



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA  
Escuela Técnica Superior de Ingenieros  
de Telecomunicación

# Prácticas de Laboratorio

*Arquitectura de computadores*  
*ETSI de Telecomunicación*

Himar Alonso Díaz

# Índice

<b>Índice</b>	<b>2</b>
<b>1. Práctica 1</b>	<b>3</b>
<b>2. Práctica 2</b>	<b>3</b>
2.1. Ejercicios de procesos . . . . .	3
2.2. Ejercicios de ficheros . . . . .	8

# 1 Práctica 1

(Introducción al lenguaje C)

# 2 Práctica 2

## 2.1. Ejercicios de procesos

### :: Ejercicio 1 ::

Responda a las siguientes preguntas, confirmando cada respuesta con un pequeño programa que la acredite:

1. Si un proceso padre muere por finalización natural, o bien debido a un `EXIT()`, ¿mueren todos sus hijos?

### :: Solución ::

No. En el siguiente programa el padre muere de forma *natural*. Si lo hace antes que el hijo (esto depende de la distribución del tiempo de CPU, pero lo más probable es que ocurra, dado que el tiempo de ejecución del hijo es mayor) comprobaremos que el hijo no muere al morir el padre.

```
[ practica2/procesos/fork1.c ]
#include <sys/types.h>
#include <stdio.h>

int main() {
    pid_t pid;

    /* Duplicamos el proceso con la llamada fork */
    pid=fork();
    switch(pid) {
    case 0:
        printf("\nEl hijo espera 5 segundos...\n");
        sleep(5);
        printf("\nEl hijo termina la espera de 5 segundos\n");
        break;
    default:
        printf("\nEl padre termina por muerte natural\n");
    }
}
```

2. Si un proceso padre muere porque lo matan desde otro proceso, ¿mueren todos sus hijos?

### :: Solución ::

No. En el siguiente programa tanto el padre como el hijo realizan una espera. Mientras tanto, en otra consola mataremos al padre manualmente con la instrucción `kill`. Comprobaremos que el hijo llega al final de la ejecución, aunque hayamos matado al padre.

```
[ practica2/procesos/fork2.c ]
#include <sys/types.h>
#include <stdio.h>

int main() {
    pid_t pid;

    /* Duplicamos el proceso con la llamada fork */
    pid=fork();
    switch(pid) {
    case 0:
        /* Ejecutamos el proceso hijo, que hara una espera.
           Durante ese tiempo se debe matar al proceso padre
           para comprobar que el hijo no muere al morir el padre.
           Esta comprobacion la haremos imprimiendo por pantalla
           un mensaje antes de terminar el proceso */
        pid=getpid();
        printf("\nEl hijo (pid=%d) espera 15 segundos \n",pid);
        sleep(15);
        printf("\nEl hijo ha terminado (por muerte natural)\n");
        break;
    default:
        /* El padre mostrara su pid y esperara a que el usuario
           lo mate (kill pid)*/
        pid=getpid();
        printf("\nEl pid del padre es%d. Tiene 15 segundos para matarlo
(kill
           %d)\n",pid,pid);
        sleep(15);
    }
}
```

3. Si un proceso hijo muere naturalmente o mediante `EXIT()`, y el proceso padre no está esperando por el, ¿queda el proceso hijo en estado *zombie*?

### :: Solución ::

No. Para comprobarlo, utilizaremos el programa del primer apartado, pero aumentando el tiempo para poder ejecutar el monitor de procesos. En ese programa el padre no espera por el hijo. Cuando el padre haya finalizado (por muerte *natural*) ejecutaremos el monitor de procesos (mediante la orden `ps ax | grep fork`) y veremos que el proceso hijo *no* se encuentra en estado *zombie*. Ésta es la salida de la consola al ejecutar el monitor:

```
$ ps ax | grep fork
6279 pts/2    S          0:00 ./fork1
6287 pts/3    R+        0:00 grep fork
$
```

### :: Ejercicio 2 ::

¿Controla Linux el número de procesos en ejecución? Responda con un programa que acredite su respuesta.

#### :: Solución ::

No. El siguiente programa hace un `fork` dentro de un *bucle infinito*. Al ejecutarlo se cuelga el sistema operativo, ya que llega un momento en el que el sistema se queda sin memoria.

```
[ practica2/procesos/controlproc.c ]
```

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

```
int main() {
    pid_t pid;
    while(1) {
        pid = fork();
    }
}
```

### :: Ejercicio 3 ::

Realice un programa en el que un proceso padre genere dos procesos hijo. Modifique algún dato local y global en cada hijo y compruebe que los valores del dato en el proceso padre y en los hijos son realmente diferentes.

