Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
UniversitatPolitècnica de València

fSO

Práctica 4
Creación de procesos en UNIX
Versión 2.0

Contenido

1. Objetivos

La **creación de procesos** es uno de los servicios más importantes que proporcionar un sistema operativo multiprogramado. Esta práctica estudia las llamadas al sistema UNIX para crear procesos. Los objetivos son:

- Adquirir destreza en la utilización de las llamadas POSIX, **fork(), exec(), exit() y wait().**
- Analizar mediante ejemplos de programas el comportamiento de las llamadas en términos de **valores de retorno, atributos heredados entre procesos.**
- Visualizar los diferentes **estados de los procesos, ejecución(R), suspendido (S), zombie (Z)**

En esta práctica se trabajan los conceptos y llamadas impartidos en el Seminario 3 titulado "*Llamadas al sistema UNIX para procesos*", de la asignatura Fundamentos de Sistemas Operativos.

1. Introducción

UNIX utiliza un mecanismo de clonación para la creación de procesos nuevos. Se crea una nueva estructura PCB que da soporte a la copiad el proceso que invoca la **llamada al sistema fork()**.Ambos procesos, creado (hijo) y creador (padre) son idénticos en el instante de creación pero con **identificadores PID y PPID distintos**. El **proceso hijo hereda** una copia de los **descriptores de archivo, el código y las variables del proceso padre**. Cada proceso tiene su propio espacio de memoria, las variables del proceso hijo están ubicadas en posiciones de memoria diferentes a las del padre, por lo que cuando se modifica una variable en uno de los procesos no se refleja en el otro. Resumiendo la llamada a *fork()* crea un nuevo proceso "hijo":

- Que es una copia exacta del creador "padre" excepto en el PID y PPID
- Que hereda las variables del proceso padre y evolucionan independientemente del padre
- Que hereda del proceso padre la tabla de descriptores pudiendo compartir archivos con él

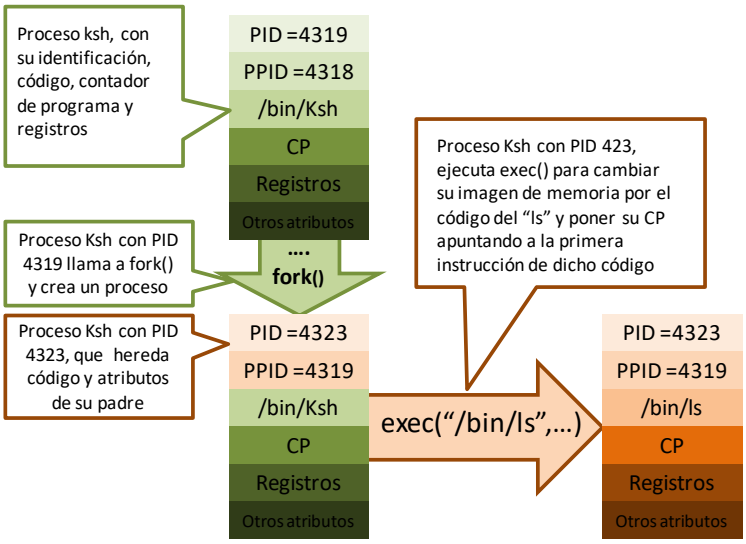
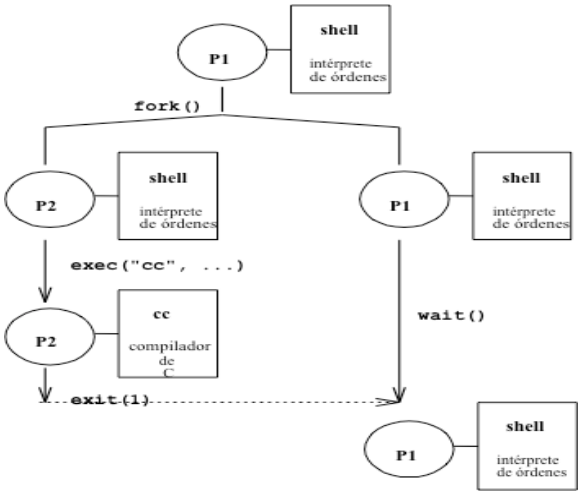


