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

ASIGNATURA *SISTEMAS OPERATIVOS*

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 1

Procesos

Profesorado: Juan Carlos Fernández Caballero
Alberto Cano Rojas

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la creación y gestión de procesos en UNIX (y por tanto en GNU/LINUX al basarse en el núcleo de UNIX). En una primera parte se dará una introducción teórica sobre procesos, 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*[1], las cuales se basan en el estándar POSIX[2]. GNU/LINUX está implementado basándose en el estándar POSIX, mientras las versiones de Microsoft Windows están implementadas bajo un conjunto de librerías llamadas WIN32[3].

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 (en la medida que se requiera), modularizar sus programas en ficheros (tantos como crea conveniente para su modularidad): *main.c, funciones.c*, *cabecera.h*. Con respecto a los ficheros de cabecera, no olvide usar la inclusión condicional para evitar problemas a la hora de compilar. 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

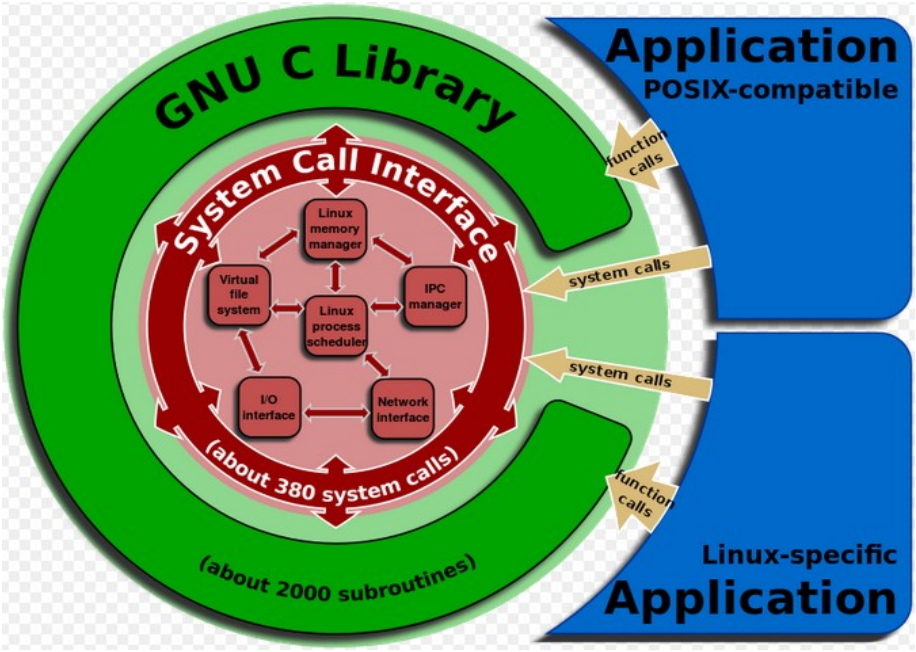
UNIX con todas sus variantes es probablemente el sistema operativo con más éxito. Aunque sus conceptos básicos ya tienen más de 30 años, siguen siendo la base para muchos sistemas operativos modernos, como por ejemplo GNU/LINUX y sus variantes y sistemas basados en BSD (*Berkeley Software Distribution),* 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

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

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. 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. Será el uso de una biblioteca u otra (basadas en POSIX) las que "traduzcan" la función que utilice el usuario a nivel de programación (la cual se basa en el estándar) a una o varias llamadas al núcleo del sistema operativo.

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

4 http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603
5 http://es.wikipedia.org/w/index.php?title=Glibc&oldid=53229698
6 http://es.wikipedia.org/wiki/ANSI_C

definidas en extensiones que no tienen que implementar obligatoriamente los sistemas, pero casi todos los sistemas modernos soportan las extensiones más importantes. Algunas de las interfaces básicas del estándar POSIX (en la asignatura trabajaremos con las dos primeras) son:

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

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.**

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 (contexto de ejecución). Una parte muy importante de estas informaciones se encuentra en el llamado bloque de control del proceso (**BCP**)[7]. 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).

