PRÁCTICA 1: PROCESOS Y COMUNICACIÓN ENTRE PROCESOS

v1.3.0

Rodrigo García Carmona

Antes de empezar…

…la práctica, copia los ficheros que se encuentran en la carpeta “SOS-Practica_1-Materiales” al directorio en el que quieras trabajar. Ahora ya estás preparado para comenzar la práctica.

Objetivos de la práctica

Esta práctica tiene como objetivos entender la forma en que Unix maneja los procesos, así como practicar su uso. Esto incluye la creación de los mismos, la ejecución de programas, la comunicación entre procesos, y la forma en que procesos relacionados comparten información (especialmente memoria y ficheros) o pueden interaccionar entre ellos.

Por consiguiente, se estudiará el uso de las llamadas a sistema *fork*, *exec*, *read*, *write*, *exit*, *wait*, *signal* y *kill* mediante una serie de programas de ejemplo.

1. Creación de procesos mediante *fork*

En Unix, un proceso es creado mediante la llamada al 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()* devuelve un valor distinto para los procesos padre e hijo: al proceso padre se le devuelve el PID del proceso hijo, mientras que al proceso hijo se le devuelve “0”.

Las acciones que implican ejecutar un *fork()* son llevadas a cabo por el núcleo (kernel) del sistema operativo Unix. Dichas 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 al proceso hijo (con excepción de la memoria compartida).
4. Asignación al proceso hijo del estado “preparado para ejecución”.
5. Devolución de los 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 “0”.

El siguiente código, ubicado en el fichero “forkprog1.c”, muestra el uso de la llamada a sistema *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); //suspende el proceso 20 segundos
            break;
        default:
            printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n", getpid(), rf);
            sleep(30); //suspende el proceso 30 segundos. Acaba antes el hijo.
    }
    printf("Final de ejecución de %d \n", getpid());
    exit(0);
}
```

Compila el programa anterior usando “gcc”.

```
$ gcc -Wall -o forkprog1 forkprog1.c
```

Ejecuta el programa “forkprog1” en segundo plano (o *background*). Para ello debes añadir al nombre del programa el carácter “&” (*ampersand*).

```
$ ./forkprog1 &
```

Verifica que se está ejecutando a través del comando “ps” (tal y cómo se explicó en la práctica anterior). Dicha orden ha de ser ejecutada antes de que finalice la ejecución del proceso.

```
$ ps -l
```

Observa los valores del PID y el PPID de cada proceso e identifica qué atributos del procesos son heredados de padre a hijo y cuáles no.

Ejercicios

1. Anota el valor mostrado inmediatamente después de lanzar el proceso en segundo plano e indica qué representa dicho valor.
2. ¿Cuáles son los PID de los procesos padre e hijo?
3. ¿Qué tamaño en memoria ocupan los procesos padre e hijo?
4. ¿Qué realiza la función “sleep”? ¿Qué proceso concluye antes su ejecución?

5. ¿Qué ocurre cuando la llamada al sistema *fork* devuelve un valor negativo?
6. ¿Cuál es la primera instrucción que ejecuta el proceso hijo?
7. Modifica el código del programa para asegurar que el proceso padre imprime su mensaje de presentación (“Soy el padre…”) antes que el hijo imprima el suyo. Guarda el nuevo programa como el archivo “ejercicio1-7.c”.
8. Modifica el código fuente del programa declarando una variable entera llamada “varfork” e inicializándola a 10. En el padre, dicha variable deberá incrementarse 10 veces y de 10 en 10. En el hijo, “varfork” se incrementará 10 veces y de 1 en 1. Además, el programa deberá imprimir por pantalla el valor final de la variable “varfork” tanto para el padre como para el hijo. Guarda el nuevo programa como el archivo “ejercicio1-8.c”.

2. Inicio de programas mediante *exec*

La llamada *exec()* sustituye el programa que la invoca por un nuevo programa. Mientras que *fork()* crea nuevos procesos, *exec()* sustituye la imagen en memoria del proceso por otra nueva (es decir, que sustituye todos los elementos del proceso: código del programa, datos, pila y heap).

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 en memoria del antiguo programa se pierde al verse sobrescrita. La imagen en 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 llamadas *fork()* y *exec()* es el mecanismo que ofrece Unix para la creación de un nuevo proceso (*fork()*) que ejecute un programa determinado (*exec()*).

De las seis posibles llamadas tipo *exec()* usaremos para este apartado de la práctica la llamada “execv()”, cuya sintaxis es:

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

Podemos seguir la ejecución del proceso gracias a la orden “ps”, usándola justo antes y justo después de la llamada “execv()”, comprobando así cómo se efectúa la sustitución de la imagen de memoria. El nuevo programa activado mantiene el mismo PID, así como otras propiedades asociadas al proceso. Sin embargo, el tamaño en 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 (“execprog1” y “execprog2”), cuyos ficheros fuente (“execprog1.c” y “execprog2.c”) se muestran a continuación.

execprog1.c:

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

int main (int argc, char *argv[]) {
    int i;
    printf("Ejecutando el programa invocador (execprog1). Sus argumentos son:\n");
    for (i = 0; i < argc; i++)
```

```
        printf("argv[%d]: %s\n", i, argv[i]);
    sleep(10);
    strcpy(argv[0], "execprog2");
    if (execv ("./execprog2", argv) < 0) {
        printf("Error en la invocacion a execprog2\n");
        exit(1);
    };


    exit(0);
}
```