Figura-1: Llamadas fork() y exec()

La llamada exec() cambia el mapa de memoria de un proceso, cambia el contenido de la memoria del proceso que lo invoca, es decir, carga nuevas regiones de datos, código etc. y pone el contador de programa de dicho proceso apuntando a la primera instrucción. El código se carga a partir de un archivo de disco que contiene código ejecutable. El sistema operativo crea las regiones correspondientes y carga sus valores en los datos, interpretando dicho archivo ejecutable (en formato ELF).

La creación de procesos genera dentro del sistema un árbol de parentesco entre ellos que vincula a los procesos en términos de sincronización. Un caso de aplicación típico de la sincronización entre padre e hijos es el Shell. Esta sincronización del Shell con su hijo se realiza mediante las llamadas *wait()* o *waitpid()* y *exit()*.En el Anexo

de la práctica se detallan, la llamada *exit()* y *wait()*. exit() finaliza la ejecución de un proceso y wait() suspende la ejecución del proceso que la invoca hasta que alguno de sus procesos hijo termine.



El Shell de Unix crea un proceso hijo por cada orden invocada desde el terminal. El proceso Shell espera a que el proceso que ejecuta el comando finalice e imprime el *prompt* en pantalla (excepto en la ejecución en background) y queda a la espera de un nuevo comando.

Figura-2: El Shell de UNIX

Utilice también el manual del sistema para consultar detalles de estas llamadas

```
$man fork
```

2. Creación de procesos en UNIX con fork()

UNIX crea procesos nuevos con la llamada al sistema `fork()`. El código de la figura-3 crea un proceso hijo mediante la llamada fork().

Cree un nuevo directorio para almacenar todos los archivos de esta práctica con:

```
$mkdir $HOME/pract4
```

```c
/***code of my_child.c***/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{    printf("Process %ld \n", (long)getpid());
     fork();
     printf("Process %ld,my parent %ld\n" ,(long)getpid(),(long)getppid());
     sleep(15);
     return 0;
}
```

Figura-3: Código de my_child.c

Ejercicio 1: Creando proceso hijo con la llamada fork() "my_child.c"

Cree un archivo denominado *my_child.c y* escriba en él el código de la figura-3, compílelo y ejecútelo en background. Rellene la tabla 1 con los PID, PPID y la orden que ejecutan los procesos creados.

```
$ gcc –o my_child my_child.c
$./my_child&
$ps -la
```

	PID	PPID	COMMAND
Proceso Padre	3085	2908	./my_child
Proceso hijo	3086	3085	./my_child

Tabla-1: Ejercicio-1, creando procesos con la llamada fork()

3. Valores de retorno del fork()

El valor de retorno de la llamada `fork()` permite identificar al proceso padre e hijo y decidir que código ejecutan cada uno. `fork()` retorna un 0 a los hijos y un valor positivo, el PID del hijo, al proceso padre. La figura-4 ilustra el concepto de los valores retornados por `fork()` al proceso padre e hijo.

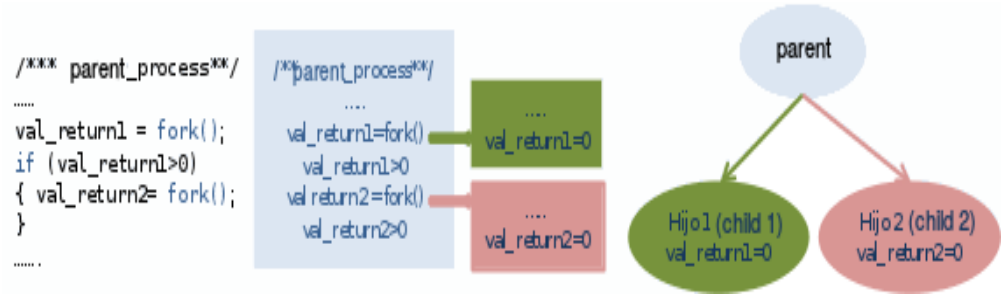


Figura-4: De izquierda a derecha se muestra, el código en c que nos permite diferenciar el valor retornado por `fork()` a los procesos padre e hijo. El árbol de procesos que se genera.

