Práctica 5. Creación de procesos en *UNIX*

1.- Objetivos

- Utilizar las llamadas al sistema *UNIX* relacionadas con la identificación, creación y terminación de procesos.
- Utilizar las llamadas al sistema *UNIX* para terminar procesos y para conocer el estado en el que finalizan otros procesos.
- Escribir programas que den lugar a la ejecución de múltiples procesos de manera concurrente.
- Ejecutar programas diferentes desde un programa escrito en lenguaje *C*.

2.- Introducción

Cuando un ordenador arranca un sistema operativo *UNIX* establece una comunicación con el disco duro, donde reside un fichero llamado *vmunix* que contiene el núcleo (*Kernel*) del sistema operativo. Cuando la carga ha finalizado, el *kernel* queda almacenado en el sistema de memoria y el microprocesador está ejecutándolo. La ejecución del *kernel* supone la creación de un entorno en el que los usuarios pueden solicitar que se ejecuten otros programas. En resumen, el *kernel* de *UNIX* crea *contextos* en los que resulta posible ejecutar otros programas.

Una vez que se empieza a ejecutar el *kernel* se ponen en marcha algunos procesos (*init*, *swaper, pagedaemon*). Estos procesos pueden crear, a su vez, nuevos procesos hijos suyos, y así sucesivamente.

El concepto de proceso es más complejo que el de programa: un proceso es un programa que se está ejecutando, que realiza un conjunto de operaciones de entrada y salida, y que se encuentra en un estado determinado en un momento dado. En los sistemas operativos multitarea, como es el caso de *UNIX*, un único procesador puede ejecutar varios procesos de forma concurrente. Para ello se utilizan algoritmos de planificación que deciden cuándo se debe desalojar a un proceso de la *CPU* para asignársela a otro.

3.- Identificadores de proceso

En *UNIX*, cada proceso tiene asociado un número entero positivo desde el momento de su creación: el identificador de procesos (*pid*). Este valor lo distingue de todos los demás procesos que se ejecutan concurrentemente con él. Además, un proceso conoce el identificador de su proceso padre (*ppid*), que es el proceso que lo ha creado.

Cuando arranca el sistema se crea siempre un proceso especial cuyo *pid*es *0*. Este proceso, crea, a su vez, otro proceso de nombre *init* y *pid* igual a *1*. A continuación, este proceso crea al proceso *swapper,* cuya misión es gestionar la memoria virtual. El proceso *init* es el encargado de arrancar los demás procesos que se ejecuten en el sistema.

El *pid* de un proceso nunca cambia durante el tiempo de vida de éste; sin embargo, sí puede verse modificado el valor de su *ppid*: cuando el proceso padre muere, el proceso huérfano pasa a

considerarse como hijo del proceso *init*, entonces el valor de su *ppid* toma el valor *1* (que es el valor del *pid* del proceso *init*).

Para conocer el valor de *pid* de los procesos de un sistema se pude utilizar el comando *ps*, pero también interesa poder solicitar esta información desde un programa escrito en lenguaje *C*. Para obtener los identificadores de un proceso y de su padre, se dispone de dos llamadas al sistema: *getpid()* y *getppid()*:

```
#include <sys/types.h>  /* Archivos de cabecera necesarios para utilizar el */
#include <unistd.h>     /* tipo pid_t y las llamadas getpid() y getppid().  */

pid_t getpid();
pid_t getppid();
```

Las llamadas a estas funciones no fallan nunca.

