sesion3.pdf

Módulo II

2º Sistemas Operativos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

Sesión 3:

Llamadas al sistema para el control de procesos

Índice:

1. Creación de procesos

1.1 Identificadores de proceso

Cada proceso tiene un único identificador de proceso (**PID**); dicho PID es un **número entero no negativo**.

Proceso init

Existen algunos **procesos especiales** como init, cuyo PID = 1. Este proceso:

- inicializa el sistema UNIX y lanza todos los procesos;
- pone el sistema a disposición de los programas de aplicación;
- no finaliza hasta que se detiene el sistema operativo;
- es un proceso normal (no de sistema) aunque se ejecute con privilegios de superusuario (root);
- 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).

El programa **/sbin/init** es el encargado de realizar los dos primeros puntos de arriba: lee los archivos de inicialización dependientes del sistema (se encuentran en /etc/rc*) y lleva al sistema al estado indicado.

Las funciones para acceder a los identificadores asociados a cualquier proceso:

- **pid_t getpid(void)**: devuelve el identificador de proceso que invoca la función;
- **pid_t getppid(void):** devuelve el PID del padre del proceso que invoca la función;
- **uid_t getuid(void):** devuelve el identificador de usuario real (UID) del proceso que la invoca;
- **uid_t geteuid(void):** devuelve el identificador de usuario efectivo (EUID) del proceso que la invoca;
- **gid_t getgid(void):** devuelve el identificador de grupo real (GID) del proceso que la invoca;
- **gid_t getegid(void):** devuelve el identificador de grupo efectivo (EGID) del proceso que la invoca.

Las estructuras pid_t, uid_t y gid_t son de tipo **entero**. Los identificadores reales y efectivos:

- El identificador de **usuario** (UID) y de **grupo real** (GID) los obtiene el sistema gracias al programa logon que realiza una comprobación del usuario en la carpeta /etc/passwd; de aquí obtiene el UID y el GID, el GID del grupo principal.
- El identificador de **usuario** (EUID) y de **grupo efectivo** (EGID) se corresponde con el UID/GID real salvo en el caso de que el programa a ejecutar tenga activado el SUID/SGID, en cuyo caso se corresponderá con el del propietario del programa a ejecutar.

1.2 fork

Para crear un proceso hijo.

```
pid_t fork();
```

- en caso de éxito, devuelve el **PID** del nuevo proceso creado (proceso hijo) al proceso padre; y devuelve **0** al proceso creado (proceso hijo);
- en caso de error, no crea el proceso hijo, devuelve -1 en el proceso padre y actualiza errno según el error.

El proceso que realiza la llamada se le conoce como **proceso padre** mientras que el nuevo proceso creado es el **proceso hijo**. Una vez realizado fork, tanto el padre como el hijo ejecutan la siguiente instrucción del programa, ejecutándose ambos procesos de forma concurrente.

- El proceso hijo es un **duplicado exacto** del proceso padre, excepto en el PID y PPID del proceso hijo y en el uso de recursos, en el que en el hijo está asignado a 0.
- El proceso hijo y el proceso padre se ejecutan en **espacios de memoria separados**.
- En el momento de fork(), ambos espacios de memoria tienen el mismo contenido.
- Las escrituras de memoria, las asignaciones de archivos y desasignaciones realizadas por uno de los procesos no afecta al otro.

Con esta llamada, el proceso padre y el proceso hijo ejecutan el mismo programa.

Los errores más comunes son:

- **EAGAIN:** hay un límite en el número de hebras impuesto por el sistema.
- **ENOMEM:** no pudo asignar las estructuras de kernel necesarias porque la memoria es insuficiente.
- **ENOSYS:** fork no es admitido en la plataforma.

Existía la llamada **vfork**, que realizaba lo mismo que fork, pero se diferenciaba en que el propósito del nuevo proceso hijo era ejecutar un nuevo programa (no el mismos que el padre). vfork realizaba un exec, la ejecución de otro programa, y el padre permanecía bloqueado hasta que el hijo terminaba.

1.3 Ejercicio 1

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.

