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 memoria compartida

Profesorado:

Enrique García Salcines

Javier Pérez Rodríguez

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], como es el caso de los actuales sistemas GNU/LINUX, además del 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 de interfaz del sistema que proporcionan a los programadores el conjunto de librerías de *glibc*[2], las cuales se basan 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 unos de los objetivos de las prácticas es potenciar su capacidad autodidacta y su capacidad de análisis de un problema. Es recomendable que, aparte de los ejercicios prácticos que se proponen, pruebe y modifique otros que se encuentren 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

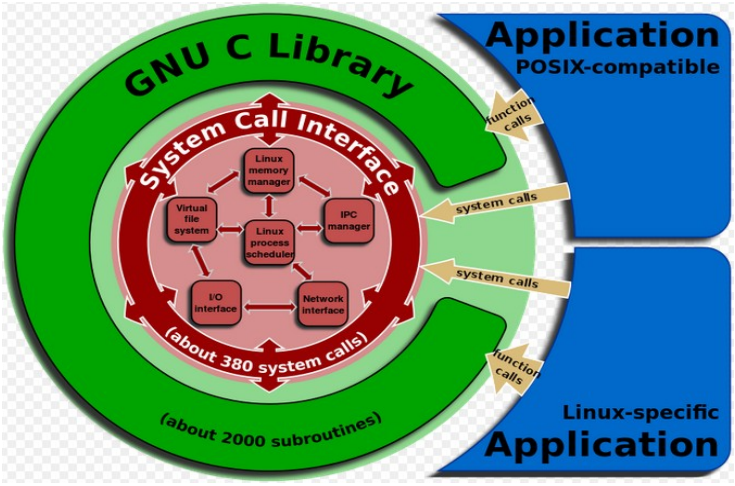
UNIX[3] con todas sus variantes es probablemente el sistema operativo con más éxito. Aunque sus conceptos básicos ya tienen más de 30 años, siguen siendo la base para muchos sistemas operativos modernos, como por ejemplo GNU/LINUX y sus variantes y sistemas basados en BSD (*Berkeley Software Distribution)*, 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 para poder programar el sistema y su propio conjunto de llamadas al 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*)[4]. POSIX es un conjunto de estándares que definen un conjunto de servicios e interfaces con que una aplicación puede contar en un sistema operativo.

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 http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603

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, de modo que los programas de aplicación puedan invocarlos siguiendo unas normas (en este caso particular a través del conjunto de bibliotecas *glibc*). 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 X) es fundamental aprender esa librería, porque es la base para muchas aplicaciones en estos sistemas. GNU C Library, comúnmente conocida como *glibc*[5], es la implementación del estándar para el lenguaje C (ANSI C[6]) en GNU/LINUX. *Glibc* 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 a nivel de núcleo se completa.



Para que se pueda decir que un sistema cumple el estándar POSIX, este 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*).

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

- 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[7]. Por tanto, si necesita crear programas que contengan gestión de procesos y señales, necesitará estudiar el conjunto de funciones de Windows que se encargan de ello; o al menos, tener algún tipo de software intermediario que se encargue de "traducir" unos tipos de llamadas a otros.

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**)[8]. El sistema operativo mantiene en memoria una lista enlazada con todos los BCP de los procesos existentes o cargados en memoria principal. 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.

7 http://es.wikipedia.org/wiki/API_de_Windows
8 http://es.wikipedia.org/wiki/Bloque_de_control_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.

Figura: BCP)

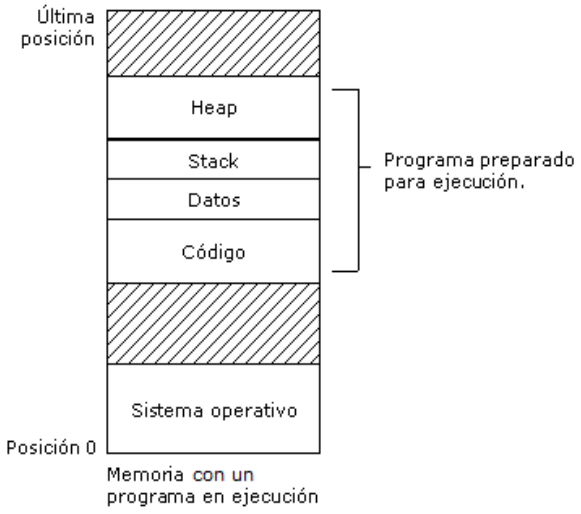
La estructura[9] de un programa en memoria principal está compuesta por (no necesariamente en este orden):

