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

Profesorado:
Juan Carlos Fernández Caballero
Enrique García Salcines

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la creación y gestión de procesos en sistemas que siguen el estándar POSIX[1] (se hablará de ello en las siguientes secciones), y con el tratamiento y uso de señales.

En una primera parte se dará una introducción teórica sobre procesos y señales, siendo en la segunda parte de la misma cuando se practicarán los conceptos aprendidos mediante programación en C, utilizando las rutinas que proporcionan a los programadores el conjunto de librerías de *glibc*[2]. Este conjunto de librerías se basan en el estándar POSIX comentado.

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, aparte 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, así como añadir información de cabecera a sus funciones (nombre, parámetros de entrada, parámetros de salida, objetivo, etc). Estos son algunos de los aspectos que se también se valorarán y se tendrán en cuenta en el examen práctico de la asignatura.

3 Conceptos teóricos

3.1 El estándar POSIX

UNIX[3] con todas sus variantes es probablemente el sistema operativo con más éxito. Aunque sus conceptos básicos ya tienen más de 30 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 X o FreeBSD. BSD es un sistema operativo basado en UNIX surgido en la Universidad de California en Berkeley en los años 70.

En un principio, por conflictos entre distintos vendedores, muchas de las variantes de UNIX tenían su propia librería y conjunto de llamadas 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 (*Portable Operating Systems Interface*). POSIX es un conjunto de estándares que definen un conjunto de servicios e interfaces con que una aplicación puede contar en un sistema operativo. Estos estándares persiguen

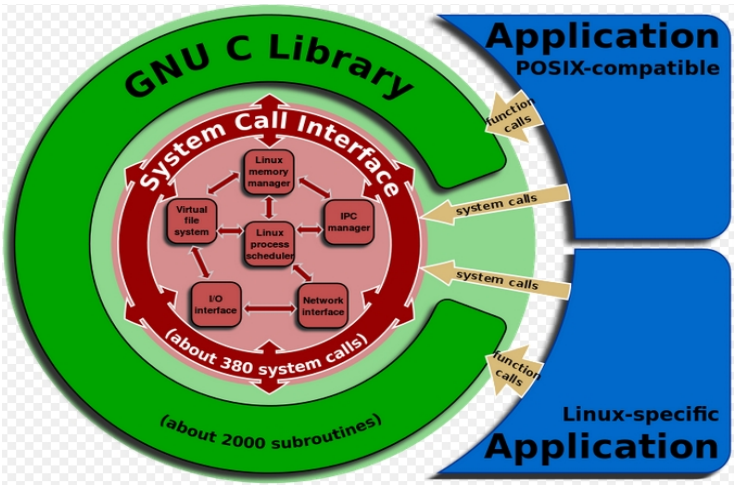
1 http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603
2 http://es.wikipedia.org/w/index.php?title=Glibc&oldid=53229698
3 https://es.wikipedia.org/wiki/Unix

generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas.

Dentro del estándar se especifica el comportamiento de las expresiones regulares, la sintaxis y semántica de los servicios del sistema operativo, de 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, de tal forma que los "implementadores" de sistemas pueden hacer la implementación que deseen, y cada sistema tendrá las suyas propias.

Todos los sistemas UNIX vienen con una librería muy potente para programar el sistema. Para los programadores de UNIX, eso incluye los programadores de GNU/LINUX y Mac OS, es fundamental aprender esa librería, porque es la base para muchas aplicaciones. GNU C Library, comúnmente conocida como *glibc*[4], sigue el estándar POSIX y proporciona e implementa llamadas al sistema y funciones de librería que son utilizadas por casi todos los programas a nivel de aplicación.

Una llamada al sistema está implementada en el núcleo de GNU/LINUX 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 por lo 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*).

4 http://es.wikipedia.org/w/index.php?title=Glibc&oldid=53229698

- Gestión de la entrada-salida.
- Comunicación sobre redes (*sockets*).

La última versión de la especificación POSIX es del año 2008, se conoce por "POSIX.1-2008", "IEEE Std 1003.1-2008" y por "The Open Group Technical Standard Base Specifications, Issue 7". Puede encontrar una completa especificación de POSIX online en la siguiente url: http://pubs.opengroup.org/onlinepubs/9699919799/. **Consulte y utilice esta especificación durante todo el curso.**

Con respecto a la librería GNU C, puede encontrar una completa descripción en http://www.gnu.org/software/libc/libc.html. **Consulte y utilice esta especificación durante todo el curso.**

Como nota, decir que Microsoft Windows no está implementado siguiendo el estándar POSIX, sino bajo un conjunto de librerías llamadas WIN32[5].

