# Prácticas de Sistemas Operativos

## Módulo II. Uso de los Servicios del SO mediante la API

## Sesión 3. Llamadas al sistema para el Control de Procesos

### 1. Objetivos principales

Esta sesión trabajaremos con las llamadas al sistema relacionadas con el control y la gestión de procesos. El control de procesos en UNIX incluye las llamadas al sistema necesarias para implementar la funcionalidad de creación de nuevos procesos, ejecución de programas, terminación de procesos y alguna funcionalidad adicional como sería la sincronización básica entre un proceso padre y sus procesos hijo.

Además, veremos los distintos identificadores de proceso que se utilizan en UNIX tanto para el usuario como para el grupo (reales y efectivos) y como se ven afectados por las primitivas de control de procesos. En concreto, los objetivos son:

- Conocer y saber usar las órdenes para crear un nuevo proceso, finalizar un proceso, esperar a la terminación de un proceso y para que un proceso ejecute un programa concreto.
- Conocer las funciones y estructuras de datos que me permiten trabajar con los procesos.
- Comprender los conceptos e implementaciones que utiliza UNIX para construir las abstracciones de procesos, la jerarquía de procesos y el entorno de ejecución de éstos.

## 2. Creación de procesos

### *2.1 Identificadores de proceso*

Cada proceso tiene un único identificador de proceso (PID) que es un número entero no negativo. Existen algunos procesos especiales como el **`init`** (PID=1). El proceso **`init`** (proceso demonio o hebra núcleo) es el encargado de inicializar el sistema UNIX y ponerlo a disposición de los programas de aplicación, después de que se haya cargado el núcleo. El programa encargado de dicha labor suele ser el **`/sbin/init`** que normalmente lee los archivos de inicialización dependientes del sistema (que se encuentran en **`/etc/rc*`**) y lleva al sistema a cierto estado. Este proceso no finaliza hasta que se detiene al sistema operativo y no es un proceso de sistema sino uno normal aunque se ejecute con privilegios de superusuario (root). El proceso **`init`** es *el proceso raíz de la jerarquía de procesos del sistema*, que se genera debido a las relaciones entre proceso creador (padre) y proceso creado (hijo).

Además del identificador de proceso, existen los siguientes identificadores asociados al proceso y que se detallan a continuación, junto con las llamadas al sistema que los devuelven.

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

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

pid_t getpid(void); // devuelve el PID del proceso que la invoca.

pid_t getppid(void); // devuelve el PID del proceso padre del proceso que
                     // la invoca.

uid_t getuid(void); // devuelve el identificador de usuario real del
                    // proceso que la invoca.

uid_t geteuid(void); // devuelve el identificador de usuario efectivo del
                     // proceso que la invoca.

gid_t getgid(void); // devuelve el identificador de grupo real del proceso
                    // que la invoca.

gid_t getegid(void); // devuelve el identificador de grupo efectivo del
                     // proceso que la invoca.
```

El siguiente programa utiliza las funciones para obtener el pid de un proceso y el pid de su proceso padre. Pruébalo y observa sus resultados.

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

void main(void)
{
      pid_t id_proceso;
      pid_t id_padre;

      id_proceso = getpid();
      id_padre = getppid();

      printf("Identificador de proceso: %d\n", id_proceso);
      printf("Identificador del proceso padre: %d\n", id_padre);
}
```

El identificador de usuario real, **`UID`**, proviene de la comprobación que realiza el programa **`logon`** sobre cada usuario que intenta acceder al sistema proporcionando la pareja: ***login***, ***password***. El sistema utiliza este par de valores para identificar la línea del archivo de passwords (**`/etc/passwd`**) correspondiente al usuario y para comprobar la clave en el archivo de shadow passwords.

El UID efectivo (**`euid`**) se corresponde con el UID real salvo en el caso en el que el proceso ejecute un programa con el bit SUID activado, en cuyo caso se corresponderá con el UID del propietario del archivo ejecutable.

El significado del GID real y efectivo es similar al del UID real y efectivo pero para el caso del *grupo principal* del usuario. El grupo principal es el que aparece en la línea correspondiente al login del usuario en el archivo de passwords. El siguiente programa obtiene esta información sobre el proceso que las ejecuta. Pruébalo y observa sus resultados.