El esquema general de código en C a utilizar para que un proceso padre cree un número n de procesos hijos en abanico, es el mostrado en la figura-5.

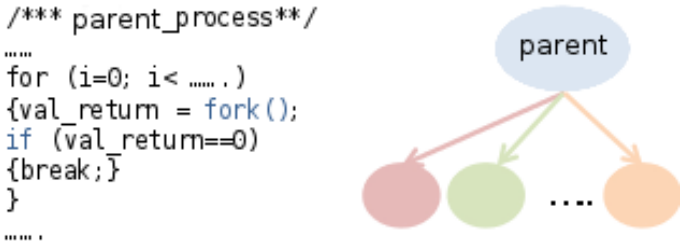


Figura-5: Esquema de procesos creados en abanico, estructura de código y árbol de parentesco.

Ejercicio 2: Valor de retorno de la llamada fork() "range_process.c"

Cree un archivo denominado *range_process.c* que contenga el código necesario para la creación de 5 procesos hijos en abanico. Introduzca las llamadas e instrucciones necesarias para que:

- Mediante **`i`** iteraciones de un bucle for, se creen 5 procesos en abanico.
- Cada hijo escriba "`printf("Hijo creado en iteración=%d",i)`" y después terminar su ejecución con una llamada a `exit(i)`.
- El proceso padre tras crear a todos los hijos realice un `sleep(10)`y a continuación `exit(0)`.

Compile el programa y ejecútelo en background. Inmediatamente ejecute "`$ps l`". Rellene la tabla -2.

	PID	PPID	COMMAND	ESTADO
Proceso Padre	8398	7403	./rp	S
Proceso hijo 1	8399	8398	[rp] <defunct>	Z
Proceso hijo 2	8400	8398	[rp] <defunct>	Z
Proceso hijo 3	8401	8398	[rp] <defunct>	Z
Proceso hijo 4	8402	8398	[rp] <defunct>	Z
Proceso hijo 5	8403	8398	[rp] <defunct>	Z

Tabla-2: Ejercicio-2, creando procesos en abanico

Pregunta-1: Justifique porqué están en ese estado los procesos hijo generados por *range_process.c*

Porque están esperando que el padre termine de ejecutar sus instrucciones porque hizo una llamada a sleep.

5. El proceso init()

En los sistemas Unix, *init*(*initialization*) es el primer proceso en ejecución tras la carga del núcleo del sistema y es el responsable de establecer la estructura de procesos. Init se ejecuta como demonio (daemon) y por lo general tiene PID 1. Todos los procesos en el sistema, excepto el proceso *swapper*, descienden del proceso *init*.

Al iniciar un sistema UNIX, se realiza una secuencia de pasos conocidos por *bootstrap* cuyo objetivo es cargar una copia del sistema operativo en memoria principal e iniciar su ejecución. Una vez cargado el núcleo, se transfiere el control a la dirección de inicio del núcleo, y comienza a ejecutarse. El núcleo inicializa sus estructuras de datos internas, monta el sistema de archivos principal y crea el proceso 0 denominado *swapper*.

El proceso 0 crea el proceso init con una llamada fork(). El proceso *init*, lee el archivo /etc/ttytab y crea un proceso login por cada terminal conectado al sistema. Después de que un usuario introduzca un login y un pasword (archivo /etc/passwd) correcto, el proceso login lanza un intérprete de órdenes o Shell.
A partir de ese momento el responsable de atender al usuario en esa terminal es el intérprete Shell, mientras que el proceso login se queda dormido hasta que se realice un logout.

Nota: Los procesos demonios (daemon) hacen funciones del sistema, pero se ejecutan en modo usuario.