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

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

        pid_t pid;
        int num;
```

```
        // comprobamos que se ha pasado un argumento
        if(argc != 2){
                //mostramos mensaje de error
                perror("ERROR en argumentos.\n El formato es: ./ejercicio1 <número>\n");
                exit(-1);
        }

        //obtenemos el número pasado como argumento, pasamos de char a int
        num = atoi(argv[1]);

        // creamos un proceso hijo y comprobamos que no ha habido error
        if( (pid = fork()) < 0) {
                perror("Error en el fork\n");
                exit(-1);
        }
        // si el que ejecuta este trozo de código es el hijo
        else if(pid == 0) {
                //comprobamos si el número es par o impar y lo mostramos por pantalla
                if(num%2 == 0){
                        printf("H: El número %i es par.\n", num);
                }
                else{
                        printf("H: El número %i es impar.\n", num);
                }
        }
        // si el que ejecuta este código es el padre
        else{
                //comprobamos si el número es divisible por 4
                if(num%4 == 0){
                        printf("P: El número %i es divisible por 4.\n", num);
                }
                else{
                        printf("P: El número %i no es divisible por 4.\n", num);
                }
        }

        exit(0);
}
```

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>
#include<stdlib.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("\n pid= %d, global= %d, var= %d\n", getpid(), global, var);
        exit(0);
}
```

El programa realiza lo siguiente:

- escribe buf en la salida estándar, es decir, muestra en la terminal "cualquier mensaje de salida";
- muestra un mensaje previo a la ejecución de fork;
- crea un proceso hijo con fork, el hijo incrementa el valor de global y var;
- el padre espera un segundo;
- después de todo esto, se muestra por pantalla el valor del pid, global y var.

La salida con (1) comentado es:

cualquier mensaje de salida
Mensaje previo a la ejecución de fork
pid= 48592, global= 7, var= 89
Mensaje previo a la ejecución de fork
pid= 48591, global= 6, var= 88

La salida con (1) descomentado es:

cualquier mensaje de salida
Mensaje previo a la ejecución de fork
pid= 48628, global= 7, var= 89
pid= 48627, global= 6, var= 88

En ambas salidas, podemos ver:

- que el PID más pequeño es el del padre y el PID más grande es el del hijo;
- el padre imprime su salida después del hijo (debido al sleep);
- el hijo imprime var y global distinto al padre.

¿Por qué global y var son distintos para hijo y padre?

El hijo y el padre tienen espacios de memoria separados, es decir, el hijo tiene su propio espacio de memoria que cuando se creó tenía lo mismo que el padre. La suma de global y var, se realiza en el espacio de memoria del hijo, lo ejecuta sólo el hijo, por lo que el valor de las variables incrementa sólo para el hijo.

Si el incremento lo hubiéramos hecho justo antes del último printf, lo habría ejecutado tanto el padre como el hijo, con lo que sí tendrían el mismo valor.

¿Qué diferencia hay entre comentar y descomentar (1)?

Hay 3 tipos de asignación de buffers:

- **sin buffer**, con lo que la información se escribe en la salida tan pronto como se escribe; es decir, si hay un printf en el programa cuando lo ejecuta lo muestra por pantalla;
- **con buffer de bloque**, se guarda y se escribe muchos caracteres como un bloque; es decir, si hay un printf en el programa, cuando se haya llenado el buffer de tamaño X, es cuando se imprime por pantalla (normalmente, los ficheros son de este tipo);
- **con buffer de línea,** los caracteres se almacenan en el buffer hasta que hay un salto de línea y se imprime.

La función setvbuf(STDOUT, NULL, _IONBF, 0), estamos especificando que:

int setvbuf(FILE *flujo, char *buf, int modo , size_t tam);

- STDOUT: la salida sea la salida estándar;
- NULL: indica que sólo el modo se ve afectado;
- _IONBF: el tipo de asignación es sin buffer;
- tam: el tamaño del buffer.

Con esto, podemos ver que:

- en la primera ejecución, con (1) comentado, el tipo es de bloque (por defecto), por ello se imprime dos veces lo de “Mensaje previo a la ejecución de fork”;
- en la segunda ejecución con (1) descomentado, establecemos el tipo a sin buffer con setvbuf, con lo que escribe “Mensaje previo a la ejecución de fork” una única vez, ya que escribe en la salida tal cual sucede.