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

```
void main(void)
{
      printf("Identificador de usuario: %d\n", getuid());
      printf("Identificador de usuario efectivo: %d\n", geteuid());
      printf("Identificador de grupo: %d\n", getgid());
      printf("Identificador de grupo efectivo: %d\n", getegid());
}
```

### *2.2 Llamada al sistema `fork`.*

La única forma de que el núcleo de UNIX cree un nuevo proceso es que un proceso, que ya exista, ejecute **`fork`** (exceptuando los procesos especiales, algunos de los cuales hemos comentado brevemente en el punto anterior).

El nuevo proceso que se crea tras la ejecución de la llamada **`fork`** se denomina *proceso hijo*. Esta llamada al sistema se ejecuta una sola vez, pero devuelve un valor distinto en cada uno de los procesos (padre e hijo). La única diferencia entre los valores devueltos es que en el proceso hijo el valor es 0 y en el proceso padre (el que ejecutó la llamada) el valor es el PID del hijo. La razón por la que el identificador del nuevo proceso hijo se devuelve al padre es porque un proceso puede tener más de un hijo. De esta forma, podemos identificar los distintos hijos. La razón por la que **`fork`** devuelve un 0 al proceso hijo se debe a que un proceso solamente puede tener un único padre, con lo que el hijo siempre puede ejecutar **`getppid`** para obtener el PID de su padre.

Desde el punto de vista de la programación, el hecho de que se devuelva un valor distinto en el proceso padre y en el hijo nos va a ser muy útil, de cara a poder ejecutar distintas partes de código una vez finalizada la llamada **`fork`**. Para esto podremos hacer uso de instrucciones de bifurcación (ver tarea4.c).

Tanto el padre como el hijo continuarán ejecutando la instrucción siguiente al **`fork`** y el hijo será una copia idéntica del padre. Ambos procesos se ejecutarán a partir de este momento de forma concurrente.

**NOTA**: Existía una llamada al sistema similar a **`fork`**, **`vfork`**, que tenía un significado un poco diferente a ésta. Tenía la misma secuencia de llamada y los mismos valores de retorno que **`fork`**, pero **`vfork`** estaba diseñada para crear un nuevo proceso cuando el propósito del nuevo proceso era ejecutar, **`exec`**, un nuevo programa. **`vfork`** creaba el nuevo proceso sin copiar el espacio de direcciones del padre en el hijo, ya que el hijo no iba a hacer referencia a dicho espacio de direcciones sino que realizaba un **`exec`**, y mientras esto ocurría el proceso padre permanecía bloqueado(estado *sleep* en UNIX).

## *Actividad 2.1 Trabajo con la llamada al sistema `fork`*

Consulta con **`man`** la llamada al sistema **`fork`**.

**Ejercicio 1.** Implementa un programa en C que tenga como argumento un número entero. Este programa debe crear un proceso hijo que se encargará de comprobar si dicho número es un número par o impar e informará al usuario con un mensaje que se enviará por la salida estándar. A su vez, el proceso padre comprobará si dicho número es divisible por 4, e informará si lo es o no usando igualmente la salida estándar.

**Ejercicio 2.** ¿Qué hace el siguiente programa? Intenta entender lo que ocurre con las variables y sobre todo con los mensajes por pantalla cuando el núcleo tiene activado/desactivado el mecanismo de buffering.

```
/*
tarea4.c
Trabajo con llamadas al sistema de Control de Procesos "POSIX 2.10 compliant"
Prueba el programa tal y como está. Después, elimina los comentarios (1) y
pruébalo de nuevo.
*/

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

int global=6;
char buf[]="cualquier mensaje de salida\n";

int main(int argc, char *argv[])
{
int var;
pid_t pid;
var=88;
if(write(STDOUT_FILENO,buf,sizeof(buf)+1) != sizeof(buf)+1) {
    perror("\nError en write");
    exit(-1);
}

//(1)if(setvbuf(stdout,NULL,_IONBF,0)) {
//      perror("\nError en setvbuf");
//   }

printf("\nMensaje previo a la ejecución de fork");
if( (pid=fork())<0) {
    perror("\nError en el fork");
    exit(-1);
} else if(pid==0) {
    //proceso hijo ejecutando el programa
    global++;
    var++;
} else //proceso padre ejecutando el programa
    sleep(1);
printf("\npid= %d, global= %d, var= %d\n", getpid(),global,var);
exit(0);
}
```

