Sistemas Operativos

Práctica 1.2

Ejecución de procesos

EJECUCIÓN DE PROCESOS

LLAMADA EXEC

```
int execl (char *path, char *arg0, char *arg1,..., NULL);
int execv (char *path, char *argv[]);
int execle(char *path, char *arg0, char *arg1,..., NULL, char *env[]);
int execve (char *path, char *argv[], char *env[]);
int execlp(char *file, char *arg0, char *arg1,..., NULL);
int execvp(char *file, char *argv[]);
```

Los procesos Unix pueden modificarse usando la llamada al sistema `exec`. A diferencia de `fork` (donde a la vuelta de la llamada existen dos procesos: el invocador o padre y el generado o hijo), la llamada `exec` produce la sustitución del programa invocador por el nuevo programa invocado. La ejecución del proceso continúa siguiendo el nuevo programa.

`fork` crea nuevos procesos, `exec` sustituye la imagen de memoria del proceso por otra nueva (sustituye todos los elementos del proceso: código del programa, datos, pila, montículo).

Las opciones de las distintas versiones de *exec* se pueden entender a través de las letras añadidas a `exec` en el nombre de las llamadas, como podemos ver en la siguiente tabla.

Letra	Significado
l	Los argumentos pasados al programa se especifican como una lista parámetros de tipo cadena, terminada con un parámetro NULL (que indica que la lista ha terminado).
v	Los argumentos pasados al programa se especifican como un vector de punteros a carácter (cadenas), terminado con un puntero nulo (que indica que el vector ha terminado).
p	Indica que para encontrar el programa ejecutable hay que utilizar la variable PATH definida, siguiendo el algoritmo de búsqueda de UNIX. Por lo tanto no es necesario especificar una vía absoluta al archivo, sino únicamente el nombre del ejecutable.
e	Indica que las variables de entorno que utilizará el programa serán las que se indique en el parámetro correspondiente mediante un vector de punteros a carácter (cadenas) terminadas en un puntero nulo, y no las variables que utiliza el proceso actual.

El nuevo programa activado mantiene el mismo PID así como otras propiedades asociadas al proceso. Sin embargo, el tamaño de memoria de la imagen del proceso cambia dado que el programa en ejecución es diferente.

Por ejemplo, si el proceso padre quiere ejecutar el comando `ls`

```
main() {
   execlp("ls", "ls", "-al", "*.c", NULL);
   perror("Error al ejecutar comando");
}
```

Si por el contrario el padre crea un hijo para que ejecute el comando `ls`

```
main() {
   printf("Este es el listado de archivos\n");
   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                execlp("ls","ls","-la",NULL);
                perror("Error en exec");
                exit(-1);
      default : /* PADRE */
                wait(&statusp)
                printf("Esto ha sido todo\n");
                exit(0);
   }
}
```

También podemos crear un programa y llamarlo desde otro. El código fuente del primer programa a ejecutar, `prog1.c`, es el siguiente:

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

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

     printf ("\nEjecutando el programa invocador (prog1). Sus argumentos
son: \n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);

     sleep( 10 );
     strcpy (argv[0],"prog2");
     if (execvp ("./prog2", argv) < 0) {
          printf ("Error en la invocacion a prog2 \n");
          exit (1);
          };
     exit (0);
}
```

El código fuente del segundo programa a ejecutar, `prog2.c`, es el siguiente:

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

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

     printf ("Ejecutando el programa invocado (prog2). Sus argumentos son:
\n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);
     sleep(10);
     exit (0);
}
```

Será necesario compilar ambos programas, usando la orden `gcc`. Tras ello, se estará en condición de ejecutarlos:

```
$ gcc -o prog2 prog2.c
$ gcc -o prog1 prog1.c
```

