# *Sistemas Operativos I*
# *Manual de prácticas*

## Grupo de Sistemas Operativos (DSIC/DISCA)

## Práctica 3: Procesos POSIX

# *ANTES DE EMPEZAR...*

...la práctica, copie los ficheros fuente que se mencionan en este enunciado en un directorio suyo. Estos ficheros se encuentran en el directorio `/practicas/asignaturas/so1/pr3/`. Por ejemplo:

```
$ cd
$ mkdir practica3
$ cd practica3
$ cp /practicas/asignaturas/so1/pr3/* .
```

Ahora ya está preparado para comenzar la práctica.

# *PRÁCTICA 3: PROCESOS POSIX*

## CREACIÓN DE PROCESOS MEDIANTE FORK

En Unix, un proceso es creado mediante la llamada del sistema *fork*. El proceso que realiza la llamada se denomina proceso padre (*parent process*) y el proceso creado a partir de la llamada se denomina proceso hijo (*child process*). La sintaxis de la llamada efectuada desde el proceso padre es:

```
valor=fork()
```

La llamada *fork* pero devuelve un valor distinto a los procesos padre e hijo: al proceso padre se le devuelve el PID del proceso hijo, y al proceso hijo se le devuelve el valor cero.. Las acciones implicadas por la petición de un *fork* son realizadas por el núcleo (*kernel*) del S.O. Unix. Tales acciones son las siguientes:

1. asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. asignación de un identificador único (PID) al proceso hijo.
3. copia de la imagen del proceso padre (con excepción de la memoria compartida).
4. asignación al proceso hijo del estado "preparado para ejecución".
5. dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega el valor cero.

El siguiente código, ubicado en el fichero forkprog.c, muestra las acciones internas de la llamada fork:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t rf;

    rf = fork();
    switch (rf)
    {
        case –1:
            printf ("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf ("Soy el hijo, mi PID es %d y mi PPID es
                     %d \n", getpid(), getppid());
            sleep (20);
            break;
        default:
        printf ("Soy el padre, mi PID es %d y el PID de
                mi hijo es %d \n", getpid(), rf);
            sleep (30);
    }
    printf ("Final de ejecución de %d \n", getpid());
    exit (0);
}
```

## TRABAJO PRÁCTICO

1) Compile el programa anterior mediante el mandato gcc.

```
$ gcc –Wall -o forkprog forkprog.c
```

2) Ejecute el programa *forkprog* en segundo plano (o *background*). Para ello, se debe añadir al nombre del programa el carácter & (*ampersand*).

```
$ forkprog &
```

3) Verifique su ejecución con la orden *ps* (explicada en la práctica nº 2) que le permitirá comprobar su funcionamiento. Dicha orden ha de ser ejecutada antes de que finalice la ejecución del proceso

```
$ ps -l
```

Observe los valores PID y PPID de cada proceso e identifique qué atributos son heredados entre padre e hijo y cuáles no.

## CUESTIONES:

1.a) Anote el valor mostrado por el shell inmediatamente después de lanzar al proceso en segundo plano e indique qué representa dicho valor.

1.b) ¿Cuáles son los PID de los procesos padre e hijo?

1.c) ¿Qué tamaño de memoria ocupan los procesos padre e hijo ?

1.d) ¿Qué realiza la función *sleep*? ¿Qué proceso concluye antes su ejecución?

1.e) ¿Qué ocurre cuando la llamada al sistema *fork* devuelve un valor negativo?

1.f) ¿Cuál es la primera instrucción que ejecuta el proceso hijo?

1.g) Modifique el código del programa para asegurar que el proceso padre imprime su mensaje de presentación (“Soy el proceso...”) antes que el hijo imprima el suyo.

1.h) Modifique el código fuente del programa declarando una variable entera llamada *varfork* e inicializándola a 10. Dicha variable deberá incrementarse 10 veces en el padre y de 10 en 10. Mientras que el hijo la incrementará 10 veces de 1 en 1. Anote el valor final de la variable *varfork* para el padre y para el hijo.