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 estas informaciones 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.

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.
  - 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

5 http://es.wikipedia.org/wiki/API_de_Windows
6 http://es.wikipedia.org/wiki/Bloque_de_control_del_proceso

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.

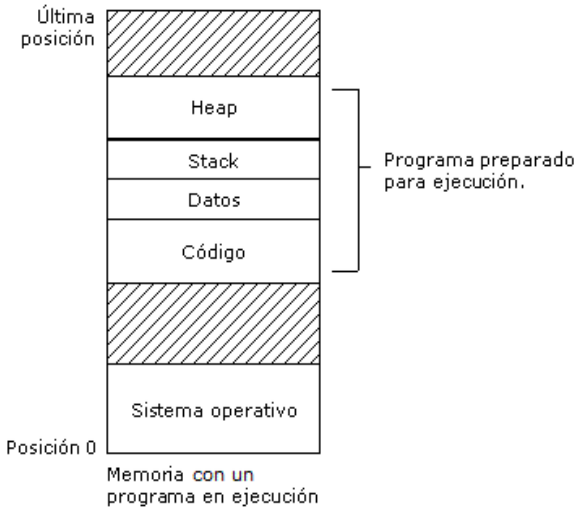
Figura: BCP

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 y los parámetros pasados a estos, variables locales de la rutina 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 UNIX se usa para la familia de llamadas *malloc()*. Puede aumentar y disminuir en tiempo de ejecución de un proceso.

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

- **Datos:** Variables globales, constantes, variables inicializadas y no inicializadas, variables de solo lectura.
- **Código del programa:** El código del programa en si.



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.

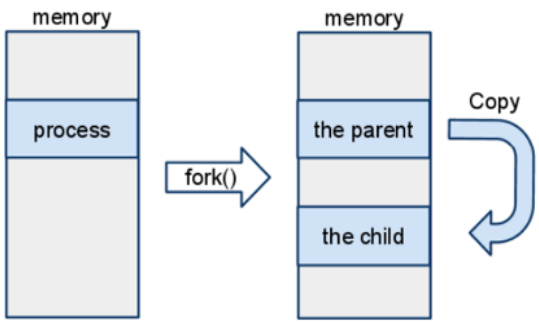
Todos los procesos en un sistema forman una jerarquía (padres-hijos-nietos-etc) con un origen común, que es el primer proceso creado durante la inicialización del sistema. Ese primer proceso se llama *init*, su identificador es 1, y es el padre del resto de procesos del sistema. Dentro de una jerarquía,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

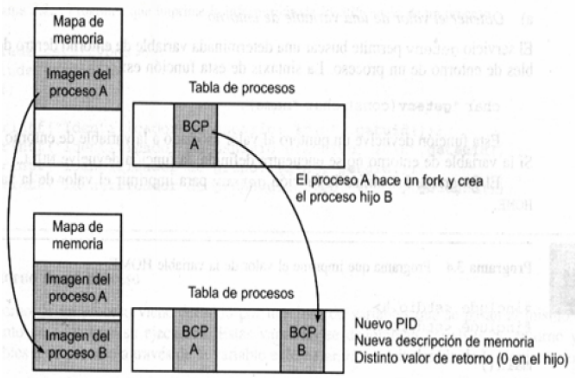
La creación de un nuevo proceso en el sistema se realiza con la llamada a la rutina *fork()*[8], y su prototipo es el siguiente:

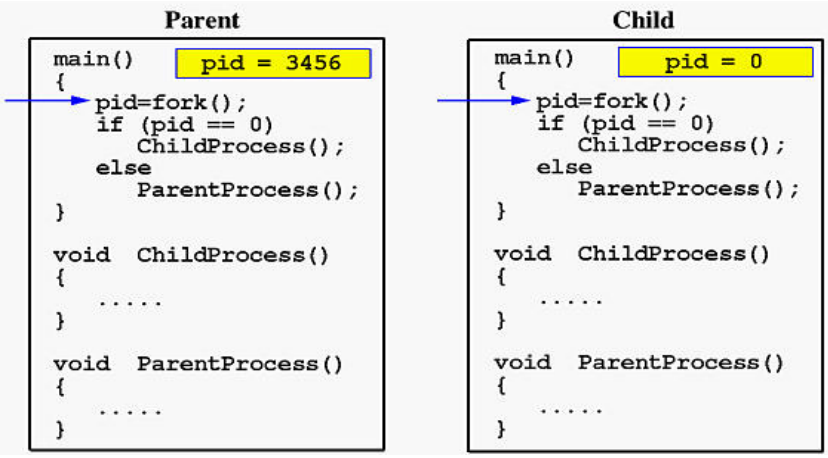
```
#include <sys/types.h>   //Varias estructuras de datos[9].
#include <unistd.h>   //API[10] de POSIX y creación de un proceso.
  pid_t  fork (void);
```