1.5 Ejercicio 3

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)       //si es el padre (childpid != 0), deja de ejecutar el bucle
                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)      //si es el hijo (childpid = 0), deja de ejecutar el bucle
                break;
}
```

Salidas, con nprocs = 5:

Jerarquía 1:

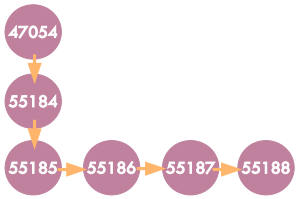
Proceso inicial: 55184 Padre: 47054
Proceso: 55184 Padre: 47054
Proceso: 55185 Padre: 55184
Proceso: 55186 Padre: 55185
Proceso: 55187 Padre: 55186
Proceso: 55188 Padre: 55187

Jerarquía 2:

Proceso inicial: 55162 Padre: 47054
Proceso: 55163 Padre: 55162
Proceso: 55162 Padre: 47054
Proceso: 55165 Padre: 55162
Proceso: 55164 Padre: 55162
Proceso: 55166 Padre: 55162

En la jerarquía 1:

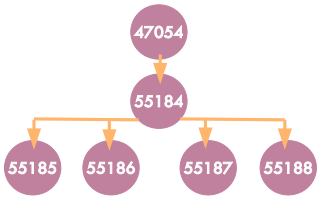
- un proceso (1) inicial crea un proceso (2) hijo con fork;
- el proceso padre (1) sale del bucle for con el if break;
- el proceso (2) hijo ejecuta el bucle y crea un proceso (3) hijo;
- el proceso (2) padre sale del bucle con fork con el if break;
- se repite así hasta que se han creado 5 procesos.



Por tanto, esta jerarquía crea un proceso (1), y dicho proceso (1) crea otro proceso (2), y este proceso (2) crea otro proceso (3) y así hasta llegar a nprocs procesos creados.

En la jerarquía 2:

- el proceso (1) inicial crea un proceso (2) hijo con fork;
- el proceso (2) hijo sale del bucle for con el if break;
- el proceso (1) vuelve al bucle y crea otro proceso (3) hijo;
- el proceso (2) hijo sale del bucle for con el if break;



Por tanto, esta jerarquía crea nprocs-1 procesos con el mismo padre, el proceso inicial (55184), excepto el proceso inicial, cuyo padre es otro distinto.

1.6 wait y waitpid

Estas llamadas suspenden la ejecución del proceso actual hasta que cambia el estado de otro proceso.

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

- **pid:** indica el pid del proceso. El valor de este pid puede ser:
  - **< -1**: espera a cualquier proceso hijo cuyo ID del proceso es igual al valor absoluto de pid.
  - **-1:** espera por cualquier proceso hijo; este es el mismo comportamiento que tiene **wait**.
  - **0:** espera por cualquier proceso hijo cuyo ID es igual al del proceso que llama.
  - **> 0:** lo que significa que espera por el proceso hijo cuyo ID es igual al valor de pid.

- **wstatus:** indica dónde se almacena la información de estado. El estado puede ser evaluado con las siguientes macros:
  - **WIFEXITED**(status): es distinto de cero si el hijo terminó normalmente.
  - **WEXITSTATUS**(status): evalúa los ocho bits menos significativos del código de retorno del hijo que terminó, que podrían estar activados como el argumento de una llamada a exit() o como el argumento de un return en el programa principal. Esta macro solamente puede ser tenida en cuenta si WIFEXITED devuelve un valor distinto de cero.
  - **WIFSIGNALED**(status): devuelve true si el proceso hijo terminó a causa de una señal no capturada.
  - **WTERMSIG**(status): devuelve el número de la señal que provocó la muerte del proceso hijo. Esta macro sólo puede ser evaluada si WIFSIGNALED devolvió un valor distinto de cero.
  - **WIFSTOPPED**(status): devuelve true si el proceso hijo que provocó el retorno está actualmente pardo; esto solamente es posible si la llamada se hizo usando WUNTRACED o cuando el hijo está siendo rastreado.
  - **WSTOPSIG**(status): devuelve el número de la señal que provocó la parada del hijo. Esta macro solamente puede ser evaluada si WIFSTOPPED devolvió un valor distinto de cero.

- **options:** el valor de options es un OR de cero o más de las siguientes constantes:
  - **WNOHANG**: que significa que vuelve inmediatamente si ningún hijo ha terminado.
  - **WUNTRACED**: que significa que también vuelve si hay hijos parados (pero no rastreados), y de cuyo estado no ha recibido notificación. El estado para los hijos rastreados que están parados también se proporciona sin esta opción.

- devuelve:
  - el ID del proceso del hijo que terminó;
  - cero si se utilizó **WNOHANG** y no hay hijos;
  - -1 en caso de error (en este caso, errno se pone a un valor apropiado).

1.7 exit

La llamada exit provoca la finalización normal de un proceso.

