UNIVERSIDAD DE CÓRDOBA
ESCUELA POLITÉCNICA SUPERIOR
DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

ASIGNATURA ***SISTEMAS OPERATIVOS***

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 1

Procesos y señales

Juan Carlos Fernández Caballero

jfcaballero@uco.es

Índice de contenidos

1 Objetivo de la práctica

La presente práctica persigue instruir al alumnado con la creación y gestión de procesos en sistemas que siguen el estándar POSIX[1] (*Portable Operating Systems Interface*), así como con el tratamiento y uso de señales.

En una primera parte de este documento aparece una introducción teórica sobre procesos y señales, siendo en la segunda parte del mismo cuando se practican mediante programación en C los conceptos aprendidos, utilizando las rutinas que proporcionan a los programadores la biblioteca *glibc*[2], la cual implementa en el estándar POSIX.

2 Recomendaciones

El lector debe completar las nociones dadas en las siguientes secciones con consultas bibliográficas, tanto en la Web como en la biblioteca de la Universidad, ya que uno de los objetivos de las prácticas es potenciar la capacidad autodidacta y de análisis de un problema.

Es recomendable también que, además de los ejercicios prácticos que se proponen, pruebe y modifique otros que encuentre en la Web (se dispone de una gran cantidad de problemas resueltos en C sobre esta temática), ya que al final de curso deberá acometer un examen práctico en ordenador como parte de la evaluación de la asignatura.

Al igual que se le instruyó en las asignaturas de Metodología de la Programación, es recomendable que siga unas normas y estilo de programación claro y consistente. No olvide tampoco comentar los aspectos más importantes de sus programas y añadir información de cabecera a sus funciones como práctica habitual de un programador (nombre, parámetros de entrada, parámetros de salida, objetivo, etc).

3 Conceptos teóricos

3.1 El estándar POSIX

UNIX[3] con todas sus variantes ha sido probablemente el sistema operativo con más éxito. Aunque sus conceptos básicos ya tienen más de 50 años, siguen siendo la base para muchos sistemas operativos modernos, como por ejemplo GNU/LINUX y sus variantes, y sistemas basados en BSD (*Berkeley Software Distribution),* como Mac OS o FreeBSD[4].

En un principio, por conflictos entre distintos vendedores, muchas de las variantes de UNIX tenían su propia API (A*pplication Programming Interface*) o conjunto de llamadas o funciones para poder programar el sistema, por lo que se producían muchos problemas de portabilidad de software. Era una suerte si un programa escrito para un sistema funcionaba también en el sistema de otro vendedor.

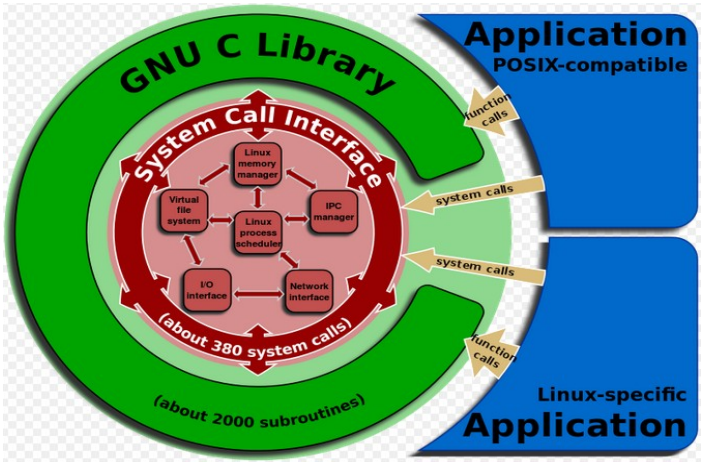
Afortunadamente, después de varios intentos de estandarización se introdujeron los estándares POSIX. POSIX es un conjunto de estándares que definen un conjunto de servicios e interfaces con que una aplicación programada mediante algún lenguaje de programación puede contar en un sistema operativo que implemente dicho estándar. Estos estándares persiguen generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas que las implementen.

---

1 http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603
2 http://es.wikipedia.org/w/index.php?title=Glibc&oldid=53229698
3 https://es.wikipedia.org/wiki/Unix
4 BSD es un sistema operativo basado en UNIX surgido en la Universidad de California en Berkeley en los años 70.

Dentro del estándar se especifica el comportamiento de las expresiones regulares, la sintaxis y semántica de los servicios del sistema operativo, la definición de datos y notaciones de manejo de ficheros, nombrado de funciones, etc. El estándar no especifica cómo deben implementarse los servicios o llamadas al sistema a nivel de núcleo del sistema operativo (llamadas nativas), de tal forma que los "implementadores" de sistemas pueden hacer la implementación que deseen, y cada sistema que implemente el estándar tendrá las suyas propias.

Una llamada al sistema (nativa) está implementada en el núcleo del sistema operativo por parte de los diseñadores del sistema. Cuando un programa llama a una función del sistema, los argumentos son empaquetados y manejados por el núcleo, el cual toma el control de la ejecución hasta que la llamada se completa.



Para que se pueda decir que un sistema cumple el estándar POSIX, tiene que implementar al menos el conjunto de definiciones base de POSIX. Otras muchas definiciones útiles están definidas en extensiones que no tienen que implementar obligatoriamente los sistemas que se quieran basar en el estándar, aunque casi todos los sistemas modernos soportan las extensiones más importantes.

Algunas de las interfaces básicas del estándar POSIX son:

- Creación y la gestión de procesos.
- Creación y gestión de hilos.
- Señales.
- Comunicación entre procesos (IPC - *InterProcess Communication*).
- Gestión de la entrada-salida.
- Comunicación sobre redes (*sockets*).

Es necesario indicar que 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. 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.

La última versión de la especificación POSIX es del año 2017, se conoce por "P*OSIX.1-2017*", "*IEEE Std 1003.1-2017*" y por "*The Open Group Technical Standard Base Specifications, Issue 7*". Puede encontrar una completa especificación de POSIX *Online* en la siguiente *ur*l:

http://pubs.opengroup.org/onlinepubs/9699919799/

GNU C Library, comúnmente conocida como ***glibc***[5], **sigue (implementa) el estándar POSIX** para sistemas GNU/LINUX y Mac OS, y proporciona llamadas al sistema (*wrappers*) y funciones de biblioteca que son utilizadas por casi todos los programas a nivel de aplicación.