Ejercicio 3: Procesos adoptados por INIT "adopted_process.c"

init es considerado el patriarca de todos los procesos en un sistema UNIX. Comprobaremos que *init* adopta a todos aquellos procesos cuyo proceso padre ha finalizado sin esperar a que ellos finalicen. Para ello tendremos que asegurarnos de que los hijos permanecen en el sistema después de que su padre haya finalizado.

Copie el archivo *range_process.c* en *adopted_process.c* utilizando la orden cp:

```
$cp range_process.c  adopted_process.c
```

Edite el archivo *adopted_process.c* y añada un `sleep(20)` antes del `exit(i)` de cada proceso hijo. Asegúrese de que los hijos están suspendidos en`sleep()`durante más tiempo que el padre. Compile *adopted_process.c* y ejecútelo en background, a continuación, ejecute "`$ps l`" varias veces seguidas hasta que finalicen los procesos. Fíjese en el PPID de los procesos hijos al inicio y al final y anótelos en la tabla-3.

	PID	PPID		COMMAND	ESTADO
Proceso Padre	8422	1467		./adopted_process	S
Proceso hijo 1	8423	8422	1	./adopted_process	S
Proceso hijo 2	8424	8422	1	./adopted_process	S
Proceso hijo 3	8425	8422	1	./adopted_process	S
Proceso hijo 4	8426	8422	1	./adopted_process	S
Proceso hijo 5	8427	8422	1	./adopted_process	S

Tabla-3: Ejercicio-3, procesos adoptados

Pregunta 2: Entre los procesos creados al ejecutar *adopted_process* ¿Qué proceso finaliza en primer lugar y como afecta esta finalización al resto de procesos?

El primer proceso en finalizar es el proceso padre cuyo pid es 8422, y al este acabar primero que sus hijos, sus hijos quedan huerfanos y son adoptados por el proceso init y sus ppid cambian a 1.

6. Espera de procesos con wait()

Hay que evitar que se generen procesos zombies ya que sobre cargan el sistema. Un proceso pasa al estado zombie cuando él invoca la llamada `exit()` y su padre todavía no ha ejecutado la llamada `wait()`. Para evitar procesos zombies, un proceso padre debe esperar siempre a sus hijos invocando `wait()` o `waitpid()`.
La llamada wait() es bloqueante ya que suspende la ejecución del proceso que la invoca, hasta que finaliza alguno de sus hijos. En el anexo se describe con detalle esta llamada.

Ejercicio 4: Padre debe esperar "parent_wait.c"

En este ejercicio modificaremos el programa *range_process.c* de manera que no genere procesos zombies al ejecutarlo. Copie el archivo *range_process.c* en otro denominado *parent_wait.c*

```
$cp range_process.c parent_wait.c
```

Edite *parent_wait.c* añada las instrucciones y llamadas necesarias para que el proceso padre espere a sus hijos. El proceso padre debe esperar con `wait()` a todos los hijos.
Recuerde que debe compilar y ejecutar

```
$ gcc  parent_wait.c  –o  parent_wait
$./parent_wait&
$ps  -la
```

Pregunta3: ¿Se generan procesos zombies al ejecutar parent_wait? Justifique su respuesta.

No se generan procesos zombies ya que ningún proceso ha finalizado y el proceso padre espera a que finalicen sus hijos que están ejecutando el sleep, y los procesos están suspendidos.

Ejercicio 5: Comunicando estado de finalización al padre "final_state.c"

La llamada exit() permite que el hijo le pase información sobre su estado de finalización al padre mediante el parámetro stat_loc de la llamada wait(int *stat_loc). Copie *parent_wait.c* en el archivo *final_state.c*

```
$cp   parent_wait.c   final_state.c
```

Modifique el código de *final_state.c* para que genere la secuencia de procesos de la figura-6, es decir:

- Utilizando un bucle for y una variable i, cree 4 procesos que muestren un mensaje indicando la iteración i en la que son creados.
- Todos los procesos deben esperar 10 segundos antes de invocar exit().
- Todos los padres deben esperar la finalización de su hijos con wait (&stat_loc), e imprimir su PID y el valor del parámetro stat_loc.