Ejemplo:

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

int main(void) {
   pid_t pidproceso, pidpadre;
   pidproceso=getpid();
   pidpadre=getppid();
   printf("Soy el proceso de pid=%d\n", pidproceso);
   printf("Mi padre es el proceso de pid=%d\n", pidpadre);
   return 0;
}
```

Ejecutar el programa anterior desde una ventana de *shell* no es otra cosa que crear un proceso hijo del *shell.* Dicho proceso hijo se encargaría, en este caso, de mostrar por pantalla los identificadores del propio hijo (*getpid()*) y del proceso padre del mismo (*getppid()*), que es el propio *shell.*

4.- Creación de procesos

La única forma de crear un proceso en el sistema *UNIX* es mediante la llamada *fork*. El proceso que invoca a *fork* es el proceso padre y el proceso creado es el proceso hijo. La declaración de *fork* es la siguiente:

```
#include <sys/types.h>  /* Archivos de cabecera necesarios para utilizar */
#include <unistd.h>     /* la llamada fork().                             */

pid_t fork();
```

La llamada a *fork* hace que el proceso actual se duplique, es decir, después de ejecutar f*ork* se tienen dos procesos en el sistema en lugar de uno. Ambos tienen una copia idéntica del contexto del nivel de usuario excepto el valor devuelto por la función *fork*, que para el proceso padre toma el valor del *pid* del proceso hijo y para el proceso hijo toma el valor 0 y no corresponde con el valor real de su

pid. En el proceso *0*, creado por el *kernel* cuando arranca el sistema, es el único que no se crea mediante una llamada a *fork*.

Si la llamada a *fork* falla, devuelve el valor *-1* y en *errno* queda el código del error producido (es posible visualizar este código utilizando la instrucción *perror()*).

Ejemplo:

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

int main(void) {
   printf("Arrancando...\n");

   pid_t mipid;
   mipid=fork(); /* Creación de un proceso hijo */

   if (mipid==-1) /* Error en la llamada a fork() */
      perror("Error en la llamada a fork()");
   else if (mipid==0) {
      /* Código que ejecuta solamente el proceso HIJO */
      printf("Hijo: Mi pid es %d\n", getpid());
   } else {
      /* Código que ejecuta solamente el proceso PADRE */
      printf("Padre: mi pid es %d\n", getpid());
   }

   /* Código común a los procesos PADRE e HIJO */
   printf("Finalizando...\n");
   return 0;
}
```

5.- Terminación de procesos

La llamada *exit* termina la ejecución de un proceso y devuelve al sistema operativo un valor entero:

```
#include <stdlib.h>

void exit(int status);
```

Para destruir un proceso desde otro diferente se dispone de la llamada *kill*, que se encarga de enviar una señal a un proceso:

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

int kill(pid_t pid, int sig);
```

`pid` es el identificador del proceso al que se envía la señal, mientras que `sig` es el número de la señal que se envía. La señal *SIGTERM* (número 15) está definida en el archivo de cabecera *signal.h* y se utiliza para indicarle a un proceso que debe terminar su ejecución.

El valor entero que devuelve *kill* es *0* cuando se ejecuta correctamente. Si tiene algún error, devuelve el valor *-1* y en *errno* queda el código de error producido.

El siguiente ejemplo nos muestra un programa que nos permite eliminar a otro proceso cuyo *pid* se introduce desde el teclado. El programa finaliza devolviendo el valor *0* al sistema operativo (en general, el valor cero indica que se ha finalizado la ejecución errores):

Ejemplo:

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

int main(void) {
   pid_t otropid;
   printf("Por favor, introduce el pid del proceso que desea terminar: ");
   scanf("%d", &otropid);
   kill(otropid, SIGTERM); /* Mata al proceso de pid=otropid */
   exit(0); /* El proceso finaliza y devuelve el valor 0 al sistema operativo
*/
}
```

6.- Sincronización de procesos: *exit* y *wait*

Un caso típico de programación tiene lugar cuando un proceso hijo permanece a la espera de que termine otro proceso hijo suyo antes de continuar ejecutándose.

Un ejemplo de esta situación se tiene cuando se escribe una orden en la línea de comandos y se pulsa *intro*: El proceso *shell* crea un proceso hijo que se encarga de ejecutar la orden solicitada y no muestra el cursor (señal de espera por una nueva orden) hasta que no se ha ejecutado completamente. Como es lógico, esto no ocurre así cuando la orden se ejecuta en segundo plano.