Puede encontrar una completa descripción (consultar sección de documentación) de *GNU C Library* en el siguiente sitio Web:

http://www.gnu.org/software/libc/libc.html

Consulte y utilice estas especificaciones durante todo el curso.

3.2 Procesos

Hay varias definiciones de proceso: 1) Programa en ejecución, 2) Entidad que se puede asignar y ejecutar en un procesador, 3) Unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

Todos los programas cuya ejecución solicitan los usuarios lo hacen en forma de procesos. El sistema operativo mantiene por cada proceso una serie de estructuras de información que permiten identificar las características de éste, así como los recursos que tiene asignados, es decir, su **contexto de ejecución**.

Una parte muy importante de esta información se encuentra en el llamado bloque de control del proceso (**BCP**)[6]. El sistema operativo mantiene en memoria una lista enlazada con todos los BCP de los procesos existentes en el sistema. Esta estructura de datos se llama **tabla de procesos.**

La tabla de procesos reside en memoria principal, pero solo puede ser accedida por parte del sistema operativo en **modo núcleo**, es decir, el usuario no puede acceder a los BCPs.

BCP (bloque de control de proceso)

Entre la información que contiene el BCP, cabe destacar:

- **Información de identificación. Esta información identifica al usuario y al proceso.**
  - Identificador del proceso.
  - Identificador del proceso padre.

[5] http://es.wikipedia.org/w/index.php?title=Glibc&oldid=53229698
[6] http://es.wikipedia.org/wiki/Bloque_de_control_del_proceso

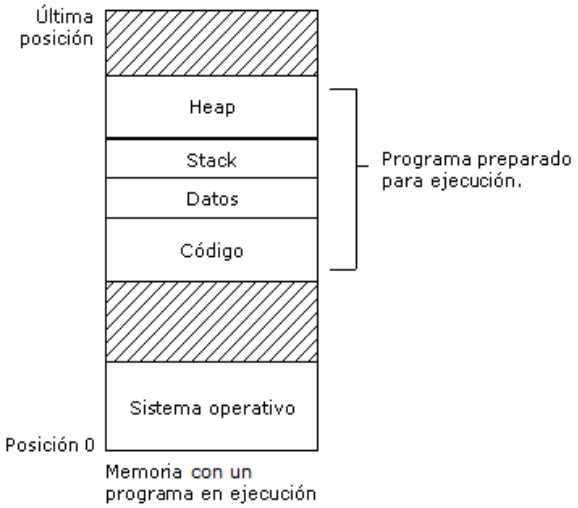
  - Información sobre el usuario (identificador de usuario e identificador de grupo).

- **Información de planificación y estado.**
  - Estado del proceso (Listo, Ejecutando, Suspendido, Parado, *Zombie*).
  - Evento por el que espera el proceso cuando está bloqueado.
  - Prioridad del proceso.
  - Información de planificación.
- **Descripción de los segmentos de memoria asignados al proceso.** Espacio de direcciones o límites de memoria asignado al proceso.
- **Punteros a memoria.** Incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos, e incluso si el proceso utiliza memoria virtual. Se almacenan también punteros a la pila y al montículo del proceso.
- **Datos de contexto.** Estos son datos que están presenten en los registros del procesador cuando el proceso está corriendo. Almacena el valor de todos los registros del procesador, contador de programa, banderas de estado, señales, etc., es decir, todo lo necesario para poder continuar la ejecución del proceso cuando el sistema operativo lo decida.
- **Recursos asignados**, tales como peticiones de E/S pendientes, dispositivos de E/S (por ejemplo, un disco duro) asignados a dicho proceso, una lista de los ficheros en uso por el mismo, puertos de comunicación asignados.
- **Comunicación entre procesos.** Puede haber varios indicadores, señales y mensajes asociados con la comunicación entre dos procesos independientes.
- **Información de auditoría.** Puede incluir la cantidad de tiempo de procesador y de tiempo de reloj utilizados, así como los límites de tiempo, registros contables, etc.

La estructura de un programa en memoria principal está compuesta por (no necesariamente en este orden):

- **Pila[7] o *stack*:** Registra por bloques llamadas a procedimientos (funciones) y los parámetros pasados a estos, variables locales de la función invocada, y la dirección de la siguiente instrucción a ejecutar cuando termine la llamada. Esta zona de memoria se asigna por el sistema operativo al cargar un proceso en memoria principal. En caso de auto llamadas recursivas podría desbordarse.
- **Montículo o *Heap*:** Zona de memoria asignada por el sistema operativo para datos en tiempo de ejecución, en sistemas POSIX se usa para la familia de llamadas *malloc()*. Puede aumentar y disminuir en tiempo de ejecución de un proceso.
- **Datos:** Variables globales, constantes, variables inicializadas y no inicializadas, variables de solo lectura.
- **Código del programa:** El código del programa en si.

[7] http://es.wikipedia.org/wiki/Pila_de_llamadas



A todo este **conjunto de elementos** o segmentos de memoria **más el BCP** de un proceso se le llama **imagen del proceso**. Para que un proceso se ejecute debe tener cargada su imagen en memoria principal.

3.3 Servicios POSIX para la gestión de procesos

A continuación se expondrán las funciones que implementa la librería *glibc* para la gestión de procesos.

3.3.1 Creación de procesos (*fork()*)

En sistemas basados en POSIX cada proceso se identifica por medio de un entero único denominado ID del proceso.

Todos los procesos en un sistema forman una jerarquía (padres-hijos-nietos-etc) con un origen común, que es el primer proceso creado durante la inicialización del sistema. Ese primer proceso se llama ***init***, **su identificador es 1**, y es el padre del resto de procesos del sistema.

Dentro de una jerarquía, si el padre de un proceso muere, otro proceso adopta a ese hijo huérfano. En sistemas basados en POSIX es el proceso *init* quien adopta a los procesos sin padre, aunque POSIX no lo exige.

