Ejecución de procesos

Ejecución de procesos

La función main y los argumentos

Habitualmente, cuando se lanza un programa a ejecución desde el shell, se le añaden parámetros o argumentos para definir exactamente qué queremos de él. Por ejemplo:

```
vi pepe.txt

ls -l /usr/include

cp pepe.c ../pipo.c
```

En el primer caso se invoca al editor `vi` especificándose que se desea trabajar con el fichero "pepe.txt". El fichero es un parámetro pasado al shell. El segundo caso es una llamada al programa `ls` que incorpora dos parámetros, como ocurre en el último ejemplo.

¿Qué son los parámetros o argumentos? En principio, puede afirmarse que son conjuntos de caracteres separados por blancos (espacios o tabuladores). Ahora bien, no se consideran parámetros los redireccionamientos, y caracteres como el ampersand "&" o el punto y coma ";" actúan de separadores (en general, eso ocurre con todos los caracteres que tengan un significado para el intérprete de órdenes).

Ejemplo:En la orden

```
ls -l /usr/include & >pepe.txt
```

los parámetros son "ls", "-l" y "/usr/include" ('&' y la redirección no cuentan).

Cada programa recibe los parámetros a través de su punto de entrada, que en el caso del lenguaje C es la función `main`. El formato mínimo que acepta esta función en UNIX es

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

donde `argc` expresa cuántos parámetros se han reconocido, y `argv` es un

vector de cadenas de caracteres que precisamente contienen los parámetros, siendo **argv[i]** el parámetro i.

Los parámetros se empiezan a numerar en 0. El parámetro 0 es el nombre del programa invocado tal y como se pasó en la línea de órdenes. En el ejemplo de **vi pepe.txt**, los valores de **argc** y **argv** serían

argc = 2

argv[0] = "vi"

argv[1] = "pepe.txt"

Funciones para ejecución de programas

La función system()

La forma más sencilla de invocar una orden UNIX desde un programa en C es mediante la función **system**, que toma como único parámetro la orden que quieren ejecutar. Reconoce redirecciones, expresiones regulares, conductos (pipes), etc. Por ejemplo, la línea

```
system("ls -l /usr/include/*.h >pepe.txt")
```

ejecuta la cadena pasada como parámetro tal y como si la hubiéramos tecleado desde la consola. La función **system** se limita a lanzar un shell hijo pasándole como parámetro de entrada la cadena suministrada en la función.

La forma de más bajo nivel para ejecutar una orden consiste en lanzar a ejecución el programa deseado mediante alguna de las llamadas al sistema que empiezan por **exec**. Existen varias modalidades que difieren en la forma de pasar los parámetros al programa (aunque realmente se trata de una sola llamada al sistema UNIX).

Las llamadas exec...

El sistema operativo UNIX ofrece una llamada al sistema llamada 'exec' para lanzar a ejecución un programa, almacenado en forma de fichero. Aunque en el fondo sólo existe una llamada, las bibliotecas estándares del C disponen de varias funciones, todas comenzando por 'exec' que se diferencian en la manera en que se pasan parámetros al programa.

La versión típica cuando se conoce a priori el número de argumentos que se van a entregar al programa se denomina **execl**. Su sintaxis es

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

Es decir, el nombre del fichero y luego todos los argumentos consecutivamente, terminando con un puntero nulo (vale con un cero). Sirva este ejemplo:

Para ejecutar

```
/bin/ls -l /usr/include
```

escribiríamos

```
execl ( "/bin/ls", "ls", "-l", "/usr/include", 0 );
```

Obsérvese que el primer argumento coincide con el nombre del programa.

En caso de desconocer con anticipación el número de argumentos, habrá que emplear la función **execv**, que tiene este prototipo:

```
execv ( char* fichero, char* argv [] );
```

El parámetro ***argv*** es una tira de cadenas que representan los argumentos del programa lanzado, siendo la última cadena un nulo (un cero). El ejemplo anterior se resolvería así:

```
char* tira [] = { "ls", "-l", "/usr/include", 0 };

...

execv ( "/bin/ls", tira );
```

En los anteriores ejemplos se ha escrito el nombre completo del fichero para ejecutar ("/bin/ls" en vez de "ls" a secas). Esto es porque tanto **execl** como **execv** ignoran la variable **PATH**, que contiene las rutas de búsqueda. Para tener en cuenta esta variable pueden usarse las versiones **execlp** o **execvp**. Por ejemplo:

```
execvp ( "ls", tira );
```

ejecutaría el programa "/bin/ls", si es que la ruta "/bin" está definida en la variable **PATH**.