Para poder sincronizar dos procesos padre e hijo, se emplean las llamadas *exit* y *wait*:

```
#include <stdlib.h>

void exit(int estado);
```

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

pid_t wait(int *estado);
```

Como ya hemos visto antes, *exit* termina la ejecución de un proceso y devuelve al sistema el valor de `estado`. Si el proceso padre del que ejecuta la llamada a *exit* está ejecutando una llamada a *wait*, se le notifica la terminación de su proceso hijo y se le envían los ocho bits menos significativos de *estado*. Con esta información, el proceso padre puede saber en qué condiciones ha terminado el proceso hijo.

Wait suspende la ejecución del proceso que la invoca hasta que alguno de sus procesos hijos finaliza. La forma de invocar a *wait* es:

```
pid_t pid;
int estado, edad;
pid=wait(&estado);
```

Se obtiene en `pid` el valor del identificador de proceso de alguno de los hijos *zombies* (un proceso *zombie* es el que acaba de terminar). En la variable `estado` se almacena el valor que el proceso hijo le envía al padre mediante la llamada a *exit* y que da idea de la condición d.e finalización del proceso hijo. Si se quiere ignorar este valor, se puede pasar como parámetro el puntero *NULL* (`pid=wait(NULL)`).

En el archivo de cabecera *sys/wait.h* se definen diferentes macros que permiten analizar el valor de `estado` para determinar la causa de la terminación del proceso. En particular, el proceso padre puede obtener el valor de los ocho bits menos significativos del parámetro que recibe desde la llamada a *exit* por parte del hijo utilizando la macro *WEXITSTATUS(estado)*.

Si durante la llamada a *wait* se produce algún error, la función devuelve *-1* y en *errno* quedará el código del tipo de error producido.

Cómo es lógico, si el proceso que invoca a *wait* no tiene ningún proceso hijo vivo, se produce un error.

Por ejemplo, sea el siguiente código correspondiente a un proceso cuyo *pid* es igual a *10* y que tiene un hijo con *pid* igual a 20:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
...
int main (void) {
   pid_t pid;
   int estado, edad, edadHijo;
   ...
   pid=wait(&estado);
   edad=WEXITSTATUS(estado);
   printf("Mi hijo %d ha fallecido a los %d años.\n", pid, edad);
   pid=wait(&estado);
   printf("pid=%d\n", pid);
   ...
}
```

Si el código del proceso hijo termina con las siguientes instrucciones:

```
   ...
   edadHijo=140;
   exit(edadHijo);
}
```

Cuando el proceso padre llega a la primera instrucción *wait* suspende su ejecución y permanece a la espera de que finalice el proceso hijo. Éste finaliza cuando alcanza la instrucción *exit* que tiene como parámetro la variable `edadHijo`, cuyo valor es el 140 en el proceso hijo.

Es ahora cuando el proceso padre se desbloquea y la instrucción *exit* devuelve como resultado el valor del *pid* del proceso hijo que ha terminado (en este caso, el valor 20) y lo almacena en la variable `pid`.

En los ocho bits menos significativos de la variable `estado` queda almacenado el valor de los ocho bits menos significativos de la variable `edadHijo` (parámetro de la instrucción *exit* del proceso hijo). Como es posible que el resto de la variable `estado` tenga valores no deseados, se utiliza la macro *WEXITSTATUS* para obtener el valor correcto del parámetro enviado por el proceso hijo.

Así, tras finalizar el proceso hijo, el proceso padre mostraría en la pantalla el siguiente mensaje:

```
Mi hijo 20 ha fallecido a los 140 años.
```

Posteriormente, el padre pretende esperar a que termine otro proceso hijo (segundo *wait*). Puesto que ya no hay ningún hijo vivo, la llamada a *wait* produce un error y devuelve el valor *-1*. Entonces se vuelva por pantalla el siguiente mensaje:

```
pid=-1
```

7.- Ejecución de programas: *exec*

Existe toda una familia de funciones *exec* útiles para lanzar programas ejecutables desde un programa escrito en *C*. Dentro de esta familia cada función tiene su sintaxis pero todas tienen aspectos comunes y obedecen al mismo tipo de funcionamiento: Se carga un programa en la zona de memoria del proceso que ejecuta la llamada sobreescribiendo los segmentos del programa antiguo con los del nuevo. Es devir, *el programa viejo es SUSTITUIDO por el nuevo y NUNCA se volverá a él para proseguir su ejecución*, pues es el programa nuevo el que pasa a ejecutarse.

