmente con el nombre de `libc6` y `gnulib` respectivamente. No debe confundirse con GLib[3], otra biblioteca que proporciona estructuras de datos avanzadas como árboles, listas enlazadas, tablas hash, etc. y un entorno de orientación a objetos en C.

En estas prácticas utilizaremos la biblioteca `glibc` como implementación de programación del API POSIX. Algunas distribuciones de GNU/Linux, como Debian o Ubuntu utilizan una variante de la `glibc` llamada `eglibc`[4], adaptada para sistemas empotrados, pero a efectos de programación no debería haber diferencias.

2. Objetivos

- Conocer algunas rutinas POSIX y su implementación `glibc`.
- Aprender a utilizar bibliotecas externas en nuestros programas.
- Aprender cómo funcionan internamente algunas partes de GNU/Linux.
- Mejorar la programación viendo ejemplos hechos por los desarrolladores de las bibliotecas.
- Aprender a comunicar aplicaciones utilizando paso de mensajes.

3. Entrega de prácticas

Se pedirá la entrega y defensa de cuatro ejercicios resumen que integran varios de los conceptos y funciones estudiados en clase. **Las prácticas deben realizarse a nivel individual** y se utilizarán mecanismos para comprobar que se han realizado y entendido.

[2] http://es.wikipedia.org/wiki/Controversia_por_la_denominaci%C3%B3n_GNU/Linux
[3] http://library.gnome.org/devel/glib/,http://es.wikipedia.org/wiki/GLib
[4] http://www.eglibc.org/home

4. Documentación de POSIX y las bibliotecas

Documentación POSIX.1-2008: Especificación del estándar POSIX. Dependiendo de la parte que documente es más o menos pedagógica[5].

Documentación *GNU C Library*: Esta documentación incluye muchos de los conceptos que ya habéis 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[6].

5. Procesado de línea de comandos tipo POSIX

5.1. Introducción y documentación

Los parámetros que procesa un programa en sistemas POSIX deben seguir un estándar de formato y respuesta esperada[7]. Un resumen de lo definido en el estándar es lo siguiente:

- Una opción es un guión (`-`) seguido de un carácter alfanumérico, por ejemplo, `-o`.
- Una opción requiere un parámetro que 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`.
- Las opciones pueden aparecer muchas veces.
- El parámetro `--` indica el fin de las opciones en cualquier caso.

La función `getopt()`[8] del estándar ayuda a desarrollar el manejo de las opciones siguiendo las directrices POSIX.1-2008. `getopt()` está implementada en `libc`[9] [10]. Puedes ver un código de ejemplo simple dentro del fichero `ejemplo-getopt.c` de los que hay en Moodle. El fichero `ejemplo-getoptlong.c` contiene un ejemplo de procesado de órdenes al estilo de GNU (por ejemplo, `--help` y `-h` como órdenes compatibles). Este segundo ejemplo no lo veremos en clase.

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

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

[7] 12.1 Utility Argument Syntax, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

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

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

[10] Notas sobre portabilidad en http://www.gnu.org/software/gnulib/manual/gnulib.html#getopt

5.2. Ejercicios

El código de ejemplo de `getopt()` de la documentación de `glibc`[11] se encuentra en el fichero `ejemplo-getopt.c`. Lee el código, compílalo, ejecútalo para comprobar que admite las opciones de parámetros POSIX. Trata de entender el código y añade más opciones (por ejemplo una sin parámetros y otra con parámetros). Debes consultar la sección *Using the `getopt` function* de la documentación[12] para saber cómo funciona `getopt`, qué valores espera y qué comportamiento tiene.

6. Variables de entorno

6.1. 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 utilizas `bash`, puedes consultar las variables de entorno de tu sesión con el comando `env`. También puedes consultar o modificar el valor de una variable de forma individual:

```
$ env
$ ...
$ echo $LANG
es_ES.UTF-8
$ export LANG=en_GB.UTF-8
```

6.2. Ejercicio de ejemplo

Utilizando la función `getenv()`[13], haz un programa que, dependiendo del idioma de la sesión de usuario, imprima un mensaje con su carpeta personal en castellano o en inglés. Este código de ejemplo se encuentra en el fichero `ejemplo-getenv.c`.

7. Obtención de información de un usuario

7.1. Introducción y documentación

En los sistemas operativos, la base de datos de usuarios puede ser local y/o remota. Por ejemplo, en GNU/Linux puedes ver los usuarios y grupos locales en los ficheros `/etc/passwd` y `/etc/group`. 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.

[11] http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html
[12] http://www.gnu.org/software/libc/manual/html_node/Using-Getopt.html
[13] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getenv.html

En el caso de GNU/Linux, la autenticación 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.). Por ejemplo, la llamada `getpwuid()` devuelve una estructura con información de un usuario. El sistema POSIX 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.