Todas las llamadas **exec...** retornan un valor no nulo si el programa no se ha podido ejecutar. En caso de que sí se pueda ejecutar el programa, se transfiere el control a éste y la llamada **exec...** nunca retorna. En cierta forma el programa que invoca un **exec** desaparece del mapa.

Procesos concurrentes: llamadas fork y wait

Para crear nuevos procesos, el UNIX dispone únicamente de una llamada al sistema, **fork**, sin ningún tipo de parámetros. Su prototipo es

```
int fork();
```

Al llamar a esta función se crea un nuevo proceso (proceso hijo), idéntico en código y datos al proceso que ha realizado la llamada (proceso padre). Los espacios de memoria del padre y el hijo son disjuntos, por lo que el proceso hijo es una copia idéntica del padre que a partir de ese momento sigue su vida separada, sin afectar a la memoria del padre; y viceversa.

Siendo más concretos, las variables del proceso padre son inicialmente las mismas que las del hijo. Pero si cualquiera de los dos procesos altera una variable, el cambio sólo repercute en su copia local. Padre e hijo no comparten memoria.

El punto del programa donde el proceso hijo comienza su ejecución es justo en el retorno de la función **fork**, al igual que ocurre con el padre.

Si el proceso hijo fuera un mero clon del padre, ambos ejecutarían las mismas instrucciones, lo que en la mayoría de los casos no tiene mucha utilidad. El UNIX permite distinguir si se es el proceso padre o el hijo por medio del valor de retorno de **fork**. Esta función devuelve un cero al proceso hijo, y el identificador de proceso (PID) del hijo al proceso padre. Como se garantiza que el PID siempre es no nulo, basta aplicar un **if** para determinar quién es el padre y quién el hijo para así ejecutar distinto código.

Con un pequeño ejemplo:

```
main()
{
 int x=1;
 if ( fork()==0 )
 {
                printf ("Soy el hijo, x=%d\n",x);
 } else {
                x=33;
                printf ("Soy el padre, x=%d\n",x);
 }
}
```

Este programa mostrará por la salida estándar las cadenas "Soy el hijo, x=1" y "Soy el padre, x=33", en un orden que dependerá del compilador y del planificador de procesos de la máquina donde se ejecute.

Como aplicación de **fork** a la ejecución de programas, véase este otro pequeño ejemplo, que además nos introducirá en nuevas herramientas:

```
1        if ( fork()==0 )
2        {
3                execlp ("ls","ls","-l","/usr/include",0);
4                printf ("Si ves esto, no se pudo ejecutar el \
4b                                              programa\n");
5                exit(1);
6        }
7        else
8        {
9                int tonta;
10              wait(&tonta);
11      }
```

Este fragmento de código lanza a ejecución la orden **ls -l /usr/include**, y espera por su terminación.

En la línea 1 se verifica si se es padre o hijo. Generalmente es el hijo el que toma la iniciativa de lanzar un fichero a ejecución, y en las líneas 2 a la 6 se

invoca a un programa con **execlp**. La línea 4 sólo se ejecutará si la llamada **execlp** no se ha podido cumplir. La llamada a **exit** garantiza que el hijo no se dedicará a hacer más tareas.

Las líneas de la 8 a la 11 forman el código que ejecutará el proceso padre, mientras el hijo anda a ejecutar sus cosas. Aparece una nueva llamada el sistema, la función **wait**. Esta función bloquea al proceso llamador hasta que alguno de sus hijos termina. Para nuestro ejemplo, dejará al padre bloqueado hasta que se ejecute el programa lanzado o se ejecute la línea 5, terminando en ambos casos el discurrir de su único hijo.

Es decir, la función **wait** es un mecanismo de sincronización entre un proceso padre y sus hijos.

La llamada **wait** recibe como parámetro un puntero a entero donde se deposita el valor devuelto por el proceso hijo al terminar; y retorna el PID del hijo. El PID del hijo es una información que puede ser útil cuando se han lanzado varios procesos hijos y se desea discriminar quién exactamente ha terminado. En nustro ejemplo no hace falta este valor, porque sólo hay un hijo por el que esperar.

Como ejemplo final, esta función en C es una implementación sencilla de la función **system**, haciendo uso de **fork**, **wait** y una función **exec**.

```
int system ( const char* str )
{
        int ret;
        if (!fork())
        {
         execlp ( "sh", "sh", "-c", str, 0 );
         return -1;
        }
        wait (&ret);
        return ret;
}
```