execprog2.c:

```
#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 (execprog2). Sus argumentos son:\n");
    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    sleep(10);
    exit(0);
}
```

Compila los programas “execprog1.c” y “execprog2.c” usando “gcc”.

```
$ gcc -Wall -o execprog1 execprog1.c

$ gcc -Wall -o execprog2 execprog2.c
```

Ejecuta en *background* (añadiendo “&”) el programa “execprog1”, para poder verificar su ejecución usando la orden “ps”. La forma de invocar el programa “execprog1” es la siguiente:

```
$ ./execprog1 arg1 arg2 ... argN &

$ ps -l
```

Para observar cuánta memoria ocupa cada programa, realiza un “ps -l” una vez el proceso haya escrito por pantalla el mensaje correspondiente (hay 10 segundos de plazo antes de realizar el “execv()” en “execprog1” y otros 10 antes de que “execprog2” termine).

Ejercicios

1. Escribe el contenido de los elementos del vector “argv” que recibe “execprog1” y los que recibe “execprog2”.

2. ¿Qué PID tiene el proceso que ejecuta “execprog1.c”? ¿Y el que ejecuta “execprog2.c”?
3. ¿Qué tamaño de memoria ocupa el proceso, según ejecute “execprog1” o “execprog2”?
4. Modifica el programa “execprog1.c” para introducir código inalcanzable (printf(“Hola\n”);) y comprueba que, efectivamente no se alcanza. Guarda el nuevo programa como el archivo “ejercicio2-4.c”.
5. ¿Puede la última línea de “execprog1.c” (exit(0);) llegar a ejecutarse alguna vez?

3. Zonas de datos entre procesos vinculados por *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 direcciones 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, nos permitirá comprobar dicha característica de la llamada *fork()*.

forkprog2.c:

```
#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()
            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", getpi
            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 el proceso padre va dando valores impares a su variable “i” privada, mientras que el proceso hijo va dando valores pares a su variable “i”, también privada, que es diferente a la del proceso padre.

Para ejecutar este programa es preciso compilarlo primero:

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

$ ./forkprog2
```

Ejercicios

1. ¿Son las variables enteras “i” y “j” del proceso padre las mismas que las del proceso hijo?
2. Cambia el código de “forkprog2.c” para que ambos procesos partan de una variable “i” con igual valor, pero uno la incremente de uno en uno y el otro de dos en dos. Guarda el nuevo programa como el archivo “ejercicio3-2.c”.

4. Ficheros abiertos entre procesos vinculados por *fork*

Los descriptores de ficheros (que indican qué ficheros tiene abierto un proceso) en el momento de hacer la llamada *fork()* son compartidos por los dos procesos que resultan de dicha llamada. Dicho de otra forma, los descriptores de ficheros en uso por el proceso padre son heredados por el proceso hijo generado.

En el siguiente programa, llamado “forkprog3.c”, dos procesos escriben distintas cadenas de caracteres en los mismos ficheros.