7.2. Ejercicios

Puedes ver las funciones y estructuras de acceso a la información de usuarios y grupos en los siguientes ficheros de cabecera: `/usr/include/pwd.h` y `/usr/include/grp.h`. La función `getpwnam()`[14] permite obtener información de un usuario. Mira el programa de ejemplo de `getpwnam()` y amplíalo para utilizar `getgrgid()`[15] para obtener el nombre del grupo del usuario a través del identificador del grupo. Puedes ver código de ejemplo en el fichero `ejemplo-infousuario.c`.

8. Ejercicio resumen 1

El fichero de código de este ejercicio será `ejercicio1.c` y el ejecutable se denominará `ejercicio1`. Realizar un programa que obtenga e imprima la información de un usuario (todos los campos de la estructura `passwd`) pasado por parámetro. La opción `-n` servirá para especificar el usuario por nombre de usuario (p.ej. `i02gupep`), mientras que la opción -u servirá para especificar el usuario por UID (p.ej. `17468`). Si además se le pasa la opción `-g`, el programa deberá buscar e imprimir la información del grupo del usuario (GID del grupo y nombre del grupo). Si se le pasa la opción `-e` imprimirá todos los mensajes en inglés, y la opción `-s` hará que los imprima en castellano. Si no se le pasa ni `-e` ni `-s` se mirará la variable de entorno `LANG` para mostrar la información. Si no se le especifica usuario se utilizará el usuario actual, definido en la variable de entorno `USER`. Por ejemplo, las siguientes llamadas serían válidas:

```
# Obtener la información de un usuario usando el idioma
# configurado en LANG
i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1 -n i02gupep
Nombre de usuario: PEDRO ANTONIO GUTIERREZ PE?A
Identifador de usuario: 17468
Contraseña: *
Carpeta inicio: /home/i02gupep
Intérprete por defecto: /bin/bash
```

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

[15] http://www.gnu.org/software/libc/manual/html_node/Group-Database.html

```
9
10 # Obtener la información del usuario actual (USER)
11 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1
12 Nombre de usuario: PEDRO ANTONIO GUTIERREZ PE?A
13 Identifador de usuario: 17468
14 Contraseña: *
15 Carpeta inicio: /home/i02gupep
16 Intérprete por defecto: /bin/bash
17
18 # Obtener la información de un usuario forzando el
19 # idioma en castellano
20 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1 -s
21 Nombre de usuario: PEDRO ANTONIO GUTIERREZ PE?A
22 Identifador de usuario: 17468
23 Contraseña: *
24 Carpeta inicio: /home/i02gupep
25 Intérprete por defecto: /bin/bash
26
27 # Obtener la información de un usuario añadiendo la
28 # información de su grupo principal e imprimiendo todo en inglés
29 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1 -n i02gupep -eg
30 User name: PEDRO ANTONIO GUTIERREZ PE?A
31 User id: 17468
32 Password: *
33 Home: /home/i02gupep
34 Default shell: /bin/bash
35 Main Group Number: 700
36 Main Group Name: upi0
37
38 # Llamadas incorrectas
39 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1 -es
40 No se puede activar dos idiomas al mismo tiempo
41 Two languages cannot be used at same time
42 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio1 -e -n i02gupep -u 17468
43 UID and user name should not be used at the same time
```

El control de errores debe ser el siguiente:

- Asegurar que se pasa el nombre del usuario válido y qué existe en la máquina.
- Las opciones `-n` y `-u` no pueden activarse a la vez.
- Las opciones `-e` y `-s` no pueden activarse a la vez.

Sugerencia: empezar por el código de ejemplo `ejemplo-getopt.c`. Primero debe procesarse la línea de comandos para asegurar que no faltan o sobran opciones. Después, debe incluirse la lógica del programa dependiendo de las opciones que se hayan reconocido. Se pueden incluir funciones para simplificar el código de `main`. Por ejemplo, una función para

imprimir la información del usuario en castellano/inglés a la que se le pase la estructura `struct passwd` y una opción booleana. El esquema general puede ser el siguiente:

1. Procesar opciones de entrada.
2. Usar la variable de entorno LANG si las *flags* de las opciones `-e` y `-s` están a cero.
3. Llamar a una función que imprima la información de un usuario en inglés o castellano con las opciones (estructura, grupo si/no, idioma).

9. Ejercicio resumen 2

El fichero de código de este ejercicio será `ejercicio2.c` y el ejecutable se denominará `ejercicio2`. Realizar un programa que imprima la información de un determinado grupo. En este ejercicio utilizaremos `getoptlong`, que nos va a permitir especificar opciones en formato corto o largo. El grupo del cual se quiere obtener información se pasa como argumento en la línea de comandos del programa, con la opción `-g` o `--group`. Si esta opción no se indicase, habría que utilizar el grupo primario del usuario que invoca el programa. La información a imprimir será la siguiente:

- `gid` del grupo (número entero).
- Nombre del grupo (cadena de texto).