La declaración de la familia de funciones *exec* es la siguiente:

```
int execl(char *ruta, char *arg0, char *arg1,..., char *argn, (char *)0);

int execv(char *ruta, char *argv[]);

int execle(char *ruta, char *arg0, char *arg1,..., char *argn, (char *)0, char
*envp[]);

int execve(char *ruta, char *argv[], char *envp[]);

int execlp(char *fichero, char *arg0, char *arg1,..., char *argn, (char *)0);

int execvp(char *fichero, char *argv[]);
```

donde:

- `ruta` es una cadena con el *path* (absoluto o relativo) de un archivo ejecutable.
- `fichero` es el nombre un un fichero ejecutable.
- `arg0`, `arg1`, …, `argn` son cadena de caracteres que constituyen la lista de parámetros que se le pasa al nuevo programa. Por convenio, al menos el `arg0` está presente siempre y coincide con `ruta` o con el último componente de `ruta`. Hay que destacar que tras `argn` se pasa un

puntero *NULL* para indicar el final de los argumentos.

- `argv` es un array de cadenas de caracteres que constituye la lista de argumentos que va a recibir el nuevo programa. Por convenio, `argv` debe tener al menos un elemento, que coincide con ruta o con el último componente de ruta. El final de `argv` se indica colocando un puntero *NULL* detrás del último parámetro.
- `envp` es un array de punteros a cadenas de caracteres que constituyen el entorno en el que se va a ejecutar el nuevo programa. `envp` también termina con un puntero *NULL*.

Un programa escrito en *C* recibe estos parámetros a través de la función *main*, que se puede declarar de la siguiente forma:

```
int main(int argc, char *argv[], char *envp[]);
```

Así, por ejemplo, se podría obtener un listado completo de los archivos del directorio actual utilizando *execl*:

```
execl("/bin/ls", "ls", "-l", "*.c", (char *)0);
```

La instrucción anterior ejecuta la orden ls de igual manera que si escribiera en la línea de comandos la siguiente orden:

```
ls -l *.c
```

Si hubiera más instrucciones posteriores a a *execl* no se ejecutarían porque el programa ls sustituye al que lo llama.

En el caso de que se deseara ejecutar más instrucciones, sería necesario crear un proceso hijo que se encargara de llamar a la orden *execl* mientras el padre podría continuar realizando otras tareas.

8.- Desarrollo de la práctica

Siempre que entro en mi sesión me gusta ejecutar una serie de programas, y tenerlos siempre funcionando y a la vista, por si necesito trabajar con ellos.

Arrancarlos todos ellos, uno a uno, ya es suficientemente pesado; si además tengo que arrancar alguno de ellos otra vez si, por alguna razón, se muere, la tarea se vuelve tediosa.

En esta práctica vamos a crear un programa que me ayude en esa tarea. Debe hacer lo siguiente:

1. Al principio, tiene que crear 5 hijos.
2. Cada uno de esos cinco hijos debe ejecutar un programa diferente. Estos programas y sus argumentos serán los siguientes (sin las comillas):
    - "xload"
    - "xeyes"
    - "xlogo"
    - "xcalc"
    - "xclock -update 1"

3. Durante toda su ejecución debe esperar y, si alguno de sus hijos termina por alguna razón, volver a crearlo y hacer que ejecute el mismo programa que estaba ejecutando anteriormente. De esta manera siempre tendremos los mismos 5 programas funcionando simultáneamente.