La creación de un nuevo proceso en el sistema se realiza con la llamada a la rutina *fork()*[8]**,** y su prototipo es el siguiente:

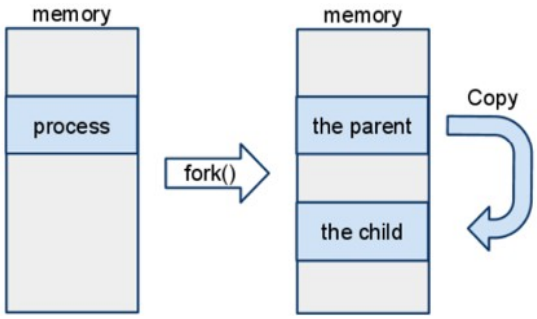
```
#include <sys/types.h>   //Varias estructuras de datos[9].
#include <unistd.h>   //API[10] de POSIX y creación de un proceso.
pid_t  fork (void);
```

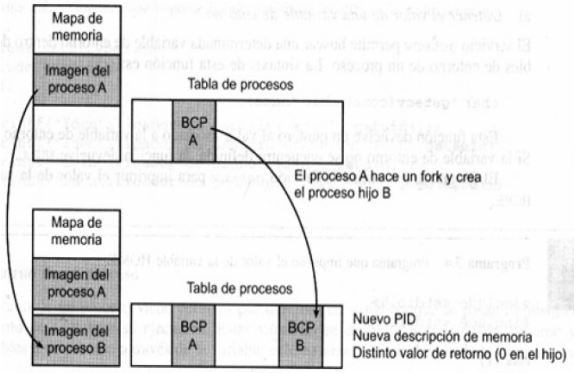
La llamada *fork()* crea un nuevo proceso hijo **idéntico al proceso padre que hace la invocación**. Eso conlleva a que tienen una copia del mismo BCP (con algunas variaciones), el mismo código fuente, los mismos archivos abiertos, la misma pila, etc; aunque padre e hijos están situados o alojados en distintos espacios o zonas de memoria. Digamos que se hereda la información del

8 http://www-h.eng.cam.ac.uk/help/tpl/unix/fork.html
9 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_types.h.html
10 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html

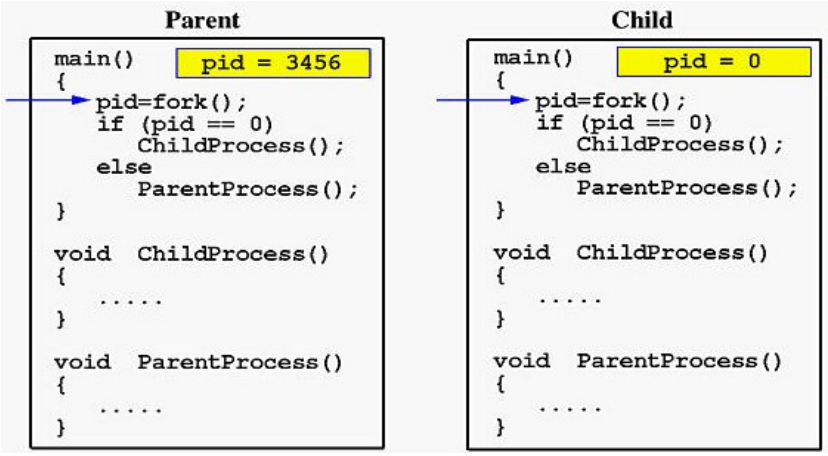
proceso padre mediante una copia, pero esa información no es compartida, sino copia.

Una vez que se crea un nuevo proceso mediante una invocación a *fork()*, surge la pregunta de por qué línea de código comienza a ejecutar el proceso hijo. No hay que caer en el error de pensar que **el proceso hijo** empieza la ejecución del código en su punto de inicio, sino que al igual que el padre, **empieza a ejecutar justo en la sentencia que hay después de la invocación a *fork()***. Por tanto, ¿qué sucede cuando un proceso padre crea a un hijo?, pues que tanto el padre como el hijo continúan la ejecución desde el punto donde se hace la llamada a *fork*().





A nivel de programación de usuario, para diferenciar al proceso que hace la llamada a *fork()* (padre) del proceso hijo, el sistema **devuelve al padre el identificador o PID del hijo creado, y al hijo devuelve un valor 0**. De esta manera se pueden distinguir los dos procesos durante el resto del código.



En caso de no poder crear una copia del proceso, la llamada a *fork()* devuelve **-1** y modifica el valor de la variable global ***errno***[11] para indicar el tipo de error[12]. Por tanto, lo que hará que se ejecute una parte u otra del código heredado o copiado, es el identificador de proceso, que se consultará mediante esquemas *if()* o *swich*().

A modo de resumen, lo que indica de manera textual la documentación de *Open Group* respecto al valor devuelto por *fork()* es lo siguiente:

"*Upon successful completion, fork() shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the fork() function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and errno shall be set to indicate the error*."

[11] http://es.wikipedia.org/wiki/Errno.h
[12] http://pubs.opengroup.org/onlinepubs/009604599/basedefs/errno.h.html

Para estudiar cómo utilizar códigos de error de los sistemas basados en POSIX, consulte el capítulo 2 del manual de *glibc*, además de la información dispuesta en *Open Group*[13],[14]. El buen uso de códigos de error se evaluará de manera positiva en el examen práctico de la asignatura.

Las **diferencias** más importantes con respecto a **BCP entre padre e hijo** están en:

- El proceso hijo tiene su propio identificador de proceso, distinto al del padre.
- El valor de retorno del sistema operativo como resultado del *fork()* es distinto. El hijo recibe un 0, el padre recibe el identificador de proceso del hijo.
- El proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga los mismos segmentos con el mismo contenido, es decir, la misma copia de código, no está en la misma zona de memoria.
- El tiempo de ejecución del hijo se pondrá a cero para estadísticas que se necesiten.
- **Las alarmas pendientes que tuviera el padre se desactivan en el hijo.** Las alarmas son señales que se activan por los temporizadores y núcleo del sistema, para indicar al proceso algún tipo de tiempo de espera o programación temporal para determinadas tareas.
- **Las señales[15] pendientes** que tuviera el padre **se desactivan en el hijo**.
- El hijo puede acceder a los **descriptores de ficheros** del padre de los que tenga copia.

Las modificaciones que realice el proceso padre sobre declaraciones de variables y estructuras de datos después de la llamada a *fork()*, no afectan al hijo y viceversa (distinción importante). Sin embargo, el hijo tiene una copia de los descriptores de fichero (punteros a fichero) que tuviera abiertos el padre, por lo que sí podría acceder a ellos y modificarlos.