Esta información se imprimirá en idioma inglés o en español según el uso de las opciones `-e/--english` y `-s/--spanish` (del mismo modo que en el ejercicio anterior). Además, se creará una opción de ayuda `-h/--help` para mostrar información sobre el uso del programa. Esa información también se mostrará cuando el usuario cometa cualquier error en la invocación del ejercicio. A continuación, se incluyen ejemplos de invocación del programa con la salida esperada:

```
1  # Obtener la información de un grupo usando el idioma
2  # configurado en LANG
3  i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 -g cc1
4  Identificador del grupo: 601
5  Nombre del grupo: cc1
6  i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 --group upi0
7  Identificador del grupo: 700
8  Nombre del grupo: upi0
9
10 # Obtener la información de un grupo forzando el
11 # idioma en inglés
12 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 -e --group cc1
13 Group Identifier: 601
14 Group name: cc1
15
16 # Obtener la información del grupo del usuario actual,
17 # forzando el idioma en castellano
18 i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 --spanish
```

```
Obteniendo el grupo primario del usuario i02gupep...
Identificador del grupo: 700
Nombre del grupo: upi0

# Ayuda
i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 --help
Uso del programa: ejercicio2 [opciones]
Opciones:
-h, --help              Imprimir esta ayuda
-g, --group=GRUPO       Grupo a analizar
-e, --english           Mensajes en inglés
-s, --spanish           Mensajes en castellano

# Errores
i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 --spanish -e
Uso del programa: ejercicio2 [opciones]
Opciones:
-h, --help              Imprimir esta ayuda
-g, --group=GRUPO       Grupo a analizar
-e, --english           Mensajes en inglés
-s, --spanish           Mensajes en castellano
i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio2 -i
Opción desconocida `-i'.
Uso del programa: ejercicio2 [opciones]
Opciones:
-h, --help              Imprimir esta ayuda
-g, --group=GRUPO       Grupo a analizar
-e, --english           Mensajes en inglés
-s, --spanish           Mensajes en castellano
```

10. Creación de procesos (`fork` y `exec`)

10.1. 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)[16]. 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:

```
$ find . -name "*.c" -print | wc -l
```

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

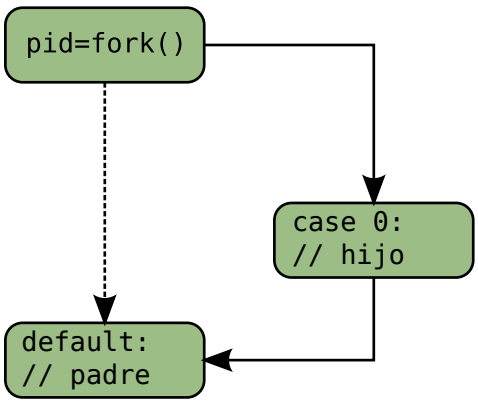

Figura 4: Esquema de llamadas y procesos generados por `fork()` en el ejemplo.

En C se crea un subproceso llamando a la función `fork()`[17] [18]. Tienes 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. En el proceso original, el valor devuelto de `fork` será el identificador del proceso hijo. En cambio, en el proceso hijo el valor devuelto por `fork` será `0`.

10.2. Ejercicios y ejemplos

El listado siguiente (`ejemplo-fork.c`) es un ejemplo de uso de `fork` que controla qué tipo de proceso es el que ejecuta el código utilizando el valor devuelto por la función, así como otras funciones POSIX para obtener información de los procesos (puedes ver un esquema de las subprocesos creados en la Figura 4):

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      pid_t rf;
8      rf = fork();
9      switch (rf)
10     {
11     case -1:
12         printf ("No he podido crear el proceso hijo \n");
13         exit(-1);
14     case 0:
15         printf ("Soy el hijo, mi PID es %d y mi PPID es %d \n",
               getpid(), getppid());
```

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

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

```
16          sleep (5);
17          break;
18      default:
19          printf ("Soy el padre, mi PID es %d y el PID de mi hijo es
                %d \n", getpid(), rf);
20          sleep (10);
21      }
22
23      printf ("Final de ejecución de %d \n", getpid());
24      exit (0);
25  }
```

Un ejemplo de llamada a este código sería:

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

En este ejemplo, el proceso padre no queda bloqueado esperando al hijo, prueba a poner un valor menor de espera (`sleep`) para el padre que para el hijo:

```
1  $ ./ejemplo-fork
2  Soy el padre, mi PID es 23437 y el PID de mi hijo es 23438
3  Soy el hijo, mi PID es 23438 y mi PPID es 23437
4  Final de ejecución de 23437
5  $ Final de ejecución de 23438
```

Si quisiéramos que el proceso padre esperase a que el proceso (o los procesos) hijos terminasen, debemos utilizar la función `wait`[19]. El valor devuelto por la función `wait` es el PID del proceso hijo que terminó en último lugar. El estado de terminación del proceso (código de error), se recoge en la variable `status` pasada como argumento. Un ejemplo:

```
1  #include <unistd.h>
2  #include <sys/wait.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8      pid_t pid;
9      int status, died;
10
11     switch(pid=fork())
12     {
13     case -1:
14         printf("can't fork\n");
```

[19] www.gnu.org/software/libc/manual/html_node/Process-Completion.html

```
15          exit(-1);
16      case 0 :
17          sleep(2); // código que ejecuta el hijo
18          exit(3);
19      default:
20          died= wait(&status); // código que ejecuta el padre
21      }
22      printf("died= %d (%d)\n",died,WEXITSTATUS(status));
23
24      return(0);
25  }
```

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()` permiten iniciar un programa dentro de otro programa. En lugar de crear una copia del proceso, como `fork()`, `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. Podemos ver este mecanismo en el siguiente código de ejemplo:

```
1   #include <unistd.h>
2   #include <sys/wait.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   int main()
7   {
8       pid_t pid;
9       int status, died;
10
11      char *args[] = {"/bin/ls", "-r", "-t", "-l", (char *) 0 };
12
13      switch(pid=fork())
14      {
15      case -1:
16          printf("can't fork\n");
17          exit(-1);
18      case 0 :
19          printf("child\n"); // código que ejecuta el hijo
20          execv("/bin/ls", args);
21      default:
22          died= wait(&status); // código que ejecuta el padre
23      }
24
25      return(0);
26  }
```

11. Señales entre procesos

11.1. Introducción y documentación

Las señales entre programas son interrupciones *software* que se generan para informar a un proceso de la ocurrencia de un evento. Otras formas alternativas o complementarias de comunicación entre procesos son las que veremos en la sección 12. 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 5, fuente Wikipedia[20]).

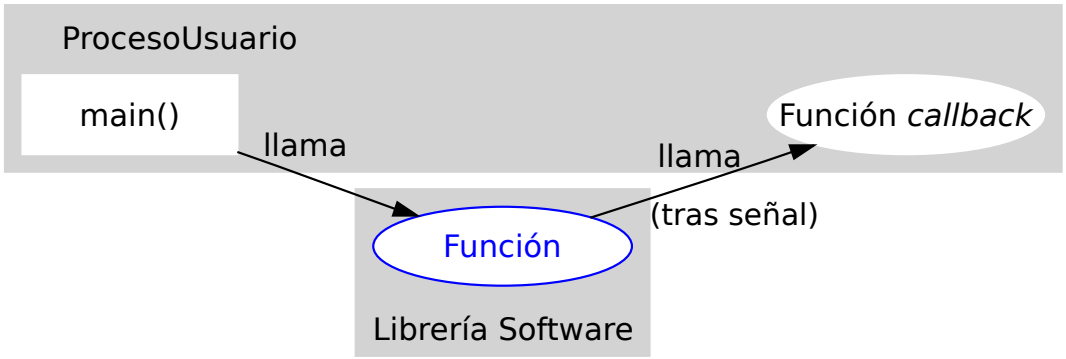

Figura 5: Esquema del funcionamiento de las *callbacks* o *retrollamadas*.

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

```
1  #include <signal.h>
2
3  // El prototipo de la función es el siguiente
4  // sighandler_t signal(int signum, sighandler_t handler);
5  // sighandler_t representa un puntero a una función que devuelve
6  // void y recibe un entero
7  ...
8
9  // Función que va a manejar la señal TERM
10 void mifuncionManejadoraTerm(int signal)
11 {
12   ....
13 }
```

[20] http://en.wikipedia.org/wiki/Callback_%28computer_science%29
[21] http://pubs.opengroup.org/onlinepubs/9699919799/functions/signal.html

```
14
15 int main(void){
16
17   ...
18   // Posible llamada
19   signal(SIGTERM, mifuncionManejadoraTerm);
20   // Donde SIGTERM es 15, y mifuncionManejadoraTerm es un manejador
         de la señal, un puntero a función
21   ...
22
23 }
```

11.2. Ejercicios y ejemplos

El código de ejemplo `ejemplo-signal.c`[22] contiene ejemplos captura de señales POSIX enviadas a un programa. Recuerda 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.

Ejemplo de ejecución:

```
1 $  ./ejemplo-signal
2 No puedo asociar la señal SIGKILL al manejador!
3 Capturé la señal SIGHUP y no salgo!
4 Capturé la señal SIGTERM y voy a salir de manera ordenada
5 Hasta luego... cerrando ficheros...
6 Hasta luego... cerrando ficheros...
7 Terminado (killed)
```

Esa salida se obtiene al utilizar en otro terminal los siguientes comandos:

```
1 $  ps aux | grep signal
2 pedroa    3613  0.0  0.0   3720   400 pts/1    S+   19:09   0:00 ./
     ejemplo-signal
3 $ kill -SIGHUP 3613
4 $ killall ejemplo-signal
5 $ killall -SIGKILL ejemplo-signal
```

Hacer un programa que pida al usuario dos números y calcule la división del primero entre el segundo. Capturar la excepción de división por cero (sin comprobar que el segundo argumento es cero) y, en ese caso, dividir por uno.