```
void exit(int status);
```

- **status:**
- devuelve al padre el valor de status & 0xFF.

Antes de finalizar el proceso, se llama a todas las funciones registradas con atexit() y on_exit() en orden inverso a su registro, y todos los flujos abiertos se vuelcan y cierran. Los ficheros creados por tmpfile() son eliminados.

1.8 Ejercicio 4

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

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

int main(){
	pid_t pid;	int estado;	const int HIJOS = 5;

	//creación de los hijos
	for(int i = 0; i < HIJOS; i++){
		//si ha habido error, salimos del programa
		if((pid = fork()) < 0){
			perror("ERROR al crear hijo\n");
			exit(-1);
		}
		//si se ha creado el proceso y es el hijo el que ejecuta
		else if(pid == 0){
			//muestra su PID por pantalla
			printf("\nSoy el hijo %i\n", getpid());
			exit(0);  //se finaliza el proceso hijo
		}
	}

	//el padre espera a los hijos
	for(int i = 0; i < HIJOS; i++){
		//espero a un proceso hijo
		pid = wait(&estado);
		printf("\nAcaba de finalizar mi hijo %i\n", pid);
		printf("Me quedan %i hijos vivos\n", HIJOS - i - 1);
	}
	return 0;
}
```

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).

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

int main(){
	const int HIJOS = 5;
	pid_t pid, hijos[HIJOS];
	int estado, h = 4;

	//creación de los hijos
	for(int i = 0; i < HIJOS; i++){
		//si ha habido error, salimos del programa
		if((pid = fork()) < 0){
			perror("ERROR al crear hijo\n");
			exit(-1);
		}
		//si se ha creado el proceso y es el hijo el que ejecuta
		else if(pid == 0){
			//muestra su PID por pantalla
			printf("\nSoy el hijo %i\n", getpid());
			exit(0); //se finaliza el proceso hijo
		}
		//si es el padre el que ejecuta, guarda en el array, el pid del hijo creado
		else{
			hijos[i] = pid;
		}
	}

	//el padre espera a los hijos creados en orden impar
	for(int i = 0; i < HIJOS; i += 2){
		//el padre espera al proceso de pid hijos[i]
		pid = waitpid(hijos[i], &estado, 0);

		printf("\nAcaba de finalizar mi hijo %i\n", pid);
		printf("Me quedan %i hijos vivos\n", h--);
	}

	//el padre espera a los hijos creados en orden     par
	for(int i = 1; i < HIJOS; i += 2){
		//el padre espera al proceso de pid hijos[i]
		pid = waitpid(hijos[i], &estado, 0);

		printf("\nAcaba de finalizar mi hijo %i\n", pid);
		printf("Me quedan %i hijos vivos\n", h--);
	}
	return 0;
}
```

2. Familia de llamadas al sistema exec

Estas llamadas ejecutan un programa distinto en el proceso hijo al que está ejecutando el padre, es decir, el padre crea un nuevo hijo que va a ejecutar otro programa distinto. Con esta llamada, el espacio de direcciones del proceso hijo se reemplaza completamente por un nuevo espacio de direcciones.