¡AVISO!: El valor máximo con el que se puede invocar la llamada a exit() es 255 (8 bits). Para obtener los 8 bits mas significativos, se utiliza WEXITSTATUS(stat_loc).

```
/*** final_state **/
int  final_state;
pid_t  val_return;
……
for (i=0; i<MAX_PROC; i++)
{val_return= fork();
  if (val_return==0)
   {printf("Son %ld created in iteration %d\n",(long) getpid(),i);
    }
  else
  { /*father out of the loop for */
   printf("Father %ld, iteration %d\n", (long)getpid(),i);
   printf("I have created a son %ld\n",(long) val_return);
   break;
  }
}

/*wait all my sons and make exit*/
While(wait(&final_state)>0)
{printf("Father %ld  iteration %d",(long) getpid(),i);
 printf( "My son said %d\n",WEXITSTATUS(final_state));
 printf( "My son said %d\n", final_state/256);
}
exit(i);
….
```

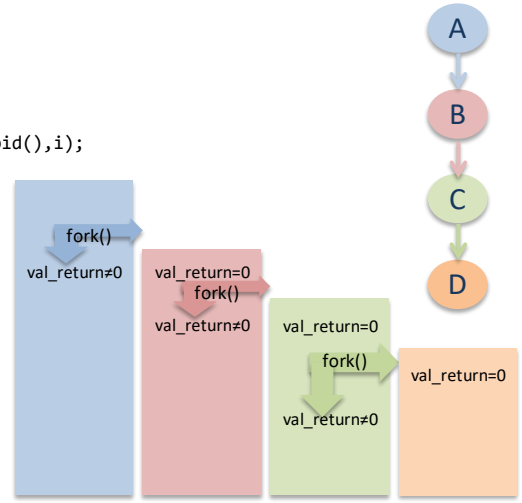


Figura-6: Esquema de procesos en vertical.

Compile *final_state.c* y ejecútelo. Anotar en la tabla los resultados:

	PID	Valor de finalización
Proceso Padre	8618	1
Proceso creado con i=0	8619	2
Proceso creado con i=1	8620	3
Proceso creado con i=2	8621	4
Proceso creado con i=3	8622	85

Tabla-4: Ejercicio-5

Pregunta4: ¿Cuál es la primera instrucción que ejecutan los procesos hijos al ser creados? ¿Cuántas iteraciones del bucle realiza cada proceso?

La primera instrucción es el printf y luego ejecuan el sleep, cada proceso ejecuta una iteración del bucle for.

7. La llamada `exec()`

La llamada `exec()` permite cambiar la imagen de memoria o el mapa de memoria del proceso que la invoca. En el anexo de esta práctica se encuentran detalladas las 6 variantes que existen de esta llamada.
La figura-7 ilustra el uso de la llamada fork y exec para crear un nuevo proceso que ejecute el comando "ls".

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

int main(int argc, char *argv[])
{ pid_t childpid;
  int status, x;
  char* argumentos [] = { "ls", "-R", "/", 0 };
```

Variantes:
`execvp(argumentos[0],argumentos);`
`execvp(argv[1], &argv[1]);`

```
 chilpid=fork();
 if (childpid == -1)
    {printf("fork failed\ n"); exit(1);}
 else if (childpid ==0)
    { if(execl("/bin/ls", "ls", "-1", NULL) <0 )
     {printf("Could not execute: ls\n");
      exit(1);}
    }
 x=wait(&status);
 if ( x!= childpid)
 printf("Child has been interrupted by a signal\n");