Pero antes de meternos con el programa en sí es necesario aclarar un aspecto muy importante sobre el mismo. Ambos procesos (padre e hijo) realizan escrituras (llamadas al sistema *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 obligatoriamente suspendido al utilizar *write()*.

forkprog3.c:

```
#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);
}
```

Ejecuta el programa y observa los resultados obtenidos usando “cat”:

```
$ gcc -Wall -o forkprog3 forkprog3.c

$ ./forkprog3

$ cat ficheroA

$ cat ficheroB
```

Ejercicios

1. La expresión “fd1 = creat(“ficheroA”, 0666)” crea un fichero, le da nombre, le asigna permisos, y lo asigna una variable entera, “fd1”, que es el descriptor de dicho fichero, utilizado por el programa para manipularlo. ¿Qué significado tiene la constante “0666”? ¿Qué permisos tienen los dos ficheros, “ficheroA” y “ficheroB”, tras la ejecución del “forkprog3.c”? ¿Por qué sucede esto?
2. 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. Modifica “forkprog3.c” para

que, mediante la utilización de la función “sleep()”, la frecuencia a la que el proceso hijo escribe en los ficheros sea menor que la del proceso padre. Es decir, que realice menos escrituras por unidad de tiempo. ¿En qué afecta eso al contenido de los ficheros? Guarda el nuevo programa como el archivo “ejercicio4-2.c”.

5. 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 hace que el proceso que la invoca quede suspendido hasta que termine alguno de sus procesos hijos.

espe1.c:

```
#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);
}
```

Compila y ejecuta el programa:

```
$ gcc -Wall -o espe1 espe1.c

$ ./espe1 &

$ ps -l
```

Ejecución sin espera

En el siguiente programa, llamado “espe2.c”, no se emplea la llamada al sistema *wait()*, pero por lo demás es idéntico al anterior.

espe2.c:

```
#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);
}
```

Compila y ejecuta este programa:

```
$ gcc -Wall -o espe2 espe2.c

$ espe2 &

$ ps -l
```

Ejercicios

1. Modifica 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. Guarda el nuevo programa como el archivo “ejercicio5-1.c”.
2. Al ejecutar “espe2.c”, ¿cuál es el PPID del proceso hijo, una vez el padre ha finalizado? ¿Por qué?

3. Manejo de señales

El programa “signal.c” hace uso de la llamada a sistema *signal()* para cambiar su comportamiento con respecto a las señales recibidas.

signal.c:

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#define VUELTAS 10000000000LL

void confirmar(int sig) {
    char resp[100];
    write(1, "Quiere terminar? (s/n):", 24);
    read(0, resp, 100);
    if (resp[0]=='s') exit(0);
}

int main(void) {
    long long int i;
    signal(SIGINT, SIG_IGN);
    write(1, "No hago caso a CONTROL-C\n", 25);
    for (i=0; i<VUELTAS; i++);
    signal(SIGINT, SIG_DFL);
    write(1, "Ya hago caso a CONTROL-C\n", 25);
    for (i=0; i<VUELTAS; i++);
    signal(SIGINT, confirmar);
    write(1, "Ahora lo que digas\n", 19);
    for (i=0; i<VUELTAS; i++);
    exit(0);
}
```

Compila el programa y ejecútalo, intentando abortarlo en varios momentos usando la combinación de teclas CTRL-C. Si los tiempos de los bucles son demasiado cortos o demasiado largos para la máquina en la que estás ejecutando el programa, ajusta el contador de los bucles.

Vuélvelo a ejecutar, pero esta vez mándale la señal de interrupción desde otro terminal, en lugar de usando CTRL-C, para así comprobar que el comportamiento es equivalente. Para ello puedes usar el comando “kill”:

```
kill -SIGINT proceso
```

Ejercicios

1. ¿Para qué sirve cada uno de los parámetros de la función “signal()”?
2. Modifica el programa para que si se le manda la señal de terminar (SIGTERM) escriba el mensaje “No quiero terminar” y continúe ejecutándose. Guarda el nuevo programa como el archivo “ejercicio6-2.c”.
3. Ejecuta el programa modificado y envíale la señal SIGTERM. ¿Qué comando has utilizado para hacerlo?

4. Envío de señales entre procesos

Los procesos se pueden enviar señales unos a otros mediante la llamada al sistema *kill()*, observa a continuación el programa “sacrificio.c”, que hace uso de esta llamada a sistema.

sacrificio.c:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#define VUELTAS 10000000000LL

void hijo(int sig) {
    printf("\t\t¡Padre! ¿Qué haces?\n");
    printf("\t\tFinal de ejecución de %d \n", getpid());
    kill(getppid(), SIGUSR1);
    exit(0);
}

void padre(int sig) {
    printf("\t¡Hijo! ¿Qué he hecho?\n");
    printf("\tFinal de ejecución de %d \n", getpid());
    exit(0);
}

int main(void) {
    long long int i;
    pid_t rf;
    rf = fork();
    switch (rf) {
        case -1:
            printf("No he podido crear el proceso hijo. \n");
            break;
        case 0:
            printf("\t\tSoy Isaac, mi PID es %d y mi PPID es %d. \n", getpid(), getppid());
            signal(SIGUSR1, hijo);
            for (i=0; i<VUELTAS; i++);
            break;
        default:
            printf("\tSoy Abraham, mi PID es %d y el PID de mi hijo es %d. \n", getpid(), rf);
            signal(SIGUSR1, padre);
            sleep(1);    //suspende el proceso 1 segundo.
            printf("\tVoy a matar a mi hijo.\n");
            sleep(15);    //suspende el proceso 15 segundos.
            kill(rf, SIGUSR1);
            for (i=0; i<VUELTAS; i++);
    }
    exit(0);
}
```

Aunque puede enviarse cualquier señal usando *kill()*, hay dos señales de propósito general sin un uso predeterminado disponibles: SIGUSR1 y SIGUSR2. El programa mostrado hace uso de la primera de ellas.

Compila y ejecuta el programa, observando qué ocurre.

Ejercicios

1. ¿Cuántas señales se han enviado?, ¿de qué proceso a qué proceso?, ¿en qué orden?
2. Modifica el programa “sacrificio.c” para permitir la intervención divina. Esta será en forma de la señal SIGUSR2. Al recibir esta señal, el proceso padre ejecutará una función nueva llamada “dios”, que deberá imprimir por pantalla la frase “\t¡El Señor ha intervenido!, ¡se ha interrumpido el sacrificio!\n”, y después terminar la ejecución del proceso mediante “exit(0)”. Guarda el nuevo programa como el archivo “ejercicio7-2.c”.
3. Compila y ejecute el programa modificado. Ahora debes actuar como ser divino enviando la señal SIGUSR2 al proceso padre usando el terminal durante los 15 segundos de margen que tienes antes de que se produzca el sacrificio. Si todo ha salido bien, el proceso hijo no debería ser sacrificado. ¿Qué comando has utilizado?

4. Comunicación mediante tuberías (*pipes*)

Las tuberías o *pipes* sirven para comunicar dos procesos, de tal forma que uno de ellos escribe en una tubería, mientras que el otro lee de ella. En Unix, además de poder conectar dos procesos mediante una tubería usando el carácter “|”, también podemos crear un tipo de tubería conocida como “*pipe* con nombre” de forma explícita. Estas tuberías reciben el apelativo de “con nombre” porque, a diferencia de las creadas usando “|”, su existencia persiste más allá de la ejecución de los procesos que la usan. Por tanto, deben ser borradas de forma explícita.

Como las pipes con nombre se comportan como una cola FIFO, también suelen recibir este nombre.

Crea una nueva *pipe* usando el siguiente comando:

```
mkfifo pruebafifo
```

Las *pipes* son manejadas por el sistema operativo como archivos, algo habitual (como seguro ya sabrás) en Unix. Puedes ver la *pipe* que acabas de crear haciendo un simple listado del directorio actual:

```
ls -al
```

Haz de la *pipe* la entrada estándar de un programa. Como, por ejemplo, de un “sort”:

```
sort < pruebafifo
```

Fíjate en que “sort” por sí mismo no hace nada. Abre otra ventana de terminal y escribe en ella lo siguiente:

```
cat > pruebafifo
```

Ahora teclea unas cuantas líneas de texto y, cuando hayas acabado, pulsa CTRL-D.

Ejercicios

1. Explica lo que ha sucedido.
2. Ahora haz justo lo contrario, ejecuta primero la línea del “cat”, después escribe unas cuantas líneas de texto, terminando con CTRL+D y, por último, ejecuta el “sort”. Explica qué ha sucedido.
3. ¿Qué comando deberías escribir para eliminar la *pipe* que has creado?