PRÁCTICA 2
CREACIÓN Y EJECUCIÓN DE PROCESOS

Introducción

Un proceso es un programa en ejecución que puede ejecutarse concurrentemente con otros procesos. El sistema operativo Unix es un sistema multitarea y multiusuario que permite a los usuarios crear varios procesos simultáneos, los cuales pueden sincronizarse, comunicarse y compartir información.

En esta práctica se aprenderá a utilizar las llamadas al sistema que nos permiten la creación y ejecución de procesos. En las siguientes prácticas, se estudiará la forma de conseguir la cooperación entre varios procesos.

Conceptos básicos

La única forma que tiene el sistema para identificar un proceso es mediante un identificador de proceso (PID), que es único en el sistema. Asociado con cada proceso se encuentra el *contexto del proceso*. El contexto de un proceso está formado por el contenido de su espacio de direccionamiento, el contenido de los registros hardware y las estructuras de datos del kernel asociadas con el proceso. Más formalmente, el contexto de un proceso es la unión del *contexto a nivel de usuario* (*user-level context*), el *contexto de registro* (*register context*) y el *contexto a nivel de sistema* (*system-level context*). El contexto a nivel de usuario consta del código, los datos y la pila, así como de la memoria compartida que pueda tener. El contexto de registro consta del contador de programa, el registro de estado del procesador, los punteros de la pila y los registros de propósito general. El contexto a nivel de sistema consta de la entrada en la tabla de procesos, la u-area (formada por los archivos abiertos, directorio actual, ...), la pila del kernel, etc.

Algunas de las llamadas al sistema que nos permiten manipular los procesos son:

- **fork** Crea un nuevo proceso.
- **exec** Permite a un proceso ejecutar un nuevo programa.
- **wait** Permite a un proceso padre (parent process) sincronizar su ejecución con la llamada exit de un proceso hijo.
- **exit** Termina la ejecución de un proceso.

Creación de procesos

Para crear un nuevo proceso en el S.O. Unix se utiliza la llamada al sistema fork. Esta llamada hace que el proceso que la ejecuta se divida en dos procesos. Al proceso que ejecuta fork se le conoce como proceso padre (parent process) y al nuevo proceso creado se le llama proceso hijo (child process). La sintaxis de fork es:

```
int fork();
```

Tras ejecutarse esta llamada al sistema, los dos procesos tendrán copias idénticas de su contexto a nivel de usuario. La única diferencia será que el valor entero que devuelve fork para el proceso padre es el PID del proceso hijo, mientras que para el proceso hijo es 0. Estos valores se utilizan para determinar el código que ejecutará cada proceso, como se puede ver en los programas 1 y 2. En caso de error, la llamada al sistema devuelve el valor –1. Además, ambos procesos compartirán los archivos que el proceso padre tenía abiertos hasta el momento de la llamada al sistema. Y por último, el proceso hijo recién creado heredará los IDs de los usuarios (real y efectivo) del proceso padre, el grupo del proceso padre, el directorio actual y el valor nice del padre, que se utiliza para calcular la prioridad de planificación.

Todos los procesos del sistema, excepto el proceso 0, se crean mediante esta llamada al sistema.

El kernel asocia dos IDs de usuario con cada proceso (heredados del proceso padre, como se ha dicho anteriormente), independientemente del ID del proceso: el ID de usuario real (RUID) y el ID de usuario efectivo (EUID) o setuid (Set User ID). El ID de usuario real identifica al usuario responsable del proceso que se va a ejecutar mientras que el ID de usuario efectivo se obtiene cuando se hereda el bit setuid del modo de permisos de un archivo y consiste en que un usuario (sólo durante la ejecución del programa) tendrá los privilegios del propietario de un archivo ejecutable (se le conoce como programas setuid), como si el usuario que está ejecutando el programa fuera el propietario del archivo. Con esto conseguimos acceder a archivos de datos que pueda manejar el programa y para los cuales no se tenía permiso. Un ejemplo se tiene con la orden passwd, el cual accede a un archivo de datos (/etc/passwd) que no podría ser modificado directamente por un usuario normal.

El sistema impone (aunque es configurable) un límite en el número de procesos que un usuario puede ejecutar simultáneamente. De esta forma evitamos que se sature la tabla de procesos ya que podría impedir que otros usuarios crearan procesos. Estas limitaciones no afectan al superusuario.

Considere los programas 1 y 2. Estos programas permiten comprobar que el código ejecutará tanto el proceso padre como el hijo una vez ejecutada la llamada al sistema fork. Para que quede más claro se ha duplicado el código del programa que será justo lo que ocurra cuando se ejecute fork. Observe que hay partes del código que son ejecutadas exclusivamente por uno de los procesos y otras partes que pueden ser ejecutadas por ambos procesos.

Nota: Cuando se duplica un programa, los nombres de las variables, constantes, etc., siguen llamándose igual pero, en realidad, son distintas ya que se encuentran en zonas distintas de memoria; cada una correspondiente al área de datos de sus respectivos procesos.