exit(0);
}
```

Figura-7: Código para crear un nuevo proceso que ejecute el "ls"

Ejercicio 6: Creando un proceso para ejecutar el "ls"

Sin ejecutar el códigode la figura-7, indique qué proceso/s padre, hijo o ambos invocan los diferentes *printf* y si lo muestran por pantalla siempre que se ejecuta el código o sólo cuando se cumplen ciertas condiciones.

	Padre ¿siempre? ¿Qué condición?	Hijo ¿siempre? ¿Qué condición?
"fork failed\n"	Invoc el printf al no poder crear un hijo.	No invoca el printf porque no ha sido creado
"Could not execute: ls\n"	No invoca el printf, porque no es un hijo.	Invoca el printf al np no poder ejecuar el exec
"Child has been interrupted by a signal\n"	Invoca el printf en caso de que su hijo halla finalizado antes de la llamada wait	No invoca el printf porque no tiene hijo

Tabla-5: Ejercicio-6

Ejercicio 7: Intercambio de memoria de un proceso "change_memory1.c"

El archivo suministrado *change_memory1.c* contiene el código de la figura-7, donde el hijo ejecuta la orden:

```
$ ls  -R  /
```

Mientras se ejecuta *change_memory1.c*, abra otra consola y ejecute un comando "$ps - l" y anote en la tabla-6 los resultados asociados a él.

	PID	PPID	COMMAND
Proceso Padre	3128	2908	./change_memory
Proceso hijo	3129	3128	ls -R /

Tabla-6: Ejercicio-7

Pregunta 5: ¿Cuál es la principal diferencia entre los resultados de esta tabla y los obtenidos en el ejercicio 1?

La principal diferencia es la linea de comando, ya que el hijo al ejecutar el execl() cambia su mapa de memoria por el del proceso lr -R /

Ejercicio 8: Orden *execl()* en *"change_memory1.c"*

Practique la versiones del *exec*(), modifique en*change_memory1.c*, y utilice la orden execl() donde se ha de especificar como strings los literales que forman parte de la misma

execl("/bin/ls", "ls", "-R", "/", NULL);

Ejercicio 9: Paso de argumentos *"change_memory2.c"*

Para aumentar la funcionalidad del programa *change_memory1.c* y poder ejecutar la orden *ls* con diferentes argumentos que se pasarán al programa mediante la línea de comandos, utilizaremos los dos argumentos predefinidos en la función *main (intargc ,char *argv[])*. Como ya conoce de la práctica de C, el parámetro *argc* es un entero que contiene el número de argumentos con que se invoca la orden. Como mínimo siempre existe un argumento el argv[0] o nombre del programa. El parámetro *argv[]* es un vector de punteros a caracteres que apuntan a las cadenas de los argumentos que se pasan en la línea de órdenes.

A partir de *change_memory1.c* cree*change_memory2.c* para que ejecute las órdenes pasadas como argumentos, utilice las variables *argc* y *argv*.

Compruebe con las siguientes ejecuciones:

```
$ ./change_memory2 ps
$ ./change_memory2 ps -l
$ ./change_memory2 ls -lh
$ ./change_memory2 ls -lht
```

Ejercicio Opcional:

En la figura-8 se muestra un código que realiza la suma de todos los elementos de una matriz. La matriz tiene de dimensiones NUMROWS y DIMROWS. Se realiza un doble bucle, uno para obtener la suma de todas las columnas y de una misma fila y otro para realizar la suma resultante de cada fila.

```
#include <stdio.h>
#include <math.h>
#define DIMROW 100
#define NUMROWS 20

typedef struct row
{ int vector [DIMROW];
  long add;
} row;

row matrix[NUMROWS];
int main()
{
  int i, j, k;
  long total_add = 0 ;
  // Initializing to 1 all the elements of the vector
  for (i =0; i< NUMROWS; i ++) {
    for (j =0; j < DIMROW; j ++)
    { matrix [i] [j] .vector = 1 ;
      array[i].add = 0 ;
    }
  }
  for (i =0; i< NUMROWS; i ++) {
    for (k =0; k < DIMROW; k ++)
     array[i].add += array[i][k].vector;
   }
  for (i =0; i< NUMROWS; i ++)
   total_add += array[i].add;
printf ("The total addition is %ld\n", total_add);
}
```

Figura-8: Código de addrows.c para sumar las filas de una matriz

Descargue el código de *addrows.c* del poliformat (), compílelo y ejecútelo. Para su ejecución utilice la orden $*time* que nos permite averiguar el tiempo que tarda en ejecutarse un programa. Para ello invoque en el shell