**Nota 1**: El núcleo no realiza buffering de salida con la llamada al sistema **`write`**. Esto quiere decir que cuando usamos **`write(STDOUT_FILENO,buf,tama)`**, los datos se escriben directamente en la salida estándar sin ser almacenados en un búfer temporal. Sin embargo, el núcleo sí realiza *buffering* de salida en las funciones de la biblioteca estándar de E/S del C, en la cual está incluida **`printf`**. Para deshabilitar el buffering en la biblioteca estándar de E/S se utiliza la siguiente función:

```
int setvbuf(FILE *stream, char *buf, int mode , size_t size);
```

**Nota 2**: En la parte de llamadas al sistema para el sistema de archivos vimos que en Linux se definen tres macros **`STDIN_FILENO`**, **`STDOUT_FILENO`** y **`STDERR_FILENO`** para poder utilizar las llamadas al sistema **`read`** y **`write`** (que trabajan con ***descriptores de archivo***) sobre la

entrada estándar, la salida estándar y el error estándar del proceso. Además, en <stdio.h> se definen tres flujos (**STREAM**) para poder trabajar sobre estos archivos especiales usando las funciones de la biblioteca de E/S del C: **stdin**, **stdout** y **stderr**.

```
extern FILE *stdin;

extern FILE *stdout;

extern FILE *stderr;
```

¡Fíjate que **setvbuf** es una función que trabaja sobre STREAMS, no sobre ***descriptores de archivo***!

**Ejercicio 3.** Indica qué tipo de jerarquías de procesos se generan mediante la ejecución de cada uno de los siguientes fragmentos de código. Comprueba tu solución implementando un código para generar 20 procesos en cada caso, en donde cada proceso imprima su PID y el del padre, PPID.

```
/*
Jerarquía de procesos tipo 1
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (childpid)
    break;
}

/*
Jerarquía de procesos tipo 2
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (!childpid)
    break;
}
```

## *Actividad 2.2 Trabajo con las llamadas al sistema `wait`, `waitpid` y `exit`*

Consulta en el manual en línea las llamadas **wait**, **waitpid** y **exit** para ver sus posibilidades de sincronización entre el proceso padre y su(s) proceso(s) hijo(s) y realiza los siguientes ejercicios:

**Ejercicio 4.** Implementa un programa que lance cinco procesos hijo. Cada uno de ellos se identificará en la salida estándar, mostrando un mensaje del tipo `Soy el hijo PID`. El

proceso padre simplemente tendrá que esperar la finalización de todos sus hijos y cada vez que detecte la finalización de uno de sus hijos escribirá en la salida estándar un mensaje del tipo:

```
Acaba de finalizar mi hijo con <PID>
Sólo me quedan <NUM_HIJOS> hijos vivos
```

**Ejercicio 5.** Implementa una modificación sobre el anterior programa en la que el proceso padre espera primero a los hijos creados en orden impar (1º,3º,5º) y después a los hijos pares (2º y 4º).

## 3. Familia de llamadas al sistema *exec*

Un posible uso de la llamada **fork** es la creación de un proceso (el hijo) que ejecute un programa distinto al que está ejecutando el programa padre, utilizando para esto una de las llamadas al sistema de la familia **exec**.

Cuando un proceso ejecuta una llamada **exec**, el espacio de direcciones de usuario del proceso se reemplaza completamente por un nuevo espacio de direcciones; el del programa que se le pasa como argumento, y este programa comienza a ejecutarse en el contexto del proceso hijo empezando en la función **main**. El **PID** del proceso no cambia ya que no se crea ningún proceso nuevo.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

El primer argumento de estas llamadas es el camino del archivo ejecutable.