```
1 $ ./ejemplo-signal-division
2 Introduce el dividendo: 1
3 Introduce el divisor: 2
4 Division=0
5 $ ./ejemplo-signal-division
6 Introduce el dividendo: 1
```

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

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

12. 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:

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

12.1. Tuberías

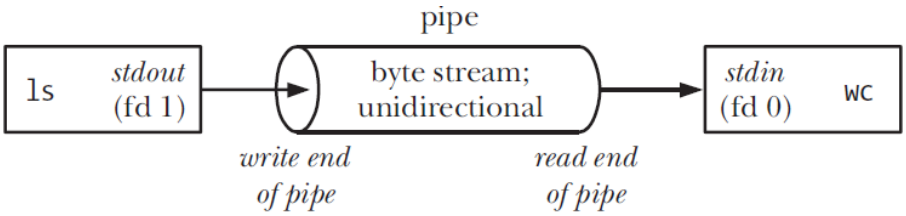

Figura 6: Intercomunicación entre procesos utilizando la tubería `ls | wc -l`. "*write end*" significa extremo de escritura y "*read end*" extremo de lectura.

Las tuberías ya se han tratado en las práctica de programación de la `shell`. 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 6.

Existen dos tipos de tuberías: tuberías anónimas y tuberías con nombre. La tubería que vimos 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`[23]:

[23] http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

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

Esta función crear una tubería anónima y te 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`[24] y `write`[25], cuyo uso es similar a `fread` y `fwrite`. Una vez utilizados los extremos de lectura y/o escritura, los podemos cerrar con `close`[26].

En el siguiente ejemplo[27], se muestra como se escribe y lee una cadena "Hola mundo" en un *pipe* anónimo, utilizando para ello `fork()`. Faltaría hacer un correcto control de errores para las funciones `read`, `write` y `close`.

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

...

  int fildes[2];
  const int BSIZE = 100;
  char buf[BSIZE];
  ssize_t nbytes;
  int status;

  status = pipe(fildes);
  if (status == -1 ) {
    // Ocurrió un error al crear la tubería
    ...
  }

  switch (fork()) {
    // Ocurrió un error al hacer fork()
    case -1:
      break;


    // El hijo lee desde la tubería
    case 0:
      // No necesitamos escribir
      close(fildes[1]);
      // Leer usando READ
      // -> Habría que comprobar errores!
      nbytes = read(fildes[0], buf, BSIZE);
      // En este punto, una lectura adicional,
```

[24] http://pubs.opengroup.org/onlinepubs/9699919799/functions/read.html
[25] http://pubs.opengroup.org/onlinepubs/9699919799/functions/write.html
[26] http://pubs.opengroup.org/stage7tc1/functions/close.html
[27] Extraído de http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

```
32        // hubiera llegado a FEOF
33        // Cerrar el extremo de lectura
34        close(fildes[0]);
35        exit(EXIT_SUCCESS);
36
37      // El padre escribe en la tubería
38      default:
39        // No necesitamos leer
40        close(fildes[0]);
41        // Escribimos datos en la tubería
42        // -> Habría que comprobar errores!
43        write(fildes[1], "Hola Mundo!!\n", 14);
44        // El hijo verá FEOF (por hacer close)
45        close(fildes[1]);
46        exit(EXIT_SUCCESS);
47    }
```

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)`[28] 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.

12.2. 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 ⇒ modo núcleo ⇒ modo usuario). Como contrapartida, se debe realizar una sincronización de las lecturas y escrituras, que implica una mayor dificultad en la programación. Aunque la memoria compartida está especificada en el estándar POSIX, no se va a tratar en esta práctica.

12.3. 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 los habéis estudiado en profundidad 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.

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

12.4. 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**.
- Solo se puede leer o escribir **un** mensaje de la cola (no se pueden hacer operaciones sobre múltiples mensajes).
- Al contrario que con las tuberías, en una cola podemos tener múltiples lectores o escritores. Por supuesto, 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) y 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[29]. Como programadores, serán tres las operaciones que realizaremos con las colas de mensajes[30]:

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`.

12.4.1. Creación o apertura de colas

La función a utilizar es `mq_open`[31]:

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

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

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

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

donde `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, `mode` corresponde a los permisos con los cuales creamos la cola y `attr` es un puntero a una estructura `struct mq_attr` que contiene propiedades de la cola. 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`). 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. Solo en aquellos casos en que indiquemos que queremos crear la cola (`O_CREAT`), tendrán sentido los argumentos opcionales `mode` y `attr`. El primero 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), mientras que el segundo 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).