```
$time ./addrows
```

Anote el tiempo devuelto por el comando y el resultado de total de la suma.

Se pide:
Cree un programa llamado addrows*pro.c* en el que se creen tantos procesos como filas, cada proceso se encargará de la suma de una fila (calcular `matrix[i].add`). Cada proceso hijo finalizará con un `exit(matrix[i].add)`. El proceso padre debe esperar a todos los procesos hijos para acumular todos estos valores en la variable `total_add`, que finalmente imprimirá por pantalla.

Ejecute el programa con la orden *$time*. El valor de la suma total debe ser el mismo que en la versión sin procesos hijo. Anote el valor de tiempo que devuelve y compare con la versión anterior

NOTA: El mecanismo de retorno mediante la llamada exit() está orientado para que los hijos comuniquen a su padre su estado codificable en 8 bits y no el valor resultante de un cálculo. Por tanto, el uso de exit() que se hace en el ejercicio anterior recae fuera de lo que sería la guía de programación, utilizándose en este ejemplo tan solo para suplir la carencia de otros mecanismos de comunicación entre procesos.

8. Anexo

Anexo1: Sintaxis llamadas al sistema

Una llamada al sistema se invoca mediante una función que siempre retorna un valor con la información acerca del servicio proporcionado o del error que se ha producido en su ejecución. Dicho retorno puede ser ignorado, pero es recomendable testearlo.

Valores de retorno de las llamadas:

- Retornan un valor de -1 o puntero a NULL cuando se produce error en la ejecución de la llamada
- Existe una variable externa *errno,* que indica un código sobre el tipo de error que se ha producido. En el archivo de *errno.h* (#include<errno.h>) hay declaraciones referentes a esta variable.
- La variable *errno* no cambia de valor si una llamada al sistema retorna con éxito, por tanto, es importante tomar dicho valor justo después de la llamada al sistema y únicamente cuando éstas retornan error, es decir, cuando la llamada retorna -1.

Anexo 2: Llamada fork()

```
#include <stdio>
pid_t    fork(void)
```

- Descripción:
  - Crea un proceso hijo que es un “clon” del padre: hereda gran parte de sus atributos.

- Atributos heredables: todos excepto PID, PPID, señales pendientes, tiempos/contabilidad.
- Valor de retorno:
  - 0 al hijo
  - PID del hijo al padre
  - -1 al padre si error.
- Error

  Se produce error cuando hay insuficiencia de recursos para crear el proceso

Anexo 3: Llamada exec()

Existen 6 variantes de la llamada exec(), que se diferencian en la forma en que son pasados los argumentos (l=lista o v=array) y el entorno (e), y si es necesario proporciona la ruta de acceso y el nombre del archivo ejecutable.

```
#include <unistd.h>
void  execl (const char *path, const char *arg0, ... , const char *argn, (char*) 0 )
void  execle(const char *path, const char *arg0, ... ,const char *argn, (char*) 0, char *constenvp[])
void  execlp(const char *file, const char *arg0, ... , const char *argn, (char*) 0)