- **Pila[10] 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

9 http://latecladeescape.com/t/Code,+Stack,+Data+y+Heap+en+la+ejecuci%C3%B3n+de+programas
10 http://es.wikipedia.org/wiki/Pila_de_llamadas

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.



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 o llamadas al sistema que implementa la librería *glibc* al seguir el estándar POSIX especificado en la IEEE (*Institute of Electrical and Electronics Engineers*, Instituto de Ingenieros Eléctricos y Electrónicos) .

3.3.1 Creación de procesos (*fork()*)

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

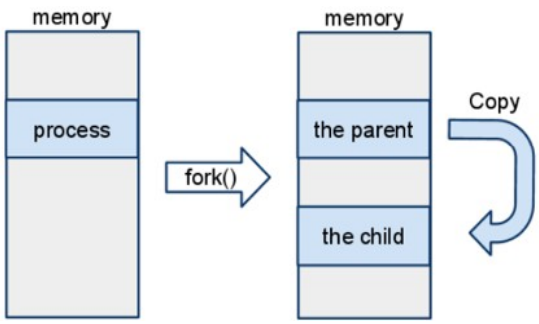
Todos los procesos en un sistema forman una jerarquía (padres-hijos-nietos-etc) con un origen común, que es el primer proceso creado durante la inicialización del sistema. Ese primer proceso se llama *init*, su identificador es 1, y es el padre del resto de procesos del sistema. Dentro de una jerarquía,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

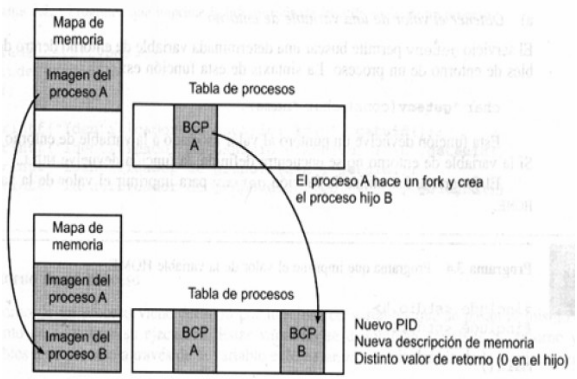
Dicho esto, la creación de un nuevo proceso en el sistema se realiza con la llamada a la rutina o función *fork()*[11]**,** y su prototipo es el siguiente:

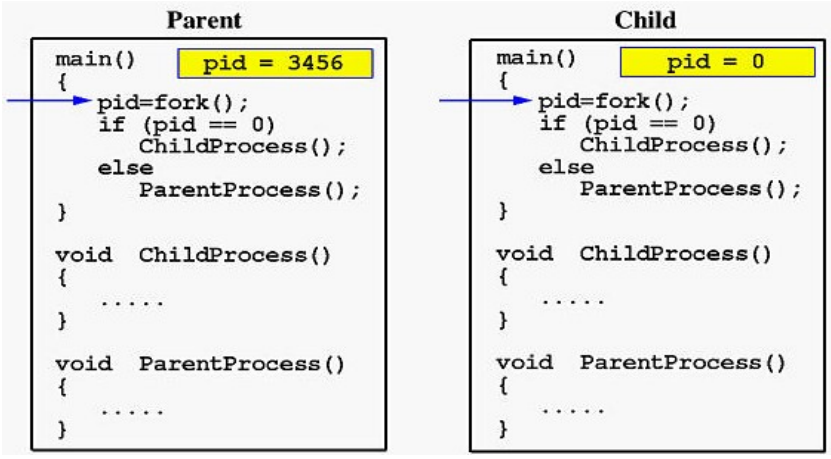
```
#include <sys/types.h>   //Varias estructuras de datos[12].
#include <unistd.h>   //API[13] de POSIX y creación de un proceso.
   pid_t  fork (void);
```

La llamada *fork()* crea un nuevo proceso hijo idéntico al proceso padre que hace la invocación, y 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, ¿que 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()*.