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

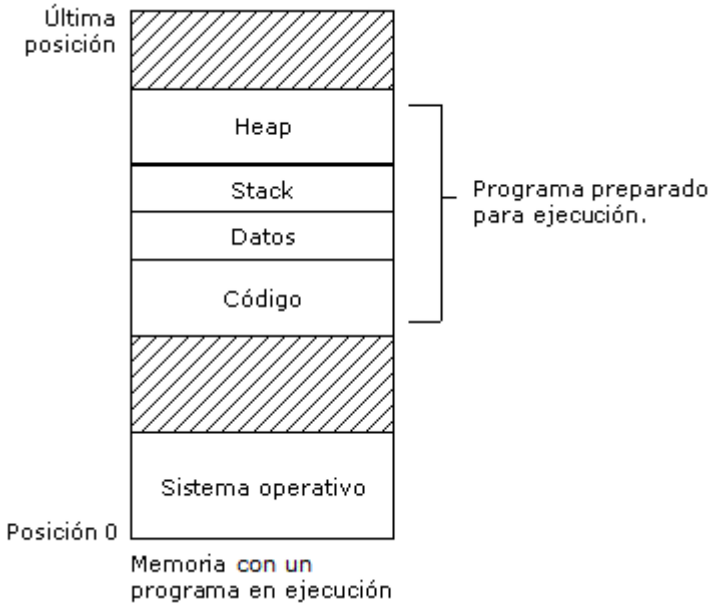
- Evento por el que espera el proceso cuando está bloqueado.
- Prioridad del proceso.
- Información de planificación.

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

Figura: Bloque de control de proceso abreviado

La estructura[8] de un programa en memoria principal está compuesta por (no necesariamente en este orden):

- **Pila[9] 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.
- **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, en bloques de memoria continuos (esto último no tiene porque ser así, dependerá del esquema de memoria que se utilice).

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) .

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

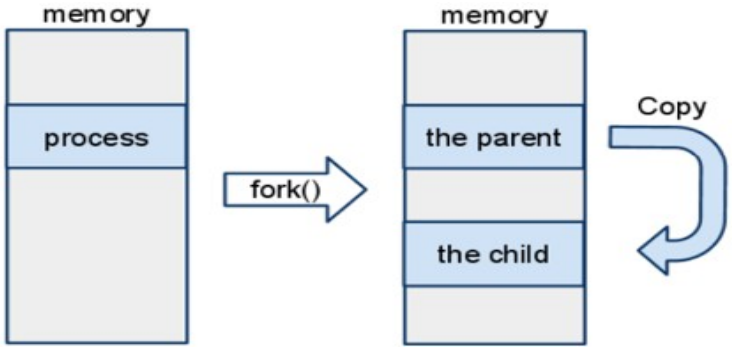
3.3.1 Creación de procesos (fork())

En UNIX el sistema identifica a cada proceso por medio de un número identificador de proceso. Para crear un proceso hay que hacer una copia de otro. El nuevo proceso se llama "proceso hijo" y el antiguo "proceso padre". Después de crear el hijo se puede substituir el programa en ejecución en el hijo por otro programa (en las siguientes secciones se comenta la llamada *exec()* para ello). Todos los procesos en el sistema han sido creados de esta manera y forman una jerarquía con un origen común: el primer proceso creado durante la inicialización del sistema (normalmente llamando ***init***, con identificador 1). Si el padre de un proceso muere otro proceso adopta a este hijo (normalmente *init* adopta a los procesos sin padre, aunque POSIX no lo exige).

La creación de un nuevo proceso se realiza con la llamada **fork()[10],** y su prototipo es el siguiente:

```
#include <sys/types.h> //Varias estructuras de datos[11].
#include <unistd.h> //API[12] de POSIX y creación de un proceso.
pid_t    fork (void);
```