Ejecución de procesos

La llamada al sistema *execve* ejecuta un programa, colocándose el código de éste en el espacio de memoria del proceso que ejecutó la llamada. Su sintaxis es la siguiente:

```
execve (filename, argv, envp);
char *filename, *argv[], *envp[];
```

donde *filename* es el nombre del archivo a ejecutar, *argv* es un puntero a una lista de cadenas que corresponderán con los parámetros del programa ejecutable (es una copia del parámetro que se utiliza en la función principal: *main (argc, argv)*) y *envp* es un puntero a una lista de punteros a cadenas que se corresponden con el entorno del programa a ejecutar. Las cadenas de caracteres del entorno son de la forma "name=value" y pueden contener información útil para los programas como puede ser el directorio de trabajo del usuario, caminos de directorios para buscar programas ejecutables, etc. Los procesos pueden también acceder a su entorno a través de la variable global *environ* (su declaración es de la forma siguiente: *extern char **environ*) inicializada por las rutinas del C.

Programa 1. Utilización de la llamada al sistema fork (proceso padre)

```
#include <fcntl.h>
#include <stdio.h>
main ()
{
        int pid, status;
        switch (pid=fork()) {
                case –1: /* Gestión del error */
                        exit (1);
                        break;
                case 0:  /* Proceso hijo */
                        printf ("\n Proceso hijo. PID = %d \n", pid);
                        break;
                default: /* Proceso padre */
                        printf ("\n Proceso padre. PID = %d \n", pid);
                        wait (&status); /* espera al final del proceso hijo */
                        break;
        } /* fin del switch */
        printf ("\n Fin de la ejecución \n");
        return 0;
} /* fin de la función main */
```

El proceso padre ejecuta primero la sección default del switch y después las instrucciones posteriores al fin del switch.

Programa 2. Utilización de la llamada al sistema fork (proceso hijo)

```
#include <fcntl.h>
#include <stdio.h>
main ()
{
        int pid, status;
        switch (pid=fork()) {
```

```
                    case –1: /* Gestión del error */
                             exit (1);
                             break;
                    case 0:  /* Proceso hijo */
                             printf ("\n Proceso hijo. PID = %d \n", pid);
                             break;
                    default: /* Proceso padre */
                             printf ("\n Proceso padre. PID = %d \n", pid);
                             wait (&status); /* espera al final del proceso hijo */
                             break;
          } /* fin del switch */
          printf ("\n Fin de la ejecución \n");
          return 0;
} /* fin de la función main */
```

El proceso hijo ejecuta primero la sección case 0 del switch y después las instrucciones posteriores al fin del switch.

Existen varias funciones en las librerías del lenguaje C que también permiten ejecutar programas como son *execl*, *execv*, *execle*, *execlp* y *execvp*. Se diferencian en el tipo de información que se le pasa como argumento. Estas funciones junto con la llamada al sistema *execve* forman la familia *exec*. Muchas de estas funciones son más sencillas de usar que la llamada al sistema, como se puede apreciar en el siguiente ejemplo.

En el programa 3 se puede ver un ejemplo de cómo se crea un proceso hijo que ejecutará un programa usando la función *execl*.

Programa 3. Utilización de la función execl

```
main ()
{
          int status;
          if (fork () == 0)
                    execl ("/bin/date", "date", 0);
          wait (&status);
} /* fin de la función main */
```

Espera y finalización de procesos

La llamada al sistema *wait* espera a que un proceso finalice. Normalmente, lo ejecuta un proceso padre para esperar a que un proceso hijo termine su ejecución. Su sintaxis es:

```
wait (estado);
int *estado;
```

Tras su ejecución, devuelve el PID del proceso hijo en cuestión y coloca en *estado* el estado de salida proporcionado por éste. Devuelve –1 si no existe ningún proceso hijo del proceso que envía la llamada. Podemos decir que la función *wait()* se utiliza para la sincronización de los dos procesos (padre e hijo) que se están ejecutando concurrentemente.

La llamada al sistema *exit()* provoca la finalización de un proceso. Los 8 bits menos significativos del argumento quedan a disposición del proceso padre.

Un proceso puede finalizar bien cuando ejecuta la función *exit()* o bien cuando termina la ejecución de su código.

Ejercicio

Crear un proceso que leerá por el terminal el nombre de un programa (orden de Unix) y seguidamente lo ejecutará. El programa a ejecutar debe residir en el directorio /bin y no debe necesitar ningún parámetro (p. ej., ls, date, time, cd, pwd, ...). No se debe de ejecutar un programa hasta que el anterior no haya acabado. El proceso terminará cuando se introduzca la cadena "salir".

Como ejercicio complementario, podemos modificar el programa anterior para permitir al usuario ejecutar programas que se encuentren en el directorio actual, o bien, si se desea mejorar esta variante, podemos permitir ejecutar cualquier programa que se encuentre en alguno de los caminos especificados en la variable de entorno *PATH*.