La llamada *fork()* crea un nuevo proceso hijo idéntico al proceso padre que hace la invocación. Eso conlleva a que tienen el 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 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()*.







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

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 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*().

Para estudiar como 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 tiene porque estar 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.

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 si tiene una copia de los descriptores de fichero (punteros a fichero) que tuviera abiertos el padre, por lo que si podría acceder a ellos.

A continuación se muestran dos ejemplos similares del uso de *fork()*. Estos códigos crean una copia del proceso actual. 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.

Compile y ejecute los ficheros "**demo1.c**" y "**demo2.c**". Trate de comprender y estudiar la salida reflejada. Infórmese para qué se utilizan las librerías .h incluidas en la gestión de procesos.

11 http://es.wikipedia.org/wiki/Errno.h
12 http://pubs.opengroup.org/onlinepubs/009604599/basedefs/errno.h.html
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 son necesarias las funciones *getppid()* y *getpid()* respectivamente:

```
#include <sys/types.h> //Consulte en  IEEE Std 1003.1-2008 online
#include <unistd.h> //Consulte en  IEEE Std 1003.1-2008 online
pid_t   getppid (void) //Consulte en  IEEE Std 1003.1-2008 online
pid_t   getpid (void) //Consulte en  IEEE Std 1003.1-2008 online
```

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. La ejecución del siguiente programa, en el cual 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*. En "**demo3.c**", 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.

3.3.3 Identificación de usuario (*getuid()* y *geteuid()*)

UNIX asocia cada proceso ejecutado con un usuario particular, conocido como propietario del proceso. Cada usuario tiene un identificador único que se conoce como ID del usuario. Un proceso puede determinar el ID de usuario de su propietario con una llamada a la función *getuid()*. El proceso también tiene un ID de usuario efectivo, que son permisos otorgados al usuario (*wxr*) sobre un fichero, independientes de los permisos que tenga la clase de usuario grupo y la clase otros. El ID de usuario efectivo puede consultar durante la ejecución de un proceso llamando a la función *geteuid()*.

El identificador del usuario real del proceso es el identificador del usuario que ha lanzado el proceso. El identificador del usuario efectivo es el identificador que utiliza el sistema para los controles de acceso a archivos para determinar el propietario de los archivos recién creados y los permisos para enviar señales a otros procesos. Consulte estas funciones en la Web y en el *IEEE Std 1003.1-2008 online.*

```
#include <sys/types.h>
#include <unistd.h>
pid_t   getuid (void)
pid_t   geteuid (void)
```

3.3.4 El entorno de ejecución (*getenv()*)

El entorno de un proceso viene definido por una lista de variables que se pasan al mismo en el momento de comenzar su ejecución. Este conjunto de variables definen y caracterizan las condiciones por defecto bajo las que se ejecuta ese proceso, de forma que el usuario no tenga que establecerlas una a una. Las variables de entorno se utilizan durante toda la sesión de trabajo de un usuario, ya que son heredadas por todos los procesos hijos del sistema.

Por lo general, en sistemas basados en UNIX las variables de entorno se escriben en mayúsculas. No hay nada que impida que vayan en minúsculas, salvo evitar confusiones con comandos. Las referencias a variables de entorno se realizan anteponiendo el signo "**$**" al nombre de la variable ($HOME, $PS1, etc.).

Una variable de entorno se define de la forma ***NOMBRE=valor,*** y para visualizar su valor desde una *shell* o consola basta con teclear ***echo $NOMBRE.***

Algunas variables usuales son:

- ERRNO: Variable que almacena el valor del último error producido.
- HOME : Variable que almacena el directorio del usuario, desde el que arrancará la shell cuando entra en el sistema.
- PATH : Variable en la que se encuentran almacenados los *paths* de aquellos directorios a los que el usuario tiene acceso directo, pudiendo ejecutar comandos o programas ubicados en ellos sin necesidad de acceder a dicho directorio explícitamente. Los diferentes directorios incluidos en la variable irán separados por dos puntos ":".
- PWD : Variable que almacena el directorio actual.
- PPID : Variable que almacena el PID del proceso padre. El PID del proceso actual se almacena en la variable $$.