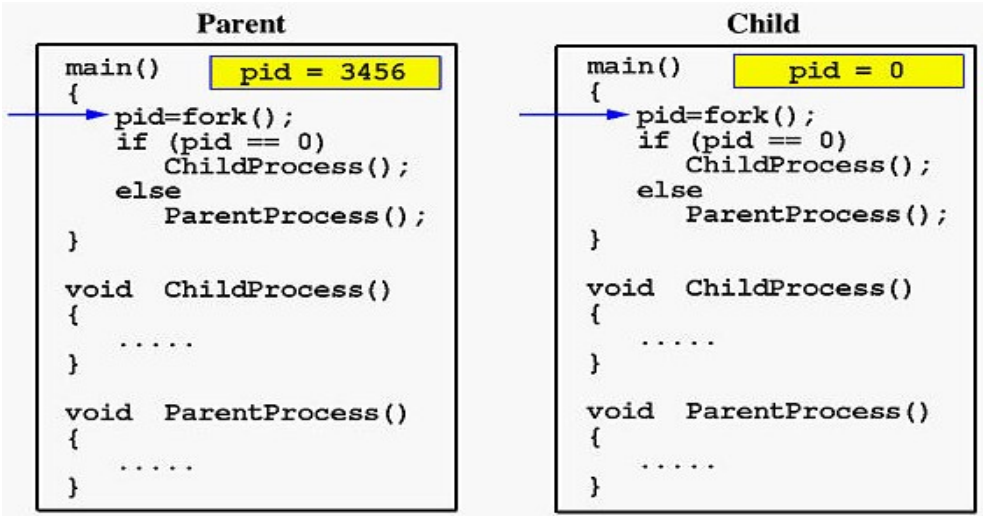
La llamada *fork()* crea un nuevo proceso hijo idéntico al proceso padre (COPIA, INSTANCIA), y eso conlleva a que tienen el mismo BCP (con algunas variaciones), el mismo código fuente, los mismos archivos abiertos, la misma pila, etc, aunque padre e hijos están situados o alojados en distintos espacios o zonas de memoria. OJO, el mismo significa COPIA, no significa COMPARTIDO.



En especial, el contador de programa de los dos procesos tiene el mismo valor, por lo que van a ejecutar la misma instrucción máquina. 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 del *fork(),* veremos algún ejemplo a continuación. Lo que hará que se ejecute una parte u otra del código heredado o copiado, es el identificador de proceso (mediante esquemas *if()* o *swich*()), que se estudia en la siguiente sección. A nivel de programación de usuario, para diferenciar al proceso que hace la llamada a *fork()*, proceso padre, del proceso creado o proceso hijo, el sistema devuelve al padre el identificador o PID del hijo creado y un valor 0 al hijo. 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

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

de la variable global *errno*[13] para indicar el tipo de error[14]. Para estudiar como utilizar códigos de error consulte el capítulo 2 del manual de *glibc*, además de la información dispuesta en Open Group[15],[16].

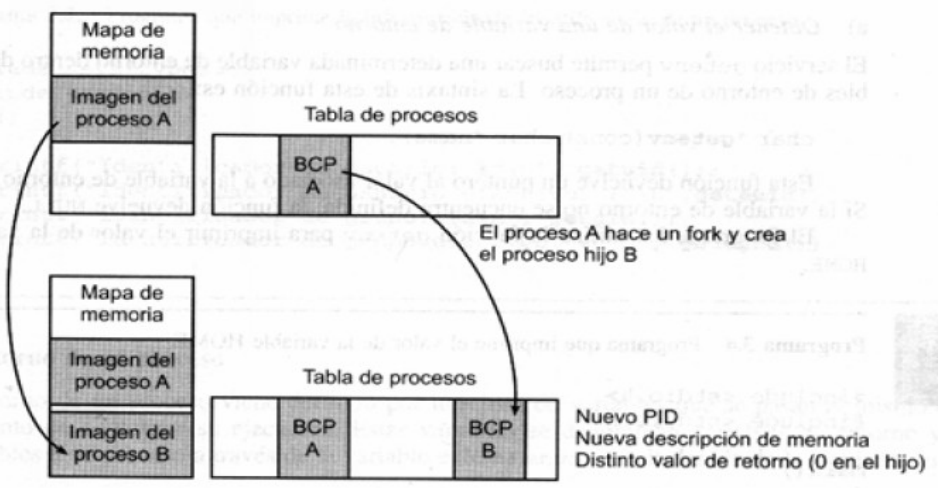


Las diferencias más importantes con respecto a BCP entre padre e hijo están en:

- 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[17] 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. 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, y sus variables tienen una copia del valor de las que tenía el padre en el momento de la creación del hijo.

13 http://es.wikipedia.org/wiki/Errno.h
14 http://pubs.opengroup.org/onlinepubs/009604599/basedefs/errno.h.html
15 http://pubs.opengroup.org/onlinepubs/9699919799/functions/errno.html
16 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/errno.h.html
17 http://es.wikipedia.org/wiki/Se%C3%B1al_%28inform%C3%A1tica%29



A continuación se muestran dos ejemplos similares del uso de *fork()*. Busque más información en la Web sobre la creación de procesos. 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.