A continuación, en el fichero "**demo1.c**" se muestran un ejemplo del uso de *fork()*. Este código crea como proceso hijo una copia del proceso actual y posteriormente lo espera mediante una llamada a la función *wait()*, la cual se estudiará más detalladamente. Dado que los dos procesos comparten el mismo código, es necesario comprobar el valor devuelto por *fork()* para distinguir padre e hijo. Ambos procesos continúan con la ejecución del mismo código después de la llamada a *fork()*, pero cada uno de los procesos tiene otro valor para la variable *hijo_pid*. Para demostrar que los procesos son realmente diferentes, los procesos utilizan la llamada *getpid()* para imprimir su identificador. Se puede utilizar también la llamada *getppid()* para obtener el identificador del padre de un proceso. Ambas funciones también se estudian a continuación.

Compile y ejecute el fichero. Trate de comprender y estudiar la salida reflejada. Infórmese para qué se utilizan las bibliotecas .h incluidas en la gestión de procesos.

[13] http://pubs.opengroup.org/onlinepubs/9699919799/functions/errno.html
[14] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/errno.h.html
[15] http://es.wikipedia.org/wiki/Se%C3%B1al_%28inform%C3%A1tica%29

3.3.2 Identificación de procesos (*getppid()* y *getpid()*)

Para determinar la identificación de un proceso padre y de un proceso hijo se pueden utilizar las funciones *getppid()* y *getpid()* respectivamente:

```
#include <sys/types.h> //Consulte en  IEEE Std 1003.1-2017 online
#include <unistd.h> //Consulte en  IEEE Std 1003.1-2017 online
pid_t   getppid (void) //Consulte en  IEEE Std 1003.1-2017 online
pid_t   getpid (void) //Consulte en  IEEE Std 1003.1-2017 online
```

En "**demo2.c**" tiene un esquema de espera de hijos más completo que en "**demo1.c**" y el tratamiento con un esquema *switch()* del valor devuelto por fork(), en vez de con un esquema *if()*. ¿Quién es el padre del proceso padre?.

3.3.3 Suspensión y espera de un proceso (*wait()*)

Un proceso padre debe esperar hasta que su proceso hijo termine. Para ello debe ejecutar una llamada a *wait()* o *waitpit()*[16], quedando el padre en esa invocación en estado suspendido. Por tanto, la llamada al sistema *wait()* detiene al proceso que llama hasta que un hijo de éste termine o se detenga.

```
#include <sys/wait.h>
pid_t wait (int *stat_loc);
```

Si *wait()* regresa debido a la terminación o detención de un hijo, el valor devuelto es positivo y es igual al ID de proceso de dicho hijo. Si la llamada no tiene éxito o no hay hijos que esperar *wait()* devuelve -1 y pone un valor en *errno.*

Un hijo puede acabar antes de que el padre invoque a *wait()*, pero aun así el valor de estado del hijo queda almacenado y puede ser recogido con *wait()* posteriormente.

El parámetro **stat_loc* es un puntero a entero modificado con un valor que indica el estado del proceso hijo al momento de concluir su actividad. Si quien hace la llamada pasa un valor distinto a NULL, *wait()* guarda el estado devuelto por el hijo.

Un hijo regresa su estado llamando a *exit(), _exit()* o *return()*. Concretamente regresará el parámetro entero que pasemos a dichas invocaciones de salida, cuyo valor debe tener una interpretación para el programador.

POSIX establece las siguientes macros que se utilizan por pares para saber sobre los estados de un hijo a esperar. Ese estado se almacena en **stat_loc.*

- ***WIFEXITED*(*stat_loc*)** y ***WEXITSTATUS*(*stat_loc*).**
- ***WIFSIGNALED*(*stat_loc*)** y ***WTERMSIG*(*stat_loc*).**
- ***WIFSTOPPED*(*stat_loc*)** y ***WSTOPSIG*(*stat_loc*)**.

Estas macros devuelven un valor que puede ser 0 o distinto de cero, en este último caso, si por ejemplo *WIFEXITED*(*stat_loc*) devuelve distinto de cero (cierto en C), significa que el hijo que se esperó terminó con normalidad y podemos usar *WEXITSTUS*(*stat_loc*) para imprimir su estado.

[16] http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html

Consulte en la web las llamadas *wait()*, y el uso de las macros comentadas con ejemplos de su uso[17],[18].

La función *waitpid()* es similar a *wait()*, pero se puede usar también para grupos de procesos o para esperar a un proceso concreto:

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int * stat_loc, int options);
```

Se recomienda utilizarla para la espera de hijos igual que *wait()*, pero ofrece más opciones. Toma tres parámetros: un PID con el identificador de un proceso específico a esperar, el puntero a la variable donde se almacenará el estado del hijo y una zona de banderas para especificar opciones.

- Si *pid* es -1, *waitpid()* espera a cualquier hijo, por lo que estaría haciendo lo mismo que si usamos *wait()*.
- Si *pid* es mayor que 0, *waitpid()* espera al hijo especificado cuyo proceso de ID es *pid*.

Si en el parámetro *options* establecemos:

- *WNOHANG*, hace que *waitpid()* chequee si algún hijo ha terminado, de forma que si ninguno lo ha hecho devuelve un 0 y se continua por la siguiente sentencia de código. Es una especie de **llamada no bloqueante**.
- El valor *WUNTRACED* hace que *waitpid()* informe del estado de los procesos hijos que son detenidos y que por lo tanto no regresarían su estado a *wait()* hasta que no se reanudasen.
- *WCONTINUED* indica si un proceso hijo ha sido reanudado.

Ya sabe que dos procesos vinculados por una llamada *fork* (padre e hijo) poseen zonas de datos propias, de uso privado, no compartidas. Obviamente, al tratarse de procesos diferentes, cada uno posee un espacio de direccionamiento independiente e inviolable. En la ejecución del programa "**demo3.c**", donde se hace uso de *waitpid()*, se asigna distinto valor a una misma variable según se trate de la ejecución del proceso padre o del hijo, permitirá comprobar tal característica de la llamada *fork*. El proceso padre visualiza los sucesivos valores impares que toma su variable *i* privada, mientras el proceso hijo visualiza los sucesivos valores pares que toma su variable *i* privada y diferente a la del proceso padre.

En el fichero "**while-wait-waitpid.c**" tiene un resumen del esquema de espera de hijos por parte de un proceso padre usando *wait()* o *waitpid()*.

3.3.4 Ejecutar un proceso (*exec()*)

Una llamada al sistema *fork()* crea una copia del proceso que la invoca. La familia de llamadas al sistema *exec()* proporciona una característica que permite reemplazar el código de un proceso en ejecución por el código del programa que se pasa como parámetro. Se puede considerar que el servicio *exec()* tiene dos fases:

1) en la primera se vacía el proceso en ejecución de casi todo su contenido y,

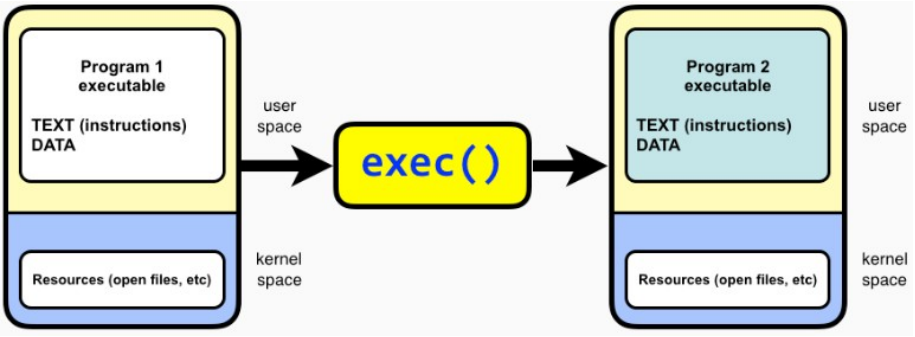
2) en la segunda se carga con el programa pasado como parámetro.

La invocación de *exec()* no significa crear un nuevo hijo con el programa pasado como parámetro para, a continuación, seguir por la siguiente línea de código, cosa que si ocurre con la llamada

[17] http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html
[18] https://support.sas.com/documentation/onlinedoc/sasc/doc750/html/lr2/zid-9832.htm

system()[19]. Consulte la Web y estudie sus diferencias.

Las llamadas *fork()* y *exec()* se suelen utilizar de manera conjunta. La manera usual de utilizar la combinación ***fork()-exec()*** es dejar que el proceso hijo creado con *fork()* ejecute una llamada *exec()* para un nuevo programa, mientras que el padre continua con la ejecución del código original.



Las seis variaciones existentes de la llamada *exec()* se distinguen por la forma en que son pasados los argumentos por la línea de comandos y el entorno de ejecución, y por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable:

- Las llamadas ***exec1 (exec1, exec1p)*** pasan los argumentos de la línea de comandos como una lista y es útil si se conoce el número de argumentos en tiempo de compilación.
- Las llamadas ***execv (execv, execvp)*** pasan los argumentos de la línea de comandos en un array de argumentos.

Si cualquiera de las llamadas *exec()* se ejecutan con éxito no se devuelve nada, en caso contrario se devuelve -1, actualizando la macro *errno* con el tipo error producido.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);
int execlp(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);  //Puntero a array de cadenas
int execvp(const char *file, char *const argv[]);
```

El parámetro *path* de *execl* es la ruta de acceso y el nombre del programa, especificado ya sea como un nombre con la ruta completa o relativa al directorio de trabajo. Después aparecen los argumentos de la línea de comandos, seguido de un puntero a NULL.

Cuando se utiliza el parámetro *file*, éste es el nombre del ejecutable y se considera implícitamente el PATH que haya en la variable de entorno del sistema.

Cuando utilice un array de cadenas *char *const argv[]* como argumento (normalmente recogido de la línea de argumentos), asegúrese de que el último elemento del array sea cero o NULL. Si lo recoge de la línea de argumentos ya está establecido por defecto.

Consulte el resto de prototipos en la Web, en la especificación de la IEEE[20] y en los ejemplos de los que dispone en la plataforma Moodle.

El ejemplo "**demo4.c**" llama al comando "ls" usando como argumento la opción "-l". El ejemplo

[19] http://pubs.opengroup.org/onlinepubs/9699919799/functions/system.html
[20] http://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

"**demo5.c**" ejecuta el mandato recibido en la línea de argumentos. Pruébelos y estúdielos. Hay muchos más ejemplos en la Web[21], consulte cuanto sea necesario sobre estas funciones.

3.3.5 Terminación de un proceso (*exit()*, *return()*)

Un proceso puede finalizar de manera normal o anormal. Se termina de manera normal en una de las tres siguientes situaciones:

- Ejecutando la sentencia *return()* dentro de una función o finalizando ésta normalmente si devuelve *void* en su prototipo.
- Ejecutando la llamada *exit()*.

Cuando un proceso finaliza se liberan todos los recursos asignados y retenidos por el mismo, como por ejemplo archivos que hubiera abiertos y sus correspondientes descriptores de ficheros, y se pueden producir varias cosas:

a) Si el padre se encuentra ejecutando un *wait()* se le notifica en respuesta a esa llamada.

b) Si el proceso hijo finalizara antes de que el padre recibiera esta llamada, el proceso hijo se convertiría de manera momentánea en un **proceso en estado *zombie*** (se conserva todavía su estado de finalización *status*), y hasta que no se ejecute la llamada *wait()* en el padre (o *waitpid()*), el proceso no se eliminará totalmente. **Si un proceso padre termina y no ejecuta la llamada a *wait()* en su código, los procesos hijos que tuviera quedarían en estado *zombie*.**

Para evitar la acumulación de procesos, los sistemas POSIX prevén un límite de procesos *zombie*, de forma que el proceso *init* se encarga de recoger su estado de finalización y liberarlos.

El prototipo de *exit()* se exponen a continuación. Infórmese de manera más específica en el *IEEE Std 1003.1-2017*.[22]

```
#include <stdlib.h>
void exit(int status);
```

exit() toma un parámetro entero, *status*, que indica el estado de terminación del programa o proceso (entero que tendrá un determinado significado para el programador). Para una terminación normal se hace uso del valor cero en *status*, los valores distintos de 0 (se suele poner 1 o -1) significan un tipo determinado de error. Lo más recomendable es usar las macros EXIT_SUCCESS, EXIT_FAILURE, ya que son más portables y asignan 0 y 1 respectivamente.