```
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[ ]);
```

- **path y file:** es el camino del archivo ejecutable;
- **arg:** los argumentos de la función;
  - el **primer argumento** debe apuntar al **nombre** del archivo asociado con el programa que se esté ejecutando;
  - el **último argumento** debe ser **NULL**, indicando el fin de argumentos;
- **argv:** un vector de punteros a cadenas de caracteres terminadas en 0, que representan la lista de argumentos; es el arg de las funciones execv y execvp.
- **envp**: especifica el entorno del proceso que ejecutará el programa. Este parámetro va después del NULL de los argumentos.

  En las otras funciones, este entorno se especifica en la variable externa **environ;**
- esta llamada sólo devuelve algo si ha fallado. En tal caso, devuelve un -1 y cambia errno.

Algunas particularidades:

- Las funciones **execlp** y **execvp** actuarán como un **shell** si la ruta especificada no es un nombre de camino absoluto o relativo. Si la variable file no está especificada, se emplea *:/bin:/usr/bin* por defecto.
- Si un archivo se le deniega el permiso (execve devuelve EACCESS), estas funciones buscarán en el resto de la lista de búsqueda. Si no encuentran otro archivo, devolverá EACCESS.
- Si no se reconoce la cabecera de un fichero (execve devuelve ENOEXEC), estas funciones ejecutarán el shell con el camino del fichero como su primer argumento. Si esto falla, no se busca más.
- Estas funciones fallarán si:
  - no tiene recursos suficientes para crear un nuevo espacio de direcciones de usuario;
  - si se utiliza de forma errónea el paso de argumentos.

Ejercicio 6. ¿Qué hace el siguiente programa?

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

int main(int argc, char *argv[]){
        pid_t pid;
        int estado;

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

La salida al ejecutar esto es:

```
./tarea5:
        linux-vdso.so.1 (0x00007ffea7d9d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fafb8279000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fafb8487000)
H=:
ldd: ./H=: No existe el archivo o el directorio
Mi hijo 63664 ha finalizado con el estado 1
```

Hay un error ya que al execl le falta como último argumento NULL. La salida con esto corregido es:

```
./tarea5
        linux-vdso.so.1 (0x00007fff91772000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1ffdad8000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1ffdce6000)
        Mi hijo 63712 ha finalizado con el estado 0
```

La orden ldd muestra las dependencias de bibliotecas compartidas, con lo que el hijo muestra las 3 primeras líneas, que son las bibliotecas compartidas requeridas por tarea5 y el padre ejecuta la última línea, en la que muestra el PID del hijo y que ha terminado con éxito.

2.2 Ejercicio 7

Ejercicio 7. Escribe un programa que acepte como argumentos el nombre de un programa, sus argumentos si los tiene, y opcionalmente la cadena "bg". Nuestro 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.

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

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

        pid_t pid;                  //para fork
        int estado, i, j, tam = 0, bcg = 0;

        //nos creamos un array con argc elementos que será el que tenga los argumentos para execv
        char *nombre;

        //comprobamos que hay argumentos
        if(argc < 2){
                //si no los hay, devolvemos error
                perror("ERROR argumentos.\n El formato es ./ejercicio7 <fichero/ruta> <argumentos>.\n");
                exit(-1);
        }

        //el nombre del fichero es el pasado en argv[1]
        nombre = argv[1];

        //si el último argumento es bg, hay que establecer bcg a true
        if(strcmp(argv[argc-1], "bg") == 0){
                //actualizamos el booleano de background a true
                bcg = 1;

                //establecemos el tamaño a argc - 3, ya que nuestros argumentos son todo argv de
                //tamaño argc menos argv[0] - el ./ejercicio7, argv[1] el nombre, argv[argc-1] - el bg
                tam = argc - 3;
        }
        //si no está especificado el background
        else{
                //el tamaño es argc - 1, sin el argv[0] ni argv[1]
                tam = argc - 2;
        }

        //argumentos tendrá tamaño tam y contendrá los argumentos para exec
        char *argumentos[tam];

        //guardamos los argumentos en el array
        for(i = 2, j = 0; i < argc && j < tam; i++, j++){
                argumentos[j] = argv[i];
        }
```

```
        //el último argumento lo ponemos a un puntero nulo
        argumentos[j] = (char*)0;

        //si hay que ejecutar en background (bcg true)
        //creamos un hijo que ejecute la orden, con lo que estaría ejecutando en background
        if(bcg){

                //creamos un hijo y comprobamos si ha habido error
                if((pid = fork()) < 0){
                        perror("ERROR en la creación del hijo.\n");
                        exit(-2);
                }
                //si el que ejecuta es el hijo
                else if(pid == 0){

                        //el hijo ejecuta la orden y si hay error, imprime
                        if(execv(nombre, argumentos) < 0){

                                //si ha devuelto un número negativo, ha habido error
                                perror("Return no esperado.\n");
                                exit(EXIT_FAILURE);
                        }
                }

                //el padre espera al hijo
                pid = wait(&estado);
                printf("\nEl hijo %d ha ejecutado el programa en background.\n", pid);
        }
        //si hay que ejecutarlo en background, será el padre el que ejecute el programa
        else{
                //el padre ejecuta la orden
                if(execv(nombre, argumentos) < 0){
                        //si ha devuelto un número negativo, ha habido error
                        perror("Return no esperado.\n");
                        exit(EXIT_FAILURE);
                }
        }

        return 0;
}
```