Cree dos ficheros .c, uno para cada ejemplo que se expone a continuación, compílelos y ejecútelos. 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.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

void main(void)
{
  pid_t pid;

  pid = fork();
        switch(pid)
        {
                case -1:   /* error del fork() */
                  perror("fork error");
                  printf("errno value= %d\n", errno);
                  break;
                case 0:    /* proceso hijo */
                  printf("Proceso hijo %d; padre = %d \n", getpid(), getppid());
                  break;
                default:   /* padre */
```

```
                printf("Proceso %d; padre = %d \n", getpid(), getppid());
        }
}
```

Segundo ejemplo:

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(void)
{
 pid_t hijo_pid;
 hijo_pid = fork();

 if (hijo_pid == -1)
 {
        perror("fork error");
        printf("errno value= %d\n", errno);
        exit(-1);
 }
 else if (hijo_pid == 0) /* hijo */
 {
       printf("hijo con pid: %ld\n", (long)getpid());
       exit(EXIT_SUCCESS);
 }
 else /* padre */
 {
      printf("padre con pid: %ld\n", (long)getpid());
      exit(EXIT_SUCCESS);
 }
}
```

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

UNIX identifica los procesos mediante un entero único denominado ID del proceso. 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
```

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 el programa, 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.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main ( )
{
int i;
int j;
pid_t rf;
rf = fork( );

switch (rf)
{
   case -1:
        perror("fork error");
        printf("errno value= %d\n", errno);
        break;
  case 0:
        i = 0;
        printf ("\nSoy el hijo, mi PID es %d y mi variable i (inicialmente a %d) es par", getpid( ),
i);
        for ( j = 0; j < 5; j ++ )
        {
          i ++;
          i ++;
          printf ("\nSoy el hijo, mi variable i es %d", i);
        }
        break;
  default:
        i = 1;
        printf ("\nSoy el padre, mi PID es %d y mi variable i (inicialmente a %d) es impar", getpid(
), i);
        for ( j = 0; j < 5; j ++ )
        {
          i++;
          i++;
          printf ("\nSoy el padre, mi variable i es %d", i);
        }
```

```
}
//Esta linea la ejecuta tanto el padre como el hijo
printf ("\nFinal de ejecucion de %d \n", getpid());
exit (0);
}
```

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 un 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 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 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()*, que 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);
```

El siguiente programa muestra el entorno del proceso actual:

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

extern char **environ;

void main(int argc, char **argv) //Equivalente a (int argc, char *argv[])
{
  int i;

  printf("Lista de variables de entorno de %s\n",argv[0]);

  for (i=0 ; environ[i] != NULL ; i++)
    printf("environ[%d] = %s\n", i, environ[i]);
}
```

El siguiente programa escribe el valor de la variable de entorno HOME:

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

void main(void)
{
  char *home = NULL;
```

```
  home = getenv("HOME");
  if (home == NULL)
    printf("$HOME no se encuentra definida\n");
  else
    printf("El valor de $HOME es %s\n", home);
}
```

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);
```

3.3.5 Ejecutar un programa (exec())

La llamada *fork()* al sistema crea una copia del proceso que la llama. La familia *exec* de llamadas al sistema proporciona una característica que permite reemplazar el código del proceso actual por el código del programa que se pasa como parámetro (ojo, esto no significa crear un nuevo hijo, al contrario que pasa con la llamada *system()*[18], que si lo hace, consulte las diferencias). La manera tradicional de utilizar la combinación *fork-exec* es dejar que el hijo ejecute el *exec* para el nuevo programa mientras que el padre continua con la ejecución del código original. Se puede considerar que el servicio tiene dos fases, en la primera se vacía el proceso de casi todo su contenido, mientras que en la segunda se carga con el programa invocado.

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 (el proceso invocador termina), 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[]);
```

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

```
//Puntero a array de cadenas (la dirección de un puntero a cadena)
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 (0) 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[19] y en los ejemplos de los que dispone en la plataforma Moodle.

El siguiente código llama al comando "ls" usando como argumento la opcion "-l".