Como es requerido por la norma ISO C, usar ***return(0)* en un *main()* tiene el mismo comportamiento que llamar *exit(0)* o EXIT_SUCCESS**. Si debe tener cuidado si usa *exit()* en una determinada subrutina si lo que quería es devolver algún tipo de dato y continuar, ya que ***exit()* termina el proceso desde el cual se llama**. Haga uso de la documentación del estándar POSIX en linea y consulte la web.

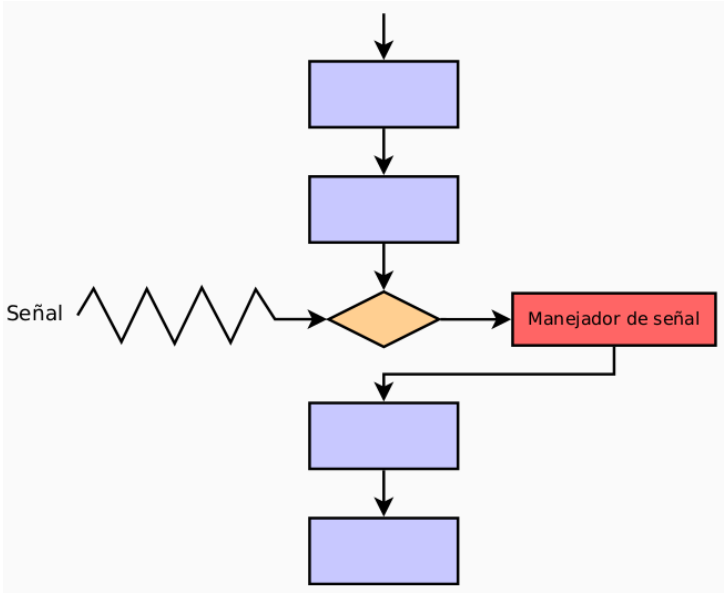
El programa "**demo6.c**" crea un proceso hijo, el proceso hijo escribe su ID en pantalla, espera 5 segundos y sale con un *exit (33)*. El proceso padre espera 1 segundo, escribe su ID, el de su hijo y espera que el hijo termine. Escribe en pantalla el valor de *exit()* del hijo. Estúdielos y ejecútelos en su computadora.

OJO, ESTOS *SLEEPS()* SON A MODO DIVULGATIVO. NO USAR *SLEEPS()* PARA SINCRONIZAR SUS PROCESOS.

[21] http://www.thegeekstuff.com/2012/03/c-process-control-functions/
[22] http://pubs.opengroup.org/onlinepubs/9699919799/functions/exit.html

3.4 Servicios POSIX para la gestión de señales

Una señal es un "aviso" que puede enviar un proceso a otro proceso. El sistema operativo se encarga de que el proceso que recibe la señal la trate inmediatamente. De hecho, termina la línea de código que esté ejecutando y salta a la función de tratamiento de señales adecuada. Este tipo de funciones se llaman técnicamente ***callbacks***, ***retrollamadas*** o **manejadores de señales**. Cuando se termina de ejecutar el ***callback***, se continua con la ejecución en la línea de código donde lo había dejado.



Las señales son sucesos asíncronos, no se sabe cuándo se van a producir, y afectan al comportamiento de un proceso. Algunas causas de la producción de una señal son:

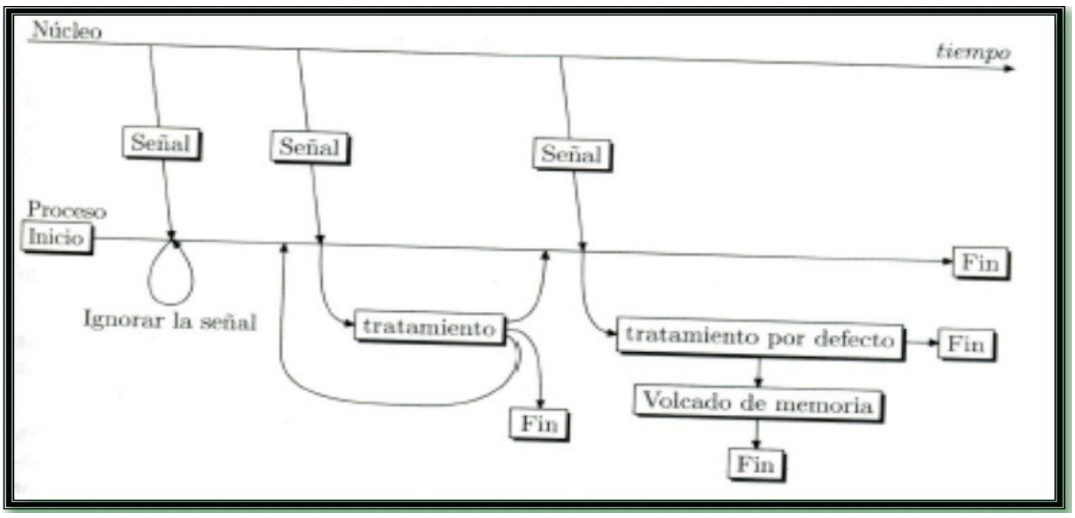
- Si en una *shell* o consola se está ejecutando un programa y pulsamos *Ctrl+C*, lo que se hace es enviar una señal de terminación al proceso en ejecución, el cual la trata inmediatamente y sale.
- Si nuestro programa intenta acceder a una zona de memoria no válida (por ejemplo, accediendo al contenido de un puntero a **NULL** o a una zona no reservada previamente), el sistema operativo detecta esta circunstancia y le envía una señal de terminación inmediata.
- Las puede mandar un proceso. Por ejemplo, desde un *shell* de comandos GNU/LINUX podemos enviar señales a otros procesos con el comando *kill*. Su uso más conocido es *kill -9 idProceso*, que envía una señal 9 al proceso *idProceso*, haciendo que éste termine.
- También es posible enviar señales desde un programa en C a otro programa en C, no solo desde la consola, sino a través de llamadas a funciones como se verá más adelante.

Las consecuencias de la recepción de una señal pueden ser también variadas:

- Pueden no tener ninguna consecuencia.
- Pueden parar la ejecución de un proceso.
- Pueden hacer reanudar la ejecución de un proceso parado.
- Pueden matar al proceso (acabar con su ejecución).
- Pueden hacer que se ejecute una función del programa previamente definida por el programador, denominada comúnmente **manejador de la señal**. Como ya se ha comentado,

un manejador es una función que un proceso llama cuando se captura una determinada señal.