El entorno de un proceso es accesible desde la variable externa *environ,* definida por *extern char ** environ*. Esta variable apunta a la lista de variables de entorno de un proceso cuando éste comienza su ejecución. Si el proceso se ha iniciado a partir de una llamada *exec()* (se estudiará en la siguiente sección), hereda el entorno del proceso que hizo la llamada. Consulte esta función en la Web y en el IEEE Std 1003.1-2008 online.

```
#include <stdlib.h>
char *getenv(const char *name);
```

Para modificar el entorno de un proceso podemos utilizar la función *setenv()*. Consulte su prototipo y busque algunos ejemplos de su uso en la Web.

```
#include <stdlib.h>
int setenv(const char *envname, const char *envval, int overwrite);
```

El programa "**demo4.c**" muestra el entorno del proceso actual. El programa "**demo5.c**" escribe el valor de la variable de entorno HOME. Pruébelos y estúdielos.

3.3.5 Ejecutar un programa (*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 linea de código, cosa que si ocurre con la llamada *system()*[16]. Consulte la Web y estudie sus diferencias.

Las llamadas *fork()* y *exec()* se suelen utilizar a menudo de manera conjunta. La manera tradicional 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* y *exec1e*) pasan los argumentos de la linea de comandos como un lista y es útil si se conoce el numero de argumentos en tiempo de compilación.
- Las llamadas *execv* (*execv*, *execvp*, *execve*) pasan los argumentos de la linea 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 execle(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/, char *const envp[]);


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

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 linea 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 linea de argumentos), asegúrese de que el último elemento del array sea cero o NULL. Si lo recoge de la linea de argumentos ya está establecido por defecto.

16 http://pubs.opengroup.org/onlinepubs/9699919799/functions/system.html

Consulte el resto de prototipos en la Web, en la especificación de la IEEE[17] y en los ejemplos de los que dispone en la plataforma Moodle.

El ejemplo "**demo6.c**" llama al comando "ls" usando como argumento la opción "-l". El ejemplo "**demo7.c**" ejecuta el mandato recibido en la linea de argumentos. Pruébelos y estúdielos. Hay muchos más ejemplos en la Web[18], consulte cuanto sea necesario sobre estas funciones.

3.3.6 Suspensión 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()*[19], 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 WEXITSTUS(*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 ultimo 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. Consulte en la web las llamadas *wait()*, y el uso de las macros comentadas con ejemplos de su uso[20],[21].

17 http://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html
18 http://www.thegeekstuff.com/2012/03/c-process-control-functions/
19 http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html
20 http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html
21 https://support.sas.com/documentation/onlinedoc/sasc/doc750/html/lr2/zid-9832.htm

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

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

Se recomienda utilizarla igual que *wait()*, ya que ofrece incluso más opciones. Toma tres parámetros: un PID con el identificador de un proceso específico a esperar, el puntero a la variable de estado y una bandera 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 *pid* es 0, *waitpid()* espera a cualquier hijo en el mismo grupo de procesos del usuario que llama.
- Por último, si pid es menor que -1, *waitpid* espera a cualquier hijo dentro del grupo de proceso especificado por el valor absoluto de 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.

Haga todos los ejercicios "**demo.c**" del guión que tengan llamadas a *wait()* con llamadas a *waitpid()*.

3.3.7 Terminación de un proceso (*exit()*, *_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()*.
- Ejecutando la llamada *_exit()*. Igual que la anterior pero no vacía los *buffers* de flujo (entrada-salida) que haya en el proceso que la llama (*exit()* si los limpia).

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, 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, UNIX prevé un límite de procesos *zombie,* de forma que el proceso *init* se encarga de recoger su estado de finalización y liberarlos.

Los prototipos de *exit()* y *_exit()* se exponen a continuación. Infórmese de manera más específica en el IEEE Std 1003.1-2008.[22]

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

Tanto *exit()* como *_exit()* toman 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 "**demo8.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 un 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.

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. Cuando termina de ejecutar esa función de tratamiento de señales, 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 UNIX 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.