Para ejecutar el programa en modo background y así poder verificar su ejecución con la orden `ps`, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ prog1 p1 p2 param3 &
Ejecutando el programa invocador (prog1). Sus argumentos son:
    argv[0] : prog1
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps -f
UID  PID PPID   C    STIME    TTY  TIME CMD
 alu 8736 9176   0 10:23:32  pts/0  0:00 prog1 p1 p2 param3
 alu 9176 6328   1 10:21:21  pts/0  0:00 -ksh
 alu 9296 9176   7 10:23:36  pts/0  0:00 ps –f
$
Ejecutando el programa invocado (prog2). Sus argumentos son:
    argv[0] : prog2
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps –f
     UID  PID PPID   C    STIME    TTY  TIME CMD
 alu 8736 9176   0 10:23:32  pts/0  0:00 prog2 p1 p2 param3
 alu 9176 6328   1 10:21:21  pts/0  0:00 -ksh
 alu 9298 9176   5 10:23:46  pts/0  0:00 ps -f
$
```

SEÑALES

El siguiente código programa al manejador de reloj mediante la llamada al sistema `alarm()` para que nos avise dentro de 5 segundos enviándonos la señal `SIGALRM`. Una vez recibida, se salta a la rutina de servicio alarma que imprime un mensaje y continúa la ejecución normal del proceso.

```
#include <signal.h>

void alarma()
{
   printf("acabo de recibir un SIGALRM\n");
}

main()
{
     signal(SIGALRM,alarma);
     printf("Acabo de programar la captura de un SIGALRM\n");
     alarm(3);
     printf("Ahora he programado la alarma en 3 seg.\n");
     pause();
     printf("Ahora continúo con la ejecución normal\n");
}
```

El sistema tiene por defecto un par de manejadores para el tratamiento de todas y cada una de las señales: SIG_DFL y SIG_IGN. El primero de ellos, llama al manejador por defecto asociado a la señal el cual realiza un exit con el número de señal que recibe, mientras que el segundo de ellos, ignora la señal que llega.

LLAMADA SIGNAL

`void (*signal(int numero, void(*funcion)(int)))(int)`

La llamada prepara a un proceso para recibir una señal, asignando una función manejadora (*handler*) a un tipo de señal. `numero` es el tipo de señal y `funcion` es una función de tipo `void` y con un parámetro entero. Devuelve un puntero a la antigua función manejadora de esa señal en caso de éxito, y `SIG_ERR` en caso de error.

Cuando un proceso recibe una señal, su ejecución se detiene y se pasa a ejecutar un manejador, que consiste en una función que se ejecuta cuando se recibe esa señal. Cuando esta función termina, el proceso continúa ejecutándose por el mismo punto en que se quedó.

Existen diversas señales, identificadas por un número entero. Cada una de ellas tiene un significado particular. Mediante el archivo <signal.h> o bien por medio del comando `kill -l` se pueden identificar las señales del sistema. Las más importantes son las siguientes:

Nombre	Número	Significado
SIGINT	2	Se ha pulsado CRTL-C

Nombre	Número	Significado
SIGQUIT	3	Se ha pulsado CRTL-/
SIGILL	4	Se ha intentado ejecutar una instrucción no válida.
SIGTRAP	5	Se ha producido una interrupción de traza.
SIGFPE	8	Se ha producido un error en una operación de punto flotante.
SIGKILL	9	Matar proceso.
SIGUSR1	10	Señal de usuario. Libre para ser reprogramada.
SIGUSR2	12	Señal de usuario. Libre para ser reprogramada.
SIGPIPE	13	Utilización de una *PIPE* que no tiene ningún proceso en el otro extremo.
SIGALRM	14	Vencimiento de alarma programada.
SIGTERM	15	Señal de terminación enviada por el sistema
SIGCHLD	17	Señal que envía un hijo a su padre cuando finaliza

Si un proceso no ha definido explícitamente una función manejadora para una determinada señal, al recibirse esa señal el proceso muere. Por tanto, antes de esperar a recibir una señal debe definirse su manejador. Sin embargo, la señal `SIGKILL` no puede ser manejada. Así, si se recibe, matará al proceso incondicionalmente.

La función manejadora es una función de tipo `void` con un parámetro de tipo entero por el cual se indica el número de la señal recibida, de manera que el mismo manejador pueda valer para distintas señales. Existen dos funciones estándar que pueden programarse como manejadores de cualquier señal (excepto `SIGKILL`):

- `SIG_DFL`: es la rutina por defecto, que mata al proceso.
- `SIG_IGN`: que hace que se ignore la señal (no hace nada, pero no mata al proceso).

La programación de una función manejadora sólo sirve una vez. Después de ejecutarse el manejador, se vuelve automáticamente a programar la función por defecto `SIG_DFL`, por lo que si queremos que sirva para más veces debemos reprogramar el manejador dentro del propio manejador. Esto ocurre en el estándar POSIX. El siguiente ejemplo lo ilustra:

```
#include <signal.h>
void alarma() {
   printf("acabo de recibir un SIGALRM\n");
}
main(){
     signal(SIGALRM,alarma);
     printf("acabo de programar la captura de un SIGALRM\n");
     alarm(3);
     printf("Ahora he programado la alarma en 3 seg.\n");
     pause();
     printf("vuelvo a programar la alarma\n");
     alarm(3);
     pause();
     printf("En POSIX esta línea nunca se ejecutaría porque me ha matado
el SIGALRM\n");
}
```

Al crearse procesos mediante `fork()` la programación de señales en el hijo se hereda de tal forma que las señales programadas con `SIG_IGN` y `SIG_DFL` se conservan y las programadas con funciones propias pasan a ser `SIG_DFL`.

La recepción de señales no se hace necesariamente en el mismo orden en que se envían. Además, no existe una memoria de señales, de manera que si llegan varias señales (iguales) a la vez, sólo se ejecutará el manejador una vez.

LLAMADA KILL

`int kill(int PID, int numero)`

La llamada `kill` permite a un proceso enviar una señal a otro proceso o así mismo a través del PID. `PID` es el PID del proceso señalado y `numero` es el tipo de señal enviada. Devuelve `−1` en caso de error y `0` en caso contrario.

Existe una limitación sobre los procesos a los que pueden enviarse señales: deben tener alguna relación de parentesco del tipo: antecesor, hermano o hijo.

Por ejemplo, el siguiente código sirve para que un proceso se suicide:

```
#include <signal.h>

main() {
   printf("Voy a suicidarme\n");
   kill(getpid(), SIGKILL);
   perror("No he muerto???");
}
```

LLAMADA ALARM

`unsigned alarm(unsigned tiempo)`

La llamada `alarm` programa una alarma, de manera que cuando hayan transcurrido `tiempo` segundos, se enviará al proceso una señal de tipo `SIGALRM`. Devuelve `0` si no había otra alarma previamente programada, o el `número de segundos` que faltaban para cumplirse el tiempo de otra alarma previamente programada.

Sólo es posible programar una alarma por proceso. Al programarse una alarma se desconecta cualquier otra que hubiera previamente.

Un ejemplo de uso de `alarm` es el siguiente: poder imprimir en el terminal durante 5 segundos.

```
#include <signal.h>

int seguir = 1; /* Variable global */

void fin(int n) {
   seguir = 0;
}

main() {
   int contador = 0;
   signal(SIGALRM, fin);
   alarm(5);
   do {
      printf("Esta es la línea %d\n", contador++);
   } while (seguir);
   printf("TOTAL: %d líneas\n", contador);
}
```

LLAMADA PAUSE

int pause()

La llamada `pause` detiene la ejecución de un proceso (mediante espera pasiva) hasta que se reciba alguna señal. Devuelve siempre `-1` y establece el error *Interrupted system call*.

Este ejemplo crea un hijo y espera su señal.

```
#include <signal.h>
main() {
   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                printf("Hola, soy el hijo. Espero 2 segundos...\n");
                sleep(2);
                kill(getppid(),SIGUSR1);
                printf("Soy el hijo. He señalado a mi padre. Adios.\n");
                exit(0);
      default : /* PADRE */
                printf("Hola, soy el padre y voy a esperar.\n");
                signal(SIGUSR1, SIG_IGN); /* Ignoro señal para no morir */
                pause();
                printf("Soy el padre y ya he recibido la señal.\n");
                exit(0);
    }
}
```

Una señal muy utilizada es `SIGCHLD` que es enviada por todo proceso hijo a su padre en el mismo instante que realiza `exit`. De ésta manera, el padre sabe que su hijo ha pedido terminar.

Por ejemplo, el siguiente programa, una vez convertido en proceso, tiene un hijo que realiza `exit(5)`. El padre captura el 5.

```
#include <signal.h>
int status,pid;
void finhijo(){
   pid=wait(&status);
}

main(){
     signal(SIGCHLD,finhijo);
     if (fork()==0) {sleep(3); exit(5);}
     pause();
     printf("mi hijo ha muerto con estado %d\n",status/256);
     printf("ahora continúo con la ejecución\n");
}
```

REFERENCIAS

BIBLIOGRAFÍA

- F. M. Márquez: UNIX. Programación avanzada, Rama, 2004.
- K. A. Robbins y S. Robbins: UNIX. Programación práctica, Prentice may, 2000.