Las consecuencias anteriores por parte de un proceso, y adelantándonos a las siguientes secciones, se pueden agrupar en que las señales se pueden **ignorar**, tratar de manera **específica** o tratarlas por **defecto**. El comportamiento por defecto de una señal involucra dos aspectos: 1) la acción que provoca el envío de la señal al proceso y 2) la consecuencia de la recepción de la señal.



A continuación se muestran las señales más usuales en forma de macro, puede consultarlas en *Open Group*. Para el uso de señales es necesario que incluya el fichero de cabecera *signal.h*[23]:

- ***SIGINT*:** Se envía a un proceso cuando se pulsa *Ctrl+C*, teniendo como consecuencia la terminación del proceso.
- ***SIGFPE*:** Se envía a un proceso cuando se produce un error en coma flotante, por ejemplo, una división por cero, teniendo como consecuencia la terminación del proceso.
- ***SIGTERM* y *SIGKILL***: Se mandan a un proceso cuando se necesita que acabe. Como consecuencia en ambos casos, el proceso termina, pero con una diferencia, *SIGTERM* puede programarse para que una vez accionada la señal se ejecute un manejador programado por el usuario, mientras que *SIGKILL* no lo permite. Esto es muy interesante porque permite realizar las acciones necesarias para que el programa termine en condiciones seguras. Por ejemplo: borrar archivos temporales, asegurar que los datos se escriben en disco y la estructura de su contenido es consistente, terminar procesos hijo a los que se les haya delegado parte del trabajo, etc.
- ***SIGSTOP* y *SIGCONT***: La señal *SIGSTOP* sirve para que un proceso pare su ejecución y *SIGCONT* para que continúe, pero no se puede cambiar el comportamiento del proceso. Existe una versión que es la que manda el terminal si se pulsa la tecla de parada, normalmente *CTRL+Z*, esta señal es ***SIGTSTP***.
- ***SIGALRM***: Es usada para que el proceso reciba la señal una vez transcurrido un tiempo prefijado. El proceso puede establecer un manejador para esta señal.
- ***SIGUSR1* y *SIGUSR2***: Estas dos señales son para uso del programador y no las utiliza el sistema operativo. Es el programador el que decide lo que debe hacer el proceso cuando recibe alguna de estas señales, es decir, es el programador el que define qué significan.

[23] http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html

3.4.1 Captura de señales

La función C que permite redefinir por parte del programador un tratamiento de señales es *signal()*[24]. Esta función se usa para recibir (capturar) señales, y admite dos parámetros:

```
#include <signal.h>
void* signal(int sig, void (*func)(int))
```

- El primer parámetro ***sig*** es un número entero con el identificador o **macro** de la señal.
- El segundo parámetro ***func*** indica cómo se manejará la señal. Si el valor de *func* es **SIG_DFL**, se usará el manejo por defecto para esa señal. Si el valor de *func* es **SIG_IGN**, la señal será ignorada. De lo contrario se apuntará a una función **manejador de señal** que hay que implementar y que se llamará cuando la señal se active.

Respecto a lo que **devuelve** la función *signal()* es lo siguiente:

- Si se puede cumplir con la solicitud, *signal()* devolverá el nombre de *func* para la señal *sig* especificada.
- De lo contrario, se devolverá **SIG_ERR** y se almacenará un valor positivo en *errno*.

La estructura para crear y usar un manejador de señal es la siguiente. Cuando ocurra la señal, se llamará a nuestra función propia **controlador()**:

```
void controlador (int);
...
signal (SIGINT, controlador);
```

3.4.2 Enviar Señales

Un proceso puede enviar señales a otro proceso. La función para ello es ***kill()***[25]. Esta función admite dos parámetros:

```
#include <signal.h>
void kill(pid_t pid, int sig)
```

- El primer parámetro ***pid_t*** es el identificador de proceso al que queremos enviar la señal. Si ponemos un número estrictamente mayor que **0**, se enviará al proceso cuyo ID de proceso coincida con el número.
- El segundo parámetro **sig** es el número o macro de la señal que queremos enviar.

3.4.3 Alarmas

Se puede generar una alarma con la función ***alarm()***[26]**.** Una vez invocada esta función, pasándole como parámetro un tiempo en segundos, se inicia una cuenta atrás de esa duración. Una vez terminada, se envía al proceso (por parte del núcleo del sistema operativo) una señal ***SIGALRM***, que se puede capturar.

[24] http://pubs.opengroup.org/onlinepubs/009695399/functions/signal.html
[25] http://pubs.opengroup.org/onlinepubs/009695399/functions/kill.html
[26] http://pubs.opengroup.org/onlinepubs/009695399/functions/alarm.html

#include <unistd.h> ***unsigned int alarm(unsigned seconds)***

- El parámetro ***seconds*** es el número de segundos que transcurrirán antes de que se ejecute la señal. Un proceso sólo puede tener una petición de alarma pendiente. Las peticiones sucesivas de alarmas no se encolan, **cada nueva petición anula la anterior**. Si se invoca a *alarm* con el parámetro cero, si hubiera alarmas pendientes se cancelarían, *alarm(0)*.
- La función *alarm*, en el caso de que hubiera una alarma previa pendiente, devuelve el tiempo restante para que venza un ***SIGALRM***.

En el programa "**demo7.c**" tiene un ejemplo sencillo que usa una señal de alarma para hacer una serie de impresiones cada cierto tiempo. En el programa "**demo8.c**" se muestra el uso de la función *signal()* con la señal **SIGINT**, para enviar señales entre los procesos. En concreto se cambia el comportamiento de la combinación de teclas *Ctrl+C*, evitando que el programa salga con el primer intento. Para ello, se programa una función controlador que se ejecutará cuando el usuario pulse *Crtl+C*. Estudie estos dos programas y ejecútelos en su computadora.

4 Ejercicios Prácticos

A continuación se plantean una serie de ejercicios que debe implementar en C.

4.1 Sobre procesos

Los siguientes ejercicios son sobre procesos.

4.1.1 Ejercicio 1

Implemente dos programas que pidiendo un número de procesos totales *N* **por línea de argumentos** cree las siguientes jerarquías de procesos:

Programa a) Cree un abanico de procesos como el que se refleja en la primera figura.
Programa b) Lo mismo, pero recreando lo que representa la segunda figura.

