Sistemas Operativos
Prácticas curso 2021-2022
Programación con llamadas al sistema: Práctica 2

1. ¿Qué llamada al sistema permite crear procesos? Explica el valor retornado por esta llamada al sistema.

   fork() permite crear un proceso a partir de otro (el padre). Retorna 2 veces cada vez que se llama, una en el padre y otra en el proceso nuevo (el hijo). En el padre retorna el PID del hijo ó un -1 si hubo un error, en el hijo retorna siempre un 0.

2. Observa cómo se crean procesos hijo en los dos programas de la transparencia 34. Posteriormente, escribe (y luego entrega) el programa indicado en la transparencia 35 e indica cómo se crean los procesos hijos (de forma análoga a como se indica en la transparencia 34) y explica por qué. ¿Qué pasaría si los procesos hijo comenzaran su ejecución desde el inicio del programa (inicio de la función ***main***)?

   P: El padre (P) entra en el bucle (i=0)
   - P: Se crea un proceso hijo (P1) (i=0)
     - P1: Finaliza iteración (i=1)
     - P1: Se crea un proceso hijo (P11) (i=1)
       - P11: Finaliza la iteración (i=2)
       - P11: Se crea un proceso hijo (P111) (i=2)
         - P111: Finaliza la iteración (i=3)
         - P111: Finaliza el bucle
       - P11: Finaliza la iteración (i=3)
       - P11: Finaliza el bucle
     - P1: Finaliza la iteración (i=2)
     - P1: Se crea un proceso hijo (P12) (i=2)
       - P12: Finaliza la iteración (i=3)
       - P12: Finaliza el bucle
     - P1: Finaliza la iteración (i=3)
     - P1: Finaliza el bucle
   - P: Finaliza la iteración (i=1)
   - P: Se crea un proceso hijo (P2) (i=1)
     - P2: Finaliza la iteración (i=2)
     - P2: Se crea un proceso hijo (P21) (i=2)
       - P21: Finaliza la iteración (i=3)
       - P21: Finaliza el bucle
     - P2: Finaliza la iteración (i=3)
     - P2: Finaliza el bucle
   - P: Finaliza la iteración (i=2)
   - P: Se crea un proceso hijo (P3) (i=2)
     - P3: Finaliza la iteración (i=3)
     - P3: Finaliza el bucle
   - P: Finaliza la iteración (i=3)
   - P: Finaliza el bucle

   (Código C++):
   ```
   #include <iostream>
   #include <sys/types.h>
   #include <unistd.h>
   using namespace std;
   int main()
   {
           for (int i=0; i < 3; i++)
                   fork();
           cout << "Proceso " << getpid()
                   << "; padre = "
                   << getppid() << endl;
   }
   ```

Si todos los hijos comenzasen el código desde el inicio igual que el padre, entrarían en un bucle infinito ya que todos crearían tantos hijos como el padre, y los los procesos creados por los hijos crearían también el mismo número de hijos.

3. Escribe (y posteriormente entrega) el programa indicado en la transparencia 36. ¿Qué variables son compartidas entre el proceso padre (proceso inicial) y sus hijos? Explica cómo están relacionadas las variables en el proceso padre y en sus procesos hijo.

   Las variables del padre se copian al crear al hijo, pero no hay ninguna relación entre ellas desde ese punto en adelante, esto es, si se modifican las de uno, las del otro no se ven afectadas.

   (Código C++):

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
using namespace std;
int glob=6;
int main () {
        int var;
        pid_t pid;
        cout << "Comienzo la ejecución" << endl;
        var = 88;
        if ((pid = fork()) == EXIT_FAILURE) exit(EXIT_FAILURE);
        if (!pid) { glob++; var++; }
        else
                sleep(2);
        cout << "pid = " << getpid() << "glob = " << glob << "var = " << ar <<
endl;
        exit(EXIT_SUCCESS);
}
```

4. ¿Por qué la llamada **exec**() nunca regresa en caso de éxito? ¿Y por qué si lo hace en caso de fallo? Implementa (y posteriormente entrega) un comando ***listado*** que ejecute un "**ls –l**"

   La llamada exec() nunca retorna en caso de éxito porque cambia el programa por completo, elimina todos los datos que existían antes, y con todo ello, su capacidad para retornar.

   En caso de fallo, el programa no es sustituido, por lo que exec() aún mantiene la capacidad de retornar el error.

   (Código C++):

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

int main () {
        if (fork()) {
                execl("/bin/ls", "ls", "-l", NULL);     // El hijo ejecuta 'ls -l'
        } else {
                sleep(1);       // El padre duerme
        }
}
```

5. Implementa (y posteriormente entrega) un comando ***orden*** que ejecute el programa (junto con sus argumentos) indicados por parámetro. Ejemplo: *orden ls –l* → Ejecutaría *ls -l*

```
(Código C++):
#include <iostream>
#include <unistd.h>
#include <sys/ypes.h>
#include <string>

int main (int argc, char* argv[]){
        argv[argc]=NULL;
        execvp(argv[1], argv+1);
}
```

6. En el directorio **/home/asignaturas/so/Shell** encontrarás las dos primeras versiones del código de un intérprete de comandos (Shell) de UNIX, cópialos a tu directorio personal. Compila y ejecuta la Version1 y trata de ejecutar varios comandos UNIX o programas. ¿Qué problema serio tiene esta primera versión? ¿Por qué? (Deberías mirar el código fuente de la función ***main*** para responder a estas preguntas). Vuelve a ejecutar la Version1 e intenta ejecutar varios programas que no existan (por ejemplo *novale*, *noexiste*, etc). ¿Qué diferencia observas? Explica por qué ocurre.

   Esta primera versión cierra el intérprete nada más introducir una orden. El comando que se ejecute, lo hace el proceso padre, por lo que una vez que se termina el comando, el proceso padre se termina y no se pueden introducir más.

   Al ejecutar comandos que no existen se permite continuar usando el intérprete porque no se comienza a ejecutar un comando (porque no es correcto).

7. Observa el código fuente de la Version1 (sobre todo la función ***main***) ¿Qué variante del **exec**() utiliza el intérprete de comandos para ejecutar los programas? ¿Por qué se utiliza esa variante y no otra? ¿Por qué es necesaria la función ***ObtieneArgumentos***?

   Utiliza execvp(), la versión vector, y path. Esto es, buscará en el PATH del proceso el comando que se mande ejecutar. Se utiliza esta forma porque es más fácil pasar los parámetros que el usuario introduzca utilizando un vector.

   Se necesita la función ObtieneArgumentos() para contar el número de argumentos del comando.

8. Compila y ejecuta ahora la Version2 y trata de ejecutar varios comandos UNIX o programas. ¿Qué se mejora respecto a la Version1? ¿Qué solución se ha adoptado en el código fuente para realizar tal mejora? ¿Cuántos procesos hijo se crean cada vez que se intenta ejecutar un programa en el intérprete? ¿Qué proceso ejecuta dicho programa? ¿En qué momento finalizan los procesos hijo? ¿Y el proceso padre?

   Ahora se pueden ejecutar varios comandos correctos de forma consecutiva.
   Para ello se crea un proceso hijo que se encarga de ejecutar la orden que introduce el usuario.
   Se crea un proceso por cada comando introducido.
   El proceso hijo finaliza en cuanto finaliza el comando introducido.