11 http://www-h.eng.cam.ac.uk/help/tpl/unix/fork.html
12 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_types.h.html
13 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()* 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*[14] para indicar el tipo de error[15]. 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[16],[17]. 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 proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga los mismo 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 (señal que se activa por los temporizadores y núcleo del sistema para indicar al proceso algún tipo de tiempo de espera o programación temporal para determinadas tareas) que tuviera el padre se desactivan en el hijo.
- Las señales[18] pendientes que tuviera el padre se desactivan en el hijo.
- 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.

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, ya que sus variables tienen una copia del valor de las que tenía el padre en el momento de la creación del hijo. El valor de esa copia es un puntero a fichero.

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. Consulte la Web e infórmese de aquí en adelante para qué se utilizan las librerías .h incluidas en la gestión de procesos.

14 http://es.wikipedia.org/wiki/Errno.h
15 http://pubs.opengroup.org/onlinepubs/009604599/basedefs/errno.h.html
16 http://pubs.opengroup.org/onlinepubs/9699919799/functions/errno.html
17 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/errno.h.html
18 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
```

En el programa "**demo3.c**" puede comprobar el uso de este tipo de funciones. 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, puede ser útil para modificar el prompt (PS1) dinámicamente.
- PPID : Variable que almacena el PID (identidad de proceso) 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 *fork()* al sistema crea una copia del proceso que la llama. 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, en la primera se vacía el proceso en ejecución de casi todo su contenido, y 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, al contrario que pasa con la llamada *system()*[19], que si lo hace, consulte las diferencias en la Web.

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 execle(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/, char *const envp[]);
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 execve(const char *path, char *const argv[], char *const envp[]);
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 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.

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.

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

El ejemplo “**demo6.c**” llama al comando “ls” usando como argumento la opcion “-l”. El ejemplo “**demo7.c**” ejecuta el mandato recibido en la linea de argumentos. Pruébelos y estúdielos. Hay muchos más ejemplos en la Web[21], 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()*[22], 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[23],[24],[25].

21 http://www.thegeekstuff.com/2012/03/c-process-control-functions/
22 http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html
23 http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html
24 https://support.sas.com/documentation/onlinedoc/sasc/doc750/html/lr2/zid-9832.htm
25 http://mij.oltrelinux.com/devel/unixprg/#tasks

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 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.

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.[26]

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

```
#include <unistd.h>
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**" imprime información sobre el estado de terminación de un proceso hijo. El ejemplo "**demo9.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 Memoria compartida. Conceptos generales

La forma más eficaz que tienen los procesos para comunicarse consiste en compartir una zona de memoria, tal que para enviar datos de un proceso a otro, sólo se han de escribir en dicha memoria y automáticamente estos datos estarán disponibles para cualquier otro proceso. La utilización de este espacio de memoria común evita la duplicación de datos y el lento trasvase de información entre los procesos.

La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtuales es local a ese proceso y cualquier intento de direccionar esa memoria desde otro proceso va a provocar una violación de segmento. Es decir, cuando se crea uno o más procesos mediante la llamada *fork( )*, se realiza una duplicación de todas las variables usadas, de forma que cualquier modificación de sus valores pasa inadvertida a los demás procesos, puesto que aunque el nombre es el mismo, su ubicación en memoria no lo es. Esto es debido a que con cada nuevo proceso se reserva una zona de memoria inaccesible a los demás. Las direcciones de las variables de esta zona son virtuales, y es el módulo de gestión de la memoria el que se encarga de traducirlas a direcciones físicas.

26 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

Para solucionar este problema, Linux ofrece la posibilidad de crear zonas de memoria con la característica de poder ser direccionadas por varios procesos simultáneamente. A estas zonas de memoria se les denominada segmentos de memoria compartida. Un proceso, antes de usar un segmento de memoria compartida, tiene que atribuirle una dirección virtual en su espacio de direcciones (mapearlo, sección 3.4), con el fin de que dicho proceso pueda acceder a él por medio de esta dirección. Esta operación se conoce como conexión, unión o enganche de un segmento con el proceso.