```
Proceso padre varfork=

Proceso hijo varfork=
```

# INICIACIÓN DE PROGRAMAS MEDIANTE EXEC

La llamada *exec* produce la sustitución del programa invocador por el nuevo programa invocado. Mientras *fork* crea nuevos procesos, *exec* sustituye la imagen de memoria del proceso por otra nueva (sustituye todos los elementos del proceso: código del programa, datos, pila, montículo).

El PID del proceso es el mismo que antes de realizar la llamada *exec*, pero ahora, ejecuta otro programa. El proceso pasa a ejecutar el nuevo programa desde el inicio y la imagen de memoria del antiguo programa se pierde al verse sobrescrita. La imagen de memoria del antiguo programa se pierde para siempre, es decir, todo el código que escribamos posteriormente a la ejecución con éxito de la llamada exec, será inalcanzable.

La combinación de las llamas *fork* y *exec* es el mecanismo que ofrece UNIX para crear un nuevo proceso (*fork*) que ejecute un programa determinado (*exec*).

De las seis posibles llamadas tipo *exec* se usará en este apartado de la práctica la llamada *execv*, cuya sintaxis es:

```
int execv (const char *filename, char *const argv[ ]);
```

El seguimiento de ejecución del proceso, mediante la orden *ps* del sistema usada antes y después de la llamada *execv*, permite comprobar cómo se efectúa la sustitución de la imagen de memoria. El nuevo programa activado mantiene el mismo PID, identificador de proceso, así como otras propiedades asociadas al proceso, sin embargo, el tamaño de memoria de la imagen del proceso cambia dado que el programa en ejecución es diferente.

En este apartado se va a practicar con dos programas (prog1 y prog2), cuyos ficheros fuentes (prog1.c y prog2.c) se muestran a continuación:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

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

     printf ("\nEjecutando el programa invocador (prog1).
              Sus argumentos son: \n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);

     sleep( 10 );
     strcpy (argv[0],"prog2");
     if (execv ("./prog2", argv) < 0) {
          printf ("Error en la invocacion a prog2 \n");
          exit (1);
          };

     exit(0);
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
     int i;
     char a[2000000];

     printf ("Ejecutando el programa invocado (prog2).
              Sus argumentos son: \n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);
     sleep(10);
     exit (0);
}
```

## TRABAJO PRÁCTICO

1) Compile los programas prog1.c y prog2.c mediante el mandato gcc.

```
$ gcc –Wall -o prog1 prog1.c
$ gcc –Wall -o prog2 prog2.c
```

**(Nota**: En el caso de prog2.c, la compilación ocasiona un aviso (*warning*), que no es significativo y puede ignorarse).

Ejecute el programa en *background* (añadiendo '&') para poder verificar su ejecución con la orden ps. La forma de invocar al programa prog1 es la siguiente:

```
$ prog1 arg1 arg2 ... argN &
$ ps -l
```

Para observar cuánta memoria ocupa cada programa, realice un "*ps –l*" una vez el proceso ha escrito en pantalla el mensaje correspondiente (hay 10 segundos de plazo antes de realizar el *execv* en **prog1** y antes de terminar en **prog2**).

## CUESTIONES:

2.a) Escriba el contenido de los elementos del vector *argv* que recibe prog1 y los que recibe prog2.

2.b) ¿Qué PID tiene el proceso que ejecuta *prog1.c*? ¿Y el de *prog2.c*?

2.c) ¿Qué tamaño de memoria ocupa el proceso, según ejecute *prog1* o *prog2*?

2.d) Modifique el programa *prog1.c* para introducir código inalcanzable (p.e. *printf("Hola\n");*) y compruebe que efectivamente no se alcanza. ¿La última línea de prog1.c (la que dice *exit(0);* ), puede llegar a ejecutarse alguna vez?