3. La familia clone

Crea un nuevo proceso hijo. Consideramos **tid** como identificador de hebra de un proceso.

```
int clone (int (*func) (void *), void *child_stack, int flags, void func_arg, ...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid*/);
```

- **(*func):** la función que va a ejecutar el proceso hijo;
- ***child_stack:** el puntero a la pila que debe utilizar el hijo y que previamente se ha reservado;
- **flags:** son los indicadores de clonación:

CLONE_CHILD CLEARTID	Limpia tid (thread identifier) 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
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

  Estos indicadores 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.

- **func_arg:** son los argumentos que se le pasan a func;

Cuando invocamos a clone:

- se crea un nuevo proceso hijo que comienza ejecutando la función dada por func;
- el hijo finaliza cuando func retorna o cuando haga un exit o _exit;

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 la tabla siguiente.

4. Preguntas de repaso

1. **¿Cómo podemos comprobar que el UID que dice el programa en C es efectivamente el del usuario que ejecuta el programa cuando hacemos getuid?**

```
int main(int argc, char *arg[]){
    uid_t id_usuario;
    //obtenemos el ID del usuario
    id_usuario = getuid();
    print("ID del usuario: %d\n", id_usuario);
}
```

El UID que muestra por pantalla es el identificador de usuario real. Suponiendo que el usuario se llama "prueba", podemos ver que se corresponde el UID con el usuario de varias formas:

1. $ cat /etc/passwd | grep prueba

   mirando en /etc/passwd y comprobando si el número UID aportado por el programa coincide con el de prueba del archivo /etc/passwd.

2. $ id prueba

   esta orden nos muestra información del usuario prueba y podemos comprobar que el número UID aportado por el programa coincide con el de prueba.

2. **¿Qué pasaría si en el siguiente trozo de código el fork lo ejecutamos antes del getpid()?**

```
int main(int argc, char *argv[]){
    pid_t id_proceso;
    pid_t res_fork;

    id_proceso = getpid();
    res_fork = fork();

    //Error
    if (res_fork < 0){
        perror("ERROR");
    }
    else if (res_fork != 0){
        //padre
        printf("Soy el padre %d \n", id_proceso);
    }
    else{
        //el hijo
        printf("Soy el hijo %d \n", id_proceso);
    }
}
```

En el código del enunciado sin modificación, tanto el padre como el hijo muestran el mismo PID. Esto se debe a que el getpid() lo ha ejecutado el padre antes del fork, por lo que cuando se ejecuta el fork, se hace una copia de lo que ya hay en el hijo, por ello, id_proceso contiene el PID del padre.

En el código con la modificación del fork arriba del getpid(), una vez se hace el fork, tanto el padre como el hijo realizan el getpid(), obtienen sus propios PIDs y los muestran por pantalla, cada uno el suyo.

3. **¿Cuánto valdría res_fork en el ejercicio anterior en los dos escenarios?**

   En ambos casos, el res_fork:

   - valdría 0 para el proceso hijo;
   - valdría el PID del proceso hijo para el padre.

4. **¿Cómo se podría implementar que un proceso padre espere su ejecución a la terminación de un proceso aleatorio?**

   1. wait(NULL);
      escribiendo la orden anterior en el padre.
   2. pid =wait(&estado);
      donde estado es un entero; wait devuelve el pid del proceso que ha terminado.
   3. waitpid(-1, &estado, 0);
      escribiendo la orden anterior en el padre.