#### :: Solución ::

```
[ practica2/procesos/independencia.c ]
```

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

```
int main() {
    pid_t pid;

    int variable = 1;

    /* Creamos un proceso hijo que tendrá pid=0 */
    pid = fork();
    if (pid == 0) {
        /* Si modificamos la variable, solo quedara modificada
           la del hijo, ya que al hacer fork() se duplican los procesos
           y tambien las variables */
        variable = 2;
        printf("La variable del primer hijo vale:%d\n",variable);
    }
}
```

```

        /* El hijo imprime por pantalla y luego sale */
        exit(0);
    } else {
        /* Comprobamos que la variable del padre no ha cambiado */
        printf("La variable del padre sigue valiendo:%d\n",variable);
    }
    /* Repetimos la operacion con otro hijo y comprobamos que la
    variable queda modificada solo en el proceso hijo */
    pid = fork();
    if (pid == 0) {
        variable = 3;
        printf("La variable del segundo hijo vale:%d\n",variable);
    } else {
        /* Comprobamos que la variable del padre no ha cambiado */
        printf("La variable del padre sigue valiendo:%d\n",variable);
    }
}

```

#### :: Ejercicio 4 ::

Realice un programa que ejecute las siguientes órdenes de forma similar a como lo hace el intérprete de comandos del sistema operativo:

```

$ programa_ejecutable      (ejecución normal)
$ programa_ejecutable &   (ejecución en modo background)

```

#### :: Solución ::

Programa en ejecución normal (*no-background*):

```

[ practica2/procesos/nobackground.c ]
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

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

    if (argc < 2) {
        /* Para duplicar el proceso */
        pid_t pid;
        /* Variable de estado para esperar por el hijo */
        int status;

        /* Comando que debera ejecutarse */
        char comando[10];

        /* Solicitamos el comando a traves del teclado */
        printf("\nEscribe un comando para ejecutar: ");
        scanf("%s", comando);
    }
}

```

```

/* Duplicamos el proceso. El padre esperara a que
   el hijo termine de ejecutar el un programa (comando)
   mediante la sentencia exec */
pid = fork();

switch(pid) {

case 0:
    /* El hijo ejecutara el comando y terminara */
    execlp(comando,comando,NULL);

default :
    /* El padre esperara que el hijo termine, y luego continuara
    ejecutandose */
    if(wait(&status)==-1) {
        printf("\nHa habido un error esperando por el hijo\n");
    } else {
        printf("\nEl padre ha esperado que el hijo terminara de ejecutar:
        %s\n",&comando);
        printf("\nAhora continua la ejecucion del padre.\n");
    }
}

} else {
    /* Ayuda */
    printf("\n Uso: $nobackground\n\n");
}
}

```

Programa que se ejecuta en segundo plano (*background*):

[ practica2/procesos/background.c ]

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

```

```

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

```

```

    if (argc < 2) {
        /* Para duplicar el proceso */
        pid_t pid;

```

```

        /* Comando que debera ejecutarse en background */
        char comando[10];

```

```

        /* Solicitamos el comando a traves del teclado */

```

```

printf("\nEscribe un comando para ejecutar en background: ");
scanf("%s", comando);

/* Duplicamos el proceso. El padre NO esperara por el hijo
   y el hijo hara una llamada exec */
pid = fork();

switch(pid) {

case 0:
    /* El hijo ejecutara el comando y terminara */
    execlp(comando,comando,NULL);

default :
    /* El padre continua ejecutandose. Hacemos un printf para
       comprobarlo: */
    printf("\nEl padre se sigue ejecutando, mientras el hijo ejecuto
el
           comando:%s\n",&comando);
    printf("\nEl padre finaliza sin esperar por el hijo.\n");
}

} else {
    /* Ayuda */
    printf("\n Uso: $background\n\n");
}
}

```

## 2.2. Ejercicios de ficheros

Todos los programas de ficheros incluyen una ayuda sobre su uso. Al ejecutarlos con la opción programa -h o bien programa --help se indica cómo debe ejecutarse el programa.

### :: Ejercicio 1 ::

Realice un programa que ejecute un comando del sistema operativo con la entrada estándar redireccionada hacia un fichero. El equivalente al ejecutar dicho comando en el intérprete de comandos del sistema operativo sería:

```
$ comando < file
```

### :: Solución ::

```
[ practica2/ficheros/entradast.c ]
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

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

```

if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
    /* Ayuda */
    printf("\n Uso: $entrada comando ficheroentrada\n\n");
} else {

    /* Descriptor del fichero de entrada */
    int df;

    /* Cambiamos la entrada estandar */
    df=open(argv[2], O_RDWR);
    close(0);
    dup(df);
    close(df);

    execlp(argv[1], argv[1], 0);
}
}

```

### :: Ejercicio 2 ::

Realice un programa similar al del ejercicio 1, pero utilizando la salida estándar:

\$ comando > file

### :: Solución ::

[ practica2/ficheros/salidast.c ]

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

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

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

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

```

    if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
        /* Ayuda */
        printf("\n Uso: $salida comando ficherosalida\n\n");
    } else {

```

```

    } else {

```

```

        /* Descriptor del fichero de salida */
        int df;

```

```

        /* Creamos el fichero de salida */
        df=open(argv[2], O_CREAT);
        close(df);

```

```

        /* Cambiamos la salida estandar */
        df=open(argv[2], O_RDWR);

```

```

        close(1);
        dup(df);
        close(df);

        execlp(argv[1],argv[1],0);
    }
}

```

### :: Ejercicio 3 ::

Realice un programa similar a los anteriores, pero redireccionando tanto la entrada como la salida estándar:

\$ comando < file\_1 > file\_2

### :: Solución ::

[ practica2/ficheros/entradasalida.c ]

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

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

    if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
        /* Ayuda */
        printf("\n Uso: $entradasalida comando ficheroentrada
                ficherosalida\n\n");
    } else {

        /* Descriptores de los ficheros de entrada y salida */
        int dfent;
        int dfsal;

        /* Creamos el fichero de salida */
        dfsal=open(argv[3],O_CREAT);
        close(dfsal);

        /* Cambiamos la entrada estandar */
        dfent=open(argv[2],O_RDWR);
        close(0);
        dup(dfent);
        close(dfent);

        /* Cambiamos la salida estandar */
        dfsal=open(argv[3],O_RDWR);
        close(1);
        dup(dfsal);
        close(dfsal);
    }
}

```

```

        execlp(argv[1], argv[1], NULL);
    }
}

```

### :: Ejercicio 4 ::

Realice un programa similar al del Ejercicio 3, pero teniendo en cuenta que al final del programa la entrada estándar debe ser nuevamente el teclado, y la salida estándar debe ser el terminal.

### :: Solución ::

[ practica2/ficheros/entradasalidarecu.c ]

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

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

    if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
        /* Ayuda */
        printf("\n Uso: $entradasalidarecu comando ficheroentrada
                ficherosalida\n\n");
    } else {

        /* Descriptores de los ficheros */
        int dfent;
        int dfsal;
        int stin;
        int stout;

        /* Para duplicar el proceso (luego veremos por que) */
        pid_t pid;

        /* Mensaje para comprobar la entrada estandar */
        char *mensaje;

        /* Creamos el fichero de salida */
        dfsal=open(argv[3], O_CREAT);
        close(dfsal);

        /* Antes de cambiar la entrada y salida estandar guardamos los
        descriptores */
        stin=dup(0);
        stout=dup(2);

        /* Cambiamos la entrada estandar */

```

```

dfent=open(argv[2],O_RDWR);
close(0);
dup(dfent);
close(dfent);

/* Cambiamos la salida estandar */
dfsal=open(argv[3],O_RDWR);
close(1);
dup(dfsal);
close(dfsal);

/* Dividimos el proceso. Uno de los dos ejecutara
la instruccion "exec" (y terminara), y el
otro recuperara la entrada y salida estandar */

pid = fork();

switch(pid) {

case 0:
    /* El hijo ejecutara el comando y terminara */
    execlp(argv[1],argv[1],NULL);

default :
    /* El padre recupera la entrada y salida estandar */

    /* Recuperamos la entrada y salida estandar (teclado y pantalla) */
    close(0);
    dup(stin);
    close(1);
    dup(stout);

    /* Probamos que funciona la entrada y salida estandar */
    printf("\nEscribe un texto para probar el teclado: ");
    scanf("%s", &mensaje);
    printf("El mensaje escrito es:%s\n",&mensaje);
}
}
}

```