El kernel no gestiona de forma automática los segmentos de memoria compartida, por lo tanto, es necesario asociar a cada segmento o grupo de segmentos un conjunto de mecanismos de sincronización, de forma que se evite los típicos choques entre los procesos, es decir, que un proceso este modificando la información contenida en el segmento mientras que otro lee sin que se hayan actualizado todos los datos.

Claves IPC

Antes de comenzar a estudiar las llamadas al sistema para la utilización de memoria compartida es necesario explicar que son las claves IPC (InterProcess Communication). Cada objeto IPC tiene un identificador único, que se usa dentro del núcleo para identificar de forma única un objeto IPC. Una zona de memoria compartida es un objeto IPC, ya que es un objeto que sirve para la comunicación entre procesos (otros objetos IPC son los mensajes o los semáforos).

Para obtener el identificador único correspondiente a un objeto IPC se debe utilizar una *clave*. Esta clave debe ser conocida por los procesos que utilizan el objeto. La clave puede ser estática, incluyendo su código en la propia aplicación. Pero en este caso hay que tener cuidado ya que la clave podría estar en uso. Otra forma es generar la clave utilizando la función `ftok` cuyo prototipo es el siguiente:

```
key_t ftok(char *nombre, char proj);
```

`ftok` devuelve una clave basada en `nombre` y `proj` que puede ser utilizada para la obtención de un identificador único de objeto IPC. El argumento `nombre` debe ser el nombre de camino de un fichero existente. Esta función devolverá la misma clave para todos los caminos que nombren el mismo fichero, cuando se llama con el mismo valor para `proj`. Devolverá valores diferentes para la clave cuando es llamada con caminos que nombren a distintos ficheros o con valores diferentes para el parámetro `proj`.

Ejemplo:

```
key_t clave;
clave=ftok(“.”,’S’);
```

3.4.1 Petición de un segmento de memoria compartida

La función *shmget*[27] permite crear una zona de memoria compartida o habilitar el acceso a una ya creada. Su declaración es la siguiente:

```
#include <sys/shm.h>
int shmget (key, size, flag);

key_t key;      /* clave del segmento de memoria compartida */
int size;       /* tamaño del segmento de memoria compartida */
int flag;       /* opción para la creación */
```

El parámetro *clave* es la clave IPC o llave de acceso tal y como se explicó en el apartado 3.1.

El segundo parámetro, *size*, indica el espacio en bytes de la zona de memoria compartida que se desea crear. Este espacio es fijo durante su utilización.

El último parámetro *flag*, es una máscara de bits que definirá los permisos de acceso a la zona de memoria y el modo de adquirir el identificador de la misma. Puede tomar los siguientes valores:

- `IPC_CREAT`: crea un segmento si no existe ya en el núcleo.
- `IPC_EXCL`: al usarlo con `IPC_CREAT`, falla si el segmento ya existe (IPC_CREAT| IPC_EXCL)

Devuelve el identificador del segmento de memoria compartida si éxito o –1 en caso de error.

El identificador del segmento que devuelve esta función es heredado por todos los procesos descendientes del que llama a esta función.

El siguiente trozo de código muestra cómo se crea una zona de memoria de 4096 bytes, donde sólo el propietario va a tener permiso de lectura y escritura:

```
#define LLAVE (key_t)  234 /* clave de acceso */
int shmid; /* identificador del nuevo segmento de memoria compartida */
if((shmid=shmget(LLAVE, 4096, IPC_CREAT | 0600)) == -1)
{
    /* Error al crear o habilitar el segmento de memoria compartida. Tratamiento del error. */
}
```

3.4.2 Conexión a un segmento de memoria compartida

Un proceso, antes de usar un segmento de memoria compartida, tiene que atribuirle una dirección virtual en su espacio de direcciones (mapearlo), con el fin de que dicho proceso pueda acceder a él por medio de esta dirección. Esta operación se conoce como conexión, unión o

27 http://pubs.opengroup.org/onlinepubs/009695399/functions/shmget.html

enganche de un segmento con el proceso. Una vez que el proceso deja de usar el segmento, debe de realizar la operación inversa (desconexión, desunión o desenganche), dejando de estar accesible el segmento para el proceso.