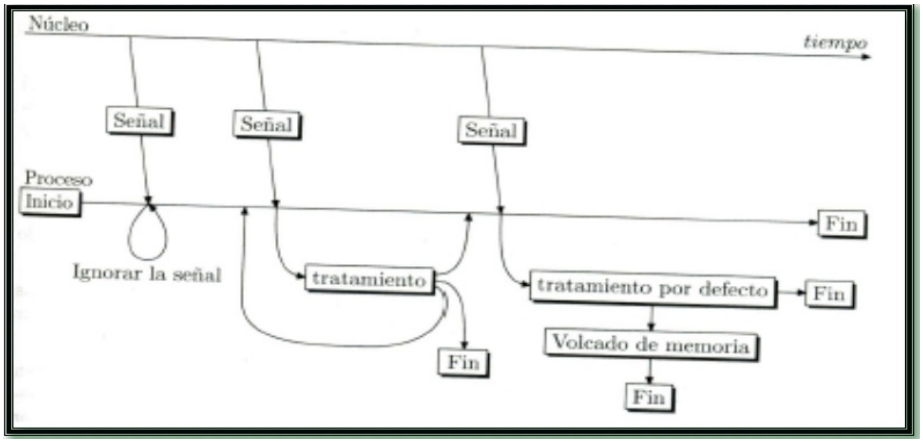
22 http://pubs.opengroup.org/onlinepubs/9699919799/functions/exit.html
http://pubs.opengroup.org/onlinepubs/9699919799/functions/_Exit.html
http://pubs.opengroup.org/onlinepubs/9699919799/functions/atexit.html

- También es posible enviar señales desde un programa en C a otro programa en C, no solo desde la consola.

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**. Una manejador es una función que un proceso llama cuando se produce una determinada señal.

Por parte de un proceso, y adelantándonos a las siguientes secciones, las señales se pueden ignorar, tratar de manera específica o tratarlas por defecto. El comportamiento por defecto de una señal, tiene que ver con 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, por ejemplo, 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,

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

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 estas señales.

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. La estructura para crear y usar un manejador de señal es la siguiente:

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

Cuando ocurra la señal, se llamará a nuestra función propia **controlador()**.

24 http://pubs.opengroup.org/onlinepubs/009695399/functions/signal.html

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 una señal **SIGALRM**, que se puede capturar.

```
#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**.

De esta forma podemos, por ejemplo, mantener en pantalla actualizado un reloj o mostrar una información durante un tiempo determinado y luego borrarla.

Para control más fino, tenemos la función **setitimer()**[27]**,** algo más compleja, pero que permite definir tiempos de micro-segundos. Esta función permite además que se nos avise repetidamente a intervalos de tiempo fijos.

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval *restrict value, struct itimerval
*restrict ovalue);
```

25 http://pubs.opengroup.org/onlinepubs/009695399/functions/kill.html
26 http://pubs.opengroup.org/onlinepubs/009695399/functions/alarm.html
27 http://pubs.opengroup.org/onlinepubs/009695399/functions/setitimer.html

- El parámetro ***value*** y ***ovalue*** son punteros a estructuras que están definidas en el fichero de cabecera *sys/time.h*. En ***value*** se establece el tiempo de alarma. Si ***ovalue*** no es un puntero a NULL, almacenar el valor previo del temporizador en la estructura apuntada por ***ovalue***. Consulte la estructura *itimerval*[28].

  Dentro de la estructura *itimerval,* si *it_value* es distinto de cero, se deberá indicar la hora de la próxima expiración del temporizador. Si *it_interval* es distinto de cero, especifica el valor para ser utilizado en la recarga de *it_value* cuando el tiempo se agota.

  Si *it_value* es 0 se desactivará un temporizador, independientemente del valor de *it_interval*.

  Establecer *it_interval* a 0 desactivará un temporizador después de su próxima expiración (suponiendo *it_value* es distinto de cero).

- El parámetro ***which*** permite indicar tres tipos de intervalos de alarma:
  - **ITIMER_REAL**. Disminuye en tiempo real. Una señal **SIGALRM** se entrega cuando este tiempo se agota.
  - **ITIMER_VIRTUAL**. Disminuye en tiempo virtual, es decir, sólo cuando el proceso se está ejecutando. Una señal **SIGVTALRM** se entrega cuando el tiempo se agota.
  - **ITIMER_PROF.** Disminuye tanto en tiempo virtual como en real. Está diseñado para ser utilizado por intérpretes en perfiles estadísticos. Cada vez que el temporizador ITIMER_PROF se agota, se entrega la señal **SIGPROF**.