Cada proceso hijo mostrará por salida estándar un mensaje incluyendo su PID y el de su padre, y finalizará su ejecución con código de salida 0 (recuerde que esto es simplemente hacer un *exit(0), return(0) o exit(EXIT_SUCCESS)*).

El padre esperará para recoger a sus hijos a su finalización e imprimirá un mensaje indicando la finalización de cada hijo y su *status*, y terminará con código 0. Utilice macros como *EXIT_FAILURE*, *WEXITSTATUS*, etc.

4.1.2 Ejercicio 2

Se dice que un proceso está en el estado de *zombie* en GNU/LINUX cuando, habiendo concluido su ejecución, está a la espera de que su padre efectúe un *wait()* para recoger su código de retorno.

Para ver un **proceso zombie**, implemente un programa que tenga un hijo que acabe inmediatamente (por ejemplo, que imprima su ID y termine). Deje dormir al padre mediante la función *sleep()* durante 20 segundos y que luego acabe usando por ejemplo *exit(EXIT_SUCCESS),* sin recoger al hijo.

Con otro terminal monitorice los procesos la orden de la "*ps -a*". Verá que en uno de los procesos se indica que el proceso hijo está *zombie* o perdido (<*defunct*>) mientras sigue ejecutándose el programa padre en la función *sleep()*. Cuando muere el padre, sin haber tomado el código de retorno del hijo mediante *wait()*, el hijo es automáticamente heredado por el proceso *init*, que se encarga de "exorcizarlo" y eliminarlo del sistema.

Otra prueba que puede hacer es que el padre, una vez superado el primer *sleep()*, espere a su hijo, y después de eso haga otro *sleep()* para posteriormente terminar. Observará que el hijo está *zombie* pero de manera momentánea, hasta que el padre ejecute la función *wait()*.

RECUERDE, LOS *SLEEPS()* SON A MODO DIVULGATIVO. NO USAR *SLEEPS()* PARA SINCRONIZAR SUS PROCESOS, ESO ES CUESTIÓN DEL NÚCLEO DEL SISTEMA OPERATIVO.

4.1.3 Ejercicio 3

Implemente un programa donde se creen dos hijos. Uno de ellos que abra la calculadora de su distribución de Linux (busque como se llama ese ejecutable) y el otro que abra un editor de textos con *N* ficheros pasados como argumentos (recuerde hacer que el padre espere a los hijos). Use para ello la familia de funciones *exec()*.

Un ejemplo de invocación sería:

```
./miPrograma gnome-calculator gedit fichero1.txt fichero2.txt ficheroN.txt
```

Implemente cada hijo en una función, tenga cuidado con el uso de punteros y argumentos.

4.1.4 Ejercicio 4

Cree un programa que reciba por la linea de argumentos un número y calcule el factorial de ese número. Compílelo y compruebe su funcionamiento. A partir de ahí cree otro programa aparte que reciba dos números enteros como parámetros en la linea de argumentos y cree dos procesos hijos, de manera que cada uno calcule el factorial de uno de los números usando el ejecutable creado anteriormente (“./a.out 3 5”), use para ello la familia de funciones *exec()*. En el programa que calcula el factorial ponga un *sleep(1)* entre los cálculos parciales para poder observar en consola como se van ejecutando los dos procesos que se lanzarán en paralelo.

RECUERDE, LOS *SLEEPS()* SON A MODO DIVULGATIVO. NO USAR *SLEEPS()* PARA SINCRONIZAR SUS PROCESOS, ESO ES CUESTIÓN DEL NÚCLEO DEL SISTEMA OPERATIVO.

4.1.5 Ejercicio 5

Use por ejemplo el ejercicio 1a) y cree una variable global de tipo entero inicializada a 0. Haga que cada hijo aumente en uno el valor de esa variable global y que el padre imprima el resultado final. ¿Qué ocurre? Correcto, su valor no se modifica porque los hijos son procesos nuevos que no comparten memoria. Para ello, y concretamente en sistemas basados en POSIX, se utilizan métodos de intercomunicación de procesos como son memoria compartida y semáforos, los cuales se estudiarán en otra práctica de la asignatura.

4.2 Sobre procesos y señales

Los siguientes ejercicios son sobre procesos y el uso de señales.

4.2.1 Ejercicio 6

Realizar un programa que capture la señal de alarma, de manera que imprima la cadena "RING" pasados 5 segundos, después pasados otros 3 segundos y por último cada segundo. Implementar esto último, utilizando un bucle infinito que vaya imprimiendo el número de timbrazos. Pasados 4 timbrazos, el proceso se debe parar utilizando para ello la función *kill()*.

4.2.2 Ejercicio 7

Realizar un programa padre que expanda un hijo y al cual le envíe cada 1 segundo una señal personalizada de usuario SIGUSR1. El hijo debe imprimir un mensaje en pantalla cada vez que recibe la señal del padre, tratándola en una función aparte llamada *tratarSennal*(). Enviados 5 mensajes los procesos deben salir. Utiliza las funciones ***signal()*** y ***kill().***

RECUERDE, LOS *SLEEPS()* SON A MODO DIVULGATIVO. NO USAR *SLEEPS()* PARA SINCRONIZAR SUS PROCESOS, ESO ES CUESTIÓN DEL NÚCLEO DEL SISTEMA OPERATIVO.

4.2.3 Ejercicio 8

Realizar un programa que este permanentemente a la espera de capturar una señal SIGUSR1 (en un bucle *while(1)* infinito por ejemplo), de forma que cuando la capture imprima su PID. Compílelo y láncelo por consola.

Creo otro programa aparte que reciba por línea de argumentos un PID de un proceso, y su cometido sea enviar una señal SIGUSR1 al proceso cuyo PID ha recibido por dicha línea de argumentos. Una vez enviada la señal SIGUSR1 esperará durante 1 segundo y enviará al mismo proceso al que envió la señal SIGUSR1 la señal de KILL. Compruebe por consola si el proceso al que ha enviado ambas señales existe ya en el sistema.

RECUERDE, LOS *SLEEPS()* SON A MODO DIVULGATIVO. NO USAR *SLEEPS()* PARA SINCRONIZAR SUS PROCESOS, ESO ES CUESTIÓN DEL NÚCLEO DEL SISTEMA OPERATIVO.