La función *shmat*[28] realiza la conexión del segmento al espacio de direccionamiento del proceso.

```
#include <sys/shm.h>
char *shmat (shmid, *shmadr, shmflag);

int shmid;          /* identificador del segmento */
char *shmadr;       /* dirección de enlace */
int flag;           /* opción de conexión */
```

`shmid`: identificador del área de memoria compartida
`shmaddr`: dirección donde se mapea el área de memoria compartida. Si es NULL, el núcleo intenta encontrar una zona no mapeada.
`shmflg`: se indica el flag SHM_RDONLY si es solo para lectura.

Si la llamada a esta función funciona correctamente, devolverá un puntero a la dirección virtual a la que está conectado el segmento. Esta dirección puede coincidir o no con adr, dependiendo de la decisión que tome el kernel. El problema principal que se plantea es el de la elección de la dirección, puesto que, no puede entrar en conflicto con direcciones ya utilizadas, o que impida el aumento del tamaño de la zona de datos y el de la pila. Por ello, si se desea asegurar la portabilidad de la aplicación es recomendable dejar al kernel la elección de comienzo del segmento, para ello basta con pasarle un puntero NULL como segundo parámetro (valor entero 0). Si por algún motivo falla la llamada a esta función, devolverá –1.

Una vez que el segmento esta unido al espacio de direcciones virtuales del proceso, el acceso a él se realiza a través de punteros, como con cualquier otra memoria dinámica de datos asignada al programa.

3.4.3 Desconexión a un segmento de memoria compartida

Una vez que ya no se necesita más acceder al segmento de memoria compartida, el proceso debe realizar la desconexión o desenlace. Este desenlace no es lo mismo que la eliminación del segmento desde el núcleo. El segmento de memoria compartida se elimina sólo en el caso de que no queden procesos enlazados. Para el desenlace se utiliza el servicio `shmdt`.

La función *shmdt*[29] efectúa el desenlace del segmento previamente conectado.

```
#include <sys/shm.h>
int shmdt (*shmadr);
char *shmadr; /* dirección de conexión del segmento a desenlazar */
```

28 http://pubs.opengroup.org/onlinepubs/009695399/functions/shmat.html
29 http://pubs.opengroup.org/onlinepubs/009695399/functions/shmdt.html

Su único parámetro shmaddr, es la dirección virtual del segmento que se desea desconectar (devuelta por la llamada a la función shmat). Es lógico que después de ejecutarse esta función, dicha dirección se convierta en una dirección ilegal del proceso. Esto no quiere decir que su contenido se borre, simplemente que se deja de tener acceso a la memoria reservada.

Si la llamada se efectúa satisfactoriamente, la función devolverá el valor 0, y en caso de que se produzca algún fallo devolverá –1.

3.4.4 Control de un segmento de memoria compartida

La función *shmctl*[30] proporciona información administrativa y de control sobre el segmento de memoria compartida que se especifique. Su declaración es la siguiente:

```
#include <sys/shm.h>
int shmctl (shmid, cmd, *buf);

int shmid;              /* identificador del segmento */
int cmd;                /* operación a efectuar */
struct shnud_ds *buf;   /* información asociada al segmento de memoria compartida  */
```

Donde `shmid` es el identificador de la zona de memoria compartida, `cmd` es la operación que se quiere realizar y `buf` es una estructura donde se guarda la información asociada al segmento de memoria compartida en caso de que la operación sea `IPC_STAT` o `IPC_SET`. Las operaciones que se pueden realizar son:

IPC_STAT: guarda la información asociada al segmento de memoria compartida en la estructura apuntada por buf. Esta información es por ejemplo el tamaño del segmento, el identificador del proceso que lo ha creado, los permisos, etc.

IPC_SET: Establece los permisos del segmento de memoria a los de la estructura `buf`.