# VISIBILIDAD DE RECURSOS ENTRE PROCESOS

## ZONAS DE DATOS ENTRE PROCESOS RELACIONADOS POR UNA LLAMADA FORK

Dos procesos vinculados por una llamada *fork* (padre e hijo) poseen zonas de datos propias, de uso privado (no compartidas). Obviamente, al tratarse de procesos diferentes, cada uno posee un espacio de direccionamiento independiente e inviolable. La ejecución del siguiente programa, en el cual se asigna distinto valor a una misma variable según se trate de la ejecución del proceso padre o del hijo, permitirá comprobar tal característica de la llamada *fork*.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ( ) {
     int i;
     int j;
     pid_t rf;

     rf = fork( );
     switch (rf) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          i = 0;
          printf ("\nSoy el hijo, mi PID es %d y mi variable i
                    (inicialmente a %d)  es par", getpid( ), i);
          for ( j = 0; j < 5; j ++ ) {
               i ++;
               i ++;
               printf ("\nSoy el hijo, mi variable i  es %d", i);
               };
          break;
     default:
          i = 1;
          printf ("\nSoy el padre, mi PID es %d y mi variable i
                    (inicialmente a %d) es impar", getpid( ), i);
          for ( j = 0; j < 5; j ++ ) {
               i ++;
               i ++;
               printf ("\nSoy el padre, mi variable i  es %d", i);
               };
     };
     printf ("\nFinal de ejecucion de %d \n", getpid());
     exit (0);
}
```

En el programa anterior (cuyo código fuente es *forkprog2.c*), el proceso padre visualiza los sucesivos valores impares que toma su variable *i* privada, mientras el proceso hijo visualiza los sucesivos valores pares que toma su variable *i* privada y diferente a la del proceso padre.

Para ejecutar este programa, es preciso compilarlo y usar su nombre, *forkprog2*:

```
$ gcc –Wall –o forkprog2 forkprog2.c
$ forkprog2
```

**CUESTIONES:**

3.a) ¿Las variables enteras `i` y `j` del proceso padre son las mismas que las del proceso hijo?

3.b) ¿Qué cambios deberían realizarse en el código para que ambos procesos partieran de igual valor de la variable entera `i`, y además un proceso realizase su cuenta de uno en uno y el otro proceso de dos en dos? Editar el programa, compilarlo y comprobar su ejecución.

## FICHEROS ABIERTOS ENTRE PROCESOS RELACIONADOS POR UNA LLAMADA FORK

Las variables descriptores, asociadas a ficheros abiertos en el momento de la llamada *fork*, son compartidas por los dos procesos existentes a la vuelta del *fork*. Es decir, los descriptores de ficheros en uso por el proceso padre son heredados por el proceso hijo generado.

El siguiente programa, llamado *forkprog3.c*, en el cual los dos procesos escriben distintas cadenas de caracteres en los mismos ficheros, permitirá comprobar tal característica de la llamada *fork*.

Como aclaración previa del siguiente programa, debería notarse los siguiente: ambos procesos (padre e hijo) realizan escrituras (llamadas "*write*") sobre los ficheros. Estas operaciones de Entrada/Salida deberían suspenderlos y, sin embargo, ambos realizan después una llamada a "*usleep*" precisamente para forzar su suspensión. La necesidad de utilizar *usleep* proviene de cómo Linux implementa la llamada *write*: en vez de realizar inmediatamente una escritura en disco (lo que sí produciría la suspensión del proceso), *write* escribe los datos en una memoria intermedia (caché), que posteriormente será volcada a disco. Por ello, el proceso no es suspendido (necesariamente) al utilizar *write*.