El **const char *arg** y los puntos suspensivos siguientes, en las funciones **execl**, **execlp**, y **execle**, son los argumentos (incluyendo su propio nombre) del programa a ejecutar: **arg0, arg1, ..., argn**. Todos juntos, describen una lista de uno o más punteros a cadenas de caracteres terminadas en cero. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. La lista de argumentos debe ser terminada por un puntero **NULL**.

Las funciones **execv** y **execvp** proporcionan un vector de punteros a cadenas de caracteres terminadas en cero, que representan la lista de argumentos disponible para el nuevo programa. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. El vector de punteros debe ser terminado por un puntero **NULL**.

La función **execle** especifica *el entorno del proceso que ejecutará el programa* mediante un parámetro adicional que va detrás del puntero **NULL** que termina la lista de argumentos de la lista de parámetros o el puntero al vector **argv**. Este parámetro adicional es un vector de punteros a cadenas de caracteres acabadas en cero y debe ser terminada por un puntero **NULL**. Las otras funciones obtienen el entorno para la nueva imagen de proceso de la variable externa **environ** en el proceso en curso.

Algunas particularidades de las funciones que hemos descrito previamente:

- Las funciones **execlp** y **execvp** actuarán de forma similar al shell si la ruta especificada no es un nombre de camino absoluto o relativo. La lista de búsqueda es la especificada en el entorno por la variable **PATH**. Si esta variable no está especificada, se emplea la ruta predeterminada "**:/bin:/usr/bin**".

- Si a un archivo se le deniega el permiso (**execve** devuelve **EACCES**), estas funciones continuarán buscando en el resto de la lista de búsqueda. Si no se encuentra otro archivo devolverán el valor **EACCES** en la variable global **errno** .

- Si no se reconoce la cabecera de un archivo (la función **execve** devuelve **ENOEXEC**), estas funciones ejecutarán el shell con el camino del archivo como su primer argumento.

- Las funciones **exec** fallarán de forma generalizada en los siguientes casos:

    - El sistema operativo no tiene recursos suficientes para crear el nuevo espacio de direcciones de usuario.

    - Utilizamos de forma errónea el paso de argumentos a las distintas funciones de la familia **exec**.

## *Actividad 3.1 Trabajo con la familia de llamadas al sistema exec*

Consulta en el manual en línea las distintas funciones de la familia **exec** y fíjate bien en las diferencias en cuanto a paso de parámetros que existen entre ellas.

**Ejercicio 6.** ¿Qué hace el siguiente programa?

```
/*
tarea5.c
Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX
2.10
*/
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>

int main(int argc, char *argv[]){

pid_t pid;
int estado;
if( (pid=fork())<0) {
    perror("\nError en el fork");
    exit(-1);
}
else if(pid==0) { //proceso hijo ejecutando el programa
    if( (execl("/usr/bin/ldd","ldd","./tarea5",NULL)<0)) {
        perror("\nError en el execl");
        exit(-1);
    }
```

```
}
wait(&estado);
/*
<estado> mantiene información codificada a nivel de bit sobre el motivo de
finalización del proceso hijo que puede ser el número de señal o 0 si alcanzó
su finalización normalmente.
Mediante la variable estado de wait(), el proceso padre recupera el valor
especificado por el proceso hijo como argumento de la llamada exit(), pero
desplazado 1 byte porque el sistema incluye en el byte menos significativo el
código de la señal que puede estar asociada a la terminación del hijo. Por eso
se utiliza estado>>8 de forma que obtenemos el valor del argumento del exit()
del hijo.
*/

printf("\nMi hijo %d ha finalizado con el estado %d\n",pid,estado>>8);
exit(0);

}
```

**Ejercicio 7.** Escribe un programa que acepte como argumentos el nombre de un programa, sus argumentos si los tiene, y opcionalmente la cadena "bg". Nuesto programa deberá ejecutar el programa pasado como primer argumento en `foreground` si no se especifica la cadena "bg" y en `background` en caso contrario. Si el programa tiene argumentos hay que ejecutarlo con éstos.