12.4.2. Recepción y envío de mensajes desde/a colas

Para recibir un mensaje desde una cola utilizaremos la función `mq_receive`[32]:

```
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`) y almacenarlo 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`.

Por último, 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.

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

Para mandar un mensaje a una cola utilizaremos la función `mq_send`[33]:

```
#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 indicamos 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`).

12.4.3. Cierre y/o eliminación de colas

Para cerrar una cola (dejar de utilizarla pero que siga existiendo) utilizaremos la función `mq_close`[34]:

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

donde `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`).

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`[35]:

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

donde `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`).

12.5. Ejemplo simple de uso de colas

Vamos a ver un primer ejemplo simple en el que hacemos uso de dos elementos de POSIX: `fork` y colas de mensajes. El código correspondiente se encuentra en el fichero `ejemplo-mq.c`, cuyo contenido es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <mqueue.h>
#include <errno.h>
```

[33] http://pubs.opengroup.org/stage7tc1/functions/mq_send.html
[34] http://pubs.opengroup.org/stage7tc1/functions/mq_close.html
[35] http://pubs.opengroup.org/stage7tc1/functions/mq_unlink.html

```
7  #include <sys/wait.h>
8
9  #define MAX_SIZE    5
10 #define QUEUE_NAME  "/una_cola"
11
12 int main() {
13   // Descriptor de la cola
14   mqd_t mq;
15
16   // Buffer para la lectura/escritura
17   char buffer[MAX_SIZE + 1];
18
19   // Atributos de la cola
20   struct mq_attr attr;
21
22   // Resultado de las operaciones
23   int resultado;
24
25   // Para realizar el fork
26   pid_t rf;
27   int died, status;
28
29   // Numero aleatorio a generar
30   int numeroAleatorio;
31
32   // Realizar el fork
33   rf = fork();
34
35   // Inicializar los atributos de la cola
36   attr.mq_maxmsg = 10;        // Maximo número de mensajes
37   attr.mq_msgsize = MAX_SIZE; // Maximo tamaño de un mensaje
38
39   switch (rf)
40   {
41     // Error
42     case -1:
43       printf ("No he podido crear el proceso hijo \n");
44       exit(1);
45
46     // Hijo
47     case 0:
48
49       /* Apertura de la cola
50          O_CREAT: si no existe, se crea
51          O_RDWR: lectura/escritura
52          O_RDONLY: solo lectura
53          O_WRONLY: solo escritura
54          0644: permisos rw-r--r--
55          attr: estructura con atributos para la cola  */
56       mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
57       if(mq==-1){
58         perror("[HIJO]: Error en la apertura de la cola");
59         exit(-1);
60       }
61
62       printf ("[HIJO]: mi PID es %d y mi PPID es %d\n", getpid(), getppid());
63
```

```
64        // Rellenamos el buffer que vamos a enviar
65        // Semilla de los números aleatorios,
66        //    establecida a la hora actual
67        srand(time(NULL));
68        // Número aleatorio entre 0 y 4999
69        numeroAleatorio = rand()%5000;
70        sprintf(buffer,"%d",numeroAleatorio);
71
72        printf("[HIJO]: genero el mensaje \"%s\"\n", buffer);
73
74        // Mandamos el mensaje
75        printf("[HIJO]: enviando mensaje...\n");
76        resultado = mq_send(mq, buffer, MAX_SIZE, 0);
77        if(resultado == -1){
78          perror("[HIJO]: Error al enviar mensaje");
79          exit(-1);
80        }
81
82        printf("[HIJO]: Mensaje enviado! Salgo...\n");
83        // Cerrar la cola
84        if(mq_close(mq) == -1){
85          perror("[HIJO]: Error cerrando la cola");
86          exit(-1);
87        }
88        break;
89
90      // Padre
91      default:
92
93        /* Apertura de la cola */
94        mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY, 0644, &attr);
95        if(mq==-1){
96          perror("[PADRE]: Error en la apertura de la cola");
97          exit(-1);
98        }
99
100       printf ("[PADRE]: mi PID es %d y el PID de mi hijo es %d \n", getpid(), rf)
              ;
101       printf ("[PADRE]: recibiendo mensaje (espera bloqueante)...\n");
102
103       // Recibimos un mensaje a través de la cola
104       resultado = mq_receive(mq, buffer, MAX_SIZE, NULL);
105       if(resultado <= 0){
106         perror("[PADRE]: Error al recibir el mensaje");
107         exit(-1);
108       }
109
110       // Imprimimos el mensaje recibido
111       printf("[PADRE]: el mensaje recibido es \"%s\"\n", buffer);
112
113       // Cerrar la cola
114       if(mq_close(mq) == -1){
115         perror("[PADRE]: Error cerrando la cola");
116         exit(-1);
117       }
118       // Eliminar la cola
119       if(mq_unlink(QUEUE_NAME) == -1){
```