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

void main(void)
{
  pid_t pid;
  int status;

  pid = fork();
  switch(pid)
  {
    case -1:   /* error del fork() */
        perror("fork error");
        printf("errno value= %d\n", errno);
        exit(-1);  //exit(EXIT_FAILURE);
    case 0:    /* proceso hijo */
        printf("Proceso hijo %d; padre = %d \n", getpid(), getppid());
        if(execlp("ls","ls","-l",NULL)==-1)
        {
                perror("Falla en la ejecucion exec de ls -l");
                printf("errno value= %d\n", errno);
                exit(EXIT_FAILURE);
        }
    default:   /* padre */
      printf("Proceso padre\n");
      while(pid != wait(&status));
/* Recuerde que en C, 0 significa falso y cualquier numero distinto de 0 es verdadero, por tanto el bucle while() anterior no es mas que una forma de esperar a todos los hijos que tenga el proceso
```

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

```
padre hasta que no queden más. La función wait() se comenta después . Mientras hay hijos el valor de la expresión será verdadera */
   }
}
```

El siguiente programa ejecuta el mandato recibido en la linea de argumentos. Hay muchos más ejemplos en la Web[20], consulte cuanto sea necesario sobre estas funciones.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
/*Pruebe por ejemplo: "./a.out tar -cf prueba.tar file1 file2 fileN" donde file1 hasta fileN son ficheros que desea comprimir en el fichero prueba.tar*/

void main(int argc, char **argv)
{
  pid_t pid;

  pid = fork();
  switch(pid)
  {
    case -1:   /* error del fork() */
         perror("fork error");
         printf("errno value= %d\n", errno);
         break;
    case 0:    /* proceso hijo */
      if ( execvp(argv[1], &argv[1]) < 0 )
      {
        perror("exec");
        printf("errno value= %d\n", errno);
      }
      break; //exit(EXIT_FAILURE);
    default:   /* padre */
      printf("Proceso padre\n");
  }
}
```

3.3.6 Suspensión de un proceso (wait() y waitpid())

¿Qué sucede con el proceso padre después de que este crea un hijo? Tanto el padre como el hijo continúan la ejecución desde el punto donde se hace la llamada a fork(). Si un padre desea esperar hasta que el hijo termine, entonces debe ejecutar una llamada a *wait()* o *waitpit()*[21], quedando el padre suspendido.

20 http://www.thegeekstuff.com/2012/03/c-process-control-functions/
21 http://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

La llamada al sistema *wait()* detiene al proceso que llama hasta que un hijo de éste termine o se detenga. 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 porque no hay hijos que esperar o estos han muerto, *wait()* devuelve -1 y pone un valor en *errno*. Un valor *errno* igual a ECHILD indica que no existen hijos a los cuales esperar, un valor igual a EINTR significa que la llamada fue interrumpida por una señal (por ejemplo, se mata al hijo con la llamada *kill*). 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()*, de forma que no quede *zombie*.

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. El hijo regresa su estado llamando a *exit(), _exit()* o *return()*. POSIX establece las siguientes macros para saber sobre los estados de un hijo a esperar: WIFEXITED(*stat_loc*), WEXITSTUS(*stat_loc*), WIFSIGNALED(*stat_loc*), WTERMSIG(*stat_loc*), WIFSTOPPED(*stat_loc*) Y WSTOPSIG(*stat_loc*) para analizar el estado devuelto por el hijo guardado en **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, es que el hijo que se esperó terminó con normalidad. **Consulte en la web las llamadas wait(), y el uso de las macros comentadas con ejemplos de su uso**[22],[23].

La función *waitpid* es similar a *wait()* pero se usa normalmente para grupos de procesos y para casos más concretos. Toma tres parámetros: un *PID* con el identificador de un proceso específico a esperar, el puntero a la variable de estado y una bandera (especificación de opciones). Si *pid* es -1, *waitpid* espera a cualquier hijo. Si *pid* es mayor que 0, *waitpid* espera al hijo especificado cuyo proceso de ID es *pid*. Otras dos posibilidades son permitidos para el parámetro *pid*. Si *pid* es 0, *waitpid* espera a cualquier hijo en el mismo grupo de procesos de la persona 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*.

El parámetro *options,* si su valor es WNOHANG , hace que *waitpid* vuelva incluso si el estado del hijo no está disponible. El valor WUNTRACED hace que *waitpid* informe del estado de los procesos que no han sido detenidos. Use el comando *man* y la documentación online para tener una descripción completa sobre ello[24],[25].

