Sesion 7. Gestion de Procesos – lectura previa

En esta práctica vamos a trabajar con llamadas al sistema relacionadas con la gestión de procesos en Unix. Los objetivos principales son:
- Aprender a crear procesos, finalizarlos y su relación de parentesco.
- Ejercitarse en la ejecución de programas y el paso de argumentos.
Las llamadas al sistema que utilizaremos son: **fork, exec, wait, waitpid, exit, getpid y getppid**.

Las llamadas exec...

El sistema operativo UNIX ofrece una llamada al sistema llamada 'exec' para ejecutar un programa, almacenado en forma de fichero. Aunque en el fondo sólo existe una llamada, las bibliotecas estándar de 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`*`, ... , char* 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", (char *)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", (char *)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, UNIX dispone 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.

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 **`switch`** para determinar quién es el padre y quién el hijo para así ejecutar distinto código.

Con un pequeño ejemplo:

```
int main ()
{
int pid,x;
char buffer[256];

x=1;
switch (pid = fork ()) {
       case -1: esc_error("",SISTEMA,TRUE);
       case 0: /* proceso hijo */
              x++;
              sprintf(buffer,"Soy el hijo %d, mi padre es %d, x=%d\n",
                     getpid(),getppid(),x);
              write (1,buffer,strlen(buffer));
              break;
       default: /* proceso padre */
              x+=5;
              sprintf(buffer,"Soy el padre %d, mi hijo es %d, x=%d\n", getpid(), pid,x);
              write (1,buffer,strlen(buffer));
       }
sprintf (buffer,"Acabo %d\n", getpid()); /* ejecutado por ambos procesos*/
write (1,buffer,strlen(buffer));
exit (0);
}
```

El orden de las frases de salida dependerá del compilador y del planificador de procesos de la máquina donde se ejecute.

Fijémonos en la instrucciones **`getpid()`** y **`getppid()`**. Ambas devuelven un entero, que es un PID. La primera devuelve el PID del proceso que la llama, y la segunda el PID del proceso padre (a no ser que haya muerto, en cuyo caso devuelve un 1).

Contesta: Que frases saldrán por pantalla y cual será el valor de x para el proceso padre y el proceso hijo? Por qué?

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",(char *)0);
4                printf ("Si ves esto, no se pudo ejecutar el \
4b                                                programa\n");
5                exit(1);
6        }
7        else
8        {
9               /* suponemos una declaración int status */
10              wait(&status);
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 ejecutable, 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 nuestro ejemplo no hace falta este valor, porque sólo hay un hijo por el que esperar.

Una cosa interesante es que lo que devuelve el exit no queda directamente en la variable que recoge el wait. De hecho exit solo puede devolver un número de 8 bits (se trunca: exit(256) es lo mismo que exit(0)), por lo que no se puede usar exit para devolver resultados. El número que incluye exit està en el segundo byte del entero que recoje wait, así en un wait(&status), si queremos mirar lo que ha devuelto el exit, debemos hacer: status >> 8.

La llamada **waitpid** es similar a la wait, pero bloquea el proceso hasta que muere un proceso cyuo PID se pasa como parámetro a waitpid. La llamada waitpid tiene 3 parámetros waitpid(pid_t pid, int *status, int options); pid es el PID del proceso que queremos esperar, status es igual que en wait, i respecto a las opciones en esta sesión no las usaremos, así que pondremos un 0 (más información en man waitpid). La llamada devuelve el PID del proceso muerto (igual que wait) o un -1 si hay error.