## 4. La llamada *clone*

A partir de este momento consideramos que **`tid`** es el identificador de hebra de un proceso. Como hemos visto en teoría, en Linux, **`fork()`** se implementa a través de la llamada **`clone()`** que permite crear procesos e hilos con un grado mayor en el control de sus propiedades. La sintaxis para esta función es

```
#define _GNU_SOURCE

#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg,
...

/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );

                  Retorna: si éxito, el PID del hijo; -1, si error
```

Cuando invocamos a **`clone`**, se crea un nuevo proceso hijo que comienza ejecutando la función dada por **`func`** (no la siguiente instrucción, como ocurre en **`fork`**), a la que se le pasa el argumento indicado en **`func_arg`**. El hijo finaliza cuando la función indicada retorna o cuando haga un **`exit`** o **`_exit`**. También, debemos pasar como argumento un puntero a la pila que debe utilizar el hijo, en el argumento **`child_stack`**, y que previamente hemos reservado.

Uno de los argumentos más interesantes de **`clone()`** son los ***indicadores de clonación*** (argumento **`flags`**). Estos tienen dos usos, el byte de orden menor sirve para especificar la *señal de terminación del hijo*, que normalmente suele ser **`SIGCHLD`**. Si esta a cero, no se envía señal. Una diferencia con **`fork()`**, es que en ésta no podemos seleccionar la señal, que siempre es **`SIGCHLD`**. Los restantes bytes de **`flags`** tienen el significado que aparece en **Tabla 1**.

**Tabla 1**.- *Indicadores de clonación*.

| Indicador | Significado |
|---|---|
| CLONE_CHILD_CLEARTID | Limpia **`tid`** (***t****hread* ***id****entifier*) cuando el hijo invoca a **`exec()`** o **`_exit()`**. |
| CLONE_CHILD_SETTID | Escribe el **`tid`** de hijo en **`ctid`**. |
| CLONE_FILES | Padre e hijo comparten la tabla de descriptores de archivos abiertos. |
| CLONE_FS | Padre e hijo comparten los atributos relacionados con el sistema de archivos (directorio raíz, directorio actual de trabajos y máscara de creación de archivos), |
| CLONE_IO | El hijo comparte el contexto de E/S del padre. |
| CLONE_NEWIPC | El hijo obtiene un nuevo namespace System V IPC |
| CLONE_NEWNET | El hijo obtiene un nuevo namespace de red |
| CLONE_NEWNS | El hijo obtiene un nuevo namespace de montaje |
| CLONE_NEWPID | El hijo obtiene un nuevo namespace de PID |
| CLONE_NEWUSER | El hijo obtiene un nuevo namespace UID |
| CLONE_NEWUTS | El hijo obtiene un nuevo namespace UTS |
| CLONE_PARENT | Hace que el padre del hijo sea el padre del llamador. |
| CLONE_PARENT_SETTID | Escribe el **`tid`** del hijo en **`ptid`** |
| CLONE_PID | Obsoleto, utilizado solo en el arranque del sistema |
| CLONE_PTRACE | Si el padre esta siendo traceado, el hijo también |
| CLONE_SETTLS | Describe el almacenamiento local (`tls`) para el hijo |
| CLONE_SIGHAND | Padre e hijo comparten la disposición de las señales |
| CLONE_SYSVSEM | Padre e hijo comparten los valores para deshacer semáforos[1] |
| CLONE_THREAD | Pone al hijo en el mismo grupo de hilos del padre |
| CLONE_UNTRACED | No fuerza `CLONE_PTRACE` en el hijo |
| CLONE_VFORK | El padre es suspendido hasta que el hijo invoca `exec()` |
| CLONE_VM | Padre e hijo comparten el espacio de memoria virtual |

Vamos a ver un programa ejemplo, para entender como se comporta la creación de procesos con la llamada **`clone`** dependiendo de los indicadores que utilicemos. No veremos todos por razones de tiempo/espacio.