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

22 http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html
23 https://support.sas.com/documentation/onlinedoc/sasc/doc750/html/lr2/zid-9832.htm
24 http://www.tutorialspoint.com/unix_system_calls/waitpid.htm
25 http://mij.oltrelinux.com/devel/unixprg/#tasks

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 en un proceso en estado *zombie*, y hasta que no se ejecute la llamada *wait()* en el padre, el proceso no se eliminará. Para evitar la acumulación de procesos, UNIX prevé un límite de procesos zombie y aquellos procesos hijos que se destruyen más tarde que sus procesos padres, al quedar huérfanos, el proceso *init* se encarga de recoger su estado de finalización y liberarlos.

c) Si el proceso padre no se encuentra ejecutando una llamada a *wait()* se salva el estado de terminación del proceso hijo (*status*) hasta que el proceso padre ejecute un *wait()*.

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 tendra un determinado significado para el programador). Para una terminación normal se hace uso del valor cero en *status*, 0, los valores distintos de 0 (normalmente 1 o -1) significan un tipo determinado de error (puede usar las macros EXIT_SUCCESS, EXIT_FAILURE, 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)*. Si debe tener cuidado si usa *exit()* en una determinada subrutina y 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.**

Ejemplo de programa que imprime información sobre el estado de terminación de un proceso hijo.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
```

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

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

/*Pruebe por ejemplo: "./a.out tar -cf prueba.tar file1 file2 fileN" donde file1 hasta fileN son ficheros que desea comprimir en el fichero prueba.tar
Pruébelo también sin usar nada en la linea de argumentos.
Por ultimo pruébelo usando un "file1" que no exista.
*/

void main(int argc, char **argv)
{
  pid_t pid;
  int valor;
  int prueba;

  pid = fork();
  switch(pid)
  {
    case -1:   /* error del fork() */
         perror("fork error");
         printf("errno value= %d\n", errno);
         exit(-1);
    case 0:    /* proceso hijo */
         printf("Soy el hijo y mi PID es:%d\n",getpid());
         if (execvp(argv[1], &argv[1]) < 0)
         {
           perror("exec");
           printf("errno value= %d\n", errno);
          exit(EXIT_FAILURE);
         }
    default:   /* padre */
         printf("Valor del PID recibido por el padre en el fork, coincide con el PID del hijo: %d\n",pid);
        while ( (prueba=wait(&valor)) != pid);
        printf("Valor del prueba, es el PID del hijo devuelto por wait():%d\n",prueba);

       if (valor == 0) //El hijo termina satisfactoriamente
            printf("El mandato se ejecuto de formal normal\n");
       else
       {
          if (WIFEXITED(valor)) //Terminación normal, nadie para al proceso ni se recibe una señal
                                //de otro proceso.
                    printf("El hijo termino normalmente y su valor devuelto fue %d\n", WEXITSTATUS(valor));

          if (WIFSIGNALED(valor))
              printf("El hijo termino al recibir la señal %d\n", WTERMSIG(valor));
       }
```

```
    }
}
```

Ejemplo de programa que 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.

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
main()
{
/* Identificador del proceso creado */
pid_t idProceso;

/* Variable para comprobar que se copia inicialmente en cada proceso y que luego puede cambiarse independientemente en cada uno de ellos. */
int variable = 1;

/* Estado devuelto por el hijo */
int estadoHijo;

/* Se crea el proceso hijo.*/
idProceso = fork();

/* Si fork() devuelve -1, es que hay un error y no se ha podido crear el proceso hijo. */
if (idProceso == -1)
{
        perror("fork error");
        printf("errno value= %d\n", errno);
        exit (-1);
}

/* fork() devuelve 0 al proceso hijo.*/
else if (idProceso == 0)
{
        /* El hijo escribe su pid en pantalla y el valor de variable */
        printf ("Hijo : Mi pid es %d. El pid de mi padre es %d\n", getpid(), getppid());

        /* Escribe valor de variable y la cambia */
        printf ("Hijo : variable = %d. La cambio por variable = 2\n", variable);
        variable = 2;
        /* Espera 5 segundos, saca en pantalla el valor de variable y sale */
        sleep (5);
        printf ("Hijo : variable = %d y salgo\n", variable);
        exit (33);
```

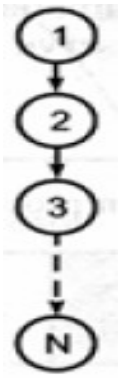
```
}
else /* fork() devuelve un número positivo al padre.*/
{
        /* Espera un segundo (para dar tiempo al hijo a hacer sus cosas y no entremezclar salida en
        la pantalla) y escribe su pid y el de su hijo */
        sleep (1);
        printf ("Padre : Mi pid es %d. El pid de mi hijo es %d\n",getpid(), idProceso);

        /* Espera que el hijo muera */
        wait(&estadoHijo);