void  execv (const char *path, char *constargv[])
void  execve(const char *path, char *constargv[],  char *constenvp[])
 void  execvp(const char *file,   char *constargv[])
```

Las llamadasexecl (execl, execlp, execle) pasan los argumentos en una lista y son útiles si se conocen los argumentos en el momento de la compilación.
Ejemplos de execl:execl("/usr/bin/ls", "ls","-l",NULL);
execlp ("ls","ls","-l",NULL);
Las llamadas execv (execv, execve, execvp) pasan los argumentos en un array.
Ejemplo de execv: execvp(argv[1], &argv[1]);

- Descripción de exec():
  - Cambia la imagen de memoria de un proceso por la definida en un fichero ejecutable.
  - El fichero ejecutable se puede expresar dando su nombre file o su ruta completa path.
  - Algunos atributos del proceso se conservan y, en particular:
    - Manejo de señales, excepto las señales capturadas cuya acción será la de defecto.
    - PID y PPID, dado que no se crea un proceso nuevo sólo cambia su imagen
    - Tiempos de CPU(contabilidad)
    - Descriptores de archivos
    - Directorio de trabajo, el directorio raíz, la máscara del modo de creación de archivos,
  - Si bit SETUID del archivo ejecutable está activado, execpone como UID efectivo del proceso al UID del propietario del archivo ejecutable.
    - Ídem con el bit SETGID
- Errores:
  - Fichero no existente o no ejecutable
  - Permisos
  - Argumentos incorrectos
  - Memoria o recursos insuficientes
- Valor de retorno:
    - –1 si exec() retorna al programa que lo llamó lo cual indica que ha ocurrido un error.

- Descripción de parámetros:

- path: Nombre del archivo ejecutable se ha de indicar su trayectoria. Ejemplo: "/bin/cp".
- file: Nombre del archivo ejecutable, se utiliza la variable de entorno PATH para localizarlo.
- arg0: Primer argumento, corresponde al nombre del programa sin la trayectoria. Ejemplo: "cp"
- arg1 ...argN: Conjunto de parámetros que recibe el programa para su ejecución.
- argv: Matriz de punteros a cadenas de caracteres. Estas cadenas de caracteres constituyen la lista de argumentos disponibles para el nuevo programa. El último de los punteros debe ser NULL. Este array contiene al menos un elemento en argv[0], nombre del programa.
- envp: Matriz de punteros a cadenas de caracteres, constituyen el entorno de ejecución del nuevo programa.

Anexo 4: Llamada exit()

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

- Descripción:
  - Termina "normalmente" la ejecución del proceso que la invoca.
  - Si no se invoca explícitamente, se hace de forma implícita al finalizar todo proceso.
  - El estado de terminación *status* se transfiere al padre que ejecuta *wait(&status)*.
  - Si el padre del proceso no está ejecutando *wait*, se transforma en un zombie.
  - Cuando un proceso ejecuta *exit*, todos sus hijos son adoptados por init y sus PPID pasan a ser 1.
- Valor de retorno:
  - Ninguno
- Errores:
  - Ninguno

Anexo 5: Llamada wait()

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

pid_t wait(int *stat_loc)
pid_t waitpid(pid_tpid, int *stat_loc, int options)
```

- Descripción:
  - Suspende la ejecución del proceso que la invoca, hasta que alguno de los hijos (wait) o un hijo en concreto (waitpid) finalice.
  - Si existe un hijo zombie, wait finaliza inmediatamente, sino, se detiene.
  - Cuando `stat_loc` no es el puntero nulo, contiene:
    - Si **Hijo termina con exit**:
      MSB: status definido por exit, LSB: 0
    - Si **Hijo termina por señal**:
      MSB: 0, LSB: número de señal de señal (bit más alto 1: coredump)
- Valor de retorno:
  - El PID del hijo que ha finalizado excepto:
  - -1: se recibe una señal o error (no existen hijos).
- Errores
  - El proceso no tiene hijos
- Descripción waitpid
  - Argumento `pid`
    - pid< -1 esperar cualquier hijo cuyo grupo de procesos sea igual al pid
    - pid = -1: esperar cualquier hijo (como wait)

        - pid= 0: esperar cualquier hijo con el mismo grupo de procesos
        - pid>0: esperar al hijo cuyo pid es el indicado
    - Argumento `options`
        - WNOHANG: retornar inmediatamente si el hijo no ha terminado

La finalización del proceso hijo con exit (status) permite almacenar el estado de finalizaciónen la dirección apuntada por el parámetro stat_loc. Estándar POSIX define macros para analizar elestado devuelto por el proceso hijo:

- WIFEXITED, terminación normal.
- WIFSIGNALED, terminación por señal no atrapada.
- WTERMSIG, terminación por señal SIGTERM.
- WIFSTOPPED, el proceso está parado.
- WSTOPSIG, terminación por señal SIGSTOP (que no se puede atrapar ni ignorar).