En el programa "**demo9.c**" tiene un ejemplo sencillo que implementa un *sleep()* con la señal de alarma. En el programa "**demo10.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.

28 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_time.h.html

4 Ejercicios Prácticos

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

4.1 Procesos

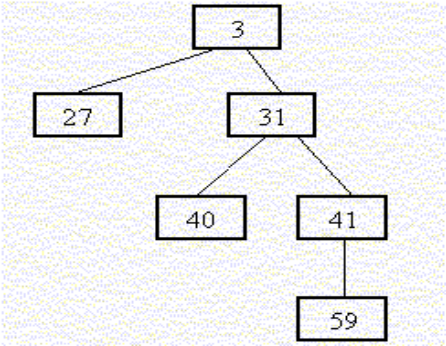
1) Cree dos programas en C (pida el número de procesos totales N por la entrada estándar del sistema):
a) Cree un abanico de procesos como el que se refleja en la primera figura.
b) Lo mismo pero recreando lo que representa la segunda figura.

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)* o *return(0)*).

El padre esperará por la finalización de ellos 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 (esto se valora en el examen final).

2) Se dice que un proceso está en el estado de *zombie* en UNIX 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)*.
Ejecute el programa en segundo plano (usando *&*) y monitorice varias veces en otra terminal los procesos con la orden de la *shell* "*ps -a*". Verá que en uno de ellos se indica que el proceso hijo está *zombie* o perdido 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.

3) Generar una serie de procesos cuyas relaciones familiares sigan el esquema siguiente:



Cuando nace un proceso, crea los hijos que le corresponda crear, duerme cinco segundos para descansar del esfuerzo procreador, espera por la muerte y va sumando los códigos de retorno de sus hijos. Al resultado de la suma le suma, a su vez, la última cifra de su PID. Al acabar, el proceso devuelve un código de retorno igual a la suma previamente calculada. De este modo, el padre de todos los procesos conocerá la suma de las últimas cifras del PID de todos sus descendientes, incluido él mismo. Tomando como ejemplo los PIDs de la figura anterior, la salida será al estilo de la siguiente:

Soy el primer hijo (pid=27) y mi suma es: 7
Soy el primer nieto (pid=40) y mi suma es: 0
Soy el bisnieto (pid=59) y mi suma es: 9
Soy el segundo nieto (pid=41) y mi suma es: 10 (9 +1)
Soy el segundo hijo (pid=31) y mi suma es: 11 (10 + 0 + 1)
Soy el padre (pid=3) y mi suma es: 21 (11 + 7 + 3)

4) Implemente un programa donde se creen dos hijos. Uno de ellos que abra la calculadora de Linux en gnome (gnome-calculator) y el otro que abra un editor de textos con N ficheros pasados como argumentos (recuerde hacer que el padre espere a los hijos). La 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).

5) Cuando un proceso padre crea a un hijo mediante fork(), los descriptores de ficheros que haya en el padre también los "hereda" el hijo. Implemente un programa en el que el padre y el hijo (o si lo prefiere un padre y dos hijos) hagan varias escrituras en un fichero de texto, intercalando un *sleep(1)* entre escritura y escritura. Puede hacer que por ejemplo el padre escriba un tipo de caracteres (+++++) y el hijo (hijos) otros distintos (--------). Al termino de la escritura (el padre debe esperar al hijo) cierre el fichero y visualícelo para ver si se ha creado correctamente. Use la linea de argumentos para proporcionar el nombre de fichero a su programa.

6) Use por ejemplo el ejercicio 1 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 UNIX y que siguen el estándar POSIX se utilizan métodos de comunicación entre procesos como son: Memoria compartida y semáforos.

4.2 Procesos y señales

7) 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()*.

8) Realizar un programa padre que expanda un hijo y 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()***.

9) Realizar 2 programas que sean finalmente dos procesos. El segundo programa tomará el PID del primer programa como argumento del ejecutable y enviará la señal definida por usuario número 1 cada 3 segundos un total de 8 veces. El primer programa escribirá el mensaje "He recibido la señal del otro programa" por cada señal recibida. El segundo programa enviará la señal de terminación no abrupta (**SIGINT**) al programa 1 antes de acabar su ejecución para indicarle que debe salirse.