        /* Comprueba la salida del hijo */
        if (WIFEXITED(estadoHijo) != 0)  //Si se termina bien, "estadoHijo" cogerá el valor
                                         //devuelto por éste, 33.
                printf ("Padre : Mi hijo ha salido. Devuelve %d\n", WEXITSTATUS(estadoHijo));

        /* Escribe el valor de variable, que mantiene su valor original */
        printf ("Padre : variable = %d\n", variable);
}
}
```

4 Ejercicios prácticos

A continuación se listan una serie de ejercicios para que practique en más profundidad con la gestión de procesos en C bajo el estándar POSIX.

4.1 Ejercicio 1

Cree dos programas en C: 1) Cree un abanico de procesos como el que se refleja en la primera figura, 2) lo mismo pero recreando lo que representa la segunda figura. Para ambos programas visualice en la salida el ID o PID de cada proceso y el de su padre. Pida el número de procesos totales "n" por la entrada estándar del sistema.



4.2 Ejercicio 2

Modifique levemente el primer ejercicio del aparatado anterior. Implementar un programa que cree

“n” procesos utilizando fork(): Cada proceso hijo mostrará por salida estándar un mensaje incluyendo su PID, esperará 10 * *n* segundos (donde *n* va de 1 a n, de acuerdo al orden de proceso creado), y finalizará su ejecución con código de salida 0 (recuerde que esto es simplemente hacer un exit(0) o return(0)).

El proceso padre mostrará por salida estándar un mensaje cada vez que cree exitosamente un hijo indicando el número de proceso creado (1..n), esperará por la finalización de todos ellos, e imprimirá un mensaje indicando la finalización de cada hijo y su estatus, y terminará con código 0. Utilice macros EXIT_FAILURE, WEXITSTATUS(status).

(Verificar en otra consola los procesos creados utilizando los comandos “ps -u” y “pstree”).

Deberá visualizar algo similar a la siguiente captura de pantalla:

```
  PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
 3146  0.0  0.0  25376  4036 pts/3    Ss   07:53   0:01 bash
19232  0.0  0.0  25324  3840 pts/4    Ss   15:36   0:00 bash
19301  0.0  0.0   4196   360 pts/3    S+   15:40   0:00 ./a.out 3
19303  0.0  0.0   4196    92 pts/3    S+   15:40   0:00 ./a.out 3
19304  0.0  0.0   4196    92 pts/3    S+   15:40   0:00 ./a.out 3
19307  0.0  0.0  21024  1328 pts/4    R+   15:40   0:00 ps -u
```

```
├─gnome-keyring-d───4*[{gnome-keyring-d}]
├─gnome-terminal─┬─bash───a.out───3*[a.out]
                 ├─bash───pstree
                 ├─gnome-pty-helpe
                 └─3*[{gnome-terminal}]
```

4.3 Ejercicio 3

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

4.4 Ejercicio 4

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 30 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.

4.5 Ejercicio 5

Implemente un programa en el que al ejecutarlo cree un hijo que duerma un determinado numero de segundos y termine con un valor de estado determinado. Tanto el número de segundos como el valor de estado lo pasaremos por la linea de argumentos, por ejemplo: ./miPrograma 30 2.

Haga que el padre espere al hijo, y cuando este termine de manera normal imprima su estado. Haga también comprobaciones para ver si el hijo muere por alguna señal. Para este último caso, desde otra terminal examine el PID del hijo (comando *ps*) y ejecute el comando *kill* para "matarlo". Al hacer eso, el padre (a través de la función *wait()* debe detectar que el hijo ha muerto a causa de una señal y no por una terminación normal con un determinado estado o código (en el ejemplo es 2).

4.6 Ejercicio 6

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).

4.7 Ejercicio 7

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 (por la salida estándar abriéndolo y recorriéndolo o usando un comando *exec* invocando a un editor). Use la linea de argumentos para proporcionar el nombre de fichero a su programa.

4.8 Ejercicio 8

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 se utilizan métodos de comunicación entre procesos como por ejemplo, la memoria compartida[27], tuberías o pipelines[28] y colas de mensajes[29].

27 http://www.infor.uva.es/~cllamas/concurr/concurrencia.html
28 http://es.wikipedia.org/wiki/Tuber%C3%ADa_%28inform%C3%A1tica%29
29 http://labsopa.dis.ulpgc.es/prog_c/MSG.HTM