El **Programa 3** nos muestra el uso de **`clone()`** en el que hemos utilizado los indicadores **`CLONE_VM|CLONE_FILES|CLONE_FS|CLONE_THREAD|CLONE_SIGHAND`**. De la Tabla 1, concluimos que el hijo comparte con su padre la memoria virtual, los archivos abiertos, el directorio raíz,

1 El kernel mantiene una lista para deshacer las operaciones de un semáforo cuando un proceso termina sin liberar los semáforos con el objetivo de minimizar la posibilidad de interbloqueo. Podeis ver http://eduunix.ccut.edu.cn/index2/html/linux/Interprocess%20Communications%20in%20Linux/ch07lev1sec2.htm.

el directorio de trabajo y la máscara de creación de archivos, pone al hijo en el mismo grupo del padre, y comparten los manejadores de señales. Es decir, creamos un hilo.

**Programa 3**.- Ejemplo de `clone()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <linux/sched.h>
#include <malloc.h>

#define _GNU_SOURCE             /* See feature_test_macros(7) */

int variable=3;

int thread(void *p) {
      int tid;

      printf("\nsoy el hijo\n");
      sleep(5);
      variable++;
      tid = syscall(SYS_gettid);
      printf("\nPID y TID del hijo:%d %d\n",getpid(),tid);
      printf("\nEn el hijo la variable vale:%d\n",variable);
}

int main() {

      void **stack;
        int i, tid;

      stack = (void **)malloc(15000);
      if (!stack)
            return -1;

         i = clone(thread, (char*) stack + 15000, CLONE_VM|CLONE_FILES|CLONE_FS|
CLONE_THREAD|CLONE_SIGHAND, NULL);
      sleep(5);
        if (i == -1)
                perror("clone");
      tid = syscall(SYS_gettid);
      printf("\nPID y TID del padre:%d %d\n ",getpid(),tid);
      printf("\nEn el padre la variable vale:%d\n",variable);

return 0;
}
```

En el programa anterior, hemos utilizado la llamada de Linux que citamos anteriormente (no es general de UNIX) **`gettid()`** que devuelve el identificador de una hebra, **`tid`**.

Si ejecutamos este proceso podemos observar como padre e hijo están en el mismo grupo puesto que su PID es igual, tal como establece el indicador **CLONE_THREAD**. Además, como hemos establecido **CLONE_VM**, el espacio de memoria es el mismo, por lo que **variable** es la misma en ambos.

```
$>./clone
PID y TID del hijo: 10549 10550
En el hijo la variable vale 4
PID y TID del padre: 10549 10549
En el padre la variable vale 4
```

Además, si durante la ejecución del programa lo paramos con **<Ctrl-Z>** podemos ver como efectivamente en el sistema hay dos hebras a través de **ps** (**NLWP** es el número de hilos) y como estas comparten la memoria ya que el tamaño de RSS (Resident Set Size – conjunto residente de memoria) es el mimo:

```
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME CMD
jagomez   6301  6297  6301  0    1  1364  2584   0 12:02 pts/1    00:00:00 /bin/bash
jagomez  11505  6301 11505  0    2   750   560   1 17:13 pts/1    00:00:00 ./c
jagomez  11505  6301 11506  0    2   750   560   1 17:13 pts/1    00:00:00 ./c
jagomez  11510  6301 11510  0    1   655   880   0 17:13 pts/1    00:00:00 ps -LfF
```

Si modificamos el programa anterior y eliminamos **CLONE_VM**, **CLONE_SIGHAND**, y **CLONE_THREAD**, los procesos padre e hijo no comparte la misma memoria virtual, ni el mismo grupo, como podemos ver si lo compilamos y ejecutamos.

```
$> ./clone
PID y TID del hijo: 10721 10721
En el hijo la variable vale 4
PID y TID del padre: 10720 10720
En el padre la variable vale 3
```

En este caso se trata de dos procesos, cada uno con una hebra y el RSS en diferente:

```
ps -LfF
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME CMD
jagomez   6301  6297  6301  0    1  1364  2584   1 12:02 pts/1    00:00:00 /bin/bash
jagomez  11408  6301 11408  0    1   750   804   1 17:09 pts/1    00:00:00 ./c
jagomez  11409 11408 11409  0    1   750   116   0 17:09 pts/1    00:00:00 ./c
jagomez  11413  6301 11413  0    1   655   884   0 17:09 pts/1    00:00:00 ps -LfF
```