```
#include <unistd.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main ( ) {
     int i;
     int fd1, fd2;
     const char string1[10]= "*********";
     const char string2[10]= "---------";
     pid_t rf;

     fd1 = creat ("ficheroA", 0666);
     fd2 = creat ("ficheroB", 0666);
     rf = fork ( );
     switch (rf) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          for ( i = 0; i < 10; i ++ ) {
               write (fd1, string2, sizeof(string2));
               write (fd2, string2, sizeof(string2));
        usleep(1); /* Abandonamos voluntariamente el procesador */
          };
          break;
      default:
          for ( i = 0; i < 10; i ++ ) {
               write (fd1, string1, sizeof(string1));
               write (fd2, string1, sizeof(string1));
        usleep(1); /* Abandonamos voluntariamente el procesador */
          };
     }
     printf ("\nFinal de ejecucion de %d \n", getpid( ));
     exit (0);
}
```

Para ejecutar este programa, no es preciso más que compilarlo y usar su nombre, *forkprog3*. Después verificar los resultados del mismo usando el mandato *cat*:

```
$ gcc –Wall -o forkprog3 forkprog3.c
$ forkprog3
$ cat ficheroA
$ cat ficheroB
```

**CUESTIONES:**

4.a) La expresión:

```
fd1 = creat ("ficheroA", 0666)
```

crea un fichero, le da nombre, le asigna permisos, y asigna una variable entera, fd1, que es el "file descriptor" utilizado por el programa para manipular el fichero.

¿Qué significado tiene el empleo de la constante 0666? ¿Qué permisos tienen los dos ficheros, ficheroA y ficheroB, tras la ejecución del programa? ¿Cuál es la explicación de tal diferencia? (Sugerencia: consultar el mandato *umask*, en la práctica 1).

4.b) La ejecución concurrente de las escrituras de los procesos padre e hijo da lugar a que las cadenas "`*********`" y "`---------`" aparezcan alternadas en los ficheros resultantes.
Consiga, mediante la utilización de la función *sleep*, que la frecuencia a la que el proceso hijo escribe en los ficheros sea menor que la frecuencia de escritura del proceso padre (es decir, que realice menos escrituras por unidad de tiempo). ¿En qué afecta eso al contenido de los ficheros?

# ESPERA DEL PROCESO PADRE AL PROCESO HIJO

## EJECUCIÓN CON ESPERA

En el siguiente programa, llamado *espe1.c*, se emplea la llamada al sistema *wait*. Esta llamada provoca que el proceso invocador quede en suspenso hasta que concluya algún proceso hijo que haya sido activado por él mismo.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <wait.h>

int main ( ) {
     pid_t rf;

     rf = fork( );
     switch (rf) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          printf ("Soy el hijo, mi PID es %d y mi PPID es %d\n",
                   getpid( ), getppid( ));
          sleep (10);
          break;
     default:
          printf ("Soy el padre, mi PID es %d y el PID de mi hijo
                   es %d\n", getpid( ), rf);
          wait (0);
     }
     printf ("\nFinal de ejecucion de %d \n", getpid( ));
     exit (0);
}
```

Para compilar y ejecutar este programa, realizando el seguimiento de su ejecución:

```
$ gcc –Wall -o espe1 espe1.c
$ espe1&
$ ps -l
```

**CUESTIONES:**

5.a) Modifique el código del programa *espe1.c* para que el proceso padre imprima el mensaje de finalización de su ejecución 10 segundos más tarde que el proceso hijo.

## EJECUCIÓN SIN ESPERA

En el siguiente programa, llamado *espe2.c*, no se emplea la llamada al sistema *wait* (el programa es idéntico al anterior *espe1.c* salvo en este detalle).

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main ( ) {
     pid_t rf;

     rf = fork( );
     switch (rf) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          printf ("\nSoy el hijo, mi PID es %d y mi PPID es %d",
                   getpid( ), getppid( ));
          sleep (10);
          break;
     default:
          printf ("\nSoy el padre, mi PID es %d y el PID de mi
                   hijo es %d", getpid( ), rf);
}
     printf ("\nFinal de ejecucion de %d \n", getpid( ));
     exit (0);
}
```

Para compilar y ejecutar este programa, realizando el seguimiento de su ejecución:

```
$ gcc –Wall -o espe2 espe2.c
$ espe2&
$ ps -l
```

## CUESTIONES:

5.b) ¿Cuál es el PPID del proceso hijo, una vez el padre ha finalizado? ¿Por qué?