```
120         perror("[PADRE]: Error eliminando la cola");
121         exit(-1);
122       }
123       printf("[PADRE]: Cola cerrada.\n");
124       died=wait(&status);
125       printf("[PADRE]: El proceso hijo con PID %d ha salido con código de error %
              d. Salgo...\n", died, status);
126   }
127
128   exit(0);
129 }
```

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

- 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`). De esta forma, hubiéramos intercambiado una cantidad menor de *bytes*. 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 (el padre 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.
- 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 muestran dos ejemplos de ejecución de este programa:

```
1 i02gupep@NEWTS:~/pas/1415/p3$ ./ejemplo-mq
2 [PADRE]: mi PID es 4216 y el PID de mi hijo es 4217
3 [PADRE]: recibiendo mensaje (espera bloqueante)...
4 [HIJO]: mi PID es 4217 y mi PPID es 4216
```

```
[HIJO]: genero el mensaje "4997"
[HIJO]: enviando mensaje...
[HIJO]: Mensaje enviado! Salgo...
[PADRE]: el mensaje recibido es "4997"
[PADRE]: Cola cerrada.
[PADRE]: El proceso hijo con PID 4217 ha salido con código de error
    0. Salgo...
i02gupep@NEWTS:~/pas/1415/p3$ ./ejemplo-mq
[PADRE]: mi PID es 4859 y el PID de mi hijo es 4860
[PADRE]: recibiendo mensaje (espera bloqueante)...
[HIJO]: mi PID es 4860 y mi PPID es 4859
[HIJO]: genero el mensaje "3506"
[HIJO]: enviando mensaje...
[HIJO]: Mensaje enviado! Salgo...
[PADRE]: el mensaje recibido es "3506"
[PADRE]: Cola cerrada.
[PADRE]: El proceso hijo con PID 4860 ha salido con código de error
    0. Salgo...
```

12.6. Ejemplo cliente-servidor de uso de colas

Vamos a ver ahora un segundo ejemplo[36] (ficheros de código `common.h`, `servidor.c` y `cliente.c`). Este segundo ejemplo contempla dos procesos independientes, de forma que el `servidor` crea una cola y espera a que el `cliente` introduzca mensajes en esa cola. Por cada mensaje recibido, el `servidor` imprime su valor en consola. El programa `cliente` lee por teclado los mensajes a enviar y realiza un envío cada vez que pulsamos `INTRO`. La comunicación finaliza y los programas terminan, cuando el `cliente` manda el mensaje de salida (establecido como "`exit`" en `common.h`). Hemos 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 debemos lanzar el `servidor`:

```
i02gupep@NEWTS:~/pas/1415/p3/codigo-ejercicios$ ./servidor
```

quedando a la espera de los mensajes del `cliente`. Posteriormente, lanzamos el `cliente` desde otra terminal:

```
i02gupep@NEWTS:~/pas/1415/p3/codigo-ejercicios$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
>
```

quedando a la espera de escribamos un mensaje. Escribimos "`hola`" y pulsamos `INTRO`:

```
i02gupep@NEWTS:~/pas/1415/p3/codigo-ejercicios$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
> hola
```

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

```
>
```

El mensaje ya se ha enviado. Si volvemos a la terminal del `servidor`, podremos comprobarlo:

```
i02gupep@NEWTS:~/pas/1415/p3/codigo-ejercicios$ ./servidor
Recibido el mensaje: hola
```

Si ahora mandamos el mensaje "`exit`" desde el `cliente`:

```
i02gupep@NEWTS:~/pas/1415/p3/codigo-ejercicios$ ./cliente
Mandando mensajes al servidor (escribir "exit" para parar):
> hola
> exit
```

veremos que el `servidor` se para. Analiza el código de los dos programas. Verás como toda la comunicación se basa en paso de mensajes, utilizando las colas de mensajes POSIX.

13. Ejercicio resumen 3

El fichero de código de este ejercicio será `ejercicio3.c` y el ejecutable se denominará `ejercicio3`. Cuando la comunicación 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*). Este el caso del código del fichero `ejemplo-mq.c`, por lo que en este ejercicio deberás modificar dicho código **para que se haga uso de *pipes* en lugar de colas de mensajes**. Consulta las funciones que se han estudiado en la sección 12.1 y el ejemplo correspondiente. Además, debes **intercambiar los roles**, de forma que el proceso padre será el que genere el número aleatorio y lo introducirá en la tubería, mientras que el proceso hijo lo leerá de la tubería. Por último, en lugar de generar un solo número aleatorio, se deberán generar **cinco números aleatorios** y **cada uno de ellos irá en un mensaje independiente**.

La salida esperada para este ejercicio es la siguiente:

```
i02gupep@NEWTS:~/pas/1415/p3$ ./ejercicio3
[PADRE]: mi PID es 10530 y el PID de mi hijo es 10531
[PADRE]: escribimos el número aleatorio 41 en la tubería...
[PADRE]: escribimos el número aleatorio 2634 en la tubería...
[PADRE]: escribimos el número aleatorio 2247 en la tubería...
[PADRE]: escribimos el número aleatorio 4118 en la tubería...
[PADRE]: escribimos el número aleatorio 4014 en la tubería...
[PADRE]: tubería cerrada.
[HIJO]: mi PID es 10531 y mi PPID es 10530
[HIJO]: leemos el número aleatorio 41 de la tubería
[HIJO]: leemos el número aleatorio 2634 de la tubería
[HIJO]: leemos el número aleatorio 2247 de la tubería
[HIJO]: leemos el número aleatorio 4118 de la tubería
[HIJO]: leemos el número aleatorio 4014 de la tubería
[HIJO]: tubería cerrada. Salgo...
```

A partir de la salida generada (que en cada ejecución será distinta), debes explicar que está sucediendo con la tubería (mensajes que hay dentro de la misma, en qué momento entran y cuándo salen). Prueba a poner un `sleep(5);` tras escribir cada mensaje por parte del padre. Vuelve a ejecutar el programa y explica de nuevo la salida generada. Todas estas explicaciones las puedes incluir en un fichero de texto plano que llames `ejercicio3.txt`.

14. Ejercicio resumen 4

Los ficheros de código utilizados en este ejercicio serán `ejercicio4-servidor.c`, `ejercicio4-cliente.c` y `common.h`. Los ejecutables generados tendrán como nombre `ejercicio4-servidor` y `ejercicio4-cliente`. Debes modificar el código de la sección 12.6 del siguiente modo:

1. Lo primero que se pide es modificar ambos programas para que el servidor compruebe si los mensajes enviados por el cliente emparejan o no una determinada expresión regular. 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. 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, incluidas en `regex.h` [37] [38] [39]. Todo esto forma parte del estándar POSIX. La expresión regular a buscar se indicará por línea comandos, utilizando la opción `-r/--regex`. Mediante la bandera `-e/--ere` especificaremos si la expresión regular es o no de tipo `ERE`. Por defecto, usaremos `BRE`. Sino especificamos ninguna expresión regular, se utilizará `'[A-Z]'`. También debemos incluir la opción de ayuda `-h/--help`, tal y como se hizo en el `ejercicio2`. A continuación, se muestran ejemplos de invocación:

```
# Buscar 'Casa'
$ ./ejercicio4-servidor --regex 'Casa'
...
# Buscar una C o una D, una a y luego la C o D que
# había al principio, usando EREs
$ ./ejercicio4-servidor --regex '([CD])a\1' -e
...
# Buscar una letra en mayúscula
$ ./ejercicio4-servidor
...
# Ayuda del programa
$ ./ejercicio4-servidor -h
Uso del programa: ejercicio4-servidor [opciones]
Opciones:
-h, --help          Imprimir esta ayuda
```

[37] Tienes un ejemplo de uso de `regex.h` en http://www.peope.net/old/regex.html

[38] Información general sobre `regex.h` en http://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html

[39] Documentación completa de `regex.h` en http://pubs.opengroup.org/stage7tc1/functions/regexec.html

```
-r, --regex=EXPR  Expresión regular a utilizar
-e, --ere         Utilizar expresiones regulares de tipo ERE
```

Para hacer el ejercicio, deberás crear una cola de mensajes adicional que contendrá los mensajes de tipo "Empareja" o "No Empareja" enviados desde el servidor al cliente. Esta cola la creará y eliminará el 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.

2. 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.

3. En el código que se ha puesto en Moodle, tanto el cliente como el servidor tienen incluidas unas funciones de *log*. Estas funciones implementan un pequeño sistema de registro o *log* (aunque no demasiado óptimo). Utilizándolas se registran los mensajes que los programas van mostrando por pantalla en ficheros de texto (`log-servidor.txt` y `log-cliente.txt`). Por ejemplo, si queremos registrar en el cliente un mensaje simple, haríamos la siguiente llamada:

```
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:

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

Utiliza estas llamadas para dejar registro en fichero de texto de todos los errores que se imprimen por consola y de los mensajes más importantes (por ejemplo, el envío y recepción de mensajes a través de las colas).

4. Captura las señales `SIGTERM`, `SIGINT` y `SIGHUP` para gestionar adecuadamente el fin del programa servidor y del programa cliente. Puedes asociar estas tres 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*. Tanto cliente como servidor, antes de salir, deberán mandar a la cola correspondiente, un mensaje de fin de sesión, que hará que el otro extremo deje de esperar mensajes. A este mensaje le podéis asociar una prioridad más alta. Por último, se deberá cerrar, en caso de que estuvieran abiertas, aquellas colas que se estén utilizando y el fichero de *log*.

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

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

[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.