IPC_RMID: Marca el segmento para borrado. No se borra hasta que no haya ningún proceso que esté asociado a él.

La llamada a esta función devuelve el valor 0 si se ejecuta satisfactoriamente, y –1 si se ha producido algún fallo.

El siguiente trozo de código muestra como se borra un segmento de memoria compartida previamente creado:

```
int shmid; /* identificador del segmento de memoria compartida */
....
```

30 http://pubs.opengroup.org/onlinepubs/009695399/functions/shmctl.html

```
if(shmctl (shmid, IPC_RMID,NULL) == -1)
{
    /* Error al eliminar el segmento de memoria compartida. Tratamiento del error. */
}
```

Resumiendo, para trabajar con memoria compartida se utilizan básicamente las siguientes llamadas al sistema:

- **shmget:** Crea una nueva región de memoria compartida o devuelve una existente.
- **shmat:** Une lógicamente una región al espacio de direccionamiento virtual de un proceso.
- **shmdt:** Desenlaza o separa una región del espacio de direccionamiento virtual de un proceso.
- **shmctl:** Manipula varios parámetros asociados con la memoria compartida. permite tomar el control sobre la memoria compartida, lo usaremos para borrarla, se debe hacer también al final del último proceso que vaya a usar la memoria compartida.

A continuación se muestra el código comentado de dos programas que utilizan memoria compartida, **demo10.c** y **demo11.c** . El primero, crea una zona de memoria para compartir con el segundo programa. Ejecute los dos programas al mismo tiempo en dos consolas distintas. Observe y analice los resultados.

4 Ejercicios Prácticos

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

4.1 Procesos

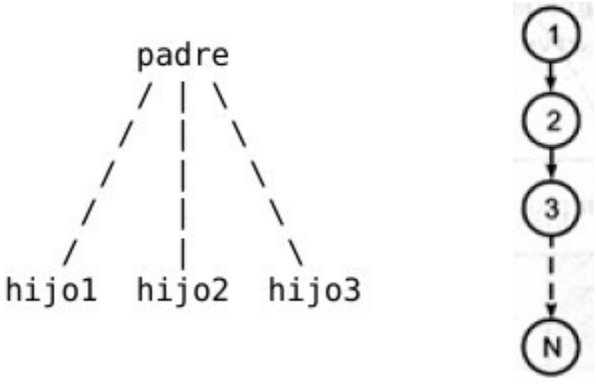
1) Cree dos programas en C (pida el número de procesos totales N por la entrada estándar del sistema):

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.

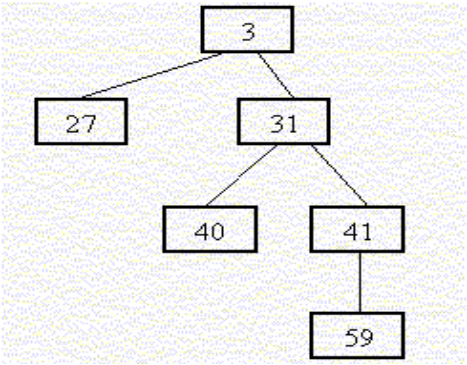
Cada proceso hijo mostrará por salida estándar un mensaje incluyendo su PID y el de su padre, y finalizará su ejecución con código de salida 0 (recuerde que esto es simplemente hacer un exit(0) o return(0)).

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 Memoria compartida

7) Como se comentó en el apartado de introducción, los procesos a diferencia de los hilos, no comparten el mismo espacio de memoria, por lo que si queremos que accedan a las mismas variables en memoria, estos deben compartirla. Realice un programa que expanda N procesos hijos. Cada hijo debe compartir una variable denominada contador, que debe estar inicializada a cero. Esta variable debe ser incrementada por cada proceso 100 veces. Imprima la variable una vez finalicen los hilos y analice el resultado obtenido. Un resultado previsible para 3 procesos sería 300, así ha sido?

8) Modifique el ejercicio anterior utilizando memoria compartida, para establecer una estrategia de paso de testigo (token ring) entre los procesos, de manera que incrementen el contador de manera exclusiva y no se produzcan inconsistencias.